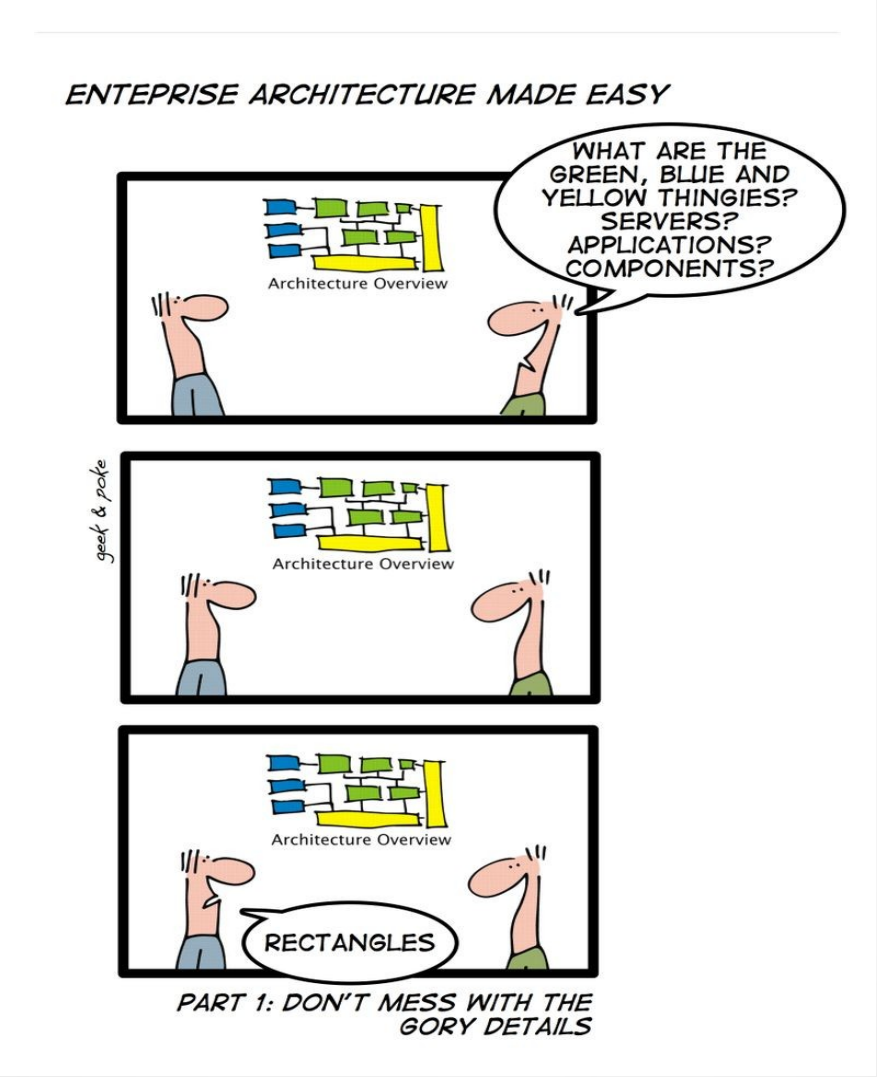
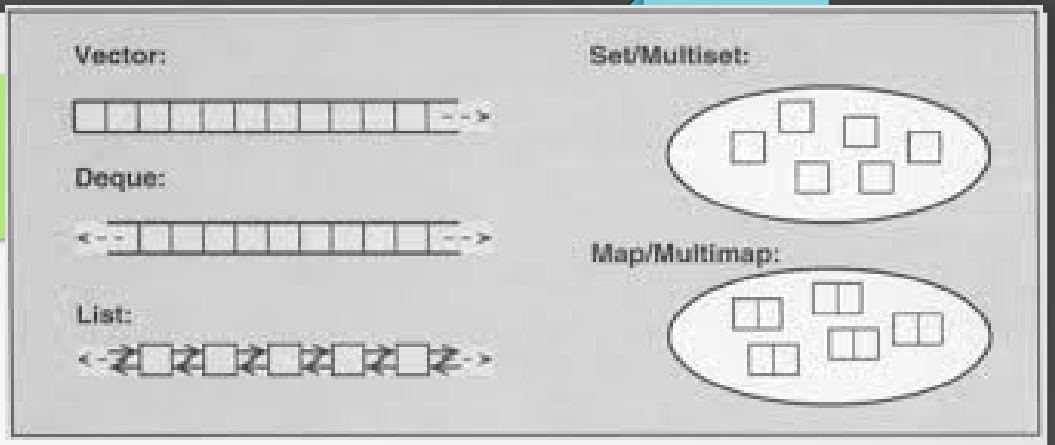
