C POINTERS, PART 2

Deitel 8th Edition, Chapter 7

TOPICS, PART 1

Memory Concepts – Review from Module 2

Pointers

Accessing Data in Variables with and without Pointers

Review of Functions

Using Pointers – Functions with Output Parameters

Functions with Output Parameters – Class Demo

TOPICS, PART 2

Brief Review from Last Module – Memory Usage and Functions

Principle of Least Privilege

Using the **const** Qualifier with Data & with Pointers

Pointers and Arrays

Pointer Arithmetic

Pointer Offset & Index Notation

Other Pointer Arithmetic Notes

Arrays and Functions

Brief Review from Last Module – Memory Usage and Functions

We Saw Three Different Examples

Function with input parameter

- Copy of a variable's **value** sent into function via **input parameter**

Function with input parameter and a return value

- Copy of a variable's **value** sent into function via **input parameter**
- Copy of function's **result** sent back from function via **return value**

Function with output parameter

- Copy of a variable's **address** sent into function via **output parameter**
- Function's **result** stored directly in original variable, via its **address**

PRINCIPLE OF LEAST PRIVILEGE

Principle of Least Privilege – Types of Access?

In general, give code only enough access to data to accomplish its task, but no more

What kinds of access are there?

Principle of Least Privilege – Types of Access

In general, give code only enough access to data to accomplish its task, but no more

What kinds of access are there?

Read data

Change data

Principle of Least Privilege w/Functions 1

Give function only enough access to the data in its parameters to accomplish its specified task, but no more

Example – a function needs an array as an argument, and only needs to print the contents of the array.

What should the arguments be, and how much access should be allowed to those arguments?

Principle of Least Privilege w/Functions 2

Example – a function that only **reads** an array:

1. Needs to know the size of array – but the size won't change
2. Needs to be able to read the data in the array, not change the data in the array

So the array, and the size value passed in should be declared as **read-only** to ensure that they're not accidentally modified

CONST **QUALIFIER = READ-ONLY ACCESS**

To prevent unauthorized changing of data, use the **const** qualifier to declare the variable's contents as constant:

```
const int  readOnlyData = 20;
```

Must give a **const** variable a value at time of declaration.

Any attempt to change the value of a const variable will not compile:

```
readOnlyData = 125;
```

USING THE CONST QUALIFIER WITH DATA

WRITEABLE AND READ-ONLY VARIABLES

Declare a variable **const** if the data should not be changed

- Read-only = constant
- Writeable = non-constant

constData.c

Within main, or within any function, it's easy to control access to data

But what happens when data is passed to a function?

Writeable and Read-Only Function Parameters

constDataWithFunctions.c

Note function **computeSumWriteableDataParameter**

To prevent the function from changing the data passed to it, declare the parameter **const**

USING THE CONST QUALIFIER WITH POINTERS

Remember What a Pointer Is…

A pointer is also just a variable that can store data

What kind of data does it store?

The Data Stored in a Pointer is an Address

That address can refer to data is writeable (non-constant) or read-only (constant)

So, we can declare *pointer variables* using **const**

And we can point a pointer at constant or non-constant data

4 Combinations of a Pointer & the Data it Points To

Non-constant pointer to non-constant data

Constant pointer to non-constant data

Non-constant pointer to constant data

Constant pointer to constant data

Non-Constant Pointer to Non-Constant Data

Highest level of data access – both data and pointer can be modified

Data can be modified:

- **Directly** via the variable's name, or
- **Indirectly**, by dereferencing the pointer to the variable

Pointer can be modified:

- To store the address of a different variable

NCPointerNCData.c

Constant Pointer to Non-Constant Data

Data can be modified; pointer cannot be modified

Data can be modified:

- **Directly** via the variable's name, or
- **Indirectly**, by dereferencing a pointer to the variable

Pointer cannot be modified

- It always points to the address assigned at initialization

constPointers.c

constPointersWithFunctions.c

Non-Constant Pointer to Constant Data

Data cannot be modified; pointer can be modified

Data cannot be modified

- Directly or indirectly

Pointer can be modified:

