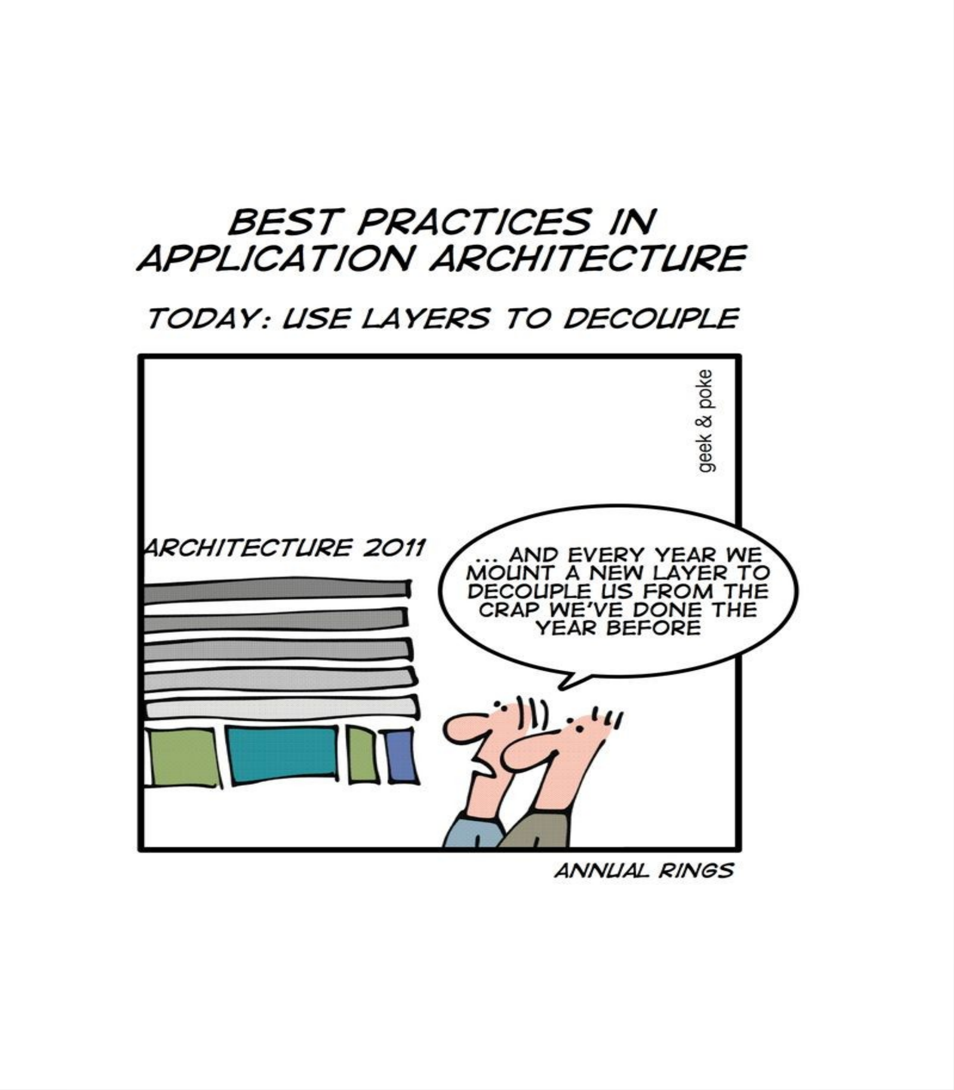
RTTI, Metaprogramming & Traits



Krishna Kumar

Run Time Type Information

- RTTI refers to the ability of the system to report on the dynamic type of an object and to provide information about that type at runtime (as opposed to at compile time).
- **Terminologies:**
  - Casting from a base class to a derived class is often called a **downcast** because of the convention of drawing inheritance trees growing from the root down.
  - Similarly, a cast from a derived class to a base is called an **upcast**.
  - A cast that goes from a base to a sibling class, like the cast from Bbwin dow to Ival_box , is called a **crosscast**

dynamic_cast<T*>(p)

- Works on pointers and references.
- Consider

```
class Interface {
public:
        virtual void GenericOp() = 0;// pure virtual function
};
class SpecificClass : public Interface {
public:
        virtual void GenericOp();
        virtual void SpecificOp();
};
```

