[11. Циклы и другие управляющие средства.](#_Toc515770133)

[11.1. Оператор цикла.](#_Toc515770134)

[11.1.1. Цикл while.](#_Toc515770135)

[11.1.1.1. Завершение цикла while.](#_Toc515770136)

[11.1.1.2. Программа подсчета слов.](#_Toc515770137)

[11.1.1.3. Программа, «рисующая» символами.](#_Toc515770138)

[11.1.2. Цикл do-while.](#_Toc515770139)

[11.1.3. Цикл for.](#_Toc515770140)

[11.1.3.1. Операция «запятая» в цикле for.](#_Toc515770141)

[11.1.3.2. Гибкость конструкции for.](#_Toc515770142)

[11.1.3.3. Философ Зенон и цикл for.](#_Toc515770143)

[11.1.4. Сравнение циклов.](#_Toc515770144)

[11.1.5. Вложенные циклы.](#_Toc515770145)

[11.1.6. Алгоритмы и псевдокод.](#_Toc515770146)

[11.2. Оператор break.](#_Toc515770147)

[11.3. Оператор continue.](#_Toc515770148)

[11.4. Совместное использование операторов break и continue.](#_Toc515770149)

[11.5. Оператор goto.](#_Toc515770150)

[11.5.1. Использование goto.](#_Toc515770151)

[11.6. Оператор exit().](#_Toc515770152)

[11.7. Оператор atexit().](#_Toc515770153)

1. Циклы и другие управляющие средства.

При усложнении решаемых задач ход выполнения программ становится более запутанным. Чтобы иметь возможность управлять процессом выполнения программ и его организацией, вам потребуются структуры и некоторые специальные операторы. Язык Си предоставляет эффективные средства реализации таких требований. Мы уже смогли убедиться в чрезвычайной ценности цикла while в том случае, когда необходимо повторить некоторую операцию несколько раз. В языке Си, кроме того, реализовано еще два типа циклов: цикл for и цикл do ... while. Здесь рассматриваются принципы работы управляющих структур и даются рекомендации, каким образом лучше всего применять каждую из них. Мы обсудим операторы break, continue, goto и операцию «запятая»: все они могут использоваться для управления ходом выполнения программы. Кроме того, мы расскажем вам еще немного о массивах, которые часто используются вместе с циклами.

## Оператор цикла.

В языке С имеется стандартный набор операторов цикла: for, while и do-while (называемый в некоторый других языках высокого уровня циклом repeat-until). Вас может, однако, удивить то, каким способом программа выходит из цикла. В С можно изменить порядок выполнения цикла четырьмя способами. Естественно, все циклы заканчиваются при выполнении заданного проверочного условия. Однако, в С цикл может также закончиться по некоторому заданному условию ошибки при помощи операторов break или exit. Кроме этого, в циклах может быть собственная управляющая логика, изменяемая при помощи оператора break или оператора continue.

Основное различие циклов for и циклов while или do-while заключается в определении числа повторений цикла. Циклы for обычно используются тогда, когда существует заранее определенное число необходимых повторений; циклы while и do-while нужны при "неизвестном" количестве повторений.

### Цикл while.

Так же как и цикл for, в С цикл while является циклом с предусловием. Это означает, что в программе проверка\_условия осуществляется до выполнения оператора или операторов, входящих в тело цикла. Благодаря этому, циклы с предусловием могут либо не выполняться вообще, либо выполняться множество раз. В С синтаксис цикла while следующий:

while(проверка\_условия)

оператор;

В циклах while с несколькими операторами необходимы фигурные скобки:

while(проверка\_условия) {

оператор1;

оператор2;

оператор3;

операторn;

}

В наших примерах в качестве выражений использовались условные выражения, но, вообще говоря, это могут быть выражения произвольного типа. В качестве оператора можно использовать простой оператор с символом «точка с запятой» в конце или составной оператор, заключенный в фигурные скобки. Если выражение истинно (или в общем случае не равно нулю), то оператор, входящий в цикл while, выполняется один раз, а затем выражение проверяется снова. Эта последовательность действий, состоящая из проверки и выполнения оператора, периодически повторяется до тех пор, пока выражение не станет ложным (или в общем случае равным нулю). Каждый такой шаг называется «итерация». Данная структура аналогична структуре оператора if. Основное отличие заключается в том, что в операторе if проверка условия и (возможное) выполнение оператора осуществляется только один раз, а в цикле while эти действия производятся, вообще говоря, неоднократно.

09_01

Обычно управляющие структуры циклов while используются тогда, когда число повторений цикла — неизвестно. В следующей программе на С цикл while используется для определения того, сколько раз переменную ivalue сдвигать вправо. Программа печатает двоичное представление целого числа со знаком.

/\*07WHILE.C

Программа на С, в которой используется цикл while с предусловием

и флагом\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

//#include <iostream.h>

using namespace std;

#define WORD 16

#define ONE\_BYTE 8

main()

{

int ivalue = 256, ibit\_position=1;

unsigned int umask = 1;

/\* Следуицее значение ... \*/

printf("The following value %d,\n",ivalue);

/\* в двоичном виде выглядит так: \*/

printf("in binary form looks like: ") ;

while(ibit\_position <= WORD) {

if ((ivalue >> (WORD - ibit\_position)) & umask)

/\*сдвинуть каждый \*/

printf ("1"); /\*разряд в 0-ю \*/

else /\* позицию и \*/

printf("0"); /\* сравнить с umask k\*/

if(ibit\_position == ONE\_BYTE)

printf (" ");

ibit\_position++;

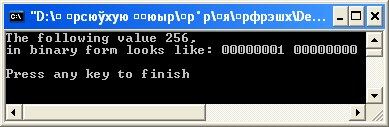
}

printf ("\n\nPress any key to finish\n"); /\* Конец работы \*/

\_getch();

return(0);

}



В начале программы определяются две константы, WORD и ONE\_BYTE, которые можно легко изменить для различных архитектур. Константа WORD используется как флаг, определяющий окончание цикла while. В цикле while переменная ivalue сдвигается, сравнивается с umask, и печатается следующий старший разряд числа. Такой алгоритм позволяет использовать простой оператор printf() для вывода результата.

В следующей программе на С у пользователя запрашиваются имена входного и выходного файлов. Затем в цикле while выполняется считывание и эхо-печать входного файла неопределенного размера.

/\*07DOWHIL.C

Программа на С, в которой цикл while используется для эхо-печати

файла. В этой программе показаны дополнительные способы файлового ввода/вывода\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

//#include <i

ostream.h>

using namespace std;

#define sz\_TERMINATOR 1 /\* sz, признак null-строки \*/

#define MAX\_CHARS 30

main()

{

int c;

FILE \*ifile, \*ofile;

char sziDOS\_file\_name [MAX\_CHARS + sz\_TERMINATOR],

szoDOS\_file\_name[MAX\_CHARS + sz\_TERMINATOR] ;

/\* Укажите имя файла: \*/

fputs("Enter the input file's name: ",stdout);

gets(sziDOS\_file\_name);

if((ifile=fopen(sziDOS\_file\_name,"r")) = NULL);

{

/\* Файл ……… невозможно открыть \*/

printf("\nFile: %s cannot be opened\n",sziDOS\_file\_name);

exit;

}

/\* Укажите имя выходного файла \*/

fputs("Enter the output file's name: ",stdout);

gets(szoDOS\_file\_name);

if((ofile=fopen(szoDOS\_file\_name,"w")) == NULL);

{

/\* Файл ... невозможно открыть \*/

printf("\nFile: %s cannot be opened\n",szoDOS\_file\_name);

exit;

}

while(!feof(ifile)) {

c=fgetc(ifile);

fputc(c,ofile);

}

printf ("\n\nPress any key to finish\n"); /\* Конец работы \*/

\_getch();

return(0);

}

В этом примере цикл while содержит два выполняемых оператора, поэтому необходимы фигурные скобки. Кроме того, программа иллюстрирует использование нескольких операторов файлового ввода/вывода: fgetc() и fputc(), а также feof().

Раньше мы интенсивно пользовались этой формой цикла, сейчас же хотим рассмотреть его работу в случае простой, возможно, даже примитивной программы, угадывающей число.

/\* угадывание числа1 \*/

/\* неэффективный способ угадывания \*/

#inlude <stdio.h>

main()

{

int guess = 1;

char response;

printf(" Задумайте целое число от 1 до 100. Я попробую угадать");

printf(" его.\nОтвечайте d, если моя догадка правильна и");

printf(" \n н, если я ошибаюсь\n");

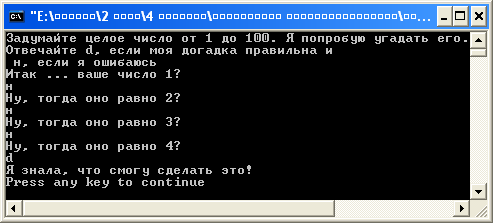
printf("Итак ... ваше число %d?\n" , guess);

while((response = getchar()) != 'd') /\* получение ответа \*/

if (response != '\n') /\* пропуск символа "новая строка" \*/

printf("Hy, тогда оно равно %d?\n", ++guess);

printf("Я знала, что смогу сделать это!\n");}



Обратите внимание на логику программы. Если вы отвечаете “д”, в программе осуществляется выход из цикла и переход на завершающий оператор печати. Программа просит вас отвечать н в случае, если ее догадка неверна, но фактически любой ответ, отличный от символа “д”, приводит к тому, что программа входит в цикл. Однако если введен символ «новая строка», то при данном прохождении тела цикла никаких операций не производится. Получение другого символа приводит к очередной попытке угадывания целого числа. (Что произошло бы в случае использования операции guess++ вместо операции ++guess?)

Строка if (response != '\n') позволяет программе игнорировать поступление постороннего символа «новая строка», когда вы нажмете клавишу [ввод].

В этом случае тело цикла while не нужно заключать в фигурные скобки, поскольку оператор if, хотя он и занимает две строки в программе, рассматривается как один оператор.

Вы, по-видимому, уже заметили, что это довольно «тупая» программа. Она написана правильно и решает поставленную задачу, но делает это крайне неэффективно. Данный пример показывает нам, что правильность написания — не единственный критерий, по которому необходимо оценивать программу. При этом очень важна ее эффективность. Мы вернемся к данной программе несколько позже и попытаемся ее немного улучшить.

Завершение цикла while.

Мы подошли к самому существенному моменту рассмотрения циклов while. При построении цикла while вы должны включить в него какие-то конструкции, изменяющие величину проверяемого выражения так, чтобы в конце концов оно стало ложным. В противном случае выполнение цикла никогда не завершится. Рассмотрим следующий пример:

index = 1;

while ( index < 5)

printf("Доброе утро!\n");

Данный фрагмент программы печатает это радостное сообщение бесконечное число раз, поскольку в цикле отсутствуют конструкции, изменяющие величину переменной index, которой было присвоено значение 1.

index = 1;

while(-- index < 5)

printf(" Как колеблются старые атомы!\n");

И этот фрагмент программы работает ненамного лучше. Значение переменной index в нем изменяется, но в «неправильном» направлении! Единственным утешением здесь служит тот факт, что выполнение данного куска программы в конце концов завершится. Это произойдет, когда величина переменной index станет меньше наименьшего отрицательного числа, допустимого в системе.

Цикл while является «условным» циклом, использующим предусловие (т. е. условие на входе). Он называется условным, потому что выполнение оператора зависит от истинности условия, описываемого с помощью выражения. Действительно ли значение переменной index меньше 5? Является ли последний введенный символ признаком EOF? Подобное выражение задает предусловие, поскольку выполнение этого условия должно быть проверено перед началом выполнения тела цикла. В ситуации, аналогичной приведенной ниже, тело цикла не выполнится ни разу, потому что используемое условие с самого начала является ложным.

index = 3;

while(index++ < 5)

printf(" Желаю хорошо провести день\n");

Измените первую строку на

index = 3;

и вы получите работающую программу.

