# COMP 2012H Final Exam - Fall 2021 - HKUST

Date: December 8, 2021 (Wednesday)
Time Allowed: 3 hours, 12:30–3:30 pm
Instructions:

1. This is a closed-book, closed-notes examination.
2. Electronic devices (except approved calculators) are NOT allowed.
3. There are **7** questions on **38** pages (including this cover page, appendix pages and 2 blank pages at the end). You may detach the appendix and blank pages from the exam paper if you wish.
4. Write your answers in the space provided in black/blue ink. NO pencil please, otherwise you are not allowed to appeal for any grading disagreements.
5. All programming codes in your answers must be written in the ANSI C++ version as taught in the class.
6. For programming questions, unless otherwise stated, you are **NOT** allowed to define additional structures, classes, helper functions and use global variables, **auto**, nor any library functions not mentioned in the questions.

| | |
|---|---|
| Student Name | SOLUTIONS & MARKING SCHEME |
| Student ID | |
| Email Address | |
| Seat Number | |

For T.A. Use Only

| Problem | Topic | Score |
|---|---|---|
| 1 | True/False Questions | / 10 |
| 2 | Stack and Recursion | / 9 |
| 3 | Order of Constructions and Destructions | / 6 |
| 4 | STL and Function Object | / 12 |
| 5 | Hashing (Double Hashing) | / 8 |
| 6 | Inheritance, Polymorphism and Operator Overloading | / 26 |
| 7 | Binary Search Tree With Duplicates | / 29 |
| | Total | / 100 |

**Problem 1 [10 points] True/False Questions**

Indicate whether the following statements are true or false by circling **T** or **F**. You get 1.0 point for each correct answer.

**T** **F** (a) The following program will NOT give any compilation errors or warnings.

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

enum COLOR { BLACK, WHITE, RED, ORANGE, YELLOW };
enum PRIMARY_COLOR { RED, GREEN, BLUE };

int main() {
  cout << WHITE << ", " << RED << ", " << BLACK << endl;
}
```

**T** **F** (b) Assume we define an array:

```
int arr[3];
```

the expressions, `arr`, `&arr`, and `&arr[0]` are exactly the same in terms of type and value.

**T** **F** (c) The following program will NOT give any compilation errors.

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

int main() {
  int* const* const p = new int* const(new int);
  **p = 10;
}
```

**T** **F** (d) The following program will NOT give any compilation errors or warnings.

```
int*& func() {
  int* p = new int;
  return p;
}
int main() {
  int*& q = func();
  delete q;
}
```

**T** **F** (e) The following program does NOT print `Default ctor`.

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

class A {
  public:
    A() { cout << "Default ctor"; }
};

int main() {
  A obj(A());
}
```

**T** **F** (f) The following program prints `ABCcopyAcopyB`.

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

class A {
  public:
    A() { cout << "A"; }
    A(const A& a) { cout << "copyA"; }
};

class B {
  public:
    B() { cout << "B"; }
    B(const B& b) { cout << "copyB"; }
};

class C {
  A a;
  B b;
  public:
    C() { cout << "C"; }
    C(const C& c) = default;
};

int main() {
  C cObj1;
  C cObj2(cObj1);
}
```

**T** **F** (g) The following program prints `Specialized`.

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

template <typename T, int N>
void func(T(&arr)[N]) {
  cout << "General";
}

template <typename T, int N>
void func(int(&arr)[5]) {
  cout << "Specialized";
}

int main() {
  int arr[5];
  func(arr);
}
```

**T** **F** (h) It is possible to overload the subscript operator, operator[ ], that accepts multiple arguments.

**T** **F** (i) The following program will NOT give any compilation errors.

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

bool greater_than_10(int value) { return (value > 10); }

int main() {
  int arr[] { 2, 6, 8, 14 };
  int* p = find_if(&arr[0], &arr[3]+1, greater_than_10);
  cout << *p << endl;
}
```

**T** **F** (j) Suppose the size of a hash table is prime, and quadratic probing is used to resolve collisions. We can always find an empty cell if the table is at most half full.

**Problem 2 [9 points] Stack and Recursion**

Stack is a very useful data structure used in many applications. In some cases, we may need to sort the items in the stack while maintaining the Last-In-First-Out property. In this question, you are required to sort the elements in a given STL stack in ascending order using recursion. A skeleton code of the program ("`sort-stack.cpp`") that can be used to sort the elements in a stack in ascending order is given below. This program has two undefined global functions, `sortedInsert` and `sortStack`.

```
#include <iostream>  /* File: sort-stack.cpp */
#include <stack>
using namespace std;

// GIVEN: A global function to print all the elements in stack s
void printStack(stack<double> s) {
  cout << "[ ";
  while(!s.empty()) {
    double n = s.top();
    cout << n << " ";
    s.pop();
  }
  cout << "]" << endl;
}

// Goal: Given a stack s where the elements in it are sorted in ascending order,
// and a given element x, insert x to s such that the elements in s remain sorted.
void sortedInsert(stack<double>& s, double x) {
  // TODO: Part(a)
}

// Goal: Given a stack s where the elements in it may not be sorted in
// ascending order. Sort the elements in s in ascending order.
// *** Hint: Make good use of sortedInsert function ***
void sortStack(stack<double>& s) {
  // TODO: Part(b)
}

int main() {
  // Define a stack container s and push 5 double type values onto s
  stack<double> s;
  s.push(125.2);  s.push(12.5);  s.push(52.6);
  s.push(29.5);   s.push(113.6);

  cout << "Before sorting: ";
  printStack(s); // Print the elements in stack s
  sortStack(s);  // Sort stack s
  cout << "After sorting: ";
  printStack(s); // Print the elements in stack s again
}
```

When the program compiles and runs, it produces the following output.

```
Before sorting: [ 113.6 29.5 52.6 12.5 125.2 ]
After sorting: [ 12.5 29.5 52.6 113.6 125.2 ]
```

Note: You may refer to the Appendix for the use of STL stack container adaptor.

