# POINTERS

Chapter 8

## Most of this Chapter is a Review

All the content in this chapter was introduced in CIS 236 – Programming in C

- The prerequisite for this class

Use this presentation as a review of those concepts

# TOPICS

Introduction

Pointer Variable Declarations and Initialization

Pointer Operators

Pass-by-Reference with Pointers

Built-In Arrays

Using const with Pointers

Sizeof Operator

Pointer Expressions and Pointer Arithmetic

Relationships between Pointers and Built-In Arrays

# INTRODUCTION TO POINTERS

# Introduction – Passing Data

Remember that passing data to functions can happen in two different ways:

1. Pass-by-value

2. Pass-by-reference

# Two Ways to Pass Arguments to Functions

## Pass-by-value

- A **copy of the data** is sent to the function
- Original data not modified
- May be inefficient when working with very large items, like objects

## Pass-by-reference

- A **reference to the original data** is sent to the function, not a copy
- Can be a performance advantage
- But may not protect original data from modification
- Can do this in C++ w/ **reference parameters** (chapter 6) or **pointers** (chapter 8)

# USES FOR POINTERS

Pass-by-reference

Dynamic data structures that can grow and shrink

- E.g., linked lists, queues, stacks and trees
- Take CIS 269 to learn about these!

Pointer notation with built-in arrays

# Pointer Variable Declarations and Initialization

# INDIRECTION

A variable contains a value

Pointer variable contains the address of a variable as its value

Using a variable's **name** to access its value is **direct access**

Using a variable's **address** to access its value is **indirect access**

So, a pointer **indirectly references** a value

Referencing a value via a pointer is called **indirection**

## DECLARING POINTER VARIABLES

Example: int *countPtr, count;

countPtr is a pointer to an integer; count is an integer

Each pointer variable name in a list must be preceded by *

Example: int *xPtr, *yPtr;

Both are pointers to integers

## INITIALIZING POINTERS

At declaration, a pointer should be initialized to a variable's address, or to the value **nullptr** (which sets the pointer to NULL)

Examples:

| | |
|---|---|
| **int *intPtr{&y};** | Declares a pointer variable intPtr and sets its value to the address of the variable y |
| **int *intPtr{nullptr};** | Declares a pointer variable intPtr and sets its value to NULL |

# POINTER OPERATORS

## Address (&) Operator Definition

Unary operator that obtains the memory address of its operand

Example:

```
int y{5};

int *yPtr{nullptr};

yPtr = &y;
```

## ADDRESS (&) OPERATOR NOTES

Note that this is not the same usage of & as in a reference variable declaration

In a **reference variable declaration, the & is always preceded by a data type name**

- So the & is part of the type

In a pointer variable assignment, the **variable name is preceded by the &**

- So it is the address operator

Cannot be applied to constants or to expressions that result in temporary values (like the results of calculations)

# INDIRECTION (*) OPERATOR

Returns an lvalue representing the object to which its pointer operand points

- Called “dereferencing the pointer”

This is **indirect access**

Example, if yPtr is pointing to y:

```
cout << *yPtr << endl;      // Displays the value of y

*yPtr = 9;                  // Sets y to the value 9

cin >> *yPtr;               // Places input value in y
```

Examples: Fig08_04.cpp

# More Pointer Operations

Dereferencing uninitialized or null pointer is undefined and is usually a fatal runtime error

- So check for null before dereferencing

Chart showing precedence and associativity of the operators discussed so far on p. 344

# Pass-by-Reference with Pointers

## PASSING ARGUMENTS IN C++

Three ways to pass arguments to a function in C++:

Pass-by-value

Pass-by-reference with reference arguments (chapter 6)

Pass-by-reference with pointer arguments

## Pass-by-Reference with Pointers – Benefits

Arguments passed by reference:

Enable called function to modify original values of the arguments

Enable passing of large data objects to avoid overhead of copying

Pointers can be used in the same way as reference parameters

- This is pass-by-reference the way C does it
- Passing the address of the object that must be modified

# Pass-by-Reference with Pointers – Examples

Pass-By-Value example

- Fig08_06.cpp

Pass-By-Reference with Pointers example

- Fig08_07.cpp

Note – a function receiving an address as an argument must define a pointer parameter to receive the address

# All Arguments are Passed by Value!

Passing a variable by reference with a pointer passes the address by value…

# Built-In Arrays

# C++ Has Built-In Arrays

In chapter 7:

- array class template – fixed-size array
- vector class template – dynamically-sized array-like class

C++ also has built-in arrays

- Fixed-size
- Same syntax as C
- Not objects like the array class

# Declaring and Accessing C++ Built-In Arrays

Declaring:

**type  arrayName[arraySize];**

Access elements with subscript operator []:

**arrayName[subscript]**

No automatic bounds checking!!

