```
class foo {
    ~~virtual void bar();~~
};
```

YOMM11
OPEN MULTI-METHODS FOR C++11

Or, down with virtual member functions

Jean-Louis Leroy - jl@leroy.nyc

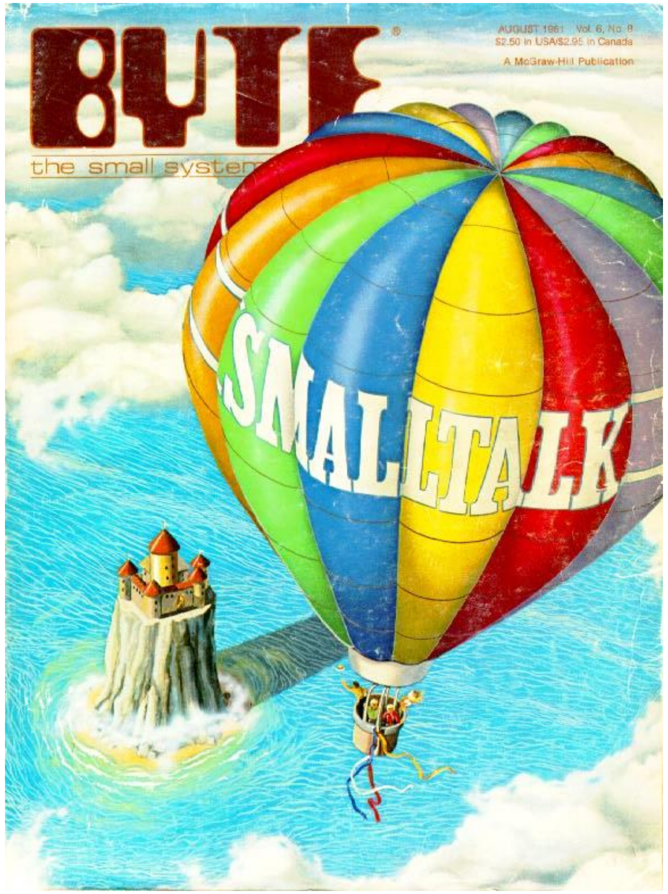
1981 – OOP GOES MAINSTREAM

Byte issue on Smalltalk

OOP hits the mainstream

The Smalltalk metaphor:

- objects communicate by sending messages
- objects react to messages according to their nature



THE PROMISES OF OOP

- reuse
- software ICs
- solve software complexity crisis
- maintain-ability
- modularity
- quality

REUSE?