(a) [5 points] Implement `sortedInsert` as a recursive function by completing the missing part in `TODO: Part(a)` of the given skeleton code.

**Answer:**

```
void sortedInsert(stack<double>& s, double x) {
  if(s.empty() || x < s.top()) // 1.5 points
    s.push(x);                 // 0.5 point
  else {
    double temp = s.top();     // 0.5 point
    s.pop();                   // 0.5 point
    sortedInsert(s, x);        // 1.5 points
    s.push(temp);              // 0.5 point
  }
}
```

0 point is given for any non-recursive implementation.

(b) [4 points] Implement `sortStack` as a recursive function by completing the missing part in `TODO: Part(b)` of the given skeleton code.

**Answer:**

```
void sortStack(stack<double>& s) {
  if(!s.empty()) {             // 0.5 point
    double x = s.top();        // 1 point
    s.pop();                   // 0.5 point
    sortStack(s);              // 1 point
    sortedInsert(s, x);        // 1 point
  }
}
```

0 point is given for any non-recursive implementation.

**Problem 3 [6 points] Order of Constructions and Destructions**

Given the following program:

```
#include <iostream>    /* File: order-constructions-destructions.cpp */
using namespace std;

class A {
  public:
    A(int a) { cout << "A's conv" << endl; }
    A(const A& a) { cout << "A's copy" << endl; }
    ~A() { cout << "A's dtor" << endl; }
};

class B {
  public:
    B() { cout << "B's default" << endl; }
    B(int b) { cout << "B's conv" << endl; }
    B(const B& b) { cout << "B's copy" << endl; }
    ~B() { cout << "B's dtor" << endl; }
};

class C {
  public:
    C() { cout << "C's default" << endl; }
    C(const C& c) { cout << "C's copy" << endl; }
    ~C() { cout << "C's dtor" << endl; }
};

class D : public C {
    B objB;
    A* ptrA = nullptr;
  public:
    D() : objB(5) {
      cout << "D's default" << endl;
    }
    D(int a, int b) : ptrA(new A(a)), objB(b) {
      cout << "D's other" << endl;
    }
    D(const D& d) : objB(d.objB), ptrA(new A(*(d.ptrA))), C(d) {
      cout << "D's copy" << endl;
    }
    ~D() {
      cout << "D's dtor" << endl;
      delete ptrA;
      cout << "After delete" << endl;
    }
};
```

```
D mystery() {
  D d(1, 2);
  return d;
}

int main() {
  D d1;
  cout << "---- Before mystery ----" << endl;
  const D& d2 = mystery();
  cout << "---- After mystery ----" << endl;
}
```

Assume the program is compiled using the command:
`g++ -std=c++11 -fno-elide-constructors order-constructions-destructions.cpp`
which disables the return value optimization (RVO). Write down the output of the program.

**Answer:**

**Answer:**

```
C's default
B's conv
D's default
---- Before mystery ----
C's default
B's conv
A's conv
D's other
C's copy
B's copy
A's copy
D's copy
D's dtor
A's dtor
After delete
B's dtor
C's dtor
---- After mystery ----
D's dtor
A's dtor
After delete
B's dtor
C's dtor
D's dtor
After delete
B's dtor
C's dtor
```

**Marking scheme:**

This question is marked in sections distributed as follows:

`Section 1: Beginning to "---- Before mystery ----"`

`Section 2: "---- Before mystery ----" to "---- After mystery ----"`

`Section 3: "---- After mystery ----" to the end`

- With each section, the sequence with the longest correct statement (except those marked with `--- ... ---`) count is selected for marking and each statement in the sequence is worth 0.24 point.
- No penalty is given for extra lines of code beyond the accepted sequence of statements.

**Problem 4 [12 points] STL and Function Object**

(a) [4 points] Given the following prototype of the `copy_if` algorithm,

```
template <typename Iterator, typename Predicate>
int copy_if(Iterator sfirst, Iterator slast, Iterator dfirst, Predicate predicate);
```

implement the algorithm, which will be put in a file called "`copy_if.h`", so that it

- copies elements in the range `[sfirst, slast)` (i.e., starting from `sfirst` and proceeding to `slast` - 1) for which `predicate` returns true to the range beginning at `dfirst`;
- returns the number of elements copied.

**Answer:**

```
template <typename Iterator, typename Predicate>
int copy_if(Iterator sfirst, Iterator slast, Iterator dfirst, Predicate predicate) {
  int count = 0;              // 0.5 point
  while(sfirst != slast) {    // 0.5 point
    if(predicate(*sfirst)) {  // 0.5 point
      *dfirst++ = *sfirst;    // 1 point
      ++count;                // 0.5 point
    }
    sfirst++;                 // 0.5 point
  }
  return count;               // 0.5 point
}
```

(b) [6 points] Write a function object class called '`IsPrimeGreaterThan`', which will be put in a file "`is_prime_greater_than.h`" that has the following:

- a private data member, **threshold**, of type `unsigned int`
- a constructor
- an overloaded parenthesis operator function

The constructor of the **IsPrimeGreaterThan** class should initialize its function objects with a threshold so that when a function object of the class is called with a positive value, it will check whether the given value is a prime number and is greater than the threshold. If so, it returns true. Otherwise, returns false.

Hint: A number x is prime if x mod i is not 0, for all i between 2 (inclusive) and $x/2$ (inclusive).

