CSE 333 – Section 4: C++ Intro; Makefiles

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

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* xptr = &x;           // Can assign to xptr, but not *xptr
int *const yptr = &y;           // Can assign to *yptr, but not yptr
const int *const zptr = &z;     // Can't assign to *zptr or zptr
```

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:

1) **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 &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`.

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(refx);
bar(ro_refx);
foo(refx);
```

d) How about this code?

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

2) 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;
}
```

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

Code Snippets	Error?

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

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

Code Snippets	Error?

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

`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.

4) 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.

Makefiles

Makefiles are used to manage project recompilation. Project structure and dependencies can be represented as a directed acyclic graph (DAG), which a makefile encodes to recursively build the minimum number of files 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.

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( … ) { … }
```