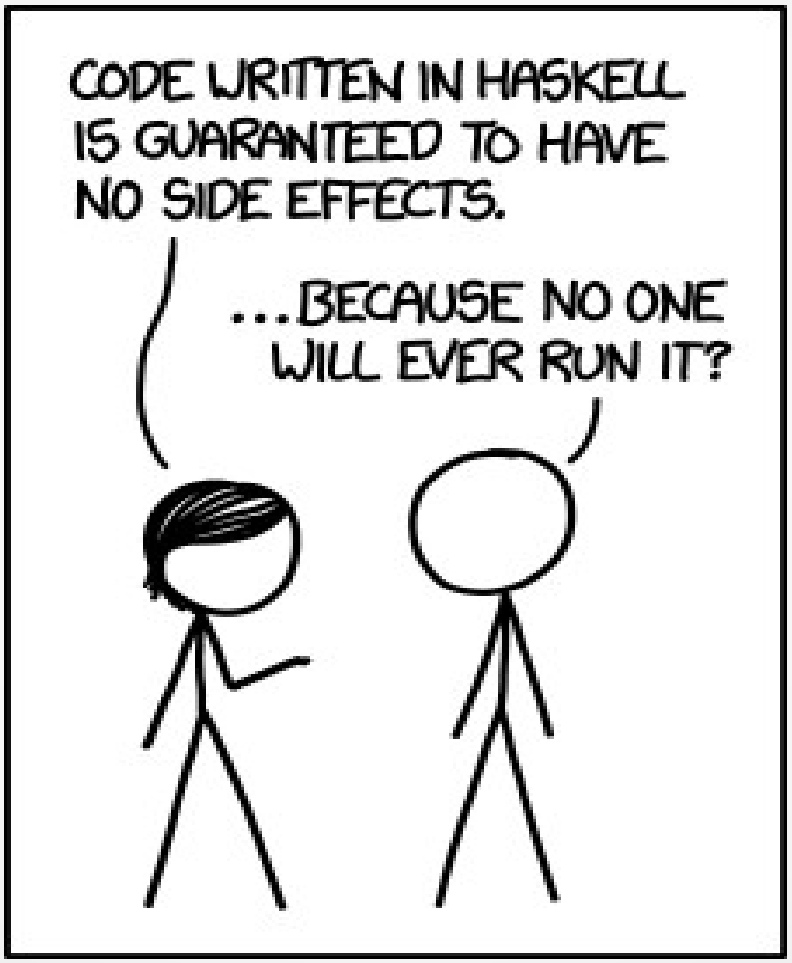
Lazy Evaluation and Reference Counting



Krishna Kumar

Lazy Evaluation - Lazy Fetching

- From the perspective of efficiency, the best computations are those you never perform at all.
- imagine you've got a program that uses large objects containing many constituent fields.
- Such objects must persist across program runs, so they're stored in a database.
- Each object has a unique object identifier that can be used to retrieve the object from the database:
- Because LargeObject instances are big, getting all the data for such an object might be a costly database operation, especially if the data must be retrieved from a remote database and pushed across a network.
- In some cases, the cost of reading all that data would be unnecessary.

Lazy Fetching

- class LargeObject { // large persistent objects
- public:
  - LargeObject(ObjectID id); // restore object from disk
  - const string& field1() const; // value of field 1
  - int field2() const; // value of field 2
  - double field3() const;
  - const string& field4() const;
  - const string& field5() const;
- };
- void restoreAndProcessObject(ObjectID id) {
  - LargeObject object(id);
  - if (object.field2() == 0) { cout << "Object " << id << ": null field2.\n";}
- } //Here only the value of field2 is required, so any effort spent setting up the other fields is wasted

Lazy Fetching (cont...)

```
LargeObject::LargeObject(ObjectID id)
    : oid{id}, field1Value{nullptr}, field2Value{nullptr}, ...{}
const string& LargeObject::field1() const {
    if (!field1Value) { // if null
        read the data for field 1 from the database and make
        field1Value point to it;
    }
    return *field1Value;
}
```

- The best way to say modifying const variable is to declare the pointer fields mutable , which means they can be modified inside any member function, even inside const member functions. That’s why the fields inside LargeObject above are declared mutable .

Lazy Fetching - Implementation

```
class LargeObject {
    public:
        LargeObject(ObjectID id);
        const string& field1() const;
        int field2() const;
        ...
    private:
        ObjectID oid;
        mutable string *field1Value;
        mutable int *field2Value;
        …
};
```

- The lazy approach to this problem is to read no data from disk when a LargeObject object is created.
- Instead, only the “shell” of an object is created, and data is retrieved from the database only when that particular data is needed inside the object.

