CSE 333 Section 5 - C++ Intro, Classes, and Dynamic Memory

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* 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:

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& 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;
x_ptr = &ro_x_ref;
ro_ptr2 = ro_ptr2 + 2;
*ro_ptr1 = *ro_ptr1 + 1;
```

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 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.

Member, Non-Member, and Friends, Oh My!

	Member	**Non-member**
Access to Private Members:		
Function call (`Func`):		
Operator call (`*`):		
When preferred:		

Constructors, Destructors, what is going on?

- ***Constructor****:* Can define any number as long as they have different parameters. Constructs a new instance of the class. The *default constructor* takes no arguments.
- ***Copy Constructor****:* Creates a new instance of the class based on another instance (it's the constructor that takes a reference to an object of the same class). Automatically invoked when passing or returning a non-reference object to/from a function.
- ***Assignment Operator****:* Assigns the values of the right-hand-expression to the left-hand-side instance.
- ***Destructor***: Cleans up the class instance, *e.g.*, free dynamically allocated memory used by this class instance.

What happens if you don't define a copy constructor? Or an assignment operator? Or a destructor? Why might this be bad?

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

Exercise 3) Order the execution of the following program:

```
class Bar {
 public:
  Bar() : num_(0) { }                            // 0-arg ctor
  Bar(int num) : num_(num) { }                   // 1-arg ctor
  Bar(const Bar& other) : num_(other.num_) { }   // cctor
  ~Bar() { }                                     // dtor
  Bar& operator=(const Bar& other) = default;    // op=
  int get_num() const { return num_; }           // getter

 private:
  int num_;
};

class Foo {
 public:
  Foo() : bar_(5) { }               // 0-arg ctor
  Foo(const Bar& b) { bar_ = b; }   // 1-arg ctor
  ~Foo() { }                        // dtor

 private:
  Bar bar_;
};

int main() {
  Bar b1(3);
  Bar b2 = b1;
  Foo f1;
  Foo f2(b2);
  return EXIT_SUCCESS;
}
```

Number the following starting with 1.

Each method may be called more than once (*i.e.*, you can put multiple numbers on the same line).

_____ Bar 0-arg ctor

_____ Bar 1-arg ctor

_____ Bar cctor

_____ Bar op=

_____ Foo 0-arg ctor

_____ Foo 1-arg ctor

_____ Foo dtor

_____ Bar dtor

Dynamically-Allocated Memory: New and Delete

In C++, memory can be heap-allocated using the keywords "`new`" and "`delete`". You can think of these like `malloc()` and `free()` with some key differences:

- Unlike `malloc()` and `free()`, `new` and `delete` are operators, not functions.
- The implementation of allocating heap space may vary between `malloc` and `new`.

New: Allocates the type on the heap, calling the specified constructor if it is a class type. Syntax for arrays is "`new type[num]`". Returns a pointer to the type.

Delete: Deallocates the type from the heap, calling the destructor if it is a class type. For anything you called "`new`" on, you should at some point call "`delete`" to clean it up. Syntax for arrays is "`delete[] name`".

Just like baking soda and vinegar, you shouldn't mix `malloc`/`free` with `new`/`delete`.

Exercise 4) Memory Leaks

```
class Leaky {
 public:
  Leaky() { x_ = new int(5); }
 private:
  int* x_;
};

int main(int argc, char** argv) {
  Leaky** dbl_ptr = new Leaky*;
  Leaky* lky_ptr = new Leaky();
  *dbl_ptr = lky_ptr;
  delete dbl_ptr;
  return EXIT_SUCCESS;
}
```

What is leaked by this program? How would you fix the memory leaks?

Exercise 5) Identify the memory error with the following code.

```
class BadCopy {
 public:
  BadCopy()  { arr_ = new int[5]; }
  ~BadCopy() { delete [] arr_; }
 private:
  int* arr_;
};

int main(int argc, char** argv) {
  BadCopy* bc1 = new BadCopy;
  BadCopy* bc2 = new BadCopy(*bc1); // cctor

  delete bc1;
  delete bc2;

  return EXIT_SUCCESS;
}
```

Hint: Draw a memory diagram. What happens when `bc1` gets deleted?