Arrays

If variables can be thought of as buckets that hold a single piece of information, then arrays can be thought of as a collection of buckets, or a big bucket with a lot of little buckets inside. Arrays are extremely useful for a lot of different programs – basically, any time you want to perform a similar operation on several variables (of the same type) – you should consider putting the variables in an array. For example, if I want to blink eight LED's at the same time, I could put them in an array, and then use a for loop to iterate over the array, like so:

```
/* this is the array that holds the pin numbers our LED's would
be connected to */
int ledPins[] = {2,3,4,5,6,7,8,9};


// in setup()we can set all the pins to output with a simple for
loop
// 8, because we have 8 elements in the array
for( int i = 0; i < 8; i++) {
// sets each ledPin in our array to OUTPUT
 pinMode(ledPins[i], OUTPUT);
}
```

The `ledPins[i]` part is important; it allows us to reference each element in our array by its place in the array, starting with 0 (which can be confusing). So, in our example above ledPins[0] == 2, since 2 is the 1st element we put into the array. This means that ledPins[1] == 3, and ledPins[7] == 9, and is the last element in our array.

See http://arduino.cc/en/Reference/Array for further explanation.

Data Types

In addition to the data types already covered like integers, booleans, and characters, there are some more data types that may prove useful for your specific application.

Float: Data type for floating point numbers (those with a decimal point). They can range from 3.4028235E+38 down to -3.4028235E+38. Stored as 32 bits (4 bytes). A word of advice: floating point arithmetic is notoriously unpredictable (e.g. 5.0 / 2.0 may not always come out to 2.5), and much slower than integer operations, so use with caution. Some readers may be familiar with the '**Double'** data type – currently, the Arduino implementation of Double is exactly the same as Float, so if you're importing code that uses doubles make sure the implied functionality is compatible with floats.

Long: Data type for larger numbers, from -2,147,483,648 to 2,147,483,647, and store 32 bits (4 bytes) of information.

String: On the Arduino, there are really two kinds of strings: strings (with a lower case 's') can be created as an array of characters (of type `char`). String (with a capital 'S'), is a String type object. The difference is illustrated in code:

```
Char stringArray[10] = "SparkFun";
String stringObject = String("SparkFun");
```

The advantage of the second method (using the String object) is that it allows you to use a number of built-in methods, such as length(), replace(), and equals().

More methods can be found here: http://arduino.cc/en/Reference/StringObject

Advanced Programming Concepts

SparkFun Electronics Summer Semester

Pointers

Pointers seem to be one of the concepts that gives some people a bit of trouble. In the simplest terms, a pointer is a memory address. In C-based languages (including Arduino), when a variable is created, like so:

```
int foo;
```

A certain chunk of memory (how big a chunk is determined by the data type) is allocated to storing the value of that variable. That chunk of memory also has an address. Most of the time we, as programmers don't have to deal with the address, but sometimes dealing directly with a pointer to an address is more efficient. To declare a variable to be a pointer, use an asterisk (*) before the variable name like so:

```
int *pnt;
```

Typically, we would say that `pnt` is declared as a pointer to int. If we want to assign our pointer to the memory address of foo, we can use the & operator, (usually called the *reference* or *address-of* operator), like so:

```
int *pnt = &foo;
```

This tells our pointer to point to the address of our foo variable (not the value of it).

To get or assign a value with a pointer, you need to use the dereference operator (*). Assigning the value from our pointer to a regular integer looks like this:

```
int foo2 = *pnt;
```

Assigning a number value to our pointer can be done like so:

```
*pnt = 100; //sets whatever pnt is pointing to a value of 100
```

Note that we cannot simply say:

```
pnt = 100;
```

This would change the memory address of the pointer to 100 – and since there's probably nothing at memory address 100, we would get an error.

There's plenty more on pointers on the internet, mostly within the C language.

Note that in Arduino programming, you may never have to use a pointer, although for certain operations it will simplify your code – but only if you know what you're doing.

Sending and Receiving data in different formats

One of the common mistakes when attempting to get two devices to communicate with each other is the improper formatting of data. While data formatting is a very broad discussion, we have a few specific tips for the Arduino environment to make sure that you don't make this mistake.

