More Pointers, Dynamic Memory

ITP 165 – Fall 2015

Week 11, Lecture 1

Dereferencing a Pointer

- Since a pointer stores a memory address, there needs to be a way to get the data at the memory address

- We can ***dereference*** a pointer in order to access the data at the memory address

- Let's look at an example...

Dereferencing a Pointer

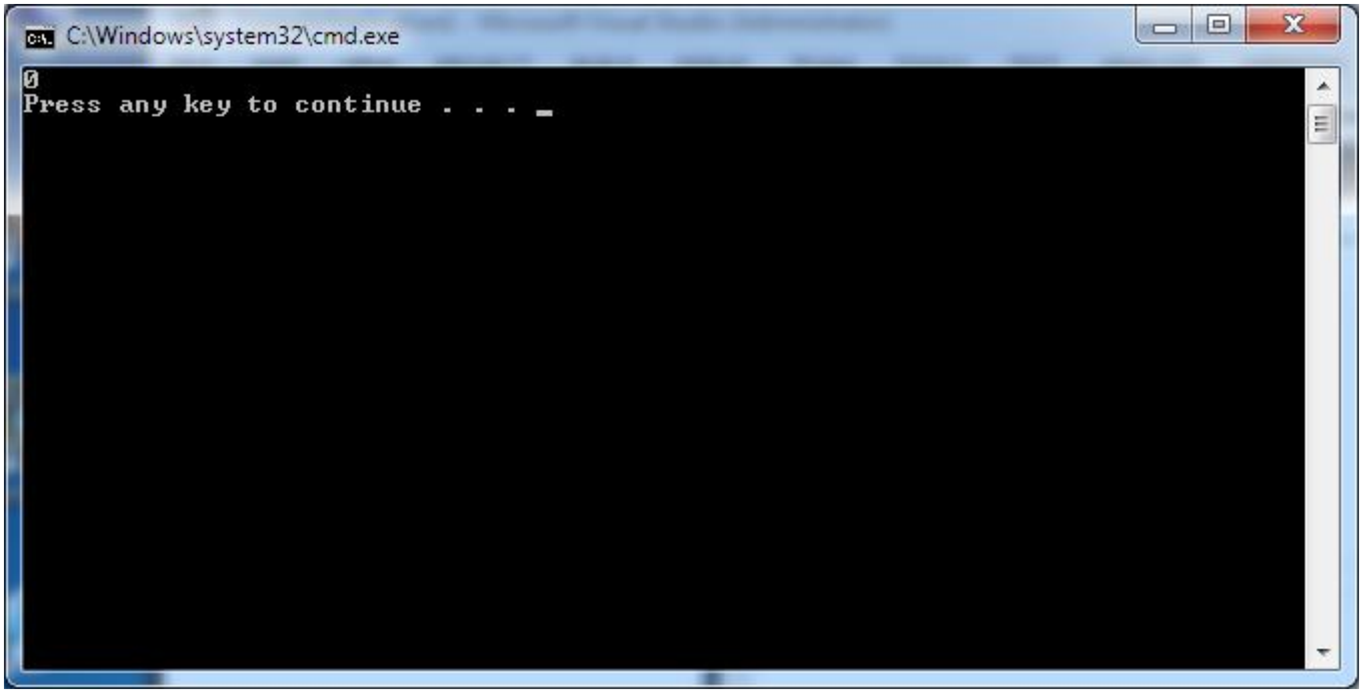
```
int x = 0;
double y = 5.0;

int* z = &x;

// *z means dereference z
std::cout << *z;
```

	Memory	
Variable	**Address**	**Value**
x	0x04	0
y	0x08	5.0
z	0x10	0x04

Dereferencing a Pointer, Full Example

```
#include <iostream>

int main() {
    int x = 0;
    double y = 5.0;
    int* z = &x;

    std::cout << *z << std::endl;
    return 0;
}
```