**Answer:**

```
class IsPrimeGreaterThan {                          // 0.5 point
  private:
    unsigned int threshold;                         // 0.5 point
  public:
    IsPrimeGreaterThan(unsigned int threshold)      // 0.5 point
       : threshold(threshold) {}                    // 0.5 point
    bool operator()(unsigned int value) {           // 0.5 point
      if (value == 0 || value == 1)                 // 0.5 point
        return false;                               // 0.25 point
      else {
        for (int i = 2; i <= value/2; ++i)          // 1 point
          if (value % i == 0)                       // 0.5 point
            return false;                           // 0.25 point
      }
      if(value > threshold)                         // 0.5 point
        return true;                                // 0.25 point
      return false;                                 // 0.25 point
    }
};
```

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

```
7
11
```

```
#include <iostream>
#include <vector>
#include "copy_if.h" // With the definition of copy_if
#include "is_prime_greater_than.h" // With IsPrimeGreaterThan class
using namespace std;

int main() {
  // You may refer to the Appendix for the use of STL vector container.
  vector<unsigned int> v { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
  vector<unsigned int> prime(v.size());

  /*
     TODO: Using copy_if from part(a) together with a function object
     of type IsPrimeGreaterThan defined in part (b), copy all prime
     numbers that are greater than 6 in the STL vector container v
     to the STL vector container "prime".

     *************** CODE BEGINS ***************




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

  prime.resize(count);
  for(int i=0; i<prime.size(); ++i)
    cout << prime[i] << endl;
}
```

**Answer:**

```
int count = copy_if(v.begin(), v.end(), prime.begin(), IsPrimeGreaterThan(6));
// 0.25 for v.begin()
// 0.25 for v.end()
// 0.25 for prime.begin()
// 1 for IsPrimeGreaterThan(6)
// 0.25 for int count
```

**Problem 5 [8 points] Hashing (Double Hashing)**

A hash table of size 11 with the following hash function is used to store keys.

$$hash(k) = k \ mod \ 11$$

Insert the following keys:

8, 19, 40, 18, 31, 109, 52, 78

in the given order (from left to right) to the hash table using double hashing to resolve collisions. Assume the following second hash function is applied to the key when a collision occurs.

$$hash_2(k) = 7 - (k \ mod \ 7)$$

Also, compute the total number of collisions for hashing the eight keys.

Note that once a key gets into the hash table, it stays in the table. To illustrate that, the insertion of the first key 8 has been done for you in the table below.

| Index | Insert 8 | Insert 19 | Insert 40 | Insert 18 | Insert 31 | Insert 109 | Insert 52 | Insert 78 |
|---|---|---|---|---|---|---|---|---|
| 0 | | | | | | | | |
| 1 | | | | | | | | |
| 2 | | | | | | | | |
| 3 | | | | | | | | |
| 4 | | | | | | | | |
| 5 | | | | | | | | |
| 6 | | | | | | | | |
| 7 | | | | | | | | |
| 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 |
| 9 | | | | | | | | |
| 10 | | | | | | | | |

Total number of collisions for hashing the eight keys using double hashing: _____

**Answer:**

| Index | Insert 8 | Insert 19 | Insert 40 | Insert 18 | Insert 31 | Insert 109 | Insert 52 | Insert 78 |
|---|---|---|---|---|---|---|---|---|
| 0 | | | | | | | | |
| 1 | | | | | | | 52 | 52 |
| 2 | | | | 18 | 18 | 18 | 18 | 18 |
| 3 | | | | | | | | 78 |
| 4 | | | | | | | | |
| 5 | | | | | | 109 | 109 | 109 |
| 6 | | | | | | | | |
| 7 | | | 40 | 40 | 40 | 40 | 40 | 40 |
| 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 |
| 9 | | | | | 31 | 31 | 31 | 31 |
| 10 | | 19 | 19 | 19 | 19 | 19 | 19 | 19 |

Total number of collisions for hashing the eight keys using double hashing: 10

**Marking scheme:**

- Each correct insertion gives 1 point. 7 points in total. If a student makes a mistake at one insertion but the next insertions are "consistent" with this mistake, then they should only lose one point for the original mistake.
- 1 point for giving number of collisions.