- To store the address of a different variable (of the appropriate data type)
- Note the syntax of the pointer declaration on line 10

NCPointerConstData.c

Constant Pointer to Constant Data

Lowest level of data access – neither data or pointer can be modified

Data:

- At that memory location cannot be modified

Pointer:

- Always points to the same memory location

constPointersConstData.c

4 Combinations of Pointers & Data – Summary

Pointer	Data	Example Declaration
Non-constant	Non-constant	**int** nonConstantData = 10; **int** * nonConstantPtr = &nonConstantData;
Constant	Non-constant	**int** nonConstantData = 10; **int** * **const** constantPtr = &nonConstantData;
Non-constant	Constant	**const int** constantData = 10; **int** * nonConstantPtr = &constantData;
Constant	Constant	**const int** constantData = 10; **const int** * **const** constantPtr = &constantData;

POINTERS AND ARRAYS

Relationship between Pointers and Arrays

Arrays and pointers often may be used interchangeably

Array name can be thought of as a **constant pointer**

Pointers can be used to do any operation involving array indexing

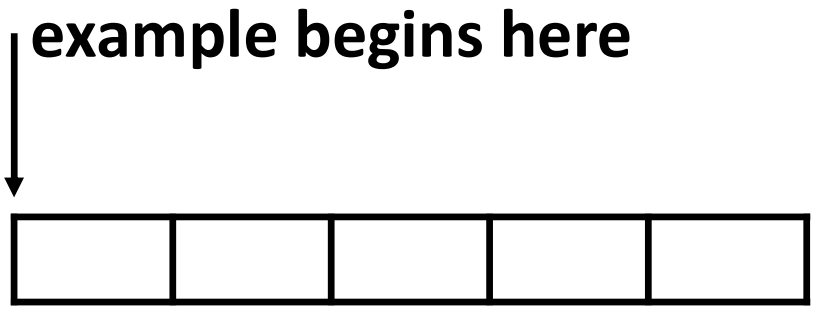
Array indexing

- Accessing the value of an array element using its index

DECLARING AN ARRAY – EXAMPLE

The name of this array is **example**

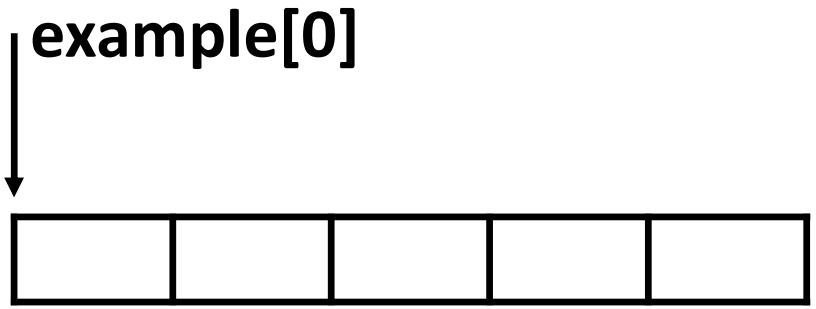
int example[5];



DECLARING AN ARRAY – EXAMPLE 1

The first element of the array is example[0]

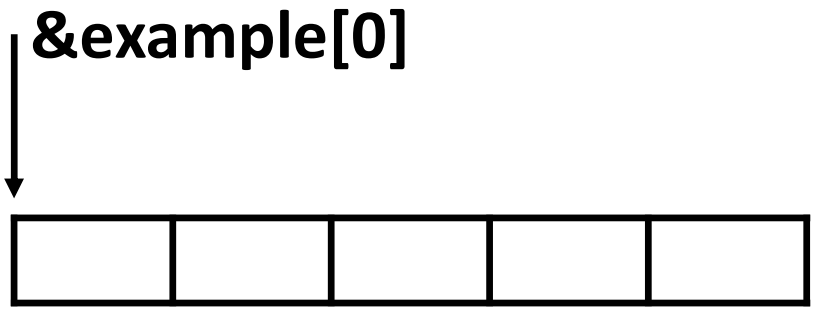
int example[5];



DECLARING AN ARRAY – EXAMPLE 2

Its address is &example[0]

