## **Тема 2.5. Основные конструкции языков программирования**

### **[2.5.1. Базовые средства и конструкции Quick Basic](#_2.5.2.1._Базовые_средства)**

### 2.5.1.1. Базовые средства Quick Basic

### 2.5.1.2. Средства программирования алгоритмов линейной структуры

### 2.5.1.3. Средства программирования разветвляющихся структур

### 2.5.1.4. Средства программирования регулярных циклических структур

### 2.5.1.5. Средства программирования итеративных циклических структур

### 2.5.1.6.Средства описания и работы с одномерными массивами данных

### 2.5.1.7. Средства описания и работы с двумерными массивами данных

### 2.5.1.8. Строковые данные и операции над строками

### 2.5.1.9. Средства процедурного программирования в среде QB

### [**2.5.2. Базовые средства и конструкции С++**](#_2.5.2._Базовые_средства)

### 2.5.2.1. Базовые средства языка С++

### 2.5.2.2. Средства программирования алгоритмов линейной структуры и

### функций

2.5.2.3. Средства программирования разветвляющихся структур

### 2.5.2.4. Средства программирования регулярных циклических структур

### 2.5.2.5. Средства программирования итеративных циклических структур

### 2.5.2.6. Средства описания и работы с одномерными массивами данных

### 2.5.2.7. Средства описания и работы с двумерными массивами данных

### 2.5.2.8. Строковые данные и операции над строками

### **[2.5.3. Контрольные вопросы по теме «Основные конструкции языков](#_2.5.3._Контрольные_вопросы)**

### **[программирования» (QB)](#_2.5.3._Контрольные_вопросы)**

### **[2.5.4. Контрольные вопросы по теме «Основные конструкции языков](#_(вставить_из_лаб.работ!))**

### **[программирования» (C++)](#_(вставить_из_лаб.работ!))**

### **[2.5.5. Тестовые задания по теме «Основные конструкции языков](#_(вставить_из_лаб.работ!))**

### **[программирования» (QB)](#_(вставить_из_лаб.работ!))**

### **[2.5.6. Тестовые задания по теме «Основные конструкции языков](#_2.5.6._Тестовые_задания)**

### **[программирования» (C++)](#_2.5.6._Тестовые_задания)**

# **2.5.1. Базовые средства и конструкции Quick Basic**

### 2.5.1.1. Базовые средства Quick Basic

**Алфавит языка**. Алфавит языка **Quick Basic (QB)** включает в себя прописные (**A-Z**) и строч­ные (**a-z**) латинские буквы, цифры **0-9** и специальные символы:

* знаки арифметических операций: + (плюс); - (минус); \* (умножение); / (деление); \ (деление нацело); ^ (возведение в степень);MOD (получение остатка от деления целых чисел);
* знаки операций отношения: **=** (равно); **<** (меньше); **>** (больше); **<=** (меньше или равно); **>=** (больше или равно); **<>** (не равно);
* знаки препинания и разделители:**,** (запятая); **.** (точка); **:** (двоеточие); ; (точка с запятой); ‘ (апостроф); () (круглые скобки); **\_** (подчеркивание); **пробел**;
* символы объявления типа данных: **%; &; !; # ; $**.

Строчные и прописные буквы в **QB** не различаются, поэтому, например, слова **tex** и **TEX** компилятором воспринимаются одина­ково.

Программа на языке **QB** содержит одну или несколько строк исходного текста. Каждая строка может содержать один или несколько операторов - специальных синтаксических конструк­ций, содержащих одно или несколько служебных слов, определяю­щих характер действия (ввести данные, проверить условие и т.п.). Операторы в одной строке разделяются двоеточием. Количество операторов в одной строке не ограничено. Однако принято в строке размещать один оператор, за исключением тех случаев, когда правила языка, требуют использования нескольких операторов, или операторы в строке образуют логически неразрывную последова­тельность.

Программная строка может содержать **метку** - последователь­ность букв или цифр, начинающуюся с буквы; в этом случае метка должна быть единственной в строке и отделяться от по­следующего текста двоеточием. Строка может начинаться с номера, являющегося целым числом в диапазоне **0-65535**. Метки и номера служат для ссылок на некоторые строки. Максимальная длина строки - **256** символов. Однако принято ограничивать ее длину **80** символами, что связано с удобством представления информации на экране.

Программа может содержать **комментарии** (примечания). Ком­ментарий может располагаться в отдельной строке или в конце программной строки. Начало текста комментария отмеча­ется символом апостроф ( **′** ) или оператором **REM**.

Некоторые слова в **QB** являются **зарезервированными** (ключевыми). К таким зарезервированным словам относятся **имена ко­манд, операторов и встроенных функций.** В дальнейшем мы будем выделять зарезервированные слова пропис­ными буквами, а для всех остальных конструкций использовать строчные буквы. Весь текст программы рекомендуется вводить строчными буквами.

**Классы, типы данных и способы их объявления**. Основная цель любой программы состоит в обработке данных, Данные различного типа хранятся и обрабатываются по-разному.

Тип данных определяет:

1. внутреннее представление данных в памяти компьютера;
2. операции и функции, которые можно применять к величинам этого типа;
3. множество значений, которые могут принимать величины этого типа.

Программы на языке **QB** оперируют с двумя основными классами данных: числовыми и строковыми. Числовые данные используются для представления и обработки в программе целых и действи­тельных чисел. Строковые данные используются для хранения и обработки последовательностей произвольных символов.

**Числовые данные** могут иметь один из четырех типов: целые обычной длины (**INTEGER**), целые двойной длины (**LONG**), веще­ственные обычной точности (**SINGLE**) и вещественные двойной точности (**DOUBLE**). Данные обычной и двойной длины различаются объемом занимаемой памяти и, соответственно, диа­пазоном возможных значений.

В табл. 2.5.1-1 приведены основные характеристики числовых данных. Суффикс является признаком типа данных и может быть использован для его описания.

Таблица 2.5.1-1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Тип** | **Длина**  **(байт)** | **Точность**  **(дес.цифр)** | **Диапазон**  **возможных значений** | **Суффикс** |
| **INTEGER** | 2 | - | -32768 - +32767 | **%** |
| **LONG** | 4 | - | -2147483648 - 2147483647 | **&** |
| **SINGLE** | 4 | 7 | -3,40\*1038 - +3,40\*1038 | **!** |
| **DOUBLE** | 8 | 15 | -1,79\*10308 - +1,79\*10308 | **#** |

**Строковые данные** или просто строки имеют тип **STRING**. Каждый символ строки занимает **1 байт** памяти. Длина строки не может превышать **256 байт**. Суффиксом строковых данных служит символ **$** (доллар).

Данные обоих классов разбиваются на **константы** и **переменные**.

**Константа** - это область памяти, которая не изменяет своего значения в процессе выполнения программы. Константы вещественных типов могут быть представлены в двух форматах: **с фиксированной точкой** и **с плавающей точкой**. Константа с фиксированной точкой содержит десятич­ную точку между целой и дробной частью числа. Например, **1.2345, -0.001**. Константа с плавающей точкой - это число, представленное в **экспоненциальной форме** и содержащее **мантиссу** и **порядок**.

**Мантисса** – это целое число или число с фиксированной точкой, записывается слева от знака экспоненты (буквы **E** или **D**).

**Порядок** – целое число справа от знака экспоненты – является показателем степени числа **10**. Значение константы определяется как произведение мантиссы и результата возведения числа **10** в степень порядка.

Например, значением константы **12.345E2** является число **12.345\*102=1234,5**, а значением константы **314159D-5** яв­ляется число **314159\*10-5=3,14159**.

**Строковая константа** - это последовательность символов, заключенная в двойные кавычки. Например, **“3.1415”**, **“Сумма”**, **“Quick BASIC”**. Строковую константу могут состав­лять любые символы клавиатуры. Исключение состав­ляет сам символ - **“**.

Язык **QB** допускает использование именованных констант, которые могут заменять в программе числовые и строковые величины. Чтобы использовать именованную константу, ее необходимо объявить при помощи ключевого слова **CONST**:

**CONST имя\_константы=выражение[,имя\_константы=выражение]...**

Например:

**CONSTpi=3.14159, message$=“Для продолжения нажми клавишу”**.

**Переменная** - это объект памяти, значение которого может изменяться в ходе выполнения программы. Каждая переменная характеризуется **именем**. **Имя переменной** может содержать до **40** символов (латинских букв, цифр, десятичных точек и символов определения типа - **%, !, &, #, $**) и должно начинаться с буквы, например, **alfa, x2, sum.of.length**. **Имя переменной** не должно быть зарезервированным словом или совпадать с именем именованной константы.

Тип переменной должен быть объявлен до ссылок на эту перемен­ную. Объявление типа переменной может быть выполнено одним из трех способов:

1. **Объявление** типа переменной с помощью **суффикса,** то есть с помощью соответствующего символа (суффикса), добавляемого к имени переменной при каждой ссылке на нее, например,  **k%, factorial&, summa!, integral#, str$** .
2. **Объявление** типа переменной **по первой букве ее имени** с помощью оператора **DEF,** имеющего следующий формат**:**

**DEFтип {буква|диапазон\_букв}[,{буква|диапазон\_букв}]...,**

где **тип** представляет собой одно из следующих зарезервированных слов: **INT** (тип **INTEGER**), **LNG** (тип **LONG**), **SNG** (тип **SINGLE**), **DBL** (тип **DOUBLE**), **STR** (тип **STRING**);  **буква** - первая буква имен переменных; **диапазон\_букв** - диапазон букв в алфавитном порядке, к которому относится данное объявление, где буквы разделены знаком - (минус).

Например, **DEFINTi-n** или **DEFSTRa**, **r-t** .

1. **Явное объявление** типа переменной в операторе **DIM:**

**DIM имя AS тип [, имя AS тип], ...**

Например,

**DIMstr1AS STRING**, **indexAS INTEGER**, **rootAS DOUBLE**.

По умолчанию тип переменной устанавли­вается **SINGLE**.

**Арифметические выражения и встроенные функции.** Арифметическое выражение представляет собой совокупность операндов: числовых констант, пере­менных и функций, связанных знаками арифметических операций и скобками. Арифметические выражения во многом похожи на алгебраи­че­ские формулы с некоторыми отличиями, определяемыми необходи­мостью записи любой формулы на одной строке. В табл. 2.5.1-2 приведен перечень арифметических операций, используемых в QB, в порядке убывания.

Кроме традиционных арифметических действий, таких как, сложение, вычитание, умножение, деление и возведение в степень, при записи арифметических выражений используются операции целочисленного деления и взятие остатка от целочисленного деления. При выполнении целочисленного деления отбрасывается дробная часть результата. Так, **46\8** равно **5**, а **9\10** равно **0**. Операция **MOD** (остаток или деление по модулю) заключается в вычислении остатка от деления двух целых чисел. Так, **46 MOD 8** равно **6**, а **9 MOD 10** равно **9**.

Таблица 2.5.1-2

|  |  |  |
| --- | --- | --- |
| **Операция** | **Назначение** | **Пример** |
| **^** | Возведение в степень | а^4 |
| **-** | Изменение знака числа | -a |
| **\*, /** | Умножение, деление | a\*b/76 |
| **\** | Целочисленное деление | 46\8 |
| **MOD** | Остаток или деление по модулю | 46 MOD 8 |
| **+, -** | Сложение, вычитание | 4+8-к1 |

Порядок выполнения действий соответствует приведенным выше приоритетам. Операции с одинаковым приоритетом вы­полняются слева направо. Порядок выполнения действий может изменяться только скобками. При вычислении арифметических выражений, в которые входят операнды различных типов, происходит преобразование операндов к типу операнда с наивысшей точностью.

Используемые в арифметических выражениях функции называют **арифметическими встроенными функциями**. Они представляют собой специальные стандартные программы (процедуры), которым передается один или несколько аргументов и которые возвращают вычисленное значение через имя функции. Полный список арифметических встроенных функций **QB** приводится в табл. 2.5.1-3

Таблица 2.5.1-3.

|  |  |
| --- | --- |
| **Имя функции** | **Результат выполнения функции** |
| **ABS(x)** | Абсолютная величина числа **х** |
| **SQR(x)** | Корень квадратный из числа **х>=0** |
| **EXP(x)** | Число е (2.718282) в степени **х** |
| **LOG(x)** | Натуральный логарифм числа **х>0** |
| **SIN(x)** | Синус числа **х** |
| **COS(x)** | Косинус числа **х** |
| **TAN(x)** | Тангенс числа **х** |
| **ATN(x)** | Арктангенс числа **х** (от **–π/2** до **+π/2**) |
| **SGN(x)** | Знак числа **x,** результат:**1**, при **x>0**;**0**, при **x=0**;  **-1**, при **x<0** |
| **RND(x)** | Число из случайной последовательности в диапазоне от **0** до **1**. |
| **INT(x)** | Ближайшее целое, не превосходящее **х** |
| **FIX(x)** | Целая часть **х** |

Сделаем несколько пояснений относительно использования функции**RND(x). RND(x)** – математическая функция, которая возвращает случайное число в интервале от **0** до **1**. Чтобы при каждом запуске программы генерировалась разная последовательность случайных чисел, следует использовать оператор **RANDOMIZETIMER**. Если у функции **RND(x)** аргумент **x≥0** или отсутствует, то возвращается следующее число из последовательности случайных чисел. Если же аргумент **x<0**, то функция **RND(x)** будет всегда возвращать одно и то же число для этого аргумента.

Обращение ко всем функциям всегда происходит по их именам, за которыми в скобках следуют аргументы. Вычисле­ние функций в арифметическом выражении предшествует выполне­нию арифметических операций. Аргументом встроенной функции может быть любое ариф­метическое выражение.

### 2.5.1.2. Средства программирования алгоритмов линейной структуры

**Алгоритмы линейной структуры** содержат базовые алгоритмические структуры только последовательного типа. Соответственно, при программировании алгоритмов линейной структуры все операторы программы выполняются последовательно в порядке их записи. Программа такого типа назы­вается **последовательной**. Последовательные программы содержат опера­торы, выполняющие: ввод исходных данных, последовательную обработку данных и получение результатов, вывод результатов на экран (или печать).

В языке **QB** отсутствует оператор, соответствующий действию начала алгоритма, однако имеется оператор **END**, который должен располагаться в последней строке программы, означающий конец программы. Этот оператор является необязательным и может отсутствовать, тогда концом программы служит последний оператор текста программы. Однако, если вставить оператор **END** после логически завершенного участка программы, то выполнение происходит только до этого оператора. На практике этот прием часто используется для отладки участков программы.

Простейшим и вместе с тем фундаментальным средством обработки данных в языке **QB** является **оператор присваивания**, имеющий следующий формат:

**[LET] переменная = выражение**.

Здесь **переменная** - имя переменной любого типа, **=** это знак операции присваивания, а **выражение** – выражение любого допустимого типа. Ключевое слово **LET** обычно опускается. Ограничимся пока рассмотрением случаев, когда перемен­ные в левой части операторов присваивания имеют один из чис­ловых типов. Тогда выражение в правой части оператора при­сваивания может быть только арифметическим.

При выполнении оператора присваивания сначала вычисляется **выражение** справа от знака **=**, затем результат вычисления, преоб­разованный к типу **переменной** в левой части оператора, запоми­нается в **переменной**, стоящей слева от знака **=**. Например:

**k = 0** - присваивание переменной **k** значения константы **0**;

**x = 0.5** - присваивание переменной **x** значения константы **0.5**;

**y=SIN(x)+1 -** присваивание переменной **y**значения вычисленного выражения;

**z = y -** присваивание переменной **z** значения переменной **y**;

**k = k+1** - увеличение переменной **k** на **1**.

При работе с оператором присваивания необходимо учитывать особенности преобразования типа во время присваивания вычисленного значения выражения. Например, при преобразовании вещественных значений к целому типу происходит округление по правилам математики.

В табл. 2.5.1-4. приведены примеры преобразования типа данных при выполнении операторов присваивания.

Таблица 2.5.1-4

|  |  |  |
| --- | --- | --- |
| Оператор | Результат | Пояснения |
| **a%=12.44** | **12** | Вещественная константа преобразована в целое значение |
| **a% =12.77** | **13** | Округление при преобразовании к целому типу |
| **a# = 6#/7** | **0.8571428571428571** | Действия и результат с двойной точностью |
| **a = 6#/7** | **0.8571429** | Результат с обычной точностью (округлен) |

Для ввода исходных данных с клавиатуры используется опера­тор **INPUT**. Он имеет следующий формат:

**INPUT [;] [подсказка { ; | , }] список\_переменных**

Здесь **подсказка** - строковая константа, выводимая на экран для пояснения - что требуется ввести; **списокпеременных** - список, содержащий перечень имен переменных, разделенных запятыми, которым будут присвоены соответствующие значения в порядке их ввода с клавиатуры. После подсказки должен следовать символ точка с запятой (после подсказки на экран выводится знак вопроса) или запятая (знак вопроса не выводится). Если после слова **INPUT** указать точку с запятой, то после ввода данных курсор остается в текущей позиции. При выполнении программы оператор **INPUT** выводит на экран подсказку, а пользова­тель должен ввести с клавиатуры данные в порядке и количестве, указанном в операторе **INPUT**, отделяя одно данное от дру­гого запятыми, а по окончании ввода нажать клавишу **Enter**. Если подсказка отсутствует (это допустимо), то зачастую сложно понять, что и в каком порядке следует вводить. Если при вводе допущена ошибка, то на экране появляется сообщение **“?Redo from start”** (повторить сначала).

Например,

**INPUT “Введите коэффициенты уравнения a, b, c “, a#, b#, c#**

**INPUT ;“Радиус круга”; r!**

При вводе строковых дан­ных их значения могут не заключаться в кавычки. Если кавычки использо­ваны, то строковый элемент будет состоять из всех символов между кавычками.

Если набор данных не меняется от одного выполне­ния программы к другому, то для его ввода могут быть исполь­зованы операторы **READ** и **DATA**. При этом оператор **DATA** является неисполняемым оператором, и может находиться в любом месте программы. Исключение составляет оператор цикла, внутри которого **DATA** располагаться не может. Он содержит список вводи­мых значений, разделенных запятыми. Оператор **READ** является исполняемым оператором. Он содержит список переменных, разделенных запятыми, которым присваи­ваются соответствующие им значения из списка оператора **DATA**. Операторы имеют следующие форматы:

**DATA список\_значений**

**READ список\_переменных.**

Количество и типы переменных в операторе **READ** должны строго соответствовать количеству и типам данных соответ­ствующего оператора **DATA**. Например,

**READff, stroke$, t1%**

**DATA 2.5, “QuickBASIC”, 10000.**

Один оператор **READ** может работать не с одним, а с несколькими операторами **DATA**, в этом случае считывание данных произво­дится в порядке следования операторов **DATA.**

Для вывода данных на экран монитора служит оператор **PRINT**, который при вводе с клавиатуры может быть заменен символом **?**:

**PRINT [список] [ {,|;} ]**

Выражения, указанные в списке, могут быть как арифметическими, так и строковыми, разделенными запятыми или точками с запятой. Позиция каждого выводимого на экран элемента определяется символом-разделителем. **QB** делит строку на пять зон по **14** сим­волов. Если разделитель - запятая, то очередной эле­мент списка выводится в начале следующей зоны, если - точка с запятой, то элемент списка выводится непо­средственно в следующую символьную позицию, причем числовые данные дополняются в конце одним пробелом, а для положительных чисел перед числом (вместо знака +) добавляется пробел. Например, оператор **PRINT “k=“; k, “a=“; a**

при **k=35** и **a=-1.25** выведет на экран, начиная с первой позиции, следующую строку: **k= 35 a=-1. 25.**

Если **|x| > 9999999,** то значение **x** выводится в виде чисел с плавающей точкой.

Если в конце списка стоит запятая или точка с запятой, то следующий оператор **PRINT** производит вывод в той же самой строке. Если в операторе **PRINT** отсутствует список данных, то выводится одна пустая строка. Это используется для пропуска строки или для перехода на следующую строку.

В операторе **PRINT** для управления позицией вывода очеред­ного элемента списка могут быть использованы функции **SPC(n)** и **TAB(n),** где **n** - арифметическое выражение целого типа. Функция **SPC(n)** разме­щает **n** пробелов после текущей позиции курсора, а функция **TAB(n)** помещает курсор в позицию **n**. При этом, если номер позиции **n** меньше текущей позиции курсора, то курсор помещается в следующей строке позиции **n**. Например, оператор

**PRINT TAB(10);”Quick”;SPC(5);”BASIC”**

выведет на экран строку

**Quick BASIC**

Для вывода на принтер используется оператор **LPRINT**, формат и порядок выполнения которого аналогичны рассмот­ренному выше оператору **PRINT.**

Для перемещения курсора в нужную позицию экрана использу­ется оператор **LOCATE**, имеющий следующий формат:

**LOCATE [строка], [колонка]**

где **строка** - номер строки экрана в диапазоне от **1** до **25**, а **колонка** - номер горизонтальной позиции символа в диапазоне от **1** до **80**.

Оператор **PRINT**, следующий за оператором **LOCATE**, выводит символы на экран, начиная с ука­занной позиции. Чтобы вывести на печать строку из одинаковых символов, часто используют функцию **STRING$(n, символ),** которая формирует строку из **n** символов.

Выполнение любой программы рекомендуется начинать с оператора **CLS**, выполняющего очистку экрана от результатов выполнения предыдущей про­граммы.

Ниже приведены примеры программ последовательного типа, использующие рассмотренные выше операторы.

### 2.5.1.3. Средства программирования разветвляющихся структур

На практике редко встречаются задачи, решение которых представляется только в виде алгоритмов линейной структуры. Как правило, приходится решать задачи, в которых в зависимости от выполнения некоторого логического условия необходимо органи­зовать вычисления по одной из нескольких ветвей алгоритма. Алгоритмы решения таких задач строятся с использованием раз­ветвляющихся алгоритмических структур. В результате проверки ***логического выражения*** (“**Истина**” или “**Ложь**”) управление передается ветке “**Да**” или “**Нет**”.

Логические выражения, в отличие от арифметических, содержат ***операции отношения*** (**=, <, >, <=, >=, <>**), а также еще пять ***логических операций***, записанных далее в порядке убывания приоритета: отрицание - **NOT**, логическое умножение (логическое “И”) - **AND**, логическое сложение (логическое “ИЛИ”) - **OR**, исключающее “ИЛИ” - **XOR**, эквивалентность - **EQV**.

Результатом выполнения логического выражения может быть “**Истина**” (**не 0)** или “**Ложь**” (**0**). Например, **NOT 7<9** всегда прини­мает значение “**Ложь**”, что соответствует значению **0**, а **4>3 AND 7<9** принимает значение “**Истина**”, что соответствует **-1**.

В табл. 2.5.1-5 приведено описание логических операций. Здесь **T** (**true**) - “**Истина**”, а **F** (**false**) “**Ложь**”.

Таблица 2.5.1-5

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Значения операндов** | | **NOT x** | **x AND y** | **x OR y** | **x XOR y** | **х EQV y** |
| X | Y |  |  |  |  |  |
| T | T | F | T | T | F | T |
| T | F | F | F | T | T | F |
| F | T | T | F | T | T | F |
| F | F | T | F | F | F | T |

 Разветвляющиеся алгоритмические структуры в языках про­граммирования реализуются **операторами условного перехода*.*** В **QB** для программирования условных переходов используются три типа операторов: **однострочный IF**, **блочныйIF** и оператор выбора **SELECT CASE**

**Однострочный оператор IF** имеет следующий формат:

**IF выражение THEN оператор(ы) [ELSE оператор(ы)]**

Порядок выполнения оператора следующий: если значение выражения не нуль (“**Истина**”), то выполняется оператор (или операторы), стоящий после **THEN**, в противном случае выполняются операторы, стоящие после **ELSE**. Конструкция **ELSE** может от­сутствовать (в формате она взята в квадрат­ные скобки) – случай усеченного разветвления.

**Выражение** может быть логическим или арифметическим. Ненулевое значение арифметического выражения интерпретируется как “**Истина**”, а нулевое - как “**Ложь**”.

Особенностью однострочного оператора **IF** является то, что он должен располагаться в одной строке программы.

Приведем примеры записи однострочного оператора **IF**:

**IF x > 0 THEN y=SQR(x) ELSE y=1**

**IF d THEN epsilon=0.001**

**IF lambda < 1 OR beta = 3 THEN PRINT lambda, beta : x=0**

**IF x < 10 THEN y=1 ELSE IF x < 100 THEN y=2 ELSE y=4**

Если запись оператора не умещается в пределах экрана, а также при программировании вложенных разветвлений, используется **блочный** (многострочный) оператор **IF**. Он имеет следующий формат:

**IF выражение\_1 THEN  
     [ блок\_операторов\_1 ]  
 [ ELSEIF выражение\_2 THEN  
     [ блок\_операторов\_2 ]  
 . . .  
 [ ELSE  
     [ блок\_операторов\_n ] ]  
 END IF**

Отметим, что здесь и в дальнейшем операторы, записанные в квадратных скобках, могут отсутствовать.

Порядок выполнения оператора следующий: если значение **выражения\_1** “**Истина**” - выполняется **блок\_операторов\_1** и далее управление передается оператору, следующему после **END IF**. Если **выражение\_1“Ложь**”, то проверяется значение следующего выражения и так далее. После выполнения любого блока операторов управление всегда передается оператору, следующему после **END IF**. Конструкций типа **ELSEIF** может быть несколько. **Блок\_операторов\_n,** следующий за**ELSE,** выполняется в том случае, если последовательные проверки всех выражений выше дали значение “**Ложь**”. Конструкции **ELSEIF** и **ELSE** могут отсут­ствовать, что и отмечается в синтаксисе оператора квадратными скобками. Особенностью оператора является то, что слово **THEN** должно быть последним в строке, а слово **ELSE** - единственным. Слово **ELSEIF** записывается слитно, а слова **END IF** - через пробел.

Примеры блочных операторов **IF**:

|  |  |  |  |
| --- | --- | --- | --- |
| 1) | **IF X>0 THEN**  **y=SQR(x)**  **z=1+y**  **p=4**  **END IF** | 2) | **IF x < 10 THEN     y=1 ELSEIF x < 100 THEN      y=2 ELSEIF x < 1000 THEN   y=3**  **ELSE**  **y=4**  **END IF** |

Оператор **SELECT CASE**  позволяет осуществить разветвление по любому фиксированному числу ветвей. Он имеет следующий формат:

**SELECT CASE выражение  
 CASE тест\_1  
 [блок\_операторов\_1]  
 [CASE тест\_2  
 [блок\_операторов\_2]]**

**. . .  
 [CASE ELSE  
 [блок\_операторов\_n]]  
 ENDSELECT**

Здесь **выражение**– арифметическое или строковое, а **тест\_i** может иметь одну из трех форм:

1)**выражение [, выражение . . .]**

2)**выражение TO выражение**

3)**IS операция\_отношения выражение**

Например, строки **CASE** могут содержать следующие тесты:

**CASE IS >3**‘проверка на превышение значения 3;  
**CASE x^2** ‘проверка на равенство значению выражения;  
 **CASE 5 TO 7**  ‘ проверка на принадлежность отрезку [5;7];  
**CASE 1 TO 4,  7 TO 9,  11, 13,  IS > Max**.

 В последнем примере через запятую перечисляются несколько тестов, при этом результат проверки выражения считается ис­тинным, если выполняется хотя бы один из тестов.