Dereferencing a Pointer, Full Example



Dereferencing and Assigning

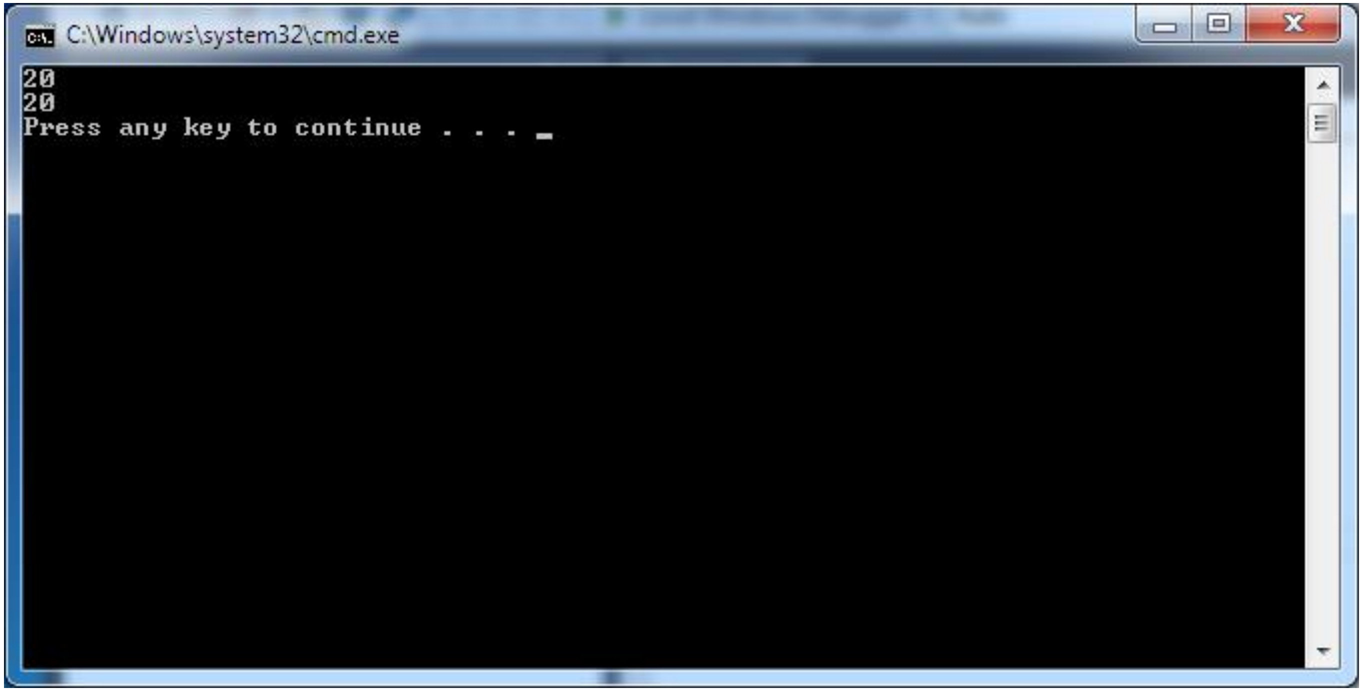
```
int x = 0;
double y = 5.0;

int* z = &x;

*z = 20;
```

	Memory	
Variable	**Address**	**Value**
x	0x04	20
y	0x08	5.0
z	0x10	0x04

Dereferencing and Assigning, Full Example

```
#include <iostream>

int main() {
    int x = 0;
    double y = 5.0;
    int* z = &x;
    *z = 20;

    std::cout << *z << std::endl;
    std::cout << x << std::endl;
    return 0;
}
```

