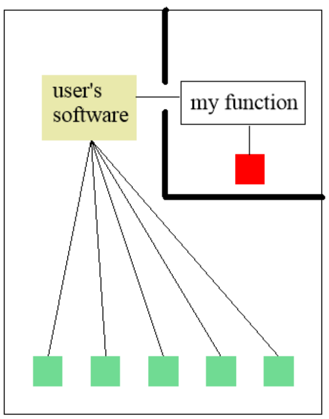
Arduino C Pointers, Version 1.2

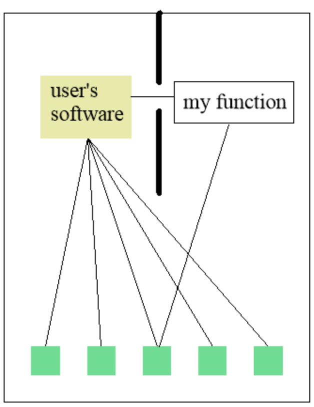
By R. G. Sparber

What and Why?

I'll answer both of these questions by giving an example.



Consider the standard way to use a function. The "user's software" calls "my function" and passes a number stored locally, a green box. While the function is running, the operating system allocates a memory location, the red box. The function stores the number in the red box and then executes its code which changes this number. Finally, the function reads the red box's contents and passes it back to the user, who places it back into its green box. When the function terminates, the operating system deallocates the red box, so it vanishes.



An alternate method is to pass the address of the user's variable into the function. Then the code in the function can directly access this variable with nothing passed back and forth.

We pass this address inside a bit of memory designated as a ***pointer***.

The pointer can point to a single variable or an array. The array can contain data or, strap on your seat belt, be filled with pointers to other memory locations. In this way, a single pointer provides access to a limitless number of variables and arrays. Yes, it is a powerful and potentially confusing tool.

Disclaimer

All of the code presented in this article has been compiled without error and tested for functionality. However, this doesn't mean I have used the best coding standards or correctly understood the underlying rules.

Why Did I write this?

There are many satisfactory tutorials on pointers, but none perfectly worked for me. I would read one, understand, and then the concept would float away. After a long time, I realized that my "mental model" was incomplete. This article documents how I think about pointers, so they make sense to me.

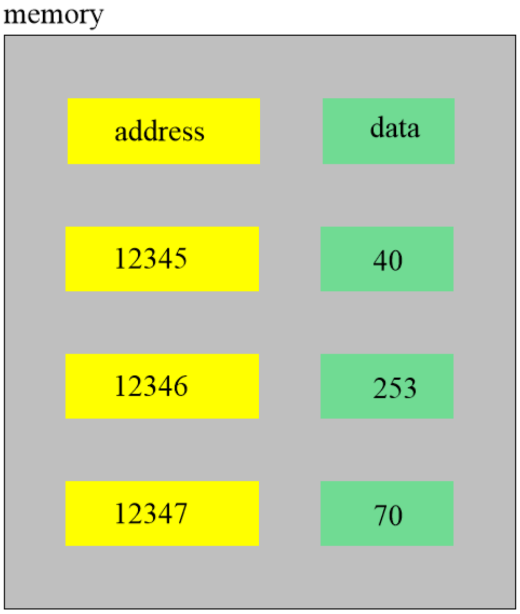
These references were helpful:

Pointers, Arrays, and Functions in Arduino C (engineerworkshop.com)
C++ Passing Arrays to Functions - Tutorialspoint
C++ Pointer to an Array - Tutorialspoint
Classes (I) - C++ Tutorials (cplusplus.com)

Contents

Hardware with Memory

I have a hardware background, so I need to anchor myself to the hardware before I venture out on this thin ice called software.



Memory consists of two parts: address and data. Give an address, and it will return the data associated with that address.

For this article, I will assume that an integer can hold the address, and the data is a single byte.

Notice that the hardware has no understanding of variable names. It only deals with receiving an address and returning a byte. It also does not know other data types. If I want to store a variable that uses four bytes, the rule for combining them must be in the compiler. Memory is a "one-trick pony," yet it is hard to find software that doesn't depend on it.

