Министерство науки и высшего образования Российской Федерации

АМУРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет математики и информатики

Т. А. Галаган

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ С++

Часть 1

Благовещенск
2021

ББК 32.973-018.1я73
Г 15

Печатается по решению редакционно-издательского совета факультета математики и информатики Амурского государственного университета

Т.А. Галаган

Программирование на языке С++. Часть 1. Учебное пособие. Для студентов направлений подготовки 01.03.02 Прикладная математика и информатика, 09.03.01 Информатика и вычислительная техника, 09.03.02 Информационные системы и технологии, 09.03.03 Программная инженерия очной формы обучения – Благовещенск: Амурский гос. ун-т, 2021.

В учебном пособие рассмотрены основные элементарные конструкции языка программирования С++, примеры простых для понимания программ. Первая часть пособия посвящена структурному и модульному подходам создания программ. Пособие предназначено для студентов, начинающих изучение программирования с нуля.

Рецензенты:

Е.Ф Алутина, канд. физ.-мат. наук, доцент кафедры информатики и методики преподавания информатики Благовещенского государственного педагогического университета;

Е.М Веселова, канд. физ.-мат. наук, доцент кафедры математического анализа и моделирования Амурского государственного университета;

ВВЕДЕНИЕ

Пособие посвящено популярному языку программирования С++. Оно предназначено студентам первого курса, начинающих свой путь в программирования.

Язык С++ менее строго структурирован по сравнению с другими языками высокого уровня и предлагает свободу выбора альтернативных способов при решении задачи. Это достаточно сложный язык с массой правил и сложных терминов. Конструкции языка, напоминающие выражения на английском языке, характерны для многих современных языков программирования. Поэтому изучение С++ позволяет в дальнейшем с легкостью освоить любой другой язык программирования. Программы на С++ отличаются высокой эффективностью и дают возможность создавать не только пользовательские, но и системные приложения.

Пособие не претендует на полное освещение языка С++. Оно предназначено для начинающих программистов изложено доступным языком, но может быть интересно и тем кто уже знаком с другими языками программирования. Например, в пособии мало внимания уделено преобразованию типов, не рассмотрены низкоуровневые возможности С++.

В пособие кроме непосредственного изложения правил синтаксических правил С++ рассматриваются методы разработки алгоритмов и создания сложных типов данных.

Пособие содержит материал лекций дисциплины «Программирование» и может использоваться при выполнение самостоятельной работы в рамках изучения указанной дисциплины. Поэтому материал изложен последовательно и позволяет создавать простые программы с самого начала изучения.

Глава 1. ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ

Этапы решения задач

Программированием называют планирование (распределение во времени) или исполнение определенных заданий или событий. Компьютерное программирование – это планирование последовательности действий (шагов), которую должен выполнить компьютер.

Любая компьютерная программа состоит из последовательности команд (инструкций), сообщающих компьютеру, что он должен делать и каким образом обрабатывать данные.

Процесс создания программы состоит из двух основных этапов: *решение задачи* и ее *реализация*. Этап решения задачи включает в себя:

- *анализ и спецификацию*, т.е. точное понимание и определение сути задачи, содержащей информацию, необходимую для ее решения, а также описание технических и функциональных требований к ее решению;
- *общее решение (алгоритм)* – разработку логической последовательности шагов, приводящих к решению поставленной задачи;
- *проверку* – подтверждение правильности решения, например, повторением всех его этапов.

Этап реализации, в свою очередь, содержит:

- *программу (конкретное решение)* – перевод алгоритма на язык программирования;
- *тестирование* – запуск программы на компьютере, получение результата и проверка результата вручную. При обнаружении не соответствия проводится анализ алгоритма и его реализации (программы), нахождения источника ошибок и их исправление.

После завершения перечисленных этапов начинается этап *сопровождения*, включающий:

- *использование* (*эксплуатацию*) программы;
- *поддержку* программы, т.е. изменение в соответствии с требованиями

пользователей, а также исправление ошибок, выявленных при ее эксплуатации.

В случае необходимости модификации программы следует повторить этапы решения и реализации для ее тех частей, которые нуждаются в доработке и изменении.

Совокупность всех трех этапов – решения, реализации и сопровождения – называется *жизненным циклом программы*.

Каждая компьютерная программа – это реализация определенного алгоритма. *Алгоритмом* называют устное или письменное описание логической последовательности действий. Алгоритм отражает последовательность действий, которые могли быть выполнены вручную.

Разработав решение, программист должен проверить его, выполняя каждый шаг самостоятельно. Если алгоритм не работает или работа приводит к неверному результату, следует вновь проанализировать задачу и искать новые пути ее решения. После получения правильного алгоритма программист переводит его на язык программирования (кодирует программу).

Язык программирования – набор правил, символов и специальных слов, используемых для построения программ.

Любой язык программирования представляет собой сильно упрощенную форму естественного языка (как правило, английского), дополненную математическими символами. За счет ограничения словаря и строгих грамматических правил язык программирования управляет компьютером. Ограниченность и однозначность конструкций языка программирования вынуждает программистов создавать простые и точные инструкции.

Процесс перевода алгоритма на язык программирования называется *кодированием* (*программированием*). Его результатом является код программы (или просто программа), которая проверяется *запуском* (или *выполнением*) на компьютере. Программа, не выдающая желаемых результатов, нуждается в *отладке*.

Отладка программы поиск причины ее ошибочного поведения и изменение фрагментов программы (или даже самого алгоритма) для устранения этой причины.

Реализацией алгоритма называется его совместное кодирование и тестирование. Если не затрачено достаточное время на обдумывание и разработку алгоритма, отладка и исправление программы может потребовать много дополнительных усилий.

Наряду с решением задачи, реализацией алгоритма и сопровождением программы важной частью процесса программирования является *разработка документации.* Документацией называют текст и комментарии, которые упрощают программу для понимания, использования и изменения.

Кодирование и запуск программы

Компьютер выполняет инструкции встроенной системы элементарных команд – машинного языка, которой состоит из команд в двоичном представлении.

Языки программирования высокого уровня позволяют создавать компьютерные программы на языке, близком к естественному языку. Их легче использовать, чем машинный код. Примерами языков программирования являются Pascal, C++, Pyton, Java и др.

Специальная программа, называемая *компилятором*, переводит программу, написанную на языке высокого уровня в машинный код.

Программа, написанная на языке высокого уровня, называется *исходной программой* и может быть выполнена на любом компьютере в соответствии с используемым компилятором. Это возможно, поскольку большинство языков высокого уровня стандартизировано.

Компилятор транслирует исходную программу в программу на машинном языке, которая называется *объектной программой*. Если программа содержит ошибки, компилятор создает *листинг* программы – текст с сообщениями об ошибках и другой полезной информацией.

Объектная программа отправляется на *выполнение*. Необходимо разделять понятия компиляции и выполнения программы. Во время компиляции запускается компилятор, создающий объектную программу. При выполнении объектная программа загружается в память компьютера, который выполняет команды в со-

ответствии с исходной программой.

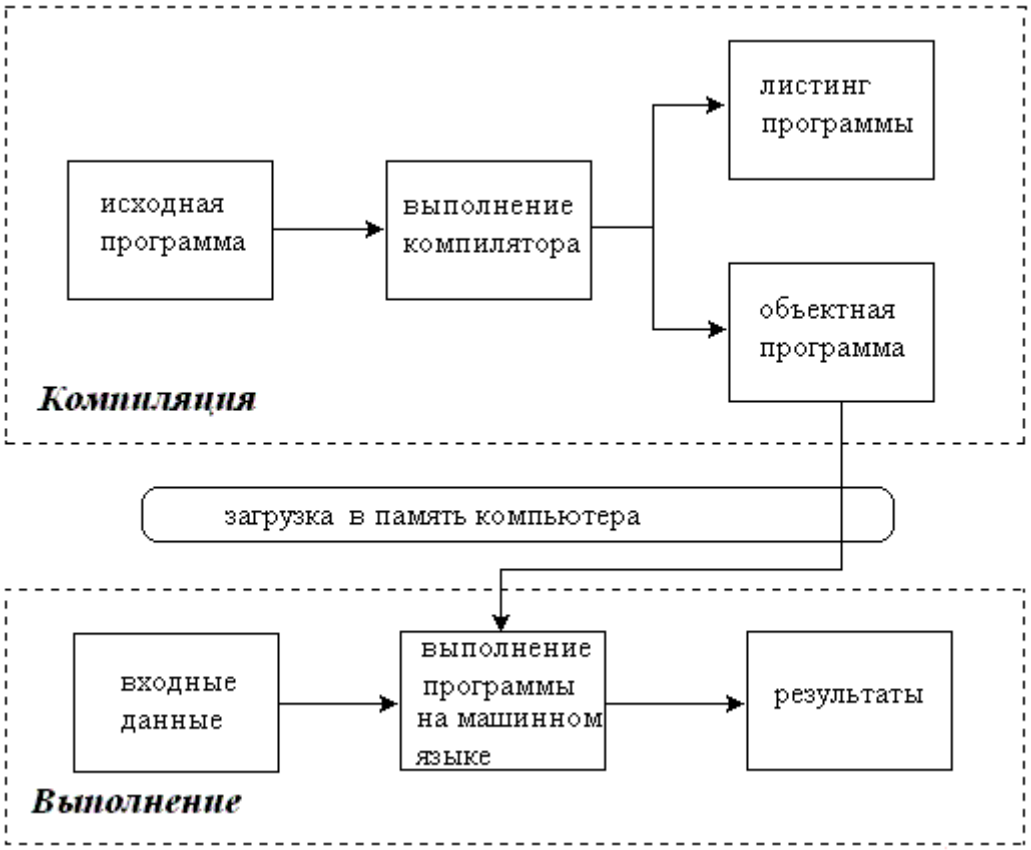


Рисунок 1. Компиляция и выполнение программы

Существуют синтаксические (грамматические) и семантические (смысловые) правила создания программы.

Синтаксис – формальный набор правил, определяющий, какие комбинации цифр и букв могут использоваться в языке программирования. Синтаксис однозначно определяет правила построения конструкций языка. При наличии синтаксических ошибок программа не будет скомпилирована.

Семантика определяет значения команд языка.

Контрольные вопросы

1. Перечислите этапы создания программы.
2. Требуется ли повторение этапов решения программы после ее модификации?
3. Встроенная в компьютер система элементарных команд называется ма-

шинным языком или языком программирования?

4. Что отражается в алгоритме?

5. В чем заключается реализация алгоритма?

6. Компьютер запускает на выполнение объектную или исходную программу?

7. В каких случаях компилятором создается листинг программы?

8. В чем состоит синтаксис языка программирования?

9. Какие цели преследует разработка документации программы?

Глава 2. БАЗОВЫЕ КОНСТРУКЦИИ ЯЗЫКА С++

Язык С++ является потомком языка программирования Си. Несмотря на свою более чем сорокалетнюю историю развития язык программирования С++, по-прежнему, относят к числу наиболее распространенных и популярных. Он по-прежнему широко применяется для разработки программного обеспечения, причем не только для разнообразных пользовательских приложений, но и для операционных систем, драйверов различных устройств. Кроме того, он оказал огромное влияние на развитие других языков программирования, таких как C#, Java, JavaScript, поэтому переключаться между этими языками не сложно.

Язык С++ поддерживает различные парадигмы программирования – полный набор структурного и объектно-ориентированного программирования: модульность, блочную структуру программ, раздельную компиляцию, характерные для языков высокого уровня. С другой стороны, он имеет ряд низкоуровневых черт, в частности, операции над битами, что позволяет программировать микроконтроллеры и драйверы.

Он хорошо подходит как для начинающих программистов, так и для профессионалов. Изучив его структуру, как основы правописания, начинающий программист с легкостью справится с освоением любого другого языка программирования.

В его основу положено значительно меньше синтаксических правил, чем у других языков программирования. Язык предоставляет программисту свободу выбора альтернативы во множестве решений одной проблемы.

Язык С++ содержит небольшое число встроенных функций. Например, в С++ нет встроенных функций ввода-вывода, отсутствуют встроенные математические функции. Взамен этого предоставляется доступ к разнообразным библиотекам, включающим все перечисленные функции и многое другое. Обращение к библиотечным функциям происходят столь часто, что эти функции можно считать составной частью языка. Но в то же время их можно легко переписать, без ущерба для структуры языка. Благодаря небольшому размеру исполняемых моду-

лей программы, написанные на C/C++, отличаются высокой эффективностью и соизмеримы по скорости работы с ассемблерными программами. Большинство компиляторов C++ позволяет обращаться к подпрограммам, написанным на ассемблере.

Для C++ создано множество легко подключаемых библиотек и разнообразие компиляторов.

Еще одно преимущество языка C++ - высокая скорость выполнения программ.

Язык C++ постоянно развивается. За последнее десятилетие выпущены четыре новых спецификации. Последняя из них: C++20 – это название стандарта ISO/IEC языка программирования C++ – опубликована в декабре 2020 года.

К слабым сторонам C++ можно отнести слабый контроль за типом данных и выход за границы массива.

Язык C++ поддерживает различные направления парадигм программирования.

В *процедурном* программировании основное внимание уделяется алгоритму, а именно его эффективности и компактности. Методы процедурного программирования особенно были важны, когда компьютеры не обладали достаточными быстродействием и объемом памяти. Но нельзя сказать, что они утратили свою актуальность сегодня.

В *структурном* программировании основное внимание уделяется организации данных. Программы делятся на модули таким образом, чтобы данные внутри модулей были скрыты. Применение методов структурного программирования позволяет создавать сложные программные продукты коллективам программистов. Поскольку каждый модуль может быть разработан, скомпилирован и отлажен отдельно. Методы структурного программирования позволяют создавать программы, удовлетворяющие критерию надежности и простые в сопровождении.

В действительности перечисленные направления не исключают, а дополняют друг друга.

Объектно-ориентированное программирование в большей степени, чем

структурное программирование, предоставляет возможность создавать программы, обладающие структурированностью, модульностью и абстракциями данных, является современным методом создания сложных программ, в которых недостаточно использования методов структурного программирования.

Состав языка

Любой естественный язык содержит четыре основных элемента: символы, слова, словосочетания, предложения. Алгоритмический язык также включает символы, на основании которых строятся элементарные конструкции (*лексемы*). Из лексем и символов строятся *выражения*, которые в свою очередь образуют *операторы*.

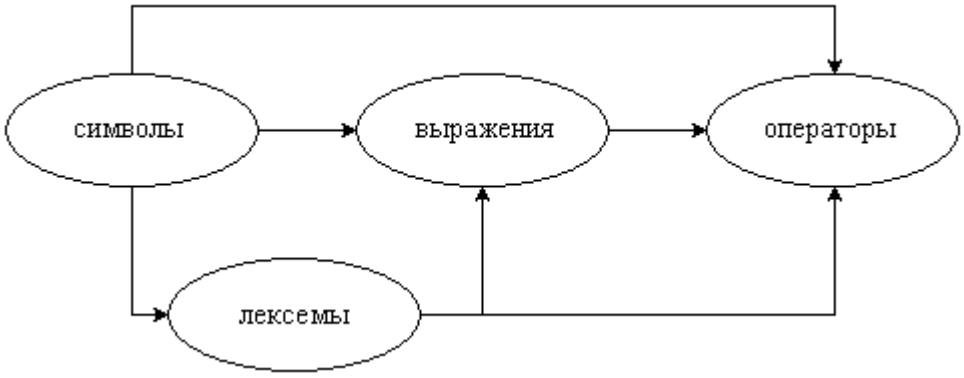


Рисунок 2. Состав алгоритмического языка

Алфавит языка включает в себя основные неделимые знаки, с помощью которых пишутся все тексты программ. Лексема является минимальной единицей языка, имеющей самостоятельный смысл. *Выражение* задает правило вычисления некоторого значения. *Оператор* представляет собой законченное описание некоторого действия. Любое выражение, заканчивающееся точкой с запятой, является оператором, выполнение которого заключается в вычислении выражения.

Операторы бывают исполняемые и неисполняемые. Первые задают действия над данными, а вторые служат для описания данных и называются операторами описания или просто описаниями.

Для описания сложного действия требуется последовательность операторов. Операторы могут объединяться в сложный оператор или блок.

Объединенная единым алгоритмом совокупность описаний и операторов образуют программу на алгоритмическом языке.

Алфавит языка

Алфавит языка программирования С++ включает в себя:

- прописные и строчные буквы латинские буквы и знак подчеркивания;
- арабские цифры от 0 до 9;
- специальные знаки: “, { } , | [ ] ( ) + - / % * . \ ‘ ^ ? < = > ! & # ~ $;
- пробельные символы: пробел, табуляция, символы перехода на новую строку.

Из символов алфавита формируются *лексемы* языка. Лексемы делятся на идентификаторы, ключевые (зарезервированные) слова, знаки операций, константы, разделители.

Переменные, идентификаторы

Идентификаторы используются в С++ для именования различных объектов. *Идентификатор* – имя, связанное с данными или функцией программы, которое используется для обращения к этому объекту или функции.

Идентификатор представляет собой последовательность символов произвольной длины, содержащую буквы, цифры и символ подчеркивания, но начинающуюся обязательно с буквы или символа подчеркивания. Прописные и строчные буквы различаются. Пробелы внутри имен не допускаются.

Длина идентификатора по стандарту не ограничена, но некоторые компиляторы и компоновщики накладывают ограничения, например, распознают только первые 31 символ. В именах нельзя использовать термины, являющиеся частью языка С++.

Ключевые слова – зарезервированные идентификаторы, которые имеют специальное значение для компилятора. Их можно использовать только в том смысле, в котором они определены. Язык С++ содержит 63 ключевых слова.

Переменная – именованная область памяти, в которой хранятся данные

определенного типа. У переменной есть имя (идентификатор) и значение. Имя служит для обращения к области памяти, в которой хранится переменная. Значение переменной может изменяться во время выполнения программы. Прежде чем использовать переменную, ее необходимо определить.

Данные, необходимые для работы программы, хранятся в памяти компьютера. Каждая область памяти имеет однозначно определенный адрес, на который ссылаются, когда необходимо сохранить или прочесть данные.

Идентификаторы используются для обозначения определенной области памяти, а компилятор транслирует имена в соответствующие адреса.

Имя переменной должно отражать смысл хранимой величины, быть легко распознаваемым и не содержать символов, которые можно перепутать друг с другом.

Каждый элемент данных должен принадлежать к определенному типу. Тип данных определяет, в каком виде они представлены в компьютере, а также какие преобразования компьютер может к ним применять.

Типы данных

Тип данных – множество допустимых значений данных и набор операций, применимых к этим значениям.

В C++ определены наиболее часто используемые типы данных. Кроме того, программист может сам определять новые типы.

Для описания стандартных (встроенных) типов в C++ используется набор ключевых слов. Ключевые слова: int, short, long, signed, unsigned, char используются для представления целых данных разной длины (по количеству занимаемых битов в памяти компьютера). Они могут появляться в программе по отдельности или в некоторых сочетаниях.

int обозначает основной целый тип, которому соответствует стандартная длина слова, принятая на используемой машине - 16 битов. Диапазон значений, как правило, зависит от системы. Для многих персональных компьютеров значение типа int меняется от -32768 до +32767.

long или long int может содержать целое значение, не меньшее максимальной величины, допускаемой типом int, или даже больше чем short или short int: максимальное целое число short не больше чем максимальное число типа int, а может и меньше. Обычно числа типа long бывают больше типа short, а тип int реализуется как один из указанных типов, все зависит от конкретной системы (для short отводится 16 бит, а для long - 32 бита). У данного типа также есть синонимы long int, signed long int и signed long

unsigned long представляет целое число в диапазоне от 0 до 4 294 967 295, занимает в памяти 32 бита. Имеет синоним unsigned long int.

long long может содержать целое число в диапазоне от −9 223 372 036 854 775 808 до +9 223 372 036 854 775 807, занимает в памяти, как правило, 64 бита, имеет синонимы long long int, signed long long int и signed long long.

unsigned long long представляет целое число в диапазоне от 0 до 18 446 744 073 709 551 615, занимает в памяти, как правило, 8 байт (64 бита).

Все целые типы имеют 2 формы: знаковую (signed) и незнаковую (unsigned).

Целые незнаковые константы записываются также как и обычные целые, с тем исключением, что использование знака запрещено. Просто unsigned соответствует unsigned int.

Необходимо помнить, что в C++ число, начинающееся с нуля, является восьмеричным, а не десятичным.

char – самое короткое целое. Значения символьного типа занимают только 1 байт. Наиболее часто этот тип применяется для описания данных, состоящих из отдельного алфавитно-цифрового символа. Их называют символьные переменные. Например, ‘a’, ‘1’, ’+’, ’?’,’з’. Они представляют один символ в кодировке ASCII. char занимает в памяти 8 бит и может хранить любое значение из диапазона от -128 до 127, либо от 0 до 255. Имеет синоним unsigned char.

signed char также представляет один символ и занимает в памяти 8 битов, но может хранить любой значение из диапазона от -128 до 127.

wchar_t представляет расширенный символ. На Windows занимает в памяти 2 байта (16 бит), на Linux - 4 байта (32 бита). Может хранить любой значение из

диапазона от 0 до 65 535 (при 2 байтах), либо от 0 до 4 294 967 295 (для 4 байт)

char16_t представляет один символ в кодировке Unicode. Занимает в памяти 2 байта (16 бит). Может хранить любой значение из диапазона от 0 до 65 535.

char32_t представляет один символ в кодировке Unicode. Занимает в памяти 4 байта (32 бита). Может хранить любой значение из диапазона от 0 до 4 294 967 295.

float и double – числа с плавающей запятой или вещественные, которые могут принимать как положительные так и отрицательные значения. Такие числа имеют целую и дробную части, разделенные точкой. Например, 7.9, 3490.725.

long double представляет вещественное число двойной точности с плавающей точкой не менее 64 бит. В зависимости от размера занимаемой памяти может отличаться диапазон допустимых значений.

bool – логический тип. Может принимать одну из двух значений true (истина) и false (ложь). Размер занимаемой памяти для этого типа точно не определен.

