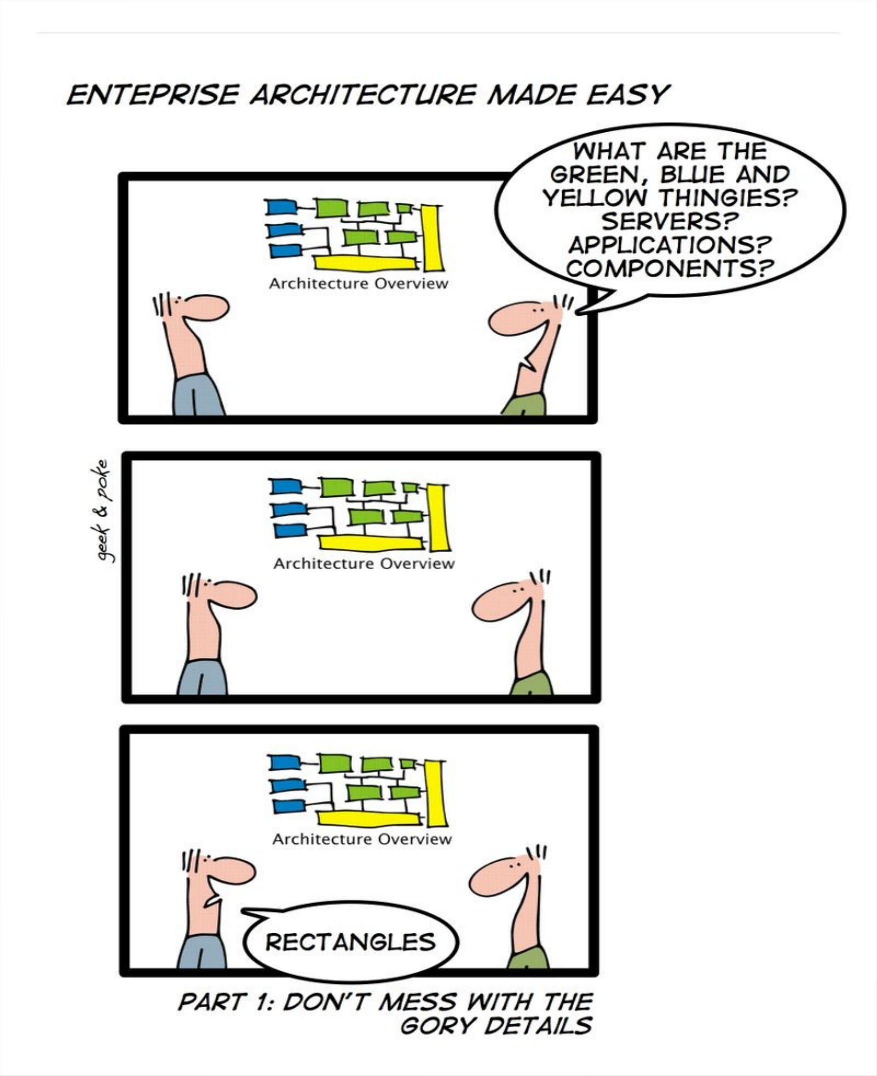
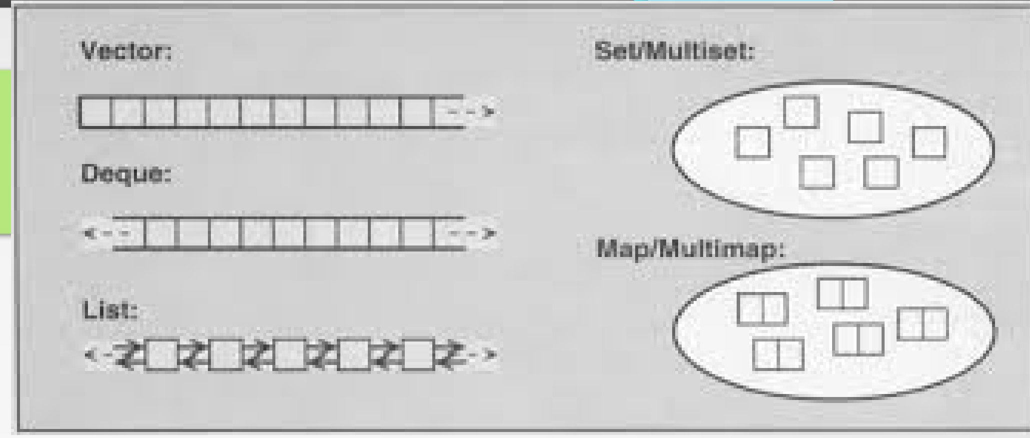
