C++ Templates

Dr. Gustavo Rodriguez-Rivera

Parameterized Types

- In C++ we have three kind of types:
  - Concrete Type:
    - It is a user defined class that is tied to a unique implementation. Example: an int or a simple class.
  - Abstract Type:
    - It is user-defined class that is not tied to a particular implementation. Example Figure is an abstract class where draw can be Line::draw, Rectangle::draw(). It uses virtual methods and subclassing.
  - Parameterized type:
    - It is a type that takes as parameter another type. Example: Stack<int> creates a stack of type int, Stack<Figure *> will build a stack of entries of type Figure *. This is the base for "Templates".

Templates

- They are parameterized types.
- They allow to implement data structures for different types using the same code, for example :
    - Stack<int> - Stack of type int
    - Stack<double>, Stack of type double
    - Stack<Figure>, Stack of type Figure.

Templates

- A generic class starts with the template definiton:

```
template <typename T>
```

- typename T indicates that T is a type parameter.
- There can be also compile time constants or functions

```
template <typename T, int SIZE>
```

Writing a Template

- Before writing a template it is recommended to write the code of the class without the parameters using a concrete type.
- For example, if you want to write a List template for any type, write first a List class for “int”s (ListInt).
- Once that you compile, test and debug ListInt, then write the template by substituting the “int” by “Data” (the parameter type).
- Also add template <typename Data> before every class, function, and struct.

A ListInt Class

```
ListInt.h
// Each list entry stores int
struct ListEntryInt {
  int _data;
  ListEntryInt * _next;
};
```

A ListInt Class

```
class ListInt {
 public:
  ListEntryInt * _head;

  ListInt();
  void insert(int data);
  bool remove(int &data);
};
```

A ListInt Class

```
ListInt::ListInt()
{
  _head = NULL;
}
```

A ListInt Class

```
void ListInt::insert(int data)
{
  ListEntryInt * e = new ListEntryInt;
  e->_data = data;
  e->_next = _head;
  _head = e;
}
```

A ListInt Class

```
bool ListInt::remove(int &data)
{
  if (_head==NULL) {
    return false;
  }

  ListEntryInt * e = _head;
  data = e->_data;
  _head = e->_next;
  delete e;
  return true;
}
```

A ListGeneric Template

- To implement the ListGeneric Template that can be used for any type we start with ListInt.
- Copy ListInt.h and ListInt.spp into ListGeneric.h.
- Add “template <typename Data> “ before any class, struct or function.
- Substitute “int” by ”Data”
- Where “ListEntryInt” is used, use “ListEntry<Data>” instead.
- Where “ListInt” is used, use “ListGeneric<Data>” instead.

A ListGeneric Template

<u>ListGeneric.h</u>

```
// Each list entry stores data
template <typename Data>
struct ListEntry {
  Data _data;
  ListEntry * _next;
};
```

A ListGeneric Template

```
template <typename Data>
class ListGeneric {
 public:
  ListEntry<Data> * _head;

  ListGeneric();
  void insert(Data data);
  bool remove(Data &data);
};
```

A ListGeneric Template

```
template <typename Data>
ListGeneric<Data>::ListGeneric()
{
  _head = NULL;
}
```

A ListGeneric Template

```
template <typename Data>
void ListGeneric<Data>::insert(Data data)
{
  ListEntry<Data> * e = new ListEntry<Data>;
  e->_data = data;
  e->_next = _head;
  _head = e;
}
```

A ListGeneric Template

```
template <typename Data>
bool ListGeneric<Data>::remove(Data &data)
{
  if (_head==NULL) {
    return false;
  }

  ListEntry<Data> * e = _head;
  data = e->_data;
  _head = e->_next;
  delete e;
  return true;
}
```

Using the Template

- To use the template include “ListGeneric.h”
  #include “ListGeneric .h”

- To instantiate the ListGeneric :

```
//List of int’s
ListGeneric<int> * listInt =
  new ListGeneric<int>();

//List of strings
ListGeneric<const char *> * listString =
        new ListGeneric<const char *>();

Or as local/global vars
ListGeneric<int> listInt; // List of int’s
ListGeneric<const char *> listString; // list of strings
```

A test for GenericList

```
#include <stdio.h>
#include <assert.h>
#include " ListGeneric.h"

int
main(int argc, char **argv)
{
  //////////////////////////////////////
  // testing lists for ints

  ListGeneric<int> * listInt = new ListGeneric<int>();

  listInt->insert(8);
  listInt->insert(9);

  int val;
  bool e;
  e = listInt->remove(val);
  assert(e); assert(val==9);

  e = listInt->remove(val);
  assert(e);
  assert(val==8);
```

Using the Template

```
//////////////////////////
  // testing lists for strings

  ListGeneric<const char *> * listString = new ListGeneric<const char *>();

  listString->insert("hello");
  listString->insert("world");

  const char * s;
  e = listString->remove(s);
  assert(e);
  assert(!strcmp(s,"world"));

  e = listString->remove(s);
  assert(e);
  assert(!strcmp(s,"hello"));
}
```

Iterator Template

- An iterator is a class that allows us to iterate over a data structure.

- It keeps the state of the position of the current element in the iteration.

```
template <typename Data>
class ListGenericIterator {
  ListEntry<Data> *_currentEntry; // Points to the
 current node
  ListGeneric<Data> * _list;
 public:
  ListGenericIterator(ListGeneric<Data> * list);
  bool next(Data & data);
};
```

Iterator Template

```
template <typename Data>
ListGenericIterator<Data>::ListGenericIterator(ListGeneric<Data> * list)
{
  _list = list;
  _currentEntry = _list->_head;
}

template <typename Data>
bool ListGenericIterator<Data>::next(Data & data)
{
  if (_currentEntry == NULL) {
    return false;
  }

  data = _currentEntry->_data;
  _currentEntry = _currentEntry->_next;

  return true;
}
```

Iterator Template

```
void testIterator() {
  ListGeneric<const char *> * listString = new ListGeneric<const char *>();
  const char * (array[]) = {"one","two","three","four","five","six"};
  int n = sizeof(array)/sizeof(const char*);

  int i;
  for (i=0;i<n;i++) {
    listString->insert(array[i]);
  }

  const char * s;
  ListGenericIterator<const char *> iterator(listString);
  while (iterator.next(s)) {
    printf(">>%s\n",s);
    i--;
    assert(!strcmp(s,array[i]));
  }

  printf("Tests passed!\n");
}
```

Default Template Parameters

- You can provide default values to templates. Example:

Stack.h

```
 template <typename T = int, int n = 20>
class Stack {
   T array[n];
    …
 };
```

- At instantiation time:

```
Stack stack1; // Stack of type int of size 20 (default)
Stack<double> stack2; // Stack of type double of size 20
Stack<Figure, 100> stack3; // Stack f type Figure of size 100
```

Function Templates

- Also functions can be parameterized.

```
 template <typename T>
 void swap(T &a, T &b) {
    T tmp = a;
    a = b;
    b = tmp;
 }
…
  int x = 3; int y = 4;
  swap(x,y); // Swaps int vars x, y
  double z1 = 3.567; double z2 = 56;
  swap(z1, z2); // Swaps double vars z1, z2
```

- The compiler will generate instances of the swap function for double and int.