Run-Time Type Information and Class Design

Annotations to Stroustrup & Lenkov

Doug Lea
Computer Science, SUNY Oswego, Oswego NY 13126
and
New York CASE Center, Syracuse University, Syracuse NY 13244
dl@g.oswego.edu

Abstract

Design applications of the run-time type identification constructs proposed by Stroustrup and Lenkov are illustrated via several examples that demonstrate their strengths and weaknesses as tools in object-oriented design.

Some people think that run-time type identification (RTTI) constructs cause programmers to sidestep many of the good design practices evident in well-crafted object-oriented programs. Others think that it is impossible to even write well-crafted object-oriented programs without run-time type support. This commentary paper briefly attempts to disentangle some of the issues behind such views. A few problems are illustrated for which RTTI might plausibly be used to help formulate a solution. These lead to a discussion of some underlying design and engineering considerations, and allow some tentative conclusions (noted within boxes throughout the paper).

Example 1

The first example involves probably the most common application of RTTI. Assume a base class, along with a subclass that possesses additional properties not listed in the base. For example:

```
class Person  { ... };
class Employee : public Person {
public:
  virtual float  salary() const;
  virtual Department* dept();
  ...
};
```

Along with a heterogeneous collection class, for example:

```
class PersonList  {
public:
  Person* first();
  Person* next(Person*);
  ...
};
```

And finally the problem:

> Write `sumSalaries(PersonList* l)`, that returns the sum of all salaries in `l`.

This is an impossible demand, since `PersonList` entries don't necessarily possess `salary` attributes. At best, we can sum the salaries for all (sub)`Person`s known to contain a `salary`. In doing so, we might arbitrarily decide to treat all others as having a salary of zero. Given this, a solution may be had using RTTI:

```
float sumSalaries(PersonList* l)
{
  float sum = 0.0;
  for (Person* p = l->first(); p != 0; p = l->next(p))
    if (Employee* e = (Employee*)p) sum += e->salary();
  return sum;
}
```

RTTI can make heterogeneous collections more usable in C++.

Post Hoc Attributes

The `sumSalaries` procedure might be seen as implicitly attaching a *new* property to class `Person`, namely:

```
class Person  {  ...
  virtual float salary() const { return 0.0; }
};
```

The fact that this was done *implicitly* seems innocent enough. But what if some other procedure having to do with salaries and persons made a different decision; e.g., that unless specified, the salary of a `Person` should be estimated as the average yearly per capita income? This is the sort of software management problem that classes, encapsulation, and inheritance were meant to *solve*, not *create*. And this is the sort of usage that gives RTTI a bad reputation.

It would have been much better to build the default `salary` attribute into `Person` to begin with. But perhaps the class "belongs" to someone else and cannot be changed. Perhaps changes would break other existing code. There are many such reasons for not touching class interfaces when you don't absolutely have to. There is a nicer-looking solution. It may be approached through a version that looks even nicer still, but does not work as naively expected:

```
float getSalary(Person* p)   { return 0.0; }              // wrong
float getSalary(Employee* p) { return p->salary(); }

float sumSalaries_2(PersonList* l)
{
  float sum = 0.0;
  for (Person* p = l->first(); p != 0; p = l->next(p))
    sum += getSalary(p);
  return sum;
}
```

This is better than the original version, since the decision to treat non-existing salaries as zero is clearly enshrined within independent procedures that all other classes and procedures may use.

Unfortunately, the code does not solve the problem. C++ does not dynamically dispatch procedures on the basis of *arguments*, only *receivers*[1]. Thus `sumSalaries_2` would always return zero. However, *this* can be fixed using RTTI:

```
float getSalary(Person* p)
{
  Employee* e;
  if (e = (Employee*)p) return e->salary(); else return 0.0;
}
```

RTTI can simulate and implement dynamic argument-based dispatching.

[1] and only when declared virtual, etc.

Class tests and feature tests

All is well with the above solution until the day someone adds:

```
class  Contractor : public Person { ...
  virtual float salary() const;
  virtual Job* job();
};
```

`Contractor`s aren't `Employee`s, yet they also have `salary` attributes. If a `Contractor` ever shows up in a `PersonList`, then both `sumSalaries` and `sumSalaries_2` will treat its salary as zero. This is probably not what anyone had in mind.

The problem is that the *class name* `Employee` was an alias for possession of the *property* (method) `salary`. This trick works only when it works.

RTTI cannot be used to infer features unless classes have been designed to support this to begin with.

In the current example there are several cures, including:

1. Finally add `salary` to `Person`.
2. Add subclass `SalariedPerson: public Person` and adjust the `Employee` and `Contractor` class declarations accordingly.
3. Add class `Salaried` as a "mixin" class, and adjust `Employee` and `Contractor` classes to multiply inherit both `Salaried` and `Person`.
4. Rewrite `getSalary` to investigate possession of `salary` through `Type_info` information rather than through the conditional cast mechanism.

