Arduino C++ Pointers with Libraries, Version 1.1

By R. G. Sparber

Scope

There are many excellent articles on pointers and on libraries. There are also equally helpful articles on arrays. I was unable to find any writings that pulled these topics together into one story. They were all written by people that know this material very well. This is not entirely helpful because they can't see their examples through my eyes, as a confused student. I have documented my journey for my fellow travelers. Maybe I can make your journey a little smoother.

Background

I am mostly self-taught when it comes to programming in C++ using the Arduino IDE. My limited knowledge was good enough to write many useful programs, including my latest at 20492 bytes. When I was asked to move this code into a library, I didn't want to start by first completing a formal education in this subject. I just wanted to learn enough to get it done. My task was completed with plenty of reading and also a lot of trial and error.

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. I invite those with more education to point out these shortcomings, so I may present better examples.

References

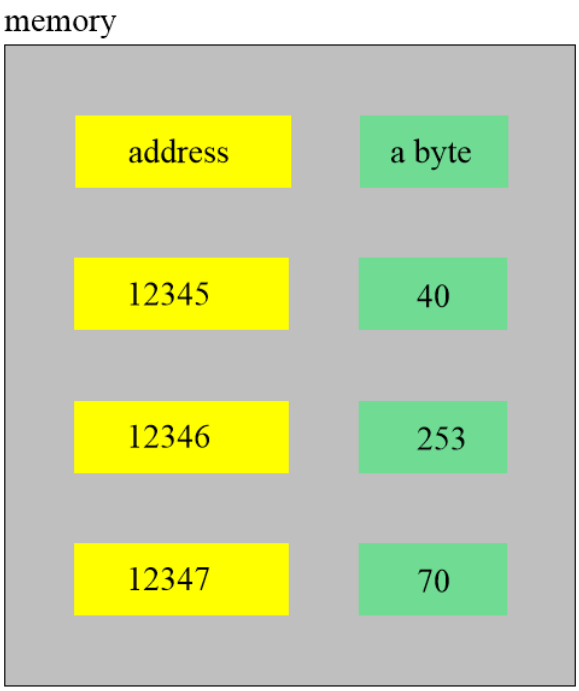
[Pointers, Arrays, and Functions in Arduino C (engineerworkshop.com)]
[C++ Passing Arrays to Functions - Tutorialspoint]
[C++ Pointer to an Array - Tutorialspoint]
[Classes (I) - C++ Tutorials (cplusplus.com)]
[How to Write Libraries for Arduino - Alan Zucconi]
[Arduino - LibraryTutorial]

I also relied on Dave Kellogg to answer some poorly formed questions resulting from my confusion.

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 has no knowledge of other data types. If I want to store a variable that uses four bytes, that must be handled in the software. Memory is a “one-trick pony,” yet it is hard to find software that doesn’t depend on it.

A Variable

When I took my first class in programming, back in 1969, we had this structure. A variable had a ***name***, an ***address***, and a ***value***. The name was used in the program and was for our benefit. The compiler replaced the name with an address. At that address was a value. By using the name, we could read and write the value.

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

I have a variable called MyHatSize. The compiler assigned it to the address 28123. It has a value of 7.5. Since I defined this variable as a float, it takes up four bytes.

Each generation of software engineers wants to have its own jargon. It is human nature.

Instead of talking about an address, we have an ***lvalue***. The "***l***" stands for location. The variable's value is now called the ***rvalue***. I don't know what "***r***" stands for.

I found this definition:

> An *lvalue* (*locator value*) represents an object that occupies some identifiable location in memory (i.e. has an address).
>
> *rvalues* are defined by exclusion, by saying that every expression is either an *lvalue* or an *rvalue*. Therefore, from the above definition of *lvalue*, an *rvalue* is an expression that *does not* represent an object occupying some identifiable location in memory.

So maybe an rvalue is a superset of what I learned to be a value. In the following discussion, I will only be dealing with cases where an rvalue is a value.

A Pointer To A Variable

I programmed as a hobbyist for many decades, dealing only with variable names. But recently, I needed to venture into the strange world of pointers.

Instead of using names, I could write code that knows about the lvalue 28123. This ability is essential when I have a piece of code that was previously compiled. Any names it used were tossed aside by the compiler. But if I give it an lvalue, it can access the rvalue put at that lvalue by another program. I'll talk about previously compiled code later when I get into Libraries.

I can write

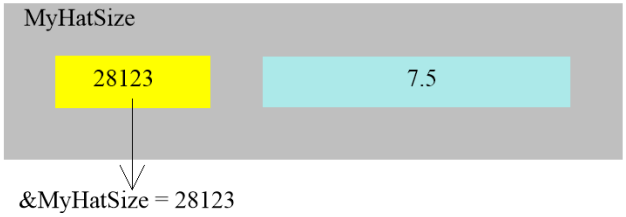
```
Float MyHatSize;
```

and the compiler will define a group of bytes to hold the value. This group is accessed by knowing its name or its lvalue. As long as the calling program knows that it is a float, all is well.

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

```
MyHatSize = 7.5;
```

OK, so how can I know this variable's lvalue? This is done by placing "`&`" in front of the variable's name.



`&` "extracts" the lname from `MyHatSize`. This lvalue tells me where to look for the contents of MyHatSize.

Notice that `&` is an operator that works on `MyHatSize.` It does not define a new name.

I could assign this lname to another variable.

```
int LittleHead;
LittleHead = &MyHatSize;
```

The lname is an integer[2] , so that is what data type I must use. Notice that there is no harm in first defining `LittleHead` and then using it to hold the lname of `MyHatSize`. `LittleHead` is just holding a number.