Dereferencing and Assigning, Full Example



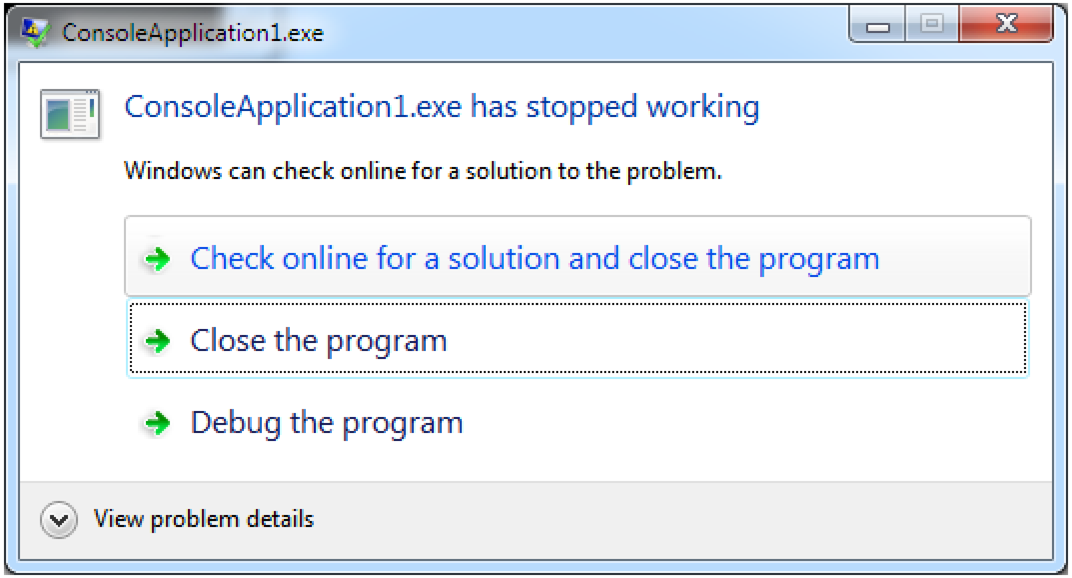
- So when we did *z = 20, we changed the value of x

Dereferencing Null Pointers

- If you dereference a null pointer, the program will crash.

- For example:

```
int* ptr = nullptr;
*ptr = 50; // Null dereference = crash!
```



So why use pointers?

- Pointers have fixed data sizes regardless of the data type they point to: 4 bytes
- We have to use pointers when using ***dynamic memory allocation***

The Stack

- When we create variables the normal way we've done it so far, they go on the ***stack***

- (This is the same stack we talked about before when we talked about function calls)

The Stack – Issues

- The amount of memory available to the stack is fairly limited.
- By default, Visual Studio only allows a 1 MB stack – if you go over, it crashes!

```
// Crash, because this array is
// approximately 4 MB in size.
int bigArray[1000000];
```

The Stack – Issues

- Remember that if we construct an array on the stack, we have to specify the size
- (Either explicitly, or by initializing it to set default values)
- So we can't ask the user for a number and then create an array of that size.

The Stack – Issues

- When we exit a scope, a variable declared on the stack no longer exists:

```
int* makeArray()
{
   int result[] = { 1, 2, 3, 4, 5 };

   // This will not work properly,
   // because the result array gets destroyed
   // when we exit the function!
   return result;
}
```

The Heap

- The solution to our problems is the ***heap***.

- The heap:
  - Has a lot more available memory
  - When we create data on the heap, it won't go away until either we manually delete it, or the program ends

- But there's a catch…if you want to use the heap in C++, you have to use pointers.

Dynamic Memory Allocation

- ***Dynamic memory allocation*** is the process by which you request memory from the heap

- There are two aspects of dynamic memory allocation:
  - When you want to request memory from the heap, you use `new`
  - When you want to free up memory you no longer need, you use `delete`

- This means we have full control of when variables are created and destroyed!

