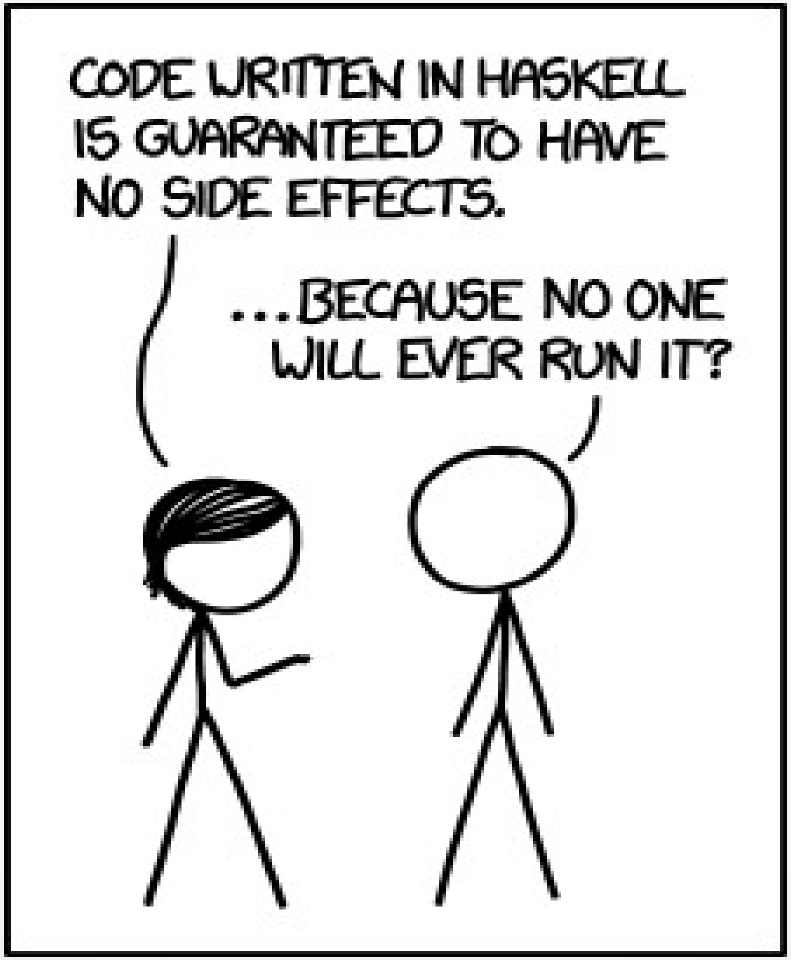
Lazy Evaluation and Reference Counting



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.

References

- Bjarne Stroustrup's “C++ Programming Language 4ed”
- Scott Meyer's “Effective Modern C++”
- Herb Sutter's Exceptional C++ and More Exceptional C++
- C++ Templates: the Complete Guide
- http://eli.thegreenplace.net/2014/variadic-templates-in-c/#id1
- http://lbrandy.com/blog/2013/03/variadic_templates/
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2080.pdf