Pointers and Arrays:

- In C, there is a strong relationship between pointers and arrays.
- The declaration `int a[10];` defines an array of 10 integers. The declaration `int *p;` defines `p` as a "**pointer to an int**".
- The assignment `p = a;` makes `p` an alias for the array and sets `p` to point to the first element of the array. (One could also write `p = &a[0];` the instructor prefers `p = &(a[0]);`)
- One can now reference members of the array using either `a` or `p`

```
a[4] = 9;
p[3] = 7;

int x = p[3] + a[4] * 2;
//Same as int x = a[3] + p[4] * 2;
```

Two Ways to Dereference Pointers:

- Let there be the declarations

```
int a[]={1,2,3,4,5,6,7,8};
int *intPtr;
```

- `a` is actually an address that can be reference with an offset; `a[0]`, so the square bracket operator `[ ]` is actually dereferencing with an offset

- In fact, we can do the same with the pointer variable.

- `intPtr=a;` or `intPtr=&(a[0]);`
- then
- `intPtr[0]` is the value 1;

- **So, we have two ways to dereference pointers:**
  - *** dereference**
  - **[index] dereference with an offset of index times sizeoftype**

- Professor Note: add avr asm equivalents to this slide

More Pointers & Arrays:

- If `p` points to a particular element of an array, then `p + 1` points to the next element of the array and `p + n` points `n` elements after `p`, REGARDLESS of the type (and/or size of each element) of the array.
- The meaning a "adding 1 to a pointer" is that `p + 1` points to the next element in the array.

- We will now introduce a terminology for this class referring to the type being pointed to or the type of an array. This will be called the "objective type".
  - Example:

```
int * p;    //The objective type is int
int a[10]; //The objective type is int
int ** p;  //The root objective type is int, but p
           //  has an objective type of int *
```

- Therefore the meaning a "adding an integer to a pointer" is that `p + i` is pointing to memory at address `p` with offset `i*sizeof(objective_type)`

More Pointers & Arrays:

- The name of an array is equivalent to a pointer to the first element of the array and vice-versa.
- Therefore, if a is the name of an array, the expression `a[ i ]` is equivalent to `*(a + i)` .
- It follows then that `&a[ i ]` and `(a + i)` are also equivalent. Both represent the address of the **i-th** element beyond `a`.
- On the other hand, if p is a pointer, then it may be used with a subscript as if it were the name of an array.
  - `p[ i ]` is identical to `*(p + i)`
- In short, **an array-and-index expression is equivalent to a pointer-and-offset expression and vice-versa.**
- Class preference is for `&(a[ i ])` over `&a[i]` for readability.

So, What's the Difference?

- If the name of an array is synonymous with a pointer to the first element of the array, and function parameters defined as arrays are "almost" like pointers, then what's the difference between an array name and a pointer?

- One difference is that an array name can only "point" to the first element of its array (there is an exception to this rule when the array is a function parameter). It can't be changed to point to anything else. A pointer may be changed to point to any variable or array of the appropriate type

- i.e. can't do this:

```
int vec[3] = {1,2,3};
int i;
vec = &i;
```

We aren't free to change the address stored in "variable" vec (may only be a compile time variable) like we can with an actual pointer. 'vec' is a like a const pointer

Array Name vs Pointer Example

```
int g, grades[ ] = {10, 20, 30, 40 }, myGrade = 100, yourGrade = 85,
*pGrades;

/* grades can be (and usually is) used as array name */
for (g = 0; g < 4; g++)
  printf("%d\n",grades[g]);

/* grades can be used as a pointer to its array if it doesn't change*/
for (g = 0; g < 4; g++)
  printf("%d\n", *(grades + g));

/* but grades can't point anywhere else */
grades = &myGrade; /* compiler error */

/* pGrades can be an alias for grades and used like an array name */
pGrades = grades; /* or pGrades = &(grades[0]); */
for( g = 0; g < 4; g++)
  printf( "%d\n", pGrades[g]);

/* pGrades can be an alias for grades and be used like a pointer that
changes */
for (g = 0; g < 4; g++)
  printf("%d\n",*(pGrades++));

/* BUT, pGrades can point to something else other than the grades array */
pGrades = &myGrade;
printf( "%d\n", &pGrades);
pGrades = &yourGrade;
printf( "%d\n", &pGrades);
```

pGrades = pGrades + 1; (after!)

Manipulation of array that are parameters

```
void testFunction(int array[]){
  int i;                int *array
  array[0])++; // no compiler error, as expected
  array=&i;    // no compiler error either, but
}              //    don't let this confuse you
```

- With respect to a function's formal parameters only, C treats an array just like a pointer **unlike other arrays that are not parameters.**

- Even though the assignment to array is allowed by the compiler, it serves no purpose here and seems like a generally confusing idea. (When we learn pointer arithmetic one could argue that array++ make sense for iterating through an array)

http://www.lysator.liu.se/c/c-faq/c-2.html

2.12: How do I declare a pointer to an array?

Usually, you don't want to. **When people speak casually of a pointer to an array, they usually mean a pointer to its first element**.

Instead of a pointer to an array, consider using a pointer to one of the array's elements. Arrays of type T decay into pointers to type T, which is convenient; subscripting or incrementing the resultant pointer accesses the individual members of the array. **True pointers to arrays, when subscripted or incremented, step over entire arrays, and are generally only useful when operating on arrays of arrays**, if at all.

If you really need to declare a pointer to an entire array, use something like "int (*ap)[N];" where N is the size of the array. If the size of the array is unknown, N can be omitted, but the resulting type, "pointer to array of unknown size," is useless.

2.13: Since array references decay to pointers, given int array[ ]; **what's the difference between array and &array?**

Under ANSI/ISO Standard C, &array yields a pointer, of type pointer-to-array-of-T, to the entire array (see also question 2.12). Under pre-ANSI C, the '&' in &array generally elicited a warning, and was generally ignored. Under all C compilers, an unadorned reference to an array yields a pointer, of type pointer-to-T, to the array's first element.

Manipulation of array that are parameters

- Note in the earlier slide that no mention was made of the type of the array. Why not? Because it **doesn't matter**!
- If "`p`" is an alias for an array of ints, then `p[ k ]` is the (k+1)-th int and so is `*(p + k)`.
- If "`p`" is an alias for an array of doubles, then `p[ k ]` is the (k+1)-th double and so is `*(p + k)`.
- Adding a constant, k, to a pointer (or array name) actually adds `k * sizeof(objective type)` to the value of the pointer.
- **This is one important reason why the type of a pointer must be specified when it's defined. Also, note that all pointers in a given system are the same size (one memory address).**

ptrAdd.c Example

```
int main()
{
  char c, *cPtr = &c;
  int i, *iPtr = &i;
  double d, *dPtr = &d;

  printf("\nThe addresses of c, i and d are:\n");
  printf("cPtr = %p, iPtr = %p, dPtr = %p\n", cPtr,
    iPtr, dPtr) ;
  cPtr = cPtr + 1 ;
  iPtr = iPtr + 1 ;
  dPtr = dPtr + 1 ;

  printf("\nThe values of cPtr, iPtr and dPtr are:\n") ;
  printf("cPtr = %p, iPtr = %p, dPtr = %p\n\n",
    cPtr, iPtr, dPtr) ;
  return 0;
}
```

Iff these are:
← 1 byte
← 2 bytes
← 4 bytes

If we start with: 0x1, 0x2, 0x3

Then the result: 0x2, 0x4, 0x7

Printing an Array

- The code below shows how to use a parameter array name as a pointer.
- Although this is not common and is more complex than is needed, it illustrates the important relationship between pointers and array names.
- Notice the expression *grades++.

same as `int * grades`

```
void printGrades( int grades[ ], int size )
{
  int i;
  for (i = 0; i < size; i++)
    printf( "%d\n", *grades++);
}
```

preferred to write as: `int (*grades)++`

- What about this prototype?

```
void printGrades( int *grades, int size);
```

- Alternate for comparison:

```
{
  int * gradesEnd = grades+size;
  for (; grades < gradesEnd; grades++)
    printf( "%d\n", *grades);
}
```

Passing Arrays

- When an array is passed to a function, the address of the array is copied onto the function parameter. Since an address is a pointer, the function parameter may be declared in either fashion. e.g.

```
int sumArray( int a[ ], int size)
```

  is equivalent to

```
int sumArray( int *a, int size)
```

  - The code in the function is free to use "a" as an array name or as a pointer as it sees fit.

- The compiler always sees "`a`" as a pointer. In fact, any error messages produced will refer to "`a`" as an `int *`.

```
int SumArray( int a[ ], int size)
{
  int k, sum = 0;
  for (k = 0; k < size; k++)
    sum += a[ k ];
  return sum;
}
```

- Note the need to pass the size, which is not typically required in higher-level languages where the size of an array can be queried with expressions similar to:
  - a.size()
    -or-
  - length(a)

- Gotcha:
  - `int a[8]` -or- `sizeof(a)` -or- `sizeof(int)` does not work here to determine the size of the array (which is why we don't teach it!)

Alternates:

```
int SumArray( int a[ ], int size)
{
  int k, sum = 0;
  for (k = 0; k < size; k++)
    sum += *(a + k);
  return sum;
}
```

```
int SumArray( int a[ ], int size)
{
  int k, sum = 0;
  for (k = 0; k < size; k++)
  {
    sum += *a;
    ++a;
  }
return sum;
}
```

Array Sizes

- Managing array sizes in C is not a minor issue. (On an exam you should be prepare for questions about determining the size of an array or ask you about designing functions using arrays.)
- Going outside the bounds of an array is not automatically checked and can lead to serious program or system crashes.
- Basic approaches for approaching the design of functions using arrays in parameters are:
  1. Use extra parameter to convey the number of elements in an array.
  2. Use a termination value in the array itself that can be discovered by iterating through the array.
  3. Use a predetermined size for the array or some other predetermined method for determining it.

Strings Revisited

- Recall that a string is represented as an array of characters terminated with a null (`\0`) character. As we've seen, arrays and pointers are closely related. A string constant may be declared as either `char[ ]` or `char*`
- as in

```
char hello[ ] = "Hello Bobby";
```

- or (almost) equivalently

```
char *hi = "Hello Bob";
```

- A typedef could also be used to simplify coding;

```
typedef char* STRING;
STRING hi = "Hello Bob";
```

- What does this code do?

```
char * ptrChar
ptrChar = &(hello[7]);
ptrChar[3] = 'b';
```

