Object－Oriented Programming

Week 13, Fall 2018

Streams

Weng Kai

Why streams?

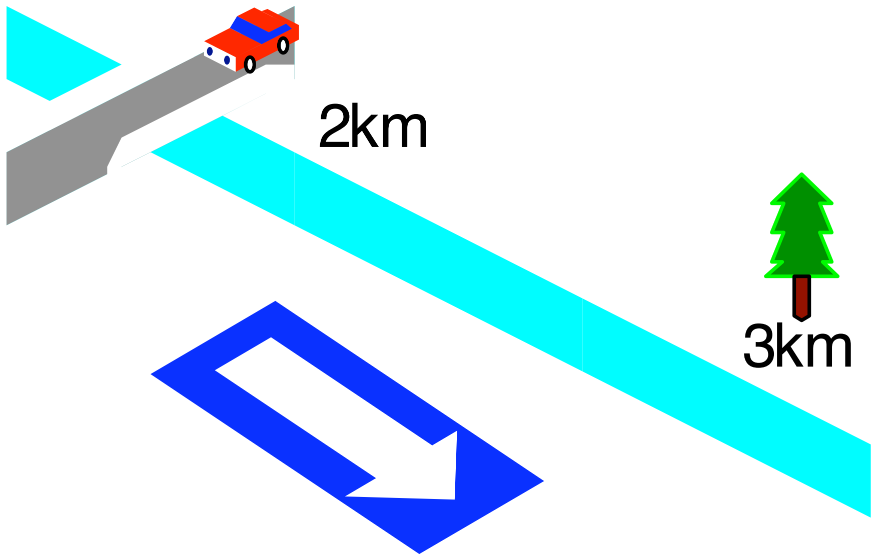
- Original C I/O used printf, scanf
- Streams invented for C++
  - C I/O libraries still work
- Advantages of streams
  - Better type safety
  - Extensible
  - More object-oriented
- Disadvantages
  - More verbose
  - Often slower

C vs. C++

- C stdio operations work
  - Don't provide “object-oriented” features
  - No overloadable operators
- C++
  - Can overload inserters and extractors
- Moral
  - When converting C to C++, leave the I/O intact

What is a stream?

- Common logical interface to a device
- Sequential
  - There is a "position" associated with each stream
- Can
  - Produce values
  - Consume values
  - Both



Stream naming conventions

	Input	**Output**	**Header**
Generic	istream	ostream	<iostream>
File	ifstream	ofstream	<fstream>
C string (legacy)	istrstream	ostrstream	<strstream>
C string	istringstream	ostringstream	<sstream>

Stream operations

- Extractors
  - –Read a value from the stream
  - –Overload the >> operator
- Inserters
  - –Insert a value into a stream
  - –Overload the << operator
- Manipulators
  - –Change the stream state
- Others

Kinds of streams

- Text streams
  - Deal in ASCII text
  - Perform some character translation
    - e.g.: newline -> actual OS file representation
  - Include
    - Files
    - Character buffers
- Binary streams
  - Binary data
  - No translations

Predefined streams

- cin
  - standard input
- cout
  - standard output
- cerr
  - unbuffered error (debugging) output
- clog
  - buffered error (debugging) output

Example

```
#include <iostream>
int i; float f; char c;
char buffer[80];
```

- Read the next character

```
  cin >> c;
```

- Read an integer

```
  cin >> i; // skips whitespace
```

- Read a float and a string separated by whitespace

```
  cin >> f >> buffer;
```

Predefined extractors

- *`istream >> lvalue`*

expression type	**output format**	**C I/O**
char	character	%c
short, int	integer	%d
long	long decimal integer	%ld
float	floating point	%g
double	double precision floating pt.	%lg
long double	long double	%Lg
char *	string	%s
void *	pointer	%p

- Extractors skip leading whitespace, in general

Defining a stream extractor

- Has to be a 2-argument free function
  - First argument is an `istream&`
  - Second argument is a *reference* to a value

```
istream&
operator>>(istream& is, T& obj) {
    // specific code to read obj
   return is;
}
```

- Return an `istream&` for chaining

```
cin >> a >> b >> c;
((cin >> a) >> b) >> c;
```

