Overloading Operators in C++

Overloading Operators

- C++ <provides a nice feature that allows basic operators such as: =, +, -, ==, <, >, etc. to be used with the objects of classes.
- In fact, almost any operator, except the following operators can be overloaded: **::**, **?:**, **.**,**.***, **sizeof, typid**
- Even operators such as new, delete can be overloaded.
- You are already familiar with overloading assignment operator. In this set of slides we will discuss how to overload other operators.

Principles of Overloading Operators in C++

Principles of Operator Overloading

- A class designer can provide a set of operators to work with objects of the class.
  - This can be achieved by defining an operator function.
- An operator function need not to be a member function, **but it must take at least one class argument**.
  - This prevents the programmer from overloading the behavior of operators for built-in data types.
- Only predefined set of C++ operators can be overloaded.
  - It means, we cannot generate or define a new operator in C++

Principles of Overloading Operator (contd.)

- Function Definition: An overloaded function can be defined same as ordinary member or non-member functions, except that an "operator" reserved word followed by operator symbol will be used as function's name.
  - If the first parameter of an overloaded function must be an object of another class, the function **MUST** be a nonmember.
  - it can be also defined as a **friend**.
- Four operator: assignment "=", subscript "[]", call "()" and member selection "->" operators are required by language to be defined as class member. These operators cannot be defined as a non-member function.

Principles of Overloading Operator (contd.)

- An operator function should not change the nature of an operator. For example, the overloaded operator function cannot convert a unary operator to a binary or vice versa.

Defining Overloaded Operators

In the next few slides we use class `String` as an example to define overloaded operators for objects of this class.

Example of overloading operators for class String

- Remember the following String class defined in previous slides and lectures. We already know how to overload operator =

```
class String {
 public:
  String();
  String(char *s);
  const char* get(){
    return storage;
  }
          …
 private:
    char * storageM ;
    int lengthM;
};
```

```
int main() {
  String s1 (“ABC”);
  String s2 (“XY”);
  s1 = s2;
  // prints ABC
  cout << s1.get()
}
```

What if we want to overload operator + for class String?

```
int main() {
  String s1 ("ABC");
  String s2 ("XY");
  s1 = s2;
  String s3;
  s3 = s1 + s2;
  // should print ABCXY
  cout << s3.get()
}
```

This operation will not be allowed unless that operator + is overloaded for class String.