Программа подсчета слов.

Теперь у нас есть возможности для написания программы подсчета числа слов в тексте. (Она может также подсчитывать символы и строки.) Решающим моментом является разработка способа, с помощью которого программа будет распознавать слова. Мы будем придерживаться сравнительно простого подхода и определим слово как последовательность символов, которая не содержит «пустых» символов. Поэтому «glymxck» и «r2d2» — это слова. Переменная word будет использоваться для хранения указания о том, является ли введенный символ частью данного слова или началом следующего. Появление «пустого символа» (которым может быть пробел, табуляция или «новая строка») служит признаком конца слова. Тогда следующий «непустой» символ будет означать начало нового слова, и мы сможем увеличить значение счетчика слов на 1. Вот эта программа:

#include <stdio.h>

#define YES 1

#define NO 0

main()

{

int ch; /\* введенный символ \*/

long nc = 0L; /\* число символов \*/

int nl = 0; /\* число строк \*/

int nw = 0; /\* число слов \*/

int word = NO; /\* = = YES, если содержимое ch — часть слова \*/

while((ch = getchar()) != EOF)

{

nc++; /\* подсчет символов \*/

if (ch = = ' \n' )

nl ++; /\* подсчет строк \*/

if (ch != ' ' && ch != '\n' && ch != ' \t' && word = = NO)

{

word = YES; /\* начало нового слова \*/

nw++; /\* подсчет слов \*/

}

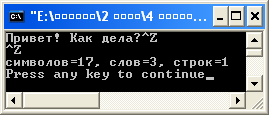
if ((ch = = ' ' || ch = = ' \n' || ch = = ' \t' ) && word == YES)

word = NO; /\* достигнут конец слова \*/

}

printf("символов = %ld, слов = %d, строк = %d\n" , nc, nw, nl);

}



Поскольку существуют три различных «пустых символа», мы должны использовать логические операции для проверки всех трех возможностей. Рассмотрим, например, следующую строку:

if(ch != ' '&& ch != '\n' && ch != ' \t' && word == NO)

В ней говорится: «если содержимое ch — не пробел, и не новая строка, и не табуляция, и не первый символ слова». (Первые три условия эквивалентны проверке, не является ли содержимое ch пустым символом.) Выполнение всех четырех условий служит признаком начала нового слова, и значение переменной nw увеличивается. Если мы в середине слова, то первые три условия оказываются выполненными, но значением переменной word окажется признак YES, и значение переменной nw не увеличивается. Когда в процессе ввода встретится очередной «пустой» символ, переменной word будет вновь присвоен признак NO.

Просмотрите программу и проверьте, правильно ли она интерпретирует случаи, когда между словами находится несколько «пустых» символов подряд.

Если вы захотите применить эту программу для работы с файлами, используйте операции переключения.

Программа, «рисующая» символами.

Давайте теперь займемся чем-нибудь менее утилитарным и более декоративным. Нашей целью является создание программы, с помощью которой вы сможете рисовать на экране геометрические фигуры, заполненные символами. Каждая выводимая строка представляет собой сплошной ряд одинаковых символов. Нам предоставляется возможность выбора символа, длины строки, а также начальной позиции, с которой она выводится на печать. Программа читает указываемые параметры до тех пор, пока не встретит признак EOF. Текст программы представлен.

Положим, мы вызываем программу sketcher. Чтобы ее выполнить, мы набираем на клавиатуре ее имя, затем вводим символ и два числа. На экране появляется отклик, после чего мы вводим другой набор параметров; отклик появляется вновь, и так до тех пор, пока мы не введем признак EOF. В среде ОС UNIX диалог может выглядеть следующим образом:

% sketcher

В 10 20

BBBBBBBBBBB

Y 12 18

YYYYYYY

[CTRL/d]

%

/\* художник-график \*/

/\* рисует сплошные фигуры \*/

#include <stdio.h>

#define MAXLENGTH 80

main()

{

int ch; /\* печатаемый символ \*/

int start, stop; /\* начальная и конечная позиции \*/

int count;

/\* счетчик позиций \*/

while((ch = getchar()) != EOF) /\* ввод символа \*/

{

if (ch != '\n') /\* пропуск символа «новая строка» \*/

{

scanf(" %d %d" , &start, &stop); /\* ввод граничных значений \*/

if (start > stop || start < 1 || stop > MAXLENGTH)

printf(" Введены неправильные граничные значения. \n");

else

{

count = 0;

while(++count < start)

putchar(' '); /\* печать пробелов вплоть до

начальной позиции \*/

while (count++ < = stop)

putchar(ch); /"печать символа до конечной позиции \*/

putchar('\n'); /\* закончить печать строки и

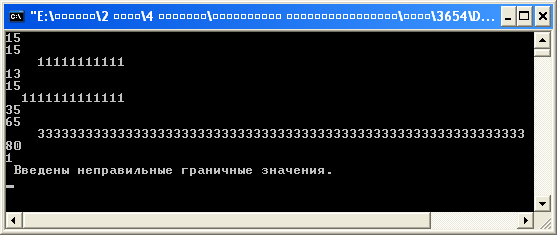
начать новую \*/

} /\* конец части else \*/

} /\* конец проверки содержимого ch \*/

} /\* конец цикла while \*/

} /\* конец программы \*/



Программа вывела на экран символ В в позициях с 10 по 20, а символ Y — с 12 по 18. К сожалению, при диалоговой работе с программой на экране наши команды перемежаются выводимыми строками. Гораздо более удобным способом использования программы является создание файла, содержащего подходящий набор данных, а затем применение операции переключения для ввода (из него) параметров в программу. Предположим, например, что в файле с именем fig содержатся следующие данные:

-30 50

| 30 50

| 30 50

| 30 50

| 30 50

| 30 50

= 20 60

: 31 49

: 30 49

: 29 49

: 27 49

: 25 49

: 30 49

: 30 49

/ 30 49

: 35 48

: 15 48

**После ввода команды sketcher < fig вы увидите результат работы программы.**

**Замечание:** на устройствах печати и экранах дисплеев отношение высоты символа к его ширине может быть различным; в результате эти фигуры при печати выглядят более сжатыми по вертикали, чем изображенные на экране.

**Анализ программы.** Это короткая программа, но она оказалась сложнее тех примеров, которые мы обсуждали до сих пор. Рассмотрим некоторые ее элементы.

**Длина строки.** Мы ввели в программу ограничение на длину печатаемой строки (она не может быть больше 80 позиций), поскольку 80 символов — это стандартный формат (по ширине) экрана дисплеев различных типов, а также число символов при нормальной ширине листа бумаги на устройстве для печати. Вы можете, однако, переопределить величину константы MAXLENGTH, если при работе с программой захотите воспользоваться устройством, имеющим другую ширину строки.

**Структура программы.** В нашей программе имеются три цикла while, один оператор if и один оператор if-else. Посмотрим, чтобы каждый из них делает: while((ch = getchar()) != EOF)

Задачей первого цикла while является ввод нескольких наборов данных. (Каждый набор данных состоит из символа и двух целых чисел, указывающих границы его вывода.) Производя вначале чтение символа, мы смогли объединить в одном выражении его ввод и проверку того, не является ли он признаком EOF. Если прочитан символ EOF, то программа останавливается, не делая попытки ввести величины, соответствующие переменным start и stop. В противном случае при помощи функции scanf() указанным переменным присваиваются введенные значения, затем программа переходит к их обработке. Этим завершается выполнение тела цикла, после чего вводится новый символ, и весь процесс повторяется снова. Обратите внимание, что для чтения данных мы использовали два оператора, а не один. Почему нельзя было воспользоваться одним оператором?

scanf( %c %d %d" , &ch, &start, &stop)

Предположим, мы это сделали. Рассмотрим, что происходит, когда программа заканчивает чтение последнего набора данных из файла. Перед началом выполнения очередного тела цикла единственным оставшимся непрочитанным элементом файла будет признак EOF. Функция scanf() читает указанный символ и присваивает его переменной ch; затем она пытается ввести значение для переменной Start, но в файле не осталось данных, которые не были бы уже прочитаны! Компьютер «выскажет» свое недовольство, и ваша программа прекратит работу. Отделяя чтение символа от ввода остальных данных, мы позволяем компьютеру обнаружить появление признака EOF перед очередной попыткой прочесть оставшиеся данные.

if (ch != '\n' )

Цель введения в программу первого оператора if состоит в том, чтобы упростить чтение данных. Мы объясним, как он работает позднее.

if (start > stop || start < 1 || stop > MAXLENGTH)

printf(" Введены неправильные граничные значения.\n" );

else

Цель применения оператора if-else состоит в том, чтобы избежать использования в программе таких значений переменных start и stop, которые могут привести к нежелательным последствиям. Этот вопрос мы также обсудим ниже. Обратите, однако, внимание на то, как мы использовали логические операции и операции отношения, чтобы обнаружить появление любого из трех «опасных» значений.

Основная часть программы представляет собой составной оператор, который следует за ключевым словом else.

count = 0;

Вначале счетчик count устанавливается на нуль.

while (++count < start)

putchar(' ');

Затем в цикле while начинается вывод на печать пробелов вплоть до позиции, определяемой значением переменной start. Если значение Start, скажем, равно 10, то печатается девять пробелов. Поэтому вывод символов на печать начнется с 10-й позиции Обратите внимание, как использование префиксной формы операции увеличения вместе с операцией < позволяет добиться указанного эффекта. Если бы вместо этого мы использовали выражение count++ < start, то сравнение проводилось бы перед увеличением значения count, и в результате мог быть напечатан один дополнительный пробел

while ( count++ <= stop)

putchar(ch);

Второй цикл while в вышеупомянутом блоке осуществляет задачу вывода на печать символа, начиная с позиции, задаваемой переменной start, и кончая позицией, задаваемой переменной stop. На этот раз мы воспользовались постфиксной формой операции увеличения и операцией < =. Такая комбинация обеспечивает желаемый результат при выводе на печать символа — верхняя граничная позиция входит в поле печати. Для проверки этого факта вы можете воспользоваться логикой или методом проб и ошибок.

putchar(' \n');

Оператор putchar(' \n') используется для завершения печати данной строки и перехода на новую.

Форма данных. При написании программы необходимо понимать, как она будет взаимодействовать с входными данными. Сейчас мы обратимся к этому вопросу.

Вводимые данные должны быть представлены в форме, совместимой с требованиями, которые налагаются функциями ввода, используемыми в программе. Поэтому вся тяжесть приведения данных к правильной форме ложится на пользователя. В более сложных программах основной объем работы по такому преобразованию переносится на саму программу. Наилучшей формой представления вводимых данных является следующая:

Н 10 40

I 9 41

где за символом следуют номера начальной и конечной позиций. Но наша программа допускает также и такую форму данных:

Н 10 40I 9 41

или такую

Н10 40I 9 41

но не

Н 10 40 I 9 41

Почему наличие одних пробелов является необязательным, а других — обязательным? Почему символ «новая строка» может быть помещен между последним целым числом из одного набора данных и первым символом из следующего набора, а пробел нет? Эти вопросы поднимают проблемы, касающиеся не только данной программы. Рассмотрим работу функций getchar () и putchar() и найдем ответы на них.

