C++11 概要 ライブラリ編

Egtra

2012 年 6 月 23 日

1　このテキストについて

この文章は，Boost.勉強会 #9 つくばで発表した内容を文字にしたものです。スライドに載せなかった情報をできる限り伝えることを目的として作成しました。また，発表中に指摘された内容も反映してあります。ご指摘くださった皆様に感謝いたします。

1.1　注意

スライドにも書きましたが，C++11 の標準ライブラリのすべてを網羅しているわけではありません。日常的に用いることになるであろうものを中心に選びました。

1.2　ライセンス

2　Misc

2.1　スマートポインタ

C++11 には，スマートポインタとして unique_ptr と shared_ptr が存在します。

```
// #include <memory>
std::unique_ptr<int> up(new int(1));
```

```
std::unique_ptr<int[]> ua(new int[]{1, 2, 3});

auto sp = std::make_shared<int>(3);
```

unique_ptr は，参照カウント方式を採用していない軽量なスマートポインタです。以下の上位互換として使えます。

- std::auto_ptr
- boost::scoped_ptr
- boost::scoped_array

これらと異なり std::vector などコンテナの要素にできます。

なお，unique_ptr の導入により，auto_ptr は非推奨になりました。

shared_ptr は，参照カウント式かつ動的削除子を採用する高機能なスマートポインタです。使用方法は，boost::shared_ptr と同じと考えて構いません。

上記の例では，shared_ptr オブジェクトの作成に make_shared 関数を使っています。make_shared 関数は，管理対象のオブジェクト（std::make_shared<int> なら int）の領域と参照カウント用の領域を 1 度にまとめてメモリ確保する優れものです。もちろん，shared_ptr のコンストラクタに new で得たポインタを渡す方法も利用可能です。

```
std::shared_ptr<int> sp2(new int(4));
```

当日質問がありましたが，残念ながら boost::shared_array に相当するものはありません。

2.2 関数オブジェクト

Boost に存在した以下のものが導入されました。

- std::function
- std::ref
- std::bind
- std::mem_fn

なお，ラムダ式を用いたほうが分かりやすいことが多いため，bind をあまり積極的に利用する必要はないと私は考えています。さらに付け加えるなら，私が何度か試した限り bind と比較すると，ラムダ式のほうが僅かながら速いコードになることが多かったです。

2.3 整数型 (C99)

C++11 では C99 のライブラリがほぼすべて導入されましたが，中でも一押しとして <cstdint> の整数型に関する typedef を紹介します。

<cstdint> をインクルードすると，環境に応じて以下のようなビット数固定の整数型が利用できます。

- std::int8_t, std::uint8_t
- std::int16_t, std::uint16_t
- std::int32_t, std::uint32_t
- std::int64_t, std::uint64_t

これらの型は対応するビット数の符号あり・符号なしの整数型が存在する場合に typedef で定義されます。今まで，何らかのライブラリで，もしくは自らの手でこのような型を定義することは多々あったと思いますが，ついに標準ライブラリで用意されることになりました。

また，ポインタと同じ大きさである整数型として，intptr_t と uintptr_t が定義されます。32 ビット環境・64 ビット環境が混在する現在，有用な型でしょう。

このほか，以下のような型が存在します。

int_fast8_t 8 ビット以上で，最も高速に処理できる符号あり整数型。すなわち対象 CPU などによっては 16 ビットや 32 ビット整数型の可能性がある。

int_least8_t 少なくとも 8 ビット以上の符号あり整数型。int8_t と異なりいかなる環境でも定義される。

intmax_t その環境で最も大きな符号あり整数型。

それぞれ，8／16／32／64 ビット（intmax_t を除く）および符号あり・符号なしのそれぞれが用意されます。

2.4 乱数

乱数を完全に失念しておりました。C++11 には，C からの rand 関数よりも良い乱数を生成するライブラリが存在します。

おまけとして本テキストでサンプルを追加しようかとも考えましたが，十分な解説がす

でにありましたのでその紹介にとどめます。

本の虫: C++0x の新しい乱数ライブラリ，random
http://cpplover.blogspot.jp/2009/11/c0xrandom.html

3 コンテナ

3.1 新コンテナ

以下のコンテナが追加されました。

Unordered コンテナ ハッシュマップなど，ハッシュを利用する連想コンテナ

- unordered_map
- unordered_multimap
- unordered_set
- unordered_multiset

array 固定長配列（boost::array 風）

forward_list 片方向リンクリスト

unordered という名前は，hash_map などでは既存のライブラリと同じあるいは似た名前となってしまうことを懸念したためです。unordered とは，イテレータによってアクセスした際に要素が順番に並んでいないという性質を表しています（逆に，map や set などはキーでソートされている）。

