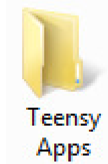
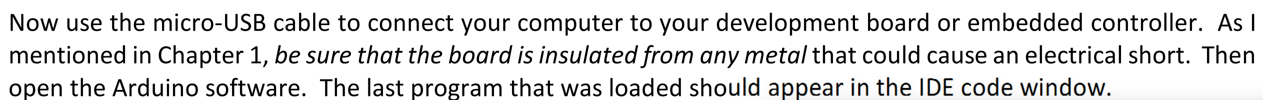
Chapter 2. Serial Output and Introduction to Functions

2

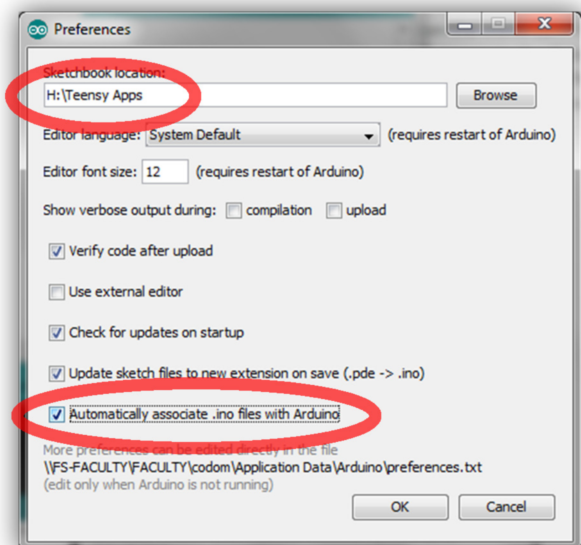
2.1. A Teensy Bit of Organizing



Now that you've gathered the materials, spend a moment setting up the working environment on your computer. Create a folder in which to save your work and store the programs you write. I recommended using this folder to store *all* your files and sketches. You can save the folder just about anywhere as long as the folder is permanent. However, I recommend that you do **not** save your work in the Arduino application folder. Trust me on this. For my purposes, I created a folder on my network drive and named it **"Teensy Apps"**. Arduino Due users may prefer a different name.



Now use the micro-USB cable to connect your computer to your development board or embedded controller. As I mentioned in Chapter 1, *be sure that the board is insulated from any metal* that could cause an electrical short. Then open the Arduino software. The last program that was loaded should appear in the IDE code window.



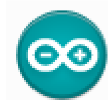
You can set the default location for your sketches by selecting **File >> Preferences** from the menu bar. Press the **Browse button** and locate the applications folder you just created. When you've done that, the path will appear in the Sketchbook location textbox as shown in in the figure to the right.

Windows users may also want to automatically associate **.ino** files with Arduino by checking the box as I have. Doing so allows you to quickly open the code file with Arduino by double-clicking on the file.[1]

Notice that you can also change the text editor's font size here. This is useful when projecting your code to an audience or classroom of students.

2.2. Hello World! Taking Your Microcontroller for a Test Drive

"Start Your Engines!" – Powering up the Development Board



Are you ready to make your very own Arduino sketch? Historically, first bit of code written when testing any new programming environment is the famous "Hello World" application, which displays a simple welcome message to the computer screen. And who are we to argue with history? Because we want to make a new program, press the **New button** on the menu bar, as shown in Figure xxx.

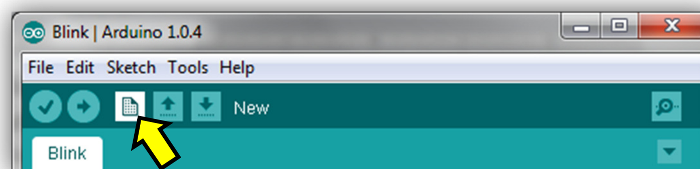


Figure xxx. Create a new sketch by clicking on the **New button**.

[1] Windows users, if you ever **upgrade** to a new version of Arduino, you may need to uninstall before installing the new software. Or you can delete the old Arduino application folder from the system registry in HKEY_CLASSES_ROOT\Applications\arduino.exe. Be extra careful when messing around with registry keys!

A new, bare-bones program window will open and should look like the one in Figure xxx below. Save (and name) your new program now by pressing the **Save button** highlighted in the figure below. Because I grew up in a time when computers were new and crashes were commonplace, I developed the habit of saving my programs often. I recommend you do the same![2]

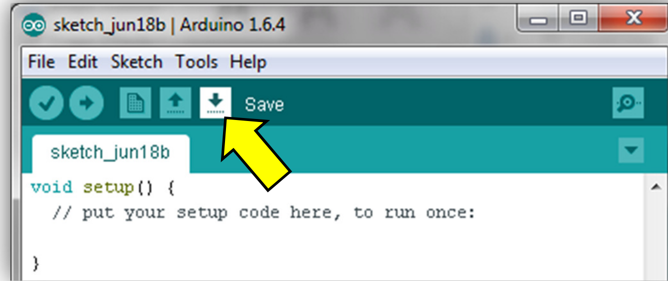


Figure xxx. Save your work early and often. It's easy with the **Save button**.

In the **Save** dialog window that opens, navigate to your application folder. Make sure that your application folder is selected as shown below. If not, navigate to your folder. Recall mine was named "Teensy Apps". Also in this window, enter the name of your sketch in the **File name** textbox. I named mine "Hello World" as shown in the figure below. You can name it anything you want, but it must **not** begin with a number. (Some other characters are also forbidden.) Any white spaces in the name will automatically be replaced by the underscore (_). Finally, press the **Save button**.

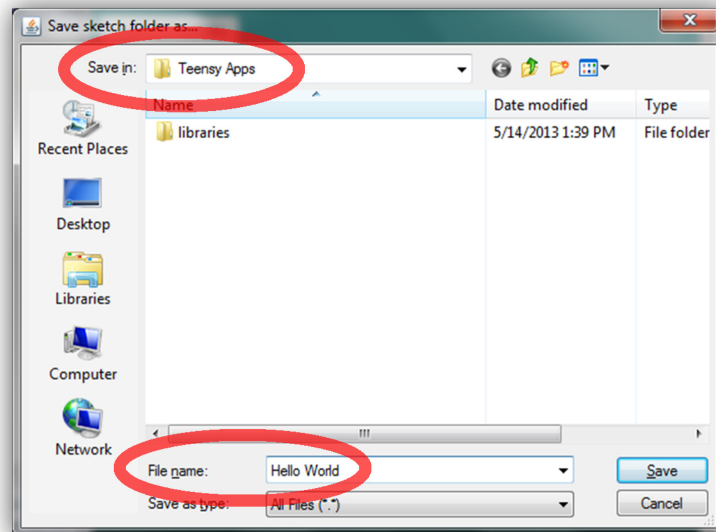
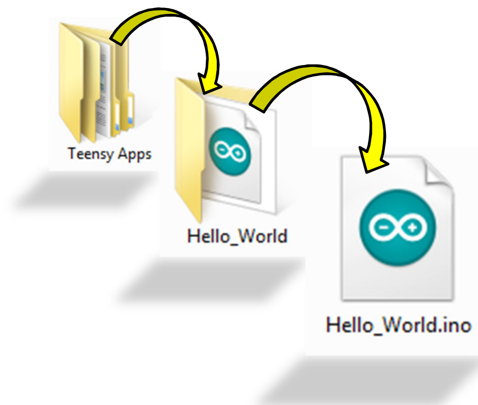


Figure xxx. When saving projects with the Arduino IDE, a folder for each project is created and the code file is saved inside the newly created folder.

Examine your application folder and you'll see a new "**Hello_World**" folder has been created. Open that folder and you'll see that it contains the "**Hello_World.ino**" code file.



[2] Starting with version 1.6, the Arduino IDE automatically saves your sketch whenever a program is uploaded to the development board!

