

Оглавление

1. Понятие алгоритма.....	4
1.1. Виды алгоритмов.....	4
1.2. Понятие блок-схемы. Основные виды блоков	8
1.3. Графическая реализация линейного алгоритма	9
1.4. Графическая реализация разветвляющегося алгоритма	12
1.5. Выполнение блок-схем	22
2. Основы программирования на языке C++	32
2.1. Базовые знания о языке программирования C++	32
2.2. Этапы разработки программ.	32
2.3. Типы данных.....	33
2.4. Рекомендации по стилю программирования.....	38
2.5. Трансляторы. Синтаксис и семантика.	39
2.6. Константа.....	40
2.7. Инструкции. Операторы.....	40
3. Ввод, вывод в C++.....	42
3.1. Вывод с помощью cout.....	42
3.2. Перегруженная операция <<	42
3.3. Ввод с помощью cin.....	49
4. Конструкция ветвления в C++	52
4.1. Оператор if.....	52
4.2. Оператор switch.....	57
5. Циклы в C++	59
5.1. Цикл for.....	59
5.2. Цикл while	61
5.3. Цикл do while	62
6. Операции инкремента и декремента	63
Задача «Табуляция». Протабулировать функцию.....	65
7. Массивы в C++.....	65
7.1. Правила инициализации массивов	68
8. Строки	69
8.1. Построчное чтение ввода	71
8.2. Строчно-ориентированный ввод с помощью getline()	72
8.3. Строчно-ориентированный ввод с помощью get()	72

9. Введение в класс string	73
9.1. Присваивание, конкатенация и добавление	76
9.2. Дополнительные сведения об операциях класса string	78
9.3. Дополнительные сведения о вводе-выводе класса string	81
10. Перечисления.....	84
11. Указатели.....	89
11.1. Объявление и инициализация указателей	92
11.2. Опасность, связанная с указателями.....	93
11.3. Указатели и числа.....	94
12. Динамический массив в C++	95
12.1. Выделение памяти с помощью операции new.....	95
12.2. Освобождение памяти с помощью операции delete.....	98
12.3. Использование операции new для создания динамических массивов	99
12.4. Использование динамического массива	102
13. Функции в C++	104
13.1. Определение функции.....	106
13.2. Прототипирование и вызов функции.....	109
13.2.1. Зачем нужны прототипы?	111
13.2.2. Синтаксис прототипа.....	112
13.3. Рекурсия.....	112
13.3.1. Рекурсия с одиночным рекурсивным вызовом	112
13.3.2. Рекурсия с множественными рекурсивными вызовами.....	115
13.4. Указатели на функции.....	117
14. Класс шаблона vector	121
14.1. Что еще можно делать с помощью векторов.....	125
14.2. Дополнительные возможности векторов.....	128
15. Файловый ввод-вывод.....	132
15.1. Простой файловый ввод-вывод.....	133
15.2. Проверка потока и is_open().....	137
15.3. Открытие нескольких файлов	138
16. Параметры командной строки в C++	139
СПИСОК ЛИТЕРАТУРЫ.....	141

Конспект лекций составлен на основе открытых электронных ресурсов:

«Уроки C++ с нуля» <https://code-live.ru/tag/cpp-manual>,

«Введение в языки программирования Си C++»

1. ПОНЯТИЕ АЛГОРИТМА

1.1. Виды алгоритмов.

Существует несколько определений понятия алгоритма. Приведем два самых распространенных.

Алгоритм – последовательность чётко определенных действий, выполнение которых ведёт к решению задачи. Алгоритм, записанный на языке машины, есть программа решения задачи.

Алгоритм – это совокупность действий, приводящих к достижению результата за конечное число шагов.

Вообще говоря, первое определение не передает полноты смысла понятия алгоритм. Используемое слово «последовательность» сужает данное понятие, т.к. действия не обязательно должны следовать друг за другом – они могут повторяться или содержать условие.

Свойства алгоритмов:

1. Дискретность (от лат. discretus — разделенный, прерывистый) – это разбиение алгоритма на ряд отдельных законченных действий (шагов).

2. Детерминированность (от лат. determinate — определенность, точность) - любое действие алгоритма должно быть строго и недвусмысленно определено в каждом случае. Например, алгоритм проезда к другу, если к остановке подходят автобусы разных маршрутов, то в алгоритме должен быть указан конкретный номер маршрута 5. Кроме того, необходимо указать точное количество остановок, которое надо проехать, скажем, три.

3. Конечность – каждое действие в отдельности и алгоритм в целом должны иметь возможность завершения.

4. Массовость – один и тот же алгоритм можно использовать с разными исходными данными.

5. Результативность – алгоритм должен приводить к достоверному решению.

Основная цель алгоритмизации – составление алгоритмов для ЭВМ с дальнейшим решением задачи на ЭВМ.

Примеры алгоритмов:

1. Любой прибор, купленный в магазине, снабжается инструкцией по его использованию. Данная инструкция и является алгоритмом для правильной эксплуатации прибора.

2. Каждый шофер должен знать правила дорожного движения. Правила дорожного движения однозначно регламентируют поведение каждого участника движения. Зная эти правила, шофер должен действовать по определенному алгоритму.

3. Массовый выпуск автомобилей стал возможен только тогда, когда был придуман порядок сборки машины на конвейере. Определенный порядок сборки автомобилей – это набор действий, в результате которых получается автомобиль.

Существует несколько способов записи алгоритмов. На практике наиболее распространены следующие формы представления алгоритмов:

1. Словесная (запись на естественном языке);
2. Псевдокоды (полуформализованные описания алгоритмов на условном алгоритмическом языке, включающие в себя как элементы языка программирования, так и фразы естественного языка, общепринятые математические обозначения и др.);
3. Графическая (изображения из графических символов – блок-схема);
4. Программная (тексты на языках программирования – код программы).

Рассмотрим подробно каждый вариант записи алгоритмов на примере следующей задачи. Требуется найти частное двух чисел.

Словесный способ записи алгоритмов представляет собой описание последовательных этапов обработки данных. Алгоритм задается в произвольном изложении на естественном языке. Ответ при этом получает человек, который выполняет команды согласно словесной записи.

Пример словесной записи:

- 1) Задать два числа, являющиеся делимым и делителем;
- 2) Проверить, равняется ли делитель нулю;
- 3) Если делитель не равен нулю, то найти частное, записать его в ответ;
- 4) Если делитель равен нулю, то в ответ записать "нет решения".

Словесный способ не имеет широкого распространения, так как такие описания: строго не формализуемы; страдают многословностью записей; допускают неоднозначность толкования отдельных предписаний.

Графическая реализация алгоритма представляет собой блок-схему. Блок-схема состоит из блоков определенной формы, соединенных стрелками. Ответ при этом получает человек, который выполняет команды согласно блок-схеме. Более подробно о блок-схемах будет рассказано в Лекции 2.

Программная реализация алгоритма – это компьютерная программа, написанная на каком-либо алгоритмическом языке программирования, например, C++, Pascal, Basic и т.д. Программа состоит из команд определенного языка программирования. Отметим, что одна и та же блок-схема может быть реализована на разных языках программирования. Ответ при этом получает ЭВМ, а не человек. Более подробно о составлении программ на языке программирования C++ смотреть Лекцию 3.

Различают три основных вида алгоритмов:

- 1) линейный алгоритм,
- 2) разветвляющийся алгоритм,
- 3) циклический алгоритм.

Линейный алгоритм – это алгоритм, в котором действия выполняются однократно и строго последовательно.

Самый простой пример реализации линейного алгоритма – путь из университета домой.

Словесный способ записи данного алгоритма:

- 1) выйти из университета на остановку;
- 2) подождать нужный автобус;
- 3) сесть на нужный автобус;
- 4) оплатить проезд;
- 5) выйти на требуемой остановке;
- 6) дойти до дома.

Очевидно, что данный пример относится к линейному алгоритму, т.к. все действия следуют одно за другим, без условий и повторений.

Разветвляющийся алгоритм – это алгоритм, в котором в зависимости от условия выполняется либо одна, либо другая последовательность действий.

Самый простой пример реализации разветвляющегося алгоритма – если на улице идет дождь, то необходимо взять зонт, иначе не брать зонт с собой.

Приведенный выше пример псевдокода по нахождению частного двух чисел также относится к разветвляющемуся алгоритму.

Циклический алгоритм – это алгоритм, команды которого повторяются некоторое количество раз подряд.

Самый простой пример реализации циклического алгоритма – при чтении книги будут повторяться одни и те же действия: прочитать страницу, перелистнуть и т.д.

Более подробно о линейном, разветвляющемся и циклическом алгоритмах смотреть Лекцию 2.

Краткие итоги:

Любая задача может быть разбита на элементарные действия. Для любой математической задачи или ситуации из жизни можно составить алгоритм решения. Алгоритм может быть описан словесно, псевдокодом, графически или программно. Задача всегда решается с помощью базовых типов алгоритма – линейного, разветвляющегося или циклического.

Вопросы

1. Что такое алгоритм?
2. В чем состоит задача алгоритмизации?
3. Какими свойствами обладает алгоритм?
4. Какие виды алгоритма бывают?

Упражнения

1. Составьте алгоритмы по походу в магазин за яблоками. Используйте линейный и разветвляющийся алгоритмы. Реализуйте их словесно.

2. Составьте алгоритм по нахождению корней квадратного уравнения через дискриминант. Используйте разветвляющийся алгоритм. Реализуйте его псевдокодом.






1.2. Понятие блок-схемы. Основные виды блоков

Блок-схема – это графическая реализация алгоритма.

Блок-схема представляет собой удобный и наглядный способ записи алгоритма.

Блок-схема состоит из функциональных блоков разной формы, связанных между собой стрелками. В каждом блоке описывается одно или несколько действий. Основные виды блоков представлены в табл. 2.1.

Таблица 2.1. Виды блоков

Форма блока	Назначение блока
	начало и конец блок-схемы
	блок ввода данных
	блок выполнения действия
	блок условия
	блок вывода данных

Любая команда алгоритма записывается в блок-схеме в виде графического элемента – блока, и дополняется словесным описанием. Блоки в блок-схемах соединяются линиями потока информации. Направление потока информации указывается стрелкой. В случае потока информации сверху вниз и слева направо стрелку ставить не обязательно. Блоки в блок-схеме имеют только один вход и один выход (за исключением логического блока – блока с условием).

Блок начала блок-схемы имеет один выход и не имеет входов, блок конца блок-схемы имеет один вход и не имеет выходов. Блок условия – единственный блок, имеющий два выхода, т.к. соответствует

разветвляющемуся алгоритму. На одном выходе указывается "да", на другом – "нет". Все остальные блоки имеют один вход и один выход. Блок выполнения действия может содержать присвоение значения переменной (например, " $x = 5$ ") или вычисление (например " $y = x - 4$ ").

Математические выражения и логические высказывания должны быть описаны математическим языком, т.к. блок-схема не должна иметь привязки к какому-то определенному языку программирования. Одна и та же блок-схема может быть реализована в программах на разных языках программирования. К примеру, функция в блок-схеме будет выглядеть таким образом: $y = x^2$, а не таким образом: $y = x^{\wedge}2$.

Все три вида алгоритмов реализуются в блок-схеме названными выше типами блоков. К примеру, в линейном алгоритме могут присутствовать все блоки, кроме блока условия. В разветвляющемся и циклическом алгоритмах могут быть использованы все названные виды блоков, но обязательным является блок условия. Внутри блока условия записывается условие, про которое можно однозначно ответить, истинно оно или ложно. Если условие истинно, то выполняются действия, соответствующие стрелке "да", иначе стрелке "нет".

1.3. Графическая реализация линейного алгоритма

Приведем простейшие примеры, соответствующие линейному алгоритму.

Пример 1. Вася хочет позвонить Пете по городскому телефону. Необходимо составить блок-схему, описывающую порядок действий Васи.

Решение. Чтобы позвонить по городскому телефону, нужно знать номер Пети. Значит, сначала надо найти номер телефона Пети, набрать его и поговорить с Петей. На этом цель Васи (поговорить с Петей по телефону) будет достигнута. Результат блок-схемы представлен на рис. 2.1.



Рисунок 2.1. Блок-схема для примера 1

Пример 2. Ученику требуется купить учебник. Составить блок-схему, описывающую порядок действий ученика.

Решение. Сначала ученику нужно взять деньги, потом прийти в книжный магазин и заплатить за учебник. На этом цель (покупка учебника) будет достигнута. Результат блок схемы представлен на рис. 2.2.



Рисунок 2.2. Блок-схема для примера 2

Пример 3. Даны числа $a = 2$, $b = 7$. Вычислить сумму S и разность R чисел a и b .

Решение. Сначала следует задать значения для чисел a и b , согласно условиям задачи. После этого их уже можно будет использовать в расчетах для

получения суммы и разности по формулам: $S = a + b$, $R = a - b$. Полученные значения суммы и разности нужно будет показать на экране, и мы используем блок вывода данных. Если не выводить данные на экран, то пользователь нашего алгоритма не узнает, какие получились значения суммы и разности. Результат блок схемы представлен на рисунке 2.3.

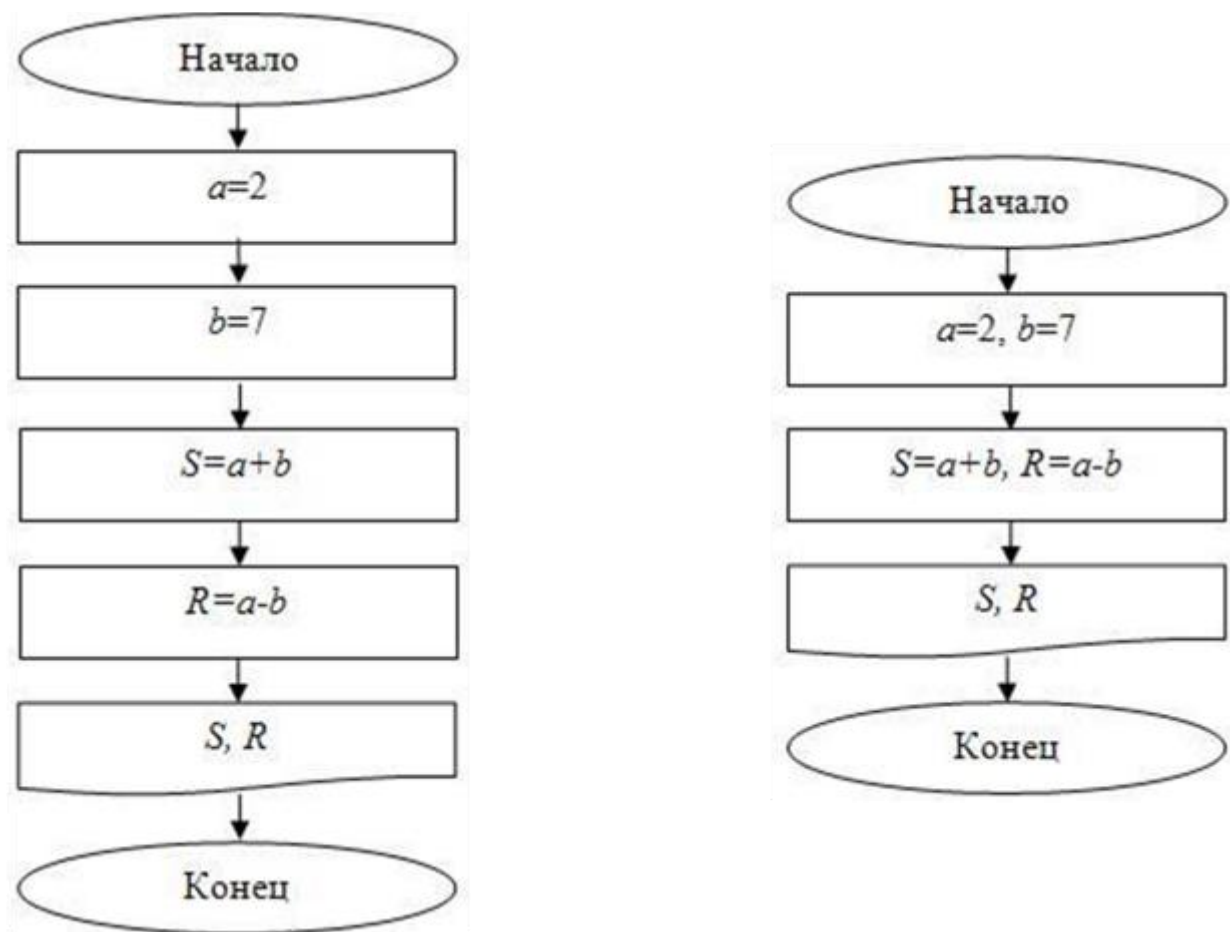


Рисунок 2.3. Блок-схема для примера 3: а) в каждом блоке по одному действию, б) действия объединены по смыслу операции

В блок-схеме на рис. 2.3а каждое действие расположено в отдельном блоке. В блок-схеме рис. 2.3б объединены между собой схожие по смыслу операции. В дальнейшем мы будем объединять некоторые действия в один блок. Это очень удобно и визуально упрощает чтение блок-схемы.

1.4. Графическая реализация разветвляющегося алгоритма

В разветвляющемся алгоритме обязательным блоком является блок условия, который представлен на рис. 2.4.

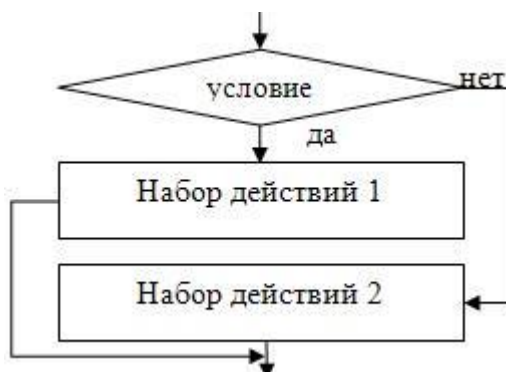


Рисунок 2.4. Использование блока условия в общем виде

Внутри блока условия записывается условие. Если данное условие верно, то выполняются блоки, идущие по стрелке "да", т.е. "Набор действий 1". Если условие оказывается неверным, т.е. ложным, то выполняются блоки, идущие по стрелке "нет", а именно "Набор действий 2". Разветвление заканчивается, когда обе стрелки ("да" и "нет") соединяются. На рис. 2.5 представлен еще один вариант использования блока условия. Бывают задачи, в которых, исходя из условия, необходимо либо выполнить действие, либо пропустить его. Если условие верно выполняется, то следуют блоки, соответствующие стрелке "да", т.е. "Набор действий 1". Если же условие оказывается ложным, то следует перейти по стрелке "нет". Т.к. стрелке "нет" не соответствует ни одного блока с действием, то ни одного действия не будет выполнено. Т.е. получается, что мы пропустили и не выполнили "Набор действий 1".



Рис. 2.5. Вариант использования блока условия

В разветвляющемся алгоритме возможна запись сразу нескольких условий, которые могут объединяться союзом "ИЛИ" или пересекаться союзом "И". Рассмотрим случай двух условий: "условие 1" и "условие 2".

Если необходимо, чтобы оба условия были верными одновременно, то следует использовать логическое пересечение "И":

"условие 1 и условие 2".

Если достаточно, чтобы только одно условие выполнялось – либо первое, либо второе, то следует использовать логическое объединение "ИЛИ":

"условие 1 ИЛИ условие 2".

Приведем простейшие примеры, соответствующие разветвляющемуся алгоритму.

Пример 4. Джон звонит Полу по городскому телефону, но трубку может взять не только Пол. Составить блок-схему, описывающую действия Джона в этом случае.

Решение. В отличие от примера 1, здесь присутствует условие – Пол ли взял трубку телефона. На данное условие можно однозначно ответить: "да", Пол, или "нет", кто-то другой. Если трубку взял Пол, то Джону нужно с ним поговорить, и цель будет достигнута. Если трубку взял кто-то другой, то необходимо позвать Пола к телефону, поговорить с ним, и цель также будет достигнута. Третьего варианта, например, "не туда попали" или "его нет дома" мы не рассматриваем. Результат блок-схемы представлен на рис. 2.6.

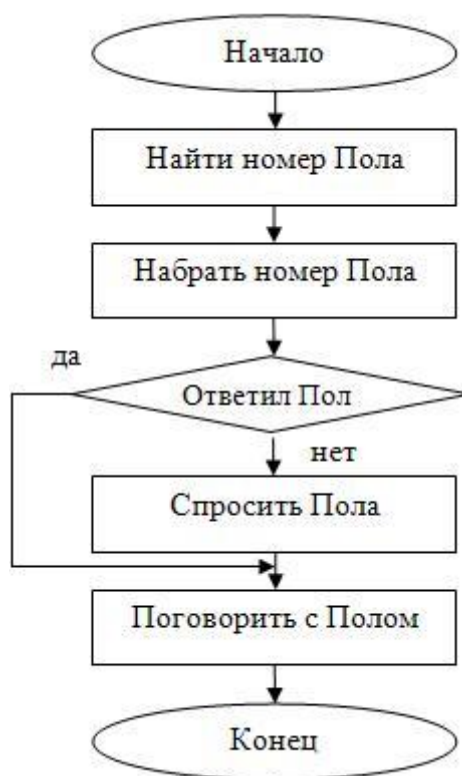


Рисунок 2.6. Блок-схема для примера 4

Пример 5. Ученику требуется купить учебник. В магазине в наличие оказался нужный учебник в жесткой и мягкой обложке. Составить блок-схему, описывающую действия ученика.

Решение. В данном примере присутствует условие: "Нужна жесткая обложка".

Ученик может согласиться с этим высказыванием, тогда он выполнит действие, соответствующее стрелке "да" и купит учебник в жесткой обложке.

Если ученик не соглашается с данным условием, то будет выполняться действие, соответствующее стрелке "нет", и в этом случае ученик купит учебник в мягкой обложке.

И в том, и в другом случае, цель будет достигнута и задача будет выполнена, т.к. ученик купит учебник.

Результат блок схемы представлен на рисунке 2.7.

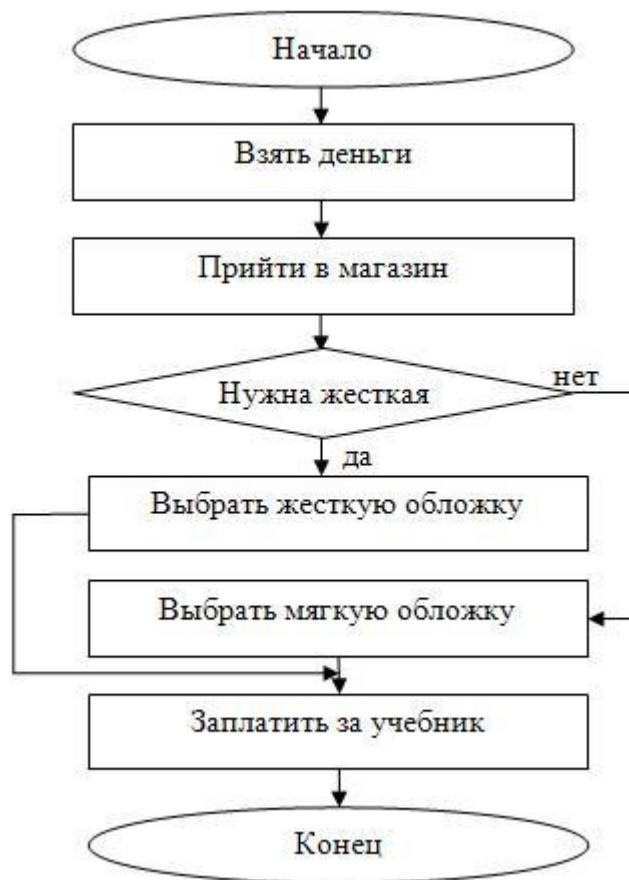


Рисунок 2.7. Блок-схема для примера 5

Пример 6. Даны числа $a = 2$, $b = 7$. Вычислить сумму S и разность R чисел a и b . Сравнить полученные значения S и R и указать большее из них.

Решение. Как и в примере 3, сначала необходимо задать значения a и b . Затем рассчитать сумму и разность по формулам: $S = a + b$, $R = a - b$ и вывести полученные числа на экран (блок вывода данных). Когда значения S и R будут получены, следует сравнить их между собой. Условие запишется в виде: $S > R$. Если полученная сумма S будет больше разности R , то мы пойдем по стрелке "да" и выведем фразу "max S". Если же условие окажется ложным (т.е. $S < R$), то пойдем по стрелке "нет" и выведем фразу "max R". Результат блок схемы представлен на рис. 2.8.

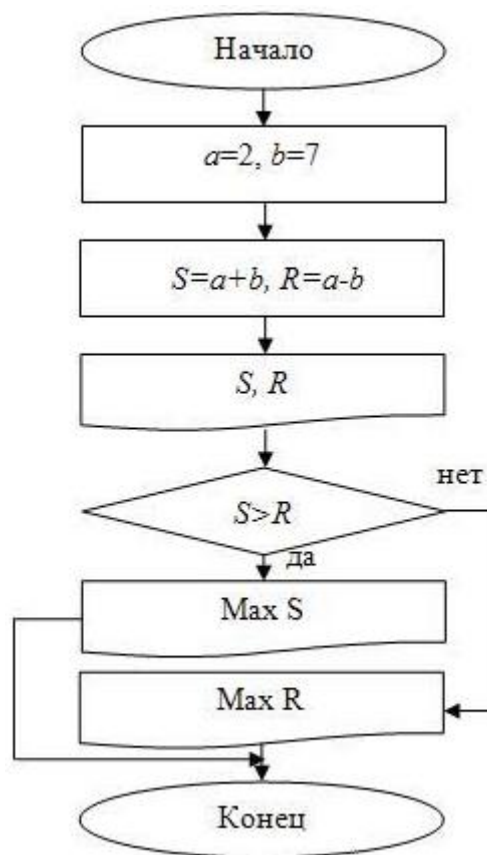


Рисунок 2.8. Блок-схема для примера 6

Графическая реализация циклического алгоритма

В рассмотрении циклического алгоритма следует выделить несколько понятий.

Тело цикла – это набор инструкций, предназначенный для многократного выполнения.

Итерация – это единичное выполнение тела цикла.

Переменная цикла – это величина, изменяющаяся на каждой итерации цикла.

Каждый цикл должен содержать следующие необходимые элементы:

1. Первоначальное задание переменной цикла,
2. Проверку условия,
3. Выполнение тела цикла,
4. Изменение переменной цикла.

Циклы бывают двух видов – с предусловием и с постусловием. В **цикле с предусловием** сначала проверяется условие входа в цикл, а затем выполняется тело цикла, если условие верно. Цикл с предусловием представлен на рис. 2.9. Цикл с предусловием также может быть задан с помощью счетчика. Это удобно в тех случаях, когда точно известно количество итераций. В общем виде блок-схема, реализующая цикл с предусловием, представлена ниже. Сначала задается начальное значение переменной цикла, затем условие входа в цикл, тело цикла и изменение переменной цикла. Выход из цикла осуществляется в момент проверки условия входа в цикл, когда оно не выполняется, т.е. условие ложно. Цикл с предусловием может ни разу не выполниться, если при первой проверке условия входа в цикл оно оказывается ложным.

