C++ プログラミング診断室

いなむのみたま　著

2019-07-11 版　　発行

はじめに

本書はプログラミングを始めたばかりの人を対象としません。初心者を脱し、その先へと進むプログラマへのメッセージをまとめたものです。

本書では特にことわりがない限り C++17 を前提とします。

本書はそうした技法について解説するための本ではありません。どうすれば安全かつキレイなコードを書くことができるのかにフォーカスしています。

仕事をしている中で「もっとこうしたほうがいい」というアドバイスをすることや、「この機能はこういう仕組みなんだよ」と解説をする機会が多くあります。

幾度も説明するうちに、本でも書いたほうがいいのではないかと思い、筆をとった次第です。

軽い自己紹介

僕は「いなむのみたま」と申します。

とあるベンチャー企業に雇われて C++ やら Rust やらを書いて一年が経ちました。

僕の得意分野はライブラリプログラミングです。ライブラリを作るためには、アプリケーションだけを作るなら一生使わないような C++ の機能を山のように使うことになります。なので、一般の C++ プログラマより少しだけ C++ に詳しいです。

いなむのみたまのかみ (@mitama_rs) という名前で Twitter に生息しており、Twitter で迷える C++ プログラマを救う活動をしています。

まえおき

ジョブチェンジしたての C++ プログラマに立ちはだかる苦悩:

- スマートポインタ使ってないどころか、new じゃなくて malloc を使うソースコード
- 完全下位互換 STL ともいうべき、オレオレ実装
- 3 万を超えるヘッダ
- 意味不明な過去の遺産
- 「template は難しいからできるだけ使うな」とか言う人
- 全然自動化されてない CMakeFile

最後の 3 つ以外はすべて Opencascade Technology というライブラリが原因です。

C++ に限らず、汚いコードは非常に汚いです。「もうお前 C++ 書くな」と言いたいこともあります。

汚いコードを書いているプログラマは決してサボっているわけではありません。それどころか、僕よりも何倍もの時間コードを書いているように見えます。ライブラリを使えばいい処理を手で書いたり、同じような少し違う処理をたくさん書いたり、意味のない条件分岐を書いたり、使わない変数を書いたり、とにかくありとあらゆる間違った技法を凝らしてコーティングを複雑にしているのです。

CMakeFile をキレイにしたり、テストを充実させるのに当てる時間もありません。実装で手一杯なのですから。

幸いにもベンチャー企業はとても動きやすかったため、すでにあったクソコードをあらかた焼き尽くすことに成功しました。ぶっちゃけると本書は、そのときの僕が流した涙やついたため息をまとめたものです。

目次

目次

第 1 章

ｱｯ!この分岐「深い」ッ！！

1.1　まえおき

読者も分岐が異常に複雑でネストが深いコードを目撃したことがあるのではないだろうか？ コントロールフローの複雑なコードはダメなコードの典型例と言っても過言ではないであろう。世の中のコードにはこういうものがあふれているのではないかとすら思う。

複雑な処理を行わなければいけない場合、コードが複雑になるのは仕方がない。しかし、あまりにも多くの場合分けを一つの関数に押し込めるのは間違いだと思う。

この章では、複雑な分岐をいかに簡潔にコーディングし、関数をどう分離するのかについて議論する。

1.2　分岐乱舞

一生通ることのない分岐を書くプログラマがいる。

典型例は

- 非負整数の負数チェック
- すでに Null 検査が済んでいるポインタのダブルチェック

などである。

非負整数の負数チェック

```
// sizeof は std::size_t を返すので絶対に真になることはない
if constexpr (sizeof(foo) < 0) {
  // ...
}
```

すでに Null 検査が済んでいるポインタのダブルチェック

```
int* ptr
assert(ptr); // 一回目のチェック
if (ptr) { // assert するかハンドルするかどっちかにしろ
  // ...
} else {
  // ...
}
```

非負整数の負数チェックの場合は修正が簡単で、単に取り除けばいい。

ポインタのダブルチェックの場合、assert するのが正しいのかハンドリングするのがわからない。そのため、git blame を行い書いた人を問い詰めるか、自分でコードを読んで考えるしかない。

1.3 正常系と異常系

正常系と異常系が分けて書かれていないというのはコードのわかりにくさの原因のひとつになる。先に異常系のチェックを行い異常値を返し、以降は正常系の処理を行うというのはよく知られた方法である。

