CS3520 Programming in C++
Generic Programming

Ken Baclawski

Fall 2016

Outline

- Generics
- Turing Completeness
- The matrix package
- The types package
- The bonacci package
- Comparison of Java and C++ templates

Generics

Generics

- Also called *templates* or parametrized types
- Both classes and methods can be templates
- Act like functions with parameters but at compile-time
- All templates are instantiated prior to compilation
- All template code is in the header
  - A template class has no source file
  - A template method is defined in the header
  - Some libraries consist only of templates: *header-only* libraries

Template Parameters

- Parameters can be:
  - class names (most common case)
  - values evaluated at compile-time
  - template name
- Coding Style requirement: Template parameters are single uppercase letters

Instantiation

- Specifying the template parameters *instantiates* the template
  - a template class to a concrete class
  - a template method to a concrete method
- All template parameters must be specified for a template class (except for default parameters)
- Template parameters need not be specified for a template method if they can be inferred
  - They are often specified even when they can be inferred for the sake of additional type checking

Specialization

- A template can be *specialized* to another one
- The specialized template has fewer template parameters
  - The specialized template can have no template parameters at all, but it is still a template
- Most commonly used for faster algorithms and more efficient data structures in special cases

Turing Completeness

Turing Completeness

- A programming language is *Turing-complete* if it can compute the same functions as any computer
- Imperative language minimal requirement
  - Loops
  - Loop termination conditions
  - Variables that can be assigned
- Functional language minimal requirement
  - Recursive functions
  - Base cases for recursion
  - Function application

Template Computations

- The template language is Turing complete
- See the bonacci package
- Computations are performed recursively
- Should only be used for computations that are appropriate for the system design
  - In practice they will be very small computations
  - If the computations get complicated, then perform the computations in the normal way and then incorporate them into the build process.

The matrix package

Requirements

- A generic matrix that can have entries of any type
- Construct a matrix with specified number of rows and columns, and an initial value for all of the entries
- Overload `operator()` to give access to one entry in the matrix
  - All both accessing and modifying the entry
- Specialize to a matrix with double entries
- Overload the multiplication operator to perform matrix multiplication
- All code is in the header file `Matrix.h`

Generic Matrix Template

```
#ifndef MATRIX_MATRIX_H
#define MATRIX_MATRIX_H

#include <stdexcept>

namespace matrix {

/**
 * A matrix is an array of arrays.  This template
 * allows a matrix to have entries with an
 * arbitrary type.  It also overloads function
 * notation so that one can obtain the matrix
 * entry of a matrix m in row i and column j by
 * writing m(i,j).
 *
 * @author Ken Baclawski
 */
template<typename T>
class Matrix {
```

Remember to put these on the first two lines of the file

Specify that this class is a template and that it has a type parameter

Generic Matrix Template

```
public:
  /**
   * Construct a matrix with initial values
   * @throws domain_error if one of the dimensions is 0.
   */
  Matrix(/** The number of rows. */ unsigned int rowCount,
         /** The number of columns. */ unsigned int columnCount,
         /** The initial value of each entry. */
         const T& initialValue) {

    // Check that the counts are nonzero
    if (rowCount == 0) {
      throw std::domain_error("Attempt to construct a matrix with no rows");
    }
    if (columnCount == 0) {
      throw std::domain_error("Attempt to construct a matrix with no columns");
    }

    // Construct one row with the initial values
    std::vector<T> row(columnCount, initialValue);

    // Copy the row to make the matrix
    for (int i = 0; i < rowCount; ++i) {
      matrix_.push_back(row);
    }
  }
```

The template parameter specifies the type of the initial value

The template parameter specifies the type of the entries in the matrix

Generic Matrix Template

```
/**
 * Get one of the entries.  The entry can be modified.
 * @return A reference to the specified entry.
 * @throw out_of_range if an index is out of range.
 */
T& operator()(/** The index of the row. */
              unsigned int rowIndex,
              /** The index of the column. */
              unsigned int columnIndex) {
  return matrix_.at(rowIndex).at(columnIndex);
}
```

