OPEN IS GOOD

CASE STUDY: AST

```
struct Node {
    virtual double value() = 0;
};
```

const, memory management etc elided, code formatted to "slide style"

CASE STUDY: AST

```
struct Number : Node {
  Number(double value) : val(value ) { }

  double value() override {
    return val;
  }

  double val;
};
```

CASE STUDY: AST

```
struct Plus : Node {
    Plus(Node& left, Node& right)
    : left(left), right(right) { }

  double value() override {
    return left.value() + right.value();
  }

  Node &left, &right;
};
```

CASE STUDY: AST

```
struct Times : Node {
    Times(Node& left, Node& right)
    : left(left), right(right) { }

    double value() override {
        return left.value() * right.value();
    }

    Node &left, &right;
};
```

CASE STUDY: AST

```
int main() {
    Node* expr =
        new Times(*new Number(2),
                  *new Plus(*new Number(3), *new Number(4)));

    cout << expr->value() << "\n"; // 14

    return 0;
}
```

CASE STUDY: AST

```
int main() {
    Node* expr =
        new Times(*new Number(2),
                  *new Plus(*new Number(3), *new Number(4)));

    cout /* string to RPN */
        << " = " << expr->value() << "\n";
    // 2 3 4 * + = 14

    return 0;
}
```

AST: ADD A VIRTUAL FUNCTION?

...if it needs to be virtual, make it a member function ???

```
struct Node {
    // as before
    virtual string toRPN() = 0;
};

struct Plus : Node {
    // as before
  string toRPN() override {
    return left.toRPN() + " " + right.toRPN() + " +";
  }
};

// same for Number and Times
```

banana -> gorilla -> jungle

AST: TYPE SWITCH?

```
string toRPN(Node& node) {
    if (auto expr = dynamic_cast<Number*>(&node)) {
        return to_string(expr->value());
    } else if (auto expr = dynamic_cast<Plus*>(&node)) {
        return toRPN(expr->left) + " " +
            toRPN(expr->right) + " +";
    } else if (auto expr = dynamic_cast<Times*>(&node)) {
        return toRPN(expr->left) + " " +
            toRPN(expr->right) + " *";
    }
    throw runtime_error("unknown node type");
}
```

needs modification each time a new Node subtype is added

AST: VISITOR?

```
struct Node {
    // as before
    struct Visitor {
        virtual void accept(Number& expr) = 0;
        virtual void accept(Plus& expr) = 0;
        virtual void accept(Times& expr) = 0;
    };

    virtual void visit(Visitor& viz) = 0;
};
```

AST: VISITOR?

```
struct Number : Node {
  // as before
  void visit(Visitor& viz) override { viz.accept(*this);
  }
};

struct Plus : Node {
  void visit(Visitor& viz) override { viz.accept(*this); }
};

struct Times : Node {
    void visit(Visitor& viz) override { viz.accept(*this); }
};
```

AST: VISITOR?

```
struct RPNVisitor : Node::Visitor {
  string result;
  void accept(Number& expr) {
    result = to_string(expr.val);
  }
  void accept(Plus& expr) {
    expr.left.visit(*this);
    string l = result;
    expr.right.visit(*this);
    result = l + " " + result + " +";
  }
  void accept(Times& expr) { ... }
};
```

ugh, yuck!

AST: VISITOR?

```
string toRPN(Node& node) {
  RPNVisitor viz;
  node.visit(viz);
  return viz.result;
}
```

my, that was a lot of work

and, what does it even gain us?

AST: FUNCTION TABLE?

```
using RPNFormatter = string (*)(Node&);
unordered_map<type_index, RPNFormatter> RPNformatters;

string toRPN(Node& node) {
  return RPNformatters[typeid(node)](node);
}
```

AST: FUNCTION TABLE?

```
namespace { struct Init {
  Init() {
    RPNformatters[typeid(Number)] = [](Node& node) {
     return to_string(static_cast<Number&>(node).val); };
    RPNformatters[typeid(Plus)] = [](Node& node) {
     auto expr = static_cast<Plus&>(node);
     return toRPN(expr.left) + " " + toRPN(expr.right)
       + " +"; };
    // same for Time
  }
};
Init init;
} }
```

not bad, actually

THE EXPRESSION PROBLEM

behaviors += types

types += behavior

MULTI-LAYER ARCHITECTURES

PRESENTATION

DOMAIN

PERSISTENCE

- presentation: PersonDlg, CriminalCaseDlg
- domain: Person, CriminalCase
- persistence: persist to database, to json...
- cross-cutting concerns

YOMM2

AST: OPEN METHODS

```
#include <yorel/yomm2/cute.hpp>

register_class(Node);
register_class(Plus, Node);
register_class(Times, Node);
register_class(Number, Node);
```

(the boring part)

AST: OPEN METHODS

```
using yorel::yomm2::virtual_;

declare_method(string, toRPN, (virtual_<const Node&>));

define_method(string, toRPN, (const Number& expr)) {
  return std::to_string(expr.val);
}

define_method(string, toRPN, (const Plus& expr)) {
  return toRPN(expr.left) + " " + toRPN(expr.right) + " +";
}

// same for Times
```

AST: WHAT ABOUT VALUE?

