# AAB University

## Faculty of Computer Sciences

## Object Oriented Programming

Week 9:

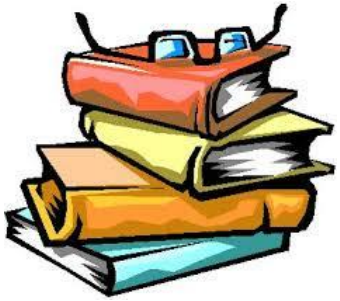
# Pointers

Asst. Prof. Dr. **Mentor Hamiti**
mentor.hamiti@universitetiaab.com

# *Last Time?!*

- **Midterm Exam!**



# *Today*

- Pointers
- Pointer Operators
- Pass-by-Reference with Pointers
- Built-In Arrays
- Using ***const*** with Pointers
- sizeof Operator
- Pointer Expressions and Pointer Arithmetic
- Relationship Between Pointers and Built-In Arrays

# *Pointers*

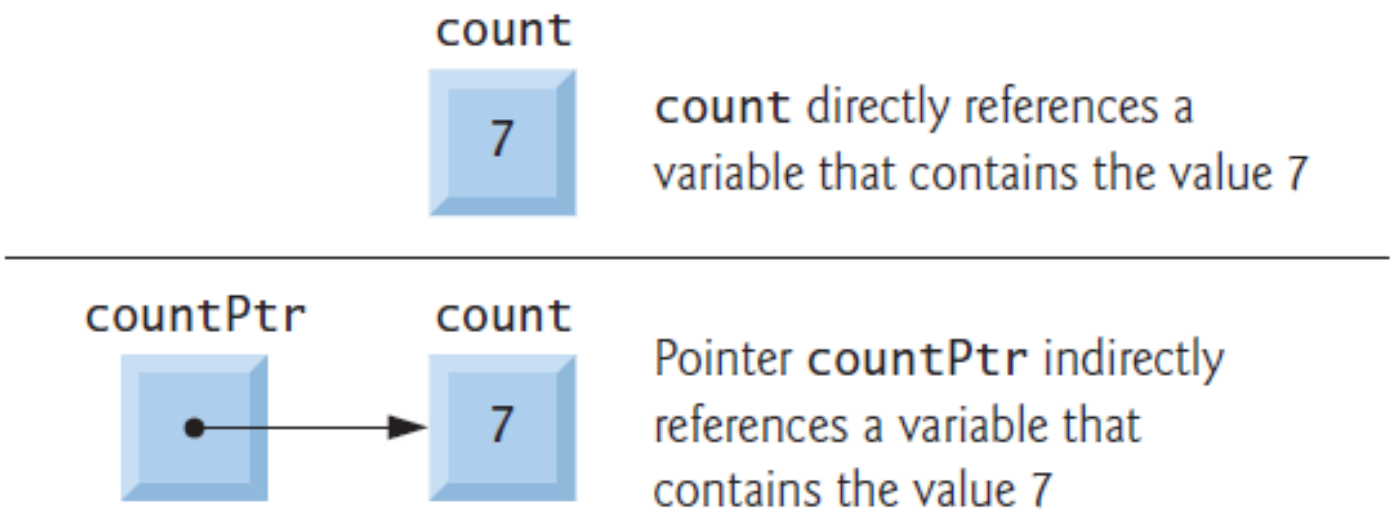
- Pointers are one of the most powerful, yet challenging to use, C++ capabilities
- Our goals here are to help you determine when it's appropriate to use pointers, and show how to use them correctly and responsibly
- Pointers also enable pass-by-reference and can be used to create and manipulate dynamic data structures that can grow and shrink, such as linked lists, queues, stacks and trees
- We also show the intimate relationship among built-in arrays and pointers

# *Pointer Variable Declarations and Initialization*

- **Indirection**
- A pointer contains the memory address of a variable that, in turn, contains a specific value
- In this sense, a variable name directly references a value, and a pointer indirectly references a value
- Referencing a value through a pointer is called **indirection**
- *Pointers can be declared to point to objects of any data type*

# Pointer Variable Declarations and Initialization

- Directly and indirectly referencing a variable




- *Diagrams typically represent a pointer as an arrow from the variable that contains an address to the variable located at that address in memory*