Функция getchar() читает первый встретившийся символ независимо от того, является ли он алфавитным символом, пробелом, символом «новая строка» или еще чем-нибудь. Функция scanf() делает то же самое, если чтение производится в формате %с (символ). Но, когда scanf() осуществляет ввод данных в формате %d (целые), пробелы и символы «новая строка» пропускаются. Поэтому символы «новая строка» или любые пробелы между символом, считываемым функцией getchar(), и следующим целым числом, считываемым функцией scanf(), игнорируются. Функция scanf() читает цифры до тех пор, пока не встретит нецифровой символ — пробел, символ «новая строка» или букву. Следовательно, между первым и вторым целыми числами необходимо помещать пробел или символ «новая строка», чтобы функция scanf() могла распознать, где кончается одно и начинается другое.

Этим объясняется, почему между символом и следующим целым числом может стоять пробел или символ «новая строка», и почему между двумя целыми числами обязательно должен быть разделитель такого вида. Но почему между целым числом, стоящим в конце набора данных, и следующим символом не может стоять пробел? Потому что в следующий раз на очередном шаге выполнения цикла while функция getchar() осуществляет ввод символа из той позиции, где «остановилась» функция scanf(). Поэтому она прочтет любой следующий символ, стоящий после целого числа,— пробел, символ «новая строка» и т. п.

Если бы мы следовали требованиям функции getchar(), структуру данных необходимо было бы организовать так:

W10 50a20 60у10 30

где между последним целым числом, стоящим в конце группы, и следующим символом разделитель отсутствует. Но такая структура выглядит неуклюже, поскольку число 50 при этом выглядит так, как будто оно помещено в одну группу с а, а не с w. Поэтому введен оператор

if(ch != '\n')

чтобы иметь возможность обнаружить, когда значение ch равно символу «новая строка». Вместо этого можно использовать данные, вводимые в виде

w10 50

а20 60

у10 30

где между числом 50 и а помещен символ «новая строка». Программа читает этот символ, игнорирует его и затем переходит к чтению следующего символа.

Контроль ошибок. Существует широко распространенная проблема, связанная с вводом в машину данных, которые должны использоваться определенным образом. Один из методов ее решения состоит в «контроле ошибок». Это означает, что, перед тем как приступить к обработке данных, программа должна проверить их правильность. В нашей программе мы сделали первую попытку осуществить такой контроль ошибок с помощью операторов:

if(start > stop || start < 1 || stop > MAXLENGTH)

printf(" Введены неправильные граничные значения.\n" );

Они входят в структуру if-else, которая определяет, что основная часть программы будет выполняться только в том случае, если ни один из трех if-тестов не окажется истинным.

С какой целью мы принимаем все эти меры предосторожности? Во-первых, совершенно неправильно размещать начальную позицию после конечной, поскольку обычно на терминал данные выводятся слева направо, а не наоборот. Поэтому с помощью выражения start > stop проверяется наличие такой потенциальной ошибки. Во-вторых, при выводе на экран первый столбец имеет номер 1; мы не можем выводить данные левее левого края. Выражение start < 1 служит средством обнаружения такой ошибки. И наконец, с помощью выражения stop > MAXLENGTH проверяется, не попытаемся ли мы вывести на печать данные правее правого края. Существуют ли еще какие-нибудь ошибочные значения, которые мы можем присвоить переменным start и stop? Можно было бы, конечно, попробовать присвоить переменной start значение большее, чем MAXLENGTH. Может ли этот вариант успешно пройти наш тест? Нет, хотя наличие подобной ошибки мы и не проверяем непосредственно. Предположим, что величина Start больше константы MAXLENGTH. Тогда либо значение stop тоже превышает величину MAXLENGTH, что обязательно приведет к обнаружению ошибки, либо stop окажется меньшей или равной MAXLENGTH. Но тогда ее значение должно быть меньше величины Start, что приведет к обнаружению этой ошибки первым тестом. Другая вероятная ошибка может состоять в том, что значение stop окажется меньше 1. Мы оставляем читателям в качестве самостоятельного упражнения проверку того, что данная ошибка также не останется незамеченной.

В нашей программе контроль ошибок выглядит весьма простым. Если вы проектируете программу для серьезных целей, вы обязаны обратить на этот вопрос больше внимания, чем мы. Например, выводимые сообщения об ошибках могли бы указывать, какие величины неверны и почему; кроме того, вы могли бы привнести в сообщения что-то от себя и придать им большую эмоциональную окраску. Приведем несколько примеров:

Указанное вами значение stop — 897654 превышает ширину экрана.

Вот эта да! У вас START больше, чем STOP. Попробуйте, пожалуйста, еще раз.

ВЕЛИЧИНА START ДОЛЖНА БЫТЬ БОЛЬШЕ 0, ИНДЮК.

Привносимые личностные моменты относятся, конечно, и к вам самим.

### Цикл do-while.

Цикл do-while отличается от циклов for и while тем, что это — цикл с постусловием. Другими словами: цикл всегда выполняется хотя бы один раз, после чего в конце первого прохода проверяется условие продолжения цикла. В отличие от этого циклы for и while могут не выполняться вообще, или выполняться множество раз в зависимости от значения переменной управления циклом. Поскольку циклы do-while выполняются, по меньшей мере, один раз, их лучше использовать тогда, когда нет сомнений о вхождении в определенный цикл. Например, если программа должна выводить пользовательское меню, то даже если пользователь захочет тут же выйти из программы, он должен увидеть это меню, чтобы определить, по какой клавише выходить из приложения.

Синтаксис цикла do-while следующий:

do

действие;

(проверка\_условия) ;

При множестве действий в операторе do-while необходимы фигурные скобки:

do {

действие1;

действие2;

действиеЗ;

действиеn;

} while(проверка\_условия) ;

Оба цикла, while и for, являются циклами с предусловиями. Проверка истинности условия осуществляется перед началом каждой итерации цикла. В языке Си имеется также конструкция цикла с постусловием (условием на выходе), где истинность условия проверяется после выполнения каждой итерации цикла. Этот подход реализуется с помощью цикла do while, который иллюстрируется следующим примером:

do

{

ch = getchar();

putchar(ch);

} while (ch != ' \n')

Это сильно отличается от записи, например, такого вида

while ((ch = getchar()) != ' \n' )

putchar(ch);

Различие начинается с того момента, когда прочитан символ «новая строка». Цикл while печатает все символы вплоть до появления первого символа «новая строка», а цикл do while — все символы вплоть до символа «новая строка» включительно. Только после печати этого символа в цикле производится проверка, является ли последний прочитанный символ символом «новая строка». В цикле while эти действия осуществляются перед проверкой истинности условия.

В общем виде цикл do while записывается следующим образом:

do

оператор

while (выражение);

Такой оператор может быть как простым, так и составным.

09_04

Тело цикла do while всегда выполняется по крайней мере один раз, поскольку проверка осуществляется только после его завершения. Тело цикла for или while, возможно, не будет выполнено ни разу, поскольку проверка осуществляется перед началом его выполнения. Использовать цикл do while лучше всего в тех случаях, когда должна быть выполнена по крайней мере одна итерация. К примеру, мы могли бы применить цикл do while в нашей программе угадывания числа. На псевдокоде алгоритм работы программы можно тогда записать следующим образом:

do

{

выдвиньте предположение

получите ответ вида д, б, или м

} while (ответ не совпадает с д)

Вы должны избегать использования цикла do while, структура которого аналогична представленной ниже в записи на псевдокоде:

спросите пользователя, хочет ли он продолжать

do

некоторый умный вздор

while (oтвет будет да)

В данном случае, после того как пользователь ответит «нет», «некоторый умный вздор» будет выполнен, поскольку проверка осуществляется слишком поздно.

В следующей программе на C++ цикл do-while используется для печати меню и получения правильного ответа пользователя. Это приложение работает только под DOS.

// 07DOWHIL.CPP

// DOS-программа, написанная на C++; иллюстрирует

// правильное использование цикла do-while с постусловием

// для отображения меню при помощи функций текстового вывода

#include <iostream.h>

#include <conio.h>

#include <graph.h>

#include <stdlib.h>

main()

{

int iuser\_response;

struct rccoord rcorig\_coords;

struct videoconfig svideo\_config;

if(!\_setvideomode(\_MAXRESMODE))

exit;

\_getvideoconfig(&svideo\_config);

\_settextcolor(9); // ярко-синий

do {

\_clearscreen(\_GCLEARSCREEN);

\_settextposition(3,23);

\_outtext(">>> Welcome to Metro-Teller <<<\n\n") ;

\_settextposition(5,29);

\_outtext("Instructions 1\n"); // Команды

\_settextposition(6,29);

\_outtext("IRA Balance 2\n"); //Баланс

\_settextposition(7,29);

\_outtext("Loan Rates 3\n"); // Ссуды

\_settextposition(8,29);

\_outtext("VISA Transaction 4\n"); // Работа с VISA-картами

\_settextposition(9,29);

\_outtext("Ready-Reserve 5\n"); // "Горячий" резерв

\_settextposition(10,29);

\_outtext("Deposit 6\n"); //Депозит

\_settextposition(11,29);

\_outtext("Withdrawal 7\n"); // Отмена операции

\_settextposition(12,29);

\_outtext("Quit 8\n"); // Выход

\_settextposition(14,25);

\_outtext("Enter your selection: "); // Ваш выбор:

rcorig\_coords = \_gettextposition();

cin >> iuser\_response;

while(iuser\_response < 1 II iuser\_response > 8) {

\_settextposition(rcorig\_coords.row,rcorig\_coords.col);

\_outtext(" ");

\_settextposition(rcorig\_coords.row,rcorig\_coords.col);

cin >> iuser\_response;

}

} while(iuser\_response != 8);

\_setvideomode(\_DEFAULTMODE);

return(0);

}

Для того чтобы сделать программу немного интереснее, в нее включены заголовочные файлы conio.h и graph.h. Эти файлы содержат много полезных функций управления монитором. Программа использует три из них: \_gettextposition(), settextposition() и \_outtext(). Перед обсуждением циклов do-while рассмотрим эти функции. Две функции, \_gettextposition() и \_settextposition(), соответственно возвращают и устанавливают текущие координаты курсора. Функция \_outtext() выводит текст на экран в графическом режиме.

Цикл do-while используется в программе для печати опций меню и выводит их всякий раз заново до тех пор, пока пользователь не выберет опцию 8 для выхода.

Обратите внимание на то, что в программе имеется также вложенный внутренний цикл while. Этот цикл обеспечивает правильность значения, введенного пользователем (числа от 1 до 8, включительно). Так как нежелательно, чтобы ошибочный ввод привел к обновлению экрана, во внутреннем цикле используется оператор \_settextposition() для фиксирования курсора на той же строке, где был предыдущий ответ. Для этого в программе запоминается первоначальное положение курсора после вывода сообщения "Enter your selection:" ("Ваш выбор:"). Именно для этой цели использована функция \_gettextposition().

Однако, если пользователь вводит некоторое значение, то координаты их у курсора меняются; поэтому для повторного обращения их начальные значения запоминаются в элементах структуры rcorig\_coords.row и rcorig\_coords.col. Если произошел вход во внутренний цикл while, первый оператор \_outtext() стирает все ранее введенные значения, а затем происходит ввод следующего числа. Этот процесс повторяется до тех пор, пока будет получено допустимое значение переменной iuser\_response. После выхода из внутреннего цикла while управление возвращается внешнему циклу do-while, в котором меню выводится снова и снова до тех пор, пока пользователь введет для выхода число 8. Сможете ли вы переписать программу при помощи \_settextposition() так, чтобы не выводить меню заново при каждом допустимом значении iuser\_response?

