CS 246: Software Abstraction and Specification

Lecture 19

C++ Templates

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

Genericity vs. Type-Safety

Genericity: we want to make our code as general and reusable as possible.

- often means choosing a "most general" parameter or element type
- want to have a stack of `strings`, `ints`, `Figures`

Type-safety: usually requires that our code be specific, so that the compiler can check for mismatches and inconsistencies.

- often means limiting or constraining parameter/element type
- shouldn't be allowed to push a `string` onto a stack of `Figures`

Problem: How to achieve both (e.g., in a generic stack)?

Approach #1: Type-Specific Implementations

```
class StringStack {
public:
    StringStac
    void push(
    string top
    string pop
private:
    string ite
    int top_;
};

StringStack::
    top_ = -1;
}
...
```

```
class IntStack
public:
    IntStack();
    void push(in
    int top();
    int pop();
private:
    int items_ [
    int top_;
};

IntStack::IntSt
    top_ = -1;
}
...
```

```
class FigureStack {
public:
    FigureStack();
    void push(Figure*);
    Figure* top();
    Figure* pop();
private:
    Figure* items_ [STACK_SIZE];
    int top_;
};

FigureStack::FigureStack() {
    top_ = -1;
}
...
```

Approach #2: Inheritance (won't work in C++)

Can create a general-purpose stack whose element type is a very general 'class' (`java.lang.object`).

The actual element type inherits from the general element type
- can set the actual element type in the stack constructor

- It works! (in Java)
- There is not a lot of type safety
- Even if you push (a reference to) a Figure, you get back (a reference to) an Object

Approach #3: Generics

Define a generic (parameterized) Stack class, whose element type is specified by a parameter when the stack is constructed.

```
// Client code

Stack<string> ss;

Stack<int> is;

Stack<Figure*> fs;
```

C++ Templates

Define a generic (parameterized) Stack class, whose element type is specified by a parameter when the stack is constructed.

```
template <typename T>    // T is element type
class Stack {
public:
   Stack();
   void push( const T& );
   T  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
```

Function Templates

A function template describes a family of functions:

```
template <typename T>
void swap (T &var1, T &var2) {
      T  temp = var1;
      var1 = var2;
      var2 = temp;
}
```

Client Code:

```
int i=1, j=2;
swap<int> (i, j);

string s="abc"
string t="xyz";
swap (s, t);  // compiler can infer template argument type

double x;
swap (i, x);  // compiler error
```

Another Example

Convert the following into a template function:

```
int* find (int array[ ], int size, int val) {
      if ( ! array || size < 1 )
            return NULL;

      for ( int ix = 0; ix < size; ix+=1) {
            if ( array[ ix ] == val ) {
                  return &array[ ix ];
            }
       }

       return NULL;
}
```

Another Example: Helper Operators

```
template <typename T, typename U>
inline bool  operator != (const T &t, const U &u)
{
   return !( t==u );
}
```

```
template <typename T, typename U>
inline bool  operator > (const T &t, const U &u)
{
   return  u < t ;
}
```

```
template <typename T, typename U>
inline bool  operator <= (const T &t, const U &u)
{
   return  ! (u < t) ;
}
```

```
template <typename T, typename U>
inline bool  operator >= (const T &t, const U &u)
{
   return  ! (t < u) ;
}
```

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);
```

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) { }
};
```

Some Design Considerations

Recall: The template definition is compiled first

- definition is legal as long it might be legal for some type T

Most compilers do not support separate compilation of templates.

- Compiler needs to see template definition (full implementation) in order to type check the individual template instantiations.
- Common Practice: include entire template definition in header file

CS247 : Using Templates

You will NOT be asked to write your own class or function templates.

But you should be able to understand basic use of templated class / function definitions.

`dynamic_cast<>` is implemented using templates

```
// simplified signature
template <typename T>
T dynamic_cast <T> (expression)
```

auto_ptr<>

auto_ptr< > is a semi-smart pointer class that owns the object it references and is responsible for deleting the object when the pointer is deleted

```
template <typename T>
class auto_ptr {
private:
   T *object_;
public:
   explicit auto_ptr (T* ptr = 0);
   auto_ptr (auto_ptr& rhs);
   ~auto_ptr() { delete object_; }  // destructor deletes object
   auto_ptr& operator= (auto_ptr& rhs);
   T& operator*() const;
   T* operator->() const;
   T* release();
   void reset (T* ptr=0);
};
```

Useful for Avoiding Memory Leaks

```
#include <memory>
using namespace std;

if (someCondition) {
   auto_ptr<Circle> rc = new Circle ("Red", 10, 10);
   rc->setSize(25);
   // do Stuff

   // no explicit delete statement
}
```

Useful for Transferring Control of Referent

```
auto_ptr<T>  Source() {
   return auto_ptr<T> ( new T );
}

void Sink ( auto_ptr<T> pt )  { ... }

auto_ptr<T> pt = Source();
```

1) `Source()` allocates a new object and returns it (and responsibility of ownership) to caller.

2) `Sink()` takes an `auto_ptr` and assumes ownership of referent.

Standard Template Library

The STL is a collection of data structures and supporting algorithms implemented using templates.

- Containers – parameterized data structures that manage collections of objects of a user-specified type

  e.g. vector<T>, set<T>, map<T>

- Iterators – used to iterate through the members of some container (or subset thereof)

- Algorithms – used to process the elements in a container

  e.g., find(), sort(), filter(), ...

Standard Template Library

The goal of the STL is to provide a set of data structures and algorithms that are:

- generic – parameterized by type
- strongly typed – e.g., vector<Figure *>
- flexible – large APIs, many possible modes of use
- extensible – inherit/extend for your own needs OR create a specialized, restricted API via *adapters*
- efficient – static method dispatch, specialized algorithms
- (relatively) easy to use

Philosophy of the STL

1) To provide a collection of useful, efficient, typesafe and generic (i.e., type-parameterized) containers.
   - Each container has (almost) no innate understanding of its elements' types.
   - Each container should define its own iterators.

2) To provide a collection of useful, efficient, generic algorithms that operate on iterators.
   - Each algorithm knows (almost) nothing about the structure it's operating on (just knows that it can be traversed by an iterator)
   - Each algorithm knows nothing about the elements in the structure.

3) To define container methods only when the generic algorithms are unsuitable or much less efficient.

Containers Define Their Own Iterator

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

    class iterator; // forward declaration
    friend class iterator;  // make iterator a friend
    class iterator {
        Stack &s_;
        int idx_;
    public:
        iterator(Stack &st): s_(st), idx_(0) {} // empty stack
        iterator(Stack &st, bool) : s_(st), idx_(st.top) {} // end sentine
        T operator*() const;
        T operator++();
        bool operator==(const iterator& rv) const;
        bool operator!=(const iterator& rv) const;
    };
   iterator begin(); { return iterator(*this); }
   iterator end(); { return iterator(*this, true);}
};
```

Containers Define Their Own Iterators

```
int main() {
  Stack<int> is;
  for(int i = 0; i < 20; i++)
    is.push(i);

  // Traverse with an iterator:
  cout << "Traverse the Stack\n";
  Stack<int>::iterator it = is.begin();
  while(it != is.end())
    cout << *(it++) << endl;

  // Create stack of strings read from input file
  ifstream in("IterStackTemplateTest.cpp");
  string line;
  Stack<string> strings;
  while(getline(in, line))
    strings.push(line);

  // Traverse with an iterator:
  cout << "Traverse the Stack\n";
  Stack<string>::iterator sb = strings.begin();
  Stack<string>::iterator se = strings.end();
  while(sb != se)
    cout << *(sb++) << endl;
}
```