We would like to use operator + to concatenate two strings

Overloading +

Overloading + Operator for Class String

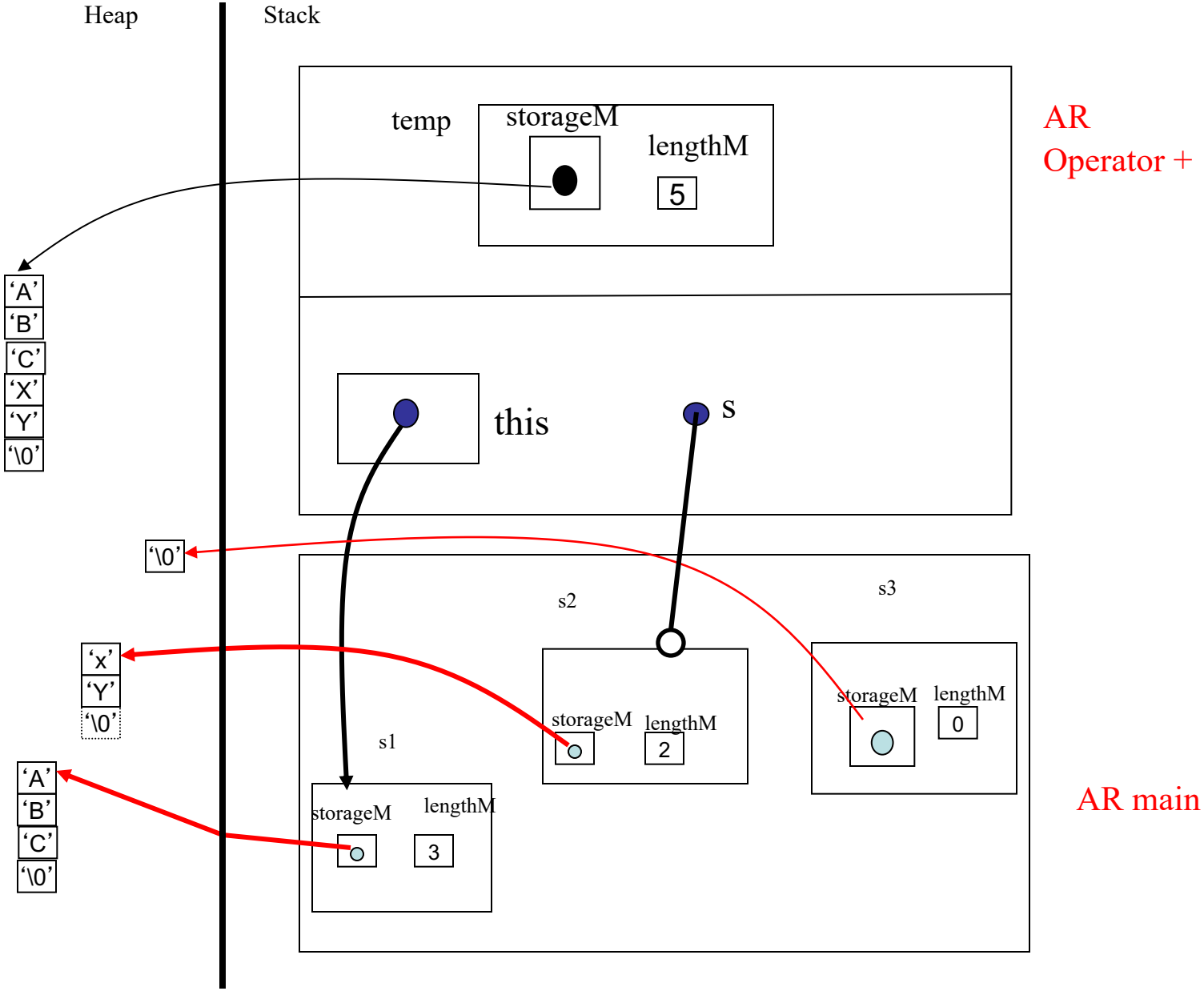
```
class String
{
  public:
     …

    String operator +(const String& s);

  private:
    char * storageM;
    int lengthM;
};
```

```
int main() {
  String s1 ("ABC");
  String s2 ("XY");
  String s3;
  s3 = s1 + s2;
  // POINT TWO
  ….
}
```

