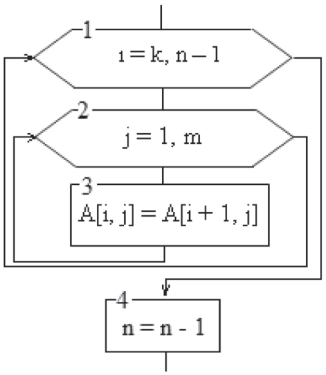
1) сдвинуть все строки, начиная со строки с номером k, вверх;

2) уменьшить количество строк в двумерном массиве.

Дано: массив $A[n, m]$, где n, m — количество строк и столбцов массива соответственно; k — номер удаляемой строки.

Найти: новую матрицу A, содержащую $(n - 1)$-ую строку и m столбцов.

Фрагмент блок-схемы алгоритма.



Тест

n	m	k	Исходная матрица	Результат
3	3	2	$\begin{pmatrix}1 & 2 & 3\\0 & 0 & 0\\4 & 5 & 6\end{pmatrix}$	$\begin{pmatrix}1 & 2 & 3\\4 & 5 & 6\end{pmatrix}$

Исполнение алгоритма.

i	j	A
2	1	$A[2, 1] = A[3, 1] = 4$
	2	$A[2, 2] = A[3, 2] = 5$
	3	$A[2, 3] = A[3, 3] = 6$

После выполнения циклов (блоки 1 и 2) матрица имеет следующий вид:

$$\begin{pmatrix}1 & 2 & 3\\4 & 5 & 6\\4 & 5 & 6\end{pmatrix}.$$

Уменьшаем количество строк матрицы (блок 4), после чего она примет искомый вид.

1.4.5. Нахождение строк или столбцов двумерного массива, обладающих заданным свойством

Пример 1.19. Дан двумерный массив $A[n, m]$. Найти количество строк, содержащих хотя бы один нуль.

Обозначим: k — количество строк, содержащих хотя бы один нуль; *Flag* — признак наличия нулей в строке. *Flag* = 1 означает, что нули в строке есть; *Flag* = 0 — нулей в строке нет.

Словесное описание алгоритма. Начинаем просматривать массив с первой строки. Берем первый элемент столбца ($j = 1$). Пока не просмотрен последний элемент столбца ($j \le m$) и не найден отрицательный элемент ($a[i, j] >= 0$), будем переходить к следующему элементу, увеличивая индекс элемента j ($j = j + 1$). Таким образом, мы закончим просмотр строки массива в одном из двух случаев:

1) просмотрели все элементы и не нашли отрицательного, тогда $Flag = 0$;

2) нашли нужный элемент, при этом $Flag = 1$. Увеличиваем значение количества строк k, содержащих нули.

Фрагмент блок-схемы алгоритма.

Блок 1. Присваиваем переменной k начальное значение.

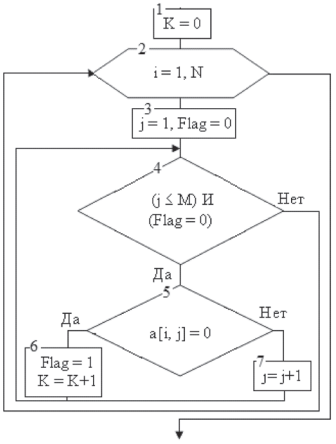
Блок 2. Берем очередную строку i.

Блок 3. Для выбранной строки задаем первый элемент строки и устанавливаем $Flag = 0$.

Блок 4. Выполняем итерационный цикл для всех элементов строки (блоки 5–7).

Блок 5. Сравниваем очередной элемент строки с нулем.

Блок 6. Если нулевой элемент найден (выход «Да» блока 5), то $Flag = 1$, увеличиваем значение k и заканчиваем просмотр строки (выход «Нет» блока 4).



Блок 7. Если нулевой элемент не найден (выход «Нет» блока 5), то продолжаем просмотр оставшихся элементов в строке, увеличивая индекс элемента j.

Покажем выполнение алгоритма на тестовом примере.

Тест

Данные			Результат
N	M	Матрица A	K
3	3	$\begin{pmatrix} 1 & 0 & 1 \\ 2 & 4 & 7 \\ 0 & 1 & 0 \end{pmatrix}$	2

Исполнение алгоритма.

i	*Flag*	*j*	(*j* <= *M*) и (*Flag* = 0)?	*A*[*i*, *j*] = 0?	K
1	0				0
		1	(1 <= 3) и (*Flag* = 0)? «Да»	*A*[1, 1] = 0? «Нет»	
		2	(2 <= 3) и (*Flag* = 0)? «Да»	*A*[1, 2] = 0? «Да»	
	1				1
			(2 <= 3) и (*Flag* = 0)? «Нет» (кц)		
2	0				
		1	(1 <= 3) и (*Flag* = 0)? «Да»	*A*[2, 1] = 0? «Нет»	
		2	(2 <= 3) и (*Flag* = 0)? «Да»	*A*[2, 2] = 0? «Нет»	
		3	(3 <= 3) и (*Flag* = 0)? «Да»	*A*[2, 3] = 0? «Нет»	
	0	4	(4 <= 3) и (*Flag* = 0)? «Нет» (кц)		
3	0				
		1	(1 <= 3) и (*Flag* = 0)? «Да»	*A*[3, 1] = 0? «Да»	2
			(4 <= 3) и (*Flag* = 0)? «Нет» (кц)		

Итак, количество строк, в которых встречается хотя бы один нуль, равно 2, что соответствует условию задачи.

1.5. АЛГОРИТМЫ СОРТИРОВКИ

Сортировка — это процесс перестановки элементов некоторого заданного множества в определенном порядке. Цель сортировки — облегчить последующий поиск элементов в отсортированном множестве.

В общем случае задача сортировки ставится следующим образом. Имеется массив, тип данных которого позволяет использовать операции сравнения («=», «>», «<», «>=» и «<=»). Задачей сортировки является преобразование исходного массива в массив, содержащий те же элементы, но в порядке возрастания (или убывания) значений.

Методы сортировки не должны требовать дополнительной памяти: все перестановки с целью упорядочения элементов массива должны производиться в пределах того же массива.

Будем выполнять сортировку по возрастанию.

Методы, сортирующие массив «на месте», делятся на три основных класса в зависимости от лежащего в их основе приема.

1. Сортировка включениями (вставкой).
2. Сортировка выбором.
3. Сортировка обменом.

1.5.1. Сортировки включениями (вставкой)

Сортировка простой вставкой

Идея алгоритма очень проста. Пусть имеется массив $a[1], a[2], \ldots, a[n]$. Пусть элементы $a[1], a[2], \ldots, a[i-1]$ уже отсортированы, и пусть имеем входную последовательность $a[i], a[i+1], \ldots, a[n]$. На каждом шаге, начиная с $i = 2$ и увеличивая i на единицу, берем i-й элемент входной последовательности и вставляем его на подходящее место в уже отсортированную часть последовательности.

Обозначим вставляемый элемент через x.

Пусть начальный массив 32 64 9 30 87 14 2 76.

$i = 2$ $x = 64$. Ищем для x подходящее место, считая, что $a[1] = 32$ — это уже отсортированная часть последовательности. Получаем *32 64* **9** 30 87 14 2 76.

$i = 3$ $x = 9$. Часть последовательности $a[1]$, $a[2]$ уже отсортирована. Ищем для x подходящее место: *9 32 63* **30** 87 14 2 76.

Аналогично выполняем последующие шаги сортировки.

$i = 4$ $x = 30$. Ищем для x подходящее место: *9 30 32 64* **87** 14 2 76.

$i = 5$ $x = 87$. Ищем для x подходящее место: *9 30 32 64 87* **14** 2 76.

$i = 6$ $x = 14$. Ищем для x подходящее место: *9 14 30 32 64 87* **2** 76.

$i = 7$ $x = 2$. Ищем для x подходящее место: *2 9 14 30 32 64 87* **76**.

$i = 8$ $x = 76$. Ищем для x подходящее место: *2 9 14 30 32 64 76* **87**.

При поиске подходящего места для элемента x мы чередовали сравнения и пересылки, т. е. как бы «просеивали» x, сравнивая его с очередным элементом $a[i]$ и либо

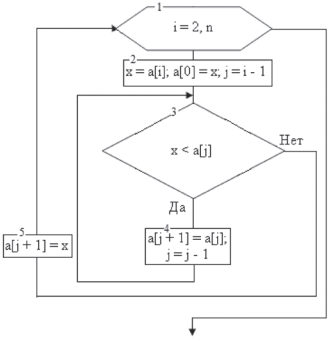
вставляя x, либо пересылая $a[i]$ направо и продвигаясь влево. Заметим, что «просеивание» может закончиться при двух различных условиях:

1) найден элемент, значение которого больше, чем x;

2) достигнут конец последовательности.