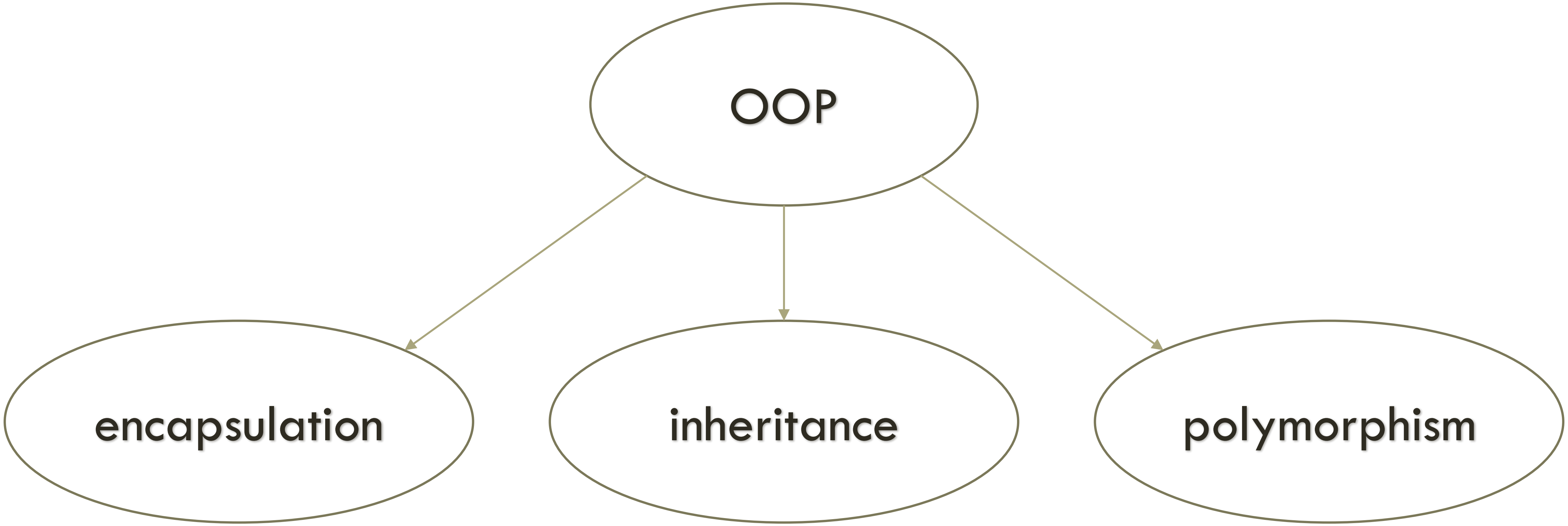
Yeah, to an extent

For example, application frameworks, some of them were ok

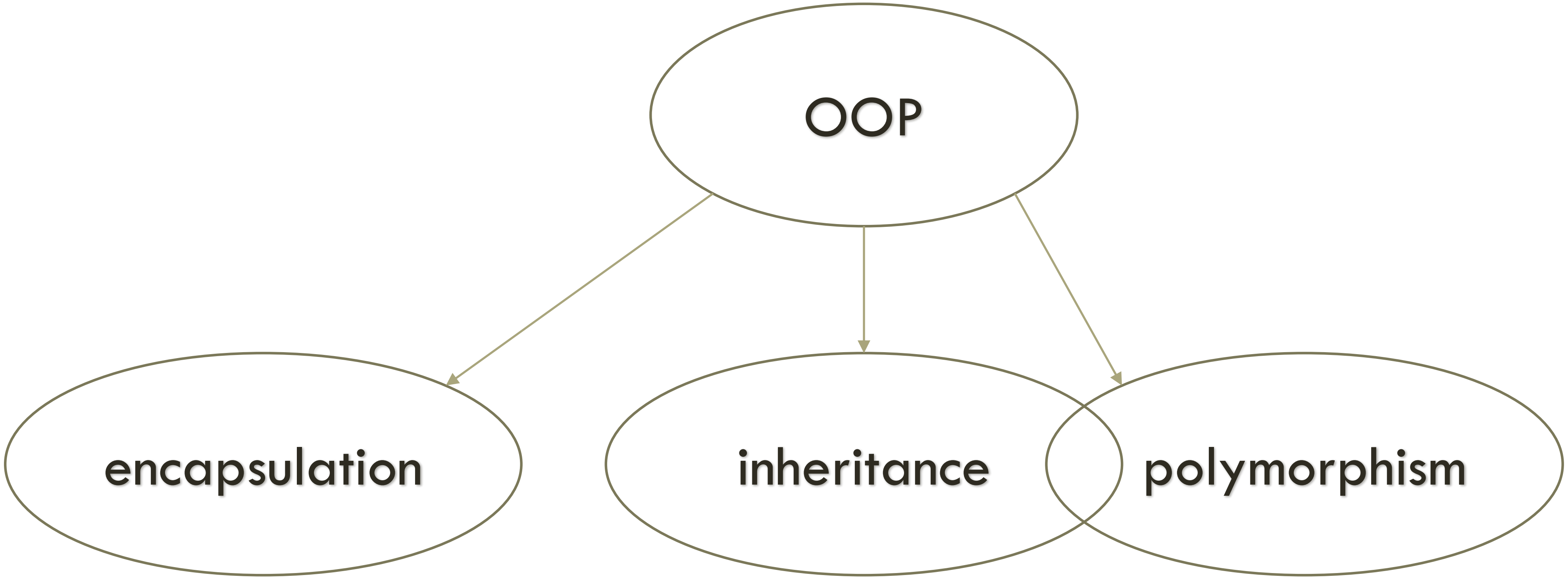
But software ICs? Hmmm...

What went wrong?

