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:

```
int i;
int j& = i;

decltype(i) x; // int - variable
decltype((j)) y; // int& - lvalue
decltype(5) z;   // int - prvalue
```

Determining return types

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

```
template <typename It>
??? find(It beg, It end, int index) {
  for (auto it = beg, int i = 0; beg != end; ++it; ++i) {
    if (i == index) {
      return *it;
    }
  }
  return end;
}
```

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

```
template <typename It>
decltype(*beg) find(It beg, It end, int index) {
        for (auto it = beg, int i = 0; beg != end; ++it; ++i) {
                if (i == index) {
                        return *it;
                }
        }
        return end;
}
```

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

```
template <typename It>
auto find(It beg, It end, int index) -> decltype(*beg) {
        for (auto it = beg, int i = 0; beg != end; ++it, ++i) {
                if (i == index) {
                        return *it;
                }
        }
        return end;
}
```

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

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

template<typename T1, typename T2>
auto print_is_same() -> void {
        std::cout << std::is_same<T1, T2>() << "\n";
}

auto main() -> int {
        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
}
```

demo850-transform.cpp

Type Transformations

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

auto main() -> int {
        using A = std::add_rvalue_reference<int>::type;
        using B = std::add_rvalue_reference<int&>::type;
        using C = std::add_rvalue_reference<int&&>::type;
        using D = std::add_rvalue_reference<int*>::type;

        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";
}
```

Shortened Type Trait Names

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

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

auto main() -> int {
        using A = std::add_rvalue_reference<int>;
        using B = std::add_rvalue_reference<int&>;

        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";
}
```

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

Examples

```cpp
#include <iostream>

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

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

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

demo851-bind1.cpp

```cpp
#include <iostream>

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

auto goo() -> std::string const {
        return "Test";
}

auto main() -> int {
        auto j = int{1};
        auto const& k = 1;

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

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*)*.

```cpp
int n;
int& lvalue = n; // Lvalue reference
int&& rvalue = std::move(n); // Rvalue reference

template <typename T> T&& universal = n;  // This is a universal reference.
auto&& universal_auto = n; // This is the same as the above line.

template<typename T>
void f(T&& param); // Universal reference

template<typename T>
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?

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

- 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?

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

What happens if wrapper needs to modify value?

Code fails to compile

What happens if we pass in an rvalue?

```cpp
// Calls fn(x)
// Should call fn(std::move(x))
wrapper(std::move(x));
```

Forwarding functions

Attempt 3: Take in a mutable reference

What's wrong with this?

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

What happens if we pass in a const object?

What happens if we pass in an rvalue?

```
const int n = 1;
wrapper(n);
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& & -> T&
- T&& & -> T&
- T& && -> T&
- T&& && -> T&&

Forwarding functions

Attempt 4: **Forwarding references**

What's wrong with this?

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

```cpp
// Instantiation generated
template <>
auto wrapper<int&>((int&)&& value) {
  return fn(value);
}

// Collapses to
template <>
auto wrapper<int&>(int& value) {
  return fn(value);
}

int i;
wrapper(i);
```

Calls fn(i)

```cpp
// Instantiation generated
auto wrapper<int&&>((int&&)&& value) {
  return fn(value);
}

// Collapses to
auto wrapper<int&&>(int&& value) {
  return fn(value);
}

int i;
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

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

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

It turns out there's a function for this already

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

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

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

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