Before you start writing code, take a moment to identify the various components of the Arduino IDE:

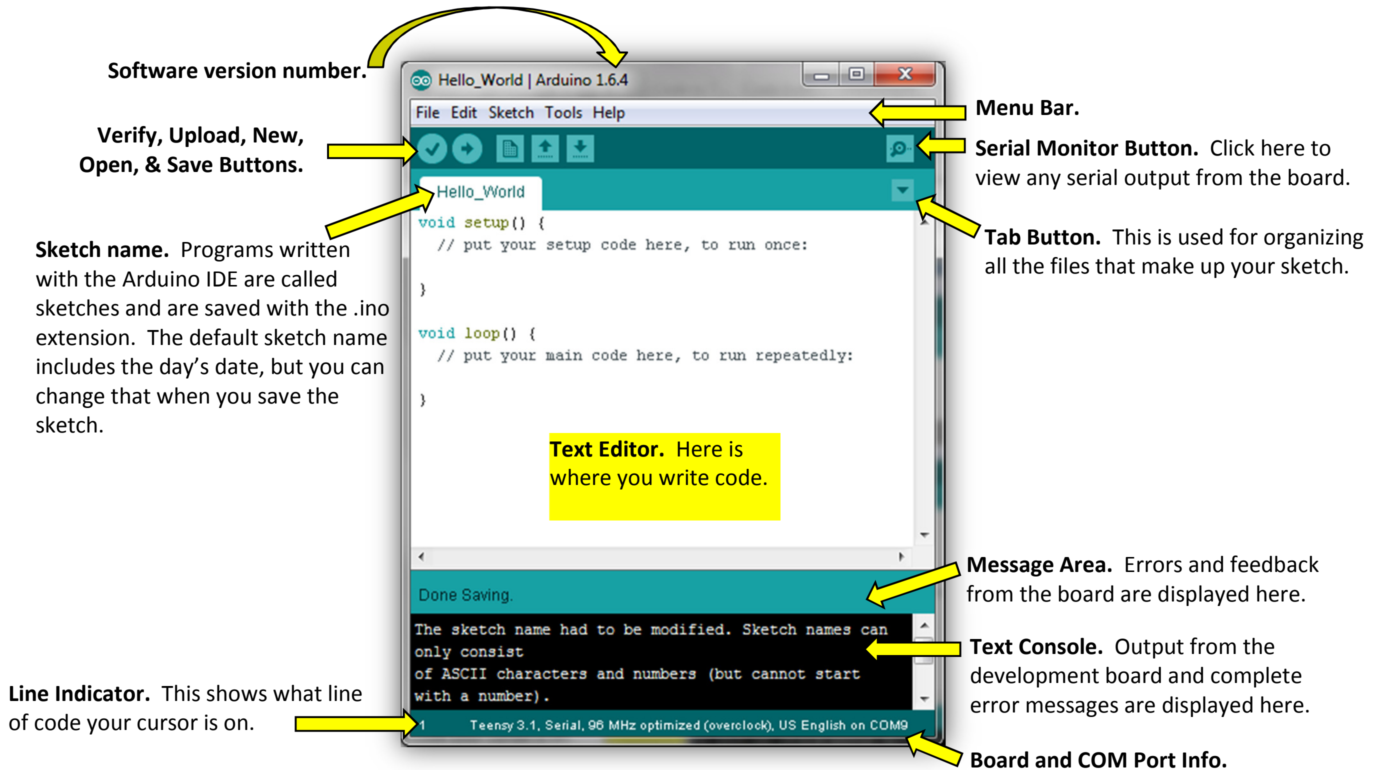
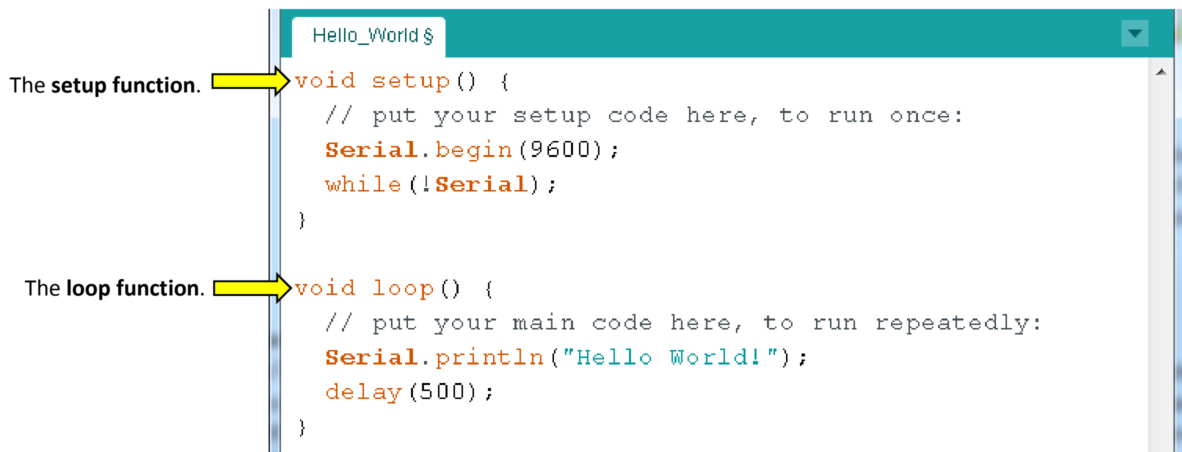


Figure xxx. All the pieces of the Arduino IDE.

"Ready, Set Go!" – Write some code

You are about to write a program from scratch. To do so, add code to the **setup** and **loop** **subprograms** that are found in *every* Arduino sketch. Put your cursor inside the **text editor** of the program window and enter the code *exactly* as it appears in Figure xxx, including capitalization and punctuation. Carefully check the spelling and don't forget to include the parentheses and semicolons!




```
void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  while(!Serial);
}

void loop() {
  // put your main code here, to run repeatedly:
  Serial.println("Hello World!");
  delay(500);
}
```

Figure xxx. The two subprograms named **setup** and **loop** must be included in all Arduino programs. Both are actually **void-functions** which return no value.

In a moment I'll explain what the code means, but for now, let's see what the subprograms do when you run the program. Double-check your spelling, capitalization, and punctuations, for if there are any mistakes the sketch will not upload. Beginners always seem to forget a semi-colon or two.

"Give 'er some Gas!" – Verifying, compiling, uploading, and running your sketch

It's almost time to upload the sketch to your development board and check out what it does – although I'm sure you already have a pretty good idea. To start, confirm that your development board is connected to the computer and check that the board is powered.

Before you can upload a sketch to your development board you must check some software settings. Ensure that the software knows which development board and which communications port you are using. You did this last chapter by selecting **Tools >> Board** and **Tools >> Port** from the menu bar. Verify that your development board and communications port are selected.

Next, check that your code is error free by clicking on the **Verify button** at the top of the IDE, as shown in Figure xxx, or press **CTRL+R** on your keyboard. Remember, anytime you alter your code, you should verify that your changes or additions are error free. **Verify your code often; you'll be happy that you do!**

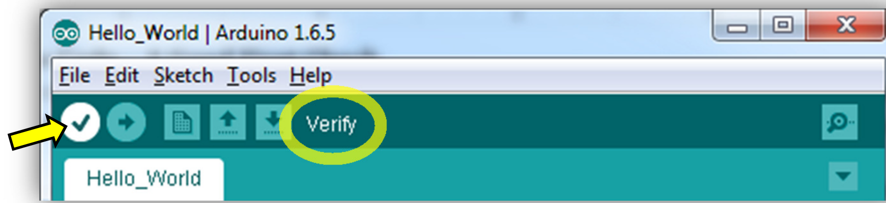


Figure xxx. Press the **Verify button** (or press **CTRL+R** on the keyboard) to *compile* the code and check for errors.

If you correctly followed all of the above steps, you should see a "*Compiling sketch…*" message and a green or blue **progress bar** below the code window, as shown in Figure xxx below.

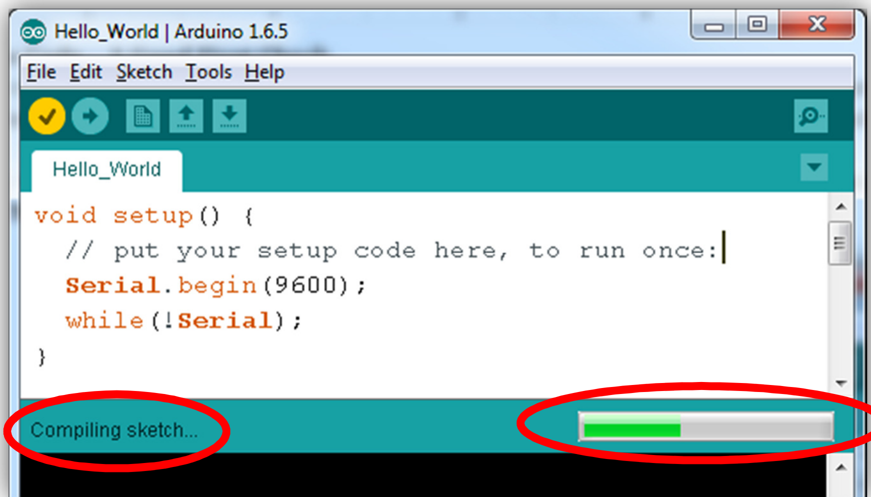
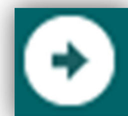


Figure xxx. The progress bar indicates the progress of the compile (verify) operation.

The Arduino software will not compile any code that contains errors, so if the sketch does contains errors the programmer must **debug the code**. If there are any errors in your code, the IDE will highlight them. (Check that you've included all of the curly braces and semicolons.) You should correct any mistakes now.

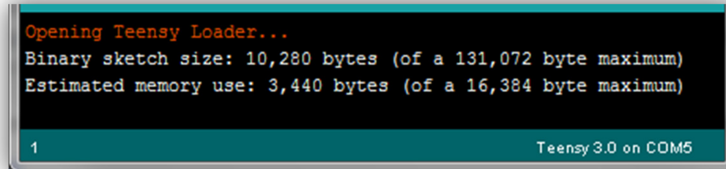
Your microcontroller is now ready to receive your program. Go get your friends and neighbors for the big event! Are you tingling with excitement?



