Username: Pralay Patoria **Book:** The C++ Standard Library: A Tutorial and Reference, Second Edition. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

7.11. Implementing Reference Semantics

In general, STL container classes provide value semantics and not reference semantics. Thus, the containers create internal copies of the elements they contain and return copies of those elements. Section 6.11.2, page 245, discusses the pros and cons of this approach and touches on its consequences. To summarize, if you want reference semantics in STL containers — whether because copying elements is expensive or because identical elements will be shared by different collections — you should use a smart pointer class that avoids possible errors. In addition, using a reference wrapper is possible.

Using Shared Pointers

As introduced in Section 5.2, page 76, the C++ standard library provides different smart pointer classes. For sharing objects between different containers, class `shared_ptr<>` is the appropriate smart pointer class. Using it for this purpose looks as follows:

Click here to view code image

```
// cont/refsem1.cpp

#include <iostream>
#include <string>
#include <set>
#include <deque>
#include <algorithm>
#include <memory>

class Item {
  private:
    std::string name;
    float  price;
  public:
    Item (const std::string& n, float p = 0) : name(n), price(p) {
    }
    std::string getName () const {
        return name;
    }
    void setName (const std::string& n) {
         name = n;
    }
    float getPrice () const {
        return price;
    }
    float setPrice (float p) {
        price = p;
    }
};

template <typename Coll>
void printItems (const std::string& msg, const Coll& coll)
{
    std::cout << msg << std::endl;
    for (const auto& elem : coll) {
        std::cout << ' ' << elem->getName() << ": "
                         << elem->getPrice() << std::endl;
    }
}

int main()
{
    using namespace std;

    // two different collections sharing Items
    typedef shared_ptr<Item> ItemPtr;
    set<ItemPtr> allItems;
    deque<ItemPtr> bestsellers;

    // insert objects into the collections
    // - bestsellers are in both collections
    bestsellers = { ItemPtr(new Item("Kong Yize",20.10)),
                    ItemPtr(new Item("A Midsummer Night's
Dream",14.99)),
                    ItemPtr(new Item("The Maltese Falcon",9.88)) };
    allItems = { ItemPtr(new Item("Water",0.44)),
                 ItemPtr(new Item("Pizza",2.22)) };
    allItems.insert(bestsellers.begin(),bestsellers.end());
```

```
    // print contents of both collections
    printItems ("bestsellers:", bestsellers);
    printItems ("all:", allItems);
    cout << endl;

    // double price of bestsellers
    for_each (bestsellers.begin(), bestsellers.end(),
              [] (shared_ptr<Item>& elem) {
                  elem->setPrice(elem->getPrice() * 2);
              });

    // replace second bestseller by first item with name "Pizza"
    bestsellers[1] = *(find_if(allItems.begin(),allItems.end(),
                               [] (shared_ptr<Item> elem) {
                                   return elem->getName() == "Pizza";
                               }));

    // set price of first bestseller
    bestsellers[0]->setPrice(44.77);

    // print contents of both collections
    printItems ("bestsellers:", bestsellers);
    printItems ("all:", allItems);
}
```

The program has the following output:

```
bestsellers:
 Kong Yize: 20.1
 A Midsummer Night's Dream: 14.99
 The Maltese Falcon: 9.88
all:
 Kong Yize: 20.1
 A Midsummer Night's Dream: 14.99
 The Maltese Falcon: 9.88
 Water: 0.44
 Pizza: 2.22

bestsellers:
 Kong Yize: 44.77
 Pizza: 2.22
 The Maltese Falcon: 19.76
all:
 Kong Yize: 44.77
 A Midsummer Night's Dream: 29.98
 The Maltese Falcon: 19.76
 Water: 0.44
 Pizza: 2.22
```

Note that using `shared_ptr<>` makes things significantly more complicated. For example, `find()` for sets, which looks for elements that have an equal value, will now compare the internal pointers returned by `new` :

```
allItems.find(ItemPtr(new Item("Pizza",2.22))) // can't be successful
```

So, you have to use the `find_if()` algorithm here.

If you call an auxiliary function that saves one element of the collections (an `ItemPtr`) somewhere else, the value to which it refers stays valid even if the collections get destroyed or all their elements are removed.

Using the Reference Wrapper

If it is guaranteed that the elements referred to in a container exist as long as the container exists, another approach is possible: using class `reference_wrapper<>` (see Section 5.4.3, page 132). For example, the following is possible, using class `Item` as introduced in the previous example:

Click here to view code image

```
std::vector<std::reference_wrapper<Item>> books;      // elements are references

Item f("Faust",12.99);
books.push_back(f);    // insert book by reference

// print books:
for (const auto& book : books) {
    std::cout << book.get().getName() << ": "
              << book.get().getPrice() << std::endl;
```

```
}

f.setPrice(9.99);     // modify book outside the containers
std::cout << books[0].get().getPrice() << std::endl;   // print price of first book

// print books using type of the elements (no get() necessary):
for (const Item& book : books) {
    std::cout << book.getName() << ": " << book.getPrice() << std::endl;
}
```

The advantage is that no pointer syntax is required. This, however, is also a risk because it's not obvious that references are used here.

Note that the following declaration isn't possible:

```
vector<Item&> books;
```

Note also that class `reference_wrapper<>` provides a conversion operator to `T&` so that the range-based `for` loop can be declared dealing with elements of type `Item&`. However, for a direct call of a member function for the first element, `get()` is necessary.

The program has following output (see *cont/refwrap1.cpp* for the complete example):

```
Faust: 12.99
9.99
Faust: 9.99
```