Other input operators

- `int get()`
  - Returns the next character in the stream
  - Returns EOF if no characters left
  - Example: copy input to output

```
int ch;
while ((ch = cin.get()) != EOF)
  cout.put(ch);
```

- `istream& get(char& ch)`
  - Puts the next character into argument
  - Similar to int get();

More input operators

- `get(char *buf, int limit, char delim = '\n')`
  - read up to `limit` characters, or to `delim`
  - Appends a null character to `buf`
  - Does not consume the delimiter
- `getline(char *buf, int limit,char delim = '\n')`
  - read up to `limit` characters, or to `delim`
  - Appends a null character to `buf`
  - *Does* consume the delimiter
- `ignore(int limit = 1, int delim = EOF)`
  - Skip over limit characters or to delimiter
  - Skip over delimiter if found

More input operators

- `int gcount()`
  - returns number of characters just read

```
char buffer[100];
cin.getline(buffer, sizeof(buffer));
cout << "read " << cin.gcount()
     << " characters"
```

- `void putback(char)`
  - pushes a single character back into the stream
- `char peek()`
  - examines next character without reading it

```
switch (cin.peek()) ...
```

Predefined inserters

- Usage
  - *ostream* << *expression*

expression type	**output format**	**C I/O**
char	character	%c
short, int	integer	%d
long	long decimal integer	%ld
float, double	double precision floating pt.	%g
long double	long double	%lg
char *	string	%s
void *	pointer	%p

Creating a stream inserter

- Has to be a 2-argument free function
  - First argument is an ostream&
  - Second argument is any value

```
ostream&
operator<<(ostream& os, const T& obj) {
    // specific code to write obj
   return os;
}
```

- Return an `ostream&` for chaining

```
cout << a << b << c;
((cout << a) << b) << c;
```

Other output operators

- `put(char)`
  - prints a single character
  - Examples

    ```
    cout.put('a');
    cerr.put('!');
    ```
- `flush()`
  - Force output of stream contents
  - Example

    ```
    cout << "Enter a number";
    cout.flush();
    ```

Formatting using manipulators

- Manipulators modify the state of the stream
    - #include <iomanip>
    - Effects hold (usually)
- Example

```
int n;
cout << "enter number in hexadecimal"
     << flush;
cin >> hex >> n;
```

Example

- A simple program

```
#include <iostream>
#include <iomanip>
main() {
   cout << setprecision(2) << 1000.243 <<endl;
   cout << setw(20) << "OK!";
   return 0;
}
```

- Prints

```
1e03
                  OK!
```

Manipulators

manipulator	**effect**	**type**
dec, hex, oct	set numeric conversion	I, O
endl	insert newline and flush	O
flush	flush stream	O
setw(int)	set field width	I, O
setfill(ch)	change fill character	I, O
setbase(int)	set number base	O
ws	skip whitespace	I
setprecision(int)	set floating point precision	O
setiosflags(long)	turn on specified flags	I, O
resetiosflags(long)	turn off specified flags	I, O

Creating manipulators

- You can define your own manipulators!

```
// skeleton for an output stream manipulator
ostream& manip(ostream& out) {
   ...
   return out;
}
ostream& tab ( ostream& out ) {
   return out << '\t';
}
cout << "Hello" << tab << "World!" << endl;
```

Stream flags control formatting

flag	**purpose (when set)**
ios::skipws	skip leading white space
ios::left, ios::right	justification
ios::internal	pad between sign and value
ios::dec, ios::oct, ios::hex	format for numbers
ios::showbase	show base of number
ios::showpoint	always show decimal point
ios::uppercase	put base in uppercase
ios::showpos	display + on positive numbers
ios::scientific, ios::fixed	floating point format
ios::unitbuf	flush on every write

Setting flags

- Using manipulators
  - –setiosflags(flags);
  - –resetiosflags(flags);
- Using stream member functions
  - –setf(flags)
  - –unsetf(flags)

