Учреждение образования
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Н. В. Пацей

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

Рекомендовано
учебно-методическим объединением высших учебных заведений
Республики Беларусь по образованию в области информатики
и радиоэлектроники в качестве учебно-методического пособия
для студентов учреждений, обеспечивающих получение
высшего образования по направлению специальности
«Информационные системы и технологии
(издательско-полиграфический комплекс)»

Минск 2010

УДК 004.4(076.5)
ББК 22.18я7
П21

Рецензенты:
кандидат технических наук, доцент кафедры электронных вычислительных машин Белорусского государственного университета информатики и радиоэлектроники *Д. И. Самаль*;
кафедра экономико-математических методов управления Института управленческих кадров Академии управления при Президенте Республики Беларусь (зав. кафедрой кандидат физико-математических наук *Б. В. Новыш*)

Пацей, Н. В.

П21 Основы алгоритмизации и программирования : учеб.-метод. пособие для студентов специальности «Информационные системы и технологии (издательско-полиграфический комплекс)» / Н. В. Пацей . – Минск : БГТУ, 2010. – 289 с.

ISBN 978-985-434-989-3.

Учебно-методическое пособие является введением в алгоритмизацию и программирование на языке C/C++. В нем описана технология разработки и отладки программ на C++ в среде Microsoft Visual Studio, методы построения и анализа алгоритмов, кратко рассмотрены системы счисления и основы языка блок-схем, изложены и систематизированы базовые элементы языка C/C++ в объеме, позволяющем составлять программы как для простых, так и достаточно сложных задач. Приведены практические задания для закрепления навыков разработки алгоритмов и программ. Материал сопровождается рисунками и фрагментами листингов программ.

Предназначено для студентов специальности «Информационные системы и технологии (издательско-полиграфический комплекс)», а также для студентов, начинающих изучать алгоритмизацию и программирование.

УДК 004.4(076.5)
ББК 22.18я7

ISBN 978-985-434-989-3

ПРЕДИСЛОВИЕ

Цель данного учебно-методического пособия – помочь студенту в изучении основ алгоритмизации и приобретении базовых навыков программирования. Программирование алгоритмов и задач рассматривается на процедурно-ориентированном языке C/C++.

Язык программирования C/C++ универсальный, он позволяет разрабатывать программы в соответствии с разными парадигмами: процедурным программированием, объектно-ориентированным, параметрическим.

Изложение материала начинается с обзоров вводных тем: системы счисления, понятие алгоритма и способы его представления, описание среды разработки MS Visual Studio 2008 и со справочных данных языка C/C++: ключевые слова, базовые типы данных, общие принципы объявления и использования переменных и констант, основные операции языка, операторы ввода-вывода.

Далее рассматриваются темы, составляющие «классику» программирования: арифметические (вычислительные) задачи, итерационные циклы, алгоритмы и технологии обработки текста, массивов, работа с динамической памятью, файлами и структурами.

В заключение описаны абстрактные типы данных и алгоритмы, которые считаются фундаментом современного компьютерного программирования. Они являются полезным инструментом при разработке программ независимо от применяемого языка программирования.

Так как пособие рассчитано на начинающих программистов, обработка ошибок опущена, а изложение некоторых алгоритмов и их реализация упрощены.

Для более глубокого и полного изучения вопросов алгоритмизации и программирования следует обратиться к списку рекомендуемой литературы.

При изучении данной дисциплины рекомендуется пользоваться локальным или сетевым MSDN (Microsoft Development Network).

Все главы содержат большое количество листингов фрагментов программ. При реализации алгоритмов не преследовалась цель получить максимально эффективный код. Основной упор делался на корректность и доступность реализации.

Практически каждая из глав заканчивается заданиями для выполнения, некоторые задания связаны между собой, поэтому рекомендуется выполнять их последовательно.

Глава 1. СИСТЕМЫ СЧИСЛЕНИЯ И КОДЫ ЧИСЕЛ

1.1. Понятие системы счисления

Системы счисления (с/с) делятся на позиционные и непозиционные. В *непозиционных* системах вес цифры (т.е. тот вклад, который она вносит в значение числа) не зависит от ее позиции в записи числа. Так, в римской системе счисления в числе XXXII (тридцать два) вес цифры X в любой позиции равен десяти.

В *позиционных* системах счисления вес каждой цифры изменяется в зависимости от ее положения (позиции) в последовательности цифр, изображающих число. Например, в числе 757,7 первая семерка означает 7 сотен, вторая – 7 единиц, а третья – 7 десятых долей единицы. Сама же запись числа 757,7 означает сокращенную запись выражения $700 + 50 + 7 + 0{,}7 = 7 \cdot 10^2 + 5 \cdot 10^1 + 7 \cdot 10^0 + 7 \cdot 10^{-1} = 757{,}7$.

Основание позиционной с/с – это количество различных знаков или символов, используемых для изображения цифр в данной системе. За основание системы можно принять любое натуральное число. Следовательно, возможно бесчисленное множество позиционных систем: двоичная, троичная, четверичная и т.д. Запись чисел в каждой из систем счисления с основанием q означает сокращенную запись выражения $a_{n-1} \cdot q^{n-1} + a_{n-2} \cdot q^{n-2} + \ldots + a_1 \cdot q^1 + a_0 \cdot q^0 + a_{-1} \cdot q^{-1} + \ldots + a_{-m} \cdot q^{-m}$, где n и m – число целых и дробных разрядов соответственно.

Основание с/с соответствует количеству цифр (знаков), используемых для записи чисел в этой с/с. Например, основание десятичной с/с есть число 10, и только десять цифр (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) используются при записи чисел в этой с/с. В двоичной с/с используются две цифры – 0 и 1, в шестнадцатеричной – 16, причем для чисел 10, 11, 12, 13, 14, 15 в этой с/с введены дополнительные цифры (знаки) – A, B, C, D, E, F соответственно, т.к. традиционно используемых цифр недостаточно.

Основание 10 не слишком удобно (в цепях электрических схем необходимо для этого иметь 10 различных сигналов). С технической точки зрения, чем меньше сигналов в схеме, тем лучше. Наименьшее основание, которое может быть у позиционной с/с – это 2. Поэтому двоичная с/с широко используется в современной вычислительной технике, в устройствах автоматики и связи.

Кроме десятичной с/с для «общения с компьютером» широко используются системы с основанием, являющимся целой степенью числа 2, а именно двоичная, восьмеричная и шестнадцатеричная.

Полезно запомнить запись в этих системах счисления первых двух десятков целых чисел (табл. 1.1).

Таблица 1.1

Представление чисел в с/с

Десятичная с/с	Двоичная с/с	Восьмеричая с/с	Шестнадцатиричная с/с
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
17	10001	21	11

Как видно из таблицы, недостатком двоичной с/с является быстрый рост числа разрядов, необходимых для записи чисел. В восьмеричной и шестнадцатеричной с/с требуется соответственно в три (восьмеричная) и в четыре (шестнадцатеричная) раза меньше разрядов, чем в двоичной системе (ведь числа 8 и 16 – третья и четвертая степени числа 2 соответственно).

1.2. Перевод чисел

Перевод из 8 с/с и 16 с/с в 2 с/с

Для перевода достаточно каждую цифру числа заменить эквивалентной ей двоичной триадой (тройкой цифр) или тетрадой (четверкой цифр).

Пример

$502_{(8\ с/с)} \rightarrow ?_{(2\ с/с)}$

$502_{(8\ с/с)} = \underline{101}\ \underline{000}\ \underline{010}_{(2\ с/с)}$

Перевод из 2 с/с в 8 с/с и 16 с/с

Для перевода нужно разбить число влево и вправо от последнего разряда (или запятой) на триады или тетрады и каждую такую группу заменить соответствующей восьмеричной или шестнадцатеричной цифрой. В случае необходимости неполные триады дополняются нулями.

Пример

$1111110_{(2\ с/с)} \rightarrow ?_{(8\ с/с)}$

$\underline{1}\ \underline{111}\ \underline{110}_{(2\ с/с)} = \underline{001}\ \underline{111}\ \underline{110}_{(2\ с/с)} = 176_{(8\ с/с)}$

$1111010101{,}1100_{(2\ с/с)} \rightarrow ?_{(16\ с/с)}$

$\underline{0011}\ \underline{1101}\ \underline{0101},\ \underline{1100}_{(2\ с/с)} = 3D5{,}C_{(16\ с/с)}$

Перевод из произвольной в 10 с/с, и наоборот

Пусть имеется с/с с основанием k и некоторое число $a_1 \ldots a_n$ в этой с/с, где $a_1, \ldots, a_n$ – цифры этого числа. Данное число можно представить в виде $a_1 \cdot k^{n-1} + a_2 \cdot k^{n-2} + \ldots + a_n \cdot k^0$.

Пример

$110011_{(2\ с/с)} \rightarrow ?_{(10\ с/с)}$

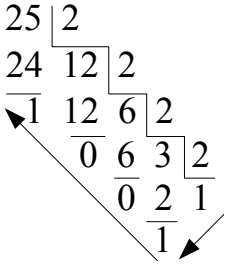
$110011_{(2\ с/с)} = 1 \cdot 10^{101} + 1 \cdot 10^{100} + 0 \cdot 10^{11} + 0 \cdot 10^{10} + 1 \cdot 10^{1} + 1 \cdot 10^{0}{}_{(2\ с/с)} = 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0{}_{(10\ с/с)} = 32 + 16 + 2 + 1 = 51_{(10\ с/с)}$

$1216{,}04_{(8\ с/с)} \rightarrow ?_{(10\ с/с)}$

$1216{,}04_{(8\ с/с)} = 1 \cdot 8^3 + 2 \cdot 8^2 + 1 \cdot 8^1 + 6 \cdot 8^0 + 4 \cdot 8^{-2} = 512 + 128 + 8 + 6 + 0{,}0625 = 654{,}0625_{(10\ с/с)}$

Перевод из 10 с/с в произвольную

Данный алгоритм является обратным к алгоритму, рассмотренному выше. Исходное число делится на основание с/с, в которую требуется перевести число. Первый шаг: разделить исходное число на r (основание новой с/с), зафиксировать остаток от деления (число от 0 до $r - 1$) и частное. Второй шаг: если частное больше r, то снова разделить его на r, продолжая фиксировать остаток от деления. Процесс деления частных продолжать до тех пор, пока частное не станет меньше r. Третий шаг: все полученные в процессе деления остатки от деления и последнее частное будут образовывать цифры исходного числа в с/с с основанием r. Выписав все найденные цифры в обратном порядке (начиная с последнего частного) получим искомое представление числа в новой с/с.



Например, требуется $25_{(10\ с/с)}$ перевести в 2 с/с.

Согласно алгоритму получаем $11001_{(2\ с/с)}$.

Проверим результат: $25_{(10\ с/с)} = 11001_{(2\ с/с)} = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 25_{(10\ с/с)}$.

Перевод дробной части числа

Чтобы правильную дробь перевести из с/с с основанием r в с/с с основанием q необходимо последовательно умножать дробную часть (сначала самого числа, а потом получающихся произведений) на новое основание с/с q до тех пор, пока: либо дробная часть получаемого произведения не станет равна нулю; либо не будет достигнута нужная точность (заданное число цифр после запятой). В новой с/с число запишется в виде последовательности целых частей получаемых произведений, начиная с первого. *Важно помнить, что все действия производятся в исходной с/с*.

```
0| 3125
      8
-------
2| 5000
      8
-------
4| 0
```

При переводе смешанной дроби из одной с/с в другую отдельно по своим правилам переводится целая часть и отдельно дробная, а результаты затем приписываются друг к другу через точку.

Например, требуется перевести $0{,}3125_{(10\ с/с)}$ в 8 с/с. Согласно алгоритму перевода получаем: $0{,}3125_{(10\ с/с)} = 0{,}24_{(8\ с/с)}$.

1.3. Выполнение арифметических операций

При сложении необходимо помнить, в какой с/с введутся расчеты. Так, если получаем число два при сложении чисел в 2 с/с, то заменяем его на 10, т.к. цифры 2 в двоичной с/с нет. При выполнении арифметических операций в с/с с основанием r необходимо иметь соответствующие таблицы сложения и умножения. Ниже представлены таблицы сложения и умножения для $r = 2$.

+	**0**	**1**
0	0	1
1	1	10

×	**0**	**1**
0	0	0
1	0	1

Далее приведены таблицы сложения и умножения для $r = 8$.

+	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	10
2	2	3	4	5	6	7	10	11
3	3	4	5	6	7	10	11	12
4	4	5	6	7	10	11	12	13
5	5	6	7	10	11	12	13	14
6	6	7	10	11	12	13	14	15
7	7	10	11	12	13	14	15	16

×	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	10	12	14	16
3	0	3	6	11	14	17	22	25
4	0	4	10	14	20	24	30	34
5	0	5	12	17	24	31	36	43
6	0	6	14	22	30	36	44	52
7	0	7	16	25	34	43	52	61

Таблицы сложения и умножения для $r = 16$ будут выглядеть следующим образом.

+	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

×	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	0	2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	1E
3	0	3	6	9	C	F	12	15	18	1B	1E	21	24	27	2A	2D
4	0	4	8	C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	0	5	A	F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	0	6	C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	0	7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	0	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	0	9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	0	A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	0	B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	0	C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	0	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	0	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	0	F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

Пример

$10000000100_{(2\ с/с)} + 111000010_{(2\ с/с)} = 10111000110_{(2\ с/с)}$

$3B3{,}6_{(16\ с/с)} + 38B{,}4_{(16\ с/с)} = 73E{,}A_{(16\ с/с)}$

$1510{,}2_{(8\ с/с)} - 1230{,}54_{(8\ с/с)} = 257{,}44_{(8\ с/с)}$

$100111_{(2\ с/с)} \times 1000111_{(2\ с/с)} = 101011010001_{(2\ с/с)}$

$1170{,}64_{(8\ с/с)} \times 46{,}3_{(8\ с/с)} = 57334{,}134_{(8\ с/с)}$

1.4. Коды чисел

В целях упрощения выполнения арифметических операций в компьютерных системах применяют специальные коды для представления чисел. Использование кодов позволяет свести операцию вычитания чисел к арифметическому сложению кодов этих чисел. Применяются *прямой*, *обратный* и *дополнительный* коды чисел.

Прямой код используется для представления отрицательных чисел, а также при умножении и делении. Обратный и дополнительный коды используются для замены операции вычитания операцией сложения, что упрощает устройство арифметического блока. К кодам выдвигаются следующие требования:

1) разряды числа в коде жестко связаны с определенной разрядной сеткой;

2) для записи кода знака в разрядной сетке отводится фиксированный, строго определенный разряд.

Например, если за основу представления кода взят один байт, то для представления числа будет отведено 7 разрядов, а для записи кода знака – один разряд.

Прямой код двоичного числа совпадает по изображению с записью самого числа. Значение знакового разряда для положительных чисел равно 0, а для отрицательных чисел – 1.

Знаковым разрядом обычно является крайний разряд в разрядной сетке. В дальнейшем при записи кода знаковый разряд от цифровых условимся отделять запятой. Если количество разрядов кода не указано, будем предполагать, что под запись кода выделен один байт.

Обратный код для положительного числа совпадает с прямым кодом. Для отрицательного числа все цифры числа заменяются на противоположные (1 на 0, 0 на 1), а в знаковый разряд заносится единица.

Дополнительный код положительного числа совпадает с прямым кодом. Для отрицательного числа дополнительный код образуется путем получения обратного кода и добавлением к младшему разряду единицы.

Пример

Для числа +1101:

прямой код	обратный код	дополнительный код
0,0001101	0,0001101	0,0001101

Для числа –1101:

прямой код	обратный код	дополнительный код
1,0001101	1,1110010	1,1110011

Особенности сложения чисел
в обратном и дополнительном кодах

При сложении чисел в дополнительном коде возникающая единица переноса в знаковом разряде отбрасывается. При сложении чисел в обратном коде возникающая единица переноса в знаковом разряде прибавляется к младшему разряду суммы кодов. Если результат арифметических действий является кодом отрицательного числа, необходимо преобразовать его в прямой код. При этом обратный код преобразуется в прямой заменой цифр во всех разрядах, кроме знакового, на противоположные. Дополнительный код преобразуется в прямой так же, как и обратный, с последующим прибавлением единицы к младшему разряду.

Пример

Сложим двоичные числа X и Y в обратном и дополнительном кодах. Если X = −101, Y = −11, по правилами двоичной арифметики получим:

X = −101;

Y = −110;

X + Y = −1011.

Сложим числа, используя коды.

Прямой код	Сложение в обратном коде	Сложение в дополнительном коде
$X_{пр} = 1{,}0000101$	$X_{обр} = 1{,}1111010$	$X_{доп} = 1{,}1111011$
$Y_{пр} = 1{,}0000110$	$Y_{обр} = 1{,}1111001$	$Y_{доп} = 1{,}1111010$
	$1\ 1{,}1110011$	$1\ )\ 1{,}1110101$
	$\longrightarrow +1$	отбрасывается ↙
	$(X + Y)_{обр} = 1{,}1110100$	$(X + Y)_{доп} = 1{,}1110101$

1.5. Практические задания

1. Перевести числа в десятичную систему счисления: $29A{,}5_{(16\ с/с)}$, $1216{,}04_{(8\ с/с)}$.

2. Перевести числа из десятичной системы счисления в двоичную: а) $464_{(10\ с/с)}$; б) $380{,}1875_{(10\ с/с)}$; в) $115{,}94_{(10\ с/с)}$ (получить пять знаков после запятой в двоичном представлении).

3. Записать число в прямом, обратном и дополнительном кодах: а) 11010; б) –11101; в) –101001; г) –1001110.

4. Перевести X и Y в прямой, обратный и дополнительный коды. Сложить их в обратном и дополнительном кодах. Результат перевести в прямой код. Проверить полученный результат, пользуясь правилами двоичной арифметики.

а) X = −11010; Y = 1001111; г) X = −10110 Y = −111011;
б) X = −11101; Y = −100110; д) X = 1111011; Y = −1001010;
в) X = 1110100; Y = −101101; е) X = −11011; Y = −10101.

Глава 2. ПОНЯТИЕ АЛГОРИТМА И СПОСОБЫ ЕГО ОПИСАНИЯ

2.1. Этапы решения задачи

Можно выделить следующие основные этапы решения задачи:

- постановка (формулировка) задачи;
- построение модели, выбор метода решения задачи;
- разработка алгоритма;
- проверка правильности алгоритма;
- реализация алгоритма;
- анализ алгоритма и его сложности;
- отладка программы, обнаружение, локализация и устранение возможных ошибок;
- получение результата;
- составление документации.

Понятие алгоритма занимает центральное место в вычислительной математике и программировании. Справедливо следующее определение. *Алгоритм* – строгая и четкая система правил, определяющая последовательность действий над некоторыми объектами и после конечного числа шагов приводящая к достижению поставленной цели.

Прежде чем понять задачу, необходимо ее четко сформулировать. Это условие не является достаточным для понимания задачи, но оно абсолютно необходимо. Следующий важный шаг в решении задачи – формулировка для нее математической модели. Выбор модели и реализация алгоритма в значительной степени может повлиять на эффективность алгоритма решения задачи. Все перечисленные выше этапы нельзя рассматривать независимо друг от друга. В особенности первые три сильно влияют на последующие.

Наиболее распространенная процедура доказательства правильности программы (следующий этап) – это прогон ее на ранних тестах. Однако это не исключает все сомнения. Необходимо доказательство конечности алгоритма, при котором будут проверены все подходящие входные данные и получены все подходящие выходные данные.

Реализация алгоритма – процесс корректного преобразования алгоритма в машинную программу. Требует также построения целой системы структур данных для представления модели.

Задача анализа алгоритма и его сложности – получение оценок или границ для объема памяти или времени работы алгоритма. Полный анализ способен выявить узкие места в программах. Критерии оценок алгоритмов будут рассмотрены далее.

Эксплуатации программы предшествует отладка – исправление синтаксических и логических ошибок. Процесс проверки программы включает экспериментальное подтверждение того факта, что программа делает именно то, что должна делать. Обычно множество входов огромно, и полная проверка невозможна. Необходимо выбрать множество вводов, которые проверяют каждый участок программы. Программы следует тестировать также для того, чтобы определить их вычислительные ограничения.

Этап документации не является последним шагом в процессе построения алгоритма. Он должен переплетаться со всем процессом построения алгоритма для того, чтобы была возможность понять программы, написанные другими.

Последние этапы обеспечивают обратную связь, которая может заставить пересмотреть некоторые из предшествующих этапов.

2.2. Свойства алгоритма

Характерными свойствами алгоритма являются определенность, массовость и результативность.

Определенность (детерминированность) алгоритма предполагает такое составление предписания, которое не оставляет места для различных толкований или искажений результата, т.е. последовательность действий алгоритма строго и точно определена.

Массовость определяет возможность использования любых исходных данных из некоторого допустимого множества. Правило, сформулированное только для данного случая, не является алгоритмом (например, таблица умножения не является алгоритмом, а правило умножения «столбиком» есть алгоритм).

Результативность (конечность) алгоритма означает, что при любом допустимом исходном наборе данных алгоритм закончит свою работу за конечное число шагов.

2.3. Классификация алгоритмов

По типу используемого вычислительного процесса различают линейные (прямые), разветвляющиеся и циклические алгоритмы.

Линейные алгоритмы описывают линейный вычислительный процесс, этапы которого выполняются однократно и последовательно один за другим.

Разветвляющийся алгоритм описывает вычислительный процесс, реализация которого происходит по одному из нескольких заранее

предусмотренных направлений. Направления, по которым может следовать вычислительный процесс, называются ветвями. Выбор конкретной ветви вычисления зависит от результатов проверки выполнения некоторого логического условия. Результатами проверки являются: «истина» (да), если условие выполняется, и «ложь» (нет), при невыполнении условия.

Циклический алгоритм описывает вычислительный процесс, этапы которого повторяются многократно. Различают простые циклы, не содержащие внутри себя других циклов, и сложные (вложенные), содержащие несколько циклов. В зависимости от местоположения условия выполнения цикла различают циклы с предусловием и циклы с постусловием. В соответствии с видом условия выполнения циклы делятся на циклы с параметром и итерационные циклы.

Многие из реально существующих алгоритмов имеют смешанный характер, т.е. могут содержать линейные участки, разветвления, циклы с известным количеством повторений и итерационные циклы. В связи с этим составление алгоритмов сразу в законченной форме затруднено. Поэтому для составления сложных алгоритмов рекомендуется использовать нисходящее проектирование программ (метод пошаговой детализации, метод последовательных уточнений). Его суть: первоначально продумывается общая структура алгоритма, без детальной проработки его отдельных частей. Далее прорабатываются отдельные блоки, не детализированные на предыдущем шаге. Таким образом, на каждом шаге разработки уточняется реализация фрагмента алгоритма, т.е. решается более простая задача.

2.4. Способы описания алгоритмов

Существуют следующие способы описания алгоритмов:

1) запись на естественном языке (словесное описание);

2) изображение в виде схемы (графическое описание);

3) запись на алгоритмическом языке (составление программы).

Способы словесного описания алгоритмов отличаются применяемыми метаязыками (языки, предназначенные для описания языка программирования).

Например, словесное описание алгоритма решения квадратного уравнения $a \cdot x^2 + b \cdot x + c = 0$ будет выглядеть следующим образом:

1) $D := b^2 - 4 \cdot a \cdot c$;

2) если $D < 0$, идти к 4;

3) $x_1 := (-b + \sqrt{D}) / (2 \cdot a)$;
   $x_2 := (-b - \sqrt{D}) / (2 \cdot a)$;

4) Останов.

Основной недостаток словесного описания – плохая наглядность.

Графическое описание алгоритма – это представление алгоритма в виде схемы (двухмерного рисунка), состоящей из последовательности блоков (геометрических фигур), каждый из которых отображает содержание очередного шага алгоритма (управляющей структуры). Внутри фигур кратко записывают выполняемое действие. Такую схему называют блок-схемой алгоритма.

На рис. 2.1. приведены основные условные обозначения, используемые при графической записи алгоритма.

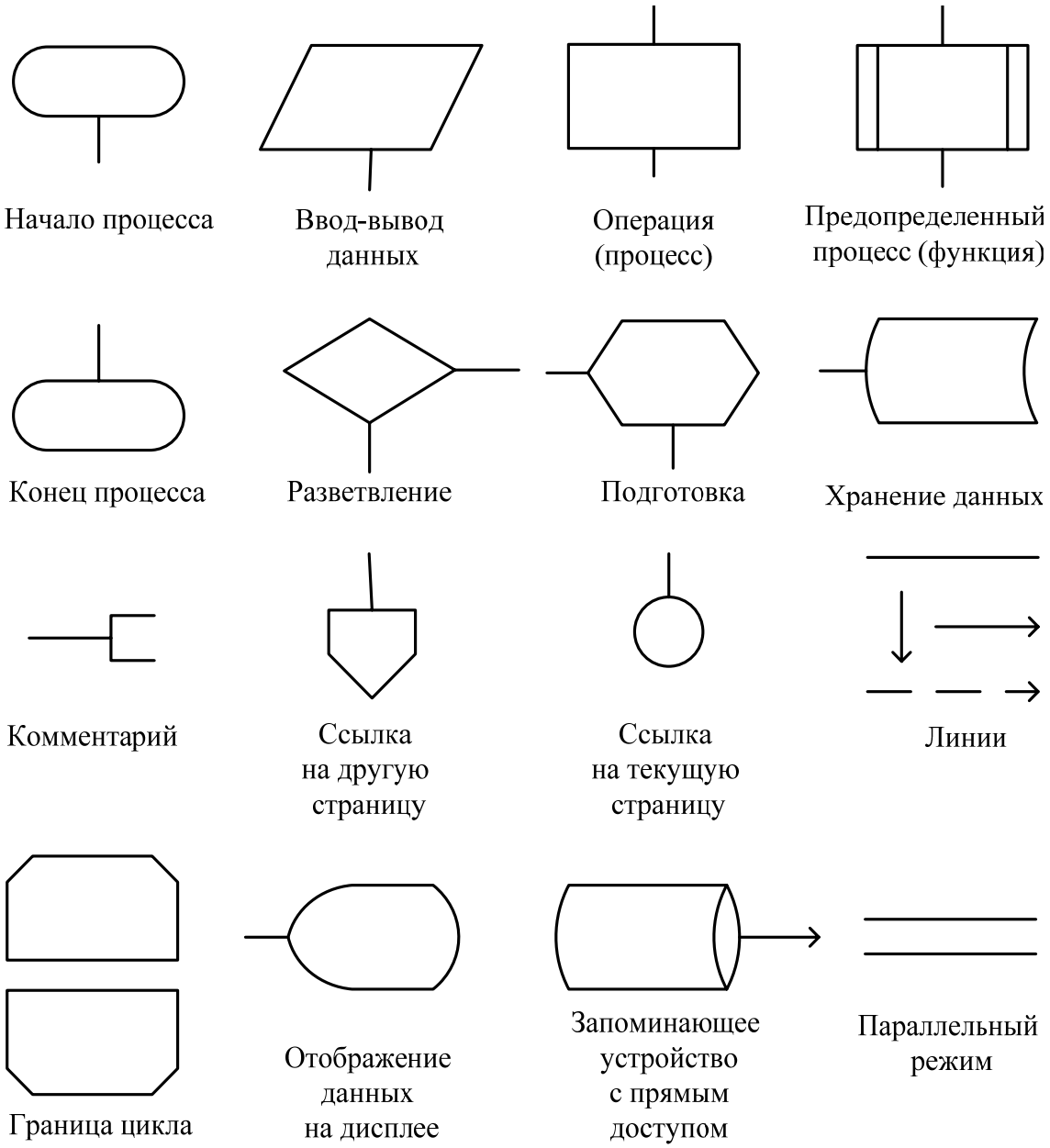


Рис. 2.1 Условные обозначения, используемые в блок-схемах

Для стандартизации и унификации языка схем алгоритмов в 1992 г. был принят ГОСТ 19.701–90 «Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения» [1]. В настоящее время данный стандарт продолжает действовать в Республике Беларусь.

На рис. 2.2 приведена блок-схема алгоритма нахождения максимального из трех заданных чисел, которая использует разветвляющийся алгоритм.

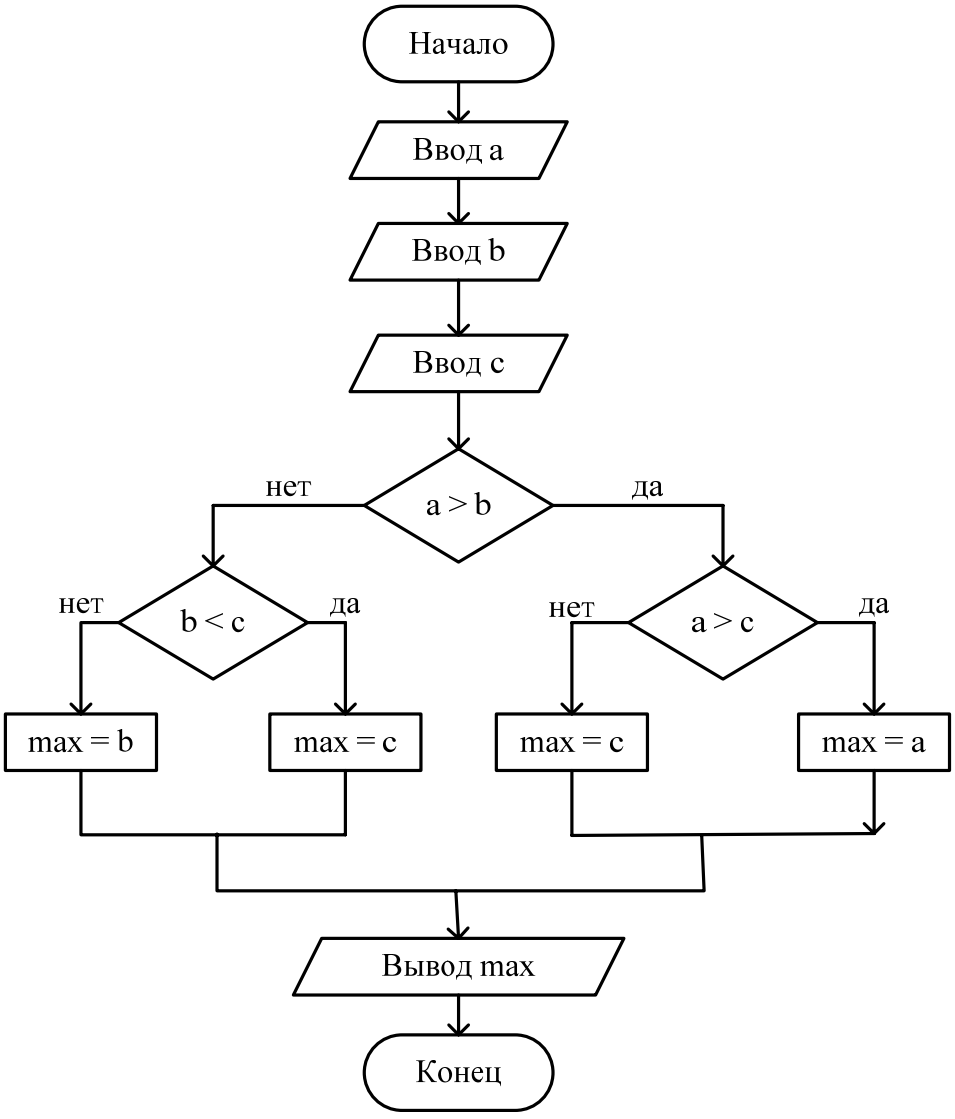


Рис. 2.2. Блок-схема алгоритма нахождения максимального числа

На рис. 2.3 приведена блок-схема алгоритма вычисления графика функции в заданном интервале, в которой используется циклический вычислительный процесс с известным числом повторений.

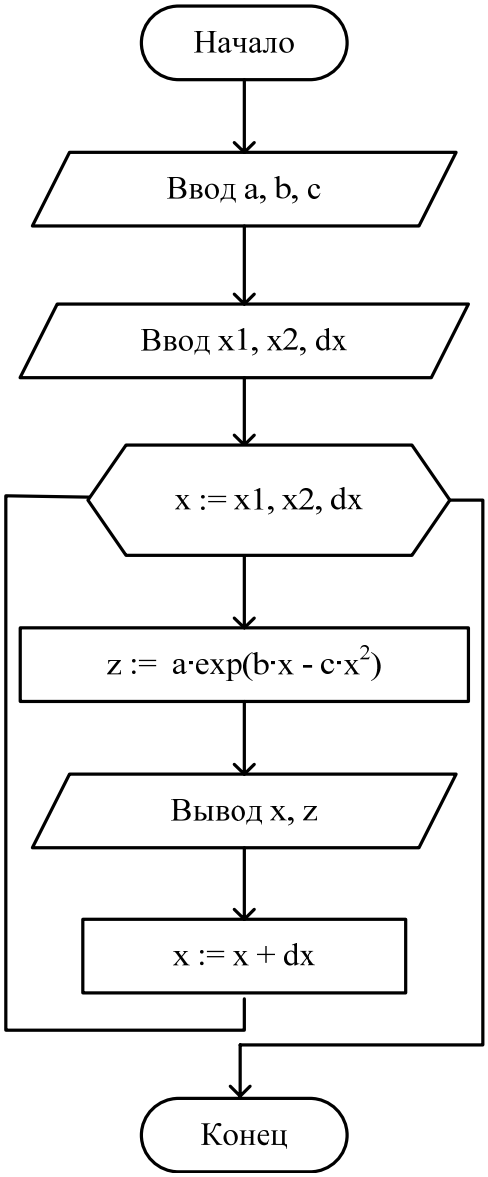


Рис. 2.3. Блок-схема алгоритма
вычисления графика функции в заданном интервале

Рассмотрим еще один пример, в котором используется цикл с неизвестным числом повторений. Необходимо вычислить значение функции $Y = \sin x$ через разложение функции в бесконечный ряд:

$$Y = \sin x = x - x^3 / 3! + x^5 / 5! - x^7 / 7! + \ldots .$$

Вычисления прекращаются, когда разность между модулями двух соседних слагаемых станет меньше величины $e = 0{,}0001$. Схема алгоритма решения данной задачи имеет вид, представленный на рис. 2.4. Как видно, здесь проверка условий окончания циклических вычислений осуществляется в конце цикла.

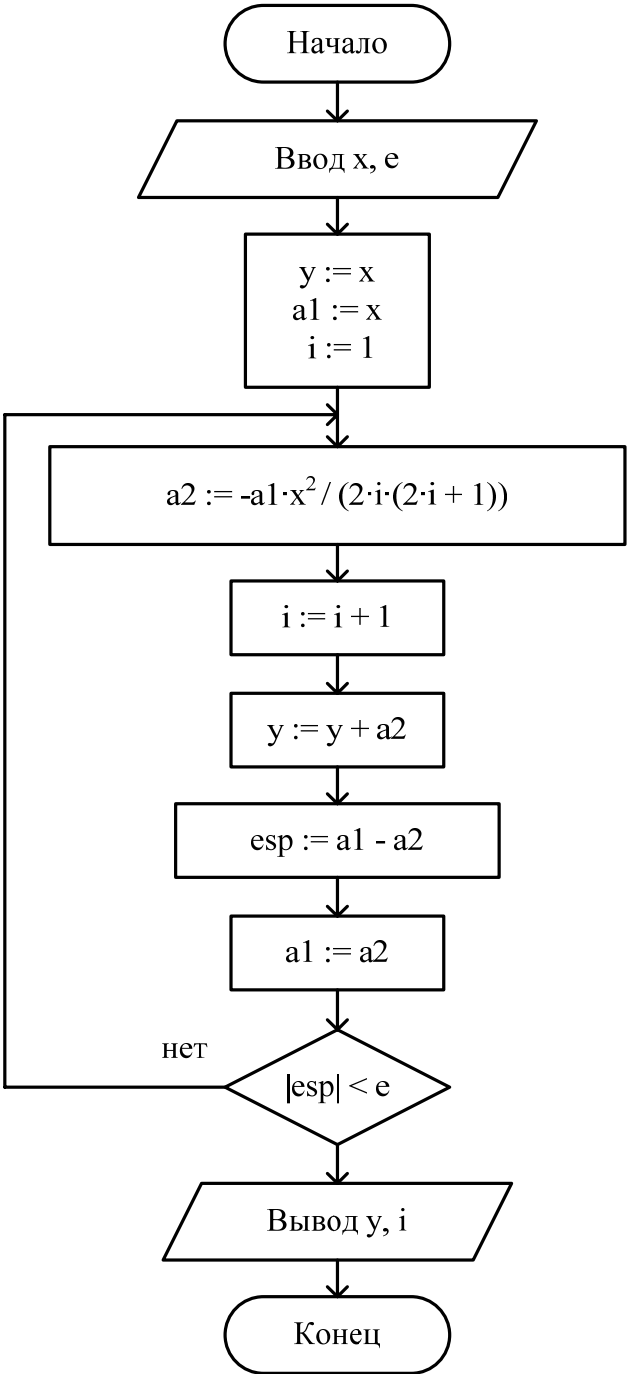


Рис. 2.4. Блок-схема алгоритма вычисления
значения функции с переменным количеством шагов

Понятия алгоритма и программы не имеют четкого разграничения. Так, программа, записанная на алгоритмическом языке – это окончательный вариант алгоритма решения задачи, ориентированный на конкретного исполнителя (компьютер или язык программирования).

2.5. Понятие структурного программирования

В 70-е годы прошлого века возник новый подход к разработке алгоритмов и программ, который получил название структурного проектирования программ. К его достоинствам можно отнести: более высокую производительность; читаемость программ; простоту тестирования и эффективность программ.

Одна из концепций структурного программирования – нисходящее проектирование. Это средство разбиения большой задачи на меньшие подзадачи так, чтобы каждую подзадачу можно было рассматривать независимо.

Все операции в программе, построенной на основе структурного программирования, должны представлять собой либо исполняемые в линейном порядке выражения, либо одну из следующих управляющих конструкций: вызовы подпрограмм; вложенные на произвольную глубину операторы условия; циклические операторы.

2.6. Практические задания

1. Составить алгоритм и блок-схему вычисления вещественной функции $ae^{(bx - cx)}$ в заданном интервале с заданным шагом.

2. Изобразить алгоритм и блок-схему нахождения минимального (максимального) числа из 4 заданных.

3. Составить алгоритм и блок-схему вычисления факториала.

4. Представить алгоритм и блок-схему ввода в массив статической последовательности неизвестной длины.

5. Составить алгоритм и блок-схему нахождения суммы чисел от 1 до заданного числа n.

Глава 3. ИНТЕГРИРОВАННАЯ СРЕДА РАЗРАБОТКИ MICROSOFT VISUAL C++

3.1. Введение

Минимальная адресованная ячейка памяти состоит из 8 двоичных позиций. В каждую позицию может быть записан либо 0, либо 1. В одной ячейке из 8 двоичных разрядов помещается объем информации в 1 байт, поэтому объем памяти принято оценивать количеством байт (1 024 байт = 1 Кб, 1 048 576 байт = 1 Мб).

При размещении данных производится их запись с помощью нулей и единиц – кодирование, при котором каждый символ заменяется последовательностью из 8 двоичных разрядов в соответствии со стандартной кодовой таблицей ASCII (прил. 1). Например, D (код 68) → 01000100; F (код 70) → 00100110; 4 (код 52) → 00110100.

Программа – это последовательность команд (инструкций), которые помещаются в памяти и выполняются процессором в указанном порядке. Программа записывается на языке высокого уровня, наиболее удобном для реализации алгоритма решения определенного класса задач. Для автоматизации процесса разработки программ используются интегрированные среды разработки (IDE – Integrated Development Environment). Обычно они содержат: входной язык системы; транслятор с входного языка на машинный язык; редактор связей; библиотеки программ; средства отладки; обслуживающие (сервисные) программы и документацию.

Исходный текст программы, введенный с помощью клавиатуры в память компьютера называется *исходный модуль* (*Source module*) (в C++ файл имеет расширение *.*cpp*).

Транслятор – программа, осуществляющая перевод текстов с одного языка на другой, т.е. с входного языка системы программирования на машинный язык. Одной из разновидностей транслятора является *компилятор* (*Compiler*), в котором трансляция отделена от выполнения программ. Другим видом транслятора является интерпретатор. *Интерпретатор* (*Interpreter*) выполняет созданную программу путем одновременного анализа и реализации предписанных действий, при использовании отсутствует разделение на две стадии – перевод и выполнение. Большинство трансляторов языка C/C++ – компиляторы.

Результат обработки исходного модуля компилятором – *объектный модуль* (*Object module*) (расширение *.*obj*), это незавершенный вариант машинной программы (например, к нему должны быть присоединены модули стандартных библиотек). Он содержит текст программы

на машинном языке и дополнительную информацию, обеспечивающую объединение этого модуля с другими независимо транслируемыми модулями. Объектный модуль готов к редактированию связей.

Исполняемый (абсолютный, загрузочный) модуль создает вторая специальная программа – *компоновщик*. Ее еще называют *редактором связей* (*Linker*). Он и создает модуль, пригодный для выполнения на основе одного или нескольких объектных модулей. Компоновщик собирает, устанавливает связи между модулями и создает еще один вид программного модуля – загрузочный модуль.

Загрузочный модуль (*Load module*) (расширение **.exe*) – это программный модуль, представленный в форме, пригодной для выполнения.

На рис. 3.1 показана схема решения задачи с помощью компьютера, в которой учтена необходимость использования языка программирования.

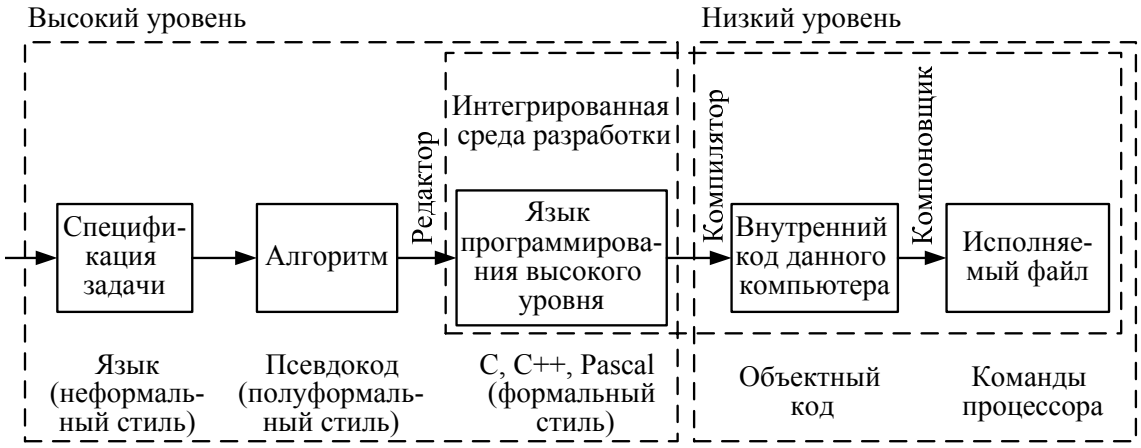


Рис. 3.1. Схема решения задачи

Библиотека программ представляет собой программы, предназначенные для решения распространенных задач. Программы, включенные в библиотеку, оформляются специальным образом, облегчающим их вызов, использование, передачу входных данных и результатов.

3.2. Интегрированная среда разработки Visual Studio 2008 и Visual C++

Среда разработки Microsoft Visual Studio (VS) 2008 отличается от предыдущих версий повышением производительности разработчиков, поддержкой новейших технологий и управлением всем циклом создания приложений. Существуют следующие варианты поставки: VS Team System 2008, VS Team System 2008 Team Suite,

VS Team System 2008 Architecture Edition, VS Team System 2008 Development Edition, VS Team System 2008 Test Edition, VS Team System 2008 Database Edition, VS Team System 2008 Team Foundation Server, VS Team System 2008 Test Load Agent, Visual C++ 2008 Express Edition и др.

Visual C++ состоит из следующих компонентов.

Средства компилятора Visual C++ 2008. Компилятор поддерживает как традиционную разработку с использованием машинного кода, так и разработку с использованием платформ виртуальных машин, таких как среда CLR (Common Language Runtime). Компилятор продолжает напрямую поддерживать архитектуру x86 и оптимизирует производительность кода для обеих платформ.

Библиотеки Visual C++ содержат общепризнанную библиотеку шаблонных классов ATL (Active Template Library), библиотеки MFC (Microsoft Foundation Class) и стандартные библиотеки, такие как библиотеки стандартных шаблонов STL (Standard Template Library) и библиотеки времени выполнения языка C CRT (Common Runtime Library). Библиотека CRT включает альтернативные функции с улучшенной безопасностью для функций с известными проблемами безопасности. Библиотека STL/CLR позволяет разработчикам, использующим управляемый код, использовать также и возможности библиотеки STL.

Среда разработки Visual C++ предоставляет всестороннюю поддержку при управлении проектами и их настройке (включая улучшенную поддержку больших проектов), редактировании исходного кода, просмотре исходного кода, а также мощные средства отладки.

Кроме традиционных приложений с пользовательским интерфейсом Visual C++ позволяет разрабатывать веб-приложения, приложения интеллектуальных клиентов для Windows, решения для мобильных устройств, использующих «тонкие клиенты» и «интеллектуальные клиенты». Язык C++, являющийся самым популярным в мире языком уровня системы, и Visual C++ вместе предоставляют разработчику высококлассное средство мирового уровня для построения программного обеспечения.

3.3. Создание проекта

Программы (чаще именуемые приложениями), создаваемые в среде разработки Microsoft Visual Studio 2008 (MS VS), представляются в виде двух типов контейнеров, вложенных один в другой. Один (главный контейнер) называется *решения* (*Solutions*), а другой – *проект* (*Project*). Проект определен как конфигурация, объединяющая

группу файлов. Окно диспетчера решения может содержать множество проектов, а проект содержит множество элементов. Такой подход к оформлению приложения позволяет работать с группой проектов как с одним целым, что ускоряет процесс разработки приложений.

При написании программы на Visual C++ с помощью Visual Studio первым этапом является выбор типа проекта. Для каждого типа проекта Visual Studio устанавливает параметры компилятора и генерирует стартовый код.

Консольное приложение

При изучении C/C++ будем пользоваться специальным видом приложений – консольными приложениями, которые формируются на основе заранее заготовленных в среде проектирования шаблонов. Консольные (опорные, базовые) приложения – это приложения без графического интерфейса, которые взаимодействуют с пользователем через специальную командную строку или запускаются специальной командой из главного меню среды. Шаблон консольного приложения добавляет в создаваемое приложение необходимые элементы, после чего разработчик вставляет в шаблон свои операторы C/C++. Общение с пользователем происходит через специальное окно, открывающееся средой после запуска приложения (в него выводятся сообщения программы, вводятся данные и выводится результат).

Создание нового проекта

Для создания нового проекта нужно выбрать команду меню **File/New/Project…**. Система предложит создать новый проект, и появится диалог, представленный на рис. 3.2.

В закладке *Visual C++* в списке различных типов выполняемых файлов необходимо выбрать *Win32*, а в окне *Templates – Win 32 Console Application*. Затем следует набрать имя проекта (например, *My_first*) в поле *Name* и имя каталога, в котором будут храниться все файлы, относящиеся к данному проекту, в поле *Location*. После этого нажать кнопку *OK*.

На следующем шаге мастера необходимо нажать кнопку *Finish*, и результатом будет заготовка консольного приложения (рис. 3.3).

Как и другие окна операционной системы (ОС) Windows, окно среды разработки содержит строку заголовка, меню и панели инструментов. В рабочей области среды разработки содержится окно редактора для ввода программного кода, окно *Обозреватель решений и проектов* (*Solution Explorer*).

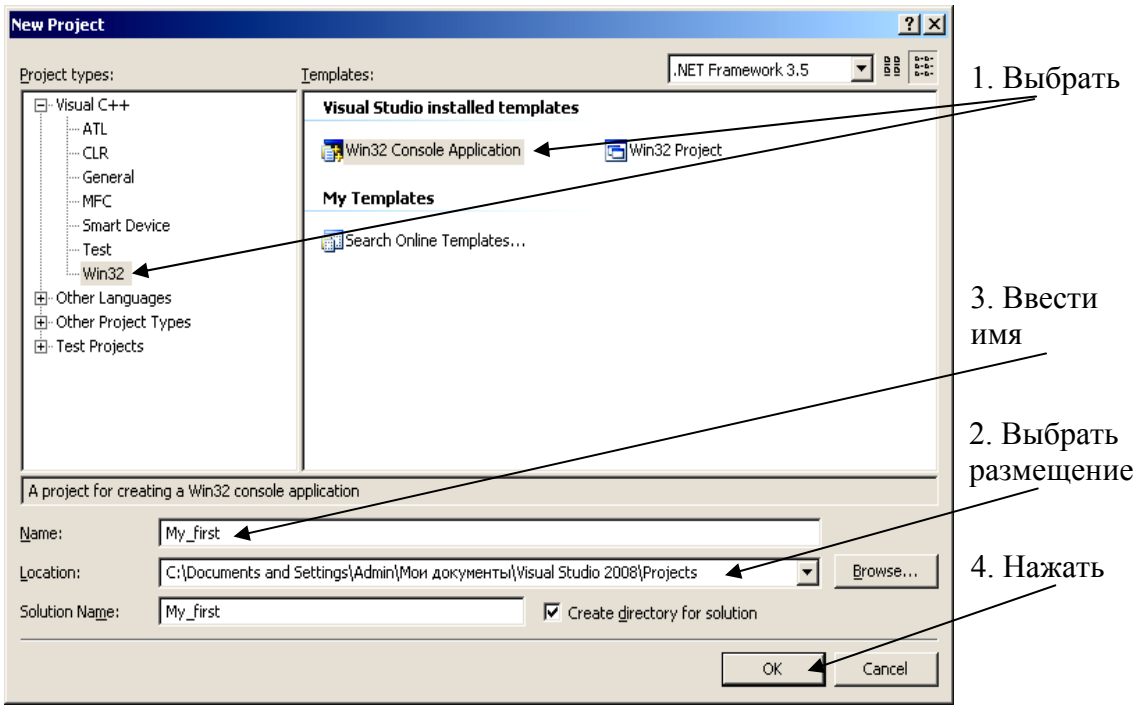


Рис. 3.2. Диалог создания нового проекта

Заготовка состоит из заголовка функции `int _tmain(int argc, _TCHAR* argv[])` и тела, ограниченного фигурными скобками (рис. 3.3). Теперь можно набирать текст как в обычном текстовом редакторе, работать необходимыми для ввода и редактирования клавишами.

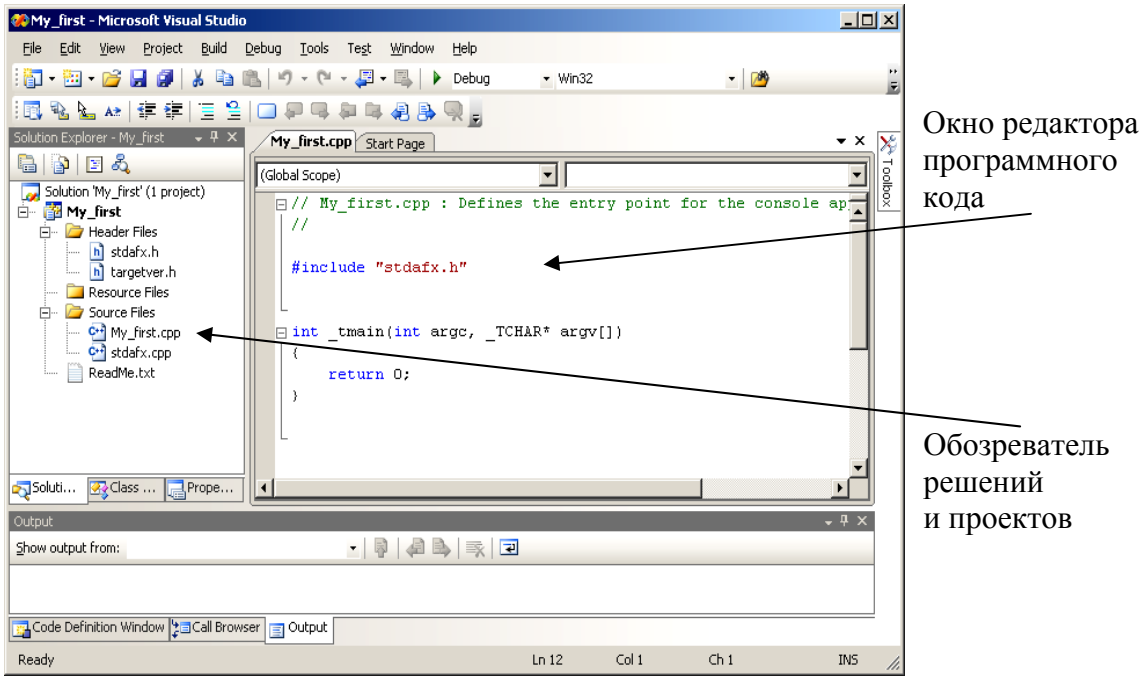


Рис. 3.3. Заготовка консольного приложения

Созданный по шаблону проект содержит: `My_first.cpp` – это главный исходный файл и точка входа в создаваемое приложение; `stdafx.cpp` – подключает специальный файл для компиляции приложения; `stdafx.h` – подключает специальные файлы для компиляции приложения, со следующим содержанием:

```
#pragma once

#include "targetver.h"

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

`targetver.h` – позволяет использовать специфические свойства Windows Vista. Содержимое файла:

```
#pragma once
#ifndef _WIN32_WINNT
#define _WIN32_WINNT 0x0600
#endif
```

`ReadMe.txt` – файл, описывающий некоторые из созданных шаблонов.

3.4. Загрузка существующего приложения

Для запуска интегрированной среды разработки и загрузки существующего приложения можно использовать: «ручной» запуск среды с последующим открытием решения или открытие приложения с автоматическим запуском среды разработки.

При первом варианте необходимо: запустить интегрированную среду – **Пуск/Программы/Microsoft Visual Studio 2008/Microsoft Visual Studio 2008**; открыть существующее приложение командой **File/Open/Project/Solution**; в появившемся диалоговом окне найти файл решения `*.sln` и щелкнуть по кнопке **Open**. Решение будет открыто, окно редактирования активизировано, и в это окно будет загружен исходный код программы.

При втором варианте загрузки приложения необходимо: найти файл решения `*.sln` с помощью стандартных средств ОС; открыть приложение двойным щелчком; в результате произойдет автоматический запуск интегрированной среды разработки.

3.5. Ввод и редактирование программного кода

Редактор обеспечивает все стандартные действия, которые доступны для любого другого редактора (набор программного кода,

редактирование, копирование, вставка, поиск и т.д.), и кроме того, обладает большим набором дополнительных возможностей. Редактор программного кода поддерживает оперативную (в процессе ввода текста) проверку программы.

Для получения справочной информации нужно установить текстовый курсор на элемент программы, для которого необходимо наличие справки, и нажать клавишу **F1** (справка будет выдана на английском языке; получение справки возможно только в случае, если на компьютере установлена справочная служба MSDN Library). Получаемая информация содержит примеры практического использования рассматриваемых элементов.

IntelliSense

Для быстрого и безошибочного набора программного кода в редакторе среды MS VS имеется специальная служба **IntelliSense**, которая обеспечивает: отображение списка методов и полей для классов, структур, пространства имен и других элементов кода (выбор нужного варианта может быть выполнен, например, при помощи двойного щелчка мыши на требуемой строке списка); отображение информации о параметрах для методов и функций (рис. 3.4.) (осуществляется автоматически после ввода имени метода или функции); отображение краткого описания элементов кода программы (происходит при наведении указателя мыши на нужный элемент кода); завершение слов при наборе наименований команд и имен функций; автоматическое сопоставление правильности расстановки скобок (скобки },],) и `#endif` выделяются более темным цветом вместе с соответствующей открывающейся скобкой). Служба IntelliSense может быть отключена.

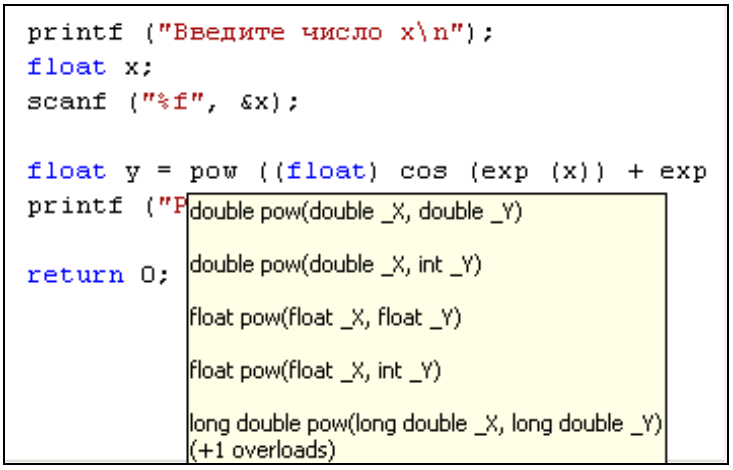


Рис. 3.4. Служба IntelliSense, отображение функций

Цветовая индикация текста в коде программы

Слова исходного текста программы отображаются разными цветами в зависимости от характера текста. С помощью цветов выделяются комментарии, ключевые слова, имена классов, имена объектов, символьные строки-литералы. По умолчанию принят следующий вариант индикации:

– комментарии – зеленый цвет;
– ключевые слова – синий цвет;
– символьные строки-литералы – коричневый цвет;
– весь остальной текст, включая имена объектов – черный цвет.

Скрытие фрагментов текста

Текстовый процессор позволяет скрывать (не отображать) фрагменты исходного кода, которые ограничены слева вертикальными линиями с флажками в виде прямоугольников (рис. 3.5).

Установленный символ «+» в флажке означает, что фрагмент кода скрыт, а символ «–» означает, что скрытого текста нет. Выполняя щелчки на флажках, можно скрывать или раскрывать фрагменты исходного кода.

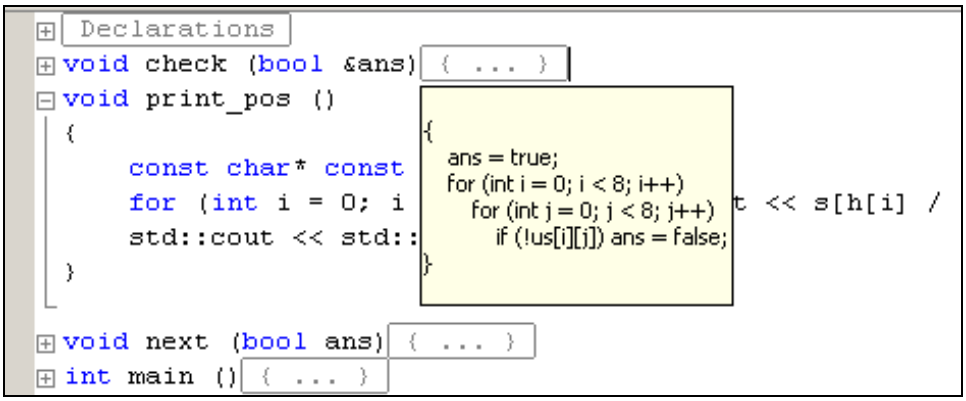


Рис. 3.5. Всплывающее окно просмотра скрытого текста

Имеется возможность просмотреть скрытый код через всплывающее окно (рис. 3.5). Это окно используется для оперативного просмотра скрытого кода без его раскрытия. Оперативный просмотр выполняется удержанием курсора мыши в рамке с многоточием. Рамка отображается на месте скрытого кода.

Скрытие кода позволяет сосредоточить внимание на том фрагменте кода, который редактируется в данный момент, если убрать из поля зрения остальные фрагменты. Особенно удобно применять эту функцию при работе с файлами большого размера. В этом случае

скрытие является хорошей альтернативой листанию файла в окне редактирования.

Редактор среды Visual C++ многооконный, можно открыть сколько угодно окон с текстами файлов, заголовков и программ. Для переключения между этими окнами используется комбинация **Ctrl+F6**.

3.6. Изменение структуры приложения

При разработке приложения может возникнуть необходимость внесения изменения не только в исходный код, но и в состав файлов проекта. Изменение структуры проекта, и как следствие, структуры приложения, можно выполнить с помощью проводника проектов и решений (рис. 3.3).

В окне обозревателя решений выделите проект (в примере папка `My_first`). В результате станут доступны средства изменения структуры проекта, в частности: добавление файла к проекту; исключение файла из проекта; исключение файла из проекта и его физическое удаление; переименование файлов проекта.

Для выполнения указанных действий надо открыть папку проекта. Щелчком левой кнопки мыши указывается нужный файл, затем щелчком правой кнопки активизируется всплывающее меню.

Команда **Exclude From Project** позволяет исключить файл из проекта, при этом файл физически сохраняется (в той папке, где размещен). Команда **Delete** позволяет исключить файл из проекта, при этом файл физически уничтожается. Команда **Rename** позволяет назначить файлу новое имя. Подключение к проекту существующего файла выполняется командой **Add/Existing Item**….

В открывшемся диалоговом окне с использованием стандартных средств ОС Windows можно добавить требуемый файл. Этот файл будет включен в проект. Физическое копирование файла в папку проекта не требуется. Аналогично можно добавлять к текущему решению проекты и удалять их.

3.7. Выполнение приложения в режиме отладки

Средства отладки программ

Процесс отладки состоит из многократных попыток выполнения программы на компьютере и анализа получившихся результатов. Ошибки, допускаемые при написании программ, разделяются на синтаксические и логические.

Синтаксические ошибки – нарушение формальных правил написания программы на конкретном языке, обнаруживаются на этапе трансляции и могут быть легко исправлены. Трансляторы выдают информацию о синтаксических ошибках, указывают места ошибок и их характер.

Логические ошибки – ошибки алгоритма и семантические, которые могут быть исправлены только разработчиком программы. Причина ошибки алгоритма – несоответствие построенного алгоритма ходу получения конечного результата сформулированной задачи. Причина семантической ошибки – неправильное понимание смысла операторов языка.

На этапе редактирования связей обнаруживаются ошибки, связанные с неправильным оформлением функций, ошибки в командах вызова функций и программ из библиотеки. На этапе выполнения обнаруживаются логические ошибки программы (например, деление на ноль, бесконечный цикл и т.п.). Среда отладки позволяет в пошаговом режиме обнаружить и локализовать ошибку.

Если программа содержит синтаксические ошибки, при выполнении построения они автоматически отображаются в окне *Output* (Вывод), по умолчанию расположенном в нижней части окна интегрированной оболочки (рис. 3.6). Каждое сообщение начинается с имени исходного файла и номера строки (в скобках), где обнаружена ошибка или предупреждение. Следом за номером строки идет номер и краткое описание предупреждения или ошибки. По номеру можно найти описание ошибки в документации. Лучше всего добиться, чтобы в окончательном варианте не было ни того, ни другого, хотя с предупреждениями исполняемый файл создается и может быть запущен. Если дважды щелкнуть на строке с сообщением об ошибке, то среда автоматически переключится в окно редактирования и сама укажет на ошибочный фрагмент программы (рис. 3.6). Следует заметить, что компилятор может «не очень точно» локализовать место возникновения ошибки, особенно если это пропущенная скобка или точка с запятой. В этом случае надо внимательно проверить текст над указанной строкой. Исправив все ошибки, необходимо повторить построение исполняемого файла.

После того, как программа становится работоспособной, проводится ее тестирование – проверка правильности ее функционирования на различных наборах исходных данных из диапазона допустимых значений.

Даже если синтаксических ошибок нет, приложение может работать не так, как ожидалось. Наиболее эффективные средства интегрированного отладчика: выполнение программы по шагам, просмотр значений любых переменных в любой точке программы, задание точек останова и др.

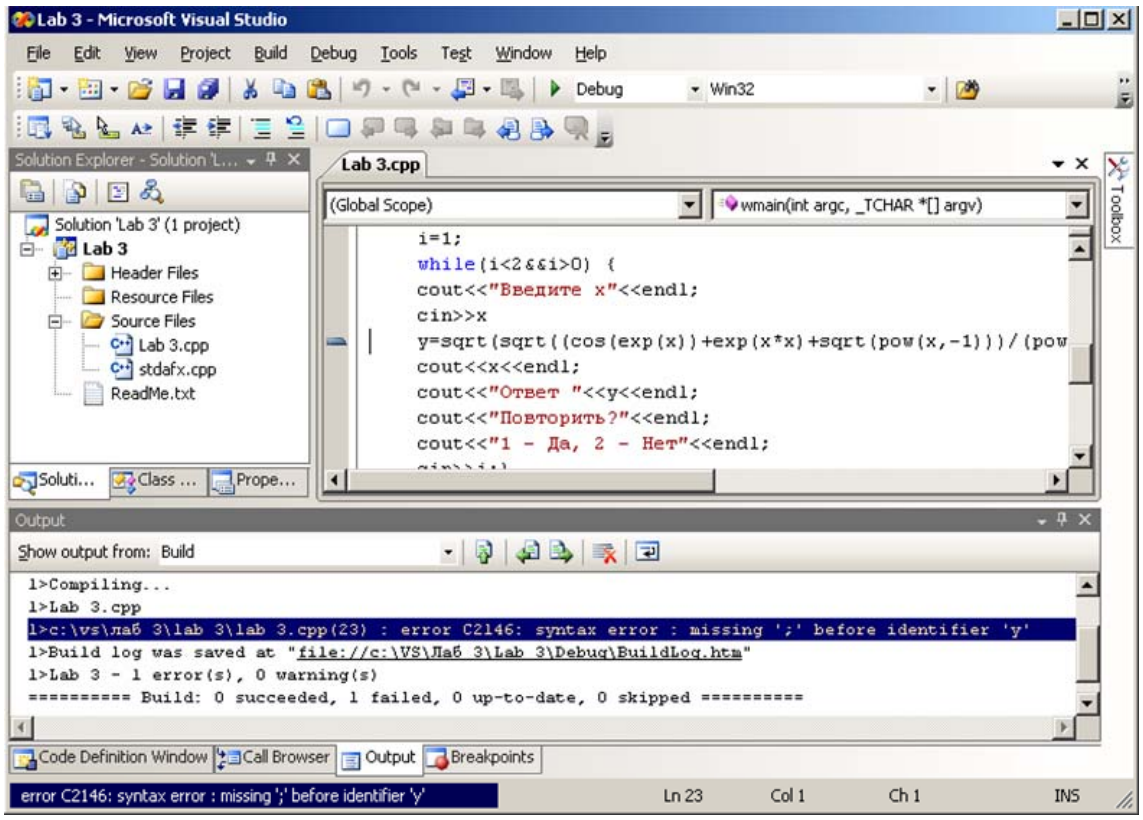


Рис. 3.6. Построение проекта с ошибками

При анализе сообщений, выдаваемых транслятором, необходимо учитывать следующие факторы:

– транслятор при запуске анализирует весь исходный код и пытается выявить все ошибки (*Errors*). Поэтому не исключено, что одна ошибка может повлечь за собой множество сообщений. Например, ошибка в объявлении переменной приведет к появлению сообщений об ошибках в тех строках исходного кода, где эта переменная используется. Признаком такого рода ошибки являются многочисленные сообщения однотипного характера. Не пытайтесь исправить абсолютно все ошибки за один раз. После внесения нескольких исправлений в исходный код можно запустить приложение. Если были внесены исправления в строку с первопричинной ошибкой, многие (а возможно все) сообщения исчезнут;

– транслятор контролирует соблюдение формальных правил записи операторов. В некоторых случаях ошибка, допущенная в операторе, не нарушает синтаксических правил в нем, но приводит к нарушению правил в других операторах. В этих случаях транслятор фиксирует следствие ошибки, а не ее первопричину. Поэтому не следует слепо доверять указаниям транслятора на характер и место ошибки.

Помимо сообщений об ошибках, транслятор может выдавать предупреждения (*Warnings*). Предупреждения выдаются при обнаружении «подозрительных» с точки зрения логики операторов, хотя синтаксические правила их записи не нарушены. По умолчанию предупреждения не препятствуют построению решения и его выполнению. Тем не менее, стоит внимательно проанализировать предупреждения. Часто предупреждения являются косвенным признаком наличия в исходном коде логических ошибок.

По характеру использования средства отладки можно разделить на две группы: средства интерактивной отладки; средства планируемой отладки. Средства первой группы позволяют выполнить программу по шагам. Каждый шаг соответствует выполнению кода, указанного в одной строке программы. На каждом шаге планируется один следующий шаг отладки, это позволяет реализовать гибкий сценарий отладки. Средства второй группы позволяют спланировать определенный сценарий процесса отладки на множестве шагов. Они хорошо приспособлены к многократным отладочным прогонам, но имеют меньшую гибкость отладочных действий.

Трассировка выполняется с точностью до одной строки исходной программы, а контроль значений – с точностью до одного объекта. Поэтому рекомендуется не располагать в одной строке программы несколько операторов и не использовать слишком сложных выражений.

Средства интерактивной и планируемой отладки могут применяться в любой последовательности и в любом сочетании исходя из целей, которые преследуются в конкретном отладочном прогоне.

Управление средствами отладки выполняется «горячими клавишами» или через главное меню среды разработки.

Выполнение приложения с использованием средств интерактивной отладки

Используют два способа пошагового выполнения приложения:

– без трассировки вызываемых методов (клавиша **F10** или иконка на панели инструментов *Debug* (рис. 3.7), или команда меню **Debug/Step Over**);

– с трассировкой вызываемых методов (клавиша **F11** или иконка на панели инструментов *Debug* (рис. 3.7), или команда меню **Debug/Step Into)**.

Рис. 3.7. Панель инструментов *Debug*

При обоих способах производится останов перед выполнением текущей строки исходного кода. Различия между командами **Step Into** и **Step Over** проявляются только тогда, когда в программе встречается вызов функции. Если выбрать команду Step Into, то отладчик войдет в функцию и начнет выполнять шаг за шагом все ее операторы. При выборе команды **Step Over** отладчик выполнит функцию как единое целое и перейдет к строке, следующей за вызовом функции. Эту команду удобно применять в тех случаях, когда в программе делается обращение к стандартной функции или созданной вами подпрограмме, которая уже была протестирована.

Для контроля значений полей и свойств объектов используются всплывающие окна. Необходимо подвести курсор мыши к имени интересующего объекта и удержать его некоторое время. Появится всплывающее окно, в котором будет указано имя объекта (поля) и его текущее значение (рис. 3.8). Сдвиг курсора мыши приводит к исчезновению этого окна.

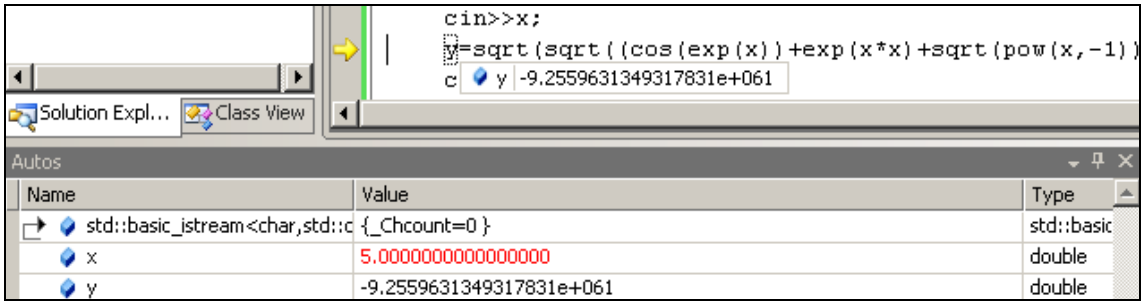


Рис. 3.8. Окно просмотра текущего значения переменной

После достижения целей, которые были запланированы на сеанс отладки, необходимо вывести приложение из отладочного режима командой **Shift+F5** или **Debug/Stop Debugging** (или одноименная кнопка на панели инструментов *Debug*).

Выполнение приложения с использованием средств планируемой отладки

При интерактивной или планируемой отладке используются точки останова (точки, в которых отладчик автоматически прерывает выполнение). Место точки основа можно выбрать по своему усмотрению (обычно место, в котором может содержаться ошибка). Точка останова назначается щелчком левой кнопки мыши в специальном поле слева от строки текста программы. Назначенная точка останова отмечается маркером в виде красного круга слева от строки (рис. 3.9). Повторный

щелчок левой кнопки мыши на маркере точки останова приводит к ее отмене. Точку останова можно назначить и отменить клавишей **F9** или командой меню **Debug/Toggle Breakpoint**. В этом случае она устанавливается на той строке кода программы, где помещен курсор.

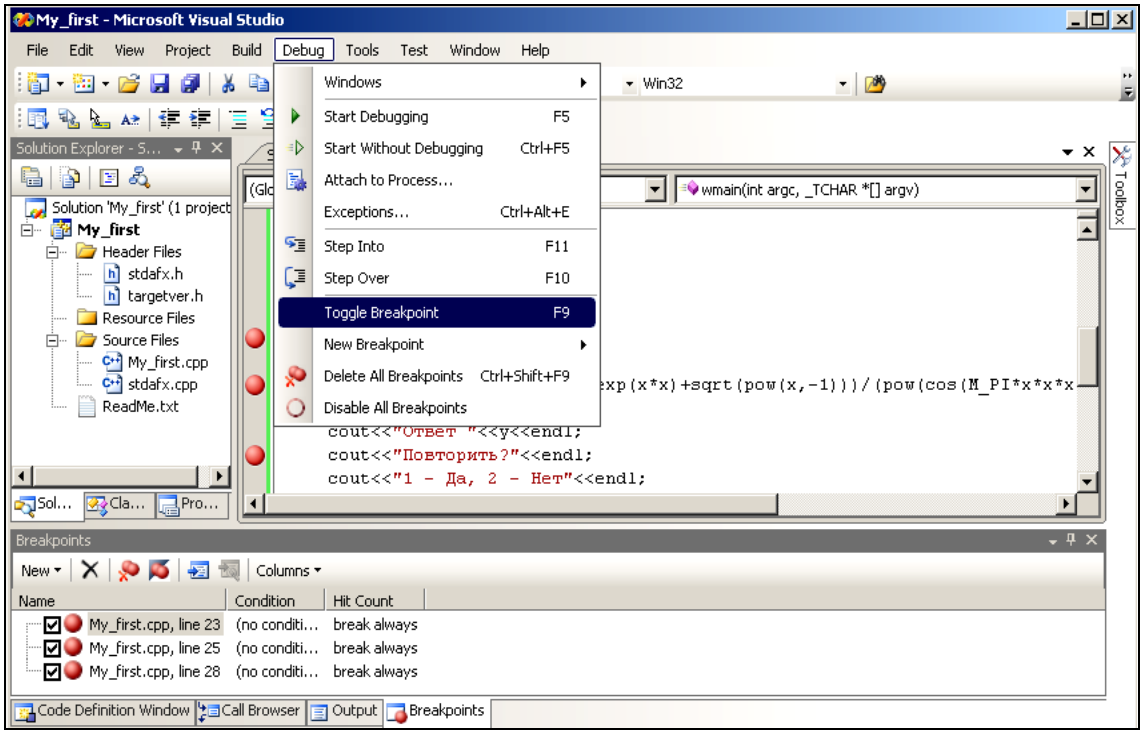


Рис. 3.9. Способы установки точки останова

После назначения точки останова приложение запускается в режиме отладки функциональной клавишей **F5** или командой меню **Debug/Start**. Приложение будет выполнено до точки останова. Достигнутая точка останова отмечается маркером в виде стрелки внутри маркера точки. Выполнить приложение до очередной точки останова можно, повторив указанные команды.

Для контроля значений рекомендуется использовать средства непрерывного контроля состояния объектов. Имена интересующих объектов после запуска программы в режиме отладки заносят в специальное окно просмотра *Watch* (рис. 3.10). Имя заносится в столбец *Name*. Последующее нажатие *ENTER* или щелчок клавишей мыши на строке с именем приводит к появлению в поле (в столбце) *Value* значения переменной (объекта).

Если рядом с именем переменной стоит знак плюс, то для этой переменной может быть отображена дополнительная информация (массивы, указатели или объекты класса). Если нажать <F10> два раза

и щелкнуть на «+» возле имени переменной, то отладчик отобразит значение, хранимое в памяти по адресу, содержащемуся в указателе.

Окно *Watch* также позволяет изменять значения переменных, за которыми ведется наблюдение. В том случае, когда ясно, что отображаемое значение не верно, можно установить корректное значение и продолжить поиск ошибок. Это средство можно использовать для пропуска первых шагов в цикле с большим количеством итераций. Чтобы изменить значение, выполните двойной щелчок на отображаемом значении переменной и введите новое.

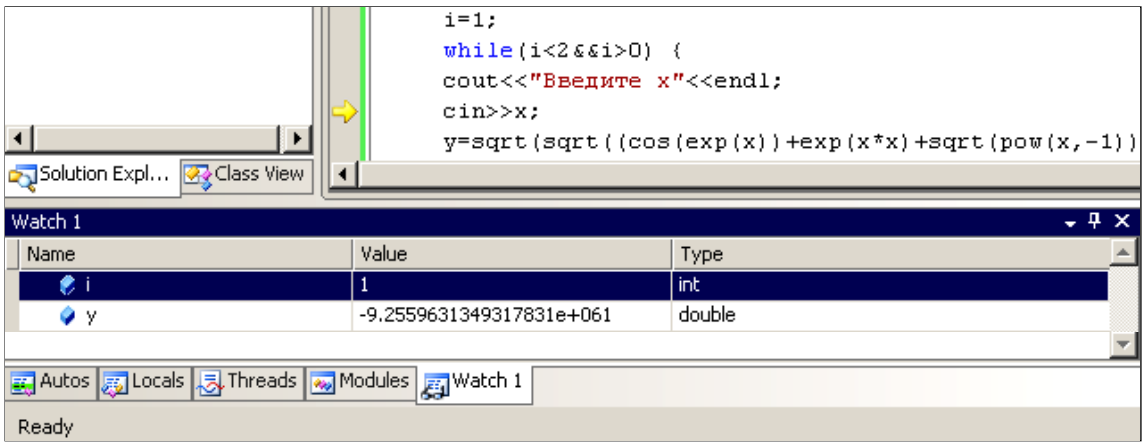


Рис. 3.10. Окно просмотра *Watch*

Может быть назначено до четырех окон просмотра с помощью команд меню: **Debug/Windows/Watch/Watch1** (*Watch2*, *Watch3* или *Watch4*). Переход от одного окна просмотра к другому выполняется щелчком мыши на закладке окна. Состояние объектов отображается в окнах просмотра, причем последнее изменение состояния отмечается красным цветом. Если надобность в контроле состояния объекта отпадает, имя объекта можно удалить из окна просмотра, выделив его курсором мыши и нажав клавишу **DEL**.

Кроме окна *Watch* имеется еще 4 вкладки в нижней части экрана (рис. 3.10). Вкладка *Autos* (Автоматические) показывает автоматические переменные, используемые в текущем операторе. Вкладка *Locals* (Локальные) показывает значения переменных, локальных по отношению к текущей функции. Вкладка *Threads* (Потоки) позволяет просматривать и управлять потомками в многопоточных приложениях. Вкладка *Modules* (Модули) перечисляет детали модулей кода, выполняемых в данный момент.

Завершение сеанса отладки выполняется командой **Shift+F5** или **Debug/Stop Debugging**.

3.8. Создание исполняемого файла без отладочной информации

Для запуска приложения в данном режиме надо выполнить команду **Debug/Start Without Debugging**. При запуске приложения последовательно реализуются два процесса:

– *построение решения*. Исходный код из файла транслируется в промежуточный код. Промежуточный код приложения сохраняется в файле `*.exe`;

– *выполнение решения*. Автоматически запускается среда исполнения, в которой выполняется код, содержащийся в файле `*.exe`.

Если в исходном коде есть ошибки, решение не будет построено. Синтаксические ошибки будут выявлены в процессе трансляции исходного файла `*.cpp` в промежуточный код. Сообщение о наличии ошибок выводится в окне, вид которого приведен на рис. 3.11.

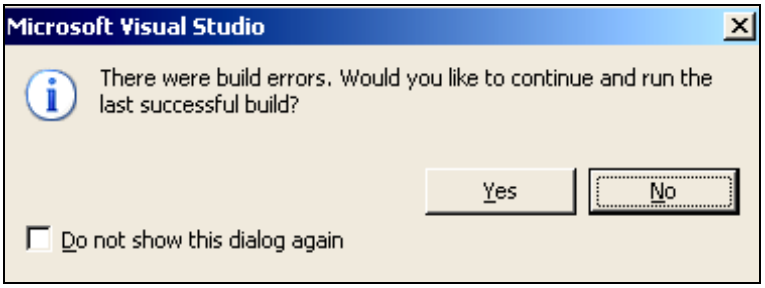


Рис. 3.11. Окно сообщений об ошибках

Надо отказаться от построения решения, щелкнув по кнопке **No**, и исправить ошибки.

До этого построение приложения осуществлялось в отладочной конфигурации (*Win32 Debug*) с включением в файл всей необходимой отладочной информации, что позволило в дальнейшем использовать возможности отладки (рис. 3.12). Эта информация сохраняется в файле `*.pdb`. После отладки и исправления всех ошибок осуществляется построение приложения в рабочей конфигурации (*Release*). Рабочая конфигурация не содержит какой-либо отладочной информации и использует заданную оптимизацию кода. Процесс оптимизации может изменять последовательность выполнения кода, чтобы сделать его более эффективным, или даже вовсе исключать излишний код. Как результат, размер исполняемого файла существенно уменьшается. Поскольку это разрушает отображение «один к одному» между исходным кодом и соответствующими блоками исполняемого машинного

кода, оптимизация может затруднить или запутать пошаговое выполнение программы. Для переключения в окончательную конфигурацию необходимо выбрать команду **Build/Configuration Manager** из меню. На экран будет выведено диалоговое окно установки активной конфигурации проекта. Выберается опция *Win32 Release*. Повторяется построение выполнением команды **Build/Rebuild All**. Каждая конфигурация проекта определяет также папки, куда будут помещены файлы с промежуточными и окончательными результатами компиляции и компоновки. По умолчанию это папки *Debug* и *Release*, которые располагаются в папке проекта.

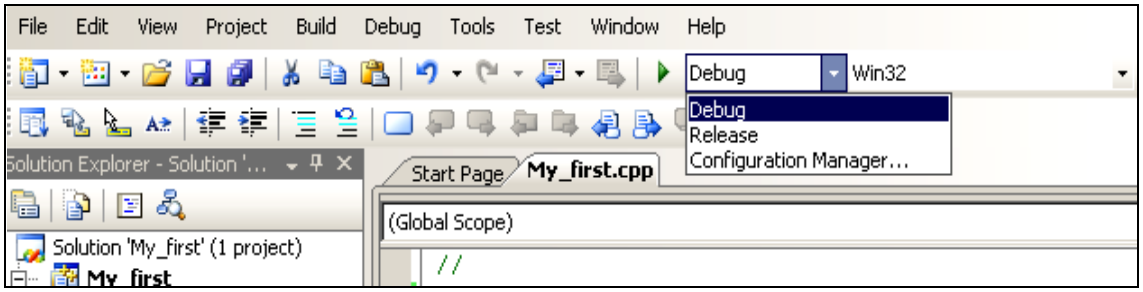


Рис. 3.12. Выбор отладочной конфигурации

Коротко рассмотрим назначение папок и файлов приложения. Папка решения содержит: файл текущего примера решения `*.sln`, файл с информацией о проектах решения и опциями решения `*.suo`, данные `Intellisense` для решения и вложенную паку с названием решения. В файле решения зафиксирован перечень проектов, входящих в решение. Во вложенной папке находятся файлы и папки проекта: файл `My_first.vcproj` в формате XML содержит перечень файлов, включенных в проект; файл `My_first.cpp` содержит исходный код программы на языке C++; `*.obj` – объектные файлы, содержащие машинный код исходных файлов проекта; `*.pch` – предварительно скомпилированный файл заголовков; `*.pdb` – файл с отладочной информацией, используемой при выполнении программы в режиме отладки; `*.idb` – файл с информацией, необходимой для перестройки всего решения и др. Папка *Debug* используется для хранения временных файлов. В этой папке размещаются файлы с программным кодом на промежуточном языке. В частности, файл `My_first.exe` содержит программный код, который реализует функциональность приложения.

В прил. 2 содержится описание команд меню интегрированной среды разработки VS 2008.

Глава 4. СТРУКТУРА ПРОГРАММЫ, БАЗОВЫЕ ЭЛЕМЕНТЫ, ФУНДАМЕНТАЛЬНЫЕ ТИПЫ ДАННЫХ И ВВОД-ВЫВОД В C/C++

4.1. История языков программирования C и C++

Программы имеют два основных аспекта – набор алгоритмов и набор данных, которыми они оперируют. Эти два аспекта оставались неизменными за всю историю программирования, зато отношения между ними (парадигма программирования) менялись.

Язык C был разработан Д. Ритчи в Bell Laboratories (США) в 1972 году как универсальный язык системного программирования в связи с разработкой операционной системы UNIX. К концу 70-х годов. C развился в то, что теперь относят к «традиционному C» или «классическому C», который относится к процедурной парадигме программирования (задача непосредственно моделируется набором алгоритмов). В 1989 году был утвержден стандарт языка C – ANSI/ISO 9899:1990.

При классификации языков программирования язык C вызывает некоторые трудности. По сравнению с ассемблером, это высокоуровневый язык. Однако C содержит много низкоуровневых средств для непосредственных операций с памятью компьютера. Поэтому он отлично подходит для написания эффективных «системных» программ.

C++ – компилируемый статически типизированный язык программирования общего назначения, был разработан Бьерном Страуструпом в начале 80-х годов в Bell Laboratories и является своего рода расширением языка C. Поэтому большинство конструкций ANSI C являются законными для C++, причем смысл их не меняется. При этом C++ поддерживает разные парадигмы программирования (процедурную и объектную), но в сравнении с его предшественником, языком C, наибольшее внимание уделено поддержке объектно-ориентированного и обобщенного программирования. Можно сказать, что C++ – это гибридный язык. Он предоставляет возможность программировать и в стиле C, и в объектно-ориентированном стиле, и в обоих стилях сразу. В 1998 году был принят стандарт языка C++ ISO/IEC 14882:1998.

После принятия технических исправлений в 2003 году вышла новая версия стандарта – ISO/IEC 14882:2003. В настоящее время рабочая группа МОС (ISO) работает над ведующей версией стандарта под названием C++0X. Новый стандарт будет включать дополнения в ядре языка и ряд расширений.

Несмотря на то, что большая часть кода C будет справедлива и для C++, C++ не является надмножеством C и не включает его в себя.

4.2. Базовые элементы языка C/C++

Программа на C++ состоит из одного или нескольких файлов. С логической точки зрения файл транслируется за несколько проходов. Первый проход состоит в препроцессорной обработке, на которой происходит включение файлов и макроподстановка. Работа препроцессора управляется с помощью команд, являющихся строками, первый символ которых, отличный от пробела, есть #. Результат работы препроцессора – это последовательность лексем, которую называют единицей трансляции.

Лексемы

Существуют лексемы пяти видов: *идентификаторы*, *служебные слова*, *литералы*, *операции* и *различные разделители*. Пробелы, вертикальная и горизонтальная табуляция, конец строки, перевод строки и комментарии (обобщенные пробелы) игнорируются, за исключением того, что они отделяют лексемы. Обобщенные пробелы нужны, чтобы разделить стоящие рядом идентификаторы, служебные слова и константы. Если входной поток разобран на лексемы до данного символа, то следующей лексемой считается лексема с максимально возможной длиной, которая начинается с этого символа.

Используемые символы

Первая группа – символы, используемые для образования ключевых слов и идентификаторов.

Прописные буквы латинского алфавита	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Строчные буквы латинского алфавита	a b c d e f g h i j k l m n o p q r s t u v w x y z
Символ подчеркивания	_

Одинаковые прописные и строчные буквы считаются различными символами, так как имеют различные коды.

Вторая группа – прописные и строчные буквы русского алфавита и арабские цифры.

Прописные буквы русского алфавита	А Б В Г Д Е Ж З И К Л М Н О П Р С Т У Ф Х Ц Ч Ш Щ Ы Ь Э Ю Я
Строчные буквы русского алфавита	а б в г д е ж з и к л м н о п р с т у ф х ц ч ш щ ъ ы ь э ю я
Арабские цифры	0 1 2 3 4 5 6 7 8 9

Третья группа – знаки нумерации и специальные символы (табл. 4.1).

Таблица 4.1

Знаки нумерации и специальные символы

Символ	Наименование	Символ	Наименование
,	Запятая	)	Круглая скобка правая
.	Точка	(	Круглая скобка левая
;	Точка с запятой	}	Фигурная скобка правая
:	Двоеточие	{	Фигурная скобка левая
?	Вопросительный знак	<	Меньше
'	Апостроф	>	Больше
!	Восклицательный знак	[	Квадратная скобка левая
|	Вертикальная черта	]	Квадратная скобка правая
/	Дробная черта	#	Номер
\	Обратная черта (слеш)	%	Процент
~	Тильда	&	Амперсанд
*	Звездочка	^	Логическое «не»
+	Плюс	=	Равно
–	Минус	"	Кавычки

Они используются для организации процесса вычислений и для передачи компилятору определенного набора инструкций.

Четверная группа – управляющие и разделительные символы: пробел, символы табуляции, перевода строки, возврата каретки, новая страница и новая строка (табл. 4.2). Эти символы отделяют друг от друга объекты, определяемые пользователем, к которым относятся константы и идентификаторы.

Таблица 4.2

Управляющие и разделительные символы

Управляющая последовательность	Наименование	Шестнадцатеричная замена
\a	Звонок	007
\b	Возврат на шаг	008
\t	Горизонтальная табуляция	009
\n	Переход на новую строку	00A
\v	Вертикальная табуляция	00B
\r	Возврат каретки	00C
\f	Перевод формата	00D
\"	Кавычки	022
\'	Апостроф	027
\0	Ноль-символ	000
\\	Обратная дробная черта	05C

Окончание табл. 4.2

Управляющая последовательность	Наименование	Шестнадцатеричная замена
\ddd	Символ набора кодов в восьмеричном представлении	
\xddd	Символ набора кодов в шестнадцатеричном представлении	

Управляющая последовательность строится на основе использования обратной дробной черты (\) (обязательный первый символ) и комбинации латинских букв и цифр.

В строковых константах всегда обязательно задавать все три цифры в управляющей последовательности. Например, отдельную управляющую последовательность \n (переход на новую строку) можно представить как \010 или \xA, но в строковых константах необходимо задавать все три цифры, в противном случае символ или символы, следующие за управляющей последовательностью, будут рассматриваться как ее недостающая часть.

Идентификаторы

Идентификатор – это последовательность букв и цифр произвольной длины. Первый символ должен быть буквой (символ подчеркивания _ считается буквой). Прописные и строчные буквы различаются. Все символы существенны. Например, два идентификатора, для образования которых используются совпадающие строчные и прописные буквы, считаются различными: abc и ABC, A128B и a128b.

Идентификатор создается на этапе объявления переменной, функции, структуры и т.п., после этого его можно использовать в последующих операторах разрабатываемой программы. Идентификатор не должен совпадать с ключевыми словами, с зарезервированными словами и именами функций библиотеки компилятора языка C/C++.

При задании идентификаторов следует придерживаться общепринятых соглашений: идентификатор переменной обычно пишется строчными буквами; идентификатор типа или функции начинается с заглавной, все остальные – строчные; константа задается заглавными буквами. Идентификатор должен нести смысл, поясняющий назначение именованного объекта в программе. Он может состоять из нескольких слов, разделенных символом подчеркивания (group_number), или каждое следующее слово – с прописной буквы (GroupNumber).

Несмотря на то, что разрешается использовать имена переменных, начинающиеся со знака подчеркивания или двух знаков подчеркивания

(_this, __that), лучше их избегать. Это может вызвать конфликт со стандартными системными переменными, имеющими такую же форму записи.

Ключевые слова

Ключевые слова – это зарезервированные идентификаторы, которые наделены определенным смыслом. Трансляторы языков C/C++, соответствующие требованиям стандарта ANSI, воспринимают только служебные слова, записанные строчными буквами. Не следует использовать имена объектов (идентификаторы), совпадающие со служебными словами. Далее приведены списки ключевых слов.

Ключевые слова для C/C++

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while
inline (C99)		restrict (C99)	

Ключевые слова только для C++

asm	bool	catch	class
const_cast	delete	dynamic_cast	explicit
false	friend	virtual	mutable
namespace	new	operator	private
protected	public	reintepret_cast	throw
static_cast	template	this	typename
true	try	typeid	
using	wchar_t		

Описание ключевых слов содержится в прил. 3.

В дополнение к этому, идентификаторы, содержащие двойное подчеркивание (__) резервируются для реализаций C++ и стандартных библиотек. Пользователи не должны употреблять их.

В представлении программы на C++ в кодировке ASCII используются в качестве операций или разделителей следующие символы:

!	*	=	[ ]	:	?
%	( )	{ }	\	"	,
^	–	|	;	<	.
&	+	~	'	>	/

а следующие комбинации символов используются для задания операций:

->	->*	>=	||	+=	&=
++	<<	==	*=	-=	^=
--	>>	!=	/=	<<=	|=
.*	<=	&&	%=	>>=	::

Каждая операция считается отдельной лексемой. В дополнение к этому, символы # и ## резервируются для препроцессора.

Константы

Константы, в отличие от переменных, являются фиксированными значениями, которые можно вводить и использовать на языках C/C++.

Разделяют четыре типа констант: целые константы, константы с плавающей запятой, символьные константы и строковые литералы.

Целые константы не имеют дробной части и не содержат десятичной точки. Они представляют целую величину в одной из следующих форм: десятичной, восьмеричной или шестнадцатеричной. Десятичная константа состоит из одной или нескольких десятичных цифр, причем первая цифра не должна быть нулем (в противном случае число будет воспринято как восьмеричное). Восьмеричная константа состоит из обязательного нуля и одной или нескольких восьмеричных цифр. Шестнадцатеричная константа начинается с обязательной последовательности `0x` или `0X` и содержит одну или несколько шестнадцатеричных цифр. Примеры целых констант:

Десятичная константа	Восьмеричная константа	Шестнадцатеричная константа
16	020	0x10
127	0117	0x2B
240	0360	0xF0

Целые константы могут быть обычной длины или длинные. Длинные целые константы оканчиваются буквой `l` или `L` (например: `5l, 6l, 128L, 0105L, OX2A11L`). Размер целых констант обычной длины зависит от реализации (для тридцатидвухразрядного процессора – 4 байта). Длинная целая константа всегда занимает 4 байта.

При использовании десятичной целой константы старший бит числа интерпретируется как знаковый (0 – положительное число, 1 – отрицательное). Для восьмеричных и шестнадцатеричных целых констант возможно представление только положительных чисел и нуля, поскольку старший разряд рассматривается как часть кода числа, а не как его знак.

Диапазон значений десятичных констант обычной длины для тридцатидвухразрядного процессора – (-2^{31}…+ (2^{31} – 1)). Диапазон значений восьмеричных и шестнадцатеричных констант обычной длины для тридцатидвухразрядного процессора – (0...2^{32} – 1).

Диапазон значений длинных десятичных констант не зависит от разрядности процессора и составляет (-2^{31}…+ (2^{31} – 1)). Диапазон значений длинных восьмеричных и шестнадцатеричных констант также не зависит от разрядности процессора и составляет (0…2^{32} – 1).

Константа с плавающей точкой – десятичное число, представленное в виде действительной величины с десятичной точкой или экспонентой. Формат имеет вид:

```
[ цифры ].[ цифры ] [ E|e [+|-] цифры ]
```

Число с плавающей точкой состоит из целой и дробной части и (или) экспоненты. Константы с плавающей точкой представляют положительные величины удвоенной точности (имеют тип `double`). Для определения отрицательной величины необходимо сформировать константное выражение, состоящее из знака минуса и положительной константы: `115.75, 1.5E-2, -0.025, .075, -0.85E2.`

Или целая часть, или дробная часть (но не обе) могут отсутствовать. Или точка, или символ e (или E) вместе с показателем могут отсутствовать (но не оба). Если число записывается без них, то получаем целое.

В языке C++, когда в конце константы с плавающей точкой отсутствуют буквы f, F, l, L, константа имеет тип `double` (8 байт или 64 бита с диапазоном значений ($\pm 1{,}7 \cdot 10^{-308}$…$\pm 1{,}7 \cdot 10^{+308}$). Если же константа заканчивается буквой f или F, то она имеет тип `float`, занимает 4 байта и диапазон значений ($\pm 3{,}4 \cdot 10^{-38}$…$\pm 3{,}4 \cdot 10^{+38}$). Аналогичным образом, при завершении константы буквами l или L, константа имеет тип `long double`, занимает 10 байт с диапазоном значений ($\pm 3{,}4 \cdot 10^{-4932}$…$\pm 3{,}4 \cdot 10^{+4932}$).

Символьные константы можно разделить на две группы: печатные и непечатные символы. Символьная константа в языках C/C++ состоит либо из одного печатного символа, заключенного в апострофы (`' '`, `'Q'`) , либо специального и управляющего кода, заключенного в апострофы (`'\n'`, `'\\'`). Управляющие коды представляют непечатные символы (см. табл. 4.2). Символьная константа рассматривается как символьный беззнаковый тип данных с диапазоном значений от 0 до 255. Часто используется константа `'\0'`, которая называется нулевым символом или нулевым байтом.

Символьная константа, которой непосредственно предшествует буква `L`, является широкой символьной константой, например, `L'ab'`.

Такие константы имеют тип `wchar_t`, являющийся целочисленным типом. Широкие символы предназначены для такого набора символов, где значение символа не помещается в один байт.

Строковая константа (литерал) – последовательность символов (включая строковые и прописные буквы русского и латинского алфавитов, а также цифры), заключенная в кавычки (""): `"город Тамбов"`, `"YZPT КОД\72"`. Для запоминания строковых констант используется по одному байту на каждый символ строки и автоматически добавляется к ней признак конца строки, которым служит нулевой байт `'\0'`. Нулевой байт является ограничителем строки.

Для составления строковых констант можно использовать любые печатные символы или управляющие коды.

Строка литералов, перед которой непосредственно идет символ L, считается широкосимвольной строкой, например, `L"asdf"`. Такая строка имеет тип «массив элементов типа `wchar_t`», где `wchar_t` – целочисленный тип.

Ключевое слово `const` – модификатор типа, указывающий, что переменная является константой. Описание констант начинается с ключевого слова const, далее указывается тип и значение:

```
const int Size=2;
```

Поскольку константе ничего нельзя присвоить, она должна быть инициализирована. Описание чего-нибудь как `const` гарантирует, что его значение не изменится в области видимости:

```
Size=2;    //ошибка
```

Раз константе нельзя присвоить значение, она должна быть инициализирована в месте своего определения. Определение константы без ее инициализации также вызывает ошибку компиляции:

```
const double pi; //ошибка: неинициализированная константа
```

Кроме констант в программе могут использоваться константные выражения.

Комментарии

Комментарий – это набор символов, которые игнорируются компилятором. Введение комментария начинается с символов `/*` и заканчивается символами `*/`. Все, что помещено между ними, игнорируется:

```
/*Эта программа выводит сообщение на экран*/
```

Комментарии /* */ не могут быть вложенными. В C++ используется пара символов //, указывающая начало строки комментария. В этом случае концом комментария считается конец строки, так что нет необходимости отмечать его специальным символом:

```
//Эта программа выводит сообщение на экран
```

Этот способ наиболее полезен для коротких комментариев.

4.3. Структура программы

C/C++-программа – совокупность одного или нескольких модулей. Модулем является самостоятельно транслируемый файл, который обычно содержит одну или несколько функций. Функция состоит из операторов языка.

Термин «функция» в языках C/C++ охватывает понятия «подпрограмма», «процедура» и «функция», используемые в других языках программирования. C/C++-программа может содержать одну функцию (главная функция main) или любое количество функций. Фактически все программы на ISO/ANSI C++ начинаются с главной функции main. Microsoft также называет эту функцию wmain, когда применяется кодировка символов Unicode и имя _tmain определено либо как main, либо как wmain. Другие функции могут быть вызваны из функции main или из какой-либо другой функции в процессе выполнения программы. Эти функции могут находиться в том же модуле (файле), что и функция main, или в других модулях.

Главная функция выглядит следующим образом:

```
void _tmain(){ /* ... */ }
```

В круглых скобках main перечисляются аргументы или параметры функции (в данном случае аргументов нет). У функции может быть результат или возвращаемое значение. Если функция не возвращает никакого значения, то это обозначается ключевым словом void. В фигурных скобках ({...}) записывается тело функции – действия, которые она выполняет. Пустые фигурные скобки означают, что никаких действий не производится.

При определении функции с аргументами:

```
int _tmain(int argc, char* argv[]) { /* ... */ }
```

argc задает число параметров, передаваемых программе окружением. Если argc не равно нулю, параметры должны передаваться как строки,

завершающиеся символом '\0', с помощью argv[0] до argv[argc-1], причем argv[0] должно быть именем, под которым программа была запущена, и должно гарантироваться, что argv[argc]==0.

В общем случае программа на C/C++ имеет следующую структуру:

```
#директивы препроцессора
. . . . . .
#директивы препроцессора
описание прототипов функций
определение глобальных переменных
функция a ( )
   {операторы}
void main ( ) //функция, с которой начинается выполнение
                 программы
{  описания
   присваивания
          операторы
          вызов функции a
          вызов функции b
          оператор пустой
                составной
                выбора
                циклов
                перехода
}
функция b ( )
   {операторы}
```

До компилятора исходный текст обрабатывается препроцессором – специальной программой, которая модифицирует текст программы по специальным командам – директивам. Программа начинается с директив препроцессора, с символа #.

Далее размещается блок определения данных, за которыми следуют операторы функции. Определения данных создают переменные, которые будут использованы в функции. Операторы задают действия, которые должны быть выполнены над переменными.

Все элементы данных должны быть определены перед их использованием. Определения данных и операторы всегда завершаются точкой с запятой. Этот символ помечает конец оператора, а не конец строки. Следовательно, один оператор может распространяться на несколько строк, если это помогает понять код, либо несколько операторов могут находиться в одной строке. Оператор программы – базовый элемент, определяющий то, что делает программа.

4.4. Фундаментальные типы данных и переменные

Программа оперирует информацией, представленной в виде различных объектов и величин. С точки зрения архитектуры компьютера, переменная – это символическое обозначение ячейки оперативной памяти программы, в которой хранятся данные. Доступ к значению возможен через имя переменной, а доступ к участку памяти – по его адресу. Каждая переменная перед использованием в программе должна быть объявлена, т.е. ей должна быть выделена память. Размер участка памяти, выделяемой для переменной, и интерпретация содержимого зависят от типа, указанного в определении переменной. Тип переменной изменить нельзя. Простейшая форма объявления переменных:

```
тип список_имен_переменных;
```

Тип переменной указывает компилятору языка C++, сколько памяти надо выделить для размещения объекта. Кроме того, он указывает компилятору, каким образом надо интерпретировать значение, содержащееся в объекте.

В C++ выделяют следующие категории типов: базовые (стандартные или основные, приведены в табл. 4.3) и производные (определяемые) типы. Базовые типы имеют имена, которые являются ключевыми словами языка. Производные типы определяются на основе базовых и делятся на скалярные (перечисления, указатели и ссылки) и структурные (массивы, структуры, объединения и классы). Дерево классификации приведено на рис. 4.1.

Таблица 4.3

Базовые типы данных

Тип данных	Назначение	Размер, байт	Диапазон значений
`char`	Для символа	1	–128…+127
`wchar_t`	Для символа из расширенного набора	2	–32 768…+32 767
`int`	Для целого значения	4	–2 147 483 648… 2 147 483 647
`bool`	Для логического значения	1	`false, true`
`float`	Для значения с плавающей точкой	4	$\pm 3{,}4 \cdot 10^{-38}$… $\pm 3{,}4 \cdot 10^{+38}$ (7 цифр)
`double`	Для значения с плавающей точкой удвоенной точности	8	$\pm 1{,}7 \cdot 10^{-308}$…$\pm 1{,}7 \cdot 10^{+308}$ (15 цифр)

Происхождение и перевод служебных слов:

- `char` (CHARacter: буква, символ);
- `wchar_t` (Wide CHaracter Type: расширенный символьный тип);
- `int` (INTeger: целое число);
- `float` (число с плавающей точкой).

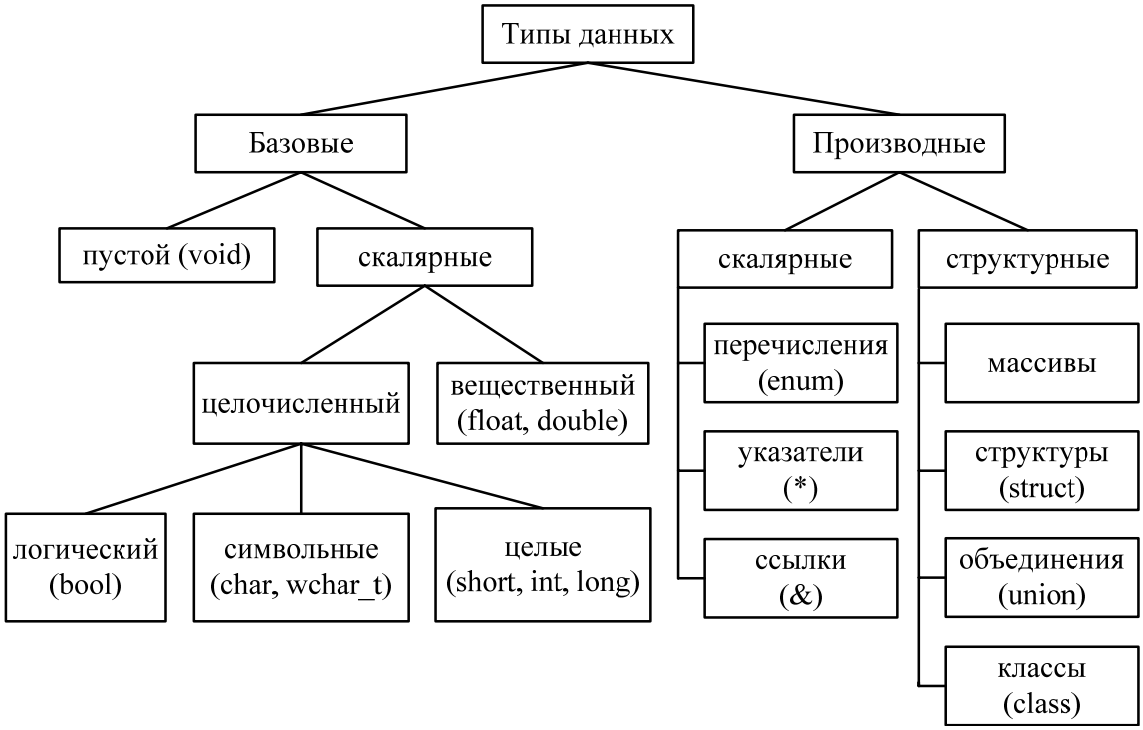


Рис. 4.1. Дерево классификации типов данных

Существуют четыре спецификатора типа (табл. 4.4), уточняющие внутреннее представление и диапазон значений стандартных типов: `unsigned` (без знака), `signed` (со знаком), `short` (короткий), `long` (длинный).

В стандартном заголовочном файле `<limits.h>` задаются в зависимости от реализации минимальные и максимальные значения каждого типа.

Переменным можно присваивать начальные значения, явно указывая их в определениях, этот прием называется *инициализацией*:

```
тип имя_переменной = начальное_значение;
```

Символьный тип (`char`)

Под величину символьного типа отводится один байт, что позволяет хранить в нем любой символ из 256-символьного набора ASCII (American Standart Code for Information Interchange):

```
char          A;
char          B;

A = 'D';
B = '!';
```

Таблица 4.4

Уточняющие спецификаторы типа

Тип данных	Назначение	Размер, байт	Диапазон значений
`unsigned char`	Для байта с неотрицательным целым значением	1	0…255
`unsigned`, `unsigned int`	Для неотрицательного целого значения	4	0…4 294 967 295
`short`, `short int`, `signed short int`	Для короткого целого значения	2	–32 768…+32 767
`unsigned short`, `unsigned short int`	Для беззнакового короткого целого	2	0…65 535
`long`, `long int`, `signed long int`	Для длинного целого	4	–2 147 483 648… 2 147 483 647
`unsigned long`, `unsigned long int`	Для беззнакового целого длинного	4	0…4 294 967 295
`long long`, `signed long long`	Для удвоенного длинного целого	8	–9 223 372 036 854 775 808… 9 223 372 036 854 775 807
`unsigned long long`	Для беззнакового удвоенного длинного целого	8	0…18 446 744 073 709 551 615
`double` `long float`	Для вещественного с двойной точностью	8	$\pm 1{,}7 \cdot 10^{-308} \ldots \pm 1{,}7 \cdot 10^{+308}$
`long double`	Для длинного вещественного	10	$\pm 3{,}4 \cdot 10^{-4932} \ldots \pm 3{,}4 \cdot 10^{+4932}$

Расширенный символьный тип (`wchar_t`)

Этот тип предназначен для работы с набором символов, для кодировки которого недостаточно одного байта (например, для набора Unicode). Символьные и строковые константы с типом `wchar_t` записываются с префиксом `L`.

Пример

```
wchar_t  letter = L'D';
```

Целые типы

Внутреннее представление величины целого типа – целое число в двоичном коде. При использовании спецификатора `signed` старший бит числа интерпретируется как знаковый (0 – положительное число, 1 – отрицательное). Спецификатор unsigned позволяет представлять только положительные числа, поскольку старший разряд рассматривается как часть кода числа. По умолчанию все целочисленные типы считаются знаковыми, то есть спецификатор `signed` можно опускать.

```
unsigned int   Stavka;
int            Symma;
```

Логический тип (`bool`)

Величины логического типа могут принимать только значения `false` и `true`, которые являются служебными словами. Внутренняя форма представления значения `false` – 0 (нуль). Любое другое значение интерпретируется как `true`. При преобразовании к целому типу `true` имеет значение 1.

Типы с плавающей точкой (`float, double`)

Внутреннее представление для типов с плавающей точкой состоит из двух частей – мантиссы и порядка. При этом величины типа `float` занимают четыре байта, из которых один двоичный разряд отводится под знак мантиссы, 8 разрядов под порядок и 23 под мантиссу. Мантисса – число большее 1.0, но меньшее 2.0. Поскольку старшая цифра мантиссы всегда равна 1, то она не хранится.

Для величин типа `double`, занимающих восемь байт, под порядок и мантиссу отводятся соответственно 11 и 52 разряда. Длина мантиссы определяет точность числа, а длина порядка – диапазон числа.

```
float pi = 3.14 , cc = 1.3456;
unsigned int year = 1999;
```

Тип (`void`)

Кроме перечисленных, к основным типам языка относится тип `void`, но множество значений этого типа пусто. Он используется для определения функций, которые не возвращают значения, для указания пустого списка аргументов функции, как базовый тип указателей и в операции приведения типов.

Чтобы проверить размер памяти, выделяемой для объекта данного типа, можно написать программу, использующую операцию `sizeof`.

Значением этой операции является размер любого объекта или спецификации типа, выраженный в байтах:

```
#include "stdafx.h"
#include <iostream>
#include <locale>

int _tmain(int argc, _TCHAR* argv[])
{
   setlocale (LC_CTYPE, "Russian");
   using std::cout;
   using std::endl;

   //размерность
   cout<<"Тип объекта Его размер в байтах" << endl;
   cout<<" Размерность int: "<<sizeof(int)<<"\n";
   cout<<" Размерность char: "<<sizeof(char)<<"\n";
   cout<<" Размерность unsignid: "<<sizeof(unsigned)<<"\n";
   cout<<" Размерность long: "<<sizeof(long)<<"\n";
   cout<<" Размерность float: "<<sizeof(float)<<"\n";
   cout<<" Размерность double: "<<sizeof(double)<<"\n";
   cout<<"\n";

   return 0;
}
```

Результат работы программы представлен на рис. 4.2.

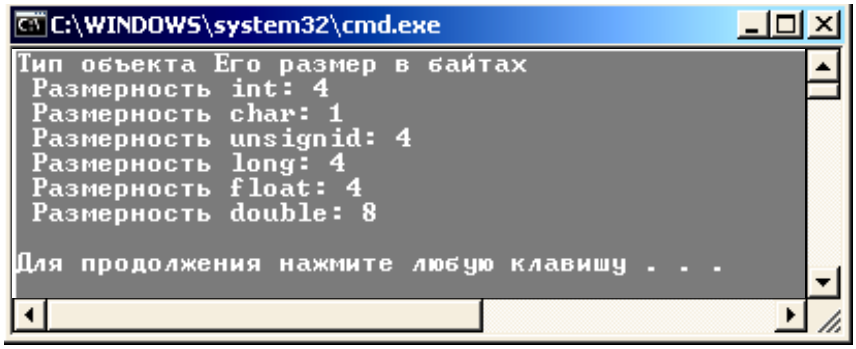


Рис. 4.2. Консольное окно с выводом результатов

Вторая строка программы `#include <iostream>` является директивой препроцессора. Препроцессор – это специальная программа, которая обрабатывает строки программы, начинающиеся со знака `#`. Данная строка дает указание препроцессору перед компиляцией программы включить в нее информацию, содержащуюся в файле, в данном случаем `<iostream>`. Файл `<iostream>` называется файлом

заголовков. Он понадобится для того, чтобы можно было использовать операторы ввода и вывода C++.

Мастер *Application Wizard* сгенерировал заголовочный файл `stdafx.h` (первая строка) как часть проекта. Там содержатся еще две директивы `#include` для заголовочных файлов стандартной библиотеки `stdio.h` и `tchar.h`, заголовок старого стиля для ввода-вывода – `stdio.h`, он обеспечивает ту же функциональность, что и `<iostream>`. Второй файл, `tchar.h`, является специальным заголовочным файлом Microsoft, опеределяющим некоторые текстовые функции.

Пространство имен `std`

Пространство имен (`namespace`) – это механизм, предназначенный для предотвращения проблем, связанных с дублированием имен. Определенное множество имен, вроде имен стандартной библиотеки, ассоциируется с общим именем, которое и представляет собой пространство имен. Все средства стандартной библиотеки C++ определены внутри пространства имен по имени `std`, поэтому каждый элемент стандартной библиотеки имеет свое собственное имя плюс наименование пространства имен `std` в качестве квалификатора. Имена `cout` и `endl` определены в стандартной библиотеке, поэтому их полные имена выглядят как `std::cout` и `std::endl`. Два двоеточия, отделяющие имя пространства имен от имени элемента, образуют операцию, называемую операцией *разрешения контекста*. Применение полных имен в программе делает код громоздким, поэтому можно использовать *объявление* `using`, сообщающее компилятору о намерении использовать имена из пространства имен `std` без указания наименования пространства имен: `using std::cout; using std::endl;`.

Кроме того, программа может содержать следующую *директиву* `using`:

```
using namespace std;
```

Теперь все имена из пространства `std` импортируются в исходный файл, и можно ссылаться на все, что определено в этом пространстве имен, без квалификационного имени. То есть можно писать `cout` вместо `std::cout`. Недостатком такого применения может быть возникновение непреднамеренных конфликтов.

Имя `cout` представляет стандартный выходной поток, который по умолчанию соответствует командной строке, а имя `endl` – символьной строке. Строка `cout<<"  ";` – оператор вывода, с помощью которого выводится на экран дисплея фраза, заключенная в кавычки.

Функция может возвращать значение в программу с помощью оператора возврата (`return`). Этот оператор также прекращает выполнение функции `main()` и завершает программу. Управление возвращается операционной системе.

Определение синонимов для типов данных

Ключевое слово `typedef` позволяет определить свое собственное имя для существующего типа данных. Альтернативное имя может использоваться наряду со встроенным именем типа:

```
typedef long int BigInt;       //определение синонима
BigInt    my_number = 0L;      //определение перменной
```

Это средство играет важную роль в упрощении сложных объявлений, что делает код более читабельным.

4.5. Время существования и область видимости переменных

В дополнение к имени и типу объекта существуют еще два атрибута: область действия и время жизни объекта. Эти характеристики взаимосвязаны и существенно влияют на возможности использования переменной в программе. Взаимосвязь определяется способом выделения памяти для переменной.

Областью действия объекта (данного) называется та часть программы, в которой можно пользоваться этим объектом. В частности, областью действия может быть:

– блок операторов (`{...}`);

– модуль (файл);

– вся программа в целом.

Временем жизни данного называется отрезок времени, в течение которого значение этого данного доступно в некоторой части программы. Бывает локальное (при выполнении блока, в котором оно объявлена) и глобальное (при выполнении всей программы) время жизни.

В языках C/C++ область действия и время жизни объекта определяются местом объявления переменной и классом хранения (модификатором). Можно использовать следующие классы: внешний; внешний статический; внутренний статический; автоматический; регистровый.

Автоматическая (`auto`) переменная или константа имеет локальную область действия и известна только внутри блока, в котором она определена. Для переменной выделяется временная память при входе

в блок, а при выходе уничтожается. По умолчанию переменная в блоке считается автоматической.

Регистровая (`register`) переменная храниться в регистре процессора, и, соответственно, доступ к ней быстрее, чем к автоматической переменной. При отсутствии свободных регистров регистровая переменная становится автоматической.

Внешняя (`extern`) переменная является глобальной переменной. Спецификатор `extern` информирует компилятор, что переменная будет объявлена (без `extern`) в другом файле, где ей будет выделена память.

Статической (`static`) переменной (константе) выделяется память после ее объявления и сохраняется до конца выполнения программы. Статические переменные при объявлении по умолчанию инициализируются нулевыми или пустыми значениями.

В табл. 4.5 приведены характеристики переменных, объявленных с использованием модификаторов выделения памяти.

Таблица 4.5

Характеристика модификаторов выделения памяти

Модификатор выделения памяти	Место объявления переменной	Область видимости	Время существования	Обобщенная характеристика
Не указан (по умолчанию `static`)	Вне функции	До конца файла	Глобальное	Глобальная переменная файла
Не указан (по умолчанию `auto`)	В функции	До конца блока	Локальное	Локальная переменная
`auto`	Запрещено вне функции	–	–	–
`auto`	В функции	До конца блока	Локальное	Локальная переменная
`register`	Запрещено вне функции	–	–	–
`register`	В функции	До конца блока	Локальное	Быстрая локальная переменная
`static`	Вне функции	До конца файла	Глобальное	Глобальная переменная файла
`static`	В функции	До конца блока	Глобальное	Сохраняемая локальная переменная
`extern`	Вне функции	До конца файла	Глобальное	Глобальная переменная программы

4.6. Базовые операции ввода-вывода

Потоковый ввод-вывод

Обмен данными между программой и внешними устройствами осуществляется с помощью операций ввода-вывода. Классы потокового ввода-вывода C++ определены в файле заголовков `#include <iostream>`. Библиотека потоков ввода-вывода определяет: `cout` – стандартный поток вывода (экран дисплея), `cin` – стандартный поток ввода (связан с клавиатурой), `cerr`, `clog` – стандартный поток сообщений об ошибках.

Вывод осуществляется с помощью операции >>, ввод – с помощью операции <<. Выражение:

```
using std::cout;
.............................
cout << "Пример вывода: " << 34;
```

напечатает строку `"Пример вывода: "`, за которой будет выведено число 34. Используя один стандартный поток вывода `cout`, можно отобразить несколько аргументов. Между собой аргументы разделяются операторами вставки:

```
using std::cout;
.............................
int age;
    age = 43;
    cout << "Вам исполнилось " << age << " года.";
```

Выражение:

```
using std::cin;
.............................
int x;
cin >> x;
```

введет целое число с консоли в переменную `x` (для того, чтобы ввод произошел, нужно напечатать число и нажать клавишу **Enter**.)

Часто бывает необходимо вывести строку или число в определенном формате. Для этого используются манипуляторы – объекты особых типов, которые управляют тем, как обрабатываются следующие аргументы:

```
endl                - при выводе перейти на новую строку;
ends                - вывести нулевой байт (признак конца
                      строки символов);
dec                 - выводить числа в десятичной системе
                      (по умолчанию);
```

```
oct                  - выводить числа в восьмеричной системе;
hex                  - выводить числа в шестнадцатеричной
                       системе счисления;
setw(int n)          - установить ширину поля вывода в n
                       символов (n целое);
setfill(int n)       - установить символ-заполнитель; этим
                       символом выводимое значение будет
                       дополняться до необходимой ширины;
setprecision(int n) - установить количество цифр после
                       запятой при выводе вещественных чисел;
setbase(int n)       - установить систему счисления для вывода
                       чисел; n может принимать значения 0, 2,
                       8, 10, 16, причем 0 означает систему
                       счисления по умолчанию, т.е. 10.
```

Для использования манипуляторов их надо вывести в выходной поток. Кроме того, в символьную строку, заключенную в двойные кавычки, можно включать управляющие последовательности (escape sequences):

```
using namespace std;
.............................
int x = 53;

cout << "Десятичный вид: \t " << dec << x << endl
     << "Восьмеричный вид: \t " << oct << x << endl
     << "Шестнадцатеричный вид: \t " << hex << x << endl;
```

Ниже приведен пример использования манипуляторов с параметрами для вывода:

```
using namespace std;
.............................
double x;  //вывести число в поле общей шириной 6 символов
           //(3 цифры до запятой, десятичная точка
           //и 2 цифры после запятой)
cout << setw(6) << setprecision(2) << x << endl;
```

Те же манипуляторы (за исключением endl и ends) могут использоваться и при вводе:

```
#include <iomanip>
.............................
using namespace std;
.............................
int x;
cin >> hex >> x; //ввести шестнадцатеричное число
```

Манипуляторы определены в заголовочном файле <iomanip>, поэтому при их использовании надо добавлять директиву #include<iomanip>.

Форматированный ввод-вывод

Обмен данными реализуется с помощью библиотеки функций ввода-вывода: #include <stdio.h>.

Функция printf() позволяет выводить на дисплей данные всех типов, работать со списком из нескольких аргументов и определять способ форматирования данных:

```
printf ( <форматная строка>,<список аргументов>);
```

<форматная строка> – строка символов, заключенных в кавычки, которая показывает, каким образом должны быть напечатаны аргументы. Например:

```
printf ("Значение числа Пи равно  %f\n", pi);
```

Форматная строка может содержать символы, печатаемые текстуально, спецификации преобразования, управляющие символы.

Каждому аргументу соответствует своя спецификация преобразования, которая начинается с символа процента (%), после которого стоит буква, указывающая тип данных:

```
%d – десятичное целое число;
%f – вещественное число типа float или double;
%c – символ;
%s – строка;
%p – указатель;
%u – беззнаковое целое число;
%o – целые числа в восьмеричной системе счисления;
%x – целые числа в шестнадцатеричной системе счисления;
%e – вещественное число в экспоненциальной форме.
```

В модификаторах формата после символа % можно указывать строку цифр, задающую минимальную ширину поля вывода, например: %5d (для целых), %4.2f (для вещественных – две цифры после запятой для поля шириной 4 символа). Если указанной ширины не хватает, происходит автоматическое расширение.

Можно управлять перемещением курсора на экране и выполнять некоторые другие функции, используя управляющие коды, называемые escape-последовательностями. Последовательность начинается с символа обратной наклонной черты (\):

```
\n – перемещает курсор в начальную позицию следующей строки;
\t – перемещает курсор в следующую позицию табуляции экрана;
```

```
\r – выполняет «возврат каретки», перемещая курсор к началу
     той же строки без перехода на следующую;
\b – передвигает курсор только на одну позицию влево.
```

Функция `scanf()` является многоцелевой функцией, дающей возможность вводить в компьютер данные любых типов. Указатели формата аналогичны тем, которые используются функцией `printf()`.

```
scanf (<форматная строка>,<список аргументов>);
```

В качестве аргументов используются указатели объектов `&`. Например:

```
scanf(" %d%f ", &x,&y);
```

Если нужно ввести значение строковой переменной, то использовать символ & не нужно. Строка – массив символов, а имя массива эквивалентно адресу его первого элемента:

```
char name[20];
.............................
scanf ("%s", name);
```

Строковый и символьный ввод-вывод

Функция `puts()` осуществляет вывод информации на экран. Требует подключения `#include <stdio.h>`. Параметром функции может быть строка:

```
puts("Всем привет!");
```

строковая константа:

```
#define MESSAGE "Всем привет"
puts(MESSAGE);
```

или строковая переменная:

```
char greeting[] = "Всем привет";
puts(geering);
```

Функция `putchar()` предназначена для вывода единичного символа на экран. Параметром функции может быть символьный литерал:

```
putchar('H');
```

символьная константа:

```
#define INITIAL 'H'
putchar(INITIAL);
```

или символьная переменная:

```
char letter;
letter='G';
putchar(letter);
```

Функция `gets()` вводит строку в переменную:

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

Функция `getchar()` вводит с клавиатуры единичный символ:

```
int letter;
letter = getchar();
```

4.7. Практические задания

1. Написать программу, осуществляющую ввод на консоль переменных всех рассмотренных типов данных и форматированный вывод введенных переменных.

2. Разработать программу, осуществляющую вывод на консоль диапазона базовых типов данных.

3. Составить программу, осуществляющую вывод на консоль переменных в 2, 16, 10, 8 с/с.

4. Написать программу, осуществляющую вывод на консоль звезды, треугольника и круга, закрашенных введенным пользователем сиволом.

Глава 5. ИСПОЛЬЗОВАНИЕ ОСНОВНЫХ ОПЕРАЦИЙ И ВЫРАЖЕНИЙ ЯЗЫКА C/C++. СТАНДАРТНЫЕ ФУНКЦИИ И ДИРЕКТИВЫ ПРЕПРОЦЕССОРА

5.1. Операции и выражения C/C++

Классификация операций

Операции бывают *унарные* (воздействуют на одно значение или выражение), *бинарные* (участвуют два выражения) и *тернарные* (три выражения). Основные унарные операции приведены в табл. 5.1. Унарные операции выполняются справа налево.

Таблица 5.1

Унарные операции

Операции	Назначение
`&`	Получение адреса операнда
`*`	Обращение по адресу (разыменование)
`–`	Унарный минус, меняет знак арифметического операнда
`~`	Поразрядное инвертирование внутреннего двоичного кода (побитовое отрицание)
`!`	Логическое отрицание (НЕ). В качестве логических значений используется 0 – ложь и не 0 – истина, отрицанием 0 будет 1, отрицанием любого ненулевого числа будет 0
`++`	Инкремент или увеличение на единицу: префиксная операция – увеличивает операнд до его использования, постфиксная операция увеличивает операнд после его использования.
`--`	Декремент или уменьшение на единицу: префиксная операция – уменьшает операнд до его использования, постфиксная операция уменьшает операнд после его использования
`sizeof`	Вычисление размера (в байтах) для объекта того типа, который имеет операнд

Основные бинарные операции приведены в табл. 5.2, они выполняются слева направо.

Таблица 5.2

Бинарные операции

Операции	Назначение	
`+`	Бинарный плюс (сложение арифметических операндов)	Аддитивные
`–`	Бинарный минус (вычитание арифметических операндов)	

Окончание табл. 5.2

Операции	Назначение	
*	Умножение операндов арифметического типа	Мультипликативные
/	Деление операндов арифметического типа (если операнды целочисленные, то выполняется целочисленное деление)	
%	Получение остатка от деления целочисленных операндов (операция по модулю)	
<<	Сдвиг влево битового представления значения левого целочисленного операнда на количество разрядов, равное значению правого операнда	Операции сдвига (определены только для целочисленных операндов)
>>	Сдвиг вправо битового представления значения правого целочисленного операнда на количество разрядов, равное значению правого операнда	
&	Поразрядная конъюнкция (И) битовых представлений значений целочисленных операндов	Поразрядные операции
|	Поразрядная дизъюнкция (ИЛИ) битовых представлений значений целочисленных операндов	
^	Поразрядное исключающее ИЛИ битовых представлений значений целочисленных операндов	
<	Меньше, чем	Операции сравнения
>	Больше, чем	
<=	Меньше или равно	
>=	Больше или равно	
==	Равно	
!=	Не равно	
&&	Конъюнкция (И) целочисленных операндов или отношений, целочисленный результат ложь (0) или истина (1)	Логические бинарные операции
||	Дизъюнкция (ИЛИ) целочисленных операндов или отношений, целочисленный результат ложь (0) или истина (1)	
=	Присваивание, присвоить значение правого операнда левому	Операции присваивания и составного присваивания
+=	Выполнить соответствующую операцию с левым операндом и правым операндом и присвоить результат левому операнду	
−=		
*=		
/=		
%=		
|=		
&=		
^=		
<<=		
>>=		

Приоритеты операций приведены в табл. 5.3.

Таблица 5.3

Приоритеты операций

Ранг	Операции
1	`( ) [ ] -> .`
2	`! ~ - ++ -- & *` (тип) `sizeof` тип ( )
3	`* / %` (мультипликативные бинарные)
4	`+ -` (аддитивные бинарные)
5	`<< >>` (поразрядного сдвига)
6	`< > <= >=` (отношения)
7	`== !=` (отношения)
8	`&` (поразрядная конъюнкция «И»)
9	`^` (поразрядное исключающее «ИЛИ»)
10	`|` (поразрядная дизъюнкция «ИЛИ»)
11	`&&` (конъюнкция «И»)
12	`||` (дизъюнкция «ИЛИ»)
13	`?:` (условная операция)
14	`= *= /= %= -= &= ^= |= <<= >>=` (операция присваивания)
15	`,` (операция запятая)

В отличие от унарных и бинарных операций в *тернарных условных операциях* используется три операнда:

```
Выражение1 ? Выражение2 : Выражение3;
```

Первым вычисляется значение выражения 1. Если оно истинно, то вычисляется значение выражения 2, которое становится результатом. Если при вычислении выражения 1 получится 0, то в качестве результата берется значение выражения 3.

```
x<0 ? -x : x ; //вычисляется абсолютное значение x
```

Выражения

Комбинация знаков операций и операндов, результатом которой является определенное значение, называется *выражением*. Каждый операнд в выражении может быть выражением. Значение выражения зависит от расположения знаков операций и круглых скобок в выражении, а также от приоритета выполнения операций. Каждое выражение состоит из одного или нескольких операндов, символов операций и ограничителей:

```
X * 12 + Y
val < 3
-9
```

Выражение, после которого стоит точка с запятой – это оператор-выражение.

Рассмотрим подробнее некоторые операции и варианты их использования.

Операция присваивания (=) рассматривается как выражение, имеющее значение левого операнда после присваивания. Присваивание может включать несколько операций присваивания, изменяя значения нескольких операндов, например:

```
long a; char b; int c, x, y, z;
a = b = c;                  //эквивалентно b = c; a = b;
x = i + (y = 3) - (z = 0);
                            //z = 0, y = 3, x = i + y - z;
```

Недопустимыми являются: *присваивание константе, присваивание функции и присваивание результату операции*.

```
2 = a + b;           //ошибка
getch() = a;         //ошибка
(a + 1) = 2 + b;     //ошибка
```

Операции инкремента и декремента (++, --) относятся к унарным арифметическим операциям, которые служат соответственно для увеличения или уменьшения значения, хранимого в переменной целого типа. Например, следующие три оператора дадут один и тот же эффект:

```
sum = sum + 1;
sum += 1;
++sum;
```

Операции инкремента и декремента не только изменяют значения переменных, но и возвращают значения. Таким образом, их можно сделать частью более сложного выражения:

```
int  i, j, a;
float m, y;
.....................................
     m *= y;         //m = m * y;
     i += 2;         //i = i + 2;
     m /= y + 1;     //m = m / (y + 1);
     --a;            //a = a - 1;
     j = i++;        //j = i; i = i + 1;
     j = ++i;        //i = i + 1; j = i;
```

Имеется *постфиксная* и *префиксная* форма операторов инкремента и декремента. В постфиксной форме записи переменная, к которой

применена операция, увеличивается (или уменьшается) только после того, как ее значение будет использовано в контексте.

Типичной ошибкой является попытка использовать в операции инкремента или декремента операнд, отличный от имени простой переменной:

```
++(x + 1); //ошибка
```

Общий вид *операций сравнения* (*отношения*):

```
<выражение 1> <знак операции> <выражение 2>
```

Выражениями могут быть любые базовые (скалярные) типы. Значения выражений перед сравнением преобразуются к одному типу. Результат операции сравнения – значение 1, если отношение истинно, или 0 в противном случае (ложно). Операция сравнения может использоваться в любых арифметических выражениях:

```
int b = 5;
int c = 10;
a = b > c;          //Запомнить результат сравнения a=0
a = (b > c)* 2;     //a= 0
```

или

```
(a < b && b < c)
                    //если ОДНОВРЕМЕННО ОБА  a < b
                    //и b < c, то истина, иначе ложь
if (a == 0 || b > 0)
                    //если ХОТЯ БЫ ОДИН a==0
                    //или b > 0, то истина, иначе ложь
!0                       //1
!10                      //0
!((x = 1) < 0)           //1
0 < x < 100              //ошибка
(0 < x) && (x < 100)     //верно
```

В C/C++ предусмотрены *битовые операции* для работы с отдельными битами. Эти операции нельзя применять к переменным вещественного типа. Операндами операций над битами могут быть только выражения, приводимые к целому типу. Операции (~, &, |, ^) выполняются поразрядно над всеми битами операндов (знаковый разряд не выделяется).

Общий вид операции инвертирования:

```
~ <выражение>
```

Остальные операции над битами имеют вид:

```
<выражение 1> <знак операции> <выражение 2>
```

Ниже приведена таблица истинности логических операций &, | и ^.

Операнд 1	Операнд 2	&	|	^
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Операция & часто используется для маскирования некоторого множества бит. Операция ! используется для включения (устанавливает в единицу те биты, которые были нулями).

Необходимо отличать побитовые операции & и ! от логических бинарных операций && и ||.

```
x = 1;
y = 2;
x & y      //результат 0, т.к. 0001 & 0010=0000
x && y     //результат 1, т.к. в операц. сравнения
           //оба операнда истина
```

Арифметические операции задают обычные действия над операндами арифметического типа.

```
i % j      //i – ( i/j ) * j
12 % 6     //0
13 % 6     //1
```

Если арифметическая операция содержит операнды различных типов, то компилятор выполняет автоматическое преобразование их типов.

Часто арифметические операции используются для обработки чисел, например:

```
int  n = 12345;
int  low, i = 6;

low = n %10;          //младшая цифра числа n
n = n / 10;           //отбросить младшую цифру числа n
if ( n % i == 0)... //определить – n делится нацело на i ?
n = n * 10 + i;       //добавить цифру i к значению числа n
```

Операции сдвига << и >> осуществляют соответственно сдвиг вправо и влево своего левого операнда на число позиций, задаваемых правым операндом. Операции сдвига выполняются также для всех разрядов с потерей выходящих за границы бит.

```
0x81 << 1       //10000001<<1=00000010=0x02
0x81 >> 1       //10000001>>1=01000000=0x40
```

Если левостоящее выражение имеет тип `unsigned`, то при сдвиге вправо освобождающиеся разряды гарантированно заполняются нулями (логический сдвиг). Выражения типа `signed` могут, но не необязательно, сдвигаться вправо с копированием знакового разряда (арифметический сдвиг). При сдвиге влево освобождающиеся разряды всегда заполняются нулями. Если правостоящее выражение отрицательно либо больше длины левостоящего выражения в битах, то результат операции сдвига не определен.

Операции сдвига вправо на *k* разрядов можно использовать для деления, а сдвиг влево – для умножения целых чисел на 2 в степени *k*:

```
x << 1;  //x * 2
x >> 1;  //x / 2
x << 3;  //x * 8
```

Операция `sizeof` выполняется на этапе компиляции программы и дает константу, которая равна числу байтов, требуемых для хранения в памяти данного объекта. Объектом может быть имя переменной, массива, структуры или просто спецификация типа.

