Pointers

Used to hold memory addresses

myData₀  3000
myData₁  4000
myData₂  4200
myDataₙ  5000
Declare and initialize an integer variable

\[ \text{int } j = 3; \]

This allocates 16 bits of memory and places the value 3 (0003H) into those 16 bits.
Declare and initialize a pointer to an integer variable.

```c
int * pa;
```

Assign it the address of `j`.

```c
pa = &j;
```

`pa` now holds the address of `j` in memory.
Declare a second integer variable.

```c
int a;
```

Assign it the value of \( j \) to it.

```c
a = *pa;
```
Declare and initialize a third integer variable.

```cpp
int b = 4;
```

Assign its value to j.

```cpp
*pa = b;
```
/ *  A First Look at Pointers
 *  Playing with pointers.
 */
#include <stdio.h>

int main()
{
    int x=1, y=2, z;
    int *pa; /* pa is a pointer to int */

    pa = &x; /* pa now points to x */

    z = *pa; /* z is now 1 */
    printf ("The value of z is: \%i\n", *pa);

    *pa = y; /* x is now 2 */
    printf ("The value of x is: \%i\n", *pa);

    (*pa)++;
    printf ("The value of x is: \%i\n", *pa);

    return 0;
}
C pointers advance in increments of the size of the thing pointed to.

```c
int* intPtr = &myInt;
float* floatPtr = &myFloat;
```

Let

- `intPtr` point to memory address 3000
- `floatPtr` point to memory address 4000

- `intPtr + 1` gives 3010
  - or
  - `intPtr++` gives 3010

- `floatPtr + 1` gives 4020
  - or
  - `floatPtr++` gives 4020
Consider the following expressions:

\[
\begin{align*}
*pa &= 0003; \\
*(pa++) &= 0010; \\
*(pa + 1) &= 0010; \\
*pa + 1 &= ?
\end{align*}
\]
Pointer Arithmetic

**Cannot**
- Add Pointers
- Multiply Pointers
- Divide Pointers
- Multiply by Scalar
- Divide by Scalar

**Can**
- Subtract Pointers
- Add Scalar
Finding the midpoint in an array.......  

Know  
First Address  
Last Address  

Compute the number of bytes above and below the midpoint:  

$\frac{(\text{highAddress} - \text{lowAddress})}{2}$  

Compute the midpoint:  

$\text{lowAddress} + \frac{(\text{highAddress} - \text{lowAddress})}{2}$
Pointer Comparison

Legal Comparisons:

==, !=
Determine if the two pointers do or do not point to the same address

<, <=, >=, >
Determine if the two pointers point to higher or lower addresses
Arrays

An array is a group of consecutive memory locations that are organized to hold a collection of values of a single type.

The values are not named but are accessed by their position in the array.

**syntax**

type identifier [ dimension ]

- **type** - specifies the type of elements stored in the array
- **identifier** - names the array
- **dimension** - specifies the number of elements in the array
The declaration:

```
int a[10];
```

**Specifies:**

Sufficient space in memory is allocated to hold 10 consecutive elements.

The elements are to be integers.

The name of the array is ‘a’.
Array Representation

Array Elements
Numbered beginning at 0

Schematically

<table>
<thead>
<tr>
<th>a[0]</th>
<th>a[1]</th>
<th>a[2]</th>
<th>a[n]</th>
</tr>
</thead>
</table>

In Memory at address 3000

| a[0] | 3000 - 300F |
| a[1] | 3010 - 301F |
| a[2] | 3020 - 302F |
|      | .          |
|      | .          |
| a[9] | 3090 - 309F |
Finding an Array Element

For element a[5]

Compute

starting address + 5 * sizeof an int

3000 + 5 * 16 = 3050
Accessing an Array Element

The declaration

\[
\text{int } j = a[5];
\]

Assigns to \( j \) the value of the element of \( a[] \) indexed by 5.
Remember, this is the 6th element

The statement

\[
a[7] = i;
\]

Assigns the element of \( a[] \) indexed by 7 the value of \( i \).
Remember, this is the 8th element
Initializing Arrays

Declare a 3 element array

```c
int a[3];
```

The declaration

Allocates 6 bytes of storage.

Does *not* set the elements to any value.
An array can be initialized in two ways

1. Use a for or while loop to assign a value to each element.

2. Specify the values as part of the declaration.

Case 1:

```c
for (i=0; i<MAXSIZE; i++)
{
    a[i] = 0;
}
```

Case 2:

```c
int a[] = { 4, 5, 6 };
```
Pointers and Arrays

An array is a collection of values

\[
\begin{array}{cccc}
\end{array}
\]

Stored in memory as

\[
\begin{array}{cccc}
  3000 & & & a[0] \\
  3010 & & & a[1] \\
  3020 & & & a[2] \\
  3030 & & & a[3] \\
  \vdots & & \vdots & \vdots \\
  30n0 & & & a[n] \\
\end{array}
\]
The value of a variable of type array is the address of the first element.

Thus:

&myArray[0] ←→ myArray

To point to an array:

```c
int myArray[10]; /* declare an array */
int* myArrayPtr; /* declare an array pointer */
myArrayPtr = myArray;
```

To point to an array element:

```c
int myArray[10]; /* declare an array */
int* myArrayPtr; /* declare an array pointer */
*(myArrayPtr + 5) = 10;
```
/*
* Pointers and Arrays.
*
* Pointing to an array and to its contents
*/
#include <stdio.h>

int main()
{
    int a[] = {1,2,3,4,5};
    int *pa = a;      /* pa is a pointer to int */
    int i;

    for (i = 0; i< 5; i++)
        printf ("The value of a[i] is:  %i\n", a[i]);

    for (i = 0; i< 5; i++)
        printf ("The value of *(pa + i) is:  %i\n", *(pa+i));

    /*  *p++ increments the pointer and is equivalent to *(pa++) */
    for (i = 0; i< 5; i++)
        printf ("The value of *pa++ is:  %i\n", *pa++);

    /*  (*pa)++ increments the contents of the array entry */
    for (i = 0, pa = a+4; i< 5; i++)
        printf ("The value of *(pa)++ is:  %i\n", (*pa)++);

    return 0;
}