新人が最初に書いた最初のプルリクエストで switch と if を駆使したコードを見せてくれた。残念ながら、紙面を圧迫するほど複雑なので疑似コード

の掲載を断念せざるを得ない。

この問題へのより良い解答は、モナドを導入することだと思われる。以下は Rust のコードである。コマンドの入力をパースして数字であれば 2 倍にして返す関数を書いてみる。match 式を使って分岐をする関数と Result のモナド機能を駆使して書いた関数の 2 つを見比べてほしい。

match 式で書いたコマンドラインのパース

```
fn twice_arg(mut argv: env::Args) -> Result<i32, String> {
    match argv.nth(1) {
        None => Err("数字を１つ指定してください。".to_owned()),
        Some(arg1) => {
            match arg1.parse::<i32>() {
                Ok(n) => Ok(2 * n),
                Err(err) => Err(err.to_string()),
            }
        }
    }
}
```

Result のモナド機能を駆使して書いたコマンドラインのパース

```
fn twice_arg(mut argv: env::Args) -> Result<i32, CliError> {
    argv.nth(1)
        .ok_or(CliError::NotEnoughArgs)?
        .parse::<i32>()
        .map(|x| x * 2)
}
```

後者の関数がエラー処理を書いてないのがおわかりですか？ Rust の詳細な解説は省きますが、モナディック関数を使うとエラー処理をわざわざ全部書かなくて済みます。エラーを合成するのです！

ok_or や map のように Option や Result を合成する関数はコンビネー

タと呼ばれます。コンビネータ指向でプログラミングをすれば、エラー処理は簡潔になり得ます。

残念ながら C++ にこのような高級な機能やライブラリは備わっていないのです。したがって、モナドライブラリを作ればよいのです。

テンプレートライブラリを書く時間がない？ しかし、コンビネータは「失敗する可能性がある単位」で関数を分離するという重要な示唆を与えてくれます。すくなくとも、エラーが起こる単位で関数に切り出すことを心がけたいところです。

1.4 この章のまとめ

モナドがない言語が何をやってもダメ。モナディック関数を作れ。

何種類ものエラーが起こるような関数を書かない。エラーが起こる最小単位くらいで関数を切り分ける。

第 2 章

無駄なコードを減らす

コードは書けば書くほどバグが混入する。無駄なコードの重複や複雑性を回避し、シンプルで美しいコードを維持することが設計の成功の鍵である。

2.1 コピペ乱舞

無駄なコードを減らすための最初の方針は共通の処理を関数として共通化することである。悲しいかな、世の中には同じコードこコピペによって増殖させてしまったコードがたくさんある。そのようなコードは一部を変更すると他の変更が漏れ、容易にバグの原因となる。

そもそもの話、自分でコードを書かずにある程度信頼できるライブラリを使うことを考えるべきである。この章では、C++ の標準ライブラリが提供するカスタマイゼーションポイントを活用する方法をいくつか紹介します。

2.2 カスタマイゼーションポイント

多くの言語では文字列化やオブジェクトの比較など、よくある操作をカスタマイズする方法は確立されています。

カスタマイゼーションポイントとは、特定の関数等の動作が呼び出してい

る関数だと思ってください。この関数を定義しておくことで、動作をカスタマイズすることができます。

C++ で主に使われるカスタマイゼーションポイントについて解説します。

Stringifiying

Java や C#では以下のようにすると上手くいきます。しかし、C++ ではプリミティブはメンバ関数を持てないので上手くいきません。

Java でありそうなコード

```
class Widget {
    Gadget a, b;
public:
    std::string toString() const {
        return a.toString() + " " + b.toString();
    }
};
```

C++17 までは以下のように std::ostream に出力する operator<<をカスタマイゼーションポイントに使うことが多いです。std::stringstream に出力して str() で文字列を取り出すのです。C++20 から文字列フォーマットが入りますので期待しましょう。

```
class Widget {
    int a, b;
public:
    friend std::ostream&
    operator<<(std::ostream& os, const Widget& w) const {
        return os << a << " " << b;
    }
};
```

比較

比較のカスタマイゼーションポイントは 2 通りあります。

方法 1: 演算子のオーバーロード

C++20 から三方比較演算子 operator<=> を定義するとすべての比較演算子が自動実装されるのですが、C++17 まではすべてを自分で実装するしかありません。==，!=，<，>，<=，>=の 6 種類の演算子を適切にオーバーロードしましょう。

方法 2: CPO（カスタマイゼーションポイントオブジェクト）

- クラスの場合

std::map や std::set は、テンプレートパラメータとして比較演算を提供するクラスを指定できます。

第 3 テンプレート引数がカスタマイゼーションポイントオブジェクトになっています。std::less を特殊化するか、自分で作ったクラスを渡してあげることで std::map のキー比較をカスタマイズすることが可能です。

std::map のを降順にカスタマイズ

```
#include <map>
#include <functional>

int main() {
    std::map<std::string, int, std::greater<>> dict{};
}
```

このように、カスタマイゼーションポイントをクラスから分離しています。何も書かなければデフォルト実装が選択されます。ユーザーが明示的にクラスを指定することで実装の詳細をユーザーが静的にインジェクションすることができるのです。このカスタマイゼーションポイントの設計はポリ

