Arrays and Pointers

CMSC 202

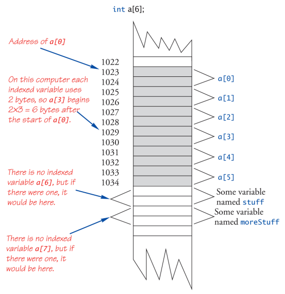
Arrays

- An array is a *collection* of *related data items* all having the *same data type*.
- Arrays can be of any data type we choose.
- Arrays are **static** in that they remain the same size throughout program execution.
- An array's data items are stored contiguously in memory
- To declare an array called "numbers" consisting of 5 integers, you would use:

```
int numbers[5];
```

An Array in Memory

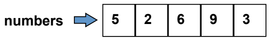
Display 5.2 An Array in Memory



Copyright © 2012 Pearson Addison-Wesley.
All rights reserved.

Array Declaration and Initialization

- This declaration sets aside a block of memory that is big enough to hold five integers:
  `int numbers[5];`
- It does not initialize the memory locations; they contain garbage data.
- Initializing an array may be done with an *array initializer*, as in :
  `int numbers[5] = { 5, 2, 6, 9, 3 };`



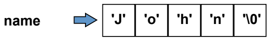
Auto-Initializing Arrays

- If fewer values than size supplied:
  - Fills from beginning
  - Fills remainder with zero of the array's base type
- If array-size is left out
  - Declares array with size required based on number of initialization values
  - Example:
    `int b[] = {5, 12, 11};`
    - Allocates array b to size 3

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

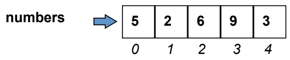
Array Declaration and Initialization

- A special case is an *array of chars*:
  `char name[5] ;`
- As mentioned earlier, a C-string is in fact an array of chars, usually ending in a 0 byte.
  - The 0-valued byte at the end is called a *null terminator*.
  - Strings do not necessarily have to be null-terminated.
- Initializing a char array may be done the usual way, as in:
  `char name[5] = {'J', 'o', 'h', 'n', 0 };`
  …or with a string constant:
  `char name[5] = "John";`



Accessing Array Elements

- You use the standard bracketed subscript notation to access elements in an array:



```
cout << "The third element is " << numbers[2];
```

would give the output

The third element is 6

- Subscripts are integers and always begin at zero.

Accessing Array Elements (con't)

- A subscript can also be any expression that evaluates to an integer.

```
numbers[(a + b) * 2];
```

- Caution! C++ does not do bounds checking for simple arrays, so you must ensure you are staying within bounds

Defined Constant as Array Size

- Always use defined/named constant for array size
- Example:

```
const int NUMBER_OF_STUDENTS = 5;
int score[NUMBER_OF_STUDENTS];
```

- Improves readability
- Improves versatility
- Improves maintainability

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

Arrays in Functions

- As arguments to functions
  - – Indexed variables
    - An individual element of an array can be a function parameter
  - – Entire arrays
    - All array elements can be passed as one entity
- As return value from function
  - – Can be done → chapter 10

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

Indexed Variables as Arguments

- Indexed variable handled same as simple variable of base type
- Given this function declaration:
  `void myFunction(double par1);`
- And these declarations:
  `int i;  double n, a[10];`
- Can make these function calls:

```
myFunction(i);     // i is converted to double
myFunction(a[3]);  // a[3] is double
myFunction(n);     // n is double
```

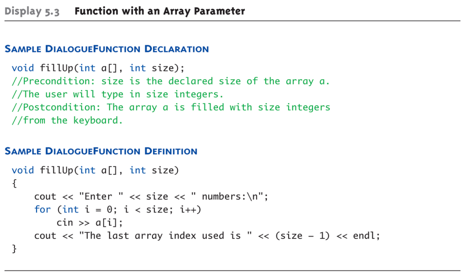
Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

Entire Arrays as Arguments

- Formal parameter can be entire array
  - – Argument passed in function call is array name
  - – Called an *array parameter*
- Send size of array as well
  - – Typically done as second parameter
  - – Simple int type formal parameter

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

Entire Array as Argument Example: **Display 5.3** Function with an Array Parameter



Display 5.3 **Function with an Array Parameter**

SAMPLE DIALOGUEFUNCTION DECLARATION

```
void fillUp(int a[], int size);
//Precondition: size is the declared size of the array a.
//The user will type in size integers.
//Postcondition: The array a is filled with size integers
//from the keyboard.
```

SAMPLE DIALOGUEFUNCTION DEFINITION

```
void fillUp(int a[], int size)
{
    cout << "Enter " << size << " numbers:\n";
    for (int i = 0; i < size; i++)
        cin >> a[i];
    cout << "The last array index used is " << (size – 1) << endl;
}
```

Copyright © 2012 Pearson Addison-Wesley.
All rights reserved.