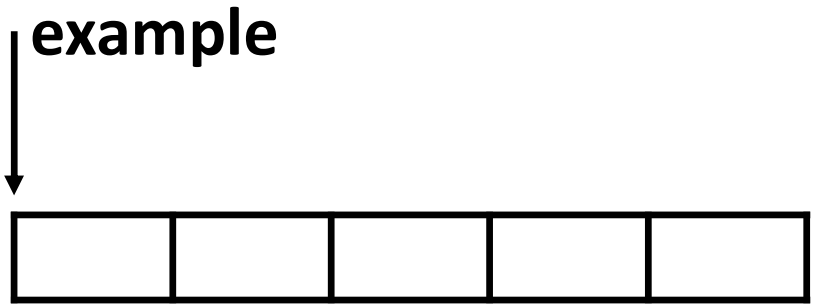
int example[5];



DECLARING AN ARRAY – EXAMPLE 3

The name of the array also indicates the beginning

int example[5];

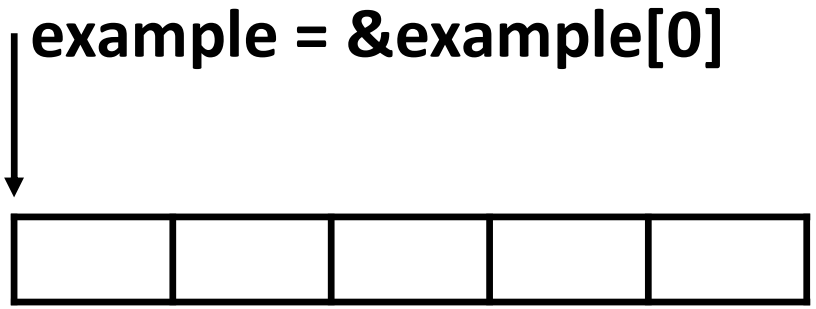


DECLARING AN ARRAY – EXAMPLE 4

So the name of the array is a pointer to the first element of the array,

Which is also the beginning of the array

int example[5];



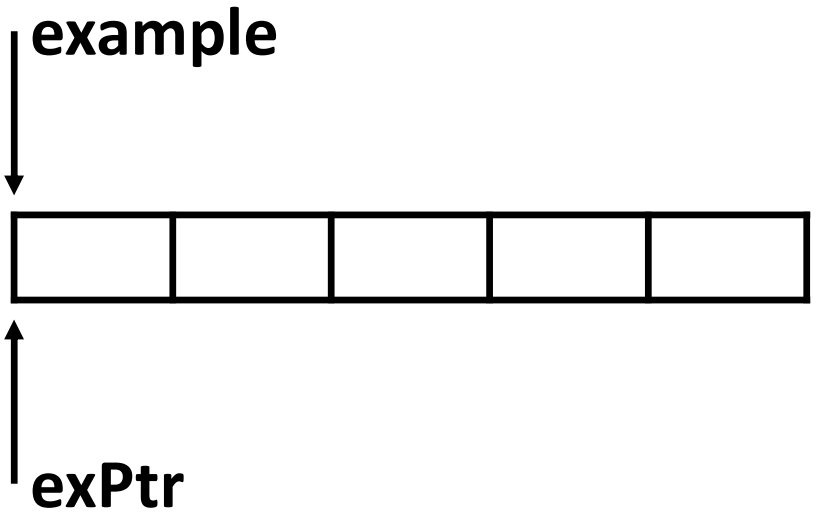
Relationship between Pointers and Arrays Example 1

Can point a pointer at an array by assigning it the name of the array

```
int example[5];

int * exPtr;

exPtr = example;
```