シーと呼ばれています（次章で解説します)。

- 関数の場合

比較を伴うアルゴリズムの関数はカスタマイゼーションポイントオブジェクトを受け取るようになっています。std::sort,std::min,std::max の第 3 引数などがそれです。

カスタマイゼーションポイントを関数から分離して引数にしています。何も書かなければデフォルト実装が選択されます。関数オブジェクトを渡すことでカスタマイズが可能であり、よくあるカスタマイゼーションポイントのもたせ方です。

std::sort のカスタマイズ

```
#include <algorithm>
#include <vector>

int main() {
    std::vector<std::pair<int, int>> vec{ {1 ,2}, {2, 3}, {3, 4} };
    std::sort(vec.begin(), vec.end(), [](auto&& p1, auto&& p2){
        return p1.second < p2.second;
    });
}
```

2.3　この章のまとめ

- コードをしっかり共通化する
- そもそも自分でコードを書かずにライブラリを探す
- 標準ライブラリのカスタマイゼーションポイントを利用する

第 3 章

賢い継承の使い方

継承は便利だが、使い方を誤るとひどいコードを生み出しがちである。

3.1 継承を使いこなせない

初心者ならわかりますが、仕事で C++ を書いている人でも継承が使いこなせないプログラマはいます。

具体的には以下のようなコードを書いてしまいます:

- final 指定をしないクラスが virtual なデストラクタを持たない
- override 指定しない virtual 関数を書く
- 継承関係が複雑すぎる

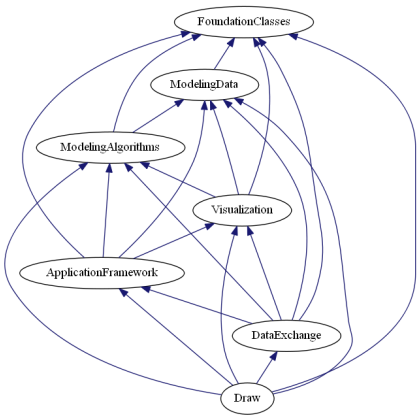

図 3.1: 複雑な継承の例

3.2 継承乱舞

仕事をしていて基底クラスを継承した派生クラスが数十個あるコードを見たことがあります。

これは俗に言う「神クラス」というアンチパターンです。共通に使う関数を基底クラスで純粋仮想関数として設計するというやり方です。

ちょっと考えるとわかりますが、この方法では基底クラスと派生クラスは密結合であり、非常に変更がしづらいのです。「神クラス」の問題を解決する前に、コンポジションについて説明します。

3.3 コンポジション

コンポジションを簡単に説明すると、クラスに機能を埋め込む設計です。メンバとしてなにかを持つというのがコンポジションのように語られているとこが多いです。C++ の場合、テンプレートの引数として関数を持つ

クラスを指定する Policy クラスという方法が一般的に用いられています。Policy クラスもコンポジションの一種です。

第 2 章で紹介した std::map でもこの手法が用いられています。

```
namespace std {
    template <
        class Key,
        class T,
        class Compare = less<Key>,
        class Allocator = allocator<pair<const Key, T> >
    >
    class map;
}
```

Compare や Allocator はメンバ map が変数としてオブジェクトを持っています（EBO のために継承になっているかもしれません）。テンプレートにカスタマイズされた比較クラスやアロケータクラスを指定し、コンストラクタにインスタンスを渡します。こうすることで、継承を使わなくても異なるクラスの動作を制御できました！

型が違うから同じコンテナに格納したりできへんやんけ！

そう、テンプレートのコンポジションでカスタマイズされたクラスは型が違うのです。ちょっとアロケータが違うだけでもです。アロケータが違うだけで別のコンテナに代入できないなんて使いにくいですよね？

3.4　局所的動的多相

型が違う、ならば動的多相なアロケータを使えばええやろ！

クラスそのものを派生クラスにせずに動的多相な Policy クラスを使う。

コンテナの場合、例えば `std::vector<T, Allocator>` の `Allocator` を動的多相化すれば、問題は解決します。

polymorphic_allocator という動的多相なアロケータクラスが C++17 の<polymorphic_resource>というヘッダに入りました。pmr という名前空間にエイリアス宣言されたコンテナはアロケータのクラスに polymorphic_allocator を指定しています。

```
namespace std {
  // C++17 から
  namespace pmr {
    template <class T>
      using vector = std::vector<T, polymorphic_allocator<T>>;
  }
}
```

神クラスを継承するのではありません。機能を分解し、ひとつひとつを動的多相化すべきなのです。

独立できる単位でポリシークラスに切り分けるべきです。なんなら人間クラスに足クラスと腕クラスくらいの細かいポリシーがあっていいです。義手と生体の腕だと機能が違うでしょうし。

