301AA - Advanced Programming
[AP-2017]

Lecturer: Andrea Corradini andrea@di.unipi.it
Tutor: Lillo Galletta galletta@di.unipi.it

Department of Computer Science, Pisa
Academic Year 2017/18

AP-2017-13: Parametric Polymorphisms
Outline

• Universal parametric polymorphism (generics)
• C++ templates
• Templates vs Macros in C++
• Specialization and instantiation of templates
• The Standard Template Library: an overview
• Java Generics
• Bounded type parameters
• Generics and inheritance: invariance, covariance and contravariance
Classification of Polymorphism

- Polymorphism
  - Universal
    - Coercion
    - Parametric
  - Ad hoc
    - Overloading
  - Overriding
    - Explicit
    - Implicit
      - Bounded
        - Covariant
        - Invariant
        - Contravariant
Parametric polymorphism, or generic programming

- [C++] **Templates**, since ~1990
  - Function and class templates; type variables
  - Each concrete instantiation produces a copy of the generic code, specialized for that type

- [Java] **Generics**, since Java 1.5 (Java 5, 2004)
  - Generic methods and classes; type variables
  - Strongly type checked by the compiler
  - Type erasure: type variables are `Object` at runtime
Function Templates in C++

• Support parametric polymorphism
• Type parameters can also be primitive types (unlike Java generics)
• Example of polymorphic square function:

```cpp
template <class T> // or <typename T>
T sqr(T x) { return x * x; }
```

• Compiler/linker automatically generates one version for each parameter type used by a program
• Parameter types are inferred or indicated explicitly (necessary in case of ambiguity)
Function Templates: \texttt{sqr}

\begin{verbatim}

template<class T> // or <typename T>
T sqr(T x) { return x * x; }

int a = 3;
double b = 3.14;
int aa = sqr(a);
double bb = sqr(b); // also sqr<double>(b)

Generates
int sqr(int x){return x*x;}

Generates
double sqr(double x){return x*x;}

\end{verbatim}
Function Templates: `sqr`

- Works for user-defined types as well

```cpp
class Complex {
public:
    double real; double imag;
    Complex(double r, double im): real(r), imag(im) {};
    Complex operator*(Complex y) {
        return Complex(
            real * y.real - imag * y.imag,
            real * y.imag + imag * y.real);
    }
};

{ ...
    Complex c(2, 2);
    Complex cc = sqr(c);
    cout << cc.real << " " << c.imag << endl;
    ...
}"
Function Templates and Type Inference: GetMax

```cpp
template <class T>
T GetMax (T a, T b) {
    T result;
    result = (a>b)? a : b;
    return (result);
}

{{...
    int i = 5, j = 6, k;
    long l = 10, m = 5, n, v;
    k = GetMax<int>(i, j); //ok
    n = GetMax(l, m);       //ok: GetMax<long>
    // v = GetMax(i, m);    //no: ambiguous
    v = GetMax<int>(i,m);   //ok
...}}
```

- Decoupling the two arguments:
  ```cpp
template <class T, class U>
T GetMax (T a, U b) {
    return (a>b)? a : b;
}
```
Templates vs Macros in C++

• Macros can be used for polymorphism in simple cases

```cpp
#define SQR(T) T sqr(T x) {return x * x; }
SQR(int);
SQR(double);

