Федеральное агентство по образованию

Государственное образовательное учреждение высшего профессионального образования

Ульяновский государственный технический университет

С.М. Наместников

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ С++

Учебное пособие

Ульяновск 2007

УДК 621.391
ББК

Рецензент

Одобрено секцией методических пособий научно-методического совета университета

Основы программирования на языке С++: Учебное пособие/Сост. С. М. Наместников. – Ульяновск: УлГТУ, 2007. – с.

УДК 621.391
ББК

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Целью написания данного учебного пособия является компактное и ясное изложение основных принципов программирования на языке C++. При отборе материала предпочтение отдавалось тем конструкциям, которые наиболее часто используются на практике. Поэтому приведенное здесь изложение языка C++ не претендует на полноту описания, но по мнению автора позволит читателю усвоить достаточный минимум для написания несложных приложений.

Чтение курса «Информатика» придает автору уверенность, что данное пособие будет полезно студентам и аспирантам при изучении языка C++. Материал, приведенный здесь, может быть использован также преподавателями вузов при подготовке и проведении занятий по соответствующим дисциплинам.

Замечания, пожелания и отзывы по книге прошу направлять автору по электронной почте: sernam@ulstu.ru.

ВВЕДЕНИЕ

При разработке большинства языков программирования преследуются сугубо прикладные цели. Например, при создании языка программирования Pascal преследовалась цель преподавание основ программирования. При проектировании BASIC планировалась высокая степень сходства с английским языком, что обеспечивало простоту изучения этого языка людьми, ранее не имевшими дело с компьютерами. При разработке языка C++ преследовалась цель создания инструментального средства, предназначенного для создания программ разной степени сложности.

Язык C++ ориентирован на удовлетворение потребностей программистов. Он предоставляет доступ к аппаратным средствам и позволяет оперировать отдельными битами оперативной памяти. Он включает широкий набор операторов, позволяющих программисту выражать свои идеи в компактном виде. Язык программирования C++ менее строгий, чем, например, язык Pascal в смысле ограничений свободы действий программиста. С одной стороны, эта гибкость является достоинством, но с другой – таит в себе некоторую опасность. Достоинство заключается в том, что многие задачи, например, преобразование типов переменных, в языке C++ решаются достаточно просто. Вместе с тем эта свобода может приводить к ошибкам, которые не возможны в других языках программирования. Таким образом, язык C++ предоставляет большую свободу действий, но и накладывает на программиста большую степень ответственности.

Язык программирования C++ имеет свои недостатки. Самым большим из них считается использование указателей. При этом программист может совершать такие программные ошибки, которые, затем, довольно трудно обнаружить. Также компактность языка C++ в сочетании с большим количеством операторов дает возможность создавать код, понимание которого чрезвычайно затруднительно. Несмотря на указанные недостатки, данный язык программирования является лидером по использованию при написании приложений разной степени сложности.

ВВЕДЕНИЕ В ЯЗЫК С++

Лучший способ начать изучать язык программирования – это посмотреть пример простой программы и понять ее структуру. Поэтому в начале первой главы приводится пример программы на С++ и описание ее структуры. Затем рассматриваются возможные типы данных, правила определения переменных разного типа и программирование арифметических операций включая поразрядные операции над числами. В заключении главы рассматриваются директивы препроцессора и широко распространенные функции ввода/вывода scanf() и printf().

1.1. Структура и этапы создания программы на языке С++

Сама по себе программа на языке С++ представляет собой текстовый файл, в котором представлены конструкции и операторы данного языка в заданном программистом порядке. В самом простом случае этот текстовый файл может содержать такую информацию:

Листинг 1.1. Пример простой программы.

```
/* Пример простой программы*/
#include <stdio.h>
int main()
{
 printf("Hello World!");
 return 0;
}
```

и обычно имеет расширение cpp, например, «ex1.cpp».

Следующий шаг – это компиляция исходного кода. Под компиляцией понимают процесс, при котором содержимое текстового файла преобразуется в исполняемый машинный код, понимаемый процессором компьютера. Однако компилятор создает не готовую к исполнению программу, а только объектный код (файл с расширением *.obj). Этот код является промежуточным этапом при создании готовой программы. Дело в том, что создаваемая программа может содержать функции стандартных библиотек языка С++, реализации которых описаны в объектных файлах библиотек. Например, в приведенной программе используется функция printf() стандартной библиотеки «stdio.h». Это означает, что объектный файл ex1.obj будет содержать лишь инструкции по вызову данной функции, но код самой функции в нем будет отсутствовать.

Для того чтобы итоговая исполняемая программа содержала все необходимые реализации функций, используется компоновщик объектных кодов. Компоновщик – это программа, которая объединяет в единый

исполняемый файл объектные коды создаваемой программы, объектные коды реализаций библиотечных функций и стандартный код запуска для заданной операционной системы. В итоге и объектный файл, и исполняемый файл состоят из инструкций машинного кода. Однако объектный файл содержит только результат перевода на машинный язык текста программы, созданной программистом, а исполняемый файл – также и машинный код для используемых стандартных библиотечных подпрограмм и для кода запуска.

Графически описанные этапы создания программы представлены на рис. 1.1. Здесь исполняемый файл имеет расширение *.exe и может быть запущен обычным образом из соответствующей операционной системы: MS-DOS или Windows.

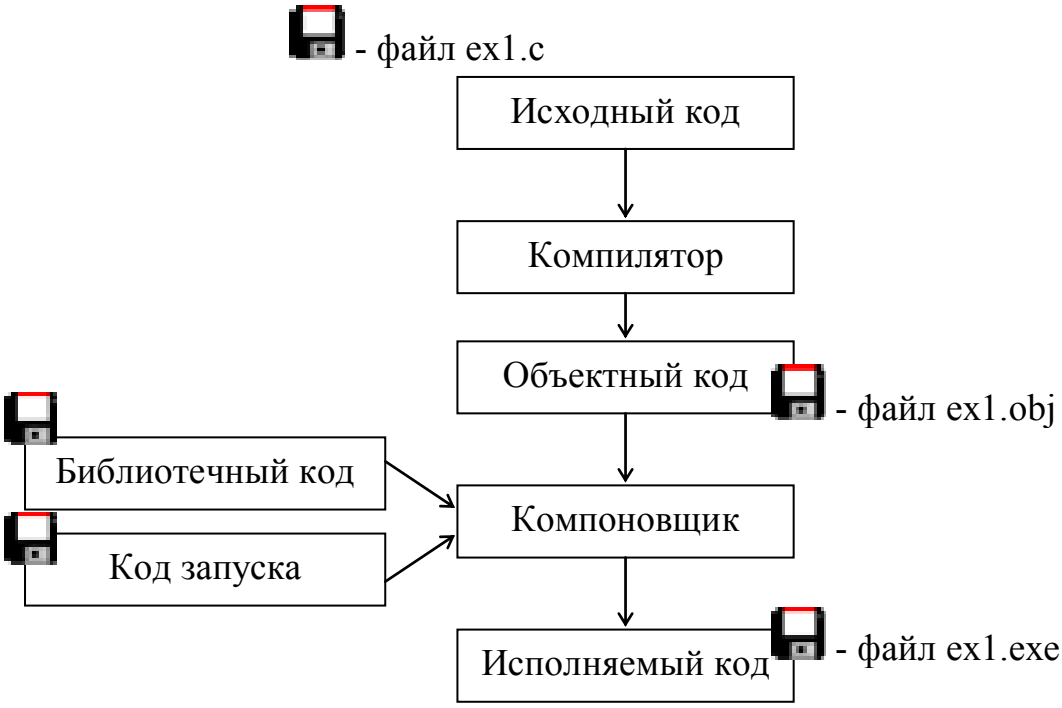

Рис. 1.1. Этапы создания программы

Рассмотрим более подробно пример программы листинга 1.1. Первая строка задает комментарии, т.е. замечания, помогающие лучше понять программу. Они предназначены только для чтения и игнорируются компилятором. Во второй строке записана директива #include, которая дает команду препроцессору языка C++ вставить содержимое файла ‘stdio.h’ на место этой строки при компиляции. В третьей строке определена функция с именем main, которая возвращает целое число (тип int) и не принимает никаких аргументов (тип void). Функция main() является обязательной функцией для всех программ на языке C++ и без ее наличия уже на этапе компиляции появляется сообщение об ошибке, указывающее на отсутствие данной функции. Обязательность данной функции обусловливается тем, что она является точкой входа в программу. В данном случае под точкой входа понимается функция, с которой начинается и которой заканчивается работа

программы. Например, при запуске exe-файла происходит активизация функции main(), выполнение всех операторов, входящих в нее и завершение программы. Таким образом, логика всей программы заключена в этой функции. В приведенном примере при вызове функции main() происходит вызов функции printf(), которая выводит на экран монитора сообщение “Hello World!”, а затем выполняется оператор return, который возвращает нулевое значение. Это число возвращается самой функцией main() операционной системе и означает успешное завершение программы. Фигурные скобки {} служат для определения начала и конца тела функции, т.е. в них содержатся все возможные операторы, которые описывают работу данной функции. Следует отметить, что после каждого оператора в языке C++ ставится символ ‘;’. Таким образом, приведенный пример показывает общую структуру программ на языке C++.

1.2. Стандарты языка C++

В настоящее время имеется множество реализаций языка C++. В идеальном случае, написанная программа на одной реализации языка должна одинаковым образом выполняться и на любой другой реализации этого же языка. Для обеспечения этого условия существуют стандарты, описывающие основные конструкции C++ и правила их построения.

По мере того, как язык C постепенно развивался сообщество пользователей этого языка осознало, что нуждается в современном и строгом стандарте. В ответ на эти потребности Американский институт национальных стандартов (American National Standards Institute (ANSI)) в 1983 г. организовал комитет (X3J11) для разработки нового стандарта, который был принят в 1989 г. Этот стандарт (ANSI C) содержит определение как языка, так и стандартной библиотеки C. Затем международная организация по стандартизации (ISO) в 1990 г. приняла свой стандарт (ISO C), который по сути не отличается от стандарта ANSI C.

В 1994 г. возобновилась деятельность по разработке нового стандарта, в результате чего появился стандарт C99, который соответствует языку C++. Объединенный комитет ANSI/ISO развил исходные принципы предыдущего стандарта, являющийся основным на сегодняшний день.

1.3. Представление данных в языке C++

По существу программа есть не что иное, как обмен и преобразование разными типами данных. В связи с этим изучать программирование целесообразно со знакомства существующих типов данных. В табл. 1.1 представлены основные базовые типы данных языка C++.

Для того чтобы иметь возможность работать с тем или иным типом данных необходимо задать переменную соответствующего типа. Это осуществляется с использованием следующего синтаксиса:

```
<тип переменной> <имя_переменной>;
```

например, строка

```
int arg;
```

объявлет целочисленную переменную с именем arg.

Таблица 1.1. Основные базовые типы данных

Тип	Описание
int	Целочисленный тип 16 либо 32 бит
long int	Целочисленный тип 32 бит
short	Целочисленный тип 8 либо 16 бит
char	Символьный тип 8 бит
float	Вещественный тип 32 бит
double	Вещественный тип 64 бит

Отметим, что при выборе имени переменной целесообразно использовать осмысленные имена. При определении имени можно использовать как верхний, так и нижний регистры букв латинского алфавита. Причем первым символом обязательно должна быть буква или символ подчеркивания ‘_’. Вот несколько примеров:

Правильные имена	Неправильные имена
arg	&arg
cnt	$cnt
bottom_x	bottom-x
Arg	2Arg
don_t	don’t

В приведенных примерах переменные arg и Arg считаются разными, т.к. язык C++ при объявлении переменных различает большой и малый регистры.

В отличии от многих языков программирования высокого уровня, в языке C++ переменные могут объявляться в любом месте текста программы.

1.4. Оператор присваивания

Для того чтобы задать то или иное значение переменной используется оператор присваивания, который записывается как знак ‘=’. Работу этого оператора представим на следующем примере:

```
int length = 5;
```

Здесь переменной с именем length присваивается значение 5, т.е. элемент слева от знака равенства является именем переменной, а элемент справа – это значение, присвоенное этой переменной. Сам символ ‘=’ называется оператором присваивания.

На первый взгляд, различие между именем переменной и значением переменной может показаться несущественным, однако рассмотрим такой пример обычного вычислительного оператора:

```
int i=2;
i=i+1;
```

С математической точки зрения такая операция не имеет смысла, т.к. переменная i не может быть равна i+1. Однако данная запись в смысле операции присваивания вполне приемлема. Действительно, компьютер сначала определит значение переменной i, а затем прибавит 1 к этому значению и полученный результат присвоит снова переменной i. Таким образом, исходное значение переменной i увеличится на 1.

Так как оператор присваивания задает значение переменной, то оператор типа

```
20 = i;
```

не будет иметь смысла, поскольку число 20 будет интерпретироваться как константа, которой не может быть присвоено какое-либо другое значение.

Теперь рассмотрим такой пример. Допустим, что имеются две переменные разного типа:

```
short agr_short = -10;
long arg_long;
```

и выполняется оператор присваивания

```
arg_long = arg_short;
```

В результате переменная arg_long будет иметь значение 10 и оператор присваивания выполнит автоматическое преобразование типов и потери данных не происходит. В другом случае, когда

```
float agr_f = 8.7;
int arg_long;
arg_long = arg_f;
```

в операторе присваивания произойдет потеря данных, т.к. целое число не может представлять числа, стоящие после запятой (точки). В результате переменная arg_long будет иметь значение 8. Для корректного преобразования

одного типа данных в другой используется операция приведения типов, имеющая следующий синтаксис:

```
<имя_переменной_1> = (тип_данных)<имя_переменной_2>;
```

Например,

```
arg_long = (long )arg_f;
```

1.5. Системы счисления

Любое число можно представлять в разной системе счисления. Во всех предыдущих примерах использовалось десятичная форма записи числа. Это значит, что число может быть разложено по степеням 10, например,

$2145 = 2 \cdot 10^3 + 1 \cdot 10^2 + 4 \cdot 10^1 + 5 \cdot 10^0$.

Однако компьютер оперирует двоичными числами, т.е. представляет число по степеням двойки, например:

$64 = 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$,

которое часто записывают в виде

Номер бита:

6	5	4	3	2	1	0
1	0	0	0	0	0	0

Учитывая, что в одном байте 8 бит, максимальное число, которое он может содержать равно

$1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 255$.

Таким образом, один байт информации может представлять десятичные числа в диапазоне от 0 до 255, т.е. 256 возможных значений.

Кроме десятичной и двоичной систем счисления при программировании часто используют шестнадцатиричную систему, т.е. числа с основанием 16. Для записи таких чисел недостаточно использовать цифры, поэтому для обозначения основания больше 9 добавляют буквы A – 10, B – 11, C – 12, D – 13, E – 14, F – 15 (или малого регистра a,b,c,d,e,f). Например, десятичное число 0 соответствует шестнадцатиричному 0, а десятичное 15, шестнадцатиричному F. Для представления числа 16 используется шестнадцатиричное $1 \cdot 16^1 + 0 \cdot 16^0 = 10$, а число 255 соответствует числу $15 \cdot 16^1 + 15 \cdot 16^0 = \mathrm{F} \cdot 16^1 + \mathrm{F} \cdot 16^0 = \mathrm{FF}$.

Следует отметить, что каждая цифра шестнадцатиричного числа представляется четырьмя битами двоичного числа. Действительно $2^3 + 2^2 + 2^1 + 2^0 = 15$ дает диапазон чисел от 0 до 15 или в шестандцатиричной записи от 0 до F. Это свойство удобно для представления байтовых чисел, где каждая половинка байта представляется одним шестнадцатиричным числом. Например,

FF = 11111111; 0F = 00001111; 11 = 00010001 и т.д.

Для представления шестнадцатиричных чисел в языке C++ используется следующий синтаксис:

```
int var = 0xff; //число 255
int arg = 0xac; //число 172
```

1.6. Арифметические операции

В языке C++ довольно просто реализуются элементарные математические операции: сложения, вычитания, умножения и деления. Допустим, что в программе заданы две переменные

```
int a, b;
```

с начальными значениями

```
a=4;
b=8;
```

тогда операции сложения, вычитания, умножения и деления будут выглядеть следующим образом:

```
int c;
c = a+b;        //сложение двух переменных
c = a-b;        //вычитание
c = a*b;        //умножение
c = a/b;        //деление
```

Представленные операции можно выполнять не только с переменными, но и с конкретными числами, например

```
c = 10+5;
c = 8*4;
float d;
d = 7/2;
```

Результатом первых двух арифметических операций будут числа 15 и 32 соответственно, но при выполнении операции деления в переменную d будет записано число 3, а не 3,5. Это связано с тем, что число 7 в языке C++ будет интерпретироваться как целочисленная величина, которая не может содержать дробной части. Поэтому полученная дробная часть 0,5 будет отброшена. Для реализации корректного деления одного числа на другое следует использовать такую запись:

```
d = 7.0/2;
```

или

```
d = (float )7/2;
```

В первом случае вещественное число делится на два и результат (вещественный) присваивается вещественной переменной d. Во втором варианте выполняется приведение типов: целое число 7 приводится к вещественному типу float, а затем делится на 2. Второй вариант удобен, когда выполняется деление одной целочисленной переменной на другую:

```
int a,b;
a = 7; b = 2;
d = a/b;
```

В результате значение d будет равно 3, но если записать

```
d = (float )a/b;
```

то получим значение 3,5. Здесь следует также отметить, что если переменная d является целочисленной, то результат деления всегда будет записан с отброшенной дробной частью. Аналогичные правила справедливы для всех арифметических операций.

В заключении рассмотрения работы с арифметическими операциями отметим, что приоритет операций умножения и деления выше приоритета операций сложения и вычитания. Это означает, что сначала выполняются операции умножения и деления и только затем операции сложения и вычитания. Следующий пример демонстрирует приоритет арифметических операций:

```
double n=2, SCALE = 1.2;
double arg = 25.0 + 60.0*n/SCALE;
```

В приведенном примере сначала будет выполнена операция умножения, затем деления и, наконец, сложения. То есть порядок вычисления соответствует математическим правилам. Для того чтобы изменить порядок вычисления (поменять приоритеты) используются круглые скобки как показано ниже

```
double arg = (25.0 + 60.0)*n/SCALE;
```

Здесь сначала выполняется операция сложения и только затем операции умножения и деления.

Кроме рассмотренных арифметических операций в C++ имеется полезная операция деления по модулю. Ее результатом является остаток от деления одного целого числа на другое. Так выражение

```
int a = 13 % 5;
```

означает, что число 13 делится по модулю 5. Учитывая, что число 5 дважды входит в число 13, то остаток получается равный 3. Эту операцию можно реализовать и на основе стандартных арифметических операций следующим образом:

```
int a = 13 – 13/5*5;
```

Следует отметить, что операция целочисленного деления % может быть реализована только для целых чисел и целочисленных переменных и не применима к другим типам данных.

Для простоты программирования в языке С++ реализованы компактные операторы инкремента и декремента, т.е. увеличения и уменьшения значения переменной на 1 соответственно. Данные операторы могут быть записаны в виде

```
i++;        // операция инкремента
++i;        // операция инкремента
i--;        // операция декремента
--i;        // операция декремента
```

Разницу между первой и второй формами записи данных операторов можно продемонстрировать на следующем примере:

```
int i=10,j=10;
int a = i++;          //значение a = 10; i = 11;
int b = ++j;          //значение b = 11; j = 11;
```

Из полученных результатов видно, что если оператор инкремента стоит после имени переменной, то сначала выполняется операция присваивания и только затем операция инкремента. Во втором случае наоборот, операция инкремента реализуется до присвоения результата другой переменной. Поэтому значение a = 10, а значение b = 11. В первом случае говорят о постпрефиксной форме, а во втором – о префиксной. Подобный приоритет операции инкремента остается справедливым и при использовании арифметических операций, например

```
int a1=4, a2=4;
double b = 2.4*++a1;       //результат b = 12.0
double c = 2.4*a2++;       //результат c = 9.6
```

Из приведенного примера видно, что операция инкремента (декремента) обладает более высоким приоритетом, чем операция умножения (соответственно и деления). Для того чтобы изменить приоритеты используются круглые скобки.

Операция декремента действует аналогично операции инкремента с той лишь разницей, что она уменьшает значение переменной на 1.

1.7. Поразрядные операции языка C++

Поразрядные операции состоят из четырех основных операций: отрицание, логическое И, логическое ИЛИ и исключающее ИЛИ. Рассмотрим данные операции по порядку.

При выполнении операции поразрядного отрицания все биты, равные 1, устанавливаются равными 0, а все биты равные нулю, устанавливаются равными 1. Для выполнения данной операции в языке C++ используется символ ‘~’ как показано в следующем примере:

```
unsigned char var = 153;  //двоичная запись 10011001
unsigned char not = ~var; //результат 01100110 (число 102)
```

В результате переменная not будет содержать число 102. В ходе выполнения операции поразрядного И результирующий бит будет равен 1, если оба бита в соответствующих операндах равны 1, т.е.

10010011 & 00111101 даст результат

00010001.

Для выполнения операции логического И используется символ & следующим образом:

```
unsigned char var = 153; //двоичная запись 10011001
unsigned char mask = 0x11;        // число 00010001 (число 17)
unsigned char res = var & mask;   // результат 00010001
```

или

```
var &= mask;    // то же самое, что и  var = var & mask;
```

В ходе выполнения двоичной операции ИЛИ результирующий бит устанавливается равным 1, если хотя бы один бит соответствующих операндов равен 1. В противном случае, результирующее значение равно 0. Для выполнения данной логической операции используется символ ‘|’ как показано ниже:

```
unsigned char var = 153;   //двоичная запись 10011001
unsigned char mask = 0x11; // число          00010001
unsigned char res = var | mask; // результат 10011001
```

Также допускается применение такой записи

```
var |= mask; // то же самое, что и var = var | mask;
```

Наконец, при операции исключающее ИЛИ результирующий бит устанавливается равным 0, если оба бита соответствующих операндов равны 1, и 1 в противном случае. Для выполнения данной операции в языке C++ используется символ ‘^’:

```
unsigned char var = 153; //двоичная запись   10011001
unsigned char mask = 0x11; // число          00010001
unsigned char res = var ^ mask; // результат 10001000
```

или

```
var ^= mask; // то же самое, что и var = var ^ mask;
```

Рассмотрим примеры использования логических операций, которые часто применяются на практике. Самой распространенной по использованию является операция логического И. Данная операция обычно используется совместно с так называемыми масками. Под маской понимают битовый шаблон, который служит для выделения тех или иных битов числа, к которому она применяется. Например, если требуется определить, является ли нулевой бит числа установленным в 1 или нет, то для этого задается маска 00000001, которая соответствует числу 1 и выполняется операция поразрядного И:

```
unsigned char flags = 3; // 00000011
unsigned char mask = 1;  // 00000001
if((flag & mask) == 1) printf(“Нулевой бит включен”);
else printf(“Нулевой бит выключен”);
```

Здесь переменная flags, представленная одним байтом, содержит восемь флаговых битов. Для того чтобы узнать установлен или нет нулевой флаговый бит задается маска со значением 1 и выполняется операция логического И. В результате все биты переменной flags будут равны нулю за исключением нулевого, если он изначально имел значение 1. Таким образом, маска является шаблоном, который как бы накладывается на битовое представление числа, из которого выделяются биты, соответствующие единичным значениям маски. Рассмотренный пример показывает, как одна байтовая переменная flags может содержать восемь флаговых значений и тем самым экономить память ЭВМ.

Следующим примером использования логических операция является возможность включать нужные биты в переменной, оставляя другие без изменений. Для этого используется логическая операция ИЛИ. Допустим, в переменной flags необходимо установить второй бит равным 1. Для этого задается маска в виде переменной mask = 2 (00000010) и реализуется операция логического ИЛИ:

```
unsigned char flags = 0; // 00000000
unsigned char mask = 2;  // 00000010
flags |= mask;
```

Этот код гарантирует, что второй бит переменной flags будет равен 1 без изменений значений других битов.

Для отключения определенных битов целесообразно использовать две логические операции: логическое И и логическое НЕ. Допустим, требуется отключить второй бит переменной flags. Тогда предыдущий пример запишется следующим образом:

```
unsigned char flags = 0; // 00000000
unsigned char mask = 2;  // 00000010
flags = flags & ~mask;
```

или

```
flags &= ~mask;
```

Работа этих двух операций заключается в следующем. Приоритет операции НЕ выше приоритета операции И, поэтому переменная mask в двоичной записи будет иметь вид 11111101. Применяя операцию логического умножения переменной flags с полученным числом ~mask все биты останутся неизменными, кроме второго, значение которого будет равно нулю.

Наконец, операция исключающее ИЛИ позволяет переключать заданные биты переменных. Идея переключения битов основывается на свойствах операции исключающего ИЛИ: 1^1 = 0, 1^0 = 1, 0^0 = 0 и 0^1 = 1. Анализ данных свойств показывает, что если значение бита маски будет равно 1, то соответствующий бит переменной, к которой она применяется, будет переключен, а если значение бита маски равно 0, то значение бита переменной останется неизменным. Следующий пример демонстрирует работу переключения битов переменной flags.

```
unsigned char flags = 0; //00000000
unsigned char mask = 2;  //00000010
flags ^= mask;           //00000010
flags ^= mask;           //00000000
```

Кроме логических операций в языке C++ существуют операции поразрядного смещения битов переменной. Операция смещения битов влево определяется знаком << и смещает биты значения левого операнда на шаг, определенный правым операндом, например, в результате выполнения команды

```
10001010 << 2;
```

получится результат 00101000. Здесь каждый бит перемещается влево на две позиции, а появляющиеся новые биты устанавливаются нулевыми. Рассмотрим особенности действия данной операции на следующем примере:

```
int var = 1;
var = var <<1; //00000010 – значение 2
var <<= 1;     //00000100 – значение 4
```

Можно заметить, что смещение битов переменной на одну позицию влево приводит к операции умножения числа на 2. В общем случае, если выполнить сдвиг битов на n шагов, то получим результат равный умножению переменной на 2^n. Данная операция умножения на число 2^n является более быстрой, чем обычное умножения, рассматриваемое ранее.

Аналогично, при операции смещения вправо >> происходит сдвиг битов переменной на шаг, указанный в правом операнде. Например, сдвиг

```
00101011 >> 2;
```

приведет к результату 00001010. Здесь, также как и при сдвиге влево, новые появляющиеся биты устанавливаются равными нулю. В результате выполнения последовательностей операций

```
int var = 128;  //1000000
var = var >> 1; //0100000 – значение 64
var >>= 1;      //0010000 – значение 32
```

значение переменной var каждый раз делится на 2. Поэтому сдвиг var >>= n можно использовать для выполнения операции деления значения переменной на величину 2^n.

1.8. Директивы препроцессора

Почти все программы на языке C++ используют специальные команды для компилятора, которые называются директивами. В общем случае директива – это указание компилятору языка C++ выполнить то или иное действие в момент компиляции программы. Существует строго определенный набор возможных директив, который включает в себя следующие определения:

#define, #elif, #else, #endif, #if, #ifdef, #ifndef, #include, #undef.

Директива #define используется для задания констант, ключевых слов, операторов и выражений, используемых в программе. Общий синтаксис данной директивы имеет следующий вид:

```
#define <идентификатор> <текст>
```

или

```
#define <идентификатор> (<список параметров>) <текст>
```

Следует заметить, что символ ‘;’ после директив не ставится. Приведем примеры вариантов использования директивы #define.