The template parameter specifies the return reference type and the type of entries in the matrix

Generic Matrix Template

```
/**
 * Add two matrices.
 * @return The sum of the matrices.
 * @throws domain_error if the matrices have incompatible sizes
 */
Matrix<T> operator+(/** The other matrix in the sum. */
                    const Matrix<T>& addend) const {

  // Get the dimensions of the matrices

  unsigned int rowCount = matrix_.size();
  unsigned int columnCount = matrix_.at(0).size();

  // Check that the matrices are the same size

  if (addend.matrix_.size() != rowCount ||
      addend.matrix_.at(0).size() != columnCount) {
    throw std::domain_error("Attempt to add matrices "
                            "with different dimensions");
  }
```

Overloading the plus operator

The template parameter

Generic Matrix Template

```
    // Compute the sum of the matrices

    Matrix<T> sum(rowCount, columnCount, matrix_.at(0).at(0));
    for (unsigned int i = 0; i < rowCount; ++i) {
      for (unsigned int j = 0; j < columnCount; ++j) {
        sum.matrix_.at(i).at(j) =
          matrix_.at(i).at(j) + addend.matrix_.at(i).at(j);
      }
    }
    return sum;
  }

private:
  /**
   * The vector of vectors.
   */
  std::vector<std::vector<T>> matrix_;
};
```

The template parameter

The template parameter

Specialized Matrix Template

```
/**
 * A numeric matrix.  This is a specialization of
 * the generic matrix.  It has the additional
 * operation of multiplication of matrices.
 *
 * @author Ken Baclawski
 */
template<>
class Matrix<double> {
public:
  /**
   * Construct a matrix with initial values.
   */
  Matrix(/** The number of rows. */ unsigned int rowCount,
         /** The number of columns. */ unsigned int columnCount,
         /** The initial value of all entries. */
         double initialValue = 0.0) {
...
```

Specialize a template by omitting one or more template parameter

Template specializations do not inherit constructors or methods

Default initial value is 0.0

Specialized Matrix Template

- The rest of the specialized matrix template is the same as the generic matrix template except for `double` instead of `T`
- The main use of template specialization is for better algorithms and more efficient data structures
- How to inherit from generic templates to specialized templates will be covered later

Matrix Test Module

```
#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE Test of the matrix package
#include <boost/test/unit_test.hpp>

#include <string>
#include <stdexcept>

#include "Matrix.h"

using namespace std;
using namespace matrix;

/**
 * @namespace matrix A generic matrix with arbitrary
 * entry type and a specialization to double.
 *
 * @author Ken Baclawski
 */
```

The usual structure of a Boost Unit Test module

Matrix Test Case

```
BOOST_AUTO_TEST_CASE(matrix_test_generic) {

  // Check that a matrix must be nonempty
  BOOST_CHECK_THROW(Matrix<string>(5, 0, ""), domain_error);

  // Construct a 2x2 matrix with empty strings as entries
  Matrix<string> smatrix(2, 2, "");

  // Change the entry in position (0,1) to the string “hello”
  smatrix(0, 1) = "hello";

  // Check that the entry in position (0,1) is the string “hello”
  BOOST_CHECK_EQUAL(smatrix(0, 1), "hello");

  // Check that the entry in position (1,0) is the empty string
  BOOST_CHECK_EQUAL(smatrix(1, 0), "");

  // Check that an entry outside the range will throw an exception
  BOOST_CHECK_THROW(smatrix(5, 5), out_of_range);

  // Check the addition operator
  BOOST_CHECK_EQUAL((smatrix + smatrix)(0, 1), "hellohello");
}
```

Overloaded addition operator

Matrix Test Case

```
BOOST_AUTO_TEST_CASE(matrix_test_specialization) {

  // Construct a pair of compatible matrices
  Matrix<double> m1(1, 2);
  Matrix<double> m2(1, 2);

  // Set some values
  m1(0, 0) = 2;
  m2(0, 0) = 2;

  // Check that the product has the right value
  BOOST_CHECK_EQUAL((m1 + m2)(0, 0), 4);
}
```

