

# COMP6771

## Advanced C++ Programming

### Week 7

### Part Two: Type Traits

2016

[www.cse.unsw.edu.au/~cs6771](http://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.

```
1 #include <iostream>
2 #include <limits>
3
4 int main() {
5     std::cout << std::numeric_limits<double>::min() << std::endl;
6     std::cout << std::numeric_limits<int>::min() << std::endl;
7     // ...
8 }
```

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

```
1 template <typename T> struct numeric_limits {
2     static T min();
3 };
4
5 template <> struct numeric_limits<int> {
6     static int min() { return -__INT_MAX__ - 1; }
7 };
8
9 template <> struct numeric_limits<float> {
10     static float min() { return __FLT_MIN__; }
11 };
```

## Type Traits cont.

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

```
1  #include <iostream>
2  #include "numlimits.hpp"    // in reality: <limits>
3
4  template <typename T>
5  T findMax(const T* data, int numItems) {
6      // Get the minimum value for type T using type trait
7      T currLargest = numeric_limits<T>::min();
8
9      for (int i=0; i < numItems; ++i)
10         if (data[i] > currLargest)
11             currLargest = data[i];
12
13     return currLargest;
14 }
15
16 int main() {
17     int iArray[] = {-1,-3,-2};
18     std::cout << findMax(iArray,3) << std::endl;
19     unsigned int iUArray[] = {1,3,2};
20     std::cout << findMax(iUArray,3) << std::endl;
21     float fArray[] = {4.1,4.3,4.2};
22     std::cout << findMax(fArray,3) << std::endl;
23 }
```

## is\_void Example

- Determining if a template parameter is void

```
1 #include <iostream>
2
3 template <typename T> struct is_void {
4     static const bool val = false;
5 };
6
7 template<> struct is_void <void> {
8     static const bool val = true;
9 };
10
11 int main() {
12     std::cout << is_void<int>::val << std::endl;
13     std::cout << is_void<void>::val << std::endl;
14 }
```

## is\_pointer Example

- Determining if a template parameter is a pointer:

```
1 #include <iostream>
2
3 template <typename T> struct is_ptr {
4     static const bool val = false;
5 };
6
7 template <typename T> struct is_ptr<T*> {
8     static const bool val = true;
9 };
10
11 int main() {
12     std::cout << is_ptr<int*>::val << std::endl;
13     std::cout << is_ptr<int>::val << std::endl;
14 }
```

## 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](http://en.cppreference.com/w/cpp/header/type_traits)

## Using type\_traits Example

Using type\_traits to control program flow in template function:

```
1  #include <iostream>
2  #include <type_traits>
3
4  template <typename T>
5  void testIfNumberType(T i) {
6      if (std::is_integral<T>::value || std::is_floating_point<T>::value) {
7          std::cout << i << " is a number" << std::endl;
8      } else {
9          std::cout << i << " is not a number" << std::endl;
10     }
11 }
12
13 int main() {
14     int i = 6;
15     long l = 7;
16     double d = 3.14;
17     testIfNumberType(i);
18     testIfNumberType(l);
19     testIfNumberType(d);
20     testIfNumberType(123);
21     testIfNumberType("Hello");
22     std::string s = "World";
23     testIfNumberType(s);
24 }
```

## 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*.

```
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  
10
```



# Type Transformation

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

```
1  template <typename It>
2  ??? fcn (It beg, It end) {
3      // loop through the range
4      return *beg; // return a reference to an element
5  }
```

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.

```
1  template <typename It>
2      decltype(*beg) fcn (It beg, It end) {
3      // loop through the range
4      return *beg; // return a reference to an element
5  }
```

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

```
1  template <typename It>
2      auto fcn (It beg, It end) -> decltype(*beg) {
3      // loop through the range
4      return *beg; // return a reference to an element
5  }
```

# 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:

```
1  template <typename It>
2  auto fcn (It beg, It end)
3  -> typename remove_reference<decltype(*beg)>::type {
4      // loop through the range
5      return *beg; // return a reference to an element
6  }
```