- Let's say that we also have a pointer of type _Interface*_ ptr_interface;

dynamic_cast<T&>(p)

- Supposing that a situation emerges that we are forced to presume but have no guarantee that the pointer points to an object of type SpecificClass and we would like to call the member SpecificOp() of that class.
- SpecificClass* ptr_specific = dynamic_cast<SpecificClass*>(ptr_interface);

```
if( ptr_specific ){    // our suspicions are confirmed -- it really was a SpecificClass
    ptr_specific->SpecificOp();
}else{   // our suspicions were incorrect -- it is definitely not a SpecificClass.
    ptr_interface->GenericOp();
};
```

- dynamic_cast, the program converts the base class pointer to a derived class pointer and allows the derived class members to be called. If the pointer that you are trying to cast is not of the correct type, then dynamic_cast will return a **null pointer.**
- **SpecificClass& ref_specific = dynamic_cast<SpecificClass&>(ref_interface);**
- References throw bad::cast exception, instead of a nullptr.

typeid (object)

- The typeid operator, used to determine the class of an object at runtime.
- It returns a reference to a std::type_info object, which exists until the end of the program, that describes the "object".
- If the "object" is a dereferenced null pointer, then the operation will throw a std::bad_typeid exception.
- The use of typeid is often preferred over dynamic_cast<class_type> in situations where just the class information is needed, because typeid, applied on a type or non de-referenced value is a constant-time procedure
- It is generally only useful to use typeid on the dereference of a pointer or reference (i.e. typeid(*ptr) or typeid(ref)) to an object of polymorphic class type (a class with at least one virtual member function).
- This is because these are the only expressions that are associated with run-time type information. The type of any other expression is statically known at compile time.

Overview of Templates

- **Function templates**

```
template <class T>  //typename T
void swap (T& x, T& y)  {
    T temp = x;
    x = y;
    y = temp;
}
```

- **Function call**
- double a = 100., b = 1000.;
- swap(a, b);

Templates & Class

- // primary class template

```
template <typename T1,
typename T2>
class MyClass {
    ...
    T1 variable;
    T2 fn(T1& arg);
};
```

- // Specialisation

```
template<>
class Stack<std::string> {
 private:
     // elements
     std::deque<std::string> elems;

public:
    void push(std::string const&);
     ...
};
```

Member Type Aliases

- The template argument name, T , is only accessible to the template itself, so for other code to refer to the element type, we must provide an alias.
- typedefs don't support templatization, but alias declarations do.
- Alias templates avoid the "::type" suffix and, in templates, the "typename" prefix often required to refer to typedefs.
- C++14 offers alias templates for all the C++11 type traits transformation

```
template<typename T>
class Vector {
 public:
    using value_type = T;
    using iterator = Vector_iter<T>;
    // ...
};
```

Virtual Member Functions

- Member function templates cannot be declared virtual.
- This constraint is imposed because the usual implementation of the virtual function call mechanism uses a fixed-size table with one entry per virtual function.
- However, the number of instantiations of a member function template is not fixed until the entire program has been translated.
- In contrast, the ordinary members of class templates can be virtual because their number is fixed when a class is instantiated:

template <typename T>

class Dynamic {

public:

virtual ~Dynamic();

- /* OK: one destructor per instance of Dynamic<T> */

template <typename T2>

virtual void copy (T2 const&);

/* ERROR: unknown number of instances of copy() given an instance of Dynamic<T> */

- };

Requiring Member Functions

- You want to write a class template C that can be instantiated only on types that have a member function named Clone() that takes no parameters and returns a pointer to the same kind of object.
- // T must provide T* T::Clone() const

```
template<typename T>
class C
{
    // ...
};
```

- It's obvious that if C writes code that just tries to invoke T::Clone() without parameters, then such code will fail to compile if there isn't a T::Clone() that can be called without parameters.It's obvious that if C writes code that just tries to invoke T::Clone() without parameters, then such code will fail to compile if there isn't a T::Clone() that can be called without parameters.

Initial attempt (sort of requires clone)

- // T must provide /*...*/ T::Clone(/*...*/)
- template<typename T>
- class C {
- public:
  - void SomeFunc(const T* t) {

    // ...

    t->Clone();

    }
- };
- The first problem with this exmple is that In a template, only the member functions that are actually used will be instantiated. If SomeFunc() is never used, it will never be instantiated.
- Put it in the constructor? What about multiple constructors?

