Pointers and Arrays

Reference: Russell Chapter 2

Nuno Alves (nalves@wne.edu), College of Engineering

Why do we need to learn this (again)?

- Pointers and Arrays are a bit tricky and we need to understand them if we want to implement queues and more advanced structures.
- Some of the RTOS code we will be using uses these structures quite often.
- Since we are dealing with embedded systems with limited resources, its always a good idea to learn how to carefully dynamically allocate memory for uses.

What is a pointer?

A pointer is a variable that contains an address, usually of another variable, but it can be anything in the addressable space.

Memory, in general, is just a block of addressable bits that may be manipulated in various-sized groups, but the bits are all physically the same.

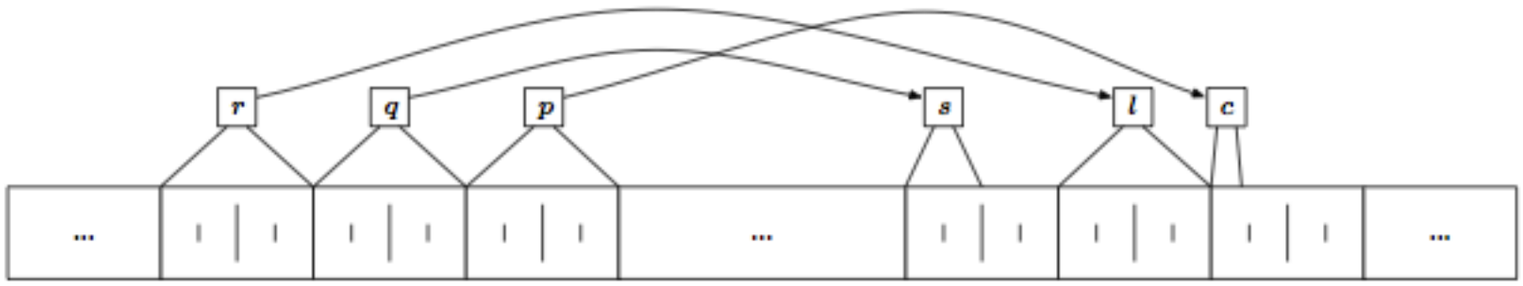
Type	Number of 8-bit Units	Total Contiguous Bits
char	1	8
short	2	16
long	4	32

Then, a pointer is a group of cells (usually 2 or 4 8-bit cells) that holds an address.

Pointer example

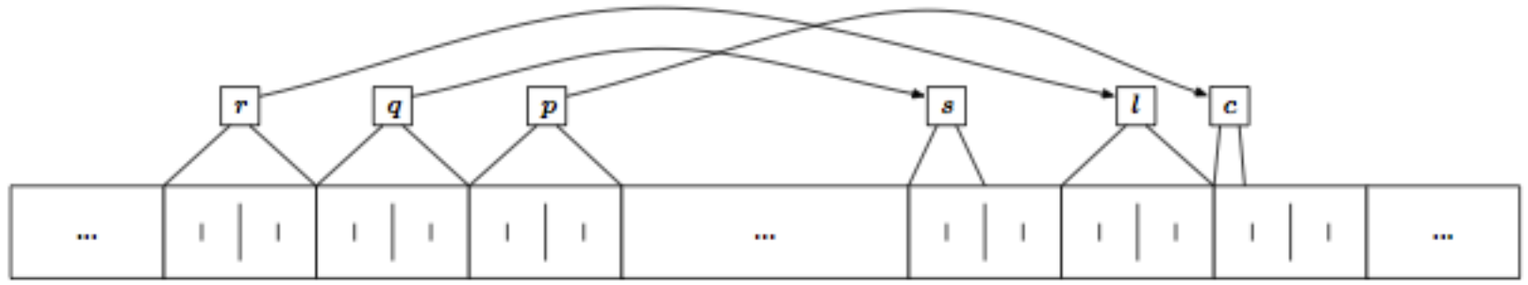
Let **r** be a pointer that points to **L**, **q** be a pointer that points to **S**, and **p** be a pointer that points to **C**.

Let **S** be a 16-bit short, **L** be a 32-bit long, and **C** be an 8-bit char. In this example, pointers are 32-bit variables meaning the processor has 32-bit addressable space.



An example of a schematic view of memory. The smallest division represents one 8-bit memory location.

Unary operators: & and *



- The amount of space reserved for any pointer is the same, no matter to what type the pointer is pointing.
- The way to make the pointer point to some variable is via the operator **&**, which gives the address of the label on its right-hand-side.

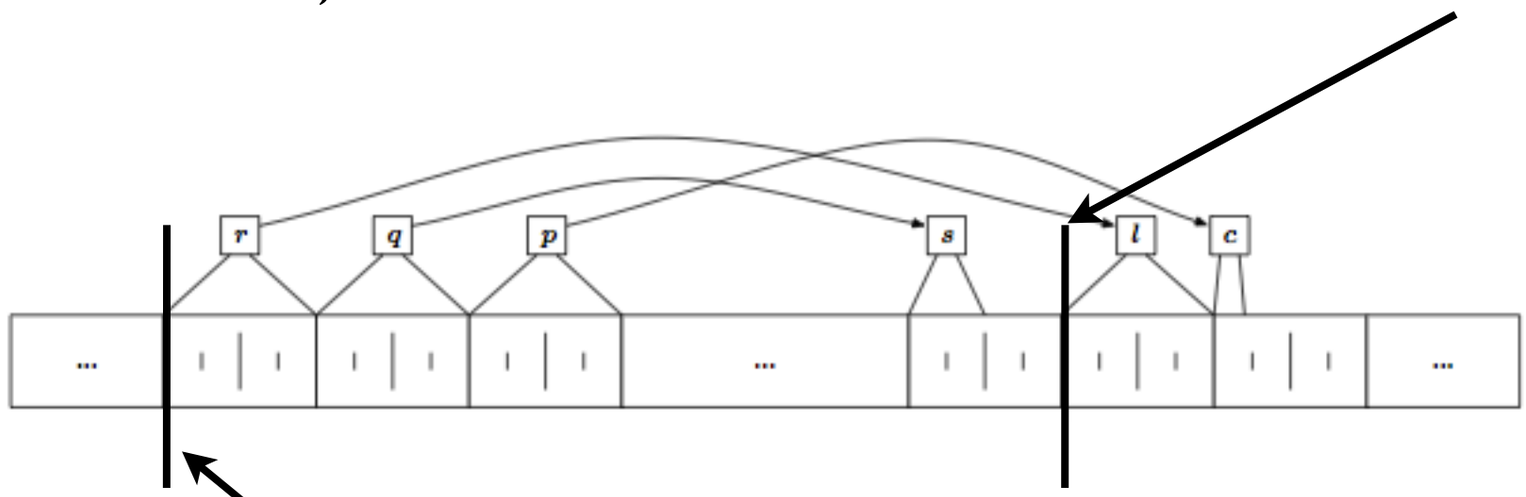
```
p = &c;
q = &s;
r = &l;
```

- To access the contents of a variable using a pointer is via the operator *, which is called the dereferencing operator.

Example

p = &C;
q = &S;
r = &L;

- p is the address of variable **C**
- q is the address of variable **S**
- r is the address of variable **L**



&r is the address of the pointer r
&q is the address of the pointer q

Another example

p = &c;

- **p** contains the address of variable **c.**
- **p** must then be a pointer.

c = 0;

- Variable **c** will now have the value 0.

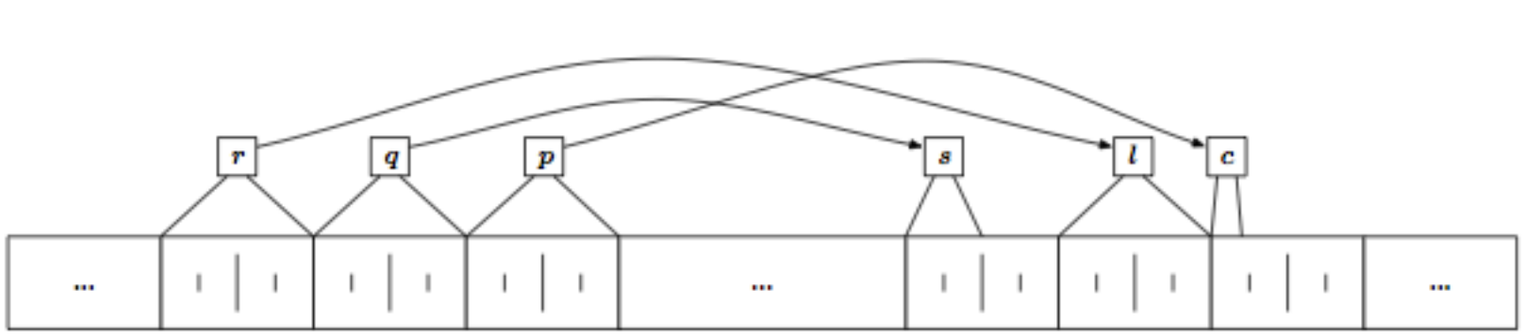
*p = 10;

- I go to the address of the pointer **p** and set it to 10.
- Variable **c** will now have the value of 10.

How to declare pointers?

- To declare a pointer, just add the * symbol to the left of the variable name.

- char *p;
- short *q;
- long *r;



- Don't forget that the space allocated to hold **p**, **q** and **r** is all the same (usually 32-bits on modern microprocessors), but what they point to is different.

Pointer precedence

- The operators *, &, ++ and -- all have the same level of operator precedence (which is very high).
- When the compiler parses a line of source code, it resolves operators with the same precedence from right-to-left.
- Thus, the statement *p++; will have a very different result compared to (*p)++;
- *p++ => increments the address stored in p first, then reads the contents of the address which p++ is pointing to.
- (*p)++ => reads the dereferenced address first and increment the resulting value without changing the address stored in p.

Precedence examples

address of variable **c** is 100

contents of address101 is 0

Assume that:

- char c = 5;
- char *p;
- p = &c;

	Before			After			
Instruction	&c = 100	101	p	&c = 100	101	p	*p
c = *p + 1;	5	0	100	6	0	100	6
*p += 1;	5	0	100	6	0	100	6
++*p;	5	0	100	6	0	100	6
(*p)++;	5	0	100	6	0	100	6
*p++;	5	0	100	5	0	101	0

- Remember *p will print out the contents of the variable that p is pointing to.
- *p += 1 is the same as *p = *p +1

Memory map example

- Consider the following code:

```
\\ variable declarations
int j=3;
int *pa;
int a;
int b = 0x2F;

\\ executable code
pa = &j;
a = *pa;
*pa = b;
```

- Assume compiler assigns memory up from 0x3000. Also assume that an integer is 32 bits (4 bytes). We can chow each byte of memory as it would be assigned by the compiler.

Name	Addr				
	3000				

Memory map example

```
\\ variable declarations
int j=3;
int *pa;
int a;
int b = 0x2F;

\\ executable code
pa = &j;
a = *pa;
*pa = b;
```

- 32 bits = $32_2 = 20_{16}$ = 0x0020
- 0x2F = 0000 0000 0010 1111_2 = 0 0 2 F

Assuming a 32 bit address space

Name	Addr				
j	3000	0	0	0	3
pa	3001	x	x	x	x
a	3002	x	x	x	x
b	3003	0	0	2	F

Executable code

Name	Addr				
j	3000	0	0	2	F
pa	3001	3	0	0	0
a	3002	0	0	0	3
b	3003	0	0	2	F

Pointers in Arduino

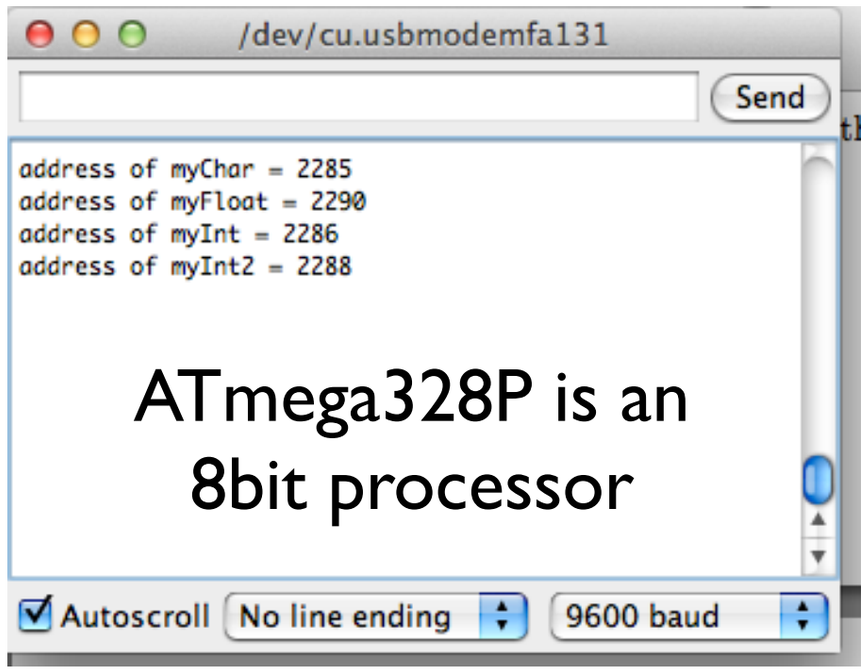
```
void setup(){}

void loop()
{
int myInt;
float myFloat;
int* intPtr = &myInt;
float* floatPtr = &myFloat;

int szint = sizeof(myInt);
int szflt = sizeof(myFloat);

printf("intPtr   = %d\n", (int)intPtr);
printf("intPtr+1 = %d\n", (int)intPtr+1);
printf("fltPtr   = %d\n", (int)floatPtr);
printf("fltPtr++ = %d\n", (int)floatPtr++ );
}
```

- This compiles just fine (after all its pure ANSI-C

- However nothing will show up in the serial monitor, since the microprocessor does not know what to do with printf.

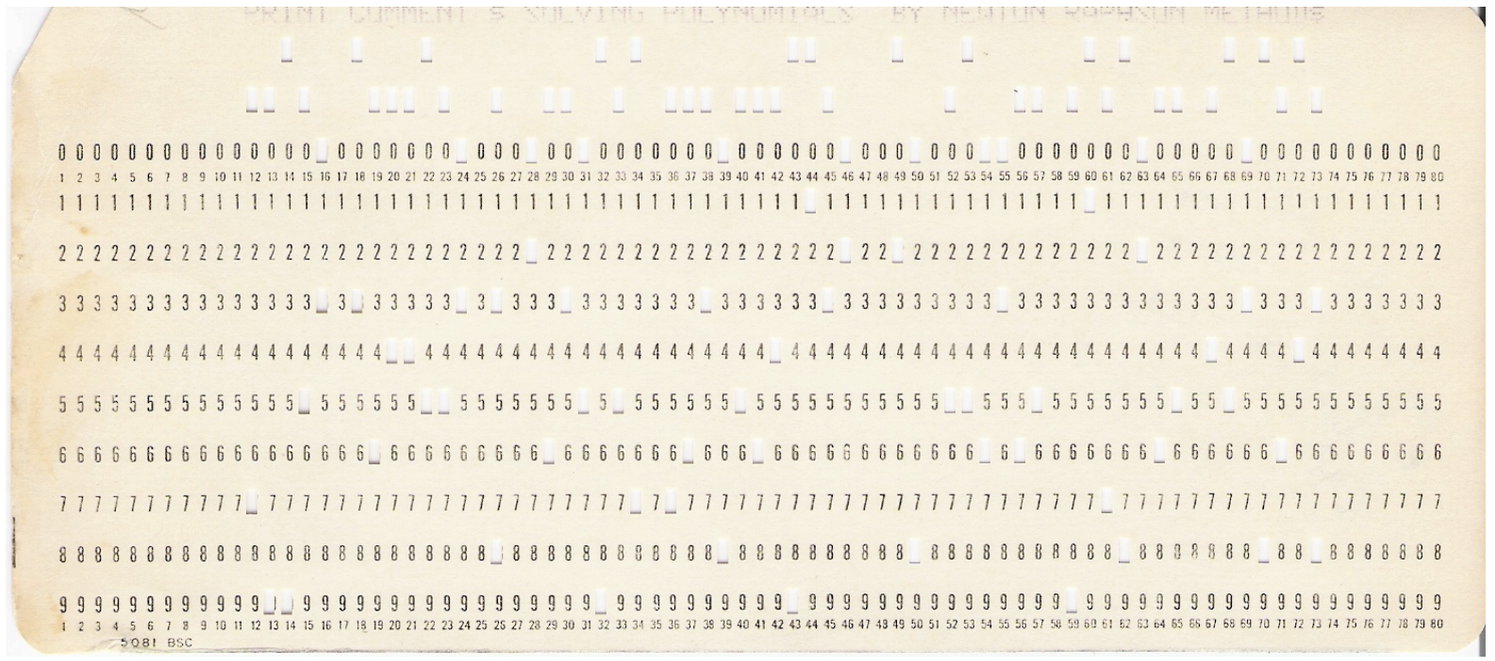
Pointers in Arduino

```
void setup() { Serial.begin(9600); }
void loop()
{
char myChar;
float myFloat;
int myInt,myInt2;

char* charPtr = &myChar;
float* floatPtr = &myFloat;
int* intPtr = &myInt;
int* intPtr2 = &myInt2;

Serial.print("address of myChar = ");
Serial.println(int (charPtr));
Serial.print("address of myFloat = ");
Serial.println(int (floatPtr));
Serial.print("address of myInt = ");
Serial.println(int (intPtr));
Serial.print("address of myInt2 = ");
Serial.println(int (intPtr2));

while (1);
}
```



- Slightly more interesting example of pointers in Arduino.
- Integers and floats require 16 bits, Chars require 8 bits. That is why myChar and myInt are separated by 8 bits (or 1 byte), while myInt2 and myFloat are separated by 16 bits (or 2 bytes).

Historical example: IBM-PC 80



	addr	col 0		col 1		col 2	
row 0	B8000	ch	cl	ch	cl	...	
row 1	B80A0	ch	cl	...			
...		ch	...				
		...					

ch = char

cl = color

- This computer had a 80 col x 40 row display buffer (80 column is also legacy!)

- This is the graphics hardware that generates the clunky screen that you use to configure your BIOS.

Writing characters to the screen

- This is how you would write a character on the screen.
- The display had a permanent memory address where anything placed in there, would be displayed to the screen.

```
#define DISP_BUFFER 0xB8000
int row, col;
char disp_char;
char *p;
//       Let's display "A" in row 4, col 20
disp_char = 'A';
row = 4;          // for example ...
col = 20;         //  "    "

p = (char *) (DISP_BUFFER + 2 *(80*row + col))
*p = disp_char;
```

trying to find which address to write into

```
#define DISP_BUFFER 0xB8000
int row, col;
char disp_char;
char *p;
//       Let's display "A" in row 4, col 20
disp_char = 'A';
row = 4;        // for example ...
col = 20;       //  "    "

p = (char *) (DISP_BUFFER + 2 *(80*row + col))
*p = disp_char;
```

	addr	col 0		col 1		col 2	
row 0	B8000	ch	cl	ch	cl	...	
row 1	B80A0	ch	cl	...			
...		ch	...				
		...					

ch = char

cl = color

...both character and colors are 8 bits... since we can address each 8 bit memory address

	col 0		col 1		col 2	
row 0 (base 16)	B8000	B8001	B8002	B8003	B8004	B8005
row 0 (base 10)	753664	753665	753664 + 1*2		753664 + 2*2	

(row 0, col 1) : $753664_{10} + 2_{10} * (80_{10} * 0_{10} + 1_{10}) = 753664_{10} + 2_{10} = B8002_{16}$

(row 0, col 2) : $753664_{10} + 2_{10} * (80_{10} * 0_{10} + 2_{10}) = 753664_{10} + 4_{10} = B8004_{16}$

(row 1, col 0) : $753664_{10} + 2_{10} * (80_{10} * 1_{10} + 0_{10}) = 753664_{10} + 160_{10} = B80A0_{16}$

(row 1, col 1) : $753664_{10} + 2_{10} * (80_{10} * 1_{10} + 1_{10}) = 753664_{10} + 162_{10} = B80A1_{16}$

Arrays

Arrays

- Arrays are blocks of consecutive types.
- An array variable is similar to a pointer of that type that has been initialized to the address of the first entry of the block.
- That is, pointers are similar to uninitialized array variables.

```
short *p;
short a[10];

p = &(a[2]);

/* The following expressions are true. */
*(p) == a[2];
*(p+1) == a[3];
```

- This code declares a pointer and an array of length 10.
- The array declaration causes the compiler to reserve a block of 10 consecutive 16-bit cells.

Moving through the array index


```
short *p;
short a[10];

p = &(a[2]);

/* The following expressions are true. */
*(p) == a[2];
*(p+1) == a[3];
```

- Adding 1 to **p** is equivalent to adding 1 to the array index.
- Assuming a short is 16 bits, the compiler knows **p** points to a 16-bit memory elements, so when altering the address which p points to, the compiler adjusts based on the type.

Another array example

- This example declares a pointer to a short type and an array of 5 longs.

```
short *p;
long a[5];
char c;

p = (short *) (&(a[0]));
```

- The pointer is loaded with the address of the base of the array.

- The compiler knows that **p** is supposed to point to types of 16-bit shorts, and it will do so.

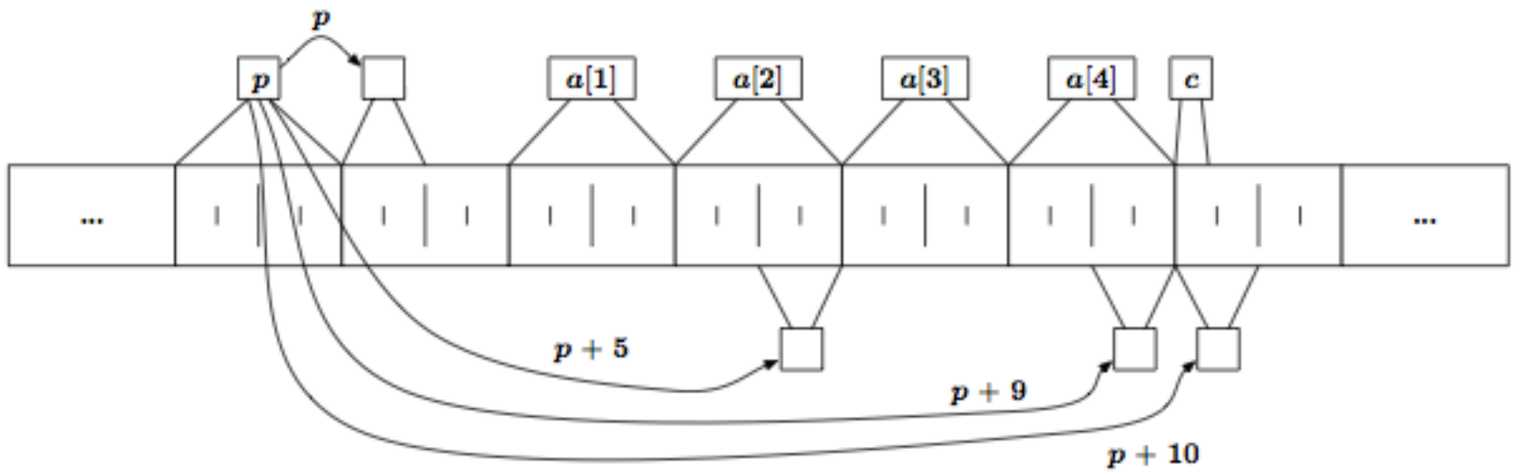
- The code simply initializes the address to which p points. After that, there is nothing to prevent the code from adding any number to the address and dereferencing the result.

- It is very easy to overwrite memory locations by accident (or on purpose – consts may not be so constant after all).

Another array example

- It is very easy to overwrite memory locations by accident (or on purpose – consts may not be so constant after all).

```
short *p;
long a[5];
char c;

p = (short *) (&(a[0]));
```



Cycling through arrays using pointers

Note: this is C++ code, and it will not work on the Arduino (especially the cout, namespace and iostream library)

```
#include <iostream>
using namespace std;

int main ()
{
    int array[10]={0,11,22,33,44,55,66,77,88,99};
    int *p = &array[0];

    for (int i=0 ; i<10 ; i++)
    {
        cout<<p<<"\t";
        cout<<*p<<endl;
        //incrementing the pointer address
        *(p++);
    }
}
```

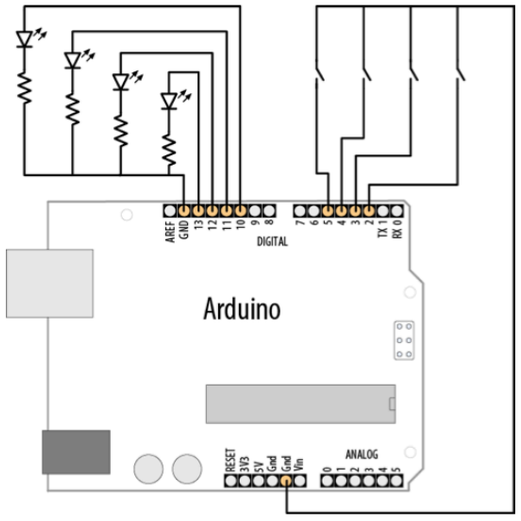
Code that cycles through the contents of an array using pointers.

Array summary

```
int q[3] = {0,1,2};

or

int i, q[3];

...

for (i=0;i<3;i++) q[i] = 0;
```

```
int a[10], *p;
p = &a[0];
a[3]  <-----> *(p+3)
are exactly the same.

AND

a <-------> &a[0]
are exactly the same.
```

Example of arrays in Arduino




```
int inputPins[] = {2,3,4,5};  // create an array of pins for switch inputs

int ledPins[] = {10,11,12,13};  // create array of output pins for LEDs

void setup()
{
  for(int index = 0; index < 4; index++)
  {
    pinMode(ledPins[index], OUTPUT);          // declare LED as output
    pinMode(inputPins[index], INPUT);         // declare pushbutton as input
    digitalWrite(inputPins[index],HIGH);      // enable pull-up resistors
                                              //(see Recipe 5.2)
  }
}

void loop(){
  for(int index = 0; index < 4; index++)
  {
    int val = digitalRead(inputPins[i]);  // read input value
    if (val == LOW)                       // check if the switch is pressed
    {
      digitalWrite(ledPins[index], HIGH); // turn LED on if switch is pressed
    }
    else
    {
      digitalWrite(ledPins[i], LOW);      // turn LED off
    }
  }
}
```

- This code creates two arrays: an array of integers for pins connected to switches and an array of pins connected to LEDs.
- The state of the switches and LEDs are stored in arrays.

Why do we care about pointers?

- Passing information in and out of function calls.

- Dynamic memory allocation.

- Especially in embedded devices, we can use pointers to access memory-mapped registers in order to manage various peripheral devices.

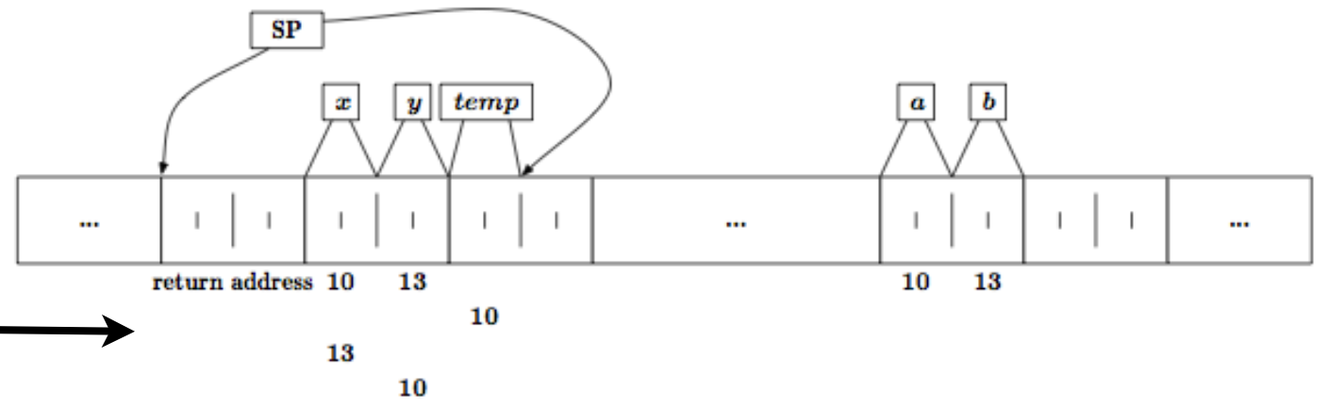
Passing by reference

Swap function

- This code does not swap the **a** and **b** values.
- Once we enter the function, the function input values are **copied** onto the stack!
- The program counter is copied into the stack (stack pointer (**SP**)).
- There are three local variables, **temp**, **x** (holds a copy of **a**) and **y** (holds a copy of **b**).

```
void main (void)
{
   short a = 10;
   short b = 13;

   swap (a,b);

   /* a == ?, b == ? */
}

void swap (short x, short y)
{
   short temp;

   temp = x;
   x = y;
   y = temp;
}
```

How the values in the local memory change after each instruction of the function is executed.



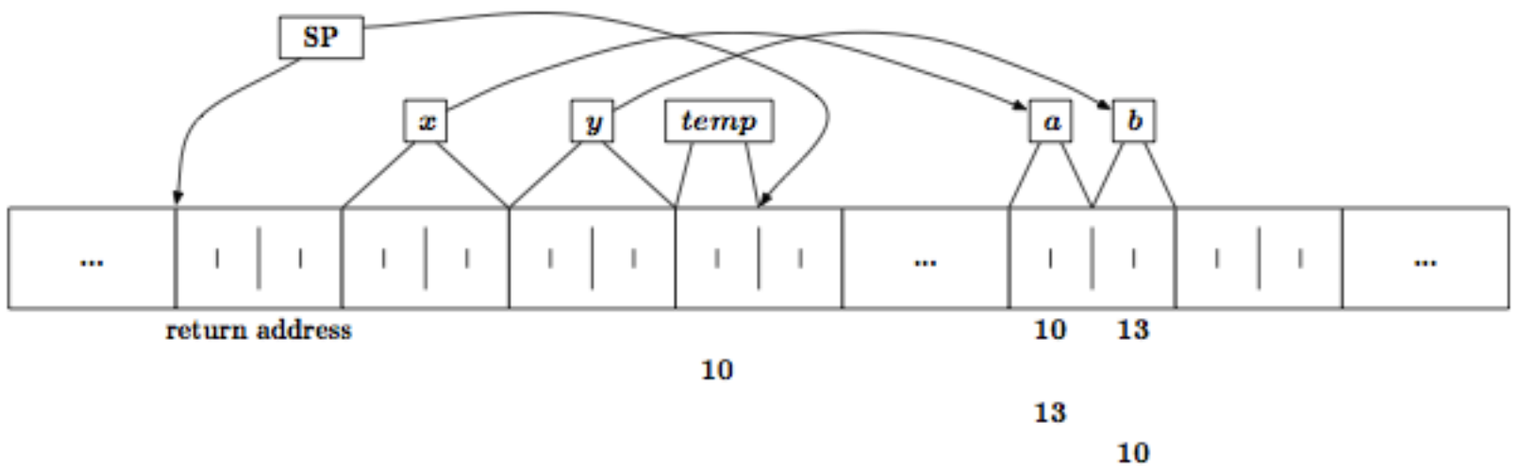
Passing by reference

```
void main (void)
{
   short a = 10;
   short b = 13;

   /* Now we pass the address of the variables we want to change. */
   swap (&a,&b);

   /* a == ?, b == ? */
}

void swap (short *x, short *y)
{
   short temp;

   temp = *x;
   *x = *y;
   *y = temp;
}
```

- This code does what we intended because the addresses of a and b are passed into the function.
- The function accesses their values by indirect reference via the pointers x and y.

Passing by reference

```
void main (void)
{
   short a = 10;
   short b = 13;

   /* Now we pass the address of the variables we want to change. */
   swap (&a,&b);

   /* a == ?, b == ? */
}

void swap (short *x, short *y)
{
   short temp;

   temp = *x;
   *x = *y;
   *y = temp;
}
```



Another reason to use pointers

- We can use pointers for passing large pieces of memory into a function.
- For example, suppose we wanted to pass a structure (we will talk about this soon) of 10,000 longs into a function.
- If you tried passing by value, the stack would need to hold all 80,000 bytes.
- However, if we used a pointer to the base of the structure, the stack only needs to hold the 4-byte base address.

Dynamic memory allocation

Dangers of dynamic memory allocation: memory leaks

- You can dynamically allocate memory at run-time and have the base be referred via a pointer.
- The compiler doesn't reserve the consecutive bytes of memory as for an array declaration. Instead, the CPU is directed to find a block of consecutive bytes in memory that are not being used and return the base address.
- WARNING: it is easy to lose memory if a function allocates memory but never frees it. This is a memory leak, and, eventually, repeated calls to the function will consume all of the available memory, causing the program to crash.

Dangers of dynamic memory allocation: debugging problems

- WARNING: run-time allocation is not a great idea for embedded programs.
- Memory leaks in PC applications are difficult enough to track down using all of the powerful debugging tools available on a host system.
- Many embedded system tools are extremely limited, and so debugging an embedded memory issue tends to be exceptionally difficult.

Example of dynamic memory allocation

The free function returns memory back to the system.

```
#define NUMBER_OF_SHORTS_TO_ALLOCATE 10

short *p;

p = (short *) malloc (sizeof (short) * NUMBER_OF_SHORTS_TO_ALLOCATE);
if (p == NULL)
{
   /* error -- need to tell the user and stop execution */
}

/* skipping code */

free (p);
```

- The C operator macro sizeof() returns the number of bytes used by the given argument.
- The function malloc() returns the base to a block of requested bytes. In this code, the processor must locate 10 consecutive unused short cells and return the base address.
- If they cannot be found, the special "invalid address" NULL is returned.

Alternative way to dynamically allocate memory

```
#define NUMBER_OF_SHORTS_TO_ALLOCATE 10

short *p;

p = (short *) calloc (NUMBER_OF_SHORTS_TO_ALLOCATE, sizeof (short));
if (p == NULL)
{
   /* error -- need to tell the user and stop execution */
}

/* skipping code */

free (p);
```

I can use the allocated memory as an array

Note: this is C++ code, and it will not work on the Arduino (especially the cout, namespace and iostream library)

```
#include <iostream>
using namespace std;

#define NUMBER_SHORTS_TO_ALLOCATE 20

int main ()
{

    short *p;
    p = (short *) calloc (NUMBER_SHORTS_TO_ALLOCATE,sizeof(short));

    if (p==NULL){ cout<<"could not allocate"<<endl; }

    cout<<p[10]<<endl;
    p[10]=12;
    cout<<p[10]<<endl;

    free(p);
}
```

Adjusting the size of the allocated memory

```c
#define NUMBER_OF_SHORTS_TO_ALLOCATE 10
#define NUMBER_OF_SHORTS_TO_REALLOCATE 12

short *p;

p = (short *) calloc (NUMBER_OF_SHORTS_TO_ALLOCATE, sizeof (short));
if (p == NULL)
{
   /* error -- need to tell the user and stop execution */
}

/* skipping code */

p = (short *) realloc (p, sizeof (short) *
    NUMBER_OF_SHORTS_TO_REALLOCATE);
if (p == NULL)
{
   /* error -- need to tell the user and stop execution */
}

free (p);
```

Multi-dimensional arrays

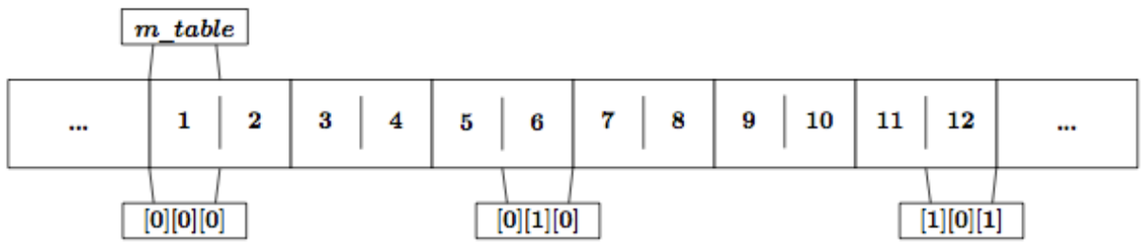
Multi-dimensional arrays

- Note that the const qualifier is not required for the array definition.
- However, many times large arrays are used for various lookup tables and so are meant to be fixed conceptually.
- Remember the first entry in every dimension is 0.

```
#define MAX_ROWS 2
#define MAX_COLS 5

const short m_table[MAX_ROWS][MAX_COLS] =
{
   {1, 2, 3, 4, 5},
   {6, 7, 8, 9, 10}
};

/* Then the following are true. */
m_table[0][1] == 2;
m_table[1][4] == 10;
```

Multi-dimensional array memory organization

- How does the compiler organizes individual elements in a multi-dimensional array?
- Note that the rightmost array index varies fastest as elements are accessed in storage order.
- Index MAX_DIM2 will vary most frequently.




```
#define MAX_DIM0 3
#define MAX_DIM1 2
#define MAX_DIM2 5

const short m_table[MAX_DIM0][MAX_DIM1][MAX_DIM2] =
{
   {
      {1, 2, 3, 4, 5},
      {6, 7, 8, 9, 10}
   },
   {
      {11, 12, 13, 14, 15},
      {16, 17, 18, 19, 20}
   },
   {
      {21, 22, 23, 24, 25},
      {26. 27. 28. 29. 30}
   },
};

/* Then the following are true. */
m_table[0][1][2] == 8;
m_table[2][1][3] == 29;
```

Generic pointers

Generic (void) pointers

- Sometimes we want a pointer which is not locked to a specific type. It can potentially point to anything.

- `void *name; //declare a generic pointer called name`

  - **name** can point to anything in the computer.
  - **name** cannot be dereferenced with *
  - Must instead assign value of void pointer to a pointer of the type you want.

```
void* myGenericPtr;
int t, *ip, myvalue = 3;
myGenericPtr = &myvalue;  // OK
t = *myGenericPtr;  // NO!!

//******************

ip = nyGenericPtr;
t = *ip;    // OK!!
```

Null pointer

- If a pointer has the value NULL, it points to nothing. NULL is a predefined constant.

```
int i,*ip = NULL;

[...]

if(ip == NULL) {
    // I haven't defined ip yet
    }
    else {  // OK, now I can use it!
    i = *ip;}
```

Function pointers

What is a function pointer?

- While a function is not a variable, it is a label and still has an address.
- As a result, it is possible to define function pointers, which can be assigned and treated as any other pointer variable.
- For example, they can be passed into other functions, in particular, callbacks into Real-Time Operating Systems (RTOSes) or hooks in an Interrupt Service Routine (ISR) vector table.

Why do we need a function pointer?

- A function pointer is a variable that stores the address of a function that can later be called through that function pointer.
- Why do we need this?
  - Sometimes we want the same function have different behaviors at different times.
  - Sometimes we just want to have a queue filled with function pointers, so as we transverse the queue, we merely execute the a function without doing any extra operations.

Function Pointer Syntax

```
void (*foo)(int);
```

- In this example, **foo** is a pointer to a function taking one argument, an integer, and that returns void.

- It's as if you're declaring a function called "***foo**", which takes an int and returns void.

- If ***foo** is a function, then **foo** must be a pointer to a function. (Similarly, a declaration like int ***x** can be read as ***x** is an int, so **x** must be a pointer to an int.)

- The declaration for a function pointer is similar to the declaration of a function but with (***func_name**) where you'd normally just put **func_name**.

Initializing function pointers

```
#include <iostream>
using namespace std;

void my_int_func(int x)
{  cout<<x<<endl;  }

int main()
{
    void (*foo)(int);
    //the ampersand(&) is optional
    foo = &my_int_func;

    return 0;
}
```

- To initialize a function pointer, you must give it the address of a function in your program.

- The syntax is like any other variable.

Note: this is C++ code, and it will not work on the Arduino (especially the cout, namespace and iostream library)

Using a function pointer

```
#include <iostream>
using namespace std;

void my_int_func(int x)
{  cout<<x<<endl;  }

int main()
{
    void (*foo)(int);
    foo = &my_int_func;

    // calling my_int_func
    //(note that you do not need
    //to write (*foo)(2)
    foo( 2 );

    //but you can... if you want
    (*foo)( 2 );

    return 0;
}
```

- To call the function pointed to by a function pointer, you treat the function pointer as though it were the name of the function you wish to call.

- The act of calling it performs the dereference; there's no need to do it yourself.

Note: this is C++ code, and it will not work on the Arduino (especially the cout, namespace and iostream library)

```
#include <iostream>
using namespace std;

// The four arithmetic operations
float Plus    (float a, float b) { return a+b; }
float Minus   (float a, float b) { return a-b; }
float Multiply(float a, float b) { return a*b; }
float Divide  (float a, float b) { return a/b; }

// Solution with a switch-statement
// <opCode> specifies which operation to execute
void Switch(float a, float b, char opCode)
{
   float result;

   // execute operation
   switch(opCode)
   {
      case '+' : result = Plus     (a, b); break;
      case '-' : result = Minus    (a, b); break;
      case '*' : result = Multiply (a, b); break;
      case '/' : result = Divide   (a, b); break;
   }

  // display result
   cout << "Switch: 2+5=" << result << endl;
}

int main()
{
   Switch(2, 5, '+');
}
```

Note: this is C++ code, and it will not work on the Arduino (especially the cout, namespace and iostream library)

- The main function performs the arithmetic operation through an intermediate function (switch).

```
#include <iostream>
using namespace std;

// The four arithmetic operations
float Plus    (float a, float b) { return a+b; }
float Minus   (float a, float b) { return a-b; }
float Multiply(float a, float b) { return a*b; }
float Divide  (float a, float b) { return a/b; }

// Solution with a function pointer
// <pt2Func> is a function pointer and points to
// a function which takes two floats and returns a
// float. The function pointer "specifies" which
// operation shall be executed.

void Switch_With_Function_Pointer(float a, float b,
float (*pt2Func)(float, float))
{
// call using function pointer
   float result = pt2Func(a, b);

   cout <<"Switch replaced by func. ptr.: 2-5=";
   // display result
   cout << result << endl;
}


int main()
{
   Switch_With_Function_Pointer(2, 5, &Minus);
}
```

Note: this is C++ code, and it will not work on the Arduino (especially the cout, namespace and iostream library)

- Solution with a function pointer
- The function pointer "specifies" which operation shall be executed

Summary of function pointers

```
return_type (* variableName)(argument_list);
```

```
int(*IntFuncPtr)();
\\   IntFuncPtr is a pointer to a function with
\\    no arguments which returns an int

double(*doubleFuncPtr)(int arg1, char arg2)
\\   doubleFuncPtr is a pointer to a function with
\\     an int and a char arg which returns a double.
```

Assignment to function pointers:

```
int (* IFP)(int x) = NULL;    // empty function pointer
int realfunction( int x);     // an actual function

IFP = &realfunction;
IFP = realfunction;    // can skip &
```

Dereferencing function pointers:

```
(*IFP)(5)    // call function realfunction with arg 5

IFP(5)       // can also use function pointer just like
             // original function name
```

How to use arrays of function pointers ?

```
#include<iostream>
using namespace std;

int DoIt (float number, char char1, char char2)
{ cout<<"... inside DoIt()"<<endl; return(number); }

int DoMore (float number, char char1, char char2)
{ cout<<"... inside DoMore()"<<endl; return (number); }

int main()
{
   // define arrays and ini each element to NULL,
   // <funcArr> is an array with 10 pointers to
   // functions which return an
   // int and take a float and two char

   int (*funcArr[10])(float, char, char) = {NULL};

   // assign the function's address 'DoIt' and 'DoMore'
   funcArr[0] = funcArr[2] = &DoIt;
   funcArr[1] = funcArr[3] = &DoMore;

   // calling a function using an index to address the
   // function pointer
   // short form for calling function (position #1)
   funcArr[1](12, 'a', 'b');

   // "correct" way of calling function (position #0)
   int return_val=(*funcArr[0])(12, 'a', 'b');
   (*funcArr[1])(56, 'a', 'b');

  cout<<(*funcArr[0])(34, 'a', 'b')<<endl;
}
```

Note: this is C++ code, and it will not work on the Arduino (especially the cout, namespace and iostream library)

- These are two silly functions that take 3 arguments, print something on the screen and returns a float... which is also the first argument

```
#include<iostream>
using namespace std;

int DoIt (float number, char char1, char char2)
{ cout<<"... inside DoIt()"<<endl; return(number); }

int DoMore (float number, char char1, char char2)
{ cout<<"... inside DoMore()"<<endl; return (number); }

int main()
{
   // define arrays and ini each element to NULL,
   // <funcArr> is an array with 10 pointers to
   // functions which return an
   // int and take a float and two char

   int (*funcArr[10])(float, char, char) = {NULL};

   // assign the function's address 'DoIt' and 'DoMore'
   funcArr[0] = funcArr[2] = &DoIt;
   funcArr[1] = funcArr[3] = &DoMore;

   // calling a function using an index to address the
   // function pointer
   // short form for calling function (position #1)
   funcArr[1](12, 'a', 'b');

   // "correct" way of calling function (position #0)
   int return_val=(*funcArr[0])(12, 'a', 'b');
   (*funcArr[1])(56, 'a', 'b');

  cout<<(*funcArr[0])(34, 'a', 'b')<<endl;
}
```

Note: this is C++ code, and it will not work on the Arduino (especially the cout, namespace and iostream library)

- Here I am creating an array of 10 positions, that will store a pointers to a function that can take a (float, char, char) as arguments.

- Initially the function pointers are all set to NULL.

```
#include<iostream>
using namespace std;

int DoIt (float number, char char1, char char2)
{ cout<<"... inside DoIt()"<<endl; return(number); }

int DoMore (float number, char char1, char char2)
{ cout<<"... inside DoMore()"<<endl; return (number); }

int main()
{
   // define arrays and ini each element to NULL,
   // <funcArr> is an array with 10 pointers to
   // functions which return an
   // int and take a float and two char

   int (*funcArr[10])(float, char, char) = {NULL};

   // assign the function's address 'DoIt' and 'DoMore'
   funcArr[0] = funcArr[2] = &DoIt;
   funcArr[1] = funcArr[3] = &DoMore;

   // calling a function using an index to address the
   // function pointer
   // short form for calling function (position #1)
   funcArr[1](12, 'a', 'b');

   // "correct" way of calling function (position #0)
   int return_val=(*funcArr[0])(12, 'a', 'b');
   (*funcArr[1])(56, 'a', 'b');

  cout<<(*funcArr[0])(34, 'a', 'b')<<endl;
}
```

Note: this is C++ code, and it will not work on the Arduino (especially the cout, namespace and iostream library)

- Here I am adding the addresses of each function to a particular element of the funcArr array

- Make sure you don't call the elements that are not assigned!

- For example, element #4 and element #5 are still NULL... which point to nothing.

```
#include<iostream>
using namespace std;

int DoIt (float number, char char1, char char2)
{ cout<<"... inside DoIt()"<<endl; return(number); }

int DoMore (float number, char char1, char char2)
{ cout<<"... inside DoMore()"<<endl; return (number); }

int main()
{
   // define arrays and ini each element to NULL,
   // <funcArr> is an array with 10 pointers to
   // functions which return an
   // int and take a float and two char

   int (*funcArr[10])(float, char, char) = {NULL};

   // assign the function's address 'DoIt' and 'DoMore'
   funcArr[0] = funcArr[2] = &DoIt;
   funcArr[1] = funcArr[3] = &DoMore;

   // calling a function using an index to address the
   // function pointer
   // short form for calling function (position #1)
   funcArr[1](12, 'a', 'b');

   // "correct" way of calling function (position #0)
   int return_val=(*funcArr[0])(12, 'a', 'b');
   (*funcArr[1])(56, 'a', 'b');

  cout<<(*funcArr[0])(34, 'a', 'b')<<endl;
}
```

Note: this is C++ code, and it will not work on the Arduino (especially the cout, namespace and iostream library)

- I am calling the function that is on the position #1 of the array, with the (12,'a','b') as the arguments.

- This is the short form notation

- Of course, the DoMore function will return something, but we are not storing it anywhere.

```
#include<iostream>
using namespace std;

int DoIt (float number, char char1, char char2)
{ cout<<"... inside DoIt()"<<endl; return(number); }

int DoMore (float number, char char1, char char2)
{ cout<<"... inside DoMore()"<<endl; return (number); }

int main()
{
   // define arrays and ini each element to NULL,
   // <funcArr> is an array with 10 pointers to
   // functions which return an
   // int and take a float and two char

   int (*funcArr[10])(float, char, char) = {NULL};

   // assign the function's address 'DoIt' and 'DoMore'
   funcArr[0] = funcArr[2] = &DoIt;
   funcArr[1] = funcArr[3] = &DoMore;

   // calling a function using an index to address the
   // function pointer
   // short form for calling function (position #1)
   funcArr[1](12, 'a', 'b');

   // "correct" way of calling function (position #0)
   int return_val=(*funcArr[0])(12, 'a', 'b');
   (*funcArr[1])(56, 'a', 'b');

  cout<<(*funcArr[0])(34, 'a', 'b')<<endl;
}
```

Note: this is C++ code, and it will not work on the Arduino (especially the cout, namespace and iostream library)

- This is the "correct", albeit confusing, form of calling the function pointer.

- The return_val will keep the return value of the function.

```
#include<iostream>
using namespace std;

int DoIt (float number, char char1, char char2)
{ cout<<"... inside DoIt()"<<endl; return(number); }

int DoMore (float number, char char1, char char2)
{ cout<<"... inside DoMore()"<<endl; return (number); }

int main()
{
   // define arrays and ini each element to NULL,
   // <funcArr> is an array with 10 pointers to
   // functions which return an
   // int and take a float and two char

   int (*funcArr[10])(float, char, char) = {NULL};

   // assign the function's address 'DoIt' and 'DoMore'
   funcArr[0] = funcArr[2] = &DoIt;
   funcArr[1] = funcArr[3] = &DoMore;

   // calling a function using an index to address the
   // function pointer
   // short form for calling function (position #1)
   funcArr[1](12, 'a', 'b');

   // "correct" way of calling function (position #0)
   int return_val=(*funcArr[0])(12, 'a', 'b');
   (*funcArr[1])(56, 'a', 'b');

   cout<<(*funcArr[0])(34, 'a', 'b')<<endl;
}
```

Note: this is C++ code, and it will not work on the Arduino (especially the cout, namespace and iostream library)

- This just displays the return value of the function.

- **Warning**: If you call a position that hasn't a valid function pointer (e.g. position #5 in the funcArr for example) you will get a segmentation fault!

```
void DoIt ()
{ Serial.println("... inside DoIt()");
}

void DoMore ()
{ Serial.println("... inside DoMore()");
}

void setup()
{
  Serial.begin(9600);

   // define arrays and ini each element to NULL,
   // <funcArr> is an array with 10 pointers to
   // functions which return an
   // int and take a float and two char
   void (*funcArr[10])(void) = {NULL};

   // assign the function's address 'DoIt' and 'DoMore'
   funcArr[0] = funcArr[2] = &DoIt;
   funcArr[1] = funcArr[3] = &DoMore;

   // calling a function using an index to address the
   // function pointer
   // short form for calling function (position #1)
   funcArr[1]();
   funcArr[2]();
}

void loop ()
{
}
```

- Arduino friendly version of the code with two functions that take and return no arguments

```
int DoIt (float number, char char1, char char2)
{ Serial.println("... inside DoIt()"); return(number);
}

int DoMore (float number, char char1, char char2)
{ Serial.println("... inside DoMore()"); return(number);
}

void setup()
{
  Serial.begin(9600);

   // define arrays and ini each element to NULL,
   // <funcArr> is an array with 10 pointers to
   // functions which return an
   // int and take a float and two char
   int (*funcArr[10])(float, char, char) = {NULL};

   // assign the function's address 'DoIt' and 'DoMore'
   funcArr[0] = funcArr[2] = &DoIt;
   funcArr[1] = funcArr[3] = &DoMore;

   // calling a function using an index to address the
   // function pointer
   // short form for calling function (position #1)
   funcArr[1](12, 'a', 'b');
   funcArr[2](15, 'a', 'b');
}

void loop ()
{
}
```

- Arduino friendly version of the code with two functions that take three arguments and return an integer

Pointer arithmetic

Pointer arithmetic

- A powerful feature of pointers is the ability to compute with them like integers. However only some operations are allowed with pointers.

Example: Find the midpoint in an array.

```
#define SIZE 100
int  length, buffer[SIZE];
int *ptr1, *ptr2, *ptr3;

ptr1 = buffer;          // points to start of array
ptr2 = ptr1 + 100;      // points to end of array
length = ptr2 - ptr1;   // length = 100
ptr3 = ptr1 + length/2  // ptr3 points to mid-point of array
```

Allowed	Not Allowed
• Add a scalar to a pointer • Subtract pointers	• Add two pointers • Multiply or Divide Pointers • Multiply by a scalar • Divide by a scalar