C++1zに現状入る予定の言語拡張

高橋 晶(Akira Takahashi)
faithandbrave@gmail.com
2016/07/23(土) Boost.勉強会 #20 東京

自己紹介

- 高橋 晶(Akira Takahashi)
- Boost.勉強会 東京の主催者
- boostjp、cpprefjpサイトのコアメンバ
- 株式会社ロングゲート所属
- システム開発、ゲーム開発、教育のお仕事などを
  しています。
- 書籍：『C++テンプレートテクニック』『C++ポケット
  リファレンス』『プログラミングの魔導書シリーズ』など

C++1zとは

- 2017年の改訂を目指しているC++の次期バージョン
- C++17 (ISO/IEC 14882:2017) になる予定
- C++14に対するメジャーバージョンアップ

- この発表は、2016年6月のオウル会議までに採択された言語機能とライブラリ機能を紹介します
- この発表で紹介する内容は、正式な仕様になるまでに変更される可能性があります。

C++1zのスケジュール

- 2016年6月の会議で決まったところまでが、
  Committee Draft (CD)として公布される

- CDに対して数ヶ月間、各国代表の投票とコメントが
  募集される

- CDに大きな問題があれば出し直し。とくに問題なければ、
  Final Committee Draft (FCD)が公布される

- 順調に行けば2016年末にはほぼ仕様が固まり、
  その後は微修正だけになる

話すこと

- 言語機能
- ライブラリ機能
- 言語策定の体制と、議論への参加方法

Chapter 1

言語機能 - Core Language -

メッセージなしのstatic_assert

- これまでのstatic_assertは、表明失敗時のメッセージを指定する必要があった。C++1zではそれが省略できる

```
constexpr int a = 1;
constexpr int b = 1;
static_assert(a == b);                          // C++1z
static_assert(a == b, "a must equal to b"); // C++14
```

- メッセージを省略した場合、表明失敗した際に出力されるメッセージは未規定

単一要素の初期化子リストをTに推論

- 単一要素の初期化子リストを
  (代入構文ではなく)直接autoで受けた場合のルールが
  変更になる

```
auto a {1}; // C++14まではinitializer_list<int>
            // C++1zではint

auto b {1, 2}; // C++14まではinitializer_list<int>
               // C++1zではコンパイルエラー

auto c = {1};   // これまで通りinitializer_list<int>
auto d = {1, 2} // これもinitializer_list<int>
```

畳み込み式 (Fold expression)

- テンプレートパラメータパックに対して、
  任意の二項演算による集計処理ができる

```
template <class... Args>
int sum(Args... args)
{
  // パラメータパックの全ての要素を足し合わせる
  return (args + ...);
}

int result = sum(1, 2, 3, 4, 5);
assert(result == 15);
```

入れ子名前空間の定義

- 入れ子になった名前空間を、 :: 区切りで定義できる

```
namespace A::B {}                    // C++1z
namespace A { namespace B {}} // C++14
```

- 属性やインラインといった細かい指定はできない

__has_include

- インクルードするファイルが存在するかをプリプロセス時に確認する __has_include 命令が追加される

```
#if __has_include(<atomic>)
  #include <atomic>
#endif
```

[[fallthrough]]属性

- case節でbreakやreturnを書かないことが
  意図的であることをコンパイラに伝える

```
switch (n) {
  case 1:
    f();
    [[fallthrough]];
  case 2:
    g();
    break;
}
```

- 一番下のラベルには[[fallthrough]]属性は書けない

[[nodiscard]]属性

- 関数の戻り値をユーザーが無視してはならないことを
  コンパイラに伝える

```
// 失敗をハンドリングすることをユーザーに強制する
[[nodiscard]] bool f();

// 戻り値を無視すると警告が出力される可能性がある
f();
```

- この属性は、関数宣言、クラス宣言、列挙型宣言に対して
  使用できる

[[maybe_unused]]属性

- 使用しない可能性のある変数であることを、
  コンパイラに伝える

```
// パラメータaは使用しない可能性がある
void f([[maybe_unused]] int a)
{
  assert(a != 0);
}
```

