[8. Операции, выражения и операторы.](#_Toc515770019)

[8.1. Основные операции.](#_Toc515770020)

[8.1.1. Операция присваивания: =.](#_Toc515770021)

[8.1.2. Операция сложения: +.](#_Toc515770022)

[8.1.3. Операция вычитания: —.](#_Toc515770023)

[8.1.4. Операция изменения знака: -.](#_Toc515770024)

[8.1.5. Операция умножения: \*.](#_Toc515770025)

[8.1.6. Операция деления: /.](#_Toc515770026)

[8.1.7. Поразрядные операции.](#_Toc515770027)

[8.1.7.1. Поразрядное И (AND).](#_Toc515770028)

[8.1.7.2. Поразрядное ИЛИ (OR).](#_Toc515770029)

[8.1.7.3. Поразрядное исключающее ИЛИ (OR).](#_Toc515770030)

[8.1.8. Поразрядный сдвиг влево и вправо.](#_Toc515770031)

[8.1.9. Операции отношения и логические операции.](#_Toc515770032)

[8.1.10. Условная операция (?:).](#_Toc515770033)

[8.1.11. Операция запятая (,).](#_Toc515770034)

[8.1.12. Порядок выполнения операций.](#_Toc515770035)

[8.2. Дополнительные операции.](#_Toc515770036)

[8.2.1. Операция деления по модулю: %.](#_Toc515770037)

[8.2.2. Операции увеличения и уменьшения: ++.](#_Toc515770038)

[8.2.3. Операция уменьшения: --.](#_Toc515770039)

[8.2.4. Старшинство операций.](#_Toc515770040)

[8.3. Выражения и операторы.](#_Toc515770041)

[8.3.1. Выражения.](#_Toc515770042)

[8.3.2. Операторы.](#_Toc515770043)

[8.3.3. Составные операторы (блоки).](#_Toc515770044)

[8.4. Преобразование типов.](#_Toc515770045)

[8.5. Операция приведения.](#_Toc515770046)

1. Операции, выражения и операторы.

Здесь же мы рассмотрим способы обработки данных — для этого язык Си имеет широкий набор возможностей. Начнем с основных арифметических операций: сложения, вычитания, умножения и деления. Чтобы сделать наши программы более интересными и поучительными, мы впервые здесь коснемся циклов. А пока, чтобы ввести вас в курс дела, приведем простую программу, выполняющую несложные арифметические действия.

// размер обуви1

#include <stdio.h>

#define OFFSET 7.64

#define SCALE 0.325

void main ()

{

// пересчет размера обуви в размер ноги в дюймах

double shoe, foot;

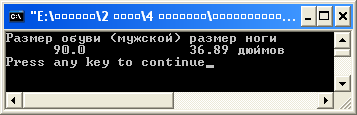
shoe = 90;

foot = SCALE\* shoe + OFFSET;

printf(" Размер обуви (мужской) размер ноги\n");

printf("%10.1f %17.2f дюймов\n", shoe, foot);

}



Здорово! в нашей программе выполняется умножение и сложение, т. е. берется ваш размер обуви (если вы носите размер 9), а вам сообщается длина стопы в дюймах. Вы скажете, что могли бы решить данную задачу в уме за меньшее время, чем потребовалось для ввода самой программы в машину. Это, конечно, правильно. Создание программы, способной оперировать только одним размером обуви, выглядит как ненужная трата времени и усилий. Мы могли бы придать программе большую эффективность, сделав ее диалоговой, но и это окажется непроизводительным использованием возможностей машины.

Нам нужно лишь каким-то образом заставить компьютер выполнять повторяющиеся вычисления. Вообще говоря, именно эта потребность является одной из главных причин использования машин для выполнения арифметических вычислений. Язык Си предлагает несколько способов реализации повторяющихся вычислений; сейчас мы кратко обсудим один из них. Данный способ, называемый "циклом while", дает возможность использовать операторы языка более интересным образом. Ниже приводится модификация нашей программы, занимающейся пересчетом размеров обуви в размер ноги.

// X10.C

// размер обуви2

#define OFFSET 7.64

#define SCALE 0.325

#include <stdio.h>

void main(void)

{

// пересчет размера обуви в размер ноги в дюймах

double shoe, foot;

printf("Размер обуви (мужской) размер ноги\n");

shoe = 3.0;

while (shoe < 18.5)

{

foot = SCALE\*shoe + OFFSET;

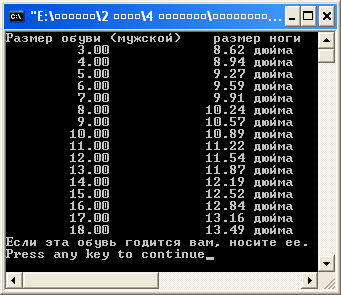
printf("%13.2f %16.2f дюйма\n", shoe, foot);

shoe = shoe + 1.0;

}

printf("Если эта обувь годится вам, носите ее.\n");

}



(Значения констант для данной программы пересчета были получены во время нашего визита инкогнито в обувной магазин. В единственном обнаруженном там классификаторе размеров приводились данные только относительно мужской обуви. Лица, интересующиеся размерами женской обуви, должны посетить обувной магазин сами).

Цикл while работает следующим образом. Когда программа в процессе выполнения впервые достигает оператора while, осуществляется проверка истинности условия, заключенного в круглые скобки. В этом случае соответствующее выражение имеет вид

shoe < 18.5

где символ < означает «меньше». Вначале переменная shoe была инициализирована значением 3.0, которое, как видно, меньше 18.5. Поэтому данное условие истинно, и осуществляется переход к следующему оператору, который переводит размер обуви в дюймы. После этого результаты выводятся на печать. Следующий оператор

shoe = shoe + 1.0;

увеличивает значение переменной shoe на 1.0, делая его равным 4.0. В этом месте программы происходит возврат к началу фрагмента while, где вышеупомянутое условие проверяется вновь. Почему именно здесь? Это происходит потому, что следующей строкой программы является закрывающая фигурная скобка } — тело цикла while заключено в фигурные скобки. Операторы, расположенные между ними, являются той частью программы, которая может выполняться повторно. Теперь давайте вернемся к нашей программе. 4 меньше 18.5? Безусловно. Поэтому весь набор операторов, заключенный в фигурные скобки и следующий за ключевым словом while, выполнится опять. (Специалисты по вычислительной технике в этом случае говорят, что программа выполняет эти операторы «в цикле»). Это продолжается, пока значение переменной shoe не достигнет величины 19.0. Тогда условие

shoe < 18.5

станет ложным, поскольку 19.0 не меньше 18.5. При этом произойдет передача управления оператору, следующему сразу за телом цикла while. В нашем случае им является завершающий оператор printf().

Вы можете легко модифицировать эту программу, чтобы она осуществляла другие преобразования. Например, замените значение константы SCALE на 1.8, а константы OFFSET — на 32.0, и вы получите программу, которая переводит температуру по Цельсию в температуру по Фаренгейту. Если заменить значение SCALE на 0.6214, a OFFSET на 0, то программа будет переводить мили в километры. Производя эти изменения, вы, по-видимому, во избежание путаницы должны будете поменять также и печатаемые сообщения.

Цикл while служит удобным и гибким средством управления выполнения программы. Вернемся теперь к обсуждению набора основных операций, которые мы можем использовать в программах.

В С существует много операций, отсутствующих в других языках. К их числу относятся поразрядные операции, операции инкремента и декремента, условные операторы, операция запятая, операции присваивания и составного присваивания.

## Основные операции.

«Операции» в языке Си применяются для представления арифметических действий. Например, выполнение операции + приводит к сложению двух величин, стоящих слева и справа от этого знака. Если слово «операция» кажется вам странным, подумайте тогда, как назвать эти понятия. Нам слово «операция» представляется лучшим термином, чем, скажем, «арифметические транзакторы». Ниже мы рассмотрим операции =,+,—,\* и /. (В языке Си нет операции возведения в степень. Позднее будет представлена программа, реализующая данную функцию).

Для эффективного использования возвращаемого операциями значения предназначен оператор присваивания (=) и его модификации: сложение с присваиванием (+=), вычитание с присваиванием (-=), умножение с присваиванием (\*=), деление с присваиванием (/=), модуль с присваиванием (%=) и ряд других, которые будут рассмотрены позже.

### Операция присваивания: =.

В языке Си знак равенства не означает «равно». Он означает операцию присваивания некоторого значения. С помощью оператора

bmw = 2002;

переменной с именем bmw присваивается значение 2002, т. е элемент слева от знака = — это имя переменной, а элемент справа — ее значение. Мы называем символ = «операцией присваивания». Еще раз хотим обратить ваше внимание на то, что смысл указанной строки не выражается словами «bmw равно 2002». Вместо этого нужно говорить так: «присвоить переменной bmw значение 2002». В этой операции действие выполняется справа налево.

Возможно, различие между именем переменной и ее значением покажется вам незначительным. В таком случае давайте рассмотрим следующий сравнительно часто используемый при программировании оператор:

i = i + 1;

С математической точки зрения это бессмыслица. Если вы прибавляете единицу к конечному числу, результат не может быть равен исходному числу. Но как оператор присваивания, данная строка имеет вполне определенный смысл, который можно выразить, например, такой длинной фразой: «Взять значение переменной с именем i, к нему прибавить 1, а затем присвоить новое значение переменной с именем i».

i = i+1;

Оператор вида 2002 = bmw; на языке Си не имеет смысла, поскольку 2002 — число. Вы не можете присвоить константе какое-то значение; ее значением является она сама. Поэтому, сидя за клавиатурой, помните, что элемент, стоящий слева от знака =, всегда должен быть именем переменной.

Тем из вас, кто предпочитает знать правильные названия понятий, скажем, что вместо использованного ранее термина «элемент» обычно употребляют слово «операнд». Операнды — это то, над чем выполняются операции. Например, вы можете описать процесс «поедания» гамбургера как применение операции «поедание» к операнду «гамбургер».

Операция присваивания в языке Си представляется несколько более интересной, чем в большинстве других языков. Попробуйте выполнить приведенную ниже короткую программу.

// таблица результатов турнира по гольфу

#include <stdio.h>

void main ()

{

int jane, tarzan, cheeta;

cheeta = tarzan = jane = 68;

printf("cheeta tarzan jane\n");

printf("Счет первой партии %4d %8d %8d \n", cheeta, tarzan, jane);

}

В то время как многие языки запрещают применять такое тройное присваивание, присутствующее в данной программе, для Си это обычная практика. Присваивания выполняются справа налево: сначала переменная jane получает значение 68, затем переменная tarzan и, наконец, переменная cheeta. Результат выглядит так:



В языке Си имеется несколько других операций присваивания, которые отличаются от операции, описанной в данном разделе, и мы обещаем рассказать о них в следующих темах.

