COMP6771
Advanced C++ Programming

Week 7
Part Two: Type Traits

2016

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

Type Traits

- A trait is a class or class template that characterises a type.
- Traits represent additional properties of a template parameter.

```
#include <iostream>
#include <limits>

int main() {
  std::cout << std::numeric_limits<double>::min() << std::endl;
  std::cout << std::numeric_limits<int>::min() << std::endl;
  // ...
}
```

- The type trait library: `limits`, looks something like this:

```
template <typename T> struct numeric_limits {
  static T min();
};

template <> struct numeric_limits<int> {
  static int min() { return -__INT_MAX__ - 1; }
};

template <> struct numeric_limits<float> {
  static float min() {  return __FLT_MIN__; }
};
```

Type Traits cont.

- Traits allow generic algorithms to be parameterised
- Consider trying to find the maximum value in an array of different types:

```
#include <iostream>
#include "numlimits.hpp"    // in reality: <limits>

template <typename T>
T findMax(const T* data, int numItems) {
  // Get the minimum value for type T using type trait
  T currLargest = numeric_limits<T>::min();

  for (int i=0; i < numItems; ++i)
    if (data[i] > currLargest)
      currLargest = data[i];

  return currLargest;
}

int main() {
  int iArray[] = {-1,-3,-2};
  std::cout << findMax(iArray,3) << std::endl;
  unsigned int iUArray[] = {1,3,2};
  std::cout << findMax(iUArray,3) << std::endl;
  float fArray[] = {4.1,4.3,4.2};
  std::cout << findMax(fArray,3) << std::endl;
}
```

is_void Example

- Determining if a template parameter is void

```
#include <iostream>

template <typename T> struct is_void {
  static const bool val = false;
};

template<> struct is_void <void> {
  static const bool val = true;
};

int main() {
  std::cout << is_void<int>::val << std::endl;
  std::cout << is_void<void>::val << std::endl;
}
```

is_pointer Example

- Determining if a template parameter is a pointer:

```
#include <iostream>

template <typename T> struct is_ptr {
  static const bool val = false;
};

template <typename T> struct is_ptr<T*> {
  static const bool val = true;
};

int main() {
  std::cout << is_ptr<int*>::val << std::endl;
  std::cout << is_ptr<int>::val << std::endl;
}
```

STL type_traits

Many type trait classes and functions:

is_void	checks if a type is void
is_integral	checks if a type is integral type
is_floating_point	checks if a type is floating-point type
is_class	checks if a type is a class type
is_pointer	checks if a type is a pointer type
is_lvalue_reference	checks if a type is lvalue reference
is_rvalue_reference	checks if a type is rvalue reference

http://en.cppreference.com/w/cpp/header/type_traits

Using type_traits Example

Using type_traits to control program flow in template function:

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

template <typename T>
void testIfNumberType(T i) {
  if (std::is_integral<T>::value || std::is_floating_point<T>::value) {
    std::cout << i << " is a number" << std::endl;
  } else {
    std::cout << i << " is not a number" << std::endl;
  }
}

int main() {
  int i = 6;
  long l = 7;
  double d = 3.14;
  testIfNumberType(i);
  testIfNumberType(l);
  testIfNumberType(d);
  testIfNumberType(123);
  testIfNumberType("Hello");
  std::string s = "World";
  testIfNumberType(s);
}
```

Type traits and `decltype(e)`

- 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 (expiring value, used for std::move), the result is `T&&`; otherwise, e is a prvalue and the result is T.

```
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

```

Type Transformation

Consider a function that uses an iterator over a templated collection and returns a reference from this collection

```
template <typename It>
??? fcn (It beg, It end) {
  // loop through the range
  return *beg; // return a reference to an element
}
```

What should the type of ??? be?

decltype and Trailing Return Types

- We know that our function takes in a It beg
- And returns a *beg
- We can use decltype(*beg) to work out the type of *beg
- However, this doesn't work as beg isn't declared until after the return type.

```
template <typename It>
decltype(*beg) fcn (It beg, It end) {
  // loop through the range
  return *beg; // return a reference to an element
}
```

- A trailing return type allows us to make this function work.

```
template <typename It>
auto fcn (It beg, It end) -> decltype(*beg) {
  // loop through the range
  return *beg; // return a reference to an element
}
```

Type Transformation

- What about if we wanted to return a copy of the data (rather than a reference)?
- We can do this using the type transformation library:

```
  template <typename It>
  auto fcn (It beg, It end)
-> typename remove_reference<decltype(*beg)>::type {
    // loop through the range
    return *beg; // return a reference to an element
  }
```

Type Transformation Library Template Classes

For *Mod*<T>, where *Mod* is	If T is	Then *Mod*<T>::type is
remove_reference	X& or X&& otherwise	X T
add_const	X&, const X, or function otherwise	T const T
add_lvalue_reference	X& X&& otherwise	T X& T&
add_rvalue_reference	X& or X&& otherwise	T T&&
remove_pointer	X* otherwise	X T
add_pointer	X& or X&& otherwise	X* T*
make_signed	unsigned X otherwise	X T
make_unsigned	signed type otherwise	unsigned T T
remove_extent	X[n] otherwise	X T
remove_all_extents	X[n1][n2]... otherwise	X T

remove_reference

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

template<typename T1, typename T2>
void print_is_same() {
  std::cout << std::is_same<T1, T2>() << std::endl;
}

int main() {
  std::cout << std::boolalpha;

  print_is_same<int, int>();    // true
  print_is_same<int, int &>();  // false
  print_is_same<int, int &&>(); // false

  print_is_same<int, std::remove_reference<int>::type>();    // true
  print_is_same<int, std::remove_reference<int &>::type>();  // true
  print_is_same<int, std::remove_reference<int &&>::type>(); // true
  print_is_same<const int, std::remove_reference<const int &&>::type>(); // true
}
```

add_rvalue_reference

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

int main() {
  typedef std::add_rvalue_reference<int>::type A;    // int&&
  typedef std::add_rvalue_reference<int&>::type B;   // int&  (no change)
  typedef std::add_rvalue_reference<int&&>::type C;  // int&& (no change)
  typedef std::add_rvalue_reference<int*>::type D;   // int*&&

  std::cout << std::boolalpha;
  std::cout << "typedefs of int&&:" << std::endl;
  std::cout << "A: " << std::is_same<int&&,A>::value << std::endl;
  std::cout << "B: " << std::is_same<int&&,B>::value << std::endl;
  std::cout << "C: " << std::is_same<int&&,C>::value << std::endl;
  std::cout << "D: " << std::is_same<int&&,D>::value << std::endl;
}
```

Parameter Binding and Overload Resolution Rules

	Expression			
Reference Type	rvalue	const rvalue	lvalue	const lvalue
T&&	yes			
const T&&	yes	yes		
T&			yes	
const T&	yes	yes	yes	yes

- `const T&` binds to everything (that is why we use it a lot)
- `T&&` binds only to non-const rvalues (these are the objects that we typically move from)

An Example Illustrating Binding for `const &`

`const &` binds to everything:

```
#include <iostream>

void foo(const std::string &a) { }

const std::string goo() { return "C++"; }

int main() {
  foo("C++");                    // rvalue
  foo(goo());                    // const rvalue
  std::string j = "C++";
  foo(j);                        // lvalue
  const std::string &k = "C++";
  foo(k);                        // const lvalue
}
```

New Binding and Overload Resolution Rules

```
template<typename T> void foo(T&& a);
```

Template rvalue references parameters binds to everything!

	Expression			
Reference Type	rvalue	const rvalue	lvalue	const lvalue
template T&&	yes	yes	yes	yes
T&&	yes			
const T&&	yes	yes		
T&			yes	
const T&	yes	yes	yes	yes

Reference Collapsing

Consider: `foo(b);`

- If `b` is an [const] lvalue of type A, Ts type is deduced as `[const] A&`. The argument type becomes `[const] A&`
- If b is an [const] rvalue of type A, Ts type is deduced as `[const] A`. The argument type becomes `[const]A&&`

T is a type:

- T& & becomes T&
- T& && becomes T&
- T&& & becomes T&
- T&& && becomes T&&

See: http://thbecker.net/articles/rvalue_references/section_08.html

An Example for Template Rvalue Parameters

Four versions of the template generated!

```
#include<iostream>

template <typename T> void foo(T &&a) { }

class X ;
const X goo() { return X(); }

int main() {
  foo(1);                    // rvalue
                             // instantiate foo(int&&)

  foo(goo());                // const rvalue
                             // instantiate foo(const X&&)
  int j = 1;
  foo(j);                    // lvalue
                             // instantiate foo(int&)
  const int &k = 1;
  foo(k);                    // const lvalue
                             // instantiate foo(const int&)
}
```

Example Problem

- Write a method that takes two templated values by reference, increments them and prints them.
- The body of the function looks like:

```
template <typename T1, typename T2>
void addAndPrint(??? t1, ??? t2) {
    std::cout << "in func: " << ++t1 << " " << ++t2 << std::endl;
}
```

- The following code can be used to test the method:

```
int main() {
  int a = 0, b = 100;
  std::cout << "in main: " << a << " " << b << " " << std::endl;
  addAndPrint(a,b);
  std::cout << "in main: " << a << " " << b << " " << std::endl;
  addAndPrint(1,b);
  addAndPrint(2,200);
}
```

The output should be:

```
in main: 0 100
in func: 1 101
in main: 1 101
in func: 2 102
in func: 3 201
```

Wrong solutions

```cpp
template <typename T1, typename T2>
void addAndPrint(T1 t1, T2 t2) {
  std::cout << ++t1 << " " << ++t2 << std::endl;
}
```

This method is not call by reference so the values will not be incremented outside of the function. Output:

```
in main: 0 100
in func: 1 101
in main: 0 100
in func: 2 101
in func: 3 201
```

Wrong solutions

```
template <typename T1, typename T2>
void addAndPrint(T1 &t1, T2 &t2) {
  std::cout << ++t1 << " " << ++t2 << std::endl;
}
```

Now `addAndPrint(1,b);` and `addAndPrint(2,200);` won't compile.