Порядок выполнения оператора **SELECT CASE** следующий: если значение **выражения** удовлетворяет **тесту** после конструк­ции **CASE**, то выполняется **блок\_операторов**, а затем управление передается оператору, следующему после конструкции **END SELECT**. В противном случае проверяется следующий **тест** и так далее. Все конструкции **SELECT CASE** рабо­тают аналогично. **Блок\_операторов\_n**, расположенный после **CASE ELSE** выполняется в том случае, если последовательные проверки всех **тестов** выше дали значе­ние “**Ложь**”.

Оператор имеет следующие особенности: **тест\_1**, **тест\_2** и так далее должны быть последними в строке, а конструкция **CASE ELSE** должна быть единственной в строке.

### 2.5.1.4. Средства программирования регулярных циклических структур

**Алгоритмы регулярной циклической структуры** характеризуются заданным (известным) числом повторений: иногда это число за­дается явно, иногда его предварительно вычисляют. К таким ал­горитмам относятся: алгоритмы вычисления значений функций одной или нескольких переменных, у которых аргументы изменяются в известном диапазоне и с заданным шагом изме­нения; алгоритмы вычисления конечных сумм и произведений и многие дру­гие. Программирование такой алгоритмической структуры на языке **QB** производится при помощи оператора **FOR...NEXT**, имеющего следующий формат:

**FOR параметр=выражение1 TO выражение2 [STEP выражение3]      блок\_операторов**

**[EXITFOR]  
NEXT параметр**

Здесь **параметр** цикла – это переменная, для которой **выражение1** является начальным значением, **выражение2** – конечным значением, а **выражение3 -** шаг изменения параметра цикла. Причем **шаг** изменения параметра может быть как положительным, так и отрицательным. По умолчанию его значение равно **1**. **Блок операторов**, находящихся между конструкциями **FOR** и **NEXT**, называется **телом цикла**, количество повторений которого опре­деляется числом возможных значений параметра цикла. Перед каждым повторением цикла, в том числе первым входом в цикл, текущее значение **параметра** цикла сравнивается со значением **выражение2**  (конечным значением). После каждого выполнения операторов **тела цикла** к переменной (**параметру** цикла) прибавляется значение **выражение3** (шаг) и снова производится сравнение параметра с конечным значением. Тело цикла перестает выполняться, когда значение параметра станет больше конечного значения (шаг положителен) или меньше конеч­ного значения (шаг отрицателен). Если условие выхода из цикла выполняется при начальном значении параметра, то тело цикла пропускается, и управление сразу же передается оператору, следующему за **NEXT**. Для выхода из цикла до его завершения мо­жно использовать оператор **EXIT  FOR**.

При организации вложенных циклов (цикл в цикле) используются пары операторов **FOR…NEXT** , которые не должны “пересекаться” друг с другом. Другими словами, каждой конструкции **FOR** должна соответствовать конструкция **NEXT**. При этом конструкция **NEXT** для внут­реннего цикла должна предшествовать конструкции **NEXT** внешнего цикла. Для преждевременного выхода из цикла по условию используется оператор **EXIT FOR**. Например:

**FOR i = 1  TO  10  
  FOR j = 1  TO  2**

**'** *тело внутреннего цикла*

**IF g=2 THEN  EXIT  FOR  
    NEXT  j**

**NEXT i**

### 2.5.1.5. Средства программирования итеративных циклических структур

Отличительной особенностью алгоритмов ***итеративных циклических структур*** является заранее неиз­вестное число повторений цикла. К ним от­носятся, например, алгоритм проверки ввода исходных данных, алгоритмы вычисления отдельных членов, суммы или произведения членов бесконечных последо­ва­тельностей, алго­ритмы вычислений по итерационным формулам и некоторые другие алгоритмы. При программировании алгоритмов итеративных циклических структур ис­пользуется оператор **DO…LOOP** в одном из двух возможных форматов.

Первый формат оператора **DO…LOOP** используется при программировании итеративной циклической структуры с **предусловием**:

**DO [{ WHILE | UNTIL } логическое\_выражение]**

**блок\_операторов**

**[EXIT DO]**

**LOOP**

Второй формат оператора **DO…LOOP** реализует структуру с **постусловием**:

**DO   
 блок\_операторов**

**[EXIT DO] LOOP [{ WHILE | UNTIL } логическое\_выражение]**

При использовании конструкции **WHILE** тело цикла будет вы­полняться до тех пор, пока значение **логического\_выражения** истинно. Как только это выражение становится ложным, происходит выход из цикла и управление передается оператору, следующему за **LOOP**. При использовании конструкции **UNTIL** тело цикла выполняется до тех пор, пока выражение оста­ется ложным. Как только оно становится истинным, происходит выход из цикла и управление передается оператору, следующему за **LOOP**. Поэтому, после **WHILE** записывается условие продолжения цикла, а после **UNTIL** - условие окончания цикла.

Если конструкция **WHILE** или **UNTIL** в операторе цикла опущена, то имеет место бесконеч­ный цикл, выход из которого на оператор, следующий после **LOOP**, осуществляется только при использовании оператора **EXIT DO**. Оператор **EXIT DO** может ис­пользоваться и при наличии конструкций **WHILE** и **UNTIL** для орга­низации досрочного выхода из цикла по некоторому дополнительному усло­вию.

### **2.5.1.6.** Средства описания и работы с одномерными массивами данных

***Массив***–это последовательность переменных одинакового типа, объединенных общим ***именем***. Массив состоит из ***элементов,***например: ***одномерный массив* а(50) состоит из 51 элемента:a(0)**, **a(1) a(2) a(3) . . . a(50).**

. В программе элементы одного и того же массива отличаются один от другого при помощи индексов. **Индексы** записываются вслед за именем в круглых скобках и могут представлять собой любое целочислен­ное выражение. Например: **d(24), a(2\*i+1), b(i,i-1).**

Количество индексов указывает ***на размерность*** массива. Так, в приведенном выше примере размерность массива **a(50)** равна единице, то есть массив одномерный. Максимальная размер­ность массива может достигать **60.**

В отличие от размерности, ***размер массива*** - это количество элементов в массиве. В нашем примере размер массива **а(50)** равен **51 (** с 0-го по 50**)**.

Перед использованием массива в программе его необходимо объявить с помощью оператора **DIM**.Этот оператор выделяет место в памяти для размещения элементов массива. Онимеетследующийформат:

**DIM v1([imin TO] imax) AS type [, . . . vn([imin TO] imax) [AS type]**

где: **v1,...,vn** - имена описываемых массивов; **imin, imax** - нижняя и верхняя границы изменения индекса (если **imin** опущено, то нижняя граница по умолчанию принимается равной нулю - **0**); служебное слово **AS** с параметром **type** устанавливает тип массива **(INTEGER, LONG, SINGLE, DOUBLE, STRING)**.

Оператор **DIM** не только объявляет, но и обнуляет элементы числовых массивов или запол­няет элементы строковых массивов пустыми строками (**“”**). Если одномерный массив, используемый в программе, не объявлен оператором **DIM**, то верхняя граница его индексов устанавливается равной **10**.

Различают два способа распределения памяти: **статическое** - на этапе компиляции до выполнения программы, и **динамическое** - в ходе вы­полнения программы. По умолчанию массив, границы которого заданы константными выражениями, считается статическим. Па­мять для размещения такого массива выделяется на этапе компиляции программы и сохраняется за ним на весь период выполнения программы. На­пример:

**DIM sigma(1 TO 5) AS INTEGER, m(3) AS INTEGER**

Если в процессе выполнения программы элементы массива требуется очистить, то используется оператор

**ERASE a[,b . . . c] ,** где **a, b, c** – имена массивов.

Если границы массива в операторе **DIM** указаны в виде пере­менных (или выражений, содержащих переменные), то массив счита­ется динамическим. Причем все переменные, определяющие границы массива, должны быть объявлены до оператора **DIM.** Например:

**m = 5**

**DIM mas1(m+2).**

Выделение памяти для хранения динамического массива осуществляется в процессе выполнения программы. Такой массив можно многократно переопределять (изменять границы индексов, и даже размерность массива) с помощью оператора **REDIM**, имеющего такой же формат, что и оператор **DIM**.

Память, отве­денная для динамического массива, может быть освобождена с помощью оператора **ERASE**, а затем в нужный момент такой массив может быть объявлен оператором **REDIM**.

Следует отметить, что независимо от конкретной задачи, ал­горитмы формирования и обработки массивов обычно строятся с исполь­зованием регулярных циклических структур, а при работе с многомерными массивами с использованием вложенных циклов (цикл в цикле).

### 2.5.1.7. Средства описания и работы с двумерными массивами данных

Как известно, **массив** - последовательность переменных одинакового типа, объединенных общим ***именем***. Например: ***двумерный массив*b(3, 2**):

**b(0,0) b(0,1) b(0,2)**

**b(1,0) b(1,1) b(1,2)**

**b(2,0) b(2,1) b(2,2)**

**b(3,0) b(3,1) b(3,2)**

Количество индексов указывает ***на размерность*** массива. Так, в приведенном выше примере размерность массива **b(3,2)** - **2** (двумерный). а размер массива **b(3,2)** равен**12**-ти.

Перед использованием двумерного массива в программе, его необходимо объявить с помощью оператора **DIM**, который выделяет место в памяти для размещения элементов массива. Например,

**DIM b(3,2) AS SINGLE**

**DIM b(1 TO 4, 1 TO 3) AS SINGLE.**

Обратите внимание, что во втором случае нумерация индексов начинается с **1**, а в первом с **0**.

Ввод, вывод и обработка двумерных массивов, как правило, основаны на ис­пользовании вложенных циклов, которые обеспечивают перебор всех элементов массива. В некоторых задачах может иметь значение порядок перебора элементов массива: “вдоль строки” или “вдоль столбца”. Если внешний цикл будет организован по первому индексу (по строкам), а внутренний цикл по второму индексу (по столб­цам), то выполняется построчный перебор элементов двумерного массива. Если внешний цикл в качестве параметра использует второй индекс, а внутренний цикл - первый индекс, то эле­менты массива перебираются по столбцам.

### 2.5.1.8. Строковые данные и операции над строками

**Строковые (символьные) данные** – это константы и переменные, значениями которых являются не числа. Последовательности символов образуют строки.

Для описания строковых данных используется, оператор **DIM**, в котором после имени данных указывается тип **STRING**. Строковые данные могут быть описаны по первому символу имени с использованием оператора **DEFSTR.** И, наконец, указать, что данные являются строковыми можно путем добавления к имени суффикса **$**.

Если значение символьной переменной определяется с помощью оператора присваивания, то строковая константа, стоящая справа от знака присваивания, заключается в двойные кавычки. Например, **a$=”Привет**”.

Для ввода значений переменных строкового типа могут ис­пользоваться операторы **INPUT** и **READ/DATA.** При этом вводимые данные могут содержать запятые, точки с запятой и пробелы, только если вводимые значения заключены в кавычки.

Специальный оператор **LINE INPUT** служит только для ввода строк, и имеет формат, аналогичный формату оператора **INPUT**:

**LINE INPUT [;] [приглашение;] переменная**

Он вводит с клавиатуры строку длиной до **256** символов. При этом все символы (в том числе запятые, точки с запятой и пробелы) вплоть до символа "возврат каретки" присваиваются **переменной**.

При вводе данных с помощью операторов **INPUT** и **LINE INPUT** вводимые символы отображаются на экране монитора. Чтобы ввести с клавиатуры значение символьной переменной, состоящей из **к** символов, без отображения вводимых символов на экране, можно воспользоваться функцией **INPUT$(k)**.

При выполнении оператора **t$=INPUT$(k)** происходит прерывание программы и ожидание ввода строки.

Элементами строковых выражений могут быть строковые кон­станты, переменные и встроенные строковые функции.

Един­ственной операцией над строковыми данными в языке **QB** является операция **сцепления (конкатенация),** обозначаемая в строковых выражениях символом "**+**". При выполнении операции сцепления двух строковых данных к последнему символу первого операнда присоединяется первый символ второго операнда.

Например, операция сцепления двух строковых констант "**Quick** " **+** "**BASIC**" образует строковое выражение, значением которого является строка "**Quick BASIC**" .

Строки можно сравнивать между собой, то есть применять к ним операции отношения **(<, >, <=, >=, =, <>**). При этом поочередно, начиная с первого символа каждой строки, производится сравнение ASCII кодов соответствующих символов. Большей считается та строка, у которой больший код встретится раньше. Ниже приведены примеры сравнения строк:

1. **a$=”abcdefg”** и **b$**=”**abcxyz**”. Тогда **a$<b$,** так как код ASCII буквы **x** больше кода буквы **d** (буква **x** по алфавиту дальше, чем **d**);
2. **a$=”Аня”** и **b$=”Алла”.** Тогда **a$>b$,** так как буква **н** в алфавите дальше**,** чем **л**;
3. **a$=”Basic”** и **b$=”Pascal”**. Тогда **a$<b$,** так как буква **B** в алфавите раньше, чем **P**.

При работе с символьными переменными часто возникает не­обходимость преобразования данных числового типа в данные символьного типа. Для этого можно воспользоваться функцией

**STR$ (арифметическое\_выражение)**

Эта функция возвращает символьное представление значения арифметического выражения. Если арифметическое выражение больше **0**, то функция возвращает строку с добавлением лидирующего пробела (вместо знака “**+**”).

Например, после выполнения опера­тора **text$=STR$(-12.5\*2)** строковая переменная **text$**  примет значение “**-25**", а после выполнения оператора **text$=STR$(12.5\*2**), переменная **text$** примет значение **“ 25”.**

Обратное преобразование переменной или константы символьного типа, пред­ставляющей число, в переменную числового типа, выполняется с помощью функции

**VAL(строка)**

Функция **VAL()** возвращает числовое представление строки. При этом она ликвидирует лидирующие пробелы, а если первый символ аргумента – нечисловой, то функция **VAL()** возвращает **0**.

Например, после выполнения операторов **t$=”55"** и **d=VAL(t$)** числовая переменная **d** примет значение **55**; после выполнения операторов **t$=” 2.5”**и**d=VAL(t$)** переменная **d** примет значение **2.5**; после выполнения операторов **t$=”D12”** и**d=VAL(t$)** – значение **d** равно **0**.

В табл. **3.9.1** приведены основные строковые функции **QB** и краткое описание их назначения.

Рассмотрим несколько примеров использования строковых функций.. Пусть имеются переменные**g$=”геометрия”** и**b$=”биология”.** Тогда:

Результатом **LEFT$(g$,3) + RIGHT$(b$, 5)** является **“геология”**

Результатом **MID$(g$,4,4)+MID$(b$,3,6)** является **"метрология”**

Результатом **"ypa"+STRING$(3," !")** является **"Ура ! ! !"**

Результатом **STRING$ (3, "Ура 1 ")** является **"Ура 1 Ура 1 Ура1 ”.**

Оператор **PRINTCHR$(100)** выведет на экран символ **d**, которому соответствует код ASCII, равный **100**.

Если **str1$="beta"** и **str2$="Beta'**, то оператор **PRINT ASC(str1$), ASC(str2$)** выведет на экран два числа: **98** и **66**, соответствующие значениям кодов ASCII латинских символов **b** и **В**.

Оператор **PRINT LEN(str1$)** выведет число**4** - количество символов в строковой переменной  **str1$.**

Таблица 2.5.1-5

|  |  |
| --- | --- |
| Имя **и параметры функции** | **Результат выполнения функции** |
| **LEFT$(s$,n)** | Возвращает **n** первых символов строки **s$** |
| **RIGHT$(s$,n)** | Возвращает **n** последних символов строки **s$** |
| **MID$(s$,n[,m] )1** | Возвращает строку из **m** символов, на­чиная с символа **n** строки **s$** |
| **SPACE$(n)** | Возвращает строку из **n** пробелов |
| **STRING$(n,”символ" )** | Возвращает строку, повторяющую **n** раз указанный символ или группу символов (строку) |
| **LCASE$(s$)**2 | Заменяет прописные буквы строки **s$** на строчные |
| **UCASE$(s$)**3 | Заменяет строчные буквы строки **s$** на прописные. |
| **LTRIM$(s$)** | Удаляет начальные пробелы строки **s$** |
| **RTRIM$(s$)** | Удаляет конечные пробелы строки **s$** |
| **INSTR([n,]s1$,s2$)** | Возвращает номер позиции вхождения (начиная с позиции **n** или с первой, если параметр **n**опущен**)** строки **s2$** в строку **s1$** |
| **CHR$(n)** | Преобразует целочисленный аргумент (значения от 0 до 255) в символьное значение в соответствии с таблицей кодов ASCII |
| **ASC(s$)** | Возвращает код ASCII первого символа строки **s$** |
| **LEN(s$)** | Возвращает количество символов (байт) строки **s$** |

**1** Функция **MID$** может находиться в левой части оператора присваивания, результатом выполнения которого является замещение указанного числа символов строковой переменной, начиная с заданной позиции, символами строкового выражения в правой части оператора. Например, после выполнения операторов **a$=”TURBOBASIC”** и **MID$(a$,1,5)=”Quick”**, переменная **a$** примет значение **“QuickBASIC”.**

Если в функции **MID$(s$,n)** не указано, сколько символов замещается, то замещение производится, начиная с **n** символа и до конца строки **s$**. Пусть **d$=’’KapMaH"** , тогда в результате выполнения оператора **MID$(d$,4)=”тон”** содержимым переменной **d$** станет слово **“Картон”**.

**2** Действие функции не распространяется на буквы русского алфавита.

**3** Действие функции не распространяется на буквы русского алфавита.

### 2.5.1.9. Средства процедурного программирования в среде QB

Практический стиль современного программирования базируется на принципах процедурного программирования.

Процедура - это последовательность логически связанных фрагментов программы, оформленных как отдельная часть программы специальным способом.

Основными свойствами процедуры являются следующие:

* процедура должна выполнять определенную (и в идеале единственную) функцию;
* процедура должна быть независимой, то есть замена ее другой (новой) процедурой, воспринимающей такие же входные данные и выдающей такие же выходные данные, не отразится на других процедурах программы;
* процедура может вызываться как управляющей программой (главной программой), так и другими процедурами, и, в свою очередь, и сама может вызывать другие процедуры.

Технологии, ориентированные на принципы процедурного программирования во-первых, позволяют сократить время создания программных комплексов; во-вторых, позволяют создавать библиотеки процедур; в-третьих, существенно облегчают процесс отладки и контроля выполнения программ.

Для использования процедур при написании программ необходимо уметь не только их описывать (объявлять) и вызывать, но и организовывать (управлять) межпроцедурные взаимодействия.

Одной из основных проблем при организации межпроцедурного взаимодействия является проблема передачи данных между различными процедурами, которая также является проблемой взаимного влияния друг на друга данных различных процедур (то есть “видимости или доступности данных одной процедуры в другой процедуре”).

Существует два основных способа передачи данных между процедурами:

* использование глобальных переменных;
* использование параметров процедуры.

Если переменные одной процедуры доступны в других процедурах без механизма передачи параметров, то такие переменные называются глобальными переменными.

При этом, если переменные с одинаковыми именами в разных процедурах являются независимыми, то такие переменные называются локальными переменными, а локальные переменные, которые сохраняют свои значения между вызовами процедур, называются статическими переменными.

Основным недостатком использования глобальных переменных является несоответствие требованию независимости, так как в этом случае любое изменение глобальной переменной в одной из процедур приводит к ее изменению в другой.

При использовании механизма передачи параметров в описании процедуры указывается список передаваемых параметров. Такие параметры называются формальными параметрами. Формальные параметры определяют имя и тип переменных и являются по сути локальными для данной процедуры.

При вызове процедуры происходит замена формальных параметров конкретными значениями (фактическими параметрами) и выполнение действий, описанных в процедуре, со значениями фактических параметров. При этом параметры могут передаваться как по значению, так и по адресу (по ссылке). В первом случае перед передачей (по значению) значение передаваемого параметра вычисляется, а результат копируется во временную локальную переменную, и эта копия передается в процедуру. Необходимо отметить, что любые возможные изменения значений параметров внутри процедуры никак не отражаются в вызывающей программе. Обычно по значению передаются входные параметры процедуры.

Во втором случае, в процедуру передается адрес переменной, и любые изменения значения параметра в процедуре приводят к изменению значения параметра в вызывающей процедуре. По ссылке передаются параметры, которые могут быть как входными, так и выходными параметрами процедуры.

К средствам процедурного программирования в языке **QB** отно­сятся однострочные и многострочные функции, процедуры-функции и процедуры.

Таким образом, программа может состоять из множества отдельных программ – процедур, среди которых одна должна быть главной, с которой начинается выполнение программы. Поэтому, в процессе проектирования программы вначале строится так называемая схема иерархии, отражающая взаимодействие отдельных процедур всей программы. Причем на верхнем уровне схемы должна находиться главная (управляющая) программа. Каждый из нижеследующих уровней содержит процедуры, необходимые для описания и реализации соответствующих подзадач данного уровня, и детализации подзадач предыдущего уровня. При проектировании структуры программы, использующей средства процедурного программирования **Q**B, необходимо учитывать местоположение в программе отдельных процедур. Так, при использовании однострочных и многострочных функций, их описания должны находиться до начала главной программы. Процедуры-функции и процедуры обычно помещаются после оператора **END** главной программы друг за другом.

Однострочные функции используются в тех случаях, когда функ­ция, вызываемая из любой программы, описывается одним арифметическим, логическим или строковым выражением, а результатом ее вы­полнения является скалярная величина. Средством определения (описания) одностроч­ной функции служит оператор **DEF FN**, имеющий следующий формат:

**DEF FNимя (список\_формальных\_параметров) = выражение**

где: имя - допустимое имя переменной, которое, вместе со служебной приставкой **FN** составляет полное имя функции. Оно может включать один из допустимых суффиксов объявления типа, определяю­щий тип функции. При отсутствии такого суффикса тип функции определяется типом переменной имя, установленным оператором **DEFтип** или по правилам умолчания; список\_формальных\_параметров имеет следующий формат:

**переменная [AS тип][, переменная [AS тип]]... ,**

где переменная является допустимым именем переменной, а тип может принимать значения **INTEGER, LONG, SINGLE, DOUBLE, STRING** (по умолчанию тип переменной SINGLE):

выражение – любое допустимое выражение (выражение может содержать глобальные переменные).

Вызов однострочной функции осуществляется по ее полному имени с указанием списка фактических параметров (аргументов), заключенных в круглые скобки. Количество аргу­ментов, порядок их следования и их тип должны соответствовать списку формальных параметров при описании этой однострочной функции. Передача параметров в однострочных функциях осуществляется по значению.

Оператор **DEF FNимя** должен быть описан до первого вызова однострочной функции.

Например, если в программе описана функция

**DEF FNch(z) = (EXP(z)+EXP(-z))/2 ,**

то в программе ее вызов осуществляется по ее полному имени, например

**x=-2.5**

**s = FNch(2) +3\*FNch(x^2)** .

Многострочные функции используются в тех случаях, когда функция при своем выполнении возвращает в качестве резуль­тата одну единственную величину, но не может быть описана од­ним выражением.

Средством определения многострочной функции служат опера­торы **DEF FN** и **END DEF**, имеющие следующий формат:

**DEF FNимя(список\_формальных\_параметров)**

**операторы**

**[EXIT DEF]**

**операторы**

**FNимя=выражение**

**END DEF**

Все, сказанное выше об однострочных функциях, в полной мере относится и к многострочным функциям. Единственная осо­бенность заключается в том, что для возврата полученного в функции результата, его значение надо присвоить переменной с тем же именем, что и имя функции.

Например, в программе описана функция:

**DEF FNSum(n AS INTEGER)**

**s=0**

**FOR i=1 TO n**

**s=s+i**

**NEXT i**

**FNSum=s/n**

**END DEF**

Тогда оператор: **FNSum=s/**n означает присвоение результата переменной, имеющее тоже имя, что и функция.

Для досрочного выхода из функ­ции может быть использован оператор **EXIT DEF**.

По умолчанию все пе­ременные внутри однострочной и многострочной функции являются глобальными. Однако можно сделать отдельные переменные локальными, если в теле многострочной функции объявить их оператором **STATIC**:

**STATIC переменная [AS тип][, переменная [AS тип]]**

Например,

***‘ Описание многострочной функции***

**DEFINTi, m**

**DEF FNSum(n AS INTEGER)**

**STATIC i AS INTEGER**

**s=0**

**FOR i=1 TO n**

**s=s+i**

**NEXT i**

**FNSum=s/n**

**ENDDEF**

**‘ *Главная программа***

**CLS**

**INPUT “Введитеm=”; m**

**FOR i=1 TO m**

**PRINT FNSum(i)**

**NEXTi**

**END**

Объявление переменной **i** как **STATIC** внутри тела функции позволяет использовать эту переменную как независимую переменную в главной программе.

Наиболее мощным средством модульного программирования в **QB** являются процедуры-функции и процедуры. Подобно много­строчным функциям, они располагают механизмом передачи пара­метров, однако их параметрами могут быть не только скаляр­ные переменные, но и массивы. Передача фактических параметров-выражений в процедуру-функцию и процедуру производится по значению, а фактических параметров-переменных и параметров-массивов - *по ссылке*. Па­раметры-переменные могут быть переданы и по значению, для чего их имена в списке фактических параметров должны быть за­ключены в круглые скобки. Все переменные, описанные внутри процедуры-функции и процедуры, по умолчанию являются локальными.

Процедуры-функции, как и многострочные функции, использу­ются в тех случаях, когда модуль воз­вращает в качестве результата одну единственную скалярную ве­личину.

Для определения процедуры-функции служат операторы **FUNCTION** и **ENDFUNCTION**:

**FUNCTION имя [(список\_формальных\_параметров)]**

**операторы**

**[EXIT FUNCTION]**

**операторы**

**имя = выражение**

**END FUNCTION**

где: имя - допустимое имя переменной, которое может включать один из суффиксов явного объявления типа функции (тип возвращаемого функцией значения). При отсутствии такого суффикса тип функции определяется типом переменной имя, установленным оператором **DEF тип** или по умолчанию; список\_формальных\_параметров имеет следующий формат:

**переменная [( )] [AS тип][, переменная [( )] [AS тип]]**

Если параметр является массивом, то после имени соответствую­щей переменной ставятся пустые круглые скобки. Тип переменной может быть указан и явно. Переменные, перечисленные в списке при описании функции, являются формальными параметрами и их имена никак не связаны с именами фактических параметров при вызове функции.

выражение - любое допустимое арифметическое, логическое или строковое выражение, соответствующее типу функции. Оно может содержать, как формальные параметры, так и любые другие пере­менные. Вычисленное значение выражения служит результатом, который воз­вращается функцией через ее имя.

Для досрочного выхода из процедуры-функции может быть ис­пользован оператор

**EXIT FUNCTION**

Вызов процедуры-функции из вызывающей процедуры осу­ществляется указанием ее имени и списка фактических параметров (аргументов), заключенных в круглые скобки. Количество аргу­ментов, порядок их следования и их типы должны соответствовать списку формальных параметров в операторе **FUNCTION**. Если фак­тическим параметром является массив, то после его имени ста­вятся пустые круглые скобки.

