Lab Assignment 4: Classes

Dr Martin Porcheron

You have two weeks to complete this lab assignment—you must have it marked on Zoom by **26th February 2021**, but I would recommend you try and complete this as soon as possible. This assignment is worth 20 marks.

This lab task involves using classes in C++ to encapsulate data and provide protected access to its member variables. For this you will implement a class and necessary constructors, destructor, getters, and setters to represent a Cat, including the name of the Cat and a number of lives the Cat has remaining.

For this lab you will not need to use dynamic allocation (`malloc`/`free`, `new`/`delete`, `new []`/`delete []`) **inside** the `Cat` class.

It is not valid to use the line `using namespace std;` or any directive to that effect anywhere in your solution.

I recommend you keep all your lab assignments in a new directory. If working on the lab machines, create this within your home directory for CSC371. Keep each lab assignment in a separate directory.

If you are using **Windows Subsystem for Linux**, I recommend you install `build-essential` which includes gcc and the library for C++ code. Do this with the commands `apt-get update` followed by `apt-get install build-essential`.

Note: Do not copy and paste code from the PDF as it will contain non-standard characters and will not compile.

This lab assignment was originally written by Dr Joss Whittle.

Task 1: A class with accessors/getters (4 marks)

In this exercise, you are going to write and instantiate a `Cat` class.

1. Download this source code from the the Canvas assignment page for Lab Assignment 4 or alternatively, create a file, *task1.cpp*, and type the following code into it:

```
#include <iostream>
#include "cat.h"

int main(int argc, char* argv[]) {
    Cat a("Garfield");

    std::cout << a.get_name() << " has "
              << a.get_lives() << " lives." << std::endl;

    Cat b("Mog" , 10);
    std::cout << b.get_name() << " has "
              << b.get_lives() << " lives." << std::endl;

    return 0;
}
```

2. Create a pair of files *cat.h* and *cat.cpp*. Place and use header guards in the header file (or use `#pragma once`—look this up in C++ documentation on how to use it correctly).
3. In *cat.h*:
   - Declare a class `Cat` with two `private` member variables, a `std::string` called `name` and an `unsigned int` called `lives`. `std::string` is a class type that can be found in the C++ `<string>` header—look this up in C++ documentation if you need help on using the class.
   - Declare 2 `public` constructors. The first constructor accepts an `std::string` and assigns it to the `name` member variable while initialising the number of `lives` to be 9. The second constructor accepts both an `std::string` and an `unsigned int` and initialises both member variables to those values.
   - Declare a `public` destructor.
   - Declare `public` getter methods for the name and lives member variables. These getters should return "constant references" to the underlying member variables to ensure that simply "getting" their values does cause a temporary duplicate to be made.
4. In *cat.cpp*:
   - Implement the two constructors with their behaviour as previously described.
     - You should make use of the "constructor initialisation list" syntax to ensure that by the time the body of the constructors are entered, the member variables have already been initialised with the values passed as parameters (or defaulted in the case of `lives` = 9).
     - In the constructor bodies you should use `std::cout` from the `<iostream>` header to print out "Constructor called with name..." and "Constructor called with name and lives..." respectively. These print statements when executed will demonstrate the order of implicit function execution that occurs when the program is run.
   - Implement the destructor. As this class does not own any manually allocated resources the destructor does not need to do anything to ensure the class is safely destructed. In the destructor body you should use `std::cout` to print out "Destructor called..." to demonstrate at runtime the order of execution for the destructors.
   - Implement the two getter methods with their behaviour as previously described.

5. Compile and link your program with your compiler.

   You can compile a C++ program using GCC directly if you include the linker flag `-lstdc++` to link the compilation against the C++ standard library. Alteratnively, you can use the G++ wrapper command.

   In either case you should use the compilation flag `--std=c++11` to ensure that the C++11 standard is used by the compiler (C++14 or C++17 would also work fine here, but we do not need the features these standards add for this lab). Linker flags must be listed after the files you would like to compile are listed.

