C++20の概要
#1 言語機能編

高橋 晶 (Akira Takahashi)
faithandbrave@gmail.com
Preferred Networks, Inc.
2019/04/17 (水) C++ MIX #3

この話の目的

- 2020年中に策定予定のC++20の機能について共有します
- 概要なので詳細までは話しません
- この話をベースとして、より詳細な調査、議論、
  フィードバックに発展させてもらいたいです

C++20の簡単な説明

- ISO/IEC 14882:2020という規格になる予定の、2020年中に策定されるC++のバージョン
  - メジャーバージョンアップ・マイナーバージョンアップとかはない
- 言語機能の目玉は、契約、コンセプト、モジュール、コルーチン、三方比較演算子による比較演算子の自動定義
- ライブラリ機能の目玉は、サブシーケンスを参照するspan、カレンダーとタイムゾーン、Range、(文字列フォーマット)

1. 標準化作業への参加方法

標準化の体制

- C++ Standard Committee (C++標準化委員会, SC22/WG21)というISO標準化のワーキンググループで規格策定が行われている
- 標準化には、基本的にはいろいろな企業の代表が1名ずつ参加し、企業および国の代表として議論を交わす
  - 国によって制度がちがうかも
- 現在、代表はMicrosoftのHerb Sutter氏
  - 『C++ Coding Standard』『Exceptional C++』などの著者

標準化作業への正式な参加方法

- 日本のワーキンググループは3ヶ月に一度、オフライン会議を開催しており、提案文書や仕様案へのフィードバックを行っている
  - Googleの稲葉さん (kinaba)、サイボウズ・ラボの光成さん (herumi)などが参加している。主査はIBMの安室さん
  - 私も個人で参加していたが、日本の標準化団体の組織改編にともなって脱退した
- 国際標準化会議が4〜6ヶ月に一度行われており、そこで提案や各国コメントに対する議論が行われる
  - それに参加するには日本の国を代表して行くことになるので、許可を得るのがとても大変。参加経験のある近藤さん (redboltz) に相談するのがよい
- 提案する場合は、提案文書を書いて国際会議で発表しないといけないので、提案ハードルはすごく高い
  - std-proposalsメーリングリストとかで話をはじめて、どこかのタイミングでほかの人に提案を移譲するケースが多い

標準化作業へのカジュアルな参加方法

- std-proposalsメーリングリスト、std-discussionメーリングリストで、比較的カジュアルに問題点や要望の話ができる
- Working Draftや規格に対する欠陥レポートは、専用のメーリングリストに投稿する
  - https://isocpp.org/std/submit-issue
- 編集レベルの修正 (typoや用語統一など) は、GitHubリポジトリにPull Requestを送る
  - https://github.com/cplusplus/draft

標準化の流れ

- C++17からは3年ごとの定期更新になったので、
  仕様が固まった機能から順次、規格書に含めてリリース
- 以下の順に仕様書のステージが変わっていく (戻ることもある)
  - Working Draft (WD、何度も更新される) C++20はまだここ
  - Committee Draft (CD、ββ版みたいなもの、ここで各国投票)
  - Final Committee Draft (FCD、各国コメントを受けた修正版)
  - Draft International Standard (DIS、FCDに微修正して国際標準の準備がほぼできたもの)
  - Final Draft International Standard (FDIS、ほぼ最終版、スキップ化)
  - International Standard (IS、publish版)
- 機能グループごとに各国承認を得るTechnical Specification (TS) もあるが、それは省略

2. 言語機能

契約プログラミングサポート

- 契約に基づく設計 (Design by Contract, DbC) を言語サポート
- 事前条件 (expects)、事後条件 (ensures)、表明 (assert) を指定できる
- 契約レベルとしてdefault、audit、axiomがあり、リリースモード、デバッグモード、コメントのような扱いとなる (コンパイルオプションで有効にする契約レベルを選択できるようになるはず)

```
void f(int x, int y)
  [[expects: x>0]]          // 事前条件
  [[expects audit: y!=0]] // auditモードでのみ検査される事前条件
  [[ensures result: result > x+y]]; // 事後条件
                                    // 戻り値にresultと命名して条件式を記述
```

