C++ Templates

Sandra Batista, Mark Redekopp, and David Kempe

1.1-1.2

FUNCTION TEMPLATES

Consider a max() function for two different data types

```
int max(int a, int b)
{
  if(a > b) return a;
  else return b;

}
double max(double a, double b)
{
  if(a > b) return a;
  else return b;
}
```

Without templates

Example reproduced from: http://www.cplusplus.com/doc/tutorial/templates/

Function Templates

1. Define a generic function for any type, T
2. May be called for type explicitly or implicitly

```
int max(int a, int b)
{
  if(a > b) return a;
  else return b;

}
double max(double a, double b)
{
  if(a > b) return a;
  else return b;
}
```

Without Templates

```
template<typename T>
T max(const T& a, const T& b)
{
  if(a > b) return a;
  else return b;
}
int main()
{
  int x = max<int>(5, 9); //or
  x = max(5, 9); // implicit max<int> call
  double y = max<double>(3.4, 4.7);
  // y = max(3.4, 4.7);
}
```

With Templates

CLASS TEMPLATES

Motivating Example

Let's consider how we implement a list of integers and list of doubles so far…

```
#ifndef LIST_INT_H
#define LIST_INT_H
struct IntItem {
  int val; IntItem* next;
};
class ListInt{
 public:
   ListInt();  // Constructor
   ~ListInt();  // Destructor
   void push_back(int newval); ...
 private:
   IntItem* head_;
};
#endif
```

```
#ifndef LIST_DBL_H
#define LIST_DBL_H
struct DoubleItem {
  double val; DoubleItem* next;
};
class ListDouble{
 public:
   ListDouble();  // Constructor
   ~ListDouble();  // Destructor
   void push_back(double newval); ...
 private:
   DoubleItem* head_;
};
#endif
```

Templates

Allows the type of variable in a class or function to be a parameter

Compiler will generate separate versions of code for instantiations

- LList<int> my_int_list generates code for int list
- LList<double> my_dbl_list generates code for double list

```
// declaring templatized code
template <typename T>
struct Item {
  T val;
  Item<T>* next;
};

template <typename T>
class LList {
public:
   LList();  // Constructor
   ~LList();  // Destructor
   void push_back(T newval); ...
 private:
   Item<T>* head_;
};


// Using templatized code
//  (instantiating templatized objects)
int main()
{
  LList<int> my_int_list;
  LList<double> my_dbl_list;

  my_int_list.push_back(5);
  my_dbl_list.push_back(5.5125);

  double x = my_dbl_list.pop_front();
  int y = my_int_list.pop_front();
  return 0;
}
```

Writing a template

- Precede class with:

  template <typename T>

  Or

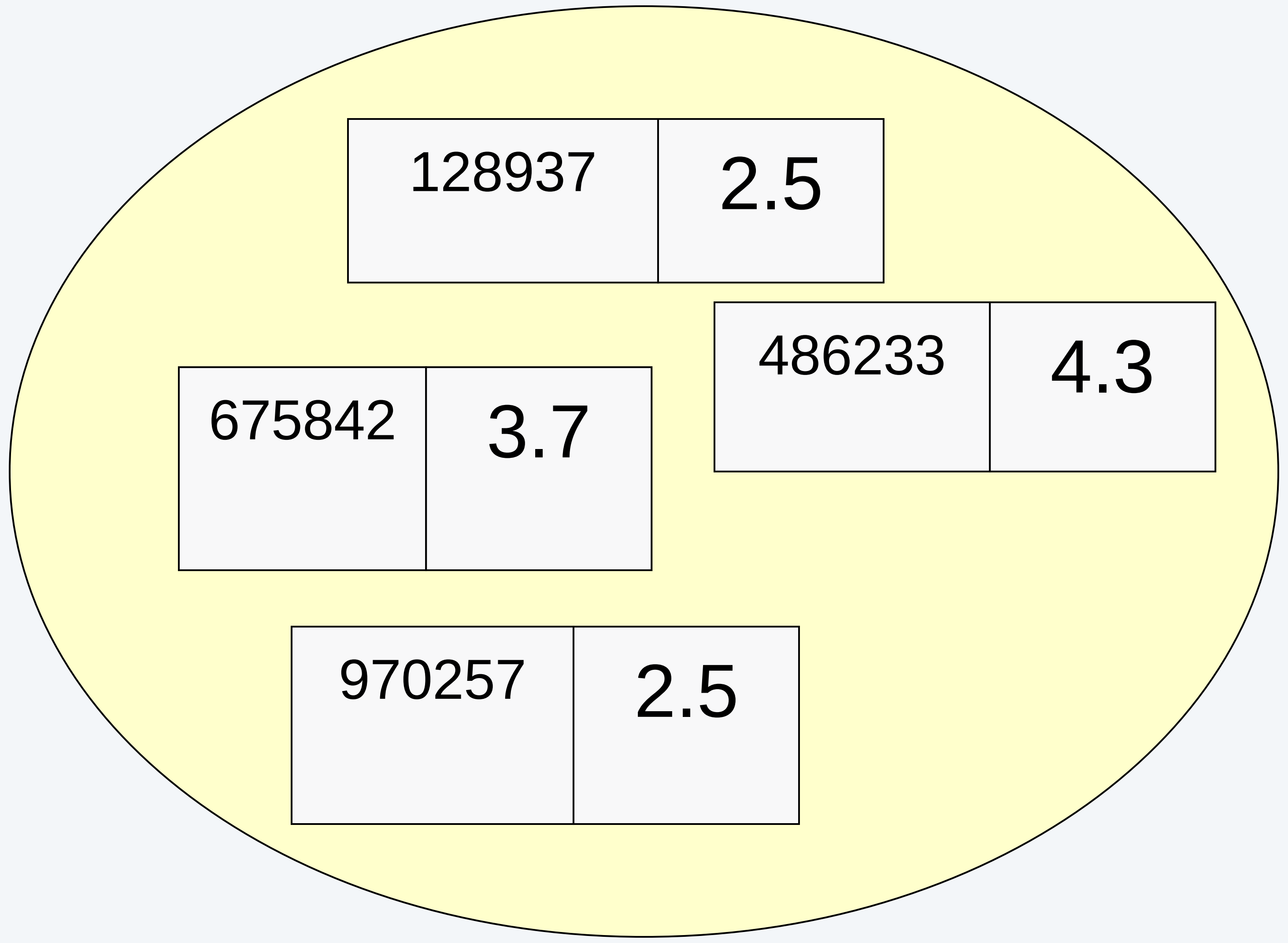
  template <class T>

- Use T or other identifier for generic type
- Precede the definition of each function with template <typename T>
- In the scope portion of the class member function, add <T>

```
#ifndef LIST_H
#define LIST_H

template <typename T>
struct Item {
  T val; Item<T>* next;
};

template <typename T>
class LList{
 public:
   LList();  // Constructor
   ~LList();  // Destructor
   void push_back(T newval);
   T& at(int loc);
 private:
   Item<T>* head_;
};

template<typename T>
LList<T>::LList()
{ head_ = NULL;
}

template<typename T>
LList<T>::~LList()
{ }

template<typename T>
void LList<T>::push_back(T newval)
{  ... }

#endif
```

Pair Template Example

This is similar to the C++ Pair Struct from the utility library that STL Map uses.




```
#include <iostream>
#include <string>
using namespace std;

template <typename T1, typename T2>
struct Pair {
   T1  first;
   T2  second;
  Pair(  T1 f, T2 s ) :  first(f), second(s)
   { }
};


int main()
{

   Pair<char, double> p1('a', 3.1);

   Pair<string, int> p2(string("hi"), 4);

   cout << p1.first << "," << p1.second << endl;
   cout << p2.first << "," << p2.second << endl;

   return 0;
}
```

a,3.1
hi,4

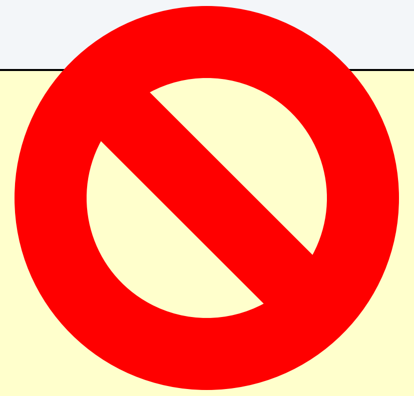
Template Class Declaration

Key Fact: Templated classes must have the implementation **IN THE HEADER FILE!**

Corollary: A templatized cannot be compiled separately in a .cpp file.

```
#ifndef LIST_H
#define LIST_H

template <typename T>
struct Item {
  T val; Item<T>* next;
};

template <typename T>
class LList{
 public:
   LList();  // Constructor
   ~LList();  // Destructor
   void push_back(T newval);
private:
   Item<T>* head_;
};
#endif
```

