Iterators Must Go

Andrei Alexandrescu

This Talk

- The STL
- Iterators
- Range-based design
- Conclusions

STLとは何か？

STLとは何か？

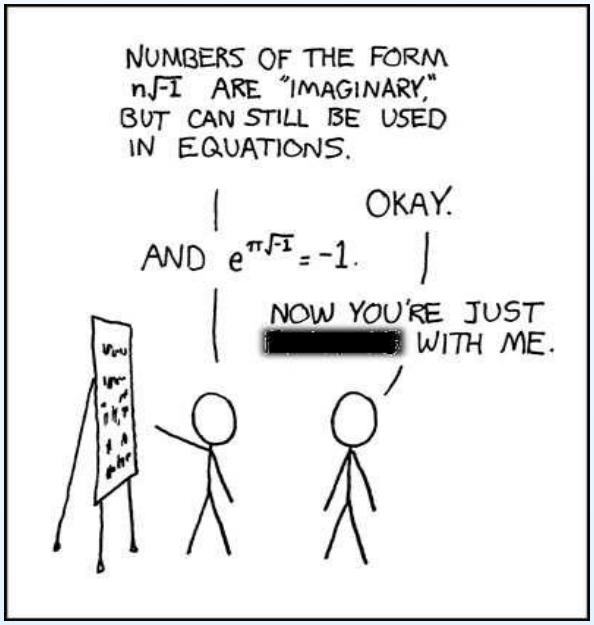
- アルゴリズムとデータ構造の(良い|悪い|醜い)ライブラリ
- `iterators = gcd(containers, algorithms);`
- **S**crumptious **T**emplate **L**ore(すばらしいテンプレートの知恵)
- **S**nout to **T**ail **L**ength(先頭から最後尾までの長さ)

STLとは何かというと

- STLでは答えよりも質問の方が重要
- “基本的なコンテナとアルゴリズムの中で最も一般的な実装はどのようなものでしょうか？”
- ほかのものは全てその余波
- STLで最も重要なこと：
  STLはひとつの答えではあるが、答えではない

STLは非直観的

- 相対性理論が非直観的なのと同じ方向
- 複素数が非直観的なのと同じ方向



n√-1形式の値は”虚数”だが
方程式で使用することができる。

非直観的

- “私は最も一般的なアルゴリズムを設計したい。”
- “もちろんできる。あなたが確実に必要とするものは、イテレータと呼ばれるものである。正確に言うとそれらのうちの5つ。”
- 証拠：どんな言語も”たまたま”STLをサポートしなかった。
  - 無慈悲な”機能戦争”にもかかわらず
  - C++とDが唯一
  - どちらもSTLのサポートを積極的に目指した
- 結果：STLは、C++とDの外から理解するのが非常に困難になった。

Fundamental vs Incidental in STL
(基礎 vs 偶発的なSTL)

- F：アルゴリズムは可能な限り狭いインタフェースで定義される
- F：アルゴリズムに必要とされる広いイテレータカテゴリ
- I：iterator primitiveの選択
- I：iterator primitiveの構文

STL:良い点

- 正しい質問ができる
- 一般的
- 効率的
- 無理なく拡張可能
- 組み込み型に統合された

STL:悪い点

- ラムダ関数のサポートが貧弱
  - STLの問題ではない
  - 高い機会費用
- いくつかのコンテナはサポートできない
  - たとえば、sentinel-terminatedコンテナ
  - たとえば、分散ストレージを持つコンテナ
- いくつかの反復法はサポートできない

STL:醜い点

- for_each等の試みは助けにならなかった
- ストリームによる統合が希薄
- 1単語：allocator
- イテレータの最低なところ
  - 冗長
  - 安全ではない
  - 貧弱なインタフェース

イテレータの取り決めは何か？

Iterators Rock

- コンテナとアルゴリズム間の相互作用をまとめ上げる
- “強度低下:”m・nの代わりにm＋nを実装
- 拡張可能:ここで、STLが日の目を見て以来、イテレータに動揺が走った

危険信号 #1

- 2001年頃にC++ Users Journalは広告キャンペーンを行った
  - 「CUJに記事を投稿してください！」
  - 「英文学を専攻している必要はありません！
    エディタで始めてください！」
  - 「私たちはセキュリティ、ネットワーク技術、C++技術、その他の技術に興味があります！」

  注：まだ他のイテレータには興味がありませんでした

- 公開されたイテレータはどれくらい生き残った？

危険信号 #2

- およそ1975年のファイルコピー

```
#include <stdio.h>
int main() {
    int c;
    while ((c = getchar()) != EOF)
        putchar(c);
    return errno != 0;
}
```

危険信号 #2

- 20年ほど早送りして…

```
#include <iostream>
#include <algorithm>
#include <iterator>
#include <string>
using namespace std;

int main() {
    copy(istream_iterator<string>(cin),
       istream_iterator<string>(),
       ostream_iterator<string>(cout,"¥n"));
}
```

(mainの中にtry/catchを書くのを忘れた)

