Introduction to C

By Alex Allain

This tutorial is designed to be a stand-alone introduction to C, even if you've never programmed before. However, because C++ is a more modern language, if you're not sure if you should learn C or C++, I recommend the C++ tutorial instead, which is also designed for people who have never programmed before. Nevertheless, if you do not desire some of C++'s advanced features or simply wish to learn C instead of C++, then this tutorial is for you!

Getting set up - finding a C compiler

The very first thing you need to do, before starting out in C, is to make sure that you have a compiler. What is a compiler, you ask? A compiler turns the program that you write into an **executable** that your computer can actually understand and run. If you're taking a course, you probably have one provided through your school. If you're starting out on your own, your best bet is to use Code::Blocks with MinGW. If you're on Linux, you can use gcc, and if you're on Mac OS X, you can use XCode. If you haven't yet done so, go ahead and get a compiler set up--you'll need it for the rest of the tutorial.

Intro to C

Every full C program begins inside a function called "main". A function is simply a collection of commands that do "something". The main function is always called when the program first executes. From main, we can call other functions, whether they be written by us or by others or use built-in language features. To access the standard functions that comes with your compiler, you need to include a header with the #include directive. What this does is effectively take everything in the header and paste it into your program. Let's look at a working program:

```
#include <stdio.h>
int main()
{
    printf( "I am alive!  Beware.\n" );
    getchar();
    return 0;
}
```

Let's look at the elements of the program. The #include is a "preprocessor" directive that tells the compiler to put code from the header called stdio.h into our program before actually creating the executable. By including header files, you can gain access to many different functions--both the printf and getchar functions are included in stdio.h.

The next important line is int main(). This line tells the compiler that there is a function named main, and that the function returns an integer, hence int. The "curly braces" ({ and }) signal the beginning and end of functions and other code blocks. If you have programmed in Pascal, you will know them as BEGIN and END. Even if you haven't programmed in Pascal, this is a good way to think about their meaning.

The printf function is the standard C way of displaying output on the screen. The quotes tell the compiler that you want to output the literal string as-is (almost). The '\n' sequence is actually treated as a single character that stands for a newline (we'll talk about this later in more detail); for the time being, just remember that there are a few sequences that, when they appear in a string literal, are actually not displayed literally by printf and that '\n' is one of them. The actual effect of '\n' is to move the cursor on your screen to the next line. Notice the semicolon: it tells the compiler that you're at the end of a command, such as a function call. You will see that the semicolon is used to end many lines in C.

The next command is getchar(). This is another function call: it reads in a single character and waits for the user to hit enter before reading the character. This line is included because many compiler environments will open a new console window, run the program, and then close the window before you can see the output. This command keeps that window from closing because the program is not done yet because it waits for you to hit enter. Including that line gives you time to see the program run.

Finally, at the end of the program, we return a value from main to the operating system by using the return statement. This return value is important as it can be used to tell the operating system whether our program succeeded or not. A return value of 0 means success.

The final brace closes off the function. You should try compiling this program and running it. You can cut and paste the code into a file, save it as a .c file, and then compile it. If you are using a command-line compiler, such as Borland C++ 5.5, you should read the compiler instructions for information on how to compile. Otherwise compiling and running should be as simple as clicking a button with your mouse (perhaps the "build" or "run" button).

You might start playing around with the printf function and get used to writing simple C programs.

Explaining your Code

Comments are critical for all but the most trivial programs and this tutorial will often use them to explain sections of code. When you tell the compiler a section of text is a comment, it will ignore it when running the code, allowing you to use any text you want to describe the real code. To create a comment in C, you surround the text with /* and then */ to block off everything between as a comment. Certain compiler environments or text editors will change the color of a commented area to make it easier to spot, but some will not. Be certain not to accidentally comment out code (that is, to tell the compiler part of your code is a comment) you need for the program.

When you are learning to program, it is also useful to comment out sections of code in order to see how the output is affected.

Using Variables

So far you should be able to write a simple program to display information typed in by you, the programmer and to describe your program with comments. That's great, but what about interacting with your user? Fortunately, it is also possible for your program to accept input.

But first, before you try to receive input, you must have a place to store that input. In programming, input and data are stored in variables. There are several different types of variables; when you tell the compiler you are declaring a variable, you must include the data type along with the name of the variable. Several basic types include char, int, and float. Each type can store different types of data.

A variable of type char stores a single character, variables of type int store integers (numbers without decimal places), and variables of type float store numbers with decimal places. Each of these variable types - char, int, and float - is each a keyword that you use when you declare a variable. Some variables also use more of the computer's memory to store their values.

It may seem strange to have multiple variable types when it seems like some variable types are redundant. But using the right variable size can be important for making your program efficient because some variables require more memory than others. For now, suffice it to say that the different variable types will almost all be used!