Встроенные типы данных языка C++ подразделят на простые, структурированные и адресные (рисунок 3).

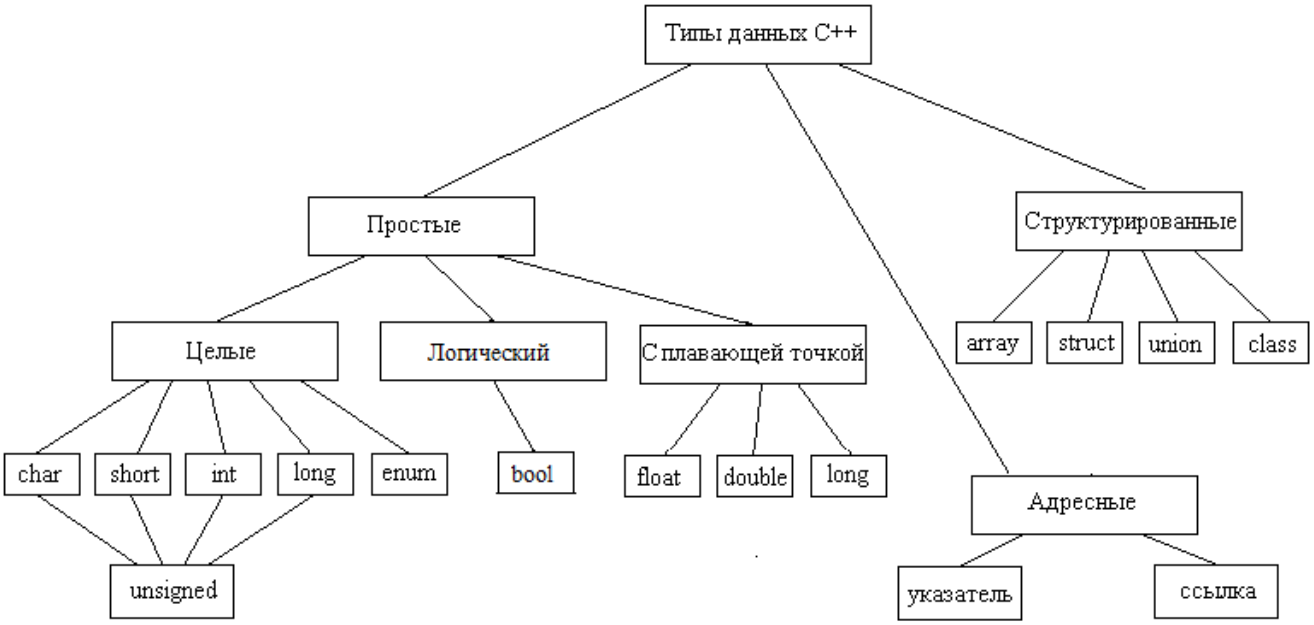


Рисунок 3. Типы данных C++

Структурированные и адресные типы данных, а также перечисляемый тип, описываемый с использованием ключевого слова enum, будут рассмотрены позднее.

Кроме перечисленных, к основным типам данных относится тип void, но множество его значений пусто. Он используется для определения функций, которые не возвращают значения и пустых указателей.

Объявить переменную означает задать ее имя и тип. Объявление сообщает компилятору, что данный идентификатор связывается с областью памяти, содержимое которого имеет определенный тип.

Синтаксис объявления переменной следующий:

имя типа идентификатор переменной;

Например,

```
int k;
```

Объявлена переменная с именем k целого типа. Объявление всегда заканчивается точкой с запятой. Таким образом, переменная k может содержать только целое значение. Если компилятор C++ встретит оператор, в котором переменной k будет присваиваться вещественное значение, то он произведет дополнительные действия для преобразования вещественного типа в целое.

Существует возможность объявить сразу несколько переменных одного типа в одном выражении. Для этого имена переменных перечисляются через запятую. Например,

```
int Number, Count;
float cost1, cost2;
unsigned positive;
```

Определяя переменную можно задать также и ее начальное значение, то есть инициализировать переменную. Инициализатор можно записывать в двух формах – со знаком равенства (=) или в круглых скобках.

Пример:

```
int d, c=5, r=4;
```

```
short b(8);
int k, l(145),  m;
unsigned s=0.5;
```

Таким образом можно собрать в один оператор описания переменные одного и того же типа, или наоборот, разбить одно описание на несколько операторов – эффект будет одинаков.

При инициализации символьных переменных их значение требуется заключать в апострофы.

```
char ch='z',  f;
```

Операции и выражения

Знак операции это один или более символов, определяющих действие над операндами. Операции делятся на унарные, бинарные и тернарную по количеству участвующих в них операндов.

Значение переменной можно изменить с помощью операции присваивания =. Например,

```
int summa = 0;
summa = 5;
```

В отличие от алгебраического уравнения в операторе присваивания сначала вычисляется выражение в правой части оператора, а затем оно присваивается отдельной переменной, стоящей слева от знака равенства. Например,

```
int k = 5, m = 1;
m = k;
```

Это означает, что значение переменной m стало равно 5, а не то, что m равно k. Кроме того, выражения m = k и k = m обозначают различные действия. В первом случае обе переменные станут равными пяти, а во втором – единице.

В C++ допускается использование нескольких присваиваний в одном выражении:

```
a = в = c = 0;
```

В любой программе требуется производить некоторые вычисления. Для вы-

числения значений используются выражения, которые состоят из операндов, знаков операций и скобок. Операнды задают данные для вычислений. Операции задают действия, которые необходимо выполнить.

Основные арифметические операции языка C++ обозначаются стандартными математическими операциями: сложение +, вычитание −, умножение *, деление /. Эти операции, как и операция присваивания, являются бинарными, так как для каждой из них требуется по два аргумента.

Унарный минус используется для определения отрицательных значений, унарный плюс практически не используется, потому что число без знака считается положительным по определению.

Особое внимание требует деление целых чисел. Необходимо помнить, что при делении целых чисел результат операции также целочисленный. Дробная часть просто отбрасывается. Для получения остатка от деления используют одноименную операцию %.

Так как 8 : 2 равно 4, то, 8 / 2 равно 4, 8 % 2 равно 0; 7 : 2 равно 3 (и 1 в остатке), поэтому 7 / 2 равно 3, а 7 % 2 равно 1.

При делении вещественных чисел получается вещественный результат.

Поскольку выражения могут содержать и переменные, допустимо использовать операции присваивания в следующем виде:

```
a = c + 5;
e = a / 2 - 11;
```

После каждого оператора обязательно ставится точка с запятой. Одна и та же переменная может встречаться с обеих сторон знака присваивания.

```
n=n + 7;
```

Это означает, что сначала складываются значение, содержащееся в переменной с именем n и семь, а затем полученное значение помещается в переменную n, тем самым, заменяя ее предыдущее значение.

Кроме стандартных арифметических операций в C++ введен ряд специальных операций. Из них наиболее часто используемыми являются операция инкремента (увеличение на единицу) ++, и операция декремента (уменьшение на еди-

ницу) - - . Использование операция инкремента и дала название языку – С++.

Эти операции унарные (операции с одной переменной). Они могут использоваться с целым и вещественным аргументом. Использование оператора

```
j++;
```

эквивалентно оператору

```
j = j + 1;
```

А соответственно j - - эквивалентно j = j – 1.

У этих операций существует особенность: они имеют две формы: префиксную, когда ее можно поместить перед переменной и постфиксную – после переменной.

Записанные в таком виде j + +; + + j; эти операторы эквивалентны – каждый из них увеличивает значение j на единицу. Поэтому в данном случае выбор одной из форм оператора является делом вкуса. Однако, язык С++ позволяет их использование в середине сложных выражений, и тогда их использование может привести к различным результатам.

Примеры:

```
int bar = 1;
cout << ++ bar;
int bar = 1;
cout << bar ++;
```

В первом случае ++bar увеличивает значение bar на единицу, а затем записывает это значение собственно в bar, то bar будет равен 2, и значение 2 будет выведено на экране.

bar ++ определяет значение bar, а потом выполняет приращение, следовательно, устанавливает bar 2, но выводит на экран 1, поскольку вывод происходит перед выполнением приращение.

В языке С++ определены операции составного присваивания. Они используются для сокращения записи операторов, содержащих в себе присваивание и арифметическую операцию. Оператор вида: операнд1 += операнд2 эквивалентен записи операнд1 = операнд1 + операнд2. Существует составное присваивание со

сложением, вычитанием, делением, умножением, с остатком от деления: += - = ; * = , / =, % =.

Выражение foo + = 3; эквивалентно foo = foo + 3;

C++ содержит операции отношения: больше > , больше или равно > = , меньше < , меньше или равно< = , равно = = , не равно! = , не !.

Также определены логические операции: И && и ИЛИ ||. Результатом логической операции является true (истина) или false (ложь). Результатом операции ЛОГИЧЕСКОЕ И является значение true, если оба операнда имеют значение true. Результат ЛОГИЧЕСКОГО ИЛИ является true, если хотя бы один из операндов имеет значение true. Логические операции выполняются слева направо. Например, результат выражения (k>=1) && (k<10) будет true, если k = 5. И это же выражение есть false, если k = 0.

Операция определения размера sizeof( ) предназначена для вычисления размера объекта или типа в байтах, и имеет 2 формы: sizeof (выражение) или sizeof(тип). Так результат выполнения sizeof(int) равен 2, так как для хранения данных этого типа в памяти выделяется 2 байта.

C++ имеет одну тернарную операцию, обозначаемую ? : .

Формат тернарной операции:

условие ? выражение1 : выражение2;

Данный оператор позволяет создавать простые условные однострочные выражения, в которых выполняется одно из двух действий в зависимости от значения условия. Данный оператор можно использовать вместо инструкции if/else. Рассмотрим пример, в котором определяется модуль числа с помощью условного оператора

```
fvalue = (fvalue>=0.0) ? fvalue : -fvalue;
```

Рассмотрим другой пример применения условной операции. Требуется, чтобы некоторая величина увеличилась на 1, если она не превышает n, иначе принимала бы значение 1:

```
y = (y < n) ? y + 1 : 1;
```

Операции выполняются в соответствии с приоритетом, который задает очередность выполнения в равнозначных ситуациях. Для изменения порядка выполнения операций в выражениях, содержащих несколько операций, используются круглые скобки.

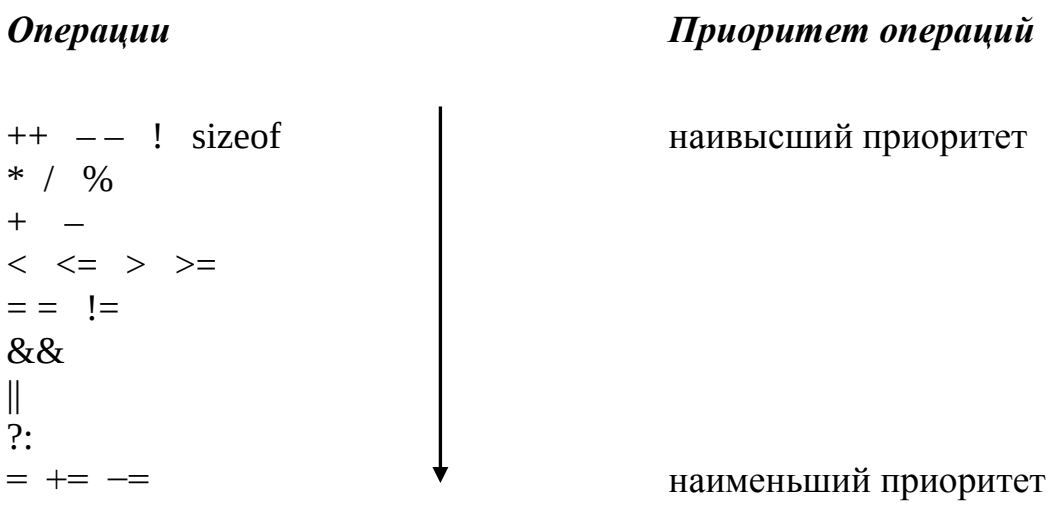


Рисунок 4. Приоритеты операций

Выражения состоят из операндов, знаков операций и скобок и используются для вычисления некоторого значения определенного типа. Каждый операнд является, в свою очередь, выражением или одним из его частных случаев – константой или переменной.

Примеры выражений:

(d + 8) / 67

x && y || !z

(t + 5*k) / (f – 56) + 470

Если в одном выражении записано несколько операций одинакового приоритета, унарные операции, условная операция и операция присваивания выполняются *справа налево*, остальные – *слева направо*. Например, выражение a = c = y означает a = (c = y), а выражение a + b + c означает (a + b) + c.

Константы

Иногда требуется, чтобы значение переменной оставалось постоянным в течении всего времени работы программы. Такие переменные называются *кон-*

стантными.

Для их создания необходимо написать определение для переменной с добавлением ключевого слова const перед типом.

```
const int Top = 12;
```

Константные переменные используются в программе также как обычные. Единственное отличие заключается в том, что начальные значения, присвоенные константам при их инициализации, не могут быть изменены в ходе выполнения программы.

С помощью констант можно передавать значение других констант или переменных.

```
int a = 10;
const int b = 7;
const int d = b;
const int x = a;
```

В C++ существует и другой способ описания констант, доставшийся в наследство от языка C – с помощью макроопределений использующих директиву препроцессора #define. Например,

```
#define TOP 12
//объявлена константа с именем TOP, равная 12
```

Точка с запятой после директивы препроцессора не ставится.

Каждая директива #define позволяет определить одну константу, имя которой следует за директивой. Принято писать имя константы заглавными буквами, что не является требованием компилятора. В отличие от предыдущего способа тип такой константы неизвестен. Процессор при трансляции программы просто заменяет имя константы на значение, определенное с помощью директивы.

Данная директива в языке C использовалась также для определения макросов (макроопределение с аргументом). Макросы являются просто текстовыми подстановками и могут не давать компилятору достаточной информации в желательном представлении данной величины. Часто встречаемой ошибкой была, например, такая

```
#define SUMMA( a, b )   a + b
double result, x = 5.2, y = 0.8,
result = SUMMA( x, y )*10;
```

После работы препроцессора получим подстановку вида:

```
result = x + y*10;
```

Для корректной подстановки рассмотренная директива должна выглядеть следующим образом:

```
#define SUMMA( a, b ) ((a)+(b))
```

Комментарии

Комментариями называются некомпилируемые фрагменты программы. Комментарий используется для пояснения алгоритма или текста программы. Комментарий либо начинается с двух символов «косая черта» («слэш») // и заканчивается переходом на новую строку, либо заключается между символами-скобками /* и */. Внутри комментария можно использовать любые допустимые на компьютере символы, а не только символы алфавита языка С++, так как компилятор комментарии игнорирует.

Ввод – вывод на экран. Введение в потоки ввода – вывода

Оператор cout (си-аут) позволяет осуществлять вывод данных на экран монитора. Переменная cout зарезервирована для обозначения выходного потока.

Для того чтобы послать значение на си-аут применяют последовательность cout<< (оператор «направить в» или оператор вставки).

Если необходимо напечатать строку символов, требуется взять ее в кавычки. Можно также напечатать несколько значений одновременно, разделяя их оператором вставки.

```
cout << “My name is” <<”Tatyana”;
```

При печати цифр их можно не помещать в кавычки. Возможно объединение текста и цифр.

```
cout <<  “ мой адрес: Институтская“ << 26;
```

В процессе печати можно использовать довольно большое количество специальных символов. Вот некоторые из них:

\n Начало новой строки

\t табулятор

\b возврат назад на один пробел

\f начало новой строки страницы

\\ печать символа обратный слэш

\’ печать символа’

\” печать символа ’’

Пример:

```
cout << “ He said:\ ” Hello:\” \ n”;
```

Оператор вставки использует два аргумента. Аргумент слева от << является потоковым выражением (потоковой переменной). Правый аргумент представляет собой строку или выражение, результат которого имеет простой тип. Оператор вставки преобразует правый операнд в последовательность символов и добавляет их выходной поток. Например,

```
cout<<”Результат равен  ” << 5*n + 90;
```

Если n равно 10, то на экране появится:

Результат равен 140

Для перехода на новую строку используют манипулятор endl, который также очищает буфер потока. Например,

```
int x=17, y=21;
cout << “x= ”<<x<<endl<<”y=”<<y;
```

На экране появится

x=17

y=21

Стандартная библиотека C++ предоставляет пользователю большое количество манипуляторов, позволяющих форматировать ввод-вывод. Манипулятор setw (сокращение от «set width» – «установить ширину») позволяет управлять количеством позиций для вывода следующего за манипулятором элемента данных. При-

меняется только для форматирования чисел и строк, но не данных типа char. Параметр данного манипулятора – целое выражение, определенное число знаковых позиций для вывода очередного элемента. Данные при выводе выравниваются по правому краю, а свободные позиции слева заполняются пробелами. Например,

```
int ans=33, num=7132;
cout<<setw(4)<<ans<<setw(5)<<num;
```

выведет на экран □□33□7132

```
cout<<setw(1)<<ans<<setw(6)<<num;
```

33□□7132 - поле автоматически расширяется, чтобы вместить двузначное число.

Установка ширины поля является одноразовым действием и влияет только на ближайший элемент вывода.

Контроль числа десятичных позиций при выводе решается с помощью манипулятора setprecision, указывающего количество знаков после запятой. Например,

```
int x=4.856;
cout<<setw(6)<<setprecision(2)<<x;
```

вывод □□□4.85

Для ввода с клавиатуры используют символ cin (си-ин или син) и >> оператор ,,взять из“

```
cin >> Number;
```

Стандартные функции ввода-вывода языка С, также доступны и в С++. Наиболее часто используемыми функциями ввода-вывода языка С, являются printf( ) и scanf( ), которые обеспечивают форматный ввод-вывод и являются достаточно универсальными, особенно при работе с числами, но из-за обилия всевозможных спецификаторов форматирования, становятся иногда громоздкими и трудно читаемыми. Операторы << и >> благодаря понятию перегрузки операторов поддерживают все стандартные типы языка С++, включая классы.

Директива #include и пространство имен

Язык C++ содержит очень небольшое число встроенных функций, но он легко расширяется дополнительными библиотеками.

Операторы cin и cout также являются частью библиотеки. Для их использования необходимо включить в программу соответствующие заголовочные файлы. Это делается с помощью команды #include.

Любая команда, начинающая с решетки называется директивой препроцессора. Она не является выражением языка C++ (поэтому не заканчивается точкой с запятой). Директивы препроцессора могут состоять только из одной строки.

Препроцессор – это программа, действующая как фильтр на этапе компиляции. Так препроцессорная директива #include приказывает компилятору загрузить включаемый файл.

Описания операторов cin и cout находятся в файле с именем iostream(input output stream – поток ввода-вывода).

Синтаксис: имя файла помещается в угловые скобки < >, что указывает препроцессору, что этот файл ищется в стандартном каталоге подключаемых файлов:

```
#include <iostream>
```

Компилятор знает, где искать стандартные заголовочные файлы. Если же требуется загрузить заголовочный файл, созданный пользователем, его имя необходимо заключить в кавычки.

```
#include " myfile.h"
```

Можно также указать полный путь

```
#include " c \ My \ myfile.h "
```

Для использования манипуляторов при выводе данных также необходимо подключить библиотеку – iomanip.

Пространство имен – декларативная область, в рамках которой определяются различные идентификаторы (имена типов, функций, переменных, и т. д.). Пространства имен используются для организации кода в виде логических групп и с целью избегания конфликтов имен, которые могут возникнуть, например, при подключении нескольких библиотек. Все идентификаторы в пределах простран-

ства имен доступны друг другу без уточнения. Идентификаторы за пределами пространства имен могут обращаться к членам с помощью полного имени для каждого идентификатора.

Использование оператор using позволяет использовать все имена в пространстве namespace для использования без имени пространства имен в качестве явного квалификатора. Использование директивы using в файле реализации.

Например, программа вида

```
#include <iostream>
using namespace std;
int main()
{
   int age;
   cout << "Input age: ";
   cin >> age;
   cout << "Your age: " << age << endl;
   return 0;
}
```

будет выглядеть несколько иначе без оператора using:

```
#include <iostream>
 int main()
{
   int age;
   std::cout << "Input age: ";
   std::cin >> age;
   std::cout << "Your age: " << age << std::endl;
   return 0;
}
```

Префикс указывает, что объекты cout, cin, endl определены в пространстве имен std. А само двойное двоеточие представляет оператор области видимости, который позволяет указать, в каком пространстве имен определен объект.

Построение программы

При создании алгоритма решения сложной задачи возможно разбиение решения на решение более простых подзадач. Подпрограммы позволяют сначала записать части программы по отдельности, а затем собрать их в единое работоспособное целое. В языке С++ подпрограммы называются *функциями*, а программа представляет собой набор из одной или более функций. Каждая функция выполняет определенное действие, а все вместе они решают задачу в целом.

Итак, любая программа, написанная на языке С++ есть последовательность выполнения функций, причем одна из них обязательно должна называться main( ). Выполнение программы всегда начинается с выполнения ряда операторов этой функции. Когда main () хочет, чтобы другая функция выполнила свое задание, она вызывает (активизирует) необходимую функцию.

Все функции в языке С++ равноправны: каждая из них (даже main( )) может быть любой другой функцией. Функция может вызывать саму себя (явление рекурсии). Компилятор не ограничивает число рекурсивных вызовов, но операционная система может наложить чисто практические ограничения.

Описание функции состоит из заголовка и тела, и в общем случае, выглядит следующим образом:

Директива препроцессора	#include <iostream> using namespace std;	
Заголовок	тип результата имя функции (список аргументов)	
Тело функции	{	
	Описание	*пять типов операторов*
	Присваивание	
	Функция	

Управление

Пустой оператор

}

Предполагается, что любая часть описание функции может не использоваться, кроме имени функции. Пример простейшей программы:

```
#include <iostream>
int main ( )
{
 cout << “HeLLo!!! ”;
return 0;
}
```

Данная программа просто выведет в консоль HeLLo!!!

