Run-Time Type Information

Rupesh Nasre.

OOAIA
January 2025

Introduction

- RTTI provides type information at run-time.
- Also referred to as *type introspection*.
- Supported in languages such as C++, PHP, ADA, and Python.
- Not supported in languages such as C, Pascal.
- In the original C++ design, Stroustrup did not include RTTI, as he thought it would get misused.
  - Prioritize compile-time checks.

RTTI via *typeid*

- C++ provides RTTI with keyword *typeid*.

```
int myint = 50;
std::string mystr = "string";
double *mydoubleptr = nullptr;

std::cout << "myint has type: " << typeid(myint).name() << '\n'
          << "mystr has type: " << typeid(mystr).name() << '\n'
          << "mydoubleptr has type: " << typeid(mydoubleptr).name() << '\n';
```

```
myint has type: i
mystr has type: NSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE
mydoubleptr has type: Pd
```

Nested Types

```
pair<int, double> simplepair;
pair<pair<pair<int, double>, char>, int> deeppair;
std::cout << "simplepair has type: " << typeid(simplepair).name() << '\n';
std::cout << "deeppair has type: " << typeid(deeppair).name() << '\n';
```

simplepair has type: St4pairIidE
deeppair has type: St4pairIS_IS_IidEcEiE

cout and printf

```
// std::cout << myint is a glvalue expression of polymorphic type; it is evaluated
 const std::type_info& r1 = typeid(std::cout << myint); // side-effect: prints 50
 std::cout << '\n' << "std::cout<<myint has type : " << r1.name() << '\n';

 // std::printf() is not a glvalue expression of polymorphic type; NOT evaluated
 const std::type_info& r2 = typeid(std::printf("%d\n", myint));
 std::cout << "printf(\"%d\\n\",myint) has type : " << r2.name() << '\n';
```

```
50
std::cout<<myint has type : So
printf("%d\n",myint) has type : i
```

Polymorphic vs. Non-polymorphic

```
struct Base2 { virtual void foo() {} }; // polymorphic
struct Derived2 : Base2 {};
...
try {
        double ******ptr = nullptr;
        std::cout << "****ptr points to " << typeid(****ptr).name() << '\n';

        // dereferencing a null pointer: not okay for a polymorphic lvalue
        Derived2* bad_ptr = nullptr;
        std::cout << "bad_ptr points to... ";
        std::cout << typeid(*bad_ptr).name() << '\n';
}
catch (const std::bad_typeid& e) {
        std::cout << " caught " << e.what() << '\n';
}
```

Does not get evaluated.

Polymorphic type, gets evaluated.

****ptr points to PPd
bad_ptr points to... caught std::bad_typeid

RTTI for Polymorphic Types

- RTTI is only for polymorphic types.
- For non-polymorphic types, the type is derivable at compile-time.
- Hence, if we do not use virtual functions, all types are available at compile-time.
- RTTI can increase the size of the binary.
  - typeid is a constant time procedure.
  - *-fno-rtti* can be used to disable RTTI. Forbids usage of *typeid* and *dynamic_cast*.

A requirement

```
void fun(Base *ptr) {
        // if ptr points to Derived, I want to do something specific.
        // else, I want to do generic stuff.
}
int main(int a, char **b) {
        Base *bp;

        if (a == 1) bp = new Derived();
        else bp = new Base();

        fun(bp);

        return 0;
}
```

A requirement

```
void fun(Base *ptr) {
    if (typeid(ptr).name() == "P7Derived")    // non-portable
        doSpecific(?);      // or ptr->doSpecific()
    else
        doGeneric(ptr);     // or ptr->doGeneric()
}
int main(int a, char **b) {
    Base *bp;

    if (a == 1) bp = new Derived();
    else bp = new Base();

    fun(bp);

    return 0;
}
```

- C-type type-cast is not recommended.
- We need a type-cast which understands inheritance.

dynamic_cast

- Specialize a variable to a derived class.
  - downcast
- The cast target must be a pointer or a reference to a class.
  - Contrast it with type-cast or static_cast
- Performs a type-safety check at runtime.
  - throws bad_cast if incompatible.

A requirement

```
void fun(Base *ptr) {
  try {
      Derived *dp = dynamic_cast<Derived *>(ptr);
      doSpecific(dp);     // or dp->doSpecific()
  } catch (std::bad_cast &e) {
      doGeneric(ptr);     // or ptr->doGeneric()
  }
}
int main(int a, char **b) {
        Base *bp;

        if (a == 1) bp = new Derived();
        else bp = new Base();

        fun(bp);

        return 0;
}
```

vs. static_cast

```
struct B {};
struct D : B { B b; };

D d;
B& br1 = d;
B& br2 = d.b;

static_cast<D&>(br1); // OK, lvalue denoting the original “d” object
static_cast<D&>(br2); // Undefined behavior: d.b is not the derived object
```

- static_cast conversion happens at compile time. No run-time checks are performed.
- static_cast cannot remove *const* and *volatile* properties.
- Therefore, it may not always be safe.
  - Programmer needs to know that the type conversion is safe.

const_cast

- Removes constness of an expression.
- Works with objects and class hierarchies.

```
int i = 3;                          // i is not declared const
const int& rci = i;
const_cast<int&>(rci) = 4;          // OK: modifies i
std::cout << "i = " << i << '\n';
```

reinterpret_cast vs. static_cast

```
int x;
int *ip = &x;
B *bp;

bp = (B*)ip;                          // Compiles, but ...
bp = static_cast<B*>(ip);             // Compile-time error
bp = reinterpret_cast<B*>(ip);        // Compiles.
```

- reinterpret_cast guarantees that if we convert from T1 to T2, and then T2 back to T1, you get the original value back.
- This is useful with opaque data types (such as external libraries or vendors).

Value Categories

Category	Has identity?	Can be moved from?
lvalue	Yes	No
xvalue	Yes	Yes
prvalue	No	Yes in C++11 No in C++17