Before you can use a variable, you must tell the compiler about it by declaring it and telling the compiler about what its "type" is. To declare a variable you use the syntax <variable type> <name of variable>;. (The brackets here indicate that your replace the expression with text described within the brackets.) For instance, a basic variable declaration might look like this:

```
int myVariable;
```

Note once again the use of a semicolon at the end of the line. Even though we're not calling a function, a semicolon is still required at the end of the "expression". This code would create a variable called myVariable; now we are free to use myVariable later in the program.

It is permissible to declare multiple variables of the same type on the same line; each one should be separated by a comma. If you attempt to use an undefined variable, your program will not run, and you will receive an error message informing you that you have made a mistake.

Here are some variable declaration examples:

```
int x;
int a, b, c, d;
char letter;
float the_float;
```

While you can have multiple variables of the same type, you cannot have multiple variables with the same name. Moreover, you cannot have variables and functions with the same name.

A final restriction on variables is that variable declarations must come before other types of statements in the given "code block" (a code block is just a segment of code surrounded by { and }). So in C you must declare all of your variables before you do anything else:

Wrong

```
#include <stdio.h>
int main()
{
    /* wrong!  The variable declaration must appear first */
    printf( "Declare x next" );
    int x;

    return 0;
}
```

Fixed

```
#include <stdio.h>
int main()
{
    int x;
    printf( "Declare x first" );

    return 0;
}
```

Reading input

Using variables in C for input or output can be a bit of a hassle at first, but bear with it and it will make sense. We'll be using the scanf function to read in a value and then printf to read it back out. Let's look at the program and then pick apart exactly what's going on. You can even compile this and run it if it helps you follow along.

```
#include <stdio.h>

int main()
{
    int this_is_a_number;

    printf( "Please enter a number: " );
    scanf( "%d", &this_is_a_number );
    printf( "You entered %d", this_is_a_number );
    getchar();
    return 0;
}
```

So what does all of this mean? We've seen the #include and main function before; main must appear in every program you intend to run, and the #include gives us access to printf (as well as scanf). (As you might have guessed, the io in stdio.h stands for "input/output"; std just stands for "standard.") The keyword int declares this_is_a_number to be an integer.

This is where things start to get interesting: the scanf function works by taking a string and some variables modified with &. The string tells scanf what variables to look for: notice that we have a string containing only "%d" -- this tells the scanf function to read in an integer. The second argument of scanf is the variable, sort of. We'll learn more about what is going on later, but the gist of it is that scanf needs to know where the variable is stored in order to change its value. Using & in front of a variable allows you to get its location and give that to scanf instead of the value of the variable. Think of it like giving someone directions to the soda aisle and letting them go get a coca-cola instead of fetching the coke for that person. The & gives the scanf function directions to the variable.

When the program runs, each call to scanf checks its own input string to see what kinds of input to expect, and then stores the value input into the variable.

The second printf statement also contains the same '%d'--both scanf and printf use the same format for indicating values embedded in strings. In this case, printf takes the first argument after the string, the variable this_is_a_number, and treats it as though it were of the type specified by the "format specifier". In this case, printf treats this_is_a_number as an integer based on the format specifier.

So what does it mean to treat a number as an integer? If the user attempts to type in a decimal number, it will be truncated (that is, the decimal component of the number will be ignored) when stored in the variable. Try typing in a sequence of characters or a decimal number when you run the example program; the response will vary from input to input, but in no case is it particularly pretty.

Of course, no matter what type you use, variables are uninteresting without the ability to modify them. Several operators used with variables include the following: *, -, +, /, =, ==, >, <. The * multiplies, the / divides, the - subtracts, and the + adds. It is of course important to realize that to modify the value of a variable inside the program it is rather important to use the equal sign. In some languages, the equal sign compares the value of the left and right values, but in C == is used for that task. The equal sign is still extremely useful. It sets the value of the variable on the left side of the equals sign equal to the value on the right side of the equals sign. The operators that perform mathematical functions should be used on the right side of an equal sign in order to assign the result to a variable on the left side.

Here are a few examples:

```
a = 4 * 6; /* (Note use of comments and of semicolon) a is 24 */
a = a + 5; /* a equals the original value of a with five added to it */
a == 5     /* Does NOT assign five to a. Rather, it checks to see if a equals 5.*/
```

The other form of equal, ==, is not a way to assign a value to a variable. Rather, it checks to see if the variables are equal. It is extremely useful in many areas of C; for example, you will often use == in such constructions as conditional statements and loops. You can probably guess how < and > function. They are greater than and less than operators.

For example:

```
a < 5  /* Checks to see if a is less than five */
a > 5  /* Checks to see if a is greater than five */
a == 5 /* Checks to see if a equals five, for good measure */
```

Quiz yourself
Next: If Statements
Back to C Tutorial Index

Popular pages

- Jumping into C++, the Cprogramming.com ebook
- How to learn C++ or C
- C Tutorial
- C++ Tutorial
- 5 ways you can learn to program faster
- The 5 most common problems new programmers face
- How to set up a compiler
- How to make a game in 48 hours

Quiz: The basics of C

If you haven't already done so, be sure to read through Cprogramming.com's introduction to C. Otherwise, best of luck with the quiz!

