CSE 333 – Section 4: C++ Intro

Const & References

1) Indicate (Y/N) which lines of code below (if any) would cause compiler errors:

Code Snippet 1	Error?
`int z = 5;`	N
`const int* x = &z;`	N
`int* y = &z;`	N
`x = y;`	N
`*x = *y;`	Y

Code Snippet 2	Error?
`int z = 5;`	N
`int* const w = &z;`	N
`const int* const v = &z;`	N
`*v = *w;`	Y
`*w = *v;`	N

2) Refer to the following *poorly-written* class declaration.

```
class MultChoice {
 public:
  MultChoice(int q, char resp) : q_(q), resp_(resp) { }  // 2-arg ctor
  int get_q() const { return q_; }
  char get_resp() { return resp_; }
  bool Compare(MultChoice &mc) const;  // do these MultChoice's match?

 private:
  int  q_;     // question number
  char resp_;  // response: 'A','B','C','D', or 'E'
};  // class MultChoice
```

a) Indicate (**Y**/**N**) which *lines* of code below (if any) would cause compiler errors:

Code Snippet 1	Error?
`const MultChoice m1(1,'A');`	N
`MultChoice m2(2,'B');`	N
`cout << m1.get_resp();`	Y
`cout << m2.get_q();`	N

Code Snippet 2	Error?
`const MultChoice m1(1,'A');`	N
`MultChoice m2(2,'B');`	N
`m1.Compare(m2);`	N
`m2.Compare(m1);`	Y

b) What would you change about the class declaration to make it better? Feel free to mark directly on the class declaration above if desired. **(optional)**
Many possibilities. Importantly, make get_resp() const and make the parameter to Compare() const. Stylistically, it makes sense to add a setter method and default constructor. Could also optionally disable copy constructor and assignment operator.

Class Review:

What happens if you don't define a copy constructor? Or an assignment operator? Or a destructor? Why might this be bad? (Hint: What if a member of a class is a pointer to a heap-allocated struct?)

In C++, if you don't define any of these, a default one will be synthesized for you.

- The synthesized copy constructor does a shallow copy of all fields.
- The synthesized assignment operator does a shallow copy of all fields.
- The synthesized destructor calls the destructors of any fields that have them.

How can you disable the copy constructor/assignment operator/destructor?

Set their prototypes equal to the keyword "delete": `~SomeClass() = delete;`

When is the initialization list of a constructor run, and in what order are data members initialized?

The initialization list is run before the body of the ctor, and data members are initialized in the order that they are defined in the class, not by initialization list ordering

What happens if data members are not included in the initialization list?

Data members that don't appear in the initialization list are *default initialized/constructed* before the ctor body is executed. Including when there is **no** initialization list!

3) Classy! Finish the implementation of a `Circle` class. A `Circle` is represented by three `float` fields: the `x` and `y` coordinates of its center, and its `radius`. Fill in the constructor using initialization lists, the `get` functions and the `set` functions. See `UseCircle.cc` below `Circle.cc` to see how the `Circle` class might be used by a client.

Circle.h:

```
class Circle {
 public:
  // Default constructor, creates a circle of radius 1 at (0,0)
  Circle() : x_(0.0), y_(0.0), rad_(1.0) { }

  // Parameterized constructor
  // Takes in the x, y coordinates of the center
  // and the radius of the circle
  Circle(const float x, const float y, const float rad) :
                                              x_(x), y_(y), rad_(rad) { }

  // Copy constructor
  // Takes in a reference to another circle
  Circle(const Circle& c) : x_(c.x_), y_(c.y_), rad_(c.rad_) { }

  // Getters
  float get_x() const { return x_; }
  float get_y() const { return y_; }
  float get_radius() const { return rad_; }

  // Setters
  // Sets the center of the circle to be the given x and y coords
  void SetCenter(const float x, const float y);

  // Scales the radius of the circle by the given scale factor
  // For simplicity, negative scale factors are allowed.
  void ScaleCircle(const float scale);

 private:
  // the location of the center of a circle and its radius
  float x_, y_, rad_;
};  // class Circle
```

Circle.cc:

```
void Circle::SetCenter(const float x, const float y) {
  x_ = x;
  y_ = y;
}

void Circle::ScaleCircle(const float scale) {
  rad_ = rad_ * scale;
}
```

UseCircle.cc:

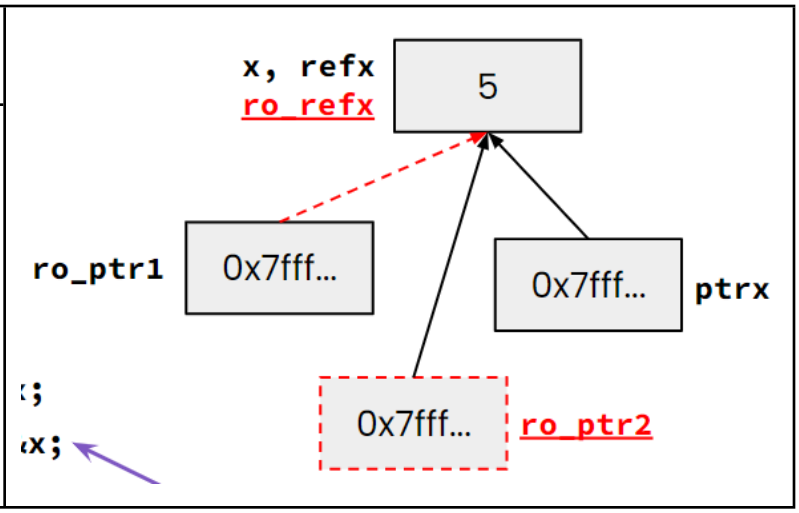
```
// Other includes and namespace usages omitted for space
#include "Circle.h"

int main(int argc, char** argv) {
  // Use default ctor
  Circle c1;

  // Use copy ctor
  Circle c2 = c1;

  // Print c1 details
  cout << "c1 is a circle of radius " << c1.get_radius();
  cout << " with its center at (" << c1.get_x() << ", ";
  cout << c1.get_y() << ")" << endl;

  // Print c2 details
  cout << "c2 is a circle of radius " << c2.get_radius();
  cout << " with its center at (" << c2.get_x() << ", ";
  cout << c2.get_y() << ")" << endl;

  // Scale c1 radius by 3
  c1.ScaleCircle(3.0);

  // Set its center to (4, 1)
  c1.SetCenter(4.0, 1.0);

  // Print c1 details
  cout << "c1 is a circle of radius " << c1.get_radius();
  cout << " with its center at (" << c1.get_x() << ", ";
  cout << c1.get_y() << ")" << endl;

  return EXIT_SUCCESS;
}
```

