

Class Templates

Version 1: Dr. Ofir Pele

Version 2: Dr. Erel Segal-Halevi

	C++	Java
Keyword	template	none
Type arguments	Any	Only objects (https://stackoverflow.com/a/18021680/827927)

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> intStack; // T = int
```

```
Stk<string> stringStack; // 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;
}
```

Containers and Iterators

Goal

- Do something with all the elements of a container.
- For example: initialize a stack from a container.
 - *container = primitive array, vector, list, stack...*

Special case: primitive array

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

Constructing a Stack

// Fancy copy from array

```
template< typename T >
```

```
Stk<T>::Stk<T>(const T* begin, const T* end) {
```

```
    for(; begin!=end; ++begin) {
```

```
        push(*begin);
```

```
    }
```

```
}
```

Generalizing to other containers

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

Usage 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, at least:

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

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...