Например:

***‘ Главная программа***

**DEFINTi, k**

**INPUT “k=”; k**

**DIM a(1 TO k) AS SINGLE**

**. . .**

**PRINT “max=”; max(a(), (k) )**

**END**

***‘Описаниепроцедуры-функции***

**FUNCTION max(b() AS SINGLE, n AS INTEGER)**

**DIM j AS INTEGER**

**m=b(1)**

**FOR j=2 TO n**

**IF b(j)>m THEN m=b(j)**

**NEXT j**

**max=m**

**END FUNCTION**

Все переменные внутри **FUNCTION** – локальные (по умолчанию). Но, если нужно, чтобы переменные или массивы, объявленные в главной программе, были доступны только в данной функции, их надо объявить оператором **SHARED** внутри этой функции:

**SHARED переменная [AS тип][, переменная [AS тип]]...**

Например:

**DEFINT i, k**

**INPUT “k=”; k**

**DIM a(1 TO k) AS SINGLE**

**. . .**

**PRINT “max=”; max( (k) )**

**END**

**‘*Описание процедуры-функции***

**FUNCTION max(n AS INTEGER)**

**SHARED a() AS SINGLE**

**DIM j AS INTEGER**

**m=a(1)**

**FOR j=2 TO n**

**IF a(j)>m THEN m=a(j)**

**NEXT j**

**max=m**

**END FUNCTION**

Если нужно, чтобы переменные или массивы, объявленные на уровне главной программы, были доступны всем процедурам, то их нужно объявить в главной программе с использованием атрибута **SHARED:**

**DIM SHARED переменная [AS тип][, переменная [AS тип]]...**

Например:

**DIM SHARED k AS INTEGER**

**INPUT “k=”; k**

**DIM a(1 TO k) AS SINGLE**

**. . ..**

**PRINT “max=”; max( a() )**

**END**

**‘*Описание процедуры-функции***

**FUNCTION max(b() AS SINGLE)**

**DIM j AS INTEGER**

**m=b(1)**

**FOR j=2 TO k**

**IF b(j)>m THEN m=b(j)**

**NEXT j**

**max=m**

**END FUNCTION**

После такого объявления переменная **k** доступна всем процедурам-функциям и процедурам без использования в них опе­раторов **SHARED**.

Отдельные локальные переменные в процедурах-функциях и процедурах могут быть объявлены оператором **STATIC**. В этом случае они становятся статическими, то есть их значе­ния сохраняются между вызовами соответствующей процедуры.

Процедура – это основной вид программного модуля, который используется в тех случаях, когда результатом выполнения процедуры являются несколько скалярных ве­личин или массивов.

Для определения процедуры служат операторы **SUB** и **END SUB**, имеющие следующий формат:

**SUB имя [(список\_формальных\_параметров)]**

**операторы**

**[EXIT SUB]**

**операторы**

**END SUB**

В отличие от процедуры-функции, имя процедуры не имеет типа и используется лишь для вызова процедуры. Главное отличие втом, что резуль­таты выполнения процедуры возвращаются через ее параметры - переменные или массивы. В остальном все, сказанное о процеду­рах-функциях, в полной мере относится и к процедурам.

Вызов процедуры из некоторой программы (процедуры) производится с по­мощью оператора **CALL**, имеющего следующий формат:

**CALL имя [(список\_фактических\_параметров)]**

Количество фактических параметров, порядок их следования и тип в операторе CALL должны соответствовать списку формальных параметров в операторе **SUB** при описании процедуры. Если фактическим параметром является массив, то после его имени ставятся пустые круглые скобки.

В списке как формальных, так и фактических параметров исходные данные и результаты не различаются. Поэтому целесообразно исходные данные передавать по значению, а результаты возвращать по адресу. Например:

**DEFINT i, k**

**INPUT “k=”; k**

**DIM a(1 TO k) AS SINGLE**

**. . ..**

**CALL Sort( a( ) , (k) )**

**. . ..**

**END**

***‘Описаниепроцедуры***

**SUB Sort( b( ) AS SINGLE, n AS INTEGER)**

**DEFINT j, l**

**FOR j=1 TO n-1**

**FOR l=j+1 TO n**

**IF b(j)<b(l) THEN swap b(j), b(l)**

**NEXT l**

**NEXT j**

**END SUB**

# **2.5.2. Базовые средства и конструкции языка С++**

### 2.5.2.1. Базовые средства языка С++

  При написании программ на языке С++ используются символы, составляющие его алфавит. Широко используется символьный набор ISO 646-1983, называемый кодом ASCII (AmericanStandartCodeforInformationInterchange). Он содержит латинские буквы, арабские цифры, специальные и управляющие символы.

Алфавит С++ включает:

* латинские прописные и строчные буквы: A,…,Z,a,…,z;
* арабские цифры: 0,1,…,9;
* специальные символы: + - \* / <> = | ! @ # % $ ^ ~ & \ ( ) [ ] { } “ ‘ , . ; : \_ ?
* символ-разделитель – пробел, который на экране не отображается;
* управляющие символы: табуляция, перевод строки, возврат каретки, новая страница и новая строка. Эти символы на экране непосредственно не отображаются.

Использование кириллических символов в некоторых случаях возможно и целесообразно (в комментариях, символьных строках и др.). Но пока нет единого стандарта на кодировку кириллических символов. Поэтому могут быть сложности при переносе таких программ с одного компьютера на другой, при переходе из одной среды выполнения в другую.

Многие элементы программы (константы, переменные, функции и др.) имеют имена. Такие имена являются идентификаторами. Имена вводятся для того, чтобы отличать различные элементы одного вида от других и производить действия над ними. Идентификатором называется последовательность символов из латинских букв, символа подчеркивания, арабских цифр, которая начинается с буквы и служит для именования элементов программы. Примеры идентификаторов: X1, bad\_call, \_limit.

Следует отметить, что одинаковые прописные и строчные буквы считаются различными символами, так как имеют различные коды.

Знаки нумерации и специальные символы. Эти символы используются с одной стороны для организации процесса вычислений, а с другой - для передачи компилятору определенного набора инструкций.

Программа на языке **С++** содержит одну или несколько строк исходного текста. Каждая строка может содержать один или несколько операторов - специальных синтаксических конструк­ций, содержащих одно или несколько служебных слов, определяю­щих характер действия (ввести данные, проверить условие и т.п.). После каждого оператора ставится точка с запятой. Количество операторов в одной строке не ограничено. Однако принято в строке размещать один оператор.

Программа может содержать **комментарии** (примечания) для пояснения отдельных частей или всей программы. Ком­ментарий может располагаться в отдельной строке или в конце программной строки. Для введения однострочного комментария используют пару символов //, после которых следует поясняющий текст до конца строки. Многострочные комментарии начинаются с пары символов /\* и заканчиваются парой символов \*/.

Некоторые слова в **С++** являются **зарезервированными** (ключевыми) в служебных целях и их нельзя использовать для именования переменных, констант и функций. К таким зарезервированным словам относятся **имена ко­манд, операторов и встроенных функций.**

Основная цель любой программы состоит в обработке данных. Данные различного типа хранятся и обрабатываются по-разному.

**Тип данных** определяет:

1. внутреннее представление данных в памяти компьютера;
2. операции и функции, которые можно применять к величинам этого типа;
3. множество значений, которые могут принимать величины этого типа.

**Данные в С++** - это объекты, над которыми возможно выполнение некоторых действий, в результате которых получаются результирующие значения. В качестве таких объектов выступают **переменные** и **константы**. Данные могут иметь три характеристики: **имя**, **тип**, **значение**. **Имя** - это **идентификатор,** под которым данное известно в программе. То есть, в качестве имени используется идентификатор. Тип данных определяет область допустимых значений и возможность применения к нему тех или иных операций. **Значения** - это текущие значения конкретных данных.

Объекты, значения которых может быть изменено во время выполнения программы, являются **переменными**. Все переменные, используемые в программе, должны быть предварительно объявлены с помощью конструкции, которая называется **объявлением**:

**[спецификатор\_класа\_памяти] [const] спецификатор\_типа**

**описатель [=инициатор] [,описатель [= инициатор] ]... ;**

**Описатель** - идентификатор переменной либо более сложная конструкция индентификатора с квадратными скобками, круглыми скобками или звездочками.

**Спецификатор\_типа** - одно или несколько ключевых слов или более сложная конструкция, содержащая звездочки и круглые и квадратные скобки, определяющая тип объявляемой переменной. В языке **С++** имеется стандартный (базовый) набор типов данных, используя который можно сконструировать свои собственные типы данных.

**Инициатор** - задает начальное значение или список начальных значений, которые присваивается переменной при ее объявлении.

**Спецификатор\_класса\_памяти** - определяется одним из четырех ключевых слов языка С++: **auto, extern, register, static**, и, с одной стороны, указывает, каким образом будет **распределяться память** под объявляемую переменную, а с другой, **область видимости** этой переменной, т.е., из каких частей программы можно к ней обратиться.

Квадратные скобки означают необязательность операнда.

Объекты, значения которых невозможно изменить во время выполнения программы, это **константы**. А поскольку они имеют всегда одно и тоже значение, имя может отсутствовать. В программе константа это некоторое значение, записанное в той или иной форме.

Константы, имеющие имя, называются **именованными константами** и их, как и переменные, нужно предварительно объявлять. При объявлении именованных констант им, как и переменным, необходимо поставить в соответствие имена, типы и значения. Объявление именованной константы можно выполнить двумя разными способами. Первый способ аналогичен объявлению переменной, но перед именем именованной константы записывается модификатор **const**, а после имени обязательно записывается инициализатор. Второй способ может быть применен только для целых констант и связан с использованием специального **перечислимого типа**.

Примеры:

**int a=5, y;**

**const float g=9.85, C=0.566685;**

В языке **С++** имеются так называемые **базовые типы данных: числовые данные**, **символьные данные** и **указатели**. Причем числовые данные используются для представления и обработки в программе, как **целых**, так и **действительных** чисел. Из этих данных можно образовывать различные совокупности данных – **так называемые структурные данные:** массивы, структуры, объединения и классы. Подмножествами данных **целого типа** являются также данные **перечислимого типа** и **битовые поля**.

В **С++** не реализован **логический тип** данных, но имеется большой набор логических операций. В этих операциях вместо логических величин можно использовать любую арифметическую величину или указатель. При этом любое значение отличное от нуля считается истиной, а нулевое значение ложью.

В **С++** имеется несколько целых типов данных, отличающихся размером памяти, отводимой для хранения значения данного, а, следовательно, и диапазоном возможных значений, и способом записи знака. Для объявления целых переменных используются спецификаторы типа представленные в табл. 2.5.2-1.

Таблица 2.5.2-1

|  |  |  |
| --- | --- | --- |
| **Спецификаторы типа** | **Размер в байтах** | **Диапозон значений** |
| **char** | 1 | от-128 до 127 |
| **int** | 2 или 4 | как long или short |
| **short** | 2 | от -32768 до 32767 |
| **long** | 4 | от -2147483648 до 2147483647 |
| **unsigned char** | 1 | от 0 до 225 |
| **unsigned int** | 2 или 4 | как unsigned long или unsigned short |
| **unsigned short** | 2 | от 0 до 65535 |
| **unsigned long** | 4 | от 0 до 4294967295 |

Заметим, что ключевые слова **signed** и **unsigned** необязательны. Они указывают, как интерпретируется нулевой бит объявляемой переменной, то есть, если указано ключевое слово **unsigned**, то нулевой бит интерпретируется как часть числа, в противном случае нулевой бит интерпретируется как знаковый. В случае отсутствия ключевого слова **unsigned** целая переменная считается знаковой. В случае, если спецификатор типа состоит из ключевого типа **signed** или **unsigned** и далее следует идентификатор переменной, то она будет рассматриваться как переменная типа **int**.

Спецификатор типа **char** требуется для представления символьных переменных, которые в **С++** могут использоваться и как арифметические данные.

Рассмотрим несколько примеров.

**intx,y;**

**short a,b;**

**unsigned i,j; // unsigned int i,j;**

**unsigned chark;**

**Перечислимый тип данных** используется для объявления констант (табл. 2.5.2-2).

Таблица 2.5.2-2

|  |  |  |
| --- | --- | --- |
| **Спецификаторы типа** | **Размер в байтах** | **Диапозон значений** |
| **enum** | 2 | от -32768 до +32767 |

В **С++** вещественные типы данных (типы с плавающей запятой) представлены тремя размерами, характеризующие точность представления вещественных чисел:

**float** – одинарной точности;

**double** – двойной точности;

**long double** – расширенной точности.

Для объявления вещественных переменных используются спецификаторы типа представленные в табл. 2.5.2-3.

Таблица 2.5.2-3

|  |  |  |
| --- | --- | --- |
| **Спецификаторы типа** | **Размер в байтах** | **Диапозон значений** |
| **float** | 4 | от 2.5\*10-38 до 2.5\*E+38 |
| **double** | 8 | от 1.7\*10-308до 1.7\*10+308 |
| **long double** | 10 | от 1.7\*10-4932 до 1.7\*10+4932 |

Рассмотрим несколько примеров.

**float v, z;**

**double u;**

Необходимо обратит внимание, что выбираемое имя переменной при объявлении должно быть уникальным и нести смысловую нагрузку, которая описывает (для всех, кто читает вашу программу) использование переменной. Например, программа могла бы использовать переменные, такие как **employee\_name, employee\_age** и т. д. Обратите внимание на точку с запятой, которая следует за объявлением именем переменной. В **C++ объявление переменной считается оператором**, далее на этом остановимся подробней.

В качестве примера рассмотрим фрагмент программы, которая объявляет три переменные, используя типы **int, float и long**:

**#include <iostream.h>**

**void main(void)**

**{**

**int test\_score;**

**float salary;**

**long distance\_to\_mars;**

**}**

Данная программа ничего не выполняет, а только объявляет переменные. Как видите, объявление каждой переменной заканчивается точкой с запятой. Если вы объявляете несколько переменных одного и того же типа, можно разделять их имена запятой. Следующий оператор, например, объявляет три переменных с плавающей точкой:

**Float salary, income\_tax, retirement\_fund;**

Еще раз напомним, что переменная представляет собой имя области памяти в компьютера. Во время выполнения программы она хранит информацию. Поэтому при создании программ необходимо объявлять переменные, сообщая компилятору C++, как минимум имя и тип переменной. Например, оператор **int age;** объявляет переменную с именем **age** типа **int**.

И так каждая переменная должна иметь уникальное имя. Чтобы сделать свои программы более легкими для чтения и понимания, следует использовать смысловые имена переменных. Например, следующий оператор объявляет три переменных с именами **х, y** и **z:int х, у, z;**

Предположим, что эти переменные хранят возраст, тестовые очки и оценку студента, тогда следующие имена переменных более понятны по смыслу для других программистов, читающих ваш исходный код: **int student\_age, test\_score, grade;**

При выборе имен переменных можно использовать комбинацию букв, цифр и символа подчеркивания. Первый символ в имени переменной должен быть буквой или подчеркиванием. Нельзя начинать имя переменной с цифры. Кроме того, в **C++** буквы нижнего и верхнего регистров считаются разными. Например,

**float MonthlySalary, IncomeTax;**

При создании имен переменных необходимо знать, что в **C++** слова, перечисленные в табл. 2.5.2-4, резервируются в качестве ключевых слов, имеющих специальное значение для компилятора**С++.** Поэтому нельзя использовать ключевые слова **C++** в качестве имен переменных.

Таблица 2.5.2-4

|  |  |
| --- | --- |
| **Ключевые слова** | **Назначение** |
| **asm** | Встраивание команд на языке ассемблера в программу на С++ |
| **auto** | Используется для объявления локальных переменных |
| **break** | Немедленный выход из операторов циклов |
| **case** | Используется для определения одного из выбираемых вариантов действия в операторе **switch** |
| **catch** | Предназначен для обработки исключений, выбрасываемых оператором **throw** |
| **char** | Спецификатор символьного типа |
| **class** | Начинает описание производного структурного типа данных «класс объектов» |
| **const** | Определяет переменную-константу |
| **default** | Используется в операторе **switch**для указания операторов, выполняющихся по умолчанию |
| **delete** | Освобождает память |
| **do** | Является началом оператора цикла |
| **double** | Задает тип вещественных чисел двойной точности |
| **else** | Обозначает начало второй необязательной части оператора ветвления **if** |
| **enum** | Используется для описания перечислимых типов данных |
| **extern** | Сообщает компилятору о том, что переменная объявлена в другом файле или месте программы |
| **float** | Задает тип вещественных чисел одинарной точности |
| **friend** | Используется для объявления функций или классов, дружественных описываемому классу |
| **goto** | Вызывает переход в процессе выполнения программы к метке |
| **if** | Означает начало условного оператора |
| **inline** | Используется для описания встраиваемых функций |
| **int** | Определяет тип целых чисел |
| **long** | Определяет тип длинных чисел |
| **new** | Осуществляет выделение памяти |
| **operator** | Используется для описания функций-операций, переопределяющих встроенные операции языка |
| **protected** | Используется для объявления защищенных членов класса |
| **public** | Используется для объявления общедоступных членов класса |
| **register** | Предназначен для объявления локальных переменных, которые должны размещаться в наиболее быстродействующей памяти |
| **return** | Выполняет возврат из тела функции |
| **short** | Определяет тип коротких целых |
| **signed** | Используется для объявления переменных знакового символьного типа данных |
| **sizeof** | Возвращает выраженную в байтах длину переменной |
| **static** | Обеспечивает статическое выделение памяти |
| **switch** | Означает начало оператора выбора |
| **template** | Используется для описания шаблонов, или параметризованных функций и классов |
| **this** | Является указателем на объект, вызвавший функцию-член |
| **throw** | Используется для выбрасывания исключений |
| **try** | Является частью механизма работы с исключениями |
| **typedef** | Позволяет описывать новые имена для существующих типов данных |
| **union** | Используется для описания произвольного структурного типа данных - объединения |
| **unsigned** | Используется для объявления переменных беззнакового символьного или целого типа данных |
| **void** | Используется для того, чтобы объявлять функции, которые не возвращают значение в место вызова |
| **volatile** | Сообщает компилятору о том, что значение объявляемой переменной может изменяться не описанным в программе образом |
| **while** | Оператор цикла с предусловием |
| **continue** | Используется для обхода оставшихся операторов в теле цикла и перехода к очередной итерации |
| **for** | Оператор цикла с параметром |
| **private** | Используется для объявления собственных членов класса |
| **struct** | Начинает описание производного структурного типа данных - структура |
| **virtual** | Используется для объявления виртуальных функций, реализующих свойство полиморфизма родственных классов |

Как вы уже знаете, переменные хранят значения во время выполнения программы. После объявления переменной можно использовать **оператор присваивания** (=), чтобы присвоить значение переменной. Следующие операторы присваивают значения нескольким разным переменным:

**age = 32;**

**salary = 25000.75;**

**distance\_to\_the\_moon = 238857;**

Фрагмент следующей программы сначала объявляет переменные, а затем использует оператор присваивания, чтобы присвоить переменным значения:

**#include <iostream.h>**

**void main(void)**

**{**

**int age;**

**float salary;**

**long distance\_to\_the\_moon;**

**age = 32;**

**salary = 25000.75;**

**distance\_to\_the\_moon = 238857;**

**}**

При объявлении переменной также можно присваивать ей начальное значение. Чтобы упростить такую процедуру, **C++** позволяет присваивать значение во время объявления переменной, как показано ниже:

**int age = 32;**

**float salary = 25000.75;**

**long distance\_to\_the\_moon = 238857;**

Таким образом, переменной можно присвоить начальное значение двумя способами. Например,

**lesson = 4; илиintlesson = 4;**

После присвоения значения переменной программы могут использовать это значение, просто обращаясь к ее имени. Следующая программа **SHOWVARS.CPP** присваивает значения трем переменным и затем выводит значение каждой переменной, используя **cout**:

**#include <iostream.h>**

**void main (void)**

**{**

**int age = 32;  float salary = 25000.75;**

**long distance\_to\_the\_moon = 238857;**

**cout << "Служащему " << age << " года (лет)" << endl;**

**cout << "Оклад служащего составляет $" << salary <<endl;**

**cout << "Отземлидолуны " << distance\_to\_the\_moon << " миль" << endl;**

**}**

Обратите внимание, что последний оператор **cout** не помещается на одной строке. В этом случае программа просто переносит слова на следующую строку. Такой перенос правомерен, поскольку **C++** использует точку с запятой для указания конца оператора. Если необходимо перенести строку, постарайтесь не делать этого в середине символьной строки (внутри двойных кавычек), используйте дополнительный отступ для перенесенной части строки, как показано выше.

После выполнения программы, на экране появится следующий вывод:

**Служащему 32 года (лет)**

**Оклад служащего составляет $25000.75**

**От земли до луны 238857 миль**

Как известно тип переменной определяет диапазон значений, которые переменная может хранить. Например, переменная типа **int** может хранить значения в диапазоне от **-32768** до **32767**. Если вы присваиваете переменной значение, которое находится вне этого диапазона, возникает **ошибка переполнения**. Например, следующая

программа **OVERFLOW.CPP** иллюстрирует, как превышение диапазона значений переменной приводит к ошибке.

**#include <iostream.h>**

**void main(void)**

**{**

**int positive = 40000;**

**long big\_positive = 4000000000;**

**char little\_positive = 210;**

**cout << "сейчас positive содержит " << positive << endl;**

**cout << "сейчас big\_positive содержит " << big\_positive**

**<< endl;**

**cout << "сейчас little\_poaitive содержит " <<**

**little\_positive<<endl;**

**}**

После выполнения программы, на экране появится следующий вывод:

**сейчас positive содержит -25536**

**сейчас big\_positive содержит -294967296**

**сейчас little\_positive содержит Т**

Видно, что программа присваивает переменным типа **int**, **long** и **char** значения, которые находятся вне диапазона хранения каждого типа, поэтому возникает ошибка переполнения. При работе с переменными вам необходимо помнить диапазон значений, которые может хранить переменная каждого типа. Ошибки переполнения плохо уловимы, и поэтому их трудно определить и исправить. Обратите внимание на значение, которое программа выводит для переменной **little\_positive**. Поскольку это переменная типа **char**, выходной поток **cout** пытается вывести ее значение в символьном виде. В этом случае выведенное значение соответствует второй половине таблицы **ASCII** со значением **210**.

Кроме ошибок переполнения при выполнении программы, могут возникнуть ошибки, связанные с потерей точностей. Например, при работе с числами с плавающей точкой (числа с десятичной точкой) компьютер не всегда может представить число с требуемой точностью. Поэтому возможны ошибки округления, которые тяжело обнаружить.

Следующая программа **PRECISE. CPP** присваивает значение чуть меньше 0.5 переменным типа **float**и **double.**

К сожалению, поскольку компьютер обладает ограниченной способностью в представлении точности чисел, переменные реально содержат не присваиваемые им значения, а число 0.5:

**#include <iostream.h>**

**void main(void)**

**{**

**float f\_not\_half = 0.49999990;**

**double d\_not\_half = 0.49999990;**

**cout << "Значение типа float 0.49999990 равно " <<**

**f\_not\_half << endl;**

**cout << "Значениетипа double 0.49999990 равно " <<**

**d\_not\_half<<endl;}**

После выполнения программы, на экране появится следующий вывод:

**Значение типа float 0.49999990 равно 0.5**

**Значение типа double 0.49999990 равно 0.5**

Очевидно, значения, присваиваемые программой переменным, и значения, которые переменные содержат в действительности, не идентичны. Такие ошибки округления происходят потому, что компьютер должен представлять числа, используя фиксированное количество единиц и нулей. В большинстве случаев компьютер может точно представлять числа. Однако иногда, как показано в этой программе, компьютер представляет числа приближенно, а не точно. При программировании вам необходимо всегда помнить о точности. В зависимости от значений, с которыми работают ваши программы, могут

возникать трудно обнаруживаемые ошибки округления.

Комбинация знаков операций и операндов, результатом которой является определенное значение, называется выражением. Знаки операций определяют действия, которые должны быть выполнены над операндами. В простейшем случае операнд это переменная или константа. В общем случае операндом может быть выражение. Значение выражения зависит от расположения знаков операций и круглых скобок в выражении, а также от приоритета выполнения операций. Операции с более высоким приоритетом выполняются до операций с более низким приоритетом, если часть выражения заключена в скобки, то вначале вычисляется выражение внутри скобок. Например,

b=5; /\* b присвоить 5 \*/

t++; /\* к t прибавить 1 \*/

printf("\n %s",mes); /\* печать сообщения \*/

Смысл использования операторов выражений заключается в действиях выполняемых во время вычисления значения (изменение значений некоторых переменных, вызов функций и т.п.). В зависимости от используемых операций выражения различают на первичные, унарные, бинарные, тернарные, выражения присваивания и  выражения приведения типа. Список операций языкаС++ приведен в таб. 2.5.2-5.

Таблица 2.5.2-5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Опе-ция** | **Назначение** | **Тип** | **Приор.** | **Направление** |
| **()**  **[]**   **- >**  **.** | **Вызов функции, скобки**  **Выбор элемента по индексу**  **Выбор элемента по указателю**  **Выбор элемента по имени** | **Первичная** | **1** | **Слева**  **направо** |
| **-**  **!**  **~**  **\***  **&** **++**  **--**  **(тип)**  **sizeof** | **Отрицание арифметическое**  **Отрицание логическое**  **Отрицание побитовое**  **Косвенная адресация**  **Вычисление адреса**  **Инкремент**  **Декремент**  **Приведение типа**  **Вычисление размера в байтах** | **Унарная** | **2** | **Справа**  **налево** |
| **\*  /  %** | **Умножение**  **Деление**  **Остаток от деления** | **Мультипли-**  **кативные** | **3** | **Слева**  **направо** |
| **+  -** | **Сложение**  **Вычитание** | **Аддитивные** | **4** | **Слева**  **направо** |
| **>>**  << | **Сдвиг вправо**  **Сдвиг влево** | **Сдвиги** | **5** | **Слева**  **направо** |
| **>** **<** **>=**  **<=** | **Больше**  **Меньше**  **Больше или равно**  **Меньше или равно** | **Отношения** | **6** | **Слева**  **направо** |
| **==  ! =** | **Равно**  **Не равно** |  | **7** | **Слева**  **направо** |
| **& ^  |** | **Поразрядное И**  И**сключающее ИЛИ**  **Поразрядное ИЛИ** | **Поразрядные** | **8**  **9**  **10** | **Слева**  **направо** |
| **&& | |** | **Логическое И**  **Логическое ИЛИ** | **Логические** | **11**  **12** | **Слева**  **направо** |
| **? :** | **Условная операция** | **Условная** | **13** | **Справа**  **налево** |
| **=** | **Присваивание** | **Присваивание** | **14** | **Справа**  **налево** |
| **\*=**  **/=**  **%=**  **+=**  **-=**  **&=**  **|=**  **>> =**  **<<=**  **^=** | **Составное  присваивание** | **Присваивание** | **14** | **Справа**  **налево** |
| **,** | **Операция запятая** | **Последовате-**  **льное вычисление** | **15** | **Справа**  **налево** |

**К** первичным выражениям **относятся вызовы функций, индексные выражения и выбор элементов. Унарная операция имеет один операнд, бинарная - два, тернарная - три. В языке С++ реализовано несколько различных операций присваивания, и все они являются выражениями, значением выражения присваивания является величина, которая присваивается.**

Операнд - это константа, литерал, идентификатор, вызов функции, индексное выражение, выражение выбора элемента или более сложное выражение, сформированное комбинацией операндов, знаков операций и круглых скобок. Операнд, имеющий константное значение, называется константным выражением. Каждый операнд имеет тип.

Унарное выражение состоит из операнда и знака унарной операции, выполняется справа налево. В языке С++ имеются следующие унарные операции:

* - арифметическое отрицание (отрицание и дополнение);
* ~ побитовое логическое отрицание (дополнение);
* ! логическое отрицание;
* разадресация (косвенная адресация);
* & вычисление адреса;
* ++ увеличение (инкремент);
* -- уменьшение (декремент);
* sizeof вычисление размера.

