今、此処にあるC++

～テンプレートとSTL～

アジェンダ

- テンプレートの解説
- コンテナ
- イテレータ
- アルゴリズム

テンプレートとは

- 型を外部から変更可能にした構文

通常の構文

```
int max(int a, int b){
  return a > b ? a : b;
}
```

テンプレートを使った構文

```
template <typename T>
T max(T a, T b){
  return a > b ? a : b;
}
```

穴埋めコラム
テンプレート内で、T::Aと書いた場合T::Aは変数とみなされる。
これがtypedefとか、内部に定義したstructとかだとコンパイルエラーが出る。
それを埋めるのが、typenameというキーワード。
typename T::A a; と書けば型と見てくれるようになる。

defineとtemplateの比較

#defineの構文

```
#define max(a, b)  ¥
  ((a)>(b) ? (a) : (b))



void hoo(int a, int b){
  int c = max(a, b);
}
```

template の構文

```
template <typename T>
T max(T a, T b){
  return a > b ? a : b;
}

void hoo(int a, int b){
  int c = max(a, b);
}
```

defineとtemplateの比較 展開後のイメージ

#defineの展開後

```
void foo(int a, int b){
  int c = a > b ? a : b;
}
```

templateの展開後

```
int max(int a, int b){
  return a > b ? a : b;
}

void foo(int a, int b){
  int c = max(a, b);
}
```

テンプレートの欠点

- テンプレート単体ではエラーが出ない
  実体を定義したとき内部の文法がチェックされる。
- エラーがわかりにくい
  大量のエラーが発生する可能性がある。
  C++0xのconceptはこれを回避する目的がある。
- サイズの肥大化
  使用した型毎にコードが生成される。
- ロジックをヘッダファイルに書くことになる
  修正の度に関連するソースすべてにリコンパイルが必要になる。

テンプレートとは

どう使えばよいのか

- defineで実装していたマクロの代替
- 式を関数に変換
- ビット幅や整数・実数を可変にした数値型
- 拡張版 void *
- 実質的兄弟クラスとしての利用
- インライン展開

STL (**Standard Template Library**)

- テンプレートを使って構成されたライブラリ
- 現在のC++で標準ライブラリとして収録
- std名前空間に属する
- 代表的なものとして以下のような要素がある
  - コンテナ
  - イテレータ(反復子)
  - アルゴリズム
  - 関数オブジェクト/関数アダプタ

コンテナ

- 物を入れる箱
- 主に以下のものが提供される
  - 可変長配列　(vector, deque, list)
  - ソート済み連想配列 (set,map)
  - ハッシュによる連想配列 (hash_set/hash_map)

穴埋めコラム
ハッシュによると書いているが、別にハッシュを使って実装する必要はない。
これは、STLが「要件を満たせば実装方法はなんでもよい」という方針だから。
（要件を考えると、実質的にハッシュを用いる事になると思うが。）

上記の理由から、TR1からはunordered_set(map)という名前に変わっている。
文字通り、順番に並んでいないというところを強調している。

可変長配列

- vector/deque/listが相当する
- 使い方はほぼ同じだが、
  処理速度やメモリ消費量が違う

型名	説明
vector	通常の配列とほぼ同じ使い方ができる 要素が減ってもメモリが自動解放されない 要素が頻繁に挿入/削除される場合には向かない
deque (double ended queue)	キュー/スタック構造に向く 先頭/末尾の挿入/削除は速いが、途中の挿入/削除は遅い
list	要素の挿入/削除が速い 挿入/削除でイテレータが参照を失わない(そのものの削除以外) メモリ消費が最も多く、ランダムアクセスできない

可変長配列の関数表

関数名	説明	V	D	L
push_back	末尾に要素を追加する	※	○	○
pop_back	末尾の要素を削除する	○	○	○
push_front	先頭に要素を追加する	―	○	○
pop_front	先頭の要素を削除する	―	○	○
insert	任意の場所に要素を追加する	×	×	○
erase	任意の場所から要素を削除する	×	×	○
clear	すべての要素を削除する	○	○	○
operator []	n番目の要素を得る	○	○	―
size	要素の個数を得る	○	○	○