Normally, when we do a print statement in Arduino we just say `Serial.print(foo)` or `Serial.println(foo)`, which sends human-readable ASCII text over the serial port. However, the Arduino print commands have a second argument that allows you specify how you want to encode the data. So, for example (from the Arduino reference):

```
Serial.print(78, BYTE) // gives "N"
Serial.print(78, BIN) // gives "1001110"
Serial.print(78, OCT) // gives "116"
Serial.print(78, DEC) // gives "78"
Serial.print(78, HEX) // gives "4E"
```

For floating point numbers, the second parameter specifies the number of digits after the decimal place to include when sending – note that the default for sending floating point numbers cuts off at 2 places past the decimal point:

```
Serial.println(1.23456, 0) // gives "1"
Serial.println(1.23456, 2) // gives "1.23"
Serial.println(1.23456, 4) // gives "1.2345"
```

Switch/Case statements

Sometimes you've got a bunch of if statements that all check the same variable. It is possible to combine all these if statements into one statement called a Switch/Case statement. The Switch/Case statement will check the value of a given variable against the values specified in the case statements and execute code if the values match. So for example (from the Arduino Reference):

```
switch (var) {
      case 1:
       //do something when var equals 1
     break;
     case 2:
       //do something when var equals 2
     break;
     default:
       // if nothing else matches, do the default
       // default is optional
}
```

The example above uses an integer variable called var and has cases that check if var is equal to 1 or 2 with a default to handle all the other values it could have. Here is some pseudo-code to explain the syntax:

```
switch (var) {
      case label:
      // code that executes when var is equal to label
      break;
      case label:
      // code that executes when var is equal to label
      break;
      default:
      // code that executes when var is equal to label
}
```

Advanced Programming Concepts

SparkFun Electronics Summer Semester

Break indicates that the computer should exit the entire switch/case statement and continue with the rest of the code in the sketch. Break is not strictly necessary when writing a switch/case statement, but if you do omit it the computer will go on to check the rest of the cases until it encounters a break or checks every single case. In this example var is the only text that you might change so that it matches the variable you are checking against. Default is optional, so you can omit it, but usual you will want to use this as a catch all in case your variable winds up with a value you have not listed in the various cases.

Constants and Define statements

If we know that the value of a variable is never going to change (or we know we don't want it to change), we have the option of declaring it a *constant.* As in, its value remains constant throughout the program.

There are a two common ways to do this: #define or const.

A define statement appears at the beginning of your program, and looks like this:

```
#define ledPin 3
```

Notice the '#', the lack of an assignment operator (no = sign), and no semicolon (;). Omitting the #, or introducing = or ; into the statement are all errors. The main advantage to using a define statement is that the constant doesn't take up any extra memory space on the chip – the compiler simply goes through the program and replaces every instance of your variable (`ledPin` in this case) with the value you defined for it (`3` in this case).

There is a downside to using define statements, however: if for example, a constant name that you #defined is included in some other constant or variable name, the text would be replaced by the #defined number (or text). Bad news.

For this reason, many programmers prefer using the `const` keyword. You can think of the `const` keyword as a modifier before declaring any variable, so the syntax looks like:

```
const int ledPin = 3;
```

This declares a constant integer value of 3 for our ledPin variable. If we try to modify it like so:

```
ledPin = 4;
```

We will get an error. However, you can use the constant value in math:

```
X = ledPin * 4;
```

Generally, define statements are fine, but using `const` is considered the preferred, safer method for declaring constants if memory space is not a concern.

Bitwise Operations and Bit Shifting

Sometimes it becomes necessary to work on the binary level, and affect changes through bitwise operations and bit shifting (for an example, check out the multiplexor code (circuit 5) from the SparkFun Inventors Kit http://www.oomlout.com/a/products/ardx/circ-05).

Note that the notations for bitwise operator are different than their logical counterparts (& instead of &&, for example).

There are four bitwise operators: NOT, AND, OR, and XOR.