To recap: I previously defined a variable called `MyHatSize`. When I put `&` in front of this name, it extracts the variable's location, or lname. I defined `LittleHead` as another variable that holds an integer. With

```
LittleHead = &MyHatSize
```

I am taking the lname of `MyHatSize` and storing it in `LittleHead`.

`LittleHead` can be printed out and even modified, but it can't be used to access `MyHatSize`. To do that, I must tell the compiler that I want to use this lname to retrieve the value stored at this location.

To perform this magic, I must define a new variable that has "`*`" as its first character. This `*` tells the compiler to treat this new variable as an alternate name for an existing variable.

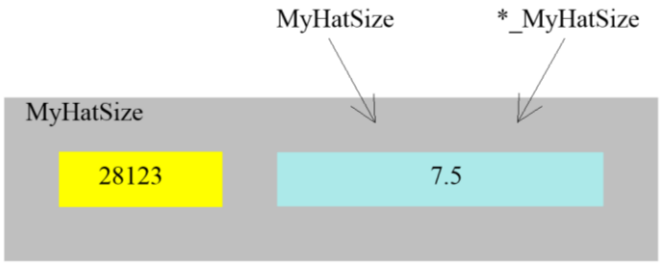
[2] This was my assumption back on page 2 but is hardware and compiler dependent.

Since this new variable and the original name both point to the same rvalue, they must have the same data type. Recall that `MyHatSize` was defined as a float, so I must use float here too

```
float *_MyHatSize = &MyHatSize;
```

`&MyHatSize` contains the location of `MyHatSize`, and it ***looks*** like I have assigned it to `*_MyHatSize`. That is not the case. When the compiler sees that *, it knows to take the number represented by `&MyHatSize` and use it as a pointer to the desired data. `*_MyHatSize` is, therefore, identical to `MyHatSize`.

Everything that I can do with `MyHatSize` can also now be done with `*_MyHatSize.`



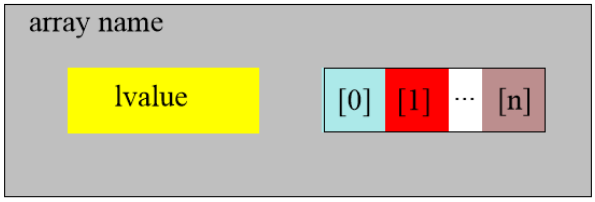
If these two methods of accessing the same data are in one program, this all must seem rather ***point***less. Why not just use MyHatSize? I need to lay a bit more foundation before I can answer.

That "_" is used to remind us that the pointer's name is linked to the name of the variable. In the general case, `*_MyHatSize` can point to any byte, so the names won't be similar.

This notation is common when we talk about libraries where the names will be similar. For example, the calling program might have the name `pin`, and we define the variable `_pin` inside the library to point to `pin`.

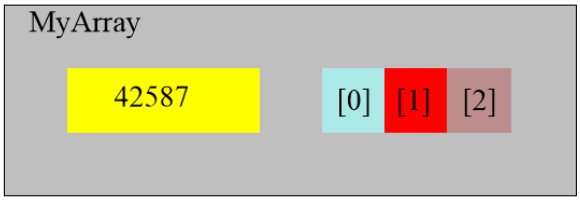
A Pointer to an Array

I will present this information in a way that didn't confuse me. Then I will show you how it is commonly used.



Here is how a one-dimensional array is laid out. It has an array name, an lvalue, and one or more rvalues.

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




```
byte MyArray[3];
```

I have defined MyArray[3]. The compiler put MyArray[0] at the lvalue of 42587. The elements of MyArray[] are bytes. This means the second element, MyArray[1], has an lvalue of 42587 + 1 = 42588. The third element, MyArray[2], is at 42587 + 2 = 42589.

Recall that the lvalue for the array points to the first element.

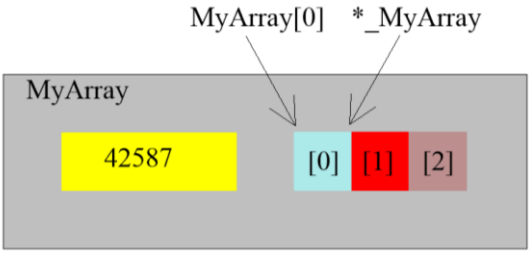
Thinking back to pointers to a (single) variable, I can extract its lvalue by using `&`. I found it comforting to find that

```
&MyArray[0]
```

does give me the array's lvalue. I can then write

```
byte *_MyArray = &MyArray[0];
```

`MyArray[]` was defined as both byte and an array. `*_MyArray` is only defined as a byte. This is because it is associated with only the first element of `MyArray[]`. If `MyArray[]` had been defined as an array of unsigned longs, `*_MyArray` would also have to be defined as an unsigned long. Note that I have fully defined `*_MyArray` in a single line.



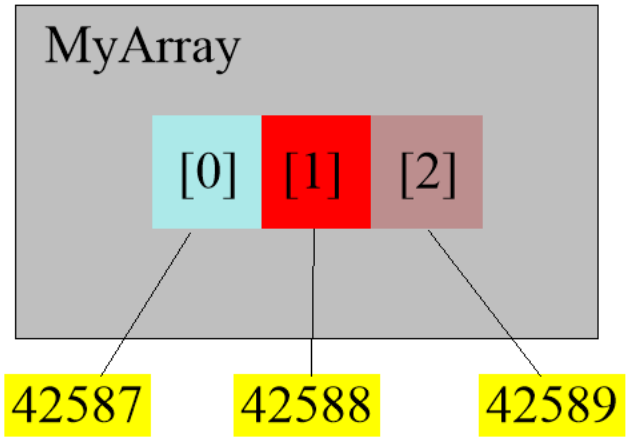
`*_MyArray` and `MyArray[0]` access the same byte. OK, what about the rest of the array?

