Templates and Generic Programming

Shahram Rahatlou

Sapienza Università di Roma

http://www.roma1.infn.it/people/rahatlou/programmazione++/

Corso di Programmazione++

Roma, 8 June 2009

Today's Lecture

- Templates in C++ and generic programming
  - What is Template?
  - What is a Template useful for?
  - Examples
  - Standard Template Library

Generic Programming

- Programming style emphasizing use of “generics” technique

- Generics technique in computer science:
  - allow one value to take different data types as long as certain contracts are kept
    - For example types having same signature
    - Remember polymorphism

- Simple idea to define a code prototype or “template” that can be applied to different kinds (types) of data

- Template can be “specialized” for different data types

- A range of related functions or types related through templates

C++ Template

- Powerful feature that allows generic programming (but not only) in C++

- Two kinds of template in C++
  - Function template: a function prototype to act in identical manner on all types of input arguments
  - Class template: a class with same behavior for different types of data

- How does template work
  - One prototype written by user
  - Code generated by compiler for different template types and compiled
    - polymorphic code at compile time with no run-time overhead

Function Template

- Functions that perform “identical” operation regardless of type of argument
  - Error at COMPILATION TIME if requested operation not implemented for particular data type

- Template syntax
  - Two keywords used to provide parameters: **`typename`** and **`class`**
    - No difference between the two
    - **`class`** is a generic name here and can refer to a built in type as well

```
template< typename T >

template< typename InputType >

template< class InputType >

template< class InputType, typename OutputType>
```

Example of Function Template

```
// example1.cpp
#include <iostream>
#include <string>
#include <typeinfo>
using namespace std;

template< typename  T >
void printObject(const T& input) {
  cout << "printObject(const T& input): with T = " << typeid( T ).name() << endl;
  cout << input << endl;
}

int main() {

  int i = 456;
  double x = 1.234;
  float  y = -0.23;
  string name("jane");

  printObject( i );
  printObject( x );
  printObject( y );
  printObject( name );

  return 0;
}
```

`typeinfo` header needed to use `typeid()` function

Format of `name()` depends on compiler

```
$ g++ -Wall -o example1 example1.cpp
$ ./example1
printObject(const T& input): with T = i
456
printObject(const T& input): with T = d
1.234
printObject(const T& input): with T = f
-0.23
printObject(const T& input): with T = Ss
jane
```

Understanding Templates

```
// example1.cpp
#include <iostream>
#include <string>
#include <typeinfo>
using namespace std;

template< typename  T >
void printObject(const T& input) {
  cout << "printObject(const T& input): with T = " << typeid( T ).name() << endl;
  cout << input << endl;
}

int main() {

  int i = 456;
  double x = 1.234;
  float  y = -0.23;
  string name("jane");

  printObject( i );
  printObject( x );
  printObject( y );
  printObject( name );

  return 0;
}
```

Comiler generates actual code for

```
printObject( const int& input )
printObject( const double& input )
printObject( const float& input )
printObject( const string& input )
```

Another Template Function

```
// example2.cpp
#include <iostream>
#include <string>
#include <typeinfo>
using namespace std;

template< class  DataType >
void printArray(const DataType* data, int nMax) {
  cout << "printObject(const T& input): with DataType = "
       << typeid( DataType ).name() << endl;
  for(int i=0; i<nMax; ++i) {
    cout << data[i] << "\t";
  }
  cout << endl;
}

int main() {

  int i[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

  const int n1 = 3;
  double x[n1] = { -0.1, 2.2, 12.21};
  string days[] = { "Mon", "Tue", "Wed", "Thur", "Fri", "Sat", "Sun"};

  printArray( i, 10 );
  printArray( x, n1 );
  printArray( days, 7 );

  return 0;
}
```

```
$ g++ -Wall -o example2 example2.cpp
$ ./example2
printObject(const T& input): with DataType = i
0       1       2       3       4       5       6       7       8       9
printObject(const T& input): with DataType = d
-0.1    2.2     12.21
printObject(const T& input): with DataType = Ss
Mon     Tue     Wed     Thur    Fri     Sat     Sun
$ g++ -Wall -o example2 example2.cpp
```

Typical Error with Template

```
// example3.cpp
#include <iostream>
#include <string>
#include <typeinfo>
using namespace std;

template< typename  T >
void printObject(const T& input) {
  cout << "printObject(const T& input): with T = " << typeid( T ).name() << endl;
  cout << input << endl;
}

class Dummy {
  public:
    Dummy(const string& name="") {
      name_ = name;
    }
  private:
    string name_;
};

int main() {

  string name("jane");
  Dummy  bad("bad");

  printObject( name );
  printObject( bad );

  return 0;
}
```

No operator<<() implemented for class Dummy!