Операции увеличения и уменьшения увеличивают или уменьшают значение операнда на единицу и могут быть записаны как справа, так и слева от операнда. Если знак операции записан перед операндом (префиксная форма), то изменение операнда происходит до его использования в выражении. Если знак операции записан после операнда (постфиксная форма), то операнд вначале используется в выражении, а затем происходит его изменение.

Бинарное выражение состоит из двух операндов, разделенных знаком бинарной операции. К бинарным операциям относятся аддитивные и мультипликативные операции, операции сдвига и отношения, поразрядные и логические операции, операция последовательного вычисления и операции присваивания. Бинарные операции выполняются слева направо, за исключением присваиваний, которые выполняются справа налево. Левый операнд операции присваивания, должен быть выражением, ссылающимся на область памяти (но не объектом, объявленным с ключевым словом const), такие выражения называются лево допустимыми и к ним относятся:

* идентификаторы переменных целого и плавающего типов, типов указателя, структуры, объединения;
* индексные выражения, исключая выражения имеющие тип массива или функции;
* выражения выбора элемента -> и . , если выбранный элемент является лево допустимым;
* выражения унарной операции разадресации \*, за исключением выражений, ссылающихся на массив или функцию;
* выражение приведения типа, если результирующий тип не превышает размера первоначального типа.

Тернарное выражение состоит из трех операндов, разделенных знаками тернарной операции ? и : и имеет формат:

операнд\_1 ? операнд\_2 : операнд\_3.

Например, результатом данного выражения

(a>0) ? а:0

будет значение а, если a>0 и нулевое значение - в противном случае.

При записи таких выражений следует помнить, что символы \*,&,! могут обозначать унарную или бинарную операцию.

Язык С++ имеет широкий набор библиотечных функций. В табл. 3.54.2-6приведены математические функции.

Таблица 2.5.2-6

|  |  |
| --- | --- |
| **Имя функции** | **Результат выполнения функции** |
| **abs(x)** | Абсолютная величина числа **х**, x-целое число |
| **fabs(x)** | Абсолютная величина числа **х,** x- вещественное число |
| **sqrt(x)** | Корень квадратный из числа **х>=0** |
| **exp(x)** | Число е (=2.718282) в степени **х** |
| **log(x)** | Натуральный логарифм числа **х>0** |
| **sin(x)** | Синус числа **х** |
| **cos(x)** | Косинус числа **х** |
| **tan(x)** | Тангенс числа **х** |
| **аsin(x)** | Арксинус числа **х** |
| **аcos(x)** | Арккосинус числа **х** |
| **atan(x)** | Арктангенс числа **х** (от **–π/2** до **+π/2**) |
| **rand()** | Число из случайной последовательности в диапазоне от **0** до 32767. |
| **ceil(x)** | Округление до большего целого |
| **floor(x)** | Округление до меньшего целого |

При выполнении большинства бинарных операций производится автоматическое преобразование типов, чтобы привести операнды выражений к общему типу или чтобы расширить короткие величины до размера целых величин, используемых в машинных командах. Выполнение преобразования зависит от специфики операций и от типа операндов. Рассмотрим общие арифметические преобразования.

1. Операнды типа **float** преобразуются к типу **double**.
2. Если один операнд **double,** то второй также преобразуется к типу **double**.
3. Любые операнды типа **char** и **short** преобразуются к типу **int.**
4. Любые операнды **unsigned char** или **unsigned short** преобразуются к типу **unsigned int**.
5. Если один операнд типа **unsigned long**, то второй преобразуется к типу **unsigned long**.
6. Если один операнд типа **long**, то второй преобразуется к типу **long.**
7. Если один операнд типа **unsigned int**, то второй операнд преобразуется к этому же типу.

Таким образом, можно отметить, что при вычислении выражений операнды преобразуются к типу того операнда, который имеет наибольший размер. Например,

**double ft,sd;**

**unsigned char ch;**

**unsigned long in;**

**int i;**

**....**

**sd=ft\*(i+ch/in);**

При выполнении оператора присваивания правила преобразования будут использоваться следующим образом. Операнд **ch** преобразуется к **unsigned int** (правило 4). Затем он преобразуется к типу **unsigned long** (правило 5). По этому же правилу **i** преобразуется к **unsigned long** и результат операции, заключенной в круглые скобки будет иметь тип **unsigned long**. Затем он преобразуется к типу **double** (правило 2) и результат всего выражения будет иметь тип **double**.

Операция **sizeof**вычисляет размер памяти занимаемой операндом. Операция **sizeof** имеет следующий формат:

**sizeof(выражение)**

В качестве выражения может быть использован любой идентификатор, либо имя типа. Отметим, что нельзя использовать тип **void**, а идентификатор не может быть именем функции. Если в качестве выражения указанно имя массива, то результатом является размер всего массива (т.е. произведение числа элементов на длину типа), а не размер указателя, соответствующего идентификатору массива.

Когда **sizeof** применяются к имени типа структуры или объединения или к идентификатору, имеющему тип структуры или объединения, то результатом является фактический размер структуры или объединения, который может включать участки памяти, используемые для выравнивания элементов структуры или объединения. Таким образом, этот результат может не соответствовать размеру, получаемому путем сложения размеров элементов структуры. Например,

**struct**

**{**

**char h;**

**int b;**

**double f;**

**} str;**

**inta1;**

**a1 = sizeof(str);**

Переменная **а1** получит значение, равное **12**, в то же время если сложить длины всех используемых в структуре типов, то получим, что  длина структуры**str** равна **7.**

К классу мультипликативных операций относятся операции умножения (\*), деления (/) и получение остатка от деления (%). Операндами операции (%) должны быть целые числа.

Операция умножения (\*) выполняет умножение операндов. Например,

**int i=5;**

**float f=0.2;**

**double g,z;**

**g=f\*i;**

Тип произведения **i** и **f** преобразуется к типу **double**, затем результат присваивается переменной **g** .

Операция деления (**/**) выполняет деление первого операнда на второй.

**int i =49, j =10;**

**float n, m;**

**n = i/j; /\* результат 4 так как оба операнда целые \*/**

**m = i/10.0; /\* результат 4.9 потому что второй операнд \*/**

**/\* имеет тип double \*/**

Операция остаток от деления (**%**) дает остаток от деления первого операнда на второй. Знак результата зависит от конкретной реализации, но в большинстве случаев совпадает со знаком делимого. Если второй операнд равен нулю, то выдается сообщение об ошибке.

**int n = 49, m = 10, i, j;**

**i = n % m; /\* результат 9 \*/**

**j = n % 8; /\* результат 1 \*/**

К аддитивным операциям относятся сложение + и вычитание -. Например,

**int i=30000, j=30000, k;**

**k=i+j;**

В результате сложения **k** получит значение равное **-5536** потому, что сумма **60000** не может быть значением типа **int**.

Операндами операции сложения могут быть указатель и целое число. При этом целое число предварительно умножается на размер адресуемого указателем объекта. В результате получается указатель, адресующий область памяти расположенную на целое число объектов дальше от первоначального указателя.

Операция вычитания (-) вычитает второй операнд из первого. Возможна следующая комбинация операндов:

1. Оба операнда целого или плавающего типа.
2. Оба операнда являются указателями на один и тот же тип.
3. Первый операнд является указателем, а второй - целым.

При вычитании из указателя целого числа производится такое же масштабирование, что и при сложении и в результате получается адрес, сдвинутый на целое число объектов перед первоначальным указателем. При вычитании одного указателя из другого получается число объектов данного типа, которые можно разместить между двумя указателями. Например,

**double d[10],\* u;**

**int i;**

**u = d+2; /\* u указывает на третий элемент массива \*/**

**i = u-d; /\* i принимает значение равное 2 \*/**

Отметим, что операции сложения и вычитания над адресами в единицах отличных от длины типа, могут привести к непредсказуемым результатам. В следующем фрагменте

**float h[7];**

**int \*w;**

**h[0]=0;**

**w=h;**

**w++;**

**\*w=3000;**

значение указателя **w** после операции **w++**не будет равно адресу элемента **h[1]**, а будет указывать на адрес внутри поля **h[0].**Поэтому присваивание **\*w=3000** изменит значение **h[0]** на значение **-6.338253e+31**.

Операции увеличения (**++**) и уменьшения (**--**) являются унарными операциями присваивания. Они соответственно увеличивают или уменьшают значения операнда на единицу.

Если знак операции стоит перед операндом (префиксная форма записи), то изменение операнда происходит до его использования в выражении и результатом операции является увеличенное или уменьшенное значение операнда.

В том случае если знак операции стоит после операнда (постфиксная форма записи), то операнд вначале используется для вычисления выражения, а затем происходит изменение операнда. Например,

**int t=1, s=5, z, f;**

**z=(t++)\*5;**

Таким образом, вначале происходит умножение **t\*5**, а затем увеличение **t.** В результате получится **z=5, t=2**.

**f=(++s)/3;**

Вначале значение **s** увеличивается, а затем используется в операции сложения. В результате получим **s=6, f=2**.

***Указатели и операции над ними.*** Эти операции используются для работы с переменными типа указатель. Операция разадресации **\*** осуществляет косвенный доступ к адресуемой величине через указатель. Операнд должен быть указателем. Результатом операции является величина, на которую указывает операнд. Типом результата является тип величины, адресуемой указателем. Результат не определен, если указатель содержит нулевое значение или значение, не выровненное на границу адресуемого объекта (значение указателя на short должно указывать на четный адрес, значение указателей на long и на float должно делиться на 4 и т.п.). При выполнении разадресации контроль за существованием какого либо объекта по указанному адресу осуществляется только программистом и некорректное использование этой операции может привести к непредсказуемым последствиям. Например:

**int \* s;**

**\*s=7;**

В этом примере объявлен указатель **s**, но выполнение присваивания **\*s=7** приводит к непредсказуемым последствиям, поскольку значение указателя s к моменту выполнения присваивания не определено.

Операция адрес **&**формирует адрес своего операнда. Операндом может быть любая переменная или элемент массива, структуры или объединения. Имя функции или массива также может быть операндом операции адрес, хотя в этом случае операция лишена смысла, так как имена массивов и функций сами по себе являются адресами. Результат операции имеет тип указателя на тип операнда.

Операция адрес не может применяться к элементам структуры, являющимися полями битов, и к объектам с классом памяти **register.**

Примеры :

**int t, f=56, \* adress;**

**adress = &t;**

**/\* переменной adress, объявленной как указатель, присваивается \*/**

**/\* адрес переменной t \*/**

**\* adress = f;**

**/\* переменной находящейся по адресу,содержащемуся в переменной \*/**

**/\* adress, присваивается значение переменной f, т.е. 56 , что \*/**

**/\* эквивалентно t=f; или t=56; \*/**

**adress = 123; \* adress = 55;**

**/\* Oшибка! Значение переменной adress не кратно длине типа int \*/**

**adress = 124;**

**\* adress = 55;**

**/\* Ошибка! Обращение к области памяти вне программы \*/**

### 2.5.2.2. Средства программирования алгоритмов линейной структуры и функций

**Алгоритмы линейной структуры** содержат базовые алгоритмические структуры только последовательного типа. Соответственно, при программировании алгоритмов линейной структуры все операторы программы выполняются последовательно в порядке их записи. Программа такого типа назы­вается **последовательной**. Последовательные программы содержат опера­торы, выполняющие: ввод исходных данных, последовательную обработку данных и получение результатов, вывод результатов на экран (или печать).

**Все операторы языка С++ можно условно разделить на следующие категории:**

* условные операторы**, к которым относятся оператор условия if и оператор выбора switch;**
* операторы цикла (**for, while, do while**)**;**
* операторы перехода (**break, continue, return, goto**)**;**
* другие операторы (**оператор выражение, пустой оператор**)**.**

**Необходимо обратить внимание, что в языке С++ нет операторов ввода\вывода. Функции ввода\вывода поддерживаются с помощью встроенных функций:**

**Здесь рассмотрим лишь операторы, которые используются при программировании алгоритмов линейной структуры и соответствующих функций.**

**Таким образом, любое выражение, которое заканчивается точкой с запятой, является оператором выражения. Выполнение оператора выражения заключается в вычислении выражения. Полученное значение выражения никак не используется, поэтому, как правило, такие выражения вызывают побочные эффекты. Заметим, что вызов функции, не возвращающей значения, можно осуществить только при помощи оператора выражения.** Например,

****clrscr(); /\* - вызов функции \*/****

****a=b+c; /\* - выражения \*/****

****i++; /\* с присваиванием \*/****

**Пустой оператор состоит только из точки с запятой. При выполнении этого оператора ничего не происходит. Этот оператор необходим для двух случаев. Во-первых, он используется в составе других операторов в том месте, где по синтаксису необходим оператор, но по смыслу программы он не требуется. Во-вторых - при необходимости пометить закрывающуюся фигурную скобку.**

**Синтаксис языка С++ требует, чтобы после метки обязательно следовал оператор. Фигурная же скобка оператором не является. Поэтому, если надо передать управление на фигурную скобку, необходимо использовать пустой оператор.** Например,

****int main ()****

****{****

****:****

****{ if (...) goto a; /\* переход на скобку \*/****

****{ ...****

****}****

****a:; } /\* помеченный пустой оператор \*/****

****return 0;****

****}****

**Составной оператор представляет собой один или несколько операторов и объявлений, заключенных в фигурные скобки:**

**{ [объявление]**

**:**

**оператор; [оператор];**

**: }**

**Заметим, что в конце составного оператора точка с запятой не ставится. Обычно составным оператором называют оператор, не содержащий объявлений, а оператор с объявлениями называется блок, но синтаксических отличий между ними нет.**

**Выполнение составного оператора заключается в последовательном выполнении составляющих его операторов.** Например,

****{ r=s; s=t; t=r; }****

**Этот составной оператор меняет местами значения переменных s и t, используя вспомогательную переменную r.**

****int main ()****

****{****

****int q,b;****

****double t,d;****

****:****

****if (...)****

****{ /\* началоблока \*/****

****int e,g;****

****double f,q;****

****e=b+t-d;****

****g=++e;****

****f=q=e/g;****

****} /\* конец блока \*/****

****:****

****return 0;****

****}****

**В отмеченном блоке объявлены переменные e,g,f,q которые будут уничтожены после выполнения составного оператора. Отметим, что переменная qявляется локальной в составном операторе, т.е. она никоим образом не связана с переменной q объявленной вначале функции main() с типом int.**

***Структура С++ программ.*** Обратите внимание, что первые примеры программ С++ придерживаются одного и того же формата: начинаются с одного или нескольких операторов #include, содержат строку void main(void), а затем набор операторов, сгруппированных между левой и правой фигурными скобками. Из этого урока вы поймете, что эти несколько запугивающие операторы реально очень просто освоить. К концу данного урока вы изучите следующие основные концепции: Оператор #include обеспечивает преимущества использования заголовочных файлов, которые содержат операторы C++ или программные определения. Основная часть программы на C++ начинается с оператора void main(void).

Программы состоят из одной или нескольких функций, которые, в свою очередь, состоят из операторов, предназначенных для решения определенной задачи. При выводе на экран ваши программы будут широко использовать выходной поток cout. Когда вы создаете программы на C++, вы реально работаете в терминах операторов, но не инструкций. Позже вы изучите оператор присваивания, который присваивает значения переменным, оператор if, который позволяет программе принимать решения и т. д. А пока мы просто будем ссылаться на содержимое вашей программы, как на операторы программы.

Ранее мы создали на **C++** программу **FIRST.CPP**, которая содержала следующие операторы:

**#include <iostream.h>**

**void main(void)**

**{**

**cout << "Учимся программировать на языке С++!";**

**}**

Каждая программа начинается с оператора **#include**, заставляет компилятор включить содержимое заданного файла в начало вашей программы. В данном случае компилятор включит содержимое файла **iostream.h**. Файлы с расширением **h**, которые вы включаете в начало (или заголовок) вашей программы, называются заголовочными файлами. Если вы посмотрите на каталог, содержащий файлы компилятора С++, то можно увидеть подкаталог с именем **INCLUDE**, в котором находятся разные заголовочные файлы. Каждый заголовочный файл содержит определения, предоставляемые компилятором для различных операций. Например, существует заголовочный файл, который содержит определения для математических операций, другой заголовочный файл описывает файловые операции и т. д. Заголовочные файлы представляют собой файлы в формате **ASCII**, следовательно, вы можете вывести их содержимое на экран или принтер.

В данный момент не беспокойтесь о содержимом заголовочных файлов. Просто поймите, что оператор **#include** позволяет вам использовать эти файлы. Все программы на **C++**, созданные вами в процессе изучения этого пособия, содержат операторы **#include**, которые вы должны применять в ваших программах.

Таким образом, каждая программа на **C++** начинается с одного или нескольких операторов **#include**. Эти операторы указывают компилятору включить содержимое заданного файла (заголовочного файла) в вашу программу, как если бы программа содержала операторы, которые находятся во включаемом файле. Заголовочные файлы содержат определения, используемые компилятором для операций различных типов. Существуют заголовочные файлы, которые определяют операции ввода/вывода C++ (**в/в**) и системные функции, например, функции, возвращающие текущие дату и время, и многое другое.

При создании программы на **C++** ваш исходный файл будет содержать множество операторов. Причем каждая программа на **C++** имеет один вход - оператор, с которого начинается выполнение программы, — заголовок главной программы. В программах на **C++** оператор **void main(void)** является заголовком главной программы и указывает стартовую точку. По мере того как ваши программы становятся больше и сложнее, вы будете делить их на несколько небольших легко управляемых частей. При этом оператор **void main(void)** указывает начальные (или главные) операторы программы — часть программы, которая выполняется первой.

Таким образом, исходные файлы языка **C++** могут содержать очень много операторов, причем оператор **void main(void)** определяет главную программу, содержащую первый выполняемый оператор. Все программы на **C++** должны всегда содержать один и только один оператор с именем **main()**.

Как только ваша программа становится более сложной, вы должны разделить ее на небольшие более легко управляемые части, называемые функциями. Функция представляет собой простой набор операторов внутри программы, выполняющих определенную задачу. Например, при создании программы платежных документов, вы могли бы создать функцию с именем **salary**, вычисляющую оклад служащих. Аналогичным образом, если вы пишете математическую программу, вы могли бы создать функции с именами **square\_root** или **cube**, которые возвращают результат работы программы. Если ваша программа использует функцию, функция выполняет свою задачу и затем возвращает свой результат программе, которая ее вызвала.

Каждая функция в вашей программе должна иметь уникальное имя. А каждая программа имеет, по крайней мере, одну функцию - с именем **main**. Программы используют слово **void** для указания того, что функция не возвращает значения или не имеет значений, передаваемых в нее. Поэтому необходимо размещать слово **void** перед **main()**, как показано ниже:

**void main (void) //—-------> Программа не возвращает значение**

Внутри своих программ на C++ вы будете использовать правую и левую фигурные скобки {}, чтобы сгруппировать связанные операторы. В простых программах, представленных в нескольких первых уроках книги, эти символы группируют операторы, которые соответствуют операторам вашей главной программы.

Все программы на **C++** выводили сообщения на экран. Чтобы вывести сообщение, программы использовали **cout** и двойной знак "меньше" **(<< "Привет, C++!";**).

Слово **cout** представляет собой выходной поток, который **C++** назначает на стандартное устройство вывода операционной системы. По умолчанию операционная система назначает стандартное устройство вывода на экран дисплея. Чтобы вывести сообщение на экран, вы просто используете двойной символ "меньше" (называемый оператором вставки) с выходным потоком **cout**.

При использовании **cout** для вывода сообщений представляйте **cout** в виде потока символов, которые операционная система отображает на экране. Другими словами, порядок, в котором ваша программа посылает символы в **cout**, определяет порядок символов, которые будут появляться на экране. Например, для следующих операторов программы:

**cout << "Это сообщение появляется первым,";**

**cout << " а за ним следует настоящее сообщение.";**

операционная система выводит поток символов следующим образом:

**Это сообщение появляется первым, а за ним следует настоящее сообщение.**

В **C++** можно использовать с **cout** специальные символы для вывода табуляции или новой.

До сих пор созданные вами программы использовали **cout** для вывода символьных строк (букв и чисел, взятых в кавычки). Теперь вы узнаете, что можно также использовать для вывода чисел. Рассмотрим программу **1001.СРР,** котораявыводит число 1001 на ваш экран:

**#include<iostream.h>**

**voidmain(void)**

**{**

**cout << 1001;**

**}**

Откомпилируйте и запустите эту программу. На экране будет отображено число **1001**.

Кроме отображения целых чисел (чисел без десятичной точки), **cout** также позволяет программам отображать числа с плавающей точкой, например 1.2345 (программа **FLOATING.CPP**):

**#include <iostream.h>**

**void main(void)**

**{**

**cout << 0.12345;**

**}**

Двойной знак "меньше" является операцией вставки (эта операция вставляет символы в выходной поток для отображения на экран). С помощью **cout** можно использовать несколько операций вставки в пределах одного оператора. Например, следующая программа **1001ТОО.СРР** использует эту операцию четыре раза для отображения числа **1001** на экран:

**#include <iostream.h>**

**void main(void)**

**{**

**cout << 1 << 0 << 0 << 1;**

**}**

После выполнение этой программы на экране будет отображено следующее: **1001**

Таким образом, каждый раз, когда в **C++** встречается операция вставки, число или символы просто добавляются к тем, что находятся в настоящее время в выходном потоке.

Следующий пример (программа SHOW1001.CPP) выводит символьную строку и число:

**#include <iostream.h>**

**void main(void)**

**{**

**cout << "Мое любимое число равно " << 1001;**

**}**

Необходимо обратить внимание на то, что пробел, следующий за словом равно (внутри кавычек), служит для отделения числа **1001** от этого слова. Без пробела число сливается со следующим словом. Подобным образом следующая программа, содержащаяся в файле **1001MID.CPP** отображает число **1001** в середине символьной строки:

**#include <iostream.h>**

**void main(void)**

**{**

**cout << "Число " << 1001 << " мне очень нравится";**

**}**

Наконец, программа, содержащаяся в файле **MIXMATCH.CPP** комбинирует строки, символы, целые числа и числа с плавающей точкой внутри одного и того же выходного потока:

**#include <iostream.h>**

**void main(void)**

**{**

**cout << "B " << 20 << " лет мой оклад был " << 493.34 << endl;**

**}**

В результате выполнения этой программы на экране появится следующее сообщение:

**В 20 лет мой оклад был 493.34**

Все программы, созданные до сих пор, отображали свой вывод в виде одной строки. Однако большинство программ, отображает несколько строк вывода. Например, предположим, что вы пишете программу, которая будет выводить адреса на экран. Вероятно, вы захотите, чтобы адреса появлялись в виде нескольких строк. Если необходимо переместить курсор в начало следующей строки, можно поместить символ новой строки (**\n**) в выходной поток. В **C++** вам предоставляется два разных способа генерации новой строки. Во-первых, вы можете поместить символы **\n** внутри символьной строки. Например, следующая программа, содержащаяся в файле **TWOLINES.CPP**, отображает свой вывод в виде двух строк, используя символ новой строки:

**#include <iostream.h>**

**void main(void)**

**{**

**cout << "Это строка один\nЭто строка два";**

**}**

На экране будет выведено следующее сообщение:

**Это строка один**

**Это строка два**

Если вы не выводите символьную строку, можете поместить символ новой строки внутри одинарных кавычек. Например, следующая программа, содержащаяся в файле**NEWLINES.CPP** выводит числа **1, 0, 0** и **1**, каждое на своей собственной строке:

**#include <iostream.h>**

**void main(void) {**

**cout << 1 << '\n' << 0 << '\n' << 0 << '\n' << 1;**

**}**

В дополнение к использованию символа новой строки для продвижения курсора в начало следующей строки ваши программы могут использовать символ **endl** (конец строки). Следующая программа, содержащаяся в файле, **ENDL.CPP** иллюстрирует использование **endl** для продвижения курсора в начало новой строки:

**#include <iostream.h>**

**void main(void)**

**{**

**cout << "А теперь..." << endl**

**<< "Учимся программировать на языке C++";**

**}**

Как и ранее, на экране будет отображен вывод в виде двух строк.

Наконец, следующая программа, содержащаяся в файле,**ADDRESS.CPP** выводит адрес издательства "Jamsa Press" в несколько строк:

**#include <iostream.h>**

**void main(void)**

**{**

**cout << "Jamsa Press" << endl;**

**cout << "2975 South Rainbow, Suite I" << endl;**

**cout << "Las Vegas, NV89102" << endl;**

**}**

В дополнение к символу новой строки, позволяющему программам продвигать курсор в начало новой строки, можно использовать специальные символы, перечисленные в табл. 2.5.2-7.

Таблица 2.5.2-7

|  |  |
| --- | --- |
| **Символ** | **Назначение** |
| **\а** | Сигнальный (или звонок) символ |
| **\Ь** | Символ возврата |
| **V** | Символ перевода страницы |
| **\n** | Символ новой строки |
| **\г** | Возврат каретки (не перевод строки) |
| **\t** | Символ горизонтальной табуляции |
| **\v** | Символ вертикальной табуляции |
| **\\** | Символ обратный слеш |
| **\?** | Знак вопроса |
| **\'** | Одинарные кавычки |
| **\"** | Двойные кавычки |
| **\0** | Нулевой символ |
| **\000** | Восьмеричное значение, например \007 |
| **\xhhhh** | Шестнадцатеричное значение, например \xFFFF |

При использовании специальных символов, перечисленных в табл. 3.5.2-1, следует располагать их внутри одинарных кавычек, если вы используете данные символы сами по себе, например **'\n'**, или внутри двойных кавычек, если вы используете их внутри строки, например "**Привeт\nMup!**".

Следующая программа, содержащаяся в файле,**SPECIAL.CPP** использует специальные символы сигнала (**\а**) и табуляции (**\t**) для выдачи звука на встроенный динамик компьютера и затем выводит слова **Звонок Звонок Звонок**, разделенные

табуляцией:

**#include <iostream.h>**

**void main(void)**

**{**

**cout << "3вонок\a\tЗвонок\a\tЗвонок\a";**

**}**

Используя **cout**, можно перенаправить вывод программы на устройство или файл

с помощью операторов переназначения вывода операционной системы. Однако если программы сталкиваются с ошибкой, вы, вероятно, не захотите, чтобы сообщение об ошибке было перенаправлено с экрана. Перенаправление сообщений об ошибках в файл может скрыть от пользователя факт появления ошибки.

Если программе нужно вывести сообщение об ошибке, необходимо использовать выходной поток **cerr**.**C++** связывает **cerr**со стандартным устройством ошибок операционной системы. Следующая программа, содержащаяся в файле,**CERR.CPP** использует выходной поток **cerr** для вывода на экран сообщения "Это сообщение появляется всегда ":

**#include <iostream.h>**

**void main(void)**

**{**

**cerr << "Это сообщение появляется всегда";**

**}**

Так как операционная система не позволит программам перенаправить вывод, записываемый на стандартное устройство ошибок, сообщение появится на вашем экране.

