301AA - Advanced Programming

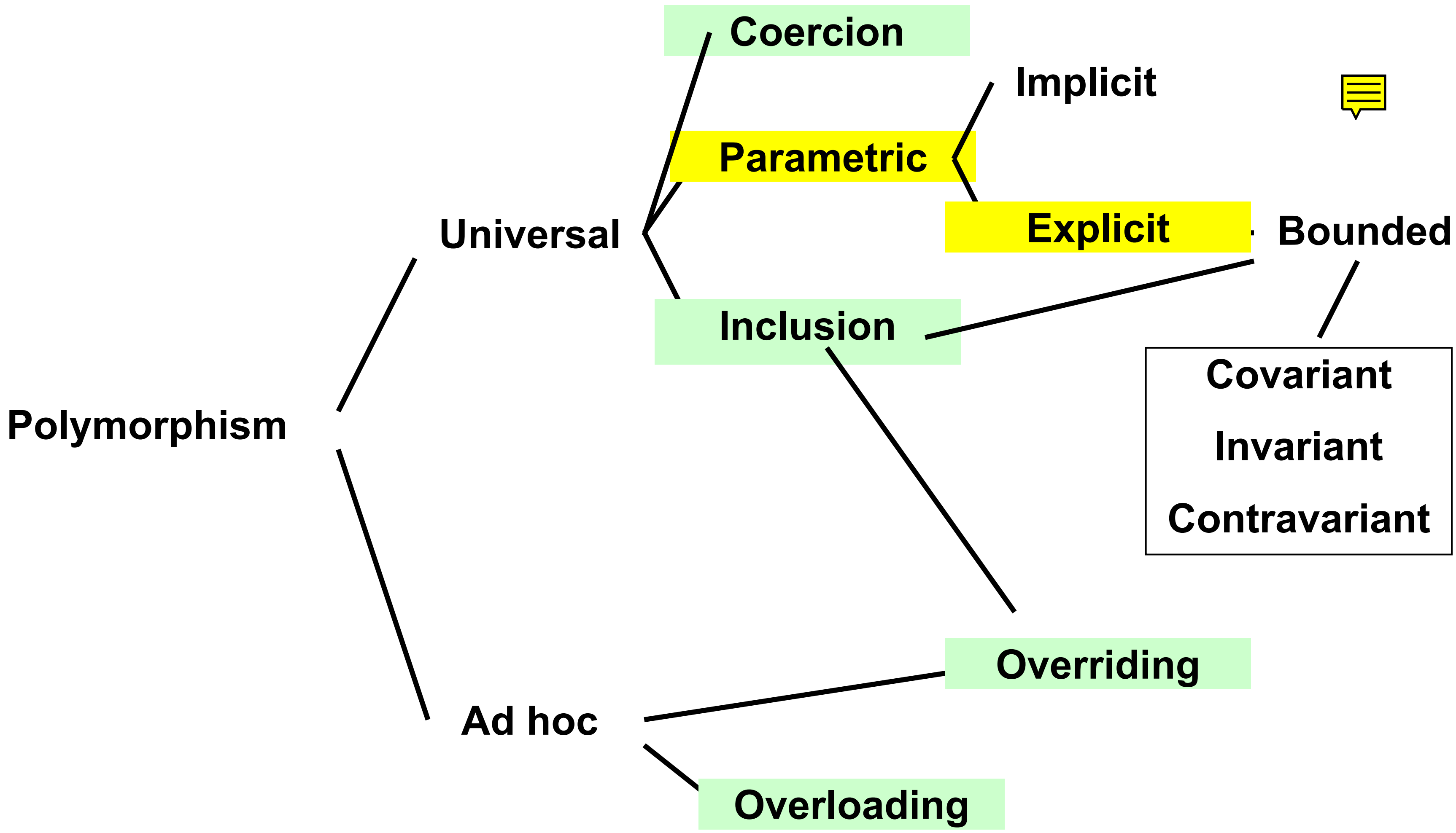
Lecturer: **Andrea Corradini**
andrea@di.unipi.it
http://pages.di.unipi.it/corradini/

AP-13: *Parametric Polymorphisms:*
C++ Templates

Outline

- Universal parametric polymorphism (generics)
- C++ templates
- Templates vs Macros in C++
- Specialization and instantiation of templates

Classification of Polymorphism



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:

```
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)
- The site **Compiler Explorer** https://godbolt.org/ allows to inspect the compiled code.

Function Templates: `sqr`

```
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;}
```

Function Templates: `sqr`

- Works for user-defined types as well

```
class Complex {
public:
    double real; double imag;
    Complex(double r, double im): real(r),imag(im){};
    Complex operator*(Complex y) {  //overloading of *
        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`

```
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:

```
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

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

{  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?

int a = 2;                  // equivalent to
int aa = sqr(a++);          // int aa = ((a++) * (a++));
// value of aa?             // aa contains 6
```

– Code is copied: side effects duplicated

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

// compilation fails because fact is not defined
```

– Recursion not possible

Non-type template arguments

- The template parameters can also include expressions of a particular type:

```
template <class T, int N>
T fixed_multiply (T val)
{
  return val * N;
}

int main() {
  std::cout << fixed_multiply<int,2>(10) << '\n';    //  20
  std::cout << fixed_multiply<int,3>(10) << '\n';    //  30
}
```

- the value of template parameters is determined on compile-time
- the second template argument needs to be a constant expression

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 parameters (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

```
// 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 ;}

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(\"Al\",\"Bob\") = " << GetMax("Al", "Bob") << endl ;
    return 0 ;
}
```

Output:	**Output of main with specialization:**
max(10, 15) = 15	max(10, 15) = 15
max('k', 's') = s	max('k', 's') = s
max(10.1, 15.2) = 15.2	max(10.1, 15.2) = 15.2
max("Al","Bob") = Al //not expected	**max("Al","Bob") = Bob**

Template Metaprogramming

- Templates can be used by a compiler to generate temporary source code, which is merged by the compiler with the rest of the source code and then compiled
- It is Turing complete
- Only constant expressions
- No mutable variables
- No support by IDE’s, compilers and other tools

Example: computing at compile time

```
#include <iostream>

int triangular(int n) {
     return (n == 1)? 1 : triangular(n-1) + n;
}

int main () {
     int result = triangular(20);
     std::cout << result << '\n';
}
```

C++ function to compute sum of first *n* integers

```
#include <iostream>

template <int t>
constexpr  int triangular() {
     return triangular<t - 1>() + t;
}
template <>
constexpr  int triangular<1>() {
     return 1;
}
int main () {
     int result = triangular<20>();
     std::cout << result << '\n';
}
```

C++ template with specialization computing the same

`constexpr` "invites" the compiler to evaulate the expression

C++ Template implementation

- Compilation on demand: the code of a template is not compiled until an instantiation is required
- Compile-time instantiation (**Static binding**)
  - 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
  - **Overloading resolution** *after* substitution
    - Fails if some operator is not defined for the type instance
    - Example: if T does not implement < in **`GetMax`**

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

On instantiation

- In C/C++ usually the *declarations* of functions (*prototypes)* are collected in a *header file* (<name>.h), while the actual *definitions* are in a separate file
- In the case of **template functions**, the compiler needs both its declaration and its definition to instantiate it.
- Thus limited forms of "separate compilation": cannot compile *definition* of template and code instantiating the template separately.
- Explicit instantiation possible. Example:

```
template int GetMax<int>(int a, int b);
```