Листинг 1.2. Примеры использования директивы #define.

```
#include <stdio.h>
#define TWO 2
#define FOUR TWO*TWO
#define PX printf("X равно %d.\n", x)
#define FMT «X равно %d.\n»
#define SQUARE(X) X*X
int main()
{
 int x = TWO;
 PX;
 x = FOUR;
 printf(FMT, x);
 x = SQUARE(3);
 PX;

 return 0;
}
```

После выполнения этой программы на экране монитора появится три строки:

```
X равно 2.
X равно 4.
X равно 9.
```

Директива #undef отменяет определение, введенное ранее директивой #define. Предположим, что на каком-либо участке программы нужно отменить определение константы FOUR. Это достигается следующей командой:

```
#undef FOUR
```

Интересной особенностью данной директивы является возможность переопределения значения ранее введенной константы. Действительно, повторное использование директивы #define для ранее введенной константы FOUR невозможно, т.к. это приведет к сообщению об ошибке в момент компиляции программы. Но если отменить определение константы FOUR с помощью директивы #undef, то появляется возможность повторного использования директивы #define для константы FOUR.

Для того чтобы иметь возможность выполнять условную компиляцию, используется группа директив #if, #ifdef, #ifndef, #elif, #else и #endif. Приведенная ниже программа выполняет подключение библиотек в зависимости от установленных констант.

```
#if defined(GRAPH)
 #include <graphics.h>  //подключение графической библиотеки
#elif defined(TEXT)
 #include <conio.h>     //подключение текстовой библиотеки
#else
 #include <io.h>        //подключение библиотеки ввода-вывода
#endif
```

Данная программа работает следующим образом. Если ранее была задана константа с именем GRAPH через директиву #define, то будет подключена графическая библиотека с помощью директивы #include. Если идентификатор GRAPH не определен, но имеется определение TEXT, то будет использоваться библиотека текстового ввода/вывода. Иначе, при отсутствии каких-либо определений, подключается библиотека ввода/вывода. Вместо словосочетания #if defined часто используют сокращенные обозначения #ifdef и #ifndef и выше приведенную программу можно переписать в виде:

```
#ifdef GRAPH
 #include <graphics.h>    //подключение графической библиотеки
#ifdef TEXT
 #include <conio.h>  //подключение текстовой библиотеки
#else
 #include <io.h>     //подключение библиотеки ввода-вывода
#endif
```

Отличие директивы #if от директив #ifdef и #ifndef заключается в возможности проверки более разнообразных условий, а не только существует или нет какие-либо константы. Например, с помощью директивы #if можно проводить такую проверку:

```
#if SIZE == 1
 #include <math.h>  // подключение математической библиотеки
#elif SIZE > 1
 #include <array.h>  // подключение библиотеки обработки
                     // массивов
#endif
```

В приведенном примере подключается либо математическая библиотека, либо библиотека обработки массивов, в зависимости от значения константы SIZE.

Данные директивы иногда используются для выделения нужных блоков программы, которые требуется использовать в той или иной программной

реализации. Следующий пример демонстрирует работу такого программного кода.

Листинг 1.3. Пример компиляции отдельных блоков программы.

```
#include <stdio.h>
#define SQUARE
int main()
{
 int s = 0;
 int length = 10;
 int width = 5;

 #ifdef SQUARE
      s=length*width;
 #else
      s=2*(length+width);
#endif

 return 0;
}
```

В данном примере происходит вычисление либо площади прямоугольника, либо его периметра, в зависимости от того определено или нет значение SQUARE. По умолчанию программа вычисляет площадь прямоугольника, но если убрать строку #define SQUARE, то программа станет вычислять его периметр.

Используемая в приведенных примерах директива #include позволяет добавлять в программу ранее написанные программы и сохраненные в виде файлов. Например, строка

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

указывает препроцессору добавить содержимое файла stdio.h вместо приведенной строки. Это дает большую гибкость, легкость программирования и наглядность создаваемого текста программы. Есть две разновидности директивы #include:

```
#include <stdio.h> - имя файла в угловых скобках
```

и

```
#include «mylib.h» - имя файла в кавычках
```

Угловые скобки сообщают препроцессору о том, что необходимо искать файл (в данном случае stdio.h) в одном или нескольких стандартных системных каталогах. Кавычки свидетельствуют о том, что препроцессору

необходимо сначала выполнить поиск файла в текущем каталоге, т.е. в том, где находится файл создаваемой программы, а уже затем – искать в стандартных каталогах.

1.9. Функции ввода/вывода printf() и scanf()

Функция printf() позволяет выводить информацию на экран при программировании в консольном режиме. Данная функция определена в библиотеке stdio.h и имеет следующий синтаксис:

```
int printf( const char *format [, argument]... );
```

Здесь первый аргумент *format определяет строку, которая выводится на экран и может содержать специальные управляющие символы для вывода переменных. Затем, следует список необязательных аргументов, которые поясняются ниже. Функуция возвращает либо число отображенных символов, либо отрицательное число в случае своей некорректной работы.

В самой простой реализации функция printf() просто выводит заданную строку на экран монитора:

```
printf("Привет мир.");
```

Однако с ее помощью можно выводить переменные разного типа: начиная с числовых и заканчивая строковыми. Для выполнения этой операции используются специальные управляющие символы, которые называются спецификаторами и которые начинаются с символа %. Следующий пример демонстрирует вывод целочисленной переменной num на экран монитора с помощью функции printf():

```
int num;
num = 5;
printf("%d", num);
```

В первых двух строках данной программы задается переменная с именем num типа int. В третьей строке выполняется вывод переменной на экран. Работа функции printf() выглядит следующим образом. Сначала функция анализирует строку, которую необходимо вывести на экран. В данном случае это «%d». Если в этой строке встречается спецификатор, то на его место записывается значение переменной, которая является вторым аргументом функции printf(). В результате, вместо исходной строки «%d» на экране появится строка «5», т.е. будет выведено число 5.

Следует отметить, что спецификатор «%d» выводит только целочисленные типы переменных, например int. Для вывода других типов следует использовать другие спецификаторы. Ниже перечислены основные виды спецификаторов:

%c – одиночный символ
%d – десятичное целое число со знаком
%f – число с плавающей точкой (десятичное представление)
%s – строка символов (для строковых переменных)
%u – десятичное целое без знака
%% - печать знака процента

С помощью функции printf() можно выводить сразу несколько переменных. Для этого используется следующая конструкция:

```
int num_i;
float num_f;
num_i = 5;
num_f = 10.5;
printf("num_i = %d, num_f = %f", num_i, num_f);
```

Результат выполнения программы будет выглядеть так:

```
num_i = 5, num_f = 10.5
```

Кроме спецификаторов в функции printf() используются управляющие символы, такие как перевод строки \n, табуляции \t и др. Например, если в ранее рассмотренном примере необходимо вывести значения переменных не в строчку, а в столбик, то необходимо переписать функцию printf() следующим образом:

```
printf("num_i = %d,\n num_f = %f", num_i, num_f);
```

Аналогично используется и символ табуляции.

Для ввода информации с клавиатуры удобно использовать функцию scanf() библиотеки stdio.h, которая имеет следующий синтаксис:

```
int scanf( const char *format [,argument]... );
```

Здесь, как и для функции printf(), переменная *format определяет форматную строку для определения типа вводимых данных и может содержать те же спецификаторы что и функция printf(). Затем, следует список необязательных аргументов. Работа функции scanf() демонстрируется на листинге 1.4.

Листинг 1.4. Пример использования функции scanf().

```
#include <stdio.h>
int main()
{
 int age;
```

```
 float weight;
 printf("Введите информацию о Вашем возрасте: ");
 scanf("%d", &age);
 printf("Введите информацию о Вашем весе: ");
 scanf("%f", &weigth);
 printf("Ваш возраст = %d, Ваш вес = %f", age, weigth);

 return 0;
}
```

Основным отличием применения функции scanf() от функции printf() является знак & перед именем переменной, в которую записываются результаты ввода.

Функция scanf() может работать сразу с несколькими переменными. Предположим, что необходимо ввести два целых числа с клавиатуры. Формально для этого можно дважды вызвать функцию scanf(), однако лучше воспользоваться такой конструкцией:

```
scanf(" %d,  %d ", &n, &m);
```

Функция scanf() интерпретирует это так, как будто ожидает, что пользователь введет число, затем – запятую, а затем – второе число. Все происходит так, как будто требуется ввести два целых числа следующим образом:

88,221

или 88, 221

Функция scanf() возвращает число успешно считанных элементов. Если операции считывания не происходило, что бывает в том случае, когда вместо ожидаемого цифрового значения вводится какая-либо буква, то возвращаемое значение равно 0.

Контрольные вопросы и задания

1. Каким образом можно задавать комментарии в программе написанной на языке C++?
2. Запишите объявление целочисленной переменной с именем var_i.
3. С каких символов должны начинаться имена переменных?
4. Как изменится значение переменной i после выполнения операции i=i+1;?
5. Какой результат получится после выполнения операции var=7/2;?
6. Запишите вещественные типы переменных.
7. Может ли переменная символьного типа char принимать целые числовые значения?
8. Приведите пример использования функции printf() для вывода значений двух целочисленных переменных на экран.
9. Запишите функцию scanf() для ввода символа с клавиатуры.
10. Как в языке C++ записывается операция умножения?

11.Какой результат получится после операции деления двух целочисленных переменных var1=7 и var2 = 2?

12.Каким образом следует записать приведенный ниже фрагмент программы для получения корректного результата деления?

```
int a = 9, b = 2;
float c = a/b;
```

13.Запишите директиву #define для задания константы с именем LENGTH равной 10.

14.Приведите пример макроса, позволяющий возводить число в квадрат.

15.С помощью каких директив можно выполнять условную компиляцию программы?

16.Каким символом обозначается операция логическое И и что она делает?

17.Как записывается операция логическое ИЛИ и для чего она предназначена?

18.Запишите операцию логическое НЕ применительно к переменной var_i.

19.Приведите пример использования операции исключающего ИЛИ и объясните полученный результат.

20.С помощью какой поразрядной операции можно выполнять деление числа на 2?

21.Запишите операцию умножения числа на 4 с помощью поразрядной операции.

ГЛАВА ВТОРАЯ

БАЗОВЫЕ КОНСТРУКЦИИ ЯЗЫКА С++

Каждый язык программирования высокого уровня имеет набор базовых конструкций (операторов), которые составляют основу при написании программ. К ним относятся условные операторы, операторы циклов и пользовательские функции. Рассмотрению данных разделов и посвящена вторая глава.

21. Условные операторы if и switch

Для того чтобы иметь возможность реализовать логику в программе используются условные операторы. Умозрительно эти операторы можно представить в виде узловых пунктов, достигая которых программа делает выбор по какому из возможных направлений двигаться дальше. Например,

требуется определить, содержит ли некоторая переменная arg положительное или отрицательное число и вывести соответствующее сообщение на экран. Для этого можно воспользоваться оператором if (если), который и выполняет подобные проверки.

В самом простом случае синтаксис данного оператора if следующий:

```
if (выражение)
 <оператор>
```

Если значение параметра «выражение» равно «истинно», выполняется оператор, иначе он пропускается программой. Следует отметить, что «выражение» является условным выражением, в котором выполняется проверка некоторого условия. В табл. 2.1 представлены варианты простых логических выражений оператора if.

Таблица 2.1. Простые логические выражения

if(a < b)	Истинно, если переменная a меньше переменной b и ложно в противном случае.
if(a > b)	Истинно, если переменная a больше переменной b и ложно в противном случае.
if(a == b)	Истинно, если переменная a равна переменной b и ложно в противном случае.
if(a <= b)	Истинно, если переменная a меньше либо равна переменной b и ложно в противном случае.
if(a >= b)	Истинно, если переменная a больше либо равна переменной b и ложно в противном случае.
if(a != b)	Истинно, если переменная a не равна переменной b и ложно в противном случае.
if(a)	Истинно, если переменная a не равна нулю, и ложно в противном случае.

Приведем пример использования оператора ветвления if. Следующая программа позволяет определять знак введенной переменной.

Листинг 2.1. Первая программа определения знака введенного числа.

```
#include <stdio.h>
int main()
{
```

```
 float x;
 printf("Введите число: ");
 scanf("%f",&x);
 if(x < 0)
  printf("Введенное число %f является отрицательным.\n", x);
 if(x >= 0)
  printf("Введенное число %f является неотрицательным.\n", x);

 return 0;
}
```

Анализ приведенного текста программы показывает, что два условных оператора можно заменить одним, используя конструкцию

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

которая интерпретируется таким образом. Если «выражение» истинно, то выполняется «оператор1», иначе выполняется «оператор2». Перепишем ранее приведенный пример определение знака числа с использованием данной конструкции.

Листинг 2.2. Вторая программа определения знака введенного числа.

```
#include <stdio.h>
int main()
{
 float x;
 printf("Введите число: ");
 scanf("%f",&x);
 if(x < 0)
  printf("Введенное число %f является отрицательным.\n", x);
 else
  printf("Введенное число %f является неотрицательным.\n", x);

 return 0;
}
```

В представленных примерах после операторов if и else стоит всего одна функция printf(). В случаях, когда при выполнении какого-либо условия необходимо записать более одного оператора, необходимо использовать фигурные скобки, т.е. использовать конструкцию вида

```
if (выражение)
{
 <список операторов>
}
```

```
else
{
 <список операторов>
}
```

Следует отметить, что после ключевого слова else формально можно поставить еще один оператор условия if, в результате получим еще более гибкую конструкцию условных переходов:

```
if(выражение1) <оператор1>
else if(выражение2) <опреатор2>
else <оператор3>
```

На листинге 2.3 представлена программа, реализующая последнюю конструкцию условных переходов.

Листинг 2.3. Третья программа определения знака введенного числа.

```
#include <stdio.h>
int main()
{
 float x;
 printf("Введите число: ");
 scanf("%f",&x);
 if(x < 0)
   printf("Введенное число %f является отрицательным.\n", x);
 else if(x > 0)
   printf("Введенное число %f является положительным.\n", x);
 else
   printf("Введенное число %f является неотрицательным.\n",x);

 return 0;
}
```

До сих пор рассматривались простые условия типа x<0. Вместе с тем оператор if позволяет реализовывать более сложные условные переходы. В языке C++ имеются три логические операции:

&& - логическое И
|| - логическое ИЛИ
! – логическое НЕТ

На основе этих трех логических операций можно сформировать более сложные условия. Например, если имеются три переменные exp1, exp2 и exp3, то они могут составлять логические конструкции, представленные в табл. 2.2.

Таблица 2.2. Пример составных логических выражений

if(exp1 > exp2 && exp2 < exp3)	Истинно, если значение переменной exp1 больше значения переменной exp2 и значение переменной exp2 меньше значения переменной exp3.
if(exp1 <= exp2 || exp1 >= exp3)	Истинно, если значение переменной exp1 меньше либо равно значения переменной exp2 или значение переменной exp2 больше либо равно значения переменной exp3.
if(exp1 && exp2 && !exp3)	Истинно, если истинное значение exp1 и истинно значение exp2 и ложно значение exp3.
if(!exp1 || !exp2 && exp3)	Истинно, если ложно значение exp1 или ложно значение exp2 и истинно значение exp3.

Подобно операциям умножения и сложения в математике, логические операции И ИЛИ НЕТ, также имеют свои приоритеты. Самый высокий приоритет имеет операция НЕТ, т.е. такая операция выполняется в первую очередь. Более низкий приоритет у операции И, и наконец самый малый приоритет у операции ИЛИ. Данные приоритеты необходимо учитывать, при составлении сложных условий. Например, условие

```
if(4 < 6 && 5 > 6 || 5 < 6)
```

проверяется таким образом. Если 4 < 6 И 5 > 6 ИЛИ 5 < 6, то выполняется переход по условию. При необходимости изменения порядка проверки следует операции с более низким приоритетом поместить в круглые скобки, как показано ниже

```
if(4 < 6 && (5 > 6 || 5 < 6))
```

Условная операция if облегчает написание программ, в которых необходимо производить выбор между небольшим числом возможных вариантов. Однако иногда в программе необходимо осуществить выбор одного варианта из множества возможных. Формально для этого можно воспользоваться конструкцией if else if … else. Однако во многих случаях оказывается более удобным применять оператор switch языка C++. Синтаксис данного оператора следующий:

```
switch(переменная)
{
 case константа1:
      <операторы>
 case константа2:
      <операторы>
```

```
 …
 default:
      <операторы>
}
```

Данный оператор последовательно проверяет на равенство переменной константам, стоящим после ключевого слова case. Если ни одна из констант не равна значению переменно, то выполняются операторы, находящиеся после слова default. Оператор switch имеет следующую особенность. Допустим, значение переменной равно значению константы1 и выполняются операторы, стоящие после первого ключевого слова case. После этого выполнение программы продолжится проверкой переменной на равенство константы2, что часто приводит к неоправданным затратам ресурсов ЭВМ. Во избежание такой ситуации следует использовать оператор break для перехода программы к следующему оператору после switch.

На листинге 2.4 представлен пример программирования условного оператора switch.

Листинг 2.4. Пример использования оператора switch.

```
#include <stdio.h>
int main()
{
 int x;
 printf("Введите число: ");
 scanf("%d",&x);
 switch(x)
 {
      case 1 : printf("Введено число 1\n");break;
      case 2 : printf("Введено число 2\n"); break;
      default : printf("Введено другое число\n");
 }
 char ch;
 printf("Введите символ: ");
 scanf("%c",&ch);
 switch(ch)
 {
      case 'a' : printf("Введен символ a\n"); break;
      case 'b' : printf("Введен символ b\n"); break;
      default : printf("Введен другой символ\n");
 }
 return 0;
}
```

Данный пример демонстрирует два разных варианта использования оператора switch. В первом случае выполняется анализ введенной цифры, во втором – анализ введенного символа. Следует отметить, что данный оператор может производить выбор только на основании равенства своего аргумента

одному из перечисленных значений case, т.е. проверка выражений типа x<0 в данном случае невозможна.

2.2. Операторы цикла языка C++

Часто при создании программ на ЭВМ требуется много раз выполнить одну и ту же группу операторов. Например, для вычисления суммы ряда длиной N или перебора элементов массива с целью определения наибольшего или наименьшего значения и т.п. Во всех этих случаях необходим инструмент для реализации повторяющихся операций и таким инструментом являются операторы цикла.

2.2.1. Оператор цикла while

С помощью данного оператора реализуется цикл, который выполняется до тех пор, пока истинно условие цикла. Синтаксис данного оператора следующий:

```
while(<условие>)
{
 <тело цикла>
}
```

Приведем пример реализации данного цикла, в котором выполняется суммирование элементов ряда $S = \sum_{i=0}^{\infty} i$ пока $S < N$:

```
int N=20, i = 0;
long S = 0L;
while(S < N)
{
 S=S+i;
 i++;
}
```

В данном примере реализуется цикл while с условием i < N. Так как начальное значение переменной i=0, а N=20, то условие истинно и выполняется тело цикла, в котором осуществляется суммирование переменной i и увеличение ее на 1. Очевидно, что на 20 итерации значение i=20, условие станет ложным и цикл будет завершен. Продемонстрируем гибкость языка C++, изменив данный пример следующим образом:

```
int N=20, i = 0;
long S = 0L;
while((S=S+i++) < N);
```

В данном случае при проверке условия сначала выполняются операторы, стоящие в скобках, где и осуществляется суммирование элементов ряда и только, затем, проверяется условие. Результат выполнения обоих вариантов программ одинаковый и S=21. Однако последняя конструкция бывает удобной при реализации опроса клавиатуры, например, с помощью функции scanf():

```
int num;
while(scanf("%d",&mun) == 1)
{
 printf("Вы ввели значение %d\n",num);
}
```

Данный цикл будет работать, пока пользователь вводит целочисленные значения и останавливается, если введена буква или вещественное число. Следует отметить, что цикл while можно принудительно завершить даже при истинном условии цикла. Это достигается путем использования оператора break. Перепишем предыдущий пример так, чтобы цикл завершался, если пользователь введет число 0.

```
int num;
while(scanf("%d",&mun) == 1)
{
   if(num == 0) break;
 printf("Вы ввели значение %d\n",num);
}
```

Цикл завершается сразу после использования оператора break, т.е. в приведенном примере, при вводе с клавиатуры нуля функция printf() выполняться не будет и программа перейдет на следующий оператор после while. Того же результата можно добиться, если использовать составное условие в цикле:

```
int num;
while(scanf("%d",&mun) == 1 && num != 0)
{
 printf("Вы ввели значение %d\n",num);
}
```

Таким образом, в качестве условия возможны такие же конструкции, что и в операторе if.

2.2.2. Оператор цикла for

Работа оператора цикла for подобна оператору while с той лишь разницей, что оператор for подразумевает изменение значения некоторой переменной и проверки ее на истинность. Работа данного оператора продолжается до тех пор, пока истинно условие цикла. Синтаксис оператора for следующий:

```
for(<инициализация счетчика>;<условие>;<изменение значения счетчика>)
{
 <тело цикла>
}
```

Рассмотрим особенность реализации данного оператора на примере вывода таблицы кодов ASCII символов.

```
char ch;
for(ch = 'a'; ch <= 'z'; ch++)
      printf("Значение ASCII для %c - %d.\n",ch,ch);
```

В данном примере в качестве счетчика цикла выступает переменная ch, которая инициализируется символом ‘a’. Это означает, что в переменную ch заносится число 97 – код символа ‘a’. Именно так символы представляются в памяти компьютера. Код символа ‘z’ – 122, и все малые буквы латинского алфавита имеют коды в диапазоне [97; 122]. Поэтому, увеличивая значение ch на единицу, получаем код следующей буквы, которая выводится с помощью функции printf(). Учитывая все вышесказанное, этот же пример можно записать следующим образом:

```
for(char ch = 97; ch <= 122; ch++)
      printf("Значение ASCII для %c - %d.\n",ch,ch);
```

Здесь следует отметить, что переменная ch объявлена внутри оператора for. Это особенность языка C++ - возможность объявлять переменные в любом месте программы.

Существует много особенностей реализации данного оператора, отметим основные из них, которые могут заметно повысить скорость написания программ. Следующим примером продемонстрируем особенности изменения значения счетчика цикла.

```
int line_cnt = 1;
double debet;
for(debet = 100.0; debet < 150.0; debet = debet*1.1, line_cnt++)
 printf("%d. Ваш долг теперь равен %.2f.\n",line_cnt, debet);
```

Следующий фрагмент программы демонстрирует возможность программирования сложного условия внутри цикла.

```
int exit = 1;
for(int num = 0;num < 100 && !exit; num += 1)
{
 scanf("%d",&mov);
```

```
 if(mov == 0) exit = 0;
 printf("Произведение num*mov = %d.\n",num*mov);
}
```

Оператор for с одним условием:

```
int i=0;
for(;i < 100;) i++;
```

и без условия

```
int i=0;
for(;;;) {i++; if(i > 100) break;}
```

В последнем примере оператор break служит для выхода из цикла for, т.к. он будет работать «вечно» не имея никаких условий.

2.2.3. Оператор цикла do while

Все представленные выше операторы циклов, так или иначе, проверяют условие перед выполнением цикла, благодаря чему существует вероятность, что операторы внутри цикла никогда не будут выполнены. Такие циклы называют циклы с предусловием. Однако бывают ситуации, когда целесообразно выполнять проверку условия после того, как будут выполнены операторы, стоящие внутри цикла. Это достигается путем использования операторов do while, которые реализуют цикл с постусловием. Следующий пример демонстрирует реализацию такого цикла.

```
const int secret_code = 13;
int code_ent;
do
{
 printf("Введите секретный код: ");
  scanf("%d",&code_ent);
}
while(code_ent != secret_code);
```

Из приведенного примера видно, что цикл с постусловием работает до тех пор, пока истинно условие, т.е. в данном случае пока значение введенного кода будет отличаться от значения секретного кода. Также следует обратить внимание на то, что после ключевого слова while должна стоять точка с запятой. При реализации данного цикла можно использовать составные условия, подобно циклу while, а также принудительно выходить из цикла с помощью оператора break.

2.2.4. Программирование вложенных циклов

Все рассмотренные выше операторы циклов допускают использование любых других операторов языка C++ внутри цикла, в том числе и операторов цикла. Это значит, что внутри одного цикла может находиться другой, что приводит к реализации вложенных циклов. Вложенные циклы необходимы для решения большого числа задач, например, вычисления двойных, тройных и т.д. сумм, просмотр элементов двумерного массива и многих других задач. В качестве примера вложенных циклов рассмотрим задачу вычисления суммы двойного ряда $S = \sum_{i=0}^{N} \sum_{j=0}^{M} i * j$:

```
long S = 0L;
int M = 10, N = 5;
for(int i = 0; i <= N;i++)
{
 for(int j = 0;j <= M;j++)
      S += i*j;
}
```

Того же результата можно добиться и с помощью оператора цикла while.

2.3. Функции

В ранее рассмотренных примерах неоднократно использовались различные функции подключаемых библиотек. Вместе с тем существующих функций языка C++ недостаточно для написания собственных программ и возникает необходимость создания своих функций. В связи с этим нужно понимать, в каких случаях целесообразно создавать свои функции. Обычно это делается для избавления много раз писать один и тот же код в программе. Например, если часто выполняются действия копирования одной строки в другую, то такую операцию лучше определить в виде функции и использовать ее по мере необходимости. Для объявления функции используется следующий синтаксис:

```
<тип> <имя функции> ([список параметров]) { <тело функции> }
```

Тип определяет возвращаемый тип функции. Имя функции служит для ее вызова в программе и ее правило определения совпадает с правилом определения имен переменных. Список параметров необходим для передачи функции каких-либо данных при ее вызове. Телов функции – это набор операторов, которые выполняются при ее вызове. Следующий пример показывает правило определения пользовательских функций.

Листинг 2.5. Пример задания функции.

```
double square(double x)
```

```
{
 x = x*x;
 return x;
}
int main()
{
 double arg = 5;
 double sq1=square(arg);
 double sq2=square(3);
 return 0;
}
```

В данном примере задается функция с именем square, которая принимает один входной параметр типа double, возводит его в квадрат и возвращает вычисленное значение вызывающей программе с помощью оператора return. Следует отметить, что работа функции завершается при вызове оператора return. Даже если после этого оператора будут находиться другие операторы, то они выполняться не будут. Например,

```
int square(int x)
{
 x = x*x;
 return x;
 printf("%d",x);
}
```

при вызове данной функции оператор printf() не будет выполнен никогда, т.к. оператор return завершит работу функции square. Оператор return является обязательным, если функция возвращает какие-либо значения. Если же она имеет тип void, т.е. ничего не возвращает, то оператор return может не использоваться.

Пользуясь рассмотренными правилами, можно создавать множество своих функций. При этом важно, чтобы объявление функции было раньше ее использования в программе подобно переменным. Именно поэтому во всех примерах объявление функций осуществляется до функции main(), в которой они вызываются.

Функция может принимать произвольное число аргументов, но возвращает только один или не одного (тип void). Для задания нескольких аргументов функции используется следующая конструкция:

```
void show(int x,int y,int z) {}
```

Здесь следует обратить внимание на то, что каждой переменной в списке аргументов функции предшествует ее тип. В отличие от объявления обычных переменных. Поэтому следующая программная строка приведет к сообщению об ошибке на этапе компиляции:

```
void show(int x, y, z) {}      //неверное объявление
```

Если число пользовательских функций велико (50 и выше), то возникает неудобство в их визуальном представлении в общем тексте программы. Действительно, имея список из 100 разных функций с их реализациями, в них становится сложно ориентироваться и вносить необходимые изменения. Для решения данной проблемы в языке C++ при создании своих функций можно пользоваться правилом: сначала задаются объявления функции, а затем их реализации. Этот подход демонстрируется в листинге 2.6.

