**Pointers and Pointer Management**

**Overview**

In this lesson will address very important issue

Addresses in memory

Explore in some depth

Variable addresses

How to work with such addresses

The pointer variable

How to work with pointers

Operations on and with pointers

Examine two special pointer types

Generic pointer

Null pointer

Examine pass by reference and pass by value

When passing variables into function

Examine passing arrays into functions

**Pointers**

**Variables Storing Data**

Variables one is most familiar with hold data

We have such declarations as

```c
int myAge = 39;
float mySpeed = 54.52;
char myAnswer = 't';
```

Know data stored in memory somewhere

Questions…. 

- How does one know where data is stored
- Are large data objects stored the same way as smaller
- How do I pass a large data object to a function
- Can I put a 200 M byte object on the stack
- Do I want to

Let’s see how to begin to answer these questions

Remember each variable we’re familiar with is holding data

That data is stored in memory

In particular at some memory location
We see data stored at several locations  
   In accompanying figure

Each memory location has an address  
   For each piece of data  
       Have a value  
       Have the place that it is stored - its address

When talk about data  
   Talk about value  
   Talk about where it is stored

When talk about variable  
   Talk about value  
   Talk about where it is stored

**Variables Storing Addresses**
   In C *value* of a variable can be a memory address  
   Give such a variable a special name  
       Call *pointer*  
       It is pointing – like a street sign – to some variable

Thus  
   Pointer is a variable  
       Exactly like the variables we’ve been studying  
       Pointers hold values that are addresses in memory

*Caution:*  
   A street sign can sometimes be wrong  
       Directing you in an incorrect direction

Any variable has a value  
   C does not support vacuum variables  
   Since pointer is a variable  
       It always has a value  
       The *value* is interpreted as an *address*
That does not mean that the address is valid
Referring to a valid piece of data

Consider standard C definition and assignment

```c
int myData0 = 3;
```

Such declaration achieves the following

- Allocates 32 bits of memory
  - Assuming 32 bit integers
- Place decimal value 3 (0003) into those 32 bits
- Associates
  - Address in memory where data stored
    - Assume address 3000
    - With symbol myData0

When write myData0

- Actually referring to data at location 3000

If wish to know where data is stored
- That is its address in memory
- Use `address of` operator `&`
  - That is
    ```c
    &myData0 (read address of myData0) is 3000
    Place where data is stored
    ```

We see then

If myData0Ptr is variable that holds address of variable
- Expression
  ```c
  myData0Ptr = &myData0;  // &myData0 evaluates to 3000
  ```
- Sets `value of` myData0Ptr to
  - Address of the variable myData0
  - Point to the variable myData0

Since myData0 is stored at 3000

Variable myData0Ptr
- Will contain value 3000
In accompanying figure
myData0Ptr stored at location CD00

We can retrieve value from place in memory
myData0Ptr points to

Use dereference operator * as prefix to identifier
Write
    *myData0Ptr
To indicate
Value at address contained in myData0Ptr
Value at address myData0Ptr refers to

From above
Code fragment
    int myData1;       // assume myData1 stored in location 5A00
    myData1 = *myData0Ptr;
Assigns the value 3 to variable myData1
Since `myData0Ptr` is pointing to memory location 3000 containing 0003

Similarly

```c
int myData3 = 0x4;
* myData0Ptr = myData3;
```

Places the value 0x4 into the memory location 3000
Let’s see how these work in real program

Example

pointers0.c

/*@*/
#include <stdio.h>
int main(void)
{
  int x=1, y=2, z;
  int *aPtr;       // aPtr is a pointer to int
  aPtr = &x;       // aPtr now points to x

  z = *aPtr;       // z is now 1
  printf( "The value of z is: %d \n" *aPtr);

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

  (*aPtr)++;        // x is now 3
  printf( "The value of x is: %d \n" *aPtr);

  *aPtr++;         // x is now ???
  printf( "The value of x is: %d \n" *aPtr);

  return 0;
}

This program will print

The value of z is: 1
The value of x is: 2
The value of x is: 3
The value of x is: hex number

