

COMP6771

Advanced C++ Programming

Week 6

Part Two: Function Templates (continued)

2017

www.cse.unsw.edu.au/~cs6771

Name Resolution

- Two parties involved about a function template:
 - designer
 - client
- **Two-step lookup** in name resolution:
 - **Type-independent names:** resolved at the definition of template – **designer** provides the declarations
 - **Type-dependent names:** resolved at the point of instantiation – **client** provides the declarations
- If there are multiple points, compiler chooses one of these as **the** point of instantiation for the template
⇒ client places all declarations required in header files

Name Resolution

- Template definition:

```
void print(const char* s) { std::cout << s; }
```

```
template <typename T>
```

```
T min(T a, T b) {
```

```
    print("The minimum of the two values: ");
```

```
    T c = (a < b ? a : b);
```

```
    print(c);
```

```
    return c;
```

```
}
```

- Client code:

```
void print(int i) { std::cout << i; }
```

```
min(1, 2); // point of instantiation
```

Important!

Start the name resolution from the Point of Instantiation!

Template Argument Deduction

- Two kinds of template parameters:
 - Template parameters (type or nontype)
 - Call parameters
- **Key observation:** both are related
- **Deduction:** the process of determining the types (of type parameters) and values of nontype parameters from the types of the function arguments
- **The return type is ignored**

Note

No argument deduction for class templates!

Template Argument Deduction

- min:

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

type parameter



call parameters



- Deduction:

```
min(1, 2);      ==> int min(int, int)
min(1.1, 2.2); ==> double min(double, double)
```

- The return type is irrelevant:

```
int i = min(1, 2);    // int min(int, int)
double i = min(1, 2); // int min(int, int)
```

Template Argument Deduction

- findmin:

type parameter

nontype parameter

```
template <typename T, int size>
T findmin(const T (&a)[size]) {
    T min = a[0];
    for (int i = 1; i < size; i++)
        if (a[i] < min) min = a[i];
    return min;
}
```

call parameters

- Deduction:

```
1 int x[] = { 3, 1, 2 };
2 findmin(x); ==> int findmin(const int (&)[3]);
3 double y[] = { 3.3, 1.1, 2.2, 4.4 };
4 findmin(y) ==> double findmin(const int (&)[4]);
```

Limited Conversions for Type Parameter Arguments

- Only three kinds of implicit conversions for type parameters:

- lvalue transformation, e.g.,

```
1 DEF: template <typename T> f(T* array) { }  
2 USE: int a[] = 1, 2; f(a); // array to pointer
```

- qualification conversion (const and volatile)

```
1 DEF: template <typename T> f(const T* array) { }  
2 USE: int a[] = 1, 2;  
3      int *pa = &a; f(pa); // int* => const int*
```

- conversion to a base class from a derived class

```
1 DEF: template <typename T> void f(Base<T> &a) { }  
2 USE: template <typename T>  
3      class Derived : public Base<T> { ... }  
4      Derived<int> d;  
5      f(d);
```

- Usual conversions done for nontype parameters

Explicit Template Arguments

- In template argument deduction, all deduced types for the same template parameter must be the same

```
min(1.0, 2); // compile-time error
```

```
1.0 ==> T is double
```

```
2 ==> T is int
```

- **Explicit template arguments:** override the template argument deduction mechanism by **explicitly** specifying the types of the arguments to be used

```
1 int i; double d;  
2 min(i, static_cast<int>d); // int min(int, int)  
3 min<int>(i, d); // int min(int, int)  
4 min(static_cast<double>i, d); // double min(double, double)  
5 min<double>(i, d); // double min(double, double)
```


Explicit Template Arguments (Pre-C++11)

- Neither T nor U works as return type

```

1 template <typename T, typename U>
2 ??? sum(T, U);
3 sum(3, 4L); // 2nd type is larger ==> want U sum(T, U)
4 sum(3L, 4); // 1st type is larger ==> want T sum(T, U)

```

- Using a type parameter for the return type:

```

1 template <typename T1, typename T2, typename T3>
2     T1 sum(T2, T3);
3 long v = sum<long>(3, 4L); // calls long sum(int, long)
4 long v = sum<long>(3L, 4); // calls long sum(long, int)

```

- The return type T1 cannot be deduced
- T1 must be the 1st **since the explicit arguments are matched to the corresponding template parameters from left to right**

Explicit Template Arguments (C++11)

- Using **decltype** in a trailing return type:

```
1 template <typename T, typename U>
2 auto sum(T a, U b) -> decltype(T{} + U{}) {
3     return a+b;
4 }
```

- Client code:

```
1 int main() {
2     std::cout << sum(3, 4L);    --> long sum(long, long)
3     std::cout << sum(3L, 4);   --> long sum(long, long)
4 }
```

Trailing Return Type Deduction using decltype

From Monday:

```
1 template <typename T, typename U>
2 auto min(T a, U b) -> decltype(a < b ? a : b) {
3     return a < b ? a : b;
4 }
5
6 min(1,0.2); // creates min(int,double)
7 min(1.2,1); // creates min(double,int)
```

- What is the return type of `min(int,double)` and `min(double,int)`?
- A double? An int? Or depends on the values of the int and the double?