```
int  i;

cout << sizeof(int) << sizeof(i);
```

Операторы

Основной источник операторов в программе – выражения. Любое из них, ограниченное символом `;`, превращается в оператор.

Запись действий, которые должен выполнить компьютер, состоит из операторов. При выполнении программы операторы выполняются один за другим, за исключением операторов управления, которые могут изменить последовательное выполнение программы. Различают операторы объявления переменных, операторы управления и операторы-выражения. Простейшей формой оператора является пустой оператор:

```
;
```

Он ничего не делает. Однако он может быть полезен в тех случаях, когда синтаксис требует наличие оператора, а он не нужен.

Любая последовательность операторов, заключенная в фигурные скобки `{}`, может выступать в любой синтаксической конструкции как один *составной оператор* (*блок*):

```
 {
    a = b + 2;
    b++;
  }
```

Он позволяет рассматривать несколько операторов как один. Область видимости имени, описанного в блоке, простирается до конца блока.

Операция запятая (,) используется при организации строго гарантированной последовательности вычисления выражений (используется там, где по синтаксису допустима только одна операция, и для организации множественных выражений, расположенных внутри круглых скобок). Форма записи:

```
выражение 1, …, выражение N;
```

Выражения вычисляются слева направо. Например:

```
char  X, Y;

(X = Y, Y = getch());

//присваивает переменной X значение Y, считывает символ,
//вводимый с клавиатуры, и запоминает его в Y

int i, j, k, n;

m = ( i = 1, j = i++, k = 6, n = i + j + k);
//i = 1, j = i = 1, i = 2, k = 6, n = 2 + 1 + 6, m = n = 9
```

Выражения, разделенные запятой, не должны быть присваиваниями. Ниже приведенный пример не является примером хорошего кода:

```
int one = 1, two = 2, three = 100, six = 0;

six = (++one, ++two, ++three);
//one = 2, two = 3, three = 101, six = 101
```

5.2. Преобразование типов

В операциях могут участвовать операнды различных типов, в этом случае они *неявно* преобразуются к общему типу в порядке увеличения их объема памяти, необходимого для хранения их значений. Поэтому неявные преобразования всегда идут от «меньших» объектов к «большим». Схема выполнения преобразований операндов арифметических операций:

```
short, char  →  int  →  unsigned  →  long   →  double
                                      float  →  double
```

```
int       ival;
float     fval;
double    dval;

ival + fval + dval;
//fval и iva1 преобразуются к double перед сложением
```

Значения типов `char` и `short` всегда преобразуются в `int`; если любой из операндов имеет тип `double`, то второй преобразуется в `double`; если один из операндов `long`, то другой преобразуется в `long`.

При присваивании значение правой части преобразуется к типу левой, который и является типом результата. При некорректном использовании операций присваивания могут возникнуть ошибки.

```
float x;  int i;

x = i;
i = x;
//float преобразуется в int, дробная часть отбрасывается
```

В любом выражении преобразование типов может быть осуществлено *явно*, в С для этого достаточно перед выражением поставить в скобках идентификатор соответствующего типа:

```
(тип) выражение;
```

В результате значение выражения преобразуется к заданному типу:

```
float a;
int i = 6, j = 4;

a = (i + j) / 3;                // a = 3
a = (float)(i + j) / 3;  →      // a = 3.333333
```

Эта форма является устаревшей и сохранена в стандарте С++ только для обеспечения обратной совместимости с программами, написанными для С и предыдущих версий С++.

В С++ явное преобразование типов производится при помощи следующих операторов: `static_cast`, `dynamic_cast`, `const_cast` и `reinterpret_cast`. Хотя иногда явное преобразование необходимо, оно служит потенциальным источником ошибок.

Явное преобразование типов используется для разыменования указателя `void*`, для того, чтобы избежать стандартного преобразования или выполнить вместо него собственное. Также может использоваться,

чтобы избежать неоднозначных ситуаций, в которых возможно несколько вариантов применения правил преобразования по умолчанию. Синтаксис операции явного преобразования:

```
cast-name< type >( expression);
```

Здесь `cast-name` – одно из ключевых слов `static_cast`, `const_cast`, `dynamic_cast` или `reinterpret_cast`, а `type` – тип, к которому приводится выражение `expression`. Так `const_cast` служит для трансформации константного типа в неконстантный. Любое иное использование `const_cast` вызывает ошибку компиляции, как и попытка подобного приведения с помощью любого из трех других операторов. С применением `static_cast` осуществляются те преобразования, которые могут быть сделаны неявно, на основе правил по умолчанию:

```
const double d = 97.0;

char ch = static_cast< char >( d );
```

Оператор `reinterpret_cast` работает с внутренними представлениями объектов (`reinterpret` – другая интерпретация того же внутреннего представления), причем правильность этой операции целиком зависит от программиста.

Оператор `dynamic_cast` применяется при идентификации типа во время выполнения (`runtime type identification`).

5.3. Стандартные математические функции

В любой программе, кроме операторов и операций, используются средства библиотек, входящих в среду программирования, которые облегчают создание программ. Часть библиотек стандартизована и поставляется с компилятором. В стандартную библиотеку входят функции, макросы, глобальные константы. Это файлы, хранящиеся в папке `include`.

Стандартные математические функции

Математические функции языка С декларированы в файлах `<cmath>` и `<stdlib.h>`.

В большинстве приведенных здесь функций аргументы `x`, `y` и результат выполнения имеют тип `double`. В табл. 5.5 приведены основные математические функции C/C++.

Таблица 5.5

Основные математические функции C/C++

Описание содержится в `math.h` | `cmath`	

`double ceil(double x);`
`float ceilf(float x);`
`long double ceill(long double x);`
	Функции округления до наименьшего целого, не меньшего, чем аргумент

`double cos(double x);`	Возвращает значение косинуса x, где x – это значение в радианах (2π радиан = 3600)
`double exp(double x);`	Возвращает значение числа e, возведенного в степень x

`double fabs(double x);`
`float fabsf(float x);`
`long double fabsl(long double x);`
	Абсолютное значение числа с плавающей точкой

`double floor(double x);`
`float floorf(float x);`
`long double loorl(long double x);`
	Наибольшее целое значение, но не большее x

`double fmod(double x, double y);`	Функция получения остатка от деления (с плавающей точкой)
`double log(double x);`	Возвращает натуральный логарифм x
`double log10(double x);`	Возвращает десятичный логарифм x
`double pow(double x, double y);`	Возвращает значение x в степени y
`int rand(void);`	Возвращает псевдослучайное число в диапазоне от нуля до `RAND_MAX`
`void srand(unsigned int seed);`	Устанавливает свой аргумент как основу (`seed`) для новой последовательности псевдослучайных целых чисел, возвращаемых функцией `rand()`
`double sin(double x);`	Возвращает значение синуса аргумента x, где x указан в радианах
`double sqrt(double x);`	Функция вычисления квадратного корня
`double tan(double x);`	Возвращает тангенс аргумента x, где x задан в радианах

Пример

```
Z = pow(x,10.0) + 3.7 * pow(x,8.0);     //Z = x^10 + 3.7·x^8
```

В прил. 4 содержится описание заголовочных файлов и стандартных функций языка C++.

5.4. Директивы препроцессора

Директивы препроцессора представляют собой инструкции, записанные в исходном тексте программы и предназначенные для выполнения

препроцессором языка. Директивы начинаются со специального знака #, помещаемого в первой позиции строки.

Директивы позволяют:

• описывать макросы, которые уменьшают трудоемкость написания программы и делают текст программы удобочитаемым и выразительным;

• включать текст из других текстовых файлов, содержащих прототипы библиотечных и разработанных пользователем функций, шаблоны структурных переменных и т.д.;

• организовывать условную компиляцию, т.е. в зависимости от заданных в командной строке или среде параметров получать различный программный код.

Директива препроцессора `#include`

Заголовочные файлы включаются в текст программы с помощью директивы препроцессора `#include`. Директива применяется для включения копии указанного в директиве файла в то место, где находится эта директива. Имя файла может быть указано двумя способами:

```
#include <file.h>
#include "file.h"
```

Различие между ними заключается в методе поиска препроцессором включаемого файла. Если имя файла заключено в угловые скобки (< >), считается, что используется стандартный заголовочный файл, и компилятор ищет этот файл в предопределенных местах. Двойные кавычки означают, что заголовочный файл – пользовательский, и его поиск начинается с того каталога, где находится исходный текст проекта.

Заголовочные файлы содержат объявления и определения (классов, структур, объединений, перечисляемых типов и прототипов функций), общие для различных программных файлов, и поэтому часто создаются и включаются в файлы программ.

Символические константы и макроподстановка

Директива препроцессора `#define` обычно используется для замены часто используемых в программе констант, ключевых слов, операторов и выражений осмысленными идентификаторами.

Идентификаторы, которые заменяют числовые или текстовые константы либо произвольную последовательность символов, называются именованными константами.

Идентификаторы, которые представляют некоторую последовательность действий, заданную операторами или выражениями языка, называются макроопределениями.

Формат директивы определяется как:

```
#define     идентификатор строка_текста
```

Директива обеспечивает замену встречающегося в тексте программы идентификатора на соответствующую строку текста, в том числе и с параметрами. Например:

```
#define   SIZE 100            //символическая константа
#define min(a,b) ((a) < (b) ? (a) : (b))  //макрос
#define PRN(number) printf(#number " = %d\n", number);
                                          //макрос
.........................
int scale = 25, param = 10;
PRN(scale);
PRN(param);
result = min(44,uplimit);
      //result = ((44) < (uplimit) ? (44) : (uplimit));
```

Как и в случае символических констант, идентификатор макроса заменяется на замещающий текст до начала компиляции программы. Макросы без параметров обрабатываются подобно символическим константам. Если макрос имеет параметры, то сначала в замещающий текст подставляются значения параметров, а затем этот расширенный макрос подставляется в текст вместо идентификатора макроса и списка его параметров.

Определения символических констант и макросов могут быть аннулированы при помощи директивы `#undef`. Она отменяет самое последнее определение поименованного макроопределения.

```
#define TRI 3
#define F 5
#undef  TRI          //TRI теперь не определен
#define F 10         //F переопределен как 10
```

В C++ отдается предпочтение использованию именованных переменных типа `const`, а не символических констант. Константные переменные являются данными определенного типа и их имена видны отладчику. Если используется символическая константа, то после того, как она была заменена на текст, только этот текст будет виден отладчику. Недостатком переменных типа `const` является то, что им требуется память в объеме, соответствующем их типу. Для символических констант дополнительной памяти не требуется.

Условные директивы

Заголовочный файл, который подключается к модулю проекта, также может содержать директивы `#include`. Поэтому некоторые заголовочные файлы могут оказаться включенными несколько раз. Избежать этого позволяют условные директивы препроцессора. Например:

```
#define FILE_H
/* содержимое файла file.h */
.............................
#ifndef FILE_H
#endif
```

Условная директива `#ifndef` проверяет, не было ли значение `FILE_H` определено ранее (`FILE_H` – это константа препроцессора; такие константы принято писать заглавными буквами). Препроцессор обрабатывает следующие строки вплоть до директивы `#endif`. В противном случае он пропускает строки от `#ifndef` до `#endif`.

Другим распространенным примером применения условных директив препроцессора является включение в текст программы отладочной информации.

Рассмотрим еще пример:

```
#ifdef O
     #include "o.h"  //выполнится, если O определен
     #define ST 10
#else
     #include "w.h"  //выполнится, если O не определен
     #define ST 20
#endif
```

Директива `#ifdef` сообщает, что если последующий идентификатор `O` определяется препроцессором, то выполняются все последующие директивы вплоть до первого появления `#else` или `#endif`. Когда в программе есть `#else`, то программа от `#else` до `#endif` будет выполняться, если идентификатор не определен.

Макрос `assert`

Макрос `assert`, определенный в заголовочном файле `<cassert>`, выполняет проверку значения выражения. Если значение выражения 0 (ложь), то макрос `assert` выводит сообщение об ошибке и вызывает функцию `abort` (из библиотеки утилит общего назначения `<cstdlib>`), которая завершает выполнение программы. Макрос удобно использовать при отладке. Например, переменная `x` в программе не должна принимать значение большее, чем 10. В этом случае макрос `assert` можно

использовать для проверки значения `x` и вывода сообщения об ошибке, если значение `x` вышло из допустимого диапазона:

```
assert ( x <= 10 ) ;
```

Если `x` будет иметь значение, большее, чем 10, то программа выдаст сообщение об ошибке, содержащее номер строки и имя файла, после чего завершит свою работу.

После того, как в тексте программы объявляется символическая константа `NDEBUG`, все последующие вызовы макроса `assert` будут игнорироваться. Таким образом, когда все они будут больше не нужны (т.е. когда отладка закончена), в начале программы достаточно добавить строку

```
#define NDEBUG;
```

вместо ручного удаления каждого макроса `assert`.

5.5. Практические задания

Для каждой из задач составить блок-схему алгоритма.

1. Дано действительное число x . Не пользуясь никакими другими арифметическими операциями, кроме умножения, сложения и вычитания, вычислить за минимальное число операций $2x^4 - 4x^3 + 2x - 1$.
2. Ввести любой символ и определить его порядковый номер, а также указать предыдущий и последующий символы.
3. Дана длина ребра куба. Найти площадь грани, площадь полной поверхности и объем этого куба.
4. Треугольник задан величинами своих углов и радиусом описанной окружности. Найти стороны треугольника.
5. Дано a. Не используя никаких функций и никаких операций, кроме умножения, получить a^8 за три операции; a^{10} и a^{16} за четыре операции.

Глава 6. ОПЕРАТОР УСЛОВИЯ И ОПЕРАТОР ВЫБОРА АЛЬТЕРНАТИВ

Все операторы управления могут быть условно разделены на следующие категории:

– условные операторы, к которым относятся оператор условия `if` и оператор выбора `switch`;

– операторы цикла (`for`, `while`, `do while`);

– операторы перехода (`break`, `continue`, `return`, `goto`).

6.1. Условный оператор `if`

Формат оператора:

```
if <выражение-условие> оператор-1; [else оператор-2;]
                                          //полная форма
if <выражение-условие> оператор-1;
                                          //сокращенная форма
```

Выполнение оператора `if` начинается с вычисления <выражения-условия>. В качестве <выражения-условия> может использоваться арифметическое выражение, отношение и логическое выражение. Далее выполнение осуществляется по следующей схеме: если выражение истинно (т.е. отлично от 0), то выполняется оператор 1, если выражение ложно (т.е. равно 0), то выполняется оператор 2, если выражение ложно и отсутствует оператор 2 (в квадратные скобки заключена необязательная конструкция), то выполняется следующий за `if` оператор. Блок-схемы операторов `if` и `if-else` приведены на рис. 6.1.

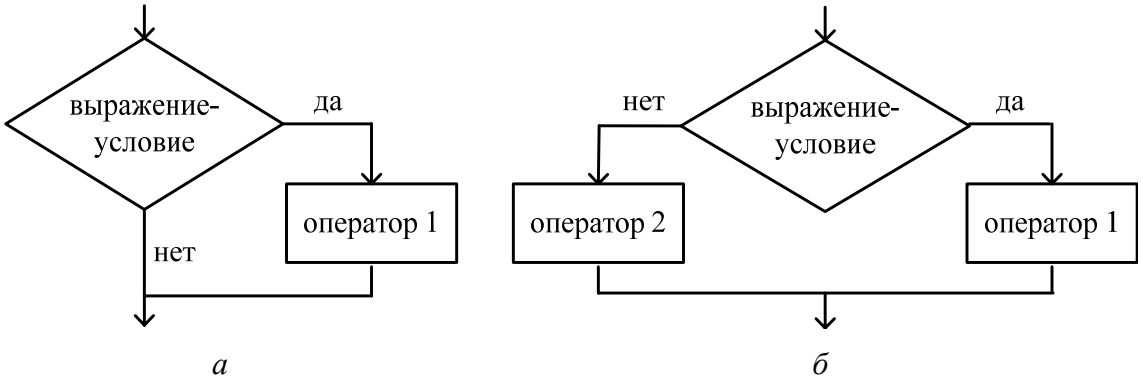


Рис. 6.1. Блок-схемы условных операторов:
а – `if`; *б* – `if-else`

Пример

```
if (i < j)          i++;
else
       {
         j =  i- 3;
         i++;
       }
```

После выполнения оператора `if` значение передается на следующий оператор программы, если последовательность выполнения операторов не будет принудительно нарушена.

Допускается использование вложенных операторов `if`. Оператор `if` может быть включен в конструкцию `if` или в конструкцию `else` другого оператора `if`. Рекомендуется группировать операторы и конструкции, используя фигурные скобки. Если же фигурные скобки опущены, то компилятор связывает каждое ключевое слово `else` с наиболее близким `if`, для которого нет `else`:

```
int t = 2, b = 7, r = 3;

   if (t > b)
      {
       if (b < r)  r = b;
      }
   else r = t;             //r станет равным 2
```

Если в программе опустить фигурные скобки, то получится:

```
int t = 2,b = 7,r = 3;

  if ( a > b )
     if ( b < c ) t = b;
     else       r = t;   //r получит значение равное 3
```

Пропуск одной или обеих фигурных скобок, ограничивающих составной оператор – типичная ошибка программирования. Для того, чтобы избежать ошибок, нужно сначала записать открывающую и закрывающую скобки составного оператора, а потом вписать требуемые операторы.

Конструкции, использующие вложенные операторы `if`, являются громоздкими и не всегда достаточно надежными.

Можно комбинировать условные выражения и логические операции. Ниже приведен пример, в котором определяется относится ли введенный символ к буквам и используется единственный оператор `if`:

```
char letter = 0;

   cout << endl
        << "Введите символ: ";
   cin >> letter;

   if(((letter >= 'A') && (letter <= 'Z')) ||
      ((letter >= 'a') && (letter <= 'z')))
      cout << endl
           << "Вы ввели букву." << endl;
   else
      cout << endl
           << "Вы ввели не букву." << endl;
```

Следующий фрагмент иллюстрирует вложенные операторы `if`:

```
char ZNAC;
int x,y,z;
 cin >> ZNAC; cin >> y; cin >> z;
     if (ZNAC == '-') x = y - z;
     else  if (ZNAC == '+') x = y + z;
           else if (ZNAC == '*') x = y * z;
                else if (ZNAC == '/') x = y / z;
                     else ...
```

Считается хорошим стилем программирования соблюдение одинакового числа пробелов в отступах структуры `if-else`.

6.2. Оператор выбора `switch`

Другим способом организации выбора из множества различных вариантов является использование специального оператора выбора `switch`. Формат оператора следующий:

```
switch (выражение)
   {[объявление]
              :
     [case  константное-выражение1]: [список-операторов1]
     [case  константное-выражение2]: [список-операторов2]
              :
              :
     [default: [список операторов]]
   }
```

Выражение, следующее за ключевым словом `switch` в круглых скобках, может быть любым выражением, допустимым в языке C/C++, значение которого должно быть целым. Значение этого выражения является ключевым для выбора из нескольких вариантов. Тело оператора `switch` состоит из нескольких операторов, помеченных ключевым словом `case` с последующим константным выражением. Обычно в качестве константного выражения используются целые или символьные константы (не может содержать переменные или вызовы функций). Все константные выражения в операторе `switch` должны быть уникальны. Кроме операторов, помеченных ключевым словом `case`, может быть, но обязательно один, фрагмент, помеченный ключевым словом `default`.

Список операторов может быть пустым либо содержать один оператор или более. В операторе `switch` не требуется заключать в фигурные скобки последовательность операторов.

Можно использовать свои локальные переменные, объявления которых находятся перед первым ключевым словом `case`, однако в объявлениях не должна использоваться инициализация.

Схема выполнения оператора `switch` следующая: вычисляется выражение в круглых скобках; вычисленные значения последовательно сравниваются с константными выражениями, следующими за ключевыми словами `case`; если одно из константных выражений совпадает со значением выражения, то управление передается на оператор, помеченный соответствующим ключевым словом `case`, если ни одно из константных выражений не равно выражению, то управление передается на оператор, помеченный ключевым словом `default`, а в случае его отсутствия управление передается на следующий после `switch` оператор. Блок-схема приведена на рис. 6.2.

Пример

```
int i=2;

switch (i)
        {
         case 1: i += 2;
         case 2: i *= 3;
         case 0: i /= 2;
         case 4: i -= 5;
         default:      ;
       }
```

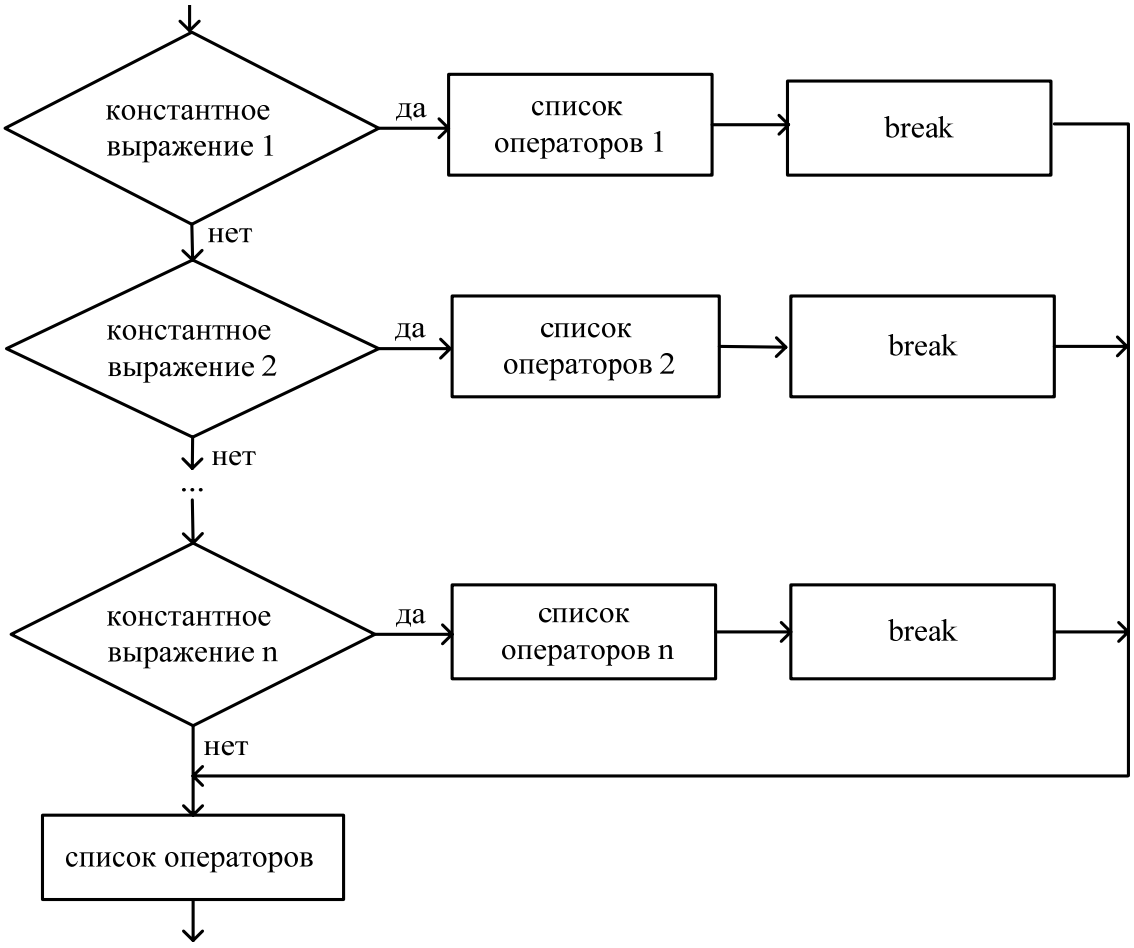


Рис. 6.2. Блок-схема оператора `switch`

Рассмотрим ранее приведенный пример, в котором иллюстрировалось использование вложенных операторов `if`, переписанных теперь с использованием оператора `switch`:

```
char ZNAC;
int x, y, z;
switch (ZNAC)
     {
          case '+':  x = y + z;     break;
          case '-':  x = y - z;     break;
          case '*':  x = y * z;     break;
          case '/':  x = u / z;     break;
          default : ;
     }
```

Использование оператора `break` позволяет в необходимый момент прервать последовательность выполняемых операторов в теле оператора `switch` путем передачи управления оператору, следующему за `switch`.

В теле оператора `switch` можно использовать вложенные операторы `switch`, при этом в ключевых словах `case` можно использовать одинаковые константные выражения:

```
switch (a)
      {
        case 1: b=c; break;
        case 2:
            switch (d)

            {   case 0:  f=s;  break;
                case 1:  f=9;  break;
                case 2:  f-=9; break;
            }

         case 3: b-=c; break;
      }
```

6.3. Практические задания

Составить блок-схему решения задачи.

1. Определить, равен ли квадрат заданного трехзначного числа кубу суммы цифр этого числа.

2. Даны целые числа *m*, *n*. Если числа не равны, то заменить каждое из них одним и тем же числом, равным большему из исходных, а если равны, то заменить числа нулями.

3. Дан круг радиуса *R*. Определить, поместится ли правильный треугольник со стороной *a* в этом круге.

4. В небоскребе *N* этажей и всего один подъезд; на каждом этаже по 3 квартиры; лифт может останавливаться только на нечетных этажах. Человек садится в лифт и набирает номер нужной ему квартиры *M*. На какой этаж должен доставить лифт пассажира?

5. Для целого числа *k* от 1 до 99 напечатать фразу «Мне *k* лет», учитывая при этом, что при некоторых значениях слово «лет» надо заменить на слово «год» или «года». Например, 11 лет, 22 года, 51 год.

Глава 7. ЦИКЛИЧЕСКИЕ КОНСТРУКЦИИ И ОПЕРАТОРЫ ПЕРЕХОДОВ

При выполнении программы нередко возникает необходимость неоднократного повторения однотипных вычислений над различными данными. Для этих целей используются циклы. *Цикл* – участок программы, в котором одни и те же вычисления реализуются неоднократно над различными значениями одних и тех же переменных (объектов).

Для организации циклов в C++ используются следующие операторы: `for`, `while`, `do while`.

7.1. Оператор цикла `for`

Цикл `for` является циклом с параметрами и обычно используется в случае, когда известно точное количество повторов вычислений. Оператор `for` – это наиболее общий способ организации цикла. Он имеет следующий формат:

```
for ( выражение 1 ; выражение 2 ; выражение 3 ) тело
```

Выражение 1 обычно используется для установления начального значения переменных, управляющих циклом. Выражение 2 – это выражение, определяющее условие, при котором тело цикла будет выполняться. Выражение 3 определяет изменение переменных, управляющих циклом после каждого выполнения тела цикла.

Схема выполнения оператора `for`: вычисляется выражение 1; вычисляется выражение 2; если значение выражения 2 отлично от нуля (истина), выполняется тело цикла, вычисляется выражение 3 и осуществляется переход к пункту 2; если выражение 2 равно нулю (ложь), то управление передается на оператор, следующий за оператором `for`. Блок-схема выполнения цикла приведена на рис. 7.1.

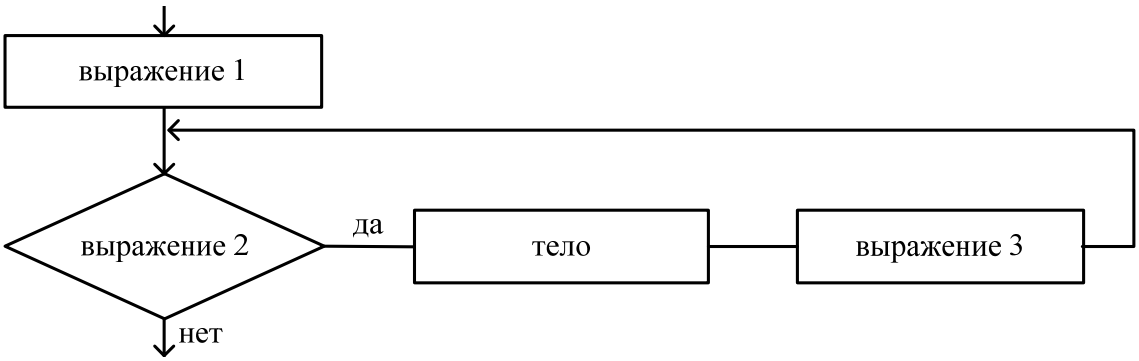


Рис. 7.1. Блок-схема цикла `for`

Проверка условия всегда выполняется в начале цикла. Это значит, что тело цикла может ни разу не выполниться, если условие выполнения сразу будет ложным. Например:

```
int i, b;
 for (i = 1; i < 10; i++) b = i * i;
             //вычисление квадратов чисел от 1 до 9
 for (i = 1; i > 10; i++) b = i * i;
             //тело цикла не выполнится
```

Желательно в разделе задания начальных значений и изменения переменных структуры `for` задавать только выражения, относящиеся к управляющей переменной. Манипуляции с другими переменными должны размещаться или до цикла (если они выполняются только один раз подобно операторам задания начальных значений), или внутри тела цикла (если они должны выполняться в каждом цикле, как, например, операторы инкремента или декремента).

Некоторые варианты использования оператора `for` повышают его гибкость за счет возможности использования нескольких переменных, управляющих циклом:

```
int top,  bot;
char string[100],  temp;
                   //для управления циклом используются
                   //две переменные top и bot

   for (top = 0, bot = 100; top < bot; top++, bot--)
      {
         temp = string[top];
          string[bot] = temp;
       }
```

Важно всегда задавать начальные значения всем счетчикам и переменным сумм цикла. Управлять количеством повторений цикла нужно с помощью целой переменной. Хорошим стилем программирования является размещение пустой строки до и после каждой управляющей структуры, чтобы она выделялась в программе.

Другим вариантом использования оператора `for` является бесконечный цикл. Для организации такого цикла можно использовать пустое условное выражение, а для выхода из цикла обычно используют дополнительное условие и оператор `break`:

```
for (;;)
 { ...
   ...  break;
   ...
 }
```

Так как согласно синтаксису языка оператор может быть пустым, тело оператора `for` также может быть пустым. Такая форма оператора может быть использована для организации поиска:

```
for (int i = 0; t[i] < 10; i++) ;
             //i – номер первого элемента массива t,
             //значение которого больше 10
```

В данном примере в теле цикла происходит объявление переменной `i`. Цикл имеет область видимости, распространяющуюся от управляющих выражений `for` до конца его тела. Так как счетчик `i` объявлен внутри области видимости цикла, к нему нельзя обращаться за пределами этой области (он исчезает).

Можно вообще исключить установку начальных значений из цикла. Если инициализировать `i` в отдельном операторе объявления, то цикл можно записать так:

```
int i=0;
for (; t[i] < 10 ; i++) ;
```

Фактически обе точки с запятой всегда должны присутствовать независимо от того, пропущено ли какое-то одно из управляющих выражений или даже все сразу.

7.2. Оператор цикла `while`

Оператор цикла `while` называется циклом с предусловием и имеет следующий формат:

```
while (выражение) тело ;
```

В качестве выражения допускается использовать любое выражение языка C/C++, в качестве тела – любой оператор, в том числе пустой или составной.

Схема выполнения оператора `while` следующая: вычисляется выражение; если выражение ложно, то выполнение оператора `while` заканчивается и выполняется следующий по порядку оператор; если выражение истинно, то выполняется тело оператора `while`; процесс повторяется сначала. Блок-схема приведена на рис. 7.2.

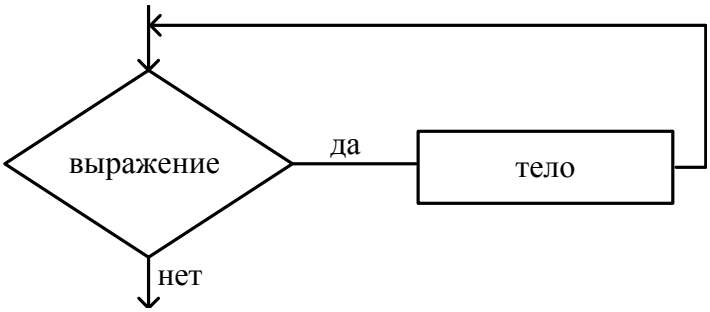


Рис. 7.2. Блок-схема выполнения оператора `while`

Рассмотрим пример:

```
double value = 0.0;              //для ввода значений
   double sum = 0.0;             //сумма значений
   int i = 0;                    //счетчик значений
   char indicator = 'y';         //индикатор продолжения

  while(indicator == 'y')        //повторять цикл пока 'y'
   {
      cout << endl
           << "Введите значение: ";
      cin >> value;              //ввести значение
      ++i;                       //увеличить счетчик
      sum += value;              //добавить значение к сумме

      cout << endl
           << " Хотите ввести еще значение ? ";
      cin >> indicator;          //ввести индикатор
   }
```

Внутри операторов `for` и `while` можно использовать локальные переменные, которые должны быть объявлены с определением соответствующих типов.

Типичной ошибкой программирования является отсутствие в теле структуры `while` действия, которое приведет со временем к ложному условию `while`. Оно называется зацикливание.

7.3. Оператор цикла `do while`

Оператор цикла `do while` называется оператором цикла с постусловием и используется в тех случаях, когда необходимо выполнить тело цикла хотя бы один раз. Формат оператора имеет следующий вид:

```
do тело while (выражение);
```

Схема выполнения оператора `do while`: выполняется тело цикла (которое может быть составным оператором); вычисляется выражение; если выражение ложно, то выполнение оператора `do while` заканчивается и выполняется следующий по порядку оператор; если выражение истинно, то выполнение оператора продолжается сначала. Блок-схема приведена на рис. 7.3.

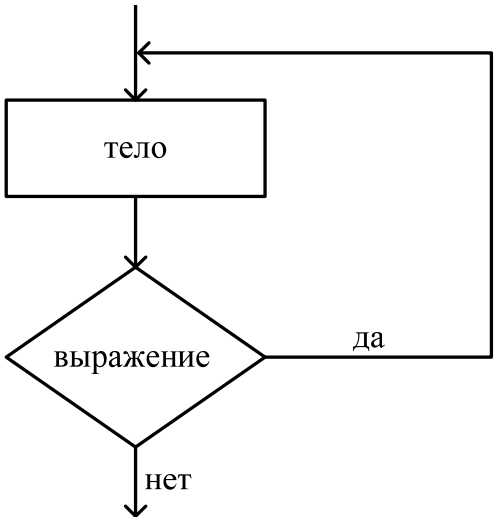


Рис. 7.3. Блок-схема выполнения оператора `do while`

Чтобы прервать выполнение цикла до того, как условие станет ложным, можно использовать оператор `break`.

Операторы `while`, `for` и `do while` могут быть вложенными:

```
do
   {
      cout << endl
           << "Введите целое число: ";
      cin >> value;

      factorial = 1;
      for(int i = 2; i <= value; i++)
         factorial *= i;

cout << "Факторила " << value << " равен " << factorial;
      cout << endl
           << "Хотите ввести еще значение (y или n)? ";
      cin >> indicator;
   } while((indicator == 'y') || (indicator == 'Y'));
```

При выборе варианта циклической конструкции лучше использовать цикл `for`. Это всегда можно сделать, если заранее известно число повторений цикла. В остальных случаях используются циклы `while` и `do while`, причем цикл `do while` следует применять, если требуется тело цикла выполнить не менее одного раза.

7.4. Оператор перехода `break`

Оператор `break` изменяет поток управления, он прерывает цикл. Его целесообразно использовать, когда условие продолжения итераций надо проверять в середине цикла:

```
//ищет сумму чисел до тех пор,
//пока не будет введено 100 чисел или 0
 for(s = 0, i = 1; i < 100; i++)

    {
       cin >> x;
       if( x == 0) break;
      //если ввели 0, то суммирование заканчивается
       s += x;
    }
```

7.5. Оператор перехода `continue`

Оператор `continue`, как и оператор `break`, используется внутри операторов цикла, но в отличие от него выполнение программы продолжается со следующей итерации, оставшаяся часть тела структуры пропускается.

Формат оператора следующий:

```
continue;
```

Пример

```
int a,b;

   for (a = 1,b = 0; a < 100; b += a,a++)
   {
   if (b % 2) continue;
//передача управления на очередную итерацию цикла for
//без выполнения операторов обработки четных сумм
   ... //обработка четных сумм
   }
```

7.6. Оператор безусловного перехода goto

Использование оператора безусловного перехода goto в практике программирования C/C++ настоятельно не рекомендуется, так как он затрудняет понимание программы и возможность ее модификации. Формат оператора следующий:

```
goto имя-метки;
     ...
имя-метки: оператор;
```

Оператор goto передает управление на оператор, помеченный меткой имя-метки. Помеченный оператор должен находиться в той же функции, что и оператор goto, а используемая метка должна быть уникальной, т.е. одно имя-метки не может быть использовано для разных операторов программы. Приведем пример организации цикла, используя только goto и if:

```
int i = 0, sum = 0;
const int max = 10;
i=1;

loop:
   sum += i; //накопление суммы целых чисел  от 1 до 10
   if (++i <= max)
      goto loop;
```

Любой оператор в составном операторе может иметь свою метку. Используя оператор goto, можно передавать управление внутрь составного оператора.

Любая программа может быть написана без goto. Однако существуют ситуации (их немного), когда goto рекомендуется использовать, например, выход из многократно вложенных циклов (поскольку оператор break осуществляет выход только из того цикла, где он использован):

```
for (...)
   for (...)

      {...
      if ( ошибка )  goto Error;
        }
                  ...
        Error :
```

В остальных случаях использовать goto не следует.

7.7. Оператор `return`

Оператор `return` завершает выполнение функции, в которой он задан, и возвращает управление в вызывающую функцию, в точку, непосредственно следующую за вызовом. Функция `main` передает управление операционной системе. Формат оператора:

```
return [выражение];
```

Значение выражения, если оно задано, возвращается в вызывающую функцию в качестве значения вызываемой функции. Если выражение опущено, то возвращаемое значение не определено. Если функция не должна иметь возвращаемого значения, то ее нужно объявлять с типом `void`.

Пример

```
int sum (int a, int b)

  {
     renurn (a+b);
   }
```

7.8. Практические задания

Для каждой из задач составить блок-схему алгоритма.

1. Дано натуральное число n. Найти сумму первой и последней цифры этого числа.
2. Составить программу, которая печатает таблицу умножения.
3. Найти все двузначные числа, сумма квадратов цифр которых кратна M.
4. Найти сумму всех n-значных чисел при $1 \le n \le 4$.
5. Дано натуральное число n. Переставить его цифры так, чтобы образовалось максимальное число, записанное теми же цифрами.

Глава 8. ОДНОМЕРНЫЕ МАССИВЫ И УКАЗАТЕЛИ

8.1. Определение массива

Массив (или *вектор*) – это группа связанных друг с другом элементов одного типа (`double`, `float`, `int` и т.п.), последовательно расположенных в памяти. Из объявления массива компилятор должен получить информацию о типе элементов массива и их количестве. Объявление массива имеет два формата:

```
спецификатор-типа описатель [константное – выражение];
спецификатор-типа описатель [ ];
```

Описатель – это идентификатор массива. Спецификатор-типа задает тип элементов объявляемого массива. Элементами массива не могут быть функции и элементы типа `void`.

Константное выражение в квадратных скобках задает количество элементов массива. Константное выражение при объявлении массива может быть опущено в следующих случаях:

- при объявлении массив инициализируется;
- массив объявлен как формальный параметр функции;
- массив объявлен как ссылка на массив, явно определенный в другом файле.

Пример определения массива:

```
int arr[5];
```

Элементы массива занимают один непрерывный участок памяти компьютера и располагаются последовательно друг за другом (рис. 8.1).

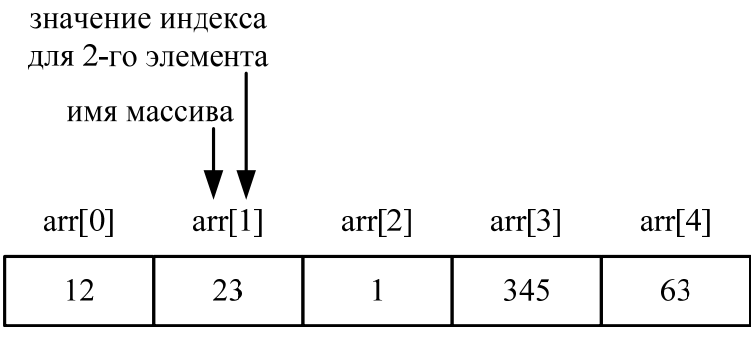


Рис. 8.1. Расположение элементов массива

Доступ к отдельным элементам массива организуется с использованием номера этого элемента или индекса (`[]`). Нумерация элементов

массива начинается с нуля и заканчивается $n - 1$, где n – число элементов массива.

Важно различать, например, седьмой элемент массива и элемент массива семь. Поскольку индексы начинаются с 0, седьмой элемент массива имеет индекс шесть, тогда как элемент массива семь имеет индекс 7 и на самом деле является восьмым элементом массива. Это источник ошибок типа завышения (или занижения) на единицу. Дело в том, что C++ не проверяет границ массивов и не предупреждает о ссылках на несуществующие элементы.

При объявлении автоматических и статических массивов для задания их размеров лучше использовать именованные константы:

```
const int array_size = 10;
int arr[array_size];
```

Это делает программу более масштабируемой, при изменении размера массива надо будет поменять число только в одном месте.

8.2. Инициализация и работа с массивами

Инициализация массива означает присвоение начальных значений его элементам при объявлении. Массивы можно инициализировать списком значений или выражений, отделенных запятой, заключенных в фигурные скобки:

```
double d[] = {1, 2, 3, 4, 5};
int n[5] = {2};
```

Длина массива вычисляется компилятором по количеству значений, перечисленных в фигурных скобках. Если размер массива задан, нельзя специфицировать значений больше, чем объявлено в массиве (синтаксическая ошибка), но меньше указать можно (минимально должен быть определен один элемент). Если массив явно не проинициализирован, то внешние и статические массивы инициализируются нулями. Автоматические массивы после объявления ничем не инициализируются и содержат неизвестную информацию.

В заключение рассмотрим задачу: сформировать одномерный статический массив целых чисел, используя датчик случайных чисел (диапазон значений от 0 до 99); подсчитать среднее арифметическое элементов в массиве; удалить элемент с заданным номером. Ниже приведен пример фрагмента листинга программы:

```
const int N = 1000;
  //N - максимальный размер статического массива

.........................................
int i;          //индекс массива
int array_size; //переменная для хранения размера массива
int avg;        //переменная для хранения
                //среднего арифметического
int arr[N];     //целочисленный статический массив длины N
int k;          //индекс удаляемого элемента
int rmin = 0, rmax = 99;
                //диапазон значений элементов массива

cout<<"Введите размер массива"<<endl;
     cin>> array_size<<endl;

//Сгенерировать массив
srand((unsigned)time(NULL));

     for(i = 0 ; i < array_size ; i++)
     {
       arr[i] = (int)(((double)rand()/(double)RAND_MAX)*
                                    (rmax-rmin)+rmin);
        //или arr[i] = rand()%99;
        cout<<arr[i]<<endl;
       }

//Подсчитать среднее арифметическое суммы элементов:
     avg = 0;
     for(i = 0; i < array_size; i++)
          {
//т.к. в теле цикла один оператор,
//фигурная скобка не обязательна
          avg += arr[i];
          }

     avg /= array_size;

cout<<"Средний элемент массива"<<avg<<endl;

cout<<"Введите номер удаляемого элемента массива"<<endl;
       cin >>k;

//удалить элемент с номером k, сдвиг элементов массива
          for(i = k; i < array_size - 1; i++)
          {
               arr[i] = arr[i+1];
          }

          array_size--; //уменьшить размер массива

     for(i=0;i< array_size; i++)  //вывод массива
           cout<<arr[i]<<endl;
.........................................
```

Рассмотрим еще один пример: пусть необходимо ввести целочисленный массив размера n. Затем удалить из массива все элементы, встречающиеся два и более раз:

```
//объявление переменных
int a[20];
int n;
//инициализация переменных
     cout<<"Введите размер массива "<<endl;
     cin>>n;
     for(int i = 0; i < n; i++)
          scanf( "%d",&a[i]); //ввести массив
                              //обработка массива
          for(int k = 0; k < n; k++)
          {
              for(int i = k + 1; i < n; i++)
              {
                  if(a[k] == a[i])
                  {
                     for(int l = i; l <n; l++)
                     a[l]=a[l+1];
                          //удалить повторяющиеся элементы
                     n--;
                          //скорректировать размер массива
                  }
              }
          }
                          //выводим результат
     for(i = 0; i < n; i++) printf( "%d ", a[i]);
     printf("\n");        //вывести массив
```

8.3. Указатели

Указатель – это символическое представление адреса. Он используется для косвенной адресации переменных и объектов. Указатели тесным образом связаны с обработкой строки и массива.

В языке C++ имеется операция определения (взятия) адреса – &, с помощью которой определяется адрес ячейки памяти, содержащей заданную переменную. Например, если a – имя переменной, то &a – адрес этой переменной (рис. 8.2).

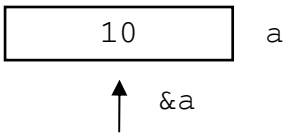


Рис. 8.2. Адрес и имя переменной a

Запись `&a` означает: «указатель на переменную `a`». Указатель на переменную – это число, которое представляет собой содержимое регистров CS:IP процессора. Операция `&` применима только к объектам, имеющим имя и размещенным в памяти (переменным, массивам, структурам, строковым литералам и т.п.). Ее нельзя применить к переменным с классом хранения `register`, а также к полям бит. Символическое представление адреса `&a` является константой типа указатель, ни одна операция на нее не действует:

```
int i = 0;
int& a = i;
    a++;
    //не увеличивает ссылку, а i увеличивается на 1
```

В C++ также существуют и переменные типа указатель. При работе с указателями, важное значение имеет операция *косвенной адресации* – `*`. Операция `*` позволяет обратиться к переменной не напрямую, а через указатель, содержащий адрес этой переменной. Данная операция является одноместной и имеет ассоциативность слева направо. Пусть `pa` – указатель, тогда `*pa` – это значение переменной, на которую указывает `pa`.

В C++ существует общепринятое соглашение – называть переменные-указатели именами, начинающимися с `p` (pointer), чтобы было ясно, что это указатель и требуется соответствующая обработка.

Формат объявления указателя:

```
спецификатор-типа *описатель;
```

Спецификатор-типа задает тип объекта и может быть любого основного типа, типа структуры, смеси (об этом будет сказано ниже) или `void`. Чтобы выполнить арифметические и логические операции над указателями или над объектами, на которые они указывают, необходимо при выполнении каждой операции явно определить тип объектов. Такие определения типов могут быть выполнены с помощью операции приведения типов. Например:

```
char *pz;
int *pk,*pi;
float *pf;
```

Здесь `*` – это операция разыменования, то есть ссылка на объект, на который указывает указатель. Операндом этой операции всегда является указатель. Результат операции – это тот объект, который адресует указатель-операнд (рис. 8.3).

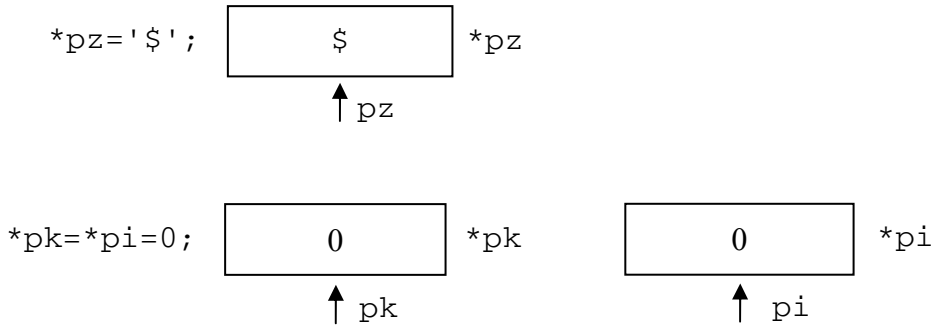


Рис. 8.3. Операции разыменования

```
int e, c, b, *pm;
..................
pm = &e ;            //в pm хранится адрес e;
*pm = c + b ;
```

Графическая интерпретация фрагмента программы приведена на рис. 8.4. Операция * в некотором смысле является обратной операции &.

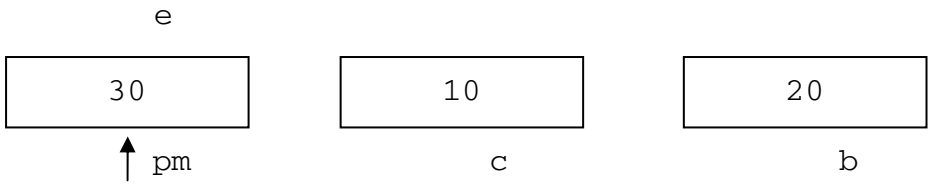


Рис. 8.4. Графическиая интерпретация примера

Использование неинициализированных указателей опасно (можно перезаписать произвольную память). Объявляя указатель, необходимо инициализировать его адресом определенной переменной или нулем. Для этого в C++ предусмотрена символическая константа NULL, которая уже определена (в iostream) как 0:

```
int *pm = NULL; //указатель, не указывающий ни на что
..................
if (pm == NULL)
  cout << endl << "нулевой указатель";
```

Это гарантирует, что указатель не содержит адреса, который воспринимается как корректный.

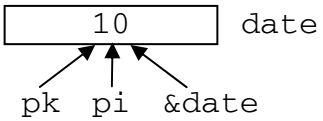
Без указателей невозможна обработка объектов. Например, передать объект некоторой функции можно только через указатель. Указатели используются для работы с динамической памятью.

8.4. Операции над указателями

Над указателями возможны операции: присваивание (=) указателей одного типа; получение значения объекта, на который ссылается указатель (*); получение адреса самого указателя (&).

Пример

```
int date = 10;
int *pi, *pk;
pi = &date;
pk = pi;
```



Допустимы также операции сравнения: ==, !=, <, <=, >, >=, адресной арифметики и все производные от нее: +, –, ++, --. С помощью унарных операций ++ и -- числовые значения переменных типа указатель меняются по-разному, в зависимости от типа данных, с которым связаны эти переменные:

```
char *pz;
int *pk,*pi;
float *pf;
pz++; //значение указателя изменяется на 1
pi++; //значение указателя изменяется на 4
pf++; //значение указателя изменяется на 4
```

При изменении указателя на 1, указатель переходит к началу следующего (предыдущего) поля той длины, которая определяется типом объекта, адресуемого указателем.

Снабжение описания указателя префиксом const делает объект, но не сам указатель, константой:

```
const char* pc = "asdf";    //указатель на константу
pc[3] = 'a';                //ошибка
pc = "abc";                 //ok
```

Чтобы описать сам указатель, а не указываемый объект, как константный, используется операция const*:

```
char *const cp = "asdf";    //константный указатель
cp[3] = 'a';                //ошибки нет
cp = "abc";                 //ошибка
```

Чтобы сделать константами оба объекта, их оба нужно описать const:

```
const char *const cpc = "asdf"//const указатель на const
cpc[3] = 'a';                    //ошибка
cpc = "abc";                     //ошибка
```

Указателю на константу можно присваивать адрес переменной. Однако нельзя присвоить адрес константы указателю, на который не было наложено ограничение, поскольку это позволило бы изменить значение объекта.

8.5. Ссылочный тип

Ссылочный тип, иногда называемый *псевдонимом*, служит для задания объекту дополнительного имени. Ссылка позволяет косвенно манипулировать объектом, точно так же, как это делается с помощью указателя. Однако эта косвенная манипуляция не требует специального синтаксиса, необходимого для указателей. Обычно ссылки употребляются как формальные параметры функций. Ссылочный тип обозначается указанием оператора взятия адреса (&) перед именем переменной.

```
int ival = 1024;
int &refVal = ival; //refVal – ссылка на ival
int &refVal2;       //ошибка: ссылка должна быть
                    //инициализирована
```

Ссылка должна быть инициализирована. Хотя ссылка очень похожа на указатель, она должна быть инициализирована не адресом объекта, а его значением.

```
int ival = 1024;
int &refVal = &ival;     //ошибка: refVal имеет тип int,
                         //а не int*
int *pi = &ival;         //ptrVal – ссылка на указатель
```

Определив ссылку, нельзя изменять ее так, чтобы работать с другим объектом. Все операции со ссылками воздействуют на адресуемые ими объекты, в том числе и операция взятия адреса. Если ссылки определяются через запятую, перед каждым объектом типа ссылки должен стоять амперсанд (&) (точно так же, как и для указателей). Например:

```
int ival = 1024, ival2 = 2048;
//определены переменные типа int

int &rval = ival, rval2 = ival2;
//определена одна ссылка и одна переменная

int ival3 = 1024, *pi = ival3, &ri = ival3;
//определена переменная, указатель и  ссылка

int &rval3 = ival3, &rval4 = ival2;
//определены две ссылки
```

Между ссылкой и указателем существуют два основных отличия. Во-первых, ссылка обязательно должна быть инициализирована в месте своего определения. Во-вторых, всякое изменение ссылки преобразует не ее, а тот объект, на который она ссылается. Например:

```
int *pi = 0;
const int &ri = 0;

int ival = 1024, ival2 = 2048;
int *pi = &ival, *pi2 = &ival2;

pi = pi2; //pi и pi2 указывают на переменную ival2

int &ri = ival, &ri2 = ival2;

ri = ri2;
//ival меняется,ссылка ri по-прежнему адресует ival
```

В реальных C++ программах ссылки редко используются как самостоятельные объекты, обычно они употребляются в качестве формальных параметров функций.

8.6. Указатели и массивы

Поскольку указатели представляют собой символические адреса объектов, то это дает возможность применять адреса и повышать эффективность программ.

Имя массива `arr` без индекса является указателем-константой (не изменяются на протяжении всей работы программы), т.е. адресом первого элемента массива (`&arr[0]`) (рис. 8.5).

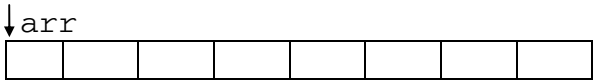


Рис. 8.5. Имя массива как указатель

```
int arr[8],*parr;
*arr = = arr[0] ;         //эквивалентно
*(arr + 1) = = arr[1];    //эквивалентно
*(arr + i) = = arr[i];    //эквивалентно
parr = arr + 5;
          //указатель на 5-й элемент массива
parr = &arr[2];
          //указатель на 2-й элемент массива
arr[3] = *(parr+1);
          //3-му элементу массива присвоить знач. 1-го
```

Необходимо иметь в виду, что указатель является переменной, так что операции `parr = arr` и `parr++` имеют смысл, но *имя массива является константой*, а не переменной, поэтому следующие конструкции будут не допустимы:

```
arr = parr;          //ошибка
arr++;               //ошибка
parr = &arr;         //ошибка
```

Листинг фрагмента программы из 8.2 с использованием указателей для обращения к элементам массива будет выглядеть следующим образом:

```
int arr[100];
int rmax = 10;
int rmin = 0;
.........................................
cout<<"Введите размер массива"<<endl;
     cin>> array_size;
//Сгенерировать массив
srand((unsigned)time(NULL));
   for(i=0;i< array_size; i++)
    {
   *(arr + i) = (int)(((double)rand()/(double)RAND_MAX)*
                                    (rmax-rmin)+rmin);
               //или *(arr + i) = rand()%99;
          cout<<*(arr + i)<<endl;
     }
//Подсчитать среднее арифметическое суммы элементов:
     avg = 0;
     for(i = 0; i < array_size; i++)
          {
               //т.к. в теле цикла один оператор,
               //фигурная скобка не обязательна
          avg += *(arr + i);
          }
     avg /= array_size;
cout<<"Средний элемент массива"<<avg<<endl;
cout<<"Введите номер удаляемого элемента массива"<<endl;
     cin >> k;
//удалить элемент с номером k, сдвиг элементов массива
for(i = k; i < array_size-1; i++)
          {
               *(arr + i) = *(arr + i + 1);
          }
               array_size--; //уменьшить размер массива
     for(i = 0; i < array_size; i++)
          cout<<*(arr + i)<<endl;
.........................................
```

8.7. Генерация случайных чисел

В приведенном примере массив заполняется случайными числами. Элемент случайности может быть введен в приложения с помощью функции rand из стандартной библиотеки:

```
i = rand () ;
```

Функция rand генерирует целое число в диапазоне от 0 до RAND_MAX (символическая константа, определенная в заголовочном файле <stdlib>). Значение RAND_MAX должно быть, по меньшей мере, равно 32767 – максимальное положительное значение двухбайтового целого числа. Для того, чтобы выработать целые числа в заданном диапазоне, используется операция вычисления остатка (%) в сочетании с rand:

```
i = rand () % 6 ;   //генерирует случайное число от 0 до 5
```

Это называется масштабированием, а число 6 называется масштабирующим коэффициентом.

Функция rand, на самом деле, генерирует псевдослучайные числа. Повторный вызов rand производит число, которое кажется случайным. Но то же самое число повторяется при каждом повторении программы. Для рандомизации необходимо использование стандартной библиотечной функции srand. Функция srand получает целый аргумент unsigned и при каждом выполнении программы задает начальное число, которое функция rand использует для генерации последовательности квазислучайных чисел.

Для того, чтобы не вводить каждый раз начальное число, можно использовать оператор:

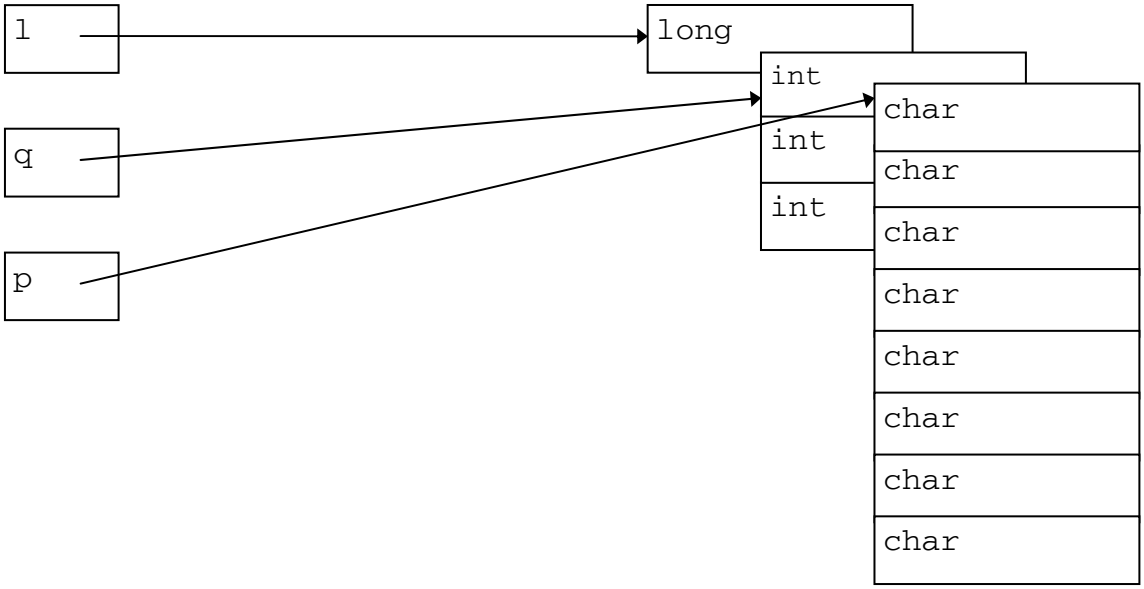
```
srand((unsigned)time(NULL));
```

Для автоматического получения начального числа считываются показания часов. Функция time (с аргументом NULL, как записано в указанном выше операторе) возвращает текущее «календарное время» в секундах. Это значение преобразуется в беззнаковое целое число и используется как начальное значение в генераторе случайных чисел. Прототип функции для time находится в <ctime>.

Функция srand должна вызываться только один раз в программе для достижения желаемого результата рандомизации. Вызов ее более одного раза является избыточным и снижает эффективность программы.

8.8. Присваивание указателей различного типа

Операцию присваивания указателей различных типов следует понимать как назначение указателя в левой части на ту же самую область памяти, на которую назначен указатель в правой. Оба указателя после присваивания содержат один и тот же адрес. Но поскольку тип указуемых переменных у них разный, то эта область памяти по правилам интерпретации указателя будет рассматриваться как заполненная переменными либо одного, либо другого типа:




```
char *p, A[20];
int  *q;
long *l;
p = A;
q = (int*)p;
l = (long*)p;
```

8.9. Практические задания

Для каждой задачи составить блок-схему алгоритма. При решении задачи использовать нотацию индексов и указателей для доступа к элементам массива.

1. Составить программу, которая заполняет квадратную матрицу порядка n натуральными числами 1, 2, 3, …, 2^n, записывая их «по спирали».

2. В целочисленной последовательности есть нулевые элементы. Создать массив из номеров этих элементов.

3. В одномерном массиве все отрицательные элементы переместить в начало массива, а остальные – в конец с сохранением порядка следования. Дополнительный массив заводить не разрешается.

4. Дан массив n действительных чисел. Требуется упорядочить его по возрастанию. Делается это следующим образом: сравниваются два соседних элемента a_i и a_{i+1}. Если $a_i \le a_{i+1}$, то продвигаются на один элемент вперед. Если $a_i > a_{i+1}$, то производится перестановка и сдвигаются на один элемент назад.

5. Дан целочисленный массив $A[n]$, среди элементов есть одинаковые. Создать массив из различных элементов $A[n]$.

Глава 9. СТРОКИ

Для представления символьной информации можно использовать символы, символьные переменные и символьные константы.

В C++ поддерживаются два типа строк – встроенный тип, доставшийся от C, и класс `string` из стандартной библиотеки C++. Класс `string` предоставляет гораздо больше возможностей и поэтому удобнее в применении.

9.1. Встроенный строковый тип

Встроенный строковый тип перешел в C++ от C. Строка символов хранится в памяти как массив, и доступ к ней осуществляется при помощи указателя типа `char*`. Количество элементов в таком массиве на один элемент больше, чем изображение строки, т.к. в конец строки добавлен `'\0'` (нулевой байт или нуль-терминатор):

символ (1 байт) строка (2 байта)

Объявление и инициализация строк

Поместить строку в массив можно либо при вводе, либо с помощью инициализаций:

```
char str1[] = "STRING";
char str2[20] = { 'S','t','r','i','n','g','\0' };
const char *str3 = "STRING\n";
char  str4[20];

     cin >> str4;
     //строка не должна превышать 19 симв. + 0 символ
```

Символьные строки хранятся в виде массивов, поэтому их нельзя приравнивать и сравнивать с помощью операций = и ==.

Типичной ошибкой является невыделение достаточного места в массиве символов для хранения нулевого символа, завершающего строку.

Строку можно присвоить символьному массиву, используя операцию `cin`. В некоторых случая желательно вводить в массив полную строку текста. С этой целью можно использовать функцию C++ `cin.getline`. Она требует три аргумента – массив символов, в котором должна храниться строка текста, длина и символ-ограничитель:

```
char str1[80];
cin.getline(str1, 80, '\n');
```

Функция прекращает считывать символы, если встречается символ-ограничитель `'\n'` и если количество считанных символов оказывается на один меньше, чем указано во втором аргументе. Третий аргумент имеет `'\n'` в качестве значения по умолчанию, так что в вызове функции его можно опустить.

Согласно общепринятому стандарту ASCII установлено соответствие между символами и кодами. Клавиатура (совместно с драйвером) кодирует нажатие любой клавиши с учетом регистровых и управляющих клавиш в соответствующий ей код (прил. 1 содержит список символьных кодов ASCII). Например:

```
' '    – 0x20,        'B'  – 0x42,
'*'    – 0x2A,        'Y'  – 0x59,
```

Манипуляции со строками и символами, на самом деле, подразумевают манипуляции с соответствующими численными кодами, а не с самими символами. Это объясняет взаимозаменяемость символов и малых целых в C++. Так как имеет смысл говорить, что один численный код больше, меньше или равен другому численному коду, можно сопоставлять различные строки и символы друг с другом.

Некоторые программы и стандартные функции обработки символов и строк используют тот факт, что цифры, прописные и строчные (маленькие и большие) латинские буквы имеют упорядоченные по возрастанию значения кодов:

```
'0' – '9'   0x30 – 0x39
'A' – 'Z'   0x41 – 0x5A
'a' – 'z'   0x61 – 0x7A
```

Тогда, для получения символа десятичной цифры из значения целой переменной, лежащей в диапазоне 0…9, а также значения целой переменной из символа десятичной цифры можно:

```
int   n = 5;
char  c;

   c = n + '0';

if (c >= '0' && c <= '9')
     n = c – '0';
```

Для преобразования строчной латинской буквы в прописную необходимо:

```
char  c;

if (c >='a' && c <='z')
      c = c – 'a' + 'A';
```

Типичной ошибкой является обработка одного символа как строки.

Работа со строкой

Обычно для перебора символов строки применяется адресная арифметика. Поскольку строка всегда заканчивается нулевым символом, можно увеличивать указатель на 1, пока очередным символом не станет нуль. Например:

```
while (*str1++ ) { ... }
```

`str1` разыменовывается, и получившееся значение проверяется на истинность. Любое отличное от нуля значение считается истинным, и, следовательно, цикл заканчивается, когда будет достигнут символ с кодом 0. Операция инкремента `++` прибавляет 1 к указателю `str1` и таким образом сдвигает его к следующему символу.

Подсчет длины строки может выглядеть следующим образом:

```
char st[] = "STRING";
int cnt = 0;
   if ( st )
        while ( *st++ )
                ++cnt;
```

Поскольку указатель может содержать нулевое значение (ни на что не указывать), перед операцией разыменования его следует проверять.

Строка встроенного типа может считаться пустой в двух случаях: если указатель на строку имеет нулевое значение (строки нет) или указывает на массив, состоящий из одного нулевого символа (строка не содержит ни одного значимого символа).

