Indian Institute of Technology, Kharagpur

Department of Computer Science and Engineering

End-Semester Examination, Spring 2014-15

Software Engineering (CS 20006): **SOLUTIONS**

Students: 135 **Date**: 21-Apr-15 (AN)

Full marks: 100 **Time**: 3 hours

Instructions:

1. Marks for every question is shown with the question.
2. No clarification to any of the questions will be provided. If you have any doubt, please make suitable assumptions and proceed. State your assumptions clearly. While making assumptions, be careful that you do not contradict any explicitly stated fact in the question.

---

1. A course on Software Construction in IIT wants to manage the assignments to *Students*, the submissions of assignments by *Students*, and the evaluations of the submissions through an **Assignment Management System (AMS)**. The requirement specifications for the system are as follows:

   (a) Participants of the course are:
   - One *Instructor*. She / he is identified by an *Employee Code*, and has *Name*, *Email* and *Mobile Number*.
   - 5 or more *Teaching Assistants (TA)*. Every *TA* is identified by a *Roll No*, and has *Name*, *Email* and *Mobile Number*.
   - 100 or more *Students*. Every *Student* is identified by a *Roll No*, and has *Name*, *Department*, *Hall*, *Email* and *Mobile Number*.

   (b) The responsibilities of the *Instructor* include:
   - *Set-up*: Design and set up the assignments, decide on the date that an assignment is to be assigned to the *Students*, and on the date of submission for the assignment. Also decide on a *Coordinating TA* for an assignment.
   - *Allocation*: Allocate *Students* to *TA*. Every *Student* has one allocated *TA*.
   - *Approve*: Approve / Disapprove extensions for submissions beyond the submission date and decide on the penalty.
   - *Compilation*: Compile the evaluations (as performed by the *TA*s) and publish the final scores of every assignments on **AMS**.

   (c) The responsibilities of a *TA* include:
   - *Mentor*: Mentor and manage the *Students* allocated to the *TA*. She / he is the primary support for the allocated *Students* before she / he should approach the *Instructor*.
   - *Upload*: The *Coordinating TA* of an assignment uploads an assignment as set by the *Instructor* and sets up the required *Assign* and *Submission Dates* on the **AMS**.
   - *Download*: Download from the **AMS** and archive the submissions for the *Students* allocated to the *TA*. This needs to be done after the submission date in every assignment.
   - *Evaluate*: Peruse the submissions, discuss with the respective *Students* for clarifications, take demonstrations (if relevant), and evaluate.
   - *Report*: Report the evaluations to *Instructor*. Requests for submission date extension with full credit, on grounds of medical or personal exigencies, are also to be reported after proper authentication. Further, *TA*s are responsible for reporting Disciplinary actions (like plagiarism), if any.

(d) The responsibilities of a *Student* include:

- *Perform*: Complete every assignment within the submission date and submit to **AMS**.
- *Appeal*: Appeal to the *Instructor* for permission for special submission without penalty. Every appeal needs to go through the respective *TA* and must be authenticated by her / him.
- *Demonstrate*: Discuss and demonstrate the submission to the allocated *TA*.

(e) An *Assignment*:

- *Ownership*: Is to be completed individually by every *Student*.
- *Type*: Is one the following types:
  - *Programming Assignment*: The assignment has one *Problem* that asks to write a single program, specifies the language for coding (like C / C++ / Java), and has a single submission date.
  - *Systems Assignment*: The assignment asks to develop a System. It has one or more *Problem/s*. Each *Problem* is about a component module that builds up the System and has a separate submission date. No coding language is specified for such assignments.
- *Assign Date*: Has an *Assign Date* on which it is given to the *Students*.
- *Submission Date*: Has a *Submission Date* by which it is to be completed and submitted. For a *Systems Assignment* every constituent *Problem* has a separate *Submission Date* but a common *Assign Date*.
- *Marks*: Every *Problem* in an assignment has specified maximum marks.
- *Coordinator*: By turn a *TA* is allocated by the *Instructor* as a coordinator for an assignment. She / he manages the upload, and the dates for the assignment.