Overloaded addition operator

The types package

C++ Types

- Run-Time Type Identification (RTTI)
- `decltype` operator returns the type of an expression
- `typeid` operator returns a `type_info` object
  - Requires the `<typeinfo>` library
  - Most useful method is `name()`
  - Unfortunately, g++ returns the name of the type in *mangled* form

Package Requirements

- A static method that demangles the type name returned by the `typeid` operator to produce a human-readable type name
- A static method that returns the demangled name of an expression
  - This must be a *template method* because it must accept a parameter of any type
- A static method that prints the demangled name of an expression and also the value of the expression

Demangler Header

```
#ifndef TYPES_DEMANGLER_H
#define TYPES_DEMANGLER_H

#include <string>
#include <typeinfo>
#include <iostream>

namespace types {

/**
 * Demangler utility.  The C++ typeid function
 * returns the mangled name of the type of an
 * expression.  The notation is obscure and
 * difficult for a human to parse.  The demangler
 * converts the mangled name of the type to a
 * human-readable form.
 *
 * @author Ken Baclawski
 */
class Demangler {
```

Remember to put these on the first two lines of the file

This class will have both a header and a source file because it only has a template method and is not a template class

Demangler Header

This is not a template method so code is in the source file

```
public:
  /**
   * Get the human-readable type name of a mangled
   * type name.
   * @return The demangled form of the string.
   */
  static std::string demangle(/** A mangled type name. */
                              const std::string& mangledName);

  /**
   * Get the human-readable type of an expression.
   * @return The type of the expression.
   */
  template<typename T>
  static std::string getExpressionType(T expression) {
    return demangle(typeid(expression).name());
  }
```

Specify that the method is a template

Code is in the header

The parameter can have any type

Demangler Header

```
  /**
   * Print the human-readable type of an
   * expression and its value if this is possible.
   */
  template<typename T>
  static void printExpressionAndType(T expression) {
    std::cout << "An expression of type "
              << getExpressionType(expression)
              << " has value " << expression
              << std::endl;
  }
};

}

#endif
```

Another template method so code is in the header

Demangler Source

```
#include <cxxabi.h>
#include <typeinfo>
#include <string>
#include "Demangler.h"
using namespace std;
using namespace types;

string Demangler::demangle(const string& mangledName) {

  // The demangler requires a C-style string and
  // returns one that must be freed.

  int status = 0;
  char* demangledName =
    abi::__cxa_demangle(mangledName.c_str(),
                        nullptr, nullptr, &status);
```

This may be specific to the compiler, so may not be portable

Demangler Source

```
  if (demangledName) {
    if (status == 0) {

      // A demangled name was found, so convert
      // it to a string and return it

      string demangledNameString(demangledName);
      free(demangledName);
      return demangledNameString;
    } else {

      // No demangled name was found, but must
      // still free the C-style string

      free(demangledName);
    }
  }
  return "Unknown Type";
}
```

Make sure that the C-style string is freed in all cases in which one was returned

Demangler Test Module

```
#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE Test of the types package
#include <boost/test/unit_test.hpp>

#include <iostream>
#include "Demangler.h"
using namespace std;
using namespace types;

/**
 * @namespace types This package has a utility
 * that gives the type of an expression in a
 * human-readable form.
 *
 * @author Ken Baclawski
 */
```

Demangler Test Case

```
BOOST_AUTO_TEST_CASE(demangler_test1) {

  // Try some simple types

  int x = 4;
  BOOST_CHECK_EQUAL(Demangler::getExpressionType(x), "int");
  Demangler::printExpressionAndType(x);

  const int y = 5;
  BOOST_CHECK_EQUAL(Demangler::getExpressionType(y), "int");
  Demangler::printExpressionAndType(y);

  int* z = &x;
  BOOST_CHECK_EQUAL(Demangler::getExpressionType(z), "int*");
  Demangler::printExpressionAndType(z);
}
```

These are template methods, but no template parameters were specified because the template arguments can be inferred