Несколько предыдущих программ выводили числа на экран. Чтобы гарантировать правильное отображение этих чисел (с правильной расстановкой пробелов), программы включали пробелы до и после чисел. При выводе на **cout** или **cerr**программы могут указать ширину вывода каждого числа, используя модификатор **setw**(установка ширины). С помощью **setw** программы указывают минимальное количество символов, занимаемое числом. Например, следующая программа, содержащаяся в файле**SETW.CPP** использует модификатор **setw** для выбора ширины 3, 4, 5 и 6 для числа 1001. Чтобы использовать модификатор **setw**, программа должна включать заголовочный файл

**iomanip.h**:

**#include <iostream.h>**

**#include < iomanip.h >**

**void main (void)**

**{**

**cout << "Моелюбимоечислоравно" << setw(3) << 1001 << endl;**

**cout << "Мое любимое число равно" << setw(4) << 1001 << endl;**

**cout << "Мое любимое число равно" << setw(5) << 1001 << endl;**

**cout << "Мое любимое число равно" << setw(6) << 1001 << endl;**

**}**

Когда вы откомпилируете и запустите эту программу, на экране появится следующий вывод:

**Мое любимое число равно1001**

**Мое любимое число равно1001**

**Мое любимое число равно 1001**

**Мое любимое число равно 1001**

Если вы указываете ширину с помощью **setw,** указывается минимальное количество символьных позиций, занимаемых числом. В предыдущей программе модификатор **setw(3**) указывал минимум три символа. Однако так как число 1001 требует больше трех символов, **cout**использовал реально требуемое количество, которое в данном случае равнялось четырем.

Следует отметить, что при использовании **setw** для выбора ширины, указанная ширина действительна для вывода только одного числа. Если необходимо указать ширину

для нескольких чисел, необходимо использовать **setw** несколько раз.

***Форматный ввод и вывод - функции printf и scanf.*** Рассмотрим две функции – printf и scanf, предназначенные для реализации форматированного вывода и ввода данных

Эти функции позволяют преобразовывать численные величины в символьное представление и обратно. Они также позволяют генерировать и интерпретировать форматные строки. Функция printf имеет следующий синтаксис:

**printf(“<управляющая строка>”[,<список аргументов>]);**

преобразует, определяет формат и печатает свои аргументы в стандартный вывод под управляющей строки. Управляющая строка содержит два типа объектов: обычные символы, которые просто копируются в выходной поток, и спецификации преобразований, каждая из которых вызывает преобразование и печать очередного аргумента **printf .**

Каждая спецификация преобразования начинается с символа

**%** и заканчивается символом преобразования. Между **%** и символом преобразования могут находиться:

**-** знак минус, который указывает о выравнивании преобразованного аргумента по левому краю его поля. А его отсутствие указывает, что данное при выводе прижимается к правому краю.

**-** Строка цифр, задающая минимальную ширину поля. Преобразованное число будет напечатано в поле, по крайней мере, этой ширины, а если необходимо, то и в более широком. Если преобразованный аргумент имеет меньше символов, чем указанная ширина поля, то он будет дополнен слева (или справа, если было указано выравнивание по левому краю) заполняющими символами до этой ширины. Заполняющим символом обычно является пробел, а если ширина поля указывается с лидирующим нулем, то этим символом будет нуль (лидирующий нуль в данном случае не означает восьмеричной ширины поля).

- Точка, которая отделяет ширину поля от следующей строки цифр.

- Строка цифр (точность), которая указывает максимальное число символов строки, которые должны быть напечатаны, или число печатаемых справа от десятичной точки цифр для переменных типа **float** или **double.**

- Модификатор длины **l,** который указывает, что соответствующий элемент данных имеет тип long, а не int.

Ниже приводятся символы преобразования и их смысл:

* **d** - аргумент преобразуется к десятичному виду.
* **o** - Аргумент преобразуется в беззнаковую восьмеричную форму (без лидирующего нуля).
* **x** - Аргумент преобразуется в беззнаковую шестнадцатеричную форму (без лидирующих **0X**).
* **u**- Аргумент преобразуется в беззнаковую десятичную форму.
* **c** - Аргумент рассматривается как отдельный символ.
* **s** - Аргумент является строкой: символы строки печатаются до тех пор, пока не будет достигнут нулевой символ или не будет напечатано количество символов, указанное в спецификации точности.
* **e** - Аргумент, рассматриваемый как переменная типа **float** или **double** в форме с плавающей точкой.
* **f** - Аргумент, рассматриваемый как переменная типа **float** или **double** в форме с фиксированной точкой.
* **g**- Используется или формат e или f, какой короче; незначащие нули не печатаются.
* **p**– значением аргумента является указатель (адрес).

Если идущий за **%** символ не является символом преобразования, то печатается сам этот символ; следовательно, символ **%** можно напечатать, указав **%%.**

Например, в результате вызова функции

**printf(“\ti=%ld; \nj=%d, a=%6.2f.\n”, i, j, a);**

при условии, что i=123456, j=127, a=86,531, будет выведена информация в виде:

**i=123456;**

**j=127, a=86.53.**

Предостережение: **printf** использует свой первый аргумент для определения числа последующих аргументов и их типов. Если количество аргументов окажется недостаточным или они будут иметь несоответственные типы, то возникнет путаница, и вы получите бессмысленные результаты.

Осуществляющая ввод функция **scanf**является аналогом **printf** и позволяет проводить в обратном направлении многие из тех же самых преобразований. Функция

**scanf (“<управляющая строка>”[,<список аргументов>]);**

читает символы из стандартного ввода, интерпретирует их в соответствии с форматом, указанном в управляющей строке, и помещает результаты в остальные аргументы. Управляющий аргумент описывается ниже; другие аргументы, каждый из которых должен быть указателем, определяют, куда следует поместить соответствующим образом преобразованный ввод.

Управляющая строка обычно содержит спецификации преобразования, которые используются для непосредственной интерпретации входных последовательностей. Управляющая строка может содержать:

* пробелы, табуляции или символы новой строки ("символы пустых промежутков"), которые игнорируются.
* обычные символы (не **%**), которые предполагаются совпадающими со следующими отличными от символов пустых промежутков символами входного потока.
* спецификации преобразования, состоящие из символа %, необязательного символа подавления присваивания \*, необязательного числа, задающего максимальную ширину поля и символа преобразования.

Спецификация преобразования управляет преобразованием следующего поля ввода. нормально результат помещается в переменную, которая указывается соответствующим аргументом. Если, однако, с помощью символа \* указано подавление присваивания, то это поле ввода просто пропускается и никакого присваивания не производится. Поле ввода определяется как строка символов, которые отличны от символов простых промежутков; оно продолжается либо до следующего символа пустого промежутка, либо пока не будет исчерпана ширина поля, если она указана. Отсюда следует, что при поиске нужного ей ввода, функция scanf будет пересекать границы строк, поскольку символ новой строки входит в число пустых промежутков.

Аргументы scanf должны быть указателями на соответствующие значения (для этого перед именем переменной, не являющейся указателем, записывается символ взятия адреса **&**).

Например, обращение

**intI;**

**floatX;**

**char NAME[50];**

**scanf("&d %f %s", &I, &X, NAME);**

со строкой на вводе

**25 54.32E-1 THOMPSON**

приводит к присваиванию I значения 25,X - значения 5.432 и NAME - строки "THOMPSON", надлежащим образом законченной символом \ 0. эти три поля ввода можно разделить столькими пробелами, табуляциями и символами новых строк, сколько вы пожелаете. В этом примере NAME является указателем и, следовательно, перед ним не нужно помещать знак &.

По мере усложнения ваших программ количество содержащихся в них операторов может сделать программы слишком трудными для понимания. Поскольку другим программистам может потребоваться понять и, возможно, изменить ваши программы, вам следует делать программы более удобочитаемыми. Способы улучшения удобочитаемости программ включают:

* использование понятных по смыслу имен переменных, описывающих их применение;
* использование подходящих отступов и выравнивания;
* использование пустых строк для разделения несвязанных операторов;
* использование комментариев, которые объясняют работу программы.

При создании программ вы можете поместить в исходном файле замечания, которые объясняют работу программы. Такие замечания (называемые комментариями) не только

помогают другим программистам понять вашу программу, но могут напомнить, почему программа содержит определенные операторы, если вы ее посмотрите через несколько месяцев. Для размещения комментария в своих программах на C++ просто поставьте два знака прямого слеша (//), как показано ниже:

**// Это комментарий**

Когда компилятор C++ встречает двойной слеш, он игнорирует весь оставшийся на этой строке текст. По крайней мере, размещайте в начале каждой программы комментарии, которые указывают, кто написал программу, когда и почему:

**// Программа: BUDGET.CPP**

**// Программист: Kris Jamsa**

**// дата создания: 1-10-96**

**//**

**// Цель: Ежемесячная информация о бюджете**.

Так же для размещения комментария можно использовать символы /\* - начало текста , \*/ - конец.

При выполнении вашей программой различных процессов вам следует поместить до или после определенных операторов комментарии, которые объясняют их назначение. Например, рассмотрим следующий оператор присваивания:

**distance\_to\_the\_moon = 238857; // Расстояние в милях**

Комментарий справа от оператора присваивания обеспечивает дополнительную информацию всем, кто читает программу. У начинающих программистов часто возникают проблемы, связанные с тем, что и когда комментировать. Как правило, вы включаете немного комментариев в ваши программы. Поэтому убедитесь, что они полезны. Следующие комментарии не дают дополнительной информации программистам, читающим код:

**аде =32; // Присвоить 32 переменной age**

**salary = 25000.75; // Присвоить 25000.75 переменной salary**

Цель использования комментариев заключается в том, чтобы объяснить работу программы. При создании программ включайте комментарии, которые объясняют работу программы. Если другим программистам понадобится изменить вашу программу, они смогут воспользоваться комментариями, чтобы понять поведение программы. Хорошие программы должны быть легки для чтения и понимания. Комментарии улучшают удобочитаемость вашей программы.

Замечание: В дополнение к использованию комментариев для улучшения удобочитаемости своей программы вам следует использовать пустые строки для разделения несвязанных операторов. Когда компилятор C++ встречает пустую строку, он просто пропускает ее.

*Определение и вызов функций*. Функция - это совокупность **объявлений**, и **операторов**, обычно предназначенная для решения определенной задачи. Каждая функция должна иметь имя, которое используется для ее объявления, определения и вызова. В любой программе на **С++** должна быть функция с именем **main()** (главная функция), именно с этой функции, в каком бы месте программы она не находилась, начинается выполнение программы.

При вызове функции ею при помощи аргументов (**формальных параметров**) могут быть переданы некоторые значения (**фактические параметры**) используемые во время выполнения функции. Функция может возвращать одно значение. Это **возвращаемое значение** и есть результат выполнения функции, который при выполнении программы подставляется в точку вызова функции, где бы этот вызов не встретился. Допускается также использовать функции без аргументов и функции не возвращающие никаких значений. Действие таких функций может состоять, например, в изменении значений некоторых переменных, выводе на печать некоторых текстов и т.п.

С использованием функций в языке С++ связаны три понятия:

* **определение функции** (описание действий, выполняемых функцией),
* **объявление функции** (задание формы обращения к функции)
* **вызов функции** (выполнение).

**Определение функции** задает тип возвращаемого значения, имя функции, типы и число формальных параметров, а также объявления переменных и операторы, называемые телом функции, и определяющие действие функции. В определении функции также может быть задан класс памяти. Например,

**int rus (unsigned char r)**

**{**

**if (r>=' А ' && c<=' ё ') return 1;**

**else return 0;**

**}**

В данном примере определена функция с именем **rus**, имеющая один параметр с именем **r** и типом **unsigned char**. Функция возвращает целое значение, равное **1**, если параметр функции является буквой русского алфавита, или **0** в противном случае.

В языке С++ нет требования, чтобы определение функции обязательно предшествовало ее вызову. **Определения используемых функций могут следовать за определением функции main(), перед ним, или находится в другом файле.**

Однако для того, чтобы компилятор мог осуществить проверку соответствия типов передаваемых фактических параметров типам формальных параметров и в случае необходимости выполнить нужные преобразования, до вызова функции требуется поместить объявление (прототип) функции.

**Объявление функции** имеет такой же вид, что и определение функции, с той лишь разницей, что тело функции отсутствует, и имена формальных параметров тоже могут быть опущены. Для функции, определенной в последнем примере, прототип может иметь вид

**int rus (unsignedcharr); или rus (unsignedchar);**

В программах на языке **С++** широко используются, так называемые, библиотечные функции, т.е. функции предварительно разработанные и записанные в библиотеки. Прототипы библиотечных функций находятся в специальных заголовочных файлах, поставляемых вместе с библиотеками в составе систем программирования, и включаются в программу с помощью директивы #include.

Если объявление функции не задано, то по умолчанию строится прототип функции на основе анализа первой ссылки на функцию, будь то вызов функции или определение. Однако такой прототип не всегда согласуется с последующим определением или вызовом функции. **Рекомендуется всегда задавать прототип функции**. Это позволит компилятору либо выдавать диагностические сообщения, при неправильном использовании функции, либо корректным образом регулировать несоответствие аргументов, устанавливаемое при выполнении программы.

Объявление параметров функции при ее определении может быть выполнено в, так называемом, "старом стиле", при котором в скобках после имени функции следуют только имена параметров, а после скобок объявления типов параметров. Например, функция **rus** из предыдущего примера может быть определена следующим образом:

**int rus (r)**

**unsigned char r;**

**{ ... /\* тело функции \*/ ... }**

В соответствии с синтаксисом языка **С++** определение функции имеет следующую форму:

**[спецификатор-класса-памяти] [спецификатор-типа] имя-функции**

**([список-формальных-параметров])**

**{ тело-функции }**

Необязательный **спецификатор-класса-памяти** задает класс памяти функции, который может быть **static** или **extern**. Подробно классы памяти будут рассмотрены далее.

**Спецификатор типа** функции задает тип возвращаемого значения и может задавать любой тип. Если **спецификатор типа** не задан, то предполагается, что функция возвращает значение типа **int**.

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

**return [выражение] ;**

Значение **выражения**, если оно задано, возвращается в вызывающую функцию в качестве значения вызываемой функции. Если выражение опущено, то возвращаемое значение не определено. Если в какой-либо функции отсутствует оператор **return**, то передача управления в вызывающую функцию происходит после выполнения последнего оператора вызываемой функции. При этом возвращаемое значение не определено. Если функция не должна иметь возвращаемого значения, то ее нужно объявлять с типом **void**.

Функция возвращает значение, если ее выполнение заканчивается оператором **return**, содержащим некоторое выражение. Указанное выражение вычисляется, преобразуется, если необходимо, к типу возвращаемого значения и возвращается в точку вызова функции в качестве результата. Если оператор **return** не содержит выражения или выполнение функции завершается после выполнения последнего ее оператора (без выполнения оператора **return**), то возвращаемое значение не определено. Функция не может возвращать массив или функцию, но может возвращать указатель на любой тип, в том числе и на массив и на функцию. Тип возвращаемого значения, задаваемый в определении функции, должен соответствовать типу в объявлении этой функции. Для функций, не использующих возвращаемое значение, должен быть использован тип **void**, указывающий на отсутствие возвращаемого значения. Если функция определена как функция, возвращающая некоторое значение, а в операторе **return** при выходе из нее отсутствует выражение, то поведение вызывающей функции после передачи ей управления может быть непредсказуемым.

**Список-формальных-параметров** - это последовательность объявлений формальных параметров, разделенная запятыми. Формальные параметры - это переменные, используемые внутри тела функции и получающие значение при вызове функции путем копирования в них значений соответствующих фактических параметров. Если функция не использует параметров, то наличие круглых скобок обязательно, а вместо списка параметров рекомендуется указать слово **void**.

Порядок и типы формальных параметров должны быть одинаковыми в определении функции и во всех ее объявлениях. Для формального параметра можно задавать класс памяти **register**, при этом для величин типа **int** спецификатор типа можно опустить.

Идентификаторы формальных параметров используются в теле функции в качестве ссылок на переданные значения. Эти идентификаторы не могут быть переопределены в блоке, образующем тело функции, но могут быть переопределены во внутреннем блоке внутри тела функции.

**Тело функции** - это составной оператор, содержащий операторы, определяющие действие функции.

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

**Объявление (прототип)** функции имеет следующий формат:

**[спецификатор-класса-памяти] [спецификатор типа]**

**имя-функции ([список-формальных-параметров]) [,список-имен-функций];**

В отличие от определения функции, в прототипе за заголовком сразу же следует точка с запятой, а тело функции отсутствует. Прототип - это явное объявление функции, которое предшествует определению функции. Тип возвращаемого значения при объявлении функции должен соответствовать типу возвращаемого значения в определении функции.

**Вызов функции** имеет следующий формат:

**адресное\_выражение ([список-выражений])**

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

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

**Фактический параметр** может быть величиной любого основного типа, структурой, объединением, перечислением, или указателем на объект любого типа. Массив и функция не могут быть использованы в качестве фактических параметров, но можно использовать указатели на эти объекты.

Функция **main()**, с которой начинается выполнение **С++** программы, может быть определена с параметрами, которые передаются из внешнего окружения, например, из командной строки. Во внешнем окружении действуют свои правила представления данных, а точнее, все данные представляются в виде строк символов. Поэтому в дальнейшем будем предполагать, что в главной функции параметры отсутствуют, и, что эта функция не возвращает ни какого значения, то есть **void main(void)**

### 2.5.2.3. Средства программирования разветвляющихся структур

На практике редко встречаются задачи, решение которых представляется только в виде алгоритмов линейной структуры. Как правило, приходится решать задачи, в которых в зависимости от выполнения некоторого логического условия необходимо органи­зовать вычисления по одной из нескольких ветвей алгоритма. Алгоритмы решения таких задач строятся с использованием раз­ветвляющихся алгоритмических структур. В результате проверки ***логического выражения*** (“**Истина**” или “**Ложь**”) управление передается ветке “**Да**” или “**Нет**”.

**В С++ не реализован логический тип данных, но имеется большой набор логических операций. В этих операциях вместо логических величин можно использовать любую арифметическую величину или указатель. При этом любое значение отличное от нуля считается истиной, а нулевое значение ложью.**

Логические выражения, в отличие от арифметических, содержат ***операции отношения*** (**=, <, >, <=, >=, <>**), а также еще пять ***логических операций***, записанных далее в порядке убывания приоритета: отрицание - **!**, логическое умножение (логическое “И”) - &&, логическое сложение (логическое “ИЛИ”) - ||, исключающее “ИЛИ” - **^**, эквивалентность - **EQV**.

Результатом выполнения логического выражения может быть “**Истина**” (**не 0)** или “**Ложь**” (**0**). Например, **NOT 7<9** всегда прини­мает значение “**Ложь**”, что соответствует значению **0**, а **4>3 AND 7<9** принимает значение “**Истина**”, что соответствует **-1**.

В табл. 2.5.2-8 приведено описание логических операций. Здесь **T** (**true**) - “**Истина**”, а **F** (**false**) “**Ложь**”.

К логическим операциям относятся операция логического И (&&) и операция логического ИЛИ. Операнды логических выражений вычисляются слева направо. Логические операции оценивают каждый операнд с точки зрения его эквивалентности нулю. Результатом логической операции является 0 или 1, тип результата int.

Операция логического **И (&&)** вырабатывает значение 1, если оба операнда имеют нулевые значения. Если один из операндов равен 0, то результат также равен 0. Если значение первого операнда равно 0, то второй операнд не вычисляется.

Операция логического **ИЛИ (||)** выполняет над операндами операцию включающего ИЛИ. Она вырабатывает значение 0, если оба операнда имеют значение 0, если какой-либо из операндов имеет ненулевое значение, то результат операции равен 1. Если первый операнд имеет ненулевое значение, то второй операнд не вычисляется.

Таблица 2.5.2-8

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Значения операндов** | | **! x** | **x&&y** | **x || y** | **x ^ y** | **х ==y** |
| X | Y |  |  |  |  |  |
| T | T | F | T | T | F | T |
| T | F | F | F | T | T | F |
| F | T | T | F | T | T | F |
| F | F | T | F | F | F | T |

 Разветвляющиеся алгоритмические структуры в языках про­граммирования реализуются **операторами условного перехода*.*** В **C++** для программирования условных переходов используются три типа операторов: **if**,**switch** и **break**.

**Оператор ifпозволяет направить вычислительный процесс на два варианта в зависимости от значения некоторого условия. Оператор if**имеет следующий**формат:**

**if (выражение) оператор\_1; [ else оператор\_2;]**

Порядок выполнения оператора следующий: если значение выражения не нуль (“**Истина**”), то выполняется **оператор\_1**, в противном случае выполняются **оператор\_2**, стоящие после **else**. Конструкция **else** может от­сутствовать (в формате она взята в квадрат­ные скобки) – случай усеченного разветвления.

**Выражение** может быть логическим или арифметическим. Ненулевое значение арифметического выражения интерпретируется как “**Истина**”, а нулевое - как “**Ложь**”.

Рассмотрим примеры записи однострочного оператора **if**:

**if (X > 0) Y=sqrt(X);**

**else Y=1;**

**if (D) EPSILON=0.001;**

**if (LAMBDA < 1 || BETA == 3 ) {cout << LAMBDA<<BETA ; X=0;}**

**if (X < 10) Y=1;**

**else if (X < 100) Y=2;**

**else Y=4;**

****if ( i<j ) { j = 0; i--; }****

****else { j = i-3; i++; }****

**Другим способом организации выбора из множества различных вариантов является использование специального оператора выбора switch.**

**Оператор switch предназначен для организации выбора из множества различных вариантов. Оператора switch имеет следующий формат:**

**switch ( выражение )**

**{ [объявление]**

**:**

**[ case константное\_выражение\_1]: [ список\_операторов\_1]**

**[ case константное\_выражение\_2]: [ список\_операторов\_2]**

**:**

**:**

**[ default: [ список\_операторов\_default ] ]**

**}**

**Выражение, следующее за ключевым словом switch в круглых скобках, может быть любым выражением, допустимыми в языке С++, значение которого должно быть целым. Отметим, что можно использовать явное приведение к целому типу, однако необходимо помнить о тех ограничениях и рекомендациях по преобразованию данных, о которых говорилось выше.**

**Значение этого выражения является ключевым для выбора из нескольких вариантов. Тело оператора switch состоит из нескольких операторов, помеченных ключевым словом case с последующим константным выражением.**

**Порядок выполнения оператора switch следующий:**

* **вычисляется выражение в круглых скобках;**
* **вычисленные значения последовательно сравниваются с константными\_выражениями следующими за ключевыми словами case;**
* **если одно из константных\_выражений совпадает со значением выражения, то управление передается на оператор, помеченный соответствующим ключевым словом case;**
* **если ни одно из константных\_выражений не равно выражению, то управление передается на оператор, помеченный ключевым словом default, а в случае его отсутствия управление передается на следующий послеswitch оператор.**

**Программист должен сам позаботиться о выходе из case, если это необходимо. Чаще всего для этого используется оператор break. Напрмер:**

****int i=2;****

****switch (i)****

****{****

****case 1: i += 2;****

****case 2: i \*= 3;****

****case 0: i /= 2;****

****case 4: i -= 5;****

****default**: ;**

****}****

**Выполнение оператора switch начинается с оператора, помеченного case 2. Таким образом, переменная i получает значение, равное 6, далее выполняется оператор, помеченный ключевым словом case 0, а затем case 4, переменная i примет значение 3, а затем значение -2. Оператор, помеченный ключевым словом default, не изменяет значения переменной.**

****char ZNAC;****

****int x,y,z;****

****switch (ZNAC)****

****{****

****case '+': x = y + z; break;****

****case '-': x = y - z; break;****

****case '\*': x = y \* z; break;****

****case '/': x = u / z; break;****

****default : ;****

****}****

**Использование оператора break позволяет в необходимый момент прервать последовательность выполняемых операторов в теле оператора switch и передать управление оператору, следующему за switch.**

**Оператор break может быть использован только внутри операторов switch, for, while или do while и обеспечивает прекращение выполнения самого внутреннего из содержащих его операторов. После выполнения оператора break управление передается оператору, следующему за прерванным.**

**Заметим, что оператор break нельзя использовать для выхода из нескольких вложенных циклов, а составной оператор, состоящий из двух операторов break, эквивалентен одному оператору break.**

**По своей сути оператор break является оператором перехода и операторы, записанные после него, выполняться не будут, если только им не будет передано управление с помощью других операторов перехода.**

### 2.5.2.4. Средства программирования регулярных циклических структур

**Алгоритмы регулярной циклической структуры** характеризуются заданным (известным) числом повторений: иногда это число за­дается явно, иногда его предварительно вычисляют. К таким ал­горитмам относятся: алгоритмы вычисления значений функций одной или нескольких переменных, у которых аргументы изменяются в известном диапазоне и с заданным шагом изме­нения; алгоритмы вычисления конечных сумм и произведений и многие дру­гие. Программирование такой алгоритмической структуры на языке **С++** производится при помощи оператора **for...**, имеющего следующий формат:

**for ( выражение\_1 ; выражение\_2 ; выражение\_3 ) тело\_цикла**

**Выражение\_1** используется для установления начального значения переменной, управляющей циклом.

**Выражение\_2** это выражение, определяющее условие, при котором тело цикла будет выполняться.

**Выражение\_3** определяет изменение переменных, управляющих циклом после каждого выполнения тела цикла.

В качестве **тела\_цикла** может быть использован любой оператор, в том числе пустой или составной.

Любое из выражений, а также все сразу, могут быть опущены, при этом разделяющие их символы « ; » пропускать нельзя.

Схема выполнения оператора **for**:

1. Вычисляется **выражение\_1**.
2. Вычисляется **выражение\_2.**
3. Если значения **выражения\_2** отлично от нуля (истина), выполняется тело цикла, вычисляется **выражение\_3** и осуществляется переход к пункту 2, если **выражение\_2** равно нулю (ложь), выполнение оператора **for** завершается и управление передается на оператор, следующий за оператором **for.** При отсутствии **выражения\_2** его значение равно истина.

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

**int main ( )**

**{**

**int i,b;**

**for ( i=1; i<10; i++) b=i\*i;**

**return 0;**

**}**

В этом примере вычисляются квадраты чисел от 1 до 9.

При организации вложенных циклов (цикл в цикле) используются пары операторов **for…** , которые не должны “пересекаться” друг с другом. Для преждевременного выхода из цикла по условию используется оператор **break;**.

**Заметим, что оператор break нельзя использовать для выхода из нескольких вложенных циклов, а составной оператор, состоящий из двух операторов break, эквивалентен одному оператору break. Т .е. вследующем фрагменте**

****for (i=0; i<100; i++)****

****for (j=0; j<100; j++)****

****{ :****

****if (a[i][j]<0) { break; break; }****

****:****

****}****

**при выполнении условия a[i][j]<0 завершится выполнение только внутреннего цикла по переменной j, а выполнение внешнего цикла по переменной i продолжится, несмотря на то, что оператор break повторен дважды.**

**По своей сути оператор break является оператором перехода и операторы, записанные после него, выполняться не будут, если только им не будет передано управление с помощью других операторов перехода.**

**Для выхода из обоих циклов можно использовать дополнительную переменную, которая будет принимать значение отличное от нуля только при необходимости выхода из внешнего цикла.** Например,

****for (i=0; i<100; i++)****

****{ for (br2=j=0; j<100; j++)****

****{ :****

****if ( a[i][j]<0 ) { br2=1; break;}****

****:****

****}****

****if (br2) break;****

****:****

****}****

**В приведенном примере сразу после завершения внутреннего цикла проверяется условие окончания внешнего цикла.**

### 2.5.2.5. Средства программирования итеративных циклических структур

Отличительной особенностью алгоритмов ***итеративных циклических структур*** является заранее неиз­вестное число повторений цикла. К ним от­носятся, например, алгоритм проверки ввода исходных данных, алгоритмы вычисления отдельных членов, суммы или произведения членов бесконечных последо­ва­тельностей, алго­ритмы вычислений по итерационным формулам и некоторые другие алгоритмы. При программировании алгоритмов итеративных циклических структур ис­пользуется операторы **while** и **do while**.

