



**Instructions:**

- Print your name in the space provided below.
- This examination is closed book and closed notes. No calculators or other computing devices may be used.
- Answer each question in the space provided. If you need to continue an answer onto the back of a page, clearly indicate that and label the continuation with the question number.
- If you want partial credit, justify your answers, even when justification is not explicitly required.
- There are 5 parts, priced as marked. The maximum score is 100.
- When you have completed the test, sign the pledge at the bottom of this page and turn in the test.
- **Note that failure to return this test, or to discuss its content with a student who has not taken it, is a violation of the Honor Code.**

**Do not start the test until instructed to do so!**

Name           **Solution**            
printed

**Pledge:** On my honor, I have neither given nor received unauthorized aid on this examination.

\_\_\_\_\_ signed

**I. Constructor/Destructor and other Method Invocations**

**(30 points)**

Consider the classes declared below:

```
class M {
private:
    int m;
public:
    M(int x):m(x) {
        cout << "Constructing M(" << x << ")" << endl;
    }
    M(const M& Source) {
        m = Source.m;
        cout << "Copy:      m = " << m << endl;
    }
    void operator=(const M& Source) {
        m = Source.m;
        cout << "operator=: m = " << m << endl;
    }
    ~M() { cout << "Destructing M" << endl;}
};

class N: public M {
public:
    N(int x): M(x) {cout << "Constructing N(" << x << ")" << endl;}
};

class P {
private:
    N a, b;
public:
    P():a(5), b(6) [
        cout << "Constructing P" << endl;
        a = b;
    ]
    ~P() {cout << "Destructing P" << endl;}
};
```

A. [6 points] What would be the output from the execution of the following program? Assume the necessary inclusions.

```
void main() {
    M M1(17);
}
```

**This is pretty trivial. The object M1 must be constructed and then destructed. M is a base class, so inheritance doesn't raise any additional issues. The output would be:**

**Constructing M(17)  
Destructing M**

- B. [6 points] What output would be produced by the execution of the following program that would not be produced in part A? Assume the necessary inclusions.

```
void main() {  
    M M1(17);  
    M M2 = M1;  
}
```

**This is slightly less trivial. Again, the object M1 must be constructed and then destructed. Now, however, the object M2 must also be initialized (note this is initialization and so the copy constructor is invoked). The output would be:**

```
Constructing M(17)  
Copy:      m = 17  
Destructing M  
Destructing M
```

- C. [6 points] What would be the output from the execution of the following program? Assume the necessary inclusions.

```
void main() {  
    N N1(0);  
}
```

**Here, the object N1 must be constructed and then destructed. Now, however, the object N2 is derived from the class M, and the specified constructor for M is invoked. The output would be:**

```
Constructing M(17)  
Constructing N(17)  
Destructing M  
Destructing M
```

- D. [12 points] What would be the output from the execution of the following program? Assume the necessary inclusions.

```
void main() {  
    P P1;  
}
```

**This is more complex. The object P1 contains two objects, a and b, of type N (derived from M). The constructors for those objects will be invoked prior to the execution of the body of the default constructor for P. Then, the constructor for P makes an assignment, which will use the base class assignment operator. Destruction takes place from the outside in. The output would be:**

```
Constructing M(5)  
Constructing N(5)  
Constructing M(6)  
Constructing N(6)  
Constructing P  
operator=: m = 6  
Destructing P  
Destructing M  
Destructing M
```

## II. Template and Related Issues

(15 points)