**Problem 6 [26 points] Inheritance, Polymorphism and Operator Overloading**

This problem involves 4 classes called '`Course`', '`CommonCoreCourse`' (derived from '`Course`' using public inheritance), '`CompCourse`' (also derived from '`Course`' using public inheritance), and '`CourseShoppingCart`'. Below are the header files of the 4 classes.

```
#ifndef COURSE_H    /* File: course.h */
#define COURSE_H

#include <iostream>
using namespace std;

// An abstract base class
class Course
{
  private:
    string code;  // Course code
    string name;  // Course name
    int credit;   // Number of credits
  public:
    // A constructor that initializes all the data members
    Course(string code, string name, int credit)
           : code(code), name(name), credit(credit) {}

    // The default virtual destructor
    virtual ~Course() = default;

    // A pure virutal function that returns a Boolean value
    // indicating whether the course is for all university
    // students or not
    virtual bool isForAll() const = 0;

    // A virtual function that prints all the data members
    virtual void print() const {
      cout << code << ", " << name << ", " << credit;
    }
};

#endif
```

```
#ifndef COMMONCORECOURSE_H    /* File: common-core-course.h */
#define COMMONCORECOURSE_H
#include "course.h"

// A derived class of the abstract base class - Course
class CommonCoreCourse : public Course {
  private:
    string area;  // The area that it belongs to
  public:
    // A constructor that initializes all the inherited data
    // members and the new data member - area
    CommonCoreCourse(string code, string name, int credit, string area)
                    : Course(code, name, credit), area(area) {}

    // isForAll() returns true as common core courses are for everyone
    virtual bool isForAll() const override { return true; }

    // print() prints all the inherited data members and the new data member - area
    virtual void print() const override {
      Course::print(); cout << ", " << area;
    }
};
#endif

#ifndef COMPCOURSE_H    /* File: comp-course.h */
#define COMPCOURSE_H
#include "course.h"

// A derived class of the abstract base class - Course
class CompCourse : public Course {
  private:
    string topicArea;  // The topic area that it belongs to
  public:
    // A constructor that initializes all the inherited data
    // members and the new data member - topicArea
    CompCourse(string code, string name, int credit, string topicArea)
               : Course(code, name, credit), topicArea(topicArea) {}

    // isForAll() returns false as COMP courses are not for everyone
    // They are for those who are major in COMP or related
    virtual bool isForAll() const override { return false; }

    // Prints all the inherited data members and the new data member - topicArea
    virtual void print() const override {
      Course::print(); cout << ", " << topicArea;
    }
};
#endif
```

```
#ifndef COURSESHOPPINGCART_H    /* File: course-shopping-cart.h */
#define COURSESHOPPINGCART_H
#include <typeinfo>
#include "common-core-course.h"
#include "comp-course.h"
using namespace std;

class CourseShoppingCart {
  private:
    // A pointer which points to an array of pointers in Course type
    Course** courseList;
    int capacity; // The capacity of the pointer array "courseList"
    int size;     // The number of elements in the pointer array "courseList"

  public:
    // TODO Part(a): Conversion constructor
    // Construct a CourseShoppingCart object by doing the following:
    // 1. Dynamically allocate an array of pointers of size "capacity"
    // 2. Initialize data member "capacity" and set "size" to 0
    // Assume the constructor should be implemented in CourseShoppingCart.cpp.
    CourseShoppingCart(const int capacity);

    // TODO Part(b): Copy constructor which performs deep copy
    // Hint:
    // 1. Two different types of objects will be pointed by the
    //    array of pointers
    // 2. typeid and static_cast/dynamic_cast may be useful
    // Assume the constructor should be implemented in CourseShoppingCart.cpp.
    CourseShoppingCart(const CourseShoppingCart& csc);

    // TODO Part(c): Destructor which deallocates all the dynamically-allocated
    // memory, including those pointed by the array of pointers
    // Assume the destructor should be implemented in CourseShoppingCart.cpp.
    ~CourseShoppingCart();

    // COMPLETED: Accessor
    int getSize() const { return size; }

    // TODO Part(d): Operator function, operator[]
    // It returns the ith pointer in the pointer array if index i is
    // in legal range, otherwise, it terminates the program
    // Assume the operator[] function should be implemented in CourseShoppingCart.cpp.
    const Course* operator[](int i) const;

    // TODO Part(e): Copy assignment operator function which performs deep copy
    // Assume the copy assignment operator function should be implemented in
    // CourseShoppingCart.cpp.
    CourseShoppingCart& operator=(const CourseShoppingCart& csc);
```

```
    // TODO Part(f): Operator function, operator+=
    // It assigns the address in parameter "course" to the next
    // available location of the pointer array "courseList". As the capacity
    // of the pointer array is fixed once it is created, you
    // need to do the following in order to store the address in the pointer array
    // 1. Allocate a new array of capacity = capacity of the original array * 2
    //    when the original array if full
    // 2. Copy all the pointers in the original array to the new array
    // 3. Assign the address in parameter "course" to the end of the new array
    // 4. Make the data member "courseList" point at the new array
    // 5. Make sure there is NO memory leak problem after performing all the
    //    above operations
    // Assume the operator+= function should be implemented in CourseShoppingCart.cpp.
    void operator+=(Course* course);

    // TODO Part(g): Operator function, operator<<, for CourseShoppingCart
    // Define and implement the operator function, operator<<,
    // here by doing the following:
    // 1. It should be made as a friend of the CourseShoppingCart
    // 2. It prints all the required information on screen.
    //    Please refer to the given sample output for what to
    //    print and the printing format
    // Assume the operator<< should be implemented in the class "CourseShoppingCart"
    // in course-shopping-cart.h.
};

#endif
```

Below is the testing program "`test-course.cpp`".

```
#include "common-core-course.h"    /* File: test-course.cpp */
#include "comp-course.h"
#include "course-shopping-cart.h"

int main()
{
  CourseShoppingCart desmondList(3);
  desmondList += (new CommonCoreCourse(
                      "HUMA1100", "Music of the World", 3, "H"));
  desmondList += (new CommonCoreCourse(
                      "SOSC1130", "Science, Technology & Society", 3, "SA"));
  desmondList += (new CompCourse(
                      "COMP2012H", "Honors OOP and Data Structures", 5, "Prog"));
  desmondList += (new CompCourse(
                      "COMP2012", "OOP and Data Structures", 4, "Prog"));

  CourseShoppingCart kelvinList(desmondList);
  kelvinList += (new CompCourse(
                     "COMP4511", "System and Kernel Prog in Linux", 3, "Systems"));

  cout << "----- Desmond's Course List -----" << endl;
  cout << desmondList << endl;

  cout << "----- Kelvin's Course List -----" << endl;
  for(int i=0; i<kelvinList.getSize(); ++i) {
    kelvinList[i]->print();
    cout << endl;
  }

  desmondList = kelvinList;

  cout << endl;
  cout << "----- Desmond's Course List -----" << endl;
  cout << desmondList << endl;
}
```

A sample run of the testing program is given as follows:

Output of the testing program

```
----- Desmond's Course List -----
List's capacity: 6
Number of courses: 4
HUMA1100, Music of the World, 3, H (For all? true)
SOSC1130, Science, Technology & Society, 3, SA (For all? true)
COMP2012H, Honors OOP and Data Structures, 5, Prog (For all? false)
COMP2012, OOP and Data Structures, 4, Prog (For all? false)

----- Kelvin's Course List -----
HUMA1100, Music of the World, 3, H
SOSC1130, Science, Technology & Society, 3, SA
COMP2012H, Honors OOP and Data Structures, 5, Prog
COMP2012, OOP and Data Structures, 4, Prog
COMP4511, System and Kernel Prog in Linux, 3, Systems

----- Desmond's Course List -----
List's capacity: 6
Number of courses: 5
HUMA1100, Music of the World, 3, H (For all? true)
SOSC1130, Science, Technology & Society, 3, SA (For all? true)
COMP2012H, Honors OOP and Data Structures, 5, Prog (For all? false)
COMP2012, OOP and Data Structures, 4, Prog (For all? false)
COMP4511, System and Kernel Prog in Linux, 3, Systems (For all? false)
```

Based on the given information in "`Course.h`", "`CommonCourse.h`", "`CompCourse.h`", and "`CourseShoppingCart.h`", implement all the missing member functions of '`CourseShoppingCart`' class in "`CourseShoppingCart.h`" and "`CourseShoppingCart.cpp`" so that the class will work with the testing program "`test-course.cpp`" and produce the given output.

