Function Overloading

- In C++, two or more functions can share the same name as long as their parameter declarations are different.

- The functions that share the same name are said to be overloaded, and this feature is referred to as function overloading.

Function Overloading

- Function overloading is the process of using the same name for two or more functions.

- The secret to overloading is that each redefinition of the function must use either-
  - different types of parameters
  - different number of parameters.
  - Return type may be same or different

Function Overloading & Ambiguity

Two functions cannot be overloaded when the only difference is that one takes a reference parameter and the other takes a normal, call-by-value parameter.

- void f(int x);
- void f(int &x); // error

In C++, unsigned char and char are not inherently ambiguous.

- char myfunc(unsigned char ch);
- char myfunc(char ch);

Function with default parameter value cannot be overloaded, if the number of parameters excluding default are same.

```
int sum(int a, int b = 20)
{
    return a + b;
}

/*int sum(int a)   //need to discard this definition
{
    return a+900;
}*/
```

```
int main()
{
    int num1= 10;
    cout<<"\n sum = "<<sum(num1);  //ambiguous
    return 0;
}
```

Constructor

- It is used to initialize the state of the object and acquire resources
- Function written by programmer but called by Compiler
- Public member Function
- Name is same as class name
- Do not have return data type, not even void
- Used for implicit initialization of object at the time of creation
- Can be overloaded
- Default constructor provided by compiler, if any type of constructor is not written in the class
- Supports default arguments; Constructor with all default arguments is equivalent to default constructor

Destructor

- It is used to release resources acquired by object
- Function written by programmer but called by Compiler
- Name is standard: ~class name
- Do not have return data type, not even void
- Do not take any parameter.
- Used for implicit un-initialization of object before it gets destroyed
- Can't be overloaded
- Default destructor provided by compiler, if not written in the class

This pointer

- Address of an **object** is passed implicitly to a member function, called on that **object**.
- Every member function of class has hidden parameter : the **this pointer.**
- Pointer **this** holds the address of the **object** invoking the member function.
- Pointer **this** is available only in member functions of the class.
- *Using the pointer **this** a member function knows on what **object** it has to work.*
- **this** is a keyword in C++
- **this** points to an individual object.
- **this** is a constant pointer
- The identity of an object is **this** pointer Because addresses are always unique.

Operator Overloading

Need to overload operator-

- Operator Overloading is a feature of C++ because of which additional meanings can be given to existing operators for user-defined datatypes.
- **operator is keyword in C++ which is used to implement** operator overloading.
- Operator overloading feature makes ADT's more natural & closer to built in data types.

Rules of Overloading Operator

- You cannot create new operators, only can overload existing ones.
- precedence or associatively of an operator can not be changed
- Meaning of operator should be similar as for built-in data types
- Following operators can NOT be overloaded
  - : . :: ?: **sizeof**

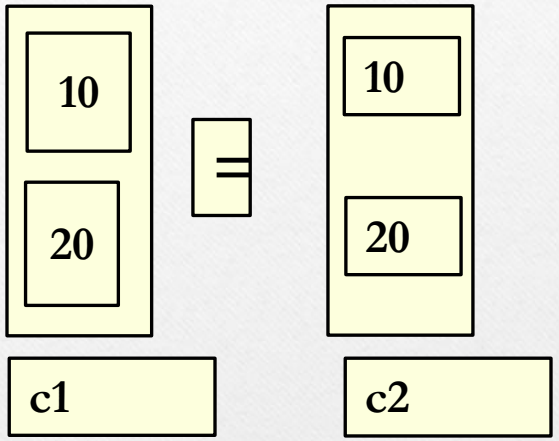
Syntax of overloading operator

- operator functions are non static member functions of a class, with following format

```
Complex Complex :: operator + (Complex &)

Complex Complex :: operator - (Complex &)
```