Это типичный пример цикла с двумя условиями окончания. При реализации алгоритма с целью окончания выполнения внутреннего цикла используют прием фиктивного элемента (барьера). Его можно установить, приняв $a[0] = x$.

Приведем фрагмент блок-схемы алгоритма и ее описание.



Блок 1. Начинаем цикл для перебора всех элементов исходного массива.

Блок 2. Присваиваем x значение очередного элемента массива. Устанавливаем барьер и продвигаемся назад по отсортированной части массива.

Блок 3. Начинам цикл для поиска подходящего места для значения x.

Блок 4. Пока x меньше очередного элемента (выход «Да» блока 3), выполняем сдвиг и продвигаемся к началу отсортированной части массива.

Блок 5. По выходу «Нет» блока 3 вставляем элемент x на нужное место.

Покажем исполнение алгоритма для рассмотренного примера.

i	x	$a[0]$	j	$x < a[j]$?	$a[j+1]$	Массив
2	64	64	1	64 < 32? «Нет»	$a[2] = x = 64$	32 64 9 30 87 14 2 76
3	9	9	2	9 < 64? «Да»	$a[3] = a[2] = 64$	32 64 64 30 87 14 2 76
			1	9 < 32? «Да»	$a[2] = a[1] = 32$	32 32 64 30 87 14 2 76
			0	9 < 9? «Нет»	$a[1] = x = 9$	9 32 64 30 87 14 2 76
4	30	30	3	30 < 64? «Да»	$a[4] = a[3] = 64$	9 32 64 64 87 14 2 76
			2	30 < 32? «Да»	$a[3] = a[2] = 32$	9 32 32 64 87 14 2 76
			1	30 < 9? «Нет»		$a[2] = x = 30$

После этого шага массив выглядит так: 9 30 32 64 87 14 2 76. Сортировка продолжается далее для оставшихся значений i.

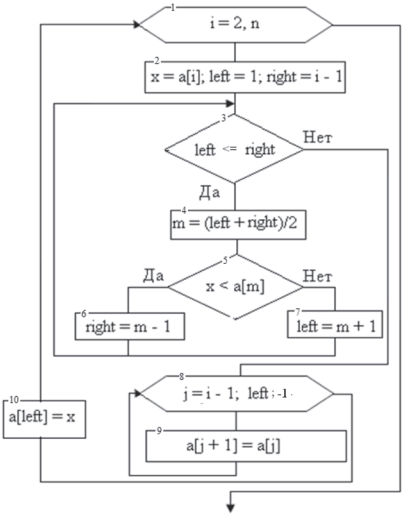
Сортировка бинарными включениями

Алгоритм сортировки простыми включениями имеет слабые места. Это, во-первых, необходимость перемещения данных, причем при вставке элементов, близких к концу массива, приходится перемещать почти весь массив. Второй недостаток — это необходимость поиска места для вставки, на что также тратится много ресурсов. Эту часть алгоритма можно улучшить, применив так называемый бинарный поиск. Этот метод на каждом шаге сравнивает x со средним (по положению) элементом отсортированной последовательности до тех пор, пока не будет найдено место включения.

Модифицированный алгоритм называется сортировкой бинарными включениями.

Обозначим *left* — левая граница отсортированного массива, *right* — его правая граница.

Приведем фрагмент блок-схемы алгоритма и ее описание.



Блок 1. Начинаем цикл для перебора всех элементов исходного массива.

Блок 2. Присваиваем x значение очередного элемента массива. Устанавливаем левую и правую границы отсортированной части массива.

Блок 3. Пока левая граница отсортированного массива не превосходит правую, выполняется тело цикла, состоящее из блоков 4–7.

Блок 4. Находится средний по положению элемент отсортированной части массива.

Блок 5. Значение x сравнивается с найденным элементом. По выходу «Да» блока 5 корректируется правая граница отсортированной части массива, по выходу «Нет» — левая.

При выходе из цикла с предусловием будет найдено положение вставляемого элемента.

Блоки 8–9 реализуют сдвиг элементов массива для вставки значения x.

Блок 10. Вставка значения в отсортированную часть массива.

Приведем выполнение алгоритма при сортировке массива 32 64 9 30 87 14 2 76.

i	*x*	*left*	*right*	*left* <= <= *right*?	*m*	*x* < *a*[*m*]?	*j*	*a*[*j* + 1]	*a*[*left*]
2	64	1	1	«Да»	1	32 < 64? «Да»			
			0	«Нет»			1	*a*[2] = 64	*a*[1] = 32
3	9	1	2	«Да»	1	9 < 64? «Да»			
			1	«Нет»			2	*a*[3] = *a*[2] = 64	
							1	*a*[2] = *a*[1] = 32	
									a[1] = 9
4	30	1	3	«Да»	2	30 < 32? «Да»			
			2	«Да»	1	30 < 9? «Нет»			
		2		«Нет»			3	*a*[4] = *a*[3] = 64	
							2	*a*[3] = *a*[2] = 32	*a*[2] = 30

После этого шага массив выглядит так: 9 30 32 64 87 14 2 76. Сортировка массива продолжается для следующих значений i.

Сортировка Шелла

Дальнейшим развитием метода сортировки включениями является сортировка методом Шелла, называемая по-другому сортировкой включениями с уменьшающимся расстоянием. Мы не будем описывать алгоритм в общем виде, ограничимся случаем, когда число элементов в сортируемом массиве является степенью числа 2. Для массива с 2^n элементами алгоритм работает следующим

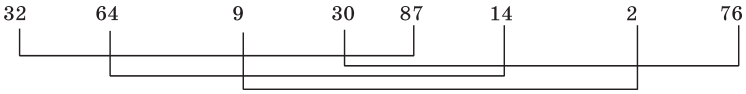
образом. На первом этапе производится сортировка включением всех пар элементов массива, расстояние между которыми равно 2^{n-1}. На втором этапе производится сортировка включением элементов полученного массива, расстояние между которыми равно 2^{n-2}. Алгоритм продолжается до тех пор, пока мы не дойдем до этапа с расстоянием между элементами, равным единице, и не выполним завершающую сортировку включениями.

Покажем сортировку Шелла на массиве, состоящем из 8 элементов.

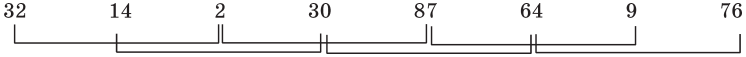
На первом проходе отдельно группируются и сортируются все элементы, отстоящие друг от друга на четыре позиции. Этот процесс называется 4-сортировкой. В нашем примере из восьми элементов каждая группа содержит ровно два элемента. После этого элементы вновь объединяются в группы с элементами, отстоящими друг от друга на две позиции, и сортируются заново. Этот процесс называется 2-сортировкой. Наконец, на третьем проходе все элементы сортируются обычной сортировкой, или 1-сортировкой.

Начальное состояние 32 64 9 30 87 14 2 76.

$n = 3$.



На первом шаге сравниваются элементы, отстоящие друг от друга на 4 позиции: $a[1]$ и $a[5]$, $a[2]$ и $a[6]$, $a[3]$ и $a[7]$, $a[4]$ и $a[8]$.



На втором шаге сравниваются элементы, отстоящие друг от друга на 2 позиции: $a[1]$ и $a[3]$, $a[2]$ и $a[4]$, $a[3]$ и $a[5]$, $a[4]$ и $a[6]$, $a[5]$ и $a[7]$, $a[6]$ и $a[8]$.

На этом шаге сравниваются все соседние элементы.

Отсортированный массив:

2 9 14 30 32 64 76 87.

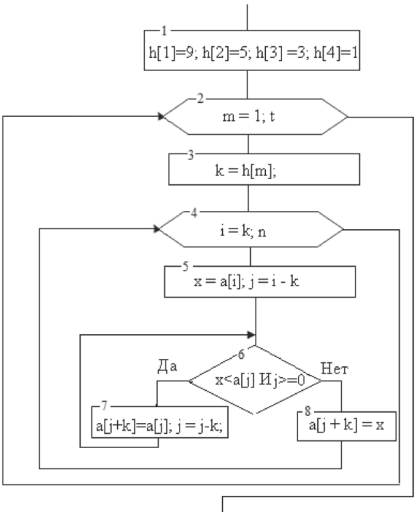
Таким образом, на каждом проходе либо участвуют сравнительно мало элементов, либо они уже довольно хорошо упорядочены и требуют относительно мало перестановок.

