How to Learn to Program in C

C is one of the older programming languages. It was developed in the 70s, but it is still very powerful thanks to how low-level it is. Learning C is a great way to introduce yourself to more complex languages as well, and the knowledge you...

Part 1 of 6:

Getting Ready

1. 

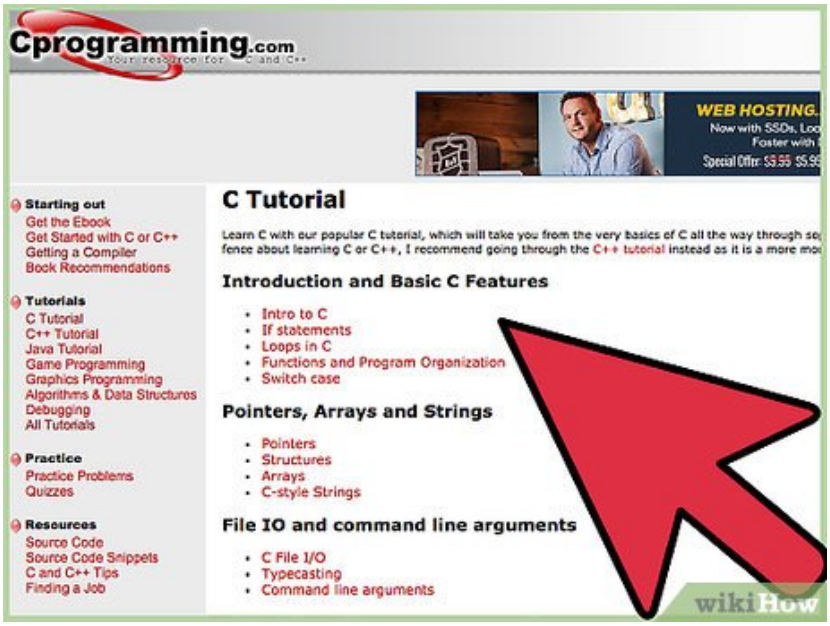
Download and install a compiler. C code needs to be compiled by a program that interprets the code into signals that the machine can understand. Compilers are usually free, and different compilers are available for different operating systems.

1. For Windows, try Microsoft Visual Studio Express or MinGW.
2. For Mac, XCode is one of the best C compilers.
3. For Linux, gcc is one of the most popular options.

2. 

Understand the basics. C is one of the older programming languages, and can be very powerful. It was designed for Unix operating systems, but has been ported and expanded for nearly all operating systems. The modern version of C is C++.

1. C is essentially comprised of functions, and in these functions you can use variables, conditional statements, loops to store and manipulate data.

3. 

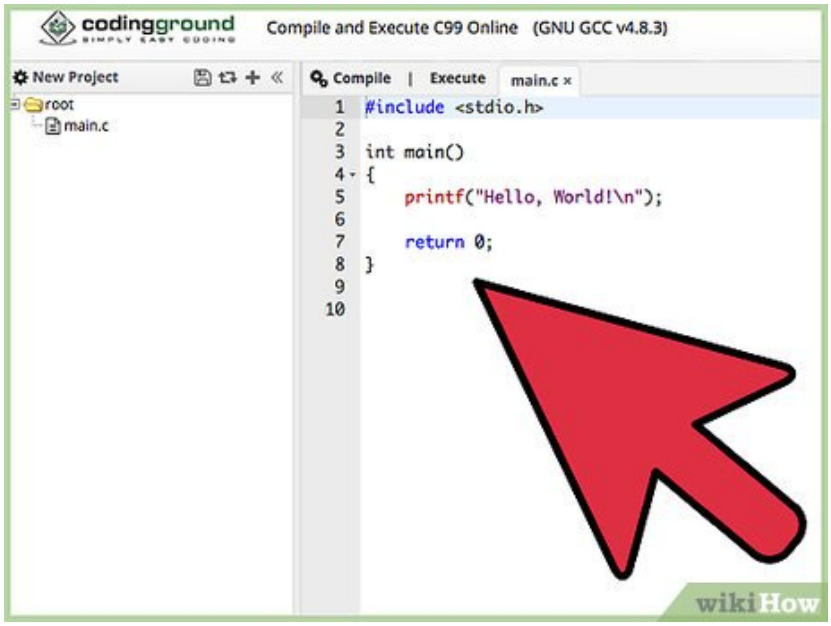
Examine some basic code. Take a look at the (very) basic program below to get a good idea about how some of the various aspects of the language work together, and to get an idea of how programs function.

```
#include int main() { printf("Hello, World!n"); getchar(); return 0; }
```

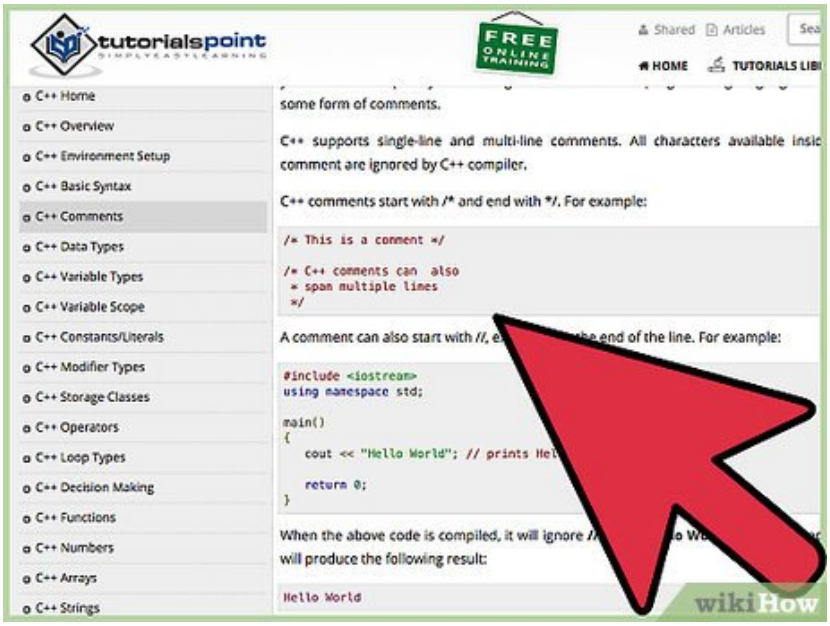
1. The `#include` command occurs before the program starts, and loads libraries that contain the functions you need. In this example, `stdio.h` lets us use the `printf()` and `getchar()`

functions.