Pointer Declaration + new

Pointer to type we want to construct on heap

Type to construct on heap

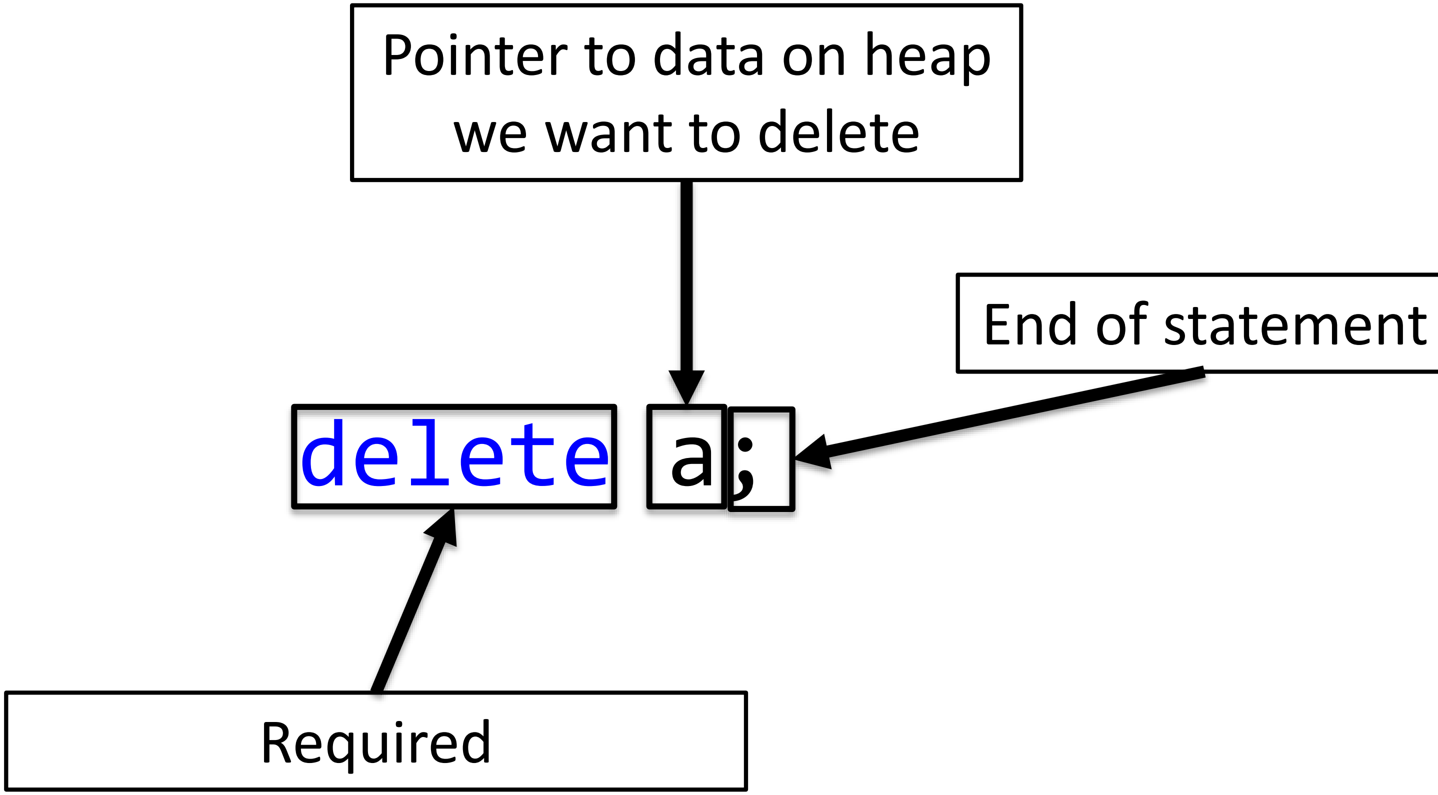
End of statement

```
int* a = new int;
```

Name of the pointer

Required

delete Syntax



Memory Leak

- A ***memory leak*** occurs when you forget to delete something from the heap when it's no longer needed
- Make sure you always delete stuff on the heap when it isn't needed anymore!

