

Building on Lists

- we will look at how our `slist` class can be extended to include
 - `length`, `reverse` methods
- use recursion and functional programming where appropriate
- we will notice a problem with C++ when applying this technique

Tutorial answers: length

- firstly we add two helper methods to our class `slist`

`c++/lists/single-list/int/slist.h`

```
...
private:
    element *e_tail (element *l);
    int *e_length (element *l);
...
```

e_tail

- `c++/lists/single-list/int/slist.cc`

```
/*
 * e_tail - given a list, l, return the list without the
 *          head element.
 *          pre-condition: non empty list.
 *          post-condition: return the list without the
 *                          first element.
 *                          The original list is unalte:
 */
element *slist::e_tail (element *l)
{
    return l->next;
}
```

e_tail

- `c++/lists/single-list/int/slist.cc`

```
/*
 * e_length - return the length of element list, l.
 */
int slist::e_length (element *h)
{
    if (h == 0)
        return 0;
    else
        return (1 + e_length (e_tail (h)));
}
```

e_tail

c++/lists/single-list/int/slist.cc

```

/*
 * length - return the length of list, l.
 */
int slist::length (void)
{
    return e_length (head_element);
}

```

reverse

- must return the list with its contents reversed
 - not a new list with a copy of the contents reversed!

c++/lists/single-list/int/slist.cc

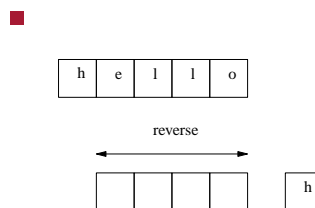
```

slist slist::reverse (void)
{
    if (is_empty ())
        return *this;
    else
        return tail ().reverse().cons (empty().cons (head ()))
}

```

reverse

- notice the use of recursion
- notice that tail removes and deletes a datum
- head obtains the first element
- cons appends the first element to an empty list
 - ie creates a list with one element
- this single element list is added to the end of the reversed list
 - the reversed list comes from the tail of the original list

reverse

cons (slist 1)

c++/lists/single-list/int/slist.cc

```

/*
 * cons - concatenate list, l, to the end of the current
 * pre-condition : none.
 * post-condition: returns the current list with
 * contents of list, l, appended
 */

slist slist::cons (slist l)
{
  if (l.is_empty ())
    return *this;
  else
    return cons (duplicate_elements (l.head_element));
}

```

Recursive version of cons (slist 1)

c++/lists/single-list/int/slist.cc

```

/*
 * cons - concatenate list, l, to the end of the current
 * pre-condition : none.
 * post-condition: returns the current list with
 * contents of list, l, appended
 */

slist slist::cons (slist l)
{
  if (l.is_empty ())
    return *this;
  else
    {
      int h = l.head (); // use h to force evaluation order
      return cons (h).cons (l.tail ());
    }
}

```

Recursive version of cons (slist 1)

- notice the *gotya*
 - we must use a temporary variable *h* to contain an intermediate result containing the result of `l.head()`
 - it ensure that the call to `head` occurs before `tail()`

Recursive version of cons (slist 1)

- if the code were re-written as:

c++/lists/single-list/int/slist.cc

```

slist slist::cons (slist l)
{
  if (l.is_empty ())
    return *this;
  else
    return cons (l.head()).cons (l.tail ());
}

```

Recursive version of cons (slist l)

- it would fail, as `l.tail()` is executed before `l.head()`

Further tutorial questions

- write some test code to generate a large list and perform reverse on the list several times
 - compare the execution time between the iterative and recursive solutions
 - which is faster, why?
- hint use `-pg` flags to `g++` and analyse the execution time with `gprof`
- see week 1 notes for further hints on using the compiler

Further tutorial questions

- enable debugging in the `slist.cc` file and watch for the addresses of the new elements created and deleted
- when reverse is called - how many new elements are created when the
 - recursive version is run
 - when the iterative version is run