This Time

- Pointers (declaration and operations)
- Passing Pointers to Functions
- Const Pointers
- Bubble Sort Using Pass-by-Reference
- Pointer Arithmetic
- Arrays and Pointers
- Function Pointers

Pointers

- Pointers are variables which contain the memory addresses of other variables.
- C and C++ produce very fast programs in part because C++ and C programmers use pointers extensively.
- Pointers are very powerful because you can reference any piece of memory you want explicitly – most other languages don't allow this.
- Because pointers are so powerful they are also very dangerous and result in a lot of bugs.

Pointer Variable Declarations

- Pointer variables
  - Contain memory addresses as values
  - The variables we have seen so far contained a specific value (**direct reference**)
  - Pointers contain the *address* of variable that has specific value (**indirect reference**)
- Indirection
  - Referencing value through a pointer

Pointer Variable Declarations

- Pointer declarations
  - **'*'** indicates a variable is a pointer

```
int *myPtr;
```

    declares a pointer to an **int**, this is a pointer of type **int***
  - Multiple pointers require multiple asterisks (stars)

```
int *myPtr1, *myPtr2;
```

- Can declare pointers to any data type (float* weight, char* string)

Pointer Variable Initialization

Pointer initialization

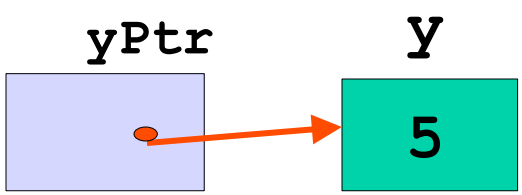
- Initialized to `0`, **`NULL`**, or address
  - `0` or **`NULL`** points to nothing and will cause an error if you try to de-reference the pointer (this is a good thing!).
  - If you don't initialize it to NULL you will get some random piece of memory and your program will only work sometimes – this is very hard to debug because the symptoms will vary from run to run.

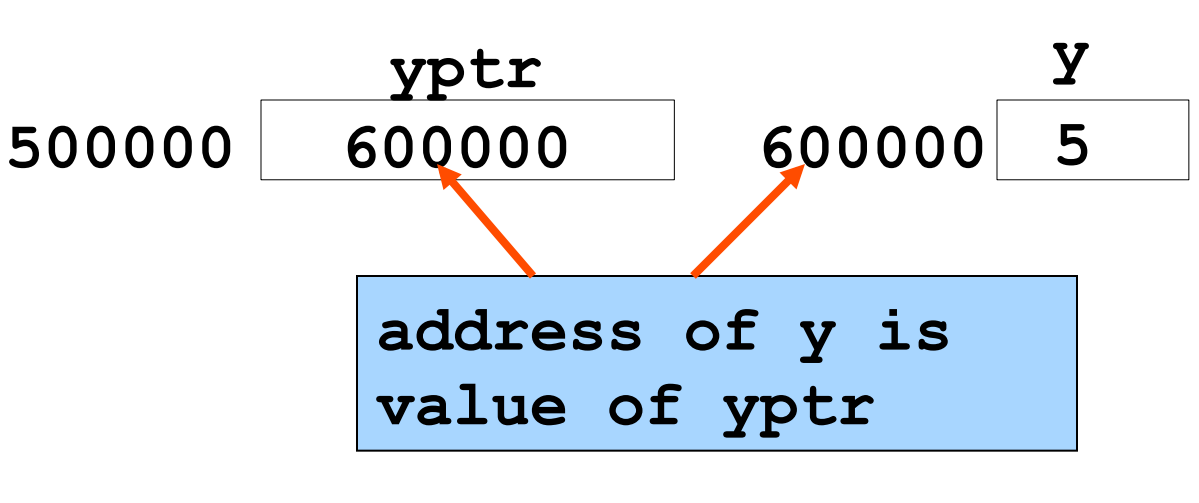
Pointer Operators '&'

- **&** (address operator)
  - Returns memory address of its operand

```
int y = 5;
int *yPtr;
yPtr = &y;// yPtr gets address of y
```

  - **yPtr** "points to" **y**





Pointer Operators

- **`*`** (indirection/dereferencing operator)
  - Provides access to the value in the memory location held by the pointer.
  - **`*yPtr`** returns **`y`** (because **`yPtr`** points to **`y`**).

