Queues, templates, stacks and deques

Basic functionality of queues
Use of templates
Basic functionality of stacks
Basic functionality of deques

Read chapter 17 of the textbook!

These slides will cover some of the contents, but the textbook is much more detailed.

Definition of queues

- A queue is an object container for which objects are inserted and removed according to the first-in-first-out (FIFO) principle
- Objects are added to the rear of the queue and are removed from the front of the queue
- Queues use a linked list as the underlying data structure

```
class Queue
{
...
   private:
   LinkedList data;
   int used;
};
```

Functionality of queue

- An instance of the queue class supports two fundamental methods:

```
void enqueue(const value_type& obj) // Add obj to the rear of the queue
value_type dequeue() // Remove and return the object at the head
                     // of the queue, error if the queue is empty
```

- Plus the supporting methods:

```
int size() const // Return the number of objects in the queue
bool is_empty() const // Return a boolean indicating whether the queue
                      // is empty
value_type& front() // Return a reference to the object at the front of
                    // the queue, NULL if the queue is empty
```

The .h file for Queue

```
class Queue
{
   public:
   typedef LinkedList::value_type value_type; // uses the same value type of LinkedList

   // Mutator member functions
   Queue();
   ~Queue();
   void enqueue(const value_type& entry);
   value_type dequeue();
   bool is_empty() const;

   // Query member functions
   int size() const;
   value_type& front();

   private:
   LinkedList data;
   int used;
};
```

Some implementations (.cpp)

```
void Queue::enqueue(const value_type& entry) {
   ++used;
   data.add_to_tail(entry); // uses add_to_tail from LinkedList
}

value_type Queue::dequeue() {
   --used;
   return data.remove_from_head(); // reused from LinkedList
}

value_type& Queue::front(){
   data.start(); // makes current move to head
   return data.getCurrent();
}
```

Templates

Basic functionality of queues
Use of templates
Basic functionality of stacks

Check pages 967-978 of the textbook!

These slides will cover some of the contents, but the textbook is much more detailed.

Introduction to templates

- The template is the C++ mechanism for producing generic functions and classes. It goes one step further than `typedef`.
- E.g. it could be used to avoid the `typedef <class> value_type;` statement in definition of our `Node` class
- It allows the use of the same data structure with multiple data types, in the same project.
- There are function templates and class templates. Let's start with function templates.

Function templates

- A function template is not a function
- It is a blueprint for what can become a function at compile time
- For example we could define a function `find_max()` that took two instances of any class and returned the instance that is 'bigger' by some class-defined criterion
- Function templates are defined in a .h file in the same way as are class profiles
- Normally, if you have several function templates in your project, you group them all in the same .h file, say `functions.h`

Function templates (cont)

```
// This is the file functions.h

// Function find_max(const value_type& arg1, const value_type& arg2)
// Returns arg1 if it is bigger than arg2, and arg2 otherwise.
// Item objects must implement the > operator. A copy constructor is also required

template <typename value_type>
const value_type& find_max(const value_type& arg1, const value_type& arg2)
{
   if (arg1 > arg2) {return arg1;}
   else {return arg2;}
}
```

- This function can be used by any class that has `#include "functions.h"`

Function templates

- Note the requirement that it must be possible for the `>` operator to be applied to the arguments
  - In the case of non-primitive types, the `>` operator must be overloaded before `find_max()` can be used for that type
- The effect of the template is that the compiler creates overloaded implementations of `find_max()` for each type to which it is applied in the code

Template example

- Consider the following statements:

```
#include "functions.h"
...
Account acc1;
Account acc2;
int i1 = 3;
int i2 = 5;
...
Account acc3 = find_max(acc1, acc2);
int i3 = find_max(i1, i2);
```

- The compiler would produce the following code:

```
const Account& find_max(const Account& arg1, const Account& arg2) {
   if (arg1 > arg2) {return arg1;} else {return arg2;}
}
const int& find_max_max(const int& arg1, const int& arg2) {
 if (arg1 > arg2) {return arg1;} else {return arg2;}
}
```

