Member Wise Initialization of Objects

Assigning (copying) Objects

- The contents of one object can be copied to another object using the assignment operator. For example:

```
Student student1("Mike Smith",654321);
Student student2;

student2 = student1;
```

- The last line will copy the member variables of `student1` into the same member variables of `student2`. The copying is done in a bitwise manner.

Initialization vs. assignment

- C and C++ depreciates between this two operations:

```
int a = 40;   // initialization
int b;
b = a ;       // assignment
```

- You may initialize the object of a class by an existing object of the same class:

```
Aclass obj1;
Aclass obj2 = obj1;  // This is initialization
```

- Every data member of instance obj1 will be copied into instance obj2. This is called member-wise copying
- An important point:
  - Declaration of obj2 does not invoke the constructor of class Aclass.
- Warning:
  - This method of copying objects of class should be done with care.
    - We will discus this in the next slides by an example

Member-wise Initialization

```
class MyString
{
public:
    MyString(char *s);
    MyString();
    ~MyString();
    … // more member functions
private:
    char* storageM;
    int lengthM;

};
```

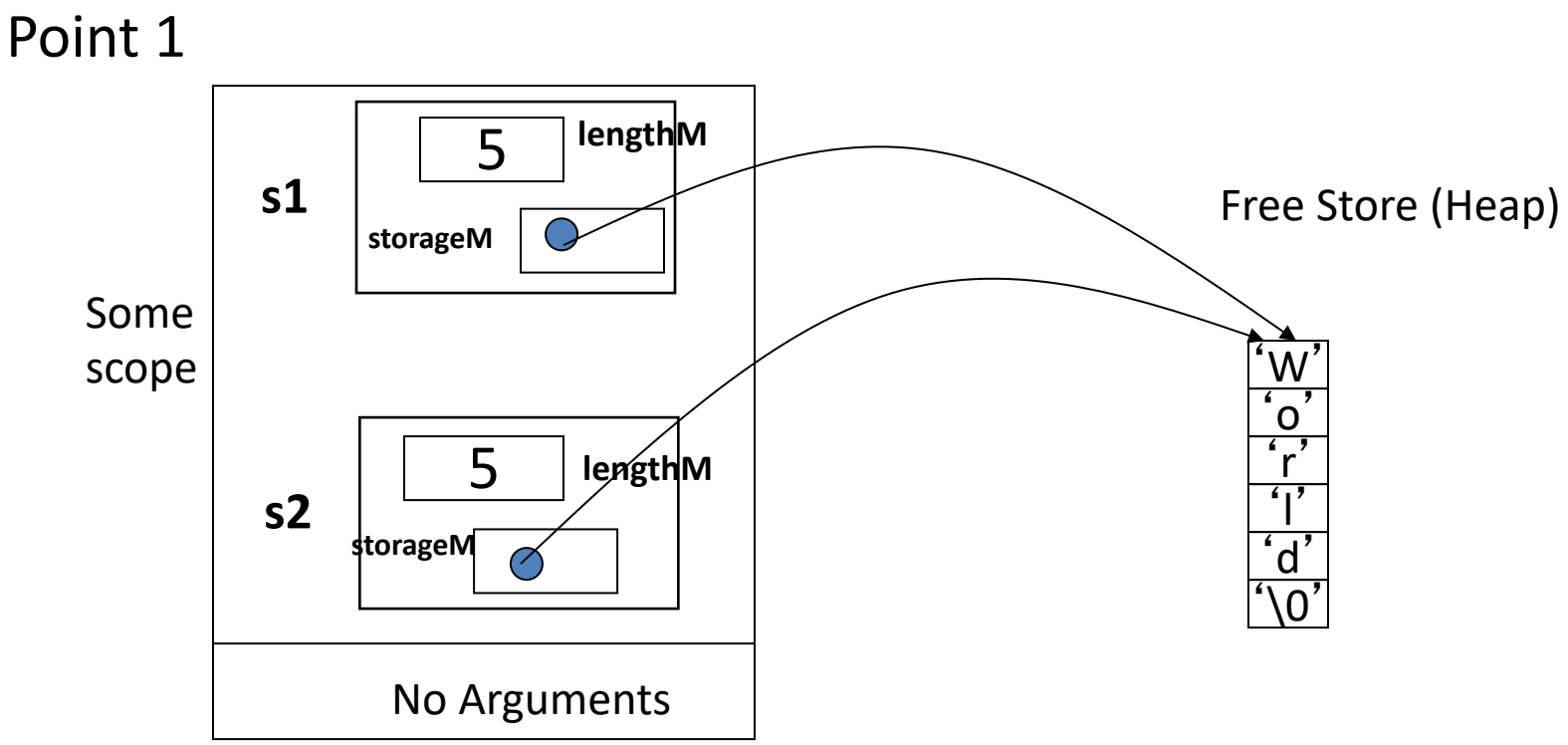
Partial implementations

```
// constructor
MyString::MyString(const char *s): lengthM((int)strlen(s)){
   storageM = new char[lengthM + 1];
   strcpy(storageM, s);
   std::cout << "\nctor called.";
}

// default constructor
MyString::MyString(): lengthM(0), storageM(new char[1]){
   storageM[0] = '\0';
   std::cout << "\ndefault ctor called.";
}

// destructor
MyString::~MyString(){
   delete [] storageM;
   std::cout << "\ndtor called.";
}
```

Member-wise Initialization

```
void main()
 {

    MyString s1("World"); // here is a call to constructor
    MyString s2 = s1;      // here is not a call to constructor
                           // s2 will be initialized with member of
                           // of s1

    // Point 1

    MyString s3 = "ABC";
    s1 = s3;               // This is assignment operation and every
                           //  data members of s3 will copied into s1

    // Point 2
}
```

Member-wise Initialization Problems



- **There are two problems here:**
  - **Problem 1: S2 is not a real copy of s1**
  - **Even more serious issue:**
    - **Destructor of MyString will be called twice when s1 and s2 go out of scope. In other word the destructor attempts to de-allocate storageM twice (the same place on the heap)**

How to solve the problem initialization of object?