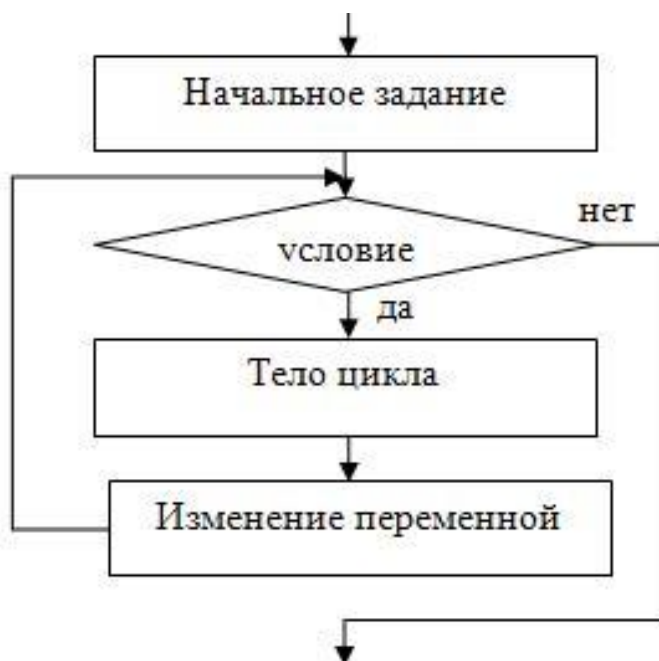


Рисунок 2.9. Циклический алгоритм с предусловием в общем виде

В цикле с постусловием сначала выполняется тело цикла, а потом проверяется условие. Циклический алгоритм с постусловием представлен на рисунке 2.10

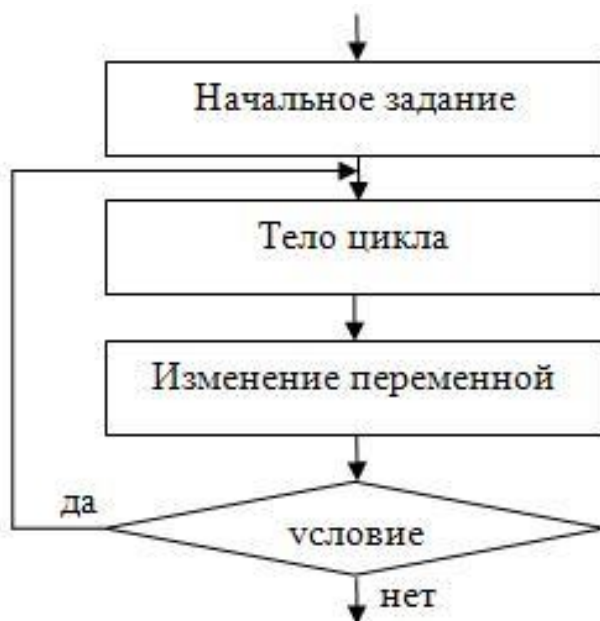


Рис. 2.10. Циклический алгоритм с постусловием в общем виде

Если условие верно, то итерация повторяется, если же неверно, то осуществляется выход из цикла. В отличие от цикла с предусловием, любой цикл с постусловием всегда выполнится хоть раз.

Примечание. Как видно из представленных блок-схем для циклов с предусловием и постусловием, условие записывается внутри блока условия (формы ромба), как и в разветвляющемся алгоритме. Принципиальная разница между разветвляющимся и циклическим алгоритмами при графической реализации состоит в том, что в циклическом алгоритме в обязательном порядке присутствует стрелка, идущая наверх. Именно эта стрелка обеспечивает многократный повтор тела цикла.

Приведем простейшие примеры, соответствующие циклическому алгоритму.

Пример 7. Вася звонит Пете, но у Пети может быть занята линия. Составить блок-схему действий Васи в этом случае.

Решение. Когда телефонная линия занята, то необходимо снова и снова набирать номер, пока Петя не закончит предыдущий разговор, и телефонная линия не окажется вновь свободной. Блок-схема представлена на рис. 2.11.

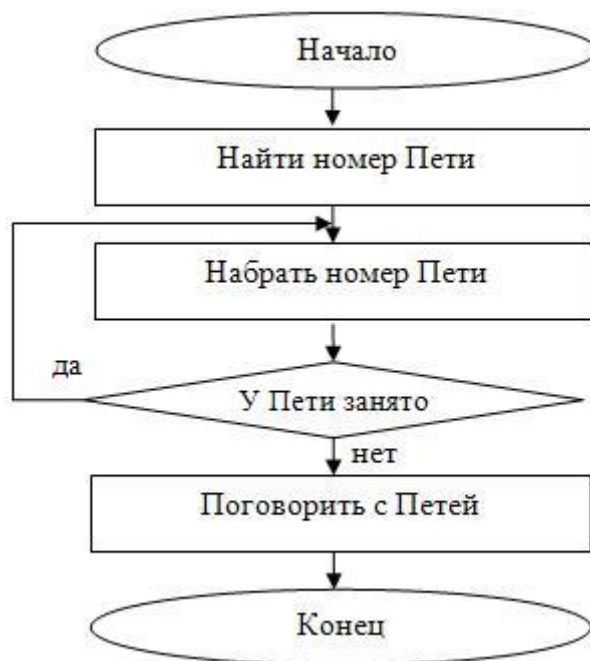


Рисунок 2.11. Блок-схема для примера 7

Здесь тело цикла состоит из одного действия "Набрать номер Пети", т.к. именно это действие следует повторять, пока линия будет занята. Под итерацией цикла понимается очередная попытка дозвониться до Пети. Как таковой переменной цикла здесь нет, т.к. ситуация взята из жизни. Выход из цикла происходит в тот момент, когда условие "У Пети занято" стало неверным, т.е. телефонная линия свободна – действительно, нет необходимости больше набирать номер Пети. В данном примере применен цикл с постусловием, т.к. сначала необходимо набрать номер Пети, ведь иначе мы не можем ответить на вопрос – занята ли линия у Пети.

Пример 8. Ученику требуется купить учебник. Составить блок-схему, описывающую действия ученика в случае, если учебника нет в ряде магазинов.

Решение. Действия ученика в данном примере очевидны: когда он приходит в первый и любой последующий магазины, то возможны два варианта – учебник имеется в наличии или учебника нет в продаже. Если учебника нет в продаже, то ученику следует пойти в другой книжный магазин и спросить данный учебник, и т.д. пока учебник не будет куплен, т.к. перед учеником стоит конечная цель – купить учебник. Мы будем использовать цикл с предусловием, т.к. сначала требуется найти магазин, имеющий в наличии данный учебник. Цикл будет выполняться, пока условие "В данном магазине нет учебника" будет верным, а выход из цикла осуществится, когда условие станет ложным, т.е. когда ученик придет в магазин, в котором есть данный учебник. Действительно, в этом случае ученик купит нужный ему учебник и не будет

больше искать книжные магазины. Результат блок-схемы представлен на рисунке 2.12.



Рисунок 2.12. Блок-схема для примера 8

Здесь тело цикла состоит из одного действия "Найти другой книжный магазин". Переменной цикла в явном виде нет, но можно подразумевать номер магазина, в который пришел ученик в очередной раз. Как любой другой цикл с предусловием, данный цикл может ни разу не выполниться (не иметь итераций), если в первом же магазине окажется нужный учебник.

Примечание. Если в данную задачу добавить условие выбора учебника в жесткой или мягкой обложке, как в примере 5, то оно появится после выхода из цикла. На реализацию циклического алгоритма данное условие не повлияет.

Лекция 3

Пример 9. Даны числа a , b . Известно, что число a меняется от -10 до 10 с шагом 5, $b = 7$ и не изменяется. Вычислить сумму S и разность R чисел a и b для всех значений a и b .

Решение. В отличие от примеров 3 и 6 здесь число a меняется от -10 до 10 с шагом 5. Это означает, что число a является переменной цикла. Сначала a равно -10 – это первоначальное задание переменной цикла. Далее a будет изменяться с шагом 5, и т.д. пока не будет достигнуто значение 10 – это соответствует изменению переменной цикла. Итерации надо повторять, пока выполняется условие " $a \leq b$ ". Итак, a будет принимать следующие значения: -10, -5, 0, 5, 10. Число b не будет являться переменной цикла, т.к. b не изменяется по условию задачи. Результат блок-схемы (с предусловием) представлен на рисунке 2.13.

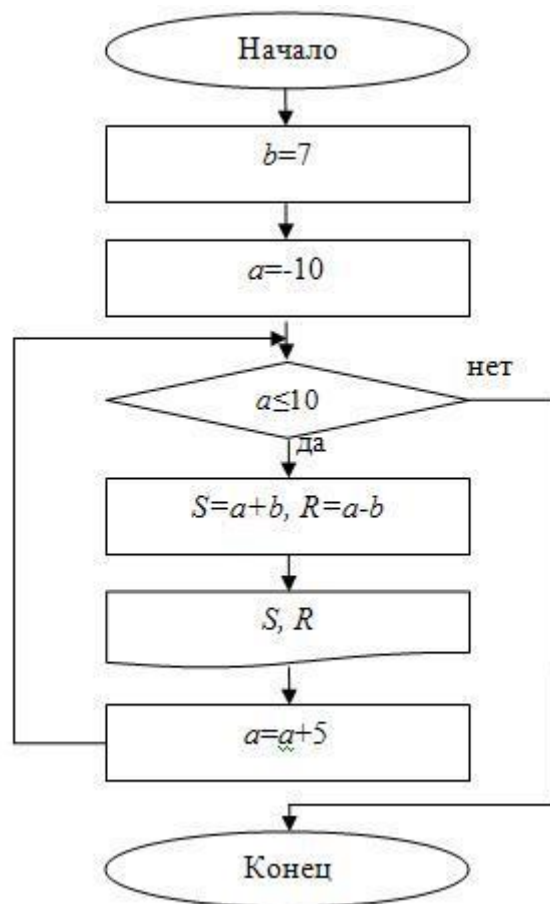


Рисунок 2.13. Блок-схема для примера 9 (с предусловием)

Тело цикла состоит из нескольких действий: вычисление суммы, вычисление разности и вывод полученных данных на экран. Таким образом, у нас получится несколько значений сумм и разностей, т.к. a изменяется. Количество сумм и количество разностей совпадет с количеством различных значений a , т.е. пять.

Данная задача может быть сделана и с циклом с предусловием, и с постусловием. В этом случае тело цикла, условие и изменение переменной цикла будут такими же, как и в цикле с предусловием, но сначала необходимо

выполнить тело цикла, а потом проверить условие для выполнения следующей итерации.

Приведем блок-схему, использующую цикл с постусловием, на рисунке 2.14.

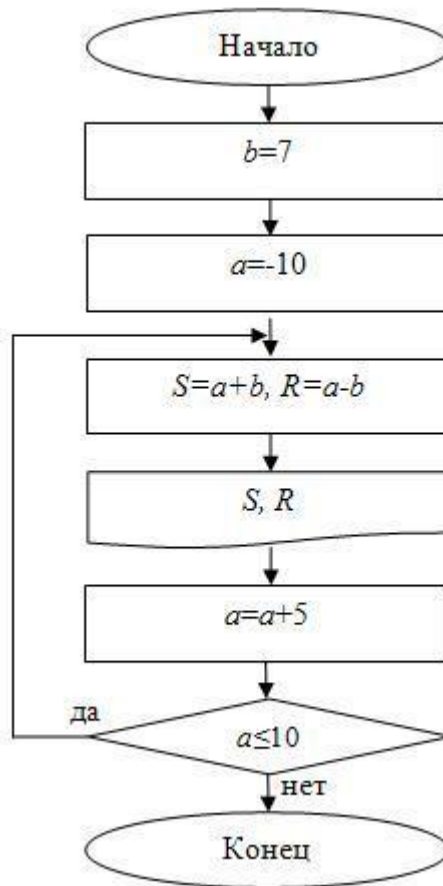


Рисунок 2.14. Блок-схема для примера 9 (с постусловием)

В данной задаче также могут быть соединены циклический и разветвляющийся алгоритмы, если по условию задачи требуется сравнить полученные значения суммы и разности, как в примере 6. В этом случае цикл можно реализовать как с предусловием, так и с постусловием, а сравнение суммы и разности добавится внутрь тела цикла, т.к. следует сравнить между собой все полученные суммы и разности. Организация самого цикла останется прежней. Приведем на рисунке 2.15а блок-схему с предусловием, а на рисунке 2.15б блок-схему с постусловием.

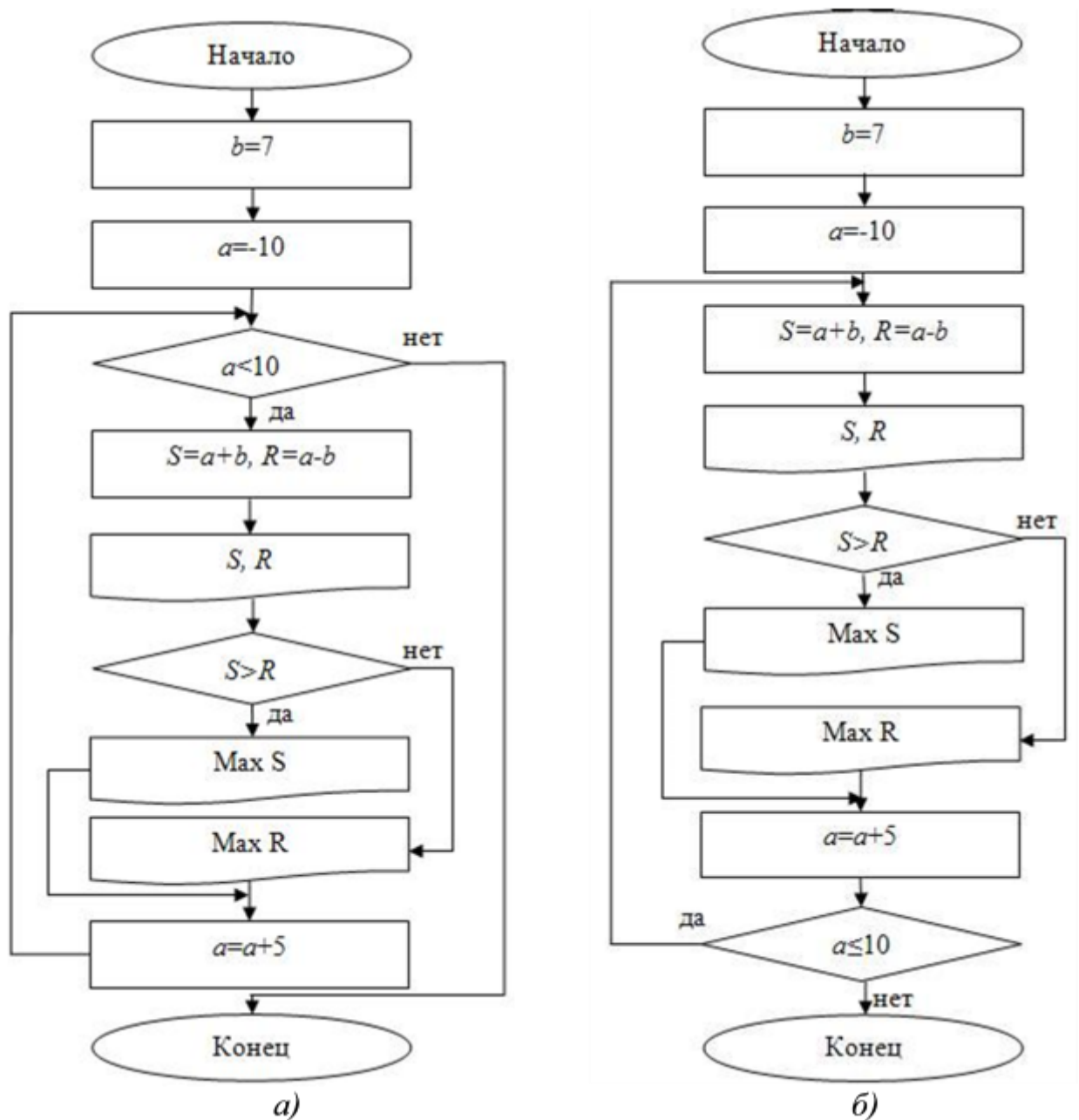


Рисунок 2.15. Блок-схема с ветвлением для примера 9: а) с предусловием, б) с постусловием

1.5. Выполнение блок-схем

Блок-схема сама по себе не содержит ответа. Чтобы получить результат, нужно выполнить блок-схему.

Выполнение блок-схемы – это прохождение всех действий блок-схемы согласно алгоритму от блока Начало до блока Конец для получения результата.

Если блок-схема составлена корректно, то, выполнив ее, человек получит ответ к своей задаче. Если же при составлении блок-схемы были

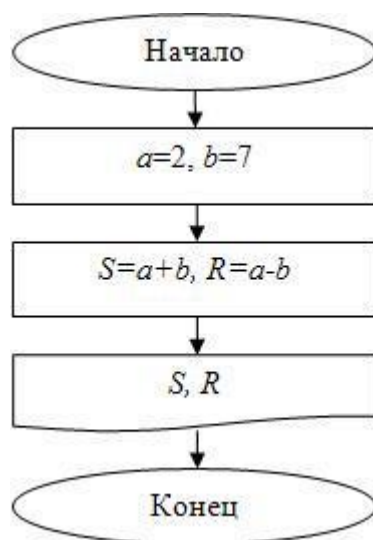
допущены ошибки, то исполнитель либо вообще не дойдет до блока Конец, либо получит неверный ответ.

Для выполнения нам понадобится поле для расчетов (аналог "оперативной памяти") и поле для вывода результата (аналог экрана для вывода данных). Экран будем показывать несколько раз в зависимости от вывода новых данных на экране.

Выполнение блок-схемы для примера 3. Даны числа $a = 2$, $b = 7$. Вычислить сумму S и разность R чисел a и b .

Выполнение блок-схемы приведем в таблице 2.3.

Таблица 2.2. Выполнение блок-схемы для примера 3



Расчеты:

Начало
 $a=2, b=7$

$$S = a + b = \{\text{подставляем значения } a \text{ и } b\} = 2 + 7 = 9$$

$$R = a - b = \{\text{подставляем значения } a \text{ и } b\} = 2 - 7 = -5$$

Выводим на экран $S=9, R=-5$:

Вывод данных (экран)

$S = 9$

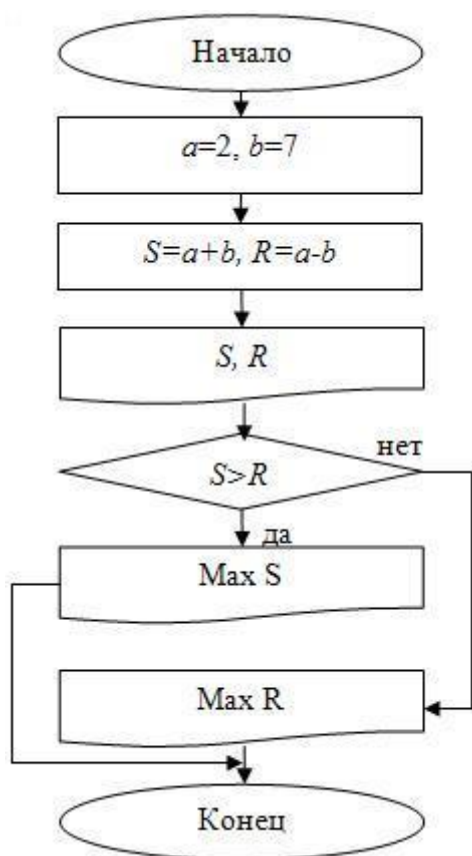
$R = -5$

Конец.

Выполнение блок-схемы для примера 6. Даны числа $a = 2$, $b = 7$. Вычислить сумму S и разность R чисел a и b . Сравнить полученные значения S и R и указать большее из них.

Выполнение блок-схемы приведем в табл. 2.3.

Таблица 2.3. Выполнение блок-схемы для примера 6



Расчеты:

Начало
 $a=2, b=7$

$S = a + b = 2 + 7 = 9$
 $R = a - b = 2 - 7 = -5$
 Выводим на экран $S=9, R=-5$:

$S > R \quad 9 > -5$ да, верно
 Выводим на экран "Max S":

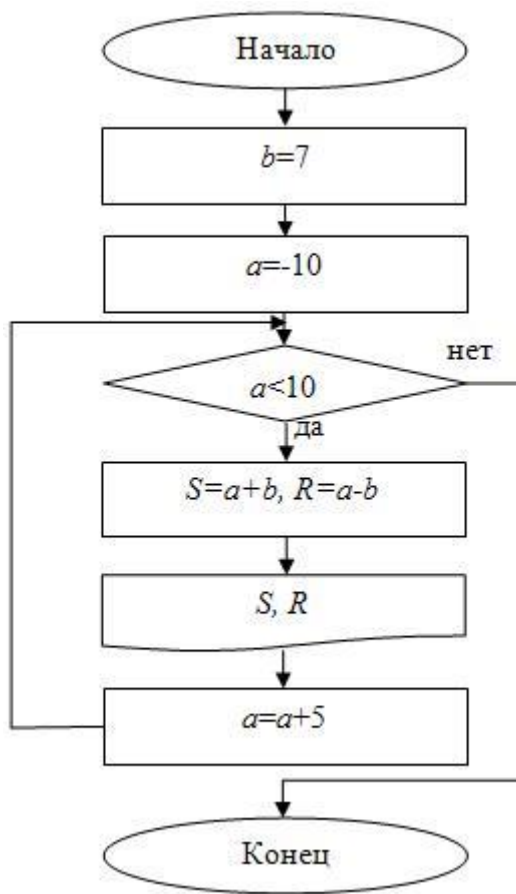
Вывод данных (экран
 $S=9$
 $R=-5$
 Max S

Конец

Выполнение блок-схемы для примера 9. Даны числа a, b . Известно, что число a меняется от -10 до 10 с шагом 5, $b = 7$ и не изменяется. Вычислить сумму S и разность R чисел a и b для всех значений a и b .

Выполнение блок-схемы с предусловием приведем в таблице 2.4.

Таблица 2.4. Выполнение блок-схемы с предусловием для примера 9



Расчеты:

Начало

$b=7$

$a=-10$

$a \leq 10$ $-10 \leq 10$ да, верно

$S=a+b=-10+7=-3$

$R=a-b=-10-7=-17$

Вывод S, R

Экран

$S=-3$ $R=-17$

$a=a+5=-10+5=-5$

{Идем по стрелке вверх}

$a \leq 10$ $-5 \leq 10$ да, верно

$S=a+b=-5+7=2$

$R=a-b=-5-7=-12$

Вывод S, R

Экран

$S=-3$ $R=-17$

$S=2$ $R=-12$

$a=a+5=-5+5=0$

{Идем по стрелке вверх}

$a \leq 10$ $0 \leq 10$ да, верно

$S=a+b=0+7=7$

$R=a-b=0-7=-7$

Вывод S, R

Экран

$S=-3$ $R=-17$

$S=2$ $R=-12$

$S=7$ $R=-7$

$a=a+5=0+5=5$

{Идем по стрелке вверх}

$a \leq 10$ $5 \leq 10$ да, верно

$S=a+b=5+7=12$

$R=a-b=5-7=-2$

Вывод S, R

Экран

$S=-3$ $R=-17$
 $S=2$ $R=-12$
 $S=7$ $R=-7$
 $S=12$ $R=-2$

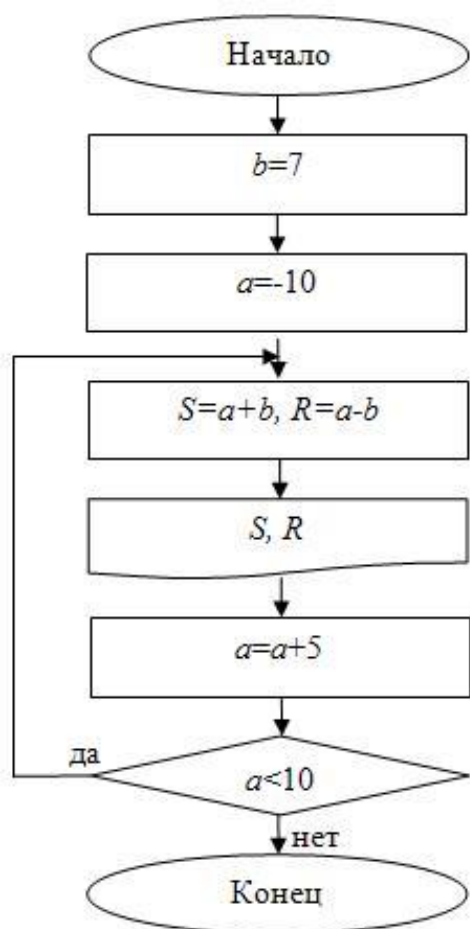
$a=a+5=5+5=10$
{Идем по стрелке вверх}
 $a \leq 10$ $10 \leq 10$ да, верно
 $S=a+b=10+7=17$
 $R=a-b=10-7=3$
Вывод S, R

Экран
 $S=-3$ $R=-17$
 $S=2$ $R=-12$
 $S=7$ $R=-7$
 $S=12$ $R=-2$
 $S=17$ $R=3$

$a=a+5=10+5=15$
{Идем по стрелке вверх}
 $a \leq 10$ $15 \leq 10$ нет, ложно
{выходим из цикла}
Конец.

Выполнение блок-схемы с постусловием приведем в таблице 2.5.

Таблица 2.5. Выполнение блок-схемы с постусловием для примера 9



Расчеты:

Начало

 $b=7$ $a=-10$ $S=a+b=-10+7=-3$ $R=a-b=-10-7=-17$

Вывод S, R

Экран

 $S=-3 \quad R=-17$ $a=a+5=-10+5=-5$ $a \leq 10 \quad -5 \leq 10$ да, верно
{Идем по стрелке вверх} $S=a+b=-5+7=2$ $R=a-b=-5-7=-12$

Вывод S, R

Экран

 $S=-3 \quad R=-17$ $S=2 \quad R=-12$ $a=a+5=-5+5=0$ $a \leq 10 \quad 0 \leq 10$ да, верно
{Идем по стрелке вверх} $S=a+b=0+7=7$ $R=a-b=0-7=-7$

Вывод S, R

Экран

 $S=-3 \quad R=-17$ $S=2 \quad R=-12$ $S=7 \quad R=-7$ $a=a+5=0+5=5$ $a \leq 10 \quad 5 \leq 10$ да, верно
{Идем по стрелке вверх} $S=a+b=5+7=12$ $R=a-b=5-7=-2$

Вывод S, R

Экран

S=-3 R=-17

S=2 R=-12

S=7 R=-7

S=12 R=-2

a=a+5=5+5=10

a≤10 10≤10 да, верно

{Идем по стрелке вверх}

S=a+b=10+7=17

R=a-b=10-7=3

Вывод S, R

Экран

S=-3 R=-17

S=2 R=-12

S=7 R=-7

S=12 R=-2

S=17 R=3

a=a+5=10+5=15

a≤10 15≤10 нет, ложно

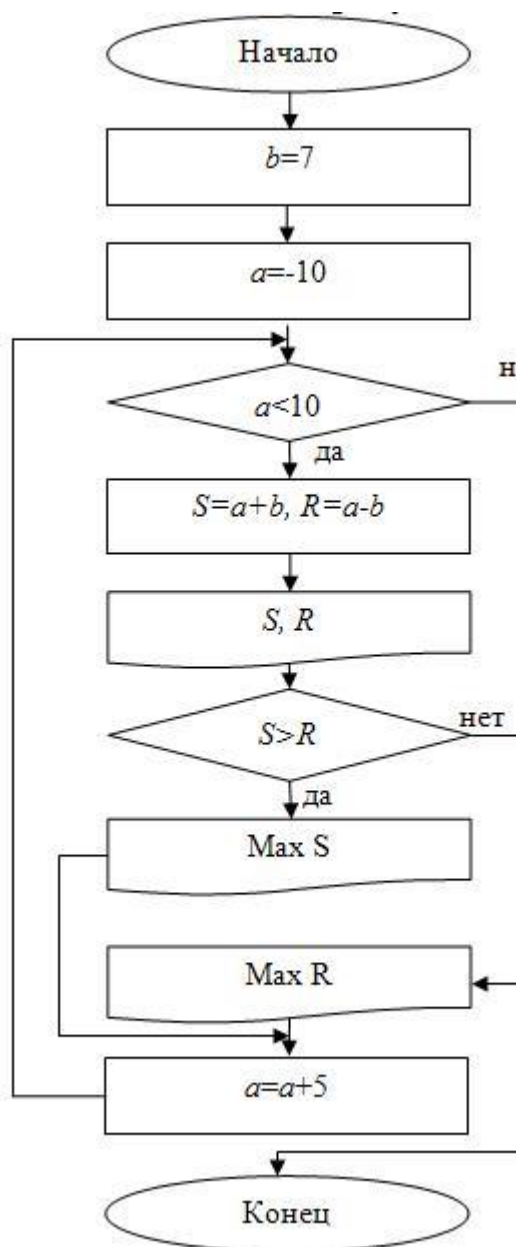
{выходим из цикла}

Конец.

Пример 10. Даны числа a , b . Известно, что число a меняется от -10 до 10 с шагом 5, $b = 7$ и не изменяется. Вычислить сумму S и разность R и сравнить полученные значения суммы и разности для всех значений a и b .

Выполнение блок-схемы с предусловием приведем в таблице 2.6.