Press either the **Upload button** on the IDE's toolbar or the **CTRL-U** keys on your keyboard to compile and upload your sketch to the development board. Remember, do this only once!

Drum roll, please! With any luck, you will see the software compile the sketch (remember to never unplug your board or re-upload your code while the progress bar is present!) …

… and display positive messages in the Message Area and Text Console …



… but where's the "Hello World!" message? There's no "Hello World!" message! *Is this thing broken?*

Actually, the microcontroller is doing *exactly* what it is supposed to be doing! To view the serial output from your development board, you must first open the **Serial Monitor** window. Before doing that, verify that the serial communications port is chosen by selecting **Tools >> Serial Port** and select the proper COM port.

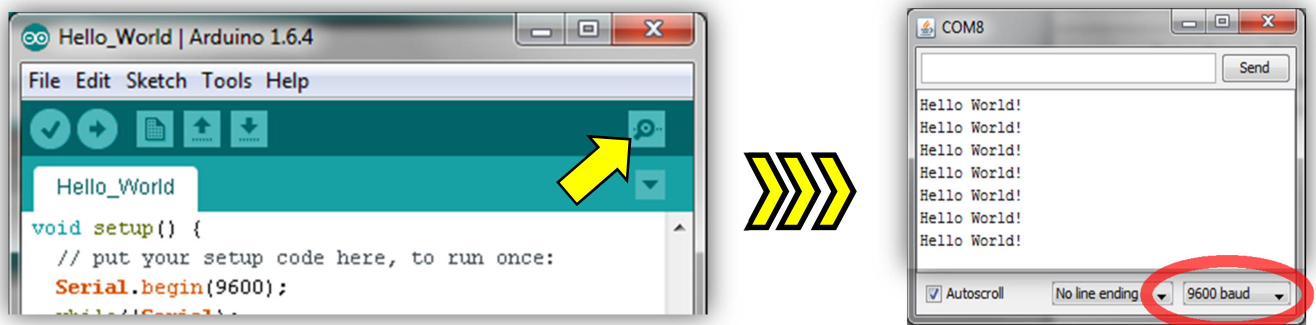
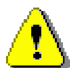


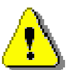
Figure xxx. Open the Serial Monitor to see the printed output of the sketch. Make sure the Serial Monitor is set to a baud rate of 9600.

Next, open the Serial Monitor by clicking on the **Serial Monitor button** shown with the yellow arrow in Figure xxx.[3] (You can also open the Serial Monitor with the hotkey combination, **CTRL-SHIFT-M**, saving you lots of time in the long run.) The Serial Monitor should *now* display a new "`Hello World!`" message every half-second. Pretty cool, huh? If the message is garbled or doesn't appear, verify that the **Serial** option is selected from the **Tools >> USB Type** menu. Arduino users should check that the baud rate setting at the bottom of the Serial Monitor window is 9600 baud as shown in Figure xxx.[4]

If the "Hello World!" message appears on your screen, everything is working properly and you are on your way!



It bears repeating that when your sketch is in the process of uploading to the embedded controller, **never unplug the development board from your computer!** Interrupting the two-way communications during an upload can permanently disable your embedded controller and you will have to buy a new one. Again, turn up the volume on your computer and listen for the tell-tale computer sounds that are generated after the sketch has been being successfully loaded onto the board.

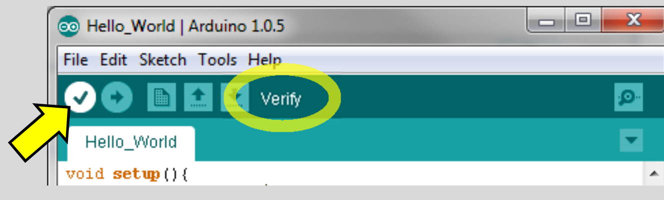


You should note that with version 1.6 of the Arduino IDE, your sketch will be *automatically saved* to your computer whenever a program is uploaded to your development board! Yay!

[3] When you press the **Serial Monitor button** the Teensy is reset and the program starts over.

[4] This should not be a problem for Teensy users because serial communication is handled through software, not hardware.

You can write and even compile code *even if there is no development board connected to your computer*! Simply write code in the Arduino IDE's text editor and then press the **Verify button** as shown below. This will compile your sketch but will **not** upload it to the chip. The compiler will still catch any mistakes in your code, but because the development board is not connected, there will be no output to the Serial Monitor. Nifty, huh?!



"Proceed with Caution!" – Dissecting the Code

Now that you have written, verified, compiled, uploaded, and run your first sketch for an embedded controller, allow me to demystify the code and explain what's going on. Don't skip this section, for real understanding comes from the details. Again, the goal is to become *self-sufficient* programmers; if you want to just copy and paste code without understanding there are plenty of places on the Internet for that.

Your sketch consists of two **subprograms** named **`setup`** and **`loop`**. Subprograms are self-contained chunks of code that carry out a set of instructions. Subprograms are powerful tools that help the programmer organize a large sketch into manageable mini-programs. Subprograms also make it easy for a sketch to execute the same set of instructions multiple times. Some programming languages differentiate between two types of subprograms, namely **methods** (or **procedures**) and **functions**. The difference between these two subprogram types is not important here. All you need to know is that *in C-based programming languages like Arduino, **subprograms are called functions***. In Arduino C there are no methods or procedures, although some online help sites use the words functions and methods interchangeably.

The `setup` and `loop` functions ***must** be included in all Arduino programs*. In fact, when you create a new sketch, the setup and loop functions are automatically created for you, as shown in Figure xxx.[5] As you will learn in Chapter xxx, you can create as many additional functions as you wish. Each function must have a unique name. Coded instructions for each function must be added *within* the **curly braces { and }** that follow each name, as shown in Figure xxx.

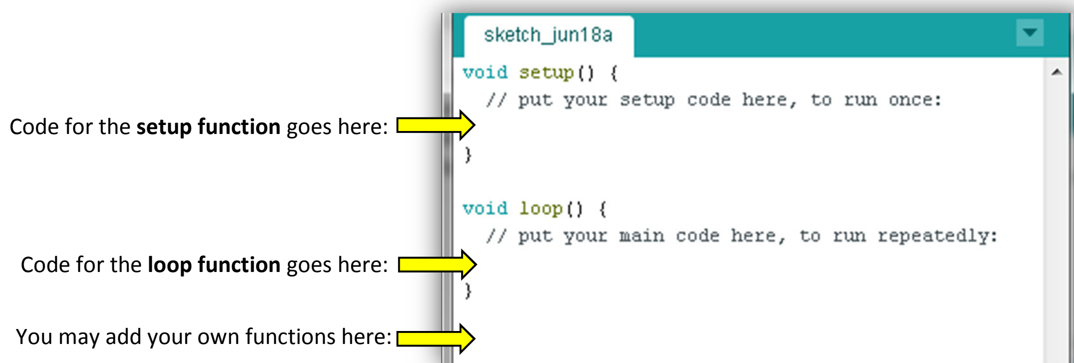


Figure xxx. The two functions named **setup** and **loop** must be included in all Arduino programs. Code must be added between the curly braces as shown. You may add functions of your own creation below these two.

[5] The mandatory setup and loop functions take a little getting used to if you are an experienced programmer.

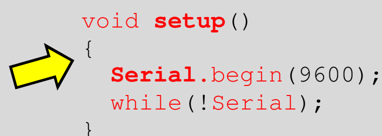
Take a careful look at the code for the **Hello_World** sketch that you've written. I've reprinted it below and have included line numbers, to which I'll refer.

```
1.    void setup(){
2.      // put your setup code here, to run once:
3.      Serial.begin(9600);
4.      while(!Serial);
5.    }
6.
7.    void loop(){
8.      // put your main code here, to run repeatedly:
9.      Serial.println("Hello World!");
10.     delay(500);
11.   }
```

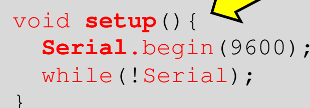
First, allow me to explain a few general observations and guidelines when coding in Arduino C:

- Notice that each function is followed by a pair of curly braces, `{` and `}`. The curly braces demark the *start* and *end* of most structures in the C-programming languages, including functions, loops, and logic statements. The curly braces *always* come in pairs. The code for the `setup` function must be located within the *first* pair of curly braces. The code for the `loop` function must be located on the lines between the *second* pair of curly braces.[6]
- The semicolon (`;`) must be added at the end of each line of code that does not end in a curly brace! This character denotes the end of a *statement*, and omitting it will result in a compiler error. Forgetting the semicolon is a common mistake, especially for beginners. An error message will usually alert you that a semi-colon is expected.
- Note the required set of parentheses `()` immediately after the `setup` and `loop` function names. They also surround the words `"Hello World!"` and the numbers 9600 and 500 in the `begin` and `delay` functions.
- Capitalization matters! If you spell `void` with a capital "V", for example, there will be errors and your code will not run. Some keywords must be capitalized, such as the "S" in `Serial`, but most often they are lowercase.
- Indenting properly takes a little practice. While not required, all code entered within the curly braces *should* be indented with at least one tab space. (My students should read this as *must* be indented!) Indenting makes the code more readable and therefore makes you a more valuable programmer. Indenting is usually handled automatically by the Arduino IDE as you enter code, but you can fix any indentation mistakes by clicking on **Tools >> Auto Format** from the menu bar or with the shortcut keystroke **CTRL+T**.

