

COMP6771

Advanced C++ Programming

Week 8

Part Two: Template Metaprogramming

2016

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

```

1 #include <iostream>
2
3 template<int n> struct Factorial {
4     static const long val = Factorial<n-1>::val * n;
5 };
6
7 template<> struct Factorial<0> {
8     static const long val = 1; // must be a compile-time constant
9 };
10
11 int main() {
12     std::cout << Factorial<6>::val << std::endl;
13 }

```

- 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)

```
1 #include <iostream>
2
3 constexpr int factorial (int n) {
4     return n > 0 ? n * factorial( n - 1 ) : 1;
5 }
6
7 int main() {
8     std::cout << factorial(6) << std::endl;
9 }
```

- 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 ($a[0] * b[0] + a[1] * b[2] \dots$):
- Could do this in a loop at run time:

```
1 double calcDotProduct(EV a, EV b, int numDim) {  
2     double result = 0;  
3     for (unsigned int i = 0; i < numDim; ++i) {  
4         result += ( a[i] * b[i] );  
5     }  
6     return result;  
7 }
```

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

```
1 double calcDotProduct(EV a, EV b) {  
2     double result = ( a[1] * b[1] );  
3     result += ( a[2] * b[2] );  
4     result += ( a[3] * b[3] );  
5     ...  
6     return result;  
7 }
```

Loop Unrolling with Metaprogramming

```
1 // primary template
2 template <int DIM, typename T>
3 struct DotProduct {
4     static T result (T* a, T* b) {
5         return *a * *b + DotProduct<DIM-1,T>::result(a+1,b+1);
6     }
7 };
8
9 // partial specialisation as end criteria
10 template <typename T>
11 struct DotProduct<1,T> {
12     static T result (T* a, T* b) {
13         return *a * *b;
14     }
15 };
```

Modified from: <http://www.informit.com/articles/article.aspx?p=30667&seqNum=7>

Loop Unrolling

- Client code:

```

1 // convenience function
2 template <int DIM, typename T>
3 inline T dot_product (T* a, T* b) {
4     return DotProduct<DIM,T>::result(a,b);
5 }
6
7 int main() {
8     int a[3] = {1, 2, 3};
9     int b[3] = {5, 6, 7};
10
11     std::cout << dot_product<3>(a,b) << std::endl; // 38
12     std::cout << dot_product<3>(a,a) << std::endl; // 14
13 }

```

- 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

```
1 #pragma once
2
3 #include <cstddef>
4 #include <vector>
5
6 template <typename T>
7 class OneDGrid {
8 public:
9     explicit OneDGrid(size_t inSize = 10);
10    virtual ~OneDGrid();
11
12    T& operator[](size_t x);
13    const T& operator[](size_t x) const;
14
15    void resize(size_t newSize);
16    size_t getSize() const { return mElems.size(); }
17
18 private:
19     std::vector<T> mElems;
20 };
```

OneDGrid.h

```

1  template <typename T>
2  OneDGrid<T>::OneDGrid(size_t inSize) {
3      mElems.resize(inSize);
4  }
5
6  template <typename T>
7  OneDGrid<T>::~~OneDGrid() {
8      // Nothing to do, the vector will clean up itself.
9  }
10
11 template <typename T>
12 void OneDGrid<T>::resize(size_t newSize) {
13     mElems.resize(newSize);
14 }
15
16 template <typename T>
17 T& OneDGrid<T>::operator[](size_t x) {
18     return mElems[x];
19 }
20
21 template <typename T>
22 const T& OneDGrid<T>::operator[](size_t x) const {
23     return mElems[x];
24 }

```

User Code

```

1  #include "OneDGrid.h"
2
3  int main() {
4      OneDGrid<int> singleDGrid;
5      OneDGrid<OneDGrid<int>> twoDGrid;
6      OneDGrid<OneDGrid<OneDGrid<int>>> threeDGrid;
7
8      singleDGrid[3] = 5;
9      twoDGrid[3][3] = 5;
10     threeDGrid[3][3][3] = 5;
11 }