Таблица 2.6. Выполнение блок-схемы с предусловием для примера 10



Расчеты:

Начало

$b=7$

$a=-10$

$a \leq 10$ $-10 \leq 10$ да, верно

$S=a+b=-10+7=-3$

$R=a-b=-10-7=-17$

Вывод S, R

$S > R$ $-3 > -17$ да, верно

Вывод "Max S"

Вывод данных (экран)

$S=-3$ $R=-17$ MaxS

$a=a+5=-10+5=-5$

{Идем по стрелке вверх}

$a \leq 10$ $-5 \leq 10$ да, верно

$S=a+b=-5+7=2$

$R=a-b=-5-7=-12$

Вывод S, R

$S > R$ $2 > -12$ да, верно

Вывод "Max S"

Вывод данных (экран)

$S=-3$ $R=-17$ MaxS

$S=2$ $R=-12$ MaxS

$a=a+5=-5+5=0$

{Идем по стрелке вверх}

$a \leq 10$ $0 \leq 10$ да, верно

$S=a+b=0+7=7$

$R=a-b=0-7=-7$

Вывод S, R

$S > R$ $7 > -7$ да, верно

Вывод "Max S"

Вывод данных (экран)

$S=-3$ $R=-17$ MaxS

$S=2$ $R=-12$ Max S

$S=7$ $R=-7$ Max S

$a=a+5=0+5=5$

$a \leq 10$ $5 \leq 10$ да, верно

{Идем по стрелке вверх}
 $S=a+b=5+7=12$
 $R=a-b=5-7=-2$
Вывод S, R
 $S>R \quad 12>-2$ да, верно
Вывод "Max S"

Вывод данных (экран)
 $S=-3 \quad R=-17 \quad \text{MaxS}$
 $S=2 \quad R=-12 \quad \text{Max S}$
 $S=7 \quad R=-7 \quad \text{Max S}$
 $S=12 \quad R=-2 \quad \text{MaxS}$

$a=a+5=5+5=10$
{Идем по стрелке вверх}
 $a \leq 10 \quad 10 \leq 10$ да, верно
 $S=a+b=10+7=17$
 $R=a-b=10-7=3$
Вывод S, R
 $S>R \quad 17>3$ да, верно
Вывод "Max S"

Вывод данных (экран)
 $S=-3 \quad R=-17 \quad \text{MaxS}$
 $S=2 \quad R=-12 \quad \text{Max S}$
 $S=7 \quad R=-7 \quad \text{Max S}$
 $S=12 \quad R=-2 \quad \text{Max S}$
 $S=17 \quad R=3 \quad \text{MaxS}$

$a=a+5=10+5=15$
{Идем по стрелке вверх}
 $a \leq 10 \quad 15 \leq 10$ нет, ложно
{выходим из цикла}
Конец.

В процессе составления блок-схемы важно "ходить" по стрелкам из блока в блок, следить, чтобы не получалось "тупиковых ситуаций". Такая ситуация возникает, если составитель блок-схемы не нарисовал стрелку из блока. Также частой ошибкой является замыкание стрелки не в тот блок, например, при реализации циклического алгоритма.

Краткие итоги

Любой алгоритм может быть реализован с помощью блок-схемы. Для каждого вида алгоритма предусмотрена своя конструкция из определенных блоков. Проверка блок-схемы и получение результата достигается при выполнении блок-схемы.

Вопросы

1. Что такое блок-схема?
2. Какие типы блоков бывают?
3. Какие блоки используются при реализации линейного, разветвляющегося, циклического алгоритмов?
4. Можно ли составить разные варианты блок-схем для одной и той же задачи?
5. Какие виды циклического алгоритма бывают?
6. Какие пункты должны присутствовать в любом цикле?
7. Что такое выполнение блок-схемы?
8. Для чего следует выполнять блок-схему?

Упражнения

1. Составьте блок-схемы для задачи по походу в магазин за яблоками. Используйте линейный, разветвляющийся и циклический алгоритмы.
2. Составьте блок-схему для нахождения корней квадратного уравнения через дискриминант. Используйте разветвляющийся алгоритм. Получите ответ, выполнив блок-схему.
3. Составьте блок-схемы для вывода на экран целых чисел от 1 до 10. Используйте цикл с предусловием, с постусловием. Выполните блок-схемы.

2. ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ C++

2.1. Базовые знания о языке программирования C++

Программа – это реализация алгоритма для выполнения задачи компьютером (ЭВМ).

С помощью программы мы формулируем алгоритм на языке, понятном компьютеру. Таким языком служит язык программирования.

На сегодняшний день самым распространенным и востребованным языком программирования является язык C++.

На языке C++ можно составлять программы для инженерных расчетов, также можно строить оконные проекты, имеющие пользовательский графический интерфейс.

2.2. Этапы разработки программ.

В процессе создания любой программы можно выделить несколько этапов:

1) Постановка задачи - выполняется специалистом в предметной области на естественном языке. Необходимо определить цель задачи, ее содержание и общий подход к решению. Возможно, что задача решается точно (аналитически) и без компьютера. Уже на этапе постановки надо учитывать эффективность алгоритма решения задачи на ЭВМ, ограничение, накладываемое аппаратным и программным обеспечением.

2) Анализ задачи и моделирование - определяются исходные данные и результат, выявляются ограничения на их значения, выполняется формализованное описание задачи, и построение (выбор) математической модели пригодном для решения на компьютере.

3) Разработка или выбор алгоритма решения задачи - выполняется на основе ее математического описания. Многие задачи можно решить различными способами. Программист должен выбрать оптимальное решение. Неточность в поставке, анализа задачи, или разработки алгоритма, могут привести к скрытой ошибке - программист получит неверный результат, считая его правильным.

4) Проектирование общей структуры программы - формируется модель решения, с последующей детализацией и разбивкой на подпрограммы, определяется архитектура программы, способ хранения информации (набор переменных, массивов и так далее).

5) Кодирование - запись алгоритма на языке программирования. Современные системы программирования позволяют ускорить процесс

разработки программы, автоматически создавая часть ее текста. Однако творческая работа лежит на программисте. Для успешной реализации цели проекта, программисте необходимо использовать методы структурного программирования.

6) Отладка и тестирование программы. Под отладкой понимается устранение ошибок в программе. Тестирование позволяет вести их поиск и в конечном счете убедиться в том, что полностью отложенная программа дает правильный результат. Для этого разрабатываются системы тестов - специально подобранных контрольных примеров с такими подборками параметров, для которых решение задачи известно. Тестирование должно охватывать все возможные ветвления в программе, то есть проверяют все ее инструкции, и включают такие исходные данные, для которых решение невозможно. Проверка особых исключительных ситуаций необходима для анализа корректности. Например программа должна отказать клиенту банка в просьбе выдать сумму, отсутствующую на его счете. В ответственных проектах большое внимание уделяется так называемой "защите от неумелого пользователя". Подразумевающего устойчивость программы к неумелому обращению пользователя. Использование специальных программ отладчиков, которые позволяют выполнять программу по отдельным шагам, просматривая при этом значения переменных, значительно упрощает этот этап.

7) Анализ результатов - если программа выполняет моделирование какого-либо известного процесса, следует сопоставить результат вычислений с результатами наблюдений. В случае существенного расхождения необходимо изменить модель.

8) Публикация результатов работы, передача заказчику для эксплуатации.

9) Сопровождение программы - включает консультации представителей заказчика по работе с программой и обучению персоналом. Недостатки и ошибки, засеченные в процессе эксплуатации устраняются.

2.3. Типы данных.

Тип определяет множество значений, которые могут принимать объекты программы (константы и переменные). А так же совокупность операций допустимых над этими значениями. Например: значение 1 и 3 относится к целочисленному типу. И над ними можно выполнять любые арифметические операции. Значения "отличное" и "учеба" принадлежат к строковому.

В языке программирования C++ определены некоторые стандартные типы данных, которые представлены в таблице 2.1.

Таблица 2.1. Стандартные типы данных C++

Типы данных	Значения
int	целый тип, размер типа int не определяется стандартом, а зависит от компьютера и компилятора, для 16-разрядного процессора под величины этого типа отводится 2 байта, для 32-разрядного — 4 байта. Примеры значений типа int: 5, 0, -1, 100.
double	вещественный тип с двойной точностью. Типы данных с плавающей точкой хранятся в памяти компьютера иначе, чем целочисленные. Внутреннее представление вещественного числа состоит из двух частей — мантиссы и порядка. Мантисса — это число, большее 1.0, но меньшее 10. Для величин типа double, занимающих 8 байт, под порядок и мантиссу отводится 11 и 52 разряда соответственно. Длина мантиссы определяет точность числа, а длина порядка — его диапазон. Примеры значений типа double: 5.0, -0.00001, 2.9987.
float	вещественный тип. В компьютерах величины типа float занимают 4 байта, из которых один двоичный разряд отводится под знак мантиссы, 8 разрядов под порядок и 23 под мантиссу. Тип float имеет меньшую точность, чем double. В большинстве случаев лучше использовать double.
char	символьный тип, под величину символьного типа отводится количество байт, достаточное для размещения любого символа из набора символов для данного компьютера, что и обусловило название типа. Как правило, это 1 байт. Каждый символ имеет свой собственный целочисленный код, согласно таблице ASCII (англ. American Standard Code for Information Interchange). Примеры значений типа char: "A" (код 65), "7" (код 55), "-" (код 189), "/" (код 191).

Лекция 5

Программа на языке C++ имеет определенную структуру.

Существует определенная последовательность заранее определенных строк кода, которая приведена в таблице 2.2.

<code>#include "stdafx.h"</code>	подключение заголовочного файла для сборки проекта. Обязательный пункт в Visual Studio, в других средах не используется
<code>#include <название_библиотеки></code>	подключение библиотек. Необязательный пункт. Подробно о библиотеках смотреть ниже.
<code>using namespace std;</code>	использование пространства имен.
<code>int main(){ } или void main(){ }</code>	главная функция программы. Именно она начинает выполняться, когда запускается программа. Обязательный пункт.
Тело_функции_main	в теле функции main записываются действия и операции, предусмотренные алгоритмом. Обязательный пункт.
<code>return 0; } или }</code>	конец программы. Самый последний оператор. Обязательный пункт

Библиотека – это файл, в котором описаны функции и операторы. Для каждой смысловой группы функций своя библиотека. Библиотек в языке C++ предусмотрено много, мы будем рассматривать только самые необходимые. Нам понадобятся функции для работы с экраном, математические функции.

Основные библиотеки и их основные функции и операторы:

1. `<iostream>` для VisualStudio. Библиотека для работы с консолью (экраном).

`cout` – оператор вывода данных на экран.

Пример использования:

`cout<<"faza";` //выведет на экран слово faza. Может вывести любой текст.

`cout<<x;` //выведет на экран число, хранящееся в переменной x.

`cin` – оператор считывания с клавиатуры.

Когда у пользователя запрашивают число, программа ждет, пока пользователь не напечатает число и нажмет ENTER. Тогда оператор `cin` записывает это значение в переменную x.

Пример использования:

`cin>>x; //присваивает переменной x значение, введенное с клавиатуры.`

`cin>>x>>y; //присваивает переменной x первое введенное с клавиатуры значение, переменной y – второе.`

`endl` – оператор перевода каретки на экране на следующую строку.

Самостоятельно не используется.

Пример использования:

`cout<<endl; //курсор перейдет на новую строку.`

`cout<<x<<endl;` //сначала на экране появится число, хранящееся в переменной, потом перейдет на новую строку. Выводимые далее данные будут печататься с новой строки.

`cout<<endl<<"faza";` // курсор перейдет на новую строку, и на новой строке появится надпись `faza`.

`precision(n)` – функция для отображения на экране дробных чисел с `n` цифрами после запятой.

Пример использования:

`cout.precision(3)<<7.897426;` //число 7.897426 выведется на экран в виде 7.897.

2. `<math.h>` - библиотека математических функций. Основные математические функции представлены в табл. 2.3

Таблица 2.3. Основные математические функции C++

Математическая функция	Программная запись	Описание
$ x $	<code>abs(x)</code>	Модуль числа.
$\sin x$	<code>sin(x)</code>	Синус числа, аргумент в радианах.
$\cos x$	<code>cos(x)</code>	Косинус числа, аргумент в радианах
$\operatorname{tg} x$	<code>tan(x)</code>	Тангенс числа, аргумент в радианах.
e^x	<code>exp(x)</code>	Экспонента числа.
$\ln x$	<code>log(x)</code>	Натуральный логарифм числа.
$\lg x$	<code>log10(x)</code>	Десятичный логарифм

		добавить числа
x^y	<code>pow(x, y)</code>	х в степени у.
\sqrt{x}	<code>sqrt(x)</code>	Квадратный корень из числа.
<code>arcsin x</code>	<code>asin(x)</code>	Арксинус числа, в радианах.
<code>arcos x</code>	<code>acos(x)</code>	Арккосинус числа, в радианах.
<code>arctg x</code>	<code>atan(x)</code>	Арктангенс числа, в радианах.
π	<code>M_PI</code>	Число $\pi = 3.141593$

3. <iomanip> для Visual Studio.

`setw(n)` – для вывода на экран отводится ячеек.

Используется при построении ровной таблицы значений функции.

Пример использования:

```
cout<<setw(5)<<x<<setw(5)<<y<<endl;
```

На экране выведутся два числа: первое (1.5) в первых пяти ячейках, второе (-73) во вторых пяти ячейках. `__ 1 . 5 __ - 7 3 .`

В программах используются переменные. Имя переменной выбирает составитель программы; имя переменной должно начинаться с буквы латинского алфавита и может содержать буквы латинского алфавита, цифры и символы подчеркивания. Заглавные и строчные буквы считаются разными. Примеры имен переменных: `x`, `y`, `summa`, `s1`, `srednee_ar` и т.д. Имена переменных не должны совпадать с ключевыми словами языка C++.

Чтобы использовать в программе переменную, необходимо:

1. Объявить переменную в начале программы, явно указав тип данных для переменной.

Пример:

```
double x; //вещественная переменная.
```

```
int m; //целочисленная переменная.
```

Если переменная не будет объявлена, но будет использоваться далее в программе, то программа не запустится, компилятор выдаст ошибку.

2. Проинициализировать переменную, т.е. задать переменной значение.

Пример:

```
x = 7.81;
```

$m = 4; z = x + m;$

Если переменная не будет проинициализирована, то компилятор не выдаст ошибки, но расчеты будут выполнены неверно.

3. Использовать далее в программе в расчетах или при выводе на экран.

Для числовых переменных определены простейшие арифметические операции, которые приведены в таблице 2.4. Для их использования не нужно подключать библиотеку.

Таблица 2.4. Логические операции C++

Сравнение в C++	Описание	Пример в программе
>	больше чем	$x > 0$
<	меньше чем	$y < z$
>=	больше или равно	$y \geq x$
<=	меньше или равно	$z \leq 8.56$
==	проверка на равенство	$x == 0.7$
!=	не равно	$x != y$
&&	логическое И	$x > 0 \ \&\& \ x < 1$ // двойное неравенство $0 < x < 1$
	логическое ИЛИ	$s < 8 \ \ s > 10$

2.4. Рекомендации по стилю программирования

Накопленный опыт программирования привел к формированию следующих рекомендаций по составлению наглядных и легко читаемых программ.

1) Стандартизация стиля программирования заключается в том, что необходимо всегда придерживаться одному способу программирования, записи программы.

2) С целью рационального размещения текста, не следует операторам программы писать сплошным текстом.

Для четкого выявления вложенности управляющих структур, требуется особым образом располагать операторы в тексте так, что служебные слова, которые начинаются или заканчиваются той или иной оператор, записывается на одной вертикали, а все вложенные на него операторы записывается с некоторым отступом вправо. При записи конструкции языка более глубоких уровней вложенности, следует сдвигать их от начала строки вправо. Каждое описание и каждый оператор следует писать с новой строки. Продолжение

описания и операторов на новые строки, надо сдвигать вправо. Следует избегать длинных строк.

3) Рекомендуются любую программу сопровождать комментарием. Поясняющее назначение всей программы и отдельных ее блоков, процедурам функций.

4) Имена для объектов программы надо выбирать так, что бы они наилучшим образом соответствовали этим объектам, отражали их назначение.

5) Списки идентификаторов в блоках описания следует упорядочивать - это облегчает поиск в них нужных элементов.

6) Программирование сверху вниз. В процессе разработки алгоритма и программы следует начинать с самой общей модели решения, постепенно уточняя ее до уровня отдельного блока, а затем детально прорабатывая отдельный блок.

2.5. Трансляторы. Синтаксис и семантика

Особое значение для программиста имеет предупреждение и исправления ошибок в алгоритме и программе решения задачи. Прежде чем выполнит программу, ее текст необходимо ввести в компьютер. Для ввода и изменения (редактирование) текстов, используется специальная программа - текстовый редактор.

Так как текст записан на языке программы и непонятен компьютеру, то требуется перевести ее на машинный язык. Такой перевод программы с языка программирования на язык машинных кодов называется трансляцией, а выполняется специальной программой - транслятором.

Существует три вида трансляторов: компиляторы, ассемблера, интерпретаторы.

Интерпритатором называется транслятор, про изводящий пооператорную (покомандую) обработку и выполнение исходной программы.

Компиляторы преобразуют (транслирует) всю программу в модуль на машинном языке, после этого программа записывается в память компьютера и лишь потом исполняется. Компиляторы анализируют программу и определяет содержит ли она ошибки. В случае их обнаружения вся работа останавливается. Если же правила языка не нарушены, то формируется модуль на машинном языке, который затем и исполняется.

Ассемблеры переводят программу, записанную на языке ассемблера (авто коды) в программу на машинном языке.

Любой транслятор решает следующие основные задачи:

1) Анализирует транслируемую программу, в частности определяет, содержит ли она синтаксические ошибки;

2) Генерирует выходную программу (ее часто называют объектной или рабочей) на языке команд ЭВМ (в некоторых случаях транслятор генерирует выходную программу на промежуточном языке, например, на языке ассемблера);

3) Распределяет память для выходной программы (в простейшем случае это заключается в назначении каждому фрагменту программы, переменной, константам, массивам и другим объектам своих адресов участков памяти).

В отличие от естественных языков таких как русский английский и так далее, язык программирования имеет очень ограниченное количество слов, понятных компиляторам, и строгие записи команд. Совокупность этих требований образует синтаксис языка программирования, а смысл команд других конструкций языка - его семантику.

2.6. Константа

Каждый элемент данных, используемый в программе, является либо константой, либо переменной. Константой называется элемент данных, значения которых в процессе выполнения программы не меняются. В языке турбо Паскаль используются константы следующих видов: числовые, логические (булевские), символьные, строковые. Числовые константы предназначены для представления числовых данных (целых и вещественных). Булевские константы используются для представления данных, имеющих смысл логических высказываний (типа: да-нет, истина-ложь, 1-0). Символьные и строковые константы - это отдельные символы и их последовательность.

2.7. Инструкции. Операторы

Алгоритм решения любой задачи состоит из отдельных довольно мелких шагов. В программе для каждого шага алгоритма записывается отдельная инструкция (команда) записывается так же для организации ветвлений и циклов. Таким образом программа состоит из отдельных инструкций или команд. Эти инструкции в программировании принято называть операторами. Часто в литературе по программированию программу определяют, как последовательность операторов.

Операторы могут объединяться в более крупные конструкции - составные операторы, процедуры и функции. Такие конструкции состоят из нескольких элементарных операторов, однако в программе могут использоваться как один оператор.

Процедуры и функции универсального назначения могут располагаться в особом образом оформленных файлах - библиотечных модулей.

3. ВВОД, ВЫВОД В C++

3.1. Вывод с помощью cout

C++ рассматривает вывод как поток байтов. (В зависимости от реализации и платформы это могут быть 8-, 16- или 32-битные байты, но все равно они будут байтами.) Однако многие виды данных в программе организованы в виде более крупных блоков, нежели отдельный байт. Например, тип `int` может быть представлен 16- или 32-битным двоичным значением, а значение типа `double` — 64- битными двоичными данными. Но при отправке потока байтов на экран желательно, чтобы каждый байт представлял значение символа. То есть для отображения на экране числа -2.34 понадобится отправить пять символов: -, 2, ., 3 и 4, а не внутреннее 64-битное с плавающей точкой представление этого значения. Поэтому одной из наиболее важных задач, стоящих перед классом `ostream`, является преобразование числовых типов, таких как `int` или `float`, в поток символов, который представляет значения в текстовой форме. Таким образом, класс `ostream` транслирует внутреннее представление данных в виде двоичных битовых последовательностей в выходной поток символьных байтов. Для выполнения такой трансляции в классе `ostream` предусмотрено несколько методов. В этой главе мы рассмотрим их, резюмируя методы, которые используются на протяжении всей книги, и описывая дополнительные методы, обеспечивающие более тонкое управление выводом.

3.2. Перегруженная операция <<

Чаще всего в этой книге мы используем `cout` с операцией `<<`, которая также называется операцией вставки:

```
int clients = 22;

cout << clients;
```

В языке C++, как и в C, по умолчанию операция `<<` используется в качестве битовой операции сдвига (см. приложение Д). Выражение `x << 3` означает: загрузить двоичное представление `x` и сдвинуть все его биты на три позиции влево. Очевидно, что это не слишком тесно связано с выводом. Но класс `ostream` переопределяет операцию `<<` для вывода. В этой ипостаси

операция << называется операцией вставки, а не операцией сдвига влево. (Операция сдвига влево обретает эту новую роль посредством своего визуального аспекта, который предполагает смещение информации влево.) Операция вставки перегружена для применения со всеми базовыми типами C++:

- unsigned char
- signed char
- char
- short
- unsigned short
- int
- unsigned int
- long
- unsigned long
- long long (C++11)
- unsigned long long (C++11)
- float
- double
- longdouble

Класс ostream предоставляет определение функции operator<<() для каждого из этих типов данных. (Функции, имена которых содержат слово operator, используются для перегрузки операций, как описано в главе 11.) Таким образом, если применять оператор показанной ниже формы, и если значение относится к одному из перечисленных ранее типов, программа на языке C++ может сопоставить его с функцией операции с соответствующей сигнатурой:

```
cout << значение;
```

Например, выражение `cout << 88` соответствует следующему прототипу метода:

```
ostream & operator<<(int) ;
```

Как вы должны помнить, такой прототип указывает, что функция `operator<< ()` принимает один аргумент типа `int`. Это соответствует параметру 88 в предыдущем операторе. Прототип указывает также, что функция возвращает ссылку на объект `ostream`. Это свойство позволяет группировать вывод, как показано в следующем примере:

```
cout << "I'm feeling sedimental over " << boundary << "\n";
```

Если вы — программист на C и страдаете от многообразия спецификаторов типа `%` и проблем, связанных с несоответствием спецификатора действительному типу значения, то использование `cout` покажется вам до неприличия простым. (Разумеется, как и ввод C++ посредством `cin`.)

Другие методы ostream

Помимо разнообразных функций `operator<< ()`, класс `ostream` предоставляет метод `put ()` для отображения символов и метод `write ()` для отображения строк.

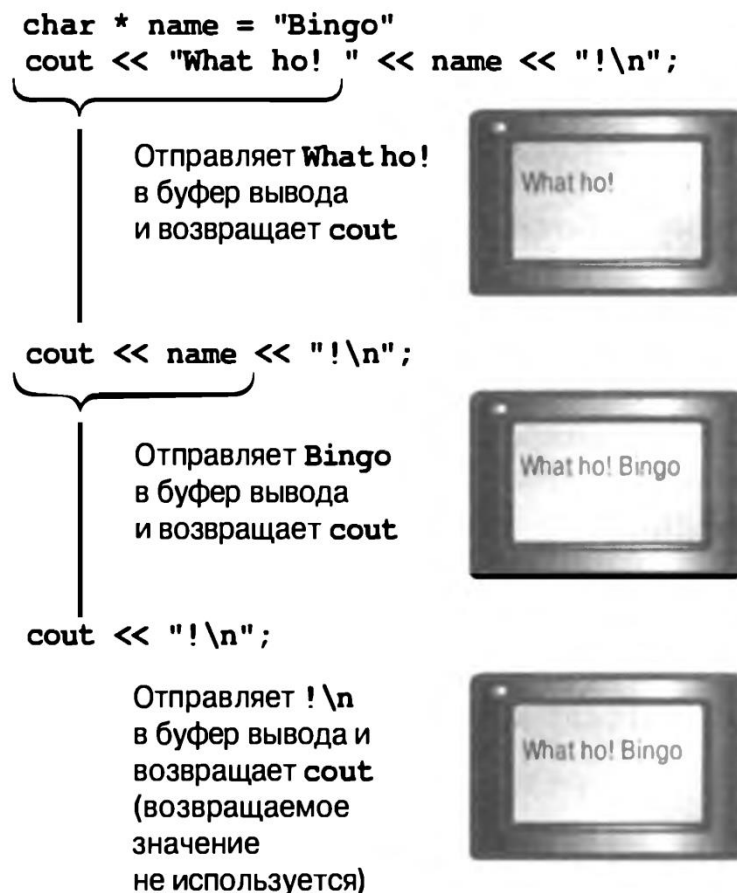


Рис. 17.4. Конкатенация вывода

Изначально метод `put ()` имел следующий прототип:

```
ostream & put(char);
```

Текущий стандарт эквивалентен первоначальному, за исключением того, что теперь метод реализован в виде шаблона, чтобы был возможен вывод значений типа `wchart`. Его вызов выполняется с использованием обычной нотации методов класса:

```
cout.put ('W'); // отображение символа W
```

Здесь `cout` — вызывающий объект, а `put ()` — функция-член класса. Подобно функциям операции `<<`, эта функция возвращает ссылку на вызывающий объект, поэтому можно выполнить ее конкатенацию с выводом:

```
cout.put('I').put ('t'); // отображение строки It с помощью  
// двух вызовов функции put()
```

Вызов функции `cout.put (' I.1)` возвращает объект `cout`, который затем служит вызывающим объектом для вызова функции `put (' t')`.

Располагая соответствующим прототипом, функцию `put ()` можно применять с аргументами числовых типов, отличных от `char`, таких как `int`, и позволять прототипируемой функции автоматически преобразовывать аргумент к корректному значению типа `char`. Например, можно использовать следующие операторы:

```
cout.put(65); // отображает символ A
```

```
cout.put(66 . 3) ; // отображает символ B
```

Первый оператор преобразует целочисленное значение 65 в значение типа `char` и отображает символ, имеющий ASCII-код 65. Аналогично, второй оператор преобразует значение 66.3 типа `double` в значение 66 типа `char` и отображает соответствующий символ.

Такое поведение полезно при работе с версиями, предшествующими C++ Release 2.0. В этих версиях языка символьные константы представляются как значения типа `int`. Поэтому оператор, подобный показанному ниже, интерпретировал бы `'W1` как значение типа `int`, и, следовательно, отобразил бы его в виде целочисленного значения 87 — ASCII-кода символа:

```
cout << 'W ;
```

Но следующий оператор работает нормально:

```
cout.put CW);
```

Поскольку в текущей версии C++ константы `char` представляются типом `char`, можно использовать любой из описанных методов.

Некоторые компиляторы ошибочно перегружают `put ()` для трех типов аргументов: `char`, `unsignedchar` и `signedchar`. Это приводит к неоднозначности при вызове `put ()` с аргументом типа `int`, поскольку `int` может быть преобразован в любой из этих трех типов.

Метод `write ()` записывает целую строку и имеет следующий шаблонный прототип:

```
basic_ostream<charT,traits>& write(const char_type* s, streamsize n) ;
```

Первый аргумент `write ()` представляет адрес строки, которую нужно отобразить, а второй аргумент указывает количество отображаемых символов. Использование `cout` для вызова `write ()` вызывает специализацию `char`, поэтому возвращаемым типом будет `ostream &`. В листинге ниже показано, как работает метод `write()`.

```
#include<iostream>

#include<cstring> // или иначестring.h

int main()

{

using std::cout;

using std::endl;

const char * statel = "Florida";

const char * state2 = "Kansas";

const char * state3 = "Euphoria";

int len = std::strlen (state2);

cout << "Increasing loop index:\n";
```

```

int i;

for (i = 1; i <= len; i + + )

{

cout.write (state2,i);

cout « endl;

}

// Конкатенация вывода

cout « "Decreasing loop index:\n";

for (i = len; i > 0; i —)

cout.write(state2,i) << endl;

// Превышение длины строки

cout « "Exceeding string length:\n";

cout .write (state2, len + 5) << endl;

return 0;

}

```

Некоторые компиляторы могут обнаружить, что программа объявляет, но не использует массивы state1 и state3. Все в порядке, поскольку эти два массива служат лишь для представления данных, находящихся в памяти перед и после массивом state2, чтобы можно было выяснить, что происходит в случае ошибки доступа к state2. Вывод программы из листинга имеет следующий вид:

Increasing loop index:

K

Ka

Kap

Kans

Kansa

Kansas

Decreasing loop index:

Kansas

Kansa

Kans

Kan

Ka

K

Exceeding string length: Kansas Euph

Обратите внимание, что вызов `cout.write()` возвращает объект `cout`. Это обусловлено тем, что метод `write()` возвращает ссылку на объект, который его вызвал, и в данном случае это — объект `cout`.