```
//pstr1 не адресует массив символов
char *pstr1 = 0;

//pstr2 адресует нулевой символ
const char *pstr2 = "";
```

При работе со строкой можно, также как и в массивах, использовать нотацию индексов:

```
for (int i = 0; (s1[i] = s2[i])!= '\0'; i++);
//копирование строки
```

а также нотацию указателей:

```
for (; (*s1 = *s2) != '\0'; s1++, s2++);
//копирование строки
```

Использование строк встроенного типа чревато ошибками из-за слишком низкого уровня реализации и невозможности обойтись без адресной арифметики. Рассмотрим типичные ошибки. Например:

```
const char *str = "STRING\n";
    int len = 0;
while ( str++ ) ++len;
//ошибка, str не разыменовывается и не изменяется
```

Указатель `str` не разыменовывается, следовательно, на равенство нулю проверяется не символ, а сам указатель. Поскольку изначально этот указатель имел ненулевое значение (адрес строки), то он никогда не станет равным нулю, и цикл будет выполняться бесконечно.

Поскольку строка представляет собой последовательность символов, большинство программ, обрабатывающих строки, используют последовательный или посимвольный просмотр строки. Если же в процессе обработки строки предполагается изменение ее содержимого, то проще всего (но не всегда эффективно) организовать его в виде посимвольного переписывания входной строки в выходную. При этом нужно помнить, что каждой строке требуется отдельный индекс, если для входной строки он может изменяться в заголовке цикла посимвольного просмотра, то для выходной строки он меняется только в моменты добавления очередного символа. Кроме того, не нужно забывать «закрывать» выходную строку символом конца строки. В качестве примера рассмотрим пример удаления лишних пробелов из строки:

```
char str1[] = "STR ING";
char str2[10];

int i,j;
for (j = 0, i = 0; str1[i] != 0; i++)
//Посимвольный просмотр строки
     {
     if (str1[i] != ' ')    //Текущий символ – не пробел
          {
          if (i!=0 && str1[i-1] == ' ') //Первый в слове -
          str2[j++] = ' ';              //добавить пробел
          str2[j++] = str1[i];    //Перенести символ слова
          }                       //в выходную строку
     }
str2[j] = 0;
```

Рассмотрим еще один пример проверки правильности расстановки круглых скобок в строке:

```
char s[256];
int i, c;  //c переменная-счетчик
      cout << "Введите строку";
      gets(s); //вводим строку

      for(c = i = 0; s[i] != 0; i++)
      {
            if(s[i] == '(')   //если скобка открывающая,
                  c++;        //увеличить счетчик;
            if(s[i] == ')')   //если скобка закрывающая,
                  c--;        //уменьшить счетчик
      }
       if(!c)
          //если с = 0, кол-во открывающих
          //и закрывающих скобок равно
            cout << "Текст сбалансирован по скобкам\n";
      else
            cout << "Баланса скобок нет\n";
```

9.2. Функции работы со строками

Стандартная библиотека C предоставляет набор функций для манипулирования строками. Стандартная библиотека C является частью библиотеки C++. Для ее использования мы должны включить заголовочный файл `#include <cstring>`. Функции приведены в таблице.

Набор функций манипулирования строками

Функция	Прототип и краткое описание функции
`strcmp`	`int strcmp(const char *str1, const char *str2);` Сравнивает строки str1 и str2. Если str1 < str2, то результат отрицательный, если str1 = str2, то результат равен 0, если str1 > str2, то результат положительный
`strcpy`	`char* strcpy(char*s1, const char *s2);` Копирует байты из строки s1 в строку s2
`strdup`	`char *strdup (const char *str);` Выделяет память и переносит в нее копию строки str.
`strlen`	`int strlen (const char *str);` Вычисляет длину строки str
`strncat`	`char *strncat(char *s1, const char *s2, int kol);` Приписывает kol символов строки s2 к строке s1
`strncpy`	`char *strncpy(char *s1, const char *s2, int kol);` Копирует kol символов строки s2 в строку s1

Окончание таблицы

Функция	Прототип и краткое описание функции
`strnset`	`char *strnset(char *str, int c, int kol);` Заменяет первые kol символов строки s1 символом c
`atoi`	`int atoi(char *str);` Преобразует строку в целое
`atof`	`float atof(char *str);` Преобразует строку в число с плавающей точкой

Строки при передаче в функцию в качестве фактических параметров могут быть определены либо как одномерные массивы типа `char[]`, либо как указатели типа `char*`. В отличие от массивов, в этом случае нет необходимости явно указывать длину строки.

Для указателя с типом указуемой переменной `char` допускаются различные интерпретации: указатель на отдельный байт; указатель на область памяти – массив байтов; указатель на отдельный символ; указатель на массив символов.

9.3. Практические задания

Для каждой из задач составить блок-схему алгоритма. Использовать индексы и указатели для доступа к элементам строки.

1. Написать программу, которая во введенной с клавиатуры строке меняет местами четные и нечетные слова.

2. Разработать программу, в которой с клавиатуры вводятся две строки символов. К строке с наибольшей длиной добавить текст, содержащийся в другой строке.

3. Создать программу, которая выводит на экран первую часть таблицы кодировки символов (символы с кодами от 0 до 127). Таблица должна состоять из восьми колонок и шестнадцати строк. В первой колонке должны быть символы с кодом от 0 до 15, во второй – от 16 до 31 и т.д.

4. Составить программу, которая выводит на экран сообщение в «телеграфном» стиле: буквы сообщения должны появляться по одной, с некоторой задержкой.

5. Написать программу, которая проверяет, является ли введенная с клавиатуры строка шестнадцатеричным числом.

Глава 10. МНОГОМЕРНЫЕ МАССИВЫ И УКАЗАТЕЛИ

10.1. Объявление и инициализация многомерных массивов

Многомерными массивами в C++ называют массивы, которые имеют 2 и более индексов. Они формализуются списком константных выражений, следующих за идентификатором массива. Причем каждое константное выражение заключается в свои квадратные скобки. Константное выражение в квадратных скобках определяет число элементов по данному измерению массива, так что объявление двухмерного массива содержит два константных выражения, трехмерного – три и т.д.

Обычным представлением многомерных массивов являются таблицы значений, содержащие информацию в строках и столбцах. Чтобы определить отдельный табличный элемент, нужно указать два индекса: первый (по соглашению) указывает номер строки, а второй (по соглашению) указывает номер столбца. Таблицы или массивы, которые требуют двух индексов для указания отдельного элемента, называются двумерными массивами. Компиляторы C++ поддерживают, по меньшей мере, 12-мерные массивы.

Инициализировать многомерные массивы можно также, как одномерные. Например:

```
int a[2][3]; /* представлено в виде матрицы
                a[0][0]  a[0][1] a[0][2]
                a[1][0]  a[1][1] a[1][2]      */
double b[10];
//вектор из 10  элементов имеющих тип double
int w[3][3] = {  { 2, 3, 4 },
                 { 3, 4, 8 },
                 { 1, 0, 9 }
              };
int ia[4][3] = { {0}, {3}, {6}, {9} };
//инициализация первых элементов строк
```

Списки, выделенные в фигурные скобки, соответствуют строкам массива. В случае отсутствия скобок инициализация будет выполнена подряд идущими элементами, недостающие элементы будут неявно инициализированы. Для массива `ia` инициализируются только первые элементы каждой строки. Оставшиеся элементы будут равны нулю.

Если необходимо инициализировать нулями все значения массива, то можно выполнить:

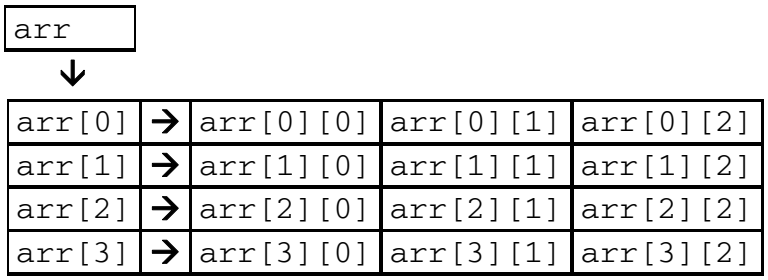
```
long arr[2][3] = { 0 };
```

В языке C/C++ можно использовать сечения массива. Сечения формируются вследствие опускания одной или нескольких пар квадратных скобок. Пары квадратных скобок можно отбрасывать только справа налево и строго последовательно. Для `int s[2][3]` при обращении к `s[0]` будет передаваться нулевая строка массива `s`.

Конструкция `arr[1,2]` является допустимой с точки зрения синтаксиса C++, однако это не объявление двумерного массива размера 1×2. Значение в квадратных скобках – это список выражений через запятую, результатом которого будет последнее значение 2. Поэтому объявление `arr[1,2]` эквивалентно `arr[2]`.

10.2. Указатели и доступ к элементам многомерного массива

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



Для доступа к элементам двумерного массива должны быть использованы два индексных выражения в форме `arr[m][n]` или эквивалентных ей `*(*(arr+m)+n)`, `*(arr[m]+n)` и `(*(arr+m))[n]`. Действительно `arr+m` ссылается на строку номер `m`. Выражение `*(arr+m)` – это адрес нулевого элемента строки `m` , поэтому `*(arr+m)+n` – адрес элемента в строке `m` со смещением `n`. Таким образом, полное выражение ссылается на конкретный элемент массива.

С точки зрения синтаксиса языка указатель `arr` и указатели `arr[0]`, `arr[1]`, `arr[2]`, `arr[3]` являются константами, и их значения нельзя изменять во время выполнения программы.

Инициализация двумерного массива с помощью вложенных циклов будет выглядеть следующим образом:

```
const int rowSize = 3;
const int colSize = 3;
int arr[ rowSize ][ colSize ];

     for ( int = 0; i < rowSize; ++i )
          for ( int j = 0; j < colSize; ++j )
               arr[ i ][ j ] = i + j ;
```

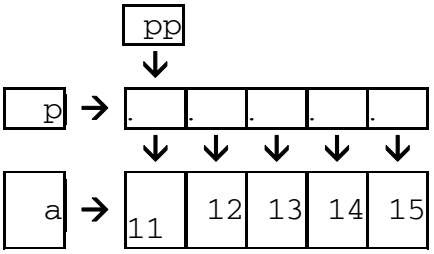
При размещении элементов многомерных массивов они располагаются в памяти *подряд по строкам*. Такой порядок дает возможность обращаться к любому элементу многомерного массива, используя адрес его начального элемента и только одно индексное выражение:

```
float *ptr, *ptr3, arr[4][4][4];
ptr = arr;
ptr3 = arr;
//тогда следующие обращения к элементам будут эквивалентными
//arr[2][3][4] == ptr[3*2+4*3+4] == ptr3[22];
```

Элементы массивов могут иметь любой тип, и, в частности, могут быть указателями на любой тип. Следующие объявления переменных:

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

порождают программные объекты, представленные на рисунке.



Если считать, что `pp = p`, то обращение `*++pp` – значение первого элемента массива `a` (т.е. значение 11), операция `++*pp` изменит содержимое указателя `p[0]` таким образом, что он станет равным значению адреса элемента `a[1]`.

Независимо от размерности массива объем занимаемой им памяти определяется достаточно просто, с помощью операции `sizeof`:

```
Размер массива = sizeof имя_массива/sizeof(тип_данных);
```

В заключение, рассмотрим пример работы с многомерными массивами. Необходимо найти минимальный элемент и запомнить его расположение (номер столбца и строки):

```
int  Matr[N][M];
int iMin = Matr[0][0];    //минимальный элемент
int iCol = 0;             //столбец
int iRow = 0;             //строка
  for(int i = 0; i < N; i++)
             for(int j = 0; j < M; j++)
             {
                   if(iMin > Matr[i][j])
                   {  iMin = Matr[i][j];
              //найти мин элемент и запомнить его место
                              iCol = i;
                              iRow = j;
                   }
             }
```

10.3. Практические задания

Для каждой из задач составить блок-схему алгоритма. Использовать индексы и указатели для доступа к элементам матриц. Дана действительная матрица порядка $n \times m$.

1. Поменять местами строку, содержащую элемент с наибольшим значением в матрице со строкой, содержащей элемент с наименьшим значением. Вывести на экран полученную матрицу. Для каждой строки с нулевым элементом на главной диагонали вывести ее номер и значение наибольшего из элементов этой строки.

2. Среди строк заданной матрицы, содержащих только нечетные элементы, найти строку с максимальной суммой модулей элементов.

3. Найти номер строки заданной матрицы, в которой находится самая длинная серия (последовательность одинаковых элементов).

4. Получить номера строк, элементы каждой из которых образуют монотонную последовательность (монотонно убывающую или монотонно возрастающую).

5. Подсчитать количество строк заданной матрицы, являющихся перестановкой чисел 1, 2, ..., 20.

Глава 11. ФУНКЦИИ

11.1. Определение функций

Функция – это совокупность объявлений и операторов, обычно предназначенная для решения определенной задачи.

Одно из главных преимуществ, предоставляемых функцией, состоит в том, что она может быть выполнена столько раз, сколько необходимо, в различных точках программы. Без такой возможности программы были бы намного больше, поскольку один и тот же код повторялся бы много раз. Необходимость в функциях вызвана также тем, что для независимой разработки и тестирования программу можно разбивать на фрагменты.

Каждая функция должна иметь имя, которое используется для ее объявления, определения и вызова. В любой программе C/C++ должна быть функция с именем `main` (главная функция), именно с этой функции, в каком бы месте программы она не находилась, начинается выполнение программы.

С использованием функций связаны три понятия – определение функции (описание действий, выполняемых функцией), объявление функции (задание формы обращения к функции) и вызов функции.

Определение функции имеет следующую форму:

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

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

Спецификатор-типа функции задает тип возвращаемого значения. Если он не задан, то предполагается, что функция возвращает значение типа `int`.

Имя-функции – либо `main` для основной функции, либо произвольный идентификатор, не совпадающий со служебными словами и именами других объектов программы.

Список-формальных-параметров – это последовательность объявлений формальных параметров вида `<обозначение_типа> <имя_параметра>`, разделенная запятыми. Формальные параметры – это переменные, используемые внутри тела функции и получающие значение при вызове функции путем копирования в них значений

соответствующих аргументов (фактических параметров). Схема отношений между формальными параметрами и аргументами (фактическими параметрами) приведена на рис. 11.1.

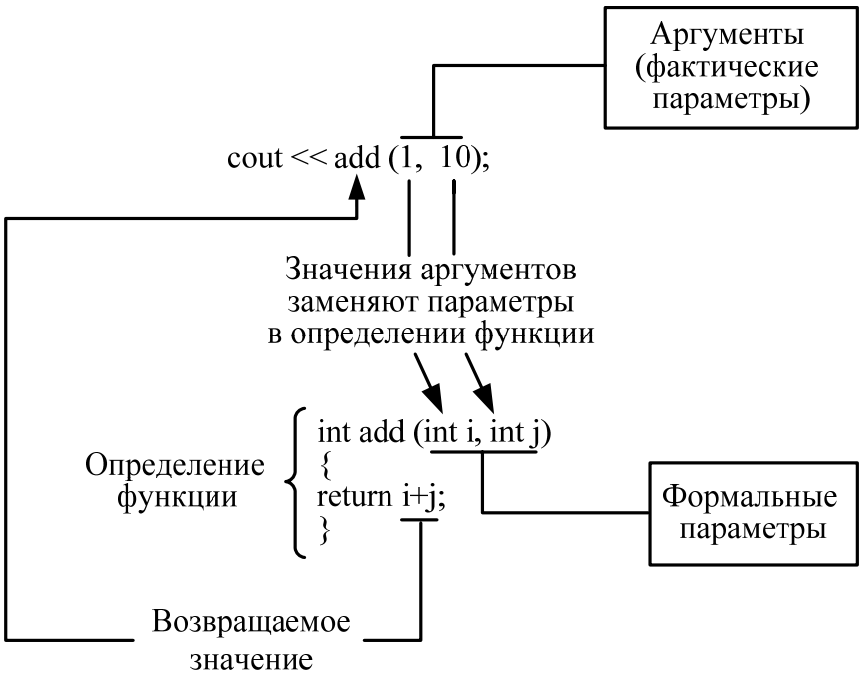


Рис. 11.1. Схема отношений между параметрами функции

Если функция не использует параметров, то наличие круглых скобок обязательно, а вместо списка параметров надо или указать слово `void`, или ничего не указывать.

Тело-функции – это часть определения функции, заключенная в фигурные скобки `{ }`, содержащая операторы, определяющие действие функции.

Для передачи результата из функции в вызывающую функцию используется оператор `return`. Он может использоваться в двух формах: `return;` – завершает функцию, не возвращающую никакого значения (т.е. перед именем функции указан тип `void`) и `return <выражение>;` – возвращает значение выражения, выражение должно иметь тип, указанный перед именем функции. Если программист не пишет оператор `return` явно, то компилятор автоматически дописывает `return` в конец тела функции перед закрывающей фигурной скобкой.

Функция не может возвращать массив или функцию, но может возвращать указатель на любой тип, в том числе и на массив, и на функцию. Тип возвращаемого значения, задаваемый в определении функции, должен соответствовать типу в объявлении этой функции. Пример:

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

Все переменные, объявленные в теле функции без указания класса памяти, имеют класс памяти `auto`, т.е. они являются локальными. При вызове функции локальным переменным отводится память в стеке и производится их инициализация. Управление передается первому оператору тела функции, и начинается выполнение функции, которое продолжается до тех пор, пока не встретится оператор `return` или последний оператор тела функции. Управление при этом возвращается в точку, следующую за точкой вызова, а локальные переменные становятся недоступными. При новом вызове функции для локальных переменных память распределяется вновь, и поэтому старые значения локальных переменных теряются.

Параметры функции передаются по значению и могут рассматриваться как локальные переменные, для которых выделяется память при вызове функции и производится инициализация значениями фактических параметров. При выходе из функции значения этих переменных теряются.

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

11.2. Объявление функций

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

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

В отличие от определения функции, в прототипе за заголовком сразу же следует *точка с запятой*, а тело функции отсутствует. Если

несколько разных функций возвращают значения одинакового типа и имеют одинаковые списки формальных параметров, то эти функции можно объявить в одном прототипе, указав имя одной из функций в качестве имени-функции. Правила использования остальных элементов формата такие же, как при определении функции. Имена формальных параметров при объявлении функции можно не указывать, а если они указаны, то их область действия распространяется только до конца объявления.

Тип возвращаемого значения при объявлении должен соответствовать типу возвращаемого значения в определении функции. Если прототип функции не задан, а встретился вызов функции, то строится неявный прототип из анализа формы вызова функции.

Пример прототипов:

```
int rus (unsigned char r);
int rus (unsigned char);
```

Синтаксической ошибкой является случай, если прототип и определение функции не согласуются.

11.3. Вызов функций

Вызов функции имеет следующий формат:

```
адресное-выражение ([список-выражений]);
```

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

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

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

```
int (*fun)(int x, int *y);
```

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

```
int *fun (intx,int *y);
```

будет интерпретироваться как объявление функции `fun`, возвращающей указатель на `int`.

Вызов функции возможен только после инициализации значения указателя `fun` и имеет вид:

```
(*fun)(i,&j);
```

Любая функция в программе может быть вызвана рекурсивно, т.е. она может вызывать саму себя. Компилятор допускает любое число рекурсивных вызовов. Пример рекурсии – это математическое определение факториала `n!`:

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

11.4. Вызов функции с переменным числом параметров

При вызове функции с переменным числом параметров задается любое требуемое число аргументов. В объявлении и определении такой функции переменное число аргументов задается многоточием в *конце списка* формальных параметров или списка типов аргументов. При этом должен быть указан как минимум один обычный параметр.

```
int sumVal(int first, ...)
  {
    // код функции
   }
```

В перемененном списке параметров нет информации о числе и типах аргументов, поэтому код должен определять, что ему передано при вызове функции. Существует два подхода: когда известно количество параметров, которое передается как обязательный параметр; когда известен признак конца списка параметров.

Примерами функций с переменным числом параметров являются функции из библиотеки (`printf`, `scanf` и т.п.).

Рассмотрим пример функции вычисления суммы значений переменного числа параметров. Список параметров состоит из одного параметра – числа дополнительных параметров:

```
int sum (int, ...); //прототип  функции
.....................
cout << sum(6, 1, 1, 2, 1, 3, 5); //вызов
.....................
int sum (int num_val, ...)
             //num_val – число суммируемых параметров
{
   int *p = &num_val; //выход на начало списка
         int s = 0;
             for (int i = 1; i <= num_val; i++)
             {
             s += *(++p);
             }
   return s;
}
```

11.5. Параметры функции по умолчанию

Часто в самом общем случае функции требуется больше параметров, чем в самом простом и более распространенном случае. Если функция объявляется:

```
extern char* func(long, int = 0);
```

то инициализатор второго параметра является параметром по умолчанию. То есть, если в вызове дан только один параметр, в качестве второго используется параметр по умолчанию.

Пример

```
cout << func(31) << func(32,3);
//интерпретируется как cout << func(31,0) << func(32,3);
```

Параметр по умолчанию проходит проверку типа во время описания функции и вычисляется во время ее вызова. Задавать параметр по умолчанию возможно только для *последних параметров*.

Аргументы по умолчанию должны быть указаны при первом упоминании имени функции – обычно в прототипе. Значения по умолчанию могут быть константами, глобальными переменными или вызовами функций.

11.6. Перегрузка имен функций

В C++ возможно определение нескольких функций с одинаковым именем, но с разными типами формальных параметров и результата.

При этом транслятор выбирает соответствующую функцию по типу аргументов. Например:

```
int max (int arr[], int size);  //прототипы перегруженных
                                //функций
long max (long arr[], int size);
double max (double arr[], int size);

double max (long arr[], int size);
                    //ошибка, недопустимая перегрузка
```

Приведенные функции разделяют одно общее имя, но разные списки параметров. Перегруженные функции можно различать по наличию параметров разного типа либо по различному количеству параметров. Функции не могут быть перегружены, если описание их параметров отличается только модификатором `const` или наличием ссылки. Разные типы возвращаемых значений не могут адекватно различать функции. Поэтому функция `double max (long arr[], int size);` неотличима от `long max (long arr[], int size);` функции.

Фактически все функции (не только перегруженные) должны иметь уникальные сигнатуры, определяемые именем и списком параметров, иначе программа не компилируется.

Назначение перегрузки – разрешить выполнять одно и то же действие с разными операндами. Если имеется серия функций, по сути выполняющих одно и то же, то их необходимо перегружать.

11.7. Передача параметров функции `main`

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

Функция `main` может иметь и третий параметр, который принято называть `argp`, и который служит для передачи в функцию `main` параметров операционной системы (среды), в которой выполняется С-программа. Заголовок функции `main` имеет вид:

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

Если командная строка имеет вид:

```
A:\>cprog working 'C program' 1
```

то аргументы `argc`, `argv`, `argp` представляются в памяти следующим образом:

```
argc  [ 4  ]
argv  [    ]--> [    ]-->   [A:\cprog.exe\0]
                  [    ]-->   [working\0]
                  [    ]-->   [C program\0]
                  [    ]-->   [1\0]
                  [NULL]
argp  [    ]--> [    ]-->   [path=A:\;C:\\0]
                  [    ]-->   [lib=D:\LIB\0]
                  [    ]-->   [include=D:\INCLUDE\0]
                  [    ]-->   [conspec=C:\COMMAND.COM\]
                  [NULL]
```

11.8. Указатели как формальные параметры и результат функций

В большинстве языков программирования параметры функциям передаются либо по ссылке (`by reference`), либо по значению (`by value`). В первом случае функция работает с адресом переменной, переданной ей в качестве фактического параметра. Во втором случае ей доступна не сама переменная, а только ее значение (копия числа). Различие заключается в том, что переменную, переданную по ссылке, функция может модифицировать в вызвавшей ее функции, а переданную по значению – нет. На рис. 11.2 приведена диаграмма механизма передачи по значению.

В языке С параметры передаются только по значению. Следствие применения механизма передачи по значению состоит в том, что функция не может напрямую модифицировать переданные ей аргументы.

Общепринятый способ обеспечить функции непосредственный доступ к какой-либо переменной из вызывающей программы заключается в том, чтобы вместо самой переменной в качестве параметра передавать ее адрес.

```
int first_val = 2;
int second_val = 5;
int result = add (first_val, second_val )
```

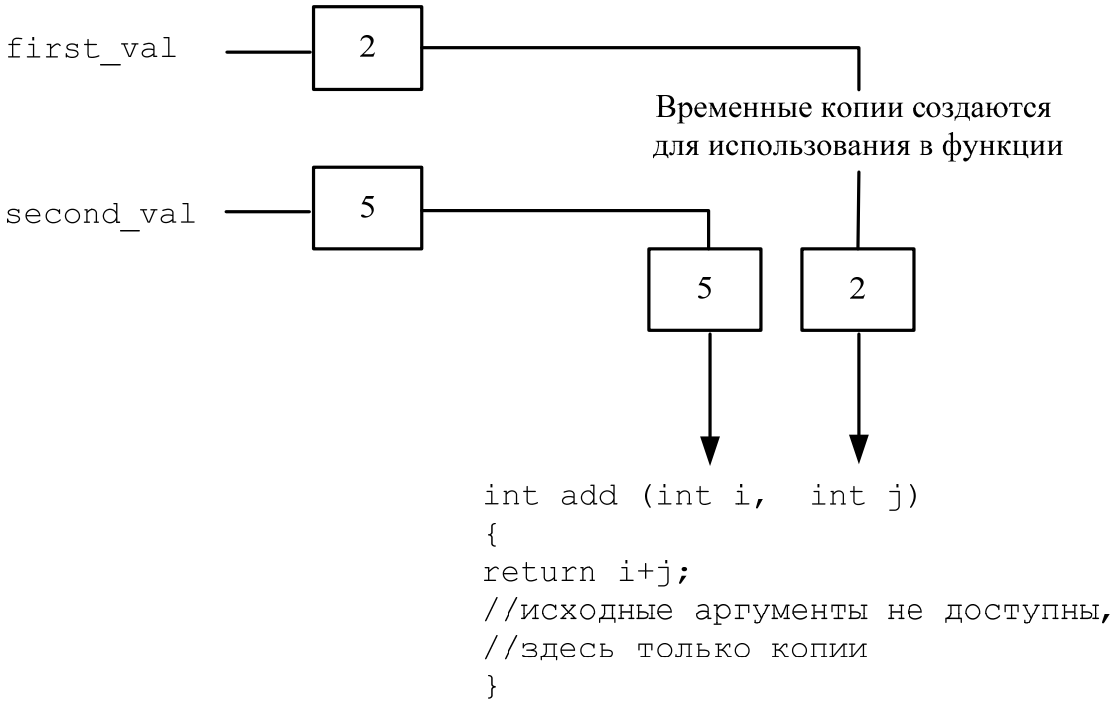


Рис. 11.2. Диаграмма механизма предачи аргумента по значению

Рассмотрим пример, в котором происходит обмен значений переменных:

```
.............................
void swap ( int *, int *); //прототип функции
.............................
int a = 10;
int b = 20;
  swap (&a, &b);    //вызов функции
.............................
void swap ( int *x, int *y)
 {
    int c;          //временная переменная
     c = *x;        //запоминаем x
     *x = *y;       //в x записываем y
     *y = c;        //в y записываем x
 }
```

При вызове функции передаются адреса переменных `a` и `b`. Это означает, что формальные параметры должны быть описаны как

указатели на соответствующий тип данных. При таком описании формальных параметров вызываемой программе становятся доступными адреса переменных из вызывающей программы, которые являются в ней локальными. Если известны адреса, то применимы операции чтения и возврата по адресу.

В функцию можно передавать массивы, но в этом случае массив не копируется, даже несмотря на то, что применяется передача по значению. Имя массива преобразуется в указатель, и копия указателя на начало массива передается в функцию по значению. Однако элементы массива могут быть изменены внутри функции, и поэтому массив – единственный тип, который может быть передан по значению. Например:

```
//вычисление суммы элементов массива вариант 1
int sum (int n, int a[])
     {
     int s = 0;
     for(int i = 0; i < n; i++ )
         s += a[i];
     return s;
     }
.............................
     int a[] = { 3, 5, 7, 9, 11, 13, 15 }; //массив
   int s = sum( 7, a ); //вызов функции sum
     cout << s;
.............................
//вычисление суммы элементов вариант 2
int sum (int n, int *a)
    {
    int s = 0;
    for(int i = 0, s = 0; i < n; s += *a++, i++);
    return s;
    }
.............................
     int a[] = { 3, 5, 7, 9, 11, 13, 15 };
   int s = sum( 7, a ); //вызов функции sum
     cout << s;
```

Во втором варианте функции `sum` везде используется нотация указателей, несмотря на то, что работа идет с массивом. В операторе цикла при суммировании элементов массива `s += *a++` увеличивается адрес массива. Механизм передачи по значению создает копию исходного адреса массива, поэтому в теле функции модифицируется копия, исходный адрес массива остается неизменный.

Возврат результата

Функция в качестве результата может возвращать указатель. В этом случае она обычно «выбирает» некоторую переменную из имеющихся или же динамически «создает» ее:

```
int  *min(int A[], int n)
//функция возвращает указатель
//на минимальный элемент массива
{
int  *pmin, i;
//рабочий указатель, содержащий результат

for (i = 1, pmin = A; i < n; i++)
     if (A[i] < *pmin)
     pmin = &A[i];
return pmin;
}
```

Указатель – результат функции, может ссылаться не только на отдельную переменную, но и на массив.

Существует «железное» правило относительно возвращаемых адресов: *никогда не возвращать из функции адрес локальной автоматической переменной*.

11.9. Ссылки как формальные параметры и результат функции

Передача параметров функции в виде ссылки изменяет метод передачи. Передаваемый параметр выступает псевдонимом передаваемого фактического параметра (аргумента). Это исключает всякое копирование и обеспечивает функции прямой доступ к аргументу. Тогда, нет необходимости в разыменовании указателей на значения, как в предыдущем примере. Например:

```
int incr (int& n)
  {
      n++;
      return n;
  }
.............................
      int s = 5;
cout << incr (s);     //вызов функции incr
```

Здесь ссылка как параметр функции будет создаваться и инициализироваться при каждом вызове функции и разрушатся по ее завершении, т.е. каждый раз получается новая ссылка. Функция

`incr` непосредственно модифицирует переменную, переданную ей в аргументе. Если в качестве аргумента передать числовое значение (например 10), то компилятор выдаст сообщение об ошибке, так как изменять значения констант нельзя.

Использование модификатора `const`

Чтобы сообщить компилятору, что модификации параметров функции не будет, можно применить модификатор `const`. Тогда компилятор будет проверять код функции на предмет изменения значения аргумента. Изменим предыдущий пример:

```
int incr (const int& n)  //функция с константным
                         //аргументом-ссылкой
     {
     // n++;         //ошибка, изменять нельзя
     return n + 1;  //возвратить увеличенное значение
     }
.............................
     int s = 5;
     const int num = 10;
     cout << incr (s) << incr (num);
                     //вызовы функции sum
```

В данной функции имеется прямой доступ к исходному аргументу вызывающего кода (позволяет избежать копирования), и получается полная защита от непреднамеренной модификации (модификатор `const`).

Возврат ссылки

Функция может возвращать ссылку. Так как ссылка не является отдельной сущностью (это псевдоним чего-то другого), важно, что объект, на который она ссылается, все еще существует после завершения выполнения функции. Ссылки как возвращаемые типы особенно важны в контексте объектно-ориентированного программирования. Рассмотрим пример. В массиве, содержащем смешанный набор значений, при вставке нового значения заменяется минимальный элемент.

Несмотря на то, что возврат значения выглядит как значение, тип объявлен как ссылка, и поэтому возвращается не значение `a[j]`, а ссылка на него. Будет ошибкой, если указать в качестве возвращаемого значения `&a[j]`. Это будет означать адрес `a[j]` элемента, то есть указатель.

Никогда не возвращайте из функции ссылку на локальную переменную.

```
double& low_val (double val[], int size);
          //прототип функции, возвращающей ссылку
.........................
       double arr[] = { 3.1, 10.0, 15.9, 23.0, 17.2};
       int arr_size = sizeof arr/ sizeof arr[0];
              //количество элементов массива
.........................
      low_val(arr, arr_size) = 1.0;
          //изменить минимальное значение на 1.0
      low_val(arr, arr_size) = 3.9;
          //изменить минимальное значение на 3.9
.........................
  double& low_val (double val[], int size)
   {
       int j = 0; //индекс наименьшего
        for (int i = 1; i < size; i++)
         if (a[j] > a[i])  //поиск минимального элемента
          j = i;           //запомнить его индекс
       return a[j];
              //вернуть ссылку на минимальный элемент
   }
```

11.10. Статические переменные в функциях

Чтобы создать переменную, чье значение сохраняется от одного вызова функции до другого, можно объявить ее внутри функции с ключевым словом `static`.

```
static int count = 0;
```

Инициализация статической переменной в функции происходит только при первом вызове функции. Затем она продолжает существовать на протяжении всего времени выполнения программы, и любое значение, которое она имеет при завершении функции, остается доступным при следующем вызове.

11.11. Указатель на функцию

Указателем на функцию называется переменная, которая содержит адрес некоторой функции (адрес точки входа в функцию). Общая форма объявления указателя на функцию выглядит следующим образом:

```
тип_возврата (*имя_функции) ([список-параметров]);
```

Соответственно, косвенное обращение по этому указателю представляет собой вызов функции. Определение указателя на функцию может иметь вид:

```
int    (*pf)();           //без контроля параметров вызова
int    (*pf)(int, char*); //с контролем по прототипу
```

Выражение вида `&имя_функции` имеет смысл – начальный адрес функции или указатель на функцию. По аналогии с именем массива использование имени функции без скобок интерпретируется как указатель на эту функцию. Указатель может быть инициализирован и при определении. Возможны следующие способы назначения указателей:

```
int       INC(int a)
          {
             return a+1;
          }
extern    int  DEC(int);
int  (*pf)(int);
pf = &INC;
pf = INC;                 //присваивание указателя
int  (*pp)(int) = &DEC;   //инициализация указателя
```

Передача указателя на функцию

Указатели на функции могут входить в более сложные структуры данных. В программах указатели на функции в основном используются для сохранения адресов программ обработчиков прерываний и для передачи функций в качестве формальных параметров другим функциям. Рассмотрим пример, демонстрирующий такую возможность при вычислении интеграла функций методом площадей:

```
float integral(float (*) (float), float, float, float);
float funct (float); //прототипы функций
.............................
float intefral_value;
intefral_value =
integral(funct(float)0.0,(float)10.0,(float)0.01)
          //вызов функции
.............................
float integral(float (*f) (float), float low, float up,
float delta)
{
     //float (*f) (float) -
     //указатель на интегрируемую функцию
     //float low - нижний предел интегрирования
     //float up - верхний предел интегрирования
     //float delta - шаг приращения по x

   float x, summa = 0.0;
   x = low + delta / 2.0;
```

```
          while (x <= up)
          {
           summa += delta * f(x);
           x = x + delta;
         }
  return summa;
}

float funct (float x)
{
  return (2*x+5); //интегрируемая функция y = 2x+5
}
```

Массивы указателей на функции

Разрешено объявлять массивы указателей на функции (по аналогии с обычными указателями). Их можно инициализировать в объявлении:

```
double one (double, double);    //прототип функции
double two (double, double);    //прототип функции
double three (double, double);  //прототип функции

double (*pfun[3]) (double, double) = {one, two, three};
     //массив указателей на  функции

pfun[1](1.0, 2.0);  //вызов функции one
```

11.12. Практические задания

Выполнить практические задания 8.9, 9.3, 10.3, используя функции для выполнения заданных действий.

Глава 12. ДИНАМИЧЕСКИЕ ОБЪЕКТЫ

Во многих случаях во время выполнения программы у компьютера есть неиспользуемая память. Эта память называется «кучей» или «свободным хранилищем». Внутри этого «хранилища» можно выделить память, и затем освободить выделенное пространство и вернуть его в «свободное хранилище» после того, как необходимость в переменных отпадет. Эта техника позволяет эффективно расходовать память и разрабатывать программы, способные решать масштабные проблемы, обрабатывая большие объемы данных.

Также при традиционном определении массива:

```
тип имя_массива [количество_элементов];
```

общее количество памяти, выделяемой под массив, задается определением и равно `sizeof имя_массива`. Но иногда бывает нужно, чтобы память под массив выделялась для решения конкретной задачи, причем ее размеры заранее не известны и не могут быть фиксированы.

Формирование объектов с переменными размерами можно организовать с помощью указателей и средств динамического распределения памяти двумя способами: с использованием библиотечных функций (C); с использованием операций `new` и `delete` (C++).

12.1. Формирование динамических переменных с использованием библиотечных функций C

При программировании на C++ можно столкнуться с большим числом программ, полученных в наследство от языка C. Поэтому необходимо дополнительно обсудить динамическое выделение памяти в стиле C.

Для выделения и освобождения динамической памяти используются функции, приведенные в табл. 12.1.

Для выделения памяти используется функция `malloc`, параметром которой является размер выделяемого участка памяти, равный `n*sizeof(int)`.

Так как функция `malloc` возвращает нетипизированный указатель `void*`, то необходимо выполнить преобразование полученного нетипизированного указателя в необходимый тип-указатель.

Освободить выделенную память можно функцией `free()`:

```
//Функция для формирования одномерного
//динамического массива
#include <malloc.h>
#include <stdlib.h>
int * make_arr(int n)
   {
int *arr;
   arr = (int*)malloc(n*sizeof(int));

         for(int  i = 0; i < n ; i++)
          arr[i] = rand()% 10;
          return arr;
   }
```

Таблица 12.1

Функции работы с динамической памятью

Функция	Прототип и краткое описание
malloc	void * malloc(unsigned s) Возвращает указатель на начало области динамической памяти длиной в s байт, при неудачном завершении возвращает NULL
calloc	void * calloc(unsigned n, unsigned m) Возвращает указатель на начало области динамической памяти для размещения n элементов длиной по m байт каждый, при неудачном завершении возвращает NULL
realloc	void * realloc(void * p, unsigned s) Изменяет размер блока ранее выделенной динамической памяти до размера s байт, p-адрес начала изменяемого блока, при неудачном завершении возвращает NULL
free	void *free(void p) Освобождает ранее выделенный участок динамической памяти, p-адрес первого байта

Программы на C++ могут поддерживать память, выделяемую с помощью malloc и уничтожаемую с помощью free. Однако лучше использовать операторы new и delete.

12.2. Формирование динамических переменных с использованием операций new и delete

Для динамического распределения памяти в C++ используются операции new и delete. Формат:

```
new имя_типа;
```

или

```
new имя_типа [инициализатор];
```

позволяет выделить и сделать доступным свободный участок памяти, размеры которого соответствуют типу данных, определяемому именем типа. В выделенный участок заносится значение, определяемое инициализатором, который не является обязательным параметром. В случае успешного выделения памяти операция возвращает адрес начала выделенного участка памяти; если участок не может быть выделен, то возвращается `NULL` и программа может быть прервана по исключению. Например:

```
int *i;
i = new int (10);
     //создать переменную типа int, инициализировать - 10
...
float *f;
f = new float;
     //создать динамическую переменную типа float
int *mas = new[5];
     //создать динамический массив из 5 целых элементов
```

Операция `delete` освобождает участок памяти, ранее выделенный операцией `new`.

```
delete i;
     //освободить память по указателю i
...
delete f;
     //освободить память по указателю f
delete [] mas;
     //освободить память, выделенную под массив mas
```

Динамическое выделение памяти – потенциальный источник ошибок, и возможно, наиболее распространенная из них в этом контексте – утечка памяти. Это происходит, когда используется операция `new` для выделения памяти, но не используется операция `delete` для ее освобождения, когда она больше не нужна.

По отношению к массивам динамическое распределение памяти особенно полезно, поскольку иногда бывают массивы больших размеров.

При формировании динамической матрицы сначала выделяется память для массива указателей на одномерные массивы, а затем в цикле с параметром выделяется память под `n` одномерных массивов (рис. 12.1). Например:

```
//Функция для формирования двумерного
//динамического массива
int ** make_matr(int n)
{
int **matr; //адрес адресов – массив указателей
int i,j;
matr = new int*[n];
     //выделить память под массив из n элементов

for (i = 0; i < n ; i++)
     {
     matr[i] = new int[n];
     //выделить память n раз под массив  из n

     for (j = 0; j < n ;j++)
     matr[i][j] = rand()% 10; //инициализация
     }
return matr;
}
```

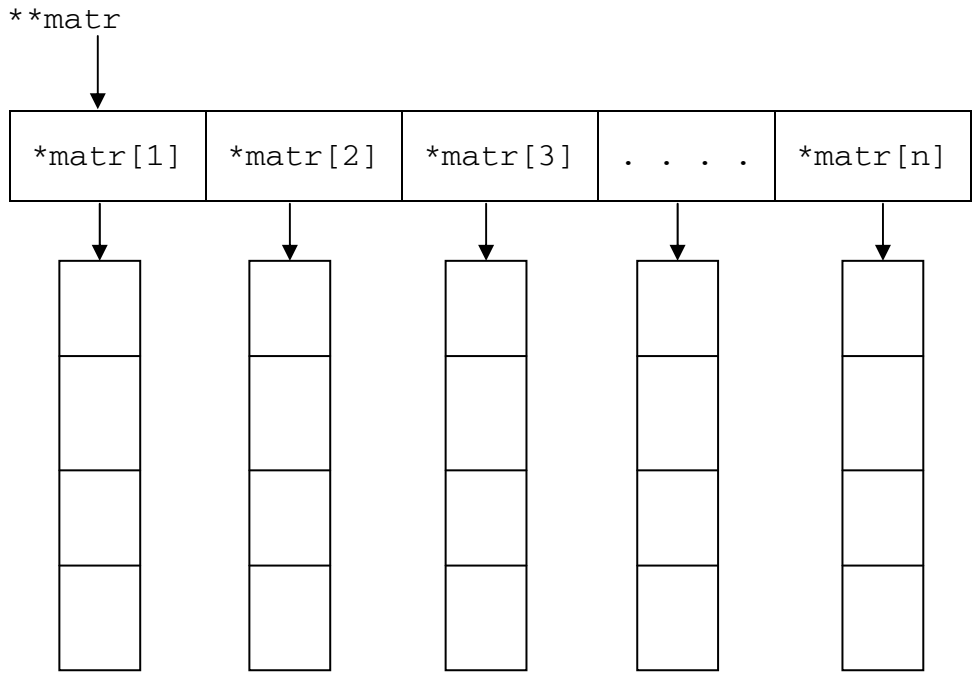


Рис. 12.1. Схема выделения памяти под n одномерных массивов

Нельзя специфицировать начальные значения элементов массива, который распределен динамически. Это следует сделать явным присвоением. Чтобы освободить память, необходимо выполнить цикл для освобождения одномерных массивов:

```
for(int i = 0; i < n ;i++)
    delete matr[i];  //После этого освобождаем память,
                     //на которую указывает указатель matr
 delete [] matr;
```

Для уничтожения динамического массива применяется оператор `delete` с `[]`. Скобки (`[]`) играют важную роль. Они сообщают оператору, что требуется уничтожить все элементы массива, а не только первый.

Константный динамический объект

Чтобы создать динамический объект и запретить изменение его значения после инициализации, необходимо объявить объект константным. Для этого применяется следующая форма оператора `new`:

```
const int *pci = new const int(1024);
```

Константный динамический объект имеет несколько особенностей. Во-первых, он должен быть инициализирован, иначе компилятор сигнализирует об ошибке (кроме случая, когда объект принадлежит к типу класса, имеющего конструктор по умолчанию; в такой ситуации инициализатор можно опустить). Во-вторых, указатель, возвращаемый выражением `new`, должен адресовать константу. В предыдущем примере `pci` служит указателем на `const int`. Константность динамически созданного объекта подразумевает, что значение, полученное при инициализации, в дальнейшем не может быть изменено. Временем его жизни управляет оператор `delete`.

Смешивание способов динамического распределения памяти в стиле `new-delete` со стилем `malloc-free` является логической ошибкой: пространство, созданное с помощью `malloc`, не может быть освобождено с помощью `delete`; объекты, созданные с помощью `new`, не могут быть уничтожены с помощью `free`.

12.3. Массивы указателей

Массив указателей является самой простой и одновременно самой распространенной динамической структурой данных. Одно из его определений имеет вид:

```
double    *p[20];
```

В соответствии с принципом контекстного определения типа данных переменную `p` следует понимать как массив, каждым элементом которого является указатель на переменную типа `double`. Исходя из принятой концепции указателя, эту структуру можно рассматривать

как массив указателей на отдельные переменные типа `double`, так и как указатели на массивы этих переменных:

```
char *pc[] = {"aaa", "bbb", "ccc", NULL};
```

Массивы указателей, как и все остальные структуры данных, содержащие указатели, допускают различные способы формирования, которые отличаются как способом создания самих элементов, так и способом установления связей между ними. Например:

```
double a1,a2,a3, *pd[] = { &a1, &a2, &a3, NULL};
//переменные - статические, указатели инициализируются
double d[19], *pd[20];
.....................

  for (i = 0; i < 19; i++)
          pd[i] = &d[i];

pd[i] = NULL;
.....................
double    *p, *pd[20];

  for (i = 0; i < 19; i++)
   {
     p = new double; //переменные создаются динамически
     *p = i;
     pd[i] = p;      //массив указателей - статически:
   }
pd[i] = NULL;
.....................
double    **pp, *p;
pp = new double *[20];
     //массив указателей создается динамически

 for (i = 0; i < 19; i++)
   {
     p = new double; //переменные создаются динамически
    *p = i;
     pp[i] = p;
   }
pp[i] = NULL;
```

12.4. Практические задания

Выполнить практические задания 8.9, 9.3, 10.3, 11.12, использовать динамические переменные (массивы) и указатели. Интерфейс пользователя осуществить в виде командного процессора: 1 – загрузить данные; 2 – вывести на экран и т.д.

Глава 13. СТРУКТУРЫ, ОБЪЕДИНЕНИЯ, БИТОВЫЕ ПОЛЯ

13.1. Структуры

Структура – это составной объект (пользовательский тип данных), в который входят элементы любых типов, логически связанных между собой. Структуры впервые появились в С, и С++ включает и расширяет понятие структуры в С. В С++ структуры функционально заменяемы классами.

В отличие от массива, который является однородным объектом, структура может быть неоднородной. Элементы структуры не обязательно сохраняются в последовательных байтах памяти. Тип структуры определяется записью вида:

```
struct { список определений } идентификатор ;
```

В структуре обязательно должен быть указан хотя бы один компонент. Определение структур имеет следующий вид:

```
тип-данных описатель;
```

где тип-данных указывает тип структуры для объектов, определяемых в описателях. В простейшей форме описатели представляют собой идентификаторы или массивы.

Пример

```
struct
   {
      double x,y;
   }  s1, s2, sm[9];

struct
   {
      int   year;
      char  moth, day;
   }
      date1, date2;
```

Переменные `s1`, `s2` определяются как структуры, каждая из которых состоит из двух компонент `x` и `y`. Переменная `sm` определяется как массив из девяти структур. Каждая из двух переменных `date1`, `date2` состоит из трех компонентов `year`, `moth`, `day`. Если объявление структуры не содержит ее имени, то переменные структурного типа могут быть объявлены только в описании структуры, а не с помощью их отдельного объявления.

Существует и другой способ ассоциирования имени с типом структуры, он основан на использовании имени структуры:

```
struct имя_структуры { список описаний; };
```

где имя_структуры является идентификатором.

В приведенном ниже примере идентификатор student описывается как имя структуры:

```
struct student
{
  char *name;
  int  id, age;
  char  prp;
 };
```

Имя используется для последующего объявления структур данного вида в форме:

```
struct имя_структуры список-идентификаторов;
```

Пример

```
struct student st1,st2;  //структурированные переменные
student arr_stud [10];   //массив структур
student *p_stud;         //указатель на структуру
```

В C++ во всех случаях использования структурированной переменной, кроме самого определения структуры, служебное слово struct можно опускать, то есть использовать только имя структурированной переменной.

Использование имен структуры необходимо для описания рекурсивных структур. Ниже рассматривается использование рекурсивных структур:

```
struct node
     {
        int data;
        node * next;
     }   st1;
```

Структура node действительно является рекурсивной, так как она используется в своем собственном описании, т.е. в формализации указателя next. Структуры не могут быть прямо рекурсивными, т.е. структура node не может содержать компоненту, являющуюся структурой node, но любая структура может иметь компоненту, являющуюся указателем на свой тип, как и сделано в приведенном примере.

Инициализация структур

Структуры могут быть инициализированы списками значений элементов, заключенных в фигурные скобки и перечисленных через запятую, как и массивы:

```
struct student st1 = { "Петров", //начальное значение name
                         1,     //начальное значение id
                        20,     //начальное значение age
                        'M'     //начальное значение prp
                     };
```

Доступ к компонентам структуры

Доступ к компонентам структуры осуществляется при помощи составных имен двумя способами. Во-первых, с использованием операции принадлежности (.) в виде:

```
идентификатор_структуры.идентификатор_поля
(*указатель_структуры).идентификатор_поля
```

Пример

```
st1.name = "Иванов";
st2.id = st1.id;
st1_node.data = st1.age;
```

Второй способ – при помощи операции косвенной адресации -> (стрелка, минус-больше), которая понимается как выделение элемента в структурированной переменной, адресуемой указателем. Она также называется операцией непрямого доступа к члену:

```
указатель_структуры->идентификатор_поля
(&идентификатор_структуры)->идентификатор_поля
```

Операндами здесь являются указатель на структуру и элемент структуры. Операция имеет полный аналог в виде сочетания операций * и . :

```
struct student *pA, A;
pA = &A;

pA->age   //эквивалентно (*pA).age
```

Работа со структурами поддерживается средствами Intellisense. По мере ввода операции выбора члена структуры вслед за именем структурированной переменной редактор показывает окно со списком всех членов данной структуры.

Объекты типа структур можно передавать как параметры функции и возвращать из функции в качестве результата. Передача структур (особенно больших структур) вызовом по ссылке является более эффективной, чем передача структур вызовом по значению, при которой необходимо копировать всю структуру.

Допустимыми встроенными операциями, которые могут быть выполнены со структурами, являются только следующие: операция присваивания одной структуры другой структуре того же типа; операция адреса (`&`) структуры; операция доступа к элементам структуры и операция `sizeof` для определения размера структуры. Остальные операции, такие как сравнение (`==` и `!=`), не определены. Однако пользователь может определить большинство операции для работы со структурами.

Два структурных типа являются различными, даже когда они имеют одни и те же члены и отличны от основных типов.

Пример

```
struct s1 { int a; };
struct s2 { int a; };
s1 x;
    s2 y = x;     //ошибка: несоответствие типов
    int i = x;    //ошибка: несоответствие типов
```

Вложенные структуры

Структуры могут быть вложенными, т.е. поле структуры может быть связующим полем с внутренней структурой, описание которой должно предшествовать по отношению к основной структуре:

```
struct date
{
   int day, month, year;
};

struct student
{
    char *name;
    date  birth_day;
} ;
student A, *pA;
.......................
A.birth_day.day = 12;     //обращение к полям структуры
pA-> birth_day.day = 20;
struct puple
{
    char *name;
    date *birth_day;
} ;
```

```
puple B, *pB;
.......................
B.birth_day->day = 24;    //обращение к полям структуры
pB-> birth_day->day = 10;
```

В заключение необходимо сказать, что понятие структуры в C++ выходит далеко за пределы оригинальной концепции языка C – оно включено в объектно-ориентированное понятие класса. Ключевые слова `struct` и `class` почти идентичны в C++, за исключением управления доступом к членам.

13.2. Объединения (смеси)

Объединение – это поименованная совокупность данных разных типов, размещаемых с учетом выравнивания в одной и той же области памяти, размер которой достаточен для хранения наибольшего элемента. Объединение подобно структуре, однако в каждый момент времени может использоваться только один из элементов объединения. Тип объединения может задаваться в следующем виде:

```
union идентификатор_объединения {
          описание элемента 1;
          ...
          описание элемента n;  };
```

Главной особенностью объединения является то, что для каждого из объявленных элементов выделяется *одна и та же область памяти*, т.е. они перекрываются. Хотя доступ к этой области памяти возможен с использованием любого из элементов, элемент для этой цели должен выбираться так, чтобы полученный результат не был бессмысленным.

В разные отрезки времени выполнения программы некоторые объекты могут быть не нужны, т.е. программе требуется только часть ее объектов. Вместо того, чтобы впустую растрачивать память на объекты, которые используются не постоянно, можно поместить их в объединение, где они будут делить между собой одну и ту же область памяти.

Память, которая соответствует переменной типа объединения, определяется величиной, необходимой для размещения наиболее длинного элемента объединения. Когда используется элемент меньшей длины, то переменная типа объединения может содержать неиспользуемую память.

Пример

```
union {
          char  name[30];
          char  adress[80];
          int   age;
          int   telephon;
      } information;
union {
         int ax;
         char al[2];
      }   A;
```

При использовании объекта `information` типа `union` можно обрабатывать только тот элемент, который получил значение, т.е. после присвоения значения элементу `information.name` не имеет смысла обращаться к другим элементам. Объединение `A` позволяет получить отдельный доступ к младшему `A.al[0]` и к старшему `A.al[1]` байтам двухбайтного числа `A.ax`.

Типичной ошибкой является инициализация объединения при его объявлении значением или выражением, тип которого отличается от типа первого элемента объединения.

Объектам с типом объединения можно присваивать любой класс памяти, кроме `register`. Ни один из членов объединения не может быть объявлен со спецификатором класса памяти `static`.

Единственными допустимыми встроенными операциями, которые могут выполняться над объединениями, являются: операция присваивания значения одного объединения другому объединению того же типа; операция вычисления адреса объединения (`&`) и доступ к элементу объединения при помощи операций доступа к элементу структуры (`.` и `->`). Над объединениями не могут выполняться операции сравнения по тем же самым причинам, по каким они не выполняются над структурами.

13.3. Битовые поля

Элементом структуры может быть битовое поле, обеспечивающее доступ к отдельным битам памяти. Битовые поля позволяют рационально использовать память с помощью хранения данных в минимально требуемом количестве битов. Элементы битового поля должны быть объявлены как тип `int` или `unsigned`.

Вне структур битовые поля объявлять нельзя. Нельзя также организовывать массивы битовых полей и нельзя применять к полям операцию определения адреса. В общем случае тип структуры с битовым полем задается в следующем виде:

```
struct {unsigned идентификатор 1 :  длина-поля  1;
        unsigned идентификатор 2 :  длина-поля  2;  }
```

Длина поля задается целым выражением или константой. Эта константа определяет число битов, отведенное соответствующему полю. Битовое поле рассматривается как целое число, максимальное значение которого определяется длиной поля. Например:

```
struct number {
           unsigned group: 4;        //4 бита от 0 до 15
           unsigned department: 3;   //3 бита от 0 до 8
           unsigned course: 3;       //3 бита от 0 до 8
              }
```

Это описание включает три битовых поля типа `unsigned`: `group`, `department` и `course`, используемых для представления номера зачетки. При объявлении битового поля вслед за указанием типа элемента `unsigned` или `int` ставится двоеточие (`:`) и пишется целочисленная константа, задающая ширину поля (т.е. число битов, в которых хранится этот член структуры). Приведенное выше описание структуры показывает, что для хранения члена `group` выделено 4 бита, для `department` – 3 бита и для `course` – 3 бита. Количество битов определяется ожидаемым диапазоном значений для каждого члена структуры. Член структуры `group` хранит значения от 0 до 12 в области памяти размером 4 бита (4 бита, выделенные для элемента `group`, могут хранить значения от 0 до 15). Член структуры `department` может хранить значения от 0 до 8 (факультеты). Область памяти размером 3 бита, выделенная для члена `course`, будет хранить значения от 0 до 4 (диапазон от 0 до 8).

Сгенерируем студентам номера зачеток:

```
number Idip [90];
.......................
void FillNumber( number * const doc )
{
     for ( int i = 0; i <= 99; i++)
       {
        doc[i].group = i % 3 + 9;
        doc[i].department = 5;
        doc[i].course = 1;
       }
}
```

Допустимы неименованные поля; они не влияют на смысл именованных полей, но могут улучшить размещение (рис. 13.1):

```
struct { unsigned a1 :  4;
         unsigned    :  2;      //неиспользуемое
         unsigned a3 :  5;
         unsigned a4 :  2;
       } prim1;
```

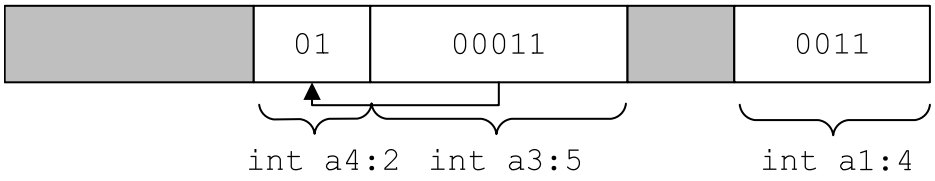


Рис. 13.1. Битовые поля

Поле нулевой длины обозначает выравнивание на границу следующего слова:

```
struct { unsigned b1 :  1;
         unsigned    :  0;
         unsigned b3 :  5;
         unsigned b4 :  2;
       } prim2;
```

Хотя битовые поля сокращают требования к памяти, их использование может привести к тому, что компилятор будет генерировать машинный код, который выполняется с низкой скоростью. Это происходит вследствие того, что приходится использовать дополнительные операции машинного языка для получения доступа к отдельным частям адресуемых элементов памяти, и является одним из множества примеров необходимости компромисса между требованиями эффективности по памяти и по времени выполнения программы.

13.4. Перечисления

Объект *перечислимого типа* представляет собой объект, значения которого выбираются из фиксированного множества идентификаторов, называемых константами перечислимого типа.

Пример

```
enum languages{ C, Java, Ada, PHP, Basic } ;
languages master;
```

или

```
enum languages{ C, Java, Ada, PHP, Basic } master;
```

определяет новый перечислимый тип `languages` с пятью целыми константами, называемыми перечислителями, и присваивает им значения. Значения перечислителей по умолчанию присваиваются начиная с 0 в порядке возрастания, т.е. это эквивалентно записи:

```
const    C = 0;
const Java = 1;
const  Ada = 2;
const  PHP = 3;
```

Перечисление может быть не именованным:

```
enum key { A, B, C };
```

Имя перечисления становится синонимом `int`, а не новым типом.

Пример

```
enum key { A, B, C };
.....................
key new;
 switch (new) {
  case A:
      ...       break;
  case B:
      ...       break;
  }
```

Задавать значения перечислителей можно явно:

```
enum languages{
               C = 1,
               Java,
               Ada = 9,
               PHP,
               Basic = 9;
              } ;
```

Тем константам, значения которых явно не заданы, присваивается значение предшествующей константы, увеличенное на единицу. Таким образом, константе `Java` соответствует значение 2, а константе `PHP` – значение 10. Разным константам перечислимого типа может соответствовать одно и то же значение (`Basic, Ada`).

Объекты перечислимого типа, которым присвоены значения констант перечислимого типа, можно использовать в любом выражении вместо целых констант. Вместо них подставляются соответствующие им целые значения. Им можно присваивать любой класс хранения, кроме `register`.

Практическое назначение перечисления – определение множества различающихся символических констант целого типа.

13.5. Практические задания

Определить структурированный тип, набор функций (в виде меню) для работы с массивом структур. В структурированной переменной предусмотреть способ отметки ее как не содержащей данных (т.е. «пустой»). Функции должны работать с массивом структур или с отдельной структурой через указатели, а также при необходимости возвращать указатель на структуру. В перечень обязательных функций входят: «очистка» структурированных переменных; поиск свободной структурированной переменной; ввод элементов (полей) структуры с клавиатуры; вывод элементов (полей) структуры с клавиатуры; поиск в массиве структуры с минимальным значением заданного поля; сортировка массива структур в порядке возрастания заданного поля (при сортировке разрешается присваивание структурированных переменных); удаление заданного элемента; изменение (редактирование) заданного элемента. Интерфейс пользователя осуществить в виде командного процессора.

1. Клиенты банка. Фамилия И.О., номер счета, сумма на счете, дата последнего изменения. Выбор по произвольному шаблону.
2. Читатели. Фамилия И.О., номер читательского билета, название книги, срок возврата. Выбор по произвольному шаблону.
3. База данных авиарейсов. Номер рейса, пункт назначения, время вылета, дата вылета, стоимость билета. Выбор по произвольному шаблону.
4. Справочник лекаря. База болезней: название, симптомы, процедуры, перечень рекомендуемых лекарств с указанием требуемого количества. Формирование рецепта после осмотра больного.
5. Преподаватели. Название экзамена, дата экзамена, фамилия преподавателя, количество оценок, оценки. Выбор по произвольному шаблону.

Глава 14. РАБОТА С ФАЙЛАМИ

14.1. Файловый ввод-вывод в С

Особенностью С является отсутствие в этом языке структурированных файлов. Все файлы рассматриваются как неструктурированная последовательность байтов. Каждый файл завершается маркером конца файла (end-of-file marker или EOF) или указанным числом байтов, записанным в служебную структуру данных поддерживающей системой.

Когда файл открывается, то создается объект и с этим объектом связывается поток. Поток – это файл вместе с предоставленными средствами буферизации. Потоки можно открывать и закрывать (связывать указатели на поток с конкретными файлами); вводить и выводить строку, символ, форматированные данные, порцию данных произвольной длины; анализировать ошибки ввода-вывода и достижения конца файла; управлять буферизацией потока и размером буфера; получать и устанавливать указатель текущей позиции в файле.

Прежде, чем начать работать с потоком, его надо инициировать (открыть). При этом поток связывается со структурой предопределенного типа `FILE`, определение которой находится в библиотечном файле `<stdio.h>`. В структуре находится указатель на буфер, указатель на текущую позицию файла и т.п. При открытии потока возвращается указатель на поток, т.е. на объект типа `FILE`:

```
#include <stdio.h>
int _tmain(int argc, _TCHAR* argv[])
{.....................
FILE *fp;
.....................
fp = fopen( "t.txt", "r");
.....................}
```

где `fopen(<имя_файла>,<режим_открытия>)` – функция для инициации файла.

В стандартной библиотеке каждый файл представлен структурированной переменной, в которой сосредоточена вся информация об открытом файле: тип, идентификатор файла в операционной системе (номер, handle), буфер на 1 блок (сектор), текущее состояние и способ работы с файлом:

```
type struct    {
short level;   //число оставшихся в буфере непрочит. байт;
               //обычный размер буфера – 512 байт;
unsigned flags;
               //флаг статуса файла: чтен., запись, дополн.
char fd;       //дескриптор файла (номер)
unsigned char hold;
               //непереданный символ, ungetch-символ;
short bsize;   //размер внутреннего промежуточного буфера;
unsigned char buffer;    //значение указателя
                         //для доступа внутри буфера;
unsigned char *curp;     //текущее значение указателя
                         //для доступа внутри буфера;
unsigned istemp;         //флаг временного файла;
short token;             //флаг при работе с файлом;
               }  FILE;
```

Часто ее называют описателем или дескриптором файла. При открытии файла функция `fopen` создает переменную – дескриптор файла, и возвращает указатель на нее. Программа должна его запомнить и в дальнейшем использовать при всех обращениях к файлу для его идентификации.

В табл. 14.1 приведены режимы для открытия файла.

Таблица 14.1

Режимы открытия файла

Режим	Описание
`"w"`	Открыть файл для записи, если файл существует, то он стирается
`"r"`	Открыть файл для чтения
`"a"`	Открыть файл для добавления, если файл существует, то он не стирается и можно писать в конец файла
`"w+"`	Открыть файл для записи и исправления, если файл существует, то он стирается, а далее можно и читать, и писать, размеры файла можно увеличивать
`"r+"`	Открыть файл для чтения и записи, но увеличить размер файла нельзя
`"a+"`	Открыть файл для добавления, т.е. можно и читать, и писать, в том числе и в конец файла

Поток можно открыть в текстовом (`t`) или двоичном (`b`) режиме. Текстовые файлы являются по своей природе файлами последовательного доступа. Двоичный файл выступает как аналог внутренней памяти компьютера и организован соответствующим образом. По умолчанию устанавливается текстовый режим. В явном виде режим указывается следующим образом: `"r+b"` или `"rb"` – двоичный (бинарный) режим. Например:

```
if ((fp = fopen("t.txt", "w") == NULL)
   {
    perror("\n ошибка при открытии файла");
//выводит строку символов с сообщением об ошибке
    exit(0);
   }
//после работы с файлом, его надо закрыть
     fclose(<указатель_на_поток>);
```

Если по какой-либо причине открытия файла не произошло, то функция `fopen` вернет `NULL`.

После того, как программа с данным файлом отработала, следует «отвязать» структуру `FILE` или закрыть файл с помощью функции `fclose (дескриптор файла)`. Она не только разрывает связь структуры с файлом, но и записывает в память оставшееся содержимое буфера ввода-вывода.

Для закрытия нескольких файлов введена функция, объявленная следующим образом:

```
void fcloseall(void);
```

Если требуется изменить режим доступа к файлу, то для этого сначала необходимо закрыть данный файл, а затем вновь его открыть, но с другими правами доступа. Для этого используют стандартную функцию:

```
FILE* freopen
    (char* имя_файла, char *режим, FILE *дескриптор_файла);
```

Эта функция сначала закрывает файл, объявленный `дескриптором_файла` (как это делает функция `fopen`), а затем открывает файл с `именем_файла` и правами доступа `режим`.

14.2. Основные функции для работы с файлами в С

После того, как файл открыт для чтения или записи, используются специальные функции:

```
int fread( void *ptr, int size, int n, FILE *fp);
```

где `void *ptr` – указатель на область памяти, в которой размещаются считываемые из файла данные; `int size` – размер одного считываемого элемента; `int n` – количество считываемых элементов; `FILE *fp` – указатель на файл, из которого производится считывание. В случае успешного считывания информации функция возвращает число прочитанных элементов (а не байтов), иначе – `EOF`;

```
int fwrite( void *ptr, int size, int n, FILE *fp);
```

где `void *ptr` – указатель на область памяти, в которой размещаются записываемые в файл данные; `int size` – размер одного записываемого элемента; `int n` – количество записываемых элементов; `FILE *fp` – указатель на файл, в который производится запись. В случае успешной записи информации функция возвращает число записанных элементов, иначе – `EOF`. Например:

```
typedef STRUCT
{
  char name [40];
  char post [40];
  float rate;
}       EMPLOYEE;
.......................
int _tmain(int argc, _TCHAR* argv[])
{
    FILE *f;                //указатель, связанный с файлом
    EMPLOYEE e;             //переменная
    EMPLOYEE mas[10];       //массив
                            //открываем файл
    if ((f = fopen("f.dat", "wb") == NULL) exit(1);
//если при открытии файла возникает ошибка,
//то выходим из функции
    int i;
      for(i = 1; i <= 10; i++)
       {
                            //формируем запись e
        printf("name="); scanf("%s",&e.name);
        printf("post="); scanf("%s",&e.post);
        printf("rate="); scanf("%f",e.rate);
                            //записываем запись e в файл
        fwrite(&e, sizeof(EMPLOYEE),1,f);
        if (ferror(f) == NULL) exit(2);
       }
    fclose(f);
                            //чтение записей из файла
    if ((f = fopen("f.dat", "rb") == NULL) exit(3);
//если при открытии файла возникает ошибка,
//то выходим из функции
    i=0;
    while(!feof(f) && i <= 10)
       {
        fread(&mas[i], sizeof(EMPLOYEE),1,f);
        i++;
       }
    fclose(f);
}
```

Строковый ввод-вывод

Для построчного ввода-вывода используются следующие функции:

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

где `char *s` – адрес, по которому размещаются считанные байты; `int n` – количество считываемых байтов; `FILE *fp` – указатель на файл, из которого производится считывание.

Прием символов заканчивается после передачи `n` байтов или при получении `\n`. Управляющий символ `\n` тоже передается в принимающую строку. В любом случае строка заканчивается `\0`. При успешном завершении считывания, функция возвращает указатель на прочитанную строку, иначе возвращает `NULL`.

