COMP 2012 Midterm Exam - Fall 2017 - HKUST

Date: Nov 4th, 2017 (Saturday)
Time Allowed: 2 hours, 2:00 – 4:00 pm
Instructions:

1. This is a closed-book, closed-notes examination.
2. There are **7** questions on **30** pages (including this cover page, 3 appendix pages, and 3 blank pages at the end).
3. Write your answers in the space provided.
4. All programming codes in your answers must be written in the ANSI C++ version as taught in the class.
5. For programming questions, you are **NOT** allowed to define additional structures, or use global variables nor any library functions not mentioned in the questions. But you may use the STL functions given in the Appendix.

Student Name	Solution & Marking Scheme
Student ID	
Email Address	
Seat Number	

For T.A. Use Only

Problem	Topic	Score
1	True or False	/ 10
2	Const-ness	/ 10
3	Order of Construction and Destruction	/ 10
4	Class Basics	/ 15
5	STL Algorithm	/ 9
6	Class Template and Operator Overloading	/ 22
7	Classes and Objects	/ 24
Total		/ 100

Problem 1 [10 points] True or False

Indicate whether the following statements are *true* or *false* by circling **T** or **F**. You get 1.0 point for each correct answer, −0.5 for each wrong answer, and 0.0 if you do not answer.

T **F** (a) To define an inline member function for a class, you can define it in a different file than the file containing the definition.

T **F** (b) C++ compiler will NOT place a function call with the body of the called recursive function.

T **F** (c) The function parameter declarations "const Person* t" and "Person const* t" are equivalent in effect.

T **F** (d) There is NO compilation error in the following code segment.

```
class A {
  public:
    int x;
    A() { x = 1; }
};

int main() {
  A obj = { 2 };
  return 0;
}
```

T **F** (e) There is NO compilation error in the following code segment.

```
class A {
  private:
    int x;
  public:
    A() { x = 1; }
    A(const A a) { x = a.x; }
};

int main() {
  A obj1;
  A obj2(obj1);
  return 0;
}
```

T **F** (f) If constructors do not perform implicit conversions, you can create an array of objects only if the class has a default constructor.

T **F** (g) Increment operator (i.e ++) can only be implemented as member functions.

T **F** (h) A class cannot declare a member function of another class as friend.

T **F** (i) The friend declaration can be placed anywhere in the class declaration and it is not affected by the access control keywords.

T **F** (j) Class templates can have default arguments for type or value parameters. For example:

```
template <typename T = int, int size = 10>
class Array {};
```

Problem 2 [10 points] Const-ness

In the following program, for the 10 statements ending with following comment:

```
/* Error:   Yes   /   No   /   Don't know   */
```

decide whether the statement is syntactically INCORRECT - that is, it will produce compilation error(s). Circle "Yes" if it will give compilation error and "No" otherwise.
You get 1 point for each correct answer, -0.5 for each wrong answer, and 0.0 if you do not answer by circling "Don't know".

```
#include <iostream>
using namespace std;

void mystery1(int* p)       { cout << "mystery1" << endl; }
void mystery2(const int* p) { cout << "mystery2" << endl; }
void mystery3(int const* p) { cout << "mystery3" << endl; }
void mystery4(int* const p) { cout << "mystery4" << endl; }
void mystery5(int& p)       { cout << "mystery5" << endl; }
void mystery6(const int& a) { cout << "mystery6" << endl; }

void mystery7(int a, const double b) { cout << "mystery7" << endl; }
const int mystery8(int a, double b)  { cout << "mystery8" << endl; }

int main() {
  const int a(20);

  mystery1(&a);                                    /* Error:  Yes   */
  mystery2(&a);                                    /* Error:  No    */
  mystery3(&a);                                    /* Error:  No    */
  mystery4(&a);                                    /* Error:  Yes   */
  mystery5(a);                                     /* Error:  Yes   */
  mystery6(a);                                     /* Error:  No    */

  /* Hint for the following questions.
     int func(int a, const double b);
     int (*p)(int, double) = func;
     - Can you pass a double value to const double b? */

  void(*p1)(int, double) = mystery7;               /* Error:  No    */
  void(*p2)(int, const double) = mystery7;         /* Error:  No    */
  const int(*p3)(int, double) = mystery8;          /* Error:  No    */
  const int(*p4)(const int, double) = mystery8;    /* Error:  No    */

  return 0;
}
```