Это позволяет выполнять конкатенацию вывода, поскольку `cout.write()` замещается возвращаемым значением — объектом `cout`:

```
cout.write(state2,i) << endl;
```

Вдобавок метод `write ()` не прекращает вывод строки по достижении нулевого ограничивающего символа. Он просто выводит указанное количество символов, даже если при этом выходит за пределы конкретной строки! В данном случае программа объединяет строку "Kansas" с двумя другими строками, как если бы соседние области памяти содержали единый элемент данных. Компиляторы могут по-разному размещать данные в памяти и по-разному выравнивать ее содержимое. Например, "Kansas" занимает 6 байтов, но, судя по всему, данный конкретный компилятор выравнивает строки, используя значения кратные 4 байтам, поэтому "Kansas" дополняется до 8 байт. Некоторые компиляторы разместят "Florida" после "Kansas". Поэтому из-за различий в компиляторах последняя строка вывода может выглядеть по-разному

Метод `write ()` можно также использовать с числовыми данными. В этом случае ему нужно передать адрес числа, приведя его тип к `char *`:

```
long val = 560031841;
```

```
cout.write ((char *) &val, sizeof (long));
```


Это не ведет к трансляции числа в корректные символы; вместо этого такой вызов передает битовое представление хранящихся в памяти данных. Например, 4-байтное значение типа `long`, такое как 560031841, будет передано в виде четырех отдельных байтов. Выходное устройство, подобное монитору, может затем попытаться интерпретировать каждый байт, как если бы он был ASCII-кодом (или каким-то другим). Поэтому число 560031841 отобразилось бы на экране в виде некоторой 4-символьной комбинации, скорее всего, бессмысленной (а может, и нет — проверьте и выясните). Однако `write ()` обеспечивает компактный, аккуратный способ сохранения числовых данных в файле. Мы вернемся к этой возможности позднее в настоящей главе.

Лекция 7

3.3. Ввод с помощью `cin`

Теперь обратимся к вводу и передаче данных в программу. Объект `cin` представляет стандартный ввод в виде потока байтов. Обычно этот поток символов генерируется клавиатурой. При вводе последовательности символов объект `cin` извлекает эти символы из входного потока. Этот ввод может являться частью строки, значением типа `int`, типа `float` или какого-либо иного типа. Таким образом, извлечение символов из потока предполагает также преобразование типа. Объект `cin` на основании типа переменной, предназначенной для приема значения, должен применять свои методы для преобразования последовательности символов в значения соответствующего типа. Обычно `cin` используют следующим образом:

```
cin >> value_holder;
```

Здесь `value_holder` идентифицирует область памяти, в которую помещается ввод. Это может быть именем переменной, ссылкой, разыменованным указателем либо членом структуры или класса. То, как `cin` интерпретирует ввод, зависит от типа данных `value_holder`. Класс `istream`, определенный в заголовочном файле `iostream`, перегружает операцию извлечения `>>` для распознавания следующих базовых типов:

- `signed char &`
- `unsigned char &`

- char &
- short &
- unsigned short &
- int &
- unsigned int &
- long &
- unsigned long &
- long long & (C++11)
- unsigned long long & (C++11)
- float &
- double &
- long double &

Их называют функциями форматированного ввода, потому что они преобразуют входные данные в соответствии с переменной назначения.

Типичная функция операции имеет прототип следующего вида:

```
istream & operator>>(int &) ;
```

И аргумент, и возвращаемое значение являются ссылками. При наличии аргумента-ссылки (см. главу 8) оператор, подобный показанному ниже, вынуждает функцию `operator>>()` работать с самой переменной `staff_size`, а не с ее копией, как это имеет место при использовании обычного аргумента:

```
cin >> staff_size;
```

Поскольку тип аргумента является ссылкой, `cin` в состоянии непосредственно модифицировать значение переменной, применяемой в качестве аргумента.

Например, предыдущий оператор непосредственно модифицирует значение переменной `staff_size`. О важности использования ссылки в качестве возвращаемого значения мы поговорим немного позже. Сначала необходимо исследовать аспект преобразования типа операцией извлечения. Для обработки аргументов каждого типа из приведенного выше списка операция извлечения

преобразует символьный ввод в значение указанного типа. Например, предположим, что `staff_size` имеет тип `int`.

В этом случае компилятор сопоставляет

```
cin » staff_size;
```

со следующим прототипом:

```
istream & operator>>(int &) ;
```

Затем функция, соответствующая прототипу, считывает поток символов, отправляемый программе — например, символы 2, 3, 1, 8 и 4. Для систем, использующих двухбайтный тип `int`, функция преобразует эти символы в двухбайтное двоичное представление целочисленного значения 23184. С другой стороны, если бы переменная `staff_size` имела тип `double`, объект `cin` использовал бы `operator>> (double &)`, чтобы преобразовать этот же ввод в восьмибайтное представление значения с плавающей точкой 23184.0.

Кстати, вместе с `cin` можно применять манипуляторы `hex`, `oct` и `dec` для указания того, что вводимое целое должно интерпретироваться в шестнадцатеричном, восьмеричном или десятичном формате. Например, следующий оператор приводит к тому, что ввод 12 или 0x12 воспринимается как шестнадцатеричное значение 12, или десятичное 18, а `ff` или `FF` считывается как десятичное 255:

```
cin >> hex;
```

Класс `istream` также перегружает операцию извлечения `>>` для типов указателей на символы:

- `signedchar *`
- `char *`
- `unsignedchar *`

Для аргументов этих типов операция извлечения считывает следующее слово из входного потока и помещает его по указанному адресу, добавляя нулевой символ для ограничения строки. Например, предположим, что имеется следующий код:

```
// Вывод приглашения на ввод имени
```

```
cout << "Enter your first name:\n";
```

```
charname[20] ;
```

```
// Вводимени
```

```
cin » name;
```

Если вы ответите на запрос вводом Liz, то операция извлечения поместит символы Liz\0 в массив name. (Как обычно, \0 представляет завершающий нулевой символ.) Идентификатор name, будучи именем массива типа char, действует как адрес первого элемента массива и имеет тип char * (указатель на массив типа char).

То, что каждая операция извлечения возвращает ссылку на вызывающий объект, позволяет выполнять конкатенацию ввода, подобно тому, как это делается в отношении вывода:

```
char name[20];
```

```
float fee;
```

```
int group;
```

```
cin >> name >> fee » group;
```

Здесь, к примеру, объект cin, возвращенный операцией cin » name, становится тем объектом, который принимает fee.

4. КОНСТРУКЦИЯ ВЕТВЛЕНИЯ В C++

4.1. Оператор if

Когда программа C++ должна принять решение о том, какое из альтернативных действий следует выполнить, такой выбор обычно реализуется оператором if. Этот оператор имеет две формы: просто if и if else. Давайте сначала исследуем простой if. Он создан по образцу обычного английского языка, как в выражении "If you have aCaptainCookiecard, yougetafreecookie" (игра слов на основе созвучности фамилии Кук и слова "cookie" (печенье) — прим. перев.). Оператор if разрешает программе выполнять оператор или блок операторов при условии истинности проверочного условия, и пропускает этот оператор или блок, если проверочное условие оценивается как ложное. Таким образом, оператор if позволяет программе принимать решение относительно того, нужно ли выполнять некоторую часть кода.

Синтаксис оператора if подобен while:

if (проверочное-условие)

оператор;

Оператор `if` служит для того, чтобы выполнить какую-либо операцию в том случае, когда условие является верным. Условная конструкция в C++ всегда записывается в круглых скобках после оператора `if`.

Внутри фигурных скобок указывается тело условия. Если условие выполнится, то начнется выполнение всех команд, которые находятся между фигурными скобками.

Пример конструкции ветвления

```
#include<iostream>

Usingnamespacestd;

Intmain() {

setlocale(0, "");

doublenum;

cout<<"Введите произвольное число: ";

cin>>num;

if (num <10) { // Если введенное число меньше 10.

cout<<"Это число меньше 10."<< endl;

}

else{ // иначе

cout<<"Это число больше либо равно 10."<< endl;

}

return0;

}
```

Если вы запустите эту программу, то при вводе числа, меньшего десяти, будет выводиться соответствующее сообщение.

Если введенное число окажется большим, либо равным десяти — отобразится другое сообщение.

```
if (num < 10) { // Если введенное число меньше 10.  
    cout << "Это число меньше 10." << endl;  
}  
else { // иначе  
    cout << "Это число больше либо равно 10." << endl;  
}
```

Здесь говорится: «Если переменная num меньше 10 — вывести соответствующее сообщение. Иначе, вывести другое сообщение».

Усовершенствуем программу так, чтобы она выводила сообщение, о том, что переменная num равна десяти:

```
if (num < 10) { // Если введенное число меньше 10.  
    cout << "Это число меньше 10." << endl;  
}  
elseif (num == 10) {  
    cout << "Это число равно 10." << endl;  
}  
else { // иначе  
    cout << "Это число больше 10." << endl;  
}
```

Здесь мы проверяем три условия:

Первое — когда введенное число меньше 10-ти

Второе — когда число равно 10-ти

Третье — когда число больше десяти

Заметьте, что во втором условии, при проверке равенства, мы используем оператор равенства `==`, а не оператор присваивания, потому что мы не изменяем значение переменной при проверке, а сравниваем ее текущее значение с числом 10.

Если поставить оператор присваивания в условии, то при проверке условия, значение переменной изменится, после чего это условие выполнится.

Каждому оператору `if` соответствует только один оператор `else`. Совокупность этих операторов — `else if` означает, что если не выполнилось предыдущее условие, то проверить данное. Если ни одно из условий не верно, то выполняется тело оператора `else`.

Если после оператора `if`, `else` или их связки `else if` должна выполняться только одна команда, то фигурные скобки можно не ставить. Предыдущую программу можно записать следующим образом:

```
#include<iostream>

Usingnamespacestd;

Intmain() {
    setlocale(0, "");
    doublenum;

    cout<<"Введите произвольное число: ";
    cin>> num;

    if (num <10) // Если введенное число меньше 10.
        cout<<"Это число меньше 10."<< endl;
    elseif (num == 10)
        cout<<"Это число равно 10."<< endl;
    else// иначе
        cout<<"Это число больше 10."<< endl;

    return0;
}
```

Такой метод записи выглядит более компактно. Если при выполнении условия нам требуется выполнить более одной команды, то фигурные скобки необходимы. Например:

```
#include<iostream>

Usingnamespacestd;

Intmain() {
    setlocale(0, "");
    double num;
    intk;

    cout<<"Введите произвольное число: ";
    cin>> num;

    if (num <10) { // Если введенное число меньше 10.
        cout<<"Это число меньше 10."<< endl;
        k = 1;
    }

    elseif (num == 10) { cout<<"Это число равно 10."<< endl;
        k = 2;
    }

    else{ // иначе
        cout<<"Это число больше 10."<< endl;
        k = 3;
    }

    cout<<"k = "<< k << endl;

    return0;
}
```


Данная программа проверяет значение переменной `num`. Если она меньше 10, то присваивает переменной `k` значение единицы. Если переменная `num` равна десяти, то присваивает переменной `k` значение двойки. В противном случае — значение тройки. После выполнения ветвления, значение переменной `k` выводится на экран.

4.2. Оператор `switch`

Предположим, что вы создаете экранное меню, которое предлагает пользователю на выбор один из четырех возможных вариантов, например, "дешевый", "умеренный", "дорогой", "экстравагантный" и "непомерный". Вы можете расширить последовательность `ifelseifelse` для обработки этих пяти альтернатив, но оператор C++ `switch` упрощает обработку выбора из большого списка. Ниже представлена общая форма оператора `switch`:

```
switch (целочисленное-выражение)
{
    case метка 1 : оператор (ы)
    case метка2 : оператор (ы)
    ...
    default : оператор (ы)
}
```

Оператор `switch` действует подобно маршрутизатору, который сообщает компьютеру, какую строку кода выполнять следующей. По достижении оператора `switch` программа переходит к строке, которая помечена значением, соответствующим текущему значению целочисленное-выражение. Например, если целочисленное - выражение имеет значение 4, то программа переходит к строке с меткой `case 4:`. Как следует из названия, выражение целочисленное-выражение должно быть целочисленным. Также каждая метка должна быть целым константным выражением. Чаще всего метки бывают константами типа `char` или `int`, такими как 1 или 'q', либо же перечислителями. Если целочисленное-выражение не соответствует ни одной метке, программа переходит к метке `default`. Метка `default` не обязательна. Если она опущена, а соответствия не найдено, программа переходит к оператору, следующему за `switch` (рис. 4.1.).

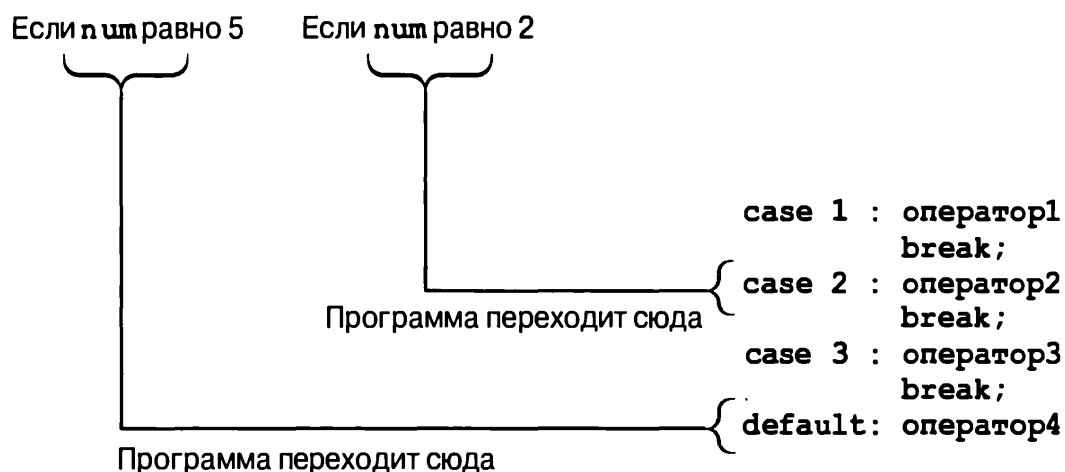


Рис. 4.1. Структура оператора switch

Оператор switch в C++ отличается от аналогичных операторов в других языках, например, Pascal, в одном очень важном отношении. Каждая метка case в C++ работает только как метка строки, а не граница между выборами.

То есть после того, как программа перейдет на определенную строку в switch, она последовательно выполнит все операторы, следующие за этой строкой внутри switch, если только вы явно не направите ее в другое место. Выполнение не останавливается автоматически на следующем case. Чтобы прекратить выполнение в конце определенной группы операторов, вы должны использовать оператор break. Это передаст управление за пределы блока switch.

Домашнее задание № 1

Задание «Имя». Написать программу, которая выводит на экран Ваше имя.

Задание «Арифметика». Ввести с клавиатуры два числа и найти их сумму, разность, произведение и, если возможно, частное от деления одного на другое.

Задание «Уравнение». Для любых введенных с клавиатуры *b* и *c* решить уравнение вида $bx + c = 0$.

Задание «Еще уравнение». Для любых введенных с клавиатуры *a*, *b* и *c* решить уравнение вида

$$ax^2 + bx + c = 0.$$

Задание «Лампа со шторой». В комнате светло, если на улице день и раздвинуты шторы или если включена лампа. Ваша программа должна, в зависимости от времени суток и состояния лампы и штор, отвечать на вопрос, светло ли в комнате.

5. ЦИКЛЫ В C++

5.1. Цикл for

Обстоятельства часто требуют от программ выполнения повторяющихся задач, таких как сложение элементов массивов один за другим или 20-кратная распечатка похвалы за продуктивность. Цикл `for` облегчает выполнение задач подобного рода.

Если мы знаем точное количество действий (итераций) цикла, то можем использовать цикл `for`. Синтаксис его выглядит примерно так:

```
for (действие до начала цикла;      условие продолжения цикла;
     действия в конце каждой итерации цикла) {      инструкция цикла;
инструкция цикла 2;      инструкция цикла N; }
```

Итерацией цикла называется один проход этого цикла

Существует частный случай этой записи, который мы сегодня и разберем:

```
for (счетчик = значение; счетчик < значение; шаг цикла) {тело цикла }
```

Счетчик цикла — это переменная, в которой хранится количество проходов данного цикла. Описание синтаксиса

1. Сначала присваивается первоначальное значение счетчику, после чего ставится точка с запятой.

2. Затем задается конечное значение счетчика цикла. После того, как значение счетчика достигнет указанного предела, цикл завершится. Снова ставим точку с запятой.

3. Задаем шаг цикла. Шаг цикла — это значение, на которое будет увеличиваться или уменьшаться счетчик цикла при каждом проходе. Пример кода

Напишем программу, которая будет считать сумму всех чисел от 1 до 1000.

```
#include<iostream>
```

```

Usingnamespacestd;

intmain() {

inti; // счетчикцикла

intsum = 0; // суммачиселот 1 до 1000.

setlocale(0, "");

for (i = 1; i <= 1000; i++) // задаем начальное значение 1, конечное 1000 и
задаем шаг цикла - 1.

{

sum = sum + i;

}

cout<<"Сумма чисел от 1 до 1000 = "<<sum<<endl;

return0;

}

```

Если мы скомпилируем этот код и запустим программу, то она покажет нам ответ: 500500. Это и есть сумма всех целых чисел от 1 до 1000. Если считать это вручную, понадобится очень много времени и сил. Цикл выполнил всю рутинную работу за нас.

Заметьте, что конечное значение счетчика я задал нестрогим неравенством (\leq — меньше либо равно), поскольку, если бы я поставил знак меньше, то цикл произвел бы 999 итераций, т.е. на одну меньше, чем требуется. Это довольно важный момент, т.к. здесь новички часто допускают ошибки, особенно при работе с массивами (о них будет рассказано в следующем уроке). Значение шага цикла я задал равное единице. $i++$ — это тоже самое, что и $i = i + 1$.

В теле цикла, при каждом проходе программа увеличивает значение переменной `sum` на `i`. Еще один очень важный момент — в начале программы я присвоил переменной `sum` значение нуля. Если бы я этого не сделал, программа вылетела бы в сегфолт. При объявлении переменной без ее инициализации что эта переменная будет хранить «мусор».

Естественно, к мусору мы ничего прибавить не можем. Некоторые компиляторы, такие как g++, инициализируют переменную нулем при ее объявлении.

5.2. Цикл `while`

Когда мы не знаем, сколько итераций должен произвести цикл, нам понадобится цикл `while` или `do...while`. Синтаксис цикла `while` в C++ выглядит следующим образом.

```
while (Условие) { Тело цикла; }
```

Данный цикл будет выполняться, пока условие, указанное в круглых скобках, является истиной. Решим ту же задачу с помощью цикла `while`. Хотя здесь мы точно знаем, сколько итераций должен выполнить цикл, очень часто бывают ситуации, когда это значение неизвестно.

Ниже приведен исходный код программы, считающей сумму всех целых чисел от 1 до 1000.

```
#include<iostream>

using namespace std;

int main() {
    setlocale(0, "");
    int i = 0; // инициализируем счетчик цикла.
    int sum = 0; // инициализируем счетчик суммы.
    while (i < 1000)
    {
        i++;
        sum += i;
    }
    cout<<"Сумма чисел от 1 до 1000 = "<<sum<<endl;
    return 0;
}
```

После компиляции программа выдаст результат, аналогичный результату работы предыдущей программы. Но поясним несколько важных моментов. Я задал строгое неравенство в условии цикла и инициализировал счетчик `i` нулем, так как в цикле `while` происходит на одну итерацию больше, потому он будет выполняться, до тех пор, пока значение счетчика перестает удовлетворять условию, но данная итерация все равно выполнится. Если бы мы поставили нестрогое неравенство, то цикл бы закончился, когда переменная `i` стала бы равна 1001 и выполнилось бы на одну итерацию больше.

Теперь давайте рассмотрим по порядку исходный код нашей программы. Сначала мы инициализируем счетчик цикла и переменную, хранящую сумму чисел.

В данном случае мы обязательно должны присвоить счетчику цикла какое-либо значение, т.к. в предыдущей программе мы это значение присваивали внутри цикла `for`, здесь же, если мы не инициализируем счетчик цикла, то в него попадет «мусор» и компилятор в лучшем случае выдаст нам ошибку, а в худшем, если программа соберется — дефолт практически неизбежен.

Затем мы описываем условие цикла — «пока переменная `i` меньше 1000 — выполняй цикл». При каждой итерации цикла значение переменной-счетчика `i` увеличивается на единицу внутри цикла.

Когда выполнится 1000 итераций цикла, счетчик станет равным 999 и следующая итерация уже не выполнится, поскольку 1000 не меньше 1000. Выражение `sum += i` является укороченной записью `sum = sum + i`.

После окончания выполнения цикла, выводим сообщение с ответом.

5.3. Цикл `do while`

Цикл `do while` очень похож на цикл `while`. Единственное их различие в том, что при выполнении цикла `do while` один проход цикла будет выполнен независимо от условия. Решение задачи на поиск суммы чисел от 1 до 1000, с применением цикла `do while`.

```
#include<iostream>
```

```
Usingnamespacestd;
```

```
Intmain() {
```

```
setlocale(0, "");
```

```

int i = 0; // инициализируем счетчик цикла.

int sum = 0; // инициализируем счетчик суммы.

do { // выполняем цикл.

    i++;

    sum += i;

}

while (i < 1000); // пока выполняется условие.

cout << "Сумма чисел от 1 до 1000 = " << sum << endl;

return 0;

}

```

Принципиального отличия нет, но если присвоить переменной *i* значение, большее, чем 1000, то цикл все равно выполнит хотя бы один проход.

6. ОПЕРАЦИИ ИНКРЕМЕНТА И ДЕКРЕМЕНТА

Язык C++ снабжен несколькими операциями, которые часто используются в циклах; давайте потратим немного времени на их изучение. Вы уже видели две из них: операция инкремента (+ +), которая получила отражение в самом названии C++, а также операция декремента (--). Эти операции выполняют два чрезвычайно часто встречающихся действия в циклах: увеличивают и уменьшают на единицу значение счетчика цикла. Однако к тому, что вы уже знаете о них, есть что добавить. Каждая из них имеет два варианта. Префиксная версия операции указывается перед операндом, как в ++x. Постфиксная версия следует после операнда, как в x++. Эти две версии имеют один и тот же эффект для операнда, но отличаются в контексте применения. Все похоже на получение оплаты за стрижку газона авансом или после завершения работы: оба метода имеют один и тот же конечный результат для вашего бумажника, но отличаются тем, в какой момент деньги в него добавляются. В листинге ниже демонстрируется разница на примере операции инкремента.

```

#include <iostream>

int main()

```

```

{
using std::cout;

int a = 20;

int b = 20;

cout << "a = " << a << ": b = " << b << "\n";

cout << "a++ = " << a++ << " : ++b = " << ++b << "\n";

cout << "a = " << a << ": b = " << b << "\n";

return 0;

}

```

Результат выполнения этой программы показан ниже:

a=20: b = 20

a++ = 20: ++b = 21

a=21: b = 21

Грубо говоря, нотация `a++` означает "использовать текущее значение `a` при вычислении выражения, затем увеличить `a` на единицу". Аналогично, нотация `++a` означает "сначала увеличить значение `a` на единицу, затем использовать новое значение при вычислении выражения". Например, мы имеем следующие отношения:

```

int x = 5;

int y = ++x; // изменить x, затем присвоить его y

// y равно 6, x равно 6

int z = 5;

int y = z++; // присвоить y, затем изменить z

//y равно 5, z равно 6

```

Операции инкремента и декремента представляют собой простой удобный способ решения часто возникающей задачи увеличения или уменьшения значений на единицу. Операции инкремента и декремента — симпатичные и компактные, но не стоит поддаваться соблазну и применять их к

одному и тому же значению более одного раза в одном и том же операторе. Проблема в том, что при этом правила "использовать и изменить" и "изменить и использовать" становятся неоднозначными. То есть, следующий оператор в различных системах может дать совершенно разные результаты:

$x = 2 * x++ * (3 - ++x)$; //не поступайте так

В C++ поведение операторов подобного рода не определено.

Домашнее задание № 2

Задача «Конус». Вычислить объем и полную поверхность усеченного конуса:

$$V = 1/3 \pi h(R^2 + Rr + r^2), S = \pi(R^2 + (R+r)l + r^2).$$

Задача «Разветвление». Для произвольных x и a вычислить

$$w = \begin{cases} a \ln |x|, & |x| < 1, \\ \sqrt{a - x^2}, & |x| \geq 1. \end{cases}$$

Задача «Функция». Для произвольных x , y и b вычислить функцию $z = \ln(b - y) \sqrt{b - x}$.

Задача «Порядок». Распечатать 10 последовательных натуральных чисел в возрастающем порядке, начиная с произвольного числа N .

Задача «Табуляция». Протабулировать функцию

$$y = \frac{x^2 - 2x + 2}{x - 1} \text{ при изменении } x \text{ от } -4 \text{ до } +4 \text{ с шагом } 0.5.$$

7. МАССИВЫ В C++

Массив — это структура данных, которая содержит множество значений, относящихся к одному и тому же типу. Например, массив может содержать 60 значений типа `int`, которые представляют информацию об объемах продаж за 5 лет, 12 значений типа `short`, представляющих количество дней в каждом месяце, или 365 значений типа `float`, которые указывают ежедневные расходы на питание в течение года. Каждое значение сохраняется в отдельном элементе массива, и компьютер хранит все элементы массива в памяти последовательно — друг за другом.

Для создания массива используется оператор объявления. Объявление массива должно описывать три аспекта:

- тип значений каждого элемента;
- имя массива;

- количество элементов в массиве.

В C++ это достигается модификацией объявления простой переменной, к которому добавляются квадратные скобки, содержащие внутри количество элементов. Например, следующее объявление создает массив по имени `months`, имеющий 12 элементов, каждый из которых может хранить одно значение типа `short`:

```
short months[12]; // создает массив из 12 элементов типа short
```

В сущности, каждый элемент — это переменная, которую можно трактовать как простую переменную.

Так выглядит общая форма объявления массива:

```
имя Типа имяМассива[размерМассива];
```

Выражение `размерМассива`, представляющее количество элементов, должно быть целочисленной константой, такой как 10, значением `const` либо константным выражением вроде `8 * sizeof(int)`, в котором все значения известны на момент компиляции. В частности, `размер Массива` не может быть переменной, значение которой устанавливается во время выполнения программы.

Большая часть пользы от массивов определяется тем фактом, что к его элементам можно обращаться индивидуально. Способ, который позволяет это делать, заключается в использовании индекса для нумерации элементов. Нумерация массивов в C++ начинается с нуля. (Это является обязательным — вы должны начинать с нуля. Это особенно важно запомнить программистам, ранее работавшим на языках Pascal и BASIC.) Для указания элемента массива в C++ используется обозначение с квадратными скобками и индексом между ними. Например, `months [0]` — это первый элемент массива `months`, а `months [11]` — его последний элемент. Обратите внимание, что индекс последнего элемента на единицу меньше, чем размер массива (рис. 4.1). Таким образом, объявление массива позволяет создавать множество переменных в одном объявлении, и вы затем можете использовать индекс для идентификации и доступа к индивидуальным элементам.