Working with flags

- Code

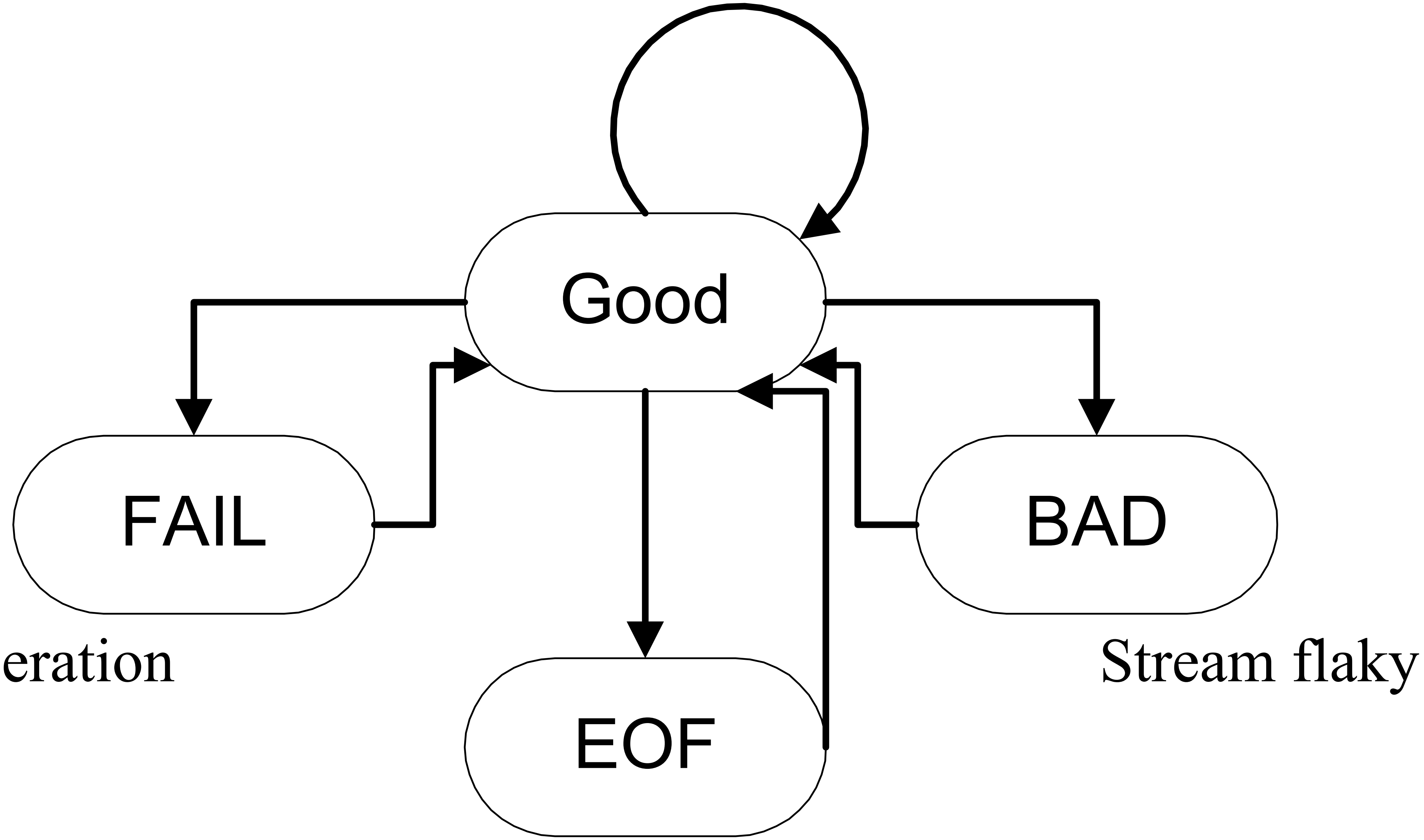
```
#include <iostream>
#include <iomanip>
main() {
   cout. setf(ios::showpos | ios::scientific);
   cout << 123 << " " << 456.78 << endl;
   cout << resetiosflags(ios::showpos) << 123;
   return 0;
}
```

- Prints

```
   +123 +4.567800e+02
   123
```

Stream error states

clear() returns stream to GOOD



Working with streams

- Error state is set after each operation
- Conversion to void* returns 0 if problem
- Can clear an error state using
  - `clear() //` Resets error state to good()
- Checking status
  - `good() //` Returns true if in valid state
  - `eof()  //` Returns true if at EOF
  - `fail() //` Returns true if minor failure or bad
  - `bad()  //` Returns true if in bad state

