Pointers
Gaddis 9.1 - 9.7
CS 2308 :: Spring 2016
Molly O'Neil
Some Caution from THE C Book

- "Pointers have been lumped with the `goto` statement as a marvelous way to create impossible-to-understand programs. This is certainly true when they are used carelessly, and it is easy to create pointers that point somewhere unexpected." - K & R

- C/C++ has very powerful and flexible (read: *dangerous!*) pointers compared to other languages
However...

• "With discipline, however, pointers can also be used to achieve clarity and simplicity."
  - K & R again

• Your best defense against pointer mistakes is to really understand pointers!
  • (Also, never be afraid to draw pictures. Pictures really help.)
Variables

• Recall that a variable is nothing more than a convenient name for a storage location

• Variable's scope defines when/if that particular memory location can be re-used
  • I.e., two different variables in different scopes may end up re-using the same physical storage location

• Variable's type defines how large the storage location needs to be, how the bits inside that memory location are interpreted
  • E.g., a float and a double are storage locations capable of holding different numbers of bits
  • E.g., an int and a float might both be 32 bits of 1s and 0s, but interpreted differently
Variables

• **A variable is a storage location**

• In addition to possibly having an associated *identifier* (i.e., *name*)...
  • Every variable has a *value*
  • Every variable has an *address*
    • A 32-bit machine has $2^{32} \approx 4$ billion (byte-sized) memory locations, and therefore 4 billion addresses
    • Compiler maps each *variable* in your program to a particular *memory address*, allocates a block of memory at that address big enough to fit the variable's type

```
int myVar = 10;
```

<table>
<thead>
<tr>
<th>myVar</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td></td>
</tr>
<tr>
<td>0x1004</td>
<td></td>
</tr>
<tr>
<td>0x1008</td>
<td></td>
</tr>
<tr>
<td>0x100C</td>
<td></td>
</tr>
</tbody>
</table>
The Address Operator (&)

• Each variable in a program is stored at a unique address
• The address operator, &, returns the address of a variable
  • Apply operator to front of variable name

```cpp
int x = 100;
cout << x << endl;
cout << &x << endl;
```

Output:

```
100
0xbfffffb0c
```

Addresses printed in hexadecimal
Pointer Variables

• A **pointer variable**, or **pointer**, is a variable that holds an address of a memory location
  
  • It's really just a variable, e.g., 32 bits of storage
  
  • Its value will be interpreted as an address
    
    • (This is just like 32 bits of 1s and 0s can be interpreted as an int, or a float, or 4 chars...)
  
  • Because a pointer variable holds the address of another piece of data (i.e., another variable), it "points" to the other variable

```c
int myVar = 10;
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>myVar</td>
<td>10</td>
<td>0x1004</td>
</tr>
<tr>
<td>myPtr</td>
<td>0x1004</td>
<td></td>
</tr>
</tbody>
</table>
```

Pointer Variables

• An asterisk, *, is used to define a pointer variable:

```c
int *ptr;
```

• "ptr is a pointer to an int", or
• "ptr can hold the address of an int"

• The following syntax is OK too (note spacing difference):

```c
int * ptr;  // same as above
int* ptr;   // same as above
```

For this class: Pick any of the 3 styles on this page, but you must pick ONE and BE CONSISTENT!
Using Pointer Variables

- Assigning an address to a pointer variable:

```cpp
int x = 100;
int *ptr;

ptr = &x;

cout << x << endl;
cout << ptr << endl;
```

Output:

```
100
0xbfffff0c
```

- Memory layout:

```
ptr 0xbfffff0c
x 100
```

```
0xbfffff0c
```
Using Pointer Variables (Another Example)

```c
int myVar = 10;
int *myPtr;
myPtr = &myVar;
```
De-Referencing a Pointer

- The unary * operator (*not related to the binary multiplication operator!*) is the *indirection operator or de-referencing operator*
  - Allows you to access the variable that a pointer points to
  - Applying the * operator is called *de-referencing* the pointer
  - *ptr* is an alias for the variable whose address *ptr* holds
    - Can be used to read the value at the variable, or change it

- If p is a pointer variable that holds the address of x, an int variable, we can now access x in two ways:
  - Using the identifier/name: x = 42; cout << x;
  - De-referencing the pointer: *p = 42; cout << *p;
  - *Both mean exactly the same thing!*
Pointer Declaration vs. Dereference

• The asterisk * is used in 2 different contexts for pointers, with two completely different meanings!