2. The `int main()` command tells the compiler that the program is running the function called "main" and that it will return an integer when it is finished. All C programs run a "main" function.
3. The `{ }` indicate that everything inside them is part of the function. In this case, they denote that everything inside is a part of the "main" function.
4. The `printf()` function displays the contents of the parentheses on the user's screen. The quotes ensure that the string inside is printed literally. The `n` sequence tells the compiler to move the cursor to the next line.
5. The `;` denotes the end of a line. Most lines of C code need to end with a semicolon.
6. The `getchar()` command tells the compiler to wait for a keystroke input before moving on. This is useful because many compilers will run the program and immediately close the window. This keeps the program from finishing until a key is pressed.
7. The `return 0` command indicates the end of the function. Note how the "main" function is an `int` function. This means that it will need an integer to be returned once the program is finished. A "0" indicates that the program has performed correctly; any other number will mean that the program ran into an error.

4. 

Try compiling the program. Enter the code into your code editor and save it as a "*.c" file. Compile it in your compiler, typically by clicking the Build or Run button.

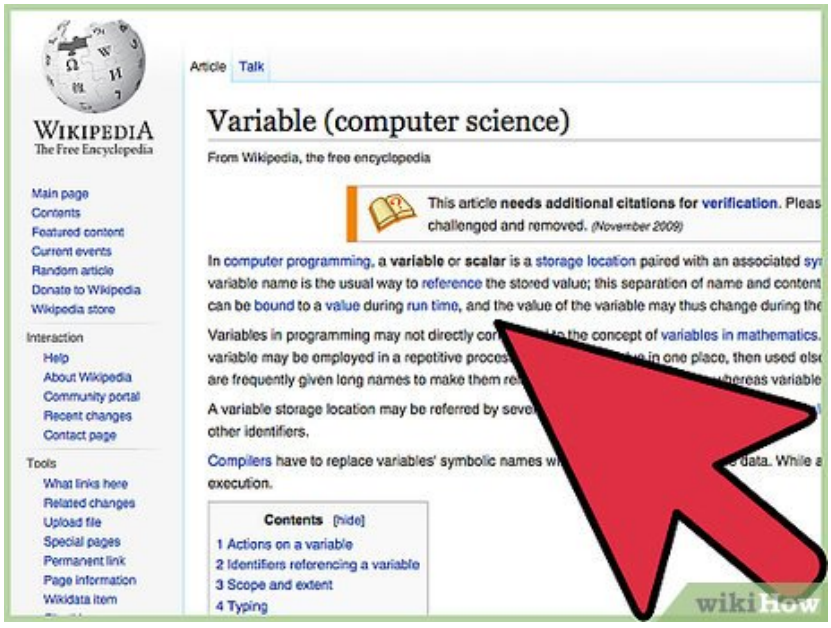
5. 

Always comment on your code. Comments are part of the code that is not compiled, but allows you to explain what is happening. This is useful for reminding yourself what your code is for, and for helping other developers who might be looking at your code.

1. To comment in C place `/*` at the start of the comment and `*/` at the end.
2. Comment on all but the most basic parts of your code.
3. Comments can be used to quickly remove parts of your code without deleting them. Simply enclose the code you want to exclude with comment tags and then compile. If you want to add the code back, remove the tags.

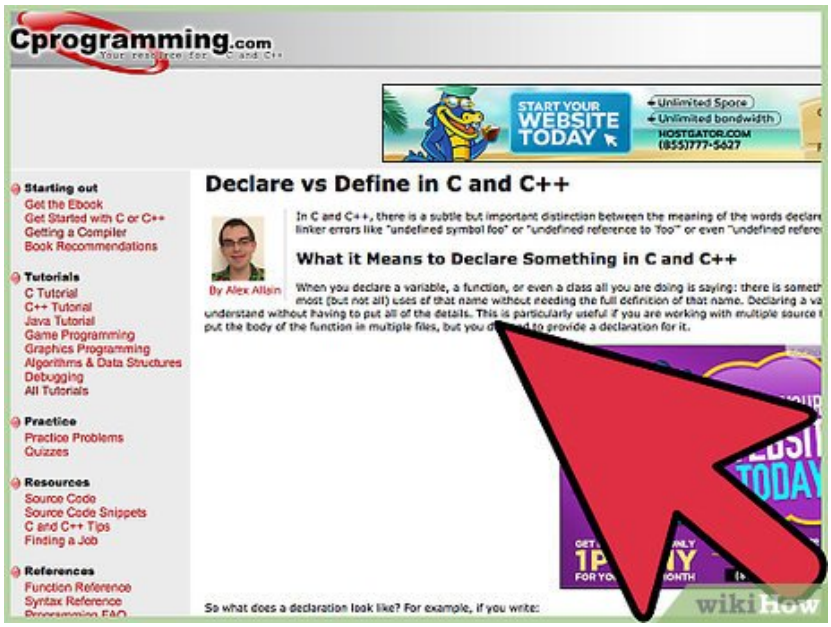
Part 2 of 6:

Using Variables

1. 

Understand the function of variables. Variables allow you to store data, either from computations in the program or from user input. Variables need to be defined before you can use them, and there are several types to choose from.

