МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«ЧЕРЕПОВЕЦКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Институт информационных технологий

Е.В. Ершов, О.Г. Ганичева, В.В. Селивановских, Л.Н. Виноградова

ПРОГРАММИРОВАНИЕ. БАЗОВЫЕ СРЕДСТВА ЯЗЫКА ПРОГРАММИРОВАНИЯ С++

Учебное пособие

Череповец
2011

ББК 32.973
УДК 681.3.06
Е 80

Рассмотрено на заседании кафедры ПО ЭВМ, протокол № 2 от 29.09.10 г.
Одобрено УМС ЧГУ, протокол № 4 от 23.12.10 г.

Программирование. Базовые средства языка программирования C++: Учеб. пособие / Е.В. Ершов, О.Г. Ганичева, В.В. Селивановских, Л.Н. Виноградова. – Череповец: ЧГУ, 2011. – 181 с. – ISBN 978 – 5 – 85341 – 467 – 9.

Учебное пособие по дисциплине «Программирование» предназначено для студентов направлений 230100 «Информатика и вычислительная техника», 231000 «Программная инженерия», 230400 «Информационные системы и технологии», 230700 «Прикладная информатика».

Р е ц е н з е н т ы : *Е.В. Королева*, канд. техн. наук (ООО «Фирма «СТО-ИК», г. Череповец); *Ю.В. Веселов*, канд. техн. наук (ОАО «Северсталь-инфоком Софт», г. Череповец)

Н а у ч н ы й р е д а к т о р : Е.В. Королева, канд. техн. наук

ISBN 978 – 5 – 85341 – 467 – 9

Введение

Предметом курса «Программирование» являются общие принципы программирования и проектирования эффективных структур данных и алгоритмов их программной обработки с использованием языка Си, типизации и структуризации данных, методологии проектирования прикладного программного обеспечения, инструментальные средства и методы их использования в процессах сопровождения программного обеспечения.

Результатом изучения этого курса является овладение навыками программирования для решения прикладных задач, составляющих содержание смежных дисциплин «Системное программное обеспечение», «Проектирование компиляторов», «Теория формальных языков и грамматик» и др.

Структурное программирование – это процесс пошагового разбиения алгоритма на более мелкие части с целью получения таких элементов, для которых можно легко написать конкретные команды. *Процедурное программирование* – это процесс создания функций с целью упорядочения алгоритмов и организации программ. *Модульное программирование* – это процесс объединения взаимосвязанных процедур и тех данных, которыми эти процедуры оперируют с целью ограничения доступа к данным. *Объектно-ориентированное программирование* – это процесс определения необходимых типов данных и полного набора операций для каждого типа. *Язык Си* – это структурированный, модульный, компилируемый, универсальный язык, традиционно используемый для системного программирования.

Глава 1

ДИРЕКТИВЫ ПРЕПРОЦЕССОРА, КОММЕНТАРИИ, ИДЕНТИФИКАТОРЫ

1.1. **Препроцессор, компилятор и загрузчик**

Для того, чтобы исходная программа на Си была оттранслирована и переведена в исполняемый машинный код, она должна пройти через три процесса: препроцессирование, компиляцию и загрузку (сборку). В задачу препроцессора входит подключение при необходимости к данной программе на Си внешних файлов, указываемых при помощи директивы #include (см. п. 1.3), и расширение макро (см. п. 1.4).

Компилятор за несколько этапов транслирует то, что вырабатывает препроцессор в объектный файл (файл с расширением .OBJ), содержащий оптимизированный машинный код, при условии, что не встречались синтаксические и семантические ошибки. Если в исходном файле с программой на Си обнаруживаются ошибки, то программисту выдается их список, в котором ошибки привязываются к номеру строки, в которой они появились. Программист циклически выполняет действия по редактированию/компиляции до тех пор, пока не будут устранены все ошибки в исходном файле.

Загрузчик связывает между собой объектный файл, получаемый от компилятора, с программами из требуемых библиотек и, возможно, с другими файлами. В результате сборки получается файл с расширением EXE (EXE-файл), который может быть исполнен компьютером.

1.2. **Комментарии**

Текст на Си, заключенный между символами /* и */, является комментарием и компилятором игнорируется.

Комментарии служат двум целям: документировать код и облегчить отладку. Так, если в программе на Си используется некоторый

специальный алгоритм, то будет уместен комментарий, содержащий ссылку на книгу или статью, в которой описывается этот алгоритм. Вообще говоря, в программу следует вносить любой текст, проясняющий код. Но комментариями не нужно злоупотреблять. Сделать код на Си самодокументированным можно, используя разумные соглашения по именованию переменных, функций и т.д. Си разрешает вложение комментариев. Вложенные комментарии особенно полезны при отладке программы. Если программа работает не так, как надо, то иногда оказывается полезным закомментировать часть кода (т.е. вынести ее в комментарий), заново скомпилировать программу и выполнить ее. Если теперь программа начнет работать правильно, то, значит, закомментированный код содержит ошибку и должен быть исправлен. Если программа не заработает, то отключаются новые фрагменты. Поскольку и сам исходный текст может содержать комментарии, то рассматриваемая возможность Си, позволяющая превращать в комментарий код, уже содержащий комментарий, позволяет быстро удалить фрагмент кода из текста программы.

1.3. **Директива include**

Как правило, в программы на Си подставляются один или несколько файлов, часто в самое начало кода главной программы main. Появление директив

```
#include <файл_1>
#include "файл_2"
...
#include <файл_n>
```

приводит к тому, что препроцессор подставляет на место этих директив тексты файлов, соответственно, файл_1, файл_2,..., файл_n.

Если имя файла заключено в угловые скобки <>, то поиск файла производится в специальном каталоге подстановочных файлов. Обычно в этот каталог помещаются все файлы с расширением .h.

Если в каталоге отсутствует искомый файл, то препроцессор выдает сообщение об ошибке и трехступенчатый процесс, описанный в п 1.1, прерывается.

Если имя файла заключено в двойные кавычки (как, например, "файл_2"), то поиск файла производится сначала в текущем каталоге. Если здесь файл не обнаруживается, то система переходит к поиску файла в каталоге подстановок. Если и здесь требуемый файл не будет найден, то препроцессор выдаст сообщение об ошибке и трехступенчатый процесс будет прерван.

В отличие от многих других операторов Си, директива include не должна оканчиваться точкой с запятой.

1.4. **Директива define**

С помощью директивы #define, вслед за которой пишутся имя макро и значения макро, оказывается возможным указать препроцессору, чтобы он при любом появлении в исходном файле на Си данного имени макро заменял это имя на соответствующее значение макро. Макро могут иметь параметры. Например, макро

```
#define cube(x)((x)*(x)*(x))
```

задает замену символа cube(аргумент) на значение (аргумент)*(аргумент)*(аргумент).

Часто макро используют для того, чтобы увязать идентификатор и значение. Как только препроцессор встречает идентификатор, он заменяет его на соответствующее значение. Например, директива

```
#define pi 3.1415962
```

связывает идентификатор pi со значением 3.1415962.

Не следует оканчивать значение макро (например, 3.1415962) точкой с запятой. Значение макро подставляется вместо имени мак-

ро полностью. Если точка с запятой присутствует, то она будет подставлена вместе с числом.

1. Назовите основные задачи препроцессора. **2.** Назовите основные задачи компилятора. **3.** Назовите основные задачи загрузчика. **4.** Для чего в программе используются комментарии? **5.** Дайте определение транслятора. **6.** Поясните разницу между компилятором и интерпретатором. **7.** К чему приводит появление директивы include? **8.** Для чего служит директива define?

Глава 2

ЛЕКСИЧЕСКИЕ ОСНОВЫ ЯЗЫКА ПРОГРАММИРОВАНИЯ СИ

2.1. **Алфавит языка Си**

Алфавит языка включает прописные и строчные буквы латинского алфавита, арабские цифры 0…9, специальные символы: + – * / = _ ‘‘ . : ; ^ ‘ # ~ [ ] ( ) { } < > : ? ! % \ | .

Из символов алфавита формируют лексемы языка: ключевые слова, идентификаторы, константы, знаки операций, знаки пунктуации.

Идентификаторы – это имена констант, переменных, меток, типов данных, функций и т.д. Идентификаторы должны состоять только из букв, цифр и знака «_», причем начинаться только с буквы или знака «_», например: a2, _W1. Прописные и строчные буквы имеют различные внутренние коды. Например, идентификаторы ind1, InD1 и IND1 различны. Значимыми являются первые 32 символа идентификатора.

Примечание. Си также разрешает использовать знак доллара ($). В Си можно изменить число символов идентификатора с помощью опции компилятора -i#, где # является числом значащих символов. 32 символа являются значащими также и для глобальных идентификаторов, берущихся из других модулей.

Ключевые слова – это идентификаторы, зарезервированные в языке для построения различных конструкций, таких как операторы, операции, имена стандартных функций, например: sin, cos и т.д.

Константы – это элементы данных, значения которых установлены в описательной части программы и в процессе выполнения программы не изменяются. Они объявляются в разделе описаний с использованием зарезервированного слова CONST:

```
Const тип имя константы=значение;
Например, Const int a=25;
```

Переменные, в отличие от констант, могут менять свои значения в процессе выполнения программы. При описании переменных необходимо указать их тип:

```
int x;
double a;
```

Константы и переменные составляют основу лексики языка.

2.2. **Средства ввода-вывода информации**

2.2.1. *Вывод информации*

Может показаться странным, что мы начинаем разговор именно с вывода, однако программ, которые ничего не выводят, почти нет. Под выводом обычно понимают форму информации, записываемой на экран (слова и картинки), на запоминающее устройство (гибкий или жесткий диски) или в порт ввода-вывода (последовательный порт, порт принтера).

Функция printf

Наиболее распространенная функция вывода в Си – подпрограмма printf. Ее целью является запись информации на экран. Ее формат выглядит так:

printf(<строка формата>, <объект>, <объект>, ...);

Строка формата – это строка, которая начинается и заканчивается двойными кавычками ("текст"); цель printf – запись этой строки на экран. Перед выводом printf заменяет все дополнительно перечисленные объекты в строке в соответствии со спецификациями формата, указанными в самой строке. Например, рассмотрим следующий оператор printf:

printf("Сумма = %d \n",sum);

Здесь %d в строке формата – это спецификация формата. Все спецификации формата начинаются с символа процента (%) и (обычно) сопровождаются одной буквой, обозначающей тип данных и способ их преобразования.

Вы должны иметь для каждого объекта только одну, соответствующую ему спецификацию формата. Если объект имеет тип данных, не соответствующий спецификации формата, то Си попытается выполнить нужное преобразование.

Сами объекты могут быть переменными, константами, выражениями, вызовами функций. Короче говоря, они могут быть чем угодно, что дает соответствующее значение спецификации формата.

%d, используемое в спецификации, говорит о том, что ожидается некоторое целое число.

Можно задать ширину поля, помещая ее между % и буквой; например, десятичное поле шириной 4 задается как %4d. Значение будет напечатано сдвинутым вправо (впереди пробелы), так что общая ширина поля равна 4.

Если нужно напечатать знак %, то вставьте %%.

\n в строке не является спецификацией формата, а употребляется (по историческим мотивам) как управляющая (escape) последовательность и представляет специальный символ, вставляемый в строку. В этом случае \n вставляет символ в начале новой строки, поэтому после вывода строки курсор передвинется к началу новой строки.

Список управляющих последовательностей представлен в табл. 2.1.

Если нужно напечатать обратную косую черту, то вставьте \\.

Таблица 2.1

Escape последовательности Си

Последовательность	Значение	Символ	Что делает
\a	0x07	BEL	Звуковой сигнал
\b	0x08	BS	<- ╞ (Забой)
\f	0x0C	FF	Перевод страницы
\n	0x0A	LF	Перевод строки
\r	0x0D	CR	Возврат каретки
\t	0x09	HT	Горизонтальная табуляция
\v	0x0B	VT	Вертикальная табуляция
\\	0x5c	\	Обратный слеш
\'	0x2c	'	Апостроф
\"	0x22	"	Двойная кавычка
\?	0x3F	?	Вопросительный знак
\DDD		любой	DDD = от 1 до 3 восьмеричных цифр
\xHHH	0xHHH	любой	HHH = от 1 до 3 шестнадцатеричных цифр

Более формально спецификацию % можно определить следующим образом:

%[флаг(и)][ширина][.точность][l,L] символ_формата,

где l или L используются для указания длинных типов, а ширина – минимальный размер поля вывода (табл. 2.2); %c – символ (char); %s – строка (char*); %d – целое число (int); %o – целое число в восьмеричном виде (int); %x – целое число в шестнадцатеричном

виде (int); %u – целое число без знака (unsigned int); %ld – длинное целое число в десятичном виде (long); %lo – длинное целое число в восьмеричном виде (long); %lx – длинное целое число в шестнадцатеричном виде (long); %lu – длинное целое число без знака (unsigned long); %f – вещественное число с фиксированной точкой (float/double); %e – вещественное число в экспоненциальной форме (float/double); %g – вещественное число в виде f или e в зависимости от значения (float/double); %lf – длинное вещественное число с фиксированной точкой (long float); %le – длинное вещественное число в экспоненциальной форме (long float); %lg – вещественное число в виде f или e в зависимости от значения (long float); %p – значение указателя.

Таблица 2.2

Спецификация

Значение флага	Назначение
– + Пусто	Выравнивание по левому краю поля Вывести знак значения – как «плюс», так и «минус» Для неотрицательных значений вместо знака «плюс» вывести пробелы
Тип значения	Точность
Целое Вещественное Строка	Число цифр Число цифр после десятичной точки Число символов
Символ формата	Тип выводимого объекта

Ниже приведена программа, демонстрирующая некоторые возможности Си по выводу информации на экран:

```
/* Использование функции printf*/

#include <stdio.h>
#define square(x) ((x)*(x))
#define pi 3.1415926
```

```
main()
{
 float x=2.5;
 int i=11;
 int j=119;

 printf("\nЗначение квадрата 2.5=%10.4f", square(x));
 printf("\nЧисло пи=%10.4f", pi);
 printf("\nЗначение 2*пи=%10.4f", 2.0*pi);
 printf("\n\nШестнадцатеричный вид числа 11=%x", i);
 printf("\n\nШестнадцатеричный вид числа 119=%x", j);
 printf("\n\nВосьмеричный вид числа 119=%o\n", j);
}
```

Функции puts и putchar для вывода

Имеются две другие функции вывода: puts и putchar. Функция puts выводит строку на экран и завершает вывод символом новой строки, например:

```
main ()
{
 puts("Hello, world");
}
```

Заметим, что в конце строки опущен \n; это не нужно, так как puts сама добавляет этот символ.

Наоборот, функция putchar выводит единственный символ на экран и не добавляет \n. Оператор putchar(ch) эквивалентен printf ("%c",ch).

Зачем же нужно использовать puts и/или putchar вместо printf? Одним из доводов является то, что программа, реализующая printf, гораздо больше. Если вы не нуждаетесь в ней (для числового вывода или специального форматирования), то, используя puts и putchar, можно сделать свою программу меньше и быстрее. Например, файл .EXE, использующий puts, значительно меньше, чем файл .EXE, для версии, использующей printf.

2.2.2. *Ввод информации*

В Си имеется несколько функций ввода; некоторые производят ввод из файла или из входного потока, другие – с клавиатуры.

Функция scanf

Для интерактивного режима ввода можно использовать в большинстве случаев функцию scanf. Scanf – это функция ввода, по смыслу эквивалентная printf; ее формат выглядит так:

scanf(<строка формата>,<адрес>,<адрес>,...)

В scanf используются многие из тех же спецификаторов формата %<буква>, что и у функции printf: %d – для целых, %f – для чисел с плавающей точкой, %s – для строк и т.д.

Однако scanf имеет одно очень важное отличие: объекты, следующие за строкой формата, должны быть адресами, а не значениями:

```
scanf("%d %d", &a, &b);
```

Этот вызов сообщает программе, что она должна ожидать ввода двух десятичных (целых) чисел, разделенных пробелом; первое будет присвоено a, а второе b. Заметим, что здесь используется операция определения адреса (&) для передачи адресов a и b функции scanf.

Промежуток между двумя командами формата %d фактически означает больше, чем просто промежуток. Он означает, что вы можете иметь любое количество «белых полей» между значениями. Что такое белое поле? Это любая комбинация пробелов, табуляций и символов новой строки. В большинстве ситуаций компиляторы и программы Си обычно игнорируют «белое поле». Но что же надо делать, если вы хотите разделить числа запятой вместо пробела? Необходимо лишь изменить строку ввода:

```
scanf ("%d, %d", &a, &b);
```

Это позволяет вам ввести значения, разделенные запятой.

Для ввода строки (последовательности символов) можно использовать следующую программу:

```
main ()
{
 char name[30];

 printf("Как Вас зовут: ");
 scanf("%s", name);
 printf ("Привет, %s\n", name);
}
```

Поскольку name является массивом символов, значение name – это адрес самого массива. По этой же причине перед именем name не используется адресный оператор &, вы просто пишете scanf ("%s", name);

Обратите внимание, что мы использовали массив символов (char name [30];) вместо указателя на символ (char *name;). Почему? Причиной этого служит тот факт, что объявление массива на самом деле резервирует память для хранения его элементов, а при объявлении ссылки этого не происходит. Если бы мы использовали объявление char *name, тогда нам бы пришлось явным образом резервировать память для хранения переменной *name.

Функции gets и getch для ввода

Использование scanf, однако, порождает другую проблему. Обратимся к последней программе, но теперь введем имя и фамилию. Заметьте, что программа использует в своем ответе только имя. Почему? Потому что введенный после имени пробел сигнализирует scanf о конце вводимой строки.

Возможны два способа решения этой проблемы. Вот первый из них:

```
main ()
 {
  char first[20],middle[20],last[20];
  printf("Как Вас зовут:");
  scanf("%s %s %s",first,middle,last);
  printf("Дорогой %s, или %s?\n",last,first);
 }
```

Это означает, что имеется три компоненты имени; в примере функция scanf не пойдет дальше, пока вы действительно не введете три строки. Но что, если необходимо прочитать полное имя как одну строку, включая пробелы? Вот второй способ решения:

```
main ()
{
 char name [60];
 printf("Как вас зовут: ");
 gets (name);
 printf ("Привет, %s\n", name);
}
```

Функция gets читает все, что вы набираете, до тех пор, пока не нажмете «Ввод». Она не помещает «Ввод» в строку, однако в конец строки добавляет нулевой символ (\0).

Наконец, есть еще функция getch(). Она читает единственный символ с клавиатуры, не выдавая его на экран (в отличие от scanf и gets). Заметим, что у нее нет параметра ch; getch является функцией типа char, ее значение может быть непосредственно присвоено ch.

Ошибки при передаче по адресу

Рассмотрим следующую программу:

```
main()
{
  int a,b,sum;
  printf("Введите два значения:");
  scanf("%d %d",a,b);
  sum = a + b;
  printf("Сумма значений равна: %d\n",sum);
}
```

Ошибка находится в операторе

```
scanf("%d %d",a,b);
```

Scanf требует передачи адресов, а не значений! То же самое относится к любым функциям, содержащим в качестве формальных параметров указатели. Программа, написанная выше, оттранслируется и выполнится, но при этом scanf возьмет какие-то случайные значения (мусор), находящиеся в a и b, и использует их как адреса, по которым будут записаны введенные вами значения.

Правильно этот оператор необходимо записать так:

```
scanf("%d %d", &a, &b);
```

Здесь функции scanf передаются адреса a и b, и введенные значения правильно запоминаются в этих переменных по их адресам.

1. Для чего служат идентификаторы? **2.** Назовите правила написания идентификаторов пользователя. **3.** Как объявляются и используются константы? **4.** Поясните назначение функции scanf. **5.** Поясните назначение функции printf. **6.** Определите тип данных для следующих констант: -6.789; 260; ‘*’; 3.7E57; 3.8L; ‘6’; 0976. **7.** Какие из представленных записей не являются идентификаторами: asd; _hg; 6yh; kj8n; rt_t; as:sa; 123; _45a; v53?

Глава 3

ОСНОВЫ ТИПИЗАЦИИ ДАННЫХ

3.1. **Понятие типа данных, абстракция данных**

Абстрактные типы данных характеризуются набором допустимых значений и соответствующим набором операций над этими значениями. Для действительно абстрактного типа данных правильность их использования гарантируется тем, что над данными указанного базового типа разрешаются только те операции, которые присутствуют в заданном наборе. Такой подход называют *инкапсуляцией данных*.

Таким образом, тип задается набором допустимых значений и набором действий, которые можно совершать над каждой переменной рассматриваемого типа.

Считается, что переменная или выражение принадлежит к данному типу, если его значение принадлежит области допустимых значений этого типа.

Переменные типизируются посредством их описаний. Выражения типизируются посредством содержащихся в них операций.

В Си имеется множество предопределенных типов данных, включая несколько видов целых, вещественных, указателей, переменных, массивов, функций, объединений, структур и тип void (отсутствие типа). Тип void не имеет ни значений, ни действий.

Целые типы включают несколько разновидностей целых и символьных данных. Арифметические типы объединяют целые и вещественные. Скалярные типы включают арифметические типы, указатели и перечислимые типы. Агрегаты, или структурные типы, включают массивы, структуры и объединения. Функции представляют особый класс.

Типы Си представлены на рис. 3.1.

В каждой реализации Си всякий тип занимает определенное число единиц памяти. За такую единицу принимается память, требуемая для хранения одного символа; и обычно она составляет 1 байт. Число единиц памяти, требуемое для размещения элемента

данного типа, может быть вычислено операцией определения размера sizeof (тип).

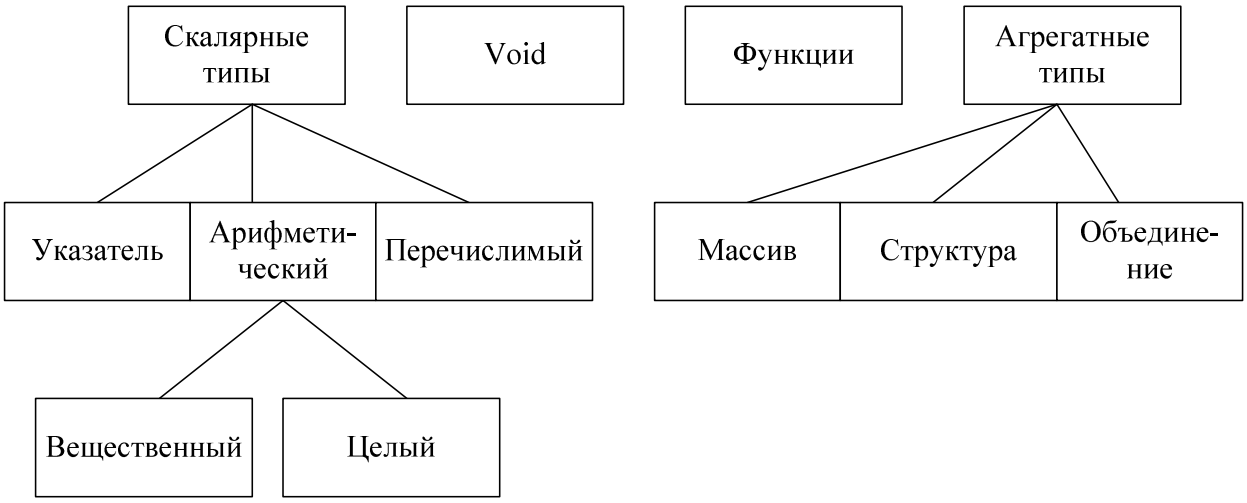

Рис. 3.1. Типы языка Си

В табл. 3.1 представлены типы данных Си, указаны их размеры в байтах и заданы диапазоны значений (наборы значений).

Таблица 3.1

Типы данных в Си, их размеры и диапазоны значений

Тип	Размер (в байтах)	Диапазон
1	2	3
unsigned char	1	0 – 255
char	1	-128 – 127
enum	2	-32 768 – 32 767
unsigned short	2	0 – 65 535
short	2	-32 768 – 32 767
unsigned int	2	0 – 65 535
int	2	-32 768 – 32 767
unsigned long	4	0 – 4 294 967 295
long	4	-2 147 483 648 – 2 147 483 647
float	4	3.4E-38 – 3.4E+38

Окончание

1	2	3
double	8	1.7E-308 – 1.7E+308
long double	8	1.7E-308 – 1.7E+308
pointer	2	(указатели near,_cs,_ds,_es,_ss)
pointer	4	(указатели far, huge)

П р и м е ч а н и е : тип long double допускается, но трактуется как double.

3.2. **Обобщенные характеристики данных: класс памяти, механизмы хранения и доступа**

Каждая переменная и функция, описанная в программе на Си, принадлежит к какому-либо классу памяти. Класс памяти переменной определяет время ее существования и область видимости.

Класс памяти для переменной задается либо по расположению ее описания, либо при помощи специального спецификатора класса памяти, помещаемого перед обычным описанием. Класс памяти для функции всегда extern, если только перед описанием функции не стоит спецификатор static.

Все переменные Си можно отнести к одному из следующих классов памяти:

auto (автоматическая, локальная)
register (регистровая)
extern (внешняя)
static (статическая)

3.2.1. *Автоматические переменные*

Автоматические переменные можно описывать явно, используя спецификатор класса памяти auto. Но такой способ описания применяется редко. Обычно указание на то, что переменная является

автоматической, задается неявно и следует из положения в программе точки описания такой переменной.

По умолчанию принимается, что всякая переменная, описанная внутри функции (локальная переменная) или внутри блока, ограниченного фигурными скобками, и не имеющая явного указания на класс памяти, относится к классу памяти для автоматических переменных.

Поле видимости автоматической переменной начинается от точки ее описания и заканчивается в конце блока, в котором переменная описана. Доступ к таким переменным из внешнего блока невозможен. Память для автоматических переменных отводится динамически во время выполнения программы при входе в блок, в котором описана соответствующая переменная. При выходе из блока память, отведенная под все его автоматические переменные, автоматически освобождается. Теперь понятно происхождение термина «автоматические переменные». Доступ к автоматической переменной возможен только из блока, где переменная описана, так как до момента входа в блок переменная вообще не существует (т.е. под нее не отведена память).

Применение автоматических переменных внутри локальных блоков позволяет приближать описание таких переменных к месту их использования, делая программу более наглядной и иногда облегчая отладку.

Скалярные автоматические переменные при их описании не обнуляются. Пользователь должен сам указать начальное значение для переменных в точке их описания.

Согласно стандарту Си, не разрешается инициализировать автоматические структурные переменные, такие как массивы, структуры и объединения. В C++ такой вид инициализации разрешен, однако при этом может быть нарушена мобильность полученного кода. Безопаснее такие структурные переменные описывать как внешние.

3.2.2. *Регистровые переменные*

Спецификатор памяти register может использоваться только для автоматических переменных или для формальных параметров функции. Такой спецификатор указывает компилятору на то, что пользователь желает разместить переменную не в оперативной памяти, а на одном из быстродействующих регистров компьютера. Компилятор не обязан выполнять такое требование. На большинстве компьютеров имеется только небольшое число регистров, способных удовлетворить желание пользователя.

Спецификацию register рекомендуется использовать для переменных, доступ к которым в функции выполняется часто. Полученный в результате код будет выполняться быстрее и станет более компактным.

Существует некоторое ограничение в использовании регистровых переменных, самое существенное из которых состоит в том, что нельзя обращаться к адресу таких переменных. Регистровыми переменными могут быть объявлены только автоматические переменные. В Си в регистры могут быть помещены лишь переменные типа short и int, а также близкие указатели.

3.2.3. *Внешние переменные и функции*

Любая переменная, описанная в файле вне какой-либо функции и не имеющая спецификатора памяти, по умолчанию относится к классу памяти для внешних переменных. Такие переменные называются *глобальными*.

Для глобальных переменных область видимости простирается от точки их описания до конца файла, где они описаны. Если внутри блока описана автоматическая переменная, имя которой совпадает с именем глобальной переменной, то внутри блока глобальная переменная маскируется локальной. Это означает, что внутри данного локального блока будет видна именно автоматическая переменная.

Для внешних переменных память отводится один раз и остается

занятой до окончания выполнения программы. Если пользователь не укажет инициирующее значение глобальным переменным, то им будет присвоено начальное значение «нуль».

Структурные внешние переменные (массивы, структуры и объединения) могут инициализироваться пользователем в точке их описания.

Внешние переменные видны загрузчику, осуществляющему сборку выполняемой программы из множества объектных файлов. Для того, чтобы переменную можно было использовать в другом файле, для нее следует задать спецификатор памяти extern.

