Object-Oriented Programming for Scientific Computing

Templates and Static Polymorphism

Ole Klein

Interdisciplinary Center for Scientific Computing
Heidelberg University
ole.klein@iwr.uni-heidelberg.de

26. Mai 2015

Generic Programming

- Often the same algorithms are required for different types of data.
- Without generic programming the same function needs to be rewritten for each data type. This is tedious and error-prone.
  Example:

```
int Square(int x)
{
    return(x*x);
}
long Square(long x)
{
    return(x*x);
}
```

```
float Square(float x);
{
    return(x*x);
}
double Square(double x);
{
    return(x*x);
}
```

- Generic programming makes it possible to write an algorithm once and parameterize it with the data type.
- The language device used is called `template` in C++ and can be used for both functions and classes.

Function Templates

- A function template starts with the keyword `template` and a list of template arguments, separated by commas and enclosed by angle brackets:

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

- `typename` designates a type and was introduced because C++ has built-in types that aren't classes (e.g. `int`). For historical reasons `class` and `typename` may be used interchangeably in the template argument list.

Template Instantiation

- At the first use of the function with a specific combination of data types the compiler automatically generates the code for these types. This is referred to as template instantiation.
- An explicit instantiation isn't necessary.
- The template parameters are determined from the types of function arguments.
- No automatic type conversion is allowed in this case (unlike normal function calls).
- As with function overloading, the type of the return value doesn't matter.
- Ambiguities can be avoided by:
  - Explicit type conversion of arguments
  - Explicit specification of template arguments in angle brackets:

```
std::cout << Square<int>(4) << std::endl;
```

- The argument types must match the declaration and the types have to provide all the necessary operations (e.g. the `operator*()`).

Example: Unary Function Template

```
#include<cmath>
#include<iostream>

template<typename T>
T Square(T x)
{
    return(x*x);
}

int main()
{
    std::cout << Square<int>(4) << std::endl;
    std::cout << Square<double>(M_PI) << std::endl;
    std::cout << Square(3.14) << std::endl;
}
```

Example: Binary Function Template

```
#include<cmath>
#include<iostream>

template<class U>
const U& Max(const U& a, const U& b)
{
    if (a > b)
        return a;
    else
        return b;
}

int main()
{
    std::cout << Max(1,4) << std::endl;
    std::cout << Max(3.14,7.) << std::endl;
    std::cout << Max(6.1,4) << std::endl; // compiler error
    std::cout << Max<double>(6.1,4) << std::endl; // unambiguous
    std::cout << Max(6.1,double(4)) << std::endl; // unambiguous
    std::cout << Max<int>(6.1,4) << std::endl; // compiler warning
}
```

Translation of Templates

- If templates aren't used and therefore not instantiated, then the template code will only be checked for gross syntax errors (e.g. missing semicolons).
- Checks wether all the function calls are valid are only made when the template is instantiated. Only then unsupported function calls are discovered, for example. This means the error messages can be quite strange.
- Since the code is generated when it is used, the compiler needs to see the complete function definition at that time, not only the declaration as for normal functions.
- Thus, the usual distinction between header file and source file is not possible for templates. The whole definition should be in the header file.

Function Overloading

- Function templates can be overloaded just like normal functions.
- There may also be non-template functions and template functions of the same name.
- If there is a suitable non-template function (without conversion of types), then it will be used.
- If a template function can be created which fits better (no conversion of types), this one is used.
- The use of a template function can be forced by adding empty angle brackets.

Example: Determination of the Maximum

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

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

template<typename T>
inline const T& max(const T& a, const T& b, const T& c){
  return ::max(a, ::max(b, c));
}

int main(int argc, char** argv) {
  ::max(7, 42, 68);    // calls the template for three arguments,
      which calls the nontemplate for two ints twice
  ::max(7.0, 42.0);    // calls max<double> (argument deduction)
  ::max('a', 'b');     // calls max<char> (argument deduction)
  ::max(7,42);         // calls nontemplate for two ints
  ::max<>(7,42);       // calls max<int> (argument deduction)
  ::max<double>(7,42);// calls max<double> (no argument deduction)
  ::max('a', 42.7);    // calls nontemplate for two ints
}
```

Example: Determination of the Maximum

Function overloading can e.g. be used if there are types for which a comparison deviates from the standard:

```
#ifndef MAX_POINTER_REFERENCE_HH
#define MAX_POINTER_REFERENCE_HH
#include<cstring>
#include"max.hh"

// pointer comparison based on content
template<typename T>
inline const T*& max(const T*& a, const T*& b)
{
  return *a < *b ? b : a;
}