1. What is the correct value to return to the operating system upon the successful completion of a program?
A. -1
B. 1
C. 0
D. Programs do not return a value.

2. What is the only function all C programs must contain?
A. start()
B. system()
C. main()
D. program()

3. What punctuation is used to signal the beginning and end of code blocks?
A. { }
B. -> and <-
C. BEGIN and END
D. (and)

4. What punctuation ends most lines of C code?
A. .
B. ;
C. :
D. '

5. Which of the following is a correct comment?
A. */ Comments */
B. ** Comment **
C. /* Comment */
D. { Comment }

6. Which of the following is not a correct variable type?
A. float
B. real
C. int
D. double

7. Which of the following is the correct operator to compare two variables?
A. :=
B. =
C. equal
D. ==

Answer Key
Next lesson: Lesson 2, if statements

Popular pages

- Jumping into C++, the Cprogramming.com ebook
- How to learn C++ or C
- C Tutorial
- C++ Tutorial
- 5 ways you can learn to program faster
- The 5 most common problems new programmers face
- How to set up a compiler
- How to make a game in 48 hours

Quiz: The basics of C

1. What is the correct value to return to the operating system upon the successful completion of a program?
A. -1
B. 1
C. 0
D. Programs do not return a value.

2. What is the only function all C programs must contain?
A. start()
B. system()
C. main()
D. program()

3. What punctuation is used to signal the beginning and end of code blocks?
A. { }
B. -> and <-
C. BEGIN and END
D. (and)

4. What punctuation ends most lines of C code?
A. .
B. ;
C. :
D. '

5. Which of the following is a correct comment?
A. */ Comments */
B. ** Comment **
C. /* Comment */
D. { Comment }

6. Which of the following is not a correct variable type?
A. float
B. real
C. int
D. double

7. Which of the following is the correct operator to compare two variables?
A. :=
B. =
C. equal
D. ==

Quiz
Next lesson: Lesson 2, if statements

Popular pages

- Jumping into C++, the Cprogramming.com ebook
- How to learn C++ or C
- C Tutorial
- C++ Tutorial
- 5 ways you can learn to program faster
- The 5 most common problems new programmers face
- How to set up a compiler
- How to make a game in 48 hours

If statements in C

By Alex Allain

The ability to control the flow of your program, letting it make decisions on what code to execute, is valuable to the programmer. The if statement allows you to control if a program enters a section of code or not based on whether a given condition is true or false. One of the important functions of the if statement is that it allows the program to select an action based upon the user's input. For example, by using an if statement to check a user-entered password, your program can decide whether a user is allowed access to the program.

Without a conditional statement such as the if statement, programs would run almost the exact same way every time, always following the same sequence of function calls. If statements allow the flow of the program to be changed, which leads to more interesting code.

Before discussing the actual structure of the if statement, let us examine the meaning of TRUE and FALSE in computer terminology. A true statement is one that evaluates to a nonzero number. A false statement evaluates to zero. When you perform comparison with the relational operators, the operator will return 1 if the comparison is true, or 0 if the comparison is false. For example, the check 0 == 2 evaluates to 0. The check 2 == 2 evaluates to a 1. If this confuses you, try to use a printf statement to output the result of those various comparisons (for example printf ("%d", 2 == 1);)

When programming, the aim of the program will often require the checking of one value stored by a variable against another value to determine whether one is larger, smaller, or equal to the other.

There are a number of operators that allow these checks.

Here are the relational operators, as they are known, along with examples:

```
>     greater than              5 > 4 is TRUE
<     less than                 4 < 5 is TRUE
>=    greater than or equal     4 >= 4 is TRUE
<=    less than or equal        3 <= 4 is TRUE
==    equal to                  5 == 5 is TRUE
!=    not equal to              5 != 4 is TRUE
```

It is highly probable that you have seen these before, probably with slightly different symbols. They should not present any hindrance to understanding. Now that you understand TRUE and FALSE well as the comparison operators, let us look at the actual structure of if statements.

Basic If Syntax

The structure of an if statement is as follows:

```
if ( statement is TRUE )
    Execute this line of code
```

Here is a simple example that shows the syntax:

```
if ( 5 < 10 )
    printf( "Five is now less than ten, that's a big surprise" );
```

Here, we're just evaluating the statement, "is five less than ten", to see if it is true or not; with any luck, it is! If you want, you can write your own full program including stdio.h and put this in the main function and run it to test.

To have more than one statement execute after an if statement that evaluates to true, use braces, like we did with the body of the main function. Anything inside braces is called a compound statement, or a block. When using if statements, the code that depends on the if statement is called the "body" of the if statement.

For example:

```
if ( TRUE ) {
    /* between the braces is the body of the if statement */
    Execute all statements inside the body
}
```

I recommend always putting braces following if statements. If you do this, you never have to remember to put them in when you want more than one statement to be executed, and you make the body of the if statement more visually clear.