コンセプト 1/2

- テンプレートに制約を付けられる
  - テンプレートパラメータの型が満たすべき条件を定義できる
  - 型Tとそのオブジェクトxに対して、
    - f(x)という呼び出しができること
    - x.f() という呼び出しができること
    - T::value_type型を持っていること
    - 整数型であること、等値比較できること
    - などの制約を定義できる
- 制約による関数オーバーロードができる
  - InputIterator or ForwardIteratorでオーバーロードとか (例：std::advance())
- 要件を満たさなかった場合のコンパイルエラーメッセージがわかりやすくなる
- <type_traits>ヘッダの判定系メタ関数 (is_copy_constructibleとか) とenable_ifの置き換え
- <concepts>ヘッダで基本的な制約がライブラリ定義される

コンセプト 2/2

```
// 制約定義
template <class T>
concept Addable = requires (T x) { x + x; };

// テンプレートパラメータTに制約Addableを付ける。
template <Addable T>
T add(T a, T b) { return a + b; }

// 別の書き方
template <class T>
requires Addable<T>
T add(T a, T b) { return a + b; }

// 簡潔表記（template構文を省略）
Addable auto add(Addable auto a, Addable auto b) { return a + b; }
```

関数テンプレートの短縮構文

- ジェネリックラムダ (C++14) と同様に、関数のパラメータもautoとすることで、お手軽に関数テンプレートにできる
- 制約付き関数テンプレートの短縮構文から、コンセプトの指定を省いた形

```
// 制約なしの関数テンプレートは、パラメータをautoで書ける。
// autoはそれぞれ別な型になる
auto f(auto a, auto b) { return a + b; }
```

ジェネリックラムダのテンプレート構文

- C++14で、ラムダ式のパラメータ型をautoとし、関数テンプレートのラムダ式を定義できるようになった
- C++20では、テンプレート構文をラムダ式で書けるようになり、より細かい指定ができるようになる
  - 型に対する操作にdecltypeを介さなくてよくなる
  - 制約を指定できる

```
[] template <class T> (T x) -> T { return x + x; };
```

型の文脈での依存名に対するtypename省略を許可

- テンプレートパラメータTに依存する型名にtypenameを付けないといけなかった
- 型しか現れない文脈では、typenameを付けなくてもよくなる
  - (逆にC++11時点で、テンプレート外でもtypenameは付けてよい)

```
template<class T> T::R f(); // OK: 戻り値の型
template<class T> void f(T::R); // エラー: 変数テンプレートと曖昧

template<class T>
struct S {
  using Ptr = PtrTraits<T>::Ptr; // OK: 型定義
  T::R f(T::P p) { // OK: クラススコープでは変数テンプレートと曖昧にならない
    return static_cast<T::R>(p); // OK: 型を指定する文脈
  }
};
```

指示付き初期化 (designated initializers)

- C99の指示付き初期化を限定的にサポート
  - 配列のインデックス番号を指定する初期化はできない
- 集成体に対してのみ、メンバ変数名を指定して集成体初期化できる
- メンバ変数の宣言順に初期化しなければならない。一部省略できる

```
struct A { int x; int y; int z; };

A b {
   .x = 1,
   .z = 2
}; // b.yは0に初期化される
```

三方比較演算子 <=> 1/2

- 別名、宇宙船演算子 (spaceship operator, Perl)、
  一貫比較演算子 (consistent comparison operator)
- operator<=>をメンバ関数として定義しておくと、比較演算子が自動定義される
  - <, <= >, >=
  - (==, !=)
- メンバ変数の型がoperator<=>を定義していれば、それを包含する型にもoperator<=>が自動定義される
- strcomp(), memcmp()みたいに、小、等値、大かを、0未満、0、0超の値として一度に返す

三方比較演算子 <=> 2/2

```
struct X {
  int a; b; c;

  auto operator<=>(const X& x) const {
    if (auto cmp = a <=> x.a; cmp != 0) return cmp;
    if (auto cmp = b <=> x.b; cmp != 0) return cmp;
    return c <=> x.c;
  }
  // こう書いてもよい（デフォルト定義）
  // auto operator<=>(const X&) const = default;
};

X x, y;
if (x < y) {}
```

モジュール

- インクルード (と.h/.cpp分割) に変わる仕組みとして、
  モジュールを導入する。インクルードと併用できる
- コンパイル速度の向上が期待できる

```
// my_module.h
module;
#include <vector>
export module my_module;

namespace my {
  export class X {};
  export inline void f() {
    printf("Hello");
  }
}
```

```
import my_module;

int main() {
  my::X x;
  my::f();
}
```

