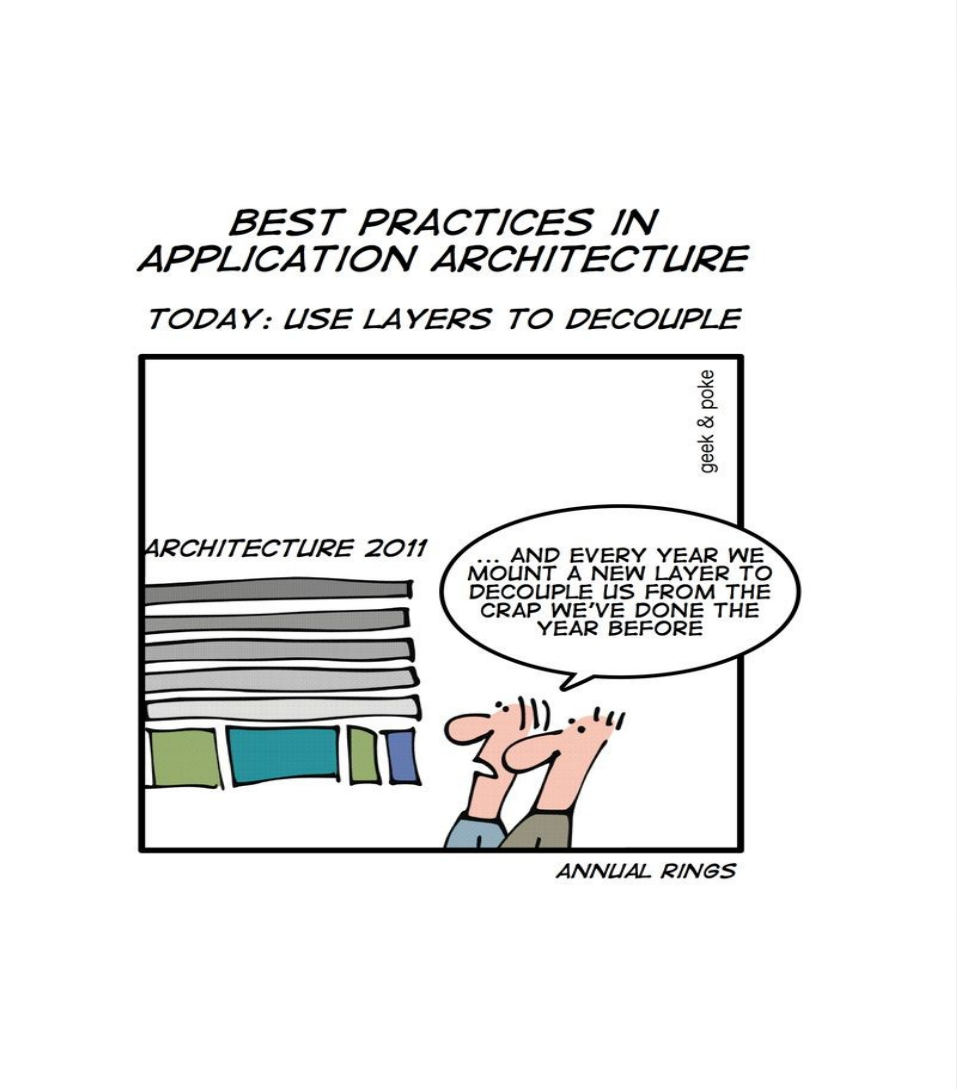
RTTI, Metaprogramming & Traits



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>  */
• };
```

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