Else

Sometimes when the condition in an if statement evaluates to false, it would be nice to execute some code instead of the code executed when the statement evaluates to true. The "else" statement effectively says that whatever code after it (whether a single line or code between brackets) is executed if the if statement is FALSE.

It can look like this:

```
if ( TRUE ) {
    /* Execute these statements if TRUE */
}
else {
    /* Execute these statements if FALSE */
}
```

Else if

Another use of else is when there are multiple conditional statements that may all evaluate to true, yet you want only one if statement's body to execute. You can use an "else if" statement following an if statement and its body; that way, if the first statement is true, the "else if" will be ignored, but if the if statement is false, it will then check the condition for the else if statement. If the if statement was true the else statement will not be checked. It is possible to use numerous else if statements to ensure that only one block of code is executed.

Let's look at a simple program for you to try out on your own.

```
#include <stdio.h>

int main()                            /* Most important part of the program!  */
{
    int age;                          /* Need a variable... */

    printf( "Please enter your age" );  /* Asks for age */
    scanf( "%d", &age );                 /* The input is put in age */
    if ( age < 100 ) {                  /* If the age is less than 100 */
        printf ("You are pretty young!\n" ); /* Just to show you it works... */
    }
    else if ( age == 100 ) {            /* I use else just to show an example */
        printf( "You are old\n" );
    }
    else {
        printf( "You are really old\n" );     /* Executed if no other statement is */
    }
  return 0;
}
```

More interesting conditions using boolean operators

Boolean operators allow you to create more complex conditional statements. For example, if you wish to check if a variable is both greater than five and less than ten, you could use the Boolean AND to ensure both var > 5 and var < 10 are true. In the following discussion of Boolean operators, I will capitalize the Boolean operators in order to distinguish them from normal English. The actual C operators of equivalent function will be described further along into the tutorial - the C symbols are not: OR, AND, NOT, although they are of equivalent function.

When using if statements, you will often wish to check multiple different conditions. You must understand the Boolean operators OR, NOT, and AND. The boolean operators function in a similar way to the comparison operators: each returns 0 if evaluates to FALSE or 1 if it evaluates to TRUE.

NOT: The NOT operator accepts one input. If that input is TRUE, it returns FALSE, and if that input is FALSE, it returns TRUE. For example, NOT (1) evaluates to 0, and NOT (0) evaluates to 1. NOT (any number but zero) evaluates to 0. In C NOT is written as !. NOT is evaluated prior to both AND and OR.

AND: This is another important command. AND returns TRUE if both inputs are TRUE (if 'this' AND 'that' are true). (1) AND (0) would evaluate to zero because one of the inputs is false (both must be TRUE for it to evaluate to TRUE). (1) AND (1) evaluates to 1. (any number but 0) AND (0) evaluates to 0. The AND operator is written && in C. Do not be confused by thinking it checks equality between numbers: it does not. Keep in mind that the AND operator is evaluated before the OR operator.

OR: Very useful is the OR statement! If either (or both) of the two values it checks are TRUE then it returns TRUE. For example, (1) OR (0) evaluates to 1. (0) OR (0) evaluates to 0. The OR is written as || in C. Those are the pipe characters. On your keyboard, they may look like a stretched colon. On my computer the pipe shares its key with \. Keep in mind that OR will be evaluated after AND.

It is possible to combine several Boolean operators in a single statement; often you will find doing so to be of great value when creating complex expressions for if statements. What is !(1 && 0)? Of course, it would be TRUE. It is true is because 1 && 0 evaluates to 0 and !0 evaluates to TRUE (i.e., 1).

Try some of these - they're not too hard. If you have questions about them, feel free to stop by our forums.

```
A. !( 1 || 0 )         ANSWER: 0
B. !( 1 || 1 && 0 )    ANSWER: 0 (AND is evaluated before OR)
C. !( ( 1 || 0 ) && 0 )  ANSWER: 1 (Parenthesis are useful)
```

If you find you enjoyed this section, then you might want to look more at Boolean Algebra.

Quiz yourself
Previous: The Basics of C
Next: Loops
Back to C Tutorial Index

C Programming Quiz: If statements

If you haven't already done so, be sure to read through Cprogramming.com's tutorial on If statements. Otherwise, best of luck with the quiz!

1. Which of the following is true?
A. 1
B. 66
C. .1
D. -1
E. All of the above

2. Which of the following is the boolean operator for logical-and?
A. &
B. &&
C. |
D. |&

3. Evaluate !(1 && !(0 || 1)).
A. True
B. False
C. Unevaluatable

4. Which of the following shows the correct syntax for an if statement?
A. if *expression*
B. if { *expression*
C. if (*expression*)
D. *expression* if

Answer Key
Next lesson: Lesson 3, loops

Popular pages

- Jumping into C++, the Cprogramming.com ebook
- How to learn C++ or C
- C Tutorial
- C++ Tutorial
- 5 ways you can learn to program faster
- The 5 most common problems new programmers face
- How to set up a compiler
- How to make a game in 48 hours

Quiz: The basics of C

1. What is the correct value to return to the operating system upon the successful completion of a program?
A. -1
B. 1
C. 0
D. Programs do not return a value.

