Data Structures II
Fall 2009

These notes are being created in conjunction with the teaching of CPSC 482 in the Fall 2009 term at the University of Northern British Columbia.

Data Structures: C++ versus Java Here are some language differences that affect *how* we implement data structures in C++ versus JAVA.

templates

- Templates in C++ are a compile-time activity that take place every time that we *use* a template. At least in theory, the `list<long>`-class and the `list<float>`-class share *no* machine code. By link-time the template notation has effectively disappeared.

  Templates in JAVA are syntactic sugar for some run-time type checking. A `LinkedList<Long>`-class and the `LinkedList<Float>`-class share the same JVM byte codes, except that where they are *used* there may be some run-time casts.

- Template arguments in C++ are anything that can be determined at compile-time, including integer constants and the addresses of externally linked functions. All types can be template arguments.

  Template arguments in JAVA are restricted to class names. Java classes that build templates around primitive types need to use wrapper classes.

Embeddedness In C++ objects can physically contain other objects. In JAVA objects only every contain [what C++ would call] pointers to other objects.

A linked list in C++ might typically look like what is shown in Figure 1, whereas in JAVA it might more look like Figure 2.

What are the advantages and dis-advantages of the C++ approach?

Storage Management In C++ without garbage collection, one is always incredibly conscious of ownership issues. In JAVA subtle memory

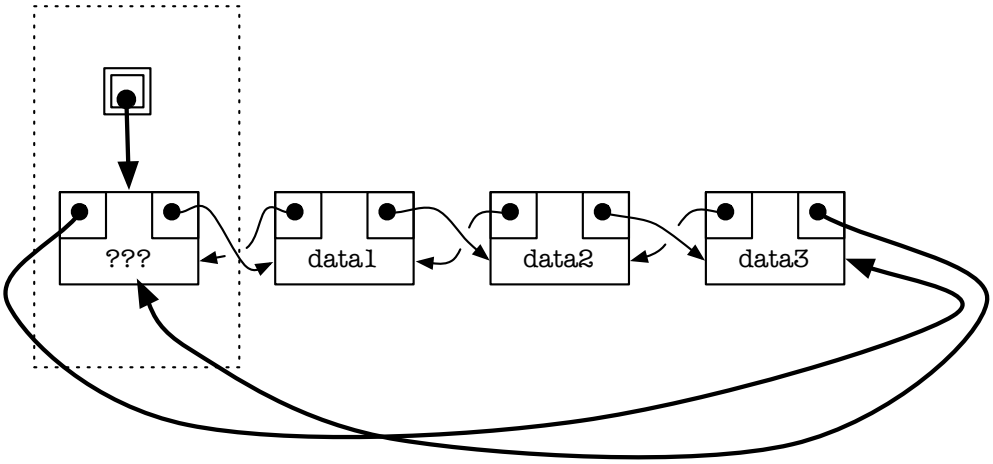

Figure 1: C++ linked list

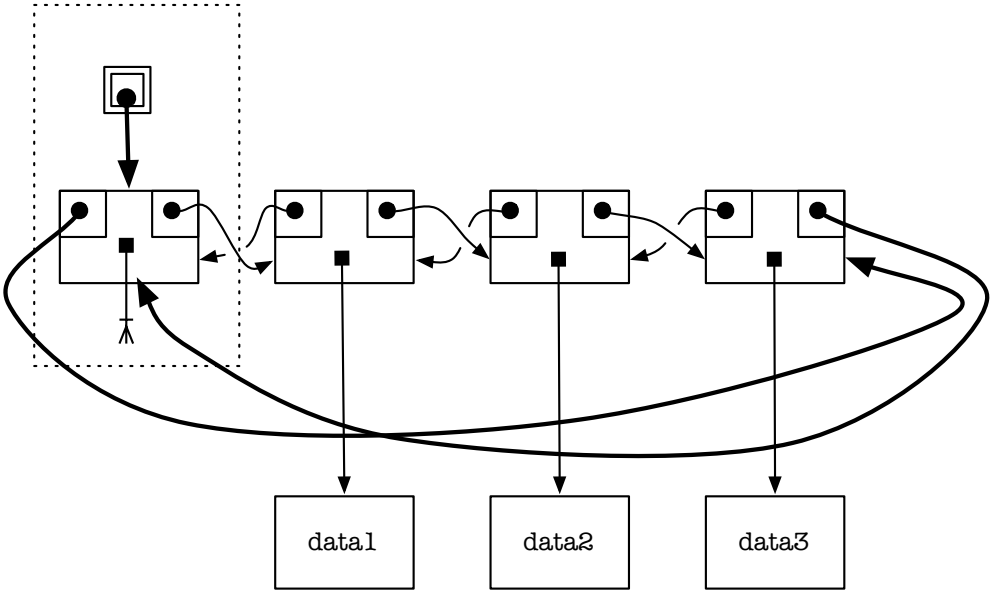

Figure 2: Java linked list

use creep can occur because someone forgets to null pointers, but it is much easier to share data.

