Sapporo.cpp 札幌C++勉強会 #13

最近、仕事でC++11以降の新規格が役に立ったシーン紹介

H.Hiro @h_hiro_
http://hhiro.net/

自己紹介

H.Hiro

- 札幌出身名古屋在住（ただいま帰省中）名古屋3年目
- C++は仕事で書くことが多いです
  - アルゴリズム作ったり、処理速度を求められる計算機実験をしたりしてます

2017年が終わったので

こんな記事を書いた
https://qiita.com/h_hiro_/items/8d4d24b6ef56ef3ad7d9

大事なこと

さっきの記事から抜粋

- 昨年は<set>や<map>も割と使っていたが、今年はかなり限られた回数しか使わなかった記憶。もう今となっては「C++11で標準化されているなら無条件に利用する」くらいの気持ちです。~~そうは言っても、先日はgcc4.4の環境を利用するという機会があった~~
- <algorithm>では主にstd::all_ofとかstd::find_ifとかを活用した。std::all_ofがC++11で規格化されて、多くの環境で普通に使えるようになったのはありがたかったです。

C++11あたりなら、当たり前に使えるようになることが多くなってます

- だいぶ楽できるようになりました
  - でもこないだ、gcc4.4を使うことになりそうな時が…
    （結局使わなくてよいことになったのですが）
- Visual C++が新規格への対応遅いほうですが（な印象）
  流石にVC2015ともなると、C++11はほぼ対応してます
  - こないだ、Windows APIの引数にラムダ式を使ってみたら普通に使えましたね
    https://qiita.com/h_hiro_/items/da1aff25bba8ff0bc458

C++11あたりなら、当たり前に使えるようになることが多くなってます

- 私は利用している主な環境がUbuntuなこともあって最近はUbuntuでデフォルトで使えるgcc5前提のことが多いです（2018.1.2現在、gcc5.4.0）
  - ちなみにWindows環境ではVCのほかにMSYS2上のgccも使うのですが、こちらのpacmanがもうgcc7.2.0まで対応してるという…

じゃあ実際どんなコードを書いたのか

最近私が使った、C++11（以降）で規格化された
言語仕様や標準ライブラリはこんなところ

- 標準ライブラリ
  - <random>
  - <unordered_set>, <unordered_map>
  - <algorithm>に追加された関数（std::all_ofなど）
- 言語仕様
  - ラムダ式
  - オブジェクトのムーブ
  - using new_type_name = type_name;

<random>

<random>：概要

疑似乱数を生成するための標準ライブラリ

C言語のrand関数に比べて進んでいるところ

- rand関数は「0からRAND_MAXまでの範囲の整数」という乱数しか生成できなかったので、所望の乱数の分布（「0から9までの数を等確率」など）を得るためには、自分の手で加工する必要があった。
  - C++の<random>なら、代表的な分布であればライブラリ内で用意されたクラスを通すだけで使える。
- 疑似乱数列を生成するメカニズムが何種類か用意され、切り替えも容易にできるようになっている。
  逆に、疑似乱数列を生成するメカニズムにこだわらない場合でも、C++の<random>を使うなら指定は必須。そこはrand関数の場合（srand関数だけ使えばいい）に比べると手間かも

<random>：構成

- 乱数の分布（uniform_int_distributionなど）
- 疑似乱数列を生成するメカニズム（mersenne_twister_engineなど）
- 乱数を直接発生させる方法（random_device）
  - 物理的な乱数なので、計算で出力する乱数（疑似乱数）の場合に問題となる「次の値の推測」が不可能
  - これまでもOS等で用意された関数を呼ぶことで利用はできたが、これがあれば環境によらず一つのコードで対応できる
  - 一部非対応の環境あり（mingwなど）
- その他