(f) A *Submission* :

- *Action*: Is performed for an *Assignment*, by a *Student* on a *Date*.
- *Valid*: Is *valid* if it is submitted within the *Submission Date* of the corresponding assignment. *Valid* submissions carry full credit.
- *Late*: Is *late* if it is done within 3 days from the *Submission Date* of the corresponding assignment. *Late* submissions carry 10% penalty.
- *Special*: Is *special* if it is done within 7 days from the *Submission Date* of the corresponding assignment. A *Student* needs to *appeal* for *Special* submissions to the *Instructor* through the *TA*. *Special* submissions are allowed on grounds of medical or personal exigency and carry no penalty.
- *Invalid*: Is *invalid* if it is not submitted within 3 days from the the *Submission Date* of the corresponding assignment and has not been granted extension as a *Special Submission*. *Invalid* submissions carry zero credit.

No submission is allowed before the *Assign Date* of the submission.

(g) The *Work flow* in the course is as follows:

- The *Instructor* designs an *Assignment* and mails it to the *Coordinating TA*.
- The *Coordinating TA* uploads the *Assignment* to the system and sets the corresponding *Assign* and *Submission Dates*.
- Once an *Assignment* has been set, **AMS** sends an email notification to all *Students*, *TA*s and the *Instructor* about the *Assignment*.
- The *Students* completes the *Assignment* and uploads the solution to **AMS**.
- Once the *Submission Dates* for an *Assignment* is over, **AMS** sends an email notification each to every *TA* with the respective list of completed submissions (*Students'* roll numbers are mentioned) by the *Students* allocated to the *TA*. Respective students are carbon-copied on the email. The *Instructor* is copied on every notification.
- Every *TA* downloads the respective submissions, discusses with the *Student*s, checks demonstrations, and evaluates the solution.
- Once a *TA* completes her / his evaluations for an *Assignment*, **AMS** sends an email notification to the *Instructor* with the *TA* on the carbon-copy. *TA* should complete the evaluations after 4 days from the *Submission Date* and before 10 days from it.

- The *Instructor* on receipt of completion report from all the *TA*s compiles the scores for an *Assignment* and publishes on the **AMS**.
- If a *Student* misses to submit by the *Submission Date*, but submits within 3 days from that, she / he is penalized by 10%. On such submissions, **AMS** sends an email notification to the *TA* and the *TA* would similarly evaluate the submission and report to the *Instructor*.
- A *Student* may appeal for an extension on grounds of medical or personal exigency. The *TA* would scrutinize the appeal and report to the *Instructor*, if authentic. **AMS** will send an email notification for the same. As the *Instructor* approves or disapproves the appeal, accordingly the *Student* and the *TA* are notified. If the appeal is approved, the process of submission and evaluation is followed.

You have been assigned as the software engineer for the **AMS**. You are required to analyse the specifications, design the system (using UML and DP) and also prepare the test plan. Answer the following questions in this background.

(a) Identify the actions in **AMS** and design the Use-Case Diagrams for the actions. Identify the actors, specify their types, and mark the relationships between the actors. Show the <<`include`>>, <<`extend`>>, and generalization relationships of the use-cases. **[4+4=8]**

(b) Design Class Diagrams for *Assignments* & *Submissions*. Show the attributes and operations with their associated properties. Highlight specialization hierarchies, if any. **[4+4=8]**

(c) Complete the Class Diagram of **AMS** showing all other classes (in addition to Question 1b) by their respective brief Diagrams (with name and key attributes). For the entire collection of classes (that is, including *Assignments* and *Submissions*) show the associations, aggregations / compositions, generalization / specialization, and abstract / concrete etc. **[6]**

(d) Design the State-Chart Diagrams for *Assignments* and *Submissions*. **[2+2=4]**

(e) Design Sequence Diagrams for the actions in **AMS** as specified in the Work flow. **[10]**

(f) Identify and justify the use of Iterator, Singleton and Command DPs in **AMS**. **[2*3=6]**

(g) Prepare a test plan for **AMS** to perform black-box tests. Clearly mark the scenarios for Unit Testing and Integration Testing. **[4+4=8]**

Answers:

Part 1a: The Use-cases of **AMS** are:

Part 1b: Class diagrams for ***Assignment*** and ***Submission***:

An Assignment can contain one or more Problems. Hence we need an auxiliary **Problem** class to define ***Assignment*** class.

