C++0xの可変引数テンプレートで型リスト処理

高橋晶(アキラ)

ブログ :「Faith and Brave – C++で遊ぼう」
http://d.hatena.ne.jp/faith_and_brave/

可変引数テンプレート

```
template <class... Args>
class tuple;
```

のように書くことで
テンプレートパラメータを可変個
受け取ることができる

```
tuple<int>                t1;
tuple<int, long>          t2;
tuple<int, long, char> t3;
```

可変個のテンプレートパラメータは型のリストと見なせる

型リストを処理するメタ関数を作ってみた

- head, tail
- length
- at
- concat
- reverse
- replicate
- take, drop
- map, filter
- tale_while, drop_while
- all, any

head : 型リストの先頭
tail : 型リストの後部

```
template <class Head, class... Tail>
struct head {
  typedef Head type;
};

template <class Head, class... Tail>
struct tail {
  typedef tuple<Tail...> type;
};

head<int, long, char>::type
→ int

tail<int, long, char>::type
→ tuple<long, char>
```

head, tailのtuple版

tupleの部分特殊化も用意しておく

```
template <class Head, class... Tail>
struct head<tuple<Head, Tail...>> {
  typedef Head type;
};

// 型リストから後部の型リストを取得
template <class Head, class... Tail>
struct tail<tuple<Head, Tail...>> {
  typedef tuple<Tail...> type;
};

head<tuple<int, long, char>>::type
→ int

tail<tuple<int, long, char>>::type
→ tuple<long, char>
```

length : 型リストの長さ

```
template <class... Args>
struct length {
  static const int value = sizeof...(Args);
};

template <class... Args>
struct length<tuple<Args...>> {
  static const int value = sizeof...(Args);
};

length<int, long, char>::value
→ 3

length<tuple<int, long, char>>::value
→ 3
```

at : 型リストのI番目

ここからはめんどくさいからtuple版のみ

```
template <int I, typename Arg>
struct at;

template <int I, typename Head, typename... Tail>
struct at<I, tuple<Head, Tail...>> {
  typedef typename at<I - 1, tuple<Tail...>>::type type;
};

template <class Head, typename... Tail>
struct at<0, tuple<Head, Tail...>> {
  typedef Head type;
};

at<1, tuple<int, long, char>>::type
→ long
```

concat : 型リストの連結

```
template <class Seq1, class Seq2>
struct concat;

template <class... Seq1, class... Seq2>
struct concat<tuple<Seq1...>, tuple<Seq2...>> {
  typedef tuple<Seq1..., Seq2...> type;
};

concat<tuple<int,  double>,
       tuple<long, char>>::type
```

→ `tuple<int, double, long, char>`

reverse：型リストを逆順にする

型の長さ分だけ再帰して、先頭を後ろに追加していく

```
template <int N, class Seq>
struct reverse_impl;

template <int N, class Head, class... Tail>
struct reverse_impl<N, tuple<Head, Tail...>> {
  typedef
    typename concat<typename reverse_impl<N-1, tuple<Tail...>>::type,
                    tuple<Head>
             >::type
  type;
};

template <class... Seq>
struct reverse_impl<0, tuple<Seq...>> {
  typedef tuple<Seq...> type;
};
```

reverse : 型リストを逆順にする

```
template <class Seq>
struct reverse;

template <class... Seq>
struct reverse<tuple<Seq...>> {
  typedef typename reverse_impl<sizeof...(Seq), tuple<Seq...>>::type type;
};

reverse<tuple<int, double, long>>::type
→ tuple<long, double, int>
```

replicate : TをN個含んだ型リストを作成

N個分だけ再帰してconcatで型を追加していく

```
template <int N, class T>
struct replicate {
  typedef
    typename concat<typename replicate<N-1, T>::type,
                    tuple<T>
             >::type
  type;
};

template <class T>
struct replicate<0, T> {
  typedef tuple<> type;
};

replicate<3, int>::type
→ tuple<int, int, int>
```

take：先頭N個の型を取得

```
template <int N, class Seq>
struct take;

template <int N, class Head, class... Tail>
struct take<N, tuple<Head, Tail...>> {
  typedef
    typename concat<tuple<Head>,
                    typename take<N-1, tuple<Tail...>>::type
              >::type
  type;
};

template <class Head, class... Tail>
struct take<1, tuple<Head, Tail...>> {
  typedef tuple<Head> type;
};

template <class Head, class... Tail>
struct take<0, tuple<Head, Tail...>> {
  typedef tuple<> type;
};

take<3, tuple<int, double, long, char, void*>>::type
→ tuple<int, double, long>
```

drop：先頭N個を除外した型リストを取得

```
template <int N, class Seq>
struct drop;

template <int N, class Head, class... Tail>
struct drop<N, tuple<Head, Tail...>> {
  typedef
    typename drop<N-1, tuple<Tail...>>::type
  type;
};

template <class Head, class... Tail>
struct drop<0, tuple<Head, Tail...>> {
  typedef tuple<Head, Tail...> type;
};

drop<3, tuple<int, double, long, char, void*>>::type
→ tuple<char, void*>
```

map : 型リストの全ての型にメタ関数を適用

mapはメタ関数をパラメータで受け取る高階メタ関数
メタ関数は、テンプレートテンプレートパラメータで受け取る

```
template <template <class T> class F, class... Seq>
struct map;

template <template <class T> class F, class... Seq>
struct map<F, tuple<Seq...>> {
  typedef tuple<typename F<Seq>::type...> type;
};

template <class T>
struct add_pointer {
  typedef T* type;
};

map<add_pointer, tuple<int, double, long>>::type
→ tuple<int*, double*, long*>
```

filter : 条件抽出(1)

