

Comprehensive Knowledge of C++(V18.06)

autor:ichiroprogrammer

イントロダクション

本ドキュメントは、C++でのプログラミング、デザイン、ソフトウェア開発におけるプロセス、ワークフロー等に対する原理、原則をまとめたものである。

ソフトウェア工学を実践的に学習した者には明らかなように、ソフトウェア工学はトレードオフを扱う学問である。従って、後述する、しないにかかわらずソフトウェアの原理、原則は、常にそれが正しいとは限らず、またそれらが互いに矛盾することもある。老子の至言「道の道とすべきは、常の道に非ず」のごときものである。

ソフトウェア開発の現場においては、このことを常に念頭に置き、慎重且つ深い洞察を以って、これらを実践すべきである。

本ドキュメントの目的

「守破離」とは、日本の伝統武道、技芸の修行者が歩むべき習熟モデルである。「守」、「破」、「離」それぞれが以下のような修行者の習熟レベルと、それに基づいた行動規範を表す。

- ・「守」 - 未熟な修行者であり、まずは、先人の残した「形(型)」を無批判に受け入れ、それを反復すること
- ・「破」 - 「形」に習熟した修行者であり、「形」に自らの工夫を加えながら、より高いレベルを探求すること
- ・「離」 - 最高レベルに到達した修行者であり、自らが「形」を作り出すこと

このドキュメントは、

- ・「守」のレベルにあるC++プログラマに「形」を提供すること
- ・そのようなプログラマの「破」への道しるべとなること
- ・このドキュメントを批判的に読むことで、「破」のプログラマに「離」への切っ掛けを与えること
- ・C++で開発を行うチームが作り出すソースコードに規律や整合性を与えること、もしくは守るべき規範のボトムラインを示すこと

を目的とする。

本ドキュメントの次に

本ドキュメントを読み、理解したプログラマの次のステップのために、卷末に「参考文献」を設けた。

それらを読破し、合理的理由でこれらに倣い、もしくは批判できるようになることで、「守」が終了し、「破」へと進むことができると考えられる。

改訂履歴

- V18.06
 - アーキテクチャに非同期処理用のサンプルコードを追加
- V18.05
 - テーブルの体裁を整えた
- V18.04
 - インデックスを廃止
 - depsにCMakeLists.txtを追加
- V18.03
 - 全体のインデックスを最小にした
 - 各章ごとにインデックスをつけた

本ドキュメントの修正・改善

本ドキュメントをソフトウェア開発のルールとして採用したチームのプログラマにとって、本ドキュメントは守るべき「形」であるが、「形」であることは不変であることを意味しない。むしろ、ソフトウェア工学の進歩やチームの習熟に合わせ、その時点で最も正しいであろうルール、プラクティスを積極的に導入することで、「形」の陳腐化を防ぐべきである。このことは、スコット・マイヤーズ氏によるEffective C++の第1版から第3版、およびEffective Modern C++までの変遷を読むことで、より深い理解が得られるはずである。従って、「破」や「離」のレベルにあるプログラマは、このドキュメントや、このドキュメントから派生した(もしくは、それとは無関係な)チームのルールの内容を定期的に見直し、より高いレベルへと改善させるよう努めてほしい。

例示したコードの説明

次章以降では、ソースコードを使って説明を行う場合がある。このような場合の注意点を述べる。

- 「...」のみで構成された行は、ソースコードの省略を表す。
- 特定の規則、法則、慣習等を説明するためのソースコードは、シンプルさを優先するため、他の規則、法則、慣習に従っていない場合がある。特に識別子に関しては、「命名規則」に従っていない場合が多い。また、一般にSTLのコンテナクラスやstd::stringをnewする必要はないが、コードの動作を示すためにあえてそのようにする場合がある。
- ソースコード内に動作説明のような本来不要なコメントがあるのは、読者にその意味を知らせるためであるため、製品コードのコメントをこのようにするべきではない。
- 例示したコードの動作の確認、明示のためにgoogle test(gtest)のアサーション(下表)を使用する。

アサーションマクロ	意味
ASSERT_TRUE(x)	xがtrue
ASSERT_FALSE(x)	xがfalse
ASSERT_EQ(x, y)	(x == y)がtrue
ASSERT_NE(x, y)	(x != y)がtrue
ASSERT_GE(x, y)	(x >= y)がtrue
ASSERT_GT(x, y)	(x > y)がtrue
ASSERT_LE(x, y)	(x <= y)がtrue
ASSERT_LT(x, y)	(x < y)がtrue
ASSERT_STREQ(x, y)	(std::string(x) == std::string(y))がtrue
ASSERT_DEATH(x, y)	xを実行するとアボートすればtrue
ASSERT_THROW(x, y)	xを実行するとy例外が発生すればtrue

- 問題を示すために掲載したコードは、実行すると不具合を発生させことがある。 そういった場合、下記のように単体テストのラベルにDISABLED_を付けることで、その実行を抑止することがある。

```
TEST(DISABLED_Xxx, yyy)
{
    // 単体テスト
}
```

本ドキュメントでの言葉の使い方の注意

- 参照
 - 「～を参照する」というような文脈で使われる「参照」はそのまま使用する。
 - C++での参照型を表す参照は使わず、代わりに「リファレンス」を使用する。
- 例外
 - 「～の場合は例外である」というような文脈で使われる「例外」はそのまま使用する。
 - C++でthrowすると発生する事象を表す例外は使わず、代わりに「エクセプション」を使用する。
- classとクラス
 - classはC++のキーワードとして使用する。
 - クラスは上記以外で使用する。
- プログラミングとコーディング、ソースコードとコード、インスタンスとオブジェクト
 - 同じような意味で使用する。

本ドキュメントの構成

- [ソフトウェアプラクティス](#)
- [プログラミング規約](#)
- [コード解析](#)
- [コーディングスタイル](#)
- [命名規則](#)
- [コメント](#)
- [SOLID](#)
- [デザインパターン](#)
- [アーキテクチャ](#)
- [開発プロセスとインフラ](#)
- [並行処理](#)
- [テンプレートメタプログラミング](#)
- [ダイナミックメモリアロケーション](#)
- [デバッグ](#)
- [開発ツール](#)
- [deps](#)
- [用語解説](#)
- [演習](#)
- [解答](#)
- [参考文献](#)
- [あとがき](#)
- [Sample Code](#)

ソフトウェアプラクティス

ピーター・ドラッカーは「プロフェッショナルの条件」の中で、

紀元前440年頃、ギリシャ彫刻家フェイディアスはアテネのパンテオンの庇に建つ彫刻郡を完成させたが、フェイディアスの仕事の請求書に対して、アテネ会計官は「彫刻の背中は見えない。見えない部分まで彫って請求してくるとは何事か」と全額の支払いを拒んだ。フェイディアスは言った。「そんなことはない。人々が見えない彫刻の背中は、神々の目が見ている。」

ドラッカーはこの逸話を引用することで、プロフェッショナルとは外部からの承認や評価を超えて、自己の内なる基準に忠実であり、目に見えない部分にも最高の品質を求める人物であるべきだと説明している。つまり、プロフェッショナルは、自分の仕事に対する誠実さと完璧を追求することによって、自己の専門性と倫理観を高める人物であるとドラッカーは示している。この逸話は、プロフェッショナリズムの本質が単に技術的なスキルや知識にあるのではなく、仕事に対する深い献身と責任感にあることを強調している。

ドラッカーの考えるプロフェッショナルに、ソフトウェアエンジニアがそのレベルに到達するために重要であると考えられる原則・法則を下記に紹介する。

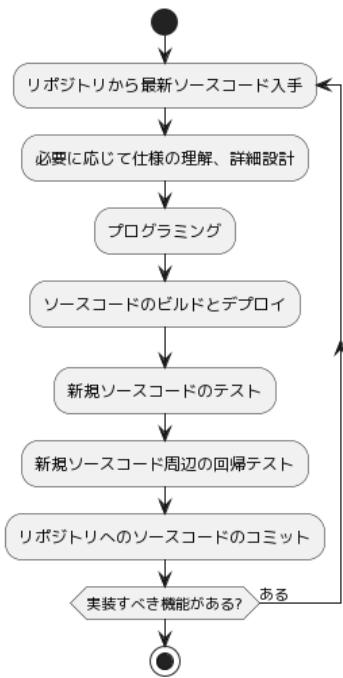
- プログラマの行動に関する原則・法則
 - DRYの原則
 - KISSの原則
 - ボイスカウトの規則
 - YAGNIの原則
 - Name and Conquer
 - ノコギリの刃を研ぐ
 - 推定有罪
- デザイン、プログラミングに関する原則・法則
 - デメタルの法則
 - コンウェイの法則
 - Inside-Outの原則
 - SOLID

DRYの原則

DRYとは、“Don’t repeat yourself.”の略であることから推測できるように、「ソフトウェア開発での繰り返し作業や成果物の重複を避けよ」という原則である。

作業の繰り返し

ソフトウェア開発におけるプログラマのワークフローは下記のようなものである。



ソフトウェア開発において、この流れの繰り返しは避けられないが、この一連の作業のうち、少なくとも「ソースコードのビルドとデプロイ」と、「新規ソースコード周辺の回帰テスト」での手作業のほとんどを自動化により回避可能である。

もし、これらにまつわる作業のほとんどが手動で行われていれば、これらに多くの工数が奪われる。また、手作業によるミスは避けられず、それによりさらに多くの工数が奪われる。

容易に推測できるように、これらの自動化は絶大な効果を発揮するため、生産性改善の必須項目となっている(「自動単体テスト」、「自動統合テスト」参照)。

様々な自動化にスクリプト言語の習得が必要ならば、それを良い学習機会と捉えるべきである。その投資対効果は間違いなく、ポジティブである。

成果物の重複

成果物の重複とは、

- コードクローン(ほぼ同じソースコードの断片)
- ソースコードに書いてあることをそのまま自然言語で説明したコメント
- ソースコードを読むことで容易にわかる詳細設計ドキュメント

等を指す。

成果物の重複の中で最も罪深いものはコードクローンである。コードクローンにより、

- ソースコード修正時にその箇所が多岐にわたるため、デグレードや修正漏れの原因になる。
- コードレビューの時間が延びる。
- ビルドが遅くなる。
- リポジトリが不要に大きくなる。

等の様々な問題が発生する。また、コードクローンは、コードクローンの原因にもなるため、これらの問題は時とともに加速度的に大きくなる。

言語仕様に精通し、プログラミングイデオムや、デザインパターンを学び、その知識を利用して定期的にリファクタリングを行うことでコードクローンの発生は軽減、回避可能である。

KISSの原則

“Keep it simple, stupid!”の頭文字からなる原則であり、意味は読んで字のごとしである。

「フェルマーの最終定理」で有名なフェルマーは友人에게手紙の冒頭で、時間不足のため手紙が長くなることを謝った。手紙の推敲不足ならば謝罪で十分だろうが、対象が設計ドキュメントや、ソースコードの場合、謝罪では問題は解決しない。なぜなら、そのドキュメントやソースコードは次の開発のベースラインとなるからである。

不要な複雑さを持ったそれらは、明日行うかもしれない次の開発を、間違いなく難しくさせる。この設計上の負債が一定量を上回れば、それ自体が不要な複雑さを増やす原因となり、問題はさらに複雑になる。

残念ながら、一旦この負のスパイラルに陥ってしまえば、これを止め、逆転させることは極めて困難であり、スクラッチから作り直すよりも、多くの工数、スキルを要する。

設計ドキュメントや、ソースコードから不要な複雑さを切り捨て、シンプルに保とう。時間的制約で、すぐにそれに取り掛かれない場合は、成果物にマーキングを行い、やるべきことを忘れないようにしよう(マーキングキーワードにはTBR(== to be refactored)を推奨)。時間ができた時には、そのキーワードを検索し、すぐに整理・整頓、リファクタリングに取り掛かろう。中国のことわざに「双葉のうちに刈らずんば、斧をもちうる」とある。斧が必要になる前に、必ずその問題を刈り取ろう。

ボイスカウトの規則

ボイスカウトの規則の元々の意味は、「山に行ったときよりも、山から帰るときの方がきれいになるよう行動せよ」というものである。転じて、「チェックアウトされたソースコードよりも、コミットされるソースコードの方がきれいになるようプログラミングせよ」という意味のソフトウェア開発における規則となった。

この規則を守るために、まずはリポジトリへのコミット前に変更差分の自己レビューを行い、以下のような不要なコードが残っていないかチェックしよう。

- 不要なコメント
- 改行コードの違い
- タイポ
- 不要なスペース文字
- printfデバッグの残骸
- 一時的な#if 0/#if 1

この規則には、もう一点重要な示唆が含まれている。グローバル変数の多さや、関数の大きさを指摘された時に、「最初から、汚くて...」というような言い訳をするプログラマは珍しくない。それはおそらく真実であろうが、それでも、もともと汚かったソースコードをさらに汚してしまったのでは、問題は大きくなるばかりである。従って、この規則には「グローバル変数やそれらへのアクセス箇所を増やさない」、「関数をこれ以上大きくしない」程度の努力は常にすべきである、ということも含まれている。

YAGNIの原則

YAGNI(You ain't gonna need it)とは、「機能(要件)は実際に必要となるまで実装しない」という原則である。これは、

- 単なる思い付きや予想により、先回りして実装したコードの多くが不要である。
- その時点で必要でない機能は、それが存在するだけで工数を浪費する。
 - ビルド時間や単体テスト時間が余計にかかる。
 - 必要な機能追加に、余計な負担を与える(コードが不要に複雑になる)。
- 本来やるべきことに集中した方が良い結果が得られる。

といったエクストリームプログラミングのエクスパートたちの主張である。なお、上記の「単なる思い付きや予想」とは、演繹的推論とは異なる。演繹的推論により、

- 近い将来その機能が必要になる。
- その機能の実装を必要になるまで先送りするよりも、今行った方が良い合理的な理由がある(記憶がホットである等)。

というような場合は、この原則の限りではない。

「コンウェイの法則」でも述べるように、「レイヤに水平分割されたパッケージ構造を持つソフトウェア」を、「各チームや個人がそれぞれのパッケージ開発に責任を持つ組織」が開発を行う場合、レイヤ毎に開発が進められ、各パッケージの結合は後回しにされる。そのため、各パッケージの開発者は実際に必要になるかどうか確定しないものを開発することになる。このスタイルは明らかにYAGNIの原則に違反する。これが、「各チームや個人がそれぞれのパッケージの開発に責務を持つ組織」が非効率になってしまう理由の一つである。

Name and Conquer

かつてデカルトは、「困難を分割せよ」といった。そのままでは複雑すぎて理解不可能な対象物を、分割して理解していく方法は、現在では「要素還元」や「divide-and-conquer(分割統治)」と呼ばれる。このdivide-and-conquerは、デカルト以前から、現在に至るまで、そして今後も、対象物を理解するための極めて有効な手段である。

これと同等に強力な手法がname-and-conquer(命名統治)である。対象物が適切に命名されてなければ、我々はそれを正確に記憶することが難しい。正確に記憶できなければ正確に理解できない。仮に理解できたとしても他者との理解を共有することは難しい。一方、それに適切な名前を付けることで、その対象物の理解が始まり、他者との理解の共有が可能となる。「クラウド・コンピューティング」という命名がその好例である。

以下のような事に気を付け、自分たちが使う概念やソフトウェア構成物に、適切な命名をしよう。

- ・ その名前が持つ意味の搖らぎをプロジェクトから排除する。
- ・ すでにグローバルスタンダードとして存在する名前を別の用途で使わない。
- ・ 自分たちが必要とする概念が、すでにグローバルスタンダードな名前を持つのであれば、それをそのまま使用する。
- ・ 新規概念を理解できていないか、それ自体が大きすぎれば、それへの命名は難しい。その場合、その概念を分割し、分割された部分に命名する。
- ・ 命名に誤りはつきものである。一旦定めた名前よりも適切なものが見つかった場合、その修正を躊躇してはならない。
- ・ XXXコントローラや、XXXマネージャ等の命名は、それらの責務が限定されない場合が多い(こういった名前を、「強すぎる名前」と呼ぶ)。もっと限定的な(弱い)名前を付ける。

「[命名規則](#)」に詳細をまとめたので参照してほしい。

ノコギリの刃を研ぐ

まずは、「[7つの習慣](#)」からの引用を紹介する。

森の中で木を倒そうと、一生懸命ノコギリを挽いている樵(きこり)に出会ったとしよう。

「何をしているんですか」

とあなたは訊く。すると

「見れば分かるだろう」

と、無愛想な返事が返ってくる。

「この木を倒そうとしているんだ」

「すごく疲れているようですが、。いつからやっているんですか」

あなたは大声で尋ねる。

「かれこれもう五時間だ。くたくたさ。大変な作業だよ」

「それじゃ、少し休んで、ついでにそのノコギリの刃を研いだらどうですか。

そうすれば仕事がもっと早く片付くと思いますけど」

あなたはアドバイスをする。

「刃を研いでいる暇なんてないさ。切るだけで精一杯だ」

と強く言い返す。

残念ながら、ソフトウェア開発現場でも、この滑稽な状況を頻繁に目にすることになる。このようになりたくないければ、定期的にノコギリの刃を研ぐことである。そのヒントを下記する。この他にもたくさんあるはずである。

- ・ タッチタイピングや開発ツールのショートカットを身に着ける。
- ・ すぐれたエディタやIDEを使いこなす。
- ・ printfデバッグのみに頼ることはやめて、すぐれたデバッガ([デバッガの使用](#)参照)を使う。
- ・ 「[DRYの原則](#)」を実践し、繰り返しの手作業をなくすためのツールを導入、開発する。
- ・ 静的・動的解析ツールを利用し、最小の努力でバグやバグの元になる質の悪いコードを見つけ改善する。
- ・ オープンソースを利用する。
- ・ 新しい言語や、使い慣れた言語の最新機能を学ぶ。
- ・ 効率的なプロセスを理解し、導入する。
- ・ 定期的に書籍やウェブから最新技術情報を入手する。

こういったことに何一つ取り掛からないのであれば、「時間がない」とつぶやきながら、切れないのでノコギリを挽き続けることになる。

推定有罪

言うまでもなく、[推定無罪](#)とは「何人も、有罪を宣告されない限り無罪である」という原則であり、この順守は近代国家であるための必要条件となっている。これは人権を守るといった観点のみでなく、「[悪魔の証明](#)」という言葉で表されるような「多くの場合、無罪の証明は技術的に困難である」といった観点でも重要な原則である。我々がそういった原則を順守している国家に属していることは大変結構なこ

とではあるが、この原則はソフトウェア開発においては当てはまらない。にもかかわらず、これをソフトウェア開発に持ち込むプログラマがいる。

そういったプログラマの行動パターンは以下のようなものである。

- プログラミングルールを守らない。
- ソフトウェア工学的に良くないことだと知りつつ、下記のようなことを行う(おそらく、下記が良くないことであると知らないプログラマはいないだろう)。
 - グローバル変数を作る。
 - 巨大なソフトウェア構成物(ファイル、クラス、関数等)を作る。
 - コピー&ペースト・プログラミングをする(コードクローンを作る)。
 - コンパイラや静的解析ツールの警告を無視する。
- ソースコードのコミットに際して、プログラマが当然すべき以下のような作業をしない。
 - コミット予定のソースコードの自己レビューをする。
 - コミット予定のソースコード周辺の回帰テストをする。
 - コミット後、それを別のリポジトリにチェックアウトする等して、コミットに抜け漏れが無いかを確認する。
 - コミット後のデグレード発生に対して、リポジトリ先頭の品質を回復させる。
- 会議等の時間を守らない。
- 自分のスキルの低さを気にしない。

等々挙げればきりがない。これらの行動は、どの一つをとっても、組織やその組織の成果物を直ちに棄損したと証明することは難しいため、推定無罪の原則を適用すれば、このプログラマの行動は無罪である。

良識あるプログラマから見れば、明らかに有罪である行動が無罪になる理由は、その論拠となる原則が間違っているからである。ソフトウェア開発の現場においては、常に**推定有罪**「無罪を証明しない限り有罪」の原則が正しい。とは言え、先に書いたようにそれは「悪魔の証明」することになり、実践的には不可能である。

故にソフトウェア開発の現場における推定有罪の原則とは、「無罪であると演繹推論できるものは無罪という前提の元、無罪を証明しない限り有罪である」というようなものである。

この観点に従って、上記推定無罪プログラマの行動パターンを矯正すると下記のようになるだろう。

- プログラミングルールを守る。
- ソフトウェア工学的に良くないことはしない。
- ソースコードのコミットに際して、最低でも下記のような作業を行う。
 - コミット予定のソースコードの自己レビューをする。
 - コミット予定のソースコード周辺の回帰テストをする。
 - コミット後、それを別のリポジトリにチェックアウトする等して、コミットに抜け漏れが無いかを確認する。
 - コミット後のデグレード発生に対して、リポジトリ先頭の品質を回復させる。
- 会議等の時間は守るか、守れない場合は前もって、その旨を開催者に伝える。
- 自分のスキルに気を配り、常に向上に努める。

当たり前のことであるが、意外なほど多くのプログラマが出来ていない。

デメテルの法則

最小知識の原則(Principle of Least Knowledge)とも呼ばれ、

- オブジェクトは、それ以外の構造やプロパティに対して(他のオブジェクトに対して)持っている知識、前提を最小限にすべきである。
- という設計上の制約である。

クラスAから生成されたオブジェクトaが、クラスBから生成されたオブジェクトbのメンバ関数「void* B::f()」を呼び出し、戻りのポインタを何らかの型にキャストして使用するようなコードは、この法則違反となる。当然ながら、型付けの厳格なC++の仕様の裏をかくこのような記述は避けなければならない。

オブジェクトaが、オブジェクトbのメンバ関数「C& B::f()」を呼び出すことにより、クラスCから生成されたオブジェクトcにアクセスし、そのメンバ関数「C::g()」を呼び出す「a.f().g()」のような多重の呼び出しあるこの法則違反となるが、この例でのクラスCがstd::string constのように状態が不变で安定した仕様を持つならば、このような多重呼び出しが問題になることは稀である。一方で、この法則に従うと、クラスBにC::g()のラッパー関数を作ることになるため、これをプロジェクト全体で守れば、多量のラッパー関数を作らざるを得なくなる。その結果として、多くのクラスの凝集性(「凝集度」参照)が低下する。

これらを総合して考えると、上記オブジェクトcが「ミュータブルである」、もしくは「仕様が流動的である」ときのみにこの法則を適用するべきである。

コンウェイの法則

コンウェイの法則とは、「システムを設計する組織は、その構造をそっくりまねた構造の設計を生み出してしまう」現象を説明するものである。

ソフトウェア開発を行う組織は、ドメイン知識に基づいた組織構造(GUIチーム、ミドルウエアチーム、ドライバチーム等)を持つことが多い。こういった組織が、コンウェイの法則に陥ってしまう。

- ・ソフトウェア全体を、水平にパッケージ分割したアーキテクチャを構築する
- ・ドメイン組織に基づいたチームは、このパッケージの開発責任を持つ

ことになる。これは合理的な組織統治に思えるが、以下のような避けがたい問題を生み出す。

- ・パッケージのインターフェースや責務の修正には、組織間での調整会議が必要となる。この会議には、多くの意思決定者が参加することになるが、この会議参加者のインセンティブはさまざまであるため(自分のチームの責任範囲を小さくしたいという意図も働く)、この会議での意思決定が技術的最適解になることは稀である。
- ・大規模なソフトウェア開発では、パッケージのこのような修正は不可避であり、そのたびに上記会議を招集することになる。参加者全員が理解できる言語は自然言語のみであるため、この会議の趣旨を説明するためのパワーポイント資料を作ることになる。このパワーポイント資料を作ることができる人間は、通常、優秀なプログラマであるため、ただでさえ不足しがちな優良なヒューマンリソースをさらに不足させてしまう(この状態が定常化しパワボ資料ばかりを作っている、かつて優秀だった元プログラマをパワポッターと呼ぶ)。

また、このような組織構造は、各チームを担当パッケージの開発に集中させてしまうため、ソフトウェア全体の開発責務(要件開発責務)の所在があいまいになる。これにより、以下のような問題も生み出す。

- ・ソフトウェア全体として統一的に必要な要件(ログ、独自のアサート機能、共通ライブラリ)がおざなりにされ、パッケージごとにそのためのライブラリが作られるか、そのようなものは一切作られないことになる。何れにしても全体として統一感のないソースコードが出来上がる。
- ・「YAGNIの原則」に違反するため、要件の実装とは無関係なコードが作られる。
- ・下位パッケージに依存する上位パッケージは、下位パッケージの進捗に強い影響を受け、上位パッケージほど進捗が遅れる。
- ・全パッケージが完成するまでソフトウェア全体のテストができないため、ソフトウェア全体の進捗率を正しく計測することができない。

かくして、このような組織は、

- ・長いミーティングが多く、そこでの意思決定は技術的最適解には程遠い。
- ・かつて優秀だったプログラマは、今ではパワーポイント資料作りを主な業務とするパワポッター。
- ・進捗率が、人の意思によってきめられており、進捗遅れが頻発する。
- ・統一感がなく、不要に複雑で、クローンだらけのソースコードを作る。

これを防ぐためには、「ドメイン知識に基づいた組織構造をそのままソフトウェア開発に持ち込む」ことをやめなければならない。

Inside-Outの原則

「Inside-Outの原則」というキーワードは「7つの習慣」でも使用されている。この名著のそれは、「何らかの問題が起ったときに、それを他責にすることなく(環境や他人の問題)、まずは自分の管理可能なこと(自分の性格、物事の捉え方、感じ方、動機)に目を向けて対処する」ことを求める。これは、これで重要な行動規範であるが、ここで説明する「Inside-Outの原則」はこれとは異なる(もしかすると、「7つの習慣」からの引用かもしれない)。