Consider the following template and main program:

```
// StackT.h
template <class T, int Size> class Stack {
private:
    int top;                // first free index
    int Sz;                // size of stack array
    T* Stk;                // stack array
public:
    Stack();                // constructs empty stack of specified Size
    void Push(T value);    // insert at top
    T Pop();                // remove and return top elem
    bool isEmpty() const;  // is stack empty?
    int getSize() const;   // return stack size
    ~Stack();              // delete stack array
};
```

```
// main.cpp
#include <iostream>
#include <string>
using namespace std;
#include "StackT.h"

const int Dim = 5;
void InitStack(Stack<string*, Dim>& myStack, string Source[], int Dim);
void PrintStack(Stack<string*, Dim> myStack, ostream& Out);

void main() {
    string Msgs[Dim] = {"One", "Two", "Three", "Four", "Five"};
    Stack<string*, Dim> myStack;

    InitStack(myStack, Msgs, Dim); // pushes contents of Msgs onto myStack in order
    PrintStack(myStack, cout);     // prints contents of myStack starting at top
}

void InitStack(Stack<string*, Dim>& myStack, string Source[], int Dim) {

    for (int Idx = 0; Idx < Dim; Idx++) {
        myStack.Push(&Source[Idx]);
    }
}

void PrintStack(Stack<string*, Dim> myStack, ostream& Out) {

    while (!myStack.isEmpty()) {
        Out << *(myStack.Pop()) << endl;
    }
}
```

A. [2 points] The code above will compile and produce output. Show the output that would be produced within the call to `PrintStack()`:

**Five**  
**Four**  
**Three**  
**Two**  
**One**

- B. [9 points] However, execution of the program will also produce a runtime error. Adding the appropriate member function to the `Stack` template can eliminate this error. Identify the problem and show the necessary function.

**The function `PrintStack()` takes a `Stack` object by value. Since there is no copy constructor for the `Stack` template, this will result in a shallow copy, meaning that the actual parameter and the formal parameter will share the same dynamically allocated array. When `PrintStack()` terminates, the `Stack` destructor will deallocate that shared array. This will result in a runtime exception when `main()` terminates and the destructor for the formal parameter is invoked. The problem is eliminated by providing a proper deep copy via a copy constructor:**

```
template <class T, int Size>
Stack<T, Size>::Stack(const Stack& Source) {

    this->top = Source.top;

    Stk = new T[Source.Sz];

    for (int Idx = 0; Idx < top; Idx++) {
        Stk[Idx] = Source.Stk[Idx];
    }
}
```

- C. [4 points] Consider the following `main()`, given the `Stack` template provided above:

```
void main() {
    Stack<string*, 100> A100;
    Stack<string*, 100> B100;
    Stack<string*, 200> A200;

    // assume initialization of A100 takes place here

    B100 = A100;           // line 1
    A200 = A100;          // line 2
}
```

Is the assignment in line 1 syntactically legal (whether logical or not)?

**Yes. A100 and B100 are of the same type, and a default (shallow) assignment is provided automatically. However, the copy will be shallow and probably undesirable.**

Is the assignment in line 2 syntactically legal (whether logical or not)?

**No. A200 and A100 are of different types, because the template parameters are different. Since the `Stack` template implementation provides no suitable assignment operator, this is not allowed.**

### III. Inheritance and Polymorphism

(30 points)

Consider the following class hierarchy and program:

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

class A {
public:
    virtual void one() = 0;
    virtual void two() {cout << "A::two()" << endl;}
    void three() {cout << "A::three()" << endl;}
    void four() {cout << "A::four()"; two();}
};

class B : public A {
public:
    void one() {cout << "B::one()" << endl;} // also virtual
    void two() {cout << "B::two()" << endl;} // also virtual
    void three() {cout << "B::three()" << endl;}
};

class C : public B {
public:
    void two() {cout << "C::two()" << endl;}
    void three() {cout << "C::three()" << endl;}
};

void main() {
    C *c = new C(); // line 1
    B *b = (B*) c; // line 2
    A *a = (A*) b; // line 3

    a->one(); // line 4
    a->three(); // line 5
    b->two(); // line 6
    c->two(); // line 7
    c->four(); // line 8
}
```

A. [5 points] What output is produced by the statement labeled line 4?

**B::one()**

**A.one() is pure virtual; reference binds to C, which inherits one() from B.**

B. [5 points] What output is produced by the statement labeled line 5?

**A::three()**

**A.three() is not virtual so there's no polymorphism here.**

C. [5 points] What output is produced by the statement labeled line 6?

**C::two()**

**A.two() is pure virtual; reference binds to C.two().**

D. [5 points] What output is produced by the statement labeled line 8?

**A::four()C::two()**

**A.four() is not virtual so reference binds to A. However, A.four() calls two(), which is pure virtual in A and hence binds to C.two().**

E. [5 points] Suppose that `main()` above were replaced by:

```
void main() {  
    C *c = new C(); // line 1  
    B b = *c;      // line 2  
  
    b.two();      // line 3  
}
```

Then...

- (a) ...the output from line 3 above would be exactly the same as that from line 6 in the original.
- (b) ...the program would not compile.
- (c) ...the program would compile, but the output from line 3 above would be different from that from line 6 in the original. (B::two() in fact)**

F. [5 points] Would it be legal (syntactically) to declare an instance of A? If not, explain why.

**No. The class A contains a pure virtual function. No such class can have an instance since there is no implementation provided for that function.**

#### IV. Miscellaneous

(15 points)

A. [5 points] The main difference between overloading and overriding is that...

- (a) ...overriding occurs in an inheritance hierarchy, whereas overloading occurs only in aggregations of classes.
- (b) ...overridden methods have the same name but different argument lists, whereas overloaded methods have the same name and argument lists.
- (c) ...overloaded methods have the same name but different argument lists, whereas overridden methods have the same name and the same argument lists.**
- (d) ...overloaded methods have the same name and argument lists, whereas overridden methods have the same name and different argument lists.

B. [5 points] Data in the protected section of a derived class...

- (a) ...can be accessed by methods in classes that are derived from it.**
- (b) ...is accessible to the base class from which it inherits.
- (c) ...is visible to users of the class and can be accessed directly.
- (d) ...is promoted to the base class.

C. [5 points] Objects of a base class...

- (a) ...cannot be created unless their derived classes have been declared.
- (b) ...have all of the public methods of their derived classes.
- (c) ...cannot be created; only objects of its derived classes may be.
- (d) ...are objects that have the data and methods defined in the base class, but none of the methods and data of its derived classes.**

**V. Polymorphic Behavior**

**(10 points)**

Assuming the class hierarchy shown on the right below, determine which code fragment shown on the left below provides the best illustration of the concept of polymorphism.

<pre>// Fragment A Dictionary *dp; Group *gp; int Size; gp = new Group; dp = (Dictionary *) gp; Size = dp-&gt;Size();</pre>	<pre>class Collection {     . . . public:     . . .     virtual int Size() = 0;     . . . };  class Group : public Collection {     . . . public:     . . .     int Size();     . . . };  class Dictionary : public Collection {     . . . public:     . . .     int Size();     . . . };</pre>
<pre>// Fragment B Collection *cp; Group *gp; int Size; gp = new Group; cp = (Collection *) gp; Size = dp-&gt;Size();</pre>	
<pre>// Fragment C Collection *cp; Group *gp; int Size; cp = new Collection; gp = (Dictionary *) cp; Size = gp-&gt;Size();</pre>	

A. [5 points] Which of the code fragments contains a syntactically illegal statement? What is that statement?

**That would be Fragment C since it declares an instance of a class, Collection, that contains a pure virtual member function.**

B. [5 points] Explain how polymorphism is illustrated by one of the other fragments.

**In Fragment A, a pointer of type Dictionary points to an object of the class Group, both of which are derived from a common base class, which contains a pure virtual member function Size(). The call dp->Size() will be resolved to the implementation of Size() for the class Group, providing polymorphic behavior.**

**In Fragment B, a pointer of type Collection points to an object of the class Group, which is derived from the base class Collection, which contains a pure virtual member function Size(). The call gp->Size() will be resolved to the implementation of Size() for the class Group, providing polymorphic behavior.**