ELECTRICAL AND INFORMATION ENGINEERING

University of the Witwatersrand, Johannesburg

Software Development II – Laboratory 3

Construction, `static`, and Smart Pointers

1 Introduction

This laboratory deals with object construction and the technical considerations which need to be taken into account when constructing objects (see Section 2). Class attributes which are defined through the use of the `static` keyword are also introduced here. Smart pointers which allow for a much safer approach to managing dynamically allocated memory are presented in Section 3.

The laboratory outcomes are:

1. You are able to successfully construct classes, and are aware of the issues surrounding object construction.
2. You appreciate situations which require the use of `static` and are able to construct and manipulate `static` objects.
3. You are able to use `unique_ptr` and `shared_ptr` to safely manage dynamically allocated memory.

As in the previous labs, one of the lab partners should download the starter code from the course website (the Laboratories section) and extract it to a local folder. Within this folder run `git init` to initialise the Git repo.

Now stage all of the untracked files and directories by typing: `git add .`
Commit the files to the *master* branch with: `git commit -m "Initial commit"`

Create a *solutions* branch on which to merge all the solutions for the exercises by typing:
`git branch solutions`

Share the repo with your partner by visiting GitHub and copying the **SSH** url from the *Quick setup* section of your Lab 3 repo. Now in Git Bash, type:
`git remote add origin <paste SSH url here>`

Push your local branches to the remote by typing:
`git push --all origin -u`

Now that the starter code, and the *master* and *solution* branches, are available in the shared GitHub repo, your lab partner needs to clone the remote repo using:
`git clone <paste SSH url from GitHub here>`

Once again, you will be creating a branch for the exercises in each section. Lab partners will alternate sections, so partner *A* will do all the exercises in Section 2 and partner *B* will do Section 3.

You are now in a position to start working on the lab exercises.

2 Object Construction and `static`

Each object that is constructed within an object-oriented program has a *state*. The state of an object encompasses all of the current values of each of its data members. It is vitally important to ensure that an object begins life, or is constructed, with a *valid* state. The state of an object will, of course, change during the object's lifetime.

Read and make sure that you understand the module for the `Date` class. Note that the enumeration type, `Month`, is cast to an integer through the use of `static_cast` in order to print the date in a numeric format. It is also useful to cast the month internally as an integer in order to perform operations such as addition and subtraction on the month.

Create a branch for this section by typing the following:
```
git branch section-2
```

Checkout out this branch:
```
git checkout section-2
```

You can now work through all of the exercises. Remember to commit after each exercise, naming the commits correctly.

Exercise 2.1

The `Date` class does not have any constructors defined so the compiler supplies a *default constructor* (a constructor which can be called with no arguments) which allows us to create objects of the class. Run the test given in `date-tests.cpp`, and verify that the default constructor supplied by the compiler does not, in fact, initialise `Date`'s data members with meaningful values.

Now comment out the above test and uncomment the next test in `date-tests.cpp`. Provide your own constructor with just enough code to make the test pass.

Exercise 2.2

It is important to ensure that a date object always represents a *valid date*, because clients of `Date` will naturally expect the date to be valid. One of the issues when writing a constructor is: How does the constructor inform the client code if the object being constructed has an invalid state? Remember, that a constructor has no return value. A good method of informing client code is to throw an exception. This forces the client to handle the error (it doesn't simply pass by unnoticed) and *prevents the object from being constructed*. So write tests which verify that an exception is thrown when attempting to construct an invalid date object. Then write code to make the tests pass.

Together, `Date`'s member functions maintain what is known as a *class invariant*. An invariant is some property or truth about objects of the class. In this case, it is the fact that the `Date` will always be valid. The invariant restricts the set of all possible Date objects to only those which represent valid dates. The constructor establishes the invariant and all other member functions should preserve it.

Exercise 2.3

It is cumbersome to have to test the equality of two dates in a data-member-by-data-member fashion, by calling `CHECK` three times. What we would really like to do is:

```
auto date_1 = Date{1, Month::January, 2000};
auto date_2 = Date{1, Month::January, 2000};

CHECK(date_1 == date_2);
```

In order for this to work we need to overload the *equality operator* for the `Date` class. Write this overloaded operator for `Date`. This operator has the following form:

```
bool Date::operator==(const Date& rhs) const
{
    // compare data members here
    // return true for identical dates, false otherwise
}
```

Before we can actually use the equality operator in our tests we need to *fully* test that it is working correctly. There are at least three tests, other than the one above, that should be written to complete the testing of the equality operator. Hint: Under what conditions would we expect equality operator to evaluate to false?