Assume the statement

```
#include "course-shopping-cart.h"
```

has been put in "`course-shopping-cart.cpp`" for you.

(a) [2 points] Implement the conversion constructor for 'CourseShoppingCart' class:

```
CourseShoppingCart(const int capacity);
```

in the space provided below. Assume the implementation is in "CourseShoppingCart.cpp".

**Answer:**

```
CourseShoppingCart::CourseShoppingCart(const int capacity)
                                       : capacity(capacity) { // 0.5 point
  size = 0;                            // 0.5 point
  courseList = new Course*[capacity];  // 1 point
}
```

(b) [2 points] Implement the copy constructor for 'CourseShoppingCart' class:

```
CourseShoppingCart(const CourseShoppingCart& csc);
```

in the space provided below. Assume the implementation is in "CourseShoppingCart.cpp".

**Answer:**

```
CourseShoppingCart::CourseShoppingCart(const CourseShoppingCart& csc)
                                       : courseList(nullptr), // 0.5 point
                                         size(0) {            // 0.5 point
  *this = csc;  // 1 point
}
```

(c) [2 points] Implement the destructor for "CourseShoppingCart" class:

```
~CourseShoppingCart();
```

in the space provided below. Assume the implementation is in "CourseShoppingCart.cpp".

**Answer:**

```
CourseShoppingCart::~CourseShoppingCart() {
  for(int i=0; i<size; ++i)  // 0.5 point
    delete courseList[i];    // 1 point
  delete [] courseList;      // 0.5 point
}
```

(d) [2 points] Implement the operator function, operator[ ], for "CourseShoppingCart" class:

```
const Course* operator[](int i) const;
```

in the space provided below. Assume the implementation is in "CourseShoppingCart.cpp".

**Answer:**

```
const Course* CourseShoppingCart::operator[](int i) const {
  if(i >= 0 && i < size) {  // 1 point
    return courseList[i];   // 0.5 point
  }
  exit(-1);                 // 0.5 point
}
```

(e) [8 points] Implement the copy assignment operator function for "CourseShoppingCart" class:

```
CourseShoppingCart& operator=(const CourseShoppingCart& csc);
```

in the space provided below. Assume the implementation is in "CourseShoppingCart.cpp".

**Answer:**

```
CourseShoppingCart& CourseShoppingCart::operator=(const CourseShoppingCart& csc) {
  if(this != &csc) {           // 0.5 point
    for(int i=0; i<size; ++i)
      delete courseList[i];    // 0.5 point
    delete [] courseList;      // 0.5 point

    capacity = csc.capacity;                      // 0.5 point
    courseList = new Course*[csc.capacity];       // 0.5 point
    size = csc.size;                              // 0.5 point
    for(int i=0; i<size; ++i) {
      Course* c = csc.courseList[i];              // 0.5 point
      if(typeid(*c) == typeid(CommonCoreCourse))  // 1 point
        // 1 point
        courseList[i] = new CommonCoreCourse(*dynamic_cast<CommonCoreCourse*>(c));
      else if(typeid(*c) == typeid(CompCourse))   // 1 point
        // 1 point
        courseList[i] = new CompCourse(*dynamic_cast<CompCourse*>(c));
    }
  }
  return *this;  // 0.5 point
}
```

(f) [6 points] Implement the operator function, operator+=, for "CourseShoppingCart" class:

```
void operator+=(Course* course);
```

in the space provided below. Assume the implementation is in "CourseShoppingCart.cpp".

**Answer:**

```
void CourseShoppingCart::operator+=(Course* course) {
  if(size == capacity) {                          // 0.5 point
    capacity *= 2;                                // 0.5 point
    Course** tempList = new Course*[capacity];    // 1 point
    for(int i=0; i<size; ++i)                     // 0.5 point
      tempList[i] = courseList[i];                // 0.5 point
    delete [] courseList;                         // 0.5 point
    courseList = tempList;                        // 1 point
  }
  courseList[size++] = course;                    // 1.5 point
}
```

(g) [4 points] Implement the operator function, operator<<, for "CourseShoppingCart" class in the space provided below. Assume the implementation is in 'CourseShoppingCart' class in "CourseShoppingCart.h".

**Answer:**

```
friend ostream& operator<<(ostream& os, const CourseShoppingCart& csc) {
  os << "List's capacity: " << csc.capacity << endl;  // 0.5 point
  os << "Number of courses: " << csc.size << endl;    // 0.5 point
  for(int i=0; i<csc.size; ++i) {                     // 0.5 point
    csc.courseList[i]->print();                       // 1 point
    // 1 point
    os << " (For all? " << boolalpha << csc.courseList[i]->isForAll() << ")" << endl;
  }
  return os;  // 0.5 point
}
```