    `*yptr = 9;` assigns 9 to y

    `cout<<*yptr;` prints y (here 9)
- **`*`** and **`&`** are inverses of each other

Pointer Operators (Example)

```
int x = 0;
int* y = NULL;


y = &x;
cout << y << x;
cout << *y << x;
cout << y << &x;
```

Calling Functions by Reference

- 3 ways to pass arguments to function
  - Pass-by-value
  - Pass-by-reference with reference arguments
  - Pass-by-reference with pointer arguments
- `return` can return one value from function
- Arguments passed to function using reference arguments
  - Modify original values of arguments
  - More than one value "returned"

Calling Functions by Reference

- Pass-by-reference with pointer arguments
  - Simulate pass-by-reference
    - Use pointers and indirection operator
  - Pass address of argument using **&** operator
  - Arrays not passed with **&** because array name already pointer
  - ***** operator used as alias/nickname for variable inside of function

Using const with Pointers

- **`const`** pointers
  - Always point to same memory location
  - Default for array name
  - Must be initialized when declared
  - Can't be changed

```
const int *cptr = &x;
```

Pointer Arithmetic

- Pointer arithmetic
  - Increment/decrement pointer (`++` or `--`)
  - Add/subtract an integer to/from a pointer (`+` or `+=` , `-` or `-=`)
  - Pointers may be subtracted from each other
  - Pointer arithmetic meaningless unless you know where your data is in memory (e.g. an array)
  - No other arithmetic operators are defined for pointers

Pointer Arithmetic

- Example:

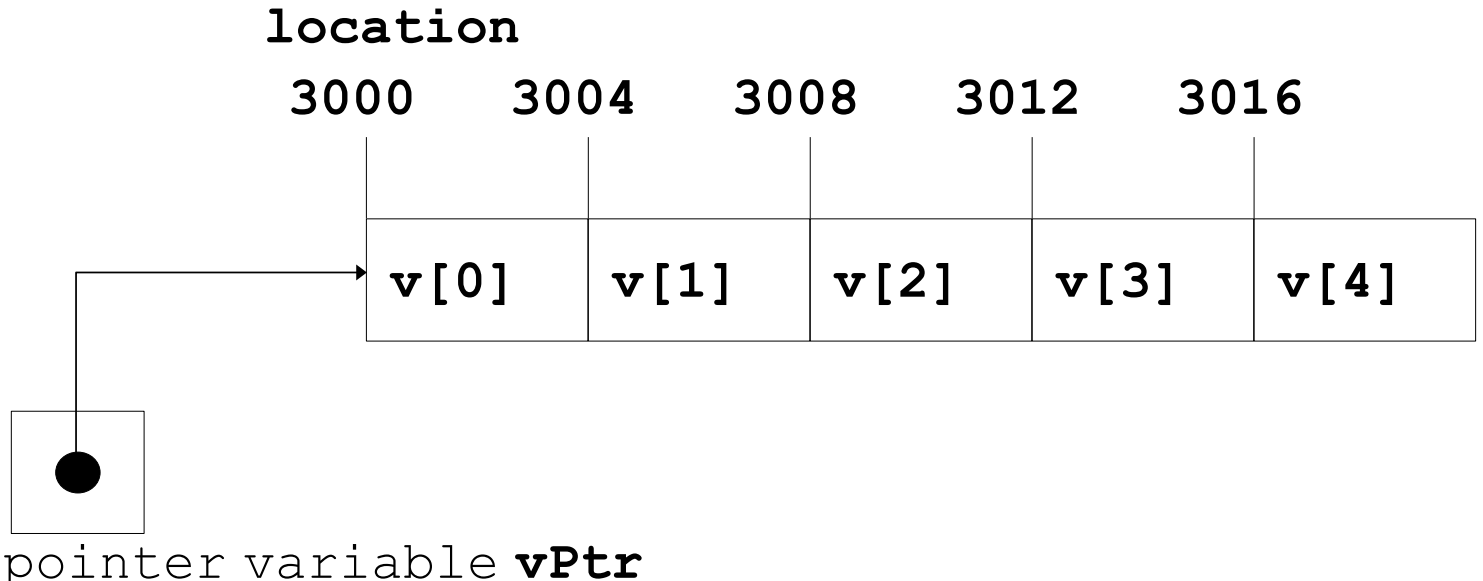
  5 element **int** array on a machine using 4 byte **int**s