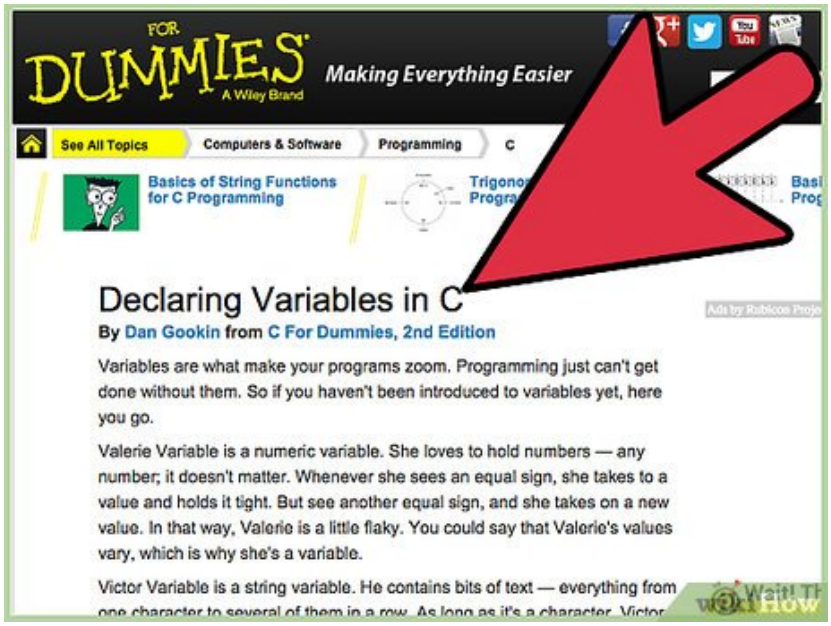
1. Some of the more common variable types include `int`, `char`, and `float`. Each one stores a different type of data.

2. 

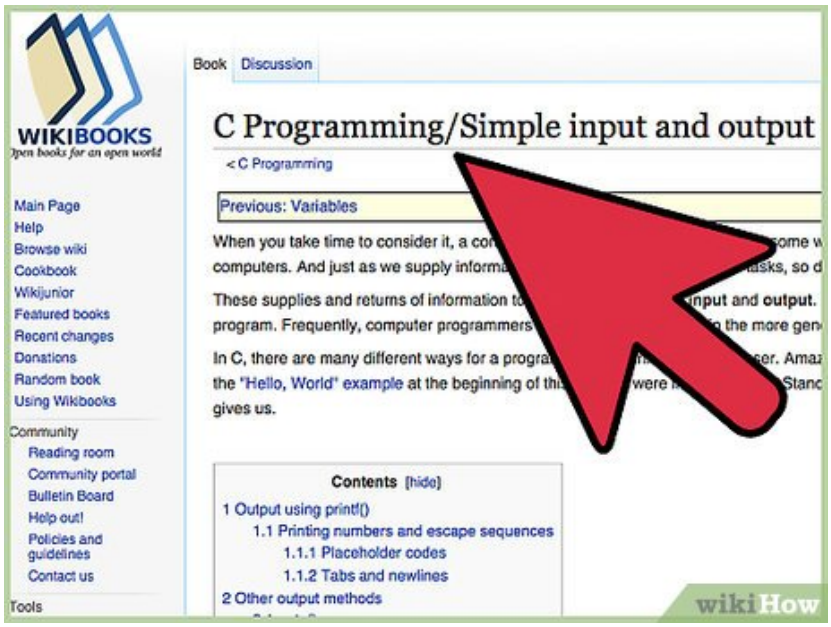
Learn how variables are declared. Variables need to be established, or "declared", before they can be used by the program. You declare a variable by entering the data type followed by the variable's name. For example, the following are all valid variable declarations:

```
float x; char name; int a, b, c, d;
```

1. Note that you can declare multiple variables on the same line, as long as they are the same type. Simply separate the variable names with commas.
2. Like many lines in C, each variable declaration line needs to end with a semicolon.

3. 

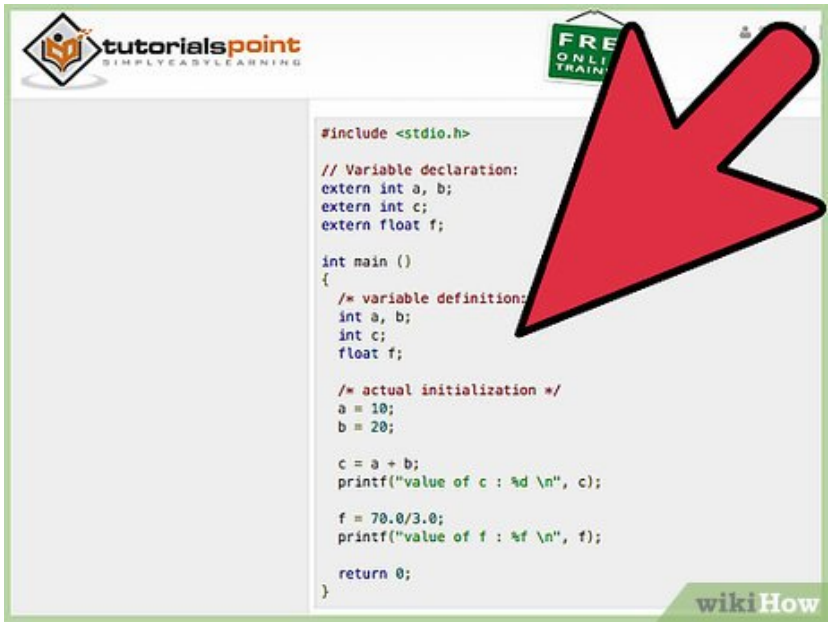
Know where to declare variables. Variables must be declared at the beginning of each code block (The parts of your code that are enclosed in {} brackets). If you try to declare a variable later in the block, the program will not function correctly.

4. 

