Name: Sundeep Kaler Net-Id: SK5437

CS1124

Spring 2015

Exam Two

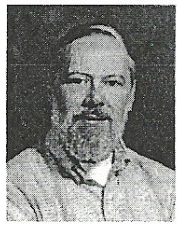
NOTE:

- There is one LONG problem at the end of the test.
- A good strategy would be to do all the short questions that you can do *quickly*,
  - then get to the LONG problem at the end of the test;
  - and finally go back through the shorter ones.

1) **DO NOT CHEAT**. (They told me I have to say that.)
2) Write CLEARLY. If we can't read it, we can't give you credit for it.
3) *Do not* tear any pages out of your Blue Book.
4) *Do not* tear any pages from this document. Be sure that you hand in *all* **19** pages of this test, including this cover sheet.
5) I didn't put any includes in white book questions.
   Assume any necessary includes were there.
6) Place your answers for questions **1–13** in this document.
7) Place your answer for **the programming question** in your Blue Book.
8) Put your name and Net ID number on the cover of your Blue Book.
9) Put your name and Net ID number as indicated on *each* page of this test.
   Please **circle** your last name. Thank you.
10) If you need "scratch" paper, use your Blue Book but cross out anything you do not want graded.
11) You are not required to write comments or #includes or using statements for any code in this test. I haven't
12) Do not begin until you are instructed to do so.
13) **Good Luck!**

Name: Sundeep Khaler Net-Id: sk5437

1. [extra credit] Who created **C?**



a. Gosling
b. Kildall
c. Wirth
d. Wall
e. Ritchie
f. Stroustrup
g. Thompson
h. van Rossum

[Questions 2 – 13 are worth four points each]

2. Given a class called `Thing` and the code

```
Thing thingOne, thingTwo;
```

What function **call** is the following line equivalent to?

```
thingTwo = thingOne;
```

a. `operator=(thingTwo, thingOne)`
b. `thingTwo.operator=(thingOne)`
c. `Thing& Thing::operator=(const Thing& rhs);`
d. Neither (a) nor (b) because it is using the `Thing` copy constructor.
e. Neither (a) nor (b) because the operator has to be overridden as a friend
f. Either (a) or (b), depending on how the programmer chose to implement the operator.
g. None of the above

lhs = rhs
operator=
- member
- not a copy
- no initialization

3. Given a vector of ints called intVec, write a "ranged for" loop, also sometimes known as a "foreach" loop, to double the values of all the elements in the vector. (Yes, you are *changing* the values in the vector.)

```
for(int& x: intVec){
    x*=2;
}
```

4. Given:

```
int* data = new int[12];
```

Pick an expression that is equivalent to: `&data[5]` -> address

a) `data*5`	d) `*data+5`	g) `&(data+5)`	j) `data+5&`
b) `&(data*5)`	e) `*(data+5)`	h) `(data+5)&`	k) `data&+5`
c) `data+5`	f) `(data+5)*`	i) `&data+5`	

address

5. Given:

```
const int x = 10;
```

Which of the following will compile?

a. `int* p = &x;`
b. `const int* q = &x;`
c. `int* const r = &x;`
d. `All of the above`
a. `None of the above`

6. What is the output of the following program:

```
class Base {
public:
    void foo(int n) { cout << "Base::foo(int)\n"; }
};
class Derived: public Base {
public:
    void foo(double n) { cout << "Derived::foo(double)\n"; }
};

int main() {
    Derived der;
    der.foo(42);
}
```

a. `Base::foo(int)`
b. `Derived::foo(double)`
c. The program does not compile
d. The program does not generate any output.
e. None of the above

Name: Sundeep Khaled Net-Id: sk5437

7. Given:

```
class Member {
public:
    Member()  {cout << 1;}
    ~Member() {cout << 2;}
};

class Base {
public:
    Base()  {cout << 3;}
    ~Base() {cout << 4;}
};

class Derived : public Base {
    Member member;
public:
    Derived()  {cout << 5;}
    ~Derived() {cout << 6;}
};

int main() {
    Derived der;
}
```

Base Member Derived
↓ ↓ ↓
3 1 5 4 2 6
construct destruct

-4

What is the output?

a. 135246
b. 135642
c. 153264
d. 153462
e. 315426
f. 315624
g. 351462
h. 351264
i. 513624
j. 513426
k. 531642
l. 531246
m. Fails to compile
n. Runtime error (or undefined behavior)

Name: Sundeep Kaler Net-Id: sk5437

8. Given:

```
class Integer {
public:
    Integer(int n) { val = n; }
private:
    int val;
};
```

What has to be added to the Integer class, so that the following will correctly display "myInt is positive" when the value in myInt is positive:

```
int main() {
    int n;
    cin >> n;
    Integer myInt(n);
    if(myInt) cout << "myInt is positive\n";
}
```

Answer: Operator bool() { return myInt>0; }

9. What is the result of compiling and running the following program?

```
class Base {
public:
    virtual void method() { cout << "Base::method\n"; }
};

class Derived : public Base {
public:
    void method() { cout << "Derived::method\n"; }
};

int main() {
    Base* bp = new Derived();
    Derived* dp = bp;
    dp->method();
}
```

a. The program compiles and runs, printing "`Base::method`"

b. The program compiles and runs, printing "`Derived::method`"

c. The program compiles and runs to completion without printing anything.

d. The program compiles and crashes when it runs.

e. The program does not compile.

f. None of the above.