Problem 3 [10 points] Order of Construction and Destruction

```
#include <iostream>
using namespace std;

class A {
  public:
    A() { cout << "A's default contor\n"; }
    A(int a) { cout << "A's conversion contor\n"; }
    ~A() { cout << "A's destor\n"; }
};

class B {
  public:
    B() { cout << "B's default contor\n"; }
    B(const B& b) { cout << "B's copy contor\n"; }
    B(A a) { cout << "B's conversion contor\n"; }
    ~B() { cout << "B's destor\n"; }
};

class C {
  private:
    A* ap;   B b1, b2;  // Three data members here
  public:
    C() : b1(), b2(b1) { ap = new A; cout << "C's default contor\n"; }
    ~C() { cout << "C's destor\n"; delete ap; }
};

class D {
  private:
    A a;   B b;   C* cp;  // Three data members here
  public:
    D(): a(10), cp(new C) { b = B(); cout << "D's default contor\n";
}
    ~D() { delete cp; }
};

int main() {
  cout << "--- Construct B object ---\n";
  B obj1(10);
  cout << "--- Construct D object ---\n";
  D obj2;
  cout << "--- Before return ---\n";
  return 0;
}
```

What are the outputs of the above program? Write the outputs below. Assume the compiler DOES NOT do any optimization.

Answer:

```
--- Construct B object ---
A's conversion contor
B's conversion contor
A's destor
A's destor
--- Construct D object ---
A's conversion contor
B's default contor
B's default contor
B's copy contor
A's default contor
C's default contor
B's default contor
B's destor
D's default contor
--- Before return ---
C's destor
A's destor
B's destor
B's destor
B's destor
A's destor
B's destor
```

Marking scheme:
0.5 point for each of the 20 output lines.
Note:

- 0.25 point for the presence of a correct term.
- 0.25 point for the correct relative position.
- -0.25 point for each wrong insertion.
- -0.25 point for each extra incorrect statement.
- Minimum score for each part is zero.

Problem 4 [15 points] Class Basics

The following program contains 6 ERRORS (syntax error, logical errors - including uninitialized data member, etc.). Study the program carefully, identify all the errors by writing down the line number where an error occurs, and explain why it is an error.

```
1  #include <iostream>
2  using namespace std;
3
4  class A {
5    private:
6      int a;
7      int b;
8      int* p = NULL;
9      const double PI;
10     int& ref;
11
12   public:
13     A(int a, int bb, int c) : a(a), PI(3.14159), ref(c) {
14       int b = bb;
15       p = new int[10];
16     }
17
18     A(int a, int b) : PI(3.14159) {
19       this->a = a;
20       this->b = b;
21       p = new int;
22     }
23
24     ~A() {
25       delete [] p;
26     }
27 };
28
29 int main() {
30   A** p = new A*[3];
31
32   A obj1;
33   A obj2(1, 2, 3);
34   A obj3(4, 5);
35   p[0] = &obj1;
36   p[1] = &obj2;
37   p[2] = &obj3;
38
39   return 0;
40 }
```

Error#	Line#	Explanation
1	8	Cannot initialize data member at its declaration.
2	13	Should not reference to local variable c.
3	14	Without initializing data member b (initialize local variable b in this case).
4	18	Without initializing reference variable ref.
5	15 / 21 / 25	Inconsistent between dynamic member allocation and de-allocation.
6	32	Object cannot be instantiated because default constructor is missing.
7	38	Without de-allocate the array of pointers using delete.

Marking scheme:

You are only required to identify any 6 errors.

0.5 for each correct line number. 2 points per correct explanation.

Problem 5 [9 points] STL Algorithm

(a) [3 points] Given the following prototype of the STL `max_element` algorithm,

```
template <typename ForwardIterator>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last);
```

Parameters:

- `first`, `last`: Input iterators to the initial and final positions of the sequence to compare.

Implement the algorithm so that it returns an iterator pointing to the element with the largest value in the range [first,last) by comparing elements using `operator<`. An element is largest if no other element does not compare less than it. If more than one element fulfills this condition, the iterator returned points to the first of such elements.