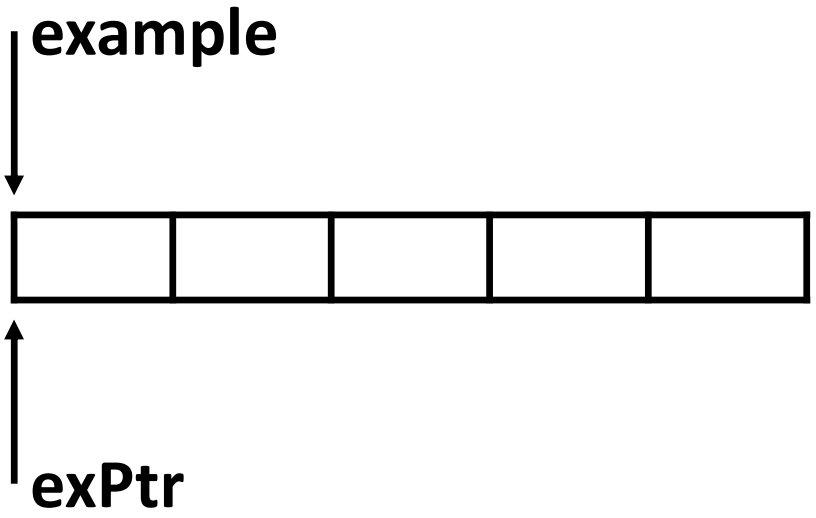
Relationship between Pointers and Arrays Example 2

Or by assigning it the address of the first element in the array

```
int example[5];

int * exPtr;

exPtr = &example[0];
```



POINTER ARITHMETIC

POINTER EXPRESSIONS & POINTER ARITHMETIC

Pointers are valid operands in arithmetic, assignments, comparisons

- Remember, they store numerical values
- But only makes sense when the pointer points to an **element of an array**

Pointer arithmetic is machine and compiler dependent

- Depends on size of data types

VALID ARITHMETIC OPERATIONS ON A POINTER

Increment (++)

Decrement (--)

Add an integer to a pointer (using + or +=)

Subtract an integer from a pointer (using - or -=)

One pointer may be subtracted from another

- Both must point to elements in the same array

Array Addresses

First, set a pointer to point at an array:

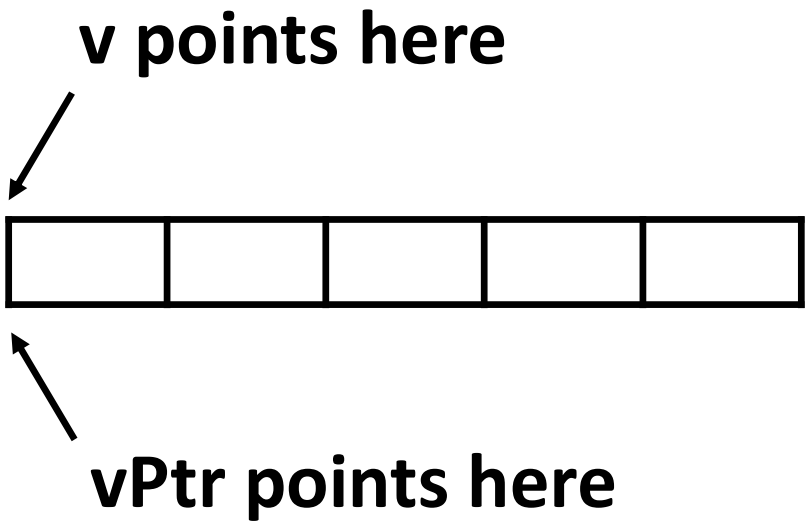
```
int v[5];

int * vPtr;


vPtr = v;

   or

vPtr = &v[0];
```