Если описание extern для переменной расположено внутри функции, то его действие распространяется только на данную функцию. Если описание extern находится вне функции, то его действие распространяется от точки описания до конца файла. Таким образом, сочетание внешних переменных и функций в одном файле и описаний extern в других файлах позволяет объединять в один выполняемый файл несколько независимо скомпилированных программ.

Возможность видеть внешние переменные извне файла, где они описываются, предоставляет программистам на Си исключительную гибкость в организации физической структуры программной системы.

Для каждого файла реализации программы (файла с расширением .c), если в нем используются внешние объекты, доступ к которым будет осуществляться из других файлов, рекомендуется создавать интерфейсные файлы (файлы с расширением .h) и помещать туда описание внешних переменных. Тогда для обеспечения доступа к внешним переменным из файлов-потребителей требуется лишь включить в эти файлы соответствующий интерфейсный файл.

По умолчанию считается, что все функции внешние. Местом определения функций является та точка программы, где задаются параметры функций и записывается ее тело. Ко всем функциям, не имеющим спецификатора класса памяти static, обращение из других файлов оказывается возможным, если там функция описывается как внешняя. Таким образом, функция определяется один раз, но может быть описана много раз (с использованием спецификатора extern).

3.2.4. *Статические переменные и функции*

Для упрятывания функций и переменных от загрузчика используется спецификатор памяти static. Функции и переменные, для которых указан такой класс памяти, видимы лишь от точки описания и до конца файла.

Если пользователь не указал инициализирующие значения, то все статические переменные, как и внешние, инициализируются значением «нуль». Инициализация структурных статических переменных выполняется по тем же правилам, что и инициализация внешних.

Если статическая переменная описана внутри функции, то она первый раз инициализируется при входе в блок функции. Значение переменной сохраняется от одного вызова функции до другого. Таким образом, статические переменные можно использовать для хранения значений внутри функции на протяжении времени работы программы, причем такие переменные будут невидимы вне файла, где они определяются.

Спецификатор static в определении функции делает ее невидимой для загрузчика, т.е. недоступной из других файлов.

3.2.5. *Переменные класса volatile*

Спецификатор volatile, введенный в новый стандарт ANSI для Си и реализованный в C++, указывает компилятору на то, что переменная может изменяться не только программой (как обычная переменная), но и вне программы, например при обработке прерывания. Для переменных, специфицированных как в volatile, компилятор не в праве сделать какое-нибудь предположение об их значениях при вычислении выражений, поскольку такие переменные могут измениться даже в момент вычисления выражения. Компилятору запрещено переводить переменную типа volatile в регистровый класс памяти.

3.2.6. *Правила видимости*

1. Переменные, имена новых типов, описываемых с помощью typedef, и перечисленные члены должны быть уникальными внутри блока, в котором они описаны. Идентификаторы, объявленные внешними, должны быть уникальными среди переменных, описанных как внешние.

2. Имена структур, объединений и перечислений должны быть уникальными внутри блока, в котором они описаны. Эти имена, описанные вне пределов какой-либо функции, должны быть уникальны среди всех соответствующих имен, описанных как внешние.

3. Имена членов структуры и объединения должны быть уникальны в структуре или объединении, в которых они описаны. Не существует никаких ограничений на тип или смещение для членов с одинаковыми именами в различных структурах.

4. Метки, на которые ссылаются операторы goto, должны быть уникальными внутри функции, в которой они определены.

3.3. **Основные типы данных**

При написании программы вы работаете с некоторым видом информации, большинство которой попадает в один из четырех основных типов: целые числа, числа с плавающей точкой, текст и указатели.

Целые числа – это числа, которые используются для счета, например: 1, 2, 5, -21 и 752.

Числа с плавающей точкой могут содержать дробные разряды и экспоненту ($5.4567*10^{65}$). Иногда их называют действительными (вещественными) числами.

Текст состоит из символов (a, Z, !, 3) и строк ("Это просто проверка").

Указатели не хранят информацию; вместо этого каждый из них содержит адрес памяти ЭВМ, в которой хранится информация.

3.4. **Описание простейших типов данных**

3.4.1. *Вещественные*

Рассмотрим программу, использующую вещественный тип float:

```
main()
{
 int a,b;
 float ratio;
 printf("Введите два числа: ");
 scanf("%d %d,&a,&b);
 ratio = a / b;
 printf("Отношение = %f \n", ratio);
}
```

Введя два значения (такие, как 10 и 3), получим результат (3.000000). Вероятно, вы ожидали ответ 3.333333. Почему же ответ оказался только 3? Потому, что a и b имеют тип int, отсюда и результат тоже типа int. Он был преобразован к типу float при присваивании его ratio, но преобразование имело место после, а не до деления. Необходимо изменить тип a и b на float, а также строку формата "%d %d" в scanf на "%f %f". Результат будет 3.333333, как и ожидалось.

Имеется также версия типа float, известная как double. Переменные типа double в два раза больше переменных типа float. Это означает, что они могут иметь больше значащих цифр и больший диапазон значений экспоненты.

3.4.2. *Три типа целых*

В дополнение к типу int Си поддерживает версии short int и long int, обычно сокращаемые до short и long. Фактические размеры short, int и long зависят от реализации. Все, что гарантирует Си, – это то, что переменная типа short не будет больше (т.е. не займет

больше байтов), чем типа long. В Си эти типы занимают 16 битов (short), 16 битов (int) и 32 бита (long).

3.4.3. *Беззнаковые*

Си позволяет вам объявлять некоторые типы (char, short, int, long) беззнаковыми (unsigned). Это означает, что вместо отрицательных значений эти типы имеют только неотрицательные (больше или равные нулю). Переменные такого типа могут поэтому хранить большие значения, чем знаковые типы. Например, в Си переменная типа int может содержать значения от -32 768 до 32 767; переменная же типа insigned int может содержать значения от 0 до 65 535. Обе занимают одно и то же место в памяти (16 битов в данном случае), но используют ее по-разному.

3.4.4. *Символьные*

Си поддерживает двухсимвольные данные, например: 'An', '\n\t' и '\007\007'. Они имеют 16-битное представление типа int, причем первый символ находится в младшем байте, а второй в старшем. Такие данные не переносятся в другие компиляторы Си.

Одиночные символы, такие как 'A', '\t' и '\007', также имеют 16-битное представление типа int. В этом случае младший байт является сигналом переполнения в старшем байте, т.е. если десятичное значение больше чем 127, то старший байт устанавливается в -1 [=0xFF]. Это может быть запрещено объявлением, что тип char по умолчанию является незначащим для этого (используется опция компилятора -k или конструкция Default char type...Unsigned в подменю Options/Compiler/Source, делающая старший байт нулевым, не считаясь со значением младшего байта).

Си поддерживает ANSI расширение, допускающее шестнадцатеричное представление кодов символов, например: '\x1F','\x82' и т.д. Кроме того, допустима запись x и X, а также использование от одной до трех цифр.

3.4.5. *Типы со знаком*

Классический Си подразумевает, что все int-типы являются типами со знаком, и потому включает модификатор типа unsigned, чтобы специфицировать обратное. По умолчанию переменные типа char считаются signed (со знаком), что подразумевает изменение их значений от -128 до 127.

Однако в настоящее время большинство компиляторов Си, реализованных для современных моделей компьютеров, воспринимают тип данных char как unsigned. В связи с этим для повышения мобильности программ на Турбо Си в компилятор пакета включена опция, позволяющая компилировать тип данных char как unsigned по умолчанию. В противном случае вы можете объявить символьную знаковую переменную как signed char.

3.5. **Тип enum-перечислимый**

Си поддерживает все перечислимые типы ANSI стандарта. Перечислимый тип данных используется для описания дискретного множества целых значений. Например, вы можете объявить следующее:

```
enum days { sun, mon, tues, wed, thur, fri, sat };
```

Имена, перечисленные в days, являются целыми константами: первая (sun) автоматически установлена в ноль, а каждая следующая имеет значение на единицу больше, чем предыдущая (mon=1, tues=2 и т.д.). Однако вы можете присвоить константам и определенные значения. Следующие имена без конкретных значений будут в этом случае, как и раньше, иметь значения предыдущих элементов с увеличением на единицу, например:

```
enum coins {penny=1, nickle=5, dime=10, quarter=25};
```

Переменной перечислимого типа может быть присвоено любое значение типа int – проверка типа не производится.

3.6. **Тип void**

В Си поддерживается тип void, определенный в ANSI стандарте. Он используется для явного описания функций, не возвращающих значений. Аналогично пустой список параметров может быть объявлен зарезервированным словом void, например:

```
void putmsg(void)
{
printf("Hello, world\n");
}
main()
{
putmsg();
}
```

Вы можете преобразовывать выражение к типу void, для того чтобы явно указать, что значение, возвращаемое функцией, игнорируется. Например, если вы хотите приостановить выполнение программы до тех пор, пока пользователь не нажмет какую-либо клавишу, вы можете написать:

```
(void) getch();
```

Наконец, вы можете объявить указатель на объект типа void. Данный указатель не будет указателем на ничто, а будет указателем на любой тип данных, причем конкретный тип этих данных знать не обязательно. Вы можете присваивать любой указатель указателю типа void (и наоборот) без приведения типов. Однако вы не можете использовать оператор косвенной адресации (*) с указателем void, так как используемый тип не определен.

3.7. **Расширенные описания – использование директивы typedef**

В классическом Си определенные пользователем типы данных именуются редко, за исключением структур и объединений, перед

любым объявлением которых вы ставите ключевые слова struct или union.

В C++ обеспечивается другой уровень информационной содержательности путем использования директивы typedef. Она позволяет связать нужный тип данных (включая struct и enum) с некоторым именем, а затем объявить переменные этого типа. Далее представлен пример определения типа и определение переменных этого нового, введенного пользователем типа данных:

```
typedef int *intptr;
typedef char namestr[30];
typedef enum ( male, female, unknown) sex;
typedef struct {
  namestr last, first;
  char   ssn[9];
  sex   gender;
  short  age;
  float  gpa;
} student;
typedef student class[100];
class  hist104,ps102;
student valedictorian;
intptr iptr;
```

Использование typedef делает программу более читабельной, а также позволяет вам не ограничиваться одним участком программы для определения типов данных, а распространить их определение на всю программу по мере их появления и использования в ней.

3.8. **Иерархия типов, совместимость и преобразование типов в Си**

3.8.1. *Преобразование типов char, int и enum*

Преобразование символьной константы к целому типу имеет результатом 16-битное значение, поскольку и одно-, и двухсимвольные константы представляются 16-битным значением. Результат

преобразования символьного объекта (такого как переменная) к целочисленному объекту автоматически получает знаковое расширение, если вы не сделали по умолчанию тип char беззнаковым, используя при компиляции опцию -k. Объекты типа signed char при преобразовании их в int всегда используют знаковое расширение; объекты типа unsigned char всегда устанавливают старший байт в ноль.

Значения типа enum преобразуются в int без модификации. Аналогично значения, имеющие тип int, могут преобразовываться в значения перечислимого типа enum, а символы преобразуются в int значения и обратно.

3.8.2. *Преобразование указателей*

В Си различные указатели программы могут быть различных размеров в зависимости от используемой модели памяти или используемого модификатора типа указатель. Например, когда вы компилируете программу, используя конкретную модель памяти, адресуемые модификаторы (near, far, huge, _cs, _ds, _es, _ss) в вашем исходном тексте могут изменять размер указателя, заданный данной моделью памяти.

Указатель должен быть объявлен как указатель на некоторый конкретный тип, даже если данный тип – void (который в действительности означает указатель на ничего). Однако, будучи объявленным, такой указатель может указывать на объект любого другого типа. Си позволяет вам переназначать указатели, но компилятор будет предупреждать вас в случаях переназначения, если только предварительно этот указатель не был объявлен как указатель на тип void. Однако указатели на типы данных не могут быть преобразованы в указатели на функции, и наоборот.

3.8.3. *Арифметические преобразования*

В Си задаются обычные арифметические преобразования, определяющие, что произойдет с любыми величинами, использованными в арифметических выражениях (операнд – оператор – операнд).

Здесь приведены шаги, используемые Си, для преобразования операндов в арифметических выражениях:

1. Любой не integer и не double тип преобразуется в соответствии с табл. 3.2. После этого любые два значения, объединенные оператором, являются целыми – int (включая long и unsigned модификаторы) или с плавающей точкой – double.

2. Если какой-либо операнд имеет тип double, то другой операнд тоже преобразуется в double.

3. В случае, если один из операндов имеет тип unsigned long, то другой операнд тоже преобразуется в unsigned long.

4. В случае, если один из операндов имеет тип long, то другой операнд тоже преобразуется в long.

5. В случае, если какой-либо из операндов имеет тип unsigned, то другой операнд тоже преобразуется в unsigned.

6. В последнем варианте оба операнда являются типа int.

Результат выражения определяется типами обоих операндов.

Таблица 3.2

Методы, используемые для определения результата в арифметических преобразованиях

Тип	Преобразуется в	Метод
char	int	Со знаком
unsigned char	int	Нулевой старший байт (всегда)
signed char	int	Со знаком (всегда)
short	int	Если unsigned, то unsigned int
enum	int	То же значение
float	double	Мантисса дополняется нулями

1. Могут ли вещественные числа быть беззнаковыми? **2.** Сколько и какие формы внутреннего представления данных с плавающей точкой использует компилятор? **3.** Какой объем памяти занимают данные типа int? **4.** В каких системах счисления могут задаваться константы типа int? **5.** Для переменной int x=25 определите результаты следующих вычислений: x/2; x%3; x++; x+4; ++x+4; x*x; x/2.; (float)x/3. **6.** Представьте в экспоненциальной форме

вещественные числа: 0.008; 456.9087; -86.66; 0.0000054; 1.234; 57.009; 300000.8.

Глава 4

ОПЕРАЦИИ И УПРАВЛЯЮЩИЕ КОНСТРУКЦИИ

4.1. **Пространство операций над простейшими типами**

4.1.1. *Операция присваивания*

Самой общей операцией является присваивание, например: ratio = = a/b или ch = getch(). В Си присваивание обозначается одним знаком равенства (=); при этом значение справа от знака равенства присваивается переменной слева.

Можно применять также последовательные присваивания, например: sum = a = b. В таких случаях присваивание производится справа налево, т.е. b будет присвоено a, которая, в свою очередь, будет присвоена sum, так что все три переменных получат одно и то же значение (а именно начальное значение b).

4.1.2. *Арифметические операции*

Си поддерживает обычный набор арифметических операций:

– умножение (*);

– деление (/);

– остаток от деления целого на целое (%);

– сложение (+);

– вычитание (-).

Си поддерживает одноместный минус (a + (-b)), который выполняет двоичное дополнение, как в расширении ANSI. Кроме того, Си поддерживает одноместный плюс (a + (+b)).

Стараясь создать эффективно компилируемые выражения, Турбо Си обычно выполняет перегруппировку выражения с переупорядочиванием коммутативных операторов (таких как * и бинарный

+). Однако Си не будет переупорядочивать выражения с унарным оператором +. Этим качеством вы можете управлять в вычислениях с плавающей точкой, которые чувствительны к ошибкам точности и переполнения, т.е. можете включать в отдельные выражения промежуточные преобразования с унарным оператором + без разделения преобразования. Например, если a, b, c и f имеют тип float, то выражение

```
f = a + + (b + c);
```

заставит сначала оценить значение (b + c), а затем прибавить результат к a.

4.1.3. *Операции приращения* (++) *и уменьшения* (--)

В Си имеются некоторые специальные одноместные и двуместные операции. Наиболее известными являются одноместные операции приращения (++) и уменьшения (--). Они позволяют использовать единственную операцию, которая добавляет 1 или вычитает 1 из любого значения. Сложение и вычитание может быть выполнено в середине выражения, причем вы можете даже решить, сделать это до или после вычисления выражения. Рассмотрим следующие строки программы:

```
sum = a + b++;
sum = a + ++b;
```

Первая означает: «сложить a и b, присвоить результат sum и увеличить b на единицу».

Вторая означает: «увеличить b на единицу, сложить a и b и присвоить результат sum».

Это очень мощные операции, расширяющие возможности языка, однако перед их использованием нужно убедиться, что вы хорошо понимаете их действие.

Пример программы, использующей операции ++ и -- (попытайтесь определить, что она выведет):

```
main()
{
  int a,b,sum;
  char *format;
  format = "a = %d  b = %d  sum = %d \n";
  a = b = 5;
  sum = a  + b; printf(format,a,b,sum);
  sum = a++ + b; printf(format,a,b,sum);
  sum = ++a + b; printf(format,a,b,sum);
  sum = --a + b; printf(format,a,b,sum);
  sum = a-- + b; printf(format,a,b,sum);
  sum = a  + b; printf(format,a,b,sum);
}
```

4.1.4. *Побитовые операции*

Для обработки на уровне битов Си имеет следующие операции:

– сдвиг влево (<<);

– сдвиг вправо (>>);

– И (&);

– ИЛИ (|);

– исключающее ИЛИ (^);

– НЕ (~).

Они позволяют производить операции очень низкого уровня. Для того, чтобы понять эффект этих операций, рассмотрим следующую программу:

```
main()
{
  int a,b,c;
  char *format1, *format2;
  format1 = " %04X %s %04X = %04X\n";
  format2 = " %c%04X = %04X\n";
```

```
 a = 0x0FF0; b = 0xFF00;
 c = a<<4;  printf(format1,a,"<<",4,c);
 c = a>>4;  printf(format1,a,">>",4,c);
 c = a & b; printf(format1,a,"& ",b,c);
 c = a | b; printf(format1,a,"| ",b,c);
 c = a ^ b; printf(format1,a,"^ ",b,c);
 c = ~a;   printf(format2,`~`,a,c);
 c = -a;   printf(format2,`-`,a,c);
}
```

Попробуйте предугадать то, что будет выводить эта программа. Заметим, что спецификаторы ширины поля выравнивают выводимые значения; спецификатор %04X указывает на использование нулей в начале числа, на ширину поля вывода (4) и на шестнадцатеричное представление (основание 16).

4.1.5. *Комбинированные операции*

Си позволяет использовать некоторые сокращения при написании выражений, содержащих многочисленные операции, описанные выше (одноместные, двуместные, приращение, уменьшение и побитовые).

Так, любое выражение вида

<переменная> = <переменная> <операция> <выражение>;

может быть заменено на

<переменная> <операция> = <выражение>;

Вот некоторые примеры таких выражений и способы их сокращения:

```
a = a + b; сокращается до a += b;
a = a – b; сокращается до a -= b;
a = a * b; сокращается до a *= b;
a = a / b; сокращается до a /= b;
```

a = a % b; сокращается до a %= b;
a = a << b; сокращается до a <<= b;
a = a >> b; сокращается до a >>= b;
a = a & b; сокращается до a &= b;
a = a | b; сокращается до a |= b;
a = a ^ b; сокращается до a ^= b.

4.1.6. *Адресные операции*

Си поддерживает две специальные адресные операции: операцию определения адреса (&) и операцию обращения по адресу (*).

Операция & возвращает адрес данной переменной. Если sum является переменной типа int, то &sum является адресом (расположением в памяти) этой переменной. С другой стороны, если msg является указателем на тип char, то *msg является символом, на который указывает msg. Рассмотрим следующую программу:

```
main()
{
int sum;
char *msg;
sum = 5 + 3;
msg = "Hello, thore\n";
printf(" sum = %d &sum = %p \n", sum, &sum);
printf("*msg = %c msg = %p \n", *msg, msg);
}
```

В первой строке печатается два значения: значение sum (8) и адрес sum (назначаемый компилятором). Во второй строке также печатается два значения: символ, на который указывает msg (H), и значение msg, которое является адресом этого символа (также назначен компилятором).

4.1.7. *Операции сравнения*

Операции сравнения позволяют сравнивать два значения, получая результат в зависимости от того, дает ли сравнение «истину»

или «ложь». Если сравнение дает «ложь», то результирующее значение равно нулю; если значение «истина», то результат равен 1. Вот список операций Си для сравнения:

– > больше;
– >= больше или равно;
– < меньше;
– <= меньше или равно;
– == равно;
– != не равно.

Почему нас должно заботить, является ли нечто «истиной» или «ложью»? Рассмотрим программу:

```
main ()
{
 float a,b,ratio;
 printf("Введите два числа: ");
 scanf("%f %f",&a,&b);
 if (b == 0.0)
  printf("Отношение не определено\n");
 else {
  ratio= a / b;
  printf("Отношение = %f \n",ratio);
 }
}
```

Оператор, находящийся в двух следующих за оператором scanf строках, известен как условный оператор if. Вы можете понимать его так: «Если значение выражения (b == 0.0) истинно, сразу вызвать printf. Если значение выражения ложно, присвоить a/b переменной ratio, затем вызвать printf».

Теперь, если вы введете 0 в качестве второго значения, то ваша программа напечатает сообщение

Отношение не определено.

Если второе значение не нулевое, то программа вычисляет и печатает ratio.

4.1.8. *Логические операции*

Имеется также три логических операции: И (&&), ИЛИ (||) и НЕ (!). Их не следует путать с описанными выше битовыми операциями (&,|,~). Логические операции работают с логическими значениями («истина» или «ложь») и позволяют составлять логические выражения. Как же их отличать от соответствующих битовых операций?

1. Эти логические операции всегда дают в результате значение либо 0 («ложь»), либо 1 («истина»), в то время как поразрядные операторы выполняются путем последовательной обработки цепочки битов до тех пор, пока не станет ясен результат.

2. Логические операторы && и !! известны как операторы типа "short circuit". Выполнение операторов такого типа прекращается, как только становится ясно, будет ли результат иметь значение «истина» или «ложь». Предположим, что вы имеете выражение вида

exp1 && exp2

Если exp1 – «ложь», значит и все выражение – «ложь». Таким образом, exp2 никогда не будет вычисляться. Аналогично, если мы имеем выражение вида

exp1 !! exp2,

то exp2 никогда не будет вычисляться, если exp1 верно.

4.1.9. *Операция следования* (*запятая*)

Вы можете использовать операцию запятая (,) для организации множественных выражений, расположенных внутри круглых скобок. Выражение внутри скобок вычисляется слева направо, и все выражение принимает значение, которое было вычислено последним. Например, если oldch и ch имеют тип char, то выражение

(oldch = ch, ch = getch())

присваивает переменной oldch значение ch, затем считывает символ, вводимый с клавиатуры, и запоминает его в ch. Результатом всего выражения в итоге будет значение введенного с клавиатуры символа. Приведем еще один пример:

```
ch='a';
if((oldch = ch, ch = 'b') == 'a')
puts("Это символ 'a'\n");
else
puts("Это символ 'b'\n");
```

Как вы считаете, какое сообщение будет выведено на экран вашего дисплея в результате выполнения приведенной выше программы?

4.1.10. *Приоритет и порядок выполнения операций*

Если в выражении не используются круглые скобки, задающие порядок выполнения операций, то группировка операндов для операций производится с учетом приоритета операций. Например, в выражении

first *= second <= third

операнд second группируется с операцией <=, поскольку операция <= имеет более высокий приоритет, чем операция *=. Приведенное выражение эквивалентно выражению

first *= (second <= third).

Заметьте, что выражение (second <= third) может принимать значение «нуль» или «единица».

В примере

first = second -= third

операции = и -= имеют одинаковый приоритет, однако операнд second группируется с операцией справа от него, поскольку операции = и -= являются правоассоциируемыми (выполняются справа налево).

Все двухместные и трехместные операции являются левоассоциируемыми (выполняются слева направо), за исключением условных операций и операций присваивания, являющихся правоассоциируемыми.

В табл. 4.1 приведены операции Си в порядке убывания приоритета.

Таблица 4.1

Операции Си в порядке убывания приоритета

Операция	Назначение
1	2
[]	Задание элемента массива
()	Вызов функции
.	Выбор поля структуры
->	Выделение поля структуры с помощью указателя
++,--	Постфиксное/префиксное увеличение и уменьшение на 1: если и то и другое встречается в одном выражении, то постфиксное имеет более высокий приоритет
sizeof	Определение размера переменной в байтах
(тип)	Приведение к типу
~	Побитовое отрицание
!	Логическое НЕ
-	Унарный минус
&	Определение адреса
*	Обращение по адресу
*,/,%	Умножение, деление и остаток – одинаковый приоритет
+, –	Сложение, вычитание – одинаковый приоритет
<<,>>	Сдвиг влево, сдвиг вправо – одинаковый приоритет

Окончание

1	2
<,>,<=,>=	Сравнение – одинаковый приоритет
==,!=	Равенство, неравенство – одинаковый приоритет
&	Побитовое И
^	Побитовое исключающее ИЛИ
|	Побитовое ИЛИ
&&	Логическое И
||	Логическое ИЛИ
?:	Условный оператор
=,+=,-=,*=, /=,<<=,>>=, &=,^=,|=	Присваивание и замещение – одинаковый приоритет
,	Операция запятая, которая предписывает последовательное вычисление выражений

4.2. **Операторы присваивания**

Любой оператор присваивания, заключенный в круглые скобки, является выражением с определенным значением, которое получается в результате этого присваивания.

Например, выражение (sum = 5 + 3) имеет значение 8, поэтому выражение ((sum = 5 + 3) <= 10) будет всегда иметь значение «истина» (так как 8 <= 10). Более экзотичен следующий пример:

```
if ((ch=getch()) == 'q')
  puts("До свидания! Программа завершена.\n");
else
  puts("Продолжаем работу!\n");
```

Как работает эта конструкция? Если в программе используется выражение ((ch=getch()) == 'q'), то она, дойдя до него, останавливается и переходит в состояние ожидания ввода символа с клавиатуры. После того, как вы введете символ, осуществляется присваивание введенного символа переменной ch и выполняется сравнение введенного символа с символом 'q'. Если введенный символ равен

'q', то на экран будет выведено сообщение "До свидания! Программа завершена". В противном случае будет выведено сообщение "Продолжаем работу!"

В языках Паскаль и Бейсик проверка на равенство производится выражением

```
if (a = b).
```

```
if (a = b) puts("Равно");
else puts("Не равно");
```

Если это программа на Паскале или Бейсике, то вы можете предполагать, что будет напечатано "Равно", если a и b имеют одинаковое значение, и "Не равно" в противном случае.

Иначе происходит с программой на Си, где выражение a = b означает «Присвоить значение b переменной a», и все выражение принимает значение b. Поэтому во фрагменте, приведенном выше на языке Си, присвоится значение b переменной a, а затем напечатается "Равно", если b имеет нулевое значение, в противном случае – "Не равно".

Правильное решение следующее:

```
if (a == b) puts("Равно");
else puts("Не равно");
```

4.3. **Операторы условного перехода**

4.3.1. *Оператор if-else*

Обратимся снова к оператору if, который фигурировал при рассмотрении предыдущих примеров. Оператор if имеет следующий основной формат:

```
if (значение)
  оператор1;
```

```
else
  оператор2;
```

где «значение» является любым выражением, которое приводится или может быть приведено к целочисленному значению. Если «значение» отлично от нуля («истина»), то выполняется «оператор1», в противном случае выполняется «оператор2».

Дадим пояснение относительно двух важных моментов по использованию оператора if-else.

Во-первых, часть «else оператор2» является необязательной частью оператора if. Другими словами, правомерно употребление следующей формы оператора if:

```
if (значение)
 оператор1;
```

В этой конструкции «оператор1» выполняется тогда и только тогда, когда «значение» отлично от нуля. Если «значение» равно нулю, «оператор1» пропускается и программа продолжает выполняться дальше.

Во-вторых, что делать если вы хотите выполнить более одного оператора в зависимости от того ложно или истинно выражение, указанное в операторе if? Ответ: используйте составной оператор if.

Составной оператор состоит из:

– левой, или открывающей, фигурной скобки ({);

– последовательности операторов, разделенных между собой точкой с запятой (;);

– правой, или закрывающей, фигурной скобки (}).

В приведенном ниже примере в предложении if используется один оператор:

```
if (b == 0.0)
printf("Отношение не определено\n");
```

а в предложении else – составной оператор:

```
else {
  ratio = a/b;
  printf( "Значение отношения равно %f\n", ratio);
}
```

Вы можете также заметить, что тело вашей программы (функции main) является подобием составного оператора if.

4.3.2. *Условный оператор* (?:)

В некоторых случаях необходимо произвести выбор между двумя альтернативами (и результирующими значениями), основанный на некотором условии. Обычно это реализуется оператором if ... else, например, так:

```
if (a < b) return(a);
else    return(b);
```

Но, как оказывается, для реализации такого типа выбора достаточно одной специальной конструкции. Ее формат следующий:

выражение 1 ? выражение 2 : выражение 3.