Фигурные скобки отмечают начало и конец тела функции. В круглых скобках в общем случае содержится информация, передаваемая этой функции.

Если тип результата не указывается, то предполагается, что функция возвращает значение типа int. Если функция не возвращает результат, указывается слово void. Некоторые компиляторы требуют обязательного указания типа возвращаемого результата перед именем функции.

Наличие списка аргументов и описаний аргументов также не является обязательным. Но круглые скобки всегда должны присутствовать после имени функции. Аргументы функции – это величины, которые необходимо передать из вызывающей функции в вызываемую функцию.

Часто употребляется описание типов аргументов сразу при их объявлении внутри круглых скобок. Тип аргумента может быть любым. Если аргументов несколько, их описания (тип плюс имя) разделяются запятыми. Если функции не передаются величины, то вместо списка аргументов необходимо задавать ключевое слово void. Локальные переменные отличные от аргументов, описываются внутри тела функции.

Возвращает значение оператор return, с помощью которого можно передать в вызывающую функцию только одно значение. Возвращаемое значение можно

брать в круглые скобки после ключевого слова return. Круглые скобки не являются обязательными.

Пример

Напишем программу, содержащую три функции: непосредственно main и функции, вычисляющие значения куба и квадрата некоторого целого числа.

```
#include <iostream>
using namespace std;
int square (int x)
   { return x*x; }
int cube (int y)
    { return y*y*y; }
int main( )
    {  cout<< “Квадрат числа 15 равен”<<square(15)<<endl;
        cout<< “Куб числа 10 равен”<<square(10)<<endl;
       return 0;
     }
```

В каждой из трех функций левая и правая фигурные скобки указывают на начало и окончание тела функции, т.е. начало и окончание исполняемых выражений. Их называют операторные скобки.

В первой строке расположена директива препроцессора, подключающая заголовочный файл iostream, необходимый для работы оператора cout.

Далее следует заголовок и тело функции square. Функция имеет один аргумент целого типа, описание которого расположено в круглых скобках после имени функции. Функция square возвращает целочисленный результат в вызываемую функцию, на это указывает ключевое слово int, стоящее перед именем функции. Возвращение результата из функции осуществляет оператор return. Результатом является выражение после указанного оператора. После оператора обязательно ставится точка с запятой (как требует синтаксис языка С++).

Затем в программе расположено описание функции cube, аналогичное предыдущей функции.

Ниже идет описание функции main. Согласно стандарта языка C++ main должна возвращать значение типа int. Если по задумке программиста этого не происходит можно воспользоваться оператором return и вернуть нуль. Однако это не является обязательным. При отсутствии последнего компилятор ошибку не выдаст. Функция не имеет аргументов – в ее заголовке пустые круглые скобки. В скобках также можно указать void,что также будет указывать на отсутствие аргументов. Выполнение любой программы всегда начинается с первого оператора функции main, имеющего в данной программе вид:

cout<< “квадрат числа 15 равен”<<square(15)<<endl;

В этом операторе происходит вызов функции square для аргумента 15, следовательно, выполняется оператор данной функции, т.е. 15 умножается на 15 и результат возвращается в main и при помощи cout выводится на экран в виде:

Квадрат числа 15 равен 225

После этого main продолжает выполнение своих операторов, напечатав сообщение

Куб числа 10 равен 100

На этом программа завершает свое выполнение.

Любая функции в C++ может возвращать не более одного значения, т.е. либо не возвращать ничего, либо возвращать единственное значение. Аргументов же у функции может быть сколько угодно, в том числе ни одного.

Пример. Опишем функцию, возвращающую среднее значение трех величин.

```
long Average (long val1, long val2, long val3 )
{
long sum = val 1 + val 2 + val 3;
return sum / 3;
}
```

Эта функция может быть вызвана любой другой. Пусть в программе объявлены некоторые переменные: long a, a1, a2, a3; тогда возможен вызов вида

```
a = Average (a1, a2, a3);
```

или функция может быть вызвана для произвольных значений типа long:

```
a = Average (525, 675, 819);
```

Найденное значение будет присвоено переменной a.

Вызов функции, возвращаемой значение, возможен также в операторе вывода.

```
int k, l, m;
cout<< “введите 3 числа”;
cin>>k>>l>>m;
cout<< “среднее арифметическое чисел” << k << “ “ << l << “ “<< m;
cout<< “ равно “ << Average (k, l, m) << endl;
```

Но в этом случае найденное значение не будет храниться в памяти компьютера и будет утеряно после вывода в консоль. Такой прием используется, если значение в дальнейшем не понадобится.

В случаях, когда вызываемая функция не возвращает значение, ее вызов представляет просто имя функции, а в круглых скобках – фактические значения аргументов функции.

В C++ для определения встраиваемой функции используется ключевое слово inline. Определение такой функции в программе на C++ будет выглядеть так:

```
inline double SUMMA(double a, double b)
{
return (a+b);
 }
```

При определении и использовании встраиваемой функции надо придерживаться следующих правил:

- определение и объявление функции должны располагаться перед первым ее вызовом;
- имеет смысл определять таким способом только маленькие функции;
- компилятор сам решает, является ли данная функция встраиваемой, при этом он руководствуется размером (до 1200 строк), если функция рекурсивна, то встраиваемой является только первый вызов, некоторые компиляторы не допускают использования во встраиваемых функциях операторы цикла и некоторые

другие.

Библиотечные функции

Некоторые вычисления, такие как извлечение квадратного корня или нахождение модуля, часто используются в программах. Для удобства программиста любой программный комплекс C++ содержит *стандартную библиотеку* (или *библиотеку стандартных функций*) – собрание ранее написанных функций, выполняющих стандартные вычисления.

Библиотека математических функций содержит:

Имя функции	*Возвращаемое значение*
acos(x)	Арккосинус x, в диапазоне от 0.0 до π
asin(x)	Арккосинус x, в диапазоне от -π/2 до π/2
atan(x)	Арктангенс x, в диапазоне от -π/2 до π/2
ceil(x)	Верхнее значение x (наименьшее целое число ≥ x)
cos(x)	Косинус x (x выражается в радианах)
exp(x)	Значение e (2.718…), возведенное в степень x
fabs(x)	Модуль x
floor (x)	Нижнее значение x (наибольшее целое число ≤ x)
log(x)	Натуральный логарифм от x
log10(x)	Десятичный логарифм от x
pow(x, y)	Значение x, возведенное в степень y
sin(x)	Синус x (x выражается в радианах)
sqrt(x)	Корень квадратный из x (для x > 0)
tan(x)	Тангенс x

Параметрами и результатами перечисленных математических функций являются переменные типа float.

Использовать библиотечную функцию несложно. Требуется разместить в начале программы директиву #include с указанием требуемого файла заголовков math. Затем можно обращаться к функции в любом месте программы.

Пример:

```
#include <iostream>
#include <math.h>
using namespace std;
int main ( )
{       float alpfa, beta;
        alpfa = sqrt(169.41 + fabs( pow( beta, 5));
        cout<< alpfa; }
```

Контрольные вопросы

1. Перечислите символы алфавита языка С++.
2. Как называется минимальная единица алгоритмического языка, имеющая самостоятельный смысл?
3. Что такое идентификатор?
4. Перечислите ключевые слова, используемые для обозначения целых типов данных? В чем их отличие?
5. Можно ли в программе изменять значение константы?
6. С какими операциями можно использовать составное присваивание?
7. Содержит ли С++ встроенные операции ввода-вывода?
8. Какой файл заголовков требуется подключить для использования манипуляторов, форматирующих вывод в операторе cout?
9. Ставится ли точка с запятой после директивы препроцессора? Почему?
10. С каких операторов начинается выполнение программы?
11. Все ли функции программы на языке С++ равноправны?
12. Что означает ключевое слово void перед именем функции?
13. Сколько значений функция может возвращать в программу? Каким образом?
14. Является ли обязательным наличие списка аргументов функции?
15. Ограничивается ли количество аргументов функции?

Глава 3. МЕТОДЫ СТРУКТУРНОГО ПРОГРАММИРОВАНИЯ

Блоки или составные выражения

Любое выражение, оканчивающееся точкой с запятой, является оператором. Оператор также может ничего не содержать – так называемый пустой оператор. Пустой оператор обозначается просто точкой с запятой (так как после каждого оператора, том числе и пустого, по требованию синтаксиса языка C++ обязательно ставится точка с запятой). Любой оператор может представлять собой объявление, выполняемый оператор или блок.

Блок можно использовать везде, где разрешено использование отдельного оператора. После окончания блока точка с запятой не ставится.

Блоки часто используются в программах, особенно как составная часть других выражений. Пропуск пары фигурных скобок может кардинально поменять смысл алгоритма программы, повлиять на ее выполнение и полученный результат.

В программе операторы блока обычно располагаются с некоторым отступом, что не является требованием компилятора. Но это позволяет выделять блок и контролировать расположение скобок. Это повышает читаемость программы, и следовательно, контроль ошибок.

Внутри блока разрешается чередовать объявления и операторы, т.к. объявления в программе C++ могут располагаться в любом месте. Необходимо только помнить, что все переменные должны быть объявлены перед использованием.

Объявление переменных, выполненное внутри блока, распространяет свое действие только на сам блок.

```
#include <iostream>
using namespace std;
int main()
{
        int x = 5;
        {
```

```
            int x = 3;
            cout << x+1 << endl;
        }
        cout << x+1 << endl;
        return 0;
}
```

В функции main() объявлены две переменные с именем x. Но второе объявление распространяется только на внутренний блок. Поэтому результатом оператора вывода в первом случае будет 4, а во втором – 6.

Тело функции также является примером блока или составного оператора. Блок является последовательностью из нуля или более операторов. Эта последовательность заключена в фигурные скобки.

Базовые конструкции структурного программирования

Программу для решения задачи любой сложности можно составить только из трех структур, называемых следованием, ветвлением и циклом.

Следование, ветвление и цикл называют *базовыми конструкциями* структурного программирования (рисунок 5).

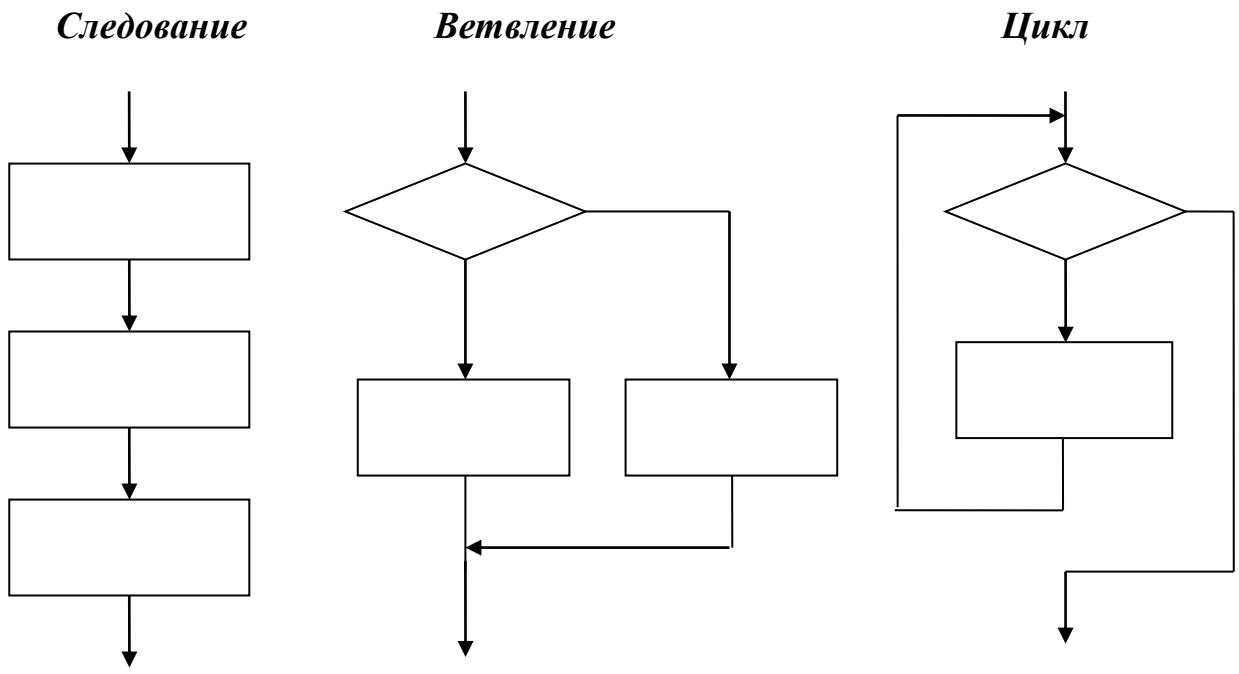


Рисунок 5. Базовые конструкции структурного программирования

Следование представляет собой последовательное выполнение нескольких операторов (простых или составных).

Ветвление задает выполнение одного из операторов в зависимости от результата выполнения какого-либо выражения, задающего условие.

Цикл задает многократное выполнение одного или последовательности нескольких операторов.

Любая базовая конструкция содержит только один вход и один выход.

Конструкции могут вкладываться друг в друга произвольным образом, тем самым, образуя структуру программы. Например, цикл может содержать следование двух ветвлений, каждое из которых содержит последовательное выполнение циклов.

В большинстве языков высокого уровня существует несколько реализаций базовых конструкций. Язык C++ содержит три вида циклов и два вида ветвлений. Их выбор осуществляется из требований алгоритма и его эффективности.

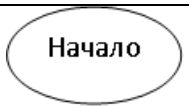
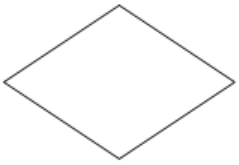
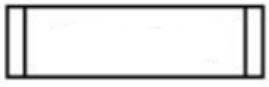
Использование блок-схем в разработки алгоритмов

Блок-схема – регламентируемый распространённый тип схем (графических моделей), описывающих алгоритмы или процессы, в которых отдельные шаги изображаются в виде блоков различной формы, соединённых между собой линиями, указывающими направление последовательности.

На территории Российской Федерации действует единая система программной документации (ЕСПД). К ней относится Государственный стандарт — ГОСТ 19.701-90 «Схемы алгоритмов программ, данных и систем», в котором отражены основные элементы блок-схем, который близок к международному стандарту ISO 5807:1985.

Таблица. Элементы блок-схем

Изображение элемента	**Описание**
	Начало обработки данных. Любой алгоритм имеет только один такой блок.

	Блок имеет единственный выход.
	Окончание алгоритма. Любой алгоритм имеет только один такой блок. Блок имеет единственный выход.
	Блок процесс. Выполнение операции или группы операций, в результате которых изменяется значение, форма или представления данных.
	Блок ввода-вывода. Ввод данных с клавиатуры, или отображение их на экране монитора.
	Блок решение. Выбор направления алгоритма происходит в зависимости от условия, размещенного внутри блока. Блок имеет один вход и не более двух выходов. Как правило, вход в блок располагается сверху. Ветви выхода помечаются словами «да» и «нет» в зависимости от значения условия внутри блока.
	Соединитель. Если блок-схема не помещается на странице, указывается данный символ, отражающий переход на следующую страницу. Тогда следующая страница начинается с такого же символа.
	Вызов внешних функций.

При соединении графических символов используются только горизонтальные или вертикальные линии. Потоки, направленные справа налево и снизу вверх,

обязательно помечаются стрелками.

Пересечение линий потоков в блок-схеме быть не должно, так как, как правило, это говорит об ошибке в алгоритме.

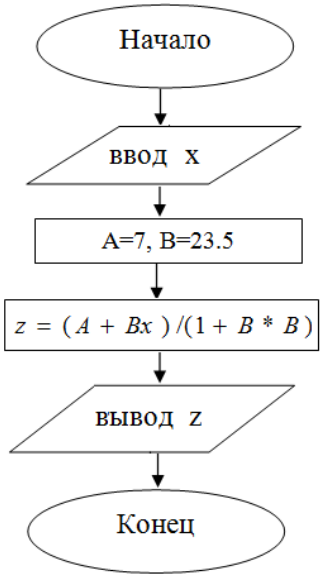


Рисунок 6. Блок-схема линейного алгоритма

На рисунке 6 представлена блок-схема алгоритма вычисления значения переменной z в зависимости от введенного с клавиатуры значения переменной x.

Операторы ветвления

Условный оператор if

Условный оператор if позволяет разветвлять вычислительный процесс на два направления. Структурная схема оператора представлена на рисунке.

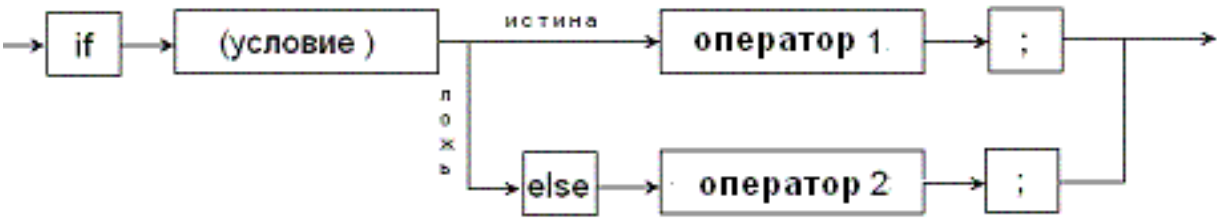


Рисунок 7. Структурная схема условного оператора

Сначала вычисляется выражение, стоящее в условие. Если оно не равно нулю или имеет значение истина, выполняется первый оператор, иначе второй. По-

сле этого управление передается следующему оператору. Таким образом, с помощью оператора if можно в ходе выполнения программы задать некоторый вопрос и в зависимости от ответа (да или нет) выполнить те или иные действия.

Синтаксис оператора if:

if (выражение) оператор1; else оператор 2;

Ветвь с ключевым словом else не является обязательной. Поэтому она взята в квадратные скобки. (В дальнейшем изложении для обозначения необязательных элементов будут всегда использоваться квадратные скобки.)

Примеры:

```
if ( fvalue>=0.0 )  fvalue = fvalue;  else fvalue = -fvalue;   // вычисляется модуль числа
if ( x<10 ) x+ =10; else x *=2;
if ( f != 0 )  c = 100 / f;
```

Если в какой-либо ветви требуется выполнить несколько операторов, их необходимо заключить в блок (операторные скобки { }), иначе компилятор не сможет определить окончание ветвления.

```
if ( a ) { a++;  v=60*a; }  else v = a;
if ( e==1000 ) { e /=10;  cout << "e= " << e; }  else { e =10+y*y;  y++; }
```

При использовании блока в условном операторе точка с запятой после правой фигурной скобки блока не ставится. Точка с запятой применяется для завершения простых операторов.

Блок может содержать любые операторы, в том числе и другие условные.

```
if ( a<b ) { if ( a<c ) m = a;  else m = c; }  else { if ( b<c ) m = b;  else m = c; }
```

Необходимо помнить, что в этом случае else относится к ближайшему из if. Операторные скобки после первого if необязательны, т.к. в него вложен простой оператор if.

Если требуется проверить несколько условий, их объединяют знакам логических операций.

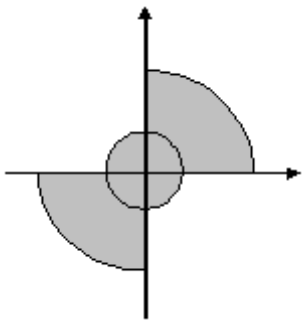
```
 if ( a<b && ( a>d || a==0 ) )  b++; else { b*=a; a=0; }
```

Записанное условие будет истинно в том случае, если выполнится одновременно условие a<b и одно из условий в скобках. Если опустить внутренние скоб-

ки, будет выполнено сначала логическое И, а потом ИЛИ.

Распространенной ошибкой является неверная запись условия проверки переменной на принадлежность диапазону. Например, чтобы проверить условие 0<x<10 нельзя записать его в условном операторе непосредственно, следует писать if (0< x && x>10).

Пример программы. Производится выстрел по мишени, изображенной на рисунке. Радиус внутреннего круга равен единице, а внешнего – тройке. Попадание в меньший круг дает 10 очков, в сегменты большого – 5 очков. Определить количество очков, набранного выстрелом, координаты которого вводятся с клавиатуры.



```
#include <iostream>
using namespace std;
int main( )
{
setlocale(LC_ALL, " ");
 float x, y;
  int score;
  cout << “введите координаты выстрела”<<endl;
  cin >> x >> y;
  if  ( x*x + y*y < 1 ) score = 10;
        else    if  ( x*x + y*y < 9 &&  x*y > 0 ) score = 5;
                  else kol = 0;
  cout << “Вы набрали” << kol << “  очков!!!”;
}
```

Выбор из множества альтернативных действий может быть запрограммирован с помощью множества условных операторов. Например, чтобы запрограммировать вывод названия месяца по его заданному порядковому номеру, допустимо использовать последовательность условных операторов без вложения:

```
if (m ==1) cout << “Январь”;
```

```
if (m ==2) cout << “Февраль”;
if (m ==3) cout << “Март”;
…
```

Однако эквивалентная вложенная структура более эффективна, поскольку требует меньшее количество сравнений:

```
if (m ==1) cout << “Январь”;
    else    if (m ==2) cout << “Февраль”;
      else   if (m ==3) cout << “Март”;
        else   if (m ==4) cout << “Апрель”;
          else   if (m == 5) cout << “Май”;
            else  if (m ==6) cout << “Июнь”;
              else if (m ==7) cout << “Июль”;
                else  if (m ==8) cout << “Август”;
                  else  if (m ==9) cout << “Сентябрь”;
                    else  if (m ==10) cout << “Октябрь”;
                      else  if (m ==11) cout << “Ноябрь”;
                        else  if (m ==12) cout << “Декабрь”;
```

В первом случае последовательно проверяется все двенадцать условий, а во втором – все сравнения прекращаются после того, как истинное условие обнаружено.

Более правильным будет добавить в конце ветвь else, которая предназначена для случая неверного ввода номера:

```
else cout<< “Неверно введенный номер”;
```

Распространенной ошибкой при записи условного оператора является использование операции присваивания (=) вместо проверки на равенство (==).