Присваивание в С отличается от присваивания в других языках программирования. Присваивание выполняет операция, а не оператор. Аналогично другим операциям С, результат операции присваивания является некоторым значением, которое тоже можно присваивать. Выражение с операцией присваивания можно использовать в длинных выражениях, подобных следующему:

value1 = 8\*(value2=5);

В этом примере сначала переменной value2 присваивается значение 5. Затем оно умножается на 8, и окончательный результат 40 заносится в переменную value1.

Чрезмерно частое использование операции присваивания может быстро привести к неудобоваримым выражениям. Эта операция обычно используется в двух случаях. Во-первых, ее можно использовать для задания нескольким переменным некоторого значения:

value1 = value2 = value3 = 0;

Во-вторых, чаще всего она используется в условном выражении цикла while, как, например:

while ((с = getchar()) != EOF)

{

}

Здесь значение, возвращаемое функцией getchar(), присваивается переменной с, а затем это значение сравнивается с EOF. Если оно равно EOF (конец файла — end of file), цикл не выполняется. Необходимы скобки, так как операция присваивания имеет более низкий приоритет, чем операция "не равно". Иначе строка интерпретировалась бы следующим образом:

с = (getchar()!= EOF)

Переменная с получала бы значение 1 (TRUE — истина) всякий раз, когда getchar() возвращала бы значение EOF.

### Операция сложения: +.

Выполнение операции + приводит к сложению двух величин, стоящих слева и справа от этого знака. Например, в результате работы оператора

printf("%d" , 4 + 20);

на печать будет выведено число 24, а не выражение

4+20

Операнды могут быть как переменными, так и константами. Поэтому при выполнении оператора

income = salary + bribes;

компьютер возьмет значения двух переменных, стоящих в правой части, сложит их и присвоит затем полученную сумму переменной income.

Операция + называется «бинарной», или «диадической». Эти названия отражают тот факт, что она имеет дело с двумя операндами.

### Операция вычитания: —.

Выполнение операции вычитания приводит к вычитанию числа, расположенного справа от знака -, из числа, стоящего слева от этого знака. Оператор

takehome = 224.00 - 24.00;

присваивает переменной takehome значение 200.

### Операция изменения знака: -.

Знак минус используется также для указания или изменения алгебраического знака некоторой величины. Например, в результате выполнения последовательности операторов

rocky = - 12;

smokey = -rocky;

переменной smokey будет присвоено значение 12.

Когда знак минус используется подобным образом, данная операция называется «унарной». Такое название указывает на то, что она имеет дело только с одним операндом.

_06_01

### Операция умножения: \*.

Операция умножения обозначается знаком: \*. При выполнении оператора

cm = 2.54 \* in;

значение переменной in умножается на 2.54, и результат присваивается переменной cm.

Вы хотите иметь таблицу квадратов натуральных чисел? В языке Си нет функции, осуществляющей возведение в квадрат; вместо этого мы можем использовать умножение.

/\* X11.C \*/

#include <stdio.h>

void main(void) // получение таблицы квадратов

// цикл WHILE

{

int n = 1;

while (n < 21)

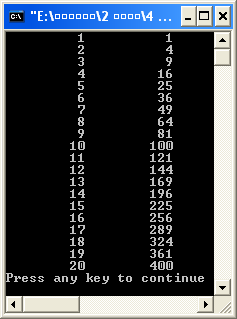
{

printf("%10d %10d\n",n,n\*n);

n = n+1;

}

}



Эта программа выводит на печать первые 20 целых чисел и их квадраты, в чем вы сами легко можете убедиться.

Теперь давайте рассмотрим более интересный пример.

Вы, наверное, слышали историю о том, как один могущественный правитель обещал наградить ученого, оказавшего ему большую услугу. Ученый, когда его спросили, что бы он хотел получить в награду, указал на шахматную доску и промолвил: «Положите одно пшеничное зерно на первую клетку, два — на вторую, четыре на третью, восемь на следующую и т. д.». Правитель, которому явно недоставало математической эрудиции, был поражен, услышав такую скромную просьбу,— ведь он уже приготовил богатые дары. Программа, приведенная ниже, показывает, в какое смешное положение попал правитель. В ней вычисляется количество зерен, которое надо положить на каждую клетку, а также промежуточные результаты (суммы зерен по числу клеток). Поскольку вы, возможно, не знакомы с урожаями пшеницы, мы, кроме того, сравниваем промежуточные суммы зерен с приблизительными цифрами годового урожая пшеницы в США.

// X12.C

// пшеница

#include <stdio.h>

#define SQUARES 64 // число клеток на доске

#define CROP 7E14 // количество зерен, выращиваемых в США за 1 год

void main(void) // эпизод из жизни умного ученого

{

double current,total,var;

int count = 1;

printf("номер число зерен сумма зерен доля\n");

printf("клетки от урожая в США\n");

total = current = 1.0; // интересная конструкция

// Где вы такое еще видели?

var = total/CROP;

printf("%4d %15.2e %13.2e %12.2e %12.2e %%\n",count,current,total,

var,var\*100.0);

while (count < SQUARES)

{

count = count + 1;

current = 2.0 \* current; // удвоение зерен в клетке

total = total + current; // коррекция суммы

var = total/CROP;

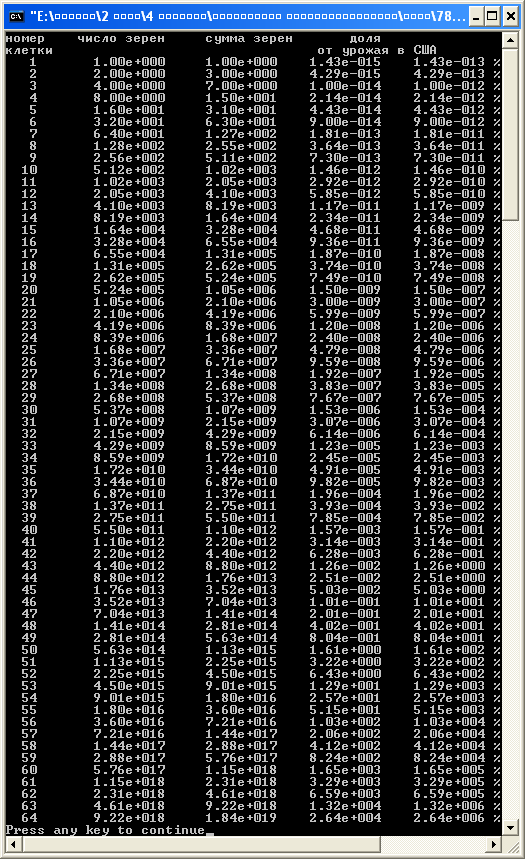
printf("%4d %15.2e %13.2e %12.2e %12.2e %%\n",count,current,total,

var,var\*100.0);

}

}

Вначале результаты работы программы выглядят довольно безобидно:



На первых 10 клетках оказалось чуть более тысячи зерен пшеницы. Но давайте посмотрим, сколько зерен на пятидесяти клетках!

Добыча ученого превысила весь годовой урожай пшеницы в США! Если вы захотите узнать, что окажется на 64 клетках, выполните программу сами.

Этот пример иллюстрирует феномен экспоненциального роста. Возрастание населения земного шара и использование нами энергетических ресурсов подчиняются тому же закону.

### Операция деления: /.

В языке Си символ / указывает на операцию деления. Величина, стоящая слева от этого знака, делится на величину, расположенную справа от него. Например, в результате выполнения оператора

four = 12.0/3.0;

переменной four будет присвоено значение 4.0. Заметим, что над данными целого типа операция деления производится не так, как над данными с плавающей точкой: в первом случае результат будет целым числом, а во втором — числом с плавающей точкой. У целого числа нет дробной части, что делает деление 5 на 3 затруднительным, поскольку результат не является целым. В языке Си принято правило, согласно которому дробная часть у результата деления целых чисел отбрасывается. Это действие называется «усечением».

Попробуйте выполнить приведенную ниже программу, чтобы посмотреть, как осуществляется усечение результата и чем деление целых чисел отличается от деления чисел с плавающей точкой.

//примеры деления

#include <stdio.h>

void main ( )

{

printf("деление целых: 5/4 это %d \n" , 5/4);

printf("деление целых: 6/3 это %d \n" , 6/3);

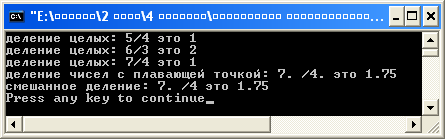
printf("деление целых: 7/4 это %d \n" , 7/4);

printf("деление чисел с плавающей точкой: 7. /4. это %2.2f \n" , 7./4.);

printf("смешанное деление: 7. /4 это %2.2f \n" , 7. /4);

}

Мы включили в нашу программу также случай «смешанных типов», осуществляя деление вещественного числа на целое. Язык Си менее строго «подходит» к подобным вопросам, чем некоторые другие языки, и позволяет выполнять такие операции, но, вообще говоря, смещения типов следует избегать. Вот результаты выполнения указанной программы:



Обратите внимание на то, что результат деления целых чисел округляется не до ближайшего целого, а всегда до меньшего целого числа. Когда мы смешиваем целые числа и числа с плавающей точкой, результат будет таким же, как если бы оба операнда были числами с плавающей точкой, поскольку в этом случае перед делением целое преобразуется в число с плавающей точкой.

Указанные свойства операции деления целых чисел оказываются довольно удобными при решении некоторых задач. Очень скоро мы приведем соответствующий пример. Нам осталось рассмотреть еще один важный вопрос: что происходит в тех случаях, когда в одном операторе используется несколько операций? Это и послужило нам темой для обсуждения, приведенной ниже.

### Поразрядные операции.

В поразрядных операциях переменная рассматривается как комбинация двоичных разрядов, а не как число. Эти операции полезны для обработки отдельных разрядов памяти, например, в видеопамяти для графического дисплея. Поразрядные операции могут использоваться только с целочисленными типами данных, и их нельзя применять к числам с плавающей точкой. Три поразрядных операции работают точно так же, как логические операции, но по отношению к отдельно взятым разрядам целого числа. Это операции И (&), ИЛИ (|) и исключающее ИЛИ (^). Еще одна операция — отрицание или дополнение до единицы (~), которая попросту инвертирует каждый разряд.

На практике довольно часто приходится отслеживать состояния различных программных объектов с помощью флагов. Для этой цели можно использовать булевские переменные. Однако, если у вас слишком много признаков, удобнее в качестве флагов использовать отдельные биты переменных. Для ПК на базе платформы Intel один байт представляет собой восемь бит информации, а каждый бит может принимать значение 0 или 1. Таким образом, в одном байте может быть закодировано любое целое число от 0 до 255 включительно (28 = 256). Биты в байте считаются справа налево.