### Цикл for.

Синтаксис цикла for следующий:

for(инициализация; проверка\_условия; коррекция)

оператор;

При выполнении цикла for сначала осуществляется инициализация. Она выполняется в начале цикла и больше никогда не повторяется. Обычно этот оператор инициализирует переменную, управляющую циклом. Затем происходит проверка\_условия, называемая условием окончания цикла. Если проверка\_условия дает значение "истина", то выполняется один или несколько операторов внутри цикла. Если произошел вход в цикл, то после выполнения всех операторов внутри цикла осуществляется коррекция. Если же проверка\_условия дала значение "ложь", то операторы внутри цикла и оператор коррекции игнорируются и управление передается оператору, следующему за циклом. Цикл for с несколькими операторами внутри выглядит следующим образом:

for(инициализация; проверка\_условия; коррекция) {

оператор\_а;

оператор\_b;

оператор\_c;

оператор\_n;

}

Если нужно выполнить несколько операторов, то необходимы фигурные скобки для их объединения с управляющей структурой цикла.

В цикле for распространен для «математических повторений. Вот пример его записи:

for (count = 1; count <= NUMBER; count ++ )

printf(" Будь моим Валентином!\n");

В круглых скобках содержатся три выражения, разделенные символом «точка с запятой». Первое из них служит для инициализации счетчика. Она осуществляется только один раз — когда цикл for начинает выполняться. Второе выражение — для проверки условия; она производится перед каждым возможным выполнением тела цикла. Когда выражение становится ложным (или в общем случае равным нулю), цикл завершается. Третье выражение вычисляется в конце каждого выполнения тела цикла. Ранее мы использовали его для увеличения значения счетчика count, но, вообще говоря, его использование этим не ограничивается. За заголовком цикла for следует простой или составной оператор. Рисунок служит иллюстрацией структуры цикла for.

09_02

Сейчас мы продемонстрируем, как цикл for используется в программе, печатающей таблицу кубов целых чисел:

/\* таблица кубов \*/

main()

{

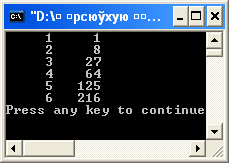
int num;

for(num = 1; num < = 6; num++ )

printf(" %5d %5d \n" , num, num\*num\*num);

}

Эта программа выводит на печать числа от 1 до 6 и их кубы:



Инициализация выполняется один раз перед началом работы цикла.

Из первой строки цикла for мы сразу можем узнать всю информацию о параметрах цикла: начальное значение переменной num, ее конечное значение, а также насколько увеличивается значение переменной num при каждом выполнении тела цикла.

Цикл for часто используется для реализации в программе временной задержки с целью согласования скорости реагирования (в данном случае замедления) машины с возможностями восприятия человека.

for(n = 1; n <= 10000; n++ )

;

Этот цикл заставляет машину считать до 10000. Единственный символ «точка с запятой», расположенный во второй строке, информирует нас о том, что никаких других действий в этом цикле не производится. Такой уединенный символ «точка с запятой» можно представлять себе как «пустой» оператор, т. е. оператор, который не выполняет никаких действий.

Рассмотрим несколько примеров цикла for.

В следующем примере выполняется суммирование первых пяти целых чисел. Подразумевается, что переменные isum и ivalue предварительно описаны как целые:

isum = 0;

for(ivalue=1; ivalue <= 5: ivalue++)

isum += ivalue;

После инициализации переменной isum значением 0, выполняется цикл for. Во-первых, переменная ivalue устанавливается в 1 (это осуществляется только один раз); во-вторых, значение ivalue сравнивается с условием окончания цикла <=5. Поскольку результат сравнения — "истина", к переменной isum добавляется 1. Как только оператор выполнился, значение переменной управления циклом (ivalue) увеличивается на 1. Этот процесс повторяется еще четыре раза до тех пор, пока значение ivalue не увеличится до 6 и цикл не закончится.

Аналогичный фрагмент можно записать на C++ следующим образом (заметите ли вы некоторое отличие?):

for(int ivalue=1; ivalue <= 5: ivalue++)

isum += ivalue;

В C++ можно объявлять и инициализировать переменную управления циклом внутри цикла for. Это очень важное дополнение для программистов, заботящихся о структурированности программы и о правильности расположения объявлений переменных. В C++ можно объявлять переменные непосредственно перед тем оператором, в котором они используются. В предыдущем примере допустимо локальное объявление переменной ivalue, так как она используется только для вычисления isum — переменной, имеющей большую, чем ivalue, область действия. Взгляните, однако, на следующий пример:

int isum = 0;

for(int ivalue=1; ivalue <= 5: ivalue++)

isum += ivalue;

Теперь нет нужды "разыскивать" переменную isum по всему тексту, поскольку она не объявлена сразу за заголовком функции. Чтобы добиться структурированности проектирования и отладки, лучше всего использовать локализованные объявления всех переменных. Редко бывает, когда в программе оправдано объявление некоторой переменной в нестандартном месте, в результате чего затрудняется чтение, проверка и возможность редактирования программы.

Значение, используемое в циклах for в качестве переменной управления циклом, не всегда изменяется на 1 (при помощи операции ++). В следующем примере суммируются все нечетные числа до 9:

iodd\_sum = 0;

for(iodd\_value=1; iodd\_value <= 9; iodd\_value+=2)

iodd\_sum += iodd\_value;

В этом примере переменная управления циклом iodd\_value инициализируется значением 1 и увеличивается на 2.

Конечно же, циклы for не всегда выполняют пересчет от меньшего значения к большему. В следующем примере цикл for используется для чтения массива символов и последующей печати символьной строки в обратном направлении:

// 07FORLP.CPP

// Программа на C++, в которой цикл for используется для

// ввода символьного массива

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

//#include <iostream.h>

using namespace std;

#define CARRAY\_SIZE 10

main ()

{

int ioffset;

char carray[CARRAY\_SIZE];

for(ioffset = 0; ioffset < CARRAY\_SIZE; ioffset++)

carray [ioffset] = getchar();

for (ioffset = CARRAY\_SIZE - 1; ioffset >= 0; ioffset--)

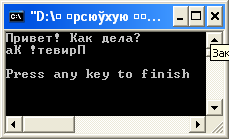
putchar(carray[ioffset]);

printf ("\n\nPress any key to finish\n"); /\* Конец работы \*/

\_getch();

return(0);

}



В этом примере в первом цикле for инициализируется переменная ioffset, получая значение 0 (это необходимо, поскольку все индексы массива представляют собой смещение относительно начального адреса первого элемента массива), затем символы считываются до тех пор, пока имеются свободные элементы в массиве carray. Во втором цикле for переменная управления циклом ioffset получает значение смещения последнего элемента массива, и до тех пор, пока в ioffset содержится допустимое смещение, символы печатаются в обратном порядке. Такая процедура может иметь смысл для синтаксического анализа некоторого инфиксного выражения, преобразованного в префиксное.

Если используются несколько циклов for, как в следующем примере, необходимо в соответствующих местах ставить фигурные скобки для правильного выполнения операторов:

/\*07NSLOP1.C

Программа на С, поясняющая необходимость осторожного использования вложенных циклов\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

//#include <iostream.h>

using namespace std;

main ()

{

int iouter\_val, iinner\_val;

for(iouter\_val = 1; iouter\_val <= 4; iouter\_val++) {

printf("\n%3d --",iouter\_val);

for(iinner\_val = 1; iinner\_val <= 5; iinner\_val++ )

printf("%3d",iouter\_val \* iinner\_val);

}

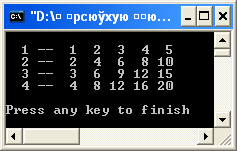
printf ("\n\nPress any key to finish\n"); /\* Конец работы \*/

\_getch();

return(0);

}

Эта программа после выполнения дает следующий результат:



Предположим, однако, что внешний цикл for был записан без фигурных скобок, как в следующем фрагменте:

/\*07NSLOP2.C

Программа на С, демонстрирующая результат выполнения вложенных циклов, в которых

отсутствуют логически необходимые фигурные скобки\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

//#include <iostream.h>

using namespace std;

main ()

{

int iouter\_val, iinner\_val;

for(iouter\_val = 1; iouter\_val <= 4; iouter\_val++)

printf("\n%3d--",iouter\_val);

for (iinner\_val = 1; iinner\_val <= 5; iinner\_val++ )

printf("%3d",iouter\_val \* iinner\_val);

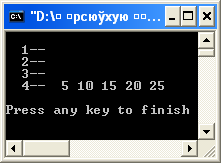
printf ("\n\nPress any key to finish\n"); /\* Конец работы \*/

\_getch();

return(0);

}

Результат был бы совсем другой:



Если вокруг первого цикла for нет фигурных скобок, то с циклом связан только первый оператор printf(). После того, как оператор printf() выполнится 5 раз, происходит переход ко второму циклу for. Во внутреннем цикле последнее значение переменной iouter\_val (5) используется для вычисления значений, печатаемых оператором printf().

Использовать или не использовать фигурные скобки — не простой вопрос, который нужно решать, учитывая соображения читабельности программы. Взгляните на два следующих примера; сможете ли вы определить: одинаков ли результат работы этих программ.

Вот первый пример:

/\*07LPDMO1.C

Еще одна программа на С, поясняющая необходимость осторожного

использования вложенных циклов\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

//#include <iostream.h>

using namespace std;

main ()

{

int iouter\_val, iinner\_val;

for(iouter\_val = 1; iouter\_val <= 4; iouter\_val++) {

for(iinner\_val = 1; iinner\_val <= 5; iinner\_val++ )

printf("%d ",iouter\_val \* iinner\_val);

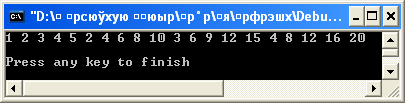
}

printf ("\n\nPress any key to finish\n"); /\* Конец работы \*/

\_getch();

return(0);

}



Сравните предыдущий пример со следующим:

/\*07LPDMO2.C

Для сравнения — программа на С, поясняющая необходимость осторожного

использования вложенных циклов\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

//#include <iostream.h>

using namespace std;

main()

{

int iouter\_val, iinner\_val;

for(iouter\_val = 1; iouter\_val <= 4; iouter\_val++)

for(iinner\_val = 1; iinner\_val <= 5; iinner\_val++ )

printf("%d ",iouter\_val \* iinner\_val);

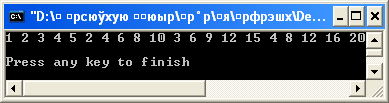
printf ("\n\nPress any key to finish\n"); /\* Конец работы \*/

\_getch();

return (0);

}

Обе программы выдают один и тот же результат:



В обоих примерах единственным оператором, связанным с внешним циклом for, является внутренний цикл for. Внутренний цикл for считается одним оператором. Это оставалось бы справедливым, даже если бы во внутреннем цикле for было множество исполняемых операторов. Поскольку фигурные скобки нужны только для блоков кода или нескольких операторов, для правильной работы программы во внешнем цикле for скобки не нужны.

Операция «запятая» в цикле for.

Операция «запятая» увеличивает гибкость использования цикла for, позволяя включать в его спецификацию несколько инициализирующих или корректирующих выражений. Например, ниже приводится программа, которая выводит на печать величины почтовых тарифов первого класса обслуживания. (Предположим, что почтовые тарифы: 20 центов за первую унцию и по 17 центов за каждую следующую.)