<random>：利用例

基本

https://wandbox.org/permlink/8nhJ1LczbQ316JSF

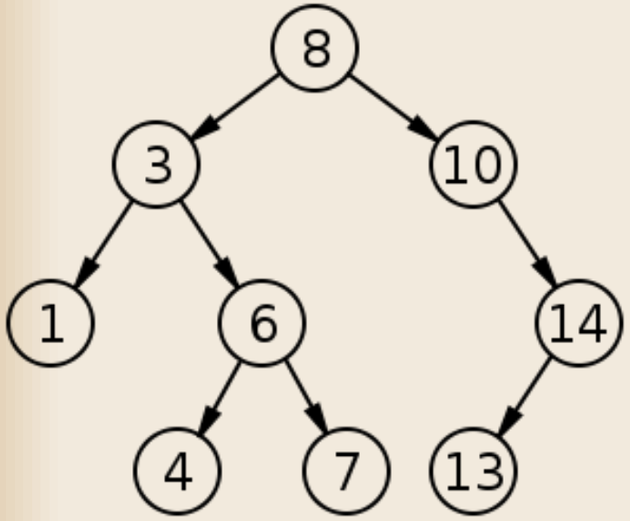
応用：「配列等に入った要素から、k個を重複なしに選ぶ」

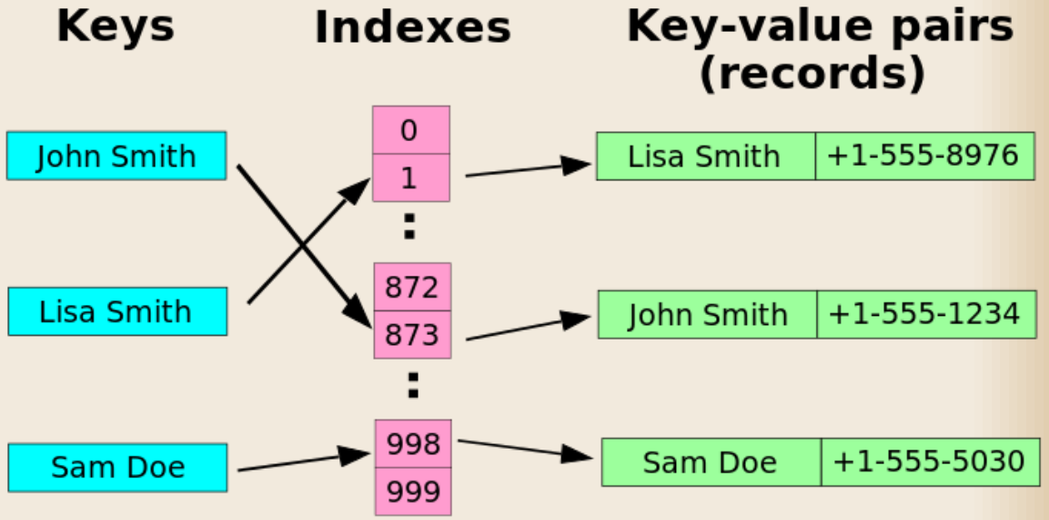
1. std::shuffle して、先頭k個を見る
   簡単です。元の配列が並び変わってもよい+元の配列がそんなに長くないのなら、これでよいでしょう。
   https://cpprefjp.github.io/reference/algorithm/shuffle.html
2. 選ばれるべき確率を計算して選ぶ
   ちょっと複雑です。2つアルゴリズムを示します。
   後者はランダムアクセス不能なデータ構造に対しても使えます。
   https://wandbox.org/permlink/nTSoo3ov43RneLAf
   https://wandbox.org/permlink/VXsn89fWJcwmBeC4

<unordered_set>,
<unordered_map>

<unordered_set>, <unordered_map>：概要

- 連想配列
- 以前からあった<set>や<map>（二分探索木で実装）と異なり、ハッシュテーブルで実装される
  - 多くの場合、より効率的に動作





左：二分探索木の例（左の子は親より小さく、右の子は親より大きい割り当て）
https://commons.wikimedia.org/wiki/File:Binary_search_tree.svg
右：ハッシュテーブルの例（名前を一定の規則で数字に対応付け呼び出し）
https://commons.wikimedia.org/wiki/File:HASHTB08.svg