Рассмотрим, например, панель из пяти выключателей SW1…SW5. Можно было бы объявить в качестве флагов 5 переменных типа bool (по 1 байту на каждый объест). Каждый выключатель может принимать состояние 'включено '(1 - true) или "выключено" (0 - falsе).

bool SW1, SW2, SW3, SW4, SW5;

Тогда, проверяя значения указанных выше переменных, можно судить о состоянии того или иного выключателя.

Принимая во внимание тот факт, что биты в байте также могут принимать значения только нуля или единицы, используя компактную запись, эту же панель можно было бы уместить всего в одном байте.



Рис. Варианты установки значений битовых переменных в байте

Поразрядное И (AND).

В операции поразрядного И сравниваются два разряда: если они оба равны 1, результат равен 1, что показано ниже:

**Логическое И**

Бит 0 Бит 1 Результат

0 0 0

0 1 0

1 0 0

1 1 1

Обратите внимание на отличие этой операции от двоичного сложения, при котором при сравнении двух единичных разрядов результат становится нулем, а флаг переноса устанавливается в 1. Очень часто операция поразрядного И используется для выделения, или маскирования, некоторых отдельных разрядов.

Поразрядное ИЛИ (OR).

В операции поразрядного ИЛИ сравниваются два разряда и в результате получается 1, если оба разряда или один из них равны 1. Это показано ниже:

**Логическое ИЛИ**

Бит 0 Бит 1 Результат

0 0 0

0 1 1

1 0 1

1 1 1

Операция поразрядного ИЛИ удобна для установки в 1 определенных разрядов.

Поразрядное исключающее ИЛИ (OR).

В операции поразрядного ИЛИ сравниваются два разряда и в результате получается 1, когда только один из разрядов равен 1. Это показано ниже:

**Исключающее ИЛИ**

Бит 0 Бит 1 Результат

0 0 0

0 1 1

1 0 1

1 1 0

Эта логическая операция может быть очень удобна, когда необходимо инвертировать определенные разряды — к примеру, в приложениях компьютерной графики.

Далее следует пример использования этих операций с константами в шестнадцатеричном и восьмеричном представлении. Для сравнения показаны двоичные значения.

|  |  |  |  |
| --- | --- | --- | --- |
| 0xF1 | & | 0x35 | равно 0x31 (шестнадцатеричное) |
| 0361 | & | 0065 | равно 061 (восьмеричное) |
| 11110011 | & | 00110101 | равно 00110011 (двоичное) |
| 0xF1 | | | 0x35 | равно 0xF5 (шестнадцатеричное) |
| 0361 | | | 0065 | равно 0365 (восьмеричное) |
| 11110011 | | | 00110101 | равно 11110111 (двоичное) |
| 0xF1 | ^ | 0x35 | равно 0xC4 (шестнадцатеричное) |
| 0361 | ^ | 0065 | равно 0304 (восьмеричное) |
| 1111001 | ^ | 00110101 | равно 00000000 11000110 (двоичное) |
| -0xF11 |  |  | равно 0xFF0E (шестнадцатеричное) |
| 0361 |  |  | равно 0177416 (восьмеричное) |
| 11110011 |  |  | равно 11111111 00001100 (двоичное) |

### Поразрядный сдвиг влево и вправо.

В С имеется две операции сдвига: влево (<<) и вправо (>>). При левом сдвиге разряды сдвигаются влево, а самый правый разряд (младший значимый) устанавливается в ноль. Самый левый (старший значимый) сдвигаемый разряд теряется.

Если рассматривать целые числа без знака, сдвиг числа влево на одну позицию и заполнение младшего значимого разряда (LSB) нулем означает умножение числа на 2. Следующий фрагмент на C++ показывает, как это записывается:

unsigned int value1 = 65;

value1 <<= 1;

cout << value1;

Если представить значение value1 в двоичном виде, можно увидеть следующие преобразования разрядов:

<< 0100 0001 (65 десятичное)

1000 0010 (130 десятичное)

При операции правого сдвига разряды сдвигаются вправо. Младшие сдвигаемые разряды отбрасываются. Чтобы разделить беззнаковое целое на 2, достаточно сдвинуть разряды на один вправо и занести ноль в старший значащий разряд (MSB). Пример на С выглядит аналогично предыдущему; отличие лишь в операторе составного присваивания (описывается ниже в этой главе) и операторе вывода:

unsigned int value1 = 10;

value1 >>= 1;

printf("%d",value1);

В двоичном виде младший байт переменной value1 меняется следующим образом:

>> 0000 1010 (10 десятичное)

0000 0101 (5 десятичное)

### Операции отношения и логические операции.

Все операции отношения используются для сравнения значений операндов. Они дают результат не ноль, если результат сравнения — ИСТИНА (TRUE), и ноль, если — ЛОЖЬ (FALSE). Ниже следует список операций отношения С и C++:

|  |  |
| --- | --- |
| **Операция** | **Значение** |
| == | Равно (не путать с присваиванием) |
| != | Не равно |
| > | Больше, чем |
| < | Меньше, чем |
| >= | Больше или равно |
| <= | Меньше или равно |

Логические операции И (&&), ИЛИ (||) и отрицание (!) дают в результате ИСТИНА (!0) или ЛОЖЬ (ноль) в зависимости от логических отношений между аргументами. Операция логическое И (&&) дает результат ИСТИНА (!0) только тогда, когда оба аргумента имеют значение ИСТИНА (!0). Операция логическое ИЛИ (||) возвращает значение ЛОЖЬ (ноль), только если оба аргумента ложны (равны нулю). Операция отрицания (!) просто инвертирует значение. Ниже перечислены логические операции С и C++:

|  |  |
| --- | --- |
| **Операция** | **Значение** |
| ! | NOT (отрицание) |
| && | AND (и) |
| || | OR (или) |

В следующей программе проверяются различные комбинации операций отношения и логических операций. Сможете ли вы предсказать результат?

/\* 06OPRS.C

Программа на С, иллюстрирующая некоторые тонкости операций

отношения и логических операций\*/

#include <stdio.h>

main()

{

float foperand1, foperand2;

printf("\nEnter foperand1 and foperand2: " ); // Введите операнд1 и

scanf("%f%f",&foperand1,&foperand2); // операнд2

printf("\n foperand1 > foperand2 is %d", (foperand1 > foperand2));

printf("\n foperand1< foperand2 is %d", (foperand1 < foperand2));

printf("\n foperand1 >= foperand2 is %d", (foperand1 >= foperand2));

printf("\n foperand1 <= foperand2 is %d", (foperand1<= foperand2));

printf("\n foperand1 == foperand2 is %d", (foperand1== foperand2));

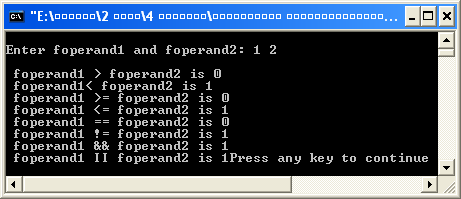
printf("\n foperand1 != foperand2 is %d", (foperand1 != foperand2));

printf("\n foperand1 && foperand2 is %d", (foperand1 && foperand2));

printf("\n foperand1 II foperand2 is %d", (foperand1 || foperand2));

return(0);

}



Вас могут удивить результаты некоторых логических операций. Запомните: точное равенство данных типа float и double возможно только при сравнении с нулем; далее, числа, которые хотя бы чуть-чуть отличаются, не равны. Кроме того, даже если некоторое число совсем немного отличается от нуля — в большую или меньшую сторону, — то оно остается истинным (!0). Далее следует программа на C++, эквивалентная рассмотренной:

// 06OPRS.CPP

// Программа на C++, иллюстрирующая некоторые тонкости операций

// отношения и логических операций

// Пример №8

#include <iostream.h>

main()

{

float foperand1, foperand2;

cout << "\nEnter foperand1 and foperand2: ";

cin >> foperand1 >> foperand2;

cout << "\n";

cout << " foperand1 > foperand2 is " << (foperand1 > foperand2) << "\n";

cout <<" foperand1 < foperand2 is " << (foperand1 < foperand2) << "\n";

cout << " foperand1 >= foperand2 is " << (foperand1 >= foperand2) << "\n";

cout << " foperand1 <= foperand2 is " << (foperand1 <= foperand2) << "\n";

cout << " foperand1 == foperand2 is " << (foperand1 == foperand2) << "\n";

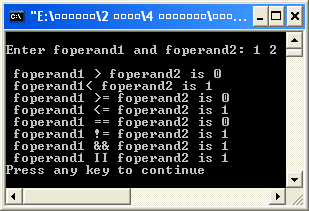
cout << " foperand1 != foperand2 is " << (foperand1 != foperand2) << "\n";

cout << " foperand1 && foperand2 is " << (foperand1 && foperand2) << "\n";

cout << " foperand1 || foperand2 is " << (foperand1 || foperand2) << "\n";

return(0);

}



### Условная операция (?:).

Условную операцию можно использовать при обычном программировании; однако, главное ее назначение — создание макросов. Операция имеет Следующий синтаксис

условие ? выражение\_если\_истина : выражение\_если\_ложь

Если условие истинно, значение условного выражения определяет операнд выражение\_если\_истина, в противном случае — выражение\_если\_ложь. Для примера рассмотрим следующий оператор:

if ('А' <= с && с <= 'Z')

printf("%с", 'а' + с - 'А');

else

printf("%c",с);

Этот оператор можно записать иначе при помощи условной операции:

printf(“%c”, ('А' <= с && с <= 'Z') ? ('а' + с - 'А') : c);

Оба оператора обеспечивают печать символа «с» в нижнем егистре.

### Операция запятая (,).

Операция запятая позволяет вычислять два выражения там, где по синтаксическим правилам возможно только одно. Значение операции запятая — значение выражения, стоящего справа. Синтаксис операции следующий:

левое\_выражение, правое\_ выражение

Операция "запятая" связывает между собой несколько выражений таким образом, что последние рассматриваются компилятором как единое выражение.

Благодаря использованию данной операции при написании программ достигается высокая эффективность. К примеру, в операторе ветвления if, рассматриваемом ниже, можно в качестве выражения ввести:

if(i = CallFunc(), i>7)

…

Тогда сначала выполнится вызов функции CallFunc() с присвоением результата переменной i, а затем произойдет сравнение значения i с числом 7.

Еще большей эффективности позволяет достичь использование операции "запятая" в операторе цикла for.

Например:

for(min=0,max=length-1; min < max; min++,max--)

{

}

Как в любой арифметике, в C++ операции выполняются в определенном порядке. Так, например, математические выражения вычисляются слева направо, в то время как оператор присвоения выполняется справа налево.