```

- 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

```

1  template <typename T, size_t N>
2  class NDGrid {
3  public:
4      explicit NDGrid(size_t inSize = 10);
5      virtual ~NDGrid();
6
7      NDGrid<T, N-1>& operator[](size_t x);
8      const NDGrid<T, N-1>& operator[](size_t x) const;
9
10     void resize(size_t newSize);
11     size_t getSize() const { return mElems.size(); }
12
13 private:
14     std::vector<NDGrid<T, N-1>> mElems;
15 };

```

- 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.

```

1  template <typename T>
2  class NDGrid<T, 1> {
3  public:
4      explicit NDGrid(size_t inSize = 10);
5      virtual ~NDGrid();
6
7      T& operator[](size_t x);
8      const T& operator[](size_t x) const;
9
10     void resize(size_t newSize);
11     size_t getSize() const { return mElems.size(); }
12
13 private:
14     std::vector<T> mElems;
15 };

```

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

Recursive Template Implementation

```

1  template <typename T, size_t N>
2  NDGrid<T, N>::NDGrid(size_t inSize) {
3      resize(inSize);
4  }
5
6  template <typename T, size_t N>
7  NDGrid<T, N>::~~NDGrid() {
8      // Nothing to do, the vector will clean up itself.
9  }
10
11 template <typename T, size_t N>
12 void NDGrid<T, N>::resize(size_t newSize) {
13     mElems.resize(newSize);
14     // Resizing the vector calls the 0-argument constructor for
15     // the NDGrid<T, N-1> elements, which constructs
16     // it with the default size. Thus, we must explicitly call
17     // resize() on each of the elements to recursively resize all
18     // nested Grid elements.
19     for (auto& element : mElems) {
20         element.resize(newSize);
21     }
22 }
23
24 template <typename T, size_t N>
25 NDGrid<T, N-1>& NDGrid<T, N>::operator[](size_t x) {
26     return mElems[x];
27 }
28
29 template <typename T, size_t N>
30 const NDGrid<T, N-1>& NDGrid<T, N>::operator[](size_t x) const {
31     return mElems[x];
32 }

```

Partial Specialization Implementation

```

1  template <typename T>
2  NDGrid<T, 1>::NDGrid(size_t inSize) {
3      resize(inSize);
4  }
5
6  template <typename T>
7  NDGrid<T, 1>::~~NDGrid() {
8      // Nothing to do, the vector will clean up itself.
9  }
10
11 template <typename T>
12 void NDGrid<T, 1>::resize(size_t newSize) {
13     mElems.resize(newSize);
14 }
15
16 template <typename T>
17 T& NDGrid<T, 1>::operator[](size_t x) {
18     return mElems[x];
19 }
20
21 template <typename T>
22 const T& NDGrid<T, 1>::operator[](size_t x) const {
23     return mElems[x];
24 }

```

User Code

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

```
1 #include "NDGrid.h"
2
3 #include <iostream>
4
5 int main() {
6     NDGrid<int, 3> my3DGrid(3);
7     my3DGrid[2][1][2] = 5;
8     my3DGrid[1][1][1] = 5;
9
10    std::cout << my3DGrid[2][1][2] << std::endl;
11
12    return 0;
13 }
```


Compile-Time Expressions

Calculating Square Root

```

1  #include <iostream>
2
3  // primary template to compute sqrt(N)
4  template <long N, long LO=1, long HI=N>
5  struct Sqrt {
6      // compute the midpoint, rounded up
7      static const long mid = (LO+HI+1)/2;
8      // search a not too large value in a halved interval
9      static const long result = (N<mid*mid) ?
10                               (long) Sqrt<N,LO,mid-1>::result
11                               : (long) Sqrt<N,mid,HI>::result;
12 };
13
14 // partial specialisation for the case when LO equals HI
15 template<long N, long M>
16 struct Sqrt<N,M,M> {
17     static const long result = M;
18 };
19
20 int main() {
21     std::cout << Sqrt<16>::result << std::endl;
22 }