Each of these has its merits. Each also requires changes to existing code after the introduction of `Contractor`. RTTI does not always eliminate the need for such alterations.

RTTI can postpone necessary refactorings.

Note however that any of these strategies *could* have been applied in our original versions. People tend not to do so though. Routine creation of extremely fine-grained classes corresponding to *each* "added" property gets pretty tedious, as does the alternative of routinely extracting `Type_info` information probing for possession of these properties. These human-factors considerations are sometimes serious barriers to extensibility. RTTI offers an incomplete solution. (Other equally incomplete solutions include `views` [5] and *conformance* based typing [4].)

Example 2

This example was made famous in a set of `Usenet` postings:

```
class Driver { ... };
class ProDriver : public Driver { ... };

class  Vehicle { ...
  virtual void Register(Driver* d) { vd(); }
};

class Truck : public Vehicle { ...
  void Register(ProDriver* d) { tp(); }
};
```

The *idea* here seems to be that a `Vehicle` may be registered to any kind of `Driver`, but a `Truck` may only be registered (in some perhaps different way, as signified by `tp()` vs `vd()`) to a `ProDriver` (professional driver).

The above declarations are not quite illegal C++[2] but do not work as expected. For example,

```
void reg(Vehicle* v, Driver* d) { v.Register(d); }

main() {
  Truck* t = new Truck;
  ProDriver* p = new ProDriver;
  t.Register(p);         // Truck::Register(ProDriver*) invoked
  reg(t, p);             // Vehicle::Register(Driver*) invoked via reg
}
```

This is a more subtle consequence of C++ rules that dynamically dispatch only on receiver, not argument types.

Multimethods

A cure may be obtained by "pulling out" the `Register` method from the classes and using RTTI.

```
void Register(Vehicle* v, Driver* d) {
  if ((Truck*)(v) && (ProDriver*)(p)) tp(); else vd();
}
```

This style of specialization based on the types of (potentially) *ALL* participants in an operation is called *multimethod dispatching.* CLOS [1] is justly famous for supporting multimethods as first-class programming constructs.[3] If C++ supported multimethods directly, then this might have been written somewhat more clearly and extensibly:

```
void Register(Vehicle* v, Driver* d)  { vd(); }

void Register(Truck* v, ProDriver* d) { tp(); }
```

RTTI can simulate and implement multimethods.

Types as Guards

The simulated multimethod solution has the advantage of predictable dispatching. This is vital in order to statically determine correctness, or even reasonableness. It's hard to say very much at all about the original version. In practice, using RTTI-simulated multimethods to control dispatching of special cases of overloaded methods and procedures is much safer and more reliable than depending on C++ "overload resolution" policies.

But in the current example, the improved clarity highlights *conceptual* problems with the design. The probable intent was to *disallow* all but `ProDriver`s from registering `Truck`s. The above solution allows `Driver`s to register them, but uses the `vd()` code in `Vehicle::Register` to do so. The use of multimethod dispatch seems like the wrong way to address this. It uses type information to *direct*, not *guard* or prohibit certain calls.

In these kinds of designs, there is simply no way to statically prohibit certain argument combinations. The special cases must be considered truly *exceptional* to the general `Vehicle-Driver` relationship. Probably the best solution here would be to explicitly indicate possible failure to clients. This could be done in several ways, including:

[2] See [2] chapter 13 for the gruesome details.

[3] The remarks in [6] about specifically not including multimethods in their proposal seem misplaced given that many *uses* of RTTI amount to their simulation. The main difference and advantage of first-class multimethods is that they are extensible – new special cases may be added without modifying existing code. In any case, multimethods and RTTI can each fully simulate the other.

```
bool Register(Vehicle* v, Driver* d) {
  if ((Truck*)(v))
    if ((ProDriver*)(p)) { tp(); return TRUE; }
    else return FALSE;
  else { vd(); return TRUE; }
}
```

RTTI can detect exceptional argument combinations that cannot be statically prohibited.

Example 3

Suppose we are building a class representing face icons that may be in any of three states, happy, sad, and asleep. One design strategy is to create three different classes, one per state and a "controller", that switches among them. This is an attractive *delegation* [3] based design:

```
class IconState {
  virtual bool  eyesOpen() const = 0;
  ...
};

class HappyIcon : public IconState {
  bool  eyesOpen() const { return TRUE; }
  ...
};

class AsleepIcon : public IconState { ... }
class SadIcon : public IconState { ... }

class Icon {
  IconState* theIcon;

  virtual bool eyesOpen const { return theIcon->eyesOpen(); }
  ...
  virtual bool isHappy const { return ((HappyIcon*)(theIcon) != 0); }
};
```

This is a situation in which RTTI is clearly the *best* alternative. How else would an `Icon` know which state it were in within `isHappy`? Alternatives like maintaining logical variables invite needless error-prone complexity.