Unordered コンテナの基本的な使用方法は，従来からの map や set などと同じです。insert や erase など，同名のメンバ関数が存在します。もちろん，unordered_map では [] 演算子が使用可能です。

3.2 コンテナの初期化

配列の初期化と同様の構文がコンテナの初期化にも使用できるようになりました。

```
// #include <array>
// #include <vector>

int ra[] = {1, 2, 3};
std::array<int, 3> a = {1, 2, 3};
```

```
std::vector<int> v = {1, 2, 3};
```

代入も可能です。

```
a = {4, 5, 6};
v = {4, 5, 6};
```

3.2.1 構造体（集成体のクラス）との併用

次のような Point クラスがあったとします。

```
struct Point {
   double x;
   double y;
};
```

次のように初期化や push_back，insert などが記述できます。

```
std::vector<Point> vp = {
   {10, 0},
   {0, 20},
};
vp.push_back({0, 20});
vp.insert(v.begin(), {0, 0});
```

deque や list などが持っている push_front でも同様です。

3.2.2 クラス（非集成体）との併用

コンストラクタを持つ型についても同様の記述が可能です。

```
class RailwayLine {
public:
   RailwayLine(
      std::string const& start, std::string const& end);
};
```

上記のようなクラスを持つコンテナにおける初期化の例です。

```
std::vector<RailwayLine> line = {
   {"Akihabara", "Tsukuba"},
};
```

line の唯一の要素は RailwayLine("Akihabara", "Tsukuba") オブジェクトとなります。

3.2.3 ストリームをコンテナの要素にする

ムーブセマンティクスへの対応の結果，fstream 類や stringstream 類がコンテナの要素として使用できるようになりました。

```
std::vector<std::fstream> vf = {
   std::fstream("a.cpp"),
   std::fstream("a.out", std::ios_base::binary),
};
```

これらはコンストラクタに explicit が指定されているため，{"a.cpp"}などといった初期化子の構文が使用できません（Thank you: @Flast_RO by https://twitter.com/Flast_RO/status/206280203046105088）。

3.2.4 コピー不可能な型と emplace_back

一部のコンテナでは，このようなコピー不可能な型もコンテナの要素にできます。

```
class SanJigen : boost::noncopyable {
public:
   SanJigen(int x, int y, int z);
};
```

その場合，emplace_back メンバ関数で要素を追加できます。

```
// 図形
std::set<SanJigen> figure;

figure.emplace_back(3, 1, 4);

// → figure[0] == SanJigen(3, 1, 4)
```

もちろん，emplace_back はコピーやムーブが可能な型でも使用できます。積極的に利用していくのも良いでしょう。

当日の発表時には vector としていましたが，実際には vector では不可能でした。vector は要素にムーブまたはコピー可能であることを要求しているためです（Thank you: @SubaruG）。

3.3 map::at と unordered_map::at

map や unordered_map の [] 演算子は const なオブジェクトに対して使用できません。そこで，const／非 const を問わず使用できる at メンバ関数が追加されました。なお，at という名前のメンバ関数は，従来より vector や deque に存在します。

```
#include <unordered_map>

std::map<std::string, std::string> const yome = {
   {"Nyaruko", "Mahiro"},
   {"Kuko", "Nyaruko"},
   {"Hasta", "Mahiro"},
};

auto x = yome.at("Hasta");
// → x == "Mahiro"
```

この例では，"Hasta"をキーに"Mahiro"を取り出しています。

キーに存在しないものを at に与えると，vector などの at メンバ関数と同じく std::out_of_range が送出されます。

```
yome.at("Mahiro");
// std::out_of_range
```

このように，キー"Mahiro"で map に追加すれば at で対応する値が取り出せるようになります。

```
auto const yome2 = yome;
yome2.insert({"Mahiro", "Shantak-kun"});
auto y = yome2.at("Mahiro");
```

3.4 unordered_map のキー対応

自分の作った型やその他標準で対応していない型を std::unordered_map をはじめとする unordered コンテナのキーとして使えるようにする方法を紹介します。

例として，次のような My::Point 型を対応させるとします。

```
namespace My {
   struct Point {
      int x, y;
   };
}
```

