University of Cape Town
Department of Computer Science

CSC3022F
Tutorial 4: Unit Testing & the Big 6

March, 2023

Before we start this tutorial, make sure to download the 'Tutorial_4.tar.gz' archive from the 'Tutorial 4' Assignment tab on Vula. You **will** need the 'catch.hpp' file for the unit testing component of this tutorial. Once you have downloaded and extracted the contents of the archive, go ahead and open up 'driver_test.cpp'.

1 Part 1: Unit Testing:

As you know from previous years, unit testing is an important part of the development cycle. You will likely be asked to unit test some of your code at some point and this part of the tutorial serves as a brief introduction on how to go about doing that in C++. Catch or 'catch.hpp' is a modern C++ native, header-only, test framework and is what will be using. To use 'catch.hpp', we simply include it by adding the following code to our 'driver_test.cpp':

```
#define CATCH_CONFIG_MAIN  // This tells catch to provide a main() function.
Only do this in one cpp file
#include "catch.hpp"
```

Now that we have included our unit testing framework, we can go ahead and create our first unit test. We will do this for the Factorial function included in 'driver_test.cpp'. In Catch2, the general form for a unit test is:

```
TEST_CASE( TEST_NAME )
{
    REQUIRE( TEST_1 );
    REQUIRE( TEST_2 );
    ...
    REQUIRE( TEST_N );
}
```

where 'TEST_NAME' is the name of your test and 'TEST_1', 'TEST_2', ... and 'TEST_N' are boolean values that are returned as a result of you testing your code. A unit test for our 'Factorial' function would look as follows:

```
TEST_CASE ("Factorial Unit Test")
{
    REQUIRE(factorial(0) == 1);
    REQUIRE(factorial(1) == 1);
    REQUIRE(factorial(2) == 2);
    REQUIRE(factorial(3) == 6);
    REQUIRE(factorial(10) == 3628800);
}
```

Note: You may see some compiler warnings but it should not effect the compilation process.

Now if you go ahead and run the code and you should see the following output:

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
driver.exe is a Catch v1.0 b31 host application.
Run with -? for options

-------------------------------------------------------------------------------
Factorial Unit Test
-------------------------------------------------------------------------------
driver_test.cpp:9
.....................................................................

driver_test.cpp:11: FAILED:
  REQUIRE( factorial(0) == 1 )
with expansion:
  0 == 1

===============================================================================
1 test case - failed (1 assertion - failed)
```

As you can see, our 'factorial' function has a bug. It does not take the value 0 into consideration and thus returns an incorrect result. Let's fix that by changing our 'factorial' function like so:

```
unsigned int factorial(unsigned int x)
{
    return x > 1 ? factorial(x - 1) * x : 1;
}
```

NOTE: If you have never seen the '?' operator before, it is called the conditional(or ternary) operator and it allows us to simplify 'if-else' statements. The general form for the '?' operator is:

```
(expression 1) ? expression 2 : expression 3
```

where 'expression 1' returns a boolean (true/false) and 'expression 2' and 'expression 3' are evaluated depending on the result of 'expression 1'. If 'expression 1' returns 'true', 'expression 2' is evaluated. If 'expression 1' returns 'false', 'expression 3' is evaluated instead.

Now if you go ahead and compile and run your program again you should see that all of the tests pass.

That is the basics of unit testing in C++. We will introduce the 'Section' keyword in part 3 of this tutorial but you should at least feel comfortable creating your own unit tests.

If you want to read more about Catch, see the documentation here: `https://github.com/catchorg/Catch2/blob/devel/docs/migrate-v2-to-v3.md#top`.

Part 2: The Big 6

In previous tutorials, we've only implemented procedural programming solutions. We are about to change that because unlike its predecessor C, C++ allows us to follow an object-orientated approach when writing applications. As you'll remember from previous years, a *class* is a user-defined "blueprint" of methods and attributes and an *object* is an instantiation of that blueprint in memory. To create a class in C++, we need to create two separate files. A header ('.h') file which contains our class' definition, attributes and method signatures and a '.cpp' file which contains our implementations of the methods we specified in the header file.

Go ahead and create a new file called 'Piece.h' and put the following code in it:

```
#ifndef __PIECE__
#define __PIECE__

namespace TUTORIAL4 // The name of this namespace
  should normally be your student number for assignments
{

    class Piece
    {

        // Local Variables
        int row, col;
        char *id;


        public: // All class members are private by default.
        // The public keyword allows us to make any method and/or attributes
        // that follow public.