この原則は、GUIアプリケーションを「MVC」系のアーキテクチャで開発する場合(それ以外の方法があるとは思えないが)、「開発は GUI(View)からではなく、Modelから始めよ」という開発手順への制約である。

ViewはModelの画面への表出であり、Viewを表すためにModelがあるわけではない。まずはModelの関係性を見極めることは、すぐれたGUIアプリケーション開発の要諦である。

ところが、多くのプログラマはGUIから作りたがる。これがModelに悪影響を与え(もしくは、Viewの中にModelを作りこみ、ViewとModelが癒着した構造を作る)、容易にデグレードを引き起こす不安定なソフトウェアを作り出す。このような悪構造は、「DRYの原則」で述べたような「回帰テストの自動化」もほとんど不可能にさせる。

Modelから開発を始め、それを外部の「回帰テスト用ソフトウェア」から実行できるようにしよう。その後、それをViewと組み合わせて実行できるようにすれば、MVCに基づいたアーキテクチャと、それを自動テストできるソフトウェア入手できる。

SOLID

SOLIDとは下記に示す5つの原則である。

- 単一責任の原則(SRP)
- オープン・クローズドの原則(OCP)
- リスコフの置換原則(LSP)
- インターフェース分離の原則(ISP)
- 依存関係逆転の原則(DIP)

この5原則はオブジェクト指向(OOD/OOP)プログラミングにおいて特に重要なものであり、すべてのプログラマはこれらに従って開発することが求められる。

これらの解説は、「プログラミング規約」の説明を行った後、「デザインパターン」の直前で行う。

プログラミング規約

組織に秩序を与える法、道徳、慣習等をここではルールと呼ぶことにする。当然ながら、秩序ある組織には良いルールがあり、混沌とした組織には悪いルールがあるか、ルールはあっても守られていないか、そもそもルールが存在しない。

秩序あるソースコードとは、

- 可読性が高い。
 - 簡潔に記述されている。
 - 記述スタイルが統一されている(「コーディングスタイル」参照)。
 - ファイルや識別子の名前に規則性があり、適切に命名されている(「Name and Conquer」、「命名規則」参照)。
 - コメントの記法が統一されており、内容が適切である(「コメント」参照)。
- 保守、テスト、移植等が容易である。
- 型安全性が配慮されている。
- コンパイル警告レベルが高く、かつ指摘がない(「g++の警告機能」参照)。

のような特性を満たすものであるが、そうあるためには秩序ある組織と同様に良いルールが必要である。本章の目的は、C++プログラミングにおけるそのようなルール(=プログラミング規約)を示すことである。

なお、型安全性とは、「正しく型付けされたソースコードは未定義動作をしない」ことが保証されるという言語の特性である。配列のオーバーランが未定義動作を引き起こすことを考えれば明らかである通り、C++は型安全性を保証しない。このことは、C++の劣等性を意味しないが、それに配慮したプログラミング(型システムの最大限の利用等)が必要となることは事実である。

この章の構成

型とインスタンス

算術型

enum

bit field

class

struct

union

配列

型エイリアス

const/constexprインスタンス

リテラル

型推論

インスタンスの初期化

rvalue

クラスとインスタンス

ファイルの使用方法

クラスの規模

アクセスレベルと隠蔽化

継承/派生

非静的なメンバ変数/定数の初期化

静的なメンバ変数/定数の初期化

mutableなメンバ変数

スライシング

オブジェクトの所有権

オブジェクトのライフタイム

非メンバ関数/メンバ関数

非メンバ関数

メンバ関数

非メンバ関数/メンバ関数共通

構文

複合文

switch文

if文

範囲for文
制御文のネスト
return文
goto文
ラムダ式
マクロの中の文

演算子
優先順位
代入演算
ビット演算
論理演算
三項演算子
メモリアロケーション
sizeof
ポインタ間の演算
RTTI
キャスト、暗黙の型変換

プリプロセッサ命令
関数型マクロ
マクロ定数

パッケージとその構成ファイル
パッケージの実装と公開
識別子の宣言、定義
依存関係
二重読み込みの防御
ヘッダファイル内の#include
#includeするファイルの順番
#includeで指定するパス名

スコープ
スコープの定義と原則
名前空間
using宣言/usingディレクティブ
ADLと名前空間による修飾の省略
名前空間のエイリアス

ランタイムの効率
前置/後置演算子の選択
operator X、operator x=の選択
関数の戻り値オブジェクト
move処理
std::string vs std::string const& vs std::string_view
extern template

標準クラス、関数の使用制限
STL
POSIX系関数

その他
assertion
アセンブラ
言語拡張機能

特に重要なプログラミング規約

型とインスタンス

算術型

整数型

- 整数型には、整数の基本型(intやlong等)を直接使わずに、`cstdint`で定義されている型エイリアスを使用する。
 - STLやPOSIX等の標準クラスや関数のインターフェースが基本型を直接使用している場合は、その型に合わせるために基本型を直接使用する。
- 整数型には、特に理由がない限りに、`int32_t`を使用する。
- 整数型の変数が負にならないのであれば、`uint32_t`を使用する。
 - 符号あり型との演算がある場合は、その変数が負にならなくとも`int32_t`を使用する。
 - 符号あり型と符号なし型との比較をしない。
- `sizeof`の値や配列の長さの保持等には、`size_t`を使用する。
- `int32_t`から`int16_t`や、`int32_t`から`uint32_t`等の値が変わることがある代入を避ける。やむを得ずそのような代入をする場合、下記のような`narrow_cast`を使いこの問題を緩和する。

```
// @@@ example/programming_convention/type_ut.cpp 9

template <typename DST, typename SRC>
DST narrow_cast(SRC v)
{
    static_assert(std::is_integral_v<DST> && std::is_integral_v<SRC>,
                 "DST, SRC should be integral-type.");
    auto r = static_cast<DST>(v);

    assert((r < 0) == (v < 0));           // 符号が変わっていないことの確認
    assert(static_cast<SRC>(r) == v);    // bit落ちしていないことの確認

    return r;
}

// @@@ example/programming_convention/type_ut.cpp 28

auto ui32 = narrow_cast<uint32_t>(128);           // 安全なint32_t -> uint32_t
ASSERT_EQ(ui32, 128);
ASSERT_DEATH(ui32 = narrow_cast<uint32_t>(-128), ""); // 危険なint32_t -> uint32_t

auto i8 = narrow_cast<int8_t>(127);           // 安全なint32_t -> int8_t
ASSERT_EQ(i8, 127);
ASSERT_DEATH(i8 = narrow_cast<int8_t>(-128), ""); // 危険なint32_t -> int8_t

i8 = narrow_cast<int8_t>(-1);           // 安全なint32_t -> int8_t
ASSERT_EQ(i8, -1);
ASSERT_DEATH(i8 = narrow_cast<int8_t>(-129), ""); // 危険なint32_t -> int8_t
```

- 演習-汎整数型の選択

char型

- `char`はascii文字の保持のみに使用する。
- `char*`を`void*`の代わりに使わない。
- `char`が`singed`か`unsigned`かは処理系に依存するため、`char型`を汎整数型として扱わない。8ビット整数には、`int8_t`または、`uint8_t`を使用する。
 - バイトストリームを表現する場合、`int8_t*`、`int8_t[]`、`uint8_t*`、`uint8_t[]`のいずれかを使う。

- 演習-汎整数型の演算

std::byte型

- `int`よりもビット幅の小さい組み込み型の演算の結果は汎整数拡張により`int`型になるため、`uint8_t`のビット演算の型も`int`となる。`int`への拡張が意図したものかどうかの判別は困難であるため、`uint8_t`インスタンスにビット演算が必要な場合、`uint8_t`の代わりに下記のように`std::byte`(「BitmaskType」参照)を用いる。

```
// @@@ example/programming_convention/type_ut.cpp 50
// uint8_tのビット演算例

auto u     = uint8_t{0b1000'0001};
auto ret0 = u << 1;
// uint8_t ret1{u << 1}; // 縮小型変換のため、コンパイルエラー
uint8_t ret1 = u << 1;
```

```

static_assert(std::is_same_v<decltype(ret0), int>); // u << 1はintになる
ASSERT_EQ(0b1'0000'0010, ret0);
ASSERT_EQ(0b0000'0010, ret1);

// @@@ example/programming_convention/type_ut.cpp 64
// uint8_tに代わりstd::byteを使用したビット演算例

auto b      = std::byte{0b1000'0001};
auto ret0 = b << 1;
auto ret1 = std::byte{b << 1};

static_assert(std::is_same_v<decltype(ret0), std::byte>); // b << 1はstd::byteになる
ASSERT_EQ(std::byte{0b0000'0010}, ret0);
ASSERT_EQ(std::byte{0b0000'0010}, ret1);

```

- std::byteの初期化には{}を用いる(static_castを使用しない)。

```

// @@@ example/programming_convention/type_ut.cpp 77

std::byte b0{0b1000'0001}; // OK
auto      b1 = std::byte{0b1000'0001}; // OK
std::byte b2 = static_cast<std::byte>(0b1000'0001); // NG
// std::byte b3 = 0b1000'0001; // NG コンパイルエラー

```

bool型

- bool型は、bool型リテラル(true/false)やbool型オブジェクトの保持のみに使用する。
- bool型を汎整数型として扱わない。bool型に++を使用しない(-はコンパイルできない)。

```

// @@@ example/programming_convention/type_ut.cpp 95

#ifndef __cplusplus < 201703L // 以下のコードはC++14以前ではコンパイルできるが、
                           // C++17以降ではコンパイルエラー

    auto b = false;

    ASSERT_EQ(1, ++b); // NG 予想通り動作するが、boolの目的外使用
    ASSERT_EQ(1, ++b); // NG bは2ではなく1
    // ASSERT_EQ(1, --b); // NG コンパイルエラー
#endif

```

- ポインタ型やboolを除く汎整数型のインスタンスをbool値として使用しない。

```

// @@@ example/programming_convention/type_ut.cpp 111

void g(int32_t* ptr0, int32_t* ptr1) noexcept
{
    if (ptr0) { // NG ポインタ型をbool値として使用
        return;
    }

    if (ptr1 == nullptr) { // OK
        return;
    }

    ...
}

```

浮動小数点型

- floatやdoubleのダイナミックレンジが必要な場合のみに、これらの型を使用する。ちなみに銀河系の直径は $1e+21$ メートル程度、プランク長は $1.616229e-35$ メートルであるため、銀河から素粒子までのサイズを一つの基本型で表す場合においても、floatのダイナミックレンジに収まる。従って、floatやdoubleが必要になる場合は極めて限られる。

	正の最小値	正の最大値
floatの範囲	1.175494351 e-38	3.402823466 e+38
doubleの範囲	2.2250738585072014 e-308	1.7976931348623158 e+308
uint32_tの範囲	0	4.294967295 e+9
uint64_tの範囲	0	1.8446744073709551615 e+19

- 演算順序等により誤差が生じることがあるため、floatやdoubleのインスタンスを、==、!=、>、<等で比較しない。

```

// @@@ example/programming_convention/float_ut.cpp 11

// 下記の0.01は2進数では循環小数となるため、実数の0.01とは異なる。
constexpr auto a = 0.01F; // 0.000001010001111...
constexpr auto b = 0.04F; // 0.0000101000111101...

// ASSERT_EQ(0.05F, a + b); // NG a + b == 0.05Fは一般には成立しない。
ASSERT_NE(0.05F, a + b);

// @@@ example/programming_convention/float_ut.cpp 22

/// @fn bool is_equal_f(float lhs, float rhs) noexcept
/// @brief float比較用関数
bool is_equal_f(float lhs, float rhs) noexcept
{
    return std::abs(lhs - rhs) <= std::numeric_limits<float>::epsilon();
}

// @@@ example/programming_convention/float_ut.cpp 34

// 下記の0.01は2進数では循環小数となるため、実数の0.01とは異なる。
constexpr auto a = 0.01F; // 0.000001010001111...
constexpr auto b = 0.04F; // 0.0000101000111101...

// floatの比較はis_equal_fのような関数を使う。
ASSERT_TRUE(is_equal_f(0.05F, a + b)); // OK

```

- 一つの式にfloatとdoubleを混在させない。

```

// @@@ example/programming_convention/float_ut.cpp 46

// 上記例と似たソースコードであるが、下記のような問題が起こる
/// @fn bool is_equal_d(double lhs, double rhs) noexcept
/// @brief double比較用関数
bool is_equal_d(double lhs, double rhs) noexcept
{
    return std::abs(lhs - rhs) <= std::numeric_limits<double>::epsilon();
}

// @@@ example/programming_convention/float_ut.cpp 59

// 下記の0.01は2進数では循環小数となるため、実数の0.01とは異なる。
constexpr auto a = 0.01F; // 0.000001010001111...
constexpr auto b = 0.04F; // 0.0000101000111101...

// a + bはfloatの精度のまま、is_equal_dの引数の型であるdoubleに昇格される。
// 一方、0.05はdoubleであるため(循環小数をdoubleの精度で切り捨てた値であるため)、
// a + b(floatの精度の値)と0.05の差はdoubleのepsilonを超える。
// ASSERT_TRUE(is_equal_d(0.05, a + b)); // NG
ASSERT_FALSE(is_equal_d(0.05, a + b));

```

```

// @@@ example/programming_convention/float_ut.cpp 73

// is_equal_dを改良して、引数の型が統一されていない呼び出しをコンパイルエラーにできるようにした。
/// @fn bool is_equal(FLOAT_0 lhs, FLOAT_1 rhs) noexcept
/// @brief 浮動小数点比較用関数
template <typename FLOAT_0, typename FLOAT_1>
bool is_equal(FLOAT_0 lhs, FLOAT_1 rhs) noexcept
{
    static_assert(std::is_floating_point_v<FLOAT_0>, "FLOAT_0 should be float or double.");
    static_assert(std::is_same_v<FLOAT_0, FLOAT_1>, "FLOAT_0 and FLOAT_1 should be a same type.");

    return std::abs(lhs - rhs) <= std::numeric_limits<FLOAT_0>::epsilon();
}

// @@@ example/programming_convention/float_ut.cpp 90

// 下記の0.01は2進数では循環小数となるため、実数の0.01とは異なる。
constexpr auto a = 0.01F; // 0.000001010001111...
constexpr auto b = 0.04F; // 0.0000101000111101...

// a + bはfloatであり、0.05はdoubleであるため、下記コードはコンパイルできない。
// ASSERT_TRUE(is_equal(0.05, a + b));
ASSERT_TRUE(is_equal(0.05F, a + b)); // OK リテラルに型を指定して、引数の型を統一

```

- INFや、NANを演算で使用しない。
- 汎整型**の演算とは違い、0除算等の浮動小数点演算のエラーは、通常、プログラム終了シグナルを発生させないため、浮動小数点演算のエラーを捕捉する必要がある場合(ほとんどの場合、そうなる)は、`std::fetestexcept()`、`std::isnan()`、`std::isinf()`等を使用してエラーを捕捉する。

```

// @@@ example/programming_convention/float_ut.cpp 102

int global_zero{0};
float div(float a, float b) noexcept { return a / b; }

// @@@ example/programming_convention/float_ut.cpp 110

std::feclearexcept(FE_ALL_EXCEPT); // エラーをクリア

div(1.0F, 0.0F); // 関数の中で0除算するが、終了シグナルは発生しない
ASSERT_TRUE(std::fetestexcept(FE_ALL_EXCEPT) & FE_DIVBYZERO); // 0除算

std::feclearexcept(FE_ALL_EXCEPT); // エラーをクリア

div(std::numeric_limits<double>::max(), 1);

auto const excepts = std::fetestexcept(FE_ALL_EXCEPT);

ASSERT_FALSE(excepts & FE_DIVBYZERO); // 0除算
ASSERT_TRUE(excepts & FE_INEXACT); // 演算が不正確
ASSERT_FALSE(excepts & FE_INVALID); // 不正な操作
ASSERT_TRUE(excepts & FE_OVERFLOW); // 演算がオーバーフローを起こした
ASSERT_FALSE(excepts & FE_UNDERFLOW); // 演算がアンダーフローを起こした

std::feclearexcept(FE_ALL_EXCEPT); // エラーをクリア

auto const a = 1.0F / global_zero; // global_zero == 0
ASSERT_TRUE(std::isnan(a));

auto const b = std::sqrt(-1);
auto const c = std::sqrt(-1);
ASSERT_TRUE(std::isnan(b));
ASSERT_FALSE(b == c); // NaN == NaNは常にfalse

```

- できる限り浮動小数点の代わりに固定小数点を使用する(全ソースコードは「[example/programming_convention/fixed_point.h](#)」に掲載)。

```

// @@@ example/programming_convention/fixed_point.h 6

/// @class FixedPoint
/// @brief BASIC_TYPEで指定する基本型のビット長を持つ固定小数点を扱うためのクラス
/// @tparam BASIC_TYPE 全体のビット長や、符号を指定するための整数型
/// @tparam FRACTION_BIT_NUM 小数点保持のためのビット長
template <typename BASIC_TYPE, uint32_t FRACTION_BIT_NUM>
class FixedPoint {
public:
    FixedPoint(BASIC_TYPE integer = 0,
               typename std::make_unsigned_t<BASIC_TYPE> fraction = 0) noexcept
        : value_{get_init_value(integer, fraction)}
    {
        ...
    }

    ...

    FixedPoint& operator+=(FixedPoint rhs) noexcept
    ...
    FixedPoint& operator-=(FixedPoint rhs) noexcept
    ...
    FixedPoint& operator*=(FixedPoint rhs) noexcept
    ...
    FixedPoint& operator/=(FixedPoint rhs) noexcept
    ...

private:
    BASIC_TYPE value_;
    ...

    friend bool operator==(FixedPoint lhs, FixedPoint rhs) noexcept
    ...

    // FixedPoint() + intのようなオーバーロードを作るためにあえてfriend
    friend FixedPoint operator+(FixedPoint lhs, FixedPoint rhs) noexcept
    ...
};

// @@@ example/programming_convention/fixed_point_ut.cpp 19

```

```

// 以下は、FixedPoint<>の使用例である。
{
    using FP4 = FixedPoint<uint8_t, 4>; // 基本型uint8_t、小数点4ビット
    auto fp0 = FP4{};
}

```

```

...
fp0 = 7;      ASSERT_EQ(7, fp0);
fp0 = 7;      ASSERT_NE(6, fp0);
fp0 += 2;     ASSERT_EQ(FP4{9}, fp0);
fp0 /= 2;     ASSERT_EQ((FP4{4, 8}), fp0);
fp0 /= 2;     ASSERT_EQ(is_equal(4.5, fp0.ToFloatPoint()), fp0);
fp0 *= 2;     ASSERT_EQ((FP4{2, 4}), fp0);
fp0 *= 4;     ASSERT_EQ(FP4{9}, fp0);
fp0 += 7;     ASSERT_EQ(FP4{0}, fp0);
}
...
{
    using FP8 = FixedPoint<int32_t, 8>; // 基本型int32_t、小数点8ビット

    auto fp0 = FP8{};
    ...
    fp0 = 3;          ASSERT_EQ((FP8{3, 0}), fp0);
    fp0 += 3;         ASSERT_EQ((FP8{6, 0}), fp0);
    fp0 -= 3;         ASSERT_EQ((FP8{3, 0}), fp0);
    ...
    fp0 = 3;
    fp0 *= 5;        ASSERT_EQ((FP8{15, 0}), fp0);
    fp0 /= 5;         ASSERT_EQ((FP8{3, 0}), fp0);
    fp0 = fp0 / 2;   ASSERT_EQ((FP8{1, 0x80}), fp0);
    ...
}

```

- 演習-浮動小数点型

enum

- C++の強力な型システムや、コンパイラの静的解析機能(switchでのcase抜け)を効果的に使用するために、一連の定数の列挙にはenumを使用する。
- 使用範囲や方法が明示しづらく、且つ整数型への算術変換が行われてしまう旧来のenum(非スコープd enum)は、一部の例外を除き定義しない。代わりに、より型安全なスコープd enumを使用する。

```

// @@@ example/programming_convention/type_ut.cpp 133

enum CarLight { CL_Red, CL_Yellow, CL_Blue };
enum WalkerLight { WL_Red, WL_Yellow, WL_Blue };

bool f(CarLight cl) noexcept
{
    switch (cl) {
        // 非スコープd enumは下記のようなコードを許容する(if文でも同様)。
        // スコープd enumであればこのような間違いはコンパイルエラーで発見できる。
        case WL_Red: // CL_Regの間違い?
            ...
            break;
        case CL_Yellow: // これは正しい
        case CL_Blue:
        default:
            ...
            break;
    }
    ...
}

```

- 列挙子に値を設定する必要がない場合(具体的な値に意味を持たない場合)には、値を設定しない。値を設定する場合にはそれらを最初に書き、同じ値を設定しない。

```

// @@@ example/programming_convention/type_ut.cpp 163

enum Colour { // NG スコープdになっていない。
    Red   = 0, // NG 配列インデックスでない場合、0を定義する必要はない。
    Green = 1, // NG 連続値を定義する必要はない。
    Blue  = 2
};

...
enum class Colour { // OK
    Red,           // OK 不要な記述がない。
    Green,

```

```
    Blue  
};
```

- enumを配列のインデックスとして使う場合は以下のようにする。
 - スコープドenumの代わりに、旧来のenumをstruct内で定義する。
 - 最初に定義されるenumメンバは0で初期化する。
 - 最後の要素のシンボル名はMaxで終わることにより、その要素が最大値であることを示す。

```
// @@@ example/programming_convention/type_ut.cpp 185

enum class Foo { FooA = 0, FooB, FooMAX };

struct Hoo // structによるスコーピング
{
    enum {
        HooA = 0, // OK
        HooB, // OK 値は暗黙に定義
        HooMAX // OK
    };
};

void f() noexcept
{
//  int32_t a0[foo::FooMAX]; // NG コンパイルエラー
    int32_t a1[static_cast<size_t>(foo::FooMAX)]; // NG castが必要になる
    int32_t a2[Hoo::HooMAX]; // OK
    ...
}
```

- enumはC++11から前方宣言できるようになったため、この機能を使用して、不要なヘッダファイルの依存関係を作らないようにする。

```
// @@@ example/programming_convention/type_ut.cpp 211

enum class IncompleteEnum;
enum class IncompleteEnum2 : uint64_t;

// このファイルから可視である範囲にIncompleteEnum、IncompleteEnum2の定義はないが、
// 前方宣言することで以下の関数宣言をすることができる。
extern void g(IncompleteEnum);
extern void g(IncompleteEnum2);
```

- アプリケーションの設定ファイルに保存された情報を復元させるような場合や、「[BitmaskType](#)」を使用する場合を除き、enumへのキャストをしない。
 - クラスのstatic constの整数定数の代わりにenumを使うことは、C++言語仕様やコンパイラの機能が不十分だった頃のテクニックであり、もはや不要である。代わりにstatic constexprインスタンス(「[constexprインスタンスと関数](#)」参照)を使用する。こうすることで定数の型を明示できる。
- [演習-定数列挙](#)
 - [演習-enum](#)

bit field

- ハードウエアレジスタにアクセスをする目的でのみ使用する。
- bit fieldの型は、unsigned intにする。

class

- 「[クラスとインスタンス](#)」を参照せよ。

struct

- メンバ変数を持つ構造体は、[POD](#)としてのみ使用する。
- メンバ変数を持つ構造体を基底クラスとした継承をしない。従ってそのような構造体は常にfinalであるが、finalの明示はしない。
- メンバ変数(static constやstatic constexprメンバは定数とする)を持たない構造体は、templateや非スコープd enumのスコーピング(「[enum](#)」参照)等に使用しても良い。
- コンストラクタ以外のメンバ関数を定義しない。
 - ディープコピー(「[コンストラクタ](#)」参照)が必要な型は、structでなくclassで表す。
 - デフォルトコンストラクタを除く特殊メンバ関数に対して、= defaultの明示をしない。

```
// @@@ example/programming_convention/type_ut.cpp 224

struct Pod final { // NG finalは不要
    Pod() = default; // OK
    ~Pod() = default; // NG = defaultは不要
    Pod(Pod const&) = delete; // OK copyを禁止する場合
    Pod operator=(Pod const&) = delete; // OK copyを禁止する場合

    int32_t x;
    int32_t y;
};
```

- Cとシェアしない構造体を無名構造体とそのtypedefで定義しない。

```
// @@@ example/programming_convention/type_ut.cpp 237

typedef struct { // NG 無名構造体
    int32_t x;
    int32_t y;
} StructNG; // NG 無名構造体をtypedef

typedef struct StructOK_C_Share { // Cとシェアする場合OK
    int32_t x;
    int32_t y;
} StructOK_C_Share;

struct StructOK { // OK Cとシェアしない場合このように書く
    int32_t x;
    int32_t y;
};
```

union

- ハードウェアレジスタにアクセスをする目的以外で使用しない(以下のような使い方のみ認められる)。

```
// @@@ example/programming_convention/type_ut.cpp 258

union XXX_REG {
    uint8_t bytes[4];
    uint32_t word32;
};

uint8_t f() noexcept
{
    // 0x14000000はハードウェアレジスタのアドレスとする
    auto& XXX_REG_INST = *reinterpret_cast<XXX_REG*>(0x14000000);
    auto byte_1 = XXX_REG_INST.bytes[1];

    return byte_1;
}
```