必要なのは，等値演算子と std::hash の特殊化です。

```
namespace My {
   bool operator==(Point, Point);
}

namespace std {
   struct hash<My::Point> {
      std::size_t operator()(Point const& pt) const {
         return ……;
      }
   }; // 特殊化
}
```

これで Unorderd のキーに使用できるようになります。

```
std::unordered_map<My::Point, SanJigen> position;
```

なおここでは扱いませんが，このほか std::unordered_map のテンプレート引数で指定する方法があります。std::map などで，< 演算子を実装する方法以外に，std::map のテンプレート引数で比較用の関数オブジェクトを指定する方法が存在することと同じことです。

4　文字列

4.1　basic_string における要素の連続

std::basic_string クラステンプレート (std::string，std::wstring，std::u16string，std::u32string) でも，std::vector や std::array 同様，要素が連続したメモリアドレスに置かれるという保証が与えられるようになりました。

Windows の関数を用いた例で申し訳ないです。GetWindowText 関数の 2 番目の引数は TCHAR*型で，TCHAR の配列へのポインタを渡すことを想定しています。この例では，そこへ basic_string<TCHAR>の要素へのポインタを渡しています。

```
// int GetWindowText(HWND hwnd, THCAR*, int);

auto len = GetWindowTextLength(hwnd);
std::basic_string<TCHAR> t(len + 1);
auto actualLength = GetWindowText(hwnd, &s[0], len + 1);

s.resize(actualLength);
```

なお，スライドでは最後に s.pop_back(); としていましたが，GetWindowText の仕様上正しくありませんでした（Thank you: @melponn）。

4.2　文字列・数値変換

std::string や std::wstring を対象に，数値型との変換を行う関数群が追加されました。

4.3　文字列から数値への変換

数値への変換関数として，stoi 関数群が追加されました。

このようなプロトタイプであり，atoi 系と strtol 系両方の上位互換として使用できるようになっています。

```
// #include <string>

int stoi(const std::string& str,
```

```
    std::size_t* idx = 0, int base = 10);
int stoi(const std::wstring& str,
    std::size_t* idx = 0, int base = 10);
```

使用例です。

```
// atoi っぽく
auto x = stoi("103");

// strtol っぽく
std::size_t pos;
auto y = stoi("BEEF␣kue", &pos, 16);
```

なお，エラー時には例外が投げられます。

stoi は int 型への変換関数です。このほかの型に対応する関数として次のものが揃っています。

stoi int
stol long
stoll long long
stoull unsigned long long
stof float
stod double
stold long double

4.4 数値から文字列への変換

逆に，数値から文字列へ変換する関数として to_string および to_wstring 関数が追加されました。こちらは多重定義されており，sto*関数が存在する数値型に対応しています。

printf 群の %d や %f などと同じ書式で変換すると定められており，iostream よりも単純化されています。

```
// #include <string>

auto s = std::to_string(201);
```

```
auto ws = std::to_wstring(233.1000);
```

4.5 ワイド・ナロー変換

char ベースの文字列と wchar_t ベースの文字列を簡単に変換できる wstring_convert クラステンプレートが用意されました。

```
// #include <locale>

std::wstring_convert<
   std::codecvt<wchar_t, char, std::mbstate_t>>
cvt(new std::codecvt_byname<
   wchar_t, char, std::mbstate_t>("␣"));
```

簡単と言いつつ最初に出てくるコードは少々複雑に見えますが仕方がありません。

これを読み解くと，wstring_convert<> 型の変数 cvt の宣言であることが分かります。テンプレート引数もコンストラクタへの引数も省略の余地がないため，このように書かざるを得ませんでした。

なお，codecvt_byname のコンストラクタへの引数として""を指定しています。この文字列には，std::setlocale 関数や std::locale クラスのコンストラクタが引数に受け取るものと同じものを指定可能です。たとえば，Linux 環境であれば"ja_JP.eucJP"などといった指定が可能ではないかと思います。

wstring_convert<> 型の変数の作成さえ完了すれば，実際の変換処理の実行はきわめて簡単です。std::wstring への変換には from_bytes メンバ関数，std::string への変換には to_bytes メンバ関数を呼び出すだけです。

```
std::wstring araragi = cvt.from_bytes('A');
std::wstring tsukihi = cvt.from_bytes("月火");

std::string koyomi = cvt.to_bytes(L'暦');
std::string aryaryagi = cvt.to_bytes(L"アララギ");
```

from_bytes および to_bytes メンバ関数はそれぞれ 4 種類の多重定義があり，以下のような引数を取ります。

- char または wchar_t
- char const*または wchar_t const*
- const std::string&または const std::wstring&
- char const* first, char const* last または wchar_t const* first, wchar_t const* last

