Generic Programming

Dietmar Kühl
dietmar.kuehl@gmail.com
Bloomberg L.P.

Copyright Notice

Road Map

- algorithms on tuples
- variadic templates
- implementing tuples
- a bit on r-values

Basic Idea

- implement algorithms such that they are
  - independent of any data structure
  - as efficient as any specific implementation
  - type-safe, easy to use, etc.

Typical Uses of GP

- abstraction from homogenous data structures
  - STL uses sequences of the same type
  - BGL uses nodes, edges , etc. which don't vary with-in an instantiation
  - MTL operates on same type matrices

Unusual Abstractions

- process elements of a structure
  - read/write or decode/encode elements
  - display/edit elements
  - type specific transformation of elements
- implement functional transformation

Comparing Structures

- lexicographic compare is actually non-trivial
- std::tuple implements operator<()
- use this to compare structures, e.g.:

```
struct S { int i; double d; std::string s; };
bool operator< (S const& a, S const& b) {
  return tie(a.i, a.d, a.s) < tie(b.i, b.d, b.s); }
```

Implement Tuple's op<()

!

```
template <typename ...T>
bool operator< (tuple<T...> const& t1,
                tuple<T...> const& t2) {
    return lexcmp(t_begin(t1), t_end(t1),
                  t_begin(t2), t_end(t2),
                  g_less());
}
```

Generic Less

```
struct g_less
{
    typedef bool result_type;
    template <typename S, typename T>
    bool operator()(S const& a,
                    T const& b) const
    { return a < b; }
};
```

Implement Tuple's op<() 2

```
template <typename P, typename C>
bool lexcmp(P, P, P, C) { return false; }
template <typename P, typename E,
          typename C>
bool lexcmp(P p1, E e, P p2, C cmp) {
  return cmp(*p1, *p2)
      || (!cmp(*p2, *p1)
          && lexcmp(++p1, e, ++p2, cmp));
}
```

Tuple as Sequence

```
template <typename T>
tuple_iter<0, T const> t_begin(T const& t)
{ return tuple_iter<0, T const>(t); }

template <typename T>
tuple_iter<tuple_size<T>::value, T const>
t_end(T const& t)
{ return tuple_iter<tuple_size<T>::value>,
                    T const>(t); }
```

Tuple Iterator

```
template <int I, typename T>
struct tuple_iter<I, T const, false> {
    tuple_iter(T const& t): t_(t) {}
    tuple_iter<I+1, T const>
    operator++() { return t_; }
    TN tuple_element<I, T>::type const&
    operator*() const { return get<I>(t_); }
    T const& t_;
};
```

Tuple End Iterator

```
template <int I, typename T,
    bool = I == tuple_size<
      typename remove_const<T>::type
    >::value>
struct tuple_iter {
    tuple_iter(T&) {}
 };
```

Another Example: copy ()

```
template <typename E, typename T>
void copy(E, E, T) {}

template <typename F, typename E,
   typename T>
void copy(F it, E end, T to) {
    *to = *it;
    copy(++it, end, ++to);
}
```

Algorithms on Tuples

- basic idea: use tuple "iterators" for access:
  - operators ++ and -- return different type
    - supports subranges
- recurse to support different element types
- similar structure for various algorithms

Variadics and Tuples

- dealing with variadic templates may be tricky
- algorithms can be used with tuples
- variadic arguments can be turned into tuples:

```
template <typename ...T >
void f(T... args) {
    std::tuple<T...> t(args...);
```

Variadic Introduction

- template may use variadic arguments:
  - typename...T:T... is a list of types
    T... args: args... is a list of parameters
  - int... I: list of integer parameters
- two operations on “parameter packs”:
  - expansion using “<something with pack>...”

