

Vectors and Pointers

CS 16: Solving Problems with Computers I
Lecture #13

Ziad Matni
Dept. of Computer Science, UCSB

Announcements

- Midterm grades will be available on Tuesday, 11/21
 - If you *need* to know it before, talk to me after class
- Your Lab #7 is due on Monday, 11/20
- You have lab on Monday, 11/20 (Lab #8 given)
- We have class on Tuesday, 11/21
- We DO NOT have class on Thursday, 11/23 (Happy Thanksgiving!)
- New Lab #8 is due Monday, 11/27 (after the break)
- New Homework #8 is due Tuesday, 11/28 (after the break)

Quick Note About Strings and Integers

If **str** is a string type var and is “47”
and **num** is an int type var and is 47

- `stoi(str) = 47` *stoi = string-to-int, found in <string>*
- `to_string(int) = “47”` *converts ints and doubles, found in <string>*

Other cool conversion functions:

- `stod(str) = 47.0` *string-to-double*

Quick Note About `getline`

- You can customize what character a `getline` stops “getting” info
 - You can define the “character delimiter”
 - By default, that’s a newline char

Example:

```
getline(cin, VariableX, 'm')    //stops at the char 'm'
```

If the standard input is “Hello, I must be going”,
then **VariableX** will be “Hello, I ”

Vectors

- Vectors are a C++ data structure
- They are like **arrays that can change size** as your program runs
- You have **less to worry** about with vectors re: size changes and mem allocations
- But vectors **consume more memory** in exchange for being dynamic and flexible

Vectors

- Vectors, like arrays, have a base type
- To declare an empty vector with base type **int**:

vector<int> v;

- **<int>** identifies vector as a *template class*
- You can use any base type in a template class:

vector<double> v;

vector<string> v;

...etc...

Accessing **vector** Elements

- Vectors elements are indexed starting with 0
 - []'s are used to read or change the value of an item:

```
v[i] = 42;  
cout << v[i];
```

- But []'s **cannot** be used to initialize a vector element

Initializing **vector** Elements

- Elements are added to a vector using the vector *member function* **push_back()**
- **push_back** adds an element in the next available position
- Example:

```
vector<double> sample;  
sample.push_back(0.0);  
sample.push_back(1.1);  
sample.push_back(2.2);
```


The size of a vector

- The member function **size()** returns the number of elements in a vector
(don't you wish you had that with arrays!?!)

Example: To print each element of a vector:

```
vector<double> sample;  
sample.push_back(0.0);  
sample.push_back(1.1);  
sample.push_back(2.2);  
  
for (int i= 0; i < sample.size( ); i++)  
    cout << sample[i] << endl;
```

The Type **unsigned int**

- The vector class member function `size` returns an **unsigned int** type of value
 - Unsigned int's are non-negative integers
- Some compilers will give a warning if the previous for-loop is not changed to:

```
for (unsigned int i= 0; i < sample.size( ); i++)  
    cout << sample[i] << endl;
```

However, g++ with **-std=c++11** seems ok with plain old “int”...

Alternate **vector** Initialization

- A vector constructor exists that takes an integer argument and initializes that number of elements
 - A **constructor** is a part of a class that is usually used for initialization purposes

- Example:

```
vector<int> v(10);  
    initializes the first 10 elements to 0  
v.size( )  
    would then return 10
```

- []'s can now be used to assign elements 0 through 9
- **push_back** is used to assign elements *greater than* 9

The **vector** Library

- To use the vector class
 - You have to include the vector library

#include <vector>

- Vector names are placed in the standard namespace so the usual using directive is needed:

using namespace std;

```

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

int main( )
{
    vector<int> v;
    cout << "Enter a list of positive numbers.\n"
         << "Place a negative number at the end.\n";

    int next;
    cin >> next;
    while (next > 0)
    {
        v.push_back(next);
        cout << next << " added. ";
        cout << "v.size( ) = " << v.size( ) << endl;
        cin >> next;
    }

    cout << "You entered:\n";
    for (unsigned int i = 0; i < v.size( ); i++)
        cout << v[i] << " ";
    cout << endl;

    return 0;
}