## INITIALIZING BUILT-IN ARRAYS

Use initializer list:

```
int n [ 5 ] = { 5, 2, 3, 1, 4 };
```

If fewer initial values than size indicated, remaining elements initialized according to the array type

Too many initial values is compiler error

If size omitted, number of initializer list elements is used as size

- Good idea to always specify a size even when using an initializer list, so compiler can generate an error message if the size and list don't match.

## Passing Built-In Arrays to Functions

Value of the built-in array's name is **implicitly** convertible to the address of the built-in array's first element:

So, the name of the array is the address of the first element

No need to get address of array when passing – just pass the name

Called function can modify original array (unless the array parameter is declared const)

Good software engineering practice to not let functions modify the original array – principle of least privilege

# Declaring Built-In Array Parameters in a Function Header

Example:

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

Unlike the array class, built-in arrays do not know their own size, so the size of the array must also be included in the parameter list

# Declaring Built-In Array Parameters in a Function Header, v2

Alternate syntax using a pointer:

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

Compiler does not differentiate between function that receives a pointer and a function that receives a built-in array

But use [] syntax for clarity, since the function would have to know that the pointer represents the beginning of an array

## STANDARD LIBRARY FUNCTIONS SORT(), BEGIN() & END()

Remember standard library function sort function from chapter 7:

**sort ( *arrayName*.begin(), *arrayName*.end() );**

Sort can be applied to built-in arrays too, using a slightly different syntax:

**sort ( begin (*arrayName*), end (*arrayName*) );**

Begin and end functions in <array> header

## Built-In Array Limitations

Cannot be compared using relational and inequality operators (must use a loop to compare each element)

Cannot be assigned to one another

Don’t know their own size

Don’t have automatic bounds checking

Sometimes built-In arrays are required, but modern C++ code uses the array or vector classes

# USING CONST WITH POINTERS

# USING CONST WITH PARAMETERS – PRINCIPLE OF LEAST PRIVILEGE

Remember the principle of least privilege:

- If a function doesn't need to modify a parameter, declare it with **const**

Example: a function that receives an array as an argument and prints the contents of the array, but doesn't need to change it

- Declare both the array and its size parameters as const

## USING CONST WITH POINTERS – FOUR COMBINATIONS

Four ways to pass a pointer to a function:

1. Nonconstant pointer to nonconstant data
2. Nonconstant pointer to constant data
3. Constant pointer to nonconstant data
4. Constant pointer to constant data

# Nonconstant Pointer to Nonconstant Data

Highest access

Data can be modified via the pointer

Pointer can be modified to point to other data

## NONCONSTANT POINTER TO CONSTANT DATA

Data cannot be modified

Pointer can be modified to point to any other data of the appropriate type

Might be used to receive a built-in array in a function that can read the elements but not modify them

## NONCONSTANT POINTER TO CONSTANT DATA EXAMPLE

Example:

**const int *countPtr;**

countPtr is a pointer to an integer constant, or more precisely, “countPtr is a nonconstant pointer to an integer constant”

Can also write this as **int const* countPtr;** to make it obvious that const applies to the int, not the pointer.

fig08_10.cpp – demonstrates error message if the value the pointer references is changed

# Nonconstant Pointer to Constant Data Notes

When passing objects, use pointers or references to avoid overhead of passing object by value

- array and vector objects are passed by value

Use pointers to constant data if the objects don't need to be changed

Pass fundamental types by value unless receiving function requires access to original

## CONSTANT POINTER TO NONCONSTANT DATA

Data can be modified

Pointer cannot be modified to point to other data – it always points to the same memory location

Const pointers must be initialized when they are declared

In a function parameter, a const pointer is initialized with pointer that is passed to the function when the function is called