Dynamic Memory Example, Step by Step

```
#include <iostream>

int main() {

   // Create a new int on the heap.
   // We get back a pointer to this int.
   int* a = new int;

   // We can then dereference the pointer...
   *a = 50;

   std::cout << *a << std::endl;

   // This will delete our int on the heap
   delete a;
   // Set a to nullptr
   a = nullptr;
   return 0;
}
```

Dynamic Memory Example, Step by Step

```cpp
#include <iostream>

int main() {

    // Create a new int on the heap.
    // We get back a pointer to this int.
    int* a = new int;

    // We can then dereference the pointer...
    *a = 50;

    std::cout << *a << std::endl;

    // This will delete our int on the heap
    delete a;
    // Set a to nullptr
    a = nullptr;
    return 0;
}
```

	Stack	
Variable	**Address**	**Value**
a	0x10	0xF20

	Heap	
Variable	**Address**	**Value**
	0xF20	(garbage)

Dynamic Memory Example, Step by Step

```
#include <iostream>

int main() {

    // Create a new int on the heap.
    // We get back a pointer to this int.
    int* a = new int;

    // We can then dereference the pointer...
    *a = 50;

    std::cout << *a << std::endl;

    // This will delete our int on the heap
    delete a;
    // Set a to nullptr
    a = nullptr;
    return 0;
}
```

	Stack	
Variable	**Address**	**Value**
a	0x10	0xF20

	Heap	
Variable	**Address**	**Value**
	0xF20	50

Dynamic Memory Example, Step by Step

```
#include <iostream>

int main() {

    // Create a new int on the heap.
    // We get back a pointer to this int.
    int* a = new int;

    // We can then dereference the pointer...
    *a = 50;

    std::cout << *a << std::endl;

    // This will delete our int on the heap
    delete a;
    // Set a to nullptr
    a = nullptr;
    return 0;
}
```

	Stack	
Variable	**Address**	**Value**
a	0x10	0xF20

	Heap	
Variable	**Address**	**Value**
	0xF20	(garbage)

Dynamic Memory Example, Step by Step

```
#include <iostream>

int main() {

    // Create a new int on the heap.
    // We get back a pointer to this int.
    int* a = new int;

    // We can then dereference the pointer...
    *a = 50;

    std::cout << *a << std::endl;

    // This will delete our int on the heap
    delete a;
    // Set a to nullptr
    a = nullptr;
    return 0;
}
```

	Stack	
Variable	**Address**	**Value**
a	0x10	nullptr

	Heap	
Variable	**Address**	**Value**
	0xF20	(garbage)

Another Dynamic Memory Example

```
int* a = new int;

double* b = new double;
```

	Stack	
Variable	**Address**	**Value**
a	0x10	0xF20
b	0x14	0xFF8

	Heap	
Variable	**Address**	**Value**
	0xF20	(garbage)
	...	
	0xFF8	(garbage)