Last value of x may or may not be different each time program is executed

Why?
Observe

    Had to write
    (*myPtr)++

What do we get with

    *myPtr++

Let myPtr point to address 3000
Since ++ is
    Left associative
    Higher precedence than *

Same as writing

    *(myPtr++)

Thus

1. myPtr ← 3010
2. Contents of location 3010 retrieved and discarded

By writing

    (*myPtr)++
    Get what myPtr is pointing at
    Then increment that value

Why does

    myPtr++ yield 3010 rather than 3001

In C

    Pointer advanced in increments
    Size of element pointed to

    myPtr declared as pointer to int
    Therefore incremented in sizeof (int)

    If int 16 bits
    3000 + 1 = 3010
If myPtr had been declared as

```c
float* myPtr;
```

Then

```c
With (myPtr) = 3000
```

```c
myPtr++
```

Gives 3020

For float = 32 bits

**Pointers and Numbers**

Although pointers are integral types

Cannot assign an integer type variable to a pointer

Let’s look at following code fragment

```c
int myInt = 10; // declare an int
int* intPtr;    // declare a pointer to an int
float myFloat = 1.25f; // declare a float

intPtr = myInt0;  // intPtr is a pointer and must contain an address
myInt0  = intPtr; // myInt0 is an integer and cannot contain an address
intPtr = myFloat; // intPtr pointer must contain address cannot contain float
intPtr = &myFloat; // intPtr is a pointer to int cannot contain address of float
```

Compiler is assuring that contents of pointer

Is the address of a variable of the pointer's type

Making sure that types are used in

Safe and consistent manner

Called *type safety*
Pointers and `printf()`

When we display a pointer using `printf`  
  What is displayed is the address contained in the pointer  
  Same as when we display an int  
  We see the value contained in the int

Observe the following code fragments

```
int data  =  10;  // declare an int
int* dataPtr = &data;  // declare a pointer to it

printf("%d ", data);  // displays 10
printf("%d ", dataPtr);  // displays the contents of variable dataPtr
                        // which is the address of the variable data
printf("%d ", *dataPtr);  // displays the contents of the variable data
```

Simple Pointer Arithmetic

Explore example a bit closer

Return to ints for the moment

Question

```
if  myPtr = 3000 and
    myPtr++ = 3010
what does
    myPtr + 1 = ?
```

Once again compiler knows  
  Type of variable pointed to

Thus

  `myPtr + 1` is no different from `myPtr++`

  As we would expect  
  Since `myPtr++` says to add 1 to `myPtr`

Consequently

  `myPtr++ ⇔ myPtr + 1`
Caution:
If we have accompanying situation
With myPtr = 0003

We know
*myPtr = 0003
*(myPtr++) = 0010
*(myPtr + 1) = 0010

What does *myPtr + 1 yield
The pointer will be evaluated first
This returns 0003
Then 1 is added to 0003
The result is 0004

If this was intended
OK
Else
Need to use parentheses

Using simple pointer arithmetic
Have seen the behavior of the ++ and -- operators
Have noted such arithmetic
Done in units of the underlying base type
If ptr points to an int
ptr + 1 points to address at
&ptr + sizeof (int)

Are other arithmetic operations legal
Yes
First must state what cannot be done

*Cannot* perform following operations on pointers

- Add Pointers
- Multiply Pointers
- Divide Pointers
- Multiply or Divide By scalar

*Can* perform following operations on pointers

- Subtract 2 pointers
  - Result is an int
  - Value of the is is given by
    \[(\text{address}_1 - \text{address}_0) / \text{sizeof (type)}\]

- Add or subtract Pointer and scalar
  - Have seen that
    - \(\text{pointer} + \text{int}\)
    - Is a legal operation
    - Yields a pointer
  - Subtraction similar

Therefore can indirectly implement

- Pointer addition
- Division by scalar

**Example**

- Want to do binary search on array
  - Need to find mid point of array

**Know**

- First address of array
- Last address of array

- Address of midpoint computed as follows
  - Number of bytes above and below mid point given as
Thus midpoint is given as

\[
\frac{\text{highAddress} - \text{lowAddress}}{2}
\]

