
# How to set up LLVM-style RTTI for your class hierarchy

*Section author: Sean Silva <silvas@purdue.edu>*

**Contents**



## Background

LLVM avoids using C++'s built in RTTI. Instead, it pervasively uses its own hand-rolled form of RTTI which is much more efficient and flexible, although it requires a bit more work from you as a class author.

A description of how to use LLVM-style RTTI from a client's perspective is given in the Programmer's Manual. This document, in contrast, discusses the steps you need to take as a class hierarchy author to make LLVM-style RTTI available to your clients.

Before diving in, make sure that you are familiar with the Object Oriented Programming concept of "is-a".

## Basic Setup

This section describes how to set up the most basic form of LLVM-style RTTI (which is sufficient for 99.9% of the cases). We will set up LLVM-style RTTI for this class hierarchy:

```
class Shape {
public:
  Shape() {};
  virtual double computeArea() = 0;
};

class Square : public Shape {
  double SideLength;
public:
  Square(double S) : SideLength(S) {}
  double computeArea() /* override */;
};

class Circle : public Shape {
  double Radius;
public:
  Circle(double R) : Radius(R) {}
  double computeArea() /* override */;
};
```

The most basic working setup for LLVM-style RTTI requires the following steps:

1. In the header where you declare `Shape`, you will want to `#include "llvm/Support/Casting.h"`, which

declares LLVM's RTTI templates. That way your clients don't even have to think about it.

```
#include "llvm/Support/Casting.h"
```

2. In the base class, introduce an enum which discriminates all of the different classes in the hierarchy, and stash the enum value somewhere in the base class.

   Here is the code after introducing this change:

```
 class Shape {
 public:
+  /// Discriminator for LLVM-style RTTI (dyn_cast<> et al.)
+  enum ShapeKind {
+    SquareKind,
+    CircleKind
+  };
+private:
+  const ShapeKind Kind;
+public:
+  ShapeKind getKind() const { return Kind; }
+
   Shape() {};
   virtual double computeArea() = 0;
 };
```

   You will usually want to keep the `Kind` member encapsulated and private, but let the enum `ShapeKind` be public along with providing a `getKind()` method. This is convenient for clients so that they can do a `switch` over the enum.

   A common naming convention is that these enums are "kind"s, to avoid ambiguity with the words "type" or "class" which have overloaded meanings in many contexts within LLVM. Sometimes there will be a natural name for it, like "opcode". Don't bikeshed over this; when in doubt use `Kind`.

   You might wonder why the `Kind` enum doesn't have an entry for `Shape`. The reason for this is that since `Shape` is abstract (`computeArea() = 0;`), you will never actually have non-derived instances of exactly that class (only subclasses). See Concrete Bases and Deeper Hierarchies for information on how to deal with non-abstract bases. It's worth mentioning here that unlike `dynamic_cast<>`, LLVM-style RTTI can be used (and is often used) for classes that don't have v-tables.

3. Next, you need to make sure that the `Kind` gets initialized to the value corresponding to the dynamic type of the class. Typically, you will want to have it be an argument to the constructor of the base class, and then pass in the respective `XXXKind` from subclass constructors.

   Here is the code after that change:

```
 class Shape {
 public:
   /// Discriminator for LLVM-style RTTI (dyn_cast<> et al.)
   enum ShapeKind {
     SquareKind,
     CircleKind
   };
 private:
   const ShapeKind Kind;
 public:
   ShapeKind getKind() const { return Kind; }

-  Shape() {};
+  Shape(ShapeKind K) : Kind(K) {};
```

```
   virtual double computeArea() = 0;
 };

 class Square : public Shape {
   double SideLength;
 public:
-  Square(double S) : SideLength(S) {}
+  Square(double S) : Shape(SquareKind), SideLength(S) {}
   double computeArea() /* override */;
 };

 class Circle : public Shape {
   double Radius;
 public:
-  Circle(double R) : Radius(R) {}
+  Circle(double R) : Shape(CircleKind), Radius(R) {}
   double computeArea() /* override */;
 };
```

