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:

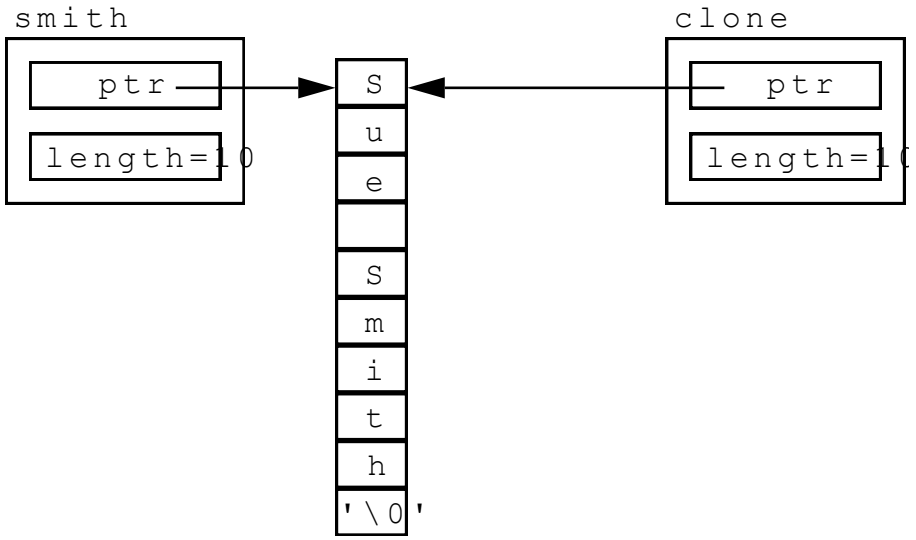
    name function_proto (name)
  - assign one object to another:

    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

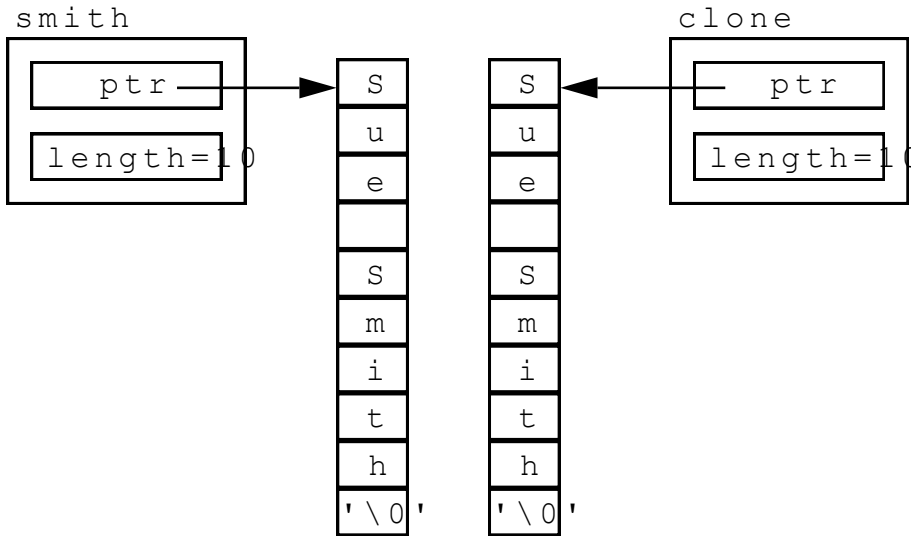
```
//name.h interface
class name {
  public:
    name(char* = "");        //default constructor
    name(const name &);      //copy constructor
    ~name();                 //destructor
    name &operator=(name &); //assignment op
  private:
    char* ptr;  //pointer to name
    int length; //length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

- Copy constructors are also used whenever passing an object of a class by value: (get_name returns a ptr to a char for the current object)

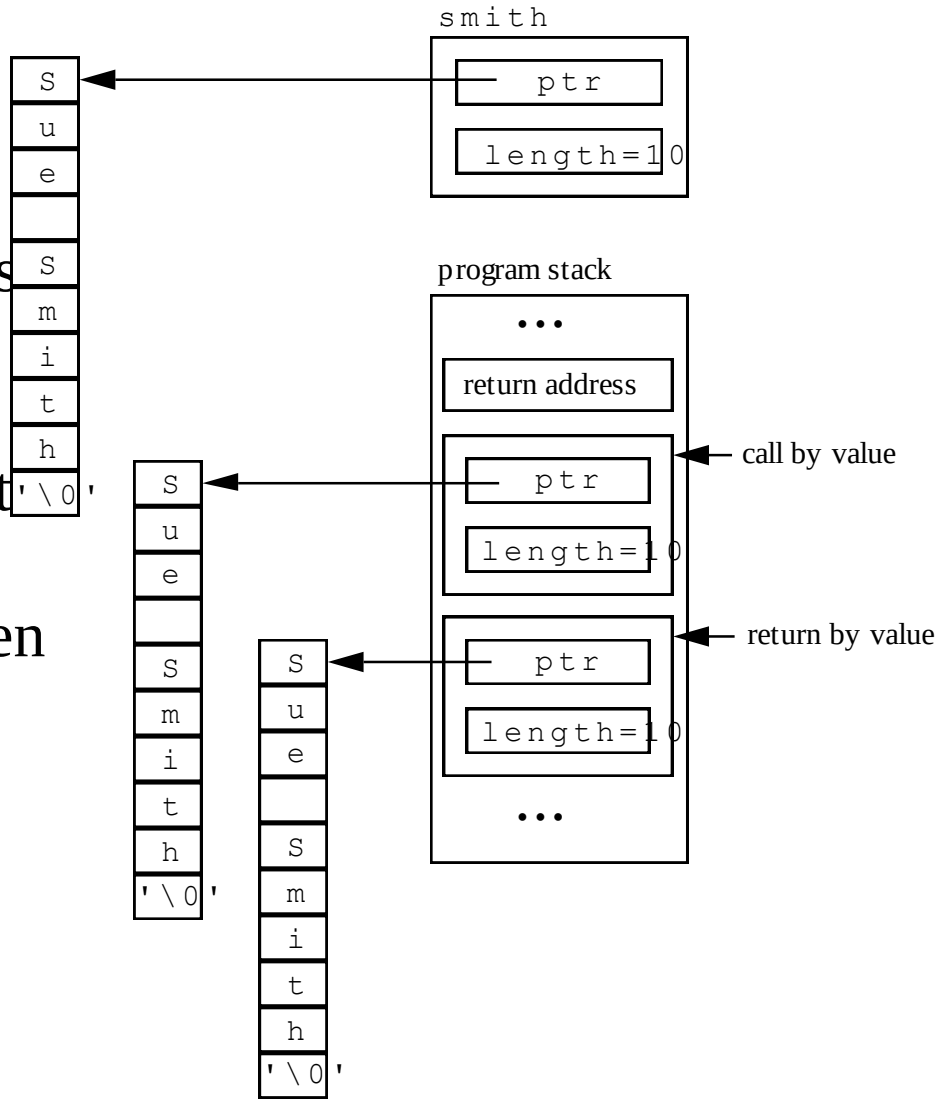
```
int main() {
  name smith("Sue Smith"); //constructor with arg used

  //call function by value & display from object returned
  cout <<function(smith).get_name() <<endl;
  return (0);
}

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

Copy Constructors

- Using a copy constructor avoids objects “sharing” memory -- but causes this 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);
}
```