- 上記のようなunionはランタイム依存性が強いため、それへの依存を最小にする。従って、unionの定義を外部パッケージに公開(「パッケージの実装と公開」参照)しない。
- 上記以外でunionのような機能が必要な場合、std::variant(「std::variantとジェネリックラムダ」参照)を使用する(std::anyはunionの代替えにはならないので、このような場合には使用しない)。

配列

- 配列をnew[]により生成しない。可変長配列が必要な場合は、std::vectorを使用する(「new」参照)。固定長配列を動的に確保する場合は、std::arrayをnewする。
- 配列からポインタへの暗黙の型変換をしない(「キャスト、暗黙の型変換」参照)。特に、オブジェクトの配列をそのオブジェクトの基底クラスへのポインタに代入しないことは重要である(「スライシング」参照)。
- 関数の仮引数を一見、配列に見える型にしない(「実引数/仮引数」参照)。
- char型の配列を文字列リテラルで初期化する場合、配列の長さを指定しない。

```
// @@@ example/programming_convention/type_ut.cpp 279

#ifndef __STDC_CONSTANT_MACROS
// g++では通常コンパイルエラーとなるが、-fpermissiveを付ければコンパイルできてしまう。
char a[3>{"abc"}; // NG aはヌル終端されない
#endif
char a[]{"abc"}; // OK
#endif
```

- 配列の全要素にアクセスするような繰り返し処理には範囲for文を使用する。
- 演習-配列の範囲for文

型エイリアス

- Cとシェアされる型エイリアスを除き、typedefではなくusingを使用する。

```
// @@@ example/programming_convention/type_ut.cpp 296

// C90スタイル
typedef unsigned int uint; // NG
typedef void (*void_func_int32)(int32_t); // NG

...

// C++11スタイル
using uint = unsigned int; // OK
using void_func_int32 = void (*)(int32_t); // OK

template <class T> // templateで型エイリアスを作ることもできる。
using Dict = std::map<std::string, T>; // OK
```

- 型へのポインタのエイリアスは、それを使用してconstポインタが定義できないため、型へのポインタ(関数ポインタを除く)のエイリアスを作らない。

```
// @@@ example/programming_convention/type_ut.cpp 320

using pint32_t = int32_t*; // pint32_tにはconstポインタを代入できない。

int32_t const pint32_0_c = nullptr; // 一見pint32_0_cはconstポインタに見えるが。
int32_t const* pint32_1_c = nullptr;

// pint32_0_cの型とpint32_1_cの型が同じであれば問題ないのだが、
// エイリアスのため結合順が変わった影響でそうはならない。
// *pint32_0_cはconstではなく、pint32_0_cがconstとなる。
static_assert(std::is_same_v<decltype(pint32_0_c), int32_t* const>);
static_assert(std::is_same_v<decltype(*pint32_0_c), int32_t&>);
static_assert(std::is_same_v<decltype(pint32_1_c), int32_t const*>);
static_assert(std::is_same_v<decltype(*pint32_1_c), int32_t const&>);
```

- 演習-エイリアス

const/constexprインスタンス

- インスタンス、インスタンスへのポインタ、インスタンスへのリファレンス等に対して、
 - constexpr(「[constexprインスタンスと関数](#)」参照)にできる場合、constexprにする。
 - constexprにはできないが、const(「[constインスタンス](#)」にはできる場合、constにする。関数の仮引数になっているリファレンスやポインタをconstにすることは特に重要である(「[実引数/仮引数](#)」、 「[三項演算子](#)」参照)。
 - 文字列リテラルのアドレスを非constポインタ型変数に代入しない(「[リテラル](#)」参照)。
 - イテレータにおいても、可能な場合は、イテレータをconstするか、const_iteratorを使う。

```
// @@@ example/programming_convention/type_const_ut.cpp 13

// name0は文字列リテラルを指すポインタなのでconstでなければならない。
char const* name0 = "hoge";

// name1は文字列リテラルでないのでconstでなくてよい。
char name1[] = "hoge";

char const* get_str();
// 左側のconstはname2の指す先をconstにする。
// 右側のconstはname2自体をconstにする。
char const* const name2 = get_str();

// name2の右辺がリテラルならば、下記のようにするべきである。
constexpr char const* name3 = "hoge";

void f(std::vector<int32_t>& vec)
{
    std::vector<int32_t>::iterator const iter = vec.begin(); // iter自体がconst
    *iter = 10;
    // ++iter; // 意図的にコンパイルエラー
```

```

std::vector<int32_t>::const_iterator const_iter_0 = vec.begin(); // *const_iter_0がconst
auto const_iter_1 = vec.cbegin(); // *const_iter_1がconst
static_assert(std::is_same_v<std::vector<int32_t>::const_iterator, decltype(const_iter_1)>);

// *const_iter_0 = 10; // 意図的にコンパイルエラー
++const_iter_0;

// *const_iter_1 = 10; // 意図的にコンパイルエラー
++const_iter_1;

...
}

```

- constは、意味が変わらない範囲で出来るだけ右側に書く。

```

// @@@ example/programming_convention/type_const_ut.cpp 52

const std::string s; // NG
std::string const t; // OK

const std::string* s_ptr; // NG
std::string const* t_ptr; // OK

const std::string& f(); // NG 関数の宣言
std::string const& g(); // OK 関数の宣言

char abc[] {"abc"};
const char* a = abc; // NG *aはconst
char const* b = abc; // OK *aはconst
char* const c = abc; // NG *aではなく、aがconstになり、意味が変わる

const char* const d = abc; // NG
char const* const e = abc; // OK

```

- 演習-constの意味
- 演習-const/constexpr

リテラル

- ヌルポインタを表すポインタリテラルとして、nullptrを使用する(0やNULLを使用しない)。

```

// @@@ example/programming_convention/type_ut.cpp 343

int32_t* a{0}; // NG オールドスタイル
int32_t* b{NULL}; // NG C90の書き方
int32_t* c{nullptr}; // OK C++11

// @@@ example/programming_convention/type_ut.cpp 362

extern int g(long a) noexcept;
extern int g(int* a) noexcept;

// NULLを使ったことで、わかりづらいバグが発生する例
g(NULL); // NG NULLはポインタではないため、この呼び出しがg(long)を呼び出す
static_assert(std::is_same_v<long, decltype(NULL)>);

g(nullptr); // OK 意図通り、g(int*)を呼び出す。
static_assert(std::is_same_v<std::nullptr_t, decltype(nullptr)>);

```

- 文字列リテラル("xxx")はconstオブジェクトとして扱う(「const/constexprインスタンス」、「std::string型リテラル」参照)。
- 長い汎整数型リテラルを使用する場合は、適切に区切りを入れる(C++14)。
- ビットマスク等2進数を使用した方が直感的な場合には2進数リテラルを使用する(C++14)。

```

// @@@ example/programming_convention/type_ut.cpp 384

auto a = 123'456'789U; // = 123456789
auto b = 0x123'456'789U; // = 0x123456789
auto c = 0b1001'0001'0101U; // = 0x915

```

- bool型を表すリテラルにはtrue、falseを使用する。代わりに0、1、!0等を使わない。

```

// @@@ example/programming_convention/type_ut.cpp 395

bool a{0}; // NG
bool b{!0}; // NG
bool c{false}; // OK
auto d = bool{0}; // NG

```

```
auto e = bool{!0}; // NG
auto f = true; // OK
```

- long値リテラルを表す文字には”l”ではなく、“L”を使う。

```
// @@@ example/programming_convention/type_ut.cpp 408

auto a = 4321; // NG 4321と区別が難しい
auto b = 432L; // OK
```

- 演習-危険なconst_cast
- 演習-リテラル

型推論

auto

- AAAスタイルに従い適切にautoを使用する。

```
// @@@ example/programming_convention/type_ut.cpp 422

void f(std::vector<std::string> const& strs)
{
    auto s0 = std::string{"hehe"}; // OK
    auto s1{std::string{"hehe"}; // OKだが、通常は代入を使用する
    auto s2 = s0; // OK
    auto s3 = get_name(); // NG get_name()の戻り値を見ないとs3の型が不明

    for (auto const& str : strs) { // OK strsの型が明らかであるため、strの型も明らか
        ...
    }
}
```

- autoを使用する場合、&、*、const等の付け忘れに注意する。

```
// @@@ example/programming_convention/type_ut.cpp 442

class A {
public:
    A() = default;

#if 0 // NG この関数を呼び出すとクラッシュ
    std::string const& Get(char first_byte) const noexcept
    {
        static std::string const empty;
        for (auto const str : strs) { // NG &の付け忘れたため、スタック上の
            if (str.at(0) == first_byte) { // オブジェクトのリファレンスをreturnする。
                return str;
            }
        }
        return empty;
    }

#else // OK 上のGetの修正。
    std::string const& Get(char first_byte) const noexcept
    {
        static std::string const empty;
        for (auto const& str : strs) { // OK &を付けて、インスタンスのオブジェクトの
            if (str.at(0) == first_byte) { // リファレンスを返せるようになった。
                return str;
            }
        }
        return empty;
    }
#endif

private:
    std::vector<std::string> strs{"aha", "ihi", "uhu"};
};
```

- autoと= {}を使用した変数の宣言には以下のような紛らわしさがあるため、そのような記述を行わない(「インスタンスの初期化」参照)。

```
// @@@ example/programming_convention/type_ut.cpp 493

auto a = 1; // OK aの型はint
auto b(1); // 別の規制でNG ()より{}を優先的に使うべき
```

```

auto c{1};           // OK cの型はint
auto d = {1};        // NG dの型はstd::initializer_list<int>
auto e = {1, 2};     // NG eの型はstd::initializer_list<int>
auto f = std::initializer_list<int>{1, 2}; // OK

static_assert(std::is_same_v<decltype(a), int>, "type not same");
static_assert(std::is_same_v<decltype(b), int>, "type not same");
static_assert(std::is_same_v<decltype(c), int>, "type not same");
static_assert(std::is_same_v<decltype(d), std::initializer_list<int>>, "type not same");
static_assert(std::is_same_v<decltype(e), std::initializer_list<int>>, "type not same");
static_assert(std::is_same_v<decltype(f), std::initializer_list<int>>, "type not same");

```

- auto、decltype, decltype(auto)の微妙な違いに気を付ける。

```

// @@@ example/programming_convention/type_ut.cpp 515

short s0{0};
short& s0_ref{s0};

{ // autoとdecltypeが同じ動作をするパターン
    auto a = s0;
    static_assert(std::is_same_v<decltype(a), short>);

    decltype(s0) d = s0;
    static_assert(std::is_same_v<decltype(d), short>);

    decltype(auto) da = s0;
    static_assert(std::is_same_v<decltype(da), short>);
}

{ // autoとdecltypeに違いが出るパターン
    auto a = s0_ref;
    static_assert(std::is_same_v<decltype(a), short>);

    decltype(s0_ref) d = s0_ref; // dはリファレンス
    static_assert(std::is_same_v<decltype(d), short&>);

    decltype(auto) da = s0_ref; // daはリファレンス
    static_assert(std::is_same_v<decltype(da), short&>);
}

short s1{0};
{ // 微妙な違いで出るパターン
    auto a = s0 + s1;
    static_assert(std::is_same_v<decltype(a), int>);

    decltype(s0) d = s0 + s1; // これが意図的ならよいが
    static_assert(std::is_same_v<decltype(d), short>);

    decltype(s0 + s1) d2 = s0 + s1; // int&&にはならない
    static_assert(std::is_same_v<decltype(d2), int>);

    decltype(auto) da = s0 + s1; // この方がクローンがないため上よりも良い
    static_assert(std::is_same_v<decltype(da), int>);
}

```

- 演習-適切なautoの使い方

インスタンスの初期化

- 関数内のオブジェクトは、出来る限りAAAスタイルを用いて宣言し、同時に初期化する。
- 算術型の宣言にAAAスタイルが使えない場合、「代入演算子を伴わない一様初期化」を使用する。「代入演算子を伴う一様初期化」、「()、=による初期化」を使用しない。

```

// @@@ example/programming_convention/type_ut.cpp 572

int32_t a0(0);           // NG
int32_t a1 = 0;           // NG
int32_t a2{0};            // OK 一様初期化
int32_t a3 = {0};          // NG 代入演算子を伴う一様初期化
auto    a4 = 0;             // OK AAAの場合は一様初期を使わなくてても問題ない
auto    a5 = int32_t{0}; // OK AAA且つ一様初期

```

- リファレンスやポインタの宣言にAAAスタイルが使えない場合、「代入演算子を伴わない一様初期化」か「=による初期化」を使用する。「代入演算子を伴う一様初期化」、「()による初期化」を使用しない。

```

// @@@ example/programming_convention/type_ut.cpp 581

int32_t& r0(a0); // NG

```

```

int32_t& r1 = a0;      // OK
int32_t& r2{a0};       // OK 一様初期化
int32_t& r3 = {a0};    // NG 代入演算子を伴う一様初期化
auto&   r4 = a0;      // OK AAAの場合は一様初期を使わなくとも問題ないが、&の付け忘れに気を付ける

int32_t* p0(&a0);     // NG
int32_t* p1 = &a0;     // OK
int32_t* p2{&a0};     // OK 一様初期化
int32_t* p3 = {&a0};    // NG 代入演算子を伴う一様初期化
auto    p4 = &a0;      // OK AAAの場合は一様初期を使わなくとも問題ない
auto*   p5 = &a0;      // OK AAAの場合は一様初期を使わなくとも問題ない

```

- 構造体やクラス型オブジェクトの宣言にAAAスタイルが使えない場合、

「代入演算子を伴わない一様初期化」を使用する。

上記では意図したコンストラクタが呼び出せない場合にのみ「()による初期化」を使用する。ただし、`std::string`、`std::string_view`に関しては「=“xxx”」を使用しても良い。

```

// @@@ example/programming_convention/type_ut.cpp 609

// 構造体の初期化
struct Struct {
    int32_t    a;
    char const* str;
};

Struct s0{1, "1"};           // OK 代入演算子を伴わない一様初期化
Struct s1 = {2, "2"};        // NG 代入演算子による一様初期化
Struct s2{};                 // OK s2.aは0、s2.strはnullptrに初期化される。
Struct s3;                   // NG s3は未初期化
auto   s4 = Struct{1, "1"};  // OK AAAスタイル
auto   s5 = Struct{};        // OK AAAスタイル

// クラスの初期化
std::unique_ptr<Widget> a{std::make_unique<Widget>()};    // OK
std::unique_ptr<Widget> b{std::make_unique<Widget>()};    // NG {}を使うべき
auto          c{std::make_unique<Widget>()};    // OK
auto          d = std::make_unique<Widget>();    // OK
// このような場合、重複を避けるため、変数宣言の型はautoが良い

// std::string、std::string_viewの初期化
std::string str0{"222"};    // OK
std::string str1 = {"222"}; // NG =は不要
std::string str2("222");    // NG {}で初期化できない時のみ、()を使う。
std::string str3(3, '2');   // NG {}では初期化できない。str3 == "222"
std::string str4 = "222";    // OK 例外的に認める
auto       str5 = std::string{"222"}; // OK AAAスタイル

std::string_view sv0 = "222";           // OK 例外的に認める
auto       sv1 = std::string_view{"222"}; // OK AAAスタイル

// {}, ()による初期化の違い
std::vector<int32_t> vec0_i{1, 2, 3}; // OK vec0_i.size() == 3 && vec0_i[0] == 1 ...
std::vector<int32_t> vec1_i{10};        // OK vec1_i.size() == 1 && vec1_i[0] == 10
std::vector<int32_t> vec2_i(10);        // OK vec1_i.size() == 10
auto            vec3_i = std::vector<1, 2, 3>; // OK vec0_iと同じ

std::vector<std::string> vec1_s{10}; // OK vec1_s.size() == 10
std::vector<std::string> vec2_s(10); // NG vec2_s.size() == 10 {}を優先するべき
auto            vec3_s = std::vector<std::string>{10}; // OK vec1_sと同じ

// vec1_i、vec2_i、vec1_sの初期化は似ているが、結果は全く異なる。
// vec1_iは、vector(initializer_list<>)を呼び出す。
// vec2_iは、vector(int)を呼び出す。
// vec1_sは、vector(int)を呼び出す。

ASSERT_EQ(3, vec0_i.size());
ASSERT_EQ(1, vec1_i.size());
ASSERT_EQ(10, vec2_i.size());
ASSERT_EQ(vec0_i, vec3_i);
ASSERT_EQ(10, vec1_s.size()); // vec1_iと同じ形式で初期化したが結果は全く異なる。
ASSERT_EQ(10, vec2_s.size());
ASSERT_EQ(10, vec3_s.size());

```

- `decltype`によるオブジェクトの宣言は、AAAスタイルと同様に行う。

```

// @@@ example/programming_convention/type_ut.cpp 677

auto a = 0;
auto& b = a;

```

```

 decltype(a)    c = a;      // OKがautoの方が良い
 decltype(a)    d = {a};    // NG
 decltype(b)    e = a;      // OK
 decltype(auto) f = b;      // OK

 static_assert(std::is_same_v<decltype(c), int>);
 static_assert(std::is_same_v<decltype(d), int>);
 static_assert(std::is_same_v<decltype(e), int>);
 static_assert(std::is_same_v<decltype(f), int>);

```

- 配列の宣言には、「代入演算子を伴わない一様初期化」を使用する。char[]に関しては、「代入演算子を伴わない一様初期化」か「=xxx」を使用する。

```

// @@@ example/programming_convention/type_ut.cpp 700

int32_t array0[3]{1, 2, 3};      // OK 代入演算子を伴わない一様初期化
int32_t array1[3] = {1, 2, 3};   // NG 代入演算子による一様初期化
int32_t array2[3]{};
int32_t array3[3] = {};          // NG 代入演算子による一様初期化

char c_str0[]{'1', '2', '\0'};    // OKだが、非推奨
char c_str1[] = {'1', '2', '\0'};  // NG 代入演算子による一様初期化
char c_str2[] = {"12"};           // NG 代入演算子による一様初期化
char c_str3[]{"12"};             // OK
char c_str4[] = "12";            // OK

```

- 宣言時にポインタ変数の初期値が決まらない場合、nullptrで初期化する(「リテラル」参照)。

```

// @@@ example/programming_convention/type_ut.cpp 723

int32_t*      ptr1 = nullptr; // OK
int32_t*      ptr2{nullptr}; // OK
char const*   pchar0 = 0;    // NG
char*         pchar1 = NULL; // NG
int32_t const* ptr0(nullptr); // NG {}か=で初期化する

```

- 初期化順序が不定になるため、別のコンパイル単位で定義された静的なオブジェクトに依存した静的オブジェクトの初期化を行わない(同じファイルの上方にある静的なオブジェクトや、Singletonに依存した初期化を行うことには問題はない)。
- コンパイル時に値が確定する「基本型」や「コンストラクタがconstexprであるクラス」のインスタンスは、constexpr(「const/constexprインスタンス」参照)と宣言する。

```

// @@@ example/programming_convention/type_ut.cpp 735

constexpr int32_t f_constexpr(int32_t a) noexcept { return a * 3; }
int32_t        f_normal(int32_t a) noexcept { return a * 3; }

```

```

// @@@ example/programming_convention/type_ut.cpp 743

constexpr auto a = f_constexpr(3); // OK
auto const    b = f_constexpr(3); // NG constexprにできる
// constexpr auto c = f_normal(3); // NG コンパイルエラー
auto const d = f_normal(3); // OK

```

- constなオブジェクトが複雑な初期化を必要とする場合、その初期化にはラムダ式を使用する。

```

// @@@ example/programming_convention/type_ut.cpp 756

int32_t len{10}; // ここではlenは固定だが、関数引数等で外部から与えられるとする

auto vc0 = std::vector<int32_t>(len); // vc0が初期化以外で変更されないのであれば、NG
std::iota(vc0.begin(), vc0.end(), 1); // vc0の初期化

auto const vc1 = [len]() { // OK vc1の初期化
    std::vector<int32_t> ret(len);
    std::iota(ret.begin(), ret.end(), 1);
    return ret;
}(); // ラムダ式の生成と呼び出し

```

- 演習-インスタンスの初期化
- 演習-vector初期化
- 演習-ポインタの初期化

rvalue

- 関数の仮引数以外のリファレンスでrvalueをバインドしない(「オブジェクトのライフタイム」参照)。

- rvalueの内部ハンドルを使用しない(「[std::string_viewの使用制限](#)」参照)。

```
// @@@ example/programming_convention/type_ut.cpp 790

char const* str = std::string{"str"}.c_str();
// strが指すポインタはこの行では解放済

ASSERT_STREQ(str, "str"); // strは無効なポインタを保持であるため、未定義動作
```

- 非constなりファレンスでrvalueをバインドしない。

クラスとインスタンス

ファイルの使用方法

- 下記の例外を除き、一つのクラスはそれを宣言、定義する1つのヘッダファイルと、一つの.cppファイルによって構成する。
 - ファイル外部から使用されないクラスは、一つの.cppファイルの無名名前空間で宣言、定義する。
 - ファイル外部から使用されるインラインクラス(クラステンプレート等)は、一つのヘッダファイルで宣言、定義する。
 - 「一つのヘッダファイル(a.h)と、一つの.cpp(a.cpp)で構成されたクラスA」のみをサポートするクラス(Aのインターフェースや実装専用に定義されたクラス(「[Pimpl](#)」参照))は、a.h、a.cppで宣言、定義する。

クラスの規模

行数

- それ以外に方法がない場合を除き、ヘッダファイル内のクラスの定義、宣言はコメントを含め200行程度に収める。
- クラス内定義関数が大きくなると下記のような問題が発生しやすくなるため、10行を超える関数はクラス内で定義しない。
 - 関数のインポートする外部シンボルが多くなり、このクラスを使用する別のクラスに不要な依存関係を作ってしまう(「[インターフェース分離の原則\(ISP\)](#)」参照)。
 - クラスの定義が間延びして、クラスの全体構造を把握することが困難になる。

メンバの数

- それ以外に方法がない場合を除き、publicメンバ関数の数は、最大7個程度に収める(ただし、オーバーロードにより同じ名前を持つ関数群は、全部で1個とカウントする)。
- オブジェクトの状態を保持するメンバ変数の数は、最大4個程度に留める。constやconstexprメンバ・インスタンスは定数(状態を保持するメンバ変数ではない)であるため、この数に含めない。

凝集度

- 単なるデータホルダー(アプリケーションの設定データを保持するようなクラス等)や、ほとんどの振る舞いを他のクラスに委譲するようなクラスを除き、凝集度が高くなるように設計する。
- [演習-凝集度の意味](#)
- [演習-凝集度の向上](#)

アクセスレベルと隠蔽化

- アクセスレベルは、特別な理由がない限り、上からpublic、protected、privateの順番で明示する。

```
// @@@ example/programming_convention/class_ut.cpp 12

class A {
    void f0(); // NG デフォルト private を使用しない。
public:
    void f1(); // OK
private:
    void f2(); // OK
protected:
    void f3(); // NG privateの前に定義すべき。
};
```

- 全てのメンバ変数はprivateにする。
 - メンバ変数にアクセスしたい場合は、Accessorメンバ関数を経由させる(「[Accessor](#)」参照)。その場合でもsetterは控えめに使用する。

- 派生クラスから基底クラスの変数の値が必要になる場合は、protectedなAccessorを定義する。
- 単体テスト用クラスでは、protectedメンバ変数を定義してよい。
- アクセスレベルによるカプセル化が破壊されるため、メンバ変数のハンドル(リファレンスやポインタ)を返さない。それが避けがたい場合においては、constハンドルを返す。

```
// @@@ example/programming_convention/class_ut.cpp 28

class B {
public:
    ...
    void* f0() noexcept // NG メンバ変数が保持するポインタを返している
    {
        return v_ptr_;
    }

    std::string* f1() noexcept // NG メンバ変数へのポインタを返している
    {
        return &str_;
    }

    std::string& f2() noexcept // NG メンバ変数へのリファレンスを返している
    {
        return str_;
    }

    std::string f3() const noexcept // OK ただし、パフォーマンスに注意
    {
        return str_;
    }

    std::string const& f4() const noexcept // OK
    {
        return str_;
    }

private:
    void* v_ptr_ = nullptr;
    std::string str_{};
};
```

- 以下のような場合を除き、friendを使用しない。
- 単体テスト用クラス
- 二項演算子をオーバーロードした関数

```
// @@@ example/programming_convention/class_ut.cpp 68

class Integer {
public:
    Integer(int32_t integer) noexcept : integer_{integer} {}

    // メンバ関数に見えるが、非メンバ関数
    friend bool operator==(Integer lhs, Integer rhs) noexcept // OK
    {
        return lhs.integer_ == rhs.integer_;
    }

private:
    int32_t const integer_;
};

bool operator!=(Integer lhs, Integer rhs) noexcept { return !(lhs == rhs); }
```

- NVI(non virtual interface)に従う。従って、virtualな関数はprivateかprotectedと宣言し、それをpublicな非仮想メンバ関数から呼び出す。

```
// @@@ example/programming_convention/class_ut.cpp 86

class Widget {
public:
    virtual int32_t DoSomething() noexcept // NG virtualでpublic
    {
        ...
    }

    int32_t DoSomething(bool b) noexcept // OK non-virtualでpublic
    {
        return do_something(b);
    }
```

```

...
private:
    virtual int32_t do_something(bool b) noexcept // OK virtualでprivate
    {
        ...
    }
    ...
};