Use variables to store user input. Now that you know the basics of how variables work, you can write a simple program that will store the user's input. You will be using another function in the program, called `scanf`. This function searches the input that is provided for specific values.

```
#include int main() { int x; printf( "Enter a number: " ); scanf( "%d",
 &x ); printf( "You entered %d", x ); getchar(); return 0; }
```

1. The `"%d"` string tells `scanf` to look for integers in the user input.
2. The `&` before the variable *x* tells `scanf` where to find the variable in order to change it, and stores the integer in the variable.
3. The final `printf` command reads back the input integer to the user.

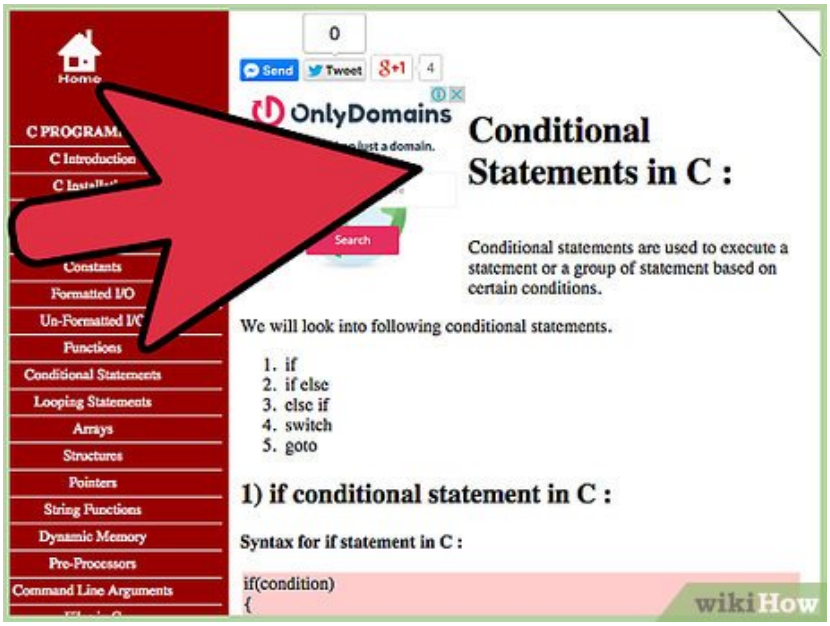
5. 

Manipulate your variables. You can use mathematical expressions to manipulate the data that you have stored in your variables. The most important distinction to remember for mathematical expressions is that a single `=` sets the value of the variable, while `==` compares the values on either side to see if they are equal.

```
x = 3 * 4; /* sets "x" to 3 * 4, or 12 */ x = x + 3;
/* adds 3 to the original value of "x", and sets the new value as the variab
 x == 15; /* checks to see if "x" equals 15 */ x  10;
/* checks if the value of "x" is less than 10 */
```

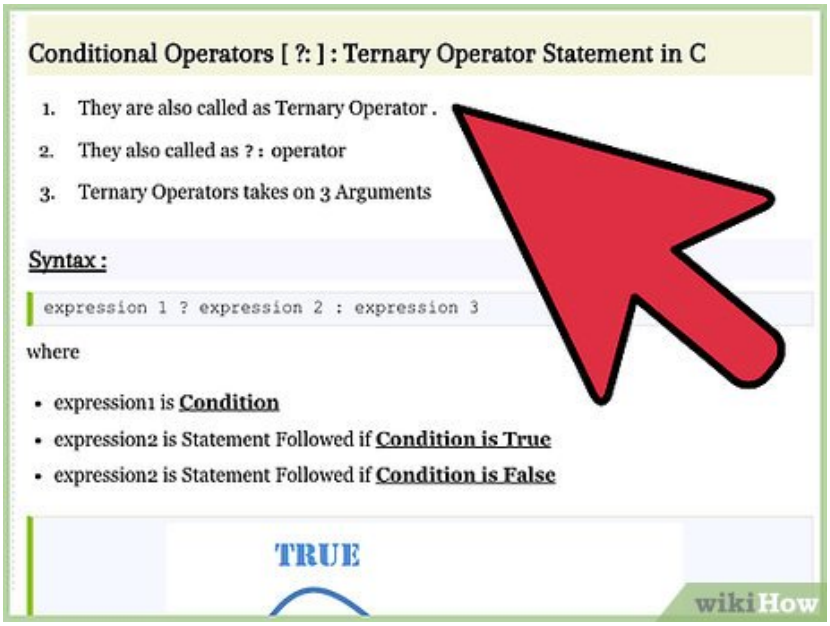
Part 3 of 6:

Using Conditional Statements

1. 

Understand the basics of conditional statements. Conditional statements are what drive most programs. They are statements that are determined to be either TRUE or FALSE, and then acted upon based on the result. The most basic of the statements is the `if` statement.

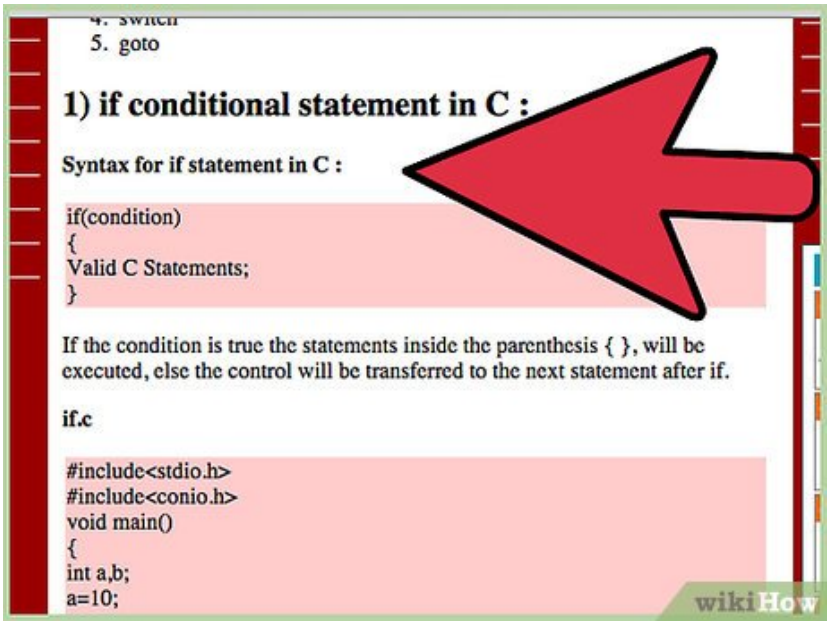
1. TRUE and FALSE work differently in C than what you might be used to. TRUE statements always end up equaling a nonzero number. When you perform comparisons, if the result is TRUE then a "1" is returned. If the result is FALSE, then a "0" is returned. Understanding this will help you see how IF statements are processed.