- Here is Solution:
  - Define a special constructor, normally called copy constructor.
  - The General format for the prototype of a copy constructor is:

X (const **X**& source);

Where **X** is the class name.

Member-wise Initialization

```
class MyString
{
public:
    MyString(char *s);

    MyString(const MyString& source);   // copy constructor

    ~MyString();
private:
    char *storageM;
    int lengthM;

};
```

Definition of copy constructor:

- A copy constructor
  - Its name must be same as its class name
  - Has no return value
  - It can use member initialization list to initialize data members.
- Here is the definition of the copy constructor:

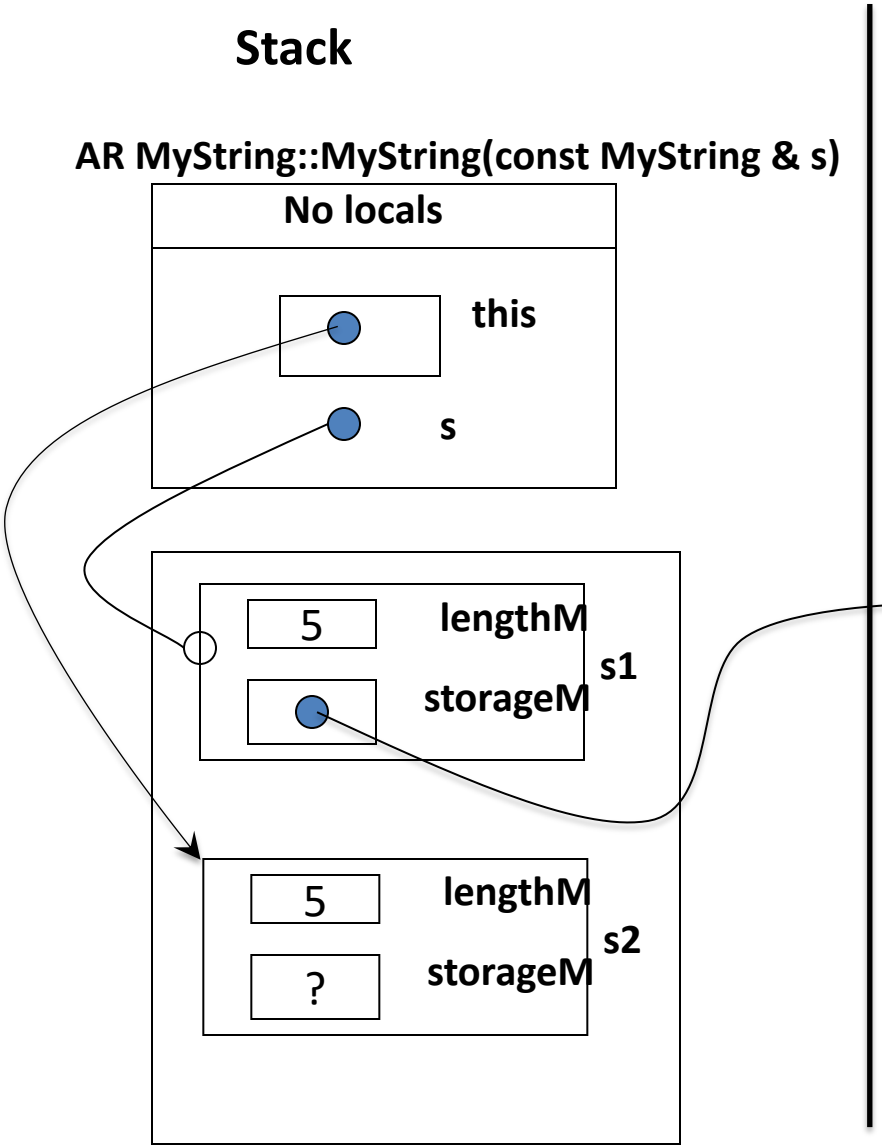
```
MyString::MyString(const MyString& s) : lengthM(s.lengthM)
{
   // point 1
   storageM = new char [lengthM+1];
   assert (storageM !=0);
   strcpy(storageM, s.storageM);
   // point 2
}
```

Using MyString Class

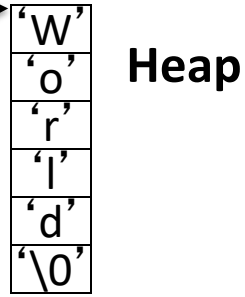
- Lets take a close look at copy constructor by drawing a few AR diagrams:
  - Two diagrams inside the copy constructor (point 1 and 2)
  - And one diagram in main (point 3)

```
void main()
{
    MyString s1("World");
    MyString s2 = s1;

    // Point 3

    MyString s3 = "ABC";
    s1 = s3;
}
```