	Problem			*Remarks*
−	id	:	String	Auto-generated
−	assignment	:	String	Set by Create(). assignment = Assignment.id
−	submissionDate	:	Date	Set by Create()
−	marks	:	Int	Set by Create()
−	body	:	Text	Set by Create()
−	/status	:	Bool[3]	Set by Create(). status[0] = True, rest False. Changes by TRIGGER
+	Create(assignment: String, submissionDate: Date, marks: Int, body: Text, language: String =""): Problem *			Constructor for Problem
+	Display(): void			Display all fields

The status values are derived as:

Index	Status	Remarks
0	new	The problem is created as a part of assignment that contains it
1	open	The problem is open, if today $\geq$ assignment.assignDate and today $\leq$ submissionDate
2	closed	The problem is closed, if today $>$ submissionDate

Assignment {*Abstract*}			Remarks
−	id	: String	Auto-generated
−	setter	: String	Set by Create(). setter = Instructor.eCode
−	assignDate	: Date	Set by Create()
−	coordinator	: String	Set by Create(). coordinator = TA.roll
−	nProblems	: Int	Set by Create(). Number of problems in the assignment
−	problems	: String[1..nProblems]	Set by Create(). problem = Problem.id
−	status	: Bool[4]	Set by Create(). status[0] = True, rest False. Changes by state-chart
+	Create(setter: String, assignDate: Date, coordinator: TA, nProblems: Int, problems: String[]): Assignment *		Create factory for Assignment
+	Display(): void		Display all fields
+	Upload(coordinator: String): void		Upload for assigning to students. coordinator = TA.roll

The status values are derived as:

Index	Status	Remarks
0	new	The assignment has been created by Instructor and mailed to Coordinating TA
1	uploaded	The assignment has been uploaded by Coordinating TA
2	open	The assignment is open, if at least one of the Problems is open
3	closed	The assignment is closed, if all the Problems are closed

There are two specializations – **ProgrammingAssignment** and **SystemsAssignment** – of ***Assignment***. These are concrete classes.

ProgrammingAssignment			Remarks
Base **Assignment**			nProblems = 1 by construction
−	language	: String	Set by Create(). One of {C, C++, Java}
+	Create(setter: String, assignDate: Date, coordinator: TA, nProblems: Int, problems: String[], language: String): Assignment *		Constructor for ProgrammingAssignment

SystemsAssignment		Remarks
Base **Assignment**		nProblems $\geq$ 1 by construction
+	Create(setter: String, assignDate: Date, coordinator: TA, nProblems: Int, problems: String[]): Assignment *	Constructor for SystemsAssignment

	Submission {*Abstract*}			*Remarks*
–	id	:	String	Auto-generated
–	student	:	String	Set by Create(). student = Student.roll
–	problem	:	String	Set by Create(). problem = Problem.id
–	date	:	Date {optional}	The date of problem submission. Set to null by Create(), finalized by FinalizeSubmission()
–	body	:	Text {optional}	The solution as submitted. Set by Create(), edited by EditSubmission()
–	marks	:	Int	Set to 0 by Create(). Set by EvalAndReport()
–	lateSubmissionAppeal	:	Bool	Set to False by Create(). Set by AppealLateSubmission()
–	lateSubmissionReason	:	String {optional}	Set to null by Create(). Set by AppealLateSubmission()
–	lateSubmissionForward	:	Bool	Set to False by Create(). Set by EvalAndReport()
–	lateSubmissionApproval	:	Bool	Set to False by Create(). Set by AllowLateSubmission()
–	/credit	:	Int	Percentage of credit (max = 100). Set to 0 by Create(). Changes by state-chart
–	/status	:	Bool[8]	Set by Create(). status[0] = True, rest False. Changes by state-chart
+	<u>Create(student: String, problem: String, body: Text = ""): Submission *</u>			Constructor for Submission
+	Display(): void			Display all fields
+	EditSubmission(): void			body of Submission edited by student
+	FinalizeSubmission(): void			Submission done (finalized) by student – no more edits allowed
+	AppealLateSubmission(reason: String): void			student appeals for late submission with reason
+	EvalAndReport(marks: Int, lateSubmission: Bool): void			TA of student reports the evaluation and / or response to appeal for late submission with reason
+	AllowLateSubmission(): void			Instructor allows late submission

The status and credit values are derived as:

Index	Status	Remarks			Credit
0	draft	The submission has been created but not submitted			0
		lateSubmissionAppeal	lateSubmissionForward	lateSubmissionApproval	0
1	appealed	True	False	False	0
2	forwarded	True	True	False	0
3	approved	True	True	True	0
4	valid	The submission has been done and date $\leq$ problem.submissionDate			100
5	late	The submission has been done, date $>$ problem.submissionDate, date $\leq$ problem.submissionDate + 3, and lateSubmissionApproval = False			90
6	special	The submission has been done, date $>$ problem.submissionDate, date $\leq$ problem.submissionDate + 7, and lateSubmissionApproval = True			100
7	invalid	The submission has been done, date $>$ problem.submissionDate + 3 and lateSubmissionApproval = False			0

Part 1c: The Class diagram for **AMS**:

Part 1d: The **Problem** has the following states that change according to the state-chart:

Present State	Action	Next State	Actor	Status			Method & Condition
				new	open	closed	
null	Create	New	Instructor	True	False	False	Create() & today $<$ assignment.assignDate
New	TRIGGER	Open	System	False	True	False	today $\geq$ assignment.assignDate & today $\leq$ submissionDate
Open	TRIGGER	Closed	System	False	False	True	today $>$ submissionDate

The ***Assignment*** has the following states that change according to the state-chart:

Present State	Action	Next State	Actor	Status				Method & Condition
				new	uploaded	open	closed	
null	Create	New	Instructor	True	False	False	False	Create()
New	Upload	Uploaded	TA	False	True	False	False	Upload()
Uploaded	TRIGGER	Open	System	False	False	True	False	At least one problem is in open state
Open	TRIGGER	Closed	System	False	False	False	True	All problems are in closed state

The ***Submission*** has the following states that change according to the state-chart:

Condition Name	Condition Expression
Cond 1	date $\leq$ problem.submissionDate
Cond 2	date $>$ problem.submissionDate $\&$ date $\leq$ problem.submissionDate $+ 3$
Cond 3	date $>$ problem.submissionDate $+ 3$
Cond 4	date $>$ problem.submissionDate $\&$ date $\leq$ problem.submissionDate $+ 7$
Cond 5	date $>$ problem.submissionDate $+ 7$

Present State	Action	Next State	Actor	Status								Method & Condition
				draft	appealed	forwarded	approved	valid	late	special	invalid	
Create												
null	Create	New	Student	T	F	F	F	F	F	F	F	Create()
Appeal for Extension												
New	Appeal	Appealed	Student	T	T	F	F	F	F	F	F	AppealLateSubmission()
Appealed	Forward	Forwarded	TA	T	T	T	F	F	F	F	F	EvalAndReport(True)
Appealed	Forward	Appealed	TA	T	T	F	F	F	F	F	F	EvalAndReport(False)
Forwarded	Approve	Approved	Instructor	T	T	T	T	F	F	F	F	AllowLateSubmission(True)
Forwarded	Approve	Forwarded	Instructor	T	T	T	F	F	F	F	F	AllowLateSubmission(False)
Edit												
New	Edit	New	Student	T	F	F	F	F	F	F	F	EditSubmission()
Appealed	Edit	Appealed	Student	T	T	F	F	F	F	F	F	EditSubmission()
Forwarded	Edit	Forwarded	Student	T	T	T	F	F	F	F	F	EditSubmission()
Approved	Edit	Approved	Student	T	T	T	T	F	F	F	F	EditSubmission()
Submission												
New	Submit	Valid	Student	F	F	F	F	T	F	F	F	FinalizeSubmission() & date $\leq$ problem.submissionDate
New	Submit	Late	Student	F	F	F	F	F	T	F	F	FinalizeSubmission() & date $>$ problem.submissionDate & date $\leq$ problem.submissionDate + 3
New	Submit	Invalid	Student	F	F	F	F	F	F	F	T	FinalizeSubmission() & date $>$ problem.submissionDate + 3
Appealed	Submit	Valid	Student	F	T	F	F	T	F	F	F	FinalizeSubmission() & date $\leq$ problem.submissionDate
Appealed	Submit	Late	Student	F	T	F	F	F	T	F	F	FinalizeSubmission() & date $>$ problem.submissionDate & date $\leq$ problem.submissionDate + 3
Appealed	Submit	Invalid	Student	F	T	F	F	F	F	F	T	FinalizeSubmission() & date $>$ problem.submissionDate + 3
Forwarded	Submit	Valid	Student	F	T	T	F	T	F	F	F	FinalizeSubmission() & date $\leq$ problem.submissionDate
Forwarded	Submit	Late	Student	F	T	T	F	F	T	F	F	FinalizeSubmission() & date $>$ problem.submissionDate & date $\leq$ problem.submissionDate + 3
Forwarded	Submit	Invalid	Student	F	T	T	F	F	F	F	T	FinalizeSubmission() & date $>$ problem.submissionDate + 3
Approved	Submit	Valid	Student	F	T	T	T	T	F	F	F	FinalizeSubmission() & date $\leq$ problem.submissionDate
Approved	Submit	Special	Student	F	T	T	T	F	F	T	F	FinalizeSubmission() & date $>$ problem.submissionDate date $\leq$ problem.submissionDate + 7
Approved	Submit	Invalid	Student	F	T	T	T	F	F	F	T	FinalizeSubmission() & date $>$ problem.submissionDate + 7
Timeout												
New	TRIGGER	Invalid	System	T	F	F	F	F	F	F	T	date $>$ problem.submissionDate + 3
Appealed	TRIGGER	Invalid	System	T	T	F	F	F	F	F	T	date $>$ problem.submissionDate + 3
Forwarded	TRIGGER	Invalid	System	T	T	T	F	F	F	F	T	date $>$ problem.submissionDate + 3
Approved	TRIGGER	Invalid	System	T	T	T	T	F	F	F	T	date $>$ problem.submissionDate + 7