2. What is the only function all C programs must contain?
A. start()
B. system()
C. main()
D. program()

3. What punctuation is used to signal the beginning and end of code blocks?
A. { }
B. -> and <-
C. BEGIN and END
D. (and)

4. What punctuation ends most lines of C code?
A. .
B. ;
C. :
D. '

5. Which of the following is a correct comment?
A. */ Comments */
B. ** Comment **
C. /* Comment */
D. { Comment }

6. Which of the following is not a correct variable type?
A. float
B. real
C. int
D. double

7. Which of the following is the correct operator to compare two variables?
A. :=
B. =
C. equal
D. ==

Quiz
Next lesson: Lesson 2, if statements

Popular pages

- Jumping into C++, the Cprogramming.com ebook
- How to learn C++ or C
- C Tutorial
- C++ Tutorial
- 5 ways you can learn to program faster
- The 5 most common problems new programmers face
- How to set up a compiler
- How to make a game in 48 hours

Loops in C

By Alex Allain

Loops are used to repeat a block of code. Being able to have your program repeatedly execute a block of code is one of the most basic but useful tasks in programming -- many programs or websites that produce extremely complex output (such as a message board) are really only executing a single task many times. (They may be executing a small number of tasks, but in principle, to produce a list of messages only requires repeating the operation of reading in some data and displaying it.) Now, think about what this means: a loop lets you write a very simple statement to produce a significantly greater result simply by repetition.

One caveat: before going further, you should understand the concept of C's true and false, because it will be necessary when working with loops (the conditions are the same as with if statements). This concept is covered in the previous tutorial. There are three types of loops: for, while, and do..while. Each of them has their specific uses. They are all outlined below.

FOR - for loops are the most useful type. The syntax for a for loop is

```
for ( variable initialization; condition; variable update ) {
  Code to execute while the condition is true
}
```

The variable initialization allows you to either declare a variable and give it a value or give a value to an already existing variable. Second, the condition tells the program that while the conditional expression is true the loop should continue to repeat itself. The variable update section is the easiest way for a for loop to handle changing of the variable. It is possible to do things like x++, x = x + 10, or even x = random (5), and if you really wanted to, you could call other functions that do nothing to the variable but still have a useful effect on the code. Notice that a semicolon separates each of these sections, that is important. Also note that every single one of the sections may be empty, though the semicolons still have to be there. If the condition is empty, it is evaluated as true and the loop will repeat until something else stops it.

Example:

```
#include <stdio.h>

int main()
{
    int x;
    /* The loop goes while x < 10, and x increases by one every loop*/
    for ( x = 0; x < 10; x++ ) {
        /* Keep in mind that the loop condition checks
           the conditional statement before it loops again.
           consequently, when x equals 10 the loop breaks.
           x is updated before the condition is checked. */
        printf( "%d\n", x );
    }
    getchar();
}
```

This program is a very simple example of a for loop. x is set to zero, while x is less than 10 it calls printf to display the value of the variable x, and it adds 1 to x until the condition is met. Keep in mind also that the variable is incremented after the code in the loop is run for the first time.

WHILE - WHILE loops are very simple. The basic structure is

while (condition) { Code to execute while the condition is true } The true represents a boolean expression which could be x == 1 or while (x != 7) (x does not equal 7). It can be any combination of boolean statements that are legal. Even, (while x ==5 || v == 7) which says execute the code while x equals five or while v equals 7. Notice that a while loop is like a stripped-down version of a for loop-- it has no initialization or update section. However, an empty condition is not legal for a while loop as it is with a for loop.

Example:

```
#include <stdio.h>

int main()
{
  int x = 0;  /* Don't forget to declare variables */

  while ( x < 10 ) { /* While x is less than 10 */
      printf( "%d\n", x );
      x++;             /* Update x so the condition can be met eventually */
  }
  getchar();
}
```

This was another simple example, but it is longer than the above FOR loop. The easiest way to think of the loop is that when it reaches the brace at the end it jumps back up to the beginning of the loop, which checks the condition again and decides whether to repeat the block another time, or stop and move to the next statement after the block.

DO..WHILE - DO..WHILE loops are useful for things that want to loop at least once. The structure is

```
do {
} while ( condition );
```

Notice that the condition is tested at the end of the block instead of the beginning, so the block will be executed at least once. If the condition is true, we jump back to the beginning of the block and execute it again. A do..while loop is almost the same as a while loop except that the loop body is guaranteed to execute at least once. A while loop says "Loop while the condition is true, and execute this block of code", a do..while loop says "Execute this block of code, and then continue to loop while the condition is true".

Example:

```
#include <stdio.h>

int main()
{
  int x;

  x = 0;
  do {
    /* "Hello, world!" is printed at least one time
      even though the condition is false */
      printf( "Hello, world!\n" );
  } while ( x != 0 );
  getchar();
}
```

Keep in mind that you must include a trailing semi-colon after the while in the above example. A common error is to forget that a do..while loop must be terminated with a semicolon (the other loops should not be terminated with a semicolon, adding to the confusion). Notice that this loop will execute once, because it automatically executes before checking the condition.

