ECE326
PROGRAMMING LANGUAGES

Lecture 22 : C++ Metaprogramming

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2019

Constant Expression

- Can be evaluated at compile time

```
const int a = 5 + 7;
// compiler would generate a = 12 directly
```

- constexpr keyword
  - Declares a compile-time variable, function, or class
    - May not exist at runtime (unlike constant variables)
  - Variable
    - Can only be assigned constant expression
  - Function
    - Arguments must only be constant expression

Constexpr

```
constexpr int a[] = { 1, 2, 3, 4 };
constexpr int sum(const int a [], unsigned n) {
      return (n == 0) ? 0 : a[0] + sum(a+1, n-1);
}

// a good compiler should generate x = 10 directly
int x = sum(a, sizeof(a)/sizeof(int));

template<int X>          // template argument only accepts
void print_const() {     // compile-time constant values
      cout << X << endl;
}

print_const<sum(a, 3)>();     // OK, prints 6
print_const<sum(a, 5)>();     // FAIL
error: array subscript value is outside the bounds of array
```

Constexpr Function

- Tells compiler to evaluate function at compile time
- Can significantly increase compile time
  - Compiler must ensure computation cannot crash itself
    - Performs extensive type-checking
    - E.g. Array out of bound check
- C++11
  - Restrictive on what's allowed in a constexpr function
    - No loops – must rely on recursion
    - Exactly one return statement allowed in body
    - No local variables, arguments only

Constexpr Examples

```
constexpr int factorial(int n) {
    return n <= 1 ? 1 : (n * factorial(n - 1));
}

/* lexicographical comparison of two constant strings */
/* returns positive if a > b, negative if a < b, 0 if equal */
constexpr int
constcmp(const char * a, const char * b, unsigned i=0)
{
    return (a[0] == '\0') ? (a[0] - b[0]) : (
        (a[0] == b[0]) ? constcmp(a+1, b+1, i+1) : a[0] - b[0]
    );
}

constcmp("he", "hello")       // -108
constcmp("hello", "hell“)     // 111  (ASCII for o)
```

Constexpr Function

- Depends on compiler implementation
  - May or may not be turned into a runtime function
- Depends on argument

```
print_const<constcmp("hello", argv[0])>();
error: ‘argv’ is not a constant expression

// function is also used at runtime
cout << constcmp("hello", argv[0]) << endl;     // 58
```

- Upgrade to C++14
  - Allows loops and local variables!

Compile-Time Function

- Useful for pre-calculating values
  - E.g. crc64 hash of constant strings
- Can be used in conjunction with templates
- Referentially transparent
  - Does not have side effects
    - Note: this is only true if the function is run at compile time. If it is converted to a run time function, it can modify global variables!
- Haskell does this *a lot*
  - The entire program may be optimized down to constants

Constexpr Class

- Its instances can be compile-time objects
  - Same restrictions apply to methods, but can use members

```
class Rectangle  {
    int _h, _w;
public:
    // a constexpr constructor
    constexpr Rectangle (int h, int w) : _h(h), _w(w) {}
    constexpr int area ()  { return _h * _w; }
};

constexpr Rectangle rekt(10, 20);   // compile-time
print_const<rekt.area()>();         // 200

Rectangle rect(5, argc);            // runtime Rectangle
cout << rect.area() << endl;        // 5 (if argc == 1)
```

Static Introspection

- Making programming decisions based on types
  - At compile time (hence “static”)
- Limited support in C
  - E.g. sizeof and typeof (non-standard)
- C++ template
  - Originally designed for generic programming
  - Its implementation allows for some introspection capability
    - Requires exploiting template substitution rules
    - Originally part of Boost library, now standardized for C++11

Type Trait

- #include <type_traits>
- is_integral<T>
  - Checks if type is some kind of integer (int, char, long, …etc)

```
template <class T>
T f(T i) {
  static_assert(std::is_integral<T>::value, “invalid type");
  return i;
}
```

- is_array<T>
  - Checks if type is an array

SFINAE

- **S**ubstitution **F**ailure **I**s **N**ot **A**n **E**rror
  - *An invalid substitution of template parameters is not an error*
- C++ creates a set of candidates for overload resolution
  - E.g. during function overloading
- For templates, if parameter substitution fails, then that template will be removed from the candidate list
  - without stopping on compilation error
  - Note: error in template body is not detected before resolution
- No error is produced if more than one candidate exists

SFINAE example

```
struct Test {
  typedef int foo;      // internal type to Test
};

template <typename T>
void f(typename T::foo) {}  // Definition #1
```