- 警告の抑制が目的。
  シリアライズのversionパラメータとか。

constexprラムダ

- ラムダ式がconstexpr関数内で使用できるようになる

```
constexpr int f()
{
  auto f = [] { return 1; }; // OK
  return f();
}
```

- ラムダ式をconstexpr修飾すると、関数呼び出し演算子がconstexprになる関数オブジェクトが定義できる

```
int main() {
  auto f = [] (int x) constexpr { return x * x; };
  static_assert(f(2) == 4);
}
```

ラムダ式で*thisをコピーキャプチャ

- ラムダ式でthisをキャプチャすると、thisポインタがコピーされ、オブジェクトコピーされない。

- C++lzでは*thisをキャプチャすることで、*thisのオブジェクトがコピーキャプチャされる。

```
std::vector<std::function<void()>> taskList;
struct X {
  void g();
  void f() {
    taskList.push_back([*this] { g(); });
  }
};
```

- 非同期プログラムでは、レスポンスが返ってきたときには*thisの寿命が尽きていることがある

浮動小数点数の16進数リテラル

- 浮動小数点数のリテラルを16進数で表記できる
- 指数が書きやすくなる

```
// 2^-126の値
float x = 0x1.0p-126f;  // 16進数
float x = 1.17549e-38f; // 10進数
```

- printf関数の"%a"フォーマットやhexfloatマニピュレータがすでに標準ライブラリで16進表記に対応しているので、そのコア言語対応

クラステンプレートのテンプレート引数推論

- コンストラクタの引数から、クラステンプレートのテンプレート引数が推論されるようになる

```
mutex m;
lock_guard guard(m); // 本来はlock_guard<mutex>と入力する
```

- ヘルパ関数を定義する手間が減る

非型テンプレート引数のauto宣言

- テンプレートパラメータをautoとすることで、
  コンパイル時定数の整数やbool値を簡単に受け取れる

```
template <auto N>
struct X {
  static constexpr decltype(N) value = N;
};

X<3> {};
X<true> {};
X<'a'> {};
```

- template <class T, T N>の簡略化。

厳密な式の評価順序

- 式の評価順が厳密に規定される。

```
b @= a; // 代入または複合代入は右辺が先に評価される
        // @は任意の演算子


map<T, V> m;
m[0] = m.size(); // {0, 0}
```

```
// ※ 前のreplaceが終わってからfindが評価される
std::string s = "but I have heard it works even "
                "if you don't believe in it";
s.replace(0, 4, "")
 .replace(s.find("even"), 4, "only") // ※
 .replace(s.find(" don't"), 6, "");  // ※
```

値のコピー省略を保証

- RVOが言語的に保証される
- 単純な値を持つクラスは、一時オブジェクトのコピーが起こらないようになる

```
struct NonMovable { // コピーもムーブもできない型
  NonMovable(int);
  NonMovable(NonMovable&) = delete;
  void NonMovable(NonMovable&) = delete;
  std::array<int, 1024> arr;
};
```

```
NonMovable make();
auto nm = make(); // OK: make()の戻り値がそのままnmになる
```

if constexpr文

- if文の条件式をconstexpr修飾すると、
  その条件分岐はコンパイル時に行われる

```
template <class T, class... Rest>
void g(T&& p, Rest&&... rs) {
    if constexpr (sizeof...(rs) > 0)
      g(rs...); // rs...が空のときのオーバーロードが不要
}
```

- テンプレート内で条件分岐した場合、
  到達しなかったコードはインスタンス化されない

- 条件式内でのコンパイルの短絡評価はされないので注意
  `has_value_type_v<T> && typename T::value_type()`
  のような条件式は、elseのときコンパイルエラーになる
  (入れ子にすること)