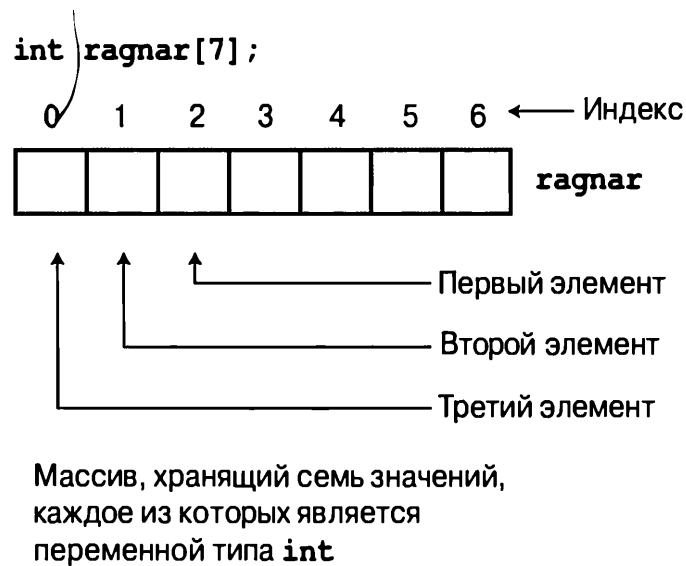


Рис. 4.1 Создание массива

Небольшая программа анализа, представленная в листинге ниже, демонстрирует несколько свойств массивов, включая их объявление, присваивание значение его элементам, а также инициализацию.

```
#include <iostream>

int main ()
{
    using namespace std;

    int yams[3] // создание массива из трех элементов
    yams[0] = 7 // присваивание значения первому элементу
    yams[1] = 8
    yams[2] = 6

    int yamcosts[3] = {20, 30, 5}; // создание и инициализация массива

    // Примечание. Если ваш компилятор C++ не может
    инициализировать

    // этот массив, используйте static int yamcosts[3] вместо int
    yamcosts[3]

    cout << "Total yams = ";

    cout << yams[0] + yams[1] + yams [2] << endl';
```

```

cout << "The package with " << yams[1] << " yams costs ";
cout << yamcosts[1] << " cents per yam.\n" ;
int total = yams[0] * yamcosts[0] + yams[1] * yamcosts [1];
total = total + yams[2] * yamcosts[2];
cout << "The total yam expense is " << total << " cents. \n";
cout << "\nSize of yams array = " << sizeof yams;
cout << " bytes. \n";
cout << "Size of one element = " << sizeof yams[0];
cout<< " bytes. \n";
return 0;
}

```

Ниже показан вывод программы из листинга:

Total yams = 21

The package with 8 yams costs 30 cents per yam.

The total yam expense is 410 cents.

Size of yams array = 12 bytes.

Size of one element = 4 bytes.

Лекция 9

7.1. Правила инициализации массивов

В C++ существует несколько правил, касающихся инициализации массивов. Они ограничивают, когда вы можете ее осуществлять, и определяют, что случится, если количество элементов массива не соответствует количеству элементов инициализатора. Давайте рассмотрим эти правила.

Вы можете использовать инициализацию только при объявлении массива. Ее нельзя выполнить позже, и нельзя присваивать один массив другому:

```
intcards [4] = {3, 6, 8, 10}; // все в порядке
```

```
inthand[4]; // все в порядке
```

```
hand[4] = {5, 6, 7, 9}; //не допускается
```

```
hand = cards; // не допускается
```

Однако можно использовать индексы и присваивать значения элементам массива индивидуально.

При инициализации массива можно указывать меньше значений, чем в массиве объявлено элементов. Например, следующий оператор инициализирует только первые два элемента массива `hotelTips`:

```
floathotelTips[5] = {5.0, 2.5};
```

Если вы инициализируете массив частично, то компилятор присваивает остальным элементам нулевые значения. Это значит, что инициализировать весь массив нулями очень легко — для этого просто нужно явно инициализировать нулем его первый элемент, а инициализацию остальных элементов поручить компилятору:

```
longtotals[500] = {0};
```

Следует отметить, что в случае инициализации массива с применением `{1}` вместо `{0}` только первый элемент будет установлен в 1; остальные по-прежнему получат значение 0.

Если при инициализации массива оставить квадратные скобки пустыми, то компилятор C++ самостоятельно пересчитает элементы. Предположим, например, что есть следующее объявление:

```
shortthings [] = {1, 5, 3, 8};
```

Компилятор сделает `things` массивом из пяти элементов.

8. СТРОКИ

Строка — это серия символов, сохраненная в расположенных последовательно байтах памяти. В C++ доступны два способа работы со

строками. Первый, унаследованный от C и часто называемый строками в стиле C, рассматривается в настоящей главе сначала. Позже будет описан альтернативный способ, основанный на библиотечном классе `string`.

Идея серии символов, сохраняемых в последовательных байтах, предполагает хранение строки в массиве `char`, где каждый элемент содержится в отдельном элементе массива. Строки предоставляют удобный способ хранения текстовой информации, такой как сообщения для пользователя или его ответы. Строки в стиле C обладают специальной характеристикой: последним в каждой такой строке является нулевой символ. Этот символ, записываемый как `\0`, представляет собой символ с ASCII-кодом 0, который служит меткой конца строки. Например, рассмотрим два следующих объявления:

```
char dog [8] = { 'b', 'e', 'a', 'u', 'x1', ' ', 'T', 'T' }; // это не строка
```

```
char cat [8] = { 'f', 'a', ' ', 't', 'e', 's', 's', 'a', '\0' }; // а это – строка
```

Обе эти переменные представляют собой массивы `char`, но только вторая из них является строкой. Нулевой символ играет фундаментальную роль в строках стиля C. Например, в C++ имеется множество функций для обработки строк, включая те, что используются `cout`. Все они обрабатывают строки символ за символом до тех пор, пока не встретится нулевой символ. Если вы просите объект `cout` отобразить такую строку, как `cat` из предыдущего примера, он выводит первых семь символов, обнаруживает нулевой символ и на этом останавливается. Однако если вы вдруг решите вывести в `cout` массив `dog` из предыдущего примера, который не является строкой, то `cout` напечатает восемь символов из этого массива и будет продолжать двигаться по памяти, байт за байтом, интерпретируя каждый из них как символ, подлежащий выводу, пока не встретит нулевой символ. Поскольку нулевые символы, которые, по сути, представляют собой байты, содержащие нули, встречаются в памяти довольно часто, ошибка обычно обнаруживается быстро, но в любом случае вы не должны трактовать нестроковые символьные массивы как строки.

Пример инициализации массива `cat` выглядит довольно громоздким и утомительным — множество одиночных кавычек плюс необходимость помнить о нулевом символе. Не волнуйтесь. Существует более простой способ инициализации массива с помощью строки. Для этого просто используйте строку в двойных кавычках, которая называется строковой константой или строковым литералом, как показано ниже:

```
char bird[11] = "Mr. Cheeps"; // наличие символа \0 подразумевается
char fish[] = "Bubbles"; // позволяет компилятору подсчитать
// количество элементов
```

Строки в двойных кавычках всегда неявно включают ограничивающий нулевой символ, поэтому указывать его явно не требуется (рис. 4.2.) К тому же разнообразные средства ввода C++, предназначенные для чтения строки с клавиатурного ввода в массив `char`, автоматически добавляют завершающий нулевой символ. (Если при компиляции программы из листинга 4.1 вы обнаружите необходимость в использовании ключевого слова `static` для инициализации массива, это также понадобится сделать с показанными выше массивами `char`.)

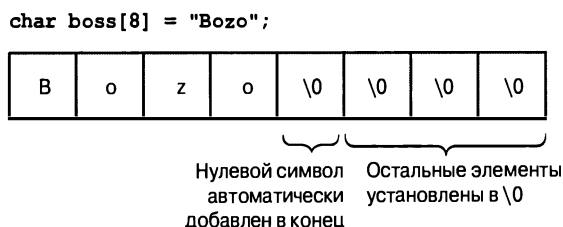


Рис. 4.2 Инициализация массива строки

Разумеется, вы должны обеспечить достаточный размер массива, чтобы в него поместились все символы строки, включая нулевой. Инициализация символьного массива строковой константой — это один из тех случаев, когда безопаснее поручить компилятору подсчет количества элементов в массиве. Если сделать массив больше строки, никаких проблем не возникнет — только непроизводительный расход пространства. Причина в том, что функции, которые работают со строками, руководствуются позицией нулевого символа, а не размером массива. В C++ не накладывается никаких ограничений на длину строки.

8.1. Построчное чтение ввода

Чтение строкового ввода по одному слову за раз — часто не является желательным поведением. Например, предположим, что программа запрашивает у пользователя ввод города, и пользователь отвечает вводом New York или Sao Paulo. Вы бы хотели, чтобы программа прочитала и сохранила полные названия, а не только New и Sao. Чтобы иметь возможность вводить целые фразы вместо отдельных слов, необходим другой подход к строковому

вводу. Точнее говоря, нужен метод, ориентированный на строки, вместо метода, ориентированного на слова. К счастью, у класса `istream`, экземпляром которого является `cin`, есть функции-члены, предназначенные для строчно-ориентированного ввода: `getline()` и `get()`. Оба читают полную строку ввода — т.е. вплоть до символа новой строки. Однако `getline()` затем отбрасывает символ новой строки, в то время как `get()` оставляет его во входной очереди. Давайте рассмотрим их детально, начиная с `getline()`.

8.2. Сточно-ориентированный ввод с помощью `getline()`

Функция `getline()` читает целую строку, используя символ новой строки, который передан клавишей `<Enter>`, для обозначения конца ввода. Этот метод иницируется вызовом функции `cin.getline()`. Функция принимает два аргумента. Первый аргумент — это имя места назначения (т.е. массива, который сохраняет введенную строку), а второй — максимальное количество символов, подлежащих чтению. Если, скажем, установлен предел 20, то функция читает не более 19 символов, оставляя место для автоматически добавляемого в конец нулевого символа. Функция-член `getline()` прекращает чтение, когда достигает указанного предела количества символов или когда читает символ новой строки — смотря, что произойдет раньше.

Например, предположим, что вы хотите воспользоваться `getline()` для чтения имени в 20-элементный массив `name`. Для этого следует указать такой вызов:

```
cin.getline(name,20);
```

Он читает полную строку в массив `name`, предполагая, что строка состоит не более чем из 19 символов.

8.3. Сточно-ориентированный ввод с помощью `get()`

Теперь попробуем другой подход. Класс `istream` имеет функцию-член `get()`, которая доступна в различных вариантах. Один из них работает почти так же, как `getline()`. Он принимает те же аргументы, интерпретирует их аналогичным образом, и читает до конца строки. Но вместо того, чтобы прочитать и отбросить символ новой строки, `get()` оставляет его во входной очереди. Предположим, что используются два вызова `get()` подряд:

```
cin.get(name, ArSize);
```

```
cin.get(dessert, ArSize); // проблема
```


Поскольку первый вызов оставляет символ новой строки во входной очереди, получается, что символ новой строки оказывается первым символом, который видит следующий вызов. Таким образом, второй вызов `get()` заключает, что он достиг конца строки, не найдя ничего интересного, что можно было бы прочитать. Без посторонней помощи `get()` вообще не может преодолеть этот символ новой строки.

К счастью, на помощь приходят различные варианты `get()`. Вызов `cin.get()` без аргументов читает одиночный следующий символ, даже если им будет символ новой строки, поэтому вы можете использовать его для того, чтобы отбросить символ новой строки и подготовиться к вводу следующей строки. То есть следующая последовательность будет работать правильно:

```
cin.get(name, ArSize); // чтение первой строки
```

```
cin.get(); // чтение символа новой строки
```

```
cin.get(dessert, ArSize) // чтение второй строки
```

Другой способ применения `get ()` состоит в конкатенации, или соединении, двух вызовов функций-членов класса, как показано в следующем примере:

```
cin.get(name, ArSize).get(); // конкатенация функций-членов
```

Такую возможность обеспечивает то, что `cin.get (name, ArSize)` возвращает объект `cin`, который затем используется в качестве объекта, вызывающего функцию `get()`. Аналогично приведенный ниже оператор читает две следующих друг за другом строки в массивы `name1` и `name2`, что эквивалентно двум отдельным вызовам `cin.getline ()`:

```
cin.getline(name1, ArSize).getline(name2, ArSize);
```

9. ВВЕДЕНИЕ В КЛАСС STRING

В стандарте ISO /ANSI C++98 библиотека C++ была расширена за счет добавления класса `string`. Поэтому отныне вместо использования символьных массивов для хранения строк можно применять переменные типа `string` (или, пользуясь терминологией C++, объекты). Как вы увидите, класс `string` проще в использовании, чем массив, и к тому же предлагает более естественное представление строки как типа.

Для работы с классом `string` в программе должен быть включен заголовочный файл `string`. Класс `string` является частью пространства имен `std`, поэтому вы должны указать директиву `using` или объявление либо же ссылаться на класс как `std::string`. Определение класса скрывает природу строки как массива символов и позволяет трактовать ее как обычную переменную. В листинге проиллюстрированы некоторые сходства и различия между объектами `string` и символьными массивами.

```
#include <iostream>

#include <string> // обеспечение доступа к классу string

int main()
{
    using namespace std;

    char charr1[20]; // создание пустого массива
    char charr2[20] = "jaguar"; // создание инициализированного массива
    string str1; // создание пустого объекта строки
    string str2 = "panther"; // создание инициализированного объекта строки

    cout << "Enter a kind of feline: ";

    // Введите животное из семейства кошачьих
    cin >> charr1;

    cout << "Enter another kind of feline: ";

    // Введите другое животное из семейства кошачьих
    cin >> str1; // использование cin для ввода

    cout << "Here are some felines:\n";

    cout << charr1 << " " << charr2 << " "
    << str1 << " " << str2 // использование cout для вывода
    << endl;

    cout << "The third letter in " << charr2 << " is "
```

```
<< charr2[2] « endl;  
cout << "The third letter in " << str2 << " is "  
<< str2[2] << endl; // использование нотации массивов  
return 0;
```

Ниже показан пример выполнения программы из листинга:

Enter a kind of feline: ocelot

Enter another kind of feline: tiger

Here are some felines:

ocelot jaguar tiger panther

The third letter in jaguar is g

The third letter in panther is n

Из этого примера вы должны сделать вывод, что во многих отношениях объект `string` можно использовать так же, как символьный массив.

- Объект `string` можно инициализировать строкой в стиле C.
- Чтобы сохранить клавиатурный ввод в объекте `string`, можно использовать `cin`.
- Для отображения объекта `string` можно применять `cout`.
- Можно использовать нотацию массивов для доступа к индивидуальным символам, хранящимся в объекте `string`.

Главное отличие между объектами `string` и символьными массивами, продемонстрированное в листинге, заключается в том, что объект `string` объявляется как обычная переменная, а не массив:

```
stringstr1; // создание пустого объекта строки  
stringstr2 = "panther"; // создание инициализированного объекта строки
```

Проектное решение, положенное в основу класса, позволяет программе автоматически обрабатывать изменение размера строк. Например, объявление `str1` создает объект `string` нулевой длины, но при чтении ввода в `str1` программа автоматически его увеличивает:

```
cin>>str1; // str1 увеличен для того, чтобы вместить ввод
```

Это делает использование объекта `string` более удобным и безопасным по сравнению с массивом. Концептуально важным является то, что массив строк — это коллекция единиц хранения отдельных символов, служащих для хранения строки, а класс `string` — единая сущность, представляющая строку.

Лекция 10

9.1. Присваивание, конкатенация и добавление

Некоторые операции со строками класс `string` выполняет проще, чем это возможно в случае символьных массивов. Например, просто присвоить один массив другому нельзя. Однако один объект `string` вполне можно присвоить другому:

```
char charr1[20]; // создание пустого массива
```

```
char charr2[20] = "jaguar"; // создание инициализированного массива
```

```
string str1; // создание пустого объекта string
```

```
string str2 = "panther"; // создание инициализированной строки
```

```
charr1 = charr2; // НЕ ПРАВИЛЬНО, присваивание массивов не разрешено
```

```
str1 = str2; // ПРАВИЛЬНО, присваивание объектов допускается
```

Класс `string` упрощает комбинирование строк. С помощью операции `+` можно сложить два объекта `string` вместе, а посредством операции `+=` можно добавить строку к существующему объекту `string`. В отношении предшествующего кода у нас есть следующие возможности:

```
string str3;
```

```
str3 = str1 + str2; // присвоить str3 объединение строк
```

```
str1 += str2; // добавить str2 в конец str1
```

В листинге приведен соответствующий пример. Обратите внимание, что складывать и добавлять к объектам `string` можно как другие объекты `string`, так и строки в стиле `C`.

```
#include <iostream>

#include<string> // обеспечениедоступакклассуstring

int main()

{

using namespace std;

string s1 = "penguin";

strings2, s3;

// Присваиваниеодногообъектаstringдругому

cout << "You can assign one string object to another: s2 = s1\n";

s2 = s1;

cout << "s1: " << s1 << ", s2: " << s2 << endl;

// Присваивание строки в стиле C объекту string

cout << "You can assign a C-style string to a string object.\n";

cout << "s2 = \"buzzard\"\n";

s2 = "buzzard";

cout << "s2: " << s2 << endl;

// Конкатенациястрок

cout << "You can concatenate strings: s3 = s1 + s2\n";

s3 = s1 + s2;

cout << "s3: " << s3 << endl;

// Добавлениестроки

cout << "You can append strings. \n";

s1 += s2;
```

```

cout <<"s1 += s2 yields s1 = " << s1 << endl;

s2 += " for a day";

cout <<"s2 += \" for a day\" yields s2 = " << s2 << endl;

return 0;

}

```

Ниже показан вывод программы из листинга:

You can assign one string object to another: s2 = si

S1: penguin, s2: penguin

You can assign a C-style string to a string object.

s2 = "buzzard"

s2: buzzard

You can concatenate strings: s3 = si + s2

s3: penguinbuzzard

You can append string's.

S1 += s2 yields s1 = penguinbuzzard

s2 += " for a day" yields s2 = buzzard for a day

9.2. Дополнительные сведения об операциях класса string

Еще до появления в C++ класса string программисты нуждались в таких действиях, как присваивание строк. Для строк в стиле C использовались функции из стандартной библиотеки C. Эти функции поддерживаются заголовочным файлом cstring (бывший string.h). Например, вы можете применять функцию strcpy() для копирования строки в символьный массив, а функцию strcat () — для добавления строки к символьному массиву:

```
strcpy(charr1, charr2); // копировать charr2 в charr1
```

```
strcat(charr1, charr2); // добавить содержимое charr2 к char1
```

В листинге сравниваются подходы с объектами string и символьными массивами.

```

#include <iostream>

#include <string> // обеспечение доступа к классу string
#include <cstring> // библиотека обработки строк в стиле C

int main ()
{
    using namespace std;

    char charr1[20];
    char charr2[20] = "jaguar";
    string str1;
    string str2 = "panther";

    // Присваивание объектов string и символьных массивов
    str1 = str2; // копирование str2 в str1
    strcpy(charr1, charr2); // копирование charr2 в charr1

    // Добавление объектов string и символьных массивов
    str1 += " paste"; // добавление " paste" в конец str1
    strcat(charr1, " juice"); // добавление " juice" в конец charr1

    // Определение длины объекта string и строки в стиле C
    int len1 = str1.size(); // получение длины str1
    int len2 = strlen(charr1); // получение длины charr1

    cout << "The string " << str1 << " contains "
    << len1 << " characters. \n";

    cout << "The string " << charr1 << " contains "
    << len2 << " characters. \n";

    return 0;
}

```

Ниже представлен вывод программы из листинга:

The string panther paste contains 13 characters.

The string jaguar juice contains 12 characters.

Синтаксис работы с объектами `string` выглядит проще, чем использование строковых функций C. Это особенно проявляется при более сложных операциях.

Например, эквивалент из библиотеки C следующего оператора:

```
str3 = str1 + str2;
```

будет таким:

```
strcpy(charr3, charr1);
```

```
strcat(charr3, charr2);
```

Более того, при работе с массивами всегда существует опасность, что целевой массив окажется слишком малым для того, чтобы вместить всю информацию.

Например:

```
char site[10] = "house";
```

```
strcat(site, " of pancakes"); // проблема с нехваткой памяти
```

Функция `strcat()` пытается скопировать все 12 символов в массив `site`, таким образом, переполняя выделенную память. Это может вызвать аварийное завершение программы, или же программа продолжит работать, но с поврежденными данными.

Класс `string`, с его автоматическим расширением при необходимости, позволяет избегать проблем подобного рода. Библиотека C предлагает функции, подобные `strcat()` и `strcpy()`, которые называются `strncat()` и `strncpy()`. Эти функции работают более безопасно, принимая третий параметр, который задает максимально допустимый размер целевого массива, но их применение усложняет написание программ.

Обратите внимание, что для получения количества символов в строке используется разный синтаксис:

```
int len1 = str1.size(); // получение длины str1
```



```
int len2 = strlen(charrl); // получение длины charrl
```

`strlen()` — это стандартная функция, которая принимает в качестве аргумента строку в стиле C и возвращает количество символов в ней. Функция `size()` обычно делает то же самое, но синтаксис ее вызова отличается. Вместо передачи аргумента ее имени предшествует имя объекта `strl`, отделенное точкой. Как вы уже видели на примере метода `put()` в главе 3, этот синтаксис означает, что `strl` — это объект, а `size()` — метод класса. Метод — это функция, которая может быть вызвана только объектом, принадлежащим классу, в котором определен данный метод. В данном конкретном случае `strl` — объект `string`, а `size()` — метод класса `string`. Короче говоря, функции C используют аргументы для идентификации требуемой строки, а объект класса C++ `string` использует имя объекта и операцию точки для указания того, какую именно строку нужно взять.

9.3. Дополнительные сведения о вводе-выводе класса `string`

Как вы уже видели, можно использовать `cin` с операцией `>>` для чтения объекта `string` и `cout` с операцией `<<` — для отображения объекта `string`, причем с тем же синтаксисом, что и в случае строк в стиле C. Однако чтение за один раз целой строки с пробелами вместо отдельного слова требует другого синтаксиса. В листинге демонстрируется это отличие.

```
#include <iostream>

#include <string>

#include <cstring>

int main ()
{
    using namespace std;

    char charr[20];

    string str;

    // Длина строки в charr перед вводом

    cout << "Length of string in charr before input: "
    << strlen(charr) << endl;
```

```

// Длина строки в str перед вводом
cout << "Length of string in str before input: "
« str.size () « endl;

cout « "Enter a line of text:\n"; // ввод строки текста
cin.getline(charr, 20); // указание максимальной длины
cout « "You entered: " « charr « endl;

cout « "Enter another line of text:\n"; // ввод другой строки текста
getline(cin, str); // теперь cin - аргумент; спецификатор длины отсутствует
cout « "You entered: " << str << endl;

// Длина строки в charr после ввода
cout « "Length of string in charr after input: "
<< strlen(charr) « endl;

// Длина строки в str после ввода
cout << "Length of string in str after input: "
<< str.size () << endl;

return 0;

}

```

Ниже показан пример выполнения программы из листинга:

Length of string in charr before input: 27

Length of string in str before input: 0

Enter a line of text:

peanut butter

You entered: peanut butter

Enter another line of text:

blueberry jam

You entered: blueberry jam

Length of string in charr after input: 13

Length of string in str after input: 13

Обратите внимание, что программа сообщает длину строки в массиве `charr` перед вводом как равную 27, т.е. больше, чем размер массива! Здесь происходят две вещи. Первая — содержимое неинициализированного массива не определено. Вторая — функция `strlen()` работает, просматривая массив, начиная с первого элемента, и подсчитывает количество байт до тех пор, пока не встретит нулевой символ. В этом случае первый нулевой символ встретился через несколько байт за пределами массива. Где именно встретится нулевой символ в неинициализированном массиве, определяется случаем, поэтому весьма вероятно, что при запуске этой программы вы получите другое значение.

Также отметьте, что длина строки `str` перед вводом равна 0. Это объясняется тем, что размер неинициализированного объекта `string` автоматически устанавливается в 0.

Следующий код читает строку в массив:

```
cin.getline(charr, 20);
```

Точечная нотация указывает на то, что функция `get line ()` — это метод класса `istream`. (Вспомните, что `cin` является объектом класса `istream`.) Как упоминалось ранее, первый аргумент задает целевой массив, а второй — его размер, используемый `getline()` для того, чтобы избежать переполнения массива.

Следующий код читает строку в объект `string`:

```
getline (cin,str);
```

Здесь точечная нотация не используется, а это говорит о том, что данная функция `getline ()` не является методом класса. Поэтому она принимает объект `cin` как аргумент, сообщающий о том, где искать ввод. К тому же нет аргумента, задающего размер строки, потому что объект `string` автоматически изменяет свой размер, чтобы вместить строку.

Так почему же одна функция `getline()` — метод класса `istream`, а вторая — нет? Класс `istream` появился в C++ до того, как был добавлен класс `string`. Поэтому `istream` распознает базовые типы C++, такие как `double` или `int`, но

ничего не знает о типе `string`. Таким образом, класс `istream` имеет методы, обрабатывающие `double`, `int` и другие базовые типы, но не имеет методов, обрабатывающих объекты `string`.

Поскольку у класса `istream` нет методов, обрабатывающих объекты `string`, вас может удивить, почему работает следующий код:

```
cin >> str; // чтение слова в объект string по имени str
```

Оказывается, что следующий код использует (в скрытом виде) функцию-член класса `istream`:

```
cin >> x; // чтение значения в переменную базового типа C++
```

Но эквивалент этого кода с классом `string` использует дружественную функцию (также в скрытом виде) класса `string`. Между тем можете смело использовать `cin` и `cout` с объектами `string`, не заботясь о его внутреннем функционировании.

Домашнее задание № 3

Задача «Заем». Месячная выплата m по займу в S рублей на n лет под процент r вычисляется по формуле:

$$m = \frac{Sr(1+r)^n}{12((1+r)^n - 1)}, \quad \text{где} \quad r = \frac{p}{100}.$$

Дано: S , p , n . Найти: m .

Задача «Ссуда». Под какой процент r выдана ссуда величиной S рублей, которая гасится месячными выплатами величиной m в течение n лет.

Задача «Копирование файла». Создать на диске текстовый файл и скопировать его на экран.

Задача «Фильтр». Вывести на экран только числа из созданного Вами на диске текстового файла, содержащего буквы и числа.

Задача «Сортировка букв». Задать строку из 30 букв и расставить их в алфавитном порядке.

Лекция 11

10. ПЕРЕЧИСЛЕНИЯ

Средство C++ `enum` представляет собой альтернативный по отношению к `const` способ создания символических констант. Он также позволяет

определять новые типы, но в очень ограниченной манере. Синтаксис `enum` подобен синтаксису структур. Например, рассмотрим следующий оператор:

```
enumspectrum {red, orange, yellow, green, blue, violet, indigo, ultraviolet};
```

Этот оператор делает две вещи.

- Объявляет имя нового типа — `spectrum`; при этом `spectrum` называется перечислением, почти так же, как переменная `struct` называется структурой.
- Устанавливает `red`, `orange`, `yellow` и т.д. в качестве символических констант для целочисленных значений 0-7. Эти константы называются перечислителями.

По умолчанию перечислителям присваиваются целочисленные значения, начиная с 0 для первого из них, 1 — для второго и т.д. Это правило по умолчанию можно переопределить, явно присваивая целочисленные значения. Чуть позже вы увидите, как это делается.

Имя перечисления можно использовать для объявления переменной с этим типом перечисления:

```
spectrum band; // band — переменная типа spectrum
```

Переменные типа перечислений имеют ряд специальных свойств, которые мы сейчас рассмотрим.

Единственными допустимыми значениями, которые можно присвоить переменной типа перечисления без необходимости приведения типов, являются значения, указанные в определении этого перечисления. Рассмотрим пример:

```
band = blue; // правильно, blue - перечислитель
```

```
band = 2000; // неправильно, 2000 — не перечислитель
```

Таким образом, переменная `spectrum` ограничена только восемью допустимыми значениями. Некоторые компиляторы выдают ошибку, если вы пытаетесь присвоить некорректное значение, в то время как другие выдают только предупреждения.

Для максимальной переносимости вы должны трактовать присваивание переменным типа `enum` значений, не входящих в определение `enum`, как ошибку.

Для перечислений определена только операция присваивания. В частности, арифметические операции не предусмотрены:

```
band = orange; // правильно
```

```
++band; // неправильно (операция ++ обсуждается в главе 5)
```

```
band = orange + red; // неправильно, но довольно хитро
```

Однако некоторые реализации не накладывают таких ограничений. Это позволяет нарушить ограничения типа. Например, если `band` равно `ultraviolet`, или 7, а затем выполняется `++band`, и если такое разрешено компилятором, то `band` получит значение, недопустимое для типа `spectrum`. Опять-таки, для достижения максимальной переносимости вы должны придерживаться ограничений.