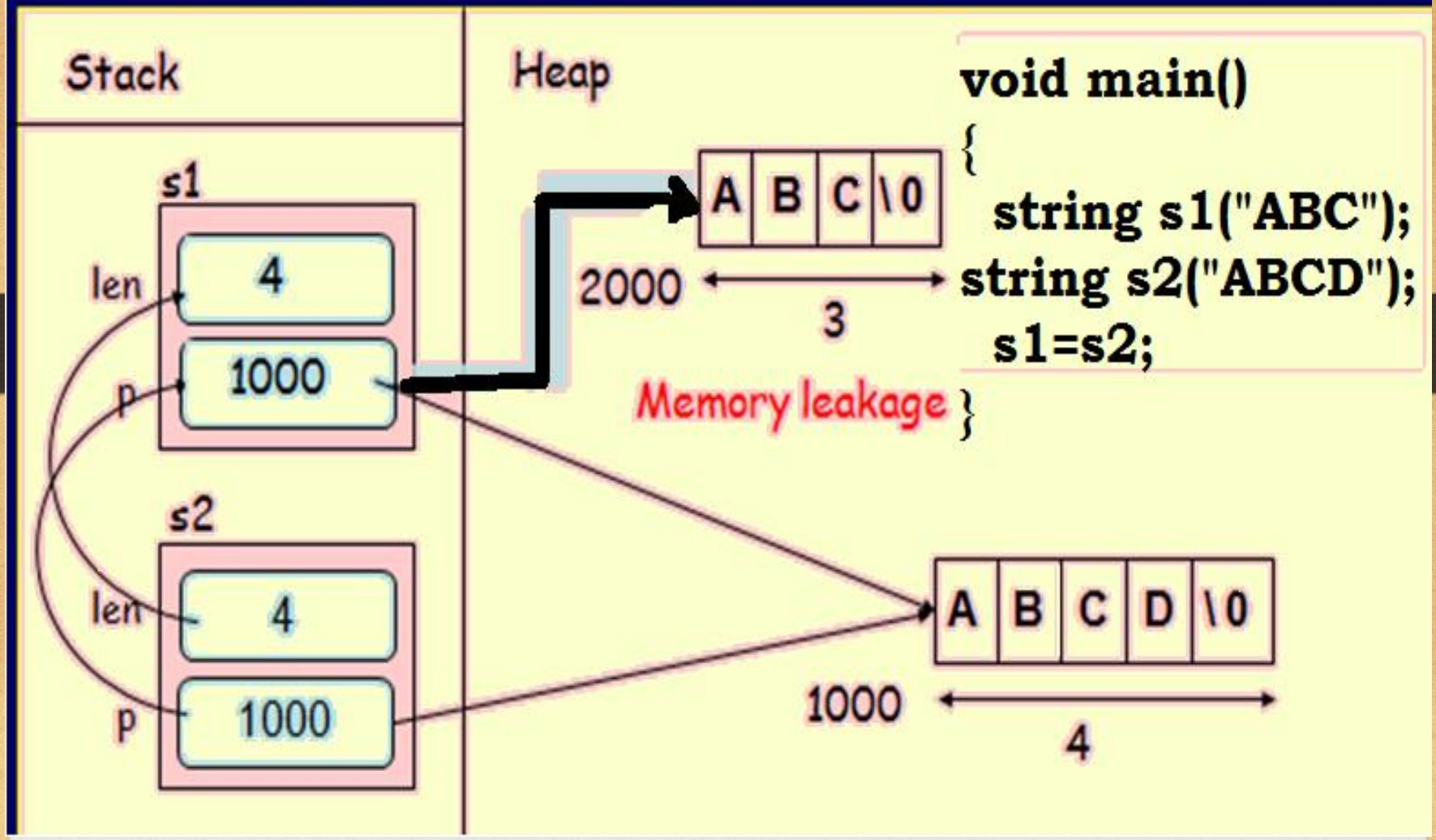
Overloading = operator

```
void main()
{
     complex c1, c2(10,20);

     c1 = c2 ;
     c1.Display();
}
```



- Default definition is provided by compiler
- Performs bit-wise copy of data from one object to another
- Ok for simple classes that do not have resources pointers
- Fails for classes that have resources pointers

- Default definition provided by compiler doest NOT work for classes having resource pointers



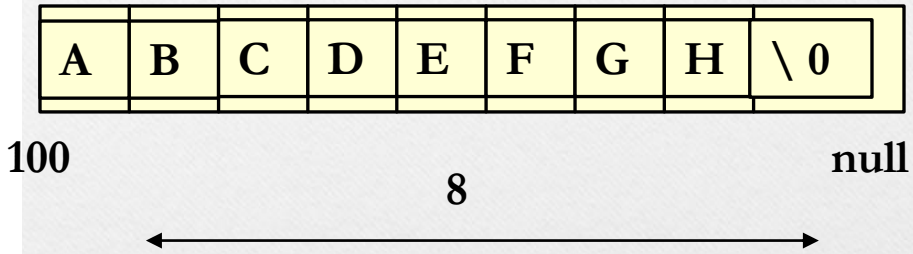
Overloading = operator

- Default definition provided by compiler doest NOT work for classes having resource pointers
- Provide our own definition. Following are the steps to be taken
  - ✓ Release the existing memory
  - ✓ Get new memory block of appropriate size
  - ✓ Copy the value to new memory