インライン変数 1/2

- インライン関数と同様の指定を変数に対してもできるようになる。
- constexpr変数は自動的にインライン変数になる

```
struct Foo {};

struct X {
  // クラス定義の外でfooを定義する必要がない
  static inline Foo foo;
};
```

インライン変数 2/2

- インライン関数と同様の指定を変数に対してもできるようになる。
- constexpr変数は自動的にインライン変数になる

```
#ifndef FOO_HEADER
#define FOO_HEADER

struct Foo {};

// グローバル変数の定義をヘッダーに書ける。
// 複数の翻訳単位を跨いでひとつのオブジェクトになる
inline Foo theFooObject;

#endif
```

構造化束縛 1/2

- タプルやクラスを分解する言語機能。tieの代わり。
  ほかの言語では多重代入と呼ばれている。

```
tuple<int, string> f();

// aにはintの値、bにはstringの値が代入される
const auto [a, b] = f();
```

```
struct Point { int x, y; };
Point f();

// xにはPoint::xの値、yにはPoint::yの値が代入される
const auto [x, y] = f();
```

構造化束縛 2/2

- 範囲for文の変数宣言にも使用できる。

- CV修飾、参照修飾もできる。

```
std::map<int, string> m;

for (const auto& [key, value] : m) {}
```

```
// 配列の各要素を取得。
// 受け取る個数が配列の要素数と一致していなければならない
int ar[] = {3, 1, 4};
auto [a, b, c] = ar;
```

if文とswitch文の初期化と条件式を分離

- if文とswitch文に書けるのは条件式だけだったが、初期化を分けて書けるようになる

```
// セミコロンの前が初期化式
// セミコロンの後が(これまで通りの)条件式
if (status_code result = f(); result != SUCCESS) {
 // result変数を使用する
}
```

```
map<T, V> m;
if (auto p = m.try_emplace(key, value); !p.second) {
  // 挿入失敗
} else {
  process(*p.first);
}
```

Chapter 2

ライブラリ - Libraries -

並列アルゴリズム

- 標準アルゴリズムのほとんどに、並列実行のオプションが追加される

```
vector<int> v = ...

sort(v.begin(), v.end());       // これまで通りの順序実行
sort(seq, v.begin(), v.end()); // 明示的に順序実行を指定

sort(par, v.begin(), v.end());       // 並列実行を許可
sort(par_unseq, v.begin(), v.end()); // 並列and/orベクトル実行を許可
```

- accumulateはreduce、
  partial_sumは(inclusive | exclusive)_scanという名前になっている

- 実行ポリシーはそれぞれ別な型

ファイルシステム

- ファイルパス、ディレクトリ、ファイルの情報取得・変更などを扱うライブラリ。Boost.Filesystem v3ベース

```
namespace fs = std::filesystem;

fs::path from = "C:/a.txt";
fs::path to = "C:/b.txt";
fs::copy_file(from, to); // ファイルをコピー

fs::path name = from.filename(); // ファイル名を取得
fs::path ext = from.extension(); // 拡張子を取得(ドット含む)
```