THE THREE PILLARS



THREE PILLARS, OR TWO AND A HALF



C++ POLYMORPHISM = VIRTUAL MEMBER FUNCTION

increased coupling

limited extensibility

encapsulation violated

no multiple dispatch

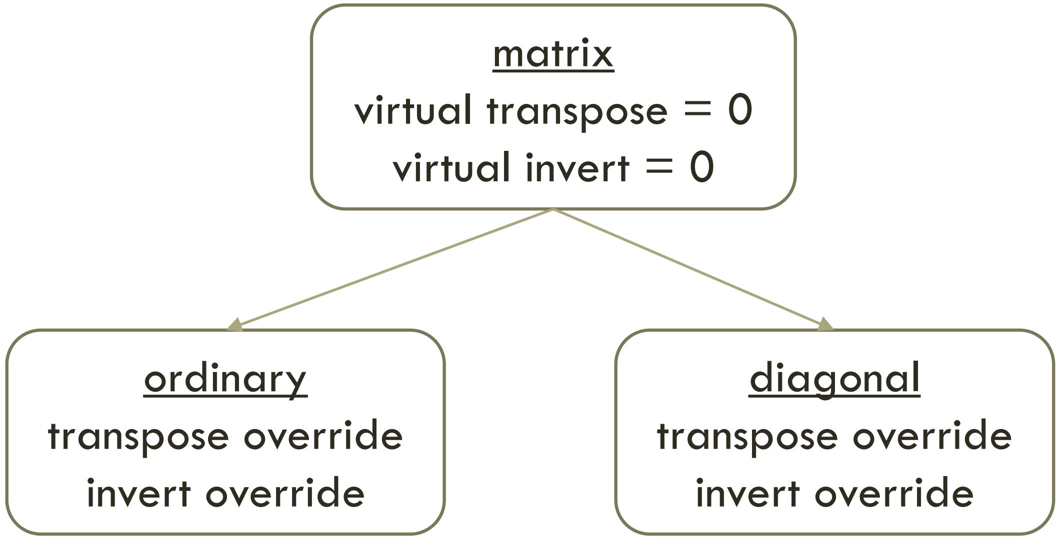
Member virtual functions are bad

AN EXAMPLE

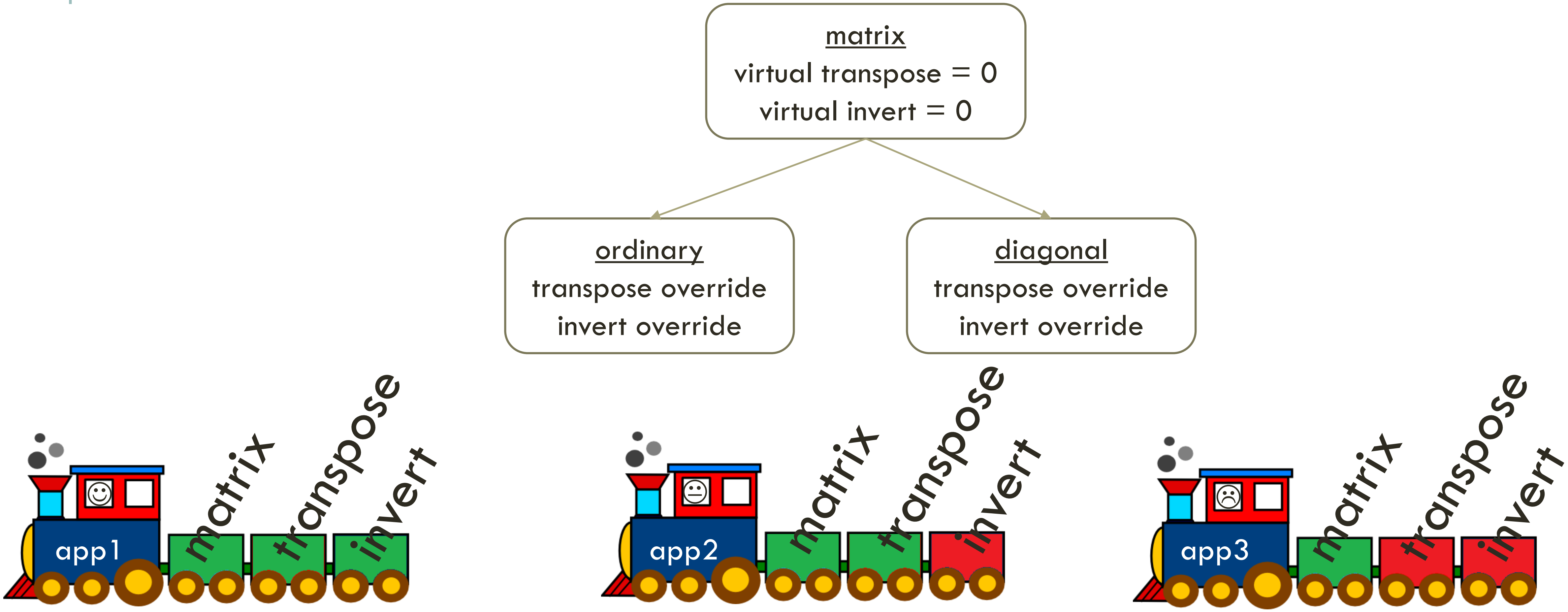
matrix: abstract base class

ordinary: store everything

diagonal: store only diagonal



MEMBER VIRTUAL FUNCTIONS INCREASE COUPLING



MEMBER VFUNCS DO NOT SUPPORT EXTENSION WELL

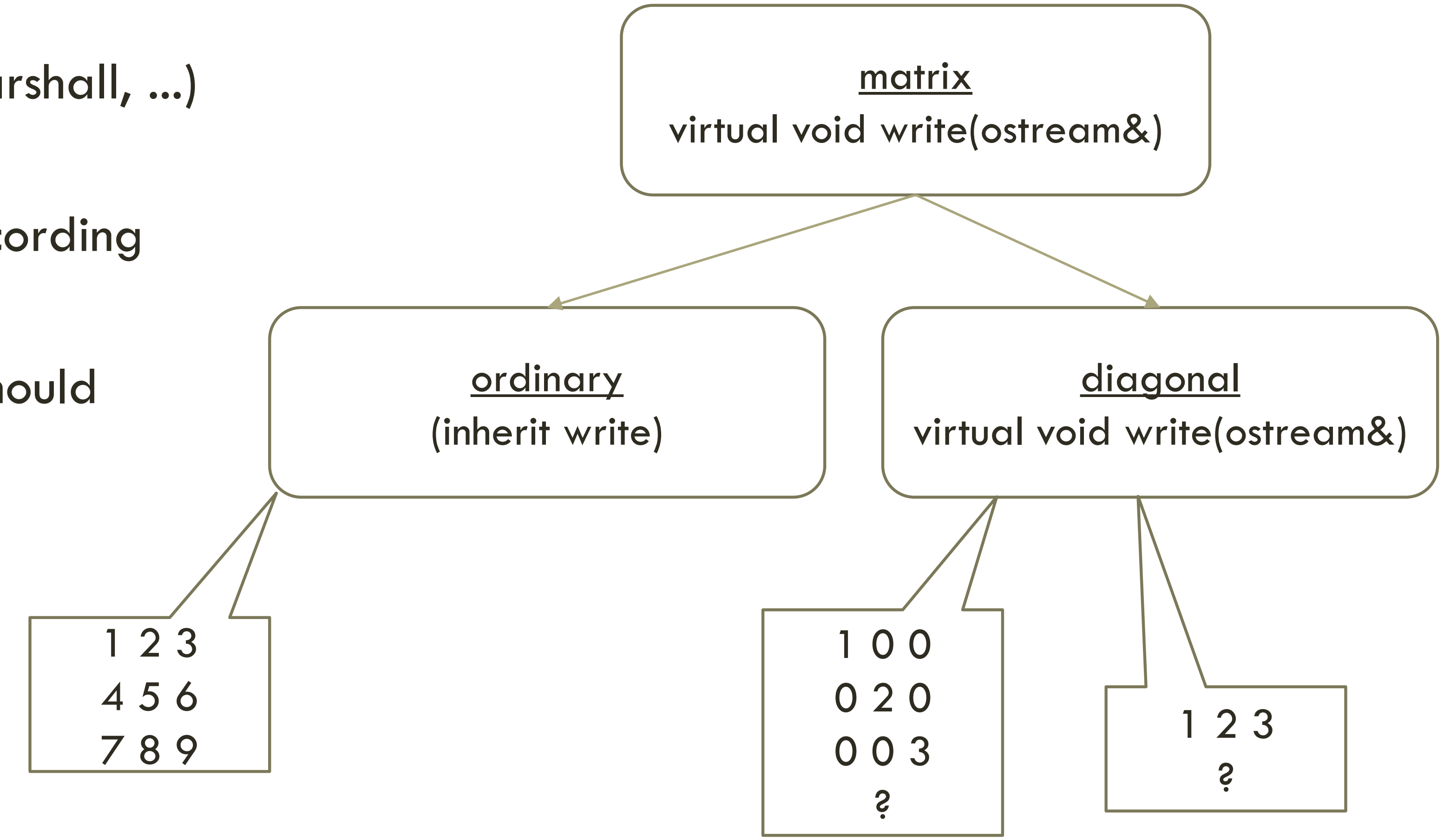
It is useful to write (serialize, marshall, …) a matrix to a stream