Lazy Expression

- template<class T>
- class Matrix { ... }; // for homogeneous matrices
- Matrix<int> m1(1000, 1000);
- Matrix<int> m2(1000, 1000); // a 1000 by 1000 matrix
- ...

  Matrix<int> m3 = m1 + m2; // add m1 and m2
- The usual implementation of operator+ would use eager evaluation; in this case it would compute and return the sum of m1 and m2 . That's a fair amount of computation (1,000,000 additions), and of course there's the cost of allocating the memory to hold all those values, too.

Lazy Expression - Implementation

- The function would usually look something like this:

```
matrix operator +(matrix const& a, matrix const& b);
```

- Now, to make this function lazy, it's enough to return a proxy instead of the actual result:

```
struct matrix_add {
    matrix_add(matrix const& a, matrix const& b) : a(a), b(b) { }
    operator matrix() const {   matrix result;        // Do the addition.
        return result;
    }
private:
    matrix const& a, b;
};
matrix_add operator +(matrix const& a, matrix const& b) {    return matrix_add(a, b); }
```

Lazy Evaluation - Reference Counting

- Consider this code:

  class String { ... }; // a string class

  - String s1 = "Hello";
  - String s2 = s1; // call String copy ctor
- A common implementation for the String copy constructor would result in s1 and s2 each having its own copy of “Hello” after s2 is initialized with s1 . Such a copy constructor would incur a relatively large expense, is called “eager evaluation”.
- The lazy approach is a lot less work. Instead of giving s2 a copy of s1’s value, we have s2 share s1 ’s value. All we have to do is a little book-keeping so we know who’s sharing what.
- cout << s1; // read s1’s value
- cout << s1 + s2; // read s1’s and s2’s values
- s2.convertToUpperCase(); // it’s crucial that only s2 ’s value be changed, not s1

Limiting the number of objects of a class

- For example, you've got only one printer in your system,
- so you'd like to somehow limit the number of printer objects to one. Or you've got only 16 file descriptors you can hand out, so you've got to make sure there are never more than that many file descriptor objects in existence.
- Each time an object is instantiated, we know one thing for sure: a constructor will be called. That being the case, the easiest way to prevent objects of a particular class from being created is to declare the constructors of that class private:

```
class CantBeInstantiated {

private:

    CantBeInstantiated();

    CantBeInstantiated(const CantBeInstantiated&);

    ...

};
```

One object of a class

- class PrintJob; // forward declaration

  class Printer {

  public:
  - void submitJob(const PrintJob& job);
  - void reset();
  - void performSelfTest();
  - friend Printer& thePrinter();

  private:
  - Printer();
  - Printer(const Printer& rhs);

  };

  Printer& thePrinter() {
  - static Printer p;
  - return p;

  }

- we can encapsulate the printer object inside a function so that everybody has access to the printer, but only a single printer object is created.
- thePrinter contains a ***static*** Printer object. That means only a single object will be created.
- Client code:
  - thePrinter().reset();
  - thePrinter().submitJob(buffer);

Using a static member function

```
class Printer {
public:
  – static Printer& thePrinter();
private:
  – Printer();
  – Printer(const Printer& rhs);
};
Printer& Printer::thePrinter() {
    static Printer p;
    return p;
}
```

- Client implementation:
  - Printer::thePrinter().reset();
  - Printer::thePrinter().submitJob(buffer);
-

Limiting Object Instantiations

- class Printer {
- public:
  - class TooManyObjects{};
  - // exception class for use
  - // when too many objects requested

    Printer();

    ~Printer();
- private:
  - static size_t numObjects;
  - Printer(const Printer& rhs);
- };

- TooManyObjects :

  // Obligatory definition of the class static

  size_t Printer::numObjects = 0;

  Printer::Printer() {

  if (numObjects >= 1)

  throw TooManyObjects();

  ++numObjects;

  }

  Printer::~Printer() {

  --numObjects;

  }

Contexts for Object Construction

- Suppose we have a special kind of printer, say, a color printer, so of course we'd inherit from it:

```
class ColorPrinter: public Printer { ...};
```

- Now suppose we have one generic printer and one color printer in our system:

```
Printer p;

ColorPrinter cp;
```

- How many Printer objects result from these object definitions? The answer is two: one for p and one for the Printer part of cp. At runtime, a TooManyObjects exception will be thrown during the construction of the base class part of cp .
- A similar problem occurs when Printer objects are contained inside other objects

Finite State Automata

- One can’t derive from classes with private constructors.
- class FSA {
- public:
  - static FSA * makeFSA(); // pseudo-constructors
  - static FSA * makeFSA(const FSA& rhs);
- private:
  - FSA();
  - FSA(const FSA& rhs);
- };
- FSA * FSA::makeFSA() { return new FSA(); }
- FSA * FSA::makeFSA(const FSA& rhs) { return new FSA(rhs); }
- Unlike the thePrinter function that always returned a reference to a single object, each makeFSA pseudo-constructor returns a pointer to a unique object. That’s what allows an unlimited number of FSA objects to be created.