<unordered_set>, <unordered_map>：概要

- 連想配列
- 以前からあった<set>や<map>（二分探索木で実装）と異なり、ハッシュテーブルで実装される
  - 多くの場合、より効率的に動作
- **基本的な使い方は<set>や<map>と変わらないので今回の説明からは省いてしまいます**
- 今回は、<set>や<map>と使い方が変わる部分である「そのままでは格納できないものをどう格納するか」を説明します

<set>にそのままでは格納できないものとは？

- 例えば、setに自前のクラスや構造体を格納しようとしても、通常はコンパイルエラーになる
（「set」を「mapのキー」と読み替えても同様）
https://wandbox.org/permlink/c4Fvh7TYez7BOLLp

<set>にそのままでは格納できないものとは？

- 例えば、setに自前のクラスや構造体を格納しようとしても、通常はコンパイルエラーになる
（「set」を「mapのキー」と読み替えても同様）
https://wandbox.org/permlink/c4Fvh7TYez7BOLLp
- なぜなら、setは**大小関係に基づいて要素を格納する**ので大小関係が比較できない型はsetに指定できない
- このように、大小関係を定義してあげればOK
https://wandbox.org/permlink/3ZcDpmiICFIhtqCv

<unordered_set>にそのままでは格納できないものとは？

- unordered_set（unordered_mapのキー）についても似た話がある
  例えば、以下のコードはコンパイルエラー
  https://wandbox.org/permlink/FdUyncB6k2g5dCyI
- ハッシュテーブルを使っているわけなので、
  ハッシュ関数を定義してあげないとならない
  - 整数型やstring型などには最初から定義されている
- 実際にはそれだけでなく、operator== も必要
- 以下のようにすれば動く
  https://wandbox.org/permlink/AkDzvBEz04TaY9gK
  - ハッシュ関数がテンプレートパラメータで指定できるのがポイント

補足：ハッシュ関数の定義方法

- ハッシュ関数を自分で定義するのは深い話があり
  実装するにもコツがある
  - 入力のランダムさに対して同等にランダムに
    なっているべき
  - 中に配列などサイズ可変のものが入っているような
    場合でも、時間はさほどかからないべき
- 私の場合のやり方
  - 要素数固定の場合は、ビットシフトと
    「^」（XOR）を組み合わせる
  - 要素数可変の場合は、「数要素ずつ読み飛ばした
    結果をXORする」（読み飛ばす数も要素に依存）
    例えばintの配列なら、
    「(総要素数/8)+値を8で割った余り」だけ読み飛ばす

<algorithm>への関数追加

<algorithm>ヘッダについて

- C++ STLの一部
- コンテナ型（vector, listなど）に対し
  「並び替え」「検索」などの汎用的な処理を提供する
  - 例：count_if（条件を満たす要素数を数える）
    ```
    std::count_if(foo.begin(), foo.end(),
        [](int x){ return x % 2 == 0; } );
    ```
    →**fooがstd::vector<int>でもstd::list<int>でも**
    その他何でも動く
    https://wandbox.org/permlink/EOz1s2uuGWyGSgpu
- C++11で、関数がいろいろと追加された

C++11で、<algorithm>に追加された関数で役だったもの(1)

std::all_of(begin, end, pred)
begin～endの要素をpredに与えたとき、
すべてtrueならばtrue、そうでなければfalse
https://cpprefjp.github.io/reference/algorithm/all_of.html

- any_of, none_of というものも同時に追加されている
  前者は「predに与えてどれか一つでもtrueならtrue」、
  後者は「predに与えてすべてfalseならばtrue」
- これができる前でも、find_ifを使って同等の処理は
  実現できたものの、より直感的に書けるようになった
  - all_of(条件) → find_if(条件の否定) で何も見つからない
  - any_of(条件) → find_if(条件) で何か見つかった
  - none_of(条件) → find_if(条件) で何も見つからない

