# Templates I

## PB173 Programming in Modern C++

Nikola Beneš, Vladimír Štill, Jiří Weiser

Faculty of Informatics, Masaryk University

spring 2016

# Outline

- basic templates recap
- overloading
- dependent `this->`, `typename`
- variadic templates

# Templates

## What you (should) already know

### Motivation

- generic programming; polymorphism
- metaprogramming; compile-time programming

### Syntax

- `template< parameters >`
- parameters can be:
  - types: `typename` or `class`
  - values: integers, enums, (pointers, references)
  - templates: more about this later
- parameters can have a default value

```
template< typename T, typename U = bool, size_t N = 10 >
```

# Templates

## What you (should) already know

### Template Instantiation

- templates instantiated at compile time
  - all templated functions/classes in header files
- methods of templated classes only instantiated if actually used
  - why?

```
template< typename T > struct X {
    T t;
    void f() { t.f(); }
    void g() { t.g(); }
};
struct A { void f() {} };
int main() {
    X<int> xi;
    X<A>   xa;
    xa.f();
}
```

# Templated Functions

## Function Overloading

- more functions applicable to the given parameters
- need to consider priority
  1. function with less conversions needed wins
  2. non-templated wins over templated
  3. more specialised template wins

*Note:* for more details, see

- `http://en.cppreference.com/w/cpp/language/overload_resolution`
- `http://en.cppreference.com/w/cpp/language/function_template#Function_template_overloading`

# Templated Functions & Overloading

07_overload_order_1.cpp

```
void f( int )  { std::cout << "int, "; }  // (1)
void f( long ) { std::cout << "long, "; } // (2)
template< typename T >
void f( T )    { std::cout << "T, "; }    // (3)

int main() {
    // what is the output?
    f( 42 );                 // (a)
    f( unsigned( 42 ) ); // (b)
    f( long( 42 ) );         // (c)
    f( short( 42 ) );        // (d)
}
```

# Templated Functions & Overloading

07_overload_order_1.cpp

```
void f( int )  { std::cout << "int, "; }  // (1)
void f( long ) { std::cout << "long, "; } // (2)
template< typename T >
void f( T )    { std::cout << "T, "; }    // (3)

int main() {
    // what is the output?
    f( 42 );                 // (a)
    f( unsigned( 42 ) ); // (b)
    f( long( 42 ) );         // (c)
    f( short( 42 ) );        // (d)
}
```

Output: int, T, long, T,

# Templated Functions & Overloading

07_overload_order_2.cpp

```
void g( int, int ) { std::cout << "ii, "; } // (1)
template< typename T >
void g( int, T )   { std::cout << "iT, "; } // (2)
template< typename T >
void g( T, int )   { std::cout << "Ti, "; } // (3)
template< typename T >
void g( T, T )     { std::cout << "TT, "; } // (4)
template< typename T, typename U >
void g( T, U )     { std::cout << "TU, "; } // (5)

int main() {
    g( 1, 1 ); g( 1, 1L ); g( 1L, 1 ); g( 1L, 1L );
    g( 1L, 1U ); g( 1U, 1U );
}
```

# Templated Functions & Overloading

07_overload_order_2.cpp

```
void g( int, int ) { std::cout << "ii, "; } // (1)
template< typename T >
void g( int, T )   { std::cout << "iT, "; } // (2)
template< typename T >
void g( T, int )   { std::cout << "Ti, "; } // (3)
template< typename T >
void g( T, T )     { std::cout << "TT, "; } // (4)
template< typename T, typename U >
void g( T, U )     { std::cout << "TU, "; } // (5)

int main() {
    g( 1, 1 ); g( 1, 1L ); g( 1L, 1 ); g( 1L, 1L );
    g( 1L, 1U ); g( 1U, 1U );
}
```

Output: ii, iT, Ti, TT, TU, TT,

# Templated Function & Specialisation

07_template_spec.cpp

```
void f( int )   { std::cout << "int, "; } // (1)
template< typename T >
void f( T )     { std::cout << "T, "; }   // (2)
template<>
void f<>( int ) { std::cout << "T:i, "; } // (3)
```

# Templated Function & Specialisation

07_template_spec.cpp

```
void f( int )   { std::cout << "int, "; } // (1)
template< typename T >
void f( T )     { std::cout << "T, "; }   // (2)
template<>
void f<>( int ) { std::cout << "T:i, "; } // (3)
```

**Recommendation:** do not use template specialisation for *functions*; see http://www.gotw.ca/publications/mill17.htm

# Dependent Names

## Names Inside Templates

- dependent vs. non-dependent
- names that depend on the template instantiation
- notably: `this` is a dependent name