When we talked about pointers to single variables, "`&`" in front of the name told the compiler to extract the lvalue. `&MyArray[0]` extracts the lvalue of the first element of the array.

I can address other elements of `MyArray[]` by adding to `&MyArray[0]`.

I will define a new variable

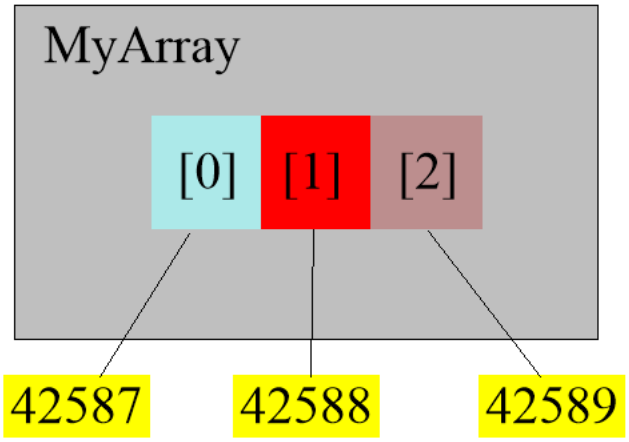
```
byte SecondElement;
```



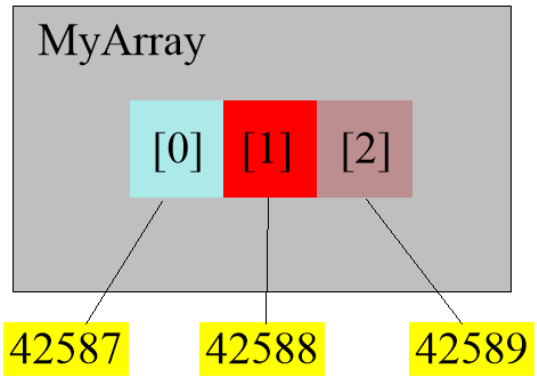
`SecondElement` will hold an element of my array, so it must be of the same type, `byte`. My goal is to have the equivalent of

```
SecondElement = MyArray[1]
```

but using pointers. In my example, the first element of the array has the lvalue 42587 which will be returned by entering `&MyArray[0]`.



Since MyArray[] is made of bytes, the next element of this array has an lvalue equal to the first element's lvalue + 1. This gets me to 42588. The same goes for the third element.



I can point to the second element of the array by adding 1 to the first element's lvalue:

```
&MyArray[0] + 1
```

This will give me 42588, the lvalue of the second element. Recall that I can access the value at this lvalue, its rvalue, by creating a new variable that starts with `*`

```
byte *_SecondElement = &MyArray[0] + 1
```

To recap: I started with `MyArray[3]`. By placing `&` in front of the first element's name, I extracted its lname. To increment to the second element of the array, I add 1. This gave me the lname of the second element. Placing * in front of a newly defined variable yields the content of this location. Note that `*_SecondElement` only accesses one element of `MyArray[]`.

Here is a little curveball: The digit 1 is not necessarily adding 1 to the array's lvalue. It is moving us to the second element. For example, by defining the array as being full of integers, This code still gives us the second element. The lvalue of the second element is not one more than that of the first element because integers take two bytes each.

Notice that you, the programmer, are responsible for staying within the array. There is no range check. So if you define MyArray[3] and then access the fourth element, you will get ***a*** result. It just won't have anything to do with this array. Silent failures like this can make you old and gray before your time.

`&MyArray[0] + 1` does let us access all elements of the array and is consistent with the conventions that apply to single variable. But it is cumbersome since I must define a different variable for each element.

Although I found it *mind-bending*, because this seems to nullify all that we just learned, there is a shortcut: you can simply use

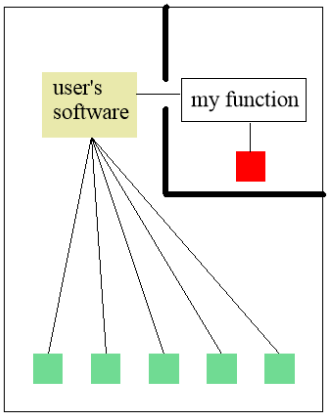
```
MyArray[Index]
```

where `Index` is a number within the range of `MyArray[3]: 0, 1, or 2.`

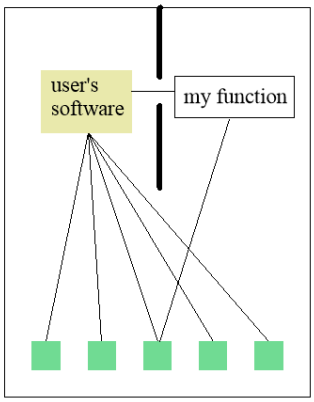
Furthermore, rather than having `&MyArray[0]` to identify its lvalue, you can simply write `MyArray.`

However, do not try going without `&` and `*` with single variables. You will get compiler errors.

A Pointer To A Variable Inside A Function



I can write a function that is self-contained[3]. If it needs a variable (that red box), it can define it. Code outside of this function knows nothing of this variable. Users of this function can pass it values, and the function can pass a value back.



If my function is given a pointer, it will have direct access to the user-defined variables. No values need to be passed between the user's software and my function.

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

```
byte AddOne(byte _Input){
     return (_Input + 1);
}
```

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

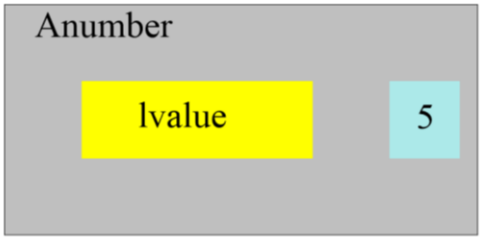
I pass this function a byte that will be called `_Input`. It adds 1 to that number and returns the result.