# Type Transformation Library Template Classes

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

## remove\_reference

```
1 #include <iostream>
2 #include <type_traits>
3
4 template<typename T1, typename T2>
5 void print_is_same() {
6     std::cout << std::is_same<T1, T2>() << std::endl;
7 }
8
9 int main() {
10     std::cout << std::boolalpha;
11
12     print_is_same<int, int>(); // true
13     print_is_same<int, int &>(); // false
14     print_is_same<int, int &&>(); // false
15
16     print_is_same<int, std::remove_reference<int>::type>(); // true
17     print_is_same<int, std::remove_reference<int &>::type>(); // true
18     print_is_same<int, std::remove_reference<int &&>::type>(); // true
19     print_is_same<const int, std::remove_reference<const int &&>::type>(); // true
20 }
```

## add\_rvalue\_reference

```
1  #include <iostream>
2  #include <type_traits>
3
4  int main() {
5      typedef std::add_rvalue_reference<int>::type A;    // int&&
6      typedef std::add_rvalue_reference<int&>::type B;   // int&  (no change)
7      typedef std::add_rvalue_reference<int&&>::type C;  // int&& (no change)
8      typedef std::add_rvalue_reference<int*>::type D;   // int*&&
9
10     std::cout << std::boolalpha;
11     std::cout << "typedefs of int&&:" << std::endl;
12     std::cout << "A: " << std::is_same<int&&,A>::value << std::endl;
13     std::cout << "B: " << std::is_same<int&&,B>::value << std::endl;
14     std::cout << "C: " << std::is_same<int&&,C>::value << std::endl;
15     std::cout << "D: " << std::is_same<int&&,D>::value << std::endl;
16 }
```

## 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:

```
1  #include <iostream>
2
3  void foo(const std::string &a) { }
4
5  const std::string goo() { return "C++"; }
6
7  int main() {
8      foo("C++");           // rvalue
9      foo(goo());           // const rvalue
10     std::string j = "C++";
11     foo(j);                // lvalue
12     const std::string &k = "C++";
13     foo(k);                // const lvalue
14 }
```



# New Binding and Overload Resolution Rules

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

Template rvalue references parameters binds to everything!

Reference Type	Expression			
	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`, `T`'s type is deduced as `[const] A&`. The argument type becomes `[const] A&`
- If `b` is an `[const]` rvalue of type `A`, `T`'s 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](http://thbecker.net/articles/rvalue_references/section_08.html)

## An Example for Template Rvalue Parameters

Four versions of the template generated!

```
1  #include<iostream>
2
3  template <typename T> void foo(T &&a) { }
4
5  class X ;
6  const X goo() { return X(); }
7
8  int main() {
9      foo(1);                                // rvalue
10                                         // instantiate foo(int&&)
11
12     foo(goo());                            // const rvalue
13                                         // instantiate foo(const X&&)
14     int j = 1;
15     foo(j);                                // lvalue
16                                         // instantiate foo(int&)
17     const int &k = 1;
18     foo(k);                                // const lvalue
19                                         // instantiate foo(const int&)
20 }
```

## Example Problem

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

```
1 template <typename T1, typename T2>
2 void addAndPrint(??? t1, ??? t2) {
3     std::cout << "in func: " << ++t1 << " " << ++t2 << std::endl;
4 }
```

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

```
1 int main() {
2     int a = 0, b = 100;
3     std::cout << "in main: " << a << " " << b << " " << std::endl;
4     addAndPrint(a,b);
5     std::cout << "in main: " << a << " " << b << " " << std::endl;
6     addAndPrint(1,b);
7     addAndPrint(2,200);
8 }
```

The output should be:

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

## Wrong solutions

```
1  template <typename T1, typename T2>
2  void addAndPrint(T1 t1, T2 t2) {
3      std::cout << ++t1 << " " << ++t2 << std::endl;
4  }
```