Solution:

```
template<typename ForwardIterator>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last) {
  if(first == last)                // 0.5 point
    return last;                   // 0.5 point
  ForwardIterator largest = first;
  while(++first != last)           // 0.5 point
    if(*largest < *first)          // 0.5 point
      largest = first;             // 0.5 point
  return largest;                  // 0.5 point
}
```

(b) [4 points] Write a class called **Student** which will be put in a file called "Student.h" that has the following:

- A private data member, **name**, of type string
- A private data member, **cgpa**, of type double
- A constructor
  It initializes its objects with arguments name and cgpa
- A public const member function, **getName**
  It returns the data member name.
- A public const member function, **getCGPA**
  It returns the data member cgpa.
- An overloaded operator function, **operator<**
  It accepts an object of Student type as argument and it checks whether the cgpa of the given object is larger than its cgpa or not. If so, return true, otherwise return false.

Solution:

```
#ifndef STUDENT_H
#define STUDENT_H

#include <string>
using namespace std;

class Student {     // 0.5 point
  private:
    string name;    // 0.5 point
    double cgpa;    // 0.5 point
  public:
    Student(string name, double cgpa) {        // 0.5 point
      this->name = name;
      this->cgpa = cgpa;
    }
    string getName() const { return name; }    // 0.5 point
    double getCGPA() const { return cgpa; }    // 0.5 point
    bool operator<(const Student& s) const {   // 0.5 point
      return (s.cgpa > cgpa) ? true : false;   // 0.5 point
    }
};

#endif /* STUDENT_H */
```

(c) [2 points] Complete the following program in the space provided after the "**TO DO**" comment lines so that it will run and give the output below:

```
Student with the highest CGPA: Wallace (4.25)
```

Solution:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include "Student.h"
using namespace std;

int main()
{
  vector<Student> students;
  students.push_back(Student("Ken", 3.2));
  students.push_back(Student("Wallace", 4.25));
  students.push_back(Student("John", 2.1));

  /*
   * TO-DO: Using max_element from part (a), find the student with the
   * highest CGPA stored in the STL vector container, students.
   *
   ********************** CODE BEGINS **********************/

   // 1 point for vector<Student>::iterator it
   // 0.5 point for students.begin()
   // 0.5 polnt for students.end()

   vector<Student>::iterator it = ::max_element(students.begin(), students.end());


   /********************** CODE ENDS **********************/

  cout << "Student with the highest CGPA: " << (*it).getName()
                                            << " (" << (*it).getCGPA()
                                            <<  ")" << endl;
  return 0;
}
```

Problem 6 [22 points] Class Template and Operator Overloading

```
#include <iostream>   /* File "SpecialContainer.h" */
#include <list>
#include <algorithm>
using namespace std;

template <typename T>
class SpecialContainer {
  private:
    list<T> s;   // A list container that stores elements

  public:
    typedef typename list<T>::iterator listiterator;

    // Construct an SpecialContainer object and add all the elements of arr to s.
    SpecialContainer(const T arr[], int num);   // Conversion constructor

    // Create a new SpecialContainer with a copy of all the elements in s, placed them
    // at the front and element e at the back. Return the new SpecialContainer.
    SpecialContainer operator+(const T& e);

    // Find element e in the container s.
    // If it exists, create a new SpecialContainer with a copy of all the elements in s
    // except e and return it. Otherwise, return the current SpecialContainer.
    SpecialContainer operator-(const T& e);

    // Add element e to the current container and return it back.
    SpecialContainer& operator+=(const T& e);
.
    // Remove element e from the current container and return it back.
    SpecialContainer& operator-=(const T& e);

    // Shift n elements to the left of the list.
    // For example: s = { 1, 2, 3 }, n = 2, list s would change to { 3, 1, 2 }.
    void operator<<(int n);

    // Shift n elements to the right of the list.
    // For example: s = { 1, 2, 3 }, n = 2, list s would change to { 2, 3, 1 }.
    void operator>>(int n);

    // If i is in the legal index range of container, return the ith element.
    // Otherwise, output "op[]: invalid dimension!" and terminate the program.
    T& operator[](int i);

    // Overload the insertion operator<< for the SpecialContainer class
    // Note: Define listiterator in your function with
    // "typedef typename list<T>::iterator listiterator;" and use it.
    template <typename S>
      friend ostream& operator<<(/* complete the prototype as well */);
};

#include "SpecialContainer.cpp"
```

Implement the 9 undefined member functions of the above template class '`SpecialContainer`' in "`SpecialContainer.cpp`" so that they will work with the testing program below to produce the following output.