## CONSTANT POINTER TO NONCONSTANT DATA EXAMPLE

Fig08_11.cpp – attempts to modify a constant pointer

Example:

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

ptr is a constant pointer to the nonconstant integer x

ptr cannot be reassigned to point at another integer

## Constant Pointer to Constant Data Notes

Least privilege

Data cannot be modified

Pointer cannot be modified to point to other data – it always points to the same memory location

This is how a built-in array should be passed to a function that only needs to read it

## CONSTANT POINTER TO CONSTANT DATA

Fig08_12.cpp

Example:

**const int *const ptr = &x;**

ptr is a constant pointer to the integer constant variable x

# SIZEOF OPERATOR

## SIZEOF OPERATOR PURPOSE

Determines size in bytes of built-in array (or any other data type) at compile time

When applied to array name, returns total number of bytes in built-in array, as a value of size_t

When applied to **pointer parameter** in a function that receives a built-in array as an argument, returns size of the **pointer** in bytes (not the array it points to)

Fig08_13.cpp demonstrates this

## SIZEOF OPERATOR EXAMPLES

Can determine number of elements in a built-in array (at compile time) with this expression:

**sizeof *arrayName* / sizeof (*arrayName*[0] )**

Can be applied to any expression or type name

- When applied to a variable (that is not a built-in array name) or an expression, it gives the number of bytes in the type
- Use parentheses when using a type name; parentheses not required for an expression

Examples of determining sizes in Fig08_14.cpp

# POINTER EXPRESSIONS AND POINTER ARITHMETIC

## Arithmetic on [Some] Pointer Expressions

Some arithmetic operations can be performed on pointers

**But only appropriate for pointers to built-in array elements**

Pointer arithmetic is machine-dependent (integers can be 4 or 8 bytes)

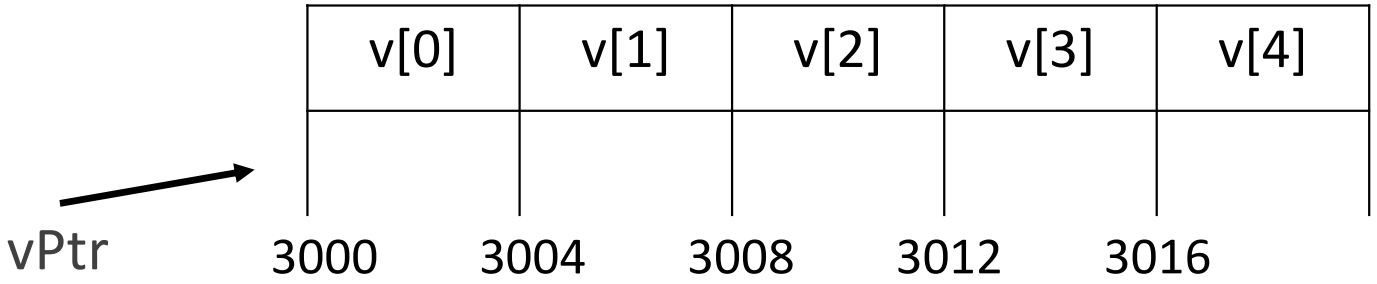
Pointer can be:

- Incremented or decremented
- Integer may be added to or subtracted to from it
- Pointer may be subtracted from pointer of same type, if they both point to elements of the same built-in array

# POINTER ARITHMETIC EXAMPLE

Example (using 4-byt ints)

int v[5];




int *vPtr{v};