3.5　この章のまとめ

- 機能を小さく分解する
- 継承を局所化してポリシーにする
- 最小の型制約を意識する

第 4 章

脱初心者に襲いかかる C++ の罠

やっと C++ 初心者を脱出し、もっと良いコードを目指すあなた。そんなあなたに襲いかかるのは C++ の難解な仕様の数々です。

この章では、ちょっと高級な機能に手を出し始めた脱初心者が陥る罠の数々を解説します。

4.1 Forwarding Reference

Forwarding Reference とは右辺値と左辺値のオーバーロードが同時に扱えるという夢のような挙動を実現する機能です。その特性から、Forwarding Reference を使ったオーバーロードはあらゆる修飾に対応するのでオーバーロードに選択されやすいという事実があります。

Forwarding Reference で引数をとる関数は意図しない呼び出しを避けるため、SFINAE で型制約をかけるべきです。しかし、SFINAE は難しいです。難しいので、脱初心者段階では Forwarding Reference に型制約を書くことができないのです。

そうして次のようなコードが生まれます。

```
template<typename T>
void f(const std::vector<T>&) { std::cout << "vector<T>"; }

template<typename T>
void f(T&&) { std::cout << "T&&"; }
```

この上の関数 f(const std::vector<T>&) は引数が const std::vector<T>の左辺値だった場合のみに呼ばれます。それ以外の場合はすべて下の関数 f(T&&) が呼ばれます。

オーバーロードをする場合は型制約をつけなければ、だいたいは ADL で選ばれるので邪魔になります。ジェネリックな関数を書いてはいけません、型制約を絶対につけなければいけません。C++14 までならば、つぎのように enable_if の SFINAE 技法を用います。

```
template <class T>
struct is_vec: std::false_type {};

template <class T>
struct is_vec<std::vector<T>>: std::true_type {};

// for vector
template<typename T>
f(const std::vector<T>&) { std::cout << "vector<T>"; }

// default
template<typename T>
std::enable_if_t<!std::is_same_v<is_vecstd::decay_t<T>>::value>
f(T&&) { std::cout << "T&&"; }
```

C++17 を使っていて、オーバーロードをしないのであれば、以下のように constexpr if を使えばよいです。

```
template<typename T>
void f(T&&) {
  if constexpr (is_vec<std::decay_t<T>>::value) {
    std::cout << "vector<T>";
  }
  else {
    std::cout << "other";
  }
}
```

4.2 構造化束縛

構造化束縛は C++17 で追加された機能です。ペアやタプル、配列や構造体を分解して各要素を取り出す機能で、非常に便利です。

`auto [a, b] = x` のように書いたとき、`a,b` がどのような型になるのか？ これについていったい何人に説明したか、数え切れません。

まず、大切なことを言っておきます。新しい機能を使う前に、`cppreference.com` で一度調べてください。雰囲気で使わないでください、おねがいします。

この機能は宣言に相当する文法です、すでに宣言された変数に代入はできません。また、型宣言は `auto` に cvr 修飾がついたものしか書くことができないです。また、`_`のようなものを書くと値が無視されるというような便利な機能もありません（代入ができないので、もちろん `std::ignore` も使えません)。構造化束縛は入れ子にできません。

`auto const& [a, b] = x` のように書いたから `a,b` が const になる、、、**とはかぎりません！**

構造化束縛の型宣言 `auto const&`は分解対象 `x` を受け取るときの仮引数宣言だと思ってください。

`x` が配列の場合順番に添字アクセスが繰り返されるだけです。

std::tuple_size<T>::value が妥当な式でないクラスの場合はメンバアクセスになります。

この 2 つの場合は a,b の型は auto const& [a, b] = x のように書いたから a,b が const になるというのは真実です。

タプルライク型の場合は参照型を持てるため話が変わります。std::tuple_size<T>::value が妥当な式であるクラスの場合は get<i>(x) の呼び出しによって行われます。よってタプルライク型の場合、i 番目の構造化束縛の変数の型は std::tuple_element<i, E>::type になります。

```
float x{};
char  y{};
int   z{};

std::tuple<float&,char&&,int> tpl(x,std::move(y),z);

const auto& [a,b,c] = tpl;

// a that refers to x; decltype(a) is float&
// b that refers to y; decltype(b) is char&&
// c that refers to the 3rd element of tpl; decltype(c) is const int
```

要素が参照型の場合、tuple の要素型がそのまま手に入ることになります。

4.3 この章のまとめ

- Forwarding Reference を使う場合はオーバーロードに型制約を設ける
- 新しい機能を使う前に、cppreference.com で一度調べる

C++ プログラミング診断室

2019 年 7 月 11 日　初版第 1 刷　発行
著　者　　いなむのみたま
イラスト　俺九番