Finite State Automata

- The fact that each pseudo-constructor calls new implies that callers will have to remember to call delete.
- Otherwise a resource leak will be introduced.
- Callers who wish to have delete called automatically when the current scope is exited can store the pointer returned from makeFSA in a shared_ptr object; such objects automatically delete what they point to when they themselves go out of scope

```
// indirectly call default FSA constructor
std::shared_ptr<FSA> pfsa1(FSA::makeFSA());
// indirectly call FSA copy constructor
std::shared_ptr<FSA> pfsa2(FSA::makeFSA(*pfsa1));
```

- Use pfsa1 and pfsa2 as normal pointers, but don’t worry about deleting them

Allowing objects to come and go

- Our use of the thePrinter function to encapsulate access to a single object limits the number of Printer objects to one, but it also limits us to a single Printer object for each run of the program
- As a result, it's not possible to write code like this:
  - create Printer object p1;
  - use p1;
  - destroy p1;
  - create Printer object p2;
  - use p2;
  - destroy p2;
- This design never instantiates more than a single Printer object at a time, but it does use different Printer objects in different parts of the program. It seems unreasonable that this isn't allowed.

Object-counting code & pseudo constructor

- class Printer {
- public:
  - class TooManyObjects{};
  - // pseudo-constructor
  - static Printer * makePrinter();
  - ~Printer();
  - void submitJob(const PrintJob& job);
  - void reset();
  - void performSelfTest();
- private:
  - static size_t numObjects;
  - Printer();
  - Printer(const Printer& rhs);
- };

// Obligatory definition of class static

size_t Printer::numObjects = 0;

- Printer::Printer() {
  - if (numObjects >= 1)
    - throw TooManyObjects();
  - proceed with normal object construction here;
  - ++numObjects;
- }
- Printer * Printer::makePrinter()

{ return new Printer; }

Object counting &pseudo-ctr (cont..

- If the notion of throwing an exception when too many objects are requested strikes you as unreasonably harsh, you could have the pseudo-constructor return a null pointer instead.
- Clients would then have to check for this before doing anything with it, of course.
- Clients use this Printer class just as they would any other class, except they must call the pseudo-constructor function instead of the real constructor:
- Printer p1; // error! default ctor is private
- Printer *p2 = Printer::makePrinter(); // fine, indirectly calls default ctor
- Printer p3 = *p2; // error! copy ctor is private
- p2->performSelfTest(); // all other functions are called as usual
- ...
- delete p2; // avoid resource leak; this would be unnecessary if p2 were a shared_ptr

Object counting & pseudo-ctr …

- class Printer {
- public:
  - class TooManyObjects{};
  - // pseudo-constructors
  - static Printer * makePrinter();
  - static Printer * makePrinter(const Printer& rhs);
- private:
  - static size_t numObjects;
  - static const size_t maxObjects = 10;
  - Printer();
  - Printer(const Printer& rhs);

  };

- // Obligatory definitions of class statics
- size_t Printer::numObjects = 0;
- const size_t Printer::maxObjects;
- Printer::Printer() {
  - if (numObjects >= maxObjects)
    - throw TooManyObjects();
- }
- Printer::Printer(const Printer& rhs) {
  - if (numObjects >= maxObjects)
    - throw TooManyObjects();
- }
- Printer * Printer::makePrinter() {
  - return new Printer; }
- Printer * Printer::makePrinter(const Printer& rhs) { return new Printer(rhs); }

An Object-Counting Base Class

- template<class BeingCounted>
- class Counted {
- public:
  - class TooManyObjects{};
  - static size_t objectCount() { return numObjects; }
- protected:
  - Counted();
  - Counted(const Counted& rhs);
  - ~Counted() { --numObjects; }
- private:
  - static size_t numObjects;
  - static const size_t maxObjects;
  - void init();
- };

- template<class BeingCounted>

  Counted<BeingCounted>::Counted()

