COMP6771
Advanced C++ Programming

Week 8
Part Two: Template Metaprogramming

2017

www.cse.unsw.edu.au/˜cs6771

Metaprogramming

- Metaprogramming is the writing of computer programs with the ability to treat other program code as their data. It means that a program could be designed to read, generate, analyse or transform other programs, or do part of the work during compile time that is otherwise done at run time.
- Metalanguage: The language in which the metaprogram is written
- Reflection: The ability of a programming language to be its own metalanguage
- Object language: the language of the programs being manipulated
- Examples:
  - compilers, lex and yacc
  - Metaprogramming involves modifying programs at run time as in Lisp, Python, Ruby and Perl
- C++ metaprogramming: static (compile time calculated values)

Computing Factorials

- Factorials: 0! = 1, 1! = 1, 2! = 2 * 1!, 3! = 3 x 2!...
- Compile-time computation of factorials using templates.

```
#include <iostream>

template<int n> struct Factorial {
  static const long val = Factorial<n-1>::val * n;
};

template<> struct Factorial<0> {
  static const long val = 1; // must be a compile-time constant
};

int main() {
  std::cout << Factorial<6>::val << std::endl;
}
```

- Such programs are called template metaprograms (i.e., programs about programs)
- The compiler recursively instantiates Factorial until the specialisation is invoked
- Turing-complete (since it supports if-else and recursion)

Computing Factorials (C++11)

```
#include <iostream>

constexpr int factorial (int n) {
  return n > 0 ? n * factorial( n - 1 ) : 1;
}

int main() {
  std::cout << factorial(6) << std::endl;
}
```

- Recall the differences between const and constexpr
- Note: this is not a template metaprogram

Loop Unrolling

- Consider calculating the dot product over a large Euclidean Vector (a[0] * b[0] + a[1] * b[2]...):
- Could do this in a loop at run time:

```
double calcDotProduct(EV a, EV b, int numDim) {
  double result = 0;
  for (unsigned int i = 0; i < numDim; ++i) {
      result += ( a[i] * b[i] );
  }
  return result;
}
```

- Code sketch of Loop Unrolling by the optimiser (elimates the counter `i` variable):

```
double calcDotProduct(EV a, EV b) {
  double result = ( a[1] * b[1] );
  result += ( a[2] * b[2] );
  result += ( a[3] * b[3] );
  ...
  return result;
}
```

Loop Unrolling with Metaprogramming

```
// primary template
template <int DIM, typename T>
struct DotProduct {
  static T result (T* a, T* b) {
    return *a * *b  +  DotProduct<DIM-1,T>::result(a+1,b+1);
  }
};

// partial specialisation as end criteria
template <typename T>
struct DotProduct<1,T> {
  static T result (T* a, T* b) {
    return *a * *b;
  }
};
```

Modified from:

http://www.informit.com/articles/article.aspx?p=30667&seqNu

Loop Unrolling

- Client code:

```
// convenience function
template <int DIM, typename T>
inline T dot_product (T* a, T* b) {
    return DotProduct<DIM,T>::result(a,b);
}

int main() {
    int a[3] = {1, 2, 3};
    int b[3] = {5, 6, 7};

    std::cout << dot_product<3>(a,b) << std::endl; // 38
    std::cout << dot_product<3>(a,a) << std::endl; // 14
}
```

- Compile-time computations:

```
DotProduct<3, int>:result(a, b)
= *a * *b + DotProduct<2, int>:result(a+1, b+1)
= *a * *b + *(a+1) * *(b+1) + DotProduct<1, int>:result(a+2, b+2)
= *a * *b + *(a+1) * *(b+1) + *(a+2) * *(b+2)
```

- Used in numeric libraries such as Blitz++ and MTL

Template Recursion

- Consider the problem of trying to make a N-Dimensional grid.
- Could make a one dimension, length 10, grid using: `std::vector<int> grid(10);`
- To make a two dimension grid (size 10x10), we would need to create: `std::vector<std::vector<int>> grid2D(10);`
- But then we would need to ensure that each inner vector is also resized to 10 before use.
- Can use the following class to wrap a vector and get this behaviour right...