   ```
   $ gcc --std=c++11 task1.cpp cat.cpp -lstdc++ -o task1
   ```

   or

   ```
   $ g++ --std=c++11 task1.cpp cat.cpp -o task1
   ```

Expected output

If you run your program using the command:

```
$ ./task1
```

...you should get the output:

```
Constructor called with name...
Garfield has 9 lives.
Constructor called with name and lives...
Mog has 10 lives.
Destructor called...
Destructor called...
```

Task 2: A class with mutators/setters (6 marks)

In this exercise you are going to modify your instance of `Cat`.

1. Copy your task 1 file and name this new version *task2.cpp*.
2. In *cat.h*:
   - Declare `public` setter methods for the `name` and `lives` member variables. These setters should return `void` and should accept "constant references" to a `std::string` and `unsigned int` respectively.
3. In *cat.pp*:
   - Implement the two setter methods. When called each method should assign the value of its parameter to the underlying member variable. Both functions should print a message to `std::cout` to state they have been executed to highlight the flow of execution, printing "set_name called..." and "set_lives called..." respectively.
   - When implementing `set_lives` you should implement the following logic/behaviour to both protect the lives member variable from taking on invalid values and also modify the name value under specific circumstances.

     Specifically:

     – If `set_lives` is called with a value that is greater than or equal to the current lives the cat has, then nothing should occur and the function should just return. That is, a cat can lose lives, but cannot gain lives.

     – If the cat currently has a positive number of lives and `set_lives` is called setting the lives to `0`, then the name of the cat should be modified to become "The Cat formerly known as X", where X is the current name of the cat. To implement this functionality you should call both `get_name` and `set_name`. The + operator can be used to concatenate an std::string with a string literal or another `std::string`.
4. In *task2.cpp*:
   - Extend the code to call the following operations on the second cat, who is currently named "Mog" and has 10 lives. After each enumerated operation print the name and lives of the cat again to demonstrate how it has changed.

     (a) Set the cat's lives to be 42 (Mog currently has 10 lives so this should not have any effect)

     (b) Set the cat's name to "Prince"

     (c) Set the cat's lives to 8 (Prince currently has 10 lives so this should succeed)

     (d) Set the cat's lives to 0 (this should change Prince's name)

     (e) Set the cat's lives to 0 again (this not change the cat's name a second time)
5. Compile and link your program with the GCC compiler:

   ```
   $ gcc --std=c++11 task2.cpp cat.cpp -lstdc++ -o task2
   ```

   or

   ```
   $ g++ --std=c++11 task2.cpp cat.cpp -o task2
   ```

Expected output

If you run your program using the command:

```
$ ./task2
```

...you should get the output:

```
Constructor called with name...
Garfield has 9 lives.
Constructor called with name and lives...
Mog has 10 lives.
set_lives called...
Mog has 10 lives.
set_name called...
Prince has 10 lives.
set_lives called...
Prince has 8 lives.
set_lives called...
set_name called...
The Cat formerly known as Prince has 0 lives.
set_lives called...
The Cat formerly known as Prince has 0 lives.
Destructor called...
Destructor called...
```

Task 3: A clowder of cats (10 marks)

In this exercise, we are going to replace our test data with data imported accepted from the command line

1. Create a new file, *task3.cpp*, with a `main` method.
2. Your program for task 3 should accept a variable number of command line parameters representing $name_0, lives_0, name_1, lives_1, ..., name_{n-1}, lives_{n-1}$. Your program must therefore dynamically allocate at runtime an array of `Cat` objects, and the enumerate over them, setting each cat to its expected values.

   If the number of parameters passed on the command line is not an even number, then the last `Cat` in the sequence should have 9 lives.

