Class Templates

Version 1: Dr. Ofir Pele
Version 2: Dr. Erel Segal-Halevi

String Stack

```
class StrStk {
public:
  StrStk():m_first(nullptr) { }
  void push (string const& s )  {m_first=new Node(s,m_first);}
  bool isEmpty() const {return m_first==nullptr;}
  const string& top () const {return m_first->m_value;}
  void pop ()
  {Node *n=m_first; m_first=m_first->m_next; delete n;}
  ~StrStk() { while (!isEmpty()) pop(); }
private:
  StrStk(StrStk const& rhs);    StrStk& operator=(StrStk const& rhs);
  struct Node {
    string m_value;
    Node* m_next;
    Node(string const& v ,Node* n):m_value(v),m_next(n) { }
  };
  Node* m_first;
};
```

Generic Classes

- The actual code for maintaining the stack has nothing to do with the particulars of the string type.
- Can we have a generic implementation of stack?

Generic Stack (folder 2)

```
template <typename T> class Stk {
public:
  Stk():m_first(nullptr) { }
  ~Stk() { while (!isEmpty()) pop(); }
  void push (const T& s)  {m_first=new Node(s,m_first);}
  bool isEmpty() const {return m_first==nullptr;}
  const T& top () const {return m_first->m_value;}
  void pop ()
  {Node *n=m_first; m_first=m_first->m_next; delete n;}
private:
  Stk(const Stk& rhs); Stk& operator=(const Stk& rhs);
  struct Node {
    T m_value;
    Node* m_next;
    Node(const T& v ,Node* n):m_value(v),m_next(n) { }
  };
  Node* m_first;
};
```

Class Templates

```
template<typename T>
class Stk
{
  ...
};

Stk<int> intList; // T = int
Stk<string> stringList; // T = string
```

Class Templates

The code is similar to non-template code, but:

- Add `template<…>` statement before the class definition
- Use template argument as type in class definition
- To implement methods outside the class definition (but still in header: .h.hpp, not in a cpp file!):

```
template <typename T>
bool Stk<T>::isEmpty() const
{
  return m_first==nullptr;
}
```

Example of generic programming - Iterators

Constructing a List

- We want to initialize a stack from a primitive array.

- int arr[6];
- We can use a pointer to initial position and **one to the position after the last**:
- `Stk<int> myStack(`
  `arr,`**`arr+sizeof(arr)/sizeof(*arr)`**`);`

Constructing a List

```
// Fancy copy from array
template< typename T >
Stk<T>::Stk<T>(const T* begin, const T* end) {
    for(; begin!=end; ++begin) {
        push(*begin);
    }
}
```

Pointer Paradigm

Code like:

```
const T* begin=theList;
const T* end=
list+sizeof(theList)/sizeof(*theList);
for(; begin!=end; ++begin)
{
   // Do something with *begin
}
```

- Applies to all elements in [begin,end-1]
- Common in C/C++ programs
- Can we extend it to other containers?

Iterator

- **Object that behaves “almost” like a pointer**
- Allows to iterate over elements of a container

Example:

```
Stk<int> L;
...
Stk<int>::iterator i;
for( i = L.begin(); i != L.end(); ++i)
    cout << " " << *i << "\n";
```

Iterators

To emulate pointers, we need:

1. copy constructor
2. `operator= (copy)`
3. `operator==, != (compare)`
4. `operator* (access value)`
5. `operator++ (increment)`

And maybe:

6. `operator[] (random access)`
7. `operator+= / -= (random jump)`
8. ...

Stk<T> iterator (folder 2)

Create an inner class, keep a pointer to a node.

```
class iterator
{
private:
   Node *m_pointer;
};
```

Provides encapsulation, since through such an iterator we cannot change the structure of the list

Initializing a Stk

We now want to initialize a stack from using parts of another stack. Something like:

```
Stk(iterator begin, iterator end) {
    for(; begin!=end; ++begin) {
        push(*begin);
    }
}
```

Initializing a Stk

Compare:

```
Stk<T>::Stk<T>(iterator begin, iterator end) {
    for(; begin!=end; ++begin) {
        push(*begin);
    }
}
```

To:

```
Stk<T>::Stk<T>(const T* begin, const T* end) {
    for(; begin!=end; ++begin) {
        push(*begin);
    }
}
```

Generic Constructor

The code for copying using

- T*
- Stk<T>::iterator

are essentially identical on purpose --- iterators mimic pointers!

Can we write the code once?

Yes: template inside template (folder 2)

Class exercise

Write a function for summing all elements of a container.

The function should work with all kinds of containers:

- Native array;
- Linked list;
- vector;
- user defined...