OneDGrid.h

```
#pragma once

#include <cstddef>
#include <vector>

template <typename T>
class OneDGrid {
public:
  explicit OneDGrid(size_t inSize = 10);
  virtual ~OneDGrid();

  T& operator[](size_t x);
  const T& operator[](size_t x) const;

  void resize(size_t newSize);
  size_t getSize() const { return mElems.size(); }

private:
  std::vector<T> mElems;
};
```

OneDGrid.h

```
template <typename T>
OneDGrid<T>::OneDGrid(size_t inSize) {
  mElems.resize(inSize);
}

template <typename T>
OneDGrid<T>::~OneDGrid() {
  // Nothing to do, the vector will clean up itself.
}

template <typename T>
void OneDGrid<T>::resize(size_t newSize) {
  mElems.resize(newSize);
}

template <typename T>
T& OneDGrid<T>::operator[](size_t x) {
  return mElems[x];
}

template <typename T>
const T& OneDGrid<T>::operator[](size_t x) const {
  return mElems[x];
}
```

User Code

```
#include "OneDGrid.h"

int main() {
  OneDGrid<int> singleDGrid;
  OneDGrid<OneDGrid<int>> twoDGrid;
  OneDGrid<OneDGrid<OneDGrid<int>>> threeDGrid;

  singleDGrid[3] = 5;
  twoDGrid[3][3] = 5;
  threeDGrid[3][3][3] = 5;
}
```

- We have solved the resizing problem.
- N dimensional grids can be generated through recursive types.
- But, the nested types are messy, e.g. `OneDGrid<OneDGrid<OneDGrid<int>>> threeDGrid;`
- It would be nicer to write: `NDGrid<int ,3> threeDGrid;`
- We can do this through a recursive template...

Recursive Template Class

```
template <typename T, size_t N>
class NDGrid {
public:
  explicit NDGrid(size_t inSize = 10);
  virtual ~NDGrid();

  NDGrid<T, N-1>& operator[](size_t x);
  const NDGrid<T, N-1>& operator[](size_t x) const;

  void resize(size_t newSize);
  size_t getSize() const { return mElems.size(); }

private:
  std::vector<NDGrid<T, N-1>> mElems;
};
```

- The private vector field is recursive on the template class!
- It creates a NDGrid class of dimension N-1.
- Also note the operator[] functions return references to NDGrid<T, N-1> objects not T objects!
- How do we stop the recursion?

Recursive Template Base Class

Stop the recursion with a partial specialization for dimension 1.

```
template <typename T>
class NDGrid<T, 1> {
public:
  explicit NDGrid(size_t inSize = 10);
  virtual ~NDGrid();

  T& operator[](size_t x);
  const T& operator[](size_t x) const;

  void resize(size_t newSize);
  size_t getSize() const { return mElems.size(); }

private:
  std::vector<T> mElems;
};
```

- The private vector and operator[] types are now of type T.

Recursive Template Implementation

```
template <typename T, size_t N>
NDGrid<T, N>::NDGrid(size_t inSize) {
  resize(inSize);
}

template <typename T, size_t N>
NDGrid<T, N>::~NDGrid() {
  // Nothing to do, the vector will clean up itself.
}

template <typename T, size_t N>
void NDGrid<T, N>::resize(size_t newSize) {
  mElems.resize(newSize);
  // Resizing the vector calls the 0-argument constructor for
  // the NDGrid<T, N-1> elements, which constructs
  // it with the default size. Thus, we must explicitly call
  // resize() on each of the elements to recursively resize all
  // nested Grid elements.
  for (auto& element : mElems) {
    element.resize(newSize);
  }
}

template <typename T, size_t N>
NDGrid<T, N-1>& NDGrid<T, N>::operator[](size_t x) {
  return mElems[x];
}

template <typename T, size_t N>
const NDGrid<T, N-1>& NDGrid<T, N>::operator[](size_t x) const {
  return mElems[x];
}
```