## Using `typename`

- tell the compiler that the following dependent name is a type

```
template< typename T >
void printVector( const std::vector< T > & v ) {
    typename std::vector< T >::size_type size = v.size();
    // does not work without typename, why?
    // ...
}
```

*More info:* `http://en.cppreference.com/w/cpp/language/dependent_name`

# Dependent Names

## Using `this`

```
int x = 42;
template< typename T >
struct A : T {
    void f() { std::cout << x << ", "; }
    void g() { std::cout << this->x << ",  "; }
};
struct X { int x = 17; };
struct Y {};

int main() {
    A<Y> ay;
    A<X> ax;
    ay.f(); ax.f(); ax.g();
}
```

# Dependent Names

**Using `this`**

```
int x = 42;
template< typename T >
struct A : T {
    void f() { std::cout << x << ", "; }
    void g() { std::cout << this->x << ",  "; }
};
struct X { int x = 17; };
struct Y {};

int main() {
    A<Y> ay;
    A<X> ax;
    ay.f(); ax.f(); ax.g();
}
```

Output: 42, 42, 17,

# Dependent Names

**Using `this`**

```
template< typename T >
struct A : T {
    void f() { this->g(); }
};

struct X {
    void g() { std::cout << "X::g"; }
};

int main() {
    A<X> ax;
    ax.f();
}
```

# Variadic Templates

## Syntax

```
// variadic class template
template< typename ... Args >
class X { /* ... */ };

// variadic function template
template< typename ... Args >
void foo( Args ... args ) { /* ... */ };

// does not have to be types
template< int ... Nums >
void foo() { /* ... */ };
```

## Size

- `sizeof...( Args )` returns number of arguments

# Variadic Templates

## Working with Argument Packs

- forward somewhere else
- recursion
- (C++17, possibly) fold

## Expanding Argument Packs

```
template< typename ... Args >
void foo( Args ... args ) {
    do_something( args... );
    // expands to: do_something( arg1, arg2, ..., argN );
    SomeClass obj{ args... };
    // expands to: SomeClass obj{ arg1, arg2, ..., argN };
    do_something( modify(args)... );
    // expands to: do_something( modify(arg1),
    //       modify(arg2), ..., modify(argN) );
}
```

# Variadic Templates

**Forwarding**

```
template< typename T, typename ... Args >
std::shared_ptr< T > make_shared( Args ... args ) {
    return std::shared_ptr< T >(
        new T( args... )
    );
}
```

- what's wrong (inefficient)?

# Variadic Templates

## Forwarding

```
template< typename T, typename ... Args >
std::shared_ptr< T > make_shared( Args ... args ) {
    return std::shared_ptr< T >(
        new T( args... )
    );
}
```

- what's wrong (inefficient)?

## Perfect Forwarding

```
template< typename T, typename ... Args >
std::shared_ptr< T > make_shared( Args && ... args ) {
    return std::shared_ptr< T >(
        new T( std::forwards< Args >( args )... )
    );
}
```

# Variadic Templates

## Expanding Argument Packs with Recursion

```
void print() { std::cout << std::endl; }
template< typename T, typename ... Args >
void print( const T & t, const Args & ... args ) {
    std::cout << t;
    print( args... );
}

int main() {
    print( "Hello, ", ' ', "world! ", 42 );
}
```

# Variadic Templates

## Expanding Argument Packs with Recursion

```
template< typename T >
const T & max( const T & t ) { return t; }

template< typename T, typename ... Args >
const T & max( const T & a, const T & b,
               const Args & ... args ) {
    return std::max( a, max( b, args... ) );
}

int main() {
    return max( 1, 7, 4, 12, 17, 42, 0, -8 );
}
```

# Tuples

**`std::tuple`**

- uses variadic templates
- usage similar to `std::pair`

```
auto myTuple = std::make_tuple( 1, 3.14,
                                std::string( "Hello" ) );
// the type is std::tuple< int, double, std::string >

std::get< 2 >( myTuple ) += ", world!";
std::cout << std::get< 1 >( myTuple ); // 3.14
```

**`std::tie`**

- creates a tuple of lvalue references
- use: unpack a tuple into variables

```
int i; double d; std::string s;
std::tie( i, d, s ) = myTuple;
```

# Exercise

**Simple Pretty-Printer Library** `07_fmt_exercise.h`

- print basic data types and standard containers / collections
  - `std::vector`, `std::set`, `std::pair`, `std::tuple`
- usage:

```
int main() {
    std::vector< std::pair< int, int > > vec{
        {1, 2}, {17, 42}, {0, -7} };
    std::cout << str::fmt( vec ) << '\n';
    std::string s =
        Printer().fmt( "x " ).fmt( vec ).str();
    std::cout << s << '\n';
}
```