Листинг 2.6. Использование прототипов функций.

```
#include <stdio.h>
double square(double x);
void my_puts(char ch, int cnt);
int main()
{
 double sq = square(4.5);
 char ch;
 ch = 'a';
 my_puts(ch,10);
 return 0;
}
double square(double x)
{
 x=x*x;
 return x;
}
void my_puts(char ch, int cnt)
{
 for(int i = 0;i < cnt;i++)
      putchar(ch);
}
```

В данном примере сначала объявляются две пользовательские функции без их реализаций (тела функции). Затем описывается функция main() с основной логикой программы, а после нее определяются реализации пользовательских функций. Такой подход позволяет более наглядно представить список введенных функций и проще в них ориентироваться при создании программы. Объявление функции без ее реализации называется прототипом функции.

Язык C++ позволяет задавать функции с одинаковыми именами, но разными типами входных аргументов. Следующий пример демонстрирует удобство использования таких функций при их вызове.

Листинг 2.7. Пример использования перегруженных функций.

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

```
double abs(double arg);
float abs(float arg);
int abs(int arg);
int main()
{
 double a_d = -5.6;
 float a_f = -3.2;
 int a_i;
 a_d = abs(a_d);
 a_f = abs(a_f);
 a_i = abs(-8);
 return 0;
}
double abs(double arg)
{
 if(arg < 0) arg = arg*(-1);
 return arg;
}
float abs(float arg)
{
 return (arg < 0) ? -arg : arg;
}
int abs(int arg)
{
 return (arg < 0) ? -arg : arg;
}
```

В представленной программе задаются три функции с именем abs и разными входными и выходными аргументами для вычисления модуля числа. Благодаря такому объявлению при вычислении модуля разных типов переменных в функции main() используется вызов функции с одним и тем же именем abs. При этом компилятор в зависимости от типа переменной автоматически выберет нужную функцию. Такой подход к объявлению функций называется перегрузкой.

В языке C++ можно задавать значения аргументов функции, которые будут использоваться по умолчанию, т.е. если программист не введет свое значение. Приведенный ниже фрагмент программы демонстрирует правило использования аргументов по умолчанию.

```
void some_func(int a = 1, int b = 2, int c = 3)
{
 printf("a = %d, b = %d, c = %d\n",a,b,c);
}
```

Благодаря начальной инициализации значений переменных, функция some_func() может быть вызвана с разным набором аргументов:

```
int main(void)
{
```

```
 show_func();
 show_func(10);
 show_func(10,20);
 show_func(10,20,30);
 return 0;
}
```

В результате, на экране появятся следующие строки:

a = 1, b = 2, c = 3
a = 10, b = 2, c = 3
a = 10, b = 20, c = 3
a = 10, b = 20, c = 30

Из полученного результата видно, что по умолчанию значения аргументов равны установленным значениям при определении функции. В случае ввода новых значений, переменные a, b и c соответственно меняют свои значения на введенные.

При использовании значений аргументов по умолчанию следует пользоваться правилом: аргументы со значениями по умолчанию должны находиться в списке аргументов функции последними. Следующий пример показывает правильные и неправильные объявления функций:

```
void my_func(int a, int b = 1, int c = 1);//правильное объявление
void my_func(int a, int b, int c = 1);     //правильное объявление
void my_func(int a=1, int b, int c = 1); //неправильное объявление
void my_func(int a, int b = 1, int c);    //неправильное объявление
```

В языке C++ допускается чтобы функция вызывала саму себя. Этот процесс называется рекурсией. В некоторых задачах программирования такой подход позволяет заметно упростить создаваемый программный код. Рассмотрим данный процесс на следующем примере.

Листинг 2.8. Пример использования рекурсивных функций.

```
#include <stdio.h>
void up_and_down(int );
int main(void)
{
 up_and_down(1);
 return 0;
}
void up_and_down(int n)
{
 printf("Уровень вниз %d\n",n);
 if(n < 4) up_and_down(n+1);
 printf("Уровень вверх %d\n",n);
}
```

Результатом работы этой программы будет вывод на экран следующих строк:

Уровень вниз 1
Уровень вниз 2
Уровень вниз 3
Уровень вниз 4
Уровень вверх 4
Уровень вверх 3
Уровень вверх 2
Уровень вверх 1

Полученный результат работы программы объясняется следующим образом. Вначале функция main() вызывает функцию up_and_down() с аргументом 1. В результате аргумент n данной функции принимает значение 1 и функция printf() печатает первую строку. Затем выполняется проверка и если n < 4, то снова вызывается функция up_and_down() с аргументом на 1 больше n+1. В результате вновь вызванная функция печатает вторую строку. Данный процесс продолжается до тех пор, пока значение аргумента не станет равным 4. В этом случае оператор if не сработает и вызовется функция printf(), которая печатает пятую строку «Уровень вверх 4». Затем функция завершает свою работу и управление передается функции, которая вызывала данную функцию. Это функция up_and_down() с аргументом n=3, которая также продолжает свою работу и переходит к оператору printf(), который печатает 6 строку «Уровень вверх 3». Этот процесс продолжается до тех пор, пока не будет достигнут исходный уровень, т.е. первый вызов функции up_and_down() и управление вновь будет передано функции main(), которая завершит работу программы.

Того же самого эффекта можно было бы добиться, если ввести 4 похожих функции fun1(), fun2(), fun3() и fun4() и вызывать их следующим образом.

Листинг 2.9. Аналог предыдущей программы без рекурсивных функций.

```
#include <stdio.h>
void fun1(int );
void fun2(int );
void fun3(int );
void fun4(int );
int main(void)
{
 fun1(1);
 return 0;
}
void fun1(int n)
{
 printf("Уровень вниз %d\n",n);
```

```
 if(n < 4) fun2(n+1);
 printf("Уровень вверх %d\n",n);
}
void fun2(int n)
{
 printf("Уровень вниз %d\n",n);
 if(n < 4) fun3(n+1);
 printf("Уровень вверх %d\n",n);
}
void fun3(int n)
{
 printf("Уровень вниз %d\n",n);
 if(n < 4) fun4(n+1);
 printf("Уровень вверх %d\n",n);
}
void fun4(int n)
{
 printf("Уровень вниз %d\n",n);
 printf("Уровень вверх %d\n",n);
}
```

Как видно здесь при достижении того же результата получается заметно более громоздкий код и, кроме того, привязанный к конкретному числу 4. Если бы потребовалось реализовать 20 вложений, то и число функций пришлось бы задавать столько же, что довольно неудобно. Поэтому рекурсия может давать заметно лучший и удобный программный код.

2.4. Область видимости переменных

В предыдущих примерах часто объявлялись и использовались различные переменные. При этом важно знать, когда эти переменные доступны для использования или другими словами определить их область видимости. Рассмотрим такой пример.

Листинг 2.10. Пример использования локальных переменных.

```
#include <stdio.h>
int main()
{
 for(int i=0;i < 10;i++)
 {
      int k = i*i;
      printf("%d\n",k);
 }
 return 0;
}
```

В этом примере объявляются две переменные i и k. Причем переменная k объявлена внутри цикла for. Спрашивается можно ли ее использовать и за пределами этого цикла? В данном случае ответ будет отрицательный, т.к.

переменная k «пропадает» за пределами тела цикла и существует только внутри него. Условия данного примера можно обобщить и сказать, что обычная переменная объявленная внутри фигурных скобок {} видна только в них и не может быт использована за их пределами. По этой причине переменные объявленные, например, внутри функции main() недоступны в других функциях и наоборот. Однако если объявить переменную вначале программы, а не внутри функции, то они становятся доступными всем функциям и будут иметь глобальную область видимости. Такие переменные называются глобальными. Следующий пример показывает объявление и использование глобальных переменных.

Листинг 2.11. Определение глобальных переменных.

```
#include <stdio.h>
int global_var = 0;
void my_func(void) {global_var = 10;}
int main()
{
 printf("%d\n",global_var);
 my_func();
 printf("%d\n",global_var);
 return 0;
}
```

Результатом выполнения программы будет следующий текст:

```
0
10
```

В этом примере объявлена глобальная переменная global_var, которая может использоваться и в функции main() и в функции my_func() и соответственно менять свое значение.

В языке C++ можно задавать переменные с одинаковыми именами и типами если они принадлежат разной области видимости. Например, можно задать глобальную переменную var типа int и такую же переменную внутри функции main(). В этом случае простое обращение к переменной по ее имени будет означать работу с локальной переменной, а если необходимо работать с глобальной, то перед ее именем необходимо поставить два двоеточия '::'. Следующий пример показывает такой способ работы с переменными.

Листинг 2.12. Работа с глобальными и локальными переменными.

```
#include <stdio.h>
int var = 5;
int main()
{
```

```
 int var = 10;
 printf("var = %d, ::var = %d",var,::var);
 return 0;
}
```

В результате получим

var = 10, ::var = 5

В языке C++ можно задавать класс переменных, которые будучи объявленными внутри фигурных скобок {} не исчезают и за их пределами, но в то же время имеют область видимости только внутри них. Класс таких переменных называется статическим и задается с помощью ключевого слова static. Поясним сказанное на следующем примере.

Листинг 2.13. Использование статических переменных.

```
#include <stdio.h>
void iter(void);
int main()
{
 for(int i=0;i < 4;i++)
      iter();
 return 0;
}
void iter(void)
{
 int var = 1;
 static int var_st = 1;
 printf("var = %d, var_st = %d\n",var++,var_st++);
}
```

В результате на экране появятся следующие строчки:

var = 1, var_st = 1
var = 1, var_st = 2
var = 1, var_st = 3
var = 1, var_st = 4

Анализ полученных результатов показывает, что статическая переменная var_st объявленная внутри функции iter() не исчезает после ее завершения и при повторном вызове функции не инициализируется заново, т.к. она уже существует.

В языке C++ переменную можно задать как константу, т.е. ее значение нельзя изменять в процессе выполнения программы. Это бывает полезно, когда программист хочет обеспечить «защиту» переменной от изменения. Это

достигается путем использования ключевого слова const, которое ставится перед типом переменной:

```
const int var_const = 10;        //правильная инициализация
const int array[] = {10,20,30}; // правильная инициализация
var_const = 5;                   //ошибка
```

Например, ранее рассмотренная функция strcpy() копирования одной строки в другую принимает следующие параметры:

```
char* strcpy(char* dest, const char* src);
```

Здесь идентификатор const гарантирует, что строка src не будет изменена внутри функции strcpy().

Контрольные вопросы и задания

1. Запишите условный оператор if для определения знака переменной var.
2. В каких случаях следует использовать оператор switch?
3. Используя условный оператор, выполните проверку на принадлежность значения переменной диапазону [10; 20).
4. Приведите программу замены малых латинских букв большими с использованием оператора switch.
5. Как записывается логическое равенство в операторе if?
6. Приведите обозначение логического знака «не равно».
7. Какими символами обозначаются логические операции И и ИЛИ в условном операторе if?
8. Запишите программу подсчета суммы ряда $\sum_{n=1}^{10} 1/n$ с помощью оператора цикла for.
9. В чем отличия между операторами while и do while?
10. Приведите программу для вычисления суммы $S = \sum_{i} 2/i$, пока $S < 50$.
11. Дайте понятие вложенного цикла.
12. Запишите прототип функции, которая принимает два целочисленных аргумента и возвращает вещественное число.
13. Допустим, даны три функции:
    int abs(int x);
    float abs(float x);
    long abs(long x);

Какая из этих трех функций будет вызвана в строке float a = abs(-6);?

14. Запишите функцию возведения числа в квадрат.
15. Дайте понятие рекурсии.
16. В каких задачах целесообразно использовать рекурсивные функции?

17.Приведите функцию с тремя аргументами, один из которых задан со значением по умолчанию.

18.Для чего используется ключевое слово const в языке C++?

19.Дайте понятие статических переменных и какие особенности их использования существуют?

ГЛАВА ТРЕТЬЯ

РАСШИРЕННОЕ ПРЕДСТАВЛЕНИЕ ДАННЫХ

В языке C++ наряду с простыми переменными существуют более сложные типы данных, такие как массивы, структуры, объединения и перечисления. Представленные типы упрощают программирование ряда задач, связанных, например, с представлением сигнала или изображения в памяти компьютера, объединением разнородных данных в одной переменной, использованием осмысленных имен вместо чисел и т.п.

3.1. Массивы

Представление данных в виде отдельных переменных не всегда достаточно при программировании реальных задач. Например, для представления поведения сигнала во времени или хранения информации об изображении удобно использовать специальный тип данных – массивы. Одномерные массивы можно ассоциировать с компонентами вектора, а двумерные – с матрицами. В общем случае массив – это набор элементов

данных одного типа, для объявления которого используется следующий синтаксис:

```
<тип данных> <имя массива>[число элементов];
```

Например,

```
int array_int[100];  //одномерный массив 100 целочисленных
                     // элементов
double array_d[25];  //одномерный массив 25 вещественных
                     // элементов
```

Как видно из примеров, объявление массивов отличается от объявления обычных переменных наличием квадратных скобок []. Также имена массивов выбираются по тем же правилам, что и имена переменных. Обращение к отдельному элементу массива осуществляется по номеру его индекса. Следующий фрагмент программы демонстрирует запись в массив значений линейной функции $f(x) = kx + b$ и вывода значений на экран:

```
double k=0.5, b = 10.0;
double f[100];
for(int x=0;i < 100;i++)
{
 f[i] = k*x+b;
 printf("%.2f ",f[i]);
}
```

В языке C++ предусмотрена возможность инициализации массива в момент его объявления, например, таким образом

```
int powers[4] = {1, 2, 4, 6};
```

В этом случае элементу powers[0] присваивается значение 1, powers[1] – 2, и т.д. Особенностью инициализации массивов является то, что их размер можно задавать только константами, а не переменными. Например, следующая программа приведет к ошибке при компиляции:

```
int N=100;
float array_f[N];    //ошибка, так нельзя
```

Поэтому при объявлении массивов обычно используют такой подход:

```
#include <stdio.h>
#define N  100
int main()
{
 float array_f[N];
```

```
 return 0;
}
```

Следует отметить, что при инициализации массивов число их элементов должно совпадать с его размерностью. Рассмотрим вариант, когда число элементов при инициализации будет меньше размерности массива.

```
#define SIZE    4
int data[SIZE]={512, 1024};
for(int i = 0;i < SIZE;i++)
      printf("%d, \n",data[i]);
```

Результат работы программы будет следующим:

512, 1024, 0, 0

Из полученного результата видно, что неинициализированные элементы массива принимаются равными нулю. В случаях, когда число элементов при инициализации превышает размерность массива, то при компиляции произойдет ошибка. Поэтому, когда наперед неизвестно число элементов, целесообразно использовать такую конструкцию языка C++:

```
int data[] = {2, 16, 32, 64, 128, 256};
```

В результате инициализируется одномерный массив размерностью 6 элементов. Здесь остается последний вопрос: что будет, если значение индекса при обращении к элементу массива превысит его размерность? В этом случае ни программа, ни компилятор не выдадут значение об ошибке, но при этом в программе могут возникать непредвиденные ошибки. Поэтому программисту следует обращать особое внимание на то, чтобы индексы при обращении к элементам массива не выходили за его пределы. Также следует отметить, что первый элемент массива всегда имеет индекс 0, второй – 1, и т.д.

Для хранения некоторых видов информации, например, изображений удобно пользоваться двумерными массивами. Объявление двумерных массивов осуществляется следующим образом:

```
int array2D[100][20];     //двумерный массив 100x20 элементов
```

Нумерация элементов также начинается с нуля, т.е. array2D[0][0] соответствует первому элементу, array2D[0][1] – элементу первой строки, второго столбца и т.д. Для начальной инициализации двумерного массива может использоваться следующая конструкция:

```
long array2D[3][2] = {{1, 2}, {3, 4}, {5, 6}};
```

или

```
long array2D[][] = {{1, 2}, {3, 4}, {5, 6}};
```

В общем случае можно задать массив любой размерности и правила работы с ними аналогичны правилам работы с одномерными и двумерными массивами.

3.2. Работа со строками

В языке C++ нет специального типа данных для строковых переменных. Для этих целей используются массивы символов (тип char). Следующий пример демонстрирует использование строк в программе:

```
char str_1[100] = {'П','р','и','в','е','т','\0'};
char str_2[100] = "Привет";
char str_3[] = "Привет";
printf("%s\n%s\n%s\n",str_1,str_2,str_3);
```

В приведенном примере показаны три способа инициализации строковых переменных. Первый способ является классическим объявлением массива, второй и третий используются специально для строк. Причем в последнем случае, компилятор сам определяет нужную длину массива для записи строки. Анализируя первый и второй способы инициализации массива символов возникает вопрос: каким образом язык C++ «знает» где заканчивается строка? Действительно, массив str_2 содержит 100 элементов, а массив str_3 меньше 100, тем не менее длина строки и в первом и во втором случаях одна и та же. Такой эффект достигается за счет использования специальных управляющих кодов, которые говорят где заканчивается строка или где используется перенос внутри одной строки и т.п. В частности символ '\0' означает в языке C++ конец строки и все символы после него игнорируются как символы строки. Следующий пример показывает особенность использования данного специального символа.

```
char str1[10] = {'Л','е','к','ц','и','я','\0'};
char str2[10] = {'Л','е','к','ц', '\0','и','я' };
char str3[10] = {'Л','е', '\0','к','ц','и','я' };
printf("%s\n%s\n%s\n",str1,str2,str3);
```

Результатом работы данного кода будет вывод следующих трех строк:

Лекция
Лекц
Ле

Из этого примера видно как символ конца строки ‘\0’ влияет на длину строк. Таким образом, чтобы подсчитать длину строки (число символов) необходимо считать символы до тех пор, пока не встретится символ ‘\0’ или не будет достигнут конец массива. В листинге 3.1 представлена программа вычисления длины строки.

Листинг 3.1. Программа вычисления длины строки.

```
#include <stdio.h>
int main(void)
{
 char str[] = “Привет мир!”;
 int size_array = sizeof(str);
 int length = 0;
 while(length < size_array && str[length] != ‘\0’) length++;
 printf(“Длина строки = %d.\n”,length);

 return 0;
}
```

В представленном примере сначала выполняется инициализация строки в массиве str. Затем вычисляется размер массива с помощью функции sizeof(), которая возвращает число байт занимаемое массивом в памяти ЭВМ. Учитывая, что тип char также представляет собой один байт, то данная функция даст размер массива. После этого инициализируется счетчик символов length и выполняется цикл while с очевидными условиями. В результате переменная length будет содержать число символов в строке, либо размер массива. Подобная функция вычисления размера строк уже реализована в стандартной библиотеке языка C++ string.h со следующим синтаксисом:

```
int strlen(char* str);
```

где char* str – указатель на строку (об указателях речь пойдет ниже). Следующая программа показывает правило использования функции strlen().

Листинг 3.2. Пример использования функции strlen().

```
#include <stdio.h>
#include <string.h>
int main(void) {
 char str[] = “Привет мир!”;
 int length = strlen(str);
 printf(“Длина строки = %d.\n”,length);
 return 0;
}
```

Результатом работы программы будет вывод на экран числа 11. Учитывая, что первый символ имеет нулевой индекс, то можно заметить, что данная функция считает и символ ‘\0’.

Теперь рассмотрим правила присваивания одной строковой переменной другой. Допустим объявлены две строки

```
char str1[] = “Это первая строка”;
char str2[] = “Это вторая строка”;
```

и необходимо выполнить оператор присваивания

```
str1 = str2;
```

При такой записи оператора присваивания компилятор выдаст сообщение об ошибке. Для того чтобы выполнить копирование необходимо перебирать по порядку элементы одного массива и присваивать их другому массиву. Это демонстрируется на следующем примере:

```
char str1[] = “Это первая строка”;
char str2[] = “Это вторая строка”;
int size_array = sizeof(str1);
int i=0;
while(i < size_array && str1[i]  != ‘\0’) {
 str2[i] = str1[i];
 i++;
}
str2[i] = ‘\0’;
```

В приведенном фрагменте программы выполняется перебор элементов массива str1 с помощью цикла while и значение i-го элемента записывается в массив str2. Данная операция выполняется до тех пор, пока либо не будет достигнут конец массива, либо не встретится символ конца строки ‘\0’. Затем, после выполнения цикла, в конец массива str2 записывается символ ‘\0’. Таким образом, выполняется копирование одной строки в другую. Подобная функция также реализована в библиотеке языка C++ string.h и имеет следующее определение:

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

Она выполняет копирование строки src в строку dest и возвращает строку dest. В листинге 3.3 показано правило использования функции strcpy().

Листинг 3.3. Пример использования функции strcpy().

```
#include <stdio.h>
#include <string.h>
int main(void) {
```

```
 char src[] = "Привет мир!";
 char dest[100];
 strcpy(dest,src);
 printf("%s\n",dest);
 return 0;
}
```

Кроме операций вычисления длины строки и копирования строк важной является операция сравнения двух строк между собой. В языке С++ две строки считаются одинаковыми, если равны их длины и элементы одной строки равны соответствующим элементам другой. Следующий алгоритм выполняет такое сравнение:

```
char str1[] = "Это первая строка";
char str2[] = "Это вторая строка";
int length = strlen(str1);
int length2 = strlen(str2);

if(length != length2) {
      printf("Срока %s не равна строке %s\n",str1,str2);
      return 0;
 }

for(int i=0;i < length;i++)
 if(str1[i] != str2[i]) {
      printf("Срока %s не равна строке %s\n",str1,str2);
      break;
 }
}
if(i == length)
      printf("Срока %s равна строке %s\n",str1,str2);
```

Приведенный пример показывает возможность досрочного завершения программы путем использования оператора return. Данный оператор вызывается, если длины строк не совпадают. Также реализуется цикл, в котором сравниваются элементы массивов str1 и str2. Если хотя бы один элемент не будет совпадать, то на экран выведется сообщение «Строка … не равна строке…» и цикл завершится с помощью оператора break. В противном случае (равенства всех элементов) переменная i будет равна переменной length и тогда на экран выводится сообщение о совпадении строк. Подобный алгоритм сравнения двух строк реализован в функции

```
int strcmp(char* str1, char* str2);
```

библиотеки <string.h>. Данная функция возвращает нуль, если строки str1 и str2 равны и не нуль в противном случае. Приведем пример использования данной функции.

```
    char str1[] = "Это первая строка";
    char str2[] = "Это вторая строка";
    if(strcmp(str1,str2) == 0) printf("Срока %s равна строке
%s\n",str1,str2);
    else printf("Срока %s не равна строке %s\n",str1,str2);
```

В языке C++ имеется несколько функций, позволяющих вводить строки с клавиатуры. Самой распространенной из них является ранее рассмотренная функция scanf(), которой в качестве параметра передается ссылка на массив символов:

```
    char str[100];
    scanf("%s",str);
```

В результате выполнения этого кода, переменная str будет содержать введенную пользователем последовательность символов. Кроме функции scanf() также часто используют функцию gets() библиотеки stdio.h, которая в качестве аргумента принимает ссылку на массив символов:

```
    gest(str);
```

Данная функция считывает символы до тех пор, пока пользователь не нажмет клавишу Enter, т.е. введет символ перевода строки '\n'. Затем она записывает вместо символа '\n' символ '\0' и передает строку вызывающей программе.

Для вывода строк на экран помимо функции printf() можно использовать также функцию puts() библиотеки stdio.h, которая более проста в использовании. Следующий пример демонстрирует применение данной функции.

```
    #define DEF "Заданная строка"
    char str[] = "Это первая строка";
    puts(str);
    puts(DEF);
    puts(&str[4]);
```

Результат работы следующий:

```
    Это первая строка
    Заданная строка
    первая строка
```

Как видно из полученных результатов, функция puts() автоматически переводит курсор на следующую строку и последующие строки выводятся с новой строки. Кроме того, строки можно задавать с помощью директивы

#define и выводить их с помощью функции puts. Также можно менять начало вывода строки путем изменения адреса ее начала.

Еще одной удобной функцией работы со строками является функция sprintf() библиотеки stdio.h. Ее действие аналогично рассмотренной ранее функции printf() с той лишь разницей, что результат вывода заносится в строковую переменную, а не на экран:

```
int age;
char name[100], str[100];
printf("Введите Ваше имя: ");
scanf("%s",name);
printf("Введите Ваш возраст: ");
scanf("%d",&age);
sprintf(str,"Здраствуйте %s. Ваш возраст %d лет",name,age);
puts(str);
```

В результате массив str будет содержать строку «Здраствуйте … Ваш возраст…».

Анализ последнего примера показывает, что с помощью функции sprintf() можно преобразовывать числовые переменные в строковые, объединять несколько строк в одну и т.п. Вместе с тем библиотека <stdlib.h> содержит специальные функции по преобразованию строк в цифры. Дело в том, что строка «100» и цифра 100 в памяти компьютера представляются по-разному. Строка «100» - это последовательность трех символов ‘1’,’0’,’0’, а число 100 – это значение, которое может быть представлено в виде одного байта или храниться в переменной типа int. При программировании часто бывает необходимо выполнять преобразование строк в числа. Это осуществляется с помощью функций atoi(), atol(), atof(). В листинге 3.4 показаны особенности применения данных функций.

Листинг 3.4. Программа преобразования строк в цифры.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
 char str_i[] = "120";
 char str_l[] = "120000";
 char str_f[] = "120.50";
 int var_i = atoi(str_i);
 long var_l = atol(str_l);
 float var_f = atof(str_f);

 return 0;
}
```

В результате выполнение данной программы, переменные var_i, var_l и var_f будут содержать значения 120, 120000 и 120.50 соответственно.

3.3. Обработка элементов массива

В практике программирования существуют устоявшиеся подходы к удалению, добавлению и сортировке элементов массива. Рассмотрим сначала алгоритм добавления элемента в массив. Допустим, что имеется строка

```
char str[100] = "Я не пропустил ниодной лекции по информатике.";
```

в которой необходимо добавить 17-й символ пробел. При добавлении нового элемента в массив уже имеющиеся элементы всегда сдвигаются вправо относительно добавляемого элемента. Это связано с тем, что при сдвиге вправо, номер первого элемента остается неизменным и равен 0. Если бы сдвиг осуществлялся влево, то номер первого элемента массива становился бы отрицательным, что недопустимо для языка С++. Таким образом, для добавления 17-го элемента нужно выполнить сдвиг вправо элементов, находящихся после 17-го символа и на 17 позицию записать символ пробела. Для реализации данного алгоритма можно воспользоваться циклом for, в котором счетчик цикла меняется, начиная с последнего символа строки до номера вставляемого элемента, в данном случае 17-го. В результате получаем следующий алгоритм:

```
int size = strlen(str);
for(int i = size;i >= 17;i--)
 str[i+1] = str[i];
str[17] = ' ';
```

Здесь функция strlen() возвращает номер символа конца строки ‘\0’ поэтому при сдвиге символов будет сдвинут и этот специальный символ, который необходим для корректного представления строк. Цикл for инициализирует переменную i величиной size, которая будет указывать на символ ‘\0’ в строке str. Данный цикл выполняется до тех пор, пока величина i не будет меньше 17 и уменьшается на единицу на каждом шаге его работы. Непосредственно сдвиг элементов массива выполняется внутри цикла for, где i+1 элементу присваивается значение i-го элемента. Наконец, после цикла 17-му элементу строки присваивается символ пробела. Подобный алгоритм добавления нового элемента остается справедливым для любых типов одномерных массивов.

При удалении элемента из массива используется операция сдвига элементов влево относительно удаляемого. Воспользуемся тем же примером строки и поставим задачу удаления 4 пробела (нумерация начинается с нуля). Для этого достаточно все элементы, имеющие индексы больше 4, сместить на один символ влево. Это также можно реализовать с помощью цикла for, в

котором счетчик цикла будет меняться от 5 и до символа конца строки. В результате получим следующий алгоритм:

```
int size = strlen(str);
for(int i = 5;i <= size;i++)
 str[i-1] = str[i];
```

Непосредственное смещение элементов выполняется в теле цикла for, в котором i-1 элементу присваивается значение i-го элемента и после этого значение i увеличивается на единицу. Цикл выполняется до тех пор, пока все символы строки, начиная с 5-го, не будут смещены на единицу влево, включая и символ конца строки.

Наконец, рассмотрим алгоритм упорядочивания элементов массива по возрастанию и убыванию. Существует много разных методов упорядочения элементов. Рассмотрим наиболее простой и распространенный известный под названием «метод всплывающего пузырька». Идея метода заключается в переборе всех элементов массива, начиная с первого (нулевого) и поиска наибольшего (или наименьшего) значения. Найденное значение ставится на место первого элемента, а первый элемент на место найденного. Данная процедура повторяется, начиная уже со второго элемента массива где также находится наибольшее (или наименьшее) значение элемента из оставшихся. Затем, выполняется замена: найденный элемент перемещается на место второго, а второй на место найденного. После такой перестановки всех элементов, исключая последний, массив будет упорядочен по убыванию (или возрастанию). Описанный алгоритм сортироваки упорядочивания по убыванию значений элементов приведен ниже.

