Object – Oriented Programming
Week 14, Spring 2009

# Smart Pointer

Weng Kai
http://fm.zju.edu.cn
Wednesday, May 27, 2008

# Exception specifications

- Declare which exceptions function *might* raise
- Part of  function prototypes

```
void abc(int a) : throw(MathErr) {
   ...
}
```

- Not checked at compile time
- At run time,
  - if an exception not in the list propagates out,

    the `unexpected` exception is raised

# Failure in constructors:

- No return value is possible
- Use an “uninitialized flag”
- Defer work to an Init() function

*Better*: Throw an exception

# Failure in constructors...

If you constructor can't complete, throw an exception.

- Dtors for objects whose ctor didn't complete *won't be called*.
- Clean up allocated resources before throwing.

# Two stages construction

- Do normal work in ctor
  - –Initialize all member objects
  - –Initialize all primitive members
  - –Initialize all pointers to 0
  - –NEVER request any resource
    - File
    - Network connection
    - Memory
- Do addition initialization work in Init()

# Exceptions and destructors

Destructors are called when:

- Normal call: object exits from scope
- During exceptions: stack unwinding invokes dtors on objects as scope is exited.

*What happens if an exception is thrown in a destructor?*

# Exceptions and destructors...

Throwing an exception in a destructor that is itself being called as the result of an exception will invoke `std::terminate()`.

- Allowing exceptions to escape from destructors should be avoided.

# Programming with exceptions

Prefer catching exceptions by reference

- Throwing/catching by value involves slicing:

```
struct X {};
struct Y : public X {};
try {
   throw Y();
} catch(X x) {
   // was it X or Y?
}
```

# Programming with exceptions...

Throwing/catching by pointer introduces coupling between normal and handler code:

```
try {
   throw new Y();
} catch(Y* p) {
   // whoops, forgot to delete..
}
```

# Catch exceptions by reference:

```
struct B {
  virtual void print() { /* … */ }
};
struct D : public B { /* … */ };
```

```
try {
   throw D(“D error”);
}
catch(B& b) {
  b.print() // print D’s error.
}
```

# Exception Hierarchies

Use inheritance hierarchies for exceptions

Problem:

```
try {
  … throw SomethingElse();
}
catch(This& t) { /* … */ }
catch(That& t) { /* … */ }
catch(Other& t) { /* … */ }
```

# Exception Hierarchies

```
class B {};
class D1 : public B {};
class D2 : public B {};
...
try {
  … throw D1();
}
catch(D2& t) { /* catch specific class here */ }
catch(B& t) { /* anything else here. */ }
```

# Unexpected exceptions

- *Exception specification* defines the exceptions a function will throw:

```
void f() throw(X, Y) {/* may throw X and Y */}
void g() throw() {/* throws no exceptions  */}
void h() {/* may throw any exception*/}
```

*What if* f() *throws something else?*

*What if* g() *throws an exception?*

# Unexpected exceptions...

- Exceptions not in the exception specification are *unexpected.*
- Unexpected exceptions become a call to std::unexpected().
- Offers a guarantee (and firewall) to callers.
- unexpected() behavior can be intercepted.

```
#include <exception>
void my_handler() {
  std::cout << "unexpected exception!\n";
  exit(1);
}
void f() throw(X, Y) {
  throw Z(); // whoops! Throwing Z
}
void main() {
  std::set_unexpected(my_handler);
  try {
    f();
  }
  catch (…) {
    std::cout << “caught it!” << endl;
  }
}
```

# Uncaught exceptions

- If an exception is thrown by not caught `std::terminate()` will be called.
- terminate() can also be intercepted.

```
void my_terminate() { /* … */ }
…
set_terminate(my_terminate);
```

# Exceptions wrapup

- Develop an error-handling strategy early in design.
- Avoid over-use of try/catch blocks. Use objects to acquire/release resources.
- Don’t use exceptions where local control structures will suffice
- Not every function can handle every error.

# Exceptions wrapup...

- Use exception-specifications for major interfaces.
- Library code should not decide to terminate a program. Throw exceptions and let caller decide.

