**Smart Pointer:**

*Smart pointers* are objects looks like built-in pointers (dumb pointer), but to offer greater functionality. They have a variety of applications, including resource management and the automation of repetitive coding tasks.

When we replace the *dumb* pointers with smart pointer then, gain control over the following aspects of pointer behaviours:

**Construction and destruction.** We can determine, what happens when a smart pointer is created and destroyed. Assign default value to the pointer (get rid from uninitialized pointers), automatically destroyed when goes out of scope (or ref count decrease when use shared ptr and destroyed when ref cnt=0) to eliminating resource leaks, lazy initialization (not initialized when created, until generate request for use) and many more.

**Copying and assignment.** We can decide the action, when a smart pointer is copied or is involved in an assignment like shallow copy or deep copy.

**Dereferencing.** What should happen when a client refers to the object pointed to by a smart pointer? For example, use smart pointers to help implement the lazy fetching strategy.

Most smart pointer templates look something like this:

template<class T> // template for smart pointer objects

class SmartPtr {

public:

SmartPtr(T\* realPtr = 0); // create a smart ptr to an obj given

// a dumb ptr to it; uninitialized

// ptrs default to 0 (null)

SmartPtr(const SmartPtr& rhs);// copy a smart ptr

~SmartPtr(); // destroy a smart ptr

// make an assignment to a smart ptr

SmartPtr& operator=(const SmartPtr& rhs);

T\* operator->() const; // dereference a smart ptr to get at

// a member of what it points to

T& operator\*() const; // dereference a smart ptr

private:

T \*pointee; // what the smart ptr points to

};

For smart pointer classes where copying and assignment are not allowed, both (copy constructor and assignment operator) would typically be declared as private.

The two dereferencing operators are declared const, because dereferencing a pointer doesn't modify it (though it may lead to modification of what the pointer points to).

Finally, each smart pointer containing a dumb pointer-to-T within it. It is this dumb pointer that does the actual pointing.

**Construction, Assignment and Destruction of smart pointer.**

Locate an object to point to (typically by using the smart pointer's constructor arguments), then make the smart pointer's internal dumb pointer point there. If no object can be located, set the internal pointer to 0 or signal an error (possibly by throwing an exception).

SmartPtr(T \*realPtr = 0) :pointee(realPtr) {}

How to implement the smart pointer's copy constructor, assignment operator(s) and

destructor.

If a smart pointer *owns* the object it points to, it is responsible for deleting that object when it (the smart pointer) is destroyed. The object pointed to by the smart pointer is dynamically allocated.

Consider the auto\_ptr template from the standard C++ library. An auto\_ptr object is a smart pointer that points to a heap-based object until it (the auto\_ptr) is destroyed. The auto\_ptr template might be implemented like:

template<class T>

class auto\_ptr {

public:

auto\_ptr(T \*ptr = 0): pointee(ptr) {}

~auto\_ptr() { delete pointee; }

auto\_ptr(auto\_ptr<T>& rhs); //copy constructor, non-const param

auto\_ptr<T>& operator=(auto\_ptr<T>& rhs); // assignment operator

// Also having non-const param.

//*Note:Usually copy ctor & assigenment operator expect const params.*

...

private:

T \*pointee;

};

But what should happen when an auto\_ptr is copied or assigned?

auto\_ptr<TreeNode> ptn1(new TreeNode);

auto\_ptr<TreeNode> ptn2 = ptn1; // call to copy ctor then, what should happen?

auto\_ptr<TreeNode> ptn3;

ptn3 = ptn2; // call to operator= then, what should happen?

Now we can do two things:

1. We just copied the internal dumb pointer and end up with two auto\_ptrs pointing to the same object (Problem: If one pointer deleted it will easily invalidate others).
2. An alternative would be to create a new copy of what was pointed to by calling new. That would guarantee we didn't have too many auto\_ptrs pointing to a single object

(Problem: Performance bottleneck, too many new object).

The problems would vanish if auto\_ptr prohibited copying and assignment, but a more flexible solution was adopted for the auto\_ptr classes: object ownership is *transferred* when an auto\_ptr is copied or assigned:

//Copy Constructor implementation

template<class T>

auto\_ptr<T>::auto\_ptr(auto\_ptr<T>& rhs)

{

pointee = rhs.pointee; // transfer ownership of \*pointee to \*this

rhs.pointee = 0; // rhs no longer owns anything

//If rhs parameter deleacred as const then rhs.pointee = 0 will //become invalid.(no ownership will transfer)

}

//Assignment operator Implementation.

template<class T>

auto\_ptr<T>& auto\_ptr<T>::operator=(auto\_ptr<T>& rhs)

{

if (this == &rhs) // do nothing if this

return \*this; // object is being assigned to itself

delete pointee; // delete currently owned object

pointee = rhs.pointee; // transfer ownership of

rhs.pointee = 0; // \*pointee from rhs to \*this

return \*this;

//If rhs parameter deleacred as const then rhs.pointee = 0 will

//become invalid. (no ownership will transfer)

}

Now, because of object ownership is transferred when auto\_ptr's copy constructor is called, passing auto\_ptrs by value is often a *very* bad idea. Here's why:

// This function will often lead to disaster.

void printTreeNode(ostream& s, auto\_ptr<TreeNode> p){

s << \*p;

}

int main()

{

auto\_ptr<TreeNode> ptn(new TreeNode);

...

printTreeNode(cout, ptn); // pass auto\_ptr by value

... // ownership transfer to p

} // now ptn is null.