It may be necessary to do it according to the matrix' subtype

Who decides what the output should look like?

It shoud not the matrix!

It should be the application!



EXTENSION – A REAL LIFE EXAMPLE

Three-tier app (presentation, business, persistence).

Complex networks of polymorphic objects (e.g. party to the case can be either legal or natural person)

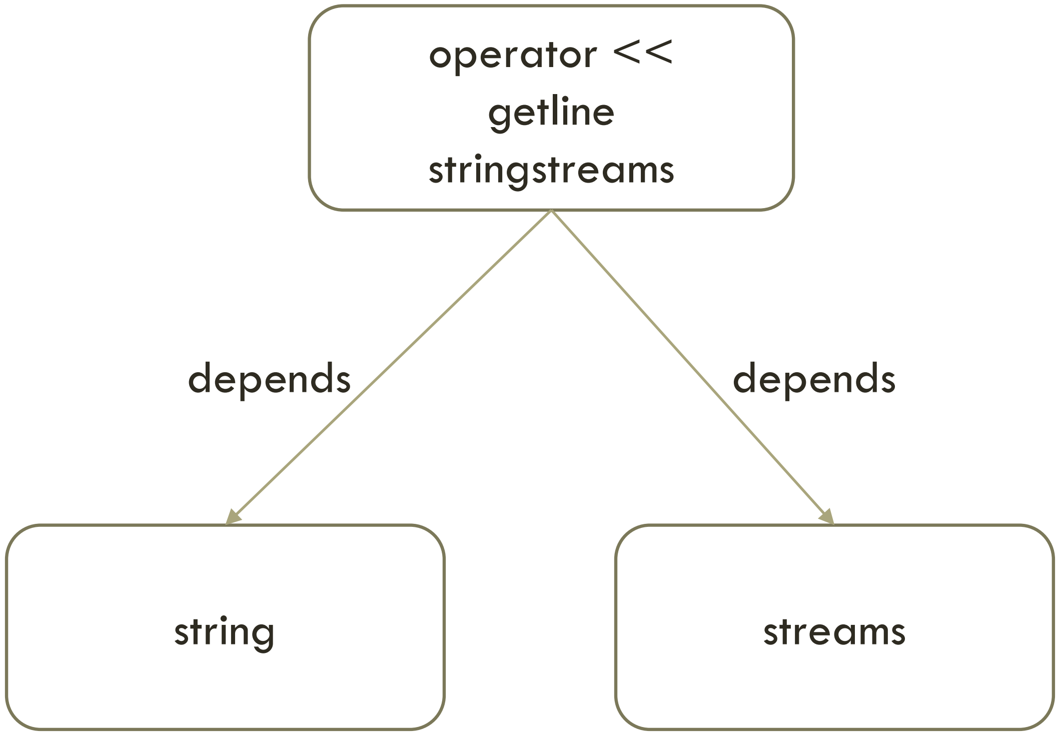
Polymorphic, nested master-detail in UI.

How make piece of UI from business object?

- Type switch? AbstractFactory? Don't handle inheritance
- Make business object return dialog…yuck
- Open method would neatly solve the problem

```
// business layer
class Person {
   virtual Dialog* dialog() = 0;
};
class NaturalPerson {
   virtual Dialog* dialog();
};
class LegalPerson {
   virtual Dialog* dialog();
};

// business stubs
Dialog* NaturalPerson::dialog() { assert(0); }
// etc

// presentation layer
Dialog* NaturalPerson::dialog() {
  return new PersonDialog(this);
}
// etc
```