/\* почтовые тарифы \*/

#define FIRST 20

#define NEXT 17

main()

{

int ounces, cost;

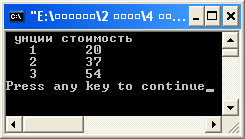
printf(" унции стоимость\n" );

for(ounces = 1, cost = FIRST; ounces < - 16; ounces++, cost + = NEXT)

printf(" %3d %7d\n" , ounces, cost);

}

Первые четыре строки результата работы программы будут выглядеть следующим образом:



Мы воспользовались операцией «запятая» в первом и третьих выражениях: в первом случае она позволяет инициализировать переменные ounces и cost; во втором — на каждой итерации увеличивать значение ounces на 1, a cost на 17 (величину константы NEXT). Все вычисления осуществляются в спецификации цикла for.

Применение операции «запятая» не ограничено только циклами for, но именно в них она используется особенно часто. Операция обладает одним дополнительным свойством: при ее использовании гарантируется, что выражения, к которым она применяется (т. е. выражения, разделенные запятой), будут вычисляться слева направо. Поэтому переменная ounces будет инициализирована до переменной cost. В данном примере это не имеет значения, но порядок инициализации мог бы оказаться существенным, если выражение, соответствующее cost, содержало бы переменную ounces.

Символ «запятая» также используется как разделитель. Поэтому запятые в операторах

char ch, date;

или

printf(" %d %d\n", chimps, chumps);

являются разделителями, а не знаками операции «запятая».

Гибкость конструкции for.

Хотя цикл for на первый взгляд очень похож на цикл DO в Фортране, цикл FOR в Паскале и цикл FOR ... NEXT в Бейсике, for в Си является гораздо более гибким средством, чем любой из упомянутых. Эта гибкость — следствие способа использования упомянутых выше трех выражений в спецификации цикла for. До сих пор первое выражение применялось для инициализации счетчика, второе — для задания его граничного значения, а третье — для увеличения его текущего значения на 1. Использованный таким образом, оператор for в языке Си совершенно аналогичен упомянутым выше соответствующим операторам в других языках. Но, кроме описанной, существует еще и много других возможностей его применения, девять из которых мы приводим ниже.

1. Можно применять операцию уменьшения для счета в порядке убывания вместо счета в порядке возрастания.

for (n = 10; n > 0; n--)

printf(" %d секунд!\n", n);

printf(" Пуск!\n");

1. При желании вы можете вести счет двойками, десятками и т. д.

for (n = 2; n < 60; n = n + 13)

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

В этом операторе значение переменной п будет увеличиваться на 13 при каждом выполнении тела цикла; будут напечатаны числа 2, 15, 28, 41 и 54.

Заметим, между прочим, что в языке Си имеется и другая сокращенная форма записи для увеличения переменной на фиксированную величину. Вместо выражения

n = n + 13

можно воспользоваться записью

n += 13

Знак + = определяет «аддитивную операцию присваивания», в результате выполнения которой величина, стоящая справа, прибавляется к значению переменной, расположенной слева. Дополнительные детали, относящиеся к этой операции, приведены ниже.

1. Можно вести подсчет с помощью символов, а не только чисел:

for (ch = 'а' ; ch <= 'z'; ch++ )

printf(" Величина кода ASCII для %с равна %d.\n", ch, ch);

При выполнении этого оператора будут выведены на печать все буквы от а до z вместе с их кодами ASCII. Этот оператор «работает», поскольку символы в памяти машины размещаются в виде чисел, и поэтому в данном фрагменте счет ведется на самом деле с использованием целых чисел.

1. Можно проверить выполнение некоторого произвольного условия, отличного от условия, налагаемого на число итераций. В нашей программе таблица кубов вы могли бы заменить спецификацию

for (num = 1; num < = 6; num++ )

на

for (num = 1; num\*num\*num <= 216; num ++ )

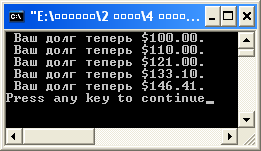
Это было бы целесообразно в случае, если бы нас больше занимало ограничение максимального значения диапазона кубов чисел, а не количества итераций.

1. Можно сделать так, чтобы значение некоторой величины возрастало в геометрической, а не в арифметической прогрессии, т. е. вместо прибавления фиксированного значения на каждом шаге цикла, выполнялось бы умножение:

for (debt = 100.0; debt < 150.0; debt = debt\* 1.1)

printf(" Ваш долг теперь $%.2f.\n" , debt);

В этом фрагменте программы значение переменной debt умножается на 1.1 на каждом шаге цикла, что увеличивает ее на 10%. Результат выглядит следующим образом:



Как вы уже смогли догадаться, для умножения debt на 1.1 также существует сокращенная запись. Мы могли бы использовать выражение

debt \*= 1.1

Для получения того же результата. Знак \*= определяет «мультипликативную операцию присваивания», при выполнении которой значение переменной, расположенной слева, умножается на величину, стоящую справа.

1. В качестве третьего выражения можно использовать любое правильно составленное выражение. Какое бы выражение вы ни указали, его значение будет меняться при каждой итерации.

for (x = 1; у <= 75; у = 5\*х+++ 10);

printf("%10d %10d\n", x, у);

В этом фрагменте выводятся на печать значения переменной х и алгебраического выражения 5\*х + 10. Результат будет выглядеть так:

Обратите внимание, что в спецификации цикла проверяется значение у, а не х. В каждом из трех выражений, управляющих работой цикла for, могут использоваться любые переменные.

Хотя этот пример и правилен, он не может служить иллюстрацией хорошего стиля программирования. Программа выглядела бы гораздо понятнее, если бы мы не смешали процесс изменения переменной цикла с алгебраическими вычислениями.

1. Можно даже опустить одно или более выражений (но при этом нельзя опустить символы «точка с запятой»). Необходимо только включить в тело цикла несколько операторов, которые в конце концов приведут к завершению его работы.

ans = 2;

for (n = 3; ans < = 25;)

ans = ans\*n;

При выполнении этого цикла величина n останется равной 3. Значение переменной ans вначале будет равно 2, потом увеличится до 6, 18, а затем будет получена окончательная величина 54. (18 меньше 25, поэтому в цикле for выполняется еще одна итерация, и 18 умножается на 3, давая результат 54). Тело цикла

for(; ;)

printf(" Я хочу сделать что-нибудь\n" );

будет выполняться бесконечное число раз, поскольку пустое условие всегда считается истинным.

1. Первое выражение не обязательно должно инициализировать переменную. Вместо этого, например, там мог бы стоять оператор printf() некоторого специального вида. Необходимо помнить только, что первое выражение вычисляется только один раз перед тем, как остальные части цикла начнут выполняться.

for ( printf(" Запоминайте введенные числа!\n"); num = = 6; )

scanf(" %d", &num);

printf("Эro как раз то, что я хочу!\n");

В этом фрагменте первое сообщение оказывается выведенным на печать один раз, а затем осуществляется прием вводимых чисел до тех пор, пока не поступит число 6.

1. Параметры, входящие в выражения, находящиеся в спецификации цикла, можно изменить при выполнении операций в теле цикла. Предположим, например, что у вас есть цикл со спецификацией следующего вида:

for(n = 1; n < 1000; n += delta)

И если после нескольких итераций ваша программа решает, что величина параметра delta слишком мала или велика, оператор if внутри цикла может изменить значение параметра. В диалоговой программе пользователь может изменить этот параметр в процессе выполнения цикла.

Короче говоря, большая свобода выбора вида выражений, управляющих работой цикла for, позволяет с помощью этой конструкции делать гораздо больше, чем просто выполнять фиксированное число итераций. Возможности цикла for могут быть еще более расширены путем использования операций, которые мы вкратце обсудим ниже.

Философ Зенон и цикл for.

Философ Зенон известен своими знаменитыми парадоксами (апориями). Посмотрим, как с помощью операции «запятая» можно разрешить старый парадокс. Греческий философ Зенон утверждал, что пущенная стрела никогда не достигнет цели. Сначала, говорил он, стрела пролетит половину расстояния до цели. После этого ей останется пролететь половину всего расстояния, но сначала она должна будет пролететь половину того, что ей осталось пролететь, и т. д. до бесконечности. Поскольку расстояние полета разбито на бесконечное число частей, для достижения цели стреле может потребоваться бесконечное время. Мы сомневаемся, однако, что Зенон вызвался бы стать мишенью для стрелы, полагаясь только на убедительность своего аргумента.

Применим количественный подход и предположим, что за одну секунду полета стрела пролетает первую половину расстояния. Тогда за последующую 1/2 секунды она пролетит половину того, что осталось от половины, за 1/4 — половину того, что осталось после этого, и т д. Полное время полета представляется в виде суммы бесконечного ряда 1 + 1/2 + 1/4 + 1/8 + 1/16 + ... . Мы можем написать короткую программу для нахождения суммы первых нескольких членов.

/\* Зенон \*/

#define LIMIT 15

main ()

{

int count;

float sum, x;

for (sum = 0.0, x = 1.0, count = 1; count <= LIMIT; count++, x \*= 2.0)

{

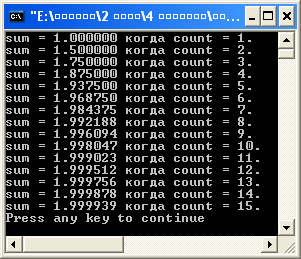
sum + = 1.0/x;

printf("sum = %f когда count = %d.\n" , sum, count);

}

}

В результате выполнения программы получим значения сумм, соответствующих первым 15 членам ряда:



Можно видеть, что, хотя мы и добавляем новые члены, сумма, по-видимому, стремится к какому-то пределу. И действительно, математики показали, что при стремлении числа членов к бесконечности сумма ряда сходится к 2,0, что и демонстрируется нашей программой. Какая радость! Если бы Зенон оказался прав, движение было бы невозможно. (Но если бы движение было невозможно, то не было бы Зенона.)

Что можно сказать по поводу самой программы? В ней показано, что в одном выражении можно использовать более, чем одну операцию «запятая». В спецификации цикла мы инициализировали переменные sum, x и count. После задания условий выполнения цикла оставшаяся часть программы оказывается очень короткой.

### Сравнение циклов.

После того как вы решили, что вам необходимо использовать оператор цикла, возникает вопрос: циклом какого вида лучше всего воспользоваться? Во-первых, решите, нужен ли вам цикл с предусловием или же с постусловием. Чаще вам нужен будет цикл с предусловием. По оценкам Кернигана и Ритчи, в среднем циклы с постусловием (do while) составляют только 5% общего числа используемых циклов. Существует несколько причин, по которым программисты предпочитают пользоваться циклами с предусловием; в их числе один общий принцип, согласно которому лучше посмотреть, куда вы прыгаете, до прыжка, а не после. Вторым моментом является то, что программу легче читать, если проверяемое условие находится в начале цикла. И наконец, во многих случаях важно, чтобы тело цикла игнорировалось полностью, если условие вначале не выполняется.

Положим, вам необходимо использовать цикл с предусловием. Что лучше: цикл for или цикл while? Отчасти это дело вкуса, поскольку все, что вы можете сделать с помощью одного, вы можете сделать и с помощью другого. Для превращения цикла for в цикл While нужно опустить первое и третье выражения:

for (; проверка условия;)

Такая спецификация эквивалентна записи

while (проверка условия)

Для превращения цикла while в цикл for необходимо предварительно осуществить инициализацию некоторых выбранных переменных и включить в тело цикла операторы, корректирующие их значения:

инициализация;

while (проверка условия)

{

тело;

коррекция;

}

Данная запись по своим функциональным возможностям эквивалентна следующей:

for (инициализация; проверка условия, коррекция)

тело;

Исходя из соображений стиля программирования, применение цикла for представляется более предпочтительным в случае, когда в цикле используется инициализация и коррекция переменной, а применение цикла while — в случае, когда этого нет. Поэтому использование цикла while вполне оправданно в случае

while((ch = getchar()) != EOF)

Применение цикла for представляется более естественным в случаях, когда в циклах осуществляется счет прохождений с обновлением индекса:

for (count = 1; count <= 100; count ++ )

### Вложенные циклы.

Вложенным называется цикл, находящийся внутри другого цикла. В этом разделе рассматривается пример, в котором вложенные циклы используются для нахождения всех простых чисел, не превышающих данного значения. Простое число — это такое число, которое делится нацело только на 1 и само на себя. Первыми простыми числами будут 2, 3, 5, 7 и 11.