10. What is the result of the following?

```
class Base {
public:
    virtual void foo() { cout << " - Base::foo()\n"; }
};
class Derived : public Base {
public:
    virtual void foo() { cout << " - Derived::foo()\n"; }
};

void func(Base& arg) {
    cout << "func(Base)";
    arg.foo();
}
void func(Derived& arg) {
    cout << "func(Derived)";
    arg.foo();
}

void otherFunc(Base& arg) {
    func(arg);
}

int main() {
    Derived d;
    otherFunc(d);
}
```

base foo → virtual → derived foo

↑ func(Base)

a. The program runs and prints:
`func(Base) - Base::foo()`

b. The program runs and prints:
`func(Base) - Derived::foo()`

c. The program runs and prints:
`func(Derived) - Derived::foo()`

d. The program runs and prints:
`func(Derived) - Base::foo()`

e. The program fails to compile

f. A runtime error (or undefined behavior)

g. None of the above

Name: Sundeep Kaler Net-Id: sk5437

11. What is the result of compiling and running the following program?

D

```
class Shape {
public:
        Shape() { display(); }
        virtual void display() { cout << "Shape"; }
};
class Square : public Shape {
public:
        void display() { cout << "Square"; }
};

int main() {
        Square sq;
}
```

virtual -> Square display

← inherits base constructor

Square() : Shape() {}

Shape::Shape()

a. The program runs and prints: Shape

b. The program runs and prints: Square

c. The program runs and prints: ShapeSquare

d. The program runs and prints: SquareShape

e. Fails to compile because Square constructor is not defined

f. Fails to compile for a reason other than (e)

g. Crashes at runtime

h. None of the above

-4

12. Given:

```
class Base { };  // Yes, the base class is empty. :-)

class Derived : public Base {
public:
    ~Derived() { cout << "~Derived\n"; }
};

int main() {
    Base* bp = new Derived;
    delete bp;
}
```

:)

goes through base, then through derived

What will be the result of compiling and running the program?

a. The program prints out:
~Base
~Derived

b. The program prints out:
~Derived
~Base

c. The program prints out:
~Base

no base destructor defined

d. The program prints out:
~Derived

e. The program fails to compile

f. The program compiles and runs but doesn't print anything

g. The compiles but crashes with no output

h. None of the above.

-4

Name: Sundeep Khaler Net-Id: sk5437

13. Given:

```
class Foo {
public:
    Foo(string s, int n = 0) { str = s; num = n; }
    void display() { cout << str << ':' << num << endl; }
private:
    string str;
    int num;
};

int main() {
    Foo thingOne("abc", 17);
    string s = "def";
    thingOne = s;
    thingOne.display();
}
```

thingOne: abc, 17
s: def, 0
thingOne = s; abc → def
17 → 0

What will be the result of compiling and running the program?

a. The program runs and prints:
```
abc:17
```

b. The program runs and prints:
```
def:17
```

c. The program runs and prints:
```
abc:0
```

d. The program runs and prints:
```
def:0
```

e. The program fails to compile

f. The program compiles and runs but doesn't print anything

g. The compiles but crashes with no output

h. None of the above.

Name: Sundeep Baker Net-Id: sk5437

Programming – Blue Book

- Place the answer to the following question in your Blue Book.
- **Comments** are **not** required in the blue book!
  However, if you think they will help us understand your code,
  feel free to add them.
- **#includes** and **using namespace** are not required in the blue book.
- Read the question *carefully!*

14. **[52 pts]** One of the most important jobs in our country is the baker. He makes all those treats that we crave and that provide us with the energy to study for (and write) exams! **You will implement a class** to represent this important national resource. (Note, you are **only** implementing the **Baker** class.)

    What do bakers do?

    - Make treats! The baker has to create treats on demand, e.g.
      `theBaker.bakes("Twinkie");`
      - Each treat will be created **on the heap** so it can have a long shelf-life.
      - The Treat class (which you are **not** writing) has
        - a constructor that takes a string which is that name of the Treat
        - an output operator that displays the name.
        - Anything else Treats have is a trade secret. (Ok, they do support copy control.)
    - Deliver the treats to a company, who will in turn package them and sell them (to us!).
      - This requires that the Baker pass the whole collection off to the bakery company. After handing the collection over, he is back to having nothing. All those treats, and in fact the container itself that held them now belongs to the vakery company. E.g.
        `aCompany.receives(theBaker.delivers()); //You are not defining!`

    Just to keep life entertaining, we will want to support copy control for our baker.