Before I move on, permit me a brief aside on the placement of the curly braces, `{ }`. Some programmers prefer to put both curly braces on their own lines (as shown below on the left) because it helps them more easily locate the start and end points of functions, loops, and statements. Others prefer the format presented below on the right because in big programs you can see more lines of code on the screen at a time. Both methods are fine and both will compile without errors.



```
void setup()
{
  Serial.begin(9600);
  while(!Serial);
}
```



```
void setup(){
  Serial.begin(9600);
  while(!Serial);
}
```

Personally I prefer to put each on their own line (left example), but to save space in my book have elected to follow the example on the right. Determine which suits you best and stick with it! It is bad form to use both methods!

---

[6] See http://www.arduino.cc/en/Reference/Braces for an excellent discussion of the curly braces.

Now let me dissect the **Hello World** code, line by line. Surprisingly, this short program will teach you a great deal:

- Notice that the **void** *keyword* precedes the `setup` and `loop` function names and is necessary to define the functions. I'll explain in more detail what this word means later on in Chapter xxx, but you should know that **void functions** are subprograms that *do* something by performing some *task*. (Soon you'll learn that *non-void functions* can *return* values. For example, the square root function on your calculator *returns* the answer to the screen after the button is pressed.) Just know that for now, the void keyword will precede each function name when it is being created.

- It is important to know that the **`setup`** function (Lines 1-5) runs only *once*. It is a place for the programmer to define a myriad of things such as the pin numbers of a sensor or the baud rate for serial communication. The **`loop`** function (Lines 7-11) is where you type in the *main code* for your application. As the name suggests, this code will run over and over in a continuous loop. There are ways to stop the loop, which I'll show you later on in this chapter.

- Again, the purpose of this sketch is to write code to send the words, "Hello World!" to the Serial Monitor. In programming parlance we say that we want to **print** the words, "Hello World!" to the screen. For this to happen you must first set the communication **baud rate** of the development board. That is, you must define the rate at which the microcontroller sends data to the Serial Monitor. The baud rate is set in Line 3 with the **`Serial.begin(9600)`** command, which tells the development board to output serial data at a rate of 9600 baud, or 9600 bits per second. If the baud rate of your board matches that of the Serial Monitor they will be in sync and the program's output will be correctly displayed onto the screen.

- Because the ARM processor is so fast, the program must wait a moment or two for the serial link to be established. This is where Line 4 and the `while(!Serial)` statement come into play. (The exclamation point (`!`) means "not" in programming languages.) Therefore, Line 4 translates as, "*While there is no serial connection, wait and continue waiting until a connection is established.*" That is, the program will *loop* over and over until a good serial connection is established between your development board and your computer. It is not critical at this time that you completely understand the **while loop**. You will see it again briefly later on in this chapter and in greater detail in *Chapter xxx*.[7]

- Line 6 is a blank line and may be omitted if you prefer. Blank lines are ignored by the compiler. It is customary, however, to separate functions with some *vertical spacing* because it makes the code easier to read.

- Focus now on the `loop` function, which begins on Line 7. The heart of the above sketch lies in Line 9, namely with the command, `Serial.println("Hello World!")`. Here, the `println` function is used to send data to the Serial Monitor. (The `println` command is pronounced "*print-linn*".) Programmers say the data is ***printed*** to the screen. In this example, the data to be printed is enclosed in quotes and is known as a ***string*** because it contains a *string of characters*. The `println` function also adds a carriage return (*i.e.*, an ENTER keystroke) to the end of the string. That means it prints to the screen whatever is enclosed in the quotation marks, and then moves the cursor to the next line.[8]

- It should come as no surprise that the `delay(500)` function on Line 10 suspends, or *delays*, the program's execution for some period of time. The number in parentheses is called the ***argument***, of the function and represents the number of milliseconds (ms) to pause the program. You may recall from middle school science classes that there are 1000ms in one second. So the command `delay(500)` will suspend the program for 500 milliseconds or 0.5 seconds.

- Finally, the `loop` function is concluded with the right curly brace `}` on Line 11. However, remember that by default the `loop` function runs continually, so when the program reaches Line 11 it bounces back up to the beginning of the `loop` function at Line 7 and `loop` is run once again. This happens *very rapidly*, which is why it is necessary to pause the output with the `delay` command. If you don't pause the program's execution, the user wouldn't be able to see the output on the Serial Monitor and the computer screen would eventually lock up!

Who knew there'd be so much to learn for such a simple program!

---

[7] This line is only required when using the Teensy 3.2. You see, Teensy boards use software to fool the USB port into thinking it is a serial port, and the Teensy 3.2 is so fast that if you don't wait for a connection you won't see any serial output. This line is not needed for the Teensy 2.0 because it isn't speedy enough to cause problems, and it is not needed for Arduino boards because they handle serial communications through hardware on the board.

[8] In contrast, the `print` command will print the data without the carriage return.

"The Final Lap" – A little experimenting

It's now time for you to have a little fun and experiment with what you've learned. Specifically, there are five little exercises I want to walk through with you. I've labeled the third and fourth "experiments" as optional, for the topics covered there are not fundamental to your understanding of Ardunio C programming. I do highly recommend spending time on the other "experiments". Feel free to play around and explore on your own, for that is truly the best way to learn anything!

1. **Change the delay.**

   The first exercise is a simple one. Simply alter the delay interval so the messages scroll up more slowly. Say you want a new message to appear every second, or 1000ms. That is, edit your code to read, `delay(1000)`. Test your newly altered sketch by pressing the **Verify button** and then the **Upload button**.[9]

2. **Change the message: using `println` and `print` to output multiple lines.**

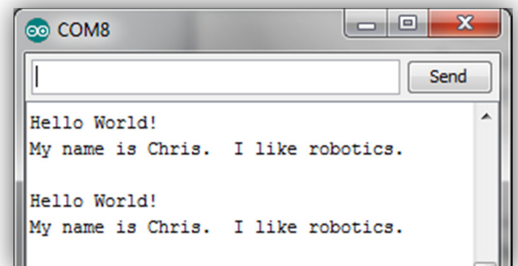
   There are several ways to print data to the Serial Monitor. Perhaps the most common way is with the `println` function, which, as you've learned, sends data to the Serial Monitor and follows it with a carriage return or ENTER keystroke. If you wish to output data **without** the carriage return, use the **`print`** function instead. Examine my additions to the `loop` function shown below and add the new lines to the function within your sketch. Feel free to alter the messages in the text strings.

```
void loop(){
  Serial.println("Hello World!");
  Serial.print("My name is Chris.  ");
  Serial.println("I like robotics.");
  Serial.println("");
  delay(1000);
}
```

   `print`, not `println`

   A blank line to make the output look nice.

   Verify and upload the new sketch to your development board. Examine the code's output on the right and match it to the above code. Note that the `print` function does **not** add a carriage return, so the strings "My name is Chris. " and "I like robotics." appear on the same line. Notice that I added some spaces within the quotes after the word "Chris." simply to offset the two sentences.



   Also notice that the command `Serial.println("")` sends a blank line to the Serial Monitor, making the output look nice with each loop. The empty quotes "" is known as the **null string**.

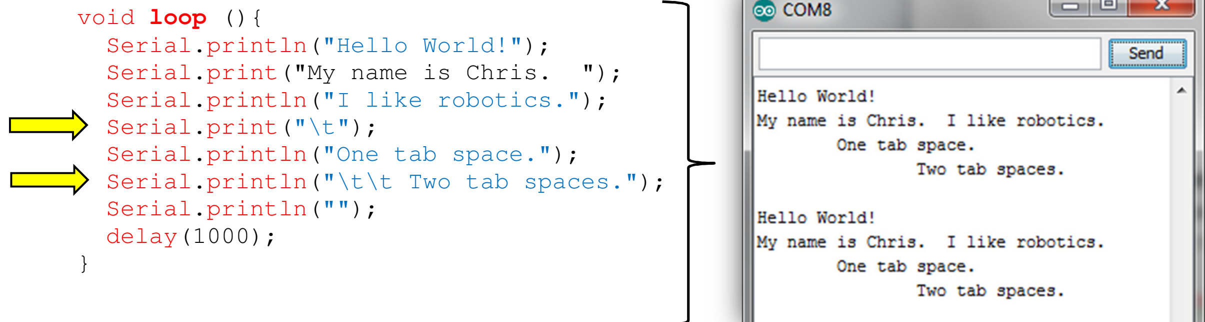
   Next, try to add a blank line between the first two lines in the above loop function, like so:

```
Serial.println("Hello World!");

Serial.print("My name is Chris.  ");
```

   Verify and upload the sketch and view the serial output. You may be surprised that the output did **not** change. You see, the compiler ignores empty lines, so if you do want a space between the lines, you must insert a `Serial.println("")` command between them.

