Laboratorio di Tecnologie dell'Informazione

Ing. Marco Bertini
bertini@dsi.unifi.it
http://www.dsi.unifi.it/~bertini/

Generic programming

What is generic programming ?

- Generic programming is a style of computer programming in which algorithms are written in terms of to-be-specified-later types that are then instantiated when needed for specific types provided as parameters.
- Generic programming refers to features of certain statically typed programming languages that allow some code to effectively circumvent the static typing requirements, e.g. in C++, a template is a routine in which some parameters are qualified by a type variable.

  Since code generation in C++ depends on concrete types, the template is specialized for each combination of argument types that occur in practice.

Generic programming

- Algorithms are written independently of data
  - data types are filled in during execution
  - functions or classes instantiated with data types
  - formally known as specialization
- A number of algorithms are data-type independent, e.g.: sorting, searching, finding n-th largest, swapping, etc.

Identical tasks for different data types

- Approaches for functions that implement identical tasks for different data types
  - Naïve Approach
  - Function Overloading
  - Function Template

Naïve approach

- Create unique functions with unique names for each combination of data types (e.g. `atoi()`, `atof()`, `atol()`)
  - difficult to keeping track of multiple function names
  - lead to programming errors

Function overloading

- The use of the same name for different C++ functions, distinguished from each other by their parameter lists:
  - Eliminates need to come up with many different names for identical tasks.
  - Reduces the chance of unexpected results caused by using the wrong function name.
  - Code duplication remains: need to code each function for each different type

Function template

- A C++ language construct that allows the compiler to generate multiple versions of a function by allowing parameterized data types.
  - One function definition (a function template).
  - Compiler generates individual functions.

Generic programming & templates

Why templates ?

- C++ templates are principally used for classes and functions that can apply to different types of data.
  - Common examples are the STL container classes and algorithms.
    E.g.: algorithms such as sort are programmed to be able to sort almost any type of items.

Generic programming in C++

- Templates = generic programming
- Two types:
  - function templates
    special functions that can operate with generic types.
  - class templates
    can have members that use template parameters as types

Templates & Inheritance

- Inheritance works hand-in-hand with templates, and supports:
  1. A template class based on a template class
  2. A template class based on a non-template class
  3. A non-template class based on a template class

Coding C++ templates

- The template begins with the heading
  `template<class T>`
- The tag T is used everywhere that the base type is required. Use whatever letter or combination of letters you prefer.
- In general, templates classes (and functions) can have multiple "type" arguments and can also have "non-type" arguments.
- Separate member functions are a separate template.

Function Templates

- Special functions using template types.
- A template parameter is a special kind of parameter used to pass a <u>type as argument</u>
  - just like regular function parameters, but pass types to a function

Function templates declaration format

- `template <class identifier> function_ declaration;`
- `template <typename identifier> function_ declaration;`
- Same functionality, different keywords
  - use `<typename ...>`
  - older compilers only used `<class...>` format

Function templates declaration format - cont.

- `template` keyword must appear before function or class templates...
- ... followed by the list of generic or template types
- can have more than one generic type
- ... followed by function declaration and/or definition

Function template example

- ```
  template <typename myType>
  myType getMax (myType a, myType b) {
   return (a>b?a:b);
  }
  ```
- or even better, using references and const-ness:
- ```
  template <typename myType>
  const myType& getMax (const myType& a, const
  myType& b) {
   return (a>b?a:b);
  }
  ```

Function template usage example

```
int main() {
  int i=5, j=6, k;
  long l=10,m=5,n;

  k=getMax<int>(i,j); // OK
  n=getMax<long>(l,m); //OK
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

