COMP6771 Week 8.2

Advanced Types

decltype

decltype(e)

- Semantic equivalent of a "typeof" function for C++
- **Rule 1**:
  - If expression **e** is any of:
    - variable in local scope
    - variable in namespace scope
    - static member variable
    - function parameters
  - then result is variable/parameters type **T**
- **Rule 2**: if e is an lvalue, result is T&
- **Rule 3**: if e is an xvalue, result is T&&
- **Rule 4**: if e is a prvalue, result is T

Non-simplified set of rules can be found here.

decltype

Examples include:

```
1 int i;
2 int j& = i;
3
4 decltype(i) x; // int; - variable
5 decltype(j) y; // int& - lvalue
6 decltype(5);   // int - prvalue
```

Determining return types

Iterator used over templated collection and returns a reference to an item at a particular index

```
1 template <typename It>
2 ??? find(It beg, It end, int index) {
3   for (auto it = beg, int i = 0; beg != end; ++it; ++i) {
4     if (i == index) {
5       return *it;
6     }
7   }
8   return end;
9 }
```

We know the return type should be **decltype(*beg),** since we know the type of what is returned is of type *beg

Determining return types

This will not work, as beg is not declared until after the reference to beg

```
1 template <typename It>
2 decltype(*beg) find(It beg, It end, int index) {
3   for (auto it = beg, int i = 0; beg != end; ++it; ++i) {
4     if (i == index) {
5       return *it;
6     }
7   }
8   return end;
9 }
```

Introduction of C++11 **Trailing Return Types** solves this problem for us

```
1 template <typename It>
2 auto find(It beg, It end, int index) -> decltype(*beg) {
3   for (auto it = beg, int i = 0; beg != end; ++it; ++i) {
4     if (i == index) {
5       return *it;
6     }
7   }
8   return end;
9 }
```

Type Transformations

A number of **add, remove**, and **make** functions exist as part of type traits that provide an ability to transform types

Type Transformations

A number of **add, remove**, and **make** functions exist as part of type traits that provide an ability to transform types

```cpp
#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
}
```

Type Transformations

A number of **add, remove**, and **make** functions exist as part of type traits that provide an ability to transform types

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

int main() {
  typedef std::add_rvalue_reference<int>::type A;
  typedef std::add_rvalue_reference<int&>::type B;
  typedef std::add_rvalue_reference<int&&>::type C;
  typedef std::add_rvalue_reference<int*>::type D;

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

Binding

	lvalue	**const lvalue**	**rvalue**	**const value**
template T&&	Yes	Yes	Yes	Yes
T&	Yes			
const T&	Yes	Yes	Yes	Yes
T&&			Yes	

Note:

- const T& binds to everything!
- template T&& binds to everything!
  - template <typename T> void foo(T&& a);

Examples

```cpp
#include <iostream>

void print(const std::string& a) {
  std::cout << a << "\n";
}

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

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

```cpp
#include <iostream>

template <typename T>
void print(T&& a) {
  std::cout << a << "\n";
}

const std::string goo() {
  return 5;
}

int main() {
  int j = 1;
  const int &k = 1;

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

Forwarding functions

- What's wrong with this?

```
1 template <typename T>
2 auto wrapper(T value) {
3   return fn(value);
4 }
```

- What can we do about it?

Forwarding functions

- What's wrong with this?

```
1 template <typename T>
2 auto wrapper(T value) {
3   return fn(value);
4 }
```

- What can we do about it?
  - Pass in a reference?

Forwarding functions

- This solves our previous problem

```
1 template <typename T>
2 auto wrapper(const T& value) {
3   return fn(value);
4 }
```

- But, it creates a new problem. What is it?
- What can we do about it?

Forwarding functions

- Problem: Won't work if **fn** takes in rvalues
- What can we do about it?
  - Make a seperate rvalue definition
  - Try template T&&, which binds to everything correctly

```
1 template <typename T>
2 auto wrapper(const T& value) {
3   return fn(value);
4 }
5
6 // Calls fn(x)
7 // Should call fn(std::move(x))
8 wrapper(std::move(x));
```

Forwarding functions

This solves our previous problem, but we still need to come up with a function that matches the pseudocode

```
1 template <typename T>
2 auto wrapper(T&& value) {
3   // pseudocode
4   return fn(value is lvalue ? value : std::move(value));
5 }
6
7 wrapper(std::move(x));
```

std::forward

- Returns reference to value for lvalues
- Returns std::move(value) for rvalues

```cpp
// This is approximately std::forward.
template <typename T>
T& forward(T& value) {
  return static_cast<T&>(value);
}

template <typename T>
T&& forward(T&& value) {
  return static_cast<T&&>(value);
}
```

```cpp
template <typename T>
auto wrapper(T&& value) {
  return fn(std::forward<T>(value));
}

wrapper(std::move(x));
```

std::forward and variadic templates

- Often you need to call a function you know nothing about
  - It may have any amount of parameters
  - Each parameter may be a different unknown type
  - Each parameter may be an lvalue or rvalue

```
1 template <typename... Args>
2 auto wrapper(Args&&... args) {
3   // Note that the ... is outside the forward call, and not right next to args.
4   // This is because we want to call
5   // fn(forward(arg1), forward(arg2), ...)
6   // and not
7   // fn(forward(arg1, arg2, ...)
8   return fn(std::forward<Args>(args)...);
9 }
```

uses of std::forward

The only real use for std::forward is when you want to wrap a function. This could be because:

- You want to do something else before or after (eg. std::make_unique / std::make_shared need to wrap it in the unique/shared_ptr variable)
- You want to do something slightly different (eg. std::vector::emplace uses uninitialised memory construction)
- You want to add an extra parameter (eg. always call a function with the last parameter as 1). This isn't usually very useful though, because it can be achieved with std::bind or lambda functions.