C++20の整数

高橋 晶 (Akira Takahashi)
cpp_akira@Twitter
faithandbrave@GitHub
2021/08/15 (日) talk.cpp

本日のお話

- C++20では、みなさんが普段から使っている「整数型 (integral types)」がより便利になりました
- 整数がC++20でどう変わったのか、なにが変わらないのか、今後どう変わるのかをお話します

C++20での整数比較

C++17までの整数比較で困ったところ

- これまでは、整数比較は以下の点で使いにくかった
  - デフォルトで選択すべき整数型がint (符号付き整数)
  - しかし標準コンテナの要素数は符号なし整数
  - 両者の変数を比較すると「signedとunsignedの比較は安全ではない」という警告が出力される

```
vector<int> v = {1, 2, 3};
for (int i = 0; i < v.size(); ++i) { // 警告！
  ...
}
```

C++20での整数比較

- C++20では、整数比較の新しい方法が2つ用意される
  - コンテナの要素数を符号付き整数として取得するstd::ssize()関数
  - 符号付きと符号なしの整数を安全に比較するstd::cmp_less()関数

```
vector<int> v = {1, 2, 3};
for (int i = 0; i < ssize(v); ++i) {}           // OK

for (int i = 0; cmp_less(i, v.size()); ++i) {} // OK
```

std::ssize()関数

- コンテナの要素数を符号付き整数型に変換して返す
- <iterator>ヘッダで定義される
- 主に要素数を保持しておくために使用する

```
vector<int> v = {1, 2, 3};
auto n = ssize(v); // nの型は（だいたい）int
                   //（size_tを符号付きに変換した型）

int ar = {4, 5, 6};
auto m = ssize(ar); // 同様
```

std::cmp_less()関数

- 第1引数と第2引数のどちらに符号付き・符号なし整数が指定されたとしても、安全に小なり比較してくれる
- <utility>ヘッダで定義される
- ほかに、cmp_equal(), cmp_not_equal(), cmp_greater()など一通りの比較演算が揃っている

```
assert(cmp_less(2, 3u) == true);
assert(cmp_less(2u, 3) == true);
assert(cmp_less(2, 3) == true);
assert(cmp_less(2u, 3u) == true);
```

値が型のとりうる範囲内かを判定

- 地味にたいへんだったのが、「値Nがint型に収まるか」の判定
- 自前で書くとN < std::numeric_limits<int>::max() だが、これも符号なし・符号付き比較の警告問題がある
- <utility>ヘッダでstd::in_range()関数が定義される

```
assert(in_range<uint8_t>(-1) == false);
assert(in_range<uint8_t>(255) == true);
assert(in_range<int8_t>(255) == false);
```

C++20での整数規定

C++20では整数の内部表現、順序、一意性が規定される

- 符号付き整数の内部表現が「**2の補数表現**」に規定される
  - 負数を「絶対値をビット反転した値に+1」した値として表現する
- これは、**-0 (マイナスゼロ) を表すビット列がない**内部表現
  - 0の負数は0
- ただし、主要な処理系はこれまでも「2の補数表現」しか
  サポートしていなかったので、実装としてはこれまでと変わらない

全ての値に大小関係が成り立ち、ハッシュ値が一意に定まる

- 符号付き整数型の内部表現を「2の補数」に規定する理由は、
  - -0がないことで、**全ての値に大小関係が成り立つ**
    - -0 == 0になりえたが、そのようなことがなくなる
  - -0がないことで、**ハッシュ値が一意に定まる**
    - 異なる値-0と0を同じハッシュ値とする、のようなことがなくなる

```
struct X {
  int a;
  char b;
  // strong_ordering(全順序)で順序が返る
  auto operator<=>(const X&) = default;
};

auto ord = x <=> y;
if (ord == 0) { … }
```

```
struct X {
  int a;
  char b;
};

X x = …;
// 自動でハッシュ値を計算してくれる（将来）
size_t hash = hash_as_bytes(x);
```

ただし、符号付き整数がオーバーフローした際は未定義動作のまま

- 2の補数表現では加算し続けると負数、減算し続けると正数になることは自明だが、**折り返しは未定義動作のまま**
- これは、コンパイラの最適化の都合
  - コンパイラは、符号付き**整数が大きくなり続けても負数にならないことを期待**して最適化する
  - 折り返す動作が規定されると最適化が阻害されてしまうため、この規定は改定されない
- ただし、主要コンパイラは**折り返し動作を規定するオプション**を提供しているため、ユーザーは任意にこの最適化をオフにでき、折り返し動作を使用できる

ビット操作の強化

ビット操作のための<bit>ヘッダ新設

- 以下のような機能が定義される
  - 内部表現が同じ別な型への変換を行うbit_cast()関数
  - 2の累乗値かの判定、2の累乗値への切り上げ、切り下げ
  - 循環ビットシフト
  - pop count (立っているビット数)、連続した0 or 1の数

```
float f = 3.14f;
// memcpy()のconstexpr版のようなもの
auto a = bit_cast<uint32_t>(f);

// 2の累乗関係
auto b = bit_ceil(127u);  // 128
auto c = bit_floor(129u); // 128
```

```
// 循環シフト
auto i = static_cast<uint8_t>(0b0000'0001u);
std::uint8_t a = rotr(i, 3);
assert(a == 0b0010'0000u);

// 立っているビット数
auto i = static_cast<uint8_t>(0b1000'1010u);
int b = popcount(i); // 3
```

整数の今後

- C++23で起こること
  - (符号付き) size_t のリテラル**z**
- C++の将来で今後起こりえること
  - 128ビット整数のサポート (ただしターゲット環境依存)
  - 多倍長整数のサポート
  - オーバーフローチェック付きの演算
  - オブジェクトのハッシュ値を自動計算してくれる機構
  - ビット数指定のリテラル

まとめ

- operator<=> (三方比較演算子) の詳細は、cpprefjpサイトかonihusubeさんのブログを参照してください！
- 整数を安全に比較できるようになりました
- 符号付き整数の内部表現、順序、一意性が規定されました
  - ただし、未定義動作のままになっているものもあります
- 整数のいまとこれからを、ご注目ください