```

継承/派生

- 派生は最大2回程度までに留める。やむを得ず階層が深くなる場合、コードの静的解析等を使用し派生関係を明確にする(「[クラスのレイアウト](#)」参照)。
- 実装の継承よりも、包含、委譲を優先的に使用する。やむを得ず実装の継承を行う場合は、private継承を使用する。実装の継承をしたクラスがfinalでないならば、protected継承を使用する([CRTP\(curiously recurring template pattern\)](#)等は例外的に認められる)。

```

// @@@ example/programming_convention/class_ut.cpp 124

// private継承。非推奨
class StringWrapper0 final : private std::string {
public:
    explicit StringWrapper0(char const* str) : std::string(str) {}

    void AddStr(char const* str) { *this += str; }

    using std::string::c_str;
};

// 移譲。こちらを優先する
class StringWrapper1 final {
public:
    explicit StringWrapper1(char const* str) : str_(str) {}

    void AddStr(char const* str) { str_ += str; }

    char const* c_str() const noexcept { return str_.c_str(); }

private:
    std::string str_;
};

```

- 派生させないクラスは、finalと宣言する。ほとんどのクラスは派生しないはずなので、ほとんどのクラスはfinalになる。
- リソースリークの原因になり得るため、非virtualなデストラクタをもつクラスを継承しない。ただし、継承したクラスが基底クラスのメンバ変数以外のメンバ変数を持たないならば、継承しても良い。

```

// @@@ example/programming_convention/class_ut.cpp 149

class A { // デストラクタの呼び出しチェック用のクラス
public:
    A(bool& destructed) noexcept : destructed_{destructed} { destructed_ = false; }
    ~A() { destructed_ = true; }

private:
    bool& destructed_;
};

class BaseNG { // NG デストラクタが非virtual
public:
    BaseNG() = default;
};

class DerivedNG : public BaseNG {
public:
    DerivedNG(bool& destructed) : a_{std::make_unique<A>(destructed)} {}

private:
    std::unique_ptr<A> a_;
};

```

```

// @@@ example/programming_convention/class_ut.cpp 183

auto a_destructed = false;
{
    std::unique_ptr<DerivedNG> d{std::make_unique<DerivedNG>(a_destructed)};
    ASSERT_FALSE(a_destructed);
}
ASSERT_TRUE(a_destructed); // OK A::~A()が呼ばれたため問題ないが、、、

```

```

{
    std::unique_ptr<BaseNG> d{std::make_unique<DerivedNG>(a_destructed)};
    ASSERT_FALSE(a_destructed);
}
ASSERT_FALSE(a_destructed); // NG A::~A()が呼ばれないため、メモリリークする

```

```

// @@@ example/programming_convention/class_ut.cpp 201

class BaseOK { // OK デストラクタがvirtual
public:
    BaseOK() = default;
    virtual ~BaseOK() = default;
};

class DerivedOK : public BaseOK {
public:
    DerivedOK(bool& destructed) : a_{std::make_unique<A>(destructed)} {}

private:
    std::unique_ptr<A> a_;
};

```

```

// @@@ example/programming_convention/class_ut.cpp 221

auto a_destructed = false;
{
    std::unique_ptr<DerivedOK> d{std::make_unique<DerivedOK>(a_destructed)};
    ASSERT_FALSE(a_destructed);
}
ASSERT_TRUE(a_destructed); // OK A::~A()が呼ばれたため問題ない

{
    std::unique_ptr<BaseOK> d{std::make_unique<DerivedOK>(a_destructed)};
    ASSERT_FALSE(a_destructed);
}
ASSERT_TRUE(a_destructed); // OK A::~A()が呼ばれたため問題ない

```

インターフェースの継承

- クラス間に「Is-a」の関係が成り立つときに限りpublic継承を行う。
 - public継承を行う場合、[リスコフの置換原則\(LSP\)](#)を守る。
 - インターフェースを継承しない場合、public継承をしない。
- C#やJavaのinterfaceが必要ならば(インターフェースと実装の完全分離をしたい場合等)、pure-virtualなメンバ関数のみを宣言したクラス(もしくはそのクラスに[NVI\(non virtual interface\)](#)を適用したクラス)を定義する。

多重継承

- それが不可避でない限り、多重継承は使用しない。
- 多重継承を使用する場合、複数個の基底クラスのうち、一つを除きメンバ変数を持ってはならない。
- 多重継承を使用する場合、継承階層内に同じ基底クラスが複数回出てきてはならない(ダイヤモンド継承をしない)。
- やむを得ずダイヤモンド継承をせざるを得ない場合、継承階層内に複数回出現する基底クラスにはvirtual継承を行う。
- virtual継承を行ったクラスのコンストラクタからは、virtual基底クラスのコンストラクタを呼び出すようにする(こうしないとvirtual基底クラスは初期化されない)。

非静的なメンバ変数/定数の初期化

- [定義] 非静的なメンバ変数の初期化には下記の3つの方法がある。
 - 初期化方法 0: 非静的メンバ変数の初期化子による初期化([NSDMI](#))
 - 初期化方法 1: コンストラクタの非静的メンバ初期化子による初期化
 - 初期化方法 2: コンストラクタ内での非静的メンバ変数の初期値の代入

```

// @@@ example/programming_convention/class_ut.cpp 243

class A0 {
public:
    A0() noexcept : b_{0} // 初期化方法 1
    {
        c_ = 0; // 初期化方法 2
    }

private:
    int32_t a_{0}; // 初期化方法 0
};

```

```
    int32_t b_;
    int32_t c_;
};
```

- [注意] 初期化方法 0 と、初期化方法 1 が同一変数に行われた場合、初期化方法 0 は実行されない。
- すべての非静的なメンバ変数は、コンストラクタ終了時までに明示的に初期化する。
- 多くのコンパイラや静的解析ツールは、初期化方法 0、1 による初期化の漏れについては容易に発見、指摘できるが、初期化方法 2 による初期化の漏れについては、発見が難しい場合がある。また、constメンバ変数は、初期化方法 2 では初期化できないため、それ以外に方法がない場合を除き、初期化方法 2 の使用を避ける。
- 非静的なメンバ変数はクラス内で定義された順序に従い初期化されるため、初期化方法 1 で羅列される初期化の順序を定義順序と同じにする（初期化方法 1 での初期化の順序は、実際の初期化の順序とは関係がない）。

```
// @@@ example/programming_convention/class_ut.cpp 262

class A_NG {
public:
    A_NG(int32_t x, int32_t y) noexcept
        : b_{x + y}, a_{x}, c_{y} // NG メンバ変数定義と初期化の順序が違う
    {
    }

private:
    int32_t a_;
    int32_t b_;
    int32_t c_;
};

class A_OK {
public:
    A_OK(int32_t x, int32_t y) noexcept : a_{x}, b_{x + y}, c_{x} // OK
    {
    }

private:
    int32_t a_;
    int32_t b_;
    int32_t c_;
};
```

- クラスがただ一つのコンストラクタを持つ場合、初期化方法 0 と初期化方法 1 を混在させない。従って、初期化方法 1 を必要とするメンバ変数が一つでもある場合は、すべての変数の初期化を初期化方法 1 で行う。

```
// @@@ example/programming_convention/class_ut.cpp 291

class A1 {
public:
    A1() noexcept {} // OK 初期化方法 0 に統一

private:
    int32_t const a_{1}; // OK 初期化方法 0 による初期化。
                        // ただし、static constexpr にすべき。
    int32_t b_[2]{0, 1}; // OK 初期化方法 0 による初期化
    int32_t c_{5};      // OK 初期化方法 0 による初期化
};

class A2 {
public:
    explicit A2(int a) noexcept // OK 初期化方法 1 に統一
        : a_{a}, b_{0, 1}, c_{5} // OK 初期化方法 1 による初期化
    {
    }

private:
    int32_t const a_;
    int32_t      b_[2];
    int32_t      c_;
};

class A3 {
public:
    explicit A3(int a) noexcept : a_{a} // NG 初期化方法の混在
    {
        c_ = 5; // NG 初期化方法 0、1 使用可能にもかかわらず初期化方法 2 を使用している
    }

private:
    int32_t const a_;
    int32_t      b_[2]{0, 1}; // NG 初期化方法の混在
```

```
    int32_t      c_;
};
```

- クラスが複数のコンストラクタを持つ場合、すべてのメンバ変数に対して初期化方法 0 を行い(デフォルト値の設定)、 デフォルト値とは異なる初期値を持つ変数に対してのみ、コンストラクタ毎に初期化方法 1 を行う。

```
// @@@ example/programming_convention/class_ut.cpp 331

class A4 {
public:
    A4() noexcept {} // OK 初期化方法 0

    A4(int32_t e) noexcept : e_{e} {} // OK 初期化方法 1 によるe_の上書き
    // 注) A4()とA4(int32_t)はデフォルト引数を使用すれば統一できるが、
    // 例の単純化のためにあえてそれぞれを定義している。

private:
    int32_t d_{5}; // OK 初期化方法 0 による初期化
    int32_t e_{0}; // OK 初期化方法 0 による初期化
};
```

- 演習-メンバ変数の初期化方法の選択
- 演習-メンバの型
- 演習-メンバ変数の初期化

静的なメンバ変数/定数の初期化

- 静的な(且つconstexprでない)メンバ変数は、ヘッダファイルで宣言し、.cppで定義、初期化する。
- クラス宣言内で初期化される基本型のstatic constメンバ定数を定義しない。代わりに、static constexprメンバ定数として定義、初期化する(クラス宣言外で初期化されるメンバ定数はstatic constにする)。
- privateなstatic constexprメンバ定数は、そのクラスが宣言されているヘッダファイル内で依存されている場合のみ使用する。
- privateなstatic constexprメンバ定数がヘッダファイル内で依存されていない場合は、.cppの無名名前空間内で定義、初期化する(つまり、クラスのメンバとして定義しない。こうすることで不要なコンパイルが防げる)。

```
// @@@ example/programming_convention/class.h 5

class StaticConstexprVar {
public:
    StaticConstexprVar() = default;

    uint32_t MultiplyBy2(uint32_t a) noexcept;
    uint32_t MultiplyBy3(uint32_t a) noexcept { return static_constexpr_var_3 * a; }
    uint32_t MultiplyBy4(uint32_t a) noexcept;

private:
    static constexpr uint32_t static_constexpr_var_2{2}; // NG クラス内で定義する必要なし
    static constexpr uint32_t static_constexpr_var_3{3}; // OK クラス内で定義する必要あり
};
```

```
// @@@ example/programming_convention/class_ut.cpp 351
//
uint32_t StaticConstexprVar::MultiplyBy2(uint32_t a) noexcept { return static_constexpr_var_2 * a; }

namespace {
constexpr uint32_t static_constexpr_var_4{4}; // OK クラス内で定義する必要なし
}

uint32_t StaticConstexprVar::MultiplyBy4(uint32_t a) noexcept { return static_constexpr_var_4 * a; }
```

mutableなメンバ変数

- 排他制御用(std::mutex等)や計算データのキャッシュ用等のメンバ変数を除き、メンバ変数をmutableと宣言しない。

```
// @@@ example/programming_convention/class_ut.cpp 365

class A {
public:
    A() = default;

    uint32_t GetValue() const // OK GetValue()をconstにするためにmutex_はmutable
    {
        auto lock = std::lock_guard{mutex_}; // constでない関数std::mutex::lock()の呼び出し

        return v_;
    }
};
```

```

void AddValue(uint32_t v) noexcept
{
    auto lock = std::lock_guard{mutex_};

    v_ += v;
}

private:
    mutable std::mutex mutex_; // OK
    uint32_t          v_{0};
};

```

スライシング

- オブジェクトのスライシングには以下のいずれかで対処する。
 - Clone(仮想コンストラクタ)を使用する。
 - copy代入演算子を= deleteする。
- スライシングと類似の問題が起こるため、オブジェクトの配列をそのオブジェクトの基底クラスへのポインタに代入しない。
- 演習-スライシング

オブジェクトの所有権

- オブジェクトaの所有権（「オブジェクトの所有権」参照）を持つオブジェクトもしくは関数は、オブジェクトaの解放責務を持つ。
- オブジェクトaの所有権を持たないオブジェクトは、オブジェクトaのハンドルをメンバ変数で保持することを出来る限り避ける（Observerパターン等、このルール順守が困難な場合は多い）。
- オブジェクトaがオブジェクトbにnewで生成された（もしくはstd::make_unique()で生成された）とすると、
 - オブジェクトaのポインタはstd::unique_ptr<（「RAII(scoped guard)」参照）で保持する。
 - オブジェクトa（正確にはオブジェクトaを管理するstd::unique_ptr<オブジェクト>）の所有権は、オブジェクトbが保持する。
 - オブジェクトbはオブジェクトaの解放責務を持つ（std::unique_ptr<>による自動解放）。
 - オブジェクトaの所有権を保持していないオブジェクトは、オブジェクトaを解放してはならない。
 - オブジェクトaの所有権を別のオブジェクトxへ移動させる場合、std::unique_ptr<>とstd::move()を使用する。
- 演習-オブジェクトの所有権

オブジェクトのライフタイム

- オブジェクトのライフタイム開始前、もしくは終了後のオブジェクトにアクセスしない。

```

// @@@ example/programming_convention/class_ut.cpp 419

// 初期化前にオブジェクトにアクセスしてしまう例
B& getB() noexcept;

// bが初期される前に（ライフタイム開始前）に、a.A::A()が呼び出される。
// a.A::A()が呼び出される前に、getB()が呼び出される。
// 従って、bが初期化される前にgetB()が未初期化のbのリファレンスを返してしまう。
A a{getB()};

B b;
B& getB() noexcept { return b; }

```

- オブジェクトaが所有権を持たないオブジェクトbへのハンドルをa自体のメンバ変数で保持する場合、オブジェクトbのライフタイムが終了する前に、オブジェクトaがオブジェクトbにアクセスできないようにする（Observerパターンのdetachメンバ関数等）。

```

// @@@ example/programming_convention/class_ut.cpp 435

// ライフタイムが終了したオブジェクトにアクセスしてしまう例
auto a = A{};
{
    auto b = B{};
    a.SetB(&b); // NG aのメンバ変数へ&bを代入。
}           // この行でbのライフタイム終了。

a.DoSomething(); // NG bのポインタを使用して何かすると不定動作。

```

- スタック上のオブジェクトのハンドルをその関数外部へ開示しない（そのハンドルは、ライフタイムが終了したオブジェクトを指している）。
- thread_localオブジェクトは、ログやデバッグ用途のみで使用する。

- rvalueはライifetime終了間際のオブジェクトであるため、関数の仮引数以外のリファレンスでrvalueをバインドしない(特にリファレンス型のメンバ変数でrvalueをバインドしないことは重要である)。rvalueをリファレンス型の引数で受け取る場合はconstリファレンス、もしくはrvalueリファレンス(T&&)を使用する(「rvalue」参照)。

```
// @@@ example/programming_convention/class_ut.cpp 466
void f0(E& noexcept;
void f1(E const& noexcept;
void f2(E&& noexcept;

// @@@ example/programming_convention/class_ut.cpp 475
// f0(E{}); NG ほとんどのコンパイラではエラー
f1(E{}); // OK rvalueはconstリファレンスにバインド可
f2(E{}); // OK rvalueはrvalueリファレンス

E const& a0 = E{"4"}; // NG rvalueを引数以外のconstリファレンスに代入
E&&      a1 = E{"5"}; // NG rvalueを引数以外のrvalueリファレンス
```

非メンバ関数/メンバ関数

非メンバ関数

- 下記のような関数を除き、グローバル名前空間に非メンバ関数を定義しない。
 - C言語から呼び出される関数
 - アセンブラー関数
- .cppファイルから、そのファイルの外部で定義された関数を呼び出す場合、その.cppファイル内での局所的な関数宣言をしない(関数が宣言、定義されているヘッダファイルをインクルードする)。
- コンパイル時に戻り値が確定する関数はconstexprと宣言する。
- 演習-非メンバ関数の宣言

メンバ関数

- 可能な場合(メンバに直接アクセスしない場合)、メンバ関数をstaticにする。
- コンパイル時に戻り値が確定するメンバ関数はconstexprと宣言する。
- オブジェクトの状態を変えないメンバ関数は、constと宣言する。
 - getter(下記の例ではGetString)はconstと宣言する。
 - 下記のSetPtrのような関数はconstにしない。

```
// @@@ example/programming_convention/func_ut.cpp 19
class A {
public:
    A() : s_ptr_{std::make_unique<std::string>("haha")}, s_inst_{"hihi"} {}

    std::string const& GetString() const noexcept // OK 必ずconst
    {
        return s_inst_;
    }

    // SetPtrと、SetInstは実質的には同じことを行っている。
    // SetInstはconstと宣言できない(コンパイルエラー)。
    // 従ってSetPtrもconstと宣言してはならない。
    // なお、この問題はstd::experimental::propagate_constを使用することで解決できるが、
    // 名前空間からわかるように、このライブラリが将来にわたって有効かどうかは不明である。
    void SetPtr(std::string_view name) const // NG このconstはつけてはならない。
    {
        *s_ptr_ = name;
    }

    void SetInst(std::string_view name) // OK
    {
        s_inst_ = name;
    }
};

private:
    std::unique_ptr<std::string> s_ptr_;
    std::string                 s_inst_;
```

```

};

void f()
{
    A const a;

    a.SetPtr("0"); // constオブジェクトaを変更できてしまう。
// a.SetInst("1"); // constオブジェクトaを変更しようとしたため、
// 正しく(constの目的通り)コンパイルエラー。
}

```

- クラス内部のハンドル（ポインタやリファレンス）を戻り値に使用しない。
 - それが避けがたい場合は、戻り値のハンドルをconstにする。
 - ハンドルがconstにできない場合(関数が非constなハンドルを返す場合)、そのハンドル経由でクラスの状態を変更できるため、その関数をconstにしない。

```

// @@@ example/programming_convention/func_ut.cpp 62

class B {
public:
    B() noexcept {}

    // GetStringsは、避けがたい理由で、strings_のリファレンスを返さざるを得ないとする。
    // この場合、GetStringsはconstにしてはならない。
    std::vector<std::string>& GetStrings() noexcept { return strings_; }

private:
    std::vector<std::string> strings_{};
};

```

- 非静的メンバのハンドルを返すメンバ関数を持つオブジェクトが`rvalue`である場合、そのオブジェクトからその関数を呼び出した戻り値(メンバへのハンドル)を変数で保持しない(そのハンドルは無効なオブジェクトを指している)。そういうたった使用方法が必要ならば`lvalue`修飾、`rvalue`修飾を用いたオーバーロード関数を定義する。

```

// @@@ example/programming_convention/func_ut.cpp 82

char const* s = std::string{"hehe"}.c_str(); // std::string{"hehe"}はrvalue
std::cout << s << std::endl; // この時点ではsは解放されている。

```

- 演習-メンバ関数の修飾

特殊メンバ関数

- 特殊メンバ関数
 - デフォルトコンストラクタ
 - copyコンストラクタ
 - copy代入演算子(operator =)
 - moveコンストラクタ
 - move代入演算子
 - デストラクタ

について、デフォルトコンストラクタ以外を定義する場合、その他の関数には以下のいずれかを選択する(不要な代入演算子を`= delete`することは特に重要である)。

コンパイラ生成関数を	定義方法
使用する	<code>= default</code>
使用しない	<code>= delete</code> 、もしくは自分で実装

- クラスを宣言、定義する場合、下記のClassStationery(クラスのひな形)を参考にし、不要なコンパイラ生成関数が作られないようにする。

```

// @@@ example/programming_convention/class_ut.cpp 489

/// @class ClassStationery
/// @brief クラスのひな形。クラスを定義、宣言するときには、このクラスの下記6関数を適切に
/// 定義、宣言すること。
class ClassStationery final {
public:
    ClassStationery() = delete;
    ~ClassStationery() = delete;

    ClassStationery(ClassStationery const&) = delete;

```

```

    ClassStationery& operator=(ClassStationery const&)      = delete;
    ClassStationery(ClassStationery&&) noexcept           = delete;
    ClassStationery& operator=(ClassStationery&&) noexcept = delete;
};


```

- リソース管理等の都合からコンパイラが生成するデストラクタでは機能が不十分な場合、プログラマがそのクラスのデストラクタを定義する。この場合、コンパイラが生成するcopyコンストラクタ、copy代入演算子、moveコンストラクタ、move代入演算子では機能が不十分であることが予測されるため、これらを使用しない(「[Copy-And-Swap](#)」参照)。

- [演習-特殊メンバ関数の削除](#)

コンストラクタ

- クラスが複数の初期化方法を提供する場合でも、デフォルト引数を使用し、できる限りコンストラクタを一つに集約する。
- 一つのコンストラクタに集約できない場合、[委譲コンストラクタ等](#)により処理の重複を防ぐ。非静的なメンバ変数/定数の初期化処理の重複を避けることは特に重要である。
- オブジェクトの初期化が完了するまでは派生クラスの仮想関数呼び出し等の[RTTI](#)機能を使うことはできないため(「[クラスのレイアウト](#)」参照)、コンストラクタの中でRTTI機能を使わない(デストラクタでも同様)。

```

// @@@ example/programming_convention/func_ut.cpp 95

class Base {
public:
    Base(std::ostream& os) : os_{os} { os_ << __func__ << "-" << Name() << " -> "; }

    virtual ~Base() { os_ << __func__ << "-" << Name(); }

    virtual std::string_view Name() const { return "Base"; }

protected:
    std::ostream& os_; // protectedなメンバ変数を定義すべきではないが、コードの動作例示のため
};

class Derived : public Base {
public:
    Derived(std::ostream& os) : Base{os} { os_ << __func__ << "-" << Name() << " -> "; }

    virtual ~Derived() { os_ << __func__ << "-" << Name() << " -> "; }

    virtual std::string_view Name() const override { return "Derived"; }
};

```

```

// @@@ example/programming_convention/func_ut.cpp 122

auto oss = std::ostringstream{};

{
    auto d = Derived{oss};
}

ASSERT_EQ("Base-Base -> Derived-Derived -> ~Derived-Derived -> ~Base-Base", oss.str());
// つまり、
//     Base::Base()とBase::~Base()からは、Base::Name()が
//     Derived::Derived()とDerived::~Derived()からは、Derived::Name()
//     が呼び出される。

```

- クラスが解放責務を持つポインタ型メンバ変数を持つならば、copyコンストラクタ、copy代入演算子に対して以下のいずれかを行い、[シャローコピー](#)が行われないようにする(このルールはファイルディスクリプタ等のリソース管理をするクラス全般に当てはまる)。
 - ディープコピーをさせる。
 - = deleteする(「[特殊メンバ関数](#)」参照)。

またこの場合、moveコンストラクタ、move代入演算子の定義を検討する(「[Copy-And-Swap](#)」参照)。

- 非explicitなコンストラクタによる暗黙の型変換が不要なクラスのコンストラクタに関しては、下記の目的のためにexplicitと宣言する。
 - 仮引数一つのコンストラクタに関しては、暗黙の型変換が行われないようにする。
 - 仮引数二つ以上のコンストラクタに関しては、代入演算子での[一様初期化](#)ができないようにする。

```

// @@@ example/programming_convention/func_ut.cpp 144

class A0 {
public:
    // NG int32_tからA0への暗黙の型変換が起こる。
    A0(int32_t a) noexcept : a_{a} {}
};

```

```

    ...
};

class A1 {
public:
    // OK int32_tからA1への暗黙の型変換をさせない。
    explicit A1(int32_t a) noexcept : a_{a} {}
    ...
};

class A2 {
public:
    // NG 代入演算子でのリスト初期化ができてしまう。
    A2(int32_t a, int32_t* b) noexcept : a_{a}, b_{b} {}
    ...
};

class A3 {
public:
    // OK 代入演算子でのリスト初期化をさせない。
    explicit A3(int32_t a, int32_t* b) noexcept : a_{a}, b_{b} {}
    ...
};

void f_A0(A0) noexcept {}
void f_A1(A1) noexcept {}
void f_A2(A2) noexcept {}
void f_A3(A3) noexcept {}

```

```

// @@@ example/programming_convention/func_ut.cpp 198

A0 a0 = 1;           // NG 1からA0への暗黙の型変換。
                     // このような変換はセマンティクス的不整合につながる場合がある。
// A1 a1 = 1;         // OK explicitの効果で、意図通りコンパイルエラー。

f_A0(1);            // NG 1からA0への暗黙の型変換のためf_A0が呼び出せてしまう。
#if 0
    f_A1(1);          // OK explicitの効果で、意図通り以下のようなコンパイルエラー。
                     // error: could not convert '1' from 'int' to 'A1'
#else
    f_A1(A1{1});      // OK f_A1の呼び出し
#endif

auto i = 3;
A2 a2 = {i, &i};    // NG 代入演算子でのリスト初期化をしている。

// A3 a3 = { i, &i }; // OK explicitの効果で、意図通りコンパイルエラー。
A3 a3{i, &i};       // OK リスト初期化
auto a4 = A3{i, &i}; // OK AAA

f_A2({i, &i});      // NG { i, &i }からA2への暗黙の型変換のためf_A2が呼び出せてしまう。
#if 0
    f_A3({i, &i});  // OK explicitの効果で、意図通り以下のようなコンパイルエラー。
                     // error: converting to A3 from initializer list would use explicit
                     //           constructor A3::A3(int32_t, int32_t*)
#else
    f_A3(A3{i, &i}); // OK f_A3の呼び出し
#endif

```

- ・派生クラスが基底クラスの全コンストラクタを必要とする場合、継承コンストラクタを使用する。
- ・デフォルト引数はインターフェース関数の呼び出しを簡略化する目的で使用するべきであるため、`private`関数にデフォルト引数を持たない。

・演習-委譲コンストラクタ

`copy`コンストラクタ、`copy`代入演算子

- ・`copy`コンストラクタ、`copy`代入演算子はcopyセマンティクスに従わせる。
- ・`copy`コンストラクタ、`copy`代入演算子の引数は`const`リファレンスにする。
- ・RVO(Return Value Optimization)により、`copy`コンストラクタの呼び出しは省略されることがあるため、`copy`コンストラクタ、`copy`代入演算子はコピー以外のことをしない。
- ・`copy`代入演算子はlvalue修飾をする。

```

// @@@ example/programming_convention/func_ut.cpp 239

class Widget {

