An introduction to C++ Templates

For : COP 3330.
Object oriented Programming (Using C++)
http://www.compgeom.com/~piyush/teach/3330

Piyush Kumar

Templates

- Are C macros on Steroids
- Give you the power to parametrize
- Compile time computation
- Performance

"The art of programming programs
that read, transform, or write other programs."
- François-René Rideau

Generic Programming

- How do we implement a linked list with a general type inside?
  - void pointers?
  - Using macros?
  - Using Inheritance?

Templates

- Function Templates
- Class Templates
- Template templates *
- Full Template specialization
- Partial template specialization

Metaprogramming

- Programs that manipulate other programs or themselves
- Can be executed at compile time or runtime.
- Template metaprograms are executed at compile time.

Good old C

- C code
  - double square(double x) { return x*x; }
    - square(3.14) ← Computed at compile time
  - #define square(x) ((x)*(x))
  - Static double sqrarg;

#**define SQR**(a) (sqrarg=(a), sqrarg*sqrarg)

Templates

- Help us write type-independent code
- Question: How do you swap two elements of any type? How do you return the square of any type?

Defining the template

min.hpp

```
template <typename T>
inline T const& min (T const& a, T const& b){
        return a < b ? a : b ;
}
```

- Template parameter T.
- Syntax: template < comma separated list of parameters >
- For historical reasons, you can use “class” instead of typename.

inline vs constexpr

- Placement: both follow template parameter list
  - Correct: *template <typename T> inline T foo(T, T);*
  - Wrong: *inline template <typename T> T foo(T, T);*
- inline: only copy/paste
- constexpr: compute during compilation
  - e.g. compute array size using a computation intensive function
  - Do it just once during compilation. Optimization.
- Why not hard code the value then?
  - Function parameters could change

Using the template

The process of replacing template parameters with concrete types is called ***instantiation.***

```
inline int const& min(int const& a, int const& b){
        return a < b ? a : b;
}
```

```
#include <iostream>
#include <string>
#include “min.hpp”

int main(){
        int i = 12;
        std::cout << “min(7,i): “ << ::min(7,i) << std::endl;
        std::string s1 = “math”;
        std::string s2 = “cs”;
        std::cout << “min(s1,s2): “ << ::min(s1,s2) << std::endl;
}
```

::min(7, i) => Get the name from the parent scope

A compile time error

```
// The following does not provide operator<
std::complex<float>  c1,c2;

…

min(c1,c2);  // error at compile time.
```

Attempt to instantiate a template for a type that doesn’t support all the operations used within it will result in a compile-time error.

Function Templates

```
template< typename T >
  inline T square ( T x ) { return x*x; }
```

- A specialization is instantiated if needed :
  - square<double>(3.14)
- Template arguments maybe deduced from the function arguments → square(3.14)
- MyType m; … ; square(m); → expands to square<MyType>(m)

Operator * must be overloaded for MyType

Function Templates

```
template<typename T>
void swap( T& a, T& b ){
        T tmp(a);   // cc required
        a = b;      // ao required
        b = tmp;
}

Mytype x = 1111;
Mytype y = 100101;
swap(x,y);   ← swap<Mytype>(...) is instantiated
```

Note reliance on T's *concepts* (properties):
In above, T must be *copyable* and *assignable*
Compile-time enforcement (*concept-checking*) techniques available

Argument deduction

```
template <typename T> // T is a template parameter
inline T const& min (T const& a, T const& b){ // a and b are call parameter.
    return a < b ? a : b ;
}

min(4,7); // ok : T is int for both arguments.
min(4,4.2); // error: First T is int, second T is double.
```

Solutions:
1. min (static_cast<double>(4), 4.2); // ok
2. min<double>(4,4.2); // ok

Template Parameters.