```
#include "SpecialContainer.h"     /* File: "test-specialcontainer.cpp" */

int main() {
  int arr[] = { 10, 20, 50, 5 };
  SpecialContainer<int> container(arr, 4);
  container += 60;
  container += 16;
  cout << "Original: \t\t\t\t" << container;

  container << 3;
  cout << "After left shifting by 3:\t\t" << container;
  container >> 4;
  cout << "After right shifting by 4:\t\t" << container;
  container >> 101;
  cout << "After right shifting by 101:\t\t" << container;

  container -= 30;
  cout << "After removing 30:\t\t\t" << container;
  container -= 20;
  cout << "After removing 20:\t\t\t" << container;

  container[2] = 99;
  cout << "After changing container[2] to 99:\t" << container;
  container[10] = 99;
  cout << "After changing container[10] to 111:\t" << container;

  return 0;
}
```

Output of the program:

```
Original:                              10, 20, 50, 5, 60, 16
After left shifting by 3:              5, 60, 16, 10, 20, 50
After right shifting by 4:             16, 10, 20, 50, 5, 60
After right shifting by 101:           10, 20, 50, 5, 60, 16
After removing 30:                     10, 20, 50, 5, 60, 16
After removing 20:                     10, 50, 5, 60, 16
After changing container[2] to 99:     10, 50, 99, 60, 16
op[]: invalid dimension!
```

Hint: You may use previously defined overloaded operator functions to define other ones. Also, you **must** use `listiterator` defined in the class in your implementation.

Solution: /* File: "SpecialContainer.cpp" */

```
template <typename T>
SpecialContainer<T>::SpecialContainer(const T arr[], int num) {
  for(int i=0; i<num; i++)                    // 0.5 point
    *this += arr[i];                          // 1 point
}

template <typename T>
SpecialContainer<T> SpecialContainer<T>::operator+(const T& e) {
  SpecialContainer<T> newContainer(*this);    // 0.5 point
  newContainer.s.push_back(e);                // 0.5 point
  return newContainer;                        // 0.5 point
}

template <typename T>
SpecialContainer<T> SpecialContainer<T>::operator-(const T& e) {
  SpecialContainer<T> newContainer(*this);    // 0.5 point
  listiterator p = find(newContainer.s.begin(), newContainer.s.end(), e);  // 1.5 points
  if(p != newContainer.s.end())               // 0.5 point
    newContainer.s.erase(p);                  // 0.5 point
  return newContainer;                        // 0.5 point
}

template <typename T>
SpecialContainer<T>& SpecialContainer<T>::operator+=(const T& e) {
  return (*this = *this + e);                 // 1 point
}

template <typename T>
SpecialContainer<T>& SpecialContainer<T>::operator-=(const T& e) {
  return (*this = *this - e);                 // 1 point
}

template <typename T>
void SpecialContainer<T>::operator<<(int n) {
  int amount = n % s.size();                  // 0.5 point
  for(int i=0; i<amount; i++) {               // 0.5 point
    T element = *(s.begin());                 // 0.5 point
    s.pop_front();                            // 0.5 point
    s.push_back(element);                     // 0.5 point
  }
}
```

```
template <typename T>
void SpecialContainer<T>::operator>>(int n) {
  int amount = n % s.size();                    // 0.5 point
  for(int i=0; i<amount; i++) {                 // 0.5 point
    listiterator cur = s.end();                 // 0.5 point
    T element = *(--cur);                       // 1 point
    s.pop_back();                               // 0.5 point
    s.push_front(element);                      // 0.5 point
  }
}

template <typename T>
T& SpecialContainer<T>::operator[](int i) {
  if(i < s.size()) {                            // 0.5 point
    listiterator it = s.begin();                // 0.5 point
    advance(it, i);                             // 0.5 point
    return *it;                                 // 0.5 point
  }
  cerr << "op[]: invalid dimension!" << endl;   // 0.5 point
  exit(1);                                      // 0.5 point
  exit(1);                                      // 0.5 point
}