**Problem 7 [29 points] Binary Search Tree With Duplicates**

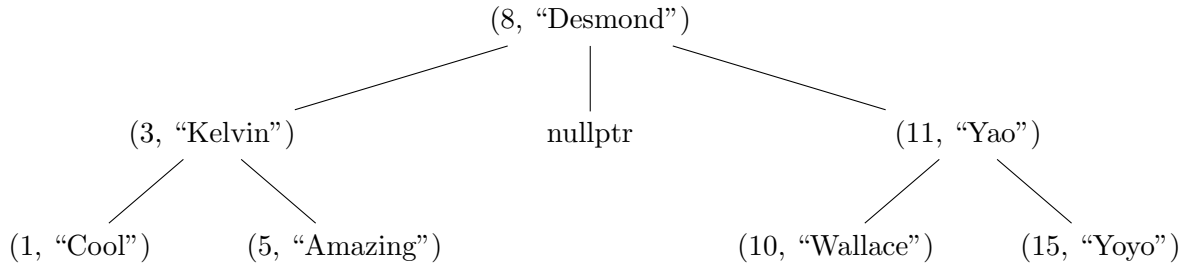
This question is about implementing a Binary Search Tree that allows duplicate elements, which is called "BST with duplicates (BSTD)".

In BSTD, if we insert a key that is already existed, we should create a BSTDnode with key and data, and then add it to the front of the BSTD named "same" as shown in the modified code of BST below.

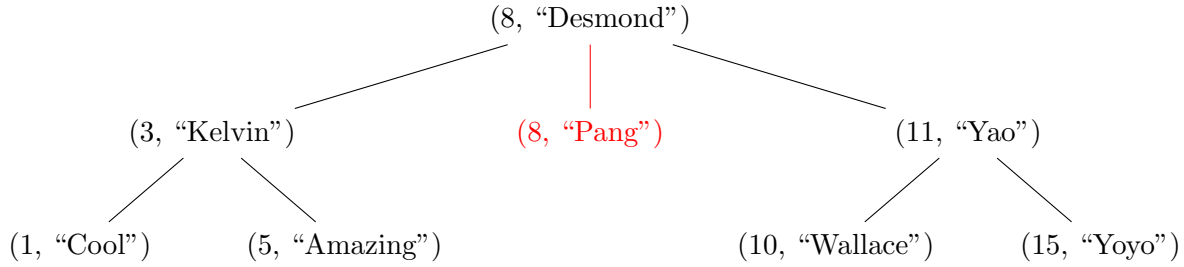
```
#include <iostream>  /* File: bstd.h */
#include <utility>
using namespace std;

template <typename T1, typename T2>
class BSTD // Binary Search Tree with duplicates
{
  private:
    struct BSTDnode // A BSTDnode stores two things: key and data
    {
      T1 key;       // Store the key (T1 type)
      T2 data;      // Store the data (T2 type)
      BSTD left;    // Left tree
      BSTD same;    // This is a BSTD that stores duplicates
      BSTD right;   // Right tree

      BSTDnode(const T1& x, const T2& y) : key(x), data(y) {}  // Constructor
      BSTDnode(const BSTDnode& node) = default; // Default copy constructor
      ~BSTDnode() = default; // Default destructor
    };

    BSTDnode* root = nullptr;

  public:
    BSTD() = default;        // Default constructor
    ~BSTD() { delete root; } // Destructor

    BSTD(const BSTD& bstd) { // Copy constructor
      if(bstd.is_empty())    // Do nothing if the tree is empty
        return;
      root = new BSTDnode(*bstd.root);
    }

    bool is_empty() const { // Check if the tree is empty
      return root == nullptr;
    }
```

```
    void print(int depth = 0) const { // Print all the data in the tree
      if(is_empty()) // If the tree is empty, nothing to print
        return;

      root->right.print(depth+1); // Print right subtree

      for(int j=0; j<depth; ++j)  // Spacing
        cout << '\t';
      // Print key and data
      cout << "(" << root->key << "," << root->data << ")" << endl;

      root->same.print(depth);    // Print "same" subtree
      root->left.print(depth+1);  // Print left subtree
    }

    /*** TODO: Member functions to be completed ***/

    // TODO Part(a):
    // Goal: To find the min key stored in the BSTD.
    // Return: The minimum key in type T1, and the corresponding data in type T2.
    // If there is more than one minimum key, return the key and data of the leaf
    // node of "same". You may refer to the Appendix for the use of STL pair.
    const pair<T1,T2> find_min() const;

    // TODO Part(b):
    // Goal: To insert the key x and data y into the current BSTD
    void insert(const T1& x, const T2& y);

    // TODO Part(c):
    // Goal: To remove the key x and data y from the current BSTD
    // Note: To handle the case where the node to be deleted with 2 children
    // (excluding "same"), replace the deleted node with the minimum node in its
    // right sub-tree. If there is more than one minimum node, replace the deleted
    // node with the leaf node of "same"
    void remove(const T1& x, const T2& y);
};

#include "bstd.tpp"
```

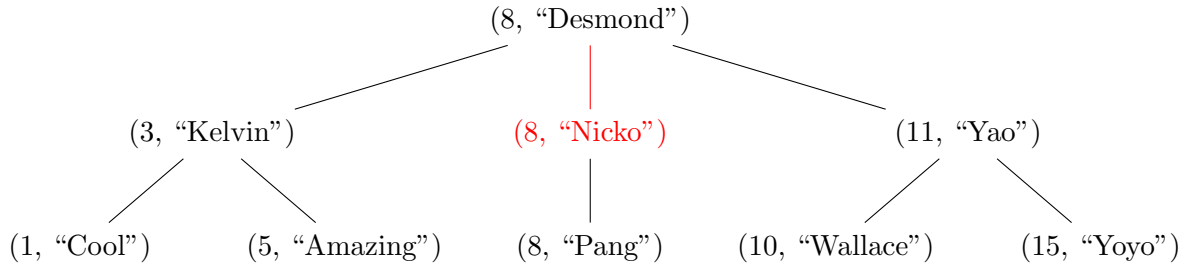
To understand the structure of a BSTD, an example is given below. Assume we insert (8, "Desmond"), (3, "Kelvin"), (11, "Yao"), (1, "Cool"), (5, "Amazing"), (10, "Wallace"), (15, "Yoyo") in sequence (from left to right) to an initially empty BSTD, we get the following.




