More C++ Idioms/Counted Body

Contents

Counted Body/Reference Counting (intrusive)

Intent

Manage logical sharing of a resource/object, prevent expensive copying, and allow proper resource deallocation of objects that use dynamically allocated resources.

Also Known As

- Reference Counting (intrusive)
- Counted Body

Motivation

When Handle/Body idiom is used, quite often it is noticed that copying of bodies is expensive. This is because copying of bodies involves allocation of memory and copy construction. Copying can be avoided by using pointers and references, but these leave the problem of who is responsible for

cleaning up the object. Some handle must be responsible for releasing memory resources allocated for the body. Usually it is the last one. Without automatic reclamation of memory resources, it leaves a user-visible distinction between built-in types and user-defined types.

Solution and Sample Code

The solution is to add a reference count to the body class to facilitate memory management; hence the name "Counted Body." Memory management is added to the handle class, particularly to its implementation of initialization, assignment, copying, and destruction.

```
#include <cstring>
#include <algorithm>
#include <iostream>

class StringRep {
  friend class String;

  friend std::ostream &operator<<(std::ostream &out, StringRep const &str) {
    out << "[" << str.data_ << ", " << str.count_ << "]";
    return out;
  }

 public:
  StringRep(const char *s) : count_(1) {
    strcpy(data_ = new char[strlen(s) + 1], s);
  }

  ~StringRep() { delete[] data_; }

 private:
  int count_;
  char *data_;
};

class String {
 public:
  String() : rep(new StringRep("")) {
    std::cout << "empty ctor: " << *rep << "\n";
  }
  String(const String &s) : rep(s.rep) {
    rep->count_++;
    std::cout << "String ctor: " << *rep << "\n";
  }
  String(const char *s) : rep(new StringRep(s)) {
    std::cout << "char ctor:" << *rep << "\n";
  }
  String &operator=(const String &s) {
    std::cout << "before assign: " << *s.rep << " to " << *rep << "\n";
    String(s).swap(*this);  // copy-and-swap idiom
    std::cout << "after assign: " << *s.rep << " , " << *rep << "\n";
```

```
    return *this;
  }
  ~String() {  // StringRep deleted only when the last handle goes out of
scope.
    if (rep && --rep->count_ <= 0) {
      std::cout << "dtor: " << *rep << "\n";
      delete rep;
    }
  }

 private:
  void swap(String &s) throw() { std::swap(this->rep, s.rep); }

  StringRep *rep;
};
int main() {

  std::cout << "*** init String a with empty\n";
  String a;
  std::cout << "\n*** assign a to \"A\"\n";
  a = "A";

  std::cout << "\n*** init String b with \"B\"\n";
  String b = "B";

  std::cout << "\n*** b->a\n";
  a = b;

  std::cout << "\n*** init c with a\n";
  String c(a);

  std::cout << "\n*** init d with \"D\"\n";
  String d("D");

  return 0;
}
```

Gratuitous copying is avoided, leading to a more efficient implementation. This idiom presumes that the programmer can edit the source code for the body. When that's not possible, use Detached Counted Body. When counter is stored inside body class, it is called intrusive reference counting and when the counter is stored external to the body class it is known as non-intrusive reference counting. This implementation is a variation of shallow copy with the semantics of deep copy and the efficiency of Smalltalk name-value pairs.

The output should be as below:

```
*** init String a with empty
empty ctor: [, 1]

*** assign a to "A"
char ctor:[A, 1]
```

```
before assign: [A, 1] to [, 1]
String ctor: [A, 2]
dtor: [, 0]
after assign: [A, 2] , [A, 2]

*** init String b with "B"
char ctor:[B, 1]

*** b→a
before assign: [B, 1] to [A, 1]
String ctor: [B, 2]
dtor: [A, 0]
after assign: [B, 2] , [B, 2]

*** init c with a
String ctor: [B, 3]

*** init d with "D"
char ctor:[D, 1]
dtor: [D, 0]
dtor: [B, 0]
```

Consequences

Creation of multiple reference counts will result into multiple deletion of the same body, which is undefined. Care must be taken to avoid creating multiple reference counts to the same body. Intrusive reference counting easily supports it. With non-intrusive reference counting, programmer discipline is required to prevent duplicate reference counts.

Known Uses

- `boost::shared_ptr` (non-intrusive reference counting)
- `boost::intrusive_ptr` (intrusive reference counting)
- `std::shared_ptr`
- the Qt toolkit, e.g. `QString`

Related Idioms

- Handle Body
- Detached Counted Body (non-intrusive reference counting)
- Smart Pointer
- Copy-and-swap

References