Partial Specialization Implementation

```
template <typename T>
NDGrid<T, 1>::NDGrid(size_t inSize) {
  resize(inSize);
}

template <typename T>
NDGrid<T, 1>::~NDGrid() {
  // Nothing to do, the vector will clean up itself.
}

template <typename T>
void NDGrid<T, 1>::resize(size_t newSize) {
  mElems.resize(newSize);
}

template <typename T>
T& NDGrid<T, 1>::operator[](size_t x) {
  return mElems[x];
}

template <typename T>
const T& NDGrid<T, 1>::operator[](size_t x) const {
  return mElems[x];
}
```

User Code

The following user code now works and is much cleaner than the earlier version.

```
#include "NDGrid.h"

#include <iostream>

int main() {
  NDGrid<int, 3> my3DGrid(3);
  my3DGrid[2][1][2] = 5;
  my3DGrid[1][1][1] = 5;

  std::cout << my3DGrid[2][1][2] << std::endl;

  return 0;
}
```

Compile-Time Expressions

Calculating Square Root

```
#include <iostream>

 // primary template to compute sqrt(N)
template <long N, long LO=1, long HI=N>
struct Sqrt {
  // compute the midpoint, rounded up
  static const long mid = (LO+HI+1)/2;
  // search a not too large value in a halved interval
  static const long result = (N<mid*mid) ?
                      (long) Sqrt<N,LO,mid-1>::result
                      : (long) Sqrt<N,mid,HI>::result;
};

// partial specialisation for the case when LO equals HI
template<long N, long M>
struct Sqrt<N,M,M> {
  static const long result = M;
};

int main() {
  std::cout << Sqrt<16>::result << std::endl;
}
```

Compile-Time Selection

- Client code:

```
std::cout << Sqrt<16>::result << std::endl;
```

- Compile-time computations:

```
Sqrt<16>::result expanded to Sqrt<16, 1, 16>::result

mid = (1+16+1)/2 = 9
result = (16 < 9*9 ? Sqrt<16,1,8>::result
                   : Sqrt<16,9.16>::result
       = Sqrt<16,1,8>::result

mid = (1+8+1)/2 = 5
result = (16 < 5*5 ? Sqrt<16,1,4>::result
                   : Sqrt<16,5.8>::result
       = Sqrt<16,1,4>::result
...

Finally, the specialisation Sqrt<16, 4, 4>::result=4 called
```

C++11 `constexpr` Version

```
#include <iostream>

static constexpr long ct_mid(long a, long b){
    return (a+b) / 2;
}

static constexpr long ct_pow(long a) {
    return a*a;
}

static constexpr long ct_sqrt(long res, long lo, long hi) {
    return
        lo == hi ? hi          // case when lo == hi
        : ct_sqrt(res, ct_pow(
            ct_mid(lo, hi)) >= res ? lo : ct_mid(lo, hi) + 1,
            ct_pow(ct_mid(lo, hi)) >= res ? ct_mid(lo, hi) : hi);
}

static constexpr long ct_sqrt(long res) {
    return ct_sqrt(res, 1, res);
}

int main() {
  std::cout << ct_sqrt(16) << std::endl;
}
```

No need for the class variables storing the mid point, but can still be confusing.

C++14 constexpr Version

```
#include <iostream>

static constexpr long ct_sqrt(long res, long lo, long hi) {
  if(lo == hi) {
    return hi;
  } else {
    const auto mid = (lo + hi) / 2;

    if(mid * mid >= res) {
      return ct_sqrt(res, lo, mid);
    } else {
      return ct_sqrt(res, mid + 1, hi);
    }
  }
}

static constexpr long ct_sqrt(long res) {
  return ct_sqrt(res, 1, res);
}

int main() {
  std::cout << ct_sqrt(16) << std::endl;
}
```

- In C++14 can use if statements and loops in `constexpr`
- Not supported in `g++ 4.9.2`! Need `clang++-3.5`
- Example from:
  http://baptiste-wicht.com/posts/2014/07/compile-intege