A Variable

A variable has a ***name***, ***address***, and a ***value***. The name is used in a function and is for our benefit. The compiler replaced each name with an address. At that address is a value. By using the name, we can read and write the value.

Commonly, the address points to the first byte, and the compiler keeps track of how many bytes to read as it assembles the value. If the value is one byte, the address points to that one byte. If the value takes up more than one byte, the compiler figured that out, and we don't have to worry about it.

I have a variable called myHatSize. The compiler assigned it the address 28123. The user sets the value to 7.5. Since I defined this variable as a float, it takes up four bytes.

A Pointer To A Variable

Recall that a pointer tells the software where to find a previously stored piece of data stored at an address. There are a few pieces to this puzzle.

The first piece of the puzzle is to set up the variable.

I write

```
float myHatSize;
```

and the compiler will define a group of bytes to hold a number. This group of bytes is accessed by knowing its name, `myHatSize`. As long as the calling function knows that it is a float, all is well. In this example, the compiler chose the address 28123.

With `myHatSize` defined, I can fill this memory location with a value

```
myHatSize = 7.5;
```

During the compilation of the function, the name of this variable is replaced by its address. If another function tries to use this name, the compiled first function will have no idea what it means. Only the address has meaning. This address is selected during compilation. If I change the code and recompile, the address can change.

The next piece of the puzzle: finding the address of a variable.

I find the address of a variable by placing "`&`" directly[2] in front of it.

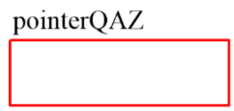
```
&myHatSize
```

[2] The compiler doesn't care if there is a blank space between & and myHatSize but it is a matter of style. When & is used for bit-wise ANDing of two numbers, we put a blank on each side of it (eg: A & B). Similarly, * with blanks around it used used in multiplying (eg. A * B). You will soon see that *myHatSize has a special meaning.

`&` "extracts" the address from `myHatSize`. Knowing this address tells me where to look for the contents of `myHatSize`. If this second function is given this address, it can access `myHatSize`'s data without ever knowing its original name. Notice that `&` is an operator that works on `myHatSize`. It does not define a new name.

Now that I know how to extract the address of a variable, we get to the third piece of the puzzle – creating a pointer.

My functional model[3] of a pointer is



The red box symbolizes a specially designated memory location. It is known to the compiler as containing a variable that it will use as an address.

As I create a pointer, I must tell the compiler about its target. It might be a byte, integer, float, boolean, etc. Then the compiler will know how many bytes to read as it assembles the value.

Say I want to create a pointer, called pointerQAZ, that will point to a float variable. I would type

```
float *pointerQAZ;
```

That * tells the compiler that I want to create a pointer.

I now have the pointer named `pointerQAZ`.

What does `pointerQAZ` contain? I haven't the slightest idea. It is whatever number that happened to be at that address. It is good practice to initialize the pointer to a known value. In this way, any odd behavior caused by using the pointer before you put in a valid address will be consistent, even though it is wrong.

I initialize the pointer to zero with

[3] This is how I visualize how pointers work. Exactly how the compiler does it is not relevant to me.

```
float *pointerQAZ = 0;
```

This is also called a null pointer because it points to address 0.

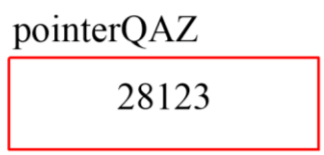
The fourth piece of the puzzle is how to place a variable's address inside a pointer.

Recall that I extract a variable's address by placing `&` in front of it. To place `myHatSize`'s address into our pointer named `pointerQAZ`, I type

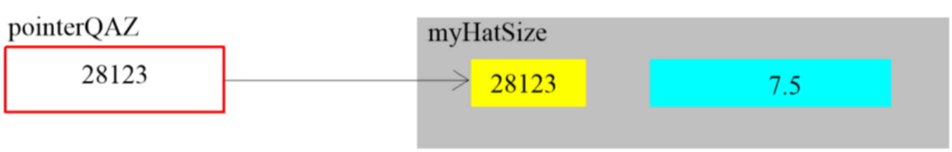
```
pointerQAZ = &myHatSize;
```

From page 4 we know that `myHatSize` was assigned address 28123 by the compiler.