2. 

Learn the basic conditional operators. Conditional statements revolve around the use of mathematical operators that compare values. The following list contains the most commonly used conditional operators.

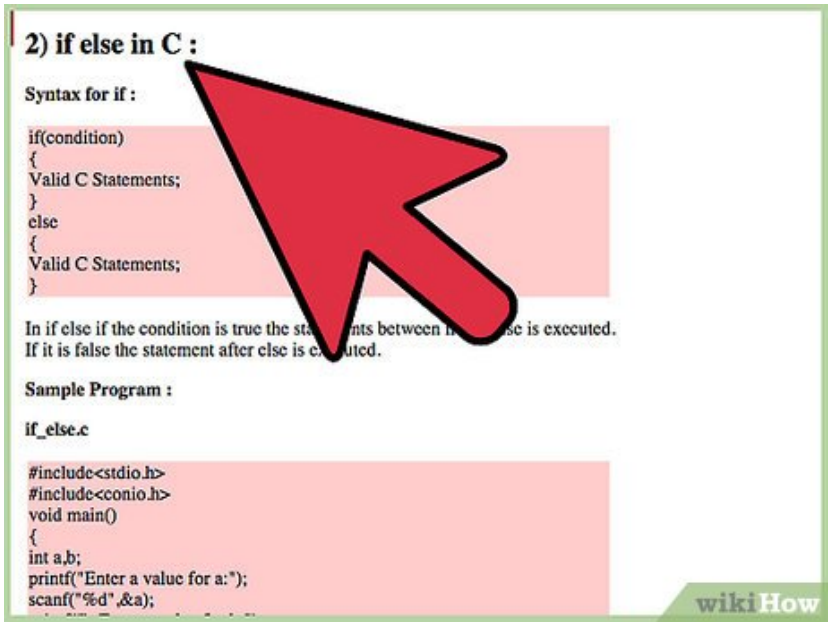
```
> /* greater than */  /* less than */ >= /* greater than or equal to */
 = /* less than or equal to */ == /* equal to */ != /* not equal to */
```

```
10 > 5 TRUE 6  15 TRUE 8 >= 8 TRUE 4 = 8 TRUE 3 == 3 TRUE 4 != 5 TRUE
```

3. 

Write a basic IF statement. You can use IF statements to determine what the program should do next after the statement is evaluated. You can combine it with other conditional statements later to create powerful multiple options, but for now write a simple one to get used to them.

```
#include int main() { if ( 3  5 ) printf( "3 is less than 5"); getchar
(); }
```

4. 

Use ELSE/ELSE IF statements to expand your conditions. You can build upon IF statements by using ELSE and ELSE IF statements to handle different results. ELSE statements run if the IF statement is FALSE. ELSE IF statements allow you to include multiple IF statements into one code block to handle various cases. See the example program below to see how they interact.

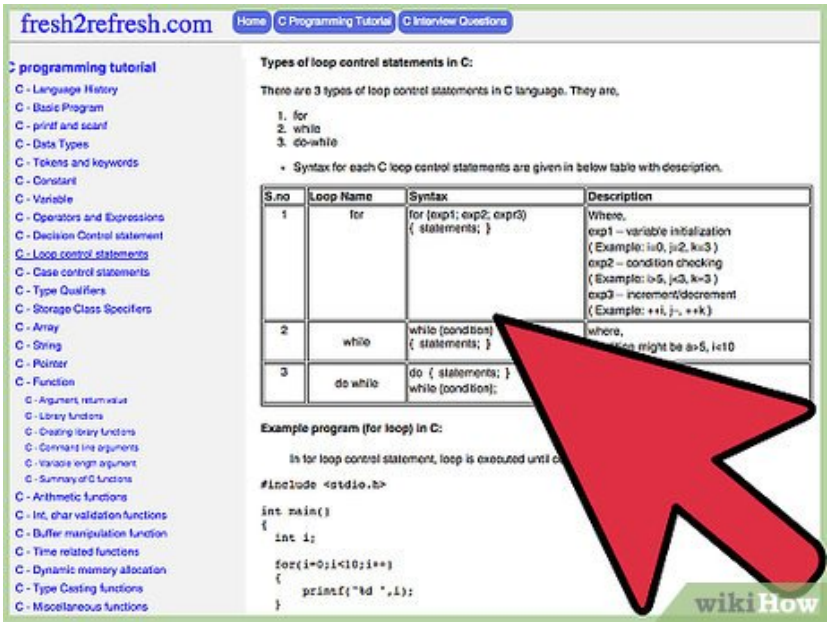
```
#include int main() { int age; printf(
"Please enter your current age: " ); scanf( "%d", &age ); if ( age = 12
 ) { printf( "You're just a kid!n" ); } else if ( age  20 ) { printf(
"Being a teenager is pretty great!n" ); } else if ( age  40 ) { printf(
 "You're still young at heart!n" ); } else { printf(
"With age comes wisdom.n" ); } return 0; }
```

[1]

1. The program takes the input from the user and takes it through the IF statements. If the number satisfies the first statement, then the first `printf` statement is returned. If it does not satisfy the first statement, it is taken through each ELSE IF statement until it finds one that works. If it doesn't match any of them, it goes through the ELSE statement at the end.

