Pointers

CS 16: Solving Problems with Computers I
Lecture #15

Ziad Matni
Dept. of Computer Science, UCSB

Announcements

- Lab #8 is due on Monday, 11/21 at 8 AM
- Homework #14 is due on Tuesday, 11/22

Lecture Outline

CH. 9

- Introduction to Pointers
- Dynamic Arrays

Memory Addresses

- Consider the integer variable **num** that holds the value 42
- **num** is assigned a place in memory. In this example the ***address*** of that place in memory is 1F
  - Generally, memory addresses use *hexadecimals*

	Address	***Data***
	1D	
1 byte	1E	
num	1F	42
	20	
	21	
	22	

- The address of a variable can be obtained by putting the ampersand character (&) before the variable name.
  - **&** is called the ***address-of*** operator
  - Example: `num_add = #`

    will result in **num_add** to hold the value 1F

Memory Address

- Recall: num = 42 and num_add = &num = 1F
- Now, let's make **bar = num**
  - Another variable, **bar**, now is assigned the same value that's in num (42)
  - Note the difference between **bar** and **num_add**

- The variable **bar** will be assigned an address
  - Let's say, that address is **77**
  - Keep in mind, by default, we have no control over address assignments
    - This is just for illustrative purposes…

- The variable that stores the address of another variable (like **num_add**) is what in C++ is called a ***pointer***.

Dereference Operator (*)

- Pointers "point to" the variable whose address they store
- Pointers can access the variable they point to directly
- Done by preceding the pointer name with the **dereference operator (*)**
  - The operator itself can be read as "value pointed to by"

Recall: num = 42 and num_add = &num = 1F

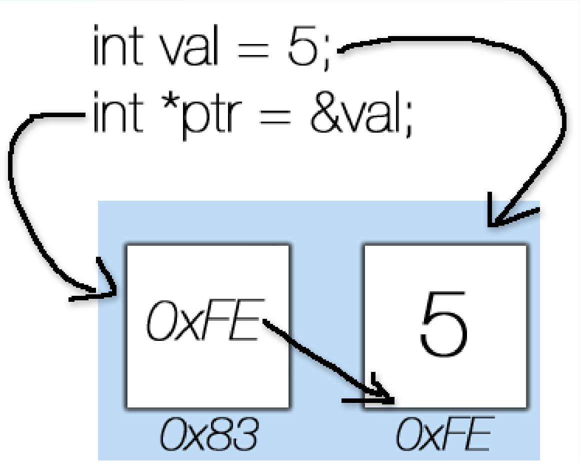
- So, while **num_add = 1F**, ***num_add = 42**

Pointers

- A pointer is the memory address of a variable
- Memory addresses can be used as names for variables
  - If a variable is stored in three memory locations, the address of the first can be used as a name for the variable
  - When a variable is used as a call-by-reference argument, it’s the actual address in memory that is passed

Pointers Tell Us (or the Compiler) Where To Find A Variable

- An address that is used to tell where a variable is stored in memory is a pointer
  - Pointers "point" to a variable by telling where the variable is located



Declaring Pointers

- Pointer variables must be declared to have a **pointer** type

- Example: To declare a pointer variable **p** that can "point" to a variable of type double:

```
double  *p;
```

- The asterisk (*) identifies **p** as a pointer variable

Multiple Pointer Declarations

- To declare multiple pointers in a statement, use the asterisk ***before*** each pointer variable

- Example:

```
int *p1, *p2, v1, v2;
```

p1 and p2 point to variables of type int
v1 and v2 are variables of type int

The address-of Operator

- The **&** operator can be used to determine the address of a variable which can be assigned to a pointer variable

- Example: `p1 = &v1;`

  p1 is now a pointer to v1
  v1 can be called v1
  or “the variable pointed to by p1”

Another Note on the Dereferencing Operator (*)

- C++ uses the ***** operator in yet another way with pointers

- The phrase "*The variable pointed to by p*" is translated into C++ as ***p**

- **p** is said to be *dereferenced*

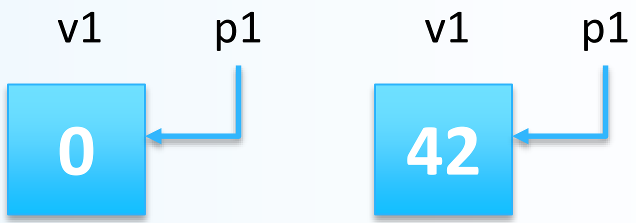
A Pointer Example

```
v1 = 0;
p1 = &v1;
*p1 = 42;
cout << v1 << endl;
cout << *p1 << endl;
```

v1 and *p1 now refer to the same variable

output:

42
42



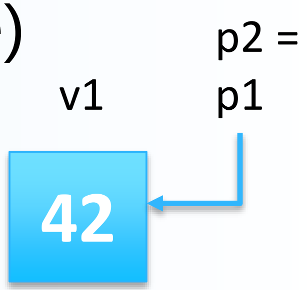
Pointer Assignment

- The assignment operator **=** is used to assign the value of one pointer to another

Example: If p1 still points to v1 (previous slide)

then the statement

p2 = p1;



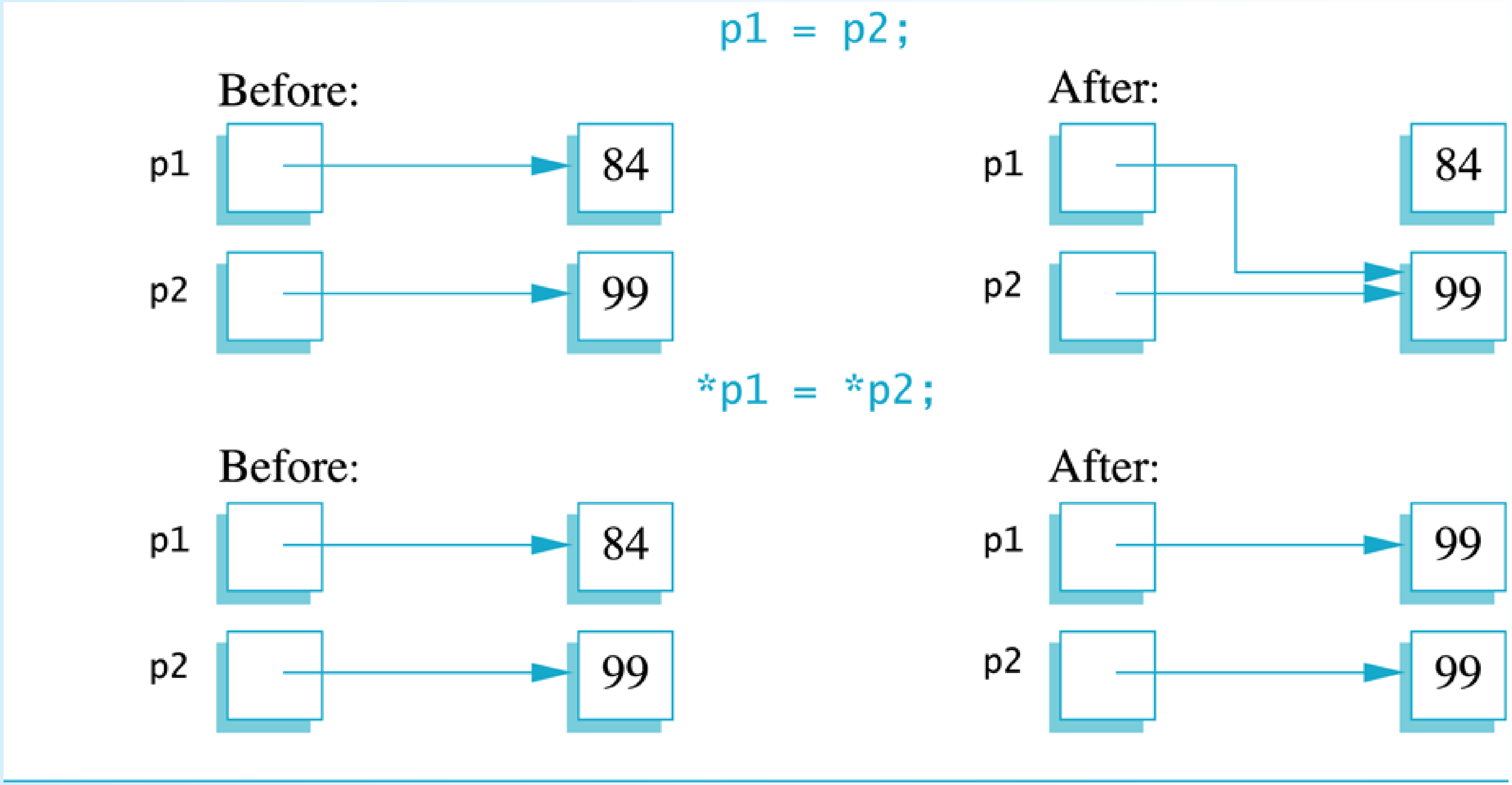
causes ***p2**, ***p1**, and **v1** all to name the same variable