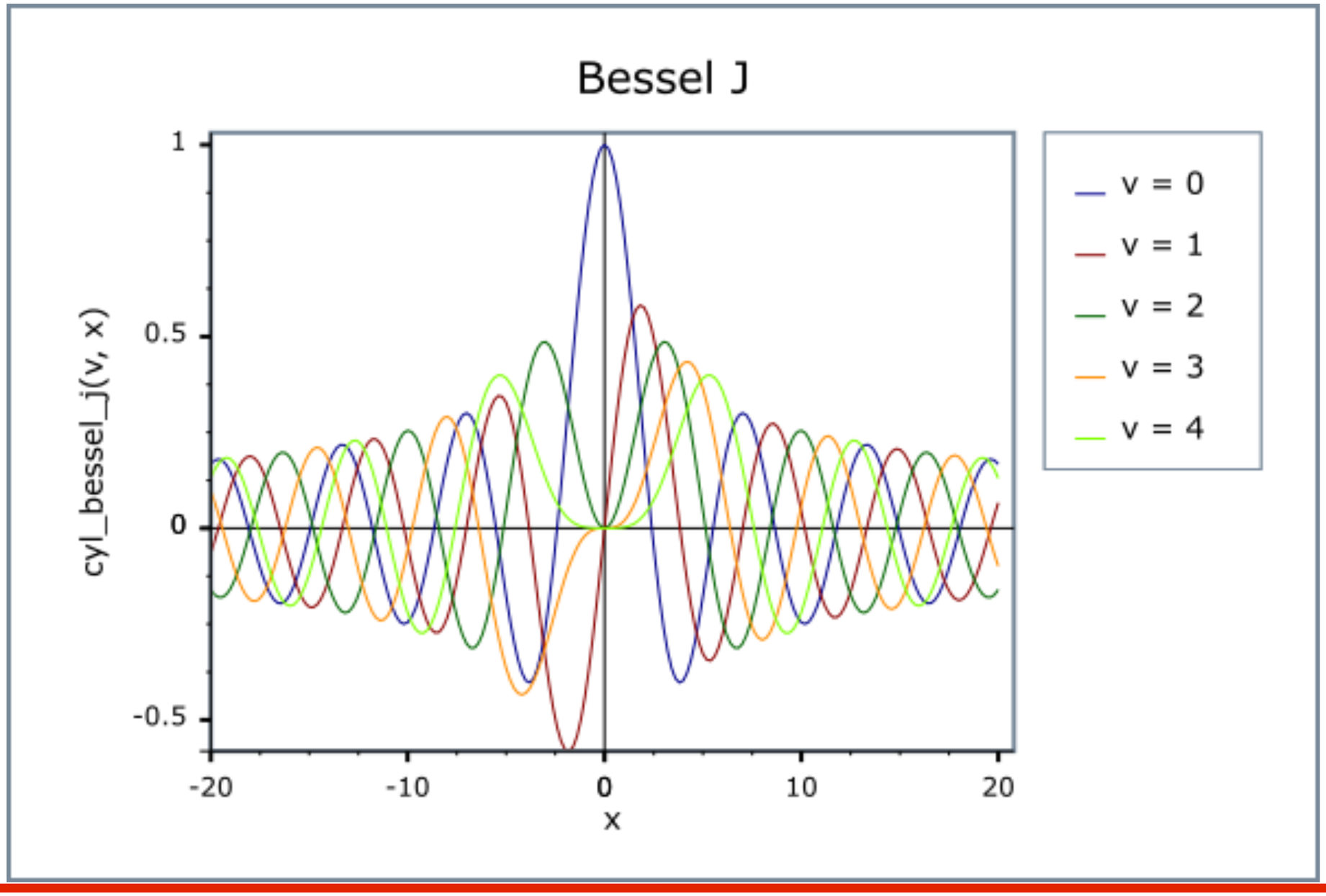
数学の特殊関数

- ガンマ関数、ベッセル関数、ベータ関数、楕円関数などが<cmath>に追加される

```
// 第一種ベッセル関数
for (float x = -20.0f; x <= 20.0f; x += 0.1f) {
  const float v = 0.0f;
  float result = cyl_bessel_j(v, x);
  cout << result << endl;
}
```

数学の特殊関数

- ガンマ関数、ベッセル関数、ベータ関数、楕円関数などが<cmath>に追加される



any

- あらゆる型のオブジェクトを代入できる型。
  Boost.Anyベース

```
// ひとつのanyオブジェクトに、
// いろいろな型のオブジェクトを代入できる
any a = 3;
a = std::string("hello");

// 中身を取り出す
std::string s = any_cast<std::string>(a);
```

optional

- 有効な値か無効状態のいずれかが代入される型。
  Boost.Optionalベース

```
optional<int> a = 3; // int型の値を代入 : 有効な状態
if (a) {             // 有効かどうかを確認
  int x = a.value(); // 有効値を取得
}

a = nullopt; // nulloptオブジェクトを代入 : 無効な状態
if (!a) {
  cout << "a is null" << endl;
}
```