AR at point 1 inside the copy constructor

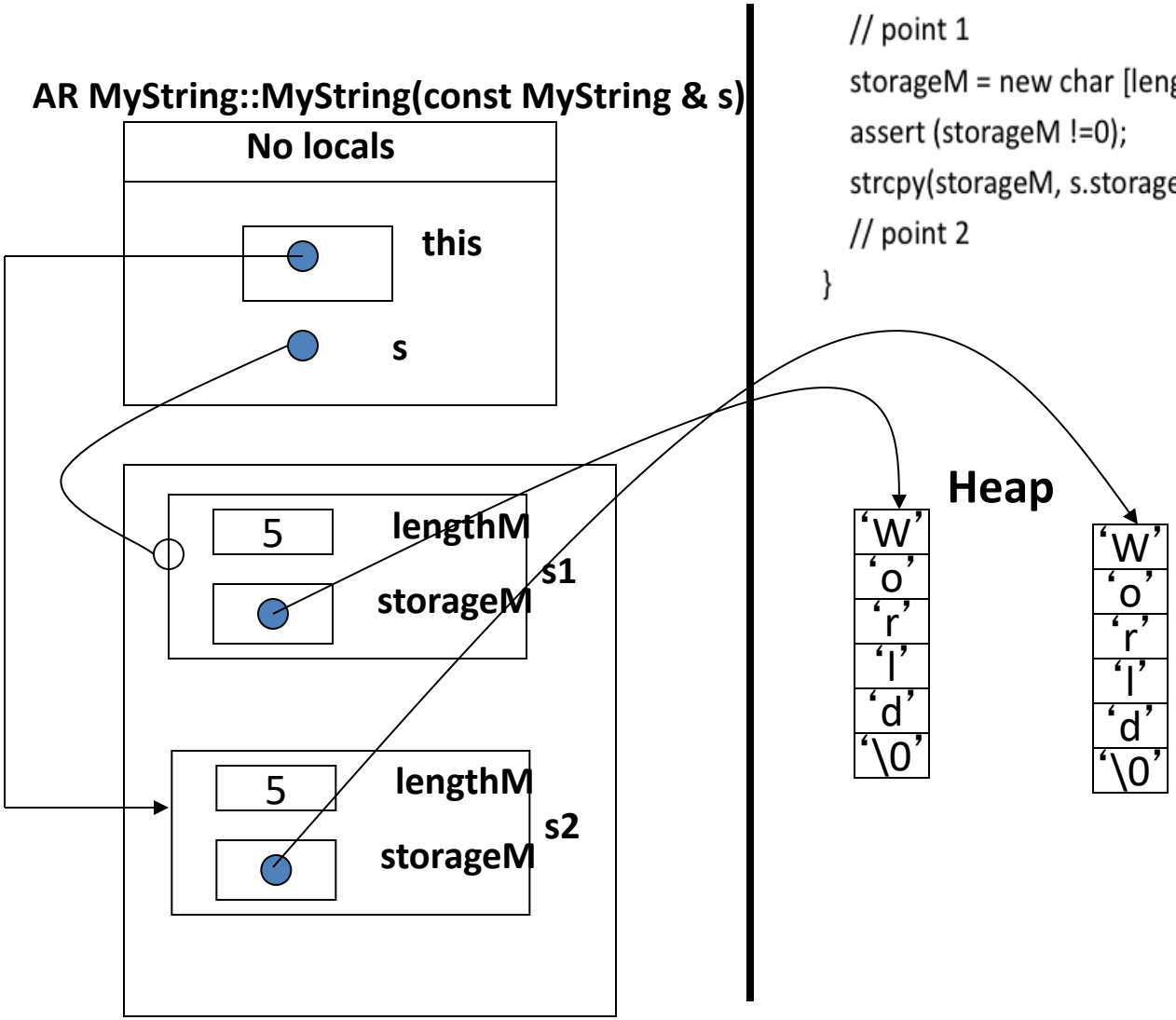



```
MyString::MyString(const MyString& s) : lengthM(s.lengthM)
{
   // point 1
   storageM = new char [lengthM+1];
   assert (storageM !=0);
   strcpy(storageM, s.storageM);
   // point 2
}
```



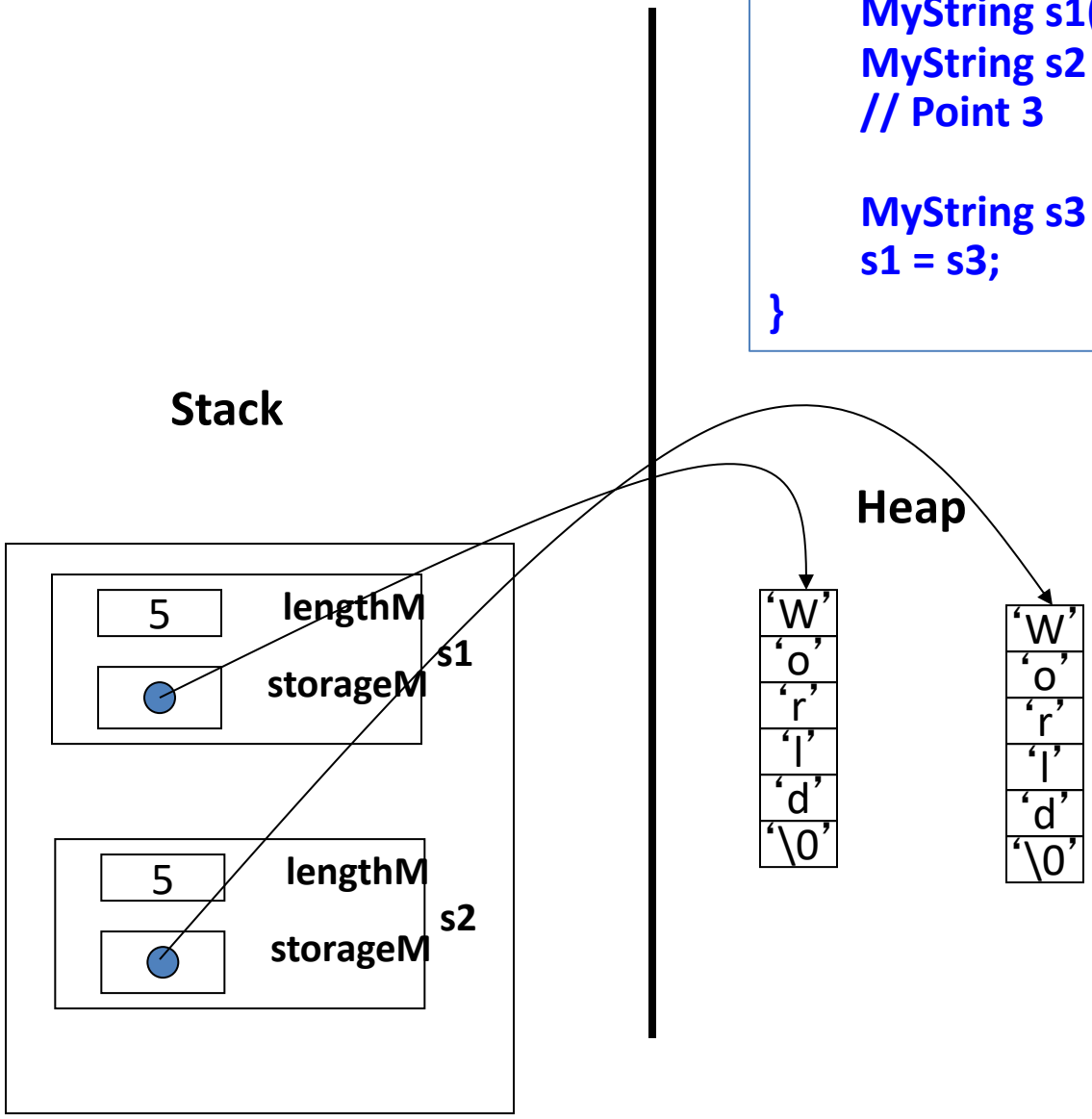
Point 2 in copy constructor

```
MyString::MyString(const MyString& s) : lengthM(s.lengthM)
{
   // point 1
   storageM = new char [lengthM+1];
   assert (storageM !=0);
   strcpy(storageM, s.storageM);
   // point 2
}
```