[3] Yes, I can write functions that are mixed into the user's code and have full access to all global variables. That is not where I'm going with this.

```
byte AddOne(byte _Input){
    return (_Input + 1);
}
```

I can define the variable `Anumber`:

```
byte Anumber = 5;
```



I defined some memory location and put the number 5 in it. Only the compiler knows its location (lvalue). I don't need to know it.

using `AddOne()`

```
Anumber = AddOne(Anumber);
```

This will cause `Anumber` to equal 6.

Consider what is going on here. I first define a location in memory and call it `Anumber`. It is filled with the number 5. When I call `AddOne()`, I pass it the ***contents*** of `Anumber`. It returns with a number that is put back into this same memory location. After `AddOne()` runs, all data associated with it is erased.

Next, consider doing this same operation using a pointer:

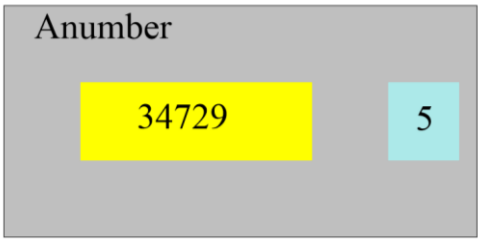
```
void AddOneWithPointer(byte *_Anumber){
    *_Anumber = *_Anumber + 1;
}
```

`byte *_Anumber` says to expect a pointer and use it to give access to the data at this location (the rvalue). We treat `*_Anumber` as a variable that we add 1 to and then put the result back into the same location.

Do you see **`void`**? It says that nothing is returned.

I directly increment a number using `AddOneWithPointer()`.

Let's use it. I again start by defining Anumber




```
byte Anumber = 5;
```

This time, I do care about its lvalue. Say it is 34729.

Then I call the function

```
AddOneWithPointer(&Anumber);
```

We again define a location in memory, `Anumber`, and fill it with the number 5. But when we call `AddOneWithPointer()`, it has the argument `&Anumber` which is its address (the lvalue): 34729.

`Anumber` is some place in memory. Any time I use this name, I am accessing its data.

To use this function, I have

```
AddOneWithPointer(&Anumber);
```

Looking back on this function's definition, we see

```
void AddOneWithPointer(byte *_Anumber){
     *_Anumber = *_Anumber + 1;
}
```

I supplied the function with `&Anumber`, which is a pointer to the data. The function's definition says to expect `*_Anumber`. This is not a mismatch but rather a variable assignment. It means

```
byte *_Anumber = &Anumber
```

```
byte *_Anumber = &Anumber
```

It says to extract the lvalue from Anumber and pass it to the function. The function takes this lvalue and uses it to retrieve its rvalue, which is assigned to `*_Anumber`.

```
void AddOneWithPointer(byte *_Anumber){
     *_Anumber = *_Anumber + 1;
}
```

Note that `*_Anumber` is fully defined in one line although that line is split between the calling of the function and the definition of the function.

For this example, I wrote

```
AddOneWithPointer(&Anumber);
```

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

```
AddOneWithPointer(&Dog);
```

Then `*_Anumber` would point to the memory location holding `Dog`. Whatever variable is passed as the argument is the one modified by the function.

A Pointer To an Array and Functions

Recall, at the end of my discussion about arrays, that I said we can pass an array's lvalue by simply writing `MyArray` and access that array using this lvalue by simply writing `MyArray[Index]`.

Consider this function

```
void ChangeAnElement(byte *_InputArray, byte Index, byte
Value){
    _InputArray[Index] = Value;
}
```

A lot is going on with these arguments:

`byte *_InputArray` says this parameter will be an lvalue of a variable in byte format, and we want it to access its rvalue.

`byte Index` and `byte Value` say to expect two numbers, each in byte format.

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 looks normal, but it is not. `_InputArray[ ]` is using the lvalue that was passed and is accessing the rvalues of the array. In other words, we are equating Value to an array that was not defined by the function's environment. The calling software defined it.

```
void ChangeAnElement(byte *_InputArray, byte Index, byte
Value){
    _InputArray[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 *_InputArray[0] = &ABC[0]
```

With the mind-bending shortcut presented on page 11, I can write

```
_InputArray[Index] = Value;
```

which is

```
_InputArray[1] = 9;
```

Since `_InputArray[1]` points to `ABC[1]`, the result will be that ABC[1] will be changed from 6 to 9.

At the risk of messing with your mind, think about adding a second array

```
void ChangeAnElement(byte *_InputArray, byte Index, byte Value){
      _InputArray[Index] = Value;
      byte InternalArray[3];
      InternalArray[Index] = Value;
}
```

When this function runs, the array passed via the first argument will be updated and survive the function's completion. `InternalArray[ ]` will vanish when the function completes.

Pointers and a Library

The place where pointers really shine is with libraries. A library is a collection of code that is packaged up and ready to be used by others. This code can be compiled, which makes it difficult for others to steal or corrupt. Even left as uncompiled code, it is a handy way to isolate code from the user.

I refer you to this excellent lesson which helped me to understand how to set up a library. I have used it as a starting point to add an example of a function with pointers.

As I studied the code presented in this lesson, questions came to mind that were not addressed. Looking at another lesson dealing with this subject, I could see similarities that imply that these conventions are rules and not the author's style. By testing various changes to the files, I was able to confirm these rules.

This is an inefficient and risky way to learn because I may get the reason for a form wrong even though it does compile and run without error. My strategy is to be as clear as possible, so those with more understanding can easily see my misconceptions. Subsequent versions of this article will reflect those corrections. My speculations will be highlighted.