{ int a = 3, aa; double b = 3.14, bb;
  aa = sqr(a);
  bb = sqr(b);
  ...
}
```

• Macros are executed by the preprocessor, templates by the compiler

• Macro expansion visible compiling with opition `--E`

• Preprocessor makes only (possibly parametric) textual substitution. No parsing, no static analysis check.
Macros’ limits

#define sqr(x) ((x) * (x)) // problem?

— Code is copied: side effects duplicated

int aa = sqr(a++); // equivalent to
    // int aa = ((a++) * (a++));
int aaa = sqr(aa); // what does it print?

#define fact(n) (n == 0) ? 1 : fact(n-1) * n
    // problem?

— Recursion not possible
More on C++ templates

• Specialization of templates
• Instantiation and Overloading resolution
• Partial support for “separate compilation”
• The Standard Template Library
  – Efficiency
Template (partial) specialization

A (function or class) template can be *specialized* by defining a template with
• same name
• more specific parameters (partial specialization) or no parameter (full specialization)

Advantages
• Use better implementation for specific kinds of types
• Intuition: similar to *overriding*
• Compiler chooses most specific applicable template
Template specialization, example

/* Primary template */
template <typename T> class Set {
// Use a binary tree
}

/* Full specialization */
template <> class Set<char> {
// Use a bit vector
}

/* Partial specialization */
template <typename T> class Set<T*> {
// Use a hash table
}
Need of template specialization, an example

```cpp
// Full specialization of GetMax for char*
template <>
const char* GetMax(const char* a, const char* b)
{  return strcmp(a, b) > 0 ? a : b ;  }
```

template <class T>
T GetMax(T a, T b)
{  return a > b ? a : b ;}

```cpp
int main()
{
  cout << "max(10, 15) = " << GetMax(10, 15) << endl ;
  cout << "max('k', 's') = " << GetMax('k', 's') << endl ;
  cout << "max(10.1, 15.2) = " << GetMax(10.1, 15.2) << endl ;
  cout << "max("Joe","Al") = " << GetMax("Joe", "Al") << endl ;
  return 0 ;
}
```

Output:
max(10, 15) = 15
max('k', 's') = s
max(10.1, 15.2) = 15.2
max("Joe","Al") = Al //not expected

Output of main with specialization:
max(10, 15) = 15
max('k', 's') = s
max(10.1, 15.2) = 15.2
max("Joe","Al") = Joe
C++ Template implementation

• Compile-time instantiation
  – Compiler chooses template that is best match
    • Based on partial (specialization) order of matching templates
    • There can be more than one applicable template
  – Template instance is created
    • Similar to syntactic substitution of parameters
    • Can be done after parsing, etc., thus language-aware (unlike the pre-processor)
  – Overloading resolution after substitution
    • Fails if some operator is not defined for the type instance
    • Example: if T does not implement < in previous slide
On instantiation

• The compiler need both the *declaration* and the *definition* of the template function to instantiate it.
• Limited forms of “separate compilation”: cannot compile *definition* of template and code instantiating the template separately.
• If the same template function definition is included in different source files, separately compiled and linked, there will be only one instantiation per type of template function
• Explicit instantiation possible. Example:
  ```cpp
  template int GetMax<int>(int a, int b);
  ```
Standard Template Library for C++

• Goal: represent algorithms in as general form as possible without compromising efficiency
• Extensive use of templates and overloading
• Only uses static binding (and inlining): not object oriented, no dynamic binding – very different from Java Collection Framework
• Use of iterators for decoupling algorithms from containers
• Iterator: abstraction of pointers
• Many generic abstractions
  – Polymorphic abstract types and operations
• Excellent example of generic programming
  – Generated code is very efficient
Main entities in STL

- **Container**: Collection of typed objects
  - Examples: array, vector, deque, list, set, map...
- **Iterator**: Generalization of pointer or address. Used to step through the elements of collections
  - forward_iterator, reverse_iterator, istream_iterator, ...
  - Pointer arithmetic supported
- **Algorithm**: initialization, sorting, searching, and transforming of the contents of containers,
  - for_each, find, transform, sort
- **Adapter**: Convert from one form to another
  - Example: produce iterator from updatable container
- **Function object**: Form of closure
  - plus, equal, logical_and
- **Allocator**: encapsulation of a memory pool
  - Example: GC memory, ref count memory, ...
1. Templates
   make algorithms independent of the data types
2. Iterators
   make algorithms independent of the containers
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> vec; // create a vector to store int
    int i;
    // display the original size of vec
    cout << "vector size = " << vec.size() << endl;
    // push 5 values into the vector
    for(i = 0; i < 5; i++) {
        vec.push_back(i);
    }
    // display extended size of vec
    cout << "extended vector size = " << vec.size() << endl;
    // access 5 values from the vector
    for(i = 0; i < 5; i++) {
        cout << "value of vec [" << i << "] = " << vec[i] << endl;
    }
    // use iterator to access the values
    vector<int>::iterator v = vec.begin();
    while( v != vec.end() ) {
        cout << "value of v = " << *v << endl;
        v++;
    }
    return 0;
}
int main(int argc, char **argv) {
    vector<string> args(argv, argv + argc);

    vector<string>::iterator iter = args.begin();
    for (; iter != args.end(); ++iter )
        cout << *iter << " ";
    cout << endl;

    vector<string>::reverse_iterator iter = args.rbegin();
    for (; iter != args.rend(); ++iter )
        cout << *iter << " ";
    cout << endl;
}

Forward and Reverse Iterator
Example of STL approach

• Function to merge two sorted lists
  – merge : range(s) \times range(t) \times \text{comparison}(u)
    \rightarrow \text{range}(u)
  
  This is conceptually right, but not STL syntax

• Basic concepts used
  – range(s) - ordered “list” of elements of type s, given by pointers to first and last elements
  – \text{comparison}(u) - boolean-valued function on type u
  – \text{subtyping} - s and t must be subtypes of u
How merge appears in STL

• Ranges represented by pairs of iterators
  – Pointing to first and last+1 element
• Comparison operator is object of class Compare
• Polymorphism expressed using template