• To declare a pointer (variable declaration/definition):

```c
int *ip;  // ip is defined to be a variable of type
          // "pointer to an int"
```

• To dereference a pointer (in an expression):

```c
int y = *ip;  // y is assigned the value of the int variable
               // that ip points to
```
De-Referencing Example

```c
int x = 1;
int y = 2;
int *ptr;

ptr = &x;    // ptr is a pointer to x

cout << x << endl;
cout << *ptr << endl;

y = *ptr;
*ptr = 3;

cout << x << endl;
cout << y << endl;
```

Output:
```
0xbffffff0c
1
1
3
1
0xbffffff0c
```

Diagram:
- `ptr`: Address 0xbfffffff0c
- `x`: Address 0xbffffff0c, Value 1
- `y`: Address 0xbffffffc28, Value 2
- `*ptr`: Value 1
- `*ptr = 3`: Value updated to 3
Pointer Reassignment

- The same pointer can "point to" one variable, then be reassigned to point to another variable
  - Just like assigning one value to `int x`, then assigning a different value to `x`
    - (Because remember, a pointer variable is just a storage location that holds a value that will be interpreted as an address)

```c
int x = 10, y = 0;
int *p;

p = &x;       // p points to x
x = x + *p;   // doubles x (equivalent to x = x + x)

p = &y;       // now p points to y
*p = 42;      // y is given value 42; x is unchanged!
```

![Diagram showing pointer reassignment]
Another Example

```c++
int x = 25, y = 50, z = 75;
int *p;

p = &x;
*p = *p + 100;

p = &y;
*p = *p + 100;

p = &z;
*p = *p + 100;

cout << x << " " << y << " " << z << endl;
```

• What does this output?

125  150  175
Test Your Understanding...

• When does this loop terminate?

```cpp
int x = 0;
int *p = &x;

while(*p == x) {
    cout << x << endl;
    *p = *p + 1;
}
```

**NEVER!!!**

*p is an alias for x
(We never change the value of p, i.e., we never redirect the pointer)
Characteristics of Pointers

• Just like any other variable...
  • Pointers can be *initialized and assigned*
  • Pointers can be *local or global variables*
  • Pointers can be *function parameters*
  • You can have an *array of pointers*

• You can use *arithmetic* (+, −, etc.) and *comparison* (<, >, ≤, etc.) *operators* on pointers (more on this later...)

Initializing Pointers

• Like other variable types, pointers can be initialized when they are defined:

```c
int x;
int *p;
p = &x;
```

```
int x;
int *p = &x;
```

NOTE: You are initializing the pointer, NOT the variable the pointer points to!

• Pointers to a type can be declared on the same line as variables of the type:

```c
int num, *numPtr = &num, x = 0;
```

• You cannot mix data types:

```c
double cost;
int *ptr = &cost; // NOT OK - WILL NOT COMPILE
```
Null Pointers

• Address zero is special
  • No variable will ever be located at address zero
  • It is never legal to store a value into the memory location at address zero (results in a segmentation fault)

• Because of the above guarantees, programmers can use address zero to indicate a "pointer to nothing"
  • This is called a null pointer

```c
int *p;
p = NULL;      // must #include <cstdlib>
p = 0;         // equivalent to the above
// (note that this is the only integer that the compiler will let you assign to a pointer without a cast)
p = nullptr;   // preferred C++ style, equivalent to above
```
When Would You Use a Null Pointer?

• Imagine writing a function that returns a pointer to the first occurrence of the letter 'z' in an array of characters (one way to represent a string)
  • What should you return if there are no 'z' chars in the string?

• To initialize a pointer if you don't know a real value for it at declaration time
  • Then you can test for an invalid address with:
    ```c
    if(!ptr) {
        ...
    }
    ```

• Sometimes a segmentation fault isn't the worst thing that can happen...
Pointers as Function Parameters

- Let's revive an example from the CS 1428 review lecture about "pass by reference" using reference parameters...