For the following discussion, I assume you have read this excellent lesson and understand it. That frees me to build on that knowledge.

I have removed as much code as possible from their example to focus on pointers and a library. I have not added any comments since I will be explaining key elements in the text. My focus includes things that confused me while I was learning.

As you will learn from the tutorial, all of the following files go into one folder placed in the library's folder.

I have arbitrarily selected the name "Box" as the folder's name and for most of the files within it. You will see that this common name links these files.

I will first present the various files and then explain how they fit together.

The Header File

Box.h contains

```
#ifndef Box_h
#define Box_h

#include "Arduino.h"

class Box1
{
  public:
    Box1(int pin);
    void dot();
    void dash();
    void ChangeAnElement(byte *InputArray, byte Index, byte Value);

  private:
    int _pin;
};

#endif
```

Here is my understanding of what each line means.

```
#ifndef Box_h
#define Box_h
…
#endif
```

`Box_h` is an arbitrary name, but this is a standard, and useful, convention. Box_h is a reminder that we are dealing with Box.h.

```
#ifndef Box_h
```

is testing to see if `Box_h` has been previously defined. If so, we do not want to define it again and possibly introduce bugs.

If `Box_h` was not previously defined, we define it with the code that ends with `#endif`

The compiler must be told that we will be using the elements of Arduino based code

```
#include "Arduino.h"
```

Next, we have

```
class Box1
{
…
}
```

which defines a group of functions and constants. There can be any number of "classes" within the same .h file.

```
class Box1
{
…
}

class Cat
{
…
}

class Dog
{
…
}
```

You will later see that the user must specify both a class and a function at each use.

I have experimentally confirmed that I get compiler errors if each class name does not start with a capital. You will later see that this name is sometimes changed to lower case. If it starts out lower case, I do not avoid errors by making it upper case later. The compiler is looking for the lower case version of the class, not the same name with the first letter's case flipped.

```
public:
```

As the name implies, items in this category are accessible by the user.

```
Box1(int pin);
```

Notice that this function has the same name as the class: `Box1`. It is called a "***constructor***" and is used to set up the environment used by other members of its class. This capability is the same as setup() in the user's program. However, the environment set up by Box1() is not the same as what the user enjoys.

Note that there is no option for returning a value. You do not see `void Box1(int pin)`. It makes sense that this function can contain arguments because the class may need information from the user to set up their environment.

I was unable to find a way to have a constructor with no arguments that would compile without errors using the Arduino IDE.

In this example, the user will pass an integer. Internal to the function, this integer will be called "pin."

I empirically found that the constructor must not be inside of any function. For example, if I call the constructor within setup(), the compiler will complain that it doesn't recognizer any of the members of its class.

Next, we have a list of functions accessible to the user. They can be inside any user defined function. Each function starts with the option for what is returned. In this case, none of the functions return any value.

```
void dot();
void dash();
void ChangeAnElement(byte *InputArray, byte Index, byte Value);
```

Notice that I list each function as if I was going to define it, yet that is not done here. Look to `Box.cpp` for that task.

I hope that `ChangeAnElement()` is familiar to you by now. If not, refer back to page 16.

```
void ChangeAnElement(byte *InputArray, byte Index, byte Value);
```

The user populates `ChangeAnElement()` arguments with values that are passed to the function. These particular values are only used by this function.

Recall that we have the line

```
Box1(int pin);
```

You will later see that the functions `dot()` and `dash()` need to know the value for pin. This value is part of these function's environment.

Next in the file we have

```
private:
  int _pin;
```

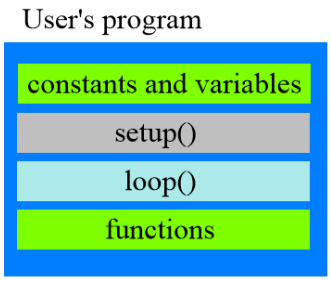
After "`private:`", we have a list of variables. OK, just one item on this list. This is how the value for `pin` is shared with `dot()` and `dash()`.

As I mentioned on page 7, private variables can have any name, but there is a convention that makes a lot of sense. A "_" is added in front of the public variable's names to indicate that it is internal, or "private." The compiler defines an integer named `_pin,` but, like the above functions, it doesn't know what to do yet.

We define the end of the `Box1` class with a "`};`" and close out the `#ifndef Box_h` with `#endif.`

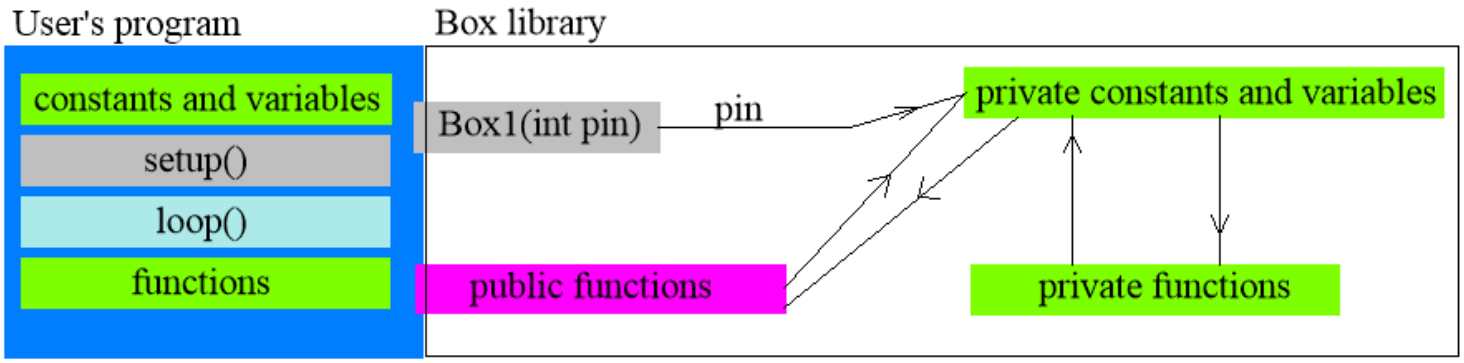
```
public:
  Box1(int pin);
  void dot();
  void dash();
  void ChangeAnElement(byte *InputArray, byte Index, byte Value);

private:
  int _pin;
```