Error at compilation time because no code can be generated

No prototype to use to generate printArray(const Dummy& input)

```
$ g++ -Wall -o example3 example3.cpp
$ g++ -Wall -o example3 example3.cpp
example3.cpp: In function `void printObject(const T&) [with T = Dummy]':
example3.cpp:28:   instantiated from here
example3.cpp:10: error: no match for 'operator<<' in 'std::cout << input'
Follwed by 100s of other error messages!
```

Compiling Template Code

- Template functions (and classes) are incomplete without specialization with specific data type
- Template code can not be compiled alone
  - Cannot put template code in source file and into the library
- Remember: code for each specialization “generated” by compiler right before compiling it
- Template functions and classes (including member functions) implemented in header files only
- Data types used must implement the operations used in template function

C++ Template and C Macros

- They “might” look similar at first glance but fundamentally VERY different
- Both Templates and Macros are expanded at compile time by compiler and no run-time overhead
- Compiler performs type-checking with template functions and classes
  - Make sure no syntax or type errors in the template code

Class Template

- Class templates are similar to template functions
  - Actual class generated by compiler based on type of parameter provided by user
  - Also referred to as parameterized types

- Class templates extremely useful to implement containers of objects, iterators, and associative maps
  - containers: **`vector<T>`**, **`collection<T>`**, and **`list<T>`** of objects have well defined behavior independently from particular type **`T`**
    - get nth element regardless of type
  - Iterators: **`vector<T>::iterator`** manipulates objects in a vector of objects of type **`T`**

  - Associative maps: **`map<typename Key, typename Value>`** can be used to relate objects of type **`Key`** to objects of type **`Value`**

Class Template Syntax

```
// example5.cpp
#include <iostream>
#include <string>
#include <typeinfo>
using namespace std;

template< typename  T >
class Dummy {
  public:
    Dummy(const T& data);
    ~Dummy();
    void print() const;
  private:
    T* data_;
};

template<class T>
Dummy<T>::Dummy(const T& data) {
  data_ = new T(data);
}

template<class T>
Dummy<T>::~Dummy() {
  delete data_;
}
```

```
template<class T>
void
Dummy<T>::print() const {
  cout << "Dummy<T>::print() with type T = "
       << typeid(T).name()
       << ", *data_: " << *data_
       << endl;

}

int main() {
  Dummy<std::string> d1( std::string("test") );

  double x = 1.23;
  Dummy<double> d2( x );

  d1.print();
  d2.print();

  return 0;
}
```

```
$ g++ -Wall -o example5 example5.cpp
$ ./example5
Dummy<T>::print() with type T = Ss, *data_: test
Dummy<T>::print() with type T = d, *data_: 1.23
```

Header and Source Files for Template Classes

```
#ifndef Dummy_h_
#define Dummy_h_

template< typename  T >
class Dummy {
  public:
    Dummy(const T& data);
    ~Dummy();
    void print() const;

  private:
    T* data_;
};
#endif
```

```
// example5-bad.cpp
#include <iostream>
#include <string>
#include <typeinfo>
using namespace std;

#include "Dummy_bis.h"

int main() {
  Dummy<std::string> d1( std::string("test") );

  double x = 1.23;
  Dummy<double> d2( x );

  d1.print();
  d2.print();

  return 0;
}
```

Can’t separate into header and source files… compiler NEEDS the source code for template class to generate specialized template code!

```
$ g++ -Wall -o example5-bad example5-bad.cpp Dummy_bis.cc
/tmp/ccRWplj8.o:example5-bad.cpp:(.text+0x1a6): undefined reference to `Dummy<std::basic_string<char,
std::char_traits<char>, std::allocator<char> > >::Dummy(std::basic_string<char, std::char_traits<char>,
std::allocator<char> > const&)'
/tmp/ccRWplj8.o:example5-bad.cpp:(.text+0x24d): undefined reference to `Dummy<double>::Dummy(double const&)'
/tmp/ccRWplj8.o:example5-bad.cpp:(.text+0x25f): undefined reference to `Dummy<std::basic_string<char,
std::char_traits<char>, std::allocator<char> > >::print() const'
/tmp/ccRWplj8.o:example5-bad.cpp:(.text+0x26a): undefined reference to `Dummy<double>::print() const'
/tmp/ccRWplj8.o:example5-bad.cpp:(.text+0x27c): undefined reference to `Dummy<double>::~Dummy()'
/tmp/ccRWplj8.o:example5-bad.cpp:(.text+0x28e): undefined reference to `Dummy<std::basic_string<char,
std::char_traits<char>, std::allocator<char> > >::~Dummy()'
/tmp/ccRWplj8.o:example5-bad.cpp:(.text+0x2f3): undefined reference to `Dummy<double>::~Dummy()'
/tmp/ccRWplj8.o:example5-bad.cpp:(.text+0x31d): undefined reference to `Dummy<std::basic_string<char,
std::char_traits<char>, std::allocator<char> > >::~Dummy()'
collect2: ld returned 1 exit status
```