А смысл таков: «если выражение 1 верно, то вычисляется выражение 2 и все выражение получает его значение; иначе вычисляется выражение 3 и передается его значение». Используя эту конструкцию, приведенный выше фрагмент программы можно представить следующим образом:

```
return((a < b) ? a : b ).
```

Более того, такие действия по определению минимального из двух чисел можно реализовать как строку макроса:

```
#define imin(a,b) ((a < b) ? a : b).
```

Теперь, где бы ваша программа ни встретила выражение imin(e1,e2), она замещает его на ((e1<e2) ? e1 : e2) и продолжает

вычисления. Это в действительности наиболее общее решение, так как a и b больше не ограничены типом int; они могут быть любого типа, с которым можно выполнить операцию сравнения.

4.4. **Оператор выбора**

Часто бывает необходимо построить длинные конструкции типа

```
  *done = 0;
  do {
 cmd = toupper(getch());
  { if (cmd == 'F') do_file_menu(done);
  else if (cmd == 'R') run_program();
  else if (cmd == 'C') do_compile();
  else if (cmd == 'M') do_make();
  else if (cmd == '?') do_project_menu();
  else if (cmd == 'O') do_option_menu();
  else if (cmd == 'E') do_error_menu();
  else handle_others(cmd,done);
   } while (!*done);
 }
```

Подобная ситуация встречается настолько часто, что в Си была введена специальная управляющая структура, которая носит название оператор switch. Вот та же функция, но записанная с использованием оператора switch :

```
#include <ctype.h>
do_main_menu(short *done)
{
  char cmd;
  *done = 0;
  do {
cmd = toupper(getch());
       switch(cmd) {
     case 'F' : do_file_menu(done); break;
     case 'R' : run_program(); break;
     case 'C' : do_compile(); break;
```

```
        case 'M' : do_make(); break;
        case '?' : do_project_menu(); break;
        case 'O' : do_option_menu(); break;
        case 'E' : do_error_menu(); break;
        default : handle_others(cmd,done);
      } while (!*done);
    }
```

Эта функция организует цикл, в котором символ считывается, преобразуется к значению на верхнем регистре, а затем запоминается в переменной cmd. Потом введенный символ обрабатывается оператором switch на основе значения cmd.

Цикл повторяется до тех пор, пока выражение *done не станет равным 1 (предположительно в функции do_file_menu или handle_ others).

Оператор switch получает значение cmd и сравнивает его с каждым значением метки case. Если они совпадают, начинается выполнение операторов данной метки, которое продолжается либо до ближайшего оператора break, либо до конца оператора switch. Если ни одна из меток не совпадает и вы включили метку default в оператор switch, то будут выполняться операторы этой метки; если метки default нет, оператор switch целиком игнорируется.

Значение value, используемое в switch(value), должно быть приведено к целому значению. Другими словами, это значение должно легко преобразовываться в целое для таких типов данных, как char, разновидности enum и, конечно, int, а также для всех его вариантов.

Нельзя использовать в операторе switch вещественные типы данных (такие как float и double), указатели, строки и другие структуры данных, но разрешается использовать элементы структур данных, совместимых с целыми значениями.

Хотя (value) может быть выражением (константа, переменная, вызов функции и другие их комбинации), метки case должны содержать константы. Кроме того, в качестве ключевого значения case может быть только одно значение. Если бы do_main_menu не использовало функцию toupper для преобразования cmd, оператор switch мог бы выглядеть следующим образом:

```
switch (cmd) {
  case 'f' :
  case 'F' : do_file_menu(done);
    break;
  case 'r' :
  case 'R' : run_program();
    break;
  ...
```

Этот оператор выполняет функцию do_file_menu независимо от того, в каком регистре поступает значение cmd. Аналогично выполняются действия для других альтернатив значения cmd.

Запомните, что для завершения данного case вы должны использовать оператор break. В противном случае будут выполняться последовательно все операторы, относящиеся к другим меткам (до тех пор, пока не встретится оператор break).

Если вы уберете оператор break после вызова do_file_menu, то при вводе символа F будет вызываться do_file_menu, а затем будет вызвана функция run_program.

Однако иногда вам нужно сделать именно так. Рассмотрим следующий пример:

```
typedef enum( sun, mon, tues, wed, thur, fri, sat, ) days;
main()
{
  days today;
  ...
  swith (today) {
    case mon :
    case tues :
    case wed :
    case thur :
    case fri : puts("Иди работать!");break;
    case sat : printf("Убери во дворе и ");
    case sun : puts("Расслабься!");
}
```

В этом операторе switch для значений от mon до fri выполняется одна и та же функции puts, после которой оператор break указывает

на выход из switch. Однако если today равно sat, то выполняется соответствующая функция printf, а затем выполняется puts ("Расслабься!"). Если же today равно sun, то выполняется только последняя функция puts.

4.5. **Циклические конструкции в Си-программах**

Наряду с операторами (или группами операторов), которые могут выполняться в зависимости от каких-либо условий, существуют еще и операторы, которые могут выполняться несколько раз в одной и той же последовательности. Такой вид конструкции в программе известен как цикл. Существуют три основных типа циклов (хотя два из них можно рассматривать как разновидность одного). Это цикл while («пока»), цикл for («для») и цикл do...while («делать ... пока»). Рассмотрим их по порядку.

4.5.1. *Оператор while*

Цикл while является наиболее общим и может использоваться вместо двух других типов циклических конструкций. В принципе можно сказать, что по-настоящему для программирования необходим только цикл while, а другие типы циклических конструкций служат лишь для удобства написания программ.

Рассмотрим пример программы:

```
#include <stdio.h>
 main()
 {
  char ch;
  int len;
  len=0;
  puts("Наберите предложение, затем нажмите <Ввод>");
   while ((ch=getch()) != '\n') {
     putchar(ch);
     len++;
     }
```

```
printf("\nВаше предложение имеет длину %d символов\n",len);
}
```

Эта программа позволяет ввести предложение с клавиатуры и подсчитать при этом, сколько раз вы нажали на клавиши клавиатуры до тех пор, пока не нажали на клавишу <Ввод> (соответствует специальному символу конца строки – '\n'). Затем программа сообщит вам, сколько символов (символ '\n' не подсчитывается) вы ввели.

Для отображения вводимых символов на экране дисплея используется функция putchar, так как функция getch не обеспечивает режима «эхо» для вводимых с ее помощью символов.

Заметьте, что в условном выражении оператора while используется оператор присваивания. Это позволяет программе читать и одновременно сравнивать считанные символы с символом, соответствующим клавише <Ввод>. Если считанный символ не <Ввод>, то программа отображает его на экране дисплея и увеличивает на единицу значение len.

Оператор while имеет следующий формат:

```
while (выражение)
  оператор;
```

где «выражение» принимает нулевое или отличное от нуля значение, а «оператор» может представлять собой как один оператор, так и группу операторов.

В процессе выполнения цикла while вычисляется значение «выражения». Если оно истинно, то «оператор», следующий за ключевым словом while, выполняется и «выражение» вычисляется снова. Если «выражение» ложно, то цикл while завершается и программа продолжает выполняться дальше. Обратите внимание на другой пример цикла while:

```
main()
{
  char *msg;
```

```
  int indx;
  msg = "Здравствуй, мир";
  indx = 1 ;
  while (indx <= 10 ) {
    printf("Время # %2d: %s\n", indx,msg);
    indx++;
    }
}
```

После компиляции и выполнения этой программы на экране будут отображены строки со следующей информацией:

```
Время # 1 : Здравствуй, мир
Время # 2 : Здравствуй, мир
Время # 3 : Здравствуй, мир
 .....
Время # 9 : Здравствуй, мир
Время # 10 : Здравствуй, мир
```

Очевидно, что оператор printf был выполнен ровно десять раз. При этом значение параметра цикла indx изменилось от 1 до 10.

Возможно переписать этот цикл несколько компактнее:

```
indx = 0 ;
while (indx++ < 10 )
  printf("Время #%2d: %s\n",indx,msg);
```

4.5.2. *Оператор for*

Цикл for является одним из основных видов циклов, которые имеются во всех универсальных языках программирования, включая Си. Однако версия цикла for, используемая в Си, как вы увидите, обладает большей мощностью и гибкостью.

Основная идея, заложенная в его функционирование, заключается в том, что операторы, находящиеся внутри цикла, выполняются фиксированное число раз, в то время как переменная цикла (известная еще как индексная переменная) пробегает определенный

ряд значений. Например, модифицируем программу, о которой говорилось выше, в следующую:

```
main()
{
  char *msg;
  int indx;
  msg = "Здравствуй, мир";
  for (indx = 1; indx <= 10; indx++ )
  printf("Время #%2d: %s\n",indx,msg);
}
```

Эта программа делает те же действия, что и программа с циклом while, которую мы уже разобрали, и является точным эквивалентом первого ее варианта.

Теперь приведем основной формат цикла for:

```
for (выр1; выр2; выр3)
    оператор;
```

Так же, как и в цикле while, «оператор» в теле цикла for обычно является одним из операторов программы, но может использоваться и составной оператор, заключенный в фигурные скобки ({...}).

Заметим, что параметры цикла for, заключенные в скобки, должны разделяться точкой с запятой (позиционный параметр), которая делит, в свою очередь, пространство внутри скобок на три сектора. Каждый параметр, занимающий определенную позицию, означает следующее:

– выр1 – обычно задает начальное значение индексной переменной;

– выр2 – условие продолжения цикла;

– выр3 – обычно задает некоторую модификацию (приращение) индексной переменной за каждое выполнение цикла.

Основной вариант цикла for эквивалентен следующей конструкции, реализованной с помощью цикла while:

```
выр1;
 while (выр2) {
```

```
  оператор;
выр3;
}
```

Вы можете опускать одно, несколько или даже все выражения в операторе for, однако о необходимости наличия точек с запятой вы должны помнить всегда. Если вы опустите «выр2», то это будет равносильно тому, что значение выражения «выр2» всегда будет иметь значение 1 («истина») и цикл никогда не завершится (такие циклы известны еще как бесконечные).

Во многих случаях вам поможет использование операции запятая (,), которая позволяет вводить составные выражения в оператор цикла for. Вот, например, еще одна правильная модификация выражений в операторе for:

```
main()
{
  char *msg;
  int up,down;
  msg = "Здравствуй, мир";
  for (up = 1, down = 9; up <= 10; up++, down--)
  printf("%s: %2d растет, %2d уменьшается \n",msg,up,down);
}
```

Заметьте, что и первое, и последнее выражение в этом цикле for состоит из двух выражений, инициализирующих и модифицирующих переменные up и down. Вы можете сделать эти выражения сколь угодно сложными. Возможно, вы слышали о легендарных хаккерах Си (hacker – программист, способный писать программы без предварительной разработки спецификаций и оперативно вносить исправления в работающие программы, не имеющие документации), которые включают большинство своих программ в три условных выражения оператора for, оставляя в теле цикла лишь несколько операторов.

4.5.3. *Оператор* do...while

Последним видом цикла является цикл do...while. Рассмотрим программу:

```
main()
{
 float a,b,ratio;
 char ch;
 do {
   printf("Введите два числа: ");
 scanf("%f %f", &a, &b);
 if (b == 0.0)
   printf("\nВнимание! Деление на ноль!");
 else {
   ratio = a/b;
   printf("\nРезультат деления двух чисел: %f",ratio);
 }
 printf("\nНажми 'q' для выхода или любую клавишу для
     продолжения")
 } while (( ch = getch()) != 'q');
}
```

Эта программа вычисляет результат деления одного числа на другое. Оба числа вводятся по запросу программы с клавиатуры. Если вы введете символ 'q', то выражение в операторе цикла while в конце программы примет значение «ложь» и цикл (а значит и программа) завершится. Если вы введете какой-либо другой символ, отличный от 'q', то выражение будет иметь значение «истина» и цикл повторится.

Формат цикла do...while можно представить в виде

```
do оператор; while (выр);
```

Основное отличие между циклом while и циклом do...while в том, что операторы внутри do...while всегда выполняются хотя бы один раз (так как проверка условия выполнения цикла осуществляется после выполнения последовательности операторов, состав-

ляющих тело цикла). Это похоже на цикл repeat...until в Паскале с одним, однако, различием: цикл repeat выполняется до тех пор (until), пока его условие – «ложь»; а цикл do...while выполняется до тех пор, пока (while) его условие – «истина».

4.6. **Операторы передачи управления**

Это дополнительные операторы, предназначенные для использования в управляющих операторах или для моделирования других управляющих структур. Оператор return позволяет вам досрочно выйти из функции. Операторы break и continue предназначены для использования в цикле и позволяют пропустить последующие операторы программы.

Один совет: подумайте дважды перед использованием каждого оператора передачи управления (за исключением, конечно, return).

Используйте их в тех случаях, когда они представляют наилучшее решение, но помните, что чаще всего вы можете решить возникшую перед вами проблему без их помощи. Особенно избегайте оператора goto: операторы return, break или continue наверняка заменят его вам.

4.6.1. *Оператор return*

Существует два основных способа использования оператора return.

Во-первых, в том случае, если функция возвращает значение и вам необходимо использовать его в зависимости от того, какое значение возвращается в вызывающую программу, например:

```
int imax(int a, int b)
{
 if (a > b) return(a);
 else    return(b);
}
```

Здесь функция использует оператор return для возвращения максимального из двух переданных ей значений.

Второй способ использования оператора return состоит в возможности выхода из функции в некоторой точке до ее завершения. Например, функция может определить условие, по которому производится прерывание. Вместо того, чтобы помещать все основные операторы функции внутрь оператора if, для выхода можно использовать оператор return. Если функция имеет тип void (т.е. не возвращает никакого значения), можно написать return без возвращаемого значения.

Рассмотрим фрагмент программы:

```
int imin(int list[], int size)
{
 int i, minindx, min;
 if (size <= 0) return(-1);
 ...
}
```

В этом примере, если параметр size меньше либо равен нулю, то массив list пуст, в связи с чем оператор return вызывает выход из функции.

Заметим, что в случае ошибки возвращается значение -1. Поскольку -1 никогда не может быть индексом массива, вызывающая программа регистрирует факт возникновения ошибки.

4.6.2. *Onepamop break*

Иногда бывает необходимо выйти из цикла до его завершения. Рассмотрим следующую программу:

```
#define LIMIT 100
#define MAX 10
main()
{
 int i,j,k,score;
```

```
int scores[LIMIT][MAX];
for (i = 0; i < LIMIT; i++) {
        j = 0;
        while (j < MAX-1) {
        printf("Введите следующее значение #%d: ",j);
        scanf("%d", score);
        if (score < 0)
          break;
        scores[i][++j] = score;
    }
    scores[i][0] = j;
 }
}
```

Рассмотрим оператор if (score < 0) break;. Он указывает, что если пользователь введет отрицательное значение score, цикл while прерывается. Переменная j используется и в качестве индекса scores, и в качестве счетчика общего количества элементов в каждой строке. Это значение записывается в первом элементе строки.

Вспомните использование оператора break в операторе switch, представленное ранее. Там break указывает программе выйти из оператора switch, здесь он указывает программе выйти из цикла и продолжить работу. Кроме оператора switch оператор break может быть использован во всех трех циклах (for, while и do...while), однако его нельзя использовать в конструкции if...else или в теле главной процедуры main для выхода из нее.

4.6.3. *Оператор continue*

Иногда нужно не выходить из цикла, а пропустить ряд операторов в теле цикла и начать его заново. В этом случае можно применить оператор continue, предназначенный специально для этого. Обратите внимание на следующую программу:

```
#define LIMIT 100
#define MAX 10
main()
```

```
{
 int i,j,k,score;
 int scores[LIMIT][MAX];
 for (i = 0; i < LIMIT; i++) {
       j = 0;
       while (j < MAX-1) {
  printf("Введите следующее значение #%d: ",j);
      scanf("%d", score);
      if (score < 0)
         continue;
      scores[i][++j] = score;
    }
    scores[i][0] = j;
 }
}
```

Когда выполняется оператор continue, программа пропускает остаток цикла и начинает цикл сначала. В результате эта программа работает иначе, чем предыдущая. При вводе пользователем числа -1 считается, что была сделана ошибка, и вместо выхода из внутреннего цикла цикл while начинается сначала. Поскольку значение j не было увеличено, программа снова просит ввести то же значение.

4.6.4. *Оператор goto*

Формат оператора goto: goto метка, где «метка» – любой идентификатор, связанный с определенным выражением. Однако наиболее разумное решение при программировании на Си – обойтись без использования оператора goto. Для этого предусмотрено три оператора цикла. Подумайте внимательно, прежде чем использовать оператор goto, действительно ли он вам нужен в создавшейся ситуации и, может быть, его можно заменить на оператор цикла?

1. Какая операция увеличивает значение операнда на 1? **2.** Какая логическая операция имеет наивысший приоритет? **3.** Запишите конструкцию оператора цикла с предусловием. **4.** Может ли оператор цикла for быть вложен-

ным? **5.** Даны три произвольных числа. Используя переменную логического типа, выведите True, если можно построить треугольник с такими длинами сторон, и False – в противном случае. **6.** Используя логическую переменную, выведите True или False в зависимости от того, имеют ли три заданных целых числа одинаковую четность или нет. **7.** Даны произвольные числа a, b и c. Если треугольник с такими длинами сторон нельзя построить, то напечатать 0, иначе напечатать 3, 2 или 1 в зависимости от того, какой это треугольник: равносторонний, равнобедренный и т.д. **8.** Вычислите: $S(x) = \sum_{1}^{n} x^n / n!$ **9.** Используя оператор цикла и условный оператор, составьте программу расчета значений функции Y при изменении аргумента от a до b с шагом h.

$y = \begin{cases} t^2 + 1 \\ \cos t \\ e^t \cdot \cos t \end{cases}$	$t < 1$ $t = 1$ $t > 1$	$a = 0,7$ $b = 1,2$ $h = 0,1$

Глава 5
СТРУКТУРА ПРОГРАММ

5.1. Структура Си-программы

Си предоставляет необычно высокую гибкость для физической организации программы или программных систем. Заметим, что обычно (но не обязательно) первой по порядку в тексте программы функцией является функция main. Рассмотрим типичную организацию программы на Си:

```
/*Заголовки и комментарии, описывающие программу*/

/*Директивы include*/
#include имя_файла_1
...
#include имя_файла_n

/*Макро*/
```

```
#define макро_1 значение_1
...
#define макро_n значение_n

/*Описание глобальных переменных*/
тип_данных глобальная_переменная_1;
...
тип_данных глобальная_переменная_n;

main()
{
/*Описания extern, обеспечивающие ссылку вперед на функции
 и используемые в теле функции main*/

/*Описания локальных переменных*/
тип_данных локальная_переменная_1;
...
тип_данных локальная_переменная_m;

/*Тело функции main*/
...
}

/*Функции, используемые в программе main*/

Тип_данных имя_функции_1 (формальные параметры)
{
/*Описания extern, обеспечивающие ссылку вперед на функции
 и используемые в теле данной программы*/

/*Описания локальных переменных*/

тип_данных локальная_переменная_1;
...
тип_данных локальная_переменная_u;

/Тело функции – 1*/
...
}
...
```

```
Тип_данных имя_функции_n (формальные параметры)
{
/*Описания extern, обеспечивающие ссылку вперед на функции
 и используемые в теле данной функции n */

/*Описания локальных переменных*/
тип_данных локальная_переменная_1;
...
тип_данных локальная_переменная_r;

/*Тело функции n */
...
}
```

Структура каждой функции совпадает со структурой главной программы (main), поэтому функции иногда еще называются п о д п р о г р а м м а м и . Подпрограммы решают небольшую и специфическую часть общей задачи.

5.2. **Функции**

Процесс разработки программного обеспечения предполагает расчленение сложной задачи на набор более простых задач и заданий. В Си поддерживаются функции как логические единицы (блоки текста программы), служащие для выполнения конкретного задания. Важным аспектом разработки программного обеспечения является функциональная декомпозиция, за последние годы получившая широкое распространение. Большинство современных языков программирования высокого уровня поддерживают функциональную декомпозицию.

Функции имеют нуль или более формальных параметров и возвращают значение скалярного типа, типа void (пусто) или указатель. Значения, задаваемые на входе, при вызове функции должны соответствовать числу или типу формальных параметров в описании функции. Если функция не возвращает значения (т.е. возвращает void), то она служит для того, чтобы изменять свои параметры

(вызывать побочный эффект) или глобальные для функции переменные.

Теоретически каждая функция возвращает некоторое значение. Практически же значения, возвращаемые большинством функций, игнорируются и целое семейство новых определений языка Си позволяет описывать и использовать в языке функции типа void, которые никогда не возвращают значений.

В Си вы можете и описывать, и определять функцию. Когда вы описываете функцию, то даете всем остальным программам (включая главный модуль main) информацию о том, каким образом должно осуществляться обращение к этой функции. Когда вы определяете функцию, вы присваиваете ей имя, по которому к ней будет осуществляться обращение, и указываете, какие конкретно действия она будет выполнять. Рассмотрим программу:

```
/* Описание функций */

void get_parms(float *p1, float *p2);
float get_ratio(float divident, float divisor);
void put_ratio (float quotient);
const float INFINITY = 3.4E+38;

/* Главная (main) функция: стартовая точка программы */

main()
{
float a,b,ratio;
char ch;
 do {
 get_parms(&a,&b);          /* ввод параметров */
 ratio = get_ratio(a,b);     /* вычисление частного */
 put_ratio(ratio);    /* печать выходного результата */
  printf("Нажми 'q' для выхода или любую клавишу для
     продолжения\n");

   } while ((ch = getch()) != 'q');
}
/* конец main */
```

```
/* Определение функций */

void get_parms(float *p1, float *p2)
{
 printf("Введите два числа: ");
 scanf("%f %f", p1, p2);
}
float get_ratio(float divident, float divisor)
{
 if (divisor == 0.0)
   return (INFINITY);
 else
   return(divident / divisor);
}
void put_ratio(float ratio)
{
 if (ratio == INFINITY)
   printf("Внимание! Деление на ноль!\n");
 else
   printf("Результат деления двух чисел: %f\n",ratio);
}
```

5.2.1. *Анализ программы*

Первые три строки программы – описание функций. Они вводятся для того, чтобы описать как тип функций, так и порядок следования их параметров с целью избежания ошибок.

Следующая строка описывает константу с плавающей точкой, которой присваивается имя INFINITY (в соответствии с принятым в Си соглашением имена констант состоят из заглавных букв). Эта константа имеет очень большое положительное значение – наибольшее из допустимых для типа float, и используется как флаг возникновения ситуации деления на ноль (divide-by-zero).

Заметьте, что, несмотря на то, что константа описана здесь, она «видима» внутри всех функций (включая главную функцию).

Далее следует функция main (главная функция), которая является основным телом вашей программы. Каждая программа на Си обязательно содержит функцию с именем main. Когда ваша про-

грамма начинает выполняться, вызывается функция main, и дальнейшее выполнение программы продолжается под ее управлением. Когда выполнение функции с именем main заканчивается, то завершается и вся программа, после чего управление передается интегрированной среде Турбо Си или, если вы выполняли программу непосредственно в DOS, монитору DOS.

Функция main может быть расположена в любом месте программы, наиболее часто – это первая функция программы. Это обусловлено тем, что расположение функции main в начале программы облегчает ее чтение, описание прототипов функций и различных глобальных переменных. Это, в свою очередь, позволяет облегчить поиск и документирование функций во всей программе.

После main следует фактическое определение трех функций, прототипы которых были описаны выше: get_parms, get_ratio и put_ratio. Рассмотрим отдельно каждое из определений.

Функция get_parms не возвращает никакого значения, так как ее тип мы определили как void. Это было сделано в связи с тем, что функция служит лишь для чтения двух значений и сохранения их в некотором месте. В каком? Сейчас поясним. Итак, мы передаем в get_parms два параметра. Эти параметры есть адреса, по которым должны быть размещены считанные функцией значения. Обратите внимание! Оба параметра не являются данными типа float, но являются указателями на переменные типа float. Другими словами, мы работаем непосредственно с адресами памяти, по которым размещены переменные типа float.

В соответствии с этим и организовано обращение к функции из main: когда мы вызываем get_parms из main, параметрами функции являются &a и &b (адреса), а не текущие значения a и b. Заметьте также, что вся информация, используемая при обращении к функции scanf, находится непосредственно внутри функции get_parms и перед параметрами p1 и p2 не стоят знаки операций адресации. Почему? Потому, что p1 и p2 уже сами по себе адреса; они являются адресами переменных a и b.

Функция get_ratio возвращает результат (типа float) обработки

двух значений типа float, передаваемых ей (divident и divisor). Возвращенный функцией результат зависит от того, равно значение переменной divisor нулю или нет. Если значение divisor равно нулю, get_ratio возвращает константу INFINITY, если нет – действительное значение частного двух чисел. Обратите внимание на вид оператора return.

Функция put_ratio не возвращает значение, так как ее тип объявлен как void. Она имеет ровно один параметр – ratio, который используется для того, чтобы определить, какое именно сообщение следует выводить на экран дисплея. Если ratio равно INFINITY, то значение частного двух чисел считается неопределенным, в противном случае значение ratio выводится на экран дисплея как результат работы программы.

5.2.2. *Использование классов памяти при организации доступа к данным*

Константы, типы данных и переменные, описанные вне функций (включая main), считаются глобальными ниже точки описания. Это позволяет использовать их внутри функций в пределах всей программы вследствие того, что они уже описаны и известны во всех функциях ниже точки описания. Если вы переместите описание INFINITY в функцию main, то компилятор выдаст сообщение о том, что им обнаружены две ошибки: одна из них – в get_ratio, а другая – в put_ratio. Причина ошибок – использование неописанного идентификатора.

5.2.3. *Описание функций*

Вы можете использовать два различных стиля описания функций – классический и современный. Классический стиль, который нашел широкое применение в большинстве программ на Си, имеет следующий формат:

```
тип имя_функции() ;
```

Эта спецификация описывает имя функции ("имя_функции") и тип возвращаемых ею значений ("тип"). Это описание не содержит никакой информации о параметрах функции, однако это не вызовет обнаружения ошибки компилятором или преобразования типов к типу, уже принятому контекстно, в соответствии с принятыми соглашениями о преобразовании типов. Если вы перепишете описания функций в предыдущей программе, используя этот стиль, то вновь полученные описания приобретут вид:

```
void  get_parms();
float get_ratio();
void  put_ratio();
```

Современный стиль используется в конструкциях расширенной версии Си, предложенной ANSI. При описании функций в этой версии Си используются специальные средства языка, известные под названием «прототип функции». Описание функции с использованием ее прототипа содержит дополнительно информацию о ее параметрах:

```
тип имя_функции(пар_инф1,пар_инф2,...);
```

где параметр пар_инф1 имеет один из следующих форматов:

```
тип
тип имя_пар
...
```

Другими словами, при использовании прототипа функции должен быть описан тип каждого формального параметра либо указано его имя. Если функция использует переменный список параметров, то после указания последнего параметра функции в описании необходимо использовать эллипсис (...).

Подход к описанию функций с помощью описания ее прототипа дает возможность компилятору производить проверку на соответствие количества и типа параметров при каждом обращении к

функции. Это также позволяет компилятору выполнять по возможности необходимые преобразования.

В начальной версии языка Си каждая функция возвращала значение некоторого типа; если же тип не был описан, то по умолчанию функции присваивался тип int. Подобно этому функция, возвращающая «сгенерированные» (нетипичные) указатели, обычно описывалась как возвращающая указатель типа char только потому, что она должна была хоть что-то возвращать.

В Си существует стандартный тип void, который представляется как разновидность «нулевого» типа. Любая функция, не возвращающая явно какое-либо значение, может быть объявлена как функция типа void. Заметим, что большинство программ, использующих динамическое распределение памяти (например malloc), описываются как имеющие тип void. Это означает, что они возвращают нетипизированный указатель, значение которого вы затем можете присвоить указателю любого типа данных без предварительного преобразования типов (хотя преобразования типов лучше использовать повсеместно, для сохранения совместимости).

5.2.4. *Определение функций*

Так же, как и в описании функций, при определении функций прослеживается два стиля – классический и современный.

Классический формат определения функций имеет следующий вид:

```
тип имя_функции(имена_параметров)
определение параметров;
{
  локальные описания;
  операторы;
}
```

Формат описания в современном стиле предусматривает определение параметров функции в скобках, следующих за «имя_функции»:

тип имя_функции(пар_инф, пар_инф, ...)

