

# Advanced Programming Concepts with C++ CSI2372 – Fall 2019

Jochen Lang &  
Mohamed Taleb  
EECS

Université d'Ottawa | University of Ottawa



uOttawa

L'Université canadienne  
Canada's university



[uOttawa.ca](http://uOttawa.ca)

# This lecture

## Do more with less

- **Macros and Templates**
- **Textbook (Lippman): Chapters 2.9.2, 6.14, 16.1-16.3, 16.5**
  - Macros and the C++ preprocessor: debugging, conditional compilation
  - Templates: template functions and classes
  - Templates: type and non-type parameters
  - Template specialization

# Macros and the Preprocessor

- **Preprocessor directives used so far in the course**

`#include`

- Paste a file into the current file

`#define`

- Define a preprocessor variable

`#ifdef #ifndef`

- Branch; check if preprocessor variable exists/does not exist.

`#endif`

- End of branch

# Macros

- **Avoid in general**
  - Use templates, inline functions and const variables instead
- **Macro Examples**

```
#define PI 3.141593
#define MIN(a,b) ((a)<(b))?(a):(b)
#define forever for(;;)

float diameter = PI * PI;
float smallNum = 1.0, largeNum = 1000.0, myNum;
myNum = MIN(smallNum, largeNum);
forever { ...
if ( check == true ) break;
}
```

# Useful Preprocessor Directives

- Include file
- Conditional include
- Conditional compile (e.g., modification of source file depending on target)
- Debugging
  - Exclude sections of code
  - Comments for debug version
  - Checking of pre- and post conditions with assert

# Using assert

- **Macro defined in <cassert>**
  - Program will exit if argument is false
  - Can be turned off by `#define NDEBUG`
  - Note defines can also be done from the command line of (most) compilers
- **Example**

```
#include <cassert>

float yValue( float x ) {
    assert( x>0 );
    return sqrt( x );
}
```

# Templates

- **Inheritance in object oriented programming enables polymorphism at run-time (virtual functions)**
- **Templates in generic programming enables polymorphism at compile-time**
  - Different specialized code is generated as required from the generic code
  - Templates do not generate class hierarchies
  - Compile-time method (run-time efficient!)
  - Templates can be used with functions and classes

# Basic Application of Function Templates

- **Simple Example**

```
int min(const int& g, const int& d) {  
    return ((g < d) ? g : d);  
}  
double min(const double& g, const double& d) {  
    return ((g < d) ? g : d);  
}  
string min(const string& g, const string& d) {  
    return ((g < d) ? g : d);  
}
```

- **C-style "solution" for very short functions**

```
#define MIN(a,b) ((a)<(b))?(a):(b))  
int i1,i2,i3;  
double d1,d2,d3;  
i3 = MIN(i1,i2); d3 = MIN(d1,d2);
```



# Function Template Example

- **Definition**

```
template <typename T>
T min(const T& g, const T& d) {
    return ((g < d) ? g : d);
}
```

- **Use and Instantiation**

```
int i1,i2,i3;
double d1,d2,d3;

i3 = min(i1,i2);
d3 = min(d1,d2);
```

# Function Templates generate Multiple Functions during Compilation

- **The example:**

```
int i1,i2,i3;  
double d1,d2,d3;  
i3 = min(i1,i2); d3 = min(d1,d2);
```

- **Generates effectively two functions:**

```
int min(const int& g, const int& d) {  
    return ((g < d) ? g : d);  
}  
double min(const double& g, const double& d) {  
    return ((g < d) ? g : d);  
}
```

# Limits to Type Inference in Template Instantiation

- No automatic type conversions
- Example

```
template <typename T> T add(T g, T d) { return g+d; }  
int iVal;  
short sVal;  
double dVal;  
short *psVal = &sVal;
```

```
add(dVal, iVal),  
add(sVal, iVal);  
add(psVal, sVal);
```

# Explicit Template Function Initialization

- **Example**

```
template <typename T> T add(T g, T d) { return g+d; }  
int iVal;  
short sVal;  
double dVal;  
short *psVal = &sVal;  
  
add<double>(dVal, iVal);  
add<int>(sVal, iVal);  
add<short*>(psVal, sVal);
```

# Multiple Template Parameters

Note: Types in templates can also be specified with `<class T>` instead of `<typename T>`

```
template <class T1, class T2, class T3>
T1 add(const T2& a, const T3& b) {
    T1 res = a + b;
    return res;
}
```

- **Use and Initialization**

```
int i1, i2;
short sum1 = add<short>(i1,i2);
long sum2 = add<long>(i1,i2);
```

# Class Templates

- **Create a set of classes**
- **Example:**

```
template <class T>
class Point2D {
    T d_x, d_y;
public:
    Point2D();
    Point2D( T _x, T _y );
    Point2D<T> add( const Point2D<T>& _oPoint ) const;
    ...
};
Point2D<int> intPt1, intPt2, intPt3;
intPt3 = intPt1.add( intPt2 );
Point2D<double> dPt1, dPt2, dPt3;
dPt3 = dPt1.add( dPt2 );
```

# Compilation of Template Code

- **Standard file organisation**

- Template class declared in header file (e.g., `point2d.h`)
- Template class defined in source (cpp) file (e.g., `point2d.cpp`).
- Instantiation in source file which uses the template class (e.g., `example.cpp`).

- **But:**

- Compiler needs to know what instantiations of a template are required in order to create code (in `point2d.cpp`).
  - Simple split with declaration in header file and definition in source (cpp) file won't work

# Inclusion Compilation Model

- Inclusion compile model simply includes the template code in every file where its initialized.