Point 3 inside main after call to copy constructor

```
void main() {
        MyString s1("World");
        MyString s2 = s1;
        // Point 3

        MyString s3 = "ABC";
        s1 = s3;
}
```



Other examples of call to copy constructor:

- Passing an objects to functions by value.
- Returning objects from functions by value.
- In the following example copy constructor is called twice:
  - When s1 is copied into argument of function func.
  - When local object in func is returned into main and copied into s2

```
int main(void)
{
    MyString s1("ABC");
    MyString s2 = func(s1);
    …
    return 0;
}
```

```
void func (Mystring copy)
{
    MyString  local("XYZ");
              …
    …
    return local;
}
```

- A combination of calls to the constructor and the copy constructor:

  MyString s3 = "KLM";
  - First constructor of class MyString is called to create a temporary instance of MyString object.
  - Then the copy constructor of MyString class is called to copy the temporary object into s3.

Note: number of calls to copy constructor because of compiler optimizer might be different from one compiler to another one. In case of s3 compilers are allowed to elide the call to copy constructor.

Copying Objects by Assignment

Using MyString Class

- Since in C++ initialization operation is differentiated from assignment operation, the following operation is not a call to copy constructor:

```
s1 = s3;
```

- Therefore we need another device to solve this type of member-wise copying.
- The solution is to overload the assignment operator.
- C++ allows to overload most of its operators, including: =, +, -, +=, ++, --, and may others.
- In ENCM 339 we only look into overloading assignment operator, to cope with issue of member-wise copying.
  - Details of overloading other operators will be discussed in ENSF 409
- The general format of overloaded assignment operator is:

```
X& operator = (const X& rhs);
```

Member-wise Initialization

```
class MyString
{
public:
      MyString(char *s);
      MyString(const MyString&);

      // Assignment operator
      MyString& operator =(MyString& s);

      ~MyString();
private:
      char *storageM;
      int lengthM;
};
```

Overloaded Assignment Operator:

```
MyString& MyString::operator =(MyString& s) {
        if(this != &s)
        {
          delete [] storageM;
          lengthM = s.lengthM;
          storageM=new char [lengthM+1];
          assert(storageM != NULL);
          strcpy(storageM,S.storageM);
        }
        return *this;
    }
```

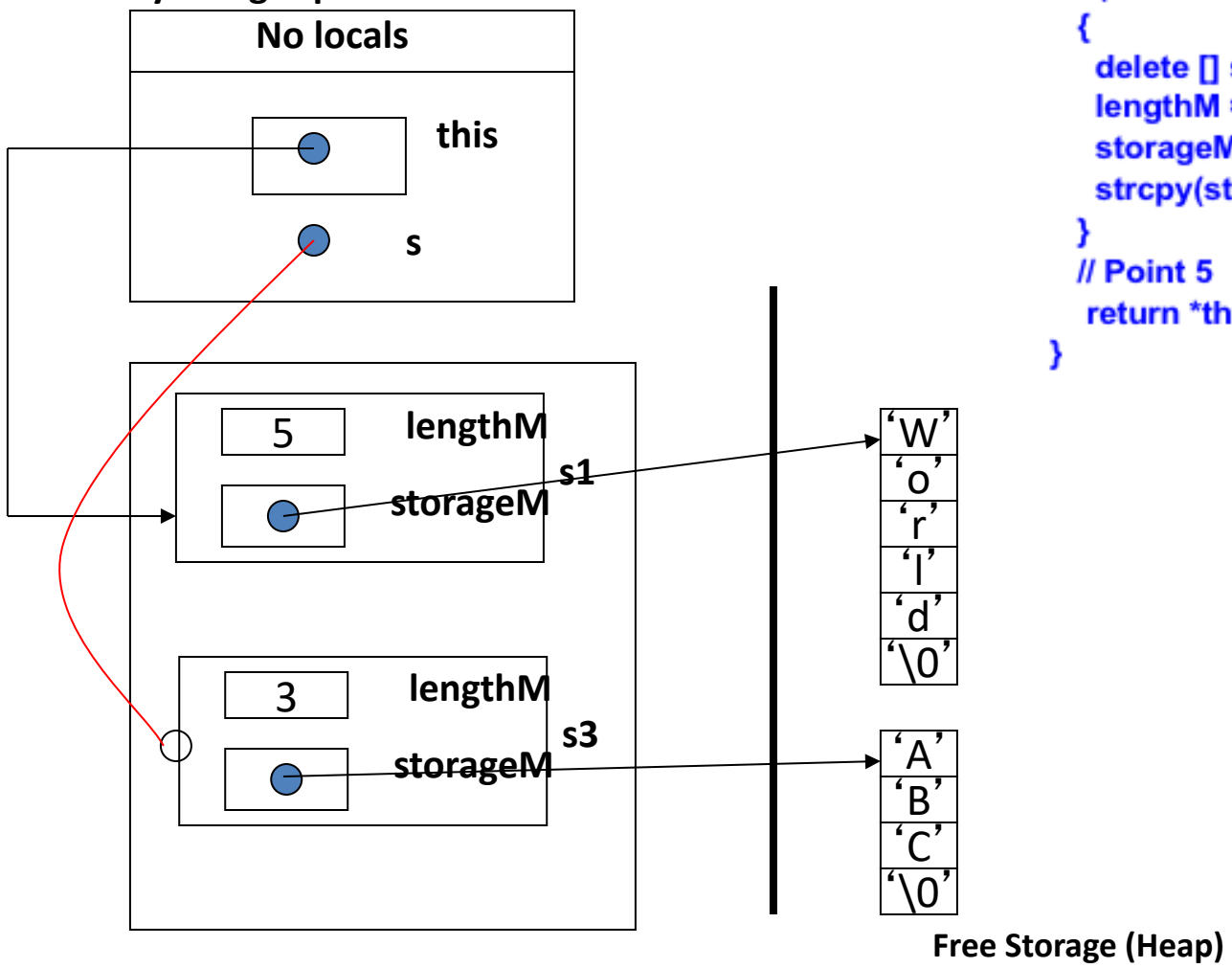
How Assignment Operator Works?