まず、条件を満たす場合のみ型リストに型を追加するメタ関数を用意する

```
template <bool B, class T, class Seq>
struct add_if_c;

template <class T, class... Seq>
struct add_if_c<true, T, tuple<Seq...>> {
    typedef tuple<Seq..., T> type;
};

template <class T, class... Seq>
struct add_if_c<false, T, tuple<Seq...>> {
    typedef tuple<Seq...> type;
};


template <template <class> class P, class T, class... Seq>
struct add_if;

template <template <class> class P, class T, class... Seq>
struct add_if<P, T, tuple<Seq...>> :
    public add_if_c<P<T>::value, T, tuple<Seq...>> {};
```

filter : 条件抽出(2)

add_ifを使って、述語を満たす型のみの型リストを作成

```
template <template <class T> class P, class Seq1, class Seq2>
struct filter_impl;

template <template <class T> class P, class... Seq, class Head, class... Tail>
struct filter_impl<P, tuple<Seq...>, tuple<Head, Tail...>> {
    typedef
        typename filter_impl<P,
                             typename add_if<P,
                                             Head,
                                             tuple<Seq...>>::type,
                             tuple<Tail...>>::type
    type;
};

template <template <class T> class P, class... Seq>
struct filter_impl<P, tuple<Seq...>, tuple<>> {
    typedef tuple<Seq...> type;
};


template <template <class T> class P, class... Seq>
struct filter;

template <template <class T> class P, class... Seq>
struct filter<P, tuple<Seq...>> {
    typedef typename filter_impl<P, tuple<>, tuple<Seq...>>::type type;
};

filter<is_integral, tuple<int, double, long>>::type
→ tuple<int, long>
```

take_while : 条件を満たす先頭部分を取り出す

```
template <template <class T> class P, class Seq1, class Seq2>
struct take_while_impl;

template <template <class T> class P, class... Seq, class Head, class... Tail>
struct take_while_impl<P, tuple<Seq...>, tuple<Head, Tail...>> {
  typedef typename
    if_c<P<Head>::value,
         typename take_while_impl<P,
                                  typename concat<tuple<Seq...>, tuple<Head>>::type,
                                  tuple<Tail...>
                >::type,
         tuple<Seq...>
    >::type
  type;
};

template <template <class T> class P, class... Seq>
struct take_while_impl<P, tuple<Seq...>, tuple<>> {
  typedef tuple<Seq...> type;
};


template <template <class T> class P, class Seq>
struct take_while;

template <template <class T> class P, class... Seq>
struct take_while<P, tuple<Seq...>> {
  typedef typename take_while_impl<P, tuple<>, tuple<Seq...>>::type type;
};


take_while<is_integral, tuple<int, long, char, double>>::type
→ tuple<int, long, char>
```

drop_while : 条件を満たす先頭部分を取り出す

```
template <template <class T> class P, class Seq>
struct drop_while;

template <template <class T> class P, class Head, class... Tail>
struct drop_while<P, tuple<Head, Tail...>> {
  typedef
    typename if_c<P<Head>::value, // trueだったら
                  typename drop_while<P, tuple<Tail...>>::type, // 無視する
                  tuple<Head, Tail...>
             >::type
  type;
};

template <template <class T> class P>
struct drop_while<P, tuple<>> {
  typedef tuple<> type;
};

drop_while<is_integral, tuple<int, long, char, double>>::type
→ tuple<double>
```

all : 型リストの要素全てが述語を満たすか

型リストの全ての型が述語を満たせばtrue
それ以外はfalseを返す

```
template <template <class T> class P, class Seq>
struct all;

template <template <class T> class P, class Head, class... Tail>
struct all<P, tuple<Head, Tail...>> {
  static const bool value = !P<Head>::value ?
                            false :
                            all<P, tuple<Tail...>>::value;
};

template <template <class T> class P>
struct all<P, tuple<>> {
  static const bool value = true;
};
```

```
all<is_integral, tuple<int, long, char>>::value
→ true

all<is_integral, tuple<int, double, char>>::value
→ false
```

any :型リストに述語を満たす型があるか

型リストのいずれかの型が述語を満たせばtrue
それ以外はfalseを返す

```
template <template <class T> class P, class Seq>
struct any;

template <template <class T> class P, class Head, class... Tail>
struct any<P, tuple<Head, Tail...>> {
  static const bool value = P<Head>::value ?
                            true :
                            any<P, tuple<Tail...>>::value;
};

template <template <class T> class P>
struct any<P, tuple<>> {
  static const bool value = false;
};

any<is_integral, tuple<int, long, char>>::value
→ true

any<is_integral, tuple<float, int, void*>>::value
→ true

any<is_integral, tuple<float, double, void*>>::value
→ false
```