Очевидно, что этот метод в результате дает упорядоченный массив, и также совершенно ясно, что каждый проход будет использовать результаты предыдущего прохода, поскольку каждая i-сортировка объединяет две группы, рассортированные предыдущей 2^i-сортировкой. Также ясно, что приемлема любая последовательность приращений, лишь бы последнее было равно 1, так как в худшем случае вся работа будет выполняться на последнем проходе. Однако менее очевидно, что метод убывающего приращения дает даже лучшие результаты, когда приращения не являются степенями двойки [1].

Таким образом, алгоритм разрабатывается вне связи с конкретной последовательностью приращений. Все t приращения обозначаются через h_1, h_2, ..., h_t, с условиями $h_1 = 1$, $h_{i+1} < h_t$.

Каждая h-сортировка программируется как сортировка простыми включениями. Для того чтобы условие окончания поиска места включения было простым, используется барьер.

Ясно, что каждая h-сортировка требует собственного барьера, и программа должна определять его место как можно проще. Поэтому исходный массив нужно дополнить несколькими h-компонентами. Этот алгоритм называется сортировкой Шелла.



Приведем блок-схему алгоритма для $t = 4$. Примем $h[1] = 9$; $h[2] = 5$; $h[3] = 3$; $h[4] = 1$.

Исполнение алгоритма.

$$n = 8; \quad a = (32\ 64\ 9\ 30\ 87\ 14\ 2\ 76); \quad t = 4.$$
$$h[1] = 9; \quad h[2] = 5; \quad h[3] = 3; \quad h[4] = 1.$$

m	*k*	*i*	*x*	*j*	*j* <= 0Ux < *a*[*j*]	*a* [*j* + *k*]
1	9	9				
2	5	5	87	0	«Нет»	
		6	14	1	«Да»	*a*[6] = 32
				−4	«Нет»	*a*[1] = 14
		7	2	2	«Да»	*a*[7] = 64
				−1	«Нет»	*a*[2] = 2
		8	76	3	«Нет»	

После этой фазы массив выглядит так: 14 2 9 30 87 32 64 76.

В конечном счете, сортировка включениями оказывается не очень подходящим методом для ЭВМ, так как включение элемента с последующим сдвигом всего ряда элементов на одну позицию неэкономна.

1.5.2. Сортировка простым выбором

Сортировка простым выбором основана на следующем правиле.

В неупорядоченном массиве выбирается и отделяется от остальных элементов наименьший элемент. Наименьший элемент записывается на *i*-е место исходного массива, а элемент с *i*-го места — на место выбранного. Уже упорядоченные элементы (а они будут расположены начиная с первого места) исключаются из дальнейшей сортировки, поэтому длина оставшегося неупорядоченного массива должна быть на один элемент меньше предыдущего.

Улучшенный алгоритм — пирамидальная сортировка требует организации массива виде дерева. Желающих изучить этот алгоритм сортировки авторы отсылают к [1], где он прекрасно описан.

Сортировку простым выбором продемонстрируем на массиве 32 64 9 30 87 14 2 76.

$i = 1$, наименьшее значение 2, меняем местами 2 и 32. Массив после этого шага:

2 64 9 30 87 14 32 76.

$i = 2$, наименьшее значение в оставшейся части массива 9, меняем местами 9 и 64. Массив после этого шага:

2 9 64 30 87 14 32 76.

$i = 3$, наименьшее значение в оставшейся части массива 14, меняем местами 14 и 64. Массив после этого шага:

2 9 14 30 87 64 32 76.

$i = 4$, наименьшее значение в оставшейся части массива 30. Обмен не происходит, так как 30 стоит на своем месте.

$i = 5$, наименьшее значение в оставшейся части массива 32, меняем местами 32 и 87. Массив после этого шага:

2 9 14 30 32 64 87 76.

$i = 6$, наименьшее значение в оставшейся части массива 64. Обмен не происходит, так как 64 стоит на своем месте.

$i = 7$, наименьшее значение в оставшейся части массива 76, меняем местами 76 и 87. Массив после этого шага:

2 9 14 30 32 64 76 87.

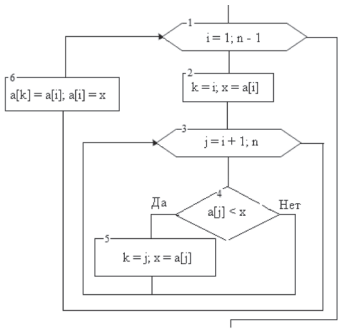
Массив отсортирован, но выполняется еще один шаг.

$i = 8$, наименьшее значение в оставшейся части массива 87. Обмен не происходит, так как 87 стоит на своем месте.

Таким образом, мы видим, что метод сортировки простым выбором основан на повторном выборе наименьшего элемента сначала среди n элементов, затем среди $(n - 1)$-го элемента и т. д. Возможна ситуация, когда обмен значений элементов не происходит.

Приведем блок-схему и описание алгоритма сортировки простым выбором.

Блок 1. Организует внешний цикл для просмотра элементов неупорядоченной части массива.



Блок 2. Запоминание индекса и значения i-го элемента, которые принимаем за начальные при поиске наименьшего элемента.

Блоки 3–5. Поиск значения наименьшего элемента и его номера в еще неупорядоченной части массива.

Блок 6. Обмен наименьшего и i-го элементов.

Приведем исполнение алгоритма для массива:

32 64 9 30 87 14 2 76; $n = 8$.

i	k	x	j	$a[j] < x$?	$a[k] = a[i]$; $a[i] = x$
1	1	32	2	a[2] < 32? «Нет»	
	3	9	3	a[3] < 32? «Да»	
			4	a[4] < 9? «Нет»	
			5	a[5] < 9? «Нет»	
			6	a[6] < 9? «Нет»	
	7	2	7	a[7] < 9? «Да»	A[7] = a[1] = 32; a[1] = 2
			кц		

После окончания внутреннего цикла массив имеет вид

2 64 9 30 87 14 32 76.

Сортировка продолжается для всех оставшихся значений параметра внешнего цикла i.

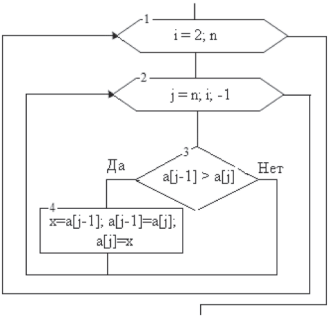
1.5.3. Обменные сортировки

Сортировка простым обменом

Простая обменная сортировка (называемая также «методом пузырька») для массива $a[1], a[2], \ldots, a[n]$ работает следующим образом. Начиная с конца массива, сравниваем два соседних элемента ($a[n]$ и $a[n-1]$). Если выполняется условие $a[n-1] > a[n]$, то значения элементов меняются местами. Процесс продолжается для $a[n-1]$ и $a[n-2]$ и т. д., пока не будет произведено сравнение $a[2]$

и $a[1]$. Понятно, что после этого на месте $a[1]$ окажется элемент массива с наименьшим значением. На втором шаге процесс повторяется, но последними сравниваются $a[3]$ и $a[2]$. На последнем шаге будут сравниваться только значения $a[n]$ и $a[n-1]$. Понятна аналогия с пузырьком, поскольку наименьшие элементы (самые «легкие») постепенно «всплывают» к верхней границе массива.

Простая реализация алгоритма.



Блок 1. Арифметический цикл. Значение параметра цикла i определяет количество сравниваемых элементов во внутреннем цикле.

Блок 2. Задание номеров элементов для сравнения.

Блок 3. Сравнение двух соседних элементов.

Блок 4. Выполняется обмен элементов массива при выходе «Да» блока 3.

Сортируем массив 32 64 9 30 87 14 2.