Однако в этом примере определение параметра «пар_инф» содержит всю информацию о передаваемом параметре: тип и идентификатор. Это позволяет рассматривать первую строку определения функции как часть соответствующего прототипа функции за одним важным исключением: эта строка определения функции не содержит точку с запятой (;) в определении, тогда как прототип функции всегда заканчивается точкой с запятой. Например, определение функции get_parms в классическом стиле выглядит следующим образом:

```
void get_parms(p1, p2)
float *p1; float *p2;
{ ... }
```

Но приверженец современного стиля программирования на Си опишет эту же функцию иначе, с использованием прототипа:

```
void get_parms(float *p1, float *p2)
{ ... }
```

Заметьте, что ряд описаний (константы, типы данных, переменные), содержащихся внутри функции (исключение составляет главная функция main), «видимы» или определены только внутри этой функции. Поэтому Си не поддерживает вложенность функций, т.е. вы не можете объявить одну функцию внутри другой.

Функции могут быть размещены в программе в различном порядке и считаются глобальными для всей программы, включая встроенные функции, описанные до их использования. Старайтесь корректно использовать функции, которые вами еще не определены и не описаны. Когда компилятор обнаружит функцию, которую прежде он не встречал, он определит тип значений, возвращаемый функцией как int.

Если вы ранее определили тип возвращаемых ею значений как, например, char*, то компилятор выдаст ошибку несоответствия типов данных.

5.3. **Типизация функций**

В Си описание extern, заданное внутри функции, имеет действие в пределах данного блока. Описание не будет распознаваться вне блока, в котором оно определено. Однако Си будет запоминать описания, для того чтобы сопоставить их с последующими описаниями тех же самых объектов.

Турбо Си поддерживает большинство предложенных ANSI расширений, в частности дополнительные модификаторы функций и прототипы функций. Си поддерживает также несколько собственных расширений и определений функций, таких как функции типа interrupt (прерывание).

Си, в дополнение к external и static, поддерживает ряд модификаторов, специфицирующих описания функций pascal, cdecl, interrupt, near, far и huge.

5.3.1. *Модификатор функции pascal*

Модификатор pascal, используемый в Турбо Си, предназначен для функций (и указателей на функции), которые используют принятую в Паскале последовательность передачи параметров. Это позволяет писать на языке Си функции, которые могут быть вызваны из программ, написанных на других языках, а также обращаться из ваших Си-программ к внешним программам, написанным на языках, отличных от Си. Для правильной работы компоновщика имя функции преобразуется к верхнему регистру.

Примечание: использование опции компилятора -p (Calling con vention...Pascal) будет вызывать обращение со всеми функциями (и указателями на эти функции), как если бы они были типа pascal. Кроме того, функции, объявленные типа pascal, могут вызываться из Си-программ, также как и Си-программа может быть вызвана из функции, имеющей тип pascal. Например, если вы объявили и откомпилировали функцию

```
pascal putnums(unt i, int j, int k)
{
```

```
 printf("And the answers are: %d, %d и %d\n",i,j,k);
}
```

то другая Си-программа может затем компоноваться с ней и обращаться к ней, используя описание:

```
pascal putnums(int i, int j, int k);
main()
{
 putnums(1,4,9);
}
```

Функции типа pascal не могут иметь различное число аргументов, как, например, функция printf. По этой причине вы не можете использовать эллипсис (...) (т.е. опускать подразумеваемый параметр) в определении функции типа pascal.

5.3.2. *Модификатор функции cdecl*

Модификатор cdecl также является специфичным для Турбо Си. Как и модификатор pascal, он используется с функциями или указателями на функции. Его действие отменяет директиву компилятора –p и разрешает вызывать функции как обычные функции Си. Например, если вы при компиляции вышеприведенной программы установили опцию -p, но захотели использовать printf, то должны поступить следующим образом:

```
extern cdecl printf();
putnums(int i, int j, int k);
cdecl main()
{
  putnums(1,4,9);
}
putnums(int i, int j, int k)
{
  printf("And the answers are: %d, %d и %d\n",i,j,k);
}
```

Если программа компилируется с опцией -p, то все функции, используемые из библиотеки времени выполнения, необходимо объявлять как cdecl. Если вы посмотрите файлы заголовков (такие как STDIO.H), то увидите, что каждая функция явно описана как cdecl, т.е. заранее подготовлена к этому. Заметьте, что главная программа main должна быть также объявлена как cdecl, поскольку действие стартового кода программы, написанного на Си, всегда начинается с вызова модуля main.

5.3.3. *Модификатор функции interrupt*

Модификатор interrupt также является специфическим для Турбо Си. Функции interrupt специально введены для использования с векторами прерываний процессоров типа 8086/8088. Турбо Си будет компилировать функцию типа interrupt в дополнительную функцию, вход и выход которой сохраняется в регистрах AX, BX, CX, DX, SI, DI, ES и DS. Другие регистры: BP, SP, SS, CS и IP – сохраняются как часть последовательности Си вызова или как часть самой обработки прерывания. Рассмотрим пример типичного определения функции типа interrupt:

```
void interrupt myhandler()
{
. . .
}
```

Вы можете объявлять функции прерываний как функции типа void. Функции прерываний могут объявляться для любой модели памяти. Для всех моделей, исключая huge, в регистр DS заносится программный сегмент данных. Для модели huge в DS заносится модульный сегмент данных.

5.3.4. *Прототипы функций*

При объявлении функций в K&R допускается только указание ее имени, типа и скобок без параметров. Параметры (если они есть) объявляются только во время явного определения самой функции.

ANSI стандарт и Турбо Си разрешают вам использовать прототипы функций для объявления функции. Эти объявления включают информацию о параметрах функции. Компилятор использует данную информацию для проверки вызовов функций на соответствие данных, а также для преобразования аргументов к требуемому типу. Рассмотрим следующий фрагмент программы:

```
long lmax(long v1, long v2);
main()
{
  int limit = 32;
  char ch  = 'A';
  long mval;
  mval = lmax(limit,ch);
}
```

Используя прототип функции для lmax, эта программа будет преобразовывать параметры limit и ch к типу long, используя стандартные правила преобразования, прежде чем они будут помещены в стек для обращения к lmax. При отсутствии прототипа функции параметры limit и ch были бы помещены в стек, соответственно, как целое значение и символ; в этом случае в lmax передавались бы параметры, не совпадающие по размеру и содержанию с ожидаемыми. Это ведет к возникновению проблем. В то время как Си в K&R не выполняет никакого контроля типа параметров или их числа, использование прототипов функций очень помогает выявлять «жучки» и другие ошибки программистов.

Прототипы функций также помогают при документировании программ. Например, функция strcpy имеет два параметра: исходную строку и выходную строку. Вопрос – как их распознать? Прототип функции

```
char *strcpy(char *dest, char *source);
```

делает это ясным. Если в файле заголовка имеются прототипы функций, то вы можете распечатать этот файл и получить большую

информацию, необходимую для написания программы, вызывающей эти функции.

Описание функции с включенным в скобки единственным словом void означает, что функция совсем не имеет аргументов:

```
f(void).
```

В противном случае в скобках указывается список параметров, разделенных запятыми. Так, объявление может быть сделано в виде

```
func(int *, long);
```

или с включением идентификаторов:

```
func(int *count, long total);
```

В обоих случаях, указанных выше, функция func принимает два параметра: указатель на тип int, названный count, и целую переменную типа long, названную total. Включение идентификатора в объявление имеет смысл лишь для вывода его в диагностическом сообщении, в случае возникновения несоответствия типа параметров.

Прототип обычно определяет функцию как функцию, принимающую фиксированное число параметров. Для Си-функций, принимающих различное число параметров (таких, как printf), прототип функции может заканчиваться эллипсисом (...), например:

```
f(int *count, long total,...).
```

У прототипов такого вида фиксированные параметры проверяются во время компиляции, а опущенные передаются, как при отсутствии прототипа.

Вот несколько примеров объявления функций и прототипов.

```
int f();   /* Функция возвращает величину типа int без */
    /* информации о параметрах. Это классический */
            /* стиль K&R */
```

```
int f(void);  /* Функция возвращает значение типа int, */
              /* параметры не передаются */

int p(int,long); /* Функция возвращает целое, а получает */
                 /* два параметра; */
          /* первый имеет тип int, второй – long */
int pascal q(void); /* Функция типа pascal, возвращающая */
           /* целое; параметры не передаются */
char far * s(char *source, int kind); /* Функция возвра- */
           /* щает указатель типа far на строку,*/
    /*а получает два параметра: типа char* и типа int */
int printf(char *format,...);   /* Функция возвращает */
         /* значение типа int, получая указатель */
          /* на фиксированный параметр типа char */
     /* и любое количество дополнительных параметров */
                  /* неизвестного типа */
int (*fp)(int);  /* Указатель на функцию, возвращающую */
           /* целое и получающую единственный */
                 /* целый параметр */
```

Ниже приводятся правила, регламентирующие работу с модификаторами языка и формальными параметрами в вызовах функций Си, как использующих прототипы, так и не использующих их.

Правило 1. Модификаторы языка для описания функций должны совпадать с модификаторами, используемыми в объявлении функции, для всех обращений к функции.

Правило 2. Функция может изменять значения формальных параметров, но это не оказывает какого-либо воздействия на значения действительных аргументов в вызывающей программе, за исключением функций прерывания.

Если прототип функции не объявлен предварительно, то Си преобразует целочисленные аргументы при обращении к функции согласно правилам, приведенным в разделе «Арифметические преобразования». Если прототип объявлен, то Си преобразует заданные аргументы к типу, назначенному для параметров.

Если прототип функции включает эллипсис (...), то Си преобразует все заданные аргументы функции, как в любом другом прототипе (до эллипсиса). Компилятор будет расширять любые аргументы, заданные после фиксированного числа параметров, по нормальным правилам для аргументов функций без прототипов.

Если есть прототип, то число аргументов должно быть соответственным (за исключением случая, когда в прототипе использован эллипсис). Типы должны быть совместимы только по размеру, для того чтобы корректно производились преобразования типов. Вы всегда должны использовать явные преобразования аргументов к типу, допустимому для прототипа функции.

Следующий пример прояснит данные замечания:

```
int strcmp(char *s1, char *s2);/* Полный прототип */
int *strcpy();        /*Прототип не определен*/
int samp1(float, int, ...);  /* Полный прототип */
samp2()
{
char *sx, *cp;
double z;
long a;
float q;
if (strcmp(sx, cp))      /* 1. Верно */
  strcpy(sx, cp, 44);   /* 2. Верно, но не перено- */
            /* симо из Турбо Си */
samp1(3, a, q);       /* 3. Корректно */
strcpy(cp);          /* 4. Ошибка при выполнении */
samp1(2);            /* 5. Ошибка при компиляции */
}
```

Пять вызовов (каждый с комментарием) примера демонстрируют различные варианты вызовов функций и прототипов.

В вызове 1 использование функции strcmp явно соответствует прототипу, что справедливо для всех случаев.

В вызове 2 обращение к strcpy имеет лишний аргумент (strcpy определена для двух аргументов, а не для трех). В этом случае Си теряет небольшое количество времени и создает код для помещения лишнего аргумента в стек. Это, однако, не является синтаксической ошибкой, поскольку компилятор не знает о числе аргументов strcpy. Такой вызов недопустим для других компиляторов.

В вызове 3 прототип требует, чтобы первый аргумент для samp1 был преобразован к float, а второй – к int. Компилятор выдаст предупреждение о возможной потере значащих цифр, поскольку при преобразовании типа long к типу int отбрасываются старшие биты. (Вы можете избавиться от такого предупреждения, если зададите явное преобразование к целому.) Третий аргумент, q, соответствует эллипсису в прототипе. Он преобразуется к типу double согласно обычному арифметическому преобразованию. Вызов полностью корректен.

В вызове 4 снова вызывается strcpy, но число аргументов слишком мало. Это вызовет ошибку при выполнении программы. Компилятор будет молчать (если даже число параметров отличается от числа параметров в предыдущем вызове той же функции), так как для strcpy не определен прототип функции.

В вызове 5 функции samp1 задано слишком мало аргументов. Поскольку samp1 требует минимум два аргумента, этот оператор является ошибочным. Компилятор выдаст сообщение о том, что в вызове не хватает аргументов.

Важное замечание: если ваш прототип функции не соответствует действительному определению функции, то Си обнаружит это в том и только в том случае, если это определение находится в том же файле, что и прототип. Если вы создаете библиотеку программ с передаваемым набором прототипов (файлом include), то вы должны учитывать включение файла с прототипами во время компиляции библиотеки с целью выявления любого противоречия между прототипами и действительными определениями функций.

5.4. **Механизмы передачи параметров**

В Си параметры функции передаются по значению, по ссылке и по указателю. При передаче параметра по значению в функции создается локальная копия, что приводит к увеличению объема используемой памяти. При вызове функции в стеке отводится память для локальных копий параметров, а при выходе из функции эта память освобождается. Рассмотренный способ использования памяти не только требует дополнительного пространства, но и отнимает часть времени счета.

Рассмотрим пример программы, демонстрирующей, что при вызове функции копии создаются для параметров, передаваемых по значению, а для параметров, передаваемых по ссылке, – не создаются:

```
#include <stdio.h>
void test_function (int first, int second, int *third)
{
printf("\nАдрес first равен %p", &first);
printf("\nАдрес second равен %p", &second);
printf("\nАдрес third равен %p", third);
*third +=1;
}
main()
{ extern void test_function (int first, int second, int *third);
 int a,b;
 int c=24;
 printf("\nАдрес a равен %p", &a);
 printf("\nАдрес b равен %p", &b);
 printf("\nАдрес c равен %p", &c);
 test_function (a,b,&c);
 printf ("\nЗначение c =%d\n", c);
}
```

У функции test_function параметры first и second передаются по значению, параметр third передается по ссылке. Термин «ссылка» здесь означает ссылку на область памяти. Поскольку параметр third

является указателем на тип int, то он, как и все параметры типа указатель и массивы, передается по ссылке.

После того как выражение *third в теле функции test_function увеличивается на единицу, новое значение присваивается переменной c, память под которую отведена в функции main. Так получается значение 25, являющееся результатом работы функции.

Программа напечатает:

```
Адрес a равен FEC6
Адрес b равен FEC8
Адрес c равен FECA
Адрес first равен FECO
Адрес second равен FEC2
Адрес third равен FECA
Значение c = 25
```

В качестве параметров функции можно использовать выражения:

```
#include <stdio.h>
void function_test (int a)
{
printf("\na=%d", a);
}
main()
{ extern void function_test (int a);
 int c=3,
   d=4,
   e=5,
   f=6;
 function_test (f= (c=d*e)+15);
 printf ("\n\nc =%d\n", c);
}
```

При задании массива в качестве параметра функции передается адрес первого элемента массива. Если в теле функции заменяются значения элементов массива, то изменяется непосредственно сам передаваемый массив.

Если в описании функции задано, что параметр передается по ссылке (т.е. он описан как указатель на тип), то в качестве параметра при вызове функции передается адрес переменной.

5.5. **Понятие рекурсии**

Если функция вызывает саму себя, то говорят, что возникает рекурсия. Рекурсию можно рассматривать просто как еще одну управляющую структуру – управление из точки рекурсивного вызова передается на начало функции. В действительности же рекурсия – это мощный инструмент для разработки программ, с помощью которого даже большие объемы действий могут быть записаны всего несколькими строками программного кода.

Рекурсией следует пользоваться осторожно и внимательно. При использовании рекурсивного вызова система сохранит в стеке значения всех автоматических переменных функции и ее параметров. По завершении рекурсивного вызова значения будут восстановлены и управление возвратится на оператор, стоящий непосредственно за оператором вызова. Для размещения в стеке автоматических переменных и значений параметров необходимы и память, и время счета.

Рассмотрим пример простого типа рекурсии:

```
#include <stdio.h>
main()
{ extern void write_name (char *name, int count);
 write_name ("Си удобен и эффективен",3);
 printf ("\n");
}
void write_name (char *name, int count)
{
if (count>0)
{
 printf("\n%s", name);
```

```
    write_name (name, count – 1);
    }
}
```

Рекурсивный вызов записан в последней строке функции write_ name. При возврате из рекурсивного вызова никаких действий в программе больше не выполняется. Такой тип рекурсии практически эквивалентен итерации. Использование параметра count функции write_name, передаваемого по значению, ведет к значительному расходу памяти. При каждом рекурсивном вызове текущее значение параметра count заносится в стек.

Программа выведет следующий текст:

```
Си удобен и эффективен
Си удобен и эффективен
Си удобен и эффективен
```

1. Могут ли быть вложенными определения функций? **2.** Для чего используется оператор return? **3.** По каким параметрам должны соответствовать друг другу формальные и фактические параметры? **4.** Чем отличается параметр-ссылка от параметра-указателя? **5.** Используя функцию для вычисления факториала, определите значение выражения: $z = \dfrac{(n+m)!}{n!+m!}$, где $n! = \begin{cases} 0, & \text{если } n < 0; \\ 1, & \text{если } n = 0; \\ n!, & \text{если } n > 0. \end{cases}$ **6.** Используя функцию для вычисления гиперболического тангенса, вычислите значение выражения: $z = \dfrac{th(x+a) - th(x+b)}{th(x-a)}$. **7.** Используя функции для нахождения максимального и минимального значений, по заданным 10-элементным вещественным массивам *A*, *B*, *C* вычислите: $y = \begin{cases} \dfrac{\max(C)}{\min(A)} + \dfrac{\min(B)}{\max(A)}, \text{если } \max(C) > \min(A); \\ \text{иначе, } \min(B) + \max(A). \end{cases}$

Глава 6

СТРУКТУРИЗАЦИЯ ДАННЫХ

6.1. **Указатели**

Переменные содержат данные, т.е. текущую информацию для работы вашей программы. Но иногда важнее знать место расположения данных, чем собственно их значение. Именно для этого и используются указатели.

Компьютер содержит в своей памяти (часто называемой RAM – Random Access Memory – память произвольного доступа) вашу программу и совокупность данных. На самом нижнем уровне память вашего компьютера состоит из битов – мельчайших электронных схем, которые могут «запомнить» (пока компьютер включен) одно из двух значений, обычно интерпретируемое как «0» и «1».

Восемь бит группируются в 1 байт. Большим группам битов, как правило, присваивается имя: обычно 2 байта составляют СЛОВО, 4 байта составляют ДЛИННОЕ СЛОВО; и для IBM PC 16 байтов составляют ПАРАГРАФ.

Каждый байт в памяти вашего компьютера имеет собственный уникальный адрес, так же, как каждый дом на любой улице. Но, в отличие от большинства домов, последовательные байты имеют последовательные адреса: если данный байт имеет адрес N, то предыдущий байт имеет адрес N-1, а следующий – N+1.

УКАЗАТЕЛЬ – это переменная, содержащая адрес некоторых данных, а не их значение. Зачем это нужно?

Во-первых, мы можем использовать указатель места расположения различных данных и различных структур данных. Изменением адреса, содержащегося в указателе, вы можете манипулировать (создавать, считывать, изменять) информацией в различных ячейках. Это позволит вам, например, связать несколько зависимых структур данных с помощью одного указателя.

Во-вторых, использование указателей позволит вам создавать новые переменные в процессе выполнения программы. Си позволяет вашей программе запрашивать некоторое количество памяти (в

байтах), возвращая адреса, которые можно запомнить в указателе. Этот прием известен как ДИНАМИЧЕСКОЕ РАСПРЕДЕЛЕНИЕ. Используя его, ваша программа может приспосабливаться к любому объему памяти в зависимости от того, как много (или мало) памяти доступно вашему компьютеру.

В-третьих, вы можете использовать указатели для доступа к различным элементам структур данных, таким как массивы, строки или структуры. Указатель, в сущности, указывает место в памяти вашего компьютера (а используя смещение относительно начального адреса, можно указать целый сегмент памяти), в котором размещены те или иные данные. Индексируя указатель, вы получаете доступ к некоторой последовательности байтов, которая может представлять, например, массив или структуру.

Прежде чем использовать указатели, их необходимо объявить. Рассмотрим следующую программу:

```
main()
{
 int ivar,*iptr;
 iptr = &ivar;
 ivar = 421;
 printf("Размещение ivar: %p\n",&ivar);
 printf("Содержимое ivar: %d\n", ivar);
 printf("Содержимое iptr: %p\n", iptr);
 printf("Адресуемое значение: %d\n",*iptr);
}
```

В ней объявлены две переменные: ivar и iptr. Первая переменная ivar – это целая переменная, т.е. содержащая значение типа int. Вторая переменная iptr – это указатель на целую переменную, следовательно, она содержит АДРЕС значения типа int. Можно также сказать, что переменная iptr – это указатель, так как перед ее описанием стоит звездочка (*). В языке Си эта звездочка называется косвенным оператором.

В основном данная программа делает следующее:

– адрес переменной ivar присваивается iptr;

– целое значение 421 присваивается ivar.

Адресный оператор (&) позволяет получить адрес, по которому размещено значение переменной ivar.

Результат работы программы:

- размещение ivar: 166E;
- содержимое ivar: 421;
- содержимое iptr: 166E;
- адресуемое значение: 421.

Первые две строки указывают адрес и содержимое ivar. Третья представляет адрес, содержащийся в iptr. Как видите, это адрес переменной ivar, т.е. место в памяти, где ваша программа решила создать переменную с идентификатором ivar. В последней строке печатается то, что хранится по этому адресу, – те же самые данные, которые уже присвоены переменной ivar.

Заметим, что в третьем обращении к функции printf используется выражение iptr, содержимое которого есть адрес ivar. В последнем обращении к printf используется выражение *iptr, которое позволяет получить данные, хранящиеся по этому адресу.

Рассмотрим теперь небольшую вариацию предыдущей программы:

```
main()
{
  int ivar,*iptr;
  iptr = &ivar;
  *iptr = 421;
 printf("Размещение ivar: %p\n",&ivar);
 printf("Содержимое ivar: %d\n", ivar);
 printf("Содержимое iptr: %p\n", iptr);
 printf("Адресуемое значение: %d\n",*iptr);
}
```

В этой программе также адрес переменной ivar присваивается iptr, но вместо присваивания числа 421 переменной ivar это значение присваивается по указателю *iptr. Каков результат? Точно такой же, как и в предыдущей программе. Почему? Потому что выражения *iptr и ivar есть одна и та же ячейка памяти. В этом случае оба оператора заносят значение 421 в одну и ту же ячейку памяти.

6.1.1. *Динамическое распределение памяти*

В операционной системе MS DOS для микропроцессора семейства 8086 и Турбо Си размер указателя (число байтов, требуемых для размещения адреса памяти под указатель) зависит от модели памяти, задаваемой при компиляции программы. В Турбо Си программы можно компилировать в расчете на шесть моделей памяти: крошечная (tiny), маленькая (small) (модель по умолчанию), средняя (medium), компактная (compact), большая (large) и огромная (huge). Программа, написанная на Турбо Си для машин семейства IBM PC в расчете на конкретную модель памяти, может оказаться не мобильной. Все синтаксические конструкции Турбо Си, поддерживающие шесть перечисленных моделей памяти, являются расширением стандарта Си.

Память для программы на Турбо Си требуется для четырех целей: для размещения ее программного кода, для размещения данных, для динамического использования и для резервирования компилятором на время выполнения программы.

Концептуальная модель этих областей приведена на рис. 6.1.

Старшие адреса памяти				Младшие адреса памяти			
Буфера, видеопамять, ПЗУ	Неиспользуемая о.п.	Стек →	Свободная память	Куча ←	Статические данные	Код программы DOS	Векторы прерываний

Рис. 6.1. Распределение памяти в Турбо Си

Область памяти под код программы в процессе работы остается неизменной. Неизменной остается и память, отводимая под статические данные. Объем памяти для кучи зависит от того, сколько памяти запрашивает программист с помощью функции calloc и malloc (эти функции подробно рассматриваются в настоящей главе ниже). Размер использованной памяти стека изменяется при активизации автоматических (локальных) переменных в функциях, а

также за счет того, что при вызовах функции в стек заносятся параметры функций.

Модели памяти, поддерживаемые в Турбо Си

Крошечная: во все четыре регистра сегментов (CS, DS, SS, ES) за сылается один и тот же адрес. Под код программы, статические данные, динамически размещаемые данные и стек отводится 64 К памяти. Такая модель налагает на задачу серьезные ограничения и используется только в тех случаях, когда особенно ощущается дефицит памяти. Переменные типа указатель в такой модели памяти занимают только 2 байта (близкие указатели). Следовательно, переменные типа указатель содержат смещение внутри фиксированного сегмента памяти.

Маленькая: под код программы отводится сегмент размером 64 К. Стек, куча и статические данные размещаются в одном сегменте размером 64 К. Такая модель памяти принимается по умолчанию и вполне подходит для многих маленьких и средних задач. Переменные типа указатель в такой модели памяти занимают только 2 байта (близкие указатели). Следовательно, переменные типа указатель содержат смещение внутри фиксированного сегмента памяти.

Средняя: размер памяти под код программы ограничен 1 Мбайтом. Это означает, что в коде программы используются далекие указатели. Стек, куча и статические данные, как и в случае маленькой модели памяти, размещаются вместе в сегменте памяти размером 64 К. Такую модель памяти рекомендуется применять при программировании очень больших программ, не использующих большого объема данных. Для адресации (указания) в коде программы служат далекие указатели (сегмент и смещение), занимающие 4 байта. Таким образом, все вызовы функций выполняются как далекие вызовы и все возвраты функций считаются

далекими. Для адресации данных используются близкие указатели, занимающие 2 байта.

Компактная: под код программы отводится 64 К. Под данные отводится 1 Мбайт. Объем статических данных ограничивается 64 К, а размер стека, как и для всех моделей, не может превысить 64 К. Такая модель памяти должна применяться при создании малых и средних по размеру программ, требующих большого объема статических данных. Адресация внутри программы выполняется с помощью близких указателей (их размер – 2 байта). Для адресации данных используются 4-байтовые далекие указатели.

Большая: размер памяти под код программы ограничен 1 Мбайтом. Под статические данные отводится 64 К. Куча может занимать до 1 Мбайта памяти. Такую модель приходится использовать во многих больших задачах. Как программа, так и данные адресуются далекими указателями, занимающими 4 байта. В большой модели памяти ни одна отдельная единица данных не может превышать 64 К.

Огромная: аналогична большой модели, но суммарный объем статических данных может превышать 64 К. Огромная модель памяти не предусматривает огромных указателей.

Изменим еще раз нашу программу:

```
#include <alloc.h>
main()
{
 int *iptr;
 iptr = (int *) malloc(sizeof(int));
 *iptr = 421;
 printf("Содержимое iptr: %p\n", iptr);
 printf("Адресуемое значение: %d\n",*iptr);
}
```

Эта версия позволяет вполне обойтись без описания переменной ivar, которое непременно фигурировало в предыдущих примерах. Вместо адреса переменной iptr присваивается значение (тоже адрес некоторой ячейки памяти), возвращаемое функцией, которая назы-

вается malloc и описана в библиотеке alloc.h (отсюда появление директивы #include в начале программы). Затем по этому адресу присваивается значение 421, и выражение *iptr вновь, как и в предыдущем примере, принимает значение 421.

Обратите внимание, что если вы теперь выполните программу, то получите иное значение iptr, чем раньше, но значение *iptr останется равным 421.

Разберем теперь, что же делает оператор

```
iptr = (int *) malloc(sizeof(int));
```

Разобьем его на части.

Выражение sizeof(int) возвращает количество байтов, требуемое для хранения переменной типа int. Для компилятора Турбо Си, работающего на IBM PC, это возвращаемое значение равно 2.

Функция malloc(num) резервирует num последовательных байтов доступной (свободной) памяти в компьютере, а затем возвращает начальный адрес размещения в памяти этой последовательности байтов.

Выражение (int *) указывает, что этот начальный адрес есть указатель на данные типа int. Выражение такого вида известно как выражение приведения типа (type casting). В данном случае Турбо Си не требует обязательного его применения. Но в связи с тем, что для других компиляторов Си это выражение является обязательным, при его отсутствии вы получите сообщение об ошибке:

" Non-portable pointer assignment."
(Непереносимое в другие системы присваивание указателя)

Из соображений переносимости программного обеспечения лучше всегда предусматривать явное приведение типов в своих программах.

Наконец, адрес, полученный с помощью функции malloc, запоминается в iptr. Таким образом, получена динамически созданная целая переменная, к которой можно обращаться при помощи идентификатора *iptr.

Весь этот оператор можно описать следующим образом: «Выделить в памяти компьютера некоторый участок для переменной типа int, затем присвоить начальный адрес этого участка переменной iptr, являющейся указателем на переменную типа int».

Необходимо ли все это? Да. Почему? Потому что без этого у вас нет гарантии, что iptr указывает на свободный участок памяти. iptr будет содержать некоторое значение, которое вы будете использовать в качестве адреса, но вам не будет известно, не используется ли уже этот раздел памяти для других целей. Правило использования указателей простое: УКАЗАТЕЛЬ ВСЕГДА ДОЛЖЕН ИМЕТЬ АДРЕС ДО СВОЕГО ИСПОЛЬЗОВАНИЯ В ПРОГРАММЕ.