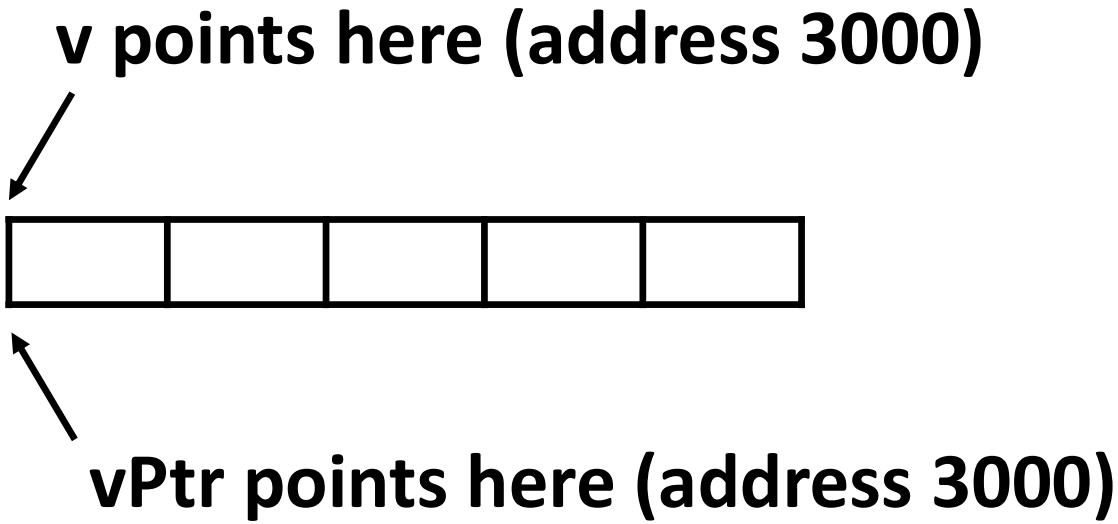


ARRAY ADDRESSES 1

Let's say the address of the array is 3000

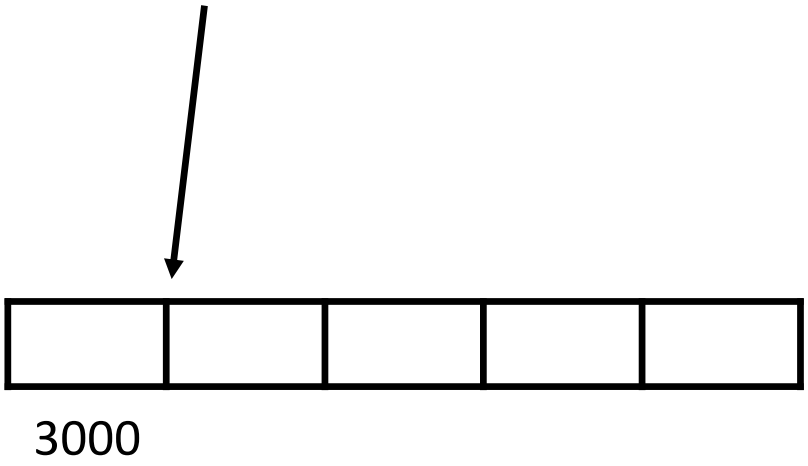
So v's value is 3000

vPtr's value is 3000



ARRAY ADDRESSES 2

What is the address of v[1]?



ARRAY ADDRESSES 3

The address of v[1] depends on the size of each element in the array

What is the size of each element?

ARRAY ADDRESSES 4

The size of each element depends on the data type

Sine this is an integer array, the size of each element is 4 bytes

3000

Adding an Integer to a Pointer 1

Example:

3000 + 2 = ?

In math class, what do these values represent?

In C class, what do these values represent?

Hint – not the same thing…

Adding an Integer to a Pointer 2

Example:

3000 + 2 = ?

