Generic Programming

Advanced Programming

Haoyuan Li

li@di.unipi.it

Dipartimento di Informatica

Università di Pisa

What is Generic Programming?

- Programming with generic parameters
  — *parametric polymorphism*
- Programming by abstracting from concrete types
  — *subtype polymorphism*
- Programming with parameterized components
  — *parameterize components by other components*
- Programming method based in finding the most abstract representation of efficient algorithms
  — *C++ Standard Template Library*
  — *Java Collections Framework*

Terms

- Parametric polymorphism
  — *declare element type as a parameter of a routine, also known as generic parameters*
- Subtype polymorphism
  — *an abstract type is defined and used in a routine implies that the routine works for all subtypes of the abstract type*
- Overloading
  — *provide different implementations of functions for the same function name*

Polymorphism

- *Polymorphism* – “the ability to have many forms”

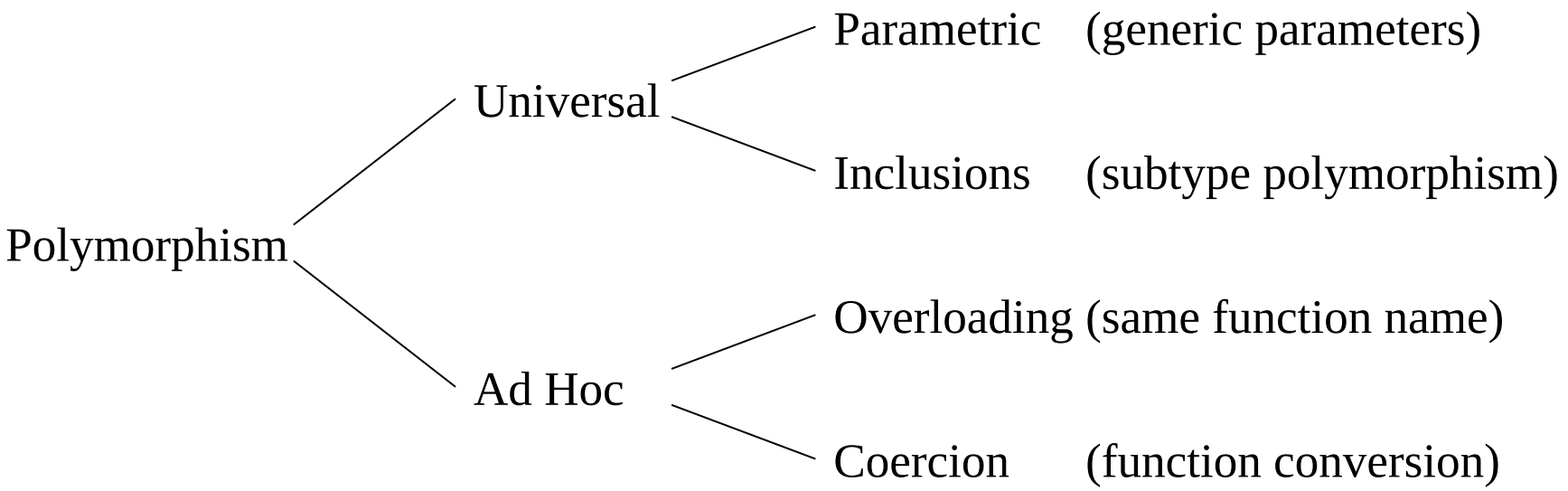

Fig 1. Classification of polymorphism by Caderlli and Wegner (1985).

