# CSE 333 Section 5 - C++ Classes, 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
```

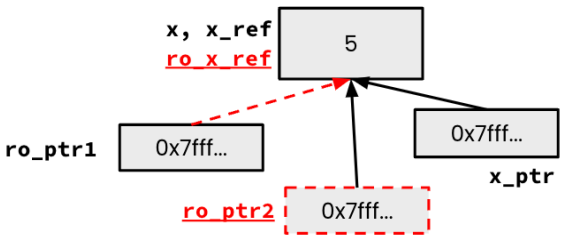
Class objects can be declared const too - a const class object can only call member functions that have been declared as const, which are not allowed to modify the object instance it is being called on.

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`.

- 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) 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);                          // 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
```

**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');`<br>`MultChoice m2(2,'B');`<br>`cout << m1.get_resp();`<br>`cout << m2.get_q();` | N<br>N<br>Y<br>N |

| **Code Snippets** | **Error?** |
|---|---|
| `const MultChoice m1(1,'A');`<br>`MultChoice m2(2,'B');`<br>`m1.Compare(m2);`<br>`m2.Compare(m1);` | N<br>N<br>N<br>Y |

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

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.

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

| | Member | Non-member |
|---|---|---|
| **Access to Private Members:** | Always | • Through getters and setters<br>• Through `friend` keyword (do not use unless needed) |
| **Function call (Func):** | `obj1.Func(obj2)` | `Func(obj1, obj2)` |
| **Operator call (*):** | `obj1 * obj2` | `obj1 * obj2` |
| **When preferred:** | • Functions that *mutate* the object<br>• "Core" class functionality | • *Non-mutating* functions<br>• Commutative functions<br>• When the class must be on the right-hand side |

## *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, *i.e.* 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? (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!

**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).

6 Bar 0-arg ctor

1,4 Bar 1-arg ctor

2 Bar cctor

7 Bar op=

3 Foo 0-arg ctor

5 Foo 1-arg ctor

8,10 Foo dtor

9,11,12,13 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**

```
#include <cstdlib>

class Leaky {
 public:
  Leaky() { x_ = new int(5); }
  ~Leaky() { delete x_; }  // Delete the allocated int
 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;
  delete lky_ptr;  // Delete of dbl_ptr doesn't delete what lky_ptr
points to
  return EXIT_SUCCESS;
}
```

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

Deleting the `dbl_ptr` doesn't automatically delete what the pointer points to. Have to also delete `lky_ptr` and then create a destructor that deletes the allocated int pointer `x_`.

**Exercise 5) Identify the memory error with the following code. Then fix it!**

```
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);  // BadCopy's cctor

  delete bc1;
  delete bc2;

  return EXIT_SUCCESS;
}
```

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

The default copy constructor does a shallow copy of the fields, so `bc2`'s `arr_` points to the same array as `bc1`'s `arr_`. When `bc1` gets deleted, so does its `arr_`. But this `arr_` is the same one `bc2`'s `arr_` points to, so when `bc2` gets deleted, its `arr_` has already been deleted, leading to an invalid delete (similar to a double `free()`).