# Pointer Variable Declarations and Initialization

- The declaration:

```
int  *countPtr,  count;
```

  - declares the variable countPtr to be of type int * (i.e., *a pointer to an int value*) and is read (*right to left*), "countPtr is a pointer to int"

- Variable count in the preceding declaration is declared to be an int, not a pointer to an int

- The * in the declaration applies only to countPtr
  - *Each variable being declared as a pointer must be preceded by an asterisk (*)*
  - *When * appears in a declaration, it is not an operator; It indicates that the variable being declared is a pointer*

# Pointer Variable Declarations and Initialization

### Common Programming Error

Assuming that the * used to declare a pointer distributes to all names in a declaration's comma-separated list of variables can lead to errors. Each pointer must be declared with the * prefixed to the name (with or without spaces in between). Declaring only one variable per declaration helps avoid these types of errors and improves program readability.

### Good Programming Practice

Although it's not a requirement, including the letters `Ptr` in a pointer variable name makes it clear that the variable is a pointer and that it must be handled accordingly.

# *Pointer Variable Declarations and Initialization*

- **Initializing Pointers**

- Pointers should be initialized to **nullptr** (*new in C++11*) or an address of the corresponding type either when they're declared or in an assignment

- A pointer with the value **nullptr** *"points to nothing"* and is known as a **null pointer**

**Error-Prevention**

Initialize all pointers to prevent pointing to unknown or uninitialized areas of memory.

# *Pointer Variable Declarations and Initialization*

- In earlier versions of C++, the value specified for a null pointer was **0** or **NULL**
- **NULL** is defined in several standard library headers to represent the value **0**
  - *Initializing a pointer to NULL is equivalent to initializing a pointer to 0, but prior to C++11,* ***0*** *was used by convention*
- The value **0** is the only integer value that can be assigned directly to a pointer variable without first casting the integer to a pointer type

- Pointers
- Pointer Operators

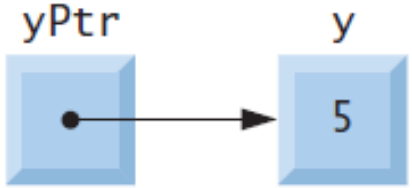
# Pointer Operators

- **Address (&) Operator**
- The **address operator** (**&**) is a unary operator that obtains the memory address of its operand

```
int   y = 5;                // declare variable y
int  *yPtr = nullptr;       // declare pointer variable yPtr

yPtr = &y;                  // assign address of y to yPtr
```

- Graphical representation of a pointer pointing to a variable in memory:



# *Pointer Operators*

- Example:
  - **A pointer** representation in memory with integer variable **y** stored at memory location **600000** and pointer variable **yPtr** stored at location **500000**
    - Representation of **y** and **yPtr** in memory:

# Pointer Operators

- **Indirection (*) Operator**
- The unary * operator - commonly referred to as the indirection operator or dereferencing operator
  - *returns an lvalue representing the object to which its pointer operand points*

```
int   y = 5;
int  *yPtr = nullptr;
yPtr =  &y;
*yPtr =  9;

cout<<yPtr<<endl;
cout<<&y<<endl;
cout<<*yPtr<<endl;
cout<<y<<endl;
```

```
004EFB44
004EFB44
9
9
```

# Pointer Operators

- Example 1: Using the Address (&) and Indirection (*) Op.

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

int main()
{
    int a; // a is an integer
    int *aPtr; // aPtr is an int * which is a pointer to an integer
    a = 7; // assigned 7 to a
    aPtr = &a; // assign the address of a to aPtr

    cout << "The address of a is " <<&a
         << "\nThe value of aPtr is " << aPtr;
    cout << "\n\nThe value of a is " << a
         << "\nThe value of *aPtr is " << *aPtr;
    cout << "\n\nShowing that * and & are inverses of "
         << "each other.\n&*aPtr = " << &*aPtr
         << "\n*&aPtr = " << *&aPtr << endl;