Importantly, this strategy extends to testing and internal integrity checking. For example, the `Icon` class requires a `beHappy` method to change state:

```
  ...
  virtual void beHappy() {
    theIcon = getHappyIcon();
    assert(isHappy());
  }
```

The main reason this works so nicely here is that we have carefully merged the notions of class membership and property (and/or property value) possession. This takes some planning.

RTTI can be used to prescribe, determine, and verify logical state.

Example 4

Suppose we need to design a long-lived application program with fault-tolerance support in case of crashes. We settle on a checkpoint/rollback scheme in which the states of all objects are periodically saved on disk.

Recovery is performed by re-constructing or reinitializing (depending on the nature of the crash) all objects to their last saved states.

Design and implementation of such mechanisms is not an easy matter. Doing a thorough job is tantamount to the construction of an object-oriented database system. But there is the widespread belief that RTTI substantially simplifies practical application-specific solutions.

"Substantially" is much too strong a term here. Most of the snags in this kind of persistence support revolve around the transformation and associated bookkeeping of *object* identities, that are internally represented through pointer values, but externally through some other scheme (e.g., integer pseudo-identities). *Class* identities must also be stored and recovered in order to allow reconstruction. RTTI *per se* can only assist in only the latter.

RTTI does not automate persistence support.

Save / Restore Mechanics

On the other hand, RTTI certainly doesn't make this any *harder*. There are many ways to design a save/restore mechanism. Here is a simplified prototypical framework[4]. On the save side, for each object to be stored:

1. Output an external pseudo-identity uniquely corresponding to its internal ID (address)
2. Output an external pseudo-identity uniquely corresponding to its maximal internal class ID.
3. Output all values of "simple" state attributes of built-in type.
4. Output all pseudo-IDs corresponding to pointer attributes.
5. Ensure that at some point values and pseudo-IDs for all `static` class data for this object's class are saved.

The restore side is mostly symmetrical. For each object to be recovered:

1. Read in an external object pseudo-ID. Map it either to an existing internal ID or to one that is to be constructed (depending on the kind of recovery).
2. Read in an external class pseudo-ID. Use it as a key to dispatch to a routine that constructs or reinitializes the object using the value and pointer information to follow. In other words, the "driver" routine is a big `switch` statement, although perhaps a well-disguised one. It may also need to queue or reorder requests in order to delay the construction of objects until their components exist.
3. Somehow separately handle `static` class data.

Metaclasses

There are a number of ways in which RTTI can make these tasks a bit easier to implement, without otherwise affecting their logic one way or the other. These mainly arise through exploitation of extensible `Type_info` structures.

- `typeid` values could be used as the internal class pseudo-IDs. This simplifies dispatch logic in the recovery routine.
- In particular, access to the (re)initialization routines could be arranged through `Type_info` structures indexed by the `typeid`.

[4] For example, among the simplifications is that it does not accommodate "embedded" objects; i.e., those that directly nest one object within another.

- Maps between internal and external IDs could be located in per-class `Type_info` structures.
- Bookkeeping on recovery of `static` data could be located in `Type_info` structures.

It should soon occur to anyone familiar with languages like Smalltalk that the logic of grouping these kinds of per-class bookkeeping routines in a central place leads to the notion of *metaclasses* as a replacement for C++-style `static` class functions including, significantly, client-accessible *constructors.* This may in turn lead to a very different style of class design in general – for many purposes, `typeid(p).info()` might as well be pronounced "p's metaclass".

RTTI and extensible `Type_info` *classes can simulate metaclasses.*

References

[1] Bobrow, D., L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, & D. Moon, "Common Lisp Object System", *SIGPLAN Notices*, September, 1988.

[2] Ellis, M., & B. Stroustrup. *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[3] Johnson, R., & J. Zweig. "Delegation in C++", *Journal of Object-Oriented Programming*, November, 1991.

[4] Russo, V. & E. Granston, "Signature-based polymorphism for C++" *Proceedings, 1991 Usenix C++ Conference*, Washington, 1991.

[5] Scholl, M, C. Laasch, & M. Tresch, "Updatable views in object-oriented databases", in C. Delobel, M. Kifer & Y. Masunaga (eds.) *Deductive and Object-Oriented Databases*, Springer-Verlhag, 1991.

[6] Stroustrup, B., & D. Lenkov, "Run-time type identification for C++". *Proceedings, 1992 Usenix C++ Conference*, Portland, 1992.