// comparison of C strings
inline const char*& max(const char*& a, const char*& b)
{
  return std::strcmp(a,b) < 0 ? b : a;
}

#endif
```

Example: Determination of the Maximum

```
#include<string>
#include<iostream>
#include"max_pointer_reference.hh"

int main(int argc, char** argv)
{
  int a = 47, b = 9;
  std::cout << ::max(a,b) << "\n"; // max for two ints

  std::string s = "hey", t = "there";
  std::cout << ::max(s,t) << "\n"; // template max for two strings

  int* p1 = &a, int* p2 = &b;
  std::cout << *::max(p1,p2) << "\n"; // template max for two
      pointers

  const char* s1 = "large", const char* s2 = "small";
  std::cout << ::max(s1,s2) << "\n"; // max for two C strings
}
```

Specialization of Function Templates

It is possible to define specific template functions for certain parameter values. This is called template specialization. It can, for example, be used for speed optimizations:

```
template <size_t N>
double scalarProduct(const double* a, const double* b)
{
    double result = 0;
    for (size_t i=0;i<N;++i)
        result += a[i]*b[i];
    return result;
};

template<>
double scalarProduct<2>(const double* a, const double* b)
{
    return a[0]*b[0]+a[1]*b[1];
};
```

Useful Function Templates

The C++ standard library already provides some useful function templates:

- `const T& std::min(const T&, const T&)`
  Minimum of a and b
  ```
  int c = std::min(a,b);
  ```
- `const T& std::max(const T&, const T&)`
  Maximum of a and b (in the example above, we always used `::max` in order to prevent the use of this template)
  ```
  int c = std::max(a,b);
  ```
- `void std::swap(T&, T&)`
  swaps the content of a and b
  ```
  std::swap(a,b);
  ```

Class Templates

- It is often useful to also parameterize classes.
- Class templates are defined exactly as function templates, e.g.

```
template<typename T1, typename T2>
class Product
{
    T1 var1;
    T2 var2;
  public:
    T2 multiply(T1 a, T2 b);
};
```

- If the template arguments are types (e.g. `typename T1`), then these can be used within the class to define attributes, function arguments and return values.

Class Templates

In particular, container classes can be used to store items of very different types. Here a stack as an example:

```
template<typename T>
class Stack
{
  private:
    std::vector<T> elems;

  public:
    void push(const T&);
    void pop();
    T top() const;
    bool empty() const
    {
      return elems.empty();
    }
};
```

Class Templates

- It is important to distinguish between the type of the class and its name:
  - The type of the class is `Stack<T>`. This is needed when objects of that class are used as function arguments or return values (e.g. the copy constructor).
  - The name of the class and thus the name of the constructors and the destructor is `Stack`.
  - The class name may be used as a shorthand for the full type (with the same template arguments) within the templatized class.

Implementation of Methods outside of the Class

- The methods of a class template can be defined normally as inline functions.
- If a method is defined outside of the class, then the compiler must be informed that the method belongs to a class template.
- For this the keyword `template`, followed by the template argument list of the class, is placed in front of the method definition.
- The template arguments are listed in angle brackets after the class name (the namespace of the class consists of both its name and the template arguments).

```
template<typename T>
void Stack<T>::push(const T& elem){
  elems.push_back(elem);
}

template<typename T>
void Stack<T>::pop(){
  if(elems.empty())
    throw std::out_of_range
          ("Stack<>::pop(): empty
          stack");
  elems.pop_back();
}
```

```
template<typename T>
T Stack<T>::top() const{
  if(elems.empty())
    throw std::out_of_range
          ("Stack<>::top(): empty
          stack");
  return elems.back();
}
```

Use of Class Templates

- To define an object of a class template, the name of the class must be followed by a list of appropriate arguments in angle brackets (e.g. `Stack<int>`).
- To save memory and compilation time, code is generated only for the methods actually called.
- This means class templates can even be instantiated for types that do not provide all the necessary operations, as long as the methods in which these are needed are never called.
- Instantiated class templates can be used just as normal types, e.g. declared as `const` or `volatile` or used in arrays. Pointers and references can also be defined of course.

Use of Class Templates

- If templates have long argument lists, then the name of the instantiated class is very long. Here `typedef`s are very helpful:

```
typedef Stack<int> IntStack;
void foo(const IntStack& s)
{
  IntStack is;
  ...
}
```

- Of course instantiated templates can themselves also serve as template arguments.