  { init(); }
- template<class BeingCounted>

  Counted<BeingCounted>::Counted(const Counted<BeingCounted>&) {
  - Init(); }
- template<class BeingCounted>

  void Counted<BeingCounted>::init() {
  - if (numObjects >= maxObjects)
    - throw TooManyObjects();
  - ++numObjects;

  }

Limited object instantiation

- class Printer: private Counted<Printer> {
- public:
  - // pseudo-constructors
  - static Printer * makePrinter();
  - static Printer * makePrinter(const Printer& rhs);
  - ~Printer();
  - void submitJob(const PrintJob& job);
  - void reset();
  - void performSelfTest();
  - using Counted<Printer>::objectCount;
  - using Counted<Printer>::TooManyObjects;
- private:
  - Printer();
  - Printer(const Printer& rhs);
- };

- // make objectCount public in Printer
- class Printer: private Counted<Printer> {
- public:

  ...

  Counted<Printer>::objectCount;

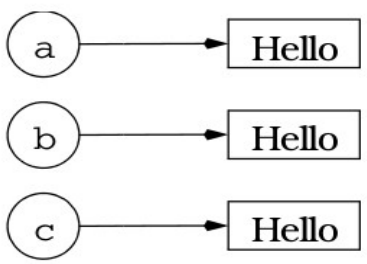
  ...
- };
- const size_t Counted<Printer>::maxObjects = 10;
- What will happen if these authors forget to provide a suitable definition for maxObjects ?
  - Simple: they'll get an error during linking, because maxObjects will be undefined.

Reference counting

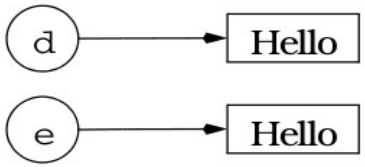
- Reference counting is a technique that allows multiple objects with the same value to share a single representation of that value.
- There are two common motivations for the technique:
  - The first is to simplify the bookkeeping surrounding heap objects. Once an object is allocated by calling new , it's crucial to keep track of who owns that object, because the owner — and only the owner — is responsible for calling delete on it. But ownership can be transferred from object to object as a program runs (by passing pointers as parameters, for example), so keeping track of an object's ownership is hard work.
  - Reference counting eliminates the burden of tracking object ownership, because when an object employs reference counting, it owns itself.
  - Second, If many objects have the same value, it's silly to store that value more than once. Instead, it's better to let all the objects with that value share its representation. Doing so not only saves memory, it also leads to faster-running programs, because there's no need to construct and destruct redundant copies of the same object value.

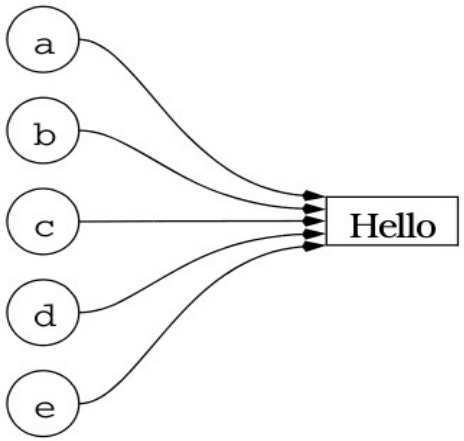
Reference counting …

- String a, b, c, d, e;
- a = b = c = d = e = "Hello";
- It should be apparent that objects a through e all have the same value, namely “Hello” .
- How that value is represented depends on how the String class is implemented, but a common implementation would have each String object carry its own copy of the value.



a





b

Reference counting …

- Here only one copy of the value “Hello” is stored, and all the String objects with that value share its representation.
- In practice, it isn’t possible to achieve this ideal, because we need to keep track of how many objects are sharing a value.
- If object is assigned a different value from “Hello” , we can’t destroy the value “Hello” , because four other objects still need it.
- On the other hand, if only a single object had the value “Hello” and that object went out of scope, no object would have that value and we’d have to destroy the value to avoid a resource leak.
- The need to store information on the number of objects currently sharing — referring to — a value means our ideal picture must be modified somewhat to take into account the existence of a reference count.

Implementing Reference Counting

- It’s important to recognize that we need a place to store the reference count for each String value.
- That place cannot be in a String object, because we need one reference count per string value, not one reference count per string object.
- That implies a coupling between values and reference counts, so we’ll create a class to store reference counts and the values they track.
- We’ll call this class StringValue , and because it is there only to help implement the String class, we’ll nest it inside String ’s private section.
- It will be convenient to give all the member functions of String full access to the StringValue data structure, so we’ll declare StringValue to be a struct .
- This is a trick worth knowing: nesting a struct in the private part of a class is a convenient way to give access to the struct to all the members of the class, but to deny access to everybody else (except, of course, friends of the class).