- `value` in the node hierarchy screams interpreter
- the AST classes should *only* represent the tree

```
declare_method(int, value, (virtual_<const Node&>));

define_method(int, value, (const Number&& expr)) {
  return expr.val;
}

define_method(int, value, (const Plus& expr)) {
  return value(expr.left) + value(expr.right);
}

define_method(int, value, (const Times& expr)) {
  return value(expr.left) * value(expr.right);
}
```

MULTIPLE DISPATCH?

Yes.

OCCASIONALLY USEFUL

```
add(Matrix, Matrix)                    -> Matrix
                                          add all elements
add(DiagonalMatrix, DiagonalMatrix) -> DiagonalMatrix
                                          just add diagonals

fight(Human, Creature, Axe)    -> not agile enough to wield
fight(Warrior, Creature, Axe)  -> chop it into pieces
fight(Warrior, Dragon, Axe)    -> die a honorable death
fight(Human, Dragon, Hands)    -> you just killed a dragon
                                  with your bare hands!
                                  incredible isn't it?
```

SYNTAX

Just use `virtual_<>` on several arguments:

```
declare_method(std::string, fight,
  (virtual_<Character&>, virtual_<Creature&>,
   virtual_<Device&>));

define_method(std::string, fight,
  (Human& x, Creature& y, Axe& z)) {
  return "not agile enough to wield";
}

define_method(std::string, fight,
  (Human& x, Dragon& y, Hands& z)) {
  return "you just killed a dragon with your bare hands."
         " Incredible isn't it?";
}
```

SELECTING THE RIGHT SPECIALIZATION

- works just like selecting from set of overloads (but at runtime!)
- or partial template specialization
- ambiguities can arise

next

calls the next most specific override

```
define_method(std::string, kick, (Dog& dog)) {
  return "bark";
}

define_method(std::string, kick, (Bulldog& dog)) {
    return next(dog) + " and bite";
}
```

next

```
define_method(void, inspect, (Vehicle& v, Inspector& i)) {
  cout << "Inspect vehicle.\n";
}

define_method(void, inspect, (Car& v, Inspector& i)) {
  next(v, i);
  cout << "Inspect seat belts.\n";
}

define_method(void, inspect, (Car& v, StateInspector& i)) {
  next(v, i);
  cout << "Check road tax.\n";
}
```

IS THIS OOP?

- brief history of OOP: Simula, Smalltalk, C++/Java/D/...
- CLOS: not objects talking to each other a la Smalltalk
- algorithms retake the front stage
- no unnecessary breach of encapsulation

INSIDE YOMM2

- purely in C++17 (no extra tooling)
- constant time dispatch
- uses tables of function pointers
- object -> dispatch data?
  - perfect integer hash of &type_info

A PAYROLL APPLICATION

- *Role*
  - Employee
    - Manager
  - Founder
- *Expense (X)*
  - Cab, Jet
  - *Public*
    - Bus, Train

THE pay (UNI-) METHOD

```
declare_method(double, pay, (virtual_<Employee&>));

define_method(double, pay, (Employee&)) {
  return 3000;
}

define_method(double, pay, (Manager& manager)) {
  return next(manager) + 2000;
}
```

THE `approve` (MULTI-) METHOD

```
declare_method(bool, approve,
  (virtual_<Role&>, virtual_<Expense&>, double));
define_method(bool, approve,
  (Role& r, Expense& e, double amount))
{ return false; }
define_method(bool, approve,
  (Employee& r, Public& e, double amount))
{ return true; }
define_method(bool, approve,
  (Manager& r, Taxi& e, double amount))
{ return true; }
define_method(bool, approve,
  (Founder& r, Expense& e, double amount))
{ return true; }
```

DECLARE_METHOD

```
declare_method(double, pay, (virtual_<Employee&>));
```

```
struct _yomm2_method_pay;

namespace {
namespace YoMm2_nS_10 {
using _yOMM2_method =
  method<void, _yomm2_method_pay,
    double(virtual_<Employee &>),
    default_policy>;
_yOMM2_method::init_method init;
}
}
```

DECLARE_METHOD

```
declare_method(double, pay, (virtual_<Employee&>));
```

```
YoMm2_nS_10::_yOMM2_method
pay(discriminator, Employee & a0);

inline double
pay(Employee & a0) {
  auto pf = reinterpret_cast<double (*)(
    Employee & a0)>(
    YoMm2_nS_10::_yOMM2_method::resolve(a0));
  return pf(a0);
};
```

DEFINE_METHOD

```
    define_method(double, pay, (Employee&)) { return 3000; }
```

```
namespace { namespace YoMm2_nS_12 {
template <typename T> struct select;
template <typename... A> struct select<void(A...)> {
  using type = decltype(
    pay(discriminator(), std::declval<A>()...));
};

using _yOMM2_method =
  select<void(Employee &)>::type;
using _yOMM2_return_t = _yOMM2_method::return_type;
_yOMM2_return_t (*next)(Employee &);
```