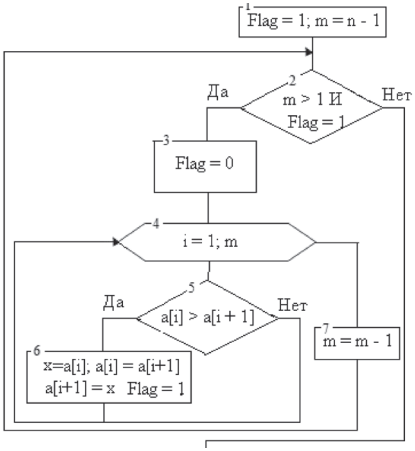
Выполнение алгоритма для $i = 2$ приведено ниже.

i	j	$a[j-1] > a[j]$?	$a[j-1]$, $a[j]$	Массив
2	8	2 > 76? «Нет»		32 64 9 30 87 14 2 76
	7	14 > 2? «Да»	$a[6] = 2$, $a[7] = 14$	32 64 9 30 87 **2** 14 76
	6	87 > 2? «Да»	$a[5] = 2$, $a[6] = 87$	32 64 9 30 **2** 87 14 76
	5	30 > 2? «Да»	$a[4] = 2$, $a[5] = 30$	32 64 9 **2** 30 87 14 76
	4	9 > 2? «Да»	$a[3] = 9$, $a[4] = 2$	32 64 **2** 9 30 87 14 76
	3	64 > 2? «Да»	$a[2] = 2$, $a[3] = 64$	32 **2** 64 9 30 87 14 76
	2	32 > 2? «Да»	$a[1] = 2$, $a[2] = 32$	**2** 32 64 9 30 87 14 76

После этого шага значение 2 встало на свое место, и алгоритм повторяется для $i = 3, 4, \ldots, n$.

Очевидный способ улучшить алгоритм — это запоминать, производился ли на данном проходе какой-либо обмен. Если нет, то алгоритм может закончить свою работу.

Используем переменную *Flag*, и пусть переменная $Flag = 1$, если на очередном шаге был выполнен обмен элементов, и $Flag = 0$, если обмена не было.



Приведем фрагмент блок-схемы и ее описание.

Блок 1. Задаем начальные значения $Flag = 1$ и $m = n - 1$.

Блок 2. Заголовок цикла. Пока не просмотрен весь массив и пока был обмен, выполняем внутренний цикл.

Блок 3. Предположим, что элементы массива уже отсортированы, тогда $Flag = 0$.

Блок 4. Организация арифметического цикла для сравнения и обмена соседних элементов массива.

Блоки 5–6. Тело внутреннего цикла. В блоке 5 выполняется сравнение соседних элементов.

При выходе «Да» блока 5 меняем местами элементы и фиксируем факт обмена в переменной *Flag*.

Блок 7. По окончании внутреннего цикла очередной элемент встал на свое место, уменьшается размер еще не отсортированной части массива.

Если был хотя бы один обмен, алгоритм повторяется.

Выполним алгоритм и отсортируем массив 32 64 9 30 87 14 2.

Flag	*m*	*m* > 1 и *Flag* = 1?	*i*	*a*[*i*] > *a*[*i* + 1]	*a*[*i*]	*a*[*i* + 1]
1	6	«Да»				
0			1	32 > 64? «Нет»		
1			2	64 > 9? «Да»	*a*[2] = 9	*a*[3] = 64
1			3	64 > 30? «Да»	*a*[3] = 30	*a*[4] = 64
			4	64 > 87? «Нет»		
1			5	87 > 14? «Да»	*a*[5] = 14	*a*[6] = 87
1			6	87 > 2? «Да»	*a*[6] = 2	*a*[7] = 87

В результате получим следующий массив: 32 9 30 64 14 2 87.

По выходе из внутреннего цикла $m = 6$ и $Flag = 1$, так как выполнялись обмены элементов массива. Начинаем внутренний цикл для нового значения переменной m.

Последний элемент находится на своем месте.

Продолжим сортировку для $m = 5$.

Flag	*m*	*m* > 1 и *Flag* = 1?	*i*	*a*[*i*] > *a*[*i* + 1]?	*a*[*i*]	a[i+ 1]
1		«Да»				
0			1	32 > 9? «Да»	*a*[1] = 9	a[2] = 32
1			2	32 > 30? «Да»	*a*[2] = 30	a[3] = 32
1	5		3	32 > 64? «Нет»		
			4	64 > 14? «Да»	*a*[4] = 14	a[5] = 64
1			5	64 > 2? «Да»	*a*[5] = 2	a[6] = 64
1			6	64 > 87? «Нет»		

Теперь массив имеет вид 9 30 32 14 2 64 87, и два последних элемента находятся на своих местах.

Так как были перестановки элементов, то алгоритм продолжается до полной сортировки массива.

Второе улучшение алгоритма связано с тем, что можно установить барьер: запоминать наименьшее значение индекса массива, для которого на текущем шаге выполнялись перестановки. Очевидно, что верхняя часть массива до элемента с этим индексом уже отсортирована, и на следующем шаге можно прекращать сравнения значений соседних элементов при достижении такого значения индекса.

В-третьих, метод пузырька работает неравноправно для так называемых «легких» и «тяжелых» значений. Один неправильно расположенный «пузырек» в «тяжелом» конце рассортированного массива всплывет на место за один проход, а неправильно расположенный элемент в «легком» конце будет опускаться на правильное место только за один шаг на каждом проходе. Например, массив

12 18 42 44 55 67 94 6

будет рассортирован за один проход, а сортировка массива

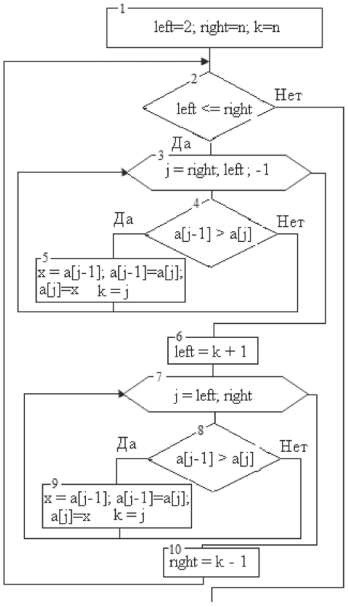
94 6 12 18 43 44 55 76

потребует семи проходов. Эта асимметрия подсказывает третье улучшение: менять местами направление следующих один за другим проходов. Этот улучшенный алгоритм называется шейкер-сортировкой.

Шейкер-сортировка

При ее применении на каждом следующем шаге меняется направление последовательного просмотра. В результате на одном шаге «всплывает» очередной наиболее легкий элемент, а на другом «тонет» очередной самый тяжелый.

Обозначим *left* — номер левой границы сортируемой части массива, *right* — номер его правой границы.



Блок 1. Установка номеров начальных границ сортируемого массива.

Блок 2. Вход в цикл. Пока левая граница не превосходит правую границу (выход «Да» блока 2) выполняем цикл.

Блок 3. Выполнение прохода массива вниз.

Блок 4. Сравнение соседних элементов.

Блок 5. Если $a[j-1] > a[j]$ (выход «Да» блока 4), производим обмен этих элементов и фиксируем номер элемента j, с которым производился обмен.

По окончании прохода вниз (выход блока 3) сдвигаем левую границу массива (блок 6) и выполняем проход вверх (блоки 7–9).

Покажем выполнение алгоритма для массива, состоящего из 7 элементов:

32 9 30 64 14 2 87.

Выполняем проход сверху вниз.

left	*right*	*k*	*left* <= *right*?	*j*	*j* > *left*?	*a*[*j* − 1] > *a*[*j*]?	Обмен
2	7	7	«Да»	7	«Да»	2 > 87 «Нет»	
		6		6	«Да»	14 > 2 «Да»	*x* = 14; *a*[5] = 2; *a*[6] = 14
		5		5	«Да»	64 > 2 «Да»	*x* = 64; *a*[4] = 2; *a*[5] = 64
		4		4	«Да»	30 > 2 «Да»	*x* = 30; *a*[3] = 2; *a*[4] = 30
		3		3	«Да»	9 > 2 «Да»	*x* = 9; *a*[2] = 2; *a*[3] = 9
		2		2	«Да»	32 > 2 «Да»	*x* = 32; *a*[1] = 2; *a*[2] = 32
				1	«Нет»		

Массив после этого прохода имеет вид 2 32 9 30 64 14 87.

Меняем направление движения и выполняем проход снизу вверх.

left	*right*	*k*	*j*	*j* <= *right*	*a*[*j* − 1] > *a*[*j*]?	
3	7	3	3	«Да»	32 > 9? «Да»	*x* = 32; *a*[2] = 9; *a*[3] = 32
		4	4	«Да»	32 > 30? «Да»	*x* = 32; *a*[3] = 30; *a*[4] = 32
			5	«Да»	32 > 64? «Нет»	
		6	6	«Да»	64 > 14? «Да»	*x* = 64; *a*[5] = 14; *a*[6] = 64
			7	«Да»	64 > 87? «Нет»	
			8	«Нет»		

Теперь массив имеет вид 2 9 30 32 14 64 87.