4. Finally, you need to inform LLVM's RTTI templates how to dynamically determine the type of a class (i.e. whether the `isa<>`/`dyn_cast<>` should succeed). The default "99.9% of use cases" way to accomplish this is through a small static member function `classof`. In order to have proper context for an explanation, we will display this code first, and then below describe each part:

```
 class Shape {
 public:
   /// Discriminator for LLVM-style RTTI (dyn_cast<> et al.)
   enum ShapeKind {
     SquareKind,
     CircleKind
   };
 private:
   const ShapeKind Kind;
 public:
   ShapeKind getKind() const { return Kind; }

   Shape(ShapeKind K) : Kind(K) {};
   virtual double computeArea() = 0;
+
+  static bool classof(const Shape *) { return true; }
 };

 class Square : public Shape {
   double SideLength;
 public:
   Square(double S) : Shape(SquareKind), SideLength(S) {}
   double computeArea() /* override */;
+
+  static bool classof(const Square *) { return true; }
+  static bool classof(const Shape *S) {
+    return S->getKind() == SquareKind;
+  }
 };

 class Circle : public Shape {
   double Radius;
 public:
   Circle(double R) : Shape(CircleKind), Radius(R) {}
   double computeArea() /* override */;
+
+  static bool classof(const Circle *) { return true; }
+  static bool classof(const Shape *S) {
+    return S->getKind() == CircleKind;
+  }
 };
```

Basically, the job of `classof` is to return `true` if its argument is of the enclosing class's type. As you can see, there are two general overloads of `classof` in use here.

1. The first, which just returns `true`, means that if we know that the argument of the cast is of the enclosing type *at compile time*, then we don't need to bother to check anything since we already know that the type is convertible. This is an optimization for the case that we statically know the conversion is OK.
2. The other overload takes a pointer to an object of the base of the class hierarchy: this is the "general case" of the cast. We need to check the `Kind` to dynamically decide if the argument is of (or derived from) the enclosing type.

To be more precise, let `classof` be inside a class `C`. Then the contract for `classof` is "return `true` if the argument is-a `C`". As long as your implementation fulfills this contract, you can tweak and optimize it as much as you want.

Although for this small example setting up LLVM-style RTTI seems like a lot of "boilerplate", if your classes are doing anything interesting then this will end up being a tiny fraction of the code.

## Concrete Bases and Deeper Hierarchies

For concrete bases (i.e. non-abstract interior nodes of the inheritance tree), the `Kind` check inside `classof` needs to be a bit more complicated. Say that `SpecialSquare` and `OtherSpecialSquare` derive from `Square`, and so `ShapeKind` becomes:

```
 enum ShapeKind {
   SquareKind,
+  SpecialSquareKind,
+  OtherSpecialSquareKind,
   CircleKind
 }
```

Then in `Square`, we would need to modify the `classof` like so:

```
   static bool classof(const Square *) { return true; }
-  static bool classof(const Shape *S) {
-    return S->getKind() == SquareKind;
-  }
+  static bool classof(const Shape *S) {
+    return S->getKind() >= SquareKind &&
+           S->getKind() <= OtherSpecialSquareKind;
+  }
```

The reason that we need to test a range like this instead of just equality is that both `SpecialSquare` and `OtherSpecialSquare` "is-a" `Square`, and so `classof` needs to return `true` for them.

This approach can be made to scale to arbitrarily deep hierarchies. The trick is that you arrange the enum values so that they correspond to a preorder traversal of the class hierarchy tree. With that arrangement, all subclass tests can be done with two comparisons as shown above. If you just list the class hierarchy like a list of bullet points, you'll get the ordering right:

```
| Shape
  | Square
    | SpecialSquare
    | OtherSpecialSquare
  | Circle
```