[9] Some versions of the Arduino IDE require that you close and reopen the Serial Monitor each time you upload a new sketch.

3. **Text formatting with the tab \t escape sequence. (*Optional*)**
   You can also output a **tab space** to the Serial Monitor with the **\t** tab **escape sequence**. In the C language, a backslash (\) followed by a letter this is known as an *escape sequence*, which are used to output certain special characters from within the `print` or `println` functions. (Be sure to use the backslash character and not the forward slash.) The tab escape sequence (\t) comes in handy when the spacing of the output is important, for example in data tables. All escape sequences must be enclosed in quotes and may appear in their own print statement or appended to another string, as shown in the code below. Take a look at the lines I added to my sketch and the code's subsequent output:



```
void loop (){
  Serial.println("Hello World!");
  Serial.print("My name is Chris.  ");
  Serial.println("I like robotics.");
  Serial.print("\t");
  Serial.println("One tab space.");
  Serial.println("\t\t Two tab spaces.");
  Serial.println("");
  delay(1000);
}
```


4. **Other escape sequences \n, \", and \\ (*Optional*)**
   There are other escape characters that deserve a *little* attention. All escape sequences begin with a backslash (\), and it is important to remember that these only work *within the quotation marks of a string*, so anytime you see a backslash within a set of quotes you know you are about to encounter an escape sequence. What follows are brief explanations of a few of these escape characters and a sample piece of code showing how they may be used. For a more complete list of escape sequences, see Appendix xxx or http://msdn.microsoft.com/en-us/library/h21280bw.

   - ***New line* escape sequence, \n.** Use the escape sequence **\n** to add one or more line breaks to your string. These are usually added at the beginning or at the end of a string, but may appear in the middle as well. Here are a couple of code snippets and their output:

```
Serial.print("Line one, \n Line two.");
```

```
Line one,
 Line two.
```

   See what happens when you add multiple line break escape sequences:

```
Serial.print("One line break: \n");
Serial.print("Two line breaks: \n\n");
Serial.print("Three line breaks: \n\n\n");
Serial.print("The End.");
```

```
One line break:
Two line breaks:

Three line breaks:


The End.
```

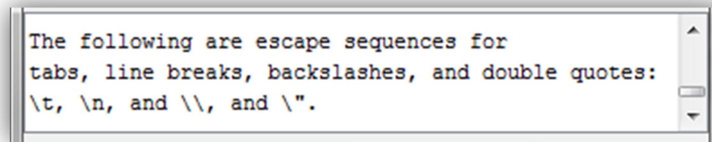
   - ***Double quote* escape sequence, \".** Use the escape sequence **\"** to print a double quotation mark within the string's quotes.

```
Serial.print("\"Never memorize something that you can look up.\" \n");
Serial.print("\t\t\t - Albert Einstein");
```

```
"Never memorize something that you can look up."
                        - Albert Einstein
```

- ***Backslash* escape sequence, `\\`.** Use the escape sequence `\\` to print a backslash within the string's quotes. See if you can follow the somewhat confusing code below. It's not, not, not that hard.

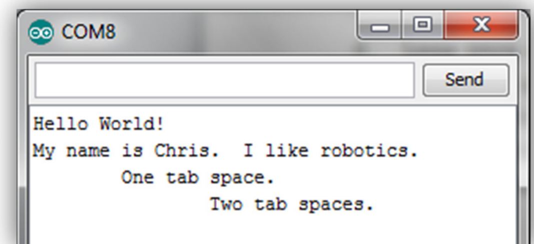
```
Serial.println("The following are escape sequences for");
Serial.println("tabs, line breaks, backslashes, and double quotes:");
Serial.println("\\t, \\n, and \\\\, and \\\".");
```



5. Loop once with a `while` statement.

If you are an experienced programmer you may be a bit put off by Arduino's required infinite `loop` function. This may be beneficial for some applications, but often you want your code to do something *just once*. For example, there's no need to print the above messages multiple times. **To make the loop run just one time**, add the line `while(true);` at the *bottom* of the `loop` function as shown below. Make sure it is added *within* in the curly braces as shown!

```
void loop (){
  Serial.println("Hello World!");
  Serial.print("My name is Chris.   ");
  ...
  delay(1000);
  while(true);
}
```



When you upload the sketch and open the Serial Monitor you *should* see that the loop function now runs only *once*. (If your Serial Monitor window is blank, jump down a couple of paragraphs to the sub-section, named "**If your serial window is blank**".)

The `while` structure will be discussed in detail in *Chapter xxx*, but I can tell you that it is called a **while statement**, and in this example it, too, will loop forever. The while loop is defined as

`while` (*logical expression*) ;

This definition can be deconstructed by translating it into English: "continue to run this line of code while the *logical expression* is *true*". Using the keyword `true` in place of the logical expression means this statement will never end because *true* can never be *false*. In other words, this command does nothing except loop forever. Therefore, by putting this *infinite* `while` loop at the end of the *infinite* `loop` function, the `loop` function *seems* to run only one time.[10]

[10] If you prefer, you may enter the `while(true){}` *loop* rather than `while(true);` *statement*. Both do the same thing.

If Your Serial Window is Blank
If your Serial Monitor is blank after uploading the sketch, it probably means that the Serial Monitor appears on the screen *after* the loop function has started running. If this happens to you, simply add a small delay – say of 100ms – after the `while(!Serial)` command in the **`setup`** function, like so:

```
void setup (){
  Serial.begin(9600);
  while(!Serial);
  delay(100);
}
```

It is probably a good idea to add this delay to any program that utilizes the Serial Monitor.

> Some Arduino programmers write their code within the `setup` function and only call the `loop` function if a looping structure is required. To me this seems sloppy. To be honest, I'm not thrilled with Arduino's mandatory `setup` and `loop` functions, but in my mind the word "setup" implies that this space is reserved for a few housekeeping details, not lengthy, wholesale code.

6. **Secret messages: swap boards with your neighbor. (*Optional*)**
   This final little exercise is to show that once the sketch is uploaded to the development board, it's the *development board* that is running the code, *not* your computer! To demonstrate this, first add a line of code to your sketch that will print a **secret message** to one of your friends. (Your friend must have the Arduino software installed on their computer.) Next upload the secret message sketch to your board and verify that the message prints properly to *your* Serial Monitor. Finally, unplug your development board from your computer and plug it in to your friend's computer.

   To see your secret message, all your friend has to do is open the Serial Monitor from his or her Arduino IDE, and like magic, the message appears! (Of course, your friend will need to make sure their IDE is setup with the correct board (**Tools >> Board**) and that the communications (COM) port on their computer is set to the correct value (**Tools >> Port**). Because this communications hand-shake takes time, you may elect to skip this "experiment".)

"The Finish Line!" – Adding comments
All that remains to do is add a few *comment statements* to your code for the benefit of those humans reading it. Comments are helpful, explanatory statements that help *document* what your program does. Some programmers do a good job adding thoughtful and coherent comments. The others are unemployed. My students are *required* to add comments throughout their code. In the next section I explain how to add comments to your code.

2.3. No Comment? Yes Comment! Turn that Code into a Tutorial

Before you put this sketch to bed, you should add a few so-called *remarks* or **comments** to your code. Good programmers include comment statements to **explain and document** what their program does. They are also used to describe the purpose of a particular line of code or clarify the motivation behind some *algorithm* or chunk of code. Comments are helpful for anyone reading your code for the first time, and they can actually serve to remind the programmer of knotty details that are easily forgotten. It's like adding a Post-It Note to your code. I've never heard anyone complain that a program has too many comment lines, only too few. Make your code as readable as possible – get into the habit of using comments!

Comments should be added as you write your code, **not** as an afterthought when you are done. Do not make the common mistake of saying to yourself that you will add the comments *after* the code is finished. You'll *never* do it. Trust me on this, and *add comments as you enter your code!*

In the C language, there are two ways to add comments to your code: single-line and multi-line comments.[11] Both are completely ignored by the compiler. Comments are strictly for human consumption.

1. **Single-line comments.** Simply type two forward slashes (`//`) and everything **after** this point *on that particular line* of code is ignored by the compiler. Here are a two of examples of the single-line comment:

```
// The following prints a message to the Serial Monitor
Serial.println("Hello World!");      // Prints string and carriage return (CR)
```

   Add the comments to your sketch and upload it to your microcontroller. (Don't forget to first **verify** the code is error-free!) Notice that "Hello World" is still printed to the screen.

