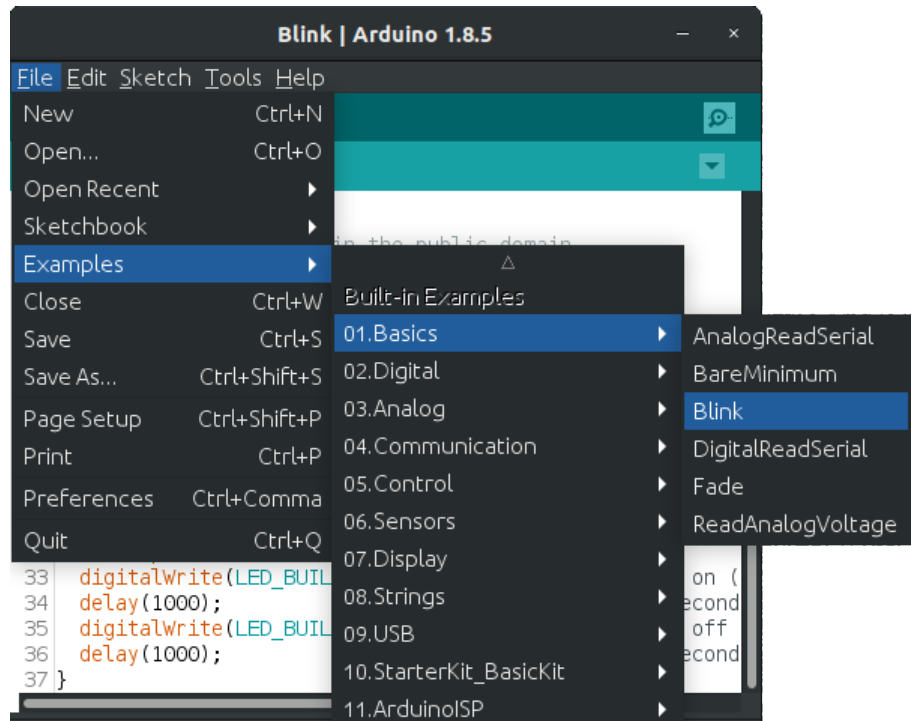


Generally speaking in C++ programming, a designer writes human readable code, and when they are ready to test it, they use a tool called a “compiler” to check whether or not the code is written following basic language rules called the language “syntax”. After the syntax check the compiler compiles or converts the code into binary which is ready for the target processor to run. C++ is a versatile programming language and can be used on desktop computers as well as microcontrollers, though microcontrollers have the additional step of uploading the binary code to the controller before it can be run.

The Arduino Uno uses the Arduino Integrated Development Environment (IDE) to check the program syntax, compile, and upload to any Arduino compatible microcontroller. Very commonly the program is not perfect after the first writing it, so testing, debugging, and reuploading is usually part of the development process.

In programming people sometimes write a “hello world” program or “blink” for Arduino that simply shows the designer that their programming system is working. In order for the end program to be generated and run properly, several underlying tools need to work together, and at times there can be various problems stopping the host computer from compiling or uploading the code to the Arduino. A simple program blinking an led can be used as a basic check to make sure that the IDE is installed properly and is able to

communicate with the target controller. The Arduino IDE has a lot of premade programs that can be used for testing by a designer, and one of them is a basic blink program great for doing this basic connection test. Controllers can be damaged, cables can be damaged, and sometimes a program itself can become so complex that it becomes unclear whether or not the basic connection to arduino is working properly, so a simple blink is a great test to make sure the host computer is connected properly to the controller. To reach



A program is instructions for what a processor should do. Simpler computers like the Uno can’t do multiple tasks at once, but they make up for it by being able to do many operations very quickly. A designer must write code carefully however because while computers are machines capable of many sophisticated tasks, they execute code exactly as written. Lets see how to control program flow.

