Object Oriented Programming

Lecture 7

Arrays of Class Objects

- Class objects can also be used as array elements

```
class Square
{ private:
    int side;
  public:
    Square(int s = 1)
    { side = s; }
    int getSide()
    { return side; }
};
Square shapes[10];  // Create array of 10
                    // Square objects
```

Arrays of Class Objects

- Like an array of structures, use an array subscript to access a specific object in the array
- Then use dot operator to access member methods of that object

```
for (i = 0; i < 10; i++)
  cout << shapes[i].getSide() << endl;
```

Initializing Arrays of Objects

- Can use default constructor to perform same initialization for all objects
- Can use initialization list to supply specific initial values for each object

```
Square shapes[5] = {1,2,3,4,5};
```

- Default constructor is used for the remaining objects if initialization list is too short

```
Square boxes[5] = {1,2,3};
```

Initializing Arrays of Objects

If an object is initialized with a constructor that takes > 1 argument, the initialization list must include a call to the constructor for that object

```
Rectangle spaces[3] =
{ Rectangle(2,5),
  Rectangle(1,3),
  Rectangle(7,7)   };
```

Arrays of Structures

- Structures can be used as array elements

```
struct Student
{
   int studentID;
   string name;
  short year;
   double gpa;
};
const int CSIZE = 30;
Student class[CSIZE]; // Holds 30
                      // Student structures
```

Arrays of Structures

- Use array subscript to access a specific structure in the array
- Then use dot operator to access members of that structure

```
cin  >> class[25].studentID;

cout << class[i].name << " has GPA "
     << class[i].gpa << endl;
```

Destructors

- A destructor is a member function that is automatically called when an object is destroyed.
    - Destructors have the same name as the class, preceded by a tilde character (~)
    - In the same way that a constructor is called then the object is created, the destructor is automatically called when the object is destroyed.
    - In the same way that a constructor sets things up when an object is created, a destructor performs shutdown procedures when an object is destroyed.

Program

```
// This program demonstrates a destructor.

#include <iostream.h>

class Demo
{
public:
    Demo(void);     // Constructor
    ~Demo(void);   // Destructor
};

Demo::Demo(void)
{
    cout << "Welcome to the constructor!\n";
}
```

Program continues

```
Demo::~Demo(void)
{
   cout << "The destructor is now running.\n";
}

void main(void)
{
   Demo demoObj;        // Declare a Demo object;
   cout << "This program demonstrates an object\n";
   cout << "with a constructor and destructor.\n";
}
```

Program Output

```
Welcome to the constructor!
This program demonstrates an object
with a constructor and destructor.
The destructor is now running.
```

Program

```
#include <iostream.h>
#include <string.h>
class InvItem
{
   private:
            char *desc;
            int units;
   public:
            InvItem(void) { desc = new char[51]; }
            ~InvItem(void) { delete desc; }
            void setInfo(char *dscr, int un) { strcpy(desc, dscr);
                                                units = un;}
            char *getDesc(void) { return desc; }
            int getUnits(void) { return units; }
};
```

Program continues

```
void main(void)
{
    InvItem stock;
    stock.setInfo("Wrench", 20);
    cout << "Item Description: " << stock.getDesc() << endl;
    cout << "Units on hand: " << stock.getUnits() << endl;
}
```

Program Output

```
Item Description: Wrench
Units on hand: 20
```

Automatic Functions

- Functions automatically created for each class:
  - Constructor: Creates objects
  - Destructor: Deletes objects
  - Copy Constructor
  - Assignment operator =

We have covered Constructor and Destructor.

Copy Constructor

- Copy constructor is a “constructor”
- It is a function with the same name as the class and no return type.
- However, it is invoked implicitly
  - An object is defined to have the value of another object of the same type.
  - An object is passed by value into a function
  - an object is returned by value from a function

Copy Constructor

- Examples

```
Fraction f1, f2(3,4)
Fraction f3 = f2;  // Copy Const.

f1 = f2;  // Not a copy but assignment
          // operator.
```

Copy Constructor

- Declaring and Defining
  - A copy constructor always has one (1) parameter, the original object.
  - Must be the same type as the object being copied to.
  - Always passed by reference (must be because to pass by value would invoke the copy constructor).
  - Copy constructor not required

```
Fraction (const Fraction &f);
Timer (const timer & t);
```

Copy Constructor

- Shallow copy vs deep copy
  - The default version is a shallow copy. I.E. the object is copies exactly as is over to the corresponding member data in the new object location.
  - Example:
    - Fraction f1(3,4);
    - The object example illustrates the definition of an object f1 of type Fraction.
    - If passed as a parameter, a shallow copy will be sufficient.

