Pointers
Introduction to Structures

CS 16: Solving Problems with Computers I
Lecture #15

Ziad Matni
Dept. of Computer Science, UCSB

Administrative

- 4 MORE CLASSES TO GO! ☺

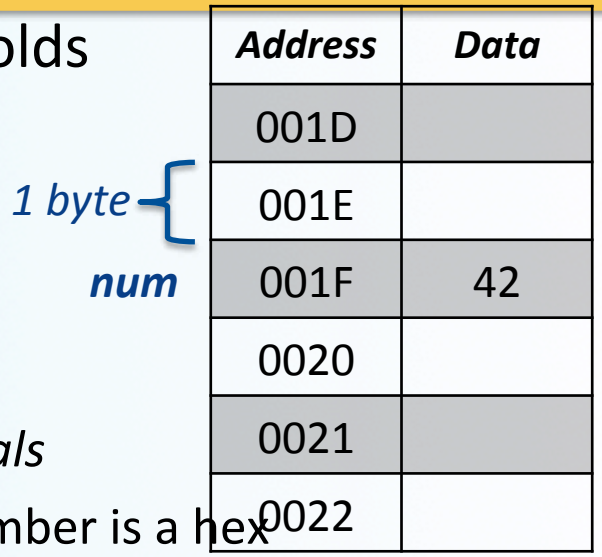
**** Due date changed!***

M	T	W	Th	F
5/29	**5/30** **LECTURE 15** HW13 due	**5/31** Lab8 issued	**6/1** **LECTURE 16** HW14 due Lab7 due*	**6/2**
6/5	**6/6** **LECTURE 17** HW15 due	**6/7**	**6/8** **REVIEW** HW16 due Lab8 due	**6/9** *Last day of the quarter*

Pointers

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*
  - The "0x" at the start is just to indicate the number is a hex



- 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 = &num;` will result in **num_add** to hold the value 001F (but expressed in decimal)

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)
  - Note the difference between **bar** and **num_add**

- 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
    - And 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 = 0x001F

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

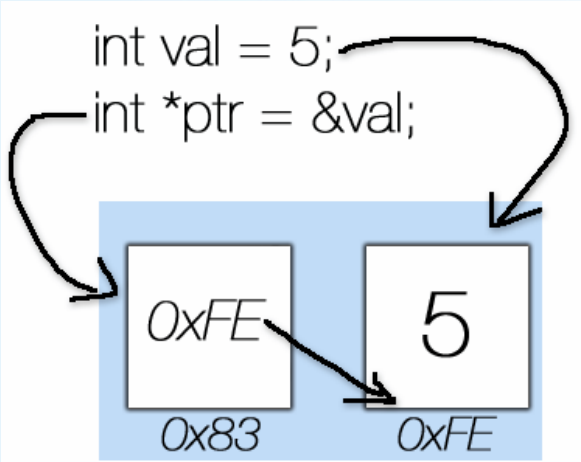
Pointers

- **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

- 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"

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*

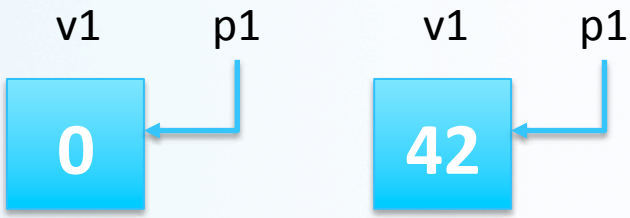
A Pointer Example

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

v1 and *p1 now refer to the same variable

output:

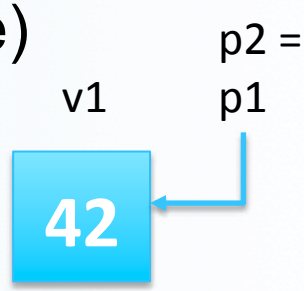
42
42



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

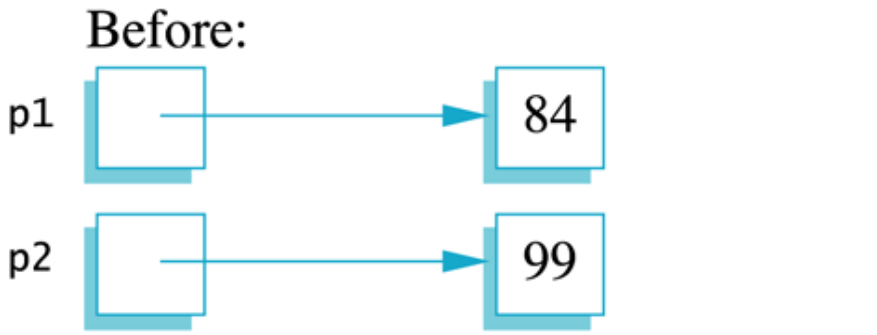
Caution! Pointer Assignments

- Some care is required making assignments to pointer variables

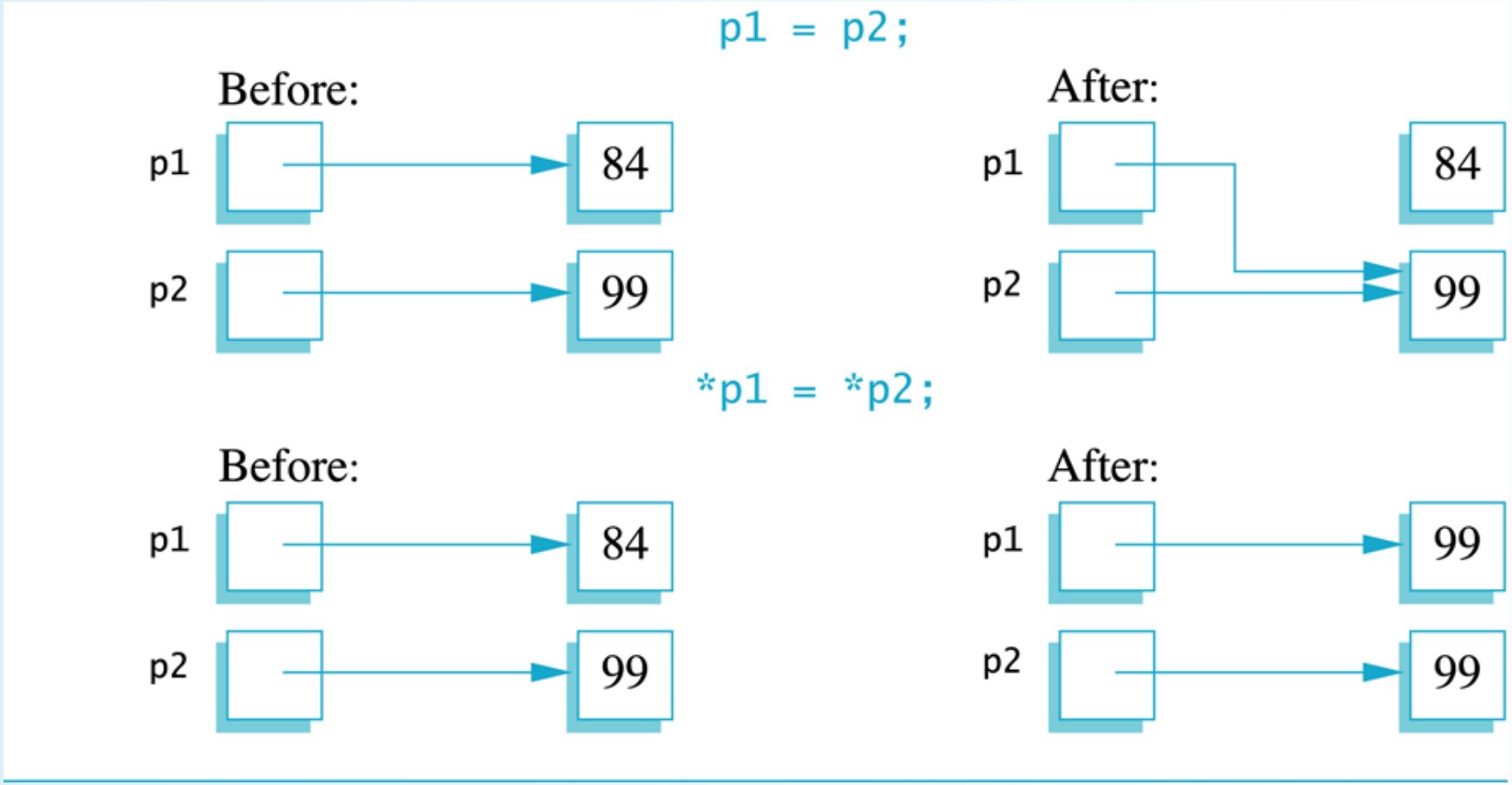
```
p1 = p3;     // changes the location that p1 "points" to

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

Uses of the Assignment Operator on Pointers

```
p1 = p2;
```



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;
```

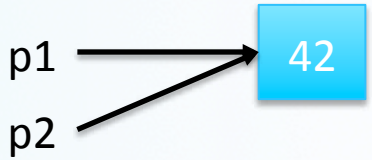
Dynamic Variables

- 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

- 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

- 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;
```

- 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);
```

Structures

What Is a Class?