The NOT operator (~) simply flips the value of each bit:

```
NOT 1010
  = 0101
```

The AND operator (&) takes two binary representations (of equal length) and preforms a logical AND operation on each corresponding pair of bits. If the first bit is 1 AND the second, corresponding bit is 1, then the result is 1 – otherwise, the result is 0. So:

```
    0101
AND 0011
  = 0001
```

The OR operator (a single pipe, |) takes two binary representations and performs a logical OR operation on each pair of corresponding bits. So, if the 1st bit OR the 2nd bit is a 1, the result is 1. For example:

```
   0101
OR 0011
 = 0111
```

The exclusive or operator XOR (^) performs the logical XOR operation on two binary representations of equal length. The result is 1 if *only* the first bit or *only* the second bit is 1, but returns 0 if *both* bits are 1 or *both* bits are 0. For example:

```
    0101
XOR 0011
  = 0110
```

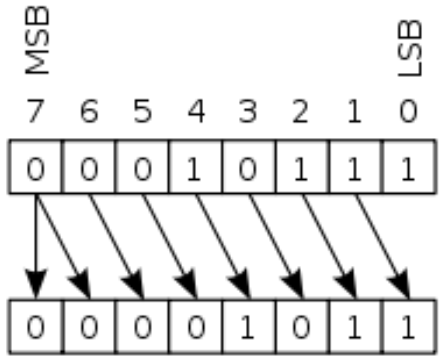
Bit Shifts, as opposed to bitwise operators, occur on only a single binary representation (not a pair). Several concepts become important when introducing bit shifting: the most significant bit (MSB) and least significant bit (LSB). The most significant bit is the bit position in the binary number that has the greatest value (typically the bit furthest to the left). The least significant bit is that bit with the lowest numerical value.

There are four typical types of bit shifts that you may encounter: Arithmetic Shift, Logical Shift, Rotate No Carry, and Rotate Through Carry. Each type of shift can either be shift-right or shift-left, indicating which direction the bits are being shifted.

Arithmetic Shifts

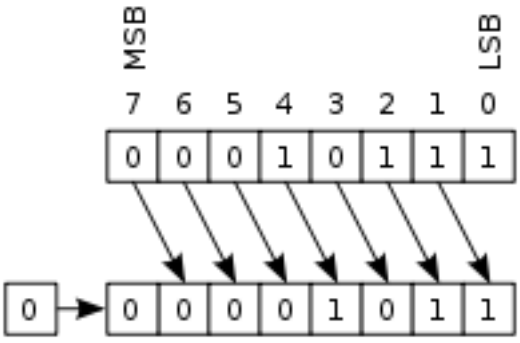
Arithmetic Shifts shift out the bit at either end and discard it. In a left shift, all the bit values shift to the left, and a 0 is inserted into the new place (on the far right). In a right shift, the sign bit is shifted in on the right (the sign bit is usually a copy of the MSB at the time, also defining the sign of the number in some cases).

The following is an arithmetic shift right:

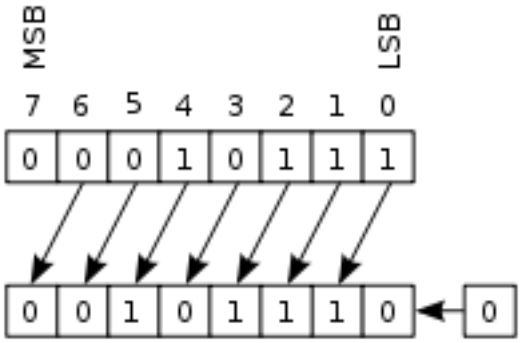


Logical shifts are very similar to arithmetic shifts, with bits shifted out being replaced by 0. In the case of a left-shift, arithmetic and logical shifts are exactly the same. In the right shift however, a 0 is always inserted with no attention paid to the sign bit.