Иными словами, не присваивайте целое значение выражению *iptr без предварительного присвоения адреса в iptr.

6.1.2. *Модификаторы типа указателей: near, far, huge*

Си имеет три модификатора, воздействующих на косвенный оператор (*) и, тем самым, модифицирующих указатели на данные. Речь идет о near, far и huge.

Си позволяет использовать при компиляции одну из нескольких моделей памяти. Модель, которую вы используете, определяет (среди прочих деталей) внутренний формат указателей на данные. Если вы используете малую модель памяти (tiny, small, medium), то все указатели имеют длину только 16 бит и задают смещение относительно регистра сегмента данных (DS). Если вы используете большую модель (compact, large, huge), то все указатели на данные имеют длину 32 бита и задают адрес сегмента и смещение.

Иногда, когда используется модель данных одного размера, у вас может возникнуть желание объявить указатель с размером или форматом другим, нежели у используемого по умолчанию. Вы можете сделать это с помощью модификаторов near, far и huge.

Указатель типа near имеет размер 16 бит; он использует текущее содержимое регистра сегмента данных (DS) для определения адреса сегмента. По умолчанию он используется для малых моделей

данных. При использовании указателей типа near данные вашей программы ограничены размером 64 К текущего сегмента данных.

Указатель типа far имеет размер 32 бита и содержит как адрес, так и смещение. По умолчанию он используется для больших моделей. При использовании указателей типа far допускаются ссылки на данные в пределах адресуемого пространства 1 Мб процессора Intel 8088/8086.

Указатель типа huge имеет размер 32 бита и аналогично предыдущему содержит адрес сегмента и смещение, однако, в отличие от указателей типа far, указатель huge всегда поддерживается нормализованным, т.е.:

– операторы отношения (==, !=, <, >, <=, >=) работают корректно с указателями типа huge, но не с указателями типа far;

– любые арифметические операции над указателем huge воздействуют как на адрес сегмента, так и на смещение (из-за нормализации); при использовании far указателей воздействие распространяется только на смещение;

– заданный указатель типа huge может быть увеличен в пределах 1 Мб адресного пространства; указатель типа far, соответственно, будет циклически переходить на начало сегмента в 64 К;

– при использовании указателей типа huge требуется дополнительное время, так как программы нормализации должны вызываться после выполнения любой арифметической операции над указателями.

6.1.3. *Указатели как формальные параметры функций*

В главе 5 мы рассмотрели, как объявлять параметры функций. Возможно, теперь вам более понятна причина использования указателей в качестве формальных параметров функции, значения которых вы можете изменять.

Рассмотрим, например, следующую функцию:

```
void swap(int *a, int *b)
{
```

```
 int temp;
 temp = *a; *a = *b; *b =temp;
}
```

Эта функция swap (обмен) объявляет два формальных параметра a и b как указатели на некие данные типа int. Это означает, что функция swap работает с адресами целых переменных (а не с их значениями). Поэтому будут обработаны данные, адреса которых переданы функции во время обращения к ней.

Далее представим программу, вызывающую swap:

```
main()
{  extern void swap(int *a, int *b);
  int i,j;
  i = 421;
  j = 53;
 printf("До обращения: i=%4d, j=%4d\n",i,j);
  swap(&i,&j);
 printf("После обращения: i =%4d, j=%4d\n",i,j);
}
```

Вы видите, что эта программа действительно заменяет значение i на значение j (переставляет их местами). Заменим эту программу на аналогичную ей, раскрыв процедуру swap в теле программы:

```
main()
{
  int i,j;
  int *a,*b,temp;
  i = 421;
  j = 53;
 printf("До обработки: i = %4d j = %4d\n",i,j);
  a = &i;
  b = &j;
  temp = *a; *a = *b; *b =temp;
 printf("После обработки: i = %4d j = %4d\n",i,j);
}
```

Эта программа, конечно, приводит к тому же результату, что и предыдущая, поскольку не отличается от нее. При вызове функции swap(&i,&j) значения двух фактических параметров (&i,&j) присваиваются двум формальным параметрам (a и b), обрабатываемым непосредственно операторами функции swap.

6.1.4. *Указатели на функции*

Указатель на функцию может быть описан как параметр и передаваться в функцию наряду с другими данными.

При использовании указателя в вызываемой функции должна быть снята ссылка на передаваемую функцию.

Разрешается определять массивы функций. Указатели на функции могут быть компонентами структур.

Массивы функций удобно применять при реализации систем, управляемых с помощью меню. Каждому пункту меню ставится в соответствие реализующая его функция, указатель на которую помещается в массив. После того как пользователь выбрал из меню интересующее его действие, по индексу, соответствующему такому выбору, из массива выбирается функция, реализующая действие.

Рассмотрим синтаксическую конструкцию, требуемую для описания указателя на функцию. Она выглядит следующим образом:

```
typedef void (*menu_action) ();
```

Здесь идентификатор menu_action определен как указатель на функцию, не имеющую параметров и возвращающую тип void. Если бы определение типа было задано по-другому:

```
typedef void *menu_action ();
```

то значение конструкции было бы совсем другим. В последнем случае идентификатор menu_action задавал бы имя функции, возвращающей указатель на тип void.

Символ ссылки * имеет более низкий приоритет, чем операция (), и поэтому круглые скобки следует использовать для задания то-

го, что идентификатор menu_action является указателем на функцию, возвращающую тип void, а не указателем на тип void.

Теперь рассмотрим еще одно определение типа:

```
typedef float (*integrand) (float r);
```

Здесь сообщено, что integrand – это указатель на функцию с одним параметром типа float, возвращающую значение типа float.

Такое определение типа можно использовать, например, для написания универсальной функции интегрирования integral, заголовок которой имеет следующий вид:

```
float integral (integrand f,float lower_limit,float upper_limit);
```

Первым параметром функции integral является функция f типа integrand. Это означает, что f – указатель на любую функцию от одного вещественного параметра, возвращающую тип float. Если функция my_function определена так:

```
float my_function(float r)
{
return r * r – 3.0;
}
```

то вызов функции integral будет выглядеть следующим образом:

```
float answer=integral(my_function, 0.0, 5.0);
```

Имя функции my_function выступает в качестве указателя на эту функцию, и поэтому не требуется указания ссылки.

Рассмотрим пример программы, управляемой с помощью меню на основе массива указателей функции:

```
#include <stdio.h>
#include <conio.h>
 typedef void(*menu_action) ();
menu_action control[6];
/*
```

Внимание: управляющая таблица может быть инициализирована статически.

```
*/
main()
{
 extern void build_table (void);
 int choice;
 build_table();
 do
 {
 clrscr();
 printf("\n1 -> Пункт 1");
 printf("\n2 -> Пункт 2");
 printf("\n3 -> Пункт 3");
 printf("\n4 -> Пункт 4");
 printf("\n5 -> Пункт 5");
 printf("\n6 -> Пункт 6");
 printf("\n7 -> Выход из программы");
 printf("\n\n Введите номер пункта:");
 scanf("%d", &choice);
 if (choice >=1 && choice <=6)
  (*contol[choice -1]) ();
 else
  break;
 }
 while (1);   /* Бесконечный цикл */
 printf("\n");
 }
 void build_table(void)
 {
 extern void menu_item_1(void);
 extern void menu_item_2(void);
 extern void menu_item_3(void);
extern void menu_item_4(void);
extern void menu_item_5(void);
extern void menu_item_6(void);
control[0]=menu_item_1;
control[1]=menu_item_2;
control[2]=menu_item_3;
```

```
 control[3]=menu_item_4;
 control[4]=menu_item_5;
 control[5]=menu_item_6;
}
 void menu_item_1(void)
{
 printf("\nВыполнилось действие по пункту 1 меню");
}
 void menu_item_2(void)
{
 printf("\nВыполнилось действие по пункту 2 меню");
}
 void menu_item_3(void)
{
 printf("\nВыполнилось действие по пункту 3 меню");
}
 void menu_item_4(void)
{
 printf("\nВыполнилось действие по пункту 4 меню");
}
 void menu_item_5(void)
{
 printf("\nВыполнилось действие по пункту 5 меню");
}
 void menu_item_6(void)
{
 printf("\nВыполнилось действие по пункту 6 меню");
}
```

Тип menu_action является указателем на функцию, возвращаемую тип void и не имеющую параметров.

С помощью оператора

```
menu_action control[6];
```

задается массив из шести указателей. В теле функции build_table каждому элементу массива присваивается значение указателя на соответствующую функцию. В нашем примере каждая из таких функций печатает простое сообщение.

В функции main выводится меню, и пользователю предлагается либо выбрать один из пунктов меню, либо завершить работу программы. Выбор, сделанный пользователем, активизирует одну из шести функций при помощи оператора:

```
(*control[choice – 1]) ();
```

Приведенный прием является удобным при разработке больших программ, управляемых с помощью меню. Добавление в такую программу новых возможностей требует лишь включения в массив указателей программы build_table имен новых функций.

6.1.5. *Адресная арифметика в Си*

Каким образом вам необходимо поступить, если вы хотите, чтобы переменная-указатель указывала на три переменных некоторого типа вместо одной? Рассмотрим одно из возможных решений:

```
#include <alloc.h>
main()
{
  #define NUMINTS 3
  int *list,i;
  list = (int *) calloc(NUMINTS,sizeof(int));
  *list = 421;
  *(list+1) = 53;
  *(list+2) = 1806;
  printf("Список адресов :");
  for (i=0; i<NUMINTS; i++)
     printf("%4p ",(list+i));
  printf("\nСписок значений :");
  for (i=0; i<NUMINTS; i++)
     printf("%4d ",*(list+i));
  printf("\n");
}
```

Вместо функции malloc эта программа использует функцию calloc с двумя параметрами: первый показывает, для скольких эле-

ментов будет происходить резервирование памяти, второй – величину каждого элемента в байтах. После обращения к функции calloc list указывает на участок памяти размером 6 (3*2) байтов, достаточный для хранения трех переменных типа int.

Более подробно рассмотрим следующие три оператора. Первый оператор вам знаком – *list=421. Он означает: «запомнить 421 в переменной типа int, расположенной по адресу, хранящемуся в переменной list».

Следующий оператор: *(list+1)=53 – особенно важен для понимания. На первый взгляд, его можно интерпретировать так: «запомнить 53 в переменной типа int, расположенной байтом дальше адреса, хранящегося в list». Если это так, то вы окажетесь в середине предыдущей int-переменной (которая имеет длину 2 байта). Это, естественно, испортит ее значение.

Однако ваш компилятор с Си не сделает такой ошибки. Он «понимает», что list – это указатель на тип int, и поэтому выражение list+1 представляет собой адрес байта, определенного выражением list+(1*sizeof(int)), и поэтому значение 53 не испортит значения 421 (т.е. для этого значения будет выделена другая ячейка памяти).

Аналогично, *(list+2)=1806 представляет адрес байта list+(2* sizeof(int)) и 1806 запоминается, не затрагивая двух предыдущих значений.

В общем, ptr+i для типа int представляет адрес памяти, определяемый выражением ptr+(i*sizeof(int)).

Результат работы программы:

– список адресов: 066A 066C 066E;

– список значений: 421 53 1806.

Заметьте, что адреса различаются не в один байт, а в два, и все три значения хранятся отдельно.

Подведем итог: если вы используете ptr, указатель на тип type, то выражение (ptr+1) представляет адрес памяти (ptr+(1*sizeof (type)), где sizeof(type) возвращает количество байтов, занимаемых переменной типа type. Сложение или вычитание указателей всегда выполняется в единицах того типа, к которому относится указатель.

Список допустимых действий над указателями приведен в табл. 6.1.

Таблица 6.1

Операции над указателями

Операция	Пояснение
ptr1 == ptr2	Сравнение на равенство
ptr1 != ptr2	Сравнение на неравенство
ptr1 < ptr2	Сравнение на меньше
ptr1 <= ptr2	Сравнение на меньше или равно
ptr1 > ptr2	Сравнение на больше
ptr1 >= ptr2	Сравнение на больше или равно
ptr2 – ptr1	Вычисление числа элементов между ptr2 и ptr1
ptr1+int_val	Вычисление указателя, отстоящего от ptr1 вверх
ptr1-int_val	Вычисление указателя, отстоящего от ptr1 вниз на int_val элементов

6.1.6. *Использование неинициализированных указателей*

Серьезная опасность таится в присвоении значения по адресу, содержащемуся в указателе, без первоначального присвоения адреса этому указателю, например:

```
main()
{
 int *iptr;
 *iptr = 421;
 printf("*iptr = %d\n",*iptr);
}
```

Эта ловушка опасна тем, что программа, содержащая ее, может быть «верна» и компилятор может не выдать никаких сообщений во время компиляции такой программы. В примере, указанном вы-

ше, указатель iptr имеет некоторый произвольный адрес, по которому запоминается значение 421. Эта программа настолько мала, что шанс что-нибудь затереть в памяти с ее помощью ничтожно мал, однако в больших программах возрастает вероятность разрушения других данных, поскольку вполне возможно, что по адресу iptr уже хранится другая информация. Если вы используете модель самой маленькой памяти (tiny), в которой сегменты программы и данных занимают одну и ту же область памяти, то вы подвергаете себя риску испортить свой же загрузочный модуль. Поэтому старайтесь внимательно писать программы, использующие указатели.

6.2. **Массивы и их реализация**

6.2.1. *Описание массивов*

Большинство языков высокого уровня, включая Си, позволяют определять МАССИВЫ, т.е. индексированный набор данных определенного типа.

Рассмотрим программу:

```
main()
{
  #define NUMINTS 3
  int list[NUMINTS],i;

  list[0] = 421;
  list[1] = 53;
  list[2] = 1806;
  printf("Список адресов:");
  for(i=0; i<NUMINTS; i++)
    printf("%p ",&list[i]);
  printf("\nСписок значений:");
  for (i=0; i<NUMINTS; i++)
    printf("%4d ",list[i]);
  printf("\n");
}
```

Выражение int list[NUMINTS] объявляет list как массив переменных типа int с объемом памяти, выделяемым для трех целых переменных.

К первой переменной массива можно обращаться как к list[0], ко второй – как к list[1] и к третьей – как к list[2].

В общем случае описание любого массива имеет следующий вид:

type name[size];
(тип имя [размер]);

где type – тип данных элементов массива (любой из допустимых в языке), name – имя массива.

Первый элемент массива – это name[0], последний элемент – name[size-1]; общий объем памяти в байтах определяется выражением size*(sizeof(type)).

Описание: type *name[size] определяет массив name из size указателей на тип type.

Другое описание: type (*name)[size] определяет указатель name на массив из size элементов типа type.

Вы, наверное, уже поняли, что существует определенная связь между массивами и указателями. Поэтому если вы выполните только что рассмотренную программу, полученный результат будет вам уже знаком:

– список адресов: 163A 163C 163E;
– список значений: 421 53 1806.

Начальный адрес другой, но это единственное различие. В самом деле, имя массива можно использовать как указатель. Более того, вы можете определить указатель как массив. Рассмотрим следующие важные тождества:

(list+i)==&(list[i]);
*(list+i)==list[i].

В обоих случаях выражение слева эквивалентно выражению справа, и вы можете использовать одно вместо другого, не принимая во внимание, описан list как указатель или как массив.

Единственное различие между описанием list как указателя или как массива состоит в размещении самого массива. Если вы описали list как массив, то программа автоматически выделяет требуемый объем памяти, а имя массива является константой-указателем. Если же вы описали переменную list как указатель, то вы сами обязательно должны выделить память под массив, используя для этого функцию calloc или сходную с ней функцию, или же присвоить этой переменной адрес некоторого сегмента памяти, который уже был определен ранее.

Не забудьте, что индекс массива начинается с элемента [0], а не с элемента [1]. Наиболее распространенная ошибка может быть проиллюстрирована на примере следующей программы:

```
main()
{
 int list[100],i;
 for (i = 1; i <= 100; i++)
 list[i] = i + 1;
}
```

Данная программа оставляет первый элемент list – list[0] неинициализированным и записывает значение в несуществующий элемент list – list[100], возможно, испортив при этом другие данные.

Правильная программа будет иметь следующий вид:

```
main()
{
 int list[100],i;
 for (i = 0; i < 100; i++)
 list[i] = i+1;
}
```

6.2.2. *Инициализация массивов*

Начальные значения элементам массива можно присвоить в месте описания массива следующим образом:

тип имя_массива[] = {знач1, знач2, ..., значN};

Начальные значения элементов массива заключаются в фигурные скобки. Если пользователь не указал в квадратных скобках размер массива, то компилятор сам задает размер массива по числу приведенных начальных значений в фигурных скобках.

Рассмотрим программу:

```
#include <stdio.h>
int data[5] = {5, 4, 3, 2, 1};
float scores[] = {3.4, 2.7, 1.8, 6.9, -24.3};
char prompt[] = {'O', 'т', 'в', 'е', 'т', ':', '', '\O'};
main();
{
printf("\nРазмер массива data = %d", sizeof(data));
printf("\nРазмер массива scores = %d", sizeof(scores));
printf("\nРазмер массива prompt = %d", sizeof(prompt));
printf("\n\npromrt = %s\n", prompt)
}
```

Здесь описаны три массива: data, scores и prompt.

При описании массива data явно указано, что он содержит пять целых. Таким образом, его размер – 10 байтов.

Размерности массивов scores[] и prompt[] заданы неявно путем инициализации значений: в первом случае – пяти вещественных чисел, а во втором – восьми символов.

Массив scores[] можно описать и как scores[5]. Соответственно, вместо описания prompt[] можно подставить и prompt[8].

Если начальные значения не заданы, то все элементы указанных массивов будут нулевыми, поскольку все статические переменные инициализируются значением «нуль».

6.2.3. *Строки и операции над ними*

Си не поддерживает отдельный строковый тип данных, но он все же предусматривает два слегка отличающихся подхода к опре-

делению строк. Один состоит в использовании символьного массива, другой заключается в использовании указателя на символ.

Рассмотрим использование символьного массива для определения строки:

```
#include<string.h>
main ()
{
  char msg[30];
  strcpy(msg, "Hello, world");
  puts(msg);
}
```

Выражение [30] после msg предписывает компилятору выделить память для 29 символов, т.е. для массива из 29 переменных типа char (30-е знакоместо должно быть заполнено нулевым символом – \0. Константа-указатель msg не содержит символьное значение; она хранит адрес (некоторого места в памяти) первого из этих 29 переменных типа char.

Когда компилятор обнаруживает оператор strcpy(msg, "Hello, world"), он делает следующее:

– создает строку "Hello, world", сопровождаемую нулевым (\0) символом (с кодом ASCII 0), в некотором месте файла объектного кода (резервирует 13 байт памяти);

– вызывает подпрограмму strcpy из стандартной библиотеки, интерфейс с которой описан в файле string.h, которая копирует символы из этой строки по одному в участок памяти, указываемый константой msg. Он делает это до тех пор, пока не будет скопирован нулевой символ в конце строки "Hello, world".

Когда вы вызываете функцию puts(msg), то ей передается значение msg – адрес первой буквы, на которую он указывает. Затем puts проверяет, не является ли символ по этому адресу нулевым. Если да, то puts заканчивает работу; иначе puts печатает этот символ, добавляет единицу (1) к адресу и делает проверку на нулевой символ снова.

Из-за этой зависимости от нулевого символа известно, что стро-

ки в Си называются «завершающиеся нулем», т.е. они представляют собой последовательности символов, заканчивающиеся нулевым символом. Этот подход позволяет снять ограничения с длины строк. Строка может быть такой длины, какой позволяет память для ее хранения.

Второй метод, который можно использовать для определения строк, – это указатель на символы. Рассмотрим программу:

```
main()
{
  char *msg;
  msg = "Hello, world";
  puts(msg);
}
```

Звездочка (*) впереди msg указывает компилятору, что msg является указателем на символ. Другими словами, msg может хранить адрес некоторого символа. Однако при этом компилятор не выделяет никакого пространства для размещения символов и не инициализирует msg каким-либо конкретным значением, а лишь резервирует память, необходимую для хранения указателя msg (два байта для малой модели памяти).

Когда компилятор находит оператор msg = "Hello, world", он делает следующее:

– как и раньше, создает строку "Hello, world", сопровождаемую нулевым символом, где-то внутри файла объектного кода;

– присваивает начальный адрес этой строки – адрес символа H – переменной msg, т.е. инициализирует указатель msg конкретным значением.

Команда puts(msg) работает так же, как и раньше, печатая символы до тех пор, пока она не встретит нулевой символ.

В соответствии с K&R строковые константы состоят обязательно из одной строки, имеющей конструкцию: двойные кавычки, текст, двойные кавычки ("текст"). Для продолжения символьной последовательности на новой строке вы должны использовать обратный слеш.

В Турбо Си разрешается использовать многостроковые элементы в строковых константах, которые могут потребоваться для конкатенации (соединения) строк. Так, например, вы можете сделать следующее:

```
 main()
{
  char  *p;
  p = "Это пример того, как Турбо Си"
    " будет автоматически\nвыполнять конкатенацию"
    " ваших очень длинных строк,\nделая наглядным"
    " общий вид программ."
  puts(p);
}
```

Вот результат работы программы:

```
Это пример того, как Турбо Си будет автоматически
выполнять конкатенацию ваших очень длинных строк,
делая наглядным общий вид программ.
```

Как мы уже говорили, строки можно объявить как указатели на тип данных char или как массивы данных типа char. Речь шла о том, что между ними существует лишь одно важное различие: если вы используете указатель на данные типа char, то память для строки не резервируется; если вы используете массив данных, то память резервируется автоматически и константа-указатель – имя массива – содержит адрес начала зарезервированной области памяти.

Недостаточное понимание этой разницы может привести к двум типам ошибок. Рассмотрим следующую программу:

```
main()
{
  char *name;
  char msg[10];
  printf("Назовите свое имя.");
  scanf("%s",name);
  msg = "Здравствуйте, ";
```

```
    printf("%s %s:",msg,name);
}
```

На первый взгляд, все законно, немного неуклюже, но вполне допустимо. Однако здесь допущены две ошибки.

Первая ошибка содержится в выражении

```
scanf("%s",name).
```

Выражение само по себе законно и корректно. Поскольку name является указателем на char, вам не нужно ставить перед ним адресный оператор (&). Однако память для name не зарезервирована; строка, которую вы введете, будет записана по какому-то случайному адресу, который окажется в name. Компилятор обнаружит это, но, поскольку эта ситуация не приведет к сбою выполнения программы (так как строка все же будет сохранена), компилятор выдаст лишь предупреждающее сообщение, но не ошибку:

```
"Possible use of 'name' before definition"
("Возможно использование 'name' до ее определения")
```

Вторая ошибка содержится в операторе msg = "Здравствуйте,". Компилятор считает, что вы пытаетесь заменить значение msg на адрес строковой константы "Здравствуйте,". Это сделать невозможно, поскольку имена массивов являются константами и не могут быть модифицированы (как, например, 7 является константой и нельзя записать "7 = i"). Компилятор выдаст вам сообщение об ошибке:

```
"Lvalue required."
("Использование константы недопустимо")
```

Каково решение этой проблемы? Простейший выход – изменить способ описания переменных name и msg:

```
main()
{
```

```
  char name[10];
  char *msg;
  printf("Назовите свое имя");
  scanf("%s",name);
  msg = "Здравствуйте,";
  printf("%s%s",msg,name);
}
```

Эта программа безупречна. Переменной name выделяется память независимо от памяти, выделяемой для строки, которую вы вводите, тогда как в msg присваивается адрес строковой константы "Здравствуйте,". Если, тем не менее, вы оставите старое описание, вам нужно изменить программу следующим образом:

```
#include <string.h>
#include <alloc.h>
main()
{
  char *name;
  char msg[10];
  name = (char *) malloc (10);
  printf("Назовите свое имя");
  scanf("%s",name);
  strcpy(msg,"Здравствуйте,");
  printf("%s%s",msg,name);
}
```

Вызов функции malloc выделяет отдельно 10 байтов памяти и присваивает адрес этого участка памяти name, решив нашу первую проблему. Функция strcpy производит посимвольное копирование из строковой константы string "Здравствуйте," в массив msg.

6.2.4. *Многомерные массивы*

Описание многомерных массивов выглядит так:

```
type name[size1][size2]...[sizeN];
(тип имя [размер1][размер2]...[размерN])
```

Рассмотрим следующую программу, которая определяет два двумерных массива, а затем выполняет их матричное умножение:

```
main()
{
 int a[3][4] = {{ 5, 3, -21, 42},
        {44, 15,  0, 6},
        {97 , 6, 81, 2}};
 int b[4][2] = {{22,  7},
        {97, -53},
        {45,  0},
        {72,  1} };

 int c[3][2],i,j,k;
 for (i=0; i<3; i++) {
   for (j=0; j<2; j++) {
     c[i][j] = 0;
     for (k=0; k<4; k++)
       c[i][j] += a[i][k] * b[k][j];
   }
 }
 for (i=0; i<3; i++) {
   for (j=0; j<2; j++)
     printf("c[%d][%d] = %d ",i,j,c[i][j]);
   printf("\n");
 }
}
```

Отметим два момента в этой программе. Синтаксис определения двумерного массива состоит из набора {...} списков, разделенных запятой. Квадратные скобки ([ ]) используются для записи каждого индекса.

Некоторые языки для определения массивов используют синтаксис [i,j]. Так можно написать и на Си, но это все равно, что сказать просто [j], так как запятая интерпретируется как оператор, означающий «определить i, затем определить j, затем присвоить всему выражению значение j».

Для полной уверенности ставьте квадратные скобки вокруг каждого индекса.

Многомерные массивы хранятся в памяти слева направо по правилу «строки – столбцы». Это означает, что последний индекс изменяется быстрее. Другими словами, в массиве arr[3][2] элементы arr хранятся в памяти в следующем порядке:

arr[0][0] arr[0][1] arr[1][0] arr[1][1] arr[2][0] arr[2][1]

Тот же принцип сохраняется для массивов трех- и большей размерности.

Выражение arr[i][j] переводится компилятором Турбо Си в эквивалентное выражение *(*(arr+i)+j). Имя массива arr является константой-указателем на строку с индексом 0. Выражение *(arr+i) задает указатель на строку с индексом i.

6.2.5. *Массивы и функции*

Что произойдет, если вы захотите передать массив в функцию? Рассмотрим следующую функцию, возвращающую индекс минимального числа массива int:

```
int imin(int list[], int size)
{
 int i, minindx, min;
 minindx = 0;
 min = list[minindx];
 for (i = 1; i < size; i++)
   if (list[i] < min) {
     min = list[i];
     minindx = i;
    }
 return(minindx);
}
```

Здесь вы видите одну из важных особенностей Си: вам не обязательно знать при трансляции величину list[]. Почему? Потому что компилятор считает list[] начальным адресом массива и не заботится о том, где его конец.

Программа, обращающаяся к функции imin, может выглядеть так:

```
#include <stdlib.h>
#define VSIZE 22
main()
{ extern int imin(int list[], int size);
  int i,vector[VSIZE];
  for (i = 0; i < VSIZE; i++) {
    vector[i] = rand();
    printf("vector[%2d] = %6d\n",i,vector[i]);
  }
  i = imin(vector,VSIZE);
  printf("minimum: vector[%2d] = %6d\n",i,vector[i]);
}
```

Может возникнуть вопрос: что именно передается в imin? В функцию imin передается начальный адрес массива vector. Это означает, что если вы производите какие-либо изменения массива list в imin, то те же изменения будут произведены и в массиве vector.

Например, вы можете написать следующую функцию:

```
        void setrand(int list[],int size);
{
  int i;
  for (i = 0; i < size; i++) list[i] = rand();
}
```

Теперь для инициализации массива vector вы можете написать в main setrand(vector,VSIZE). Следует заметить, что массиву vector будут присвоены некие случайные числа, являющиеся результатом работы датчика случайных чисел, эмулируемого функцией Турбо Си rand(), из диапазона 0-32767.

А как передавать многомерный массив? Имеется ли такая возможность? Предположим, вы хотите модифицировать setrand для работы с двумерным массивом. Вы должны написать приблизительно следующее:

```
void setrand(int matrix[][CSIZE],int rsize)
{
  int i,j;
  for (i = 0; i < rsize; i++) {
    for (j = 0; j < CSIZE; j++)
      matrix[i][j] = rand();
  }
}
```

Здесь CSIZE – это глобальная константа, определяющая второе измерение массива. Другими словами, любой массив, передаваемый setrand, получит второе измерение массива, равное CSIZE.