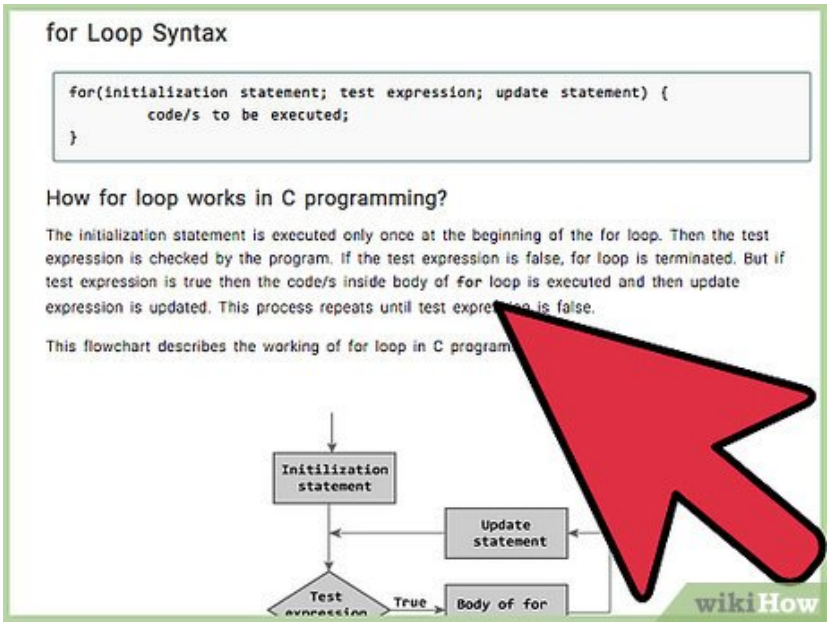
Part 4 of 6:

Learning Loops

1. 

Understand how loops work. Loops are one of the most important aspects of programming, as they allow you to repeat blocks of code until specific conditions are met. This can make repeating actions very easy to implement, and keeps you from having to write new conditional statements each time you want something to happen.

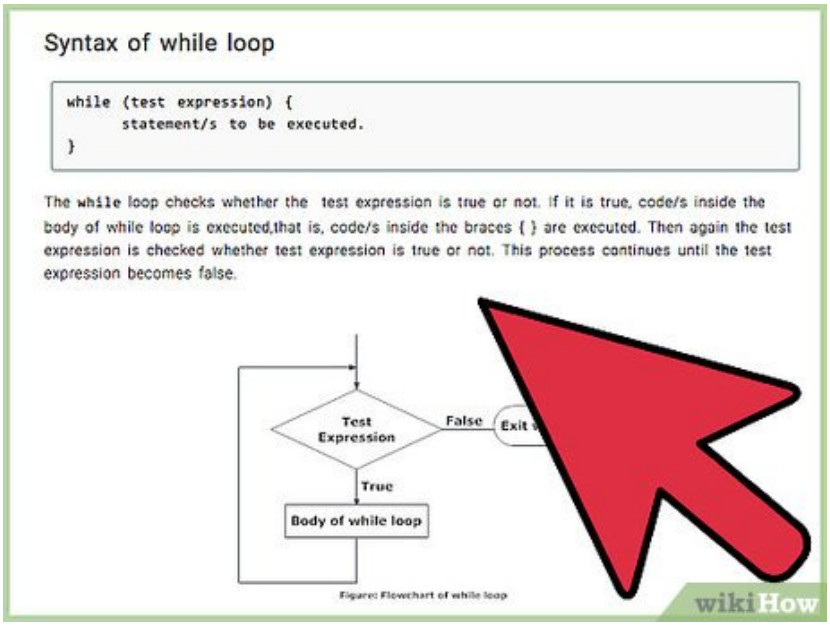
1. There are three main types of loops: FOR, WHILE, and DO...WHILE.

2. 

Use a FOR loop. This is the most common and useful loop type. It will continue running the function until the conditions set in the FOR loop are met. FOR loops require three conditions: initializing the variable, the condition to be met, and the way the variable is updated. If you don't need all of these conditions, you will still need to leave a blank space with a semicolon, otherwise the loop will run forever. [2]

```
#include int main() { int y; for ( y = 0; y  15; y++;){ printf( "%dn", y ); } getchar(); }
```

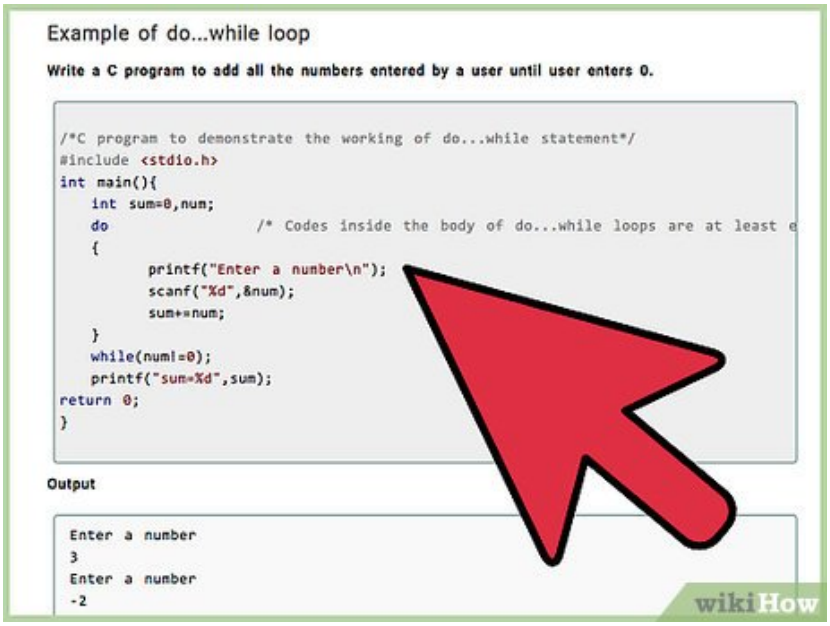
1. In the above program, *y* is set to 0, and the loop continues as long as the value of *y* is less than 15. Each time the value of *y* is printed, 1 is added to the value of *y* and the loop is repeated. Once *y* = 15, the loop will break.