- A ***class*** is a data type whose variables are ***objects***
- Some pre-defined data types you have used are:
  - int
  - char
- Some pre-defined classes you have used are:
  - **ifstream**
  - **string**
- You can define your own classes as well

Class Definitions

- To define a “class”, we need to…
  - Describe the kinds of values the variable can hold
    - Numbers? Characters? Both? Others?
  - Describe the member functions
    - What can we do with these values?

- We will start by defining ***structures*** as a first step toward defining classes

Structures

- A structure can be viewed as an **object**
- Let's say it does not contain any member functions (for now…)
- It does contain multiple values of possibly different types
- We'll call these **member variables**

Structures

- These multiple values are logically related to one another and come together as a single item
  - Examples:
    A bank Certificate of Deposit (CD) which has the following values:

    a balance
    an interest rate
    a term (how many months to maturity)

    What kind of values should these be?!

  - A student record which has the following values:

    the student's ID number
    the student's last name
    the student's first name
    the student's GPA

    What kind of values should these be?!

The CD Structure Example: Definition

- The Certificate of Deposit structure can be defined as

```
struct CDAccount
{
    double balance;
    double interest_rate;
    int term;
} ;
```

Remember this semicolon!

- Keyword **struct** begins a structure definition
- **CDAccount** is the structure ***tag*** – this is the structure's **type**
- Member names are ***identifiers*** declared in the braces

Using the Structure

- Structure **definition** should be placed outside any function definition
  - This makes the structure type available to all code that follows the structure definition

- To declare two variables of type **CDAccount**:

```
CDAccount  my_account, your_account;
```

  - `my_account` and `your_account` contain distinct member variables `balance`, `interest_rate`, and `term`

The Structure Value

- **Structure Value** consists of all the values of the member variables

- The value of an object of type **CDAccount** consists of the values of the member variables

```
balance
interest_rate
term
```

Specifying Member Variables

- Member variables are specific to the structure variable in which they are declared
- Syntax to specify a member variable (note the '.')
  Structure_Variable_Name . Member_Variable_Name
  - Given the declaration:

```
CDAccount  my_account, your_account;
```

  - Use the **dot operator** to specify a member variable

```
my_account.balance
my_account.interest_rate
my_account.term
```

```
//Program to demonstrate the CDAccount structure type.
#include <iostream>
using namespace std;

//Structure for a bank certificate of deposit:
struct CDAccount
{
    double balance;
    double interest_rate;
    int term;//months until maturity
};

void get_data(CDAccount& the_account);
//Postcondition: the_account.balar
//have been given values that the
```

Note the struct definition is placed before main()

```
int main( )
{
    CDAccount account;
    get_data(account);

    double rate_fraction, interest;
    rate_fraction = account.interest_rate/100.0;
    interest = account.balance*rate_fraction*(account.term/12.0);
    account.balance = account.balance + interest;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "When your CD matures in "
         << account.term << " months,\n"
         << "it will have a balance of $"
         << account.balance << endl;
    return 0;
}
```

Note the declaration of CDAccount

Note the calculations done with the structure's member variables

Note the use of the structure's member variables with an input stream

```
//Uses iostream:
void get_data(CDAccount& the_account)
{
    cout << "Enter account balance: $";
    cin >> the_account.balance;
    cout << "Enter account interest rate: ";
    cin >> the_account.interest_rate;
    cout << "Enter the number of months until maturity\n"
         << "(must be 12 or fewer months): ";
    cin >> the_account.term;
}
```

Sample Dialogue

```
Enter account balance: $100.00
Enter account interest rate: 10.0
Enter the number of months until maturity
(must be 12 or fewer months): 6
When your CD matures in 6 months,
it will have a balance of $105.00
```

</LECTURE>