One thing that confused me was how the library's structure compares to a standard Arduino program which has `setup()`, `loop()`, and the rest of the space is for global variables and global functions. I have always arranged the text with constants and variables on top, then `setup()` followed by `loop()`. Space for functions is at the bottom.

Inside a global function, I can define variables, and they would be local to that function.



The small overlap of some boxes between the user's program and the Box library signifies that the user has access to them.

If `Box1(int pin)` is called by the user's program, it sets up the environment for the functions in its class. This environment includes knowing the value of `pin` which is equated to a private variable. The other private functions can then use it.

Public constants, variables, and functions are accessible to the user. A public function can equate any of these constants and variables to private variables so they are also accessible to private functions. Not shown in this diagram is that private functions can directly access the user's constants and variables if given their lvalues.

The C++ File

The header file Box.h is linked to its code by having the same name but with the extension `.cpp`.

`Box.cpp` contains

```
#include "Arduino.h"
#include "Box.h"

Box1::Box1(int pin)
{
  pinMode(pin, OUTPUT);
   _pin = pin;
}

void Box1::dot()
{
  digitalWrite(_pin, HIGH);
  delay(250);
  digitalWrite(_pin, LOW);
  delay(250);
}

void Box1::dash()
{
  digitalWrite(_pin, HIGH);
  delay(1000);
  digitalWrite(_pin, LOW);
  delay(250);
}

void Box1::ChangeAnElement(byte *InputArray, byte Index, byte Value)
{
      InputArray[Index] = Value;
}
```

I will again talk about each line.

```
#include "Arduino.h"
```

We must again tell the compiler we will be using the elements of Arduino based code. This might be saying that Box.cpp can be compiled separately from Box.h.

```
#include "Box.h"
```

The compiler must be told to include our `Box.h`. I am guessing that we must do this because it leaves open the option of including alternate .h files. The quotes mean to look for Box.h in the same directory as Box.cpp.

```
Box1::Box1(int pin)
{
…
}
```

Recall that the function `Box1(int pin)` is the constructor – it contains all items that will set up the class's environment. The format shown here must be followed. The first name, `Box1` is the class, as introduced on page 21. Then we have `::` which is followed by the name of this special function. These names must match.

Anything between the `{ }` will end up in the class's environment.

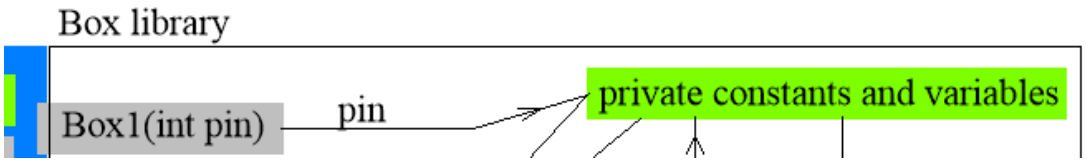
```
Box1::Box1(int pin)
{
  pinMode(pin, OUTPUT);
  _pin = pin;
}
```

As is common within setup(), we define a General Purpose Input Output (GPIO) pin as an `OUTPUT`. Which pin is defined came from the `Box1()`'s argument so is set by the user. If `pin` was only used within the constructor, we would just use its name. You will later see that we have functions that will be placed in loop() that also need to know the value of `pin`.

This next line

```
_pin = pin;
```

deals with a private variable, `_pin`, being set equal to a public variable, `pin`.



Remember `_pin`? It was defined as a private variable in the Box.h file.

The rest of the .cpp file contains functions that are available to the user.

```
void Box1::dot()
{
  digitalWrite(_pin, HIGH);
  delay(250);
  digitalWrite(_pin, LOW);
  delay(250);
}
```

Each function starts with the optional return value format. In this case, nothing is returned so we have `void`. This is followed by the name of the class (as explained on page 21), `Box1` and then ::

The function's name, `dot()`, is next. Then we have its content starting with

```
digitalWrite(_pin, HIGH);
```

We are writing the output pin, `_pin`, to HIGH, which means a 1. Ah, that is how `_pin` is used.

If `_pin` was not defined as private, we would either have to hard code a number

```
digitalWrite(5, HIGH);
```

or have an argument passed

```
void Box1::dot(byte fish)
{
     digitalWrite(fish, HIGH);
     delay(250);
     digitalWrite(fish, LOW);
     delay(250);
}
```

We would also have to change the `.h` file to reflect the fact that `dot() has` an argument

```
public:
    Box1(int pin);
    void dot(byte fish);
    void dash();
```

The rest of the functions that will be available to the user are similarly defined.

Keywords.txt

We have one more file included within this library's folder. It is needed by the compiler to tie it all together. `Keywords.txt` contains

```
Box1   KEYWORD1
dot    KEYWORD2
dash   KEYWORD2
ChangeAnElement     KEYWORD2
```

All classes, as defined in Box.h, are identified with KEYWORD1. All functions available to users are identified by KEYWORD2.

After each name, a single tab must be used before entering `KEYWORD1` or `KEYWORD2`. Do not use spaces.

