Templates

Announcement: Midsem exam

- Midsem exam next week (March 7) during the lab slot
  - In-lab challenge(s)
  - Open-book, open-notes, open-internet.
- Syllabus: All the lab sessions so far (including today)
- Exam will begin sharply at 2 PM, end at 5 PM, with no extensions.
  - Grace days are not applicable.
- Please be seated in the lab by 2 PM.

Introduction

- Templates provide a way to define one function or class which can handle many different data types.
  - Very useful if the functionality to be implemented is independent of the underlying data type, or can be generalised across multiple types.
- We can define both function templates and class templates.

Function Template

Suppose we want to find whether an element is present in an array.
This can be implemented generically, without knowing the type of array or element

```
template<class elemType>
int findElem(elemType * arr,
     elemType elem, int size)
{
    for (int i = 0; i < size; i++)
        if (arr[i] == elem)
            return i;
    return -1;
}
```

```
int main()
{
    int intArr[] = {1,2,3,4,5,6};
    double doubArr[] = {1.0, 2.0, 3.0,
                        4.0, 5.0, 6.0};
    IntCell intCellArr[] = {1,2,3,4,5,6};
    cout << findElem(intArr, 1, 6) << endl;
    cout << findElem(doubArr, 1.0, 6) << endl;
    cout << findElem(intCellArr, IntCell(1), 6)
        << endl;
}
```

Class Template

```
template <class Object>
class MemoryCell
{
  private:
    Object storedValue;
  public:
    explicit MemoryCell(const Object & initialValue = Object()) :
storedValue(initialValue) {}
    const Object & read() const
    {
      return storedValue;
    }
    void write(const Object & x)
    {
      storedValue = x;
    }
};
```

Class Template

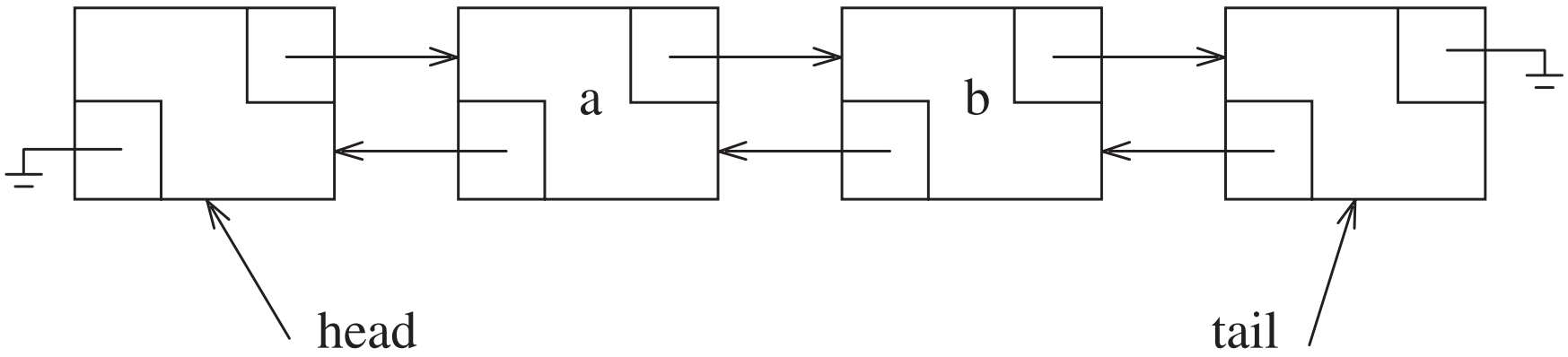
```
1: int main()
2: {
3:   MemoryCell<int> m1;
4:   string str = "hello";
5:   MemoryCell<string> m2(str);
6:   m1.write(37);
7:   m2.write(m2.read() + "world");
8:   cout << m1.read() << " " << m2.read() << endl;
9: }
```

Question: What happens if we update str to str + “world”, at line-7?
Will it give the same output as above?

Case Study: Doubly Linked List Implementation from Weiss, Chapter 3

```
1   template <typename Object>
2   class List
3   {
4     private:
5       struct Node
6         { /* See Figure 3.13 */ };
7
8     public:
9       class const_iterator
10        { /* See Figure 3.14 */ };
11
12      class iterator : public const_iterator
13        { /* See Figure 3.15 */ };
14
78
79      private:
80        int   theSize;
81        Node *head;
82        Node *tail;
83
```