C++ Templates and Parametric Polymorphism

Generic Parameters

- Generic = Not specific
- Compile-time type checking in statically typed programming languages

```
// With function overloading, e.g., C++
int square(int x) { return x * x; }
double square(double x) { return x * x; }
```

```
/* Without function overloading, e.g., C */
int sqr_int(int x) { return x * x; }
double sqr_double(double x) { return x * x; }
```

C++ Template Class

- Type parameter is required

```
template<class _T>
class Number {
public:
  Number(_T n);
  _T value();
private:
  _T __number;
};

Number<int> *x = new Number<int>(0);
Number<double> *y = new Number<double>(0.0);
Number<Complex> z(Complex(2, 3));
```

C++ Template Function

- Compiler automatically generates a version of function for each parameter type

```
template<class _T>
_T square(_T x) { return x * x; }

int a = 3;
double b = 3.14;
int aa = square(a); // square() for _T = int
double bb = square(b); // square() for _T = double
```

Macro versus Generic Parameters

- Does macro do the same things?

```
#define square_x(x) ((x) * (x))
#define SQR_T(T) T square_t(T x) { return x * x; }
SQR_T(int);     // square_t() for T = int
SQR_T(double);  // square_t() for T = double

int a = 3;
double b = 3.14;

int aa1 = square_x(a);     // aa1 == 9
int aa2 = square_t(a);     // aa2 == 9
double bb1 = square_x(b);  // bb1 == 9.8596
double bb2 = square_t(b);  // bB2 == 9.8596
```

Macro's Limits (1)

- Does macro *always* do the same things?

```
#define SQUARE(x) ((x) * (x))

template<class _T>
_T square(_T x) { return x * x; }

int a = 3, b = 3;
int aa1 = square(a);    // aa1 == 9
int bb1 = SQUARE(b);    // bb1 == 9
int aa2 = square(a++);  // aa2 == ?
int bb2 = SQUARE(b++);  // bb2 == ?
```

Macro's Limits (2)

- Does macro *always* do the same things?

```
#define SQUARE(x) ((x) * (x))

template<class _T>
_T square(_T x) { return x * x; }

int a = 3, b = 3;
int aa1 = square(a);    // aa1 == 9
int bb1 = SQUARE(b);    // bb1 == 9
int aa2 = square(a++);  // aa2 == 9,  a == ?
int bb2 = SQUARE(b++);  // bb2 == 9,  b == ?
```

Macro's Limits (3)

- Does macro *always* do the same things?

```
#define SQUARE(x) ((x) * (x))

template<class _T>
_T square(_T x) { return x * x; }

int a = 3, b = 3;
int aa1 = square(a);    // aa1 == 9
int bb1 = SQUARE(b);    // bb1 == 9
int aa2 = square(a++);  // aa2 == 9,  a == 4
int bb2 = SQUARE(b++);  // bb2 == 9,  b == 5

//  bb2 = ((b++) * (b++));
```

Compile-time Type Checking

- Allow us to ensure proper typing at compile time

```
template<class _T>
void swap(_T &a, _T &b) {
  const _T temp = a;
  a = b;
  b = temp;
}

int a = 3, b = 4;
swap(a, b); // OK.
double c = 5.0;
swap(a, c); // Compilation error.
```

Parameterized Components (1)

- A sorting algorithm

```
template<class _T>
void sort (_T a[], size_t size) {
  for (size_t i = 0; i < size; i++)
    for (size_t j = i + 1; j < size; j++)
      if (a[i] > a[j])
        swap(a[i], a[j]);
}
// ...
int x[] = {1, 2, 5, 4, 3};
sort(x, sizeof(x));
// ...
double y[] = {1.1, 2.2, 5.5, 4.4, 3.3};
sort(y, sizeof(y));
```

Parameterized Components (2)

- Design components

```
template<class _T>
class greater {
public:
  bool operator()(const _T &a, const _T &b) const
  { return a > b; }
};

template<class _T>
class less {
public:
  bool operator()(const _T &a, const _T &b) const
  { return a < b; }
};
```

Parameterized Components (3)

- Parametrize the sorting algorithm

```
template<class _T, class _C>
void sort (_T a[], size_t size, const _C &comp) {
  for (size_t i = 0; i < size; i++)
    for (size_t j = i + 1; j < size; j++)
      if (comp(a[i], a[j]))
        swap(a[i], a[j]);
}
// ...
int x[] = {1, 2, 5, 4, 3};
sort(x, sizeof(x), greater<int>());
// ...
double y[] = {1.1, 2.2, 5.5, 4.4, 3.3};
sort(y, sizeof(y), less<double>());
```