When printTreeNode's parameter p is initialized (by calling auto\_ptr's copy constructor), ownership of the object pointed to by ptn is transferred to p. When printTreeNode finishes executing, p goes out of scope and its destructor deletes what it points to. Now ptn, no longer points to anything (pointing to null).

Instead of that, if we write a method like this: (Pass-by-reference-to-const)

void printTreeNode(ostream& s, const auto\_ptr<TreeNode> &p){

//No copy ctor call, for p.

s << \*p; //Since no new obj created, so no ownership transfer

}

***Note****: copy ctor & assignment operator functions normally take const parameter. But in*

*Case of auto\_ptr there is changes in parameters during the copy or the assignment. In other words, auto\_ptr objects are modified if they are copied or are the source of an assignment. So, it cannot be const. Let suppose, if we necessarily declared parameter as const, C++ offers const\_cast to remove const-ness from object.*

A smart pointer's destructor often looks like this:

template<class T>

SmartPtr<T>::~SmartPtr()

{

if (*\*this owns \*pointee*) {

delete pointee;

}

}

There is no need to check if condition for auto\_ptr, because auto\_ptr always owns what it points to. But for another smart pointer (like shared\_ptr) need to check total ref cnt before delete the pointee.

**Implementing the Dereferencing Operators:**

template<class T>

T& SmartPtr<T>::operator\*() const{

*perform "smart pointer" processing;*

return \*pointee;

}

**Similarly,**

T\* SmartPtr<T>::operator->() const{

*perform "smart pointer" processing;*

return pointee;

}

First the function does whatever processing is needed to initialize. For example, if lazy fetching is being used, the function performs required action to make Pointee valid. Once pointee is valid, the operator\* or-> function just returns a reference or pointer respectively to the pointed-to object.

***Note:*** *De-reference function return type is either pointer or reference not an object. pointee need not point to an object of type T; it may point to an object of a class derived from T. If that is the case and dereferencing function returns a T object instead of a pointer or reference to the actual derived class object, our function will return an object of the wrong type! (slicing problem).*

**Testing Smart Pointers for Null-ness:**

SmartPtr<TreeNode> ptn;

...

We test Null like below:

if (ptn == 0) ... // error!

if (ptn) ... // error!

if (!ptn) ... // error!

Error, because of required function is absent form SmartPtr class. One way is adding isNull member function to our smart pointer classes, to test the nullability. But we want to keep SmartPtr behavior like dumb pointer. How we can do that?

We can write the implicit conversion operator method which convert pointee type to void\*.

template<class T>

class SmartPtr {

public:

...

operator void\*(){ ,

if(pointee==NULL)

return NULL; // returns 0 if the smart ptr is null

else

return (void\*)1; // otherwise return nonzero

};

In below case every time it calls conversion function and return zero if null otherwise return non zero value.

if (ptn == 0) ... // fine

if (ptn) ... // fine

if (!ptn) ... // fine

But this approach has major disadvantage, Lets suppose we have a two smart pointer of different type:

SmartPtr<Apple> pa;

SmartPtr<Orange> po;

...

if (pa == po) ... // Even we don’t have operator == inside

// SmartPtr, this compile!

Why? Because both smart pointers can be implicitly converted into void\* pointers, and there is a built-in comparison function (compiler generated) which makes comparison.

We may think to overload the operator!, But problem will still remain same.

template<class T>

class SmartPtr {

public:

...

bool operator!() const; // returns true if and only

... // if the smart ptr is null

};

SmartPtr<TreeNode> ptn;

...

if (!ptn) { // fine

... // ptn is null

}

else {

... // ptn is not null

}

but below checks are failed:

if (ptn == 0) ... // still an error

if (ptn) ... // also an error

The only risk for mixed-type comparisons is statements such as these:

SmartPtr<Apple> pa;

SmartPtr<Orange> po;

...

if (!pa == !po) ... // this compiles and gives un-desirable result

We will discuss later, how to implement the null check for SmartPtr class.

**Converting Smart Pointers to Dumb Pointers:**

Consider the below class:

class Person {

int age;

char\* pName;

char\* initName(char \*nm) {

int size = strlen(nm);

char \*tmp = new char[size];

memcpy(tmp, nm, strlen(nm));

tmp[size] = '\0';

return tmp;

}

public:

Person() : pName(0), age(0) {}

Person(char\* nm, int age) : pName(initName(nm)), age(age){ }

~Person() {

delete pName;

age = 0;

}

void Display()const {

printf("Name = %s Age = %d \n", pName, age);

}

void Shout()const{

printf("Ooooooooooooooooo\n");

}

};

Bool validatePerson (const Pesron \*p) {

//Validate the person base on some logic return true if success otherwise false.

} //or both are same.

Bool validatePerson (const SmartPtr<Person> &p){

}

Now,

SmartPtr<Person> sp1(new Person(“Rajeev Sharma”,32));

validatePerson(sp1); //Error, cannot convert SmartPtr<Person> to Person\*

Yes, we can do it like this: validatePerson(&\*sp1); But it is look like ugly.

The call can be made to succeed by adding to the smart pointer-to-T template an implicit conversion operator to a dumb pointer-to-T:

template<class T> // as before

class SmartPtr {

public:

...

operator T\*() { return pointee; }

...