Dynamic Memory and Arrays

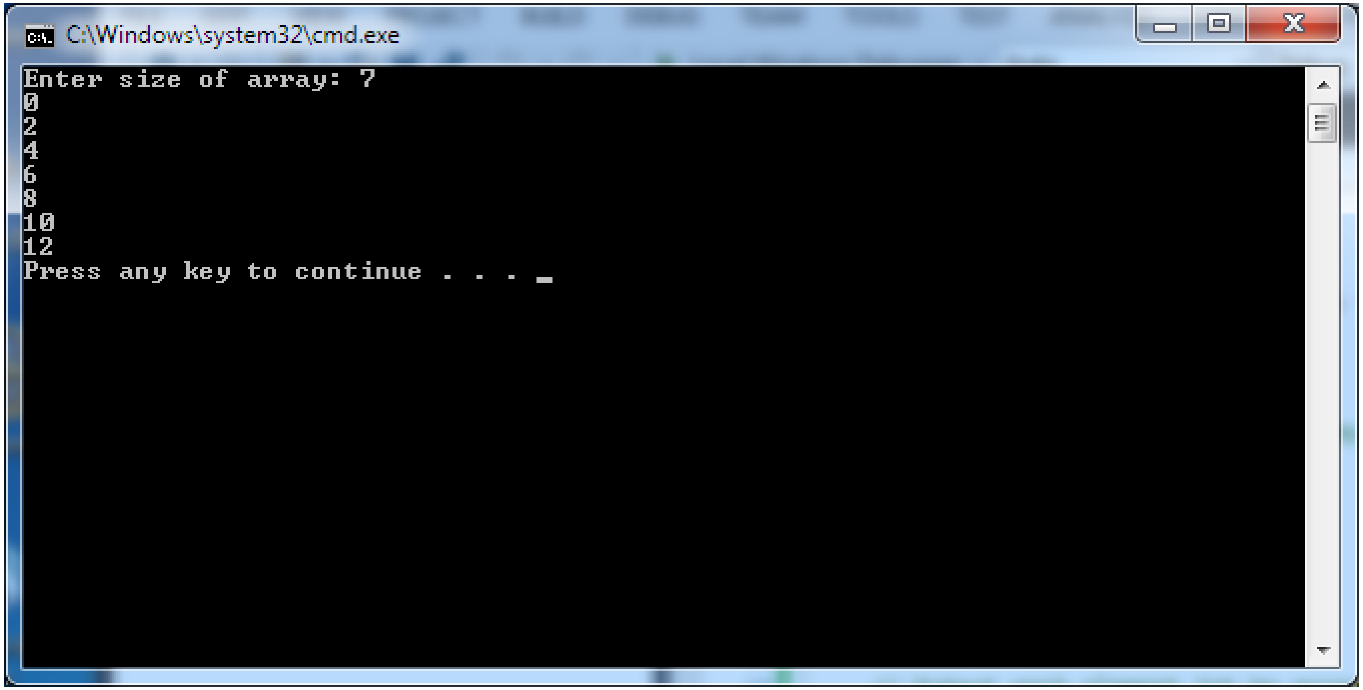
- Dynamic allocation doesn't make a great deal of sense for single variables
- But for arrays, it becomes super useful!
- We use `new` and `delete` like before, but now with some square brackets

```
// Create an array of 20 ints
int* intArray = new int[20];

// Delete the array of ints
delete[] intArray;
```

Dynamic Memory and Arrays, Example

```cpp
#include <iostream>
int main() {
   std::cout << "Enter size of array: ";
   int size = 0;
   std::cin >> size;

   // Dynamically allocate a new array...
   int* myArray = new int[size];
   // Set each value in the array
   for (int i = 0; i < size; i++) {
      myArray[i] = i * 2;
   }
   // Output each element in the array
   for (int i = 0; i < size; i++) {
      std::cout << myArray[i] << std::endl;
   }
   // Delete the array
   delete[] myArray;

   return 0;
}
```

Dynamic Memory and Arrays, Example



Arrays as Return Values

- If we construct an array using dynamic allocation, we can then return a pointer to the array, and it will work properly.

- For example:

```
int* makeArray(int size) {
   // Dynamically allocate a new array...
   int* retVal = new int[size];

   // Set each value in the array
   for (int i = 0; i < size; i++) {
      retVal[i] = i * 2;
   }

   return retVal;
}
```

Arrays as Return Values, Cont'd

- We could then change main to:

```
int main() {
   std::cout << "Enter size of array: ";
   int size = 0;
   std::cin >> size;

   int* myArray = makeArray(size);

   // Output each element in the array
   for (int i = 0; i < size; i++) {
      std::cout << myArray[i] << std::endl;
   }

   // Delete the array
   delete[] myArray;

   return 0;
}
```

Arrays as a Parameter

- Of course, we can always pass in an array as a parameter:

```
void outputArray(int array[], int size) {
    for (int i = 0; i < size; i++) {
        std::cout << array[i] << std::endl;
    }
}
```

