Const & References

1) Consider the following functions and variable declarations - Also covered in lecture slides.

   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 &refx = x;
      int *ptrx = &x;
      const int &ro_refx = 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(refx);   // No issues
   bar(ro_refx); // Compiler error - ro_refx is const
   foo(refx);   // No issues
   ```

   d) How about this code?

   ```
   ro_ptr1 = (int*) 0xDEADBEEF; // No issues
   ptrx = &ro_refx;             // Compiler error - ro_refx is const
   ro_ptr2 = ro_ptr2 + 2;      // Compiler error - ro_ptr2 is const
   *ro_ptr1 = *ro_ptr1 + 1;    // Compiler error - (*ro_ptr1) is const
   ```
e) In a function \( \text{const int } f(\text{const int } a); \) are the \text{const} declarations useful to the client? How about the programmer? What about this function needs to change to make \text{const} matter?

The \text{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.

2) What does the following program print out? (5 min) Hint: box-and-arrow diagram!

```c
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;
}
```
3) Refer to the following poorly-written class declaration. (10 min)

```cpp
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 the snippets of code below (if any) would cause compiler errors:

<table>
<thead>
<tr>
<th>Code Snippets</th>
<th>Error?</th>
</tr>
</thead>
<tbody>
<tr>
<td>int z = 5;</td>
<td>N</td>
</tr>
<tr>
<td>const int *x = &amp;z;</td>
<td>N</td>
</tr>
<tr>
<td>int *y = &amp;z;</td>
<td>N</td>
</tr>
<tr>
<td>x = y;</td>
<td>N</td>
</tr>
<tr>
<td>*x = *y;</td>
<td>Y</td>
</tr>
<tr>
<td>const MultChoice m1(1,'A');</td>
<td>N</td>
</tr>
<tr>
<td>MultChoice m2(2,'B');</td>
<td>N</td>
</tr>
<tr>
<td>cout &lt;&lt; m1.get_resp();</td>
<td>Y</td>
</tr>
<tr>
<td>cout &lt;&lt; m2.get_q();</td>
<td>N</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code Snippets</th>
<th>Error?</th>
</tr>
</thead>
<tbody>
<tr>
<td>int z = 5;</td>
<td>N</td>
</tr>
<tr>
<td>int *const w = &amp;z;</td>
<td>N</td>
</tr>
<tr>
<td>const int *const v = &amp;z;</td>
<td>N</td>
</tr>
<tr>
<td>*v = *w;</td>
<td>Y</td>
</tr>
<tr>
<td>*w = *v;</td>
<td>N</td>
</tr>
<tr>
<td>const MultChoice m1(1,'A');</td>
<td>N</td>
</tr>
<tr>
<td>MultChoice m2(2,'B');</td>
<td>N</td>
</tr>
<tr>
<td>m1.Compare(m2);</td>
<td>N</td>
</tr>
<tr>
<td>m2.Compare(m1);</td>
<td>Y</td>
</tr>
</tbody>
</table>

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 parameter to Compare() const. Stylistically, probably makes sense to add a setter method and default constructor. Could also optionally disable copy constructor and assignment operator.
4) Mystery Functions (10 min)
Consider the following C++ code, which has __???__ in the place of 3 function names in `main`:

```cpp
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:**

<table>
<thead>
<tr>
<th>Point</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>Thing: 5, true</td>
</tr>
<tr>
<td>(1)</td>
<td>Thing: 6, false</td>
</tr>
<tr>
<td>(2)</td>
<td>Thing: 3, true</td>
</tr>
<tr>
<td>(3)</td>
<td>Thing: 3, true</td>
</tr>
</tbody>
</table>

**Possible Functions:**

<table>
<thead>
<tr>
<th>Possible Functions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>void f1(Thing t);</td>
</tr>
<tr>
<td>void f2(Thing &amp;t);</td>
</tr>
<tr>
<td>void f3(Thing *t);</td>
</tr>
<tr>
<td>void f4(const Thing &amp;t);</td>
</tr>
<tr>
<td>void f5(const Thing t);</td>
</tr>
</tbody>
</table>

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.
**Makefiles**

Makefiles are used to manage project recompilation. Project structure and dependencies can be represented as a directed acyclic graph (DAG), which a makefile can codify in a way to recursively check what sources need to be rebuilt for a specified target. The direction of the arrows in a DAG are not important (point to dependency vs. point to target) as long as you are consistent. Makefile entries are triplets of the form:

```
target:  src1 src2 ... srcN
command/commands
```

**Exercise:**

5) **Given the snippets of the following files, draw out the DAG and write a suitable Makefile.**

It should produce the executables UsePoint, UseThing, and Alone and have ‘all’ and ‘clean’ phony targets. (5 min)

```plaintext
Point.h
class Point { ... };

UsePoint.cc
#include "Point.h"
#include "Thing.h"
int main( ... ) { ... }

UseThing.cc
#include "Thing.h"
int main( ... ) { ... }

Point.cc
#include "Point.h"
// defs of methods

Thing.h
struct Thing { ... };
// full struct def here

Alone.cc
int main( ... ) { ... }
```

CFLAGS = -Wall -g -std=c++11

all: UsePoint UseThing Alone

UsePoint: UsePoint.o Point.o
g++ $(CFLAGS) -o UsePoint UsePoint.o Point.o

UsePoint.o: UsePoint.cc Point.h Thing.h
g++ $(CFLAGS) -c UsePoint.cc

Point.o: Point.cc Point.h
g++ $(CFLAGS) -c Point.cc

UseThing: UseThing.cc Thing.h
g++ $(CFLAGS) -o UseThing UseThing.cc

Alone: Alone.cc
g++ $(CFLAGS) -o Alone Alone.cc

clean:
rm UsePoint UseThing Alone *.o *~