Template example (cont)

- `Account` must have an overloaded `>` operator:

```
// In Account.h
bool operator >(const account& a1, const account& a2);

// In Account.cpp
bool operator >(const account& a1, const account& a2) {
    return (a1.balance() > a2.balance());
};
```

- The operator `>` is already defined for `int` in the C++ language so it can be used by the `find_max()` function when applied to `int`
- Note that we can now apply `find_max()` to any type for which `>` is defined

Class templates

- Class templates are defined similarly to function templates.
- For example, a class template for `Node` would be defined as follows:

```
template <typename value_type>
class Node {
   public:
   Node();
   ~Node();
...
   private:
   value_type data;
   Node* next;
}
```

- In simple terms, the `value_type` from the `typedef` declaration will now refer to the `value_type` from the template declaration.

Comments

- Note that if you wanted to use `LinkedList` and `Node` with, say, `int` and `Account` in the same code without templates, you would need to implement two `Node` classes:

`NodeInt` with `typedef int value_type`, and
`NodeAcc` with `typedef Account value_type`.

- And two `LinkedList` classes

```
LinkedListInt intList; // with typedef NodeInt:value_type value_type
LinkedListAcc accountList; // with typedef NodeAcc:value_type value_type
```

- That would generate too much duplicated code... and imagine if you wanted to use `LinkedList` with 5 different types.

Example

- Let's change `LinkedList` and `Node` from last week to class templates

DEMO

Stacks

Basic functionality of stacks

Read Chapter 17 of the textbook!

These slides will cover some of the contents, but the textbook is much more detailed.

Definition of stacks

- A stack is an object container for which objects are inserted and removed according to the *last-in-first-out* (LIFO) principle
- Only the most recently inserted (or "pushed") object can be removed (or "popped") at any time
- It also uses a linked list as the underlying data structure

```
class Stack
{
...
   private:
   LinkedList data;
   int used;
};
```

Functionality of stacks

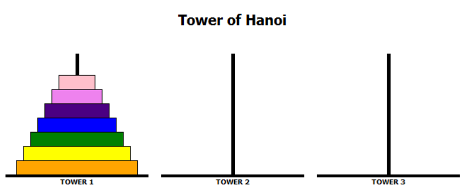
- An instance of the stack class supports two fundamental methods:

```
void push(const value_type& obj) // Insert obj at the top of the stack
value_type pop() // Remove and return the top object from the stack,
                 // error if the stack is empty
```

- Plus the supporting methods:

```
int size() const // Return the number of objects in the stack
bool is_empty() const // Return a boolean indicating whether the stack
                      // is empty
value_type& top() // Return a reference to the object at the top of
                  // the stack, NULL if the stack is empty
```

An example – Tower of Hanoi



Stack underflow and overflow

- Stacks are so important that C++ provides them as a template class of the STL
  - `push(obj), pop(), top(), size() and empty()`
  - Assignment and the copy constructor can be used with stack objects
- An attempt to `pop()` from an empty stack creates a *stack underflow* condition
- An attempt to `push()` onto a full stack causes a *stack overflow* condition

The .h file for Stack (template version)

```
#include "LinkedList.h"
template <typename value_type>
class Stack
{
   public:
   // Mutator member functions
   Stack();
   ~Stack();
   void push(const value_type& entry);
   value_type pop();
   bool isEmpty() const;

   // Query member functions
   int size() const;
   value_type& top();

   private:
   LinkedList<value_type> data;
   int used;
};
#include "Stack.template"
```

Comments on class templates

- In the spirit of separating the profile from the implementation
  - Provide the profile in `TemplateName.h`
  - Provide the "implementation" in a file `TemplateName.template` or with any extension other than `.cpp`
- You *must* have