C++11で、<algorithm>に追加された関数で役だったもの(2)

std::minmax_element(begin, end[, pred])

コンテナ中の最小要素・最大要素を検索する
（結果はイテレータで返す）
https://cpprefjp.github.io/reference/algorithm/minmax_element.html

- std::min_element, std::max_elementという関数はすでに存在していた
- ただし、それらを両方知りたい場合はstd::min_elementとstd::max_elementを両方呼んではループが二回かかって非効率であり、それを避けるには自分でループを書くしかなかった

ラムダ式

ラムダ式

- みんな大好きラムダ式
  - これができる前は、関数ポインタを使うか
    関数オブジェクト（operator()が定義された構造体）で
    別の場所に定義を書く必要があった
  - 私はRubyで、ラムダ式のありがたさを最初に知った
- 基本的な書き方

```
[キャプチャしたい変数](引数){処理};
```

  - C++の場合、特に指定がなければ、ラムダ式の中から
    外のローカル変数は参照できないので
    参照したいものを指定するのが「キャプチャ」
    https://cpprefjp.github.io/lang/cpp11/lambda_expressions.html

どんな場面で使ったか

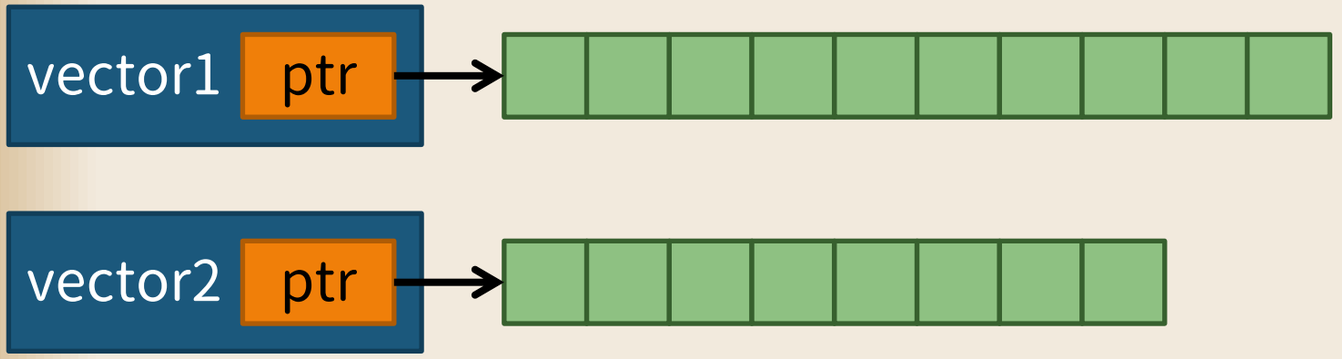
- さっき出てきた<algorithm>の各種関数に与えるのは
  もう当たり前にやってます
- それ以外にも、std::functionに代入して
  処理を切り替えるのに使ったり
  https://wandbox.org/permlink/UdrzAG3pUwq63zqp

オブジェクトのムーブ

オブジェクトの「ムーブ」とは

- C++11で規格化された言語仕様
- オブジェクトの内容を丸ごと移動してしまう（移動元からは内容は消える）
- 例えばstd::vectorのように、「実際に保持している内容は多いこともあるけど、その場合でも内容を丸ごと入れ替えるコストは小さい」場合に有用（実際、ポインタを移動してやるだけでよい）