(何かどこか、逆にひどくなってしまった)

危険信号 #3

- イテレータは定義するのが非常に難しい
- かさばった実装と多くの了解事項(gotcha)
- Boostはイテレータの定義を支援する完全なライブラリを含んでいる
- 本質的なプリミティブは…3つ？
  - 終了(At end)
  - アクセス(Access)
  - 衝突(Bump)

危険信号 #4

- イテレータはポインタの構文とセマンティクスを使用する
- 勝利／敗北のポインタとの統合
- しかし、これはイテレーションの方法を制限する
  - パラメータを持った++が必要なので、深いツリーを歩かせることができない
  - OutputIteratorはひとつの型しか受け取ることができない：
    それらは全て同じ場所へ行くが、output_iteratorは出力するそれぞれの型でパラメータ化されなければならない

最終的な命取り

- 全てのイテレータプリミティブは基本的に安全ではない
- ほとんどのイテレータは、与えられたイテレータのために
  - 比較できるかどうかを記述することができない
  - インクリメントできるかどうかを記述することができない
  - 間接参照できるかどうかを記述することができない
- 安全なイテレータを書くことができる
  - 高いサイズ+速度のコストで
  - たいていは、設計をカットできなかったというよい議論
    (訳注：かなり怪しい訳)

Enter Range

- これらの不便さを部分的に回避するため、Rangeが定義された
- Rangeは、begin/endイテレータの組をパックしたもの
- そのため、Rangeはより高レベルのcheckable invariant(訳注：チェック可能な不変式？)を持つ
- BoostとAdobeのライブラリはRangeを定義した

それらはよい方向におもしろい一歩を踏み出した

より一層理解されなければならない

ほら、どこにもイテレータがない！

- イテレータの代わりにRangeをイテレーション用の
  基本構造として定義してはどうだろう？
- Rangeは、イテレーションに依存しない基本処理を定義するべき
- 今後イテレータはなく、Rangeだけが残るだろう(訳注：怪しい)
- Rangeはどのプリミティブをサポートしなければならないか
  begin/endがオプションではないことを思い出してほしい
  人々が個々のイテレータを貯め込んでいたら、我々は振り出しに戻らなければならない

Rangeの定義

- <algorithm>は全てRangeで定義でき、他のアルゴリズムも同様にRangeで実装できる
- Rangeプリミティブは、少ないコストでチェック可能でなければならない
- イテレータより非効率であってはならない

Input/Forward Range

```
template<class T> struct InputRange {
    bool empty() const;
    void popFront();
    T& front() const;
};
```

証明可能？

```
template<class T> struct ContigRange {
    bool empty() const { return b >= e; }
    void popFront() {
        assert(!empty());
        ++b;
    }
    T& front() const {
        assert(!empty());
        return *b;
    }
private:
    T *b, *e;
};
```

検索

```
// オリジナル版のSTL
template<class It, class T>
It find(It b, It e, T value) {
    for (; b != e; ++b)
    if (value == *b) break;
    return b;
}
...
auto i = find(v.begin(), v.end(), value);
if (i != v.end()) ...
```

設計質問

- Rangeでのfindはどのように見えなければならないか？
  1. 範囲のひとつの要素(見つかった場合)、
     もしくは0個の要素(見つからなかった場合)を返す？
  2. 見つかった要素以前のRangeを返す？
  3. 見つかった要素以降のRangeを返す？
- 正解：(もしあった場合)見つかった要素で始まるRangeを返し、そうでなければ空を返す

  なぜ？

検索

```
// Rangeを使用
template<class R, class T>
R find(R r, T value) {
    for (; !r.empty(); r.popFront())
        if (value == r.front()) break;
    return r;
}
...
auto r = find(v.all(), value);
if (!r.empty()) ...
```

エレガントな仕様

```
template<class R, class T>
R find(R r, T value);
```

「frontがvalueと等しいか、rが使い尽くされるまで、
左から範囲rを縮小する」

Bidirectional Range

```
template<class T> struct BidirRange {
    bool empty() const;
    void popFront();
    void popBack();
    T& front() const;
    T& back() const;
};
```

Reverse Iteration

```
template<class R> struct Retro {
    bool empty() const { return r.empty(); }
    void popFront() { return r.popBack(); }
    void popBack() { return r.popFront(); }
    E<R>::Type& front() const { return r.back(); }
    E<R>::Type& back() const { return r.front(); }
    R r;
};
template<class R> Retro<R> retro(R r) {
    return Retro<R>(r);
}
template<class R> R retro(Retro<R> r) {
    return r.r; // klever(訳注：clever？ ： 賢いところ)
}
```

find_endはどうだろう？

```
template<class R, class T>
R find_end(R r, T value) {
    return retro(find(retro(r));
}
```

- rbegin, rendは必要ない
- コンテナは、範囲を返すallを定義する
- 後ろにイテレートする：retro(cont.all())

