コンテナ・文字列の中間インタフェース spanとstring_view

高橋 晶 (Akira Takahashi)
faithandbrave@gmail.com
Preferred Networks, Inc.
2024/06/14 (金) C++ MIX #11

新標準：コンテナと文字列の受け取り方

- C++20で入った`std::span`はコンテナを受け取るための型
- C++17で入った`std::string_view`は文字列を受け取るための型
- これらは日常的に使うユーティリティですが、罠もあります
- 一段階ラップすることで、より便利になるケースもあります
- 新たな当たり前となる作法を学んでいきましょう

span (C++20) 1/2

- spanは、メモリ連続性のあるコンテナ (vectorや配列) を共通して受け取るインタフェースとして使用できる型
- 所有権をもたず、先頭要素へのポインタとサイズだけをもつ

```
void process(span<int> data) {
  for (int x : data) { println("{}", x); }
}

vector<int> v = {1, 2, 3};
int ar[] = {4, 5, 6};

process(span<int>{v}); // 明示的な変換が必要
process(ar);
```

span (C++20) 2/2

- 要素のコピーが発生しないので、気軽に部分配列の操作ができる
- ただし、所有権をもっておらず (データは参照しているだけ)
  持ち運びには向かないので注意

```
void process(span<int> data) {
  // 一部配列を取り出したりしても要素のコピーが発生しない
  process_header(data.front());
  process_body(span{data.begin() + 1, data.end()});
}
```

string_view (C++17)

- string_viewは、文字配列とstringの共通インタフェース
- 文字配列に対してstringの便利なメンバ関数を使える
- 要素のコピーやメモリ確保が発生しないので
  部分文字列の操作も気軽にできる
- 所有権をもたないので、持ち運びにはあまり向かない
- 文字列リテラルはstaticな寿命をもつので、値として文字列リテラルを
  もつ場合は持ち運べる

```
void process(string_view sv) {
  println("{}", sv.substr(1, 3));
}

process("Hello");
process(string{"World"});
```

罠その1: 部分配列で末尾がずれる

- const char*で文字列を受け取る関数に部分文字列を渡すと
  切り抜いた範囲ではなく末尾まで渡ってしまう
- f(const char* s, int size)形式になっていないと困る
- そのような状況では、一旦stringに変換したりする必要がある
- もしくは、末尾を削れないよう安全にラップすることも考えられる

```
void f(const char* s) {
  cout << s << endl;
}

void process(string_view sv) {
  f(sv.substr(1, 3).data());
}
process("Hello"); //  「ell」を期待するが「ello」が出力される
```

罠その2: 持ち運びにくい

- spanとstring_viewは所有権をもたないので持ち運びにくい
- 関数の戻り値型にはしにくいし、メンバ変数としてもつのもむずかしい

```
class X {
  vector<int> _data;
public:
  // こういう使い方はOK
  span<int> f() {
    return {_data.begin() + 1, _data.end()};
  }

  // コンパイルは通るけど寿命切れ
  span<int> g() {
    vector<int> x = {1, 2, 3};
    return {x.begin() + 1, x.end()};
  }
};
```

所有権をoptional (省略可) にもたせる拡張

- string_viewは軽量だし、だいたいのデータは文字列リテラル (staticな寿命) だから、できればstring_viewで持ち運びたい
- だけどたまに文字列フォーマットした文字列を持たせたい
- こういう場合のために、所有権をもつオブジェクトをoptionalに持たせる拡張が考えられる

```
life_string_view f(life_string_view sv)
{ return sv.substr(1, 3); }

life_string_view g() {
  string s = "Hello";
  return f(life_string_view::allocate(move(s)));
}
```

所有権をoptional (省略可) にもたせる拡張

- shared_ptr<void>で所有権をいっしょに持ち運ぶ
- 所有権をもたさなければ空のshared_ptr (ポインタとatomic参照カウント) だけ余分にもつ

```
struct life_string_view {
  std::string_view _data;
  std::shared_ptr<void> _life;

  template <class T>
  static life_string_view allocate(T&& data) {
    auto p = std::make_shared<T>(std::forward<T>(data));
    return {*p, p};
  }

  …string_viewと同じメンバ関数を実装…
```

補足

- string_viewに所有権をもたせるアイディアは、async_mqttライブラリ (近藤貴俊さん作) で実際に使われています
  - https://github.com/redboltz/async_mqtt/blob/36bf0622a73d8a702d1fe729ab5e2b9e54ef088a/include/async_mqtt/util/buffer.hpp
- spanとstring_viewはC++の新常識として日常的に使われていくことになりますが、陥りやすい罠もあります
- 作法を身につけて、安全に使っていきましょう
- ＊ 配列を多次元配列としてアクセスできるようにするmdspanもあります (線形代数ライブラリで多用していくことになる)