AN EXAMPLE OF GOOD DESIGN



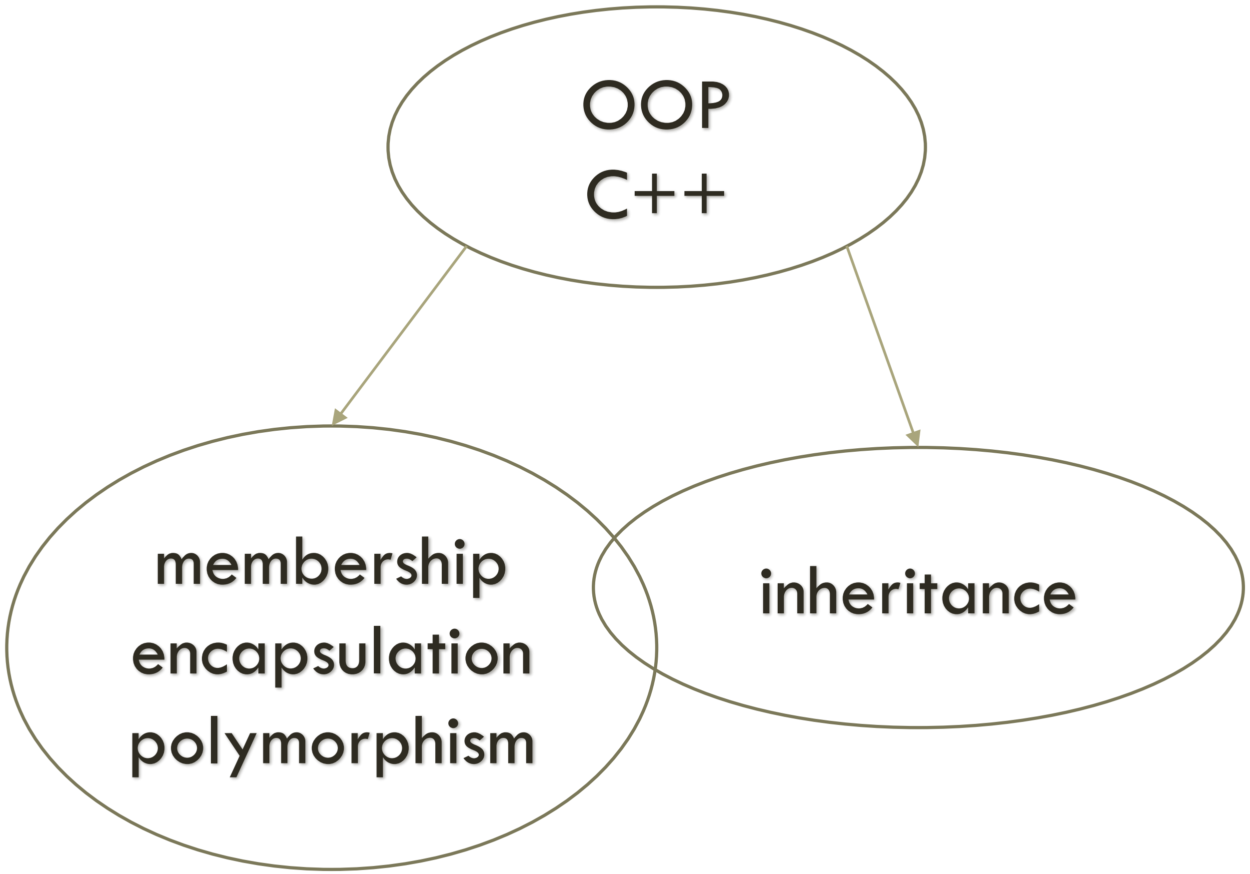
VIRTUAL *MEMBER* FUNCTIONS VIOLATE ENCAPSULATION

I want to be polymorphic

Therefore I must be a member function

Yay, access to the private parts!

OOP THE C++ WAY


OOP
C++
membership
encapsulation
polymorphism
inheritance

NO MULTIPLE DISPATCH

```
ordinary + ordinary = ordinary
diagonal + diagonal = diagonal
ordinary + diagonal = ordinary
diagonal + ordinary = ordinary
```

Alternatives exist:

- type switch
- double dispatch
- visitor

but are not satisfying:

- not extensible
- complex
- error-prone
- unmaintainable

THE RIGHT WAY : OPEN METHODS

```
class matrix {
  // ...
  virtual void write(ostream& os);
};

void matrix::write(ostream& os) {
  // ...
}

class diagonal : public matrix {
  virtual void write(ostream& os);
};

void diagonal::write(ostream& os) {
  // ...
}
```

```
class matrix {
  // ...
};

class diagonal : public matrix {
  // ...
};

void write(ostream& os, virtual matrix& m) {
  // ...
}

void write(ostream& os, virtual diagonal& m)
{
  // ...
}
```

pseudo C++ : no const, etc

IF NEEDED : OPEN *MULTI*-METHODS

```
non-extensible ugly machinery to
implement add() via type switch,
double dispatch or visitor, which
would not fit on this page
```

```
shared_ptr<matrix> add(
  virtual matrix& a, virtual matrix& b) {
  // general algorithm, add everything
}

shared_ptr<matrix> add(
  virtual diagonal& a, virtual diagonal& b) {
  // just add diagonals
  // return a diagonal matrix
}
```

INTRODUCING YOMM11

Open multi-methods in a library

Inspired by “Open Methods for C++” paper by Pirkelbauer, Solodkyy and Stroustrup

Also by “Optimizing multi-method dispatch using compressed dispatch tables” by Amiel, Gruber and Simon

Nice, easy to use syntax: unlimited number of virtual parameters freely mixed with non-virtual parameters