Each time you open up the Arduino Integrated Development Environment (IDE), and call on this library, it checks the `Keywords.txt` file. So if you make changes to this text file, **don't forget to close the IDE and restart it.**

The User's Program

We are finally ready to try out our new library.

```
#include <Box.h>

Box1 box1(5);

void setup()
{
    Serial.begin(19200);
}

void loop()
{
    byte ABC[3]={3,4,5};
    byte Ind = 1;
    byte Val = 9;
    box1.ChangeAnElement(ABC, Ind, Val);


    Serial.print(F(" is this 9? "));
    Serial.println( ABC[1] );


    if(ABC[Ind] == Val){
        box1.dot();box1.dot();box1.dot();
        box1.dash(); box1.dash(); box1.dash();
        box1.dot(); box1.dot(); box1.dot();
    }

    delay(3000);
}
```

We need to include our new library

```
#include <Box.h>
```

The < > around Box.h means the preprocessor will use a predetermined directory path to find Box.h. This is in contrast to " " which says to look in the same directory as the one containing this command.

Now, our little bit of code is barely worth the trouble of creating a library, but you can imagine having a massive project in there. We get all of that functionality with this one `#include`. Furthermore, we get rid of all of that code by removing this one line.

```
Box1 box1(5);
```

This line confused me for a long time. It constructs the environment for the class which is defined in the library. The value, 5, is part of that set up. `Box1()` is the function. **`B`**`ox1()`? Is that a typo? I see **`b`**`ox1(5)`! Nope, it must be this way unless you enjoy reading compiler errors. Notice that it is not within `setup()` or `loop()`.

```
void setup()
{
    Serial.begin(19200);
 }
```

As is standard in Arduino code, I specify my setup(). Since I will be printing to an external device via the USB cable, I must set up that interface.

Recall that in `Box.cpp` I defined a pin as an output. If the user defined it as an input, I don't know which one would be last on the list, so win out. In any case, some software wouldn't work as expected. Although I can have multiple (software) environments, there is only one piece of hardware. This hardware is essentially global.

```
void loop()
{
    byte ABC[3]={3,4,5};
    byte Ind = 1;
    byte Val = 9;
    box1.ChangeAnElement(ABC, Ind, Val);

    …

}
```

Here is some of my `loop()`. I define the array `ABC[]` and populate its elements. Then I define the variables `Ind` and `Val`. Note that they are all defined as `bytes`, which is what our function `ChangeAnElement()` expects.

We are now prepared to call a library function. To do this, we must identify the class and the function's name.

```
box1.ChangeAnElement(ABC, Ind, Val);
```

Looking back to page 21 at Box.h, we see that the class is called **B**ox1. No, this isn't a typo; we use the name with the first letter lower case. You will get compiler errors if you name a class with the initial character lower case.

This is the big moment! We have loaded our function, `ChangeAnElement()`, which was defined in our library, with parameters and are calling it.

If all goes well, the library's code will reach over to our locally defined array, `ABC[]`, and change `ABC[1]` from `4` to `9`.

The remainder of my user-defined code is

```
Serial.print(F(" is this 9? "));
Serial.println( ABC[1] );


if(ABC[Ind] == Val){
  box1.dot(); box1.dot(); box1.dot();
  box1.dash(); box1.dash(); box1.dash();
  box1.dot(); box1.dot(); box1.dot();
}
  delay(3000);
```

Since I put

```
Serial.begin(19200);
```

In `setup()`, I am able to print out diagnostic messages:

```
Serial.print(F(" is this 9? "));
Serial.println( ABC[1] );
```

On my computer screen, I see

is this 9? 9

Success!

Looking back at the code, we have

```
if(ABC[Ind] == Val){
  box1.dot(); box1.dot(); box1.dot();
  box1.dash(); box1.dash(); box1.dash();
  box1.dot(); box1.dot(); box1.dot();
}
```

It is flashing the LED tied to logical pin 5 to tell us the code worked.

These two responses confirm that my Box.h, Box.cpp, keywords.txt, and the user's code all work together.

To recap: I filled the array ABC[] with {3, 4, 5}, set Index to 1, and set Value to 9. I then called the library function ChangeAnElement with the needed arguments:

ABC - by only passing the name of the array, we are passing the lvalue of the first element.

Ind – the library function will use this number to index through ABC[]

Val – this is the number that will be put into ABC[Ind]

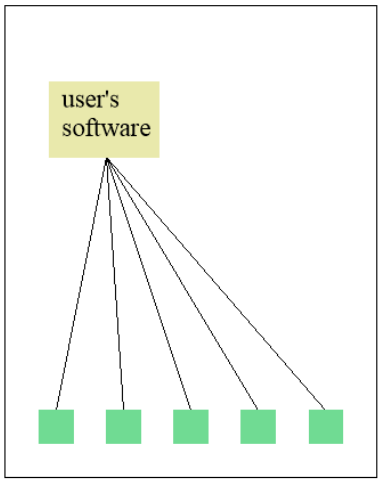
The library function uses the first argument, the lvalue, to locate ABC[], the second argument, Index, to move to the needed element, and the third argument, Value, to store in this element.

After the user calls this library function, they can access ABC[].

The user accesses the library function `dot()` using `box1.dot()`. What should I do if I wanted to call `dash()` from within `dot()`? By experimenting, I found that we should just write `dash()` and not `box1.dash()`. Yes, this is intuitively obvious but that doesn't always translate into something that the compiler will accept. I did not experiment with a library function calling a function found in a different class. I suspect that I would use the `class.function()` format.

So why is this useful?