Для того чтобы компилятор мог разобраться, какое же действие осуществлять первым при выполнении операций, последние обладают важным свойством - приоритетом. В первую очередь выполняются операции с наивысшим приоритетом.

### Порядок выполнения операций.

Рассмотрим следующую строку:

butter = 25.0 + 60.0\*n/SCALE;

В этом операторе имеются операции сложения, умножения и деления. Какая операция будет выполнена первой? Будет ли 25.0 складываться с 60.0, затем результат 85.0 умножаться на n, а произведение делиться на значение константы SCALE? Или 60.0 умножается на n, результат складывается с 25.0, а сумма затем делится на величину SCALE? Или же существует какой-то другой порядок выполнения операций? Пусть переменная n равна 6.0, а константа SCALE — 2.0. Если вы выполните данные операции, используя эти значения, вы найдете, что при первом способе вычисления результат равен 255, а при втором — 192.5. При выполнении данной Си-программы на машине реализуется, по-видимому, какой-то другой порядок вычислений, поскольку на деле переменная butter получит значение 205.0.

Совершенно очевидно, что изменение порядка выполнения действий может приводить к различным результатам, поэтому язык Си нуждается в наборе непротиворечивых правил, указывающих, какое действие осуществлять первым. Язык Си делает это, задавая приоритет той или иной операции. Каждой операции назначается уровень старшинства. Умножение и деление имеют более высокий уровень, чем сложение и вычитание, поэтому они выполняются первыми. Если же две операции имеют один и тот же уровень старшинства, они выполняются в том порядке, в котором присутствуют в операторе. Для большинства операций обычный порядок — слева направо. (Операция = является исключением из этого правила). Поэтому в операторе

butter = 25.0 + 60.0\*n/SCALE; порядок операций следующий:

60.0\*n — первое умножение (или, возможно, деление) (если n = 6, то 60.0\*n = 360.0)

360.0/SCALE — второе умножение (или, возможно, деление) и, наконец (поскольку SCALE = 2.0)

25.0 + 180. — первое сложение (или, возможно, вычитание) дает 205.0. Многие программисты предпочитают представлять порядок вычислений с помощью диаграммы специального вида, называемой «деревом выражения».

Если вы захотите, скажем, чтобы сложение выполнялось перед делением, тогда вы должны делать то же, что и мы в приведенной ниже строке:

flour = (25.0 + 60.0\*n)/SCALE;

В первую очередь выполняется все, что заключено в скобки; внутри скобок действуют обычные правила. В данном примере сначала выполняется умножение, а затем сложение. С помощью этих действий вычисляется выражение в скобках, и только потом результат делится на значение константы SCALE.

Мы можем составить таблицу правил, касающихся уже использованных нами операций.

|  |  |
| --- | --- |
| **ОПЕРАЦИИ** | **ПОРЯДОК ВЫЧИСЛЕНИЯ** |
| () | : слева направо |
| - (унарный) | : слева направо |
| \*/ | : слева направо |
| + - (вычитание) | : слева направо |
| = | : справа налево |

Заметим, что два различных по смыслу употребления знака минус имеют разные приоритеты (уровни старшинства). Столбец «порядок вычисления» указывает, как операция связана со своими операндами. Например, унарный знак минус связан с величиной, стоящей справа от него, а при делении левый операнд делится на правый.

Попытаемся применить эти правила на более сложном примере.

// применение правил старшинства

#include <stdio.h>

void main()

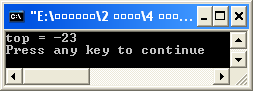
{

int top, score;

top = score = -(2 + 5)\*6 + (4 + 3\*(2 + 3));

printf("top = %d \n", top);

}



Какое значение будет выведено на печать в результате работы данной программы? Вначале вычислите его сами, а затем выполните программу или прочитайте нижеследующее объяснение, чтобы проверить свой ответ. (Надеемся, что вы получите правильный результат).

Итак, выражения, стоящие в скобках, имеют наивысший приоритет. Двигаясь слева направо, встречаем первое выражение в скобках (2 + 5). Вычисляя его, получаем

top = score = -7\*6 + (4 + 3\*(2 + 3))

Следующее выражение в скобках — это (4 + 3\*(2 + 3)). Отбрасывая скобки, получаем 4 + 3\*(2 + 3). Вот как! Еще одни скобки! Тогда первым шагом является нахождение суммы 2+3. Выражение примет вид

top = score = -7\*6 + (4 + 3\*5)

Мы должны еще завершить вычисление выражения в скобках. Поскольку умножение \* имеет приоритет более высокий, чем сложение +, выражение теперь выглядит так

top - score = -7\*6 + (4 + 15)

а затем

top = score = -7\*6 + 19

Что же дальше? Если вы предполагаете, что нужно найти произведение 7\*6, то вы ошибаетесь. Заметим, что унарный минус (изменение знака) имеет более высокий приоритет, чем умножение \*. Поэтому сначала число 7 заменяется на -7, а затем -7 умножается на 6. Строка примет вид

top = score = -42 + 19

после этого в результате сложения получим

top = score = -23

Затем переменной score присваивается значение -23, и, наконец, переменная top получает то же значение -23. Напомним, что операция = выполняется справа налево.

Порядок вычисления некоторого выражения в С определяется компилятором. Этот порядок обычно не влияет на значение выражения, если выражение не написано с учетом побочных эффектов. Побочный эффект дают операции, в которых значения переменных не только используются, но и изменяются, например операции инкремента и декремента. Другие операции с побочными эффектами — присваивание и составное присваивание.

Вызовы функций, изменяющих значения внешних переменных, также имеют побочные эффекты. Рассмотрим пример:

inum1 = 3;

ianswer = (inum1 =4) + inum1;

Это выражение можно вычислить двумя способами: 1) присвоить inum1 значение 4 и ianswer присвоить 8 (4+4); 2) сложить значение 3, полученное из inum1, и 4, которое затем присваивается inum1, и получить в результате 7.

Существуют, однако, четыре операции, для которых гарантирован порядок вычисления слева направо: логическое И (&&), логическое ИЛИ (||), операция запятая (,) и условная операция (?:). Поскольку по умолчанию имеется очередность выполнения выражения, можно написать такую типичную проверку:

while((c=getchar()!=EOF) && (c!='\n'))

Вторая часть операции логического И (&&) выполняется после того, как значение символа присвоено переменной с.

В этом разделе перечислены все операции С и C++ с указанием их приоритета, начиная с высшего и кончая низшим, и показан порядок вычисления (слева направо или справа налево). Все операции внутри строк имеют один приоритет.

В таблице приводятся используемые в C++ операции с указанием их приоритета и направления выполнения.

|  |  |  |  |
| --- | --- | --- | --- |
| **Оператор** | **Описание** | **Приоритет** | **Направление** |
| :: | Разрешение видимости | 16 | => |
| [] | Индекс массива | 16 | => |
| () | Вызов функции | 16 | => |
| . | Выбор члена структуры или класса | 16 | => |
| -> |
| ++ | Постфиксный инкремент | 15 | <= |
| -- | Постфиксный декремент | 15 | <= |
| ++ | Префиксный инкремент | 14 | <= |
| -- | Префиксный декремент | 14 | <= |
| sizeof  sizeof () | Размер | 14 | <= |
| (тип) | Преобразование типа | 14 |  |
| ^ | Побитовое исключающее ИЛИ | 14 | <= |
| ! | Логическое НЕ | 14 | <= |
| - | Унарный минус | 14 | <= |
| + | Унарный плюс | 14 | <= |
| & | Получение адреса | 14 | <= |
| \* | Разыменование | 14 | <= |
| new  new[] | Создание динамического объекта | 14 | <= |
| delete  delete[] | Удаление динамического объекта | 14 | <= |
| casting | Приведение типа | 14 |  |
| \* | Умножение | 13 | => |
| / | Деление | 13 | => |
| % | Остаток от деления | 13 | => |
| + | Сложение | 12 | => |
| - | Вычитание | 12 | => |
| >> | Сдвиг вправо | 11 | => |
| << | Сдвиг влево | 11 | => |
| < | Меньше | 10 | => |
| <= | Меньше или равно | 10 | => |
| > | Больше | 10 | => |
| >= | Больше или равно | 10 | => |
| == | Равно | 9 | => |
| != | Не равно | 9 | => |
| & | Побитовое И | 8 | => |
| ^ | Побитовое исключающее ИЛИ | 7 | => |
| | | Побитовое ИЛИ | 6 | => |
| && | Логическое И | 5 | => |
| | | | Логическое ИЛИ | 4 | => |
| ? : | Условие | 3 | <= |
| = | Присваивание | 2 | <= |
| \*= | Умножение с присваиванием | 2 | <= |
| /= | Деление с присваиванием | 2 | <= |
| % = | Модуль с присваиванием | 2 | <= |
| += | Сложение с присваиванием | 2 | <= |
| - = | Вычитание с присваиванием | 2 | <= |
| <<= | Сдвиг влево с присваиванием | 2 | <= |
| >>= | Сдвиг вправо с присваиванием | 2 | <= |
| &= | Побитовое И с присваиванием | 2 | <= |
| ^= | Исключающее ИЛИ с присваиванием | 2 | <= |
| != | Побитовое ИЛИ с присваиванием | 2 | <= |
| throw | Генерация исключения | 2 | <= |
| , | Запятая | 1 | => |

Язык C++ предоставляет программисту возможность самовольно задавать порядок выполнения вычислений. С этой целью, как и в математике, операнды группируются при помощи скобок, которые могут быть вложенными друг в друга.

## Дополнительные операции.

В языке Си имеется около 40 операций, но некоторые из них используются гораздо чаще, чем другие. Те операции, которые мы только что рассмотрели, являются наиболее общеупотребительными. Кроме того, нам хотелось бы привести еще три полезные операции.

### Операция деления по модулю: %.

Операция деления по модулю используется в целочисленной арифметике. Ее результатом является остаток от деления целого числа, стоящего слева от знака операции, на число, расположенное справа от него. Например, 13 % 5 (читается как «13 по модулю 5») имеет значение 3, поскольку справедливо равенство 13 = 2\*5 + 3.

Не пытайтесь производить данную операцию над числами с плавающей точкой: она просто не будет выполняться.

На первый взгляд эта операция может показаться некоторым экзотическим средством, используемым лишь математиками, но на самом деле она применяется на практике и довольно удобна при программировании ряда задач. Одно широко распространенное применение — содействие пользователю в управлении ходом программы. Предположим, например, что вы пишите программу обработки счетов, которая должна предусматривать дополнительную плату раз в три месяца. Для этого нужно только вычислить остаток от деления номера месяца на 3 (т. е. month % 3), проверить, равен ли результат 0, и, если равен, добавить к счету величину дополнительной платы. После того как вы познакомились с «оператором if», вы сможете лучше представить себе, как все это работает.