Break and Continue

Two keywords that are very important to looping are break and continue. The break command will exit the most immediately surrounding loop regardless of what the conditions of the loop are. Break is useful if we want to exit a loop under special circumstances. For example, let's say the program we're working on is a two-person checkers game. The basic structure of the program might look like this:

```
while (true)
{
    take_turn(player1);
    take_turn(player2);
}
```

This will make the game alternate between having player 1 and player 2 take turns. The only problem with this logic is that there's no way to exit the game; the loop will run forever! Let's try something like this instead:

```
while(true)
{
    if (someone_has_won() || someone_wants_to_quit() == TRUE)
    {break;}
    take_turn(player1);
    if (someone_has_won() || someone_wants_to_quit() == TRUE)
    {break;}
    take_turn(player2);
}
```

This code accomplishes what we want--the primary loop of the game will continue under normal circumstances, but under a special condition (winning or exiting) the flow will stop and our program will do something else.
Continue is another keyword that controls the flow of loops. If you are executing a loop and hit a continue statement, the loop will stop its current iteration, update itself (in the case of for loops) and begin to execute again from the top. Essentially, the continue statement is saying "this iteration of the loop is done, let's continue with the loop without executing whatever code comes after me." Let's say we're implementing a game of Monopoly. Like above, we want to use a loop to control whose turn it is, but controlling turns is a bit more complicated in Monopoly than in checkers. The basic structure of our code might then look something like this:

```
for (player = 1; someone_has_won == FALSE; player++)
    {
        if (player > total_number_of_players)
        {player = 1;}
        if (is_bankrupt(player))
        {continue;}
        take_turn(player);
    }
```

This way, if one player can't take her turn, the game doesn't stop for everybody; we just skip her and keep going with the next player's turn.

Quiz yourself
Previous: If Statements
Next: Functions
Back to C Tutorial Index

Starting out
How to begin
Get the book

Tutorials
C tutorial
C++ tutorial
Game programming
Graphics programming
Algorithms
More tutorials

Practice
Practice problems
Quizzes

Resources
Source code
C and C++ tips
Getting a compiler
Book recommendations
Forum

References
Function reference
Syntax reference
Programming FAQ

C Programming Quiz: Loops

If you haven't already done so, be sure to read through Cprogramming.com's tutorial on Loops. Otherwise, best of luck with the quiz!

1. What is the final value of x when the code int x; for(x=0; x<10; x++) {} is run?
A. 10
B. 9
C. 0
D. 1

2. In the while statement, while(x<100)..., when does the statement controlled by the condition execute?
A. When x is less than one hundred
B. When x is greater than one hundred
C. When x is equal to one hundred
D. While it wishes

3. Which is not a loop structure?
A. for
B. do while
C. while
D. repeat until

4. How many times is a do while loop guaranteed to loop?
A. 0
B. Infinitely
C. 1
D. Variable

Answer Key
Next lesson: Lesson 4, functions

Popular pages

- Jumping into C++, the Cprogramming.com ebook
- How to learn C++ or C
- C Tutorial
- C++ Tutorial
- 5 ways you can learn to program faster
- The 5 most common problems new programmers face
- How to set up a compiler
- How to make a game in 48 hours

C Programming Quiz Solutions: Loops

If you didn't do as well you as would have liked, be sure to read through Cprogramming.com's tutorial on loops in C.

1. What is the final value of x when the code int x; for(x=0; x<10; x++) {}
is run?
A. 10
B. 9
C. 0
D. 1

Note: This quiz question probably generates more email to the webmaster than any other single item on the site. Yes, the answer really is 10. If you don't understand why, think about it this way: what condition has to be true for the loop to stop running?

2. When does the code block following while(x<100) execute?
A. When x is less than one hundred
B. When x is greater than one hundred
C. When x is equal to one hundred
D. While it wishes

3. Which is not a loop structure?
A. For
B. Do while
C. While
D. Repeat Until

4. How many times is a do while loop guaranteed to loop?
A. 0
B. Infinitely
C. 1
D. Variable

Quiz
Next lesson: Lesson 4, functions

Functions in C

By Alex Allain

Now that you should have learned about variables, loops, and conditional statements it is time to learn about functions. You should have an idea of their uses as we have already used them and defined one in the guise of main. Getchar is another example of a function. In general, functions are blocks of code that perform a number of pre-defined commands to accomplish something productive. You can either use the built-in library functions or you can create your own functions.

Functions that a programmer writes will generally require a prototype. Just like a blueprint, the prototype gives basic structural information: it tells the compiler what the function will return, what the function will be called, as well as what arguments the function can be passed. When I say that the function returns a value, I mean that the function can be used in the same manner as a variable would be. For example, a variable can be set equal to a function that returns a value between zero and four.

For example:

```
#include <stdlib.h>   /* Include rand() */

int a = rand(); /* rand is a standard function that all compilers have */
```

Do not think that 'a' will change at random, it will be set to the value returned when the function is called, but it will not change again.