Example

```
int n;
cout << "Enter a value for n, then [Enter]" << flush;
while (cin.good()) {
  cin >> n;
  if (cin) { // input was ok
      cin.ignore(INT_MAX, '\n'); // flush newline
       break;
   }
   if (cin.fail()) {
      cin.clear();  // clear the error state
      cin.ignore(INT_MAX, '\n'); // skip garbage
      cout << "No good, try again!" << flush;
   }
}
```

File streams

- ifstream, ofstream connect files to streams
  - #include <fstream>
  - Open modes specify how to create files

mode	**purpose**
ios::app	append
ios::ate	position at end of file
ios::binary	do binary I/O
ios::in	open for input
ios::out	open for output
ios::nocreate	don't create file if not there
ios::noreplace	don't replace file if present
ios::trunc	truncate file if present

File streams

```
#include <iostream>
#include <fstream>
int main(int argc, char *argv[]) {
   if (argc != 3) {
     cerr << "Usage: copy file1 file2" << endl;
     exit(1);
   }
   ifstream in(argv[1]);

   if (!in) {
      cerr << "Unable to open file " << argv[1];
      exit(2);
   }
```

File streams

```
   ofstream  out(argv[2]);
   if (!out) {
      cerr << "Unable to open file " << argv[2];
      exit(2);
   }
   char c;
   while (in >> c) {
      out << c;
   }
}
```

More stream operations

- `open(const char *, int flags, int)`
  - Open a specified file

```
ifstream inputS;
inputS.open("somefile", ios::in);
if (!inputS) {
     cerr << "Unable to open somefile";
...
```

- `close()`
  - Closes stream

IO stream buffers

- Every IO stream has a stream buffer
- Class `streambuf` defines the buffer abstraction
- The member function `rdbuf()` returns a pointer to the stream buffer
- The << operation is overloaded for `streambufs`
  - It connects buffers directly!

Copy a file to standard out

```
#include <fstream>
#include <assert>

main(int argc, char *argv[]) {
  assert(argc == 2);
  ifstream in(argv[1]);
  assert(in);  // check that stream opened
  cout << in.rdbuf(); // Drain file!
}
```

String streams (legacy)

- I/O to character buffers is modeled using streams
  - `#include <strstream.h>`
  - Input: `istrstream` class
  - Output: `ostrstream` class

```
istrstream in("2.3 47 This is a stream");
int i; float f; char buf[123];
in >> f >> i >> buf;
cout << " i = " << i;
cout << " f = " << f;
cout << " buf = " << buf << endl;
cout << in.rdbuf(); // print remainder!
```

ostrstreams and storage allocation

- Input streams are initialized with a buffer

```
istrstream mystr("hi bob");
```

- Output streams have two allocation methods
  - User allocates storage

```
char buffer[SIZE];
ostrstream(buffer, SIZE , ios::out);
```

  - Stream handles storage

```
ostrstream A;
A << cin.rdbuf(); // read file into string!
```

    - You can get the buffer, but programming gets messy
      - `char *str()` returns the buffer...

Notes

- use string and stringstream (not strstream)
  - example
- You can create your own manipulators

```
// newline without a flush
ostream & nl ( ostream& os ) {
   return os << '\n';
}

cout << "newline" << nl;
```

C vs. C++

- C stdio operations work
  - –Don't provide “object-oriented” features
  - –No overloadable operators
- C++
  - –Can overload inserters and extractors
- Moral
  - –When converting C to C++, leave the I/O intact

Putting it All Together

Templates

Inheritance

Reference Counting

Smart Pointers

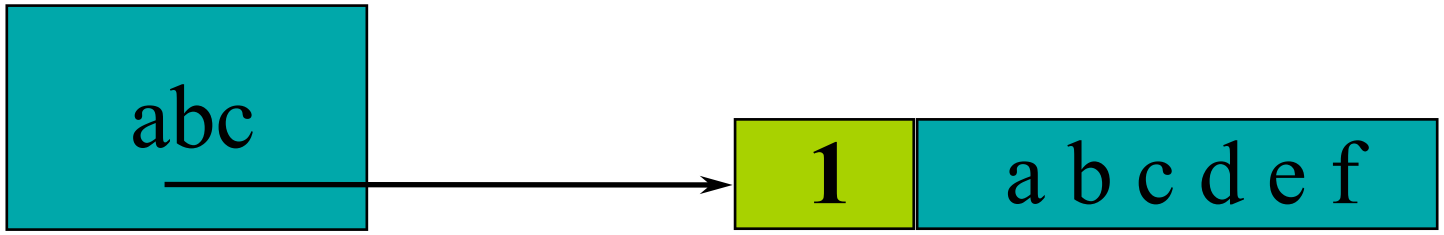
Reference: *C++ Strategies and Tactics*, Robert Murray, 1993