```
char *fputs(char *s, FILE *F);
```

где `char *s` – адрес, из которого берутся записываемые в файл байты; `FILE *fp` – указатель на файл, в который производится запись. Например:

```
int MAXLINE=255; //максимальная длина строки
FILE    *in,     //исходный файл
        *out;    //принимающий файл
char buf[MAXLINE];
     //строка, с помощью которой выполняется копирование
     //строк одного файла в другой
while (fgets (buf, MAXLINE, in)!=NULL)
fputs(buf,out);
```

Позиционирование в файле

С открытым файлом связано понятие «текущей позиции» (позиционера). *Текущая позиция* – номер байта, начиная с которого производится очередная операция чтения-записи. При открытии файла текущая позиция устанавливается на начало файла, после чтения-записи порции данных перемещается вперед на размерность этих данных. Для дополнения файла новыми данными необходимо установить текущую позицию на конец файла и выполнить операцию записи. Текущая позиция представляется в программе переменной типа `long`. Для работы с ней в стандартной библиотеке имеются две функции:

```
long  ftell(FILE *fp);
```

Она возвращает текущую позицию в файле. Если по каким-то причинам текущая позиция не определена, функция возвращает `-1L`. Вторая функция устанавливает текущую позицию в файле на байт с номером `pos`:

```
int    fseek(FILE *fp, long pos, int mode);
```

Параметр `mode` определяет, относительно чего отсчитывается текущая позиция в файле, и имеет следующие символические и числовые значения (установленные в `stdio.h`):

```
#define  SEEK_SET 0 //относительно начала файла –
                    //позиция 0
#define  SEEK_CUR 1 //относительно текущей позиции,
                    //>0 – вперед, <0 – назад
#define  SEEK_END 2 //относительно конца файла
                    //(значение pos – отрицательное)
```

Функция `fseek` возвращает значение 0 при успешном позиционировании и `-1` (`EOF`) – при ошибке. Получить текущую длину файла можно позиционированием:

```
long fsize;
fseek(fl,0L,SEEK_END);
                    //установить позицию на конец файла
fsize = ftell(fd);  //прочитать значение текущей позиции
```

Есть еще одна функция позиционирования в файле на начало потока:

```
rewind (fp);
```

Функция эквивалентна функции `fseek` за исключением того, что сбрасывается индикатор конца файла и индикатора ошибок. После функции `rewind` можно выполнять операции обновления файлов.

Функция

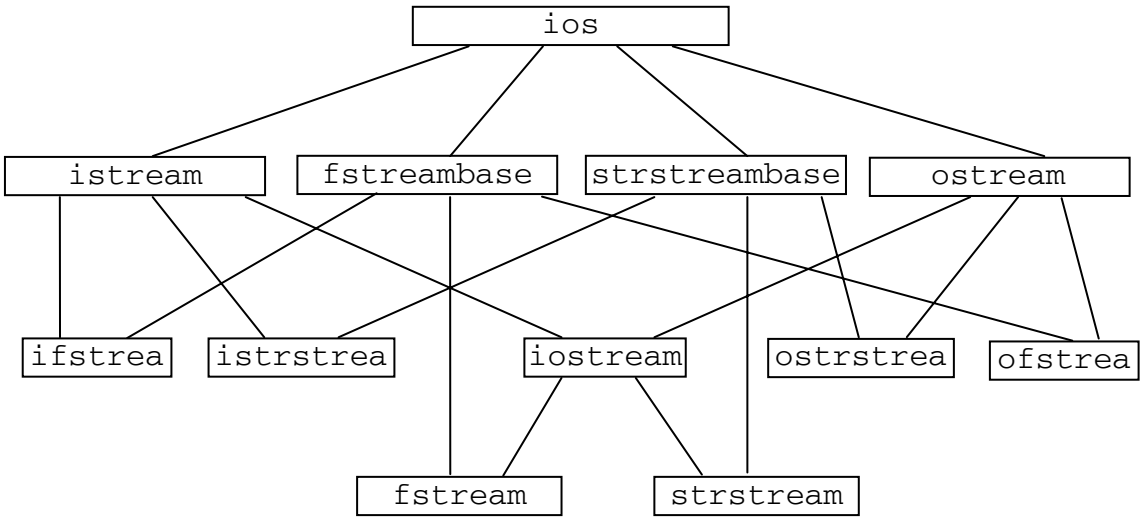
```
ferror(fp);
```

тестирует поток на ошибки чтения-записи. Если в файле была обнаружена ошибка, то функция `ferror` возвращает ненулевое значение. Для обработки ошибок следует записывать эту функцию перед блоком работы с файлом.

14.3. Файловый ввод-вывод в C++

Для обработки файлов в C++ должны быть включены заголовочные файлы `<iostream>` и `<fstream>`. Файл `<fstream>` включает определения классов потоков `ifstream` (для ввода в файла), `ofstream` (для вывода из файл) и `fstream` (для ввода-вывода файлов). Файлы открываются путем создания объектов этих классов потоков. Эти классы потоков являются производными (т.е. наследуют функциональные возможности) от классов `istream`, `ostream` и `iostream` соответственно.

Таким образом, функции-члены, операции и манипуляторы для работы с потоками, описанные в главе 4, могут быть также применены и к потокам файлов. Отношения наследования классов ввода-вывода представлены на рисунке.



Отношения наследования классов ввода-вывода

Здесь `istream` – класс входных потоков; `ostream` – класс выходных потоков; `iostream` – класс ввода-вывода; `istrstream` – класс входных строковых потоков; `ifstream` – класс входных файловых потоков и т.д.

При использовании поточных классов языка C++ в программе требуется использовать стандартное пространство имен (`using namespace::std`).

Файловый ввод-вывод организован с помощью переопределенных в поточных классах операций включения (<<) и извлечения (>>). Операции << и >> имеют два операнда. Левым операндом является объект класса `istream` (`ostream`), а правым – данное.

Каждый поток имеет связанное с ним состояние. Состояния потока описываются в классе `ios` в виде перечисления `enum`:

```
enum io_state  {
     goodbit,  //нет ошибки 0X00
     eofbit,   //конец файла 0X01
     failbit,  //последняя операция не выполнилась 0X02
     badbit,   //попытка использования недопустимой
               //операции 0X04
     hardfail  //фатальная ошибка 0X08
               };
```

Флаги, определяющие результат последней операции с объектом `ios`, содержатся в переменной `state`. Получить значение этой переменной можно с помощью функции `int rdstate()`.

Кроме того, проверить состояние потока можно следующими функциями:

```
int bad();      //1, если badbit или hardfail
int eof();      //1, если eofbit
int fail();     //1, если failbit, badbit или hardfail
int good();     //1, если goodbit
```

Работа с классом `fstream`

Члены этого класса позволяют открывать файл, записывать в него данные, перемещать указатель позиционирования, читать данные.

Объекты классов файлового ввода-вывода разрешено определять и без указания имени файла. Позже к этому объекту можно присоединить файл с помощью функции открытия файла со следующей сигнатурой:

```
void open
    (const char* name, int mode, int p = filebuf::openprot);
```

Пример

```
fstream inoutFile;
inoutFile.open("a.txt");
if ( ! inoutFile )     //открытие успешно
```

При открытии файла задается режим открытия файла (`mode`) (табл. 14.2), он определен в базовом классе `ios`, поэтому обращение к значениям в классе `fstream` должно идти с указанием класса родителя.

Таблица 14.2

Режимы открытия файла

Режим	Описание
`ios_base::app`	Открыть файл для дозаписи в его конец
`ios_base::binary`	Открыть файл в бинарном режиме
`ios_base::in`	Открыть файл для чтения
`ios_base::out`	Открыть файл для записи, если файл не существует, он будет создан
`ios_base::trunc`	Уничтожить содержимое файла, если файл существует (очистить файл)
`ios_base::ate`	Установить указатель позиционирования файла на его конец

Для задания нескольких режимов используется оператор побитового ИЛИ.

Для сброса буфера потока, отсоединения потока от файла и закрытия файла используется функция:

```
void fstreambase::close();
```

Эту функцию необходимо явно вызвать при изменении режима работы с потоком. Автоматически она вызывается только при завершении программы.

Объект класса `fstream` можно позиционировать с помощью функций-членов `seekg()` или `seekp()`. Здесь буква `g` обозначает позиционирование для чтения (`getting`) символов (используется с объектом класса `ofstream`), а `p` – для записи (`putting`) символов (используется с объектом класса `ifstream`). Эти функции делают текущим тот байт в файле, который имеет указанное абсолютное или относительное смещение. У функций есть два варианта. В первом варианте текущая позиция устанавливается в некоторое абсолютное значение, заданное аргументом `current_position`, причем значение 0 соответствует началу файла:

```
seekg( pos_type current_position);
//смещение от текущей позиции в том или ином направлении
```

Второй вариант устанавливает указатель рабочей позиции файла на заданное расстояние от текущей, от начала файла или от его конца в зависимости от аргумента `dir`, который может принимать следующие значения: `ios_base::beg` – от начала файла; `ios_base::cur` – от текущей позиции; `ios_base::end` – от конца файла:

```
seekg(off_type offset_position, ios_base::seekdir dir);
```

Текущая позиция чтения в файле типа `fstream` возвращается любой из двух функций-членов `tellg()` или `tellp()`. Рассмотрим пример, в котором записывается строка текста в файл, затем она читается из файла в буфер:

```
#include <fstream>
#include <iostream>
.......................
int _tmain(int argc, _TCHAR* argv[])
{
using namespace std;
char p[100];
fstream inout;
inout.open ("a.txt", ios:: base::in | ios:: base::out
                              | ios:: base::trunc);
                    //создание двунаправленного потока
```

```
inout << "This is string" << endl;  //вывод в файл
inout.seekg (0);       //установка позиционера на начало
inout.getline (p,50);  //чтение строки из файла
inout.seekg (0);  //установка позиционера на начало файла
cout << endl << inout.rdbuf();
                  //вывод содержимого потока на экран
inout.close();    //закрыть поток
}
```

Работа с классом `ofstream`

Если файл будет использоваться только для вывода, определяется объект класса `ofstream`. Например:

```
ofstream outfile("copy.out", ios::base::out );
ofstream outfile2("copy.out");  //эти объекты эквивалентны
```

Класс `ofstream` является производным от `ostream` (см. рисунок). Поэтому все определенные в `ostream` операции применимы и к `ofstream`.

В классе `ostream` определены следующие функции:

```
ostream& put(char C);
```

альтернативный метод вывода символа `C` в выходной поток `ostream`.

Функция-член `write()` класса `ostream` дает альтернативный метод вывода массива символов. Вместо того, чтобы выводить символы до завершающего нуля, она выводит указанное число символов, включая и внутренние нули, если таковые имеются. Сигнатура функции:

```
ostream& write(const char* buffer,int size);
```

записывает в `ostream` содержимое буфера `buffer` количеством `size` символов. Буфер записывается без форматирования.

Пример использования класса `ofstream`:

```
#include <fstream>
.......................
int _tmain(int argc, _TCHAR* argv[])
{
ofstream out;      //объявляем переменную типа ofstream
.......................
out.open ("a.txt");  //вызываем метод открытия файла
if ( out == NULL) return 0;  //выход, если не открыли файл
for (int i = 0; i < 2; i++)
   out << "string" << i << endl;  //вывод в файл
   out.close();                   //закрытие файла
.......................
}
```

Работа с классом `ifstream`

Чтобы открыть файл только для чтения, применяется объект класса `ifstream`, производного от `istream`.

Иногда необходимо прочитать из входного потока последовательность не интерпретируемых байтов, а типов данных, таких как `char`, `int`, `string` и т.д. Функция-член `get()` класса `istream` читает по одному байту, а функция `getline()` читает строку, завершающуюся либо символом перехода на новую строку, либо каким-то иным символом, определяемым пользователем. Сигнатура `getline()`:

```
getline (char * buffer, streamsize size, char delimiter='\n');
```

Парной для функции `write()` из класса `ostream` является функция `read()` из класса `istream` с сигнатурой:

```
read (char* buffer, streamsize size);
```

читает `size` соседних байт из входного потока и помещает их, начиная с адреса `buffer`. Функция `gcount()` возвращает число байт, прочитанных при последнем обращении к `read()`. В свою очередь `read()` возвращает объект класса `istream`, для которого она вызвана.

```
#include <fstream>
.......................
int _tmain(int argc, _TCHAR* argv[])
{
ifstream in;        //объявляем переменную типа ifstream
char c;
.......................
in.open ("c:\\a.txt");   //вызываем метод открытия файла
while (!in.eof());       //пока не конец файла
   {
     c = in.peek();
               //считываем символ из файла без извлечения
     if (c == 'A')       //если символ A
         {
           in.seekg(in.tellg()+1);
                         //передвинуть позиционер на один
                         //пропустить символ
           continue;     //на продолжение цикла
          }
     in.get(c);          //чтение символа
     cout << c;
    }
in.close();
}
```

14. 4. Практические задания

1. Разработать программу управления матрицами (добавление и удаление строк или столбцов). Матрица хранится в файле. Выполнить задание с использованием функций С и потоковых классов С++.

2. Разработать программу удаления в тексте, содержащемся в файле, лишних пробелов. Выполнить задание с использованием функций С и потоковых классов С++.

3. Определить, равны ли два заданных файла.

4. Подсчитать число цифр в тексте и их сумму. Текст находится в заданном текстовом файле.

5. Определить, какая буква чаще всего встречается в тексте, находящемся в заданном текстовом файле.

Глава 15. СЛОЖНОСТЬ АЛГОРИТМОВ. АЛГОРИТМЫ СОРТИРОВКИ

15.1. Принципы анализа алгоритмов

Процесс создания компьютерной программы для решения практической задачи состоит из нескольких этапов: формализация и создание технического задания; разработка алгоритма решения задачи; написание, тестирование, отладка и документирование программы; получение решения исходной задачи путем выполнения законченной программы. В процессе решения прикладных задач выбор подходящего алгоритма вызывает определенные трудности. Иногда необходимо, чтобы алгоритм отвечал противоречащим друг другу требованиям (минимальное время выполнения, минимальная память).

На время выполнения программы влияют следующие факторы: ввод (порядок ввода) исходной информации в программу; качество скомпилированного кода исполняемой программы; машинные инструкции (естественные и ускоряющие), используемые для выполнения программы; временная сложность алгоритма.

Поскольку время выполнения программы зависит от ввода исходных данных, его можно определить как функцию от исходных данных. Обычно говорят, что время выполнения программы имеет порядок $T(n)$ от входных данных размера n. Например, некая программа имеет время выполнения $T(n) = cn^2$, где c – константа. Для многих программ время выполнения действительно является функцией входных данных, а не их размера. В этой ситуации $T(n)$ определяется как время выполнения в наихудшем случае, т.е. как максимум времени выполнения по всем входным данным размера n. Величина $T_{ср}(n))$ будет рассматриваться как среднее (в статистическом смысле) время выполнения по всем входным данным размера n. Хотя $T_{ср}(n)$ является достаточно объективной мерой времени выполнения, однако часто нельзя предполагать (или обосновать) равнозначность всех входных данных. На практике среднее время выполнения найти сложнее, чем наихудшее время выполнения, так как математически это трудноразрешимая задача и, кроме того, зачастую не имеет простого определения понятие «средних» входных данных. Поэтому в основном используется наихудшее время выполнения как мера временной сложности алгоритмов.

Второй и третий фактор (компилятор и машина, на которой выполняется программа) влияют на то, что для измерения времени выполнения $T(n)$ невозможно применить стандартные единицы измерения,

такие как секунды или миллисекунды. Поэтому можно только делать заключения вида «время выполнения такого-то алгоритма пропорционально n^2». Константы пропорциональности также нельзя точно определить, поскольку они зависят от компилятора, компьютера и других факторов.

15.2. Асимптотические соотношения

Для описания скорости роста функций используется *O*-нотация. *O*-нотация используется по трем основным причинам: чтобы ограничить ошибку, возникающую при отбрасывании малых слагаемых в математических формулах; чтобы ограничить ошибку, возникающую тогда, когда не учитываются те части программы, которые дают малый вклад в анализируемую сумму; чтобы классифицировать алгоритмы согласно верхней границе их общего времени выполнения.

Например, когда говорят, что время выполнения $T(n)$ некоторой программы имеет порядок $O(n^2)$ (читается «*о*-большое от n в квадрате» или «*O* от n в квадрате»), то подразумевается, что существуют положительные константы c и n_0 такие, что для всех n, больших или равных n_0, выполняется неравенство $T(n) \le cn^2$.

O-нотация используется, прежде всего, для исследования фундаментального асимптотического поведения алгоритма и представляет собой верхнюю асимптотическую оценку *трудоемкости алгоритма*. Она позволяет определить, как быстро растет трудоемкость алгоритма с увеличением объема данных.

Определение. Функция $T(n)$ имеет порядок $O(f(n))$, если существуют константы c и n_0 такие, что для всех $n \ge n_0$ выполняется неравенство $T(n) \le cf(n)$. Для программ, у которых время выполнения имеет порядок $O(f(n))$, говорят, что они имеют порядок (или степень) роста $f(n)$.

Например, функция $T(n) = 3n^3 + 2n^2$ имеет степень роста $O(n^3)$ или время исполнения задачи растет не быстрее, чем куб количества элементов. Чтобы это показать, надо положить $n_0 = 0$ и $c = 5$, тогда, для всех целых $n \ge 0$ выполняется неравенство $3n^3 + 2n^2 \le 5n^3$.

При оценке трудоемкости используют *правило сумм*: в общем случае трудоемкость выполнения конечной последовательности программных фрагментов, без учета констант, имеет порядок фрагмента с наибольшим временем выполнения.

Например, пусть есть три фрагмента с временами выполнения соответственно $O(n^2)$, $O(n^3)$ и $O(n\log n)$. Тогда время последовательного выполнения первых двух фрагментов имеет порядок $O(\max(n^2, n^3))$, т.е. $O(n^3)$. Время выполнения всех трех фрагментов имеет порядок $O(\max(n^3, n\log n))$, это то же самое, что $O(n^3)$.

При оценке трудоемкости используют *правило произведений*: если $T1(n)$ и $T2(n)$ имеют степени роста $O(f(n))$ и $O(g(n))$ соответственно, то произведение $T1(n)T2(n)$ имеет степень роста $O(f(n)g(n))$. Из правила произведений следует, что $O(cf(n))$ эквивалентно $O(f(n))$, если c – положительная константа. Например, $O(n^2 / 2)$ эквивалентно $O(n^2)$.

В табл. 15.1 приведены числа, иллюстрирующие скорость роста для нескольких разных функций. Используемый параметр n может быть степенью полинома, размером файла при сортировке или поиске, количеством символов в строке или некоторой другой абстрактной мерой размера рассматриваемой задачи. Однако, чаще всего, он прямо пропорционален величине обрабатываемого набора данных.

Таблица 15.1

Скорость роста функций

n	$\lg n$	$\sqrt{n}$	$n\lg n$	$n(\lg n)^2$	$n^{3/2}$	n^2
10	3	3	33	110	32	100
100	7	10	664	4414	1000	10 000
1000	10	32	9966	99 317	31 623	1 000 000
10 000	13	100	132 877	1 765 633	1 000 000	100 000 000
100 000	17	316	1 660 964	27 588 016	31 622 777	10 000 000 000
1 000 000	20	1000	19 931 569	397 267 426	1 000 000 000	1 000 000 000 000

Как видно из табл. 15.1 для небольших задач используемый метод практически не влияет на скорость решения задачи. Но по мере роста задачи числа, с которыми мы имеем дело, становятся огромными. Когда количество исполняемых инструкций в медленном алгоритме становится по-настоящему большим, время, необходимое для их выполнения, становится недостижимым (например, 10^7 секунд равно 3,8 месяца, а 10^8 секунд равно 3,1 года).

Для *нижней асимптотической оценки* роста функции $T(n)$ используется Ω. Она задает и определяет класс функций, которые растут не медленнее, чем $f(n)$ с точностью до постоянного множителя.

Определение. Функция $T(n)$ имеет порядок $\Omega(f(n))$, если существуют константы c и n_0 такие, что для всех $n \geq n_0$ выполняется неравенство $T(n) \leq cf(n)$.

Например, запись $T(n) = \Omega(n\log n)$ обозначает класс функций, которые растут не медленнее, чем $f(n)$ $n\log n$, в этот класс попадают все полиномы со степенью большей единицы, равно как и все степенные функции с основанием, большим единицы.

Для задания одновременно верхней и нижней оценки роста функции $T(n)$ используется Θ.

Определение. Оценка сложности алгоритма $T(n) = \Theta(f(n))$, если при $g > 0$ и $n > 0$ существуют положительные c_1, c_2, n_0, такие, что $c_1 f(n) \le T(n) \le c_2 f(n)$ при $n > n_0$.

В этом случае говорят еще, что функция $f(n)$ является асимптотически точной оценкой функции $T(n)$, так как по определению функция $T(n)$ не отличается от функции $f(n)$ с точностью до постоянного множителя. Важно понимать, что $\Theta(f(n))$ представляет собой не функцию, а множество функций, описывающих рост $T(n)$ с точностью до постоянного множителя. Равенство $T(n) = \Theta(f(n))$ выполняется тогда и только тогда, когда $T(n) = O(f(n))$ и $T(n) = \Omega(f(n))$.

Алгоритмы, которые будут рассмтариваться далее, обычно имеют сложность, пропорциональную одной из следующих функций из табл. 15.2.

Таблица 15.2

Функции сложности

$T(n)$	Описание
1	Большинство инструкций большинства программ запускается один или несколько раз. Тогда, говорят, что время выполнения программы постоянно
$\log n$	Программа становится медленнее с ростом n. Такое время выполнения обычно присуще программам, которые сводят большую задачу к набору меньших задач, уменьшая на каждом шаге размер задачи на некоторый постоянный фактор. Основание логарифма не сильно изменяет константу. При удвоении $\log n$ растет на постоянную величину, а удваивается лишь тогда, когда n достигает n^2
n	Когда время выполнения программы является линейным, это значит, что каждый входной элемент подвергается небольшой обработке. Эта ситуация оптимальна для алгоритма, который должен обработать n вводов (или произвести n выводов)
$n\log n$	Возникает тогда, когда алгоритм решает задачу, разбивая ее на меньшие подзадачи, решая их независимо и затем объединяя решения. Когда n удваивается, тогда время выполнения более чем удваивается
n^2	Когда время выполнения алгоритма является квадратичным, он полезен для практического использования для относительно небольших задач. Квадратичное время выполнения обычно появляется в алгоритмах, которые обрабатывают все пары элементов данных (возможно, в цикле двойного уровня вложенности)

Таким образом, асимптотические оценки можно представить в порядке увеличения скорости их роста:

$O(1) < O(\log_2 n) < O(n) < O(n\log_2 n) < O(n^2) < O(n^3) < O(2^n)$.

15.3. Классы сложности

В рамках классической теории осуществляется классификация задач по классам сложности (P-сложные; NP-сложные, экспоненциально сложные и др.). К *классу P* относятся задачи, которые могут быть решены за время, полиномиально зависящее от объема исходных данных, с помощью детерминированной вычислительной машины (например, машины Тьюринга). К *классу NP* относятся задачи, которые могут быть решены за полиномиально выраженное время с помощью недетерминированной вычислительной машины, то есть машины, следующее состояние которой не всегда однозначно определяется предыдущими. В частности, к классу NP относятся все задачи, решение которых можно проверить за полиномиальное время. Класс P содержится в классе NP.

Поскольку класс P содержится в классе NP, принадлежность той или иной задачи к классу NP зачастую отражает наше текущее представление о способах решения данной задачи и носит неокончательный характер. В общем случае нет оснований полагать, что для той или иной NP-задачи не может быть найдено P-решение. Вопрос о возможной эквивалентности классов P и NP (то есть о возможности нахождения P-решения для любой NP-задачи) считается многими одним из основных вопросов современной теории сложности алгоритмов.

Все классы сложности находятся в иерархическом отношении: одни включают в себя другие. Однако про большинство включений неизвестно, являются ли они строгими.

15.4. Эффективность алгоритмов поиска

В качестве еще одного примера рассмотрим два алгоритма поиска: последовательный и бинарный поиск элемента в массиве.

Последовательный поиск

При последовательном поиске элемента в массиве, имеющем длину n, элементы просматриваются по очереди, начиная с первого, пока не обнаружится искомый либо не будет достигнут конец массива. В наилучшем случае искомым элементом является первый. Для его обнаружения понадобится только одно сравнение. Следовательно, в наилучшем случае сложность алгоритма последовательного поиска равна $O(1)$. В наихудшем случае искомый элемент является последним. Для того, чтобы его найти, понадобится n сравнений. Следовательно, в наихудшем случае сложность алгоритма последовательного

поиска равна $O(n)$. В среднем случае искомый элемент находится в средней ячейке массива и обнаруживается после $n / 2$ сравнений.

Бинарный поиск

Алгоритм бинарного поиска предназначен для поиска элемента в упорядоченном массиве и основан на повторяющемся делении частей массива пополам. Алгоритм определяет, в какой из двух частей находится элемент, если он действительно хранится в массиве, а затем повторяет процедуру деления пополам. В ходе очередного разбиения массива алгоритм выполняет сравнения. Можно вычислить максимальное количество сравнений, т.е. наихудший вариант. Допустим, что $n = 2^k$, где k – некоторое натуральное число.

Чтобы проверить среднюю ячейку массива, сначала нужно поделить массив пополам. После того, как массив, состоящий из n элементов, поделен пополам, делится пополам одна из его половин. Эти деления продолжаются до тех пор, пока не останется только один элемент. Для этого потребуется выполнить k разбиений массива. Это возможно, поскольку $n / 2^k = 1$. В наихудшем случае алгоритм выполнит k разбиений и, следовательно, k сравнений. Поскольку $n = 2^k$, получаем, что $k = \log_2 n$. Если число n не будет степенью двойки, то k – наименьшее число, удовлетворяющее условию $2^{k-1} < n < 2^k$. (Например, если n равно 30, то $k = 5$, поскольку $2^4 = 16 < 30 < 32 < 2^5$).

Таким образом, сложность алгоритма бинарного поиска в наихудшем случае имеет порядок $O(\log_2 n)$ для любого значения n.

Бинарный поиск намного лучше последовательного (при $n = 1\,000\,000$, $\log_2 n = 19$, т.е. при последовательном поиске в наихудшем случае будет миллион сравнений, в то время как алгоритм бинарного поиска выполнит не более 20) при условии упорядоченности массива.

15.5. Алгоритмы сортировки

Сортировка – процесс упорядочения набора элементов в возрастающем или убывающем порядке.

Алгоритмы сортировки можно классифицировать по нескольким признакам. По размещению элементов сортировки бывают внутренними – в памяти и внешними – в файле. По виду структуры данных, содержащей сортируемые элементы, можно выделить сортировки массивов (строк), массивов указателей, списков, объектов и других структур данных. Если в объектах содержатся несколько данных-членов, нужно указать, какая переменная определяет порядок следования объектов. Эта переменная-член называется ключом сортировки (key). Например,

если объекты хранят информацию о людях, их можно сортировать по имени, возрасту или почтовому индексу. По сохранению относительного порядка размещения элементов с дублированными ключами сортировки бывают устойчивые и неустойчивые. Сортировки делятся на адаптивные (выполнение различных последовательностей операций в зависимости от результата сравнения) и неадаптивные (последовательность операций не зависит от порядка следования данных). По способу выбора элементов сортировки делятся на сортировки подсчетом, вставками, выбором, слиянием, разделением, обменные сортировки.

Для простоты будем предполагать, что сортировка применяется к числам. Все рассматриваемые алгоритмы ориентируются на возрастающий порядок.

15.6. Обменные сортировки

Метод пузырька

Алгоритм сортировки методом пузырька сравнивает между собой соседние элементы и меняет их местами, если они нарушают порядок. Для этого приходится несколько раз просматривать одни и те же элементы. Во время первого прохода сравниваются два первых элемента массива; если они нарушают порядок, их меняют местами. Затем сравнивается другая пара, т.е. 2-й и 3-й элементы. Если они нарушают порядок, их меняют местами. Просмотр, сравнение и обмен двух элементов выполняется до тех пор, пока не будет достигнут конец массива. На рис. 15.1 представлен пример работы этого метода.

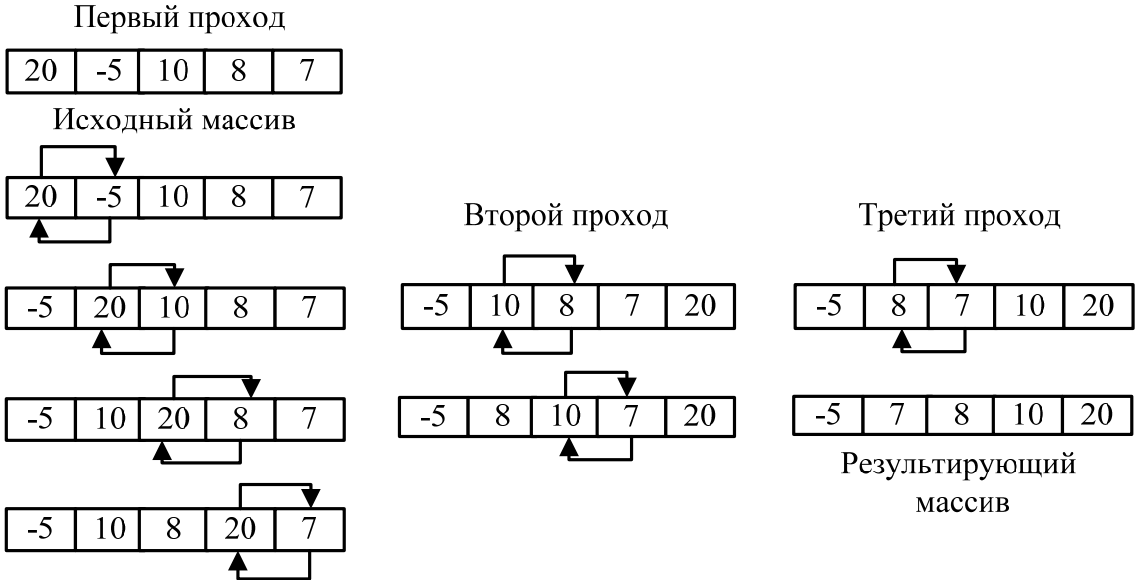


Рис. 15.1. Сортировка методом пузырька

Рассмотрим функцию, выполняющую сортировку массива методом пузырька:

```
void BubbleSort(int arr[],int n)
{    int temp;
     for(int i = n-1; i > 0; --i)
                          //для каждого элемента массива
          for(int j= 0; j < i; ++j)
               if(arr[j] > arr[j+1])  //процесс всплытия
               {
                    temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
               }
}
```

Анализ

При первом проходе выполняется не больше, чем (n – 1) сравнений и (n – 1) перестановок. При втором проходе выполняется (n – 2) сравнений и не больше (n – 2) перестановок. Следовательно, в худшем случае при сортировке методом пузырька будет выполнено (n – 1) + (n – 2) + + … + 1 = n (n – 1) / 2 ≈ n^2 / 2 сравнений и столько же перестановок. При каждой перестановке выполняется три присваивания. Таким образом, общее количество основных операций в худшем случае равно $2n(n-1) = 2n^2 - 2n$. В худшем случае сложность алгоритма сортировки методом пузырька равна $O(n^2)$. Пузырьковая сортировка не требует дополнительной памяти. Данная реализация алгоритма – устойчивая.

Шейкер-сортировка

Сортировку методом пузырька можно в некоторой степени улучшить и тем самым улучшить ее временные характеристики. Можно, например, заметить, что сортировка обладает одной особенностью: расположенный не на своем месте в конце массива элемент достигает своего места за один проход, а элемент, расположенный в начале массива, очень медленно достигает своего места. Необязательно все просмотры делать в одном направлении. Вместо этого всякий последующий просмотр можно делать в противоположном направлении и фиксировать нижнюю и верхнюю границы неупорядоченной части, т.к. просмотр имеет смысл делать не до конца массива, а до последней перестановки на предыдущем просмотре. В этом случае сильно удаленные от своего места элементы будут быстро перемещаться в соответствующее место. Ниже показана улучшенная версия сортировки пузырьковым методом,

получившая название «челночной сортировки» из-за соответствующего характера движений по массиву или шейкер-сортировки:

```
void ShakerSort(int arr[], int n)
{
  int buf, first=0, mode=1, last = n;
    for (; first<last; mode>0?++first:--last)
      {    for(int i= mode>0?first:last;
               mode>0?(i<last):(i>first); mode>0?++i:--i)
               {
               if ((arr [i]> arr [i+1])&&(mode>0))
                  {
                    buf = arr [i];
                    arr [i]= arr [i+1];
                    arr [i+1]= buf;
                  }
               if ((m[i]<m[i-1])&&(mode<0))
                  {
                    buf = arr [i];
                    arr [i]= arr [i-1];
                    arr [i-1]= buf;
                  }
               }
               mode=-mode;
      }
}
```

Хотя эта сортировка является улучшением метода пузырька, ее нельзя рекомендовать для использования, поскольку время выполнения по-прежнему зависит квадратично от числа элементов. Число сравнений не изменяется, а число обменов уменьшается лишь на незначительную величину.

15.7. Сортировка выбором

Прямая выборка

Работает по принципу выбора наименьшего элемента из числа неотсортированных. Отыскивается наименьший элемент массива, затем он меняется местами с элементом, стоящим первым в сортируемом массиве. Далее, находится второй наименьший элемент и меняется местами с элементом, стоящим вторым в исходном массиве. Этот процесс продолжается до тех пор, пока весь массив не будет отсортирован. На рис. 15.2 представлен пример работы этого метода.

Недостаток сортировки выбором заключается в том, что время ее выполнения лишь в малой степени зависит от того, насколько упорядочен исходный массив. Процесс нахождения минимального элемента за один проход дает очень мало сведений о том, где может находиться минимальный элемент на следующем проходе.

Рис. 15.2. Сортировка выбором

Рассмотрим функцию, выполняющую сортировку массива методом выбора:

```
void SelectSort(int arr[],int n)
{
  int k,min;
  //индекс минимального элемента, минимальный элемент
  for(int i = 0;i < n;i++)
  {
    k = i;  //индекс min
    min = arr[i];
    for(int j = i+1; j < n; j++) //цикл поиска минимального
      if(arr[j] < min) //в неотсортированной части
      {
        k = j;         //запомнить индекс
        min = arr[j];  //запомнить минимальный
      };
    arr[k] = arr[i];   //поменять местами
    arr[i] = min;
  }
}
```

Анализ

Как следует из описания алгоритма, сортировка сводится к сравнениям, присваиваниям и перестановкам элементов. Общее количество сравнений будет $(n-1) + (n-2) + \ldots + 1 = n(n-1)/2 \approx n^2/2$. Максимальное число перестановок $(n-1)$ и $4n + n(n-1)$ операций присваивания. В сумме алгоритм сортировки методом выбора выполняет $n(n-1)/2 + n - 1 + 4n + n(n-1) = n^2 + 9n/2 - 1$ основных операций. Применяя свойства O-нотации, можем отбросить слагаемые с младшими степенями и константные множители – получаем окончательную оценку $O(n^2)$. Хотя алгоритм выполняет $O(n^2)$ сравнений, в ходе сортировки осуществляется только $O(n)$ перестановок. Алгоритм сортировки методом выбора можно применять, когда перестановки представляют собой затратные операции, а сравнения – нет.

Квадратичная выборка

Данный метод по сравнению с прямой выборкой уменьшает число сравнений, но требует дополнительного объема памяти. Сортируемый массив, состоящий из n элементов, разделяется на n групп по $\sqrt{n}$ элементов в каждой. Если n не является точным квадратом, то таблица разделяется на n' групп, где n' – ближайший точный квадрат, больший n. В каждой группе выбирается наименьший элемент, который пересылается во вспомогательный массив. Вспомогательный массив просматривается, и наименьший его элемент пересылается в зону вывода (в зоне вывода формируется отсортированный массив). Далее из группы, содержащей элемент, посылаемый в зону вывода, выбирается новый наименьший элемент, который помещается во вспомогательный массив. Затем другой просмотр вспомогательного массива выбирает новый наименьший элемент, который является вторым по величине во всем массиве. Он пересылается в зону вывода. Элементы групп, которые уже посланы во вспомогательный массив, заменяются большими фиктивными величинами.

Таким образом, при сортировке квадратичной выборкой попеременно просматриваются то вспомогательный список, то группа до тех пор, пока все группы не будут исчерпаны. Такое состояние наступает, когда все группы посылают во вспомогательный массив фиктивные величины. Модификацией данного метода является квадратичная выборка с предварительной сортировкой. В этом методе группы сначала полностью упорядочиваются, а затем уже выполняются сравнения между группами. Количество действий, требуемое для сортировки квадратичной выборкой, несколько меньше, чем в предыдущих

методах и равно n^2, но требуется дополнительная память. Общее число сравнений для случая точного квадрата равно приблизительно $2n\sqrt{n}$, необходимый резерв памяти – поле длиной $(n+\sqrt{n})$ элемент.

15.8. Сортировка вставками

Простая вставка

Весь массив делится на две части: упорядоченную и неупорядоченную. Вначале весь массив неупорядочен. На каждом шаге метода вставок из неупорядоченной части извлекается первый элемент, который затем вставляется в нужное место упорядоченной части.

Первый шаг: переместить нулевой элемент из неупорядоченной части в упорядоченную. Тот факт, что элементы в упорядоченной части расположены в порядке возрастания, является инвариантом алгоритма. Поскольку на каждом шаге размер упорядоченной части увеличивается на единицу, а размер неупорядоченной части, соответственно, на единицу уменьшается, в момент окончания алгоритма весь массив окажется упорядоченным. На рис. 15.3 представлен пример работы этого метода.

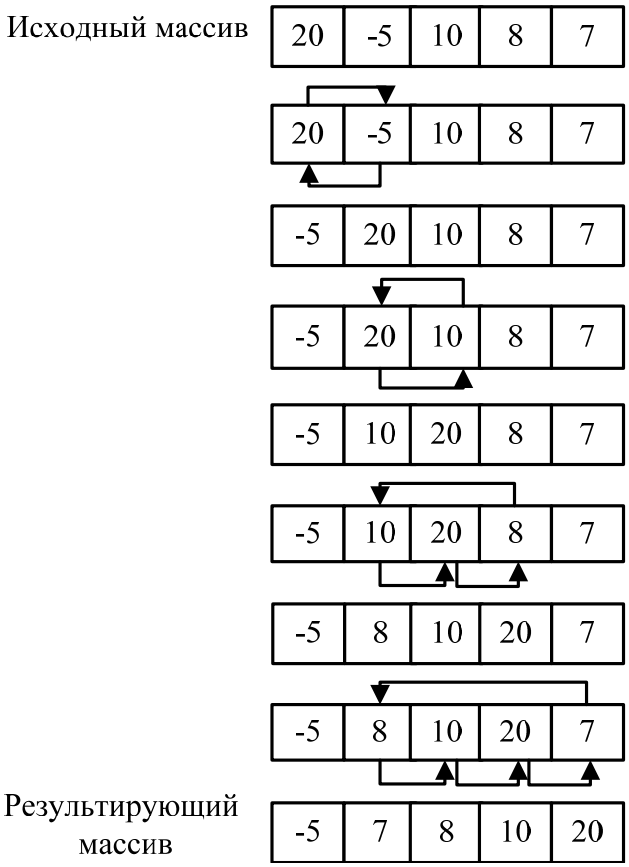


Рис. 15.3. Сортировка вставкой

Рассмотрим функцию, выполняющую сортировку массива методом вставок:

```
void InsertSort(int arr[],int n)
{
     int temp,i,j;
     for(i = 1;i < n;i++) //для очередного элемента
     {
       temp = arr[i];    //сохранить
       for(j = i-1;j >= 0 && arr[j] > temp; j--)
               //поиск места вставки
                arr[j+1] = arr[j];
               //сдвиг элементов массива
          arr[j+1] = temp; //вставка очередного
                          //на место первого большего его
     }
}
```

Анализ

Внешний цикл в функции выполняется $n - 1$ раз. Этот цикл содержит внутренний цикл, который выполняется не больше чем $n - 1$. Таким образом, в худшем случае алгоритм выполняет $1 + 2 + ... + (n - 1) = n(n - 1) / 2$ сравнений. Кроме того, в худшем случае столько же раз внутренний цикл сдвигает элементы. Во внешнем цикле перемещение элементов на каждой итерации выполняется дважды, т.е. в сумме $2(n - 1)$ раз. Итак, в наихудшем варианте выполняется $n(n - 1) + 2(n - 1) = n^2 + n - 2$ основных операций. В среднем приблизительно $n^2 / 4$ операций сравнения и $n^2 / 4$ операций полуобмена элементов местами (перемещений). Следовательно, сложность алгоритма сортировки методом вставок равна $O(n^2)$.

Метод хорош устойчивостью сортировки, удобством для реализации на списках и, самое главное, естественностью поведения. То есть уже частично отсортированный массив будут досортирован им гораздо быстрее, чем многими более эффективными методами. Не требует дополнительной памяти. В отличие от сортировки выбором, время выполнения сортировки вставками зависит главным образом от исходного порядка ключей при вводе. Для больших массивов этот метод неэффективен.

Метод Шелла

Метод Шелла (сортировка с убывающим шагом) существенно превосходит метод простых вставок. Элементы перемещаются большими

скачками. Упорядочиваемый массив разделяется на группы элементов, каждая из которых упорядочивается методом вставки. В процессе упорядочения размеры таких групп увеличиваются до тех пор, пока все элементы таблицы не войдут в упорядоченную группу. Группой называют последовательность элементов, номера которых образуют арифметическую прогрессию с разностью h (h называют шагом группы). В начале процесса упорядочения выбирается первый шаг группы h_1, который зависит от размера массива. Шелл предложил брать $h_1 = [n / 2]$, а $h_i = h_{(i-1)} / 2$. В более поздних работах Хиббард показал, что для ускорения процесса целесообразно определить шаг h_1 по формуле $h_1 = 2^k + 1$, где $2^k < n \le 2^{(k+1)}$.

После выбора h_1 методом вставки упорядочиваются группы, содержащие элементы с номерами позиций: i, $i + h_1$, $i + 2h_1$, ..., $i + m_i h_1$, при этом $i = 1, 2, ..., h_1$; m_i – наибольший целый индекс группы. Затем выбирается шаг h_2 и упорядочиваются группы, содержащие элементы с номерами позиций i, $i + h_2$, ..., $i + m_i h_2$. Эта процедура, со все уменьшающимися шагами, продолжается до тех пор, пока очередной шаг станет равным единице. Этот последний этап представляет собой упорядочение всего массива методом вставки. Но так как исходная таблица упорядочивалась отдельными группами с последовательным объединением этих групп, то общее количество сравнений значительно меньше, чем требуется при методе вставки. Число операций сравнения пропорционально $n(\log_2 n)^2$. Ниже приведен пример функции метода Шелла:

```
void ShellSort(int arr[],int n)
{
   for(int step = n/2;step/2 != 0;step/ = 2)
                                //цикл выбора шага
   {
      for(int i = 0; i < (n-step); ++i)
      {
         int j = i ; //запомнить индекс текущего
                     //сравнить все элементы группы
         while((j >= 0) && (arr[j] > arr[j+step]))
         {
         int temp = arr[j]; //упорядочить методом вставок
         arr[j] = arr[j+step];
         arr[j+step] = temp;
         --j;
         }
      }
   }
}
```

15.9. Сортировка разделением

Алгоритм сортировки разделением основан на разбиении последовательности на две подпоследовательности по некоторому правилу. При этом каждая из них может не являться упорядоченной, но применение указанного правила приводит к упорядоченности последовательности. Примером этого класса алгоритмов является алгоритм быстрой сортировки.

Быстрая сортировка

Быстрая сортировка была разработана Ч. Хоаром в 1962 г. и представляет собой рекурсивный алгоритм, основанный на принципе декомпозиции. Сортировки, основанные на принципе разделения массива, разделяют его на две части относительно некоторого значения, называемого медианой. Сами части не упорядочены, но обладают таким свойством, что элементы в левой части меньше медианы, а элементы правой – больше. Благодаря такому свойству эти части можно сортировать независимо. Для этого нужно вызвать ту же самую функцию сортировки, но уже не по отношению к массиву, а к его частям. Рекурсивный вызов продолжается до тех пор, пока очередная часть массива не станет содержать единственный элемент. На рис. 15.4 представлен пример работы этого метода.

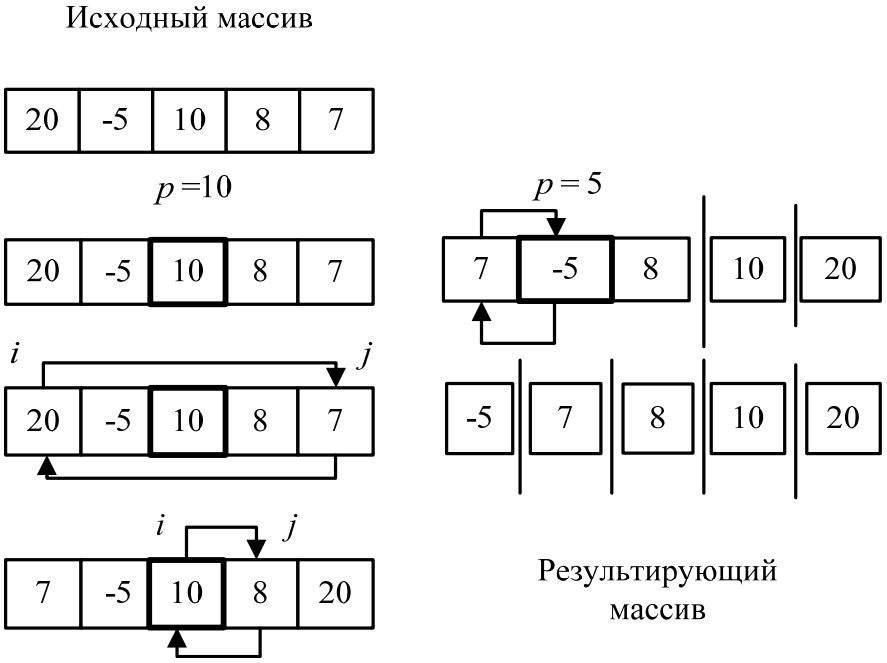


Рис. 15.4. Быстрая сортировка

Ниже приведен пример реализации быстрой сортировки:

```
void QuickSort(int arr[],int n)
{
     int i = 0,j = n;
     int temp,p;
     p = arr[n>>1]; //выбрать медиану - средний элемент
     //p = arr[i];  //медиана -  первый элемент
     //p = arr[i+rand()%n-i];
                    //медиана - случайный элемент
     do
     {              //процедура разделения
     while(arr[i] < p) i++;
     while(arr[j] > p) j--;
     if(i <= j)     //обмен
        {
         temp = arr[i];
             arr[i] = arr[j];
             arr[j] = temp;
             i++;
             j--;
        }
     }
     while(i <= j);
         //если есть что сортировать
     if( j > 0 ) QuickSort(arr,j);
         //рекурсивный вызов для левой части
     if( n > i ) QuickSort(arr+i,n-i);
         //рекурсивный вызов для правой части
}
```

На практике для увеличения быстроты (но не симптотики) можно произвести несколько улучшений. Во-первых, в качестве медианы выбирать элемент, расположенный в середине (как в приведенном примере). Такой выбор улучшает оценку среднего времени работы, если массив упорядочен лишь частично. Наихудшая для этой реализации ситуация возникает в случае, когда каждый раз в качестве медианы выбирается максимальный или минимальный элемент. Можно также выбрать средний из первого и последнего элементов. Тогда количество проходов уменьшится в 7/6 раз. Во-вторых, для коротких массивов можно вызывать сортировку вставками. Из-за рекурсии и других «накладных расходов» быстрый поиск оказывается не столь уж быстрым для коротких массивов. Поэтому, если в массиве меньше 12 элементов, вызывается сортировка вставками. В-третьих, если последний оператор функции является вызовом этой функции, говорят о хвостовой рекурсии. Ее имеет смысл заменять на итерации – в этом случае лучше

используется стек. В-четвертых, после разбиения сначала необходимо сортировать меньший подмассив. Это также приводит к лучшему использованию стека, поскольку короткие разделы сортируются быстрее и им нужен более короткий стек. Требования к памяти уменьшаются с n до $\log n$. Реализация быстрой сортировки, входящая в стандартную библиотеку C, использует многие из этих улучшений.

Анализ

Ключевым моментом алгоритма является выбор медианы, относительно которой происходит разбиение массива на две части. Идеален такой выбор, при котором массив разбивается на два равных подмассива. Сам Хоар предлагал выбирать медиану случайным образом.

Время работы методом быстрой сортировки зависит от упорядоченности массива. Оно будет минимальным, если на каждом шаге разбиения получаются подмассивы приблизительно равной длины, и тогда требуется около $C_n = 2C_{n/2} + n$ шагов, $2C_{n/2}$ – соответствует затратам на сортировку двух подмассивов, n – затраты на проверку каждого элемента.

Рекуррентное соотношение имеет решение $C_n \approx n\log n - O(n\log n)$. В наихудшем случае требуется около $n^2 / 2$ шагов – $O(n^2)$.

Сортировка не требует дополнительной памяти. Алгоритм неустойчив. Поведение неестественное, практически отсортированный массив будет сортироваться столько же, сколько и полностью разупорядоченный.

Алгоритм быстрой сортировки часто используется для сортировки больших массивов. Причина этого заключается в исключительном быстродействии, несмотря на оценку наихудшего случая. Этот вариант встречается крайне редко, и на практике алгоритм отлично работает. Значительное различие между оценками сложности в среднем и наихудшем вариантах выделяет алгоритм быстрой сортировки среди остальных рассмотренных алгоритмов. Если порядок записи элементов в исходном массиве является случайным, алгоритм работает, по крайней мере, не хуже любого другого алгоритма, использующего сравнения элементов. Если исходный массив совершенно неупорядочен, алгоритм быстрой сортировки работает лучше всех.

15.10. Сортировка подсчетом

Упорядоченный массив получается из исходного путем сравнения всех пар элементов массива, для каждого из них подсчитывается количество элементов, меньших его. Это дает новое местоположение

этого элемента в выходном массиве. Если допускается ситуация, когда несколько элементов имеют одно и то же значение, то данную схему придется модифицировать, поскольку разместить подобные элементы в одной и той же позиции нельзя. Для сортировки массива требуются два дополнительных массива для размещения результирующей последовательности и временного массива для счетчиков. На рис. 15.5 представлен пример работы этого метода.

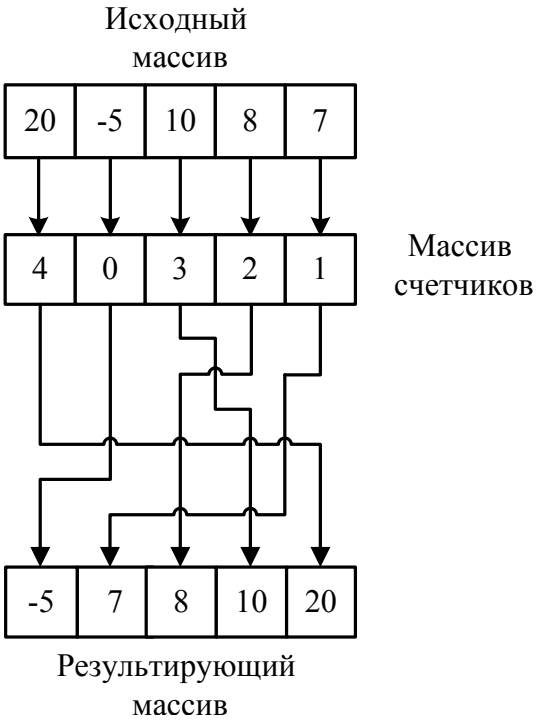


Рис. 15.5. Сортировка подсчетом

Ниже приведен пример реализации сортировки подсчетом:

```
void CountSort(int arr_in[],int arr_out[],int n)
//неполный подсчет
   {
    int i,j ,cnt;
        for (i = 0; i < n; i++)
       {
        for (cnt = 0,j = 0; j < n; j++)
             if (arr_in[j] < arr_in[i])
             cnt++; //счетчик элементов, больших текущего
        arr_out[cnt] = arr_in[i];
        //определяется его место в выходном массиве
       }
   }
```

Время работы алгоритма $\Theta(n)$ или $\Theta(n + k)$, где k – размер массива счетчиков. Таково же и количество дополнительной памяти, необходимой для работы алгоритма. Алгоритм устойчив. Эффективность алгоритма подсчета больше, чем у любого из ранее рассмотренных, поскольку в нем не сравниваются элементы последовательности – вместо этого непосредственно используются их значения. Обобщенная реализация сортировки на C++ не имеет смысла в силу специфичности и привязки к конкретному типу.

15.11. Пирамидальная сортировка

Еще один алгоритм сортировки, обеспечивающий теоретическую временную эффективность – пирамидальная сортировка. Назовем пирамидой последовательность элементов h_l, h_2, ..., h_r такую, что $h_i \le h_{2i}$ и $h_i \le h_{2i+1}$ для всякого $i = l$, ..., r / 2. Геометрическая интерпретация пирамиды приведена на рис. 15.6.

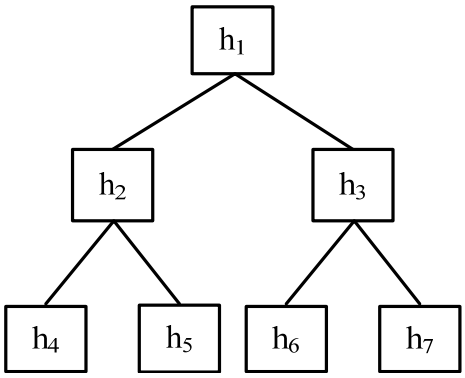


Рис. 15.6. Пирамидальная сортировка

Алгоритм состоит из двух фаз.

Фаза 1: построение пирамиды

На каждом шаге добавляется новый элемент и «просеивается» на свое место. При добавлении элемента в готовую пирамиду: 1 – новый элемент помещаем в вершину дерева; 2 – смотрим на элемент слева и элемент справа и выбираем наименьший; 3 – если элемент меньше нового – меняем их местами и идем к шагу 2, иначе – конец. На рис. 15.7 представлен пример построения пирамиды по массиву.

Специальная структура пирамид позволяет компактно размещать их в памяти. В частности, пирамиду с n вершинами можно разместить в массиве, в котором две непосредственно следующие за i вершины помещаются в $2i$ и $(2i + 1)$ элементы.

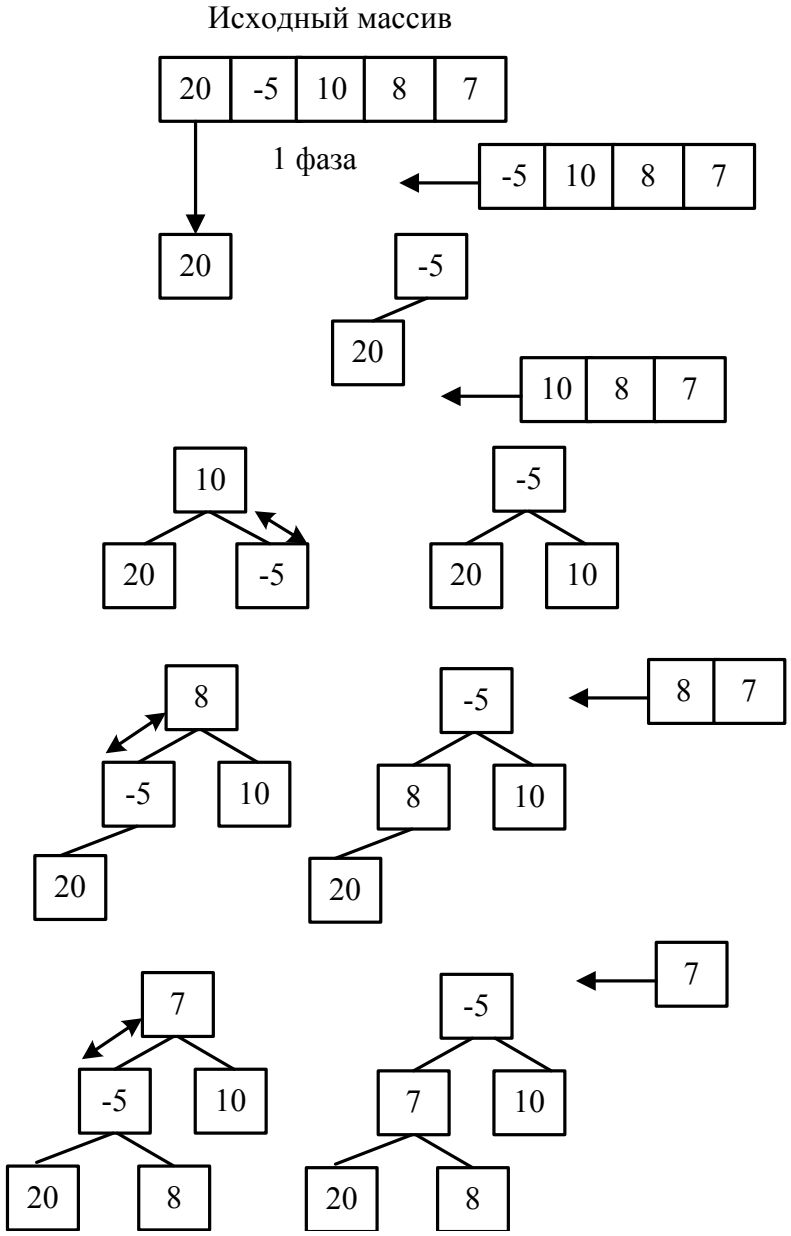
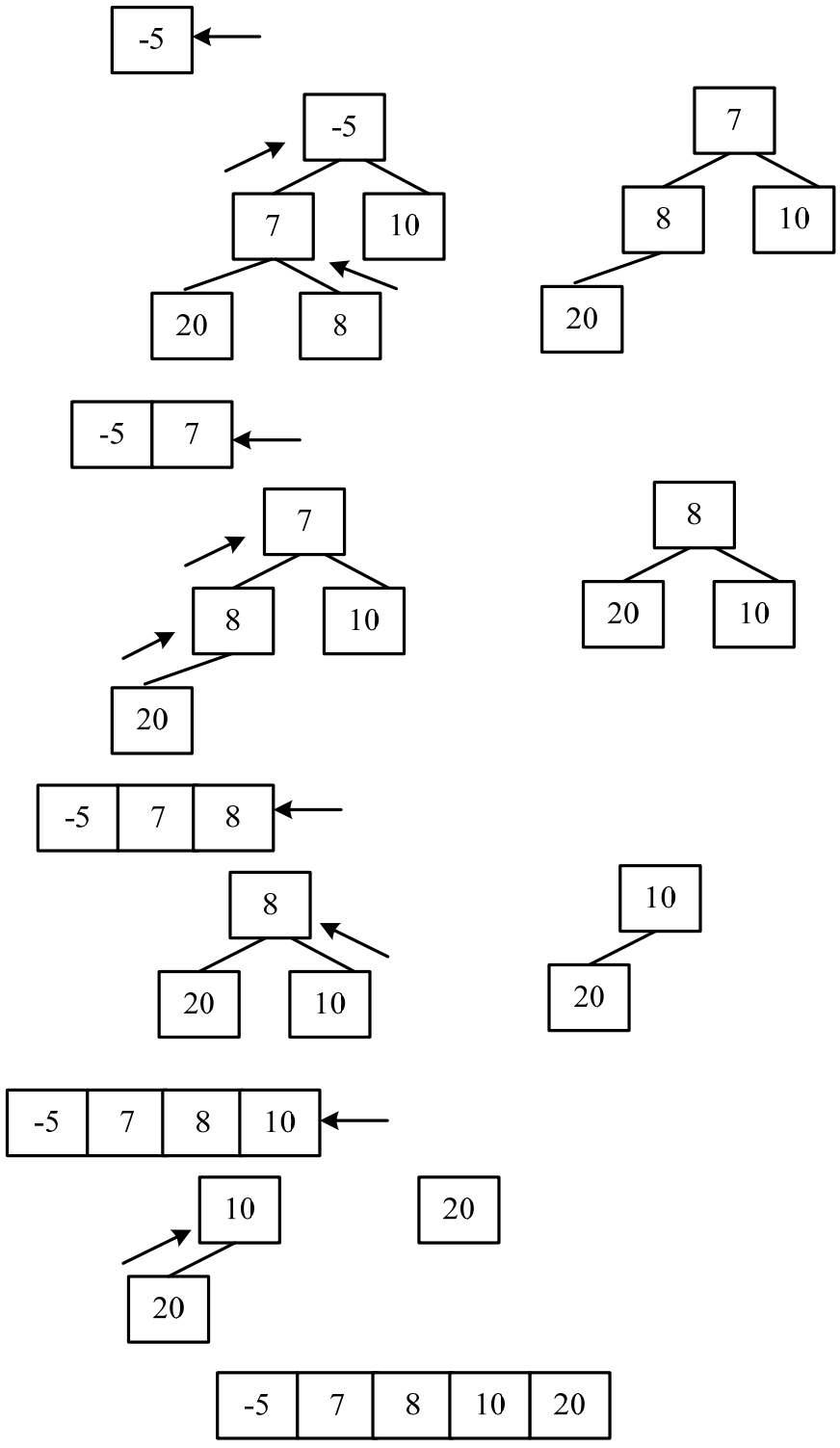


Рис. 15.7. Фаза 1: построение пирамиды

Фаза 2: сортировка

Для того чтобы отсортировать элементы, необходимо выполнить *n* шагов просеивания. После каждого шага очередной элемент берется с вершины пирамиды. На каждом шаге выбирается минимальный из следующих за ним узлов пирамиды и помещается в вершину и т.д. по цепочке. Выбранный с вершины элемент просеивается на свое «законное» место. В этом случае необходимо выполнить (n – 1) шагов. На рис. 15.8. представлен пример построения отсортированного массива по пирамиде.



Результирующий
массив

Рис. 15.8. Фаза 2: сортировка

Рассмотрим реализацию пирамидальной сортировки:

```
//Поддержка основного свойства пирамиды (бинарной кучи)
void heapify(int arr[], long i, long n)
   {
     if(n != 0)
     {  int tmp = arr[i];
        long child;
        bool b = true;
        while (i <= n/2 && b)
        {   child = 2*i;
            //левый ребенок элемента k
            if(child < n && arr[child]>arr[child+1]))
            ++child;
            //выбираем наименьшего ребенка
               if(tmp<arr[child])) b = false;
               //если родитель больше всех своих детей
               else
               {   arr[i] = arr[child];
                   i = child;   }
               //иначе меняем arr[k] с наименьшим ребенком
               }
               arr[i] = tmp;
            }
        }
void PyramidSort(int arr[], long n, long b)
     {   int tmp;
         long i;
         for(i = b/2; i >= n; --i) heapify(arr, i, b-1);
         //строим пирамиду
         for(i = b; i > n; --i)
         {
         tmp = arr[n]; arr[n] = arr[i]; arr[i] = tmp;
         //обмен
         heapify(arr, n, i-1);
         }
     }
```

Анализ

Прекрасной характеристикой этого метода является то, что среднее число пересылок – ($n\log n$) / 2 и отклонения от этого значения сравнительно малы. Наихудшего случая нет, сложность всегда – $O(n\log n)$. Определенная сложность операций с пирамидой увеличивает постоянный множитель в $O(n\log n)$, так что в среднем пирамидальная сортировка проигрывает быстрой по эффективности. Часто применяется в смежных задачах. Поведение алгоритма неестественно. Пирамидальная сортировка не требует дополнительной памяти.

15.12. Сортировка слиянием

Сортировка слиянием применяет подход «разделяй и властвуй» и в среднем столь же эффективна, как и быстрая, однако в ней не возникает проблемы наихудшего случая. Данный алгоритм сортировки требует дополнительного массива того же размера для временного хранения данных.

Алгоритм сортировки является рекурсивным. Его эффективность не зависит от порядка следования элементов в исходном массиве. Исходный массив делится пополам, затем рекурсивно упорядочиваются обе половины, а затем объединяются в одно целое. В ходе слияния элементы, стоящие в разных частях массива, попарно сравниваются друг с другом, и меньший элемент отправляется во временный массив. Этот процесс продолжается до тех пор, пока не будет использована одна из двух частей массива. Теперь достаточно просто скопировать оставшиеся элементы во временный массив. В заключение содержимое временного массива копируется обратно в исходный массив. На рис. 15.9 представлен пример сортировки слиянием.

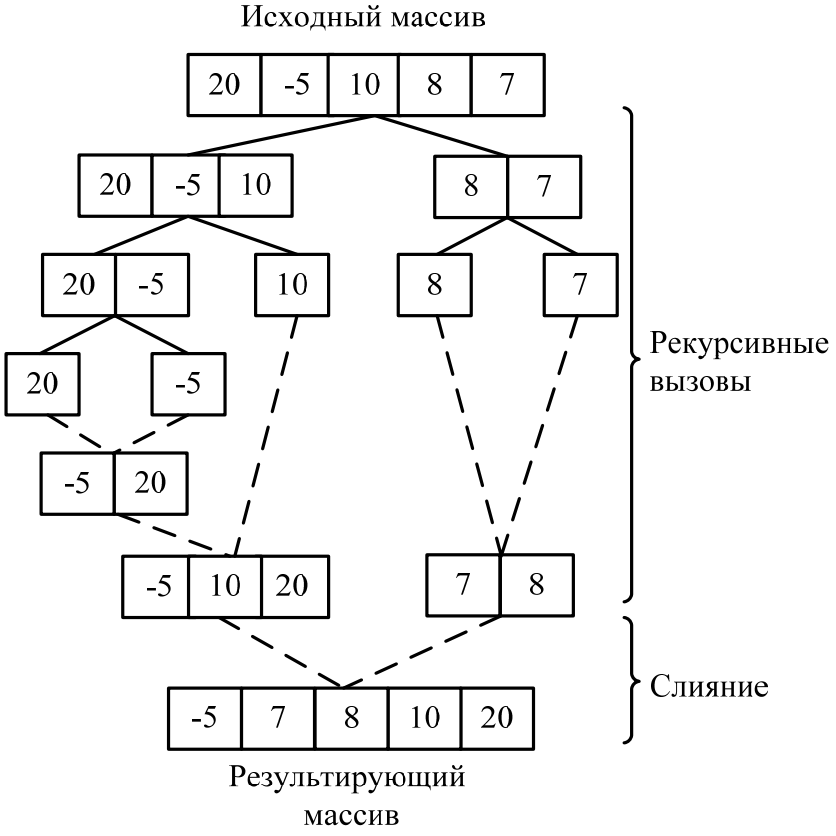


Рис. 15.9. Сортировка слиянием

Ниже приведена функция, реализующая алгоритм сортировки методом слияний.

```
//Слияние частей для сортировки слиянием
void fusion(int arr[], long a, long s, long b)
     {
     long p1 = a, p2 = s+1, p = 0;
//позиции чтения: p1 - из первой последовательности
//p2 - из второй последовательности,
//p - позиция записи в tmp
     int *tmp = new int[b-a+1];    // буфер
     while (p1 <= s && p2 <= b)
//слияние, пока есть хотя бы один
//элемент в каждой последовательности
     ( arr[p1]<arr[p2])) ? (tmp[p++] = arr[p1++]):
                           (tmp[p++] = arr[p2++]);
     //одна последовательность закончилась
     //копируем остаток второй в конец буфера
          while (p2 <= b) tmp[p++] = arr[p2++];
     //пока вторая последовательность непуста
          while (p1 <= s) tmp[p++] = arr[p1++];
     //пока первая последовательность непуста
          for(p = 0; p <= b - a; ++p)
          arr[a+p] = tmp[p];
     //скопировать буфер tmp в arr[a...b]
          delete [] tmp;
     }
//Сортировка слиянием
void MergeSort(int arr[], long a, long n)
     {
     if(a < n)
       {
       long s = (a + n)/2;
     //индекс разделения массива
       MergeSort(arr, a, s, f);
     //сортировать левую половину
       MergeSort(arr, s+1, n, f);
     //сортировать правую половину
       fusion(arr, a, s, n); //слияние
       }
     }
```

Анализ

Если общее количество элементов объединяемых отрезков массива равно n, то при их слиянии потребуется выполнить $(n - 1)$ сравнений. Кроме того, после сравнений осуществляется копирование n элементов

временного массива в исходный. Таким образом, на каждом шаге выполняется ($3n$ – l) основных операций. В функции `MergeSort` выполняются два рекурсивных вызова. Если число n является степенью ($n = 2^k$) двойки, то глубина рекурсии равна $k = \log_2 n$ (если нет, то $k = \log_2 n + 1$). На m уровне рекурсии выполняются $2m$ вызовов функции `MergeSort`. Каждый из этих вызовов приводит к слиянию $n / 2^m$ элементов, а общее количество операций равно $3(n / 2^m) - 2$. В целом, $2m$ рекурсивных вызова функции `MergeSort` порождает $(3n - 2^m)$ операций. Таким образом, на каждом уровне рекурсии выполняется $O(n)$ операций. Поскольку количество уровней рекурсии равно $\log_2 n$ или $\log_2 n + 1$, в наихудшем и среднем вариантах функция `MergeSort` имеет сложность $O(n\log_2 n)$. Хотя алгоритм сортировки слиянием имеет высокое быстродействие, у него есть один недостаток: необходим вспомогательный массив, состоящий из n элементов. Если объем доступной памяти ограничен, это требование может оказаться неприемлемым.

15.13. Поразрядная сортировка

Алгоритмы поразрядной сортировки рассматривают ключи как числа, представленные в системе счисления с основанием r при различных значениях r (основание системы счисления), и работают с отдельными цифрами чисел. Поразрядная сортировка основана на том, что все числа сортируются при помощи устойчивой сортировки сначала по младшему разряду, затем по всем остальным в порядке их возрастания.

Предположим, что элементы исходного массива B есть T-разрядные положительные десятичные числа $D(j, n)$ – j-я справа цифра в десятичном числе $n >= 0$, т.е. $D(j, n) = \text{floor}(n / m)\%10$, где $m = 10^{j-1}$. Пусть $B0, B1, \ldots, B9$ – вспомогательные массивы (карманы), вначале пустые. Для реализации поразрядной сортировки выполняется процедура, состоящая из двух процессов, называемых распределение и сборка для $j = 1, 2, \ldots, T$.

Распределение заключается в том, что элемент Ki ($i = 1, n$) из B добавляется как последний в массив Bm, где $m = D(j, Ki)$, и таким образом получаем десять массивов, в каждом из которых j-е разряды чисел одинаковы и равны m.

Сборка объединяет массивы $B0, B1, ..., B9$ в этом же порядке, образуя один B.

На рис. 15.10 представлен пример поразрядной сортировки.

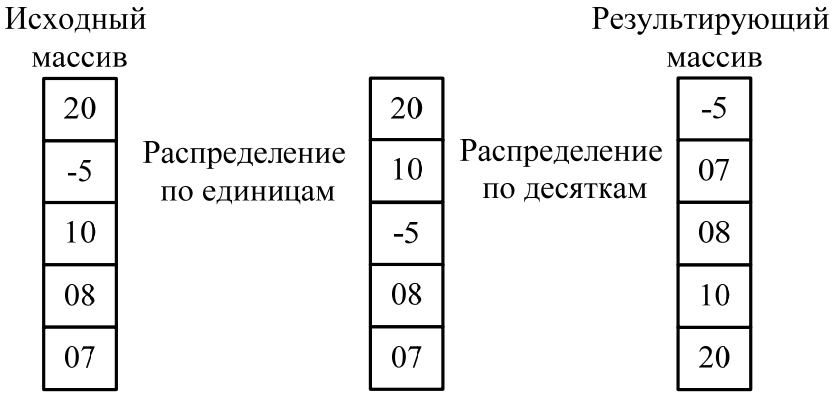


Рис. 15.10. Поразрядная сортировка

Анализ

Временная эффективность поразрядной сортировки – $\Theta(T(n + k))$, где T – количество разрядов, а k – диапазон значений разряда. Недостатком этого метода является необходимость использования дополнительной памяти.

15.14. Сравнение алгоритмов сортировки

Как было показано выше, существует много различных алгоритмов сортировки. Сложность сортировки непосредственно связана с количеством сравнений и количеством обменов, происходящих во время сортировки. Время работы в лучшем и худшем случаях влияет на эффективность сортировки при условии, что одна из этих ситуаций будет встречаться довольно часто. Алгоритм сортировки зачастую имеет хорошее среднее время выполнения, но в худшем случае он работает очень медленно.

В табл. 15.3 показаны приближенные оценки сложности рассмотренных алгоритмов сортировки, в наихудшем и среднем случаях.

Таблица 15.3

Оценки сложности алгоритмов

Алгоритм сортировки	Наихудший случай	В среднем
пузырек	n^2	n^2
выбора	n^2	n^2
вставок	n^2	n^2
слиянием	$n\log_2 n$	$n\log_2 n$
быстрая	n^2	$n\log n$

Каждый программист должен располагать широким набором алгоритмов сортировки. Несмотря на то, что в среднем случае оптимальной

является именно быстрая сортировка, она не является лучшей во всех случаях. Например, при сортировке очень маленького размера входных данных дополнительный объем работы, создаваемый рекурсивными вызовами быстрой сортировки, может перекрыть преимущества алгоритма. В таких случаях один их простых методов сортировки – возможно, даже пузырьковая сортировка – может работать быстрее. Кроме того, если известно, что данные уже почти упорядочены, какой-либо другой алгоритм подойдет лучше, чем быстрая сортировка.

15.15. Практические задания

Ввести размер и элементы массива. Найти для массива порядковые статистики. Выполнить сортировку для массивов размером 1000, 2000, 3000, 4000, 5000.

Массивы заполняются случайными числами. Алгоритмы сортировки реализовать в виде функций в отдельном модуле. Подсчитать трудоемкость каждого из алгоритмов (количество сравнений, операций записи) и вывести зависимость от количества элементов для каждого из алгоритмов, сравнить с известными оценками сложности. Произвести сравнение эффективности алгоритмов. Построить графики зависимости времени сортировки от количества сортируемых элементов.

1. Ввести массив *A*. В массив *B* скопировать все элементы массива *A*, имеющие четный индекс и четное значение. Массив *B* отсортировать по убыванию, используя алгоритмы выбора, пузырька и вставок.

2. Ввести массив *A*. В массив *B* перенести все элементы массива *A*, имеющие четный индекс, справа от которых расположены элементы с нечетным значением. Массив *B* отсортировать по убыванию, используя алгоритмы слияния, быструю и поразрядную сортировку.

3. Ввести массив *A*. В массив *B* перенести все элементы массива *A*, стоящие правее максимального элемента и имеющие нечетный индекс. Массив *B* отсортировать по возрастанию, используя алгоритм слияния, подсчета и пирамидальную сортировку.

4. Ввести массивы *A* и *B*. В массив *C* перенести те элементы массива *A*, которые больше минимального элемента массива *B*, и те элементы массива *B*, которые больше максимального элемента массива *A*. Массивы *A*, *B* и *C* отсортировать по возрастанию, используя методы пузырька, Шелла и шейкер-сортировки.

5. Ввести массивы *A* и *B*. В массив *C* перенести четные элементы массива *A* и нечетные элементы массива *B*. Массивы *A*, *B* и *C* отсортировать по убыванию, используя сортировку методом вставок, выбором и пирамидальную сортировку.

Глава 16. РЕКУРСИЯ

16.1. Введение

Рекурсивным называется способ построения объекта (понятия, системы), в котором определение объекта включает аналогичный объект в виде некоторой его части. Примеры: рекурсивное определение в синтаксисе языка; рекурсивная структура данных – элемент структуры данных содержит один или несколько указателей на такую же структуру данных (односвязный список); рекурсивная функция – тело функции содержит прямой или косвенный собственный вызов.

Функция называется *рекурсивной*, если ее значение для данного аргумента определяется через значения той же функции для предшествующих аргументов.

Рекурсивная задача в общем случае разбивается на ряд этапов. Рекурсивная функция «знает», как решать только простейшую часть задачи – базовую (или несколько). Если функция вызывается для решения базовой задачи, она возвращает результат. Если функция вызывается для решения более сложной задачи, она делит эту задачу на две части: одну часть, которую функция умеет решать, и другую, которую функция решать не умеет. Чтобы сделать рекурсию выполнимой, последняя часть должна быть похожа на исходную задачу, но быть по сравнению с ней проще и меньше. Поскольку эта новая задача подобна исходной, функция вызывает копию самой себя, чтобы начать работать над меньшей проблемой – это называется *рекурсивным вызовом* или *шагом рекурсии*. Шаг рекурсии выполняется до тех пор, пока исходное обращение к функции не закрыто, т.е. пока не закончено выполнение функции. Чтобы завершить процесс рекурсии, каждый раз, как функция вызывает саму себя, должна формироваться последовательность все меньших и меньших задач, в конце сводящихся к базовой задаче. В этот момент функция распознает базовую задачу, возвращает результат предыдущей функции, и последовательность возвратов повторяет весь путь назад, пока не дойдет до первоначального вызова и не возвратит конечный результат.

16.2. Виды рекурсии

Линейная рекурсия

Простейшим примером рекурсии является линейная рекурсия, при которой функция содержит единственный условный вызов самой

себя. В таком случае рекурсия становится эквивалентной обычному циклу. Любой циклический алгоритм можно преобразовать в линейно-рекурсивный и наоборот. Примером линейной рекурсии является вычисление факториала: $n! = 1 \cdot 2 \cdot 3 \ldots (n-1) \cdot n = (n-1)! \cdot n$. Математически можно записать вычисление факториала в следующем виде:

$$n! = \begin{cases} 1, & n = 0, \\ n(n-1)!, & n > 0. \end{cases} \qquad (16.1)$$

В первой строке формулы (16.1) явно указано, как вычислить факториал, если аргумент равен нулю. В любом другом случае для вычисления $n!$ необходимо вычислить предыдущее значение $(n-1)!$ и умножить его на n. Уменьшающееся значение гарантирует, что в конце концов возникнет необходимость найти 0!, который вычисляется непосредственно.