3. 

Use a WHILE loop. WHILE loops are more simple than FOR loops. They only have one condition, and the loop acts as long as that condition is true. You do not need to initialize or update the variable, though you can do that in the main body of the loop.

```
#include int main() { int y; while ( y = 15 ){ printf( "%dn", y ); y++; } getchar(); }
```

1. The `y++` command adds 1 to the *y* variable each time the loop is executed. Once *y* hits 16 (remember, this loop goes as long as *y* is less than *or equal to* 15), the loop breaks.

4. 

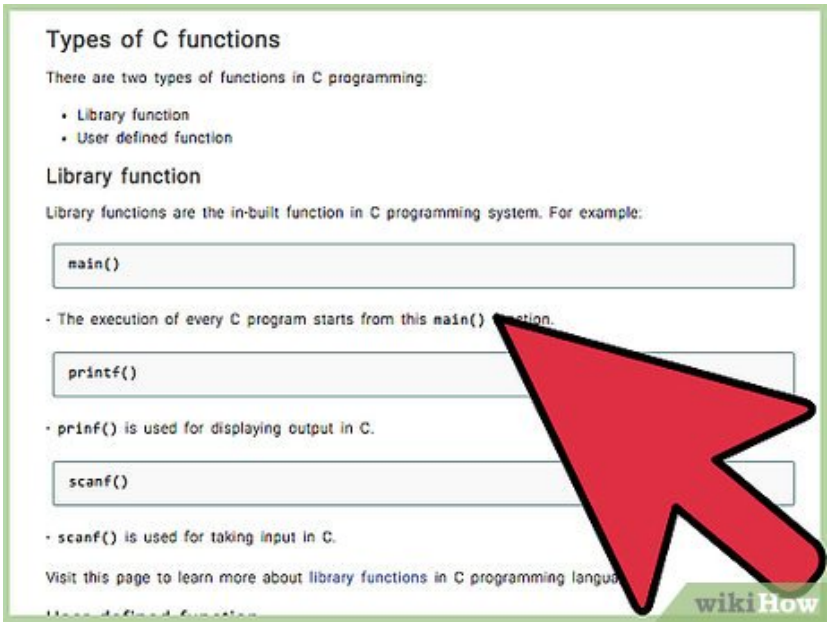
Use a DO...WHILE loop. This loop is very useful for loops that you want to ensure run at least once. In FOR and WHILE loops, the condition is checked at the beginning of the loop, meaning it could not pass and fail immediately. DO...WHILE loops check conditions at the end of the loop, ensuring that the loop executes at least once.

```
#include int main() { int y; y = 5; do { printf("This loop is running!n
"); } while ( y != 5 ); getchar(); }
```

1. This loop will display the message even though the condition is FALSE. The variable *y* is set to 5 and the WHILE loop is set to run when *y* does not equal 5, so the loop terminates. The message was already printed since the condition is not checked until the end.
2. The WHILE loop in a DO...WHILE set must be ended with a semicolon. This is the only time a loop is ended with a semicolon.

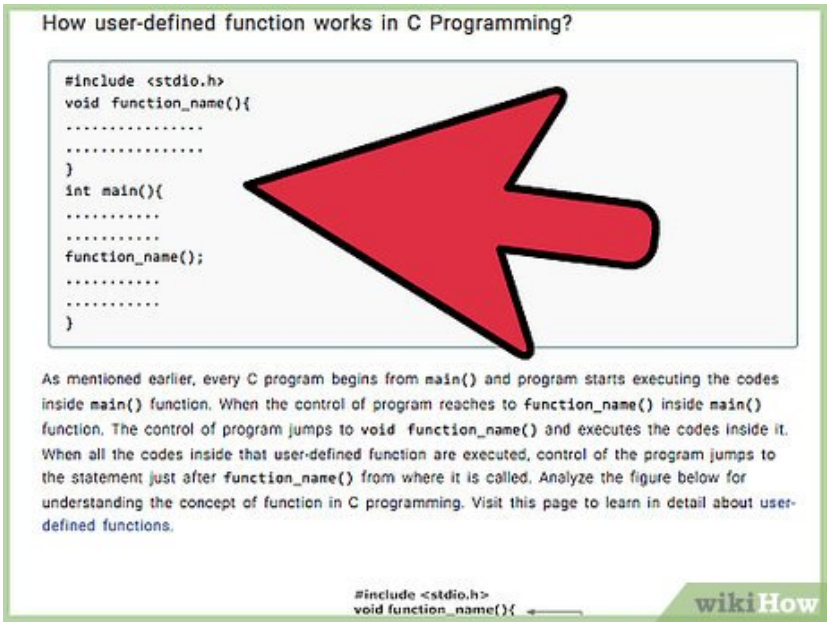
Part 5 of 6:

Using Functions

1. 

Understand the basics of functions. Functions are self-contained blocks of code that can be called upon by other parts of the program. They make it very easy to repeat code, and they help make the program simpler to read and change. Functions can include all of the previously-covered techniques learned in this article, and even other functions.