> If you want to ignore a chunk of code but you don't want to permanently delete it, you can highlight the offending code and then select **Edit >> Comment/Uncomment** from the menu bar, or press **CTRL+/**. This will add single-line comment marks at the start of each line of highlighted code.
>
> If you later wish to uncomment that code, simply highlight the code and select **Edit >> Comment/Uncomment** once again. This is a very handy feature of the IDE!

2. **Multi-line comments.** This type of comment can span *multiple lines*, making it easy to add **prologue comments** at the beginning of a program and making it useful to "comment out" large blocks of code within the body. Begin the multi-line comment with a forward slash and asterisk (`/*`) and end the comment with an asterisk and forward slash (`*/`). Everything in between is ignored by the compiler. It is a good practice to included prologue comments to the top of each of your sketches. (For *my* students this is a *required* practice.) Below is an example of a prologue comment; I hope your sketch names are more descriptive and you are more informative with your sketch description.

```
/*
  MyProgram.ino
  by C.D.Odom on May 15, 2013
  This program does a lot of this and little of that
*/
```

> If you plan to distribute your sketch to the public at large, you should consider adding a **copyright** in the prologue to protect the intellectual rights to your code. In the United States, the copyright notice must contain the word "copyright" or the symbol © or (c). It also must include the date and the name of the owner. For example:
>
> ```
> /*
>   MyProgram.ino
>   Copyright (c) 2013 Chris D. Odom
>   This program does a lot of this and little of that
> */
> ```

[11] Excellent examples of comments can be found here: http://javadude.com/articles/comments.html.

The finished code for my **Hello_World** sketch is printed below, complete with comments. I cannot stress enough the importance of including good comments within your sketches. The Arduino IDE turns all comments gray within the text editor, making them easily identifiable.

```
/*
  Hello_World.ino by C.D.Odom 2013
  Diagnostic program illustrating println, print, delay, escape sequences, and comments.
*/

void setup(){
  Serial.begin(9600);              // set the baud rate for the serial port
  while(!Serial);                  // wait for a serial connection to be established
  delay(100);                      // wait for Serial Monitor to open before looping
}

void loop(){
  // The following prints a message to the Serial Monitor
  Serial.println("Hello World!");               // includes carriage return (CR)
  Serial.print("My name is Chris.  ");          // no CR
  Serial.println("I like robotics.");
  Serial.print("\t");                           // add one tab space.
  Serial.println("One tab space.");
  Serial.println("\t\t Two tab spaces.");       // add two tab spaces
  Serial.println("");                           // print a blank line
  Serial.println("Line one, \n Line two.");     // a line break escape sequence
  Serial.println("");
  Serial.print("Line one, \n Line two.\n");     // print with a line break = println
  Serial.println("");
  Serial.print("One line break: \n");
  Serial.print("Two line breaks: \n\n");
  Serial.print("Three line breaks: \n\n\n");
  Serial.print("The End.");
  Serial.println("/n");                // println + \n = 2 line breaks

  // Double quote escape sequence demo:
  Serial.print("\"Never memorize something that you can look up.\" \n");
  Serial.print("\t\t\t - Albert Einstein");
  Serial.println("\n");

  // Backslash escape sequence demo:
  Serial.println("The following are escape sequences for");
  Serial.println("tabs, line breaks, backslashes, and double quotes:");
  Serial.println("\\t, \\n, and \\\\, and \\\".");
  delay(1000);                         // pause for 1000ms (1.0s)
  while(true);                         // an infinite loop that pauses the main loop
}
```

Why Should You Comment Your Code?

1. Think of each sketch you write as a chapter in **your own textbook**. More than any other source, you are likely to refer to your own programs as a reminder how to do specific tasks. If your code lacks comments, you may waste a great deal of time trying to decipher a program you wrote just a few weeks before! I frequently refer to programs that I wrote over ten years ago, and I would be lost without my own comment lines.
2. **Others will read your code**. You'll find that well-commented code is used much more frequently than poorly-commented code. Comment your code and you could be famous! (Or, at least gainfully employed.)
3. Prologue comments at the beginning of each program impart useful **documentation** to the reader. They should include the name of the program, who wrote it, the date, and a short description.[12] You may also wish to copyright your work here.[13]
4. I find it useful to *comment out* code that didn't work rather than delete it. This way, I can come back to the program at some later time to find and correct my mistake.

---

[12] You'll find that I omit these header comments throughout my book. I do this simply to conserve space, not because I am lazy.
[13] See http://www.copyright.gov/ and http://www.copyrightadviser.com/ for more information about protecting your work.

My students ask how many comments they should have in their sketches. The beginner may wish to add comments to nearly every line, while the advanced user may comment only the more knotty problems. Think of your code as a textbook for yourself and others. The more comments you have the better the instruction. A good guideline is if you have to think about a piece of code for more than 60 seconds or you have to refer to some reference material for assistance, you should add a comment statement explaining your code. Trust me, you'll be happy you did.

2.4. Functions – A Brief Introduction

In this short concluding section I introduce you to programming with **functions**, which is a handy way of breaking up a large program into smaller, self-contained pieces, or *modules*. In Arduino C, the modules are known as **functions**, and we invoke each function by "calling" it. Some functions are called only once in each sketch, while other functions may be called numerous times. My primary motivation for teaching this topic now is to demonstrate a way to organize your programs, specifically your solutions to the Challenge Problems at the end of each chapter. While modular programming techniques are helpful for even short programs, they are indispensable when your programs grow in length and complexity. In *Chapter xxx*, we will dive more deeply into this important topic.

Create a *new sketch* and call it "**Functions Intro**". In this sketch, we will have some fun with Friedrich Nietzsche's quote, "*The doer alone learneth*". By breaking the quote into three smaller strings ("The doer", "alone", and "learneth"), we can then use functions to build the quote by printing each string to the screen, in order, one at a time. There are much easier ways to accomplish this, but my goal here is to create a simple, easy-to-understand example demonstrating how to use homemade functions.

A Straightforward Way to Call Functions

Get started by creating five void functions: **`setup`** and **`loop`** as always, and then three new, homemade functions named **`msg1`**, **`msg2`**, and **`msg3`**. Enter your code to match mine. As you can see, there's not much going on. There's the `while(true)` statement that prevents the loop function from looping forever, and there are three `Serial.print` commands that display the three pieces of Nietzsche's quote.

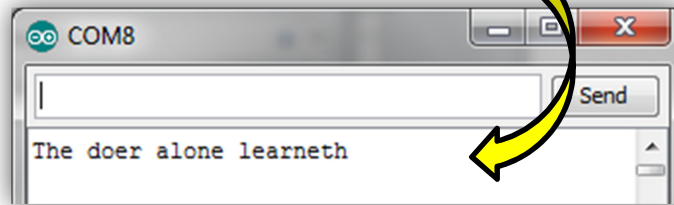
```
void setup(){
  Serial.begin(9600);          // set the baud rate
  while(!Serial);              // wait for a serial connection to be established
  delay(100);                  // give the Serial Monitor time to open
}

void loop(){

  while(true);                 // an infinite "pause" loop
}

void msg1(){
  Serial.print("The doer ");
}

void msg2(){
  Serial.print("alone ");
}

void msg3(){
  Serial.print("learneth");
}
```

So, what will happen if you upload this code and then open the Serial Monitor? You may *think* the phrase, "The doer alone learneth" will appear on the screen. If so you are wrong and didn't take to heart Nietzsche's words. If you were a *doer* and actually entered the code and uploaded it, then you *know* that nothing printed to the Serial Monitor. There are no errors here. In the words of Brian Patton, CEO of Patton Robotics: "Robots never do what you *want* them to do; only what you *tell* them to do." And you told your microcontroller to do nothing!

Examine your code. In every Arduino sketch, the program first runs the **setup** function and then runs the **loop** function. In the above code, neither of these functions *do* much of anything. So it's up to the programmer to do some additional things. Somehow you have to *tell* the program to execute the msg1 function, then execute the msg2 function, and finally execute the msg3 function. In programming parlance, we say the program must ***call*** msg1, then ***call*** msg2, then ***call*** msg3.

There are a number of ways to tell the program this. For example, to make all the calls from within the existing loop function *add* the following highlighted lines to that function, as shown below. Verify this sketch, upload it to your development board, and then open the Serial Monitor. *Now*, you should find that Nietzsche's quote has indeed been printed to the screen:

```
void loop(){
  // build the sentence:
  msg1();          // call the msg1 void function
  msg2();          // call the msg2 void function
  msg3();          // call the msg3 void function

  while(true);     // an infinite "pause" loop
}
```



Do you see what is going on here? Quite simply, you *called* each function by entering the names of the functions complete with an empty set of parentheses and concluding semicolon.[14] When each called function completed the execution of its code, it *returned* to the place that called it, and the program resumed running from that point, calling the next function in line.

Permit me to walk you through the code step-by-step:

1. First, msg1 was called from inside loop. This caused the program to jump down to the msg1 function, which printed "The doer " to the Serial Monitor.

