- Welcome
- Notes
- Workshops
- Translation
- Types
- Move Copy
- Templates
- Containers
- Lambda Expressions
- STL Containers
- STL Algorithms
- Smart Pointers
- Multi-Threading
- Assignments
- Handouts
- Practice
- Resources

Part F - Memory Model

Smart Pointers

Workshop 9

In this workshop, you merge two lists and use a smart pointer to ensure that memory is deallocated in the possible handling of an exception.

Learning Outcomes

Upon successful completion of this workshop, you will have demonstrated the abilities to

- create a program component of quadratic complexity
- use a smart pointer to move an object
- reflect on the topics learned in this workshop

Specifications

This workshop merges a description list with a price list to create a user-friendly price list. The workshop is in two parts:

1. Merging Data Sets
2. Smart Pointers

Files Provided

The three source files for this workshop are listed below

- Element.h (first version complete)
- List.h (first version complete)
- w9.cpp (incomplete)

The data files are at

- Prices.dat
- Descriptions.dat
- BadPrices.dat

Element Classes

The Description class holds a product code and a user-friendly description. The Price class holds a product code and the current price. The Product class holds a user-friendly description and the current price of a product.

Element Classes

The Description, Price and Product classes hold elemental information for the different lists.

```
// Workshop 9 - Smart Pointers
// Element.h

#include <iostream>
#include <iomanip>
#include <string>
#include <fstream>

extern const int FWC;
extern const int FWD;
extern const int FWP;

namespace w9 {

    struct Description {
        unsigned code;
        std::string desc;
        bool load(std::ifstream& f) {
            f >> code >> desc;
            return f.good();
        }
        void display(std::ostream& os) const {
            os << std::setw(FWC) << code << std::setw(FWD)
               << desc << std::endl;
        }
    };

    struct Price {
        unsigned code;
        double price;
        bool load(std::ifstream& f) {
            f >> code >> price;
            return f.good();
        }
        void display(std::ostream& os) const {
            os << std::setw(FWC) << code << std::setw(FWP)
               << price << std::endl;
        }
    };

    struct Product {
        std::string desc;
        double price;
        Product() {}
        Product(const std::string& str, double p) : desc(str), price(p) {}
        void display(std::ostream& os) const {
            os << std::setw(FWD) << desc << std::setw(FWP)
               << price << std::endl;
        }
    };
}
```

List Template

The List template defines a class that retrieves a list of types stored in a text file, holds the elements in an STL vector, provides access to them by index and displays them to an output stream.

```
// Workshop 9 - Smart Pointers
// List.h

#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <fstream>

namespace w9 {
    template <typename T>
    class List {
            std::vector<T> list;
          public:
            List() { }
            List(const char* fn) {
                std::ifstream file(fn);
                if (!file)
                    throw std::string("*** Failed to open file ") +
                     std::string(fn) + std::string(" ***");
                while (file) {
                    T e;
                    if (e.load(file))
                        list.push_back(*new T(e));
                }
            }
            size_t size() const { return list.size(); }
            const T& operator[](size_t i) const { return list[i]; }
            void operator+=(T* p) {
                list.push_back(*p);
            }
            void display(std::ostream& os) const {
                os << std::fixed << std::setprecision(2);
                for (auto& e : list)
                    e.display(os);
            }
        };

        template<typename T>
        std::ostream& operator<<(std::ostream& os, const List<T>& l) {
            l.display(os);
            return os;
        }
    }
```

Part 1 - Merging Data Sets

The main() program that uses the Element classes and the List template is shown below. Your first task is to complete the coding of the merge() function.