# Uncaught exceptions

- If an exception is thrown by not caught `std::terminate()` will be called.
- terminate() can also be intercepted.

```
void my_terminate() { /* … */ }
…
set_terminate(my_terminate);
```

# Comparison w/ Java

- Can throw anything
- No final
- No throws
- … for catching all
- No stack trace print out after termination

# Putting it All Together

Templates

Inheritance

Reference Counting

Smart Pointers

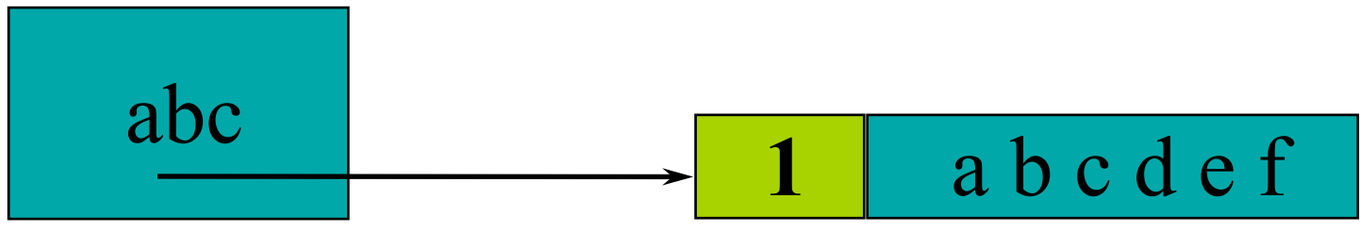
Reference: *C++ Strategies and Tactics*, Robert Murray, 1993

# Goals

- Introduce the code for maintaining reference counts
  - A reference count is a count of the number of times an object is shared
  - Pointer manipulations have to maintain the count
- Class UCObject holds the count
  - "Use-counted object"
- UCPointer is a *smart pointer* to a UCObject
  - A smart pointer is an object defined by a class
  - Implemented using a template
  - Overloads  operator-> and unary operator*