variant

- テンプレート引数で指定した型候補であれば、
  なんでも代入できる型。型安全な共用体

```
// intかstringのどちらかが代入される
variant<int, string> v; // デフォルト構築 : int()

v = "hello"; // 文字列を代入
v = 3;       // 整数を代入

int x = get<int>(v); // 値を取得

// 型ごと、もしくは全ての候補型に共通の操作を行う
visit([](auto x) { cout << x << endl; });
```

string_view

- 文字配列を参照してbasic_stringのインタフェースを使用できるようにするクラス

```
const char s[] = "Hello World";

// char配列の部分文字列を取得
// 文字列のコピーは発生しない
string_view view = string_view(s).substr(6, 5);

// 部分文字列を取得 : 「World」が出力される
for_each(view.begin(), view.end(), [](char c) {
  cout << c;
});
```

文字列検索アルゴリズム

- std::search関数を拡張して、
  文字列検索のアルゴリズムが追加される

```
// ボイヤー-ムーア法で、文字列内の特定文字列を検索
std::string corpus = "…"; // 検索対象
std::string pattern = "…"; // 対象の中に見つけたい文字列
auto it = search (
  corpus.begin(),
  corpus.end(),
  make_boyer_moore_searcher(
    pattern.begin(),
    pattern.end()
  )
);
```

ランダムサンプリング

- 範囲からランダムにN個の要素を取り出すアルゴリズム

```
vector<int> input = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

const size_t N = 3;
vector<int> selected(N);

// 配列inputからN個の要素を抽出し、selectedに挿入
sample(input.cbegin(), input.cend(), selected.begin(), N);
```

タプルを展開して関数を呼び出すapply

- タプルの各要素を、関数の引数リストとして渡して呼び出す

```
void f(int, char, const string&) {}

int main()
{
  tuple<int, char, string> t {3, 'a', "hello"};

  // タプルtを展開して関数f()の引数にして呼び出す
  apply(f, t);
}
```

タプルからT型オブジェクトを構築する make_from_tuple関数

- pairのpiecewise_constructで行われているような
  タプルからT型への変換を、ユーザーが使えるようになる

```
// pairのコンストラクタ
template <class T1, class T2>
template <class... Args1, class... Args2>
pair<T1,T2>::pair(piecewise_construct_t,
                  tuple<Args1...> first_args,
                  tuple<Args2...> second_args)
    : first(make_from_tuple<T1>(first_args))
    , second(make_from_tuple<T2>(second_args))
{}
```

タプルからT型オブジェクトを構築する make_from_tuple関数

- pairのpiecewise_constructで行われているような
  タプルからT型への変換を、ユーザーが使えるようになる

```
struct Point {
  Point(int x, int y) {}
};

pair<Point, Point> p(piecewise_construct, {1, 2}, {3, 4});
```

値を範囲内に収めるclamp関数

- 値を、指定した最小値と最大値の範囲に丸める
- `std::min(std::max(min_value, x), max_value)`と同等

```
// vの値を[0, 2]の範囲内に収める
int v = 3;
int result = clamp(v, 0, 2); // result == 2
```

最大公約数と最小公倍数

- 最大公約数を求めるgcd関数、最小公倍数を求めるlcm関数

```
int x = gcd(12, 18); // x == 6
int y = lcm(2, 3);   // y == 6
```

述語を反転させるnot_fn

- これまであったnot1とnot2は引数の数が限られていた
- 新しく追加されるnot_fnはいくつでも引数を受け取れる

```
auto pred = not_fn([](int, char, string) {
  return true;
});

bool result = pred(1, 'a', "hello");
assert(!result);
```

shared_ptrに対するweak_ptr

- shared_ptrにtypedefとしてweak_typeを追加
- weak_ptrの型を取得できる

```
using ptr = shared_ptr<int>;
ptr p(new int(3));

ptr::weak_type w = p;
if (ptr sp = w.lock()) {
}
```