Оператор выбора switch

Оператор switch (переключатель) предназначен для разветвления процесса вычислений на несколько направлений.

Синтаксис оператора switch:

switch (выражение) {

case константное выражение 1 : операторы1; [break;]

case константное выражение 2 : операторы2 ; [break;]

…

case константное выражение n : операторы n; [break;]

[default : операторы;]

}

Структурная схема оператора приведена на рисунке 8.

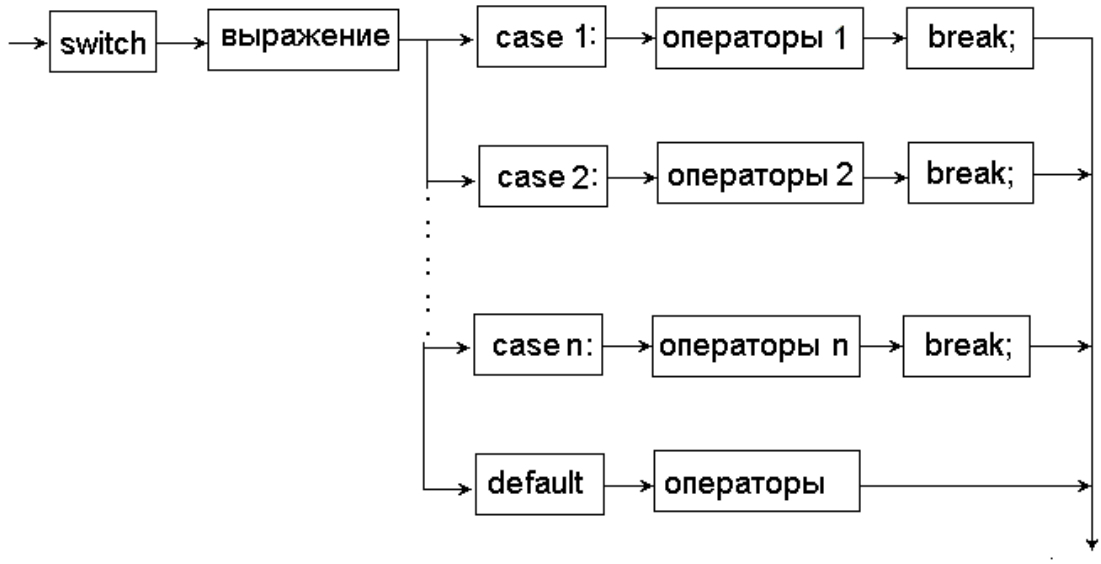


Рисунок 8. Структурная схема оператора switch

Выполнение оператора начинается с вычисления выражения, которое должно быть целочисленным. Затем управление передается первому оператору из списка, помеченному константным выражением, значение которого совпало с вычисленным.

После этого, если выход из переключателя явно не указан (отсутствует break), последовательно выполняются все нижележащие ветви.

Выход из переключателя обычно выполняется с помощью оператора break или return. Оператор break выполняет выход из самого внутреннего из объемлющих его операторов. А оператор return выполняет выход из функции, в которой он описан.

Все константные выражения, расположенные после case, должны быть раз-

личны. Если совпадения ни с одним оператором не произошло, выполняются операторы, расположенные после ключевого слова default.

Ветвь default может отсутствовать. В этом случае выполнение программы передается следующему за switch оператору.

Пример предыдущий раздела (печать названия месяца по порядковому номеру) может быть переписан с использованием switch следующим образом:

```
#include <iostream>
using namespace std;
int main ( )
{ setlocale(LC_ALL, " ");
  int x;
  cout << “Введите номер месяца” << endl;
  cin >> x;
  switch (x ) {
        case 1: cout << “Январь”<< endl;          break;
        case 2: cout << “Февраль”<< endl;         break;
        case 3: cout << “Март”<< endl;            break;
        case 4: cout << “Апрель” ”<< endl;        break;
        case 5: cout << “Май” ”<< endl;           break;
        case 6: cout << “Июнь” ”<< endl;          break;
        case 7: cout << “Июль” ”<< endl;          break;
        case 8: cout << “Август” ”<< endl;        break;
        case 9: cout << “Сентябрь” ”<< endl;      break;
        case 10: cout << “Октябрь” ”<< endl;      break;
        case 11: cout << “Ноябрь” ”<< endl;       break;
        case 12: cout << “Декабрь”<< endl;        break;
        default: cout << “Неверный номер” << endl;      }
return 0;
}
```

Несколько меток могут следовать подряд, предполагая выполнение одина-

ковых действий. Программа вывода времени года по номеру месяца:

```
#include <iostream>
using namespace std;
int main ( )
{ setlocale(LC_ALL, " ");
  int x;
  cout << “Введите номер месяца” << endl;
  cin >> x;
  switch (x ) {
        case 12:  case 1:   case 2:      cout << “зима”<< endl;    break;
        case 3:   case 4:   case 5:      cout << “весна”<< endl;   break;
        case 6:   case 7:   case 8:      cout << “лето”<< endl;    break;
        case 9:   case 10:  case 11:     cout << “осень”<< endl;   break;
        default: cout << “Неверный номер” << endl; }
return 0;
}
```

Инструкции перехода

В языке C++ имеется четыре инструкции перехода: goto, break, continue, return.

Оператор безусловного перехода goto имеет формат:

goto метка;

Тогда в теле той же функции должна присутствовать ровно одна конструкция вида:

метка: оператор;

Метка является обычным идентификатором. Оператор goto передает управление на помеченный оператор.

Наличие конструкции безусловного перехода рассматривается как плохой стиль программирования, и считается, что в четко структурированной и грамотно написанной программе этой конструкции быть не должно.

Использование данного оператора оправдано в случаях:

- принудительного выхода вниз по тексту программы из нескольких вложенных циклов или переключателей;

- перехода из нескольких операторов функции на один.

Инструкция break используется внутри циклов или переключателей и позволяет переход в точку программы, находящуюся непосредственно за оператором, внутри которого находится. Таким образом, break обеспечивает выод из цикла еще до того, как условие цикла станет ложным. По своему действию она напоминает команду goto, только в данном случае не указывается точный адрес перехода. Управление передается первой строке, следующей за телом оператора.

Инструкция continue заставляет программу пропустить все оставшиеся строки цикла, но сам цикл при этом не завершается. Для решения некоторых задач удобно комбинировать инструкции break и continue.

Иногда необходимо прервать выполнение программы задолго до того, как будут выполнены все ее строки. Для этого используют return, которая завершает выполнение той функции, в которой она была вызвана. Если же вызов произошел в функции main( ), то завершается сама программа.

Операторы цикла

Операторы цикла используются для организации многократно повторяющихся вычислений. Любой цикл состоит из тела цикла, т. е. операторов, которые повторяются несколько раз, начальных установок, модификации параметра цикла и проверки условия продолжения выполнения цикла.

Один повтор выполнения операторов тела цикла называется *итерацией*. Проверка условия выполнения цикла производится на каждой итерации.

Параметрами цикла называются переменные, изменяющиеся в теле цикла и используемые при проверке условия продолжения цикла, называются параметрами цикла.

Начальные установки могут явно не присутствовать в программе, их смысл состоит в том, чтобы до входа в цикл задать значения переменным, которые в нем

используются.

Цикл завершается, если условие его выполнения не выполняется. Возможно принудительное завершение, как текущей итерации, так и тела цикла. Для этого используются конструкции перехода.

Для удобства в C++ существуют три разных оператора цикла – while, do while, for.

Цикл с предусловием (while)

В цикле с предусловием проверка условия продолжения цикла выполняется перед телом цикла (рисунок 9).

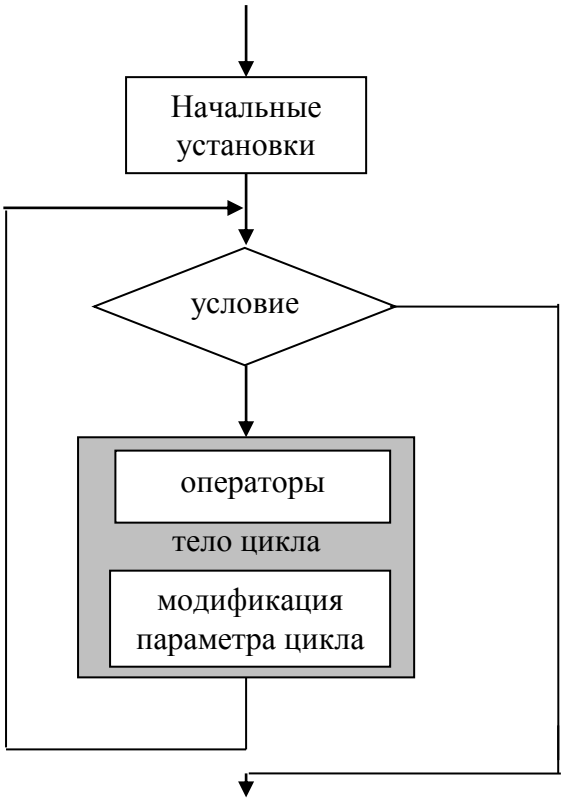


Рисунок 9. Цикл с предусловием

Цикл с предусловием реализован в C++ оператором цикла while, структурная схема которого представлена на рисунке 10.

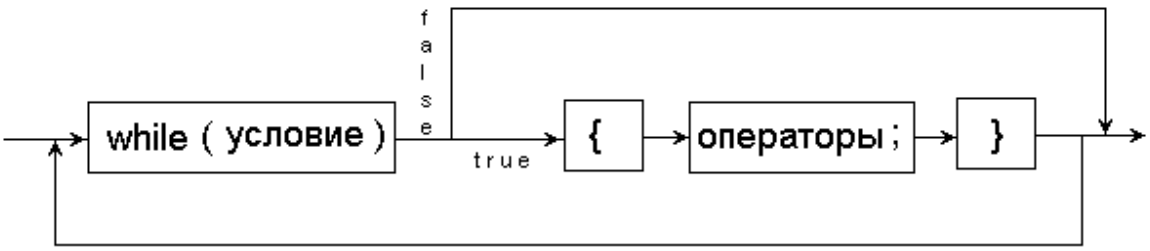


Рисунок 10. Структурная схема оператора цикла while

Выражение, стоящее в круглых скобках, определяет условие повторения тела цикла, представленного простым или составным оператором. Если оператор простой операторные скобки { } не ставятся.

Выполнение оператора цикла начинается с вычисления выражения, стоящего в условии. Если оно истинно, выполняется тело цикла. Если при первой проверке выражение ложно (false), цикл не выполнится ни разу.

Тип выражения должен быть арифметическим или приводимым к нему. Выражение вычисляется перед каждой итерацией цикла.

```
#include <iostream>
using namespace std;
int main( )
{ setlocale(LC_ALL, " ");
  int i = 0;
   int sum = 0;
    while (i < 1000)
   {
      i++;
      sum += i;
    }
   cout << "Сумма чисел от 1 до 1000 = " << sum << endl;
   return 0;
}
```

Важным при вычислении суммы в цикле является задание начального зна-

чения. Сумма должна быть равна нулю, чтобы при первой итерации значение переменной было определено.

Для вычисления произведения его начальное значение доцикла должно быть равно 1.

Пример. Программа печатает таблицу значений функции $y = x^3 - x$ с определенным шагом во вводимом диапазоне.

```
#include <iostream>
#include <iomanip.h>
int main ( )
{ setlocale(LC_ALL, " ");
  float x1, xn, d;
   cout << " введите диапазон и шаг изменения x"<< endl;
   cin >> x1 >> xn >> d;
   cout << "|        x        |        y       |" << endl;     // шапка таблицы
   float x = x1;
   while ( x <=  xn )
         { cout << "|" << setw(6) << setprecision(3) << x << "|";
           cout << "|" << setw(6) << setprecision(3)<< x*x*x – x << "|" << endl;
           x+=d;
         }
return 0;
}
```

В этом примере параметром цикла является переменная x, ее начальное значение задано до начала цикла.

Цикл while обычно используется в тех случаях, когда число повторений заранее неизвестно.

Цикл с постусловием (do while)

Тело цикла с постусловием всегда выполняется хотя бы один раз, после чего проверяется, надо ли его выполнять еще раз. Цикл с постусловием реализован в

языке C++ оператором цикла do … while и имеет вид:

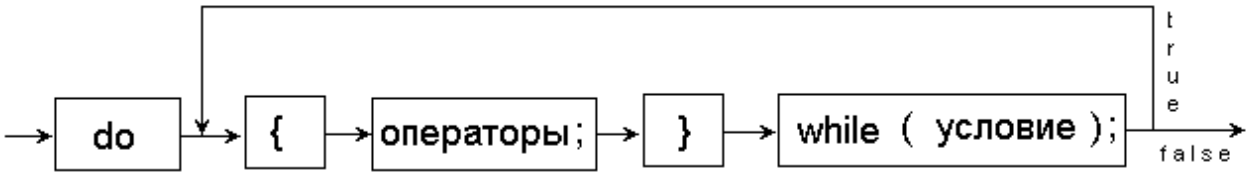


Рисунок 11. Структурная схема оператора цикла do while

Сначала выполняется простой или составной оператор, составляющий тело цикла, а затем вычисляется выражение, составляющее условие выполнение цикла. Даже если условие заведомо ложно, цикл выполнится один раз. Если условие истинно, тело цикло выполнится еще раз. Цикл завершается, когда выражение станет равным false, или в теле цикла будет выполнен оператор передачи управления.

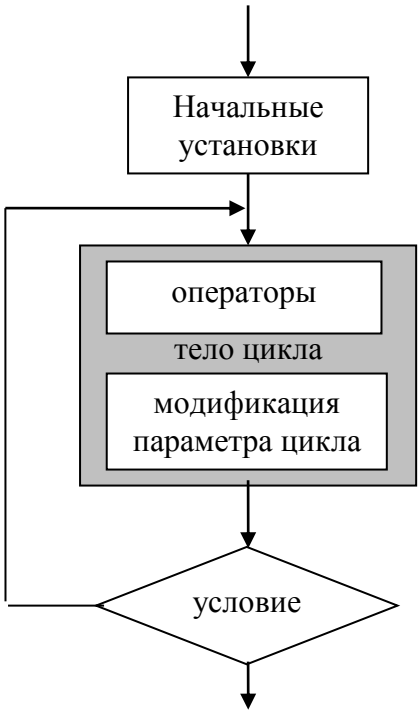


Рисунок 12. Цикл с постусловием

Данный цикл гарантирует выполнение тела хотя бы один раз. Например,

```
int i = -1;
do
{
```

```
   cout << i << endl;
    i --;
}
while(i > 0);
}
```

Вывод значения -1 произойдет, несмотря на заведомо ложное условие, которое проверится позднее.

Цикл с параметром (for)

Цикл for называют также циклом с заданным числом повторений. Он имеет следующий формат:

for (инициализация; выражение (условие) ; модификации) оператор;

Инициализация используется для объявления и присвоения начальных значений величинам, используемым в цикле. Инициализация выполняется один раз перед выполнением тела цикла.

Цикл с параметром реализуется как цикл предусловием. Выражение определяет условие выполнения цикла: если его результат равен истине, то цикл выполняется. Модификации выполняются после каждой итерации цикла и служат обычно для изменения параметра цикла.

Тело цикла представляет собой простой или составной оператор.

Любое из трех выражений, указанных в скобках, является необязательным. Точки с запятой должны всегда оставаться на своих местах, даже в случае, если все три выражения отсутствуют.

Примеры

```
for ( ; ; );        // пример бесконечного цикла
for ( y = 2; y < 20; y++ )  r += y;
```

В инициализации и модификации параметра можно писать несколько операторов, разделенных запятой.

```
for ( i = 1, s = 1; i < 11; i++ ) s *= i; // вычисления факториала 10
```

Цикл for обычно используется в тех случаях, когда можно точно определить

необходимое число повторов.

Допускается объявление переменной прямо в строке инициализации цикла for. Значение счетчика цикла может не только увеличиваться, но и уменьшаться, причем на произвольный шаг, который может быть не только целым, но и числом с плавающей точкой.

```
for ( float k = 11.5; k > 0.5; k - = 0. 5 ) s += k;
```

В качестве параметра цикла можно использовать и символьную переменную.

```
for (char ch=’a’; ch< ‘z’; ch++) …
```

Если тело цикла содержит более одной команды, следует использовать фигурные скобки и руководствоваться определенными правилами оформления, чтобы сделать текст более наглядным.

```
#include <iostream>
using namespace std;
int main( )
{
   int result = 0;
   for (int i=0; i<10; i++)
   {
      if (i % 2 == 0) continue;
      result +=i;
   }
   cout << "result = " << result << endl;              // 25
   return 0;
}
```

Любой цикл while может быть приведен к эквивалентному ему циклу for и наоборот. Операторы цикла взаимозаменяемы, но можно привести рекомендации по выбору наилучшего в каждом конкретном случае.

Оператор do while обычно используют, когда цикл требует обязательно выполнить хотя бы раз (например, если в цикле производится ввод данных).

Оператором while удобнее пользоваться в случаях, когда число итераций заранее неизвестно, нет очевидных параметров цикла или модификацию параметров удобнее записывать не в конце тела цикла.

В большинстве остальных случаев предпочтительнее использование оператора for.

Часто встречающиеся ошибки при проектировании циклов – использование в теле цикла неинициализированных переменных или неверная запись условия выполнения цикла. Во избежание ошибок рекомендуется:

- проверить, всем ли переменным, встречающимся в правой части операторов присваивания в теле цикла, присвоены до этого начальные значения (а также возможно ли выполнение других операторов);

- проверить, изменяется ли в цикле хотя бы одна переменная, входящая в условие выхода из цикла;

- предусмотреть аварийный выход из цикла по достижению некоторого количества итераций;

- не забывать о том, что если в теле цикла требуется выполнить более одного оператора, их нужно заключать в фигурные скобки.

Если телом цикла является циклическая структура, то такие циклы называются вложенными или сложными.

При работе с вложенными циклами необходимо обращать внимание на правильную расстановку фигурных скобок, чтобы четко разделить границы циклов.

Цикл, содержащий в себе другой цикл, называется внешним. Цикл, содержащий в теле другого цикла, называется внутренним.

Внутренний и внешний циклы могут быть любыми из трех рассмотренных видов: циклом с параметрами, циклом с предусловием, циклом с постусловием. Правила организации как внешнего, так и внутреннего цикла, такие же, как и для простого цикла каждого из этих видов. Однако при построении вложенных циклов необходимо соблюдать следующее дополнительное условие: все операторы внутреннего цикла должны полностью лежать в теле внешнего цикла.

Некоторые структуры вложенных циклов представлены на рисунке 13.

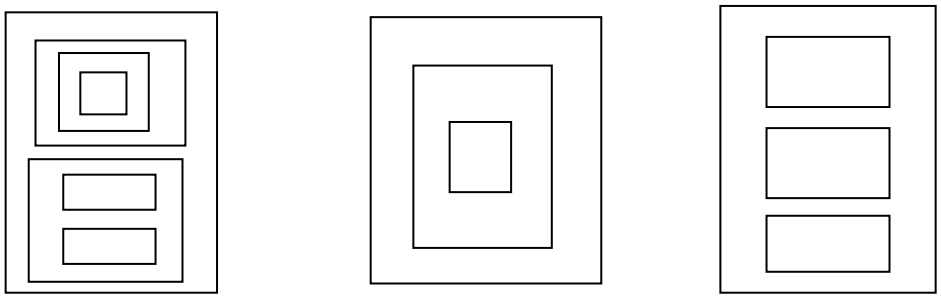


Рисунок 13. Структуры вложенных циклов

Параметры циклов разных уровней не изменяются одновременно. Сначала все свои значения изменит параметр цикла низшего уровня вложенности при фиксированных (начальных значениях) параметров циклов с высшим уровнем. Затем изменяется на один шаг значение параметра цикла следующего уровня, и снова полностью выполняется самый внутренний цикл, и т.д. до тех пор, пока параметры циклов не примут все требуемые значения.

Генератор псевдослучайных чисел

Возможность генерировать случайные числа часто бывает полезной в программах научного или статистического моделирования, а также в игровых приложениях.

Генератор псевдослучайных чисел – это программа, которая принимает начальное(стартовое) значение и выполняет с ним определенные математические операции, позволяющие конвертировать его в другое число, которое совсем не связано с первоначальным. Затем используется новое сгенерированное значение и выполняет с ним те же математические операции, что и с начальным числом, чтобы получить третье, которое не связано ни с первым, ни со вторым. Таким образом, программа может генерировать целый ряд новых чисел, которые будут казаться случайными.

Язык C++ имеет свой собственный встроенный генератор случайных чисел. Функция srand() устанавливает передаваемое пользователем значение в качестве стартового. Ее следует вызывать только один раз – в начале программы, как правило, в верхней части функции main( ). Функция srand( ) находится в заголовочном

файле cstdlib.

Однако при многократном запуске этой функции можно заметить, что в результатах всегда находятся одни и те же числа. Для исправления этого недостатка нужен способ выбора стартового числа, которое не будет фиксированным значением. Общепринятым решением является использование системных часов.

Функция time() возвращает в качестве времени общее количество секунд, прошедшее от полуночи 1 января 1970 года. Чтобы использовать эту функцию, нужно подключить заголовочный файл ctime, а затем инициализировать функцию srand() вызовом функции time(0).

```
#include <iostream>
#include <ctime>
#include <cstdlib>
using namespace std;
int main()
{
srand(time(NULL));
int n;
 n = rand( )%1000;
cout<< n;
return 0;
}
```

Выведется случайное число из промежутка от 0 до 1000. Для получения целого числа из симметричного промежутка можно воспользовавшись выражением вида rand( )%k – k/2, где k – целое число.

Для получения вещественного значения можно воспользоваться умножением полученного значения на другое вещественное. Например, выражение вида: n = (rand( )%1000 – 500) * 0.1 задаст значение из диапазона от -50.0 до +50.0.