Part 1e: The Sequence Diagram for major activities in **AMS** are:

Part 1f: The Design Patterns in **AMS** are:

Situation	Pattern	Marks
Generate assignment objects of appropriate specialization.	Abstract Factory, Factory Method	0 mark – Not covered in class
Browse / Display lists of assignments, problems in an assignment, students, TAs etc.	Iterator	2 marks
There is only one instructor	Singleton	2 marks
Instructor asks Coordinating TA to upload assignment, students appeal to TAs, TAs forward to instructor, etc. Asking for the action and the execution of the action are separated in time as is needed in Command Pattern.	Command	2 marks

Part 1g:

(a) **Unit Testing**:

i. **Testing for Classes**:

For every class tests need to be performed for:

- Constructor / Destructor / Copy Constructor / Copy Assignment Operator etc
- Member Functions
- Static Member Functions

In addition, some important scenarios for every class must be tested:

- Class Instructor: The class must be a singleton.
- Class TA
- Class Student
- Class Problem
  - Trigger (Time) based tests – Before Assign Date,Between Assign Date and Submission Date, After Submission Date
- Class Assignment
  - Test for Specializations
  -
- Class Submission

ii. **Testing Sub-Systems**:

(b) **Integration Testing**:

i. Login to **AMS**

2. Let `unique_ptr` be an *Exclusive Ownership – No Copy* smart pointer. It is not possible to copy such pointers. The ownership can only be changed through `swap()`. The interface for `unique_ptr` is given below:

```
template<class T> class unique_ptr { T* ptr_;   // The raw pointer
    unique_ptr(unique_ptr<T>& p);               // Copy constructor
    unique_ptr& operator=(unique_ptr<T>& p);    // Copy assignment
public:
    explicit unique_ptr(T* p) throw();          // RAII Constructor
    unique_ptr() throw();                       // Default Constructor
    ~unique_ptr();                              // Releases the pointer (with destruction)
    T& operator*() const;                       // Dereference operator
    T* operator->() const throw();              // Indirection operator
    operator bool() const throw();              // Returns whether the unique_ptr is not empty
    T* get() const throw();                     // Gets the raw pointer (ptr_)
    T* release() throw();                       // Returns the raw pointer (ptr_) & nulls its value
                                                // (w/o destruction)
    void swap (unique_ptr& u) throw();          // Exchanges the contents (ptr_) of the unique_ptr
                                                // object with those of u -- w/o destruction
};
```

(a) Implement the unique_ptr class. **[1*7+1.5*2=10]**

Answer:

```
template<typename T> unique_ptr<T>::unique_ptr(T* p) throw(): ptr_(p) {}

template<typename T> unique_ptr<T>::unique_ptr() throw(): ptr_(0) {}

template<typename T> unique_ptr<T>::~unique_ptr() { delete ptr_; }

template<typename T> T& unique_ptr<T>::operator*() const { return *ptr_; }

template<typename T> T* unique_ptr<T>::operator->() const throw() { return ptr_; }

template<typename T> unique_ptr<T>::operator bool() const throw() { return ptr_ != 0; }

template<typename T> T* unique_ptr<T>::get() const throw() { return ptr_; }

template<typename T> T* unique_ptr<T>::release() throw() { T* t = ptr_; ptr_ = 0; return t; }

template<typename T> void unique_ptr<T>::swap (unique_ptr& x) throw() {
    T* p = x.ptr_; x.ptr_ = ptr_; ptr_ = p;
}
```