コンテナの要素情報にアクセスする非メンバ関数

- コンテナの要素数を取得するstd::size()関数
  コンテナが空か判定するstd::empty()関数
  コンテナの生データを取得するstd::data()関数
  が追加される

```
// 配列の要素数を取得する
int ar[] = {1, 2, 3};
constexpr size_t n = size(ar); // n == 3
```

- 配列とコンテナの共通インタフェースとして使用できる

コンテナに不完全型の最小サポートを追加

- vector、list、forward_listの要素型に不完全型を指定することが許可される

```
struct Entry {
  std::list<Entry> messages; // OK
  // …
};
```

- コンテナのなんらかのメンバ関数を呼び出す前に、要素型が完全型になっていること

多相アロケータ

- これまでのアロケータは、型ごとに異なるアロケータオブジェクトが必要だった。
- 多相アロケータは、異なる型同士でメモリリソースを一元管理する仕組み

```
std::pmr::synchronized_pool_resource mem_res;

// vとsでメモリリソースを共有する
std::pmr::vector<int> v(&mem_res);
std::pmr::string s(&mem_res);
```

- std::pmr名前空間で、多相アロケータを使用するコンテナの別名が定義される

連続イテレータ

- 連続イテレータ(contiguous iterator)という分類が追加される
- 標準ライブラリのコンテナが、連続したストレージを持っていることを保証するため
- C++1zでは、`bool`以外を要素とする`vector`、`basic_string`、`array`、`valarray`が連続イテレータを持つ

連想コンテナの接合(splice) 1/3

- mapとsetに、以下のメンバ関数が追加される
- extractで指定した要素を取り出し(元のコンテナから削除)、insertでほかのコンテナに要素を移す
- mergeはコンテナそのものを、ほかのコンテナにくっつける

```
node_type extract(const_iterator position);
node_type extract(const key_type& x);
iterator insert(node_type&& np);

template<class C2>
void merge(set<Key, C2, Allocator>& source);
template<class C2>
void merge(set<Key, C2, Allocator>&& source);
```

連想コンテナの接合(splice) 2/3

```
// mapの要素をほかのコンテナに移す
map<int, string> src {
  {1,"one"}, {2,"two"}, {3,"buckle my shoe"}
};
map<int, string> dst {{3,"three"}};

dst.insert(src.extract(src.find(1))); // イテレータ版
dst.insert(src.extract(2));           // キー版
auto r = dst.insert(src.extract(3));  // 重複

// src == {}
// dst == { “one” ,  “two” ,  “three” }
// r.position == dst.begin() + 2
// r.inserted == false
// r.node ==  “buckle my shoe”
```

連想コンテナの接合(splice) 3/3

```
set<int> src {1, 3, 5};
set<int> dst {2, 4, 5};
dst.merge(src); // srcをdstにマージ

// src == {5}
// dst == {1, 2, 3, 4, 5}
```

shared_mutex

- C++14で、タイムアウト機能付きのshared_timed_mutexが入ったので、タイムアウトなし版のshared_mutexが入る
- readers/writerパターンのミューテックス

低レイヤーの文字列・数値変換 1/2

- ロケールの状態に依存せず、フォーマット解析せず、
  動的メモリ確保せず、エラーハンドリングできる、
- 高速な文字列・数値間で変換をする関数が追加される

```
// 文字列から整数
from_chars_result from_chars(const char* begin,
                             const char* end,
                             Integral& value,
                             int base = 10);

// 文字列から浮動小数点数
from_chars_result from_chars(const char* begin,
                             const char* end,
                             FloatPoint& value,
                             chars_format fmt = chars_format::general);
```