Function Object

- As a parameter in template function declaration

```
void sort (_T a[], size_t size, const _C &comp) {
```

- The object is passed to a template function

```
sort(x, sizeof(x), greater<int>());
```

- The object is called within the template function

```
if (comp(a[i], a[j]))
```

C++ Standard Template Library

- Purpose: to represent algorithms in as *general* a form as possible without compromising their *efficiency*
  — *extensively uses templates*
  — *exclusively uses static binding and inlining*
- Key design aspect: the use of iterators to decouple algorithms from containers
  — *iterator is an abstraction of pointers*

C++ STL Organization

- Containers
  — `deque`, `list`, `map`, `set`, `vector`, ...
- Algorithms
  — `find`, `for_each`, `sort`, `transform`, ...
- Iterators
  — `forward_iterator`, `reverse_iterator`, ...
  — `istream_iterator`, `ostream_iterator`, ...
- Function objects
  — `equal`, `logical_and`, `plus`, `project`, ...
- Allocators

STL Example: Forward Iterator

- Use (forward) `iterator` with a `vector` container

```
#include <iostream>
#include <vector>
using namespace std;

// ...

vector<int> v;
for (int i = 0; i < 10; i++)
  v.push_back(i);

vector<int>::iterator it;
for (it = v.begin(); it != v.end(); it++)
  cout << *it << endl;
```

STL Example: Reverse Iterator

- Use `reverse_iterator` with a `vector` container

```
#include <iostream>
#include <vector>
using namespace std;

// ...

vector<int> v;
for (int i = 0; i < 10; i++)
  v.push_back(i);

vector<int>::reverse_iterator it;
for (it = v.rbegin(); it != v.rend(); it++)
  cout << *it << endl;
```

STL Example: Inner Product

- Compute cumulative inner product of range

```
#include <iostream>
#include <numeric>
using namespace std;

// ...

int a[] = {10, 20, 30};
int b[] = {1, 2, 3};
int init = 100;
cout << inner_product(a, a + 3, b, init) << endl;

// Output 240
// 100 + 1 * 10 + 2 * 20 + 3 * 30
```

Inside STL: Inner Product

- Definition of `inner_product`

```
template<class InputIterator1, class InputIterator2, class T,
         class BinaryOperation1, class BinaryOperation2>
T inner_product(InputIterator1 first1,
                InputIterator1 last1,
                InputIterator2 first2,
                T init,
                BinaryFunction1 binary_op1,  // default is +
                BinaryFunction2 binary_op2)  // default is *
{
  while (first1 != last1)
    init = binary_op1(init, binary_op2(*first1++, *first2++));
  return init;
}

// x = *ptr++ equals { x = *ptr; ptr++; }
```

Advanced Use of Inner Product

- String concat by `inner_product` and function object

```
class cat {
  // ...
};

//...

string first_name[] = {"Frank", "Karel", "Piet"};
string last_name[] = {"Brokken", "Kubat", "Plomp"};
cout << inner_product(first_name, first_name + 3,
  last_name, string("\t"), cat("\n\t"), cat(" "))
  << endl;

// Please complete the class cat.
```

Template Enumeration

- Design STL container enumeration

```
template<class _T>
class vector_enum {
public:
  class enumeration {
    // ...
  };
  enumeration first() {
    return enumeration(&__vector);
  }
private:
  std::vector<_T> __vector;
};
```

Enumerable Vector

- Definition of `vector_enum`

```
template<class _T>
class enumeration {
public:
  enumeration(std::vector<_T> const *v):
    __vector(v), __index(0) {}
  _T const &next()
  { return __vector[__index++]; } // Attention!!!
  bool has_next()
  { return __index < __vector->size(); }
private:
  size_t __index;
  std::vector<_T> __vector;
};
```