In math class, these values represent whole numbers, so the result would be 3002

In C class, these values represent **addresses**

Specifically, addresses within an array…

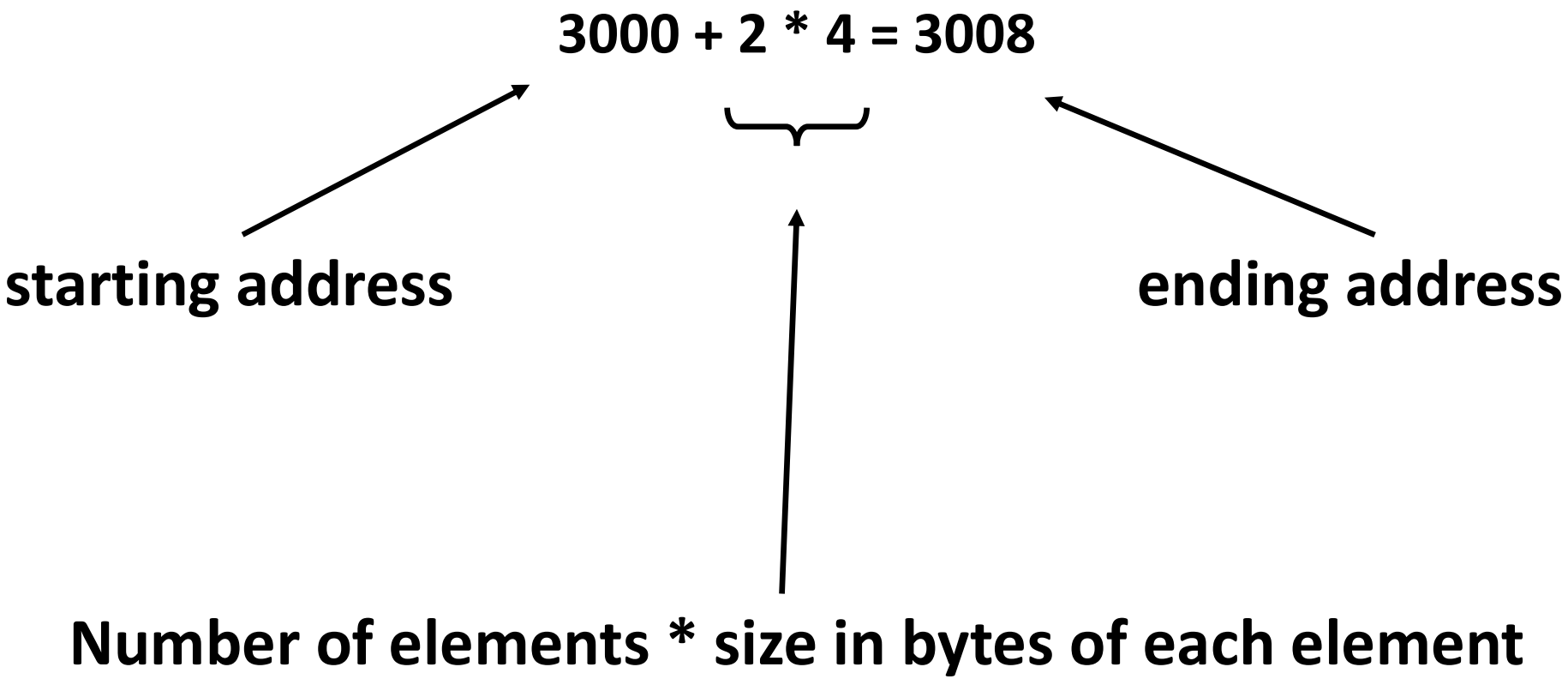
Adding an Integer to a Pointer 3

In C class:

3000 + 2 = 3008

Why?

Adding an Integer to a Pointer 4

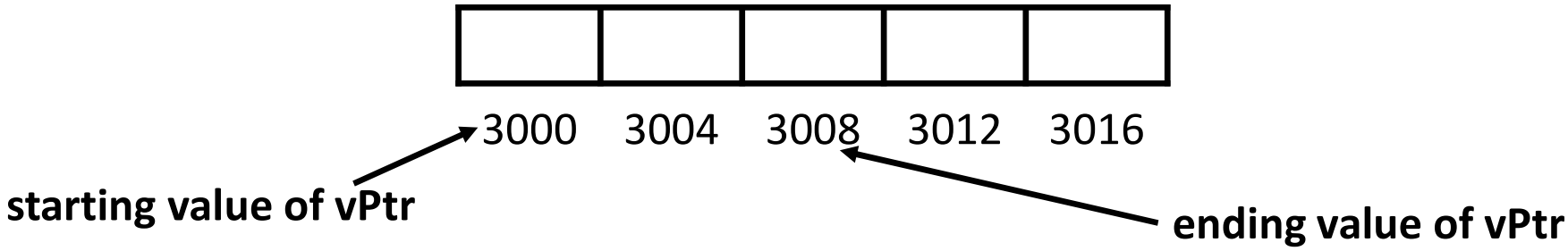


Adding an Integer to a Pointer 5

So if vPtr was 3000:

vPtr += 2; produces 3008

In the array v, vPtr would now point to v[2]



COMMON PROGRAMMING ERROR

Using pointer arithmetic on a pointer that does not refer to an element in an array.