```


C++11 constexpr Version

```

1  #include <iostream>
2
3  static constexpr long ct_mid(long a, long b){
4      return (a+b) / 2;
5  }
6
7  static constexpr long ct_pow(long a) {
8      return a*a;
9  }
10
11 static constexpr long ct_sqrt(long res, long lo, long hi) {
12     return
13         lo == hi ? hi // case when lo == hi
14         : ct_sqrt(res, ct_pow(
15             ct_mid(lo, hi)) >= res ? lo : ct_mid(lo, hi) + 1,
16             ct_pow(ct_mid(lo, hi)) >= res ? ct_mid(lo, hi) : hi);
17 }
18
19 static constexpr long ct_sqrt(long res) {
20     return ct_sqrt(res, 1, res);
21 }
22
23 int main() {
24     std::cout << ct_sqrt(16) << std::endl;
25 }

```

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

C++14 constexpr Version

```

1  #include <iostream>
2
3  static constexpr long ct_sqrt(long res, long lo, long hi) {
4      if(lo == hi) {
5          return hi;
6      } else {
7          const auto mid = (lo + hi) / 2;
8
9          if(mid * mid >= res) {
10             return ct_sqrt(res, lo, mid);
11          } else {
12             return ct_sqrt(res, mid + 1, hi);
13          }
14      }
15  }
16
17  static constexpr long ct_sqrt(long res) {
18      return ct_sqrt(res, 1, res);
19  }
20
21  int main() {
22      std::cout << ct_sqrt(16) << std::endl;
23  }

```

- 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/>

Passing a Meta Template Function as a Parameter

```

1 // Passes a "meta template function" as a parameter at compile time.
2 #include <iostream>
3
4 // Accumulates the results of F(0)..F(n)
5 // class template with a template template parameter F
6 template<int n, template<int> class F>
7 struct Accumulate {
8     static const int val = Accumulate<n-1, F>::val + F<n>::val;
9 };
10 // The stopping criterion (returns the value F(0))
11 template<template<int> class F>
12 struct Accumulate<0, F> {
13     static const int val = F<0>::val;
14 };
15
16 // Various "functions":
17 template<int n> struct Identity {
18     static const int val = n;
19 };
20 template<int n> struct Square {
21     static const int val = n*n;
22 };
23 template<int n> struct Cube {
24     static const int val = n*n*n;
25 };
26
27 int main() {
28     std::cout << Accumulate<4, Identity>::val << std::endl; // 10
29     std::cout << Accumulate<4, Square>::val << std::endl;   // 30
30     std::cout << Accumulate<4, Cube>::val << std::endl;     // 100
31 }

```

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?

$$F(n) + F(n-1) + \dots + F(0)$$

- For three different functions passed:

$$\text{Identity}(4) + \text{Identity}(3) + \dots + \text{Identity}(0) = 10$$

$$\text{Square}(4) + \text{Square}(3) + \dots + \text{Square}(0) = 30$$

$$\text{Cube}(4) + \text{Cube}(3) + \dots + \text{Cube}(0) = 100$$

- See http://en.cppreference.com/w/cpp/language/template_parameters#Template_template_arguments

BubbleSort I

```

1  #include <iostream>
2  // function to swap two values
3  template<int I, int J>
4  struct IntSwap {
5      static inline void compareAndSwap(int* data) {
6          if (data[I] > data[J])
7              std::swap(data[I], data[J]);
8      }
9  };
10
11 // loop to go through array
12 template<int I, int J>
13 class IntBubbleSortLoop {
14 private:
15     static const bool go = (J <= I-2);
16 public:
17     static inline void loop(int* data) {
18         IntSwap<J, J+1>::compareAndSwap(data);
19         IntBubbleSortLoop<go ? I : 0, go ? (J+1) : 0>::loop(data);
20     }
21 };

```

BubbleSort II

```

22 |
23 | template <>
24 | struct IntBubbleSortLoop<0,0> {
25 |     static inline void loop(int*) { }
26 | };
27 |
28 | // recursive metafunction to keep looping until sorted
29 | template<int N>
30 | struct IntBubbleSort {
31 |     static inline void sort(int* data) {
32 |         IntBubbleSortLoop<N-1,0>::loop(data);
33 |         IntBubbleSort<N-1>::sort(data);
34 |     }
35 | };
36 |
37 | template <>
38 | struct IntBubbleSort<1> {
39 |     static inline void sort(int* data) { }
40 | };