Перечисления — целочисленные типы, и они могут быть представлены в виде `int`, однако тип `int` не преобразуется автоматически в тип перечисления:

```
int color = blue; // правильно, тип spectrum приводится к int
```

```
band = 3; // неправильно, int не преобразуется в spectrum
```

```
color = 3 + red; // правильно, red преобразуется в int
```

Обратите внимание, что хотя значение 3 в этом примере соответствует перечислителю `green`, все же присваивание 3 переменной `band` вызывает ошибку несоответствия типа. Но присваивание `green` переменной `band` законно, потому что оба они имеют тип `spectrum`. И снова, некоторые реализации не накладывают такого ограничения. В выражении `3 + red` сложение не определено для перечислений. Однако `red` преобразуется в тип `int`, в результате чего получается значение типа `int`. Благодаря преобразованию перечисления в `int` в данной ситуации, вы можете использовать перечислители в арифметических выражениях, комбинируя их с обычными целыми, даже несмотря на то, что такая арифметика не определена для самих перечислителей.

Предыдущий пример

```
band = orange + red; // неправильно, но довольно хитро
```

не работает по другой причине. Да, действительно, операция `+` не определена для перечислителей. Но также верно и то, что перечислители преобразуются в целые числа, когда применяются в арифметических выражениях, поэтому выражение `orange + red` превращается в `1 + 0`, что вполне

корректно. Но это выражение имеет тип `int`, поэтому оно не может быть присвоено переменной `band` типа `spectrum`.

Вы можете присвоить значение `int` переменной `enum`, если полученное значение допустимо и применяется явное приведение типа:

```
band = spectrum(3); // приведение 3 к типу spectrum
```

Но что будет, если вы попытаетесь выполнить приведение типа для недопустимого значения? Результат не определен, в том смысле, что попытка не будет воспринята как ошибочная, но вы не можете полагаться на полученное в результате значение:

```
band = spectrum(40003); // результат не определен
```

(Обсуждение того, какие значения являются приемлемыми, а какие неприемлемыми, ищите в разделе "Диапазоны значений перечислителей" далее в этой главе.)

Как видите, правила, которым подчиняются перечисления, достаточно строги. На практике перечисления чаще используются как способ определения взаимосвязанных символических констант, нежели как средство определения новых типов. Например, вы можете применять перечисления для определения символических констант для операторов `switch`. (Примеры ищите в главе 6.) Если вы собираетесь только использовать константы и не создавать переменные перечислимого типа, то в этом случае можете опустить имя перечислимого типа, как показано ниже:

```
enum {red, orange, yellow, green, blue, violet, indigo, ultraviolet};
```

Конкретные значения элементов перечислений можно устанавливать явно посредством операция присваивания:

```
enum bits {one = 1, two = 2, four = 4, eight = 8};
```

Присваиваемые значения должны быть целочисленными. Можно также явно устанавливать только некоторые из перечислителей:

```
enum bigstep{first, second = 100, third};
```

В этом случае `first` получает значение 0 по умолчанию. Каждый последующий неинициализированный перечислитель увеличивается на единицу по сравнению с предыдущим. Поэтому `third` имеет значение 101.

И, наконец, допускается указывать одно и тоже значение для нескольких перечислителей:

```
enum {zero, null = 0, one, numero_uno = 1};
```

Здесь `zero` и `null` имеют значение 0, `aone` и `numero_uno` — значение 1. В ранних версиях C++ элементам перечислений можно было присваивать только значения типа `int` (или неявно преобразуемые к `int`), но теперь это ограничение снято, и также можно использовать значения типа `long` или даже `longlong`.

Изначально правильными значениями перечислений являются лишь те, что названы в объявлении. Однако C++ расширяет список допустимых значений, которые могут быть присвоены перечислимому переменным, за счет использования приведения типа. Каждое перечисление имеет диапазон и с помощью приведения к типу переменной перечисления можно присвоить любое целочисленное значение в пределах этого диапазона, даже если данное значение не равно ни одному из перечислителей.

Например, предположим, что `bits` и `my flag` определены следующим образом:

```
enum bits {one = 1, two = 2, four = 4, eight = 8};
```

```
bits myflag;
```

В таком случае показанный ниже оператор является допустимым:

```
myflag = bits (6); // правильно, потому что 6 находится в пределах диапазона
```

Здесь 6 не является значением ни одного из перечислителей, однако находится в диапазоне этого определенного перечисления.

Диапазон определяется следующим образом. Для нахождения верхнего предела выбирается перечислитель с максимальным значением. Затем ищется наименьшее число, являющееся степенью двойки, которое больше этого максимального значения, и из него вычитается единица. (Например, максимальное значение `bigstep`, как определено выше, равно 101. Минимальное число, представляющее степень двойки, которое больше 101, равно 128, поэтому верхним пределом диапазона будет 127.) Для нахождения минимального предела выбирается минимальное значение перечислителя. Если оно равно 0 или больше, то нижним пределом диапазона будет 0. Если же минимальное значение перечислителя отрицательное, используется такой же

подход, как при вычислении верхнего предела, но со знаком минус. (Например, если минимальный перечислитель равен -6, то следующей степенью двойки будет -8, и нижний предел получается равным -7.)

Идея состоит в том, чтобы компилятор мог выяснить, сколько места необходимо для хранения перечисления. Он может использовать от 1 байт или менее для перечислений с небольшим диапазоном, и до 4 байт — для перечислений со значениями типа `long`.

11. УКАЗАТЕЛИ

Существует три основные свойства, которые должна отслеживать компьютерная программа, когда она сохраняет данные.

- где хранится информация;
- какое значение сохранено;
- разновидность сохраненной информации.

Пока что вы использовали только одну стратегию: объявление простых переменных. В операторе объявления предоставляется тип и символическое имя значения. Он также заставляет программу выделить память для этого значения и внутренне отслеживать ее местоположение.

Давайте рассмотрим другую стратегию, важность которой проявляется при разработке классов C++. Эта стратегия основана на указателях, которые представляют собой переменные, хранящие адреса значений вместо самих значений. Но прежде чем обратиться к указателям, давайте поговорим о том, как явно получить адрес обычной переменной. Для этого применяется операция взятия адреса, обозначаемая символом `&`, к переменной, адрес которой интересует. Например, если `home` — переменная, то `&home` — ее адрес. В листинге 4.14 демонстрируется использование этой операции.

```
#include <iostream>
```

```
int main()
```

```
{
```

```
using namespace std;
```

```
int donuts = 6;
```

```
double cups = 4.5;
```

```

cout << "donuts value = " << donuts;

cout << " and donuts address = " << Sdonuts << endl;

// ПРИМЕЧАНИЕ: может понадобиться использовать
// unsigned (Sdonuts) и unsigned (&cups)

cout << "cups value = " << cups;

cout << " and cups address = " << &cups << endl;

return 0;

}

```

Ниже показан вывод программы из листинга 4.14 в одной из систем:

```

donuts value = 6 and donuts address = 0x0065fd40

cups value = 4.5 and cups address = 0x0065fd44

```

В показанной здесь конкретной реализации `cout` используется шестнадцатеричная нотация при отображении значений адресов, т.к. это обычная нотация, применяемая для указания адресов памяти. (Некоторые реализации применяют десятичную нотацию.) Наша реализация сохраняет `donuts` в памяти с меньшими адресами, чем `cups`. Разница между этими двумя адресами составляет `0x0065fd44-0x0065fd40`, или 4 байта. Конечно, в разных системах вы получите разные значения этих адресов. К тому же некоторые системы могут сохранять `cups` перед `donuts`, и разница между адресами составит 8, потому что `cups` имеет тип `double`. Другие системы могут даже разместить эти переменные в памяти далеко друг от друга, а не рядом.

Таким образом, использование обычных переменных трактует значение как именованную величину, а ее местоположение — как производную величину. Теперь рассмотрим стратегию указателей, которая представляет важнейшую часть философии программирования C++ в части управления памятью.

Новая стратегия хранения данных изменяет трактовку местоположения как именованной величины, а значения — как производной величины. Для этого предусмотрен специальный тип переменной — указатель, который может хранить адрес значения. Таким образом, имя указателя представляет местоположение. Применяя операцию `*`, называемую косвенным значением или операцией разыменования, можно получить значение, хранящееся в

указанном месте. (Да, это тот же символ *, который применяется для обозначения арифметической операции умножения; C++ использует контекст для определения того, что подразумевается в каждом конкретном случае — умножение или разыменование.) Предположим, например, что `manly` — это указатель. В таком случае `manly` представляет адрес, а `*manly` — значение, находящееся по этому адресу. Комбинация `*manly` становится эквивалентом простой переменной типа `int`. Эти идеи демонстрируются в листинге 4.15. Также там показано, как объявляется указатель.

```
#include<iostream>

intmain()

{

usingnamespacestd;

intupdates =6; // объявлениепеременной

int * p_updates; // объявлениеуказателянаint

p_updates = Supdates; // присвоитьадресintуказателю

// Выразить значения двумя способами

cout << "Values: updates = " << updates;

cout << ", *p_updates = " << *p_updates << endl;

// Выразитьадресадвумяспособами

cout << "Addresses: Supdates = " << Supdates;

cout << ", p_updates = " << p_updates << endl;

// Изменить значение через указатель

*p_updates = *p_updates + 1;

cout << "Now updates = " << updates << endl;

return 0;

}
```

Ниже показан пример выполнения программы из листинга 4.15:

Values: updates = 6, *p_updates = 6

Addresses: &updates = 0x0065fd48, p_updates = 0x0065fd48

Nowupdates = 7

Как видите, переменная `updates` типа `int` и переменная-указатель `p_updates` — это две стороны одной монеты. Переменная `updates` в первую очередь представляет значение, а для получения его адреса используется операция `&`, в то время как `p_updates` представляет адрес, а для получения значения применяется операция `*`. Поскольку `p_updates` указывает на `updates`, конструкции `*p_updates` и `updates` полностью эквивалентны. Вы можете использовать `*p_updates` точно так же, как используете переменную типа `int`. Как показано в примере из листинга 4.15, можно даже присваивать значения `*p_updates`. Это изменяет значение указываемой переменной — `updates`.

11.1. Объявление и инициализация указателей

Давайте рассмотрим процесс объявления указателей. Компьютеру нужно отслеживать тип значения, на которое ссылается указатель. Например, адрес `char` обычно выглядит точно так же, как и адрес `double`, но `char` и `double` использует разное количество байт и разный внутренний формат представления значений. Поэтому объявление указателя должно задавать тип данных указываемого значения.

Например, предыдущий пример содержит следующее объявление:

```
int * p_updates;
```

Этот оператор устанавливает, что комбинация `*p_updates` имеет тип `int`.

Поскольку вы используете операцию `*`, применяя ее к указателю, сама переменная `p_updates` должна быть указателем. Мы говорим, что `p_updates` указывает на тип `int`. Мы также говорим, что тип `p_updates` — это указатель на `int`, или точнее, `int *`. Итак, повторим еще раз: `p_updates` — это указатель (адрес), а `*p_updates` — это `int`, а не указатель (рис. 11.1). К слову, пробелы вокруг операции `*` не обязательны. Традиционно программисты на C используют следующую форму:

```
int *ptr;
```

Это подчеркивает идею, что комбинация `*ptr` является значением типа `int`.

С другой стороны, многие программисты на C++ отдают предпочтение такой форме:

```
int* ptr;
```

Это подчеркивает идею о том, что `int*` — это тип "указатель на `int`". Для компилятора не важно, с какой стороны вы поместите пробел. Можно даже записать так:

```
int*ptr;
```

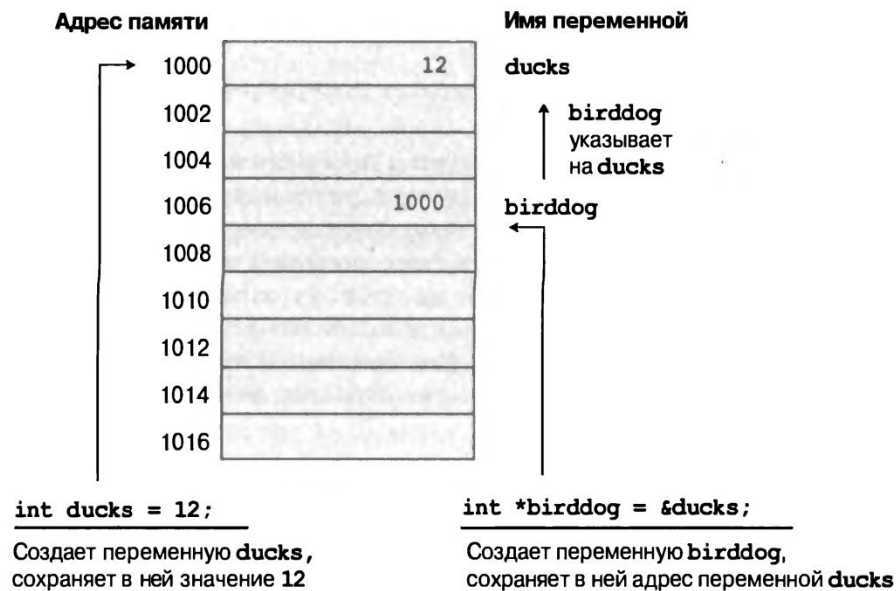


Рис. 11.1 Указатели хранят указатели

Однако учтите, что следующее объявление создает один указатель (`p1`) и одну обычную переменную типа `int` (`p2`):

```
int* p1, p2;
```

Знак `*` должен быть помещен возле каждой переменной типа указателя.

Лекция 12

11.2. Опасность, связанная с указателями

Опасность подстерегает тех, кто использует указатели неосмотрительно. Очень важно понять, что при создании указателя в коде C++ компьютер выделяет память для хранения адреса, но не выделяет памяти для хранения данных, на которые указывает этот адрес. Выделение места для данных требует

отдельного шага. Если пропустить этот шаг, как в следующем фрагменте, то это обеспечит прямой путь к проблемам:

```
long * fellow; // создать указатель на long
```

```
*fellow = 223323; // поместить значение в неизвестное место
```

Конечно, `fellow` — это указатель. Но на что он указывает? Никакого адреса переменной `fellow` в коде не присвоено. Так куда будет помещено значение 223323? Ответить на это невозможно. Поскольку переменная `fellow` не была инициализирована, она может иметь какое угодно значение. Что бы в ней ни содержалось, программа будет интерпретировать это как адрес, куда и поместит 223323. Если так случится, что `fellow` будет иметь значение 1200, компьютер попытается поместить данные по адресу 12 00, даже если этот адрес окажется в середине вашего программного кода. На что бы ни указывал `fellow`, скорее всего, это будет не то место, куда вы хотели бы поместить число 223323. Ошибки подобного рода порождают самое непредсказуемое поведение программы и такие ошибки очень трудно отследить.

11.3. Указатели и числа

Указатели — это не целочисленные типы, даже несмотря на то, что компьютеры обычно выражают адреса целыми числами. Концептуально указатели представляют собой типы, отличные от целочисленных. Целочисленные значения можно суммировать, вычитать, умножать, делить и т.д. Но указатели описывают местоположение, и не имеет смысла, например, перемножать между собой два местоположения. В терминах допустимых над ними операций указатели и целочисленные типы отличаются друг от друга. Следовательно, нельзя просто присвоить целочисленное значение указателю:

```
int * pt;
```

```
pt = 0xB8000000; // несоответствие типов
```

Здесь в левой части находится указатель на `int`, поэтому ему можно присваивать адрес, но в правой части задано просто целое число. Вы можете точно сказать, что 0xB8000000 — комбинация сегмент-смещение адреса видеопамяти в устаревшей системе, но этот оператор ничего не говорит программе о том, что данное число является адресом. В языке C до появления C99 подобного рода присваивания были разрешены. Однако в C++ применяются более строгие соглашения о типах, и компилятор выдаст сообщение об ошибке, говорящее о несоответствии типов. Если вы хотите

использовать числовое значение в качестве адреса, то должны выполнить приведение типа, чтобы преобразовать числовое значение к соответствующему типу адреса:

```
int * pt;
```

```
pt = (int *) 0xB8000000; // теперь типы соответствуют
```

Теперь обе стороны оператора присваивания представляют адреса, поэтому такое присваивание будет допустимым. Обратите внимание, что если есть значение адреса типа `int`, это не значит, что сам `pt` имеет тип `int`. Например, может существовать платформа, в которой тип `int` является двухбайтовым значением, то время как адрес — четырехбайтовым значением.

Указатели обладают и рядом других интересных свойств, которые мы обсудим, когда доберемся до соответствующей темы. А пока давайте посмотрим, как указатели могут использоваться для управления выделением памяти во время выполнения.

12. ДИНАМИЧЕСКИЙ МАССИВ В C++

12.1. Выделение памяти с помощью операции `new`

Теперь, когда вы получили представление о работе указателей, давайте посмотрим, как с их помощью можно реализовать важнейший прием выделения памяти во время выполнения программы. До сих пор мы инициализировали указатели адресами переменных; переменные — это именованная память, выделенная во время компиляции, и каждый указатель, до сих пор использованный в примерах, просто представлял собой псевдоним для памяти, доступ к которой и так был возможен по именам переменных. Реальная ценность указателей проявляется тогда, когда во время выполнения выделяются неименованные области памяти для хранения значений. В этом случае указатели становятся единственным способом доступа к такой памяти. В языке C память можно выделять с помощью библиотечной функции `malloc()`. Ее можно применять и в C++, но язык C++ также предлагает лучший способ — операцию `new`.

Давайте испытаем этот новый прием, создав неименованное хранилище времени выполнения для значения типа `int` и обеспечив к нему доступ через указатель. Ключом ко всему является операция `new`. Вы сообщаете `new`, для какого типа данных запрашивается память; `new` находит блок памяти нужного

размера и возвращает его адрес. Вы присваиваете этот адрес указателю, и на этом все. Ниже показан пример:

```
int * pn = new int;
```

Часть `new int` сообщает программе, что требуется некоторое новое хранилище, подходящее для хранения `int`. Операция `new` использует тип для того, чтобы определить, сколько байт необходимо выделить. Затем она находит память и возвращает адрес. Далее вы присваиваете адрес переменной `pn`, которая объявлена как указатель на `int`. Теперь `pn` — адрес, а `*pn` — значение, хранящееся по этому адресу. Сравните это с присваиванием адреса переменной указателю:

```
inthiggins;
```

```
int * pt = &higgins;
```

В обоих случаях (`pn` и `pt`) вы присваиваете адрес значения `int` указателю. Во втором случае вы также можете обратиться к `int` по имени `higgins`. В первом случае доступ возможен только через указатель. Возникает вопрос: поскольку память, на которую указывает `pn`, не имеет имени, как обращаться к ней? Мы говорим, что `pn` указывает на объект данных. Это не "объект" в терминологии объектно-ориентированного программирования. Это просто объект, в смысле "вещь". Термин "объект данных" является более общим, чем "переменная", потому что он означает любой блок памяти, выделенный для элемента данных. Таким образом, переменная — это тоже объект, но память, на которую указывает `pn`, не является переменной. Метод обращения к объектам данных через указатель может показаться поначалу несколько запутанным, однако он обеспечивает программе высокую степень управления памятью.

Общая форма получения и назначения памяти отдельному объекту данных, который может быть, как структурой, так и фундаментальным типом, выглядит следующим образом:

```
имяТипа * имя_указателя = new имяТипа;
```

Тип данных используется дважды: один раз для указания разновидности запрашиваемой памяти, а второй — для объявления подходящего указателя. Разумеется, если вы уже ранее объявили указатель на корректный тип, то можете его применить вместо объявления еще одного. В листинге 4.17 демонстрируется применение `new` для двух разных типов.

```
#include<iostream>
```



```

int main()
{
    using namespace std;

    int nights = 1001;

    int * pt = new int; // выделение пространства для int
    *pt = 1001; // сохранение в нем значения

    cout << "nights value = "; // значение nights
    cout << nights << " : location " << &nights << endl; // расположение nights
    cout << "int "; // значение и расположение int
    cout << "value = " << *pt << " : location = " << &pt << endl;

    double * pd = new double; // выделение пространства для double
    *pd = 10000001.0; // сохранение в нем значения double
    cout << "double ";

    cout << "value = " << *pd << " : location = " << &pd << endl;
    // значение и расположение double
    cout << "location of pointer pd: " << &pd << endl;
    // расположение указателя pd
    cout << "size of pt = " << sizeof(pt);
    cout << " : size of *pt = " << sizeof(*pt) << endl;
    cout << "size of pd = " << sizeof(pd);
    cout << " : size of *pd = " << sizeof(*pd) << endl;

    return 0;
}

```

Ниже показан вывод программы из листинга 4.17:

```
nights value = 1001: location 0028F7F8
```

int value = 1001: location = 00033A98

double value = 1e+007: location = 000339B8

location of pointer pd: 0028F7FC

size of pt = 4: size of *pt = 4

size of pd = 4: size of *pd = 8

Естественно, точные значения адресов памяти будут варьироваться от системы к системе.

12.2. Освобождение памяти с помощью операции delete

Использование операции `new` для запрашивания памяти, когда она нужна — одна из сторон пакета управления памятью C++. Второй стороной является операция `delete`, которая позволяет вернуть память в пул свободной памяти, когда работа с ней завершена. Это — важный шаг к максимально эффективному использованию памяти. Память, которую вы возвращаете, или освобождаете, затем может быть повторно использована другими частями программы. Операция `delete` применяется с указателем на блок памяти, который был выделен операцией `new`:

```
int * ps = new int; // выделить память с помощью операции new
```

```
... // использовать память
```

```
delete ps; // по завершении освободить память
```

```
// с помощью операции delete
```

Это освобождает память, на которую указывает `ps`, но не удаляет сам указатель `ps`. Вы можете повторно использовать `ps` — например, чтобы указать на другой выделенный `new` блок памяти. Вы всегда должны обеспечивать сбалансированное применение `new` и `delete`; в противном случае вы рискуете столкнуться с таким явлением, как утечка памяти, т.е. ситуацией, когда память выделена, но более не может быть использована. Если утечки памяти слишком велики, то попытка программы выделить очередной блок может привести к ее аварийному завершению.

Вы не должны пытаться освобождать блок памяти, который уже был однажды освобожден. Стандарт C++ гласит, что результат таких попыток не определен, а это значит, что последствия могут оказаться любыми. Кроме того,

вы не можете с помощью операции delete освобождать память, которая была выделена посредством объявления обычных переменных:

```
int * ps = new int; // нормально
```

```
delete ps; // нормально
```

```
delete ps; // теперь- не нормально!
```

```
int jugs = 5; // нормально
```

```
int * pi = &jugs; // нормально
```

```
delete pi; // не допускается, память не была выделена new
```

Обратите внимание, что обязательным условием применения операции delete является использование ее с памятью, выделенной операцией new. Это не значит, что вы обязаны применять тот же указатель, который был использован с new - просто нужно задать тот же адрес:

```
int * ps = new int; // выделение памяти
```

```
int * pq = ps; // установка второго указателя на тот же блок
```

```
delete pq; // вызов delete для второго указателя
```

Обычно не стоит создавать два указателя на один и тот же блок памяти, т.к. это может привести к ошибочной попытке освобождения одного и того же блока дважды. Но, как вы вскоре убедитесь, применение второго указателя оправдано, когда вы работаете с функциями, возвращающими указатель.

12.3. Использование операции new для создания динамических массивов

Если все, что нужно программе — это единственное значение, вы можете объявить обычную переменную, поскольку это намного проще (хотя и не так впечатляет), чем применение new для управления единственным небольшим объектом данных. Использование операции new более типично с крупными фрагментами данных, такими как массивы, строки и структуры. Именно в таких случаях операция new является полезной. Предположим, например, что вы пишете программу, которой может понадобиться массив, а может, и нет — это зависит от информации, поступающей во время выполнения. Если вы создаете массив простым объявлением, пространство для него распределяется раз и навсегда — во время компиляции. Будет ли

востребованным массив в программе или нет — он все равно существует и занимает место в памяти. Распределение массива во время компиляции называется статическим связыванием и означает, что массив встраивается в программу во время компиляции. Но с помощью `new` вы можете создать массив, когда это необходимо, во время выполнения программы, либо не создавать его, если потребность в нем отсутствует. Или же вы можете выбрать размер массива уже после того, как программа запущена. Это называется динамическим связыванием и означает, что массив будет создан во время выполнения программы. Такой массив называется динамическим массивом. При статическом связывании вы должны жестко закодировать размер массива во время написания программы. При динамическом связывании программа может принять решение о размере массива вовремя своей работы.

Пока что мы рассмотрим два важных обстоятельства относительно динамических массивов: как применять операцию `new` для создания массива и как использовать указатель для доступа к его элементам.

Создание динамического массива с помощью операции `new`

Создать динамический массив на C++ легко; вы сообщаете операции `new` тип элементов массива и требуемое количество элементов. Синтаксис, необходимый для этого, предусматривает указание имени типа с количеством элементов в квадратных скобках. Например, если необходим массив из 10 элементов `int`, следует записать так:

```
int * psome = new int [10] ; // получение блока памяти из 10 элементов
типа int
```

Операция `new` возвращает адрес первого элемента в блоке. В данном примере это значение присваивается указателю `psome`.

Как всегда, вы должны сбалансировать каждый вызов `new` соответствующим вызовом `delete`, когда программа завершает работу с этим блоком памяти. Однако использование `new` с квадратными скобками для создания массива требует применения альтернативной формы `delete` при освобождении массива:

```
delete [] psome; // освобождение динамического массива
```

Присутствие квадратных скобок сообщает программе, что она должна освободить весь массив, а не только один элемент, на который указывает

указатель. Обратите внимание, что скобки расположены между delete и указателем.

Если вы используете new без скобок, то и соответствующая операция delete тоже должна быть без скобок. Если же new со скобками, то и соответствующая операция delete должна быть со скобками. Ранние версии C++ могут не распознавать нотацию с квадратными скобками. Согласно стандарту ANSI/ISO, однако, эффект от несоответствия форма new и delete не определен, т.е. вы не должны рассчитывать в этом случае на какое-то определенное поведение. Вот пример:

```
int * pt = new int;  
  
short * ps = new short [500] ;  
  
delete [] pt; // эффект не определен, не делайте так  
  
delete ps; // эффект не определен, не делайте так
```

Короче говоря, при использовании new и delete необходимо придерживаться перечисленных ниже правил.

- Не использовать delete для освобождения той памяти, которая не была выделена new.
- Не использовать delete для освобождения одного и того же блока памяти дважды.
- Использовать delete [], если применялась операция new[] для размещения массива.
- Использовать delete без скобок, если применялась операция new для размещения отдельного элемента.
- Помнить о том, что применение delete к нулевому указателю является безопасным (при этом ничего не происходит).

Теперь вернемся к динамическому массиву. Обратите внимание, что psome — это указатель на отдельное значение int, являющееся первым элементом блока. Отслеживать количество элементов в блоке возлагается на вас как разработчика. То есть, поскольку компилятор не знает о том, что psome указывает на первое из 10 целочисленных значений, вы должны писать свою программу так, чтобы она самостоятельно отслеживала количество элементов.

На самом деле программе, конечно же, известен объем выделенной памяти, так что она может корректно освободить ее позднее, когда вы воспользуетесь операцией `delete []`. Однако эта информация не является открытой; вы, например, не можете применить операцию `sizeof`, чтобы узнать количество байт в выделенном блоке.

Общая форма выделения и назначения памяти для массива выглядит следующим образом:

```
имя_типа имя_указателя = new имя_типа [количество_элементов] ;
```

Вызов операции `new` выделяет достаточно большой блок памяти, чтобы в нем поместилось `количество_элементов` типа `имя_типа`, и устанавливает в `имя_указателя` указатель на первый элемент. Как вы вскоре увидите, `имя_указателя` можно использовать точно так же, как обычное имя массива.

Лекция 13

12.4. Использование динамического массива

Как работать с динамическим массивом после его создания? Для начала подумаем о проблеме концептуально. Следующий оператор создает указатель `psome`, который указывает на первый элемент блока из 10 значений `int`:

```
int * psome = new int [10]; // получить блок для 10 элементов типа int
```

Представьте его как палец, указывающий на первый элемент. Предположим, что `int` занимает 4 байта. Перемещая палец на 4 байта в правильном направлении, вы можете указать на второй элемент. Всего имеется 10 элементов, что является допустимым диапазоном, в пределах которого можно передвигать палец. Таким образом, операция `new` снабжает всей необходимой информацией для идентификации каждого элемента в блоке.