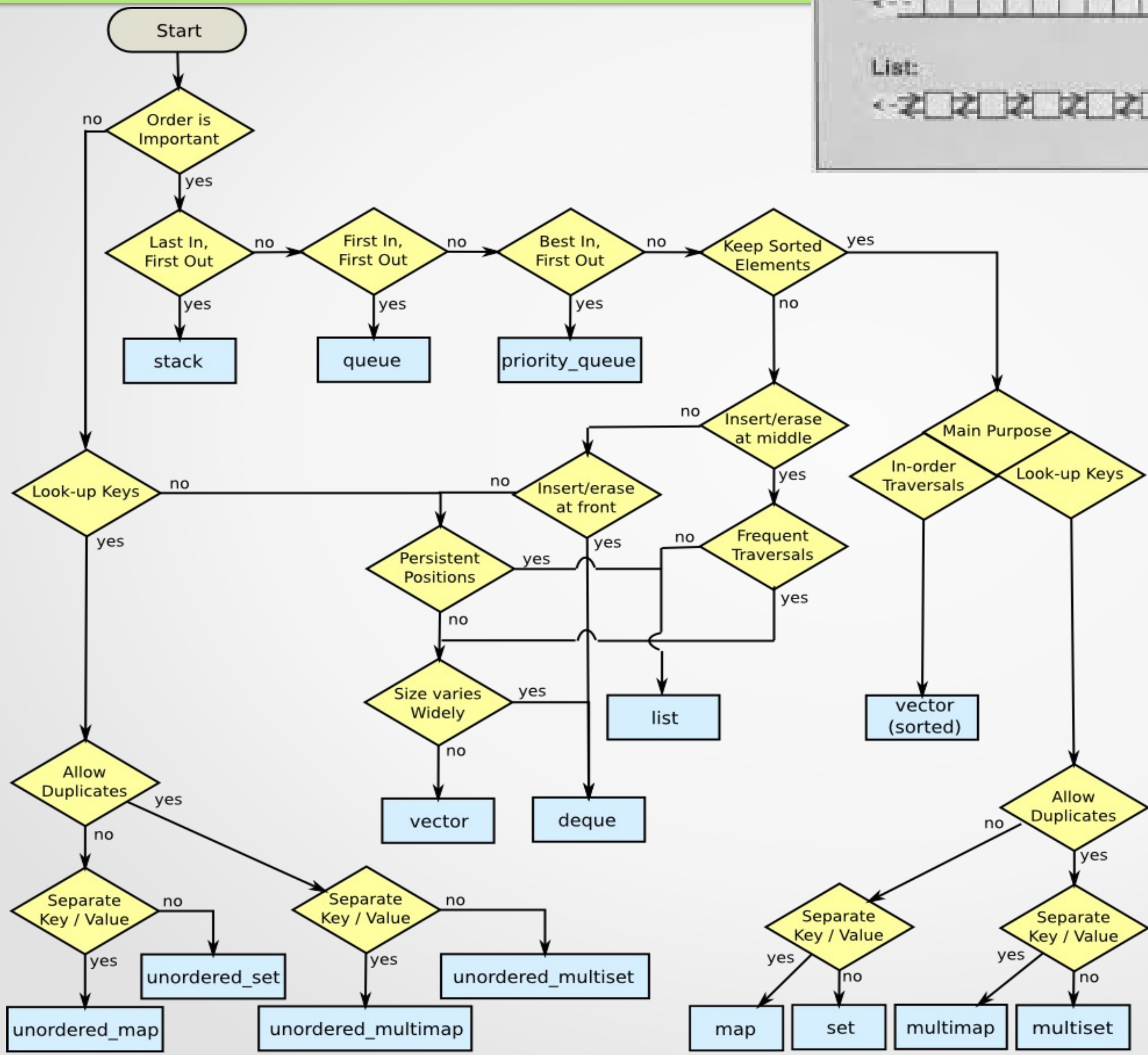
Tuple & Variadic Templates