Use Template Enumeration

- How to use enumerable vector?

```
// ...

vector_enum<int> v;

for (size_t i = 0; i < 10; i++)
  v.push_back(i);

vector_enum<int>::enumeration en = v.first();
while (en.has_next())
  cout << en.next() << endl;

// ...
```

C++ Template Instantiation (1)

- Separately declare and define a function

```
// sum.h
extern int sum(int *, unsigned int);
```

```
// sum.cpp
#include "sum.h"
int sum(int *array, unsigned int n) {
  int x = 0;
  for (unsigned int i = 0; i < n; i++)
    x += array[i];
  return x;
}
```

C++ Template Instantiation (2)

- And then be separately compiled and linked

```
// sumtest.cpp
#include "sum.h"
// ...
int x[] = {2, 3};
cout << sum(x, 2) << endl;
// ...
```

```
% g++ -c sum.cpp                    # generate sum.o
% g++ -o sumtest sum.o sumtest      # generate sumtest
```

C++ Template Instantiation (3)

- Separately declare and define a template function

```
// sum.h
template<class _T>
extern _T sum(_T *, unsigned int);
```

```
// sum.cpp
#include "sum.h"
template<class _T>
_T sum(_T *array, unsigned int n) {
  _T x(0);
  for (unsigned int i = 0; i < n; i++)
    x += array[i];
  return x;
}
```

C++ Template Instantiation (4)

- Cannot be separately complaied and linked

```
% g++ -c sum.cpp                      # generate sum.o
% g++ -o sumtest sum.o sumtest  # linking error!
Undefined symbols:
  "int sum<int>(int*, unsigned int)", referenced
from:
      _main in cc2qcRGC.o
ld: symbol(s) not found
collect2: ld returned 1 exit status
```

- Template function has not been instantiated
- Solution: include template function definition

C++ Template Instantiation (5)

- Explicit instantiations

```
// sum.cpp
#include "sum.h"

template<class _T>
_T sum(_T *array, unsigned int n) {
  // ...
}

template
int sum<int>(int *, unsigned int);

template
double sum<double>(double *, unsigned int);
```

Parametric Polymorphism in C++

- Templates correspond to parametric polymorphism

```
class A {
  public: void hello() { cout << "A" << endl; }
};

class B {
  public: void hello() { cout << "B" << endl; }
};

template<class _T>
void test(const _T &v) { v.hello(); }
```

Subtype Polymorphism in C++

- A quite problematic for a statically typed language

```
class HelloClass {
  public: virtual void hello() = 0;
};

class A: public HelloClass {
  public: void hello() { cout << "A" << endl; }
};

class B: public HelloClass {
  public: void hello() { cout << "B" << endl; }
};

void test(const HelloClass &v) { v.hello(); }
```

Java Generics and Bounded Polymorphism

Polymorphism in Java

- Before JDK 1.5, Java provides only a limited variety of polymorphism
  — *supports subtype polymorphism*
  — *discards parametric polymorphism*
  — *uses run-time type identification (RTTI)*
- Parametric polymorphism appears since JDK 1.5 (also known as J2SE 5.0, released in Sept. 2004)
  — *uses compile-time type checking*
  — *supports bounded polymorphism*
  — *supports static methods*
  — *excludes primitive types*

Evolution of Java Generics

- Before JDK 1.5, subtype polymorphism only

```
List intList = new LinkedList();
intList.add(new Integer(0));
Integer x = (Integer) intList.iterator().next();
```

- Generic code since JDK 1.5

```
List<Integer> intList = new LinkedList<Integer>();
intList.add(new Integer(0));
Integer x = intList.iterator().next();
```

- Primary motivation is to avoid runtime exception

Runtime Exception

- Raise runtime class cast exception

```
List intList = new LinkedList();
intList.add(new Double(0.0));

// Raise java.lang.ClassCastException
Integer x = (Integer) intList.get(0);
```

- Compile-time checking in generic code

```
List<Integer> intList = new LinkedList<Integer>();

// Compile-time error
intList.add(new Double(0.0));
```

Defining Java Generics

- Definitions of the interfaces `List` and `Iterator`

```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();
}
```

```
public interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

Generics and Subtyping

- Is the following code snippet legal?