(b) Write an application to black-box test all methods of the unique_ptr class as implemented. **[10]**

Answer:

```
void using_unique_ptr() {
    unique_ptr<int> foo(new int(10));
    unique_ptr<int> bar;
    cout << "foo: " << ((foo)? *(foo.get()): -100000000) << endl;
    cout << "bar: " << ((bar)? *(bar.get()): -100000000) << endl;
    foo.swap(bar);
    cout << "foo: " << ((foo)? *(foo.get()): -100000000) << endl;
    cout << "bar: " << ((bar)? *(bar.get()): -100000000) << endl;
    const int* p = bar.get();
    cout << "p: " << ((p)? *p: -100000000) << endl;
    cout << "bar: " << ((bar)? *(bar.get()): -100000000) << endl;
    const int *q = bar.release();
    cout << "q: " << ((q)? *q: -100000000) << endl;
    cout << "bar: " << ((bar)? *(bar.get()): -100000000) << endl;

    delete q;

    return;
}

int main() {
    using_unique_ptr();

    return 0;
}
```

3. Write the output for the following program: **[1*9=9]**

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

struct Excp: public exception {
    int data;
    Excp(int d): data(d) {
        cout << "Excp(" << data << ")" << endl; }
    ~Excp() {
         cout << "~Excp(" << data << ")" << endl; }
};

int f(int n) {
    try {
        if (0 == n) {
            throw Excp(n);
            cout << "recur for n = " << n << endl;
        }
        return f(n-1);
    }
    catch (Excp& e) {
        throw Excp((e.data == 0)? 1: -n*e.data);
    }
    cout << "param = " << n << endl;
}
```

```
int main() {
    int n = 2, r = 0;
    try {
        r = f(n);
        cout << "output = " << r << endl;
    }
    catch (Excp& e) { r = e.data; }

    cout << "result = " << r << endl;
    return 0;
}


Answer:

Excp(0)
Excp(1)
~Excp(0)
Excp(-1)
~Excp(1)
Excp(2)
~Excp(-1)
~Excp(2)
result = 2
```

4. The following code uses two types of smart pointers from the `memory` component of C++ Standard Library:

- The behaviour of `auto_ptr` is defined as:

  `auto_ptr` objects have the peculiarity of taking ownership of the pointers assigned to them: An `auto_ptr` object that has ownership over one element is in charge of destroying the element it points to and to deallocate the memory allocated to it when itself is destroyed.
  When an assignment operation takes place between two `auto_ptr` objects, ownership is transferred, which means that the object losing ownership is set to no longer point to the element (it is set to the null pointer).

  Hence, an `auto_ptr` is an *Exclusive Ownership – Destructive Copy* smart pointer.

- The behaviour of `shared_ptr` is defined as:

  Objects of `shared_ptr` types have the ability of taking ownership of a pointer and share that ownership: once they take ownership, the group of owners of a pointer become responsible for its deletion when the last one of them releases that ownership.

  Hence, a `shared_ptr` is a *Shared Ownership – Reference Counting* smart pointer.

Read the code carefully to understand the resource (memory) management by the smart pointers and the ensuing lifetime of the objects. Based on your understanding write the output from the code.

The marks for this question are as follows:

Output from Block	Marks	Remarks
`using_shared_ptr() Blk`	**0.5*4 = 2**	Write output from this block only – not the nested blocks
`auto_ptr Blk`	**0.5*10 = 5**	Some output from this block may be printed after ...`END`
`shared_ptr Blk 1`	**0.5*4 = 2**	Some output from this block may be printed after ...`END`
`shared_ptr Blk 2`	**0.5*4 = 2**	Some output from this block may be printed after ...`END`
`shared_ptr Blk 3`	**0.5*2 = 1**	Some output from this block may be printed after ...`END`
Interleaving between blocks	**3**	Some blocks are nested in others
Total	[**15**]	

Code for Q 4. Write the output.

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

struct Node {
    int data;
    shared_ptr<Node> hard;
    Node* soft;

    Node(int d=0): data(d), hard(0), soft(0)
        { cout << "A::A() Data = "
               << data << endl; }
    ~Node() { cout << "A::~A() Data = "
                  << data << endl; }
};

void Write_auto_ptr(string name,
         auto_ptr<Node>& a) {
    cout << name << ": ";
    cout << ((a.get())? "!0 = " : "0");
    if (a.get())
        cout << a->data << endl;
    else
        cout << endl;
}

void using_shared_ptr() {
    cout << "using_shared_ptr() Blk START\n";

    Node n;

    {
        cout << "auto_ptr Blk START\n";

        auto_ptr<Node> p(new Node(100));
        auto_ptr<Node> q(new Node(200));
        Write_auto_ptr("p", p);
        Write_auto_ptr("q", q);

        q = p;
        Write_auto_ptr("p", p);
        Write_auto_ptr("q", q);

        auto_ptr<Node> r(q);
        Write_auto_ptr("r", r);
        Write_auto_ptr("q", q);

        cout << "auto_ptr Blk END\n\n";
    }
```

```
    shared_ptr<Node> p1(new Node(111));

    {
        cout << "shared_ptr Blk 1 START\n";

        shared_ptr<Node> p2(new Node(222));
        p2 = 0;

        shared_ptr<Node> p3(new Node(333));
        shared_ptr<Node> p3_copy(p3);
        p3 = 0;

        cout << "shared_ptr Blk 1 END\n\n";
    }

    {
        cout << "shared_ptr Blk 2 START\n";

        Node *p4 = new Node(444);
        shared_ptr<Node> p5(new Node(555));
        p5->hard = shared_ptr<Node>(p4);
        p4->soft = p5.get();

        cout << "shared_ptr Blk 2 END\n\n";
    }

    {
        cout << "shared_ptr Blk 3 START\n";

        Node *p6 = new Node(666);
        shared_ptr<Node> p7(new Node(777));
        p7->hard = shared_ptr<Node>(p6);
        p6->hard = p7;

        cout << "shared_ptr Blk 3 END\n\n";
    }

    cout << "using_shared_ptr() Blk END\n";
    return;
}

int main() {
    using_shared_ptr();

    return 0;
}
```