```
g++ -std=c++11 -Wall -Werror -O2 -o addAndPrint addAndPrint.cpp
addAndPrint.cpp: In function int main():
addAndPrint.cpp:13:20: error: invalid initialization of non-const reference of type
int& from an rvalue of type int
     addAndPrint(1,b);
                    ^
addAndPrint.cpp:4:6: note: in passing argument 1 of void addAndPrint(T1&, T2&)
[with T1 = int; T2 = int]
 void addAndPrint(T1& t1, T2& t2) {
      ^
addAndPrint.cpp:14:22: error: invalid initialization of non-const reference of type
int& from an rvalue of type int
     addAndPrint(2,200);
                      ^
addAndPrint.cpp:4:6: note: in passing argument 1 of void addAndPrint(T1&, T2&)
[with T1 = int; T2 = int]
 void addAndPrint(T1& t1, T2& t2) {
      ^
make: *** [addAndPrint] Error 1
```

Solution

```
template <typename &&T1, typename &&T2>
void addAndPrint(T1 &&t1, T2 &&t2) {
  std::cout << ++t1 << " " << ++t2 << std::endl;
}
```

This will work because of reference collapsing.

std::forward

```
template<typename T>
T&& forward(typename remove_reference<T>::type& a) noexcept {
  return static_cast<T&&>(a);
}
```

`forward<T>(a)` is equivalent to:

- `static_cast<[const]T&&>(a)` when a is an rvalue
- `static_cast<[const]T&>(a)` when a is an lvalue

Perfect Forwarding: lvalues stay as lvalues and rvalues stay as rvalues and constness is preserved

Perfect Forwarding

Perfect forwarding allows a function template to pass its arguments through to another function while retaining the original lvalue/rvalue/const nature of the function arguments. It avoids unnecessary copying and avoids the programmer having to write multiple overloads for different combinations of lvalue and rvalue references.

Perfect Forwarding: An Example

```cpp
#include <utility>
#include <iostream>

// function with lvalue and rvalue reference overloads:
void overloaded (const int& x) { std::cout << "[lvalue]"; }
void overloaded (int&& x) { std::cout << "[rvalue]"; }

template <class T> void foo(T&& x) {
  overloaded (x);                    // always an lvalue
  overloaded (std::forward<T>(x));  // rvalue if argument is rvalue
}

int main () {
  int a;
  std::cout << "calling foo with lvalue: ";
  foo(a);
  std::cout << std::endl;
  std::cout << "calling foo with rvalue: ";
  foo(0);
  std::cout << std::endl;
}
```

Output:

```
calling foo with lvalue: [lvalue][lvalue]
calling foo with rvalue: [lvalue][rvalue]
```

Understanding `std::move`

```
template <typename T>
typename remove_reference<T>::type&&
move(T&& t) noexcept {
  return static_cast<typename remove_reference<T>::type&&>(t);
}

std::string s1("C++"), s2;
s2 = std::move(std::string("C++11"));
s2 = std::move(s1);
```

s2 = std::move(std::string("C++11"))

1. The deduced type of T is std::string
2. std::remove_reference<std::string>::type is std::string
3. The return type of move is std::string&&
4. The argument type of move is std::string&&
5. std::static_cast<std::string&&>(t)
6. The move is instantiated as:

   ```
   std::string&& move(std::string &&t)
   ```
7. The call is equivalent to:

   ```
   s2 =
   std::static_cast<std::string&&>(std::string("C++11"));
   ```

s2 = std::move(s1)

1. The deduced type of T is std::string&
2. std::remove_reference<std::string&>::type is std::string
3. The return type of move is std::string&&
4. The argument type of move is std::string& &&, i.e., std::string &
5. std::static_cast<std::string&&>(t)
6. The move is instantiated as:

   ```
   std::string&& move(std::string &t)
   ```

7. The call is equivalent to:

   ```
   s2 = std::static_cast<std::string&&>(s1)
   ```

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);
}
```

Perfect Forwarding

Perfect forwarding allows a function template to pass its arguments through to another function while retaining the original lvalue/rvalue/const nature of the function arguments. It avoids unnecessary copying and avoids the programmer having to write multiple overloads for different combinations of lvalue and rvalue references. A class can use perfect forwarding with variadic templates to "export" all possible constructors of a member object at the parent's level.

Variadic Templates

```
#include <iostream>
#include <vector>

class Blob {
  std::vector<std::string> _v;
public:
   // variadic templated constructor
   template<typename... Args>
   Blob(Args&&... args) : _v(std::forward<Args>(args)...) {  }
};

int main(void) {
  const char * shapes[3] = { "Circle", "Triangle", "Square" };

  Blob b1(10, "C++11");  // uses vector's fill constructor
  Blob b2(shapes, shapes+3); // uses vector's range constructor
}
```

Reading

- Chapter 16
- C++ Rvalue References Explained: http://thbecker.net/articles/rvalue_references/section_01.html