Bonus: Const++

a) Draw a memory diagram for the variables declared in `main`.

```
void foo(const int& arg);
void bar(int& arg);
```

```
int main(int argc, char **argv) {
  int x = 5;
  int& x_ref = x;
  int* x_ptr = &x;
  const int& ro_x_ref = x;
  const int* ro_ptr1 = &x;
  int* const ro_ptr2 = &x;
  // ...
}
```



b) When would you prefer `void func(int &arg);` to `void func(int *arg);`? Expand on this distinction for other types besides `int`.

- When you don't want to deal with pointer semantics, use references
- When you don't want to copy stuff over (doesn't create a copy, especially for parameters and/or return values), use references
- Style wise, we want to use **references for input parameters** and **pointers for output parameters**, with the output parameters declared last

c) What does the compiler think about the following lines of code:

```
bar(x_ref);                 // No issues
bar(ro_x_ref);              // Error - ro_x_ref is const
foo(x_ref);                 // No issues
```

d) How about this code?

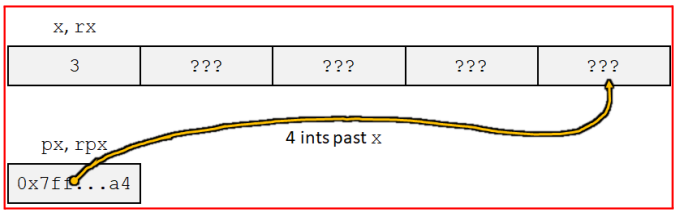
```
ro_ptr1 = (int*) 0xDEADBEEF;        // No issues
x_ptr = &ro_x_ref;                  // Error - ro_x_ref is const
ro_ptr2 = ro_ptr2 + 2;              // Error - ro_ptr2 is const
*ro_ptr1 = *ro_ptr1 + 1;            // Error - (*ro_ptr1) is const
```

e) In a function `const int f(const int a);` are the `const` declarations useful to the client? How about the programmer? What about this function needs to change to make `const` matter?

The `const` return and parameter both don't affect the client at all, since they work with copies of the parameter/return value. This enforces the programmer not to modify `a` at all. If `f` used references for the parameter/return, then it would matter to both the client and the programmer.

Bonus: What does the following program print out? Hint: box-and-arrow diagram!

```
int main(int argc, char** argv) {
  int x = 1;       // assume &x = 0x7ff...94
  int& rx = x;
  int* px = &x;
  int*& rpx = px;

    rx = 2;
  *rpx = 3;
   px += 4;

  cout << "   x: " <<    x << endl;  //   x: 3
  cout << "  rx: " <<   rx << endl;  //  rx: 3
  cout << " *px: " <<  *px << endl;  // *px: ??? (garbage)
  cout << "  &x: " <<   &x << endl;  //  &x: 0x7ff...94
  cout << " rpx: " <<  rpx << endl;  // rpx: 0x7ff...a4
  cout << "*rpx: " << *rpx << endl;  // *rpx = *px: ??? (garbage)
  return 0;
}
```



Bonus: Mystery Functions

Consider the following C++ code, which has **`___???___`** in the place of 3 function names in `main`:

```
struct Thing {
  int a;
  bool b;
};

void PrintThing(const Thing& t) {
  cout << boolalpha << "Thing:  " << t.a << ", " << t.b << endl;
}

int main() {
  Thing foo = {5, true};
  cout << "(0) ";
  PrintThing(foo);

  cout << "(1) ";
  ___???___(foo);   // mystery 1:  f2
  PrintThing(foo);

  cout << "(2) ";
  ___???___(&foo);  // mystery 2:  f3
  PrintThing(foo);

  cout << "(3) ";
  ___???___(foo);   // mystery 3:  f1, f2, f4, or f5
  PrintThing(foo);

  return 0;
}
```

Program Output:	**Possible Functions:**
(0) Thing: 5, true (1) Thing: 6, false (2) Thing: 3, true (3) Thing: 3, true	void **f1**(Thing t); void **f2**(Thing& t); void **f3**(Thing* t); void **f4**(const Thing& t); void **f5**(const Thing t);

List *all* of the possible functions (**`f1`** - **`f5`**) that could have been called at each of the three mystery points in the program that would compile cleanly (no errors) and could have produced the results shown. There is at least one possibility at each point; there might be more.

- Hint: look at parameter lists and types in the function declarations and in the calls.