4.6 正規表現

C++ にも正規表現のライブラリが導入されました。

内容は Boost.Regex をベースとした API となっています。最初に std::regex オブジェクト（または std::wregex オブジェクト）を構築し，それを用いて，std::regex_match などの関数を使用するという使用方法になります。

4.6.1 一致判定

regex_match 関数は，文字列全体が正規表現のパターンと一致するか否かを判定する関数です。このほか，文字列中から正規表現で一致する部分を検索する regex_search 関数が存在します。

この例は，regex_match の多重定義の中で最も引数の数が少ないものを利用しています。グループキャプチャ（"x(y+)z"における括弧で囲まれた部分）の取得などが可能な他の多重定義も存在します。

```
// #include <regex>

std::regex meruado_kamo(".+@.+");
if (std::regex_match("hoge@example.jp", meruado_kamo))
{
   std::cout << "メルアドかも\n";
}
```

またまた余談となりますが，正規表現によるメールアドレスの完璧な判定は意外と大変です。注意しましょう。

参考：メールアドレスの正規表現
http://www.din.or.jp/~ohzaki/mail_regex.htm

4.6.2 置換

正規表現を用いた置換を行う関数として，regex_replace 関数があります。この例も，regex_replace の多重定義のうち最も引数の数が少ないものを使用しています。

```
// #include <regex>

std::regex last_part("^(?:.*/)+([^/]*)");
std::string src = "/usr/bin/cc";
std::string replace = "$1";
auto file = std::regex_replace(src, last_part, replace);
// file == "cc"
```

4.7 日時入出力

iostream に，tm 型を入出力するためのマニピュレータが追加されました。それぞれ std::put_time と std::get_time です。

```
// #include <ctime>
// #include <iomanip>

auto time = std::time(nullptr);
auto tm = std::localtime(&time);
std::cout.imbue(std::locale(""));
std::cout << std::put_time(tm, "%c") << std::endl;
```

put_time の 2 番目の引数では，strftime で使用できるものと同じ書式指定文字列を指定します。get_time も同様に書式文字列を指定して入力を行う作りとなっています。

4.7.1 日時入出力（それ Boost で）

参考までに，Boost.DateTime で同様の処理を行う例を紹介します。

```
ptime pt = second_clock::local_time();
std::locale loc(std::locale(""),
new time_facet<ptime, char>("%c"));
```

```
std::cout.imbue(loc);
std::cout << pt << std::endl;
```

5　並行処理関係

5.1　Atomic 演算

アトミック演算を実現する std::atomic クラステンプレートが追加されました。

Windows ではこのように書いていたものが（たびたび Windows での例で申し訳ありません），

```
long x;
InterlockedIncrement(&x);
InterlockedDecrement(&x);
auto old = InterlockedCompareExchange(
   &x, newValue, comparand);
```

このようになります。

```
std::atomic<int> y;
y++;
y--;
int old = newValue;
bool b = x.compare_exchange_strong(&old, comparand);
```

std::atomic クラステンプレート自体は，int 型以外でも使用可能です。また，四則演算やビット演算なども可能です。

5.2　非同期実行（スレッド）

スレッドを用いて非同期に実行する方法として，最も簡単（高水準）なのは，std::async 関数に std::launch::async を指定する方法です。

```
int hoge(std::string const& arg1, int arg2);

std::future<int> f = std::async(
```

```
    std::launch::async, hoge, "rofi", 3);

……
int result = f.get(); // 待機する
```

std::async(std::launch::async, ……) で，呼び出し元とは別のスレッドで hoge の実行が開始されます。このとき，戻り値を受け取るための std::future オブジェクトが返されるのでこれを受け取っておきます。

そして，f.get() で hoge からの戻り値を受け取ります。その時点で hoge の処理が実行中であれば，終了するまで待機します。なお，hoge が例外を投げた場合，f.get() からその例外が送出されます。その辺りの面倒をみてくれるのは，std::async が高水準なライブラリであるがゆえです。

5.3 非同期実行（現 Boost 風）

Boost.Thread 風の std::thread クラスも存在します。こちらは std::async 関数と比べればやや低水準に位置づけられます。

```
void g(……);
std::thread th(g, ……);
th.join(); // 待機する
```