DEFINE_METHOD

```
    define_method(double, pay, (Employee&)) { return 3000; }
```

```
struct _yOMM2_spec {
  static double body(Employee &);
};

register_spec<_yOMM2_return_t, _yOMM2_method,
                    _yOMM2_spec, void(Employee &)>
_yOMM2_init((void **)&next);
} }

double YoMm2_nS_12::_yOMM2_spec::body(Employee &)
{ return 3000; }
```

UPDATE_METHODS

- process the info registered by static ctors
- build representation of class hierarchies
- build all the dispatch data inside a single vector
- find a perfect hash function over relevant type_info's
  - H(x) = (M * x) >> S

DISPATCHING A UNI-METHOD

- pretty much like virtual member functions
- method table contains a pointer to the effective function
- only it is not at a fixed offset in the method table

DISPATCHING A 1-METHOD

during `update_methods`

```
method<pay>::slots_strides.i = 1;

// method table for Employee
mtbls[ H(&typeid(Employee)) ] = {
  ..., // used by approve,
  wrapper(pay(Employee&))
};

// method table for Manager
mtbls[ H(&typeid(Manager&)) ] = {
  ..., // used by approve,
  wrapper(pay(Manager&))
};
```

DISPATCHING A 1-METHOD

```
pay(employee)
```

=>

```
mtbls[ H(&typeid(employee)) ]      // mtable for type
  [ method<pay>::slots_strides.i ] // pointer to fun
(employee)                         // call
```

PERFORMANCE?

```
double call_pay(Employee& e) { return pay(e); }
```

```
;;; g++-6 -DNDEBUG -O2 ...
movq    mtbls(%rip), %rax                          ; hash table
movb    mtbls+32(%rip), %cl                        ; shift factor
movslq  method<pay>::slots_strides(%rip), %rdx     ; slot
movq    (%rdi), %rsi                               ; vptr
movq    -8(%rsi), %rsi                             ; &type_info
imulq   mtbls+24(%rip), %rsi                       ; * M
shrq    %cl, %rsi                                  ; >> S
movq    (%rax,%rsi,8), %rax                        ; method table
jmpq    *(%rax,%rdx,8)                             ; call
```

DISPATCHING A MULTI-METHOD

- it's a little more complicated
- use a multi-dimensional dispatch table
- size can grow very quickly
- the table must be "compressed", devoid of redundancies
- in fact the "uncompressed" table never exists
- work in terms of class *groups*, not classes

DISPATCHING A MULTI-METHOD

	Expense+Jet	Public+Bus+Train	Cab
Role	R,X	R,X	R,X
Employee	R,X	E,P	R,X
Manager	R,X	E,P	M,C
Founder	F,X	F,X	F,X

(column major)

DISPATCHING A MULTI-METHOD

```
method<approve>::.slots_strides.pw = { 0, 4, 0 };

mtbls[ H(&typeid(Employee)) ] = {
  // & of (Employee,Expense+Jet) cell
  // used by pay
};
mtbls[ H(&typeid(Manager)) ] = {
  // & of (Manager,Expense+Jet) cell
  // used by pay
};

mtbls[ H(&typeid(Expense)) ] = { 0 }; // also for Jet
mtbls[ H(&typeid(Public))  ] = { 1 }; // also for Bus, Train
mtbls[ H(&typeid(Cab))     ] = { 2 };
```

DISPATCHING A MULTI-METHOD

```
approve(role, expense, amount)
```

=>

```
word* slots_strides = method<approve>::.slots_strides.pw;

mtbls[ H(&typeid(r)) ]       // method table for 'r'
  [ slots_strides.pw[0].i ] // ptr to cell in 1st column
  [ mtbls [ H(&typeid(x)) ] // method table for 'x'
    [ slots_strides[2].i *  // column
      slots_strides[1].i ]  // stride
  ]                         // pointer to function
(role, expense, amount)     // call
```

PERFORMANCE SUMMARY

		gcc6	clang6
normal inheritance			
virtual function	1-method	16%	17%
double dispatch	2-method	25%	35%
virtual inheritance			
virtual function	1-method	19%	17%
double dispatch	2-method	40%	33%

YOMM2 VS OTHER SYSTEMS

- Pirkelbauer - Solodkyi - Stroustrup (PSS)
- yomm11
- (Cmm)
- (Loki)

YOMM2 VS PSS

- Solodon's papers on open methods etc:
  - Open Multi-Methods for C++
  - Design and evaluation of C++ open multi-methods
  - Simplifying the Analysis of C++ Programs
- yomm2 overrides are not available for overloading
- yomm2 overrides cannot specialize multiple methods
- PSS attempts harder to resolve ambiguities
- yomm2 supports smart pointers
- yomm2 supports `next`

YOMM2 VS YOMM11

- no need to instrument classes to get speed
- methods are ordinary functions (overload, etc)
- seamless overriding across namespaces

LINKS

- github: https://github.com/jll63/yomm2
- this presentation: https://jll63.github.io/yomm2/cppnow2018/
- contact: Jean-Louis Leroy - jl@leroy.nyc