Templates and Generic Programming

Shahram Rahatlou

Computing Methods in Physics

`http://www.roma1.infn.it/people/rahatlou/cmp/`

Anno Accademico 2018/19

SAPIENZA
UNIVERSITÀ DI ROMA

Today's Lecture

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

- Error handling in C++ with Exceptions

Generic Programming

- Programming style emphasising 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 *specialised* 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;

#include "Vector3D.h"

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");
  Vector3D v(1.2, -0.3, 4.5);

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

  return 0;
}
```

typeinfo header needed to use typeid() function

Format of name() depends on each compiler

```
$ g++ -o /tmp/app example1.cpp Vector3D.cc
$ /tmp/app
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 =
NSt3__112basic_stringIcNS_11char_traitsIcEENS_9
allocatorIcEEEE
jane
printObject(const T& input): with T = 8Vector3D
(1.2 , -0.3 , 4.5)
```

Understanding Templates

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

#include "Vector3D.h"

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");
  Vector3D v(1.2, -0.3, 4.5);

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

  return 0;
}
```

Compiler generates actual code for

```
printObject( const int& input )
printObject( const double& input )
printObject( const float& input )
printObject( const string& input )
printObject( const Vector3D& 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++ -o /tmp/example2 example2.cpp
$ /tmp/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 = NSt3__112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEEE
Mon         Tue         Wed         Thur        Fri         Sat         Sun
```

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++ -o /tmp/example3 example3.cpp
example3.cpp:10:8: error: invalid operands to binary expression ('std::__1::ostream' (aka 'basic_ostream<char>') and
      'const Dummy')
  cout << input << endl;
  ~~~~ ^  ~~~~~
example3.cpp:28:3: note: in instantiation of function template specialization 'printObject<Dummy>' requested here
  printObject( bad );
[…..] Followed by 100s of other error messages!
```

Compiling Template Code

- Template functions (and classes) are incomplete without specialisation 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 specialisation “generated” by compiler at compilation time

- ***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 parameterised 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 behaviour independently from particular type **`T`**
    - get n^{th} 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++ -o /tmp/example5 example5.cpp
$ /tmp/example5
Dummy<T>::print() with type T =
NSt3__112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEEE, *data_: test
Dummy<T>::print() with type T = d, *data_: 1.23
```

Header and Source Files for Template Classes

```
#ifndef DummyBis_h_
#define DummyBis_h_

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

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

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

#include "DummyBis.h"

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

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

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

  return 0;
}
```

```
$ g++ -o /tmp/bad example5-bad.cpp
Undefined symbols for architecture x86_64:
  "DummyBis<std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> >
>::DummyBis(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> > const&)", referenced from:
      _main in example5-bad-ad84c5.o
  "DummyBis<std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> > >::~DummyBis()", referenced from:
      _main in example5-bad-ad84c5.o
  "DummyBis<double>::DummyBis(double const&)", referenced from:
      _main in example5-bad-ad84c5.o
  "DummyBis<double>::~DummyBis
      _main in example5-bad-ad8
  "DummyBis<std::__1::basic_str
      _main in example5-bad-ad8
  "DummyBis<double>::print() const", referenced from:
      _main in example5-bad-ad84c5.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

Can't separate into header and source files… compiler **needs** the source code for template class to generate specialized template code!