We end up with



The last piece of the puzzle is using `pointerQAZ` to access `myHatSize`.



I do this by placing a * in front of the pointer's name:

```
*pointerQAZ
```

`*pointerQAZ` is now interchangeable with `myHatSize.`

I can write

```
myHatSize = 81.3;
```

and my value of `7.5` is replaced by `81.3`.

```
*pointerQAZ = 81.3;
```

does the exact same thing.

To recap,

I defined a variable and set it's value: `float myHatSize = 7.5;`

I defined a pointer to a float: `float *pointerQAZ = 0;`
I set my pointer to point to my variable: `pointerQAZ = &myHatSize;`

I used my pointer to access my variable: `*pointerQAZ = 81.3;`

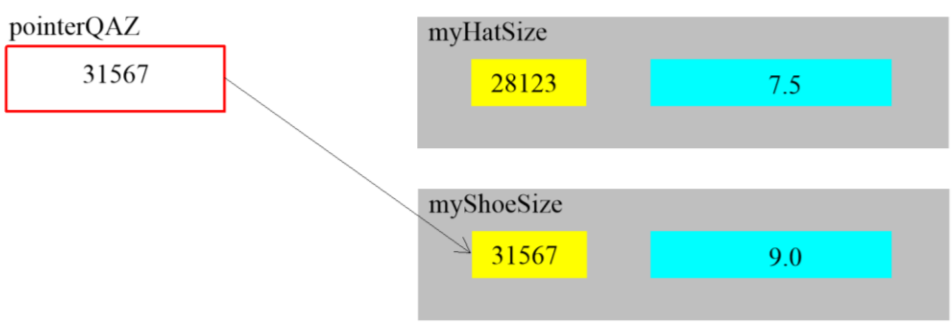
I can write a function that uses `*pointerQAZ`. By changing the contents of this pointer, I am accessing any variable I want. For example, when I had

```
pointerQAZ = &myHatSize;
```

I was accessing the data contained in `myHatSize`. If I then write

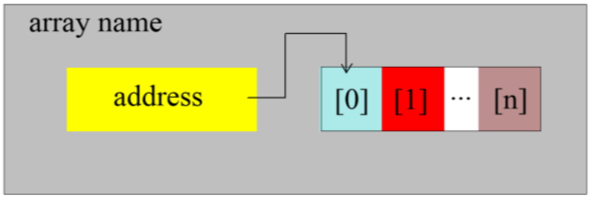
```
pointerQAZ = &myShoeSize;
```

it switches my pointer to `myShoeSize`.



So far, I have only been pointing to a variable. We can also point to arrays.

A Pointer to an Array

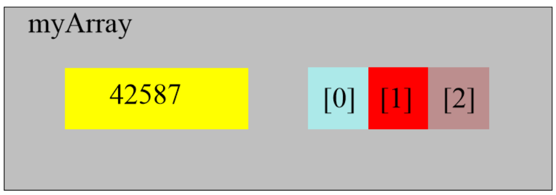


Here is how a one-dimensional array exists in memory. It has an array name, an address, and one or more values.

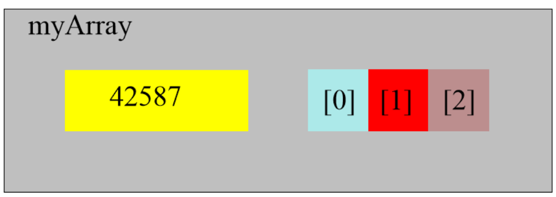
That address is associated with the first element of the array, [0].

I type

```
byte myArray[3];
```



which sets the elements of `myArray[]` to bytes. Say the compiler puts this array at the address 42587. This means that the first element of the array is at this address.