```
int main() {
  int i=5, j=6, k;
  long l=10,m=5,n;

  k=getMax(i,j); // OK
  n=getMax(l,m); // OK
  cout << k << endl;
  cout << n << endl;

  k=getMax(i,l); // Wrong:
         // can't mix
types !
  return 0;
}
```

Multiple template types

- define all the required template types after the template keyword, e.g.:
-
```
template <typename T1, typename T2>
const T1& getMax (T1& a, T2& b) {
 return (a>(T1)b?a:(T1)b);
}
```

Instantiating a function template

- When the compiler instantiates a template, it substitutes the template argument for the template parameter throughout the function template.
- Template function call:

```
Function  <TemplateArgList> (FunctionArgList)
```

Function template specialization example

- Function template specialization allows to specialize the code for certain specific types, e.g.:
-
```
template<typename T> inline std::string stringify(const T& x) {
   std::ostringstream out;
   out << x;
   return out.str();
 }
```
-
```
template<> inline std::string stringify<bool>(const bool& x) {
   std::ostringstream out;
   out << std::boolalpha << x;
   return out.str();
 }
```

Class templates

- Classes can have members that use template parameters as type, e.g. a class that stores two elements of any valid type:
-
```
template <class T>
class mypair {
private:
    T values [2];
public:
    mypair (T first, T second) {
      values[0]=first; values[1]=second;
    }
};
```

Class template: non-inline definition

- To define a function member outside the declaration of the class template, always precede that definition with the `template <...>` prefix:

```
template <class T>
class mypair {
    T values [2];
public:
  mypair(T first, T second) {
    values[0]=first;
    values[1]=second;
  }
  T getMax();
};
```

```
template <class T>
T mypair<T>::getMax() {
  T retval;
  retval = a>b? a : b;
  return retval;
}

int main () {
  mypair<int> myobject (100, 75);
  cout << myobject.getMax();
  return 0;
}
```

Templates and polymorphism

- In OO programming the choice of the method to be invoked on an object may be selected a runtime (i.e. polymorphism, with virtual methods)
- In generic programming it's chosen at compile time, when instantiating a template

C++ templates: so many Ts !

- There are three T's in this declaration of the method:
  - first one is the template parameter.
  - second T refers to the type returned by the function
  - third T (the one between angle brackets) specifies that this function's template parameter is also the class template parameter.

Default parameters

- Class templates can have default type arguments.
- As with default argument values of a function, the default type of a template gives the programmer more flexibility in choosing the optimal type for a particular application. E.g.:

```
template <class T, class S = size_t > class Vector {
/*..*/
};
Vector <int> ordinary; //second argument is size_t
Vector <int, unsigned char> tiny(5);
```

Non-type parameters

- Templates can also have regular typed parameters, similar to those found in functions
-
```
template <class T, int N>
class mysequence {
    T memblock [N];
  public:
    void setMember (int x, T value);
    T getMember (int x);
};
```

Non-type parameters - cont.

- ```
  template <class T, int N>
  T mysequence<T,N>::getMember (int x) {
    return memblock[x];
  }
  ```

- ```
  int main () {
    mysequence <int,5> myints;
    mysequence <double,5> myfloats;
    myints.setMember (0,100);
    myfloats.setMember (3,3.1416);
    cout << myints.getMember(0) << '\n';
    cout << myfloats.getMember(3) << '\n';
    return 0;
  }
  ```

Class template specialization

- It is used to define a different implementation for a template when a specific type is passed as template parameter
- Explicitly declare a specialization of that template, e.g.: a class with a sort method that sorts ints, chars, doubles, floats and also need to sort strings based on length, but the algorithm is different (not lexicographic sorting)
- Need to explicitly create template specialization for the sort method when string is passed as type

Class template specialization example

```
template <class T>
class MyContainer {
private:
  T element[100];
public:
  MyContainer(T* arg)
{...};
  void sort() {
    // sorting algorithm
  }
};
```

```
// class template specialization:
template <>
class MyContainer <string> {
  string element[100];
public:
  MyContainer (string *arg) {...};
  void sort() {
          // use a string-length
          // based sort here
  }
};
```

Instantiating a class template

- Class template arguments must be explicit.
- The compiler generates distinct class types called template classes or generated classes.
- When instantiating a template, a compiler substitutes the template argument for the template parameter throughout the class template.
  - for efficiency, the compiler uses an "instantiate on demand" policy of only the required methods

Instantiating a class template - cont.

- To create lists of different data types from a GList template class:

```
// Client code
GList<int> list1;
GList<float> list2;
GList<string> list3;

list1.Insert(356);
list2.Insert(84.375);
list3.Insert("Muffler bolt");
```

Instantiating a class template - cont.

- To create lists of different data types from a GList template c

```
// Client code
GList<int> list
GList<float> li
GList<string> l

list1.Insert(356);
list2.Insert(84.375);
list3.Insert("Muffler bolt");
```

The compiler generates 3 distinct class types:

```
GList_int list1;
Glist_float list2;
GList_string list3;
```

Class template

A complete example

```
1  // Fig. 22.3: tstack1.h
2  // Class template Stack
3  #ifndef TSTACK1_H
4  #define TSTACK1_H
5
6  template< class T >
7  class Stack {
8  public:
9     Stack( int = 10 );    // default constructor (stack size 10)
10          ~Stack() { delete [] stackPtr; } // destructor
11          bool push( const T& ); // push an element onto the stack
12          bool pop( T& );        // pop an element off the stack
13       private:
14          int size;             // # of elements in the stack
15          int top;              // location of the top element
16          T *stackPtr;          // pointer to the stack
17
18          bool isEmpty() const { return top == -1; }      // utility
19          bool isFull() const { return top == size - 1; } // functions
20       };
21
22       // Constructor with default size 10
23       template< class T >
24       Stack< T >::Stack( int s )
25       {
26          size = s > 0 ? s : 10;
27          top = -1;                // Stack is initially empty
28          stackPtr = new T[ size ]; // allocate space for elements
29       }
```

- Class template definition
- Function definitions
- Stack constructor
- push
- pop

```
30
31      // Push an element onto the stack
32      // return 1 if successful, 0 otherwise
33      template< class T >
34      bool Stack< T >::push( const T &pushValue )
35      {
36         if ( !isFull() ) {
37            stackPtr[ ++top ] = pushValue; // place item in Stack
38            return true;  // push successful
39         }
40         return false;     // push unsuccessful
41      }
42
43      // Pop an element off the stack
44      template< class T >
45      bool Stack< T >::pop( T &popValue )
46      {
47         if ( !isEmpty() ) {
48            popValue = stackPtr[ top-- ];  // remove item from Stack
49            return true;  // pop successful
50         }
51         return false;     // pop unsuccessful
52      }
53
54      #endif
```

- Class template definition
- Function definitions
- Stack constructor
- push
- pop

```
55      // Fig. 22.3: fig22_03.cpp
56      // Test driver for Stack template
57      #include <iostream>
58
59      using std::cout;
60      using std::cin;
61      using std::endl;
62
63      #include "tstack1.h"
64
65      int main()
66      {
67         Stack< double > doubleStack( 5 );
68         double f = 1.1;
69         cout << "Pushing elements onto doubleStack\n";
70
71         while ( doubleStack.push( f ) ) { // success true returned
72            cout << f << ' ';
73            f += 1.1;
74         }
75
76         cout << "\nStack is full. Cannot push " << f
77              << "\n\nPopping elements from doubleStack\n";
78
79         while ( doubleStack.pop( f ) )  // success true returned
```

- Include header
- Initialize doubleStack
- Initialize variables
- Function calls
- Function calls
- Output