template <typename T>
ostream& operator<<(ostream& os, SpecialContainer<T>& container) {  // 0.5 point
  typename list<T>::iterator cur = container.s.begin();             // 1 point
  for(int i=0; i<container.s.size()-1; i++) {                       // 0.5 point
    os << *cur << ", ";                                             // 0.5 point
    ++cur;                                                          // 0.5 point
  }
  os << *cur << endl;                                               // 0.5 point
  return os;                                                        // 0.5 point
}
```

Note: 1 point will be deducted for EACH of the following cases.

- Missing `template <typename T>` several times.
- Missing `<T>` after `SpecialContainer`.

Problem 7 [24 points] Classes and Objects

(a) [2 points] The following shows a typical class definition for a Book. Complete the missing parts in the space provided under Part(a)(i)-(a)(ii) "`ADD YOUR CODE HERE`" by declaring (1) the non-member function, `operator<<` as a friend of Book, (2) declaring equality operator `operator==` for the Book class.

Solution:

```
#ifndef BOOK_H   /* Book.h */
#define BOOK_H

#include <iostream>
#include <string>
using namespace std;

class Book {
    friend class ShoppingCart;

    // Make the non-member function, operator<< a friend of Book class
    // Part (a)(i) - ADD YOUR CODE HERE

    friend ostream& operator<<(ostream&, const Book&);  // 1 point

  private:
    string title;  // Book title
    string isbn;   // ISBN number of the book
    double price;  // Price of the book

  public:
    Book(string title = "", string isbn = "", double price = 0.0);

    // Declare equality operator, operator==, for the Book class
    // Part (a)(ii) - ADD YOUR CODE HERE

    bool operator==(const Book& book);                  // 1 point

};

#endif /* BOOK_H */
```

(b) [4 points] Provide the implementation of the overloaded operators, (1) `operator==` and (2) `operator<<` for Book in the space provided under Part(b)(i)-(b)(ii) "`ADD YOUR CODE HERE`".

- Assume the result of equality test between Books is based on all the data members. For equality test of price, you can take absolute value of the price difference between two Book objects. If the value is less than the value returned by `numeric_limits<double>::epsilon()`, return true. Otherwise, return false.
- The output format of a Book object is as follows:

  `<book title>, <isbn number>, $<price>`

  The following shows a sample output of a book:

  `C++ How to Program, 978-0134448237, $325`

Solution:

```
#include <cmath>   /* File "Book.cpp" */
#include <limits>
#include "Book.h"
using namespace std;

Book::Book(string title, string isbn, double price) {
  this->title = title;
  this->isbn = isbn;
  this->price = price;
}

// Implement the member function operator==
// Part (b)(i) - ADD YOUR CODE HERE

bool Book::operator==(const Book& book) {
  return (book.title == title && book.isbn == isbn &&
          abs(book.price - price) < numeric_limits<double>::epsilon());

  // 0.5 point for comparing book title
  // 0.5 point for comparing isbn
  // 1.5 points for comparing book price
}

// Implement the non-member function operator<<
// Part (b)(ii) - ADD YOUR CODE HERE

ostream& operator<<(ostream& os, const Book& b) {
  return (os << b.title << ", " << b.isbn << ", $" << b.price); // 1.5 point
}
```

(c) [18 points] Now suppose you need to develop an e-Commerce shopping cart using the Book class defined in part (a) and (b). To accomplish the task, a C++ class named `ShoppingCart` has been defined for you as follows:

```
#ifndef SHOPPINGCART_H   /* ShoppingCart.h */
#define SHOPPINGCART_H

#include "Book.h"
using namespace std;

class ShoppingCart {
  private:
    Book** books;   // An array of pointers to books
    int capacity;   // The capacity of the pointer array
    int numBooks;   // The number of book objects pointed by the book array

  public:
    ShoppingCart();                         // Default constructor
    ShoppingCart(const ShoppingCart& sc); // Copy constructor
    ~ShoppingCart();                        // Destructor

    // Add a book to the shopping cart
    // Note: The book object is passed by value
    void addBook(Book book);

    // Remove a book from the shopping cart
    // Note: The book object is passed by value
    bool removeBook(Book book);

    double payAmount() const;   // The pay amount of all the books

    // Implement operator= in a way that it can handle self-assignment properly
    ShoppingCart& operator=(const ShoppingCart& cart);
};

#endif /* SHOPPINGCART_H */
```

Assume the member functions

```
ShoppingCart()
~ShoppingCart()
double payAmount() const
```

have been implemented in "`ShoppingCart.cpp`" as shown on the next page.