Опять меняем направление движения. Смена направления движения выполняется до тех пор, пока выполняется условие, записанное в блоке 2.

Анализ показывает [1], что сортировка обменом и ее небольшие улучшения хуже, чем сортировка включениями и выбором. Алгоритм шейкер-сортировки рекомендуется использовать в тех случаях, когда известно, что массив «почти упорядочен».

В 1962 г. Чарльз Хоар предложил метод сортировки разделением. Этот метод является развитием метода простого обмена и настолько эффективен, что его стали называть методом быстрой сортировки — QuickSort.

Метод требует использования рекурсивных функций и в рамках данного пособия не рассматривается.

1.5.4. Сравнение методов сортировки

На основании анализа, приводимого в [1], можно сделать следующие выводы об эффективности рассмотренных алгоритмов сортировки.

1. Преимущество сортировки бинарными включениями по сравнению с сортировкой простыми включениями мало, а в случае уже имеющегося порядка вообще отсутствует.

2. Сортировка методом «пузырька» является наихудшей среди сравниваемых методов. Ее улучшенная версия — шейкер-сортировка — все-таки хуже, чем сортировка простыми включениями и простым выбором.

3. Сортировка простым выбором является лучшим из простых методов.

1.6. АЛГОРИТМЫ ПОИСКА

Алгоритмы поиска занимают очень важное место среди прикладных алгоритмов, и это утверждение не нуждается в доказательстве.

Все алгоритмы поиска разбиваются на две большие группы в зависимости от того, упорядочен или нет массив данных, в котором проводится поиск. Рассмотрим простые алгоритмы поиска заданного элемента в одномерном массиве данных.

1.6.1. Последовательный поиск

Наиболее примитивный, а значит, наименее эффективный способ поиска — это обычный последовательный просмотр массива.

Пусть требуется найти элемент X в массиве из n элементов. Значение элемента X вводится с клавиатуры.

В данном случае известно только значение разыскиваемого элемента, никакой дополнительной информации о нем или о массиве, в котором его надо искать, нет. Поэтому для решения задачи разумно применить очевидный метод — последовательный перебор элементов массива и сравнение значения очередного элемента с заданным образцом.

Пусть $Flag = 1$, если значение, равное X, в массиве найдено, в противном случае $Flag = 0$. Обозначим k — индекс найденного элемента. Элементы массива — целого типа.

Элементы массива

Присвоим начальные значения переменным $Flag$ и k и возьмем первый элемент массива (блок 1).

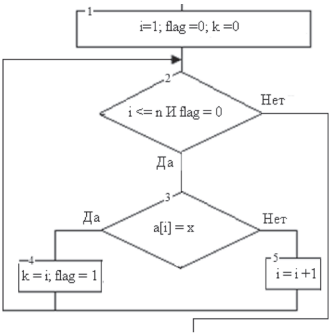
Пока не просмотрены все элементы и пока элемент, равный X, не найден (выход «Да» блока 2), выполняем цикл.

Сравниваем очередной элемент массива со значением X, и если они равны (выход «Да» блока 3), то запоминаем его индекс и поднимаем $Flag = 1$. Если элемент еще не найден (выход «Нет» блока 2), продолжаем просмотр элементов массива.

Выход из цикла (блок 2) возможен в двух случаях:

1) элемент найден, тогда $Flag = 1$, а в переменной k сохранен индекс найденного элемента;

2) элемент не найден, тогда $Flag = 0$, а k также равен 0.



1.6.2. Бинарный поиск

Если известно, что массив упорядочен, то можно использовать бинарный поиск.

Пусть даны целое число X и массив размера n, отсортированный в порядке неубывания, т. е. для любого k ($1 <= k < n$) выполняется условие $a[k - 1] <= a[k]$. Требуется найти такое i, что $a[i] = X$, или сообщить, что элемента X нет в массиве.

Идея бинарного поиска состоит в том, чтобы проверить, является ли X средним по положению элементом

массива. Если да, то ответ получен. Если нет, то возможны два случая.

1. X меньше среднего по положению элемента, следовательно, в силу упорядоченности массива можно исключить из рассмотрения все элементы массива, расположенные правее этого элемента, так как они заведомо больше его, который, в свою очередь, больше X, и применить этот метод к левой половине массива.

2. X больше среднего по положению элемента, следовательно, рассуждая аналогично, можно исключить из рассмотрения левую половину массива и применить этот метод к его правой части.

Средний по положению элемент и в том, и в другом случае в дальнейшем не рассматривается. Таким образом, на каждом шаге отсекается та часть массива, где заведомо не может быть обнаружен элемент X.

Пусть $X = 6$, а массив состоит из 10 элементов:

3 5 6 8 12 15 17 18 20 25.

1-й шаг. Найдем номер среднего по положению элемента: $m = [(1 + 10) / 2] = 5$. Так как $6 < a[5]$, то далее можем рассматривать только элементы, индексы которых меньше 5. Об остальных элементах можно сразу сказать, что они больше, чем X, вследствие упорядоченности массива и среди них искомого элемента нет.

3 5 6 8 **12** 15 17 18 20 25.

2-й шаг. Сравниваем лишь первые 4 элемента массива; значение $m = [(1 + 4) / 2] = 2$. $6 > a[2]$, следовательно, рассматриваем правую часть подмассива, а первый и второй элементы из рассмотрения исключаются:

3 **5** *6 8* 12 15 17 18 20 25.

3-й шаг. Сравниваем два элемента; значение $m = [(3 + 4) / 2] = 3$:

3 5 **6** *8* 12 15 17 18 20 25.

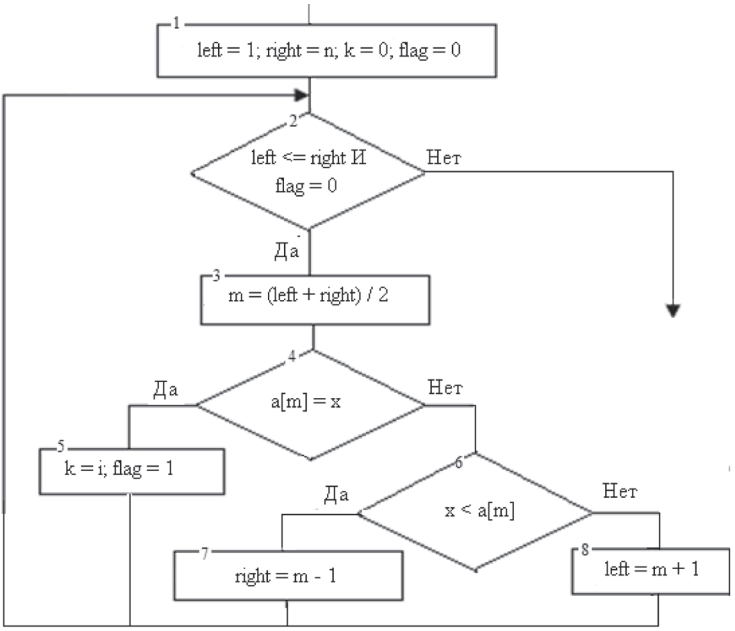
$a[3] = 6$. Элемент найден, его номер — 3.

В общем случае значение *m* равно целой части от дроби **[(*left* + *right*) / 2]**, где *left* — индекс первого, а *right* — индекс последнего элемента рассматриваемой части массива.

Если *Flag* = 1, то нужное значение в массиве найдено, а если *Flag* = 0, то значения, равного *X*, в массиве нет.

Алгоритм бинарного поиска можно применять только для упорядоченного массива. Это происходит потому, что данный алгоритм использует тот факт, что индексами элементов массива являются последовательные целые числа. По этой причине бинарный поиск практически бесполезен в ситуациях, когда массив постоянно меняется в процессе решения задачи. Алгоритм также используется при сортировке массивов методом бинарного включения (см. п. 1.5.1).

Ниже приведен фрагмент блок-схемы алгоритма бинарного поиска.



ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Дайте определение алгоритма. Перечислите свойства алгоритма.
2. Назовите отличия программного способа записи алгоритмов от других способов.
3. Назовите базовые алгоритмические структуры и дайте им краткую характеристику.
4. Дайте определение цикла с заданным числом повторений. Когда целесообразно применять циклы этого вида?
5. Что такое итерационные циклы? Когда возникает необходимость в их использовании?
6. Определите основные отличия между циклами с постусловием и предусловием. Как они выполняются?
7. Что называется рекуррентной формулой? Когда она применяется?
8. Дайте определение массива. Поясните, почему для хранения его элементов используется непрерывная память.
9. Можно ли при вводе или выводе элементов массива использовать цикл с предусловием или с постусловием?
10. Укажите, как изменится алгоритм нахождения наибольшего значения (пример 1.10), если все элементы массива — отрицательные числа.
11. Если известно, что в массиве обязательно имеется отрицательный элемент, то как изменится алгоритм решения задачи 1.14?
12. Дайте определение двумерного массива. Поясните особенности хранения элементов двумерного массива.
13. Почему при составлении алгоритмов для решения задач с использованием двумерного массива применяется вложенный цикл?
14. Можно ли при решении задачи 1.19 использовать цикл с постусловием? Ответ поясните.
15. Перечислите простые алгоритмы сортировки и укажите их основные отличия.
16. Почему сортировка включениями является неэкономным методом?
17. Какими характеристиками должен обладать массив, чтобы применение шейкер-сортировки было эффективным?
18. Чем сортировка Шелла отличается от сортировки простыми вставками?
19. Назовите метод сортировки, который является лучшим среди простых методов. Поясните, за счет чего это достигается.
20. Почему алгоритм бинарного поиска превосходит «слепой» поиск? Какими характеристиками должен обладать массив, в котором применяется алгоритм бинарного поиска?

ГЛАВА 2

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ C++

Программа — это алгоритм, записанный на каком-либо языке программирования (формальном языке).

Программа предназначена для обработки данных (управления данными). В процессе работы приложения сама программа (или ее часть) и данные (или их часть) находятся в оперативной памяти компьютера (ОЗУ).

Язык программирования является инструментом для написания программ. Языки программирования, в том числе C++, относятся к *формальным языкам*. Задачей формальных языков является описание алгоритмов и функций, которые выполняет компьютер. Формальные языки отличает небольшой набор изобразительных средств соответственно целям этих языков. В состав системы правил алгоритмического языка входят две группы.

1. *Синтаксис* — формальный набор правил, определяющий способ построения любых конструкций языка.

2. *Семантика* — множество правил, определяющих смысл синтаксических конструкций.

Алгоритм решения задачи записывается программистом согласно синтаксическим правилам языка, и это программа. Семантика реализована в компиляторе, который переводит код во внутреннее представление. Компилятор обнаруживает синтаксические ошибки, допущенные в коде программы.

Для разработки приложений в современном программировании используются интегрированные среды разработчика (Integrated Developer Environment — IDE, или

Rapid Application Development — RAD), которые позволяют программировать быстро и надежно. Современные среды разработчика являются многоязыковыми и позволяют в одном проекте кодировать на разных языках. Так, Visual Studio.Net поддерживает языки C++, C#, Basic и Fortran#.

Обязательными этапами обработки кода программы на языке C++ являются:

- препроцессорное преобразование;
- компиляция;
- компоновка.

Задачей *препроцессора* является преобразование текста программы до ее компиляции. Препроцессор сканирует текст, находит в нем команды, называемые директивами препроцессора, и выполняет их. В результате происходят изменения в исходном тексте: вставка фрагментов текста и замена некоторых его фрагментов. Полученный текст называется полным текстом программы.

На этапе *компиляции* полный код программы преобразуется во внутреннее машинное представление, некоторую последовательность команд, которая понятна компьютеру. Компилятор находит ошибки несоответствия кода программы синтаксическим правилам языка.

На этапе *компоновки* происходит редактирование связей и сборка исполнимого кода программы. Компоновщик обрабатывает все вызовы библиотечных функций и выполняет их подключение. Таким образом, к компилированному исходному коду добавляются необходимые функции стандартных библиотек. Готовый код является исполнимым и может быть выполнен компьютером.

2.1. НАЧАЛЬНЫЕ СВЕДЕНИЯ О ЯЗЫКЕ ПРОГРАММИРОВАНИЯ C++

Чтобы писать хорошие программы, недостаточно знать правила записи синтаксических конструкций, необходимо отчетливо представлять себе механизмы, встроенные в реализацию компилятора, которые, собственно, и являются той самой семантикой.

2.1.1. Алфавит языка С++ и лексемы

Алфавит — набор символов, разрешенных для построения синтаксических конструкций. Алфавит языка С++ содержит четыре группы символов.

1. *Буквы*. Разрешается использовать буквы латинского алфавита, прописные (A–Z) и строчные (a–z). Русские буквы не входят в алфавит, но используются в комментариях и текстовых константах, там, где они не влияют на смысл программы.

2. *Цифры*. Используются арабские цифры 0, 1, …, 9.

3. *Специальные символы*. Они могут быть разделены на подгруппы:

- знаки препинания (разделители): , ; . : ;
- знаки операций: +, –, *, /, % , &, |, ?, !, <, =, >;
- парные скобки: [ ], { }, ( ), " ", ' ';
- прочие символы: _, #, ~, ^.

4. *Невидимые символы*. Они могут считаться разделителями, их особенность в том, что символы существуют (каждый имеет код), но в редакторе не видны. Это такие символы, как пробел, табуляция, разделитель строк. Их общее название — обобщенные пробельные символы.

Из символов алфавита строятся все конструкции языка.

Лексема — это единица текста (конструкция, слово), воспринимаемая компилятором как единое неделимое целое. Можно выделить пять классов лексем.

1. *Имена* (идентификаторы) для именования произвольных объектов программы, например, x1, Alpha, My_file.

2. *Служебные* (ключевые) слова, обозначающие конструкции языка (имена операторов), например, for, while, do.

3. *Константы*, например, 1, 12.5, “Василий”.

4. *Операции* (знаки операций), например, ++, >=, !=, >.

5. *Разделители* (знаки пунктуации), например, [ ], ( ), { }.

2.1.2. Концепция данных в языке C++

Данные — все, что подлежит обработке с помощью программы. Данные, используемые приложением, должны быть описаны в программном коде. Классификацию данных можно выполнять по нескольким категориям.

1. *По типу*. Каждое данное имеет тип. Типы данных можно разделить на *базовые*, т. е. такие, правила организации которых предопределены реализацией языка, и *конструируемые*, т. е. те, которые пользователь строит по определенным правилам для конкретной задачи.

2. *По способу организации*. Для каждого из простых (базовых) типов каждое данное может быть неизменяемым или изменяемым. По способу организации данные делятся на два класса:

- *константа* — данное, которое не меняет своего значения при выполнении программы и присутствует в тексте программы явным образом. Тип константы определен ее записью;
- *переменная* — данное, которое изменяется при выполнении программы и в тексте присутствует своим именем (идентификатор). Тип переменной величины должен быть объявлен в тексте программы.

Каждое данное, независимо от способа организации, обладает типом. Тип данного очень важен, так как он определяет:

- *механизм* выделения памяти для записи значений переменной;
- *диапазон* значений, которые может принять переменная;
- *список* операций, которые разрешены над данной переменной.

Механизм выделения памяти и внутреннего представления данного важнее всего, диапазон значений и операции являются следствием из первого.

Тип любого данного программы должен быть объявлен обязательно. Для констант тип определен явно записью константы в тексте программы. Для переменных тип должен быть задан в объявлении объекта. При объявле-

нии переменной происходит выделение области памяти, размер которой определен типом переменной и в которой хранится значение переменной.

Классификация типов данных:

- *простые типы* (базовые, скалярные, внутренние, предопределенные);
- *конструируемые типы* (массивы, строки, структуры, функции и др.).

Основные типы данных определены ключевыми словами, список которых приведен в таблице 2.1.

Таблица 2.1

Ключевые слова, определяющие основные типы данных

Имя типа	Значение типа	Размер	Диапазон значений
char	Символьный (целое)	1 байт	–128–127
int	Целый	4 байта	±2 147 483 649
float	С плавающей точкой (действительный)	4 байта	3.4e-38–3.4e38
double	Двойной точности (действительный)	8 байт	1.7e-308–1.7e308
void	Любой или никакой		Не ограничен

Замечания

1. В языке С++ для определения размера памяти, занимаемой объектом, есть операция sizeof(), например, sizeof(int); sizeof(double); sizeof(My_obj).

В качества аргумента операции можно указать имя типа (int, double) или имя объекта (My_obj).

2. Существует специальный тип данных void, который относится к базовым, но имеет существенные особенности. Про данное такого типа говорят, что оно «не имеет никакого типа». На самом деле тип void используется при работе с динамическими данными и может адресовать пространство произвольного размера, где можно разместить данное любого типа. Значит, можно считать, что данное типа void может представлять «данное любого типа».

