Lab assignment 5: Inheritance

Dr Martin Porcheron

You have two weeks to complete this lab assignment—you must have it marked/signed off by a lab helper in a timetabled lab session. You must get this assignment marked by **4th March 2022** but I recommend you try and complete this as soon as possible. This assignment is worth 20 marks.

This lab task involves using C++ inheritance to create a hierarchy over 2D shapes.

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.

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

Most of this lab assignment was originally written by Dr Joss Whittle.

Task 1: Creating the base class and a `Circle` (5 marks)

In this exercise, you are going to write a base virtual Shape class and a `Circle` derived class.

1. Download this source code from the Canvas assignment page for lab assignment 5or alternatively, create a file, *task1.cpp*, and type the following code into it:

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

   int main(int argc, char *argv[]) {
       Shape *x = new Circle(0, 0, 1); // x, y, r

       std::cout << x->to_string() << std::endl;
       x->centre_at(2, 3);

       std::cout << x->to_string() << std::endl;

       delete x;
       return 0;
   }
   ```

2. Create a file *Shape.h* and populate it with a class declaration for a type called Shape and the function stubs for a constructor and virtual destructor for the Shape class.

   - Shape should not have any member variables and should have pure virtual functions called:
     `virtual std::string to_string() const = 0;` and
     `virtual void centre_at(double x, double y) = 0;`
   - These functions should not be implemented in Shape and should be implemented only in the derived classes.

3. In *Shape.h* and *Shape.cpp* declare and implement a class `Circle` which inherits from Shape and provides implementations for `to_string()` and `centre_at()`. `Circle` should have three member fields that are doubles named `x`, `y` (for position), and `r` (for radius). The constructor should take a value for each of these three member fields.

4. The function `to_string()` should make use of `std::stringstream` and `std::string` to construct a C++ `string` object that prints the following string with the correct values:

   ```
   Circle centred at (0, 0) with radius 1...
   ```

   - `std::stringstream` can be found in the `<sstream>` header.
   - You can construct a `std::string` using a `std::stringstream` object. Once the correct data has been put into the stream we can convert it to a `std::string` value by using the `.str()` member function of the `std::stringstream`.
   - For example:

     ```
     std::stringstream sstr;
     sstr << "Hello world. " << 42 << std::endl;
     std::string str = sstr.str(); // "Hello world. 42"
     ```

5. The function `centre_at(x, y)` should move the x, y coordinate of the circle to the given location. For a circle this is very straightforward.

   Bear in mind, re-centring a shape should never change its size or side lengths.

6. From the command line compile and link your program with the GCC 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, or 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 Shape.cpp -lstdc++ -o task1
```

or

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

Expected output

If you run your program using the command:

```
$ ./task1
```

...you should get the output:

```
Circle centred at (0, 0) with radius 1...
Circle centred at (2, 3) with radius 1...
```

Task 2: Deriving a `Rectangle` (5 marks)

In this exercise you are going to modify your existing code to implement a `Rectangle`.

1. Copy your task 1 file and name this new version *task2.cpp*.
2. In *Shape.h* and *Shape.cpp* declare and implement a class `Rectangle` which inherits from `Shape` and provides implementations for the two pure virtual functions.
   - `Rectangle` should have four member fields `doubles` `x0`, `y0` and `x1`, `y1` representing the *top-left* and *bottom-right* corners of the rectangle.
   - `Rectangle` should not store member variables for its width and height.
   - The function `to_string()` should print (with the correct values...)