Entire Array as Argument Example

- Given previous example:
- In some `main()` function definition, consider this call:

```
int score[5], numberOfScores = 5;
fillUp(score, numberOfScores);
```

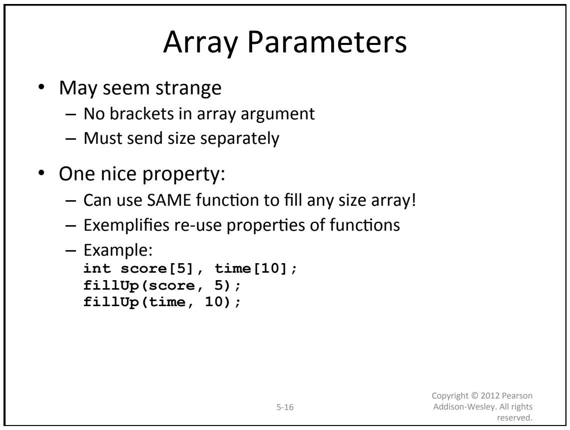
    - 1st argument is entire array
    - 2nd argument is integer value
  - Note: **no brackets in array argument**!

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

Array as Argument: How?

- What's really passed?
- Think of array as 3 components:
  - Address of first indexed variable (`arrName[0]`)
  - Array base type
  - Size of array
- Only 1st piece is passed!
  - Just the beginning address of array
  - Very similar to pass-by-reference

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

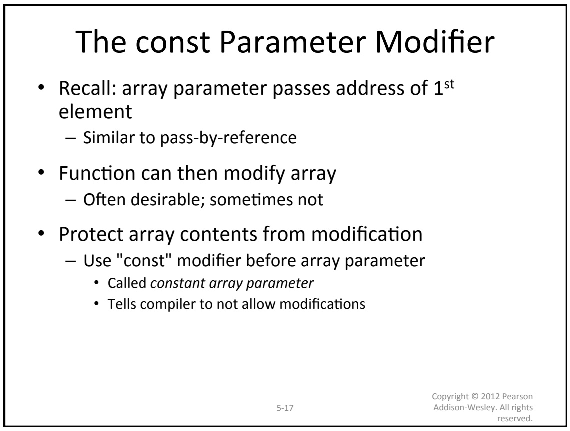


Array Parameters

- May seem strange
  - No brackets in array argument
  - Must send size separately
- One nice property:
  - Can use SAME function to fill any size array!
  - Exemplifies re-use properties of functions
  - Example:

```
int score[5], time[10];
fillUp(score, 5);
fillUp(time, 10);
```

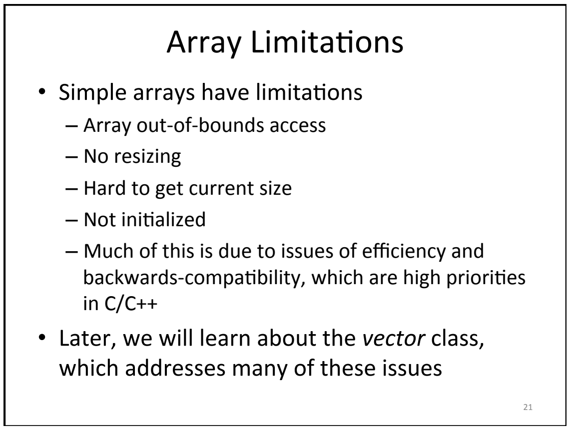
Copyright © 2012 Pearson Addison-Wesley. All rights reserved.



The const Parameter Modifier

- Recall: array parameter passes address of 1st element
  - Similar to pass-by-reference
- Function can then modify array
  - Often desirable; sometimes not
- Protect array contents from modification
  - Use "const" modifier before array parameter
    - Called *constant array parameter*
    - Tells compiler to not allow modifications

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.



Array Limitations

- Simple arrays have limitations
  - Array out-of-bounds access
  - No resizing
  - Hard to get current size
  - Not initialized
  - Much of this is due to issues of efficiency and backwards-compatibility, which are high priorities in C/C++
- Later, we will learn about the *vector* class, which addresses many of these issues

Basic Pointers

- A *pointer* is a variable that contains the address of a variable.
  - Address can be thought of as an integer value
  - Typical machines have 32- or 64-bit addresses
- Pointers are necessary for various reasons:
  - In C, allows functions to modify arguments
  - To access dynamic objects (more on that later…)
  - To pass an array or complex object to a function efficiently

Basic Pointers

- We can do this in both directions:
  - Put an address into a variable and tell the processor to do an operation on the value in the location *pointed to* by the first value
  - Given a variable (again, a memory location), take it's memory address in RAM, which is a number, and store this number inside some other variable
    - This requires the cooperation of the compiler, which decides, and therefore knows, where the various variables are being stored in RAM.