# Reference counts in action

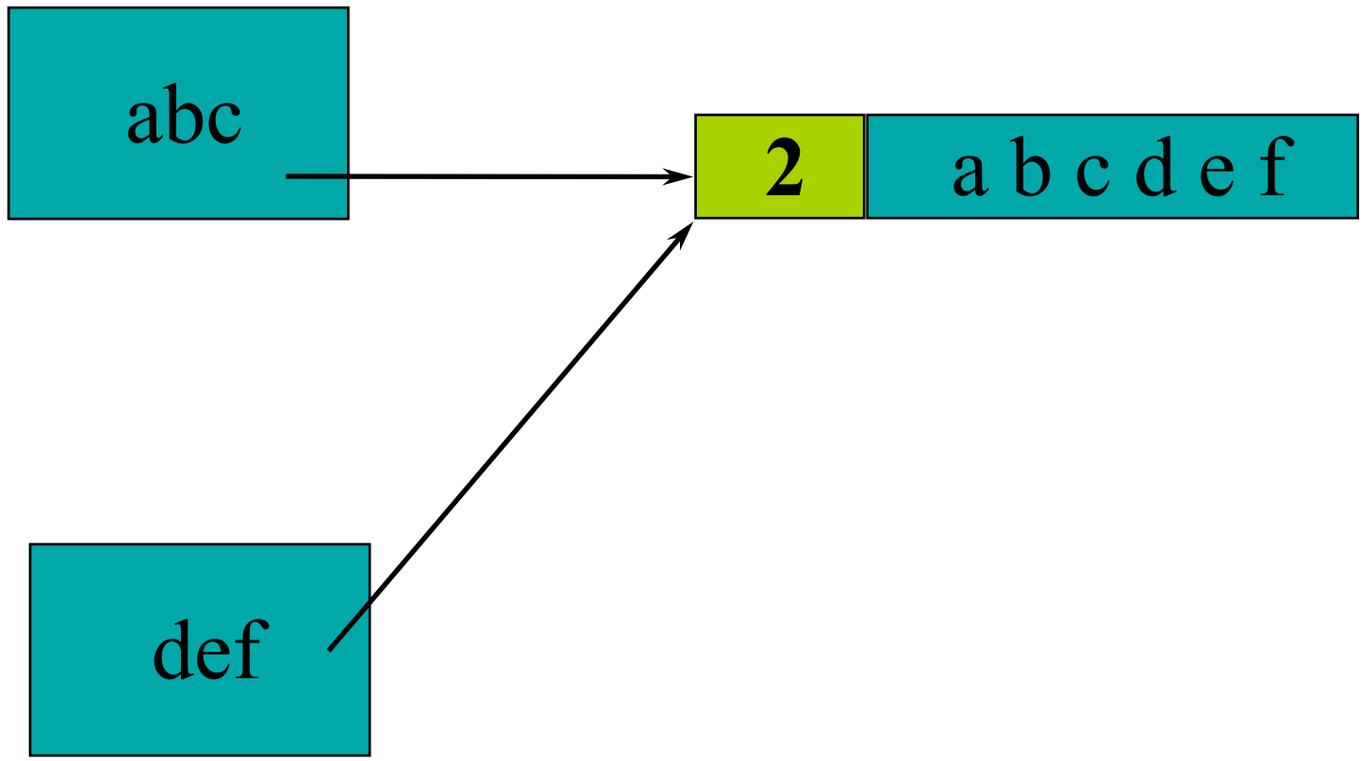
```
String abc("abcdef");
```




Shared memory maintains a count of how many times it is shared

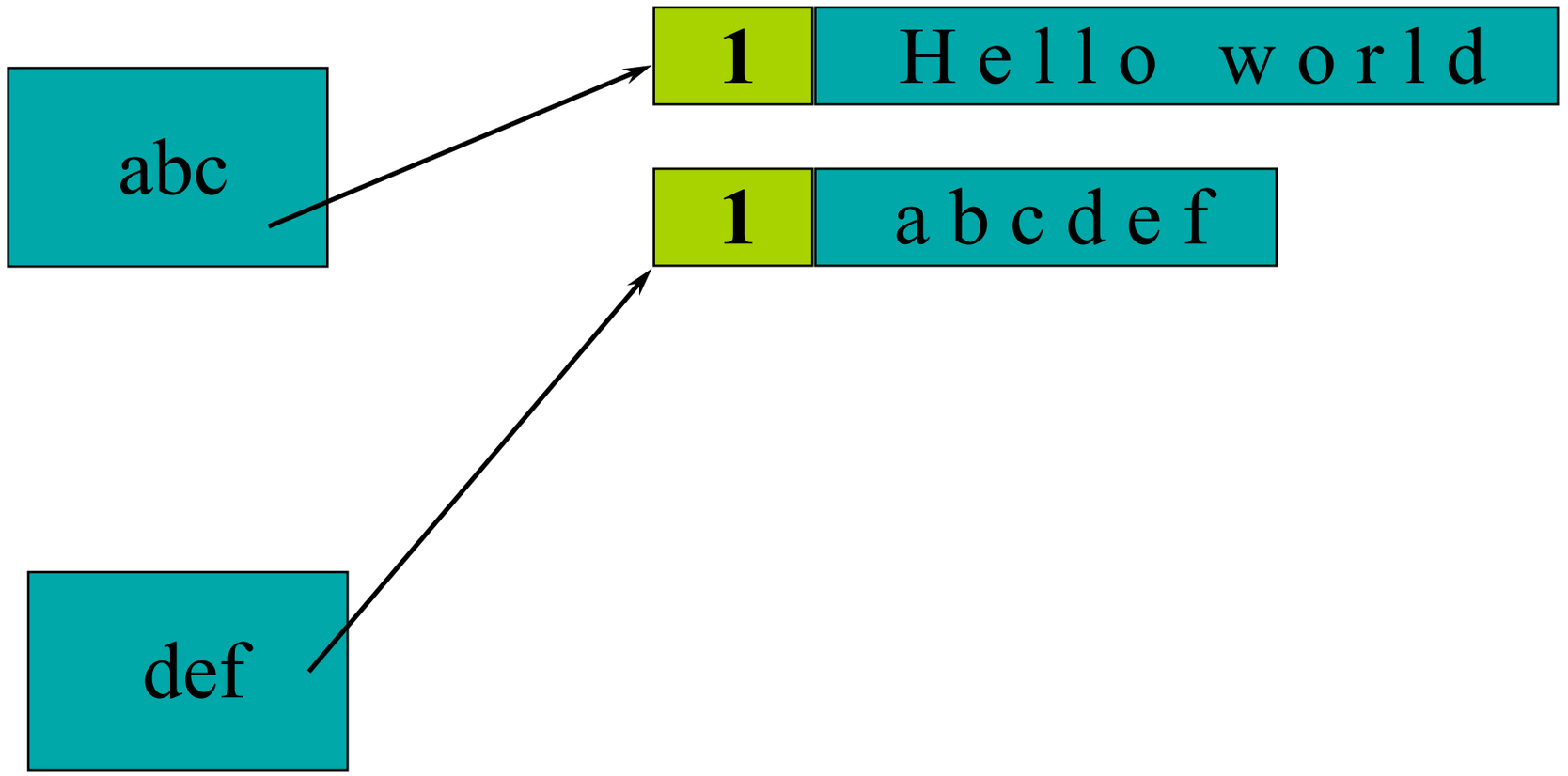
# Reference counts in action

```
String abc("abcdef");
String def = abc; // shallow copy of abc
```