Template Class in Header Only

```
#ifndef Dummy_h_
#define Dummy_h_

#include<iostream>

template< typename  T >
class Dummy {
  public:
    Dummy(const T& data);
    ~Dummy();
    void print() const;

  private:
    T* data_;
};

template<class T>
Dummy<T>::Dummy(const T& data) {
  data_ = new T(data);
}


template<class T>
Dummy<T>::~Dummy() {
  delete data_;
}

template<class T>
void
Dummy<T>::print() const {
  std::cout << "Dummy<T>::print() with type T = "
       << typeid(T).name()
       << ", *data_: " << *data_
       << std::endl;
}
#endif
```

```
// example5bis.cpp
#include <iostream>
#include <string>
#include <typeinfo>
using namespace std;

#include "Dummy.h"

int main() {
  Dummy<std::string> d1( std::string("test") );

  double x = 1.23;
  Dummy<double> d2( x );

  d1.print();
  d2.print();

  return 0;
}
```

Template classes must be implemented in the header file

Can still separate declaration and implementation as long as it stays within header file

Including header file provides source code to compiler in order to generate code for specialized templates

Standard Template Library (STL)

- Powerful library implementing template-based and reusable components using generic programming

- Provides comprehensive set of common data structures and algorithms to manipulate such data structures
  - Particularly useful in industrial applications
  - STL now part of the Standard C++ Library

- Popular and key components largely used
  - Containers: templated data structures that can store any type of data
  - Iterators provide pointers to individual elements of containers
  - Algorithms to manipulate data structures: insert, delete, sort, copy elements of containers

Example: map<string,int> and iterators

```cpp
// example4.cpp
#include <iostream>
#include <string>
#include <map>

int main() {

  std::map<std::string, int> days;

  days[std::string("Jan")] = 31;
  days[std::string("Feb")] = 28;
  days[std::string("Mar")] = 31;
  days[std::string("Apr")] = 30;
  days[std::string("May")] = 31;
  days[std::string("Jun")] = 30;
  days[std::string("Jul")] = 31;
  days[std::string("Aug")] = 31;
  days[std::string("Sep")] = 30;
  days[std::string("Oct")] = 31;
  days[std::string("Nov")] = 30;
  days[std::string("Dec")] = 31;

  std::map<std::string, int>::const_iterator iter;

  for( iter = days.begin(); // first elemnt
       iter!= days.end();  // last element
       ++iter  // step
     ) {
    std::cout << "key iter->first: " << iter->first
              << " value iter->second: " << iter->second
              << std::endl;
  }

  // lookup non-exisiting key
  std::string december("december");

  iter = days.find(december);

  if( iter != days.end() ) {
    std::cout << days["Dec"] << std::endl;
  } else {
     std::cout << "Bad key: " << december << std::endl;
  }

  return 0;
}
```

```
$ g++ -Wall -o example4 example4.cpp
$ ./example4
key iter->first: Apr value iter->second: 30
key iter->first: Aug value iter->second: 31
key iter->first: Dec value iter->second: 31
key iter->first: Feb value iter->second: 28
key iter->first: Jan value iter->second: 31
key iter->first: Jul value iter->second: 31
key iter->first: Jun value iter->second: 30
key iter->first: Mar value iter->second: 31
key iter->first: May value iter->second: 31
key iter->first: Nov value iter->second: 30
key iter->first: Oct value iter->second: 31
key iter->first: Sep value iter->second: 30
Bad key: december
```

Both map and iterator classes are template classes!

Same code can be used for any data type

polymorphism at compilation time!

Non-Type Parameter for Class Template

```
template<class T, int maxSize=7>
class Vector {
  public:
    Vector() {
      size_ = maxSize;
      for(int i=0; i<size_; ++i) {
        data_[i] = T();
      }
    }

  private:
    int size_;
    T data_[maxSize];
};
```

- Template can be used also with non-type parameters
- Helpful to instanciate data members in the constructor
- Replace dynamic allocation with automatic objects
  - Easier memory management
  - Fatser code since variables are automatic and no new/delete needed!