```

Sample Dialogue

```

Enter a list of positive numbers.
Place a negative number at the end.
2 4 6 8 -1
2 added. v.size( ) = 1
4 added. v.size( ) = 2
6 added. v.size( ) = 3
8 added. v.size( ) = 4
You entered:
2 4 6 8

```

See textbook, pg. 493

Defining **vector** Elements Beyond Vector Size

- Attempting to use [] to set a value beyond the size of a vector *may not generate an error, but it is not correct to do!*
- Example: assume integer vector **v** has 3 elements in it
 - Performing an assignment like `v[5] = 4` isn't the "correct" thing to do
 - INSTEAD you should use **push_back()** enough times to get to element 5 first before making changes
- Even though you may not get an error, you have messed around with memory allocations and the program will probably misbehave in other ways

vector Efficiency

- A vector's **capacity** is the number of “spaces” in memory that are put aside for vector elements
- **size()** is the number of elements *initialized*
- **capacity()** is the number of elements that are *put aside* (*automatically reserved*)
- When a vector runs out of space, *the capacity is automatically increased!*
- A common scheme by the compiler is to *double* the size of a vector

Controlling **vector** Capacity

- When efficiency is an issue and you want to control memory use (i.e. and not rely on the compiler)...
- Use member function **reserve()** to increase the capacity of a vector

Example:

```
v.reserve(32);           // at least 32 elements
v.reserve(v.size( ) + 10); // at least 10 more
```

- **resize()** can be used to shrink a vector

Example:

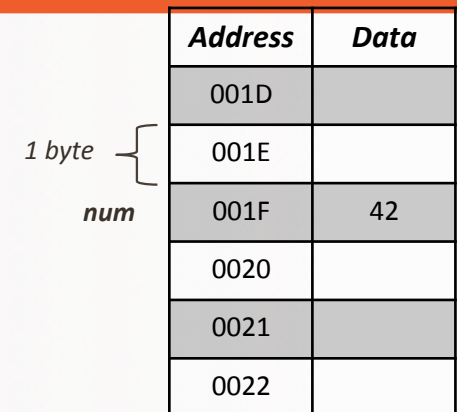
```
v.resize(24);           //elements beyond 24 are lost
```

INTRO TO POINTERS

Section 9.1 in book

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 0x001F
 - Generally, memory addresses use *hexadecimals (just not only 4 of them...)*
 - The “0x” at the start is just to indicate the number is a hex



Address	Data
001D	
001E	
001F	42
0020	
0021	
0022	

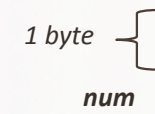
Memory Addresses

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

- Example:

```
int num_add = &num;
```

will result in **num_add** to hold the value **001F** (but expressed in decimal)



Address	Data
001D	
001E	
001F	42
0020	
0021	
0022	

Memory Address

Recall: `num = 42` and `num_add = &num = 0x001F`

- Now, let's make **`bar = num`**
 - Another variable, **`bar`**, now is assigned the same value that's in `num` (42)
- The variable **`bar`** will be assigned an address
 - Let's say, that address is **`0x3A77`**
 - Keep in mind, by default, we have no control over address assignments
- The variable that stores the address of another variable (like **`num_add`**) is called a ***pointer***.

Dereference Operator (*)

- Pointers “point to” the variable whose address they store
- Pointers can **access** the variable they point to directly
- This access is 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 = 0x001F`

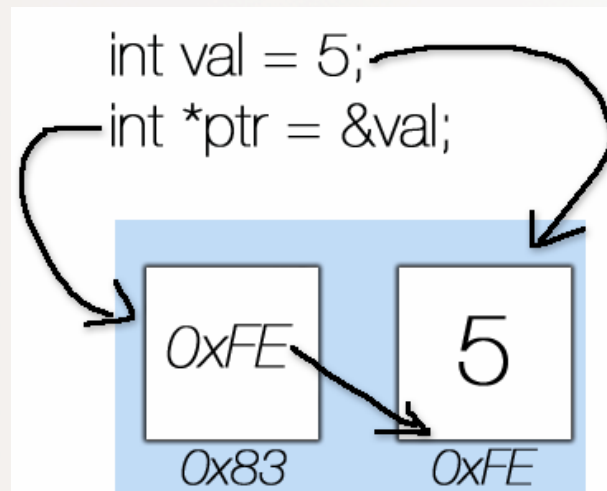
- So, while `num_add = 0x001F,`
`*num_add = 42`