# Reference counts in action

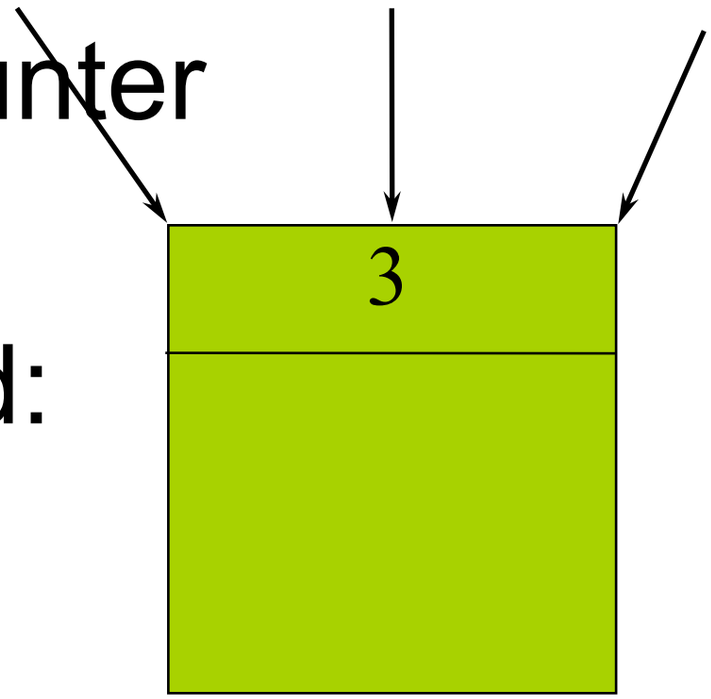
```
String abc("abcdef");
String def = abc;          // shallow copy of abc
abc = "Hello world";       // copy on write
```



# Reference counting

- Each sharable object has a counter
- Initial value is 0
- Whenever a pointer is assigned:

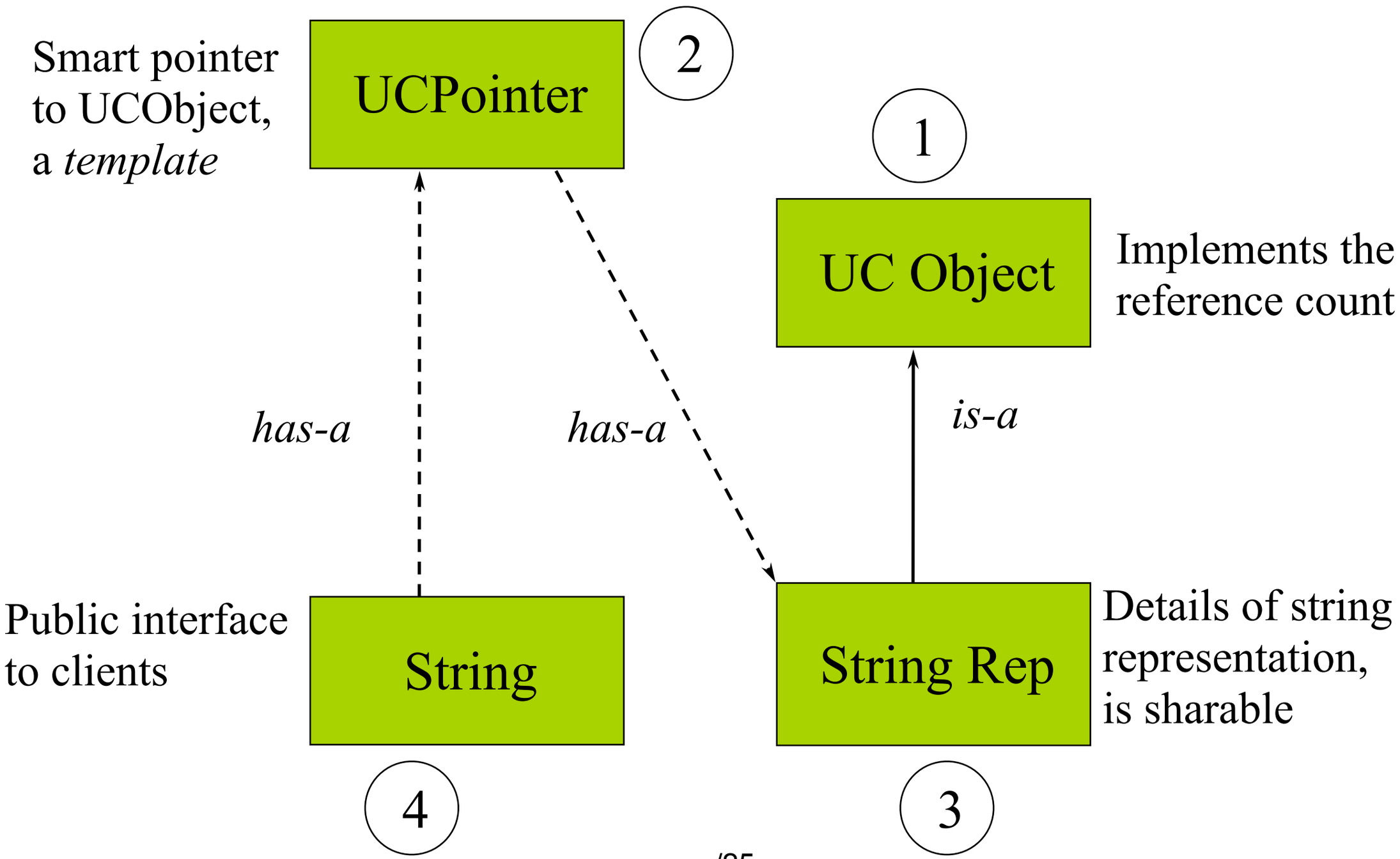
```
p = q;
```




- Have to do the following

```
p->decrement(); // p's count will decrease
p = q;
q->increment(); // q/p's count will increase
```

# The four classes involved



# Reusing reference counting

```
#include <assert.h>
class UCObject {
public:
   UCObject() : m_refCount(0) { }
   virtual ~UCObject() { assert(m_refCount == 0);};
  UCObject(const UCObject&) : m_refCount(0) { }
   void incr() { m_refCount++; }
   void decr();
   int references() { return m_refCount; }
private:
   int m_refCount;
};
```

# UCObject continued

```
inline void UCObject::decr() {
   m_refCount -= 1;
    if (m_refCount == 0) {
      delete this;
    }
}
```

- "Delete this" is legal
  - –But don't use *this* afterwards!

# Class UCPointer

```
template <class T>
class UCPointer {
private:
   T* m_pObj;
   void increment() { if (m_pObj) m_pObj->incr(); }
   void decrement() { if (m_pObj) m_pObj->decr
   (); }
public:
   UCPointer(T* r = 0): m_pObj(r) { increment();}
   ~UCPointer() { decrement(); };
   UCPointer(const UCPointer<T> & p);
   UCPointer& operator=(const UCPointer<T> &);
   T* operator->() const;
   T& operator*() const { return *m_pObj; };
};
```

# UCPointer copy constructor

```
template <class T>
UCPointer<T>::UCPointer(const UCPointer<T> & p){
    m_pObj = p.m_pObj;
    increment();
}
```

# UCPointer assignment

```
template <class T>
UCPointer<T>&
UCPointer<T>::operator=(const UCPointer<T>& p){
   if (m_pObj != p.m_pObj) {
      decrement();
      m_pObj = p.m_pObj;
      increment();
    }
   return *this;
}
```

# The -> Operator

- `operator->`() is a unary operator
  - Result must support the -> operation
- C++ allows you to overload
  - [] -- subscripting
  - () -- "function call"
  - ->() -- pointer chasing
  - *() -- unary pointer dereference