Forwarding Packs

```
template <typename F> struct wrap {
    wrap(F f): f_(f) {}
    template <typename...T>
    typename std::result_of<F(T...)>::type
    operator()(T&&... a) {
        format(cout, a...);
        return f_(std::forward<T>(a)...); }
    F f_;
};
```

Pack Elements: Head/ Tail

```
template <typename T>
void format(std::ostream& o, T const& a) {
    o << a;
}
template <typename T, typename... S>
void format(std::ostream& o, T const& a,
            S const&... b) {
    format(o << a << " ", b...);
}
```

Pack Elements: via tuple

```
void format(std::ostream&) {}
template <typename ...T>
void format(std::ostream& o, T const&... a){
    std::tuple<T const&...> t(a...);
    copy(t_begin(t), --t_end(t),
         ostream_iterator(o, " "));
    o << *--t_end(t);
}
```

Threaded Function

- execute function call in a different thread
  - call puts arguments into a queue
    - another thread is waiting for the queue
    - arguments are dequeued to call function
- useful in pipeline style processing

Threaded Function: API

```
template <typename> class TF;
template <typename ...T> struct TF {
  template <typename F> TF(F f);
  void operator()(T... t);
private:
  tuple<T...> next(); void run();
  mutex mutex_; condition_variable cond_;
  deque<tuple<T...> > q_;
  function<void(T...)> f_;
};
```

Threaded Function: Call

```
template <typename ...T>
void TF<void(T...)>::operator()(T&&... t) {
   {
      unique_lock<std::mutex> l(mutex_);
      q_.push_back(std::tuple<T...>(t...));
   }
   cond_.notify_one();
}
```

Threaded Function: Next

```
template <typename ...T>
std::tuple<T...> TF<T...>::next() {
    unique_lock<mutex> l(mutex_);
    while (q_.empty())
        cond_.wait(l);
    tuple<T...> t(q_.front());
    q_.pop_front();
    return t;
}
```

Threaded Function: Run

```
template <typename ...T>
void TF<T...>::run() {
    while (true) {
        tuple<T...> args = next();
        // f_(???);
        call(f_, args);
    }
}
```

Function Call vs. Tuple

```
template <class F, class... T, int... I>
void call(F&& f, tuple<T...>&& t, IL<I...>*) {
    f(get<I>(t)...);
}
template <class F, class... T>
void call(F&& f, tuple<T...>&& t) {
    call(forward<F>(f),
      forward<tuple<T...> >(t), indices<T...>());
}
```

Making Index Lists I

```
template <int...> struct IL {};

template <class, int> struct AI;
template <int... I, int S>
struct AI<IL<I...>, S> {
    typedef IL<I..., S> type;
};
```

Making Index List II

```
template <int S> struct MI:
  AI<typename MI<S-1>::type, S-1> {};
template<>
struct MI<0> { typedef IL<> type; };

template <typename...T>
typename MI<sizeof...(T)>::type* indices() {
    return 0; }
```

