Pointers

When we covered how variables are passed by reference, we alluded to the notion of pointers – the memory address where data is stored is actually passed along, resulting in a change to the original data. There is a method in C++ to explicitly get the address of an identifier. The addresses are called **pointers** and is one of the simple data types.

To declare something to be a pointer, use a * in the definition of the variable.
For example, the following defines a variable to be a pointer to a long:

```
long lngVal=0;
long *lngPtr;
```

This declares two variables. One (lngVal) is a normal variable with enough space allocated to it to hold a long, initialized in this case to zero. The other (lngPtr) is a variable with enough space allocated to it to hold a memory address. It is intended that this memory address be that of a long somewhere in memory.

This is illustrated in the figure below for memory ranging from 0000-FFFF (2 bytes):

	Address (2 bytes)	Data
lngVal	F120	00000000
lngPtr	F124	????

The variable lngVal is allocated enough memory to hold a long at address F120. The variable lngPtr is allocated enough memory to hold a memory address at address F124. Note that at this point, lngPtr does not "point" to anything yet. Memory has simply been allocated for the pointer itself (i.e. the memory address).

Consider the following:

```
char charVal='A', *charPtr;
```

This allocates enough memory to hold a char in charVal and enough memory to hold a memory address in charPtr:

	Address (2 bytes)	Data
charVal	F120	41
charPtr	F121	????

Note that we still have two bytes allocated for charPtr because we need two bytes to hold a memory address. However, charVal only has one byte allocated to it because a char only requires one byte of storage. This means that on a particular system, a pointer for a char, int, float, or any other data type is actually going to have the same number of bytes allocated for it, and that it will be easy to treat one type like another by changing pointer values.

One last point, it is common to initialize pointers to NULL. This stores the value "0" into a pointer and can be used to test if a pointer has been set or not:

```
long *lngPtr = NULL;
```

The fact that the constant NULL is actually the number 0 leads to an ambiguity problem. Consider the overloaded function below:

```
void func(int *p);
void func(int i);
```

Which function will be invoked if we call `func(NULL)`? Since NULL is the number 0, both are equally valid. C++11 resolves this problem by introducing a new constant, `nullptr`. `nullptr` is not the integer zero, but it is a literal constant used to represent a null pointer. Use `nullptr` anywhere you would have used NULL for a pointer. For example, we can write:

```
double *there = nullptr;
```

If we invoked func(nullptr) then C++11 can determine that we mean to invoke the first function with *p, not the second with int i.

Pointer Operators

The **&** or **address** operator returns the address of its operand. To use it, precede an identifier by & to retrieve its address. For example:

```
long lngVal=0,  *lngPtr;

cout << &lngVal  << endl;
```

This will print out the address of the variable lngVal. Using our contrived example of where this is stored in memory would result in outputting **F120** (hex – actually the decimal equivalent would be output).

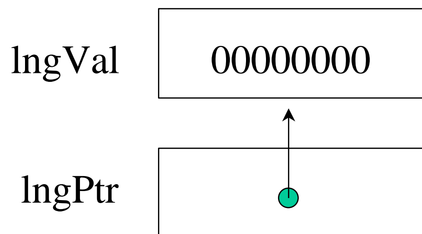
We can assign a memory address to a pointer variable. To assign the address of lngVal to lngPtr we use the assignment operator:

```
lngPtr = &lngVal;
```

This copies the address of lngVal into the contents of lngPtr:

	Address (2 bytes)	Data
lngVal	F120	00000000
lngPtr	F124	F120

Often this is depicted graphically:



What happens if we print the value of lngPtr? Let's look at two properties of lngPtr:

```
cout << lngPtr << endl;
cout << &lngPtr << endl;
```

The first statement prints out the contents of lngPtr. In this case, the contents are the memory address F120 (hex) , so F120 (its decimal equivalent) will be output first.

The second statement prints out the address of lngPtr. Yes, we can get the address of a pointer just like any other variable! This prints out the value F124 (hex).

If we want to access the value being referenced by the pointer, then we must use the ***** or **indirection / dereferencing operator**. To use it, add the operator in front of the pointer to dereference:

```
cout << *lngPtr << endl;
```

This will output 0. The dereference operator follows the pointer and references the value being pointed at. In other words, this operator fetches the memory address stored in the pointer. Then it goes to that memory address and fetches the data stored at that address.

Sample exercise: What is the output of the following code?

```
int x, *xPtr=NULL, y, *yPtr=NULL;

x=10;
y=20;
xPtr = &x;
yPtr = &y;
```

```
cout << x << *xPtr << y << *yPtr << endl;
yPtr = xPtr;
cout << *yPtr << endl;
cout << xPtr << &x << endl;
```

The first cout statement will print the values of x and y:

```
10 10  20 20
```

The second cout statement will print 10 because yPtr is changed to the address of x:

```
10
```

The last statement will print the address of x, which will vary when run:

```
1045123 1045123
```

What is the output of the following?

```
int x, *xPtr=NULL, y, *yPtr=NULL;
x=10;
xPtr = &x;
*xPtr = 20;
cout << x << *xPtr << endl;
```

By setting xPtr to the address of x, and then changing it by dereference, the contents of x are changed and the output is 20:

```
20 20
```

What is wrong with the following?

```
int x, *xPtr=NULL, y, *yPtr=NULL;
x = 10;
*xPtr = 20;
cout << x << *xPtr << endl;
```

In this case, we are trying to store the value 20 into memory address NULL (0). This is not allowed and will likely cause the program to crash. We should only dereference a pointer after it is pointing to some allocated memory. In this case, the pointer is not pointing to any allocated memory. We can point to allocated memory by setting a pointer to another variable using the & operator. We can also use the **new** operator described next to allocate memory for us.

The above is a very common bug, so watch out and make sure that your pointers hold valid references!

Dynamic Variables

In the previous discussion, we created a variable and stored its address in a pointer variable. By now, you should be asking, but why? Why would you go to this much trouble if it is only an alternative way to do something you can already do? The ability to store and manipulate the addresses of variables allows us to create a new kind a variable: **dynamic**. Previously, we described two kinds of data: static and automatic. Static variables exist for the lifetime of the program. Automatic variables are created when control reaches their declaration and deleted when control exits the block in which they are declared. Dynamic variables, in contrast, are created and destroyed as needed by the program itself. **new** is an operator that allocates a dynamic variable; **delete** is an operator that deallocates it. The format is :

```
ptrVar = new <type>
delete ptrVar;
```

Here is a code sample:

```
char*  charPtr=NULL;
int*   intPtr=NULL;
char*  aCString=NULL;

charPtr = new char;              // Allocate a character
intPtr  = new int;               // Allocate an integer
aCString = new char[10];         // Allocate 10 characters

*charPtr = 'A';
*intPtr  = 55;
strcpy(aCString, "foo");

cout << aCString ;
cout << " "  << *charPtr  << "." ;
cout << *intPtr   << endl;

delete  charPtr;
delete  intPtr;
delete[]  aCString;                        // Note [] for an array
```

This code segment produces the output:

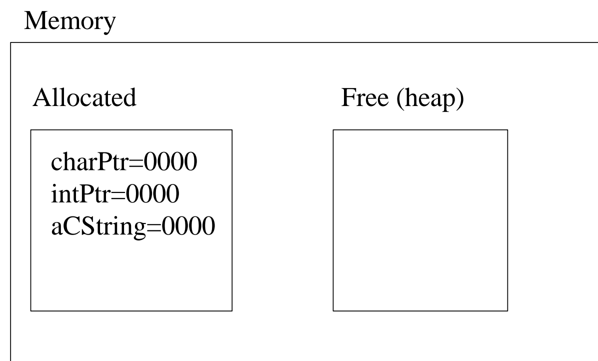
```
foo A 55
```

In this example, the variables created by **new** to hold the data exist only from the execution of **new** to the execution of **delete**. The data is allocated from a large block of free memory called the **heap**. The locations that were allocated for them can now be

allocated for some other variables. Here, we are talking about three variables, so space is not important. However, when you graduate to writing very large programs, this technique of using space only during the time that you need it becomes important.

The variables generated by new are called dynamic variables because they are created at run time. Dynamic variables are used just like any other variables of the same type, but pointer variables should only be assigned to one another, tested for equality, and dereferenced.

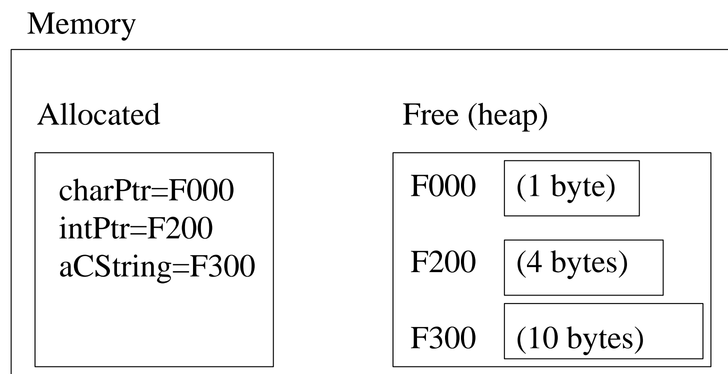
If we trace memory usage through the execution of this program, the picture looks something like the following. Initially, we have to allocate memory to hold the pointers themselves. They are initialized to NULL.



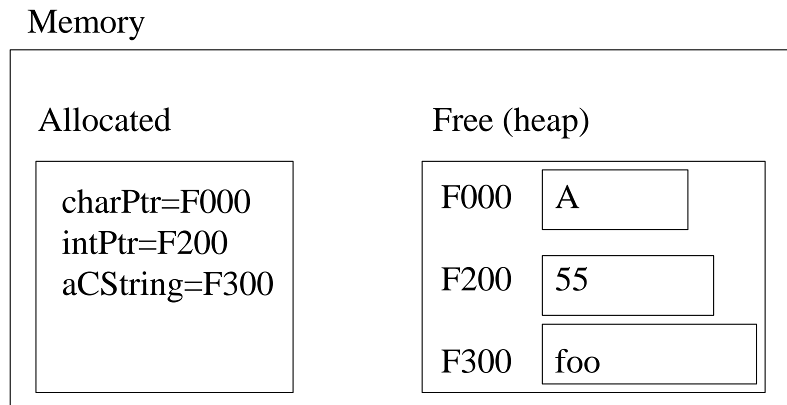
Upon executing the **new** instructions:

```
charPtr = new char;              // Allocate a character
intPtr  = new int;               // Allocate an integer
aCString = new char[10];         // Allocate 10 characters
```

We will allocate memory from the heap and return a pointer to that newly allocated memory:



When we execute the assignments we store data into the allocated memory:



Finally, when we **delete** the pointers, the memory that was allocated to them is released, and a subsequent **new** might claim memory that was once allocated to charPtr, intPtr, or aCString.

Important:

- Never try to store data into a pointer that does not have space allocated for it.

```
int *xPtr;
*xPtr = 10;          ← NO!  Memory has not been allocated for the data
```

- Never delete a pointer that was not allocated with new. If the pointer was assigned to an auto variable, there is no need to delete it.

```
int *xPtr, x=5;
xPtr = &x;
delete xPtr;         ← NO!
```

- Always delete a pointer that is allocated with new when you are finished! If you do not explicitly delete the memory, the computer will consider it to still be "in use". Consider:

```
int *xPtr;
xPtr = new int;
*xPtr = 5;
xPtr = NULL;         ← Lose handle to allocated memory!
```

  In this case we changed xPtr, it is no longer pointing to the allocated memory! By changing the pointer, we lost the "handle" to the allocated memory. There is now no way to regain this memory unless the operating system reclaims it. Errors like this are called **memory leaks**. If you notice your application growing in memory

usage over time when it shouldn't, it is probably due to a memory leak. These bugs are difficult to track down.

Passing Parameters By Reference Using Pointers

Pointers give us another way to pass parameters to functions and have the functions change the contents of the variable back in the caller. This is the way that call-by-reference was accomplished in C programs. When calling a function, the address of the variable should be passed in the caller using &, and in the prototype a pointer should be expected using *. Here is an example using a function that computes the cube of a value:

```
void CubeIt(int *valPtr);          // Prototype

void CubeIt(int *valPtr)           // Definition
{
        *valPtr = (*valPtr) * (*valPtr) * (*valPtr);
}

int main()
{
        int x;
        x=10;

        CubeIt(&x);
        cout << x << endl;

        return 0;
}
```

The above CubeIt function will dereference the pointer passed in, and change the value of the original data. Consequently, 1000 will be output (10^3). In terms of functionality, the code given above is equivalent to a program that passes variables by reference instead:

```
void CubeIt(int &valPtr);          // Prototype

void CubeIt(int &valPtr)           // Definition
{
        valPtr = (valPtr) * (valPtr) * (valPtr);
}

int main()
{
        int x;
        x=10;
```

```
		CubeIt(x);
		cout << x << endl;

		return 0;
	}
```

Some people actually prefer to pass variables by pointers because it becomes explicit what data might change by looking at the function call. When passing by reference, it is impossible to tell what might change without looking at the function prototype. However, a drawback of passing by pointer is the need to dereference the pointer variables inside the function.

One note on passing arrays, since an array is already a pointer, there is no need to use the & and pass an array via pointers. As we will see shortly, **&arr[0]** is actually equivalent to just **arr**.

Using the const Qualifier with Pointers

If you have a value that should not be modified, use the const qualifier. This is typically used in function calls where you want to make sure that the function shouldn't modify a pointer's value:

```
	void func(const int *xPtr)
	{
		*xPtr = 5;		//	← Not allowed!  Constant
	}
```

This is sometimes used to pass large structures to functions. Pass-by-value can be inefficient, since a large variable needs to be copied on the stack. It is more efficient to just pass a pointer to the large structure instead. By declaring the pointer value to be constant, we sort of simulate "pass by value" by using pointers. More on structures in an upcoming lesson.

Pointer Arithmetic

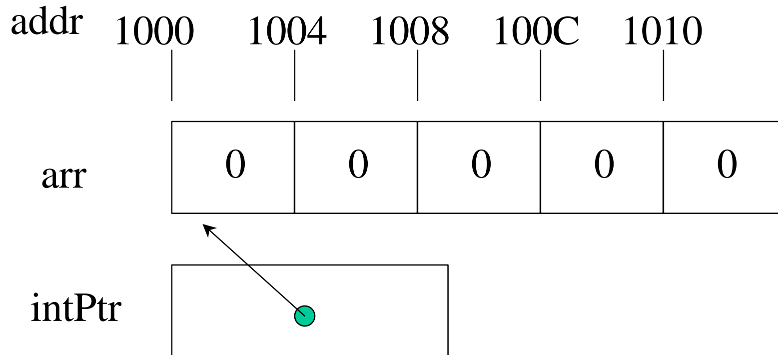
A limited set of arithmetic can be performed on pointers. For example, we can add a value to a pointer. But what does it mean to increment a pointer? The answer is that the pointer is changed by an amount equal to the size of the type that the pointer is referencing.

For example, let's say that we declare an array of integers:

```
	int arr[5]={0};
	int intPtr;
```

```
intPtr = &arr[0];                    (equivalent to   intPtr = arr);
```

The allocation in memory looks something like this:



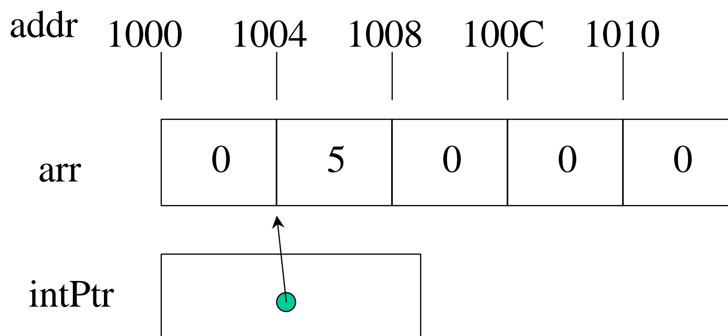
In this picture, intPtr is pointing to the beginning of the array, or it contains address 1000 (hex).

What happens if we do intPtr = intPtr + 1?

In normal arithmetic, we would get 1000 +1 or 1001. However, this is not the case with pointer arithmetic. Instead, the pointer is changed by the integer times the size of the object to which the pointer refers. In this case, each integer is 4 bytes. So by adding one to intPtr, we are actually adding 4, to hold 1004 and point to the next element in the array. If we execute the following:

```
intPtr = intPtr +1;
*intPtr = 5;
```

The picture is changed to:

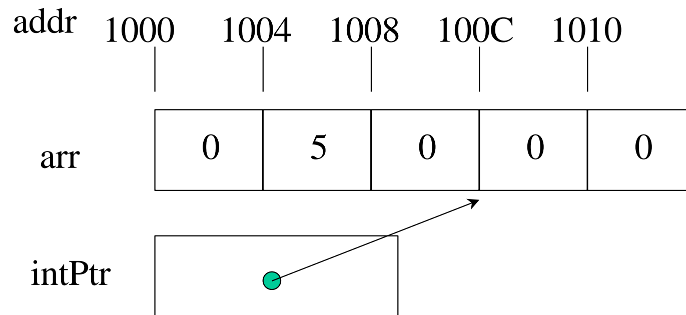


If we were to print out arr[1] then we would be outputting the value 5!