```cpp
template <class InputIterator1, class InputIterator2, 
class OutputIterator, class Compare>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1, 
InputIterator2 first2, InputIterator2 last2, 
OutputIterator result, Compare comp)
```
Comparing STL with other libraries

- C:
  ```c
  qsort( (void*)v, N, sizeof(v[0]), compare_int );
  ```
- C++, using raw C arrays:
  ```c
  int v[N];
  sort( v, v+N );
  ```
- C++, using a vector class:
  ```c
  vector v(N);
  sort( v.begin(), v.end() );
  ```
Efficiency of STL

• Running time for sort

<table>
<thead>
<tr>
<th></th>
<th>N = 50000</th>
<th>N = 500000</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1.4215</td>
<td>18.166</td>
</tr>
<tr>
<td>C++ (raw arrays)</td>
<td>0.2895</td>
<td>3.844</td>
</tr>
<tr>
<td>C++ (vector class)</td>
<td>0.2735</td>
<td>3.802</td>
</tr>
</tbody>
</table>

• Main point
  – Generic abstractions can be convenient and efficient!
  – But watch out for code size if using C++ templates...
An example of STL algorithm: Inner Product

Definition: Computes inner product (i.e. sum of products) or performs ordered map/reduce operation on the range [first1, last1) and the range beginning at first2.

```cpp
template< class InputIt1, class InputIt2, class T >
T inner_product( InputIt1 first1, InputIt1 last1,
                 InputIt2 first2, T value );
```

Use:

```cpp
vector<unsigned> ia1 = {1, 2, 3, 4, 5, 6, 7};
vector<unsigned> ia2 = {7, 6, 5, 4, 3, 2, 1};

// computes the dot product
inner_product(ia1.begin(), ia1.end(), ia2.begin(), 0);
```
inner_product: more general definition