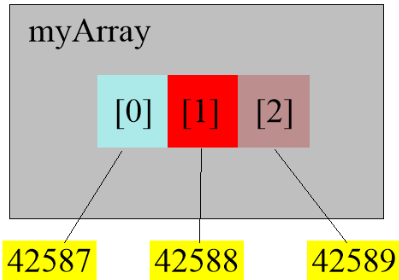
Thinking back to pointers to a single variable, I extract the address by using `&`. Arrays are different. The array's name ***is*** the address:

```
myArray
```

When I write

```
myArray[1]
```

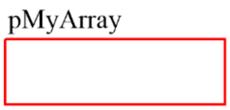
the square bracket tells the compiler it is dealing with an array. It starts at the address `myArray` and moves to the second element.



Since `myArray` is defined as containing bytes[4], the first element is at address 42587 + 0, and the second element is at address 42587 + 1 = 42588.

I write

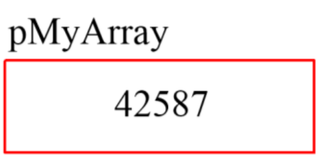
```
byte *pMyArray = 0;
```



to set up a pointer called `pMyArray` that points to a `byte.`

Then I can write

```
pMyArray = myArray;
```

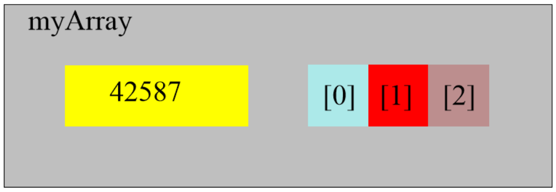


This says I want `pMyArray` to point to `myArray.` Remember that we are talking about an array, so its name ***is*** its address. Had this been a variable, we would have needed to put `&` in front of the name.

Alternately, I could have combined these lines and written

```
byte *pMyArray = myArray;
```

[4] The index is pointing to elements of the array, not the number of bytes. So if the array was, say, integer, we would move two bytes at a time when using the Arduino IDE. Other compilers can use a different number of bytes to hold an integer.



To access the second element of the array with my pointer, I write

`pMyArray[1]`

The brackets tell the compiler it is dealing with an array. It uses the pointer inside of `pMyArray` as the address. `[1]` tells the compiler we want the second element.

Recall that with variables, I put a * in front of the pointer's name to tell the compiler I want to access the address held inside the pointer. I don't do that with an array.

To recap:

I started with `myArray[3]`. Its address is simply `myArray`.

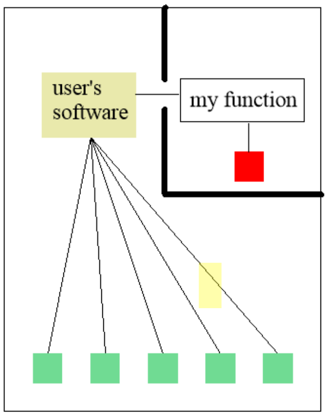
I defined a pointer and set it to the array's address with

```
*pMyArray = myArray;
```

I can then access elements of the array using `pMyArray[`***i***`]`, where ***i*** is the array's index.

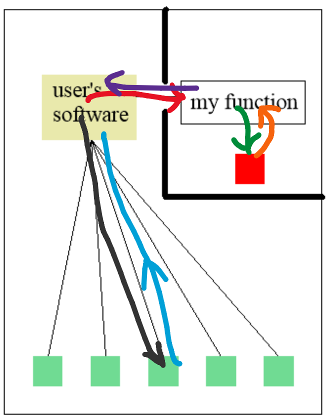
A Pointers and Functions

I will start with an example with the user's software and a function needing to access the same data.



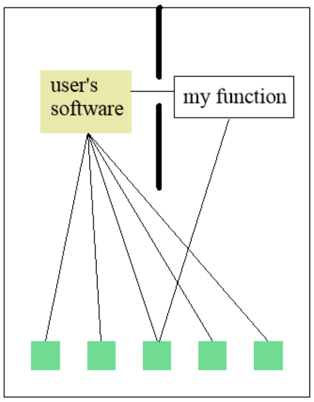
The user's software manipulates five variables (those green boxes).

I have "my function." It is passed a value, modifies it, and returns a value. When my function starts to execute, the variable it uses (that red box) is defined by the operating system as a memory location. When my function terminates, this memory is released, and its data is lost.