Similarly, if we now set:

```
intPtr +=2;
```

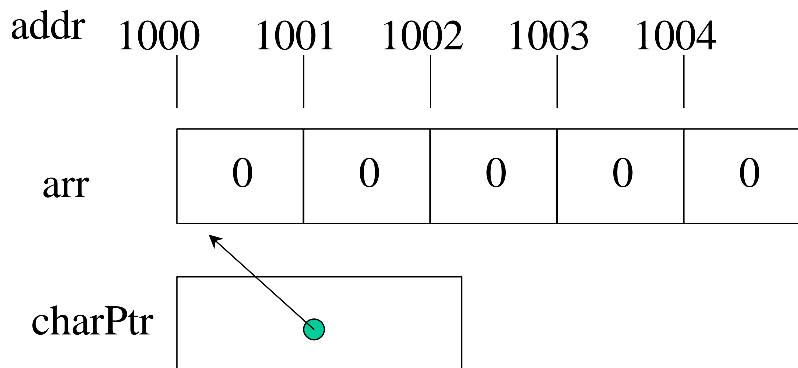
The pointer is changed to move down two elements, referencing 100C:



Note what happens if we do the same thing on a char array:

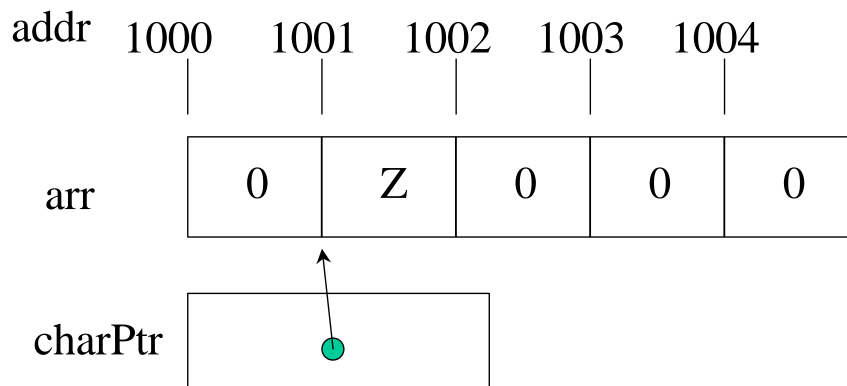
```
char arr[5]={0};
char *charPtr;

charPtr = arr;
```

Since a char only takes up one byte, memory is allocated as:



If we now change charPtr = charPtr + 1 then we really will be incrementing charPtr's contents by one, because the size of the data that charPtr points to is only one byte long:

```
charPtr++;
*charPtr = 'Z';
```



Exercise: What does the following code do?

```
char s[6]="foobar";
char *ptr1,*ptr2, c;

ptr1=&s[0];
ptr2=&s[4];
while (ptr1<ptr2) {
        c=*ptr1;
        *ptr1=*ptr2;
        *ptr2=c;
        ptr1++;
        ptr2--;
}
cout << s << endl;
```

Arrays and Pointers

Arrays and pointers are intimately related in C++. They may be used almost interchangeably. An array name is basically a constant pointer; it cannot be changed.

Consider the following pointer and array:

```
int arr[10], intPtr=NULL;
intPtr = &arr[0];
```

The above is equivalent to:

```
intPtr = arr;
```

Array element arr[3] could also be referenced as:

```
*(intPtr + 3)
```

The number three above is the offset into the pointer. Since pointer arithmetic will add the proper amount based on the size of each element, we'll be guaranteed to get the fourth value (i.e. the value in arr[3]).

If we wanted the address of arr[3] we could use:

```
&arr[3]
```

this is equivalent to:

```
(intPtr + 3);
```

So we can use a pointer to access data within an array by just adding in the proper offset. We can also treat an array like a pointer!

```
&(arr + 3)
```

Returns the same thing as (intPtr + 3), because intPtr and arr are just both pointers to the beginning of the array.

We can even treat pointers like they are arrays:

```
intPtr[3]   will access the same element as arr[3]
```

To recap, we can apply almost all pointer operations to arrays, and vice versa:

```
int arr[10], intPtr=NULL;
intPtr = &arr[0];

*(intPtr + 3)    ==    arr[3];
*(arr + 3)       ==    intPtr[3];
(intPtr + 3)     ==    &arr[3];
(arr + 3)        ==    &intPtr[3];
```

One place where arrays are NOT like pointers is that you cannot change an array. Arrays are like constant pointers:

```
intPtr = intPtr + 1;        // VALID, intPtr now points to arr[1]
arr = arr + 1;              // INVALID, can't change an array!
```

As an example of the interchangeability between pointers and arrays, sometimes function parameters are passed as pointers:

```
int SumArray(int *p, int numElements)
{
        int sum=0;

        for (int j=0; j<numElements; j++, p++) sum+=(*p);
        return sum;
}

int main()
{
        int arr[5]={1,4,3,2,1};
        int total;
        total = SumArray(arr, 5);
}
```

The previous example computes and returns the sum of the array elements passed in. Note that we are passing an array, but the prototype calls for a pointer to an integer. The compiler doesn't complain, because internally the two types (int array, int pointer) are equivalent.

Function Pointers

It turns out that variables aren't the only things we can pass around using pointers. One of the neat things we can do in C++ is to actually pass a function as a parameter by passing the pointer to the function! For obvious reasons, this technique is called function pointers.

Let's revisit the selection-sort routine that we worked on earlier:

```
void SelectionSort(int a[], int lastindex)
{
	int i,j, index, value, temp;

	for (j=0; j<=lastindex; j++) {
		// Find minimum between j and lastindex
		index = j;
		value = a[j];
		for (i=j; i<=lastindex; i++) {
			if (a[i]<value) {
				index = i;
				value = a[i];
			}
		}
		// Swap with j
		temp = a[j];
		a[j] = a[index];
		a[index] = temp;
	}
}
```

Since this routine finds the minimum each time, we get the array in ascending sorted order, from smallest to largest.

What if we want the data in reverse order? Then we'll need a whole new function. Or do we? All we really need to do is replace the line:

```
	if (a[i]<value) {
```

with

```
	if (a[i]>value) {
```

And the result will be a list in descending order! One way to make this change is to pass a function that makes this comparison, instead of writing a whole new function (or checking a flag and putting in some logic to test either). First lets write two functions, one for less than and one for greater than;

```
bool LessThan(int num1, int num2)
{
	return (num1 < num2);
}

bool GreaterThan(int num1, int num2)
{
	return (num1 > num2);
}
```

Now we can change our SelectionSort function to accept a function as a parameter! Note the syntax: we need parentheses around the function name, and we also need to specify the input parameters:

```
void SelectionSort(int a[], int lastindex, bool (*compare)(int num1, int num2))
{
	int i,j, index, value, temp;

	for (j=0; j<=lastindex; j++) {
		// Find minimum between j and lastindex
		index = j;
		value = a[j];
		for (i=j; i<=lastindex; i++) {
			if ((*compare)(a[i],value)) {
				index = i;
				value = a[i];
			}
		}
		// Swap with j
		temp = a[j];
		a[j] = a[index];
		a[index] = temp;
	}
}
```

Now we can invoke this from main, using the appropriate function if we want the data in ascending or descending order:

```
int main()
{
	int a[5]={55, 34, 4, 11, 21};

	SelectionSort(a,4, LessThan);  // Ascending order
	for (int i=0; i<5; i++) cout << a[i] << endl;

	cout << endl;

	SelectionSort(a,4, GreaterThan);  // Descending order
	for (int i=0; i<5; i++) cout << a[i] << endl;

	return 0;
}
```

Output:

```
4
11
21
34
55

55
34
21
11
4
```

Function pointers are powerful constructs, which lend themselves to particular problems. One place they can be useful is in a menu-driven system. A master array can actually hold pointers to functions that should be invoked when particular selections are made. This allows a good deal of abstraction in that if desired, entirely new functions or functionality could be invoked by changing the master array instead of having to go in and change the menu function.