Notice the difference in a right logical shift:

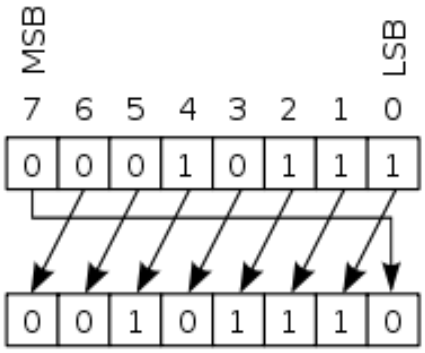


And here is a left logical shift:



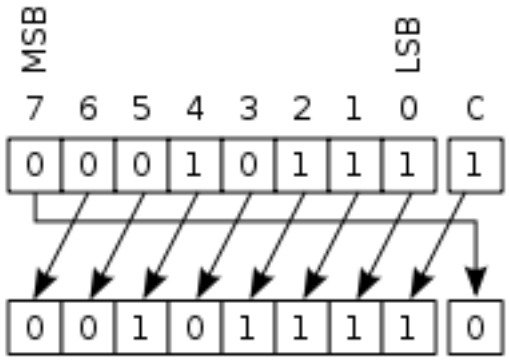
Next we have Rotate No Carry (also called Circular) and Rotate Through Carry shifts. In a Rotate No Carry shift, the bits are rotated as if they were joined at either end – the bit that gets shifted off gets moved around to the other side instead of being discarded.

Pictured is a rotate left bit shift (no carry):



The difference between rotate through carry and rotate no-carry refers to the presence (or absence) of a carry flag. The carry flag is basically a holder for another bit, and the holder can be thought of as in between the two ends of the register. This means that whatever bit gets shifted out get placed into the carry flag, and the old value that was in the carry flag gets shifted in on the appropriate side (depending on left or right shift).

Here's a diagram of a left rotate through carry:



Notice how the carry bit (labeled 'C') gets shifted in and replaced by the bit being shifted out.

Fortunately, in C++ (and in Arduino) there are only two bit shifting operations, bitshift left (<<<) and bitshift right (>>>).

Luckily, these operations are equivalent to the 'arithmetic' shift mentioned above. For example:

```
int a = 5;          // binary: 0000000000000101
int b = a << 3;     // binary: 0000000000101000, or 40 in decimal
int c = b >> 3;     // binary: 0000000000000101, or back to 5
like we started with
```

So there's your (not so) brief introduction to bit wise operations and bit shifting.

Casting

Say you want to map a sensor value from it's normal range (0 to 1023) to a new range (0 to 100). You want to use the map function because it does all the math for you (yay!), but you can't because your sensor value is an integer and the map function requires a float (in Arduino it doesn't, but in Processing it does). Are you stuck? No! Although you could go back and change your code to make all the integers into floats, you can also perform a *cast* – translating one variable type into another.

It looks something like this:

```
int sensorValue = analogRead(A0); //our integer variable

float newValue = (float) sensorValue;  //new float variable
takes the int and casts it into a float – neat!
```

Basically, the rule is to put the data type that you want your variable to become in parentheses when assigning it to a new variable.

For our earlier example, we could save a step by casting our variable directly in the map statement, like so:

int sensorValue = analogRead(A0);

sensorValue = map((float)sensorValue, 0, 1023, 0, 100);

Note that there are sometimes consequences for casting. For instance, when casting from a float to an integer, the decimal places will be lost (not rounded) – so 3.7 becomes 3, as does 3.1.

Bootloaders and ICSP

Every Arduino board that you buy comes with a small program – called a bootloader - already pre-programmed onto the chip. The Arduino bootloader (there's a few different ones depending on the chip) allows you to write and upload programs onto the chip from the Arduino software environment.

ICSP (In-Circuit Serial Programming) is simply the method by which you can (if you wish) directly access the chip that is the 'brain' of the Arduino board (Usually an ATmega of some kind). This is handy to know about for a few reasons: if you want to replace the ATmega chip in your Arduino, you can simply buy a new chip, put it in the Arduino board and burn the Arduino bootloader onto it (instead of buying a whole new board - much more cost effective). Also, if you need to clear 1-2KB from the flash memory of your chip to fit your program on, removing the bootloader is one way to do it.

In order to make use of external programmers using ICSP, you'll need do to one of three things:

- Get an AVR-ISP (http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2726)
- Get a USBtinyISP (http://www.ladyada.net/make/usbtinyisp/)
- Build a Parallel Programmer (http://arduino.cc/en/Hacking/ParallelProgrammer)

A great page to started with the bootloader and ICSP with Arduino: http://arduino.cc/en/Hacking/Bootloader

Enjoy!