Caution! Pointer Assignments

- Some care is required making assignments to pointer variables

```
 p1 = p3;     // changes the location that p1 "points" to

*p1 = *p3;    // changes the value at the location that
              // p1 "points" to
```

Uses of the Assignment Operator on Pointers



The **new** Operator

- Using pointers, variables can be manipulated even if there is no identifier for them

- To create a pointer to a new “nameless” variable of type int:

```
p1 = new int;
```

- The new variable is referred to as ***p1**

- ***p1** can be used anyplace an integer variable can

Example:

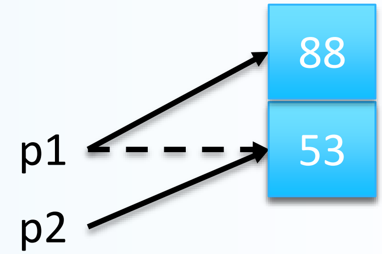
```
cin >> *p1;
*p1 = *p1 + 7;
```

Dynamic Variables

- Variables created using the **new** operator are called ***dynamic variables***
- *Dynamic variables* are created and destroyed while the program is running
  - We don't have to bother with naming them, just their pointers

Basic Pointer Manipulations

```
//Program to demonstrate pointers and dynamic variables.
#include <iostream>
using namespace std;

int main( )
{
    int *p1, *p2;

    p1 = new int;
    *p1 = 42;
    p2 = p1;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    *p2 = 53;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    p1 = new int;
    *p1 = 88;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    cout << "Hope you got the point of this example!\n";
    return 0;
}
```



Sample Dialogue

```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
Hope you got the point of this example!
```

Basic Memory Management

- An area of memory called the **freestore** or the **heap** is reserved for dynamic variables
  - New dynamic variables use memory in the freestore
  - If all of the **freestore** is used, calls to **new** will fail
    - So you need to manage your unused dynamic variables…
- Unneeded memory can be recycled
  - When variables are no longer needed, they can be deleted and the memory they used is returned to the **freestore**

The **delete** Operator

- When dynamic variables are no longer needed, **delete** them to return memory to the **freestore**

- Example:

  delete p;

- The value of p is now undefined and the memory used by the variable that **p** pointed to is back in the **freestore**

Dangling Pointers

- Using **delete** on a pointer variable destroys the dynamic variable pointed to
- If another pointer variable was pointing to the dynamic variable, that variable is also now undefined
- Undefined pointer variables are called ***dangling pointers***
  - Dereferencing a dangling pointer (*p) is usually disastrous

Automatic Variables

- Variables declared in a function are created by C++ and then destroyed when the function ends

- These are called ***automatic variables*** because their creation and destruction is controlled automatically

- However, the programmer must ***manually*** controls creation and destruction of **pointer** variables with operators **new** and **delete**

Type Definitions

- A name can be assigned to a type definition, then used to declare variables

- The keyword **typedef** is used to define new type names

- Syntax:

```
typedef Known_Type_Definition  New_Type_Name;
```

where, *Known_Type_Definition* can be any type

Defining Pointer Types

- To help avoid mistakes using pointers,
  define a pointer type name

- Example: `typedef int* IntPtr;`

Defines a new *type*, **IntPtr**, for pointer variables containing pointers to **int** variables

```
IntPtr p;
```

is now equivalent to saying: `int *p;`

Multiple Declarations Again

- Using our new pointer type defined as

```
typedef int* IntPtr;
```

- Prevents error in pointer declaration:
- For example, if you want to declare 2 pointers, instead of this:

```
int *p1, p2;
 // Careful! Only P1 is a pointer variable!
```

do this:

```
IntPtr p1, p2;
// p1 and p2 are both pointer variables
```

Pointer Reference Parameters

- A second advantage in using **typedef** to define a pointer type is seen in parameter lists

