

## List implementation in C++

- many times in Computer Science we need to maintain dynamically adjustable lists
- consider the following C function definitions

```
/*  
 * create an empty list.  
 * pre-condition: none.  
 * post-condition: returns an empty list.  
 */  
  
list empty (void);
```

# List specification

```
/*  
 * is_empty  
 *   pre-condition:  an initialised list.  
 *   post-condition: returns true if list is empty.  
 */  
  
bool is_empty (list l);
```

## List specification

```
/*  
 * cons  
 *   pre-condition: an initialised list.  
 *   post-condition: adds, i, to the end of list, l.  
 */  
list cons (list l, int i);
```

## List specification

```
/*  
 * head  
 *   pre-condition: an initialised list containing 1 or more  
 *                  elements.  
 *   post-condition: returns the data associated with the first  
 *                  element, the list is unaltered.  
 */  
int head (list l);
```

## List specification

```
/*  
 * tail  
 *   pre-condition: an initialised list containing 1 or more  
 *                  elements.  
 *   post-condition: removes and deletes the first element of the list.  
 *                  returns the remainder of the list.  
 */  
list tail (list l);
```

## C++ definition of the list methods in a class

■ `c++/lists/single-list/int/slist.h`

```
#if !defined(SLISTH)
#  define SLISTH

/*
 * single linked list implementation.
 */

class element
{
public:
  element *next;
  int      data;
};
```

## C++ definition of the list methods in a class



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

```
class slist
{
    ...    // missing code

public:
    ...    // missing code

    slist empty (void);
    bool  is_empty (void);
    slist cons (int i);
    int   head (void);
    slist tail (void);
};
#endif
```

## Comments on the class

- notice that because we are using a class we no longer need to pass the list as a parameter
  - it is implied by the object and reference as `this`
- notice that we specify which methods are visible for the user (via the keyword `public:`)



## Example 1: how might this code be used?

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

```
/*  
 * test-slist.cc - test code for the slist.cc module.  
 */  
  
#include <slist.h>  
#include <cassert>  
  
main ()  
{  
    {  
        slist l;    // created and automatically initialised  
        assert (l.is_empty ());  
        assert (l.empty ().is_empty ());    // recreated empty  
    }  
}
```

## Example 2



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

```
{  
  slist l;  
  assert (l.cons (12).head () == 12);  
}
```

## Example 3, 4

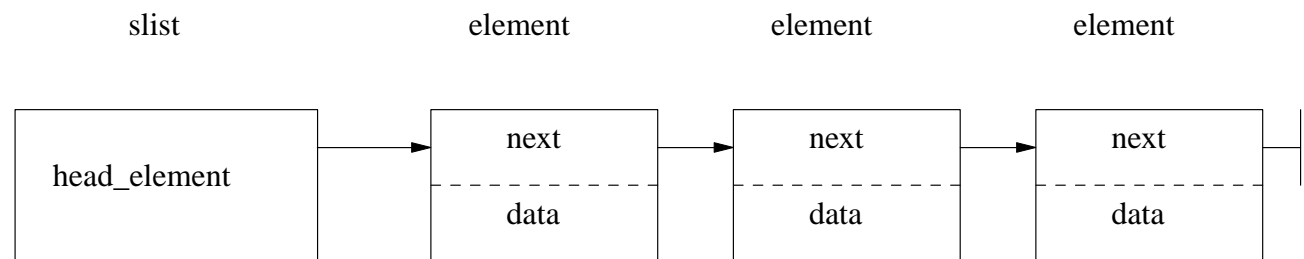
■ `c++/lists/single-list/int/test-slist.cc`

```
{  
  slist l;  
  assert (l.cons (12).tail ().is_empty ());  
}
```

■

```
{  
  slist l;  
  
  l = l.cons (12);  
  l = l.tail ();  
  assert (l.is_empty ());  
}
```

# Diagram showing a possible implementation of the list



# slist.h revisited

■ `c++/lists/single-list/int/slist.h`

```
#if !defined(SLISTH)
#  define SLISTH

/*
 * single linked list implementation.
 */

class element
{
public:
  element *next;
  int     data;
};
```

## slist.h revisited

■ `c++/lists/single-list/int/slist.h`

```
class slist
{
private:
    element *head_element;
    element *duplicate_elements (element *e);
    element *delete_elements (element *h);

public:
    slist (void);
    ~slist (void);
    slist (const slist &from);
    slist& operator= (const slist &from);

    slist empty (void);
    bool is_empty (void);
    slist cons (int i);
    int head (void);
    slist tail (void);
};
#endif
```

## slist.h revisited

- notice we introduce the `head_element` field as a private field
- we also have some private helper methods: `duplicate_elements` and `delete_elements`
- which duplicate and delete all elements

## The C++ rule of three!

- most important!
- you must consider implementing : destructor, copy and assignment, methods for your class

■ `c++/lists/single-list/int/slist.h`

```
~slist (void);           // destructor  
slist (const slist &from); // copy  
slist& operator= (const slist &from); // assignment
```

- the compiler will call these methods when evaluating expressions



## The C++ rule of three!

```
static int a, b, c, t;  
...  
t = c * (a+b)  
...
```

- notice the compiler will need to compute  $a+b$  first and store this in a temporary before multiplying it by  $c$
- in a similar way the C++ compiler will need to make temporary copies of the list when translating:

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

```
l.cons (12).tail ().is_empty ()
```

# Output

- we can add an output method for this class overloading the traditional C++ shift operator <<

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

```
friend std::ostream& operator<< (std::ostream& os, const slist& l);
```

- check that this is inside `slist.h` and examine the implementation in `slist.cc`

## slist::slist (void)

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

```
/*  
 * slist - constructor, builds an empty list.  
 *     pre-condition:  none.  
 *     post-condition: list is created and is empty.  
 */  
  
slist::slist (void)  
    : head_element(0)  
{  
}
```

## Deconstructor (first of the three!)

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

```
/*  
 * ~slist - deconstructor, releases the memory attached to the list.  
 *      pre-condition:    none.  
 *      post-condition:   list is empty.  
 */  
  
slist::~~slist (void)  
{  
    head_element = delete_elements (head_element);  
}
```

## slist::delete\_elements

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

```
/*
 * delete_elements - delete all elements specified by, h.
 *                   pre-condition:  h points to a list of elements.
 *                   post-condition: zero is returned and all
 *                   elements are deleted.
 */

element *slist::delete_elements (element *h)
{
    while (h != 0) {
        element *t = h;
        h = h->next;
        if (debugging)
            printf ("wanting to delete 0x%p\n", t);
        else
            delete t;
    }
    return 0;
}
```

## Copy operator (second of the big three!)



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

```
/*  
 * copy operator - redefine the copy operator.  
 * pre-condition : a list.  
 * post-condition: a copy of the list and its elements.  
 */  
  
slist::slist (const slist &from)  
{  
    head_element = duplicate_elements (from.head_element);  
}
```

## slist::duplicate\_elements

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

```
/*  
 * duplicate_elements - return a copy of all elements found in, e.  
 *                   pre-condition: e points to a list of elements.  
 *                   post-condition: a duplicate list is returned.  
 */  
  
element *slist::duplicate_elements (element *e)  
{  
    element *h = 0;  
    element *l = 0;  
    element *n;
```

## slist::duplicate\_elements

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

```
while (e != 0)
{
    n = new element;
    n->data = e->data;
    n->next = 0;
    if (h == 0)
        h = n;
    else
        l->next = n;
    l = n;
    e = e->next;
}
return h;
}
```



## Assignment operator (3rd of the big three!)

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

```
/*  
 * operator= - redefine the assignment operator.  
 *           pre-condition : a list.  
 *           post-condition: a copy of the list and its elements.  
 *           We delete 'this' lists elements.  
 */  
  
slist& slist::operator= (const slist &from)  
{  
    head_element = delete_elements (head_element);  
    head_element = duplicate_elements (from.head_element);  
}
```

## slist::empty

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

```
/*  
 * empty - returns a new empty list.  
 *       pre-condition:  none.  
 *       post-condition: a new empty list is returned.  
 */  
  
slist slist::empty (void)  
{  
    slist *l = new slist;  
    return *l;  
}
```

## slist::is\_empty

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

```
/*  
 * is_empty - returns true if list is empty.  
 */  
  
bool slist::is_empty (void)  
{  
    return head_element == 0;  
}
```

## slist::cons (int i)

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

```
/*  
 * cons - concatenate i to slist.  
 *      pre-condition: none.  
 *      post-condition: returns the list which has i at its head  
 *                      and the remainder of contents as, slist.  
 */  
  
slist slist::cons (int i)  
{  
    element *e = new element;  
  
    e->data = i;  
    e->next = head_element;  
    head_element = e;  
    return *this;  
}
```

## slist:head

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

```
/*
 * head - returns the data at the front of the list.
 *       pre-condition : slist is not empty.
 *       post-condition: data at the front of the list is returned.
 *                       slist is unchanged.
 */

int slist::head (void)
{
    assert (! is_empty());
    return head_element->data;
}
```

## slist::tail

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

```
/*  
 * tail - opposite of cons. Remove the head value and return  
 * the remainder of the list.  
 * pre-condition: non empty list.  
 * post-condition: return the list without the first element.  
 */  
  
slist slist::tail (void)  
{  
    element *e = head_element;  
  
    assert (! is_empty());  
    head_element = head_element->next;  
    if (debugging)  
        printf ("wanting to delete 0x%p\n", e);  
    else  
        delete e;  
    return *this;  
}
```

## Tutorial Questions

- check out the example code

```
$ git clone https://github.com/gaiusm/examples  
$ cd examples/c++/lists/single-list/int  
$ make
```

- now single step the `a.out` executable using using `gdb`
  - satisfy your self that the big three are being invoked (copy, assignment and destructor methods)

## Tutorial Questions

- implement the method
  - `int length (void);` and add this too the class `slist`
  - `length` returns the number of elements in `slist`
  - write some test code for `length`
  
- implement the method
  - `slist cons (slist l);`
  - this method must concatenate the contents of list, `l`, onto `this`
  - take case as `l` might be the same as `this`
  
- implement the method
  - `slist reverse (void);`



## Tutorial Questions

### ■ implement

```
/*  
 * slice - return a slice of the list.  
 *     pre-condition:  an initialised non empty list.  
 *     post-condition: the original list is unaltered.  
 *                   A new list is returned which is  
 *                   a copy of elements, l..r-1  
 *                   Negative values of l and r  
 *                   index from the right (aka Python).  
 */  
  
slist slist::slice (int l, int r);
```