Introductory C++ exercises

Rupert Nash

r.nash@epcc.ed.ac.uk

The files for this are on Github.

To check out the repository run:

```
git clone https://github.com/EPCCed/APT-CPP.git
cd APT-CPP/practicals/01
```

A linked list class

Check that you understand a linked list - see exercise 0 implemented in C if not. A solution to that exercise is in this directory (`list.c`).

Reimplement this as a class

The goal of this is to implement a very similar linked-list as a C++ class.

In `APT-CPP/practicals/01-list-array/1-list` there is a Makefile, a test program, a header file containing the class definition, and a partial implementation in `list.cpp`. Running `make` will try to build the executable `test`. This program generates some random numbers and adds them to an instance of your list class, keeping them sorted.

When complete your program will produce output like:

```
$ ./test 100
Time to insert 100 integers = 0.000142364 s
Were correctly ordered
$ ./exA 1000
Time to insert 1000 integers = 0.0025844 s
Were correctly ordered
```

You need to edit `list.cpp` and complete the code. The design I have used is very similar to the C implementation and is *not* idiomatic C++. If you can see improvements - go ahead and try them - but note we will come back to this.

When you have this working, try to answer the following

1. How does it scale as you increase N? Try plotting on a log-log scale. Is this what you expected? You may wish to `make clean` and recompile with higher optimisation (add `-O3` to the `CXXFLAGS` variable in the Makefile).

2. Point out a few flaws in this design. Things to consider include: const-correctness, RAII, having to use a non-standard iteration syntax.

Array

The array is a fundamental data structure, especially for processing large amounts of data, as it allows the system to take advantage of the cache hierarchy.

Recall the array template examples from the lecture - in `APT-CPP/practicals/01-list-array/2-array` is a basic implementation and a (hopefully) working test program, very similar to the previous one.

Compile this and run it for a few problem sizes. What is the scaling? How does this compare to the linked list?

We need to take a decision about copying - do we wish to allow implicit copying which for large arrays is very slow? If not, should we add an *explicit* method to do this? What would its signature be? How would we tell the compiler not to allow this?

Libraries

Memory

While we've taken a RAII approach here, it comes with some overhead: we had to implement (or delete) five functions: the destructor, the copy constructor, the move constructor, the copy assignment operator, and the move assignment operator. This concept is known as "the rule of five" (before C++11 it only had three).

A more idiomatic approach is to wrap the resource into a class that does nothing but manage a resource, then it can be used elsewhere and the compiler will produce correct implicit constructors, destructor and assignment operators with no boilerplate code!

See one of the below for an in-depth discussion: * http://en.cppreference.com/w/cpp/language/rule_of_three * http://scottmeyers.blogspot.co.uk/2014/03/a-concern-about-rule-of-zero.html

Fortunately the standard library includes several "smart pointers" that will do this for you for memory! They can be accessed using the `<memory>` header.

They are:

- `std::unique_ptr` - this uniquely owns the pointed-to object. The object is deleted (can be customised) when the smart pointer destructs or you assign a new value. You cannot copy a `unique_ptr`. *This should be your default pointer type!*

- `std::shared_ptr` - this shares ownership of the pointed-to object. All the child `shared_ptr`s point to the same object. The object will be deleted when *all* the pointers are either destructed or assigned a new value.
- `std::weak_ptr` - much like a `shared_ptr` but it doesn't own a share of the object. It can become invalid. Used to break reference cycles.

(There also exists a `std::auto_ptr`. This is deprecated and has been removed from C++17, so do not use it.)

Have a look at the reference and re-implement `Array` using either a unique or shared pointer.

Containers

The standard library also has a number of containers for objects. These include a `list` template class (typically a doubly-linked list) and `vector` which is a contiguous, dynamically-sized array, a bit like our `Array<T>`. There are also hash tables, queues (single and double ended), stacks, etc.

The full list is here http://en.cppreference.com/w/cpp/container.

Replace your list with `std::list<int>` and compare performance.

Try the same with replacing your array with `std::vector`.