- Example:

```
void sample_function(IntPtr& pointer_var);
```

is less confusing than

```
void sample_function(int*& pointer_var);
```

Dynamic Arrays

Dynamic Arrays

A dynamic array is an array whose size is determined when the program is running, not when you write the program

Pointer Variables and Array Variables

- Array variables are actually pointer variables that point to the first indexed variable
  - Remember when calling an array in a function?
    - funcA(a) … not … funcA(a[ ])
  - Take, for instance:

```
int  a[10];
typedef int* IntPtr;
IntPtr p;
```

    - NOTE: Variables **a** and **p** are the same kind of variable

- Since **a** is a pointer variable that points to **a[0]**,

  p = a;

  causes **p** to point to the same location as **a**

Pointer Variables As Array Variables

```
int  a[10];
typedef int* IntPtr;
IntPtr p = a;
```

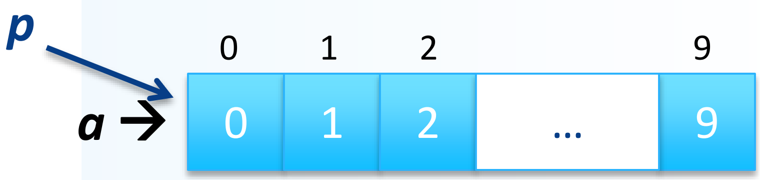
- Continuing with the previous example: Pointer variable **p** can be used as if it were an array variable
- So, p[0], p[1], …p[9] are all legal ways to use p
- Variable **a** can be used as a pointer variable except the pointer value in **a** cannot be changed
  - So, this is not legal:

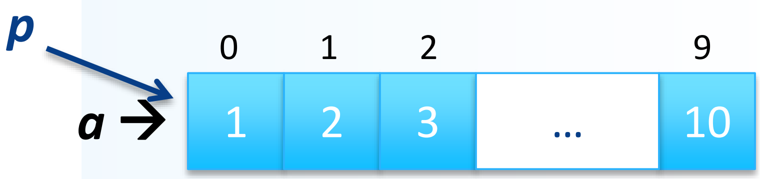
```
IntPtr p2;   // p2 is assigned a value
a = p2       // attempt to change a
```

Arrays and Pointer Variables

```
//Program to demonstrate that an array variable is a kind of pointer variable.
#include <iostream>
using namespace std;

typedef int* IntPtr;

int main( )
{
    IntPtr p;
    int a[10];
    int index;

    for (index = 0; index < 10; index++)
        a[index] = index;

    p = a;

    for (index = 0; index < 10; index++)
        cout << p[index] << " ";
    cout << endl;

    for (index = 0; index < 10; index++)
        p[index] = p[index] + 1;

    for (index = 0; index < 10; index++)
        cout << a[index] << " ";
    cout << endl;

    return 0;
}
```

Note that changes to the array p are also changes to the array a.





Output

```
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10
```

Creating Dynamic Arrays

- Normal arrays require that the programmer determine the size of the array *when the program is written*
  - What if the programmer estimates too large?
    - Memory is wasted
  - What if the programmer estimates too small?
    - The program may not work in some situations

- Dynamic arrays can be created with just the right size *while the program is running*

Are Dynamic Arrays *aka* Vectors?!

- Not exactly…
  - ***vector*** is an *implementation* of dynamic arrays

- The biggest difference is:
  - Vectors automatically increase their capacity
  - Dynamic arrays have to do this with **new** and **delete**

- The advantage of vectors is that they are well-defined and you don't have to worry about size changes, capacity adjustments in memory, etc…

Creating Dynamic Arrays

- Dynamic arrays are created using the **new** operator

- Example:
  To create an array of 10 elements of type double:

```
typedef double* DoublePtr;
DoublePtr d;
d = new double[10];
```

d can now be used as if it were an ordinary array!

Dynamic Arrays (cont.)