Which is the address we had hoped find by

\[
\text{lowAddress} + \frac{\text{highAddress} - \text{lowAddress}}{2}
\]

Which is not an address but an integer type

**Pointer Comparison**

Can compare pointers

In reality comparing addresses

Only meaningful if

Comparing addresses in the same address space

Legal comparisons

\[
==, \neq, \leq, \geq, >
\]

The two addresses

Point to or do not point to the same address

\[
<, \leq, \geq, >
\]

The two addresses

Point to higher or lower addresses

**Generic and NULL Pointers**

Pointers dealt familiar with

Point to a particular type

Often would like to use same pointer

To point to any data type

In C== will see this is a simple form of polymorphism

ANSI C introduces

*Generic Pointer*
Generic Pointer

Pointer to type void

`void *`

Traditionally done using pointer to char

Cast to proper type

Before dereferencing

Has some potential problems

No way for compiler to ensure proper cast taken place

`void *` pointer guaranteed to be large enough to hold pointer

To any type of object

Thus Pointer to any object

Except function type

Converted to type `void *` and back again

No loss of information

Dereferencing

Generic pointer

Cannot be dereferenced with these operators

`*`

subscript

On many machines

Pointers the same size

Not true on all machines

Must be aware of this to ensure max portability

Because of potentially different sizes

`void *` pointer

Must converted back to pointer of appropriate type before dereferencing

Can be accomplished by assigning to

Variable declared as intended pointer type
Example

```c
void * myGenericPtr;
int myValue = 3;

// side effect cast – bad style
myGenericPtr = &myValue;
int tempValue = *myGenericPtr; // illegal
```

Example

Can dereference as:

```c
void * myGenericPtr;
int * myIntPtr;
int myIntValue = 3;
myGenericPtr = &myIntValue;

myIntPtr = myGenericPtr;
int tempValue = *myIntPtr;
```

Method in last example uses two side effect casts
- Pointer to int converted to pointer void
  - By assigning to int* pointer to void* pointer
- void* pointer later converted to pointer to int
  - By assigning to int* pointer

As discussed earlier
- Side effect casts are bad coding style

Example

Better dereference as:

```c
void * myGenericPtr;
int myIntValue = 3;

// make the cast explicit
myGenericPtr = (void*) &myIntValue;

// make the cast explicit then dereference
int tempValue = *(int*)myIntPtr;
```
Consider following example

```c
Example
void * myGenericPtr;
int myIntValue = 3;

myGenericPtr = (void*) &myIntValue;

float tempValue = *(float*)myIntPtr;
```

**Caution:**
The generic pointer
- Contains address of an int type variable
- Cast states address is that of float type variable
- Dereference retrieves value (number of bits)
- To form a float
- Clearly this is an error

Compiler may not warn of error

**Null Pointer**
As we’ve learned
- Value of pointer is address of some variable
Unfortunately
- Number 0 is a value
- Some random number is a value

Dereferencing pointer
- That has not been initialized to or assigned valid value
- Yields undetermined results
- That has a value of 0
- Illegal since no variables have address of 0

**Good Style**
- In general casting can lead to unanticipated side effects and should be avoided
- If you must cause a cast – make it explicit
Uninitialized pointers when declared
   Contain whatever value is in memory at time
Dereferencing
   Treats value as an address
   No idea what address pointing to
      Could refer to another part of the program
      Operating system
      etc.