Теперь взглянем на проблему практически. Как можно получить доступ к этим элементам? С первым элементом проблем нет. Поскольку `psome` указывает на первый элемент массива, то `*psome` и есть значение первого элемента. Но остается еще девять элементов. Простейший способ доступа к этим элементам может стать сюрпризом для вас, если вы не работали с языком C; просто используйте указатель, как если бы он был именем массива. То есть можно писать `psome [0]` вместо `*psome` для первого элемента, `psome [1]` — для

второго и т.д. Получается, что применять указатель для доступа к динамическому массиву очень просто, хотя не вполне понятно, почему этот метод работает. Причина в том, что С и С++ внутренне все равно работают с массивами через указатели. Подобная эквивалентность указателей и массивов — одно из замечательных свойств С и С++. (Иногда это также и проблема, но это уже другая история.) Ниже об этом речь пойдет более подробно. В листинге ниже показано, как использовать `new` для создания динамического массива и доступа к его элементам с применением нотации обычного массива. В нем также отмечается фундаментальное отличие между указателем и реальным именем массива.

```
#include <iostream>

int main()

using namespace std;

double * p3 = newdouble [3]; // пространство для 3 значений double

p3[0] = 0.2; // трактовать p3 как имя массива

p3[1] = 0.5

p3[2] = 0.8

cout<<"p3[1] is " << p3[1] << ".\n"; // вывод p3[1]

p3 = p3 + 1; // увеличение указателя

cout<< "Nowp3[0] is " <<p3[0] « " and "; // выводp3[0]

cout<< "p3[1] is " <<p3[1] << " .\n" ; // выводp3[1]

p3 = p3 - 1; // возврат указателя в начало

delete [] p3; // освобождение памяти

return 0;

}
```

Ниже показан вывод программы из листинга:

p3[1] is 0.5.

Now p3[0] is 0.5 and p3[1] is 0.8.

Как видите, `arraynew.err` использует указатель `p3`, как если бы он был именем массива: `p3 [0]` для первого элемента и т.д. Фундаментальное отличие между именем массива и указателем проявляется в следующей строке:

```
p3 = p3 + 1; // допускается для указателей, но не для имен массивов
```

Вы не можете изменить значение для имени массива. Но указатель — переменная, а потому ее значение можно изменить. Обратите внимание на эффект от добавления 1 к `p3`. Теперь выражение `p3 [0]` ссылается на бывший второй элемент массива. То есть добавление 1 к `p3` заставляет `p3` указывать на второй элемент вместо первого. Вычитание 1 из значения указателя возвращает его назад, в исходное значение, поэтому программа может применить `delete[]` с корректным адресом.

Действительные адреса соседних элементов `int` отличаются на 2 или 4 байта, поэтому тот факт, что добавление 1 к `p3` дает адрес следующего элемента, говорит о том, что арифметика указателей устроена специальным образом. Так оно и есть на самом деле.

13. ФУНКЦИИ В C++

Для того чтобы использовать функцию в C++, вы должны выполнить следующие шаги:

- предоставить определение функции;
- представить прототип функции;
- вызвать функцию.

Если вы планируете пользоваться библиотечной функцией, то она уже определена и скомпилирована. К тому же вы можете, да и должны пользоваться стандартным библиотечным заголовочным файлом, чтобы предоставить своей программе доступ к прототипу. Все что вам остается — правильно вызвать эту функцию. В примерах, которые рассматривались до сих пор в настоящей книге, это делалось много раз. Например, перечень стандартных библиотечных функций C включает функцию `strlen ()` для нахождения длины строки. Ассоциированный стандартный заголовочный файл `cstring` содержит прототип функции для `strlen ()` и ряда других связанных со строками функций. Благодаря предварительной работе, выполненной создателями компилятора, вы используете `strlen ()` без всяких забот.

Когда вы создаете собственные функции, то должны самостоятельно обработать все три аспекта — определение, прототипирование и вызов. В листинге ниже демонстрируются все три шага на небольшом примере.

```
#include<iostream>

void simple (); // прототип функции

int main ()
{
    using namespace std;

    cout << "main () will call the simple () function: \n";

    simple(); // вызов функции

    cout << "main() is finished with the simple () function.\n";

    // cin.get();

    return 0;
}

// Определение функции

void simple ()
{
    using namespace std;

    cout << "I'm but a simple function. \n";

}
```

Ниже показан вывод программы из листинга:

main() will call the simple() function:

I'm but a simple function.

main() is finished with the simple() function.

Выполнение программы в main () останавливается, как только управление передается функции simple (). По завершении simple () выполнение программы возобновляется в функции main (). В этом примере внутри каждого

определения функции присутствует директива `using`, потому что каждая функция использует `cout`. В качестве альтернативы можно было бы поместить единственную директиву `using` над определением функции либо использовать `std::cout`.

Давайте рассмотрим перечисленные выше шаги подробнее.

13.1. Определение функции

Все функции можно разбить на две категории: те, которые не возвращают значений, и те, которые их возвращают. Функции, не возвращающие значений, называются функциями типа `void` и имеют следующую общую форму:

```
void имяФункции(списокПараметров)
{
    оператор(ы)
    return; // не обязательно
}
```

Здесь `списокПараметров` указывает типы и количество аргументов (параметров), передаваемых функции. Позднее в этой главе мы исследуем эту часть более подробно. Необязательный оператор `return` отмечает конец функции. При его отсутствии функция завершается на закрывающей фигурной скобке. Тип функции `void` соответствует процедуре в Pascal, подпрограмме FORTRAN, а также процедурам подпрограмм в современной версии BASIC. Обычно функция `void` используется для выполнения каких-то действий. Например, функция, которая должна напечатать слово "Cheers!" заданное число раз (`n`) может выглядеть следующим образом:

```
void cheers(int n) // возвращаемое значение отсутствует
{
    for (int i = 0; i < n; i++)
        std::cout << "Cheers! ";
    std::cout << std::endl;
```

```
}
```

Параметр `int n` означает, что `cheers ()` ожидает передачи значения типа `int` в качестве аргумента при вызове функции.

Функция с возвращаемым значением передает генерируемое ею значение функции, которая ее вызвала. Другими словами, если функция возвращает квадратный корень из 9.0 (`sqrt (9.0)`), то вызывающая ее функция получит значение 3.0. Такая функция объявляется, как имеющая тот же тип, что и у возвращаемого ею значения.

Вот общая форма:

```
имяТипа имяФункции(списокПараметров)
```

```
{
```

```
    оператор (ы)
```

```
    return значение; // значение приводится к типу имяТипа
```

```
}
```

Функции с возвращаемыми значениями требуют использования оператора `return` таким образом, чтобы вызывающей функции было возвращено значение. Само значение может быть константой, переменной либо общим выражением. Единственное требование — выражение должно сводиться по типу к `имяТипа` либо может быть преобразовано в `имяТипа`. (Если объявленным возвращаемым типом является, скажем, `double`, а функция возвращает выражение `int`, то `int` приводится к `double`.) Затем функция возвращает конечное значение в вызывавшую ее функцию. Язык C++ накладывает ограничения на типы возвращаемых значений: возвращаемое значение не может быть массивом. Все остальное допускается — целые числа, числа с плавающей точкой, указатели и даже структуры и объекты. (Интересно, что хотя функция C++ не может вернуть массив непосредственно, она все же может вернуть его в составе структуры или объекта.)

Как программист, вы не обязаны знать, каким образом функция возвращает значение, но это знание может существенно прояснить концепцию. Обычно функция возвращает значение, копируя его в определенный регистр центрального процессора либо в определенное место памяти. Затем вызывающая программа читает его оттуда. И возвращающая, и вызывающая функции должны подчиняться общему соглашению относительно типа данных,

хранящихся в этом месте. Прототип функции сообщает вызывающей программе, что следует ожидать, а определение функции сообщает программе, что именно она возвращает. Предоставление одной и той же информации в прототипе и определении может показаться излишней работой, но это имеет глубокий смысл. Конечно, если вы хотите, чтобы курьер взял что-то с вашего рабочего стола в офисе, то вы увеличите шансы на правильное выполнение этой работы, если предоставите описание того, что требуется, как курьеру, так и кому-то, кто находится в офисе.

Функция завершается после выполнения оператора `return`. Если функция содержит более одного оператора `return`, например, в виде альтернатив разных выборов `if else`, то в этом случае она прекращает свою работу по достижении первого оператора `return`. Например, в следующем коде конструкция `else` излишняя, однако она помогает понять намерение разработчика:

```
int bigger (int a, int b)
{
    if (a > b )
        return a; // если a > b, функция завершается здесь
    else
        return b; // в противном случае функция завершается здесь
}
```

(Обычно наличие в функции множества операторов `return` может сбивать с толку, и некоторые компиляторы предупреждают об этом. Тем не менее, приведенный выше код понять достаточно просто.)

Функции, возвращающие значения, очень похожи на функции в языках Pascal, FORTRAN и BASIC. Они возвращают значение вызывающей программе, которая затем может присвоить его переменной, отобразить на экране либо использовать каким-то другим способом. Ниже показан простой пример функции, которая возвращает куб значения типа `double`:

```
double cube (double x) // x умножить на x и еще раз умножить на x
{
    return x * x * x; // значение типа double
}
```

```
}
```

Например, вызов функции `cube (1.2)` вернет значение 1.728. Обратите внимание, что здесь в операторе `return` находится выражение. Функция вычисляет значение выражения (в данном случае 1.728) и возвращает его.

13.2. Прототипирование и вызов функции

Вы уже знакомы с тем, как вызываются функции, но, возможно, менее уверенно себя чувствуете в том, что касается их прототипирования, поскольку зачастую прототипы функций скрываются во включаемых (с помощью `#include`) файлах.

В листинге ниже демонстрируется использование функций `cheers ()` и `cube ()`; обратите внимание на их прототипы.

```
#include <iostream>

void cheers(int); // прототип: нет значения возврата
double cube(double x); // прототип: возвращает double

int main()
{
    using namespace std;

    cheers(5); // вызов функции
    cout << "Give me a number: ";

    double side;

    cin >> side;

    double volume = cube(side); // вызов функции
    cout << "A " << side << "-foot cube has a volume of ";
    cout << volume << " cubic feet.\n";

    cheers (cube(2)); // защита прототипа в действии

    return 0;
}
```

```

void cheers(int n)
{
    using namespace std;
    for (int i = 0; i < n; i++)
        cout << "Cheers! ";
    cout << endl;
}

double cube(double x)
{
    return x * x * x;
}

```

Программа из листинга помещает директиву `using` только в те функции, которые используют члены пространства имен `std`. Вот пример запуска:

```
Cheers.' Cheers .' Cheers.' Cheers.' Cheers.'
```

```
Give me a number: 5
```

```
A 5-foot cube has a volume of 125 cubic feet.
```

```
Cheers! Cheers! Cheers! Cheers! Cheers! Cheers! Cheers! Cheers!
```

Обратите внимание, что `main()` вызывает функцию `cheers()` типа `void` с использованием имени функции и аргументов, за которыми следует точка с запятой: `cheers (5);`. Это пример оператора вызова функции. Но поскольку `cube ()` возвращает значение, `main ()` может применять его как часть оператора присваивания:

```
double volume = cube(side);
```

Но как говорилось ранее, необходимо сосредоточиться на прототипах. Что вы должны знать о прототипах? Для начала вы должны понять, почему C++ требует их. Затем, поскольку C++ требует прототипы, вам должен быть известен правильный синтаксис их написания. И, наконец, вы должны оценить, что они дают. Рассмотрим все эти вопросы по очереди, используя листинг 7.2 в качестве основы для обсуждения.

13.2.1. Зачем нужны прототипы?

Прототип описывает интерфейс функции для компилятора. Это значит, что он сообщает компилятору, каков тип возвращаемого значения, если оно есть у функции, а также количество и типы аргументов данной функции.

```
double volume = cube(side);
```

Во-первых, прототип сообщает компилятору, что функция `cube ()` должна принимать один аргумент типа `double`. Если программа не предоставит этот аргумент, то прототипирование позволит компилятору перехватить такую ошибку. Во-вторых, когда функция `cube ()` завершает вычисление, она помещает возвращаемое значение в некоторое определенное место — возможно, в регистр центрального процессора, а может быть и в память. Затем вызывающая функция — `main ()` в данном случае — извлекает значение из этого места. Поскольку прототип устанавливает, что `cube ()` имеет тип `double`, компилятор знает, сколько байт следует извлечь и как их интерпретировать. Без этой информации он может только предполагать, а это то, чем заниматься он не должен.

Но вы все еще можете задаваться вопросом: зачем компилятору нужен прототип? Не может ли он просто заглянуть дальше в файл и увидеть, как определена функция? Одна из проблем такого подхода в том, что он не слишком эффективен. Компилятору пришлось бы приостановить компиляцию `main ()` на то время, пока он прочитает остаток файла. Однако имеется еще более серьезная проблема: функция может и не находиться в том же самом файле. Компилятор C++ позволяет разбивать программу на множество файлов, которые компилируются независимо друг от друга и позднее собираются вместе. В таком случае компилятор может вообще не иметь доступа к коду функции во время компиляции `main ()`. То же самое справедливо и в ситуации, когда функция является частью библиотеки. Единственный способ избежать применения прототипа функции — поместить ее определение перед первым использованием. Это не всегда возможно. Кроме того, стиль программирования на C++ предусматривает размещение функции `main ()` первой, поскольку это в общем случае предоставляет структуру программы в целом.

13.2.2. Синтаксис прототипа

Прототип функции является оператором, поэтому он должен завершаться точкой с запятой. Простейший способ получить прототип — скопировать заголовок функции из ее определения и добавить точку с запятой. Это, собственно, и делает программа из листинга 7.2 с функцией `cube ()`:

```
double cube(double x) ; // добавление ; к заголовку для получения прототипа
```

Однако прототип функции не требует предоставления имен переменных-параметров; достаточно списка типов. Программа из листинга 7.2 строит прототип `cheers ()`, используя только тип аргумента:

```
void cheers(int); // в прототипе можно опустить имена параметров
```

В общем случае в прототипе можно указывать или не указывать имена переменных в списке аргументов. Имена переменных в прототипе служат просто заполнителями, поэтому если даже они заданы, то не обязательно должны совпадать с именами в определении функции.

13.3. Рекурсия

А теперь поговорим совершенно о другой теме. Функция C++ обладает интересной характеристикой — она может вызывать сама себя. (Однако, в отличие от C, в C++ функции `main ()` не разрешено вызывать саму себя.) Эта возможность называется рекурсией. Рекурсия — важный инструмент в некоторых областях программирования, таких как искусственный интеллект, но здесь мы дадим только поверхностные сведения о принципах ее работы.

13.3.1. Рекурсия с одиночным рекурсивным вызовом

Если рекурсивная функция вызывает саму себя, затем этот новый вызов снова вызывает себя и т.д., то получается бесконечная последовательность вызовов, если только код не включает в себе нечто, что позволит завершить эту цепочку вызовов. Обычный метод состоит в том, что рекурсивный вызов помещается внутрь оператора `if`. Например, рекурсивная функция типа `void` по имени `recurs ()` может иметь следующую форму:

```
void recurs(списокАргументов)
{
    операторы1
```



```

if (проверка)
    recurs {аргументы}
операторы2
}

```

В какой-то ситуации проверка возвращает false, и цепочка вызовов прерывается.

Рекурсивные вызовы порождают замечательную цепочку событий. До тех пор, пока условие оператора if остается истинным, каждый вызов recurs () выполняет операторы1 и затем вызывает новое воплощение recurs (), не достигая конструкции операторы2. Когда условие оператора if возвращает false, текущий вызов переходит к операторы2. Когда текущий вызов завершается, управление возвращается предыдущему экземпляру recurs (), который вызвал его. Затем этот экземпляр выполняет свой раздел операторы2 и прекращается, возвращая управление предшествующему вызову, и т.д. Таким образом, если происходит пять вложенных вызовов recurs (), то первый раздел операторы1 выполняется пять раз в том порядке, в котором произошли вызовы, а потом пять раз в обратном порядке выполняется раздел операторы2.

После входа в пять уровней рекурсии программа должна пройти обратно эти же пять уровней. Код в листинге ниже демонстрирует описанное поведение.

```

#include<iostream>

void countdown(int n);

int main ()
{
    countdown (4) ; // вызов рекурсивной функции
    return 0;
}

void countdown(int n)
{
    using namespace std;

```

```
cout << "Counting down ... " <<n<<endl;

if (n > 0)

    countdown(n-1); // функция вызывает сама себя

cout << n << " : Kaboom!\n";
```

Ниже приведен аннотированный вывод программы из листинга:

добавление уровней рекурсии

```
Countingdown ...4
Countingdown ...3
Counting down ...2
Counting down ...1
Counting down ...0
0: Kaboom!
1: Kaboom!
2: Kaboom!
3: Kaboom!
4: Kaboom!
```

Обратите внимание, что каждый рекурсивный вызов создает собственный набор переменных, поэтому на момент пятого вызова она имеет пять отдельных переменных по имени *n* — каждая с собственным значением. Вы можете убедиться в этом, модифицировав код в листинге 7.16 таким образом, чтобы отображать адрес *n* наряду со значением:

```
cout <<"Counting down ...<< n << " (n at " <<&n <<") " << endl;

...

cout << n << ": Kaboom!"; << " (n at " <<&n << ")" << endl;
```

Если сделать так, то вывод программы примет следующий вид:

```
Counting down ....4    (n at 0012FE0C)
Counting down ....3    (n at 0012FD34)
```

Counting down2	(n at 0012FC5C)
Counting down1	(n at 0012FB84)
Counting down0	(n at 0012FAAC)
0: Kaboom!	(n at 0012FAAC)
1: Kaboom!	(n at 0012FB84)
2: Kaboom!	(nat 0012FC5C)
3: Kaboom!	(nat 0012FD34)
4: Kaboom!	(nat 0012FE0C)

Как видите, переменная *p*, имеющая значение 4, размещается в одном месте (в данном примере по адресу 0012FE0C), переменная *p* со значением 3 находится в другом месте (адрес памяти 0012FD34) и т.д. Кроме того, обратите внимание, что адрес переменной *p* для определенного уровня во время этапа Counting down совпадает с ее адресом для того же уровня во время этапа Kaboom!

13.3.2. Рекурсия с множественными рекурсивными вызовами

Рекурсия, в частности, удобна в тех ситуациях, когда нужно вызывать повторяющееся разбиение задачи на две похожие подзадачи меньшего размера. Например, рассмотрим применение такого подхода для рисования линейки. Сначала нужно отметить два конца, найти середину и пометить ее. Затем необходимо применить ту же процедуру для левой половины линейки и правой ее половины. Если требуется больше частей, эта же процедура применяется для каждой из существующих частей. Такой рекурсивный подход иногда называют стратегией "разделяй и властвуй". В листинге ниже данный подход иллюстрируется на примере рекурсивной функции `subdivide()`. Она использует строку, изначально заполненную пробелами, за исключением символов `|` на каждом конце. Затем главная программа запускает цикл из шести вызовов `subdivide()`, каждый раз увеличивая количество уровней рекурсии и печатая результирующую строку. Таким образом, каждая строка вывода представляет дополнительный уровень рекурсии. Чтобы напомнить о подобной возможности, вместо директивы `using` в программе применяется квалификатор `std::`.

```
#include <iostream>
```

```

const int Len = 66;

const int Divs = 6;

void subdivide (char ar[], int low, int high, int level);

int main ()
{
    char ruler[Len];

    int i;

    for (i = 1; i < Len - 2; i + + )

        ruler[i] = ' ';

    ruler [Len - 1] = '\0';

    int max = Len - 2;

    int min = 0;

    ruler [min] = ruler [max] = T;

    std::cout << ruler << std::endl;

    for (i = 1; i <= Divs; i + + )
    {
        subdivide(ruler,min,max, i) ;

        std::cout << ruler << std::endl;

        for (int j = 1; j < Len - 2; j+ + )

            ruler[j] = ' '; // очистка линейки
    }

    return 0;

}

void subdivide (char ar[], int low, int high, int level)
{

```

```

if (level == 0)

return;

int mid = (high + low) / 2;

ar[mid] = 'I';

subdivide (ar, low, mid, level - 1);

subdivide(ar, mid, high, level - 1);

}

```

13.4. Указатели на функции

Любой разговор о функциях C и C++ будет неполным, если не упомянуть указатели на функции. Рассмотрим кратко эту тему, оставив более полное ее раскрытие специализированным источникам.

Функции, как и элементы данных, имеют адреса. Адрес функции — это адрес в памяти, где находится начало кода функции на машинном языке. Обычно пользователю ни к чему знать этот адрес, но это может быть полезно для программы. Например, можно написать функцию, которая принимает адрес другой функции в качестве аргумента. Это позволяет первой функции найти вторую и запустить ее. Такой подход сложнее, чем простой вызов второй функции из первой, но он открывает возможность передачи разных адресов функций в разные моменты времени. То есть первая функция может вызывать разные функции в разное время.

Основы указателей на функции

Проясним этот процесс на примере. Предположим, что требуется спроектировать функцию `estimate ()`, которая оценивает затраты времени, необходимого для написания заданного количества строк кода, и вы хотите, чтобы этой функцией пользовались разные программисты. Часть кода `estimate ()` будет одинакова для всех пользователей, но эта функция позволит каждому программисту применить собственный алгоритм оценки затрат времени. Механизм, используемый для обеспечения такой возможности, будет заключаться в передаче `estimate ()` адреса конкретной функции, которая реализует алгоритм, выбранный данным программистом. Чтобы реализовать этот план, понадобится сделать следующее:

- получить адрес функции;

- объявить указатель на функцию;
- использовать указатель на функцию для ее вызова.

Получение адреса функции

Получить адрес функции очень просто: вы просто используете имя функции без скобок. То есть, если имеется функция `think ()`, то ее адрес записывается как `think`. Чтобы передать функцию в качестве аргумента, вы просто передаете ее имя. Удостоверьтесь в том, что понимаете разницу между адресом функции и передачей ее возвращаемого значения:

```
process(think); // передача адреса think() функции process ()
```

`thought(think());` // передача возвращаемого значения `think()` функции `thought()`. Вызов `process ()` позволяет внутри этой функции вызвать функцию `think()`. Вызов `thought ()` сначала вызывает функцию `think ()` и затем передает возвращаемое ею значение функции `thought ()`.

Объявление указателя на функцию

Чтобы объявить указатель на тип данных, нужно явно задать тип, на который будет указывать этот указатель. Аналогично, указатель на функцию должен определять, на функцию какого типа он будет указывать. Это значит, что объявление должно идентифицировать тип возврата функции и ее сигнатуру (список аргументов). То есть объявление должно предоставлять ту же информацию о функции, которую предоставляет и ее прототип. Например, предположим, что одна из функций для оценки затрат времени имеет следующий прототип:

```
double pam(int); // прототип
```

Вот как должно выглядеть объявление соответствующего типа указателя:

```
double (*pf)(int); // pf указывает на функцию, которая принимает
// один аргумент типа int и возвращает тип double
```

Объявление требует скобок вокруг `*pf`, чтобы обеспечить правильный приоритет операций. Скобки имеют более высокий приоритет, чем операция `*`, поэтому `*pf (int)` означает, что `pf ()` — функция, которая возвращает указатель, в то время как `(*pf) (int)` означает, что `pf` — указатель на функцию:

```
double (*pf)(int); // pf указывает на функцию, возвращающую double
```

```
double *pf(int); // pfO — функция, возвращающая указатель на double
```

После соответствующего объявления указателя `pf` ему можно присваивать адрес подходящей функции:

```
double pam(int);
```

```
double (*pf) (int) ;
```

```
pf = pam; // pf теперь указывает на функцию pam()
```

Обратите внимание, что функция `pam ()` должна соответствовать `pf` как по типу возврата, так и по сигнатуре. Компилятор отклонит несоответствующие присваивания:

```
double ned(double);
```

```
intted(int);
```

```
double (*pf) (int) ;
```

```
pf = ned; // неверно — несоответствие сигнатуры
```

```
pf = ted; // неверно — несоответствие типа возврата
```

Вернемся к упомянутой ранее функции `estimate ()`. Предположим, что вы хотите передавать ей количество строк кода, которые нужно написать, и адрес алгоритма оценки — функции, подобной `pam ()`. Тогда она должна иметь следующий прототип:

```
void estimate(int lines, double (*pf)(int));
```

Это объявление сообщает, что второй аргумент является указателем на функцию, принимающую аргумент `int` и возвращающую значение `double`. Чтобы заставить

`estimate ()` использовать функцию `pam ()`, вы передаете ей адрес `pam`:

```
estimate(50, pam); // вызов сообщает estimate(),
```

```
// что она должна использовать pam()
```

Очевидно, что вся сложность использования указателей на функцию заключается в написании прототипов, в то время как передавать адрес очень просто.

Использование указателя для вызова функции

Теперь обратимся к завершающей части этого подхода — использованию указателя для вызова указываемой им функции. Ключ к этому находится в объявлении указателя. Вспомним, что там (*pf) играет ту же роль, что имя функции. Поэтому все, что потребуется сделать — использовать (*pf), как если бы это было имя функции:

```
double pam(int);
```

```
double (*pf)(int);
```

```
pf = pam; // pf теперь указывает на функцию pam()
```

```
double x = pam(4); // вызвать pam(), используя ее имя
```

```
double y = (*pf)(5); // вызвать pam(), используя указатель pf
```

В действительности C++ позволяет использовать pf, как если бы это было имя функции:

```
double y = pf(5); // также вызывает pam(), используя указатель pf
```

Первая форма вызова более неуклюжа, чем эта, но она напоминает о том, что код использует указатель на функцию.

Домашнее задание № 4

Задача «Файл». Создать файл, записать в него 10 чисел, закрыть, потом вновь открыть файл и найти сумму чисел.

Задача «Знак числа». Определить знак введенного с клавиатуры числа, используя подпрограмму-функцию

$$\text{sign} x = \begin{cases} 1, & x > 0 \\ 0, & x = 0 \\ -1, & x < 0 \end{cases}$$

Задача «Геометрические фигуры». Вычислить площади прямоугольника, треугольника, круга, используя подпрограммы-функции.

Задача «Былая слава». В 1912 году американский флаг «Былая слава» имел 48 звезд (по одной на каждый штат) и 13 полос (по одной на колонию). Напечатать «Былую славу 1912 года».

Задача «Синусоида». Напечатать график функции $y = \sin x$.

Задача «Автоматный распознаватель». Декодировать римскую запись числа, состоящего из любого количества знаков. Правила: $I \rightarrow 1$, $V \rightarrow 5$, $X \rightarrow 10$, $L \rightarrow 50$, $C \rightarrow 100$, $D \rightarrow 500$, $M \rightarrow 1000$. Значение римской цифры не зависит от позиции, а знак — зависит.

Задача «Генератор случайных чисел». Построить генератор псевдослучайных чисел по формуле $s = (m \cdot s + i) \bmod c$, где m , i , c — целые числа. I вариант: $m = 37$, $i = 3$, $c = 64$. II вариант: $m = 25173$, $i = 13849$, $c = 65537$.

Задача «Умножение матриц». Три продавца продают 4 вида товаров. Количество продаваемого товара представлено таблицей A. В таблице B представлены цена каждого товара и комиссионные, полученные от продажи, например:

Таблица A					Таблица B		
Товар \ Продавец	1	2	3	4	Товар	Цена	Комиссионные
1	5	2	0	10	1	1,20	0,50
2	3	5	2	5	2	2,80	0,40
3	20	0	0	0	3	5,00	1,00
					4	2,00	1,50

Задать соответствующие таблицам матрицы A и B, получить матрицу $C = AxV$ и определить: 1) какой продавец выручил больше всего денег с продажи, какой — меньше; 2) какой получил наибольшие комиссионные, какой — наименьшие; 3) чему равна общая сумма денег, вырученных за проданные товары; 4) сколько всего комиссионных получили продавцы; 5) чему равна общая сумма денег, прошедших через руки продавцов?

Задача «Системы счисления». Программа должна считывать с клавиатуры число, записанное в одной системе счисления, и выводить на экран это число в записи по другому основанию, например: исходное число — 112D, старое основание — 16, новое основание — 8, результат — 10455.

