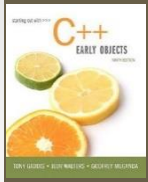
COP2334

Introduction to Object Oriented Programming with C++

D. Singletary

Module 8

Ch. 10 Pointers

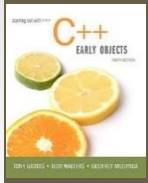


Pointers and the Address Operator

- Every variable is assigned a memory location whose address can be retrieved using the '&' ("address-of") operator
- The address of a memory location is called a pointer

```
int i = 10;
double d = 3.14;
char c = 'A';
```

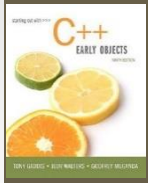
Sizes in Memory	
c	1 byte
d	8 bytes
i	4 bytes




```
int i = 10;
double d = 3.14;
char c = 'A';
```

```
sizeof(i) = 4
sizeof(d) = 8
sizeof(c) = 1
i's address = 2686716
d's address = 2686704
c's address = 2686703
```

```
cout << "sizeof(i) = " << sizeof(i) << endl;
cout << "sizeof(d) = " << sizeof(d) << endl;
cout << "sizeof(c) = " << sizeof(c) << endl;

cout << "i's address = " << reinterpret_cast<unsigned>(&i) << endl;
cout << "d's address = " << reinterpret_cast<unsigned>(&d) << endl;
cout << "c's address = " << reinterpret_cast<unsigned>(&c) << endl;
```




```
sizeof(i) = 4
sizeof(d) = 8
sizeof(c) = 1
i's address = 2686716
d's address = 2686704
c's address = 2686703
```

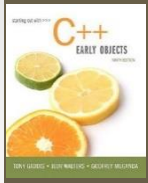
Memory		
c	1 byte	2686703
d	8 bytes	2686704
i	4 bytes	2686716

Pointer Variables

- A pointer variable stores the address of a memory location

```
int *p;
```

- The '*' symbol indicates p is a pointer to an integer
  - p is a variable, so its value can change; it can point to different memory locations
- * is known as the indirection operator



```
// pointersimple.cpp
// simple pointer demo

#include <iostream>
using namespace std;

int main()
{
   int x = 25;
   int *pX = &x;  // pX points to x's memory address

   cout << "The value of x is " << x << endl;
   cout << "The address of x is " << pX << endl;

   return 0;
}
```

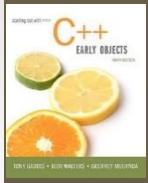
```
The value of x is 25
The address of x is 0x28fef8
```

The hexadecimal (base 16) memory address 0x28fef8 in decimal (base 10) is 2686712. Since x is an integer, the variable uses 4 bytes, so its address takes up memory addresses 2686712 through 2686715

- We can use a reinterpret_cast to cast pointer addresses into a decimal format
  - reinterpret_cast forces the compiler to interpret one type as if it were actually another (which can be dangerous in some situations)

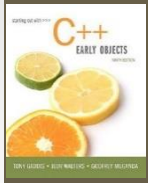
```
cout << "The value of x is " << x << endl;
cout << "The address of x is " << pX << endl;

cout << "The value of x is " << x << endl;
cout << "The address of x is " << reinterpret_cast<int>(pX);
```

```
The value of x is 25
The address of x is 0x28fef8
The value of x is 25
The address of x is 2686712
```



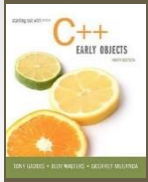
- A pointer can be used to indirectly access and modify the variable being pointed to.
  - This is known as dereferencing the pointer

```
int x = 25;
int *pX;
pX = &x;
// dereference pX to assign:
*pX = 50;   // x is now == 50
```



```
// pointersimple.cpp
// simple pointer demo

#include <iostream>
using namespace std;

int main()
{
      int x = 25, y = 50, z = 75;
      int* ptr = NULL;

      // display x, y, and z
      cout << "x = " << x << ", y = " << y << ", z = " << z << endl;

      ptr = &x;
      *ptr *= 2;

      ptr = &y;
      *ptr *= 2;

      ptr = &z;
      *ptr *= 2;

      cout << "after assignments..." << endl;
      cout << "x = " << x << ", y = " << y << ", z = " << z << endl;

      return 0;
}
```

```
x = 25, y = 50, z = 75
after assignments...
x = 50, y = 100, z = 150
```