The user's software reads the data from one of its variables (blue line) and passes it to my function (red line). My function stores the value to its memory (green line) as it processes it. Then my function reads this value (orange line) and passes it back to the user's software (purple line) which overwrites the variable (black line).

Now consider doing this same task using our shiny new toy. I modified my function to use a pointer.



The user's software calls my function and passes it a pointer to the data. My function now has direct access to the user-defined variable, so just as the user's software reads and writes this address, so does the function.

Next, we will look at some code.

At the risk of insulting your intelligence, consider this function:

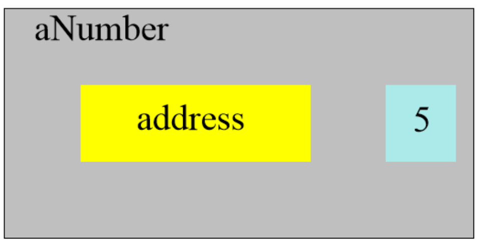
```
byte addOne(byte qInput){
    return (qInput + 1);
}
```

See that **`byte`** in front of addOne? It says that it will return a byte.

I pass this function a byte that will be internally called `qInput`. It adds 1 to that number and returns the result.

I can define the variable `aNumber`:

```
byte aNumber = 5;
```



The compiler selects some address and puts the number `5` in it. The compiler knows this address, but I don't need to know it.

using `addOne()`, I write

```
aNumber = addOne(aNumber);
```

which causes `aNumber` to change from 5 to 6.

Consider what is going on here. The compiler defines a location in memory that corresponds to what I call `aNumber`. It is filled with the number `5`. When I call `addOne()`, I pass this function the ***contents*** of `aNumber`. It returns with a number that I put back into this same memory location. After `addOne()` finishes running, the variable `qInput` vanishes.

Next, consider doing this same operation using a pointer:

```
void addOneWithPointer(byte *pAnumber){
     *pAnumber = *pAnumber + 1;
}
```

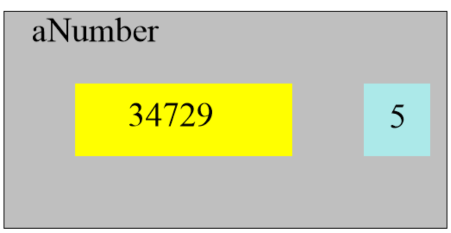
`byte *pAnumber` says I have defined a pointer to a byte. This byte will be our data.

We want the data stored at the address stored in our pointer, so put `*` in front of the pointer's name: `*pAnumber`. This enables us to read the data, increment it by 1, and write it back to the same address.

Do you see **`void`**? It says that the function returns nothing. Yeah! I directly increment a number using `addOneWithPointer()`.

Let's use our new function.

I again start by defining aNumber




```
byte aNumber = 5;
```

This time, I do care about its address. Say the compiler put it at 34729.

`addOneWithPointer()` expects to receive the address of a variable that will be incremented by 1. Recall that I extract the address of `aNumber` by putting `&` in front of it. `&aNumber` equals `34729`.

I call the function

```
addOneWithPointer(&aNumber);
```

`&aNumber` is the address of the data. The function's definition says to expect `*pAnumber`. This is equivalent to saying

```
byte *pAnumber = &aNumber
```

which tells the compiler to extract the address of `aNumber` and pass it to the function. The function takes this address and assigns it to the pointer `pAnumber`.

In other words, `*pAnumber` within the function `addOneWithPointer()` is the same as the variable `aNumber` in the calling software.

In the above example, I wrote

```
addOneWithPointer(&aNumber);
```

For clarity, I chose to use `&aNumber` to be similar to the function's internal name of `*pAnumber`. I could have just as easily written

```
byte myDog = 123;
addOneWithPointer(&myDog);
```

`myDog` will then change from 123 to 124.

`&myDog` is the address of `myDog`. Whatever variable is passed as the argument is the one modified by the function.

```
void addOneWithPointer(byte *pAnumber){
     *pAnumber = *pAnumber + 1;
}
```

Passing A Pointer to an Array into a Function

Recall that we can pass an array's address by using its name, for example, `myArray,` and access that array using this address by writing its name followed by an index inside square brackets - `myArray[index]`.