Demangler Test Case

```
BOOST_AUTO_TEST_CASE(demangler_test2) {

  // Now try some more complex types

  vector<double> list;
  BOOST_CHECK_EQUAL(Demangler::getExpressionType(list),
           "std::vector<double, std::allocator<double> >");

  vector<vector<double>> matrix;
  BOOST_CHECK_EQUAL(Demangler::getExpressionType(matrix),
    "std::vector<std::vector<double, std::allocator<double> >, "
    "std::allocator<std::vector<double, "
    "std::allocator<double> > > >");
}
```

Demangler Test Case

```
BOOST_AUTO_TEST_CASE(demangler_test3) {

  // Finally, try a complex lambda expression

  vector<vector<double>> matrix;
  vector<double> v;
  vector<double> w;
  auto lambda = [=](const vector<double>& list) mutable
    { v.push_back(1.0); return matrix.size() * list.size(); };
  BOOST_CHECK_EQUAL(Demangler::getExpressionType(lambda),
          "demangler_test3::test_method()::"
          "{lambda(std::vector<double, "
          "std::allocator<double> > const&)#1}");
}
```

The bonacci package

Leonardo Bonacci

- Considered the most talented Western mathematician of the Middle Ages
- His nickname was Fibonacci
- Best known for the Fibonacci numbers
- Main contribution was the popularization of the number system most commonly used today: the Hindu-Arabic numeral system
  - Popularized the system of arithmetic computation now used throughout the world
  - Developed modern accounting methods for business
  - Many other contributions to mathematics

Fibonacci Numbers

- Solution to the problem of population growth of rabbits
- Formula: $f_n = f_{n-1} + f_{n-2}$, $f_0=0$, $f_1=1$
- Commonly used as an example of recursive versus non-recursive algorithm
- Many applications in Computer Science
- Occurs in nature

Fibonacci Template

- Illustrates the use of integer template parameters
- Shows how a function can be defined recursively using templates
- Shows why the C++ template language is Turing-complete
  - One can define functions recursively with templates
  - Recursion is terminated by template specialization
  - One can apply functions by instantiating templates

Fibonacci Template

```
#ifndef BONACCI_FIBONACCI_H
#define BONACCI_FIBONACCI_H

namespace bonacci {

/**
 * Computation of the fibonacci sequence using templates.
 * The fibonacci sequence is defined recursively by
 * f<sub>n</sub> = f<sub>n-1</sub> + f<sub>n-2</sub>,
 * where f<sub>0</sub> = 0 and f<sub>1</sub> = 1.
 *
 * @param N The index of the fibonacci sequence.
 * In other words, N is the subscript in f<sub>N</sub>.
 *
 * @author Ken Baclawski
 */
template<int N>
class Fibonacci {
```

Remember to put these on the first two lines of the file

Fibonacci Template

```
public:
  /**
   * The value of the fibonacci sequence
   * at index N computed at compile-time.
   */
  static const long long value =
    Fibonacci<N-1>::value + Fibonacci<N-2>::value;
};
```

Recursive computation

Fibonacci Template Specialization

```
/**
 * Specification of the base case at 0 for
 * computing the fibonacci sequence.
 * <p>
 * This is called an explicit specialization
 * and must be preceded by template&lt;&gt;.
 */
template<>
class Fibonacci<0> {
public:
  /**
   * The base value of the fibonacci sequence.
   */
  static const long long value = 0;
```

Specialize the template

One of the bases of recursion

Fibonacci Run-Time Definition

```
  /**
   * The computation of the fibonacci sequence at
   * run-time.
   */
  static long long
  getValue(/** The index of the required
               fibonacci number. */ int n) {
    if (n == 0) return 0;
    else if (n == 1) return 1;
    else return getValue(n-1) + getValue(n-2);
  }
};
```

This was added to the template so that one can compare the performance for computing at compile-time versus run-time

Fibonacci Template Specialization

```
/**
 * Specification of the base case at 1 for
 * computing the fibonacci sequence.
 * <p>
 * This is called an explicit specialization
 * and must be preceded by template&lt;&gt;.
 */
template<>
```

Specialize the template

```
class Fibonacci<1> {
public:
  /**
   * The base value of the fibonacci sequence.
   */
  static const long long value = 1;
};
```