    cin.get(); return 0;
}
```

```
The address of a is 0025FAA0
The value of aPtr is 0025FAA0

The value of a is 7
The value of *aPtr is 7

Showing that * and & are inverses of each other.
&*aPtr = 0025FAA0
*&aPtr = 0025FAA0
```

# Pointer Operators

- The address (&) and dereferencing operator (*) are unary operators on the fourth level

| Operators | Associativity | Type |
|---|---|---|
| :: () | *[See caution in Fig. 2.10]* | highest |
| () [] | left to right | function call/array access |
| ++ -- static_cast<*type*>(*operand*) | left to right | unary (postfix) |
| ++ -- + - ! & * | right to left | unary (prefix) |
| * / % | left to right | multiplicative |
| + - | left to right | additive |
| << >> | left to right | insertion/extraction |
| < <= > >= | left to right | relational |
| == != | left to right | equality |
| && | left to right | logical AND |
| \|\| | left to right | logical OR |
| ?: | right to left | conditional |
| = += -= *= /= %= | right to left | assignment |
| , | left to right | comma |

- Pointers
- Pointer Operators
- Pass-by-Reference with Pointers

# *Pass-by-Reference with Pointers*

- There are three ways in C++ to pass arguments to a function:
  - 1. pass-by-value, 2. pass-by-reference with reference arguments and 3. pass-by-reference with pointer arguments
- Pointers, like references, can be used to modify one or more variables in the caller or to pass pointers to large data objects to avoid the overhead of passing the objects by value
- Pointers and the indirection operator (*) can be used to accomplish pass-by-reference
- When calling a function with an argument that should be modified, the address of the argument is passed

# Pass-by-Reference with Pointers

- Example 2:

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

int cubeByValue( int ); // prototype

int main()
{
    int number = 5;

    cout << "The original value of number is " << number;
    number = cubeByValue( number ); // pass number by value to cubeByValue
    cout << "\nThe new value of number is " << number << endl;

    cin.get();  return 0;
}

// calculate and return cube of integer argument
int cubeByValue( int n )
{
return n * n * n;   // cube local variable n and return result
};
```

```
The original value of number is 5
The new value of number is 125
```

# Pass-by-Reference with Pointers

- Example 3:

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

void cubeByReference( int * ); // prototype

int main()
{
    int number = 5;

    cout << "The original value of number is " << number;
    cubeByReference( &number ); // pass number address to cubeByReference
    cout << "\nThe new value of number is " << number << endl;

    cin.get();  return 0;
}

// calculate cube of *nPtr; modifies variable number in main
void cubeByReference( int *nPtr )
{
*nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
} // end function cubeByReference
```

```
The original value of number is 5
The new value of number is 125
```

# *Pass-by-Reference with Pointers*

- All Arguments Are Passed By Value
- In C++, all arguments are always passed by value
- Passing a variable by reference with a pointer does not actually pass anything by reference—a pointer to that variable is passed by value and is copied into the function's corresponding pointer parameter
- The called function can then access that variable in the caller simply by dereferencing the pointer, thus accomplishing pass-by-reference.

# Graphical Analysis of Pass-By-Value (Ex. 2)

Step 1: Before `main` calls `cubeByValue`:

```
int main()
{
   int number = 5;

   number = cubeByValue( number );
}
```

number: 5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: undefined

Step 2: After `cubeByValue` receives the call:

```
int main()
{
   int number = 5;

   number = cubeByValue( number );
}
```

number: 5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: 5

Step 3: After `cubeByValue` cubes parameter `n` and before `cubeByValue` returns to `main`:

```
int main()
{
   int number = 5;

   number = cubeByValue( number );
}
```

number: 5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

125

n: 5

# Graphical Analysis of Pass-By-Value (Ex. 2)

Step 4: After `cubeByValue` returns to `main` and before assigning the result to `number`:

```
int main()
{
   int number = 5;

   number = cubeByValue( number );
}
```

number: 5

125

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: undefined

Step 5: After `main` completes the assignment to `number`:

```
int main()
{
   int number = 5;

   number = cubeByValue( number );
}
```

number: 125

125 125

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: undefined

# Graphical Analysis of Pass-By-Reference (Ex. 3)

Step 1: Before `main` calls `cubeByReference`:

```
int main()
{
   int number = 5;

   cubeByReference( &number );
}
```

number: 5

```
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;
}
```

nPtr: undefined

Step 2: After `cubeByReference` receives the call and before `*nPtr` is cubed:

```
int main()
{
   int number = 5;

   cubeByReference( &number );
}
```

number: 5

```
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;
}
```

nPtr

*call establishes this pointer*

Step 3: After `*nPtr` is cubed and before program control returns to `main`:

```
int main()
{
   int number = 5;

   cubeByReference( &number );
}
```

number: 125

```
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;
}
```

125

nPtr

*called function modifies caller's variable*

- Pointers
- Pointer Operators
- Pass-by-Reference with Pointers
- Built-In Arrays

# *Built-In Arrays*

- Declaring a Built-In Array

```
// c is a built-in array of 12 integers
int  c [12];
```

- Accessing a Built-In Array's Elements
  - As with array objects, the subscript ([]) operator is used to access the individual elements of a built-in array

- Initializing Built-In Arrays

```
int  n[ 5 ] = { 50, 20, 30, 10, 40 };
```

  - Initialized - fundamental **numeric types** are set to **0**, **bools** are set to **false**, **pointers** are set to **nullptr** and **class objects** are initialized by their default **constructors**

# *Built-In Arrays*

- Passing Built-In Arrays to Functions

- The value of a built-in array's name is implicitly convertible to the address of the built-in array's first element
  - So **arrayName** is implicitly convertible to **&arrayName[0]**

- For built-in arrays, the called function can modify all the elements of a built-in array in the caller - unless the function precedes the corresponding built-in array parameter with **const** to indicate that the elements should not be modified

# Built-In Arrays

- Declaring Built-In Array Parameters

- A built-in array parameter can be declared in a function header:

```
int  sumElements (const int values[], const size_t
        numberOfElements )
```

  - *which indicates that the function's first argument should be a one-dimensional built-in array of ints that should not be modified by the function*

- The preceding header can also be written as:

```
int sumElements( const int *values, const size_t
        numberOfElements )
```

# *Built-In Arrays*

- Built-in arrays have several limitations:
  - They *cannot be compared* using the relational and equality operators - you must use a loop to compare two built-in arrays element by element
  - They *cannot be assigned* to one another
  - They *don't know their own size* - a function that processes a built-in array typically receives *both* the built-in array's *name* and its *size* as arguments
  - They *don't provide automatic bounds checking* - you must ensure that array-access expressions use subscripts that are within the built-in array's bounds
- Objects of class templates **array** and **vector** are safer, more robust and provide more capabilities than built-in arrays

# *Built-In Arrays*

- Sometimes Built-In Arrays Are Required
  - There are cases in which built-in arrays must be used, such as processing a program's **command-line arguments**
    - Such arguments typically pass options to a program
  - On a **Windows** computer, the command: **dir /p** uses the **/p** argument to list the contents of the current directory, pausing after each screen of information
  - On **Linux** or **OS X**, the following command uses the **-la** argument to list the contents of the current directory with details about each file and directory: **ls -la**

- Pointers
- Pointer Operators
- Pass-by-Reference with Pointers
- Built-In Arrays
- Using ***const*** with Pointers

# *Using const with Pointers*

- Many possibilities exist for using (or not using) **const** with function parameters

- **Principle of least privilege**:

  *Always give a function enough access to the data in its parameters to accomplish its specified task, but no more*

**Software Engineering Observation**

If a value does not (or should not) change in the body of a function to which it's passed, the parameter should be declared `const`.

# *Using const with Pointers*

- There are four ways to pass a pointer to a function:
  1. A nonconstant pointer to nonconstant data
  2. A nonconstant pointer to constant data
  3. A constant pointer to nonconstant data
  4. A constant pointer to constant data
- Each combination provides a different level of access privilege

# *1. Nonconstant Pointer to Nonconstant Data*

- The highest access is granted by a **nonconstant pointer** to **nonconstant data**
  - The *data can be modified* through the dereferenced pointer, and the pointer can be modified to point to other data

- Such a pointer's declaration:

```
int *countPtr
```

does not include **const**

# 2. Nonconstant Pointer to Constant Data

- A **nonconstant pointer** to **constant data**
  - A pointer that can be modified to point to any data item of the appropriate type, but the data to which it points cannot be modified through that pointer
- Might be used to receive a built-in array argument to a function that should be allowed to read the elements, but not modify them
- Any attempt to modify the data in the function results in a compilation error
- Sample declaration: const int *countPtr;
  - Read from right to left as "countPtr is a pointer to an integer constant" or more precisely, "countPtr is a non-constant pointer to an integer constant."

# 2. Nonconstant Pointer to Constant Data

- Example 4:

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

void f( const int * ); // prototype


int main()
{
    int y=0;
    f( &y );    // f attempts illegal modification
}


void f(const int *xPtr )
{
    *xPtr = 100;    // error: cannot modify a const object
};
```

```
error C3892: 'xPtr' : you cannot assign to a variable that is const
```

# 3. Constant Pointer to Nonconstant Data

- A **constant pointer to nonconstant data** is a pointer that always points to the same memory location, and the data at that location can be modified through the pointer
- Pointers that are declared **const** must be *initialized when they're declared*

# 3. Constant Pointer to Nonconstant Data

- Example 5:

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

int main()
{
    int x, y;

    // ptr is a constant pointer to an integer that can
    // be modified through ptr, but ptr always points to the
    // same memory location.
    int * const ptr = &x; // const pointer must be initialized

    *ptr = 7; // allowed: *ptr is not const

    ptr = &y; // error: ptr is const; cannot assign to it a new address

}
```

```
error C3892: 'ptr' : you cannot assign to a variable that is const
```

# 4. Constant Pointer to Constant Data

- The minimum access privilege is granted by a **constant pointer to constant data**
  - Such a pointer always points to the same memory location, and the data at that location cannot be modified via the pointer
  - This is how a built-in array should be passed to a function that only reads from the built-in array, using array subscript notation, and does not modify the built-in array

# 4. Constant Pointer to Constant Data

- Example 6:

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

int main()
{
    int x = 5, y;

    // ptr is a constant pointer to a constant integer.
    // ptr always points to the same location; the integer
    // at that location cannot be modified.

    const int *const ptr = &x;

    cout << *ptr << endl;

    *ptr = 7; // error: *ptr is const; cannot assign new value
    ptr = &y; // error: ptr is const; cannot assign new address

}
```

```
error C3892: 'ptr' : you cannot assign to a variable that is const
error C3892: 'ptr' : you cannot assign to a variable that is const
```

- Pointers
- Pointer Operators
- Pass-by-Reference with Pointers
- Built-In Arrays
- Using ***const*** with Pointers
- sizeof Operator

# *sizeof Operator*

- The unary operator **sizeof** determines the size in bytes of a built-in array or of any other data type, variable or constant during program compilation
- When applied to a built-in array's name, the sizeof operator returns the total number of bytes in the built-in array as a value of type size_t
- When applied to a pointer parameter in a function that receives a built-in array as an argument, the sizeof operator returns the size of the pointer in bytes—not the built-in array's size

# sizeof Operator

- Example 7:

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

size_t getSize(double *); // prototype

int main()
{
    double array[ 20 ]; // 20 doubles; occupies 160 bytes on our system
    cout <<"The number of bytes in the array is " << sizeof( array );

    cout << "\nThe number of bytes returned by getSize is " <<getSize( array ) << endl;