   You will have to look up the standard function to convert a `string` to an `int` in C++. When searching, I recommend using results from either cppreference.com or cplusplus.com. I prefer cppreference.com, but its search functionality is ppor (where as cplusplus.com uses Google and is quite good).
3. Calculate the number of cats to be parsed from the number of command line parameters given and store this in a constant variable within the main method.
4. In order to allocate an array of `Cat` objects using the C++ `new []` and `delete []` allocator, the `Cat` class must be default constructable. In *cat.h* declare a default constructor which takes no parameters, in addition to the two existing constructors.
5. In *cat.cpp* implement the default constructor. A default constructed `Cat` is naturally named "Tom" and has 9 lives.

   In the constructor body you should use `std::cout` from the `<iostream>` header to print out "Default constructor called...".
6. In *task3.cpp* allocate the correct number of `Cat` elements (which implicitly default constructs each of them) and then iterate over them assigning their values to be those parsed from the command line arguments.

   You can do this in two ways, either by calling the setter methods of each `Cat`, or by invoking the "move assignment" operator (which we'll be covering in lecture 10) to assign a locally constructed `Cat` within the loop (check the documentation online!).

   Note that if you use the setter methods directly you will not be able to update a default constructed Cat to have more than 9 lives. It is therefore preferable to use the move-assignment operator, although you will not lose any marks for simply using the setter methods we previously defined.
7. Iterate over the clowder and print the name and lives of each `Cat` as we have done in the previous tasks.
8. Finally, ensure that you safely deallocate the clowder before the program terminates
9. Compile your program:

   ```
   $ g++ --std=c++11 task3.cpp cat.cpp -o task3 -o task3
   ```

   or:

   ```
   $ gcc --std=c++11 task3.cpp cat.cpp -lstdc++ -o task3 -o task3
   ```

Expected output

If your run your program using the command:

```
$ ./task3 Tom 5 Dick 6 Harry 42
```

...you should get the output:

```
Allocating clowder of size 3
Default constructor called...
Default constructor called...
Default constructor called...
Constructor called with name and lives...
Destructor called...
Constructor called with name and lives...
Destructor called...
Constructor called with name and lives...
Destructor called...
Tom has 5 lives.
Dick has 6 lives.
Harry has 42 lives.
Destructor called...
Destructor called...
Destructor called...
```

If your run your program using the command:

```
$ ./task3 Lister 1 Holly 6000 Kryten 64 "The Cat"
```

…you should get the output:

```
Allocating clowder of size 4
Default constructor called...
Default constructor called...
Default constructor called...
Default constructor called...
Constructor called with name and lives...
Destructor called...
Constructor called with name and lives...
Destructor called...
Constructor called with name and lives...
Destructor called...
Constructor called with name...
Destructor called...
Lister has 1 lives.
Holly has 6000 lives.
Kryten has 64 lives.
The Cat has 9 lives.
Destructor called...
Destructor called...
Destructor called...
Destructor called...
```

Bonus Task: Converting `intvector` to a class (10 marks)

As a bonus activity, you should convert the `intvector` code we developed in Lecture 7 into C++ code with a class. You can download the code from the Lecture 7 webpage on Canvas.

I strongly recommend this as a useful way to practice getting used to C++. Remember, your coursework and your exam will require you to demonstrate strong C++ skills. This is a nice opportunity to get some practice.

We have not yet covered how to implement copy constructor or copy assignment operators in C++, thus I do not expect you to implement this in your class (but feel free to try and implement these). You may want to stick to `malloc()`/`realloc()`/`free()` inside your `intvector` class for simplicity.

Some more hints:

- Where we previously used `NULL`, you should use the C++ 'pointer literal' `nullptr` instead
- Where we previously used the pointer `iv` in each function, you can use the pointer `this` inside the function
- You do not need to call the destructor inside `resize()` anymore
- For printing, you should overload the << operator using this function prototype in your class declaration:

  ```
  friend std::ostream& operator<<(std::ostream &os, const intvector &iv)
  ```

  For the implementation of this function, you need to output information to the `std::ostream` (standard output stream) reference passed in to the function. Note that in the implementation, this function is *not* a member of your `intvector` class) and is simply an overloaded function in the global scope.

Bonus marks are awarded, and will be added to your final lab mark for the module. The cumulative total mark for all the labs in this module is capped at 100.