```
int array[10] = {2,5,3,7,9,10,4,6,1,8};
for(int i = 0;i < 9;i++)
{
     int max = array[i];
     int pos_max = i;
     for(int j = i+1;j < 10;j++)
          if(max < array[j]) {
               max = array[j];
               pos_max = j;
          }
     int temp = array[i];
     array[i] = array[pos_max];
     array[pos_max] = temp;
}
```

В данном алгоритме задается цикл, в котором последовательно просматриваются элементы массива array. Внутри цикла инициализируется переменная max, в которую записывается максимальное найденное значение, и переменная pos_max, в которой хранится значение индекса найденного максимального элемента. Затем, реализуется вложенный цикл for, в котором

выполняется перебор элементов массива, начиная с i+1, и заканчивая последним 10. В теле вложенного цикла выполняется проверка для поиска максимального элемента, и если текущий элемент считается таковым, то значения переменных max и pos_max меняются. После вложенного цикла осуществляется перестановка элементов массива и процесс повторяется.

Аналогичным образом реализуется алгоритм упорядочения по возрастанию, с той лишь разницей, что на каждом шаге работы алгоритма выполняется поиск не максимального, а минимального значения элемента:

```
for(int i = 0;i < 9;i++)
{
     int min = array[i];
     int pos_min = i;
     for(int j = i+1;j < 10;j++)
          if(min > array[j]) {
               min = array[j];
               pos_min = j;
          }
     int temp = array[i];
     array[i] = array[pos_min];
     array[pos_min] = temp;
}
```

3.4. Структуры

При разработке программ важным является выбор эффективного способа представления данных. Во многих случаях недостаточно объявить простую переменную или массив, а нужна более гибкая форма представления данных. Таким элементом может быть структура, которая позволяет включать в себя разные типы данных, а также другие структуры. Приведем пример, в котором использование структуры позволяет эффективно представить данные. Таким примером будет инвентарный перечень книг, в котором для каждой книги необходимо указывать ее наименование, автора и год издания. Причем количество книг может быть разным, но будем полгать, что не более 100. Для хранения информации об одной книге целесообразно использовать структуру, которая задается в языке C++ с помощью ключевого слова struct, за которым следует ее имя. Само определение структуры, т.е. то, что она будет содержать, записывается в фигурных скобках {}. В данном случае структура будет иметь следующий вид:

```
struct book {
 char title[100];    //наименование книги
 char author[100];   //автор
 int year;           //год издания
};
```

Такая конструкция задает своего рода шаблон представления данных, но не сам объект, которым можно было бы оперировать подобно переменной или массиву. Для того чтобы объявить переменную для структуры с именем book используется такая запись:

```
struct book lib;     //объявляется переменная типа book
```

После объявления переменной lib имеется возможность работать со структурой как с единым объектом данных, который имеет три поля: title, author и year. Обращение к тому или иному полю структуры осуществляется через точку: lib.title, lib.author и lib.year. Таким образом, для записи в структуру информации можно использовать следующий фрагмент программы:

```
printf("Введите наименование книги: ");
scanf("%s",lib.title);
printf("Введите автора книги: ");
scanf("%s",lib.author);
printf("Введите год издания книги: ");
scanf("%d",&lib.year);
```

После этого в соответствующие поля будет записана введенная с клавиатуры информация и хранится в единой переменной lib. Однако по условиям задачи необходимо осуществлять запись не по одной, а по 100 книгам. В этом случае целесообразно использовать массив структур типа book, который можно задать следующим образом:

```
struct book lib[100];
```

В этом случае программу ввода и хранения информации по книгам можно записать в виде:

Листинг 3.5. Инвентарный перечень книг.

```
#include <stdio.h>
struct book {
 char title[100];    //наименование книги
 char author[100];   //автор
 int year;      //год издания
};

int main()
{
 int cnt_book = 0, ch;
 struct book lib[100];
 do
 {
      printf("Введите наименование книги: ");
      scanf("%s",lib[cnt_book].title);
```

```
        printf("Введите автора книги: ");
        scanf("%s",lib[cnt_book].author);
        printf("Введите год издания книги: ");
        scanf("%d",&lib.year);
        printf("Нажмите q для завершения ввода: ");
        cnt_book++;
    }
    while(scanf("%d",ch) == 1 && cnt_book < 100);
    return 0;
}
```

Данный пример показывает удобство хранения информации по книгам. Тот же алгоритм в общем случае можно реализовать и без структуры, но тогда пришлось бы использовать два двумерных массива символов и один одномерный массив для хранения года издания. Несмотря на то, что формально такая запись была бы корректной с точки зрения языка C++, но менее удобна в обращении. Графически массив структур можно представить в виде таблицы, в которой роль столбцов играют поля, а роль строк элементы массива структур.

	название	автор	год издания
lib[0]	lib[0].title	lib[0].author	lib[0].year
lib[1]	lib[1].title	lib[1].author	lib[1].year
lib[2]	lib[2].title	lib[2].author	lib[2].year
⋮			
lib[99]	lib[99].title	lib[99].author	lib[99].year

Структуры можно автоматически инициализировать при их объявлении подобно массивам, используя следующий синтаксис:

```
struct bool lib = {
  "Евгений Онегин",
  "Пушкин А.С.",
  1995
};
```

При выполнении данного фрагмента программы в переменные структуры title, author и year будет записана соответственно информация: “Евгений Онегин”, “Пушкин А.С.”, 1995. Здесь следует обратить внимание, что последовательность данных при инициализации должна соответствовать последовательности полей в структуре. Это накладывает определенные ограничения, т.к. при инициализации необходимо помнить последовательность полей в структуре. Стандарт C99 допускает более гибкий механизм инициализации полей структуры:

```
struct book lib = {.year = 1995,
                   .author = “Пушкин А.С.”,
                   .title = “Евгений Онегин” };
```

или

```
struct book lib = { .year = 1995,
                   .title = “Евгений Онегин” };
```

или

```
struct book lib = {.author = “Пушкин А.С.”,
                   .title = “Евгений Онегин”,
                   1995 };
```

В первом и во втором примерах при инициализации указываются наименования полей через точку. При этом их порядок и число не имеет значения. В третьем примере первые два поля указаны через имена, а последнее инициализируется по порядковому номеру – третьему, который соответствует полю year.

В некоторых случаях имеет смысл создавать структуры, которые содержат в себе другие (вложенные) структуры. Например, при создании простого банка данных о сотрудниках предприятия целесообразно ввести, по крайней мере, две структуры. Одна из них будет содержать информацию о фамилии, имени и отчестве сотрудника, а вторая будет включать в себя первую с добавлением полей о профессии и возрасте:

```
struct tag_fio {
 char last[100];
 char first[100];
 char otch[100];
};
struct tag_people {
 struct tag_fio fio; //вложенная структура
 char job[100];
 int old;
};
```

Рассмотрим способ инициализации и доступ к полям структуры people на следующем примере.

Листинг 3.6. Работа с вложенными структурами.

```
int main()
{
 struct people man = {
```

```
        {“Иванов”, “Иван”, “Иванович”},
        “Электрик”,
        50 };
 printf(“Ф.И.О.:%s %s %s\n”,man.fio.last,man.fio.first,
                                              man.fio.otch);
 printf(“Профессия : %s \n”,man.job);
 printf(“Возраст : %d\n”,man.old);
 return 0;
}
```

В данном примере показано, что для инициализации структуры внутри другой структуры следует использовать дополнительные фигурные скобки, в которых содержится информация для инициализации полей фамилии, имени и отчества сотрудника. Для того чтобы получить доступ к полям вложенной структуры выполняется сначала обращение к ней по имени man.fio, а затем к ее полям: man.fio.last, man.fio.first и man.fio.otch. Используя данное правило, можно создавать многоуровневые вложения для эффективного хранения и извлечения данных.

Структуры, как и обычные типы данных, можно передавать функции в качестве аргумента. Следующий пример демонстрирует работу функции отображения полей структуры на экран.

Листинг 3.7. Передача структур через аргументы функции.

```
#include <stdio.h>
struct tag_people {
 char name[100];
 char job[100];
 int old;
};
void show_struct(struct tag_people man);
int main()
{
 struct tag_people person = {“Иванов”,”Электрик”,30};
 show_struct(person);
 return 0;
}
void show_struct(struct tag_people man)
{
 printf(“Имя: %s\n”,man.name);
 printf(“Профессия: %s\n”,man.job);
 printf(“Возраст: %d\n”,man.old);
}
```

В приведенном примере используется функция с именем show_struct, которая имеет тип аргумента struct tag_people и переменную-структуру man. При передаче структуры функции создается ее копия, которая доступная в теле функции show_struct под именем man. Следовательно, любые изменения полей структуры с именем man никак не повлияют на содержание структуры с

именем person. Вместе с тем иногда необходимо выполнять изменение полей структуры функции и возвращать измененные данные вызывающей программе. Для этого можно задать функцию, которая будет возвращать структуру, как показано в листинге 3.8.

Листинг 3.8. Функции, принимающие и возвращающие струткуру.

```
#include <stdio.h>
struct tag_people {
 char name[100];
 char job[100];
 int old;
};
void show_struct(struct tag_people man);
struct tag_people get_struct();
int main()
{
 struct tag_people person;
 person = get_struct();
 show_struct(person);
 return 0;
}
void show_struct(struct tag_people man)
{
 printf("Имя: %s\n",man.name);
 printf("Профессия: %s\n",man.job);
 printf("Возраст: %d\n",man.old);
}
struct tag_people get_struct()
{
 struct tag_people man;
 scanf("%s",man.name);
 scanf("%s",man.job);
 scanf("%d",man.old);
 return man;
}
```

В данном примере используется функция get_struct(), которая инициализирует структуру с именем man, запрашивает у пользователя ввод значений ее полей и возвращает введенную информацию главной программе. В результате выполнения оператора присваивания структуры man структуре person, происходит копирование информации соответствующих полей и автоматическое удаление структуры man.

Функциям в качестве аргумента можно также передавать массивы структур. Для этого используется следующее определение:

```
void show_struct(struct people mans[], int size);
```

Здесь size – число элементов массива, которое необходимо для корректного считывания информации массива mans. Следующий пример показывает принцип работы с массивами структур.

Листинг 3.9. Передача массив структур функции.

```
#include <stdio.h>
#define N 2
struct tag_people {
 char name[100];
 char job[100];
 int old;
};
void show_struct(struct people mans[], int size);
int main()
{
 struct people persons[N] = {
 { “Иванов”, «Электрик», 35 },
 { “Петров”, «Преподаватель», 50 },
 };
 show_struct(persons, N);
}
void show_struct(struct people mans[], int size)
{
 for(int i = 0;i < size;i++) {
      printf(“Имя: %s\n”,mans[i].name);
      printf(“Профессия: %s\n”,mans[i].job);
      printf(“Возраст: %d\n”,mans[i].old);
 }
}
```

При передаче аргумента persons выполняется копирование информации в массив mans и указывается дополнительный параметр size, для определения числа элементов массива mans. Затем в функции show_struct() реализуется цикл, в котором выполняется отображение информации массива структуры на экран монитора.

3.5. Битовые поля

Стандарт C99, который часто является основой языка C++, позволяет описывать данные на уровне битов. Это достигается путем использования битовых полей, представляющие собой переменные типов signed или unsigned int, у которых используются лишь несколько бит для хранения данных. Такие переменные обычно записываются в структуру и единую последовательность бит. Рассмотрим пример, в котором задается структура flags, внутри которой задано 8 битовых полей:

```
struct {
```

```
 unsigned int first : 1;
 unsigned int second : 1;
 unsigned int third : 1;
 unsigned int forth : 1;
 unsigned int fifth : 1;
 unsigned int sixth : 1;
 unsigned int sevnth : 1;
 unsigned int eighth : 1;
} flags;
```

Теперь, для определения того или иного бита переменной flags достаточно воспользоваться операцией

```
flags.first = 1;
flags.third = 1;
```

В этом случае будут установлены первый и третий биты, а остальные равны нулю, что соответствует числу 5. Данное значение можно отобразить, воспользовавшись функцией printf():

```
printf("flags = %d.\n",flags);
```

но переменной flags нельзя присваивать значения как обычной переменной, т.е. следующий программный код будет неверным:

```
flags = 5; //неверно, так нельзя
```

Также нельзя присваивать значение flags переменным, например, следующая запись приведет к сообщению об ошибке:

```
int var = flags; //ошибка, структуру нельзя присваивать переменной
```

Так как поля first,…, eighth могут содержать только один бит информации, то они принимают значения 0 или 1 для типа unsigned int и 0 и -1 - для типа signed int. Если полю присваивается значение за пределами этого диапазона, то оно выбирает первый бит из присваиваемого числа.

В общем случае можно задавать любое число бит для описания полей, например

```
struct {
 unsigned int code1 : 2;
 unsigned int code2 : 2;
 unsigned int code3 : 8;
} prcode;
```

Здесь создается два двух битовых поля и одно восьмибитовое. В результате возможны следующие операции присваивания:

```
prcode.code1 = 0;
prcode.code2 = 3;
prcode.code3 = 128;
```

Структуры flags и prcode удобно использовать в условных операторах if и switch. Рассмотрим пример использования структуры битовых полей для описания свойств окна пользовательского интерфейса.

```
struct tag_window {
 unsigned int show : 1    //показать или скрыть
 unsigned int style : 3   //WS_BORDER, WS_CAPTION, WS_DLGFRAME
 unsigned int color : 3   //RED, GREEN, BLUE
} window;
Определим следующие константы:
#define WS_BORDER  1      //001
#define WS_CAPTION 2      //010
#define WS_DLGFRAME 4     //100
#define RED     1
#define GREEN 2
#define BLUE 4
#define SW_SHOW 0
#define SW_HIDE 1
```

Пример инициализации структуры битовых полей

```
window.show = SW_SHOW;
window.style = WS_BORDER | WS_CAPTION;
window.color = RED | GREEN | BLUE;  //белый цвет
```

3.6. Объединения

Еще одним важным типом представления данных являются объединения. Это тип данных, который позволяет хранить различные типы данных в одной и той же области памяти (начиная с одного и того же адреса в памяти). Объединение задается с помощью ключевого слова union подобно структуре. Покажем особенность применения объединений на примере хранения данных, которые могут быть и вещественными и целыми или представлять собой символ. Так как наперед неизвестно какой тип данных требуется сохранять, то в объединении необходимо задать три поля:

```
union tag_value {
 int var_i;
 double var_f;
 char var_ch;
};
```

Данное объединение будет занимать в памяти 8 байт, ровно столько, сколько занимает переменная самого большого объема, в данном случае var_f типа double. Все остальные переменные var_i и var_ch будут находиться в той же области памяти, что и переменная var_f. Благодаря такому подходу происходит экономия памяти, но платой за это является возможность хранения значения только одной переменной из трех в определенный момент времени. Как видно из постановки задачи такое ограничение не будет влиять на работоспособность программы. Если бы вместо объединения использовалась структура tag_value, то можно было бы сохранять значения всех трех переменных одновременно, но при этом требовалось бы больше памяти.

Для того чтобы программа «знала» какой тип переменной содержит объединение tag_value, необходимо ввести переменную, значение которой будет указывать номер используемой переменной: 0 – var_i, 1 – var_f и 2 – var_ch. Причем эту переменную удобно представить с объединением в виде структуры следующим образом:

```
struct tag_var {
 union tag_value value;
 short type_var;
};
```

Таким образом, получаем следующий алгоритм записи разнородной информации в объединение tag_value.

Листинг 3.10. Пример использования объединений.

```
int main()
{
 struct tag_var var[3];
 var[0].type_var = 0;
 var[0].value.var_i = 10;
 var[1].type_var = 1;
 var[1].value.var_f = 2.3;
 var[2].type_var = 2;
 var[2].value.var_ch = 'd';
 for(int i = 0;i < 3;i++)
 {
      switch(var[i].type_var)
      {
      case 0:printf("var = %d\n",var[i].value.var_i);break;
      case 1:printf("var = %f\n",var[i].value.var_f);break;
      case 2:printf("var = %c\n",var[i].value.var_ch);break;
      default: printf("Значение переменной не определено\n");
      }
 }
 return 0;
}
```

Как видно из примера объединение tag_value позволяет эффективно, с точки зрения объема памяти, сохранять переменные разного типа. Выигрыш в объеме памяти для данного случая незначителен около 9-15 байт (в зависимости от стандарта языка С), но если бы таких переменных было 1000 и более, то выигрыш был бы ощутимым. В этом и заключается особенность объединений – экономия памяти при хранении данных.

3.7. Перечисляемые типы

В предыдущем примере была использована переменная type_var, которая указывала номер используемой переменной, что не особенно удобно при написании сложных программ, где запомнить номер той или иной переменной структуры довольно сложно. Проще если переменную type_var инициализировать специальными словами, например,

VT_NONE – тип переменной не определен;

VT_INT – целочисленный тип;

VT_FLOAT – вещественный тип;

VT_CHAR – символьный тип.

Этого можно достичь, если type_var определить как перечисляемый тип, который задается с помощью ключевого слова enum следующим образом:

```
enum tag_type {VT_NONE, VT_INT, VT_FLOAT, VT_CHAR};
```

При этом объявление перечисляемого типа выполняется подобно объявлению структуры или объединения:

```
enum tag_type type_var;
```

В результате для введенной переменной перечисляемого типа допустимо использование следующих операторов:

```
type_var = VT_INT; //переменная type_var принимает значение VT_INT
if(type_var == VT_NONE) // проверка
for(type_var = VT_NONE; type_var <= VT_CHAR; type_var++)    //цикл
```

Анализ последнего оператора for показывает, что значения перечисляемого типа VT_NONE,…,VT_CHAR являются числами, которые имеют свои имена. Причем, VT_NONE = 0, VT_INT = 1, …, VT_CHAR = 3. В некоторых случаях использование имен удобнее использования цифр, т.к. их запоминать и ориентироваться в них проще, чем в числах.

Перепишем пример хранения разнородной информации с использованием перечисляемого типа, получим:

Листинг 3.11. Пример использования перечисляемого типа.

```
enum tag_type {VT_NONE, VT_INT, VT_FLOAT, VT_CHAR};
struct tag_var {
 union tag_value value;
 enum tag_type type_var;
};
int main()
{
 struct tag_var var[3];
 var[0].type_var = VT_INT;
 var[0].value.var_i = 10;
 var[1].type_var = VT_FLOAT;
 var[1].value.var_f = 2.3;
 var[2].type_var = VT_CHAR;
 var[2].value.var_ch = 'd';
 for(int i = 0;i < 3;i++)
 {
  switch(var[i].type_var)
  {
  case VT_INT:printf("var=%d\n",var[i].value.var_i);break;
  case VT_FLOAT:printf("var=%f\n",var[i].value.var_f);break;
  case VT_CHAR:printf("var=%c\n",var[i].value.var_ch);break;
  default: printf("Значение переменной не определено\n");
  }
 }
 return 0;
}
```

Из приведенного примера видно, что использование перечисляемого типа делает программу более понятной и удобной при программировании. Здесь следует отметить, что если объявлены два перечисления

```
enum color {red, green, blue} clr;
enum color_type {clr_red, clr_green, clr_blue} type;
```

то числовые значения red, green, blue будут совпадать с соответствующими числовыми значениями vt_int, vt_float, vt_char. Это значит, что программный код if(clr == red) будет работать также как и if(clr == clr_red). Часто это не имеет принципиального значения, но, например, следующий оператор выдаст сообщение об ошибке:

```
switch(clr)
{
case red:printf("Красный цвет\n");break;
case green:printf("Зеленый цвет\n");break;
case clr_red:printf("Красный оттенок\n");break;
};
```

Ошибка возникнет из-за того, что значение red и значение clr_red равны одному числу – 0, а оператор switch не допускает такой ситуации. Для того чтобы избежать такой ситуации, при задании перечисляемого типа допускаются следующие варианты:

```
enum color_type {clr_red = 10, clr_green, clr_blue};
enum color_type {clr_red = 10, clr_green = 20, clr_blue = 30};
enum color_type {clr_red, clr_green = 20, clr_blue};
```

В первом случае значения clr_red = 10, clr_green = 11, clr_blue = 12. Во втором - clr_red = 10, clr_green = 20, clr_blue = 30. В третьем - clr_red = 0, clr_green = 20, clr_blue = 21. Как видно из полученных значений, величины могут инициализироваться при задании перечисления, а если они не объявлены, то принимают значение на единицу больше предыдущего значения.

3.8. Типы, определяемые пользователем

Язык C++ допускает создание собственных типов данных на основе базовых, таких как int, float, struct, union, enum и др. Для этого используется ключевое слово typedef, за которым следует описание типа и его имя.

Рассмотрим действие оператора typedef на примере создания пользовательского типа с именем BYTE для объявления байтовых переменных, т.е. переменных, значения которых меняются в диапазоне от 0 до 255, и которые занимают один байт в памяти ЭВМ:

```
typedef unsigned char BYTE;
```

Здесь unsigned char – пользовательский тип; BYTE – имя введенного типа. После такого объявления слово BYTE можно использовать для определения переменных в программе:

```
BYTE var_byte;
```

Создание имени для существующего типа может показаться нецелесообразным, но иногда это имеет смысл. Так, применение оператора typedef повышает степень переносимости программного кода с одной платформы на другую. Например, тип, возвращаемый оператором sizeof, определен как size_t. Это связано с тем, что в разных реализациях языка C++ size_t определен или как unsigned int или как unsigned long для лучшей адаптации к той или иной операционной системы. Таким образом, составленный текст программы достаточно откомпилировать на соответствующей платформе и оператор sizeof автоматически «подстроится» под нее без переделки самой программы.

Кроме объявлений простых пользовательских типов оператор typedef можно использовать и при объявлении новых типов на основе структур. Например, удобно ввести тип COMPLEX для объявления переменных комплексных чисел. Для этого можно воспользоваться следующим кодом:

```
typedef struct complex {
 float real;
 float imag;
} COMPLEX;
```

и работать с комплексными числами

```
COMPLEX var_cmp1, var_cmp2, var_cmp3;
var_cmp1.real = 10;
var_cmp1.imag = 5.5;
var_cmp2.real = 6.3;
var_cmp2.imag = 2.5;
var_cmp3.real = var_cmp1.real + var_cmp2.real;
var_cmp3.imag= var_cmp1.imag + var_cmp2.imag;
```

Ключевое слово typedef можно использовать с любыми стандартными типами данных и типами объявленными ранее.

Контрольные вопросы и задания

1. Каким образом задаются массивы в языке C++?
2. Запишите массив целых чисел с начальными значениями 1, 2 и 3.
3. Сформулируйте идею алгоритма упорядочивания элементов массива по возрастанию (убыванию).
4. Как задаются строки в программе на C++?
5. Для чего предназначена функция strcpy() и в какой библиотеке она определена?
6. Запишите возможные способы начальной инициализации строки.
7. Какой управляющий символ соответствует концу строки?
8. Что выполняет функция strcmp()?
9. Какую роль играют структуры в программировании?
10. Что возвращает функция strlen()?
11. Запишите структуру для хранения имени, возраста и места работы сотрудника.
12. Как задаются переменные на структуры?
13. Чем объединения отличаются от структур?
14. Задайте объединение для хранения целых, вещественных чисел и символов.
15. Как задаются перечисления в языке C++?
16. Для чего предназначена функция sprintf()?

17.Создайте свой тип данных для представления беззнаковых целых (unsigned int) чисел.

18.Задайте структуру с битовыми полями для хранения шести свойств окна OS Windows.

19.Напишите программу для преобразования малых букв в строке в большие.

20.Опишите перечисления для оперирования константами TOP, BOTTOM, LEFT и RIGTH.

ГЛАВА ЧЕТВЕРТАЯ

УКАЗАТЕЛИ И ДИНАМИЧЕСКОЕ ПРЕДСТАВЛЕНИЕ ДАННЫХ

Гибкость языка C++ во многом достигается благодаря наличию механизма работы с памятью компьютера через указатели. Это позволяет создавать динамические переменные, массивы и более сложную организацию данных, которые существенно облегчают задачи программирования.

4.1. Указатели

При объявлении переменных, структур, объединений и т.п. операционная система выделяет необходимый объем памяти для хранения данных программы. Например, задавая целочисленную переменную

```
int a = 10;
```

в памяти ЭВМ выделяется либо 2, либо 4 байта (в зависимости от стандарта языка С), которые расположены друг за другом, начиная с определенного адреса. Здесь под адресом следует понимать номер байта в памяти, который показывает, где начинается область хранения той или иной переменной или каких-либо произвольных данных. Условно память ЭВМ можно представить в виде последовательности байт (рис. 4.1).

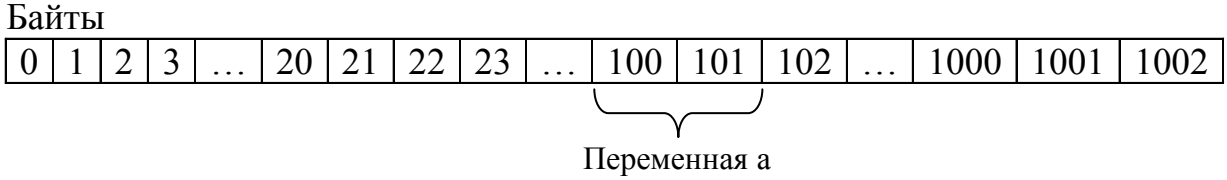

Рис. 4.1. Условное представление памяти ЭВМ с расположением переменной a

На рис. 4.1 переменная a расположена в 100 и 101 ячейках и занимает соответственно два байта. Адрес этой переменной равен 100. Учитывая, что значение переменной a равно 10, то в ячейке под номером 100 будет записано число 10, а в ячейке под номером 101 – ноль. Аналогичная картина остается справедливой и при объявлении произвольных переменных и структур, только в этом случае расходуется разный объем памяти в зависимости от типа переменной.

В языке С++ имеется механизм работы с переменными через их адрес. Для этого необходимо объявить указатель соответствующего типа. Указатель объявляется также как и переменная, но перед его именем ставится символ ‘*’:

```
int *ptr_a;
char *ptr_ch, *ptr_var;
```

Для того чтобы с помощью указателя ptr_a работать с переменной a он должен указывать на адрес этой переменной. Это значит, что значение указателя ptr_a должно быть равно адресу переменной a. Здесь возникает две задачи: во-первых, необходимо определить адрес переменной, и, во-вторых, присвоить этот адрес указателю. Для определения адреса в языке С++ используется символ ‘&’ как показано ниже:

```
unsigned long ptr = &a;
```

В результате переменной ptr будет присвоен адрес переменной a. Аналогичным образом можно выполнить инициализацию указателя ptr_a:

```
ptr_a = &a;      //инициализация указателя
```

По существу получается, что указатель это переменная, которая хранит адрес на заданную область памяти. Но в отличие от обычной переменной

позволяет еще, и работать с данной областью, т.е. записывать в нее значения и считывать их. Допустим, что переменная a содержит число 10, а указатель ptr_a указывает на эту переменную. Тогда для того чтобы считывать и записывать значения переменной a с помощью указателя ptr_a используется следующая конструкция языка C:

```
int b = *ptr_a; //считывание значения переменной a
*ptr_a = 20;    //запись числа 20 в переменную a
```

Здесь переменной b присваивается значение переменной a через указатель ptr_a, a, затем, переменной a присваивается значение 20. Таким образом, для записи и считывания значений с помощью указателя необходимо перед его именем ставить символ ‘*’ и использовать оператор присваивания.

Для каких задач программирования необходимо использовать указатели? Рассмотрим пример, представленный в листинге 4.1.

Листинг 4.1. Пример использования указателей.