Приведем пример программы, использующей операцию %.

// X13.C

// секунды в минуты

// перевод секунды в часы, в минуты и секунды

#include <stdio.h>

#define SM 60

void main(void) // пример операции %

{

int sec,min,har,lefts,leftm;

sec = 1;

while (sec > 0)

{

printf("укажите количество секунд ");

printf("(0 - конец работы) > ");

scanf("%d",&sec);

min = sec/SM; //число минут

lefts = sec % SM; // остаток секунд

har = min/SM; // число часов

leftm = min % SM; //остаток минут

printf("%d секунд составляют > ",sec);

printf("%d час(а/ов) ",har);

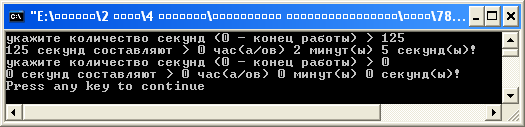
printf("%d минут(ы) ",leftm);

printf("%d секунд(ы)!\n",lefts);

}

}

Вот результат ее работы:



Недостатком этой диалоговой программы является то, что она обрабатывает только одну входную величину. Сумеете ли вы сами изменить программу так, чтобы она предлагала вам вводить новые значения? Мы вернемся к этой задаче в разделе вопросов, но, если вы найдете свое собственное решение, мы будем очень рады.

### Операции увеличения и уменьшения: ++.

Операция увеличения осуществляет следующее простое действие: она увеличивает значение своего операнда на единицу. Существуют две возможности использования данной операции: первая, когда символы ++ находятся слева от переменной (операнда), — «префиксная» форма, и вторая, когда символы ++ стоят справа от переменной, — «постфиксная» форма. Эти две формы указанной операции различаются между собой только тем, в какой момент осуществляется увеличение операнда. Сначала мы обсудим сходство указанных двух форм, а затем вернемся к различиям. Короткий пример, приведенный ниже, показывает, как выполняется данная операция.

// X14.C

// выполнение сложения

#include <stdio.h>

void main(void) // увеличение: префиксная и постфиксная формы

{

int ultra = 0, super = 0;

while( super < 6)

{

super ++;

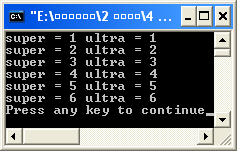
++ ultra;

printf("super = %d ultra = %d\n",super,ultra);

}

}

Результаты работы программы «выполнение сложения» выглядят следующим образом:



Вот это да! Мы досчитали до 5! Дважды! Одновременно! (Если вы захотите считать дальше, вам необходимо будет только изменить параметр, определяющий верхний предел счета в операторе while).

Признаемся, что мы могли бы получить тот же результат, заменив два оператора увеличения следующими операторами присваивания:

super = super + 1;

ultra = ultra + 1;

Данные операторы выглядят достаточно простыми. В связи с этим возникает вопрос: зачем нужен еще один дополнительный оператор, не говоря уже о двух, да еще в сокращенной форме?

Во-первых, компактная форма делает ваши программы более изящными и легкими для понимания. Эти операции придают им блеск, что приятно само по себе.

Например, мы можем переписать часть программы «размер обуви2» следующим образом:

// X15.C

#include <stdio.h>

#define OFFSET 7.64

#define SCALE 0.325

void main(void)

// 1-я модификация размера обуви с использованием автоувеличения

{

double size,foot;

size = 3.0;

while(size < 18.5)

{

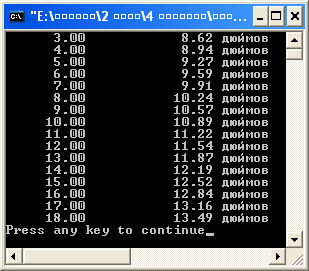
foot = SCALE \* size + OFFSET;

printf("%10.2f %15.2f дюймов\n",size,foot);

++ size;

}

}



При этом способе мы еще не воспользовались всеми преимуществами операции увеличения. Мы можем сократить данный фрагмент так:

// X16.C

#include <stdio.h>

#define OFFSET 7.64

#define SCALE 0.325

void main(void)

// 2-я модификация размера обуви с использованием автоувеличения

{

double size,foot;

size = 2.0;

while(++ size < 18.5)

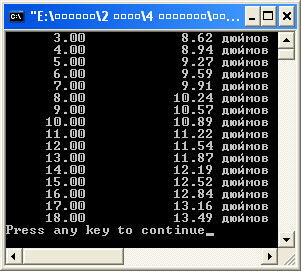
{

foot = SCALE \* size + OFFSET;

printf("%10.2f %15.2f дюймов\n",size,foot);

}

}



Здесь мы объединили в одном выражении операцию увеличения значения переменной на 1 и проверку истинности условия в операторе while. Подобного типа конструкция настолько часто встречается в языке Си, что заслуживает более подробного рассмотрения.

Во-первых, как она работает? Очень просто: значение переменной size увеличивается на единицу, а затем сравнивается с 18.5. Если оно меньше, то выполняются операторы, заключенные в фигурные скобки. После этого переменная size увеличивается на единицу еще раз и т. д. Данный цикл повторяется до тех пор, пока значение переменной size не станет слишком большим. Мы изменили начальное значение переменной size с 3.0 на 2.0, чтобы скомпенсировать увеличение переменной size перед ее первоначальным использованием для вычисления переменной foot.

_06_02

Во-вторых, чем нас привлекает этот способ? Своей компактностью. Кроме того (что еще более важно), с его помощью можно объединить в одном выражении два процесса, управляющие циклом. Первый процесс — это проверка: можно продолжать или нет? В данном случае проверяется, меньше ли размер обуви 18.5. Второй процесс заключается в изменении переменной, значение которой проверяется, — в данном случае размер обуви увеличивается на 1. Предположим, мы забыли изменить ее значение. Тогда переменная size всегда будет меньше 18.5, и выполнение цикла никогда не закончится. При выполнении программы компьютер, «пойманный в бесконечный цикл», будет выводить на печать одну за другой идентичные строки. В конце концов, вы можете потерять интерес, ожидая результатов, и должны будете каким-то образом прекратить выполнение программы. Наличие проверки и изменения параметра цикла в одном выражении помогает программистам не забывать вводить в программу коррекцию параметра цикла.

Дополнительное преимущество использования операции увеличения заключается в том, что обычно в результате компиляции получается несколько более эффективный объектный код, поскольку она идентична соответствующей машинной команде.

И, наконец, эти операции имеют еще одну особенность, которую можно использовать в ряде затруднительных ситуаций. Чтобы узнать, что это за особенность, попробуйте выполнить следующую программу:

// X17.C

#include <stdio.h>

void main(void) // автоизменения ++

{

int a = 1, b = 1;

int aplus,plusb;

aplus = a ++; // постфиксная форма

// a изменяется ПОСЛЕ того, как его значение используется

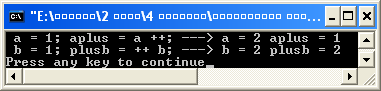
plusb = ++ b; // префиксная форма

// b изменяется ПЕРЕД тем, как ее значение используется

printf(" a = 1; aplus = a ++; ---> a = %d aplus = %d\n",a,aplus);

printf(" b = 1; plusb = ++ b; ---> b = %d plusb = %d\n",b,plusb);

}



Если вы все сделали правильно, то, насколько мы помним, в качестве результата вы должны получить следующие строки:

a aplus b plusb

2 1 2 2

Как и предполагалось, значения обеих переменных, а и b, увеличились на 1. Однако переменной aplus значение а было присвоено перед изменением а, в то время как переменной plusb значение b было присвоено после изменения b. В этом и заключается разница между префиксной и постфиксной формами.

aplus = а++ — постфикс: переменная а изменяется после того, как ее значение используется

plusb = ++b — префикс: переменная b изменяется перед тем, как ее значение используется

06_03

В тех случаях, когда одна из этих операций увеличения используется сама по себе, как, например, в операторе ego++, не имеет значения, какой формой вы пользуетесь. Выбор приобретает смысл, когда операция и ее операнд являются частью некоторого «большего» выражения, как, например, в операторах присваивания, которые мы только что рассматривали. В подобной ситуации необходимо иметь представление о результате, который вам хотелось бы получить. Напомним, например, следующий оператор:

while(++size < 18.5)

При его использовании мы получили таблицу перевода вплоть до размера 18. Но, если бы мы вместо этого записали операцию увеличения в виде size++, в таблицу попал бы и размер 19, поскольку значение переменной size увеличивалось бы после сравнения, а не до него.

Конечно, вы могли бы использовать менее красивый способ — оператор присваивания

size = size + 1;

но тогда никто не поверит вам, что вы умеете по-настоящему программировать на языке Си.

Мы думаем, что при чтении теории вы уже обратили внимание на приведенные примеры использования операций увеличения. Как вы думаете, могли ли мы всегда пользоваться любой из них или же внешние условия диктовали нам конкретный выбор? Говоря о примерах, нам необходимо привести еще один.

Спят ли когда-нибудь компьютеры? Конечно, спят, но они обычно не рассказывают нам об этом. Программа, приведенная ниже, показывает, что происходит в действительности.

// X18.C

#include <stdio.h>

#define MAX 40

void main(void) // пример на засыпание IBM/PC

{

int count = 0;

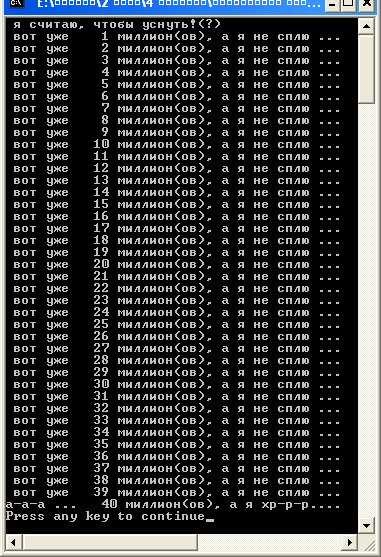
printf(" я считаю, чтобы уснуть!(?)\n");

while(++ count < MAX)

printf(" вот уже %4d миллион(ов), а я не сплю ...\n",count);

printf("a-a-a ... %4d миллион(ов), а я хр-р-р....\n",count);

}



Попробуйте выполнить ее и посмотрите, работает ли она так, как должна по вашему мнению. Конечно, значение константы МАХ для вашего компьютера можно взять другим. (Кстати, что произойдет при замене префиксной формы операции увеличения постфиксной формой?)

### Операция уменьшения: --.

Каждой операции увеличения соответствует некоторая операция уменьшения, при этом вместо символов ++ мы используем --.