The general format for a prototype is simple:

```
return-type function_name ( arg_type arg1, ..., arg_type argN );
```

arg_type just means the type for each argument -- for instance, an int, a float, or a char. It's exactly the same thing as what you would put if you were declaring a variable.

There can be more than one argument passed to a function or none at all (where the parentheses are empty), and it does not have to return a value. Functions that do not return values have a return type of void. Let's look at a function prototype:

```
int mult ( int x, int y );
```

This prototype specifies that the function mult will accept two arguments, both integers, and that it will return an integer. Do not forget the trailing semi-colon. Without it, the compiler will probably think that you are trying to write the actual definition of the function.

When the programmer actually defines the function, it will begin with the prototype, minus the semi-colon. Then there should always be a block (surrounded by curly braces) with the code that the function is to execute, just as you would write it for the main function. Any of the arguments passed to the function can be used as if they were declared in the block. Finally, end it all with a cherry and a closing brace. Okay, maybe not a cherry.

Let's look at an example program:

```
#include <stdio.h>

int mult ( int x, int y );

int main()
{
  int x;
  int y;

  printf( "Please input two numbers to be multiplied: " );
  scanf( "%d", &x );
  scanf( "%d", &y );
  printf( "The product of your two numbers is %d\n", mult( x, y ) );
  getchar();
}

int mult (int x, int y)
{
  return x * y;
}
```

This program begins with the only necessary include file. Next is the prototype of the function. Notice that it has the final semi-colon! The main function returns an integer, which you should always have to conform to the standard. You should not have trouble understanding the input and output functions if you've followed the previous tutorials.

Notice how printf actually takes the value of what appears to be the mult function. What is really happening is printf is accepting the value returned by mult, not mult itself. The result would be the same as if we had use this print instead

```
printf( "The product of your two numbers is %d\n", x * y );
```

The mult function is actually defined below main. Because its prototype is above main, the compiler still recognizes it as being declared, and so the compiler will not give an error about mult being undeclared. As long as the prototype is present, a function can be used even if there is no definition. However, the code cannot be run without a definition even though it will compile.

Prototypes are declarations of the function, but they are only necessary to alert the compiler about the existence of a function if we don't want to go ahead and fully define the function. If mult were defined before it is used, we could do away with the prototype--the definition basically acts as a prototype as well.

Return is the keyword used to force the function to return a value. Note that it is possible to have a function that returns no value. If a function returns void, the return statement is valid, but only if it does not have an expression. In other words, for a function that returns void, the statement "return;" is legal, but usually redundant. (It can be used to exit the function before the end of the function.)

The most important functional (pun semi-intended) question is why do we need a function? Functions have many uses.

For example, a programmer may have a block of code that he has repeated forty times throughout the program. A function to execute that code would save a great deal of space, and it would also make the program more readable. Also, having only one copy of the code makes it easier to make changes. Would you rather make forty little changes scattered all throughout a potentially large program, or one change to the function body? So would I.

Another reason for functions is to break down a complex program into logical parts. For example, take a menu program that runs complex code when a menu choice is selected. The program would probably best be served by making functions for each of the actual menu choices, and then breaking down the complex tasks into smaller, more manageable tasks, which could be in their own functions. In this way, a program can be designed that makes sense when read. And has a structure that is easier to understand quickly. The worst programs usually only have the required function, main, and fill it with pages of jumbled code.

Quiz yourself
Previous: Loops
Next: Switch/case
Back to C Tutorial Index

Quiz: Functions

If you haven't already done so, be sure to read through Cprogramming.com's tutorial on Functions in C. Otherwise, best of luck with the quiz!

1. Which is not a proper prototype?
A. int funct(char x, char y);
B. double funct(char x)
C. void funct();
D. char x();

2. What is the return type of the function with prototype: "int func(char x, float v, double t);"
A. char
B. int
C. float
D. double

3. Which of the following is a valid function call (assuming the function exists)?
A. funct;
B. funct x, y;
C. funct();
D. int funct();

4. Which of the following is a complete function?
A. int funct();
B. int funct(int x) {return x=x+1;}
C. void funct(int) {printf("Hello");
D. void funct(x) {printf("Hello"); }

Answer Key
Next lesson: Lesson 5, switch case

Quiz: Functions

If you didn't do as well as you would have liked, be sure to read through Cprogramming.com's tutorial on functions. Otherwise, well done!