イテレータでのfind_endは最低だ

```
// reverse_iteratorを使用したfind_end
template<class It, class T>
It find_end(It b, It e, T value) {
    It r = find(reverse_iterator<It>(e),
        reverse_iterator<It>(b), value).base();
    return r == b ? e : --r;
}
```

- Rangeが圧倒的に有利:はるかに簡潔なコード
- 2つを同時に扱うのではなく、1つのオブジェクトのみから構成されるので、容易な構成になる

さらなる構成の可能性

- Chain：いくつかのRangeをつなげる
  要素はコピーされない！
  Rangeのカテゴリは、全てのRangeの中で最も弱い
- Zip：密集行進法(lockstep)でRangeを渡る
  Tupleが必要
- Stride：一度に数ステップ、Rangeを渡る
  イテレータではこれを実装することができない！
- Radial：中間(あるいはその他のポイント)からの距離を増加させる際にRangeを渡る

3つのイテレータの関数はどうだろうか？

```
template<class It1, class It2>
void copy(It1 begin, It1 end, It2 to);
template<class It>
void partial_sort(It begin, It mid, It end);
template<class It>
void rotate(It begin, It mid, It end);
template<class It, class Pr>
It partition(It begin, It end, Pr pred);
template<class It, class Pr>
It inplace_merge(It begin, It mid, It end);
```

「困難なところにはチャンスがある。」

"Where there's hardship, there's opportunity."

– I. Meade Etop

3-legged algos ⇒ mixed-range algos

```
template<class R1, class R2>
R2 copy(R1 r1, R2 r2);
```

- 仕様:r1からr2へコピーして、手をつけていないr2を返す

```
vector<float> v;
list<int> s;
deque<double> d;
copy(chain(v, s), d);
```

3-legged algos ⇒ mixed-range algos

```
template<class R1, class R2>
void partial_sort(R1 r1, R2 r2);
```

- 仕様：最も小さな要素がr1の中で終了するように、r1とr2の連結を部分的にソートする

- あなたは、vectorとdequeをとり、両方の中で最も小さな要素を配列に入れることができる

```
vector<float> v;
deque<double> d;
partial_sort(v, d);
```

ちょっと待って、まだある

```
vector<double> v1, v2;
deque<double> d;
partial_sort(v1, chain(v2, d));
sort(chain(v1, v2, d));
```

- アルゴリズムは今、余分な労力なしで任意のRangeの組み合わせにおいても途切れることなく動作することができる
- イテレータでこれを試してみて！

ちょっと待って、さらにある

```
vector<double> vd;
vector<string> vs;
// 密集行進法(lockstep)で2つをソートする
sort(zip(vs, vd));
```

- Rangeコンビネータは、無数の新しい使い方ができるようになる
- イテレータでも理論上は可能だが、(再び)構文が爆発する

Output Range

- ポインタ構文からの解放されたので、異なる型をサポートすることが可能になった

```
struct OutRange {
    typedef Typelist<int, double, string> Types;
    void put(int);
    void put(double);
    void put(string);
};
```

stdinからstdoutにコピーする話に戻ろう

```
#include <...>
int main() {
    copy(istream_range<string>(cin),
        ostream_range(cout, "¥n"));
}
```

- 最後にもう一歩：1行(one-line)に収まる寸言 (one-liner) [1]
- ostream_rangeにはstringを指定する必要はない

[1] スライド制限にもかかわらず

Infinite Range(無限の範囲)

- 無限についての概念はRangeでおもしろくなる
- ジェネレータ、乱数、シリーズ、... はInfinite Range
- 無限は、5つの古典的カテゴリとは異なる特性;
  あらゆる種類のRangeが無限かもしれない
- ランダムアクセスなRangeさえ無限かもしれない！
- 無限について静的に知っておくことは、アルゴリズムを助ける

has_size

- Rangeが効率的に計算されたsizeを持っているかどうかは他の独立した特性
- 索引項目 : list.size, 永遠の討論
- Input Rangeさえ、既知のサイズを持つことができる
  (たとえば、100個の乱数をとるtake(100, rndgen))
  - rが無限の場合、take(100, r)は100の長さ
  - rが長さを知っている場合、長さはmin(100, r.size())
  - rが未知の長さで有限の場合、未知の長さ

予想外の展開(A Twist)

- Rangeで<algorithm>をやり直すことができる？
- Dのstdlibは、std.algorithmとstd.rangeモジュールで<algorithm>のスーパーセットを提供している(googleで検索してほしい)
- RangeはD全体に渡って利用されている:
  アルゴリズム、遅延評価、乱数、高階関数、foreach文…
- いくつかの可能性はまだ簡単には挙げられなかった
  – たとえばfilter(input/output range)
- Doctor Dobb's Journalの「The Case for D」をチェックしておいてください。
  coming soon...

結論

- Rangeは優れた抽象的概念である
- より良いチェック能力(まだ完全ではない)
- 容易な構成
- Rangeに基づいた設計は、イテレータに基づいた関数をはるかに超えるものを提供する
- 一歩進んだSTLによる刺激的な開発