C++ Core Guidelines の紹介

- talk.cpp 2021/08/15

C++ Core Guidelines

C++の創始者 Bjarne Stroustrup が主導してまとめている
コーディングガイドライン

- https://github.com/isocpp/CppCoreGuidelines
- 全項目のタイトルの日本語訳:
  https://qiita.com/tetsurom/items/322c7b58cddaada861ff

現代的なC++による効率的なコーディングをサポートすることを目的としている。

コードのレイアウトではなく設計のガイドライン。

ガイドラインに沿っているかコンパイル時にチェックする機能も開発されている(例:MSVC)。

ルールの分類

C++ Core Guidelines には非常に多くのルールがあり、いくつかのカテゴリーに分類されている。

主なカテゴリー

- I: インターフェース
- F: 関数
- C: クラスとクラス階層
- R: リソース管理
- ES: 式と文
- T: テンプレートとジェネリックプログラミング

オブジェクトの構築と破棄に関するルール

オブジェクトの動的確保には `new` ではなく `make_unique()` または `make_shared()` を使用する。

- `malloc()` と `free()` を避ける (R.10)
- 明示的な `new` と `delete` を避ける (R.11)
- `unique_ptr` や `shared_ptr` を使用してオブジェクトを確実に解放する (C.149)
- `unique_ptr` で保持するオブジェクトは `make_unique()` で作る (C.150)
- `shared_ptr` で保持するオブジェクトは `make_shared()` で作る (C.151)

所有権に関するルール

所有権を表すには `unique_ptr` または `shared_ptr` を使用し、生ポインターや参照を使わない。

- 生ポインターで所有権を表さない (R.3)
- 参照で所有権を表さない (R.4)
- 所有権を表すのに `unique_ptr` か `shared_ptr` を使う (R.20)
- 所有権の共有が必要でない限り `shared_ptr` より `unique_ptr` を使う (R.21)
- 決して生ポインターや参照で所有権を渡さない (I.11)
- ポインターによる所有権の移動には `unique_ptr<T>` を使用する (F.26)
- 所有権の共有には `shared_ptr<T>` を使用する (F.27)

引数の渡し方

- 入力用引数は、安価にコピーできる型は値で渡し、それ以外はconst参照で渡す (F.16)
- デフォルト引数とオーバーロードを選べるときはデフォルト引数を使う (F.51)
- 配列をポインター1つだけで渡さない (I.13)
- 引数の型に `[]` をつけるのを避け、`span` を使う (R.14)
  「長さとポインター」スタイルではなく、1つの引数で渡す。
- 関数が `T` の所有権を取る場合は `unique_ptr<T>` 型の引数を取る (R.32)
- 関数が `T` を置き換える場合は `unique_ptr<T>&` 型の引数を取る (R.33)
  `unique_ptr<T>` を対象とした出力用引数、入出力用引数の場合。
- 関数が `widget` を共同所有する場合は `shared_ptr<widget>` 型の引数を取る (R.34)

値の戻し方

- 値を返すには、出力用引数よりも戻り値を使う (F.20)
  - ムーブも遅い場合を除く
- 複数の値を返すには、タプルか構造体を使う (F.21)
  - 構造化束縛や `tie` で受け取ると良い
- 位置を知らせる場合に限りポインターを返す (F.42)
- コピーが望ましくなく、かつ無効値が必要なければ参照を返す (F.44)
  - `nullptr` が必要ならポインター
- 右辺値参照 `T&&` を返さない (F.45)
- ローカル変数をmoveして返さない (F.48)
  - 書く必要がないだけでなく、戻り値最適化を阻害してしまう

関数への入出力のまとめ (F.15)

- 入力の場合 (F.16)
  - コピー禁止、またはコストが小さい ... 値 `f(X)`
  - それ以外 ... const 参照 `f(const X&)`
- 出力の場合 (F.20)
  - ムーブのコストが大きい ... 非 const 参照(出力引数) `f(X&)`
    長い `std::array` など、型自体が大きい場合、ムーブしてもコピーと変わらない
  - それ以外 ... 値 `X f()`
- 入出力(兼用)の場合 (F.17)
  - 参照 `f(X&)`

コストが小さい: キャッシュされている(hot) `int` 5個くらいのコピー
コストが中程度: 1KB以内の連続している、またはキャッシュされている(hot)メモリーのコピー
コストが大きい: `sizeof` が大きい型(長い `std::array` など)のコピー

クラス設計

- クラスが不変条件を持つなら `class` を、メンバーが独立に変更できるなら `struct` を使う (C.2)
- publicではないメンバーがあるなら構造体ではなくクラスを使う (C.8)
- 関数はクラス内部に直接アクセスする必要がある場合に限りメンバーにする (C.4)
  メンバー関数よりもフリー関数の方が拡張性が高く、汎用的である。
- クラス階層よりも具象クラスを使う (C.10)
  クラス階層は必然性があるときだけ作る。その方がシンプルで設計しやすく利用しやすい。
  ポリモーフィックなクラスは仮想関数呼び出しによるコストがかかる。
- データメンバーを `const` または参照にしない (C.12)
  これらがあるとコピーできなくなり、不便である。

アクセス制御

- 自明なゲッター/セッターを避ける (C.131)
  それよりも `struct` にしたほうが良い。
- すべての非constなメンバー変数を同じアクセスレベルにする (C.134)
  ただのデータの集まり(`struct`)か、そうでないか、クラスの役割がわかりやすくなる。

クラス階層

- 仮想関数には `virtual` , `override` , `final` のうち1つだけを必ず指定する (C.128)
- 多相クラスのディープコピーには、コピーコンストラクタ/代入よりも仮想clone関数を使う (C.130)
- 訳もなく仮想関数にしない (C.132)
- データメンバーを `protected` にしない (C.133)

その他いろいろなルール

- 符号ありと符号なしの計算を混在させない (ES.100)
  STLで、`size()` を符号なしにしたのは失敗だった。(今更言われても……)
- テンプレートメタプログラミングは本当に必要なときだけにする (T.120)
- 他のコンテナを使う理由がなければまずは `vector` を使う (SL.con.2)
- 文字列を参照するには `string_view` や `span` を使う (SL.str.2)
- 文字列リテラルにユーザー定義リテラル `s` を付けて `std::string` であることを示す (SL.str.12)
- `endl` を避ける (SL.io.50)
  `flush` する必要はほとんどないので、改行文字を出力するだけで良い。
- 列挙型の名前を全て大文字の名前にしない (Enum.5)
- すべて大文字の名前はマクロの名前だけに使う (NL.9)

GSL (ガイドラインサポートライブラリ)

C++規格に足りていないボキャブラリー型などをいくつか提供している。
GSLを使うことが前提のルールも複数存在する。

- `gsl::not_null<T*>` : nullにならないことを表明するポインターのラッパー (assertあり)
  `not_null<T*>` 、 `not_null<unique_ptr<T>>` のように使用する。
- `gsl::owner<T*>` : 所有権を持っていることを表明するポインターのラッパー
  スマートポインターが使えない時はこれを使う。
- `zstring` / `czstring` : C文字列を指す `char*` / `const char*`

対応する項目:

- nullになってはいけないポインターは、`not_null` として宣言する (I.12)
- 1つのオブジェクトを指すのに `T*` か `owner<T*>` を使う (F.22)
- C文字列を指すには `zstring` か `not_null<zstring>` を使う (F.25)

まとめ

自明な項目も多いが、当たり前だと思えたらモダンC++に慣れてきた証拠

👇 全項目の見出しの日本語訳も作成しています 👇
https://qiita.com/tetsurom/items/322c7b58cddaada861ff