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

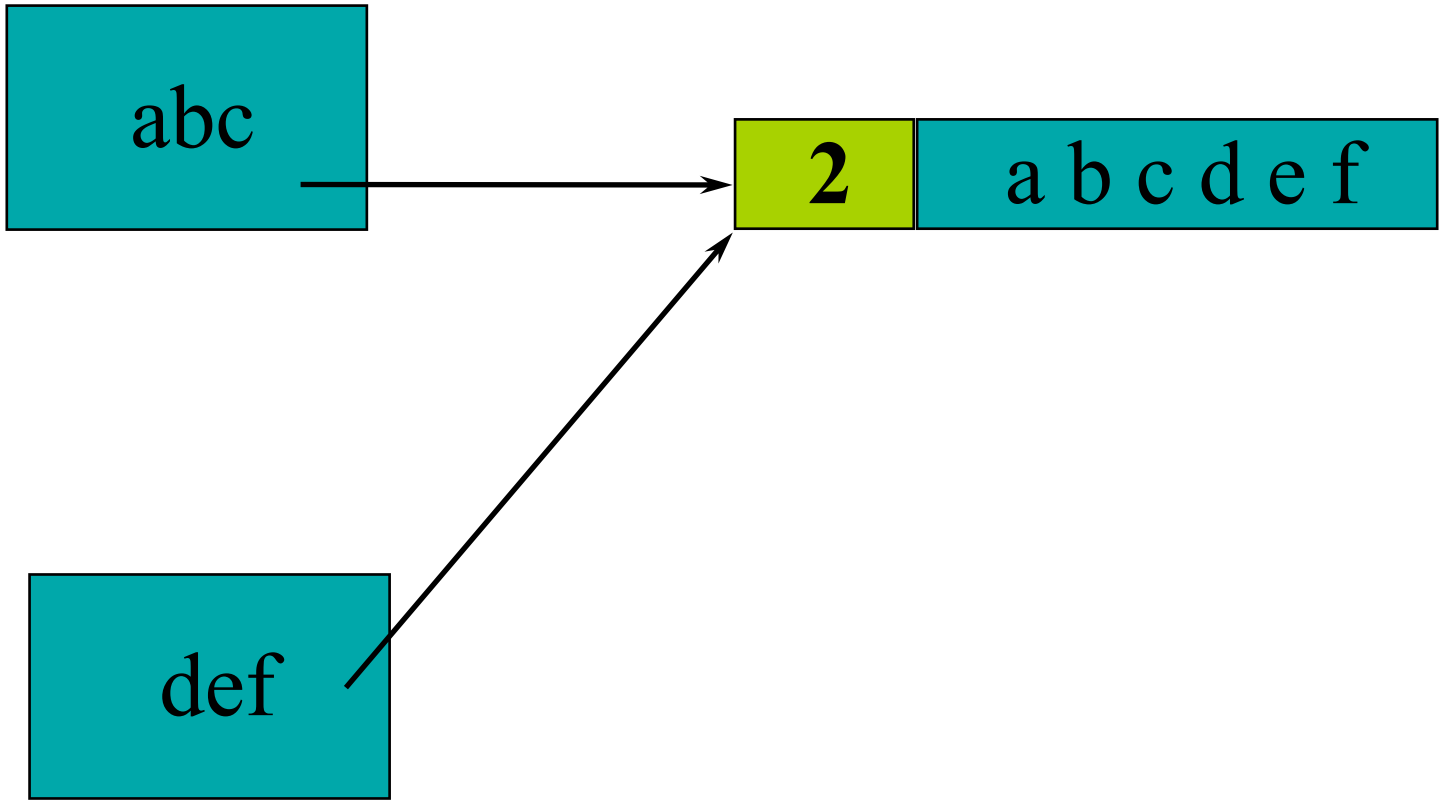
```
String abc("abcdef");
```



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