Passing a Meta Template Function as a Parameter

```
// Passes a "meta template function" as a parameter at compile time.
#include <iostream>

// Accumulates the results of F(0)..F(n)
// class template with a template template parameter F
template<int n, template<int> class F>
struct Accumulate {
  static const int val = Accumulate<n-1, F>::val + F<n>::val;
};
// The stopping criterion (returns the value F(0))
template<template<int> class F>
struct Accumulate<0, F> {
  static const int val = F<0>::val;
};

// Various "functions":
template<int n> struct Identity {
  static const int val = n;
};
template<int n> struct Square {
  static const int val = n*n;
};
template<int n> struct Cube {
  static const int val = n*n*n;
};

int main() {
  std::cout << Accumulate<4, Identity>::val << std::endl; // 10
  std::cout << Accumulate<4, Square>::val << std::endl;   // 30
  std::cout << Accumulate<4, Cube>::val << std::endl;     // 100
}
```

Passing a Function as a Parameter at Compile Time

- Using a template template parameter to simulate passing a function as a parameter to another function
- What does it compute?

  `F(n) +_F(n-1) + ... F(0)`
- For three different functions passed:

  `Identity(4) +_Identity(3) + ... Identity(0) = 10`
  `Square(4) +_Square(3) + ... Square(0) = 30`
  `Cube(4) +_Cube(3) + ... Cube(0) = 100`
- See http://en.cppreference.com/w/cpp/language/template_para

BubbleSort I

```
#include <iostream>
// function to swap two values
template<int I, int J>
struct IntSwap {
  static inline void compareAndSwap(int* data) {
    if (data[I] > data[J])
      std::swap(data[I], data[J]);
  }
};

// loop to go through array
template<int I, int J>
class IntBubbleSortLoop {
private:
  static const bool go = (J <= I-2);
public:
  static inline void loop(int* data) {
    IntSwap<J,J+1>::compareAndSwap(data);
    IntBubbleSortLoop<go ? I : 0, go ? (J+1) : 0>::loop(data);
  }
};
```

BubbleSort II

```

template <>
struct IntBubbleSortLoop<0,0> {
  static inline void loop(int*) { }
};

// recursive metafunction to keep looping until sorted
template<int N>
struct IntBubbleSort {
  static inline void sort(int* data) {
    IntBubbleSortLoop<N-1,0>::loop(data);
    IntBubbleSort<N-1>::sort(data);
  }
};

template <>
struct IntBubbleSort<1> {
  static inline void sort(int* data) { }
};
```

Client Code

```
int main() {
  int a[] = {3, 1, 2, 5};
  IntBubbleSort<4>::sort(a);
  for (int i = 0; i < 4; i++)
    std::cout << a[i] << " ";
  std::cout << std::endl;

  // sort is called at runtime
  // metatemplate compiles required templates at compile time.
  a[0] = 100;
  IntBubbleSort<4>::sort(a);
  for (int i = 0; i < 4; i++)
    std::cout << a[i] << " ";
  std::cout << std::endl;
}
```

Output:

```
1 2 3 5
2 3 5 100
```

Client Instantiations

- Ideally, some extensive instantiations may lead to (when N=4):

```
inline void IntBubbleSort4(int* data) {
    if (data[0] > data[1]) std::swap(data[0], data[1]);
    if (data[1] > data[2]) std::swap(data[1], data[2]);
    if (data[2] > data[3]) std::swap(data[2], data[3]);
    if (data[0] > data[1]) std::swap(data[0], data[1]);
    if (data[1] > data[2]) std::swap(data[1], data[2]);
    if (data[0] > data[1]) std::swap(data[0], data[1]);
}
```

- The specialisation is several times faster than the standard algorithm

Benefits and Drawback of C++ Metaprogramming

- Compile-time versus execution-time tradeoff
- Generic programming
- Readability (can be hard to understand)
- Code bloat (due to some extensive instantiations/specialisations)