        Piece(); // Default Constructor
        Piece(const int row, const int col, const char id); // Custom Constructor

        ~Piece(); // Destructor
```

```
        Piece(const Piece& p); // Copy Constructor
        Piece(Piece && p); // Move constructor


        Piece& operator=(const Piece& rhs); // Copy Assignment Operator
        Piece& operator=(Piece&& rhs); //Move Assignment Operator


    };

}

#endif
```

What we've done here is create the blueprint of our class. Let's go over it quickly. The following code is what is known as a an 'include guards':

```
#ifndef NAME
#define NAME

...

#endif
```

As you know, when we '#include' a file, the compiler is literally copy-pasting the contents of that file into the desired source file. What happens if we '#include' the same file twice? Well, the compiler will throw errors at you because we can't have duplicate method signatures, class definitions, etc. This is where the include guards comes in. The above code essentially checks to see if we've defined a macro called 'NAME' and, if we haven't, we create it and the compiler will copy the rest of the files contents across (everything inbetween '#ifndef' and '#endif'). If the macro does exist, the compiler will simply ignore its contents and move on with the rest of the compilation process. An alternative approach to the include guards is '#pragma once'. This directive just tells the compiler that we can only ever copy the contents of this file once. '#pragma once' is the non-standard approach but you may come across it.

You'll also notice that we've encapsulated our custom class within a namespace. This is not necessary but is a best practice (Even widely used classes and functions are still included in the 'std' namespace).

The rest of the code is simply our custom class definition. We've created a few local attributes ('row', 'col' and 'id'), and a few local methods (We will cover these shortly). You may have also heard of something called a 'struct'. In C++, a 'class' and a 'struct' are virtually identical. The only big difference they have is that class' attributes and methods are private by default whereas the attributes and methods of a struct are public by default. Structs are often useful when you just need a simple data container but if you

are concerned about hiding the implementation details of a custom object, you should use a class.

Note: It is good practice to separate your class definition and implementation into separate files ('.h' and '.cpp' respectively). You will be penalized in assignments if you do not follow this standard.

All that is left now is to discuss the the methods we've defined in our header file. These functions are actually known as the Big 6 and are a core part of what is known as the RAII/RRID paradigm. Resource Acquisition Is Initialization / Resource Release Is Destruction is at the core of every C++ object. When an object is created, the constructor is called. When an object goes out of scope or is destroyed, the destructor is called. The essential idea behind RAII/RRID is that if one of our custom objects ever needs to make use of dynamic memory, it allocates that memory when the constructor is called and releases said memory when the destructor is called. This prevents memory leaks and keeps our programs safe.

Now that the introductions are out of the way, lets implement some of these functions. Create a file called 'Piece.cpp' and open it up (also include the 'Piece.h' file).

1.1 Default Contructor

The first and simplest method to implement is the default constructor. This method is called when we create an object without supplying it with any input parameters like so: 'Piece()'.

Let's implement the method as follows:

```
TUTORIAL4::Piece::Piece() : row(0), col(0), id(nullptr)
{}
```

As you can see, this was pretty simple to implement. We used an initializer list (which are really useful) to set the default values for all the attributes in our custom 'Piece' class. **NOTE:** If you have any non static const attributes or reference attributes, you have to use an initializer list to set their values when the constructor is called.

As you would've noticed, we also have a custom constructor method signature defined. Custom constructors are not a part of the big 6 but you will often need to implement them. Let's implement ours like so:

```
TUTORIAL4::Piece::Piece(int row, int col, char id) : row(row), col(col),
id(new char(id))
{}
```

Again, pretty simple although our class is also pretty simple. Again we used an initializer list and this time just set the values of 'row', 'col' and 'id' to be whatever was supplied to the custom constructor. You'll also notice that despite the fact 'id' is of type 'char *', we asked for a 'char' to be passed into the constructor. We did this because we wanted to follow the RAII/RRID principles. If we passed in a 'char *' to the object, we wouldn't be able to successfully manage that resource. Take a look at the following example to see why:

```
For the sake of the example, our custom constructor's method signature
looks like this: Piece(int row, int col, char & *id)


// Example 1:

char id = 'P';
Piece p = Piece(0, 0, &id);

// Example 2:

Piece p = Piece(0, 0, new char('P'));
```

In example 1, we do not need to make sure that Piece's id is destroyed (it will happen automatically when the 'id' variable is no longer in scope) whereas in example 2, we do need to release it when our object goes out of scope. Both examples are valid, but need to handled completely differently from a resource management perspective.

NOTE: You could make the 'id' constructor argument a 'char *', you would just have to change 'id(new char(id))' to 'id(new char(*id))' to account for this.

1.2 The Destructor

Just as a constructor is invoked when an object is created, the destructor is invoked when an object leaves scope or is destroyed. The destructor is the second component of the RAII/RRID paradigm and is concerned with the releasing of any resources we acquired when the constructor is invoked. A destructor for our class would look as follows:

```
TUTORIAL4::Piece::~Piece()
{
    if(this->id != nullptr) // We have to make sure we're not trying
    // to release a bit of memory that doesn't exist.
    {
        delete this->id;
    }
}
```

That's it! All our destructor needs to do is release any memory we've allocated to it in the constructor (in this case it is 'id'). If we have an array, you can just use 'delete [] var_name' (from the previous tutorial). We also need make sure that we don't try to release memory that doesn't exist (this can happen when an object that has had its resources moved, has its destructor invoked).

Note: You need to think critically about the resources your object has allocated. If you have an object that utilizes 'layers' of resources (like a 'int **' 2d array for example), your destructor needs to release those resources as well. Simply deleting the 'int **' is not sufficient, you need release the resources the 2D array is pointing to as well.

1.3 The Copy Constructor and Copy Assignment Operator

We will often need to copy the contents of an object. C++ offers two ways to do that. The first of these methods is the copy constructor. The method signature of a copy constructor looks as follows:

```
CustomClass(const CustomClass & src);
```

You'll notice we used the 'const' keyword as well as the '&' symbol. When we use 'const' we are telling the compiler that the object being passed to this function cannot be modified (ie: It must remain constant). The '&' symbol denotes that the object being passed must be passed by reference as opposed to pass by value (which is C++ default behaviour). We use 'const' and '&' for two reasons: a copy constructor should not modify the contents of the object it is copying and secondly, because we want to copy the values of the 'src' object directly. Our implementation of the copy constructor could look as follows:

```
TUTORIAL4::Piece::Piece(const TUTORIAL4::Piece & p) : row(p.row), col(p.col),
id(p.id)
{}
```

Again, we simply use an initializer list to set the values our new object to those of the object supplied. Do you notice anything wrong with this? You may have caught on that what we've done is implement a shallow-copy function which is actually quite problematic. A shallow-copy of an object retains all of the original object's references. In our case, our new 'Piece' object's 'id' attribute points to the exact same address as the original 'Piece' object's id. This introduced another resource management problem since we essentially have two objects that are responsible for managing the same piece of memory. To fix this, we must make sure that we are creating a deep-copy of the original object like so:

```
TUTORIAL4::Piece::Piece(const TUTORIAL4::Piece & p) : row(p.row), col(p.col),
id(nullptr)
{
    if(p.id != nullptr)
    {
        id = new char(*p.id);
    }
}
```

This code solves our problem. We simply create a copy of the value being stored at the address pointer 'p,id' points to. We also make sure that 'p.id' does not point to 'nullptr' since we cannot copy that value as it would throw an exception. In general, you do not need to worry about this when dealing with primitives or other objects that follow the RAII/RRID paradigm but as soon as your class manages any kind of dynamic memory, you must take care to ensure that deep-copies are being created.

This brings us to the to the copy assignment operator. Just like the copy constructor the copy assignment operator is called when we want to create a copy of an object except that the copy assignment operator occurs when we use the '=' character. For example:

```
Piece p1(5,6,'p');
Piece p2;

p2 = p1; //Calls the copy assignment operator
```

The general form of the copy assignment operator is as follows:

```
CLASS & operator=(const CLASS & rhs);
```

For our example, the copy assignment operator looks as follows:

```
TUTORIAL4::Piece& TUTORIAL4::Piece::operator=(const TUTORIAL4::Piece & rhs)
{
    if(this != &rhs) // Checks to make that we are not performing a self-assignment
    {
        this->row = rhs.row;
        this->col = rhs.col;

        if(this->id != nullptr)
        {
            delete this->id; // 'this' may already be managing a bit of
              // memory so we must release it to prevent any memory leaks.
            this->id = nullptr;
        }

        if(rhs.id != nullptr)
        {
            this->id = new char(*rhs.id);
        }
    }

    return *this;
}
```

As you can see, the Copy Constructor and Copy Assignment Operator are remarkably similar bar the 'this != &rhs' self assignment check. Again, we've ensured that we are creating a deep-copy by giving our new object its own bit of memory to manage and we made sure to pass the object we want to copy ('rhs') by reference using the '&' character. Our return type is 'TUTORIAL4::Piece&'. This simply means that we are returning a reference to a 'Piece' object (We need to return our object since we are no longer working with constructors). 'this' refers to the object whose copy assignment operator was called (In general, 'this' refers to the object who had its function invoked) and '$this->$' is the same as saying '(*this)' since 'this' is actually a pointer to our object in memory. We've also included the following bit of code:

```
if(this->id != nullptr)
```

```
{
    delete this->id; // 'this' may already be managing a
    // bit of memory so we must release it to prevent any memory leaks.
    this->id = nullptr;
}
```

Given that we may be invoking the copy assignment operator on an object that has already been instantiated, it is possible for 'this' to already be managing a bit of memory. The above code simply checks to see if that is case and, if so, we simply release that bit of memory.

1.4 The Move Constructor and Move Assignment Operator

As the name implies, the act of 'moving' consists of the transferring of resources owned by one object to another object. This typically occurs when we are moving an rvalue's resources to an lvalue. An rvalue is an expression that does not have a memory address while an lvalue is an expression with a memory address.

Take a look at the following example:

```
std::vector<Piece> pieces;
pieces.push_back( Piece() ); // Create a 'Piece' object using the default
 // constructor and add it to our vector.
```

In this code snippet, 'Piece()' is an rvalue since it has no memory address associated with it. Without move semantics, the above bit of code would call the default constructor of 'Piece' and then the copy constructor of 'Piece' when the object is pushed to the back of the vector 'pieces'. As one can imagine, this is incredibly inefficient and scales very poorly as our objects grow in complexity. Move semantics allow us to create custom behaviour for when our classes receive rvalues as input. To specialize our functions for rvalues, we follow the following general form:

```
type && var_name = rvalue; // This form allows you to create a variable
 // that references an rvalue

return_type func(input_type && param) // Use this when you want to
 // specialize your function to operate on rvalues
{

}
```

As you can see, you can identify rvalue references by the '&&' symbols.

That brings us to the move constructor. The move constructor for our 'Piece' class would look as follows:

```
TUTORIAL4::Piece::Piece(TUTORIAL4::Piece && p) : row(p.row), col(p.col),
id(p.id)
{
    p.id = nullptr;
}
```

This code looks extremely similar to the copy constructor. The only difference here is that instead of making a deep-copy we are transferring ownership of the 'id' pointer from 'p' to our new object. We also make sure we set 'p.id = nullptr' because 'p' is no longer responsible for that bit of memory. In general, you only need to worry about managing the ownership of memory your object is responsible for. For example, say our 'Piece' class had another attribute called 'Piece * neighbour'. If we implemented everything correctly, our piece object would not need to manage any of the memory created by 'neighbour' and we could simply use the 'std::move' function to move the contents of 'p.neighbour' to 'neighbour' like so:

```
TUTORIAL4::Piece::Piece( TUTORIAL4::Piece && p) : row(p.row), col(p.col),
id(p.id)
{
neighbour = std::move(p.neighbour);
p.id = nullptr;
}
```

Note: 'std::move' can be used to invoke functions that require rvalues. You may never need to use this but it is good to know regardless. For more info on 'std::move', see the following link here: `https://en.cppreference.com/w/cpp/utility/move)`.

That covers the move constructor. The move assignment operator is similar to copy assignment operator except that, like the move constructor, we are moving the resources rather than copying them. Our move assignment operator should look as follows:

```
TUTORIAL4::Piece& TUTORIAL4::Piece::operator=(TUTORIAL4::Piece && rhs)
{
    if(this != &rhs) // Checks to make that we are not performing a
    // self-assignment
    {

        this->row = rhs.row;
        this->col = rhs.col;

        if(this->id != nullptr)
        {
            delete this->id; // 'this' may already be managing a bit
            // of memory so we must release it to prevent any memory leaks.
            this->id = nullptr;
        }

        if(rhs.id != nullptr)
```

```
        {
            this->id = rhs.id;
            rhs.id = nullptr; // rhs is no longer responsible for the bit
            // of memory 'rhs.id' points to so we set it to nullptr
        }
    }

    return *this;
}
```

The above function is a little more complex but it is very similar to the move constructor. Again we must make sure we are relieving 'rhs' of its control over the memory 'rhs.id' points to. We must also make sure that we are not performing any self assignment (yes, it is possible using 'std::move') and, just like the copy assignment operator, our 'this' object may already be managing resources and we should release them to avoid memory leaks.

2 Part 3: Task

Now that we've gone through all of the big 6 functions, your job is to write unit tests for each of the big 6 member functions and the custom constructor (do not worry about the destructor as it can be quite hard to properly unit test this function) in the 'driver_test.cpp' file.

Note: You can divide up your unit tests into sections like so:

```
TEST_CASE( "Piece Class Unit Tests") {


    SECTION( "Default Constructor" ) {
        // Write your tests here
    }
    SECTION( "Custom Constructor" ) {
        // Write your tests here
    }

    ...

    SECTION( "Move Assignment Operator" ) {
        // Write your tests here
    }
}
```

- Once you have completed the task, go ahead and submit your work to Vula
- **(Optional)** Take a look at Part 4 of this tutorial

3 Part 4: (Optional) Task

Our custom 'Piece' class is a bit lonely. Let's fix that! You task is to create a 'Board' class. This 'Board' class should have the following attributes:

- height : int
- id : char *
- gameBoard : Pieces **

You must implement all of the big 6 functions we covered in this tutorial as well as a custom constructor that accepts a 'height' and 'id' as input. This custom constructor should create a 2D array of size '[height][8]' and fill the board with pieces such that the first and last two rows contain pieces with the following IDs:

```
[C,N,B,Q,K,B,N,C]
[P,P,P,P,P,P,P,P]
.
.
.
[P,P,P,P,P,P,P,P]
[C,N,B,K,Q,B,N,C]
```

You are also expected write unit tests for all these functions (again, do not worry too much about the destructor). These tests should sufficiently test the aforementioned functions to ensure they work correctly.

Note: You may assume that the height of the board will always be greater than 4.