Class Vector with Template

Vector.h

```
#include <iostream>

template<class T, int maxSize=7>
class Vector {
  public:
    Vector() {
      size_ = maxSize;
      for(int i=0; i<size_; ++i) {
        data_[i] = T();
      }
    }

    ~Vector() {};
    int size() const { return size_; }
    T& operator[](int index);
    const T& operator[](int index) const;
    //friend std::ostream& operator<<(ostream& os,
                         const Vector<T,maxSize>& vec);

  private:
    int size_;
    T data_[maxSize];
};

template<typename T, int maxSize>
T& Vector<T,maxSize>::operator[](int index){
  return data_[index];
}
```

```
template<typename T, int maxSize>
const T& Vector<T,maxSize>::operator[](int index) const{
  return data_[index];
}

template<typename T, int maxSize>
std::ostream&
operator<<(std::ostream& os, const Vector<T,maxSize>&
vec) {
  os << "vector with " << vec.size() << " elements: " <<
endl;
  for(int i=0; i<vec.size(); ++i) {
    os << "i: " << i
       << " v[i]: " << vec[i] << endl;
  }
  return os;
}
```

```
// example6.cpp
#include <iostream>
#include <string>
using namespace std;

#include "Vector.h"

int main() {
  Vector<string> vstr;
  vstr[0] = "test";
  vstr[1] = "foo";
  cout << vstr << endl;
  Vector<double,1000> v1;
  return 0;
}
```

```
$ g++ -o example6 example6.cpp
$ ./example6
vector with 7 elements:
i: 0 v[i]: test
i: 1 v[i]: foo
i: 2 v[i]:
i: 3 v[i]:
i: 4 v[i]:
i: 5 v[i]:
i: 6 v[i]:
```

auto_ptr<class T>

- **`auto_ptr`** is an example of smart pointers in C++
  - Behaves exactly as a regular pointer
    - You can dereference an **`auto_ptr`** or use it any other way you would use a regular pointer
  - Takes care of deleting of object it points to
  - Correctly transfers ownership when copying pointers

http://gcc.gnu.org/onlinedocs/libstdc++/latest-doxygen/classstd_1_1auto__ptr.html

```
Public Member Functions
auto_ptr (auto_ptr_ref< element_type > __ref) throw ()
template<typename _Tp1> auto_ptr (auto_ptr< _Tp1 > &__a) throw ()
auto_ptr (auto_ptr &__a) throw ()
auto_ptr (element_type *__p=0) throw ()
element_type * get () const throw ()
element_type & operator * () const throw ()
template<typename _Tp1> operator auto_ptr () throw ()
template<typename _Tp1> operator auto_ptr_ref () throw ()
element_type * operator-> () const throw ()
auto_ptr & operator= (auto_ptr_ref< element_type > __ref) throw ()
template<typename _Tp1> auto_ptr & operator= (auto_ptr< _Tp1 > &__a) throw ()
auto_ptr & operator= (auto_ptr &__a) throw ()
element_type * release () throw ()
void reset (element_type *__p=0) throw ()
~auto_ptr ()
```

what is throw() ?

Exceptions… will be discussed later today

Example of `auto_ptr<T>`

```
// autoptr.cpp

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

int main() {

  auto_ptr<string> p1( new string("name") );

  string s1 = *p1;

  cout << "p1: " << &p1 << "\t*p1: " << *p1
       << "\t&s1: " << &s1 << "\ts1: " << s1 << endl;

  auto_ptr<string> p2;
  p2 = p1;

  if( p1.get() == 0 ) {
     cout << "p1 passed ownership to p2" << endl;
  }

  cout << "p1.get(): " << p1.get()
       << "\tp2.get(): " << p2.get() << endl;

  cout << *p2 << endl;
  cout << *p1 << endl;

  return 0;
}
```

Use p1 as a regular pointer
by using the dereferencing operator *

Copy value of string pointed to by p1
in s1

p1 transfers ownership of dynamic
object to p2 and points to NULL

p1 is an object not a pointer
call p1.get() not p1->get()

operator*() is overloaded to
make p1 'look' like a pointer for user

```
$ g++ -Wall -o autoptr autoptr.cpp
$ ./autoptr
p1: 0x23eef0    *p1: name       &s1: 0x23eee0   s1: name
p1 passed ownership to p2
p1.get(): 0     p2.get(): 0x4a0288
name
Segmentation fault (core dumped)
```

Segmentation violation caused
by accessing a null pointer
contained in p1