Polytechnic Tutoring Center

Exam 2 Review - CS 2124, Spring 2021

Disclaimer: This mock exam is only for practice. It was made by tutors in the Polytechnic Tutoring Center and is not representative of the actual exam given by the CS Department.

1.

```
int somenumber = 12;        \\ line A
const int* p = &somenumber; \\ line B
somenumber = 42;            \\ line C
*p = 28;                    \\ line D
cout << *p << endl;         \\ line E
```

Given the above code, which lines would result in a compilation error?

a. line B
b. line B and line C
c. line C and line D
d. line B and line D
e. line C only
f. line D only
g. line D and line E
h. line E only

2.

```
int* arr = new int[17];
```

Given the above declaration, which of the following expressions is `arr[7]` equivalent to?

a. *(arr + 7)
b. *arr + 7
c. &arr + 7
d. &(arr + 7)
e. arr + 7
f. arr& + 7
g. arr* + 7

3. What will be the output of the following code?

```
class A {
public:
    A() { cout << “A()\n”; }
   ~A() { cout << “~A()\n”; }
};

class B : public A {
public:
    B() { cout << “B()\n”; }
    ~B() { cout << “~B()\n”; }
};

int main() {
    B b;
    cout << “Finished!” << endl;
}
```

a.
```
A()
B()
~B()
~A()
Finished!
```

b.
```
A()
B()
~A()
~B()
Finished!
```

c.
```
B()
A()
~B()
~A()
Finished!
```

d.
```
A()
B()
Finished!
~B()
~A()
```

e.
```
A()
B()
Finished!
~A()
~B()
```

4. What is the result of the below code?

```
class Foo {
public:
    Foo() {}
    void func(int x) { cout << x << endl; }
};

class Bar : public Foo {
public:
    Bar() {}
    void func() { cout << “Some Number” << endl; }
};

int main() {
    Bar b;
    b.func(17);
}
```

   a. **Compilation error**
   b. 17
   c. Some Number
   d. Runtime error

5. What is the result of the following code?

```
class A {
private:
    string name;
public:
    A(string& aString) : name(aString) {}
    A() : name() {}
    virtual void getName() const {
        cout << name << endl;
    }
};

class B : public A {
public:
    B() {}
};

class C : public B {};

int main {
    A a;        \\ line A
    B b;        \\ line B
    A* ap = &a; \\ line C
    A* bp = &b; \\ line D
}
```

a. Compilation error at line B
b. Compilation error at line C
c. Compilation error at line D
d. Runtime error at line B
e. Runtime error at line C
f. Runtime error at line D
g. **None of the above**

6. Given the same class definitions in question 5, what is the result of the following main function?

```
int main() {
    A a("a");    \\ line A
    B b("b");    \\ line B
    b.getName(); \\ line C
    a.getName(); \\ line D
}
```

a.
```
a
b
(the output)
```

b. `Compilation error at line A`
c. **`Compilation error at line B`**
d. `Compilation error at line C`
e. `Compilation error at line D`

7. Given the same class definitions in question 5, along with the below function definitions, what is the output of the program?

```
void func(A& a) { cout << “foo(A)\n”; };
void func(B& b) { cout << “foo(B)\n”; };

int main() {
    A a;
    B b;
    C c;
    func(a);
    func(b);
    func(c);
};
```

a. foo(A)
   foo(A)
   foo(A)

b. foo(A)
   foo(B)
   foo(A)

c. foo(B)
   foo(B)
   foo(B)

d. **foo(A)**
   foo(B)
   foo(B)

8. What is the result of the following?

```
class Shape {
public:
    Shape() {}
    virtual void draw() = 0;
};

class Circle : public Shape {
public:
    Circle() {}
    Void draw() { cout << “I’m a Circle” << endl; }
};

class Triangle : public Shape {
public:
    Triangle() {}
```

```
    void display() { cout << “I’m a Triangle” << endl; }
};

int main() {
    Circle c;
    Triangle t;
    c.draw();
    t.display();
}
```

a. I’m a Circle
   I’m a Triangle

b. Runtime error due to function overriding
c. Runtime error due to function overloading
d. Compilation error due to function overloading
e. Compilation error due to function overriding
f. **None of the above**