Fast dispatch via tables of pointers to functions

Compact dispatch tables

USING YOMM11- ADAPTING CLASSES

Intrusive mode:

- fast – almost as fast as native vfuncs
- stores a pointer to dispatch table in object
- MM_CLASS registers class
- MM_INIT adjusts dispatch pointer – must be called in each ctor

There is a non-intrusive mode:

- much slower
- uses map to get dispatch table from typeid

```
class matrix : public selector {
public:
  MM_CLASS(matrix);

  matrix(int rows, int cols) {
    MM_INIT();
    // ...
  }

  int rows() const;
  int cols() const;
  virtual double get(int, int) const = 0;
  virtual const vector<double>&
  elements(vector<double>&) const = 0;
  // ...
};

using any_matrix = shared_ptr<matrix>;
```

USING YOMM11- ADAPTING DERIVED CLASSES

List bases in MM_CLASS

- multiple and virtual inheritance ok

```
class ordinary : public matrix {
public:
  MM_CLASS(ordinary, matrix);
   // ...
};

class diagonal : public matrix {
public:
  MM_CLASS(diagonal, matrix);
};
```

USING YOMM11- DECLARING METHODS

- MULTI_METHOD declares a method
  - in header or implementation file
  - virtual_<> denotes virtual arguments
  - any number of vargs, anywhere
  - no-virtual args ok
  - overload not supported – use wrappers

```
MULTI_METHOD(add, // method name
  any_matrix,     // return type
  const virtual_<matrix>& m1,  // varg 1
  const virtual_<matrix>& m2); // varg 2

inline any_matrix operator +(
  const any_matrix& m1,
  const any_matrix& m2) {
   return add(*m1, *m2);
 }

MULTI_METHOD(write, void,
  ostream& os, // not a varg
  const virtual_<matrix>& m);

inline ostream& operator <<(ostream& os,
  const any_matrix& m) {
  write(os, *m); return os;
}
```

SPECIALIZING METHODS

BEGIN/END SPECIALIZATION

- in implementation file
- add a specialization to a method
- no virtual_<> around parameters here !
- inside specialization parameters have the right type – via static_cast if possible, dynamic_cast if it must
- specializations ARE NOT VISIBLE as overloads (unlike P/S/S multi-methods)

```
BEGIN_SPECIALIZATION(add, any_matrix,
  const matrix& m1, const matrix& m2) {
  // all purpose algorithm
  // return an ordinary matrix
} END_SPECIALIZATION;

BEGIN_SPECIALIZATION(add, any_matrix,
  const diagonal& m1,
  const diagonal& m2) {
  // only add diagonals
  // return a diagonal matrix
} END_SPECIALIZATION;
```

CALLING THE SUPER-METHOD

Inside a specialization, next(...) calls the next most specialized method.

```
BEGIN_SPECIALIZATION(write, void,
  ostream& os, const matrix& m) {
  // write all elements
} END_SPECIALIZATION;

BEGIN_SPECIALIZATION(write, void,
  ostream& os, const ordinary& m) {
  os << "ordinary\n";
  next(os, m);
} END_SPECIALIZATION;

BEGIN_SPECIALIZATION(write, void,
  ostream& os, const diagonal& m) {
  os << “diagonal\n";
  next(os, m);
} END_SPECIALIZATION;
```

CALLING A METHOD

The method name acts like a function taking the same parameters as passed to MULTI_METHOD (sans virtual_<> markers).

```
yorel::multi_methods::initialize();

double content[] = { 1, 2, 3, 4, 5, 6, 7,
8, 9 };

any_matrix m1 = make_shared<ordinary>(
  3, 3, content);
any_matrix m2 = make_shared<diagonal>(
  3, content + 2);
any_matrix sum = add(*m1, *m2);
cout << sum << "\n";

ordinary
4 2 3
4 9 6
7 8 14
```