Pointer Introduction

- We use the '*' (*points to*) and '&' (*address of*) unary operators to work with pointers.
- Note distinction between a pointer – which is a numerical address and therefore always a certain size (number of bytes) on a given computer—and the type of data it points to, which can be of different sizes.

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

Pointer Variables

- Pointers are "typed"
  - Can store pointer in variable
  - Not int, double, etc.
    - Instead: A POINTER to int, double, etc.
- Example:
  `double *p;`
  - p is declared a "pointer to double" variable
  - Can hold pointers to variables of type double
    - Not other types (unless typecast, but could be dangerous)

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

Declaring Pointer Variables

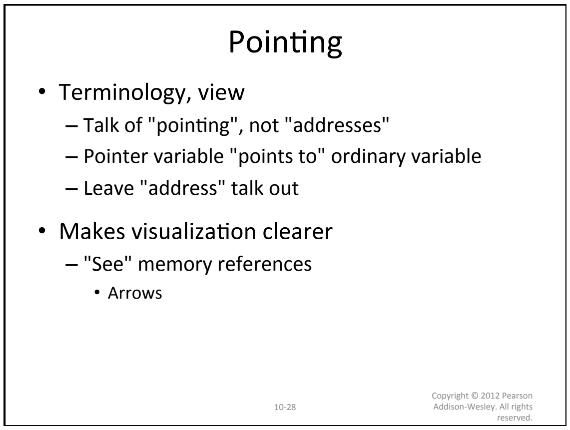
- Pointers declared like other types
  - Add "*" before variable name
  - Produces "pointer to" that type
- "*" must be before each variable
- **`int *p1, *p2, v1, v2;`**
  - p1, p2 hold pointers to int variables
  - v1, v2 are ordinary int variables

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

Addresses and Numbers

- Pointer is an address
- Address is an integer
- Pointer is NOT an integer!
  - Not crazy → abstraction!
- C++ forces pointers to be used as addresses
  - Cannot be used as numbers
  - Even though it "is a" number

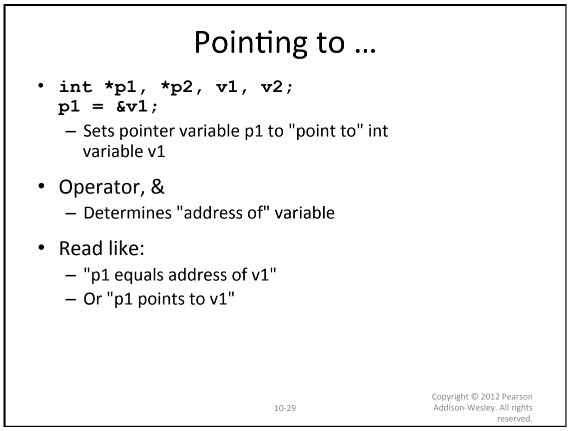
Copyright © 2012 Pearson Addison-Wesley. All rights reserved.



Pointing

- Terminology, view
  - Talk of "pointing", not "addresses"
  - Pointer variable "points to" ordinary variable
  - Leave "address" talk out
- Makes visualization clearer
  - "See" memory references
    - Arrows

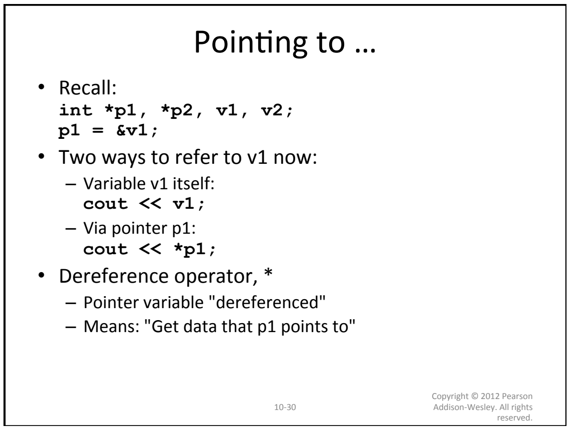
Copyright © 2012 Pearson Addison-Wesley. All rights reserved.



Pointing to …

- **`int *p1, *p2, v1, v2;`**
  `p1 = &v1;`
  - Sets pointer variable p1 to "point to" int variable v1
- Operator, &
  - Determines "address of" variable
- Read like:
  - "p1 equals address of v1"
  - Or "p1 points to v1"

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.