};

validatePerson(sp1); //Now this call will work. ***And apparently, this function also eliminates the problem of testing for null-ness:***

if (sp1 == 0) ... // fine, converts sp1 to a Person\*

if (sp1) ... // ditto

if (!sp1) ... // ditto (reprise)

However, it also has same some loopback. They make it easy for clients to program directly with dumb pointers bypassing the smart pointer object like:

Void UpdateID(SmartPtr<Person> &person){

Person \*p= person; //Converts SmartPtr<Person> to Person\*

…

//Now client do anything they want using row pointer \*p, bypassing SmartPtr.

//Even if parameter is const, then apply const\_cast to remove the const-ness

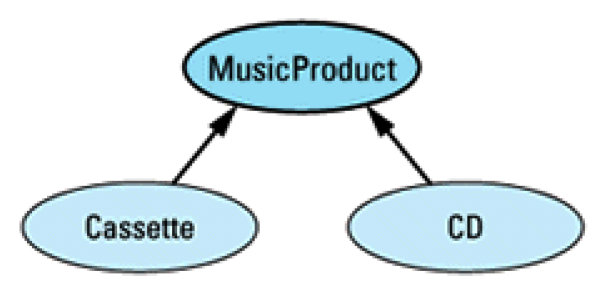
}

Usually, the "smart" behavior provided by a smart pointer is an essential component of our design, so allowing clients to use dumb pointers typically leads to disaster. For example, if SmartPtr implements the reference-counting strategy, allowing clients to manipulate dumb pointers directly will almost certainly lead to bookkeeping errors that corrupt the reference-counting data structures.

***Moral****: Don't provide implicit conversion operators to dumb pointers unless there is a compelling reason to do so. Provide isNull() method to check the nullability* ***(according to my observation).***

**Smart Pointers and Inheritance-Based Type Conversions:**

Suppose we have a public inheritance hierarchy modeling consumer product for storing music:



class **MusicProduct** {

public:

MusicProduct(const string& title);

virtual void play() const = 0;

virtual void displayTitle() const = 0;

...

};

class **Cassette**: public **MusicProduct** {

public:

Cassette(const string& title);

virtual void play() const;

virtual void displayTitle() const;

...

};

class **CD**: public **MusicProduct** {

public:

CD(const string& title);

virtual void play() const;

virtual void displayTitle() const;

...

};

Suppose we have a function that, given a MusicProduct object, displays the title of the product and then plays it:

void displayAndPlay(const MusicProduct\* pmp, int numTimes)

{

for (int i = 1; i <= numTimes; ++i) {

pmp->displayTitle();

pmp->play();

}

}

SmartPtr<Cassette> funMusic(new Cassette("Alapalooza"));

SmartPtr<CD> nightmareMusic(new CD("Disco Hits of the 70s"));

displayAndPlay(funMusic, 10); // error!

displayAndPlay(nightmareMusic, 0); // error!

We can remove above error if, we write a conversion method form smartPtr to dumb pointer.

But we have already seen, writing those conversion in SmartPtrs are not advisable.

look what happens if we replace the dumb pointers with their allegedly smart counterparts:

void displayAndPlay(const SmartPtr<MusicProduct>& pmp, int numTimes);

Still, they won't compile because there is no conversion from a SmartPtr<CD> or a SmartPtr<Cassette> to a SmartPtr<MusicProduct>

We may think, to provide an implicit type conversion operator for smart pointer class to which it should be implicitly convertible. Like:

template<class T> // as before

class SmartPtr {

public:

...

operator SmartPtr<T>() {

return SmartPtr<T>(pointee);

}

...

Private:

T \*pointee;

};

The above conversion method will say, pointee is converted to SmartPtr whatever it has its type. Suppose if pointee type is CD then conversion method converts it into SmartPtr<CD>.

displayAndPlay(funMusic, 10); // error!

displayAndPlay(nightmareMusic, 0); // error!

A reference of type "SmartPtr<MusicProduct> &" cannot be initialized with a value of type "SmartPtr<Cassette >" and “SmartPtr<CD>” respectively. It will Compile if we change the respective function like below:

displayAndPlay(const SmartPtr<Cassette>& pmp,int numTimes);//For Cassette

displayAndPlay(const SmartPtr<CD>& pmp, int numTimes); //For CD.

But this is not what we want. Now we changed our SmartPtr class like below:

template<class T> // template class for smart

class SmartPtr {

public:

SmartPtr(T\* realPtr = 0);

T\* operator->() const;

T& operator\*() const;

...

...

template<class newType> //template function for

operator SmartPtr<newType>() //implicit conversion

{

return SmartPtr<newType>(pointee);

}

Private:

T \*pointee

...

};

Suppose a compiler has a smart pointer-to-T object (Cassette or CD) and need to convert into a smart pointer-to-base-class-of-T(MusicProduct). The compiler checks the class definition for SmartPtr<T> to see, if the requisite conversion operator (below) is declared or not:

operator SmartPtr<MusicProduct>() {

return SmartPtr<MusicProduct>(pointee);

}

Whereas our T \*pointee is either CD or cassette. And obviously this method not found.

Then compiler checks to see if there's a member function template it can instantiate that would let it perform the conversion it's looking for. It finds such a template (the one taking the formal type parameter newType), so it instantiates the template with newType bound

to the base class of T that's the target of the conversion.

At that point, the only question is whether the code for the instantiated member function will compile. For it to compile, it must be legal to pass the (dumb) pointer pointee to the constructor for the smart pointer-to-base-of-T. pointee is of type T, so it is certainly legal

to convert it into a pointer to its (public or protected) base classes. Hence, the code for the type conversion operator will compile, and the implicit conversion from smart pointer-to-T to smart pointer-to-base-of-T will succeed.