Первый оператор **while** используется при программировании итеративной циклической структуры с **предусловием** и имеет следующий формат:

**while ( выражение ) тело\_цикла ;**

В качестве выражения допускается использовать любое выражение языка **С++,** а в качестве тела любой оператор, в том числе пустой или составной. Схема выполнения оператора **while** следующая:

1. Вычисляется **выражение**.
2. Если **выражение** ложно, то выполнение оператора **while** заканчивается и выполняется следующий по порядку оператор. Если выражение истинно, то выполняется **тело\_цикла** оператора **while**.
3. Процесс повторяется с пункта 1.

Так же как и при выполнении оператора **for**, в операторе **while** вначале происходит проверка условия. Поэтому оператор **while** удобно использовать в ситуациях, когда тело оператора не всегда нужно выполнять. Например,

**double summa (float x, int n)**

**{**

**double s=0, a=pow(x,2);**

**int i=0;**

**while(i<n)**

**{**

**s=s+a;**

**a= -a\*x/((2\*i+2)\*(2\*i+3));**

**i++;**

**}**

**return s;**

**}**

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

**do тело\_цикла while (выражение);**

Схема выполнения оператора **do while**:

1. Выполняется **тело\_цикла** (которое может быть составным оператором).
2. Вычисляется **выражение.**
3. Если **выражение** ложно, то выполнение оператора **do while** заканчивается и выполняется следующий по порядку оператор. Если выражение истинно, то выполнение оператора продолжается с пункта 1.

Чтобы прервать выполнение цикла до того, как условие станет ложным, можно использовать оператор **break**. Рассмотрим следующий пример:

**float s=0;**

**int n=4;**

**int i=0;**

**do{**

**s=s+ (2\*i+1);**

**i++;**

**} while (i<=n);**

### 2.5.2.6. Средства описания и работы с одномерными массивами данных

**Массив** – это последовательность переменных одинакового типа, объединенных общим **именем**. Массив состоит из **элементов,** например: **одномерный массива(50) состоит из 51 элемента:a(0)**, **a(1) a(2) a(3) . . . a(50).**

В программе элементы одного и того же массива отличаются один от другого при помощи индексов. **Индексы** записываются вслед за именем в круглых скобках и могут представлять собой любое целочислен­ное выражение. Например: **d(24), a(2\*i+1), b(i,i-1).**

Количество индексов указывает **на размерность** массива. Так, в приведенном выше примере размерность массива **a(50)** равна единице, то есть массив одномерный. Максимальная размер­ность массива может достигать **60.**

В отличие от размерности, **размер массива** - это количество элементов в массиве. В нашем примере размер массива **а(50)** равен **51 (** с 0-го по 50**)**.

**Массивы** - это совокупность элементов одинакового типа (базового типа, такого как **double** или **float**, или более сложного) занимающих смежные участки памяти. Из объявления массива компилятор должен получить информацию о типе элементов массива и их количестве. Объявление массива имеет формат:

**спецификатор\_типа описатель [константное\_выражение];**

**Описатель** - это идентификатор.

**Спецификатор\_типа** задает тип элементов объявляемого массива. Элементами массива не могут быть функции и элементы типа **void**.

**Константное\_выражение** в квадратных скобках задает количество элементов массива. **Константное\_выражение** при объявлении массива может быть опущено (но сами скобки [] остаются), если при объявлении массив инициализируется или если массив объявлен как формальный параметр функции. Например,

**int array[4]; /\* array - массив из 4 элементов типа int \*/**

**double dd[10]; /\* dd - масив из 10 элементов типа double \*/**

**float \* ff [8] /\* ff - масив из 8 указателей на float \*/**

**enum { False, True } bool [6];**

**/\* bool - массив из 6 элементов перечислимого типа \*/**

Для обращения к элементам массива следует указывать имя массива и индекс элемента заключенный в квадратные скобки. Например, **array[2], dd[3].** Заметим, что в языке С++ первый элемент массива имеет индекс равный **0**, и обращение **dd[3]** обозначает обращение к четвертому элементу массива **dd**.

Элементы массивов могут участвовать в любых операциях как простые переменные соответствующего типа. Как и у всякой переменной типа массив должно быть значение. В языке СИ значением переменной типа массив является адрес памяти, используемой для хранения значений элементов массива. Значение переменной типа массив нельзя изменять во время выполнения.

**Индекс** массива часть имени переменной.   На самом деле индексация - это

1. выбор элемента в массиве;
2. справа от присваиваний и в выражениях - еще и разыменование, то есть взятие вместо имени переменной - значения, в ней хранящегося.

Массивы НЕЛЬЗЯ присваивать целиком, язык Си этого не умеет.

**int a[5]; int b[5];   a = b; /\* ошибка \*/**

Также нельзя присвоить значение сразу всем элементам (ячейкам) массива:

**a = 0; /\* ошибка \*/**

не делает того, что нами ожидалось, а является ошибкой. Для обнуления всех ячеек следует использовать цикл:

**int i;   for(i=0; i < 5; i++) /\* для каждого i присвоить a[i] = 0; \*/ a[i] = 0;**

 В языке C++ существует сильная взаимосвязь между указателями и массивами, настолько сильная, что указатели и массивы действительно следует рассматривать одновременно. Например, когда объявляется массив в виде **int array[25]**, то происходит не только выделение памяти для двадцати пяти элементов массива, но и для указателя с именем **array**, значение которого равно адресу первого по счету (нулевого) элемента массива, т.е. сам массив остается безымянным, а доступ к элементам массива осуществляется через указатель с именем **array**. С точки зрения синтаксиса языка указатель **array** является константой, значение которой можно использовать в выражениях, но изменить это значение нельзя. Любую операцию, которую можно выполнить с помощью индексов массива, можно сделать и с помощью указателей. Вариант с указателями обычно оказывается более быстрым, но и несколько более трудным для непосредственного понимания, по крайней мере, для начинающего. Описание

**int a[10];**

определяет массив размера 10, т.е. Набор из 10 последовательных объектов, называемых **a[0], a[1], ..., a[9].** Запись **a[i]** соответствует элементу массива через **i** позиций от начала. Если **pa** - указатель целого, описанный как  **int \*pa;** то присваивание **pa = &a[0];**

приводит к тому, что pa указывает на нулевой элемент массива **a.** Это означает, что **pa** содержит адрес элемента **a[0].** Теперь присваивание**x = \*pa;** будет копировать содержимое **a[0]** в **x.**

Если **pa** указывает на некоторый определенный элемент массива **a,** то по определению **pa+1** указывает на следующий элемент, и вообще **pa-i** указывает на элемент, стоящий на**i** позиций до элемента, указываемого **pa**, а **pa+i** на элемент, стоящий на **i**позиций после. Таким образом, если **pa** указывает на **a[0],** то**\*(pa+1)** ссылается на содержимое **a[1].** Выражение **pa+i**-это адрес элемента **a[i],** а **\*(pa+i)** содержимое **a[i].**

Эти замечания справедливы независимо от типа переменных в массиве **a**. Суть определения "добавления 1 к указателю", а также его распространения на всю арифметику указателей, состоит в том, что приращение масштабируется размером памяти, занимаемой объектом, на который указывает указатель. Таким образом, **i** в **pa+i** перед прибавлением умножается на размер объектов, на которые указывает **pa.**

Очевидно, существует очень тесное соответствие между индексацией и арифметикой указателей. В действительности компилятор преобразует ссылку на массив в указатель на начало массива. В результате этого имя массива является указательным выражением. Отсюда вытекает несколько весьма полезных следствий. Так как имя массива является синонимом местоположения его нулевого элемента, то присваивание **pa=&a[0]** можно записать как **pa= a.**

Еще более удивительным, по крайней мере, на первый взгляд, кажется тот факт, что ссылку на **a[i]** можно записать в виде **\*(a+i).** Эти две формы совершенно эквивалентны. Если применить операцию & к обеим частям такого соотношения эквивалентности, то мы получим, что **&a[i]** и **a+i** тоже идентичны: **a+i** - адрес **i**-го элемента от начала **a.** С другой стороны, если **pa** является указателем, то в выражениях его можно использовать с индексом: **pa[i]** идентично **\*(pa+i).**Любое выражение, включающее массивы и индексы, может быть записано через указатели и смещения и наоборот, причем даже в одном и том же утверждении.

Имеется одно различие между именем массива и указателем, которое необходимо иметь в виду: указатель является переменной, так что операции **pa = a** и **pa++** имеют смысл. Но имя массива является константой, а не переменной: конструкции типа **a = pa** или **a++,** или **p = &a** будут неправильными.

Когда имя массива передается функции, то на самом деле ей передается местоположение начала этого массива. Внутри вызванной функции такой аргумент является точно такой же переменной, как и любая другая, так что имя массива в качестве аргумента действительно является указателем, т.е. переменной, содержащей адрес.

Если память выделяется под объект в момент компиляции (до выполнения программы), то размер объекта обязательно известен. Такое выделение памяти называется статическим.

|  |  |
| --- | --- |
| **Статическое создание** | **Динамическое создание** |
| Создание в момент компиляции.  Размер массива известен  **float mas [100];** | Создание в момент выполнения. Размер массива неизвестен.  Создание объекта типа массив в 2 этапа:  1.В момент компиляции создается указатель на тип **float \*mas;**  2. Используется оператор **newchar \* psymb;**  **psymb=new char;** |
| -Если объект локальный, то удаляется автоматически по выходу из блока.  -Если объект глобальный, то удаляется автоматически, когда программа перестает работать | Для удаления используется **delete**  **float\*mas;**  **mas=new float[n];**  **delete[ ] mas;** |

Оператор **new** выделяет блок памяти, необходимый для размещения переменной или массива и возвращает указатель на этот блок. В операторе **new** необходимо указывать тип переменной, а для массива еще и число элементов. Например,

**int \*t, \*s;**

**t=new int;**

**s=new int [10];**

Указатель **t** будет указывать на переменную типа **int**, а указатель **s** на массив из 10 элементов типа **int.**

Оператор **delete** освобождает ранее занятую память. Например,

**//Статическое распределение памяти массива**

**#include <iostream.h>**

**int minmaxel (float\*mas, int n, int&min)**

**{**

**float max=\*mas; min=mas[0];**

**int imax; imax=imin=0;**

**for (int i=1; i<n; i++)**

**{**

**if (min>mas[i])**

**{**

**imin=i; min=\*(mas+i);**

**}**

**if (max<mas[i])**

**{**

**max=mas[i]; imax=i;**

**}**

**}**

**return imax;**

**}**

**void main ()**

**{**

**float mas [] {1, -3, 7, 8, 15, 16, 17, 1, 5};**

**int imin;**

**int imax=minmaxel(mas, 9, imin);**

**cout <<”максимальный элемент “<< mas [ imax ]<<” минимальный**

**элемент “<<mas[imin];**

**}**

**//Динамическое распределение памяти массива**

**#include < iostream . h >**

**int minmaxel (float\*mas, int n, int&min)**

**{**

**float max=\*mas;**

**min=mas[0];**

**int imax;**

**imax=imin=0;**

**for (int i=1; i<n; i++)**

**{**

**if (min>mas[i])**

**{**

**imin=i;**

**min=\*(mas+i);**

**}**

**if (max<mas[i])**

**{**

**max=mas[i];**

**imax=i;**

**}**

**}**

**return imax;**

**}**

**float \* vvod (int n)**

**{**

**float\* mas= new float [n];**

**cout<<” Введите ”<<n<<” чисел ”;**

**for (int i=0; i<n ; i++)**

**cin>>mas [i];**

**return mas;**

**}**

**void main ()**

**{**

**int n=4;**

**float\*mas=vvod(n); // 16, -2, 4, 2**

**int imin;**

**int imax=minmaxel(mas, 9, imin);**

**cout <<”максимальный элемент “<< mas [ imax ]<<” минимальный**

**элемент “<<mas[imin];**

**delete [] mas; }**

С++ автоматически передает массивы функциям, используя моделируемый вызов **по ссылке** — вызываемые функции могут изменять значения элементов в исходных массивах источника вызова.

Значение имени массива является адресом первого элемента массива. Поскольку в функцию передается начальный адрес массива, вызываемая функция знает, где хранится массив. Поэтому, когда вызываемая функция модифицирует элементы массива в теле функции, она модифицирует реальные элементы массива в их истинных ячейках памяти.

Передача массивов с помощью моделируемого **вызова по ссылке** ощутимо влияет на производительность. Если бы массивы передавались по значению, передавалась бы копия каждого элемента. Для больших, часто передаваемых массивов это привело бы к значительному потреблению времени и памяти для хранения копий массивов .

Для предотвращения модификации исходного массива внутри тела функции можно в определении функции применять к параметру массив спецификатор типа **const** . Это еще один пример принципа наименьших привилегий. Функциям не должно быть позволено модифицировать массивы без крайней необходимости .

Следует отметить, что независимо от конкретной задачи, ал­горитмы формирования и обработки массивов обычно строятся с исполь­зованием регулярных циклических структур, а при работе с многомерными массивами с использованием вложенных циклов (цикл в цикле).

### 2.5.2.7. Средства описания и работы с двумерными массивами данных

Как известно, **массив** - последовательность переменных одинакового типа, объединенных общим **именем**. Например: **двумерный массивb(3, 2**):

**b(0,0) b(0,1) b(0,2)**

**b(1,0) b(1,1) b(1,2)**

**b(2,0) b(2,1) b(2,2)**

**b(3,0) b(3,1) b(3,2)**

Количество индексов указывает **на размерность** массива. Так, в приведенном выше примере размерность массива **b(3,2)** - **2** (двумерный). А размер массива **b(3,2)** равен**12**-ти.

В языке **С++** определены только одномерные массивы, но поскольку элементом массива может быть массив, можно определить и многомерные массивы. Они формализуются списком константных\_выражений следующих за идентификатором массива, причем каждое константное\_выражение заключается в свои квадратные скобки.

Каждое константное\_выражение в квадратных скобках определяет число элементов по данному измерению массива, так что объявление двухмерного массива содержит два константных\_выражения, трехмерного три и т.д. Например:

**int w[2][3]; /\* двумерный массив из двух строк и трех столбцов**

**w[0][0] w[0][1] w[0][2]**

**w[1][0] w[1][1] w[1][2] \*/**

**double f[10]; /\* вектор из 10 элементов имеющих тип double \*/**

Указатели на многомерные массивы в языке **С++** - это массивы массивов, т.е. такие массивы, элементами которых являются массивы. При объявлении таких массивов в памяти компьютера создается несколько различных объектов. Например, при выполнении объявления двумерного массива **int arr[4][3]** в памяти выделяется участок для хранения значения переменной **arr** , которая является указателем на массив из четырех указателей. Для этого массива из четырех указателей тоже выделяется память. Каждый из этих четырех указателей содержит адрес массива из трех элементов типа**int** , и, следовательно, в памяти компьютера выделяется четыре участка для хранения четырех массивов чисел типа **int**, каждый из которых состоит из трех элементов. Такое выделение памяти показано на схеме.

|  |
| --- |
| **Arr** |

|  |  |  |  |
| --- | --- | --- | --- |
| **arr[0]** | **arr[0][0]** | **arr[0][1]** | **arr[0][2]** |
| **arr[1]** | **arr[1][0]** | **arr[1][1]** | **arr[1][2]** |
| **arr[2]** | **arr[2][0]** | **arr[2][1]** | **arr[2][2]** |
| **arr[3]** | **arr[3][0]** | **arr[3][1]** | **arr[3][2]** |

Распределение памяти для двумерного массива.

Таким образом, объявление **arr2[4][3]** порождает в программе три разных объекта: указатель с идентификатором **arr**, безымянный массив из четырех указателей и безымянный массив из двенадцати чисел типа **int**.

Для доступа к безымянным массивам используются адресные выражения с указателем **arr**. Доступ к элементам массива указателей осуществляется с указанием одного индексного выражения в форме **arr2[2]** или **\*(arr2+2).** Для доступа к элементам двумерного массива чисел типа **int** должны быть использованы два индексных выражения в форме **arr2[1][2]** или эквивалентных ей **\*(\*(arr2+1)+2)** и **(\*(arr2+1))[2]**. Следует учитывать, что с точки зрения синтаксиса языка С++ указатель **arr** и указатели **arr[0],arr[1], arr[2], arr[3]** являются константами и их значения нельзя изменять во время выполнения программы.

Наличие размерностей индексов в объявлении параметра дает возможность компилятору сообщить функции о том, как расположены элементы в массиве. В двумерном массиве каждая строка по существу является одномерным массивом. Чтобы определить местоположение элемента в некоторой строке, функция должна точно знать, сколько элементов находится в каждой строке, чтобы она смогла пропустить соответствующее количество ячеек памяти при обращении к массиву. Таким образом, при обращении к   
**a [ l ] [2]** функция знает, что для доступа ко второй строке (строка 1) нужно пропустить в памяти три элемента первой строки, а затем обратиться к третьему элементу этой строки (элементу 2).

**// Пример использования двумерного массива**

**# include<iostream . h>**

**# include < iomanip . h >**

**const intstudents = 3; // количествостудентов**

**const intexams = 4; // количествоэкзаменов**

**int minimum(int [][exams], int, int);**

**int maximum(int [][exams], int, int);**

**float average(int [], int);**

**void printArray(int [][exams], int, int);**

**int main()**

**{**

**int studentGrades [students][exams] = {{77,68,86,73}, {96,87,89,78}, {70,90,86,81}};**

**cout « " Массив :" « endl;**

**printArray(studentGrades, students, exams);**

**cout « endl « endl « " Наихудшая оценка : "**

**« minimum(studentGrades, students, exams) « endl « " Наилучшая оценка : "**

**« maximum(studentGrades, students, exams) « endl;**

**for (int person = 0; person < students; person++)**

**cout « " Средняя оценка студента " « person « " равна "«**

**setiosflags(ios::fixedios::showpoint«**

**setprecision(2)average(studentGrades[person], exams) « endl;**

**return 0; }**

**// Поиск минимальной оценки**

**int minimum(int grades[][exams], int pupils, int tests)**

**{**

**int lowGrade = 100;**

**for (int i = 0; i < pupils; i++)**

**for (int j = 0; j < tests; j++)**

**if (grades[i][j] < lowGrade) lowGrade = grades[il [j];**

**return lowGrade;**

**}**

**// Поиск максимальной оценки**

**int maximum(int grades [][exams], int pupils, int tests)**

**{**

**int highGrade = 0;**

**for (int i = 0; i < pupils; i++)**

**for (int j = 0; j < tests; j++)**

**if (grades [i][j] > highGrade) highGrade = grades [i][j];**

**return highGrade ;**

**}**

**// Определение средней оценки для отдельного студента**

**float average ( int setOfGrades [], int tests )**

**{**

**int total = 0;**

**for (int i = 0; i < tests; i++)**

**total += setOfGrades[i];**

**return (float) total / tests;**

**}**

**// Печать массива**

**void printArray(int grades [][exams], int pupils, int tests)**

**{**

**for (int i = 0; i < pupils; i++)**

**{**

**cout « endl « " Оценки студента”;**

**for (int j = 0; j <tests; j++)**

**cout « setiosflags(ios::left) « grades [i] [j];**

**}**

**}**

**Массив :**

**Оценки студента [0]**

**Оценки студента [1]**

**Оценки студента [2]**

**[0] 77 96 70**

**[1] 68 87 90**

**[2] 86 89 86**

**[3] 73 78 81**

**Наихудшая оценка : 68**

**Наилучшая оценка: 96 Средняя оценка студента 0 равна 76.00**

**Средняя оценка студента 1 равна 87.50 а\* Средняя оценка студента 2 равна 81.75**

Внешняя структура **for** начинается с установки **i** (т.е. индекса строки) равным нулю, чтобы элементы первой строки можно было сравнивать с переменной **lowGrade**в теле внутренней структуры **for** .

Внутренняя структура **for** циклически обрабатывает четыре оценки каждой строки и сравнивает каждую оценку с **lowGrade .** Если оценка меньше, чем **lowGrade , lowGrade** устанавливается равной этой оценке. Затем внешняя структура **for** увеличивает индекс строки до значения **1.**

Элементы второй строки сравниваются с переменной **lowGrade .** Затем внешняя структура **for** увеличивает индекс строки до значения 2. Элементы третьей строки сравниваются с переменной **low Grade .** Когда выполнение вложенной структуры заканчивается, **lowGrade** содержит наименьшую оценку в двумерном массиве. Функция **maximum** работает аналогично функции **minimum .**

Функция **average** принимает два аргумента — одномерный массив результатов экзаменов для одного студента и количество результатов экзаменов в массиве. При вызове **average** первый аргумент — это **studentGrades [ stu**dent ], который указывает, что в **average** передается отдельная строка двумерного массива **studentGrades** .

Например, аргумент **studentGrades [ l ]** представляет собой четыре значения (одномерный массив оценок), хранимый во второй строке двумерного массива **studentGrades .**

Двумерный массив можно рассматривать как массив с элементами, представляющими собой одномерные массивы. Функция **average** подсчитывает сумму элементов массива, делит ее на количество результатов экзаменов и возвращает результат в форме с плавающей запятой.

Ввод, вывод и обработка двумерных массивов, как правило, основаны на ис­пользовании вложенных циклов, которые обеспечивают перебор всех элементов массива. В некоторых задачах может иметь значение порядок перебора элементов массива: “вдоль строки” или “вдоль столбца”. Если внешний цикл будет организован по первому индексу (по строкам), а внутренний цикл по второму индексу (по столб­цам), то выполняется построчный перебор элементов двумерного массива. Если внешний цикл в качестве параметра использует второй индекс, а внутренний цикл - первый индекс, то эле­менты массива перебираются по столбцам.

Рассмотрим пример, где указатели и двумерные массивы используются в качестве параметров функций.

**Пример 2.5.2.7-1 .** Составить программу с функцией, которая возвращает в качестве результата минимальный элемент матрицы d размером (m:n).

include <iostream.h>

float min(int m, int n, float \*p[])

void main()

{  
 float d[3][4]={ 1, 2,-2, 4,

5, 0,-3,18,

-9, 6, 7, 9};

float \*r[]=

{

(\*float) &d[0], (\*float) &d[1], (\*float) &d[2]

};

int m=3, n=4;

cout<<”\n Минимальный элемент матрицы равен ”<<min(m,n,r);

}

float min(int m, int n, float \*p[])

{

float x=p[0][0];

for(int i=0;i<m;i++)

for(int j=0;j<n;j++)

if(x>p[i][j]) x=p[i][j];

return x;

}

В результате работы программы на экран будет выведено сообщение:

**Минимальный элемент матрицы равен -9**

Функция min() в качестве параметров использует int m - число строк, int n - число столбцов, float \*p[] - массив указателей на одномерные массивы указателей типа float. В функции main() объявлена инициализированная матрица d[3][4], имеющая фиксированные размеры. Такой размер нельзя использовать непосредственно в качестве аргумента функции, поэтому объявлен дополнительный вспомогательный массив указателей \*r[]. В качестве значений элементов этого массива присваиваются адреса первых элементов строк матрицы, т.е. (\*float) &d[0], (\*float) &d[1], (\*float) &d[2], преобразованные к типу float.

Часто при решении инженерных задач размеры массива заранее не известны, они определяются в ходе решения поставленной задачи. Эти массивы называются динамическими. Организовать динамические массивы можно с помощью операции **new**, которая позволяет выделить в динамической области памяти участок для размещения массива соответствующего типа.

**Пример 2.5.2.7-2 .** Составить программу с функцией, которая заполняет элементы матрицы размером (m\*n) последовательно значениями целых чисел: 0,1,2,3,…. Для формирования матрицы в основной программе использованы динамические массивы, т.к. размеры матрицы заранее не известны.

include <iostream.h>

void fun(int m, int n, int \*\*uc)

{

int k=0;

for(int i=0;i<m;i++)

for(int j=0;j<n;j++)

uc[i][j]=k++;

}

void main()

{

int \*\*pi;

int m1,n1;

int i,j;

cout<<”\n Введите число строк матрицы : ”;

cin>>m1;

cout<<”\n Введите число столбцов матрицы : ”;

cin>>n1;

pi=new int[m1];

for( i=0;i<m1;i++)

pi[i]=new int[n1];

fun(m1,n1,pi);

for( i=0;i<m1;i++)

{

cout<<”\n Строка ”<<i+1<<”:”;

for( j=0;j<n1;j++)

cout<<”\t”<<p[i][j];

}

for( i=0;i<m1;i++)

delete pi[i];

delete pi;

}

Если при вводе задать размерность матрицы 3:4, то на экране появится:

**Строка 1 : 0 1 2 3**

**Строка 2 : 4 5 6 7**

**Строка 3 : 8 9 10 11**

Память выделяется с помощью оператора **new** и освобождается в конце программы - **delete**.

### 2.5.2.8. Строковые данные и операции над строками

Константой типа массив символов char[] является строковая константа. Эта константа записывается в виде последовательности символов заключенной в кавычки. Каждый символ в строковой константе может быть представлен либо в виде символа, либо в виде специальной последовательности символов.

Следует отметить, что в строковых констатах всегда обязательно задавать все три цифры в управляющей последовательности.

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

Например: "ABCDE\0773GH" данная строка будет воспринята как слово ABCDE?3GH, поскольку \077 это символ ?. В случае если указать неполную управляющую строку "ABCDE\773GH", то строка будет содержать ABCDE;GH, так как компилятор воспримет последовательность \773 как символ ; .

Каждая строковая константа автоматически дополняется символом \0, помещаемым в конце строки. Например константа "Язык Си" состоит из 8 следующих символов Я,з,ы,к,пробел,С,и,\0.

Несмотря на то, что строковые константы это массивы символов, значение строковой константы это не массив символов, образующих строку, а адрес памяти, начиная с которого расположены символы, образующие строку, т.е. тип значения строковой константы это указатель на тип char.

**char string[20];**

string[0] = 'П'; string[1] = ' р ';

string[2] = ' и '; string[3] = ' в ';

string[4] = ' е '; string[5] = ' т ';

string[6] = '\0';

printf("%s\n", string);

%s - формат для печати СТРОК.

Никакие другие массивы не могут быть напечатаны целиком одним оператором.

char string[20];

string[0] = ' П '; string[1] = ' р ';

string[2] = ' и '; string[3] = ' в ';

string[4] = ' е '; string[5] = ' т ';

string[6] = '\n'; /\* Перевод строки - тоже буква \*/

string[7] = '\0';   printf("%s", string);  // или даже просто   printf(string);

Такие массивы можно записать в виде строки букв в ""   char string[20] = "Привет\n";   Оставшиеся неиспользованными символы массива от string[8] до string[19] содержат МУСОР.

**ПОЧЕМУ ДЛЯ СТРОК ИЗОБРЕЛИ СИМВОЛ "ПРИЗНАК КОНЦА"?**

Строка - это ЧАСТЬ массива букв. В разное время число букв в строке может быть различным, лишь бы не превышало размер массива (тогда случится сбой программы). Значит, следует где-то хранить текущую длину строки (число использованных символов).

Есть три решения:

(1) В отдельной переменной. Ее следует передавать во все функции обработки данной строки (причем она может изменяться).

char str[32]; /\* массив для строки \*/ int slen; /\* брать первые slen букв в этом массиве \*/ ... func(str, &slen); /\* ДВА аргумента для передачи ОДНОЙ строки \*/ ...

Этот подход работоспособен, но строка разбивается на два объекта: сам массив и переменную для его длины. Неудобно.

(2) Хранить текущую длину в элементе str[0], а буквы - в str[1] ... и т.д. Плохо тем, что в str[0] можно хранить лишь числа от 0 до 255, и если строка длиннее - то такой подход неприменим.