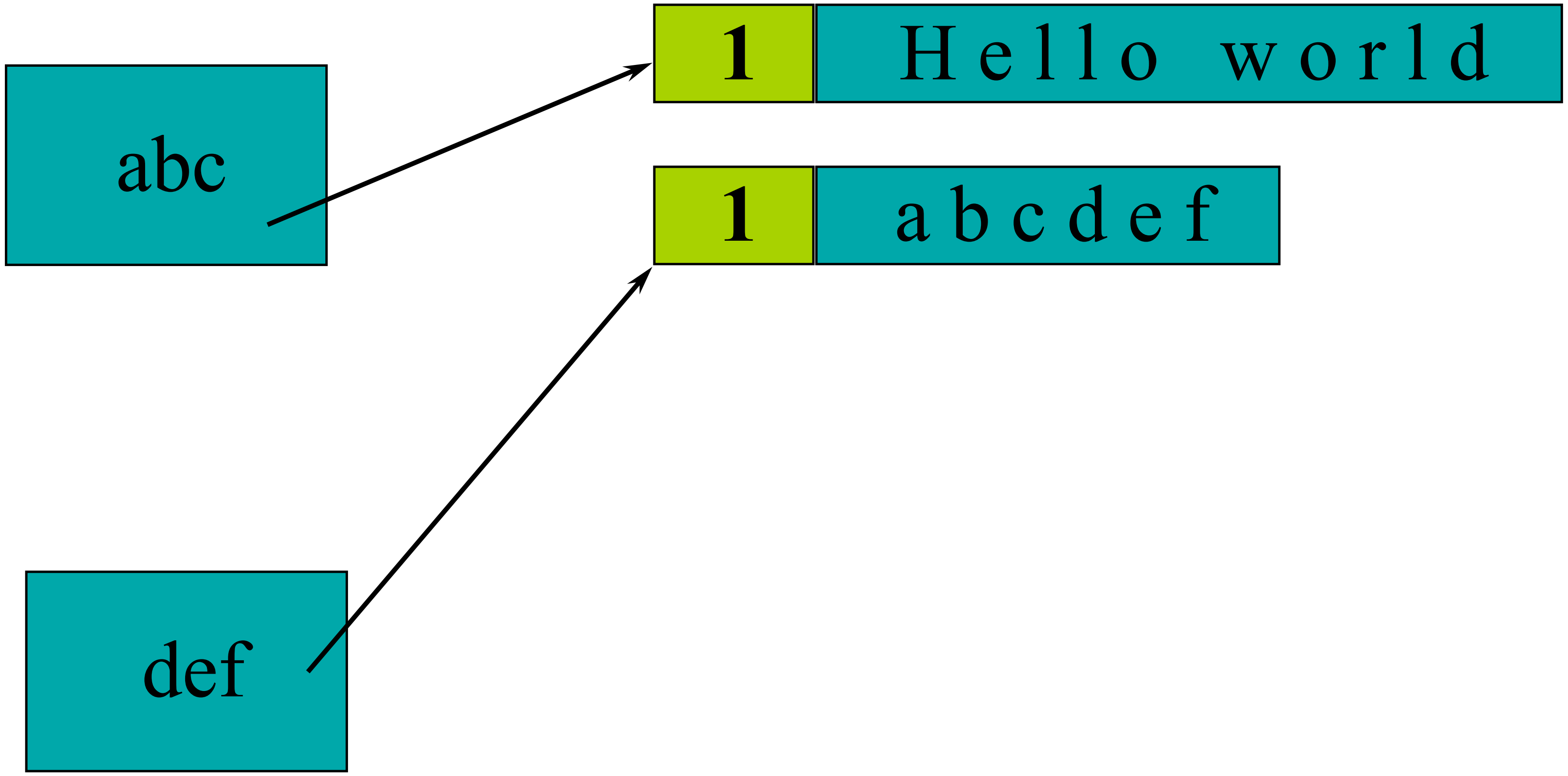
Reference counts in action

```
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
```