Exercise 2.4

Provide a new member function for the `Date` class which will increase the date by one day. Adopt a test-driven development approach and write the tests first. Think carefully about which test cases to write. You want to focus on boundary conditions and write tests which are likely to expose logical errors. Remember, that this new function needs to preserve the class invariant by ensuring that the new date is always valid.

Hint: To make this easier to implement, consider casting the `Month` to an integer.

Exercise 2.5

Having defined a constructor in an earlier exercise, the default constructor *ceases to exist*. If a class designer provides constructors then only those constructors may be called; the default constructor supplied by the compiler effectively disappears. This a desirable situation because it forces the class user to supply a (hopefully meaningful) date for any `Date` object that is constructed, ensuring that arbitrary `Date` objects do not exist.

Unfortunately, certain commercial container class libraries, as well as, the built-in array type require the contained objects to provide a default constructor (this is not the case with STL containers). It is therefore worthwhile knowing how to define a default constructor.

There are two ways of defining a default constructor for our `Date` class. The first option is to have a single constructor which has *default arguments*[1] for every parameter:

```
// assume we want a default date of 1/1/1900
Date (int day = 1, Month month = Month::January, int year = 1900);
```

This approach lacks flexibility because a user of the class has no way of changing the default date.

A more flexible option is to provide two constructors:

[1] A function declaration can specify default arguments for all or only a subset of it parameters. The rightmost parameter must be supplied with a default argument before a default argument for the parameter to its left can be supplied. This is because arguments to a function call are resolved by position. Default arguments only appear in the function's declaration (the `.h` file) — not in the function's definition (the `.cpp` file).

```
// create a date from a given day, month and year
Date (int day, Month month, int year);

Date(); // default constructor requiring no arguments
```

Here, the default constructor will use an existing date object for initialising any new date objects that are created. This existing date object represents the default date for the class. It makes sense for the default date to exist as a `private`, `static` data member within the `Date` class for two reasons:

- the `private` keyword ensures that the default date is hidden from any code outside the class scope.
- the `static` keyword ensures that the default date is identical for all objects of the class, and that it will exist prior to any calls to the default constructor.

Create a new private, static data member, of type `Date`, called `default_` within the `Date` class to store the default date. Remember, that there is a single instance of this `static` data member which is shared by all `Date` objects. `default_` must be initialised outside of the class definition because the entire definition needs to be parsed by the compiler before objects of the type can be constructed and initialised. Initialisation is therefore done in `date.cpp` using `Date`'s ordinary constructor as follows:

```
Date Date::default_{1, Month::January, 1900};
```

The above line of code is explained step by step as follows:

1. Firstly, we specify the type of the static data member: `Date`
2. This is followed by the name of the static data member (prefixed by the scope): `Date::default_`. The scope needs to be specified because this initialisation takes place outside the `Date` class.
3. Finally, we are calling `Date`'s constructor, so we need to supply the list of arguments: `{1, Month::January, 1900}`

To allow users to change the default date an additional member function is required:

```
static void setDefaultDate(int day, Month month, int year);
```

Write tests for, and implement, both the default constructor (requiring no arguments) and the `setDefaultDate` function. Note that only a single default constructor may be defined for a class.

The choice of a default date is fairly arbitrary — the 1/1/1900 is no more or less reasonable than 1/1/1800. Ultimately, when you use a default constructor an object which has no particular significance is constructed and could potentially be misused.

> Avoid providing default constructors unless circumstances require them.

You now need to *merge* your commits on the *section-2* branch into the *solutions* branch. After doing this push your *solutions* branch to GitHub. First pull down the remote *solutions* branch to ensure that you have an up-to-date version.

3 Smart Pointers

Dynamic memory allocation is the process in which a program is given memory upon request while the program is running. The memory that is allocated is taken from a region of memory known as the *heap* or *free store*. The heap is a finite resource, consequently, a dynamic memory request may not be honoured if this resource is exhausted. If a memory allocation request does not succeed then a `bad_alloc` exception is thrown. Dynamic memory allocation is performed in C++ using the `new` operator. If a memory allocation request is successful `new` returns a pointer to the memory just allocated.

When the requested portion of memory is no longer needed it should be released using the `delete` keyword so that it will be available for future use, such as subsequent dynamic memory requests. The use of `new` and `delete` enable the programmer to control the lifetime of dynamically created objects.

Dynamic memory allocation is normally used when the amount of memory required by a program is unknown at compile time, and instead depends on decisions made during the execution of the program (at run-time). Both `vector` and `string` depend on dynamically allocating memory because their size is not predetermined at run-time.