Trailing Return Type Deduction using decltype

- Consider:

```
1  ??? min(int a, double b) {  
2      return a < b ? a : b;  
3  }
```

- How does the expression `a < b` work?
- `a` is automatically promoted to a temporary double (a prvalue).
- Therefore regardless of if `a` or `b` is returned, the return type is always a double.
- See:

http://thbecker.net/articles/auto_and_decltype/section_

decltype

```
1 int i;  
2 const &j = i;  
3 int *p = i;  
4 int k;  
5  
6 decltype(i) x;           // int x: i is a variable  
7 decltype(j) y = k;       // int &y = k: j is an lvalue  
8 decltype(*p) z = k;      // int &z = k: *p is an lvalue  
9 decltype((i)) w = k;     // int &w = k: (i) is an lvalue
```

- If the expression *e* refers to a variable in local or namespace scope, a static member variable or a function parameter, then the result is that variable's or parameter's declared type
- Otherwise, if *e* is an lvalue, `decltype(e)` is `T&`, where *T* is the type of *e*; if *e* is an xvalue, the result is `T&&`; otherwise, *e* is a prvalue and the result is *T*.

xvalues

- An rvalue is an xvalue if it is one of the following:
- A function call where the function's return value is declared as an rvalue reference. An example would be `std::move(x)`.
- A cast to an rvalue reference. An example would be `static_cast<A&&>(a)`.
- All other rvalues are prvalues.

C++ String Literals

- The types:
"CSE" ==> const char[4]
"COMP4001" ==> const char[9]
- For backward compatibility with C, a string literal can be assigned to: const char*
- **Problem:** what is the output of this program

```
1 #include <iostream>
2
3 template <typename T>
4 T min(T a, T b) { return a < b ? a : b; }
5
6 int main() {
7     const char *s = "zyx";
8     const char *t = "abc";
9     std::cout << min(s,t) << std::endl;
10 }
```

- (a) Won't compile as s and t are different types, or
(b) abc or (c) zyx?

Specialisation

- Needed for correctness or efficiency reasons or both
- The semantics of min for **char*** are wrong:

```
1 template <typename T>
2     T min(T a, T b) {
3         return a < b ? a : b;
4     }
```

- The correct version:

```
1 typedef const char* PCC;
2 template<> PCC min<PCC>(PCC a, PCC b) {
3     return (strcmp(a, b) < 0) ? a : b;
4 }
```

- Exact match required (no conversions for the arguments)
- Client code:

```
const char* s1 = "xyz";
const char* s2 = "abc";
min(s1, s2);
```


Specialisation – File Organisation

```
1 // min.h:
2 template <typename T>
3     T min(T a, T b) {
4         return a < b ? a : b;
5     }
6
7 typedef const char *PCC;
8 template<> PCC min<PCC>(PCC a, PCC b) {
9     return (strcmp(a, b) < 0) ? a : b;
10 }
11
12 // min-user.cpp:
13 #include "min.h"
14 // ...
```

- Alternatively, can put the specialisation in the .cpp file that requires it.
- But, this may cause issues if the specialisation is required in another file and it isn't found!

Function Template Overloading

```

1  #include <iostream>
2  #include <cstring>
3
4  template <typename T>
5  T min(T a, T b) {
6      return a < b ? a : b;
7  }
8
9  typedef const char *PCC;
10 template<> PCC min<PCC>(PCC a, PCC b) {
11     return (strcmp(a, b) < 0) ? a : b;
12 }
13
14 double min(double a, double b) {
15     return a < b ? a : b;
16 }
17
18 int main() {
19     const char *s = "zyx";
20     const char *t = "abc";
21
22     std::cout << min(1, 2) << std::endl;           // template
23     std::cout << min(1.1, 2.2) << std::endl;        // ordinary
24     std::cout << min(1, 2.2) << std::endl;          // ordinary
25     std::cout << min(s, t) << std::endl;            // specialisation
26 }

```

Resolution rules: more specialised functions are preferred

- <http://accu.org/index.php/journals/268>

- Text. §16.3

Template Default Arguments (C++11)

```
1  #include <iostream>
2  #include <functional>
3
4  template <typename T, typename Pred=std::less<T>>
5  T minTD(T a, T b, Pred cmp = Pred()) {
6      return cmp(a, b) ? a : b;
7  }
8
9  bool cmpstr(std::string a, std::string b) {
10     return a.size() < b.size();
11 };
12
13 int main() {
14     std::cout << minTD(1,2) << std::endl;
15     std::string s1 = "xyz";
16     std::string s2 = "abc";
17     std::cout << minTD(s1, s2, cmpstr) << std::endl;
18     // pass in a lamda.
19     std::cout << minTD(s1, s2, [](std::string a, std::string b) {
20         return a.size() < b.size(); })
21     ) << std::endl;
22 }
```

A Variadic Function Template

```
1  #include <iostream>
2  #include <typeinfo>
3
4  template <typename T>
5  void print(const T& msg) {
6      std::cout << msg << " ";
7  }
8
9  template <typename A, typename... B>
10 void print(A head, B... tail) {
11     print(head);
12     print(tail...);
13 }
14
15 int main() {
16     print(1, 2.0f);
17     std::cout << std::endl;
18     print(1, 2.0f, "Hello");
19     std::cout << std::endl;
20 }
```

Output:

```
1  1 2
2  1 2 Hello
```

The Instantiations of `print(1, 2.0f, "Hello")`

```
1 void print(const char* const &c) {  
2     std::cout << c << " ";  
3 }  
4  
5 void print(const float &b) {  
6     std::cout << b << " ";  
7 }  
8  
9 void print(float b, const char* c) {  
10     print(b);  
11     print(c);  
12 }  
13  
14 void print(const int &a) {  
15     std::cout << a << " ";  
16 }  
17  
18 void print(int a, float b, const char* c) {  
19     print(a);  
20     print(b, c);  
21 }
```

Reading

- Chapter 16, C++ Primer:
- Chapter 5, Bruce Eckel, Thinking in C++, Vol 2
- C++ FAQ Lite: templates:
<http://yosefk.com/c++faq/templates.html>

Assn 3 is harder than Assns 1 and 2. So start working on it earlier.

Next Class: Class Templates