2. After msg1 had completed running its code, the program returned to the loop function at the point it left.

3. The next command within the loop function was a call to msg2, which printed the word "alone" to the screen.

4. After that, the program returned once again to the loop function where msg3 was called and "learneth" was printed.

5. Finally, the program returned to the main loop where the infinite while(true) statement was executed. Because true is never false, this little loop never stopped looping and the program essentially ended here.

Another Way to Call Functions

In the example above, you called your functions from within the main **loop**. But did you know that functions can also be called from within *any function* in the sketch? Examine the following code, noting that msg1 is the only function called from within the main loop.

```
// Functions_Intro.ino
// An intro to modular programming using functions by C.D.Odom, George School, 2013

void setup(){
  Serial.begin(9600);       // set the baud rate
  while(!Serial);           // wait for a serial connection to be established
  delay(100);
}
```

[14] The parentheses are empty because they are empty at the function-level. See *Chapter xxx* to learn how to pass arguments and fill up those parentheses.

```
void loop(){
  // build the sentence:
  msg1();                    // call the msg1 void function

  while(true);               // an infinite "pause" loop
}

void msg1(){
  Serial.print("The doer ");
  msg2();                    // call the msg2 void function
}

void msg2(){
  Serial.print("alone ");
  msg3();                    // call the msg3 void function
}

void msg3(){
  Serial.print("learneth");
}
```

If you were careful, you might have predicted that the output of the above code was also, “The doer alone learneth”. The step-wise progression of *this* sketch proceeds like so:

1. The `loop` function is automatically called.
2. The `msg1` function is called from within `loop` and the message “The doer ” is printed to the Serial Monitor.
3. Just before `msg1` is finished executing, it places a call to the `msg2` function, which displays the message, “alone” to the screen.
4. Just before `msg2` is finished executing, it calls `msg3`, which displays the message, “learneth” to the screen.
5. After that, the program jumps back to where it was last called, which was `msg2`, in this case.
6. But there are no more commands in the `msg2` function to be executed, so the program jumps back to the place that called `msg2`, which was `msg1`, in this case.
7. But there are no more commands in the `msg1` function to be executed, so the program jumps back to the place that called `msg1`, which was `loop`, in this case.
8. Finally, after the program returns to the main `loop`, the infinite `while(true)` statement is executed. Because true is never false, this little loop never stopped looping and the program essentially ended here.

Take this little example to heart, for programming with functions is powerful stuff!

When I began writing code, I can’t tell you how many times I forgot to include the empty parentheses **()** when calling a function. The code compiles just fine but the function isn’t called because in the land of computers `myFunction` does **not** equal `myFunction()`. As a result, my programs would not behave as expected and it would take me a while to locate the problem. Fortunately this problem is easy to fix: **don’t forget the parentheses**!

Naming Conventions for Functions
When coming up with names for your homemade functions, there are a couple of steadfast rules you *must* follow, and there are a number of recommended suggestions that you *should* follow when programming in Arduino C. Programmers should put some thought into the naming of their functions and follow the guidelines listed below:

The mandatory naming rules:

- Names cannot contain spaces. In addition to spaces, names *cannot* contain special characters including !, @, #, $, %, ^, &, *, -, <, >, /, \, :, and ?.
- Functions cannot *begin* with a number. For example, `2Sensors` is not allowed, but `twoSensors` and `sensor2` are both permitted.

The suggested naming conventions:

- Multiple-word names may be compressed into one by using the *underscore* character, such as `problem_25`. The more common compression method is simply omitting the spaces and starting subsequent words with capital letters, such as `distanceFromObject` or `timeOfDay`. The lowercaseUppercase convention is common with many languages, and may be referred to as **camelCase**, **mixedCase**, or **humpBack** notation.
- While not required, it is good practice to start the function name with a lowercase letter.
- Case matters; `sensorPin` is *not* the same as `sensorpin`.
- Function names should be short but descriptive. It may seem logical to have very short function names because they are easier to type. However, longer and more descriptive names will usually save you time when you debug your code, read over old code, or share your code with others.
- Finally, you cannot use keywords that are reserved for the C and Arduino languages, such as `print`, `loop`, `setup`, `true`, and `HIGH`. For a list of these reserved keywords words, see *Appendix xxx: Reserved Keywords*.

For more on naming guidelines, see http://en.wikipedia.org/wiki/Naming_convention_(programming).

2.5. In Conclusion: Formatting and Saving Your Code

To conclude this chapter, here are two last little pointers:

1. When you think your sketch is finished, use the handy **Auto Format** tool to repair any sloppy formatting on your part and make your code look presentable. This is important if you are sharing your work with others (or if your work is being graded)! You can find the Auto Format tool by clicking **Tools >> Auto Format** on the menu bar, or simply pressing **CTRL+T** on your keyboard.

2. Don't forget to save your work! Do this often. If the IDE crashes, you'll lose everything since your last save. If you see the **section sign (§)** next to the sketch name it means the sketch is *not* saved. Remember that your sketch is automatically saved to your computer anytime you upload the sketch to the development board.

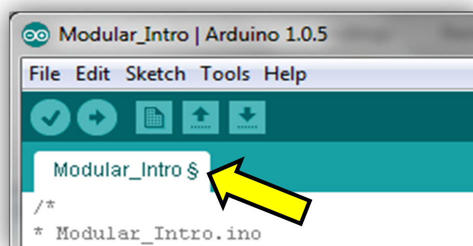


Figure xxx. The section sign (§) indicates an unsaved sketch.

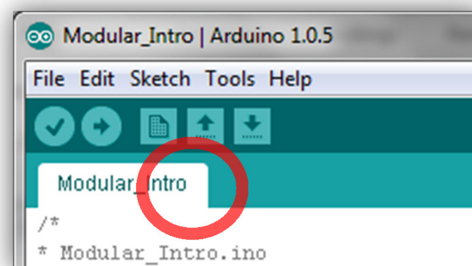
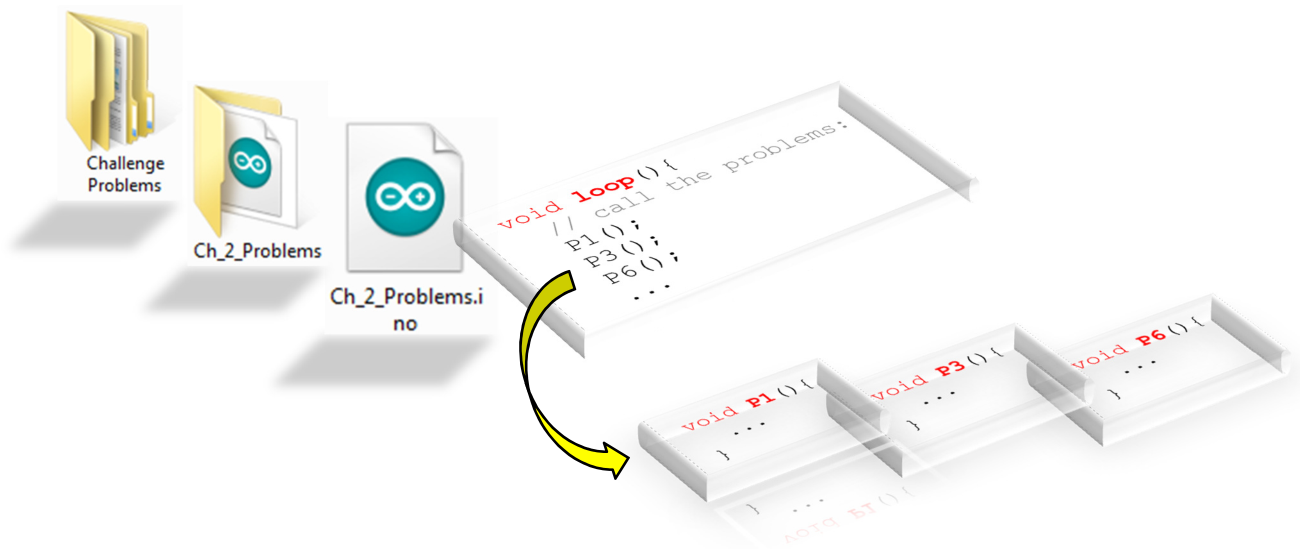


Figure xxx. The saved sketch.

Well, that's it. You are now on your way to becoming a robotics programmer! In the following chapters, you will take your development board out on the open road and see what it can do. Good luck and welcome to embedded programming!

Challenge Problems

Before getting down to solving the following **Challenge Problems**, you should first create a new folder in which to store the solutions for all Challenge Problems in this book. Then, for each new chapter's Challenge Problems, create a new sketch into which you'll enter your solution code. (For this chapter, I named mine, "Ch 2 Problems", for example.) Do **not** create a new *sketch* for each of the assigned Challenge Problems below. Rather, I recommend that you create a new *function* for each of the assigned problems. For example, I named my solution functions "P1", "P2", "P3", etc., (for Problems 1, 2, 3, etc.) and then called them from the main **loop**, as shown in the graphic below.