    - Deep copy, of course.
    - To keep this exam to a reasonable length, you only have to implement the **assignment operator**. Completely skip the other two functions of the Big 3.

    And naturally you should provide a reasonable **output operator** displaying him and his products. See the sample test code and output on the next page.

    And finally, let's have an **equality operator**. We will consider two bakers to be "equal" if they currently have the same number of treats.

    So, to repeat, what do *you* have to implement? **Just the Baker class!!!**

    - You don't have to worry about the bakery company or even defining the Treat class.
      (Well, worry all you like, but you are not writing either of those classes.)
    - The baker needs:
      - default constructor. To begin, the baker does **not** have a collection.
      - bakes method
      - delivers method
      - output operator
      - equality operator
      - assignment operator

Ok, so **how will you represent** all of this? You should be able to work out a good design, but let me "help" you.

- Since the baker hands over the collection of treats that he has baked, he needs to have a ***pointer*** to the collection. If the baker does not currently have any treats, e.g. when he is defined, then he also should not have a collection.
- No, don't ask me what sort of collection to use. Use whatever you like. But the baker better be free to bake and store as many treats in your container as needed. We don't know how many that will be.
- And the **collection itself needs to be on the heap**.
- When the baker bakes a treat (on the heap), if he has no place to put it, either because he just came on the job or because he just delivered his collection to a company, he will first need to get / create another collection. Where? Again, obviously on the heap.

Sample test code:

```
int main() {
    Baker fred("fred");
    cout << fred << endl;
    fred.bakes("Twinkie");
    fred.bakes("Cupcake");
    fred.bakes("Twinkie");
    fred.bakes("Twinkie");
    fred.bakes("Cupcake");
    fred.bakes("Wonderbread");
    cout << fred << endl;

    Baker joe("joe");
    cout << joe << endl;
    joe = fred;
    cout << fred << endl;  // fred won't have changed
    cout << joe << endl;   // joe should look like fred (same name, etc.)

    // Do not implement Bakery
    Bakery hostess("Hostess");
    hostess.receives(fred.delivers());
    cout << fred << endl;  // fred gave away his treats
    cout << joe << endl;   // joe still has his.  (But his name is "fred".)
}
```

Resultant output:

```
Baker: fred; No treats :-(
Baker: fred; Twinkie Cupcake Twinkie Twinkie Cupcake Wonderbread.
Baker: joe; No treats :-(
Baker: fred; Twinkie Cupcake Twinkie Twinkie Cupcake Wonderbread.
Baker: fred; Twinkie Cupcake Twinkie Twinkie Cupcake Wonderbread.
Baker: fred; No treats :-(
Baker: fred; Twinkie Cupcake Twinkie Twinkie Cupcake Wonderbread.
```

```
class Baker {
    class Treat;
private:
    string name;
    vector<Treat*>* collection;
public:
    Baker() {}
    Baker(const string& name) : name(name) {}
    void bakes(const string& treatName);
    vector<Treat*>* delivers();
    friend ostream& operator<<(ostream& os,
                               const Baker& rhs);
    bool operator==(const Baker& rhs);
    Baker& operator=(const Baker& rhs);

};

void Baker::bakes(const string& treatName) {
    if(collection == nullptr) {
        collection = new vector<Treat*>;
    }
    collection->push_back(new Treat(treatName));
}
```

collection = nullptr

-4

-1

10

```
vector<Treat*>* Baker::delivers() {
    vector<Treat*>* delivery = collection;
    collection = nullptr;
    return delivery;
}

ostream& operator<<(ostream& os, const Baker& rhs){
    os << "Baker:" << name << ": ";
    if (collection == nullptr) {
        os << "No treats :-(\n";
    }
    else{
        for(Treat* treat : *collection){
            os << *treat;
        }
        os << ",\n";
    }
    return os;
}

bool Baker::operator==(const Baker& rhs){
  return (collection->size() == rhs.collection->size());
}
```

const ✓3

nullptr? ✓1

```
Baker& Baker::operator=(const Baker& rhs){
    if (collection == nullptr) {
        collection = new vector<Treat*>*;
    }
    for (size_t i=0; i< collection->size(); i++)
        delete (*collection)[i];
    }
    collection->clear();
    for (Treat* treat: *(rhs.collection)){
        collection->push_back(new Treat(treat));
    }
    name= rhs.name;
}
    return *this;
}
if (this != &rhs){ // Check for self
```

check if rhs collection not null

-2