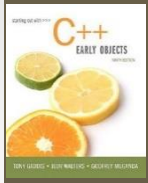


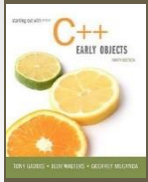
Arrays and Pointers

short numbers[] = { 10, 20, 30, 40, 50 };

- An array referenced by name only (no brackets or subscript) represents the starting address of the array
- In other words, an array name is a pointer

cout << "Element 1 of the numbers array is " << *numbers << endl;





- In order to reference other elements of the array, we can perform pointer arithmetic

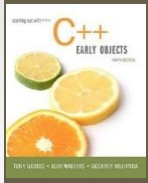
```
cout << "Element 2 of the numbers array is " <<
*(numbers + 1) << endl;
```

- *(numbers + 1)

  is equivalent to numbers[1]

```
cout << "Element 1 (index 0) of numbers = " << *numbers << endl;
cout << "Element 2 (index 1) of numbers = " << *(numbers + 1) << endl;
cout << "Element 3 (index 2) of numbers = " << *(numbers + 2) << endl;
cout << "Element 4 (index 3) of numbers = " << *(numbers + 3) << endl;
cout << "Element 5 (index 4) of numbers = " << *(numbers + 4) << endl;
```

```
Element 1 (index 0) of numbers = 10
Element 2 (index 1) of numbers = 20
Element 3 (index 2) of numbers = 30
Element 4 (index 3) of numbers = 40
Element 5 (index 4) of numbers = 50
```

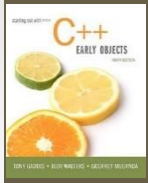


- When using pointer arithmetic, the value added to the base pointer is actually the value times the size of the element type
  - The size of a short is 2 bytes
  - Adding 1 to numbers using pointer arithmetic adds 2 to the actual address
  - If the numbers array starts at memory address 500000, then

&numbers == 500000

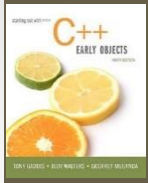
&(numbers + 1) == 500002

&(numbers + 2) == 500004



- Parentheses are critical when dereferencing pointers using pointer arithmetic:

```
    // assign 200 to numbers[1]
*(numbers + 1) = 200; // this is OK

*numbers + 1 = 200; // this is an invalid expression
    // "error: lvalue required as left operand of assignment"
    // *numbers + 1 is evaluated as "the contents of
    // numbers + 1". The compiler doesn't consider this
    // to be an assignable value (an lvalue is an expression
    // which something can be assigned to)

    // increment the value of numbers[0]
    // and assign to numbers[1]
*(numbers + 1) = *numbers + 1;
    // this is legal
    // *numbers + 1 as an rvalue is acceptable
```

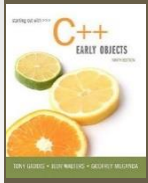


```
// process an array using a for loop and pointer arithmetic
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   const int SIZE = 4;
   int intArray[SIZE];

   for (int i = 0; i < SIZE; i++)
   {
      cout << "Please enter a value: ";
      cin >> *(intArray + i);
   }

   cout << "Your values are: " << endl;
   for (int i = 0; i < SIZE; i++)
      cout << *(intArray + i) << endl;

   return 0;
}
```

```
Please enter a value: 2
Please enter a value: 5
Please enter a value: 7
Please enter a value: 10
Your values are:
2
5
7
10
```

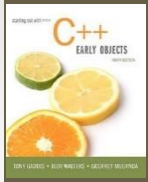
- We can use the postfix and prefix operators to increment/decrement a pointer when processing an array in a loop.

```
cout << "Your values are: " << endl;
for (i = 0, pInt = intArray; i < SIZE; i++)
      cout << *pInt++ << endl;

cout << "Now going backwards: " << endl;
for (i = 0; i < SIZE; i++)
      cout << *--pInt << endl;
```



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

int main()
{
   const int SIZE = 4;

   int intArray[SIZE];
   int *pInt = nullptr;
   int i = 0;

   for (i = 0, pInt = intArray; i < SIZE; i++)
   {
      cout << "Please enter a value: ";
      cin >> *pInt++;
   }

   cout << "Your values are: " << endl;
   for (i = 0, pInt = intArray; i < SIZE; i++)
      cout << *pInt++ << endl;

   cout << "Now going backwards: " << endl;
   for (i = 0; i < SIZE; i++)
      cout << *--pInt << endl;

   return 0;
}
```

```
Please enter a value: 2
Please enter a value: 4
Please enter a value: 6
Please enter a value: 8
Your values are:
2
4
6
8
Now going backwards:
8
6
4
2
```

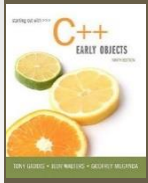


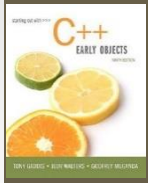
- In the previous slide, we can move backward immediately after moving forward since our pointer is already positioned at the end of the array plus one element (after the final postfix). Note that we decrement the pointer before accessing the array using a prefix decrement in the second loop's first iteration:

```
cout << "Your values are: " << endl;
for (i = 0, pInt = intArray; i < SIZE; i++)
      cout << *pInt++ << endl;

cout << "Now going backwards: " << endl;
for (i = 0; i < SIZE; i++)
      cout << *--pInt << endl;
```

- To "reset" the pointer to the last element of the array so we can move backwards again:

```
pInt = intArray + SIZE; // pointer arithmetic!
```

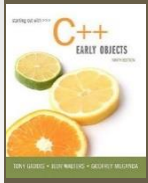




- We can use prefix and postfix increment/decrement to modify the values pointed to by a pointer.
  - Notice placement of ++ and * for prefix notation compared to using it when processing in a loop; we do not want to change the pointer's address value for this scenario, we want to change the value of the variable pointed to
  - Parentheses must be used for postfix to override operator precedence

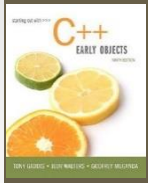
```
// increment iVal using a pointer
// and prefix/postfix operators
int iVal = 20;
int *pIVal = &iVal;
cout << ++*pIVal << endl;
cout << (*pIVal)++ << endl;
cout << *pIVal << endl;
```

```
21
21
22
```



- what will the following code display?

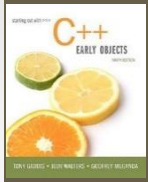
```
char c = 'A';
char *pC = &c;
cout << *pC << endl;
cout << ++*pC << endl;
cout << (*pC)++ << endl;
cout << --*pC << endl;
cout << (*pC)-- << endl;
cout << *pC << endl;
```



- what will the following code display?

```
char c = 'A';
char *pC = &c;
cout << *pC << endl;
cout << ++*pC << endl;
cout << (*pC)++ << endl;
cout << --*pC << endl;
cout << (*pC)-- << endl;
cout << *pC << endl;
```

```
A
B
B
B
B
A
```



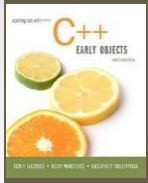
// Identify the error in the following code:

```
double dVal = 20.0;
double *pDVal = &dVal;
// increment dVal after displaying its value
cout << *pDVal++ << endl;
cout << *pDVal << endl;
```

// Identify the error in the following code:

```
double dVal = 20.0;
double *pDVal = &dVal;
// increment dVal after displaying its value
cout << *pDVal++ << endl;
cout << *pDVal << endl;
```

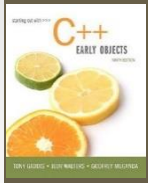
- With no parentheses, this code displays the current value of dVal and then increments the memory address pDVal points to; pDVal no longer points to dVal's address
- **(*pDVal)++** in the first cout will fix it



Initializing Pointers

- Pointers are designed to point to objects of a specific type

```
int myValue = 0;
int *pVal = &myValue;  // pointer to int

int ages[20];
int *pAges = ages;  // pointer to array of int

float myFloat = 0.0;
int *pVal = &myFloat;  // illegal
```

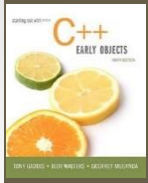


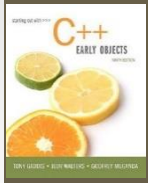
Initializing Pointers to NULL

- If a pointer is not initialized at declaration, initialize it to NULL

```
int *pVal = NULL;
```

- NULL points to address 0, effectively pointing to "nothing", and by convention is used to indicate an invalid memory location





- C++ 11 also defines the **nullptr** keyword to act as NULL

```
int *pVal = nullptr;
```

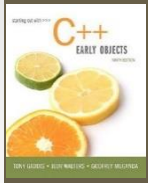
- Internally, NULL and nullptr refer to address 0, which is also legal as an initial pointer value (inherited from C and older C++ compilers), but use NULL or nullptr

```
int *pVal = 0; // legal, but use NULL or nullptr
```

Testing Pointers Against NULL

- All of the following are equivalent:

```
if (pVal == NULL) { ...

if (pVal == nullptr) { ...

if (pVal) { ...

if (pVal == 0) { ...
```



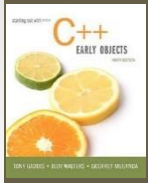
Sidebar: Initializing Variables to Defaults

- Appending braces to a variable at initialization assigns it to its default value

```
int myInt = 0;
double myDouble = 0.0;
int *ptrToInt = nullptr;
```

- The following statements are equivalent to the statements shown above

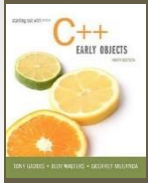
```
int myInt{};
double myDouble{};
int *ptrToInt{};
```



Comparing Pointers

- Pointers may be compared using any of the relational operators

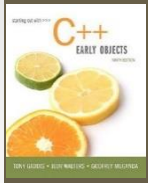
```
int array[] = { 4, 3, 2, 1 };

if (array[1] > array[3])  // true
cout << "array[1] is > array[3]" << endl;

if (&array[1] > &array[3])  // false
cout << "&array[1] > &array[3] << endl;
```



Pointers as Function Parameters

- A pointer can be used as a function parameter
  - The pointer gives the function access to the original variable, similar to a call by reference
  - This is known as "pass by address"
    - Passing pointers can be more efficient than references

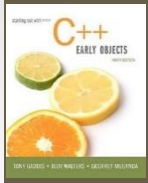
```
void doubleValue(double *val)
{
      // double the value stored in val
      *val *= 2;
}
```



Calling a Function Using Pass by Address

- To call the function on the previous slide, we can use the address-of operator when passing a local variable to the function:

```
int main()
{
    double dVal = 1.0;
    // double the value stored in dVal
    doubleValue(&dVal);
    ...
```



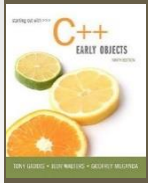
Array Pointers as Function Parameters

- We don't need to use the address-of operator when passing an array to a function, but we do need to use the indirection operator in the parameter declaration:

```
int main()
{
      const int SIZE = 3;
      double dArray[SIZE] = { 1.0, 2.0, 3.0 };
     // double the contents of the array
      doubleArray(dArray, SIZE);
      ...

void doubleArray(double *array, int size)
{
       for (int i = 0; i < size; i++)
            *array++  *=  2.0;
}
```

- Note on the previous slide that we manipulated the array pointer (passed as a parameter) instead of declaring a separate double pointer in the function as an array index
  - We can do this since the original address is copied into the parameter; modifying this local copy does not impact the original array address.
  - We can't do this with the original array variable since the compiler will complain that it isn't a modifiable value

```
void doubleArray(double *array, int size)
{
        for (int i = 0; i < size; i++)
                *array++ *= 2.0;
}
```

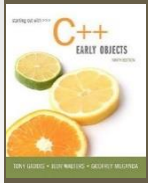


Pointers to Constants and Constant Pointers

- Passing the address of a constant and attempting to modify it results in a compiler error:

```
const int CONSTVAL = 20;
modifyValue(&CONSTVAL);
...

void modifyValue(int *val)
{
   *val = 50;
}
```

error: invalid conversion from 'const int*' to 'int*'



- We can prevent modification of a value in a function by declaring the parameter to be const:

```
int aVal = 20;
modifyValue(&aVal);
...
void modifyValue(const int *val)
{
   *val = 50;
}
```

error: assignment of read-only location '* val'

- If you write a function which includes pointer parameters that should not be modified, use "const" to prevent future modifications which could break that restriction

Memory Management in C++

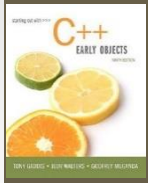
- Local variables are mapped into a specific memory location known as the stack.
- Stack memory is limited; in order to acquire larger amounts of memory for data structures we use a reserved memory space called the heap.

Memory
Stack
Compiled program code
Global Variables
Heap (dynamic memory)

Dynamic Memory Allocation

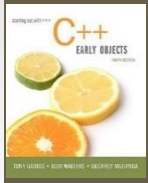
- Dynamic memory allocation
  - acquires memory during a program's execution through the use of pointers and the new operator:

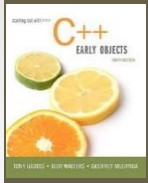
```
int *iptr = new int;

*iptr = 25;

cout << "iptr = " << *iptr;
```



- New allocates memory from the heap
  - Example: allocate an array from the heap:

```
const int SIZE = 100;
int *iptr = new int[SIZE];

for (int count = 0; count < SIZE; count++)
        iptr[count] = count;
```

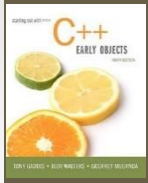




- Once the memory is no longer needed, it should be released back to the heap by using the delete operator:

```
delete iptr;
iptr = nullptr;
```

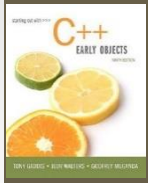
- After deleting the memory, set the pointer to nullptr to indicate there is no longer any memory associated with it



- If the pointer refers to a dynamically allocated array, use [] in the delete statement:

```
delete [] iptrArray;
iptrArray = nullptr;
```

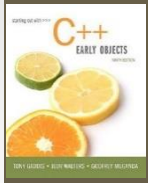
- Textbook Note: if it is clear that a pointer will no longer be referenced in a program, delete isn't required, but it doesn't hurt to do it anyway to increase maintainability.
- Instructor's Note: delete is always required for this course.



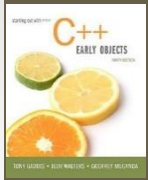
- Failing to delete heap memory can lead to memory leaks.
  - Heap memory is large, but not infinite; when it runs out, your application can crash or experience unpredictable behavior
- Failing to set a pointer to nullptr after deleting its associated memory results in a dangling pointer.
  - Accessing a dangling pointer (e.g. assigning a value to it) can crash your application or cause unpredictable behavior
  - But remember the pointer is now set to nullptr, don't try to dereference it

Returning Pointers from Functions

- Functions can return pointers, but the object pointed to cannot go out of scope (ie. must continue to exist)
- Don't return pointers to (addresses of) local variables!
- Allocate space for the object and return a pointer to the allocated space
- The caller of the function must remember to delete the memory when no longer needed



```
// squares returns a dynamically allocated array
// of squared integers up to the passed value
int* squares(int n)
{
    // allocate an array of size n
    int *sqarray = new int[n];

    // fill the array with squares
    for (int k = 0; k < n; k++)
       sqarray[k] = (k+1) * (k+1);

    // return base address of allocated array
    return sqarray;
}
```

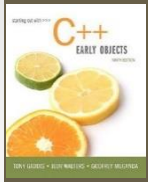


```
// ptrsquares.cpp
// demonstrates returning a pointer from a function
#include <iostream>
using namespace std;

int* squares(int n);

int main()
{
   const int SIZE = 4;

   int *squaresArray = squares(SIZE);

   for (int i = 0; i < SIZE; i++)
      cout << "squares[" << i << "] = " << squaresArray[i] << endl;

   delete [] squaresArray;
   squaresArray = nullptr;

   return 0;
}

// squares returns a dynamically allocated array of squared integers up to the passed value
int* squares(int n)
...
```

```
squares[0] = 1
squares[1] = 4
squares[2] = 9
squares[3] = 16
```

```
int* squares(int n)
{
     int sqarray[n];  // ERROR!

    // fill the array with squares
    for (int k = 0; k < n; k++)
       sqarray[k] = (k+1) * (k+1);

    // return base address of allocated array
    return sqarray;
}
```

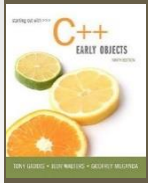
```
squares[0] = 1
squares[1] = 2686528
squares[2] = 9
squares[3] = 9
```



Avoiding Memory Leaks

- Ideally, a function which dynamically allocates memory should also be the one to delete it, but this isn't always possible
- When dynamically allocating memory in a class, do the allocation in the constructor, and the delete in the destructor
  - Verify that the pointer is not NULL before deleting

    if ptr != nullptr
- (see squaresclass.cpp on Blackboard)



Pointers to Classes and Structures

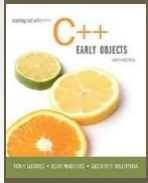
- Using pointers for classes and structures is the same as using pointers for other objects

```
Rectangle r;
Rectangle *pR = &r;
```

- To access a class member:

```
(*pR).width = 5;
```

- Parentheses are required since the dot '.' has a higher precedence than *



The Structure Pointer Operator

- Since the combination of dots, asterisks, and parentheses are difficult to read, C++ provides the Structure Pointer Operator -> as an alternative:

```
pR->width = 5;
```

- This operator can also be used to call member functions:

```
pr->getWidth();
```

- (see ptrstructclass.cpp on Blackboard)