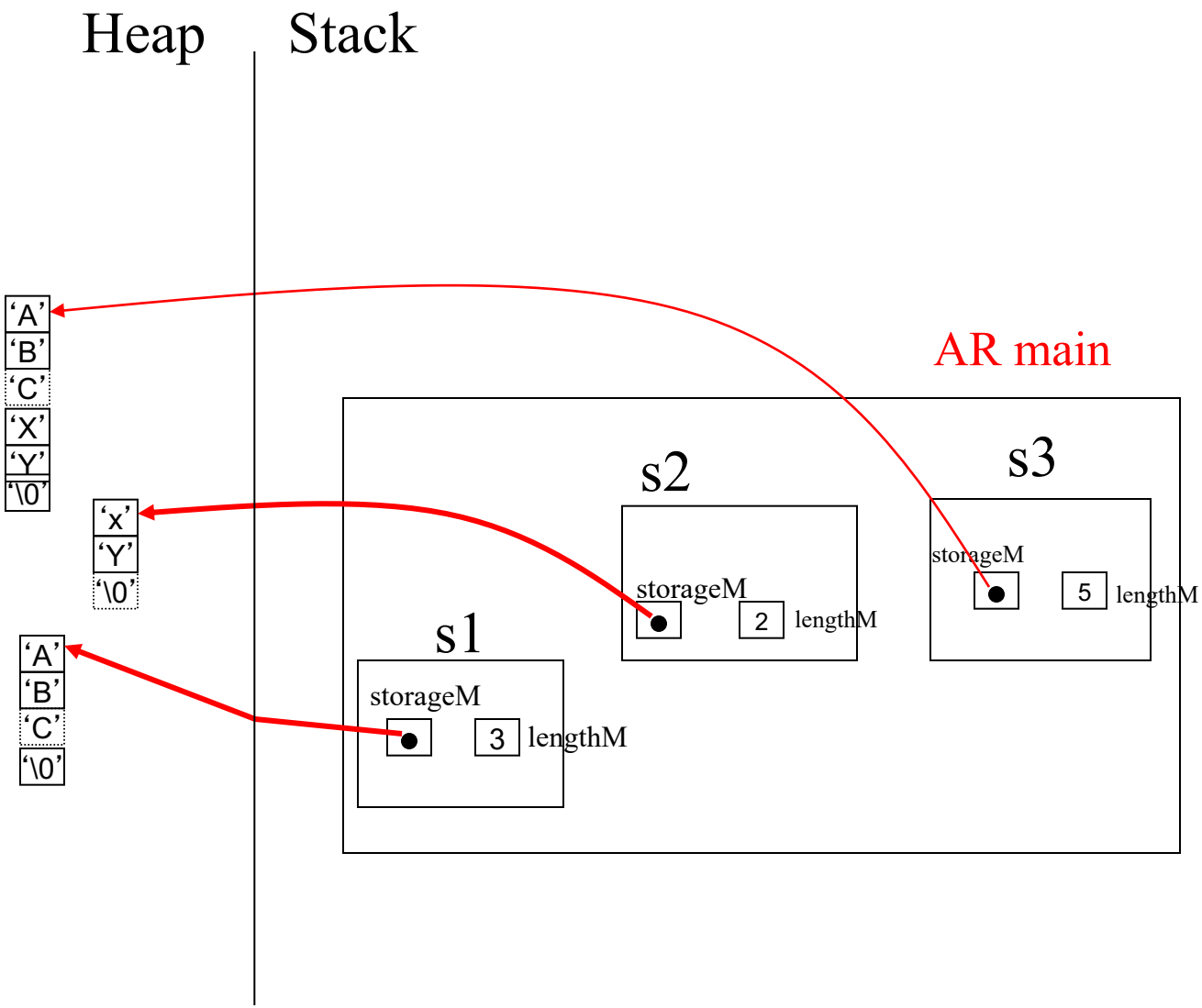
```
String String::operator + (const String& s)
{
String  temp;
temp.lengthM = lengthM + s.lengthM;
delete [] temp.storageM;
temp.storageM = new char[temp.lengthM+1];
strcpy(temp.storageM,  storageM);
strcat(temp.storageeM,   s.storageM);
// POINT ONE
return temp;
}
```

Let's try to draw a memory diagram for points ONE and Two.

AR Diagram for Point ONE



AR Diagram for Point TWO



Overloading +=

Overloading +=

- += operator in the String class can be overloaded to be used for string concatenation (see the function definition in the next slide):
  - String s1 = " Hello ";
  - String s2 = "World";
  - s1 += s2;
- += operator is used to concatenate the strings s1 and s2. Therefore, s1will change to: "Hello World".
- **Class Exercise: Let's Write the definition of overloaded operator +=.**

Overloading +=

```
String& String::operator += (const String& s) {
    length += s.length;
    char *p = new char[length+1];
    assert (p !=0);
    strcpy (p , storageM);
    strcat(p , s.storageM);
    delete storageM;
    storageM = p;
    return *this;
}
```

Member or Non-member Overloaded Operators

Member or nonmember?