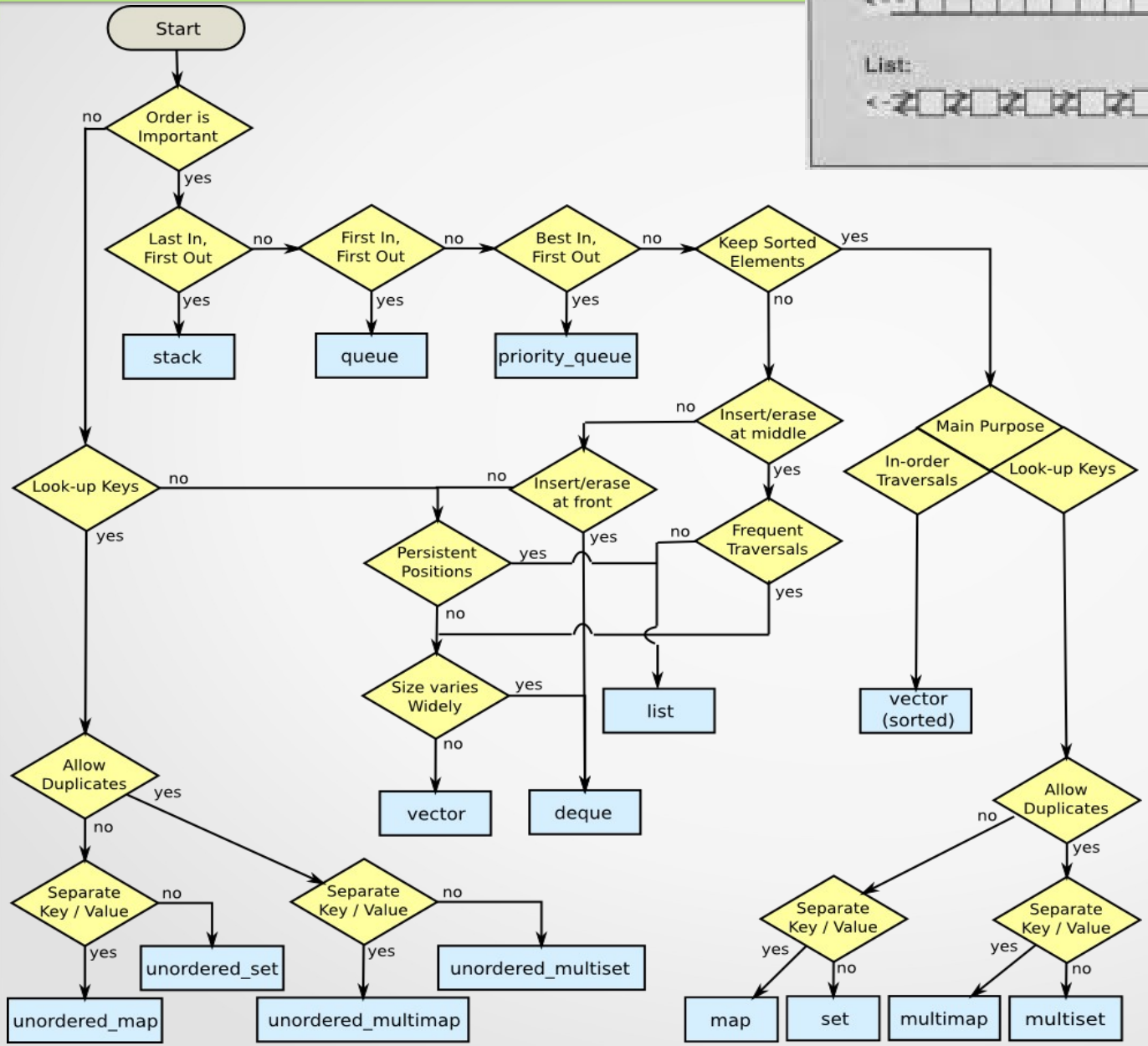
Tuple & Variadic Templates