```
Stack<Stack<int> > iss; // note the space between closing
    brackets
```

Use of Class Templates

```
#include<iostream>
#include<string>
#include<cstdlib>
#include"stack.hh"

int main(int argc, char** argv){
  try{
    Stack<int> intStack;
    Stack<std::string> stringStack;

    intStack.push(7);
    std::cout << intStack.top() << std::endl;

    stringStack.push("hello");
    std::cout << stringStack.top() << std::endl;
    stringStack.pop();
    stringStack.pop();
  }catch(const std::exception& e){
    std::cerr << "Exception␣" << e.what() << std::endl;
    return 1;
  }
}
```

Useful Class Template: Pair

A useful class template is `pair`:

```
std::pair<int,double> a;
a.first = 2;
a.second = 5.;
std::cout << a.first << "␣" << a.second << std::endl;
```

`pair` e.g. makes functions with two return values possible.

Spezialisation of Class Templates

- Class templates can be specialized for certain argument values as well, either because for this combination special behavior is needed or for optimization purposes.
- This is somewhat similar to function overloading.
- All methods have to be specialized:

```
template<>
class Stack<std::string>
{
  private:
    std::vector<std::string> elems;

  public:
    void push(const std::string&);
    void pop();
    std::string top() const;
    bool empty() const
    {
      return elems.empty();
    }
};
```

Specialization of Class Templates: Method Definition

```
void Stack<std::string>::push(const std::string& elem){
  elems.push_back(elem);
}

void Stack<std::string>::pop(){
  if(elems.empty())
    throw std::out_of_range("Stack<>::pop(): empty stack");
  elems.pop_back();
}

std::string Stack<std::string>::top() const{
  if(elems.empty())
    throw std::out_of_range("Stack<>::top(): empty stack");
  return elems.back();
}
```

Partial Specialization

A class as the following

```
template<typename T1, typename T2>
class MyClass { ... };
```

allows several partial specializations:

```
// both template parameters are equal
template<typename T>
class MyClass<T,T>{ ... };

// second parameter has a specific type, e.g. an int
template<typename T>
class MyClass<T,int>{ ... };

// partial specialisation for pointers
template<typename T1, typename T2>
class MyClass<T1*,T2*>{ ... };
```

Partial Specialisation

This leads to the following assignments:

```
MyClass<int, float> mif;  // use MyClass<T1,T2>
MyClass<float,float> mff; // use MyClass<T,T>
MyClass<float,int> mfi;   // use MyClass<T,int>
MyClass<int*,float*> mpi; // use MyClass<T1*,T2*>
```

But it can also lead to ambiguities:

```
MyClass<int,int> mii; // matches MyClass<T,T> and MyClass<T,int>
MyClass<int*,int*> m; // matches MyClass<T,T> and MyClass<T1*,T2*>
```

In these cases, the result is a compiler error (that is difficult to resolve).

Template Default Arguments

- Default values can also be defined for the arguments of class templates.
- These may also depend on the previous template arguments.
- As with function arguments, only the last arguments can have default values.
- Example: Define a stack with an additional selectable container:

```
template<typename T, typename C = std::vector<T> >
class Stack
{
  public:
    typedef C Container;
  private:
    Container elems;

  public:
    void push(const T&);
    void pop();
    T top() const;
    bool empty() const
    {
      return elems.empty();
    }
};
```

Example for Default Template Arguments

- Specifying a default argument does not remove the obligation to specify any template arguments for function definitions outside the class.

```
template<typename T, typename C>
void Stack<T,C>::push(const T& elem){
  elems.push_back(elem);
}

template<typename T, typename C>
void Stack<T,C>::pop(){
  if(elems.empty())
    throw std::out_of_range("Stack<>::pop(): empty stack");
  elems.pop_back();
}

template<typename T, typename C>
T Stack<T,C>::top() const{
  if(elems.empty())
    throw std::out_of_range("Stack<>::top(): empty stack");
  return elems.back();
}
```

Example for Default Template Arguments

- The stack can be used the same way as before.
- If the second template parameter is omitted, then as before a `std::vector` will be used to store the elements.
- In addition, a different type of container can be used, e.g. a `std::deque`.

```
int main(int argc, char** argv){
  try{
    Stack<int> intStack;
    Stack<std::string,std::deque<std::string> > stringStack;

    intStack.push(7);
    std::cout << intStack.top() << std::endl;

    stringStack.push("hello");
    std::cout << stringStack.top() << std::endl;
    stringStack.pop();
    stringStack.pop();
  }catch(const std::exception& e){
    std::cerr << "Exception␣"<<e.what() << std::endl;
    return 1;
  }
}
```

Template Parameters that aren't Types

- Template parameters aren't necessarily types.
- It is also possible to use constant values, as long as they are known at compile time.
- They can be used in class and function templates.

