ECE326

PROGRAMMING LANGUAGES

Lecture 26 : Template Metaprogramming (C++11)

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2019

decltype

- Returns type of expression

```
A a;
decltype(a.serialize()) test = "test";   // requires ‘a’
decltype(A().serialize()) t2 = "test 2"; // prefer this
```

- What if A does not have default constructor?

```
struct Default { int foo() const { return 1; } };

struct NoDefault {
      NoDefault(const NoDefault&) {}
      int foo() const { return 1; }
};

decltype(NoDefault().foo()) nd = 2; // FAIL
```

Fake Reference

- Recall in container_of macro

```
#define container_of(ptr, type, member) ({ \
  const typeof( ((type *)0)->member ) *__mptr = (ptr); \
  (type *)( (char *)__mptr - offsetof(type, member) ); \
})
```

- We can use this to create a fake reference to avoid constructing objects for the expression

```
#define FAKEREF(T) (*(T *)nullptr)
decltype(FAKEREF(NoDefault).foo()) ndf = 5;     // OK
cout << test << ndf << endl;

5
```

std::declval

- Template version of FAKEREF
- Should only be used in *unevaluated context*
  - Inside sizeof or decltype
- Use this instead of FAKEREF in C++

```
decltype(std::declval<NoDefault>().foo()) nd2 = ndf;  // OK
decltype(std::declval<NoDefault &>().foo()) nd3;      // OK

// error: cannot declare pointer to ‘struct NoDefault&’
decltype(FAKEREF(NoDefault &).foo()) nd3 = 7;
```

Method Check (C++11)

- Using constexpr, decltype, and declval

```
template <class T> struct has_serialize {
  // Test if T has serialize using decltype and declval
  // comma operator returns the value of the latter expression
  template<typename C>
  static constexpr decltype(std::declval<C>().serialize(), bool())
  test(int) {
    return true;
  }

  template<typename C> static constexpr bool test(...) {
    return false;
  }

  // Argument is used to give precedence to first overload of ‘test’
  static constexpr bool value = test<T>(0);
};
```

Tests

- New has_serialize now works for functors too!

```
struct A {
  string serialize() const { return "I am a A!"; }
};
struct B { int x; };
struct C { string serialize; };
struct D {
  struct Functor {
    string operator()() { return "I am a D!"; }
  } serialize;
};

cout << has_serialize<A>::value << endl; // 1 – has serialize function
cout << has_serialize<B>::value << endl; // 0 – no serialize
cout << has_serialize<C>::value << endl; // 0 – serialize not function
cout << has_serialize<D>::value << endl; // 1 – serialize is callable
```

Alternative Implementation

- std::true_type

```
struct true_type { enum { value = true }; };
```

- std::false_type

```
struct false_type { enum { value = false }; };
```

- Use partial template specialization for precedence

```
template<typename T, typename = std::string>
struct can_serialize : false_type {};

template <typename T>
struct can_serialize<T, decltype(std::declval<T>().serialize())>
    : true_type {};
```

General Approach

```
template<typename T, typename = void >
struct can_do_x : false_type {};

template<typename T>
struct can_do_x<T, std::void_t<decltype(
      /* checks if T can do x */)>> : true_type{};
```

- Example

```
template<typename, typename = void>
struct is_incrementable : std::false_type { };

template<typename T>
struct is_incrementable<T,
      std::void_t<decltype(++std::declval<T&>())>
> : std::true_type { };
```

std::void_t

- Maps a sequence of any types to void
- Enables partial template specialization
- Used to detect ill-formed types in SFINAE context

```
template<typename T>
struct is_incrementable<T,
       /* this becomes void, otherwise does not specialize */
       std::void_t<decltype(++std::declval<T&>())>
> : std::true_type { };

cout << is_incrementable<int>::value << endl;    // 1
cout << is_incrementable<string>::value << endl; // 0
cout << is_incrementable<A>::value << endl;      // 0
```

Example

- Two template parameters
  - Can type T be assigned type U

```
template<typename, typename, typename=void>
struct can_assign : std::false_type { };

template<typename T, typename U>
struct can_assign<T, U,
  std::void_t<decltype(std::declval<T&>() = std::declval<U&>())>
> : std::true_type { };

cout << can_assign<int, double>::value << '\n';  // 1
cout << can_assign<int, string>::value << '\n';  // 0
cout << can_assign<double, long>::value << '\n'; // 1

int i; double d = 5.312;
i = d; d = 123L;              // valid assignment without casts
```