[DATA TYPES]

Recall that every bit of information in digital devices must be broken down and represented using binary. C++ generally and the Arduino Uno specifically have several types of data that it is able to represent using binary since different types of numbers are useful for different contexts, for example sometimes fractional values are needed and other times only a number for counting is needed. Here are

a list of some commonly used basic data types and what they're used for. All of these data types can be stored as a variable or used directly as a "literal".

- **bool** – The most basic unit of binary data, representing a logic TRUE/FALSE. A single logic 1 or 0 is called a "bit". Generally if any other data type has any binary 1 in it, the whole variable can be treated as a boolean true, where if all bits are zero, then it can be treated as false.
- **byte** – a collection of 8 "bits" creates a byte, an unformatted 8 bit number from 0 to 255.
- **int** – A 16 bit number representing whole signed numbers, meaning positive or negative. The numbers that can be represented with a 16 bit signed int are from (-2^{15}) to $(2^{15}-1)$ or -32768 to 32767. When declaring an int variable, unsigned can be specified to reserve all 16 bits for positive numbers, creating a range of 0 to 65535.
- **long** – A 32 bit integer number for working with numbers larger than a normal int can represent. Long also comes in signed and unsigned for representing positive/negative whole numbers or for reserving all 32 bits for positive values. An unsigned long can represent numbers from 0 to 4294967295. Earlier examples use unsigned long to store a timestamp representing how long an Arduino has been running in milliseconds. To compare, a millisecond timer would reach an unsigned int max value in around 66 *seconds*, where it would reach the number cap of an unsigned long in around 50 *days*.
- **float** – A 32 bit number representing fractional values with decimal points. The data formatting for this is more complex than integers and harder/slower for 8 bit controllers like the Uno to operate with, so work with floats are often discouraged in this context for their inefficiency. Floats can represent values as large as 3.4×10^{38} and as low as -3.4×10^{38} . More generally, C++ also has a type called "double" which is a floating point number that is double the bit size of a float, 64 bits for higher precision. However because of the limitation of the Uno and other ATmega controller processors, when doubles are created in the Arduino IDE the system uses 32 bit floats instead of the full 64 bit representation. This means that doubles aren't normally usable on an Uno, where normal floats are just discouraged for their computational inefficiency.
- **char** – An 8 bit number used to represent individual text characters. The 8 bits are stored as a signed number that represent characters using the ASCII encoding scheme. The numerical encoding for each character can be found on an ASCII chart, for example 65 is 'A', 97 is 'a', and 126 is '~'. Since the character encoding is based on a number, it's possible to do math on chars, for example 'A' + 1 would be 'B', which is ASCII 66. Chars are notes with single quotes.
- **string** – A collection of chars to create words or phrases. Generally there are two kinds of strings, C-strings, and string objects denoted as String(). A C-string is directly a series of chars with a 'null' character at the end of the series to show the computer that it has reached the end of the series. On the other hand, a String() object is a complex data type that contains a c-string along with extra functions that simplify certain string operations. The tradeoff is that c-strings are more memory efficient, but all operations on it must be done manually, where String() objects can be easier to program with, but take up significantly more space in program memory. Generally speaking String() object use is discouraged with 8 bit controllers like the Uno because of it's relatively low amount of program space and RAM. Strings are noted with double quotes.
- **Arrays** – Arrays are not a data type themselves, but instead are groups of the previous stated data types. A designer can make a boolean array which would be a series of boolean values, a c-string is directly an array of chars with a null char to signal the end of the string. A particular value in an array can be used by referencing its position in the array starting from position 0.

```

dataTypes
1 bool myBool = 0;
2 int myInt = 12;
3 float pi = 3.14;
4 char aCharacter = '&';
5 char myCString[] = "I'm a C-string. An array of chars";
6 String myStringObject = "I'm a big fat string";
7
8 void setup() {
9     Serial.begin(9600);
10
11     Serial.println(myBool);
12     Serial.println(myInt);
13     Serial.println(pi);
14     Serial.println(aCharacter);
15
16     Serial.println(myCString);
17     Serial.println(myCString[6]);
18     Serial.println(myStringObject);
19 }
20
21
22
23
24 void loop() {
25     delay(1000);
26 }

