CS 247: Software Engineering Principles

C++ Templates

Reading: Eckel, Vol. 2
Ch. 5 Templates in Depth

C++ Function Templates

Suppose we want to create our own generic classes and functions?

A function template describes a family of functions:

```
template <typename T>
int compare(const T &v1, const T &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

Client Code:

```
compare (1, 3);         // compare<int>
compare (3.14, 2.7);    // compare<double>
```

Template Instantiation

```
// compiler generates and compiles type-specific versions

int compare(const int &v1, const int &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}

int compare(const string &v1, const string &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

Explicit Arguments

can explicitly state template parameter argument types
(important for specifying return types)

```
template <typename T1, typename T2, typename T3>

T1 sum ( const &T2 a, const T3 &b) {
   return a + b;
}

//client code
float f = sum<float>(10, 3.14); // OK:  sum<float,int,float>
```

Another Example

```
template <class InputIter, class OutputIter, class Predicate>

OutputIter copy_if (InputIter first, InputIter last, OutputIter result,
   Predicate pred)

{
    for ( ; first != last; ++first)
        if ( pred(*first) ) *result++ = *first;
    return result;
}
```

C++ Class Templates

Define a generic (parameterized) classes

- e.g., a container whose element type is specified by a parameter

```
template <typename T>    // T is element type
class Stack {
public:
   Stack();
   void push( const T& );
   T  top() { return items_[ top_ ]; }
   T  pop();
private:
   T  items_ [STACK_SIZE];
   int top_;
};

template <typename T>
void Stack<T>::push( const T &elem )  {
    top_ += 1;
    items_ [top_] = elem;
}
...
```

Non-Type Template Parameters

Can have non-type template parameters, which are treated as compile-time constants.

- can provide a default value

```
template <typename T, int size = 100>
class Stack {
public:
    Stack();
    void push( const T& );
    T  top();
    T  pop();
private:
    T  items_ [ size ];
    int top_;
};
```

Client code provides a compile-time value for size:

```
Stack<int,99> mystack1;   // stack of size 99
Stack<int>    mystack2;   // stack of size 100
```

Friends

There are three kinds of friend declarations that may appear in a class template. Each kind of declaration declares friendship to one or more entities:

1. A friend declaration for an ordinary nontemplate class or function, which grants friendship to the specific named class or function.

2. A friend declaration for a class template or function template, which grants access to all instances of the friend.

3. A friend declaration that grants access only to a specific instance of a class or function template.

A Template's Implicit Interface

How the template definition uses variables of type T will impose some requirements on allowable instantiations

```
template <typename T>
T mumble (T val) {
    T newVal = val;
    T *p = 0;
    val.speak();
    cout <<  "val = "  <<  val  <<  endl;
    if ( val < newVal)
       return "success";
}
```

Design Considerations

1. How should parameters of template type be passed?
   - Pass-by-value (appropriate for built-in types)?
   - Pass-by-reference (appropriate for class types)?
2. Consider the initialization of objects with parameterized members

---

```
template <typename T, typename U>
struct pair {
   T first;
   U second;

   pair() : first (T()), second (U()) {}

   pair( const T &t, const U &u)
      : first (t), second (u) {}


   template <typename V, typename W>
    pair (const pair<V,W> &p)
      : first(p.first), second(p.second) { }
};
```

Template Compilation

1. The template definition is compiled first.
   - If the code might be legal for some type T, then the definition is considered to be legal.

```
// file max.h
template <typename T>
T max(T a, T b) {
        return a > b ? a : b;
}
```

2. When the template is instantiated, and the instantiated class/function is type checked again.

```
// Client code
#include "max.h"
#include "MyObject.h"
...
 MyObject a(...), b(...);
 MyObject c = max<MyObject>(a,b);
```

Template Compilation

3. The compiler must *see* the template *definition* in order to instantiate and compile the template.
   a) include template definitions in header file
      can #include implementation to maintain separation of header and implementation code.

```
// header file utlities.h
    #ifndef UTLITIES_H // header gaurd (Section 2.9.2, p. 69)
    #define UTLITIES_H
    template <class T> int compare(const T&, const T&);
    // other declarations

    #include "utilities.cc" // get the definitions for compare etc.
    #endif

    // implemenatation file utlities.cc
    template <class T> int compare(const T &v1, const T &v2)
    {
        if (v1 < v2) return -1;
        if (v2 < v1) return 1;
        return 0;
    }
    // other definitions
```