```

```

public:
    Widget& operator=(Widget const& rhs) // NG lvalue修飾無し
    {
        // 何らかの処理
        return *this;
    }

    Widget& operator=(Widget&& rhs) noexcept // NG lvalue修飾無し
    {
        // 何らかの処理
        return *this;
    }
    ...
};

```

// @@@ example/programming_convention/func_ut.cpp 268

```

Widget w0{1};
Widget w1{2};

w0 = w1;          // これには問題ない
w1 = Widget{3}; // これにも問題ない

Widget{2} = w0; // NG lvalue修飾無しのcopy代入演算子であるため、コンパイルできる
Widget{3} = Widget{4}; // NG lvalue修飾無しのmove代入演算子であるため、コンパイルできる

```

// @@@ example/programming_convention/func_ut.cpp 287

```

class Widget { // 上記の修正
public:
    Widget& operator=(Widget const& rhs) & // OK lvalue修飾
    {
        // 何らかの処理
        return *this;
    }

    Widget& operator=(Widget&& rhs) & noexcept // OK lvalue修飾
    {
        // 何らかの処理
        return *this;
    }
    ...
};

```

// @@@ example/programming_convention/func_ut.cpp 316

```

Widget w0{1};
Widget w1{2};

// Widget{2} = w0;      lvalue修飾の効果でコンパイルエラー
// Widget{3} = Widget{4}; lvalue修飾の効果でコンパイルエラー

```

- 演習-copyコンストラクタ

moveコンストラクタ、move代入演算子

- moveコンストラクタ、move代入演算子はmoveセマンティクスに従わせ、エクセプションをthrowさせない(「エクセプション処理」参照)。
- [注意] noexceptでないmoveコンストラクタ、move代入演算子を持つクラスをSTLコンテナのtemplate引数として使用した場合、moveコンストラクタ、move代入演算子の代わりに、copyコンストラクタ、copy代入演算子が使用され、パフォーマンス問題を引き起こす場合がある。
- move代入演算子はlvalue修飾(「copyコンストラクタ、copy代入演算子」参照)をする。
- 演習-moveコンストラクタ

初期化子リストコンストラクタ

- 初期化子リストコンストラクタは、コンテナクラスの初期化のためのみに定義する。
- 初期化子リストコンストラクタと同じ仮引数を取り得るコンストラクタを定義しない。

デストラクタ

- デストラクタの中でRTTI機能を使わない(「コンストラクタ」参照)。

- デストラクタは`noexcept`であり、`throw`するとプログラムが終了するため、デストラクタで`throw`しない。

オーバーライド

- オーバーライドとオーバーロードの違いに注意する。
- オーバーライドしたメンバ関数には、オーバーライドされたメンバ関数の機能の意味を踏襲させる。
- オーバーライドする/される一連の仮想関数(デストラクタを含む)について、
 - 全ての宣言には`virtual`を付ける。
 - 一連の仮想関数の最初でも最後でもないものの宣言には、`override`を付ける。
 - 一連の仮想関数の最後のものの宣言には、`override`を付けず、`final`を付ける。

```
// @@@ example/programming_convention/func_ut.cpp 334

class Base {
public:
    virtual ~Base(); // OK
    virtual void f(int32_t) noexcept; // OK
    virtual void g() noexcept; // OK
};

// Derived_0::fは、Base::fのオーバーライドのつもりであったが、タイプのため新たな関数の宣言、
// 定義になってしまった。この手のミスは、自分で気づくのは難しい
class Derived_0 : public Base {
public:
    virtual ~Derived_0(); // NG overrideが必要
    virtual void f(uint32_t) noexcept; // NG Derived_0::fはBase::fのオーバーライドではない
    virtual void g() noexcept override; // OK
};

class Derived_1 : public Base {
public:
    // NG 下記が必要
    // virtual ~Derived_1() override;
    virtual void f(uint32_t) override; // OK overrideと書いたことで、
                                    // コンパイルできないため、タイプに気づく
};

class Derived_2 : public Base {
public:
    virtual ~Derived_2() override; // OK Derived_2はfinalではない
    virtual void f(int32_t) noexcept override final; // NG overrideは不要
    virtual void g() noexcept final; // OK これ以上オーバーライドしない
};
```

- オーバーライド元の関数とそのオーバーライド関数のデフォルト引数の値は一致させる。オーバーライド元の関数にデフォルト引数を持たせないのであれば、そのオーバーライド関数にもデフォルト引数を持たせない。

```
// @@@ example/programming_convention/func_ut.cpp 373

class Base {
public:
    virtual int32_t GetArg(int32_t a = 0) const noexcept { return a; }
    ...
};

class Derived : public Base {
public:
    // NG Base::GetArgのデフォルト引数と違う
    virtual int32_t GetArg(int32_t a = 1) const noexcept override { return a; }

    ...
};
```

```
// @@@ example/programming_convention/func_ut.cpp 397

auto d = Derived{};
Base& b{d};

// 同じオブジェクトであるにもかかわらず、その表層型でデフォルト引数の値が変わってしまう。
ASSERT_EQ(0, b.GetArg());
ASSERT_EQ(1, d.GetArg());
```

- privateやprotectedなオーバーライド関数にはデフォルト引数を持たさない(「実引数/仮引数」参照)。さらにNVI(non virtual interface)にも従うことにより、上の条項の示した一連のオーバーライド関数のデフォルト引数の一貫について考慮の必要がなくなり、且つこのクラスのユーザはデフォルト引数が使用できるようになる。

- 演習-オーバーライド関数の修飾

非メンバ関数/メンバ関数共通

サイクロマティック複雑度

- サイクロマティック複雑度は15以下が好ましい。
- 特別な理由がない限り、サイクロマティック複雑度は20以下にする。

行数

- 10行程度が好ましい。
- 特別な理由がない限り、30行以下で記述する。
- [注意] C++の創始者であるビャーネ・ストラウストラップ氏は、プログラミング言語C++ 第4版の中で、下記のように述べている。

約 40 行を関数の上限にすればよい。
私自身は、もっと小さい平均 7 行程度を理想としている。

- 演習-関数分割

オーバーロード

- オーバーライドとオーバーロードの違いに注意する。
- オーバーロードされた関数は実行目的を同じにする。異なる目的のためには異なる名前の関数を用意する。
- オーバーライドを除き、基底クラスのメンバ関数と同じ名前を持つメンバ関数を派生クラスで宣言、定義しない。これに反すると name-hidingのため、基底クラスのメンバ関数の可視範囲を縮小させてしまう。

```
// @@ example/programming_convention/func_ut.cpp 411

// NGな例
class Base {
public:
    virtual ~Base() = default;
    void f() noexcept
    {
        ...
    }
};

class DerivedNG : public Base {
public:
    void f(int32_t a) noexcept // NG DerivedNG::f(int32_t)がBase::fを隠す(可視範囲の縮小)
    {
        ...
    }
};

void f() noexcept
{
    auto d = DerivedNG{};
    d.f(0);
#if 0
    d.f(); // NG DerivedNG::f(int32_t)がBase::fを隠す(可視範囲の縮小)ためコンパイルエラー
#endif
}

// DerivedNGの修正
class DerivedOK : public Base {
public:
    void f(int32_t a) noexcept
    {
        ...
    }
    using Base::f; // OK Base::fをDerivedOKに導入。
};

void g() noexcept
{
    auto d = DerivedOK{};
    d.f(0);
```

```
    d.f(); // OK usingにより、Base::fが見える
}
```

- 仮引数の型が互いに暗黙に変換できるオーバーロード関数の定義、使用には気を付ける。

```
// @@@ example/programming_convention/func_ut.cpp 468
```

```
int32_t f(int32_t) { return 0; }
int32_t f(int16_t) { return 1; }
```

```
// @@@ example/programming_convention/func_ut.cpp 476
```

```
auto i16 = int16_t{1};

ASSERT_EQ(1, f(i16)); // f(int16_t)が呼ばれる
ASSERT_EQ(0, f(i16 + i16)); // f(int32_t)が呼ばれる
```

- 暗黙の型変換による関数の使用範囲の拡張を防ぐには、オーバーロード関数を`= delete`する。

```
// @@@ example/programming_convention/func_ut.cpp 488
```

```
// 実引数がdoubleを認めないパターン
int32_t f0(double) = delete;
int32_t f0(int32_t a) noexcept { return a / 2; }

// 実引数がint32_t以外を認めないパターン
template <typename T>
int32_t f1(T) = delete;
int32_t f1(int32_t a) noexcept { return a / 2; }

// 実引数がunsigned以外を認めないパターン
template <typename T, std::enable_if_t<!std::is_unsigned_v<T>>* = nullptr>
uint64_t f2(T t) = delete;

template <typename T, std::enable_if_t<std::is_unsigned_v<T>>* = nullptr>
uint64_t f2(T t) noexcept
{
    uint64_t f2_impl(uint64_t) noexcept;

    // Tがsignedで、tが-1のような値の場合、f2_impl(uint64_t)の呼び出しによる算術変換により、
    // tは巨大な値に変換されるが、
    // Tがunsignedならば、f2_impl(uint64_t)の呼び出しによる算術変換は安全

    return f2_impl(t);
}
```

```
// @@@ example/programming_convention/func_ut.cpp 518
```

```
char c{'c'};
int8_t i8{1};
int32_t i32{1};
uint32_t ui32{1};
uint64_t ui64{1};
double d{1.0};

f0(c);
f0(i8);
f0(i32);
// f0(d); 呼び出そうとしたf0(double)はdeleteされているので意図通りエラー

f1(i32);
// f1(ui32); 呼び出そうとしたf1<uint32_t>(uint32_t)はdeleteされているので意図通りエラー

f2(ui32);
f2(ui64);
// f2(c); 呼び出そうとしたf2<char>(char)はdeleteされているので意図通りエラー
// f2(i32); 呼び出そうとしたf2<int32_t>(int32_t)はdeleteされているので意図通りエラー
```

- 演習-オーバーライド/オーバーロード
- 演習-オーバーロードによる誤用防止

演算子オーバーロード

- 演算子をオーバーロードする場合、
 - 単項演算子はメンバ関数で定義する。
 - 二項演算子は非メンバ関数で定義する。
- `&&`, `||`, カンマ`(,)`をオーバーロードしない。

- 型変換オペレータの宣言、定義を多用しない。
- boolへの型変換オペレータは、explicit付きで定義する。

```
// @@@ example/programming_convention/func_ut.cpp 547

class A0 {
public:
    operator bool() const noexcept // NG intへの型変換が可能
    {
        return state_;
    }

private:
    bool state_{true};
};

class A1 {
public:
    explicit operator bool() const noexcept // OK explicitすることで誤使用を避ける。
    {
        return state_;
    }

private:
    bool state_{true};
};

void f()
{
    auto a0 = A0{};
    auto a1 = A1{};

    std::cout << a0 + 1; // NG コンパイルできてしまう。
#ifndef 0
    std::cout << a1 + 1; // OK 意図通りコンパイルエラー
#endif

    ...
}
```

- 演算子をオーバーロードする場合、それが自然に使えるようにする。
 - operator == を定義するならば、operator != も定義する (<, >等のその他の例も同様)。
 - operator+ を定義するならば、operator += も定義する(+以外も同様)。
 - copy(またはmove)代入演算子を定義する場合、copy(またはmove)コンストラクタも定義する(その際、コードクローンを作りがちな
ので注意する(「[Copy-And-Swap](#)」参照))。

```
// @@@ example/programming_convention/func_ut.cpp 586

class Integer {
public:
    Integer(int32_t integer) noexcept
        : integer_{integer} {} // int32_tの暗黙の型変換が必要なのでexplicitしない

    // copyコンストラクタ、copy代入演算子の定義
    Integer(Integer const&) = default;
    Integer& operator=(Integer const&) = default;

    friend bool operator==(Integer lhs, Integer rhs) noexcept
    {
        return lhs.integer_ == rhs.integer_;
    }

    Integer& operator+=(Integer rhs) noexcept
    {
        integer_ += rhs.integer_;
        return *this;
    }

private:
    int32_t integer_;
};

inline bool operator!=(Integer lhs, Integer rhs) noexcept
{
    return !(lhs == rhs); // operator==の活用
}

inline Integer operator+(Integer lhs, Integer rhs) noexcept
```

```

    {
        lhs += rhs; // operator+=の活用
        return lhs;
    }

```

- ユーザ定義リテラル演算子のサフィックスには、アンダーバーから始まる3文字以上の文字列を使用する。

```

// @@@ example/programming_convention/func_ut.cpp 628

constexpr int32_t one_km{1000};

// ユーザ定義リテラル演算子の定義
constexpr int32_t operator""_kilo_meter(unsigned long long num_by_mk) // OK
{
    return num_by_mk * one_km;
}

constexpr int32_t operator""_km(unsigned long long num_by_mk) // NG STDでリザーブ
{
    return num_by_mk * one_km;
}

constexpr int32_t operator""_meter(unsigned long long num_by_m) // OK
{
    return num_by_m;
}

constexpr int32_t operator""_m(unsigned long long num_by_m) // NG 短すぎる
{
    return num_by_m;
}

```

```

// @@@ example/programming_convention/func_ut.cpp 656

auto km = int32_t{3_kilo_meter}; // ユーザ定義リテラル演算子の利用
auto m = int32_t{3000_meter};   // ユーザ定義リテラル演算子の利用

ASSERT_EQ(m, km);

```

実引数/仮引数

- 仮引数(「実引数/仮引数」参照)の数は、4個程度を上限とする。
- 仮引数を関数の戻り値として利用しない場合(且つ仮引数が関数テンプレートのユニバーサルリファレンスでない場合)、
 - 基本型やその型のエイリアス、enumは値渡しにする。
 - それ以外のオブジェクトはconstリファレンス渡しにする(「const/constexprインスタンス」参照)。ただし、数バイトの小さいオブジェクトは値渡ししても良い。
 - 「引数がnullptrである場合の処理をその関数が行う」場合、constポインタ渡しにする。

```

// @@@ example/programming_convention/func_ut.cpp 674

void f(int32_t a, enum EnumArg b, NotSmall const& c, Small d, NotSmall const* e) noexcept
{
    // a : 基本型
    // b : enum
    // c : サイズが小さくないオブジェクトで、nullptrでないことが前提
    // d : サイズ小さいオブジェクト
    // e : サイズが小さくないオブジェクトを指すが、nullptrである場合も処理の対象

    if (e == nullptr) {
        ...
    }
    else {
        ...
    }
    ...
}

```

- [注意] 仮引数をconstリファレンス渡しやconstポインタ渡しにすることで、
 - 値渡しに比べて、ランタイムでの処理が速くなる。
 - リファレンスやポインタ経由で引数に使用されたオブジェクトが変更されるのを防ぐ(値渡しあれば引数に使用されたオブジェクトが変更されることはない)。
- 仮引数を関数の戻り値として利用する場合、

- 「関数が、仮引数が`nullptr`である場合の処理を行う」場合、ポインタ渡しにする。
- 「関数が、仮引数が`nullptr`でないことを前提している」場合、リファレンス渡しにする。
- ユニバーサルリファレンスを仮引数とする関数テンプレートでは、仮引数を関数の戻り値として利用しない場合でも、仮引数は非`const`にする(ユニバーサルリファレンスは`rvalue`にもバインドできるため`const`にすることはできない)。
- 繙承の都合等で、使用しないにもかかわらず定義しなければならない仮引数には名前を付けない。
- 関数`f()`の仮引数が2つ以上であり、`f()`に渡す引数をそれぞれに生成する関数があった場合、引数を生成する関数の戻り値を直接`f()`に渡さない。

```
// @@@ example/programming_convention/func_ut.cpp 698

class A {
public:
    int32_t f0() noexcept { return a_++; }

    int32_t f1() noexcept { return a_--; }
    ...
};

void f(int32_t a0, int32_t a1) noexcept
{
    ...
}

void g(A& a) noexcept
{
    f(a.f0(), a.f1()); // NG f0()、f1()が呼ばれる順番は未定義。

    auto a0 = a.f0();
    auto a1 = a.f1();
    f(a0, a1); // OK f0()はf1()よりも先に呼ばれる。
}
```

- copyコンストラクタ、copy代入演算子、moveコンストラクタ、move代入演算子の仮引数名は`rhs`(「略語リスト」参照)にする。
- 二項演算子の仮引数名は、左側を`lhs`、右側を`rhs`にする。

```
// @@@ example/programming_convention/func_ut.cpp 731

class B {
public:
    B(A const& rhs);
    B& operator=(A const& rhs);

private:
    int32_t b_{0};

    friend bool operator==(B const& lhs, B const& rhs) noexcept { return lhs.b_ == rhs.b_; }
};
```

- 仮引数の意味を明示するために、関数宣言の仮引数の名前は省略しない。
- 仮引数がない関数の`()`の中には何も書かない。Cからリンクされる場合に限り、関数の`()`の中には`void`と書く。

```
// @@@ example/programming_convention/func_ut.cpp 747

class C {
public:
    void SetValue(int32_t number_of_people); // OK
// void SetValue(int32_t); // NG 仮引数名を書く
    void SetValue(D const& d); // OK
// void SetValue(D const&); // NG 仮引数名を書く
// int32_t GetValue(void) const; // NG void不要
    int32_t GetValue() const; // OK
};

extern "C" int32_t XxxGetValue(void); // OK Cからリンクされる
```

- 実引数として使用される配列がポインタ型へ暗黙に変換されることを前提に、仮引数をポインタ型にしない。また、仮引数を一見、配列に見えるポインタ型にしない(「スライシング」で述べたように、特に基底クラスを配列にすることは危険である)。代わりに配列へのリファレンスを使用する。

```
// @@@ example/programming_convention/func_ut.cpp 770

class Base {
public:
    Base(char const* name) noexcept : name0_{name} {}
```

```

char const* Name0() const noexcept { return name0_; }

...
private:
    char const* name0_;
};

class Derived final : public Base {
public:
    Derived(char const* name0, char const* name1) noexcept : Base{name0}, name1_{name1} {}
    char const* Name1() const noexcept { return name1_; }

    ...
private:
    char const* name1_;
};

std::vector<std::string> f(Base const* array, uint32_t n) // NG 誤用しやすいシグネチャ
{
    auto ret = std::vector<std::string>{n};

    std::transform(array, array + n, ret.begin(), [](Base const& b) noexcept { return b.Name0(); });

    return ret;
}

std::vector<std::string> g(Base const array[10], uint32_t n) // NG 誤用しやすいシグネチャ
{
    // str_arrayは一見、配列に見えるが、実際はポインタであるため。
    // この関数のシグネチャはf(Base const* str_array, uint32_t n)と同じ。
    // 配列の長さに見える10はシナックス上の意味を持たない。
    auto ret = std::vector<std::string>{n};

    std::transform(array, array + n, ret.begin(), [](Base const& b) noexcept { return b.Name0(); });

    return ret;
}

```

```

// @@@ example/programming_convention/func_ut.cpp 821

TEST(ProgrammingConvention, use_convert_to_ptr)
{
    Base    b[]{"0", "0"};
    Derived d[]{"0", "1"}, {"2", "3"};

    ASSERT_EQ((std::vector<std::string>{"0", "0"}), f(b, array_length(b))); // OK これは良いが
    ASSERT_EQ((std::vector<std::string>{"0", "0"}), g(b, array_length(b))); // OK これは良いが

#ifndef _MSC_VER // 本来なら、下記のようになるべきだが、
    ASSERT_EQ((std::vector<std::string>{"0", "2"}), f(d, array_length(d))); // NG
    ASSERT_EQ((std::vector<std::string>{"0", "2"}), g(d, array_length(d))); // NG
#else // レイアウトずれにより、下記のようになる
    ASSERT_EQ((std::vector<std::string>{"0", "1"}), f(d, array_length(d))); // NG
    ASSERT_EQ((std::vector<std::string>{"0", "1"}), g(d, array_length(d))); // NG
#endif
}

```

```

// @@@ example/programming_convention/func_ut.cpp 842

// ポインタではなく、配列へのリファレンスを使用することで、
// 上記のようなバグを避けることができる

std::vector<std::string> f_ref_2(Base const (&array)[2]) // OK
{
    auto ret = std::vector<std::string>{array_length(array)};

    // arrayの型はポインタではなく、リファレンスなのでstd::endが使える
    std::transform(array, std::end(array), ret.begin(),
                  [](Base const& b) noexcept { return b.Name0(); });

    return ret;
}

template <uint32_t N>
std::vector<std::string> f_ref_n(Base const (&array)[N]) // OK
{
    auto ret = std::vector<std::string>{N};

    std::transform(array, std::end(array), ret.begin(), [](auto& b) noexcept { return b.Name0(); });

    return ret;
}

```

```

}

template <typename T, uint32_t N>
std::vector<std::string> g_ref(T const (&array)[N]) // OK
{
    auto ret = std::vector<std::string>{N};

    std::transform(array, std::end(array), ret.begin(), [](auto& b) noexcept { return b.Name0(); });

    return ret;
}

// NULLを渡す必要がある場合、配列へのリファレンスの代わりに、
// 配列へのポインタを使うことができる

template <typename T, uint32_t N>
std::vector<std::string> g_ptr(T const (*array)[N]) // OK
{
    if (array == nullptr) {
        return std::vector<std::string>{};
    }

    auto ret = std::vector<std::string>{N};

    std::transform(*array, std::end(*array), ret.begin(),
                  [](auto& b) noexcept { return b.Name0(); });

    return ret;
}

```

```

// @@@ example/programming_convention/func_ut.cpp 899

TEST(ProgrammingConvention, not_use_convert_to_ptr)
{
    Base    b[]{"0", "0"};
    Derived d[]{{"0", "1"}, {"2", "3"}};

    ASSERT_EQ((std::vector<std::string>{"0", "0"}), f_ref_2(b)); // OK
    ASSERT_EQ((std::vector<std::string>{"0", "0"}), f_ref_n(b)); // OK
    ASSERT_EQ((std::vector<std::string>{"0", "0"}), g_ref(b)); // OK

    // ASSERT_EQ((std::vector<std::string>{"0", "2"}), f_ref_2(d)); // OK 誤用なのでコンパイルエラー
    ASSERT_EQ((std::vector<std::string>{"0", "2"}), g_ref(d)); // OK

    // 配列へのポインタを使う場合
    ASSERT_EQ((std::vector<std::string>{"0", "0"}), g_ptr(&b)); // OK

    Derived(*d_null)[3]{nullptr};
    ASSERT_EQ((std::vector<std::string>{}), g_ptr(d_null)); // OK
}

```

- ・デフォルト引数は関数のプロトタイプ宣言もしくはクラス宣言内のメンバ関数宣言のみに記述する（「オーバーライド」参照）。
- ・メンバ関数のデフォルト引数は、そのクラス外部からのメンバ関数呼び出しを簡潔に記述するための記法であるため、非publicなメンバ関数にデフォルト引数を持たさない。
- ・デフォルト引数の初期化オブジェクトは定数、もしくは常に等価なオブジェクトにする。デフォルト引数の初期化オブジェクトは関数呼出し時に評価されるため、引数の初期化オブジェクトが等価でない場合、関数の処理が初期化オブジェクトの現在の状態に依存してしまう。

```

// @@@ example/programming_convention/func_ut.cpp 921

int32_t default_arg{0};
int32_t get_default_arg(int32_t a = default_arg) noexcept { return a; }

```

```

// @@@ example/programming_convention/func_ut.cpp 930

ASSERT_EQ(0, get_default_arg()); // default_arg == 0

default_arg = 2;

ASSERT_EQ(2, get_default_arg()); // default_arg == 2

```

- std::unique_ptr<T> const&を引数とする関数は、その引数が指すオブジェクトが保持しているT型オブジェクトを書き換えることができるため、そのような記述をしない。関数がそのT型オブジェクトを書き換える必要があるのであれば引数をT&とする。書き換える必要がないのであれば引数をT const&とする。

```

// @@@ example/programming_convention/func_ut.cpp 942

void f0(std::unique_ptr<std::string> const& str) // NG *strは書き換え可能

```

```

{
    *str = "it can be changed";

#if 0 // strはconstなので以下はできない
    str = std::make_unique<std::string>("haha");
#endif
}

void f1(std::string& str) // OK
{
    str = "it can be changed";
}

void f2(std::string const& str) // OK
{
#if 0 // strは変更できない
    str = "it can NOT be changed";
#endif
}

void g()
{
    auto s = std::make_unique<std::string>("hehe");

    f0(s); // sは変更されないが、sが保持しているstd::stringオブジェクトは変更できる
    f1(*s); // sは変更されないが、sが保持しているstd::stringオブジェクトは変更できる
    f2(*s); // sも、sが指しているstd::stringオブジェクトも変更されない
}

```

- 演習-仮引数の修飾

自動変数

- 一つの文で複数の変数の宣言をしない。
- 自動変数は、それを使う直前に定義することでスコープを最小化する。
- 自動変数は、定義と同時に初期化する。

```

// @@@ example/programming_convention/func_ut.cpp 1008

int32_t a, b; // NG 一度に2つの変数定義
int32_t index;

// Do something
...

index = get_index(); // NG 定義と使用箇所が離れている
int32_t index2{get_index()}; // OK
auto index3 = get_index(); // OK AAA
auto index4 = int32_t{get_index()}; // OK 型を明示したAAA

int32_t i;

for (i = 0; i < max; ++i) { // NG 定義と使用箇所が離れている
    // Do something
}

for (int32_t i{0}; i < max; ++i) { // OK
    // Do something
    ...
}

for (auto i = 0; i < max; ++i) { // OK AAAスタイル
    // Do something
    ...
}

auto& w0 = Widget::Inst(); // if文後にはw0を使用しないならばNG
if (w0.GetStatus() == Widget::Success) {
    w0.DoSomething();
}
else {
    w0.DoSomething(Widget::None);
}
// この後w0を使用しない
...