Copy Constructor

- When there is a pointer to dynamic data, a shallow copy is not sufficient.
- Why? Because a default or shallow copy will only copy the pointer value (Address). Essentially both objects are pointing to the same item. Here we need a deep copy.

Copy constructor

- Deep Copy

```
Directory::Directory (const Directory & d)
{
 maxsize = d.maxsize;
 currentsize = d.currentsize;
 entryList = new Entry[d.maxsize];
 for (int i=0; i<currentsize; i++)
   entryList[i] = d.entryList[i];
}
```

Copy Constructors

- **Shallow Copy:**
  - The data members of one object are copied into the data members of another object without taking any dynamic memory pointed to by those data members into consideration. (“memberwise copy”)

- **Deep Copy:**
  - Any dynamic memory pointed to by the data members is duplicated and the contents of that memory is copied (via copy constructors and assignment operators -- when overloaded)

Copy Constructors

- In every class, the compiler automatically supplies both a copy constructor and an assignment operator if we don't explicitly provide them.
- Both of these member functions perform copy operations by performing a memberwise copy from one object to another.
- In situations where pointers are not members of a class, memberwise copy is an adequate operation for copying objects.
- However, it is not adequate when data members point to memory dynamically allocated within the class.

Copy Constructors

- Problems occur with shallow copying when we:
  - initialize an object with the value of another object:

    ```
    name s1;   name s2(s1);
    ```

  - pass an object by value to a function or when we return by value:

    ```
    name  function_proto (name)
    ```

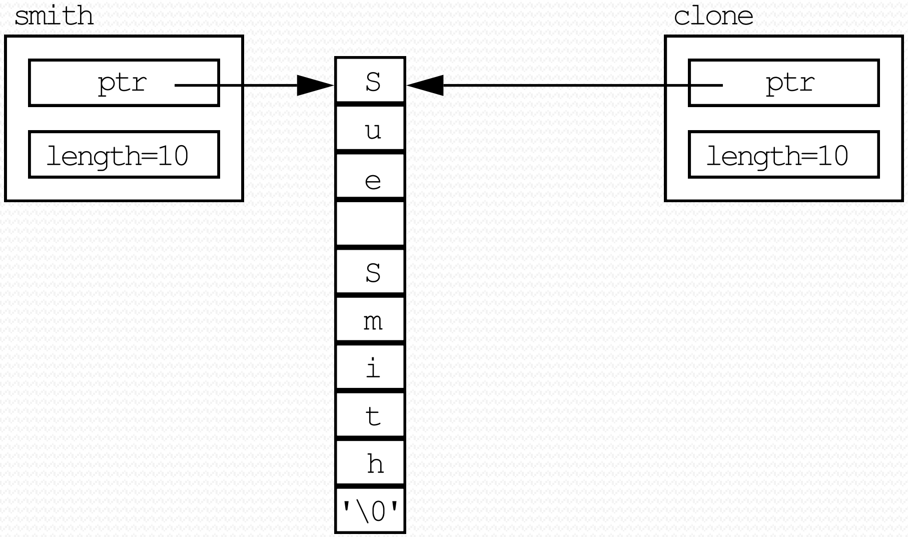
  - assign one object to another:

    ```
    s1 = s2;
    ```

Copy Constructors

- If name had a dynamically allocated array of characters (i.e., one of the data members is a pointer to a char),
  - the following shallow copy is disastrous!

```
name smith("Sue Smith"); //one arg constructor used
name clone(smith);       //default copy constructor used
```



Copy Constructors

- To resolve the pass by value and the initialization issues, we must write a copy constructor whenever dynamic member is allocated on an object-by-object basis.
- They have the form:

```
class_name(const class_name &class_object);
```

- Notice the name of the "function" is the same name as the class, and has no return type
- The argument's data type is that of the class, passed as a constant reference (think about what would happen if this was passed by value?!)