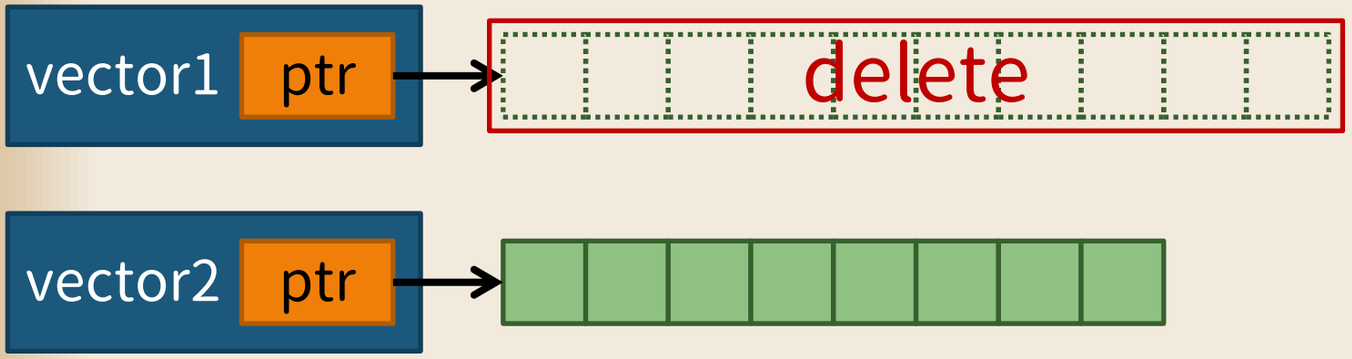
```
vector1 = std::move(vector2);
```



オブジェクトの「ムーブ」とは

- C++11で規格化された言語仕様
- オブジェクトの内容を丸ごと移動してしまう（移動元からは内容は消える）
- 例えばstd::vectorのように、「実際に保持している内容は多いこともあるけど、その場合でも内容を丸ごと入れ替えるコストは小さい」場合に有用（実際、ポインタを移動してやるだけでよい）

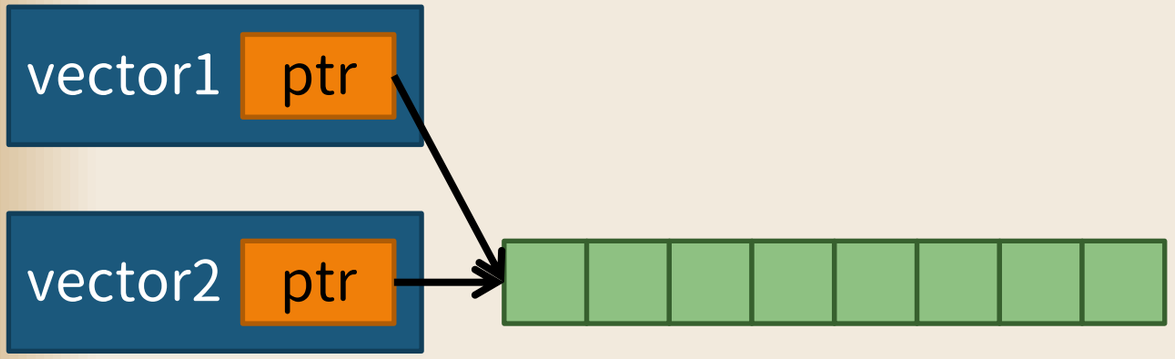
vector1 = std::move(vector2);



オブジェクトの「ムーブ」とは

- C++11で規格化された言語仕様
- オブジェクトの内容を丸ごと移動してしまう（移動元からは内容は消える）
- 例えばstd::vectorのように、「実際に保持している内容は多いこともあるけど、その場合でも内容を丸ごと入れ替えるコストは小さい」場合に有用（実際、ポインタを移動してやるだけでよい）