     ```
     Rectangle at [(0, 1), (1, 0)] with width 1, height 1...
     ```

   - The function `centre_at(x, y)` should move the `x0`, `y0` and `x1`, `y1` coordinates of the rectangle to appropriate locations so that **the centre of the rectangle is at the desired coordinate**. You'll have to calculate the centre of the rectangle from the two sets of coordinates you have. To do this, get the width and height of the rectangle from the coordinates and then divide both values by 2!
3. In *task2.cpp*, write code to instantiate a `Rectangle` object as a `Shape` pointer and demonstrate the `centre_at` and `to_string` functions work as intended by starting the rectangle at $(0, 1), (1, 0)$, printing the details and then re-centring the rectangle at $(-2.5, 2.5)$.
4. From the command line compile and link your program with the GCC compiler:

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

   or

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

Expected output

If you run your program using the command:

```
$ ./task2
```

...you should get outputs similar to:

```
Rectangle at [(0, 1), (1, 0)] with width 1, height 1...
Rectangle at [(-3, 3), (-2, 2)] with width 1, height 1...
```

Task 3: Deriving a `Triangle` (10 marks)

In this exercise you are going to modify your existing code to implement a `Triangle`.

1. Copy your task 2 file and name this new version *task3.cpp*.
2. In *Shape.h* and *Shape.cpp* declare and implement a class `Triangle` that internally stores 6 double variables `x0`, `y0`, `x1`, `y1`, and `x2`, `y2`.
   - When implementing the `centre_at(x, y)` function you should use the *Triangle Centroid* to position the shape. *Triangle Centroid* is defined as the average location of the x and y of each corner of the triangle.

     Once you have the centroid of the current triangle, subtract x and y from it to give you the size of the translation (i.e. how much the triangle is due to move in the x and y directions). Apply these differences to each coordinate.
   - `Triangle` should not store member variables for its edge lengths.
   - The function `to_string()` should print (with the correct values...)

     ```
     Triangle at [(0, 0), (1, 1), (0, 1)] with side lengths 1, 1, and
     1.41...
     ```
   - To calculate the side lengths of a triangle, look up how to derive them from the Pythagorean Theorem.
   - Hint: for both calculating the centroid and the side lengths, I recommend you spend some time working this out on paper or with an online calculator first. The calculations are not particularly tough, but if you haven't done maths like this in a while, they can take some time to figure it out.

     For re-centring the triangle, you will need to add the 3 *x*-coordinates together for the triangle and divide by 3. Subtracting the x value passed into the `centre_at` function will give you the difference you need to apply to each of the triangle's x coordinates. Repeat the same for the y coordinates.
3. In *task3.cpp*:
   - Allocate an array of *10* Shape pointers and populate the array with mixed instances of `Circle`, `Rectangle`, and `Triangle` with different parameters (you can pick your own values):

     ```
     Shape **xs = new Shape*[10];
     xs[0] = new Circle(0, 0, 1);
     // ...
     ```
   - Loop over the elements of shape array and call the `to_string()` function to show the current state of each shape.
   - Loop over the elements of the Shape array and call the `centre_at(x, y)` functions to move each to a different location (again, it's your choice where you move the shape to). Use the `to_string()` function to demonstrate each shape has been correctly moved.
   - After you have looped over the elements in the array, make sure you deallocate each of the Shape objects individually and that you also deallocate the array of Shape pointers.
4. Compile your program:

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

   or:

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

Expected output

If your run your program using the command:

```
$ ./task3
```

...you should get outputs similar to (note, you should be coming up with your own test data):

```
Circle centred at (0, 0) with radius 1...
Rectangle at [(0, 1), (1, 0)] with width 1, height 1...
Triangle at [(0, 0), (1, 1), (0, 1)] with side lengths 1, 1, and
1.41...

Circle centred at (2, 3) with radius 1...
Rectangle at [(-3, 3), (-2, 2)] with width 1, height 1...
Triangle at [(4, -3), (5, -2), (4, -2)] with side lengths 1, 1, and 1.41...
```

Bonus Task: Deriving a variadic `Polygon`

As a bonus activity, let's now try and generalise what you've built into a new `Polygon` class that takes a variable number of sides. We can implement this as a varidiac function. These are functions that take a variable number of arguments (e.g., like `printf()`). This is a feature from the C language that we are going to mix into our C++ code, and demonstrates how C++ builds upon the C language. It is implemented using the header `cstdarg`. C++ has its own mechanism for variadic constructors using template metaprogramming (we will look at template metaprogramming in a later lecture).

