

ECE326

PROGRAMMING LANGUAGES

Lecture 22 : SFINAE (C++)

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2019

Assignment 3

- Object Relational Mapping
 - Maps database rows into in-memory objects
- Relational Database
 - Data are organized into tables (analogous to classes) and rows (analogous to instances)
 - Like classes, format of database tables must be specified
 - The specification is called database schema
- EasyDB
 - Very simple in-memory database, does not save to disk
 - You will implement a (small) part of it for assignment 4

Database Schema

- EasyDB Schema Language
- Support four data types
 - string (any length)
 - float (64 bits)
 - integer (64 bits)
 - foreign key
- Foreign Key
 - Analogous to a pointer
 - References another row in another table

```
User {  
    firstName: string;  
    lastName: string;  
    height: float;  
    age: integer;  
}
```

```
Account {  
    user: User; // foreign  
    type: string;  
    balance: float;  
}
```

Database Client

- First milestone
- Write the client code to communicate with the server
 - Over the network, through TCP/IP
- Requires sending and receiving *packets*
- Packet
 - Data sent over the network
 - Requires serialization and deserialization
 - Converting in-memory data to/from network format

ORM Layer

- Second milestone
- Convert raw formats to/from Python objects
- Provides object-oriented interface instead of database commands
- Performs extensive type safety checks
- Coding requires use of advanced Python features
 - Metaclass
 - Descriptors

Database Schema

- Written in Python for the ORM
 - Can be exported to EasyDB schema language
 - Includes options for more rigorous type checking

```
class User(orm.Table):
    firstName = orm.String()
    lastName = orm.String()
    height = orm.Float(blank=True)
    age = orm.Integer(blank=True)

class Account(orm.Table):
    user = orm.Foreign(User)
    type = orm.String(choices=["Savings", "Chequing", ],
                      default="Chequing")
    balance = orm.Float()
```

ORM Interface

- Underlying database abstracted away
- Provides user with a way to work with any database

```
# create new object
>> joe = User(db, firstName="Joe",
lastName="Harris", age=32)
>> joe.save() # save to database

# search for objects in database
>> result = User.filter(lastName="Harris")
>> result
[<User: Joe Harris>, <User: Matt Harris>]
>> joe = result[0]
>> joe.age
32
```

```
# update existing object
>> joe.height = 7.4
>> joe.save()

# delete an object
>> result[1].delete()

# verify deletion
>> User.count(
..     lastName="Harris")
1
```

Type Support

- Most database only supports few data types
 - EasyDB only supports integer, float, and string
- At the ORM layer, we can support more!
- Third milestone
- DateTime Field
 - Corresponds to Python's DateTime class
- Coordinate Field
 - A tuple of two values: longitude and latitude

SFINAE

For C++98

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

- **Substitution Failure Is Not An Error**
 - *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 does not have a
type named foo (1st
substitution fails)

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

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 (C++98)

```
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 == sizeof(&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(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(0)) == sizeof(yes)` is false, so
`has_serialize::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(...)
```

Test 4

- Current template does not support functors
 - It should, we will fix this in C++11

```
struct D {  
    struct Functor {  
        string operator()() {  
            return "I am a D!";  
        }  
    };  
    Functor serialize;  
};
```

```
D d;  
cout << d.serialize() << endl;           // "I am a D!"  
cout << has_serialize<D>::value << endl; // Output 0.
```

Applying Method Check

- `make_packet` function
 - Calls `serialize` if class has it, otherwise use `to_string()`

```
template <class T> string make_packet(const T& obj) {  
    if (has_serialize<T>::value) {  
        // error: no member named 'serialize' in 'A'.  
        return obj.serialize();  
    } else {  
        return to_string(obj);  
    }  
}
```

```
A a;  
make_packet(a);
```

Applying Method Check

- It doesn't work
 - Static type system is conservative.
 - Dead code is still type checked.

```
template <class T> string make_packet(const T& obj) {  
    /* has_serialize<T>::value is 0 at compile time */  
    if (0) {  
        // error: no member named 'serialize' in 'A'.  
        return obj.serialize();  
    } else {  
        return to_string(obj);  
    }  
}
```


Workaround

- `enable_if`
 - If condition B is true, typedef T in the struct named type

```
// This struct doesn't define "type" and will trigger SFINAE
template<bool B, class T = void> struct enable_if {};
```

```
// partial specialization if B is true, struct defines "type" to T
template<class T> struct enable_if<true, T> { typedef T type; };
```

```
// OK enable_if<true, int> defines 'type' to int
enable_if<true, int>::type t1;
enable_if<has_serialize<A>::value, int>::type t2; // OK
```

```
// FAIL - enable_if<false, int> does not define 'type'
enable_if<false, int>::type t3;
enable_if<has_serialize<B>::value, int>::type t4;
```

Final Solution

- Use `enable_if` to switch between function overload

```
template <class T>
typename enable_if<has_serialize<T>::value, string>::type
make_packet(const T& obj) {
    return obj.serialize();
}

template <class T>
typename enable_if<!has_serialize<T>::value, string>::type
make_packet(const T& obj) {
    return to_string(obj);
}

A a; B b; C c;

cout << make_packet(a) << endl;    // calls a.serialize()
cout << make_packet(b) << endl;    // calls to_string(b)
cout << make_packet(c) << endl;    // calls to_string(c)
```