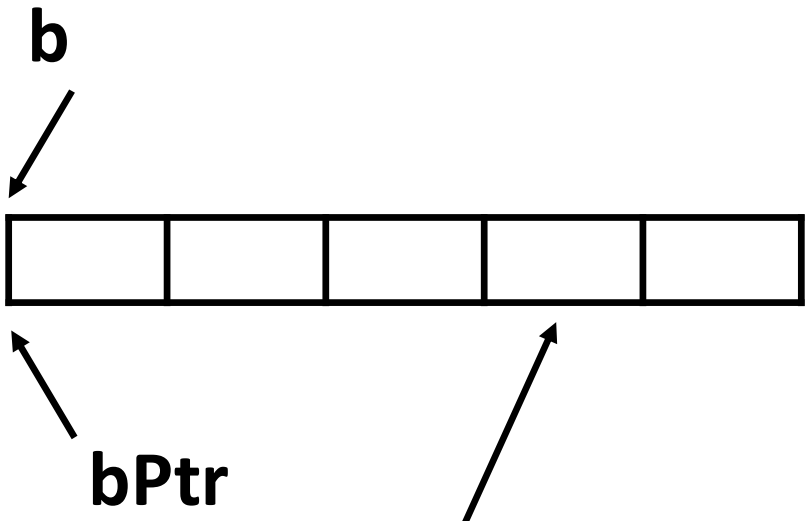
POINTER OFFSET & INDEX NOTATION

POINTER/OFFSET NOTATION 1

If a pointer points to an array, the value of an element in the array can be accessed with a pointer expression:

int b[5];

bPtr = b;



***(bPtr + 3)** can be used to access this element's value:

POINTER/OFFSET NOTATION 2

***(bPtr + 3)**

3 is the **offset** to the pointer

Offset value is identical to the array index

Parentheses are necessary because the precedence of * is higher than the precedence of +

POINTER/OFFSET NOTATION – INCORRECT SYNTAX

***bPtr + 3**

Without the parentheses, the above expression would add 3 to the value of the expression *bPtr

So 3 would be added to b[0], assuming bPtr points to the beginning of the array

POINTER/INDEX NOTATION

A pointer to an array can reference elements using []

- Just like the name of an array

If bPtr has the value b, then bPtr[1] refers to array element b[1]

CANNOT MODIFY AN ARRAY NAME WITH POINTER ARITHMETIC

b += 3 is invalid

The missing * is an error

Demonstrating Pointer Indexing and Offsets

Fig07_20.c

Uses 4 methods for referring to array elements:

- Array indexing
- Pointer/offset with array name as pointer
- Pointer indexing
- Pointer/offset with a pointer

Other Operations w/Pointer Arithmetic

Other Pointer Arithmetic Notes

If a pointer is being incremented or decremented by one, the increment (++) and decrement (--) operators can be used

Pointer variables may be subtracted from one another

- Undefined unless performed on an array

A pointer can be assigned to another pointer if both have the same type

POINTER TO VOID

void *

- Generic pointer that can represent any pointer type

Pointer to void cannot be dereferenced

Simply points at a memory location for an unknown data type

COMPARING POINTERS

Pointer comparisons compare the addresses stored in the pointers

Pointers can be compared using equality and relational operators

Meaningless unless the pointers point to elements of the same array

ARRAYS AND FUNCTIONS

USE & AND * TO ACCOMPLISH PASS-BY-REFERENCE

Remember, arrays are not passed using operator **&** because C automatically passes the starting location in memory of the array:

The name of an array is equivalent to &arrayName[0]

When the address of a variable is passed to a function, the indirection operator * may be used in the function to modify the value at that location in the caller's memory

Functions That Receive One-Dimensional Arrays

Compiler does not differentiate between a function that receives a pointer and one that receives a one-dimensional array

Function must “know” when it’s receiving an array or simply a single variable pointer

Function parameter for a one-dimensional array using the array notation **int b[]** converted by compiler to the pointer notation **int *b**