**Smart Pointers and const:**

SmartPtr<CD> p; // non-const object, non-const pointer

SmartPtr<const CD> p; // const object, non-const pointer

const SmartPtr<CD> p = &goodCD; // non-const object, const pointer

const SmartPtr<const CD> p = &goodCD; // const object, const pointer

we can initialize const pointers with non-const pointers and we can initialize pointers to const objects with pointers to non-consts.

CD \*pCD = new CD("Famous Movie Themes");

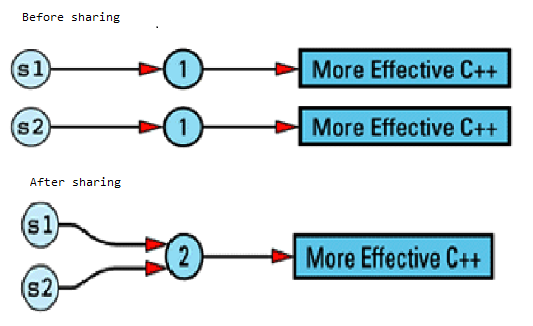
const CD \* pConstCD = pCD; // fine

But look what happens if we try the same thing with smart pointers:

SmartPtr<CD> pCD = new CD("Famous Movie Themes");

SmartPtr<const CD> pConstCD = pCD; // fine?

**Reference Counting. (Smart Reference Proxy)**



class RCString

{

private:

struct StringValue {

int refCount;

char \*data;

StringValue(const char \*initValue);

~StringValue();

};

StringValue \*value; // value of this String

public:

RCString(const char \*initValue = "");

RCString(const RCString& rhs);

RCString& operator=(const RCString& rhs);

char& operator[](int index)const; //For Const String

char& operator[](int index); //For non-Const String.

~RCString();

void showStr()const {

cout <<"Data:: "<<value->data<<", Address::"<< &(value->data)<<endl;

cout << "Ref cnt:: " << value->refCount<<endl;

cout << "---------------\n";

}

};

//Implementation of the const version of this function is straightforward, because it's a //read-only operation; the value of the string can't be affected

char& RCString::operator[](int index)const

{

if ((strlen(value->data) - 1) > index)

return value->data[index];

///else throw ArrayOutOfBoundExeception

}

//Non-const version, Since we don’t know whether it is read or write, so we are creating //new set of stringValue therefore it cannot create problem while write operation happening.

char& RCString::operator[](int index) //throw ArrayOutOfBoundExeception,

//Implement later..

{

// if we're sharing a value with other String objects, break off a

// separate copy of the value for ourselves

if (value->refCount > 0)

{

// decrement current value's refCount, because we won't be using

// that value any more

value->refCount--;

value = new StringValue(value->data); // make a copy of the value

// for ourselves.

}

// return a reference to a character inside our unshared StringValue

// object

return value->data[index];

}

RCString& RCString::operator=(const RCString& rhs)

{

if (this->value == rhs.value)

return \*this;

this->~RCString();

this->value = rhs.value;

this->value->refCount++;

return \*this;

}

RCString::~RCString()

{

if ((--value->refCount) == 0) {

value->~StringValue();

}

}

RCString::RCString(const char \*initValue): value(new StringValue(initValue))

{

}

RCString::RCString(const RCString& rhs):value(rhs.value) {

++value->refCount;

}

RCString::StringValue::StringValue(const char \*initValue):refCount(1) {

data = new char[strlen(initValue) + 1];

strcpy(data, initValue);

}

RCString::StringValue::~StringValue(){

delete[] data;

}

int main() {

Problem with Write operation-When we modify a String's value, we must be careful to avoid modifying the value of other String objects. Because we all share same value among classes and unfortunately, there is no way for C++ compilers to tell us whether a use of operator [] is for a read or a write.

//Write Operation

RCString str1("RajeevKumarSharma"); //str1->refCount=1

RCString str2(str1); //str1->refCount=2,str2->refCount=2

//Both have same copy of string.

str2[0] = 'X'; //Now we will be modifying str2, but It should not

//be impact str1.

str1.showStr(); //str1->refCount=1

str2.showStr(); //str2->refCount=1, It have separate copy of string //"XajeevKumarSharma"

//Read Operation.

RCString str3("RajeevKumarNayan");

RCString str4(str3);

cout << str3[0] << endl; //Read, Unfortunately in operator[] call, we cannot

//determine whether it is called by read or write

//operation.

str3.showStr(); //str3->refCount=1

str4.showStr(); //str4->refCount=1

//Still having seperate copy of string //"RajeevKumarNayan" which is illogical.

const RCString str5("ThisIsConstString");

str5.showStr();

char ch1 = str5[3]; //Non Const RCString Read.

str5.showStr(); //No address change after read, because of const

RCString str6("ThisIsNonConstString")

str6.showStr();

char ch2 = str6[1]; //String changed while reading.

str6.showStr(); //Address changed.

getchar();