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*/
};
```

C++11 constructor inheritance through variadic templates

```
template <typename T>
class Base {
public:
    Base(int a) { /* ... */ }
    Base(double a) { /* ... */ }
    Base(char a) { /* ... */ }
    Base(int a, double b, char c) { /* ... */ }
    // etc...
Protected:
    T member_;
};
```

- // Derived class
- class Derived1 : public Base<int> {
- public:
  - template <typename... Args>
  - Derived1(Args... args) : Base(args...) { }
- // other Derived1 ctors
- };

Variadic function

- Variadic template parameters to functions have almost the same syntax as for classes, the only difference being that variadic function arguments can also be passed. For example, in the following code, the function output prints all its arguments separated by spaces.

```
template <typename T>
void output(T&& t) {   std::cout << t; }


template <typename First, typename... Args>
void output(First&& f, Args&&... args) {
    std::cout << f << " ";
    output(std::forward<Args>(args)...);
}
output(1, 2, 45); // Will print 1 2 45
```

- typename... Args is called a template parameter pack, and Args.. args is called a function parameter pack

Variadic templates for forwarding

- Constructor is a commonly used construct that is inherently "variadic" when viewed from a template parameter point of view.
- Given a generic type T, to invoke the constructor of T, we may need to pass in an arbitrary number of arguments.
- Unlike function types that specify their arguments at compile time, given just a generic type T we don't know which constructor(s) it has and how many arguments the constructor accepts.
  - `std::unique_ptr<FooType> f = std::make_unique<FooType>(1, "str", 2.13);`
  - `template<typename T, typename... Args>`
  - `unique_ptr<T> make_unique(Args&&... args) {`
  - `    return unique_ptr<T>(new T(std::forward<Args>(args)...));`
  - `}`

Template metaprogramming

- Metaprogramming consists of "programming a program." In other words, we lay out code that the programming system executes to generate new code that implements the functionality we really want.
- Usually the term metaprogramming implies a reflexive attribute: The metaprogramming component is part of the program for which it generates a bit of code/program.
- Let us compute: 3N = 3 * 3N -1
  - template<int N>
  - class Pow3 {

```
    public:
        enum { result=3*Pow3<N-1>::result };
};
```

Metaprograms to Unroll Loops

- One of the first practical applications of metaprogramming was the unrolling of loops for numeric computations.
- Numeric applications often have to process n-dimensional arrays or mathematical vectors.
- One typical operation is the computation of the so-called dot product.
- The dot product of two mathematical vectors a and b is the sum of all products of corresponding elements in both vectors.
- For example, if each vectors has three elements, the result is

  a[0]*b[0] + a[1]*b[1] + a[2]*b[2]

- A mathematical library typically provides a function to compute such a dot product.

Optimising unroll loops

- Result: dot_product(3,a,b) = 38
- This is correct, but it takes too long for serious high-performance applications. Even declaring the function inline is often not sufficient to attain optimal performance.
- The problem is that compilers usually optimize loops for many iterations, which is counterproductive in this case. Simply expanding the loop to would be a lot better:
- a[0]*b[0] + a[1]*b[1] + a[2]*b[2]
- Of course, this performance doesn't matter if we compute only some dot products from time to time. But, if we use this library component to perform millions of dot product computations, the differences become significant.
- We could write the computation directly instead of calling dot_product(), or we could provide special functions for dot product computations with only a few dimensions, but this is tedious.
- Template metaprogramming solves this issue for us: We "program" to unroll the loops.

Optimising unroll loops

- Instead of dot_product(3,a,b) → dot_product<3>(a,b)
- This expression instantiates a convenience function template that translates the call into
- DotProduct<3,int>::result(a,b) starts the metaprogram.
- The result is the product of the first elements of a and b plus the result of the dot product of the remaining dimensions of the vectors starting with their next elements.
- the instantiation process computes the following:

  DotProduct<3,int>::result(a,b)

  = *a * *b + DotProduct<2,int>::result(a+1,b+1)

  = *a * *b + *(a+1) * *(b+1) + DotProduct<1,int>::result(a+2,b+2)

  = *a * *b + *(a+1) * *(b+1) + *(a+2) * *(b+2)
- Widely used in BLITZ, MKL, etc.

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://eli.thegreenplace.net/2014/variadic-templates-in-c/#id1
- http://lbrandy.com/blog/2013/03/variadic_templates/
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2080.pdf