9. Given the following class definition for `Thing`, write the `operator=` function for that class.

```
class Thing {
public:
    Thing(int x) { p = new int(x); }
    Thing(const Thing& anotherThing) {
        p = new int(*anotherThing.p);
    }
    ~Thing() {
        delete p;
        p = nullptr;
    }
private:
    int* p;
};
```

```
Thing& operator=(const Thing& another) {
        if (this != &another) {
                delete p;
                p = new int(*another.p);
        }
        return *this;
}
```

10. Given a vector of `Instrument` pointers, which may contain a variety of different types of instruments (i.e. having the class `Instrument` as its base class), and given that each `Instrument` has its own "play" method defined in the base class as being pure virtual, then use polymorphism to play these instruments, each using their own derived class' "play" methods…

```
vector<Instrument*> instruments = ...;

for (Instrument* inst : instruments) {
        inst->play();
}
```

11. Write two classes, `Employer` and `Employee` or `Boss` and `Employee` (whatever floats your boat), that have the following characteristics…

    - An Employer, has a set of underlings which he may boss around
    - He should know who his Employees are
    - All Employees should know who their Employer is if they have one
    - Any Employee should be able to quit at any given time
    - Any Employer should be able to fire any of their Employees at any given time
    - An Employer should also be able to hire an Employee so long as he is not already employed
    - In the event that an Employer hires an Employee, that Employer becomes the care taker of that particular Employee, if the Employer loses that Employee, things could get messy
    - These quitting and hire functions should not fail silently
    - All Employees are created on the heap
    - Implement the big three for Employers
    - When an Employee is first born, he does not have a boss, but he is always born with a name
    - When an Employer is first born, he does not have any Employees but is always born with a name unless he is copied from another Employer
    - The main idea here is that the two classes sort of know about each other, find a way to make this possible
    - Note that Employees and Employers can have the same name but still be different. Thus, for a company to be able to differentiate between Employees, the name should not be the distinguishing factor!
    - The fact that Employees may be different despite having the same name, suggests that it may be more convenient for the Employer to store its Employees in a vector of Employee pointers. In fact, this isn't a suggestion, it is a must.

```
class Employee;

class Employer {
private:
	vector<Employee*> emps;
	string name;
public:
	Employer(const string& name);
	bool remove_employee(Employee* emp);
	~Employer();
	Employer(const Employer& other);
	Employer& operator=(const Employer& other);
	bool fire(Employee* emp);
	bool hire(Employee* emp);
};

class Employee {
private:
	Employer* boss;
	string name;
public:
	Employee(const string& aName) : name(aName) {
```

```
		boss = nullptr;
	}
	void set_boss(Employer* employer) {
		boss = employer;
	}
	const string& get_name() const {
		return name;
	}
	const Employee* get_boss() const {
		return boss;
	}
	bool quit() {
		return boss->remove_employee(this);
	}
};

Employer::Employer(const string& name) : name(name) {}
bool Employer::remove_employee(Employee* emp) {
	for (size_t i = 0; i < emps.size(); i++) {
		if (emps[i] == emp) {
			delete emps[i];
			emps[i] = emps[emps.size() - 1];
			emps.pop_back();
			return true;
		}
	}
	return false;
}
Employer::~Employer() {
	for (Employee* emp : emps) {
		delete emp;
		emp = nullptr;
	}
	emps.clear();
}
Employer::Employer(const Employer& other) {
	for (Employee* emp : other.emps) {
		emps.push_back(new Employee(emp->get_name()));
	}
	for (Employee* emp : emps) {
		emp->set_boss(this);
	}
	name = other.name;
}
Employer& Employer::operator=(const Employer& other) {
	if (this != &other) {
		for (size_t i = 0; i < emps.size(); i++) {
			delete emps[i];
			emps[i] = nullptr;
		}
		emps.clear();
		for (size_t i = 0; i < other.emps.size(); i++) {
			emps.push_back(new Employee(other.emps[i]->get_name()));
			emps[i]->set_boss(this);
		}
	}
	name = other.name;
	return *this;
```

```
}
bool Employer::fire(Employee* emp) {
	return remove_employee(emp);
}
void Employer::hire(Employee* emp) {
	if (!emp->get_boss()) {
		emps.push_back(emp);
		return true;
	}
	return false;
}
```