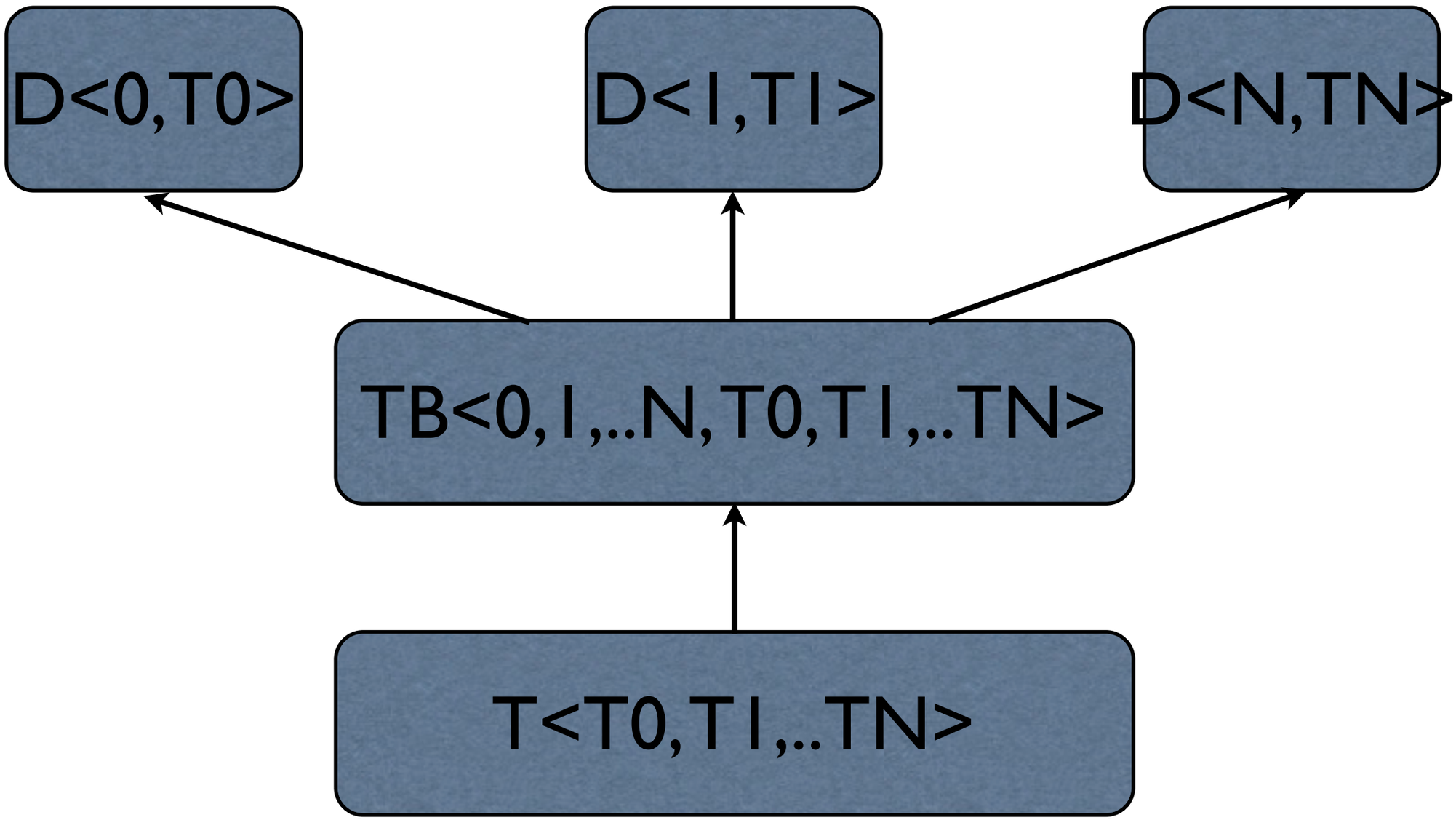
Index List Summary

- typename MI<sizeof...(T)>::type provides IL<0, ..., sizeof...(T) - 1>
- indices can be used to join up arguments
- call() creates a new list of tuple elements using the list of indices
- other uses are possible as well

Variadics vs. Compilers

- available e.g. in gcc-4.* using -std=c++0x
- work-arounds where unavailable:
  - class template: use defaulted parameters
  - functions: use overloading or specialization
  - the work-arounds are rather painful...!

Implementing tuple



tuple Base

```
template <class, class ...> struct TB;
template <int... I, class ...T>
struct TB<IL<I...>,T...>:TD<I,T>... {
    template <typename ...S>
        TB(S&&... s):
            TD<I,T>(forward<S>(s))...
    {}
};
```

tuple_element I

```
template <class ...> struct TL {};
template <int, class> struct TI;
template <int I, class H, class ...T>
struct TI<I,TL<H,T...> >
{ typedef typename
      TI<I-1,TL<T...> >::type type; };
template <class H, class ...T>
struct TI<0,TL<H,T...> >
{ typedef H type; };
```

tuple_element 2

```
template <int I, class> struct TE;
template <int I, class ...T>
struct TE<I, TU<T...> >:TI<I, TL<T...> > {};
```

```
template <int I, class ...T>
typename TE<I, TU<T...> >::type&
get(TU<T...>& v) {
  return static_cast<D<I,
   typename TE<I, TU<T...> >::type>&>(v).v_;
}
```

R-Value Deduction

- call details determine r-value types
- template <typename T> void f(T&&):
  - int i(1); f(i) ⇒ int&
  - int const ic(2); f(ic) ⇒ int const&
  - f(int(3)) ⇒ int

Perfect Forwarding

- template <typename T> void f(T&& t) { g (t); }
- reference [to const] forwarded OK
- temporary parameters turned into references
- std::forward<T>(t) to prevent this: template <typename T, typename U>

Summary

- tuples for heterogenous sequences
- variadic templates for variable number of
  - arguments
  - elements
- r-values for forwarding values