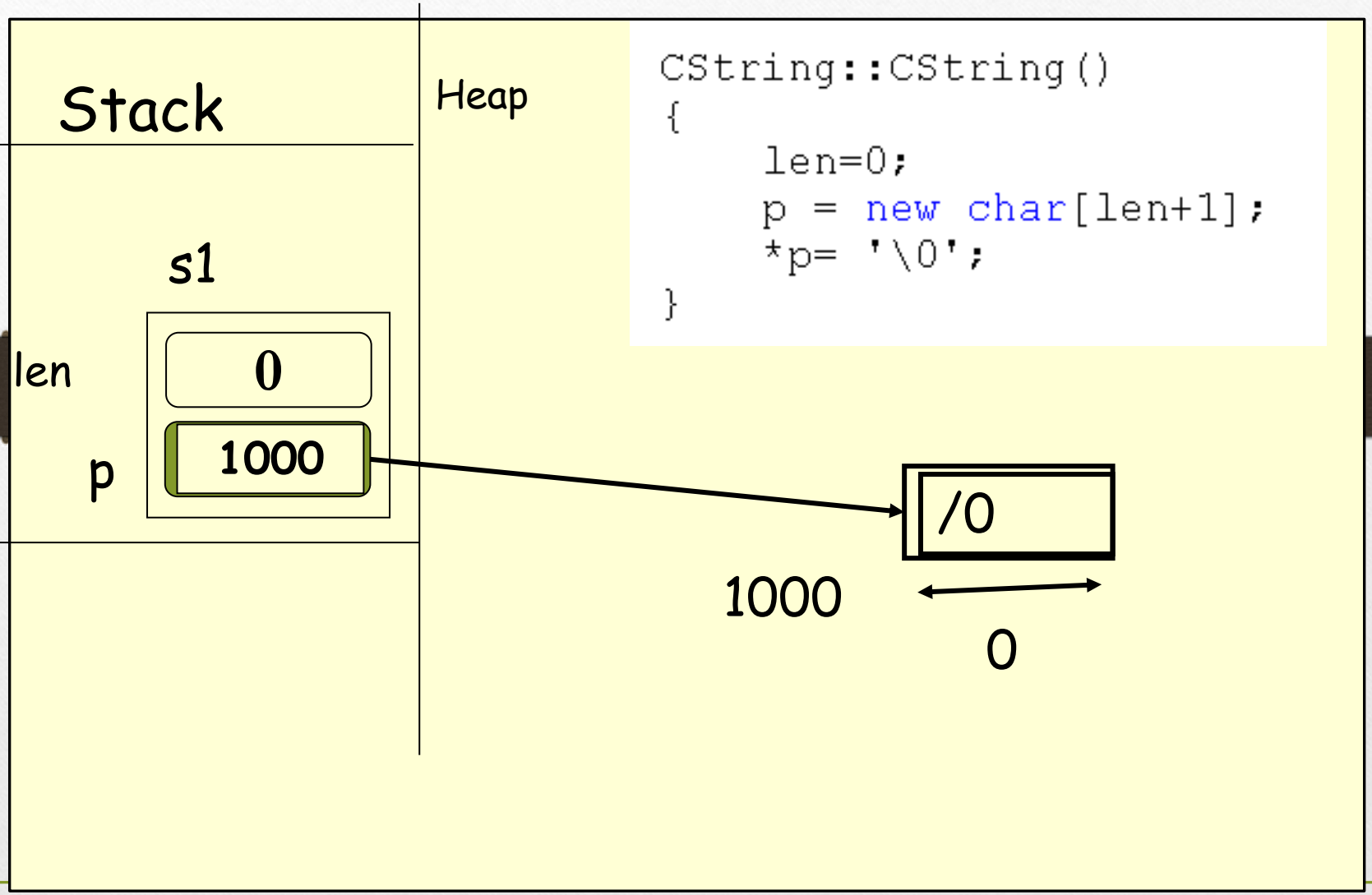
String Class & Copy Constructor

String Class

```
class CString
{
    char *p;
    int len;

public:

    CString();

    void Display();
};
```