- Pointer variable d is a pointer to d[0]
- When finished with the array, it should be **delete**d to return memory to the **freestore**
  - Example: **delete [ ] d;**
  - The brackets tell C++ that a dynamic array is being deleted so it must check the size to know how many indexed variables to remove
  - Do not forget the brackets!
- Display 9.6 in the book has an example of use

Multidimensional Dynamic Arrays

- Example: Create a 3x4 multidimensional dynamic array

- Recall: view multidimensional arrays as arrays of arrays…
  - So a 3x4 array = 3-element array, each of which is a 4-element array

- First create a one-dimensional dynamic array
  - Start with a new definition:

```
typedef int* IntArrayPtr;
```

  - Now create a dynamic array of pointers named **m**:

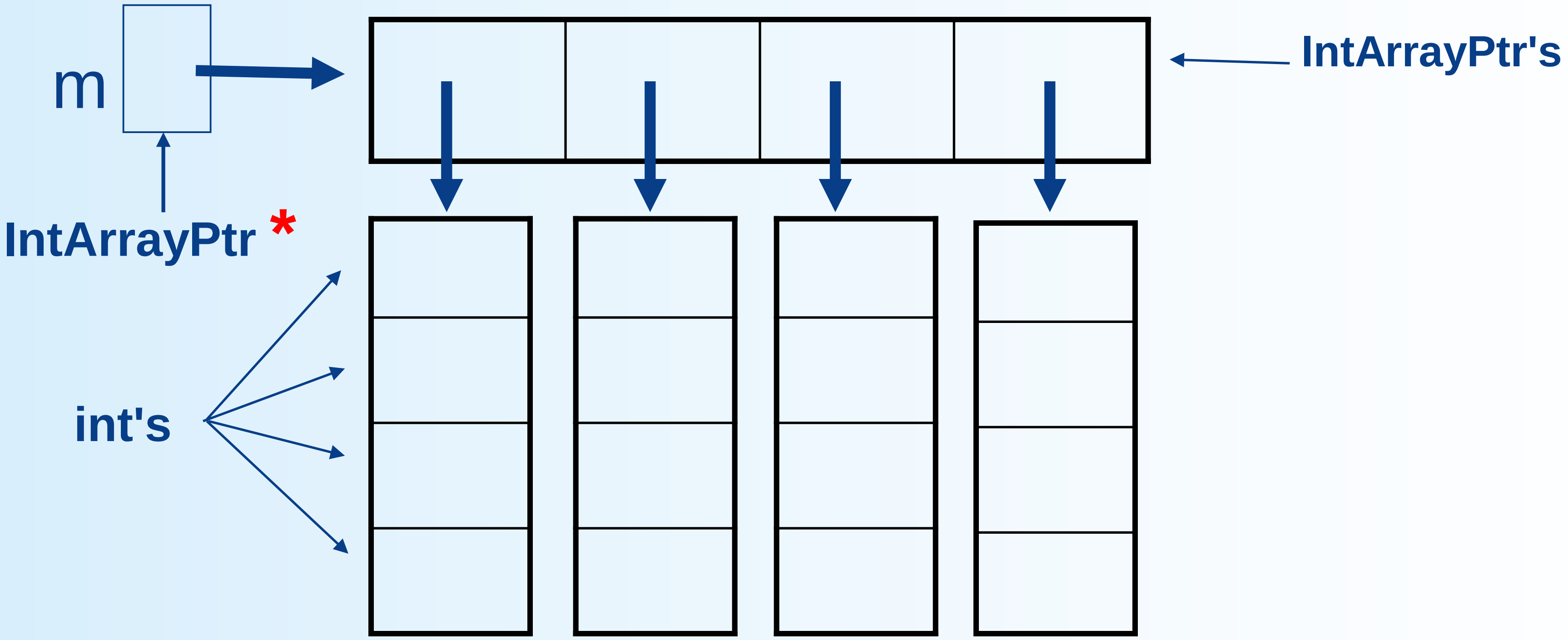
```
IntArrayPtr *m = new IntArrayPtr[3];
```

- For each pointer in **m**, create a dynamic array of int's

```
for (int i = 0; i < 3; i++)
    m[i] = new int[4];
```

A Multidimensional Dynamic Array

- The dynamic array created on the previous slide could be visualized like this:



Deleting Multidimensional Arrays

- To delete a multidimensional dynamic array
  - Each call to **new** that created an array must have a corresponding call to **delete[ ]**
  - Example: To delete the dynamic array created on the previous slide:

```
for ( i = 0; i < 4; i++)
     delete [ ] m[i]; //delete the arrays of 4 int's
 delete [ ] m; // delete the array of IntArrayPtr's
```

To Dos

- Homework #13 for Thursday
- Lab #8 for Monday (8AM)
  - New Lab #9 will be posted by end of the weekend

</LECTURE>