```
String str = "Hello world!";
Object obj = str;
```

- Is the following code snippet legal?

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;
```

- What happens?

```
lo.add(new Object());
String s = ls.get(0);
```

Wildcards (1)

- Without generics

```
void printCollection(Collection c) {
    Iterator i = c.iterator();
    for (k = 0; k < c.size(); k++) {
        System.out.println(i.next());
    }
}
```

- With generics: a naive attempt

```
void printCollection(Collection<Object> c) {
    for (Object e : c)
        System.out.println(e);
}
```

Wildcards (2)

- `Collection<Object>` is *not* a super-type of all kinds of collections!
- The super-type of all kinds of collections is written `Collection<?>`: *collection of unknown*

```
void printCollection(Collection<?> c) {
    for (Object e : c)
        System.out.println(e);
}
```

Wildcards (3)

- Wildcard type is always safe

```
Collection<?> c = new ArrayList<String>();
c.add(new Object()); // compile-time error
```

- We don't know what the element type of `c` stands for
- Any parameter we pass to add would have to be a subtype of this unknown type
- Since we don't know what type that is, we cannot pass anything in
- So that `c` is read-only

Bounded Wildcards (1)

- Consider a drawing application

```
public abstract class Shape {
    public abstract void draw(Canvas c);
}

public class Circle extends Shape {
    private int x, y, radius;
    public void draw(Canvas c) { ... }
}

public class Rectangle extends Shape {
    private int x, y, width, height;
    public void draw(Canvas c) { ... }
}
```

Bounded Wildcards (2)

- All shapes can be drawn on a canvas

```
public class Canvas {
    public void draw(Shape s) {
        s.draw(this);
    }
}
```

- A drawing will typically contain a number of shapes

```
public void drawAll(List<Shape> shapes) {
    for (Shape s: shapes) {
        s.draw(this);
    }
}
```

Bounded Wildcards (3)

- Type rule constraints that `drawAll()` can only be called on lists of exactly `Shape`

```
public void drawAll(List<Shape> shapes) { ... }
```

- So `drawAll()` cannot be called on a list of `Circle` (i.e., `List<Circle>`)
- How can `drawAll()` be called on a `List<Circle>`?
- How to let the method to accept a list of *any* kind of shape?

Bounded Wildcards (4)

- Bound the wildcard: replace the type `List<Shape>` with `List<? extends Shape>`

```
public void drawAll(List<? extends Shape>
                    shapes) {
    // ...
}
```

- Now `drawAll()` will accept lists of any subclass of `Shape`
- The `?` stands for an unknown subtype of `Shape`
- The type `Shape` is the *upper bound* of the wildcard

Bounded Wildcards (5)

- There is a price to be paid for the flexibility of using wildcards

```
public void addCircle(List<? extends Shape>
                          shapes) {
    shapes.add(new Circle());
}
```

- What will happen? Why?

When to Use Wildcards?

- Wildcards or not, it's a question...

```
interface Collection<E> {
    public boolean
            containsAll(Collection<?> c);
    public boolean
            addAll(Collection<? extends E> c);
}
```

```
interface Collection<E> {
    public <T> boolean
            containsAll(Collection<T> c);
    public <T extends E> boolean
            addAll(Collection<T> c);
}
```

More Fun with Wildcards

- Write-only: using *lower bounded wildcards*

```
interface Sink<T> { ... }
public static <T>
    T writeAll(Collection<T> c, Sink<T> s) { ... }
Sink<Object> s;
Collection<String> cs;
String str = writeAll(cs, s);
// ...
public static <T>
    T writeAll(Collection<? extends T> c,
               Sink<T> s) { ... }
public static <T>
    T writeAll(Collection<T> c,
               Sink<? super T> s) { ... }
```