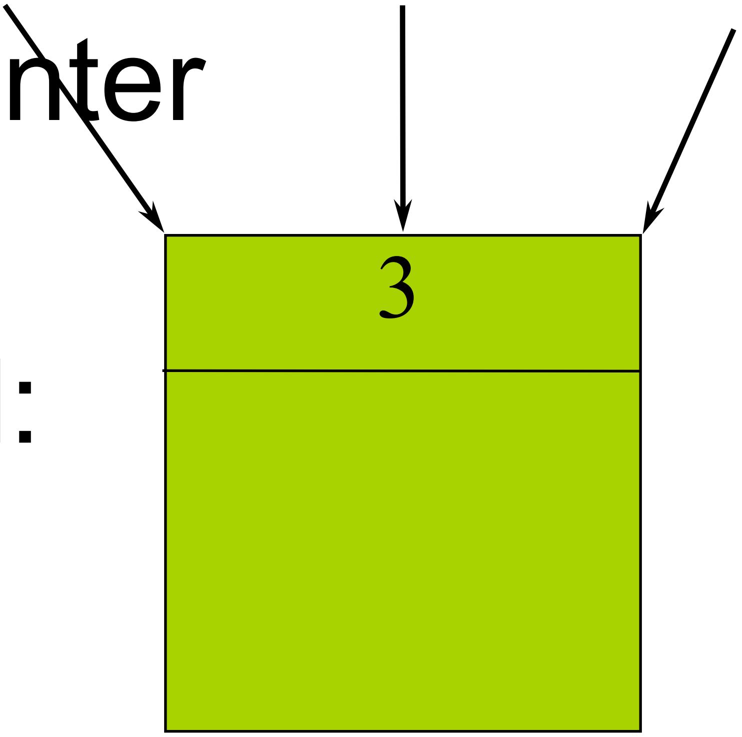
Reference counting

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

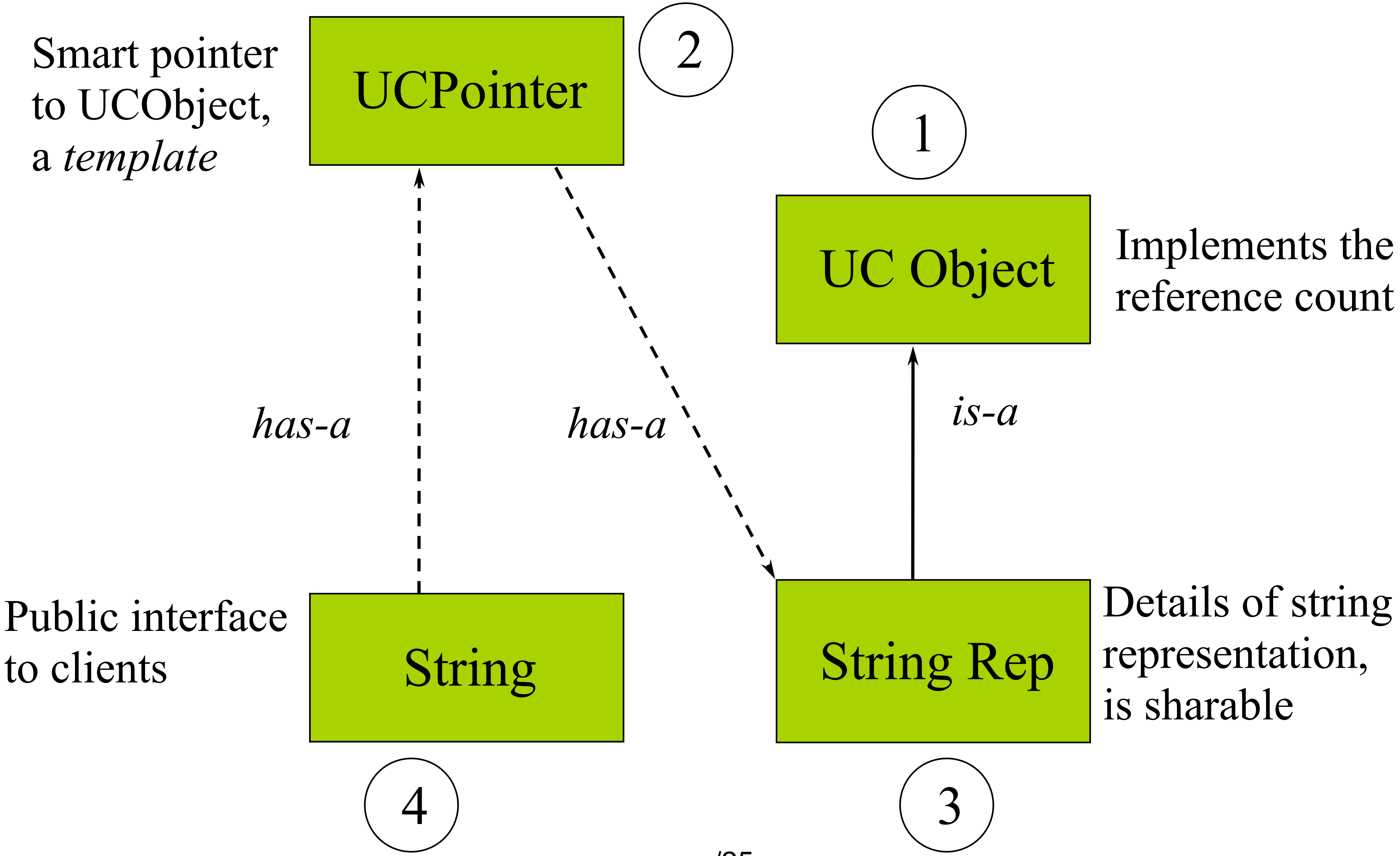
```
p = q;
```

- 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



Smart pointer to UCObject, a *template*

UCPointer

2

1

UC Object

Implements the reference count

has-a

has-a

is-a

