Pointers

CS 16: Solving Problems with Computers I
Lecture #12

Ziad Matni
Dept. of Computer Science, UCSB

MIDTERM IS COMING!

- **Thursday, 5/17** in this classroom
- **Starts at 2:00 PM **SHARP****
  - Please start arriving 5-10 minutes before class
- **I may ask you to change seats**
- Please bring your UCSB IDs with you

- **Closed book**: no calculators, no phones, no computers
- ***Only*** allowed ONE 8.5”x11” sheet of notes − one sided only
  - You have to turn it in with your exam
- **You will write your answers on the exam sheet itself.**

What's on the Midterm#2?
EVERYTHING *From Lectures 7 – 12, including…*

- Makefiles
- Debug Techniques
- Numerical Conversions
- Strings: C++ vs C-strings
- Strings and Characters: Member Functions & Manipulators
- File I/O
- Arrays
- Pointers (whatever we finish today)

Lecture Outline

- Pointers

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

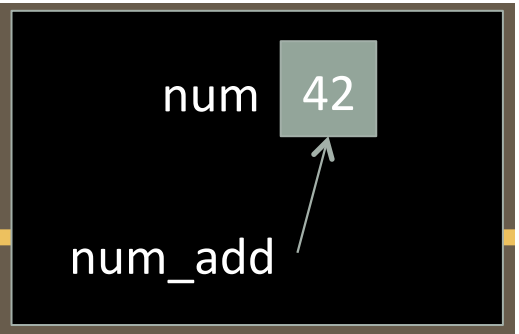
Memory Addresses

- The address of a variable can be obtained by putting the ampersand character (&) before the variable name.
- **&** is called the ***address-of*** operator
- So, while **num** = 42, **&num** = 0x1F = 31
- You can assign a variable to an address-of *another* variable too!
- Example: **num_add = &num**

1 byte

Address	*Data*	
0x001D	0	
0x001E	-25	
0x001F	42	**num**
0x0020	1332	
0x0021	-4009	
0x0022	7	
…	…	
0x98A0	31	**num_add**

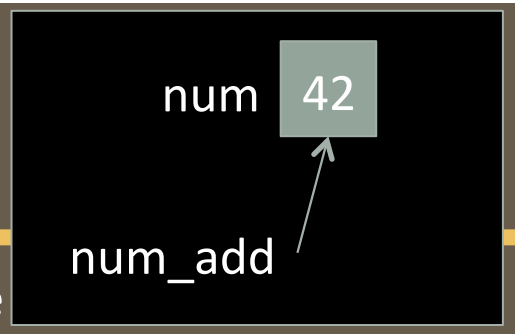
Memory Address



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

- 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"

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

  ***num_add = 42**

Pointers

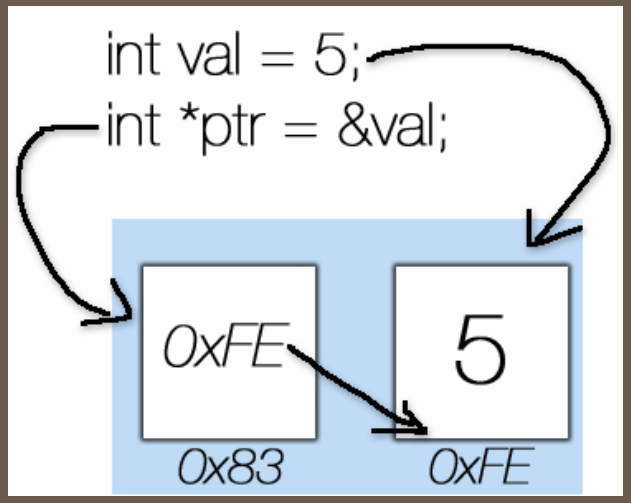
AGAIN:

- 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

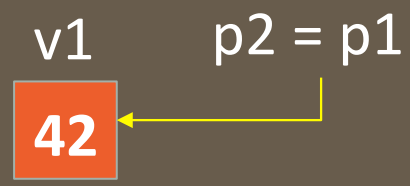
output:

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

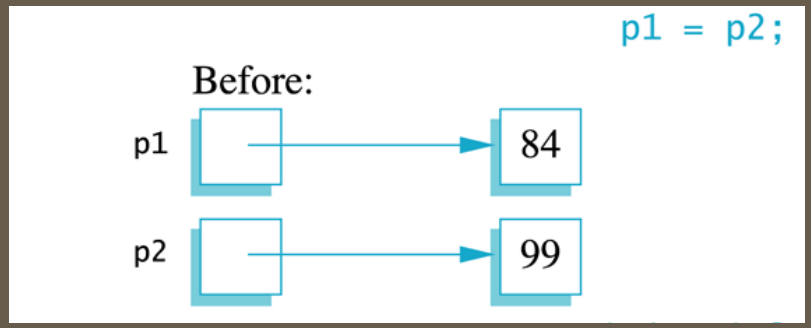
Caution! Pointer Assignments

- Some care is required making assignments to pointer variables

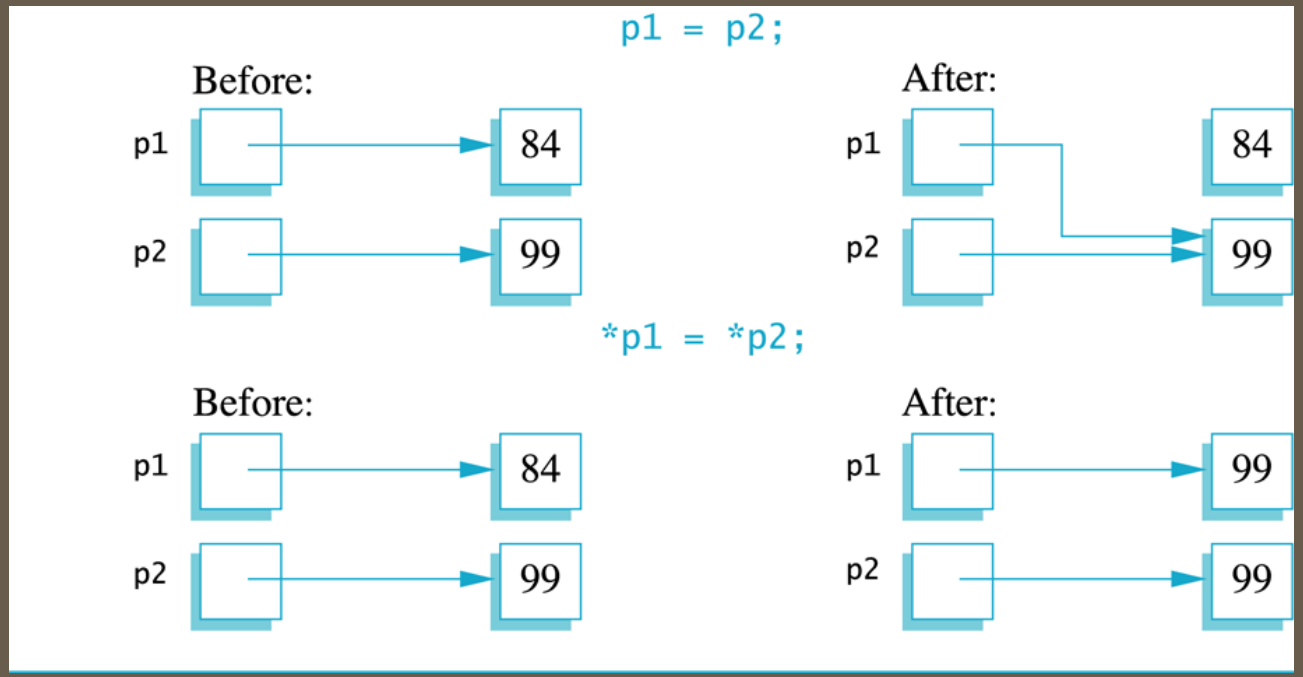
Assuming p1 and p3 are pointers

```
 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



Pointer Assignment – Example 1

Consider this code:

```
int *p1, *p2, x;
p1 = &x;
p2 = p1;
```

Figure A

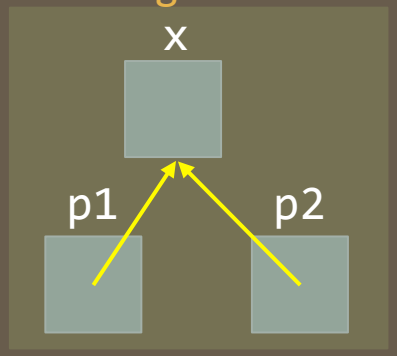
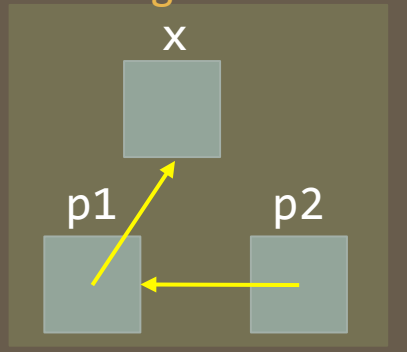


Figure B

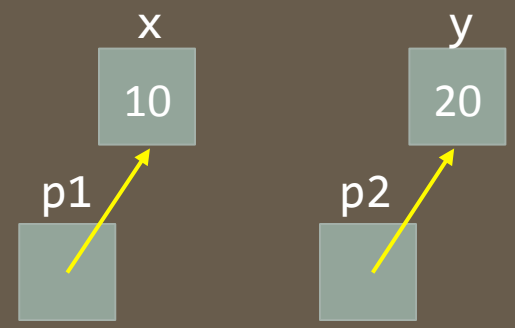


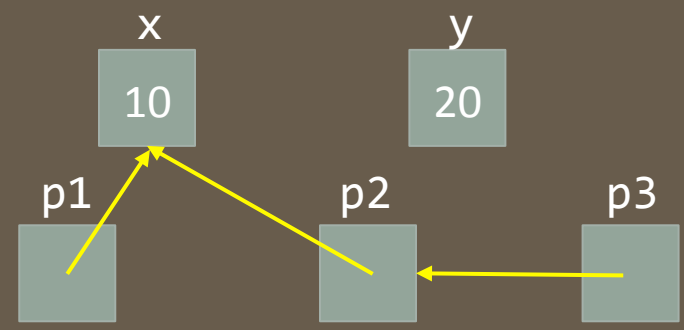
Which figure best represents this code?

A. Figure A
B. Figure B
C. Neither: the code is incorrect

Pointer Assignment – Example 2

```
int x=10, y=20;
int *p1 = &x, *p2 = &y;
p2 = p1;
int **p3;
p3 = &p2;
```





Q: How can I print out the value "10" using p2?

A: cout << *p2;

Q: How can I print out the value "10" using p3?

A: cout << **p3;

Passing by Pointers! A Better Way!

```
void swapValue(int *x, int *y)
{
   int tmp = *x;
   *x = *y;
   *y = tmp;
}
int main()
{
   int a(30), b(40);
   cout << a << " " << b << endl;
   swapValue(&a, &b);
   cout << a << " " << b << endl;
}
```

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

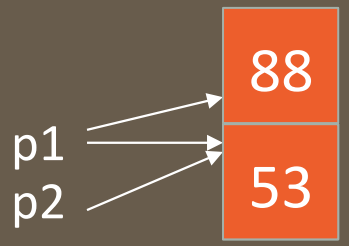
  dynamic variables

- *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;
```

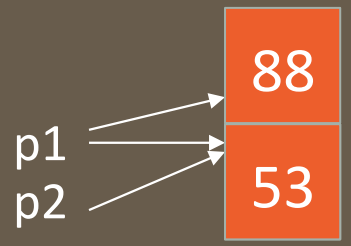


Basic Pointer Manipulations

```
//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***

- 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 data 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;
```

Pointer Reference Parameters

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

- Example:

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

is less confusing than:

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

YOUR TO-DOs

- ❑ Start Lab 7 on Wednesday
- ❑ Do HW12 by next **Tuesday**
- ❑ Study for your Midterm #2 on Thursday!

- ❑ Visit Prof's and TAs' office hours if you need help!

- ❑ Sleep more than you study. Study more than you party. And don't forget to party…

</LECTURE>