```

```

if (auto& w1 = Widget::Inst(); w1.GetStatus() == Widget::Success) { // OK C++17より使用可能
    w1.DoSomething();
}
else {
    w1.DoSomething(Widget::None);
}

...

auto const& w2 = Widget::InstConst(); // switch文後にw2を使用しないならばNG
switch (w2.GetStatus()) {
case Widget::Success:
    // Do something
    break;
...
default:
    // Do something
    break;
}
// この後w2を使用しない

...

switch (auto const& w3 = Widget::InstConst(); w3.GetStatus()) { // OK C++17より使用可能
case Widget::Success:
    // Do something
    break;
...
default:
    // Do something
    break;
}

```

戻り値型

- メモリアロケータ以外の関数の戻り値をvoid*にしない。
- 避けがたい理由なしに「戻り値の型を後置する関数宣言構文」を使用しない。

```

// @@@ example/programming_convention/func_ut.cpp 1111

auto f(int32_t a, int32_t b) noexcept -> decltype(a + b) // NG
{
    return a + b;
}

template <typename T>
auto f(T a, T b) noexcept -> decltype(a + b) // OK 後置構文以外に方法がない
{
    return a + b; // T = uint8_tとすると、a + bの型はint32_t
}

```

- 戻り値を比較的大きなオブジェクトにする場合、パフォーマンスに注意する（「関数の戻り値オブジェクト」参照）。
- 関数が複数の値を返す場合、std::pair、std::tuple、構造体オブジェクトを戻り値にして返す。パフォーマンスに著しい悪影響がない限り、その値をリファレンス引数で返さない。

```

// @@@ example/programming_convention/func_ut.cpp 1124

void g0(int32_t a, int32_t b, int32_t& quotient, int32_t& remainder) // NG
{
    quotient = a / b;
    remainder = a % b;
}

int32_t g1(int32_t a, int32_t b, int32_t& remainder) // NG
{
    remainder = a % b;
    return a / b;
}

std::pair<int32_t, int32_t> g_pair(int32_t a, int32_t b) // OK
{
    return {a / b, a % b};
}

struct Result {
    int32_t quotient;
}

```

```

        int32_t remainder;
    };

    Result g_struct(int32_t a, int32_t b) // OK
    {
        return {a / b, a % b};
    }

// @@@ example/programming_convention/func_ut.cpp 1157

{
    int32_t quotient;
    int32_t remainder;
    g0(7, 3, quotient, remainder); // NG quotient、remainderが戻り値かどうかわかりづらい
    ASSERT_EQ(2, quotient);
    ASSERT_EQ(1, remainder);
}

{
    int32_t remainder;
    int32_t quotient{g1(7, 3, remainder)}; // NG remainderが戻り値かどうかわかりづらい
    ASSERT_EQ(2, quotient);
    ASSERT_EQ(1, remainder);
}

{
    auto ret = g_pair(7, 3); // OK
    ASSERT_EQ(2, ret.first);
    ASSERT_EQ(1, ret.second);
}

{
    auto [quotient, remainder] = g_struct(7, 3); // OK C++17 構造化束縛
    ASSERT_EQ(2, quotient);
    ASSERT_EQ(1, remainder);
}

{
    auto ret = g_struct(7, 3); // OK
    ASSERT_EQ(2, ret.quotient);
    ASSERT_EQ(1, ret.remainder);
}

```

- 処理の成否をbool等で通知し、成功時の戻り値をリファレンス引数で戻す関数や、処理の成功時の値と、失敗時の外れ値を戻り値で返す関数を作らない。代わりにC++17で導入されたstd::optionalを使用する。

```

// @@@ example/programming_convention/func_ut.cpp 1215

bool h0(int32_t a, int32_t b, int32_t& remainder) // NG
{
    if (b == 0) {
        return false;
    }

    remainder = a % b;

    return true;
}

int32_t h1(uint32_t a, uint32_t b) // NG 余りが-1になる場合(外れ値)、エラー通知
{
    if (b == 0) {
        return -1;
    }

    return a % b;
}

std::pair<bool, int32_t> h_pair(int32_t a, int32_t b) // NG
{
    if (b == 0) {
        return {false, 0};
    }

    return {true, a % b};
}

struct Result2 {
    bool is_success;
    int32_t remainder;
};

Result2 h_struct(int32_t a, int32_t b) // NG
{

```

```

    if (b == 0) {
        return {false, 0};
    }

    return {true, a % b};
}

std::optional<int32_t> h_optional(int32_t a, int32_t b) // OK
{
    if (b == 0) {
        return std::nullopt;
    }

    return a % b;
}

```

```

// @@@ example/programming_convention/func_ut.cpp 1272

{
    int32_t remainder;

    auto result = h0(7, 0, remainder);
    ASSERT_FALSE(result); // エラー時にremainderが有効か否かわからない
}

{
    auto remainder = h1(7, 0);
    ASSERT_EQ(-1, remainder); // エラー通知がわかりづらい
}

{
    auto [result, remainder] = h_pair(7, 0);
    ASSERT_FALSE(result); // エラー時にremainderが有効か否かわからない
}

{
    auto [result, remainder] = h_struct(7, 0);
    ASSERT_FALSE(result); // エラー時にremainderが有効か否かわからない
}

{
    auto result = h_optional(7, 0);
    ASSERT_FALSE(result);

    result = h_optional(7, 4);
    ASSERT_TRUE(result);
    ASSERT_EQ(3, *result); // 成功時の値取り出し
}

```

constexpr関数

- 多くの使用箇所で戻り値がコンパイル時に確定する関数テンプレートもしくはinline関数は、constexprと宣言する(「[constexprインスタンスと関数](#)」参照)。
- コンパイル時に値が確定するラムダ式を戻り値に持つ関数はconstexprと宣言する。
- [演習-constexpr関数](#)

リエントラント性

- 関数、メンバ関数はなるべくリエントラントに実装する。
- 複数のスレッドから呼び出される関数は必ずリエントラントにする。

```

// @@@ example/programming_convention/func_ut.cpp 1328

int32_t var{0};

int32_t f() noexcept // リエントラントでない関数f()
{
    return ++var;
}

int32_t f(int32_t& i) noexcept // リエントラントな関数f()
{
    return ++i;
}

```

エクセプション処理

- 関数はそれが不可避でない限り、[no-fail保証](#)をする。

- `throw`をせざるを得ない場合、最低でも基本保証をする。
- STLコンテナ(`std::string`, `std::vector`等)が発生させるエクセプションはtry-catchせず（`catch`してデバッグ情報を保存するような場合を除く）、プログラムをクラッシュさせる。`try-catch`してもできることはない。
- 特別な理由がない限り、コンストラクタ呼び出しは`noexcept`と宣言する。ネットワーク接続等、簡単にエラーすることをコンストラクタ内で行わない。
- オープン・クローズドの原則(OCP)、リスコフの置換原則(LSP)に違反する場合が多いため、「`throw`キーワードによるエクセプション仕様」を使用しない(C++17で廃止)。
- エクセプションを`throw`しないことが確定している関数は、`noexcept`と宣言する。`move`代入演算子を`noexcept`と宣言することは特に重要である。

```
// @@@ example/programming_convention/func_ut.cpp 1359

int32_t f() noexcept; // OK fはno-fail保証

class Derived : public Base {
    ...
    // オブジェクトの状態を変えず(const)、エクセプションを発生させず(noexcept)
    // f()の最後(final)のoverride
    virtual int32_t f() const noexcept final
    {
        ...
    }
};
```

- `try-catch`が不可避である場合、以下の理由により`const`リファレンスで受け取る。
 - 実態で受け取るとオブジェクトのスライシングが起こる場合がある。
 - 受け取ったエクセプションオブジェクトを書き換えるべきではない。
- エクセプションによるリソースリークを避けるためRAII(`scoped_guard`)でリソースを管理する。
- 一連の`catch`節では、`catch`するエクセプションの型の最もマッチ率の高い`catch`節で処理されるのではなく、マッチした最上位の`catch`節で処理されるため、`catch`するエクセプションの型に継承関係があるのであれば、継承順位が低い順番に`catch`する。また、`catch(...)`は一番最後に書く(関数tryブロックの場合も同様にする)。

```
// @@@ example/programming_convention/func_ut.cpp 1379

struct ExceptionA : std::exception {};
struct ExceptionB : ExceptionA {};
struct ExceptionX : std::exception {};

void order_of_catch() noexcept
{
    try {
        ...
    }
    catch (ExceptionB const& e) { // ExceptionAの前に書く。
        ...
    }
    catch (ExceptionA const& e) { // std::exceptionの前に書く。
        ...
    }
    catch (ExceptionX const& e) { // std::exceptionの前に書く。
        ...
    }
    catch (std::exception const& e) { // catch(...)の前に書く。
        ...
    }
    catch (...) { // 必ず一番最後に書く。
        ...
    }
}

void order_of_catch_with_try() noexcept
try { // 関数tryブロック
    ...
}
catch (ExceptionB const& e) { // ExceptionAの前に書く。
    ...
}
catch (ExceptionA const& e) { // std::exceptionの前に書く。
    ...
}
catch (ExceptionX const& e) { // std::exceptionの前に書く。
    ...
}
catch (std::exception const& e) { // catch(...)の前に書く。
    ...
}
```

```

    }
    catch (...) { // 必ず一番最後に書く。
        ...
    }
}

```

- エクセプションをthrowする場合、独自定義したオブジェクトを使用しない。代わりにstd::exceptionか、これから派生したクラスを使用する。また、throwされたオブジェクトのwhat()から、throwされたファイル位置が特定できるようにする（「[ファイル位置を静的に保持したエクセプションクラスの開発](#)」参照）。

- noexceptと宣言された関数へのポインタへ、noexceptでない関数のポインタを代入しない（C++17ではill-formedになる）。

```

// @@@ example/programming_convention/func_ut.cpp 1442

int32_t f0() // noexceptではないため、エクセプションを発生させることがある。
{
    ...
}

int32_t f1() noexcept
{
    ...
}

#ifndef __cplusplus < 201703L // 以下のコードはC++14以前ではコンパイルできるが、
                           // C++17以降ではコンパイルエラー
int32_t (*f_ptr0)() noexcept = &f0; // NG f_ptr0()はnoexceptだが、
                                   //     f0はエクセプションを発生させる可能性がある。
#endif
int32_t (*f_ptr1)() noexcept = &f1; // OK
int32_t (*f_ptr2)()          = &f0; // OK
int32_t (*f_ptr3)()          = &f1; // OK f1はエクセプションを発生させない。

class A {
public:
    int32_t f0() // noexceptではないため、エクセプションを発生させることがある。
    {
        ...
    }

    int32_t f1() noexcept
    {
        ...
    }
};

#ifndef __cplusplus < 201703L // 以下のコードはC++14以前ではコンパイルできるが、
                           // C++17以降ではコンパイルエラー
int32_t (A::*mf_ptr0)() noexcept = &A::f0; // NG mf_ptr0()はnoexceptだが、
                                         //     f0はエクセプションを発生させる可能性がある。
#endif
int32_t (A::*mf_ptr1)() noexcept = &A::f1; // OK
int32_t (A::*mf_ptr2)()          = &A::f0; // OK
int32_t (A::*mf_ptr3)()          = &A::f1; // OK f1はエクセプションを発生させない。

```

• [演習-エクセプションの型](#)

ビギーループ

- 待ち合わせにビギーリープを使わない。イベントドリブンにする。

```

// @@@ example/programming_convention/func_ut.cpp 1504

// NG イベントドリブンにするべき
void wait_busily() noexcept
{
    while (1) {
        sleep(1);
        if (xxx_flag) {
            ...
            break;
        }
    }
}

// OK selectでイベント発生を待つ。
void wait_event(fd_set const& rfds, uint32_t wait_sec) noexcept

```

```

{
    while (1) {
        auto rfd2 = rfds;
        auto tv    = timeval{wait_sec, 0};

        auto retval = select(1, &rfds2, 0, 0, &tv);

        ...
    }
    ...
}

```

- [注意] C++11からイベント通知のためにstd::condition_variable (「[並行処理](#)」参照)が導入された。

構文

複合文

- if, else, for, while, do後には複合文を使う。

```

// @@@ example/programming_convention/syntax_ut.cpp 22

if (a == 0) {
    b = 0; // OK
}

if (a == 0)
    b = 0; // NG 複合文でない

if (a == 0) {
    b = 0; // OK
}
else
    b = 1; // NG 複合文でない

for (auto i = 0; i < a; ++i) { // OK
    c[i] = i;
}

for (auto i = 0; i < a; ++i) // NG
    c[i] = i;

```

- 空の複合文には、何もすることがないという意図を表現するため、“;”だけの文を置き、空の複合文である事を明示する。

```

// @@@ example/programming_convention/syntax_ut.cpp 53

while (volatile_flag) {
} // NG ;が無い

while (volatile_flag) {
    ; // OK
}

while (volatile_flag)
    ; // NG whileの文が複合文でない

```

switch文

- caseラベル、defaultラベルに関連付けられた一連の文はできだけフォールスルーさせない。実装がシンプルになる等の理由からフォールスルーサせる場合、それが意図であることを明示するため以下のような記述する。

// fallthrough // C++14以前

[[fallthrough]]; // C++17以降

- defaultラベルは省略せず、switch文の末尾に書く。

- defaultラベルに関連付けられた処理がない場合は、breakのみを記述する。

- 論理的にdefaultラベルに到達しないのであれば、defaultラベルに続いてassert(false)を実行することで、そこを通過してはならないことを明示する(「[assertion](#)」参照)。

```

// @@@ example/programming_convention/syntax_ut.cpp 79

```

```

switch (a) {
case 0:
    b = 0;
    break; // OK
case 1:
    e = 2; // NG break無しで抜けているのにコメントが無い
case 2:
    c = 1;
    // fallthrough C++14以前であればOK
case 3:
    e += 2;
    [[fallthrough]]; // OK C++17
case 4:
    d = 1;
    break; // OK
default:
    assert(false); // OK 論理的にここには来ないのならば、defaultを省略せずにassert
}

```

if文

- if-else-ifと連続する場合は、else文で終了させる。最後のelseのブロックでやるべき処理がないのであれば、そのブロックに;のみを記述する。

```

// @@@ example/programming_convention/syntax_ut.cpp 111

if (a == 1) {
    ...
}
else if (a == 2) {
    ...
} // NG elseで終了していない

if (a == 1) {
    ...
}
else if (a == 2) {
    ...
}
else { // OK else文でやることがない場合は、;のみ記述
    ;
}

```

- if-else-ifの最後のelseのブロックに論理的に到達しないのであれば、そのブロックでassert(false)を実行する(「[assertion](#)」参照)。

```

// @@@ example/programming_convention/syntax_ut.cpp 134

if (a == 1) {
    ...
}
else if (a == 2) {
    ...
}
else { // OK
    assert(false); // ここに来るのはバグの場合。
}

```

- 条件が2つ以上且つ、switchで表現できる条件文には、ifではなくswitchを使用する。
- ifの条件式が、コンパイル時に定まるのであれば、[constexpr](#) if文を使用する。

範囲for文

- 配列やコンテナの全要素にアクセスするような繰り返し処理には、[off-by-oneエラー](#)が避けられ、従来よりもシンプルに記述できる範囲for文を使用する。

```

// @@@ example/programming_convention/syntax_ut.cpp 155

auto vect = std::vector<uint32_t>{0, 1, 2, 3, 4};

// NG oldスタイル
for (auto i = 0U; i < vect.size(); ++i) {
    std::cout << vect[i] << " ";
}

// NG C++03スタイル

```

```

for (std::vector<uint32_t>::iterator it = vect.begin(); it != vect.end(); ++it) {
    *it = 3;
}

for (std::vector<uint32_t>::const_iterator it = vect.cbegin(); it != vect.cend(); ++it) {
    std::cout << *it << " ";
}
...

// OK C++11スタイル
for (auto const& a : vect) {
    std::cout << a << " ";
}
...

```

- 独自のコンテナクラスを定義する場合、STLコンテナと同様の要件を満たす`begin()`、`end()`や、`cbegin()`、`cend()`も定義し、そのコンテナに範囲for文を適用できるようにする（「[デバッグ用イテレータ](#)」参照）。

- [演習-コンテナの範囲for文](#)

制御文のネスト

- `break`との関係がわかりづらい、ブロックが巨大になる等の問題があるため、`if`, `for`, `while`, `do-while`, `switch`文に付随するブロックの中に`switch`文を書かない。

return文

- `return`の後に括弧をつけない。

```

// @@@ example/programming_convention/syntax_ut.cpp 194

...
if (xxx) {
    // decltype(retval)は、int32_t
    // decltype(retval)は、(retval)がlvalueであるためint32_t&
    // この違いは通常問題にはならないが、関数の戻り値を型推測させると問題になる。
    return (retval); // NG ()は不要
}
else {
    return retval2; // OK
}

```

- 下記のような`return`しない関数には、`[[noreturn]]`をつけて宣言、定義する。

```

// @@@ example/programming_convention/syntax_ut.cpp 215
// @fn terminate(char const* message)
// @brief メッセージを出してプログラムを終了させる。
// @param const char* message 上記メッセージ
[[noreturn]] void terminate(char const* message)
{
    auto const str = std::string("unrecoverable error") + message;

    ...
    throw std::runtime_error{str};
}

```

goto文

- 二重以上のループを抜ける目的以外で`goto`を使用しない。
- 二重以上のループを抜ける目的で`goto`を使用する場合、`goto`のジャンプ先ラベルはそのループの直後に置く。

ラムダ式

- [ラムダ式を複雑にしない。](#)
 - できるだけワンライナーにする。
 - 必ず10行以下にする。

```

// @@@ example/programming_convention/syntax_ut.cpp 235
// ラムダ式はワンライナーが基本
auto itr = std::find_if(strs.begin(), strs.end(), [](auto const& n) noexcept {

```

```
    return (n.at(0) == '\n') && (n.size() > 5);
};
```

- デフォルトのキャプチャ方式は、ローカル変数を無限にキャプチャしてしまうため使用しない。
 - C++11では、キャプチャする変数ごとに代入キャプチャか参照キャプチャを使用する。
 - C++14以降では、初期化キャプチャを使用する。

```
// @@@ example/programming_convention/syntax_ut.cpp 260

// NG デフォルトのキャプチャ方式
class A {
public:
    ...
    std::vector<std::string> GetNameLessThan(uint32_t length) const
    {
        auto ret = std::vector<std::string>{};

#If 1 // 手が滑って、strをstrs_としてしまったバグ([=]によってthisが導入されている)。
        std::copy_if(strs_.cbegin(), strs_.cend(), std::back_inserter(ret),
                    [=](auto const& str) noexcept { return (strs_.size() < length); });

#else // 本来は下記のように書きたかったが、
// キャプチャ範囲が大きすぎるため上記バグを生み出てしまった。
        std::copy_if(strs_.cbegin(), strs_.cend(), std::back_inserter(ret),
                    [=](auto const& str) noexcept { return (str.size() < length); });
#endif
        return ret;
    }

private:
    std::vector<std::string> strs_;
};
```

```
// @@@ example/programming_convention/syntax_ut.cpp 295

// OK 限定したキャプチャにより、ラムダ式から可視である変数が限定された
class A {
public:
    ...
    std::vector<std::string> GetNameLessThan(uint32_t length) const
    {
        auto ret = std::vector<std::string>{};

#if CPP_VER == 11 // C++11
// [length]を代入キャプチャと呼ぶ。
        std::copy_if(strs_.cbegin(), strs_.cend(), std::back_inserter(ret),
                    [length](std::string const& str) noexcept { return (str.size() < length); });

#elif CPP_VER == 14 // C++14以降
// [length = length]を初期化キャプチャと呼ぶ。
// 左のlengthのスコープはラムダ式内。右のlengthのスコープはGetNameLessThan内。
        std::copy_if(strs_.cbegin(), strs_.cend(), std::back_inserter(ret),
                    [length = length](auto const& str) noexcept { return (str.size() < length); });
#else
        static_assert(false, "CPP_VER should be 11 or 14");
#endif
        return ret;
    }
    ...
};
```

- 外部のオブジェクトを参照キャプチャしたクロージャを、その外部オブジェクトのライフタイムを超えて使用しない。
 - 関数で作られたクロージャがその関数のローカル変数のハンドルを使用するのであれば、そのクロージャをその関数外で使用しない。
 - オブジェクトで作られたクロージャがそのメンバ変数のハンドルを使用するのであれば、そのオブジェクトの終了後にそのクロージャを使用しない。

```
// @@@ example/programming_convention/syntax_ut.cpp 331

class B {
public:
    ...
    std::function<bool(int)> GenLambda(int max)
    {
```

```

    // NG
    // この関数が返すはクロージャがリファレンスしているmaxはこの関数が終了すると無効になる。
    return [&max](int n) noexcept { return n < max; };
}

std::function<bool(int)> GenLambda()
{
    // NG
    // この関数が返すクロージャがリファレンスしているmin_=min_はBオブジェクトが終了すると無効になる。
    return [&min_ = min_](int n) noexcept { return n > min_; };
}

private:
    int min_;
};

```

- C++17以降では、コンパイル時に戻り値が確定するラムダ式の呼び出し式はリテラルにできるため、そのようなラムダ式を戻り値に持つ関数はconstexprと宣言する。

```

// @@@ example/programming_convention/syntax_ut.cpp 369

auto square1 = [](int32_t n) { return n * n; };

static_assert(square1(2) == 4); // C++17以降、square1(2)はリテラル

auto i = 2;
// static_assert(square1(i) == 2); // iはconstexprではないので、コンパイルエラー

constexpr auto j      = 2;
constexpr auto square2 = [n = j](){ { return n * n; }; // constexprの宣言が必要

static_assert(square2() == 4); // C++17以降、square2()はリテラル

auto square3 = [n = j](){ { return n * n; };

// static_assert(square3() == 4); // square3()はリテラルではないので、コンパイルエラー

```

```

// @@@ example/programming_convention/syntax_ut.cpp 390

constexpr int32_t square4(int32_t n) // OK nがconstexprであれば、ラムダはconstexpr
{
    return [n] { return n * n; }();
}

static_assert(square4(2) == 4); // C++17以降、square4(2)はリテラル

constexpr auto square5(int32_t n) // OK nがconstexprであれば、ラムダはconstexpr
{
    // nがconstexprならば、ラムダ式はリテラル
    auto f = [n] { return n * n; };

    // fの戻り値がリテラルならば、gもリテラル
    auto g = [f] { return f(); };
    return g;
}

static_assert(square5(2)() == 4); // C++17以降、square5(2)はリテラル

```

- 演習-ラムダ式
- 演習-ラムダ式のキャプチャ

マクロの中の文

- マクロの中に文がある場合、do-while(0)イデオムを使用する(「関数型マクロ」参照)。

```

// @@@ example/programming_convention/syntax_ut.cpp 418

// do-while(0)イデオムによる関数型マクロ
#define INIT_ARRAY(array_, x_) \
    do { \
        for (auto& a_ : (array_)) { \
            a_ = (x_); \
        } \
    } while (0)

void f(uint32_t (&a)[10])
{
    // INIT_ARRAYがdo-whileではなく、単なるブロックで囲むと、";"が余計になる。

```

```
    INIT_ARRAY(a, 3);
}
```

演算子

優先順位

- 優先順位が分かりづらい式では、順序を明示するために丸括弧を使う。

```
// @@@ example/programming_convention/operator_ut.cpp 20

// 論理演算子例
if (a < b && c < d || e < f) // NG 優先順位がわからない
{
    ...
}

if (((a < b) && (c < d)) || (e < f)) // OK
{
    ...
}

// シフト演算子例
auto a0 = b << 16 + 1; // NG
auto a1 = b << (16 + 1); // OK
auto a2 = (b << 16) + 1; // OK

// ビット演算ではないが
std::cout << a0 + 1; // NG
std::cout << (a1 + 1); // OK

// 三項演算子例
auto e0 = a ? b : c = d; // NG
auto e1 = ((a ? b : c) = d); // OK
auto e2 = (a ? b : (c = d)); // OK 上記NG式と同じ意味
```

- [注意] 複合代入式と、それと等価に見える式での演算順序の違いに気を付ける。

```
// @@@ example/programming_convention/operator_ut.cpp 62
{
    auto a = 4;

    a = a * 3 / 2;
    ASSERT_EQ(6, a);
}
{
    auto a = 4;

    a *= 3 / 2; // この式は、a = a * 3 / 2と等価ではない
    ASSERT_EQ(4, a);
}
```

代入演算

- 単純代入のみからなる文を除き、1つの文で複数の代入を行わない。

```
// @@@ example/programming_convention/operator_ut.cpp 89

a = b = 0; // OK
b      = (a += 1) + 2; // NG
b      = (a++) + (c++); // NG

b = b++; // NG unary operators assign itself.

++a; // OK
auto i = a; // OK

a = 0; // OK
```

- 一部の例外を除き、ifの条件文の中で代入しない。

```
// @@@ example/programming_convention/operator_ut.cpp 105

if (c = b) { // NG ifの条件文の中で代入
    return 0;
}
```

```

if ((fd1 = socket(AF_INET, SOCK_STREAM, 0)) < 0) { // OK このような場合は代入していることが明確
    return false;
}

if (auto fd2 = socket(AF_INET, SOCK_STREAM, 0); fd2 < 0) { // OK C++17
    return false;
}
else {
    // fd2を使った処理
    ...
}