Pointers

- A pointer is the **memory address** of a 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*

- 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 “the variable pointed to by p1”

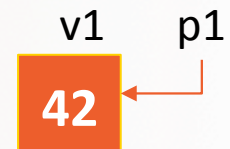
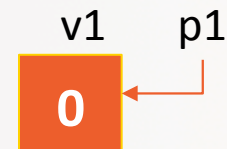
A Pointer Example

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

v1 and *p1 now refer to the same variable

output:

42
42



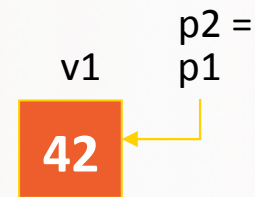
Pointer Assignment

- 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

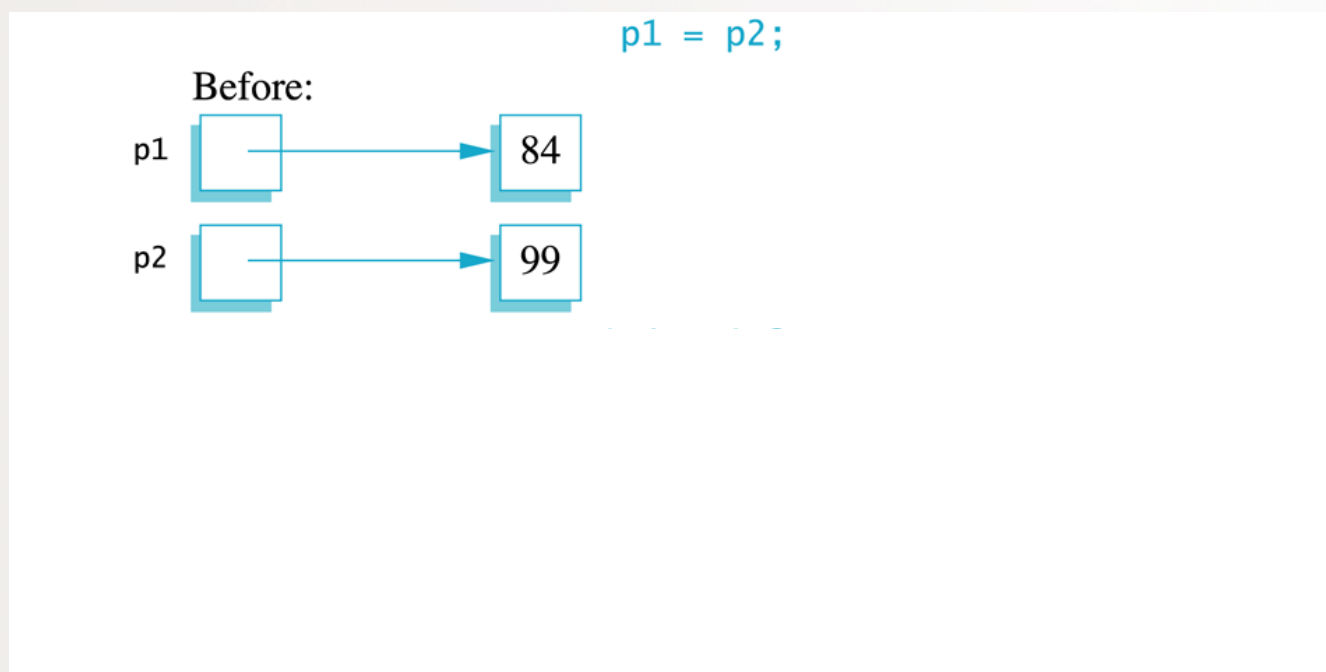
- Some care is required making assignments to pointer variables