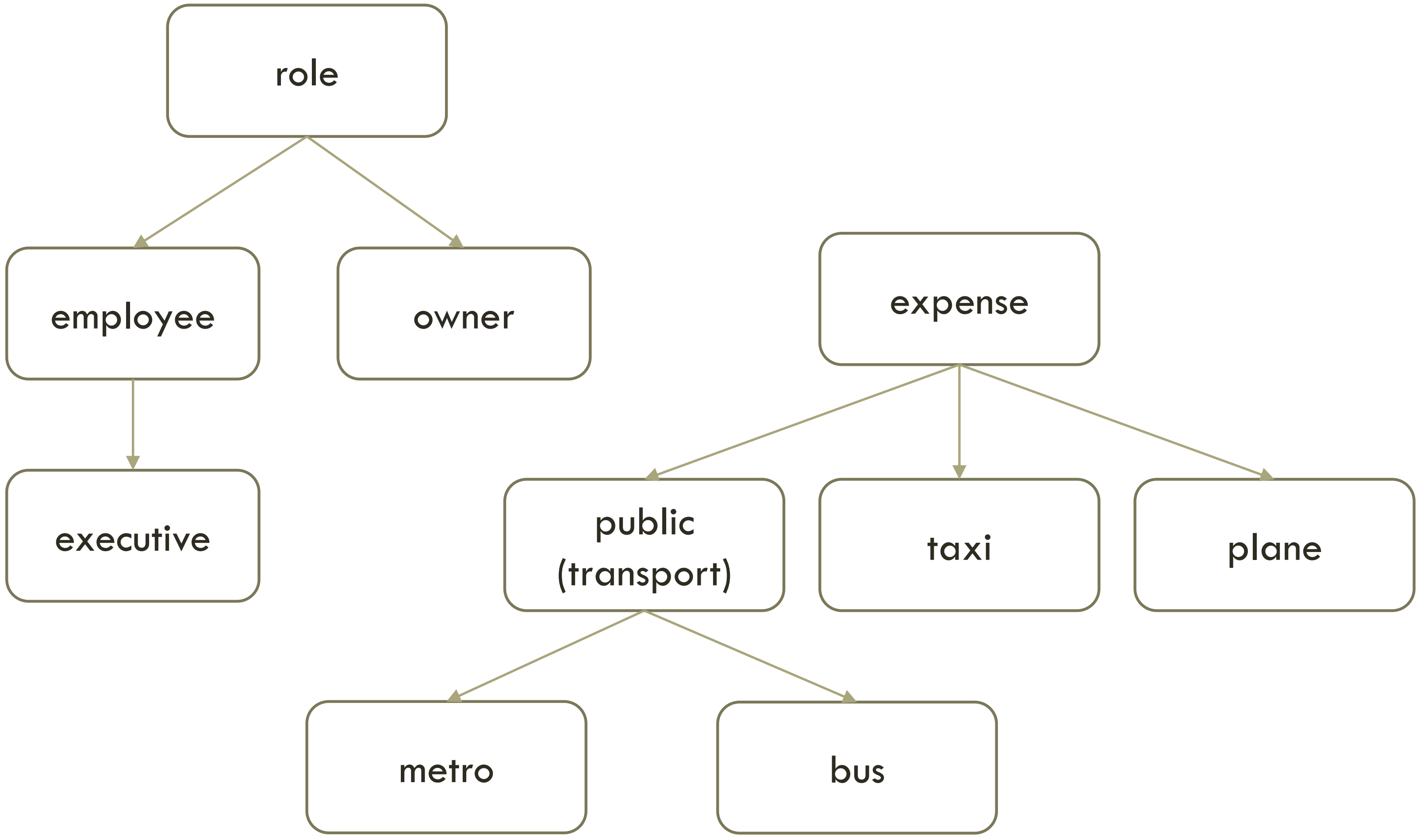
```
m1 = make_shared<diagonal>(
  3, content);
m2 = make_shared<diagonal>(
  3, content + 3);
any_matrix sum = add(*m1, *m2);
cout << sum << "\n";

diagonal
5 0 0
0 7 0
0 0 9
```

HOW DOES IT WORK?

Use tables of pointers to functions for speed

Use variadic macros and templates

(ANOTHER) EXAMPLE



bool pay(virtual_<employee>&)	
employee	salary
executive	salary + bonus

bool approve(virtual_<role>&, virtual_<expense>&)	
role, expense	false
employee, public	true
executive, taxi	true
owner,expense	true

DISPATCHING

Methods contain:

- a dispatch table – a N-dimensional table of pointers to functions (used during call)
- a method index – to convert from class to class index in one dimension of the dispatch table (used during call)
- a table of offsets - to convert tuple indexes in N dimensions to linear index (used during call)
- a vector of specializations (used during initialization)

Classes contain:

- a vector of class indexes, one per method defined or inherited by the class (used during call)
- pointers to base and derived classes, etc (used during initialization)

Objects contain (only in intrusive mode):

- one or several pointer to the object's class (used during call)

THE DISPATCH TABLE (NAIVE)

approve	expense	public	bus	metro	taxi	plane
role	(role,exp)	(role,exp)	(role,exp)	(role,exp)	(role,exp)	(role,exp)
employee	(role,exp)	(empl,public)	(empl,public)	(empl,public)	(role,exp)	(role,exp)
executive	(role,exp)	(empl,public)	(empl,public)	(empl,public)	(exec,taxi)	(role,exp)
owner	(owner,exp)	(owner,exp)	(owner,exp)	(owner,exp)	(owner,exp)	(owner,exp)

table is filled with "best" specialization for each pair

best = same as in overload resolution

can lead to ambiguities

approve	
role, expense	false
employee, public	true
executive, taxi	true
owner,expense	true

THE DISPATCH TABLE – "COMPRESSED"

approve	expense	public bus metro	taxi	plane
role	(role,exp)	(role,exp)	(role,exp)	(role,exp)
employee	(role,exp)	(empl,public)	(role,exp)	(role,exp)
executive	(role,exp)	(empl,public)	(exec,taxi)	(role,exp)
owner	(owner,exp)	(owner,exp)	(owner,exp)	(owner,exp)

approve	
role, expense	false
employee, public	true
executive, taxi	true
owner,expense	true

DISPATCHING A CALL

METHOD INDEXES

method	index	
approve	0	0
pay	1	

TABLE OFFSETS

method	offsets	
pay	1	
approve	1	4

CLASS INDEXES

class	offsets	
role	0	
empl	1	0
exec	2	1
owner	3	
exp	0	
public	1	
bus	1	
metro	1	
taxi	2	
plane	3	

DISPATCH TABLES

pay	
empl	(empl)
exec	(exec)

approve	exp	public bus metro	taxi	plane
role	(role,exp)	(role,exp)	(role,exp)	(role,exp)
empl	(role,exp)	(empl,public)	(role,exp)	(role,exp)
exec	(role,exp)	(empl,public)	(exec,taxi)	(role,exp)
owner	(owner,exp)	(owner,exp)	(owner,exp)	(owner,exp)

approve(exec,plane) ➔ method index = 0 ➔ row = exec[0] = 2 , col = plane[0] = 3
➔ call approve[2][3] ➔ pay[2 * 1 + 3 * 4](exec, plane)