Answer:

```
using_shared_ptr() Blk START
A::A() Data = 0
auto_ptr Blk START
A::A() Data = 100
A::A() Data = 200
p: !0 = 100
q: !0 = 200
A::~A() Data = 200
p: 0
q: !0 = 100
r: !0 = 100
q: 0
auto_ptr Blk END

A::~A() Data = 100
A::A() Data = 111
shared_ptr Blk 1 START
A::A() Data = 222
A::~A() Data = 222
A::A() Data = 333
shared_ptr Blk 1 END

A::~A() Data = 333
shared_ptr Blk 2 START
A::A() Data = 444
A::A() Data = 555
shared_ptr Blk 2 END

A::~A() Data = 555
A::~A() Data = 444
shared_ptr Blk 3 START
A::A() Data = 666
A::A() Data = 777
shared_ptr Blk 3 END

using_shared_ptr() Blk END
A::~A() Data = 111
A::~A() Data = 0
```

5. Write the output from the following code: **[0.5*12=6]**

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

struct product: public unary_function<double, void> { product() : prod(1), count(0) {}
        int prod; unsigned int count; void operator()(int i) { prod *= i; ++count; }
};

struct gen { gen() : item(0) {}
    int item; int operator()() { return (++item % 2)? -item: item; }
};

struct compare: public binary_function<int, int, bool> {
        bool operator()(int x, int y) { return x > y; }
};

int main() {
    vector<int> V(5);

    generate(V.begin(), V.end(), gen());
    cout << "Filled Vector is:" << endl;
    for(vector<int>::const_iterator it = V.begin(); it != V.end(); ++it)
        cout << *it << " ";
    cout << endl << endl;

    product result = for_each(V.begin(), V.end(), product());
    cout << "Product of " << result.count << " numbers is " << result.prod << endl << endl;

    sort(V.begin(), V.end(), compare());
    cout << "Sorted Vector is:" << endl;
    for(vector<int>::const_iterator it = V.begin(); it != V.end(); ++it)
        cout << *it << " ";
    cout << endl;

    return 0;
}
```

Answer:

```
Filled Vector is:
-1 2 -3 4 -5

Product of 5 numbers is -120

Sorted Vector is:
4 2 -1 -3 -5
```