- If the first parameter of an overloaded function must be an object of another class, the function must be a nonmember.
  - However, if the function needs direct access to the data members, it can be also defined as a friend.
- **Let's look at the overloaded operator << for class String objects.**
- **Questions to be asked:**
  - **Why do we need such operator to be available to class String?**
  - **What type of object is on the left side of the operator <<?**
  - **What type of object is on the right side of the operator <<?**
  - **What type of object should be return from operator function? Why?**
  - **Should this overloaded-operator function be a member of class String? Yes or No; Why?**

Overloading <<

- This function MUST be a global function/non-member

```
ostream&  operator << (ostream& os, String& s)
{
    return os << s.storageM;
}
```

- We can declare this function as a friend for class String:

```
class String {
   ostream&  operator << (ostream& os, String& s);
   …
};
```

Another Class Exercise

- **Let's try to overloaded operator >> for class String objects.**
  - Here are examples of using operator >> for objects of class String:

```
int main(){
    String s1, s2;
  int age;
    cout << "Please enter your name: "
  cin >> s1;
    cout << "Please enter your age: "
  cin >> age;
    cout << "Please enter your address: "
  cin >> s2;
  cout << s1 << s2;
}
```

Overloading >>

- This function also MUST be a global function/non-member

```
istream&  operator >> (istream& is, String& s)
{
    return is >> s.storageM;
}
```

- We can declare this function as a friend for class String:

```
class String {
   istream&  operator >> (istream& is, String& s);
   …
};
```

Overloading []

Member or nonmember (Continued)

- As mentioned earlier, subscript operator“[]”, is required by language to be defined as class member
- **Let’s write the definition this operator for class String.**
- **Questions to be asked:**
  - **Is this operator Unary or Binary?**
  - **Should this operator be a member of the class. Why?**
  - **What should be the return type of the function. Why?**
  - **What should be the arguments of the the function?**

Overloading []

```
char& String::operator [ ](int index)
{
  assert (index >= 0 && index <length);
  return storageM[index];
}
```

Overloading Type Conversion
&
Explicit vs Implicit Type Conversion

Implicit Type conversion

In C++, constructors with only one argument act as an implicit type conversion operator to convert the given argument to the type of the class.

```
String::String (int len){
  storageM = new char [len + 1];
  …
}
```

```
String s = 100;    // implicit type conversion
```

Explicit Type Conversion

- What is Explicit type conversion and when do we need this type of overloaded operator.

```
char* st = (char*) s;
```

- What is the solution? Let's write the operator (char*) for class String:

Questions to be asked:

- Is this a unary or binary operator?
- Can be this operator a non-member function?
- What should be the return type of this operator-overloaded?
- What should be returned?

Explicit Type Conversion

- Explicit type cast can be overloaded:

Notice: no return type

```
String::operator char* ()
{
   return storageM;
}
```

- Note: no return type

```
--------------------------------
String s ("ABCD");
char* st = (char*) s;
```

Overloading Increment and Decrement Operators

Overloading ++ and --

- Now let's try to overload prefix and post fix increment operators ++ and -- for class String.
- Here are the questions to be answered:
  - Is this a unary or binary operator?
  - How is compiler supposed to recognize a post-fix from refix?
  - Here is the answer:

```
class String {
 public:
  String();
  String(char *s);
      …
 private:
  char* cursorM;
  char * storagM ;
  int lengthM;
```

```
String::String(const char *s)
 : length((int)strlen(s))
{
 storageM = new char[length + 1];
 strcpy(storageM, s);
 cursor = storageM;
}
```

```
// prefix
 char String::operator ++ ()
 {

    …
 }

// post-fix
 char String::operator++(int)
 {
  char ret = *cursorM;
  cursorM++;
  return ret;
 }
```