- To take a closer look at the definition of overloaded assignment operator lets draw two AR diagrams inside the overloaded operator= (next slides):
- First consider the following main function:

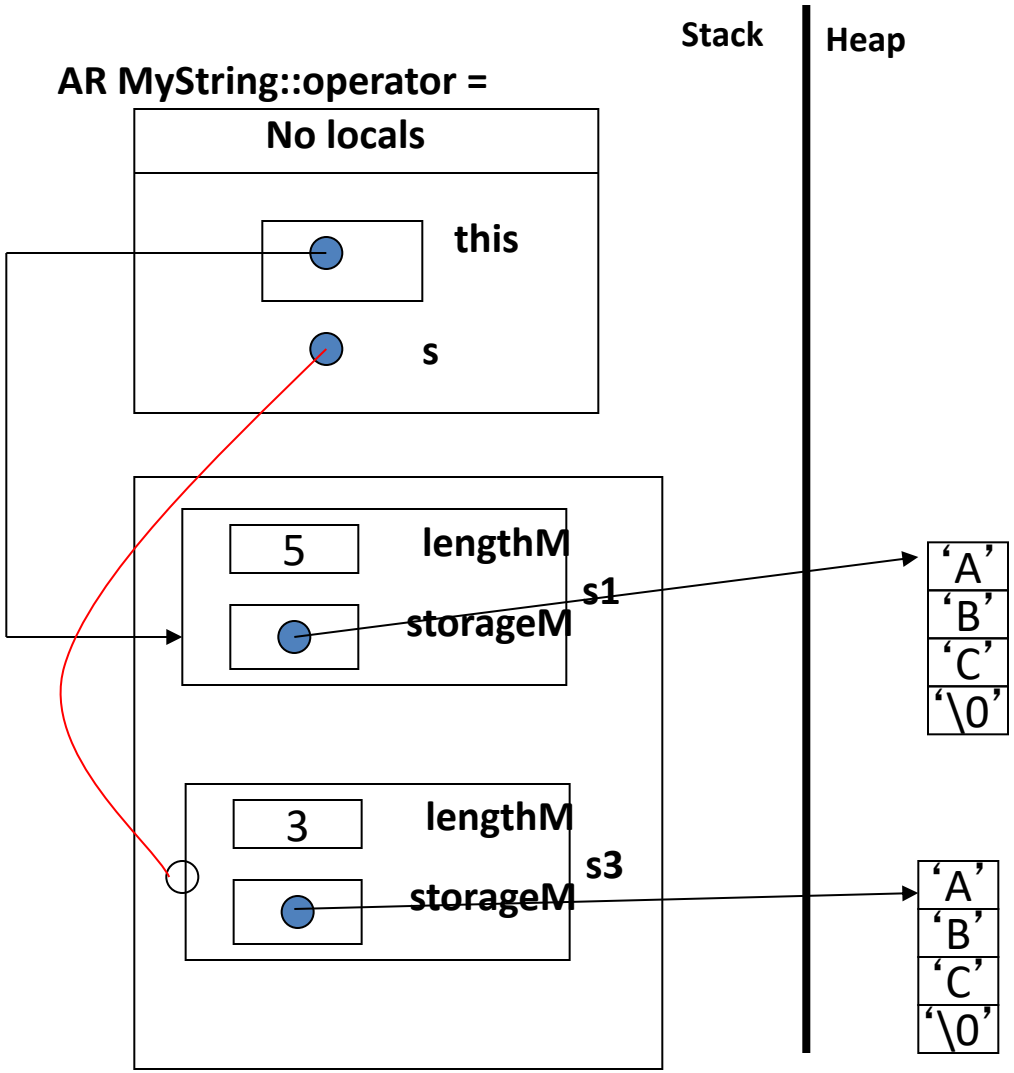
```
void main()
{
    MyString s1 (“World”);
    MyString s3 (“ABC”);
    s1 = s3;
}
```

AR at point 4 inside the assignment operator

```
MyString& MyString::operator =(MyString& s)
{
 // Point 4
  if(this != &s)
  {
    delete [] storageM;
    lengthM = S.lengthM;
    storageM=new char [lengthM+1];
    strcpy(storageM,S.storageM);
  }
  // Point 5
   return *this;
}
```