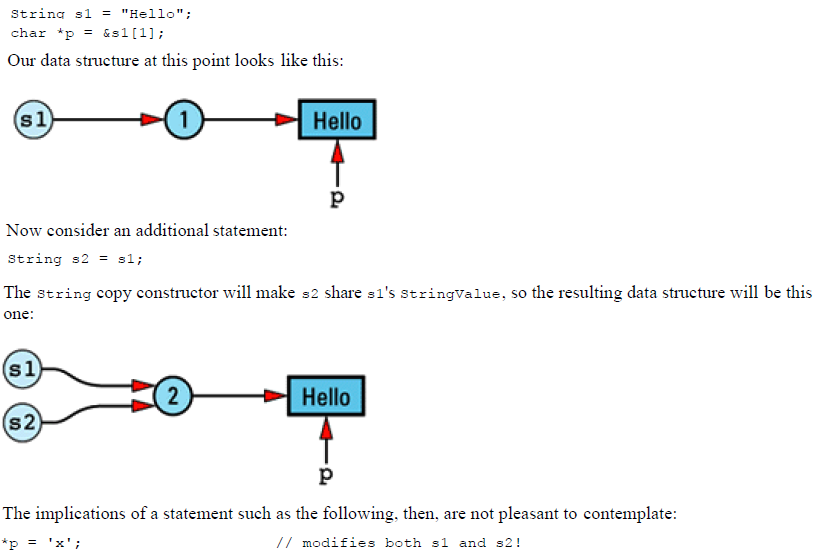
return 0;

}

**Pointers, References, and Copy-on-Write:**

This idea — that of sharing a value with other objects until we must write on our own copy of the value — has a long and distinguished history in Computer Science, especially in operating systems, where processes are routinely allowed to share pages until they want to modify data on their own copy of a page. The technique is common enough to have a name: ***copy-on-write***.

Copy-on-write allows us to preserve both efficiency and correctness.



There is no way the String copy constructor can detect this problem, because it has no way to know that a pointer into s1's ***StringValue*** object exists. And this problem isn't limited to pointers, it would exist in previous example, when we call to ***RCString*** non-const operator[].

So how we resolve this problem?

Its implement is not difficult, but it can reduce the amount of value sharing between objects. We can add a flag to each StringValue object indicating whether that object is shareable. Turn the flag on initially (the object is shareable), but turn it off whenever the

non-const operator[] is invoked on that object. Once the flag is set to false, it stays for forever.

class String {

private:

struct StringValue {

int refCount;

bool shareable; // added this

char \*data;

StringValue(const char \*initValue);

~StringValue();

};

...

};

String::StringValue::StringValue(const char \*initValue): refCount(1),

shareable(true) // added this

{

data = new char[strlen(initValue) + 1];

strcpy(data, initValue);

}

There is not much changes required to implements this. Of course, String's member functions must be updated to take the shareable field into account. Here's how the copy constructor implemented:

String::String(const String& rhs)

{

if (rhs.value->shareable) {

value = rhs.value; //If sharable is true then only

++value->refCount; //increase the ref cnt.

}

else {

value = new StringValue(rhs.value->data);

//Otherwise creates new copy.

}

}

And Finally, The non-const version of operator[] would be the only function to set the shareable flag to false.

char& String::operator[](int index)

{

if (value->refCount > 1) {

--value->refCount;

value = new StringValue(value->data);

}

value->shareable = false; // add this

return value->data[index];

}

Note: If we use the proxy class technique of to distinguish read usage from write usage in operator[], then we can reduce the number of StringValue objects that must be marked un-shareable.

**A Reference-Counting Base Class:**

Reference counting is not only useful for String object, but for any class in which different objects may have common values. So, if we could somehow write the reference counting code in a context-independent manner.

The first step is to create a base class, RCObject, for reference-counted objects. Any class wishing to take advantage of automatic reference counting must inherit from this class.

RCObject encapsulates the reference count itself, as well as functions for incrementing and decrementing that count. It also contains the code for destroying a value when it is no longer in use, i.e., when its reference count becomes 0. Finally, it contains a field that keeps track of whether this value is shareable, and it provides functions to query this value and set it to false. There is no need for a function to set the shareability field to true, because all values are shareable by default. As noted above, once an object has been tagged un-shareable, there is no way to make it shareable again. RCObject's class definition looks like this:

class RCObject {

public:

RCObject();

RCObject(const RCObject& rhs);

RCObject& operator=(const RCObject& rhs); //Useless method.

virtual ~RCObject() = 0;

void addReference();

void removeReference();

void markUnshareable();

bool isShareable() const;

bool isShared() const;

private:

int refCount;

bool shareable;

};

RCObjects can be created and destroyed; they can have new references added to them and can have current references removed; their shareability status can be queried and can be disabled; and they can report whether they are currently being shared.

RCObject::RCObject(): refCount(1), shareable(true) {}

RCObject::RCObject(const RCObject&): refCount(1), shareable(true) {}

RCObject& RCObject::operator=(const RCObject&){

return \*this;

}

RCObject::~RCObject() {} // virtual dtors must always be implemented,

// even if they are pure virtual and do nothing

void RCObject::addReference() { ++refCount; }

void RCObject::removeReference(){

if (--refCount == 0)

delete this;

}

void RCObject::markUnshareable(){

shareable = false;

}

bool RCObject::isShareable() const{

return shareable;

}

bool RCObject::isShared() const{

return refCount > 1;

}

To take advantage of our new reference-counting base class, we modify StringValue to inherit its reference counting capabilities from RCObject.

class String

{

private:

struct StringValue :public RCObject {

StringValue(const char \* initValue) {

int size = strlen(initValue);

data = new char[size + 1];

memset(data, 0, size + 1);

memcpy(data, initValue,size);

}

~StringValue() {

delete[] data;

}

char \*data;

};

StringValue \*value; // value of this String

public:

String(const char \*initValue = "") :value( new StringValue(initValue)) {}

String(const String& rhs):value(rhs.value) {

value->addRef(); //Point to same mem loc, increase ref cnt

}

~String() {

value->removeRef(); //Decrement the ref cnt. If ref cnt zero,

//then only delete the string value(data)

}

//Assignment operator overload.