- **Example:**

```
#ifndef POINT2D_H_  
#define POINT2D_H_  
template <class T>  
class Point2D { ... };  
#include "point2d.cpp"  
#endif
```

- **Problems:**

- Template code is repeated everywhere where the header is included
- Remember to not compile template cpp file(s)



# Working with the Inclusion Compilation Model

- Avoid the include of a cpp file by declaring and defining the template class in the header file
  - Not really any better
- Generate a separate cpp where all the instantiation are repeated
  - Avoids duplicate code
  - Must remember to copy all instantiation to separate file
- **Many compiler support the inclusion model and eliminate duplicate template code during link stage**

# Added Support in C++11

## extern template

- Force the compiler to *not* instantiate a template
- It must be instantiated somewhere else
- Reduces compile time and reduces object file size if the template is not instantiated everywhere but only once

```
template<typename T> T inc( const T& _in ) {  
    return _in++;  
}
```

```
// Must be instantiated elsewhere without extern  
extern template inc<int>();
```

```
int i = 3;  
i = inc( i );
```

# Debugging Template Code

- **Errors possible in different Stages**
  - Compilation of template definition
    - Possible source of errors: syntax
  - Compilation of code which uses template
    - Possible source of errors: number and type of arguments
  - Linking
    - Possible source of errors: type-related errors, template function definition not found etc.

# Non-Type Parameters

- Can use other parameters except type specifiers
- Must be compile-time constant expression
- **In general:**
  - 3 types of parameters: type, non-type, template

# Example: N Dimensional Point

```
template <class T, const int NUM>
class Point{
    T d_components[ NUM ];
public:
    Point();
    Point( T* _components );
    Point<T, NUM> add( const Point2D<T, NUM>& _oPoint ) const;
    void print() const; ...
};
```

```
Point<int, 2> intPt1, intPt2, intPt3;
intPt3 = intPt1.add( intPt2 );
```

```
Point<double, 3> dPt1, dPt2, dPt3;
dPt3 = dPt1.add( dPt2 );
```

# Specialization

- **Specify a method, function or class for a specific initialization of the template**
  - Template class may work for many types but not all, need a specialized version for these types
  - Some types make better implementation possible (e.g., more efficient, more versatile etc.), specialize for these types
  - Some extra methods for specific types

# Function Specialization

- Template function parameters must match exactly (no automatic conversions are applied)

```
template <typename T>
inline T min(T g, T d) {
    return ((g < d) ? g : d);
}
```

```
template <>
inline const char* min<const char*>( const char* g,
                                     const char* d ) {
    if ( strcmp( g, d ) < 0 ) return g;
    else return d;
}
```

# Class Specialization and Partial Specialization

- **Example: Specialize Point for 2-D**
  - Note: Methods and attributes of class may change

```
template <class T>
class Point<T,2>{
    T d_components[2];
public:
    Point();
    Point( T* _components );
    Point( T& _x, T& _y );
    Point<T,2> operator+( const Point2D<T,2>& _oPoint ) const;
    void print() const;
};

template <class T>
Point<T,2>::Point<T,2>( T& _x, T& _y )
: d_component[0] = _x, d_component[1] = _y
{ }
```



# Specialization of Selected Methods

- **Specialize only a specific method(s)**

```
template <class T, NUM>
class Point{
    T d_component[NUM];
public:
    void print() const;
...
};

template<> void Point<double,2>::print() const
{
    cout << "2D Point: ( " << d_component[0];
    cout << ", " << d_component[1] << " ) " << endl;
    return res;
}
```

**Function specialization  
never allows for partial  
specialization!**

# C++11: template typedefs

- **Templates**

- Typedefs for templates with the keyword `using`

```
// typedef with templates
template<class T> using ref = T&;
double y;
ref<double> x = y;

// Assume a template class Point as before:
// template <class T, const int NUM> class Point;
template<class T, const int NUM> using PointNDim = Point<T,NUM>;
template<class T> using Point4Dim = Point<T,4>;

PointNdim<double,5> pt5D;
Point4Dim<double> oPt4D;
```

# C++11: template default arguments

- **Similar idea than default function arguments**

```
// default template arguments
template <typename T = int> bool isLess(T g, T d) {
    return g < d;
}
```

- **Probably most often used with callables**
  - to be discussed

```
// default template arguments, std comparator
template <typename T, typename F = std::less<T> >
    bool isLess(T g, T d) {
        return F(g, d);
    }
```

# C++11: static\_assert

- Check assumptions at compile time
- Used for templates to check type assumptions
  - Header `<type_traits>` contains classes to infer type information at compile time
- Syntax: `static_assert( bool_constexpr, string );`

```
#include <type_traits>
template <class T> T copy( T& _in ) {
    static_assert(std::is_copy_constructible<T>::value,
                  "Need T with c-ctor");

    T res(_in);
    return res;
}
```

# Traits

- **Traits specify the properties of a type**
- **E.g.: `char_traits` class used with strings and streams contains, e.g.,**
  - typedefs for `char_type`, `int_type`, `off_type`, `pos_type`, `state_type` for specific type (char or wchar)
  - member functions
    - `assign()` assignment for specific type
    - `compare()` comparison for specific type
    - `eof()` method which returns eof for specific type
    - etc.
  - Used in templates to shield the user from implementation detail and simplify template initialization

# Next Lecture

## Write even less code

- **Callable Objects**
  - Textbook (Lippman): Chapters 6.7, 10.3
  - Passing a function: function pointers, functors
  - C++11 bind
  - Lambdas