In C++ shallow copies are quite rare, as are immutable data structures.

A guide to the C++ Standard Template Library.

The Standard Template Library was developed by Alexander Stepanov at Silicon Graphics (and implemented by Meng Lee at Hewlett-Packard) at the same time that Bjarne Stroustrup was fleshing out the details of the template mechanism in C++. Consequently the STL is very tightly integrated with the C++ programming language, and it is now true that most of the C++ libraries rest on a templated foundation.

Parts The components of the Standard Template Library are:

I. Containers,
II. Algorithms,
III. Iterators,
IV. Function Objects, and
V. Adapters.

Containers ...

Containers in the STL are what we tend to think of as data structures. They are further divided into sequential containers and associative containers.

Sequential containers store elements in a finite ordered sequence. The three kinds of sequential containers supported by the Standard Template Library are

1. `vector`'s,
2. `deque`'s, and
3. `lists`.

Associative containers store information in an order defined by the data. The associative containers in the STL are

1. `set`'s,
2. `multiset`'s,

3. `map`'s, and
4. `multimap`'s.

Algorithms

- some examples by name: `rotate`, `reverse`, `sort`.
- the complexity problem of ALGORITHMS × CONTAINERS.
- the Standard Template Library solution: iterators.

Iterators

- an iterator (in any programming language) is a means of traversing a container to access its elements, with the speed of access explicitly under programmer control (as opposed to JAVA extended for loop syntax.)
  Iterators allow multiple simultaneous access to a collection at the same time.
- C++ has operator overloading and pointers, so C++ STL iterators *loook like* pointers. There is no STL requirement that iterators implement a common interface (abstract base class).
- the entire interface between algorithms and containers in the C++ Standard Template Library is through iterators.
- In the C++ STL iterators are *not* allowed to change the shape of a container by themselves. In JAVA they are.
  In C++, it is the programmers responsibility to ensure that iterators remain valid when a container's shape is modififed. In JAVA, a container invalidates all of its iterators when it is modified.

The Iterator range convention In the C++ STL, pairs of iterators are frequently used to represent a range of data. The convention used is is that the iterator pair (b, e) represents the data range $[b, e)$. Consequences include the following:

- iterators need to be able to represent one beyond the end.
- there are multiple representations of the empty data range.

In the JAVA Collections classes, the claim is that iterators point *between* elements rather than *at* elements. Whence the notion that list iterators have `.next()` and `.previous()` methods.

The JAVA `List` interface provides a `.subList(int,int)` method for representing ranges.

C++ iterators for Java speakers Here is a concrete algorithm from the STL.

```
template<typename InputIterator, typename OutputIterator>
OutputIterator
copy(InputIterator b, InputIterator e, OutputIterator dest)
{
    while (b != e)
        *dest++ = *b++;
    return dest;
}
```

Let's translate this into JAVA.

Kinds of Iterators There are

1. input,
2. output,

Table 1: C++ Iterator operations

	input	**output**	**forward**	**bidirectional**	**random**
* (left)		Y	Y	Y	Y
* (right)	Y		Y	Y	Y
++ (pre)			Y	Y	Y
++ (post)	Y	Y	Y	Y	Y
-- (pre)				Y	Y
-- (post)	Y	Y		Y	Y
==, !=	Y	Y	Y	Y	Y
<=, >=, <, >					Y
(arithmetic)					Y

3. forward,
4. bi-directional, and
5. random access

iterators.

These iterators all support ++, ==, !=. In addition, they support

input * on the right hand side of assignment.

output * on the left hand side of assignment.

forward essentially both input and output. (Think iterator to singly-linked list.)

bi-directional (Think iterator to `list`.) These also support --.

random access (Think pointer.) These also support pointer arithmetic, `[]`, and all of the comparison operators.

Figure 3: Input Iterators translated into Java

Figure 4: Output Iterators translated into Java

Figure 5: Forward Iterators translated into JAVA

Figure 6: Bi-directional Iterators translated into JAVA

Figure 7: Random Access Iterators translated into Java

Sources of Iterators Each of the Standard Template Library containers provides `.begin()`, `.end()`, `rbegin()` and `.rend()` member functions, each of which returns an iterator. Each container also provides `typedefs` for these: for instance,

- `std::vector<int>::iterator`
- `std::vector<int>::const_iterator`
- `std::vector<int>::reverse_iterator`
- `std::vector<int>::const_reverse_iterator`

The precise type of the iterator depends on the container. The `vector` and `deque` classes give random access iterators; most of the rest give bi-directional iterators.

Another source of iterators is the `<iterator>` standard library. This provides

- adapters like `back_inserter`,
- templated class adapters like `reverse_iterator<...>`,
- `istream_iterator`,
- `ostream_iterator`.

The connection between algorithms and iterators revisited Now that we have seen the various types of iterators, we can see that algorithms must have certain requirements of the iterators that they use. For instance, the `sort` and `random_shuffle` algorithms require random access iterators, which means that they cannot be used on `lists`, which provide bi-directional iterators.