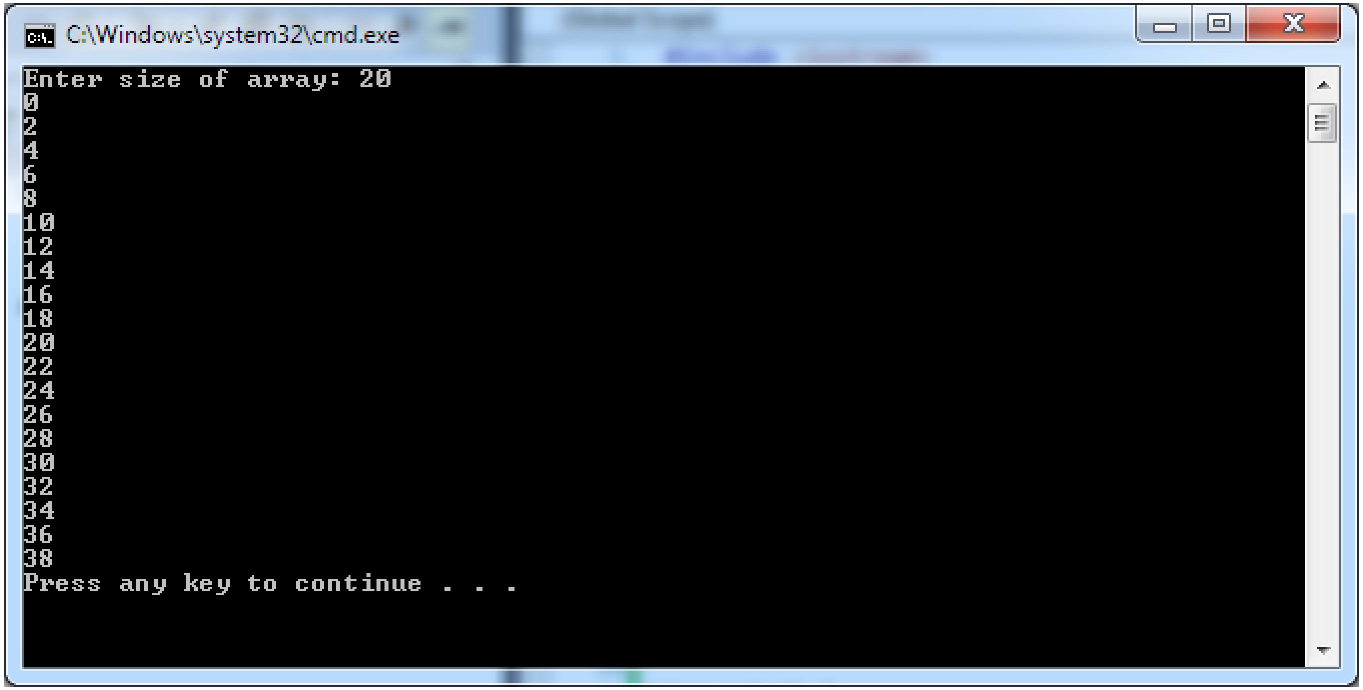
Arrays as a Parameter, Cont'd

- Then main can be:

```
int main() {
   std::cout << "Enter size of array: ";
   int size = 0;
   std::cin >> size;

   int* myArray = makeArray(size);

   outputArray(myArray, size);

   // Delete the array
   delete[] myArray;

   return 0;
}
```

```cpp
#include <iostream>
int* makeArray(int size) {
   // Dynamically allocate a new array...
   int* retVal = new int[size];

   // Set each value in the array
   for (int i = 0; i < size; i++) {
      retVal[i] = i * 2;
   }

   return retVal;
}

void outputArray(int array[], int size) {
   for (int i = 0; i < size; i++) {
      std::cout << array[i] << std::endl;
   }
}

int main() {
   std::cout << "Enter size of array: ";
   int size = 0;
   std::cin >> size;

   int* myArray = makeArray(size);
   outputArray(myArray, size);
   delete[] myArray;

   return 0;
}
```

Final Example



Lab Practical #18