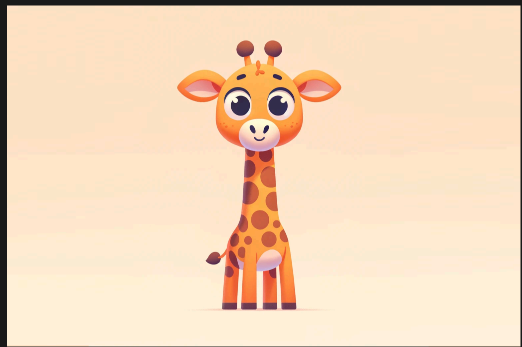
COMP6771

🧪 Metaprogramming & Other

Lecture 9.1

Author(s): Hayden Smith



(Download as PDF)

Decltype

- 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:

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

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

Decltype

Determining return types

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

```cpp
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

🦞 Decltype

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

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

transform1.cpp

🤖 Type Transformations

```cpp
#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";
}
```

transform2.cpp

🤖 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_v<int&&, A> << "\n";
  std::cout << "B: " << std::is_same_v<int&&, B> << "\n";
}
```

transform3.cpp

👔 Binding

	Arguments				
Parameters		**lvalue**	**const lvalue**	**rvalue**	**const rvalue**
	template T&&	✅	✅	✅	✅
	T&	✅	❌	❌	❌
	const T&	✅	✅	✅	✅
	T&&	❌	❌	✅	❌

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

Binding

```cpp
#include <iostream>

auto main() -> int {
  int i;
  int& j = i;

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

  (void)x;
  (void)y;
  (void)z;
}
```

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&)
}
```

bind2.cpp

🤏 Constexpr

- We can provide default arguments to template types (where the defaults themselves are types)
- It means we have to update all of our template parameter lists
- Either:
  - A variable that can be calculated at compile time
  - A function that, if its inputs are known at compile time, can be run at compile time

🤏 Constexpr

```cpp
#include <iostream>

constexpr long long factorial(long long n) {
  if (n == 100000000000000) {
    return 0;
  }
  return n + factorial(n + 1);
}

auto main() -> int {
  constexpr long long ninefactorial = factorial(1);
  std::cout << ninefactorial << "\n";
}
```

constexpr.cpp

🤏 Constexpr

- Benefits:
  - Values that can be determined at compile time mean less processing is needed at runtime, resulting in an overall faster program execution
  - Shifts potential sources of errors to compile time instead of runtime (easier to debug)

🐍 Variadic Templates

```cpp
#include <iostream>
#include <typeinfo>

template <typename T>
auto print(const T& msg) -> void {
  std::cout << msg << " ";
}

template <typename A, typename... B>
auto print(A head, B... tail) -> void {
  print(head);
  print(tail...);
}

auto main() -> int {
  print(1, 2.0f);
  std::cout << "\n";
  print(1, 2.0f, "Hello");
  std::cout << "\n";
}
```

variadic.cpp

```cpp
auto print(const char* const& c) -> void {
  std::cout << c << " ";
}

auto print(float const& b) -> void {
  std::cout << b << " ";
}

auto print(float b, const char* c) -> void {
  print(b);
  print(c);
}

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

auto print(int a, float b, const char* c) -> void {
  print(a);
  print(b, c);
}
```

These are the instantiations that will have been generated

👂 Feedback

Or go to the form here.