В общем случае если многомерный массив передается в качестве параметра функции, то необходимо задавать все его размерности, кроме первой.

Однако есть еще одно решение. Предположим, у вас есть массив matrix[15][7], который вы хотите передать в setrand. Если вы используете описание

```
setrand(int list[],int size),
```

то обращение к функции будет иметь вид:

```
setrand(matrix,15*7);
```

Массив matrix будет рассматриваться функцией setrand как одномерный массив, содержащий 105 элементов (15 строк * 7 столбцов), с которым будут произведены необходимые вам действия.

6.2.6. *Свободные массивы*

Свободными называются двумерные массивы, или матрицы, размер каждой из строк которых может быть различным.

Пример описания двумерного массива names, который инициализируется с использованием массива из трех строк (указателей типа char):

```
char *names[]=
  {
   "Андрей",
   "Петр",
   "Николай"
  };
```

Здесь names[0]="Андрей";
names[1]="Петр";
names[2]="Николай";
имя массива names является адресом начала строки "Андрей".

6.3. **Структуры**

6.3.1. *Описание структур*

Массивы и указатели позволяют вам создавать список элементов одного типа. А что, если вы хотите создать нечто, содержащее элементы различного типа? Для этого используются СТРУКТУРЫ.

Структура – это конгломерат элементов различного типа. Допустим, вы хотите сохранить информацию о звезде: ее имя, спектральный класс, координаты и т.д. Вы можете описать это следующим образом:

```
typedef struct {
 char  name[25];
 char  class;
 short subclass;
 float decl,RA,dist;
} star ;
```

Здесь определена структура (struct) типа star. Сделав такое описание в начале своей программы, вы можете дальше использовать этот определенный вами тип данных:

```
main()
{
 star mystar;
```

```
    strcpy(mystar.name,"Епсилон Лебедя");
    mystar.class  = 'N';
    mystar.subclass = 2;
    mystar.decl   = 3.5167;
    mystar.RA     = -9.633;
    mystar.dist   = 0.303;
    /* конец функции main() */
   }
```

Вы обращаетесь к каждому элементу структуры, используя его составное имя, состоящее из имени структурной переменной (на первом месте), и имен образующих ее элементов в порядке иерархической подчиненности, разделенных точками (.). Конструкция вида varname.memname (имя переменной.имя элемента) считается эквивалентной имени переменной того же типа, что и memname, и вы можете выполнять с ней те же операции.

Вы можете описывать указатели на структуры точно так же, как и указатели на другие типы данных. Это необходимо для создания связных списков и других динамических структур данных, элементами которых, в свою очередь, являются структуры данных.

Фактически указатели на структуры так часто используются в Си, что существует специальный символ для ссылки на элемент структуры, адресованной указателем. Рассмотрим следующий вариант предыдущей программы:

```
#include <alloc.h>
main()
{
 star *mystar;
 mystar = (star *) malloc(sizeof(star));
 strcpy(mystar -> name,"Эпсилон Лебедя");
 mystar -> class  = 'N';
 mystar -> subclass = 2;
 mystar -> decl   = 3.5167;
 mystar -> RA     = -9.633;
 mystar -> dist   = 0.303;
 /* Конец функции main() */
}
```

В этом варианте mystar объявляется как указатель типа star, а не как переменная типа star. Память для mystar резервируется путем обращения к функции malloc. Теперь, когда вы ссылаетесь на элементы mystar, используйте ptrname -> memnane. Символ -> означает, что «элемент структуры направлен в ...». Это сокращенный вариант от обозначения (*ptrname).memnane, принятый в Си.

В общем случае структуры конструируются следующим образом:

```
struct имя_структуры
{
 тип_1 поле_1;
 тип_2 поле_2;
 ...
 тип_n поле_n;
};
```

Обратите внимание на точку с запятой после закрывающей скобки в описании структуры. Отсутствие такого разделителя является частой синтаксической ошибкой.

Описание переменных структурного типа выглядит следующим образом:

```
struct struct_name x, y, z;
```

Структурным переменным можно присваивать значение агрегатно (сразу по всей структуре), но их нельзя сравнивать на равенство или неравенство. Для сравнения структур программисту следует написать специализированные функции, зависящие от приложения.

В тех случаях, когда допускается использовать унарную операцию & получения адреса для структуры в целом, можно эту же операцию использовать и для получения адреса элемента структуры.

Так как при описании переменной структурного типа требуется, чтобы в описании присутствовал спецификатор struct, обычно пол-

ное имя структурного типа оформляют в виде макро. Покажем такой прием на следующем примере.

Пусть нам нужно выполнить действия над комплексными типами. Поскольку комплексное число может быть представлено двумя вещественными, то воспользуемся следующей структурой:

```
#define COMPLEX struct complex_type
COMPLEX
{

float real;
float imag;
};
```

Приведенное макроопределение COMPLEX позволяет описывать комплексные переменные в естественной форме.

Если переменные c1, c2 и c3 описаны как комплексные, то операция умножения c1 на c2 для вычисления c3 может быть реализована следующим образом:

```
COMPLEX c1, c2, c3;
c3.real = c1.real * c2.real – c1.imag * c2.imag;
c3.imag = c1.imag * c2.real + c1.real * c2.imag;
```

Удобно ввести абстрактные арифметические операции над комплексными числами, реализовав для каждой такой операции соответствующую функцию. Заметим, что непосредственный доступ к полям структур – это плохой стиль программирования.

Структуры можно использовать при построении абстрактных типов данных. Все операции, которые разрешены применительно к структуре, должны быть при этом реализованы в виде отдельных функций.

В старых версиях Си при передаче структуры в качестве параметра функции требовалось указывать адрес структуры, а не ее саму. В связи с этим формальным параметром функции должен был служить указатель на структуру. В Турбо Си и в стандарте Си ANSI разрешается передавать, изменять и возвращать структуры по зна-

чению. Тем не менее, необходимо все-таки использовать указатели для передачи структур, с тем чтобы обеспечивать совместимость программ с другими системами программирования Си.

Еще один пример описания структурной переменной:

```
struct complex_type
{
 double real;
 double imag;
} my_complex;
```

Имя переменной указывается вслед за описанием структуры. Структуры могут задаваться не полностью. Поле структуры может быть описано как указатель на саму структуру.

6.3.2. *Битовые поля*

Целочисленные элементы могут быть помещены в маленький объем памяти с использованием битовых полей. Битовые поля часто используются в машинозависимых приложениях, когда требуется работать с битовыми картами, описывающими конфигурацию аппаратуры.

В некоторых реализациях Си для каждого битового поля разрешено использовать только беззнаковый тип. В Турбо Си для битовых полей можно применять как тип int, так и тип unsigned int. Поля целого типа представляются в виде двоичного дополнительного кода, где самый левый (больший) бит соответствует знаку.

Существенно, что в Турбо Си битовые поля располагаются от меньших номеров к большим. В других компиляторах с Си может быть принят иной порядок. Программируя на Турбо Си, следует внимательно кодировать работу с битовыми полями, чтобы не столкнуться с проблемами мобильности.

Рассмотрим пример работы с битовыми полями:

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

```
{
unsigned i:2;
unsigned j:6;
int    :3;
unsigned k:3;
unsigned l:2;
} bit;
main ()
{
bit.i=2;
bit.j=9;
bit.k=7;
bit.l=2;
printf(bit=%u\n", bit);
}
```

Первое поле i имеет размер 2 бита. Второе поле j занимает 6 битов. Следующее поле не имеет имени. Вы видите здесь пример неполного описания структуры. Следующее поле k ограничено 3 битами. Последнее поле l ограничено 2 битами.

На рис. 6.2 показано распределение памяти под переменную bit.

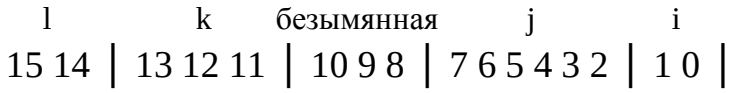

Рис. 6.2. Распределение памяти под переменную bit

Битовое представление переменной bit показано на рис. 6.3.

Рис. 6.3. Битовое представление переменной bit

Программа выведет 47142.

6.3.3. *Инициализация структур*

Структуры можно инициализировать непосредственно в месте описания. Рассмотрим пример инициализации структурной переменной и передачи ее адреса в качестве параметра функции:

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>
#define RECORD struct data_record
RECORD
{
 char *last_name;
 char *first_name;
 long id_number;
};
 RECORD my_record = { "Ричард",
          "Уинер",
          12345678
         };
main()
 {
  extern void enter_data(RECORD *data);
  extern void print_record(RECORD *data);
  print_record(&my_record);
  enter_data(&my_record);
  print_record(&my_record);
  printf("\n");
 }
 void enter_data(RECORD *data)
 {
  char info[40];
printf("\nВведите фамилию:");
scanf("%s", info);
data -> last_name = (char *)
        malloc(strlen(info) + 1);
strcpy(data -> last_name, info);
printf("\nВведите имя:");
scanf("%s", info);
data -> first_name = (char *)
```

```
        malloc(strlen(info) + 1);
 strcpy(data -> first_name, info);
 printf("\nВведите номер:");
 scanf("%1d", &data -> id_number);
}
void print_record(RECORD *data)
{
 printf("\n\nФамилия: %s", data -> last_name);
 printf("\nИмя: %s", data -> first_name);
 printf("\nНомер: %ld", data -> id_number);
}
```

Здесь задается структура RECORD(struct data_record), которая содержит три поля: last_name (строка), first_name (строка), id_number (длинное целое), а также описывается и инициализируется структурная переменная my_record.

Начальные значения структурной переменной задаются внутри набора скобок. Соответствие между константами и полями устанавливается по порядку записи. Это означает, что первая по порядку константа связывается с первым по порядку полем, вторая – со вторым и т.д.

Параметром функции enter_data является адрес структурной переменной типа RECORD. Память под строки first_name и last_name отводится внутри функции enter_data. Это гарантирует, что для размещения элемента указанной структуры будет отведено ровно столько памяти, сколько нужно.

С помощью функции print_record осуществляется вывод содержимого структурной переменной, адрес которой передается в качестве параметра.

6.3.4. *Массивы структур*

Комбинируя структуры и массивы, можно строить универсальные и гибкие структуры данных. Рассмотрим пример:

```
#include <stdio.h>
#define RECORD struct data_record
```

```
RECORD
{
 char *last_name;
 char *first_name;
 long id_number;
};
RECORD data_base[ 3 ] = { "Уинер", "Ирвин", 1,
            "Уинер", "Марк", 2,
            "Уинер", "Эрик", 3
           };
main()
{
 extern void print_records( RECORD *data, int size);
 print_records( data_base, 3 );
 printf( "\n" );
}
void print_records( RECORD *data, int size)
{
 int i;
 for ( i = 0; i < size; i++ )
{
 printf( "\n\nФамилия : %s", ( data + i ) -> last_name );
 printf( "\nИмя : %s", ( data + i ) -> first_name );
 printf( "\nНомер : %ld", ( data + i ) -> id_number );
}
}
```

Описывается массив структур data_base типа RECORD (struct data_record). Массив инициализируется значениями при его описании.

Для снятия ссылки с полей last_name, first_name и id_number параметра data_base, передаваемого подпрограмме, используется операция ->.

Рассмотрим еще один вариант реализации функции print_records (результаты работы обеих программ совпадают):

```
 main()
 {
  extern void print_records(RECORD data [], int size );
```

```
 print_records( data_base, 3 );
 printf( "\n" );
}
void print_records( RECORD data[], int size )
{
 int i;
 for ( i = 0; i < size; i++ )
 {
  printf( "\n\nФамилия : %s", data [ i ].last_name );
  printf( "\nИмя : %s", data [ i ].first_name );
  printf( "\nНомер : %ld", data [ i ]. id_number );
 }
}
```

6.4. **Объединения**

Используя объединения (union), можно в одной и той же области памяти размещать данные различных типов. Естественно, что в данный момент времени в памяти могут быть размещены значения только одного включенного в объединение типа.

Рассмотрим пример:

```
#include <stdio.h>
#define ALTERNATIVE_DATA union data_record
ALTERNATIVE_DATA
{
 char s[ 4 ];
 float r;
 int i;
 char ch;
 unsigned j;
};
main()
{
 ALTERNATIVE_DATA data;
 printf("\nsizeof( data ) = %d\n",
 sizeof( data ) );
}
```

Здесь определяется объединение ALTERNATIVE_DATA.

Программа выводит следующую строку:

sizeof(data) = 4.

Размер памяти, требуемой для размещения объединения, определяется размером наибольшего поля, длина самого большого поля равна 4 байтам.

Для создания эквивалента записей с вариантами, имеющимися в Паскале или Модуле-2, можно использовать объединения внутри структур. Такой прием использован в программе:

```
#include <stdio.h>
#define DATA struct data_type
enum employee_type {salary,hourly};
DATA
{
 char last_name[10];
 char first_name[10];
 enum employee_type tag;
 union
 {
 float hourly_wage;
 float annual_salary;
 } wage;
};
main()
{
 DATA person;
 /* ... */
}
```

Структура DATA содержит поле с именем wage типа union_type. Поскольку не имеет смысла хранить для одного человека и размер его почасовой ставки, и оклад (работник может быть либо почасовиком, либо штатным сотрудником), то в объединении поля hourly_wage (почасовой тариф) и annual_salary (оклад) совмещены, что обеспечивает экономию памяти.

В структуру включено поле tag, позволяющее узнавать, какое из двух полей (hourly_wage или annual_salary) является активным в каждом конкретном случае. Не существует другого способа узнать, какое из полей активно.

6.5. **Файлы и их типизация**

6.5.1. *Определение файла*

Термин «файл» происходит из представления о последовательной структуре информационных записей. Такой файл располагается на диске. Базовыми операциями над файлом являются:

– считывание блока данных из файла в оперативную память (одна или несколько записей);

– запись блока данных из оперативной памяти в файл;

– обновление блока данных в файле;

– считывание определенной записи данных из файла;

– занесение определенной записи данных в файл.

Состав файла задается структурой FILE, описание которой содержится в файле stdio.h. Эта структура, скопированная непосредственно из файла stdio.h системы Турбо Си, имеет вид:

```
typedef struct
{
short level;      /*Уровень буфера*/
unsigned flags;     /*Флаги статуса файла*/
char fd;        /*Дескриптор файла*/
char hold;       /*Предыдущий символ, если нет буфера*/
short bsize;      /*Размер буфера*/
unsigned char *buffer; /*Буфер передачи данных*/
unsigned char *curp;  /*Текущий активный указатель*/
short token;      /*Для проверки корректности*/
} FILE
```

Глобальные файловые переменные stdin, stdout и stderr инициируются системой перед началом работы любой прикладной программы.

В начале работы файл stdin приписывается к клавиатуре, но с использованием стандартных команд переадресации ввода MS DOS он может быть переназначен. Такие команды переадресации ввода происходят от появившихся ранее и более устоявшихся команд переадресации в системе UNIX.

Файл stdout приписывается в выводному терминалу.

Файл stderr приписывается к пользовательскому выводному терминалу и не может быть переназначен на другой файл. Таким образом, если пользователь желает послать сообщение на терминал без учета того, что интерпретация файла по умолчанию могла быть изменена, он должен использовать выводной поток stderr.

Функции, реализующие работу с файловым вводом и выводом в Турбо Си, собраны в библиотеке stdio.h. Все такие функции совместимы с аналогичными функциями в системе UNIX.

6.5.2. *Текстовые и двоичные типы файлов*

Текстовый файл состоит из последовательности символов, разбитой на строки. Для деления на строки используется управляющий символ '\n'. Текстовые файлы оказываются переносимыми с одного типа компьютера на другой, если символы, содержащиеся в символьном потоке, принадлежат стандартному набору символов.

В Турбо Си, как и в большинстве реализаций Си для машин с процессором типа 80x86, применяется отображение целых чисел в символы в соответствии со стандартом ANSI. При построении текстовых файлов в Турбо Си разрешено использовать 256 символов, включающих и псевдографические символы фирмы IBM.

Жесткий стандарт языка Си ANSI, поддерживаемый системой Турбо Си, предписывает, чтобы реализация обеспечивала работу не менее чем с 254 символами.

Двоичный файл – это последовательность значений типа char. Использование двоичных файлов делает программы немобильными при их переносе из одной среды в другую. Любой набор данных может быть представлен как набор символов, но такое представление может изменяться при переходе из одной реализации Си к другой.

6.5.3. *Функции работы с файлами*

М а к р о EOF

Макро EOF определяется следующим образом:

```
#define EOF ( -1 )
```

Этот макро в операциях ввода/вывода служит для обозначения и проверки конца файла. Подразумевается, что макро EOF имеет тип значения символьный знаковый. Если символьный тип беззнаковый, то использовать EOF нельзя.

Ф у н к ц и я fopen

Функция fopen используется для открытия файла. Интерфейс с функцией fopen описывается следующим образом:

```
FILE *fopen ( char *filename, char *type );
```

В качестве первого параметра функции должно передаваться правильное имя файла.

Второй параметр определяет тип открываемого файла. Допустимы следующие типы файлов:

– "r" – открыть уже существующий файл на ввод (чтение);

– "w" – создать новый файл или очистить уже существующий файл и открыть его на вывод (запись);

– "a" – создать новый файл для вывода или осуществить вывод в конец уже существующего файла (добавление);

– "r+" – открыть существующий файл для обновления, которое будет проводиться с начала файла (чтение с обновлением);

– "w+" – создать новый или очистить существующий файл для обновления его содержимого (запись с обновлением);

– "a+" – создать новый файл или подстроиться в конец существующего файла для обновления его содержимого (добавление с обновлением).

Дополнительно к каждой из приведенных строк можно добавить символ 'b', указывающий на то, что открывается двоичный файл, или 't' – текстовый файл.

Функция fopen возвращает указатель на структуру FILE, описывающую файл.

Открытие в текстовом режиме заключается в том, что вывод символа новой строки влечет за собой внесение в файл пары символов cr-lf, а ввод из файла пары символов cr-lf трактуется как ввод символа новой строки. По умолчанию принимается режим открытия, определяемый глобальной переменной _fmode. Если ей присвоено значение вида O_TEXT, то открытие произойдет в текстовом режиме, а если значение вида O_BINARY, то открытие произойдет в двоичном режиме. Переменная _fmode описана в файле fcntl.h, а ее начальным значением является O_TEXT.

Пример вывода в файл TARGET символов j, cr, lf, b:

```
/*fopen*/
#include <stdio.h>
#include <fcntl.h>
main ()
{
FILE *Out;
extern_fmode;
Out = fopen("TARGET","wt");
fprintf(Out,"j\n");
fclose(Out);
_fmode = o_BINARY;
Out = fopen("TARGET","a+");
fprintf(Out,"b");
fclose(Out);
}
```

Если при открытии файла произошли какие-либо ошибки, то в глобальную переменную errno будет записан код (номер) ошибки и будет возвращен нулевой (0) указатель.

Ф у н к ц и я fflush

Функция fflush служит для вывода каждого внутреннего буфера в файл (stream). Интерфейс с этой функцией выглядит следующим образом:

```
int fflush (FILE *stream);
```

После вызова функции файл остается открытым. Если при выполнении функции появятся ошибки, то будет возвращено значение EOF; в противном случае будет возвращен 0.

Пример:

```
/*fflush*/
#include <stdio.h>
main ()
{
FILE *Out;
Out = fopen("TEXT","w");
fprintf(Out,"Kaja");
fflush(Out);
abort();
}
```

Несмотря на аварийное завершение выполнения программы, произошел вывод в файл TEXT надписи Kaja.

Ф у н к ц и я freopen

Функция freopen служит для закрытия существующих файлов и открытия их заново. Чаще всего эта функция используется для переназначения стандартных потоков (stdin, stdout, stderr) на другие файлы. Интерфейс с функцией freopen имеет вид

```
FILE *freopen ( char *filename, char *type, FILE *stream);
```

Происходит закрытие файла, идентифицируемого аргументом

stream, установка файла с именем filename и открытие его в режиме, указанном в аргументе type таким образом, чтобы его можно было идентифицировать аргументом stream.

Результат функции: переменная типа (FILE *), идентифицирующая файл, или пустое значение, если открытие файла окажется невозможным.

Пример вывода буквы j в стандартный выходной файл и буквы b в файл WORK:

```
/*freopen*/
#include <stdio.h>
main ()
{
putchar('j');
(void)freopen("WORK","w",stdout);
putchar('b');
}
```

Ф у н к ц и я fclose

Интерфейс с функцией fclose выглядит следующим образом:

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

С помощью этой функции в файл выводятся соответствующие буферы и указанный файл закрывается. Если закрытие выполняется правильно, то вырабатывается значение «нуль», в противном случае вырабатывается значение EOF.

Пример:

```
/*fclose*/
#include <stdio.h>
main ()
{
FILE *Out;
Out = fopen("TARGET","w");
putc('j', Out);
```

```
putc('b', Out);
printf(fclose(Out) ? "Fail": "Done");
}
```

Функции fgetc и getc

Интерфейс с функцией fgetc описывается следующим образом:

```
int fgetc (FILE *stream);
```

С помощью этой функции из указанного файла считывается очередной символ и его значение переводится в тип int. Если при считывании обнаруживается ошибка или достигается конец файла, то возвращается значение EOF.

Макро getc действует точно так же, как и функция fgetc.

Пример копирования символов из стандартного входного файла в стандартный выходной файл:

```
/*fgetc*/
#include <stdio.h>
main ()
{
int Chr;
while((Chr = fgetc(stdin)) ! = EOF)
putc(Chr, stdout);
}
```

Функция getchar

Интерфейс с функцией getchar выглядит следующим образом:

```
int getchar ();
```

С помощью этой функции из стандартного входного потока stdin считывается очередной символ и его значение переводится в тип int. Если переназначения стандартного входного потока не производилось, то ввод осуществляется с клавиатуры. В противном слу-

чае ввод будет осуществляться из файла, назначенного для входного потока и указанного в командной строке при вызове программы.

Пример копирования содержимого стандартного входного файла в файл TEXT:

```
/*getchar*/
#include <stdio.h>
main ()
{
FILE *Out;
int Chr;
Out = fopen("TEXT","w");
while((Chr = getchar()) ! = EOF)
 putc(Chr,Out);
}
```

Ф у н к ц и я ungetc

Интерфейс с функцией ungetc выглядит следующим образом:

```
int ungetc (char ch, FILE *stream);
```

С помощью этой функции символ ch возвращается обратно в файл. Если сразу же будет вызвана функция fgetc, getc или getchar, то такой символ будет считан первым. Если при возвращении не произошло ошибок, то будет считано значение этого символа, в противном случае будет считано значение EOF.

Функцию ungetc удобно использовать в том случае, если требуется просмотреть файл на один символ вперед, не нарушая сам файл.

Пример вывода буквы j:

```
/*ungetc*/
#include <stdio.h>
#include <conio.h>
main ()
{
```

```
char Chr;
ungetc('j',stdin);
Chr = getc(stdin);
putchar(Chr);
}
```

Ф у н к ц и я fseek

Интерфейс с функцией fseek выглядит следующим образом:

```
int fseek (FILE *stream, long offset, int wherefrom);
```

Эта функция служит для произвольного доступа к байтам, обычно внутри двоичных файлов.

Первый аргумент задает файл, к которому должен осуществляться прямой доступ.

Второй аргумент offset является длинным целым числом со знаком и указывает число байтов смещения от точки, определяемой третьим параметром функции.

Третий параметр wherefrom указывает точку, от которой следует начинать отсчет смещения, заданного вторым аргументом:

– значение 0 – смещение от начала файла;

– значение 1 – смещение от текущей позиции файла;

– значение 2 – смещение от конца файла.

Для облегчения работы с функцией fseek определены следующие константы:

– #define SEEK_SET 0;

– #define SEEK_CUR 1;

– #define SEEK_END 2.

Например, для установки стрелки (указателя) файла на конец файла нужно воспользоваться таким обращением:

```
fseek (stream_name, OL, SEEK_END);
```

Пример вывода надписи Iza:

```
/*fseek*/
#include <stdio.h>
main ()
{
FILE *Upd;
char Name[3];
Upd = fopen("Work", "w+t");
fprintf(Upd, "Ewa-Iza-Jan");
fseek(Upd,-7,SEK_CUR);
fscanf(Upd, "%3s", Name);
printf("%3s", Name);
}
```

Ф у н к ц и я rewind

Интерфейс с функцией rewind выглядит следующим образом:

```
void rewind (FILE *stream);
```

С помощью этой функции стрелка файла перемещается на начало файла, при этом обнуляется указатель конца файла и указатель ошибки. Аналогичное действие может быть выполнено и с помощью вызова

```
fseek (stream_name, OL, SEEK_SET);
```

Пример троекратного вывода в стандартный выходной файл содержимого файла AUTOEXEC.BAT:

```
/*rewind*/
#include <stdio.h>
main ()
{
int Count = 3;
int Chr;
FILE *Inp = fopen("\\AUTOEXEC.BAT", "r");
while(Count --){
 while((Chr = getc(Inp))! = EOF)
```

```
 putc(Chr, stdout);
putc('\n', stdout);
rewind(Inp);
}
}
```

Ф у н к ц и я fgets

Интерфейс с функцией fgets выглядит следующим образом:

```
char *fgets ( char *s, int n, FILE *stream);
```

С помощью этой функции в строку s считываются символы до тех пор, пока не будет выполнено одно из условий:

1. Начнется новая строка.
2. Достигнут конец файла.
3. Условия 1 или 2 не выполнились, но прочитано n-1 символов.

После того как из файла в строку s будут прочитаны символы, строка дополняется символом «нуль». Если при чтении встретился символ конца строки (условие 2), то он переносится в строку s и нулевой символ записывается за ним. Если операция считывания прошла успешно, то возвращается адрес строки s, в противном случае возвращается значение «нуль».

Пример:

```
/*fgets*/
#include <stdio.h>
main ()
{
char Buf[20];
printf("%s", fgets (Buf, 20, stdin));
}
```

В случае, когда первая строка стандартного входного файла содержит последовательность символов Jan, осуществляется вывод в стандартный выходной файл символов 'J', 'a', 'n' и '\n'.

Функция gets

Интерфейс с функцией gets имеет вид:

```
char *gets (char *s);
```

С помощью функции gets выполняется считывание символов из стандартного входного файла stdin. Если входной файл прерывается символом перехода на новую строку '\n', то этот символ отбрасывается и не попадает в строку s.

Поскольку в функции gets (в отличие от функции fgets) не задается числовой параметр, ограничивающий длину вводимой строки, то следует соблюдать осторожность, так как число символов, введенных из файла stdin, может превысить размер памяти, отведенный под строку s.

Пример ввода и вывода одной строки текста:

```
/*gets*/
#include <stdio.h>
main ()
{
char Line[81];
printf("%s", gets (Line));
}
```

Функции fputc и putc

Интерфейс с функцией fputc выглядит следующим образом:

```
int fputc( char ch, FILE *stream);
```

С помощью этой функции символ ch записывается в указанный файл. Если запись прошла успешно, то возвращается значение ch целого типа, в противном случае возвращается значение EOF.

Действие функции putc аналогично действию функции fputc, однако первая обычно оформляется в виде макро.

Пример вывода буквы E в стандартный выходной файл и возможного вывода знака вопроса в стандартный файл для сообщения об ошибках:

```
/*fputc*/
#include <stdio.h>
main ()
{
if (fputc('E',stdout) == EOF){
  fputc('?', stderr);
  exit(1);
}
}
```

Ф у н к ц и я putchar

Интерфейс с функцией putchar описывается следующим образом:

```
int putchar( char ch);
```

Функция putchar действует аналогично функции fputc, но записывает символ ch в стандартный выходной файл stdout. Обращение к функции putchar можно заменить на эквивалентное:

```
putc(ch, stdout);
```

Пример вывода в стандартный выходной файл надписи jb:

```
/*putchar*/
#include <stdio.h>
main ()
{
putchar (putchar('j')-8);
}
```

Ф у н к ц и я fputs

Интерфейс с функцией fputs выглядит следующим образом:

```
int fputs( char *s, FILE *stream);
```

Строка s, ограниченная символом «нуль», переписывается в выходной файл, причем символ «нуль» отбрасывается. Если при перезаписи возникает ошибка, то возвращается значение EOF, в противном случае возвращается ненулевое значение кода последнего выводимого символа.

Пример создания файла ONELINE, содержащего надпись Kaja:

```
/*fputs*/
#include <stdio.h>
main ()
{
FILE *Out = fopen ("ONELINE","w");
fputs("Kaja\n", Out);
exit(1);
}
```

Ф у н к ц и я puts

Интерфейс с функцией puts оформляется следующим образом:

```
int puts(char *s);
```

Эта функция выполняется аналогично функции fputs, за исключением того, что символы переписываются в стандартный выходной файл stdout и строка s независимо от того, содержит она символы или нет, дополняется символом конца строки '\n'.

Пример вывода надписи "Hello, world":

```
/*puts*/
#include <stdio.h>
main ()
```

```
{
puts("Hello, world");
}
```

Ф у н к ц и я fread

Интерфейс с функцией fread выглядит следующим образом:

```
int fread(void *ptr,unsigned elem_size,int count,FILE *stream);
```

С помощью этой функции из входного файла считывается и по адресу *ptr записывается не более чем count элементов размером elem_size байтов каждый. Функция возвращает число фактически считанных элементов.

Пример вывода надписи Count = 3:

```
/*fread*/
#include <stdio.h>
char Arr[30];
main ()
{
FILE *Upd = fopen("ARRAYS", "w+");
int Rep, Count;
for(Count = 1; Count! = 4; Count ++){
  Arr[0] = Count;
  fwrite(Arr, sizeoff Arr, 1, Upd);
}
 rewind(Upd);
 Count = 0;
 do{
  Rep = fread(Arr, sizeoff Arr, 1, Upd);
  if(Rep<1);
   break;
  Count++;
} while(1);
  printf("Count = %d", Count);
}
```

Функция fwrite

Интерфейс с функцией fwrite описывается следующим образом:

```
int fwrite(void *ptr,unsigned elem_size,int count,FILE *stream);
```

С помощью этой функции, начиная с адреса *ptr, считывается не более чем count элементов размером elem_size байтов каждый, и эти элементы записываются в выходной файл. Функция возвращает число фактически записанных элементов.

Функция feof

Интерфейс с функцией feof выглядит следующим образом:

```
int feof(FILE *stream);
```

Если при чтении из указанного файла достигнут конец файла, то возвращается значение «нуль», в противном случае возвращается ненулевое значение.

Если не предпринималась попытка прочитать из файла отсутствующий символ, следующий за последним, то функция feof не будет сигнализировать о том, что достигнут конец файла.

Функция ferror

Интерфейс с функцией ferror имеет вид

```
int ferror(FILE *stream);
```

Функция ferror позволяет опросить состояние признака ошибки указанного файла после чтения или записи в него. Возвращаемое нулевое значение показывает, что ошибок не было, в то время как ненулевое значение сигнализирует об ошибке.

Для чистки признака ошибки используется функция clearerr.

Ф у н к ц и я clearerr

Интерфейс с функцией clearerr описывается следующим образом:

```
void clearerr(FILE *stream);
```

С помощью указанной функции устанавливается в нуль состояние признака ошибки в указанном файле. Признак ошибки файла устанавливается в нуль и при закрытии файла.

Ф у н к ц и я rename

Интерфейс с функцией rename выглядит следующим образом:

```
int rename(char *oldname, char *newname);
```

С помощью указанной функции имя файла oldname меняется на новое имя newname. Если переименование прошло успешно, то возвращается значение «нуль», в противном случае возвращается ненулевое значение.

Пример перемещения и переименования файла:

```
/*rename*/
#include <stdio.h>
main ()
{
printf("%d", rename("\\ISA\\JAN", "\\EVA\\KAJA"));
}
```

Ф у н к ц и и fprintf, printf, sprintf и cprintf

С помощью функций fprintf, printf, sprintf и cprintf выполняется форматный вывод, соответственно, в указанный выходной файл (или в стандартный выходной файл stdout), в указанную строку или на терминал.

Рассмотрим интерфейсы с указанными функциями:

– int fprintf(FILE*stream,char *format,<дополнительные аргументы>);

– int printf(char *format, <дополнительные аргументы>);

– int sprintf(char *s, char *format, <дополнительные аргументы>);

– int cprintf(char *format, <дополнительные аргументы>);

Каждая из описываемых функций в случае возникновения ошибки возвращает значение EOF, в противном случае она возвращает число символов, переданных по назначению.

Пример вывода надписи (+123.457):

```
/*fprintf.1*/
#include <stdio.h>
main ()
{
fprintf(stdout, "(%-+9.3f)", 123.4567);
}
```

Элемент 9 определяет ширину внешнего поля, элемент .3 – число цифр после десятичной точки, элементы – (минус) и + (плюс) указывают, соответственно, что число в выходном поле выравнивается влево и что числу предшествует знак «плюс» или «минус».

Ф у н к ц и я ftell

Интерфейс с функцией ftell оформляется следующим образом:

```
long ftell(FILE *stream );
```

Результат функции – текущий указатель на файл.

Пример вывода числа символов, содержащихся в файле STDIO.H:

```
/*ftell*/
#include <stdio.h>
main ()
{
FILE *Inp;
```

```
Inp = fopen("D:\\tc\\include\\stdio.h", "r");
fseek(Inp, 0, SEEK_END);
printf("%ld", ftell(Inp));
}
```

Ф у н к ц и и scanf, fscanf, sscanf и cscanf

Интерфейс с функциями scanf, fscanf, sscanf и cscanf имеет вид:

– int scanf(char *format, <дополнительные аргументы>);

– int fscanf(FILE *stream,char *format,<дополнительные аргументы>);

– int sscanf(char *string,char *format,<дополнительные аргументы>);

– int cscanf(char *format, <дополнительные аргументы>).

С помощью функции scanf осуществляется ввод из стандартного файла stdin. Функция fscanf вводит данные из файла, указанного пользователем. Функция sscanf вводит данные из заданной строки. Функция cscanf вводит данные с консоли.

Все функции семейства scanf вводят поля символ за символом, переводя их в соответствии с указанным форматом.

Пример ввода из стандартного входного файла последовательности символов в виде числа и вывод информации о его значении:

```
/*fscanf.1*/
#include <stdio.h>
main ()
{
int Val = -13;
fscanf(stdin, "Val = %d", &Val);
printf("\n\n Val = %d",Val);
}
```

Числу должна предшествовать надпись Val=. Перед словом Val, а также с обеих сторон от знака равенства могут находиться интервалы. Если входная последовательность имеет другой вид, то результатом выполнения программы будет вывод числа -13.

Ф у н к ц и я setbuf

Интерфейс с функцией setbuf выглядит следующим образом:

```
void setbuf( FILE *stream, char *buffer );
```

С помощью этой функции системе ввода/вывода предлагается для выполнения обменов использовать буфер, указанный в качестве параметра функции, вместо буфера, подключаемого автоматически. Если указан нулевой буфер, то ввод/вывод будет небуферизированным. В противном случае он будет полностью буферизироваться. Размер буфера должен быть равен BUFSIZ байтам (соответствующая константа описана в файле stdio.h).

Если обмены не буферизируются, то символы передаются потребителю напрямую, без предварительного накопления.

Обращаем внимание на то, что следует обязательно закрывать файлы, работа с которыми ведется через динамически определяемые буферы пользователя. Закрытие рекомендуется осуществлять при выходе из функции, в которой такие буфера заводятся.

Пример вывода в файл WORK символа * (звездочка) (это действие выполняется несмотря на аварийное завершение выполнения программы):

```
/*setbuf*/
#include <stdio.h>
main ()
{
FILE *Out;
Out = fopen("WORK","w");
setbuf(Out, NULL);
putc('*',Out);
abort();
}
```

1. Что является признаком массива при описании? **2.** К указателям какого типа нельзя применять операцию разыменования? **3.** Какие операции над указателями допустимы в языке C++? **4.** Что такое ссылка? **5.** В массиве из 20

целых чисел найдите наибольший элемент и поменяйте его местами с первым элементом. **6.** В массиве из 10 целых чисел найдите наименьший элемент и поменяйте его местами с последним элементом. **7.** Сформируйте одномерный массив случайным образом и определите, есть ли в нем элементы с одинаковыми значениями. Выведите их на экран. **8.** Введите с клавиатуры двумерный массив. Замените строки столбцами и выведите исходный и полученный массивы.

Глава 7
СЛОЖНЫЕ МОДЕЛИ ДАННЫХ

7.1. **Линейные ссылочные структуры**

Теория структур данных предоставляет широкий спектр моделей для построения таких повторно используемых компонентов программного обеспечения, как стеки, очереди, списки и деревья. Благодаря тому, что перечисленные структуры данных имеют множество возможных применений, они являются важнейшими фундаментальными понятиями в программной инженерии.

7.1.1. *Стек*

Стек часто называют структурой типа «первым вошел – последним вышел».

Базовыми операциями со стеком являются:

– push – добавить в стек новый элемент;

– pop – удалить из стека последний элемент;

– peek – считать элемент с «верхушки» стека, не изменяя при этом весь стек.

Максимальное число элементов, которые можно разместить в стеке, не должно лимитироваться программным окружением. По мере добавления в стек новых элементов и удаления старых память под стек должна динамически дозапрашиваться и освобождаться.

Рассмотрим интерфейс с пакетом программ работы со стеком (предполагается, что базовый тип элементов такого стека – целый):

```
/*Интерфейс со стеком*/
/*stack.h*/
#define STACK struct stack
STACK {
 int info;
 STACK *next;
};
extern void push(STACK **s, int item);
extern int pop(STACK **s, int *error);
extern int peek(STACK **s, int *error);
```

На примере описания структуры STACK видно, что можно описывать не все ее поля. Второе поле структуры STACK – указатель на структуру STACK. Такой способ рекурсивного определения, при котором структура содержит ссылку на саму себя, является вполне допустимым. Компилятор отводит под указатель next требуемое количество памяти независимо от того, на какой объект этот указатель ссылается.

В функциях push, pop и peek используются двойные ссылки. Благодаря этому каждая из таких функций может возвращать в качестве результата своей работы указатель на новый элемент типа STACK (используется передача параметров по ссылке). Входным параметром указанных функций является адрес указателя на стек, с использованием которого вычисляется и возвращается новый адрес элемента. После каждой операции добавления или удаления указатель на связанный список меняется.

Функции pop и peek имеют параметр error целого типа, передаваемый по ссылке. Если в стеке имеется хотя бы один элемент, то функции выдают *error=0. Если стек пуст, а следовательно, бессмысленно удалять или считывать что-либо из стека, то *error принимает значение 1.

Реализация функций работы со стеком:

```
/*Реализация стека*/
/*stack.h*/
#include <alloc.h>
#include "stack.h"
```

```
void push (STACK **s, int item)
{
 STACK *new_item=(STACK*)malloc(sizeof(STACK));
 new_item->info=item;
 new_item->next=*s;
 *s=new_item;
}
int pop(STACK **s, int *error)
{
 /* *error = 0, если операция POP выполнена успешно, иначе = 1 */
 STACK *old_item=*s;
 int old_info=0;
 if (*s)
 {
 old_info=old_item->info;
 *s=old_item->next;
  free(old_item);
  *error=0;
 }
else
 *error =1 ;
return (old_info);
}
int peek (STACK **s, int *error)
{
 /**error = 0 , если в стеке не меньше одного элемента,
  *error = 1 , если в стеке нет элементов
*/
if (*s)
{
 *error=0;
 return(*s)->info;
 }
else
{
*error =1;
 return 0;
 }}
```

Внимательно изучим текст функции push.

Память под новый элемент запрашивается из кучи с помощью первого оператора:

```
STACK *new_item=(STACK*)malloc(sizeof(STACK));
```

Преобразование типа (STACK *) нужно для того, чтобы привести указатель, выдаваемый функцией malloc, к виду, при котором он может ссылаться на STACK. Для вычисления числа байтов, требуемых для размещения структуры, служит встроенная функция size of.

В следующей строке программы

```
new_item->info=item;
```

полю info структуры присваивается значение переменной item. Операция снятия ссылки -> использована здесь потому, что new_item – это указатель на тип STACK.

Далее при помощи оператора

```
new_item->next=*s;
```

полю next присваивается значение адреса «верхушки» связанного списка *s.

И, наконец, с помощью оператора

```
*s=new_item;
```

указателю *s присваивается новое значение адреса «верхушки» списка new_item, и этот указатель возвращается в качестве результата функции.

Функция pop, перед тем как сделать попытку обратиться к списку, проверяет значение *s. Если значение равно 0, то список пуст, и в этом случае выражению *error присваивается значение 1.

Тестовая программа, иллюстрирующая работу со стеком:

```
/*Тестовая программа, использующая переменные типа стек*/
```

```
#include <stdio.h>
#include "stack"
STACK *s1,*s2;
main()
{
int error;
push(&s1,12);
printf("\npeek(s1)=%d",peek(&s1,&error));
push(&s1,13);
printf("\npeek(s1)=%d",peek(&s1,&error));
push(&s1,14);
printf("\npeek(s1)=%d",peek(&s1,&error));
push(&s1,15);
printf("\npeek(s1)=%d",peek(&s1,&error));
 push(&s2,pop(&s1,&error));
 push(&s2,pop(&s1,&error));
 push(&s2,pop(&s1,&error));
 push(&s2,pop(&s1,&error));
printf("\npop(&s2)=%d",pop(&s2,&error));
printf("\npop(&s2)=%d",pop(&s2,&error));
printf("\npop(&s2)=%d",pop(&s2,&error));
printf("\npop(&s2)=%d\n",pop(&s2,&error));
}
```

Существенно, что стековые переменные s1 и s2 типа указатель описаны как глобальные. Это гарантирует, что каждая переменная будет инициализирована нулем. Если бы переменные были описаны как локальные, то начальные значения их были бы не определены, что, в свою очередь, могло бы привести к ошибке в функциях pop и peek.

В тестовой программе не используется значение переменной error.

Результат работы программы выглядит следующим образом:

– peek(s1) = 12;
– peek(s1) = 13;
– peek(s1) = 14;
– peek(s1) = 15;
– pop(&s2) = 12;

– pop(&s2) = 13;
– pop(&s2) = 14;
– pop(&s2) = 15.

7.1.2. *Очередь*

Очередью называют структуру данных, организованную по принципу «первым вошел – первым вышел».

Базовыми операциями над очередью являются:

– insert – добавить в очередь новый элемент;

– take_out – удалить из очереди первый элемент.

Как и для стека, максимальное число элементов в очереди не должно лимитироваться используемым программным обеспечением. Память для очереди должна запрашиваться и освобождаться динамически по мере того, как в очередь добавляются и из очереди удаляются элементы.

Предполагается, что базовым типом элементов очереди является тип int.

Интерфейс с функциями обслуживания очереди:

```
/*Интерфейс с пакетом программ работы с очередью*/
/*Файл queue.h*/
#define QUEUE struct queue
QUEUE
{
int info;
QUEUE *next;
};
extern void insert(QUEUE **q, int item);
extern int take_out(QUEUE **q, int *error);
```

Определяется структура QUEUE. Здесь же описываются интерфейсы с функциями insert и take_out.

Функция take_out имеет параметр *error, передаваемый по ссылке. Значение параметра равно нулю, если в очереди содержится

один или более элементов (целых чисел), и равно единице, если очередь пуста.

Ниже приведены подпрограммы, реализующие работу с очередью:

```
/*Реализация очереди*/
/*Файл queue.c*/
#include <alloc.h>
#include "queue.h"
void insert (QUEUE **q, int item)
{
QUEUE *current=*q;
QUEUE *previous=0;
QUEUE *new_node;
while (current)
{
 privious=current;    /*privious=*q=new_node_1*/
 current=current->next; /*current=(*q)->next=0*/
 }
new_node=(QUEUE*)malloc(sizeof(QUEUE));
new_node ->info=item;
if (previous)
{
 new_node->next=previous->next; /* = 0 */
 previous->next=new_node;
 }
else
 {
 *q=new_node;   /* *q=new_node_1*/
 (*q)->next=0;  /*new_node_1->next=0*/
 }
}
int take_out(QUEUE **q, int *error)
{
int value=0;
QUEUE *old_header=*q;
if (*q)
  {
  value=old_header->info;
  *q=old_header->next;
```

```
  free(old_header);
  *error=0;
  }
  else *error=1;
  return value;
}
```

Рассмотрим подробно функцию insert.

Описывается локальная переменная current типа указатель и инициализируется значением *q. Описывается локальная переменная previous типа указатель и инициализируется значением 0.

В цикле типа WHILE связанный список просматривается до конца. Указатель previous содержит адрес последнего элемента QUEUE в списке. Затем отводится память под очередь QUEUE new_node, полю info новой структуры QUEUE присваивается значение item, переменной previous присваивается значение указателя на new_node, а new_node устанавливается в 0, если очередь не пуста, и принимает значение указателя на новый элемент в голове очереди, если очередь прежде была пустой.

Код для функции take_out сходен с кодом для функции pop.

Тестовая программа, использующая очередь:

```
/*Тестовая программа, использующая переменные типа очередь*/
/*Файл queue.c*/
#include <sdtio.h>
#include <alloc.h>
#include "queue.h"
QUEUE *q1,*q2;
main()
{
 int error;
 insert(&q1,12);
 insert(&q1,13);
 insert(&q1,14);
 insert(&q1,15);
 insert(&q2,take_out(&q1,&error));
 insert(&q2,take_out(&q1,&error));
 insert(&q2,take_out(&q1,&error));
```

```
 insert(&q2,take_out(&q1,&error));
 printf("\nremove(&q2)=%d",take_out(&q2,&error));
 printf("\nremove(&q2)=%d",take_out(&q2,&error));
 printf("\nremove(&q2)=%d",take_out(&q2,&error));
 printf("\nremove(&q2)=%d\n",take_out(&q2,&error));
}
```

Результатом ее работы будет:

– remove(&q2) = 12;
– remove(&q2) = 13;
– remove(&q2) = 14;
– remove(&q2) = 15.

7.1.3. *Связанные списки*

Списки являются весьма популярными структурами и применяются для представления абстрактного аппарата поиска элемента. Элементы в списках обычно располагаются в возрастающем или убывающем порядке.

Стеки и очереди являются специальными разновидностями обобщенных списков.

Простой список может быть определен при помощи следующих операций:

– insert – добавить новый элемент в список, сохраняя установленный порядок следования;

– take_out – удалить элемент из списка, сохранив установленный порядок следования. Если элемент отсутствует, то функция не выполняет никаких действий;

– is_present – определить, содержится ли в списке заданный элемент. Если содержится, то возвращается значение «единица», в противном случае возвращается значение «нуль»;

– display – вывести по порядку все элементы списка;

– destory – освободить память, занимаемую списком.

Как для стека и очереди, так и для связанного списка не должно существовать ограничения на максимальное количество его элементов, налагаемого программным окружением. Память под список

должна динамически запрашиваться и освобождаться по мере того, как элементы добавляются и удаляются.

Существует множество способов реализовать связанный список.

Это однонаправленные списки, закольцованные списки, двунаправленные списки и т.д.

Рассмотрим пример реализации простого однонаправленного списка:

```
/*Интерфейс с программами работы со связанным списком персонала
университета*/
/*Файл list.h*/
#define STAFF struct stuff_type
STAFF
{
 int years_of_service;
 float hourly_wage;
 };
#define STUDENT struct student_type
STUDENT
{
 float grade_pt_average;
 int level;
};
#define PROFESSOR struct prof_type
PROFESSOR
{
 int dept_number;
 float annual_salary;
};
#define NODE_TYPE enum node_type
typedef NODE_TYPE {student,professor,staff};
#define LIST struct list
LIST
{
 char last_name[10];
 char first_name[10];
 int age;
 LIST *next;
 NODE_TYPE tag;
```

```
union
{
STUDENT student;
PROFESSOR professor;
STAFF staff;
} node_tag;
};

extern void insert(LIST **lst, LIST *item);
extern void display(LIST *lst);
extern int is_present(LIST *lst, LIST *item);
extern int take_out(LIST **lst, LIST *item);
extern void destroy_list(LIST **lst);
```

Список содержит вложенные структуры и вложенные объединения. Возможности здесь безграничны!

Вначале определяются структуры STAFF, STUDENT и PROFESSOR, содержащие каждая по два числа (поля). Далее описывается перечисляемый тип NODE_TYPE. С помощью этого типа будет описываться поле списочной структуры, указывающее на тип элемента поля.

Списочная структура (LIST) включает пять обычных полей: last_name (последнее имя), first_name (первое имя), age (возраст), next (следующее) и tag (флаг), а также объединение node_tag. Это объединение состоит из трех членов: student (студент), professor (профессор) и staff (персонал). Каждая переменная типа LIST может содержать элементы только одного типа из объединения. Такая структура, похожая на запись с вариантами, позволяет экономно расходовать память, поскольку она занимает пространство, требуемое для размещения первых пяти обычных элементов структуры, плюс пространство, занимаемое элементом максимальной длины из объединения.

В файле list.h кроме описания структуры данных для списка задан еще и интерфейс с пятью функциями: insert, display, is_present, take_out и destroy_list.

Реализация программ работы со связанными списками:

```
/*Файл list.c*/
#include <stdio.h>
#include <string.h>
#include <alloc.h>
#include "list.h"
static LIST* create_node(LIST *item)
{
 LIST *node;
 node=(LIST*)malloc(sizeof(LIST));
 *node =*item;
 return node;
}
void destroy_list(LIST **lst)
{
 LIST *current_node=*lst;
 LIST *previous_node =0;
 while (current_node !=0)
 {
 previous_node =current_node;
 current_node=current_node->next;
 free(prefious_node);
 }
 *lst=0;
 }
int take_out(LIST **lst, LIST *item)
{
 LIST *current_node=*lst;
 LIST *previous_node=0;
 while (current_node !=0 &&
    strcmp(current_node->last_name,item->last_name)!=0)
 {
 previous_node=current_node;
 current_node=current_node->next;
 }
 if (current_node !=0 && previous_node ==0 )
  {
  *lst=current_node->next;
  free(current_node);
}
else if (current_node !=0 && previous_node!=0)
```

```
      {
      previous_node->next=current_node->next;
      free(current_node);
      }
}
void insert (LIST **lst, LIST *item)
{
 /* Фамилия используется как ключевое поле в списке */
 char key[10];
 LIST *current_node=*lst;
 LIST *previous_node=0;
 LIST *new_node;

 strcpy(key,item->last_name);
 while (current_node !=0 && strcmp(current_node->last_name,
    key)<0)
 {
 previous_node=current_node;
 current_node=current_node->next;
 }
new_node=create_node(item);
new_node->next=current_node;
if (previous_node==0)
  *lst=new_node;
else previous_node->next=new_node;
}
void display (LIST *lst)
{
 LIST *current=lst;
 while (current)
 {
 printf("\n%s, %s",current->last_name,current->first_name);
 printf("\n Возраст =%d"current->age);
 swith (current->tag)
 {
 case student: printf("\nСпециализация: %.2f",
          current->node_tag.student.grage_pt_average);
        printf("\nКурс: %d\n",
          current->node_tag.student.level);
 break;
```

```
 case professor: printf ("\nНомер кафедры: %d",
           current->node_tag.professor.dept_number);
         printf ("\nОклад: $%.2f\n",
           current->node_tag.professor.annual_salary);
 break;
 case staff: printf ("\nСрок службы: %d",
          current->node_tag.staff.years_of_service);
       printf ("\nПочасовой тариф: $%.2f\n",
          current->node_tag.staff.hourly_wage);
 }
 current=current->next;
 }
}
int is_present(LIST *lst, LIST *item)
{
/* Фамилия используется как ключевое поле в списке */
LIST *current_node=lst;
while (current_node && strcmp(item->last_name,current_node->
    last_name)!=0)
   if (strcmp(item->last_name,current_node->last_name)!=0)
     current_node=current_node->next;
     return(current_node !=0);
}
```

Функция create_node (создать элемент), описанная как

```
static LIST* create_node(LIST *item)
```

спрятана от загрузчика и других программ системы, поскольку задана статически. Ее назначение – отводить память под новый элемент списка и передавать данные, помещенные по адресу item (указатель на структуру), в новый элемент.

В функции используется множественное присваивание:

```
*node =*item;
```

для пересылки всей информации, расположенной по адресу item, в область памяти по адресу *node.

Цикл типа WHILE в функции destroy_list использован для освобождения памяти, занимаемой элементами списка. После завершения цикла WHILE значение переменной *lst, передаваемой по ссылке, оказывается равным нулю.

В функции take_out в цикле типа WHILE перебираются элементы списка до тех пор, пока либо не будет достигнут конец списка, либо не будет установлено соответствие между строкой в поле last_name переменной item и полем last_name в текущем элементе current_node. Если цикл прекращается, а элемент previous_node равен нулю, то изменяется указатель на голову списка *lst, и память, занимаемая старым головным элементом списка, освобождается (исключается всегда элемент, находящийся в голове списка). В противном случае операторы

```
previous_node->next=current_node->next;
free(current_node);
```

продвигают указатель по списку и освобождают память из-под удаляемого элемента.

В функции insert указатель LIST current_node инициализируется значением *lst и переменная previons_node инициализируется нулем. В цикле типа WHILE просматривается список либо до конца, либо до тех пор, пока значение поля last_name переменной current_node не перестанет быть меньше, чем поле last_name добавляемого элемента, поскольку список упорядочен по возрастанию.

Динамически запрашивается память под указатель new_node, и данные из элемента item пересылаются по адресу new_node. Новый элемент new_node подключается к списку при помощи оператора

```
new_node->next=current_node;
```

Если добавляемый элемент является первым в первоначально пустом списке (previous_node равен нулю), то указатель на голову списка *lst устанавливается на new_node. Иначе указатель продвигается по списку при помощи операторов

```
previous_node->next=new_node;
```

Самая большая программа требуется для функции display, поскольку ее работа существенно зависит от типа выводимого элемента.

После того как будут выведены поля last_name, first_name и age, при помощи переключателя определяется, как выводить остальные поля. Предположим, что значение поля tag есть student, т.е. мы должны распечатать информацию о студенте. Попробуйте разобраться в том, каким образом осуществляется доступ к глубоко вложенной информации grade_pt_average.

Тестовая программа, использующая большинство функций из пакета программ работы со списками, имеет вид:

```
#include <string.h>
#include <stdio.h>
#include <alloc.h>
#include "list.h"
LIST *my_list;
main()
{
 LIST *item1=(LIST*)malloc(sizeof(LIST));
 LIST *item2=(LIST*)malloc(sizeof(LIST));
 LIST *item3=(LIST*)malloc(sizeof(LIST));
 strcpy(item1->last_name,"Smith");
 strcpy(item1->first_name,"Robert");
 item1->age=26;
 item1->tag=staff;
 item1->node_tag.staff.years_of_service=3;
 item1->node_tag.staff.hourly_wage=5.25;
 insert(&my_list,item1);
  strcpy(item2->last_name,"Jones");
  strcpy(item2->first_name,"Richard");
 item2->age=56;
 item2->tag=professor;
 item2->node_tag.professor.dept_number=7;
 item2->node_tag.professor.annual_salary=48321.0;
 insert(&my_list,item2);
  strcpy(item3->last_name,"Evans");
```

```
 strcpy(item3->first_name,"Henry");
item3->age=19;
item3->tag=student;
item3->node_tag.student.level=1;
item3->node_tag.student.grade_pt_average=3.1;
insert(&my_list,item3);
 display(my_lst);
 if (is_present(my_list,item1))
  printf("\nЭлемент 1 присутствует в списке.\n");
else
  printf("Элемент 1 отсутствует в списке.\n");
take_out(&my_list,item3);
display(my_list);
take_out(&my_list,item2);
display(my_list);
take_out(&my_list,item1);
display(my_list);
printf("\n");
}
```

Сначала инициализируются три указателя на списки: item1, item2 и item3.

В список добавляются три структуры, значения полей которых задаются в программе. Программа выведет следующий текст:

```
Evans, Henry
Возраст=19
Специализация: 3.10
Курс: 1
Jones, Richard
Возраст=56

Номер кафедры: 7
Оклад: $48321.00

Smith, Robert
Возраст=26
Стаж: 3
Почасовой тариф: $5.25
```

```
Элемент 1 присутствует в списке.

Jones, Richard
Возраст=56
Номер кафедры: 7
Оклад: $48321.00

Smith, Robert
Возраст=26
Стаж: 3
Почасовой тариф: $5.25

Smith, Robert
Возраст=26
Стаж: 3
Почасовой тариф: $5.25
```

7.2. **Обобщенные (родовые) структуры данных**

Базовые операции определения абстракций списка и дерева поиска не зависят от типа данных, содержащихся в структуре списка или дерева. Поэтому такие абстракции являются идеальными кандидатурами для их обобщенной (родовой) реализации.

Родовой пакет структур данных при условии его эффективной реализации может стать повторно используемым программным компонентом. Программисту не придется для каждого нового используемого типа данных создавать свою реализацию структуры.

7.2.1. *Родовой список*

Рассмотрим интерфейс с программами, реализующими родовой список с однотипными элементами.

