Programmazione

Prof. Marco Bertini
marco.bertini@unifi.it
http://www.micc.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.

What is generic programming ?

Static type checking means that type checking occurs at compile time. No type information is used at runtime in that case.
Dynamic type checking occurs when type information is used at runtime.

- 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 compilation
    - 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 value, 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<typename 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 type as argument
  - just like regular function parameters, but pass types to a function

Function templates declaration format

- `template <typename identifier> function_declaration;`
- `template <class 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
  std::cout << k << std::endl;
  std::cout << n << std::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
  std::cout << k << std::endl;
  std::cout << n << std::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 <typename 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 <typename T>
class mypair {
    T values [2];
public:
  mypair(T first, T second) {
    values[0]=first;
    values[1]=second;
  }
  T getMax();
};
```

```
template <typename 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;
}
```

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.

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 class template:

```
// Client
GList<int>
GList<floa
GList<stri

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;
```

Instantiating a class template - cont.

- It is possible to specialize a template with another template:

```
template<typename T>
class Array {
  // . . .
};

Array<Array<int>> aai;
```

Instantiating a class template - cont.

- It is possible to specialize a template with another template:

```
template<typename T>
class Array {
  // . . .
};

Array<Array<int>> aai;
```

Before C++11 you had to have a space between the >>:

```
> >
```

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

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

Default parameters

If a template has default values for all its parameters it can be instantiated without using any parameter:

```
template <typename T=int>
class Foo {
};

Foo<float> foo1;
Foo<double> foo2;
Foo<> foo3; // it's a Foo<int>
```

Non-type parameters

- Templates can also have regular typed parameters, similar to those found in functions. They can have default parameters.

- ```
  template <typename T, int N>
  class MySequence {
      T memblock [N];
    public:
      void setMember (int x, T value);
      T getMember (int x);
  };
  ```
  ```

Non-type parameters - cont.

- ```
  template <typename 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 <typename 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
  }
};
```

About typename

Qualified and unqualified names

- A qualified name is one that specifies a scope. For instance, the following references to `cout` and `endl` are qualified names:

```
std::cout << "Hello world!" <<
std::endl;
```

- The following instances instead are unqualified:

```
using namespace std;
cout << "Hello world!" << endl:
```

Dependent and non-dependent names

- A dependent name is a name that depends on a template parameter:

```
template <class T>
class MyClass {
  int i;              // non-dependent name
  vector<int> vi; // non-dependent name

  T t;              // dependent name
  vector<T> vt;  // dependent name
  typedef T another_name_for_T;
  another_name_for_T u; // dependent name
};
```

Disambiguating dependent qualified type names

- Let us consider the following code:

```
template <typename T>
void foo() {
    T::iterator * iter; // qualified and
                        // dependent name
}
```

- What did the programmer intend this bit of code to do? Is there a static `iterator` member in class T or is `T::iterator` a type ?
  - In the first case we are multiplying two variables,
  - In the second case we want a pointer to T::iterator type…
- The compiler can not understand the intent: we must specify.

Solution: typename

- The `typename` keyword tells the compiler to interpret a particular name as a type.
- Assuming the programmer intended line 3 as a declaration, they would have to write:

```
template <typename T>
void foo() {
   typename T::iterator * iter;
  / / ...
}
```

- Without typename, there is a C++ parsing rule that says that **qualified dependent** names should be parsed as non-types even if it leads to a syntax error, e.g.:

```
T::iterator iter; // syntax error...
```

Solution: typename

- The `typename` keyword tells the compiler to interpret a particular name as a type.
- Assuming the programmer intended line 3 as a declaration, they would have to write:

```
template <typename T>
void foo() {
   typename T::iterator * iter;
  / / ...
}
```

- Simple rule: if your type is a qualified name that involves a template argument, you must use `typename`.

Class template

A complete example

```
#ifndef TSTACK_H
#define TSTACK_H

template<typename T>
class Stack {
public:
    Stack(int n = 10);
    ~Stack() {
        delete[] stackPtr;
    }
    bool push(const T& pushValue);
    bool pop(T& popValue);

private:
    int size;
    int top;
    T* stackPtr;

    bool isEmpty() const {
        return(top == -1);
    }
    bool isFull() const {
        return(top == (size-1));
    }
};
```

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

```
// Constructor with default size 10
template<typename T>
Stack<T>::Stack(int n) {
    size = n > 0 ? n : 10;
    top = -1; // stack is initialy empty
    stackPtr = new T[size]; // allocate space for stack elements
}

// push an element on the stack
// return true if successful, false if stack is full
template<typename T>
bool Stack<T>::push(const T& pushValue) {
    if(!isFull()) {
        stackPtr[ ++top ] = pushValue; // update top and set value
        return true; // push successfull
    }
    return false; // push unsuccessfull
}

template<typename T>
bool Stack<T>::pop(T& popValue) {
    if(!isEmpty()) {
        popValue = stackPtr[ top-- ]; // get value and update top
        return true;// pop successful
    }
    return false; // pop unsuccessful
}
#endif // TSTACK_H
```

