SmartPointer

HERA-B Note 01-006

T. Glebe, C. Borgmeier

January 18, 2002

Abstract

SmartPointer is a template class which implements a pointer with reference counting. This simplifies significantly the object destruction and avoids memory leaks or segmentation violations problems.

Contents

1 Introduction

In C and C++ pointers always have the disadvantage that they potentially can point to obects on the heap which do not exist any more. While this problem is well under control in small software projects, it can be very difficult and time consuming to assure consistency between pointers and data in a large scale software project. On the other hand it may happen that pointers are never deleted because the control flow of the software has changed and a routine is left before the cleanup is done (e.g. if an exception occurs). The result is a memory leak.

A solution is an intelligent pointer which manages the object on the heap and counts how many pointers point to the object. Due to the reference counting the destructor of the last pointer pointing to the object will delete the object itself and thus consistency between pointers and data is automatically assured.

2 Code and compilation

The SmartPointer code consists of two files:

SmartPointer.hh Definition of the SmartPointer class

SmartPointer.icc SmartPointer implementation file

The `SmartPointer` class can be used by including the `SmartPointer.hh` header file:

```
// This is file myappl.C
#include "SmartPointer.hh"
```

SmartPointer will be distributed together with the HERA-B analysis framework BEE.

3 A simple example

The functionality of the `SmartPointer` class is shown in the following:

```
// A simple class
class point {
public:

  point() : x_(0.),y_(0.),z_(0.) {}
  point(float x, float y, float z) : x_(x), y_(y), z_(z) {}

  std::ostream& print(std::ostream& os) const {
    return os << " x: " << x_ << " y: " << y_ << " z: " << z_;
  }

  inline float x() const { return x_; }
  inline float y() const { return y_; }
  inline float z() const { return z_; }

  // sort according to z
  inline bool operator<(const point& rhs) const {
    return z_ < rhs.z_;
  }

private:
  float x_;
  float y_;
  float z_;
};
```

```
std::ostream& operator<<(std::ostream&os, const point& rhs) {
  return rhs.print(os);
}

// MAIN
int main(void) {

  point A(1.,2.,3.);

  // generate a copy of A on heap which is controlled
  // by the SmartPointer ptr
  SmartPointer<point> ptr(A);
  cout << " ptr: " << ptr << endl;

  SmartPointer<point> ptr2 = ptr;
  cout << "ptr2: " << ptr2 << endl;

  return 0;
}
```

The output is like this:

```
 ptr: 0x8059218 count: 1, object pointer: 0x8059228, object:  x: 1 y: 2 z: 3
ptr2: 0x8059218 count: 2, object pointer: 0x8059228, object:  x: 1 y: 2 z: 3
```

The `count` field shows the reference counter. The destruction of the first `SmartPointer` object will not affect the `point` object on the heap, but the destruction of the second one will automatically destroy the copy of A on the heap.

Note: It is not necessary to call `delete` for a SmartPointer object! The `SmartPointer` destructor will delete the object on the heap at the right moment.

4 Constructors

The `SmartPointer` class provides several constructors:

Empty `SmartPointer` The default constructor generates a SmartPointer which points to no object:

```
SmartPointer<point> ptr;
cout << "ptr: " << ptr << endl;
```

The output is:

```
ptr: (nil)
```

Default object SmartPointer can construct an object via its default constructor:

```
SmartPointer<point> ptr(true);
cout << "ptr: " << ptr << endl;
```

The output is:

```
ptr: 0x80595a8 count: 1, object pointer: 0x80595b8, object:  x: 0 y: 0 z: 0
```

Copy constructor A copy of an object can be constructed by its copy constructor:

```
point A(1.,2.,3.);
SmartPointer<point> ptr(A);
cout << "ptr: " << ptr << endl;
```

The output is:

```
ptr: 0x8059588 count: 1, object pointer: 0x8059598, object:  x: 1 y: 2 z: 3
```

Object on heap In case an object is already on the heap, you can hand it over to a SmartPointer. The SmartPointer will then do the destruction:

```
point* Aptr = new pointer(1.,2.,3.);
SmartPointer<point> ptr(Aptr);
cout << "ptr: " << ptr << endl;
```

The output is:

```
ptr: 0x8059588 count: 1, object pointer: 0x8059598, object:  x: 1 y: 2 z: 3
```

5 Access methods

A SmartPointer behaves like a usual pointer:

```
int main(void) {

  point A(1.,2.,3.);
  SmartPointer<point> ptr(A);

  cout << "A: " << *ptr << endl;
  cout << "A.x(): " << ptr->x() << endl;

  return 0;
}
```

The output is;

```
A:  x: 1 y: 2 z: 3
A.x(): 1
```

6 SmartPointer NULL

From section 4 it became clear that a SmartPointer can represent a NULL pointer. Therefore even with `SmartPointer` objects one has to be careful before using a dereference operator. To check for a NULL pointer one can use the `null()` member function:

```
int main(void) {

  SmartPointer<point> ptr = ...;

  if( !ptr.null() ) { cout << ptr->x() << endl; }

  return 0;
}
```

7 SmartPointer comparison

A comparison of SmartPointer objects is also defined:

```
int main(void) {

  SmartPointer<point> ptr1 = ...;
  SmartPointer<point> ptr2 = ...;

  // do ptr1 and ptr2 point to the same object?
  if( ptr1 == ptr2 ) { ... }
  // do ptr1 and ptr2 not point to same object?
  if( ptr1 != ptr2 ) { ... }

  return 0;
}
```

8 Sorting SmartPointers

In order to sort `SmartPointer` objects, the `operator<()` must be defined:

```
int main(void) {

  SmartPointer<point> ptr1 = ...;
  SmartPointer<point> ptr2 = ...;

  // check sort criterion
  // by calling point::operator<()
  if(ptr1 < ptr2) { ... }

  return 0;
}
```

Note: `SmartPointer::operator<()` does **not** compare the pointers to the objects, but the objects itself. Therefore the comparison will only work if the object has a `operator<()` method defined. In our example this means that the `point` class must have an `operator<(const point& rhs)` defined.

9 Final Remark

A reference counting pointer is not part of STL [1]. The only intelligent pointer in the STL is the `auto_ptr` class. An `auto_ptr` has a defined ownership and can only pass this ownership from one `auto_ptr` object to another `auto_ptr` object. This solves already the problem of memory leaks but doesn't allow that several `auto_ptr` objects point to the same object at the same time.

References

[1] Stepanov, A. and Meng Lee, The Standard Template Library, ANSI X3J16-94-0095 / ISO WG21-NO482