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

void zeroMyValue(int &);

int main() {
    int myValue = 42;
    cout << "Inside main: myValue = " << myValue << endl;

    zeroMyValue(myValue);

    cout << "Back inside main: myValue = " << myValue << endl;
    return 0;
}

void zeroMyValue(int &val) {
    val = 0;
    cout << "Inside zeroMyValue: val = " << val << endl;
}
```
Pointers as Function Parameters

• We can implement pass-by-reference using pointers instead:

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

void zeroMyValue(int *);

int main() {
    int myValue = 42;
    cout << "Inside main: myValue = " << myValue << endl;

    zeroMyValue(&myValue);

    cout << "Back inside main: myValue = " << myValue << endl;
    return 0;
}

void zeroMyValue(int *val) {
    *val = 0;
    cout << "Inside zeroMyValue: val = " << *val << endl;
}
```
Pointers & Array Names

- You can use the *name of an array* as if it were a **pointer to the first element of the array** (in other words, the address of the first element of the array).

```cpp
int numbers[] = { 10, 20, 30, 40 };

cout << "first element value: " << numbers[0] << endl;
cout << "first element value: " << *numbers << endl;
cout << "addr of 1st elem: " << &numbers[0] << endl;
cout << "addr of 1st elem: " << numbers << endl;
```

Output:
```
first element value: 10
first element value: 10
addr of 1st elem: 0xbfffffb04
addr of 1st elem: 0xbfffffb04
```
Array Names as Function Parameters

- We've actually seen this before...
- Remember how we learned in CS 1428 that "arrays are automatically passed as references when you pass the array name as a function argument"?

```c++
double totalSales(double arr[], int size) {
    double sum = 0.0;
    for(int i = 0; i < size; i++) {
        sum += arr[i];
    }
    return sum;
}

int main() {
    double sales[NUM];
    // put some data in the sales array here
    cout << "Total sales: " << totalSales(sales, NUM) << endl;
}
```

We pass the array name as the argument: this name is a pointer to the first element of the array!
Array Names as Function Parameters

• The below is exactly equivalent:

```c
double totalSales(double *arr, int size) {
    double sum = 0.0;
    for(int i = 0; i < size; i++) {
        sum += arr[i];
    }
    return sum;
}
```

```c
int main() {
    double sales[NUM];
    // put some data in the sales array here
    cout << "Total sales: " << totalSales(sales, NUM) << endl;
}
```

• What changed?

  • Syntactically: changed `double arr[]` → `double *arr` in function parameter list

  • Meaningfully: **absolutely nothing!**
Pointers & Array Names

- Pointer can be used as an array name, including with subscript operations:

```c++
int list[] = { 4, 7, 11 };
int *listptr = list;
cout << listptr[1]; // displays 7
```

- Array name can be used as a pointer, including with arithmetic operations:

```c++
int list[] = { 4, 7, 11 };
cout << *(list + 1); // displays 7 (stay tuned for why)
```

- Only difference: an array name is a pointer constant, i.e., you cannot change its value (you cannot point it at some other variable/location instead)

```c++
int list[10];
int *iPtr;

iPtr = list; // OK
list = iPtr; // NOT OK - WILL NOT COMPILE
```
Pointer Arithmetic

- Every data type has a size
  - E.g., on many systems an `int` is 32 bits (or 4 bytes)
- When pointer arithmetic is performed, the computation result is computed based on the size of the variable type that the pointer points at
  - E.g., if you *increment* (*add one to*) an `int *`, you are adding 4 bytes to the address stored in the `int *`

Assume `sizeof(int)` is 4
`numbers` is stored at `0xbffffffb04`

```
| numbers[0] | 10 | 0xbffffffb04 |
| numbers[1] | 20 | 0xbffffffb08 |
| numbers[2] | 30 | 0xbffffffb0c |
| numbers[3] | 40 | 0xbffffffb10 |
```

```
Assume sizeof(int) is 4
numbers is stored at 0xbffffffb04

numbers+1 is really 0xbffffffb04 + (1*4)
numbers+2 is really 0xbffffffb04 + (2*4)
numbers+3 is really 0xbffffffb04 + (3*4)
```
Array Access with Pointers

• Conversion: `array[i]` equivalent to `*(array + i)`

• Array elements can therefore be accessed in many ways...