```


Client Code

```

1 int main() {
2     int a[] = {3, 1, 2, 5};
3     IntBubbleSort<4>::sort(a);
4     for (int i = 0; i < 4; i++)
5         std::cout << a[i] << " ";
6     std::cout << std::endl;
7
8     // sort is called at runtime
9     // metatemplate compiles required templates at compile time.
10    a[0] = 100;
11    IntBubbleSort<4>::sort(a);
12    for (int i = 0; i < 4; i++)
13        std::cout << a[i] << " ";
14    std::cout << std::endl;
15 }

```

Output:

```

1 1 2 3 5
2 2 3 5 100

```

Client Instantiations

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

```
1 inline void IntBubbleSort4(int* data) {  
2     if (data[0] > data[1]) std::swap(data[0], data[1]);  
3     if (data[1] > data[2]) std::swap(data[1], data[2]);  
4     if (data[2] > data[3]) std::swap(data[2], data[3]);  
5     if (data[0] > data[1]) std::swap(data[0], data[1]);  
6     if (data[1] > data[2]) std::swap(data[1], data[2]);  
7     if (data[0] > data[1]) std::swap(data[0], data[1]);  
8 }
```

- 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.

```

1  #include <iostream>
2  #include <string>
3  #include <type_traits>
4
5  template<typename T1, typename T2>
6  void same(const T1& t1, const T2& t2) {
7      bool sameType = std::is_same<T1, T2>::value;
8      std::cout << "'" << t1 << "' and '" << t2 << "' are ";
9      std::cout << (sameType ? "the same types." : "different types.") << std::endl;
10 }
11
12 int main() {
13     same(1, 32);
14     same(1, 3.01);
15     same(3.01, std::string("Test"));
16 }

```

- Output:

```

1  '1' and '32' are the same types.
2  '1' and '3.01' are different types.
3  '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

```

1  template<typename T1, typename T2>
2  typename std::enable_if<std::is_same<T1, T2>::value, bool>::type
3  check_type(const T1& t1, const T2& t2) {
4      std::cout << "'" << t1 << " and '" << t2 << " ";
5      std::cout << "are the same types." << std::endl;
6      return true;
7  }
8
9  template<typename T1, typename T2>
10 typename std::enable_if<!std::is_same<T1, T2>::value, bool>::type
11 check_type(const T1& t1, const T2& t2) {
12     std::cout << "'" << t1 << " and '" << t2 << " ";
13     std::cout << "are different types." << std::endl;
14     return false;
15 }
16
17 int main() {
18     check_type(1, 32);
19     check_type(1, 3.01);
20     check_type(3.01, std::string("Test"));
21 }

```

- 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

```

1  #include <iostream>
2
3  // this overload is always in the set of overloads
4  // ellipsis parameter has the lowest ranking for overload resolution
5  void test(...) {
6      std::cout << "Catch-all overload called" << std::endl;
7  }
8
9  // this overload is added to the set of overloads if
10 // C is a reference-to-class type and F is a member function pointer
11 template<class C, class F>
12 auto test(C c, F f) -> decltype((void) (c.*f) (), void()) {
13     std::cout << "Reference overload called" << std::endl;
14 }
15
16 // this overload is added to the set of overloads if
17 // C is a pointer-to-class type and F is a member function pointer
18 template<class C, class F>
19 auto test(C c, F f) -> decltype((void) ((c->)*f) (), void()) {
20     std::cout << "Pointer overload called" << std::endl;
21 }

```

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

Trailing return, Variadic and SFINAE template

User code:

```
1 struct X { void f() {} };
2
3 int main(){
4     X x;
5     test( x, &X::f);
6     test(&x, &X::f);
7     test(42, 1337);
8 }
```

Output:

```
1 Reference overload called
2 Pointer overload called
3 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 particular type trait?

```

1  #include <type_traits>
2
3  class Base1 {};
4  class Base1Child : public Base1 {};
5
6  class Base2 {};
7  class Base2Child : public Base2 {};
8
9  template<typename T>
10 void process(const T& t) {
11     static_assert(std::is_base_of<Base1, T>::value,
12         "Base1 should be a base for T.");
13 }
14
15 int main(){
16     process(Base1());
17     process(Base1Child());
18     process(Base2()); // Compile Error
19     process(Base2Child()); // Compile Error
20 }
```