Copy Constructors

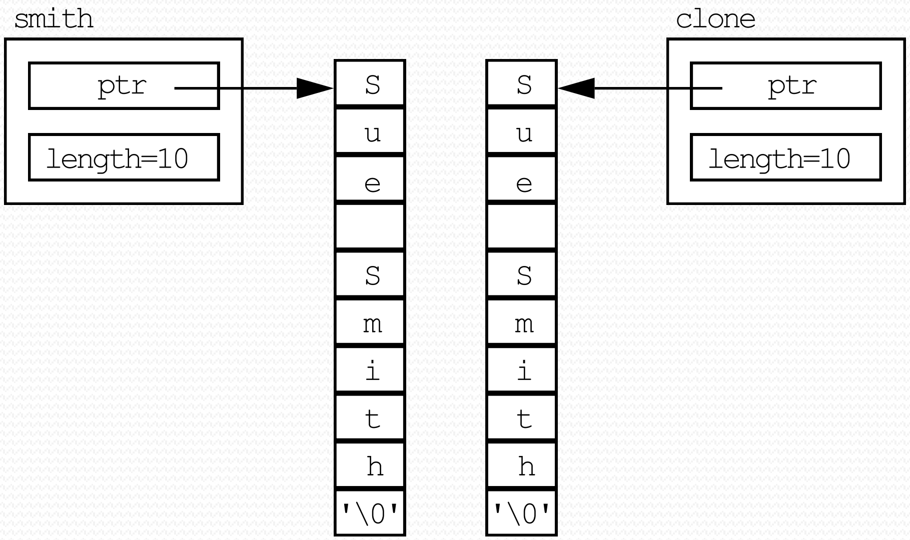
```
//name.h interface
class name {
  public:
    name(char* = "");        //default constructor
    name(const name &);      //copy constructor
    ~name();                 //destructor
    name &operator=(name &); //assignment op
  private:
    char* ptr;  //pointer to name
    int length; //length of name including nul char
};

#include "name.h"                              //name.c implementation
name::name(char* name_ptr) {   //constructor
  length = strlen(name_ptr);   //get name length
  ptr = new char[length+1];    //dynamically allocate
  strcpy(ptr, name_ptr);       //copy name into new space
}
name::name(const name &obj) {  //copy constructor
  length = obj.length;         //get length
  ptr = new char[length+1];    //dynamically allocate
  strcpy(ptr, obj.ptr);        //copy name into new space
}
```

Copy Constructors

- Now, when we use the following constructors for initialization, the two objects no longer share memory but have their own allocated