  • Array name and `[ ]`  
    ```c
    int list[10], *ptr;
    ptr = list;
    list[2] = 10;
    ```

  • Pointer to array and `[ ]`  
    ```c
    ptr[2] = 10;
    ```

  • Array name and pointer arithmetic  
    ```c
    *(list + 2) = 10;
    ```

  • Pointer to array and pointer arithmetic  
    ```c
    *(ptr + 2) = 10;
    ```

• No bounds checking! (Regardless of whether you're using array name or pointer)
Pointer Arithmetic Example

• Note that de-reference operator (*) has higher precedence than arithmetic ops, so parenthesis required!

```cpp
int numbers[] = { 10, 20, 30, 40 };

cout << "second element value: " << numbers[1] << endl;
cout << "second element value: " << *(numbers + 1) << endl;

cout << "size: " << sizeof(int) << endl;
cout << "addr of 1st elem: " << numbers << endl;
cout << "addr of 2nd elem: " << numbers + 1 << endl;
```

**Output:**

```
second element value: 20
second element value: 20
size: 4
addr of 1st elem: 0xbfffb04
addr of 2nd elem: 0xbfffb08
```
Pointer Arithmetic Operations

• Supported arithmetic operations on pointer variables of data type \( d \):

• \( \text{ptr} + n \) and \( \text{ptr} - n \) (where \( n \) is an \text{int})
  • Adds (or subtracts) \( n \times \text{sizeof}(d) \) to(/from) \( \text{ptr} \)

• \( \text{ptr}++ \) and \( \text{ptr}-- \)
  • Increments/decrements \( \text{ptr} \) to next/prev variable of type \( d \)

• \( \text{ptr} += n \) and \( \text{ptr} -= n \) (where \( n \) is an \text{int})
  • Adds (or subtracts) \( n \times \text{sizeof}(d) \) to(/from) \( \text{ptr} \) and stores result in \( \text{ptr} \)

• \( \text{ptrA} - \text{ptrB} \) where result is an \text{int}
  • Result is the # of values of type \( d \) between the two memory addresses

\[ d \ast \text{ptr}; \]
Revisiting an Earlier Example...

```c
double totalSales(double *arr, int size) {
    double sum = 0.0;
    for(int i = 0; i < size; i++) {
        sum += *(arr++);       // used to be arr[i]
    }
    return sum;
}

int main() {
    double sales[NUM];
    // put some data in the sales array here
    cout << "Total sales: " << totalSales(sales, NUM) << endl;
}
```

- **What?!** `sum += *(arr++)` ??
- Equivalent to: `sum = sum + *arr;`  
  `arr = arr + 1;`
Pointer Comparison Operations

- Pointers can also be compared using the relational operators: `<  <=  >  >=  ==  !=
- Comparison is based on the address values
- Like the arithmetic operators, the less-than/greater-than operators really only make sense in regards to arrays

```cpp
int arr[25];
cout << (&arr[1] > &arr[0]) << endl;
cout << (arr == &arr[0]) << endl;
cout << (arr <= &arr[20]) << endl;
cout << (arr > arr + 5) << endl;
```

- Pointer comparison is NOT based on the contents of the variables pointed at!

```cpp
double *d1, *d2;
d1 < d2;       // compares pointers
*d1 < *d2;     // compares contents
```
Keys to Understanding Pointers

- **Key #1:** Understand that *variables are names for memory locations*

- **Key #2:** Understand what it means to dereference something
  - If \( p \) is *a pointer to (the memory address of)* the variable \( x \), then \( *p \) is the variable \( x \)
    - If \( *p \) is on the LHS of an assignment, we're storing a value into \( x \)
    - If \( *p \) is on the RHS of an assignment, we're referring to the value in \( x \)
  - \( *(anything) \) means:
    - Figure out the value of "anything"
    - Interpret that as a memory location address
    - Use that memory location for whatever it is you're trying to do (e.g., as the lvalue in an assignment, as an operand in an arithmetic expression)

- **Key #3:** Remember the auto-scaling performed by pointer arithmetic