Suppose we further insert (8, "Pang") to the tree above, a BSTDnode with key 8 and data "Pang" is created and added to the front of the "same" BSTD, which looks like the following.




Now, if we insert (8, "Nicko") to the tree, another BSTDnode with key 8 and data "Nicko" is created and added to the front of the "same" BSTD.



Based on the given information and the descriptions in the "BSTD.h", implement the 3 undefined member functions of the class 'BSTD' in "bstd.tpp" so that they will work with the testing program 'test-bstd.cpp" to produce the expected output.

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

int main() {
  BSTD<int,string> bstd;
  while(true) {
    char choice;
    int key;
    string name;
    cout << "Action: i/m/p/q/r (insert/min/print/quit/remove): ";
    cin >> choice;
    switch(choice) {
      case 'i':
        cout << "Key to insert: ";
        cin >> key;
        cout << "Name to insert: ";
        cin >> name;
        bstd.insert(key, name);
        break;
      case 'm':
        if(bstd.is_empty())
          cerr << "Can't search an empty tree!" << endl;
        else {
          pair<int, string> p = bstd.find_min();
          cout << p.first << ", " << p.second << endl;
        }
        break;
      case 'p':
        bstd.print();
        break;
      case 'q':
        return 0;
      case 'r':
        cout << "Key to remove: ";
        cin >> key;
        cout << "Name to remove: ";
        cin >> name;
        bstd.remove(key, name);
        break;
    }
  }
}
```

Expected output of the testing program

```
Action: i/m/p/q/r (insert/min/print/quit/remove): i
Key to insert: 8
Name to insert: Desmond
Action: i/m/p/q/r (insert/min/print/quit/remove): i
Key to insert: 3
Name to insert: Kelvin
Action: i/m/p/q/r (insert/min/print/quit/remove): i
Key to insert: 11
Name to insert: Yao
Action: i/m/p/q/r (insert/min/print/quit/remove): i
Key to insert: 1
Name to insert: Cool
Action: i/m/p/q/r (insert/min/print/quit/remove): i
Key to insert: 5
Name to insert: Amazing
Action: i/m/p/q/r (insert/min/print/quit/remove): i
Key to insert: 10
Name to insert: Wallace
Action: i/m/p/q/r (insert/min/print/quit/remove): i
Key to insert: 15
Name to insert: Yoyo
Action: i/m/p/q/r (insert/min/print/quit/remove): p
                (15,Yoyo)
        (11,Yao)
                (10,Wallace)
(8,Desmond)
                (5,Amazing)
        (3,Kelvin)
                (1,Cool)
Action: i/m/p/q/r (insert/min/print/quit/remove): i
Key to insert: 8
Name to insert: Pang
Action: i/m/p/q/r (insert/min/print/quit/remove): p
                (15,Yoyo)
        (11,Yao)
                (10,Wallace)
(8,Desmond)
(8,Pang)
                (5,Amazing)
        (3,Kelvin)
                (1,Cool)
Action: i/m/p/q/r (insert/min/print/quit/remove): i
Key to insert: 8
Name to insert: Nicko
```

```
Action: i/m/p/q/r (insert/min/print/quit/remove): p
                (15,Yoyo)
        (11,Yao)
                (10,Wallace)
(8,Desmond)
(8,Nicko)
(8,Pang)
                (5,Amazing)
        (3,Kelvin)
                (1,Cool)
Action: i/m/p/q/r (insert/min/print/quit/remove): m
1, Cool
Action: i/m/p/q/r (insert/min/print/quit/remove): r
Key to remove: 3
Name to remove: Kelvin
Action: i/m/p/q/r (insert/min/print/quit/remove): p
                (15,Yoyo)
        (11,Yao)
                (10,Wallace)
(8,Desmond)
(8,Nicko)
(8,Pang)
        (5,Amazing)
                (1,Cool)
Action: i/m/p/q/r (insert/min/print/quit/remove): r
Key to remove: 5
Name to remove: Amazing
Action: i/m/p/q/r (insert/min/print/quit/remove): p
                (15,Yoyo)
        (11,Yao)
                (10,Wallace)
(8,Desmond)
(8,Nicko)
(8,Pang)
        (1,Cool)
Action: i/m/p/q/r (insert/min/print/quit/remove): r
Key to remove: 8
Name to remove: Desmond
Action: i/m/p/q/r (insert/min/print/quit/remove): p
                (15,Yoyo)
        (11,Yao)
                (10,Wallace)
(8,Nicko)
(8,Pang)
        (1,Cool)
Action: i/m/p/q/r (insert/min/print/quit/remove): q
```

(a) [6 points] Implement the member function:

```
const pair<T1,T2> find_min() const;
```

of 'BSTD' class in the space provided below. Assume the implementation is in "bstd.tpp". Note: You may refer to the Appendix for the use of STL pair.

**Answer:**

```
// 0.5 points - definition of template parameters
template <typename T1, typename T2>
// 0.5 points - function header
const pair<T1,T2> BSTD<T1,T2>::find_min() const {
  const BSTDnode* node = root;

  // 2 points - reaching the left-most node
  while(!node->left.is_empty())
    node = node->left.root;

  // 2 points - reaching the leaf node of same of the left-most node
  while(!node->same.is_empty())
    node = node->same.root;

  // 0.5 points - return pair<T1,T2>
  // 0.5 points - node->key, node->data
  return pair<T1,T2>(node->key, node->data);
}
```

(b) [8 points] Implement the member function:

```
void insert(const T1& x, const T2& y);
```

of 'BSTD' class in the space provided below. Assume the implementation is in "bstd.tpp".

**Answer:**