Самый легкий способ узнать, является ли число простым, состоит в делении его на все числа между 1 и им самим. Если оно делится нацело на какое-нибудь число из этого ряда, то оно — не простое. Мы воспользуемся операцией деления по модулю (%) для проверки, выполнялось ли деление нацело. (Вы не забыли еще, конечно, операцию деления по модулю? Ее результатом является остаток от деления первого операнда на второй. Если одно число делится на другое нацело, результатом операции деления по модулю будет 0.) При обнаружении какого-нибудь одного делителя числа дальнейшие проверки потеряют смысл. Поэтому в программе процесс проверки данного числа завершается после того, как найден его делитель.

Начнем с программы, проверяющей делимость одного числа; в ней имеется всего один оператор цикла.

/\* простое число1 \*/

main()

{

int number, divisor;

printf("О каком числе вы хотите знать, простое ли оно?\n");

scanf(" %d" , &number); /\* получение ответа \*/

while (number < 2) /\* число отвергается \*/

{

printf(" Извините, мы не принимаем чисел меньше 2.\n");

printf(" Пожалуйста, попробуйте еще раз.\n");

scanf(" %d" , &number);

}

for (divisor = 2; number % divisor != 0; divisor ++ )

; /\* проверка, простое число или нет, осуществляется

внутри спецификации цикла \*/

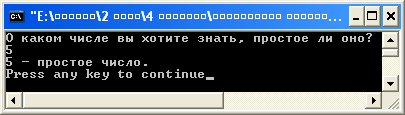
if (divisor = = number) /\* выполняется после завершения цикла \*/

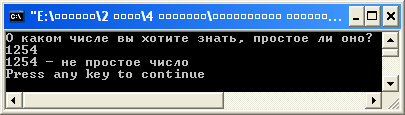
printf("%d — простое число.\n", number);

else

printf("%d — не простое число\n", number);

}





Мы воспользовались структурой цикла while, чтобы избежать ввода значений, которые могли бы привести к аварийному завершению программы.

Обратите внимание, что все вычисления выполняются внутри спецификации цикла for. Величина переменной number последовательно делится на возрастающие значения делителей до тех пор, пока не произойдет деление нацело (т. е. number % divisor станет равным 0). Если первым делителем, который приведет к такому результату окажется само это число, то значение переменной number — простое число. В противном случае данное число будет иметь меньший делитель, и это приведет к тому, что цикл завершится раньше.

Для нахождения всех простых чисел, меньших некоторой задан-ной величины, нам нужно будет заключить наш цикл for в некоторый другой цикл. На псевдокоде это будет выглядеть следующим образом:

для числа (number) = 1 до верхнего предела limit проверять, является ли число простым

Вторая строка представляет собой нашу предыдущую программу. Переводя эту запись на язык Си, получим\_программу:

/\* простые числа2 \*/

main()

{

int number, divisor, limit;

int count = 0;

printf(" Укажите, пожалуйста, верхний предел для поиска простых чисел.\n");

printf(" Верхний предел должен быть 2 или больше.\n");

scanf(" %d" , &limit);

while( limit < 2) /\* вторая попытка, если ошибка при вводе \*/

{

printf(" Вы были невнимательны! Попробуйте еще раз.\n");

scanf(" %d" , &limit);

}

printf(" Сейчас будут печататься простые числа!\n");

for (number = 2; number < = limit; number++) /\* внешний цикл \*/

{

for (divisor =2; number % divisor != 0; divisor++)

;

if(divisor = = number)

{

printf(" %5d" , number);

if (++count % 10 = = 0)

printf(" \n" ); /\* новая строка начинается через каждые 10 простых чисел \*/

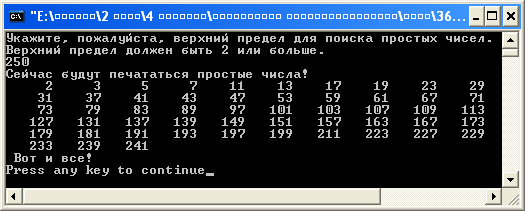
}

}

printf(" \n Вот и все!\n");

}

Во внешнем цикле каждое число, начиная с 2 и кончая величиной limit, последовательно берется для проверки. Указанная проверка осуществляется во внутреннем цикле. Мы использовали переменную count для хранения счетчика получаемых простых чисел. При печати каждое одиннадцатое простое число мы начинаем с новой строки. Ниже приводится пример результатов, получаемых с помощью такой программы:



Этот способ решения довольно прост, но он не является самым эффективным. Например, если вы хотите узнать, является ли число 121 простым или нет, нет необходимости проверять, существуют ли у него делители, превышающие 11. Дело в том, что если у данного числа существует делитель, больший 11, то результатом деления будет число, меньшее 11; тогда этот делитель был бы обнаружен раньше. Поэтому требуется проверять только делители, не превышающие величину квадратного корня из числа, но в данном случае программа будет несколько сложнее. Мы оставляем ее в качестве упражнения любознательному читателю. (Указание: вместо того чтобы сравнивать делитель с величиной квадратного корня из числа, сравнивайте квадрат делителя с самим числом.)

### Алгоритмы и псевдокод.

А теперь вернемся к нашей «тупоумной» программе, угадывающей число. Недостаток этой программы кроется не в программировании самом по себе, а в «алгоритме», т. е. методе, используемом для отгадывания числа. Этот метод можно описать следующим образом:

попросите пользователя задумать число

компьютер начинает угадывание с 1

до тех пор пока догадка неверна, предлагаемое значение увеличивается на 1

Эта запись, между прочим, служит примером «псевдокода», представляющего собой способ выражения смысла программ на упрощенном языке и являющегося некоторым аналогом языка машины. Псевдокод очень эффективен при разработке логики программы. После того как логика покажется вам правильной, вы можете обратить основное внимание на детали перевода псевдокода на реальный язык программирования. Преимущество использования псевдокода состоит в том, что он позволяет сконцентрироваться на логике и структуре программы, не заботясь пока о способе перевода этих идей на язык машины.

Если мы хотим улучшить программу, нам в первую очередь необходимо улучшить алгоритм. Один из методов заключается в том, чтобы выбрать число где-нибудь посередине между 1 и 100 (число 50 нам вполне подходит) и попросить пользователя ответить, больше ли это число задуманного, меньше его или равно ему. Если он сообщает, что данное число слишком велико, то тем самым из рассмотрения немедленно исключаются все числа между 50 и 100. Следующей догадкой программы является число, выбранное где-то посередине между 1 и 49. И снова ответ на вопрос, велико или мало это число, позволит исключить из рассмотрения половину оставшихся возможных чисел; программа продолжает указанный процесс, быстро сужая поле поиска до тех пор, пока задуманное число не будет угадано. Давайте запишем эти логические рассуждения на псевдокоде. Пусть highest — максимально возможная величина отгадываемого числа, a lowest — его минимально возможное значение. Вначале этими величинами будут соответственно 100 и 1, поэтому алгоритм запишется следующим образом:

установить highest равным 100

установить lowest равным 1

попросить пользователя задумать число

предложенное значение (guess) равно

(highest + lowest)/2

пока догадка неверна, делать следующее:

{если предложенное значение велико, установить highest равным этому предложенному значению минус 1

если предложенное значение мало, установить lowest равным этому предложенному значению плюс 1

новое предложенное значение равно (highest + lowest)/2}

Обратите внимание на логику алгоритма: если предложенное значение, равное 50, велико, то максимально возможная величина задуманного числа будет равна 49. Если же значение 50 мало, то минимально возможная величина числа будет равна 51.

Сейчас мы переведем текст, указанный выше, на язык Си. Полученная программа представлена ниже.

/\* угадывание числа2 \*/

/\* более эффективный способ угадывания \*/

#include <stdio.h>

#define HIGH 100

#define LOW 1

main()

{

int guess = (HIGH + LOW)/2;

int highest = HIGH;

int lowest = LOW;

char response;

printf(" Задумайте число от %d до %d. Я попробую" , LOW, HIGH);

printf(" угадать его.\n Отвечайте д, если моя догадка правильна,");

printf(" б, если \n больше, и м, если");

printf(" меньше. \n);

printf("Итак ... ваше число %d?\n", guess);

while((response = getchar()) != 'д')

{

if( response != ' \n')

{

if (response == ' 6')

{/\* уменьшение верхнего предела. Если предло-

женное значение слишком велико \*/

highest = guess - 1;

guess = (highest + lowest)/2;

printf(" Гм ... слишком велико. Ваше число %d?\n" , guess);

}

else

if (response == ' м')

{/\* увеличение нижнего предела, если предложенное значение слишком мало \*/

lowest = guess + 1;

guess = (highest + lowest)/2;

printf('Гм ... слишком мало. Ваше число %d?\n", guess);

}

else

{/\* подводите пользователя к правильному ответу \*/

printf("Я не понимаю; введите, пожалуйста, д, б");

printf(" или м.\n");

}

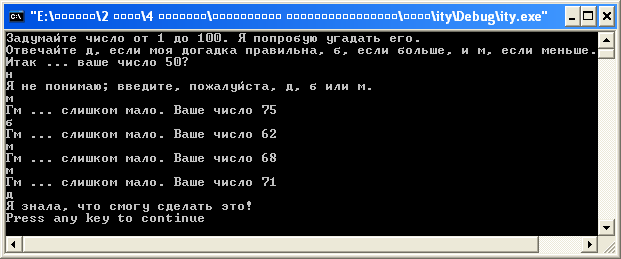
}

printf("Я знала, что смогу сделать это!\n");

}

Наличие в программе завершающей части else предоставляет пользователю дополнительную возможность правильно ответить на нестандартный «отклик» программы. Заметим также, что мы использовали символические константы, чтобы сделать процесс изменения диапазона чисел достаточно простым.

Работает ли данная программа? Ниже приводятся результаты одного прогона. Задуманное число — 71.



Что может быть неправильного в этой программе? Мы реализовали в ней защиту от ошибок, вызванных тем, что пользователи могут указывать неверные символы, поэтому здесь не должно быть никаких проблем. Единственное, что может повлиять на правильность работы программы: если вы вместо м укажете б, или наоборот. К сожалению, не существует способа заставить пользователя говорить правду и не делать ошибок. Тем не менее, если вы заинтересованы в этом, можете предпринять некоторые шаги. (Например, если захотите поразить свою шестилетнюю племянницу.) Во-первых, обратите внимание на то, что наш способ требует самое большее семи попыток для угадывания любого числа. (Каждая попытка уменьшает число возможностей наполовину. За семь попыток можно угадать любое число в диапазоне от 1 до 27- 1, или 127, что вполне достаточно для работы в диапазоне или 1 до 100.) Вы можете модифицировать программу так, чтобы она подсчитывала число попыток, и если окажется, что оно превышает 7, то тогда можно вывести на печать сообщение с выражением недовольства, а затем восстановить первоначальные значения переменных highest, lowest и счетчика. Дополнительные изменения, которые можно внести в программу, заключаются в такой модификации операторов if, в результате которой допускался бы ввод как прописных, так и строчных букв.