```

In this screenshot, some variables of all the discussed types are created with their respective names and are printed to the serial monitor screen in setup. The loop then does nothing. On line 5 a c-string is created which shows how to create arrays of any data type, and line 17 shows how a specific position in the array can be addressed. Below is a screenshot of what gets printed to the serial monitor.

```

0
12
3.14
&
I'm a C-string. An array of chars
C
I'm a big fat string

```

[IF STATEMENTS]

When a computer isn't just doing math, it's making decisions based on some condition. The basic form is a simple "if" check. If checks have optional "else if" and "else" blocks for subsequent condition checks and a default case that runs when all condition checks are false. Here is a screenshot of a program comparing a variable containing the number 360 to the number 180 written as a literal (direct instead of variable). Like before, all of this is done in setup to execute once and just a delay is put in the loop function along with a view of what was printed to the serial monitor. There are a series of if checks using all of the standard logical comparisons available.

Another type of condition check called a "switch case" is available in C++ and has slightly different behavior, but generally it can be thought of as equivalent to a series of normal "if" checks.

```

condition §
1 int number = 360;
2 void setup() {
3     Serial.begin(9600);
4
5     // == (equal to)
6     if( number == 180 ){
7         Serial.println("== pass");
8     }
9     else{
10        Serial.println("== fail");
11    }
12    // != (not equal to)
13    if( number != 180 ){
14        Serial.println("!= pass");
15    }
16    else{
17        Serial.println("!= fail");
18    }
19
20    // < (less than)
21    if( number < 180 ){
22        Serial.println("< pass");
23    }
24    else{
25        Serial.println("< fail");
26    }
27    // <= (less than or equal to)
28    if( number <= 180 ){
29        Serial.println("<= pass");
30    }
31    else{
32        Serial.println("<= fail");
33    }
34
35    // > (greater than)
36    if( number > 180 ){
37        Serial.println("> pass");
38    }
39    else{
40        Serial.println("> fail");
41    }
42    // >= (greater than or equal to)
43    if( number >= 180 ){
44        Serial.println(">= pass");
45    }
46    else{
47        Serial.println(">= fail");
48    }
49 }
50 }

```

[LOOPS]

While simple computers like the Uno can be thought of as dumb for only being able to do one thing at a time, its ability to use a fast clock to loop operations very quickly can make up for those shortcomings. Take the Uno pinout which has 14 digital only pins that can be used for lighting up leds. A designer can use a trick to increase the number of leds usable with those pins from 14 to 49 by arranging the led connections into a grid pattern with 7 rows and 7 columns. This would mean multiple leds can't be lit at the same time, however each led can be scanned quickly enough to make them appear to light at the same time to the human eye.

There are 3 basic methods for looping, the “for” loop, which is used when the designer knows how many time they want a loop to run, a “while” loop, which is used for looping an unknown number of times, and a “do...while” loop, which is similar to a normal “while” loop but always executes its contents at least once.

A “for” loop has 3 parts to its mechanics, a starting count value, a condition to check every time the loop restarts, and an operation that runs after each time the loop finishes. The “for” loop repeats continuously until the condition evaluates to false. “While” and “do...while” loops only have the condition check, but “do...while” has the body of code before the condition check. Here in an example program with its serial output to show these loops behaving.



```
1 loops
2
3 void setup() {
4   Serial.begin(9600);
5
6   // start    condition    increment
7   for( int count = 0; count < 10; count++) {
8     Serial.print(count);
9   }
10  Serial.print('\n'); // adds a newline.
11
12  int whileCounter = 9;
13  while( whileCounter > 0 ){
14    Serial.print(whileCounter);
15    whileCounter = whileCounter - 1;
16  }
17  Serial.print('\n'); // adds a newline.
18
19  int doCounter = 5;
20  do{
21    Serial.println("runs at least once");
22  }
23  while(doCounter > 6);
24 }
25
```