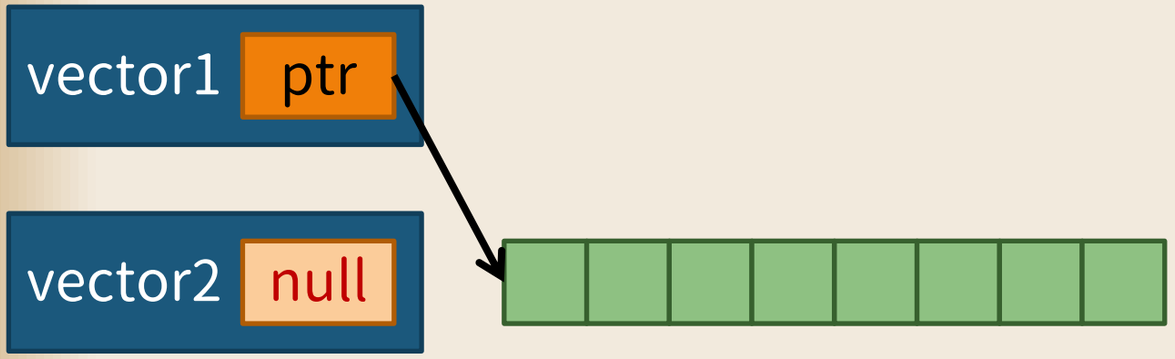
vector1 = std::move(vector2);



オブジェクトの「ムーブ」とは

- C++11で規格化された言語仕様
- オブジェクトの内容を丸ごと移動してしまう（移動元からは内容は消える）
- 例えばstd::vectorのように、「実際に保持している内容は多いこともあるけど、その場合でも内容を丸ごと入れ替えるコストは小さい」場合に有用（実際、ポインタを移動してやるだけでよい）

```
vector1 = std::move(vector2);
```



実際、どんな場面でmoveが必要になったのか

- 「配列を別の配列に書き換える」、という処理を何度も繰り返すプログラム
  ただし、n回目の配列は(n－1)回目の配列を見ながらでないと生成できないので、1つの配列では対応できない
- コード（の概念）はこんな具合
  https://wandbox.org/permlink/jpzaBmyzE3QHOUF7
- この例であれば、最初から2つ配列を用意するとかすることも考えられるのですが、実際には配列のサイズも可変であることから、moveがよいと判断しました

```
using new_type_name
= type_name;
```

usingによる
型のエイリアス（別名）定義

C++11では、typedef以外にusingでも型のエイリアスが定義できるようになった

https://qiita.com/Linda_pp/items/44a67c64c14cba00eef1

- 以下の2つは同じ
  - typedef int foo;
  - using foo = int;
- 以下の2つは同じ
  - typedef int (*bar)(int);
  - using bar = int(*)(int); // 見やすい！

usingによる
型のエイリアス（別名）定義

usingはtypedefと違い、テンプレート化もできる

https://qiita.com/Linda_pp/items/44a67c64c14cba00eef1

- ~~template <class T> typedef int (*bar)(T);~~ // できない
- template <class T> using bar = int(*)(T); // OK

もともとテンプレート化するという需要はあったけど
typedefしかなかったときはそれができなかったので
代わりにtemplateで構造体を定義したりしていた
（上記記事参照）

どんな場面で使ったか

C++11対応環境ならさっきの<random>で乱数を生成し
そうでなければrand()で乱数を生成するためのコード
https://wandbox.org/permlink/OFU97u5z6Xtdaiu5
すごい長いけど、ポイントはここ

```
// 整数の一様分布を定義
#ifdef USE_LEGACY_RANDOM
  // C++11のrandomが使えないときは、自前の実装を利用
  template <class T>
  class DistInt{
    // 略
  };
#else // USE_LEGACY_RANDOM
  // C++11のrandomが使えるときは、単にそれを利用
  template <class T> using DistInt = std::uniform_int_distribution<T>;
#endif // USE_LEGACY_RANDOM
```

おわりに

おわりに

- C++11標準の機能を使うとこんなところで便利になるよ！というのを、
  実例（経験）を踏まえて紹介させていただきました
- もうそろそろ、（使うコンパイラが対応しているなら）
  このあたりの機能は普通に使うべきかなと感じてます
- C++14, C++17が普通に使える環境が
  早く広まりますように…