String Class: Memory Layout

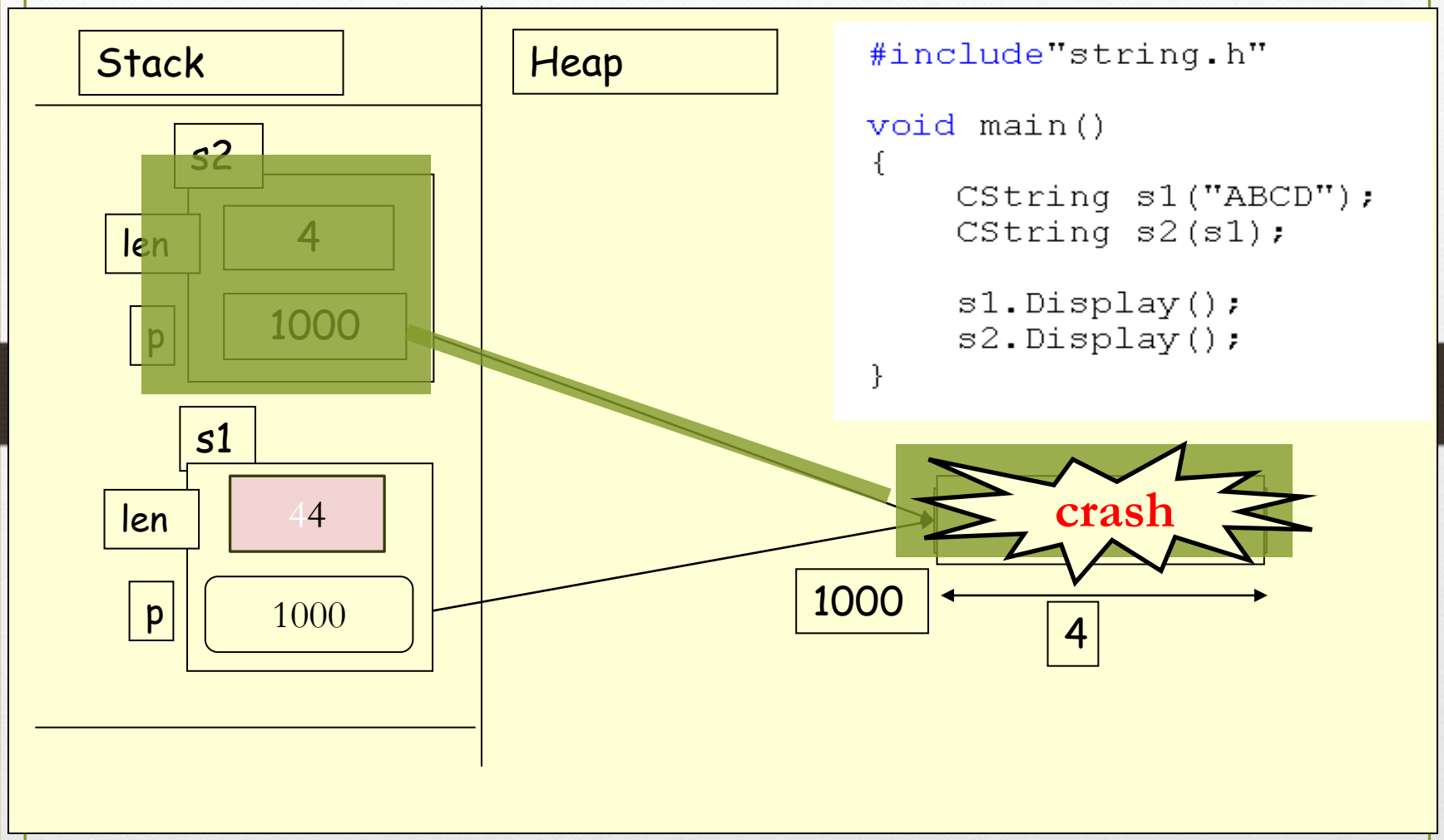



```
CString::CString()
{
     len=0;
     p = new char[len+1];
     *p= '\0';
}
```

