University of Cape Town
Department of Computer Science

CSC3022H – C++ Tutorial 2
Container, Iterators and Operator Overloading

March, 2012

The objective of this tutorial is to create a string *container class*, supporting *iterator classes* and string functions. Conceptually, a container is a data structure storing multiple values of some type, and is rather heavyweight in that it contains a large amount of data. However, the underlying internal structure of the container is not exposed directly to the user. Instead, a user accesses and modifies this data using *iterators*, which are lightweight classes that only store information necessary to represent a particular location within the container. Consequently iterators are used to move through container and provide mediated access to data within the container.

In this tutorial you will implement, for want of a better term, a **bucket string** class. The computational benefits of this datastructure are unclear, but it should provide good insight into C++ memory management and containers.

1 The bucket_string class

Internally the bucket string should be represented by a linked list of **buckets**. Each bucket should contain a dynamically allocated array of characters which will represent different sections of the string. The bucket string class should support:

- The Four Special Member Functions: Default Constructor, Copy Constructor, Copy Assignment Operator and Destructor.
- `operator<<` and `operator>>` overloaded for the class, so that strings can be read from `istream &` or written to `ostream &` objects.
- `char & operator[]`, (parenthesis operator) returning a non-const reference to the character at the supplied index argument
- `std::size_t length()`, returns the length of the bucket string. `std::size_t` is an unsigned integer type.
- `iterator begin()`: returns an iterator pointing at the first character of the string.
- `iterator end()`: returns an iterator pointing *one location past* the last string character.
- `replace(iterator first, iterator last, bucket_string bs)`: replaces the iterator range, exclusive of last, with bs.

- `insert(iterator first, bucket_string bs)`: inserts a `bucket_string` at first.
- `bucket_string substr(iterator first, iterator last)`: returns a `bucket_string` corresponding to the iterator range, exclusive of last.

When constructing, the `bucket_string` class, the user should be able to specify the bucket size used for that particular class, but the default bucket size should be seven characters.

2 The iterator class

This class should, at a design level, store a location within the bucket_string and provide methods for accessing data at the location and moving backwards and forwards from that location. The iterator should support:

- The Four Special Member Functions:
- `*operator`: dereferences the bucket_string character pointed to by the iterator location.
- `operator++`: advances the iterator location forward by one position.
- `operator--`: moves the iterator location one position back.
- `operator+`: advances iterator location by an integer value
- `operator-` moves iterator location back by an integer value.

We haven't covered operator overloading in class yet. You can implement the functionality of the above operators with normal functions and then replace them with the formal operator overloads later. e.g.

- `get()` and `set()` for `*operator`.
- `next()` and `prev()` for `operator++` and `operator--`.
- `next(std::size_t i)` and `prev(std::size_t i)` for `operator+` and `operator-`.

Your iterator constructors must be **private**. In fact iterators should only be constructed in the begin() and end() methods of the bucket_string class, since only the container class understand its internal structure. The user should obtain iterators from these methods and iterator forward and backward with them. However, this means that the bucket_string class will need access to the private constructors of the iterator class, in a manner similar to package private access functionality in Java.

Note that substr and replace will invalidate iterators: Thats OK for this tut.

3 Testing

Utilise the cmdline_parser code to specify an input file containing a long test string. Then, provide test functions showing that your bucket_string works for:

- Four Special Member Functions.
- `operator<<` and `operator>>`
- replace and substr. using a variety of iterator ranges

4 Notes

- You must use raw pointers and manually manage your memory. No `shared_ptr`'s' or `auto_ptr`'s allowed.
- You may not use vectors or other containers inside the `bucket_string` class.
- You should try to use iterators as much as possible when implementing the various string manipulation functions, or even constructors...
- Iterators are conventionally passed by value. They are small classes, and relative to the amount of data that they process, the cost of deep copying their internal data is inconsequential.
- Make everything in your classes public first and get everything working. Then make the relevant parts private and introduce the necessary access mechanisms.
- Implement the parenthesis operator first. Then the unary `operator*` within the iterator. They do the same thing in different ways.
- The `bucket_string` and `iterator` classes should be in a namespace named with your student number.
- I suggest implementing
  - Week One: general `bucket_string` class
  - Week Two: `iterator` class + associated methods.
  - Week Three: `replace` and `substr` methods.

Handin Date: 27th March 2013, 9AM

Please Note:

1. A working *makefile* must be submitted. If the tutor cannot compile your program on the senior lab machines by typing make, you will receive **50%** of your final mark.
2. You must provide a README file explaining what each file submitted does and how it fits into the program as a whole. The README file should **not** explain any theory that you have used. These will be used by the tutors if they encounter any problems.
3. Do not hand in any binary files.
4. Please ensure that your tarball works and is not corrupt (you can check this by trying to extract the contents of your tarball - make this a habit!). Corrupt or non-working tarballs will not be marked - **no exceptions!**
5. A 10% penalty per day will be incurred for all late submissions. No hand-ins will be accepted if later than 5 days.
6. **DO NOT COPY. All code submitted must be your own.** ***Copying is punishable by 0 and can cause a blotch on your academic record.*** **Scripts will be used to check that code submitted is unique.**