PERFORMANCE

	body	time	d%
virtual function	empty	75.5	
open method, intrusive	empty	100.6	33%
open method, foreign	empty	1397.8	
virtual function	math	1541.1	
open method, intrusive	math	1608.2	4%
open method, foreign	math	2607.8	
virtual function, 2-dispatch	empty	250.8	
open method with 2 args, intrusive	empty	150.8	-40%
open method with 2 args, foreign	empty	2832.3	

```
math = log(a * x * x + b * x + c)
```

Greatly depends on compiler's willingness to inline

Close to native vfuncs for 1 argument

Beat double dispatch

CONCLUSION, QUESTIONS, ANSWERS

Virtual member functions are BAD !

Use them only when you need both polymorphism and access to private parts

Open multi-methods are a natural extension of C++

Hope to see them in the language some day...the sooner the better

In the meantime Yomm11 lets you experiment and use OMM

Questions?

LINKS, CONTACT

P. Pirkelbauer, Y. Solodkyy and B. Stroustrup, Open Multi-Methods for C++

E. Amiel, E. Dujatrdin and E. Simon, Fast Algorithms for Compressed Multi-Method Dispatch Tables Generation

Yomm11 on Code Project

Yomm11 on GitHub

Yomm11 documentation

Author: Jean-Louis Leroy, jl@leroy.nyc

```
class foo {
    virtual void bar();
};
```

ANNEXES

MULTI_METHOD(...)

```
MULTI_METHOD(approve, bool,
  const virtual_<role>&, const virtual_<expense>&);

template<typename Sig> struct approve_specialization;

const ::yorel::multi_methods::
multi_method<
  approve_specialization,
  bool(const virtual_<role>&, const virtual_<expense>&)
> approve;
```

BEGIN_SPECIALIZATION(...)

```
BEGIN_SPECIALIZATION(approve, bool, const role&, const
expense&) { return false; } END_SPECIALIZATION

template<> struct approve_specialization<Sig> :
decltype(approve)::specialization<
  approve_specialization<Sig> >
{
  virtual void* _yomm11_install() {
    return &...::register_spec<decltype(approve),
        approve_specialization>::the; }
  using body_signature = ...;
  static bool body(const role&, const expense&) { {
    return false;
  } }
};
```

ALLOCATION OF CLASS INDEXES

let V be a vector of bit masks, initially empty
sort classes, bases first

for each class C_i in sorted vector of classes:

- let M_i be the mask associated to class C_i
- for each method, for each virtual argument of this class:
  - search V for a mask V_j such that V_j & M_i = 0
  - if there is one:
    - $P_i = j$
    - $V_j = V_j \mid M_i$
  - otherwise:
    - P_i = size of V
    - append M_i to V
  - append P_i to the method's slot table

CONSTRUCTION OF THE DISPATCH TABLE - DEFINITIONS

Let A and B be two classes:

- $A = B$ means that A and B are the same class.
- $A < B$ means that A is a subclass of B.
- $A <= B$ means either $A = B$ or $A < B$

Applicable specialization: A specialization $S(A_1,...,A_n)$ is *applicable* for argument *i* to a class B iff $A_i <= B$. The specialization is applicable to a n-uplet of classes $\{ B_1,...,B_n \}$ if it is applicable for all its arguments.

More specific specialization:
A specialization $S_1(A_1,...,A_n)$ is *more specific* than a specialization $S_2(B_1,...,B_n)$ iff:

- There is at least one argument *i* for which $A_i < B_i$
- There is no argument *i* for which $B_i < A_i$

Most specific specialization: Given a collection of specialisations of the same method; the *most specific specialization* is the specialization that is more specific than all the others. It may not exist.

CONSTRUCTION OF THE DISPATCH TABLE - ALGORITHM

Given:

- A method $M(B_1,\ldots,B_n)$
- A collection of specialisations $S_i(D_1,\ldots,D_n)$ of this method, where $C_i <= H_i$.

1. First let us consider the arguments taken separately. For each argument *i*:
   a. Make the set H_i of classes H_{ij} such that $H_{ij} <= B_i$. In other words, H_i is the hierarchy rooted in B_i.
   b. For each class in H_i, make the set $A(H_i)$ of the applicable specialisations for argument *i.*
   c. Make the partition P_i according to the equivalence relationship: $A(X) = A(Y)$.
      In other words, group the classes that have the same sets of applicable specialisations for argument *i.*
      Each group G_{ij} becomes a row in dimension *i* of the dispatch table and its index *j* is written in the index table of each class belonging to the group.
2. Make the cross product of partitions P_i. Elements X_k of this product consist in sets of G_{ij} groups of classes, and correspond to cells in the dispatch table.
   For each element X_k of the cross product:
   a. For each group G_{ij}: make the set A of the specialisations applicable to the classes in Gij.
   b. Make the intersection S_k of the sets A_{ij} - it is a set of specialisations.
   c. Find the most specific specialization S_1 in S_k. If it exists, write it to the dispatch table.
      Find the most specific specialization S_2 in the set Sk - S1. If it exists, it is the specialization returned by next in S_1.