AR at point 5 inside the assignment operator



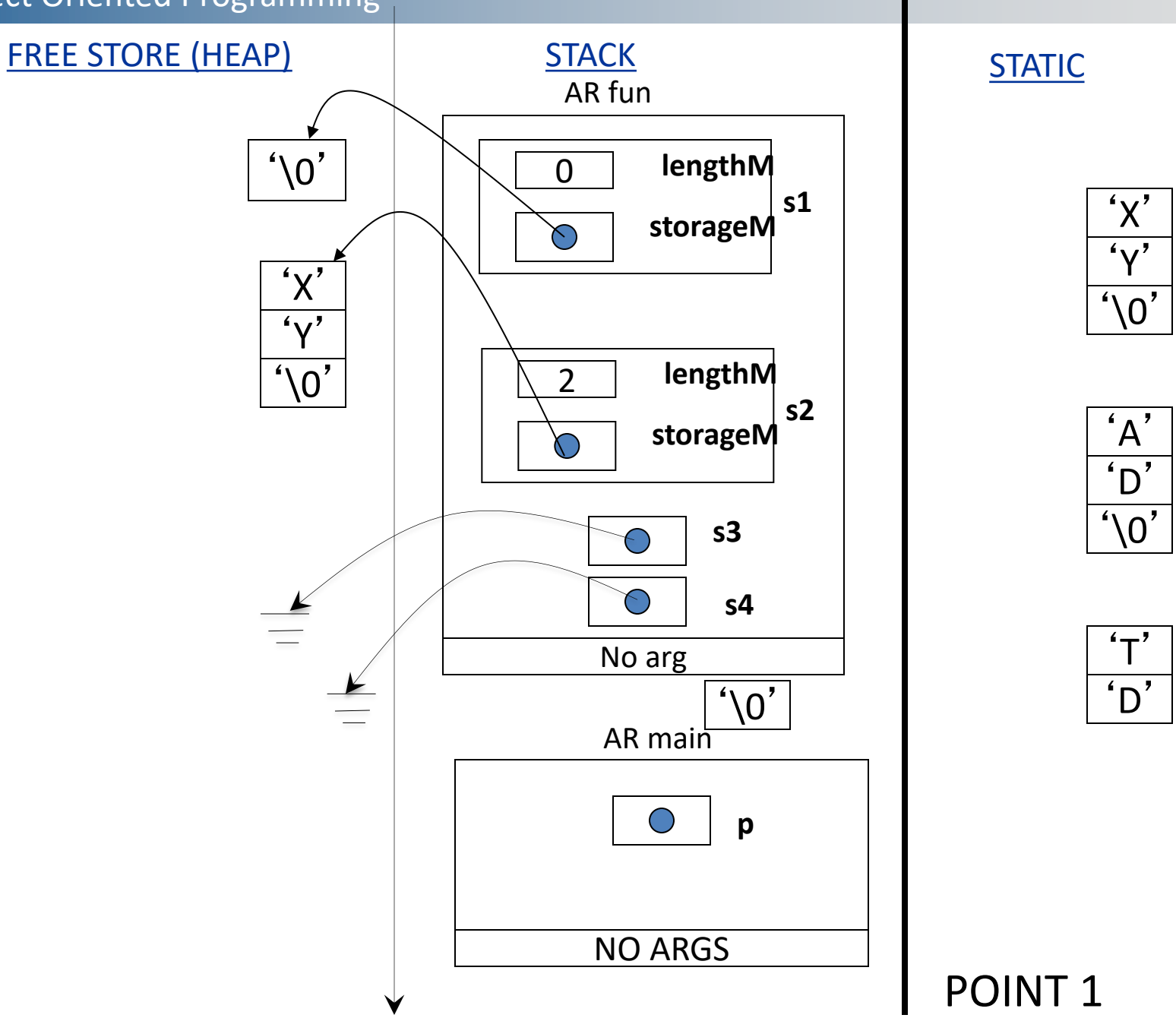

```
MyString& MyString::operator =(MyString& s)
{
 // Point 4
  if(this != &s)
  {
    delete [] storageM;
    lengthM = S.lengthM;
    storageM=new char [lengthM+1];
    strcpy(storageM,S.storageM);
  }
  // Point 5
   return *this;
}
```