```
#include "ShoppingCart.h"

ShoppingCart::ShoppingCart() {
  books = new Book*[5];
  for(int i=0; i<5; i++)
    books[i] = NULL;
  capacity = 5;
  numBooks = 0;
}

ShoppingCart::~ShoppingCart() {
  for(int i=0; i<numBooks; i++)
    delete books[i];
  delete [] books;
}

double ShoppingCart::payAmount() const {
  double total = 0;
  for(int i=0; i<numBooks; i++)
    total += books[i]->price;
  return total;
}

// *** 4 missing member functions: ***
// - ShoppingCart(const ShoppingCart& sc)
// - void addBook(Book book)
// - bool removeBook(Book book)
// - ShoppingCart& operator=(const ShoppingCart& cart)
```

Your task is to complete the remaining 4 member functions that satisfy the following requirements:

- Perform **deep copy** in the copy constructor and assignment operator function, i.e. `operator=`.
- For `addBook`, duplicate the book object and link it to the array. If the number of books in the shopping cart reaches the capacity of the shopping cart, replace the original array with a new array of size doubling the original capacity and move all the items over.
- For `removeBook`, check whether the book is in the shopping cart. If not, return false. Otherwise, remove the book, move all the books backwards, decrease the number of books in the shopping cart by 1, and return true.
- The given testing program "`main.cpp`" will compile, run, and produce the output on the next page.

```
#include "Book.h"
#include "ShoppingCart.h"

int main() {
  ShoppingCart cart1;
  Book book1("C++ How to Program", "978-0134448237", 325);
  Book book2("Introduction to Java Programming", "978-0132936521", 350);
  Book book3("Harry Potter and the Philosopher's Stone", "978-0747549550", 420);
  cout << book1 << endl << book2 << endl << book3 << endl;

  cart1.addBook(book1);
  cart1.addBook(book2);
  cart1.addBook(book3);
  cout << "Pay amount for all books in cart1: $" << cart1.payAmount() << endl;

  ShoppingCart cart2(cart1);
  ShoppingCart cart3;
  cart3 = cart1;

  cart1.removeBook(book2);
  cout << "After removeBook, pay amount for all books in cart1: "
       << cart1.payAmount() << endl;

  cout << "Pay amount for all books in cart2: $" << cart2.payAmount() << endl;
  cout << "Pay amount for all books in cart3: $" << cart3.payAmount() << endl;

  return 0;
}
```

Output of the testing program:

```
C++ How to Program, 978-0134448237, $325
Introduction to Java Programming, 978-0132936521, $350
Harry Potter and the Philosopher's Stone, 978-0747549550, $420
Pay amount for all books in cart1: $1095
After removeBook, pay amount for all books in cart1: 745
Pay amount for all books in cart2: $1095
Pay amount for all books in cart3: $1095
```

Solution: /* File: "ShoppingCart.cpp" */

Implement all the missing member functions of the class `ShoppingCart` here:

```
ShoppingCart::ShoppingCart(const ShoppingCart& sc) {
  numBooks = 0;                                  // 0.25 point
  books = NULL;                                  // 0.25 point
  *this = sc;                                    // 0.5 point
}

void ShoppingCart::addBook(Book book) {
  if(numBooks >= capacity) {                     // 0.5 point
    Book** temp = new Book*[capacity*2];         // 0.5 point
    capacity *= 2;                               // 0.5 point
    for(int i=0; i<numBooks; i++)                // 0.5 point
      temp[i] = books[i];                        // 0.5 point
    delete [] books;                             // 0.5 point
    books = temp;                                // 0.5 point
  }
  books[numBooks++] = new Book(book);            // 1.5 point
}