0123456789
987654321
runs at least once

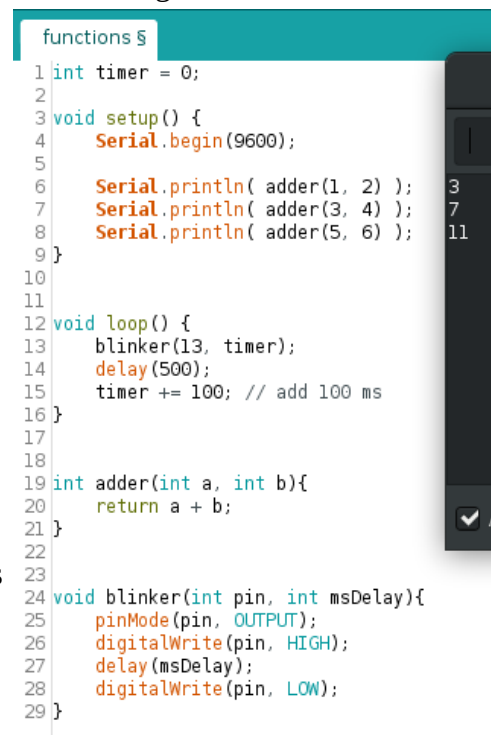
Autoscroll

[FUNCTIONS]

The “setup” and “loop” blocks of code are called functions, and a designer can write their own functions that have their own contents to run in a program.

Functions can have inputs and outputs, but like variables, they have to be “declared” before being able to use them. Normally in C++ variables and functions must be declared at the top of the program before any use anywhere else, but the Arduino IDE does some fancy work to make them accessible if declared after the setup and loop functions. The “setup” and “loop” functions have no output, which is why they both have “void” as their output type. The parentheses by those function names are where a functions inputs can go, which setup and loop have none.

This example program demonstrates two functions, one that has two integer inputs and outputs their sum using the “return” keyword. This means that when the function runs, it executes its contents inside the brackets and is replaced with whatever the returned value is. The other function is a void function that takes in a pin number and a delay duration and turns on that pin number for the given delay duration. The setup function uses the adder 3 times to display its behavior and then the loop function runs the blink and continuously increases the blink on-time in 100ms steps.



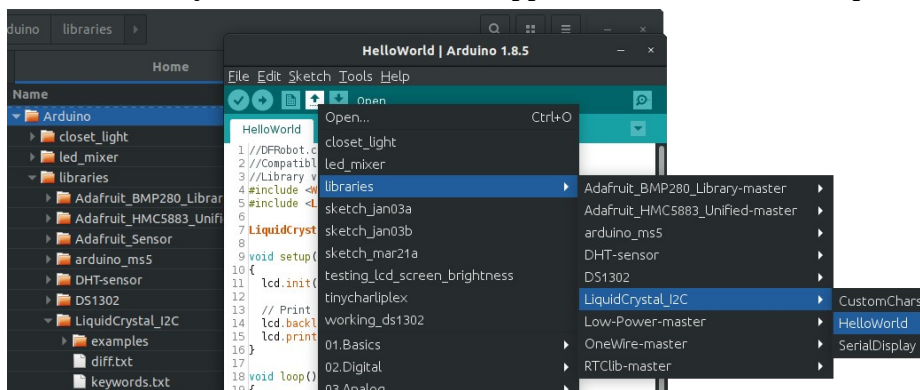
```
1 functions
2
3 int timer = 0;
4
5 void setup() {
6   Serial.begin(9600);
7
8   Serial.println( adder(1, 2) );
9   Serial.println( adder(3, 4) );
10  Serial.println( adder(5, 6) );
11 }
12
13 void loop() {
14   blinker(13, timer);
15   delay(500);
16   timer += 100; // add 100 ms
17 }
18
19 int adder(int a, int b){
20   return a + b;
21 }
22
23
24 void blinker(int pin, int msDelay){
25   pinMode(pin, OUTPUT);
26   digitalWrite(pin, HIGH);
27   delay(msDelay);
28   digitalWrite(pin, LOW);
29 }
```

3
7
11

[LIBRARIES]

A library is a package of functions and variables that is created to simplify certain actions. Our example payload uses several libraries. SPI and SD libraries contain functions for using the SPI protocol and using micro sd cards. Wire is a library for communicating using the I²C port. Many sensors have libraries that are created by the supplier to simplify use of the sensor. A designer can write their own libraries as well just as they can write their own functions, but at a lower programming experience level usually the designer will use already made libraries as opposed to creating their own. Because of this, we will simply cover how to use an already existing library, our subject will be a basic 16X2 character LCD screen controlled with I²C.