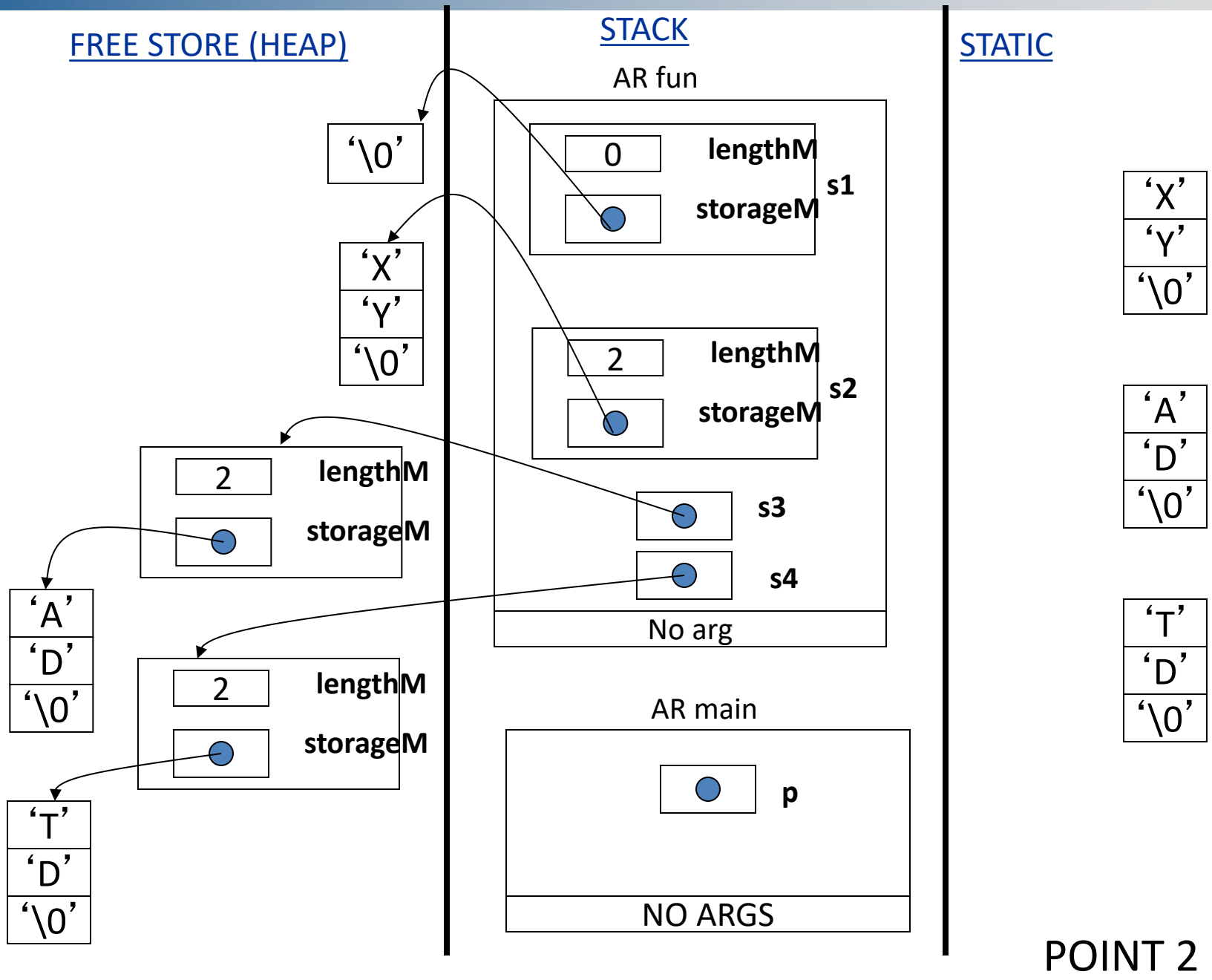
Dynamic Allocation of Objects and Their Lifetime and Scope

Using MyString Class

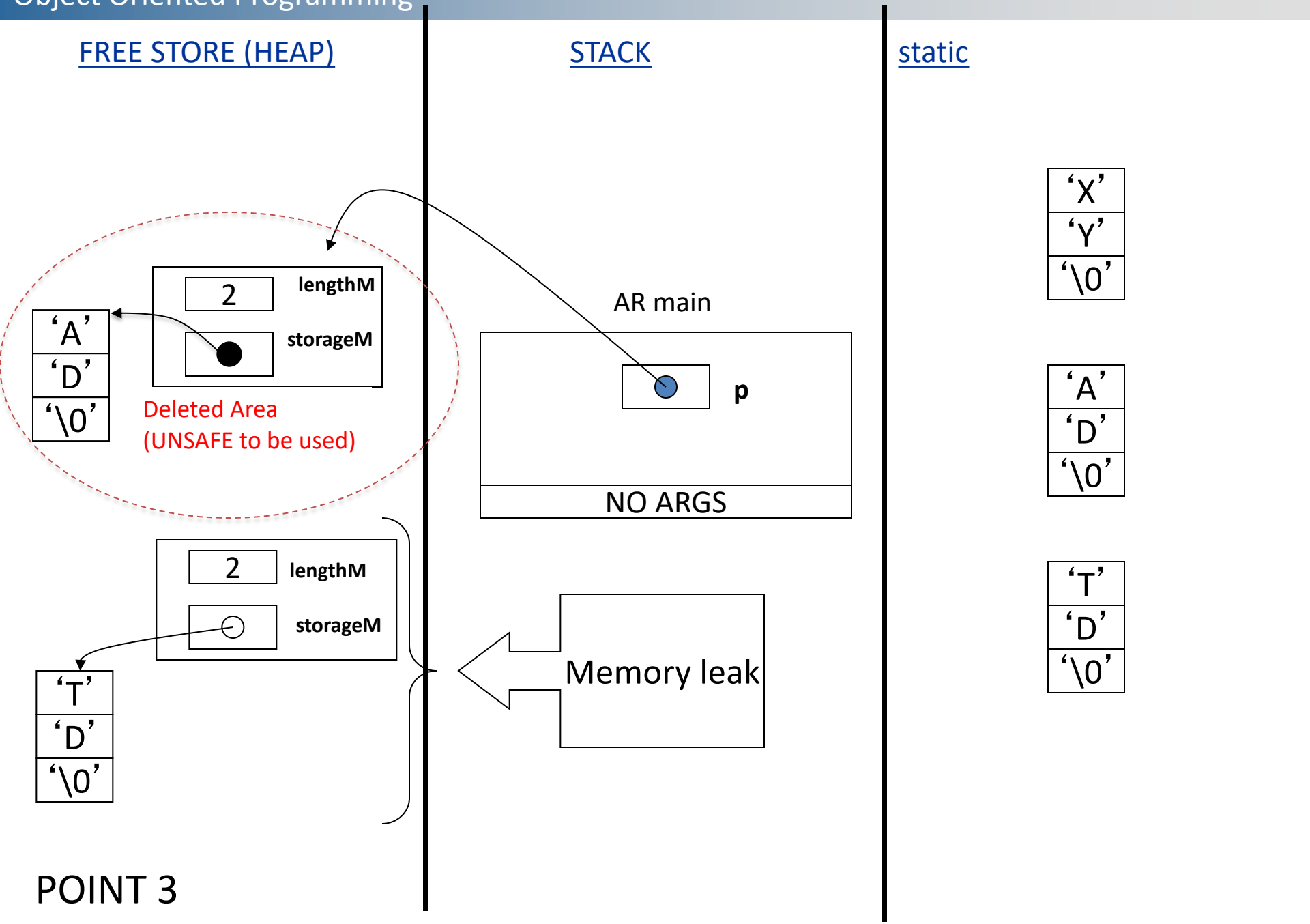
```
#include <iostream.h>
#include <string.h>
#include <assert.h>
void main()
{
        MyString * p = NULL;
        p = fun();
       // point 3

       // call to the destructor
       delete p;
       // Point 4
       return 0;
}
```

```
MyString* fun()
{
   // call to default constructor
   MyString s1;

  // call to constructor
   MyString s2("XY") ;
   MyString *s3 = NULL;
   MyString *s4 = NULL;
   // Point 1

   // call to constructor
   s3= new MyString ("AD");
  // call to constructor
   s4= new MyString ("TD");
   // Point 2
   return s3; // two calls to the destructor constructor
}
```