Рассмотрим программу для угадывания загаданного числа.

```
#include <iostream>
#include <ctime>
```

```
#include <cstdlib>
#include <locale.h>
using namespace std;
int main( )
{
srand( time( NULL ) );
float x, y;
    y = rand( )%10;            // выбор числа в диапазоне от 0 до 10
  do
  {  cout << " введите произвольное число меньшее 10" << endl;
     cin >> x;
     if ( x == y) { cout <<" вы угадали!";  break; }
     else       if  ( x< y ) cout<< "введите меньшее"<<endl;
                else    cout<< "введите большее"<<endl;
    }  while (x != y);
return 0;
}
```

Для использования системной константы NULL требуется подключить библиотеку cstdlib.h.

Массивы

При использовании простых переменных каждой области памяти для хранения данных соответствует свое имя. Если с группой переменных одного типа требуется выполнить различные действия, им дают одно имя и отличают по порядковому номеру. Это позволяет записывать множество действий и операций над этими элементами через циклы.

Конечная именованная последовательность однотипных величин называется массивом.

Для создания массива компилятору необходимо знать тип данных, количество элементов в массиве и требуемый класс памяти. Массивы могут иметь те же

типы данных, что и простые переменные.

Синтаксис объявления массива:

тип данных имя массива [размерность массива];

Примеры:

```
int array[45];                  //массив из 45 элементов целого типа с именем array
double rt[12];                  // массив rt, состоящий из 12 элементов типа double
const int ARRAY_SIZE=90;
float alp[ARRAY_SIZE];      // вещественный массив alp из 90 элементов
```

Размерность массива предпочтительнее задавать с помощью именованных констант, как это сделано в примере. При таком подходе для изменения во всей программе достаточно изменить значение константы в ее объявлении.

Возможна также инициализация массива при его объявлении. Для этого значения элементов массива перечисляются через запятую в фигурных скобках после имени массива.

```
int a[5] = {4, 90, 71, 45, 3};
```

Количество элементов должно соответствовать размеру массива. Если их окажется меньше, то недостающие элементы будут инициализированы нулями (или мусором, хранящимся в памяти). Если же элементов окажется больше – компилятор выдаст ошибку.

При инициализации допускается использование пустых скобок в объявлении массива, и тогда компилятор сам определяет размерность массива. Например,

```
float x[ ] = {4, 8, 19}; // размерность x равна 3
```

Для доступа к элементу массива после его имени указывается номер элемента (индекс), заключенный в квадратные скобки. Нумерация элементов массива начинается с нуля. Следовательно, диапазон значений для объявленного в примере массива x лежит в пределах от 0 до 2. Следовательно,

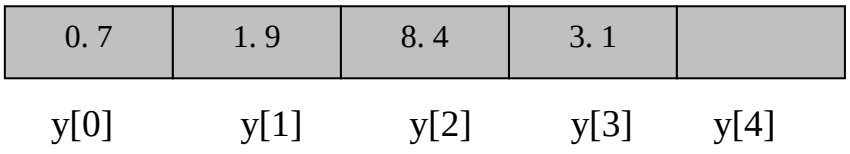
x[0]=4 x[1]=8 x[2]=19

В памяти компьютера элементы массива располагаются последовательно друг за другом. Место в памяти для хранения элементов массива выделяется компилятором сразу после обработки его объявления.

Так объявление вида

```
float y[5 ]= {0.7, 1.9, 8.4, 3.1 };
```

выделит место для расположения пяти элементов типа float. Так как выполнена инициализация четырех первых элементов, они сразу займут свое место в памяти компьютера, а ячейки памяти для пятого элемента пока останутся свободными:



В С++ не осуществляется проверка значений индексов на выход за пределы массива, ни в процессе компиляции, ни в процессе выполнения программы. Так если, в тексте программы напишем выражение

```
y[5] = 2;
```

компьютер сохранит значение 2 в ячейке памяти, следующей за ячейкой, соответствующей последнему элементу массива. При этом старое значение данной ячейки будет затерто. Поэтому проверка значения индекса – обязанность программиста.

К элементам массива можно применять все операции, допустимые для обычной переменной типа, соответствующего типу элементов массива. Можно присваивать ему значения

```
y[0] = 56;
```

применять арифметические операции

```
y[2] = 7 * y[0] + 3;
```

считывать или выводить значения

```
cin >> y[0];    cout << y[1];
```

передавать его в качестве параметров функций

```
x = sqrt (y[1]);
```

Пример: Программа подсчета суммы элементов массива.

```
#include<iostream>
using namespace std;
```

```
int main ( )
 {
  const int n=10;
  int i, m[n], s=0;
  cout << " введите" << n << "чисел" <<endl;
                    // ввод элементов массива с клавиатуры
  for ( i = 0; i < n; i++) cin >> m[ i];
  for ( i = 0; i < n; i++) s+=m[ i ] ;
  cout << "сумма введенных элементов равна "<<s<<endl;
}
```

Для улучшения эффективности алгоритма ввод элементов массива и подсчет их суммы можно объединить в один цикл:

```
  for ( i = 0; i < 10; i++) { s += m[i];    cin >> m[i]; }
```

Пример: Программа выводит на экран количество дней в месяце для не високосного года.

```
#include<iostream>
using namespace std;
int main( )
{
int days[ ]={31,28,31,30,31, 30, 31, 31, 30, 31, 30, 31};
for (int index=0; index < sizeof(days)/(sizeof(int)); index++)
        cout << "месяц" << index+1 << " имеет" << days[index] << "дней \n";
return 0;
}
```

Размерность массива определяется компилятором, а в цикле определение размерности массива происходит с использованием операции sizeof( ).

Обработка одномерных массивов числовых данных

Классическими задачами при работе с одномерными массивами являются нахождение суммы и произведения элементов массива, определение минималь-

ного и максимального элементов массива, сортировка (упорядочивание) элементов.

Пример нахождения суммы элементов массива рассмотрен в предыдущем разделе. Суммирование элементов происходит в цикле, но необходимо помнить, что значение переменной, предназначенной для хранения значения суммы, должно обязательно равно нулю до начала цикла. При подсчете произведения элементов массива значение произведения до цикла должно быть равно единице.

```
#include<iostream>
using namespace std;
int main ( ) {
  const int n=15;
  int i, m[n], p=1;
  cout << " введите" << n << "чисел" <<endl;
 // ввод элементов массива с клавиатуры
  for ( i=0; i < n; i++)
          {
            cin >> m[ i];
           if ( m[ i] > 0 ) p*= m[ i ] ;
          }
  cout << "произведение положительных элементов равно "<<p<<endl;
return 0;
}
```

Стандартный алгоритм нахождения максимального значения в одномерном массиве:

1. Предполагаем, что элемент, расположенный в нулевой позиции, является максимальным.

2 В цикле до конца массива сравниваем каждый следующий элемент с предполагаемым максимальным. Если он больше максимального (обнаруженного на данный момент), то максимальным становится текущий элемент (запоминается его позиция).

Для реализации данного алгоритма нужно не забыть ввести вспомогательную переменную для хранения позиции максимального элемента.

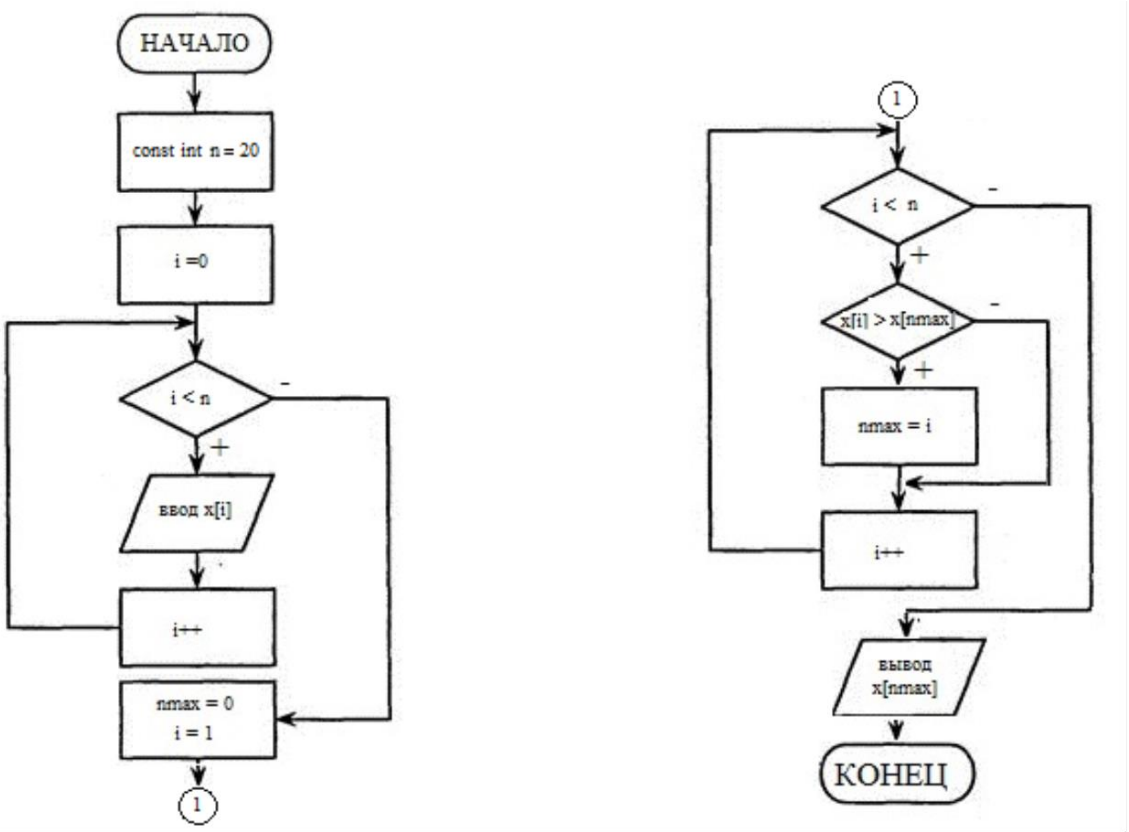


Рисунок 14. Блок-схема алгоритма для нахождения максимального элемента из 20 чисел, введенных с клавиатуры

Программа, реализующая представленный на рисунке 14 алгоритм:

```
#include<iostream>
using namespace std;
int main ( )
{
const int n=20;
int i, X[n], nmax;
for ( i=0; i < n; i++)
     { cout << " введите X[ " << i << " ] ";  cin >> X[ i ]; }
nmax = 0;
for ( i=1; i < n; i++)
```

```
    if ( X[ i ] > X[ nmax ] )   nmax= i ;
cout << ”максимальный элемент массива “<< X[ nmax ] << endl;
return 0;
 }
```

Рассмотренный алгоритм неэффективен, поскольку элементы массива перебираются дважды: при заполнении массива и при непосредственном поиске максимального элемента. При объединении этих действий в один цикл скорость выполнения программы увеличится, а результат выполнения останется прежним.

```
#include<iostream>
using namespace std;
int main ( )
{
const int n=20;
int i, X[n], nmax;
nmax = 0;
for ( i=0; i< n; i++)
    { cout << “ введите X[ ” << i << “ ] ”;  cin >> X[ i ];
       if ( X[ i ] > X[ nmax ] )   nmax= i ;
     }
cout << ”максимальный элемент массива “<< X[ nmax ] << endl;
return 0;
 }
```

Наиболее часто встречающая ошибка при создании такой программы заключается в том, что студенты забывают, что записи вида nmax = i ; и i = nmax; являются не эквивалентными. В первой записи переменной nmax присваивается значение i при этом само значение nmax остается без изменения. Во втором случае наоборот.

Для нахождения минимального элемента нужно в условном операторе знак меньше поменять на знак больше. Максимума (минимума) не среди всех элементов, а среди некоторых. Например, поиск максимального элемента только среди

отрицательных. В этом случае следует предполагать, что за первоначальное значение максимума будет приниматься отрицательный элемент, найденный первым при переборе массива. Блок-схема алгоритма представлена на рисунке 15.

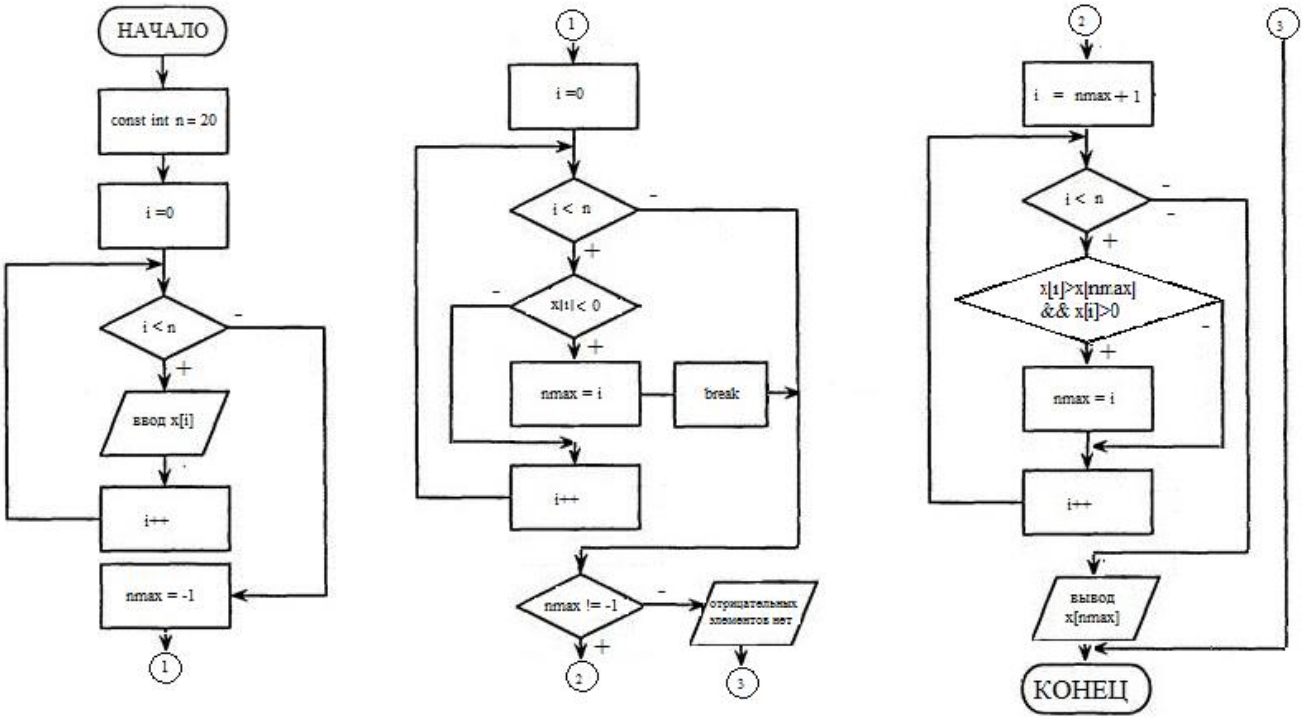


Рисунок 15. Блок-схема алгоритма поиска минимального элемента среди отрицательных

Важным моментом в алгоритме является поиск первого отрицательного элемента. После этого алгоритм сводится к стандартному алгоритму поиска минимума. Кроме того, алгоритм рассматривает ситуацию полного отсутствия отрицательных элементов в массиве.

```
#include<iostream>
using namespace std;
int main ( )
{
const int n = 20;
int i, X[n],  nmax;
for ( i = 0; i< n;  i++)
     { cout << " введите X[ " << i << " ] ";  cin >> X[ i ]; }
```

```
nmax = -1;
if ( nmax != -1)
   {
     for ( i = 0;  i< n;  i++)     if ( X[ i ] < 0)   { nmax= i ; break;}
     for ( i =nmax+ 1; i<n; i ++)  if ( X[ i ] > X[ nmax ] && X[i]< 0 )  nmax= i;
     cout << ”максимальный элемент массива среди отрицательных“ <<
      X[nmax ]  << “ его номер” << nmax << endl;
    }
  else   cout<< “отрицательных  элементов в массиве нет” << endl;
return 0;
 }
```

При разработке программного обеспечения распространенной операцией является сортировка значений, т.е. расположение списка в некотором порядке (например, слов по алфавиту или чисел в возрастающем или убывающем порядке). Один из способов сортировки – пузырьковый. В нем при каждом проходе (переборе элементов) попарно сравниваются рядом стоящие элементы, и если второй больше первого (при сортировке по убыванию) элементы меняются местами. На рисунке 16 представлены результаты выполнения шагов сортировки.

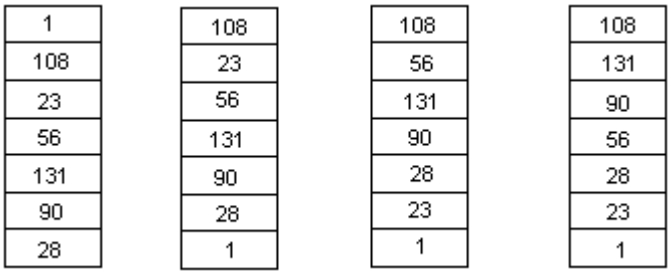


Рисунок 16. Сортировка «пузырьком»

Для данного случая массив из 7 элементов полностью упорядочился за 3 шага. Однако это зависит от расположения значений в массиве. Доказано, что количество проходов (шагов сортировки) для гарантированного успеха должно быть на единицу меньше количества элементов в массиве.

На рисунке 17 представлена блок-схема алгоритма пузырьковой сортиров-

ки. Обратите внимание, что перебор элементов заканчивается предпоследним из списка, поскольку последним просто не с чем будет сравнивать. В языке C++ не отслеживается выход за границы массива. Поэтому включение во внутренний цикл последнего элемента приведет к неверному результату.

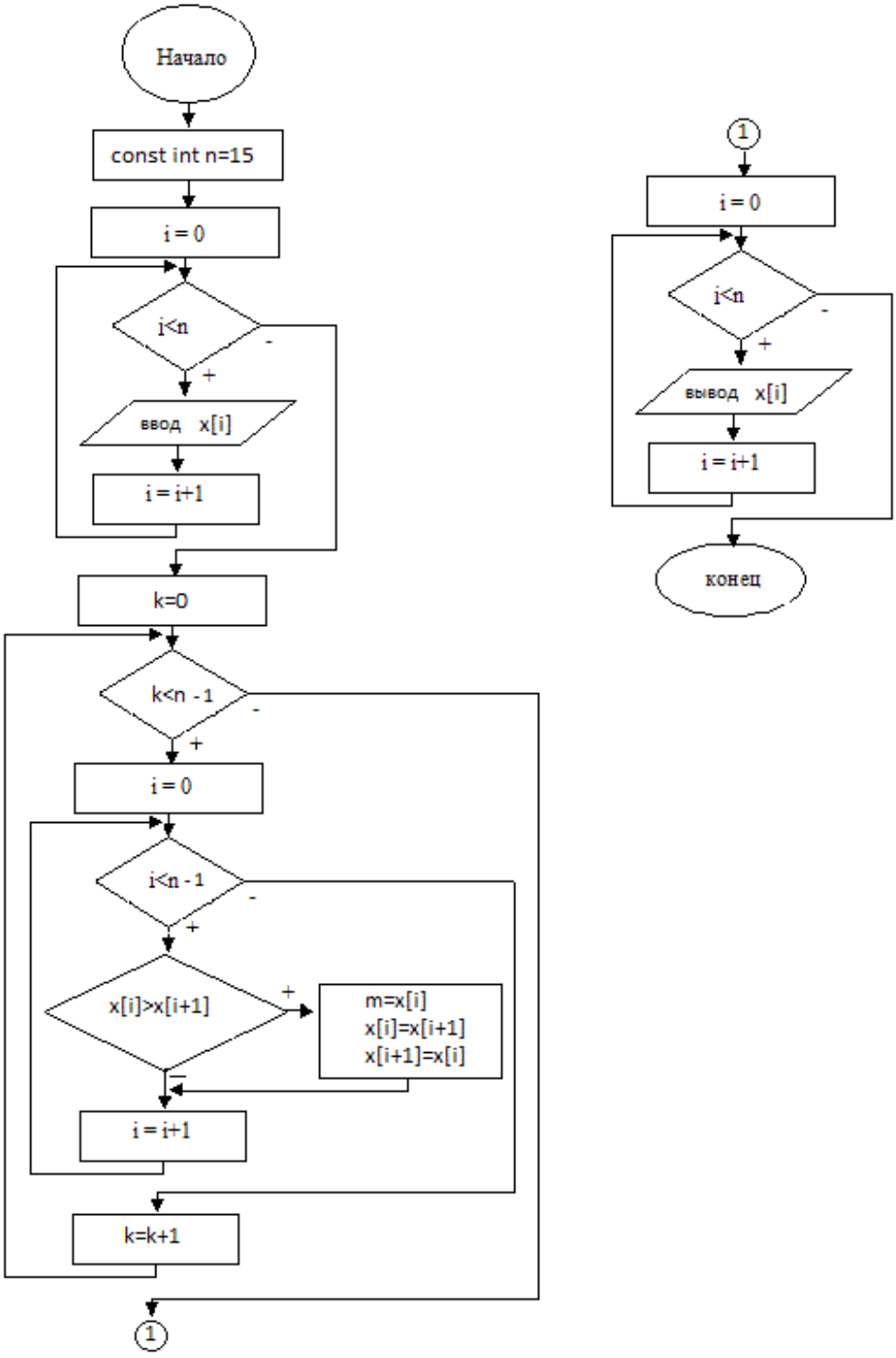


Рисунок 17. Блок-схема алгоритма пузырьковой сортировки

Хотя в предложенном примере сортировка выполнилась за четыре шага,

проверка элементов будет продолжаться еще две итерации, т.к. полное число итераций на единицу меньше размерности массива. Сортировка пузырьковым методом является неэффективным методом, вследствие большого числа сравнений.

Существуют различные модификации данного метода и множество других.