Assuming p1 and p3 are pointers

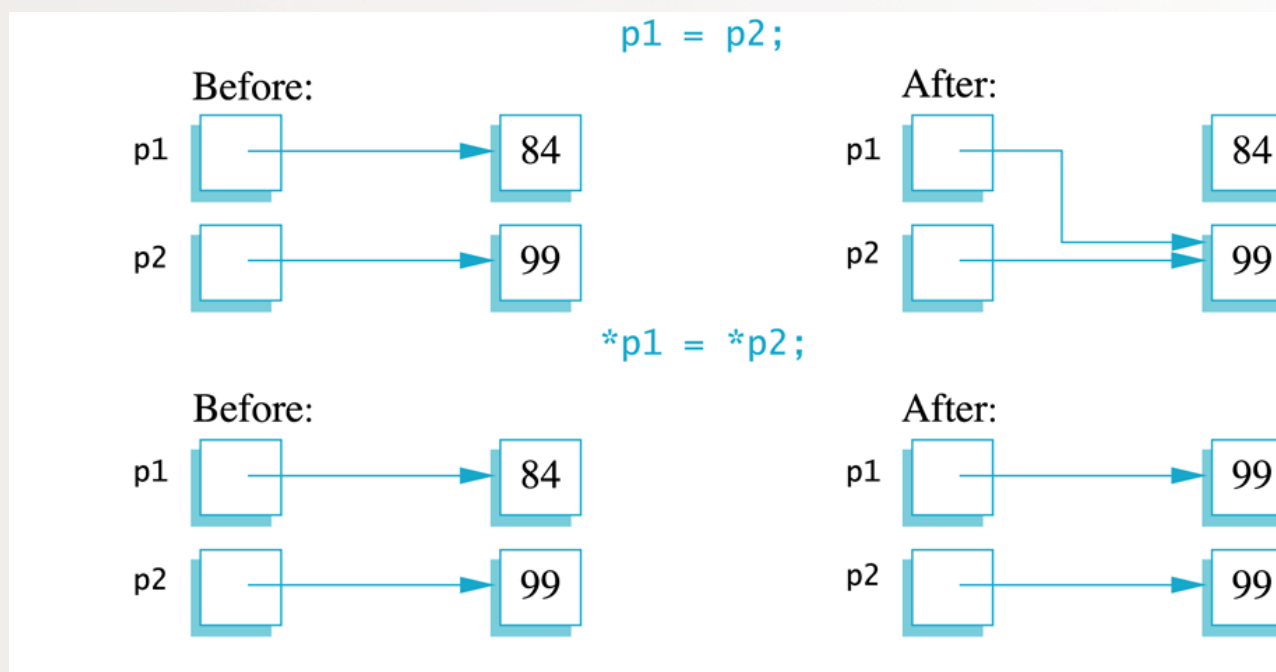
```
p1 = p3;    // changes the location that p1 "points" to
```

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

Uses of the Assignment Operator on Pointers



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

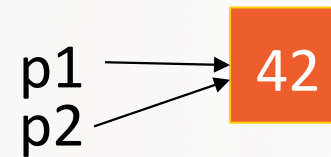
Example: `cin >> *p1;`
`*p1 = *p1 + 7;`

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

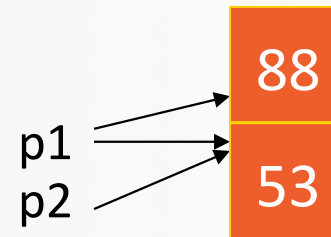
Basic Pointer Manipulations

```
//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;  
}
```



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;  
}
```



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;

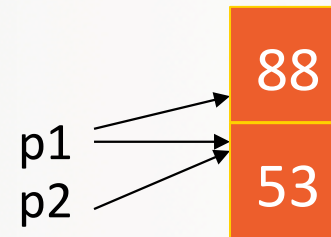
    *p2 = 53;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    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: The Heap

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

The **delete** Operator

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

Dangling Pointers

- Using **delete** on a pointer variable *destroys* the dynamic variable pointed to (frees up memory too!)
- 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

- As you know: 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*** control 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;
Helps prevent errors 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;  
int 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, like...

- Example:

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

is less confusing than:

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


YOUR TO-DOs

- ☐ HW 8 due **Tue.** 11/28
- ☐ Lab 8 due **Mon.** 11/27
- ☐ Visit Prof's and TAs' office hours if you need help!
- ☐ Rinse, Wash, Repeat

</LECTURE>