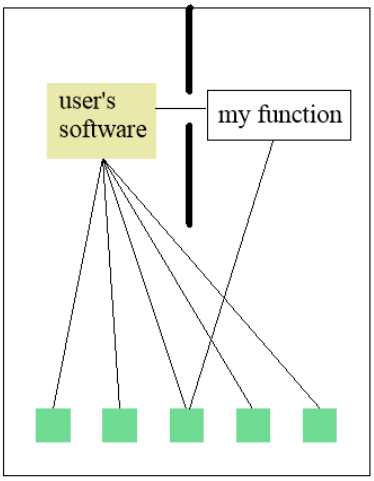
I find that having the least amount of functionality in examples is helpful in focusing on the new material. However, it doesn't let me appreciate the power of my newly earned understanding.



Here I have some software written by a user. They defined a series of variables and access them.

I write a function that the user wants. They could copy my function and put it in with their code. Any variables used by the function and their software would have to match up. If I make any changes to my code, the user's code is now out of date.

For the above example, these changes are no big deal. But what if hundreds of variables were involved along with thousands of lines of code. It gets messy plus prone to errors.



Instead, the user does a #include of my library. My code knows nothing of the user's variables. Yet, the user can call my function, which includes the lvalue of all involved variables, and it all works.

Debugging A Library

I found that bugs in Box.h and Box.cpp caused compiler errors in my user-defined code that made no sense. For example, I forgot to have the last character in my class definition be "**;**" and it caused my Arduino IDE to print, in part

```
UsersCodeTry2:3:6: error: expected initializer before 'box1'
 Box1 box1(5);
      ^~~~
…UsersCodeTry2.ino: In function 'void loop()':
UsersCodeTry2:15:1: error: 'box1' was not declared in this scope
 box1.ChangeAnElement(ABC, Ind, Val);
 ^~~~
```

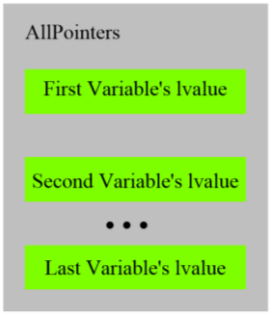
There is nothing wrong with these lines.

A sane approach to debugging a library is to start with a known good framework for the .h, .cpp, and keywords files. It should have a sample of variables and functions. Verify that it compiles without errors. Then remove the sample lines and add a few lines of your variables and code. You will likely need to modify the keywords file, so don't forget to restart the IDE. Run the compiler and verify there are no errors. If there are, start commenting out lines until you find the culprit. Fix the error before you add more variables and functions. Do not add more lines than you are willing to back out.

In general, I treat compiler errors related to a library as telling me I did ***something*** wrong. I do not try to make sense of its details.

Giving a Library Full Access to Your Variables

I have demonstrated how to pass a single array into the library. Not too exciting.



But consider the case of the user building an array that contains hundreds of lvalues. Then they pass the library the lvalue of just this array. With that one lvalue, the library knows where these hundreds of variables are located. Now, ***that*** is powerful!

As I developed this example, I ran into two surprises.

The user calls the library function `PassPointer()` with the name of the array that holds all of my lvalues, `PointerArray`. Looking in the .cpp file, I have

```
void Fcom::PassPointer(int *PointerArray){
     for(byte index = 0; index < 16; index++)
      {
      _PointerArray[index] = PointerArray[index];
      }
}
```

This code is similar to that found on page 16. It identifies the argument as a pointer to an integer. Since this integer is the first element of an array, I can index through it.

I must create the variable `*PointerArray` within this function so I can have `PointerArray[index]`. This means that `PointerArray[index]` is only defined within my PassPointer function. If I want other library functions to access the PointerArray, I must have a global array. I do this by first defining `_PointerArray[]` as a global private array in my .h file:

```
int _PointerArray [16];
```

Then I copy the local array `PointerArray[]` into `_PointerArray[]`. Any function in the library can then access `_PointerArray[]`.

There may be a far better way to accomplish this task, but I could not find it.

My second surprise was how another library function must use `_PointerArray[]`.

In this first example, I define the function `PutInLoop()`. Back in the user's code, I put the lvalue of the Boolean variable, `Modem`, into `PointerArray[4]`. This value can be accessed with `_PointerArray[4]` presented on the last page.

```
void Fcom::PutInLoop(){
    bool *Modem_Bool = _PointerArray[4];
```

The key thing to notice here is that I must define a local variable, `*Modem_Bool`. I found no way to define `*Modem_Bool` globally. This is because I must define variables in the .h file and give them values in the .cpp file. This prevents me from defining ***and*** assigning the variable on the same line.

If another library function needs access to the `Modem` Boolean, they must create their own local copy.

My second example involves making a local copy of an array. In my user's code, I assigned the lvalue of `NavigationData[]` to `PointerArray[2]`. This was copied over to `_PointerArray[2]` by `PassPointer()`.

```
void Fcom::GlobalNavigationSatelliteSystem(){
    byte *_NavigationData = PointerArrayInt[2];
```

Within `GlobalNavigationSatelliteSystem()`, I need to access `NavigationData[]`. I do this by defining `*_NavigationData`. It points to the first element of `NavigationData[]`, which happens to be an array of bytes.

As shown on page 16, I can then use `_NavigationData` as an array and read from it

```
Altitude = _NavigationData[6];
```

or write to it

```
NavigationDataByte[6] = Altitude;
```

Conversion of the Library to Compiled Code

I haven't yet arrived at my final step – compiling my library so it cannot be modified by the casual user. I do understand that this is how commercial software is distributed. It can be reverse-engineered, but it is not easy.

Acknowledgments

I am in debt to Dave Kellogg for helping me on this journey. I am also grateful to all of those people that wrote such great tutorials.

Thanks to "pert" at forum.arduino.cc for getting me back on track while I was confused about private variables.

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