Pointers
   Initialized to or assigned value of 0
Dereferencing
   Usually gives runtime error
   Program termination

ANSI C offers solution
   Defines special pointer
      null pointer
   Value points to no object or function

Value
   (void *) 0

Special macro NULL
   Defined in header file <stddef.h>
   Define NULL ((void *) 0 )

**Good Style**
- Ensure all pointers have value NULL when not assigned to object or function
- Must still assign valid value before dereferencing
Functions and Pointers

Now examine

Pointer type arguments to functions
Returning value from function using pointer

Before we start need to introduce two concepts

• Pass by value
• Pass by reference

Pass by Value

In earlier discussion of functions

Learned that when variable passed into function

Copy of value of variable mace
That copy stored on stack
Where function can see it

We say that the variable was passed into function by value
If we make changes to that value
Such changes not reflected
In corresponding changes to original variable
They are only changes to the copy

Several possible problems with such an approach

• May want function to be able to change value of original variable
• Copying complex variable may take long time
• Original variable may be very big
  Stack is of limited size
  May overflow the stack

Pass by Reference

Possible solution

As have learned – pointer contains address of variable

• If copy of value of address of variable rather than copy of value of variable passed
  Can go directly to address and change value of original
• Copying value of address faster than copying complex variable
• Address of variable typically size of integer
  Doesn’t take up much room on stack
Passing copy of address of variable rather than copy of variable
  Called *pass by reference*

Caution:
Passing variable address rather than copy of variable value
  Still a copy
Changing value of address – pointer value – in function
  Only changes value of copy
Value of original address unchanged

*Example*
pointer1.c

```c
#include <stdio.h>

// Function prototype
int myFunction0(int* myPtr);

int main(void)
{
    // define a variable and a pointer to the variable
    int a = 10;
    int *aPtr = &a;

    // pass the pointer into the function
    myFunction0(aPtr);
    return 0;
}

int myFunction0(int* myPtr)
{
    // dereference the pointer and print value of variable
    printf("The value of the variable a is: \%i\n", *myPtr);
    return 0;
}
```

Now consider following code fragment

*Example*

`pointer2.c`

```c
#include <stdio.h>
#include <stddef.h>

// Function prototype
int* myFunction0(int* myPtr);

int main()
{
    // define a variable and a pointer to the variable
    int a = 10;
    int *aPtr = &a;
    int *retPtr = NULL;

    // pass the pointer into the function and get return value
    retPtr = myFunction0(aPtr);
    printf("The value of the returned variable is: %i\n", *retPtr);
    return 0;
}

int* myFunction0(int* myPtr)
{
    int retVal = 20;
    int *retPtr = &retVal;
    // dereference the pointer and print value of variable
    printf("The value of the variable a is: %i\n", *myPtr);
    return retPtr;
}
```

Printed result will not be what was expected

In `myFunction()`:

Variable `retVal` is local variable is auto variable
- Exists
  - Only on stack
  - While in scope of `myFunction()`
- Once flow of control returns to `main()` function
  - `retVal` popped off stack and is gone

*Caution*

Never try to return pointer to local variable
Arrays and Functions

Often need to pass array to function

However array can be very large container

Do not want to copy such a structure onto stack

In fact

All variables in C passed by value – array is only type passed by reference

Recall from discussion of arrays

Name of array is address of 0th element

There is a reason for this

When we pass array to function as parameter

We pass

• Name of array
• Number of elements in array

Consider following code fragment

Example
arrayPtr0.c

#include <stdio.h>

// function prototypes
void displayArray (int a[], int numElements);
void incArray (int a[], int numElements);
int main(void)
{
    int a[] = {1,2,3,4,5};
    int i;
    // display the array
    displayArray (a, 5);
    // increment each element in the array
    incArray (a, 5);
    // display the array after each element has been incremented
    for (i = 0; i < 5; i++)
    {
        printf ("Element I is: %i\n", a[i]);
    }
    return 0;
}
void displayArray (int a[], int numElements)
{
    int i;
    for (i = 0; i < numElements; i++)
    {
        printf ("Element i is:  %i\n", a[i]);
    }
    return;
}

void incArray (int a[], int numElements)
{
    int i;
    for (i = 0; i < numElements; i++)
    {
        a[i]++;
    }
    return;
}

Summary and Review of Objectives
You should now be beginning to get comfortable with
• Variable addresses in memory and how to work with such addresses
• The pointer variable
  How to work with pointers
  Operations on and with pointers
• Passing pointers as function arguments
• Not returning pointers to local variables
• Special pointer types
  Generic pointer
  Null pointer
• Pass by value and pass by reference
• Passing arrays to functions