Doubly Linked List



struct Node

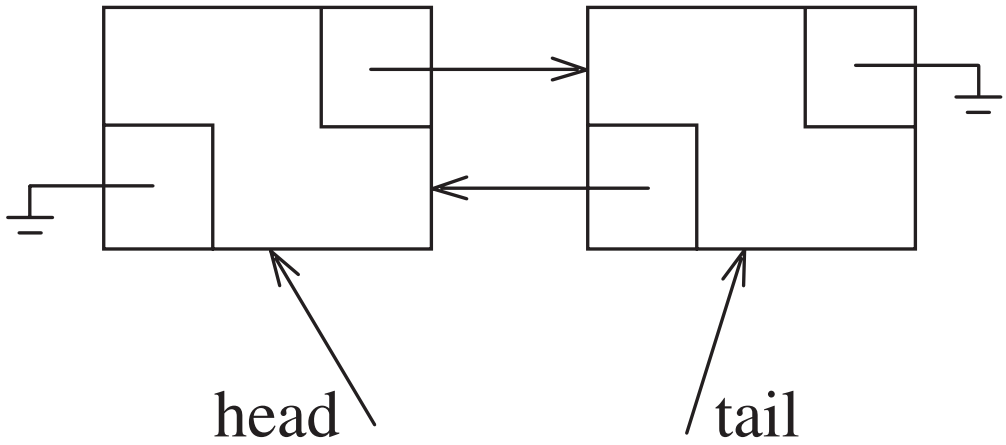
```
1       struct Node
2       {
3           Object  data;
4           Node   *prev;
5           Node   *next;
6
7           Node( const Object & d = Object{ }, Node * p = nullptr,
8                                               Node * n = nullptr )
9             : data{ d }, prev{ p }, next{ n } { }
10
11          Node( Object && d, Node * p = nullptr, Node * n = nullptr )
12            : data{ std::move( d ) }, prev{ p }, next{ n } { }
13      };
```

Zero-parameter list constructor

```
1       List( )
2         { init( ); }
3
```

```
43      void init( )
44      {
45          theSize = 0;
46          head = new Node;
47          tail = new Node;
48          head->next = tail;
49          tail->prev = head;
50      }
```

Empty linked list just after initialisation



const_iterator class : Protected members

const_iterator class is simply a wrapper around a Node* pointer

```
class const_iterator
{
    ...

  protected:
    Node *current;

    Object & retrieve( ) const
      { return current->data; }

    const_iterator( Node *p ) : current{ p }
      { }

    friend class List<Object>;
```

Use of const_iterator, iterator in List class

```
28
29        iterator begin( )
30          { return { head->next }; }
31        const_iterator begin( ) const
32          { return { head->next }; }
33        iterator end( )
34          { return { tail }; }
35        const_iterator end( ) const
36          { return { tail }; }
37
```

const_iterator class : Public members

```
public:
  const_iterator( ) : current{ nullptr }
    { }

  const Object & operator* ( ) const
    { return retrieve( ); }

  const_iterator & operator++ ( )
  {
      current = current->next;
      return *this;
  }
```

```
const_iterator operator++ ( int )
{
    const_iterator old = *this;
    ++( *this );
    return old;
}

bool operator== ( const const_iterator & rhs ) const
  { return current == rhs.current; }
bool operator!= ( const const_iterator & rhs ) const
  { return !( *this == rhs ); }
```

Note that this function returns-by-value

Note that * operator returns-by-constant-reference

iterator class

```
class iterator : public const_iterator
{
  public:
    iterator( )
      { }

    Object & operator* ( )
      { return const_iterator::retrieve( ); }
    const Object & operator* ( ) const
      { return const_iterator::operator*( ); }

    iterator & operator++ ( )
    {
        this->current = this->current->next;
        return *this;
    }

    iterator operator++ ( int )
    {
        iterator old = *this;
        ++( *this );
        return old;
    }

  protected:
    iterator( Node *p ) : const_iterator{ p }
      { }

    friend class List<Object>;
};
```

*** operator can return a non-const reference**

Copy constructor and operator: Deep Copying

```
List( const List & rhs )
{
    init( );
    for( auto & x : rhs )
        push_back( x );
}

List & operator= ( const List & rhs )
{
    List copy = rhs;
    std::swap( *this, copy );
    return *this;
}
```

Iterates through all the nodes in rhs

Creates and inserts a new node containing object x

Range for loops require begin(), end(), ++, and * operators

Use the copy constructor

Move the copy to ‘this’ object

auto: Automatic type inference

- `auto` automatically infers the type of a declared variable using its initialization or using function signatures.
  - Very useful for complicated types.

```
List<int> l;
...
for (List<int>::iterator it = l.begin(); it != l.end(); it++)
    ...
```

auto: Automatic type inference

- `auto` automatically infers the type of a declared variable using its initialization or using function signatures.
  - Very useful for complicated types.

It is recommended to use auto as much as possible

```
List<int> l;
...
for (auto it = l.begin(); it != l.end(); it++)
    ...
```