```

ビット演算

- オーバーフロー、アンダーフローしたときの符号の扱い方が未定義であるため、`signed`変数へのビット演算を使用しない。（「2の階乗での除算は、ビット演算に置き換えることで実行速度が速くなる」というのは都市伝説である）。
- [注意] ビット演算には`std::bitset`や`BitmaskType`を使用することもできる。

論理演算

- `&&`や`||`の論理演算子の右オペランドで副作用のある処理をしない。

```

// @@@ example/programming_convention/operator_ut.cpp 136

if (a == 0 && ++b > 3) { // NG ++bが上記の副作用
    ...
}

// ↑のような記述は、↓とは意味が違う
++b;
if (a == 0 && b > 3) { // OK
    ...
}

if (a == 0) { // OK 上記NGのif文と同じ意味
    ++b;
    if (b > 3) {
        ...
    }
}

```

三項演算子

- 単純なif文よりも、三項演算子を優先して使用する(`const`変数の条件付き初期化は三項演算子でのみ可能である)。

```

// @@@ example/programming_convention/operator_ut.cpp 168

int const a0{xxx ? 3 : 4}; // OK constで定義、初期化

int a1; // NG a1をconstにできない
        // 定義と初期が分離してしまう
if (xxx) {
    a1 = 3;
}
else {
    a1 = 4;
}

```

- 演習-三項演算子

メモリアロケーション

new

- オブジェクトのダイナミックな生成には、特別な理由がない限り`new`を使用せず、`std::make_unique`や`std::make_shared`を使用する。また、特別な理由で`new`した場合、そのポインタは`std::unique_ptr`や`std::shared_ptr`で管理する。
- `new(throw)`、プレースメント`new`は使用しない。
- 配列型オブジェクトのダイナミックな生成を避け、代わりに`std::array`をダイナミックに生成するか、`std::vector`を使用する。
- `new`の戻り値が`nullptr`であることはない、もしくは`new`が`nullptr`を返してきた場合、リカバリーすることはできないため、`new`演算子の戻り値を`nullptr`と比較しない。

- operator newを独自に実装した場合でも、newはnullptrを返してはならない。メモリが不足した場合、assert(false)させるか std::bad_allocをthrowする。
- スタック上で生成しても差し支えないオブジェクトをダイナミックに生成しない。
- newを禁止したいクラスには、privateなoperator new()を宣言する(定義は不要)か、= deleteする。

delete

- 不完全型のオブジェクトへのポインタをdeleteしない。特に「Pimpl」を使用する場合には注意が必要である。

```
// @@@ example/programming_convention/operator_ut.cpp 202

void deleteA(A* a_ptr) noexcept
{
    // Aが不完全型だった場合、deleteAは、A::~A()にアクセスできないため、A::~A()は呼び出されない
    // これはリソースリークにつながる
    delete a_ptr;
}

// やむを得ず、deleteAのような関数を作る場合、下記のようにstatic_assertをdelete行の直前に書く
// こうすることによりAが不完全型であった場合、コンパイルエラーとなる
void deleteA2(A* a_ptr) noexcept
{
    static_assert(sizeof(*a_ptr) != 0, "incomplete type");
    delete a_ptr;
}

// やむを得ず、deleteAのような関数を作る場合、std::unique_ptr<>を使用することもできる
// こうすることによりAが不完全型であった場合、コンパイルエラーとなる
void deleteA3(A* a_ptr) noexcept { std::unique_ptr<A> a(a_ptr); }
```

- 不完全型と同じような不具合が起こるためvoid*をdeleteしない。

```
// @@@ example/programming_convention/operator_ut.cpp 224

void delete_ptr(void* v_ptr) noexcept
{
    // NG
    // 任意の型のポインタは、キャストすること無しでこの関数に渡すことができる
    // そのポインタがクラス型であった場合でも、void*として扱われるため、
    // そのクラスのデストラクタは呼び出されない
    delete v_ptr;
}

void deleteA4(A* ptr) noexcept
{
    delete_ptr(ptr); // NG ptrはvoid*へ暗黙のキャストが行われる
                     //      delete_ptrでは、A::~A()は呼び出されない
}
```

- deleteはオペランドがnullptrであった場合、何もしないため、delete対象ポインタをnullptrと比較しない。

```
// @@@ example/programming_convention/operator_ut.cpp 244

if (ptr != nullptr) { // NG nullptrとの比較は不要
    delete ptr;
}

...

delete ptr; // OK ptrがnullptrでも問題ない
```

- 演習-delete

メモリ制約が強いシステムでの::operator new

- [注意]このルールは以下のようなソフトウェアを対象とする。

- 使用できるメモリが少なく、且つほどんど再起動されない。
- メモリリークの可能性を否定できない3rdパーティライブラリを使っている。
- MISRA/AUTOSAR C++等のヒープの使用制限が強い規約を守る必要がある(ヒープを使った場合の最長処理時間の決定が難しいためリアルタイム性に問題がある)。

このようなソフトウェア開発においてはこのルールは重要であるが、逆にそのような制限のないソフトウェア開発においては不要である。

- デフォルトのグローバルnewを使用しない。
 - リアルタイム性に制約のあるシステムでは、「グローバルnew/deleteのオーバーロード」で述べたようなnewを実装する。
 - メモリ制限が強いシステムでは、ダイナミックなオブジェクト生成を避け、やむを得ない場合、「クラスnew/deleteのオーバーロード」で述べたようなクラス毎のnewを実装する。
- エクセプションの送出にダイナミックなメモリアロケーションを使用している場合(多くのコンパイラはmalloc/newを用いてエクセプション送出を行っている)、エクセプションの送出をしない(「エクセプション処理機構の変更」参照)。

sizeof

- sizeof(型名)とsizeof(インスタンス名)の両方が使える場合、sizeof(インスタンス名)を優先的に使用する。
- ポインタ型変数に関して、それが指しているインスタンスのサイズを獲得する場合は、sizeof(*ポインタ型変数名)を使用する(そのポインタがnullptrであってもデリファレンスされないので問題ない)。

```
// @@@ example/programming_convention/operator_ut.cpp 270

uint8_t a = 0;
uint8_t* b = &a;

auto s_0 = sizeof(uint8_t); // NG aのサイズをs_0に代入したい場合
auto s_1 = sizeof(a);      // OK aのサイズをs_1に代入したい場合
auto s_2 = sizeof(*b);     // OK *bのサイズをs_2に代入したい場合
```

- 上記例を除き、sizeof演算子のオペランドは一見副作用を持っているような式を含んではならない。

```
// @@@ example/programming_convention/operator_ut.cpp 282

a = 0;

auto size_3 = sizeof(++a); // NG おそらく意図通りには動かない
                           // この行でもa == 0(++aは効果がない)
```

- C++03のテンプレートの実装でよく使われたsizeofによるディスパッチを行わない。

```
// @@@ example/programming_convention/operator_ut.cpp 295

// 下記のようなsizeofディスパッチはC++03ではよく使われたが、
// C++11ではtype_traitsを使えば、もっとスマートに実装できる
struct True {
    uint8_t temp[2];
};

struct False {
    uint8_t temp[1];
};

constexpr True sizeof_dispatch(int32_t);
constexpr False sizeof_dispatch(...);

// @@@ example/programming_convention/operator_ut.cpp 312

static_assert(sizeof(sizeof_dispatch(int{})) == sizeof(True), "int32_t is int");
static_assert(sizeof(sizeof_dispatch(std::string{})) != sizeof(True), "int32_t is not string");

// 上記はC++11では下記のように実装すべき
static_assert(std::is_same_v<int, int32_t>, "int32_t is int");
static_assert(!std::is_same_v<std::string, int32_t>, "int32_t is not string");
```

- 一見、配列に見えるポインタをsizeofのオペランドにしない。(「実引数/仮引数」参照)。

```
// @@@ example/programming_convention/operator_ut.cpp 327

void f(int8_t arg_array0[5], int8_t arg_array1[], int8_t (&arg_array2)[5]) noexcept
{
    int8_t* ptr;
    int8_t array[5];

    // arg_array0, arg_array1の型は、int8_t*
    // 従って、sizeof(arg_array0)の値は、sizeof(int8_t) * 5ではなく、sizeof(int8_t*)である

    // 64bit環境でコンパイルポインタサイズは8バイト
    static_assert(8 == sizeof(arg_array0), "arg_array0 is a pointer but an array");
    static_assert(8 == sizeof(arg_array1), "arg_array1 is a pointer but an array");
    static_assert(5 == sizeof(arg_array2), "arg_array2 is an array");
    static_assert(8 == sizeof(ptr), "ptr must be 8 bytes on 64bit environment");
    static_assert(5 == sizeof(array), "int8_t[5] is 5 bytes");
}
```

- 演習 sizeof

ポインタ間の演算

- 同一オブジェクト(配列等)の要素を指さないポインタ間の除算や比較をしない。

```
// @@@ example/programming_convention/operator_ut.cpp 363

int8_t a0[5];
int8_t a1[5];
int8_t* end0{&a0[5]};
int8_t* end1{&a1[5]};

for (int8_t* curr{a0}; curr < end0; // OK currもend0もa0のどこかを指している
     ++curr) {
    *curr = 0;
}

for (int8_t* curr{a0}; curr < end1; // NG currとend1は別々のオブジェクトを指している
     ++curr) {
    *curr = 0;
}
```

RTTI

- [注意] RAII(scoped_guard)との混乱に気を付ける。
- Run-time Type Informationを使用したランタイム時の型による場合分けは、それ以外に解決方法がない場合や、実装が大幅にシンプルになる場合を除き行わない(「等価性のセマンティクス」参照)。
 - 単体テストやロギングでの typeid の使用は問題ない。
 - 派生クラスの型によって異なる動作にしたい場合には、仮想関数を使うか、Visitorパターン等により実現できる。

```
// @@@ example/programming_convention/operator_ut.cpp 383

class Base {
public:
    virtual ~Base() = default;
    ...
};

class Derived_0 : public Base {
    ...
};

class Derived_1 : public Base {
    ...
};

...

// NGの例
void b_do_something(Base const& b) noexcept
{
    auto name = std::string_view{typeid(b).name()};

    // bの実際の型を使った場合分けによる最悪のコード
    // dynamic_castによる場合分けも、下記のコードより大きく改善するわけではない
    if (name == "4BBase") { // マングリングされたBase
        ...
    }
    else if (name == "9Derived_0") { // マングリングされたDerived_0
        ...
    }
    else if (name == "9Derived_1") { // マングリングされたDerived_1
        ...
    }
    else {
        assert(false);
    }
}
```

```
// @@@ example/programming_convention/operator_ut.cpp 471

// OKの例
// 上記の b_do_something にポリモーフィズムを適用しリファクタリング
class Base {
public:
    void DoSomething() noexcept { do_something(); }
```

```

...
private:
    virtual void do_something() noexcept
    {
        ...
    }
};

class Derived_0 : public Base {
private:
    virtual void do_something() noexcept override
    {
        ...
    }
    ...
};

class Derived_1 : public Base {
public:
    virtual void do_something() noexcept override
    {
        ...
    }
    ...
};

// virtual Base::do_something()により醜悪なswitchが消えた
void b_do_something(Base& b) noexcept { b.DoSomething(); }

```

- コンストラクタやデストラクタ内でRTTIの機能を使わない（「[コンストラクタ](#)」、「[デストラクタ](#)」参照）
- [演習-dynamic_castの削除](#)

キャスト、暗黙の型変換

- キャストが必要な式等は、設計レベルの問題を内包していることがほとんどであるため、設計を見直す。
- Cタイプキャストは使用しない。
- const_cast、dynamic_castはそれ以外に解決方法がない場合や、実装が大幅にシンプルになる場合を除き使用しない。
- reinterpret_castはハードウエアレジスタ等のアドレスを表す目的以外で使用しない。
- ダウンキャストを行う目的でstatic_castを使用しない。

```

// @@@ example/programming_convention/operator_ut.cpp 524
class Base {
    ...
};

class Derived : public Base {
    ...
};

void f() noexcept
{
    auto d = Derived{};
    Base* b_ptr = &d; // ここまででは良い

    auto d_ptr = static_cast<Derived*>(b_ptr); // ダウンキャスト、動作保証はない
}

```

- strlenや、memcpyのような例を除き、void*への暗黙の型変換を行わない（これをしてると、後にダウンキャストが必要になる）。

```

// @@@ example/programming_convention/operator_ut.cpp 548

class A {
public:
    A() : str_{std::make_unique<std::string>("sample")}
    {
        ...
    }

    ~A()
    {
        ...
    }

private:
    std::unique_ptr<std::string> str_; // ~unique_ptr()は~A()から呼ばれる

```

```

};

class B;

// @@@ example/programming_convention/operator_ut.cpp 576

void* v = new A; // 暗黙の型変換
// これ自体は問題ないが、vをdeleteすると
// A::~A()が呼び出されないためメモリリークする

B* b = static_cast<B*>(v); // ダウンキャストの一種で、極めて悪質なことが多い
// 実際、このコードの中にクラスBの定義はないが
// このようなことができてしまう

delete v; // このdeleteは、A::~A()を呼び出さない

```

- `strncpy`や、`memcpy`のような例を除き、配列からポインタへの暗黙の型変換をしない。配列を関数の仮引数にしたい場合は、配列へのリファレンスを使う。これにより、その関数内でも配列の長さが使用できる（「実引数/仮引数」、「sizeof」、「関数型マクロ」参照）。

演習_キヤスト

プリプロセッサ命令

- ソースコードの可読性劣化や単体テストの阻害要因となるため、.cpp内で`#if`/`#ifdef`等を使用しない。

```

// @@@ example/programming_convention/preprocessor_ut.cpp 14

bool f() noexcept
{
#ifdef DEBUG // NG
    std::cout << __func__ << ":" << __LINE__ << std::endl;
#endif

    ...

#if 0 // NG
    return true;
#else // NG
    return false;
#endif
}

// やむを得ず条件付きコンパイルが必要な場合、下記のように書き、関数ブロックの中に
// #ifdefは書かない。

#ifdef DEBUG
#define DEBUG_COUT() std::cout << __func__ << ":" << __LINE__ << std::endl
#else
#define DEBUG_COUT()
#endif

bool g() noexcept
{
    DEBUG_COUT();

    ...

    return false;
}

```

- ヘッダファイル内での`#if`/`#ifdef`/`#ifndef`に関しては、以下の用途以外では使わない。

- 二重インクルードガード（「二重読み込みの防衛」参照）
- Cとシェアするヘッダファイルでの下記例

```

// @@@ example/programming_convention/preprocessor.h 3

#ifndef __cplusplus
extern "C" {
#endif // __cplusplus

extern bool func_shared_c();

#ifndef __cplusplus
}
#endif // __cplusplus

```

- ##によるシンボルの生成を使用しない。

```
// @@@ example/programming_convention/preprocessor_ut.cpp 53

#define GEN_SYMBOL(x_, y_) x_##y_

int32_t h() noexcept
{
    int32_t GEN_SYMBOL(a, b); // int ab;と同じ

    ab = 3;

    return ab;
}
```

- 出荷仕向け等の理由から、やむを得ずプリプロセッサ命令を使わざるを得ない場合、#if等で囲まれて区間をなるべく短くする。これによりより多くのコードがコンパイルされるようにできる(『[using宣言/usingディレクティブ](#)』参照)。

```
// @@@ example/programming_convention/preprocessor_ut.cpp 72
//
// ヘッダファイルでの宣言(NGのパターン)
//

enum class ShippingRegions { Japan, US, EU };

struct ShippingData {
    // 何らかの宣言
};

void ShippingDoSomething(ShippingData const& region_data);

#ifndef SHIP_TO_JAPAN && !defined(SHIP_TO_US) && !defined(SHIP_TO_EU) // NG
constexpr ShippingRegions shipping_region = ShippingRegions::Japan;
ShippingData const region_data{
    // 何らかのデータ
};

#ifndef SHIP_TO_JAPAN && defined(SHIP_TO_US) && !defined(SHIP_TO_EU) // NG
constexpr ShippingRegions shipping_region = ShippingRegions::US;
ShippingData const region_data{
    // 何らかのデータ
};

#ifndef SHIP_TO_JAPAN && !defined(SHIP_TO_US) && defined(SHIP_TO_EU) // NG
constexpr ShippingRegions shipping_region = ShippingRegions::EU;
ShippingData const region_data{
    // 何らかのデータ
};

#else
static_assert(false, "SHIP_TO_JAPAN/US/EU must be defined");
#endif

//
// .cppファイルでの定義(NGのパターン)
//
void ShippingDoSomething(ShippingData const& region_data)
{
#ifndef SHIP_TO_JAPAN && !defined(SHIP_TO_US) && !defined(SHIP_TO_EU) // NG
    // 何らかの処理
#endif
#ifndef SHIP_TO_JAPAN && defined(SHIP_TO_US) && !defined(SHIP_TO_EU) // NG
    // 何らかの処理
#endif
#ifndef SHIP_TO_JAPAN && !defined(SHIP_TO_US) && defined(SHIP_TO_EU) // NG
    // 何らかの処理
#endif
else
    static_assert(false, "SHIP_TO_JAPAN/US/EU must be defined");
}
}

// @@@ example/programming_convention/preprocessor_ut.cpp 140
//
// ヘッダファイルでの宣言(OKのパターン)
//
enum class ShippingRegions { Japan, US, EU };
```

```

struct ShippingData {
    // 何らかの宣言
};

void ShippingDoSomething(ShippingData const& region_data);

extern ShippingData const& region_data;

#if defined(SHIP_TO_JAPAN) && !defined(SHIP_TO_US) && !defined(SHIP_TO_EU) // OK
constexpr ShippingRegions shipping_region = ShippingRegions::Japan;
#endif

#if !defined(SHIP_TO_JAPAN) && defined(SHIP_TO_US) && !defined(SHIP_TO_EU) // OK
constexpr ShippingRegions shipping_region = ShippingRegions::US;
#endif

#if !defined(SHIP_TO_JAPAN) && !defined(SHIP_TO_US) && defined(SHIP_TO_EU) // OK
constexpr ShippingRegions shipping_region = ShippingRegions::EU;
#endif

#else
static_assert(false, "SHIP_TO_JAPAN/US/EU must be defined");
#endif

// .cppファイルでの定義(OKのパターン、以下にはプリプロセッサ命令は出てこない)
//

void ShippingDoSomething(ShippingData const& region_data)
{
    if constexpr (shipping_region == ShippingRegions::Japan) {
        // 何らかの処理
    }
    else if constexpr (shipping_region == ShippingRegions::US) {
        // 何らかの処理
    }
    else if constexpr (shipping_region == ShippingRegions::EU) {
        // 何らかの処理
    }
    else {
        static_assert(shipping_region == ShippingRegions::Japan
            || shipping_region == ShippingRegions::US
            || shipping_region == ShippingRegions::EU);
    }
}

template <ShippingRegions sr>
ShippingData const& gen_shipping_data()
{
    if constexpr (sr == ShippingRegions::Japan) {
        static ShippingData const region_data{
            // 何らかのデータ
        };
        return region_data;
    }
    else if constexpr (sr == ShippingRegions::US) {
        static ShippingData const region_data{
            // 何らかのデータ
        };
        return region_data;
    }
    else if constexpr (sr == ShippingRegions::EU) {
        static ShippingData const region_data{
            // 何らかのデータ
        };
        return region_data;
    }
    else {
        static_assert(sr == ShippingRegions::Japan || sr == ShippingRegions::US
            || sr == ShippingRegions::EU);
    }
}

ShippingData const& region_data = gen_shipping_data<shipping_region>();

```

関数型マクロ

- 関数型マクロ以外に方法がない場合を除き、関数型マクロを定義しない。その代わりに関数テンプレートを定義する。こうすることで下記のような誤用を防ぐことができる。

```

// @@@ example/programming_convention/preprocessor_ut.cpp 242

#define ARRAY_LENGTH(array_) (sizeof(array_) / sizeof(array_[0])) // NG

template <typename T, size_t N> // OK
constexpr size_t array_length(T const (&)[N]) noexcept
{
    return N;
}

// arrayは配列へのリファレンスだが関数中では配列
size_t f0(bool use_macro, int32_t (&array)[5]) noexcept
{
    if (use_macro) {
        return ARRAY_LENGTH(array); // この場合は、関数型マクロでも正しく処理できるが好ましくない
    }
    else {
        return array_length(array); // OK
    }
}

// fake_arrayは配列に見えるが実際にはポインタ
size_t f1(bool use_macro, int32_t fake_array[5]) noexcept
{
    if (use_macro) {
        return ARRAY_LENGTH(fake_array); // NG 誤用でもコンパイルできてしまい不正値を返す
    }
    else {
        // return array_length(fake_array); // OK 誤用のためコンパイルエラー
        auto array = reinterpret_cast<int32_t(*)[5]>(fake_array); // 無理やりコンパイル
        return array_length(*array);
    }
}

```

- 関数型マクロの中に文がある場合、do-while(0)イデオムを使用する（「[マクロの中の文](#)」参照）。

マクロ定数

- マクロ定数以外に方法がない場合を除き、マクロ定数を定義しない。その代わりにconstexpr uint32_t等や、enumを定義する。

```

// @@@ example/programming_convention/preprocessor_ut.cpp 293

#define XXX_LENGTH 5 // NG

constexpr uint32_t YyyLength{5}; // OK

#define XXX_TYPE_A 0 // NG
#define XXX_TYPE_B 1 // NG
#define XXX_TYPE_C 2 // NG

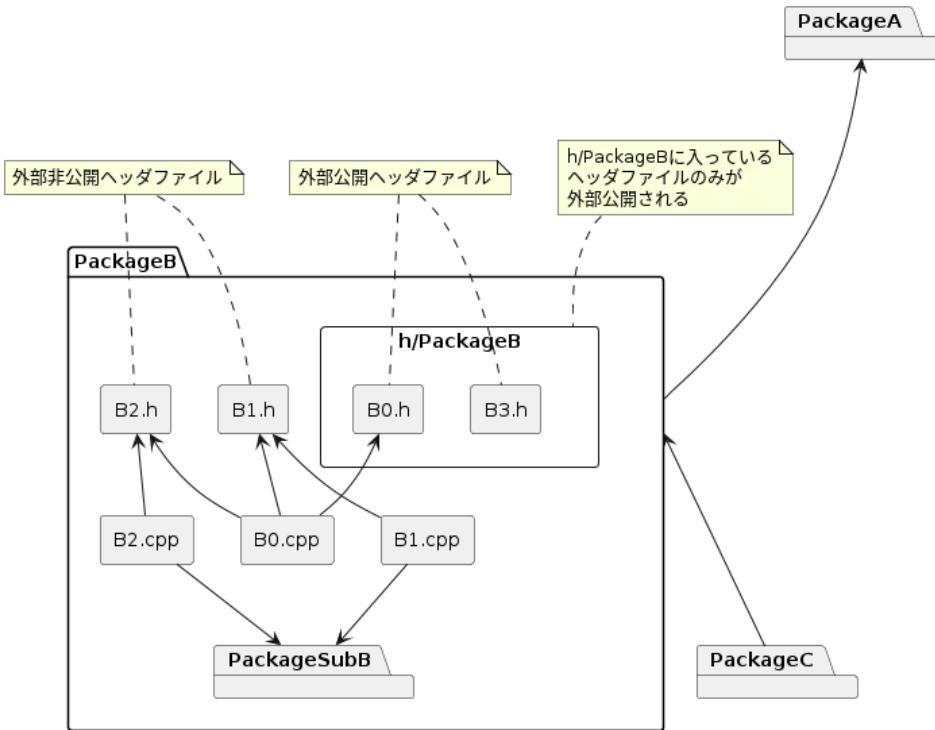
enum class XxxType { // OK
    A = 0,           // Aの値が必要だと前提
    B,
    C
};

```

パッケージとその構成ファイル

パッケージの実装と公開

- パッケージに関して、以下の構造を持つようにソースコードを構成する（「[アーキテクチャとファイル構造](#)」参照）。
 - ソフトウェアはパッケージに分割される。
 - パッケージは、複数のヘッダファイルと複数の.cppファイルから作られる。
 - パッケージを構成するファイルは、このパッケージ専用ディレクトリに保存される。
 - パッケージは他のパッケージとは排他的な名前空間を持ち、パッケージ内で宣言、定義され、外部パッケージに公開される識別子はその名前空間に含まれる。外部パッケージに公開されない識別子は、そのパッケージでのみ使用される名前空間か、無名名前空間に含まれるようにする。
 - パッケージは、サブパッケージを内包する場合がある。サブパッケージは、パッケージの要件を満たす。
- 上記前提のもと、パッケージが外部にサービスを提供する場合、そのパッケージで定義されるヘッダファイルは以下のどちらかのみを行うように構成する。
 - 外部へ提供するクラス等を公開する（外部公開ヘッダファイル）。
 - パッケージ内部のみで使用するクラス等の宣言、定義を行う（外部非公開ヘッダファイル）。



- h/PackageBのディレクトリPackageBは冗長に見えるが、h配下にそれを保持するディレクトリ名と同じ名前のディレクトリを持つことは、可読性の観点から重要である。このディレクトリため、他のパッケージから識別子をインポートするためのインクルードディレクトリは以下のように記述されることになる。このような記述を可能するために、PackageBがエクスポートするヘッダを使用するコードのコンパイルに「インクルールパスにPackageB/hを指定する」オプションを使用する。

```
// @@@ example/programming_convention/pkg.cpp 1

// PackageC内の*.cppファイルの内部とする
#include <memory> // システムヘッダ
#include <mutex> // システムヘッダ
#include <string> // システムヘッダ

// PackageBがエクスポートするするヘッダを使用するために、
// 以下のようなコンパイルオプションが必要になる
// -I<DIR>/PackageA/h
// <DIR>はコンパイラが実行されるディレクトリ
//
#include "PackageA/xxx.h" // PackageAのインクルード
#include "PackageB/b0.h" // PackageBのインクルード
#include "PackageB/b3.h" // PackageBのインクルード
#include "internal.h" // パッケージ外部非公開ヘッダのインクルード
```