Более эффективным является метод прямого выбора, при котором последовательно происходит просмотр всего списка значений и выбор из него минимального или максимального (в зависимости от порядка сортировки), далее расположение его на нужном месте обменом местами с элементом, стоявшим там ранее. Этот алгоритм представлен на рисунке 18.

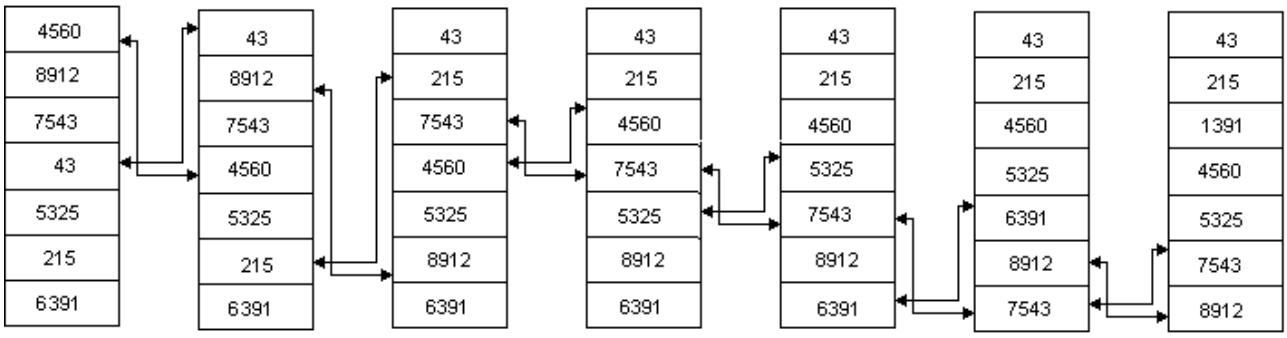


Рисунок 18. Сортировка методом прямого выбора

Для программной реализации этого алгоритма необходимо введение некоторой временной переменной, чтобы при обмене значения переменных не терялись. Функция, с помощью которой реализуется сортировка методом прямого выбора, приведена ниже.

```
#include<iostream>
using namespace std;
int main( )
// сортируем элементы массива по возрастанию
{
const int n=20;
 int list [n];
 int temp;                    //переменная для временного хранения
 int i;                       //переменная управления циклом
```

```
// пропущен фрагмент кода для задания элементов массива
 int place;                    //переменная управления циклом
 int minIndex;                 //индекс минимального значения
    for ( i=0; i < n –1; i++ )
   {
        minIndex = i;
// ищем индекс минимального элемента среди значений list [ i … n -1]
        for (place = i+1; place< n; place++)
              if ( list[place] < list[ minIndex ] )  minIndex = place;
//меняем местами list[ minIndex ] и list[ i ]
        temp = list[ minIndex ];
        list[ minIndex ]= list[ i ];
        list[ i ] = temp;
    }
// после сортировки элементы массива нужно вывести вновь
}
```

Следует отметить, что поиск минимального значения во внутреннем цикле происходит в оставшейся части списка.

Чтобы произвести сортировку по убыванию, нужно на каждом шаге вместо минимального значения находить максимальное значение.

Многомерные массивы

Для объявления многомерного массива необходимо после его имени задать несколько размеров, заключенных в квадратные скобки. Размерность массивов в C++ не ограничивается. Она может зависеть от возможностей компьютера и особенностей конкретного компилятора. Как правило, на практике используются двумерные массивы.

Двумерный массив в C++ представляет собой массив одномерных массивов.

```
float dam[4] [5];
```

В этом случае массив будет содержать 4 строки и 5 столбцов. Инициализация двумерного массива происходит следующим образом:

```
float art [5] [2]={  { 1. 2, 1. 5 },
                     { -4. 0, 3. 6 },
                     { 2. 3,  -6.1 },
                     { 7. 3,  0. 4 },
                     { 0. 0, -2. 7 }  };
```

При этом внутренние фигурные скобки могут быть опущены. Например,

```
float art [5][2] = {1. 2, 1. 5, -4. 0, 3. 6, 2. 3, -6. 1, 7. 3, 0. 4, 0. 0, -2. 7} ;
```

Все сказанное может быть распространено на трехмерный массив.

```
int rum [3][7][2]; // каждый элемент этого массива есть двумерный массив
```

Если одномерный массив используется для представления списка, то двумерный массив – для представления таблицы, содержащей строки и столбцы. При этом подразумевается, что все элементы принадлежат одному типу данных. При обращении к отдельному элементу двумерного массива нужно указать его позицию в строке и столбце. Нумерация строк и столбцов начинается с нуля. Массив из примера можно представить как таблицу на рисунке 15. Выделенный на рисунке элемент есть art [1][2].

[0] [1] столбцы

[0]	1. 2	1. 5
[1]	-4. 0	3.6
[2]	2. 3	- 6. 1
[3]	7. 3	0. 4
[4]	0. 0	- 2. 7

Строки

Рисунок 19. Двумерный массив

Обработка данных в двумерных массивах означает обращение к массиву одним из четырех способов: случайным образом, по строкам, по столбцам, по

всему массиву. Каждый из этих способов может включать обработку части массива.

Самым простым способом получения значения элемента массива является точное указание местоположения элемента. Такой процесс называется случайным доступом, потому что пользователь может ввести любую случайную комбинацию координат, например

```
cout<< art [0][4];
```

Работа со строками

Часто встречаются ситуации, когда требуется обратиться к элементам массива в определенном порядке (например, найти максимальный элемент в каждой строке матрицы).

Пусть задан двумерный массив, содержащий 5 строк и 6 столбцов:

```
int A[5] [6];
```

Предположим, что нужно сложить все элементы строки с номером 3 (четвертая строка) в массиве А и вывести результат. Это можно легко сделать с помощью цикла for:

```
total = 0;
for ( int col = 0; col < 6; col++ )       total+= A[3][col];
        cout << “результат равен” << total << endl;
```

Этот цикл проходит по всем столбцам массива А, но номер строки всегда остается равным 3. Каждое значение из этой строки прибавляется к переменной total.

Если нужно получить сумму в двух строках – номер 1 и 2, то конечно можно дважды привести предыдущий фрагмент дважды: в первый раз указывать индекс 1, а во второй – 2.

```
total = 0;
for ( int col = 0; col < 6; col++ )       total += A[1] [col];
cout << “результат равен” << total << endl;
total = 0;
```

```
for ( int col = 0; col < 6; col++ )       total += A[2] [col];
cout << “результат равен” << total << endl;
```

Но правильнее создать вложенные циклы, а индекс строки сделать переменной - параметром внешнего цикла.

```
for ( int row = 1; row < 3; row++)
        { total = 0;
          for ( int col = 0; col < 6; col++ )      total +=A [row] [col];
           cout << “результат” << row << “ строки равен” << total << endl;
        }
```

Этот способ короче и его значительным преимуществом является несложная его модификация для случая обработки любого диапазона строк.

Внешний цикл управляет изменением номера строки, а внутренний – номером столбца. Для каждого значения переменной row обрабатываются все столбцы, затем внешний цикл переходит к следующей строке.

Таким образом, обращение к элементам массива производится в следующем порядке:

A[1][0] A[1][1] A[1][2] A[1][3] A[1][4] A[1][5]

На второй итерации внешнего цикла переменная row увеличится на единицу и становится равной 2, а индекс столбца изменяется от 0 до 5:

A[2][0] A[2][1] A[2][2] A[2][3] A[2][4] A[2][5].

Таким способом можно обрабатывать все элементы в таблице.

В программах для обозначения строк очень часто используется переменная i, а для столбцов – j.

Пример. В матрице размером 4x7 определить максимальные элементы каждой строки и записать их в одномерный массив и вывести его на печать.

```
#include<iostream>
using namespace std;
int main ( )
{ const int I =4;        // количество строк
  const int J =7;        // количество столбцов
```

```
 int m[I][J]={ 4, 7, 0, 9, 6, 2, 3, 68, 23, 0, -6, 3, 8, 5, 7, 9, -4, 3, 1, 3, 91, 3, 7, 6, 0, 4, 6, 1};
 int mmax [ I ], max, i, j;
 for (  i = 0; i < I; i++)
        {          max= m [ i ][ 0 ];
                   for (  j = 1; j < J; j++ )
                          if  ( m [ i ][ j ] > max )  max = m [ i ][ j ];
                    mmax[ i] = max;
        }
        for (  i = 0; i < I; i++)    cout << "максимумы  строк равны" <<mmax << "  ";
return 0;
}
```

Работа по столбцам

При работе со столбцами внешний цикл организуют по второму индексу массива, а внутренний по первому. Так для нахождения всех положительных элементов в каждом столбца и вывода полученных результатов можно написать:

```
 for ( int  j = 0; j < J; j++)
        { summa = 0;
          for ( int i = 0; i < I; i++ )       if ( m[ i ][ j ] > 0)  m += m [ i ][ j ];
          cout << "результат" << j << " столбца равен" << summa << endl;
        }
```

Сначала складываются все положительные элементы первого столбца, выводится результат, и только потом индекс внешнего цикла меняется и происходит обращение к элементам следующего столбца.

Не всегда требуется обработка элементов массива в каждом столбце.
Если необходимо обработать элементы каждого второго столбца можно в условие выполнения цикла добавить дополнительное условие:

```
for (  int j = 1; j < n  && (j + 1)%2 == 0; j ++) // цикл по второму индексу
        for ( int   i = 0; i < n; i++ )
```

Однако этот алгоритм неэффективен, поскольку перебирает все столб-

цы. Пропустить "лишние" столбцы можно, изменив оператор, предназначенный для изменения параметра цикла:

```
for (  int j = 1; j < n; j += 2 )
        for (   int i = 0; i < n; i++ )
```

Также обратите внимание, что внешний цикл начинается с j равного единицы, поскольку столбец с нулевым индексом должен быть пропущен.

Инициализация таблицы

Как и в случае одномерных массивов, двумерные массивы можно инициализировать при объявлении. Это непрактично, если таблица имеет большую размерность. Для ввода значений с клавиатуры также можно использовать вложенные циклы.

```
for ( int  j = 0; j < J; j++)
for ( int i = 0; i < I; i++ )        cin >> m [ i ][ j ];
```

Здесь запись производится построчно, для ее выполнения по столбцам внутренний и внешний циклы нужно поменять местами.

При задании массива произвольным образом более эффективно использование элементов счетчика случайных чисел.

Вывод таблицы

Еще одним случаем обработки таблицы является задача вывода ее содержимого на экран в общепринятом виде (таблицей значений). Как правило, это подразумевает ее построчный вывод:

```
for ( int i = 0; i < I; i++ )
{       for ( int  j = 0; j < J; j++)       cout << m [ i ][ j ]<< “  “;
cout << endl;
}
```

Внешний цикл позволяет выполнить вывод всех элементов строки, изменением второго индекса во внутреннем цикле, и переход на новую строчку перед каждым изменением параметра внешнего цикла.

Примеры программ обработки двумерных массивов

Рассмотрим алгоритм нахождение суммы отрицательных элементов в каждой втором столбце двумерного массива.

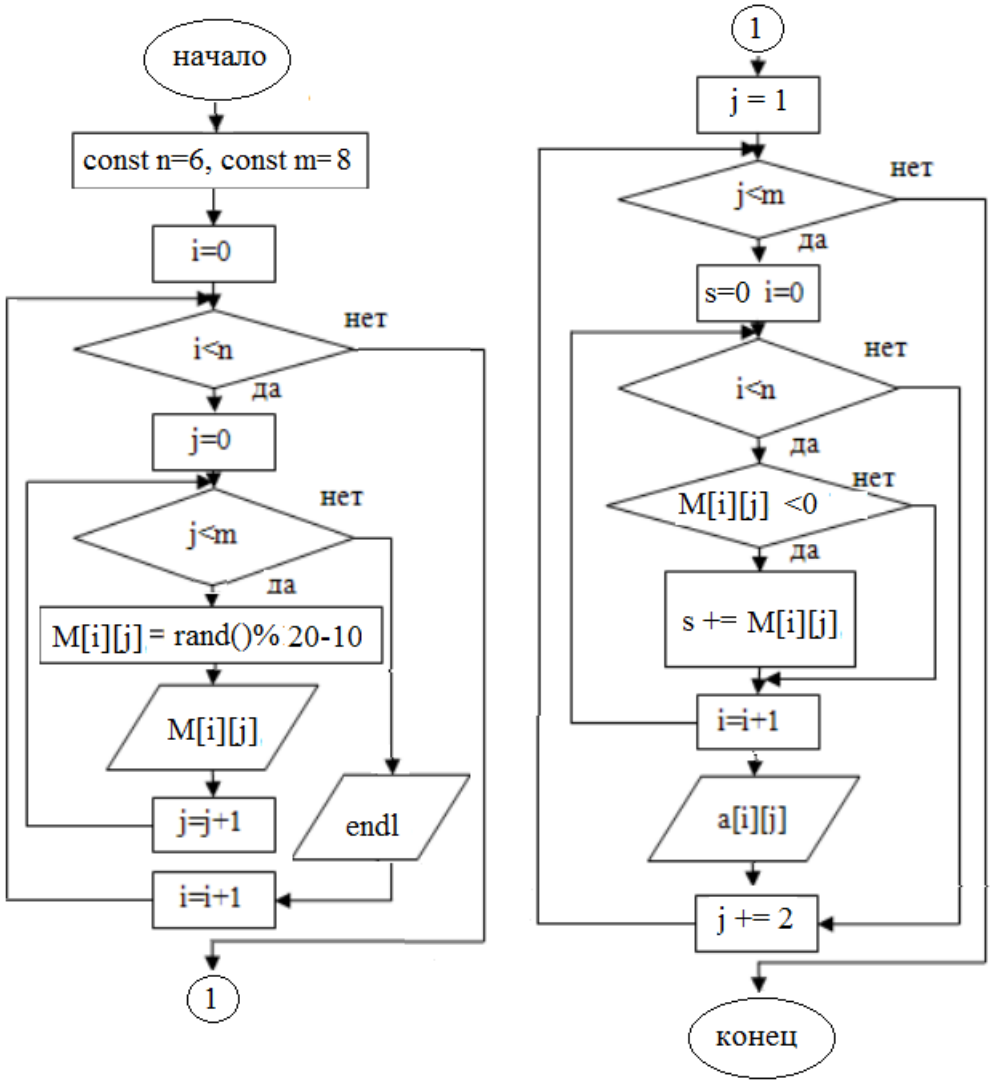


Рисунок 20. Блок-схема алгоритма обработки двумерного массива по столбцам

Программа, реализующая алгоритм, представленный на рисунке 20:

```
#include <iostream>
#include <cstdlib>
#include <iomanip>
using namespace std;
int main ( )
{
```

```
setlocale(LC_ALL, "");
const int n = 6;
const int m = 8;
int M[n][m], i, j, s;
cout<< " элементы массива, заданные случайным образом "<<endl;
  for (  i = 0; i < n; i++ )
                  {
                  for (   j = 0; j < m; j++)
                  { M[ i ][ j ]=rand( )%20 - 10;  // задание значений в диапазоне от -10 до 10
            cout << setw(5) << M[ i ][ j] ;
           }
        cout<<endl;
        }
          for (  j = 1; j < m; j += 2 )
    { s = 0;
          for (   i = 0; i < n; i++ )
             if ( M[ i ][ j ]< 0)  s += M[ i ][ j ];
    cout<< endl<< "сумма отрицательных элементов в столбце с индексом "<< j << "
равна " << s <<endl;
    }
    return 0;
}
```

Пример. Определение минимального значения среди элементов матрицы M(6, 6), расположенных над главной диагональю.

```
#include<iostream>
#include <ctime>
#include <cstdlib>
#include <locale.h>
using namespace std;
```

```
int main ( )
{
const int n = 6;
int M[n][n], i, j, min;
srand( time( NULL ) );
for (  i = 0; i < n; i++ )
        for (   j = 0; j < n; j++)
        M [ i ][ j ]=rand() %20 - 10;            // задание значений в диапазоне от -10 до 10
for (  i = 0; i < n; i++ )                       // вывод на экран полученной матрицы
        {       for ( int  j = 0; j < n; j++)      cout << M [ i ][ j ]<< "  ";
        cout << endl;
        }
min = M[0] [1];
for (  i = 0; i < n; i++ )
        for (   j =i+1; j < n; j++)              / * условие расположения элементов
                                                        над     главной диагональю */
        if ( M[ i ][ j ]<min)  min = M[ i ][ j];
 cout<< "значение минимума "<<min;
        }
```

Если требуется преобразовать элементы, расположенные в матрице под главной диагональю и на ней, используют вложенные циклы вида

```
for (  i = 0; i < n; i++ )
        for (   j =i; j < n; j++)
```

В случаях работы с элементами матрицы, лежащими на главной диагонали, достаточно использовать один цикл.

Пример. Поменять местами максимальный и минимальный элементы главной диагонали матрицы T(7, 7).

```
#include<iostream>
#include <ctime>
#include <cstdlib>
```

```
#include <locale.h>
using namespace std;
int main ( )
{
const int n = 7;
int T[n][n], i, j, min, max, nmin, nmax;
srand( time( NULL ) );
for (  i = 0; i < n; i++ )
        for (   j = 0; j < n; j++)
        T [ i ][ j ]=rand() %30- 15;    //задание значений в диапазоне от -15 до 15

for (  i = 0; i < n; i++ )                          // вывод на экран полученной матрицы
        {       for ( int  j = 0; j < n; j++)       cout << T [ i ][ j ]<< "  ";
        cout << endl;
        }
min = max=T[0] [0];
nmin=nmax=0;
for (  i = 1; i < n; i++ )
                {
                if ( min > T[ i ][ i ] )    { min = T[ i ][ i ];     nmin = i; }
                else if ( max < T[ i ][ i ] )      { max = T[ i ][ i ];    nmax = i; }
                }
int t = T[nmin][nmin];                  // вспомогательная переменная для обмена
T[nmin][nmin] =T[nmax][nmax];
T[nmax][nmax] =t;
        for (  i = 0; i < n; i++ )                          // вывод на экран измененной матрицы
        {       for ( int  j = 0; j < n; j++)       cout << T [ i ][ j ]<< "  ";
        cout << endl;
        }
}
```

Библиотека ввода-вывода stdio.h

Аббревиатура stdio.h означает standard input output header, т.е. заголовок стандартного ввода-вывода. Данная библиотека содержит прототипы функций стандартного ввода вывода. Она широко использовалась в языке С и поддерживается в С++.

Функция printf( ) – форматный вывод. Синтаксис обращения к printf( ):

printf(“управляющая строка”[, параметр1 [, параметр2 [, . . ., параметрn]]]);

где параметр1, параметр2, . . ., параметрn – выводимые на экран значения, которые могут быть переменными, константами или выражениями, вычисляемые перед выводом.

Управляющая строка задает формат вывода значений на экран. Управляющей строкой служит фраза, помещенная в кавычки и содержащая информацию двух видов: тест, непосредственно выводимый на экран, и инструкции, зависящие от типа выводимых значений (форматы вывода) и размещенные на месте предполагаемого вывода значений.

Формат	*Тип переменной*
%d	Десятичное целое число
%c	Один символ
%s	Строка символов
%e	Экспоненциальная запись числа с плавающей точкой
%f	Десятичная запись числа с плавающей точкой
%u	Десятичное число без знака
%o	Восьмеричное целое без знака
%x	Шестнадцатеричное целое без знака

Таким образом, управляющая строка показывает, в каком виде будет осуществлен вывод информации на экран.

Например, запись вида

```
const float pi= 3.14;
printf (“значение числа pi равно %f \n”, pi);
```

выведет на экран:

значение числа pi равно 3.14

после чего осуществиться переход на новую строку.

Каждому аргументу из списка, следующему за управляющей строкой, должна соответствовать отдельная инструкция:

```
int x=125, y;
y=x*x;
printf (“Квадрат числа %d равен %d \n”, x, y);
```

Это приведет к выводу

Квадрат числа 125 равен 15625

Указанием в инструкции числа (спецификатора точности) определяет ширину поля и точность. Например,

```
int k = 12;
float x = 5.12, y = 4.275, z;
z=x*y;
printf (“значение числа k равно %2d \n”, k);
printf (“x = %5.2f, y = %6.3, z = %7.3f  \n”, x, y, z);
```

Первая цифра означает количество знаков для вывода всего числа, а вторая количество знаков после запятой. Если цифр в числе меньше указанного формата лишние цифры просто отбрасываются.

Если спецификатор точности не задан, точность будет установлена по умолчанию: 1 – для форматов d, o, u, x; 6 – для формата e.

scanf( ) – функция форматного ввода. В ней также указывается управляющая строка и список аргументов. Основное отличие от printf( ) – в управляющей строке указывается только инструкции формата, а в списке параметров перед именем переменных обязательно располагается &. Таким образом, функция scanf() использует адреса расположения переменных в памяти (указатели). Особенным является использование формата строковой переменной %s, перед именем переменной знак & не указывается. Например,

```
printf (“укажите Ваше имя, возраст и состояние:”);
scanf( “%s  %d  %f ”, name, &age, &st );
printf ( “%s  -  %d лет, %f тыс. руб. \n”, name, age, st);
```

Ввод-вывод одного символа осуществляется с использованием функций

getchar( ) и putchar( ). Функция getchar( ) – получает один символ, поступающий с клавиатуры и возвращает его, putchar( ) выводит один символ на экран.

```
char ch=getchar( );
putchar( ‘s’ );
putchar( ch);
putchar(getchar( ));
```

Библиотека stdio.h не ограничивается перечисленным и содержит множество других полезных функций.

Символьные строки

Символьная строка – последовательность одного и более символов, заключенная в двойные кавычки. Для формирования символьных строк, занимающих несколько строк программы, используется комбинация символов \ и \n. Кавычки не являются частью строки, они служат для обозначения ее начала и конца.

Строки представляются в виде массива элементов типа char. Число элементов символьного массива должно быть на единицу больше числа символов, которые предстоит хранить в строке. Дополнительная ячейка памяти требуется для размещения нуль-символа (/0), который автоматически добавляется в качестве последнего байта в памяти для обозначения окончания строки.