Use of internal typedef in templates requires prefixing the type alias with typename

```
template <typename T>
void f(T) {}                // Definition #2

int main() {
  f<Test>(10);  // Call #1.
  f<int>(10);   // Call #2 without error. (SFINAE)
}
```

int does not have a type named foo (1st substitution fails)

sizeof operator

- Returns size of an *expression* at compile time

```
typedef char type_test[42];
type_test& f();

// f() won't actually be called at runtime
cout << sizeof(f()) << endl;  // 42
```

- Can be exploits by SFINAE
- Running example
  - Want to check if class has serialize function
    - If yes, call it, otherwise, call to_string() instead

Member Function Pointer

- Similar to function pointer, except must specify class
  - Has a different type than normal functions

```
struct A {
    string serialize() const { return "I am a A!"; }
};

typedef string (A::* afunc_t)();

A a;
afunc_t af = &A::serialize;
cout << (a.*af)() << endl;    // call member function
A * ap = &a;
cout << (a->*af)() << endl;   // call member function
```

Method Check

```
template <class T> struct has_serialize {
    typedef char yes[1]; typedef char no[2]; static char tm[2];

    /* checks if class T really has serialize method (not field) */
    template<typename U, U u> struct really;

    /* class T has serialize */
    template<typename Z> static yes& test(
        really<string(Z::*)(), &Z::serialize>*) { return tm; }
    template<typename Z> static yes& test(
        really<string(Z::*)() const, &Z::serialize>*) { return tm; }

    /* SFINAE - class t does not have serialize */
    template<typename> static no& test(...) { return tm; }

    // The constant used as a return value for the test.
    static const bool value = sizeof(test<T>(0)) == sizeof(yes);
};
```

Test 1

```
struct A {
    string serialize() const { return "I am a A!"; }
};
cout << has_serialize<A>::value << endl;  // 1 – it has serialize

template<typename U, U u> struct really; // (for reference)
```

- 3 candidates for Test<A>(0)

```
// 1. NO: type U = string(A::*)() != typeof(&A::serialize)
template<A> static yes& test(really<string(A::*)(), &A::serialize>*)

// 2. YES: type U = string(A::*)() const == typeof(&A::serialize)
template<A> static yes& test(
                      really<string(A::*)() const, &A::serialize>*)

// 3. YES: this template cannot fail, but has lowest precedence
template<typename> static no& test(...)
```

Test 1

```
struct A {
    string serialize() const { return "I am a A!"; }
};
cout << has_serialize<A>::value << endl;  // 1 – it has serialize
```

Compiler chooses candidate 2, which returns *yes*.
`sizeof(test<A>(0)) == sizeof(yes)` is true. so `has_serialize<A>::value` is also true.

```
// 2. YES: type U = string(A::*)() const == typeof(&A::serialize)
template<A> static yes& test(
                        really<string(A::*)() const, &A::serialize>*)

// 3. YES: this template cannot fail, but has lowest precedence
template<typename> static no& test(...)
```

Test 2

```
struct B {
    int x;   /* does not have serialize method */
};
cout << has_serialize<B>::value << endl;  // 0 – does not have serialize

template<typename U, U u> struct really;
```

- 3 candidates for Test<B>(0)

```
// 1. NO: B::serialize does not exist!
template<B> static yes& test(really<string(B::*)(), &B::serialize>*)

// 2. NO: B::serialize does not exist!
template<B> static yes& test(
                       really<string(B::*)() const, &B::serialize>*)

// 3. YES: this template cannot fail, but has lowest precedence
template<typename> static no& test(...)
```

Test 2

```
struct B {
    int x;    /* does not have serialize method */
};
cout << has_serialize<B>::value << endl;  // 0 – does not have serialize
```

Compiler chooses candidate 3, which returns *no*.
`sizeof(test<B>(0)) == sizeof(yes)` is false, so
`has_serialize<B>::value` is also false.

```
// 2. NO: B::serialize does not exist!
template<B> static yes& test(
                      really<string(B::*)() const, &B::serialize>*)

// 3. YES: this template cannot fail, but has lowest precedence
template<typename> static no& test(...)
```

Test 3

```
struct C {
    string serialize;      /* serialize is not a method */
};
cout << has_serialize<C>::value << endl;

template<typename U, U u> struct really;
```

typeof(&C::serialize) is string * (pointer to a string), not a member function pointer.

- 3 candidates for Test<C>(0)

```
// 1. NO: type U = string(C::*)() != typeof(&C::serialize)
template<C> static yes& test(really<string(C::*)(), &C::serialize>*)

// 2. NO: type U = string(C::*)() const != typeof(&C::serialize)
template<C> static yes& test(
                       really<string(C::*)() const, &C::serialize>*)

// 3. YES: this template cannot fail, but has lowest precedence
template<typename> static no& test(...)
```