String& operator=(const String& rhs) {

if (value == rhs.value)

return \*this;

value->removeRef(); //Decrement lhs refCnt and delete the

//data if refCnt=0 by calling removeRef()

value = rhs.value; //Assign value to the lhs.

value->addRef(); //Add ref cnt of lhs.

return \*this;

}

const char& operator[](int index)const { //For Const String

return value->data[index];

}

char& operator[](int index) { //For non-Const String.

if (value->isShared()) {

value->removeRef();

value = new StringValue(value->data);

}

value->markUnsharable();

return value->data[index];

}

void showStr()const {

cout << "Data:: " << value->data << ", Ref cnt:: " <<

value->getRefCnt() << ", Address::" << &(value->data) << endl;;

}

};

//Note: Still this item is incomplete….

**Item 30: Proxy classes:**

This much is legal:

int data[10][20]; // 2D array: 10 by 20

The corresponding construct using variables as dimension sizes, however, is not:

void processInput(int dim1, int dim2)

{

int data[dim1][dim2]; // error! array dimensions

... // must be known during compilation

}

It's not even legal for a heap-based allocation:

int \*data = new int[dim1][dim2]; // error!

**Implementing Two-Dimensional Arrays**:

we can define a class template for two -dimensional arrays:

template<class T>

class Array2D {

public:

Array2D(int dim1, int dim2);

...

};

Now we can define the arrays we want:

Array2D<int> data(10, 20); // fine

Array2D<float> \*data = new Array2D<float>(10, 20); // fine

void processInput(int dim1, int dim2){

Array2D<int> data(dim1, dim2); // fine

...

}

Let’s user want to perform operation:

cout << data[3][6];

but how do we declare the indexing operator in Array2D Class ?

template<class T>

class Array2D {

public:

// declarations that won't compile

T& operator[][](int index1, int index2);

const T& operator[][](int index1, int index2) const;

...

};

template<class T>

class Array2D {

private:

T \*\*arr;

int r, c;

public:

Array2D(int row, int col):r(row),c(col) {

arr = new T \* [r];

for (int i = 0;i < c;i++)

arr[i] = new T[c];

}

~Array2D() {

for (int i = r - 1;i >= 0;i--)

{

cout << "deleting mem loc::" << arr[i] << endl;

delete[] arr[i];

}

cout << "finally deleted arr::" << arr << endl;

delete []arr;

}

void setElement() {

//Set Elements into the array.

}

void showElement() {

//Show elements from array.

}

//Now Implementation of Proxy Class.

class Array1D {

private:

T \*oneDArray;

public:

Array1D(T \*mVar):oneDArray(mVar){}

T& operator[](int index) {

return oneDArray[index];

}

const T& operator[](int index) const{} //For const.

};

Array1D operator[](int index) {

return Array1D(arr[index]);

}

const Array1D operator[](int index) const {} //For const.

};

* Each Array1D object stands for a one - dimensional array that is absent from the conceptual model used by clients of Array2D.
* Objects that stand for other objects are often called proxy objects, and the classes that give rise to proxy objects are often called proxy classes.
* In this example, Array1D is a proxy class. Its instances stand for one - dimensional arrays that, conceptually, do not exist. The terminology for proxy objects and classes is far from universal; objects of such classes are also sometimes known as surrogates.

Array2D<int> a1(10, 10);

a1.setElement();

a1.showElement();

cout << a1[9][9]; //Now It will Work.

Above syntax interpreted as: (A1.operator[9]).operator[9];

**Distinguishing Reads from Writes via operator []**

A reference-counted string type that supports operator []. A common problem is,

How to distinguish a call of operator [] for read and write?

A string type supporting operator[] allows clients to write code like this:

String s1, s2; // A string-like class; the use of proxies

// keeps this class from conforming to the

// standard string interface.

cout << s1[5]; // read s1

s2[5] = 'x'; // write s2

s1[3] = s2[8]; // write s1, read s2

Note that operator[] can be called in two different contexts: to read a character or to write a character. Reads are known as *rvalue* usages; writes are known as *lvalue* usages.

lvalue goes on the left-hand side (modifiable) of an assignment and a rvalue goes on the right-hand side (non-modifiable).

So, how we distinguish between lvalue and rvalue usage of operator[]because, especially for reference counted data structures, reads can be much less expensive to implement than writes (writes of reference-counted objects may involve copying an entire data structure, but reads require simple returning of a value).

Unfortunately, inside operator[], there is no way to determine the context in which the function was called; it is not possible to distinguish lvalue usage from rvalue usage within operator[].

Based on our last implementation, we had overloaded operator [] based on its const-ness, and that allows us to distinguish reads from writes.

class String {

public:

const char& operator[](int index) const; // for reads

char& operator[](int index); // for writes

...

};

But this also won't work. Compilers choose between const and non-const member functions by looking only at whether the *object* invoking a function is const. No consideration is given to the context in which a call is made. Hence:

String s1, s2;

...

cout << s1[5]; // calls non-const operator[],because s1 isn't const

s2[5] = 'x'; // also calls non-const operator[]: s2 isn't const

s1[3] = s2[8]; // both calls are to non-const operator[], because

// both s1 and s2 are non-const objects.

So, Overloading operator[], then, fails to distinguish reads from writes.