The use of ordinary pointers for managing memory is difficult, especially in the presence of exceptions, and often results in subtle program errors. To counter these problems it is generally advisable to use smart pointers. Smart pointers are "smart" in the sense that they own or share ownership of the memory that is being pointed to. They are guaranteed to:

- automatically release the memory that they own when they go out of scope. Hence, there is no need to remember to call `delete`.
- never be left dangling. If ownership is shared among smart pointers then reference counting is used to make sure that the memory pointed to is *always* valid for each smart pointer.
- prevent corruption of the heap. If a smart pointer is not pointing to memory that it owns then it is set to point to 0 (null). It is therefore impossible to accidently delete memory twice.

We will make use of two smart pointers that are part of the C++11 standard library. In particular, we will use `unique_ptr` and `shared_ptr`.

> Prefer smart pointers to ordinary pointers.

Create a branch for this section by typing the following:
`git branch section-3`

Checkout out this branch, and commit after each exercise, naming the commits correctly.

Exercise 3.1

Examine the code of `person-main.cpp`, which is given in Listing 1, as well as `person.h`, and answer the following questions. Provide your answers as comments in `person-main.cpp`.

1. Identify when each of the following smart pointers or objects goes out of scope.
   (a) `thabo`

(b) `maryanne_ptr`

(c) `person_ptr`

(d) `sandile_ptr`

(e) `thabo_in_main`

(f) `person_ptr_in_main`

2. Identify when the memory for each of the *pointees*, corresponding to the above pointers, is released.
3. Determine the number of times that each smart pointer is copied.
4. Which will go out of scope first, `sandile_ptr` or `thabo_in_main`, and why?

Make sure that you can compile and run the code in Listing 1. Note, to make use of smart pointers the `memory` header file has to be included.

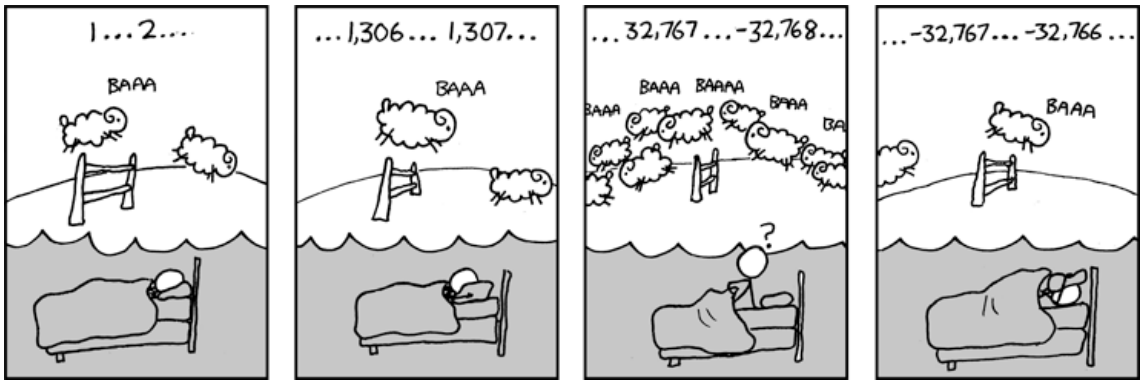
```
Person printName()
{
   Person thabo{"Thabo",12};
   cout << thabo.name() << endl;
   return thabo;
}

shared_ptr<Person> printName2()
{
   auto maryanne_ptr = make_shared<Person>("Maryanne",12);
   cout << maryanne_ptr->name() << endl;
   return maryanne_ptr;
}

void printName3(shared_ptr<Person> person_ptr)
{
   cout << person_ptr->name() << endl;
   return;
}


int main()
{
    auto sandile_ptr = make_unique<Person>("Sandile",15);

    auto thabo_in_main = printName();
    cout << thabo_in_main.name() << endl;

    auto person_ptr_in_main = printName2();
    printName3(person_ptr_in_main);

    sandile_ptr = make_unique<Person>("Sandile2",11);
    cout << sandile_ptr->age() << endl;

    return 0;
}
```

Listing 1: Smart Pointer Exercise

Now *merge* your commits on the your local *section-3* branch into your local *solutions* branch.

Push your *solutions* branch to GitHub. First pull down the remote *solutions* branch to ensure that you have an up-to-date version.

Once both partners have completed their section of the lab, submit it using a pull request on GitHub following the instructions in the Git/GitHub guide. The lab will be assessed according to the same criteria given in Lab 1. Your *solutions* branch must have a clean commit history.



Source: http://xkcd.com/571/

Source: http://dilbert.com/strips/comic/1995-09-17/

Version 3.0, 2020 (`elen3009-lab3.tex`)