V:vector D:deque L:list

○:速い ×:遅い ―:関数がない
※ vectorのpush_backはメモリ拡張が起こる時遅くなる

vectorのサンプルプログラム

```
vector<int> v;

for (int i = 0; i < 100; i++){
   v.push_back(i);
}

for (int i = 0; i < v.size(); i++){
  printf(“%d¥n”, v[i]);
}
```

要素の挿入
push_backを使うと、
メモリが許す限り挿入が可能

要素の参照
通常の配列と同じ感覚で
使用することができる

しかし、listの場合はoperator [] がないので
別の書き方が必要・・・

イテレータ

listの全要素を表示するには以下のように記述する。

```
list<int> l;
list<int>::iterator it;
for(it = l.begin(); it != l.end(); it++){    printf(“%d¥n”, *it);   }
```

そして、vector/dequeも同じように書ける

```
vector<int> v;
vector<int>::iterator it;
for(it = v.begin(); it != v.end(); it++){    printf(“%d¥n”, *it);   }
```

ここの書式がほぼ一緒

ループ内部にコンテナの変数名が
まったく登場していない

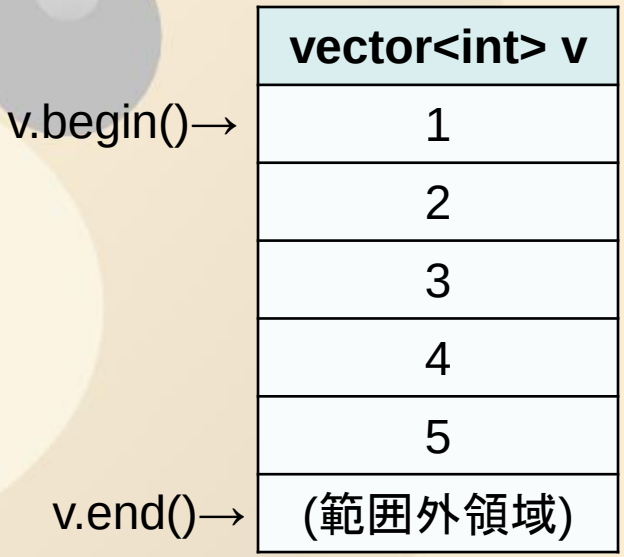
イテレータ

- シーケンスの各要素の参照の抽象化
  要は、データの集合に対し順番にアクセスする方法
- イテレータは以下の要素をもつ
  - 間接参照(*it)
  - 前進(it++)
  - 位置の比較(it1 != it2)
- Cのポインタと互換性をもたせる工夫がある

穴埋めコラム
上の説明は外部イテレータの事であり、他に内部イテレータというものもある。
foreachのように前進とループ終了判定を書かなくていいものが内部イテレータ。
C#のyieldの項目で出てくる反復子は内部イテレータのこと。

イテレータと範囲

1. it == v.end()でループを抜ける
2. v.end()の要素はループを通らない
3. 右図のように、v.end()の要素は
   参照してはいけない
   領域を指している



穴埋めコラム
v.end()の内容を参照した場合は未定義である。
これ以外に要素が0なのにpop_backしたり、
参照が失われたイテレータを操作したりしても
未定義であり、例外が出るわけではない。
未定義とは、何が起こっても知らんって意味。
STLは例外を出さない実装が多い。

なぜ終了点を含まないのか

- for文で簡潔に書ける
- first=lastとすることで、空リストを表せる
- insertで挿入できる場所は
  「要素の数＋１」ある

穴埋めコラム
C#のIEnumerator はResetとMoveNextで外部イテレータを実装している。
Resetで取得できる場所は、先頭要素の1つ前を想定した場所であり、
MoveNextを一度行うことで先頭要素を示すことになる。
サンプルプログラムには最初のMoveNextを無視するフラグが入ってたりする。

```
var it = array.Reset();
while (it.MoveNext()){ Console.WriteLine(it.Current()); }
```