Our approach is based on the fact that though it may be impossible to tell whether operator[] is being invoked in a lvalue or an rvalue context from within operator[]. We can still treat reads differently from writes if we *delay* our lvalue-versus-rvalue actions until we see how the result of operator []is used. All we need is a way to postpone our decision on whether our object is being read or written until *after* operator[] has returned. (This is an example of *lazy evaluation).*

A proxy class allows us to modify operator[] to return a *proxy* for a string character instead of a string character itself. We can then wait to see how the proxy is used. If it's read, we can call to operator[] as a read. Otherwise call operator[] as a write.

First, we must understand the proxies being used in this case. There are only three things we can do with this proxy:

* Create it, i.e., specify which string character it stands for.
* Use it as the target of an assignment, in which case we are really making an assignment to the string character it stands for. When used in this way, a proxy represents a lvalue use of the string on which operator[]was invoked.
* Use it in any other way. When used like this, a proxy represents a rvalue use of the string on which operator[] was invoked.

The class definitions for a reference-counted String using a proxy class to distinguish between lvalue and rvalue usages of operator[]:

class RCObject {

public:

RCObject() :refCount(1) {}

RCObject(const RCObject &rhs) :refCount(1) {}

RCObject& operator =(const RCObject &rhs) { return \*this; }

virtual ~RCObject() {}

void addRef() { ++refCount; }

void removeRef() {

if (--refCount == 0)

delete this;

}

bool isShared()const {

return refCount > 1;

}

private:

short refCount;

};

class String { // Reference-counted strings Class

public:

class CharProxy { // Proxies for string chars

public:

CharProxy(String& str, int index); // creation

CharProxy& operator=(const CharProxy& rhs); // lvalue uses

CharProxy& operator=(char c); // lvalue uses

operator char() const; // rvalue uses

private:

String & theString; // string this proxy pertains

// to char within that string

int charIndex; // this proxy stands for.

};

const CharProxy operator[](int index) const; //for const Strings

CharProxy operator[](int index); //for non-const Strings

struct StringValue :public RCObject { //StringValue class

StringValue(const char \* initValue) {

int size = strlen(initValue);

data = new char[size + 1];

memset(data, 0, size + 1);

memcpy(data, initValue, size);

}

~StringValue() {

delete[] data;

}

char \*data;

};

String(const char \*initValue = "") :value(new StringValue(initValue)) {}

~String() {

value->removeRef(); //Decrement the ref cnt. If ref cnt zero,

}

String(const String& rhs) :value(rhs.value) {

value->addRef(); //Point to same mem loc, increase ref cnt

}

String& operator=(const String& rhs) {

if (value == rhs.value)

return \*this;

value->removeRef(); //Decrement lhs refCnt and delete the

//data if refCnt=0 by calling removeRef()

value = rhs.value; //Assign value to the lhs.

value->addRef(); //Add ref cnt of lhs.

return \*this;

}

private:

StringValue \*value; // value of this String

};

String::CharProxy& String::CharProxy::operator=(const CharProxy& rhs)

{

// if the string is sharing a value with other String objects,

// break off a separate copy of the value for this string only

if (theString.value->isShared()) {

theString.value->removeRef();

theString.value = new StringValue(theString.value->data);

}

// now make the assignment, assign the value of the char

// represented by rhs to the char represented by \*this

theString.value->data[charIndex] =

rhs.theString.value->data[rhs.charIndex];

return \*this;

}

String::CharProxy& String::CharProxy::operator=(char c)

{

if (theString.value->isShared()) {

theString.value->removeRef();

theString.value = new StringValue(theString.value->data);

}

theString.value->data[charIndex] = c;

return \*this;

}

const String::CharProxy String::operator[](int index) const

{

return CharProxy(const\_cast<String&>(\*this), index);

}

String::CharProxy String::operator[](int index)

{

return CharProxy(\*this, index);

}

String::CharProxy::CharProxy(String& str, int index)

: theString(str), charIndex(index) {}

String::CharProxy::operator char() const

{

return theString.value->data[charIndex];

}

Consider first this statement:

String str1=”Rajeev”;

cout<<str1[3];

The expression str1[0] return CharProxy object using method:

CharProxy operator[](int index);

Now, returned CharProxy object have reference of str1 and its index ( in this case 3). After that it will be looking for ostream operator overload in CharProxy class and unfortunately not found in same class. It finds one: the implicit conversion from CharProxy to char declared in the CharProxy class. It automatically invokes this conversion operator and convert CharProxy-to-char and later this char used for printing.

Lvalue usage is handled differently. Consider below expression.

String str1=”Rajeev”;

String str2(str1);

Str2[0]=’r’;

Before assignment both str1 and str2 share same memory location and having refCnt=2.

Now Str[0]=’r’; We had seen earlier Str[0] expression returns CharProxy object, after that it invoke assignment operator overload method with parameter char(‘r’).

**Note**: *The target of the assignment is a CharProxy, so the assignment operator that's called is in the CharProxy class. This is crucial, because inside a CharProxy assignment operator, we know that the CharProxy object being assigned to is being used as a lvalue.*

String::CharProxy& String::CharProxy::operator=(char c)

{

if (theString.value->isShared()) {

theString.value->removeRef();

theString.value = new StringValue(theString.value->data);

}

theString.value->data[charIndex] = c;

return \*this;

}

Now it is checking for string is shared or not. If shared, then

* Decrease the refCnt by 1
* Create new string with same string value and finally
* Return reference of character on specified index.

Else directly return the reference of character on specified index.