FREE STORE (HEAP)
STACK
AR fun
STATIC
'\0'
0
lengthM
storageM
s1
'X'
'Y'
'\0'
2
lengthM
storageM
s2
s3
s4
No arg
'\0'
AR main
p
NO ARGS
'X'
'Y'
'\0'
'A'
'D'
'\0'
'T'
'D'
POINT 1


FREE STORE (HEAP)
STACK
STATIC
AR fun
'\0'
0
lengthM
storageM
s1
'X'
'Y'
'\0'
2
lengthM
storageM
s2
2
lengthM
storageM
s3
s4
'A'
'D'
'\0'
No arg
2
lengthM
storageM
AR main
'T'
'D'
'\0'
p
NO ARGS
'X'
'Y'
'\0'
'A'
'D'
'\0'
'T'
'D'
'\0'

POINT 2

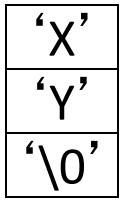

FREE STORE (HEAP)
STACK
static
'X'
'Y'
'\0'
2
lengthM
storageM
'A'
'D'
'\0'
Deleted Area
(UNSAFE to be used)
AR main
p
NO ARGS
'A'
'D'
'\0'
2
lengthM
storageM
'T'
'D'
'\0'
Memory leak
'T'
'D'
'\0'

POINT 3

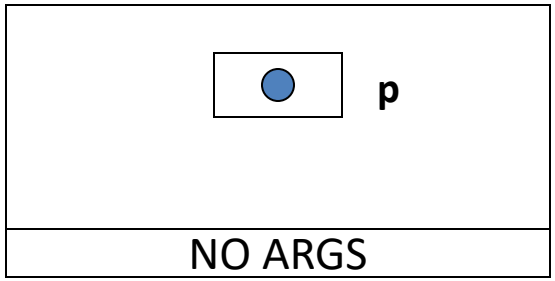
FREE STORE (HEAP)

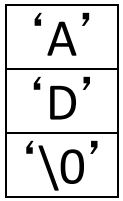
STACK

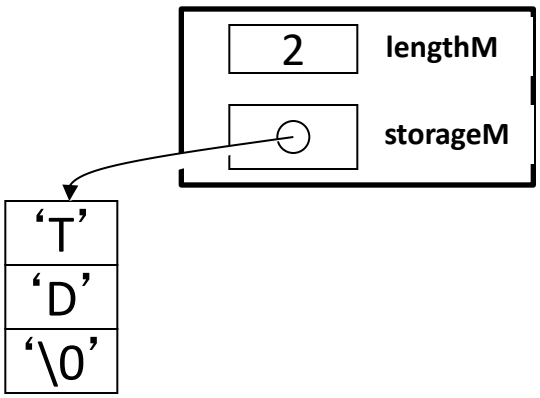
STATIC:
STRING CONSTANT AREA

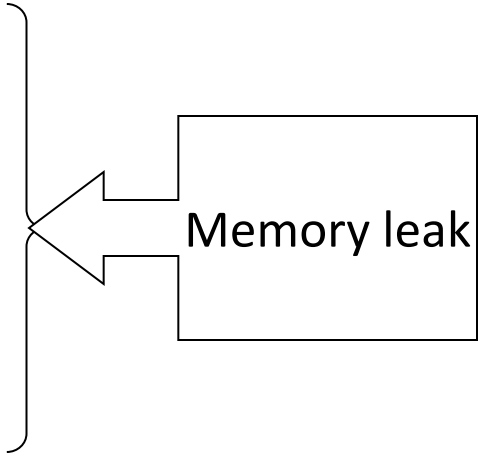


AR main









POINT 4

Questions:

- When do we need a destructor?
- When do we need assignment operator?
- When do we need copy constructor?
  - Initialization of a new object
  - Passing an object to a function, by value
  - Returning an object from a function, by value
- When do we need default constructor?
  - To initialize the object to default values
  - To be able to create array of objects
- When is constructor called?
  - When a new object is created on the stack or on the heap by new operator.
- When is destructor called?
  - When an object goes out of scope on the stack. Within a function the scope can be indicated by a pair of opening and closing braces:
  - When a pointer to an object created on the heap is deleted by delete operator.
- What is the law of Big 3?

How many times constructor (ctor), destructor (dtor), assignment operator, default constructor (default ctor), and copy constructor are called:

```
int main(void) {
        MyString s1("ABC");
        MyString s2("XY");
        {
                MyString s3 ("KLM");
                MyString *s4;
                s4 = new MyString("BAR");
                MyString s5 =s1;
                s3 = s2;
                MyString s6[2];
                delete s4;
                //Point one
        }
        // point two
        MyString s7 = fun(s1, s2, &s1);
        S2 = fun(s1, s2, &s7);
        // point three
        Return 0
}
```

```
MyString fun (MyString x, MyString& y, MyString *z){
  MyString w;
 // Some code…

 return w;
}
```

Answer:

At point one:

- 4 calls to ctor
- 1 call to copy constructor
- 1 call to copy assignment operator
- 2 calls to **default** ctor
- 1 call dtor

At point two:

- 4 more calls to dtor

At point three:

- 3 more calls to copy constructor
- 2 more calls to default ctor
- 4 more calls to destructor
- 1 more call to assignment operator

When main ends: 3 more calls to dtor

Note: number of calls to copy ctor due to use of compiler optimizer is not guaranteed.