```cpp
T inner_product(InputIterator first1,
                 InputIterator last1, InputIterator first2,
                 T init, BinaryFunction op1, BinaryFunction op2);
```

- Ordered map/reduce
- Initializes result to init
- For each \( i_1 \) in \([\text{first1}, \text{last1})\), and \( i_2 = \text{first2} + (i_1 - \text{first1}) \) updates result as follows:

  \[
  \text{result} = \text{op1}(\text{result}, \text{op2}(\ast i_1, \ast i_2))
  \]
Inner Product (string concat)

class Join {
    public:
        Join(string const &sep): sep(sep) {}  
        string operator()(string const & s1, string const & s2) {
            return s1 + sep + s2; }
    private:
        string sep;
};

vector<string> names1= {"Frank", "Karel", "Piet"};
vector<string> names2 = {"Brokken", "Kubat", "Plomp"};

inner_product(names.begin(), names1.end(), names2.begin(),
    "\t", Join("\n"), Join(" ")) ;
Classification of Polymorphism

- Polymorphism
  - Universal
    - Parametric
      - Implicit
    - Inclusion
      - Explicit
        - Bounded
          - Covariant
          - Invariant
          - Contravariant
    - Coercion
      - Implicit
  - Ad hoc
    - Overriding
  - Overloading
Java Generics
Explicit Parametric Polymorphism

• Classes, Interfaces, Methods can have type parameters
• The type parameters can be used arbitrarily in the definition
• They can be instantiated by providing arbitrary (reference) type arguments
• We discuss only a few issues about Java generics...

interface List<E> {
    boolean add(E n);
    E get(int index);
}

List<Integer>
List<Number>
List<String>
List<List<String>>
...

Tutorial on Java generics:
https://docs.oracle.com/javase/tutorial/java/generics/index.html
Generic methods

• Methods can use the type parameters of the class where they are defined, if any
• They can also introduce their own type parameters
• Invocations of generic methods must instantiate all type parameters, either explicitly or implicitly
  – A form of type inference
Bounded Type Parameters

interface List<E extends Number> {
    void m(E arg) {
        arg.asInt();  // OK, Number and its subtypes
        // support asInt()
    }
}

• Only classes implementing Number can be used as type arguments
• Method defined in the bound (Number) can be invoked on objects of the type parameter
Type Bounds

\[
\langle \text{TypeVar extends SuperType}\rangle
\]

\- upper bound; SuperType and any of its subtype are ok.

\[
\langle \text{TypeVar extends ClassA \& InterfaceB \& InterfaceC \& …}\rangle
\]

\- Multiple upper bounds

\[
\langle \text{TypeVar super SubType}\rangle
\]

\- lower bound; SubType and any of its supertype are ok

\- Type bounds for methods guarantee that the type argument supports the operations used in the method body

\- Unlike C++ where overloading is resolved and can fail after instantiating a template, in Java type checking ensures that overloading will succeed
A generic algorithm with type bounds

```java
public static <T> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e > elem) // compiler error
            ++count;
    return count;
}
```

```java
public interface Comparable<T> {
    // classes implementing
    public int compareTo(T o); // Comparable provide a
} // default way to compare their objects
```

```java
public static <T extends Comparable<T>>
    int countGreaterThan(T[] anArray, T elem) {
        int count = 0;
        for (T e : anArray)
            if (e.compareTo(elem) > 0) // ok, it compiles
                ++count;
        return count;
    }
```
Generics and subtyping

- Integer is subtype of Number
- Is List<Integer> subtype of List<Number>?
- NO!
What are Java rules?

If \texttt{Type2} and \texttt{Type3} are different, then \texttt{Type1<Type2>} is not a subtype of \texttt{Type1<Type3>}

• Formally: subtyping in Java is \textit{invariant} for generic classes.
In both directions the Substitution Principle is not satisfied, thus List<Number> is neither a supertype nor a subtype of List<Integer>: Java rules are adequate here.
But in more specific situations...

interface List<T> {
    T get(int index);
}

type List<Number>:
    Number get(int index);

type List<Integer>:
    Integer get(int index);

A covariant notion of subtyping would be safe:
   - List<Integer> can be subtype of List<Number>
   - Not in Java

• In general: covariance is safe if the type is read-only
Viceversa... contravariance!

interface List<T> {
    boolean add(T elt);
}

type List<Number>:
    boolean add(Number elt);

type List<Integer>:
    boolean add(Integer elt);

A *contravariant* notion of subtyping would be safe:

- List<Number> can be a subtype of List<Integer>
- But Java ..... 

In general: *contravariance* is safe if the type is *write-only*