3. Основные типы данных можно изменить (модифицировать) с использованием вспомогательных ключевых слов long и unsigned:

- long (длинный) увеличивает объем выделяемой памяти в 2 раза;
- unsigned (без знака) не использует знаковый бит, за счет чего хранимое значение может быть увеличено.

2.1.3. Константы в языке C++

Синтаксис языка выделяет пять типов констант: целые, действительные (вещественные), символьные, перечислимые, нулевой указатель.

Целые константы

Синтаксис языка позволяет использовать константы трех систем счисления: десятичные, восьмеричные, шестнадцатеричные. Основание определяется префиксом в записи константы. По умолчанию основание 10, префикс 0 предваряет восьмеричную константу, префикс 0x или 0X предваряет шестнадцатеричную константу. В остальном запись целых констант соответствует общепринятой.

Примеры записи целых констант приведены в таблице 2.2.

Таблица 2.2

Примеры записи целых констант

Десятичные	Восьмеричные	Шестнадцатеричные
127 –256	012 –014	0xA –0x10

Целые числа в памяти компьютера представлены в форме с фиксированной точкой. Эта форма позволяет представить значение с абсолютной точностью. На рисунке 2.1 показано представление целого числа в двухбайтовой ячейке памяти.

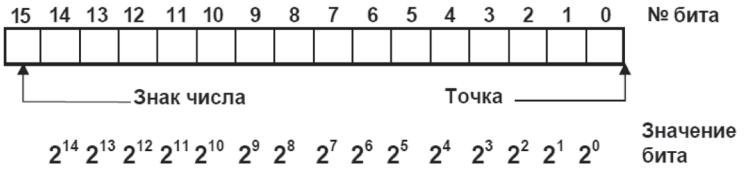


Рис. 2.1
Представление целых чисел

Действительные константы

Вещественные (действительные) числа представлены в памяти компьютера в форме с плавающей точкой в виде $M \cdot 10^p$, где M — мантисса вещественного числа; 10 — основание системы счисления; p — показатель десятичной степени, целое число со знаком. Основание системы счисления в языке С++ заменяется буквой *e* или *E*. Некоторые из составляющих могут быть опущены.

Примеры записи действительных констант:

44. 3.1415926 44.e0 .31459E1 0.0

На рисунке 2.2 показано представление вещественного числа в четырехбайтовой ячейке памяти.

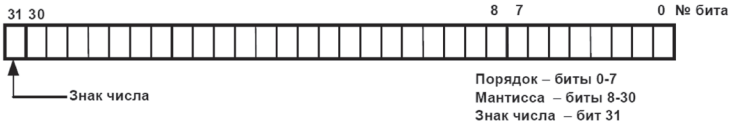


Рис. 2.2
Представление вещественных чисел

Символьные константы

Символьная константа — это лексема, которая содержит произвольный символ, заключенный в одинарные кавычки (апостроф). Хранится в одном байте памяти в виде целого числа, определяющего целочисленный код символа. Символьной константой может быть любой символ, изображаемый на экране в текстовом режиме, или не имеющий отображения (пробельный символ).

Примеры символьных констант: '!', '.', 's', 'A', '+', '2'.

Существуют также управляющие символы (*ESC*-последовательности), которые используются в строковых константах и переменных. Они имеют признак '\' (backslash или обратный слеш), который используется для указания управляющего воздействия. Вот их неполный список:

'\0' — нулевой символ;
'\n' — перевод строки;
'\r' — возврат каретки;

'\t' — табуляция;

'\' — апостроф;

'\\' — обратный слэш;

'\ddd' — восьмеричное представление символьной константы, где ddd — ее восьмеричный числовой код, например, '\017' или '\233';

'\xhh' или '\Xhh' — шестнадцатеричное представление символьной константы, где hh — ее числовой код, например, '\x0b' или '\x1F'.

Символьные константы имеют целый тип и могут использоваться в выражениях.

Строковые константы

Формально строки не относятся к константам языка C++, а представляют отдельный тип его лексем. Строковая константа определяется как набор символов, заключенных в двойные кавычки " ".

Например,

```
#define STR "Программа"
```

В конце строковой константы добавляется нулевой символ '\0'. В тексте строки могут встретиться *ESC*-последовательности, которые будут выполнять управляющее воздействие.

Например,

```
"\nПри выводе \nтекст может быть \nразбит на строки."
```

Здесь "\n" выполняет управляющее воздействие, и вывод будет таким:

```
При выводе
текст может быть
разбит на строки.
```

При хранении строки все символы, в том числе управляющие, хранятся последовательно, и каждый занимает ровно один байт.

Типы числовых констант и модификаторы типов

Для модификации типа числовых констант используются префиксы и суффиксы, добавляемые к значению константы. Суффикс l(L) от слова long увеличивает длину данного в два раза, суффикс u(U) от слова unsigned

используется для чисел, не содержащих знак, т. е. положительных, при этом увеличивается число разрядов, отводимых под запись числа, а значит, и его возможное значение.

Например, длинные константы 12345l, –54321L; константы без знака 123u, 123U; шестнадцатеричная длинная константа 0xb8000000l; восьмеричная длинная константа без знака 012345LU.

2.1.4. Переменные в языке С++

Одним из основных понятий в языках программирования является объект. Будем использовать это понятие как объект программного кода.

Объект — это некоторая сущность, обладающая определенным набором свойств и способная хранить свое состояние.

Объектами программного кода являются константы, переменные, функции, типы и т. д.

Переменная — это объект, для использования которого необходимо определить три характеристики:

- имя;
- тип;
- значение (не обязательно).

Имя переменной (идентификатор) обозначает в тексте программы величину, изменяющую свое значение. Для каждой переменной выделяется некоторая область памяти, способная хранить ее значение. Имя позволяет осуществить доступ к области памяти, хранящей значение переменной.

Тип переменной определяет размер выделяемой памяти и способ хранения значения (см. рис. 2.1, 2.2).

Значение переменной не определено вначале, изменяется при присваивании переменной значения.

Имя переменной (идентификатор) включает в себя латинские буквы, цифры, знак подчеркивания "_". Первым символом должна быть буква.

Примеры идентификаторов: x1, Price, My_file1, alpha, PI. В качестве имен рабочих переменных используются простые имена i, j, k1, k2 и им подобные.

Ограничения:

- длина имени содержит не более 32-х символов;
- в качестве имен нельзя использовать ключевые слова языка C++, например, названия операторов for, if, do;
- в качестве имен не рекомендуется использовать имена объектов стандартных библиотек, например, имена функций sin(), sqrt(), pow();
- имена, которые начинаются со знака подчеркивания, зарезервированы для использования в библиотеках и компиляторах, поэтому их не следует выбирать в качестве прикладных имен, например, _acm, _AX, _BX.

Перечень основных, не всех служебных слов приведен в таблице 2.3.

Таблица 2.3

Служебные слова в языке C++

Типы данных	char, double, enum, float, int, long, short, struct, signed, union, unsigned, void, typedef
Квалификаторы типа	const, volatile
Классы памяти	auto, extern, register, static
Названия операторов	break, continue, do, for, goto, if...else, return, switch, while, case, default, sizeof
Модификаторы	asm, far, interrupt, pascal и др.
Псевдопеременные	Названия регистров _AH _DS _DX и др.

Замечания

1. Рекомендуется использовать имена переменных, отражающие их первоначальный смысл, например, Count (количество), X_coord (координата по x), Old_value_of_long (предыдущее значение длины).

2. Заглавные и строчные буквы считаются различными. Так, разными объектами будут переменные, имеющие имена Alpha и alpha.

3. Идентификаторы, зарезервированные в языке, являются служебными (ключевыми) словами. Следовательно, их нельзя использовать как имена свободных объектов программы. К ним относятся типы данных, квалификаторы типа, классы памяти, названия операторов, модификаторы, псевдопеременные.

2.1.5. Объявление переменных

Каждая переменная должна быть объявлена в тексте программы перед первым ее использованием. Начинающим программистам рекомендуется объявлять переменные в начале тела программы перед первым исполнимым оператором. Синтаксис объявления переменной требует указать ее тип и имя:

Имя_типа Имя_переменной;

Можно одновременно объявить несколько переменных, тогда в списке имен они отделяются друг от друга запятой, например:

```
int a, b, c;              // целые со знаком
char ch, sh;              // однобайтовые символьные
long l, m, k;             // длинные целые
float x, y, z;            // вещественные
long double u, v          // длинные двойные
```

На этапе компиляции для объявленных переменных в ОЗУ будет выделена память для записи их значений. Имя переменной определяет адрес выделенной для нее памяти, тип переменной определяет способ ее хранения в памяти. Как следствие, тип определяет диапазон ее возможных значений и операции, разрешенные над этой величиной.