Pointing to …

- Recall:
  `int *p1, *p2, v1, v2;`
  `p1 = &v1;`
- Two ways to refer to v1 now:
  - Variable v1 itself:
    `cout << v1;`
  - Via pointer p1:
    `cout << *p1;`
- Dereference operator, *
  - Pointer variable "dereferenced"
  - Means: "Get data that p1 points to"

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

"Pointing to" Example

- Consider:
```
v1 = 0;
p1 = &v1;
*p1 = 42;
cout << v1 << endl;
cout << *p1 << endl;
```
- Produces output:
```
42
42
```
- p1 and v1 refer to same variable

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

& Operator

- The "address of" operator
- Also used to specify call-by-reference parameter (more on this later)
  - No coincidence!
  - Recall: call-by-reference parameters pass "address of" the actual argument
- Operator's two uses are closely related

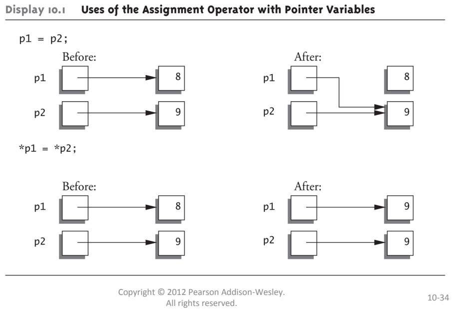
Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

Pointer Assignments

- Pointer variables can be "assigned":
```
int *p1, *p2;
p2 = p1;
```
  - Assigns one pointer to another
  - "Make p2 point to where p1 points"
- Do not confuse with:
```
*p1 = *p2;
```
  - Assigns "value pointed to" by p1, to "value pointed to" by p2

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

Pointer Assignments Graphic: **Display 10.1** Uses of the Assignment Operator with Pointer Variables



Copyright © 2012 Pearson Addison-Wesley.
All rights reserved.

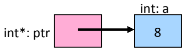
10-34

Pointer Operator Summary

- &
  - Address of pointee
  - Syntax:
    - `type* ptr = &variable;`
    - `ptr = &variable2;`
- *
  - Dereferencing, Value of pointee
  - Syntax:
    - `*ptr = value;`
    - `variable = *ptr;`
- =
  - Assignment, point to something else
  - Syntax:
    - `ptrA = ptrB;`

- Examples:
  - `int a = 3;`
  - `int* ptr = &a;`
  - `*ptr = 8;`

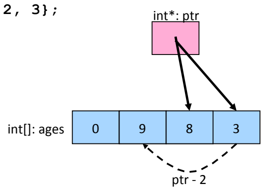
  - `int b = 5;`
  - `int* ptr2 = &b;`
  - `ptr = ptr2;`



Arrays and Pointer Arithmetic

- Tricky stuff…
  - Arrays are simply a kind of pointer
  - Points to first item in collection
  - Index into array is "offset"
    - Example

```
int ages[4] = {0, 1, 2, 3};
int* ptr = &ages[2];
*ptr = 8;
ptr++;
*(ptr - 2) = 9;
```



Simulated "Pass by Reference"

- Some programming languages provide mechanism for called function to have direct access to variables used in the calling function
- We can simulate this by using pointers (see following slide)
- C++ added true "call by reference" – we will see this later on

Simulated "Pass by Reference"

- Calling function:

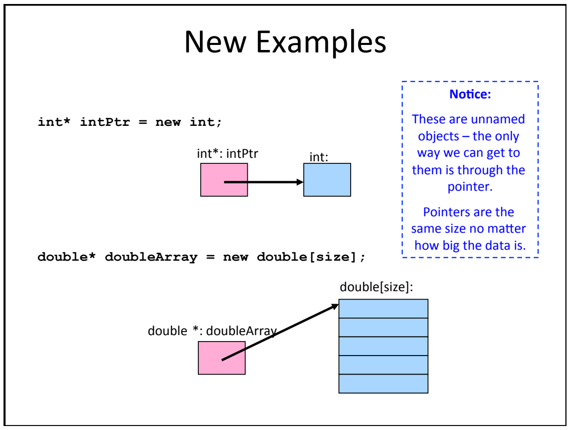
```
int x = 1;

// pass in reference to (actually, pointer to)
// our argument variable "x"
add1(&x);
cout << x;  // will output 2!
```

- Called function:

```
void add1(int *var) {
    *var = *var + 1;
}
```

Dynamic Memory

- **`new`** creates a "new" variable or array
  - Works with primitives
  - Works with class-types (more on this later)
- Syntax:
  - `type *ptrName = new type;`
- Example:
  - `int *newInt = new int;`
  - `double doubleArray = new double[size];`



New Examples

`int* intPtr = new int;`

`double* doubleArray = new double[size];`

Notice:

These are unnamed objects – the only way we can get to them is through the pointer.

Pointers are the same size no matter how big the data is.

Deletion of Objects

- delete
  - Called on the pointer to an object
  - Works with primitives & class-types
- Syntax:
  - delete ptrName;
- Example:
  - `delete intPtr;`
  - `intPtr = NULL;`
  - `delete [] doubleArray;`
  - `doubleArray = NULL;`

Set to NULL so that you can use it later – protect yourself from accidentally using an uninitialized pointer.