```
// 0.5 points - definition of template parameters
template <typename T1, typename T2>
// 0.5 points - function header
void BSTD<T1,T2>::insert(const T1& x, const T2& y) {

  // create new node if empty
  // 1 point - condition: root is nullptr
  if(is_empty())
    // 0.5 points - new BSTDnode(x, y)
    // 0.5 points - assign new node to root
    root = new BSTDnode(x, y);
  // -0.5 points if fail to insert to empty tree

  // 1 point - go to left if key is smaller
  else if(x < root->key)
    root->left.insert(x, y);

  // 1 point - go to right if key is larger
  else if(x > root->key)
    root->right.insert(x, y);

  // insert to same
  // 1 point - condition: key is the same
  else {
    // 1 point - new BSTDnode(x, y)
    BSTDnode* node = new BSTDnode(x, y);

    // 1 point - insert to the correct location in same
    node->same.root = root->same.root;
    root->same.root = node;
  }
}
```

(c) [15 points] Implement the member function:

```
void remove(const T1& x, const T2& y);
```

of 'BSTD' class in the space provided below. Assume the implementation is in "bstd.tpp".

**Answer:**

```
// 0.5 points - definition of template parameters
template <typename T1, typename T2>
void BSTD<T1,T2>::remove(const T1& x, const T2& y) { // 0.5 points - function header
  // 1 point - termination if x is not found and check if root is nullptr
  if(is_empty()) return;

  // 0.5 points - goto left if key is smaller
  if(x < root->key)
    root->left.remove(x, y);

  // 0.5 points 0 goto right if key is larger
  else if(x > root->key)
    root->right.remove(x, y);

  // key is found
  else {
    // 1 point - goto same if data is not found
    if(root->data != y)
      root->same.remove(x, y);

    // data is found, delete the current node
    else {
      // case 1: same is not empty
      // 0.5 points - condition
      if(!root->same.is_empty()) {
        // save the node to be deleted
        BSTDnode *node = root; // Save the root to delete first

        // 0.5 point - set the next node in same as root
        root = node->same.root;

        // 1 point - set the left and right (or key and data)
        root->left.root = node->left.root;
        root->right.root = node->right.root;

        // 0.5 points - set subtrees to nullptr before removal
        // due to recursive destructor
        node->left.root = node->right.root = node->same.root = nullptr;

        // 0.5 points - delete root
        delete node;
      }
```

```
      // case 2: no children
      // 0.5 points - condition: both left and right are empty
      else if(root->left.is_empty() && root->right.is_empty()) {
        // 0.5 points - delete root
        delete root;

        // 0.5 points - set current node as empty
        root = nullptr;
      }

      // case 3: one child
      // 0.5 points - condition: either left or right is empty
      else if(root->left.is_empty() || root->right.is_empty()) {
        // save the node to be deleted
        BSTDnode *node = root;

        // 0.5 points - set the (left or right) node as root
        root = (root->left.is_empty()) ? root->right.root : root->left.root;

        // 0.5 points - set subtrees to nullptr before removal due to
        // recursive destructor
        node->left.root = node->right.root = node->same.root = nullptr;

        // 0.5 points - delete root
        delete node;
      }

      // case 4: two children
      // 0.5 points - condition: both left and right are not empty
      else {
        pair<T1,T2> p = root->right.find_min();

        // 1 point - set the key and data (or left and right)
        root->key = p.first;
        root->data = p.second;

        // 1 point - remove min node of right
        root->right.remove(root->key, root->data);
      }

      // 2 points - correct handling of all 4 cases
      // when same is not empty, always move up the next node in same
    }
  }
}
```

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

# Appendix

---

### stack - STL Container Adaptor

```
template <class T, class Container = deque<T> >
class stack;
```

Defined in the standard header `stack`.

Stack is a type of container adaptor, specifically designed to operate in a LIFO context (last-in first-out), where elements are inserted and extracted only from one end of the container.

- **Parameters**:
  - `T`: Type of elements.
  - `Container`: Type of the internal underlying container object where the elements are stored.
- **Useful member functions**:
  - `explicit stack()`:
    Constructs a stack container adaptor object.
  - `bool empty()`:
    Returns whether the stack is empty. Returns true if the underlying container's size is 0, false otherwise.
  - `unsigned int size() const`:
    Returns the number of elements in the stack.
  - `T& top()` / `const T& top() const`:
    Returns a reference/const reference to the top element in the stack.
  - `void push(const T& val)`
    Inserts a new element, `val`, at the top of the stack, above its current top element.
  - `void pop()`:
    Removes the element on top of the stack, effectively reducing its size by one.

### vector - STL Sequence Container

```
template <class T>
class vector;
```

Defined in the standard header `vector`.
Vector is a sequence container representing an array that can change in size.

- **Parameters**:
  - `T`: Type of elements.
- **Useful member functions**:
  - `explicit vector()`:
    Constructs a vector container object.
  - `explicit vector(int n)`:
    Constructs a vector container object with `n` elements.
  - `void resize(int n)`:
    Resizes the container so that it contains `n` elements.
  - `unsigned int size() const`:
    Returns the number of elements in the vector container.
  - `T& operator[](int n)` / `const T& operator[](int n) const`:
    Returns a reference/const reference to the element at position `n` in the vector container.

### pair - STL utility

```
template <class T1, class T2>
struct pair;
```

Defined in the standard header `utility`.
Pair is a class template that provides a way to store two heterogeneous objects as a single unit.

- **Parameters**:
  - `T1`: Type of the first element.
  - `T2`: Type of the second element.
- **Useful data members and member function**:
  - `T1 first`
    The first element (`T1` type)
  - `T2 second`
    The second element (`T2` type)
  - `pair(const T1& x, const T2& y)`:
    Constructs a pair type object by assinging `x` to `first`, and `y` to `second`.

/* Rough work */

/* Rough work */