Как и любую другую переменную строку можно инициализировать при объявлении. При этом размерность можно не указывать, компилятор определит ее самостоятельно. Инициализировать строку можно всю целиком или посимвольно:

```
char message [10] = {‘с’, ‘о’, ‘о’, ‘б’, ‘щ’, ’е’, ‘н’, ‘и’, ‘е’};
char message [ ] =  “сообщение”;
```

Строка может быть строковой константой или переменной (символьным массивом). В обоих случаях можно выводить элементы строки с помощью оператора cout<< до тех пор, пока не встретится нуль-символ.

Например:

```
cout<<”results are:”;
char msg[ ]=”welcome”;
```

```
cout<<msg;
```

Для ввода строк существует несколько возможностей. Первая – использование оператора cin. При чтении вводимых данных этот оператор будет пропускать все предшествующие непечатаемые символы: пробелы и символы перевода строки. Он также автоматически добавляет нуль-символ в конец строки.

Кроме того, данный оператор нельзя использовать для ввода строк, содержащих пробелы. Еще один его недостаток заключается в том, что если строковая переменная недостаточно велика, чтобы вместить в ней последовательность вводимых символов и нуль-символ, то данные из потока ввода будут записываться в память за пределами массива.

Для ввода строк лучше использовать функцию cin.get ( ), которая имеет два параметра: строковую переменную и целое типа int, отвечающее за количество вводимых символов в строке плюс один (для нуль-символа).

Например:

```
char line[51];
cin.get(line, 51);
```

В этом случае непечатные символы не пропускаются. Функция считывает и сохраняет полностью всю строку ввода (длиной не более чем указано вторым параметром). Чтобы правильно считать две последовательные строки, нужно не забывать о символе передачи строки:

```
char dummy='/n';
cin.get(line1, 51);
cin.get(dummy);       //убрать '/n' из потока ввода перед использованием get
cin.get(line2, 51);
```

Язык C++ предлагает большой ассортимент полезных функций для работы со строками. Прототипы всех функций работы со строками содержатся в файле string.h. Рассмотрим несколько из них.

Функция strlen() вычисляет длину строки без нуль-символа. Функция имеет один аргумент – имя строки.

```
#include <string.h>
```

```
#include <iostream>
using namespace std;
int  main ( )
{
char str[ ] = ”Hello, my friend!”;
cout<< strlen(str);              // результат 17
return 0;
}
```

Функция strcmp( ) сравнивает две строки. Она имеет два аргумента и возвращает целое значение, которое равно 0, если строки полностью совпадают.

Значение strcmp(str1, str2) является целым числом меньше нуля, если str1<str2 и целое >0, если str1>str2. Сравнение строк происходит в лексикографическом порядке, т.е. в порядке их расположения в словаре.

Пример. Функция, проверяющая правильность введенного пароля с трех попыток.

```
#include <string.h>
#include <iostream>
using namespace std;
int main ( )
{
char s[5], pass[ ] =”лето”;
int i_true=0;
for (int i=0; i<3; i++)
        { cin>>s;
        if (strcmp (s, pass) ==0)  {  i_true = 1; break; }
        }
if (i_true = = 0) { cout <<”пароль неверен”; return 1; }
        else { cout<<”пароль верен”; return 0; }
}
```

Функция strcpy( ) копирует содержимое второй, из указанных в качестве па-

раметра, строки в первую, замещая прежние данные, включая ‘/0’. При этом торая строка не изменяется. Первая из указанных строк должна иметь достаточный размер, иначе копирование не выполняется.

```
char mystr[100];
strcpy(mystr, “abcdefgh”);
```

Функция strcat( ) присоединяет строку, указанную в качестве второго параметра, к первой указанной строке.

Работа с символьным массивом аналогична работе с числовым массивом.

Пример. Определение количества вхождений каждого символа в заданную строку.

```
#include <iostream>
#include <string.h>
using namespace std;
int  main ( )
{ int k;
  char str[100];
  cout<<“Введите любую последовательность символов без пробелов”;
 cin>> str;
  for (char ch=’a’; ch<=’z’; ch++)
          { k=0;
            for ( int i=0; i < strlen(str); i++)
             if (str [ i ] ) == ch) k++;
             if (!k)   cout<<”Количество символа” << ch << “равно”<< k<<endl;
          }
}
```

Программа, сортирующая заданной строки в алфавитном порядке.

```
#include<iostream>
#include<string.h>
using namespace std;
int main( )
```

```
{
char m[ ]=" ночевала тучка    золотая на груди            утеса-великана ";
int i;
for (char c='a'; c<'я'; c++)
        {  i=0;
           if ( m[0]==c)          { cout<<m[0];
                                    while (m[i+1]!='  ')          { cout<<m[i+1];  i++; }
                                    cout<<endl;
                                  }
          for  ( i=i; i<strlen(m); i++ )
                if (m[ i ]== '  ' && m[i+1]==c)          { while (m[i+1]!=' ') { cout<<m[i+1];  i++; }
                                                           cout<<endl;
                                                         }
          }
}
```

В примере первый цикл организован для пропуска лишних пробелов между словами.

Массив символьных строк выглядит как двумерный массив. К его строкам применимы все функции для работы со строками. Можно работать и непосредственно с отдельными его элементами. Так инициализировать символьный массив, содержащий четыре строки можно следующим образом:

static char fruit[4][ ]={“Слива”, “Персик”, “Яблоко”, “Апельсин”};

Теперь компилятор автоматически определит количество элементов в строке – девять, т.к. самое длинное слово содержит восемь символов плюс один для размещения нуль-символа.

Функции

Ранее говорилось, что любая программа, написанная на языке C++ есть последовательность выполнения функций, причем одна из них обязательно должна называться main( ). Выполнение программы всегда начинается с

выполнения ряда операторов этой функции. Все функции в языке С++ равноправны: каждая из них (даже main( )) может быть любой другой функцией. Функция может вызывать саму себя (явление рекурсии). Компилятор не ограничивает число рекурсивных вызовов, но операционная система может наложить ограничения.

Функция – это именованная последовательность описаний и операторов, выполняющая какое-либо законченное действие. Функция может принимать параметры и возвращать значение.

Фигурные скобки отмечают начало и конец тела функции. В круглых скобках после ее имени, в общем случае, содержится информация, передаваемая этой функции – описание параметров (тип и имя).

Наличие списка параметров и их описаний не является обязательным. Но круглые скобки всегда должны присутствовать после имени функции. Тип параметра может быть любым. Если параметров несколько, их описания разделяются запятыми. Если функции не передаются величины, то вместо списка аргументов можно задавать ключевое слово void.

Некоторые компиляторы требуют обязательного указания типа возвращаемого результата перед именем функции. Если тип результата не указывается, то предполагается, что функция возвращает значение типа int. Если функция не возвращает результат, указывается слово void.

Возвращает значение оператор return, с помощью которого можно передать в вызывающую функцию только одно значение. Возвращаемое значение берется в круглые скобки после ключевого слова return. Круглые скобки не являются обязательными, но считаются правилом хорошего тона среди программистов.

Переменные, используемые в функции, отличные от параметров, должны описываются внутри тела функции. Они называются локальными.

Пример. Функция, возвращающая минимальное значение из трех величин.

```
        // определение функции min
float min (float val1, float  val2, float val3 ) {
        if ( val1< val2 &&  val1<val3) return val1;
```

```
        else if ( val2 < val1 &&  val2<val3) return val2;
            else if ( val3 < val2 &&  val3<val1) return val3;
}
```

Эта функция может быть вызвана любой другой функцией. Вызов может выглядеть, например, так

```
a = min (525, 675, 819);
```

Если в программе объявлены некоторые переменные: `float a, a1, a2, a3;` тогда возможен вызов

```
a = min (a1, a2, a3);
```

Параметры (аргументы) функции, используемые в ее описании, называются *формальными*, параметры, содержащиеся в вызове функции и располагаемые на их месте, называются *фактическими*.

Так в описании функции min формальными параметрами являются val1, val2, val3, а фактическими параметрами в ее вызове становятся 525, 675, 819 и a1, a2, a3.

Количество и типы формальных и фактических параметров должны совпадать. Иначе компилятор выдаст ошибку.

В случаях, когда вызываемая функция не возвращает значение, ее вызов представляет собой просто имя функции, за которым в круглых скобках указываются фактические параметры, если таковые имеются.

Функции могут располагаться в тексте программы в любом порядке, но следует помнить, что они должны быть определены до своего вызова.

Так пример программы, содержащей функцию min( ) может выглядеть так:

```
#include <iostream>
using namespace std;
// определение функции min
float min (float val1, float  val2, float val3 )
{                                                    // начало функции min
    if ( val1< val2 &&  val1<val3) return val1;
        else if ( val2 < val1 &&  val2<val3) return val2;
            else if ( val3 < val2 &&  val3<val1) return val3;
```

```
}                                                    // окончание функции min
// определение функции main
int main ( )
{                                                    // начало функции main
  int x, y, z;
  cout<< " Введите три числа";
  cin>> x >> y >> z;
  cout<< "минимум из этих чисел"<< min (x, y, z);   //вызов функции min
  return 0;
}                                                    // окончание функции main
```

Необходимо отметить, что приведенный пример является "неграмотным" с точки зрения эффективности программы. Поскольку функции в программе должны создаваться с целью упрощения алгоритма программы, структуризации сложной программы, ее наглядности и читаемости. В данной программе нецелесообразно выделять отдельную функцию нахождения минимума, так как это только усложняет ее текст. Функции используются, как правило, когда требуется выполнить одинаковые действия с данными, имеющими различными значениями одинакового типа.

Разделение программы на функции позволяет избежать избыточности кода, поскольку определение функции записывается один раз, а вызывать ее можно многократно из разных точек программы.

Тело функции похоже на любой другой фрагмент кода, за исключением того, что оно содержится в отдельном блоке внутри программы. При разработке функции нужно формально описать ее поведение и механизм взаимодействия с ней, т. е. входные и выходные значения.

Нередким является случай, когда функция вызывается до того, как будет объявлена. В этом случае используется прототип функции. Прототип функции имеет следующий вид:

возвращаемый тип имя функции (параметры);

Таким образом, прототипом является заголовок функции, оканчивающийся

точкой с запятой. В списке параметров обычно указывается тип и имя для каждой переменной; элементы списка разделяются запятыми. Указание имени параметров в прототипе не обязательно, но, как правило, применяется.

Прототип информирует компилятор о существовании функции, о типе возвращаемого значения, а также о типе и количестве аргументов, которые ей передаются.

Так для функции из вышеприведенного примера прототип выглядит следующим образом:

```
void print_str( int x);
```

или

```
void print_str( int );
```

Прототипы функций, как правило, располагают после директив препроцессора, до описания основной функции. Рассмотрим пример программы нахождения площадей треугольников с использованием прототипов функций.

```
#include <iostream>
#include <math. h>
using namespace std;
void print_area (float, float, float);   //объявление прототипа функции print_area
int  main ( )                            //описание функции main
{
  int n; float a, b, c;
  cout<<"Введите количество треугольников"<<endl;
  cin >> n;
  for (int i=1; i<=n; i++)
       { cout << "Введите стороны треугольников a, b, c >0" << endl;
        cout << "a = ";  cin >> a;
       cout << "b = ";  cin >> b;
```

```
        cout << ”c = “;  cin >> c;
        print_area (a, b, c);
        }
}
void print_area (float x, float y, float z)      //описание функции print_area
{
        if ( ( x+ y > z )&&( x+ z > y )&&( y+ z > x ) )
        { float p=( x + y+ z) / 2;
          cout <<”Площадь равна “<<sqrt(p*(p-x)*(p-y)*(p-z))<<endl;
        }
        else cout<<”Треугольник невозможно построить”<<endl;
}
```

Нужно разделять понятия «определение» и «объявление» функции. Объявление функции это заголовок или прототип. Определение функции, кроме объявления, содержит тело функции.

Все функции в программе, созданной на языке С++, равноправны, т.е. любая функция (включая main) может вызвать другую, в том числе и саму себя. Вызов функцией самой себя называется рекурсией. В языке С++ количество рекурсивных вызовов не ограничивается, а определяется лишь возможностями компьютера.

Рекурсия

В программирование рекурсией называют вызов себя из себя. Функция, вызывающая саму себя, называется рекурсивной. Использование такой функции называется рекурсией.

Рекурсия бывает прямой и косвенной. В первом случае функция содержит в своем теле вызов самой себя. В случае косвенной рекурсии одна функция вызывают другую функцию, а вызванная функция имеет отношение к дальнейшему вызову первой функции.

Классическим примером применения рекурсии является вычисление факториала числа.

Организуем ввод пользователем числа, которое является фактическим параметром функции. Условием окончания рекурсии будет равенство введённого числа нулю. В этом случае функция более не вызывает саму себя.

```
#include<iostream>
using namespace std;
unsigned long int fact(unsigned long int n);   // прототип функции
int main( )
{
    unsigned long int a;
    cout << "Enter number: "<< endl;      cin >> a;
    cout << fact(a) << endl;
     return 0;
}
 unsigned long int fact( unsigned long int n)   // описание функции
{
    unsigned long int f;
    if (n == 0)         f = 1;
                else   f = n * fact(n - 1);
    return f;
}
```

Важно в создание рекурсивных функций предусмотреть основной (базовый) случай. В данном примере этот случай – факториал нуля.

Например, при вызове fact(5) получится следующая цепь вызовов функции:

5 * fact(4)

5*4*fact(3)

5*4*3*fact(2)

5*4*3*2*fact(1)

5*4*3*2*1*fact(0)

В программе использован тип unsigned long int , поскольку значение факториала - как правило, очень большое число.

Другим распространенным примером рекурсивной функции служит функция, вычисляющая значение чисел Фиббоначчи – n-й член последовательности чисел Фибоначчи определяется по формуле: f(n)=f(n – 1) + f(n –2), причем f(0) = 0, а f(1) = 1. Значения f(0) = 0 и f(1) =1 – базовые варианты для данного случая.

```
#include <iostream>
using namespace std;
int fibonachi(int);
 int main( )
{
   int n;
   for(int i = 0; i < 10; i++)
   {
      n = fibonachi(i);
      cout << n << "\t";
   }
   cout << endl;
   return 0;
}

int fibonachi(int n)
{
   if (n == 0)
      return 0;
   if (n == 1)
      return 1;
   return fibonachi(n - 1) + fibonachi(n - 2);
}
```

Рекурсивные функции чаще применяются для реализации рекурсивных алгоритмов. Любую рекурсивную функцию можно реализовать и без применения рекурсии.

Достоинством рекурсии является компактная запись, а недостатком - расход на повторные вызовы функции и передачу ей копий параметров, а также опасность переполнения стека, в котором хранятся копии значений параметров функции.

Область действия и время жизни переменных

Как уже говорилось, переменные в языке C++ могут быть объявлены в любом месте программы. Локальные переменные – это переменные, объявленные внутри блока, например, такого как тело функции или условный оператор. К локальным переменным нельзя обращаться вне блока, который их содержит.

```
if (k>0)
{ int n;
  cin>>n;
  k=k+n;
}
```

Поскольку в данном примере переменная n – локальная, то к ней нельзя обращаться в любом выражении вне блока. То же правило доступа относится и к локальным именованным константам.

Но переменные могут быть объявлены и вне блока. Такие переменные называются глобальными. Глобальные переменные видны во всех функциях программы, где не описаны локальными переменные с теми же именами. Использовать их для передачи между функциями очень легко. Тем не менее, это делать не рекомендуется, поскольку затрудняется отладка программы. Использование глобальных переменных считается плохим стилем программирования.

Если перечислить все фрагменты программы, из которых к идентификатору можно обращаться, то получится описание области видимости идентификатора

или его *области действия*. C++ определяет три категории области действия для любого идентификатора:

- Область действия класса. Этот термин относится к типу данных, называемому классом.
- Локальная область действия. Область действия идентификатора, объявленного внутри блока, простирается от точки объявления до конца этого блока.
- Глобальная (файловая) область действия. Область действия идентификатора, объявленного снаружи всех классов и функций, простирается от точки объявления до конца всего файла.

Когда функция объявляет локальный идентификатор с тем же самым именем, что и у глобального идентификатора, локальный идентификатор имеет приоритет внутри тела функции. Другими словами, область действия идентификатора не включает вложенные блоки, которые содержат локально объявленный идентификатор с точно таким же именем.

Имена функций C++ имеют глобальную область действия, и не существует такого понятия, как локальная функция – то есть запрещается вкладывать описание одной функции в другую.

Глобальные переменные встречаются достаточно редко. Имеются отрицательные аспекты их использования. Область действия глобальной переменной или константы простирается от точки ее объявления до конца файла.

Понятие, и связанное и отдельное от области действия переменной – это *время жизни* – период времени в процессе выполнения программы, когда идентификатор фактически занимает место в памяти. Память для локальных переменных выделяется в момент перехода управления на функцию. Затем переменные «оживают» на время работы функции, и память освобождается при выходе из функции. Время жизни глобальной переменной – это время работы всей программы. Память для глобальных переменных выделяется только один раз, когда программа начинает выполняться, и освобождается только при завершении программы.

Необходимо отметить, что понятие области действия связано с этапом ком-

пиляции, а время жизни – с этапом выполнения программы.

В C++ каждая переменная имеет класс памяти, который определяет время жизни. Существует четыре спецификатора класса памяти: auto, register, static, extern.

Спецификатор класса памяти может предшествовать объявлению переменных и функций, указывая компилятору, как следует хранить переменные в памяти и как получать доступ к переменным или функциям. Переменные, объявленные со спецификаторами auto и register, являются локальными, а со спецификаторами static и extern – глобальными. Смысл спецификатора класса памяти несколько различается в зависимости от места объявления переменной или функции.

Переменная, объявленная на внешнем уровне (вне любой функции), по умолчанию имеет класс памяти extern. Область ее действия распространяется до конца файла. С помощью спецификатора extern можно объявить переменную, которая будет доступна из любого места программы. Это может быть ссылка на переменную, описанную в другом файле или ниже в том же файле.

Если в одном из файлов создана переменная с классом памяти static, то она может быть объявлена с тем же именем и с тем же спецификатором static в любом другом исходном файле. Так как статические переменные доступны только в пределах своего файла, конфликтов имен не возникает.

При объявлении переменной на внутреннем уровне (в теле функции) можно использовать любой из четырех спецификаторов памяти (по умолчанию устанавливается класс auto). Переменные, имеющие класс памяти auto, имеют область видимости ограниченную блоком, в котором они объявлены.

Спецификатор register указывает компилятору, что данную переменную необходимо сохранить в регистре процессора, если это возможно. В результате сокращается время доступа к данным и упрощается программный код. Область действия регистровых переменных такая же, как и у автоматических. В случае отсутствия свободных регистров переменной присваивается класс auto, и она сохраняется в памяти.

Переменная, объявленная на внутреннем уровне со спецификатором static,

будет глобальной, но доступ к ней получить можно только внутри блока. По умолчанию статической переменной присваивается нулевое значение.

Спецификатор extern на уровне блока используется для создания ссылки на переменную с те же именем, объявленную на внешнем уровне в любом исходном файле программы. Если в блоке определяется переменная с тем же именем, но без спецификатора extern, то внешняя переменная становится недоступной в данном блоке.

Пример:

```
//исходный файл 1
extern int i;    //объявление, ссылающееся на данное ниже определение  i
int main( )
{
i=i+1;
cout<<i;        //значение i равно 4
next( );
}
int i = 3;                  // определение i

void next( )
{
i = i+1;
cout << i;       // значение i равно 5
other( );
}
//исходный файл 2
extern int i; // объявление i, ссылающееся на определение i в первом исходном файле
void other( )
 {
i = I + 1;
cout << i;       // значение i равно 6
```

```
}
```

В примере два исходных файла в совокупности содержат три внешних объявления переменной i. Только в одном содержится ее инициализация. Самое первое объявление extern в первом исходном файле делает глобальную переменную доступной прежде ее определения в файле. Объявление переменной во втором исходном файле делает глобальную переменную доступной во втором файле.

Если бы переменная не была инициализирована ни в одном из объявлений, она бы была неявно инициализирована нулевым значением при компиляции. В этом случае программа напечатала бы значения 1, 2, 3.

Указатели

При обработке любых объявлений переменных компилятором в памяти выделяется место для хранения значения переменной в соответствии с ее типом. Все дальнейшие обращения в программе к переменной по имени заменяются компилятором на начальный адрес выделенной для нее области памяти.

Некоторые языки не позволяют работать с указателями, что однозначно дает языку C++ преимущество. С другой стороны, при программировании на C++ работа с указателями становится неизбежной.

Программист может также определять переменные для хранения адресов памяти. Типами таких переменных являются указатели. Указатель не является самостоятельным типом, он всегда связан с каким-либо другим типом.

Синтаксис объявления указателя:

тип указываемых данных * имя указателя;

Таким образом, описание указателя определяется типом данных, на которые указатель в дальнейшем ссылается.

Примеры объявления указателя:

```
int *int_ptr;          // указатель на целое
double *d_ptr;         // указатель на тип double
char *c;               // указатель на символьную переменную
int *array[10];        // объявление массива указателей, каждый из которых
```

```
                                    // указывает на  значение int
int (*pointer)[10];                 // объявлен указатель с именем pointer, который
                                    // указывает на  массив из 10 элементов
```

Возможно также использование указателя на тип void, который может указывать на объект любого типа. Его обычно называют пустым указателем. При выполнении операций над пустым указателем, либо над объектом, на который он указывает, необходимо явно привести тип указателя к типу отличному от void.

Как и любые другие переменные, указатели можно инициализировать при их объявлении. Например, в следующем фрагменте создаются две именованные ячейки памяти result и p_result.

```
int result;
int *p_result=&result;
```

Идентификатор result представляет собой обычную целочисленную переменную, а p_result – указатель на переменную типа int. Одновременно с объявлением указателя p_result происходит его инициализация адресом переменной result. Сама переменная result остается неинициализированной.