コルーチン

- 関数実行を中断・再開する仕組みとしてコルーチンが導入される
- C#のawait / async構文由来で、co_await、co_yield、co_return キーワードを使用する
- 非同期処理が書きやすくなったり、遅延リストが作れたりする

```
my_generator<int> f() {
  co_yield 1; // 値1を生成して関数実行を中断（suspend）
  co_yield 2;
  co_yield 3;
}

for co_await (int x : f()) {
  cout << x << endl;
} // 関数f()の実行を再開
```

符号付き整数が2の補数表現であることを規定

- 最近のマシンは2の補数表現以外を使わず、MSVC、GCC、Clangもほかの表現をサポートしていないので、2の補数表現に規定する
- C11はほかの表現 (1の補数、符号ビット付き絶対値) を許可している
- 2の補数表現であることを前提にビット演算とかができる
- 値-0を表すビット表現がないので、符号付き整数型がstrong orderingになり、ハッシュ値が一意に決まる

```
std::int8_t x = 11;
assert(x == 0b00001011);

std::int8_t y = -x;
assert(y == (~x + 1)); // 負数は、ビット反転して+1した値
assert(y == static_cast<std::int8_t>(0b11110101));
```

入れ子名前空間定義でのインライン指定

- namespace A::B::C {} 構文で、各名前空間にインライン指定 (透明化指定) ができるようになる

```
namespace A::inline B::C {
}

// 以下と同じ
namespace A {
inline namespace B {
namespace C {
}}}
```

必ず定数式評価される関数

- consteval修飾した関数は、即実行可能な即時関数 (immediate function) となる
- constexprは、式の左辺がconstexprでないと定数式評価されないが、constevalを付けると、それと関係なく定数式評価される
- numeric_limitsのような (計算して) 定数を返す関数とかで使える

```
consteval int f() { return 3; }

static_assert(f() + 4 == 7);

template <int N> struct X{};
X<f()> x;
```

- また、関数が定数式評価されているか判定する関数として、<utility>にstd::is_constant_evaluated()が導入される

コンストラクタの条件付きexplicit

- コンストラクタのexplicitに、パラメータとしてbool定数式を指定することで、条件付きでexplicitにできるようになる
- tupleやpairで必要になる。オーバーロードでがんばらなくてよくなる
- 凝ったライブラリでもないと必要ない

```
struct X {
  explicit(true) X(int) {}
};
```

__VA_OPT__識別子

- 可変引数マクロで引数が空じゃなかったときに置き換えられるトークンを指定する機能
- ロギング用にオレオレprintfを作るのが簡単になる

```
// __VA_ARGS__が空じゃなかったら、msgと__VA_ARGS__の間にカンマが入る
#define LOG(msg, ...) printf(msg __VA_OPT__(,) __VA_ARGS__)

LOG("Hello");        // printf("Hello"); カンマがつかない
LOG("Hello %d", 3); // printf("Hello", 3);
```

空オブジェクトに対するヒントの属性

- EBO (Empty Base Optimization) をサポートするため、状態を持たないオブジェクトに[[no_unique_address]]を付けてコンパイラに伝えられるようにする

```
template <typename Key, typename Value, typename Hash,
          typename Pred, typename Allocator>
class hash_map {
  [[no_unique_address]] Hash hasher;      // これらは状態を持たない
  [[no_unique_address]] Pred pred;        // 最適化によって、メンバ
  [[no_unique_address]] Allocator alloc;  // 変数を0バイトにできる
  Bucket *buckets;
  // ...
public:
  // ...
};
```

分岐予測に対するヒントの属性

- 条件分岐で当たる可能性の高い・低いものをコンパイラに伝える属性として、[[likely]]と[[unlikely]]を追加する

```
if (is_success()) [[likely]] {
}
else [[unlikely]] {
}

switch (n) {
  [[likely]]   case 0: break;
  [[unlikely]] case 8: break;
}
```

本日はここまで

- 細かい言語機能や、各言語機能の詳細までは説明しきれません
- この発表をきっかけに、各自で調べて理解を深めてください
- 次回はライブラリ編をやります