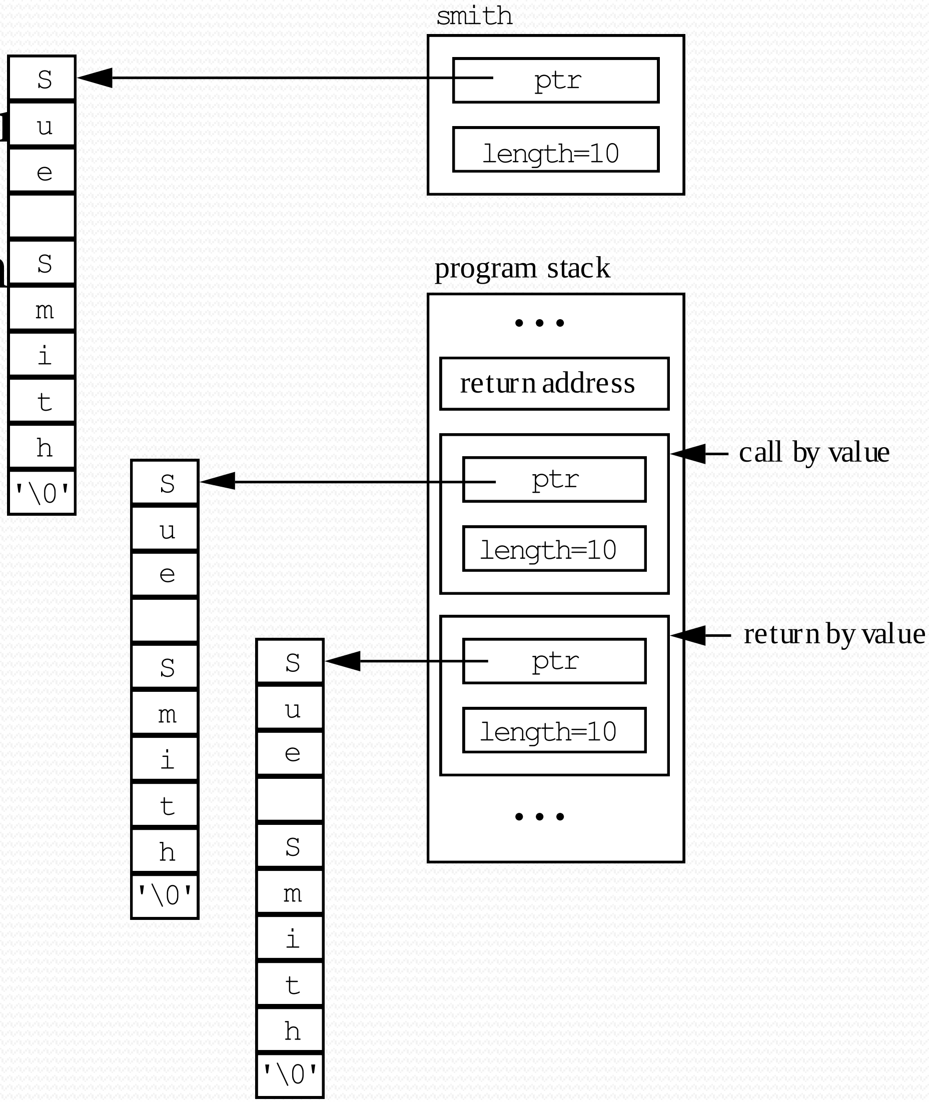
```
name smith("Sue Smith"); //one arg constructor used
name clone(smith);       //default copy constructor used
```



Copy Constructors

- Using a copy constructor avoids objects “sharing” memory -- but causes th behavior

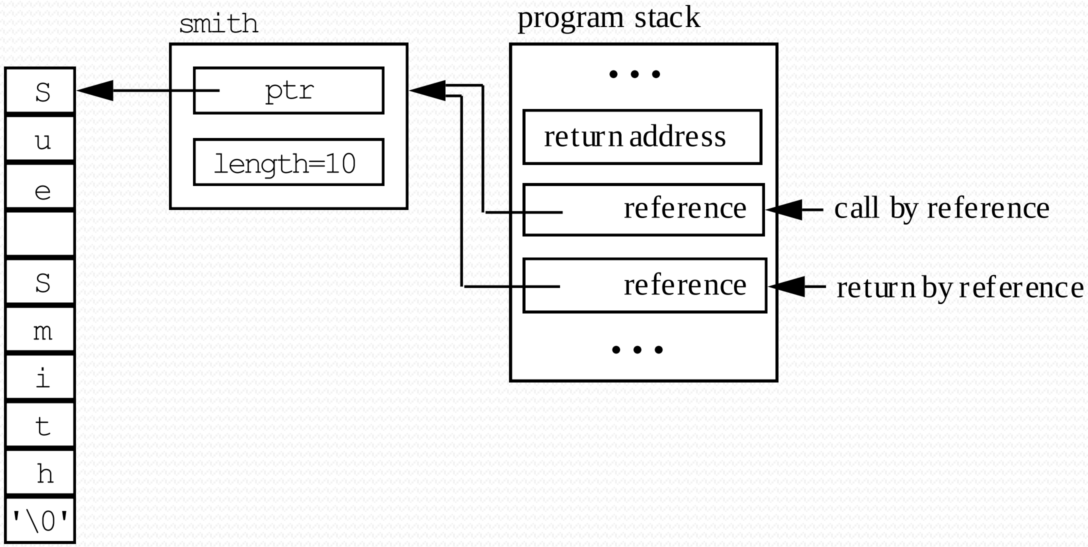
- This should convince us to avoid pass by value whenever possible -- when passing or returning objects of a class!



Copy Constructors

- Using the reference operator instead, we change the function to be: (the function call remains the same)

```
name &function(name &obj) {
  cout <<obj.get_name() <<endl;
  return (obj);
}
```



smith

ptr

length=10

S
u
e

S
m
i
t
h
'\0'

program stack

• • •

return address

reference

reference

• • •

call by reference

return by reference