```
/*Файл genlist.h*/
#define LIST struct header
#define NODE struct node
typedef void (*display_function) (char *data);
```

```
LIST
{
 NODE *next;
 int elem_size;
 display_function display;
};
NODE
{
 char *info;
 NODE *next;
};
extern void define (LIST **lst,int size,
        display_function disp_fun);
extern void insert_front (LIST **lst, char *data);
extern void insert_back (LIST **lst, char *data);
extern char *get_front(LIST **lst);
extern void display_list(LIST *lst);
```

Определяются структуры LIST и NODE.

Структура LIST является заголовком связного списка. Структура включает три поля: next, elem_size и display. Поле next содержит указатель на первый элемент списка, имеющий тип NODE. При инициализации поле получает значение нулевого указателя (0). Размер каждого элемента списка в байтах помещается в поле elem_size.

Поле display содержит указатель на определяемую пользователем функцию, с помощью которой можно вывести значение элемента. Эта функция вывода приписывается заголовку списка при вызове функции define.

Пользователь должен сам позаботиться о разработке программы вывода значений элементов для каждого конкретного списка. Невозможно организовать вывод значений с помощью самого родового пакета, поскольку заранее не известен базовый тип элементов списка и, следовательно, не определена форма представления значений.

Структура NODE содержит поля info и next. Поле info описывается как указатель на минимальный предопределенный тип, а именно на тип char, данные в поле info должны передаваться байт

за байтом. Поле next указывает на следующий элемент списка типа NODE.

На рис. 7.1 показана структура данных родового связного списка.

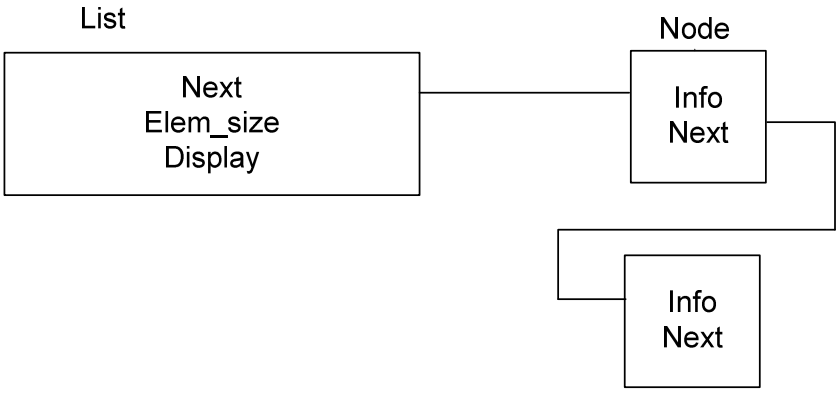

Рис. 7.1. Родовой связный список

Далее приведен интерфейс с функциями define, insert_front, insert_back, get_front и display_list.

В предлагаемом простом родовом пакете программ для работы со списками допускаются только два способа включения новых элементов в список: добавление в начало списка (функция insert_front) и добавление в конец списка (insert_back). Удалить элемент из списка можно единственным способом, а именно исключив элемент из начала списка.

Перед началом работы с пакетом следует обратиться к функции define. С помощью этой функции в заголовок списка заносится размер элемента списка в байтах (поле elem_size) и указатель на программу вывода значений, определяемую пользователем (поле display).

Реализация программ работы с родовым списком:

```
/*Файл genlist.h*/
#include "genlist.h"
#include <alloc.h>
void define (LIST **lst,int size, display_function disp_fun)
{
```

```
 *lst=(LIST*)malloc(sizeof(LIST));
 (*lst)-> next = 0;
 (*lst)->elem_size = size;
 (*lst)->display = disp_fun;
}
void insert_front (LIST **lst,char *data)
{
 NODE *new_node;
 int index;
 /*Отводится память для структуры new_node и ее полей*/
 new_node = (NODE*)malloc(sizeof(NODE));
 new_node->info=(char*)malloc((*lst)->elem_size);
 /*Пересылка данных в элемент new_node*/
 for (index=0;index<(*lst)->else_size;index++)
 new_node ->info[index]=data[index];
 /*Присоединение элемента new_node к списку*/
 if ((*lst)->next) new_node->next=(*lst)->next;
 else
 new_node ->next=0;
 (*lst)->next=new_node;
}
void insert_back (LIST **lst,char *data)
{
 NODE *previous,*current,*new_node;
 int index;
 /*Отводится память для структуры new_node и ее полей*/
 new_node=(NODE*)malloc(sizeof(NODE);
 new_node->info =(char*)malloc((*lst)->elem_size);
 new_node->next=0;
 /*Пересылка данных в элемент new_node*/
 for (index=0;index<(*lst)->elem_size;index++)
 new_node->info[index]=data[index];
 if ((*lst)->next)
 {
 /*Поиск конца списка*/
 previous=(*lst)->next;
 current =previous->next;
 while (current!=0)
 {
  previous=current;
```

```
  current=current->next;
 }
 /*Присоединение элемента new_node к списку*/
 previous->next=new_node;
 }
else
 /*Элемент new_node является первым в списке*/
(*lst)->next=new_node;
}
char *get_front (LIST **lst)
{
 NODE *old_node=(*lst)->next;
 char *ret_info;
 int index;
 if ((*lst)->next==0)
 {
 printf("\nОшибка: список пуст");
 return 0;
 }
else
{
 ret_info=(char*)malloc((*lst)->elem_size);
 for (index=0;index<(*lst)->elem_size;index++)
   ret_info[index]=old_node->info[index];
 (*lst)->next=old_node->next;
 free(old_node);
 return ret_info;
 }
}
void display_list (LIST *lst)
{
 NODE *current=lst->next;
 while (current)
 {
 lst->display(current->info);
 current=current->next;
 }
}
```

Как уже отмечалось, с помощью функции define с заголовком списка связывается соответствующая функция вывода значений

элемента и задается размер элемента списка. Используя передачу параметра по ссылке, первый оператор функции define

```
*lst=(LIST*)malloc(sizeof(LIST));
```

возвращает адрес нового списка.

Следующие три оператора инициируют список и заносят в соответствующие поля заданные пользователем значения elem_size и disp_fun.

В программе, реализующей функцию insert_back, сначала отводится память для указателя new_node. Далее отводится память для одного из полей этой структуры – поля new_node->info. Число байтов, отведенных по адресу new_node->info, равно значению elem_size. В первом операторе адрес, полученный от функции malloc для структуры new_node, преобразуется к виду указателя на тип NODE. Во втором операторе адрес, полученный от функции malloc для поля new_node->info, преобразуется в указатель на тип char. Далее полю next структуры *new_node присваивается значение «нуль».

Затем осуществляется побайтовая пересылка данных из массива data в поле new_node->info. Функции insert_back не известен тип информации, которая заносится в список, но, несмотря на это, пересылка информации все-таки осуществляется.

Если создается уже не первый элемент в списке, то для перебора элементов используется цикл WHILE. По достижении конца списка элемент new_node присоединяется к списку.

Если в список заносится первый элемент, то в поле next структуры *lst заносится указатель на структуру new_node.

Функция get_front возвращает указатель на значение типа char. В функции отводится память для возвращаемой информации, и адрес такой информации передается в виде указателя на тип char.

Если список не пустой, то по адресу ret_info отводится elem_size байтов. Затем осуществляется побайтовая пересылка информации из поля old_node->info в массив ret_info. Изменяются ссылки между элементами списка, и в качестве результата функции возвращается адрес массива ret_info.

В функции display_list осуществляется обход списка, и вызывается функция display:

```
lst->display(current->info);
```

Тестовая программа построения двух списков, использующая пакет программ работы с родовыми списками:

```
/*Программа работает с родовым списком*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "genlist.h"
LIST *number_list;
LIST *record_list;
#define RECORD struct record
RECORD
{
 char name[30];
 float id;
};
main ()
{
 extern void display_number(char *info);
 extern void display_record(char *info);
 {
 int item;
 int *value;
 define (&number_list,sizeof(item),display_number);
 item=4;
 insert_front(&number_list,(char*)&item);
 item=5;
 insert_front(&number_list,(char*)&item);
 item=6;
 insert_front(&number_list,(char*)&item);
 item=-12;
insert_back(&number_list,(char*)&item);
value =(int*)get_front(&number_list);
printf("\nэлемент=%d",*value);
display_list(number_list);
```

```
}
{
 RECORD *item=(RECORD*)malloc(sizeof(RECORD));
 RECORD *value;
 define(&record_list,sizeof(RECORD),display_record);
 strcpy(item->name,"Aaaaa");
 item->id=1.2345;
 insert_front(&record_list,(char*)item);
 strcpy(item->name,"Bbbbb");
 item->id=2.2345;
 insert_front(&record_list,(char*)item);
 strcpy(item->name,"Ccccc");
 item->id=3.2345;
 insert_front(&record_list,(char*)item);
 strcpy(item->name,"Ddddd");
 item->id=4.2345;
 insert_back(&record_list,(char*)item);
 value=(RECORD*)get_front(&record_list);
 printf("\n\nИмя %s",value->name);
 display_list(record_list);
 }
printf("\n");
}
void display_number(char *info)
{
 printf("\n%d",*((int*)info));
}
void display_record(char *info)
{
 RECORD *temp=(RECORD*)info;
 printf("\nИмя -> %s",temp->name);
 printf("\nНомер ->%f",temp->id);
}
```

Определяются два списка: number_list и record_list. Каждый элемент списка record_list содержит поле name (строка длиной до 30 символов) и поле id (число с плавающей точкой).

В тестовой программе определяются две функции вывода значений display_number и display_record.

В функции display_number используется преобразование типа (int *), с помощью которого вначале результат переводится в тип указателя на тип int, который, в свою очередь, служит для вывода значения элемента списка как целого числа.

В функции display_record определяется локальная переменная temp, которая инициализируется значением (RECORD*). Результатом работы функции будут значения полей элементов списка temp->info и temp->id.

Функция main состоит из двух блоков. В первом блоке в список заносится несколько элементов: сначала с помощью функции insert_front, а затем с использованием функции insert_back. Далее из списка удаляется элемент с помощью оператора

```
value =(int*)get_front(&number_list);
```

Функция get_front возвращает указатель на значение типа char. Такой адрес преобразуется к виду указателя на значение типа int. Затем по адресу выбирается значение, и оно печатается.

И наконец, осуществляется вывод списка number_list.

Аналогичные действия выполняются и для второго списка record_list.

Программа выведет:

```
Элемент = 6
5
4
-12
Имя Ccccc
Имя -> Bbbbb
Номер -> 2.2345000
Имя -> Aaaaa
Номер -> 1.2345000
Имя -> Ddddd
Номер -> 4.2345000
```

7.2.2. *Родовое дерево поиска*

Деревья относятся к наиболее важным структурам данных, используемым в информатике. Деревья применяются при синтаксическом анализе, поиске, сортировке, управлении базами данных, в игровых алгоритмах и других важных сферах приложений.

Дерево представляет собой конечное множество элементов, каждый из которых может быть либо пустым, либо содержать корневой узел и, возможно, другие узлы. Эти узлы можно разделить на два непересекающихся подмножества, каждое из которых само является двоичным деревом. Такие подмножества называются л е в ы м и п р а в ы м п о д д е р е в ь я м и. Каждый узел двоичного дерева может иметь 0, 1 или 2 поддерева. Если у узла нет поддеревьев (потомков), то он называется л и с т о м.

Будем предполагать, что каждый узел дерева имеет идентифицирующий его ключ.

Дерево поиска является разновидностью дерева, обладающей следующим свойством: все узлы дерева, лежащие левее и ниже данного узла, имеют значение ключа, меньшее, чем значение ключа данного узла, а все узлы, лежащие правее и ниже данного узла, имеют значение ключа, большее, чем значение ключа данного узла. Приведенное свойство позволяет эффективно организовать поиск по двоичному дереву.

На рис. 7.2. показано дерево, являющееся деревом поиска.

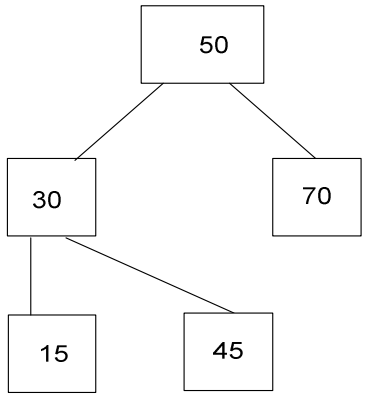

Рис. 7.2. Дерево поиска

На рис. 7.3 изображено дерево, для которого нарушено требование из определения дерева поиска. Корневой узел имеет два левых потомка, значения ключей у которых больше чем 30.

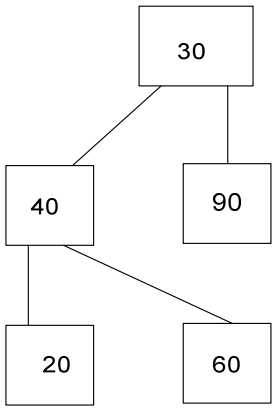

Рис. 7.3. Дерево, не являющееся
деревом поиска

Деревья поиска являются весьма важными с практической точки зрения, поскольку они относительно сбалансированы: число уровней дерева приблизительно равно двоичному логарифму от числа узлов. Например, относительно сбалансированное дерево с 1 000 000 узлов имеет лишь 20 уровней.

Алгоритмы, используемые для добавления, исключения и поиска элементов в дереве, зависят не от количества узлов, а от количества уровней дерева. Таким образом, нелинейная структура дерева существенно влияет на достижение эффективности вычислений.

Рассмотрим интерфейс с программами работы с родовым деревом поиска, все элементы которого имеют одинаковый тип:

```
/*Файл gentree.h*/
#define TREE struct tree
#define NODE struct node
typedef void (*display_function)(char *data);
typedef int (*less_than_function)(char *data1, char *data2);
typedef int (*equal_function)(char *data1, char *data2);
TREE
{
```

```
 NODE *next;
 int elem_size;
 display_function display;
 less_than_function less_than;
 equal_function equal; };
NODE
{
 char *info;
 NODE *left,*right;
};
extern void define (TREE **tree,int size,display_function disp,
        less_than_function lthan,equal_function eq);
extern void insert (TREE **tree ,char *item);
extern void take_out(TREE **tree,char *item);
extern void display_tree(TREE *root);
extern int is_present (TREE *root,char *item);
extern void destroy(TREE **root);
```

Определяются структуры TREE и NODE. Структура данных родового дерева изображена на рис. 7.4.

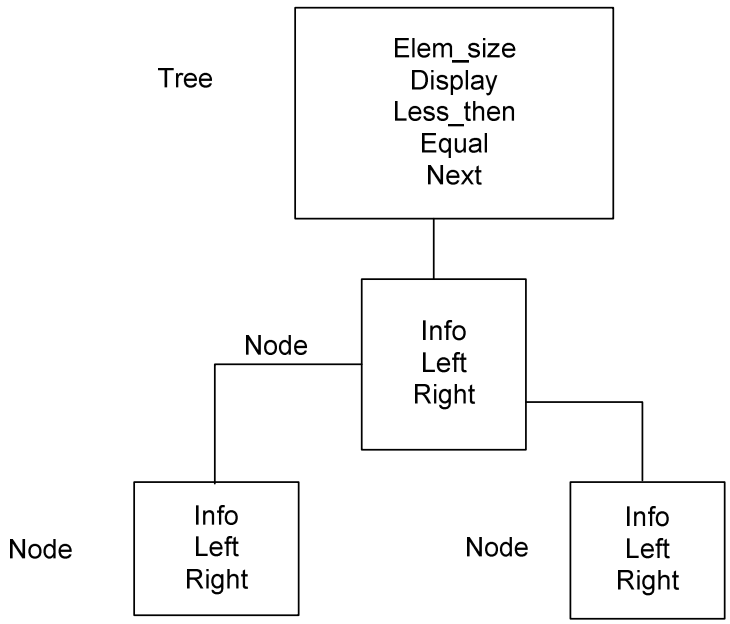

Рис. 7.4. Родовое дерево поиска

Определяются три типа, устанавливающие указатели на функции. Тип display_function задает указатель на функцию, имеющую один параметр data и возвращающую значение типа void. Тип less_than_function и equal_function задает указатель на функцию, использующую два параметра, data1 и data2, которые указывают на значения типа char и возвращают значение типа int.

Три рассмотренных определения типа являются прототипами для описываемых пользователем функций, используемых в функции define. Поскольку пакет программ для родового дерева не рассчитан на работу с каким-то конкретным базовым типом данных, то средствами пакета невозможно ни определить способ сравнения двух элементов базового типа, ни задать способ вывода значений элементов.

Функции disp, lthan и eq, определяемые пользователем, связываются с корневым узлом дерева с помощью функции define.

С применением этой же функции в корень дерева заносится и размер элемента в байтах. Для обеспечения целостности работы программ пакета вызовам функций должно предшествовать обращение к функции define.

Первым параметром каждой их трех функций define, insert и take_out является адрес указателя на структуру TREE. Любая из приведенных функций может что-либо изменить в корневом узле дерева. Так, функция define всегда изменяет корневой узел. Функции insert и take_out изменяют корневой узел, если только к дереву добавляется новый элемент или из дерева удаляется элемент.

Реализация программ работы с родовым деревом:

```
/*Файл gentree.c*/
#include <string.h>
#include <alloc.h>
#include "gentree.h"
static NODE* create_node (char *item,int size)
{
 NODE *node;
 int i;
 node=(NODE*)malloc(sizeof(NODE));
 node->info = (char*)malloc(size);
```

```
/* выполняется побайтная пересылка */
for (i=0;i<size;i++)
  node ->info[i]=item[i];
  return node;
}
static void post_order(NODE *current)
{
 if (current)
 {
 post_order(current->left);
 post_order(current->right);
 free(current->info);
 free(current);
 }
}
void destroy (TREE **root)
{
 NODE *current=(*root)->next;
 post_order(current);
 (*root)->next=0;
}
void define (TREE **tree,int size,
      display_function disp,
      less_than_function lthan,
      equal_function eq)
{
 *tree=(TREE*)malloc(sizeof(TREE));
 (*tree)->next=0;
 (*tree)->display=disp;
 (*tree)->less_than=lthan;
 (*tree)->elem_size=size;
 (*tree)->equal=eq;
}
int take_out (TREE **root,char *item)
{
NODE *previous=0,*present=(*root)->next,*replace,*s,*parent;
int found=0;
while (present && !found)
{
 if ((*root)->equal(present->info,item)) found=1;
```

```
 else
 {
 previous=present;

 if ((*root)->less_than(item,present->info))
    present=present->left;
   else present=present->right;
 }
}
if (found)
{
 if (present->left==0) replace=present->right;
  else if (present->right==0) replace=present->left;
     else
      {
       parent=present;
       replace=s;
       s=replace->left;
       while (s!=0)
      {
       parent=replace;
       replace=s;
       s=replace->left;
      }
      if (parent!=present)
      {
      parent->left=replace->right;
      replace->right=present->right;
      }
      replase->left=present->left;
      }
 if (present==0) (*root)->next=replace;
   else if (present==previous->left) previous->left=replace;
        else
          previous->right=replace;
 free(present->info);
 free(present);
 }
}
void insert (TREE **root,char *item)
```

```
{
 NODE *parent=0,*current=(*root)->next;
 NODE *new_node;
 int found=0;
while (current && !found)
{
 if ((*root)->equal(current->info,item)) found=1;
  else
  {
  parent=current;
  if ((*root)->less_than(item,current->info))
     current=current->left;
    else current = current->right;
  }
}
if (found==0)
{
 if (parent==0)
/* первый узел дерева */
{
 new_node=create_node(item,(*root)->elem_size);
 new_node->left=new_node->right=0;
 (*root)->next=new_node;
}
else
{
 new_node=create_node(item,(*root)->elem_size);
 new_node->left=new_node->right=0;
 if ((*root)->less_than(item,parent->info))
   parent->left=new_node;
  else parent->right=new_node;
  }
}
}
static void traverse(NODE* current,display_function display)
{
 if (current)
  {
  traverse(current->left,display);
  (*display)(current->info);
```

```
  traverse(current->right,display);
  }

}
void display_tree(TREE *root)
{
NODE *current=root->next;
traverse(current,root->display);
}
int is_present(TREE *root,char *item)
{
NODE *current=root->next;
int found=0;
while (current && !found)
{
 if (root->equal(item,current->info)) found=1;
 else
 {
 if (root->less_than(current->info,item)) current=current->left;
   else current=current->right;
 }
}
return found;
}
```

Первая функция create_node невидима для загрузчика, так как снабжена спецификатором static. Обращение к этой функции выполняется из функции insert. Очевиден родовой характер реализации функции create_node.

Новый узел дерева описывается как объект типа NODE. Память отводится как для структуры узла, так и для информационного поля info. Пересылка данных из элемента в узел осуществляется побайтно. В качестве результата функции возвращается адрес нового узла.

В функции destroy описывается локальный указатель current, при этом ему присваивается значение поля next структуры *root. Затем вызывается рекурсивная функция post_order, параметром которой является указатель current. Обход дерева в глубину реализуется фрагментом программы:

```
if (current)
{
post_order(current->left);
post_order(current->right);
free(current->info);
free(current);
}
```

где для доступа к узлам применяется рекурсия, при которой всегда сначала обращаются к левому, а затем к правому потомку узла. Под доступом подразумевается, что сначала освобождается память для поля info структуры NODE, а только потом – для самой структуры NODE. При использовании алгоритма обхода дерева в глубину узлы удаляются от нижних к верхним, причем связи между узлами никогда раньше времени не разрываются.

В функции define отводится память под новый корневой узел. Затем в этот узел заносятся размер элемента в байтах (elem_size) и указатели на определяемые пользователем функции disp, lthan и eq.

В функциях take_out и insert для сравнения полей last_name заданной структуры используются определяемые пользователем функции equal и less_than.

Положение нового узла в дереве определяется по следующим правилам:

Значение ключа элемента, добавляемого к дереву, сравнивается со значением ключа корневого узла. Если ключ элемента меньше, чем ключ корневого узла, то нужно переместиться к левому потомку узла; в противном случае следует переместиться к правому потомку узла. Сравнения повторяются, причем каждый раз ключ элемента сопоставляется либо с ключом левого потомка, либо с ключом правого потомка. Если обнаруживается совпадение, то сравнения прекращаются; в противном случае сравнения продолжаются, пока не будет достигнут низ дерева.

После завершения обхода дерева указатель parent всегда указывает на узел, лежащий на один уровень выше, чем узел, на который

указывает указатель curent, и задает положение предка текущего узла. Если указатель current принимает нулевое значение (т.е. достигнут низ дерева), то вновь создаваемый узел подключается к родительскому узлу либо как левый, либо как правый потомок в зависимости от значения ключа узла.

В функции display_tree описывается локальный указатель current, которому присваивается значение адреса корневого узла дерева. Затем вызывается функция traverse. Вторым параметром функции traverse передается адрес определяемой пользователем функции display.

В функции travense, описанной как static, для обхода дерева вглубь используются операторы if (current). При обходе с помощью функции display, указатель на которую передается в качестве параметра, выводятся значения, содержащиеся в каждом узле.

В функции is_present в процессе обхода дерева вызывается определяемая пользователем функция equal. Если сравнение значений ключей с помощью функции equal показало их совпадение, то функция is_present возвращает 1, в противном случае результат функции – нуль.

Тестовая программа, использующая пакет программ работы с родовым деревом поиска:

```
#include <stdio.h>
#include <alloc.h>
#include <stdlib.h>
#include <string.h>
#include "gentree.h"
TREE *my_tree;
#define RECORD struct record
RECORD
 {
 char name[30];
 float id;
 };
main()
{
 extern void display_record(char *info);
```

```
 extern int less_than(char *item1,char *item2);
 extern int equal (char *item1,char *item2);
 RECORD *item=(RECORD)malloc(sizeof(RECORD));
 RECORD *value;
 define (&my_tree,sizeof(RECORD),display_record,less_than,equal);
 strcpy(item->name,"Ccccc");
 item->id=3.2345;
 insert(&my_tree,(char*)item);
 strcpy(item->name,"Eeeee");
 item->id=5.2345;
 insert(&my_tree,(char*)item);
 strcpy(item->name,"Aaaaa");
 item->id=1.2345;
 insert(&my_tree,(char*)item);
 strcpy(item->name,"Ddddd");
 item->id=4.2345;
 insert(&my_tree,(char*)item);
 strcpy(item->name,"Fffff");
 item->id=6.2345;
 insert(&my_tree,(char*)item);
 strcpy(item->name,"Bbbbb");
 item->id=2.2345;
 insert(&my_tree,(char*)item);
 take_out(&my_tree,(char*)item);
 display_tree(my_tree);
 destroy(&my_tree);
 printf("\n");
}
void display_record(char *info)
{
RECORD *temp=(RECORD*)info;
printf("\nИмя -> %s",temp->name);
printf("\n Номер -> %f",temp->id);
}
int less_than(char *item1,char *item2)
{
 RECORD *first=(RECORD*)item1;
 RECORD *second=(RECORD*)item2;
 return strcmp(first->name,second->name)<0;
}
```

```
int equal(char *item1,char *item2)
{
RECORD *first=(RECORD*)item1;
RECORD *second=(RECORD*)item2;
return strcmp(first->name,second->name)==0;
}
```

В тестовой программе определяется структура RECORD, содержащая два поля. Первое поле name служит ключом, на основе которого строится дерево. Второе поле id содержит числовое значение типа float.

Дерево my_tree описывается как глобальная структура.

В качестве параметров функции define передаются три определяемые пользователем функции, display_record, less_than и equal. В качестве первого параметра функции define передается адрес структуры my_tree. Указатель item на структуру RECORD многократно используется для занесения в дерево новых параметров. Затем один из элементов удаляется из дерева. Значения элементов дерева выводятся на печать, после чего дерево уничтожается. Программа выводит следующие результаты:

```
Имя -> Aaaaa
Номер -> 1.234500
Имя -> Ccccc
Номер -> 3.234500
Имя -> Ddddd
Номер -> 4.234500
Имя -> Eeeee
Номер -> 5.234500
Имя -> Fffff
Номер -> 6.234500
```

1. В чем сходство и различие между линейными и нелинейными структурами? **2.** В чем особенность описания типов для создания стека и очереди? **3.** Сколько указателей требуется для работы с очередью? **4.** Изобразите структуру родового дерева. **5.** Из стека L, состоящего из вещественных чисел, постройте две очереди: L1 – из положительных элементов и L2 – из остальных. **6.** Создайте список, значениями которого являются элементы типи-

зированного файла F. **7.** Составьте программу, формирующую родовое дерево, узлами которого являются элементы структуры.

Список литературы

1. *Бабэ Б.* Просто и ясно о Borland C++. – М.: БИНОМ, 1995.
2. *Белецкий Я.* Энциклопедия языка Си. – М., 1992.
3. *Вирт Н.* Алгоритмы и структуры данных. – М.: Мир, 1989.
4. *Гудман С., Хидетниеми С.* Введение в разработку и анализ алгоритмов. – М.: Мир, 1981.
5. Инструментальные средства персональных ЭВМ: В 10 кн. Кн. 7: Программирование в среде Турбо Си: Практическое пособие / *Л.Е. Агабеков, Е.А. Просуков, А.В. Кононенков* и др. – М.: Высш. шк., 1993.
6. *Ирэ П.* Объектно-ориентированное программирование с использованием C++. – К.: ДиаСофт, 1995.
7. *Керниган Б., Ритчи Д.* Язык программирования Си. – М.: Финансы и статистика, 1992.
8. *Кнут Д.* Искусство программирования для ЭВМ: В 3 т. – М.: Мир, 1976.
9. *Страуструп Б.* Язык программирования C++: В 2 ч. – К.: ДиаСофт, 1993.

Оглавление

Глава 5. **Структура программ**

Глава 6. **Структуризация данных**

Глава 7. **Сложные модели данных**

Ершов Евгений Валентинович,
Ганичева Оксана Георгиевна,
Селивановских Вера Витальевна,
Виноградова Людмила Николаевна

ПРОГРАММИРОВАНИЕ. БАЗОВЫЕ СРЕДСТВА ЯЗЫКА ПРОГРАММИРОВАНИЯ С++

Учебное пособие

Редактор Н.С. Менькина
Техническое редактирование: М.Н. Авдюхова
Лицензия А № 165724 от 11.04.06.

Подписано в печать 11.11.11. Формат $60 \times 84\ ^{1}/_{16}$.
Гарнитура таймс. Уч.-изд. л. 6,4. Усл. п. л. 10,6.
Тир. 300. Зак.

ФГБОУ ВПО «Череповецкий государственный университет»
162600 г. Череповец, пр. Луначарского, 5.