```
int factorial(int n)
{
   if ( n == 0 ) return 1;
   return n * factorial(n-1);
}
```

Чтобы понять, как будет выполняться эта функция, вспомним, что на время выполнения вспомогательного алгоритма основной алгоритм приостанавливается. При вызове новой копии рекурсивного алгоритма вновь выделяется место для всех переменных, объявляемых в нем, причем переменные других копий будут недоступны. При удалении копии рекурсивного алгоритма из памяти удаляются и все его переменные. Активизируется предыдущая копия рекурсивного алгоритма, становятся доступными ее переменные.

Пусть необходимо вычислить 4!. Основной алгоритм: вводится $n = 4$, вызов `factorial(4)`. Основной алгоритм приостанавливается, вызывается и работает `factorial(4)`: $4 \neq 0$, поэтому `factorial:= := factorial(3)·4`. Работа функции приостанавливается, вызывается и работает `factorial(3)`: $3 \neq 0$, поэтому `factorial:= factorial(2)·3`. Заметьте, что в данный момент в памяти компьютера две копии функции `factorial`. Вызывается и работает `factorial(2)`: $2 \neq 0$, поэтому `factorial:= factorial(1)·2`. В памяти компьютера уже три копии функции `factorial` и вызывается четвертая. Вызывается и работает `factorial(1)`: $1 \neq 0$, поэтому `factorial:= factorial(0)·1`. Вызывается и работает `factorial(0)`: $0 = 0$ поэтому `factorial(0)= 1`. Работа этой функции завершена, продолжает работу `factorial(1)`:

`factorial(1):= factorial(0)·1 = 1 * 1 = 1`. Работа функции завершена и работает `factorial(2):= factorial(1) · 2 = 1 · 2 = 2`. Работа этой функции также завершена, и продолжает работу функция `factorial(3):  factorial(3):= factorial(2) · 3 = 2 · 3 = 6`. Завершается работа и этой функции, и продолжает работу функция `factorial(4): factorial(4):= factorial(3) · 4 = 6 · 4 = 24`. Управление передается в основную программу.

Линейная рекурсия – наиболее простой и самый распрстраненный вид рекурсии.

Смешанная (непрямая) рекурсия

При этом виде рекурсии две или более функции вызывают друг друга циклически. Условия завершения могут содержаться во всех или в одной из функций. В качестве примера рассмотрим задачу определения четности числа. Число является четным, если предыдущее число нечетное, и наоборот – число нечетное, если предыдущее четное. Математически:

$$\text{isOdd}(n) = \begin{cases} \text{false}, & n = 0, \\ \text{isEven}(n-1), & n > 0; \end{cases}$$

$$\text{isEven}(n) = \begin{cases} \text{true}, & n = 0, \\ \text{isOdd}(n-1), & n > 0. \end{cases}$$

Код будет выглядеть следующим образом:

```
bool isEven(int n)
{ //условие окончания
        if (n == 0)
        return true;
        else  return isOdd(n - 1);
}
 bool isOdd(int n)
{ //условие окончания
        if (n == 0)
        return false;
else    return isEven(n - 1);
}
```

Графически схема выполнения функций для $n = 4$ представлена на рис. 16.1.

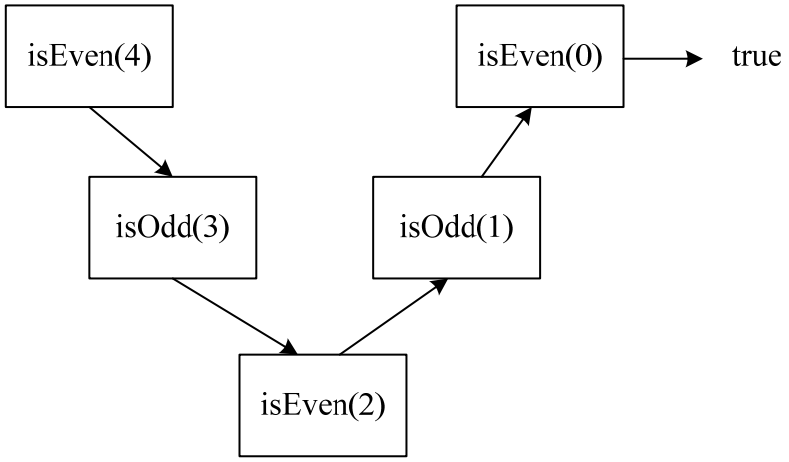


Рис. 16.1. Схема выполнения функций проверки четности

Ветвящаяся рекурсия

В случае ветвящейся рекурсии функция вызывается более одного раза. Частный случай этого вида – бинарная рекурсия (вызов других функций). Рассмотрим пример – вычисления последовательности чисел Фибоначчи. Последовательность чисел Фибоначчи: `0, 1, 1, 2, 3, 5, 8, 13, 21, ...` начинается с `0` и `1` и обладает тем свойством, что каждый следующий член последовательности представляет собой сумму двух предыдущих членов. Эта последовательность часто встречается в природе, в частности она описывает форму спирали. Математически последовательность Фибоначчи можно записать следующим образом:

$$\text{Fibonacci}(n) = \begin{cases} 0, & n = 0, \\ 1, & n = 1, \\ \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2), & n > 1. \end{cases}$$

Код программы будет выглядеть следующим образом:

```
int Fibonacci (int n)
    {
        if (n < 1) return -1;   //ошибка
        if (n == 0)
            return 0;
        else
            if (n == 1)
                return 1;
            else
                return Fibonacci (n-1)+ Fibonacci (n-2);
    }
```

На рис. 16.2 приведена схема вычисления чисел Фибоначчи для $n = 3$.

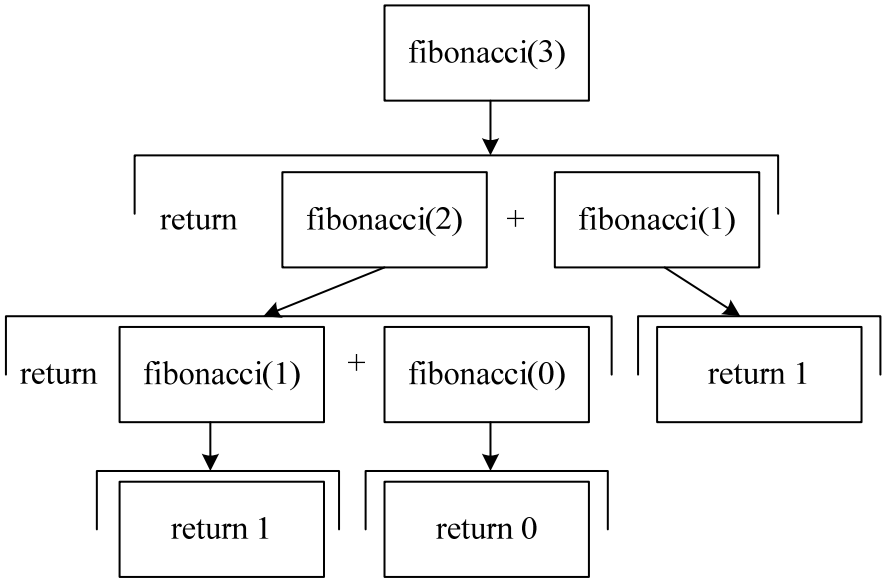


Рис. 16.2. Схема вызова функции вычисления чисел Фибоначчи

Вложенная (гнездовая) рекурсия

Выше рассмотренные рекурсии могут быть заменены итерационным циклом или итерационным циклом со стеком. Однако этот вид рекурсии сложно представить в виде циклической конструкции.

Одним из примеров вложенной рекурсии может быть функция Аккермана. Она определяется следующим образом:

$$\text{Akkerman}(m,n)=\begin{cases} n+1,\ m=0, \\ \text{Akkerman}(m-1,1),\ m>0 \ \text{ и } \ n=1, \\ \text{Akkerman}((m-1),\ \text{Akkerman}(m,n-1)),\ m>0 \text{ и } n>0. \end{cases}$$

Код программы будет выглядеть следующим образом:

```
int Ackkerman(int m, int n)
{
   if (m < 0 || n < 0)
            return -1;
   if (0 == m)
            return n + 1;
      //линейная рекурсия
      else
          if (m > 0 && 0 == n)
          return Ackkerman(m-1, 1);
      //вложенная
          else
             return Ackkerman(m-1, Ackkerman(m, n-1));
}
```

16. 3. Особенности программирования рекурсивных функций

Техника рекурсивных вызовов функций

Очевидно, что рекурсия не может быть безусловной, в этом случае она становится бесконечной. Следовательно, рекурсия должна иметь внутри себя условие завершения, по которому очередной рекурсивный вызов уже не производится.

Рассмотрим пример вызова рекурсивной функции `S`. Движемся по ее тексту до тех пор, пока не встретим ее вызова, после чего опять начинает выполняться та же самая функция сначала, при этом ее первый вызов еще не закончился. Создается впечатление, что текст функции воспроизводится (копируется) всякий раз, когда функция сама себя вызывает:

```
void main()     void S()  void S()  void S()
{               {         {
S();            ..if()S();     ...if()S();    ...if()S();
}               }         }         }
```

Копируется при этом не весь текст функции (не вся функция), а только ее части, связанные с данными (формальные, фактические параметры, локальные переменные и точка возврата). Создается копия локальных данных по следующей схеме:

- в стеке резервируется место для формальных параметров, в которые записываются значения фактических параметров (обычно в порядке, обратном их следованию в списке);
- при вызове функции в стек записывается точка возврата – адрес той части программы, где находится вызов функции;
- в начале тела функции в стеке резервируется место для локальных (автоматических) переменных.

Алгоритм (операторы, выражения) рекурсивной функции не меняется, поэтому он присутствует в памяти компьютера в единственном экземпляре. Таким образом, формальные параметры рекурсивной функции, внешние и локальные переменные не могут быть взаимозаменяемы. Кроме того, каждый новый рекурсивный вызов порождает новый «экземпляр» формальных параметров и локальных переменных, причем старый «экземпляр» не уничтожается, а сохраняется в стеке по принципу вложенности. Здесь имеет место единственный случай, когда в процессе работы программы одному имени переменной соответствуют несколько ее «экземпляров» (рис. 16.3).

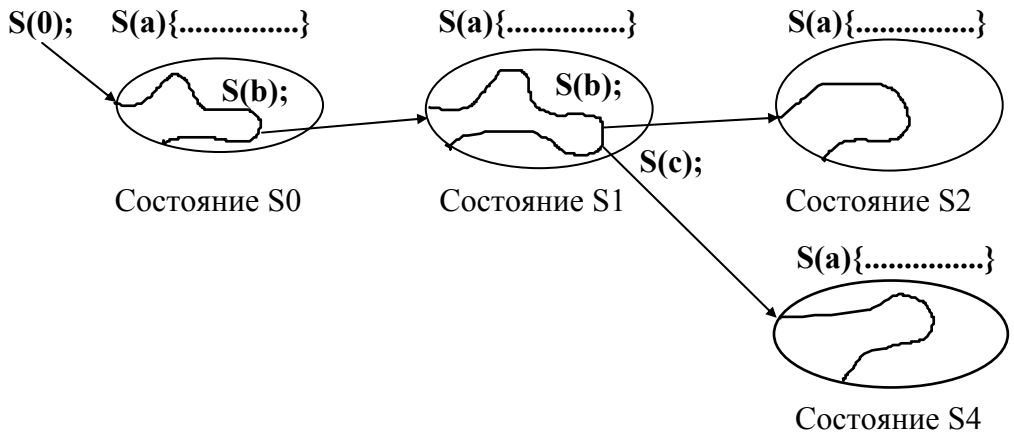


Рис. 16.3. Схема вывоза рекурсивной функции

При завершении рекурсии программа возвращается к предыдущей версии рекурсивной функции и к предыдущему фрейму в стеке.

Принципы программирования рекурсивных функций

Принцип программирования рекурсивных функций имеет много общего с методом математической индукции. Этот метод используется для доказательства корректности утверждений для бесконечной последовательности состояний и неявно применяется при разработке рекурсивных функций. Действительно, сама рекурсивная функция представляет собой переход из n-го в $(n + 1)$-е состояние некоторого процесса. Если этот переход корректен, то есть соблюдение некоторых условий на входе функции приводит к их соблюдению на выходе (то есть в рекурсивном вызове), то эти условия будут соблюдаться во всей цепочке состояний (при безусловной корректности начального).

Опираясь на метод математической индукции, можно сформулировать следующие правила:

- рекурсивная функция разрабатывается как обобщенный шаг процесса, который вызывается в произвольных начальных условиях и который приводит к следующему шагу в некоторых новых условиях;
- обобщенные начальные условия шага – формальные параметры функции;
- начальные условия следующего шага – фактические параметры рекурсивного вызова;
- рекурсивная функция должна проверять условия завершения рекурсии, при которых следующий шаг процесса не выполняется;
- локальными переменными функции должны быть объявлены все переменные, которые имеют отношение к протеканию текущего шага процесса.

Рекурсии и итерации

Как упоминалось выше, некоторые рекурсивные функции могут быть реализованы итеративно. Сравним эти два подхода.

Как итерации, так и рекурсии включают повторение: итерации используют структуру повторения явным образом, рекурсии реализуют повторение посредством повторных вызовов функции. Как итерации, так и рекурсии включают проверку условия окончания: итерации заканчиваются после нарушения условия продолжения цикла, рекурсии заканчиваются после распознавания базовой задачи. Как итерации, так и рекурсии могут оказаться бесконечными.

Несмотря на то, что запись рекурсивных функций бывает очень короткая, они требует больших накладных расходов. При каждом вызове в стеке должны быть размещены все параметры и локальные переменные. На эту работу (как и на последующее освобождение) расходуется и время, и пространство – в программу вставляются соответствующие команды, которые выполняются при каждом вызове, расходуется память под стек. Поскольку рекурсивный вызов выполняется до окончания выполнения функции – размер стека увеличивается при каждом вызове до тех пор, пока не будет достигнута точка, когда выполняется возврат. Размер стека пропорционален глубине рекурсии. *Глубина рекурсии* – это максимальная степень вложенности рекурсивных вызовов. В общем случае глубина будет зависеть от входных данных. Поэтому при разработке рекурсивных функций необходимо минимизировать количество и размеры локальных переменных и параметров. Существует также и другой недостаток – лишние вычисления. Одним из методов борьбы с этим недостатком является сокращение количества рекурсивных обращений в тексте функции (если есть возможность вместо двух вызовов оставить один).

Вывод: рекурсивные вызовы требуют времени и дополнительных затрат. Тогда, в решениях, где требуется высокая эффективность, следует избегать рекурсии.

Тем не менее, рекурсивный подход обычно предпочитают итеративному. Он более естественно отражает задачу и ее результаты, т.е. более нагляден, легче отлаживается и считается хорошим стилем программирования. Другая причина предпочтения рекурсивного решения состоит в том, что итеративное решение может не быть очевидным.

Результат рекурсивной функции

При реализации поисковых задач следует обратить внимание на результат рекурсивной функции. Здесь возможны варианты:

• рекурсивная функция сама выводит выбранный элемент в случае успешного поиска (это приемлемо, если осуществляется поиск первого попавшегося варианта, но не очень хорошо с точки зрения технологии программирования);

• выбранный элемент записывается в область глобальных данных, в которых моделируется стек для возврата результата (само по себе использование глобальных данных не является хорошим решением);

• в качестве результата функции используются более сложные динамические структуры данных, например, список результатов.

Проиллюстрируем все вышесказанное на примере рекурсивных программ.

16.4. Рекурсия и поисковые задачи

С помощью рекурсии легко решаются задачи, связанные с поиском, основанным на полном или частичном переборе возможных вариантов. Принцип рекурсивности заключается здесь в том, что процесс поиска разбивается на шаги, на каждом из которых выбирается и проверяется очередной элемент из множества, а алгоритм поиска повторяется, но уже для «оставшихся» данных. При этом вовсе не важно, каким образом цепочка шагов достигнет цели и сколько вариантов будет перебираться. Единственное, что важно – корректность очередного шага.

Поиск выхода в лабиринте

С точки зрения математики лабиринт представляет собой граф, а алгоритм поиска выхода из него производит поиск пути, соединяющего заданные вершины. Однако для более простого представления лабиринта его можно задавать в виде двумерного массива, в котором значение 1 будет обозначать «стенку», а 0 – «проход»:

```
int LB[10][10]={
{1,1,0,1,1,1,1,1,1,1},
{1,1,0,1,1,1,1,1,1,1},
{1,1,0,0,1,0,0,0,1,1},
{1,1,1,0,0,0,1,0,1,1},
{1,0,1,1,1,0,0,0,1,1},
{0,0,0,0,0,0,1,1,1,1},
{1,1,1,1,1,0,1,1,1,1},
{1,1,1,1,1,0,0,0,1,1},
{1,1,1,1,1,1,1,1,1,1},
{1,1,1,1,1,1,1,1,1,1}};
```

Шаг алгоритма состоит в проверке возможности сделать ход в одном из четырех направлений. Рекурсивный характер алгоритма состоит в том, что в каждой соседней точке реализуется тот же самый алгоритм поиска. Формальными параметрами рекурсивной функции являются координаты точки, из которой в данный момент осуществляется поиск. Фактические параметры – координаты соседней точки, в которой реализуется рекурсивный алгоритм:

```
void step(int x,int y)
{
   step(x+1,y);...   //вверх
   step(x,y+1);...   //направо
   step(x-1,y);...   //налево
   step(x,y-1);...   //вниз
}
```

Результат рекурсивной функции логический – он показывает, можно ли через данную точку достигнуть выхода. Рассмотрим подробнее логику формирования результата на очередном шаге. Если в текущем шаге рекурсии через некоторую соседнюю точку можно достигнуть выхода, то рекурсивный вызов возвратит результат `true`, который должен быть передан и в предыдущую точку (то есть в предыдущий вызов). Если ни одна соседняя точка не возвращает положительного результата, то результат текущего шага также отрицателен:

```
int step(int x,int y)
{...
   if  (step(x+1,y)) return 1;
   if  (step(x,y+1)) return 1;
   if  (step(x-1,y)) return 1;
   if  (step(x,y-1)) return 1;
  return 0;
}
```

Фиксируется только факт того, что выход найден, но найденный путь не возвращается. Последнее можно сделать разными способами. Например, использовать глобальные данные для отметки прохода:

```
int step(int x,int y)
{...
    if  (step(x+1,y)) { LB[x][y]=2; return 1;}
    if  (step(x,y+1)) { LB[x][y]=2; return 1;}
    if  (step(x-1,y)) { LB[x][y]=2; return 1;}
    if  (step(x,y-1)) { LB[x][y]=2; return 1;}
  return 0;
}
```

Следующий шаг – отсечение недопустимых точек, в данном случае это стенки лабиринта. Здесь же вводится ограничение рекурсии – выход найден, если достигнут край лабиринта:

```
int step(int x,int y)
{
   if (LB[x][y]==1) return 0;
   if (x==0 || x==9 || y==0 || y==9) return 1; ...
}
```

Последнее ограничение: алгоритм не должен возвращаться в ту точку, из которой он только что пришел. Если в лабиринте есть замкнутые пути, то произойдет зацикливание, даже если помнить предыдущую точку. Выход – в отметке всех пройденных точек лабиринта. Но при этом следует заметить, что каждая точка отмечается только на время рекурсивного вызова, если эти вызовы дают отрицательный результат, то отметка снимается. Предлагаемый ниже вариант с отметками пройденных точек, кроме всего прочего, сохраняет отмеченным путь, который ведет к выходу из лабиринта:

```
int step(int x,int y)
{
   if (LB[x][y]==1) return 0;    //стенки
   if (x==0 || x==9 || y==0 || y==9) return 1;  //края
   LB[x][y]=2;       //отметить точку
    if  (step(x+1,y)) return 1;
    if  (step(x,y+1)) return 1;
    if  (step(x-1,y)) return 1;
    if  (step(x,y-1)) return 1;
   LB[x][y]=0;       //снять отметку
  return 0;
}
```

Обход шахматной доски

В качестве примера рассмотрим проектирование функции для поиска последовательности обхода конем шахматной доски.

Алгоритм обхода можно разбить на последовательность шагов. Каждый шаг характеризуется начальным состоянием – текущим положением коня. Алгоритм на каждом шаге предполагает проверку возможности сделать ход в каждом из 8 допустимых направлений. Схема алгоритма имеет следующий вид:

```
void step(int x0, int y0)
{          //формальные параметры - текущая позиция коня
           //приращения координат для 8 ходов
```

```
static int xy[8][2] = {{1,-2},{1, 2},{-1,-2},{-1, 2},
                       {2,-1},{2, 1},{-2, 1},{-2,-1}};
     int  i;   //локальный параметр - номер хода
     if (x0 < 0 || x0 >=7 || y0 < 0 || y0 >=7)
     return;   //выход за пределы доски
     for (i=0; i<8; i++)
     step(x0 + xy[i][0], y0 + xy[i][1]);
               //выполнить следующий ход
}
```

Предыдущий шаг дает полный перебор возможных последовательностей ходов коня. Функция имеет результат `void`. Необходимо определить, как производить выбор необходимой последовательности. Очевидно, функция должна давать логический результат – 0, если при выполнении очередного хода в данном направлении задача не решается, и 1, если решается успешно. При этом цикл, в котором происходит рекурсивный вызов – просмотр очередных ходов, выполняется до первого успешного вызова:

```
int  step(int x0, int y0)
{
static int xy[8][2] = {{ 1,-2},{ 1, 2},{-1,-2},{-1, 2},
                       { 2,-1},{ 2, 1},{-2, 1},{-2,-1}};
   int    i;         //локальный параметр - номер хода
   if (x0 < 0 || x0 >=7 || y0 < 0 || y0 >=7 )
     return;         //выход за пределы доски
  for (i=0; i<8; i++)
     if (step(x0 + xy[i][0], y0 + xy[i][1]))
        return 1;    //поиск успешного хода
   return 0;         //последовательность не найдена
 }
```

Необходимо определить условия начала и завершения рекурсивного процесса, а также условия взаимодействия шагов между собой. Условием завершения процесса является обход всех 64 полей доски. При этом не должно производиться повторное попадание на то же самое поле. Необходимо фиксировать и отмечать количество пройденных полей. Это можно сделать, используя внешние переменные, так как они проверяются на разных шагах рекурсии. Номер шага будем использовать также для отметки поля. По завершению программы массив полей будет содержать номера шагов коня:

```
int  desk[8][8];           //поля доски
int  nstep;                //номер шага
int  step(int x0, int y0)
```

```
{
static int xy[8][2] = {{ 1,-2},{ 1, 2},{-1,-2},{-1, 2},
                       { 2,-1},{ 2, 1},{-2, 1},{-2,-1}};
   int    i;         //локальный параметр – номер хода
   if (x0 < 0 || x0 >=7 || y0 < 0 || y0 >=7 )
     return 0;       //выход за пределы доски
   if (desk[x0][y0] !=0)
     return 0;                 //поле уже пройдено
     desk[x0][y0] = nstep++;  //отметить свободное поле
   if (nstep == 64)
     return 1;       //все поля отмечены – успех
   for (i=0; i<8; i++)
     if (step(x0 + xy[i][0], y0 + xy[i][1]))
         return  1;      //поиск успешного хода
     nstep--;            //вернуться на ход назад
     desk[x0][y0] = 0;   //стереть отметку поля
         return  0;      //последовательность не найдена
}
void   Path(int x0, int y0)
{
   int    i, j;          //очистка доски
   for (i=0; i<8; i++)
     for (j=0; j<8; j++)
          desk[i][j] =0;
   nstep = 1;            //установить номер шага
   step(x0,y0);    //вызвать функцию для исходной позиции
}
```

Ханойские башни

Ни одно рассмотрение рекурсии не было бы полным без рассмотрения старинной задачи о ханойских башнях. Имеется три стержня и *n* дисков, которые помещаются на трех стержнях. Диски различаются размерами и вначале размещаются на одном из стержней от самого большого внизу до самого маленького вверху (рис. 16.4). Задача состоит в перемещении дисков на соседнюю позицию (стержень) при соблюдении следующих правил: одновременно можно перемещать только один диск; ни один диск не может быть помещен поверх диска меньшего размера. Легенда гласит, что конец света наступит раньше, чем группа монахов справится с задачей размещения 40 золотых дисков на трех алмазных стержнях.

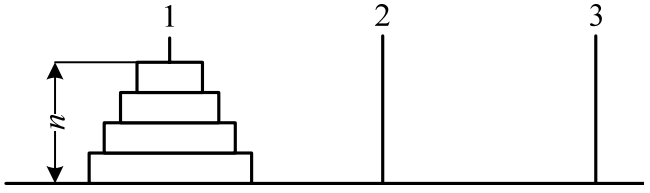


Рис. 16.4. Постановка задачи о ханойских башнях

Для решения задачи можно применить метод, называемый «разделяй и властвуй», при котором используется два рекурсивных вызова, каждый из которых работает приблизительно с половиной входных данных.

Введем обозначения: 1-й стержень – *i*; 2-й – *w*; 3-й – *j*. Для решения простейшего случая задачи, когда пирамида состоит только из одного диска, необходимо выполнить одно действие – перенести диск со стержня *i* на стержень *j*, что очевидно (этот перенос обозначается $i \rightarrow j$). Общий случай задачи изображен на рис. 16.5, когда требуется перенести *n* дисков со стержня *i* на стержень *j*, считая стержень *w* вспомогательным. Сначала следует перенести $(n - 1)$ диск со стержня *i* на стержень *w* при вспомогательном стержне *j*, затем перенести один диск со стержня *i* на стержень *j* и, наконец, перенести $(n - 1)$ дисков с *w* на стержень *j*, используя вспомогательный стержень *i*. Итак, задача о переносе *n* дисков сводится к двум задачам о переносе $(n - 1)$ дисков и одной простейшей задаче. Схематически это можно записать так: $T(n, i, j, w) = T(n - 1, i, w, j), T(1, i, j, w), T(n - 1, w, j, i)$.

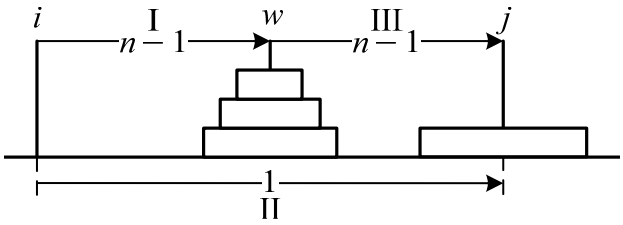


Рис. 16.5. Схема перемещения дисков

В правильности работы этого решения можно удостовериться методом индукции. Рекурсивный алгоритм «разделяй и властвуй» решения задачи о ханойских башнях дает решение, приводящее к $(2^n - 1)$ перемещениям. Количество перемещений дисков, удовлетворяющее условию рекуррентности, определяется формулой $T_n = 2T_{n1} + 1$, при $n > 2$, $T_1 = 1$.

Ниже приведена программа, которая вводит число *n* и печатает список перемещений, решающая задачу о ханойских башнях при количестве дисков *n*:

```
.............................
int _tmain(int argc, _TCHAR* argv[])
{.............................
   void tn(int, int, int, int);    //функция
   int n;
   scanf(" %d",&n);
   Tn (n,1,2,3);
}
 .............................
```

```
void Tn(int n, int i, int j, int w)      //рекурсивная
   {  if (n > 1)                         //функция
      {  Tn (n-1,i,w,j);
         Tn (1,i,j,w);
         Tn (n-1,w,j,i);
      }
      else printf(" \n %d -> %d",i,j);
      return ;
   }
```

Используется внутренняя рекурсивная функция `Tn(n,i,j,w)`, печатающая перемещения, необходимые для переноса *n* дисков со стержня *i* на стержень *j* с использованием вспомогательного стержня *w* при $\{i, j, w\} = \{1, 3, 2\}$. Каждый раз при вызове функции `Tn` под параметры *n*, *i*, *j*, *w* выделяется память и запоминается место возврата. При возврате из `Tn` память, выделенная под параметры *n*, *i*, *j*, *w*, освобождается, и становится доступной память, выделенная под параметры *n*, *i*, *j*, *w* предыдущим вызовом, а управление передается в место возврата.

Другие рекурсивные задачи

Рекурсия лежит в основе ранних теоретических исследований природы вычислений. Рекурсивные функции и программы играют главную роль в математических исследованиях, в которых предпринимается попытка разделения задач на поддающиеся решению на компьютере и на непригодные для этого. На основе рекурсии реализуются генераторы перестановок; доказательства утверждений (с помощью систем искусственного интеллекта; построением контрпримеров; полным перебором и проверкой правильности всех возможных случаев, возникающих при рассмотрении утверждения; синтетическим методом); алгоритмы сортировки (простым выбором, обменом, слиянием, пирамидальная, быстрая), алгоритмы работы с матрицами (нахождение обратной матрицы, разложение матриц, функции решения линейных алгебраических уравнений); задачи, основанные на переборе с возвратом (латинские квадраты, задача о назначениях, задача коммивояжера, задача о рюкзаке); финансовые функции; решение рекуррентных соотношений; фракталы (кривые Гилберта, Серпинского, Коха, функция Веерштрасса и т.д.); операции над множествами; перевод чисел из одной с/с в другую и многие другие задачи.

16.5. Практические задания

1. Составить рекурсивную функцию подсчета количества $x(m)$ разбиений натурального числа m, то есть его представлений в виде суммы натуральных чисел.

2. Составить рекурсивную функцию подсчета количества всех положительных делителей натурального числа n.

3. Для целых неотрицательных чисел n, m разрешены операции: нахождения последующего числа $(n + 1)$ и предыдущего числа $(n - 1)$ $(n > 0)$. Промоделировать с помощью рекурсивных функций операции нахождения суммы $(n + m)$, разности $(n - m)$, умножения $(n \times m)$, возведения в степень n^m $(n > 0)$.

4. Последовательность из латинских букв строится следующим образом. На нулевом шаге она пуста. На каждом последующем шаге последовательность удваивается, то есть приписывается сама к себе, и к ней слева добавляется очередная буква алфавита (a, b, c, …). По заданному числу n определить символ, который стоит на n-м месте последовательности, получившейся после шага 26.

5. Пусть требуется «развернуть» текстовую строку, в которой повторяющиеся фрагменты заключены в скобки, а после открывающейся скобки может находиться целая константа, задающая число повторений этого фрагмента в выходной строке. Например: «aaa(3bc(4d)a(2e))aaa» разворачивается в «aaabcddddaeebcddddaeebcddddaeeaaa».

Глава 17. АБСТРАКТНЫЙ ТИП ДАННЫХ. СПИСКИ

17.1. Абстрактные типы данных

Абстрактные типы данных (АТД) позволяют создавать программы с использованием высокоуровневых абстракций. За счет применения абстрактных типов данных появляется возможность отделять абстрактные (концептуальные) преобразования, которые программы выполняют над данными, от любого конкретного представления структуры данных и любой конкретной реализации алгоритма.

Абстрактные конструкции более высокого уровня часто создаются на основе более простых конструкций. На всех уровнях действует один и тот же основной принцип – необходимо найти наиболее важные операции и наиболее важные характеристики данных, затем точно определить и те, и другие на абстрактном уровне и разработать эффективные конкретные механизмы для их поддержки.

Абстрактный тип данных – это тип данных (набор значений и совокупность операций для этих значений), доступ к которому осуществляется только через *интерфейс*. Программу, которая использует АТД, будем называть *клиентом*, а программу, в которой содержится спецификация этого типа данных – *реализацией*.

Программисты часто определяют интерфейсы в заголовочных файлах, описывающих наборы операций для некоторых структур данных; при этом реализации находятся в каком-нибудь другом, независимом файле программы. Такой порядок представляет собой соглашение между пользователем и разработчиком и служит основой для создания стандартных библиотек, доступных в средах программирования на языке С.

Элементы данных называются *данными-членами*. В объекте могут быть также определены *функции-члены*, которые реализуют операции, связанные с этим типом данных. В языке С++ (но не в С) у структур могут быть связанные с ними функции, через которые можно выразить операции объекта.

Для доступа к функциям-членам текущего объекта может использоваться ключевое слово `this` – указатель на объект, для которого он вызывается.

Когда к данным-членам применяется ключевое слово `static`, это означает, что существует только одна копия этой переменной (относящаяся к структуре), а не множество копий (относящихся к отдельным объектам).

Абстрактные типы данных возникли в качестве эффективного механизма поддержки модульного программирования как принципа организации больших современных систем программного обеспечения. Они являются средством, позволяющим ограничить размеры и сложность интерфейса между (потенциально сложными) алгоритмами и связанными с ними структурами данных, с одной стороны, и программами (потенциально – большим количеством программ), использующими эти алгоритмы и структуры данных, с другой стороны.

В настоящее время часто используется термин контейнер. *Контейнер* – это структура данных, главной задачей которой является хранение других объектов, причем способ их хранения четко определен. Примером контейнера может служить массив, структура, встроенная в язык. *Списки* представляют собой пример контейнерных классов.

17.2. Список как динамическая структура данных

Динамическая структура данных представляет собой множество переменных, связанных между собой указателями. Каждый элемент такой структуры содержит один или несколько указателей на аналогичные элементы. Например:

```
struct    list
{
     int  value;
     list *next;          //указатель
     list *next,*pred;    //два указателя
     list *links[10];     //ограниченное кол-во указателей
     list **plinks;       //произвольное кол-во указателей
};
```

Из данного описания нельзя определить ни количество переменных в структуре данных, ни характер связи между ними (последовательный, циклический, произвольный). Следовательно, конкретный тип динамической структуры данных (список, дерево, граф) зависит от функций, которые работают с этой структурой.

Рассматриваемые ниже структуры данных являются динамическими по двум причинам: сами переменные таких структур создаются как динамические переменные; количество связей между переменными и их характер также определяются динамически в процессе работы программы.

Последовательность обхода зависит не от физического размещения элементов в памяти, а от последовательности их связывания указателями. Нумерация элементов списка – *логический номер* элемента в списке, номер, получаемый им в процессе движения по списку.

Динамические структуры данных доступны, как правило, через указатель на некоторый ее элемент, который называется *заголовком*.

Наиболее простыми динамическими структурами данных являются списки. *Список* представляет собой линейную последовательность элементов, каждый из которых содержит указатели на аналогичные элементы (соседи). Элементы определяются ссылками на узлы, поэтому связные списки иногда называют самоссылочными структурами.

Основное преимущество связных списков перед массивами заключается в возможности эффективного изменения расположения элементов. За эту гибкость приходится жертвовать скоростью доступа к произвольному элементу списка, поскольку единственный способ получения элемента состоит в отслеживании связей от начала списка.

Списки подразделяют на:

- односвязные – каждый элемент списка имеет указатель на следующий (рис. 17.1);
- двусвязные – каждый элемент списка имеет указатель на следующий и на предыдущий элементы;
- двусвязные циклические – первый и последний элементы списка ссылаются друг на друга (рис. 17.2).

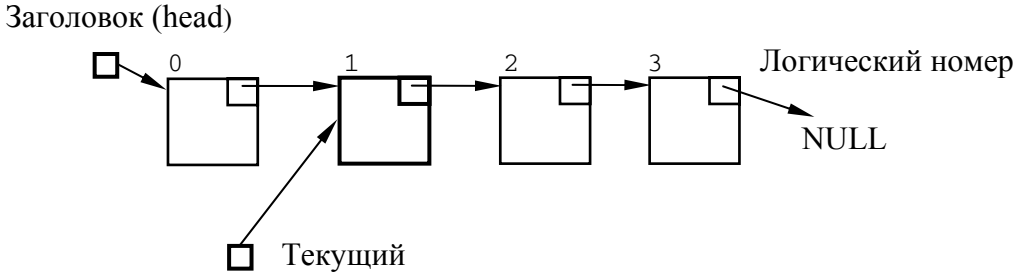


Рис. 17.1. Односвязный список

Односвязные списки являются наиболее простыми. Основным их недостатком является возможность просмотра только в одном направлении – от начала к концу. Без дополнительных «ухищрений» нельзя получить указатель на предыдущий элемент списка, который необходим при удалении текущего. Для односвязного списка наиболее простыми являются операции включения и исключения элементов в начале и конце списка, соответственно, они используются для моделирования таких структур данных, как стеки и очереди.

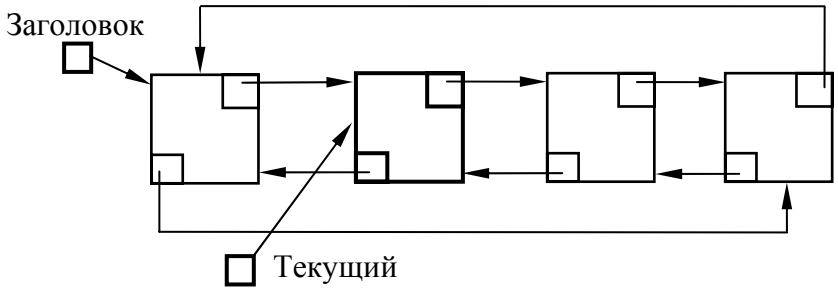


Рис. 17.2. Двусвязный список

Двусвязные списки дают возможность просмотра элементов в обоих направлениях и являются наиболее универсальными. Операции включения-исключения элементов имеют примерно одинаковый уровень сложности. Они используются для создания цепочек элементов, которые допускают частые операции включения, исключения, упорядочения, замены и прочие.

В односвязных и двусвязных списках последний элемент содержит указатель `NULL` для обозначения факта окончания последовательности. Аналогично первый элемент двусвязного списка содержит указатель `NULL` на предыдущий элемент. В этом случае работа с первым и последним элементом списка имеет свои особенности. В качестве альтернативы может быть предложен циклический список, у которого последний элемент ссылается на первый, а первый – на последний. Даже если данная замкнутая структура используется для представления обычной линейной последовательности, работающие с ним функции являются более простыми.

Все эти структуры данных предполагают различные способы организации множества переменных. При этом обычный массив и список имеют противоположные свойства: если массив допускает произвольный доступ к своим элементам и ускоренный (двоичный) поиск при их упорядоченности, то список допускает только последовательный просмотр. Следовательно, массив обладает преимуществами с точки зрения операций поиска. Но, с другой стороны, изменение порядка следования переменных в массиве сопровождается их физическим перемещением, а в списке приводит только к «перебросе» соответствующих указателей. Таким образом, списки предпочтительнее для структур данных, в которых операции поиска встречаются значительно реже, чем операции по изменению порядка следования элементов, и наоборот. В отношении массивов указателей и списков можно заметить, что массивы указателей критичны к количеству элементов в структуре данных – при увеличении этого количества их необходимо расширять,

при уменьшении – желательно сокращать. Для списков такой проблемы не существует. Таким образом, списки предпочтительнее в структурах, где количество элементов меняется в широких пределах, массивы указателей – в противном случае.

17.3. Работа со списками

Элемент списка состоит из информационной части – ключа и данных, и полей связок – указателей на аналогичные элементы. Пример структуры двусвязного списка будет выглядеть следующим образом:

```
//------------------list.h
struct Element      //элемент списка
{
     void*    Data; //данные
     int      Key;  //ключ
     Element* Prev; //указатель на предыдущий элемент
     Element* Next; //указатель на следующий элемент
}
```

В таком списке каждый элемент содержит два указателя, один (`prev`) указывает на предыдущий элемент, а другой (`next`) – на следующий, и два информационных поля: `Key` – ключ и `Data` – данные. В процессе работы со списками будет получено множество экземпляров этой структуры, по одному для каждого элемента. Как только возникает необходимость использовать новый узел, для него следует зарезервировать память.

В языке C++ принято инициализировать область хранения, а не только выделять для нее память. В связи с этим обычно в каждую описываемую структуру включается *конструктор* (constructor). Конструктор представляет собой функцию, которая описывается внутри структуры и имеет такое же имя. Он предназначен для предоставления исходных значений данным структуры. Для этого конструкторы автоматически вызываются при создании экземпляра структуры. Например, если описать элемент списка при помощи следующего кода:

```
struct Element      //элемент списка
{
     void*    Data; //данные
     int      Key;  //ключ
     Element* Prev; //указатель на предыдущий элемент
     Element* Next; //указатель на следующий элемент
                    //конструктор
```

```
      Element
      (Element* prev, void* data, int key, Element* next)
      {
            Prev = prev;
            Data = data;
            Key  = key;
            Next = next;
      }
}
```

Тогда, оператор

```
Element* A = new Element(NULL,data,key,Head);
```

не только резервирует достаточный для элемента объем памяти и возвращает указатель на него в переменной `A`, но и присваивает полю `Data` элемента значение `data`, `Key` значение `key`, а указателям поля – значение `prev` и `next`. Конструкторы помогают избегать ошибок, связанных с инициализацией данных.

Особенности работы с динамическими структурами данных заключаются в частом использовании операции косвенного обращения по указателю к структуре или к ее элементу – `*` или `->`. При этом обычно используется указатель, который ссылается на текущий элемент структуры данных. Весь просмотр структуры данных заключается в циклическом переходе от одного текущего элемента к другому.

Для успешной работы со списками необходимо прежде всего научиться интерпретировать средствами языка такие понятия, как предыдущий, текущий, последующий, первый, последний элементы, переход к предыдущему, следующему и т.д.

Интерпретация перемещений указателей

При работе со списками каждый указатель имеет определенный смысл – первый, текущий, следующий, предыдущий и т.п. элементы списка. Поля `prev`, `next` также интерпретируются как указатели на предыдущий и следующий элементы списка, доступные через указатели. Например, если `p` – указатель на новый элемент, а `q` – указатель на текущий, то выражение `q->prev->next = p` интерпретируется как «указателю на следующий элемент списка в предыдущем от текущего присвоить указатель на новый».

Операция перемещения указателя реализуется через операцию присваивания одному указателю значения другого. На рис. 17.3 это соответствует перенесению (копированию) требуемого указателя из

одной ячейки в другую. В левой части операции присваивания должно находиться обозначение ячейки, в которую заносится новое значение указателя. Причем ячейка может быть достижима только через имеющиеся рабочие указатели. Этому соответствует цепочка операций `q->prev->next`. В правой части операции присваивания должно находиться обозначение ячейки, из которой берется значение указателя – `p`.

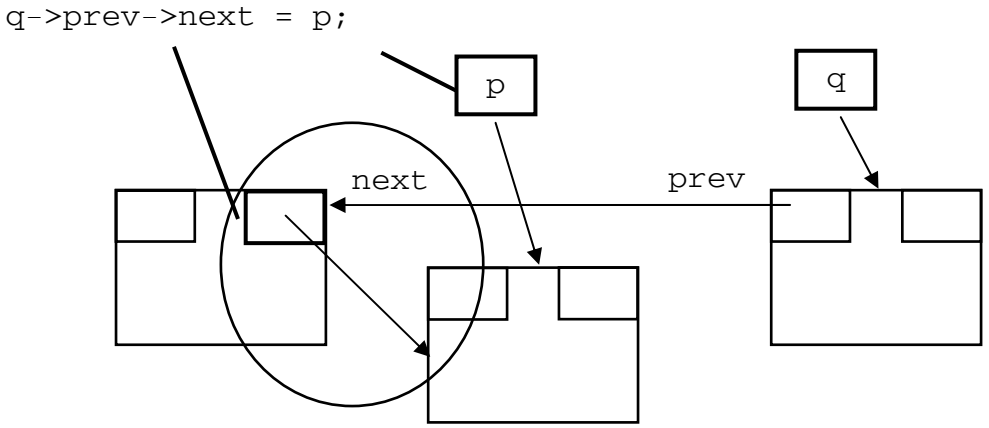


Рис. 17.3. Схема присваения укзателя

При работе со списками часто используется соглашение: в начале каждого списка содержится фиктивный узел, называемый началом или заголовком списка (head node). Поле элемента ведущего узла игнорируется, но ссылка узла сохраняется в качестве указателя узла, содержащего первый элемент списка.

```
Element* Head;
Head = NULL;
```

Ситуация, в которой удобно использовать начальный элемент, возникает, когда необходимо передать спискам указатели в качестве аргументов функций. Эти аргументы могут изменять список таким же образом, как это выполняется для массивов. Использование ведущего узла позволяет функции принимать или возвращать пустой список. При отсутствии ведущего узла функция нуждается в механизме информирования вызывающей функции в случае, когда оставляется пустой список. Одно из решений для C++ состоит в передаче указателя на список как ссылочного параметра. Второй механизм предусматривает прием функциями обработки списков указателей на списки, ввода в качестве аргументов и возврат указателей на списки вывода.

Рассмотрим интерфейсные функции работы с двусвязным списком. Структура `List` объявляет набор функций, которые реализуют

базовый список операций, что позволяет избегать повторения кода и зависимости от деталей реализации:

```
//------------------list.h----------------------
struct List
{
Element* Head;
List(){Head = NULL;};
Element* GetFirst(); //получить первый элемент списка
Element* GetLast();  //получить последний элемент списка
Element* Search (void* data);
                     //найти  первый элемент по значению
void PrintList(void(*f)(void*));
                     //f обработка элементов списка
int  CountList();    //подсчет количества элементов списка
bool Insert(void* data, int key);
                     //добавить элемент по ключу
bool InsertEnd(void* data,int key);
                     //добавить эл. по ключу в кон.
bool Delete(Element* e);  //удалить первый по ссылке
bool Delete(void* data);  //удалить первый по значению
bool Sort();              //сортировать
bool isListEmpty();       //проверка на пустоту
bool DeleteList();        //очистить список
}
.............................
int _tmain(int argc, _TCHAR* argv[])
{.............................
//создание объекта списка в клиентской программе
List* A = new List();
.............................
}
```

Перемещение по списку

Данные операции являются вспомогательными и используются другими функциями. Чтобы получить первый элемент списка необходимо вернуть – `Head`. Для текущего элемента следующий будет `Next`, а предыдущий `Prev`:

```
//------------------------получить первый элемент списка
Element* List::GetFirst() { return Head; };
//----------------------------получить следующий
Element* List::GetNext()  {return this->Next;};
//----------------------------получить предыдущий
Element* List::GetPrev()  {return this->Prev;};
//---------------------получить последний элемент списка
```

Для получения последнего элемента списка в функции `GetLast()` организуется движение по указателям, начиная с заголовка `Head`. Если текущий `temp==NULL`, это значит, что достигнут конец списка, и функция возвратит предыдущий перед пустым `NULL`. Для организации перемещения по списку используется функция получения следующего `GetNext()`:

```
Element* List::GetLast()
{
     Element* temp = Head, *x = temp;
     while (temp != NULL)     //пока не конец списка
   {
     x = temp;
     temp = temp ->GetNext(); //перейти к следующему
   };
     return x;
}
```

Функция `CountList()` возвращает размер списка, подсчитывая его элементы:

```
//------------------------------подсчет количества
элементов списка int List::CountList()
  {
     Element* temp = Head;       //начиная с  начала
     int count = 0;              //обнулить счетчик
     while (temp != NULL)        //пока не конец
     {
       count ++;                 //увеличить счетчик
       temp = temp ->GetNext(); //перейти к следующему
     };
     return count;
  }
```

Поскольку предполагается, что такая операция не будет частой, лучше затратить здесь дополнительное время, чем терять его на модификацию переменной размера списка при каждом включении-удалении элемента.

Поиск

Функция `Search()` находит в списке с помощью простого линейного поиска первый элемент, имеющий значение `data`. Точнее говоря, она возвращает указатель на этот элемент или `NULL`, если элемент не найден:

```
//----------------------найти  первый элемент по значению
Element* List::Search (void* data)
{
   Element* temp  = Head;        //начиная с заголовка
                                 //пока не конец списка
                                 //и не найден элемент
   while((temp != NULL) && (temp->Data != data))
          temp = temp ->Next;    //перейти к следующему
   return temp;        //вернуть указатель на найденный
}
```

Поиск в списке из n элементов требует в худшем случае $\Theta(n)$ операций.

Вставка

Функция вставки `Insert()` добавляет элемент с ключом `key` и данными `data` к текущему списку, помещая его в начало списка.

```
//------------добавить элемент в начало списка по ключу
bool List::Insert (void* data, int key)
{                   //вставить в начало
  if (Head == NULL) Head = new
                           Element(NULL,data,key,Head);
  else
Head = (Head->Prev = new Element(NULL, data,key,Head));
  return true;
}
```

Функция выполняется за время $O(1)$ (не зависит от длины списка).

Следующая функция `InsertEnd()` добавляет элемент с ключом `key` и данными `data` после последнего (в конец), если список не пуст, и в начало при пустом списке:

```
//------------- добавить элемент по ключу в конец списка

bool List::InsertEnd  (void* data, int key)
{
 if (Head == NULL) Head = new
                          Element(NULL,data,key,Head);
 else {
        Element * temp = GetLast();   //после последнего
        temp ->Next = new Element(temp, data,key,NULL);
                                      //вставить
      }
 return true;
}
```

На рис. 17.4 приведена графическая интерпретация вставки элемента в конец при непустом списке.

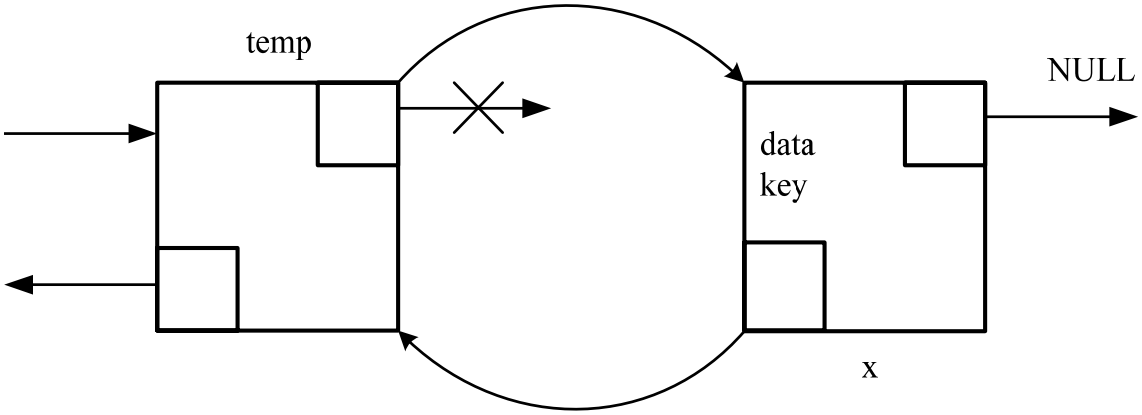


Рис. 17.4. Схема вставки элемента в конец

Удаление

Для операции удаления `Delete()` не требуется дополнительной информации об элементе, предшествующем удаляемому (либо следующим за ним) в списке (как это имеет место для односвязных списков), эта информация содержится в самом элементе. Действительно, главная особенность двусвязных списков состоит в возможности удаления элемента, когда ссылка на него является единственной информацией. В данном примере указатель на удаляемый элемент передается при вызове функции в качестве аргумента:

```
//--------------------------- удалить первый по ссылке
bool  List::Delete(Element* x)
{
   if  (x!= NULL)
       {
        if (x->Next != NULL) x->Next->Prev = x->Prev;  //1
        if (x->Prev != NULL) x->Prev->Next = x->Next;  //2
        else { Head = x->Next; Head->Prev=NULL; }
                       //удаляем из начала
        delete x;      //осободить память
        return true;
       }
   else
        return false;
}
```

Как было сказано выше, указатель элемента предоставляет достаточно информации для удаления узла, что видно из рис. 17.5.

Для данного x указателю x->Next->Prev присваивается значение x->Prev (ссылка 1), а указателю x->Prev->Next – значение x->Next (ссылка 2).

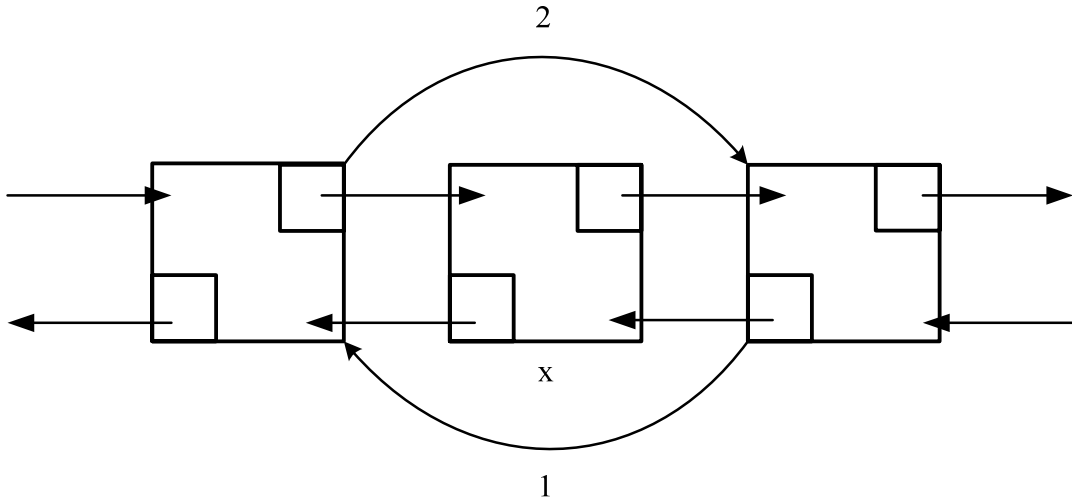


Рис. 17.5. Схема удаления элемента списка

При удалении элемента из списка задача сводится к перераспределению ссылок таким образом, чтобы элемент больше не был привязан к списку. Система утилизирует пространство, занимаемое элементом, чтобы всегда иметь возможность выделять его под новый узел в операторе new.

В функции Delete() необходимо предусмотреть обработку ошибочных ситуаций, поскольку попытка удаления элемента из пустого списка является довольно распространенной ошибкой и может нарушить структуру динамической памяти, что скажется только впоследствии.

Если задано значение data, по которому необходимо удалить элемент, то перед удалением надо найти его указатель с помощью функции Search():

```
//-------------------------- удалить первый по значению
bool List::Delete(void* data)
  {  return Delete(Search(data)); };
     //найти и удалить
```

Так как при добавлении элементов в список под них выделялась память, то при удалении списка память необходимо вернуть системе для повторного использования. Для этого можно поэлементно, начиная с конца списка, удалять элементы и освобождать память до тех пор, пока список не будет пустым. Функция вызывается рекурсивно:

```
//---------------------------- очистить список
bool List::DeleteList()
{
   Element *temp =GetLast();  //начиная с последнего
   if(temp)                   //пока есть элементы
   {
       if (temp ->Prev != NULL)
              temp ->Prev->Next = temp ->Next;
       else   Head = temp ->Next;
       delete temp;           //освободить память
       DeleteList();          //рекурсивный вызов
    }
    return true;
}
```

Вывод списка

Для обработки элементов списка используется функция `Print-List()`, которая для каждого из элементов списка, начиная с заголовка `Head` вызывает функцию `f()` обработки поля `Data`:

```
//--------------------------f обработка элеметов списка
void List::PrintList(void(*f)(void*))
{
   Element* temp = Head;
   while (temp != NULL)
   {    f(temp ->Data);
          temp = temp ->GetNext();
   };
}
```

Проблема концов списка и циклические списки

Как видно из рассмотренных выше функций, основной сложностью при работе со списками является необходимость проверки множества вариантов при выполнении операций над элементом списка: список пустой; элемент единственный; элемент в начале списка; элемент в конце списка; элемент в середине списка.

Для решения проблемы «концов списка» удобно бывает сделать его циклическим, то есть вместо указателей `NULL` записывать:

- указатель на последний элемент в качестве указателя на предыдущий в первом элементе списка;
- указатель на первый элемент в качестве указателя на последующий в последнем элементе списка.

Тогда, единственный элемент в циклическом списке будет иметь указатели на самого себя, а в процессе просмотра, конец списка будет определяться фактом возвращения на его начало.

При выполнении операций над циклическим списком необходимо следить за его заголовком. Например, операция включения в начало списка будет отличаться от включения в конец списка только перемещением заголовка на новый элемент.

Представление списков массивами

Использование указателей не единственный способ представления списков. Например, если язык программирования не позволяет использовать указатели и допускает только работу с массивами, то двусвязный список можно получить на основе, скажем, трех массивов (моделирование указателей с помощью курсоров, т.е. целых чисел, которые указывают на позиции элементов в массивах). В первом массиве могут храниться значения элементов списка, во втором и третьем – индексы предыдущего и последующего элементов списка.

17.4. Нелинейные разветвленные списки

Нелинейным разветвленным списком является список, элементами которого могут быть тоже списками. Если один из указателей каждого элемента списка задает порядок, обратный к порядку, устанавливаемому другим указателем, то такой двусвязный список будет линейным. Если же один из указателей задает порядок произвольного вида, не являющийся обратным по отношению к порядку, устанавливаемому другим указателем, то такой список будет нелинейным. При обработке нелинейный список определяется как любая последовательность атомов и списков (подсписков), где в качестве атома берется любой объект, который при обработке отличается от списка тем, что он структурно неделим.

Разветвленные списки описываются тремя характеристиками: порядком, глубиной и длиной.

Типичный пример применения разветвленного списка – представление алгебраического выражения в виде списка. Алгебраическое выражение можно представить в виде последовательности элементарных двухместных операций вида:

```
< операнд 1 > < знак операции > < операнд 2 >.
```

Выражение

```
(a + b)*(c - (d/e)) + f
```

будет вычисляться в следующем порядке:

```
a + b; d/e; c – (d/e);
(a + b)*(c – d/e); (a + b)*(c – d/e) + f.
```

При представлении выражения в виде разветвленного списка каждая тройка «операнд-знак-операнд» представляется в виде списка, причем, в качестве операндов могут выступать как атомы – переменные или константы, так и подсписки такого же вида. На рис. 17.6 представлена схема списка алгебраического выражения.

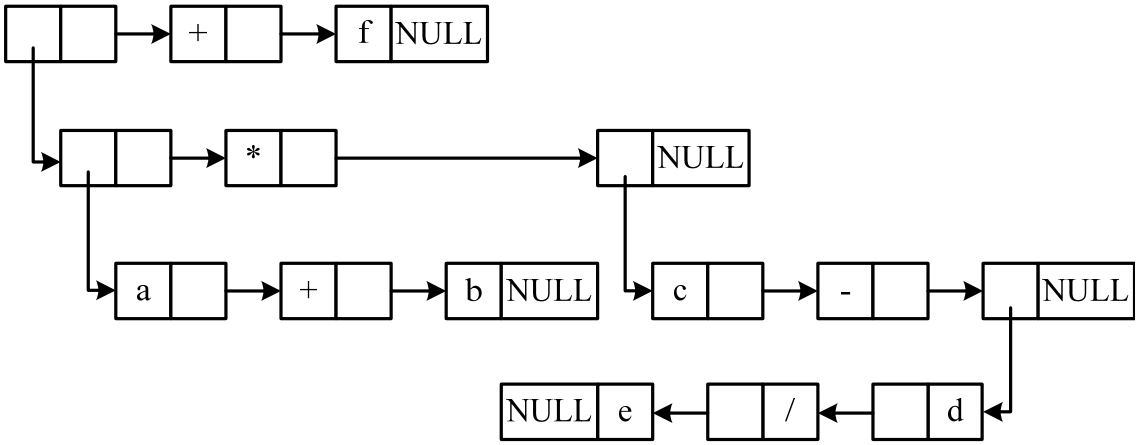


Рис. 17.6. Схема вычисления выражения на основе списка

Глубина этого списка равна 4, длина 3. Скобочное представление данного выражения будет иметь вид:

```
(((a,+,b),*,(c,-,(d,/,e)),+,f).
```

17.5. Практические задания

Создать интерфейс работы со списком на основе рассмотренных функций. Добавить функции: `CopyList` – копирование списков, `AppendList` – добавление всех элементов одного списка в конец другого списка, `SortList` – функцию сортировки списка. Сравнить результат с сортировкой массива. Добавить функции по вариантам.

1. `DeleteKFirst(int k)` – удаление `k` первых элементов списка.
2. `DeleteEveryM (int m)` – удаление каждого `m`-го элемента списка.
3. `DeleteDouble` – удаление повторяющихся (имеющих одинаковые поля или одно из полей) элементов.
4. `DeleteKLast(int k)` –удаление `k` последних элементов списка.
5. `FindMin` – поиск минимального элемента списка по одному из выбранных полей.

Глава 18. СТЕКИ И ОЧЕРЕДИ

18.1. Стек

Стек – одномерная структура данных (частный случай связного списка), в которой размещение новых элементов и удаление существующих производится с одного конца, называемого вершиной (top). В англоязычной литературе для обозначения стеков используется аббревиатура LIFO (Last In First Out: последний вошел – первый вышел).

Начало последовательности называется дном стека, конец последовательности, в который производится добавление элементов и их исключение – вершиной стека. Переменная, которая указывает на последний элемент последовательности в вершине стека – указатель стека. Таким образом, указатель стека sp (stack pointer) содержит в любой момент времени индекс (адрес) текущего элемента, который является единственным элементом стека, доступным в данный момент времени для обработки (рис. 18.1).

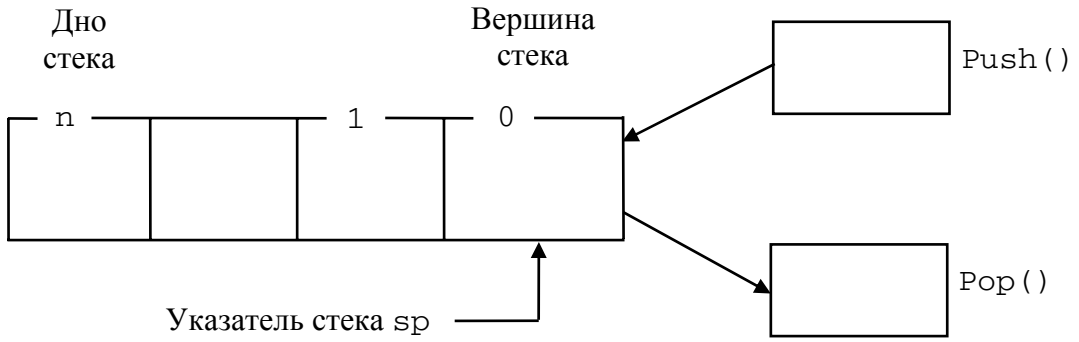


Рис. 18.1. Стек

Примеры стека: винтовочный патронный магазин, книги, сложенные в стопку, или стопка тарелок. Во всех этих моделях взять можно только верхний предмет, а добавить новый объект можно, только положив его на верхний.

При манипуляциях со стеком основными функциями являются:

- добавление в стек – `Push`;
- извлечение из стека – `Pop`;
- чтение элемента с вершины стека (без извлечения) – `Peek`;
- очистка стека – `Clear`.

Стеки могут быть реализованы статически – на основе массивов и динамически – на основе списков.

18.2. Реализация стеков на основе массивов

Стек с максимальным объемом `max_size` элементов можно реализовать в виде массива, например `STACK[max_size]`, и индекса последнего заполненного элемента массива `top`. Таким образом, значение `top = -1`, соответствует пустому стеку:

```
const int max_size = 100;      //размер стека
int STACK[max_size];           //стек
int top = -1;                  //вершина стека
```

Протестировать стек на наличие в нем элементов можно с помощью операции `IsStackEmpty()`, которая сводится к проверке значения индекса `top`:

```
//----------------------------- проверка на пустоту
bool IsStackEmpty(int top)
{    return (top < 0);  }
```

Если значение `top` превосходит `max_size`, то стек переполняется:

```
//----------------------------- проверка на переполнение
bool IsStackFull(int top)
{    return (top >= max_size - 1);  }
```

Операции добавления и извлечения элемента требуют предварительной проверки возможности выполнения операций. Операция извлечения элемента состоит в выборке значения, на которое указывает указатель стека, и модификации указателя стека (в направлении, обратном при включении). После выборки слот, в котором размещался выбранный элемент, считается свободным. При добавлении элемента в стек указатель модифицируется таким образом, чтобы он указывал на следующий свободный элемент. Затем элемент записывается на место, определяемое указателем стека:

```
//---------------------------- извлечь элемент из стека
int Pop (int &top, int STACK[])
{   int x = 0;
    if (!isStackEmpty(top)) x = STACK[top--];
    return x;
}
//---------------------------- добавить элемент в стек
bool Push (int STACK[], int &top, int NewItem)
{   bool x = true
    if (x = !isStackFull()) STACK [++top] = NewItem;
    return x;
}
```

Любая из рассмотренных операций имеет сложность $O(1)$. Основной недостаток данной реализации стека – статичность: до использования массива необходимо знать его максимальный размер, чтобы распределить под него оперативную память. Данный недостаток не является недостатком стека. Это результат выбора реализации стека на базе массива.

18.3. Реализация стека на основе динамического массива и списка

Можно также организовать стек на базе массива, который будет увеличиваться и уменьшаться динамически (например, при заполнении половины стека, его размер увеличивается на фиксированную величину). Например, стек можно представить следующей структурой:

```
struct Stack
{
 int    Top;         //вершина стека
 int    Size;        //размер стека
 void** Data;        //данные стека
};
```

Чтобы стек мог рационально увеличиваться и уменьшаться, можно отдать предпочтение его организации на основе связного списка. Реализацию списков можно рассматривать как реализацию стеков, поскольку стеки с их операторами являются частными случаями списков с операторами, выполняемыми над списками. Для этого необходимо представить стек в виде однонаправленного списка. В этом случае операторы `push` и `pop` будут работать только с ячейкой заголовка и первой ячейкой списка.

Представление элементов односвязного списка организовано традиционно:

```
struct STACK
{
     void* Data;   //данные элемента стека
     STACK *next;  //указатель на следующий
};
```

Для добавления элемента (операция `Push()`) создается новый элемент и добавляется в начало списка:

```
//----------------------------- добавить в стек
void Push(STACK **ppStack, void* nItem)
{
  STACK *pNewItem;
  pNewItem = new STACK;
     //запрашиваем память под структуру для стека
  pNewItem->Data = nItem;
     //заполняем поля структуры
  pNewItem->next = *ppStack;
     //устанавливаем новый указатель на вершину стека
  *ppStack = pNewItem;
}
```

Для извлечения элемента, проверяется возможность выполнения операции (если стек не пустой), извлекается элемент из стека (удаление из начала списка):

```
//----------------------------- извлечь из стека
void* Pop(STACK **ppStack, int *nError)
{
  STACK *pOldData = *ppStack;
     //запомнить старый адрес вершины
  void* nOldData = NULL;
  if(*ppStack)
  {
     nOldData = pOldData->Data;
     //если стек не пустой, извлечь
     *ppStack = (*ppStack)->next;
     delete pOldData;
     //и освободить память
  }
  else *nError = 1;
     //в противном случае выставить ошибку
  return nOldData;
}
```

Обратите внимание, что в функциях `Push()` и `Pop()` используются двойные ссылки. Благодаря этому, каждая из функций может возвращать в качестве результата своей работы указатель на новый элемент `STACK` (используется передача параметров по адресу). Чтобы можно было вернуть значение указателя из функции, необходимо использовать именно двойные ссылки. Если в стеке есть хоть один элемент, то функции возвращают `nError = 0`, если стек пуст – `nError = 1`.

По аналогии выполняется операция чтения элемента с вершины без извлечения `Peek()`:

```
//---------------------------- прочитать не извлекая
void* Peek(STACK **ppStack, int *nError)
{
   if(*ppStack)     //если стек не пустой
   {
      *nError = 0;
      return(*ppStack)->Data; //прочитать элемент
   }
   else
   {
      *nError = 1;  //если стек пуст, установить ошибку
      return NULL;
   }
}
```

Очистка стека `Clear()` заключается в последовательном удалении элементов стека, начиная с заголовка:

```
//---------------------------- очистить стек
void Clear(STACK **ppStack)
{
   STACK *pDelItem = *ppStack;
   while((*ppStack) != NULL        //пока стек не пустой
   {
      pDelItem = *ppStack;
      *ppStack = (*ppStack)->next; //перейти к следующему
      delete pDelItem;             //удалить элемент
   }
}
```

Итак, реализация стека на базе массива использует объем памяти, необходимый для размещения максимального числа элементов, которые может вместить стек в процессе вычислений; реализация на базе списка использует объем памяти пропорционально количеству элементов, но при этом всегда расходует дополнительную память для одной связи на каждый элемент, а также дополнительное время на распределение памяти при каждой операции «добавить» и освобождение памяти при каждой операции «извлечь». Если требуется стек больших размеров, который обычно заполняется практически полностью, то предпочтение надо отдать реализации на базе массива. Если же размер стека варьируется в широких пределах и присутствуют другие структуры данных, которым требуется память, не используемая во время, когда в стеке находится несколько элементов, предпочтение следует отдать реализации на базе связного списка.

Несмотря на то, что реализация стека на базе связного списка создает впечатление, что стек может увеличиваться неограниченно в практических условиях, такой стек невозможен: рано или поздно, когда запрос на выделение еще некоторого объема памяти не сможет быть удовлетворен, оператор `new` сгенерирует исключение.

18.4. Стеки в вычислительных системах

Исключительная популярность стека в программировании объясняется тем, что при заданной последовательности записи элементов в стек исключение их происходит в обратном порядке. А именно, такая последовательность действий соответствует таким понятиям, как вложенность вызовов функций, вложенность определений конструкций языка и т.д. Следовательно, там, где речь идет о вложенности процессов, структур, определений, везде механизмом реализации такой вложенности является стек.

Большинство трансляторов используют стек (в том смысле, что генерируют команды, работающие со стеком) для реализации механизма вызова функций.

В архитектуре практически всех компьютеров используется аппаратный стек. Он представляет собой обычную область внутренней (оперативной) памяти компьютера, с которой работают специальные регистры – указатель стека SS:SP. С его помощью процессор может выполнять операции `Push` и `Pop` по сохранению и восстановлению из стека байтов и машинных слов различной размерности. Единственным отличием аппаратного стека является его заполнение, от старших адресов к младшим. Пара команд – `Push` и `Pop` – обеспечивает использование аппаратного стека для программного решения других задач.

18.5. Очередь

Очередь – одномерная структура данных, для которой загрузка или извлечение элементов осуществляется с помощью указателей начала (head) и конца (tail) очереди в соответствии с правилом FIFO (First In First Out: первым пришел – первым ушел), другими словами, включение производится с одного, а исключение – с другого конца (рис. 18.2).

При манипуляциях с очередью основными функциями являются:

- добавление элемента в конец очереди – `Enqueue`;
- извлечение элемента из начала очереди – `Dequeue`;
- чтение элемента из начала очереди (без извлечения) – `Peek`;
- очистка очереди – `Clear`.

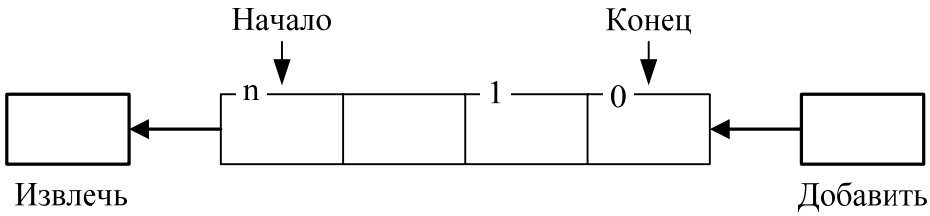


Рис. 18.2. Очередь

Также, как и стеки, очереди могут быть реализованы на основе массивов и на основе списков.

18.6. Реализация очереди на основе массива

Очередь с максимальным объемом `max_size` элементов можно реализовать в виде массива – `QUEUE[max_size]`. Для организации очереди необходимы две индексные переменные `Head`, указывающие на первый элемент очереди, и `Tail`, указывающая позицию, в которую будет добавляться элемент.