```
void interchange(int* arg1, int* arg2);
int main()
{
 int arg1 = 10, arg2 = 20;
 interchange(&arg1,&arg2);
 return 0;
}
void interchange (int* arg1, int* arg2)
{
 int temp = *arg1;
 *arg1 = *arg2;
 *arg2 = temp;
}
```

Здесь реализована функция, которая в качестве аргументов принимает два указателя на переменные arg1 и arg2. Благодаря такому подходу значения локальных переменных, объявленных в функции main() можно менять через указатели в функции interchange(). Реализовать проще или также данную задачу с помощью обычных переменных (без использования указателей) невозможно. Учитывая, что в основе работы компьютера лежит работа с адресами той или иной области памяти, считывания от туда данных и записи новых, то указатели позволяют программисту добиться более эффективного исполнения программ.

Каждый раз при работе с указателями необходимо выполнять их инициализацию, т.е. задавать адрес на выделенную область памяти. Сложность работы с указателями заключается в том, что при их объявлении они указывают на произвольную область памяти, с которой можно работать как с обычной переменной. Приведем такой пример

```
int* ptr;
```

```
*ptr = 10;
```

В результате в произвольную область памяти будет записано два байта со значениями 10 и 0. Это может привести к необратимым последствиям в работе программы и к ее ошибочному завершению. Поэтому перед использованием указателей всегда нужно быть уверенным, что они предварительно были инициализированы.

Рассмотрим возможность использования указателей при работе с массивами. Допустим, что объявлен массив целочисленного типа int размерностью в 20 элементов:

```
int array[20];
```

Элементы массивов всегда располагаются друг за другом в памяти ЭВМ, начиная с первого, индекс которого равен 0 (рис. 4.2).

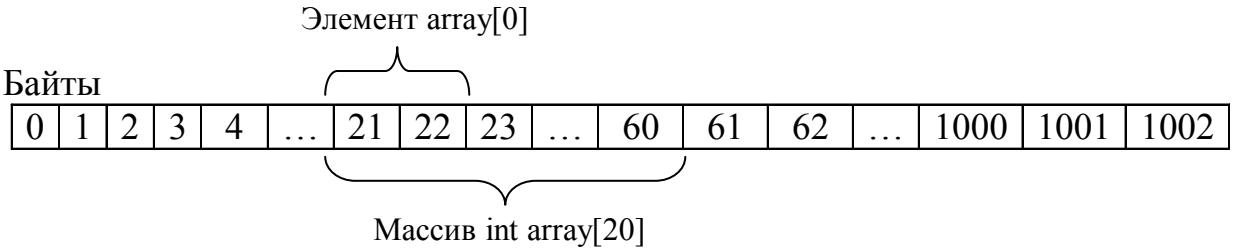

Рис. 4.2. Условное расположение массива int array[20] в памяти ЭВМ

Из рис. 4.2 видно, что для получения адреса массива array достаточно знать адрес его первого элемента array[0], который можно определить как адрес переменной и присвоить его указателю:

```
int* ptr_ar = &array[0];
```

Однако в языке C++ предусмотрена более простая конструкция для определения адреса массивов, которая записывается следующим образом:

```
int* ptr_ar = array;
```

т.е. имя массива задает его адрес. Следует отметить, что величины &array[0] и array являются константами, т.е. не могут менять своего значения. Это означает, что массив (как и переменная) не меняют своего адреса, пока существуют в зоне своей видимости.

Формально, имея указатель на массив, можно считывать и записывать в него значения элементов. Вначале, когда указатель указывает на первый элемент, его значение можно менять следующим образом:

```
int a = *ptr_ar;
*ptr_ar = 20;
```

Для того чтобы перейти к следующему элементу массива, достаточно выполнить операцию

```
ptr_ar += 1;
```

или

```
ptr_ar++;
```

Особенностью применения данной операции является то, что адрес, т.е. значение указателя ptr_ar изменится не на единицу, а на четыре, ровно на столько, сколько занимает один элемент в памяти ЭВМ, в данном случае четыре байта. В результате указатель ptr_ar будет указывать на следующий элемент массива, с которым можно работать также как и с предыдущим.

Пример изменения адреса указателя ptr_ar показывает необходимость правильно задавать его тип, в данном случае int. Если бы тип указателя ptr_ar был char, то при использовании оператора инкремента ++, его значение увеличилось бы на единицу, а не на четыре и переход к следующему элементу массива осуществлялся бы некорректно.

Приведем пример использования указателя на массив, показывающий особенности применения указателей.

Листинг 4.2. Работа с элементами массива через указатели.

```
#include <stdio.h>
#define SIZE 5
int main()
{
 int array[SIZE] = {10,20,30,40,50};
 int *ptr_ar = array;
 for(int i = 0;i < SIZE;i++)
      printf("Значение элемента %d, адрес элемента
%p\n",*(ptr_ar+i), ptr_ar+i);
 return 0;
}
```

В данном примере используется указатель ptr_ar, который всегда указывает на первый элемент массива array[0]. Для перехода к следующим элементам прибавляется значение i и выводится результат на экран.

В языке C++ при работе с массивами через указатель допускается более простая форма чем рассмотренная ранее. Допустим, что ptr_ar указывает на первый элемент массива array. Тогда для работы с элементами массива можно пользоваться следующей записью:

```
int *ptr_ar = array;
ptr_ar[0] = 10;
ptr_ar[1] = 20;
```

т.е. доступ к элементам массива осуществляется по его индексу.

Массивы удобно передавать функциям через указатели. Пусть имеется функция вычисления суммы элементов массива:

```
int sum(int* ar, int n);
```

и массив элементов

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

Тогда для передачи массива функции sum следует использовать такую запись:

```
int s = sum(array,5);
```

т.е. указатель ar инициализируется по имени массива array и будет указывать на его первый элемент.

Следует отметить, что все возможные изменения, выполненные с массивом внутри функции sum(), сохраняются в массиве array. Это свойство можно использовать для модификации элементов массива внутри функций. Например, рассмотренная ранее функция strcpy(char *dest, char* src), выполняет изменение массива, на который указывает указатель dest. Для того чтобы «защитить» массив от изменений следует использовать ключевое слово const либо при объявлении массива, либо в объявлении аргумента функции как показано ниже.

```
char* strcpy(char* dest, const char* src)
{
 while(*src != '\0') *dest++ = *src++;
 *dest = *src;
 return dest;
}
```

В этом объявлении указатель src не может вносить изменения в переданный массив при вызове данной функции, а может лишь передавать значения указателю dest. В приведенном примере следует обратить внимание на использование конструкции *dest++ и *src++. Дело в том, что приоритет операции ++ выше приоритета операции *, поэтому эти выражения аналогичны выражениям *(dest++) и *(src++). Таким образом, в строке

```
*dest++ = *src++;
```

сначала выполняется присваивание значений соответствующих элементов, на которые указывают dest и src, а затем происходит увеличение значений указателей для перехода к следующим элементам массивов. Благодаря такому подходу осуществляется копирование элементов одного массива в другой.

Последняя строка примера *dest = *src, присваивает символ конца строки ‘\0’ массиву dest.

В общем случае можно выполнять следующие операции над указателями:

pt1 = pt2; //Присвоение значения одного указателя другому
pt1 += *pt2; //Увеличение значения первого указателя на величину *pt2
pt1 -= *pt2; //Уменьшение адреса указателя на величину *pt2
pt1-pt2; //Вычитание значений адресов первого и второго указателей
pt1++; и ++pt1; //Увеличение адреса на единицу информации
pt1--; и --pt1; //Уменьшение адреса на единицу информации

Если указатели pt1 и pt2 имеют разные типы, то операция присваивания должна осуществляться с приведением типов, например:

```
int* pt1;
double* pt2;
pt1 = (int *)pt2;
```

Язык С++ допускает инициализацию указателя строкой, т.е. будет верна следующая запись:

```
char* str = “Лекция”;
```

Здесь задается массив символов, содержащих строку «Лекция» и адрес этого массива передается указателю str. Таким образом, получается, что есть массив, но нет его имени. Есть только указатель на его адрес. Подобный подход является удобным, когда необходимо задать массив строк. В этом случае возможна такая запись:

```
char* text[] = {«Язык С++ имеет»,
                «удобный механизм»,
                «для работы с памятью.»};
```

При таком подходе задается массив указателей, каждый из которых указывает на начало соответствующей строки. Например, значение *text[0] будет равно символу ‘Я’, значение *text[1] – символу ‘у’ и значение *text[2] – символу ‘д’. Особенность такого подхода состоит в том, что здесь задаются три отдельных одномерных массива символов никак не связанных друг с другом. Каждый массив – это отдельная строка. В результате не расходуется лишний объем памяти характерный для двумерного массива символов, который условно можно представить в виде таблицы (рис. 4.3).

```
char text_array[][] = {«Язык С++ имеет»,
                       «удобный механизм»,
                       «для работы с памятью.»};
```

Я	з	ы	к		С	+	+		и	м	е	е	т	\0							
у	д	о	б	н	ы	й		м	е	х	а	н	и	з	м	\0						
д	л	я		р	а	б	о	т	ы		с		п	а	м	я	т	ь	ю	.	\0

Не используемые символы

Рис. 4.3. Условное представление текста в двумерном массиве text_array

Благодаря большей гибкости представления информации, которую дают указатели их часто используют для хранения и обработки текстовой информации. Вместе с тем при работе с текстом через указатели всегда следует обращать внимание на какую область памяти они указывают. Например, следующий фрагмент программы показывает ошибочное использование указателя при вводе строки с клавиатуры.

```
char* name;
scanf("%s",name);
```

В результате выполнения данных операторов символы, введенные с клавиатуры, будут записаны в произвольную область памяти, что приведет к ошибочному завершению всей программы. Простым решением данной проблемы является резервирование памяти с помощью массива и передачи адреса в функции scanf() на него:

```
char buff[100];
char* name = buff;
scanf("%s",name);
```

Язык C++ также позволяет инициализировать указатели на структуры. Допусти, что имеется структура

```
struct tag_person {
 char name[100];
 int old;
} person;
```

и инициализируется следующий указатель:

```
struct tag_person* pt_man = &person;
```

В этом случае, для доступа к элементам структуры можно использовать следующий синтаксис:

```
(*pt_man).name;
pt_man->name;
```

Последний вариант показывает особенность использования указателя на структуры, в котором для доступа к элементу используется операция ->. Данная операция наиболее распространена по сравнению с первым вариантом и является предпочтительной.

Каждый раз при инициализации указателя использовался адрес той или иной переменной. Это было связано с тем, что компилятор языка C++ автоматически выделяет память для хранения переменных и с помощью указателя можно без последствий работать с этой выделенной областью. Вместе с тем существуют функции malloc() и free(), позволяющие выделять и освобождать память по мере необходимости. Данные функции находятся в библиотеке <stdlib.h> и имеют следующий синтаксис:

```
void* malloc(size_t);           //функция выделения памяти
void free(void* memblock);      //функция освобождения памяти
```

Здесь size_t – размер выделяемой области памяти в байтах; void* - обобщенный тип указателя, т.е. не привязанный к какому-либо конкретному типу. Рассмотрим работу данных функций на примере выделения памяти для 10 элементов типа double.

Листинг 4.3. Программирование динамического массива.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
 double* ptd;
 ptd = (double *)malloc(10 * sizeof(double));
 if(ptd != NULL)
 {
      for(int i = 0;i < 10;i++)
           ptd[i] = i;
 } else printf("Не удалось выделить память.");
 free(ptd);
 return 0;
}
```

При вызове функции malloc() выполняется расчет необходимой области памяти для хранения 10 элементов типа double. Для этого используется функция sizeof(), которая возвращает число байт, необходимых для хранения одного элемента типа double. Затем ее значение умножается на 10 и в результате получается объем для 10 элементов типа double. В случаях, когда по каким-либо причинам не удается выделить указанный объем памяти, функция malloc() возвращает значение NULL. Данная константа определена в нескольких библиотеках, в том числе в <stdio.h> и <stdlib.h>. Если функция malloc() возвратила указатель на выделенную область памяти, т.е. не равный NULL, то выполняется цикл, где записываются значения для каждого

элемента. При выходе из программы вызывается функция free(), которая освобождает ранее выделенную память. Формально, программа написанная на языке C++ при завершении сама автоматически освобождает всю ранее выделенную память и функция free(), в данном случае, может быть опущена. Однако при составлении более сложных программ часто приходится много раз выделять и освобождать память. В этом случае функция free() играет большую роль, т.к. не освобожденная память не может быть повторно использована, что в результате приведет к неоправданным затратам ресурсов ЭВМ.

Использование указателей досталось в "наследство" от языка C. Чтобы упростить процесс изменения параметров в C++ вводится такое понятие как ссылка. Ссылка представляет собой псевдоним (или второе имя), который программы могут использовать для обращения к переменной. Для объявления ссылки в программе используется знак & перед ее именем. Особенность использования ссылок заключается в необходимости их инициализации сразу же при объявлении, например:

```
int var;
int &var2 = var;
```

Здесь объявлена ссылка с именем var2, которая инициализируется переменной var. Это значит, что переменная var имеет свой псевдоним var2, через который возможно любое изменение значений переменной var. Преимущество использования ссылок перед указателями заключается в их обязательной инициализации, поэтому программист всегда уверен, что переменная var2 работает с выделенной областью памяти, а не с произвольной, что возможно при использовании указателей. В отличие от указателей ссылка инициализируется только один раз, при ее объявлении. Повторная инициализация приведет к ошибке на стадии компиляции. Благодаря этому обеспечивается надежность использования ссылок, но снижает гибкость их применения. Обычно ссылки используют в качестве аргументов функций для изменения передаваемых переменных внутри функций. Следующий пример демонстрирует применение такой функции:

Листинг 4.4. Пример использования ссылок.

```
void swap(int& a, int& b)
{
 int temp = a;
 a = b;
 b = temp;
}
int main()
{
 int agr1 = 10, arg2 = 5;
 swap(arg1, arg2);
 return 0;
```

```
}
```

В данном примере функция swap() использует два аргумента, представляющие собой ссылки на две переменные. Используя имена ссылок a и b, осуществляется манипулирование переменными arg1 и arg2, заданных в основной функции main() и переданных как параметры функции swap(). Преимущество функции swap() (которая использует ссылки, а не указатели на переменные) заключается гарантии того, что функция в качестве аргументов будет принимать соответствующие типы переменные, а не какую-либо другую информацию, и ссылки будут инициализированы корректно перед их использованием. Это отслеживается компилятором в момент преобразования текста программы в объектный код и выдается сообщение об ошибке, если использование ссылок неверно. В отличие от указателей со ссылками нельзя выполнять следующие операции:

- нельзя получить адрес ссылки, используя оператор адреса C++;
- нельзя присвоить ссылке указатель;
- нельзя сравнить значения ссылок, используя операторы сравнения C++;
- нельзя выполнять арифметические операции над ссылкой, например, добавить смещение;

4.2. Стек

Применение указателей позволяет создавать различные динамические структуры для хранения данных. Наиболее простой из них является стек, представляющий собой последовательность объектов данных, связанных между собой с помощью указателей. Особенность организации структуры стека показана на рис. 4.4.

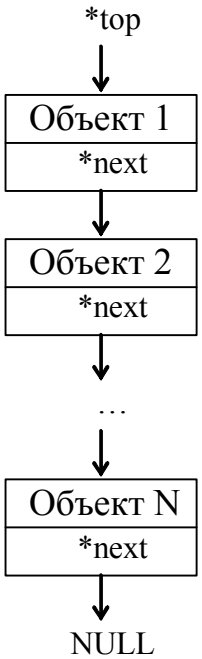

Рис. 4.4. Организация структуры стека

Из рис. 4.4 видно, что каждый объект стека связан с последующим с помощью указателя next. Если указатель next равен NULL, то достигнут конец стека. Особенность работы со стеком заключается в том, что новый объект всегда добавляется в начало списка. Удаление также возможно только для первого элемента. Таким образом, данная структура реализует очередь по принципу «первым вошел, последним вышел». Такой принцип характерен при вызове функций в рекурсии, когда адрес каждой последующей функции записывается в стеке для корректной передачи управления при ее завершении.

Для описания объектов составляющих стек можно воспользоваться структурами, например, следующее определение задает шаблон для описания объекта данных стека:

```
typedef struct tag_obj {
 char name[100];
                      struct tag_obj* next;
                    } OBJ;
```

Здесь поле name – символическое имя объекта, а next – указатель на следующий объект. Зададим указатель top как глобальную переменную со значением равным NULL:

```
OBJ* top = NULL;
```

и введем функцию для добавления нового объекта в стек:

```
void push(char* name)
{
 OBJ* ptr = (OBJ *)malloc(sizeof(OBJ));
 strcpy(ptr->name,name);
 ptr->next = top;
 top = ptr;
}
```

Данная функция в качестве аргумента принимает указатель на строку символов, которые составляют имя добавляемого объекта. Внутри функции инициализируется указатель ptr на новый созданный объект. В поле name записывается переданная строка, а указатель next инициализируется на первый объект. Таким образом, добавленный объект ставится на вершину списка.

Для извлечения объекта из стека реализуется следующая функция:

```
void pop()
{
 if(top != NULL)
 {
      OBJ* ptr = top->next;
      printf("%s - deleted\n",top->name);
      free(top);
      top = ptr;
 }
}
```

В данной функции сначала выполняется проверка указателя top. Если он не равен значению NULL, то в стеке имеются объекты и самый верхний из них следует удалить. Перед удалением инициализируется указатель ptr на следующий объект для того, чтобы он был доступен после удаления верхнего. Затем вызывается функция printf(), которая выводит на экран сообщение об имени удаленного объекта. Наконец, вызывается функция free() для удаления самого верхнего объекта, а указатель top инициализируется на следующий объект.

Для анализа работы данных функций введем еще одну вспомогательную функцию, которая будет выводить имена объектов, находящихся в стеке.

```
void show_stack()
{
 OBJ* ptr = top;
 while(ptr != NULL)
 {
      printf("%s\n",ptr->name);
      ptr = ptr->next;
 }
}
```

Работа данной функции реализуется с помощью цикла while, который работает до тех пор, пока указатель ptr не достигнет конца стека, т.е. пока не будет равен значению NULL. Внутри цикла вызывается функция printf() для вывода имени текущего объекта на экран и указатель ptr перемещается на следующий объект.

Рассмотрим применение данных функций в функции main():

```
int main()
{
 char str[100];
 for(int i = 0;i < 5;i++) {
      sprintf(str,"Object %d",i+1);
      push(str);
 }
 show_stack();
 while(top != NULL) pop();
 return 0;
}
```

Здесь создается стек, состоящий из 5 объектов, с помощью оператора цикла for. Внутри цикла инициализируется переменная str с именем объекта, которая, затем, передается в качестве аргумента функции push(). После вызова функции show_stack() на экране появляются следующие строки:

Object 5
Object 4
Object 3
Object 2
Object 1

Полученные результаты показывают, что последний 5-й добавленный объект находится на вершине стека, а первый – в конце. При вызове функции pop() в цикле while() осуществляется удаление элементов стека из памяти ЭВМ. В результате на экране появляются строки:

Object 5 - deleted
Object 4 - deleted
Object 3 - deleted
Object 2 - deleted
Object 1 – deleted

Таким образом, функция pop() удаляет верхние объекты стека с 5-го по 1-й.

4.3. Связные списки

Рассмотренные ранее типы данных и работа с ними позволяют писать программы разной степени сложности. Однако существуют задачи, в которых традиционное представление информации на основе переменных, структур, объединений и т.п. является не эффективным. Классическим примером такого рода может стать обработка табличных данных, у которых есть заданные поля, т.е. набор стандартных типов данных, и записи, представляющие собой конкретное наполнение таблицы. Формально для описания таблицы можно использовать простой или динамический массив структур. Но в этом случае возникает несколько проблем. Во-первых, наперед часто сложно указать приемлемое число записей (размер массива) для хранения информации. Во-вторых, при большом размере массива сложно выполнять добавление и удаление записей, находящихся между другими записями таблицы. И, наконец, любой используемый массив не будет эффективно использовать память ЭВМ, т.к. всегда будут зарезервированы не используемые записи на случай добавления новых. Эти основные проблемы обусловили необходимость создания нового механизма представления данных в памяти ЭВМ, который получил название связные списки.

Идея связных списков состоит в представлении данных в виде объектов, связанных друг с другом указателями (рис. 4.5).

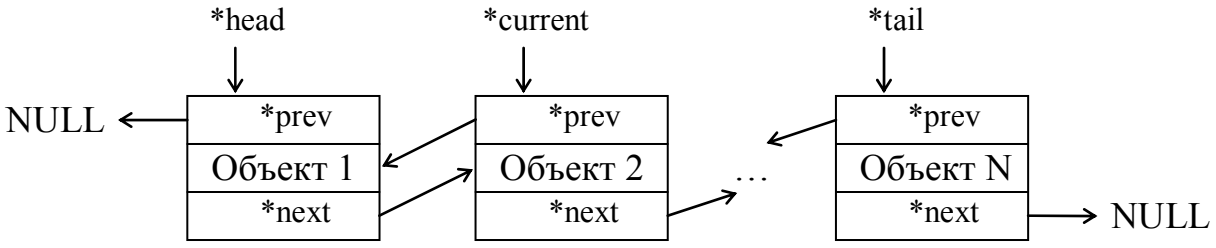

Рис. 4.5. Графическая интерпретация связных списков

Здесь *prev и *next – указатели на предыдущий и следующий объекты соответственно; *head и *tail – указатели на первый и последний объекты; *current – указатель на текущий объект, с которым идет работа. Если предыдущего или последующего объекта не существует, то указатели *prev и *next принимают значение NULL. Указатели *head и *tail являются вспомогательными и служат для быстрого перемещения к первому и

последнему объекту соответственно. Рассмотрим работу связных списков на примере представления следующей информации.

	название	автор	год издания
lib	lib.title	lib.author	lib.year
lib	lib.title	lib.author	lib.year
lib	lib.title	lib.author	lib.year
⋮			
lib	lib.title	lib.author	lib.year

Строки данной таблицы можно описать с помощью структуры:

```
typedef struct tag_lib {
 char title[100];
 char author[100];
 int year;
} LIB;
```

Каждая строка хранится в памяти независимо от местоположения предыдущих и последующих строк, но имеет указатель на предыдущую и последующую строки таблицы. Благодаря этому обеспечивается единство таблицы. Таким образом, необходимо ввести еще одну структуру, которая будет описывать связи между строками таблицы, и представлять собой объект данных:

```
typedef struct tag_obj {
 LIB lib;
 LIB* prev, *next;
} OBJ;
```

Здесь *prev и *next – указатели на предыдущую и следующую строки соответственно.

По умолчанию указатели head и tail равны NULL:

```
OBJ* head = NULL, *tail = NULL;
```

При добавлении записей выполняется инициализация этих указателей, а также prev и next внутри объектов:

```
OBJ* add_obj(char* title, char* author, int year)
{
 OBJ* current = (OBJ *)malloc(sizeof(OBJ));
 strcpy(current->lib.title, title);
 strcpy(current->lib.author, author);
 current->lib.year = year;
```

```
 current->prev = tail;
 current->next = NULL;
 if(tail != NULL) tail->next = current;
 if(head == NULL) head = current;
 tail = current;
 return current;
}
```

Данная функция использует три параметра для ввода данных в структуру LIB. В первой строке функции создается новая структура типа OBJ. Во второй, третьей и четвертой строках осуществляется запись информации в структуру LIB. Затем, инициализируются указатели prev и next добавленного объекта. Учитывая, что добавление осуществляется в конец списка, то указатель next должен быть равен NULL, а указатель prev указывать на предыдущий объект, т.е. быть равен указателю tail. В свою очередь, объект, на который указывает указатель tail, становится предпоследним и его указатель next должен указывать на последний объект, т.е. быть равным указателю current. Затем проверяется, является ли добавляемый объект первым (head == NULL), и если это так, то указатель head приравнивается указателю current. Наконец, указатель tail инициализируется на последний объект. Последняя строка функции возвращает указатель на созданный объект.

Для полноценной работы связного списка необходимо ввести функцию удаления элемента, которая может быть записана следующим образом:

```
OBJ* del_obj(OBJ* current)
{
 if(current == head)
      if(current->prev != NULL) head = current->prev;
      else head = current->next;
 if(current == tail)
      if(current->next != NULL) tail = current->next;
      else tail = current->prev;
 if(current->prev != NULL)
      current->prev->next = current->next;
 if(current->next != NULL)
      current->next->prev = current->prev;
 free(current);
 return head;
}
```

Функция del_obj() в качестве аргумента использует указатель на объект, который следует удалить. Сначала выполняется проверка для инициализации указателя head, в том случае, если удаляется первый объект, на который он указывает. Аналогичная проверка осуществляется для tail. Затем осуществляется проверка: если предыдущий объект относительно текущего существует, то его указатель на следующий объект следует переместить. Аналогичная проверка выполняется и для следующего объекта относительно

удаляемого. После настройки всех указателей вызывается функция free() для удаления объекта из памяти и возвращается указатель на первый объект.

Введенные функции в программе можно использовать следующим образом:

```
int main()
{
 OBJ *current = NULL;
 int year;
 char title[100], author[100];
 do
 {
      printf("Введите название книги: ");
      scanf("%s",title);
      printf("Введите автора: ");
      scanf("%s",author);
      printf("Введите год издания: ");
      scanf("%d",&year);
      current = add_obj(title,author,year);
      printf("Для выхода введите 'q'");
 } while(scanf("%d",&year) == 1);
 current = head;
 while(current != NULL)
 {
      printf("Title: %s, author %s, year = %d\n",
current->lib.title, current->author.old, current->lib.year);
      current = current->next;
 }
 while(head != NULL)
      del_obj(head);
 return 0;
}
```

Функция main() осуществляет ввод названия книги, автора и года издания в цикле do while(). Там же вызывается функция add_obj() с соответствующими параметрами, которая формирует связанный список на основе введенных данных. Пользователь выполняет ввод до тех пор, пока не введет какой либо символ на последний запрос. В результате цикл завершится, и указатель current передвигается на первый объект. Затем, в цикле while осуществляется вывод информации текущего объекта на экран, а указатель current передвигается на следующий объект. Данная процедура выполняется до тех пор, пока указатель не станет равным NULL, что означает достижение конца списка. Перед выходом из программы связный список удаляется с помощью функции del_obj(), у которой в качестве аргумента всегда используется указатель head. Если данный указатель принимает значение NULL, то это означает, что связный список пуст и не содержит ни одного объекта. После этого программа завершает свою работу.