bool ShoppingCart::removeBook(Book book) {
  int foundIndex = -1;                           // 0.5 point
  for(int i=0; i<numBooks; i++) {                // 0.5 point
    if(*(books[i]) == book) {                    // 1 point
      foundIndex = i;                            // 0.5 point
      break;                                     // 0.5 point
    }
  }
  if(foundIndex != -1) {                         // 0.5 point
    delete books[foundIndex];                    // 0.5 point
    for(int i=foundIndex; i<numBooks-1; i++)     // 0.5 point
      books[i] = books[i+1];                     // 0.5 point
    --numBooks;                                  // 0.5 point
    return true;                                 // 0.5 point for return value
  }
  return false;
}
```

```
ShoppingCart& ShoppingCart::operator=(const ShoppingCart& cart) {
  if(this != &cart) {                          // 0.5 point
    for(int i=0; i<numBooks; i++)              // 0.5 point
      delete books[i];                         // 0.5 point
    delete [] books;                           // 0.5 point
    books = new Book*[cart.capacity];          // 1 point
    for(int i=0; i<cart.numBooks; i++) {       // 0.5 point
      books[i] = new Book;                     // 0.5 point
      *(books[i]) = *(cart.books[i]);          // 1 point
    }
    numBooks = cart.numBooks;                  // 0.25 point
    capacity = cart.capacity;                  // 0.25 point
  }
  return *this;                                // 0.5 point
}
```

/*** Continue Your Answer For Problem 7 On This Page ***/

/*** Continue Your Answer For Problem 7 On This Page ***/

---------- END OF PAPER ----------

Appendix

(1) STL Sequence Container: Vector

```
template <class T, class Alloc = allocator<T> > class vector;
```
Defined in the standard header **vector**.

Description:

Vectors are sequence containers representing arrays that can change in size. Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.

Some of the member functions of the vector<T> container class where T is the type of data stored in the vector are listed below.

Member function	**Description**
vector( )	Default constructor
iterator begin( ) const_iterator begin( ) const	Returns an iterator pointing to the first element in the vector. If the vector object is const-qualified, the function returns a const_iterator. Otherwise, it returns iterator.
iterator end( ) const_iterator end( ) const	Returns an iterator referring to the past-the-end element in the vector container. If the vector object is const-qualified, the function returns a const_iterator. Otherwise, it returns iterator.
void push_back(const T& val)	Adds a new element, val, at the end of the vector, after its current last element. The content of val is copied (or moved) to the new element.

(2) STL Sequence Container: list

```
template <class T, class Alloc = allocator<T> > class list;
```

Defined in the standard header **list**.

Description:

Lists are sequence containers that are implemented as doubly-linked lists and allow insert and erase operations anywhere within the sequence.

Some of the member functions of the list<T> container class where T is the type of data stored in the list are listed below.

Member function	**Description**
list( )	Default constructor
iterator begin( ) const_iterator begin( ) const	Returns an iterator pointing to the first element in the list. If the list object is const-qualified, the function returns a const_iterator. Otherwise, it returns iterator.
iterator end( ) const_iterator end( ) const	Returns an iterator referring to the past-the-end element in the list container. If the list object is const-qualified, the function returns a const_iterator. Otherwise, it returns iterator.
void push_front(const T& val)	Adds a new element, val, at the beginning of the list, before its current first element. The content of val is copied (or moved) to the new element.
void push_back(const T& val)	Adds a new element, val, at the end of the list, after its current last element. The content of val is copied (or moved) to the new element.
void pop_front()	Removes the first element in the list container, effectively reducing its size by one.
void pop_back()	Removes the last element in the list container, effectively reducing its size by one.
iterator erase(iterator position)	Removes from the list container a single element at position pointing by an iterator, position. It returns an iterator pointing to the element that followed the last element erased by the function call.

(3) STL's **find** function which returns the first element in the range [*first*, *last*) that is equal to *value*.

```
template <typename InputIt, typename T>
InputIt find(InputIt first, InputIt last, const T& value);
```

(4) STL's **advance** function advances the iterator *it* by *n* element positions.

```
template <typename InputIt, typename Distance>
void advance(InputIt& it, Distance n);
```

/* Rough work */

/* Rough work */

/* Rough work */