CSE 333 – Section 4: Makefiles, C++ Intro, HW2 Overview

Welcome back to section! We're glad that you're here :)

1. Refer to the following file definitions.

Point.h

```
class Point { … };
```

Point.cc

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

UsePoint.cc

```
#include "Point.h"
#include "Thing.h"
int main( … ) { … }
```

Thing.h

```
struct Thing { … };
// full struct def here
```

UseThing.cc

```
#include "Thing.h"
int main( … ) { … }
```

Alone.cc

```
int main( … ) { … }
```

1. Draw out Point's DAG
   (The direction of the arrows is not important, but be consistent)

Write the corresponding Makefile for Point.

References

References create *aliases* that we can bind to existing variables. References are not separate variables and cannot be reassigned after they are initialized. In C++, you define a reference using: **`type& name = var`**. The '`&`' is similar to the '`*`' in a pointer definition in that it modifies the type and the space can come before or after it.

Const

Const makes a variable *unchangeable* after initialization, and is enforced at compile time.

```
const int x = 5;                // Can't assign to x
const int* x_ptr = &x;          // Can assign to x_ptr, but not *x_ptr
int* const y_ptr = &y;          // Can assign to *y_ptr, but not y_ptr
const int* const z_ptr = &z;    // Can't assign to *z_ptr or z_ptr
```

Class objects can be declared const too - a const class object can only call member functions that have been declared as const, which are not allowed to modify the object instance it is being called on.

Exercises:

2. Consider the following functions and variable declarations.

a) Draw a memory diagram for the variables declared in `main`. It might be helpful to distinguish variables that are constant in your memory diagram.

```
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`.

c) If we have functions `void foo(const int& arg);` and `void bar(int& arg);`, what does the compiler think about the following lines of code:

```
bar(x_ref);
bar(ro_x_ref);
foo(x_ref);
```

d) How about this code?

```
ro_ptr1 = (int*) 0xDEADBEEF;
ptrx = &ro_x_ref;
ro_ptr2 = ro_ptr2 + 2;
*ro_ptr1 = *ro_ptr1 + 1;
```

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?

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

```
class MultChoice {
 public:
  MultChoice(int q, char resp) {      // 2-arg ctor
    q_= q; resp_ = resp; }
  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:

Code Snippets	Error?

`const MultChoice m1(1,'A');`
`MultChoice m2(2,'B');`
`cout << m1.get_resp();`
`cout << m2.get_q();`	

Code Snippets	Error?

`const MultChoice m1(1,'A');`
`MultChoice m2(2,'B');`
`m1.Compare(m2);`
`m2.Compare(m1);`	

b) What would you change about the class declaration to make it better? Feel free to mark directly on the class declaration above.

Bonus: Const Const Const

Which of the following lines will result in a compiler error?

Code Snippets	Error?

`int z = 5;`
`const int* x = &z;`
`int* y = &z;`
`x = y;`
`*x = *y;`	

Code Snippets	Error?

`int z = 5;`
`int* const w = &z;`
`const int* const v = &z;`
`*v = *w;`
`*w = *v;`	

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;
     cout << "  rx: " <<   rx << endl;
     cout << " *px: " <<  *px << endl;
     cout << "  &x: " <<   &x << endl;
     cout << " rpx: " <<  rpx << endl;
     cout << "*rpx: " << *rpx << endl;

     return EXIT_SUCCESS;
}
```

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
  PrintThing(foo);

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

  cout << "(3) ";
  ???(foo);   // mystery 3
  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.