

## C++ Templates

- during this lecture we will rewrite our `int` based `slist` to take any type
- we will learn how to create and use templates

## C++ Templates

- recall the `list` class which manipulated lists of `ints`
- if we wanted lists of `fracts` we could obviously rewrite our `list` class
  - problem with maintainence
- we might consider writing the `slist` class using a huge C/C++ macro
  - this would work but would be the wrong way
- fortunately C++ has templates
  - which provide this functionality together with language rigour

## slist int class

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

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

## slist int class

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

```
class slist
{
private:
    element *head_element;
    element *duplicate_elements (element *e);
    element *delete_elements (void);
    slist cons (element *e);
    friend std::ostream& operator<< (std::ostream& os, const slist& l);

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);
    slist cons (slist l);
    slist reverse (void);
};
```

## slist int class

- of course we would also need to change all occurrences of int in `++/lists/single-list/int/slist.cc`

## Template class for slist

- `++/lists/single-list/template/slist.h`

```

/*
 * single linked list implementation.
 */

template <class T> class element
{
public:
    element<T> *next;
    T          data;
};

```

- notice the keywords: `template`, `<class` and identifier `T`
  - we are declaring a template and creating a parameter to this class, a typename `T`

## Template class for slist

- can use the keyword `typename` if desired, exactly the same meaning:

```

template <typename T> class element
{
public:
    element<T> *next;
    T          data;
};

```

## Template class for slist

- `++/lists/single-list/template/slist.h`

```

/* prototype for our shift left operator */
template <typename T> std::ostream& operator<< (std::ostr
const sli

template <class T> class slist
{
private:
    element<T> *head_element;
    element<T> *e_dup (element<T> *e);
    element<T> *e_delete (element<T> *h);
    element<T> *e_tail (element<T> *l);
    int e_length (element<T> *l);
};

```

## Template class for slist

c++/lists/single-list/template/slist.h

```
public:
    slist (void);
    ~slist (void); // deconstructor
    slist (const slist &from); // copy
    slist& operator= (const slist &from); // assignment
    friend std::ostream& operator<< >> (std::ostream& os,
                                         const slist<T>& l);

    slist empty (void);
    bool is_empty (void);
    slist cons (T i);
    T head (void);
    slist tail (void);
    slist dup (void);
    int length (void);
};
```

## constructor slist

c++/lists/single-list/template/slist.h

```
/*
 * slist - constructor, builds an empty list.
 * pre-condition: none.
 * post-condition: list is created and is empty.
 */

template <class T> slist<T>::slist ()
    : head_element(0)
{
}
```

## Implementing the slist class

- notice that when we built the int based slist class we had two files:

- slist.h

- for the class definition

- slist.cc

- for the class implementation

- when building a template version of slist we put all the class and code into one header file
  - in effect the compiler macro processes it to generate type specific instances of the same file (internally)

## deconstructor slist

c++/lists/single-list/template/slist.h

```
/*
 * ~slist - deconstructor, releases the memory attached
 * pre-condition: none.
 * post-condition: list is empty.
 */

template <class T> slist<T>::~~slist (void)
{
    head_element = e_delete (head_element);
}
```

**e\_delete**

c++/lists/single-list/template/slist.h

```

/*
 * e_delete - delete all elements specified by, h.
 */
template <class T> element<T> *slist<T>::e_delete (element
{
    while (h != 0) {
        element<T> *t = h;
        h = h->next;
        if (debugging)
            printf ("wanting to delete 0x%p\n", t);
        else
            delete t;
    }
    return 0;
}

```

**copy slist**

c++/lists/single-list/template/slist.h

```

/*
 * copy operator - redefine the copy operator.
 *                 pre-condition : a list.
 *                 post-condition: a copy of the list a
 */
template <class T> slist<T>::slist (const slist<T> &from)
{
    head_element = e_dup (from.head_element);
}

```

**assignment operator**

c++/lists/single-list/template/slist.h

```

/*
 * operator= - redefine the assignment operator.
 *             pre-condition : a list.
 *             post-condition: a copy of the list and
 *             We delete 'this' lists
 */
template <class T> slist<T>& slist<T>::operator= (const s
{
    if (this->head_element == from.head_element)
        return *this;

    head_element = e_delete (head_element);
    head_element = e_dup (from.head_element);
}

```

**cons**

c++/lists/single-list/template/slist.h

```

/*
 * cons - concatenate i to slist.
 *        pre-condition: none.
 *        post-condition: returns the list which has i
 *        and the remainder of contents
 */
template <class T> slist<T> slist<T>::cons (T i)
{
    element<T> *e = new element<T>;

    e->data = i;
    e->next = head_element;
    head_element = e;
    return *this;
}

```

**Output << method**

c++/lists/single-list/template/slist.h

```

/*
 * operator<< - shift left (output) operator.
 *             pre-condition : an initialised list.
 *             post-condition: list printed and stre.
 */

```

**Output << method**

c++/lists/single-list/template/slist.h

```

template<class T>
std::ostream& operator<< (std::ostream& os, const slist<T>& l)
{
    element<T> *e = l.head_element;
    bool seen = false;

    os << "[";
    while (e != 0)
    {
        if (seen)
            os << ", ";
        os << e->data;
        e = e->next;
        seen = true;
    }
    os << "]";
    return os;
}

```

**int test code**

c++/lists/single-list/template/slist.h

```

{
    slist <int> l;

    l = l.cons (1).cons (2).cons (3).cons (4);
    printf("length of l = %d\n", l.length ());
    assert (l.length () == 4);
}

```

**fract test code**

c++/lists/single-list/template/test-flist.cc

```

#include <slist.h>
#include <cassert>
#include <fract.h>

main ()
{
    {
        slist <fract> l;

        assert (l.is_empty ());
        assert (l.empty ().is_empty ());
    }
}

```

## fract test code

■ `++/lists/single-list/template/test-flist.cc`

```
{
  slist <fract> l;
  fract f12 = fract (1, 2);
  fract f13 = fract (1, 3);
  fract f14 = fract (1, 4);
  fract f15 = fract (1, 5);

  l = l.cons (f12).cons (f13).cons (f14).cons (f15);
  printf("length of l = %d\n", l.length ());
  assert (l.length () == 4);
  std::cout << l << "\n";
}
```

## Running test code

■

```
$ make test-fract
g++ -c -I. -g -O0 -I../../fractions test-flist.cc
g++ -c -I. -g -O0 -I../../fractions ../../fractions
fract list tests
g++ test-flist.o fract.o
valgrind ./a.out
==15456== Memcheck, a memory error detector
==15456== Copyright (C) 2002-2011, and GNU GPL'd, by Juli
==15456== Using Valgrind-3.7.0 and LibVEX; rerun with -h
==15456== Command: ./a.out
==15456==
length of l = 4
[1/5, 1/4, 1/3, 1/2]
```

## Running test code

- notice the layered overloading of <<
- our list can be written out using <<
- in turn each element is written out using <<
- in our example this called the `fract` version of << which then used string and `int` versions of <<

## Tutorial

- currently the template `slist` library does not have `reverse` and `cons (slist l)` as implemented in a previous lecture in the `int slist` library
  - implement these in the template `slist.h` library

## Tutorial

- implement a 3x3 matrix template library for any type
- which allows you to create a 3x3 matrix of any type
  - multiply two matrices
- get and set contents of a matrix
- print out a matrix