4.4. Бинарные деревья

Связные списки не охватывают весь спектр возможных представлений данных. Например, с их помощью сложно описать иерархические структуры подобные каталогам и файлам или хранения информации генеалогического древа. Для этого лучше подходит модель известная как бинарные деревья. Графически бинарные деревья можно изобразить как последовательность объектов, каждый из которых может быть связан с двумя последующими (рис. 4.6).

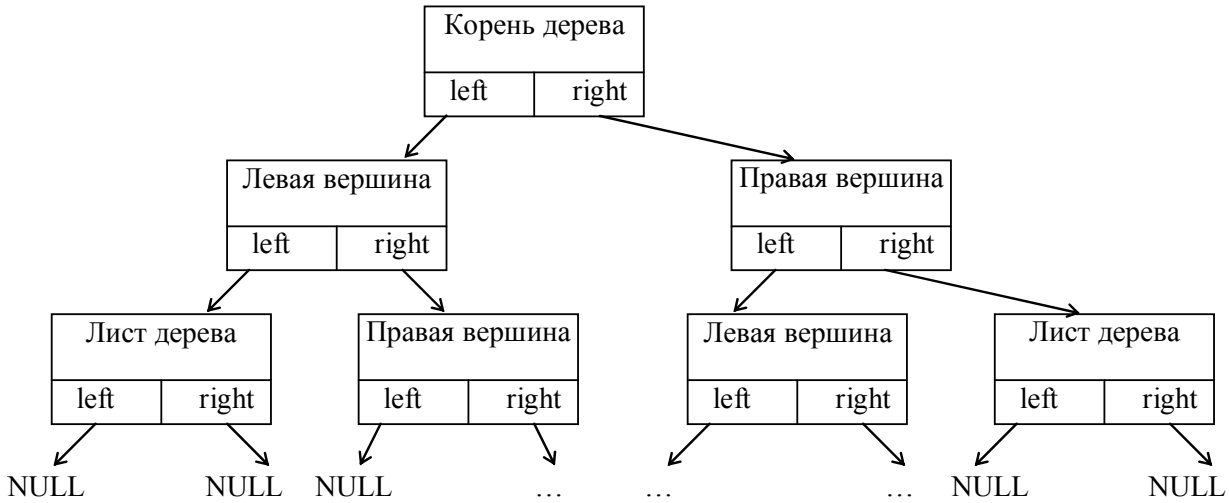

Рис. 4.6. Графическая интерпретация бинарного дерева

Каждый объект бинарного дерева имеет два указателя: на «левый» и «правый» вершины. Самый верхний уровень называется корнем дерева. Если указатели объекта left и right равны NULL, то он называет листом дерева.

При описании структуры каталогов с помощью бинарного дерева можно воспользоваться следующим правилом. Переход по «левым» указателям будет означать список файлов, а переход по «правым» – список каталогов. Например, для описания простой структуры (рис. 4.7), бинарное дерево будет иметь вид, представленный на рис. 4.8.

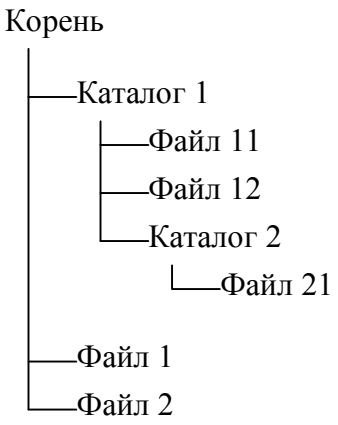

Рис. 4.7. Структура каталогов

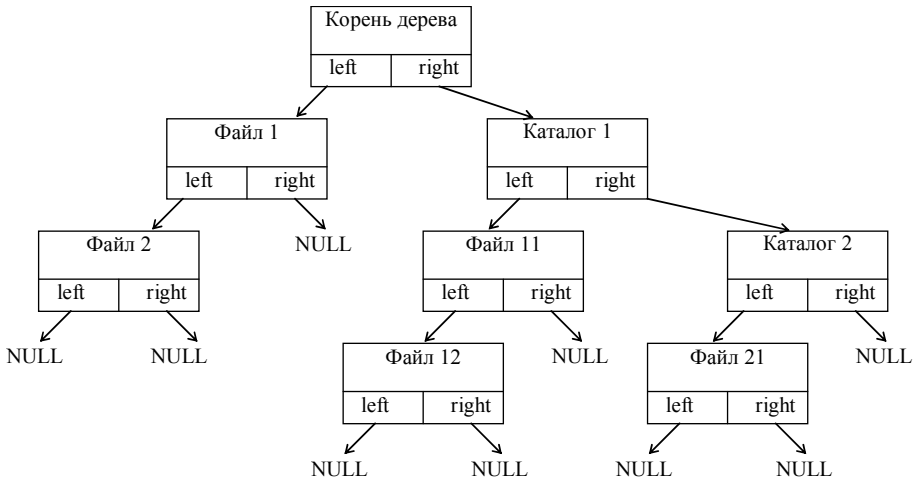

Рис. 4.8. Структура бинарного дерева

Объекты, из которых состоит бинарное дерево удобно представить в виде структур. Также как и в связных списках, первая структура будет описывать данные, хранящиеся в вершинах дерева, а вторая представлять связи между вершинами.

```
typedef struct tag_data {
 char name[100];
} DATA;

typedef struct tag_tree {
 DATA data;
 struct tag_tree* left, *right;
} TREE;
TREE* root = NULL;
```

Для формирования дерева введем функцию add_node(), которая в качестве аргументов будет принимать указатель на вершину дерева, к которому добавляются новые вершины, имя вершины и тип вершины: левая или правая. Кроме того, данная функция будет возвращать указатель на новую созданную вершину.

```
TREE* add_node(TREE* node,char* name, TYPE type = LEFT)
{
 TREE* new_node = (TREE *)malloc(sizeof(TREE));
 if(type == LEFT && node != NULL) node->left = new_node;
 else if(node != NULL) node->right = new_node;
 strcpy(new_node->data.name,name);
 new_node->left = NULL;
 new_node->right = NULL;
 return new_node;
}
```

Последний аргумент функции имеет тип TYPE, который удобно определить как перечисляемый тип:

```
typedef enum tag_type {RIGHT, LEFT} TYPE;
```

и определить параметр по умолчанию LEFT.

Для отображения дерева целесообразно воспользоваться рекурсивными функциями show_next(), которые вызываются из функции show_tree() следующим образом:

```
void show_next(TREE* node,int off)
{
 if(node != NULL)
 {
      for(int i=0;i < off;i++) putchar(' ');
```

```
        printf("%s\n",node->data.name);
        show_next(node->left,off);
        show_next(node->right,off+1);
  }
}
void show_tree()
{
 if(root != NULL)
 {
        printf("%s\n",root->data.name);
        show_next(root->left,0);
        show_next(root->right,1);
 }
}
```

Первый параметр функции show_next() служит для перемещения к следующим вершинам дерева, а второй – для смещения при отображении строк на экране монитора, принадлежащих разным вершинам. Рекурсия функций show_next() выполняется до тех пор, пока указатель на левую или правую вершину не станет равным NULL. В этом случае работа функции завершается, и управление передается предыдущей вызываемой функции. Таким образом, происходит отображение всего бинарного дерева.

При завершении программы необходимо удалить созданные объекты дерева. Для этого также удобно воспользоваться рекурсивными функциями. По аналогии введем рекурсивную функцию del_next(), и основную del_tree(), из которой вызывается функция del_next(). Реализация этих функций приведена ниже:

```
void del_next(TREE* node)
{
 if(node != NULL)
 {
        del_next(node->left);
        del_next(node->right);
        printf("node %s - deleted\n",node->data.name);
        free(node);
 }
}
void del_tree()
{
 if(root != NULL)
 {
        del_next(root->left);
        del_next(root->right);
        printf("node %s - deleted\n",root->data.name);
        free(root);
 }
}
```

Аргумент функции del_next() используется для перехода к следующим вершинам дерева. В самой функции выполняется проверка: если следующая вершина существует, т.е. указатель не равен NULL, то выполняется просмотр дерева сначала по левой вершине, а затем по правой. При достижении листьев дерева функции del_next() вызываются с аргументом NULL и не выполняют никаких действий, поэтому программа переходит к функции printf(), которая вывод на экран сообщение об имени удаляемой вершины, а затем вызывается функция free() для освобождения памяти, занятой данным объектом. После этого осуществляется переход к предыдущей функции del_next() и описанный процесс повторяется до тех пор, пока не будут удалены все объекты кроме корневого. Корень дерева удаляется непосредственно в функции del_tree(), после чего можно говорить об удалении всего дерева.

Использование описанных функций в функции main() реализуются следующим образом:

```
int main()
{
 root = add_node(NULL,"Root");
 TREE* current = add_node(root,"File 1",LEFT);
 current = add_node(current,"File 2",LEFT);
 current = add_node(root,"Folder 1",RIGHT);
 current = add_node(current,"File 11",LEFT);
 current = add_node(current,"File 12",LEFT);
  current = add_node(root->right,"Folder 2",RIGHT);
 current = add_node(current,"File 21",LEFT);
 show_tree();
 del_tree();
 root = NULL;
 return 0;
}
```

Данная функция сначала инициализирует глобальный указатель root как корень дерева и в качестве первого аргумента функции add_node() записывает NULL, что означает, что предыдущего объекта не существует. Затем вводится вспомогательный указатель current, с помощью которого выполняется создание двоичного дерева, описывающее файловую структуру (рис. 4.7). После создания дерева вызывается функция show_tree() для его отображения на экран, затем оно удаляется и программа завершает свою работу.

Контрольные вопросы и задания

1. Для чего предназначены и как задаются указатели в языке C++?
2. Что такое адрес переменной?
3. Объявите целочисленную переменную и проинициализируйте на нее указатель.

4. Чему будет равно значение указателя int* ptr = 0; после выполнения операции ptr++?

5. Каким образом можно задавать указатель на массив?

6. Для чего предназначена функция malloc()?

7. Запишите программу копирования одной строки в другую с помощью указателей на эти строки.

8. Что делает функция free() и в какой библиотеке она определена?

9. Какие операции с указателями допустимы?

10. Опишите структуру стека.

11. Объясните принцип работы функции вывода на экран элементов стека.

12. Дайте понятие связного списка.

13. Какие удобства хранения информации представляет связный список по сравнению с массивом.

14. Объясните работу функцию удаления элементов связного списка.

15. Как в программе описывается объект связного списка?

16. Опишите структуру бинарного дерева.

17. Какой тип информации удобно представлять с помощью бинарных деревьев?

18. Объясните принцип работы рекуррентных функций для отображения и удаления элементов бинарного дерева.

ГЛАВА ПЯТАЯ

ОСНОВЫ РАБОТЫ С ФАЙЛАМИ

До сих пор полагалось, что все данные хранятся непосредственно в памяти компьютера. Однако эти данные могут существовать только в момент выполнения программы и теряются при ее завершении. Если существует необходимость в долгом хранении какой-либо информации, например, текста введенного с клавиатуры, то ее целесообразно помещать в файл, который затем можно загрузить и восстановить последние данные. Файл можно представлять как специализированное хранилище данных, который располагается либо на жестком диске, либо на дискетке, либо в Flash-памяти, на CD, DVD и др. носителях, в которых информация может сохраняться длительное время. Например, текст программы, написанный на языке Visual C++, сохраняется в

файле с расширением cpp, который, как правило, хранится на жестком диске. Программа может создавать свои файлы и записывать в них самую разнообразную информацию, которая затем может быть считана.

5.1. Работа с текстовыми файлами

Для работы с файлами в языке C++ имеется набор функций, определенных в библиотеке stdio.h. Перед началом работы с файлом его следует открыть, что достигается с помощью функции fopen(), имеющей следующий синтаксис:

```
FILE *fopen( const char *filename, const char *mode );
```

Здесь filename – строка, содержащая путь и имя файла; mode – строка, определяющая режим открытия файла: на чтение или на запись; FILE – специальный тип данных для работы с файлами. Данная функция возвращает значение NULL, если файл не был успешно открыт, иначе – другое значение. Рассмотрим последовательность действий по созданию простого текстового файла на языке C++ и записи в него текстовой информации.

Листинг 5.1. Запись текстовой информации в файл.

```
#include <stdio.h>
int main()
{
 char str_file[]="Строка для файла";
 FILE* fp = fopen("my_file.txt","w");
 if(fp != NULL)
 {
      printf("Идет запись информации в файл…\n");
      for(int i=0;i < strlen(str_file);i++)
           putc(str_file[i],fp);
 }
 else printf("Невозможно открыть файл на запись.\n");
 fclose(fp);
 return 0;
}
```

В данном примере задается специализированный указатель fp типа FILE, который инициализируется функцией fopen(). Функция fopen() в качестве первого аргумента принимает строку, в которой задан путь и имя файла. Вторым параметром определяется способ обработки файла, в данном случае, значение “w”, которое означает открытие файла на запись с удалением всей прежней информации из него. Если файл открыт успешно, то указатель fp не будет равен NULL и с ним возможна работа. В этом случае с помощью функции putc() выполняется запись символов в файл, на который указывает указатель fp. Перед завершением программы открытый файл следует закрыть во избежание в нем потери данных. Это достигается функцией fclose(), которая

принимает указатель на файл и возвращает значение 0 при успешном закрытии файла, иначе значение EOF.

Рассмотрим теперь пример программы считывания информации из файла.

Листинг 5.2. Считывание текстовой информации из файла.

```
#include <stdio.h>
int main()
{
 char str_file[100];
 FILE* fp = fopen("my_file.txt","r");
 if(fp != NULL)
 {
      int i=0;
      char ch;
      while((ch = getc(fp)) != EOF)
           str_file[i++]=ch;
      str_file[i] = '\0';
      printf(str_file);
 }
 else printf("Невозможно открыть файл на чтение.\n");
 fclose(fp);
 return 0;
}
```

В приведенном листинге функция fopen() открывает файл на чтение, что определяется значением второго аргумента равного «r». Это значит, что в него невозможно произвести запись данных, а только считывание. Сначала выполняется цикл while, в котором из файла считывается символ с помощью функции getc() и выполняется проверка: если считанное значение не равно символу конца файла EOF, то значение переменной ch записывается в массив str_file. Данный цикл будет выполняться до тех пор, пока не будут считаны все символы из файла, т.е. пока не будет достигнут символ EOF. После завершения цикла формируется строка str_file, которая выводится на экран с помощью функции printf(). Перед завершением программы также выполняется функция закрытия файла fclose().

Работа с текстовыми файлами через функции putc и getc не всегда удобна. Например, если необходимо записать или считать строку целиком, то желательно иметь функции, выполняющие эту работу. В качестве таковых можно воспользоваться функциями fputs() и fgets() для работы со строками. Перепишем предыдущие примеры с использованием данных функций.

Листинг 5.3. Использование функций fpust() и fgets().

```
#include <stdio.h>
int main()
{
 char str_file[]="Строка для файла";
```

```
 FILE* fp = fopen(“my_file.txt”,”w”);
 if(fp != NULL) fputs(str_file,fp);
 else printf(“Невозможно открыть файл на запись.\n”);
 fclose(fp);

 fp =      fopen(“my_file.txt”,”r”);
 if(fp != NULL)
 {
      fgets(str_file,sizeof(str_file),fp);
      printf(str_file);
 }
 fclose(fp);

 return 0;
}
```

Аналогичные действия по записи данных в файл и считывания информации из него можно выполнить и с помощью функций fprintf() и fscanf(). Однако эти функции предоставляют большую гибкость в обработке данных файла. Продемонстрируем это на следующем примере. Допустим, имеется структура, хранящая информацию о книге: название, автор, год издания. Необходимо написать программу сохранения этой информации в текстовый файл и их считывания. Пример использования данных функций представлен в листинге 5.4.

Листинг 5.4. Использование функций fprintf() и fscanf().

```
#include <stdio.h>
#define N 2
struct tag_book
{
 char name[100];
 char author[100];
 int year;
} books[N];

int main(void)
{
 for(int i=0;i < N;i++)
 {
      scanf("%s",books[i].name);
      scanf("%s",books[i].author);
      scanf("%d",&books[i].year);
 }

 for(i=0;i < N;i++)
 {
      puts(books[i].name);
      puts(books[i].author);
      printf("%d\n",books[i].year);
```

```
 }

 FILE* fp = fopen("my_file.txt","w");
 for(i=0;i < N;i++)
      fprintf(fp,"%s %s %d\n",books[i].name,books[i].author,
books[i].year);
 fclose(fp);
 fp = fopen("my_file.txt","r");
 for(i=0;i < N;i++)
      fscanf(fp,"%s %s %d\n",books[i].name,books[i].author,
&books[i].year);
 fclose(fp);
 printf("-------------------------------------------------\n");
 for(i=0;i < N;i++)
 {
      puts(books[i].name);
      puts(books[i].author);
      printf("%d\n",books[i].year);
 }
    return 0;
}
```

При выполнении данной программы вводится информация по книгам в массив структур books и выводится введенная информация на экран. Затем открывается файл my_file.txt на запись, в который заносится информация по книгам в порядке: наименование, автор, год издания. Так как число книг в данном случае равно двум, то выходной файл будет выглядеть следующим образом:

Onegin Pushkin 1983
Oblomov Griboedov 1985

Затем, файл my_file.txt открывается на чтение и с помощью функции scanf() осуществляется считывание информации в элементы структуры. В заключении считанная информация выводится на экран монитора.

Представленный пример показывает возможность структурированной записи информации в файл и ее считывания. Это позволяет относительно просто сохранять разнородные данные в файле для их дальнейшего использования в программах.

При внимательном рассмотрении предыдущих примеров можно заметить, что функции считывания информации из файла «знают» с какой позиции следует считывать очередную порцию данных. Действительно, в последнем примере функция fscanf(), вызываемая в цикле, «знает» что нужно считать сначала первую строку из файла, затем вторую и т.д. И программисту нет необходимости задавать позицию для считывания данных. Все происходит автоматически. Вследствие чего появляется такая особенность работы? Дело в том, что у любого открытого файла в программе написанной на C++ имеется

указатель позиции (номера), с которой осуществляется считывание данных из файла. При открывании файла на чтение номер этой позиции указывает на начало файла. Поэтому функция fscanf(), вызванная первый раз, считывает данные первой строки. По мере считывания информации из файла, позиция сдвигается на число считанных символов. И функция fscanf() вызванная второй раз будет работать уже со второй строкой в файле. Несмотря на то, что указатель позиции в файле перемещается автоматически, в языке C++ имеются функции fseek() и ftell(), позволяющие программно управлять положением позиции в файле. Синтаксис данных функций следующий:

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

где *stream – указатель на файл; offset – смещение позиции в файле (в байтах); origin – флаг начального отсчета, который может принимать значения: SEEK_END – конец файла, SEEK_SET – начало файла; SEEK_CUR – текущая позиция. Последняя функция возвращает номер текущей позиции в файле.

Рассмотрим действие данных функций на примере считывания символов из файла в обратном порядке.

Листинг 5.5. Использование функций fseek() и ftell().

```
#include <stdio.h>
int main(void)
{
 FILE* fp = fopen("my_file.txt","w");
 if(fp != NULL)
 {
      fprintf(fp,"It is an example using fseek and ftell
functions.");
 }
 fclose(fp);
 fp = fopen("my_file.txt","r");
 if(fp != NULL)
 {
      char ch;
      fseek(fp,0L,SEEK_END);
      long length = ftell(fp);
      printf("length = %ld\n",length);
      for(int i = 1;i <= length;i++)
      {
           fseek(fp,-i,SEEK_END);
           ch = getc(fp);
           putchar(ch);
      }
 }
 fclose(fp);
 return 0;
}
```

В данном примере сначала создается файл, в который записывается строка “It is an example using fseek and ftell functions.”. Затем этот файл открывается на чтение и с помощью функции fseek(fp,0L,SEEK_END) указатель позиции помещается в конец файла. Это достигается за счет установки флага SEEK_END, который перемещает позицию в конец файла при нулевом смещении. В результате функция ftell(fp) возвратит число символов в открытом файле. В цикле функция fseek(fp,-i,SEEK_END) смещает указатель позиции на –i символов относительно конца файла, после чего считывается символ функцией getc(), стоящий на i-й позиции с конца. Так как переменная i пробегает значения от 1 до length, то на экран будут выведены символы из файла в обратном порядке.

5.2. Работа с бинарными файлами

Следует отметить, что во всех рассмотренных выше примерах функция fopen() в режимах “r” и “w” открывает текстовый файл на чтение и запись соответственно. Это означает, что некоторые символы форматирования текста, например возврат каретки ‘\r’ не могут быть считаны как отдельные символы, их как бы не существует в файле, но при этом они там есть. Это особенность текстового режима файла. Для более «тонкой» работы с содержимом файлов существует бинарный режим, который представляет содержимое файла как последовательность байтов где все возможные управляющие коды являются просто числами. Именно в этом режиме возможно удаление или добавление управляющих символов недоступных в текстовом режиме. Для того чтобы открыть файл в бинарном режиме используется также функция fopen() с последним параметром равным “rb” и “wb” соответственно для чтения и записи. Продемонстрируем особенности обработки бинарного файла на примере подсчета числа управляющих символов возврата каретки ‘\r’ в файле, открытый в текстовом режиме и бинарном.

Листинг 5.6. Программа подсчета числа символов ‘\r’ в файле.

```
#include <stdio.h>
int main(void)
{
 FILE* fp = fopen("my_file.txt","w");
 if(fp != NULL)
 {
      fprintf(fp,"It is\nan example using\nan binary file.");
 }
 fclose(fp);
 char ch;
 int cnt = 0;
 fp = fopen("my_file.txt","r");
 if(fp != NULL)
```

```
    {
        while((ch = getc(fp)) != EOF)
            if(ch == '\r') cnt++;
    }
    fclose(fp);
    printf("Text file: cnt = %d\n",cnt);
    cnt=0;
    fp = fopen("my_file.txt","rb");
    if(fp != NULL)
    {
        while((ch = getc(fp)) != EOF)
            if(ch == '\r') cnt++;
    }
    fclose(fp);
    printf("Binary file: cnt = %d\n",cnt);
    return 0;
}
```

Результат работы будет следующий:

Text file: cnt = 0
Binary file: cnt = 2

Анализ полученных данных показывает, что при открытии файла в текстовом режиме, символы возврата каретки ‘\r’ не считываются функцией getc(), а в бинарном режиме доступны все символы.

Еще одной особенностью текстового формата файла является запись чисел в виде текста. Действительно, когда в предыдущих примерах выполнялась запись числа в файл с помощью функции fprintf(), например, года издательства книги, то число заменялось строкой. А когда она считывалась функцией fscanf(), то преобразовывалась обратно в число. Если мы хотим компактно представлять данные в файле, то числа следует хранить как числа, а не как строки. При этом целесообразно использовать бинарный режим доступа к файлу, т.к. будет гарантия, что любое записанное число не будет восприниматься как управляющий символ и будет корректно считан из файла.

Для работы с бинарными файлами предусмотрены функции fread() и fwrite() со следующим синтаксисом:

```
    size_t fread( void *buffer, size_t size, size_t count, FILE
*stream );
```

где *buffer – указатель на буфер памяти, в который будут считываться данные из файла; size – размер элемента в байтах; count - число считываний элементов; *stream – указатель на файл.

```
    size_t fwrite( void *buffer, size_t size, size_t count, FILE
*stream );
```

где *buffer – указатель на буфер памяти, из которого будут считываться данные в файл; size – размер элемента в байтах; count - число записей; *stream – указатель на файл.

Приведем пример использования функций fwrite() и fread().

Листинг 5.7. Использование функций fwrite() и fread().

```
#include <stdio.h>
void main( void )
{
   FILE *stream;
   char list[30];
   int  i, numread, numwritten;
   if( (stream = fopen( "fread.out", "wb" )) != NULL )
   {
      for ( i = 0; i < 25; i++ )
         list[i] = (char)('z' - i);
      numwritten = fwrite( list, sizeof( char ), 25, stream );
      printf( "Wrote %d items\n", numwritten );
      fclose( stream );
   }
   else  printf( "Problem opening the file\n" );
   if( (stream = fopen( "fread.out", "rb" )) != NULL )
   {
      numread = fread( list, sizeof( char ), 25, stream );
      printf( "Number of items read = %d\n", numread );
      printf( "Contents of buffer = %.25s\n", list );
      fclose( stream );
   }
   else printf( "File could not be opened\n" );
}
```

В данном примере массив list выступает в качестве буфера для вывода и ввода информации из бинарного файла. Сначала элементы буфера инициализируются буквами латинского алфавита от z до b, а затем записываются в файл с помощью функции fwrite(list, sizeof(char), 25, stream). Здесь оператор sizeof(char) указывает размер элемента (буквы), а число 25 соответствует числу записываемых букв. Аналогичным образом осуществляется считывание информации из файла fread(list, sizeof(char), 25, stream), где в массив list помещаются 25 символов, хранящихся в файле.

Функции fwrite() и fread() удобно использовать при сохранении данных структуры в файл. Запишем пример хранения информации по двум книгам в бинарном файле.

Листинг 5.8. Пример сохранения структур в бинарном файле.

```
#include <stdio.h>
#define N 2
```

```
struct tag_book
{
 char name[100];
 char author[100];
 int year;
} books[N];

int main(void)
{
 for(int i=0;i < N;i++)
 {
      scanf("%s",books[i].name);
      scanf("%s",books[i].author);
      scanf("%d",&books[i].year);
 }
 FILE* fp = fopen("my_file.txt","wb");
 fwrite(books, sizeof(books),1,fp);
 fclose(fp);

 fp = fopen("my_file.txt","rb");
 fread(books,sizeof(books),1,fp);
 fclose(fp);
 printf("-------------------------------------------------\n");
 for(i=0;i < N;i++)
 {
      puts(books[i].name);
      puts(books[i].author);
      printf("%d\n",books[i].year);
 }
    return 0;
}
```

В данном примере с помощью функции fwrite() целиком сохраняется массив books, состоящий из двух элементов, а оператор sizeof(books) определяет размер массива books. Аналогичным образом реализуется и функция fread(), которая считывает из файла сразу весь массив. По существу функции fwrite() и fread(), в данном примере, осуществляют копирование заданной области памяти в файл, а затем обратно. Это их свойство удобно использовать при хранении «сложных» форм данных, когда простая поэлементная запись данных в файл становится трудоемкой или невозможной.

Следует отметить, что функция fopen() при открытии файла на запись уничтожает все данные из этого файла, если они были. Вместе с тем существует необходимость добавлять данные в файл, не уничтожая ранее записанную информацию. Это достигается путем открытия файла на добавление информации. В этом случае функции fopen() третьим аргументом передается строка “a” или “ab”, что означает открыть файл на добавление информации в его конец. Продемонстрируем работу данного режима на следующем примере.

Листинг 5.9. Добавление информации в файл.

```
#include <stdio.h>
#define N 2
struct tag_book
{
 char name[100];
 char author[100];
 int year;
} books[N];
int main(void)
{
 for(int i=0;i < N;i++)
 {
      scanf("%s",books[i].name);
      scanf("%s",books[i].author);
      scanf("%d",&books[i].year);
 }
 FILE* fp = fopen("my_file.txt","wb");
 fwrite(&books[0], sizeof(tag_book),1,fp);
 fclose(fp);
 fp = fopen("my_file.txt","ab");
 fwrite(&books[1], sizeof(tag_book),1,fp);
 fclose(fp);
 fp = fopen("my_file.txt","rb");
 fread(books,sizeof(books),1,fp);
 fclose(fp);
 printf("-------------------------------------------------\n");
 for(i=0;i < N;i++)
 {
      puts(books[i].name);
      puts(books[i].author);
      printf("%d\n",books[i].year);
 }
    return 0;
}
```

В данном примере сначала создается файл my_file.txt, в который записывается информация по первой книге. Затем открывается этот же файл в режиме добавления и записывается информация по второй книге. В результате файл my_file.txt содержит информацию по обеим книгам, что подтверждается считыванием данных из этого файла и выводом информации на экран.