(3) Не хранить длину НИГДЕ, а ввести символ-признак конца строки. Теперь в   func(str); /\* ОДИН аргумент - сам массив \*/   передается только сам массив, а его текущая длина может быть при нужде вычислена при помощи некоей функции, вроде такой:

/\* функция от массива букв \*/

int strlen(char s[])

{

int counter = 0; /\* счетчик и одновременно индекс \*/

while (s[counter] != '\0') /\* пока не встретился признак конца текста \*/

counter++; /\* посчитать символ \*/

return counter; /\* сколько символов, отличных от '\0' \*/

}

Тут никаких ограничений нет. Именно этот подход и был избран в языке Си, хотя в принципе можно самому пользоваться и другими. На самом деле в языке есть такая СТАНДАРТНАЯ функция strlen(s) (вам не надо писать ее самому, ее уже написали за вас).

Строчная константа, как, например,   "I AM A STRING"   является массивом символов. Компилятор завершает внутреннее представление такого массива символом \0, так что программы могут находить его конец. Таким образом, длина массива в памяти оказывается на единицу больше числа символов между двойными кавычками.

По-видимому, чаще всего строчные константы появляются в качестве аргументов функций, как, например, в   printf ("HELLO, WORLD\N");   когда символьная строка, подобная этой, появляется в программе, то доступ к ней осуществляется с помощью указателя символов;

функция printfфактически получает указатель символьного массива. Конечно, символьные массивы не обязаны быть только аргументами функций. Если описать MESSAGE как char \*MESSAGE;   то в результате оператора   MESSAGE = "NOW IS THE TIME";   переменная MESSAGE станет указателем на фактический массив символов. Это не копирование строки; здесь участвуют только указатели. В языке "C" не предусмотрены какие-либо операции для обработки всей строки символов как целого.

Первая функция - это strcpy(S,T), которая копирует строку T в строку S. Аргументы написаны именно в этом порядке по аналогии с операцией присваивания, когда для того, чтобы присвоить T к S обычно пишут   S = T,   сначала приведем версию с массивами:

strcpy(S, T) /\* копирование строки как массива T в S \*/

char S[], T[];

{

int I; I = 0;

while ((S[I] = T[I]) != '\0')

I++;

}

Для сопоставления ниже дается вариант strcpy с указателями.

strcpy(S, T) /\* копирование строки T в S через указатели 1 способ \*/

char \*S, \*T;

{

while ((\*S = \*T) != '\0')

{ S++; T++; }

}

  Так как аргументы передаются по значению, функция strcpy может использовать S и T так, как она пожелает. Здесь они с удобством полагаются указателями, которые передвигаются вдоль массивов, по одному символу за шаг, пока не будет скопирован в S завершающий в T символ \0.

На практике функция strcpy была бы записана не так, как мы показали выше. Вотвтораявозможность:

strcpy (S, T) /\* копирование строки T в S через указатели 2 способ \*/

char \*S, \*T;

{ while ((\*S++ = \*T++) != '\0') ; }

Здесь увеличение S и T внесено в проверочную часть. Значением \*T++ является символ, на который указывал T до увеличения; постфиксная операция ++ не изменяет T, пока этот символ не будет извлечен. Точно так же этот символ помещается в старую позицию S, до того как S будет увеличено. Конечный результат заключается в том, что все символы, включая завершающий \0, копируются из T в S. И как последнее сокращение мы опять отметим, что сравнение с \0 является излишним, так что функцию можно записать в виде

strcpy (S, T) /\*копирование строки T в S через указатели 3 способ \*/

char \*S, \*T;

{ while (\*S++ = \*T++) ; }

хотя с первого взгляда эта запись может показаться загадочной, она дает значительное удобство. Этой идиомой следует овладеть уже хотя бы потому, что вы с ней будете часто встречаться в "C"-программах.

Вторая функция - strcmp(S, T), которая сравнивает символьные строки S и T, возвращая отрицательное, нулевое или положительное значение в соответствии с тем, меньше, равно или больше лексикографически S, чем T. Возвращаемое значение получается в результате вычитания символов из первой позиции, в которой S и T не совпадают.

strcmp (S, T)

char S[], T[];

{

int I;   I = 0;

while (S[I] == T[I])

if (S[I++] == '\0') return (0);

return (S[I]-T[I]);

}

 Вот версия strcmp с указателями:

**strcmp(S, T)**

char \*S, \*T;

{

for ( ; \*S == \*T; S++, T++)

if (\*S == '\0') return(0);

return (\*S-\*T);

}

так как ++ и -- могут быть как постфиксными, так и префиксными операциями, встречаются другие комбинации \* и ++ и --, хотя и менее часто. Например, \*++P увеличивает Pдо извлечения символа, на который указывает P, а \*--P   сначала уменьшает P.

Массив символов имеет несколько специфических особенностей. Массиву символов можно задать начальные значения, используя литеральную константу. Например, объявление

char stringl [ ] = " first ";

присваивает элементам массива stringl в качестве начальных значений отдельные символы строки " first ". Размер массива stringl в предыдущем объявлении, определяется компилятором на основе длины строки. Важно заметить, что строка " first " содержит пять символов плюс специальный символ окончания строки, называемый *нулевым символом*. Таким образом, массив stringl на самом деле содержит шесть элементов. Нулевой символ представляется символьной константой '\0'. Все строки заканчиваются этим символом. Сим­вольный массив, представляющий строку, должен всегда объявляться доста­точно большим для того, чтобы в него можно было поместить количество символов в строке и завершающий нулевой символ.

Символьному массиву можно также задать в качестве начального значения список отдельных символьных констант, указанных в списке инициализации.

Например, предыдущее объявление эквивалентно следующему

char stringl [ ] = {' f ', ' i ', ' r ', ' s ', ' t ', '\0'}

Поскольку строка является массивом символов, мы можем получить доступ к отдельным символам строки, используя индексную запись массива.

Например, stringl [0] — это символ ' f ', а stringl [3] — это символ ' s '.

|  |  |
| --- | --- |
| Прототип функции | Действие |
| **char\*strcpy(char\*S1,const char\*s2)** | копирует строку s2 в строку s1.  Возвращает значение s1 |
| **char\*strncpy(char\*S1,const char\*s2,size t n)** | копирует не более n символов из строки s2 в строку s1.  Возвращает значение s1 |
| **char\*strcat(char\*S1,const char\*s2)** | Добавляет строку S2 строке S1 Первый символ строки s2 записывается поверх завершающего нулевого символа строки S1 . Возвращает значение s1 |
| **char\*strncat(char\*S1,const char\*s2,size t n)** | Добавляет не более n символов строки S2 в строку S1 Первый символ строки s2 записывается поверх завершающего нулевого символа строки S1 . Возвращает значение s1 |
| **int strcmp(const char\*S1,const char\*s2)** | Сравнивает строки s1 и s2. Функция возвращает значение 0, меньшее, чем 0 или большее, чем 0, если s1 соответственно равна, меньше или больше, чем s2. |
| **int strncmp(const char\*S1,const char\*s2,size t n)** | Сравнивает до n символов строки s1 со строкой s2. Функция возвращает значение 0, меньшее, чем 0 или большее, чем 0, если s1 соответственно равна, меньше или больше, чем s2. |
| **size t strlen(const char \*s)** | Определяет длину строки s. Возвращает количество символов, предшествующих завершающему нулевому символу. |

**Пример 2.5.2.8-1. Функция** char\* small (char\* str)**получает строку** str**и «сжимает» строку, удалив из нее все, кроме одного, одинаковые символы, следующие подряд. Если таких символов в строке не найдено, то функция возвращает** NULL**, в противном случае адрес «сжатой» строки** str.

Программа со стандартного устройства ввода считывает фразу, состоящую из **N** слов (точное количество слов во фразе задает пользователь заранее), и выдает на экран преобразованную фразу, в которой слова «сжаты» с помощью функции **small.** Программа подсчитывает количество непреобразованных слов в исходной фразе.

# include<string . h>

char\* delIN(char\*pstr,int k);

char\* smallIN (char\*pstr)

{

char\*st;

int sch=0;

for (int i=0;pstr[i]!='\0';i++)

{

if(pstr[i-1]==pstr[i])

{

sch=1;

st=delIN(pstr+i,1);

i--;

}

}

return sch? pstr: NULL;

}

#include <string.h>

char\* delADR(char\*pstr,int k);

char\* smallADR (char\*pstr)

{

char\*st;

int sch=0;

for(char\*str=pstr;\*(str+1)!='\0';str++)

{ if(\*str==\*(str+1))

{

sch=1;

st=delADR(str,1);

str--;

}

}

return sch? pstr: NULL;

}

#include <string.h>

char\*delIN(char\*pstr,int k)

{ for(int iE=0,i=k;pstr[i-1]!='\0';iE++,i++)

pstr[iE]=pstr[i];

return pstr;

}

#include <string.h>

char\* delADR (char\*pstr,int k)

{ for(char\* iE=pstr,\*i=pstr+k;\*(i-1)!='\0';iE++,i++)

\*iE=\*i;

return pstr;

}

#include <string.h>

#include <iostream.h>

char\* smallADR (char\*pstr);

char\* smallIN (char\*pstr);

void testirovanie ()

{ char str1[30]="fddgyft";

char str2[30]="fdtrev";

int op;

cout<<"Viberite rejim raboti: "<<endl;

cout<<"1.Rabota s adresom-naberite 1 "<<endl

<<"2.Rabota s indeksom-naberite 2 "<<endl;

cin>>op;

char\* str3,\*str4;

switch (op)

{ case 1 : str3=smallADR(str1);

str4=smallADR(str2);

break;

case 2 : str3=smallIN(str1);

str4=smallIN(str2);

}

cout<<"Rezultiruyushaya stroka1 "<<str3<<endl

<<"Rezultiruyushaya stroka 2 \*"<<str4<<"\*";

}

#include <conio.h>

#include <iostream.h>

void testirovanie();

void reshenie();

void main (void)

{ int op;

do{

clrscr();

cout<<"Viberite rejim raboti: "<<endl;

cout<<"1.Rejim testirovaniya- 1 "<<endl

<<"2.Rejim resheniya konkretnoy zadachi- 2 "<<endl

<<"3.Vihod- 3 "<<endl;

cin>>op;

switch(op)

{ case 1: testirovanie(); break;

case 2: reshenie();

}

}while (op<3);

getch();

}

Применение указателей и их массивов позволяет весьма рационально решать задачи сортировки сложных объектов с неодинаковыми размерами. Например, для упорядочения (хотя бы по алфавиту) списка строк можно менять местами не сами строки, а переставлять значения элементов массива указателей на эти строки. Накладными расходами при этой "косвенной" сортировке списков объектов является требование к памяти, необходимой для массива указателей. Выигрыш- существенное ускорение сортировки.

Выделение памяти под массив строк является динамическим.

# **2.5.3. Контрольные вопросы по теме «Основные конструкции языка программирования QB»**

# (вставить из лаб.работ!)**2.5.4. Тестовые задания по теме «Основные конструкции языка программирования QB»**

1. **Внутреннее представление данных в памяти компьютера определяется**
2. типом данных
3. системой счисления
4. системой программирования
5. типом транслятора
6. **Последовательность символов, заключенная в кавычки, в языке QB называется**
7. строковой константой
8. строковой переменной
9. числовой переменной
10. числовой константой
11. **Величина, не изменяющая своего значения в ходе выполнения программы, называется**
12. константой
13. функцией
14. переменной
15. оператором
16. **Квадратный корень числа вычисляет функция**
17. **SQR (x)**
18. **ATN (x)**
19. **ABS (x)**
20. **INT (x)**
21. **Ошибочной записью числовой переменной в языке QB является**
22. **3A**
23. **A4**
24. **T45**
25. **SS**
26. **Правильная запись числовой переменной в языке QB выглядит так**
27. **SK2**
28. **3KS**
29. **K!1**
30. нет верного ответа
31. **Все величины, представленные в памяти компьютера, называются**
32. данными
33. информацией
34. алфавитом
35. символами
36. **Величина, к которой обращаются по имени, принимающая различные значения в ходе выполнения программы, называется**
37. переменной
38. строкой
39. символом
40. константой
41. Знак **<>** в языке **QB** означает
42. «не равно»
43. «присвоить»
44. «тождественно»
45. нет верного ответа
46. **Значение модуля числа вычисляет функция**
    1. **ABS (x)**
    2. **SQR (x)**
    3. **ATN (x)**
    4. **INT (x)**
47. **Показательная функция записывается как**
48. **EXP (x)**
49. **ABS (x)**
50. **SQR (x)**
51. **ATN (x)**
52. **Ошибочной записью переменной является**
53. **Ф1**
54. **SK2**
55. **K12$**
56. **T$**
57. **Правильная запись символьной переменной в языке QB**
58. **SK$**
59. **SK2**
60. **K$1**
61. **$K1**
62. **Употребление в выражении более одной пары скобок**
63. допускается
64. не допускается
65. допускается только при сложении и вычитании
66. допускается только при умножении и делении
67. **Из перечисленных действий наивысший приоритет имеет**
68. возведение в степень
69. сложение
70. вычитание
71. умножение
72. **В арифметические выражения могут входить**
73. переменные
74. машинные коды
75. команда MS-DOS
76. нет верного ответа
77. **К названию типа переменных относится**
78. целые
79. числовые
80. указатели
81. все ответы верные
82. **В название типа данных не входит понятие**
83. переменные
84. числовые
85. символьные
86. нет верного ответа
87. **Стандартные функции используются для**
88. вычисления часто употребляемых выражений
89. вычисления выражений с заданной точностью
90. вычисления логических выражений
91. нет верного ответа
92. **В качестве аргумента стандартной функции можно применять**
93. любое арифметическое выражение
94. число
95. логическую операцию
96. все ответы верные
97. **Стандартная функция SQR вычисляет**
98. квадратный корень
99. остаток от деления
100. модуль числа
101. определяет знак числа
102. **Стандартная функция MOD вычисляет**
103. остаток от деления
104. модуль числа
105. квадратный корень числа
106. определяет знак числа
107. Функция **SGN** вычисляет
108. определяет знак числа
109. модуль числа
110. остаток от деления
111. квадратный корень числа
112. **Числовые переменные могут быть**
113. целыми и вещественными
114. целыми и символьными
115. простыми и сложными
116. нет верного ответа
117. **Аргумент для функции SIN задается**
118. в радианах
119. в градусах
120. в секундах
121. в любых единицах
122. **Функция SGN вычисляет**
123. знак числа\*
124. модуль числа
125. остаток от деления
126. квадратный корень числа
127. **Укажите правильный порядок выполнения операций в арифметическом выражении**
128. возведение в степень, изменение знака числа, деление, целочисленное деление, вычитание
129. изменение знака числа, возведение в степень, деление, целочисленное деление, вычитание
130. возведение в степень, изменение знака числа, целочисленное деление, деление, вычитание
131. любой порядок
132. **Оператором вывода в языке QB является оператор**
133. **PRINT**
134. **INPUT**
135. **REM**
136. **RUN**
137. **Оператором ввода в языке QB является оператор**
138. INPUT
139. PRINT
140. REM
141. RUN
142. **Оператор – это**
143. конструкция, описывающая действия, выполняемые в процессе работы программы
144. величина, над которой выполняется операция
145. числовое данное
146. структура, определяющая этапы программы
147. **Действие оператора PRINT – это**
148. переход на новую строку
149. вывод через пробел
150. ошибка в записи
151. ничего не сделает
152. **Оператор PRINT A предназначен**
153. для вывода на экран
154. для ввода данных с клавиатуры
155. для вывода числовых данных
156. для печати результатов на принтере
157. **Оператор IF – это**
158. условный оператор
159. оператор цикла
160. безусловный оператор
161. оператор выбора
162. **Оператор SELECT CASE– это**
163. оператор выбора
164. условный оператор
165. безусловный оператор
166. оператор цикла
167. **Оператор IF сначала**
168. проверяет свое условное выражение
169. выполняет операторы из тела программы, идущие после него
170. подключает процедуры
171. передает управление на конец программы
172. **Оператор IF THEN –это оператор**
173. усеченного разветвления
174. стандартного разветвления
175. вложенного разветвления
176. блочного разветвления
177. **Оператор IF THEN ELSE – это оператор**
178. стандартного разветвления
179. усеченного разветвления
180. блочного разветвления
181. вложенного разветвления
182. **Оператор IF предназначен**
183. для выбора по условию
184. для выполнения линейного алгоритма
185. для перехода на нужную строку
186. для программирования циклов
187. **В операторе условного перехода между словами IF… THEN может быть**
188. арифметическое или логическое выражение
189. логическое выражение
190. оператор присваивания
191. любой оператор
192. **Необязательные ключевые слова в однострочном операторе условного перехода**
193. ELSE
194. THEN, ELSE
195. ELSEIF
196. END IF
197. **Необязательные ключевые слова в блочном операторе условного перехода**
     1. ELSE, ELSEIF
     2. ELSEIF
     3. ELSE, ELSEIF, END IF
     4. нет верного ответа
198. **В блочном операторе IFдолжны быть единственными в своих строках ключевые слова**
199. ELSE, END IF
200. ELSE, ELSEIF, END IF
201. THEN, ELSE, ELSEIF, END IF
202. нет верного ответа
203. **В операторе условного перехода IF обязательно должно присутствовать слово**
204. THEN
205. THAN
206. ELSE
207. THEN, ELSE
208. **Ошибочно записан оператор**
209. все записаны верно
210. IF A>B THEN C=1 ... D=5
211. IF A+B THEN C=1 ... D=5
212. IF A>B THEN C=1 ELSE D=5
213. **Ошибочно записан оператор**

**1)**SELECTCASEA>B

CASE 7

X=10

## END SELECT

**2)** SELECT CASE A

CASE 7

X=10

END SELECT

**3)** SELECT CASE B

CASE A+2

X=7

CASE IS > 3

X=5

ENDSELECT

**4)**все операторы тзаписаны верно

1. **В операторе условного перехода IF после слова THEN может быть**
2. любой оператор QB
3. оператор присваивания
4. арифметическое или логическое выражение
5. нет верного ответа
6. **В операторе условного перехода IF обязательно должно присутствовать слово**
7. THEN
8. THAN
9. ELSE
10. THEN, ELSE
11. **Ключевое слово RETURN**
12. не относится к оператору условного перехода
13. обязательно должно присутствовать в операторе условного перехода
14. необязательное для оператора условного перехода
15. **Ключевое слово WEND**
16. не относится к оператору условного перехода
17. обязательно должно присутствовать в операторе условного перехода
18. необязательное для оператора условного перехода
19. **Оператор IF – это**
20. условный оператор
21. оператор цикла
22. безусловный оператор
23. оператор выбора
24. **ОператорSELECT CASE– это**
25. оператор выбора
26. условный оператор
27. безусловный оператор
28. оператор цикла
29. **Оператор IF сначала**
30. проверяет свое условное выражение
31. выполняет операторы из тела программы, идущие после него
32. подключает процедуры
33. передает управление на конец программы
34. **Оператор IF THEN – это оператор**
35. усеченного разветвления
36. стандартного разветвления
37. вложенного разветвления
38. блочного разветвления
39. **Оператор IF THEN ELSE– это оператор**
40. стандартного разветвления
41. усеченного разветвления
42. блочного разветвления
43. вложенного разветвления
44. **Оператор FOR – это**
45. оператор регулярного цикла
46. оператор выбора
47. оператор итеративного цикла
48. составной оператор
49. **Переменная, изменяющая свое значение при каждом новом вхождении в цикл, называется**
50. параметром цикла
51. шагом цикла
52. индексом
53. размером
54. **Многократно повторяющаяся часть алгоритма называется**
55. телом цикла
56. выбором
57. перебором
58. шагом цикла
59. **Циклическая алгоритмическая структура может быть**
60. регулярная, итеративная
61. регулярная, разветвляющаяся
62. итеративная, разветвляющаяся
63. нет верного ответа
64. **В регулярной циклической структуре число повторений операторов тела цикла**
65. заранее известно или может быть предварительно вычислено
66. может быть известно или неизвестно заранее
67. заранее неизвестно
68. нет верного ответа
69. **Телом цикла в операторе FOR … NEXT могут быть**
70. любые операторы\*
71. только оператор условного перехода или оператор присваивания
72. только арифметические или логические выражения
73. нет верного ответа
74. **Тело цикла в операторе FOR … NEXT**
75. может ни разу не выполниться\*
76. обязательно выполнится хотя бы 1 раз
77. выполняется несколько раз
78. нет верного ответа
79. **Для досрочного прекращения регулярного цикла используется оператор**
80. EXIT FOR
81. EXITFOR
82. EXIT
83. BREAK
84. **C использованием регулярных циклических структур можно вычислять**
85. значения функции в заданном диапазоне, произведения и суммы с конечным числом слагаемых и сомножителей
86. произведения и суммы конечных или бесконечных рядов
87. значения суммы бесконечных рядов
88. все вышеперечисленное верно
89. **Для того, чтобы операторы тела цикла выполнились необходимое число раз, параметр цикла**
90. должен быть переменной целого типа
91. должен быть переменной вещественного типа
92. должен быть переменной строкового типа
93. должен быть переменной целого или вещественного типа
94. **Ключевое слово STEP**
95. необязательное для оператора регулярного цикла FOR\*
96. обязательно должно присутствовать в операторе регулярного цикла FOR
97. не относится к оператору регулярного цикла FOR
98. **Ключевое слово NEXT**
99. обязательно должно присутствовать в операторе регулярного цикла FOR
100. необязательное для оператора регулярного цикла FOR
101. не относится к оператору регулярного цикла FOR
102. **Ключевое слово DO**
103. не относится к оператору регулярного цикла FOR
104. обязательно должно присутствовать в операторе регулярного цикла FOR
105. необязательное для оператора регулярного цикла FOR
106. **Ключевое слово ELSE**
107. не относится к оператору регулярного цикла FOR
108. обязательно должно присутствовать в операторе регулярного цикла FOR
109. необязательное для оператора регулярного цикла FOR
110. **Ключевое слово LOOP**
111. не относится к оператору регулярного цикла FOR
112. обязательно должно присутствовать в операторе регулярного цикла FOR
113. необязательное для оператора регулярного цикла FOR
114. **Ключевое слово IF**
115. не относится к оператору регулярного цикла FOR
116. обязательно должно присутствовать в операторе регулярного цикла FOR
117. необязательное для оператора регулярного цикла FOR
118. **Ключевое слово WHILE**
119. не относится к оператору регулярного цикла FOR
120. обязательно должно присутствовать в операторе регулярного цикла FOR
121. необязательное для оператора регулярного цикла FOR
122. **Ключевое слово UNTIL**
123. не относится к оператору регулярного цикла FOR
124. обязательно должно присутствовать в операторе регулярного цикла FOR
125. необязательное для оператора регулярного цикла FOR
126. **Для программирования циклических алгоритмических структур с известным числом повторений используется оператор, в котором обязательно присутствуют слова**
127. FOR , NEXT
128. FOR , NEXT , STEP, IF
129. FOR , NEXT , EXIT FOR , DO
130. Все ответы верны
131. нет верного ответа
132. **Для программирования циклических алгоритмических структур с известным числом повторений используется оператор**
133. FOR … NEXT
134. DO … UNTIL
135. SELECT CASE …END SELECT
136. Все ответы верны
137. **Если в операторе регулярного цикла FOR … NEXT слово STEP отсутствует, это означает, что после каждого выполнения тела цикла**
138. параметр цикла увеличивается на 1
139. параметр цикла уменьшается на 1
140. параметр цикла изменяется произвольно
141. параметр цикла не изменяется
142. будет сообщение об ошибке
143. **Оператор DO…LOOP – это**
144. оператор итеративного цикла
145. оператор выбора
146. оператор регулярного цикла
147. составной оператор
148. **В итеративной циклической структуре число повторений операторов тела цикла**
149. заранее неизвестно
150. может быть известно заранее
151. заранее известно или может быть предварительно вычислено
152. нет верного ответа
153. **Телом цикла в операторе DO…LOOP могут быть**
154. любые операторы
155. только оператор условного перехода или оператор присваивания
156. только арифметические или логические выражения
157. нет верного ответа
158. **Тело цикла в операторе DO…LOOP**
159. может ни разу не выполниться или обязательно выполнится хотя бы 1 раз в зависимости от типа оператора
160. может ни разу не выполниться
161. обязательно выполнится хотя бы 1 раз
162. выполняется заданное число раз
163. **Для досрочного прекращения итеративного цикла используется оператор**
164. EXIT DO
165. EXIT LOOP
166. EXIT
167. BREAK
168. нет верного ответа
169. **Для программирования циклических алгоритмических структур с неизвестным числом повторений используется оператор**
170. DO … LOOP
171. FOR … NEXT
172. SELECT CASE …END SELECT
173. все ответы верны
174. **Алгоритмическая структура цикла итеративного типа может быть**
175. с предусловием или с постусловием
176. только с предусловием
177. только с постусловием
178. безусловная
179. **Ключевое слово STEP**
180. не относится к оператору итеративного цикла DO … LOOP
181. необязательное для оператора итеративного цикла DO … LOOP
182. обязательно должно присутствовать в операторе цикла DO … LOOP
183. **Ключевое слово NEXT**

**1)** не относится к оператору итеративного цикла DO … LOOP\*

1. необязательное для оператора итеративного цикла DO … LOOP
2. обязательно должно присутствовать в операторе цикла DO … LOOP
3. **Ключевое слово DO**
4. обязательно должно присутствовать в операторе итеративного цикла
5. не относится к оператору итеративного цикла
6. необязательное для оператора итеративного цикла с постусловием
7. **Ключевое слово ELSE**
8. не относится к оператору итеративного цикла
9. обязательно должно присутствовать в операторе итеративного цикла
10. необязательное для оператора итеративного цикла
11. **Ключевое слово LOOP**

1) обязательно должно присутствовать в операторе итеративного цикла

2) не относится к оператору итеративного цикла

1. необязательное для оператора итеративного цикла с постусловием
2. **Ключевое слово IF**
3. не относится к оператору итеративного цикла DO … LOOP
4. обязательно должно присутствовать в операторе итеративного цикла

DO … LOOP

1. необязательное для оператора итеративного цикла DO … LOOP
2. **Ключевое слово WHILE**
3. может записываться либо после слова DO, либо после слова LOOP
4. должно быть записано только после слова DO
5. должно быть записано только после слова LOOP
6. не относится к оператору итеративного цикла DO … LOOP
7. **Ключевое слово UNTIL**
8. может записываться либо после слова DO, либо после слова LOOP\*
9. должно быть записано только после слова DO
10. должно быть записано только после слова LOOP
11. не относится к оператору итеративного цикла DO … LOOP
12. **Оператор DO WHILE…LOOP – это**
13. оператор итеративного цикла с предусловием
14. оператор выбора
15. оператор регулярного цикла
16. оператор итеративного цикла с постусловием
17. нет верного ответа
18. **Оператор DO UNTIL…LOOP – это**
19. оператор итеративного цикла с предусловием
20. оператор выбора
21. оператор регулярного цикла
22. оператор итеративного цикла с постусловием
23. нет верного ответа
24. **Оператор DO …LOOP WHILE – это**
25. оператор итеративного цикла с постусловием
26. оператор выбора
27. оператор регулярного цикла
28. оператор итеративного цикла с предусловием
29. нет верного ответа
30. **Оператор DO …LOOP UNTIL – это**
31. оператор итеративного цикла с постусловием
32. оператор выбора
33. оператор регулярного цикла
34. оператор итеративного цикла с предусловием
35. нет верного ответа
36. **Если при программировании циклической структуры используется оператор DO WHILE…LOOP, то тело цикла**
37. может ни разу не выполниться
38. обязательно выполнится хотя бы 1 раз
39. выполняется заданное число раз
40. оператор не относится к средствам программирования итеративного цикла
41. **Если при программировании циклической структуры используется оператор**