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

```
#include <iostream>
#include "TStack.h"

int main(int argc, char *argv[]) {
    Stack<double> doubleStack(5);
    double f = 3.14;

    std::cout << "Pushing elements on stack... ";
    while( doubleStack.push(f) ) {
        std::cout << f << " ";
        f *= 2;
    }
    std::cout << " . " << std::endl;

    std::cout << "Popping elements from stack... ";
    while( doubleStack.pop(f) ) {
        std::cout << f << " ";
    }
    std::cout << " . " << std::endl;
```

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

```
std::cout << "Popping elements from stack... ";
    while( doubleStack.pop(f) ) {
        std::cout << f << " ";
    }
    std::cout << " . " << std::endl;

    Stack<int> intStack;
    int i = 1;

    std::cout << "Pushing elements on stack... ";
    while( intStack.push(i) ) {
        std::cout << i << " ";
        i *= 2;
    }
    std::cout << " . " << std::endl;

    std::cout << "Popping elements from stack... ";
    while( intStack.pop(i) ) {
        std::cout << i << " ";
    }
    std::cout << " . " << std::endl;

}
```

- 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> class X {...
.. // some code within a class template..
.. T t1, t2; // implies existence of default c'tor
.. t1 + t2   // implies a sum 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 and functions are coded in the header file: the compiler needs both declaration and definition to produce the specialization.
  - Very few compilers supported 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++11 standard has deprecated `export`, so just write your templates in a header file
  - ...templates can be seen as advanced textual substitution.

Templates vs. other techniques

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 implemented with void pointers can be implemented with templates instead.
- 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.

Issues and solutions

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( ... );
 }
 // ...
};
```

Ten sses - 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) {
   //...
   }
   //...
};
```

Templates and base classes - cont.

- Still it's not enough: in `LoggingMsgSender` we have to tell the compiler to look at the `MsgSender` 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.

Reducing code bloat in C++11

- The fact that instantiations are generated when a template is used means that the same instantiation may appear in multiple object files: if two or more separately compiled source files use the same template with the same template arguments, there is an instantiation of that template in each of those files !
- C++11 has introduced "extern templates" to avoid this

extern templates (C++11)

- extern templates (a.k.a. explicit instantiation) is based on the keyword extern:

```
extern template declaration;
// instantiation declaration
```

where declaration is a class or function declaration in which all the template parameters are replaced by the template arguments.

extern templates (C++11)

- When the compiler sees an `extern` template declaration, it will not generate code for that instantiation in that file.
- Declaring an instantiation as extern is a promise that there will be a **nonextern** use of that instantiation elsewhere in the program.
- There may be several extern declarations for a given instantiation but there must be exactly one definition for that instantiation.
- Because the compiler automatically instantiates a template when we use it, the extern declaration must appear before any code that uses that instantiation.

extern templates: example

- Let us have:

```
template <typename T>
int compare(const T&, const T&);
template <typename T> class Blob;
```

-

```
// Application.cpp
// these template types must be instantiated elsewhere in the program
extern template class Blob<string>;
extern template int compare(const int&, const int&);

Blob<string> sa1, sa2; // instantiation will appear elsewhere
Blob<int> a1(10);
// Blob<int> and its constructor instantiated in this file
Blob<int> a2(a1); // copy constructor instantiated in this file
int i = compare(a1[0], a2[0]); // instantiation will appear elsewhere
```

The file Application.o will contain instantiations for `Blob<int>`, along with the constructor and copy constructors for that class. The `compare<int>` function and `Blob<string>` class **will not** be instantiated in that file. There must be definitions of these templates in some other file in the program.

- ```
  // Application.cpp
  // these template types must be instantiated elsewhere in the program
  extern template class Blob<string>;
  extern template int compare(const int&, const int&);

  Blob<string> sa1, sa2; // instantiation will appear elsewhere
  Blob<int> a1(10);
  // Blob<int> and its constructor instantiated in this file
  Blob<int> a2(a1); // copy constructor instantiated in this file
  int i = compare(a1[0], a2[0]); // instantiation will appear elsewhere
  ```
  ```

extern templates: example

- ```
  // templateBuild.cc
  /* instantiation file must provide a
  (nonextern) definition for every type and
  function that other files declare as
  extern */
  template int compare(const int&, const
  int&);
  template class Blob<string>;
  // instantiates all members of
  // the class template
  ```
  ```

extern templates: example

When the compiler sees an instantiation definition (as opposed to a declaration), it generates code.
Thus, the file `templateBuild.o` will contain the definitions for `compare` instantiated with `int` and for the `Blob<string>` class.
When we build the application, we must link `templateBuild.o` with the `Application.o` files.
There must be an explicit instantiation definition somewhere in the program for every instantiation declaration.

Instantiation definition

- An instantiation definition for a class template instantiates all the members of that template including inline member functions: the compiler cannot know which member functions the program uses, thus instantiates all the members of that class, unlike the ordinary class template instantiations.
- Even if we do not use a member, that member will be instantiated. Consequently, we can use explicit instantiation only for types that can be used with all the members of that template.

Reading material

- M. Bertini, "Programmazione Object-Oriented in C++", cap. 6 - pp. 129-142
- B. Stroustrup, "C++, Linguaggio, libreria standard, principi di programmazione", cap. 23, 27
- B. Stroustrup, "C++, guida essenziale per programmatori" - cap. 5
- L.J. Aguilar, "Fondamenti di programmazione in C++. Algoritmi, strutture dati e oggetti" - cap. 21
- Thinking in C++, 2nd ed. Volume 1, cap. 16

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