```
80            cout << f << ' ';
81
82         cout << "\nStack is empty. Cannot pop\n";
83
84         Stack< int > intStack;
85         int i = 1;
86         cout << "\nPushing elements onto intStack\n";
87
88         while ( intStack.push( i ) ) { // success true returned
89            cout << i << ' ';
90            ++i;
91         }
92
93         cout << "\nStack is full. Cannot push " << i
94              << "\n\nPopping elements from intStack\n";
95
96         while ( intStack.pop( i ) )  // success true returned
97            cout << i << ' ';
98
99         cout << "\nStack is empty. Cannot pop\n";
100        return 0;
101     }
```

- Include header
- Initialize doubleStack
- Initialize variables
- Function calls
- Function calls
- Output

Static members and variables

- Each template class or function generated from a template has its own copies of any static variables or members
- Each instantiation of a function template has it's own copy of any static variables defined within the scope of the function

Template constraints

- the operations performed within the implementation of a template implicitly constrain the parameter types; this is called "constraints through use":

```
template <typename T>
.. // some code within a class template..
.. T t1, t2; // implies existence of default ctor
.. t1 + t2   //   and plus operator
..
```

- the code implies that + should be supported by the type T:
  - true for all built-in numerical types
  - can be defined for user-defined types (classes)
- if missing, generates a compile-time error and reported immediately by the compiler

Template constraints - cont.

- a template is partially checked at the point of its definition
- template parameter dependent code can be checked only when the template becomes specified at its instantiation
- the code may work for some type arguments, and fail for some other type arguments (at compile time)
- the implicit constraints of a class/function template are required only if the template becomes instantiated (at compile time)
- and all templates are instantiated only when really needed: an object is created or its particular operation is called
- note that all parameter types need not satisfy all requirements implied by the full template definition - since only some member functions may be actually needed and called for some code

C++ templates: source code organization

- Template classes are coded in the header file.
  - Very few compilers support the coding of template functions in a separate source file and the use of the keyword `export` to make them available in other compilation units.
- C++0x standard has deprecated `export`, so just write your templates in a header file
  - ...templates can be seen as advanced textual substitution.

C++ templates: source code organization - cont.

- It is possible to keep the implementation of a function or the methods of a class in a .cpp file and include this file in the header that contains their declaration (the client code will include the header).

```
// header file compare.h
#ifndef _COMPARE_H_
#define _COMPARE_H_
template<class > int compare(const T& a,
const & b);
// other declarations...
#include “compare.cpp” //contains
implementation
#endif // _COMPARE_H_
```

```
// implementation file compare.cpp
template<class T> int compare(const T& a,
const T& b) {
  if( a < b ) return -1;
  if( b < a ) return 1;
  return 0;
}
// other definitions...
```

Templates and base classes

- There may be some issues with inheritance and templates, e.g. if a base class template is specialized and the specialization does not have the same interface of the general template (remind: in templates interfaces are "implicit"):

```
class CompanyA {
public:
 void sendCleartext(const
string& msg);
 void sendEncrypted(const
string& msg);
 //...
};
```

```
class CompanyZ {
public:
 void sendEncrypted(const
string& msg);
 //...
};

class MsgInfo {...};
```

Templates and base classes - cont.

```
template<typename Company>
class MsgSender {
public:
 //...
 void sendClear(const MsgInfo&
info)
 {
  string msg;
  //create msg from info;
  ...
  Company c;
  c.sendCleartext(msg);
 }
 void sendSecret(const MsgInfo&
info) { ... };
 //...
};
```

```
template<typename Company>
class LoggingMsgSender : public
MsgSender<Company> {
public:
 //...
 void sendClearMsg(const MsgInfo&
info)
 {
  logMsg( ... );
  sendClear(info); // does NOT
compile !
  logMsg( ... );
 }
 // ...
};
```

Temp es - cont.

This template does not work with CompanyZ: the sendClear requires a working sendCleartext that is not available !