Type Relations, Type Traits and enable_if

- Recall the type_traits library from week 7.
- We can use the type_trait `is_same` to figure out if two variables are of the same type.

```
#include <iostream>
#include <string>
#include <type_traits>

template<typename T1, typename T2>
void same(const T1& t1, const T2& t2) {
  bool sameType = std::is_same<T1, T2>::value;
  std::cout << "'" << t1 << "' and '" << t2 << "' are ";
  std::cout << (sameType ? "the same types." : "different types.") << std::endl;
}

int main() {
  same(1, 32);
  same(1, 3.01);
  same(3.01, std::string("Test"));
}
```

- Output:

```
'1' and '32' are the same types.
'1' and '3.01' are different types.
'3.01' and 'Test' are different types.
```

SFINAE and enable_if

- What if we want to only create a template function for certain types?
- `enable_if` can be used to disable certain function overloads based on type traits.
- `enable_if` takes two type parameters, the first is a bool indicating if to enable or disable this template.
- `type_traits` can be used to determine the value of the bool (i.e., metaprogramming).
- The second type parameter is passed through to the nested type `::type` if the bool is true.
- If the bool is false, the second type parameter is not passed through and this can lead to compilation errors.
- But!, `enable_if` is an example of *Substitution Failure Is Not An Error* (SFINAE) which disables a function overload and backtracks to find another function overload that will compile. If no function is found than an error will be generated.

enable_if Example

```
template<typename T1, typename T2>
typename std::enable_if<std::is_same<T1, T2>::value, bool>::type
check_type(const T1& t1, const T2& t2) {
  std::cout << "'" << t1 << "' and '" << t2 << "' ";
  std::cout << "are the same types." << std::endl;
  return true;
}

template<typename T1, typename T2>
typename std::enable_if<!std::is_same<T1, T2>::value, bool>::type
check_type(const T1& t1, const T2& t2) {
  std::cout << "'" << t1 << "' and '" << t2 << "' ";
  std::cout << "are different types." << std::endl;
  return false;
}

int main() {
  check_type(1, 32);
  check_type(1, 3.01);
  check_type(3.01, std::string("Test"));
}
```

- Return type of the function check_type is determined by enable_if, if the enable_if fails then no return type is provided.
- This causes an error, but because of SFINAE, the second overload will be generated.
- Leaving out either version of check_type will compile error.

Trailing return, Variadic and SFINAE template

```
#include <iostream>

// this overload is always in the set of overloads
// ellipsis parameter has the lowest ranking for overload resolution
void test(...) {
  std::cout << "Catch-all overload called" << std::endl;
}

// this overload is added to the set of overloads if
// C is a reference-to-class type and F is a member function pointer
template<class C, class F>
auto test(C c, F f) -> decltype((void)(c.*f)(), void()) {
  std::cout << "Reference overload called" << std::endl;
}

// this overload is added to the set of overloads if
// C is a pointer-to-class type and F is a member function pointer
template<class C, class F>
auto test(C c, F f) -> decltype((void)((c->*f)()), void()) {
  std::cout << "Pointer overload called" << std::endl;
}
```

http://en.cppreference.com/w/cpp/language/sfinae

Trailing return, Variadic and SFINAE template

User code:

```
struct X { void f() {} };

int main(){
  X x;
  test( x, &X::f);
  test(&x, &X::f);
  test(42, 1337);
}
```

Output:

```
Reference overload called
Pointer overload called
Catch-all overload called
```

Static Assert

Consider if we want a function that operates only on classes that inherit from a particular base class or have a partiular type trait?

```
#include <type_traits>

class Base1 {};
class Base1Child : public Base1 {};

class Base2 {};
class Base2Child : public Base2 {};

template<typename T>
void process(const T& t) {
  static_assert(std::is_base_of<Base1, T>::value,
      "Base1 should be a base for T.");
}

int main(){
  process(Base1());
  process(Base1Child());
  process(Base2());      // Compile Error
  process(Base2Child()); // Compile Error
}
```

unique_ptr.h Implementation

- We now have all the understanding to read the implementations of the STL classes in the compiler!
- Let's look at the unique_ptr.h source
- https://android.googlesource.com/platform/prebuilts/gcc