How about in the destructor?

- There is only one destructor, right?
- // T must provide /*...*/ T::Clone(/*...*/)
- template<typename T>
- class C {
- public:
  - ~C() {
    - const T t; // kind of wasteful, plus also requires
      // that T have a default constructor
    - t.Clone();
  - }
- };
- just trying to call T::Clone() without parameters would also succeed in calling a Clone() that has defaulted parameters and/or does not return a T* for eg., void T::Clone().

Alternative way

- // T must provide T* T::Clone() const
- template<typename T>
- class C {
  - bool ValidateRequirements() const {
    - T* (T::*test)() const = &T::Clone;
    - test; // suppress warnings about unused variables
    - return true;
  - }
- public:
  - ~C() {// in C's destructor (easier than putting it in every C constructor):
    - assert(ValidateRequirements());
  - }
- };

Constraints

- template<class Container>
- void draw_all(Container& c)
  { for_each(c.begin(),c.end(),mem_fun(&Shape::draw)); }
- If there is a type error, it will be in the resolution of the fairly complicated for_each() call. For example, if the element type of the container is an int, then we get some kind of obscure error related to the for_each() call (because we can't invoke Shape::draw() for an int).
- To catch such errors early, we can write:

```
template<class Container>
void draw_all(Container& c)  {
        Shape* p = c.front(); // accept only containers of Shape*s
        for_each(c.begin(),c.end(),mem_fun(&Shape::draw));
 }
```

- The initialization of the spurious variable "p" will trigger a comprehensible error message from most current compilers.

Constraints

- template<class Container>
- void draw_all(Container& c) {
  - typedef typename Container::value_type T;
  - Can_copy<T,Shape*>(); // accept containers of only Shape*s
  - for_each(c.begin(),c.end(),mem_fun(&Shape::draw));
- }
- This makes it clear that I'm making an assertion. The Can_copy template can be defined like this:
- template<class T1, class T2> struct Can_copy {
  - static void constraints(T1 a, T2 b) { T2 c = a; b = a; }
  - Can_copy() { void(*p)(T1,T2) = constraints; }
- };
- Can_copy checks (at compile time) that a T1 can be assigned to a T2. Can_copy<T,Shape*> checks that T is a Shape* or a pointer to a class publicly derived from Shape or a type with a user-defined conversion to Shape*.

Constraints Classes

- // HasClone requires that T must provide T* T::Clone() const

```
template<typename T>
class HasClone {
public:
    static void Constraints()   {
        T* (T::*test)() const = &T::Clone;
        test; // suppress warnings about unused variables
    }
    HasClone() { void (*p)() = Constraints; }
};
template<typename T>
class C : HasClone<T> { // … };
```

- The idea is simple: Every C constructor must invoke the HasClone<T> default constructor, which does nothing but test the constraint. If the constraint test fails, most compilers will emit a fairly readable error message. The HasClone<T> derivation amounts to an assertion about a characteristic of T in a way that's easy to diagnose.

The Problem

- In the iostream library, the interface to streambuf (as in stdio before it) depends on a value of EOF which is distinct from all character values. In traditional libraries, therefore, the type of EOF was int, and the function that retrieves characters returned an int:

```
class streambuf {
    int sgetc();  // return the next character, or EOF.
    int sgetn(char*, int N);  // get N characters.
};
```

- What happens when we parameterize streambuf on the character type? We need not only a type for the character, but for the type of the EOF value. Here's a start:

```
template <class charT, class intT>
class basic_streambuf {
    intT sgetc();
    int sgetn(charT*, int N);
};
```

- The extra template parameter clutters things up. Users of iostream don't care what the end-of-file mark is, or its type, and shouldn't need to care. Worse, what value should sgetc() return at end-of-file? Must this be another template parameter? The effort is getting out of hand.

References

- Bjarne Stroustrup's “C++ Programming Language 4ed”
- Scott Meyer's “Effective Modern C++”
- https://www.youtube.com/watch?v=cO1lb2MiDr8
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2027.html
- http://thbecker.net/articles/rvalue_references/section_01.html