Implementing Reference Counting ...

- class String {
- private:
  - struct StringValue {
    - size_t refCount;
    - char *data;
    - StringValue(const char *initValue);
    - ~StringValue();
  - };
- ...
- };
- String::StringValue::StringValue(const char *initValue) : refCount(1) {
  - data = new char[strlen(initValue) + 1];
  - strcpy(data, initValue);
- }
- String::StringValue::~StringValue() {delete [] data; }

Implementing Reference Counting ...

- The primary purpose of StringValue is to give us a place to associate a particular value with a count of the number of String objects sharing that value. StringValue gives us that, and that's enough. We're now ready to walk our way through String's member functions. We'll begin with the constructors:
- class String {
  - public:
  - String(const char *initValue = "");
  - String(const String& rhs);
  - … };
- The first constructor is implemented about as simply as possible. We use the passed-in char* string to create a new StringValue object, then we make the String object we're constructing point to the newly minted StringValue :
  - String::String(const char *initValue) : value(new StringValue(initValue)) {}
- For client code that looks like this,
  - String s("More Effective C++"); // s =====> 1 ====> "More Effective C++"

Copy constructor

- The String copy constructor is efficient: the newly created String object shares the same StringValue object as the String object that's being copied:
- String::String(const String& rhs) : value(rhs.value) {
  - ++value->refCount;
- }
- Graphically, code like this,
  - String s1("More Effective C++");
  - String s2 = s1;
- results in this data structure:
- S1 ==|
  | ===> (2) ==> "More Effective C++"

  S2 ==|

String destructor

- The String destructor is also easy to implement, because most of the time it doesn't do anything. As long as the reference count for a StringValue is non-zero, at least one String object is using the value; it must therefore not be destroyed.
- Only when the String being destructed is the sole user of the value — i.e., when the value's reference count is 1 — should the String destructor destroy the StringValue object:
- class String {
- public:
  - ~String();
  - ...
- };
- String::~String() {
  - if (--value->refCount == 0) delete value;
- }
- Compare the efficiency of this function with that of the destructor for a non-reference-counted implementation. Such a function would always call delete and would almost certainly have a nontrivial runtime cost. Provided that different String objects do in fact sometimes have the same values, the implementation above will sometimes do nothing more than decrement a counter and compare it to zero.

Copy-on-Write

- Pursuing the example of reference-counting strings a bit further, we come upon a second way in which lazy evaluation can help us. Consider this code:
- String s = "Homer’s Iliad"; // Assume s is referencecounted
- ...
- cout << s[3]; // call operator[] to read s[3]
- s[3] = ’x’; // call operator[] to write s[3]
- To round out our examination of reference-counted strings, consider an array-bracket operator ([]), which allows individual characters within strings to be read and written

Copy-on-write ...

- class String {

  public:

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

  };
- 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:
- const char& String::operator[](int index) const {
  - return value->data[index];

  }
- This function performs sanity checking on index in the grand C++ tradition, which is to say not at all. As usual, if you'd like a greater degree of parameter validation, it's easy to add.

Copy-on-write (Non-const)

- The non- const version of operator[] is a completely different story.
- This function may be called to read a character, but it might be called to write one, too:
  - String s;
  - ...
  - cout << s[3]; // this is a read
  - s[5] = ’x’; // this is a write
- We’d like to deal with reads and writes differently. A simple read can be dealt with in the same way as the const version of operator[] above, but a write must be implemented in quite a different fashion
- When we modify a String ’s value, we have to be careful to avoid modifying the value of other String objects that happen to be sharing the same StringValue object. Unfortunately, there is no way for C++ compilers to tell us whether a particular use of operator[] is for a read or a write, so we must be pessimistic and assume that all calls to the non- const operator[] are for writes.

Copy-on-write (Non-const)

- To implement the non- const operator[] safely, we must ensure that no other String object shares the StringValue to be modified by the presumed write.
- In short, we must ensure that the reference count for a String 's StringValue object is exactly one any time we return a reference to a character inside that StringValue object.
- Char& String::operator[](int index) {

```
    // if we're sharing a value with other String objects,
    // break off a separate copy of the value for ourselves
    if (value->refCount > 1) {
        --value->refCount;
        // decrement current value's  refCount, because we won't  be using that value any more
        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];
  }
```

- This idea — that of sharing a value with other objects until we have to write on our own copy of the value - **Copy-on-write**

References

- Scott Meyer's “Effective C++: 35 New Ways to Improve Your Programs and Designs”
- Dov Bulka & David Mayher's “Efficient C++ Performance Programming Techniques”