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

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

## Wrong solutions

```
1  template <typename T1, typename T2>
2  void addAndPrint(T1 &t1, T2 &t2) {
3      std::cout << ++t1 << " " << ++t2 << std::endl;
4  }
```

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

```
1  g++ -std=c++11 -Wall -Werror -O2 -o addAndPrint addAndPrint.cpp
2  addAndPrint.cpp: In function int main():
3  addAndPrint.cpp:13:20: error: invalid initialization of non-const reference of type
4  int& from an rvalue of type int
5      addAndPrint(1,b);
6      ^
7  addAndPrint.cpp:4:6: note: in passing argument 1 of void addAndPrint(T1&, T2&)
8  [with T1 = int; T2 = int]
9  void addAndPrint(T1& t1, T2& t2) {
10     ^
11  addAndPrint.cpp:14:22: error: invalid initialization of non-const reference of type
12  int& from an rvalue of type int
13      addAndPrint(2,200);
14      ^
15  addAndPrint.cpp:4:6: note: in passing argument 1 of void addAndPrint(T1&, T2&)
16  [with T1 = int; T2 = int]
17  void addAndPrint(T1& t1, T2& t2) {
18     ^
19  make: *** [addAndPrint] Error 1
```

## Solution

```
1  template <typename &&T1, typename &&T2>
2  void addAndPrint(T1 &&t1, T2 &&t2) {
3      std::cout << ++t1 << " " << ++t2 << std::endl;
4  }
```

This will work because of reference collapsing.

## std::forward

```
1 template<typename T>
2 T&& forward(typename remove_reference<T>::type& a) noexcept {
3     return static_cast<T&&>(a);
4 }
```

`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

```
1 #include <utility>
2 #include <iostream>
3
4 // function with lvalue and rvalue reference overloads:
5 void overloaded (const int& x) { std::cout << "[lvalue]"; }
6 void overloaded (int&& x) { std::cout << "[rvalue]"; }
7
8 template <class T> void foo(T&& x) {
9     overloaded (x); // always an lvalue
10    overloaded (std::forward<T>(x)); // rvalue if argument is rvalue
11 }
12
13 int main () {
14     int a;
15     std::cout << "calling foo with lvalue: ";
16     foo(a);
17     std::cout << std::endl;
18     std::cout << "calling foo with rvalue: ";
19     foo(0);
20     std::cout << std::endl;
21 }
```

### Output:

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

## Understanding std::move

```
1 template <typename T>
2 typename remove_reference<T>::type&&
3 move(T&& t) noexcept {
4     return static_cast<typename remove_reference<T>::type&&>(t);
5 }
6
7 std::string s1("C++"), s2;
8 s2 = std::move(std::string("C++11"));
9 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)`

- ❶ The deduced type of `T` is `std::string&`
- ❷ `std::remove_reference<std::string&>::type` is `std::string`
- ❸ The return type of `move` is `std::string&&`
- ❹ The argument type of `move` is `std::string& &&`, i.e., `std::string &`
- ❺ `std::static_cast<std::string&&>(t)`
- ❻ The move is instantiated as:  
`std::string&& move(std::string &t)`
- ❼ The call is equivalent to:  
`s2 = std::static_cast<std::string&&>(s1)`

## 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 }
```

## 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

```
1 #include <iostream>
2 #include <vector>
3
4 class Blob {
5     std::vector<std::string> _v;
6 public:
7     // variadic templated constructor
8     template<typename... Args>
9     Blob(Args&&... args) : _v(std::forward<Args>(args)...) { }
10 };
11
12 int main(void) {
13     const char * shapes[3] = { "Circle", "Triangle", "Square" };
14
15     Blob b1(10, "C++11"); // uses vector's fill constructor
16     Blob b2(shapes, shapes+3); // uses vector's range constructor
17 }
```

# Reading

- Chapter 16
- C++ Rvalue References Explained: [http://thbecker.net/articles/rvalue\\_references/section\\_01.html](http://thbecker.net/articles/rvalue_references/section_01.html)