- You may have as many template parameters as you like.

```
template <typename T1, typename T2>
inline T1  min (T1 const& a, T2 const& b){
        return a < b ? a : b ;
}

min(4,4.2); // ok : but type of first argument defines return type.
// drawback : min (4,4.2) is different compared to min(4.2,4)
// return is created by an implicit typecast and can not be returned as
// a reference.
```

Template parameters.

```
template <typename T1, typename T2 , typename RT >
inline RT  min (T1 const& a, T2 const& b){
        return static_cast<RT>(a < b ? a : b);
}

min<int,double,double>(4,4.2); // ok: but long and tedious
```

```
template < typename RT, typename T1, typename T2 >
inline RT  min (T1 const& a, T2 const& b){
        return static_cast<RT>(a < b ? a : b);
}

min<double>(4,4.2); // ok: return type is double.
```

Function Template Specialization

- **template<>**
  void swap<myclass>(myclass& a, myclass& b){
  a = b = 0;
  }

Custom version of a template for a specific class

Class Templates

```
template<typename NumType, unsigned D>
class dpoint{
    public:
        NumType x[D];
    };
```

A simple 3-dimensional point.

```
dpoint<float,3> point_in_3d;
point_in_3d.x[0] = 5.0;
point_in_3d.x[1] = 1.0;
point_in_3d.x[2] = 2.0;
```

Note the explicit instantiation dpoint<float,3>. Was optional for function templates.

Class templates.

Implementing a member function.

```
template<typename NumType, unsigned D>
class dpoint {
public:
        inline virtual void normalize (void);
}

template<typename NumType, unsigned D>
void dpoint<NumType,D>::normalize (void){
        NumType len = sqrt(sqr_length());
        if (len > 0.00001)
            for(unsigned i = 0; i < D; ++i){ x[i] /= len; }
}
```

Friend templates

```
template<typename NumType, unsigned D>
class dpoint {
public:
        template<typename NT, unsigned __DIM>
        friend bool operator!= (const dpoint<NT,__DIM>& p,
                                const dpoint<NT,__DIM>& q);
}

// Function template
template<typename NT, unsigned __DIM>
bool operator!= (const dpoint<NT,__DIM>& p,
                 const dpoint<NT,__DIM>& q){
        //…
}
```

Member Templates

```
template <typename T>
class Stack {
  private:
    std::deque<T> elems;   // elements

  public:
    void push(const T&);   // store new top element
    void pop();            // remove top element
    T top() const;         // return top element
    bool empty() const {   // return whether the stack is empty
        return elems.empty();
    }

    // assign stack of elements of type T2
    template <typename T2>
    Stack<T>& operator= (Stack<T2> const&);
};
```

Member Templates

```
template <typename T>
template <typename T2>
Stack<T>& Stack<T>::operator= (Stack<T2> const& op2)
{
    if ((void*)this == (void*)&op2) {    // assignment to itself?
        return *this;
    }

    Stack<T2> tmp(op2);              // create a copy of the assigned stack

    elems.clear();                   // remove existing elements
    while (!tmp.empty()) {           // copy all elements
        elems.push_front(tmp.top());
        tmp.pop();
    }
    return *this;
}
```

Parameterized container

```
template <typename T, typename CONT = std::deque<T> >
class Stack {
  private:
    CONT elems;           // elements

  public:
    void push(T const&);   // push element
    void pop();            // pop element
    T top() const;         // return top element
    bool empty() const {   // return whether the stack is empty
        return elems.empty();
    }

    // assign stack of elements of type T2
    template <typename T2, typename CONT2>
    Stack<T,CONT>& operator= (Stack<T2,CONT2> const&);
};
```

Member template again.

```
template <typename T, typename CONT>       // for enclosing class template
template <typename T2, typename CONT2>   // for member template
Stack<T,CONT>&
Stack<T,CONT>::operator= (Stack<T2,CONT2> const& op2)
{
    if ((void*)this == (void*)&op2) {    // assignment to itself?
        return *this;
    }

    Stack<T2,CONT2> tmp(op2);       // create a copy of the assigned stack

    elems.clear();                  // remove existing elements
    while (!tmp.empty()) {          // copy all elements
        elems.push_front(tmp.top());
        tmp.pop();
    }
    return *this;
}
```

Spot the problem

```
#include <string>

// note: reference parameters
template <typename T>
inline const T & max (const T& a, const T& b)
{
   return  a < b  ?  b : a;
}

int main()
{
   std::string s;

   ::max("apple","peach");   // OK
   ::max("apple","tomato");  // ERROR
   ::max("apple",s);         // ERROR
}
```