Consider this function

```
void changeAnElement(byte *pInputArray, byte index, byte value){
    pInputArray[index] = value;
}
```

A lot is going on with these arguments:

`byte *pInputArray` defines a pointer to a byte. The value passed via this argument will be placed into this pointer.

`byte index` and `byte value` say to expect two numbers, each of the type byte.

The function has `void` at the beginning, which means nothing is returned. We don't need to because we have direct access to the array.

The one line of code,

```
pInputArray[index] = value;
```

looks "normal", but it is not. `pInputArray` is a pointer. Since it is pointing to an array, we add square brackets and a number to index into the array.

We are writing `value` to an array that was not defined within `changeAnElement()`. The calling software defined it.

```
void changeAnElement(byte *pInputArray, byte index, byte value){
     pInputArray[Index] = value;
}
```

To use this function, I need to first define the variables it will be passed:

```
byte aBC[3] = {5,6,7};
byte ind = 1;
byte val = 9;
```

Then I can call the function and pass these variables as its arguments.

```
changeAnElement(aBC, ind, val);
```

Inside `changeAnElement()` I have the equivalent of

```
byte *pInputArray = aBC
```

This is saying that I want to define a pointer to a byte and fill this pointer with the address of an array called `aBC`.

My pointer, `pInputArray`, is now used with an index and is identical to `aBC[index]`

```
pInputArray[index] = value;
```

which is

```
pInputArray[1] = 9;
```

Since `pInputArray[1]` points to `aBC[1]`, the result is that aBC[1] changes from 6 to 9.

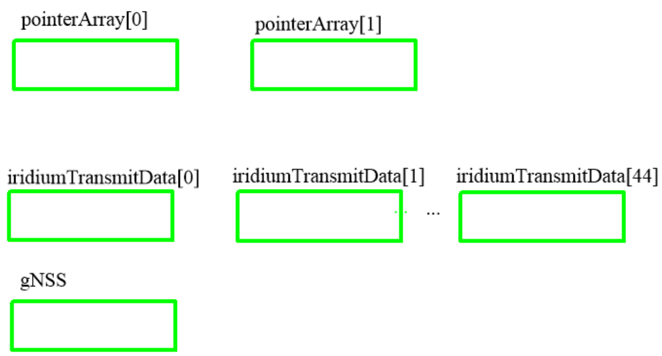
`changeAnElement()` lets me modify ***any*** element of ***any*** array of bytes.

A Pointer to an Array Filled with Pointers

I hope you are comfortable with the idea of having a pointer to an array and then passing this pointer as an argument into a function. Inside the function, this information permits the code to access all elements of the array.

So far, I have filled the array with numbers for use in calculations. I could fill the array with numbers for use as pointers. Huh?

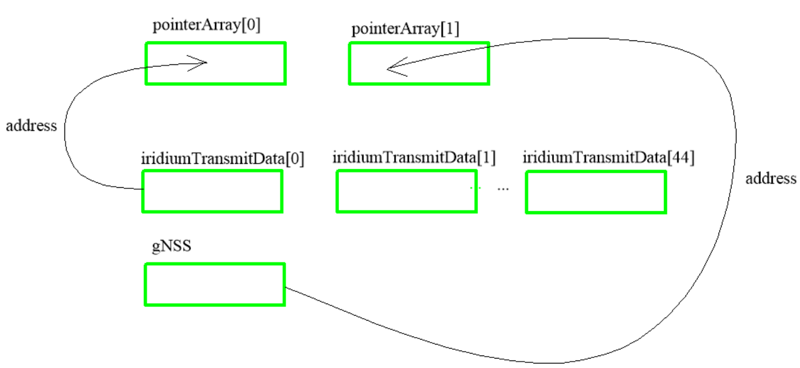
Consider



I have defined the two element integer array, `pointerArray[]`,

a 45 element byte array, `iridiumTransmitData[]`, and

a Boolean variable, `gNSS`.