--count; /\* префиксная форма операции уменьшения \*/

count--; /\* постфиксная форма операции уменьшения \*/

Ниже приводится пример, иллюстрирующий, как машины могут быть законченными лириками.

// X19.C

#include <stdio.h>

#define MAX 10

void main(void) // автоуменьшение

{

int count = MAX + 1;

while(-- count > 0)

{

printf(" Видишь!! Уже осталось только %d бутылок!\n",count);

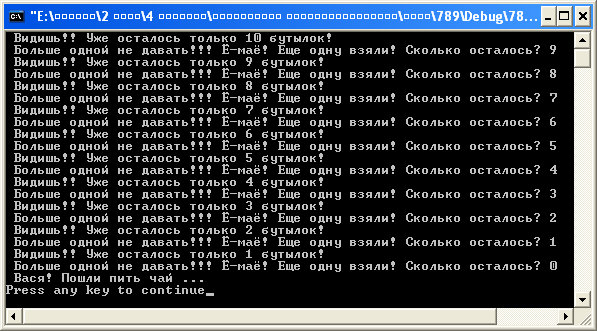
printf(" Больше одной не давать!!!");

printf(" Ё-маё! Еще одну взяли! Сколько осталось? %d\n",count-1);

}

printf(" Вася! Пошли пить чай ...\n");

}



Пример и результат не совпадают.

Начальные результаты выглядят так:

100 бутылок пива на полке, 100 бутылок пива!

Сними одну и пусти ее по кругу,

99 бутылок пива!

99 бутылок пива на полке, 99 бутылок пива!

Сними одну и пусти ее по кругу,

98 бутылок пива!

Постепенно количество бутылок сходит на нет, и программа завершает свою работу следующим образом:

1 бутылок пива на полке, 1 бутылок пива!

Сними одну и пусти ее по кругу,

0 бутылок пива!

По-видимому, у нашего законченного лирика имеются трудности со склонением существительных с количественными числительными, но это можно устранить, используя условные операторы.

Необходимо заметить, что смысл операции > словами выражается как «больше». Так же как и операция <, она является «операцией отношения». Подробнее операции отношения мы рассмотрим позднее.

### Старшинство операций.

В соответствии с принятым в языке Си порядком вычислений операции увеличения и уменьшения имеют очень высокий уровень старшинства; только круглые скобки обладают более высоким приоритетом. Поэтому выражение х\*у++ означает (х)\*(у++), а не (х\*у)++, что очень удобно, поскольку последнее выражение смысла не имеет. (Операции увеличения и уменьшения применяются к переменной, в то время как произведение х\*у само по себе не является переменной в отличие от сомножителей).

Не путайте только старшинство этих операций с порядком вычислений. Предположим, у нас есть последовательность операторов

y = 2; n = 3; nextnum = (у+n++)\*6;

Какое значение примет переменная nextnum? Подставляя в выражение соответствующие значения, получаем

nextnum = (2 + 3)\*6= 5\*6 = 30

Только после того как выражение вычислено, значение переменной n увеличивается до 4. Старшинство операций говорит, что операция ++ имеет отношение только к n; кроме того, оно указывает, когда значение переменной n используется при вычислении выражения, но момент изменения значения n определяется семантикой данной операции.

Не будьте слишком умными. Вы можете попасть в глупое положение, если попытаетесь использовать операцию увеличения в неподходящих случаях. Например, вы могли бы захотеть улучшить нашу программу вывода на печать целых чисел и их квадратов, заменив имеющийся там цикл while следующей конструкцией:

while(num < 21)

{

printf(%10d %10d\n", num\*num++ );}

Эта модификация выглядит разумной. Мы печатаем число num, умножаем его само на себя, чтобы получить его квадрат, а затем увеличиваем значение num на единицу. На некоторых машинах эта программа даже может работать. Но не на всех. Проблема состоит в том, что при выполнении функции printf(), когда определяются печатаемые значения, вычисление последнего аргумента может выполниться сначала, и приращение переменной n произойдет до того, как будет определен первый аргумент. Поэтому, вместо, скажем, такой строки

5 25

будет напечатано

6 25

Правила языка Си предоставляют компилятору возможность выбрать, какой аргумент функции вычислять первым; это повышает эффективность работы компилятора, но может приводить и к некоторым проблемам, если операция увеличения выполняется над одним из аргументов функции.

Другим возможным источником неприятностей служит оператор вида

ans = num/2+5\*(l+num++);

Опять проблема заключается в том, что компилятор может выполнять действия не в том порядке, который вы ожидали. Вы можете считать, например, что сначала он определит значение num/2, а затем перейдет к другой части выражения. Но компилятор может вычислить сначала последний член, увеличить переменную num, а затем использовать новое значение при нахождении num/2. Никакой гарантии в этом случае не существует.

Избежать этих трудностей достаточно просто:

1. Не применяйте операции увеличения или уменьшения к переменной, присутствующей в более чем одном аргументе функции.
2. Не применяйте операции увеличения или уменьшения к переменной, которая входит в выражение более одного раза.

**Приоритет операций C/C++**

|  |  |  |
| --- | --- | --- |
| Значение | Операции | Порядок вычисления |
| Вызов функции | () | Слева направо |
| Индекс массива | [] |  |
| Выделение элемента структуры, объединения, класса | **.** |  |
| Выделение элемента (структуры, объединения, класса), адресуемого указателем | > |  |
| Указатель на метод (объекты) | .\* |  |
| Указатель на метод (указатели) | .>\* |  |
| Постфиксный инкремент | ++ | Справа налево |
| Постфиксный декремент | -- |  |
| Операция базы (не для 32-разрядной компиляции) | :> |  |
| Доступ к области действия | :: |  |
| Логическое отрицание NOT | ! | Справа налево |
| Поразрядное отрицание | - |  |
| Унарный минус | - |  |
| Унарный плюс | + |  |
| Префиксный инкремент | ++ |  |
| Префиксный декремент | -- |  |
| Определение адреса | & |  |
| Обращение по адресу | \* |  |
| Размер объекта в байтах | sizeof |  |
| Приведение типа | (тип) |  |
| Динамическое выделение памяти | new |  |
| Динамическое освобождение памяти | delete |  |
| Умножение | \* | Слева направо |
| Деление | / |  |
| Определение остатка | % |  |
| Сложение | + | Слева направо |
| Вычитание | - |  |
| Сдвиг влево | << | Слева направо |
| Сдвиг вправо | >> |  |
| Меньше, чем | < | Слева направо |
| Больше, чем | > |  |
| Меньше или равно | <= | Слева направо |
| Больше или равно | >= |  |
| Равно | == | Слева направо |
| Не равно | != |  |
| Поразрядное И | & | Слева направо |
| Поразрядное исключающее ИЛИ | ^ |  |
| Поразрядное ИЛИ | | |  |
| Логическое И | && | Слева направо |
| Логическое ИЛИ | || |  |
| Условна операция | ?: | Справа налево |
| Присваивание | = | Справа налево |
| Умножить и присвоить | \*= |  |
| Разделить и присвоить | /= |  |
| Разделить по модулю и присвоить | %= |  |
| Сложить и присвоить | += |  |
| Вычесть и присвоить | -= |  |
| Сдвинуть влево и присвоить | <<= |  |
| Сдвинуть вправо и присвоить | >>= |  |
| Поразрядное И и присвоить | &= |  |
| Поразрядное исключающее ИЛИ и присвоить | ^= |  |
| Поразрядное ИЛИ и присвоить | |= |  |
| Запятая | , | Слева направо |

## Выражения и операторы.

Мы использовали термины «выражение» и «оператор» на протяжении всех предыдущих примеров; теперь настало время изучить их более подробно. Операторы служат основными элементами, из которых строится программа на языке Си; большинство же операторов состоит из выражений. Исходя из этого, вначале разумно рассмотреть выражения, что мы и сделаем.

### Выражения.

Выражение представляет собой объединение операций и операндов. (Напомним, что операндом называется то, над чем выполняется операция). Простейшее выражение состоит из одного операнда; отталкиваясь от него, вы можете строить более сложные конструкции. Выражение в C++ представляет собой последовательность операторов, операндов и знаков пунктуации, воспринимаемую компилятором как руководство к определенному действию над данными. Всякое выражение, за которым идет точка с запятой, образует предложение или инструкцию языка:

выражение;

На практике возможны также случаи, когда сами операнды в выражениях могут быть представлены выражениями. Приведем несколько выражений:

4;

-6;

4+21;

a\*(b + c/d)/20;

q = 5\*2

х = ++ q % 3

q > 3

Нетрудно заметить, что операнды могут быть константами, переменными или их сочетаниями. Некоторые выражения состоят из меньших выражений, которые мы можем назвать подвыражениями. Например, c/d — это подвыражение в нашем четвертом примере.

Важным свойством языка Си является то, что каждое выражение в С имеет значение. Чтобы определить это значение, мы выполняем операции в порядке, определяемом уровнями старшинства. Значения первых нескольких выражений очевидны, но что можно сказать относительно выражений со знаком =? Они просто имеют те же значения, что и переменная, стоящая слева от знака =. Эта переменная получает его в результате вычисления выражения, стоящего справа от знака. А выражение q > 0? Подобное выражение, связанное с операцией отношения, имеет значение 1, если оно истинно, и 0, если оно ложно. Приведем несколько выражений и их значения.

**Выражение Значение**

-4+6 2

с = 3 + 8 11

5 > 3 1

6 + (с = 3 + 8) 17

Последний пример выглядит довольно странно! Но он полностью соответствует правилам языка Си, поскольку данное выражение представляет собой сумму двух подвыражений, каждое из которых имеет значение.

### Операторы.

Операторы служат основными строительными блоками программы. Программа состоит из последовательности операторов с добавлением небольшого количества знаков пунктуации. Оператор является законченной инструкцией для компьютера. В языке Си указанием на наличие оператора служит символ «точка с запятой», стоящий в конце него. Поэтому

legs = 4

это всего лишь выражение (которое может быть частью большего выражения), но

legs = 4;

является оператором.

Что делает инструкцию законченной? Она должна выполнять некоторое действие полностью. Выражение

2+2

не является законченной инструкцией, а служит указанием компьютеру сложить 2 и 2, но не говорит, что делать с результатом.

kids =2+2;

служит указанием компилятору (а затем компьютеру) поместить результат (4) в ячейку памяти, помеченную именем kids. После занесения в память числа 4 компьютер может приступить к выполнению следующих действий.

До сих пор мы познакомились с четырьмя типами операторов. Ниже приводится краткий пример, в котором используются все четыре типа.

