COMP6771
Advanced C++ Programming

Week 6
Part Two: Function Templates (continued)

2016

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
  $\Longrightarrow$ 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:

type parameter

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

call parameters

- Deduction:

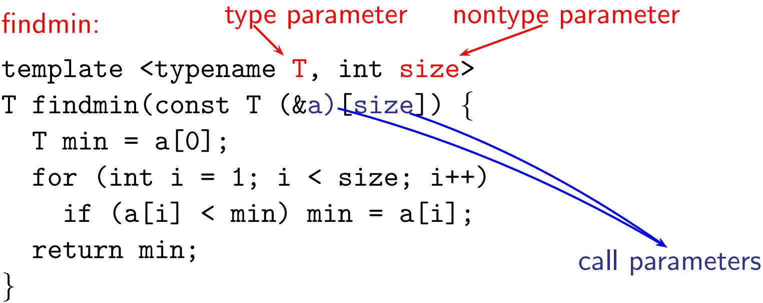
```
min(1, 2);     ==> int min(int, int)
min(1.1, 2.2); ==> double min(double, double)
```

- 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

call parameters

```
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;
}
```

- Deduction:

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

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

  - qualification conversion (const and volatile)

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

  - conversion to a base class from a derived class

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

- Usual conversions done for nontype parameters

Explicit Template Arguments

- In template argument deduction, all deducted 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

```
int i;  double d;
min(i, static_cast<int>d);     // int min(int, int)
min<int>(i, d);                // int min(int, int)
min(static_cast<double>i, d); // double min(double, double)
min<double>(i, d);             // double min(double, double)
```

Explicit Template Arguments (Pre-C++11)

- Neither T nor U works as return type

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

- Using a type parameter for the return type:

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

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

- Client code:

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

Trailing Return Type Deduction using decltype

From Monday:

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

min(1,0.2);  // creates min(int,double)
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:

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

- 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_07.html

decltype

```
int i;
const &j = i;
int *p = i;
int k;

decltype(i) x;       // int x: i is a variable
decltype(j) y = k;   // int &y = k: j is an lvalue
decltype(*p) z = k;  // int &z = k: *p is an lvalue
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

```
#include <iostream>

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

int main() {
  const char *s = "zyx";
  const char *t = "abc";
  std::cout << min(s,t) << std::endl;
}
```

- (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:

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

- The correct version:

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

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

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

Specialisation – File Organisation

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

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

// min-user.cpp:
#include "min.h"
// ...
```

- 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

```
#include <iostream>
#include <cstring>

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

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

double min(double a, double b) {
  return a < b ? a : b;
}

int main() {
  const char *s = "zyx";
  const char *t = "abc";

  std::cout << min(1, 2) << std::endl;       // template
  std::cout << min(1.1, 2.2) << std::endl;   // ordinary
  std::cout << min(1, 2.2) << std::endl;     // ordinary
  std::cout << min(s, t) << std::endl;     // specialisation
}
```

Resolution rules: more specialised functions are preferred

- http://accu.org/index.php/journals/268
- Text, §16.3

Template Default Arguments (C++11)

```
#include <iostream>
#include <functional>

template <typename T, typename Pred=std::less<T>>
T minTD(T a, T b, Pred cmp = Pred()) {
    return cmp(a, b) ? a : b;
}

bool cmpstr(std::string a, std::string b) {
  return a.size() < b.size();
};

int main() {
  std::cout << minTD(1,2) << std::endl;
  std::string s1 = "xyz";
  std::string s2 = "abc";
  std::cout << minTD(s1, s2, cmpstr) << std::endl;
  // pass in a lamda.
  std::cout << minTD(s1, s2, [](std::string a, std::string b) {
    return a.size() < b.size(); }
  ) << std::endl;
}
```

A Variadic Function Template

```
#include <iostream>
#include <typeinfo>

template <typename T>
void print(const T& msg) {
  std::cout << msg << " ";
}

template <typename A, typename... B>
void print(A head, B... tail) {
  print(head);
  print(tail...);
}

int main() {
  print(1, 2.0f);
  std::cout << std::endl;
  print(1, 2.0f, "Hello");
  std::cout << std::endl;
}
```

Output:

```
1 2
1 2 Hello
```

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

```
void print(const char* const &c) {
  std::cout << c << " ";
}

void print(const float &b) {
  std::cout << b << " ";
}

void print(float b, const char* c) {
  print(b);
  print(c);
}

void print(const int &a) {
  std::cout << a << " ";
}

void print(int a, float b, const char* c) {
  print(a);
  print(b, c);
}
```

Reading

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

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

Next Class: Class Templates