14. КЛАСС ШАБЛОНА VECTOR

В вычислительной технике термин вектор соответствует массиву, а не математическим векторам, которые обсуждались в главе 11. (С точки зрения математики N-мерный математический вектор может быть представлен набором из N компонентов, и в этом смысле математический вектор подобен на N-мерному массиву. Однако математический вектор обладает дополнительными свойствами, такими как скалярное и векторное произведение, которые для компьютерного вектора не обязательны.) Компьютерный вектор — это набор однотипных значений, к которым можно обращаться в произвольном порядке. То есть, например, с помощью индекса можно получить непосредственный доступ к десятому элементу вектора без

необходимости считывания девяти предыдущих элементов. Итак, класс `vector` должен обладать возможностями, аналогичными предоставляемым классами `valarray` и `ArrayTP`, а также классом `array`. Это значит, что можно создать объект `vector`, присвоить один объект `vector` другому и применять операцию `[]` для доступа к отдельным элементам объекта `vector`. Чтобы сделать этот класс обобщенным, его нужно реализовать в виде класса шаблона. Именно это и делает библиотека STL, определяя шаблон `vector` в заголовочном файле `vector` (ранее известный как `vector.h`).

Для создания объекта шаблона `vector` используется обычная нотация `<тип>`, с помощью которой указывается необходимый тип. Кроме того, шаблон `vector` использует динамическое выделение памяти, и для указания количества элементов вектора можно применять инициализирующий аргумент:

```
#include vector

using namespace std;

vector<int> ratings(5); // вектор из 5 значений типа int

int n;

cin >> n;

vector<double> scores(n); // вектор из n значений типа double
```

После создания объекта `vector` перегрузка операции `[]` позволяет обращаться к элементам вектора, используя обычную нотацию массивов:

```
ratings[0] = 9;

for (int i = 0; i < n; i++)

    cout << scores[i] << endl;
```

Применение класса `vector` в достаточно непротивительном приложении продемонстрировано в листинге ниже. Эта программа создает два объекта `vector`, один из которых содержит элементы типа `int`, а второй — `string`. В каждом объекте находится по 5 элементов.

```
#include<iostream>

#include<string>

#include <vector>
```

```

const int NUM. = 5;

int main ()
{
using std::rvector;
using std::string;
using std::cin;
using std::cout;
using std::endl;

vector<int> ratings (NUM.) ;
vector<string> titles(NUM) ;

cout << "You will do exactly as told. You will enterXn"
<< NUM << " book titles and your ratings (0-10) . \n";

// запрос книг и их рейтингов
int i;
for (i = 0; i < NUM; i + + )
{
cout << "Enter title #" << i + 1 << " : "; // ввoд названия книги
getline(cin,titles[i]) ;
cout << "Enter your rating (0-10): "; // ввoд рейтинга книги
cin » ratings[i];
cin.get ();
}

cout << "Thank you. You entered the following:\n"
<< "Rating\tBook\n"; // вывод списка книг с рейтингами
for (i = 0; i < NUM; i + + )

```

```

{
cout << ratings [i] << "\t" << titles [i] << endl;
}

return 0;
}

```

Ниже приведен пример запуска программы из листинга:

You will do exactly as told. You will enter

5 book titles and your ratings (0-10) .

Enter title #1: The Cat Who Knew C++

Enter your rating (0-10): 6

Enter title #2: Felonious Felines

Enter your rating (0-10): 4

Enter title #3: Warlords of Wonk

Enter your rating (0-10): 3

Enter title #4: Don't Touch That Metaphor

Enter your rating (0-10) : 5

Enter title #5: Panic Oriented Programming

Enter your rating (0-10) : 8

Thank you. You entered the following:

Rating Book

6 The Cat Who Knew C++

4 Felonious Felines

3 Warlords of Wonk

5 Don't Touch That Metaphor

8 PanicOrientedProgramming

Все, что делает эта программа — использует шаблон `vector` в качестве удобного средства создания динамического массива. В следующем разделе будет показан пример применения других методов этого класса.

14.1. Что еще можно делать с помощью векторов

Что же еще, помимо выделения памяти, позволяет делать шаблон `vector`? Все контейнеры STL предоставляют набор базовых методов. К ним относятся: `size()`, который возвращает количество элементов в контейнере; `swap()`, обменивающий содержимое двух контейнеров; `begin()`, возвращающий итератор, который ссылается на первый элемент в контейнере; `end()`, возвращающий итератор, который представляет область памяти, следующую за последним элементом контейнера.

Что собой представляет итератор? Это обобщение указателя. В действительности он может быть указателем. Или же он может быть объектом, для которого определены операции над указателями, такие как разыменование (например, `operator*()`) и инкремент (например, `operator++()`). Как станет видно в дальнейших примерах, обобщение указателей позволяет STL предоставлять однотипный интерфейс для множества классов-контейнеров, включая те, для которых обычные указатели не работают. В каждом классе-контейнере определяется соответствующий итератор. Типом этого итератора будет `typedef` по имени `iterator` с областью видимости класса.

Например, для объявления итератора для класса `vector` типа `double` применяется следующий синтаксис:

```
vector<double>::iterator pd; // pd — это итератор
```

Предположим, что `scores` — это объект `vector<double>`:

```
vector<double> scores;
```

Теперь итератор `pd` можно применять в коде, как показано ниже:

```
pd = scores.begin(); // обеспечение того, чтобы pd указывал на первый элемент
```

```
*pd = 22.3; // разыменование pd и присваивание значения первому элементу
```

```
++pd; // обеспечение того, чтобы pd указывал на следующий элемент
```

Как видите, итератор ведет себя подобно указателю. Кстати говоря, автоматическое выведение типа C++11 может быть полезно еще в одной ситуации. Вместо оператора:

```
vector<double>::iteratorpd = scores.begin();
```

можно использовать следующий оператор:

```
autopd = scores.begin(); // автоматическое выведение типа C++11
```

Но что подразумевается в примере под областью памяти, следующей за последним элементом? Это итератор, который указывает на элемент, следующий за последним элементом в контейнере. Идея применения этого итератора сходна с идеей использования нулевого символа, расположенного непосредственно за последним символом в строке стиля C, за исключением того, что нулевой символ — это значение элемента, а следующий за последним элемент — это указатель (или итератор) элемента. Функция-член `end()` определяет позицию, следующую за последним элементом контейнера. Если установить итератор на первый элемент контейнера и увеличивать его, то, в конце концов, будет достигнут элемент, следующий за последним — т.е. все содержимое контейнера было пройдено. Таким образом, если `scores` и `pd` определены как в предыдущем примере, то все содержимое контейнера можно отобразить с помощью следующего кода:

```
for (pd = scores .begin () ; pd != scores .end() ; pd+ + )  
  
    cout<< *pd<<endl;;
```

Описанные выше методы имеются у всех контейнеров. В шаблоне `vector` есть также некоторые методы, которые присутствуют не во всех контейнерах STL. Один из полезных методов — `push_back()` — добавляет элемент в конец объекта `vector`. При выполнении этой операции осуществляется дополнительное выделение памяти, и размер вектора увеличивается, чтобы в него поместились добавляемые элементы. Это означает, что можно использовать код, подобный следующему:

```
vector<double>scores; // создание пустого вектора  
  
double temp;  
  
while (cin >> temp && temp >= 0)  
  
    scores.push_back(temp);
```


диапазон. Обычно этот диапазон является частью другого объекта контейнера. Например, следующий код вставляет все элементы вектора `newv`, за исключением первого, после первого элемента вектора `old_v`:

```
vector<int> old_v;  
  
vector<int> new_v;  
  
...  
  
old_v.insert(old_v.begin (), newv.begin() + 1, new_v.end());
```

Кстати, здесь может пригодиться метод `end ()`, поскольку он облегчает добавление элементов в конец вектора. В этом коде новые данные добавляются после позиции `old.end ()`, т.е. после последнего элемента вектора:

```
old_v.insert(old_v.end(), new_v.begin() + 1, newv.end());
```

14.2. Дополнительные возможности векторов

Существует множество задач, которые часто выполняются с массивами, такие как поиск, сортировка, тасование и т.д. Содержит ли класс шаблона вектора методы для выполнения этих часто выполняемых операций? Нет! В STL применяется более широкий подход, заключающийся в определении автономных (не являющихся членами класса) функций для выполнения этих операций. Таким образом, вместо того чтобы определять отдельную функцию-член `find()` для каждого класса контейнера, в библиотеке определяется одна автономная функция `find()`, которая может использоваться для всех классов контейнеров. Такой подход позволяет значительно уменьшить дублирование кода. Например, предположим, что имеется 8 контейнеров и 10 операций над ними. Если бы в каждом классе определялась своя функция-член для каждой операции, то потребовалось бы 8x10, или 80, отдельных определений функций-членов. Но при использовании подхода, применяемого в STL, потребуется всего лишь 10 отдельных определений автономных функций. И при соблюдении соответствующих рекомендаций 10 существующих автономных функций можно было бы использовать для выполнения операций поиска, сортировки и т.д. также в определении нового класса контейнера.

Вместе с тем, в ряде случаев STL определяет функцию-член даже при наличии автономной функции для решения той же самой задачи. Причина этого в том, что специфичные для класса алгоритмы выполнения некоторых действий эффективнее более общих алгоритмов. Поэтому функция `swap ()` класса `vector` будет эффективнее автономной функции `swap ()`. С другой стороны,

автономная версия позволит обменивать содержимое двух контейнеров различного типа.

Давайте более подробно рассмотрим три типичных функции STL: `for_each ()`, `random_shuffle ()` и `sort ()`. Функция `for_each ()` работает с любым классом-контейнером. Она принимает три аргумента. Первые два аргумента — это итераторы, определяющие диапазон, а третий аргумент — указатель на функцию. (В более общем смысле последний аргумент представляет собой объект функции (функтор). Функторы рассматриваются далее в этой главе.) Затем функция `for_each ()` применяет указанную в аргументе функцию ко всем элементам контейнера в указанном диапазоне. Функция, указанная в аргументе, не должна изменять значение элементов контейнера. Функцию

```
for_each () можно использовать вместо цикла for. Например, код  
  
vector<Review>::iterator pr;  
  
for (pr = books.begin () ; pr != books.end () ; pr++)  
  
ShowReview(*pr) ;
```

можно заменить следующим кодом:

```
foreach(books.begin (), books.end(), ShowReview);
```

Это позволяет избежать явного использования переменных итераторов.

Функция `random_shuffle ()` принимает в качестве аргументов два итератора, которые указывают границы диапазона, и тасует элементы в этом диапазоне случайным образом. Например, следующий оператор изменяет случайным образом порядок следования всех элементов вектора `books`:

```
random_shuffle(books.begin (), books.end());
```

В отличие от функции `for_each ()`, которая работает с любым классом-контейнером, `random_shuffle ()` требует, чтобы класс контейнера разрешал доступ к своим элементам в произвольном порядке. Класс `vector` удовлетворяет этому требованию.

Функция `sort ()` также требует, чтобы контейнер поддерживал произвольный доступ. Существуют две версии этой функции. Первая использует два итератора, определяющих границы диапазона, и сортирует элементы этого диапазона с помощью операции `<`, определенной для типа элемента, который хранится в контейнере. Например, следующий код

сортирует содержимое `coolstuff f` по возрастанию с применением встроенной операции `<` для сравнения значений:

```
vector<int> coolstuff;

....

sort(coolstuff.begin(), coolstuff.end());
```

Если элементами контейнера являются объекты, типы которых определены пользователем, то для этого типа объектов должна быть определена функция `operator< ()`, в противном случае функция `sort ()` работать не будет. Например, вектор, содержащий объекты `Review`, можно было бы сортировать либо с помощью функции-члена класса

`Review`, либо с помощью автономной функции `operator< ()`. Поскольку `Review` — это структура, ее члены открыты, и в этом случае можно применять автономную функцию:

```
bool operator<(const Review & r1, const Review & r2)
{
    if (r1.title < r2.title)
        return true;

    else if (r1.title == r2.title && r1.rating < r2.rating)
        return true;

    else
        return false;
}
```

Используя подобную функцию, можно отсортировать вектор объектов `Review` (такой как `books`):

```
sort(books.begin(), books.end());
```

Эта версия функции `operator< ()` сортирует члены `title` в лексикографическом порядке. Если у двух объектов поля `title` совпадают, объекты сортируются по полю `rating`. Но предположим, что требуется выполнить сортировку в убывающем порядке или по рейтингам `rating`, а не по

заглавиям `title`. В этом случае можно использовать вторую форму функции `sort()`. Она принимает три аргумента. Первые два, как и в предыдущем случае, являются итераторами, определяющими диапазон. Третий аргумент — указатель на функцию (точнее — функтор), которая будет использоваться вместо `operator<()` для выполнения сравнения. Функция должна возвращать значение, которое можно преобразовать в тип `bool`, причем значение `false` означает, что аргументы функции расположены в неправильном порядке. Вот пример такой функции:

```
bool WorseThan(const Review & r1, const Review & r2)
{
    if (r1.rating < r2.rating)
        return true;
    else
        return false;
}
```

Располагая этой функцией, можно написать следующий оператор для сортировки вектора `books`, состоящего из объектов `Review`, по возрастанию рейтинга:

```
sort(books.begin(), books.end(), WorseThan);
```

Обратите внимание, что функция `WorseThan()` сортирует объекты `Review` менее точно, чем `operator<()`. Если член `title` объектов совпадает, функция `operator<()` осуществляет сортировку по полю `rating`. Но если и эти два поля объектов совпадают, функция `WorseThan()` считает их эквивалентными. Первый вид упорядочения называется полным упорядочением, а второй — строгим квазиупорядочением. При полном упорядочении, если оба выражения $a < b$ и $b < a$ ложны, то a должно быть идентично b . При строгом квазиупорядочении это не так. Объекты могут быть полностью идентичными, а могут совпадать только по одному критерию, такому, как поле `rating` в примере с функцией `WorseThan()`. Поэтому при строгом квазиупорядочении лучше говорить, что объекты эквивалентны, а не идентичны.

15. ФАЙЛОВЫЙ ВВОД-ВЫВОД

Большинство компьютерных программ работает с файлами. Текстовые процессоры создают файлы документов. Программы баз данных создают файлы и ищут в них информацию. Компиляторы считывают файлы исходного кода и генерируют исполняемые файлы. Сам по себе файл — это группа байтов, сохраненных на некотором устройстве, возможно, магнитной ленте, оптическом диске или жестком диске. Как правило, операционная система управляет файлами, отслеживая их местоположение, размеры, дату их создания и т.п. Если только программирование не выполняется на уровне операционной системы, обычно не нужно беспокоиться о подобных нюансах. Все что требуется — это способ подключения программы к файлу, способ считывания программой его содержимого и способ создания и записи файлов программой. Перенаправление (описанное ранее в настоящей главе) может предоставить некоторую поддержку файлов, но значительно более ограниченную, чем явный ввод-вывод, осуществляемый из программы. К тому же перенаправление обеспечивается операционной системой, а не C++, поэтому оно доступно не во всех системах. В этой книге уже затрагивалась тема файлового ввода-вывода, а в этой главе она освещается более подробно.

Пакет классов ввода-вывода C++ управляет файловым вводом и выводом в основном так же, как он делает это со стандартным вводом и выводом. Чтобы записывать в файл, вы создаете объект `ofstream` и используете такие его методы, как операция вставки `<<` или `write()`. Чтобы читать из файла, вы создаете объект `ifstream` и применяете методы `istream` наподобие операции извлечения `>>` и `get()`. Однако файлы требуют больше внимания, нежели стандартный ввод и вывод. Например, только что открытый файл нужно ассоциировать с потоком. Файл можно открыть в режиме только для чтения, только для записи либо для чтения и записи. Если вы записываете в файл, может потребоваться создание нового файла, замена старого либо добавление информации в существующий файл. Или же может возникнуть необходимость в перемещении по файлу назад и вперед. Чтобы помочь в решении этих задач, в C++ определено несколько новых классов в заголовочном файле `fstream` (бывший `fstream.h`), включая класс `ifstream` для файлового ввода и класс `ofstream` для файлового вывода. В C++ также определен класс `fstream` для одновременного файлового ввода и вывода. Эти классы являются

производными от классов, определенных в заголовочном файле `iostream`, поэтому объекты новых классов могут использовать методы, которые уже были изучены ранее.

15.1. Простой файловый ввод-вывод

Предположим, программа должна выполнять запись в файл. Понадобится предпринять следующие действия.

1. Создать объект `ofstream` для управления выходным потоком.

2. Ассоциировать этот объект с конкретным файлом.

3. Использовать объект так же, как нужно было бы использовать `cout`. Единственным отличием будет то, что вывод направляется в файл вместо экрана.

Чтобы достичь этого, нужно начать с подключения заголовочного файла `fstream`. Его подключение в большинстве, хотя и не во всех реализациях, автоматически подключает файл `iostream`, поэтому явное подключение `iostream` не обязательно. Затем нужно объявить объект типа `ofstream`:

```
ofstream fout; // создание объекта fout типа ofstream
```

Именем объекта может быть любое допустимое в C++ имя, такое как `fout`, `outFile`, `cgate` или `didid`.

Затем этот объект нужно ассоциировать с конкретным файлом. Это можно сделать с помощью метода `open()`. Предположим, например, что требуется открыть файл `jar.txt` для вывода. Это можно было бы сделать следующим образом: `fout.open("jar.txt");` // связывание `fout` с файлом `jar.txt`

Эти два шага (создание объекта и ассоциация файла с ним) можно совместить в одном операторе, используя другой конструктор:

```
ofstream fout("jar.txt"); // создание объекта fout и его ассоциирование  
//с файлом jar.txt
```

После того, как все это сделано, `fout` (или любое другое выбранное вами имя) можно будет использовать таким же образом, как и `cout`. Например, если требуется поместить слова `Dull Data` в этот файл, это можно сделать следующим образом:

```
fout << "Dull Data";
```

Действительно, поскольку ostream — это базовый класс для ofstream, можно применять все методы ostream, включая разнообразные операции вставки, а также методы форматирования и манипуляторы. Класс ofstream использует буферизованный вывод, поэтому при создании объекта типа ofstream, такого как fout, программа выделяет пространство для выходного буфера. Если вы создадите два объекта ofstream, программа создаст два буфера — по одному для каждого объекта. Объект ofstream, подобный fout, накапливает выходные данные программы байт за байтом, а затем, когда буфер заполняется, передает его содержимое в файл назначения. Поскольку дисководы спроектированы для передачи данных более крупными порциями, а не побайтно, буферизованный подход значительно увеличивает скорость передачи данных из программы в файл.

Такое открытие файла для вывода создает новый файл, если файла с указанным именем не существует. Если же файл с этим именем существовал ранее, то действие по его открытию усекает его так, чтобы вывод начинался в пустой файл. Позднее в этой главе будет показано, как открыть существующий файл и сохранить его содержимое.

Действия для выполнения чтения из файла во многом подобны тем, которые необходимы для выполнения записи в файл.

1. Создать объект ifstream для управления входным потоком.
2. Ассоциировать этот объект с конкретным файлом.
3. Использовать объект так же, как нужно было бы использовать с in.

Шаги для чтения из файла похожи на те, которые нужно выполнить для записи в файл. Для начала, конечно, нужно подключить заголовочный файл fstream. Затем необходимо объявить объект ifstream и ассоциировать его с именем файла. Для этого можно использовать два оператора или же один:

```
ifstream fin; // создать объекта fin типа ifstream
```

```
fin.open("jellyjar.txt"); // открытие файла jellyjar.txt для чтения
```

```
// Один оператор
```

```
ifstream fis("jamjar.dat"); // создание объекта fis и его ассоциирование
```

```
// с файлом Два оператора jamjar.txt
```

Затем объект `fin` или `fis` можно использовать почти так же, как `cin`. Например, можно использовать следующий код:

```
char ch;

fin >> ch; // считывание символа из файла jellyjar.txt

char buf[80];

fin >> buf; // считывание слова из файла

fin.getline(buf, 80); // считывание строки из файла

string line;

getline(fin, line); // считывание из файла в строковый объект
```

Ввод, как и вывод, также буферизуется, поэтому создание объекта `ofstream`, такого как `fin`, создает входной буфер, которым управляет объект `fin`. Как и в случае вывода, буферизация обеспечивает гораздо более быстрое перемещение данных, чем при побайтной передаче.

Соединение с файлом закрывается автоматически, когда объекты ввода и вывода уничтожаются, например, по завершении программы. Кроме того, соединение с файлом можно закрыть явно, используя для этого метод `close ()`:

```
fout.close (); // закрытие соединения вывода с файлом

fin.close (); // закрытие соединения ввода с файлом
```

Закрытие такого соединения не уничтожает поток; оно просто отключает его от файла. Однако средства управления потоком никуда не деваются. Например, объект `fin` продолжает существовать, как и входной буфер, которым он управляет. Как будет показано позже, этот поток можно заново подключить к тому же или к другому файлу.

Рассмотрим краткий пример. Программа в листинге ниже запрашивает имя файла. Она создает файл с этим именем, записывает в него некоторую информацию и закрывает файл. Закрытие файла очищает буфер, тем самым гарантируя обновление файла. Затем программа открывает тот же файл для чтения и отображает его содержимое. Обратите внимание, что программа использует `fin` и `fout` таким же образом, как если бы применялись `cin` и `cout`. Кроме того, программа считывает имя файла в объект `string`, а затем использует метод `c_str ()` для передачи конструкторам `ofstream` и `ifstream` аргументов — строк в стиле C.

```

#include <iostream> // для многих систем не требуется

#include <fstream>

#include <string>

int main (J

{

using namespace std;

string filename;

cout << "Enter name for new file: "; // запрос имени нового файла

cin >> filename;

// Создание объекта выходного потока для нового файла и назначение
ему имени fout

ofstream fout(filename.c_str ());

fout << "For your eyes only!\n"; // запись в файл

cout << "Enter your secret number: "; // вывод на экран

float secret;

cin >> secret;

fout << "Your secret number is " << secret << endl;

fout.close (); // закрытие файла

// Создание объекта входного потока для нового файла и назначение ему
имени fin

ifstream fin(filename.c_str ());

cout << "Here are the contents of " << filename << " :\n" ;

char ch;

while (fin.get (ch)) // чтение символа из файла

cout << ch; // и его вывод на экран

cout << "Done\n";

```



```
fin.close () ;  
  
return 0;  
  
}
```

Ниже приведен пример выполнения программы из листинга:

```
Enter name for new file: pythag  
  
Enter your secret number: 3.14159  
  
Here are the contents of pythag:  
  
For your eyes only!  
  
Your secret number is 3.14159  
  
Done *
```

Если вы просмотрите каталог, содержащий программу, то найдете там файл `pythag` и, загрузив его в любом текстовом редакторе, увидите то же содержимое, что и в выводе программы.

15.2. Проверка потока и `is_open()`

Классы файловых потоков C++ наследуют член, описывающий состояние потока, от класса `ios_base`. Этот член, как упоминалось ранее, хранит информацию, отражающую состояние потока — о том, что все в порядке, что достигнут конец файла, о том, произошел ли сбой операции ввода-вывода, и т.д. Если все в порядке, состояние потока равно нулю (отсутствие новостей — уже хорошая новость). Разнообразные другие состояния описываются установкой конкретных битов в единицу. Классы файловых потоков также наследуют методы `ios_base`, которые сообщают о состоянии потока и перечислены в табл. 17.4. Можно проверить состояние потока, чтобы выяснить, успешно ли завершилась последняя операция с этим потоком. Для файловых потоков это включает в себя проверку успешности или неудачи операции открытия файла. Например, попытка открытия для ввода несуществующего файла устанавливает флаг `failbit`. Поэтому можно было бы выполнить проверку следующим образом:

```
fin.open(argv[file]);  
  
if (fin.failO) // попытка открытия не удалась
```

```
{
    ....
}
```

Или же, поскольку объект `ifstream`, подобно `istream`, преобразуется в тип `bool`, когда ожидается именно этот тип, можно было бы использовать следующий код:

```
fin.open(argv[file]);

if (!fin) // попытка открытия не удалась
{
    ...
}
```

Однако новые реализации C++ предлагают лучший способ проверки того, открыт ли файл — метод `is_open ()`.

```
if (!fin.is_open ()) // попытка открытия не удалась
{
    ...
}
```

Преимущество этого способа состоит в том, что он проверяет также наличие некоторых незначительных проблем, которые остаются незамеченными другими формами проверки.

15.3. Открытие нескольких файлов

Иногда может требоваться, чтобы программа открывала более одного файла. Стратегия открытия нескольких файлов зависит от того, как они будут использоваться. Если требуется, чтобы два файла были открыты одновременно, нужно создать отдельный поток для каждого файла. Например, программа, которая сравнивает два отсортированных файла и отправляет результат в третий, должна создать два объекта `ifstream` для двух входных файлов и один объект `ofstream` — для выходного файла. Количество файлов, которые можно открыть одновременно, зависит от операционной системы.

Однако можно запланировать последовательную обработку файлов. Предположим, что требуется подсчитать, сколько раз имя появляется в наборе из 10 файлов. В этом случае можно открыть единственный поток и по очереди ассоциировать его с каждым из этих файлов. При этом ресурсы компьютера используются экономнее, чем при открытии отдельного потока для каждого файла. Чтобы применить такой подход, нужно объявить объект `ifstream` без его инициализации, а затем с помощью метода `open()` ассоциировать поток с файлом. Например, последовательное считывание двух файлов можно было бы организовать следующим образом:

```
ifstream fin; // создание потока конструктором по умолчанию

fin.open("fat.txt"); // ассоциирование потока с файлом fat.txt

...           // выполнение каких-либо действий

fin.close(); // разрыв связи потока с файлом fat.txt

fin.clear(); // сброс fin (может не требоваться)

fin.open("rat.txt"); // ассоциирование потока с файлом rat.txt

...

fin.close();
```

16. ПАРАМЕТРЫ КОМАНДНОЙ СТРОКИ В C++

При запуске программы из командной строки, ей можно передавать дополнительные параметры в текстовом виде. Например, следующая команда

```
ping -t 5 google.com
```

Будет отправлять пакеты на адрес `google.com` с интервалом в 5 секунд. Здесь мы передали программе `ping` три параметра: «-t», «5» и «google.com», которые программа интерпретирует как задержку между запросами и адрес хоста для обмена пакетами.

В программе эти параметры из командной строки можно получить через аргументы функции `main` при использовании функции `main` в следующей форме:

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

Первый аргумент содержит количество параметров командной строки. Второй аргумент — это массив строк, содержащий параметры командной строки. Т.е. первый аргумент указывает количество элементов массива во втором аргументе.

Первый элемент массива строк (`argv[0]`) всегда содержит строку, использованную для запуска программы (либо пустую строку). Следующие элементы (от 1 до `argc - 1`) содержат параметры командной строки, если они есть. Элемент массива строк `argv[argc]` всегда должен содержать 0.

Пример 1

```
#include<iostream>

Usingnamespacestd;

Intmain(int argc, char *argv[]) {

for (inti = 0; i<argc; i++) { // Выводим список аргументов в цикле

cout<<"Argument "<< i <<" : "<< argv[i] << endl;

}

Return0;

}
```

Откройте командную строку и запустите оттуда скомпилированную программу.

Для получения числовых данных из входных параметров, можно использовать функции `atoi` и `atol`.

СПИСОК ЛИТЕРАТУРЫ

1. Бьярне Страуструп. Программирование. Принципы и практика с использованием С++. – М.: Вильямс, 2016. – 1328 с.
2. Стивен Язык программирования С++. Лекции и упражнения. – М.: Вильямс, 2017. – 1248 с.
3. Эндрю Кениг, Барбара Э. Му. Эффективное программирование на С++. Практическое программирование на примерах. – М.: Вильямс, 2016. – 368 с.
4. Алексей Васильев. Программирование на С++ в примерах и задачах. – М.: Эксмо, 2016. – 368 с.
5. Учебник по С++ для начинающих [Электронный ресурс] <http://www.programmersclub.ru/main>
6. Литвиненко Н. А. Технология программирования на С++ [Электронный ресурс] http://www.proklondike.com/books/cpp/technology_of_programming_on_cplusplus.html
7. Стефан Р. Дэвис. С++ для чайников [Электронный ресурс] http://www.proklondike.com/books/cpp/cplusplus_for_beginners.html