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/linux-x86/host/x86_64-w64-mingw32-4.8/+l-preview/x86_64-w64-mingw32/include/c++/4.8.3/bits/unique_ptr.h

Struct template in unique_ptr.h

```

1  /// Primary template, default_delete.
2  template<typename _Tp>
3  struct default_delete {
4      constexpr default_delete() noexcept = default;
5
6      template<typename _Up, typename = typename
7          enable_if<is_convertible<_Up*, _Tp*>::value>::type>
8          default_delete(const default_delete<_Up>&) noexcept { }
9
10     void operator()(_Tp* __ptr) const {
11         static_assert(sizeof(_Tp)>0,
12             "can't delete pointer to incomplete type");
13         delete __ptr;
14     }
15 };

```

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

Template Specialization in unique_ptr.h

```

1 // DR 740 - omit specialization for array objects with a
2 // compile time length
3 /// Specialization, default_delete.
4 template<typename _Tp>
5 struct default_delete<_Tp[]> {
6     // ...
7     void operator() (_Tp* __ptr) const {
8         static_assert(sizeof(_Tp)>0,
9             "can't delete pointer to incomplete type");
10        delete [] __ptr;
11    }
12
13    template<typename _Up>
14    typename enable_if<__is_derived_Tp<_Up>::value>::type
15    operator() (_Up*) const = delete;
16 };

```

Template specialization and deleting a function based on enable_if

Inner class in Unique_ptr class in unique_ptr.h

```

1  /// 20.7.1.2 unique_ptr for single objects.
2  template <typename _Tp, typename _Dp = default_delete<_Tp> >
3  class unique_ptr {
4      // use SFINAE to determine whether _Del::pointer exists
5      class _Pointer {
6
7          template<typename _Up>
8          static typename _Up::pointer __test(typename _Up::pointer*);
9
10         template<typename _Up>
11         static _Tp* __test(...);
12
13         typedef typename remove_reference<_Dp>::type _Del;
14     public:
15         typedef decltype(__test<_Del>(0)) type;
16     };
17

```

Default arguments, Inner classes, SFINAE, decltype

Constructors in Unique_ptr class in unique_ptr.h

```

1  // Constructors.
2  constexpr unique_ptr() noexcept : _M_t()
3  { static_assert(!is_pointer<deleter_type>::value,
4    "constructed with null function pointer deleter"); }
5
6  unique_ptr(pointer __p,
7    typename remove_reference<deleter_type>::type&& __d) noexcept
8  : _M_t(std::move(__p), std::move(__d))
9  { static_assert(!std::is_reference<deleter_type>::value,
10    "rvalue deleter bound to reference"); }
11
12  constexpr unique_ptr(nullptr_t) noexcept : unique_ptr()
13
14  // Move constructors.
15  unique_ptr(unique_ptr&& __u) noexcept
16  : _M_t(__u.release(), std::forward<deleter_type>(__u.get_deleter()))
17

```

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

Operators in Unique_ptr class in unique_ptr.h

```

1 // Destructor.
2 ~unique_ptr() noexcept {
3     auto& __ptr = std::get<0>(_M_t);
4     if (__ptr != nullptr)
5         get_deleter()(__ptr);
6     __ptr = pointer();
7 }
8
9 // Assignment.
10 unique_ptr& operator=(unique_ptr&& __u) noexcept {
11     reset(__u.release());
12     get_deleter() = std::forward<deleter_type>(__u.get_deleter());
13     return *this;
14 }
15
16 unique_ptr& operator=(nullptr_t) noexcept {
17     reset();
18     return *this;
19 }

```

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

Member functions in Unique_ptr class

```
1 void reset(pointer __p = pointer()) noexcept {  
2     using std::swap;  
3     swap(std::get<0>(_M_t), __p);  
4     if (__p != pointer())  
5         get_deleter()(__p);  
6 }  
7  
8 void swap(unique_ptr& __u) noexcept {  
9     using std::swap;  
10    swap(_M_t, __u._M_t);  
11 }  
12  
13 };
```

Member functions, using statements, namespaces, STL algorithms

Reading

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