merge Function

This function returns the user-friendly price list. Each element in that list is of Product type. Your function compares elements in the two received lists for common product codes and builds the user-friendly list from the matching pairs. Your function allocates dynamic memory for each user-friendly element and adds that element to the new list by passing the object's raw pointer to the list's += operator. For this part of the workshop do not change any code in the header files provided.

```
 // Workshop 9 - Smart Pointers
 // w9.cpp

 #include <iostream>
 #include <iomanip>
 #include "Element.h"
 #include "List.h"

 const int FWC =  5;
 const int FWD = 12;
 const int FWP =  8;

 w9::List<w9::Product> merge(const w9::List<w9::Description>& desc,
  const w9::List<w9::Price>& price) {
     w9::List<w9::Product> priceList;



     // complete this part




     return priceList;
 }

 int main(int argc, char** argv) {
     std::cout << "\nCommand Line : ";
     for (int i = 0; i < argc; i++) {
         std::cout << argv[i] << ' ';
     }
     std::cout << std::endl;
     if (argc != 3) {
         std::cerr << "\n***Incorrect number of arguments***\n";
         return 1;
     }

     try {
         w9::List<w9::Description> desc(argv[1]);
         std::cout << std::endl;
         std::cout << std::setw(FWC) << "Code" <<
          std::setw(FWD) << "Description" << std::endl;
         std::cout << desc << std::endl;
         w9::List<w9::Price> price(argv[2]);
         std::cout << std::endl;
         std::cout << std::setw(FWC) << "Code" <<
          std::setw(FWP) << "Price" << std::endl;
         std::cout << price << std::endl;
         w9::List<w9::Product> priceList = merge(desc, price);
         std::cout << std::endl;
         std::cout << std::setw(FWD) << "Description" <<
          std::setw(FWP) << "Price" << std::endl;
         std::cout << priceList << std::endl;
     }
     catch (const std::string& msg) {
         std::cerr << msg << std::endl;
     }
     catch (const char* msg) {
         std::cerr << msg << std::endl;
     }

     std::cout << "\nPress any key to continue ... ";
     std::cin.get();
}
```

Output

The output from a completed version of this program should look like:

```
 Command Line : w9 Descriptions.dat Prices.dat

  Code Description
  4662    tomatoes
  4039   cucumbers
  4056    brocolli
  4067      lemons
  4068     oranges


  Code   Price
  4067    0.99
  4068    0.67
  4039    1.99
  4056    2.49


  Description   Price
    cucumbers    1.99
     brocolli    2.49
       lemons    0.99
      oranges    0.67


 Press any key to continue ...
```

Part 2 - Smart Pointers

Your second task is to introduce exception handling in the Product class.

Validate Price

Upgrade the Product class to include a validate() member function. This function throws an exception if the stored price is a negative value. The message thrown is shown in the output sample below. The exception is caught by the main() function.

Finally, upgrade your merge() function to validate the product being added to the new list. If the Product object throws an exception, your merge() function abandons building its new list and does not catch the exception.

To avoid memory leaks convert the raw pointer in your merge() function to a unique smart pointer. Also, convert the raw pointer parameter in the += operator in the List template to match.

Output

The output from a completed version of this program should look like:

```
 Command Line : w9 Descriptions.dat BadPrices.dat

  Code Description
  4662    tomatoes
  4039   cucumbers
  4056    brocolli
  4067      lemons
  4068     oranges


  Code   Price
  4067    0.99
  4068    0.67
  4039    1.99
  4056   -2.49

 *** Negative prices are invalid ***

 Press any key to continue ...
```

Submission

Typescript

On matrix, create a typescript of your complete solution using the following commands:

```
+ At the prompt, type: script w9.txt
+ At the prompt, type: whoami
+ At the prompt, type: cat w9.cpp Element.h List.h
+ At the prompt, type: g++ -o w9 w9.cpp
+ At the prompt, type: w9 Description.dat Prices.dat
+ At the prompt, type: w9 Description.dat BadPrices.dat
+ At the prompt type: exit
```

These commands will produce a file named w9.txt.

Download your typescript file to your local computer.

Moodle

- Login to
- Select OOP345 if necessary
- Select W9 under Workshops
- Upload your typescript file to Moodle
- Press "Edit"
- Summarize to your instructor the concepts that you have learned in doing this particular workshop. Add any other comments you wish to make.
- Press "Save Changes"
- When ready to submit, press "Send for Marking"

MySeneca

- Login to
- Select OOP345 if necessary
- Select Assignments or Workshops
- Select W9
- Press "Browse My Computer" to upload your typescript
- Press "Edit"
- Summarize to your instructor the concepts that you have learned in doing this particular workshop. Add any other comments you wish to make in the comment box provided.
- Press "Submit" IMPORTANT: If you "Save As Draft" your instructor does not receive your submission unitl you press "Submit"