// сумма

#include <stdio.h>

void main() //нахождение суммы первых 20 целых чисел

{

int count, sum; // оператор описания

count = 0; // оператор присваивания

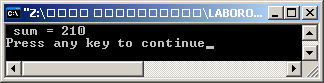
sum = 0; // то же самое

while (count++ < 20) // while

sum = sum + count; // оператор

printf(" sum = %d\n", sum); // вызов функции

}



Давайте обсудим этот пример. К данному моменту оператор описания должен быть вам уже довольно хорошо знаком. Тем не менее, мы напомним, что с его помощью определяются имена и типы переменных и им отводятся ячейки памяти.

Оператор присваивания — это основная рабочая сила большинства программ: с его помощью переменной присваивается некоторое значение. Он состоит из имени переменной, за которым следует знак операции присваивания (=), а затем выражение, оканчивающееся символом «точка с запятой». Отметим, что оператор while включает в себя оператор присваивания.

Оператор вызова функции приводит к выполнению последовательности операторов, образующих тело функции. В нашем примере функция printf () вызывается для того, чтобы вывести на печать результаты работы программы.

Оператор while состоит из трех различных частей: это ключевое слово while, затем проверяемое условие, заключенное в круглые скобки, и, наконец, оператор, который выполняется в том случае, если условие истинно. Тело цикла состоит из одного оператора. Он может быть простым, как в данном примере (причем в этом случае не требуется заключать его в фигурные скобки), или составным, как в некоторых предыдущих примерах (тогда фигурные скобки абсолютно необходимы). Вы сможете прочесть о составных операторах чуть позже.

**_06_04**

Оператор while принадлежит к классу операторов, иногда называемых «структурированными операторами», поскольку они обладают структурой более сложной, чем структура простого оператора присваивания. Дальше мы познакомимся с другими типами структурированных операторов.

### Составные операторы (блоки).

«Составной оператор» представляет собой два или более операторов, объединенных с помощью фигурных скобок; он называется также «блоком». В нашей программе размер обуви2 мы использовали такой оператор, чтобы иметь возможность включить в оператор while несколько простых операторов. Сравните между собой фрагменты программы:

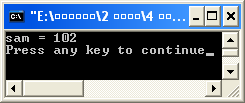
/\* фрагмент1 \*/

index = 0;

while (index++ < 10)

sam = 10\* index+ 2;

printf("sam = %d\n", sam);



/\* фрагмент2 \*/

index = 0;

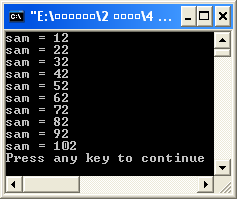
while(index++ < 10)

{

sam = 10\*index + 2;

printf("sam = %d\n", sam);

}



В фрагменте 1 в цикл while включен только оператор присваивания. (При отсутствии фигурных скобок область действия оператора while распространяется от ключевого слова while до следующего символа «точка с запятой».) Печать данных будет произведена только один раз — после завершения цикла.

В фрагменте 2 наличие фигурных скобок гарантирует, что оба оператора являются частью цикла while, и печать результатов будет производиться на каждом шаге работы цикла. Весь составной оператор рассматривается как один оператор, являющийся составной частью оператора while.

_06_05

**Советы по стилю программирования.** Давайте опять посмотрим на фрагменты, содержащие цикл while, и обратим внимание на то, как мы использовали отступы от поля в строках для выделения тела циклов while. Для компилятора отступы в строке не имеют никакого значения; для решения вопроса о том, как интерпретировать наши инструкции, он использует фигурные скобки и свое «знание» правил формирования структуры цикла while. Отступы в строках предназначены для нас, чтобы с первого взгляда можно было понять, как устроена программа.

Ранее мы показали вам один популярный способ расстановки фигурных скобок, служащий для указания блока или составного оператора. Другой, тоже довольно распространенный способ выглядит следующим образом:

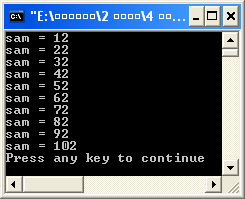
while(index++ < 10)

{

sam = 10\*index + 2;

printf("sam = %d\n", sam);

}



Этот способ акцентирует внимание на том, что данные операторы образуют блок, в то время как способ, обсуждавшийся выше, указывает на принадлежность блока оператору while. Заметим снова, что поскольку дело касается компилятора, обе формы являются абсолютно идентичными.

Подводя итоги, скажем: применяйте отступы от поля в строках, чтобы сделать структуру программы наглядной.

**Резюме: выражения и операторы.**

***Выражения****.* Выражение состоит из операций и операндов. Примерами простейших выражений являются константы или переменные (операция отсутствует), такие, как 22 или beebop. Более сложные выражения — это 55 + 22 и vap = 2\*(vip + (uyf = 4)).

***Операторы***. Оператор служит командой компьютеру. Операторы бывают простыми и составными: Простые операторы оканчиваются символом «точка с запятой». Примеры:

1. Операторы описания: int toes;
2. Операторы присваивания: toes = 12;
3. Операторы вызова функции: printf(" %d\n", toes);
4. Управляющие операторы: while (toes < 20)

toes = toes + 2;

1. Пустой оператор: ;

Составные операторы, или блоки, состоят из одного или более операторов (которые в свою очередь сами могут быть составными), заключенных в фигурные скобки. Оператор while, приведенный ниже, содержит, например, составной оператор:

while (years < 100)

{

wisdom = wisdom + 1;

printf(" %d %d\n", years, wisdom);

}

## Преобразование типов.

В операторах и выражениях, вообще говоря, должны использоваться переменные и константы только одного типа. Если все же вы смешиваете типы в одном выражении, то компилятор с языка Си не считает программу неправильной, как это произошло бы при программировании на Паскале. Вместо этого компилятор использует набор правил для автоматического преобразования типов. Это очень удобно, но может оказаться и опасным, особенно если вы допустили смешение типов нечаянно. (Например, программа lint, работающая в операционной системе UNIX, проверяет несоответствие типов). Нам представляется разумным привести несколько основных правил, касающихся преобразования типов:

1. Если операция выполняется над данными двух различных типов, обе величины приводятся к «высшему» из двух типов. Этот процесс называется «повышением» типа.
2. Последовательность имен типов, упорядоченных от «высшего» типа к «низшему», выглядит так: double, float, long, int, short и char. Применение ключевого слова unsigned повышает ранг соответствующего типа данных со знаком.
3. В операторе присваивания конечный результат вычисления выражения в правой части приводится к типу переменной, которой должно быть присвоено это значение. Данный процесс может привести к «повышению» типа, как описано выше, или к «понижению», при котором величина приводится к типу данных, имеющему более низкий приоритет.

«Повышение» типа обычно происходит гладко, в то время как «понижение» может привести к затруднениям. Причина этого проста: все число целиком может не поместиться в элементе данных низшего типа. Переменная типа char может иметь целое значение 101, но не 22334.

Пример, приведенный ниже, иллюстрирует применение этих правил.

// преобразования

#include <stdio.h>

void main()

{

char ch;

int i;

float fl;

fl = i = ch = ' A'; // строка 8

printf("ch = %с, i = %d, fl = %2.2f\n", ch, i, fl);

ch = ch + 1; // строка 10

i = fl + 2\*ch; // строка 11

fl = 2.0\*ch + i; // строка 12

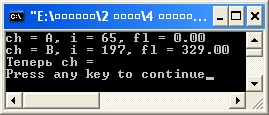
printf("ch = %c, i = %d, fl = %2.2f\n", ch, i, fl);

ch=2.0E30; // строка 14

printf("Теперь ch = %c \n", ch);

}

Выполнив программу «преобразования», получим следующие результаты:



Вот что происходит в программе.

Строки 8 и 9: Величина 'А' присваивается символьной переменной ch. Переменная i получает целое значение, являющееся преобразованием символа ' А' в целое число, т. е. ' 65'. И, наконец, переменная fl получает значение 65.00, являющееся преобразованием числа 65 в число с плавающей точкой.

Строки 10 и 13: Значение символьной переменной 'А' преобразуется в целое число 65, к которому затем добавляется 1. После этого получившееся в результате число 66 преобразуется в код символа В и помещается в переменную ch.

Строки 11 и 13: При умножении на 2 значение переменной ch преобразуется в целое число (66). При сложении с величиной переменной fl получившееся в результате число (132) преобразуется в число с плавающей точкой. Результат (197.00) преобразуется в число целого типа и присваивается переменной i.

Строки 12 и 13: Перед умножением на 2.0 значение переменной ch('B') преобразуется в число с плавающей точкой. Перед выполнением сложения величина переменной i(197) преобразуется в число с плавающей точкой, а результат операции (329.00) присваивается переменной fl.

Строки 14 и 15: Здесь производится попытка осуществить преобразование типов в порядке убывания старшинства — переменная ch полагается равной сравнительно большому числу. Результаты оказываются неутешительными. Независимо от переполнения и усечения, которые имеют место, в итоге на нашей системе мы получили код, соответствующий какому-то непечатаемому знаку.

На самом деле существует еще один вид преобразования типов. Для сохранения точности вычислений при арифметических операциях все величины типа float преобразуются в данные типа double. Это существенно уменьшает ошибку округления. Конечный результат, естественно, преобразуется обратно в число типа float, если это диктуется соответствующим оператором описания. Вам нет необходимости заботиться о выполнении подобных преобразований, но должно быть приятно осознавать, что компилятор стоит на страже ваших интересов.

## Операция приведения.

Самое лучшее — это вообще избегать преобразования типов особенно в порядке убывания ранга. Но иногда оказывается удобным применять такие преобразования при условии, что вы ясно представляете себе смысл выполняемых действий. Преобразования типов, которые мы обсуждали до сих пор, выполнялись автоматически. Кроме того, существует возможность точно указывать тип данных, к которому необходимо привести некоторую величину. Этот способ называется «приведением» типов и используется следующим образом: перед данной величиной в круглых скобках записывается имя требуемого типа. Скобки и имя типа вместе образуют «операцию приведения».

Рассмотрим две нижеследующие строки, где mice — это переменная типа int. Вторая строка содержит две операции приведения к типу int.

mice = 1.6 + 1.7;

mice = (int)1.6 + (int)1.7;

В первом примере используется автоматическое преобразование типов. Сначала числа 1.6 и 1.7 складываются — результат равен 3.3. Затем путем отбрасывания дробной части полученное число преобразуется в 3 для согласования с типом int переменной mice. Во втором примере 1.6 и 1.7 преобразуются в целые числа 1, так что переменной mice присваивается значение, равное 1+1, или 2.

Вообще говоря, вы не должны смешивать типы; вот почему в некоторых языках это запрещено. Но бывают ситуации, когда это оказывается полезным. Философия языка Си заключается в том, чтобы не устанавливать барьеров на вашем пути, но при этом возложить на вас всю ответственность за злоупотребление предоставленной свободой.

Ниже перечислены операции, которые мы уже обсудили.