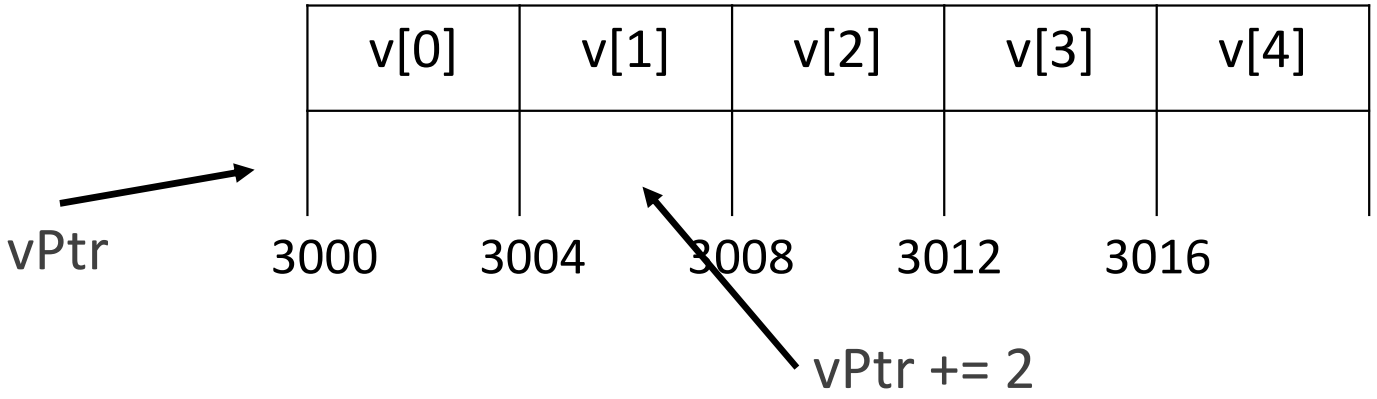
or

int *vPtr{&v[0]};

# POINTER ARITHMETIC EXAMPLE, CONTINUED

Example (using 4-byt ints)

int v[5];




Pointer is incremented or decremented by the integer in the expression multiplied by the size of object to which the pointer refers

So **vPtr += 2** → 3008 (3000 + 2 * 4)

## SUBTRACTING POINTERS

If vPtr contains address 3000, and v2Ptr contains address 3008, then

**x = v2Ptr – vPtr**

assigns to x the number of array elements from v2Ptr to vPtr (2 in this case, if ints stored using 4 bytes)

Pointer arithmetic only meaningful on pointers that point to the same built-in array

And no bounds checking

## POINTER ASSIGNMENT

Pointer can be assigned to another pointer if both of same type

Can be assigned to a different type using a cast

- Usually reinterpret_cast (chapter 14)

Exception is pointer to void

# POINTER TO VOID

**void * ptrName**

Generic pointer capable of representing any pointer type

- Also called a void pointer

Contains a memory address for an unknown data type

- Not the same as null pointer, which is no address

Pointer to any fundamental or class type can be assigned to a void pointer without casting

# Operations on Pointer to void

Allowed operations:

- Comparing void pointer to other pointers
- Casting void pointer to other pointer types
- Assigning addresses to void pointer

Operations that are not allowed:

- Cannot dereference a void *

## COMPARING POINTERS

Use equality and relational operators

Compares the **addresses** stored in the pointers

Meaningless unless pointers are both pointing to same built-in array elements

Commonly used to compare a pointer to **nullptr**, **0**, or **NULL**

# Relationships between Pointers and Built-In Arrays

## Pointers and Built-In Array Names

May be used almost interchangeably

Pointers can be used to do any operation involving array subscripting

Called pointer offset notation, and pointer subscript notation

## POINTER OFFSET NOTATION EXAMPLE

```
int b[5];

int *bPtr;

bPtr = b;
```

**Element b[3]** can be referenced with the pointer expression: ***( bPtr + 3 )**

3 is the **offset** to the pointer

Parentheses necessary because * is higher precedence than **+**

**Address &b[3]** can be written as: **bPtr + 3**

## POINTER OFFSET NOTATION EXAMPLE, CONTINUED

**int b[5];**

**int *bPtr;**

**bPtr = b;**

Can also use the array's name as the pointer:

***(b + 3)** refers to element b[3]

But **b += 3** is a compilation error

Attempts to modify the value of the built-in array's name with pointer arithmetic

## POINTER SUBSCRIPT NOTATION

**int b[5];**

**int *bPtr;**

**bPtr = b;**

Pointers can be subscripted just like built-in arrays

**bPtr[1]** refers to **b[1]**

## POINTER NOTATION BEST PRACTICE

Use built-in array notation instead of pointer notation for clarity

Fig08_17.cpp demonstrates the four notations just discussed:

- array subscript notation
- pointer/offset notation with the built-in array's name as a pointer
- pointer subscript notation
- pointer/offset notation with a pointer