```
template<class T, int VAL>
T addValue(const T& x){
  return x + VAL;
}
```

- Not allowed are floating point numbers, null pointer constants or string literals.
- String literals can be used by defining a variable with external linkage:

```
template<const char* name>
class MyClass { ... }

extern const char s[] = "hello";

MyClass<s> x;
```

Allowed Template Parameters

```
template <typename T, T nontype_param> class C;

C<int,33> c1;           // integer
int a;
C<int*,&a> c2;          // address of a variable
void f();
void f(int);
C<void(*)(int),f> c3; // function pointer to f(int)

class X {
  public:
    int n;
    static bool b;
};
C<bool&,X::b> c4;       // static class members
C<int X::*,&X::n> c5; // pointer to members
template<typename T>
void templ_func();
C<void(),&templ_func<double> > c6; // function templates
                                   // are also functions
```

Prohibited Templates Parameters

```
template <typename T, T nontype_param> class C;

class Base {
  public:
    int i;
    static bool b;
};
Base base;

class Derived : public Base {
};
Derived derived;

C<Base*,&derived> err1; // no automatic conversion to base class
C<int &,base.i> err2;   // attributes of objects are no variables
int a[10];
C<int*,&a[0]> err3;     // addresses of individual array elements
                        // are not allowed
```

Inheritance in Class Templates

```
template<typename T>
class MyNumericalSolver : public NumericalSolver<T,3>
{
    T variable;
  public:
    MyNumericalSolver(T val) : NumericalSolver<T,3>(),
                               variable(val)
    {};
}
```

- When a class is derived from a class template, then the template arguments must be specified as part of the base class name.
- This also applies to calling the base class.

Example: Numerical Integration of $\cos(x-1)$

This example realizes the integration of $\cos(x-1)$ with the midpoint rule using templates instead of virtual functions. The files are:

- `cosinetemp.h`: contains the definition and implementation of the functor for $\cos(ax+b)$
- `midpointtemp.h`: contains the definition of a function template, which gets a functor as a template argument and applies the midpoint rule.
- `integrationtemp.cc`: contains the main program that uses the template for the midpoint rule to integrate $\cos(x-1)$ over the range $[1 : \frac{\pi}{2}+1]$.

cosinetemp.h

```
#ifndef COSINECLASS_H
#define COSINECLASS_H

#include <cmath>

// realization of a function cos(a*x+b)
class Cosine
{
  public:
    Cosine(double a=1.0, double b=0.0) : a_(a), b_(b)
    {}
    double operator()(double x) const
    {
        return cos(a_*x+b_);
    }
  private:
    double a_,b_;
};

#endif
```

midpointtemp.h

```
template<typename T>
double MidpointRule(const T& f,double a=0.0, double b=1.0, size_t
    n=1000)
{
    double h = (b-a)/n; // lenght of a single interval

    // compute the integral boxes and sum them
    double result = 0.0;
    for (int i=0; i<n; ++i)
    {
        // evaluate poynomial at midpoint and sum integral value
        result += h * f(a + (i+0.5)*h);
    }

    return result;
}
```

integrationtemp.cc

```
// include system headers
#include <iostream>
// own headers
#include "midpointtemp.h"
#include "cosinetemp.h"

int main()
{
  Cosine cosine(1.0,-1.0);
  std::cout << "Integral␣of␣cos(x-1)␣in␣the␣interval␣[1:Pi/2+1]␣is␣
      "
            << MidpointRule(cosine,1.0,M_PI_2+1.0) << std::endl;

  return 0;
}
```

Summary Static Polymorphism

Static polymorphism:

- uses templates and function overloading.
- isn't supported with native C++ language constructs (so far).
- allows to select the version to be used at compile time.
- creates no overhead.
- allows all optimizations.
- resulting in longer compilation times.

$\Rightarrow$ Static polymorphism is particularly suitable when many short function calls are required (such as access to matrix elements...)

Dynamic versus Static Polymorphism

- Polymorphism with inheritance is limited and dynamic:
  - Limited means that the interface of all realizations is determined by the definition of the common base class.
  - Dynamic means that the decision which class is used to realize the interface is made on the fly.
- Polymorphism implemented with templates is unlimited and static:
  - Unlimited means that the interfaces of all the classes participating in the polymorphism are not fixed.
  - Static means that the class that implements the interface is already chosen at compile time.

Dynamic versus Static Polymorphism

- Dynamic Polymorphism:
  - Allows elegant management of heterogeneous sets.
  - Leads to smaller program size.
  - Libraries can be distributed as pure binary code. It is not necessary to publish the source code of the implementation.
- Static Polymorphism:
  - Easy implementation of (homogeneous) container classes.
  - Most of the time faster program execution.
  - Classes that implement only parts of the interface can be used, as long as only these parts are actually used.