Public interface to clients

String

String Rep

Details of string representation, is sharable

4

3

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

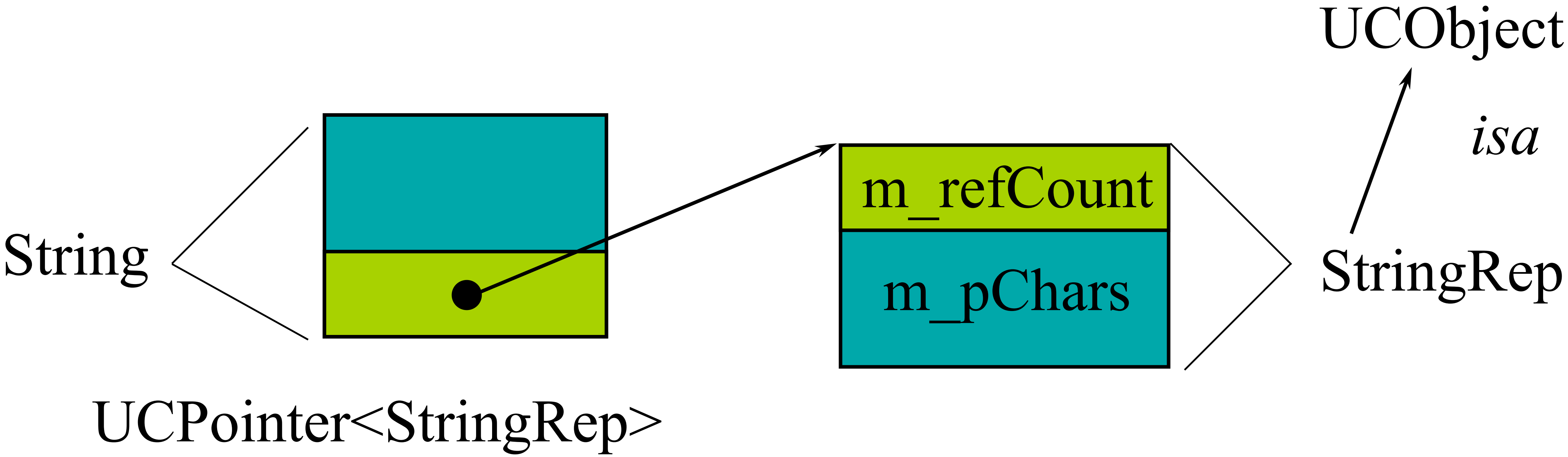
```
template<class T>
T* UCPointer<T>::operator->() const {
   return m_pObj;
}
```

- 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();
   ...
   };
```