Similarly, the below statement :

String str1=”Rajeev”;

String str2(str1);

String str3=”Sharma”

Str1[2]=str3[2];

Calls the assignment operator for two CharProxy objects, and inside that operator we know the object on the left is being used as a lvalue and the object on the right as a rvalue.

This time it called below method:

String::CharProxy& String::CharProxy::operator=(const CharProxy& rhs)

{

// if the string is sharing a value with other String objects,

// break off a separate copy of the value for this string only

if (theString.value->isShared()) {

theString.value->removeRef();

theString.value = new StringValue(theString.value->data);

}

// now make the assignment, assign the value of the char

// represented by rhs to the char represented by \*this

theString.value->data[charIndex] =

rhs.theString.value->data[rhs.charIndex];

return \*this;

}

Lsh part having value “Rajeev” and refCnt =2, similarly rhs having value “Sharma” and refCnt=1. Both lsh and rhs are CharProxy objects. Since lsh value is shared then if condition is true. Now it decreases refCnt and create new copy of lsh string. After that make the assignment in lsh string[index] with rhs string [index].

Now another statement is:

Const String str1=”Rajeev”;

Cout<<str1[0];

Now str1 is const string so it invokes the const version of operator[] in class String.

const String::CharProxy String::operator[](int index) const{

return CharProxy(const\_cast<String&>(\*this), index);

}

Above method will return the CharProxy object after removing const-ness of string object. Then called implicit conversion method operator char() const (because operator << is not defined in CharProxy) and return the desire index of string (char) which is printed in console.

**Limitation:**

The use of a proxy class is a nice way to distinguish lvalue and rvalue usage of operator[]. If String::operator[] returns a CharProxy instead of a char&, that code will no longer compile:

String s1 = "Hello";

char \*p = &s1[1]; // error!

The expression s1[1] returns a CharProxy, so the type of the expression on the right-hand side of the "=" is CharProxy\*. There is no conversion from a CharProxy\* to a char\*, so the initialization of p fails to compile.

To eliminate this problem, need to overload the address-of operators for the CharProxy class:

class String {

public:

class CharProxy {

public:

...

char \* operator&();

const char \* operator&() const;

...

};

...

};

These functions are easy to implement. The const function just returns a pointer to a const version of the character represented by the proxy:

const char \* String::CharProxy::operator&() const{

return &(theString.value->data[charIndex]);

}

The non-const function is a bit more work, because it returns a pointer to a character that may be modified.

char \* String::CharProxy::operator&()

{

// make sure the character to which this function returns

// a pointer isn't shared by any other String objects

if (theString.value->isShared()) {

theString.value = new StringValue(theString.value->data);

}

// we don't know how long the pointer this function

// returns will be kept by clients, so the StringValue

// object can never be shared

theString.value->markUnshareable();

return &(theString.value->data[charIndex]);

}

***Complete Auto\_ptr Implementation:***

template<class T>

class auto\_ptr {

public:

explicit auto\_ptr(T \*p = 0);

template<class U> // copy constructor member

auto\_ptr(auto\_ptr<U>& rhs); // initialize a new auto\_ptr

// with any compatible auto\_ptr

~auto\_ptr();

template<class U> // assignment operator member template

auto\_ptr<T>& operator=(auto\_ptr<U>& rhs); // assign from any compatible // auto\_ptr.

T& operator\*() const;

T\* operator->() const;

T\* get() const; // return value of current dumb pointer

T\* release(); // relinquish ownership of current dumb

// pointer and return its value

void reset(T \*p = 0); // delete owned pointer; assume ownership of p

private:

T \*pointee;

template<class U> // make all auto\_ptr classes

friend class auto\_ptr<U>; // friends of one another

};

template<class T>

inline auto\_ptr<T>::auto\_ptr(T \*p): pointee(p){}

template<class T>

inline auto\_ptr<T>::auto\_ptr(auto\_ptr<U>& rhs): pointee(rhs.release()){}

template<class T>

inline auto\_ptr<T>::~auto\_ptr(){

delete pointee;

}

template<class T>

template<class U>

inline auto\_ptr<T>& auto\_ptr<T>::operator=(auto\_ptr<U>& rhs){

if (this != &rhs) reset(rhs.release());

return \*this;

}

template<class T>

inline T& auto\_ptr<T>::operator\*() const{

return \*pointee;

}

template<class T>

inline T\* auto\_ptr<T>::operator->() const{

return pointee;

}

template<class T>

inline T\* auto\_ptr<T>::get() const{

return pointee;

}

template<class T>

inline T\* auto\_ptr<T>::release(){

T \*oldPointee = pointee;

pointee = 0;

return oldPointee;

}

template<class T>

inline void auto\_ptr<T>::reset(T \*p)

{

if (pointee != p) {

delete pointee;

pointee = p;

}

}

template<class T>

class auto\_ptr {

public:

explicit auto\_ptr(T \*p = 0) : pointee(p) {}

template<class U>

auto\_ptr(auto\_ptr<U>& rhs) : pointee(rhs.release()) {}

~auto\_ptr() { delete pointee; }

template<class U>

auto\_ptr<T>& operator=(auto\_ptr<U>& rhs){

if (this != &rhs) reset(rhs.release());

return \*this;

}

T& operator\*() const { return \*pointee; }

T\* operator->() const { return pointee; }

T\* get() const { return pointee; }

T\* release(){

T \*oldPointee = pointee;

pointee = 0;

return oldPointee;

}

void reset(T \*p = 0)

{

if (pointee != p) {

delete pointee;

pointee = p;

}

}

private:

T \*pointee;

template<class U> friend class auto\_ptr<U>;

};