2.1.6. Инициализация переменных

Переменные, объявленные в программе, не имеют значения, точнее имеют неопределенные значения. Так как память, выделенная под запись переменных, не очищается, то значением переменной будет «мусор» из того, что находилось в этой памяти ранее.

Инициализация — это прием, который позволяет присвоить значения переменным при их объявлении. Синтаксис инициализации:

Имя_типа Имя_переменной = Начальное_значение;

Например,

```
float pi=3.1415;
unsigned Year=2014;
```

2.1.7. Именованные константы

В языке C++, кроме переменных, можно использовать *именованные* константы, т. е. константы, имеющие фиксированные названия (имена). Имена могут быть произвольными идентификаторами, не совпадающими с другими именами объектов. Принято использовать в качестве имени константы большие буквы и знаки подчеркивания, что визуально отличает имена переменных от имен констант. Способов определения именованных констант три.

Первый способ — это константы перечисляемого типа, имеющие синтаксис:

enum тип_перечисления {список_именованных_констант};

Здесь «тип_перечисления» — это его название, необязательный произвольный идентификатор; «список_именованных_констант» — это разделенная запятыми последовательность вида:

имя_константы = значение_константы

Примеры.

```
enum BOOL {FALSE, TRUE};
enum DAY {SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
                    THURSDAY, FRIDAY, SATURDAY};
enum {ONE=1, TWO, THREE, FOUR, FIVE};
```

Если в списке нет элементов со знаком «=», то значения констант начинаются с 0 и увеличиваются на 1 слева направо. Так, FALSE равно 0, TRUE равно 1. SUNDAY равно 0 и так далее, SATURDAY равно 6. Если в списке есть элементы со знаком «=», то каждый такой элемент получает соответствующее значение, а следующие за ним увеличиваются на 1 слева направо, так, значение ONE = 1, TWO = 2, THREE = 3, FOUR = 4, FIVE = 5.

Второй способ — это именованные константы, заданные с ключевым словом const в объявлении типа:

const Имя_типа Имя_константы = Значение_константы;

Ключевое слово const имеет более широкий смысл и используется также в других целях, но здесь его суть в том, что значение константы не может быть изменено.

Примеры.

```
const float Pi=3.1415926;
const long M=999999999L;
```

Попытка присвоить значение любой из констант Pi, M будет неудачной. Визуализация имен констант с помощью использования больших букв необязательна.

Третий способ ввести именованную константу дает препроцессорная директива #define, подробный синтаксис и механизм выполнения которой будет рассмотрен далее.

2.2. ОПЕРАЦИИ И ВЫРАЖЕНИЯ ЯЗЫКА С++

Операция — это символ или лексема вычисления значения. Операции используются для формирования и вычисления значений выражений и для изменения значений данных.

2.2.1. Классификация операций

Классифицировать операции можно по нескольким признакам. Приведем два варианта: по типу возвращаемого значения и по числу операндов.

1. *Тип возвращаемого значения* является важным свойством любой операции. В таблице 2.4 приведена классификация операций по типу возвращаемого значения.

Таблица 2.4

Классификация операций по типу возвращаемого значения

Арифметические	Логические
+ сложение	> больше
– вычитание	< меньше
* умножение	== равно
/ деление (для целых операндов — целая часть от деления)	!= не равно
% остаток от деления (только для целых)	>= больше или равно
	<= меньше или равно
	&& логическое «И»
	‖ логическое «ИЛИ»
	! логическое «НЕ»

Арифметические операции имеют обычный математический смысл. Для целого типа данных существуют две операции деления: целочисленное деление / и остаток от деления %. Каждая из этих операций, примененная к данным целого типа, возвращает целое значение.

Пример 2.1.

```
int x=5, y=3;
x/y             // результат = 1
x%y             // результат = 2
y/x             // результат = 0
y%x             // результат =3
x=–5            // знак сохраняется
x/y             // результат=–1
x%y             // результат= –2
y/x             // результат=0
y%x             // результат=3
```

Логические операции предназначены для проверки условий и для вычисления логических значений.

Замечание. Логического типа данных в стандарте языка C++ нет, его имитирует тип int (условно принято, что 0 — это значение «Ложь», а все отличные от нуля значения — это «Истина»). Компилятор Visual Studio 2013 в качестве логического типа позволяет использовать тип bool, например:

```
bool yes;
yes=x>0;
```

Логическими являются операции, возвращающие логическое значение. Их можно, в свою очередь, разбить на две группы.

Операции отношения. Связывают данные числовых или символьных типов и возвращают логическое значение, например:

```
3>1             // истина, не равно 0
'a'>'b'         // ложь равна 0
x>=0            // зависит от x
y!=x            // зависит от x и от y
```

Значение операции отношения может быть присвоено целой переменной или переменной логического типа, например:

```
int a=5, b=2;
bool c;
c=a>b;          // c = 1
c=a<b;          // c = 0
c=a==b;         // c =0
c=a!=b;         // c = 1
```

Операции отношения могут участвовать в записи выражения наряду с арифметическими операциями, тогда при вычислении значения выражения важен порядок вычисления, например:

```
c=10*(a>b);       // a > b = 1, 10 * 1 = 10, значит, c = 10
c=10*(10*a<b); // 10 * a = 50, 50 < b = 0, 10 * 0 = 0, c =0
c=1<a&&a<10;  // 1 < a = 1, a < 10 = 1, значит, c = 1
```

Собственно логические операции. Связывают данные логического типа (целые) и возвращают логическое значение. Это логические операции конъюнкция &&, дизъюнкция || и отрицание !, которые применяются к значениям операндов, имеющих логическое значение. Если операндом является отношение, то предварительно вычисляется его значение, например:

```
// оба операнда (x>0 и y>0) одновременно истинны
                                              x>0 && y>0;
// хотя бы один операнд (x % 2==0 или y % 2==0) истинен
x%2==0||y%2==0;
// значение, обратное значению операнда
// (x*x+y*y<=r*r)
!(x*x+y*y<=r*r);
```

2. Второй вид классификации операций языка С++ — классификация по *числу операндов*.

1) *Унарные* операции имеют только один операнд, например:

```
+10, –a, !(x>0), ~a.
```

2) *Бинарные* операции имеют два операнда, например:

```
x+5, x–y, b&&c.
```

3) *Тернарная* операция имеет три операнда, это операция *условия*, приведем ее синтаксис здесь:

```
Логическое_выражение ? Выражение1 : Выражение2
```

Операция *условия* выполняется в два этапа. Сначала вычисляется значение логического выражения. Затем, если оно истинно, вычисляется значение второго операнда, который назван «Выражение1», оно и будет результатом, иначе вычисляется значение третьего операнда, который назван «Выражение2», и результатом будет его значение.

Например, вычисление абсолютного значения некоторого *i*:

```
int abs;
abs=(i<=0)?–i:i;
```

2.2.2. Операции изменения данных

Операции изменения данных имеют особое значение, позволяя выполнить присваивание значения переменной.

Список операций изменения данных приведен в таблице 2.5.

Таблица 2.5

Операции изменения данных

Основные операции	Арифметические операции с присваиванием
= операция присваивания	+= сложение с присваиванием
++ увеличение на единицу	−= вычитание с присваиванием
−− уменьшение на единицу	*= умножение с присваиванием
	/= деление с присваиванием
	%= остаток от деления с присваиванием

Операция присваивания в C++ — это обычная операция, которая является частью выражения, но имеет очень низкий приоритет и поэтому выполняется в последнюю очередь. Семантика этой операции заключается в том, что вычисляется выражение правой части и присваивается левому операнду. Это означает, что слева может быть только переменная величина, которая способна хранить и изменять свое значение (левостороннее выражение). В правой части, как правило, записывается вычисляемое выражение.

Пример 2.2.

```
x=2;                  // x принимает значение константы
cond=(x<=2);          // cond принимает логическое
                      // значение
3=5;                  // ошибка, слева константа
x=x+1;                // x принимает значение,
                      // увеличенное на 1
```

Расширением операции присваивания являются операции вычисления с присваиванием, которые еще называются «присваивание после» и позволяют сократить запись выражений, например: