Pointers

Used to hold memory addresses
Declare and initialize an integer variable

```c
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.

    int a;

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

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

\[ \text{int } b = 4; \]

Assign its value to \( j \).

\[ *\text{pa} = b; \]
#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)++; /* x is now 3 */
    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:

*pa = 0003;
*(pa++) = 0010;
*(pa + 1) = 0010;
*pa + 1 = ?
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:

\[(\text{highAddress} - \text{lowAddress}) / 2\]

Compute the midpoint:

\[\text{lowAddress} + (\text{highAddress} - \text{lowAddress}) / 2\]
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:
for (i=0; i<MAXSIZE; i++)
{
    a[i] = 0;
}

Case 2:
int a[] = { 4, 5, 6 };
Pointers and Arrays

An array is a collection of values

```
a[0]  a[1]  a[2]  a[n]
```

Stored in memory as

```
3000  3010  3020  3030  ...
    |     |     |     |   ...
    |     |     |     |   ...
    |     |     |     |   ...
30n0  30n1  30n2  30n3  ...
```

```
```
Pointers and Memory Addresses

The value of a variable of type array is the address of the first element.

Thus:

\[
&\text{myArray}[0] \quad \leftrightarrow \quad \text{myArray}
\]

To point to an array:

\[
\begin{align*}
\text{int myArray}[10]; & \quad /* \text{declare an array */ } \\
\text{int* myArrayPtr; } & \quad /* \text{declare an array pointer */ } \\
\text{myArrayPtr} = \text{myArray};
\end{align*}
\]

To point to an array element:

\[
\begin{align*}
\text{int myArray}[10]; & \quad /* \text{declare an array */ } \\
\text{int* myArrayPtr; } & \quad /* \text{declare an array pointer */ } \\
*(\text{myArrayPtr} + 5) = 10;
\end{align*}
\]
#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;
}
More Pointers

Will now examine several advanced pointer types

Generic Pointer

NULL Pointer

Function Pointer
Generic Pointers

A generic pointer is designed to be able to point to any type object.

**syntax**

```c
void * pointer-name
```

**Generic Pointers**

Guaranteed to be large enough to hold a pointer to any object except a function type

Cannot be dereferenced with the * or subscripting operators.

Must be converted to pointer of the appropriate type before dereferencing.
Dereferencing

Dereference as a generic pointer

```c
void* myGenericPtr;
int myValue = 3;
myGenericPtr = (void*)&myValue;
int tempValue = *myGenericPtr;       // illegal
```

Proper dereference

```c
void* myGenericPtr;
int myValue = 3;
myGenericPtr = (void*) &myValue;

// side effect cast - don’t use
myIntPtr = myGenericPtr;

// explicit cast
myIntPtr = (int*)myGenericPtr;
int tempValue = *(int*)myGenericPtr;    // ok

// or
int tempValue = *(int*)myGenericPtr;
```
Null Pointer

ANSI C defines a special pointer whose value points to no object or function.

\textbf{value}

\begin{verbatim}
(void *) 0
\end{verbatim}

Defined as a macro in the header file:

\texttt{<stdlib.h>
Function Pointers

C pointers may point to functions as well as data.

**syntax**

\[
\text{type (} \ast \text{ functionPointer)(<arg}_0, \text{ arg}_1, \ldots\text{arg}_n>)
\]

the arg list may be empty
int (*intFunctPtr)();

Read as:
    intFunctPtr is
    a pointer to a function
    taking no args
    returning an int

double (*doubleFunctPtr)(int, char);

Read as:
    doubleFunctPtr is
    a pointer to a function
    taking two args: an int and a char
    returning a double
Pointing to a Function

A value may be assigned to a pointer to a function in several ways

```c
int (* intFunctPtr) ();
int myFunction();

intFunctPtr = &myFunction;
intFunctPtr = myFunction;
```
Dereferencing a Pointer to a Function

A pointer to a function may be dereferenced in several ways:

\[
\begin{align*}
(*\text{functPtr})(\text{argument-list}); \\
\text{functPtr}(\text{argument-list}); \\
\text{int }(*\text{intFunctPtr})(); \\
\text{double }(*\text{doubleFunctPtr})(\text{int, char}); \\
\text{int myFunction}(); \\
\text{double yourFunction}([\text{int, char}]); \\
\text{intFunctPtr} = \text{myFunction}(); \\
\text{doubleFunctPtr} = \text{yourFunction}(); \\
(*\text{intFunctPtr})(); \\
(*\text{doubleFunctPtr})(\text{arg}_0, \text{arg}_1);
\end{align*}
\]
/*
* Pointers to Functions
*/
#include <stdio.h>
#include <stdlib.h>

int add(int a1, int a2);
int sub(int a1, int a2);

/*
* myFunction has a single parameter, a pointer to a function taking 2 ints
* as arguments and returning an int.
*/
int myFunction (int (*fPtr)(int, int));

int main()
{
    int sum, difference;
    /*
    * Declare fPtr as a pointer to a function taking 2 ints as arguments and
    * returning an int
    */
    int (*fPtr)(int a1, int a2);
    fPtr = add;        /* fPtr points to the function add */
    sum = fPtr(2,3);    /* dereference fPtr */
    printf ("The sum is: %d\n", sum);

    sum = (*fPtr)(2,4); /* dereference fPtr */
    printf ("The sum is: %d\n", sum);

    fPtr = &sub;        /* fPtr points to the function sub */
    difference = myFunction(fPtr); /* pass fPtr to myFunction() */
    printf ("The difference is: %d\n", difference);
    return 0;
}
int myFunction (int (*fPtr)(int a1, int a2))
{
    return ((*fPtr)(4,5));
}

int add(int a1, int a2)
{
    return (a1+a2);
}

int sub(int a1, int a2)
{
    return (a1-a2);
}
Indirect Reference

In addition to an address of a variable, a pointer can contain the address of another pointer.

To retrieve a value, we must apply the dereference operator twice.

**syntax**

```
type ** pointerName;
```
int **ptr;
int myInt = 3;

Let

&myInt be memory address 3000
address of myInt be stored at memory address 2000

*ptr point to memory address 2000
int myValue = **ptr;
int myValue = 4;

**ptr = myValue;
Multidimensional Arrays

If a [ ] is an array of ints,

Represented as:

```
| a[0] | a[1] | a[2] | a[n] |
```

Stored in memory as:

```
3000  a[0]   
3010  a[1]   
3020  a[2]   
3030  a[3]   
.     .      
.     .      
.     .      
30n0  a[n]   
```
Two Dimensional Arrays

**syntax**

```
    type arrayName [ row-size ] [ column-size ]
```

```
    int year [ 12 ] [ 31 ];
    float budget [ 4 ] [ 5 ];
    char* weekDays [ ] = {"mon", "tues", "wed", "thurs", "fri"};
```
A two dimensional array is implemented as an array of pointers to arrays.
Addresses in a Two Dimensional Array

Declare an array:

```c
int b[2][3];
```

The elements are stored as:

```c
b[0][0], b[0][1], b[0][2], b[1][0], b[1][1], b[1][2]
```

Element b[1][2] is addressed as:

`*(b + 1) + 2`

1. b is a 2 by 3 array. b points to the first element of the first 3 element subarray.
2. b + 1 points to the second 3 element subarray.
3. *(b + 1) points to the first element of the second 3 element subarray.
4. *(b + 1) + 2 points to the third element of the second subarray.
5. `*(b + 1) + 2` is the third element of the second subarray.
Pointers and Memory Addresses

```plaintext
\[ a + 1 \]

\[ *(a + 1) + 2 \]

\[ a_0[0, 1, 2] \]

\[ a_1[0, 1, 2] \]

\[ *(a + 1) \]
```
Multidimensional Arrays

C places no restrictions on the number of dimensions an array may have.

Multidimensional arrays are always stored in row major order - Elements differing only in the last subscript are stored adjacently.

Addressing is a simple extension of that for two dimensional arrays. An expression of the form $i \times j \times k$ is converted into the form `pointer to j \times k`. 
Allocating Storage

To allocate storage for an array, its size must be known.

There are several cases when an incomplete or no specification may be permitted.

Case1: External Declared

extern int a [];

Case2: Single dimensioned array that is a function parameter

int a [ 10 ];
myFunct ( a );
int myFunct ( int a [ ] ) { . . . }

Case 3: Multidimensional array that is a function parameter.

int a [ 2 ] [ 3 ], b [ 2 ] [ 3 ][ 4 ];
myFunct ( a );
yourFunct ( b );

int myFunct ( int a [ ] [ 3 ] ) { . . . }
int yourFunct ( int b [ ] [ 3 ][ 4 ] ) { . . . }