```
const int max_size = 100;      //размер очереди
int QUEUE[max_size];           //циклическая очередь
int Head;                      //индекс начала очереди
int Tail;                      //индекс конца очереди
```

Чтобы извлечь элемент, он удаляется из начала (`Head`) очереди, после чего индекс `Head` увеличивается на единицу. Чтобы добавить элемент, он добавляется в конец (`Tail`) очереди, а индекс `Tail` увеличивается на единицу. По мере добавления элементов в очередь ее конец будет продвигаться к концу массива, то же самое будет происходить с началом при их извлечении. Выход из создавшегося положения – зациклить очередь, то есть считать, что за последним элементом массива следует опять первый. Подобный способ организации очереди на массиве еще иногда называют циклической (или кольцевой) очередью. Элементы очереди будут расположены в последовательных слотах `QUEUE[Head]`, `QUEUE[Head+1]` … `QUEUE[Tail-1]`, которые циклически замкнуты. Циклическая очередь устроена так, что при достижении конца массива осуществляется переход на его начало, то есть слот 1 следует сразу же после слота с номером `max_size`. Определение размера очереди состоит в вычислении разности индексов с учетом циклической природы очереди.

Первоначально `Head = Tail = 0`. Равенство этих двух индексов (при любом их значении) `Head == Tail` является признаком пустой очереди. Тогда, при попытке удалить из нее элемент происходит ошибка опустошения.

Если в процессе работы с циклической очередью число операций добавления превышает число операций исключения, то может возникнуть ситуация, в которой индекс конца «догонит» индекс начала. Это ситуация заполненной очереди, но если индексы сравняются, то ситуация будет неотличима от ситуации пустой очереди. Для различения этих двух ситуаций к циклической очереди предъявляется требование, чтобы между индексами конца и начала оставался зазор из свободных элементов. Когда этот зазор сокращается до одного элемента, очередь считается заполненной. Таким образом, если `Head == (Tail + 1) % max_size)`, то очередь заполнена, и попытка добавить в очередь элемент приводит к переполнению. Рассмотрим функции работы с очередью: очистки `Clear()` и добавления `Enqueue()`:

```
//------------------------ очистить очередь
void Clear()
{  Head = Tail = 0;  }   //обнулить индексы начала и конца
//------------------------ добавить в конец очереди
int Enqueue (int NewItem)
{
   int next;
   if ((next = (Tail + 1) % max_size) == Head)
                     //если переполнение очереди
        return 0;    //ошибка
   QUEUE[Tail] = NewItem;
   //добавить элемент в конец очереди
        Tail = next;
   //индекс конца очереди переместить на следующий
        return 1;
}
```

Функция извлечения `Dequeue()` будет выглядеть следующим образом:

```
//---------------------------- извлечь из начала очереди
int Dequeue ()
{
   int Item;
   if (Head == Tail)  //если очередь пуста
       return 0;      //ошибка
   else
   {Item = QUEUE[Head++];  //извлечь элемент из начала
   Head %= max_size;  //по достижении Head == max_size
                      //индекс начала очереди сбросить в 0
   return Item;       //вернуть  извлеченный  элемент
   }
}
```

Следующий вариант реализации очереди во многом похож на предыдущий, за исключением того, что очередь описывается структурой и массив для хранения элементов задается динамически в соответствии с запрашиваемым максимальным (не фактическим) размером `Size`:

```
struct QUEUE              //циклическая очередь
{
  int    Head;            //индекс начала очереди
  int    Tail;            //индекс конца очереди
  int    Size;            //размер очереди
  int*   Data;            //данные очереди
  QUEUE(int size)
    {
    Head = Tail = 0;      //очередь пуста
    Data = new int[Size = size+1];
                          //выделить память под данные
    };
  bool isFull()
    {
    return (Head%Size == (Tail+1)%Size); //переполнение
    };
  bool isEmpty()
    {
    return (Head%Size == Tail%Size);};   //очередь пуста
    };
}
```

Рассмотрим последовательность функций для манипуляции с данными очереди, лежащих в основе ее абстрактного представления. Конструктор очереди:

```
//----------------- выделить ресурс для очереди размера n
Queue CreateQueue(int n)
{  return *(new QUEUE (n)); };
```

Добавление нового элемента в очередь:

```
//------------------- добавить NewItem в конец очереди q
bool  Enqueue(QUEUE& q, int NewItem)
{
   bool x = true;
   if (x = !q.isFull())  //если нет переполнения очереди
   {q.Data[q.Tail] = NewItem;
               //добавить новый элемент в конец очереди
   q.Tail=(q.Tail+1)%q.Size;  //индекс конца очереди
               //увеличить и скорректировать (циклическая)
   }
return x;
};
```

Функция извлечения элемента:

```
//----------------- извлечь элемент из начала очереди
int Dequeue(QUEUE& q)
{
int x = 0;
if (!q.isEmpty())   //если очередь не пуста
  {
  x = q.Data[q.Head];
          //считать данные с начала очереди
  q.Head=(q.Head+1)%q.Size;
          //скорректировать индекс начала
  }
return x;           //вернуть элемент
}
```

Функция чтения элемента без извлечения:

```
//---------------------- прочитать  из начала очереди
int Peek(const QUEUE& q)
{
int x = 0;
if (!q.isEmpty())   //если очередь не пуста
  x = q.Data[q.Head];
                    //прочитать элемент из начала очереди
return x;           //вернуть элемент
}
```

Функция очистки очереди сводится к записи одного и того же (не обязательно начального) значения в оба индекса:

```
//---------------------------- очистить очередь
void  ClearQueue(QUEUE& q)
{  q.Tail = q.Head = 0;  //обнулить индексы очереди
}
```

Функция удаления очереди сводится к освобождению памяти, занимаемой массивом, и установке индексов в ноль:

```
//----------------------- освободить ресурсы очереди
void ReleaseQueue(QUEUE & q)
{
  delete[] q.Data;       //вернуть  память
  q.Size = 1;
  q.Head = q.Tail = 0;   //сбросить индексы
}
```

Для очереди операции извлечения и добавления имеют постоянное время выполнения $O(1)$.

18.7. Реализация очереди на основе списка

Как и для стеков, любая реализация списков допустима для представления очередей.

```
struct QUEUE        //очередь
{    void* Data;    //данные элемента очереди
     QUEUE *next;   //указатель на следующий
};
```

Каждая операция «извлечь» уменьшает размер очереди на 1, а каждая операция «добавить» увеличивает размер очереди на 1. Новые элементы добавляются в конец очереди (списка). Поэтому в функции добавления `Enqueue()`, если очередь содержит элементы, необходимо выделить память под новый элемент (`nItem`), получить указатель на последний элемент (`pQ`) и добавить в очередь (список) новый элемент, связывая его с элементом, на который ссылается указатель на последний (`tail`). Если очередь пуста, то необходимо добавить элемент в пустую очередь (очередь содержит единственный элемент и установление связей с другими элементами не требуется). Ниже приведена рассмотренная функция:

```
//-------------------- добавить новый элемент  в очередь
void Enqueue (QUEUE **ppQueue, void* nItem)
{
  if (*ppQueue)                 //если очередь не пустая
  {
     QUEUE *pNewItem = new QUEUE;  //выделить память
     QUEUE *pQ;
     pQ = (*ppQueue);
     while(pQ->next)            //перейти в конец очереди
           pQ = pQ->next;
     pNewItem->Data = nItem;    //заполнить поля структуры
     pNewItem->next = NULL;
     pQ->next = pNewItem;       //добавить в конец
  }
  else
  {
     (*ppQueue) = new QUEUE;    //выделить память
     (*ppQueue)->Data = nItem; //заполнить поля структуры
     (*ppQueue)->next = NULL;
  }
}
```

Для извлечения элемента из очереди (списка) – `Dequeue()`, при условии, что очередь не пуста, извлекается и удаляется элемент из начала

очереди (списка) так же, как это делалось в случае стека. Если очередь пуста, будет возвращаться указатель NULL и выставляться ошибка *nError = 1. Реализация приведена ниже:

```
//-------------------------- извлечь элемент из очереди
void* Dequeue (QUEUE **ppQueue, int *nError)
{
   QUEUE *pOldData = *ppQueue;
          //запомнить старый адрес начала
   void* nOldData = NULL;
   if(*ppQueue)
          //если очередь не пуста
   {
     nOldData = pOldData ->Data;
          //извлечь данные
     *ppQueue = (*ppQueue)->next;
          //начало установить на следующий
     delete pOldData;
          //удалить старый начальный элемент очереди
   }
   else *nError = 1;      //выставить ошибку
   return nOldData;       //вернуть результат
}
```

Функция Clear() идентична аналогичной функции для стека с организацией на базе связного списка (см. 8.3).

```
//---------------------------- очистить очередь
void Clear(QUEUE **ppQueue)
{
     QUEUE *pDelItem = *ppQueue;
     while((*ppQueue) != NULL)
          //последовательно до конца очереди
     {
     pDelItem = *ppQueue;
     *ppQueue = (*ppQueue)->next;
     delete pDelItem;
          //удалить поэлементно
     }
}
```

Очевидным преимуществом в случае представления очереди на базе связного списка является то, что оперативная память используется пропорционально числу элементов в структуре данных. Это происходит за счет дополнительного расхода памяти на связи (между элементами) и дополнительного расхода времени на распределение и освобождение памяти для каждой операции.

18.8. Очереди в вычислительных системах

У очередей имеется множество применений в вычислительных системах. Если у компьютера только один процессор, то в один и тот же момент времени может быть обслужен только один процесс. Запросы других процессоров помещаются в очередь. Каждый запрос постепенно продвигается в очереди вперед по мере того, как происходит обслуживание процессов. Запрос в начале очереди является очередным кандидатом на обслуживание.

Очереди также применяются для поддержания процесса буферизации потоков данных, выводимых на печать. Выходные данные в многопользовательской среде могут записываться в буферизированный файл на диске, где они ожидают в очереди, пока принтер не станет для них доступным.

Информационные пакеты ожидают своей очереди в компьютерных сетях. Пакет может поступать на узел сети в любой момент времени, а затем он должен быть отправлен к следующему узлу сети по направлению своего конечного пункта назначения. Узел маршрутизации направляет в каждый момент времени один пакет; поэтому пакеты помещаются в очередь до тех пор, пока программа маршрутизации не обработает их.

Файл-сервер в компьютерной сети обрабатывает запросы многих клиентов сети. Серверы имеют ограниченную пропускную способность обслуживания запросов клиентов. Когда такая пропускная способность превышена, запрос клиента ожидает своей очереди.

18.9. Очереди с приоритетами

В реальных задачах иногда возникает необходимость в формировании очередей, отличных от порядка обслуживания FIFO. Порядок выборки элементов из таких очередей определяется приоритетами элементов. Приоритет в общем случае может быть представлен числовым значением, которое вычисляется либо на основании значений каких-либо полей элемента, либо на основании внешних факторов.

```
struct PR_QUEUE1          //приоритетная очередь
{
  int    Head;            //начало очереди
  int    Tail;            //конец очереди
  int    Size;            //размер очереди (максимальное
                          //количество элементов + 1)
  int*   Data;            //данные очереди
```

```
int*   Prioritet;   //приоритеты очереди
};
//.............................
struct PR_QUEUE2                //приоритетная очередь
{
void*       Data;               //данные очереди
int         priority_value;     //приоритеты очереди
PR_QUEUE    *next;
};
```

Так, и стеки, и LIFO-очереди могут трактоваться как приоритетные очереди, в которых приоритет элемента зависит от времени его включения в очередь. Приоритетная очередь – это абстрактный тип данных, предназначенный для представления *взвешенных множеств* (*куч*). Множество называется взвешенным, если каждому его элементу однозначно соответствует число, называемое ключом или весом. При выборке элемента всякий раз выбирается элемент с наибольшим приоритетом, т.е. «первым включается – с высшим приоритетом исключается». Основными операциями над приоритетной очередью являются следующие операции:

- вставить новый элемент со своим ключом;
- найти с максимальным (минимальным) приоритетом. Если элементов несколько, то находится один из них. Найденный элемент не удаляется из очереди;
- удалить элемент с максимальным (минимальным) приоритетом. Если элементов несколько, то удаляется один из них;
- увеличить (уменьшить) приоритет указанного элемента на заданное положительное число.

Приоритетная очередь естественным образом используется в таких задачах, как сортировка элементов массива, поиск во взвешенном неориентированном графе минимального остовного дерева, поиск кратчайших путей от заданной вершины взвешенного графа до его остальных вершин и во многих других.

18.10. Деки

Дек – особый вид очереди (от англ. *deq* – double ended queue, т.е очередь с двумя концами), в котором как включение, так и исключение элементов может осуществляться с любого из двух концов. Частные случаи дека – дек с ограниченным входом и дек с ограниченным

выходом. Логическая и физическая структуры дека аналогичны логической и физической структуре циклической FIFO-очереди. Однако, применительно к деку целесообразно говорить не о начале и конце, а о левом и правом конце. Над деком разрешены операции:

- включение элемента справа;
- включение элемента слева;
- исключение элемента справа;
- исключение элемента слева;
- определение размера;
- очистка.

Задачи, решение которых требует использования структуры дека, встречаются в вычислительной технике и программировании гораздо реже, чем задачи, реализуемые на структуре стека или очереди. Как правило, вся организация дека выполняется программистом без каких-либо специальных средств системной поддержки.

18.11. Практические задания

Для организации абстрактных типов данных использовать структуры. Создать интерфейсные функции для работы с очередью, приоритетной очередью, стеком и деком. Добавить функцию в соответствии с вариантом.

1. Разработать функцию сортировки дека и функцию, которая удаляет из дека первый отрицательный элемент, если такой есть.

2. Создать функцию, которая подсчитывает количество элементов стека, у которых равные «соседи», и функцию, которая определяет, есть ли в стеке хотя бы один элемент, равный следующему за ним.

3. Написать функцию, которая в очереди переставляет в обратном порядке все элементы между первым и последним вхождением элемента E, если E входит в очередь не менее двух раз.

4. Разработать функции работы с приоритетной очередью. Постановка запросов в очередь выполняется по приоритету, снятие – подряд из старших адресов (конец очереди). Приоритет: минимальное значение числового параметра, при совпадении параметров – LIFO.

5. Реализовать функцию `translate(infix, postfix)`, которая переводит выражение, записанное в обычной (инфиксной) форме в текстовом файле `infix`, в постфиксную форму и в таком виде записывает его в текстовый файл `postfix`. Использовать следующий алгоритм перевода. В стек записывается открывающая скобка,

и выражение просматривается слева направо. Если встречается операнд (число или переменная), то он сразу переносится в файл `postfix`. Если встречается открывающая скобка, то она заносится в стек, а если встречается закрывающая скобка, то из стека извлекаются находящиеся там знаки операций до ближайшей открывающей скобки, которая также удаляется из стека, и все эти знаки (в порядке их извлечения) записываются в файл `postfix`. Когда встречается знак операции, то из стека извлекаются (до ближайшей скобки, которая сохраняется в стеке) знаки операций, старшинство которых больше или равно старшинству данной операции, и они записываются в файл `postfix`, после чего рассматриваемый знак заносится в стек. В заключение выполняются такие действия, как если бы встретилась закрывающая скобка.

Глава 19. ДЕРЕВЬЯ. БИНАРНЫЕ ДЕРЕВЬЯ

19.1. Деревья

В отличие от структур, очередей, стеков, деревья представляют собой иерархическую структуру некой совокупности элементов. Примерами деревьев могут служить генеалогические, организационные диаграммы, оглавление книги. Деревья используются при анализе электрических цепей, для представления структур математических формул, для организации информации в системах управления базами данных и представления синтаксических структур в компиляторах программ.

Название «дерево» проистекает из логической эквивалентности древовидной структуры абстрактному дереву в теории графов.

Дерево – это граф, который характеризуется следующими свойствами:

- существует единственный элемент (*узел* или *вершина*), на который не ссылается никакой другой элемент и который называется *корнем*;
- начиная с корня и следуя по определенной цепочке указателей, содержащихся в элементах, можно осуществить доступ к любому элементу структуры;
- на каждый элемент, кроме корня, имеется единственная ссылка, т.е. каждый элемент адресуется единственным указателем.

Таким образом, дерево представляет собой либо отдельную вершину, либо вершину, имеющую ограниченное число указателей – *ветвей*. Нижележащие деревья для текущей вершины называются *поддеревьями*, а их вершины – *потомками* (дочерними вершинами, сыновьями). По отношению к потомкам текущая вершина называется *предком* (рис. 19.1). Те узлы, которые не ссылаются ни на какие другие узлы дерева, называются *листьями* (или терминальными вершинами). Узел, не являющийся листом или корнем, считается промежуточным или *узлом ветвления* (нетерминальной или внутренней вершиной).

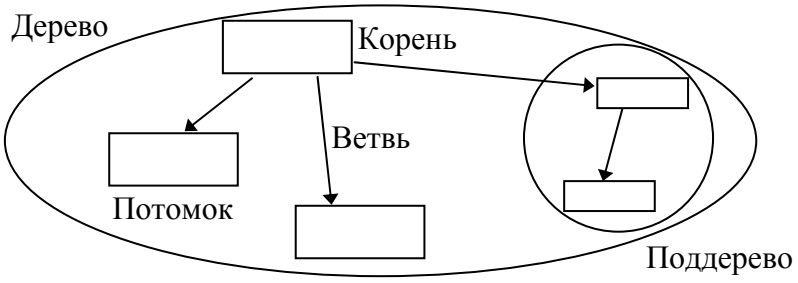


Рис. 19.1. Представление дерева

Вершину дерева можно определить таким образом:

```
struct    TREE
{
int  val;                          //значение элемента
TREE *child[4];                    //указатели на потомков
};
```

Из определения элемента дерева следует, что оно имеет ограниченное число указателей на подобные элементы. Как и во всех динамических структурах данных, характер связей между элементами определяется функциями, которые их устанавливают. Рекурсивное определение дерева требует и рекурсивного характера функций, работающих со связями. Простейшей функцией является функция обхода всех вершин дерева:

```
void ScanTree  (TREE *pTree)      //обход дерева
{ int     i;
  if (pTree == NULL)
          return;                 //следующей вершины нет
  for (i=0; i<4; i++)             //рекурсивный обход
     ScanTree(pTree ->child[i]); //потомков с передачей
}                                 //указателей на них
```

Само дерево обычно задается в программе указателем на его корень. Часто обход дерева используется для получения информации, которая затем возвращается через результат рекурсивной функции, например, функция определения минимальной длины ветвей дерева:

```
int  MinDepth(TREE *pTree)
{  int    i, min, nn;
   if (pTree == NULL)    return 0;
                                  //следующей вершины нет
   for (min = MinDepth(pTree->child[0], i=1; i<4; i++)
   {
      nn = MinDepth(pTree ->child[i]);
                                  //обход потомков
      if (nn > max) max = nn;
   }                              //возвращается глубина
   return min + 1;                //с учетом текущей вершины
}
```

Другой распространенной функцией является включение нового элемента в дерево. Здесь есть проблема, общая для всех деревьев и рекурсивных алгоритмов. Войдя в поддерево, невозможно производить какие-либо действия для вершин, расположенных на том же уровне,

но в других поддеревьях. Поэтому используется функция включения с просмотром дерева на заданную глубину, а сама глубина просмотра, в свою очередь, задается равной длине минимальной ветви дерева:

```
int Insert(TREE *pTree, int newItem, int d)
//pTree - указатель на текущую вершину
//d  - текущая глубина включения
{if (d == 0) return 0;           //ниже не просматривать
   for (int i=0; i<4; i++)
   if (pTree->child[i] == NULL)
      {
      TREE *pn=new TREE;
      pTree->child[i] = pTree;
      pTree->val = newItem;
      for (i=0; i<4; i++) pTree->child[i] = NULL;
      return 1;
               //результат логический - вершина включена
      }
   else
      if (Insert(pTree->child[i], v , d-1)) return 1;
   return 0;
}
int _tmain(int argc, _TCHAR* argv[]).
{
    tree PH={1,{NULL,NULL,NULL,NULL}};
                                    //пример вызова функции
    Insert(&PH, 5, MinDepth(&PH));
}
```

Из последнего примера видно, что количество просматриваемых вершин от уровня к уровню растет в геометрической прогрессии. Таким образом, деревья можно эффективно использовать для поиска данных.

19.2. Бинарные деревья

Бинарным (или *двоичным*) *деревом* называется дерево, каждая вершина которого имеет не более двух потомков. При определении бинарного дерева поиска на данные, хранимые в вершинах дерева, вводится следующее правило упорядочения: значения вершин левого поддерева всегда меньше, а значения вершин правого поддерева – больше значения в самой вершине (рис. 19.2). Это свойство позволяет применить в дереве алгоритм двоичного поиска. Действительно, каждое сравнение искомого значения и значения в вершине двоичного дерева позволяет выбрать для следующего шага правое или левое поддерево.

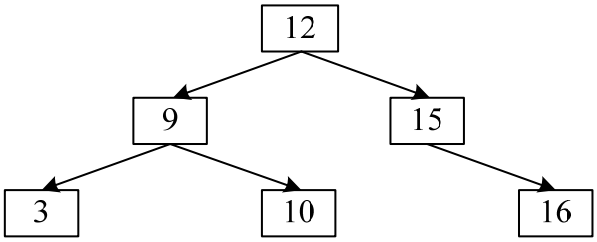


Рис. 19.2. Бинарное дерево

Вид дерева бинарного поиска (или BST – Binary Search Tree), соответствующий набору меняющихся данных, зависит от последовательности, в которой значения помещаются в дерево.

Узел бинарного дерева можно представить следующей структурой:

```
struct Node       //узел бинарного дерева
{ Node* Parent;   //указатель на родителя
  Node* Left;     //указатель на левое поддерево
  Node* Right;    //указатель на правое поддерево
  void* Data;     //данные
};
```

Нумерация вершин в деревьях. Способы обхода дерева

В любой структуре данных имеется естественная нумерация элементов по их расположению в ней. Так, каждый элемент списка или массива имеет свой логический номер в линейной последовательности, соответствующей его размещению в памяти (массив) или направлению последовательного обхода (списки). В деревьях обход вершин возможен с использованием рекурсии, поэтому и логическая нумерация вершин производится согласно последовательности их рекурсивного обхода. Рекурсивная функция в этом случае получает текущий счетчик вершин, который она увеличивает на 1 при обходе текущей вершины и который она передает и получает обратно из поддеревьев.

Бинарное дерево можно обходить тремя основными способами: *нисходящим*, *смешанным* и *восходящим* (возможны также обход слева-направо, справа-налево). Принятые выше названия методов обхода связаны с временем обработки корневой вершины: до того, как обработаны оба ее поддерева (Preorder); после того, как обработано левое поддерево, но до того, как обработано правое (Inorder); после того как, обработаны оба поддерева (Postorder). Используемые в переводе названия методов отражают направление обхода в дереве: от корневой вершины вниз к листьям – нисходящий обход; от листьев вверх к корню –

восходящий обход и смешанный обход – от самого левого листа дерева через корень к самому правому листу.

Определим следующую структуру узла с конструктором, операциями перехода к следующему, предыдущему, минимальному (в соответствии со свойством дерева – крайне левый лист), максимальному элементу (крайне правый лист) и с функциями обхода бинарного дерева:

```
struct Node                 //узел бинарного дерева
{
      Node* Parent;         //указатель на родителя
      Node* Left;           //указатель на левое поддерево
      Node* Right;          //указатель на правое поддерево
      void* Data;           //данные
      Node(Node* p, Node* l, Node* r, void* d)
                            //конструктор
      { Parent = p; Left  =  l; Right =  r; Data  =  d; }
  Node* Next();             //следующий
  Node* Prev();             //предыдущий
  Node* Min();              //минимум в поддереве
  Node* Max();              //максимум в поддереве
  void DescendingScan(void (*f)(void* n));
                            //нисходящий обход
  void Scan(void (*f)(void* n));
                            //восходящий обход
  void MixedScan(void(*f)(void* n));
                            //смешаный обход
  int TreeH(int h = 0);
};
```

Нисходящий обход

Далее рассмотрен пример обхода вершин дерева слева-направо, сверху-вниз (рис. 19.3). Он заключается в посещении текущего узла, обходе левого поддерева, обходе правого поддерева:

```
//---------------------- нисходящий обход
void Node:: DescendingScan(void (*f)(void* n))
{
     f(this->Data);
             //обработка узла дерева
     std::cout<<std::endl;
     if (this->Left  != NULL) this->Left->Scan(f);
             //рекурсивный вызов для левого поддерева
     if (this->Right != NULL) this->Right->Scan(f);
             //рекурсивный вызов для правого поддерева
}
```

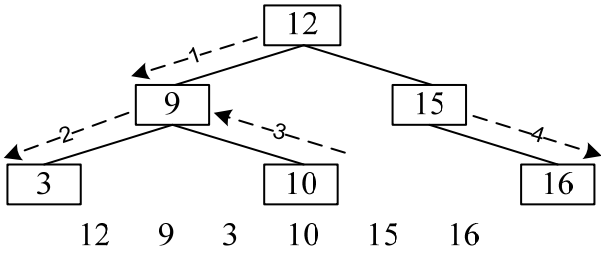


Рис. 19.3. Нисходящий обход дерева

Обход дерева можно реализовать без рекурсии с использованием стека. Алгоритм обхода бинарного дерева в соответствии с нисходящим способом может выглядеть следующим образом:

1) в качестве очередной вершины взять корень дерева, перейти к пункту 2;

2) произвести обработку очередной вершины в соответствии с требованиями задачи, перейти к пункту 3;

3а) если очередная вершина имеет обе ветви, то в качестве новой выбрать ту вершину, на которую ссылается левая ветвь, а вершину, на которую ссылается правая ветвь, занести в стек; перейти к пункту 2;

3б) если очередная вершина является конечной, то выбрать в качестве новой очередной вершины вершину из стека, если он не пуст, и перейти к пункту 2; если же стек пуст, то это означает, что обход всего дерева окончен, перейти к пункту 4;

3в) если очередная вершина имеет только одну ветвь, то в качестве очередной вершины выбрать ту вершину, на которую эта ветвь указывает, перейти к пункту 2;

4) конец алгоритма.

Восходящий обход

Он заключается в посещении при обходе левого узла поддерева, обходе правого поддерева, посещении текущего узла (рис. 19.4):

```
//------------------------ восходящий обход
void Node:: Scan(void (*f)(void* n))
{
     std::cout<<std::endl;
     if (this->Left  != NULL) this->Left->Scan(f);
                  //рекурсивный вызов для левого поддерева
     if (this->Right != NULL) this->Right->Scan(f);
                  //рекурсивный вызов для правого поддерева
     f(this->Data);
                  //обработка узла дерева
}
```

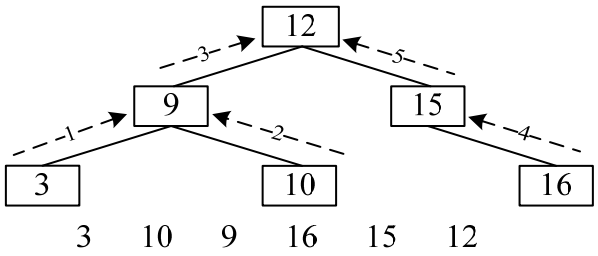


Рис. 19.4. Восходящий обход дерева

Трудность организации восходящего обхода с использованием стека заключается в том, что в отличие от нисходящего алгоритма в этом алгоритме каждая вершина запоминается в стеке дважды: первый раз – когда обходится левое поддерево, и второй раз – когда обходится правое поддерево. Таким образом, в алгоритме необходимо различать два вида стековых записей: 1-й означает, что в данный момент обходится левое поддерево; 2-й – что обходится правое, поэтому в стеке запоминается указатель на узел и признак (код-1 и код-2 соответственно). Алгоритм восходящего обхода можно представить следующим образом:

1) спуститься по левой ветви с запоминанием вершины в стеке как 1-й вид стековых записей;

2) если стек пуст, то перейти к пункту 5;

3) выбрать вершину из стека, если это первый вид стековых записей, то возвратить его в стек как 2-й вид стековых записей, перейти к правому «сыну», перейти к пункту1, иначе перейти к пункту 4;

4) обработать данные вершины и перейти к пункту 2;

5) конец алгоритма.

Смешанный обход

Рекурсивный смешанный обход описывается следующим образом: смешанный обход левого поддерева; обработка узла; смешанный обход правого поддерева (рис. 19.5):

```
//----------------------- смешанный обход
void Node::MixedScan(void (*f)(void* n))
  {
     std::cout<<std::endl;
     if (this->Left  != NULL) this->Left->Scan(f);
           // рекурсивный вызов для левого поддерева
      f(this->Data);
          //обработка узла дерева
     if (this->Right != NULL) this->Right->Scan(f);
          // рекурсивный вызов для правого поддерева
  }
```

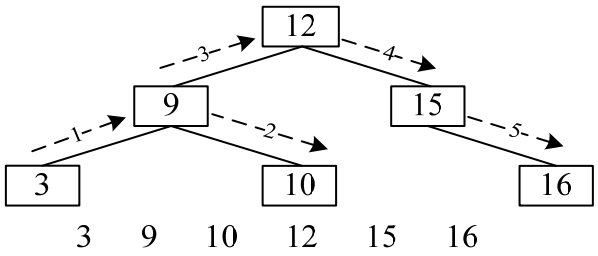


Рис. 19.5. Смешанный обход дерева

Аналогичный обход в порядке справа-налево дает в двоичном дереве последовательность в порядке убывания.

Смешанный обход с использованием стека можно описать следующим образом:

1) спуститься по левой ветви с запоминанием вершин в стеке;

2) если стек пуст, то перейти к пункту 5;

3) выбрать вершину из стека и обработать данные вершины;

4) если вершина имеет правого «сына», то перейти к нему; перейти к пункту 1.

5) конец алгоритма.

Для обхода всего дерева требуется $O(n)$ времени.

Операции поиска

Основные операции при работе с бинарными деревьями поиска связаны с поиском определенного значения, поиском наименьшего, наибольшего элемента дерева, предшествующего и последующего элементов для данного.

Выполнение операции поиска основано на том, что, находясь на определенной вершине, можно всегда однозначно указать, в каком из поддеревьев находится искомое значение, так как согласно свойству бинарного дерева поиска все значения узлов в левом поддереве не больше, а в правом не меньше значения в корне. Таким образом, функция поиска заданного значения имеет следующий вид:

```
//-------------------------------------поиск заданного
Node* Search(void* Item, Node* pTree)
    { Node* x = pTree;
      if (x != NULL)              //если дерево не пустое
         {  if (Item < pTree->Data)
            x = Search(Item, pTree->Left);
            else    if (Item > pTree->Data))
                    x = Search(Item, pTree->Right);
         }
    return x;
    }
```

В случае, если искомое значение отсутствует, в дереве будет возвращен указатель на корень.

Чтобы достичь наименьшего (наибольшего) значения в дереве поиска, надо двигаться по левым (или соответсвенно правым) ветвям дерева до тех пор, пока это возможно:

```
//------------------------------ поиск минимального
Node* Min(Node* pTree)
    { Node* x = pTree;
      if (x->Left != NULL) x = Min(x->Left);
      return x;
    }
//------------------------------ поиск максимального
Node* Max(Node* pTree)
    { Node* x = pTree;
      if (x->Right != NULL) x = Max(x->Right);
      return x;
    }
```

Поиск очередного и предшествующего узла – задача более сложная, чем предшествующие. Задача решается с использованием исключительно знаний о структуре дерева. Если правое поддерево текущего непустое, то следующий за текущим – минимальный элемент правого поддерева. Если правое поддерево текущего пустое и существует, то следующий – наименьший предок текущего, левый наследник которого также является предком текущего:

```
//---------------------------------- поиск следующего
Node* Next(Node* pTree)
    { Node* next = pTree, *x = pTree;
      if (next->Right != NULL) next = Min(next->Right);
      else
      {
      next = pTree->Parent;
      while (next != NULL && x == next->Right)
          {
          x = next;
          next = next->Parent;
          }
      }
      return next;
    }
```

Функция поиска узла, предшествующего данному, симметрична функции поиска последующего узла:

```
//---------------------------------- поиск предыдущей
Node* Prev(Node* pTree)
    { Node* prev = pTree, *x = pTree;
      if (prev->Left != NULL) prev = Max(prev->Left);
      else
      {
      prev = pTree ->Parent;
      while (prev != NULL && x == prev->Left)
          {
          x = prev;
          prev = prev->Parent;
          }
      }
      return prev;
     }
```

Включение вершины в бинарное дерево

Последние две рассматриваемые операции над бинарным деревом поиска – включение узла в дерево и исключение. Алгоритмы включения и исключения вершин дерева не должны нарушать основное свойство.

При включении узла в дерево сначала выполняется поиск места вставки, а затем узел вставляется с изменением поля у вставляемого и родительского узлов:

```
//------------------------------------ включение узла
bool Insert(Node* pTree, void* NewItem)
   {Node* temp = pTree, *n = NULL;
     bool x = true;
     while (x == true && temp != NULL)
                  //поиск места включения
     {
     n = temp;
     if (NewItem < temp->Data) temp = temp->Left;
     else    if (NewItem > temp->Data) temp = temp->Right;
             else  x = false;
     }
     if (x == true && n == NULL)
     pTree = new  Node(NULL, NULL, NULL, NewItem);
             //включить в корень
     else    if (x == true && NewItem < n->Data)
             n->Left  = new  Node(n, NULL, NULL, NewItem);
             //включить слева
             else  if (x == true && NewItem > n->Data))
             n->Right = new  Node(n, NULL, NULL, NewItem);
             //включить справа
 return x;
};
```

Функция начинает работу с корневого узла и перемещает указатель `temp` вниз. Указатель `n` постоянно указывает на родительский по отношению к `temp` узел, а сам указатель перемещается в соответствии с результатами сравнения. После того, как указатель `temp` становиться равным 0, он находится в той позиции, куда следует поместить новый узел `NewItem`.

Удаление вершины из бинарного дерева

Функция удаления более сложная, поскольку необходимо рассматривать различные варианты. Если удаляемый узел – лист, то удаление сводится к обнулению в родительском узле указателя на удаляемый (рис. 19.6).

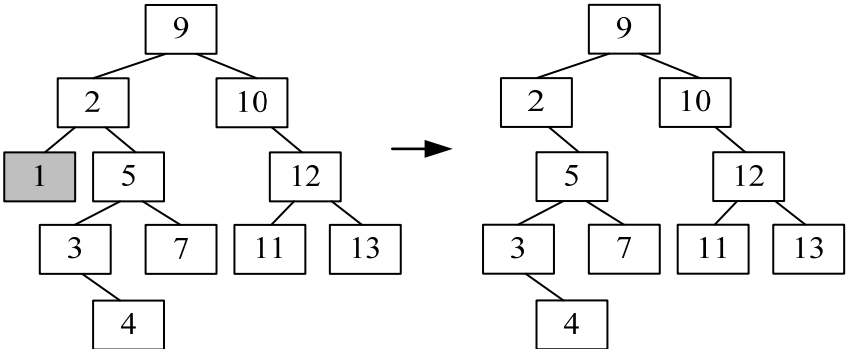


Рис. 19.6. Удаление листа дерева

Если у удаляемого только одно поддерево, то указатель в родительском должен указывать на дочерний по отношению к удаляемому и должен быть исправлен указатель на родительский узел в дочернем по отношению к удаляемому (рис. 19.7).

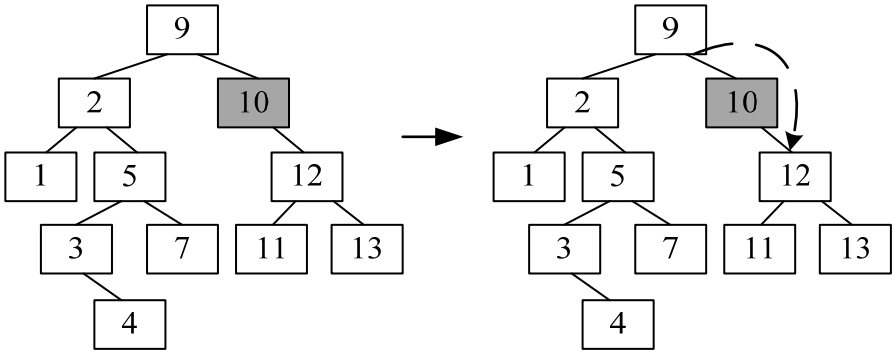


Рис. 19.7. Удаление узла дерева с одним потомком

Если у удаляемого узла два дочерних, то надо найти следующий за ним узел, извлечь его из дерева и заменить им удаляемый (рис. 19.8).

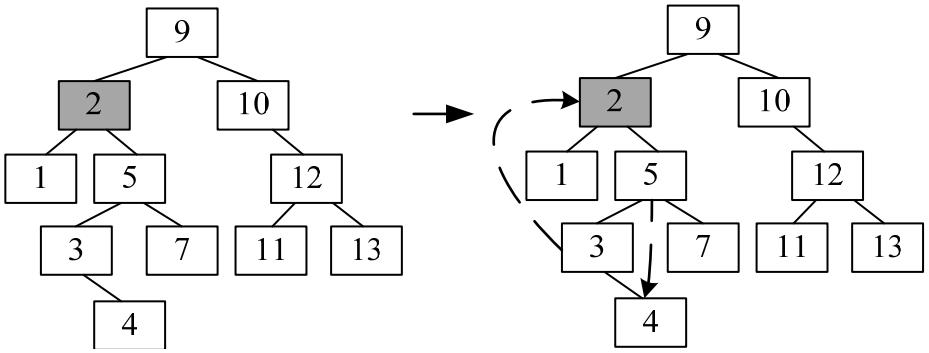


Рис. 19.8. Удаление узла дерева с двумя потомками

Реализация алгоритма представлена ниже:

```
//------------------------------------ удаление узла
bool  Delete(Node* pTree,  Node* n )
{
  bool x = true;
  if (x = (n !=  NULL))
  {
     if (n->Left == NULL && n->Right == NULL)
     //узел - лист
     {
         if (n->Parent == NULL) pTree = NULL;
         else   if (n->Parent->Left == n)
                     n->Parent->Left = NULL;
                else n->Parent->Right = NULL;
                delete n;
     }
     else   if (n->Left==NULL && n->Right!=NULL)
            //есть правое поддерево
     {
            if (n->Parent == NULL) pTree = n->Right;
            else   if (n->Parent->Left == n)
                        n->Parent->Left  = n->Right;
                   else n->Parent->Right = n->Right;
                        n->Right->Parent = n->Parent;
                   delete n;
     }
     else   if (n->Left!=NULL && n->Right==NULL)
            //есть левое поддерево
     {
            if (n->Parent == NULL)  pTree = n->Left;
            else   if (n->Parent->Left == n)
                        n->Parent->Left  = n->Left;
                   else n->Parent->Right = n->Left;
                   n->Right->Parent = n->Parent;
                   delete n;
     }
```

```
    else   if (n->Left != NULL && n->Right != NULL)
           //есть оба поддерева
    {
           Node* temp = n->Next();
           n->Data = temp->Data;
           x = Delete(temp);
    }
  }
  return  x;
}
```

Все приведенные операции имеют сложность $O(h)$, где h – высота дерева. Когда дерево приближается к полному, т.е. когда для каждого внутреннего узла по два потомка, высота дерева составляет примерно $\log_2 n$, где n – общее количество узлов дерева. В наихудшем случае эффективность всех описанных операций над бинарным деревом поиска составляет $O(\log n)$. Однако в вырожденном дереве (вырождается в одну цепочку) эффективность падает до $O(n)$.

19.3. Сбалансированные деревья

Одной из наиболее часто встречающихся задач является поиск необходимых данных. Одним из методов, улучшающих время поиска в бинарном дереве, является создание сбалансированных деревьев, обладающих минимальным временем поиска.

Дерево является *сбалансированным* тогда и только тогда, когда для каждого узла высота его двух поддеревьев различается не более чем на 1.

С тем, чтобы предупредить появление несбалансированного дерева, вводится для каждого узла (вершины) дерева показатель сбалансированности, который может принимать одно из трех значений, левое (L), правое (R), сбалансированное (B), в соответствии со следующими определениями:

– узел *левоперевешивающий*, если самый длинный путь по его левому поддереву на единицу больше самого длинного пути по его правому поддереву;

– узел *сбалансированный*, если равны наиболее длинные пути по обоим его поддеревьям;

– узел *правоперевешивающий*, если самый длинный путь по его правому поддереву на единицу больше самого длинного пути по его левому поддереву.

В сбалансированном дереве каждый узел должен находится в одном из этих трех состояний. Если в дереве существует узел, для которого это условие несправедливо, такое дерево называется *несбалансированным*.

Процесс включения узла в сбалансированное дерево состоит из последовательности трех этапов:

1) следовать по пути поиска (по ключу), пока не будет найден ключ или окажется, что ключа нет в дереве;

2) включить новый узел и определить новый показатель сбалансированности;

3) пройти обратно по пути поиска и проверить показатель сбалансированности у каждого узла.

В общем, процесс удаления элемента состоит из следующих этапов:

1) следовать по дереву, пока не будет найден удаляемый элемент;

2) удалить найденный элемент, не разрушив структуры связей между элементами;

3) произвести балансировку полученного дерева и откорректировать показатели сбалансированности.

Простыми случаями являются удаление терминальных узлов и удаление узлов с одним потомком. Если же узел, который надо удалить, имеет два поддерева, будем заменять его самым правым узлом левого поддерева.

На каждом шаге должна передаваться информация о том, увеличилась ли высота поддерева (в которое произведено включение). Поэтому, можно ввести в список параметров переменную *h*, означающую: высота поддерева увеличилась. Таким образом, структура узла дерева:

```
struct AVL_Node           //узел бинарного дерева
  {
     AVL_Node* Parent;    //указатель на родителя
     AVL_Node* Left;      //указатель на левое поддерево
     AVL_Node* Right;     //указатель на правое поддерево
     void* Data;          //данные
     int  Height;         //высота узла
     int  Weight;         //вес узла
     char bf;             //баланс-фактор
  }
```

Необходимые операции балансировки полностью заключаются в обмене значениями ссылок. Фактически ссылки обмениваются значениями по кругу, что приводит:

• к однократному правому (RR);
• однократному левому (LL);
• двукратному лево-правому (LR);
• двукратному право-левому (RL) «повороту» узлов.

Рассмотрим пример удаления различных узлов из сбалансированного дерева.

Пусть в бинарном дереве на рис. 19.9 надо удалить узел со значением 12. Удаление узла 12 само по себе просто, т.к. он представляет собой терминальный узел. Однако при этом появляется несбалансированность в корневом узле 9. Его балансировка требует однократного левого поворота.

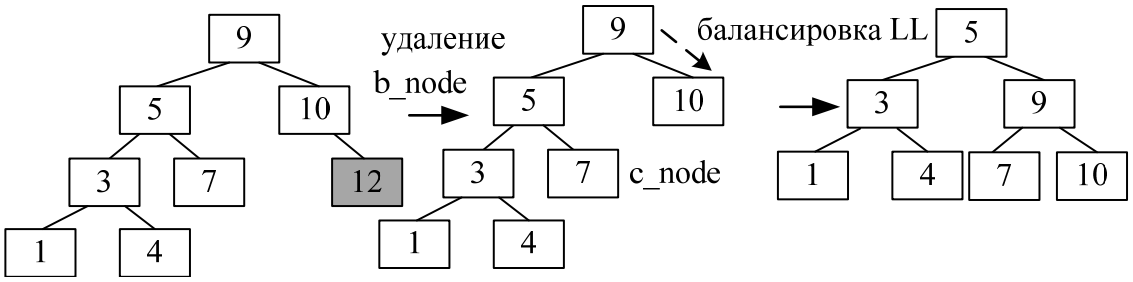


Рис. 19.9. Удаление узла и LL-балансировка дерева

При однократном левом повороте (LL) необходимо ввести следующие обозначения: левого «потомка», поворачиваемого узла обозначим переменой `b_node`, а правого «потомка» `b_node` обозначим `c_node` (рис. 19.9). Функции левого поворота `LLRotate()` передается два параметра: указатель на узел, для которого выполняется поворот `pNode`, и корень дерева `m_pRoot`. Реализация функции будет выглядеть следующим образом:

```
bool LLRotate(AVL_NODE* pNode, AVL_NODE* m_pRoot)
{
     AVL_NODE*  b_node = pNode->Left;
     AVL_NODE*  c_node = b_node->Right;

//первое - меняем местами pNode и b_node
     if(pNode->Parent)
     {
        if(pNode->Parent->Left == pNode)
           pNode->Parent->Left = b_node;
        else
           pNode->Parent->Right = b_node;
     }
     else m_pRoot = b_node;
     b_node->Parent = pNode->Parent;
```

```
//второе - меняем местами правого сына b_node и pNode
     b_node->Right = pNode;
     pNode->Parent = b_node;

//третье - меняем местами левого сына pNode и c_node
     pNode->Left = c_node;
     if(c_node) c_node->Parent = pNode;

//последнее, обновить высоту и баланс-фактор pNode и b_node
     if(c_node)
     {
pNode->Height = (c_node->Height >= pNode->Right->Height)?
(c_node->Height+1) : (pNode->Right->Height+1);
pNode->bf = (char)(c_node->Height - pNode->Right->Height);
     }
     else
        {   if(pNode->Right)
            {  pNode->Height = 1;
               pNode->bf = -1;
            }
            else
            {  pNode->Height = 0;
               pNode->bf = 0;
            }
     }
b_node->Height = (b_node->Left->Height >= pNode->Height)?
(b_node->Left->Height+1) : (pNode->Height+1);
b_node->bf =(char)(b_node->Left->Height - pNode->Height);
return true;
}
```

Пусть в бинарном дереве на рис. 19.10 надо удалить узел со значением 3. Для устранения разбалансировки в узле 5 требуется однократный правый поворот. Технология поворота продемонстрирована на рис. 19.10. Функция правостороннего поворота `RRRotate()` будет симметрична функции `LLRotate()`.

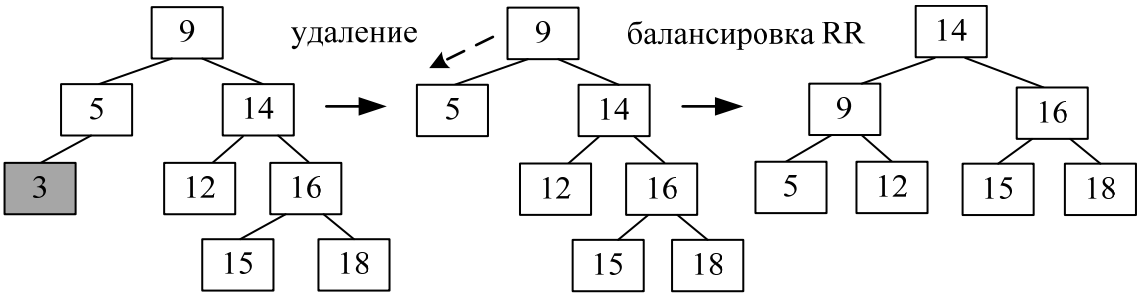


Рис. 19.10. Удаление узла и RR-балансировка дерева

Технически сложнее будет выглядеть третий случай – двукратный поворот направо и налево (RL). Например, при удалении узла 3 для дерева на рис. 19.11 балансировка узла 9 требует двукратного RL-поворота.

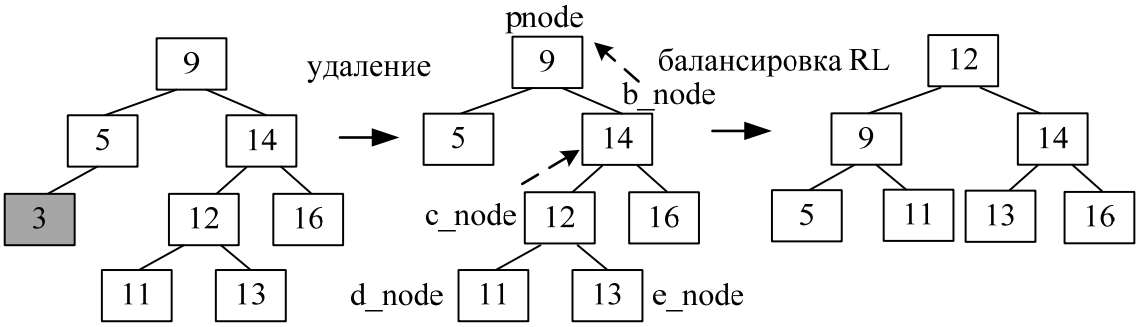


Рис. 19.11. Удаление узла и RL-балансировка дерева

Введем дополнительные обозначения: правого «потомка» поворачиваемого узла `pNode` обозначим переменной `b_node`, левого «потомка» `b_node` обозначим `c_node`, левого «потомка» `c_node` обозначим `d_node` и правого «потомка» `c_node` обозначим `e_node` (рис. 19.11). Тогда функция `RLRotate()` будет выглядеть следующим образом:

```
bool RLRotate(AVL_NODE* pNode, AVL_NODE* m_pRoot)
{
     AVL_NODE* b_node = pNode->Right;
     AVL_NODE* c_node = b_node->Left;
     AVL_NODE* d_node = c_node->Left;
     AVL_NODE* e_node = c_node->Right;
//поменять местами pNode и  c_node
     if(pNode->Parent)
     {   if (pNode->Parent->Left == pNode)
             pNode->Parent->Left = c_node;
        else pNode->Parent->Right = c_node;
     }
     else m_pRoot = c_node;
     c_node->Parent = pNode->Parent;
//поменять левого потомка c_node с pNode
//и правого потомка с b_node
     c_node->Left = pNode;
     c_node->Right = b_node;
     pNode->Parent = b_node->Parent = c_node;
//поменять правого потомка pNode с d_node
//и левого потомка  b_node с e_node
     pNode->Right = d_node;
     if(d_node) d_node->Parent = pNode;
                b_node->Left = e_node;
     if(e_node) e_node->Parent = b_node;
```

```
//обновление высоты и баланс-факторов узлов
//обновить pnode
     if(d_node)
{Node->Height = (pNode->Left->Height >= d_node->Height)?
(pNode->Left->Height+1) : (d_node->Height+1);
pNode->bf = (char)(pNode->Left->Height - d_node->Height);
     }
     else
        {  if(pNode->Left)
           {  pNode->Height = 1;
              pNode->bf = 1;
           }
           else
              {   Node->Height = 0;
                  Node->bf = 0;
              }
           }
//обновить b_node
     if(e_node)
{b_node->Height =(e_node->Height >= b_node->Right->Height)?
(e_node->Height+1) : (b_node->Right->Height+1);
b_node->bf =(char)(e_node->Height - b_node->Right-
>Height);}
     else
     {   if(b_node->Right)
         {  b_node->Height = 1; b_node->bf = -1;  }
         else
         {  b_node->Height = 0; b_node->bf = 0;}
     }
//обновить c_node
c_node->Height = (pNode->Height >= b_node->Height)?
(pNode->Height+1) : (b_node->Height+1);
c_node->bf =(char)(pNode->Height - b_node->Height);
return true;
}
```

Симметрично рассмотренному будет выглядеть двукратный поворот налево и направо (LR) (рис. 19.12).

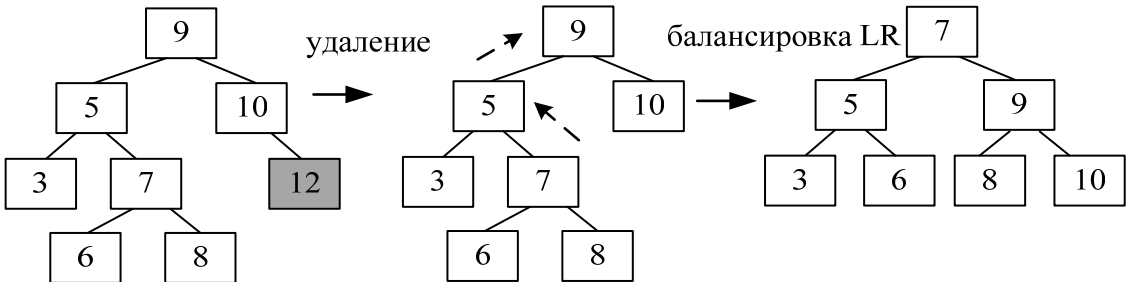


Рис. 19.12. Удаление узла и LR-балансировка дерева

Сбалансированные деревья называют иногда *AVL-деревья* (в честь их двух изобретателей Г. М. Адельсона-Вельского и Е. М. Ландиса).

19.4. Красно-черные деревья

Красно-черные деревья, или *RB-деревья* (от англ. Red-Black Trees), были введены Р. Байером в 1972 г. В стандартной библиотеке классов языка C++ множество и нагруженное множество (классы `set` и `map`) реализованы именно как красно-черные деревья.

Вместо баланс-фактора, применяемого в AVL-деревьях, RB-деревья используют цвета узлов.

Определение. Каждый узел красно-черного дерева окрашен либо в красный, либо в черный цвет (в реализации за цвет отвечает логическая переменная). При этом должны выполняться дополнительные условия:

1) каждый терминальный (или нулевой) узел считается черным;
2) корневой узел дерева черный;
3) у красного узла «дети» черные;
4) всякий путь от корня дерева к произвольному узлу имеет одно и то же количество черных узлов.

Последний пункт определения означает сбалансированность дерева по черным узлам. Ниже на рис. 19.13 приведен пример красно-черного дерева. Черные узлы изображены темно-серым цветом, красные – белым.

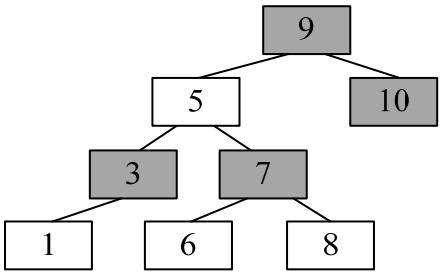


Рис. 19.13. Красно-черное дерево

Из пункта 3 определения следует, что в произвольном пути от корня к терминальному узлу не может быть двух красных узлов подряд. Это означает, что, поскольку число черных узлов в любом пути одинаково, длины разных путей к терминальным узлам отличаются не более чем вдвое. Это свойство близко по своей сути к сбалансированности. Несложно показать, что для красно-черного дерева справедлива следующая оценка сверху на высоту дерева в зависимости от числа вершин: $h \le 2\log_2(n + 1)$. Из этого следует, что поиск в красно-черном дереве также выполняется за логарифмическое время.

Новый узел добавляется в красно-черное дерево как терминальный после процедуры поиска (этим RB-дерево ничем не отличается от других упорядоченных деревьев). Новый узел окрашивается в красный цвет. При этом пункт 3 в определении красно-черного дерева может нарушиться. Поэтому после добавления, а также удаления узла выполняется процедура восстановления структуры дерева, играющая ту же роль, что и восстановление балансировки AVL-дерева. Преимущество красно-черных деревьев состоит в том, что процедура восстановления более простая. Во многих случаях она ограничивается перекрашиванием узлов. В ней также могут выполняться операции вращения узла влево и вправо, но число вращений может быть не больше двух при добавлении элемента и не больше четырех при удалении. Всего число операций при восстановлении структуры RB-дерева оценивается сверху через высоту дерева: *число операций* $\le Kh$, где h – высота дерева, K – константа. Поскольку для высоты RB-дерева справедлива приведенная выше логарифмическая оценка от числа вершин n, получаем оценку: *число операций* $\le C\log_2 n$, где C – константа. Таким образом, добавление и удаление элементов выполняется в случае красно-черных деревьев за логарифмическое время в зависимости от числа вершин дерева.

19.5. Практические задания

Для организации бинарного дерева создать структуру определенного вида. Создать интерфейсные функции для работы с деревом: добавление, извлечение, поиск, балансировка узлов дерева, нисходящий, восходящий, смешанный обход дерева и обход по уровням. Добавить функцию в соответствии с вариантом.

1. Вершина бинарного дерева содержит вещественное число. Разработать функцию вычисления среднего арифметического всех узлов дерева.

2. Вершина бинарного дерева содержит ключ и строку. Написать функцию, которая подсчитывает число ветвей от корня до ближайшей вершины с заданным ключом и вывести часть дерева от вершины до данного элемента на экран.

3. Вершина бинарного дерева содержит ключ и строку. Написать функцию определения числа ветвей n-го уровня этого дерева и вывода этих элементов на экран.

4. Вершина бинарного дерева содержит ключ и строку. Описать рекурсивную функцию, которая печатает элементы из всех листьев дерева.

5. Вершина бинарного дерева содержит ключ и N целых значений. Написать функцию удаления вершины с минимальной суммой N целых значений узла.

Глава 20. ХЭШ-ТАБЛИЦЫ

20.1. Введение

Хэш-таблицы (hash tables – перемешанные таблицы, таблицы с вычисляемыми адресами) – одно из величайших изобретений информатики. Сочетание массивов и списков с небольшой добавкой математики позволило создать эффективную структуру для хранения и получения динамических данных (рис. 20.1).

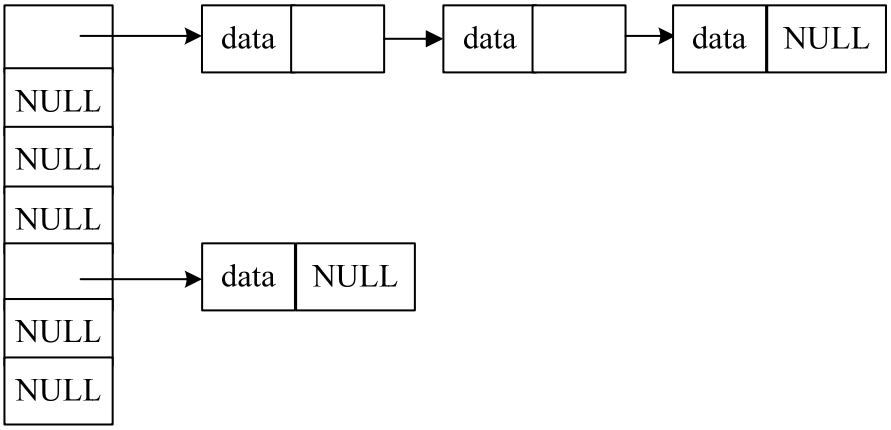


Рис. 20.1. Хэш-таблица

Идея хэш-реализации состоит в том, что работа с одним большим множеством сводится к работе с массивом небольших множеств. Рассмотрим, к примеру, записную книжку. Она содержит список фамилий людей с их телефонами (телефоны – это нагрузка элементов множества). Страницы записной книжки помечены буквами алфавита; страница, помеченная некоторой буквой, содержит только фамилии, начинающиеся с этой буквы. Таким образом, все множество фамилий разбито на 28 подмножеств, соответствующих буквам русского алфавита. При поиске фамилии мы сразу открываем записную книжку на странице, помеченной первой буквой фамилии, и в результате поиск значительно убыстряется.

Типичное применение хэш-таблиц – компилятор, который использует хэш-таблицу для управления информацией о переменных в программе; Web-браузер использует хэш-таблицу для хранения адресов страниц, которые недавно посещали для оперативного хранения (cache – кэширования) недавно использованных доменных имен и их IP-адресов. Хэш-таблицы часто применяются в базах данных. Один из наиболее эффективных способов реализации словаря – хэш-таблица.

Таким образом, хэширование полезно, когда широкий диапазон возможных значений должен быть сохранен в малом объеме памяти, и нужен способ быстрого, практически произвольного доступа.

В программировании *хэш-таблица* – динамическое множество, поддерживающее словарные операции: добавление, поиск и удаление элемента, использующее специальные методы адресации. Основное отличие таблиц от других динамических множеств – вычисление адреса элемента по значению ключа.

Алгоритмы поиска, которые используют хэширование, состоят из двух отдельных частей. Первый шаг – вычисление хэш-функции, которая преобразует ключ поиска в адрес в таблице. В идеале различные ключи должны были бы отображаться на различные адреса, но часто два и более различных ключа могут преобразовываться в один и тот же адрес в таблице. Поэтому вторая часть поиска методом хэширования – процесс разрешения конфликтов, который обрабатывает такие ключи.

20.2. Хэш-функции

Прежде всего, необходимо решить задачу вычисления хэш-функции, которая занимается преобразованием ключей в адреса в таблице или, другими словами, выполняет разбиение множества на подмножества. Она должна быть подобрана таким образом, чтобы:

1) ее можно было легко вычислять;

2) она могла принимать всевозможные различные значения приблизительно с равной вероятностью;

3) желательно, на близких значениях аргумента она принимала далекие друг от друга значения (свойство, противоположное математическому понятию непрерывности).

Модульное хэширование

Это простой, эффективный и наиболее часто используемый метод хэширования. Он заключается в выборе в качестве размера таблицы *hashTableSize* простого числа и вычислении хэша как остатка от деления *Key* на *hashTableSize:* $h(key) = key \% hashTableSize$ для любого целочисленного ключа *Key*. Такая функция называется модульной хэш-функцией и изменяется от 0 до (*hashTableSize* – 1). Это можно представить следующим образом:

```
typedef int HashIndexType;
HashIndexType Hash(int Key)
{ return Key % hashTableSize; } // модульное хэширование
```

Рассмотрим пример. Пусть множество ключей *key* = {1, 3, 56, 4, 32, 40, 23, 7, 41, 13, 6, 7} и пусть размер хэш-таблицы *hashTableSize* = 5. Тогда получим соответствующие хэш-значения *h*(*key*) = {1, 3, 1, 4, 2, 0, 3, 2, 1, 3, 1, 2}.

Для успеха этого метода очень важен выбор подходящего значения *hashTableSize*. Чтобы получить более случайное распределение ключей, в качестве *hashTableSize* нужно брать простое число, не слишком близкое к степени двух.

Модульное хэширование применяется во всех случаях, когда имеется доступ к разрядам, образующим ключи, независимо от того, являются ли они целыми числами, представленными машинным словом, последовательностью символов, упакованных в машинное слово, или представлены одним из множества других возможных вариантов.

Мультипликативный метод

Размер таблицы *hashTableSize* выбирается равным степени два (*hashTableSize* = 2^p). Значение *key* умножается на константу *A* из диапазона от $0 \le A \le 1$. В качестве такой константы в [8] рекомендуется выбирать золотое сечение *A* = (sqrt(5) – 1) / 2 = 0,6180339887499. А затем выполнить деление по модулю *hashTableSize*. Другими словами, необходимо использовать функцию $h(key) = \lfloor hashTableSize(key \times A \bmod 1) \rfloor$.

Например, *key* = 123456; *hashTableSize* = 10000; *A* = 0,618, тогда

h(*key*) = [10000 · (123456 · 0,61803…mod1)] =
= [10000 · (76500,0041151…mod1)] = [10000 · 0,0041151…] =
= [41,151…] = 41.

Аддитивный метод

Для строк переменной длины (размер таблицы равен 256) аддитивный метод дает вполне разумные результаты. В этом случае результат *h* заключен между 0 и 255:

```
typedef unsigned char HashIndexType;
HashIndexType Hash(char *str)
{   HashIndexType h = 0;
    while (*str) h += *str++;
    return h;
}
```

Исключающее ИЛИ

Используется для строк переменной длины (размер таблицы равен 256). Этот метод аналогичен аддитивному, но успешно различает схожие

слова и анаграммы (аддитивный метод даст одно значение для *XY* и *YX*). Метод заключается в том, что к элементам строки последовательно применяется операция «исключающее или». В нижеследующем алгоритме добавляется случайная компонента, чтобы еще улучшить результат:

```
typedef unsigned char HashIndexType;
unsigned char Rand8[256];
HashIndexType Hash(char *str)
{   unsigned char h = 0;
    while (*str) h = Rand8[h ^ *str++];
    return h;
}
```

Здесь `Rand8` – таблица из 256 восьмибитовых случайных чисел. Их точный порядок не критичен. Корни этого метода лежат в криптографии, но он вполне эффективен для вычисления хэша.

Универсальное хэширование

Основано на выборе хэш-функции во время исполнения программы случайным образом из некоторого множества.

Теоретически идеальная универсальная хэш-функция – это функция, для которой вероятность конфликта между двумя различными ключами в таблице размером *hashTableSize* равна в точности 1/*hashTableSize*. Использование в качестве коэффициента *A* в мультипликативном методе последовательности различных случайных значений вместо фиксированного произвольного значения преобразует мультипликативное хэширование в универсальную хэш-функцию. Однако затраты на генерирование нового случайного числа для каждого символа в ключе, скорее всего, окажутся неприемлемыми.

Можно предложить следующий вариант универсального хэширования. Для аппроксимации вероятности возникновения конфликтов для двух несовпадающих ключей до значения 1/*hashTableSize* вместо фиксированных оснований системы счисления применяются псевдослучайные значения коэффициентов. С целью минимизации нежелательных временных затрат при вычислении хэш-функции используется грубый генератор случайных чисел:

```
typedef int HashIndexType;
HashIndexType Hash (char *v, int hashTableSize)
{  int  h, a = 31415, b = 27183;
   for (h = 0; *v != 0; v++, a = a*b % (hashTableSize -1))
   h = (a*h + *v) % hashTableSize;
   return (h < 0) ? (h + hashTableSize) : h;
}
```

Типичная ошибка в реализациях хэширования заключается в том, что хэш-функция всегда возвращает одно и то же значение. Такая ошибка называется ошибкой производительности, поскольку использующая подобную хэш-функцию программа вполне может выполняться корректно, но крайне медленно (т.к. ее эффективная работа возможна только, если хэш-значения распределены равномерно). Однострочные реализации функций легко тестировать, поэтому рекомендуется проверять, насколько успешно они работают для типов ключей, которые могут встретиться в любой конкретной реализации таблицы символов.

Для проверки гипотезы, что хэш-функция создает случайные значения, можно использовать функцию статистического распределения χ^2, но на практике достаточно использовать проверку того, что значения распределены до такой степени, чтобы ни одно из них не доминировало.

20.3. Прямая адресация

Хэш-функции преобразуют ключи в адреса таблицы. Второй компонент алгоритма хэширования – определение способа обработки случая, когда два ключа представляются одним и тем же адресом. Существует два варианта хэш-таблиц: с прямой и открытой адресацией. Хэш-таблица содержит некоторый массив *H*, элементы которого есть пары (хэш-таблица с открытой адресацией) или списки пар (хэш-таблица с прямой адресацией).

При прямой адресации (direct-adress table, называют иногда раздельное связывание – separate chaining) необходимо построить для каждого адреса таблицы *H* связный список элементов, ключи которых отображаются на этот адрес. Данный подход ведет непосредственно к обобщению метода элементарного поиска в списке.

Коллизия (*collision* – ситуация, когда для различных ключей получается одно и то же хэш-значение) разрешается с помощью цепочек (chaining). Элементы множества, которым соответствует одно и то же значение хэш, связываются в цепочку-список. В строке таблицы *H* с номером *j* хранится не сам элемент, а указатель на голову списка тех элементов, у которых хэш-значение ключа равно *j*, если нет – то `NULL` (рис. 20.1).

Среднее время выполнения операций «поиск» и «вставка» в хэш-таблице с прямой адресацией равно коэффициенту заполнения и имеет константное время выполнения.

Прямая адресация применима, если количество возможных ключей невелико. Основной недостаток: если множество ключей велико, то хранить в памяти массив непрактично и иногда невозможно. Если

число реально присутствующих в таблице записей мало по сравнению с мощностью массива *H*, то много памяти тратиться зря.

Реализация

Хэш-таблица с прямой адресацией может быть представлена массивом списков:

```
struct Object_Hash                          //хэш-таблица
     {
     int hashTableSize;                     //размер таблицы
     int (*GetKey)(void*);                  //вычисление ключа
     List* Hash;                            //массив списков
     Object(int size,int (*f)(void*))       //конструктор
     {
              hashTableSize =size;
              GetKey=f;
     Hash=new List[size];
     };
     int HashFunction(void* data);          //хэш-функция
     bool Insert(void* data);               //вставить
     List::Element* Search(void* data);     //найти
     bool Delete(void* data);               //удаление
     void Scan(void(*f2)(void*)));          //обход таблицы
     };
```

Для выполнения основных операций необходимо найти хэш-функцию для заданного элемента, это будет определять номер списка. И затем вызвать одноименную функцию для определенного списка и решить задачу:

```
//----------------------------
int Object_Hash::HashFunction(void* data)
{return (GetKey(data)% hashTableSize);};
//----------------------------
bool Object_Hash::Insert(void* data)
{return (Hash[HashFunction(data)].Insert(data));};
//----------------------------
bool Object_Hash::Delete(void* data)
{return (Hash[HashFunction(data)].Delete(data));};
//----------------------------
List::Element* Object_Hash::Search(void* data)
{return Hash[HashFunction(data)].Search(data);};
//----------------------------
void Object_Hash::Scan(void(*f2)(void*))
{
for (int i = 0; i < this-> hashTableSize; i++)
     //для каждого списка
```

```
      if(this->Hash[i].Head != NULL)
      {
      std::cout<<std::endl;
      (this->Hash)[i].PrintList(f2); //вывести список
      }
}
```

20.4. Открытая адресация

В открытой адресации хэш-списков нет, все записи хранятся в самой хэш-таблице, каждая ячейка содержит либо значение динамического множества, либо NULL. Число хранимых элементов не может быть больше размера таблицы: коэффициент заполнения таблицы не больше 1. При открытой адресации указатели не используются, последовательность просматриваемых ячеек вычисляется, т.е. зависит от ключа. При добавлении нового элемента мы просматриваем таблицу, пока не найдем свободное место. К хэш-функции добавляем второй аргумент – номер попытки $h(key, i)$.

Недостаток – время поиска может оказаться большим даже при низком коэффициенте заполнения.

Рассмотрим способы вычисления последовательности испробованных мест при открытой адресации (фактически это способы перестановки последовательностей).

Линейный (линейное зондирование)

Линейный алгоритм последовательности проб основан на формуле $h(key, i) = (h'(key) + i) \bmod hashTableSize$.

Например, пусть необходимо вставить в таблицу хэширования ключи: 8881234, 8882345, 8883456, 8884321, 8886543. Размер таблицы *hashTableSize* = 11. Тогда для *hash(key)* = *key* mod 11:

1) *hash*(8881234) = 8881234 mod 11 = 10;

2) *hash*(8882345) = 8882345 mod 11 = 10 (коллизия):

эта коллизия решается на основе соотношения

$p_1 = (p_0 + 1) \bmod 11 = 11 \bmod 11 = 0$;

3) *hash*(8883456) = 8883456 mod 11 = 10 (коллизия):

коллизия решается на основе соотношения

$p_1 = (p_0 + 1) \bmod 11 = 11 \bmod 11 = 0$;

эта коллизия решается на основе соотношения

$p_2 = (p_0 + 2) \bmod 11 = 12 \bmod 11 = 1$

и т.д. В результате после определенного числа шагов получим хэш-таблицу, представленную на рис. 20.2.

0	8882345
1	8883456
2	NULL
3	NULL
4	
5	NULL
6	8884321
7	8885432
8	8886543
9	NULL
10	8881234

Рис. 20.2. Хэш-таблица
с открытой адресацией

Как видно из рис. 20.2, основной недостаток алгоритма – образование кластеров.

Квадратичный метод

Квадратичный алгоритм последовательности проб основан на выражении

$h(key, i) = (h'(key) + c_1 i + c_2 i^2) \bmod hashTableSize$,

где c_1 и c_2 – константы, не равные нулю. Выбор c_1, c_2 и *hashTableSize* не может быть произвольным. При $c_1 = \frac{m+2}{2}$, $c_2 = -\frac{m}{2}$ квадратичная последовательность проб становится линейной. Считается, что кластеров нет, но есть эффект образования так называемых вторичных кластеров.

Двойное хэширование

Один из лучших методов разрешения коллизий. Основная стратегия остается той же, что и при выполнении линейного зондирования. Отличие состоит в том, что вместо исследования каждой позиции таблицы, следующей за конфликтной, используется вторая хэш-функция для получения постоянного шага, который будет использоваться для последовательности зондирования.

Формула вычисления хэш-функции при двойном хэшировании:

$h(key, i) = (h_1(key) + ih_2(key)) \bmod hashTableSize$,

где $h_1(key)$ и $h_2(key)$ – хэш-функции. Фактически, это арифметическая прогрессия по модулю *hashTableSize* с первым членом $h_1(key)$ и шагом $h_2(key)$.

Реализация хэш-таблицы

Предположим, что значения ключей – отдельные небольшие числа, вычисляемые на основе содержания данных. В этом случае простейшая реализация хэш-таблицы основана на сохранении элементов в массиве, индексированном по ключам. В структуру таблицы включены также: фактический и максимально возможный размер, основные операции включения, поиска, удаления и конструктор объекта:

```
struct Object_Hash                      //хэш-таблица
{
   void** Data;                         //массив
   Object_Hash(int, int (*)(void*));//конструктор
   int hashTableSize;                   //размер таблицы
   int N;                               //количество элементов
   int (*GetKey)(void*);                //вычисление ключа
   bool Insert(void*);                  //вставить
   int SearchInd(int key);              //поиск по ключу
   void* Delete(int key);               //удалить по ключу
};
```

Конструктор инициализирует поля таблицы: динамически выделяет память и обнуляет элементы, устанавливает функцию вычисления ключа, переменная-счетчик непустых позиций таблицы устанавливается в ноль:

```
//-------------------- констуктор
Object_Hash::Object_Hash (int size,int(*getkey)(void*))
{  //инициализация хэш-таблицы
     N=0;

     this->hashTableSize = size;
     this->GetKey=getkey;
     this->Data=new void*[size];
     for(int i=0;i< size;++i)Data[i]=NULL;
}
```

Для вставки элемента при условии, что в таблице есть свободные места, необходимо получить индекс. Индекс в таблице рассчитывается на основе модульной хэш-функции. В случае коллизии используется линейный алгоритм ее разрешения, рассмотренный выше. Новый элемент вставляется при наличии свободного места по вычисленному индексу, в противном случае, поиск продолжается:

```
//---------------------------- вставка элемента
bool Object_Hash::Insert(void* NewItem)
{
bool x=false;
if(N!= hashTableSize)
//проверка на переполнение
//найти место вставки, используя линейное зондирование
for (int i=0,t=GetKey(NewItem),
     j=HashFunction(t, hashTableSize,0);
     i!= hashTableSize &&!x;
     j=HashFunction(t, hashTableSize,++i))
          if(Data[j]==NULL||Data[j]==-1)
                                //если свободно
          {
             Data[j]= NewItem;  //вставить элемент
             N++;               //увеличить их количество
             x=true;
          }
          return x;
}
```

При поиске данных по ключу вычисляется хэш-функция, которая определяет индекс данных в таблице. При наличии таких данных, для них вычисляется ключ и сравнивается с искомым ключом. Если они совпадают, запоминается индекс найденного элемента:

```
//-------------найти по ключу, вернуть индекс в таблице
int Object_Hash::SearchInd(int key)
{
  int index=-1;
  bool x=false;
  if(N!=0)  //если таблица не пуста, вычислить хэш-функцию
  for(int i=0,j=HashFunction(key, hashTableSize,0);
      Data[j]!=NULL && i!= hashTableSize &&!x;
      j=HashFunction(key, hashTableSize,++i))
      if(Data[j]!=-1)           //если элемент содержится
         if(GetKey(Data[j])==key)//если совпадают ключи
         {  index =j;//запомнить индекс
            x=true;
         }
      return index;
}
```

Операция удаления сводится к задаче поиска по заданному ключу, и в случае успешного результата слот в хэш-таблице помечается как удаленный. В хэш-таблицах с открытой адресацией невозможно удалить элемент из таблицы: просто поставить NULL нельзя, т.к. это может

прервать цепочку поиска алгоритма. Поэтому, как правило, при удалении элемента, его просто помечают признаком «удален»: (–1).

```
//------------------------ удаление элемента по ключу
void* Object_Hash::Delete(int key)
{
  int i=SearchInd(key); //найти индекс удаляемого элемента
  void* oldItem=Data[i];
  if(oldItem!=NULL)     //если элемент есть
    {
        Data[i]=-1;     //пометить как удаленный
        N--;            //уменьшить количество элементов
    }
   return oldItem;
}
//------------------------ хэш-функция
int HashFunction(int key,int TableSize,int p)//хэш-функция
{    return (p+key)% TableSize;  }
```

20.5. Свойства хэш-таблиц

Одним из основных параметров хэш-таблиц, от которого зависит среднее время выполнения операций, является *коэффициент заполнения*. Это число хранимых элементов *n*, деленное на размер массива *hashTableSize* (число возможных значений хэш-функции), называемое коэффициентом заполнения хэш-таблицы (load factor) $\alpha = n / hashTableSize$ (может быть меньше или больше 1).

Важное свойство хэш-таблицы состоит в том, что все три операции в среднем выполняются за время $O(1)$. Но при этом не гарантируется, что время выполнения отдельной операции мало. Это связано с тем, что при достижении некоторого значения коэффициента заполнения необходимо осуществить перестройку индекса хэш-таблицы: увеличить значение размера массива *H* и заново добавить в пустую хэш-таблицу все пары.

Если рассматривать хэш-таблицу как совокупность связанных списков, то по мере того, как таблица растет, увеличивается количество списков и, соответственно, среднее число узлов в каждом списке уменьшается. Если количество элементов равно *n*, а размер таблицы равен *hashTableSize* = 1, то таблица вырождается в один список длиной *n*. Если размер таблицы равен *hashTableSize* = 2 и хэширование идеально, то получается два списка по *n/hashTableSize* элементов в каждом. Это сильно уменьшает длину списка, в котором нужно искать. Как видно из табл. 20.1, имеется значительная свобода в выборе длины таблицы.

Таблица 20.1

Зависимость времени поиска от размера таблицы

Размер таблицы, hashTableSize	Время поиска, μs	Размер таблицы, hashTableSize	Время поиска, μs
1	869	128	9
2	432	256	6
4	214	512	4
8	106	1024	4
16	54	2048	3
32	28	4096	3
64	15	8192	3

Установлено, что для эффективного хэширования коэффициент загрузки α должен быть от 0,2 до 0,7; количество ячеек в хэше *hashTableSize* – простое число; алгоритм хэширования необходимо предварительно тестировать на простых и критических наборах; коллизии лучше разрешать с помощью связанных цепочек.

20.6. Практические задания

Для организации хэш-таблицы: разработать интерфейсные функции: включение, удаление, поиск, обход; добавить функцию расчета коэффициента заполнения хэш-таблицы и функцию в соответствии с вариантом.

1. За основу взять хэш-таблицу с открытой адресацией. Реализовать рассмотренные функции вычисления хэша для решения коллизий. Провести исследования, сделать вывод.

2. За основу взять хэш-таблицу с прямой адресацией (на основе списков). Для вычисления хэш-функции использовать метод универсального хэширования.

3. За основу взять хэш-таблицу с прямой адресацией (на основе списков). Для вычисления хэш-функции использовать алгоритм на основе «исключающее ИЛИ» для поля строки.

4. За основу взять хэш-таблицу с открытой адресацией. Реализовать динамическое расширение хэш-таблицы на фиксированную величину при достижении максимального размера таблицы.

5. За основу взять хэш-таблицу с открытой адресацией. Использовать рассмотренные методы генерации хэш-функций. Провести исследования, сделать вывод.

Глава 21. МЕТОДЫ РАЗРАБОТКИ АЛГОРИТМОВ

Метод декомпозиции

Возможно, самым важным и наиболее широко применимым методом проектирования эффективных алгоритмов является метод, называемый методом *декомпозиции* (метод «*разделяй и властвуй*», или метод разбиения). Этот метод предполагает такую декомпозицию (разбиение) задачи размера n на более мелкие задачи, что в результате позволяет, на основе решений этих более мелких задач, получить решение исходной задачи. Данный метод уже применялся в пособии для решения задач сортировки слиянием, в деревьях двоичного поиска, рекурсивного решения задачи о ханойских башнях.

Рассмотрим пример. Пусть требуется решить задачу умножения двух n-битовых целых чисел X и Y. Вспомним, что алгоритм умножения n-битовых (или n-разрядных) целых чисел, изучаемый в средней школе, связан с вычислением n промежуточных произведений размера n и поэтому является алгоритмом $O(n^2)$, если на каждом шаге выполняется умножение или сложение одного бита или разряда. Один из вариантов метода декомпозиции применительно к умножению целых чисел заключается в разбиении каждого из чисел X и Y на два целых числа по $n/2$ битов в каждом (для простоты предполагается, что n является степенью числа 2):

$X = A2^{n/2} + B$;

$Y = C2^{n/2} + D$,

где A, C, B и D – соответственно старшие и младшие разряды чисел X и Y. Теперь произведение чисел X и Y можно записать в виде

$XY = AC2^n + (AD + BC)2^{n/2} + BD$.

Таким образом, получаем четыре умножения $(n/2)$-битовых целых чисел (AC, AD, BC и BD), три сложения целых чисел, содержащих не более $2n$ битов, и два сдвига (умножение на 2^n и $2^{n/2}$).

Поскольку сложения и сдвиги требуют $O(n)$ шагов, можно составить следующее рекуррентное соотношение для $T(n)$ – общего количества операций с битами, требуемого для умножения n-битных целых чисел по формуле

$T(1) = 1$;

$T(n) = 4T(n/2) + cn$.

Значение константы c равно 1. Тогда однородное и частное решения имеют порядок $O(n^2)$. Асимптотическая эффективность будет, таким образом, не больше, чем при использовании стандартного метода.

Динамическое программирование

Часто не удается разбить задачу на небольшое число подзадач, объединение решений которых позволяет получить решение исходной задачи. В таких случаях можно попытаться разделить задачу на столько подзадач, сколько необходимо, затем каждую подзадачу разделить на еще более мелкие подзадачи и т.д. Если бы весь алгоритм сводился именно к такой последовательности действий, в результате был бы получен алгоритм с экспоненциальным временем выполнения.

Но чаще удается получить лишь полиномиальное число подзадач, и поэтому ту или иную подзадачу приходится решать многократно. Если вместо этого отслеживать решения каждой решенной подзадачи и просто отыскивать в случае необходимости соответствующее решение, можно получить алгоритм с полиномиальным временем выполнения.

С точки зрения реализации иногда бывает проще создать таблицу решений всех подзадач, которые когда-либо придется решать. Таблица заполняется независимо от того, нужна ли будет на самом деле конкретная подзадача для получения общего решения. Заполнение таблицы подзадач для получения решения определенной задачи получило название *динамического программирования* (это название происходит из теории управления). Формы алгоритма динамического программирования могут быть разными.

«Жадные» алгоритмы

Суть метода заключатся в следующем: на каждой отдельной стадии «*жадный»* алгоритм выбирает тот вариант, который является локально оптимальным в том или ином смысле. Не каждый «жадный» алгоритм позволяет получить оптимальный результат в целом. Жадная стратегия иногда обеспечивает лишь сиюминутную выгоду, в то время как в целом результат может оказаться неблагоприятным.

Примером «жадных» алгоритмов могут быть: алгоритм построения кратчайшего пути Дейкстры; алгоритм построения остовного дерева минимальной стоимостью Крускала (задача коммивояжера). Алгоритм кратчайшего пути Дейкстры является «жадным» в том смысле, что он всегда выбирает вершину, ближайшую к источнику, среди тех, кратчайший путь которых еще неизвестен. Алгоритм Крускала также «жадный», он выбирает из остающихся ребер, которые не создают цикл, ребро с минимальной стоимостью.

Существуют задачи, для которых ни один из известных «жадных» алгоритмов не позволяет получить оптимального решения, тем

не менее имеются «жадные» алгоритмы, которые с большой вероятностью позволяют получать хорошие решения. Нередко вполне удовлетворительным можно считать почти оптимальное решение, характеризующееся стоимостью, которая лишь на несколько процентов превышает оптимальную. Если единственным способом получить оптимальное решение является использование метода полного поиска, тогда «жадный» алгоритм или другой эвристический метод получения хорошего решения может оказаться единственным реальным средством достижения результата.

Полный перебор

Иногда невозможно применить ни один из рассмотренных выше методов, способных помочь отыскать оптимальный вариант решения, и остается прибегнуть к *полному перебору* (называемому *поиском с возвратом*).

Например, при построении алгоритма игры можно строить ассоциированное с ней дерево, называемое деревом игры. Каждый узел такого дерева представляет определенную позицию. Начальная позиция соответствует корню дерева. Если позиция *x* ассоциируется с узлом *n*, тогда потомки узла *n* соответствуют совокупности допустимых ходов из позиции *x*, и с каждым потомком ассоциируется соответствующая результирующая позиция. Каждому узлу дерева соответствует определенная цена. Сначала назначаются цены листьям. Эти цены распространяются вверх по дереву. Для оценивания внутренних узлов применяются правила: взять максимум среди потомков на тех уровнях, где предстоит сделать ход игроку 1, и минимум среди потомков на тех уровнях, где предстоит сделать ход игроку 2.

Игры не единственная категория задач, которые можно решать полным перебором всего дерева возможностей. Широкий спектр задач, в которых требуется найти минимальную или максимальную конфигурацию того или иного типа, поддаются решению путем поиска с возвратом. Узлы такого дерева можно рассматривать как совокупности конфигураций, а каждый потомок узла *n* представляет некоторое подмножество конфигураций. Каждый лист представляет отдельную конфигурацию или решение соответствующей задачи; каждую такую конфигурацию можно оценить и попытаться выяснить, не является ли она наилучшей среди уже найденных решений. Если просмотр организован достаточно рационально, каждый из потомков некоторого узла будет представлять намного меньше конфигураций, чем соответствующий узел.

Алгоритмы локального поиска

Для улучшения текущего решения некоторой задачи можно применить к нему какое-либо преобразование из некоторой заданной совокупности преобразований. Это улучшенное решение становится новым текущим решением. Повторение этой процедуры можно выполнять до тех пор, пока ни одно из преобразований в заданной их совокупности не позволит улучшить текущее решение. Результирующее решение может, хотя и необязательно, оказаться оптимальным.

Если совокупность преобразований невелика, естественно рассматривать решения, которые можно преобразовывать одно в другое за один шаг, как близкие. Такие преобразования называются *локальными*, а соответствующий метод называется *локальным поиском*.

Одной из задач, которую можно решить именно методом локального поиска, является задача нахождения минимального остовного дерева.

Алгоритмы локального поиска проявляют себя с наилучшей стороны как эвристические алгоритмы для решения задач, точные решения которых требуют экспоненциальных затрат времени.

ПРИЛОЖЕНИЕ 1

ASCII-КОДЫ СИМВОЛОВ

Код	Символ	Код	Символ	Код	Символ	Код	Символ
0	NULL	32	(пробел)	64	@	96	‘
1	SOH ☺	33	!	65	A	97	a
2	STX ☻	34	“	66	B	98	b
3	ETX ♥	35	#	67	C	99	c
4	EOT ♦	36	$	68	D	100	d
5	ENQ ♣	37	%	69	E	101	e
6	ACK♠	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS ◘	40	(	72	H	104	h
9	HT ○	41	)	73	I	105	i
10	LF ◙	42	*	74	J	106	J
11	VT ♂	43	+	75	K	107	k
12	FF ♀	44	,	76	L	108	l
13	CR ♪	45	-	77	M	109	m
14	SO♫	46	.	78	N	110	n
15	SI ☼	47	/	79	O	111	o
16	DLE ►	48	0	80	P	112	p
17	DC1 ◄	49	1	81	Q	113	q
18	DC2 ↕	50	2	82	R	114	r
19	DC3 !!	51	3	83	S	115	s
20	DC4 ¶	52	4	84	T	116	t
21	NAK §	53	5	85	U	117	u
22	SYN ▬	54	6	86	V	118	v
23	ETB ↨	55	7	87	W	119	w
24	CAN ↑	56	8	88	X	120	x
25	EM ↓	57	9	89	Y	121	y
26	SUB →	58	:	90	Z	122	z
27	ESC ←	59	;	91	[	123	{
28	FS ∟	60	<	92	\	124	|
29	GS ↔	61	=	93	]	125	}
30	RS ▲	62	>	94	^	126	~
31	US ▼	63	?	95	_	127	DEL ⌂

Примечание. Коды 0-31 и 127 зарезервированы за управляющими символами и не выводятся на печать.

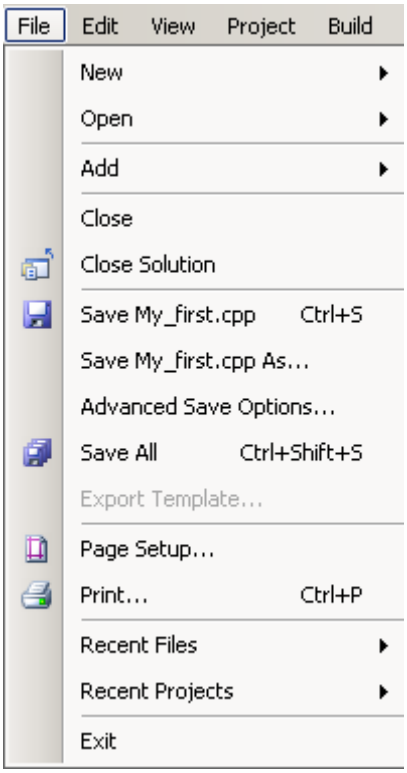
ПРИЛОЖЕНИЕ 2

КРАТКОЕ ОПИСАНИЕ КОМАНД МЕНЮ И ВКЛАДОК ИНТЕГРИРОВАННОЙ СРЕДЫ РАЗРАБОТКИ MS VS 2008

Меню File (Файл)

Содержит стандартные команды для работы с файлами, встречающиеся во многих приложениях Windows.

New (*Новый*) – открывает диалоговое окно для выбора нового проекта, файла, Web-сайта или другого документа. Обычно с этой команды начинается создание любой программы. Пункту меню *New* соответствует кнопка на панели инструментов.



Open (*Открыть*) – предназначена для открытия уже существующего и сохраненного на диске проекта/решения, файла или Web-сайта. Пункт вызывает появление стандартного окна диалога.

Close (*Закрыть*) – закрывает открытый файл. Если открыто несколько файлов, будет закрыто активное (текущее) окно. Активное окно имеет фокус ввода и изображается цветом, установленным в системе для отображения активных окон. Обычно у активных окон выделен цветом заголовок.

Close Solution (*Закрыть решение*) – закрывает текущее решение.

Save as… (*Сохранить как...*) – позволяет создать копию содержимого активного окна под другим именем.

Save All (*Сохранить все*) – записывает в соответствующие файлы содержимое всех открытых окон.

Page Setup… (*Настройка параметров страницы...*) – используется для задания колонтитулов и установки размера полей, используемых при печати.

Print… (*Печать...*) – печать содержимого активного окна.

Recent Files, Recent Projects (*Последние файлы*, *Последние Проекты*) – содержат списки последних использовавшихся файлов и

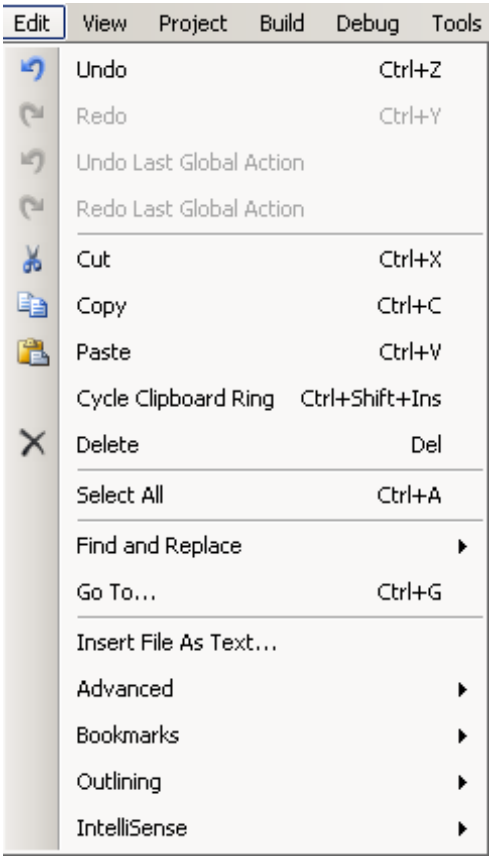
проектов. Чтобы открыть любой из них, достаточно щелкнуть левой кнопкой мыши на нужном имени.

Exit (*Выход*) – завершает работу среды. Если какие-то файлы не сохранены, среда автоматически выдаст предупреждение и даст возможность сохранить их.

Меню Edit (Правка)

Команды этого меню позволяют быстро находить и исправлять текст в активном окне. Команды Undo, Redo, Cut, Copy, Paste, Delete, Select All, Find and Replace стандартны, поэтому их описание не приводится.

Go To… (*Перейти…*) – позволяет быстро переместить курсор к определенному месту текущего документа. После выбора этой команды откроется диалоговое окно, в котором можно задать номер строки программы, куда следует перейти. Если будет введено значение, превышающее число строк программы, то курсор будет перемещен в конец файла.



Advanced (*Расширенный*) – предоставляет дополнительные возможности редактирования текста.

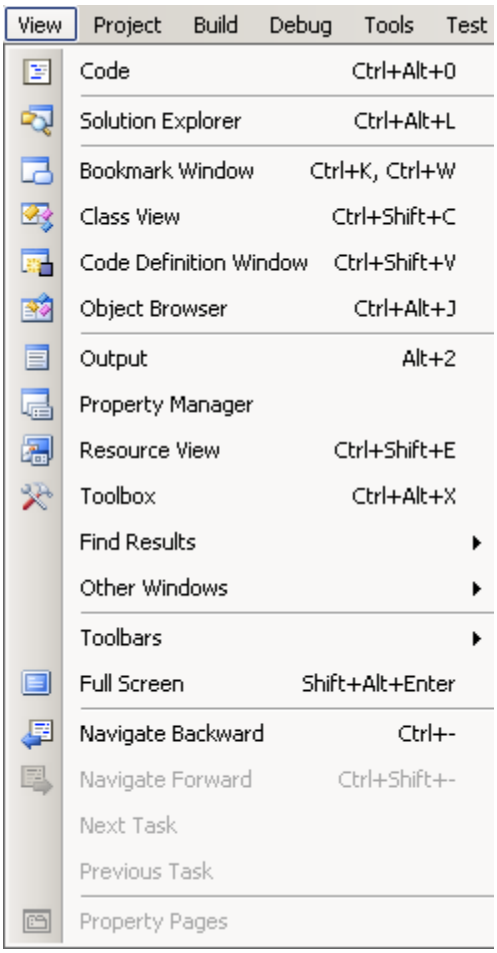
Bookmarks (*Закладки*) – позволяет установить, удалить или перейти к закладке. Закладками можно отмечать отдельные строки в тексте программы, к которым придется обращаться позже.

Outlining (*Структурирование*) – содержит команды работы с блоками. Позволяет сворачивать, разворачивать сегметы блока, включать и отключать автоматическую структуризацию, сворачивать блоки в определения.

IntelliSence – обеспечивает поддержку данной технологии. Позволяет выводить списки членов выбранного объекта (*List Members*), получать информацию о параметрах функции (*Parameter Info*), быструю информацию об объекте (*Quick Info*).

Меню View (Вид)

Содержит команды, позволяющие настраивать внешний вид рабочего пространства, переключаться между окнами рабочего пространства: редактор кода (*Code*), проводник решений (*Solution Explorer*), окно закладок (*Bookmark Window*), представление классов (*Class View*, не распознает классы, если они не зарегистрированы в файле базы данных), проводник объектов (*Object Browser*), окна вывода (*Output*), диспетчер свойств (*Property Manager*), представление ресурсов (*Resourse View*) т.д.

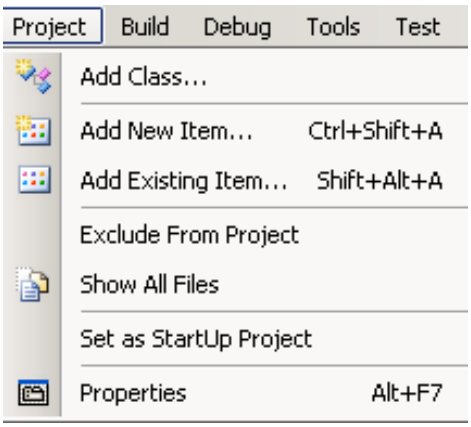


Меню Project (Проект)

Содержит команды, позволяющие управлять содержимым проекта. В зависимости от выбранного обозревателя, набор команд изменяется. Если активизировано окно обозревателя решений, то меню содержит следующие команды.

Add Class… (*Добавить класс...*) – открывает диалоговое окно автоматизированного добавления в проект нового класса.

Add New Item… (*Добавить новый...*) – открывет диалоговое окно добавления в проект новго объекта на основе имеющихся шаблонов (рисунки, иконки, Windows Form, *.cpp и т.д.).



Add Existing Item… (*Добавить существующий...*) – открывает диалоговое окно добавления в проект существующего модуля.

Exclude From Project (*Исключить из проекта*) – исключает из проекта текущий модуль.

Show all Files (*Показать все файлы*) – отображает все файлы проекта, имеющиеся на диске.

Set as StartUp Project (*Сделать стартовым проектом*) – позволяет назначить стартовый проет. Если проектов несколько, то при запуске отладки будет запущен стартовый проект.

Properties (*Свойства*) – вызывает диалоговое окно (рис. П2.1) для установки важных параметров проекта. Здесь же осуществляется настройка компилятора и компоновщика. В большинстве случаев можно пользоваться установками по умолчанию.

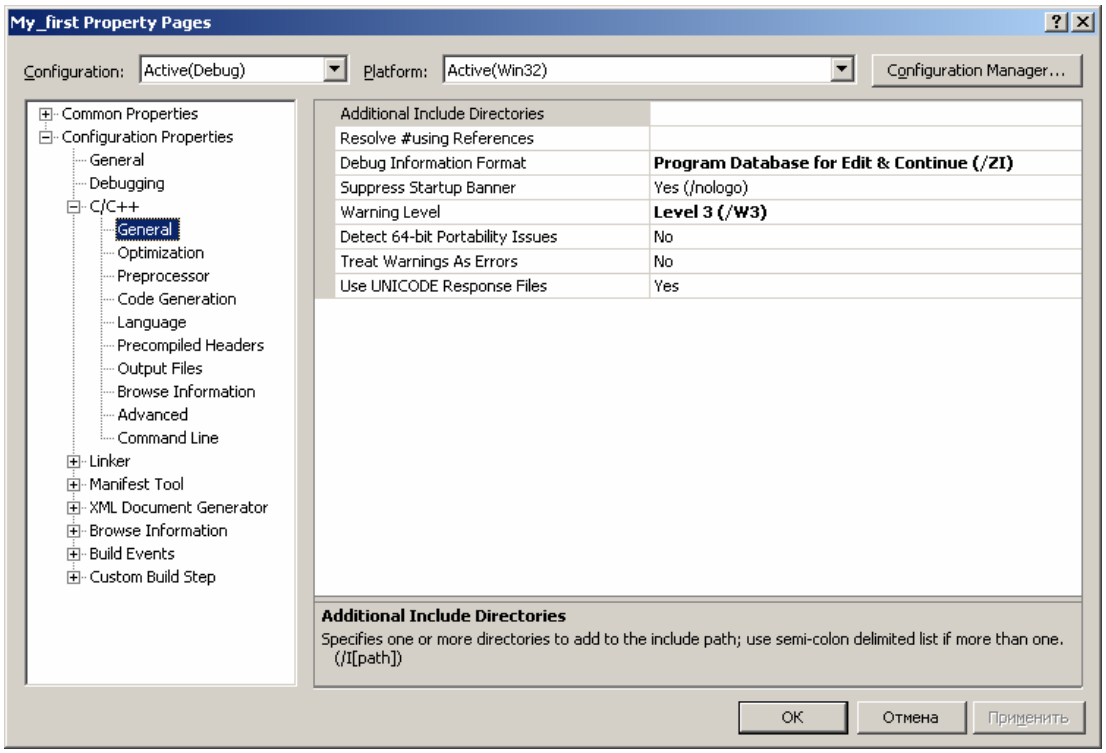


Рис. П2.1. Дилоговое окно Свойства

В поле *Configuration* (*Конфигурация*), вверху окна свойств проекта, можно выбрать настраиваемую конфигурацию. На рис. П2.1 выбрана отладочная конфигурация. Об этом говорит слово *Active* (активный, текущий). Для настройки релизной конфигурации нужно выбрать соответствующий пункт.

Для настройки компилятора надо перейти к пункту *Общие* (*General*). Если для поля *Supress Startup Banner* (*Подавлять начальное сообщение*) выбирать *No*, то при компиляции в окно вывода будут выведены параметры командной строки. Поле *Warning Level* (*Уровень предупреждений*) определяет насколько компилятор внимательно следит за вашим кодом. На первом уровне компилятор не выдаст предупреждения. Самый жесткий уровень – четвертый (по умолчанию выбран третий уровень).

Следующий пункт настроек компилятора – *Optimization* (*Оптимизация*). Первое поле *Optimization* определяет оптимизацию программы. Отладочная версия программы собирается без оптимизации (флажок /Od). Отключение оптимизации обеспечивает самую быструю компиляцию. Другие флажки оптимизации: /O1, /O2, /Ox. Флажок /Ox нужно использовать только в релизный версиях.

Optimization	**Disabled (/Od)**
Inline Function Expansion	Default
Enable Intrinsic Functions	No
Favor Size or Speed	Neither
Omit Frame Pointers	No
Enable Fiber-safe Optimizations	No
Whole Program Optimization	No

В поле *Favor Size of Speed* (*Предпочесть размер или скорость*) по умолчанию выбран пункт *Neither* (ни то, ни другое). С флажком /Ot программа будет работать быстрее, а с /Os – иметь меньший размер.

Поле *Whole Program Optimization* (*Полная оптимизация программы*) задается флажком /GL. Если включается данный пункт, то в оптимизации компоновщика нужно добавить флажок /ltcg. Полная оптимизация программы используется только в релизной версии.

Пункт *Command Line* (*Командная строка*) позволяет увидеть все настройки компилятора, заданные флажками (или ключами). Она может выглядеть следующим образом:

```
/Od /D "WIN32" /D "_DEBUG" /D "_CONSOLE" /D "_UNICODE" /D "UNICODE" /Gm /EHsc /RTC1 /MDd
/Yu"stdafx.h" /Fp"Debug\My_first.pch" /Fo"Debug\\" /Fd"Debug\vc90.pdb" /W3 /nologo /c /ZI /TP
/errorReport:prompt
```

В результате компиляции в папке *Debug* (или *Release*) появятся объектные файлы *.obj. В исполняемый файл *.exe их соберет компоновщик.

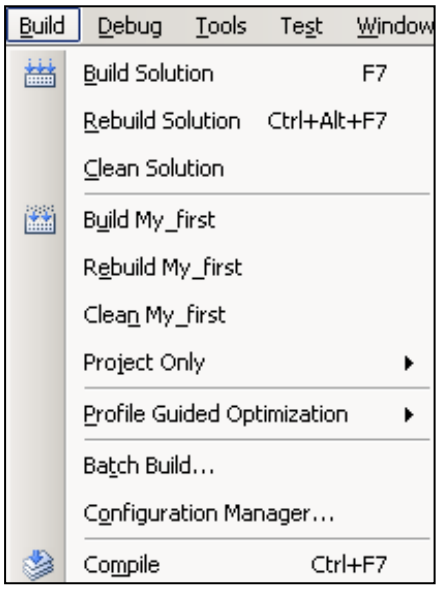
Таким же образом можно настроить свойства *Общие* (*General*), *Компоновщика* (*Linker*) и *Отладчика* (*Debugger*).

Меню Build (Построение)

Содержит пункты, необходимые для генерации исполняемого файла.

Build Solution (*Построение решения*) – в процессе построения анализируются все файлы проекта и затем компилируются и компонуются лишь те из них, которые были изменены. Прежде чем выбрать команду Build, надо принять решение, следует ли в конечный файл включать

отладочную информацию (конфигурация *Debug*) или же исключить эти данные из файла (конфигурация *Release*).



Rebuild Solution (*Перестроение решения*) – выполняет компиляцию и построение всех файлов независимо от изменений.

Clean Solution (*Очистка решения*) – удаляет промежуточные и выходные файлы проекта для его эффективного хранения (рекомендуется использовать для значительного сокращения размера решения).

Далее содержаться аналигочные команды, но для активного модуля.

Batch Build… (*Пакетное построение...*) – может создать в одном проекте сразу несколько целевых файлов (конфигураций одного проекта).

Configuration Manager… (*Диспетчер конфигурации...*) – открывает диалоговое окно для добавления и удаления конфигураций проекта.

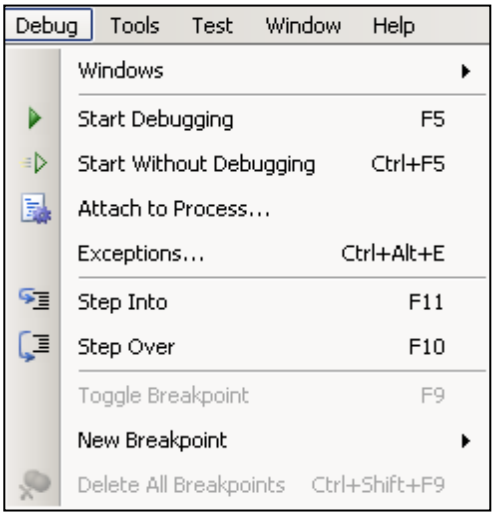
Compile (*Компиляция*) – выполняет компиляцию текущего проекта.

Меню Debug (Отладка)

Windows… (*Окна...*) – позволяет добавить к интерфейсу отладки дополнительные окна просмотра.

Start Debuging (*Начать отладку*) – выполнение приложения с использованием средств интерактивной отладки.

Start Without Debuging (*Запустить без отладки*) – построение и выполнение решения. Если в исходном коде есть ошибки, решение не будет построено.



Attach to Process… (*Присоединиться к процессу...*) – позволяет отлаживать программу, которая уже запущена. Отображает список процессов, которые уже запущены на машине, и позволяет выбрать процесс, который надо отладить. Это средство для опытных разработчиков.

Exeptions… (*Исключения...*) – вызывает диалоговое окно, которое позволяет настроить исключения, вызывающие останов приложения.

Step Into (*Шаг с заходом*) – пошаговое выполнение приложения. Выполняется по одному оператору за раз, заходя в каждый вложенный блок, в каждую вызываемую функцию (в том числе и в библиотечные функции).

Step Over (*Шаг с обходом*) – пошаговое выполнение метода без трассировки вызываемых методов.

Toggle Breakpoint (*Точка останова*) – назначить и отменить точку останова. Назначенная точка останова отмечается маркером в виде красного круга слева от текущей строки.

Delete All Breakpoints (*Удалить все точки останова*) – убирает все установленные точки останова.

Stop Debugging (*Остановить отладку*) – вывести приложение из отладочного режима.

Меню Tools (Сервис)

Содержит команды вызова вспомогательных утилит, макросов программирования и настройки среды.

Attach to Process… (*Присоединиться к процессу...*) – вызывает диалоговое окно, позволяющее присоединить и отладить любой процесс, запущенный на компьютере.

Device Security Manager... (*Диспетчер безопасности устройства...*) и *Device Emulator Manager...* (*Диспетчер эмулятора устройства...*) – тестовые инструменты для разработчиков приложений на Windows Mobile.

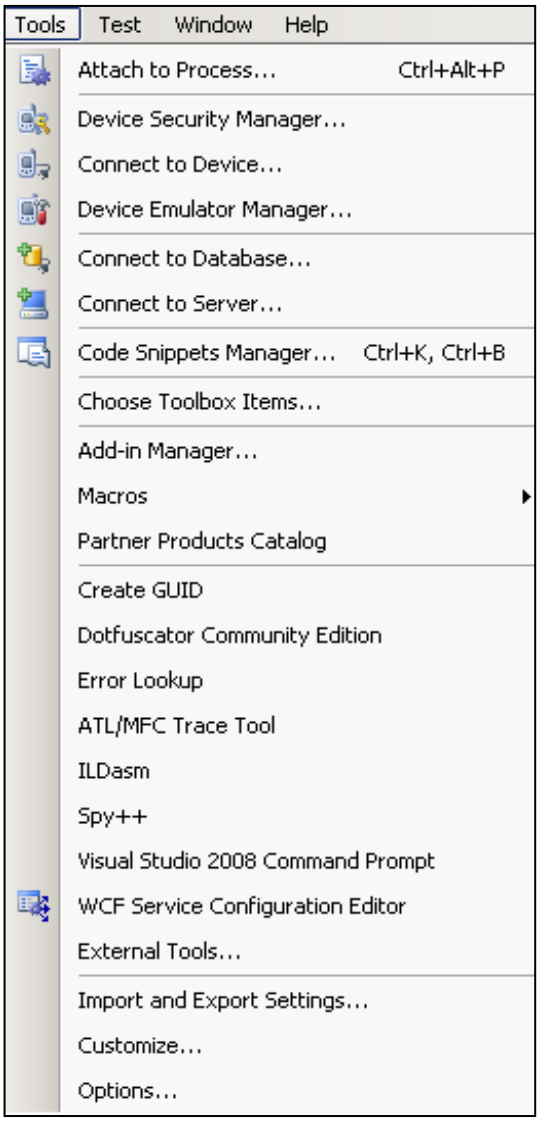
Connect to Device... (*Подключиться к устройству...*) – устанавливает подключение к устройству или эмулятору.

Connect to Database... (*Подключиться к базе данных...*), *Connect to Server...* (*Подключиться к серверу...*) – устанавливает подключение к базе данных и серверу соответственно.

Code Snippets Manager... (*Диспетчер фрагментов кода...*) – позволяет устанавливать катлоги и отдельные фрагменты, которые должны быть вставлены в код.

Macros (*Макрос*) – эти команды используются для создания и воспроизведения макросов на VBScript. Позволяют значительно упростить и ускорить работу в среде Visual C++.

Error Lookup (*Поиск ошибки*) – утилиту используют при необходимости получить текст сообщений, связанных с кодами системных ошибок. Код ошибки вводится в поле Value, и в поле ErrorMessage



автоматически отобразится связанное с ним сообщение.

MFC/ATL Tracer Tool (*Инструмент трассировки*) – содержит дополнительные возможности для отладки оконных приложений, построенных на основе *MFC/ATL*. Эта утилита отображает в окне отладки сообщения о выполнении операций, связанных с использованием библиотеки MFC, а также предупреждения об ошибках, если при выполнении приложения происходят какие-либо сбои.

Spy++ (*Шпион*) – выводит информацию о выполняющихся системных процессах и потоках, существующих окнах и поступающих оконных сообщениях. Указанная утилита также предоставляет набор инструментов, облегчающих поиск нужных процессов, потоков и окон.

Customize... (*Настройка...*) – при выборе данной команды открывается диалоговое окно *Customize*, которое позволяет настраивать меню и панели инструментов, а также назначать различным командам сочетания клавиш.

Options... (*Параметры...*) – данная команда открывает диалоговое окно *Options*, в котором задаются различные параметры среды Visual C++.

Среда содержит следующие **вкладки**:

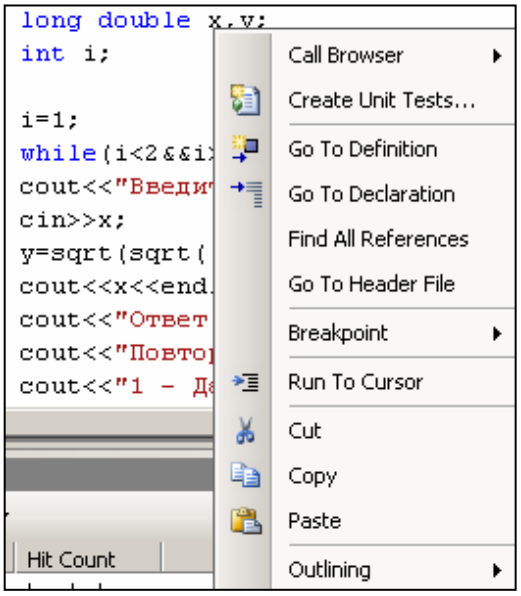
Solution Explorer (*Обозреватель решений*) – предоставляет обзор всех проектов текущего решения и файлов, которые они содержат;

Class View (*Окно классов*) – отображает классы, определенные в проекте, а также содержимое каждого из классов;

Resource View (*Представление ресурсов*) – отображает диалоговые окна, пиктограммы, панели меню и другие ресурсы, используемые программой;

Property Manager (*Диспетчер свойств*) – показывает свойства, установленные для отладочной и рабочей версии проекта.

Если щелнуть правой кнопкой мыши на какой-нибудь идентификатор в редакторе кода, то откроется **контекстное меню**.



Пункты *Go To Definition* (*Перейти к определению*), *Go To Declaration* (*Перейти к объявлению*), *Find All References* (*Найти все ссылки)* позволяют быстро находить определения, объявления и ссылки на текущий объект.

Имеются средства редактирования текста: *Cut* (*Вырезать*), *Paste* (*Вставить*), *Copy* (*Копировать*) *и Outlining* (*Структурирование*).

Visual C++ содержит ряд встроенных редакторов: редакторы ресурсов (позволяют создавать и модифицировать ресурсы Windows, такие как растровые изображения, указатели мыши, значки, меню, диалоговые окна и т.д); редактор диалоговых окон (средство, позволяющее создавать сложные диалоговые окна); редактор изображений; редактор двоичных кодов (позволяет вносить изменения непосредственно в двоичный код ресурса); редактор строк (ресурс, содержащий список идентификаторов и значений всех строковых надписей, используемых в приложении; наличие единой таблицы позволяет легко менять язык интерфейса программы).

ПРИЛОЖЕНИЕ 3

ОПИСАНИЕ КЛЮЧЕВЫХ СЛОВ C/C++

Ключевое слово	Описание
Ключевые слова C/C++	
auto	объявить локальную переменную
break	выйти из цикла
case	определенная ветка в операторе ветвления
char	объявить символьную переменную
const	объявить неизменяемые данные или функцию, которая не изменяет данные
continue	пропустить код до конца цикла и начать новую итерацию
default	вариант по умолчанию в case
do	оператор цикла
double	объявить вещественное число двойной точности
else	ветка оператора if, которая выполняется при ложном условии
enum	создание перечисляемого типа
extern	указание компилятору, что переменная объявлена в другом файле
float	объявить вещественную переменную
for	оператор цикла
goto	безусловный переход
if	оператор условия
int	объявить переменную целого типа
long	объявить длинное целое
register	запрос компилятору на оптимизацию переменной по скорости
return	возврат из функции
short	объявить короткое целое
signed	сделать данный целый тип знаковым
sizeof	возвратить размер переменной или типа
static	создать статическую переменную
struct	определить новую структуру
switch	оператор ветвления
typedef	создание нового типа на основе существующего

Ключевое слово	Описание
Ключевые слова только для C++	
union	структура, содержащая несколько переменных в одной области памяти
unsigned	объявить беззнаковое целое
void	объявить функцию или переменную без типа
volatile	предупреждение компилятору, что переменная может измениться сама
while	оператор цикла
inline (C99)	оптимизация вызовов для функций (встраиваемые функции)
restrict (C99)	наложение ограничений
asm	вставить код на ассемблере
bool	объявить булеву переменную
catch	обработать исключение от throw
class	объявить класс
const_cast	приведение типа от константной переменной
delete	освобождение памяти, выделенной new
dynamic_cast	выполнить приведение типов во время выполнения
explicit	использовать конструктор только при полном соответствии типов
false	константа для ложного значения булевой переменной
friend	разрешить другим функциям доступ к приватным данным класса
virtual	создать виртуальную функцию
mutable	перекрыть константность
namespace	определить новое пространство имен
new	выделить динамическую память под новую переменную
operator	создание перегруженных операторов
private	объявить приватное поле класса
protected	объявить защищенное поле класса
public	объявить общее поле класса
reintepret_cast	изменить тип переменной
static_cast	сделать не полиморфное приведение типов
template	создать шаблонную функцию

Ключевое слово	Описание
this	указатель на текущий объект
true	константа для истинного значения булевой переменной
throw	выбросить исключение
try	выполнить код, который может выкинуть исключение
typename	возвратить имя класса или не определено
typeid	описать объект
using	импортировать полностью или частично указанное пространство имен внутрь текущего блока
wchar_t	объявить переменную типа wide-character
Альтернативные в C++	
and	альтернатива оператору &&
and_eq	альтернатива оператору &=
bitand	альтернатива оператору bitwise &
bitor	альтернатива оператору |
compl	альтернатива оператору ~
not	альтернатива оператору !
not_eq	альтернатива оператору !=
or	альтернатива оператору ||
or_eq	альтернатива оператору |=
xor	альтернатива оператору ^
xor_eq	альтернатива оператору ^=

ПРИЛОЖЕНИЕ 4

ЗАГОЛОВОЧНЫЕ ФАЙЛЫ И СТАНДАРТНЫЕ ФУНКЦИИ ЯЗЫКА C++

В этом приложении приведен список заголовочных файлов, используемых в языке C++. Имена старых версий этих файлов указаны в скобках.

cassert (assert.h)

Эта библиотека содержит только функцию assert, применяемую для проверки диагностических утверждений.

assert (диагностическое утверждение);

Если диагностическое утверждение ложно, функция assert выводит на экран сообщение об ошибке и прекращает выполнение программы. Все вызовы функции assert в программе можно заблокировать с помощью макроса #define NDEBUG, который следует поместить перед директивами include.

cctype (ctype.h)

Большинство функций этой библиотеки позволяет распознавать буквы, цифры и т.д. Остальные функции преобразуют строчные буквы в прописные, и наоборот.

Функции, предназначенные для распознавания, возвращают значение true, если символ ch принадлежит указанной группе; в противном случае они возвращают значение false.

`isalnum(ch)`	Возвращает значение true, если символ ch является буквой или десятичной цифрой
`isalpha(ch)`	Возвращает значение true, если символ ch является буквой
`iscntrl(ch)`	Возвращает значение true, если символ ch является управляющим (т.е. его ASCII-код равен 127 или изменяется от 0 до 31)
`isdigit(ch)`	Возвращает значение true, если символ ch является десятичной цифрой

`islower(ch)`	Возвращает значение true, если символ ch является строчной буквой
`ispunct(ch)`	Возвращает значение true, если символ ch является знаком пунктуации
`isspace(ch)`	Возвращает значение true, если символ ch является пробельным: пробелом, знаком табуляции, командой перехода на новую строку или прогона бумаги
`isupper(ch)`	Возвращает значение true, если символ ch является прописной буквой
`toascii(ch)`	Вернуть ASCII-код символа ch
`tolower(ch)`	Если символ ch является строчной буквой, преобразовать ее в прописную; в противном случае вернуть символ ch
`toupper(ch)`	Если символ ch является прописной буквой, преобразовать ее в строчную; в противном случае вернуть символ ch

`cfloat (float.h)`

В этой библиотеке определены именованные константы, указывающие диапазон изменения значений с плавающей точкой.

`climits (limits.h)`

В этой библиотеке определены именованные константы, указывающие диапазон изменения целочисленных значений.

`cmath (math.h)`

Функции, содержащиеся в этой библиотеке, предназначены для стандартных математических вычислений. Эти функции являются перегруженными и выполняют вычисления с числами, имеющими тип float, double и long double. Если не указано иное, каждая функция имеет один аргумент, а возвращаемое значение и аргумент имеют одинаковый тип (float, double и long double).

`acos`	Вычисляет арккосинус
`asin`	Вычисляет синус
`at an`	Вычисляет арктангенс
`atan2`	Вычисляет арктангенс x/y для аргументов x и y

`ceil`	Выполняет округление с избытком
`cos`	Вычисляет косинус
`cosh`	Вычисляет арккосинус
`exp`	Вычисляет экспоненту
`fabs`	Вычисляет абсолютную величину
`floor`	Выполняет округление с недостатком
`fmod`	Возвращает остаток от деления аргумента x на аргумент y
`fexp`	Для аргументов x и eptr, где $x = m \cdot 2^e$, возвращает число m и устанавливает eptr равным e
`log`	Возвращает значение натурального логарифма
`log10`	Возвращает значение десятичного логарифма
`pow`	Для аргументов x и y возвращает значение x^y
`sin`	Вычисляет синус
`sinh`	Вычисляет гиперболический синус
`sqrt`	Вычисляет квадратный корень
`tan`	Вычисляет тангенс
`tanh`	Вычисляет гиперболический тангенс

`cstdlib (stdlib.h)`

`abort`	Аварийно завершает выполнение программы
`abs`	Вычисляет абсолютную величину числа
`atof`	Преобразовывает строку в число с плавающей точкой
`atoi`	Преобразовывает строку в целое число
`exit`	Прерывает выполнение программы
`rand`	Вычисляет псевдослучайное целое число
`srand`	Инициализирует генератор псевдослучайных чисел аргументом, а в отсутствие аргументов – единицей

`cstring (string.h)`

Библиотека содержит функции, позволяющие манипулировать строками языка C, завершающимися нулевым символом \0. Если не указано иное, функции возвращают указатель на результирующую строку, модифицируя один из аргументов. Аргумент ch является символом, n – целое число, остальные параметры являются строками.

`strcat(toS, fromS)`	Копирует строку fromS в конец строки toS
`stncat(toS, fromS, n)`	Копирует не более n символов строки fromS в конец строки toS и дописывает символ \0
`strcmp(strl, str2)`	Возвращает отрицательное целое число, если strl < str2; NULL, если strl = str2; положительное целое число, если strl > str2
`stricmp(strl, str2)`	Аналогична функции strcmp, но игнорирует различия между прописными и строчными буквами
`strncmp(strl, str2, n)`	Аналогична функции strcmp, но сравнивает только первые n символов каждой строки
`strncpy(toS, fromS, n)`	Копирует n символов строки fromS в toS, при необходимости обрывая ее или дополняя нулевыми символами \0
`strspn(strl, str2)`	Возвращает количество первых последовательных символов строки strl, которые отсутствуют в строке str2
`strcspn(strl, str2)`	Возвращает количество первых последовательных символов строки strl, которые принадлежат строке str2
`strlen(str)`	Возвращает длину строки str, не учитывая символ \0
`strlwr(str)`	Преобразовывает прописные буквы строки str в строчные, не изменяя других символов
`strupr(str)`	Преобразовывает строчные буквы строки str в прописные, не изменяя других символов
`strchr(str, ch)`	Возвращает указатель на первое вхождение символа ch в строку str; если в строке символа нет, возвращает NULL
`strrchr(str, ch)`	Возвращает указатель на последнее вхождение символа ch в строку str; если символа в строке нет, возвращает NULL

`strpbrk(strl, str2)`	Возвращает указатель на последнее вхождение символа строки strl в строку str2; если таких символов нет, возвращает M7LL
`strstr(strl, str2)`	Возвращает указатель на последнее вхождение символа строки str2 в строку strl; если таких символов нет, возвращает M7LL
`strtok(strl, str2)`	Находит в строке strl следующую лексему, за которой следует строка str2, возвращает указатель на эту лексему и записывает непосредственно после нее символ M7LL

`fstream (fstream.h)`

В данной библиотеке объявлены классы, поддерживающие ввод и вывод.

`iomanip (iomanip.h)`

Манипуляторы, содержащиеся в этой библиотеке, влияют на формат ввода и вывода. Обратите внимание, что в библиотеке iostream есть дополнительные манипуляторы.

`setbase(b)`	Задает основание счисления b = 8, 10 или 16
`setfill(f)`	Задает символ заполнения f
`setprecision(n)`	Параметр n задает точность представления чисел с плавающей точкой
`setw(n)`	Параметр n задает ширину поля вывода

`iostream (iostream.h)`

Манипуляторы, содержащиеся в этой библиотеке, влияют на формат ввода и вывода. Обратите внимание, что в библиотеке iomanip есть дополнительные манипуляторы.

`dec`	Вынуждает операторы, выполняемые в последующем, использовать десятичное представление чисел

`endl`	Вставляет символ перехода на новую строку \n и очищает выходной поток
`ends`	Вставляет символ окончания строки \0 в выходной поток
`flush`	Очищает выходной поток
`hex`	Вынуждает операторы ввода-вывода, выполняемые в дальнейшем, использовать шестнадцатеричное представление чисел
`oct`	Вынуждает операторы ввода-вывода, выполняемые в дальнейшем, использовать восьмеричное представление чисел
`ws`	Извлекает из входного потока пробельные символы

`string`

Эта библиотека позволяет манипулировать со строками языка C++. Ниже представлены некоторые из функций, предусмотренные в этой библиотеке. Кроме того, к строкам можно применять операторы =, +, ==, !=, <, < =, >, >=, << и >>. Обратите внимание, что нумерация позиций строки начинается с нуля.

`erase ()`	Стирает содержимое строки, делая ее пустой
`erase (pos, len)`	Удаляет из строки подстроку, начинающуюся с позиции pos и содержащую len символов
`find (substring)`	Возвращает позиции подстроки в строке
`length()`	Возвращает количество символов в строке (то же, что и функция size)
`replace (pos, len, str)`	Заменяет подстроку, начинающуюся с позиции pos и содержащую len символов, строкой str
`size ()`	Возвращает количество символов в строке (то же, что и функция length)
`substr(pos, len)`	Возвращает подстроку, начинающуюся с позиции pos и содержащую len символов

ЛИТЕРАТУРА

1. Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения: ГОСТ 19.701–90. – Введ. 01.01.90. – М., 2005. – 24 с.

2. ISO/IEC 14882:2003 specifies requirements for implementations of the C++ programming language and standard library [Electronic resource]. – Mode of access: http://www.open-std.org/jtc1/sc22/wg21. – Date of access: 25.06.2010.

3. Хортон, А. Visual C++ 2005. Базовый курс / А. Хортон. – М.: Вильямс, 2007. – 1152 с.

4. Харви, Д. Как программировать на C++ / Д. Харви, П. Дейтел. – М.: БИНОМ, 2007. – 1156 с.

5. Макаров, В. Л. Программирование и основы алгоритмизации: учеб. пособие / В. Л. Макаров. – СПб.: СЗТУ, 2003. – 110 с.

6. Глухова, Л. А. Основы алгоритмизации и структурного проектирования программ: учеб. пособие / Л. А. Глухова, В. В. Бахтизин. – Минск: БГУИР, 2003. – 72 с.

7. Бусько, В. Л. Основы алгоритмизации и программирования: конспект лекций для студ. всех специальностей и форм обучения БГУИР / В. Л. Бусько, А. Г. Корбит, Т. М. Кривоносова. – Минск: БГУИР, 2004. – 103 с.

8. Структуры данных и алгоритмы: пер. с англ.: учеб. пособие / А. Ахо [и др.]. – М.: Вильямс, 2000. – 384 с.

9. Пахомов, Б. И. C/C++ и MS Visual C++ 2008 для начинающих / Б. И. Пахомов. – СПб.: БХВ-Петербург, 2009. – 624 с.

10. Седжвик, Р. Фундаментальные алгоритмы на C++. Анализ. Структуры данных. Сортировка. Поиск: пер. с англ. / Р. Седжвик. – Киев: Диасофт, 2001. – 688 с.

11. Каррано, Ф. М. Абстракция данных и решение задач на C++. Стены и зеркала: пер. с англ. / Ф. М. Каррано. – 3-е изд. – М.: Вильямс, 2003. – 848 с.

12. Методы сортировок и их реализации: методические указания к выполнению лабораторных работ / сост. И. В. Беляева, К. С. Беляев. – Ульяновск: УлГТУ, 2006. – 48 с.

13. Красиков, И. В. Алгоритмы. Просто как дважды два / И. В. Красиков, И. Е. Красикова. – М.: Эксмо, 2007. – 256 с.

ОГЛАВЛЕНИЕ

Учебное издание

Пацей Наталья Владимировна

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

Учебно-методическое пособие

Редактор *М. А. Юрасова*
Компьютерная верстка *П. В. Прохоровская*

Подписано в печать 03.12.2010. Формат $60\times84^{1}/_{16}$.
Бумага офсетная. Гарнитура Таймс. Печать офсетная.
Усл. печ. л. 16,8. Уч.-изд. л. 17,3.
Тираж 150 экз. Заказ .

Отпечатано в Центре издательско-полиграфических
и информационных технологий учреждения образования
«Белорусский государственный технологический университет».
220006. Минск, Свердлова, 13а.
ЛИ № 02330/0549423 от 08.04.2009.
ЛП № 02330/0150477 от 16.01.2009.

Переплетно-брошюровочные процессы
произведены в ОАО «Полиграфкомбинат им. Я. Коласа».
220600. Минск, Красная, 23. Заказ .