Next, I will extract the address of `iridiumTransmitData[]`, and put it into `pointerArray[0]`.

The address of my Boolean will go into `pointerArray[1]`.

Notice that `pointerArray[]` is holding numbers.

I can then extract the address of `pointerArray`[] and pass it around. Whoever gets this address can access its two elements. By assigning each element's value to a pointer, they are able to access `iridiumTransmitData[]` and `gNSS`.

Next, we will go through this again, looking at the code.

```
integer pointerArray[2];

byte iridiumTransmitData[45];
boolean gNSS = false;

pointerArray[0] = iridiumTransmitData;
pointerArray[1] = &gNSS;
```

I have defined an integer array, `pointerArray[]`, with two elements. Then I defined a byte array, `iridiumTransmitData[];`, with 45 elements, and a Boolean, `gNSS`, that I set to false.

Next, I put the address of `iridiumTransmitData[]` into `pointerArray[0]`. Since `iridiumTransmitData` is an array, I get its address by using its name without an index. I put the address of my Boolean into `pointerArray[1]`. In this case, I am dealing with a variable, so use `&` to extract the address.

When I pass the address of `pointerArray[]` to a function, I am providing access to `iridiumTransmitData[]` and `gNSS`. I just need to unpack `pointerArray[]`.

Let's give this a try.

In preparation for passing the array of pointers, I have

```
int pointerArray[2];

byte iridiumTransmitData[45];
iridiumTransmitData[11] = 111;

boolean gNSS = false;

pointerArray[0] = iridiumTransmitData;
pointerArray[1] = &gNSS;
```

`pointerArray[2]` is defined as holding integers because my addresses are integers.

I define `iridiumTransmitData[]` as an array of bytes and set element 11 to 111. The Boolean variable, gNSS, is set to false.

Next, I fill up `pointerArray[]` with my addresses. Element `0` holds the address of my array, and element `1` holds the address of my Boolean.

I define my function:

```
void iridiumAndFlag(int *aPointerArray){

   byte *myIridiumTransmitData = aPointerArray[0];
   boolean *myGNSS = aPointerArray[1];

   Serial.print("is this 111? ");
   Serial.println(myIridiumTransmitData[11]);
   Serial.print("is this 0? ");
   Serial.println(*myGNSS);

}
```

This function expects to be passed an integer pointer that will be called `aPointerArray` inside the function.

I define a pointer to a byte, `myIridiumTransmitData`, and fill it with `aPointerArray[0]`.

Next, I define a pointer to a Boolean, `myGNSS`, and fill it with `aPointerArray[1]`.

To see if this all works, I want to see if I can access the 11th element of the array and the value of my Boolean.

For this test, I print `"is this 111?"` and then, hopefully, print the value of `myIridiumTransmitData[11],` which should be 111. Similarly, I print `"is this 0?"` and then the value of my Boolean, which was set to false so has the numerical value of 0. Since we are dealing with a variable and not an array, I need to put `*` in front of the pointer to say I want the data stored at the address defined by my pointer, `myGNSS`.

When I run this code, I get the unremarkable result

```
is this 111? 111
is this 0? 0
```

I have demonstrated that I can pass a function an array filled with pointers, and the function can take this apart and access the variables accessed by these pointers.

I know my example is trivial. But I could define an array filled with a boatload of pointers. Each of these pointers can point to an array of a ton of numbers. By passing a function a single pointer that contains the address of my array, I am passing it an almost unlimited number of variables. All of a sudden, this isn't so trivial.

Acknowledgment

Thanks to Justisse Mulligan for helping me to understand some critical errors in my thinking. Thanks to "BuffaloJohn" for key insights and finding a few more errors in my thinking.

I welcome your comments and questions.

If you want me to contact you each time I publish an article, email me with "Subscribe" in the subject line. In the body of the email, please tell me if you are interested in metalworking, software plus electronics, kayaking, and/or the Lectric XP eBike so I can put you on the right distribution list.

If you are on a list and have had enough, email me "Unsubscribe" in the subject line. No hard feelings.

Rick Sparber
Rgsparber.ha@gmail.com
Rick.Sparber.org