

COMP6771

Advanced C++ Programming

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 (i.e. reference), result is T&
- **Rule 3:** if e is an xvalue, result is T&&
- **Rule 4:** if e is a prvalue, result is T

xvalue/prvalue are forms of rvalues. We do not require you to know this.

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) z; // 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

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

Type Transformations

```
1 #include <iostream>
2 #include <type_traits>
3
4 auto main() -> int {
5     using A = std::add_rvalue_reference<int>::type;
6     using B = std::add_rvalue_reference<int&>::type;
7     using C = std::add_rvalue_reference<int&&>::type;
8     using D = std::add_rvalue_reference<int*>::type;
9
10    std::cout << std::boolalpha
11    std::cout << "typedefs of int&&:" << "\n";
12    std::cout << "A: " << std::is_same<int&&, A>>::value << "\n";
13    std::cout << "B: " << std::is_same<int&&, B>>::value << "\n";
14    std::cout << "C: " << std::is_same<int&&, C>>::value << "\n";
15    std::cout << "D: " << std::is_same<int&&, D>>::value << "\n";
16 }
```


Shortened Type Trait Names

Since C++14/C++17 you can use shortened type trait names.

```
1 #include <iostream>
2 #include <type_traits>
3
4 auto main() -> int {
5     using A = std::add_rvalue_reference<int>;
6     using B = std::add_rvalue_reference<int&>;
7
8     std::cout << std::boolalpha
9     std::cout << "typedefs of int&&:" << "\n";
10    std::cout << "A: " << std::is_same<int&&, A>>::value << "\n";
11    std::cout << "B: " << std::is_same<int&&, B>>::value << "\n";
12 }
```

Binding

Arguments

Parameters

	lvalue	const lvalue	rvalue	const rvalue
template T&&	Yes	Yes	Yes	Yes
T&	Yes			
const T&	Yes	Yes	Yes	Yes
T&&			Yes	

Note:

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

Examples

```
1 #include <iostream>
2
3 auto print(std::string const& a) -> void {
4     std::cout << a << "\n";
5 }
6
7 auto goo() -> std::string const {
8     return "C++";
9 }
10
11 auto main() -> int {
12     auto j = std::string{"C++"};
13     auto const& k = "C++";
14     print("C++"); // rvalue
15     print(goo()); // rvalue
16     print(j); // lvalue
17     print(k); // const lvalue
18 }
```

demo851-bind1.cpp

```
1 #include <iostream>
2
3 template<typename T>
4 auto print(T&& a) -> void {
5     std::cout << a << "\n";
6 }
7
8 auto goo() -> std::string const {
9     return "Test";
10 }
11
12 auto main() -> int {
13     auto j = int{1};
14     auto const& k = 1;
15
16     print(1); // rvalue,          foo(int&&)
17     print(goo()); // rvalue      foo(const int&&)
18     print(j); // lvalue          foo(int&)
19     print(k); // const lvalue    foo(const int&)
20 }
```

demo852-bind2.cpp

Forwarding references

If a variable or parameter is declared to have type **T&&** for some **deduced type** T, that variable or parameter is a *forwarding reference* (AKA *universal reference* in some older texts).

```
1 int n;
2 int& lvalue = n; // Lvalue reference
3 int&& rvalue = std::move(n); // Rvalue reference
4
5 template <typename T> T&& universal = n; // This is a universal reference.
6 auto&& universal_auto = n; // This is the same as the above line.
7
8 template<typename T>
9 void f(T&& param); // Universal reference
10
11 template<typename T>
12 void f(std::vector<T>&& param); // Rvalue reference (read the rules carefully).
```

For more details on forwarding references, see this [blog post](#)

Forwarding functions

Attempt 1: Take in a value

What's wrong with this?

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

- What if we pass in a non-copyable type?
- What happens if we pass in a type that's expensive to copy

Forwarding functions

Attempt 2: Take in a const reference

What's wrong with this?

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

What happens if wrapper needs to
modify value?

Code fails to compile

What happens if we pass in an rvalue?

```
1 // Calls fn(x)
2 // Should call fn(std::move(x))
3 wrapper(std::move(x));
```

Forwarding functions

Attempt 3: Take in a mutable reference

What's wrong with this?

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

What happens if we pass in a const object?

What happens if we pass in an rvalue?

```
1 const int n = 1;
2 wrapper(n);
3 wrapper(1)
```

Interlude: Reference collapsing

- An rvalue reference to an rvalue reference becomes (“collapses into”) an rvalue reference.
- All other references to references (i.e., all combinations involving an lvalue reference) collapse into an lvalue reference.

- $T\& \& \rightarrow T\&$
- $T\&\& \& \rightarrow T\&$
- $T\& \&\& \rightarrow T\&$
- $T\&\& \&\& \rightarrow T\&\&$

Forwarding functions

Attempt 4: Forwarding references

What's wrong with this?

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

```
1 // Instantiation generated
2 template <>
3 auto wrapper<int&>((int&&) value) {
4     return fn(value);
5 }
6
7 // Collapses to
8 template <>
9 auto wrapper<int&>(int& value) {
10     return fn(value);
11 }
12
13 int i;
14 wrapper(i);
```

Calls fn(i)

```
1 // Instantiation generated
2 auto wrapper<int&&>((int&&) value) {
3     return fn(value);
4 }
5
6 // Collapses to
7 auto wrapper<int&&>(int&& value) {
8     return fn(value);
9 }
10
11 int i;
12 wrapper(std::move(i));
```

Also calls fn(i)

The parameter is an rvalue, but inside the function, value is an lvalue

Forwarding functions

Attempt 4: Forwarding references

We want to generate this

```
1 // We want to generate this.
2 auto wrapper<int&>(int& value) {
3     return fn(static_cast<int&>(value));
4 }
```

```
1 // We want to generate this
2 auto wrapper<int&&>(int&& value) {
3     return fn(static_cast<int&&>(value));
4 }
```

It turns out there's a function for this already

```
1 template <typename T>
2 auto wrapper(T&& value) {
3     return fn(std::forward<T>(value));
4     // Equivelantly (don't do this, forward is easier to read).
5     return fn(static_cast<T&&>(value));
6 }
```

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

uses of `std::forward`

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

- You want to do something else before or after
 - `std::make_unique` / `std::make_shared` need to wrap it in the `unique/shared_ptr` variable
 - A benchmarking library might wrap a function call with timers
- You want to do something slightly different
 - `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, because it can be achieved with `std::bind` or lambda functions.

Feedback