  `#include "TemplateName.template"` at the end of `TemplateName.h`
  - The profile and its implementation must be in the same file (as far as the compiler is concerned – the template implementation is not a real implementation. Implementation is done during the compilation.)
- Never have any `using` directives in the implementation file

Comments on class templates (cont)

- Every member function implementation must start with the template header `template <typename value_type>`
- The name of the template class must be provided in a form that provides both the class name and the notional application class `value_type& `**`LinkedList<value_type>`**`::getCurrent()`
- The constructor's name does *not* change
  - E.g. `LinkedList<value_type>::LinkedList()`
- When the template class is instantiated, the application class is defined:

```
Stack<char> charStack;
Stack<int> intStack;
```

Comments on class templates (cont)

- Your `Makefile` includes the `.h` file for the template class(es) under the SOURCE field:

```
SOURCES=demo.cpp Node.h LinkedList.h Stack.h
```

Some implementations (.cpp)

```
template <typename value_type>
Stack<value_type>::Stack(){}

template <typename value_type>
Stack<value_type>::~Stack(){}

template <typename value_type>
void Stack<value_type>::push(value_type& obj)
{
    data.add_to_head(obj);
}

template <typename value_type>
value_type Stack<value_type>::pop()
{
    return data.remove_from_head();
}
// etc
```

Deques

Basic functionality of deques

Read Chapter 17 of the textbook!

These slides will cover some of the contents, but the textbook is much more detailed.

Definition of deques

- A deque is a double-ended queue. It supports addition and removal from both the beginning and the end.
- It also uses a linked list as the underlying data structure

```
class Deque
{
...
   private:
   LinkedList list;
   int used;
};
```

Functionality of deques

- An instance of the deque class supports four fundamental methods:

```
void insert_first(const value_type& obj) // Insert obj at the beginning
void insert_last(const value_type& obj) // Insert obj at the end
value_type remove_first() // Remove and return the first element
value_type remove_last() // Remove and return the last element
```

- Plus the supporting methods:

```
int size() const     // Return the number of objects in the deque
bool is_empty() const // Return a boolean indicating whether the deque
                      // is empty
value_type& first() // Return a reference to the first object, NULL if
                    // the deque is empty
value_type& last()  // Return a reference to the last object, NULL if
                    // the deque is empty
```

Comments

- Deques can be used to implement *stacks* and *queues*
- The correspondences between stack and deque methods respectively are:

Stack – to implement	Deque – use the method
size()	size()
is_empty()	is_empty()
top()	first()
push()	insert_first()
pop()	remove_first()

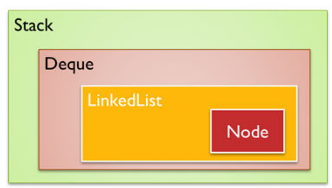
Comments

- Deques can be used to implement *stacks* and *queues*
- The correspondences between stack and deque methods respectively are:

Queue – to implement	Deque – use the method
size()	size()
is_empty()	is_empty()
front()	last()
enqueue()	insert_last()
dequeue()	remove_first()

Wrapping around classes again

```
template <typename value_type>
class Deque
...
private:
      LinkedList<value_type> list;
...
```



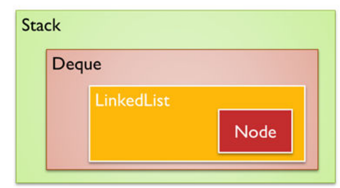
Implementation:

```
template <typename value_type>
void insert_last(const value_type& obj){list.add_to_tail(obj);}

template <typename value_type>
void insert_first(const value_type& obj){list.add_to_head(obj);}
...
```

Wrapping around classes again

```
template <typename value_type>
class Stack
...
private:
      Deque<value_type> data;
...
```



Implementation:

```
template <typename value_type>
void Stack<value_type>::push(const value_type& obj){data.insert_first(obj);}

template <typename value_type>
value_type Stack<value_type>::pop(){data.remove_first(obj);}
...
```

DEMO

See you next week!