    cin.get();  return 0;
}

size_t getSize( double *ptr )
{
return sizeof( ptr );
}
```

```
The number of bytes in the array is 160
The number of bytes returned by getSize is 4
```

- Pointers
- Pointer Operators
- Pass-by-Reference with Pointers
- Built-In Arrays
- Using ***const*** with Pointers
- sizeof Operator
- Pointer Expressions and Pointer Arithmetic

# *Pointer Expressions and Pointer Arithmetic*

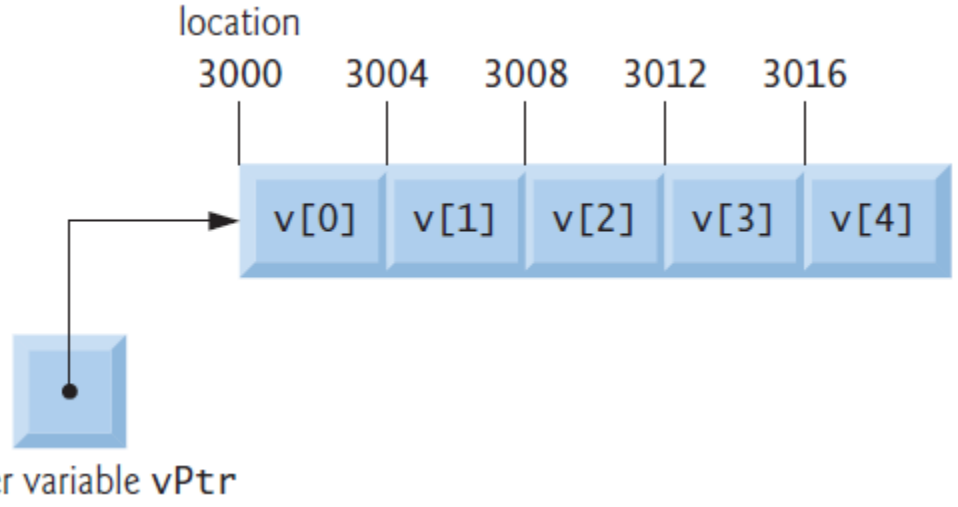
- Pointers are valid operands in arithmetic expressions, assignment expressions and comparison expressions

- C++ enables **pointer arithmetic** - a few arithmetic operations may be performed on pointers:
  - increment (++)
  - decremented (--)
  - an integer may be added to a pointer (+ or +=)
  - an integer may be subtracted from a pointer (- or -=)
  - one pointer may be subtracted from another of the same type
    - *this particular operation is appropriate only for two pointers that point to elements of the same built-in array*

# *Pointer Expressions and Pointer Arithmetic*

- Assume that int **v[5]** has been declared and that its first element is at memory location **3000**
- Assume that pointer **vPtr** has been initialized to point to **v[0]** (i.e., the value of **vPtr** is **3000**)
- Variable **vPtr** can be initialized to point to **v** with either of the following statements (*for a machine with four-byte integers*):

```
int *vPtr = v;
int *vPtr = &v[ 0 ];
```



# *Pointer Expressions and Pointer Arithmetic*

- Adding Integers to and Subtracting Integers from Pointers
- In conventional arithmetic, the addition 3000 + 2 yields the value 3002
  - This is normally not the case with pointer arithmetic
  - When an integer is added to, or subtracted from, a pointer, the pointer is not simply incremented or decremented by that integer, but by that integer times the size of the object to which the pointer refers
  - The number of bytes depends on the object's data type

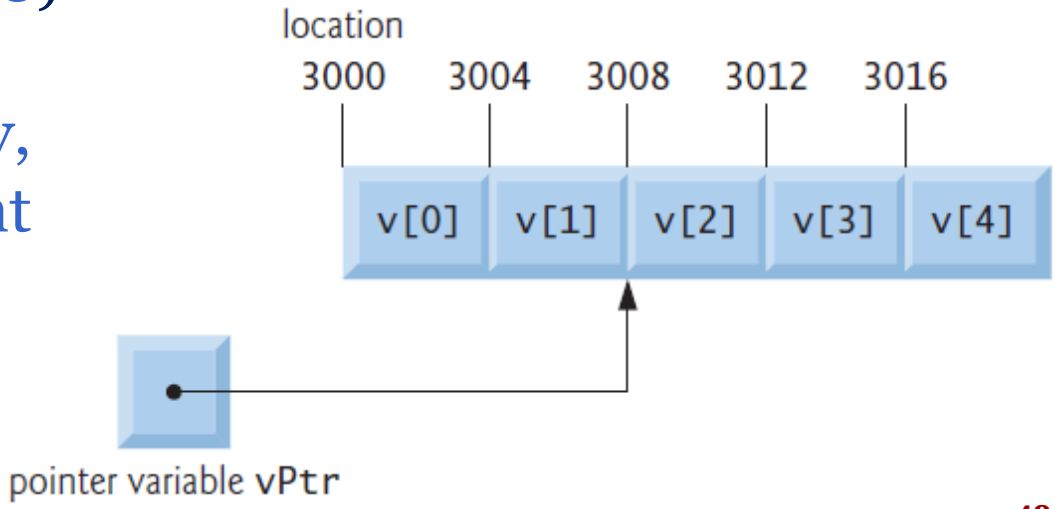
# Pointer Expressions and Pointer Arithmetic

- Example:

  vPtr += 2;

  - This statement would produce **3008** (from the calculation 3000 + 2 * 4), assuming that an **int** is stored in four bytes of memory
  - If an integer is stored in eight bytes of memory, then the preceding calculation would result in memory location **3016** (3000 + 2 * 8)

- In the built-in array **v**, **vPtr** would now point to v[2]



# *Pointer Expressions and Pointer Arithmetic*

- Subtracting Pointers
- Pointer variables pointing to the same built-in array may be subtracted from one another
- Example:
  - if **vPtr** contains the address **3000** and **v2Ptr** contains the address **3008**, the statement

    x = v2Ptr - vPtr;

  - would assign to **x** the number of built-in array elements from **vPtr** to **v2Ptr** - in this case, **2**
- Pointer arithmetic is meaningful only on a pointer that points to a built-in array

# Pointer Expressions and Pointer Arithmetic

- Pointer Assignment
- A pointer can be assigned to another pointer if both pointers are of the same type
- Otherwise, a **cast operator** must be used to convert the value of the pointer on the right of the assignment to the pointer type on the left of the assignment
  - *Exception to this rule is the pointer to void (i.e., void *)*
- Any pointer to a fundamental type or class type can be assigned to a pointer of type void * without casting
  - *The compiler must know the data type to determine the number of bytes to dereference for a particular pointer - for a pointer to void, this number of bytes cannot be determined*

# *Pointer Expressions and Pointer Arithmetic*

- Comparing Pointers

- Pointers can be compared using equality and relational operators
- Comparisons using relational operators are meaningless unless the pointers point to elements of the same built-in array
- Pointer comparisons compare the addresses stored in the pointers
- A common use of pointer comparison is determining whether a pointer has the value **nullptr**, **0** or **NULL** (i.e., the pointer does not point to anything)

- Pointers
- Pointer Operators
- Pass-by-Reference with Pointers
- Built-In Arrays
- Using ***const*** with Pointers
- sizeof Operator
- Pointer Expressions and Pointer Arithmetic
- Relationship Between Pointers and Built-In Arrays

# Relationship Between Pointers and Built-In Arrays

- Pointers can be used to do any operation involving array subscripting
- Example:

```
// create 5-element int array b; b is a const pointer
int b[ 5 ];

// create int pointer bPtr, which isn't a const pointer
int *bPtr;

// assign address of built-in array b to bPtr
bPtr = b;

// also assigns address of built-in array b to bPtr
bPtr = &b[ 0 ];
```

# Relationship Between Pointers and Built-In Arrays

- Example:

```
// Built-in array element b[ 3 ] can alternatively be
// referenced with the pointer expression
 *( bPtr + 3 )

// Just as the built-in array element can be referenced
// with a pointer expression
&b[ 3 ]

// The address can be written with the pointer expression
bPtr + 3

// Next expression also refers to the element b[ 3 ]
*( b + 3 )
```

# Relationship Between Pointers and Built-In Arrays

- Example 8:

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

int main()
{
    int b[] = { 10, 20, 30, 40 }; // create 4-element array b
    int *bPtr = b; // set bPtr to point to array b

    // output array b using array subscript notation
    cout << "Array b printed with:\n\nArray subscript notation\n";
    for ( int i = 0; i < 4; ++i )
        cout << "b[" << i << "] = " << b[ i ]<< '\n';

    // output array b using the array name and pointer/offset notation
    cout << "\nPointer/offset notation where "
        << "the pointer is the array name\n";
    for ( int offset1 = 0; offset1 < 4; ++offset1 )
        cout << "*(b + " << offset1 << ") = " << *( b + offset1 )<< '\n';

    // output array b using bPtr and array subscript notation
    cout << "\nPointer subscript notation\n";
    for ( int j = 0; j < 4; ++j )
        cout << "bPtr[" << j << "] = " <<bPtr[ j ] << '\n';
    cout << "\nPointer/offset notation\n";

    // output array b using bPtr and pointer/offset notation
    for ( int offset2 = 0; offset2 < 4; ++offset2 )
        cout << "*(bPtr + " << offset2 << ") = "
        <<*( bPtr + offset2 )<< '\n';

    cin.get();  return 0;
}
```

```
Array subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

Pointer/offset notation where
the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

Pointer subscript notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40
```

- Questions?!

mentor.hamiti@universitetiaab.com