  - **vPtr** points to first element **v[0]**, which is at location 3000

    vPtr = 3000

  - **vPtr += 2**; sets **vPtr** to 3008

    vPtr points to **v[2]**



Pointer Arithmetic

- Subtracting pointers
  - Returns the number of memory locations that must be traversed to get from one to the other

```
vPtr2 = v[ 2 ];
vPtr = v[ 0 ];
vPtr2 - vPtr == 2;
```

i.e.

The difference between the two pointers.

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

int main()
{
        int x[10];
        int *test1 = NULL, *test2 = NULL;

        test1 = &(x[0]); // 1st position in the array
        test2 = &(x[3]); // 4th position in the array

        cout << "sizeof(int) is " << sizeof(int) << endl;
        cout << "test1 is " << test1 << endl;
        cout << "test1 + 2 is " << test1 + 2 << endl;
        cout << "test1 - 2 is " <<test1 - 2 << endl;
        cout << "test2 - test1 is " << test1 - test2 << endl;

        return 0;
}
```

sizeof(int) is 4

test1 is 0012FF58

test2 is 0012FF64

test1 + 2 is 0012FF60

test1 - 2 is 0012FF50

test2 - test1 is 3

returns the number of memory locations test1 is from test2

Pointer Arithmetic

- Pointer assignment
  - A pointer can be assigned to another pointer if they are the same type
  - If they are not the same type, a cast operator must be used
  - Exception: pointer to **`void`** (type **`void*`**)
    - Generic pointer, represents any type
    - No casting needed to convert pointer to **`void`** pointer
    - **`void`** pointers cannot be dereferenced

Pointer Expressions

- Pointer comparison
  - Use equality and relational operators
  - Comparisons meaningless unless pointers point to members of same array
  - Compare addresses stored in pointers
  - Example: could show that one pointer points to higher numbered element of array than other pointer
  - Most common use (pointer == NULL)? Check to see if pointer points to anything

Relationship Between Pointers and Arrays

- Array name like constant pointer
- Accessing array elements with pointers
  - Element `b[ n ]` is the same as `*(bPtr + n)`
    - **Called pointer/offset notation**
  - Addresses
    - `&b[ 3 ]` same as `bPtr + 3`
  - Array name can be treated as pointer
    - `b[ 3 ]` same as `*( b + 3 )`
  - Pointers can be subscripted (pointer/subscript notation)
    - `bPtr[ 3 ]` same as `b[ 3 ]`

Function Pointers

- Pointers to functions
  - Function pointers contain the address of a function
  - Similar to how array name is address of first element
  - Function name is starting address of the code that defines the function
- Function pointers can be
  - Passed to functions
  - Returned from functions
  - Stored in arrays
  - Assigned to other function pointers

Function Pointers

- Calling functions using pointers
  - Declare a function pointer like this:

    `bool ( *compare ) ( int, int )`

    Where bool is the return type of your function and int, int are the argument types
  - Execute function with either
    - **`( *compare ) ( int1, int2 )`**
      - Dereference pointer to function to execute

    OR

    - **`compare( int1, int2 )`**
      - User may think **`compare`** name of actual function in program