低レイヤーの文字列・数値変換 2/2

- ロケールの状態に依存せず、フォーマット解析せず、
  動的メモリ確保せず、エラーハンドリングできる、
- 高速な文字列・数値間で変換をする関数が追加される

```
// 整数から文字列
to_chars_result to_chars(char* begin,
                         char* end,
                         Integral value,
                         int base = 10);

// 浮動小数点数から文字列
to_chars_result to_chars(char* begin,
                         char* end,
                         FloatingPoint value,
                         chars_format fmt,
                         int precision = 6);
```

型特性いくつか 1/4

- `template <class...> using void_t = void;`
- 型の式が有効かを確認するために使用できる

```
// 型Tがtypeという型を持っているかを判定する
template <class, class = void>
struct has_type_member
  : false_type { };

template <class T>
struct has_type_member<T, void_t<typename T::type>>
  : true_type { };

static_assert(has_type_member<int>::value == false);
```

- decltypeとdeclvalを使って、任意の式が有効かも判定できる

型特性いくつか 2/4

- is_callableとis_nothrow_callable
- 型Fが引数リストArgs...と戻り値の型Rで呼び出し可能かを判定する

```
// 関数オブジェクトFがintとdoubleを引数にとって、
// intを返す呼び出しが可能か
static_assert(is_callable<F(int, double), int> {});
```

型特性いくつか 3/4

- 型Tがswap可能かを判定するis_swappableとis_nothrow_swappable
- 型Tと型Uがswap可能かを判定するis_swappable_withとis_nothrow_swappable_with

型特性いくつか 4/4

- 型特性has_unique_object_representationsが追加される
- memcpy可能な型とその配列とかが該当する
- オブジェクトから自動的にハッシュ値を求めるために使用できる

constexpr assert

- assertマクロがconstexpr関数内で使用できるようになる
- NDEBUG定義時のみ有効

```
constexpr const T& operator[](unsigned i) const {
  assert(i < N);
  return data[i];
};
```

削除されるライブラリ機能

- 非推奨になっていたいくつかの機能がC++1zから削除される
  - `auto_ptr`
  - `random_shuffle`
  - `bind1st`, `bind2nd`関連
  - `unary_function`, `binary_function`
  - `ptr_fun`, `mem_fun`, `mem_fun_ref`関連
  - lostreamの古いエイリアス

非推奨化されるライブラリ機能 1/2

- C++1zでいくつかの機能が非推奨化される
  - not1(), not2()
  - unary_negate, binary_negate
  - 標準関数オブジェクトのメンバ型
    - result_type
    - argument_type
    - first_argument_type
    - second_argument_type

非推奨化されるライブラリ機能 2/2

- functionのアロケータサポート：仕様が不明確で、あまり実装されていなかった
- iterator
- allocator<void>特殊化
- allocatorのメンバいろいろ(allocator_traitsによって必要なくなったもの)
- raw_storage_iterator
- get_temporary_buffer()、return_temporary_buffer()
- is_literal_type
- memory_order_consume

ほかにもいろいろありますが

- 主要な更新は一通り紹介しました
- 更新内容は、cpprefjpサイトと私のブログで、随時紹介していきます
- http://cpprefjp.github.io/implementation-status.html
- https://github.com/cpprefjp/site/wiki/cpp17
- これらのページから、詳細情報にたどり着けるようにしてあります
- それと、https://isocpp.org はチェックしておきましょう

議論への参加方法

- https://isocpp.org/forums
- このページから、各議論場所を辿れます
- オープンな議論場所なので、誰でも参加できます
- 提案もしくは提案文書に対する意見は、Future Proposals メーリングリスト
- 仕様の確認は、Discussionメーリングリストに行きましょう
- 専門分野ごとのメーリングリストもあります

ドラフト仕様へのpull request

- https://github.com/cplusplus/draft
- 編集上の修正(typoや用語統一など)は、上記リポジトリにpull requestできます

C++17の次

- C++17の次は、C++20(2020)年を予定しています
- https://isocpp.org/std/status
- ネットワーク、並行コンテナ、モジュールシステム、
  コンセプト、Range、コルーチン、
  トランザクショナル・メモリなどが議論されています