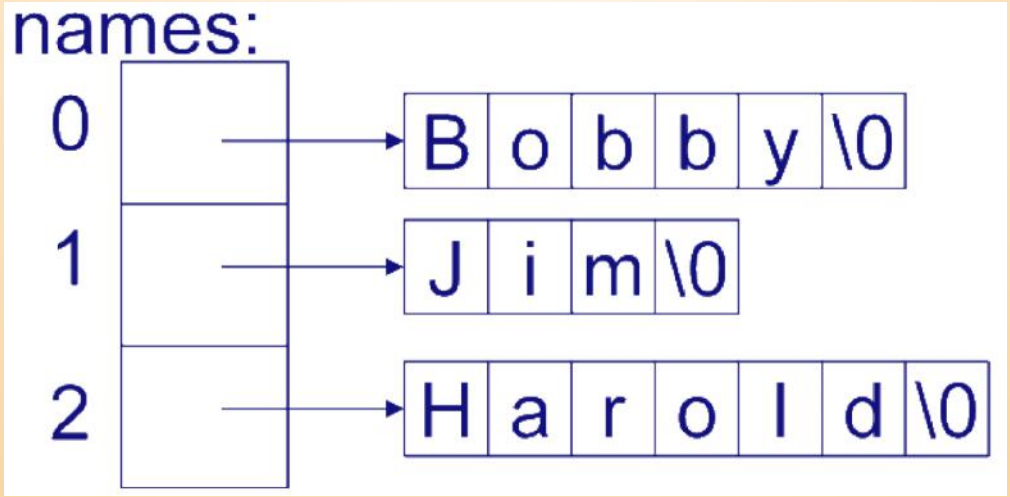
- -And what does this code do?

```
char hello[ ] = "Hello Bobby";
char * ptrChar;

ptrChar = &(hello[7]);

//What is printed from each of the following?
printf("%s\n",hello);
printf("%s\n",ptrChar );
printf("%s\n",&(hello[7]));
printf("%s\n",hello + 7);
printf("%s\n",hello[7]); //x
```

Strings Revisited

- Since a pointer is a variable type, we can create an array of pointers just like we can create any array of any other type.
- Although the pointers may point to any type, the most common use of an array of pointers is an array of char*to create an array of strings.
- A common use of an array of pointers is to create an array of strings. The declaration below creates an initialized array of strings `(char *)` for some boy's names. This diagram illustrates the memory configuration.

```
char *names[] = { "Bobby", "Jim", "Harold" };
```



char array vs. char***

- Because the name of an array is a synonym for the address of the first element of the array, the name of the array may be treated like a pointer to the array. As a result, the name of an array of strings (for example) may be defined as either `char *[ ]` or `char** .`

- For example, the boy's name array

```
char *name[] = {
"Bobby", "Jim", Harold"
};
```

- may also be defined (almost equivalently see next slide) as

```
char **name = {
"Bobby", "Jim", Harold"
};
```

- In particular, the parameters for main may be written as either

```
int main (int argc, char *argv[ ])
```

- or

```
int main( int argc, char **argv)
```

char array vs. char***

- So, given the declaration:
  `char name [10];`
- name is a constant pointer to characters as in
  `char * const name`
  - `name++` is not allowed.
  - `name[0]++` is allowed
  - `(*name)++` is allowed.
- But as a formal parameter declaration things are different
  - `void function(char name[10])`
- Is converted to
  - `void function(char * name)`
- Really there is no such thing as an array formal parameter declaration in C.
- So on the previous slide,
  - `char *name[10]` not as a parameter produces
  - `char * * const name`
- As a parameter declaration `char *argv[ ]` produces `char ** argv`

So, what would this output look like?

```
#include <stdio.h>
#include <stdlib.h>

int main(){
  char * name[]={"Bobby","Jim","Harold" }; //may be
                          //stored in read-only memory

  printf("%s",name[1]);

  fflush(stdout); //needed to ensure output displayed
                  //  before seg fault (useful note
                  //  for projects)

  name[1][2]='r'; // here
  printf("%s",name[1]);
  return 0;
}
```

- Note: This code compiles with no errors or warnings – but, the code output would be:

```
JimSegmentation fault
```

Command Line Arguments

- Command line arguments are passed to your program as parameters to main.

```
int main( int argc, char *argv[ ] )
```

  - `Argc` is the number of command line arguments (and hence the size of argv)
  - `Argv` is an array of strings which are the command line arguments. Note that `argv[0]` is always the name of your executable program.

- For example, typing '`myprog hello world 42`' at the linux prompt results in
  - argc = 4
  - argv[0] = "`myprog`"
  - argv[1] = "`hello`"
  - argv[2] = "`world`"
  - argv[3] = "`42`"

- Note that to use `argv[3]` as an integer, you must convert if from a string to an int using the library function `atoi( )`.
- e.g.: `int age = atoi( argv[3] );`