```
template<typename Company>
class MsgSender {
public:
 //...
 void sendClear(const MsgInfo&
info)
 {
  string msg;
  //create msg from info;
  ...
  Company c;
  c.sendCleartext(msg);
 }
 void sendSecret(const MsgInfo&
info) { ... };
 //...
};
```

```
template<typename Company>
class LoggingMsgSender : public
MsgSender<Company> {
public:
 //...
 void sendClearMsg(const MsgInfo&
info)
 {
  logMsg( ... );
  sendClear(info); // does NOT
compile !
  logMsg( ... );
 }
 // ...
};
```

Templates and base classes - cont.

- To solve the problem create a specialized version of MsgSender, that does not have a sendClear method:

```
template<>
class MsgSender<CompanyZ> {
public:
 //...
 void sendSecret(const MsgInfo& msg) {
 //...
 }
 //...
};
```

- Still it's not enough: we have to tell the compiler to look at the base class to check if the interface is completely implemented:
  - preface base class calls with this->:

```
void sendClearMsg(const MsgInfo& info)
 {
  logMsg( ... );
  this->sendClear(info); // OK: assumes that it will be inherited
  logMsg( ... );
 }
```

  - use a using declaration, to force compiler to search base class scope:

```
template<typename Company>
class LoggingMsgSender : public MsgSender<Company> {
public:
 using MsgSender<Company>::sendClear; // OK: tell compilers it's in base class
```

Code bloat

- Because templates are handled by textual substitution, multiple instances of the same template with different types will result in multiple instances of the code.
- Made worse by the common requirement to place all member functions inline (resulting in additional multiple copies).
  - every call to a template function or the member functions of a template class will be inlined potentially resulting in numerous copies of the same code.

Reducing code bloat

- Code a template class as a wrapper class that does relatively little, but it can inherit from a non-template class whose member functions can then be coded in a separately compiled module.
- This technique is employed in STL for many standard containers and standard algorithms whereby the non-template base class handles a generic `void*` pointer type. The template class provides a type-safe interface to the unsafe base class

Why use templates ?

- Add extra type checking for functions that would otherwise take void pointers: templates are type-safe. Since the types that templates act upon are known at compile time, the compiler can perform type checking before errors occur.
- Create a type-safe collection class (for example, a stack) that can operate on data of any type.
- Create only one generic version of your class or function instead of manually creating specializations.
- Code understanding: templates can be easy to understand, since they can provide a straightforward way of abstracting type information.

Templates vs. Macros

- C++ templates resemble but are not macros:
- the once instantiated name identifies the same generated class-instance at all places
- compiler typically represents the class with some generated internal name and places the instantiation into some internal repository for future use
- any free (parameter-independent) names inside a template are bound at the point of the definition of the template

Templates vs. Macros

```
#define min(i, j) (((i) < (j)) ? (i) : (j))
```

vs.

```
template<class T> T min (T i, T j) { return ((i < j) ? i : j) }
```

- Here are some problems with macros:
- There is no way for the compiler to verify that the macro parameters are of compatible types. The macro is expanded without any special type checking.
- The i and j parameters are evaluated twice. For example, if either parameter has a post-incremented variable (e.g.: `min(i++, j)`), the increment is performed two times.
- Because macros are expanded by the preprocessor, compiler error messages will refer to the expanded macro, rather than the macro definition itself. Also, the macro will show up in expanded form during debugging.

Templates vs. void pointers

- Many functions that are now implemented with void pointers can be implemented with templates.
  Void pointers are often used to allow functions to operate on data of an unknown type. When using void pointers, the compiler cannot distinguish types, so it cannot perform type checking or type-specific behavior such as using type-specific operators, operator overloading, or constructors and destructors.

Templates vs. void pointers - cont.

- With templates, we can create functions and classes that operate on typed data. The type looks abstracted in the template definition. However, at compile time the compiler creates a separate version of the function for each specified type. This enables the compiler to treat class and function templates as if they acted on specific types. Templates can also improve coding clarity, because you don't need to create special cases for complex types such as structures.

Credits

- These slides are (heavily) based on the material of:
  - Dr. Ian Richards, CSC2402, Univ. of Southern Queensland
  - Prof. Paolo Frasconi, IIN 167, Univ. di Firenze
  - S.P. Adam, University of Athens
  - Junaed Sattar, McGill University