リバースイテレータ

- 逆向きに動作するイテレータ
  ++ で前の要素、--で次の要素に移動する
- begin,endの代わりにrbegin,rendを使用する

```
vector<int> v;
vector<int>::reverse_iterator it;
for(it = v.rbegin(); it != v.rend(); it++){
  printf("%d¥n", *it);
}
```

穴埋めコラム
reverse_iteratorは、iteratorにキャストできない。
同じ要素を指すiteratorに変換したい場合はこう書くとよい。
it = --rev_it.base();
なお、v.rbegin().base() == v.end() が成り立つ。

インサートイテレータ

- ＝演算子で値を入れることで
  コンテナに要素が挿入されるイテレータ
- 通常のイテレータと違い、
  前進(it++)や間接参照(*it)は意味を持たない

型名	説明
front_insert_iterator	push_frontを使って挿入する vectorではpush_frontがないため、使えない
back_insert_iterator	push_backを使って挿入する
insert_iterator	insertを使って挿入する

インサートイテレータ

- インサートイテレータの使用方法

```
vector<int> v;
back_insert_iterator<vector<int> > ins(v);

for(int i = 0; I < 10; i++){
  // v.push_back(i); と同じ意味
  *ins++ = i;
}
```

上のサンプルの「*ins++ = i」は、「ins = i」でも全く問題ない。
「*ins」も「ins++」も何も処理を行わず自分自身を返すため、
コンパイル後は全く同じコードが生成されることになる。
それでも、このように書く理由がある。

インサートイテレータを使う

コピー関数をテンプレートで作成

```
template<typename A, typename B>
void copy(A first, A last, B ins){
  for(A it = first; it != last; it++){
    *ins++ = *it;
  }
}
```

配列を渡す場合は先頭と末尾を渡し、
受け取る時はインサートイテレータを
考慮した構文である。

使い方例

```
int a[5], b[5];
copy(a, a + 5, b);

// for(int *it = a; it != a + 5; it++){
//  *b++ = *it;
// }

vector<int> v;
back_insert_iterator bi(v);
copy(a, a + 5, bi);

// for(int *it = a; it != a + 5; it++){
//   *bi++ = *it;
// }
```

アルゴリズム

- テンプレートを使ったライブラリ集
- 今回作ったmax,copyも収録されている
- シーケンスに対する操作を行う関数が多い
  その際、イテレータを使って範囲を渡す

穴埋めコラム
アルゴリズムの中にはコールバック関数を伴うものもある。
通常は関数ポインタを渡すが、関数の呼び出し風に記述できれば何でもよい。
簡単な処で#defineのマクロが思いつくが、これは型を持たないのでNGである。

そこで、クラスを作り、operator()をオーバーロードしたものを用意する。
これを関数オブジェクトと言い、インライン展開されるので高速に動作する。
現在は、かなり野暮ったい構文になるのでなかなか使いづらいが、
C++0xのラムダ式が加わると、直観的な記述が可能になる。

アルゴリズムの関数の抜粋

関数名	説明
find	[first,last)区域からvalueと同値のイテレータを返す。
count	[first,last)区域からvalueと同値の要素の数を数える。
copy	[first,last)区域をinsert_iteratorにコピーする。
fill	[first, last)区域にvalueを代入する。
reverse	[first, last)区域の値を逆にする。
random_shuffle	[first, last)区域をランダムに入れ替える。
sort	[first, last)区域をソートする。
lower_bound	[first, last)区域からvalueと同値かそれ未満のイテレータを返す。
binary_search	[first, last)区域からバイナリサーチを行い、要素があるか返す。
next_permutation	[first, last)区域を次の順列にする。
swap	AとBを入れ替える。
max	AとBの大きい方を返す。

最後に

- STLはあらゆるものを抽象化しようとしている
- 紹介しきれなかった機能で便利なもの
  - map
  - ストリーム
  - string
- でもまだまだあると便利なものは多い
  →Boostに続く