Krishna Kumar

STL Containers





Tuple

- Tuples are objects that pack elements of -possibly- different types together in a single object, just like pair objects do for pairs of elements, but generalized for any number of elements.
- One way to imagine using a tuple is to hold a row of data in a database. The row might contain the attributes of a person, such as the person's name, account number, address, and so on. All the elements might have different types, but they all belong together as one row.
- Conceptually, they are similar to plain old data structures (C-like structs) **but instead of having named data members, its elements are accessed by their order in the tuple.**
- The selection of particular elements within a tuple is done at the template-instantiation level, and thus, it must be specified at compile-time, with helper functions such as get and tie.

Manipulating a tuple

Tuple Function	**Explanation**
make_tuple	Pack values in a tuple
forward_as_tuple	Pack Rvalue reference in the tuple
std::get<i>(mytuple)	Get element “i” in the tuple - mytuple
std::tie	Unpack values from a tuple
tuple_size<decltype(mytyple)>::value	Size of tuple
tuple_element<i, decltype(mytuple)>::type	Get element type of element “i” in mytuple
tuple_cat (mytuple, std::tuple<int,char>(mypair))	Concatenate tuples

Tuple implementation

- a class or function template to accept some arbitrary number (possibly zero) of “extra” template arguments.
- This behaviour can be simulated for class templates in C++ via a long list of defaulted template parameters.
- Tuple implementation:

```
struct unused;
template <typename  T1 = unused,typename T2 = unused,
            typename T3 = unused, typename T4 = unused,
            /∗up to∗ / typename TN = unused>
class tuple;
```

- This tupletype can be used with anywhere from zero to N template arguments. Assuming that N is “large enough”, we could write:
  - **typedef tuple<char, short, int, long, long long> integraltypes;**

Tuple implementation (cont...)

- Of course, this tuple instantiation actually contains N − 5 unused parameters at the end, but presumably the implementation of tuple will somehow ignore these extra parameters. In practice, this means changing the representation of the argument list or providing a host of partial specializations:

```
template <>
class tuple<> { / ∗ handle zero-argument version. ∗ / } ;

Template < typename T1 >
Class tuple <T1>{/ ∗ handle one-argument version. ∗ /} ;

Template < typename T1, typename T2>
Class tuple <T1, T2> {/∗ handle two-argument version. ∗ /};
```

- Unfortunately, it leads to a huge amount of code repetition, very long type names in error message.
- It also has a fixed upper limit on the number of arguments that can be provided.

Variadic templates

- Variadic templates let us explicitly state that tuple accepts one or more template type parameters
  - **template <typename... Elements> class tuple;**
- The ellipsis to the left of the Elements identifier indicates that Elements is a template type parameter pack.
- A parameter pack is a new kind of entity introduced by variadic templates.
- Unlike a parameter, which canonly bind to a single argument, multiple arguments can be “packed” into a single parameter pack.
- Eg., declare an array class that supports an arbitrary number of dimensions:

```
template <typename T, unsigned PrimaryDim, unsigned... Dim>
class array { /* implementation */ };
array<double, 3, 3> rotation_matrix; // 3 x 3 matrix
```

Packing and Unpacking Parameter Packs

- To do anything with parameter packs, one must unpack (or expand) all of the arguments in the parameter pack into separate arguments, to be forwarded on to another template.
  - **template <typename... Args> struct count;**
- Basis case, zero arguments:

```
template<>
struct count<> {
    static const int value = 0;
};
```

Packing & Unpacking (recursive)

- The idea is to peel off the first template type argument provided to count, then pack the rest of the arguments into a template parameter pack to be counted in the final sum:

```
template <typename T, typename... Args>
struct count<T, Args...> {
    static const int value = 1 + count<Args...>::value;
}
```

- Whenever the ellipsis occurs to the right of a template argument, it is acting as a *meta-operator* that unpacks the parameter pack into separate arguments.
- In the expression count <Args...>::value, this means that all of the arguments that were packed into Args will become separate arguments to the count template.
- The ellipsis also unpacks a parameter pack in the partial specialization count<T, Args...>, where template argument deduction packs “extra” parameter passed to count into Args.
- For instance, the instantiation of count<char, short, int>would select this partial specialization, binding T to char and Args to a list containing short and int.

Tuple implementation – Variadic template

```
template<typename... Elements>
class tuple; template<typename Head, typename... Tail >
class tuple < Head, Tail... > : private tuple < Tail... > {
    Head head;
    public:
        /∗ implementation   ∗/
    };
template<>
class tuple<> {
    /∗zero-tuple implementation∗/
};
```

References

- Bjarne Stroustrup's “C++ Programming Language 4ed”
- Scott Meyer's “Effective Modern C++”
- Herb Sutter's Exceptional C++ and More Exceptional C++
- C++ Templates: the Complete Guide
-
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2080.pdf