1. Which is not a proper prototype?
A. int funct(char x, char y);
B. double funct(char x)
C. void funct();
D. char x();

2. What is the return type of the function with prototype: "int func(char x, float v, double t);"
A. char
B. int
C. float
D. double

3. Which of the following is a valid function call (assuming the function exists)?
A. funct;
B. funct x, y;
C. funct();
D. int funct();

4. Which of the following is a complete function?
A. int funct();
B. int funct(int x) {return x=x+1;}
C. void funct(int) { printf("Hello");
D. void funct(x) { printf("Hello"); }

Quiz
Next lesson: Lesson 5, switch case

Popular pages

- Jumping into C++, the Cprogramming.com ebook
- How to learn C++ or C
- C Tutorial
- C++ Tutorial
- 5 ways you can learn to program faster
- The 5 most common problems new programmers face
- How to set up a compiler
- How to make a game in 48 hours

Switch case in C

By Alex Allain

Switch case statements are a substitute for long if statements that compare a variable to several "integral" values ("integral" values are simply values that can be expressed as an integer, such as the value of a char). The basic format for using switch case is outlined below. The value of the variable given into switch is compared to the value following each of the cases, and when one value matches the value of the variable, the computer continues executing the program from that point.

```
switch ( <variable> ) {
case this-value:
  Code to execute if <variable> == this-value
  break;
case that-value:
  Code to execute if <variable> == that-value
  break;
...
default:
  Code to execute if <variable> does not equal the value following any of the cases
  break;
}
```

The condition of a switch statement is a value. The case says that if it has the value of whatever is after that case then do whatever follows the colon. The break is used to break out of the case statements. Break is a keyword that breaks out of the code block, usually surrounded by braces, which it is in. In this case, break prevents the program from falling through and executing the code in all the other case statements. An important thing to note about the switch statement is that the case values may only be constant integral expressions. Sadly, it isn't legal to use case like this:

```
int a = 10;
int b = 10;
int c = 20;

switch ( a ) {
case b:
  /* Code */
  break;
case c:
  /* Code */
  break;
default:
  /* Code */
  break;
}
```

The default case is optional, but it is wise to include it as it handles any unexpected cases. It can be useful to put some kind of output to alert you to the code entering the default case if you don't expect it to. Switch statements serve as a simple way to write long if statements when the requirements are met. Often it can be used to process input from a user.

Below is a sample program, in which not all of the proper functions are actually declared, but which shows how one would use switch in a program.

```
#include <stdio.h>

void playgame()
{
    printf( "Play game called" );
}
void loadgame()
{
    printf( "Load game called" );
}
void playmultiplayer()
{
    printf( "Play multiplayer game called" );
}

int main()
{
    int input;

    printf( "1. Play game\n" );
    printf( "2. Load game\n" );
    printf( "3. Play multiplayer\n" );
    printf( "4. Exit\n" );
    printf( "Selection: " );
    scanf( "%d", &input );
    switch ( input ) {
        case 1:            /* Note the colon, not a semicolon */
            playgame();
            break;
        case 2:
            loadgame();
            break;
        case 3:
            playmultiplayer();
            break;
        case 4:
            printf( "Thanks for playing!\n" );
            break;
        default:
            printf( "Bad input, quitting!\n" );
            break;
    }
    getchar();

}
```

This program will compile, but cannot be run until the undefined functions are given bodies, but it serves as a model (albeit simple) for processing input. If you do not understand this then try mentally putting in if statements for the case statements. Default simply skips out of the switch case construction and allows the program to terminate naturally. If you do not like that, then you can make a loop around the whole thing to have it wait for valid input. You could easily make a few small functions if you wish to test the code.

Quiz yourself
Previous: Functions
Next: Pointers
Back to C Tutorial Index

Quiz 5: switch case

If you haven't already done so, be sure to read through Cprogramming.com's tutorial on Switch..Case. Otherwise, best of luck with the quiz!

1. Which follows the case statement?
A. :
B. ;
C. -
D. A newline

2. What is required to avoid falling through from one case to the next?
A. end;
B. break;
C. Stop;
D. A semicolon.

3. What keyword covers unhandled possibilities?
A. all
B. contingency
C. default
D. other

4. What is the result of the following code?

```
int x=0;

switch(x)

{

  case 1: printf( "One" );

  case 0: printf( "Zero" );

  case 2: printf( "Hello World" );

}
```

A. One
B. Zero
C. Hello World
D. ZeroHello World

Answer Key
Next lesson: Lesson 6, pointers

Quiz 5: switch case Solutions

If you didn't do as well as you like, be sure to read through Cprogramming.com's tutorial on Switch..Case. Otherwise, congratulations!

1. Which follows the case statement?
A. :
B. ;
C. -
D. A newline

2. What is required to avoid falling through from one case to the next?
A. end;
B. break;
C. Stop;
D. A semicolon.

3. What keyword covers unhandled possibilities?
A. all
B. contingency
C. default
D. other

4. What is the result of the following code?

```
int x=0;

switch(x)

{

  case 1: printf( "One" );

  case 0: printf( "Zero" );

  case 2: printf( "Hello World" );

}
```

A. One
B. Zero
C. Hello World
D. ZeroHello World

Quiz
Next lesson: Lesson 6, pointers

Popular pages

- Jumping into C++, the Cprogramming.com ebook
- How to learn C++ or C
- C Tutorial
- C++ Tutorial
- 5 ways you can learn to program faster
- The 5 most common problems new programmers face
- How to set up a compiler
- How to make a game in 48 hours