**Операция присваивания.**

= Присваивает величину справа от знака переменной слева от него

**Арифметические операции.**

+ Прибавляет величину справа от знака к величине слева от него

- Вычитает величину справа от знака из величины слева от него

- Унарная операция; изменяет знак величины справа от знака

\* Умножает величину справа от знака на величину слева от него

/ Делит величину слева от знака на величину справа от него. Результат усекается, если оба

операнда целые числа

% Дает остаток при делении величины слева от знака на величину справа от него (только

для целых чисел)

++ Прибавляет 1 к значению переменной слева от знака (префиксная форма) или к значению

переменной справа от знака (постфиксная форма)

-- Аналогичная операции ++, но вычитает 1

**Дополнительные операции.**

Sizeof Дает размер операнда, стоящего справа, в байтах. Операнд может быть спецификацией

типа, заключенного в круглые скобки, как, например, sizeof (float), или именем конкретной

переменной, массива и т. п., например sizeof too.

(тип) Операция приведения: приводит величину, стоящую справа, к типу, определяемому

ключевым словом (или словами) в скобках. Например, (float) 9 преобразует целое число 9 в

число с плавающей точкой 9.0.

**Пример программы.** Ниже приведена программа, которая может оказаться полезной тем, кто занимается бегом, и которая иллюстрирует некоторые положения данной темы. Она выглядит довольно длинной, но все вычисления в ней выполняются шестью операторами, помещенными в конце. Большей частью программа занимается организацией диалога между машиной и пользователем. Мы ввели в программу достаточно большое число комментариев, чтобы сделать ее почти самодокументируемой. Просмотрите ее, а затем мы объясним некоторые ее моменты.

// бег

#include <stdio.h>

void main()

{

float distk, distm; // дистанция в км и милях

float rate; // средняя скорость в милях в час

int min, sec; // время бега в минутах и секундах

int time; // время бега в секундах

float mtime; // время пробега одной мили в секундах /

int mmin, msec; // время пробега одной мили в минутах и секундах

printf("Эта программа пересчитывает ваше время пробега дистанции, выраженной в км,\n");

printf("во время, требуемое для пробега одной мили, и вашу среднюю\n");

printf(" скорость в милях в час,\n");

printf("Укажите, пожалуйста, дистанцию в километрах.\n");

scanf("%f",&distk);

printf("Введите затем время в минутах и секундах.\n");

printf("Начните с ввода минут.\n");

scanf("%d",&min);

printf("Теперь вводите секунды.\n");

scanf("%d",&sec);

time = SM\*min + sec; // переводит время в секунды

distm = MK\*distk; // переводит километры в мили

rate = distm/time\*SH; // число миль в сек х число сек в час= число миль в час

mtime = (float)time/distm; // время/дистанция — время на милю

mmin = (int)mtime/SM; // находит целое число минут

msec = (int)mtime % SM; // находит остаток в секундах

printf("Вы пробежали %1.2f км (%1.2f мили) за %d мин %d c.\n", distk, distm, min, sec);

printf("Эта скорость соответствует пробегу мили за %d мин", mmin);

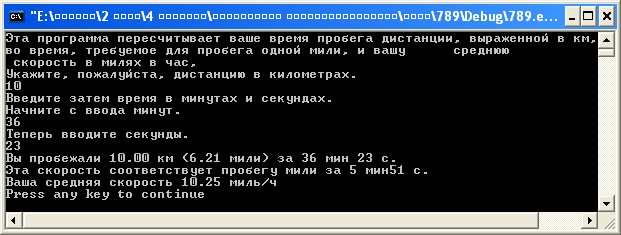
printf("%d c.\n Ваша средняя скорость %l,2f миль/ч\n", msec, rate);

}

Здесь мы применили тот же подход, который использовали в программе секунды в минуты для перевода времени, выраженного в секундах, в минуты и секунды. Кроме того, нам пришлось воспользоваться преобразованиями типов. Почему? Потому что для той части программы, которая занимается переводом секунд в минуты, нам требуются аргументы целого типа, а при преобразовании из метрической системы в мили используются числа с плавающей точкой. Мы применили операцию приведения для того, чтобы сделать эти преобразования явными.

Честно говоря, нашу программу можно было бы написать, используя только автоматическое преобразование типов. Мы так и делали, применяя операцию приведения переменной mtime к типу int, чтобы при вычислении времени все операнды были целого типа. Однако такая версия компилятора работает всего на одной из двух доступных нам систем. Использование операции приведения не только проясняет ваш замысел человеку, знакомящемуся с вашей программой, но и упрощает ее компиляцию.

Вот результат работы данной программы.



**Что вы должны были узнать.**

* Как использовать операции: +, - , \*, /, %, ++, -- , (type)
* Что такое операнд: это — величина, над которой выполняется операция.
* Что такое выражение: совокупность операций и операндов.
* Как вычислять значение выражения: в соответствии с порядком старшинства.
* Как распознать оператор: по символу;
* Виды операторов: операторы описания, присваивания, while, составной.
* Как сформировать составной оператор: заключить последовательность операторов в фигурные скобки { }.
* Как формируется оператор while: while (проверка условия) оператор;
* Как вычисляются выражения со смешанными типами данных: с помощью автоматического преобразования типов.

**Вопросы и ответы.**

**Вопросы.**

1. Предположим, все переменные имеют тип int. Определите значение каждой из нижеследующих переменных:

а. х = (2 + 3)\*6;

б. х = (12 + 6)/2\*3;

в. у = х = (2 + 3)/4;

г. у = 3 + 2\*(х = 7/2),

д. х = (int)3.8 +33;

2. Мы подозреваем, что в программе, приведенной ниже, имеется несколько ошибок. Сумеете ли вы помочь нам их обнаружить?

main()

{

int i = 1,

float n;

printf(" Внимание! Сейчас появится несколько дробей! \n");

while (i < 30)

n = 1/i;

printf("%f", n);

printf(" Вот и все! Конец!\n");

}

3. Ниже приведена первая попытка сделать программу «секунды в минуты» диалоговой. Программа нас не удовлетворяет. Почему? Как ее улучшить?

#define SM 60

main()

{

int sec, min, left;

printf("Эта программа переводит секунды в минуты и секунды.\n");

printf("Укажите число секунд.\n");

printf("Для окончания работы программы необходимо ввести 0.\n");

while (sec < 0){

scanf("%d", &sec);

mm = sec/SM;

left = sec % SM;

printf("%d с это %d мин %d c.\n", sec, min, left);

printff"Введите следующее значение.\n");

printf("Дo свидания!\n");

}

**Ответы.**

а. 30.

б. 27(а не 3). Результат 3 можно получить в случае (12 + 6)/(2\*3).

в. х = 1, у = 1 (деление целых чисел).

г. х = 3 (деление целых чисел) и у = 9.

д. х = 6, так как (int)3.8 = 3; 3 + 3.3 = 6.3, это число будет преобразовано в число 6, поскольку х имеет тип int

2. Строка 3 должна оканчиваться точкой с запятой, а не запятой.

Строка 7: оператор while представляет собой бесконечный цикл, потому что величина переменной i остается равной 1 и всегда будет меньше 30. По всей видимости, мы собирались написать while(i++ < 30).

Строки 7—9: отступы в строках подразумевают, по-видимому, что из операторов, расположенных в строках 8 и 9, мы намеревались сформировать блок, но отсутствие фигурных скобок означает, что цикл while включает в себя только оператор, расположенный на строке 8; поэтому фигурные скобки должны быть обязательно добавлены.

Строка 8: поскольку 1 и i — оба целого типа, результат деления будет равен 1 при i, равном 1, и 0 — для всех больших значений. Необходимо писать так х = 1.0/i; перед делением значение переменной i будет приведено к типу данных с плавающей точкой и будет получен ненулевой результат.

Строка 9: мы опустили символ «новая строка» в управляющей строке; это приведет к тому, что числа будут печататься на одной строке, если так допускается устройством вывода.

3. Основная трудность лежит в согласовании между оператором, выполняющим проверку (величина переменной sec больше 0 или нет?), и оператором scanf(), осуществляющим ввод значения переменной sec. В частности, когда проверка выполняется первый раз, переменная sec в программе еще не получает своего значения, и поэтому сравнение будет производиться с некоторой случайной величиной (мусором»), которая может оказаться в соответствующей ячейке памяти. Одно решение, хотя и некрасивое, заключается в инициализации переменной sec, скажем величиной 1, в результате чего первый раз сравнение выполнится. Но здесь обнаруживается вторая проблема. Когда при окончании работы мы набираем величину 0, чтобы остановить программу, оказывается, что значение переменной sec проверяется только после завершения шага цикла и происходит вывод на печать результатов для 0 секунд. На самом деле нам хотелось бы, чтобы оператор scanf() выполнялся перед тем, как осуществляется проверка в операторе while. Этого можно достичь путем следующей модификации средней части программы:

scanf("%d," &sec);

while (sec > 0){

min = sec/SM;

left = sec % SM;

printf("%d с это %d мин %d c.\n", sec, min, left);

printf(" Введите следующее значение.\n");

scanf("%d", &sec);

}

В первый раз ввод указанной величины в программу осуществляется функцией scanf(), помещенной перед циклом, а ввод каждой последующей величины будет выполняться функцией scanf в конце цикла (и, следовательно, как раз перед тем, как начнется выполнение очередного шага цикла). Этот подход является общим способом решения проблем подобного сорта.

**Упражнения.**

Ниже приводятся задачи, решения которых мы не даем. Чтобы узнать, работает ли ваша программа, необходимо выполнить ее на вашей машине.

1. Измените нашу программу «сумма» так, чтобы она определяла сумму первых 20 чисел. (Если хотите, можете считать, что эта программа вычисляет, сколько денег вы получите за 20 дней, если в первый день вы получите 1 долл., во второй — 2, в третий — 3 и т. д.). Модифицируйте потом свою программу таким образом, чтобы вы могли в диалоговом режиме указать ей, до какого дня следует вести расчет, т. е. замените константу 20 переменной, значение которой присваивается в результате операции ввода.
2. А теперь модифицируйте свою программу так, чтобы она вычисляла сумму квадратов целых чисел. (Или, если вам так больше нравится, сколько вы всего получите денег, если в первый день вам заплатят 1 долл., во второй — 4, в третий — 9 и т. д. Это гораздо более прибыльное дело!). Учтите, что в языке Си нет функции возведения в квадрат, но вы можете использовать тот факт, что квадрат числа n — это просто n\*n.
3. Измените свою программу так, чтобы после завершения вычислений она запрашивала у вас новое значение переменной и выполняла вычисления повторно. Окончание работы программы должно происходить при вводе 0. (Указание: используйте такую конструкцию, как цикл в цикле. См. также вопрос 3 и решение к нему).