Class Templates: Unlike function templates

- Class template arguments can't be deduced in the same way, so usually written explicitly:
  dpoint <double, 2> c; // 2d point
- Class template parameters may have *default values*:
  template< typename T, typename U = int >
  class MyCls { … };
- Class templates may be *partially specialized*:
  template< typename U >
  class MyCls< bool, U > { … };

Using Specializations

- First declare (and/or define) the general case:
  template< typename T >
  class C { /* handle most types this way */ };
- Then provide either or both kinds of special cases as desired:
  template< typename T >
  class C< T * > { /* handle pointers specially */ };
  template<> // note: fully specialized
  class C< int * > { /* treat int pointers thusly */ };
- Compiler will select the most specialized applicable class template

Template Template Parameters

- A class template can be a template argument
- Example:

```
template<
   template<typename ELEM> class Bag
>
class C { //…
   Bag< float >  b;
};
```

- Or even:

```
template< class E,
   template<typename ELEM> class Bag
>
class C { //…
   Bag< E >  b;
};
```

More Templates

```
template<unsigned u>
struct MyClass {
   enum { X = u };
};

Cout << MyClass<2>::X << endl;
```

Template Metaprograms

- Factorials at compile time

```
template<int N> class Factorial {
      public:
         enum { value = N * Factorial<N-1>::value };
};

template<>
class Factorial<1> {
      public:
         enum { value = 1 };
};
```

```
int w = Factorial<10>::value;
```

Template Metaprograms

- Metaprogramming using C++ can be used to implement a turning machine. (since it can be used to do conditional and loop constructs).

Template Metaprograms

```
template< typename NumType, unsigned D, unsigned I > struct origin
{
  static inline void eval( dpoint<NumType,D>& p )
  {
      p[I] = 0.0;
    origin< NumType, D, I-1 >::eval( p );
  }
};

// Partial Template Specialization
template <typename NumType, unsigned D> struct origin<NumType, D, 0>
{
  static inline void eval( dpoint<NumType,D>& p )
  {
      p[0] = 0.0;
  }
};
```

```
const int D = 3;
inline void move2origin() { origin<NumType, D, D-1>::eval(*this); };
```

Food for thought

- You can implement
  - IF
  - WHILE
  - FOR

Using metaprogramming. And then use them in your code that needs to run at compile time ☺

More challenges: Implement computation of determinant / orientation / volume using metaprogramming.

Template Metaprogramming Examples.

```
/*
Fibonacci's Function as
C++ Template Metaprogram
*/
template< unsigned long N > struct Fib
 {
 enum { value = Fib<N-1>::value +  Fib<N-2>::value } ;
 } ;

struct Fib< 1 >
 {
 enum { value = 1 } ;
 } ;

struct Fib< 2 >
 {
 enum { value = 1 } ;
 } ;
```

Template Metaprogramming examples.

```
/*
Binary-to-Decimal as
Template Metaprogram
*/

template< unsigned long N > struct Bin
 {
 enum { value = (N % 10) +
        2 * Bin< N / 10 > :: value } ;
 } ;

struct Bin< 0 >
 {
 enum { value = 0 } ;
 } ;
```

```
usigned long const one
  = binary<1001>::value;
```

More metaprogramming

- Learn and use the Boost MTL Library.

Traits Technique

- Operate on “types” instead of data
- How do you implement a “mean” class without specifying the types.
  - For double arrays it should output double
  - For integers it should return a float
  - For complex numbers it should return a complex number

Traits

```
Template< typename T >
struct average_traits{
   using T_average = T;
};

Template<>
struct average_traits<int>{
    using T = float;
}
```

Traits

average_type(T) = T
average_type(int) = float

Traits

```
template<typename T>
typename average_traits<T>::T_average
average(T* array, int N){
    typename avearage_traits<T>::T_average
      result = sum(array,N);
  return result/N;
}
```

Sources

- C++ Templates: The complete guide by Vandevoorde and Josuttis.
- Template Metaprogramming by Todd **Veldhuizen**
- C++ Meta<Programming> Concepts and results by Walter E. Brown
- C++ For Game programmers by Noel Llopis
- C++ Primer by Lippman and Lajoie