You may wish to use the following **template** for the Challenge Problems in this and all sequent chapters.

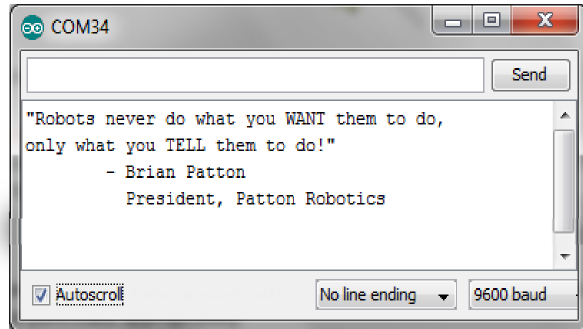
```
// Chapter 2 Solution sketch by C.D.Odom on 7.13.15

void setup(){
  Serial.begin(9600);
  while(!Serial);
  delay(100);
}

void loop(){
  // call the problems:
  P1();
  P2();
  P3();

  while(true);    // indefinitely pause the loop function
}

void P1(){
  // code for this problem goes here...
  Serial.println("\nProblem 1.");
}

void P2(){
  // code for this problem goes here...
  Serial.println("\nProblem 2.");
}

void P3(){
  // code for this problem goes here...
  Serial.println("\nProblem 3.");
}
```

So what are you waiting for? Get coding!

2-1. **Predict** the output of the following snippet of code. Do ***not*** upload the code to a development board. Your instructor may want this done on a piece of paper, or perhaps in your sketch as a comment block.

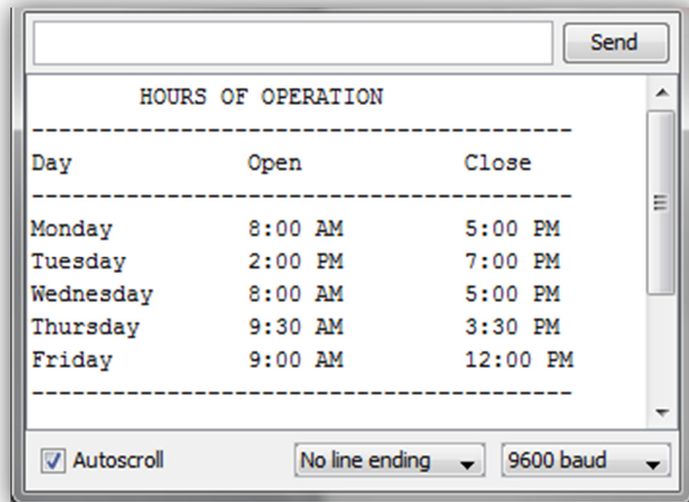
```
Serial.println("Hello");
Serial.print("A");
Serial.print("B");
Serial.println("C");
Serial.println("Good-bye");
Serial.print("1");
Serial.print("2");
Serial.print("3");
```

2-2. Use the `println` and `print` functions along with the tab and double quote escape sequences (`\t` and `\"`) to re-create the serial output at the right:

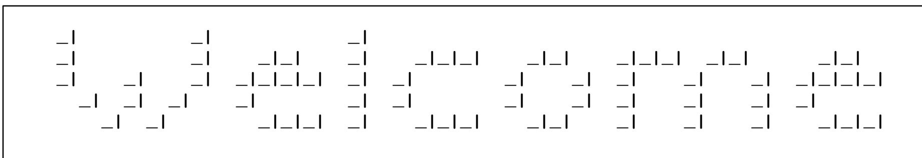


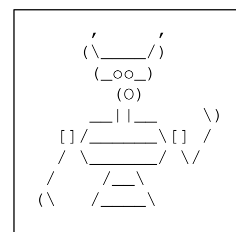
2-3. Write a program that will print a few lines of your favorite song or poem to the screen. Print also the title and author, and indent and punctuate appropriately.

2-4. Use the `println` and `print` functions along with the tab escape sequence (`\t`) to re-create the serial at the right:



2-5. Write code that will display to the Serial Monitor your own ASCII art or welcome message. Visit http://www.asciiworld.com/ or http://www.network-science.de/ascii/ for some handy tools. If you are out of ideas, reproduce one of the following ASCII images. You may need to use an escape sequence or two. (For example, see page 40.) The vertical lines, or "pipes", shown below are created by holding down the SHIFT key and then pressing the backslash ("\") key.





2-6. Print the words to the song, "Happy Birthday to You", on the computer screen in time with the song. There's no real need to have the music actually playing – just play the tune in your head. You should use the `delay` function to make the words appear at the appropriate times – Karaoke style.

2-7. Which of the following function names are **not** valid and would yield an **error message** when compiling?

a. myFunction()
b. my@!#$Function()
c. display Distance()
d. SpinMotorClockwise()
e. spinMotorCounterClockwise()
f. problem5()
g. 5problem()
h. exit_Room()
i. readSensorThenCalculateRangeThenExitRoom()
j. x()

2-8. Which of the following function names are valid **and** follow the suggested naming convention guidelines? That is, which are the most properly named functions?

a. myFunction()
b. my@!#$Function()
c. display Distance()
d. SpinMotorClockwise()
e. spinMotorCounterClockwise()
f. problem5()
g. 5problem()
h. exit_Room()
i. readSensorThenCalculateRangeThenExitRoom()
j. x()

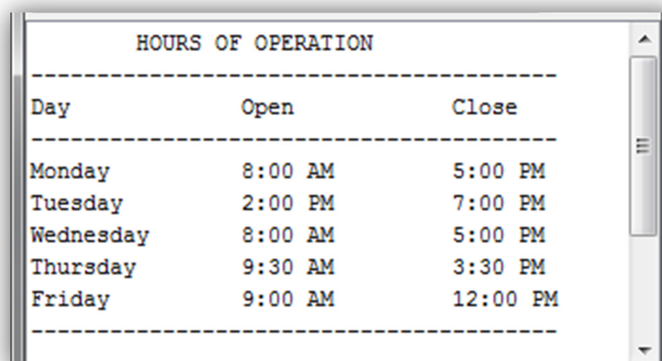
2-9. In the "**Functions_Intro.ino**" sketch that you created in this chapter, the Nietzsche quote, "The doer alone learneth," was printed to the screen by calling three functions from within the loop function like so:

```
void loop(){
  // build the sentence:
  msg1();             // call the msg1 void function
  msg2();             // call the msg2 void function
  msg3();             // call the msg3 void function

  while(true);        // an infinite "pause" loop
}
```

Modify that sketch so it prints the quote in this slightly different order: "The doer learneth alone". Do **not** alter the messages. Rather, simply alter the *order* that you call each message to produce this nonsensical output.

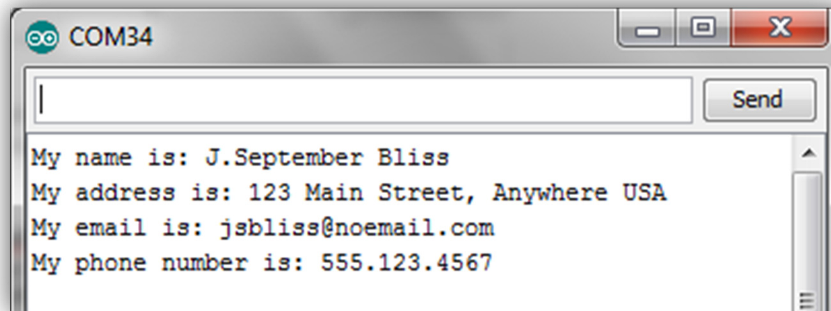
2-10. This is a repeat of Problem 2-4 above. But this time, copy the following "**dashedLine**" function and add it to your sketch. Next, use the **dashedLine** function along with the `println` and `print` functions and the tab escape sequence (`\t`) to re-create the serial output below. **Hint:** you will need to call the **dashedLine** function three times. (If you were not assigned Problem 2-4, add the `dashedLine` function to visually set-off one of your other problems.)

```
void dashedLine(){
  // Add this function to your sketch.
  // Use it anytime you wish to display a long string of dashed lines.
  Serial.println("----------------------------------------");
}
```



2-11. Create a new void function named "**countdown**". Add some `print` and `delay` functions within **countdown** so the numbers 5, 4, 3, 2, 1, and 0 are displayed with a one-second delay between the numbers. You can see an example of this output via my YouTube channel at https://youtu.be/oLbxcz1FAGI .

2-12. Create four new void functions named "**myName**", "**myAddress**", "**myEmail**", and "**myNumber**". Within each function, write code that will display to the Serial Monitor your name, your home address, your email address, and your telephone number, respectively. Next, create another new function and name it "**personalData**" or "**P12**" or whatever. Finally, from within that function you just created, call the four functions above one at a time so your personal information appears on the Serial Monitor. A screenshot of my output is shown below to serve as a visual reverence.



2-13. Examine the code window below and determine if the sketch has been saved recently or if it needs to be saved.