List.h



```
#include "List.h"

template<typename T>
LList<T>::push_back(T newval)
{
  if(head_ = NULL){
    head_ = new Item<T>;
    // how much memory does an Item
    //  require?
  }
}
```

List.cpp

Using Templatized Classes

The compiler generates code for the type of data when the objects are instantiated with certain types

Main.cpp

```
#include "List.h"

int main()
{
  LList<int> my_int_list;
  LList<double> my_dbl_list;

  my_int_list.push_back(5);
  my_dbl_list.push_back(5.5125);

  double x = my_dbl_list.pop_front();
  int y = my_int_list.pop_front();
  return 0;
}
```

```
#ifndef LIST_H
#define LIST_H

template <typename T>
struct Item {
  T val; Item<T>* next;
};

template <typename T>
class LList{
 public:
   LList();  // Constructor
   ~LList();  // Destructor
   void push_back(T newval);
   T& at(int loc);
 private:
   Item<T>* head_;
};

template<typename T>
LList<T>::LList()
{ head_ = NULL;
}

template<typename T>
LList<T>::~LList()
{ }

template<typename T>
void LList<T>::push_back(T newval)
{  ... }

#endif
```

List.h

- When accessing members of a templated base class provide the full scope or precede the member with this->

```
#include "llist.h"
template <typename T>
class Stack : private LList<T>{
 public:
   Stack();  // Constructor
   void push(const T& newval);
   T const & top() const;
};

template<typename T>
Stack<T>::Stack() : LList<T>()
{ }

template<typename T>
void Stack<T>::push(const T& newval)
{  // call inherited push_front()
   push_front(newval); // may not compile
   LList<T>::push_front(newval); // works
   this->push_front(newval);     // works
}

template<typename T>
void Stack<T>::push(const T& newval)
{ // assume head is a protected member
  if(LList<T>::head)          // works
     return LList<T>::head->val;
  if(this->head)              // works
     return this->head->val;

}
```

typename & Nested members

Precede the nested type with the keyword 'typename' when

- Not in the scope of the templated class AND
- The template type is still generic

```
#include <iostream>
#include <vector>
using namespace std;

template <typename T>
class Stack {
public:
  void push(const T& newval)
    { data.push_back(newval); }
  T& top();
private:
  std::vector<T> data;
};

template <typename T>
T& Stack<T>::top()
{
  //vector<T>::iterator it = data.end();
  typename vector<T>::iterator it = data.end(); //good
  return *(it-1);
}

int main()
{
  Stack<int> s1;
  vector<int>::iterator it;
  s1.push(1); s1.push(2); s1.push(3);
  cout << s1.top() << endl;
  return 0;
}
```

When the template type is still generic and you scope a nested type, precede with typename

When the template type is specific there is no need to use typename

A Basic Templatized Link List Example

```
template <typename T>
   struct Item {
   T val;
   unique_ptr<Item<T>> next;
};

template <typename T>
   class LListBasic {
   public:
   LListBasic() {  size_ = 0; }
   ~LListBasic();
   bool empty() const { return size_ == 0; }
   int size() const   { return size_; }
   void prepend(const T& val);
   T& get(int loc);

   private:
   unique_ptr<Item<T>> head_;
   int size_;

};
```

A Basic Templatized Link List Example

```
template <typename T>
  LListBasic<T>::~LListBasic()
  {
    while(head_){
      head_ = move(head_->next);
    }
  }

template <typename T>
  void LListBasic<T>::prepend(const T& val)
  {
    unique_ptr<Item<T>> old_head(move(head_));
    head_ = make_unique<Item<T>>();
    head_->val = val;
    head_->next = move(old_head);
    size_++;
  }
```

A Basic Templatized Link List Example

```
template <typename T>
   T& LListBasic<T>::get(int loc)
   {
 // How can this be fixed for appropriate error checking? What if the head is nullptr?
      Item<T>* temp = head_.get();
      while(temp  && loc != 0){
         temp = temp->next.get();
         loc--;
      }
      return temp->val;
   }

int main()
{
   LListBasic<int> LL;
   for(int i=9; i >= 0; i--){
      LL.prepend(i);
   }
   cout << "Size is " << LL.size() << endl;
   for(int i=0; i <= 9; i++){
      cout << LL.get(i) << endl;
   }
   return 0;
}
```