Add this code for a `Coordinates` struct and `Polygon` class to your *Shape.h*:

```
struct Coordinates {
  double x, y;
};

class Polygon : public Shape {
protected:
  Coordinates **coordinates;
  const unsigned int num_sides;
  virtual double get_side_length(const unsigned int &side) const;

public:
  Polygon(const unsigned int num_coords, ...);
  virtual ~Polygon();
  virtual std::string to_string() const;
  virtual void centre_at(const double x, const double y);
};
```

On the member variables:

- In `Polygon`, we'll store an array of pointers to `Coordinates` objects stored in the heap. As we progress through this module, we will see easier ways to store data inside our objects, building upon the ideas of abstraction and encapsulation using Standard Library containers.
- `num_sides` is `const`, so you will have to set it using a member initialisation list, calculating it from `num_coords` passed into the constructor

On the constructor:

- You'll note the three ellipsis in the constructor for `Polygon`. This says we are accepting a variable number of arguments.
- In more recent versions of C++ (including C++11), we can only pass primitive types such as `int` in varidadic functions. Therefore, you should construct your `Polygon` object as such, passing in the number of coordinates, e.g. 8 for a rectangle or square, and then 8 `double`s where the first two correspond to the first x and y coordinates, the next two correspond to the second x and y coordinates, and so on. You'll need to calculate the number of sides from this input.

  ```
  Polygon myRectangle(8, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0);
  ```

- You'll also need to initialise the `coordinates` member variable either in the initialisation list or alternatively but less ideal, like so:

  ```
  this->coordinates = new Coordinates*[this->num_sides];
  ```

  Generally speaking though, we should try to keep all our initialisation in the initialisation list for consistency.
- Check out the CPPRef documentation for variadic functions, including the functions `va_start`, `va_arg` (which you need to get at a specific argument), and `va_end`. Summarily, you need the following at the start of your constructor:

```
va_list args;
va_start(args, num_coords);
```

- Loop over the constructor parameters, retrieving the variable argument using the function `va_arg(args, double);`. As you go, construct new instances of `Coordinates` using the aggregate initialisation syntax, e.g.:

  ```
  this->coordinates[i] = new Coordinates{ 1.0, 1.2 }; // set x to 1.0
                                                      // and y to 1.2
  ```

  Read up on aggregates in C++ and where you can use this syntax.

- At the end of the constructor, be sure to use `va_end(args);`.

Other tips:

- In the destructor, be sure to `delete`/`delete []` anything you have created on the heap
- Note with our `Polygon` class, in comparison to the specialised rectangle above, you'll have to provide the four sets of coordinates instead of the two we previously provided. For calculating the side length, you'll have to look at the coordinates and use Pythagorean Theorem to calculate the desired side length. For `to_string()` and `centre_at()` you should base your code on the code you've previously written (e.g. for the `Triangle` class).
- As we're using floating point numbers, we're likely to end up with some ugly outputs. To ensure we print in fixed decimal notation, and with only two decimal places, use the following code to set the precision for a given `stream` in the `to_string()` function (from the `iomanip` library):

  ```
  stream << std::setprecision(2) << std::fixed;
  ```

You can test your code by comparing it to the specific implementations from Tasks 1–3 (note that your side lengths may be in a different order, that's OK and to be expected).

```
Rectangle:
Polygon at [(0.00, 0.00), (0.00, 1.00), (1.00, 1.00), (1.00, 0.00)]
with side lengths 1.00, 1.00, 1.00, 1.00...
Polygon at [(-3.00, 2.00), (-3.00, 3.00), (-2.00, 3.00), (-2.00, 2.00)]
with side lengths 1.00, 1.00, 1.00, 1.00...

Triangle:
Polygon at [(0.00, 0.00), (1.00, 1.00), (0.00, 1.00)] with side lengths
1.41, 1.00, 1.00...
Polygon at [(4.00, -3.00), (5.00, -2.00), (4.00, -2.00)] with side
lengths 1.41, 1.00, 1.00...
```