C++のRanges Library

～難しい版～

tomolatoon @tomosann_tomo

Siv3D勉強会（2022/3/28）

https://siv3d.connpass.com/event/242313/

このスライドについて

- Siv3D勉強会はLT
  - それほど時間がない
  - そこまで難しいことを離せない
- というわけでボツになった分です
  - 文字が多い・長い
  - 扱う内容が難解
  - 最後の方内容を扱いきれていない

お約束

- ハイライト
  - C++20、C++23 を示すためにハイライトを用います
- コード
  - `using namespace std;` しています
  - `main` 関数の中だと考えてください
  - シンタックスハイライトは PowerSyntax を使っています

Ranges Libraryの機能・特徴

- Concepts & CPO ベースの設計
- Views
- Range Factories
- Range adaptors
- Range algorithms
- その他ユーティリティ

名前空間について

- 基本的に `std::ranges` 以下に定義
  - Range Factories と Range Adaptors は `std::ranges::views` 以下
  - `std::views` は `std::ranges::views` のエイリアス
- 詳しくは下を参照

```
namespace std {
        namespace ranges {
                namespace views {
                }
        }
        namespace views = ranges::views;
}
```

Concepts & CPO ベースの設計

- C++20 では Concepts が導入された
  - インスタンスが行える動作を決められる
- Ranges Library では意味論を表すため、積極的に使用
  - 既存の型も Range として使用可能に
  - よりわかりやすい実装、よりわかりやすいコンパイルエラー（？）

```
// 同じ型同士で足し算が出来ることを表すコンセプト
template <class T> concept addable = requires(T lhs, T rhs) { lhs + rhs; };

// 足し算が出来ない型を Concepts で弾ける
template <addable T> T add(T lhs, T rhs) { return lhs + rhs; }
```

Views

- 軽量な Range
  - コピーして扱える（コピー/ムーブ構築と破壊のコストは低い）
  - 多くは他の Range を参照するラッパー
- 操作を適用した Range を意味論的に表すことも
  - 操作は View がイテレートされるまで遅延
  - 大体の仕事を Iterator に投げることで実現

```
// view 自体が Range を持っている訳ではない（std::ranges::owning_view は例外）
std::vector v      = { 1, 2, 3, 4, 5 };
auto reverse_view = v | std::views::reverse;
```

Range Factories

- View として Range を生成するオブジェクト（views 以下に定義）
  - 多くは関数オブジェクト
- 現在あるもの
  - `views::empty`（サイズ 0 の Range を生成）
  - `views::single`（サイズ 1 の Range を生成）
  - `views::iota`（整数列の Range などを生成）
  - `views::istream`（std::basic_istream をソースにする Range を生

```
empty<T>; single(Value); iota(First); iota(First, Last); istream(Istream);
```

Range Adaptors (1/2)

- 任意の Range を元に View を生成する（views 以下に定義）
  - Range Adaptor Object を使用する
  - パイプライン記法で操作をチェイン出来る
  - 最終的に生成されるのは View なので操作は遅延
- 実装する側はちょっと面倒

```
std::vector v = {1, 2, 3, 4, 5};

// どちらとも {5, 4, 3, 2, 1} を表す
auto reversed_range_by_function = std::views::reverse(range);
auto reversed_range_by_pipeline = v | std::views::reverse;
```

Range Adaptors (2/2)

- Range Adaptor Object 同士で新しい操作を作れる
  - Range と操作を分けて記述できるので読みやすい
- 実装する側はとんでもない程面倒（C++23 で改善）

```
std::vector v = {1, 2, 3, 4, 5};

// 操作は変数に保存したり組み合わせたり出来る
auto transform = std::views::transform([](int32 e) { return e * e; });
auto reverse   = std::views::reverse;
auto transform_and_revrese = transform | reverse;

// どちらも {25, 16, 9, 4, 1} を表す
auto result1 = v | transform | reverse;
auto result2 = v | transform_and_revrese;
```

Range Adaptors など一覧 (1/3)

- `views::all`
  - Range の参照ラッパーな Viewを返す
- `views::filter(Function)`
  - Function が true を返す要素だけ残した View を返す
- `views::transform(Function)`
  - Range のそれぞれの要素に Function を適用した結果を要素とする View を返す
- `views::take(Integer)`
  - Range の先頭 Integer だけを取り出した View を返す
- `views::take_while(Function)`
  - Range の先頭から Function が false を返す前までを取り出した View を返す
- `views::drop(Integer)`
  - Range の先頭 Integer だけを取り除いた View を返す
- `views::drop_while(Function)`
  - Range の先頭から Function が false を返す前までの要素を取り除いた View を返す

Range Adaptors など一覧(2/3)

- `views::join`
  - Range が ネストしていればそれを平坦にした View を返す
- `views::join_with(Delimiter)`
  - トップレベルのネストを平坦にした時 Delimiter を挿入した上で、join のように平坦にした View を返す
- `views::split(Range, Delimiter)` （**NOT** Range Adaptor Objects）
  - 文字列に関して、Delimiter ごとに Range を分割した View を返す
- `views::lazy_split(Range, Delimiter)` （**NOT** Range Adaptor Objects）
  - 任意の Range に関して、Delimiter ごとに Range を分割した View を返す
- `views::common`
  - Iterator と Sentinel が同じ型になるような View を返す
- `views::reverse`
  - Range を逆順にした View を返す
- `views::elements<Integer>`
  - Range の各要素について、std::get<Integer> を適用した結果を要素とする View を返す

Range Adaptors など一覧(3/3)

- `views::adjacent<Integer>`
  - Range の先頭から末端の Integer – 1 個前までの要素について、その要素を起点とした Integer 個の要素の参照をタプルとして要素に持つ View を返す
- `views::adjacent_transform<Integer>(Function)`
  - Range の先頭から末端の Integer – 1 個前までの要素について、その要素を起点とした Integer 個の要素を引数にとる Function を適用した結果を要素とする View を返す
- `views::slide(Integer)`
  - Range の先頭から末端の Integer – 1 個前までの要素について、その要素を起点とした Integer 個の要素の参照をタプルとして要素に持つ View を返す
- `views::chunk(Integer)`
  - Range の先頭から Integer 個ごとに分割した Range を View とした要素として持つ View を返す
- `views::chunk_by(Function)`
  - Range の先頭から末端の 1 個前までの要素について、その要素と次の要素を引数に取る Function が false を返した所を切れ目として分割した Range を View とした要素として持つ View を返す
- `views::counted(Iterator, Integer)` （**NOT** Range Adaptor Objects）
  - Iterator を起点とした長さ Integer の View を返す
- `views::zip(Ranges…)` （**NOT** Range Adaptor Objects）
  - 複数の Range のインデックスが同じ要素への参照をタプルとして要素に持つ View を返す
- `views::zip_transform(Function, Ranges…)` （**NOT** Range Adaptor Objects）
  - 複数の Range のインデックスが同じ要素を引数に取る Function を適用した結果のオブジェクトを要素とする View を返す