## Оператор break.

В С оператор break может использоваться для выхода из некоторого цикла до того, как условие проверки получит значение ЛОЖЬ. Во многом оператор break похож на оператор goto; различие в том, что точка перехода точно не известна. При выходе из цикла по команде break программа продолжается с оператора, следующего за циклом. Взгляните на очень простой пример:

/\*07BREAK.C

Программа на С, иллюстрирующая использование оператора break\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

using namespace std;

main ()

{

int itimes = 1, isum = 0;

while(itimes < 10){

isum += isum + itimes;

if(isum > 20)

break;

itimes++;

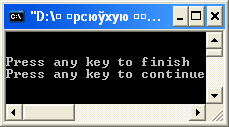
}

printf ("\n\nPress any key to finish\n"); /\* Конец работы \*/

\_getch();

return(0);

}



Бывает, что break используется для выхода из цикла в тех случаях, когда заданы два разных условия прекращения его работы. Ниже приводится цикл, реализующий эхо-печать символов и завершающийся при чтении либо признака EOF, либо символа «новая строка».

while((ch =getchar()) != EOF)

{

if (ch = = ' \n')

break;

putchar(ch);

}

Мы сделаем логику этого фрагмента программы более понятной, если объединим обе проверки в одном выражении:

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

putchar(ch);

Если вы обнаружите, что break является частью оператора if, посмотрите, нельзя ли по-другому выразить это условие (как мы только что сделали), чтобы необходимость его использования отпала.

## Оператор continue.

В С существует небольшое различие между операторами break и continue. Как вы уже видели в последнем примере, break полностью прекращает выполнение цикла. В отличие от этого, оператор continue приводит к игнорированию всех следующих за ним операторов, однако не препятствует инкременту переменной управления циклом или выполнению проверки условия продолжения цикла. Другими словами: если переменная управления циклом продолжает отвечать условию выполнения цикла, то цикл повторяется.

Эта мысль иллюстрируется следующей программой, представляющей собой игру "угадай число":

/\*07CONTNU.C

Программа на С, иллюстрирующая использование оператора continue\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

using namespace std;

#define TRUE 1

#define FALSE 0

main()

{

int ilucky\_number=77, iinput\_val, inumber\_of\_tries=0,

iam\_lucky=FALSE;

while(!iam\_lucky){

/\* Введите предполагаемое число: \*/

printf("Please enter your lucky guess: ") ;

scanf("%d",&iinput\_val);

inumber\_of\_tries++;

if(iinput\_val == ilucky\_number)

iam\_lucky=TRUE;

else

continue;

/\* Для угадывания вам понадобилось всего ... попыток! \*/

printf("It only took you %d tries to get lucky!", inumber\_of\_tries);

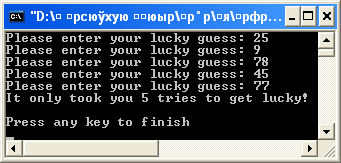
}

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}



Этот оператор может использоваться во всех трех типах циклов, но не в операторе switch. Как и в случае оператора break, он приводит к изменению характера выполнения программы. Однако вместо завершения работы цикла наличие оператора continue вызывает пропуск «оставшейся» части итерации и переход к началу следующей. Заменим оператор break в последнем фрагменте на continue:

while( (ch = getchar() ) != EOF)

{

if(ch = = ' \n')

continue;

putchar(ch);

}

В версии, использующей оператор break, работа цикла полностью прекращается, как только при вводе встречается символ «новая строка». В версии с оператором continue просто пропускаются символы «новая строка», а выход из цикла происходит, только когда читается признак EOF.

Этот фрагмент, конечно, более компактно можно записать следующим образом:

while((ch = getchar()) != EOF)

if(ch != '\n')

putchar(ch);

Очень часто, так же как и в данном случае, изменение условия в операторе if на обратное позволяет исключить необходимость введения в цикл оператора continue.

С другой стороны, оператор continue помогает иногда сократить некоторые программы, особенно если они включают в себя вложенные операторы if else.

## Совместное использование операторов break и continue.

Операторы break и continue можно использовать вместе для решения некоторых интересных задач программирования. Посмотрите на следующий пример на C++:

// 07BRACNTG.CPP

// Программа на C++, показывающая преимущества от совместного

// использования операторов break и continue

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

using namespace std;

#define NEWLINE '\n'

main ()

{

int c;

while((c=getchar())!= EOF)

{

if (isascii(c) == 0)

{

cout << "Not an ASCII character; "; // He ASCII-символ;

cout << "not going to continue/n"; // продолжения нет

break;

}

if(ispunct(c) || isspace(c)) {

putchar(NEWLINE) ;

continue;

}

if (isprint(c) == 0) {

c = getchar();

continue;

}

putchar(c);

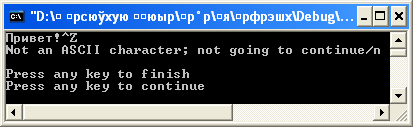
}

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}



До того, как рассмотреть принцип работы этой программы, взгляните на ее входные значения:

word control ^В exclamation' apostrophe' period.

^Z

Также посмотрите на символы, полученные на выходе:

word

control

В

exclamation

apostrophe

period

Программа считывает символы до тех пор, пока не введен символ EOF (end-of-file — конец файла) ^Z. Затем введенные символы анализируются: удаляются все непечатаемые символы и каждое "слово" помещается на отдельной строке. Для этого используются некоторые очень интересные функции, описанные в файле ctype.h, в том числе isascii(), ispunct(), isspace() и isprint(). Каждой функции в качестве параметра передается символ, а возвращается ноль или некоторое другое значение, показывающее результат сравнения.

Функция isascii() показывает, относится ли указанный символ к набору допустимых ASCII-символов; функция ispunct() — является ли символ знаком пунктуации; функция isspace() — не является ли этот символ пробелом; и наконец функция isprint() — относится ли он к печатаемым символам.

Используя эти функции, программа определяет, продолжать ли выполнение и, если продолжать, то, что необходимо делать с каждым введенным символом.

При первой проверке в цикле while выясняется, имеет ли файл символьный формат. К примеру, входные данные могли бы быть в двоичном формате, что делает программу бесполезной. В подобном случае выполняется соответствующий оператор if, печатается предупреждающее сообщение, и цикл while полностью прекращается по команде break.

Если все в порядке, выполняется второй оператор if; проверяется, не является ли введенный символ знаком пунктуации или пробелом. В любом из этих случаев выполняются входящие в if операторы: выдается символ перевода строки и выполняется оператор continue. При выполнении continue все остающиеся проверки и операторы вывода пропускаются, однако цикл продолжается. Это говорит о том, что тип символа определен правильно и нужно считывать новый символ.

Если файл имеет приемлемый формат и введенный знак не является ни знаком пунктуации, ни пробелом, то в третьем операторе if проверяется, является ли этот символ печатаемым или нет. Эта проверка отслеживает различные управляющие коды. Обратите внимание на то, что в примере ввода программы встречается управляющий символ ^В. Поскольку этот символ непечатаемый, в операторе if считывается новый символ, а затем выполняется оператор continue. Аналогичным образом, этот оператор continue говорит о том, что текущий символ проанализирован, соответствующее действие выполнено, и необходимо считывать следующий символ. Кроме того, при выполнении continue игнорируется оператор putchar(), но цикл while не заканчивается.

## Оператор goto.

Оператор goto — одно из важнейших средств Бейсика и Фортрана — также реализован и в Си. Однако на этом языке в отличие от двух других можно программировать, совершенно не используя указанное средство. Керниган и Ритчи считают оператор goto «чрезвычайно плохим» средством и предлагают «применять его как можно реже или не применять совсем».

Сначала мы покажем, как его использовать, а затем объясним, почему этого не нужно делать.

Оператор goto состоит из двух частей — ключевого слова goto и имени метки. Имена меток образуются по тем же правилам, что и имена переменных. Приведем пример записи оператора

goto part2;

Чтобы этот оператор выполнился правильно, необходимо наличие другого оператора, имеющего метку part2; в этом случае запись оператора начинается с метки, за которой следует двоеточие.

part2: printf(" Уточненный анализ:\n");

### Использование goto.

В принципе вы никогда не обязаны пользоваться оператором goto при программировании на Си. Но если ваш предыдущий опыт связан с работой на Фортране или Бейсике, в каждом из которых требуется его использовать, то у вас могли выработаться навыки программирования, основанные на применении данного оператора. Чтобы помочь вам преодолеть эту привычку, ниже вкратце приводится несколько знакомых вам ситуаций, реализуемых с помощью goto, а затем показывается, как это можно осуществить другими средствами, в большей степени соответствующими духу языка Си.

1. Работа в ситуации, когда в операторе if требуется выполнить более одного оператора:

if(size > 12)

goto a;

goto b;

a: cost = cost \* 1.05;

flag - 2;

b: bill = cost \* flag;

(В стандартных Бейсике и Фортране только один оператор, непосредственно следующий за if-условием, считается относящимся к оператору if. Мы выразим это с помощью эквивалентного фрагмента на Си.)

Обычный подход, применяемый в языке Си и заключающийся в использовании составного оператора, или блока, упрощает понимание смысла программы:

if(size > 12);

{

cost = cost \* 1.05;

flag = 2;

}

bill = cost \* flag;

1. Осуществление выбора из двух вариантов:

if(size > 14)

goto a;

sheds = 2;

goto b;

a: sheds = 3;

b: help = 2 \* sheds;

Наличие в языке Си структуры if-else позволяет реализовать такой выбор более наглядно:

if(ibex > 14)

sheds = 3;

else

sheds = 2;

help = 2 \* sheds;

1. Реализация бесконечного цикла:

readin: scanf(" %d", &score);

if(score < 0)

goto stage2;

большое количество операторов;

goto readin:

stage2: дополнительная чепуха;

Эквивалентный фрагмент, в котором используется цикл while, выглядит так:

scanf("%d", &score);

while(score > = 0)

{

большое количество операторов;

scanf(" %d", &score);

дополнительная чепуха;

1. Пропуск операторов до конца тела цикла: используйте оператор continue.
2. Выход из цикла: используйте оператор break. Фактически break и continue являются специальными формами оператора goto. Преимущество их использования заключается в том, что, во-первых, названия этих операторов говорят об осуществляемых ими функциях, а во-вторых, поскольку они не используют меток, отсутствует опасность пометить не тот оператор программы Выполнение переходов к различным частям программы непредсказуемым образом: так программировать нельзя!

Существует один случай, когда использование оператора goto допускается опытными программистами, работающими на языке Си,— это выход из вложенного набора циклов при обнаружении каких-то ошибок. (Применение оператора break дает возможность осуществить выход только из самого внутреннего цикла.)

while(funct > 0)

{

for(i = 1; i < 100; i++ )

{

for(j = 1; j<= 50; j ++ )

{

большое число операторов;

if(большая ошибка)

goto help;

операторы;

}

еще некоторое количество операторов;

}

и еще некоторое количество операторов;

}

и еще операторы;

help: устранение ошибки;

Как вы можете заметить из наших примеров, альтернативные формы представления программ более понятны, чем формы, использующие оператор goto. Эти различия станут еще большими, если вы объедините некоторые из рассмотренных случаев. Какие операторы goto используются при реализации операторов if, какие моделируют конструкции if-else, какие управляют работой циклов, а какие появляются лишь потому, что пользователь написал программу так, что не может без них обойтись? Чрезмерное увеличение оператором goto приводит к созданию лабиринта в логике программы. Дадим вам совет: если вы совсем не знакомы с оператором goto, то не применяйте его вовсе; если вы привыкли пользоваться им, попытайтесь отучить себя от этого. Ирония заключается в том, что в языке Си, который вовсе не нуждается в операторе goto, его структура оказывается лучшей, чем в большинстве других языков программирования, поскольку в качестве меток можно использовать смысловые имена, а не числа.