The other base of recursion

```
}

#endif
```

Test program

```
#include <iostream>
#include "Fibonacci.h"
using namespace std;
using namespace bonacci;

/**
 * @namespace bonacci Computation of the fibonacci
 * numbers using templates.  The fibonacci numbers
 * were introduced by Leonardo Bonacci, also known
 * as Fibonacci.  Bonacci was responsible for
 * introducing the Hindu-Arabic numeral system to
 * the Western world.
 * <p>
 * The C++ template language is Turing-complete.
 * This package gives a simple example of how to
 * compute an interesting function recursively
 * using templates.
 *
 * @author Ken Baclawski
 */
```

This test program does not use the Boost Unit Test Framework because it is testing compile-time versus run-time performance

Test program

```
/**
 * Main program for computing the fibonacci
 * numbers using templates at compile-time.
 * @return The exit status.  Normal status is 0.
 */
int main() {

  // The following is the maximum value that can
  // be computed for the fibonacci numbers.  The
  // next higher case will fail at compile-time
  // because the values of the functions are not
  // representable with a 64-bit integer.

  cout << "The value of fibonacci(92) is "
       << Fibonacci<92>::value << endl;
```

Fibonacci Test Program

```
  // Now compute the same fibonacci number both at
  // compile-time and run-time.

  cout << "The value of fibonacci(45) at compile-time is "
       << Fibonacci<45>::value << endl;
  cout << "The value of fibonacci(45) at run-time is "
       << Fibonacci<0>::getValue(45) << endl;

  return 0;
}
```

The run-time computation takes much longer! Why?

Comparison of Java and C++ Templates

Java Templates

- Only allows class types as template parameter
  - No primitive type parameters
  - No primitive type arguments
- No default template parameters
- No template specialization
- Uses type erasure, not separate compilation
  - Constructing an array with a template parameter type produces a compiler warning
- Requires explicit interface extension to use operations
- Class must be defined that implements the interface

Matrix Example in Java

- The Matrix template can be written in Java
- However, it requires additional interfaces and classes
  - An interface specifying that one can compute the sum of two objects
  - For each type one wants to use as a template argument one must define a class that implements the interface
  - Each class would need constructors, and methods for hashing, comparing and conversion, in addition to the method in the interface
- When using the Matrix template all entries must be constructed explicitly

Interface for Matrix Example in Java

The `Addable.h` file:

```
public interface Addable<T extends Addable<T>> {
  public T plus(T a);
}
```

Template for Matrix Example in Java

- The `Matrix.java` file:

```
public class Matrix<T extends Addable<T>> {
  public Matrix(int rowCount, int columnCount, T initialValue)
    throws Exception {
    ...
  }

  // Getters and setters
  ...

  public Matrix<T> plus(Matrix<T> addend) throws Exception {
    ...
  }
}
```

Java Template Example for doubles

- The `DoubleAddable.java` file

```
public class DoubleAddable implements Addable<DoubleAddable> {
  // Define plus and times for DoubleAddable objects
  ...
}
```

Using the Java Matrix Template

```
DoubleAddable zero = new DoubleAddable(0.0);
DoubleAddable two = new DoubleAddable(2.0);
Matrix<DoubleAddable> m1 = new Matrix<DoubleAddable>(1, 2, zero);
Matrix<DoubleAddable> m2 = new Matrix<DoubleAddable>(1, 2, zero);
m1.setEntry(0, 0, two);
m2.setEntry(0, 0, two);
System.out.println("Entry 0,0 of m1 + m2 = " +
                   m1.plus(m2).getEntry(0, 0));
```

Comparison

- C++ is a much richer template language
  - Many more features
- Programming and using templates is much easier
  - Operator overloading
  - No need for interfaces and special classes
- Better run-time performance
  - Specialization
  - No special classes

- Java compilation might be faster
  - Erasure rather than separate instantiation
- Java compiled code might be smaller
  - No separate instantiations
- However, interfaces and special classes might result in longer compilation and larger compiled code

Next Class

- Copying, Assigning and Moving
- Assignment #9