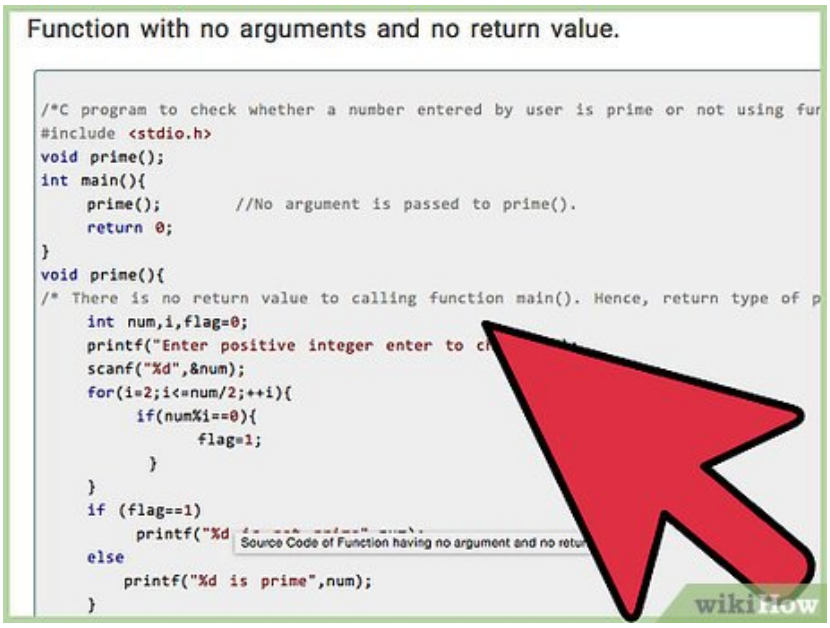
1. The `main()` line at the beginning of all of the above examples is a function, as is `getchar()`
2. Functions are essential to efficient and easy-to-read code. Make good use of functions to streamline your program.

2. 

Start with an outline. Functions are best created when you outline what you want it to accomplish before you begin the actual coding. The basic syntax for functions is "return_type name (argument1, argument2, etc.);". For example, to create a function that adds two numbers:

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

1. This will create a function that adds two integers (*x* and *y*) and then returns the sum as an integer.

3. 

Add the function to a program. You can use the outline to create a program that takes two integers that the user enters and then adds them together. The program will define how the "add" function works and use it to manipulate the input numbers.

```
#include int add ( int x, int y ); int main() { int x; int y; printf( "Enter two numbers to add together: " ); scanf( "%d", &x ); scanf( "%d" , &y ); printf( "The sum of your numbers is %dn" , add( x, y ) ); getchar(); } int add ( int x , int y ) { return x + y; }
```

1. Note that the outline is still located at the top of the program. This tells the compiler what to expect when the function is called and what it will return. This is only necessary if you want to define the function later in the program. You could define `add()` before the `main()` function and the result would be the same without the outline.
2. The actual functionality of the function is defined at the bottom of the program. The `main()` function collects the integers from the user and then sends them to the `add()` function to be processed. The `add()` function then returns the results to `main()`
3. Now the `add()` has been defined, it can be called anywhere in the program.

Part 6 of 6:

Continuing to Learn

1. 

Find a few C programming books. This article covers the basics, but it only scratches the surface of C programming and all the associated knowledge. A good reference book will help you solve problems and save you from a lot of headaches down the road.

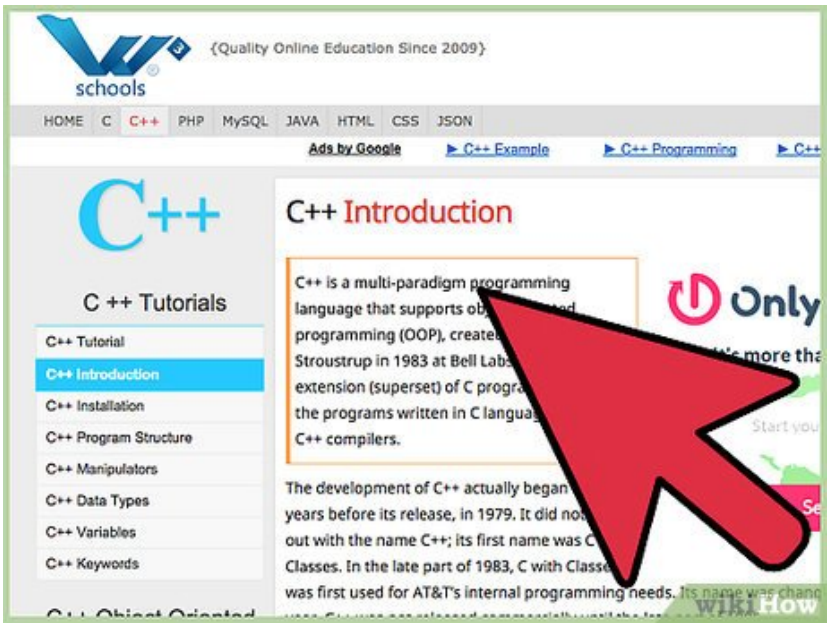
2. 

Join some communities. There are lots of communities, both online and in the real world, dedicated to programming and all of the languages that entails. Find some like-minded C programmers to swap code and ideas with, and you will soon find yourself learning a lot.

1. Attend some hack-a-thons if possible. These are events where teams and individuals have time limits to come up with programs and solutions, and often foster a lot of creativity. You can meet a lot of good programmers this way, and hack-a-thons happen regularly across the globe.

3. 

Take some classes. You don't have to go back to school for a Computer Science degree, but taking a few classes can do wonders for your learning. Nothing beats hands-on help from people who are well-versed in the language. You can often find classes at local community centers and junior colleges, and some universities will allow you to audit their computer science programs without having to enroll.

4. 

Consider learning C++. Once you have a grasp of C, it wouldn't hurt to start taking a look at C++. This is the more modern version of C, and allows for a lot more flexibility. C++ is designed with object handling in mind, and knowing C++ can enable you to create powerful programs for virtually any operating system.