I downloaded this LCD screen library from DFRobot, though many others are available from other sources, and most of the time libraries come in zip files. The library can be added to the Arduino IDE by clicking the menus Sketch → Include Library → Add ZIP Library, but sometimes this can have issues working, so let's look at how to add libraries manually. Locate the Arduino folder on the host computer, the location will differ based on operating system. Windows and Mac has the Arduino folder in the Documents folder, and Linux has the Arduino folder in the Home folder. Inside the Arduino folder, there is a “libraries” folder, extract the contents of the library zip file here. The direct contents of that new library file should be some “.cpp” and “.h” files and an optional “examples” folder. The



Arduino folder can also be located by checking the location in the IDE preferences. After the new library folder and files have been moved into the IDE library folder, restart the IDE and click the “open” button, and the new library can be found under the “libraries” menu.

```
1 #include <Wire.h>
2 #include <LiquidCrystal_I2C.h>
3
4 LiquidCrystal_I2C lcd(0x27,16,2); // set the LCD address to 0x27 for a 16 chars and 2 line display
5
6 void setup()
7 {
8   lcd.init(); // initialize the lcd
9
10  // Print a message to the LCD.
11  lcd.backlight();
12  lcd.print("Hello, world!");
13 }
```

Here is a “hello world” example that was included with the library that simply prints the hello statement on the screen, demonstrating that the library and LCD screen itself is working properly. Lines 1 and 2 have “include” statements that are used

for allowing a program to access the contents of a particular library. Here, the example program includes the Wire library for using the I²C hardware port on the Uno board, and the “LiquidCrystal_I2C” line is for adding the LCD library to this program. Line 4 creates an instance of the library called “lcd” and initializes this instance by setting the screen address to “0x27” and telling the library that the screen is 16 chars wide with 2 lines. If there were multiple instances needed, like if using 2 barometers using the same library, the instance names must be different the same way that two variables can’t have the same name because the computer won’t know the difference between the two. Since the instance of the display library is named “lcd”, that name can then be used to access all of the functions contained inside the library. Line 8 initializes the LCD screen hardware using a function called init() that’s part of the library, the backlight() function on line 8 turns the screen backlight on, and lcd.print(“Hello, world”) statement on line 12 actually prints the greeting to the screen. Often times the example programs won’t show the full capabilities of the library, so it can be useful to have a peak inside the “.h” or header file in the library to find all available functions. Here is a peak inside.

```
55 class LiquidCrystal_I2C : public Print {
56 public:
57     LiquidCrystal_I2C(uint8_t lcd_Addr,uint8_t lcd_cols,uint8_t lcd_rows);
58     void begin(uint8_t cols, uint8_t rows, uint8_t charsize = LCD_5x8DOTS );
59     void clear();
60     void home();
61     void noDisplay();
62     void display();
63     void noBlink();
64     void blink();
65     void noCursor();
66     void cursor();
67     void scrollDisplayLeft();
68     void scrollDisplayRight();
69     void printLeft();
70     void printRight();
71     void leftToRight();
72     void rightToLeft();
73     void shiftIncrement();
74     void shiftDecrement();
75     void noBacklight();
76     void backlight();
77     void autoscroll();
78     void noAutoscroll();
79     void createChar(uint8_t, uint8_t[]);
80     void setCursor(uint8_t, uint8_t);
81 #if defined(ARDUINO) && ARDUINO >= 100
82 | virtual size_t write(uint8_t);
83 #else
84 | virtual void write(uint8_t);
85 #endif
86     void command(uint8_t);
87     void init();
```