При определении указателей надо стремится выполнить их инициализацию. Непреднамеренное использование неинициализированных указателей – распространенный вид ошибок в программах.

Можно также записывать инициализатор после имени указателя в круглых скобках:

```
int result;
int *p_result (&result);
```

или явно присваивать указателю адрес области памяти:

```
float* r=  (float*)0xB8000000;
```

где 0xB8000000 – шестнадцатеричная константа.

Для присваивания указателю пустого значения используют 0 или константу NULL, определенную в заголовочных файлах C++.

Операции с указателями

Для указателей, как и для переменных других типов, существует ряд операций: присваивание, сложение с константой, вычитание, инкремент, декремент, сравнение, приведение типа, косвенная адресация или разадресация (разыменовывание) – * и операция получения адреса – &.

Результатом операция косвенной адресации является величина, помещенная в ячейку с указанным адресом. Ее можно использовать для получения и изменения значения некоторой величины.

Операция получения адреса (&) можно применять далеко не с каждым выражением. Когда за этим знаком следует имя переменной, результатом операции является номер ячейки памяти, в которой она хранится.

Недопустимо попытка получения адреса в случаях:

- константного выражения, например ptr=&57;

- в выражениях с арифметическими операторами

```
int rezult=0;
ptr=&(result+35);
```

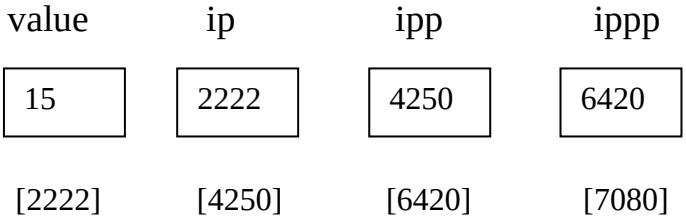
- с переменными класса памяти register.

В C++ можно создавать указатели на другие указатели, которые, в свою очередь, содержат имена реальных переменных. Чтобы объявить в программе указатель, который в свою очередь будет хранить адрес другого указателя, нужно просто удвоить число звездочек в объявлении. Количество указателей в цепочке, задающее уровень косвенной адресации, соответствует числу звездочек перед именем указателя. Уровень косвенной адресации определяет, сколько раз следует выполнить операцию раскрытия указателя, чтобы получить значение конечной переменной. В следующем фрагменте создается ряд указателей с различными уровнями косвенной адресации.

```
int value=15;
int *ip;
int **ipp;
int ***ippp;
```

```
ip=&value;
ipp=&ip;
ippp=&ipp;
```

В четырех первых строчках объявлены четыре переменные: value типа int, указатель ip на переменную типа int (первый уровень косвенной адресации), указатель ipp (второй уровень адресации) и указатель ippp (третий уровень адресации). Можно создать указатель любого уровня. В пятой строке указателю первого уровня ip присваивается адрес переменной value. Теперь значение переменной value (15) может быть получено с помощью выражения *ip и т.д.



В результате вычитания целого значения указатель будет ссылаться на элемент, смещенный на указанную величину ячеек влево по отношению к текущей ячейке. Соответственно использование операций инкремента и декремента к указателю будет производить сдвиг на количество ячеек памяти соответствующие типу переменной, на которую ссылается указатель. Предполагается, что оба указателя одного типа и связаны с одним и тем же массивом. Иначе результат невозможно предсказать.

Результатом разности указателей будет целое число, соответствующее числу элементов между ячейками, на которые они ссылались.

При записи выражений с указателями нужно помнить о приоритетах операций. Например,

```
*ptr++=25;
```

Операции разадресации и инкремента имеют одинаковый приоритет и выполняются справа налево. Но инкремент используется в постфиксной форме, он выполняется после операции присваивания. Следовательно, сначала по адресу, записанному в ptr, поместится 25, а затем увеличится значение указателя, т.е. аналогично *ptr=25; ptr++;

Указатели и массивы

При использовании указателей происходит работа с адресом ячейки памяти и получение лишь косвенного доступа к ее содержимому.

Часто указатели используются при работе с массивами. Указатели и массивы логически связаны друг с другом. Имя массива является константой, содержащей адрес первого элемента массива. Вследствие чего значение имени массива не может быть изменено оператором присваивания или каким-либо другим оператором. Так, если объявлен массив

```
float temp[10];
```

то `temp==&temp[0]`

Используя операции сложения, вычитания инкремента и декремента к имени массива (или к указателю) можно передвигаться по элементам массива. Т.е. temp++ будет указывать на второй элемент массива, т.е. элемент с индексом 1. Таким образом, можно работать с массивами, не используя их стандартного обращения к собственным элементам. Используя операцию косвенной адресации можно обращаться к значениям элементов массива. Выражение *(temp+1) эквивалентно temp[1].

Это верно и для многомерных массивов. Предположим, есть описание:

```
int zippo [4][2];
```

Напомним, что первое число означает количество строк, второе – количество столбцов. Тогда zippo = = &zippo[0][0], а zippo+ 5 на zippo[2][1].

Обращаться к элементу массива можно и следующим способом:

```
*(zippo[ i ] + j) или *( * (zippo+ i ) + j)
```

Двумерный массив является одномерным массивом, элементы которого - массивы, следовательно, имя его первой строки zippo[0], а имя четвертой строки - zippo[3]. Однако имя массива является также указателем на этот массив в том смысле, что оно ссылается на его первый элемент. Следовательно,

```
zippo[0]= =&zippo[0][0]
zippo[2]= =&zippo[2][0].
```

Передача массива в функцию выполняется следующим механизмом: в описании функции в списке формальных параметров указывается тип элементов массива, его имя, а после пустые квадратные скобки, которые указывают, что данный аргумент является массивом. При вызове функции в списке фактических параметров указывается имя массива. Например,

```
int sum (int n, int array[ ]) { … }     // объявление функции
```

И тогда, в действительности, функция получает адрес первого элемента массива, и следовательно, она может быть вызвана и следующим образом:

```
int c = sum(15, &arr[0]);       // где int arr[15];
```

или более простой конструкцией:

```
int c = sum(15, arr);
```

В случае передачи в функцию в качестве параметра двухмерного массива в первой паре квадратных скобок размерность не указывается, а во второй паре – скобок должна быть указана обязательно. Например,

```
int sort(int ard[ ][7]);
```

В этом случае фактическим параметром также может служить имя массива.

При передаче в функцию, как одномерного массива, так и двумерного, можно в качестве формальных параметров использовать напрямую указатели, но при этом требуется следить, чтобы не выйти за пределы массива.

Пример.

Программа, вычисляющая значение $z = (a,b) - (b,c)(d,e)$, где a, b, c – векторы размерности 10; a, d и e векторы размерности 12. Запись вида $(x,\ y)$ означает скалярное произведение векторов. Координаты векторов хранятся в одномерных массивах.

```
#include<iostream>
#include <ctime>
#include <cstdlib>
#include <locale.h>
using namespace std;
int scalar (int* x1,  int* x2, int n);
```

```
// прототип функции для подсчета скалярного произведения
// два первых параметра – указатели на целый тип,  для передачи массивов
// третий параметр – размерности массивов
int main ( )
{
srand( time( NULL ) );
const int n = 10;
const int m =12;
 int a[n], b[n], c[n], d[m], e[m];
int i;
for ( i = 0; i < n; i++)
 {        a[ i ] = random (20) -10;
          b[ i ] = random (20) -10;
          c[ i ] = random (20) -10;
}
for ( i=0; i < m; i++)
 {        d[ i ] = random (20) -10;
          e[ i ] = random (20) -10;
}
int z = scalar(a, b, n) – scalar(b, c, n) * scalar(d, e, m);
cout << “z = ” << z;
}
int scalar (int* x1,  int* x2, int n)       // описание функции
{
int sc = 0;
for ( int k = 0; k < n; n++)
sc +=  *( x1+ k )* *(x2 + k );
return sc;
}
```

Поскольку одномерный массив является массивом, состоящим из массивов,

о чем уже говорилось, это позволяет использовать функцию, предназначенную для работы с одномерным массивом, для работы со строками двумерного массива.

Пусть описана функция нахождения среднего арифметического массива целых чисел:

```
float mean(int array[ ], int n)
{
int index,  sum=0;
 if (n>0)
                {
                 for (index=0,sum=0; index<n; index++)    sum+ = *(array + index);
                return (float)sum / n;          // явное приведение типа
                }
 else           {
                cout<<"Нет массива"<<endl;
                 return 0;
                }
}
```

Параметром функции является целочисленный массив. Поэтому сумма элементов также будет целой. Но в С++ при делении целого на целое результат также целый. Поэтому используется явное приведение типа. Заметим, что к типу float, приводим только первый операнд, и тогда при делении вещественного числа на целое получим вещественный результат. Если привести к типу float все выражение, т.е. (float)(sum / n); – то дробная часть будет равна нулю.

Данную функцию можно использовать для нахождения среднего арифметического строки матрицы:

```
int main ( )
{
 int nk[ 3] [4] = { {2,4,6,9}, {10, 20, 40,10}, {3, 7, 0, 9} };
for ( int line = 0; line < 3; line++)
printf ("Среднее арифметическое %d строки равно %d \n",line+1,mean(nk[line], 4);
```

```
}
```

Динамическая память

Во время компиляции программ на языках C/C++ память компьютера делится на четыре области: программного кода, глобальных данных, стек и динамическую область. Последняя отводится для хранения временных данных и управляется функциями распределения динамической памяти.

Создатели языка C++ посчитали оперирование свободной памяти столь важной задачей для работы программы, что добавили дополнительные операторы new и delete, аналогичные выше рассмотренным функциям. Аргументом оператора new служит выражение, возвращающее количество байтов, необходимых для резервирования. Этот оператор возвращает указатель на начало выделенного блока памяти. Аргументом оператора delete выступает адрес первой ячейки освобождаемого блока. Операторы new и delete являются встроенными компонентами языка, и подключение дополнительных библиотек не требуется.

```
#include <iostream>
#define SIZE 512
using namespace std;
int main ( )
{
int *pr_buffer;
pr_buffer=new int[SIZE];
if (pr_buffer ==NULL)        cout << «недостаточно памяти»;
        else             cout  << «память зарезервирована»;
delete (pr_buffer);
return 0;
}
```

Оператор new автоматически выполняет определение точного размера блока памяти, основываясь на заданном типе объекта. В примере резервируется 512 ячеек типа int.

В качестве аргумента можно использовать как стандартные типы, так и типы, определяемые пользователем.

В случае выделения памяти для массивов оператор new возвращает адрес первого элемента массива:

```
float &i;
i=new float[100];
```

или

```
int n=100;
float *p=new float[100];
```

В этих случаях в динамической памяти отводится непрерывная область, достаточная для размещения 100 элементов вещественного типа, и адрес ее начала записывается в указатель p или ссылку i. Динамические массивы нельзя при создании инициализировать, и они не обнуляются автоматически. Доступ к элементам динамического массива происходит обычным способом.

Преимущество динамических массивов заключается в том, что размерность может быть переменной, т.е. объем памяти, выделяемой под массив, определяется на этапе выполнения программы. При выделении памяти для многомерного массива все размерности, кроме первой, должны быть константами (константными выражениями). Первая размерность может быть задана переменной, значение которой будет известно к моменту использования оператора new. В этом случае указатель в левой части должен иметь правильный тип, например

```
int_ptr = new int[j][4][5];
int (*p)[ ] = new int[2][3];
```

Ссылки

Ссылка это еще одно название для указателя, который не требует разыменовывания при использовании. Разница между указателем и ссылкой заключается в том, что программист может использовать ссылку как обычный объект, несмотря на то, что к объекту будет производиться косвенный доступ, в то время как к указателю необходимо присвоить явно значение адреса объекта.

```
int i = 123;
int *p = &i;
```

Выражение &i является ссылкой на объект с именем i. Через адрес оно ссылается на объект с именем i.

C++ позволяет объявлять ссылочные переменные. Например, объявление &j = i; создает переменную j – ссылку(второе имя) для i .

```
int main( )
 {
int i = 3;
int  j = 2;
…….
}
```

После выполнения присваивания переменная i, также как и j, будет иметь значение 2.

Ссылки безопаснее указателей, поскольку адреса ссылок невозможно переприсваивать. Необходимо помнить что, однажды инициализировав ссылку, ей уже нельзя присвоить другое значение. В отличие от указателей ссылки всегда связаны с объектом. Они используются в качестве переменных, параметров и результатов функций.

Ссылки полезны в функциях, которые возвращают несколько значений. Ведь с помощью оператора return можно возвращать только одно значение. Например, рассмотрим функцию, меняющую между собой значения двух переменных.

```
#include <iostream>
using namespace std;
void exchange(int &a, int &b)
{
  int c = a;    a = b;    b = c;
}
int main( )
```

```
{
int a = 100, b = 10;
 cout << "До обмена: a =" << a << ", b = "<< b << endl;
exchange (a, b);
cout << " После обмена: a=" << a << ", b=" << b << endl;
return 0;
}
```

Здесь доступ к переменной осуществляется как через ее имя, так и через имя-синоним (ссылку) в вызываемой функции. После завершения вызываемой программы (функции) имя-синоним уничтожается, однако измененное значение переменной в вызываемой функции сохраняется. Таким образом, при обращении к переменным через ссылки появляется возможность работать с локальными переменными как с глобальными. Это позволяет возвращать из функции любое количество значений.

Рекомендации по созданию программы

Главная цель при создании программы с использованием структурного и процедурного методов программирования – получение легко читаемого кода возможно более простой структуры.

Первый шаг состоит в продумывании и записи алгоритма будущей программы на естественном языке или с использованием блок-схемы. Это позволяет продумать алгоритм в деталях, разбить программу на логические блоки, определить их последовательность, продумать комментарии к программе.

Необходимо стараться (если это возможно) разбить алгоритм на последовательность законченных действий. Каждое из них можно оформить в виде функции.

Функция не должна быть слишком большой, как правило, ее текст должен не превышать два экрана.

Если некоторые действия в программе повторяются несколько раз, их также требуется оформить в виде функции, что сделает программу нагляднее и сократит

ее размер.

При расположении последовательности функций нужно помнить, что функция может быть вызвана только после ее объявления. Если программа содержит две-три функции, такой порядок достаточно легко определить. Иначе лучше использовать прототипы функций, расположив их в начале программы, сразу после директив препроцессора.

Всю информацию, требуемую для работы функции, необходимо передавать в качестве параметров. В вызове функции строго соблюдать соответствие количества, типов и порядок следования фактических параметров формальным.

Правильный выбор имен улучшают читаемость программы. Для этого существуют несколько рекомендаций. Считается, что несколько первых символов имени должны объяснять содержимое переменной, тем самым, создавая документированность программы. Также популярны, так называемые венгерские конвенции (или венгерская запись имен), которые были разработаны программистом из Microsoft Ч. Симони. В соответствии с ними перед именем указателя следует помещать букву p, перед дальними указателями lp, перед функциями fn.

Обычно, чем больше область видимости переменной, тем ее имя длиннее. Для параметров коротких циклов лучше использовать однобуквенные имена. Имена макросов и констант предпочтительнее записывать заглавными буквами.

Нельзя использовать в качестве имен переменных имена типов и ключевые слова языка.

Желательно инициализировать переменные при их объявлении, а объявлять их как можно ближе к месту непосредственного использования. В небольших функциях удобно все объявления локальных переменных располагать в начале блока.

Если ввод переменных осуществляется через клавиатуру, требуется предварять его выводом сообщения на экран, которое обязательно должно быть информативным. Лучше предусмотреть и правильность ввода переменных. Например, при введении даты нужно проверить, чтобы число не превышало 31, а номер месяца 12.

Использование локальных переменных предпочтительнее глобальных. Если глобальная переменная необходима, лучше объявить ее статической, что ограничит область ее действия одним файлом. Изменение глобальных переменных сложно отслеживать в большой программе.

Не рекомендуется использования в программе чисел в явном виде. Лучше использовать константу. Особенно это актуально в случаях, неоднократного ее использования. Тогда при изменении кода программы, легко изменить ее значение. Константы также должны иметь осмысленные имена.

При использовании ветвления следует избегать проверки лишних условий. Неэффективным кодом считается проверка на равенство нулю. Например, вместо if (k == 0) лучше писать if (k).

При организации циклов лучше размещать инициализацию и приращения счетчика, проверку условия выхода из цикла в одном месте. Требуется предусматривать аварийный выход из цикла по достижению заданного максимального количества итераций.

Вложенные циклы и блоки должны иметь отступ друг от друга в три-четыре символа, причем блоки одного уровня вложенности должны быть выровнены по вертикали. Желательно, чтобы закрывающаяся фигурная скобка была расположена строго под соответствующей ей открывающейся.

Необходимо проверять коды возврата ошибок и предусматривать печать сообщений в тех точках программы, куда управление программой при ее нормальной работе передаваться не должно. Например, оператор switch должен иметь ветвь default, в случае, когда в нем не перечислены все возможные значения переключателя.

Сообщение об ошибке должно быть информативным и подсказывать пользователю методы ее исправления. Например, при вводе неверного значения в сообщении об ошибке должен быть указан допустимый диапазон.

Написанный программный код нужно тщательно отредактировать. Убрать ненужные фрагменты, сгруппировать описания, оптимизировать проверки условий, проверить условия выхода из циклов, проверить оптимальность разбиения на

функции.

Следует сопроводить программу комментариями, которые должны представлять собой правильные предложения без сокращений и со знаками препинания. Но они и не должны подтверждать очевидное. Если комментарий занимает несколько строк, лучше его разместить до комментируемого фрагмента. Абзацный отступ комментария должен соответствовать отступу комментируемого блока. Для улучшения читаемости можно помечать комментарием окончание длинного составного оператора.

Не следует размещать в одной строке программы множество операторов. Важно, чтобы строка программы не выходила за пределы экрана. Между крупными блоками и функция лучше располагать пустую строку.

Для грамотно написанной программы недостаточно подтверждения ее работы и даже получения верного результата. Программа должна иметь четкую структурированность, наглядность, читаемость, сопровождение комментариями, возможность легкой модификации, и желательно, грамотно разработанный, эффективный алгоритм.

Контрольные вопросы

1. Из каких базовых конструкций состоит программа?
2. Перечислите операторы ветвления?
3. Сколько направлений содержит вычислительный процесс при использовании оператора if?
4. В чем заключается отличие цикла с предусловием от цикла с постусловием?
5. Могут ли отсутствовать все три выражения, идущие в круглых скобках оператора for?
6. В каких случаях обусловлено использование оператора цикла for?
7. Может ли массив содержать данные разного типа?
8. С какой цифры начинается нумерация элементов массива в C++?

9. Если предполагается, что в строке символов будет храниться слово из 10 букв, какова длина такого массива при объявлении?

10. Сколько значений в языке C++ может возвращать функция?

11. Дайте определения формальных и фактических аргументов функции.

12. Что такое прототип функции?

13. Перечислите области действия идентификатора.

14. Какие переменные называются локальными?

15. Возможно ли объявление указателя, несвязанного с типом?

16. Какие операции допустимы при работе с указателями?

17. Какова связь массивов и указателей?

18. Допускается ли инициализация динамических массивов?

19. В каком качестве получили широкое применение ссылки?

20. Предпочтительнее использование локальных или глобальных переменных?

Библиографический список

1. Бабэ Б. Просто и ясно о C++. М.: БИНОМ, 1995. – 400 с.
2. Галаган Т.А., Соловцова Л.А. Язык программирования С++ в примерах и задачах. Практикум. Благовещенск: АмГУ. 2005. - 102 с.
3. Галаган, Т.А. Алгоритмические языки и программирование. Язык С++. Курс лекций (Рек. ДВРУМЦ) / Т.А. Галаган – Благовещенск: изд-во АмГУ, 2007. – 147 с.
4. Дейл Н., Уимз Ч., Хедингтон М. Программирование на С++. М.: ДМК. 2000. – 672 с.
5. Павловская Т.А. С/С++. Программирование на языке высокого уровня. СПб.: Питер, 2004. – 461 с.
6. Павловская Т.А., Щупак Ю.А. С/С++. Структурное программирование. Практикум. СПб.: Питер, 2004.- 239 с.
7. Павловская Т.А., Щупак Ю.А. С++. Объектно-ориентированное программирование. Практикум. СПб.: Питер, 2011. – 265 с.
8. Паппас К., Мюррей У. Программирование на С и С++. Киев: Издательская группа BHV. 2000. – 320 с.
9. Подбельский В.В. Язык Си++: Учебное пособие. – М.: Финансы и статистика. 2003. – 560 с.
10. Сайт о программирование https://metanit.com/cpp/

Содержание

Введение	3
Глава 1. Введение в программирование	4
Этапы решения задачи	4
Кодирование и запуск программы	6
Контрольные вопросы	7
Глава 2. Базовые конструкции языка C++	9
Состав языка	11
Алфавит языка	12
Переменные, идентификаторы	12
Типы данных	13
Операции и выражения	17
Константы	21
Комментарии	23
Ввод-вывод на экран. Введение в потоки ввода-вывода	23
Директива #include и пространство имен	26
Построение программы	28
Контрольные вопросы	34
Глава 3. Методы структурного программирования	35
Блоки и составные выражения	33
Использование блок-схем в разработке алгоритмов	37
Операторы ветвления	39
Инструкции перехода	45
Операторы цикла	46
Генератор псевдослучайных чисел	54
Массивы	56
Обработка одномерных массивов числовых данных	59
Многомерные массивы	67

Работа со строками 69
Работа по столбцам 71
Инициализация таблицы 72
Вывод таблицы 72
Примеры программ обработки двумерных массивов 73
Библиотеки stdio.h и conio.h 77
Символьные строки 79
Функции 83
Рекурсия 88
Область действия и время жизни переменных 91
Указатели 95
Операции с указателями 97
Указатели и массивы 99
Динамическая память 103
Ссылки 104
Рекомендации по созданию программы 106
Контрольные вопросы 109

Татьяна Алексеевна Галаган,
доцент кафедры ИиУС АмГУ

Программирование на языке С++. Часть 1