Struct template in unique_ptr.h

```
/// Primary template, default_delete.
template<typename _Tp>
struct default_delete {
  constexpr default_delete() noexcept = default;

  template<typename _Up, typename = typename
      enable_if<is_convertible<_Up*, _Tp*>::value>::type>
    default_delete(const default_delete<_Up>&) noexcept { }

  void operator()(_Tp* __ptr) const {
    static_assert(sizeof(_Tp)>0,
      "can't delete pointer to incomplete type");
    delete __ptr;
  }
};
```

In one struct: constexpr, noexcept, enable_if, type traits, functors, static assert

Template Specialization in unique_ptr.h

```
// DR 740 - omit specialization for array objects with a
// compile time length
/// Specialization, default_delete.
template<typename _Tp>
struct default_delete<_Tp[]> {
  // ...
  void operator()(_Tp* __ptr) const {
      static_assert(sizeof(_Tp)>0,
        "can't delete pointer to incomplete type");
      delete [] __ptr;
    }

    template<typename _Up>
    typename enable_if<__is_derived_Tp<_Up>::value>::type
    operator()(_Up*) const = delete;
};
```

Template specialization and deleting a function based on enable_if

Inner class in Unique_ptr class in unique_ptr.h

```
/// 20.7.1.2 unique_ptr for single objects.
template <typename _Tp, typename _Dp = default_delete<_Tp> >
class unique_ptr {
  // use SFINAE to determine whether _Del::pointer exists
  class _Pointer {

    template<typename _Up>
    static typename _Up::pointer __test(typename _Up::pointer*);

    template<typename _Up>
    static _Tp* __test(...);

    typedef typename remove_reference<_Dp>::type _Del;
  public:
    typedef decltype(__test<_Del>(0)) type;
  };

```

Default arguments, Inner classes, SFINAE, decltype

Constructors in Unique_ptr class in unique_ptr.h

```
// Constructors.
constexpr unique_ptr() noexcept : _M_t()
{ static_assert(!is_pointer<deleter_type>::value,
  "constructed with null function pointer deleter"); }

unique_ptr(pointer __p,
typename remove_reference<deleter_type>::type&& __d) noexcept
: _M_t(std::move(__p), std::move(__d))
{ static_assert(!std::is_reference<deleter_type>::value,
  "rvalue deleter bound to reference"); }

constexpr unique_ptr(nullptr_t) noexcept : unique_ptr()

// Move constructors.
unique_ptr(unique_ptr&& __u) noexcept
: _M_t(__u.release(), std::forward<deleter_type>(__u.get_deleter()))

```

Constructors, Member initialization lists, std::move, Delegating Constructors

Operators in Unique_ptr class in unique_ptr.h

```
// Destructor.
~unique_ptr() noexcept {
  auto& __ptr = std::get<0>(_M_t);
  if (__ptr != nullptr)
    get_deleter()(__ptr);
  __ptr = pointer();
}

// Assignment.
unique_ptr& operator=(unique_ptr&& __u) noexcept {
  reset(__u.release());
  get_deleter() = std::forward<deleter_type>(__u.get_deleter());
  return *this;
}

unique_ptr& operator=(nullptr_t) noexcept {
  reset();
  return *this;
}
```

Destructors, Operator overloading, r-value references, std::forward, Function overloading

Member functions in Unique_ptr class

```
  void reset(pointer __p = pointer()) noexcept {
    using std::swap;
    swap(std::get<0>(_M_t), __p);
    if (__p != pointer())
      get_deleter()(__p);
  }

  void swap(unique_ptr& __u) noexcept {
    using std::swap;
    swap(_M_t, __u._M_t);
  }

};
```

Member functions, using statements, namespaces, STL algorithms

Reading

- Metaprogramming
  http://www.ibm.com/developerworks/linux/library/l-metap
- Chapter 5, Thinking in C++ (Eckel)
- Chapter 15, C++ Templates (Vandevoorde and Josuttis)
- C++ Template Metaprogramming (David Abrahams and Aleksey Gurtovoy), 2005.