# The UCPointer -> operator

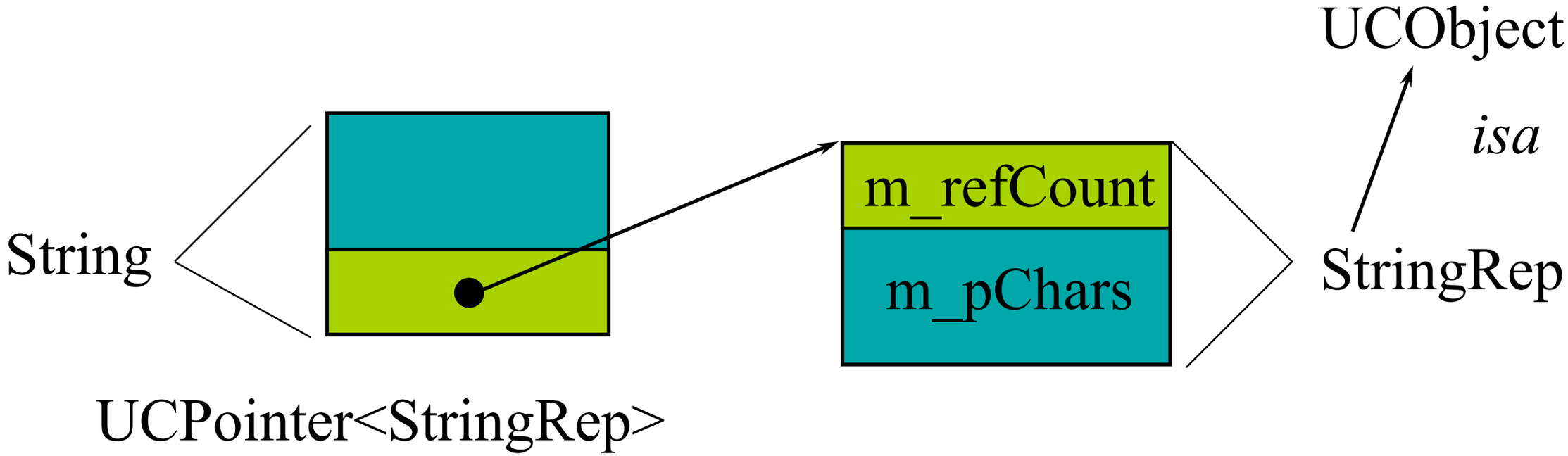
```
template<class T>
T* UCPointer<T>::operator->() const {
   return m_pObj;
}
```

- Example: Shape inherits from UCObject.

```
Ellipse elly(200F, 300F);
UCPointer<Shape*> p(&elly);
p->render(); // calls Ellipse::render() on
elly!
```

# Envelope and Letter

- Envelope provides protection
- Letter contains the contents



# String Class

```
class String {
public:
   String(const char *);
   ~String();
   String(const String&);
   String& operator=(const String&);
   int operator==(const String&) const;
   String operator+(const String&) const;
   int length() const;
   operator const char*() const;
private:
   UCPointer<StringRep> m_rep;
};
```

# Class StringRep

```
class StringRep : public UCObject {
public:
     StringRep(const char *);
     ~StringRep();
     StringRep(const StringRep&);
     int length() const{ return strlen(m_pChars); }
     int equal(const StringRep&) const;
private:
     char *m_pChars;
    // reference semantics -- no assignment op!
    void operator=(const StringRep&) { }
};
```

# StringRep implementation

```
StringRep::StringRep(const char *s) {
  if (s) {
       int len = strlen(s) + 1;
    m_pChars = new char[len];
    strcpy(m_pChars , s);
  } else {
       m_pChars = new char[1];
    *m_pChars = '\0';
  }
}
StringRep::~StringRep() {
  delete [] m_pChars ;
}
```

# StringRep implementation

```
StringRep::StringRep(const StringRep& sr) {
  int len = sr.length();
  m_pChars = new char[len + 1];
  strcpy(m_pChars , sr.m_pChars );
}

int StringRep::equal(const StringRep& sp)
  const {
  return (strcmp(m_pChars, sp.m_pChars) ==
  0);
}
```

# String implementation

```
String::String(const char *s) : m_rep(0) {
  m_rep = new StringRep(s);
}

String::~String() {}

// Again, note constructor for rep in list.
String::String(const String& s) : m_rep(s.m_rep)
   { }

String&
String::operator=(const String& s) {
  m_rep = s.m_rep; // let smart pointer do work!
  return *this;
}
```

# String implementation

```
int
String::operator==(const String& s) const {
  // overloaded -> forwards to StringRep
  return m_rep->equal(*s.m_rep); // smart
  ptr *
}

int
String::length() const {
  return m_rep->length();
}
```

# Critique

- UCPointer maintains reference counts
- UCObject hides the details of the count String is very clean
- StringRep deals only with string storage and manipulation
- UCObject and UCPointer are reusable
- Objects with cycles of UCPointer will never be deleted

# Other smart pointers

- Standard library holder for raw pointers on stack
- Releases resource when destroyed (latest)

```
template <class X> std::auto_ptr {
public:
    explicit auto_ptr(X* = 0) throw();
    auto_ptr(auto_ptr&) throw();
    auto_ptr& operator=(auto_ptr&) throw();
    ~auto_ptr();
    X& operator*() const throw();
    X* operator->() const throw();
    ...
    };
```