## Оператор exit().

В некоторых случаях программу необходимо закончить до того, как выполнились все ее операторы или условия. Для таких особых случаев в С имеется библиотечная функция exit(). Эта функция может иметь один целочисленный аргумент, называемый статусом. Операционные системы UNIX и MS-DOS интерпретируют значение статуса, равное нулю, как успешное завершение программы, а все ненулевые значения статуса говорят о различного вида ошибках.

Конкретное значение статуса, переданное функции exit(), может использоваться тем процессом, который вызвал данную программу для выполнения некоторых действий. Например, если программа вызвана из командной строки и значение статуса свидетельствует о некоторого рода ошибке, операционная система может выдать сообщение. Функция exit() кроме того, что завершает программу, выполняет также запись во все открытые для записи файлы и закрывает все открытые файлы.

В следующей программе на C++ выполняется поиск среднего значения из списка, имеющего до 30 чисел. Выход из программы происходит, если пользователь просит усреднить большее, чем LIMIT, количество цифр.

// 07EXIT1.CPP

// Программа на C++, иллюстрирующая использование функции exit

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

using namespace std;

#define LIMIT 30

main()

{

int irow,irequested\_qty,iscores[LIMIT];

float fsum=0,imax\_score=0,imin\_score=100, faverage;

// Введите количество усредняемых величин:

cout << "\nEnter the number of scores to be averaged: ";

cin >> irequested\_qty;

if(irequested\_qty > LIMIT) {

// Вы можете задать только ... чисел для усреднения,

cout << "\nYou can only enter up to " << LIMIT << \

" scores" << " to be averaged.\n";

// Программа окончена.

cout << "\n >>> Program was exited. <<<\n";

exit;

}

for(irow = 0; irow < irequested\_qty; irow++) {

// Введите число

cout << "\nPlease enter a grade " << irow+1 << ": ";

cin >> iscores[irow];

}

for(irow = 0; irow < irequested\_qty; irow++)

fsum = fsum + iscores[irow];

faverage = fsum/(float)irequested\_qty;

for(irow = 0; irow < irequested\_qty; irow++) {

if(iscores[irow] > imax\_score)

imax\_score = iscores[irow];

if(iscores[irow] < imin\_score)

imin\_score = iscores[irow];

}

cout << "\nThe maximum grade is " << imax\_score; // Максимум

cout << "\nThe minimum grade is " << imin\_score; // Минимум

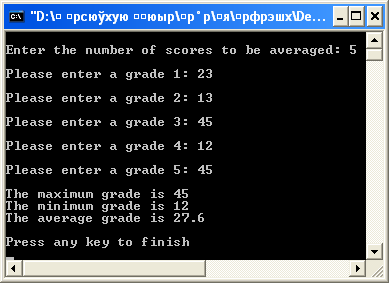
cout << "\nThe average grade is " << faverage; // Среднее значение

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}



В начало программы включен заголовочный файл process.h. Для того чтобы иметь прототип функции exit (), можно включать либо process.h, либо stdlib.h. Константа LIMIT, объявленная со значением 30, используется для определения размера целочисленного массива iscores. После того, как объявлены остальные переменные, программа запрашивает у пользователя ввод количества элементов массива iscores. В данной программе это, значение нужно напечатать следом за сообщением-подсказкой.

Полученное значение заносится в переменную irequested\_qty и используется при сравнениях в операторе if. Если пользователь пытается усреднить чисел больше, чем помещается в массив iscores, печатаются два предупреждающих сообщения и выполняется оператор exit(), в результате чего программа заканчивается.

Сможете ли вы найти два небольших отличия между предыдущей программой и следующей:

// 07EXIT2.CPP

// Программа на C++, иллюстрирующая использование функции exit

// и различия между заголовочными файлами process.h и stdlib.h

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

using namespace std;

#define LIMIT 30

main()

{

int irow,irequested\_qty,iscores[LIMIT];

float fsum=0,imax\_score=0,imin\_score=100,faverage;

cout << "\nEnter the number of scores to be averaged: ";

cin >> irequested\_qty;

if(irequested\_qty > LIMIT) {

cout << "\nYou can only enter up to " << LIMIT << \

" scores" << " to be faveraged.\n";

cout << "\n >>> Program was exited. <<<\n";

printf ("\n\nEXIT will be naw\n");

\_getch();

exit(EXIT\_FAILURE);

}

for(irow = 0; irow < irequested\_qty; irow++) {

cout << "\nPlease enter a grade " << irow+1 << ": ";

cin >> iscores[irow];

}

for(irow = 0; irow < irequested\_qty; irow++)

fsum = fsum + iscores[irow];

faverage = fsum/(float)irequested\_qty;

for(irow = 0; irow < irequested\_qty; irow++) {

if(iscores[irow] > imax\_score)

imax\_score = iscores [irow] ;

if(iscores[irow] < imin\_score)

imin\_score = iscores[irow];

}

cout << "\nThe maximum grade is " << imax\_score;

cout << "\nThe minimum grade is " << imin\_score;

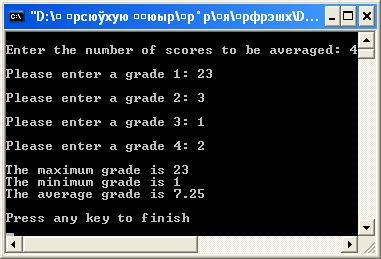
cout << "\nThe average grade is " << faverage;

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}



Благодаря тому, что вместо заголовочного файла process.h используется stdlib.h, становятся видимыми два описания: EXIT\_SUCCESS (которое возвращает значение ноль) и EXIT\_FAILURE (которое возвращает значение, сигнализирующее об ошибке). В данной программе используется описание EXIT\_FAILURE, чтобы иметь более читабельный параметр для функции exit().

## Оператор atexit().

Всякий раз, когда программа вызывает функцию exit() или происходит нормальное завершение программы, можно также вызывать любую зарегистрированную "функцию выхода", занесенную в atexit(). Следующая программа на С иллюстрирует эту возможность:

/\*07ATEXIT.C

Программа на С, показывающая зависимость между функцией

atexit и порядком выполнения объявленных функций\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

using namespace std;

void atexit\_fn1(void);

void atexit\_fn2(void);

void atexit\_fn3(void);

main()

{

atexit(atexit\_fn1);

atexit(atexit\_fn2) ;

atexit(atexit\_fn3) ;

printf("Atexit program entered.\n"); /\* Вход в программу Atexit.\*/

printf("Atexit program exited.\n\n"); /\* Выход из программы Atexit.\*/

printf (">>>>>>>>>> <<<<<<<<,\n\n") ;

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

void atexit\_fn1(void)

{

printf("atexit\_fn1 entered.\n"); /\* Вход в функцию atexit\_fn1.\*/

}

void atexit\_fn2(void)

{

printf("atexit\_fn2 entered.\n"); /\* Вход в функцию atexit\_fn2.\*/

}

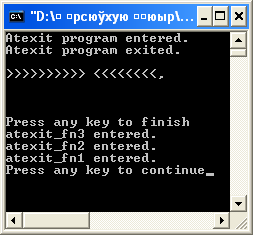
void atexit\_fn3(void)

{

printf("atexit\_fn3 entered.\n"); /\* Вход в функцию atexit\_fn3.\*/

}

На выходе программы получается следующее:



Функция atexit() использует имя некоторой функции в качестве единственного параметра и регистрирует эту функцию как функцию выхода. Если программа завершается нормально, как в предыдущем примере, или она вызывает функцию exit(), то выполняются все функции, объявленные в atexit()

С технической стороны, всякий раз, когда оператор atexit() встречается в исходном тексте программы, указанная функция добавляется к списку функций, выполняемых по завершению программы. Когда программа заканчивается, все функции, переданные в atexit() выполняются, причем последняя добавленная функция выполняется первой. Этим объясняется то, почему выходное сообщение atexit\_fn3 напечаталось раньше аналогичного сообщения atexit\_fn1. Функции в atexit() обычно используются для удаления динамически созданных объектов. Поскольку один объект (В) может быть создан на основе другого (А), функции в atexit() выполняются в обратном порядке, при этом сначала будет удален объект В, а затем А.

**Что вы должны были узнать.**

Три типа циклов в языке Си: while, for и do while.

Различие между циклами с предусловием и с постусловием.

Почему циклы с предусловием используются гораздо чаще, чем циклы с постусловием.

Дополнительные операции присваивания: += -= \*= /= %=

Как пользоваться операцией «запятая».

Когда использбвать операторы break и continue: по возможности редко.

Когда использовать оператор goto: когда вы хотите иметь неудобные, трудные для понимания программы.

Как использовать оператор while для защиты программы от ошибок при вводе данных.

Что такое массив и как его описать: long arms[8]

**Вопросы и ответы вопросы.**

1. Определите значение переменной quack после выполнения каждого оператора из приведенной ниже их последовательности.

int quack = 2;

quack + = 5;

quack \*= 10;

quack - = 6;

quack /= 8ж

quack %= 3;

1. Что будет получено на выходе в результате работы следующего цикла?

for (value = 36; value > 0; value /= 2)

printf ("%3d", value);

1. Как можно модифицировать операторы if в программе угадывание числа2 чтобы был возможен ввод как прописных, так и строчных букв?
2. Мы подозреваем, что следующая программа не совсем правильная. Какие ошибки вы сможете в ней обнаружить?

main() /\* строка 1 \*/

{ /\* строка 2 V

int i, j, hst(10); /\* строка 3 \*/

for (i = 1, i <= 10, i++ ) /\* строка 5 \*/

{ /\* строка 6 \*/

list[i] = 2\*i + 3;/\* строка 7 \*/

for (j = 1, j >= i, j++ ) /\* строка 8 \*/

printf("%d\n", list[j]); /\* строка 9 \*/

}/\* строка 10 \*/

1. Воспользуйтесь вложенными циклами при написании программы, выводящей на печать следующую фигуру:

$$$$$$$$

$$$$$$$$

$$$$$$$$

$$$$$$$$

1. Напишите программу, которая создает массив из 26 элементов и помешает в него 26 строчных букв.

**Ответы.**

2, 7, 70, 64, 8, 2

36 18 9 4 2 1 Вспомните, как выполняется деление целых чисел. Результатом деления 1 на 2 будет 0, поэтому работа цикла завершится после того, как переменная value станет равной 1.

if (response == 'б' || response == 'Б').

строка 3: должно быть list[10]

строка 5: вместо запятых должны стоять символы «точка с запятой»,

строка 5: переменная i должна изменяться в диапазоне от 0 до 9, а не от 1 до 10.

строка 8: вместо запятых должны стоять символы «точка с запятой»,

строка 8: знак > = должен быть заменен на < =. В противном случае при значении i, равном 1, цикл никогда не завершится.

строка 10: между строками 9 и 10 должна находиться еще одна закрывающая фигурная скобка. Одна скобка закрывает составной оператор, а другая тело программы.

main()

{

int i, j;

for(i = 1; i <= 4; i++ )

{

for (j = 1; j <= 8; j++ )

printf("$");

}

printf ("\n");

}

main()

{

int i;

char ch, alpha[26];

for (i = 0, ch = 'a'; i <= 26; i++ , ch++ )

alpha[i] = ch;

}

**Упражнения.**

1. Модифицируйте программу угадывание чнсла2 в соответствии с нашими предположениями об улучшении ее работы.
2. Реализуйте наше предложение о повышении эффективности работы программы нахождения простых чисел.
3. Воспользуйтесь вложенными циклами при написании программы, выводящей на печать следующую фигуру:

$

$$

$$$

$$$$

$$$$$