String Class : Constructors

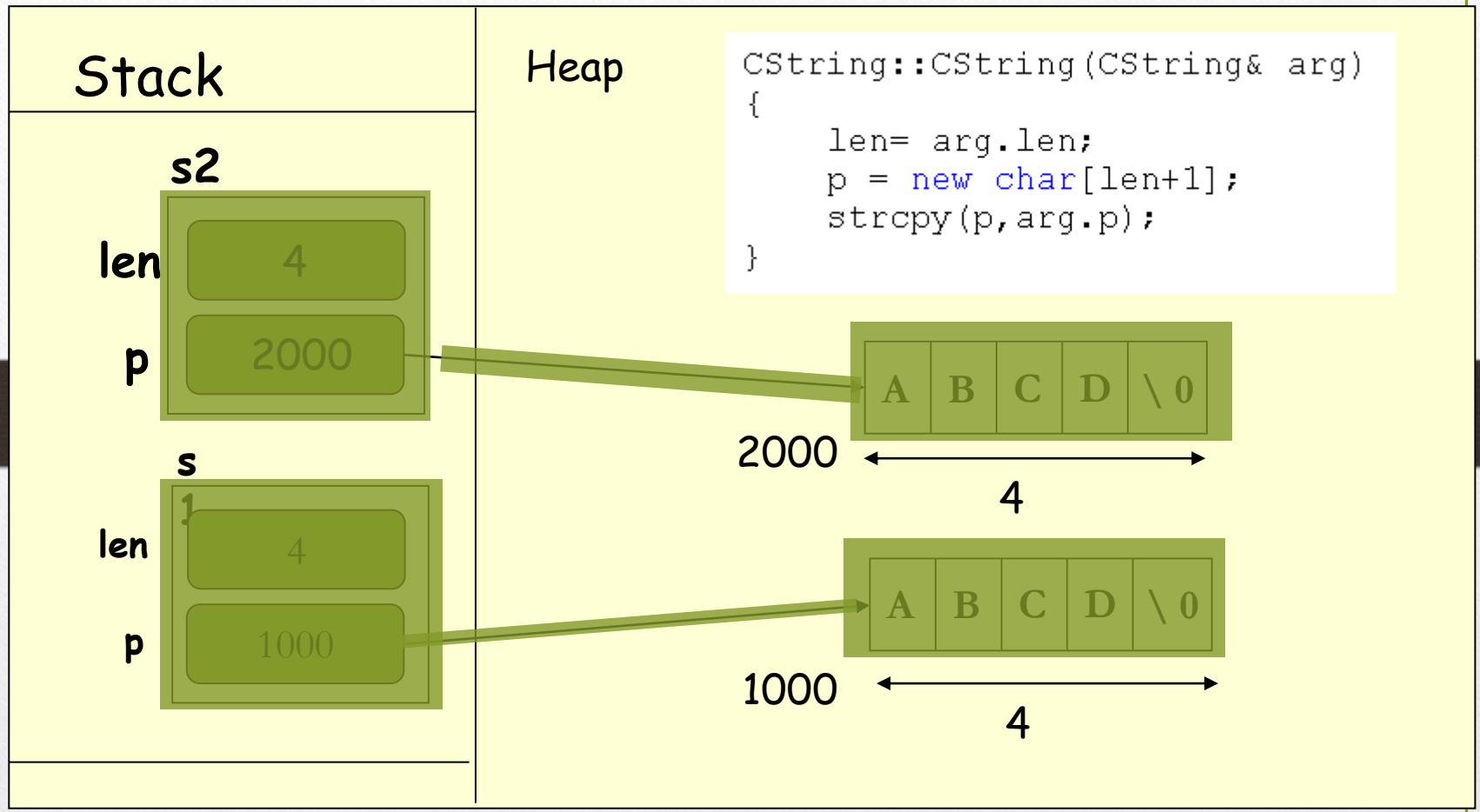
String Class : Destructor

Problem with Default Copy constructor




```
#include"string.h"

void main()
{
     CString s1("ABCD");
     CString s2(s1);

     s1.Display();
     s2.Display();
}
```

Solution


Stack
Heap
CString::CString(CString& arg)
{
len= arg.len;
p = new char[len+1];
strcpy(p,arg.p);
}
s2
len
4
p
2000
A B C D \ 0
2000
4
s
1
len
4
p
1000
A B C D \ 0
1000
4

Default functions in a class

- Functions given by compiler in each class:
  - Constructor: called when object is created
  - Destructor: called when object dies
  - Copy Constructor: called when object is created using another object
  - Assignment operator = : called when one object is assigned to another object

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

- Declaring and Defining
  - A copy constructor always has one (1) parameter
  - Must be the same type as the object being copied to.
  - Always passed by reference (because pass by value would invoke the copy constructor again).

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
- 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). Here both objects will point to the same item. Here we need a deep copy.