Когда стандартные функции возвращают EOF, это обычно означает, что они достигли конца файла. Однако это также может означать ошибку ввода информации из файла. Для того чтобы различить эти две ситуации в языке C++ существую функции feof() и ferror(). Функция feof() возвращает значение отличное от нуля, если достигнут конец файла и нуль в противном случае. Функция ferror() возвращает ненулевое значение, если произошла ошибка

чтения или записи, и нуль – в противном случае. Пример использования данных функций представлен в листинге 5.10.

Листинг 5.10. Использование функции ferror().

```
#include <stdio.h>
void main( void )
{
   int  count, total = 0;
   char buffer[100];
   FILE *fp;

   if( (fp = fopen( "my_file.txt", "r" )) == NULL )
      return;
   while( !feof( fp ) )
   {
      count = fread( buffer, sizeof( char ), 100, fp );
      if( ferror( fp ) )      {
         perror( "Read error" );
         break;
      }
      total += count;
   }
   printf( "Number of bytes read = %d\n", total );
   fclose( fp );
}
```

В языке C++ имеются также функции remove() и rename() для удаления и переименования файлов. Их синтаксис следующий:

```
int remove( const char *path );
```

где *path – путь с именем удаляемого файла. Данная функция определена в библиотеках stdio.h и io.h, возвращает нуль при успешном удалении и -1 в противном случае.

```
int rename( const char *oldname, const char *newname );
```

где *oldname – имя файла для переименования; *newname – новое имя файла. Данная функция определена в библиотеках stdio.h и io.h, возвращает нуль при успешном удалении и не нуль в противном случае.

5.3. Пример программирования. Простой словарь

Рассмотрим некоторые приемы программирования на примере создания программы простого словаря. Данная программа будет позволять добавлять новые слова в словарь и переводить русские слова на английские. Кроме того, все введенные слова будут сохраняться в файл и загружаться при очередном

запуске программы. Для реализации всех этих возможностей подключим необходимые библиотеки стандартных функций:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
```

Создание любой программы должно начинаться с анализа способа представления данных в программе. В данном случае русское слово и его английский эквивалент целесообразно объеденить в структуру, которая и будет представлять элемент данных программы. Так как количество слов в словаре может быть любым, то следует использовать связный список для их описания. Таким образом, получаем следущий тип структуры для описания слов в словаре (объекта данных):

```
typedef struct tag_word {
 char wd_eng[100];        //русское слово
 char wd_rus[100];        //английский эквивалент
 tag_word* prev, *next;   //указатели для работы связного
списка
} WORD_DICT;
```

Объявим следующие вспомогательные глобальные переменные для работы связного списка:

```
WORD_DICT* head = NULL, *tail = NULL;
```

Работа программы должна начинаться с загрузки ранее введенной информации (словаря) в память компьютера. Для этого введем новую функцию LoadDict() и опишем ее следующим образом:

```
void LoadDict()
{
 FILE* fp = fopen("dict.dat","rb");
 if(fp != NULL)
 {
      while(!feof(fp))
      {
           WORD_DICT* word = (WORD_DICT *)
                              malloc(sizeof(WORD_DICT));
           fread(word,sizeof(WORD_DICT),1,fp);
           List_dict_add(word);
      }
      fclose(fp);
 }
}
```

В данной функции в цикле while сначала создается новый объект WORD_DICT, а затем в него записывается информация из файла dict.dat. После этого новый созданный объект необходимо поместить в связный список. Это осуществляется с помощью функции List_dict_add(), реалзиация которой приведена ниже:

```
void List_dict_add(WORD_DICT* add_word)
{
 if(head == NULL) head = add_word;

 add_word->prev = tail;
 add_word->next = NULL;

 if(tail != NULL) tail->next = add_word;
 tail = add_word;
}
```

Данная функция в качестве аргумента принимает указатель на объект, который должент быть помещен в конец связного списка. Для этого достаточно настроить его указатели prev и next соответственно на предыдущий и последующий объекты, если они существуют. Если объект add_word является первым в списке, то указатель head должен указывать на него. Это выполняется в первом условии данной функции. Во второй строке указателю prev объекта присваивается адрес последнего объекта в списке (указатель tail всегда указывает на последний объект). Учитывая, что объект добавляется в конец списка, то указатель next должен быть равен NULL. Наконец, в последнем условии проверяется, существует ли последний объект в списке, и если существует, то его указатель next должен указывать на вновь добавленный объект, а сам указатель tail указывать на него.

После считывания информации из файла и формирования начального связного списка, следует на экран вывести меню, в котором пользователь может выбирать нужную последовательность действий: добавить новое слово, выполнить перевод, выйти из программы. Это можно реализовать с помощью функции ShowMainMenu() следующим образом:

```
int ShowMainMenu()
{
 int ret = 2;
 printf("\n");
 printf("1. Add words to a dictionary\n");
 printf("2. Translate\n");
 printf("3. Exit from the application\n");
 printf("\n");
 printf("Choose an menu item: ");scanf("%d",&ret);

 return ret;
}
```

Данная функция выводит на экран строки меню и предлагает пользователю сделать выбор. Номер выбранного пункта возвращается функцией главной программе. Там же осуществляется проверка номера выбранного пункта и выполняются соответствующие действия. Таким образом, функция main() принимает вид:

```
int main(int argc, char* argv[])
{
 LoadDict();
 int item = ShowMainMenu();
 while(item != 3)
 {
      switch(item)
      {
      case 1:Add_words();break;
      case 2:Translate();break;
      case 3:break;
      }
      item = ShowMainMenu();
 }

 SaveDict();
 FreeDict();

 return 0;
}
```

Здесь сначала вызывается функция загрузки словаря, а затем выводится главное меню. Для того чтобы меню было доступно в процессе работы программы, реализуется цикл, в котором осуществляется проверка номера выбранного пункта и выполняются соответствующие действия. Можно заметить, что если пользователь выбрал третий пункт меню (выход из программы), то работа цикла while() завершается, и программа переходит к функциям SaveDict() и FreeDict(), которые сохраняют информацию в файл и освобождают память (удаляют связный список).

Рассмотрим работу функции Add_words(), которая добавляет новые слова в словарь:

```
void Add_words()
{
 char rus_word[100], eng_word[100];
 char ch;

 do
 {
      printf("\n--------Add words-------------\n");
      printf("Input a russian word: ");scanf("%s",rus_word);
      printf("Input an english word: ");scanf("%s",eng_word);
      printf("Add the word to a dictionary? Y/N: ");
```

```
        ch = getch();
        if(ch == 'Y' || ch == 'y')
        {
             WORD_DICT* word = (WORD_DICT *)
                                  malloc(sizeof(WORD_DICT));
             strcpy(word->wd_eng,eng_word);
             strcpy(word->wd_rus,rus_word);
             List_dict_add(word);
        }
        printf("\nWould you like input any word? Y/N: ");
        ch = getch();
  } while(ch != 'N' && ch != 'n');
}
```

Здесь вводятся русское и английское слова, которые затем, при согласии пользователя, добавляются в связный список. Для этого, сначала создается объект типа WORD_DICT в памяти компьютера, затем инициализируются его поля wd_eng и wd_rus соответственно английским и русским введенными словами и с помощью функции List_dict_add() объект добавляется в конец связного списка. Данная процедура продолжается до тех пор, пока пользователь не выйдет из цикла do while(), т.е. пока на последний вопрос не ответит символом ‘N’.

Реализация функции Translate() подобна функции Add_words() и имеет вид:

```
void Translate()
{
 char rus_word[100];
 char ch;

 do
 {
      printf("\n--------Translate english words to russian----
---------\n");
      printf("Input a russian word: ");scanf("%s",rus_word);

      WORD_DICT* current = head;
      while(current != NULL)
      {
           if(strcmp(current->wd_rus,rus_word) == 0)
           {
                printf("English: %s\n",current->wd_eng);
                break;
           }
           current = current->next;
      }
      if(current == NULL) printf("The word don't find\n");

      printf("\nWould you like input any word? Y/N: ");
      ch = getch();
```

```
  } while(ch != 'N' && ch != 'n');
}
```

Здесь после ввода русского слова с клавиатуры выполняется цикл while, в котором просматриваются все объекты связного списка и проверяется условие: если введенное русское слово совпадает с русским словом из словаря, то на экран выводится английское слово и цикл досрочно завершается оператором break. После цикла проверяется условие на равенство указателя current зачению NULL. Это возможно только в том случае, если в словаре не было найдено нужного русского слова. Поэтому на экран, в этом случае, будет выведено сообщение «Слово не найдено».

Функция SaveDict() вызывается перед завершением программы и имеет следующую реализацию:

```
void SaveDict()
{
 FILE* fp = fopen("dict.dat","wb");
 if(fp != NULL)
 {
      WORD_DICT* current = head;
      while(current != NULL)
      {
           fwrite(current,sizeof(WORD_DICT),1,fp);
           current = current->next;
      }
 }
 fclose(fp);
}
```

Наконец, последняя функция FreeDict() удаляет выделенную память перед выходом из программы и опредляется как:

```
void FreeDict()
{
 while(head != NULL) List_dict_remove(head);
}
```

где функция List_dict_remove()определена в виде:

```
WORD_DICT* List_dict_remove(WORD_DICT* crn)
{
 if(crn->prev != NULL) crn->prev->next = crn->next;
 if(crn->next != NULL) crn->next->prev = crn->prev;

 if(crn == head) if(crn->prev != NULL) head = crn->prev;
 else head = crn->next;

 if(crn == tail)     if(crn->next != NULL) tail = crn->next;
```

```
 else tail = crn->prev;

 free(crn);
 return tail;
}
```

Данная функция удаляет объект, на который указывает crn из связного списка. Для этого следует переопределить указатели предыдущего и последующего объектов в списке (если они существуют) относительно удаляемого, а также указатели head и tail, в случае, когда удаляется или первый или последний объект из списка. Данную функцию удобно использовать при удалении как одного отдельного слова, так и всего списка в целом. Подумайте, как можно скорректировать программу для добавления возможности удаления слова из словаря.

Таким образом, реализована простая программа с 8 пользовательскими функциями, которые удобно записать в виде прототипов перед функцией main(), а их реализации после нее.

Контрольные вопросы и задания

1. Дайте понятие файла.
2. Для чего предназначена функция fopen() и в какой библиотеке она определена?
3. Чему должен быть равен второй аргумент функции fopen() для открытия файла на чтение?
4. Какое значение возвращает функция fopen() при неудачном открытии файла?
5. Дайте понятие текстового режима доступа к файлу.
6. Для чего предназначены функции getc(), fgets() и fscanf()?
7. Запишите программу для записи информации по книгам в файл с помощью функции fprintf().
8. В чем отличие режима добавления информации в файл от режима записи информации?
9. Что делают функции fseek() и ftell()?
10. Дайте понятие бинарного режима доступа к файлу.
11. Какие функции позволяют записывать и считывать информацию из бинарного файла?
12. Для чего предназначена функция fclose()?
13. Какой символ соответствует концу файла?
14. Приведите функции для переименования и удаления файла.
15. Для чего нужны функции ferror() и feof()?
16. Приведите программу записи структуры в бинарный файл.

ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

В предыдущих примерах программы представляли собой набор функций и директив, идущих друг за другом в определенном порядке. Это структурный подход к программированию. Ему присущи некоторые недостатки, связанные с формализацией задач программирования. Такие задачи, как правило, связаны с манипулированием разнородных объектов. Например, создается графический редактор, где необходимо выполнить работу с графическими примитивами: линией, эллипсом и прямоугольником. При этом в соответствующих функциях программы возникает необходимость перебирать типы объектов, например, для их добавления, редактирования и отображения на экране. В результате получается громоздкий текст программы, в котором многократно описываются одни и те же действия. Теперь представим, что в рамках данной задачи общее число возможных графических элементов не три, а сто. В этом случае получится еще более сложная программа, в которой станет сложно ориентироваться. Кроме того, на практике часто наперед неизвестно общее число возможных вариантов (типов графических объектов) и исправление текста программы для добавления возможности работы с новым объектом становится трудоемкой задачей для программиста.

Представленный пример создания графического редактора показывает лишь один из недостатков структурного программирования. В общем случае существует большое множество разнообразных задач плохо поддающихся описанию, вследствие чего и возникла необходимость разработки нового подхода к программированию, которым стало объектно-ориентированное программирование (ООП).

6.1. Понятие классов в C++

Идея ООП заключается в описании задачи на уровне объектов, которые в языке C++ называются классами. Например, класс может описывать объект линию, эллипс, прямоугольник. Но в отличие от структур, которые также могут комплексно описывать свойства каких-либо объектов, между классами возможны взаимодействия, которые выражаются тремя категориями: наследование, полиморфизм и инкапсуляция.

Наследование – это механизм создания нового класса на основе ранее созданного. Наследование имеет смысл, если множество разнородных объектов имеют общие характеристики или функции. Так, в случае с графическими примитивами тип линии, цвет и толщина описываются одинаково на уровне языка программирования и логически относятся к одной категории – свойства графического примитива. Поэтому эти элементы целесообразно выделить в отдельный класс – базовый и на основе него

создавать новые классы – дочерние для более детального описания линии, эллипса и прямоугольника, используя механизм наследования.

Полиморфизм – это процесс вызова и переопределение функций базового класса в дочерних. Полиморфизм позволяет общие функции дочерних классов выносить в базовый, а затем их вызывать из дочернего, полагая, что они определены в нем. Вместе с тем это не исключает возможности переопределения функций базового класса в дочерних.

Инкапсуляция – это способ представления класса в виде «черного ящика». Это значит, что конечному пользователю класса (программисту) доступен лишь определенный набор функций и переменных для работы с классом. Часто ограничение доступа применяется для записи значений в переменные класса через функции, при запрещенном непосредственном доступе к переменным.

Класс в языке C++ задается с помощью ключевого слова class, за которым следует его имя и в фигурных скобках {} дается его описание. После определения класса ставится точка с запятой. Ниже приведен пример описания класса для хранения координат графических примитивов:

```
class CPos
{
 int sp_x, sp_y;     //координата начала
 int ep_x, ep_y;     //координата конца
};
```

Каждый класс имеет специальные функции, которые называются конструктор и деструктор. Конструктор класса вызывается всякий раз, когда объект создается в памяти ЭВМ и служит обычно для инициализации данных класса. Конструктор имеет то же имя, что и имя класса. Деструктор вызывается при удалении класса из памяти и используется, как правило, для освобождения ранее выделенной памяти под какие-либо данные этого класса. Имя деструктора совпадает с именем класса, но перед ним ставится символ ‘~’. Рассмотрим пример реализации конструктора и деструктора для класса CPos.

```
class CPos
{
public:
 CPos() {printf("Вызов конструктора.\n");}
 ~CPos() {printf("Вызов деструктора.\n");}

 int sp_x, sp_y;     //координата начала
 int ep_x, ep_y;     //координата конца
};
```

Здесь ключевое слово public используется для обеспечения общего доступа к функциям и переменным класса.

Для создания нового экземпляра класса в памяти ЭВМ используется оператор new языка C++, а для удаления – оператор delete. Использование

данных операторов для создания экземпляра класса CPos и его удаления выглядит следующим образом:

```
CPos *pos_ptr = new CPos();    //создание объекта
delete pos_ptr;                //удаление объекта
```

В результате выполнения этих двух строк программы на экране появятся сообщения:

Вызов конструктора.
Вызов деструктора.

Экземпляр класса также можно создать подобно обычным переменным без использования указателей, как показано ниже

```
CPos pos;
```

В этом случае переменная pos называется представителем класса, у которого также вызывается конструктор при его создании и деструктор при его удалении из памяти.

Следует отметить, что при создании нового экземпляра класса можно выполнять инициализацию различных переменных путем передачи их значений через конструктор. В этом случае конструктор должен быть объявлен с набором необходимых аргументов, например, так:

```
class CPos
{
public:
 CPos(int x1, int y1, int x2,int y2)
 {
      sp_x = x1; sp_y = y1;
      ep_x = x2; ep_y = y2;
 }
 ~CPos() {}

 int sp_x, sp_y;
 int ep_x, ep_y;
};
```

и процесс создания экземпляра класса принимает вид:

```
CPos *pos_ptr = new CPos(10,10,20,20);
```

или

```
CPos pos(10,10,20,20);
```

Такой способ описания и вызова конструктора представляет дополнительное удобство инициализации данных при создании нового объекта. При этом конструктор, как и любую функцию, можно перегружать.

Это значит, что можно задать несколько типов конструкторов (с разным набором входных параметров) в одном и том же классе. Например, если создается экземпляр класса графического примитива, но для него неизвестны начальные и конечные координаты, то целесообразно вызвать конструктор CPos() без аргументов, а если координаты известны, то выполнить их инициализацию путем вызова конструктора с аргументами. Для описания нескольких типов конструкторов в одном классе достаточно дать их определения в нем:

```
class CPos
{
public:
 CPos() {}
 CPos(int x1, int y1, int x2,int y2)
 {
      sp_x = x1; sp_y = y1;
      ep_x = x2; ep_y = y2;
 }
 ~CPos() {}

 int sp_x, sp_y;
 int ep_x, ep_y;
};
```

В классах помимо переменных, конструкторов и деструкторов можно задавать описания и обычных функций, которые, в этом случае, называются методами. Например, в классе CPos для задания значений координат примитива целесообразно добавить функцию для ввода значений в переменные sp_x, sp_y, ep_x и ep_y. Это позволит, во-первых, не запоминать программисту имена этих переменных, а оперировать только одной функцией и, во-вторых, в самой функции можно реализовать необходимые проверки на истинность переданных значений координат перед их присваиванием переменным. Такую функцию можно описать в классе следующим образом:

```
class CPos
{
public:
 CPos() {}
 ~CPos() {}

 void SetParam(int x1, int y1, int x2, int y2)
 {
      if(x1 >= 0 && x1 <= MAX_SIZE) sp_x = x1;
      if(y1 >= 0 && y1 <= MAX_SIZE) sp_y = y1;
      if(x2 >= 0 && x2 <= MAX_SIZE) ep_x = x2;
      if(y2 >= 0 && y2 <= MAX_SIZE) ep_y = y2;
 }

 int sp_x, sp_y;
```

```
 int ep_x, ep_y;
};
```

В приведенном примере реализована функция SetParam(), которая перед присваиванием значений переменных выполняет проверку на их истинность. Здесь некоторое неудобство представляет то, что данная функция полностью описана в классе CPos, а описание большого числа функций в одном классе делает текст программы трудночитаемым. Поэтому обычно в классах записывают лишь прототипы функций, а их реализации приводят отдельно после описания класса. Для того чтобы описать реализацию функции SetParam() вне класса CPos перед именем функции ставится имя класса с оператором глобального разрешения ‘::’ как показано ниже:

```
void CPos::SetParam(int x1, int y1, int x2, int y2)
{
 if(x1 >= 0 && x1 <= MAX_SIZE) sp_x = x1;
 if(y1 >= 0 && y1 <= MAX_SIZE) sp_y = y1;
 if(x2 >= 0 && x2 <= MAX_SIZE) ep_x = x2;
 if(y2 >= 0 && y2 <= MAX_SIZE) ep_y = y2;
}
```

а перед ней должно идти следующее определение класса:

```
class CPos
{
public:
 CPos() {}
 ~CPos() {}

 void SetParam(int x1, int y1, int x2, int y2);

 int sp_x, sp_y;
 int ep_x, ep_y;
};
```

Аналогичным образом можно давать описание конструкторов и деструкторов за пределами класса. Учитывая, что данные функции ничего не возвращают вызывающей программе и не имеют типов, то их внешняя реализация будет иметь вид:

```
CPos::CPos()
{
 //операторы конструктора
}

CPos::~CPos()
{
 //операторы деструктора
}
```

Функцию SetParam() можно вызывать через указатель на класс, используя оператор ‘->’ или через представитель с помощью оператора ‘.’:

```
CPos* pos_ptr = new CPos();
CPos pos;

pos_ptr->SetParam(10,10,20,20);
pos.SetParam(10,10,20,20);
```

Таким же образом можно обращаться и к переменным класса:

```
pos_ptr->sp_x = 10;
pos.sp_x = 20;
```

Здесь можно заметить, что значения переменных sp_x, sp_y, ep_x и ep_y могут быть заданы как непосредственно при обращении к ним, так и с помощью функции SetParam(). В результате проверка, реализованная в данной функции, может быть проигнорирована программистом. Часто такая ситуация недопустима, например, при использовании готовых классов библиотек MFC, VCL, OWL и др. В связи с этим в классах для переменных и функций предусмотрена возможность установки разных уровней доступа, которые определяются тремя ключевыми словами: public, private и protected.

Ключевое слово public означает общий доступ к переменным и функциям класса. Уровень доступа private указывает на частный способ доступа к элементам класса и устанавливается по умолчанию при описании класса. Частный уровень доступа дает возможность обращаться к переменным и функциям только внутри класса и запрещает извне, например, через представители или указатели на класс. Режим доступа protected также как и private запрещает доступ к элементам класса через представители и указатели, но разрешает обращаться к ним из дочерних классов при наследовании.

Учитывая эти три режима доступа, класс для работы с координатами графических объектов целесообразно записать в таком виде:

```
class CPos
{
public:
 CPos() {}
 ~CPos() {}

 void SetParam(int x1, int y1, int x2, int y2);

private:
 int sp_x, sp_y;
 int ep_x, ep_y;
};
```

Здесь раздел private ограничивает доступ пользователю класса к переменным sp_x, sp_y, ep_x и ep_y только функцией SetParam(). Следует также отметить, что отсутствие раздела public вначале описания класса привело бы к тому, что все функции класса CPos имели бы область видимости private. В результате доступ к конструктору и деструктору был бы запрещен, и создание нового объекта стало бы невозможным. Аналогичная картина имеет место и в режиме доступа protected, но в отличие от private класс можно использовать как базовый в механизме наследования. Это свойство полезно использовать для запрета создания экземпляров класса, что бывает необходимым, если он является лишь промежуточным звеном в иерархии объектов и не представляет ценности как отдельный объект.

6.2. Наследование

Рассмотренный класс CPos не описывает особенностей работы с конкретными графическими примитивами: линией, прямоугольником, эллипсом, а содержит лишь общую для них информацию. Поэтому данный класс следует рассматривать как базовый, на основе которого можно построить дочерние для более детальной работы с графическими объектами, используя механизм наследования.

Предположим, создается дочерний класс с именем CLine для работы с линией на основе базового CPos. Для этого после имени дочернего класса CLine ставится символ ‘:’, а затем пишется имя базового класса CPos с указанием уровня доступа:

```
class CPos
{
public:
 CPos() {}
 CPos(int x1, int y1, int x2, int y2) {SetParam(x1,y1,x2,y2);}
 ~CPos() {}

 void SetParam(int x1, int y1, int x2, int y2);

protected:
 int sp_x, sp_y;
 int ep_x, ep_y;
};

class CLine : public CPos
{
public:
 CLine() {}
 CLine(int x1,int y1, int x2, int y2) {SetParam(x1,y1,x2,y2);}
 ~CLine() {}

 void Draw() {MoveTo(sp_x,sp_y); LineTo(ep_x,ep_y);}
};
```

В результате наследования с уровнем доступа public класс CLine имеет доступ ко всем переменным и функциям класса CPos, которые не являются частными (private). Ключевое слово public перед именем класса CPos означает, что все общие (public) элементы этого класса остаются с таким же уровнем доступа и в классе CLine. Следует также отметить, что описание класса CPos должно предшествовать описанию класса CLine, а переменные sp_x, sp_y, ep_x и ep_y должны быть описаны в разделе protected для возможности их использования в функции Draw() дочернего класса CLine и в то же время не доступными извне.

Класс CLine содержит два конструктора, деструктор и функцию Draw() для рисования линии на экране. При этом процедура задания координат графического объекта целиком находится в базовом классе CPos и по мере необходимости используется в дочернем классе CLine. Такое разделение оказывается удобным, т.к. при описании работы с новыми графическими объектами процедура работы с их координатами будет оставаться одной и той же и находится в одном классе. Если бы в данном случае использовался структурный подход к программированию, то алгоритм работы с координатами графических примитивов пришлось бы прописывать каждый раз для всех типов объектов, что привело бы к заметному усложнению текста программы.

Для работы с дочерним классом, также как и с обычным, необходимо создать его экземпляр либо с помощью оператора new, либо через представитель, как показано ниже:

```
    CLine* line_ptr = new CLine();
```

или

```
    CLine line;
```

При создании нового объекта CLine вызывается сначала конструктор CPos() базового класса, а затем конструктор дочернего – CLine(). Таким образом, создается как бы два объекта: CPos и CLine, но они представляются как единое целое объекта CLine.

В представленном классе CLine предусмотрено два конструктора: с параметрами и без них. В случае вызова конструктора с параметрами

```
    CLine line(10,10,20,20);
```

вызывается конструктор CPos() базового класса, а затем конструктор CLine(int x1, int y1, int x2, int y2) дочернего, в котором выполняется функция SetParam() для записи значений координат графического объекта.

Последние два приведенных примера создания объекта CLine показывают, что вне зависимости от типа вызываемого конструктора дочернего класса всегда вызывается один и тот же конструктор CPos() базового класса, даже если в последнем определено несколько конструкторов. Это не всегда удобно и

кроме того, если конструктор CPos() не описан в базовом классе, то создание дочернего класса CLine станет невозможным, т.к. конструктор по умолчанию CPos() не будет найден. Для того чтобы исправить такую ситуацию необходимо в дочернем классе указать, какой именно конструктор базового класса следует вызывать, следующим образом:

```
class CLine : public CPos
{
public:
 CLine() : CPos()
 {
 }
 CLine(int x1,int y1, int x2, int y2) : CPos(x1,y1,x2,y2)
 {
 }
 ~CLine() {}

 void Draw() {MoveTo(sp_x,sp_y); LineTo(ep_x,ep_y);}
};
```

В приведенном примере конструктор CLine() будет вызывать конструктор CPos() базового класса, а конструктор CLine(int x1, int y1, int x2, int y2) конструктор CPos(int x1, int y1, int x2, int y2). При этом функция SetParam() в CLine(int x1, int y1, int x2, int y2) может быть опущена, т.к. необходимая инициализация переменных будет выполнена при вызове конструктора CPos(int x1, int y1, int x2, int y2) базового класса.

В рассматриваемой задаче программирования графического редактора, класс CPos является вспомогательным, т.е. он служит для создания описания новых классов как базовый. При этом нет необходимости создавать его экземпляр в памяти ЭВМ для непосредственной работы с ним. Поэтому целесообразно защитить его от возможности создания путем помещения конструкторов данного класса в раздел protected. Такие классы называются абстрактными, т.е. они не могут существовать как самостоятельные объекты, а служат для создания новых, дочерних классов. Описание абстактного класса CPos и дочернего от него CLine показано ниже:

```
class CPos
{
protected:
 CPos() {}
 CPos(int x1, int y1, int x2, int y2) {SetParam(x1,y1,x2,y2);}
 ~CPos() {}

public:
 void SetParam(int x1, int y1, int x2, int y2);

protected:
 int sp_x, sp_y;
```

```
 int ep_x, ep_y;
};
```

Функции классов CPos и CLine можно вызывать, например, через представитель класса CLine, следующим образом:

```
CLine line;
line.SetParam(10,10,20,20);
line.Draw();
```

Обратите внимание, что благодаря полиморфизму, функция SetParam(), заданная в классе CPos, вызывается через представитель line как будто она определена в классе CLine. В результате, единожды объявленная функция SetParam() может быть многократно использована в разных классах, производных от CPos.

Для работы с другими графическими примитивами (прямоугольником и эллипсом) подобным образом можно создать дочерние классы от CPos, отличающихся друг от друга реализацией функции Draw():

```
class CRect : public CPos
{
public:
 CRect() : CPos()
 {
 }
 CRect(int x1,int y1, int x2, int y2) : CPos(x1,y1,x2,y2)
 {
 }
 ~CRect() {}

 void Draw() {Rectangle(sp_x,sp_y,ep_x,ep_y);}
};

class CEllipse : public CPos
{
public:
 CEllipse() : CPos()
 {
 }
 CEllipse(int x1,int y1, int x2, int y2) : CPos(x1,y1,x2,y2)
 {
 }
 ~CEllipse() {}

 void Draw() {Ellipse(sp_x,sp_y,ep_x,ep_y);}
};
```

В результате построения объектов получается следующая иерархия (рис. 6.1).

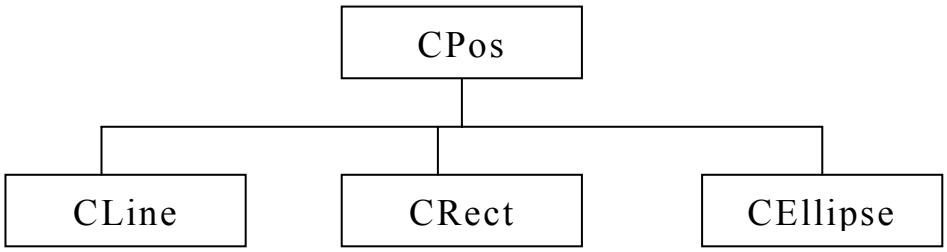

Рис. 6.1. Иерархия классов графических примитивов

У каждого из представленных дочерних объектов CLine, CRect и CEllipse имеется один базовый объект CPos. Вместе с тем язык C++ предоставляет возможность создавать дочерние объекты на основе нескольких базовых, что приводит к концепции множественного наследования.

В рамках данной задачи множественное наследование будет иметь смысл, если добавить еще один абстрактный класс с именем CProp, который будет отвечать за свойства графических примитивов: толщина и цвет линии:

```
class CProp
{
protected:
 CProp() {}
 CProp(int wdt, int clr) {SetProperty(wdt,clr);}
 ~CProp();

public:
 void SetProperty(int wdt, int clr)
 {
      if(wdt >= 0 && wdt <= MAX_WIDTH) width = wdt;
      if(clr >= 0 && clr <= MAX_COLOR) color = clr;
 }
protected:
 int width, color;
};
```

Теперь дочерние классы CLine, CRect и CEllipse можно образовывать от двух базовых CPos и CProp, которые являются не связанными друг с другом. Для того чтобы построить класс на основе двух базовых они указываются друг за другом через запятую следующим образом:

```
class CLine : public CPos, public CProp
{
public:
 CLine() : CPos(), CProp() {}
 CLine(int x1, int y1, int x2, int y2, int w, int clr) :
           CPos(x1,y1,x2,y2), CProp(w,clr) {}
 ~CLine() {}

 void Draw() {SetWidth(width); SetColor(color);
```

```
                    MoveTo(sp_x, sp_y); LineTo(ep_x, ep_y);
                    }
};
```

Аналогичным образом строятся классы CRect и CEllipse. Здесь следует отметить, что конструктор CLine(int x1, int y1, int x2, int y2, int w, int clr) класса CLine вызывает конструкторы двух базовых классов, которые перечислены через запятую с указанием в них конкретных переменных. Работа с функциями класса CLine через его представитель имеет следующий вид:

```
CLine line;
line.SetProperty(1,0);
line.SetParam(10,10,20,20);
line.Draw();
```

Благодаря полиморфизму, функции SetProperty() и SetParam() базовых классов вызываются непосредственно из класса CLine.

6.3. Дружественные классы и функции

В рассмотренных примерах наследования функция Draw() дочерних классов использует переменные sp_x, sp_y, ep_x, ep_y, width и color базовых классов, которые необходимо было объявлять в разделе protected. Вместе с тем лучшую защиту этих переменных можно обеспечить, объявив их частными (private). Но тогда при наследовании они оказываются недоступными в производных классах, а реализация функции Draw() невозможной. Чтобы разрешить эту проблему, дочерние классы, использующие частные переменные базовых классов, необходимо объявить как дружественные к соответствующим базовым.

Для объявления дружественного класса используется ключевое слово friend, за которым следует имя класса. Следующий пример демонстрирует объявление дружественного класса CLine классам CPos и CProp:

```
class CLine;          //идентификатор класса

class CPos
{
protected:
 CPos() {}
 CPos(int x1, int y1, int x2, int y2) {SetParam(x1,y1,x2,y2);}
 ~CPos() {}

public:
 friend CLine;  //объявление дружественного класса

 void SetParam(int x1, int y1, int x2, int y2);
```

```
protected:
 int sp_x, sp_y;
 int ep_x, ep_y;
};

class CProp
{
protected:
 CProp() {}
 CProp(int wdt, int clr) {SetProperty(wdt,clr);}
 ~CProp();

public:
 friend CLine;  //объявление дружественного класса

 void SetProperty(int wdt, int clr);

protected:
 int width, color;
};
```

В данном примере класс CLine является производным от классов CPos и CProp, поэтому он объявляется после них. Однако, чтобы сделать класс CLine дружественным базовым классам он должен быть объявлен до них, иначе компилятор C++ выдаст синтаксическую ошибку. Чтобы разрешить эту проблему язык C++ допускает использование идентификатора класса, который говорит компилятору о том, что такой класс есть, но его описание будет дано ниже. Благодаря этому удается организовать перекрестные ссылки между классами, стоящие на разных уровнях иерархии. В результате такой организации частные элементы sp_x, sp_y, ep_x, ep_y, width и color классов CPos и CProp оказываются доступными только одному производному классу CLine и никакому другому, что обеспечивает их лучшую защиту по сравнению с уровнем доступа protected.

Дружественными можно объявлять не только классы, но и отдельные функции классов. Например, для класса CLine важно, чтобы переменные sp_x, sp_y, ep_x, ep_y, width и color были доступны только функции Draw(). Поэтому было бы целесообразно ее и сделать дружественной, а не весь класс целиком. Однако для этого потребовалось бы ее прототип описать до классов CPos и CProp, что сделать в данном случае невозможно, т.к. класс CLine, в котором находится функция Draw(), описан в последнюю очередь. Но, в общем, дружественные функции можно задавать, как показано в следующем фрагменте программы:

```
class CPos;
class CLine
{
public:
 CLine() {}
```

```
 ~CLine() {}

 void Draw(CPos* pos);
};

class CPos
{
public:
 CPos() {}
 ~CPos() {}

 friend void CLine::Draw(CPos* pos);

private:
 int sp_x, sp_y;
 int ep_x, ep_y;
};

void CLine::Draw(CPos* pos)
{
 MoveTo(pos->sp_x, pos->sp_y);
 LineTo(pos->ep_x, pos->ep_y);
}
```

Особенностью организации классов CLine и CPos является то, что функция Draw() класса CLine использует в качестве аргумента указатель на класс CPos, который объявлен ниже. Поэтому реализации функции Draw() должна быть объявлена после определения класса CPos, иначе компилятор C++ выдаст сообщение об ошибке. Благодаря тому, что функция Draw() является дружественной классу CPos, она может получать доступ к частным элементам этого класса через переданный ей указатель.

6.4. Виртуальные функции

Описанные классы CLine, CRect и CEllipse могут быть использованы для хранения и отображения на экране соответствующих графических примитивов. Предположим, что в программе создается три графических объекта: линия, прямоугольник и эллипс, с некоторыми заданными координатами и свойствами. В этом случае алгоритм хранения и отображения объектов может быть представлен следующим образом.

Листинг 6.1. Пример работы с классами CLine, CRect и CEllipse

```
#include <stdio.h>

#define TOTAL_OBJ 3
#define MAX_SIZE 1024

typedef enum type {OBJ_LINE, OBJ_RECT, OBJ_ELLIPSE} TYPE;
```

```
class CLine;
class CRect;
class CEllipse;

class CObj
{
protected:
 CObj() {}
 ~CObj() {}

public:
 TYPE type_obj;
};

class CPos : public CObj
{
protected:
 CPos() {}
 CPos(int x1, int y1, int x2, int y2) {SetParam(x1,y1,x2,y2);}
 ~CPos() {}

public:
 void SetParam(int x1, int y1, int x2, int y2)
 {
      if(x1 >= 0 && x1 <= MAX_SIZE) sp_x = x1;
      if(y1 >= 0 && y1 <= MAX_SIZE) sp_y = y1;
      if(x2 >= 0 && x2 <= MAX_SIZE) ep_x = x2;
      if(y2 >= 0 && y2 <= MAX_SIZE) ep_y = y2;
 }

 friend CLine;
 friend CRect;
 friend CEllipse;

private:
 int sp_x, sp_y;
 int ep_x, ep_y;
};

class CProp
{
protected:
 CProp() {}
 CProp(int w, int clr) {SetProperty(w,clr);}
 ~CProp() {}

public:
 void SetProperty(int w, int clr)
 {
      if(w >= 0 && w <= 5) width = w;
      if(clr >= 0 && clr <= 15) color = clr;
```

```
     }

     friend CLine;
     friend CRect;
     friend CEllipse;

    private:
     int width, color;
    };

    class CLine : public CPos, public CProp
    {
    public:
     CLine() : CPos(), CProp() {type_obj = OBJ_LINE;}
     CLine(int x1,int y1, int x2, int y2, int w, int clr) :
                              CPos(x1,y1,x2,y2), CProp(w,clr)
                              {type_obj = OBJ_LINE;}

     ~CLine() {printf("Удаление объекта линия\n");}

     void Draw()
     {
          printf("Рисование линии с координатами (%d, %d) и (%d,
%d)\n", sp_x, sp_y, ep_x, ep_y);
     }
    };

    class CRect : public CPos, public CProp
    {
    public:
     CRect() : CPos(), CProp() {type_obj = OBJ_RECT;}
     CRect(int x1,int y1, int x2, int y2, int w, int clr) :
                              CPos(x1,y1,x2,y2), CProp(w,clr)
                              {type_obj = OBJ_RECT;}
     ~CRect() {printf("Удаление объекта прямоугольник\n");}

     void Draw()
     {
          printf("Рисование прямоугольника с координатами (%d, %d)
и (%d, %d)\n", sp_x, sp_y, ep_x, ep_y);
     }
    };

    class CEllipse : public CPos, public CProp
    {
    public:
     CEllipse() : CPos(), CProp() {type_obj = OBJ_ELLIPSE;}
     CEllipse(int x1,int y1, int x2, int y2, int w, int clr) :
                              CPos(x1,y1,x2,y2), CProp(w,clr)
                              {type_obj = OBJ_ELLIPSE;}
     ~CEllipse() {printf("Удаление объекта эллипс\n");}
```

```
     void Draw()
     {
          printf("Рисование эллипса с координатами (%d, %d) и (%d,
%d)\n", sp_x, sp_y, ep_x, ep_y);
     }
    };

    int main(int argc, char* argv[])
    {
     CObj* obj[TOTAL_OBJ];

     obj[0] = new CLine(10,10,20,20,1,0);
     obj[1] = new CRect(30,30,40,40,1,0);
     obj[2] = new CEllipse(50,50,60,60,1,0);

     for(int i = 0;i < TOTAL_OBJ;i++)
     {
       switch(obj[i]->type_obj)
       {
       case OBJ_LINE:{CLine* line = (CLine *)obj[i];
                      line->Draw();
                      break;}
       case OBJ_RECT:{CRect* rect = (CRect *)obj[i];
                      rect->Draw();
                      break;}
       case OBJ_ELLIPSE:{CEllipse* ellipse = (CEllipse *)obj[i];
                      ellipse->Draw();
                      break;}
       }
     }

     for(i = 0;i < TOTAL_OBJ;i++)
     {
          switch(obj[i]->type_obj)
          {
          case OBJ_LINE:delete (CLine *)obj[i];break;
          case OBJ_RECT:delete (CRect *)obj[i];break;
          case OBJ_ELLIPSE:delete (CEllipse *)obj[i];break;
          }
     }

     return 0;
    }
```

В данном примере был введен новый класс CObj, который является базовым по отношению к CPos. Но, учитывая, что класс CPos является базовым для классов CLine, CRect и CEllipse, то класс CObj оказывается общим для всех них. Таким образом, получаем иерархию объектов, показанную на рис. 6.2.

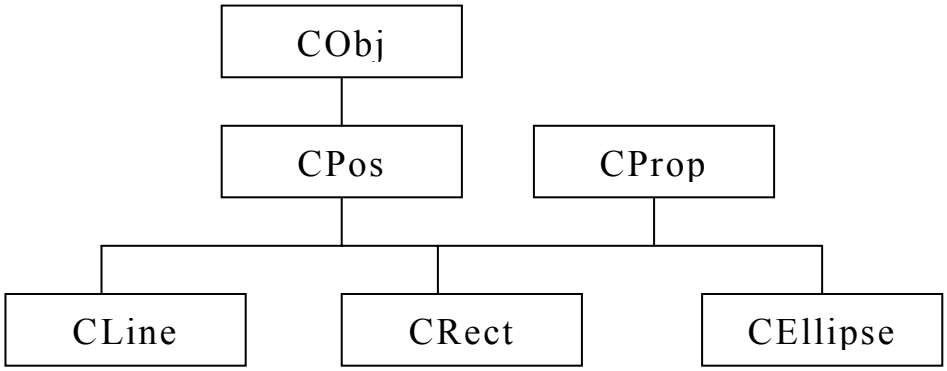

Рис. 6.2. Иерархия объектов программы листинга 6.1

Благодаря введению класса CObj в иерархию объектов, классы CLine, CRect и CEllipse можно рассматривать с единых позиций: как объект CObj. Это значит, что любой указатель на эти три класса можно присвоить указателю типа CObj без операции приведения типа, и наоборот, имея указатель типа CObj на один из трех объектов, можно его преобразовать в указатель на конкретный объект CLine, CRect или CEllipse. Однако чтобы выполнить такое преобразование, необходима операция приведения типов, которая в представленной программе выглядит следующим образом:

```
CLine* line = (CLine *)obj[i];
```

Именно поэтому в цикле for необходимо использовать оператор switch для определения типа объекта. При этом тип объекта хранится в переменной type_obj класса CObj и означивается в момент создания экземпляров графических объектов в соответствующих конструкторах. Такой прием программирования позволяет всегда иметь достоверную информацию о типе используемого объекта на разных уровнях иерархии классов. Кроме того, введение класса CObj позволяет создавать удобное однотипное хранилище для разнотипных объектов CLine, CRect и CEllipse, которое в программе представлено в виде массива

```
CObj* obj[TOTAL_OBJ];
```

В заключение программы выполняется операция удаления созданных объектов. При этом следует удалять объекты графических примитивов, а не объекты типа CObj. Поэтому здесь также организуется цикл for, в котором определяется тип объекта, а затем используется оператор delete для его удаления.

Основным недостатком представленной программы является необходимость определения типа объекта перед началом работы с ним, что приводит к громоздкому тексту программы подобно структурному подходу к

программированию. Исправить этот недостаток можно путем использования виртуальных функций.

Идея виртуальных функций заключается в том, что они могут быть описаны в базовом классе, а их конкретные реализации в производных. Применительно к задаче работы с графическими рбъектами функцию Draw() целесообразно сделать виртуальной, которая будет задана в базовом классе по отношению к классам CLine, CRect и CEllipse, а реализации находиться в дочерних. В связи с этим, имеет смысл ввести еще один новый класс CObjView, отвечающий за рисование графических примитивов и очевидно, он должен находиться в иерархии объектов, как показано на рис. 6.3.

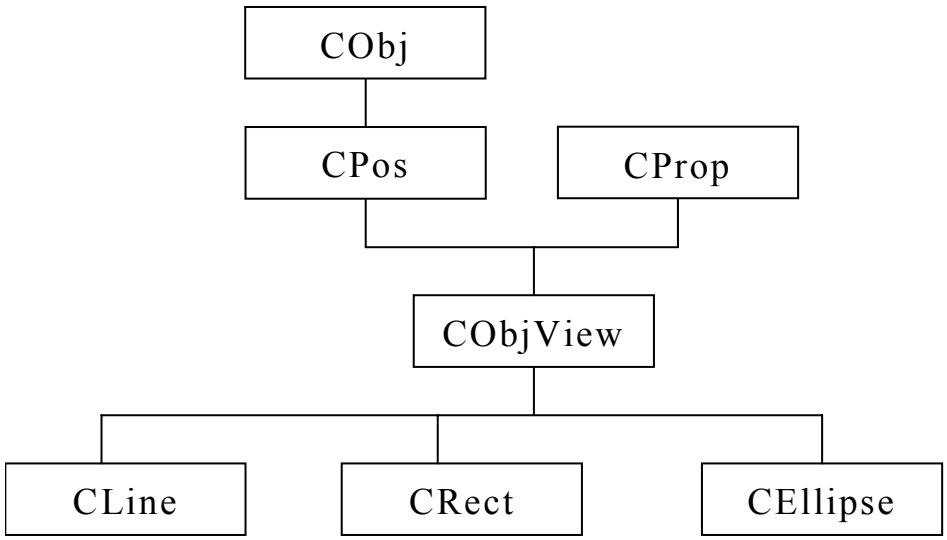

Рис. 6.3. Иерархия объектов с новыми классом CObjView

Для того чтобы задать виртуальную функцию используется ключевое слово virtual, за которым следует прототип функции. Таким образом, получаем следующее описание класса CObjView с виртуальной функцией Draw():

```
class CObjView : public CPos, public CProp
{
protected:
 CObjView() {}
 CObjView(int x1,int y1, int x2, int y2, int w, int clr) :
                          Pos(x1,y1,x2,y2), CProp(w,clr) {}
 ~CObjView() {}

public:
 virtual void Draw() {};
};
```

Затем следует описание класса CLine:

```
class CLine : public CObjView
{
public:
```

```
     CLine() {type_obj = OBJ_LINE;}
     CLine(int x1,int y1, int x2, int y2, int w, int clr) :
                              CObjView(x1,y1,x2,y2,w,clr)
                              {type_obj = OBJ_LINE;}

     ~CLine() {printf("Удаление объекта линия\n");}

     void Draw()
     {
          printf("Рисование линии с координатами (%d, %d) и (%d,
%d)\n", sp_x, sp_y, ep_x, ep_y);
     }
    };
```

и аналогично описываются классы CRect и CEllipse. Благодаря использованию виртуальной функции, фрагмент программы для отображения графических примитивов можно записать следующим образом:

```
    for(int i = 0;i < TOTAL_OBJ;i++)
    {
          CObjView* obj_v = (CObjView *)obj[i];
          obj_v->Draw();
    }
```

Цикл for здесь содержит всего две строки программы. Причем они останутся неизменными при любом числе графических объектов, что представляет большое удобство для программиста. Последний пример показывает как ООП позволяет заметно упростить текст программы по сравнению со структурным подходом к программированию.

6.5. Перегрузка операторов

Классы в языке C++ предоставляют возможность переопределять работу некоторых операторов таких как ‘=’, ‘+’, ‘*’, ’new’, ‘delete’, ‘==’, ‘>’, ‘<’ и др. Это дает возможность упрощать текст программы и повышать качество программирования. Преимущества от перегрузки операторов хорошо проявляются при работе со строками, которые в базовом языке C++ представляют собой массивы символов и операции присваивания, добавления и сравнения строк становятся сложными. Однако, используя классы можно сделать работу со строками подобно работе с обычными переменными.

Зададим класс string, в котором будут описаны алгоритмы обработки строковой информации:

```
    class string
    {
    public:
     string() {buff[0]='\0';}
     string(const char* str) {strcpy(buff,str);}
```

```
 ~string() {}

 char* GetString() {return buff;}
 void SetString(const char* str) {strcpy(buff,str);}

private:
 char buff[MAX_LENGTH];
};
```

В данном классе описано два конструктора для возможности создания пустой и некоторой начальной строк. Кроме того, задано два метода: GetString() и SetString(), через которые осуществляется доступ к массиву buff[], являющимся частным элементом класса. Представленный класс описывает минимальный набор функций для работы со строками. С помощью него можно лишь задать какую-либо строку и получить ее, например, для вывода на экран. Все возможные манипуляции со строками с помощью класса string демонстрирует текст программы, представленный ниже:

```
string str;
str.SetString("Hello World");
printf("%s",str.GetString());
```

При этом операции присваивания одной строки другой, добавления, сравнения строк между собой и т.п. остаются нереализованными. Формально для выполнения этих операций можно задать набор функций в классе string, но ими будет менее удобно пользоваться, чем общеизвестными операторами ‘=’, ‘+’и ‘==’. Для того чтобы сказать, что должен делать конкретный оператор при работе с классом string, его нужно перегрузить, т.е. определить алгоритм, который будет выполняться при реализации того или иного оператора.

Перегрузка оператора выполняется с помощью ключевого слова operator, за которым следует символ оператора, затем в круглых скобках аргумент, т.е. то, что предполагается будет стоять справа от оператора. После этого в бигурных скобках задается алгоритм, который будет выполняться при вызове перегруженного оператора. Пример перегрузки оператора ‘=’ для присваивания строки показан ниже:

```
class string
{
public:
 string() {buff[0]='\0';}
 string(const char* str) {strcpy(buff,str);}
 ~string() {}

 char* GetString() {return buff;}
 void SetString(const char* str) {strcpy(buff,str);}

 void operator = (const char* str) {strcpy(buff,str);}
```

```
private:
 char buff[MAX_LENGTH];
};
```

Обратите внимание, что перед оператором ставится возвращаемый им тип, в данном случае void. Возвращаемый тип необходим при реализации, например, условных операторов, о которых речь пойдет ниже.

После перегрузки оператора присваивания с классом string становятся возможны следующие действия:

```
string str;
str = “Hello World”;
```

Здесь при выполнении операции присваивания активизируется алгоритм, записанный в теле оператора, а именно функция strcpy(), в которой указатель str указывает на строку символов, стоящих справа от оператора присваивания. Таким образом, происходит копирование переданной строки в массив buff класса string.

Следует также отметить, что описанное действие оператора присваивания распространяется только на класс string. Это значит, что при выполнении, например, операций над числами:

```
int a = 10;
int b = a;
```

оператор присваивания будет работать корректно, обычным образом.

Особенностью перегрузки данного оператора является то, что он будет работать только в том случае, когда его правый аргумент является массивом символов и не будет выполняться в случае присваивания одного класса string другому:

```
string str, dst(“Hello”);
str = dst; //работать не будет
```

Вместе с тем такая запись вполне логична и естественна и желательно чтобы ее можно было использовать наряду с простыми массивами. Это достигается путем добавления в класс string еще одной реализации перегрузки оператора присваивания, следующим образом:

```
class string
{
public:
 string() {buff[0]='\0';}
 string(const char* str) {strcpy(buff,str);}
 ~string() {}

 char* GetString() {return buff;}
```

```
 void SetString(const char* str) {strcpy(buff,str);}

 void operator = (const char* str) {strcpy(buff,str);}
 void operator = (string& str) {strcpy(buff,str.GetString());}

private:
 char buff[MAX_LENGTH];
};
```

В итоге, при реализации алгоритма

```
string str, dst("Hello");
str = dst;
```

будет вызываться второй вариант перегрузки оператора присваивания. Причем компилятор сам определит, какой из вариантов перегрузки вызывать в каждом конкретном случае.

В представленном примере перегрузки аргументом является ссылка на класс string, а не представитель этого класса. Это сделано, для того чтобы при реализации оператора присваивания не происходило копирование класса string при передаче аргумента и, таким образом, экономилась память ЭВМ и увеличивалась скорость работы. Поэтому, формально, программа будет работать, если вместо ссылки использовать представитель класса.

Следующим шагом оптимизации работы со строками выполним перегрузку оператора добавления одной строки другой, которая будет определяться символом '+'. Для этого, по аналогии, добавим в класс string следующие строки:

```
void operator + (const char* str) {strcat(buff,str);}
void operator + (string& str) {strcat(buff, str.GetString();}
```

который возможно использовать только таким образом:

```
string str;
str + "Hello";
```

Это несколько непривычная запись оператора добавления строки. Было бы правильнее использовать запись вида

```
string str;
str = str + "Hello";
```

но здесь используется сразу два оператора: присваивания и добавления. Поэтому для реализации такой конструкции необходимо, чтобы оператор добавления '+' возвращал сформирванную строку оператору присваивания. Следовательно, описание оператора добавления в классе string должно выглядеть так:

```
char* operator + (const char* str) {return strcat(buff,str);}
char* operator + (string& str)
{
      return strcat(buff, str.GetString();
}
```

Наконец, последним шагом выполним перегрузку условного оператора сравнения двух строк между собой. Реализация этого оператора очевидна и выглядит следующим образом:

```
bool operator == (const char* str)
{
 if(strcmp(str,buff) == 0) return true;
 else return false;
}

bool operator == (string str)
{
 if(strcmp(str.GetString(),buff) == 0) return true;
 else return false;
}
```

В результате получаем следующее описание класса string:

```
class string
{
public:
 string() {buff[0]='\0';}
 string(const char* str) {strcpy(buff,str);}
 ~string() {}

 char* GetString() {return buff;}
 void SetString(const char* str) {strcpy(buff,str);}

 void operator = (const char* str) {strcpy(buff,str);}
 void operator = (string& str) {strcpy(buff,str.GetString());}

 char* operator + (const char* str) {return strcat(buff,str);}
 char* operator + (string& str)
 {
          return strcat(buff, str.GetString();
 }

 bool operator == (const char* str)
 {
      if(strcmp(str,buff) == 0) return true;
      else return false;
 }
```

```
 bool operator == (string str)
 {
      if(strcmp(str.GetString(),buff) == 0) return true;
      else return false;
 }

private:
 char buff[MAX_LENGTH];
};
```

который можно использовать следующим образом:

```
int main()
{
 string str;
 str = "Hello World";
 string dst;
 dst = dst + str;

 if(str == dst) printf("src is equal dst");
 else printf(src is'nt equal dst");

 return 0;
}
```

Контрольные вопросы и задания

1. В чем основные отличия класса от структуры?
2. Дайте понятие наследования классов.
3. Опишите класс для хранения имени, места работы и возраста сотрудника с двумя конструкторами: без аргументов и с аргументами для инициализации указанных полей.
4. Какими способами можно создавать экземпляры классов?
5. Дайте понятие полиморфизма.
6. В какой последовательности вызываются конструкторы базовых классов при создании экземпляра дочернего класса?
7. При каком режиме доступа возможно обращение ко всем элементам класса?
8. Придумайте и запишите какой-либо метод класса для задания значений его частным элементам.
9. Каким образом выполняется наследование классов в C++?
10. Как задается описание функции класса за его пределами?
11. В чем особенность режима доступа protected и чем он отличается от режима private?
12. Дайте понятие множественного наследования.
13. Что такое дружественные функции и для чего они предназначены?
14. Как задаются виртуальные функции класса?

15. Запишите двухуровневую иерархию для описания объема хранимых денежных средств в разной валюте и в базовом классе реализуйте виртуальную функцию для вывода доступных средств в соответствующих денежных единицах.

16. Поясните, что понимается под перегрузкой операторов.

17. Запишите класс для работы с комплексными числами, используя механизм перегрузки операторов.