**DO UNTIL…LOOP , то тело цикла**

1. может ни разу не выполниться
2. обязательно выполнится хотя бы 1 раз
3. выполняется заданное число раз
4. оператор не относится к средствам программирования итеративного цикла
5. **Если при программировании циклической структуры используется оператор**

**DO…LOOP WHILE, то тело цикла**

1. обязательно выполнится хотя бы 1 раз
2. может ни разу не выполниться
3. выполняется заданное число раз
4. оператор не относится к средствам программирования итеративного цикла
5. **Если при программировании циклической структуры используется оператор**

**DO…LOOP UNTIL, то тело цикла**

1. обязательно выполнится хотя бы 1 раз
2. может ни разу не выполниться
3. выполняется заданное число раз
4. оператор не относится к средствам программирования итеративного цикла
5. **Если при программировании циклической структуры используется оператор**

**SELECTCASE, то тело цикла**

1. оператор не относится к средствам программирования итеративного цикла
2. обязательно выполнится хотя бы 1 раз
3. может ни разу не выполниться
4. выполняется заданное число раз
5. **Использование слова WHILE в операторе итеративного цикла DO…LOOP означает, что тело цикла будет выполняться до тех пор, пока значение выражения, записанного после WHILE**
6. истинно
7. ложно
8. слово не относится к оператору итеративного цикла DO…LOOP
9. **Использование слова UNTIL в операторе итеративного цикла DO…LOOP означает, что тело цикла будет выполняться до тех пор, пока значение выражения, записанного после UNTIL**
10. ложно
11. истинно
12. слово не относится к оператору итеративного цикла DO…LOOP
13. Для того, чтобы итеративный цикл продолжался пока некоторое условие остается истинным, это условие должно быть записано
14. после слова WHILE
15. перед словом WHILE
16. после слова UNTIL
17. перед словом UNTIL
18. нет верного ответа
19. **Для того, чтобы итеративный цикл продолжался пока некоторое условие остается ложным, это условие должно быть записано**
20. после слова UNTIL
21. после слова WHILE
22. перед словом WHILE
23. перед словом UNTIL
24. нет верного ответа
25. **C использованием итеративных циклических структур можно вычислять**
26. все утверждения верны
27. значения функции в заданном диапазоне, произведения и суммы с конечным числом слагаемых и сомножителей
28. произведения и суммы конечных или бесконечных рядов
29. значения членов конечных и бесконечных рядов
30. все утверждения неверны
31. **После ключевых слов WHILE или UNTIL в операторе итеративного цикла**

**DO…LOOP записывается**

1. арифметическое или логическое выражение
2. оператор выбора
3. любой оператор
4. нет верного ответа
5. **Использование слова REPEAT в операторе итеративного цикла DO…LOOP означает, что тело цикла будет выполняться до тех пор, пока значение выражения, записанного после REPEAT**
6. слово не относится к оператору итеративного цикла DO…LOOP\*
7. истинно
8. ложно
9. **Использование слова RETURN в операторе итеративного цикла DO…LOOP означает, что тело цикла будет выполняться до тех пор, пока значение выражения, записанного после RETURN**
10. слово не относится к оператору итеративного цикла DO…LOOP
11. истинно
12. ложно
13. **Если условие, записанное после слова WHILE в операторе итеративного цикла**

**DO…LOOP ложно, то**

1. цикл завершается
2. цикл продолжается
3. цикл продолжится еще один раз
4. слово не относится к оператору итеративного цикла DO…LOOP
5. **Если условие, записанное после слова UNTIL в операторе итеративного цикла**

**DO…LOOP ложно, то**

1. цикл продолжается
2. цикл завершается
3. цикл продолжится еще один раз
4. слово не относится к оператору итеративного цикла DO…LOOP
5. **Если условие, записанное после слова WHILE в операторе итеративного цикла**

**DO…LOOP истинно, то**

1. цикл продолжается
2. цикл завершается
3. цикл продолжится еще один раз
4. слово не относится к оператору итеративного цикла DO…LOOP
5. **Если условие, записанное после слова UNTIL в операторе итеративного цикла**

**DO…LOOP истинно, то**

1. цикл завершается
2. цикл продолжается
3. цикл продолжится еще один раз
4. слово не относится к оператору итеративного цикла DO…LOOP
5. **Если условие, записанное после слова RETURN в операторе итеративного циклаDO…LOOP истинно, то**
6. слово не относится к оператору итеративного цикла DO … LOOP
7. цикл завершается
8. цикл продолжается
9. цикл продолжится еще один раз
10. **Использование слова WEND в операторе итеративного цикла означает, что тело цикла будет выполняться до тех пор, пока значение выражения, записанного после него**
11. после этого слова нет выражения
12. истинно
13. ложно
14. слово не относится к оператору итеративного цикла
15. **Использование слова SELECT в операторе итеративного цикла означает, что тело цикла будет выполняться до тех пор, пока значение выражения, записанного после него**
16. слово не относится к оператору итеративного цикла
17. после этого слова нет выражения
18. истинно
19. ложно
20. **Для завершения итеративного цикла необходимо**
21. все ответы верны
22. чтобы условие, записанное после слова WHILE, было ложным
23. чтобы условие, записанное после слова UNTIL, было истинным
24. чтобы выполнился оператор EXIT DO
25. **Преобразование итеративного цикла с предусловием в цикл с постусловием**
26. всегда возможно
27. возможно иногда
28. невозможно
29. **Преобразование итеративного цикла с постусловием в цикл с предусловием**
30. всегда возможно
31. возможно иногда
32. невозможно
33. **Массив – это** 
    1. совокупность данных одного типа, объединенных общим именем
    2. совокупность данных одного типа
    3. набор индексированных данных
    4. набор разных данных
34. **Индексом массива может быть**
35. любое целочисленное выражение
36. выражение любого типа
37. только целочисленные переменные
38. переменные любого типа
39. **На размерность массива указывает**
40. количество индексов
41. значения индексов
42. суммарное количество индексов
43. сумма значений индексов
44. **В программе массив объявляется оператором**
45. DIM
46. REM
47. FOR
48. DO
49. **Максимальная размерность массива может быть равна**
50. 60
51. 10
52. 3
53. 5
54. **Оператор DIM**
55. резервирует область памяти для элементов массива
56. резервирует имя для элементов массива
57. выстраивает элементы массива в линейку
58. подсчитывает количество элементов массива
59. **Если одномерный массив не объявлен в программе оператором DIM, то его верхняя граница устанавливается равной**
60. 10
61. 5
62. 20
63. 3
64. **При объявлении массива служебное слово AS с параметром type устанавливает**
65. тип массива
66. размерность массива
67. размер массива
68. значения элементов массива
69. **Чтобы инициализировать элементы числового массива нулями, используется оператор**
70. DIM
71. FOR
72. IF
73. REM
74. **По распределению памяти массивы могут быть**
75. статические и динамические
76. статические и фактические
77. динамические и фактические
78. в списке нет правильного ответа
79. **В статических массивах индексы являются**
80. константами
81. переменными целого типа
82. переменными любого типа
83. переменными строкового типа
84. **В динамических массивах индексы являются**
85. переменными целого типа
86. могут быть строковыми переменными
87. константами
88. в списке нет правильного ответа
89. **В числовых статических массивах оператор ERASE**
90. обнуляет элементы массива
91. освобождает память
92. объявляет новый массив
93. заполняет элементы массива минимальными значениями данного типа данных
94. **При объявлении строкового массива его элементы заполняются пустыми строками оператором**
95. DIM
96. INPUT
97. FOR
98. LET
99. **В числовых динамических массивах оператор ERASE**
100. освобождает память
101. объявляет новый массив
102. обнуляет элементы массива
103. заполняет элементы массива максимальными значениями данного типа данных
104. **С помощью оператора REDIM можно переопределять**
105. динамические массивы
106. статические массивы
107. любые массивы
108. только строковые массивы
109. **В строковых статических массивах оператор ERASE**
110. заполняет элементы массивов пустыми строками
111. освобождает память
112. объявляет новый массив
113. заполняет элементы массива максимальными значениями данного типа данных
114. **В динамических массивах переменные, определяющие границы массива, должны быть объявлены**
115. до оператора DIM
116. после оператора DIM
117. в конце программы
118. в любом месте программы
119. **В динамических строковых массивах оператор ERASE**
120. освобождает память
121. обнуляет элементы массива
122. заполняет элементы массива минимальными значениями данного типа данных
123. в списке нет правильного ответа
124. **Для статических массивов память выделяется**
125. на этапе компиляции
126. в ходе выполнения программы
127. в начале выполнения программы
128. в конце выполнения программы
129. **Для динамических массивов память выделяется**
130. в ходе выполнения программы
131. в конце выполнения программы
132. на этапе компиляции
133. в списке нет правильного ответа
134. **Ввод и вывод элементов массива удобнее всего осуществлять с помощью оператора**
135. FOR
136. IF
137. SELECT CASE
138. в списке нет правильного ответа
139. **Сортировка массива – это**
140. упорядочивание элементов массива либо по возрастанию, либо по убыванию
141. перезапись элементов массива в обратном порядке
142. удаление нулевых элементов массива
143. в списке нет правильного ответа
144. **Значением параметра цикла в операторе FOR при вводе и обработке массивов является**
145. номер элемента массива
146. значение элемента массива
147. размерность элемента массива
148. размер массива
149. **Для вывода элементов массива в одну строку в списке вывода оператора PRINT нужно поставить**
150. точку с запятой
151. запятую
152. двоеточие
153. восклицательный знак
154. **При объявлении массива диапазон изменения индексов указывается следующим образом**
155. Imin TO Imax
156. Imin, Imax
157. Imin-Imax
158. Imin/Imax
159. **Двумерный массив – это массив, у которого количество индексов равно**
160. 2
161. 4
162. 1
163. 3
164. **Если данные представлены в табличной форме, то для их обработки удобнее использовать**
165. двумерный массив
166. одномерный массив
167. трехмерный массив
168. любой массив
169. **Индексами двумерного массива могут быть**
170. любые целочисленные выражения
171. выражения любого типа
172. только целочисленные переменные
173. переменные любого типа
174. **Первый индекс в двумерном массиве указывает**
175. номер строки
176. номер столбца
177. не имеет значения
178. все зависит от программиста
179. **Укажите размер данного двумерного массива A (2,3)**
180. 12
181. **Второй индекс в двумерном массиве указывает**
182. номер столбца
183. номер строки
184. все зависит от программиста
185. не имеет значения
186. **Перед использованием двумерного массива в программе его необходимо объявить с помощью оператора**
187. DIM
188. REM
189. INPUT
190. FOR
191. **При объявлении массива диапазон изменения индексов указывается**

**следующим образом**

* 1. Imin TO Imax
  2. Imin, Imax
  3. Imin - Imax
  4. Imin/ Imax

1. **Ввод и обработка двумерных массивов производится при использовании**
2. вложенных циклов
3. вложенных разветвлений
4. вложенных процедур
5. любых конструкций языка программирования
6. **Значениями параметров цикла в операторе FORпри вводе, обработке и выводе массивов является**
7. номера строк и столбцов
8. значения элементов массива
9. размерность массива
10. минимальные значения индексов массива
11. **При вводе и обработке массива по строкам внешний цикл должен быть организован**
12. по первому индексу
13. по второму индексу
14. по любому из индексов
15. зависит от программиста
16. **При вводе и обработке массива по столбцам внешний цикл должен быть организован**
17. по второму индексу
18. по любому из индексов
19. по первому индексу
20. все зависит от программиста
21. **Для организации вывода двумерного массива в виде таблицы в списке вывода оператора PRINT нужно поставить** 
    1. точку с запятой
    2. запятую
    3. двоеточие
    4. пробел
22. **Для вывода двумерного массива в виде таблицы после завершения внутреннего цикла нужно поставить**
23. оператор PRINT
24. два пробела
25. двоеточие
26. пробел
27. **Укажите количество объявленных массивов**

**DIM A (0 to 3, 4)... DIM A (3, 4)... DIMA (3, 0 to 4)... DIMA (0 to 3, 0 to 4)**

1. 1
2. 2
3. 3
4. 4
5. **Определите, какой тип распределения памяти имеет объявленный массив**

**DIM A (1 TO 3, 1 TO 5) AS SINGLE**

1. статический
2. динамический
3. фактический
4. вещественный
5. **Символьным (строковым) выражением в языке программирования QBявляется**
6. последовательность букв русского или латинского алфавита, цифр или других символов, взятая в кавычки
7. последовательность букв русского или латинского алфавита
8. последовательность букв русского алфавита
9. последовательность букв русского или латинского алфавита, цифр или других символов, начинающаяся с латинской буквы и взятая в кавычки
10. **Ошибочной записью символьной переменной в языке QB является**
11. C$1
12. B$
13. C1$
14. нет верного ответа
15. **Операция склеивания символов обозначается знаком**
16. &
17. /
18. нет верного ответа
19. **Подсчитать количество символов в выражении позволяет функция**
20. LEN
21. LEFT
22. ASC
23. MID
24. **Вырезать любое количество нужных символов из символьного выражения можно с помощью функции**
25. MID
26. LEFT
27. ASC
28. LEN
29. **Определить правильно написание имени символьной переменной**
30. d34$
31. 2A$
32. ЭF$
33. DD
34. **Операции, выполняемые с помощью символьных функций, возможны**
35. с любой информацией, хранимой в символьной переменной
36. только с буквами и фразами
37. с числовой информацией
38. с любой информацией, хранимой в числовой переменной
39. **Вырезать определенное количество символов слева из выражения позволит функция**
40. LEFT
41. ASC
42. LEN
43. нет верного ответа
44. **Значением функции LEN («СИМВОЛЬНЫЕ ПЕРЕМЕННЫЕ») является**
45. 21 \*
46. 0
47. 14
48. 2
49. **Операции, допустимые для строковых данных**
50. >, < , = , < = , > = , <> , +
51. + , =
52. =
53. нет верного ответа
54. **Преобразовать данные числового типа в данные символьного типа**
55. можно
56. нельзя
57. не всегда возможно
58. **Преобразовать данные строкового типа в данные числового типа**
59. не всегда возможно
60. можно
61. нельзя
62. **Функция** LEN (A$)
63. возвращает количество символов строки А$
64. возвращает код ASCII строки А$
65. удаляет пробелы из строки A$
66. не применяется к строкам
67. нет верного ответа
68. **Функция LEFT$ (A$, n)**
69. возвращает n первых символов строки A$
70. возвращает n последних символов строки A$
71. не применяется к строкам
72. находит символ, стоящий на позиции n в строке A$
73. нет верного ответа

**15 ФункцияRIGHT$ (A$, n)**

1. возвращает n последних символов строки A$
2. возвращает n первых символов строки A$
3. не применяется к строкам
4. находит символ, стоящий на позиции n в строке A$
5. нет верного ответа
6. **Функция MID$ (A$, n, m)**
7. вырезает из строки A$ , начиная с n-ого, m символов
8. вырезает из строки A$ , начиная с m-ого , n cимволов,
9. не применяется к строкам
10. нет верного ответа
11. **Функция MID$ (A$, n)**
12. вырезает из строки все символы, начиная с n-ого
13. вырезает из строки n первых символов
14. вырезает из строки n последних символов
15. нет верного ответа
16. **Функция SPACE$ (n)**
17. возвращает строку из nпробелов
18. возвращает строку из n первых символов
19. возвращает строку из n последних символов
20. удаляет n начальных пробелов
21. удаляет n конечных пробелов
22. **Функция STRING$ (n, «\*»)**
23. возвращает строку из n звездочек
24. вставить символ «\*» в n-ую позицию
25. ошибочно записана
26. нет верного ответа
27. **Функция LCASE$ (A$)**
28. заменяет прописные буквы на строчные
29. заменяет строчные буквы на прописные
30. удаляет начальные пробелы строки
31. удаляет конечные пробелы строки
32. нет верного ответа
33. **Функция UCASE$ (A$)**
34. заменяет строчные буквы на прописные
35. заменяет прописные буквы на строчные
36. удаляет начальные пробелы строки
37. удаляет конечные пробелы строки
38. нет верного ответа
39. **Функция LTRIM$ (A$)**
40. удаляет начальные пробелы строки
41. заменяет прописные буквы на строчные
42. заменяет строчные буквы на прописные
43. удаляет конечные пробелы строки
44. нет верного ответа

**22 Функция RTRIM$ (A$)**

1. удаляет конечные пробелы строки
2. заменяет прописные буквы на строчные
3. заменяет строчные буквы на прописные
4. нет верного ответа
5. **Функция INSTR (A$, B$)**
6. определяет номер позиции, в которой впервые встретилась строка В$ в A$
7. определяет номер позиции, в которой впервые встретилась строка A$ в строке B$
8. сравнивает строки A$ и B$
9. производит обмен значениями строк A$ и B$
10. не применяется к строкам
11. Функция **INSTR (n, A$, B$)**
12. начиная с n-й позиции строки A$, определяет позицию вхождения в нее строки В$
13. начиная с n-й позиции строки B$, определяет позицию вхождения в нее строки A$
14. заменяет n символов строки A$ на n символов строки B$
15. ошибочно записана
16. не применяется к строкам
17. **Функция CHR$ (n)**
18. возвращает символ, код ASCII которого равен числу n
19. возвращает строку из n пробелов
20. возвращает строку из n первых символов
21. возвращает строку из n последних символов
22. **Функция ASC (A$)**
23. возвращает ASCII код первого символа строки A$
24. возвращает количество символов в строке А$
25. преобразует число в строку A$
26. не применяется к строкам
27. **Функция VAL (A$)**
28. преобразует строку A$ в число, если это возможно
29. возвращает длину строки
30. определяет ASCII код строки
31. не применяется к строкам
32. **Функция STR$ (n)**
33. преобразует число n в строковое представление
34. преобразует число n в строковое представление, если это возможно
35. возвращает строку из n пробелов
36. нет верного ответа
37. **Функция RND (A$)**
38. не применяется к строкам
39. удаляет начальные пробелы из строки
40. удаляет конечные пробелы из строки
41. вставляет случайное число в строку А$
42. преобразует строку А$ в число, если это возможно
43. **Функция SGN (A$)**
44. не применяется к строкам
45. удаляет начальные пробелы из строки
46. удаляет конечные пробелы из строки
47. вставляет случайное число в строку А$
48. преобразует строку А$ в число, если это возможно
49. **Функция RND (A$, B$)**
50. не применяется к строкам
51. определяет номер позиции, в которой впервые встретилась строка A$ в строке B$
52. сравнивает строки A$ и B$
53. производит обмен значениями строк A$ и B$
54. вставляет в строку А$ строку B$ случайным образом
55. **Оператор SWAP A$, B$**
56. производит обмен значениями строковых переменных A$ и B$
57. сравнивает строки A$ и B$
58. определяет номер позиции, в которой впервые встретилась строка В$ в строке A$
59. определяет номер позиции, в которой впервые встретилась строка A$ в строке B$
60. не применяется к строкам
61. **Оператор SWAP (A$, n)**
62. нет верного ответа
63. преобразует число n в строку A$
64. преобразует строку A$ в число n
65. удаляет n символов из строки A$
66. **Процедура – это**
67. последовательность логически связанных фрагментов программы, оформленная как отдельная часть программы специальным способом
68. последовательность отдельных фрагментов программы
69. последовательность только циклических фрагментов программы
70. последовательность только разветвляющихся фрагментов программы
71. **Глобальные переменные – это**
72. переменные одной процедуры, доступные в других процедурах без механизма передачи параметров
73. все переменные одной отдельной процедуры
74. все переменные всех используемых процедур
75. в списке нет правильного ответа
76. **Локальные переменные – это**
77. переменные с одинаковыми именами в разных процедурах, которые являются независимыми
78. все переменные с одинаковыми именами во всех процедурах
79. переменные только одной процедуры
80. переменные всех процедур
81. **Статические переменные – это**
82. локальные переменные, которые сохраняют свои значения между вызовами процедур
83. все переменные, передаваемые в процедуру
84. переменные, передаваемые в процедуру без механизма передачи параметров
85. переменные, передаваемые в вызывающую процедуру
86. **Формальными параметрами называется**
87. список передаваемых параметров, указанных в описании процедуры
88. список параметров, указанных в главной программе
89. все переменные, используемые в процедуре
90. в списке нет правильного ответа
91. **Формальные параметры являются для данной процедуры**
92. локальными
93. глобальными
94. фактическими
95. формализованными
96. **Формальные параметры определяют**
97. имя и тип переменных
98. только имена переменных
99. только типы массивов
100. в списке нет правильного ответа
101. **При вызове процедуры происходит замена формальных параметров**
102. фактическими
103. локальными
104. глобальными
105. статическими
106. **Фактические параметры – это**
107. конкретные значения переменных, на которые заменяются формальные параметры\*
108. глобальные переменные
109. локальные переменные
110. статические переменные
111. **Параметры могут передаваться в процедуру**
112. по значению или по адресу (ссылке)
113. только по значению
114. только по адресу
115. любым образом
116. **При передаче параметра по значению в процедуру передается**
117. копия передаваемого параметра
118. адрес передаваемого параметра
119. адрес и значение передаваемого параметра
120. ничего не передается
121. **По значению могут передаваться**

1) входные параметры процедуры

2) выходные параметры процедуры

3) как входные, так и выходные параметры процедуры

4) только глобальные переменные

1. **По адресу могут передаваться**

1) как входные, так и выходные параметры процедуры

2) только входные параметры процедуры

3) только выходные параметры процедуры

4) в списке нет правильного ответа

1. **Изменение в процедуре параметра, передаваемого по значению**
2. не приводит к изменению значения этого параметра в вызывающей процедуре
3. приводит к изменению значения этого параметра в вызывающей процедуре
4. приводит к изменению значений всех параметров вызывающей процедуры
5. в списке нет правильного ответа
6. **Изменение в процедуре параметра, передаваемого по адресу**
7. приводит к изменению значения этого параметра в вызывающей процедуре
8. не приводит к изменению значения этого параметра в вызывающей процедуре
9. приводит к изменению значений всех параметров вызывающей процедуры
10. приводит к изменению адреса передаваемого параметра
11. **Если функция описывается одним арифметическим, логическим или строковым выражением, то для ее описания (определения) лучше использовать**
12. однострочную функцию
13. многострочную функцию
14. процедуру
15. процедуру-функцию
16. **К средствам процедурного программирования в QuickBASIC относятся**
17. однострочные и многострочные функции, процедуры- функции, процедуры
18. многострочные функции, процедуры
19. однострочные функции, процедуры- функции
20. gроцедуры
21. **Передача параметров в однострочные и многострочные функции осуществляется**
22. по значению
23. по адресу
24. как по значению, так и по адресу
25. в однострочные функции по значению, в многострочные - по адресу
26. **Результатом выполнения функции является**
27. скалярная величина
28. несколько скалярных величин
29. массивы
30. массивы и скалярные величины
31. **В качестве передаваемых параметров в однострочных и многострочных функциях используются**
32. скалярные величины
33. массивы
34. массивы и скалярные величины
35. в списке нет правильного ответа
36. **Переменные внутри однострочных и многострочных функций являются**
37. глобальными
38. локальными
39. статическими
40. фактическими
41. **Чтобы сделать отдельные переменные внутри многострочнойфункции локальными, их нужно объявить в теле функции оператором**
42. STATIC \*
43. SHARED
44. DIM
45. DEF FN
46. **Для досрочного выхода из многострочной функции используется оператор**
47. EXIT DEF
48. EXIT FUNCTTION
49. EXIT DO
50. EXIT SUB
51. **При вызове функций и процедур контролируется**
52. количество и тип аргументов, их порядок следования
53. количество аргументов, их порядок следования
54. порядок следования аргументов
55. количество и тип аргументов
56. **Передача параметров в процедуру и в процедуру-функцию производится**
57. как по адресу, так и по значению
58. только по адресу
59. только по значению
60. в процедуру-функцию по значению, а в процедуру – по адресу
61. **Передача фактических параметров-выражений в процедуру и в процедуру-функцию производится**
62. по значению
63. по адресу
64. как по адресу, так и по значению
65. в процедуру - по адресу, в процедуру-функцию – по значению
66. **Передача параметров-массивов в процедуру и в процедуру-функцию производится**
67. по адресу
68. по значению
69. как по адресу, так и по значению
70. в процедуру - по адресу, в процедуру-функцию – по значению
71. **Для того, чтобы передать параметр-переменную по значению, нужно в списке фактических параметров**
72. его имя заключить в круглые скобки
73. после его имени поставить пустые круглые скобки
74. перед его именем поставить пустые круглые скобки
75. просто указать его имя
76. **Параметры-переменные в процедуру и в процедуру-функцию могут передаваться**
77. как по адресу, так и по значению
78. по значению
79. по адресу
80. в процедуру - по адресу, в процедуру-функцию – по значению
81. **Переменные, описанные внутри процедуры и процедуры-функции, являются**
82. локальными
83. глобальными
84. статическими
85. формальными
86. **Если предаваемый параметр является массивом, то в списке параметров нужно**
87. после его имени поставить пустые круглые скобки
88. перед его именем поставить пустые круглые скобки
89. его имя заключить в круглые скобки
90. **Для досрочного выхода из процедуры-функции используется оператор**
91. EXIT FUNCTION
92. EXIT SUB
93. EXIT DEF
94. EXIT DO
95. **Чтобы переменные или массивы, объявленные в главной программе, были доступны только данной процедуре, их надо объявить в теле этой процедуры оператором**
96. SHARED
97. STATIC
98. DIM
99. DEFSNG
100. **Вызов процедуры-функции из вызывающей программы осуществляется**
101. указанием ее имени и списка фактических параметров
102. указанием ее имени и списка формальных параметров
103. оператором CALL с указанием ее имени и списка фактических параметров
104. оператором CALL с указанием ее имени
105. **Чтобы локальные переменные в процедурах и процедурах-функциях сохраняли свои значения между вызовами, нужно в теле процедур объявить их оператором**
106. STATIC
107. SHARED
108. SINGLE
109. DIM
110. **Результатом выполнения процедуры может быть**
111. несколько скалярных величин или массивов
112. только одна скалярная величина
113. только один массив
114. в списке нет правильного ответа
115. **Вызов процедуры из вызывающей программы осуществляется**
116. оператором CALL с указанием ее имени и списка фактических параметров
117. указанием ее имени и списка фактических параметров
118. оператором CALL с указанием ее имени
119. оператором CALL с указанием ее имени и списка формальных параметров
120. **В качестве выходного параметра входной параметр может использоваться**
121. в процедурах
122. в процедурах-функциях
123. и в процедурах, и в процедурах-функциях
124. в многострочных функциях
125. **Процедуры и процедуры-функции должны быть описаны (определены)**
126. после конечного END главной программы
127. в начале главной программы
128. 3) перед конечным END главной программы
129. 4) в любом месте главной программы
130. **Если входной параметр является и выходным параметром процедуры, то он передается**
131. по адресу
132. по значению
133. и по адресу, и по значению
134. в списке нет правильного ответа
135. **Досрочный выход из процедуры осуществляется оператором**
136. EXIT SUB
137. EXIT DEF
138. EXIT FUNCTION
139. EXIT FOR
140. **Описания однострочных и многострочных функций могут находиться**
141. в главной программе до обращения к этим функциям
142. в конце главной программы
143. в любом месте главной программы
144. после конечного END главной программы

# **2.5.5. Контрольные вопросы по теме «Основные конструкции языка программирования С++»**

# **2.5.6. Тестовые задания по теме «Основные конструкции языка программированияC++»**