- 外部公開ヘッダファイルは、ビルド時に他のパッケージから参照できるディレクトリに配置する。
 - 外部公開ヘッダファイルは、外部非公開ヘッダファイルをインクルードしない。
 - 外部非公開ヘッダファイルは、ビルド時に他のパッケージから参照できるディレクトリに配置しない。

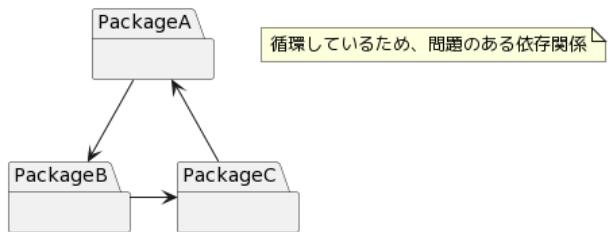
識別子の宣言、定義

- ODRを守る。つまり、一つの識別子は全ソースコード内にただ一つの定義を持つようにする。
- 一つの.cppファイル内で使用される識別子は、その.cppファイル内の無名名前空間にその定義や宣言を持つ。
- ヘッダファイルで宣言された識別子が.cppファイル内に定義をもつ場合、その.cppファイルにそのヘッダファイルをインクルードせざる。特に、非メンバ関数や変数の宣言と定義を矛盾させないために、このルールは特に重要である（非メンバ関数 参照）。

依存関係

- 不要/不適切な依存関係（インターフェース分離の原則(ISP) や 依存関係逆転の原則(DIP)への違反）を作らない。
 - 依存関係を最小にとどめるために、前方宣言を適切に使用する。
 - STLクラス（やクラスのエイリアス）の前方宣言をしない（例えば、「std::stringをクラス宣言することでstringヘッダファイルへの依存関係を作らない」といった方法は、std::stringがクラスでないため想定通りに動かない）。
 - SOLIDの原則やデザインパターン、イデオム等を適切に使用することにより、依存関係を適切に保つ。
 - 依存関係の伝搬を回避したい場合、Pimplイデオムを使い実装の詳細を隠蔽する。

- 上位概念が下位概念に依存することを避ける場合、「依存関係逆転の原則(DIP)」での例やObserver等を適用する。
- パッケージ間の相互、循環依存関係を作らない。



二重読み込みの防御

- 二重インクルードを防ぐため、ヘッダファイルには#includeガードを付ける。ガード用のマクロは、<パス名>_<ファイル名>_H_とする。

```
// @@@ example/programming_convention/lib/inc/xxx.h 1
// lib/inc/xxx.hでの#includeガード

#ifndef LIB_INC_XXX_H_
#define LIB_INC_XXX_H_

extern void XxxInitialize();

...

#endif // LIB_INC_XXX_H_
```

- コンパイラが#pragma onceをサポートしている場合は、上記方法ではなく下記をヘッダファイル先頭に記述する。

```
// @@@ example/programming_convention/lib/inc/xxx.h 16
#pragma once
```

ヘッダファイル内の#include

- 不要な依存関係を作らないようにするため、ヘッダファイルは、そのコンパイルに不要なヘッダファイルをインクルードしない。
- ヘッダファイルが外部からインポートする型(class、struct、enum)のデリファレンスがそのヘッダファイル内で不要な場合、前方宣言を使い依存関係を小さくする。

```
// @@@ example/programming_convention/header.h 3
// Pod0, Pod1の定義は別ファイルでされていると前提。
struct Pod0;
struct Pod1;

// 下記関数宣言のコンパイルには、Pod0、Pod1の完全な定義は必要ない。
extern void forward_decl(Pod0 const* pod_0, Pod1* pod_1) noexcept;
extern Pod1 forward_decl(Pod0 const* pod_0) noexcept;
extern void forward_decl(Pod0 pod_0) noexcept;

// @@@ example/programming_convention/header_ut.cpp 12
// Pod0, Pod1の定義がなくても宣言があるためコンパイルできる
forward_decl(nullptr, nullptr);

// 下記のソースコードのコンパイルには、Pod0の定義が必要なのでコンパイルできない
// forward_decl(nullptr);
```

#includeするファイルの順番

- ユーザ定義ヘッダファイルより、システムヘッダファイルの#includeを先に行う。
- システムヘッダファイルは、アルファベット順に#includeを行う。
- ユーザ定義ヘッダファイルは、アルファベット順に#includeを行う。

#includeで指定するパス名

- ユーザ定義のヘッダファイルは”“で囲み、システムヘッダファイルは、<>で囲む。

- 他のパッケージの外部非公開ヘッダファイルを読み込まないようにするために、#includeのパス指定に”..”(ディレクトリ上方向への移動)を使用しない。

```
// @@@ example/programming_convention/lib/header.cpp 1

#include <string> // OK

#include "../h/suppress_warning.h" // NG 上方向へのファイルパスは禁止

#include "../header.h" // NG 上方向へのファイルパスは禁止
#include "inc/xxx.h" // OK
```

- ヘッダファイル以外のファイル(.cppファイル等)をインクルードしない。

スコープ

スコープの定義と原則

この章で扱うスコープを下記のように定義する(「パッケージの実装と公開」参照)。

1. グローバル
2. パッケージ外部公開名前空間
3. パッケージ外部非公開名前空間
4. ファイル(無名名前空間と関数外static)
5. クラス内
6. 関数内
7. ブロック内

リンクの観点からは、2と3の識別子は同じスコープを持つが、その識別子はパッケージ外部非公開なヘッダファイルに宣言、定義されているため、パッケージ外から(まともな方法では)アクセスできない。

- 識別子のスコープは最小になるように配置する。
 - 1のスコープを持つ識別子を宣言しない(特にグローバルなスコープ内のオブジェクトは定義しない)。
 - 2、3のスコープを持つ静的な変数を宣言しない。
 - パッケージ外部公開ヘッダファイルでの識別子の宣言、定義を最小にする。
 - クラス内部でのみ使用される識別子は、privateもしくはprotectedで宣言する。
 - 一つの.cppファイルのみで使用する識別子は、その.cppファイル内の無名名前空間で宣言、定義する(staticを使用しない)。
 - 単一関数のみで使用する変数は、その関数内で定義する。
 - 自動変数は、使用直前に定義する。
- スコープが重複する「名前のない名前空間内」(例えば、ブロックとそれを内包するブロック)にある「同一名を持つ識別子」を宣言、定義しない(「name-hiding」参照)。

```
// @@@ example/programming_convention/scope.h 7

extern uint32_t xxx; // NG 外部から参照可能な静的変数
extern uint64_t yyy; // NG 同上
```

```
// @@@ example/programming_convention/scope_ut.cpp 18

uint32_t xxx; // NG 外部から参照可能な静的変数
uint32_t yyy; // NG 同上

uint32_t f(uint32_t yyy) noexcept
{
    auto xxx = 0; // NG 関数外xxxと関数内xxxのスコープが重なっており区別が付きづらい

    return xxx + yyy;
}
// なお、
// scope.h内では、 uint64_t yyy;
// scope.cpp内では、 uint32_t yyy;
// となっており、宣言と定義が矛盾している。
// この問題は、このファイルからscope.hをインクルードすれば防げる。
```

名前空間

- グローバル名前空間には下記以外の識別子を定義、宣言しない。
 - main関数

- グローバルnewのオーバーロード
- Cとシェアする識別子
- パッケージ毎に名前空間を定義する(「パッケージの実装と公開」、名前空間名参照)。
- 外部リンクエージの不要な識別子は.cpp内の無名名前空間で宣言、定義する。
- 名前空間Xxx内で定義されたテンプレートやinline関数から参照されるため、外部公開用ではないにもかかわらずヘッダファイル内に定義が必要な識別子は、名前空間Xxx::Inner_内で定義する。名前空間Xxx::Inner_内の識別子は、単体テストを除き他のファイルから参照しない。

```
// @@@ example/programming_convention/scope2.h 9

template <size_t N>
class StaticString { // StaticStringは外部公開
    ...
};

namespace Inner_ { // equal_nは外部非公開
template <size_t N>
constexpr bool equal_n(size_t n, StaticString<N> const& lhs, StaticString<N> const& rhs) noexcept
{
    if (n == N) {
        return true;
    }
    else {
        return lhs.String()[n] != rhs.String()[n] ? false : equal_n(n + 1, lhs, rhs);
    }
} // namespace Inner_

template <size_t N1, size_t N2> // operator==は外部公開
constexpr bool operator==(StaticString<N1> const&, StaticString<N2> const&) noexcept
{
    return false;
}

template <size_t N> // operator==(は外部公開)
constexpr bool operator==(StaticString<N> const& lhs, StaticString<N> const& rhs) noexcept
{
    return Inner_::equal_n(0, lhs, rhs);
}
```

using宣言/usingディレクティブ

- 識別子のインポートのためのusing宣言は下記のような場合のみに使用する(「継承コンストラクタ」、「オーバーライドとオーバーロードの違い」参照)。

```
// @@@ example/programming_convention/scope_ut.cpp 55

using std::string; // NG この関数内でのstd::stringの使用箇所が少ないのであれば、
                  // using宣言ではなく、名前修飾する
auto s_0 = string{"str"}; // NG
auto s_1 = std::string{"str"}; // OK

// 大量のstd::stringリテラルを使用する場合
using std::literals::string_literals::operator""s; // OK

auto s_2 = "str"s; // OK
// ...
auto s_N = "str"s; // OK

// クラス内でのusing宣言
struct Base {
    void f() {};
};

struct Derived : Base {
    using Base::Base; // OK 継承コンストラクタ
    using Base::f; // OK B::fのインポート
    void f(int){};
};
```

- 下記のような場合を除き、usingディレクティブは使用しない。

```
// @@@ example/programming_convention/scope_ut.cpp 84

using namespace std; // NG
```

```

auto s0 = string{"str"};
auto s1 = std::literals::string_literals::operator""s("str", 3); // NG
static_assert(std::is_same_v<std::string, decltype(s1)>);

using namespace std::literals::string_literals; // OK 例外的にOK

auto s2 = "str"s;
static_assert(std::is_same_v<std::string, decltype(s2)>);

```

- 出荷仕向け等の理由を除き、inline namespaceを使用しない(「プリプロセッサ命令」参照)。

```

// @@@ example/programming_convention/scope_ut.cpp 106

namespace XxxLib {
namespace OldVersion {
int32_t f() noexcept
{
    ...
}
} // namespace OldVersion

inline namespace NewVersion { // NG inline
int32_t f() noexcept
{
    ...
}
} // namespace NewVersion

int32_t g() noexcept
{
    return f(); // NG NewVersion::f()が呼ばれる。
}

```

```

// @@@ example/programming_convention/scope_ut.cpp 141
// 例外的にOKな例

#ifndef SHIP_TO_JAPAN
#define INLINE_JAPAN inline
#define INLINE_US
#define INLINE_EU
#endif

#ifndef SHIP_TO_US
#define INLINE_JAPAN
#define INLINE_US inline // OK
#define INLINE_EU
#endif

#ifndef SHIP_TO_EU
#define INLINE_JAPAN
#define INLINE_US
#define INLINE_EU inline // OK
#endif

#else
static_assert(false, "SHIP_TO_JAPAN/US/EU must be defined");
#endif

namespace Shipping {
INLINE_JAPAN namespace Japan // OK
{
    int32_t DoSomething() { return 0; }
}

INLINE_US namespace US // OK
{
    int32_t DoSomething() { return 1; }
}

INLINE_EU namespace EU // OK
{
    int32_t DoSomething() { return 2; }
}
} // namespace Shipping

```

```

// @@@ example/programming_convention/scope_ut.cpp 183

// SHIP_TO_JAPAN/US/EUを切り替えることで、対応したDoSomethingが呼ばれる
// この例ではSHIP_TO_JAPANが定義されているため、Shipping::Japan::DoSomethingが呼ばれる
ASSERT_EQ(0, Shipping::DoSomething());

// 名前修飾することで、すべてのDoSomethingにアクセスできるため、単体テストも容易

```

```
ASSERT_EQ(0, Shipping::Japan::DoSomething());
ASSERT_EQ(1, Shipping::US::DoSomething());
ASSERT_EQ(2, Shipping::EU::DoSomething());
```

- 演習usingディレクティブ

ADLと名前空間による修飾の省略

- 名前空間の修飾を省略した識別子のアクセスには、下記のような副作用があるため、ADLを使用する目的以外で使用しない（「識別子の命名」を順守することで、識別子の偶然の一一致を避けることも必要）。

```
// @@@ example/programming_convention/scope_ut.cpp 200

namespace NS_0 {
    class X {};

    std::string f(X, int32_t) // 第2引数int32_t
    {
        return "in NS_0";
    }
} // namespace NS_0

namespace NS_1 {

    std::string f(NS_0::X, uint32_t) // 第2引数uint32_t
    {
        return "in NS_1";
    }

    TEST(ProgrammingConvention, adl)
    {
        // in NS_1
        // 下記関数fの探索名前空間には、
        // * 第1引数の名前空間がNS_0であるため、ADLにより、
        // * この関数の宣言がNS_1で行われているため、
        // NS_0、NS_1が含まれる。
        // これにより、下記fの候補は、NS_0::f、NS_1::fになるが、第2引数1がint32_t型であるため、
        // 下記は、NS_0::fの呼び出しになる。

        ASSERT_EQ("in NS_0", f(NS_0::X(), 1)); // NS_0::fが呼ばれる。

        ASSERT_EQ("in NS_1", NS_1::f(NS_0::X(), 1)); // NS_1::fの呼び出しには名前修飾が必要
    }
} // namespace NS_1
```

名前空間のエイリアス

- ネストされた長い名前空間を短く簡潔に書くための名前空間エイリアスは、関数スコープで宣言する。
- 名前空間のエイリアスをusing宣言/usingディレクティブで使用しない。

```
// @@@ example/programming_convention/scope_ut.cpp 238

std::vector<std::string> find_files_recursively(
    std::string const& path, std::function<bool(std::filesystem::path const&)> condition)
{
    namespace fs = std::filesystem; // OK 長い名前を短く

    auto files = std::vector<std::string>{};
    auto parent = fs::path{path.c_str()};

    using namespace fs; // NG エイリアスをusing namespaceで使用しない

    std::for_each(fs::recursive_directory_iterator{parent}, // OK namespaceエイリアス
                 fs::recursive_directory_iterator{},           // OK namespaceエイリアス
                 ...
    );

    using fs::recursive_directory_iterator; // NG エイリアスをusing宣言で使用しない

    std::for_each(recursive_directory_iterator{parent}, // NG
                 recursive_directory_iterator{},           // NG
                 ...
    );
}

return files;
}
```

ランタイムの効率

- ・ ランタイム効率と、可読性のトレードオフが発生する場合、可読性を優先させる。
- ・ やむを得ず可読性を落とすコードオプティマイゼーションを行う場合は、プロファイリング等を行い、ボトルネックを確定させ、必要最低限に留める。また、開発早期での可読性を落とすコードオプティマイゼーションは行わない。

前置/後置演算子の選択

- ・ 後置演算子の一般的な動作は、下記のようになるため前置演算子の実行に比べて処理が多い。どちらを使用してもよい場合は前置演算子を使う。
 1. 自分(オブジェクト)をコピーする。
 2. 自分に前置演算子を実行する。
 3. コピーされたオブジェクトを返す。

```
// @@@ example/programming_convention/runtime_ut.cpp 14

class A {
public:
    A& operator++() noexcept // 前置++
    {
        ++a_; // メンバ変数のインクリメント
        return *this;
    }

    A operator++(int) noexcept // 後置++
    {
        A old{*this}; // リターンするためのオブジェクト
        ++(*this); // 前置++
        return old; // oldオブジェクトのリターン(オブジェクトのコピー)
    }

    operator int() const noexcept { return a_; }

private:
    int32_t a_{0};
};
```

```
// @@@ example/h/measure_performance.h 4

// パフォーマンス測定用
template <typename FUNC>
std::chrono::milliseconds MeasurePerformance(uint32_t count, FUNC f)
{
    auto const start = std::chrono::system_clock::now();

    for (auto i = 0U; i < count; ++i) {
        f();
    }

    auto const stop = std::chrono::system_clock::now();

    return std::chrono::duration_cast<std::chrono::milliseconds>(stop - start);
}
```

```
// @@@ example/programming_convention/runtime_ut.cpp 41

constexpr auto count = 10000000U;

auto a_post = A{};
auto post   = MeasurePerformance(count, [&a_post] {
    a_post++; // NG 効率が悪い。
});

auto a_pre = A{};
auto pre   = MeasurePerformance(count, [&a_pre] {
    ++a_pre; // OK 上記に比べると効率が良い。
});

ASSERT_GT(post, pre); // 前置++の処理は後置++より効率が良い。
```

- ・ ソースコードの統一性のため、このオーバーヘッドがない基本型についても、同じルールを適用する。

operator X、operator x=の選択

- 前置/後置演算子と同じような問題が発生するため、どちらを使っても問題ない場合は、operator Xではなく、operator X=を使う。

```
// @@@ example/programming_convention/runtime_ut.cpp 68

class A {
public:
    explicit A(int32_t a) noexcept : a_{a} {}

    A& operator+=(A const& rhs) noexcept
    {
        a_ += rhs.a_;
        return *this;
    }
    ...
    friend A operator+(A const& lhs, A const& rhs) noexcept // メンバ関数に見えるが、非メンバ関数
    {
        A tmp{lhs}; // operator +=に対して、
        tmp += rhs; // 「tmpを作り、それを返す」をしなければならない。
        return tmp;
    }
};

void f() noexcept
{
    auto a = A{1};
    auto b = A{2};

    a = a + b; // NG 無駄な動作が多い。
    std::cout << "a:" << a.GetA() << std::endl; // a:3と表示

    a += b; // OK
    std::cout << "a:" << a.GetA() << std::endl; // a:5と表示
}
```

- ソースコードの統一性のため、このオーバーヘッドがない基本型についても、同じルールを適用する。

関数の戻り値オブジェクト

- 基本型やenum、std::unique_ptr<>、std::optional<>等のサイズの小さいクラス以外のオブジェクトを関数の戻り値にしない。
- [注意] ローカルオブジェクトに対してRVO(Return Value Optimization)が有効であれば、そのオブジェクトを戻り値にしても良い。
- [注意] stdのコンテナは、RVOが有効でなくてもmoveが行われるため、関数の戻り値として使用しても良い。
- [注意] std::stringについては、RVOに加えて、SSO(Small String Optimization)が使用されていることが多い。そのようなコンパイラを使用している場合、std::stringは小さいオブジェクトとして扱って良い。

```
// @@@ example/programming_convention/runtime_ut.cpp 115

struct HugeClass {
    int32_t a{0};
    int32_t array[100000]{};
};

HugeClass f() noexcept // NG 巨大なオブジェクトのリターン
{
    auto obj = HugeClass{};

    ...

    return obj; // RVOが使えない場合パフォーマンス問題を引き起こす可能性がある。
}

class A {
public:
    // RVO、SSOをサポートしているコンパイラを使用している場合、下記の2つのGetNameの
    // パフォーマンスに大差はない(ほとんどのC++コンパイラはRVO、SSOをサポートしている)。
    // 使い勝手は、std::string GetName()の方が良い。
    static std::string GetName() // OK この程度なら問題はない
    {
        return "sample";
    }

    static void GetName(std::string& s) // OK
    {
        s = "sample";
    }
};
```

move処理

- ディープコピーの実装を持つクラスへのcopy代入の多くがrvalueから行われるのであれば、moveコンストラクタや、move代入演算子も実装する。
- 関数の戻り値にローカルオブジェクトを使用する場合、[RVO\(Return Value Optimization\)](#)の阻害になるため、そのオブジェクトをstd::moveしない。

```
// @@@ example/programming_convention/runtime_ut.cpp 150

std::string MakeString(int a, int b)
{
    auto ret = std::string{};

    ...
    // 文字列操作
    ...

#ifndef NDEBUG
// NG
// std::moveのため、RVOが抑止される。
// -Wpessimizing-moveを指定してg++/clang++でコンパイルすれば、
// "moving a local object in a return statement prevents copy elision"
// という警告が出る。

    return std::move(ret);
#else
// OK
// ローカルオブジェクトには通常RVOが行われるため、std::moveするよりも無駄が少ない。

    return ret;
#endif
}
```

std::string vs std::string const& vs std::string_view

- 文字列を受け取る関数の仮引数の型に関しては下記のような観点に気を付ける。以下に示す通り、このような仮引数の型をstd::string const&にすることが最適であるとは限らない。

```
// @@@ example/programming_convention/runtime_ut.cpp 184
// テスト0用関数

void f0([[maybe_unused]] std::string const& str) {}
void f1([[maybe_unused]] std::string str) {}
void f2([[maybe_unused]] std::string_view str) {}

// @@@ example/programming_convention/runtime_ut.cpp 195
// テスト0-0

auto str      = std::string{_func_};
auto f0_msec = MeasurePerformance(10000000, [&str] { f0(str); });
auto f1_msec = MeasurePerformance(10000000, [&str] { f1(str); });
auto f2_msec = MeasurePerformance(10000000, [&str] { f2(str); });

// このドキュメントを開発しているPCでは上記の結果は以下の様になる。
// f0 : 50 msec
// f1 : 222 msec
// f2 : 55 msec
// つまり、f0 < f2 < f1であり、f0とf2は大差がなく、f1は極めて非効率である。
// 従って、文字列を関数に渡す場合の引数の型は、
// std::string const&か、std::string_viewとするのが効率的である。
```

```
// @@@ example/programming_convention/runtime_ut.cpp 219
// テスト0-1

auto f0_msec = MeasurePerformance(10000000, [] { f0(_func_); });
auto f1_msec = MeasurePerformance(10000000, [] { f1(_func_); });
auto f2_msec = MeasurePerformance(10000000, [] { f2(_func_); });

// このドキュメントを開発しているPCでは上記の結果は以下の様になる。
// f0 : 674 msec
// f1 : 662 msec
// f2 : 115 msec
// つまり、f2 < f1 < f0であり、f0、f1は極めて非効率である。
// 従って、文字列を関数に渡す場合の引数の型は、std::string_viewとするのが効率的である。
//
// テスト0-0、テスト0-1の結果から、
// * 文字列リテラルからstd::string型テンポラリオブジェクトを作るような呼び出しが多い場合、
```

```
//     std::string_view
//     * 上記のような呼び出しがほとんどない場合、std::string const&
// を引数型とすべきである。
// 使用方法が想定できない場合、極めて非効率なテスト0 - 1のf0のパターンを避けるため、
// std::string_viewを選択すべきだろう。
```

```
// @@@ example/programming_convention/runtime_ut.cpp 248
// テスト1用クラス
```

```
class A0 {
public:
    A0(std::string const& str) : str_{str} {}

private:
    std::string str_;
};

class A1 {
public:
    A1(std::string str) : str_{std::move(str)} {}

private:
    std::string str_;
};

class A2 {
public:
    A2(std::string_view str) : str_{str} {}

private:
    std::string str_;
};
```

```
// @@@ example/programming_convention/runtime_ut.cpp 303
// テスト1-1
```

```
auto a0_msec = MeasurePerformance(10000000, [] { A0 a{__func__}; });
auto a1_msec = MeasurePerformance(10000000, [] { A1 a{__func__}; });
auto a2_msec = MeasurePerformance(10000000, [] { A2 a{__func__}; });

// このドキュメントを開発しているPCでは上記の結果は以下の様になる。
// A0 :834 msec
// A1 :774 msec
// A2 :704 msec
// つまり、A2 < A1 < A0であり、A0の効率がやや悪い。
// 従って、文字列を関数に渡す場合の引数の型は、
// std::stringか、std::string_viewとするのが効率的である。
//
// コンストラクタのインターフェースとしては、
// 実引数オブジェクトのライフタイムを考慮しなくて良いため、A0よりもA1の方が優れている。
// この観点と、テスト1 - 0、テスト1 - 1の結果を総合的に考えれば、
// このような場合の引数の型は、std::stringを選択すべきだろう。
```

extern template

- 何度もインスタンス化が行われ、それによりROMの肥大化やビルドの長時間化を引き起こすようなクラステンプレートに対しては、`extern template`を使う。

```
// @@@ example/programming_convention/string_vector.h 4
// このファイルをインクルードすると、
// そのファイルでのstd::vector<std::string>のインスタンス化は抑止される。
extern template class std::vector<std::string>;
```

```
// @@@ example/programming_convention/string_vector.cpp 3
// std::vector<std::string>はこのファイルでインスタンス化される。
template class std::vector<std::string>;
```

標準クラス、関数の使用制限

STL

- C++のバージョン毎に定められた非推奨の機能、関数、クラスを使わない。これらについては、[C++日本語リファレンス](#)の
 - [C++11の機能変更](#)
 - [C++14の機能変更](#)

- C++17の機能変更に詳細が書かれている。
- g++/clang++等の優れたコンパイラを適切なオプションで使用することで、非推奨の機能、関数、クラスの使用を防ぐ。

スマートポインタの使用制限

- std::auto_ptrを使用しない(C++17で廃止)。
- ダイナミックに生成したオブジェクトの管理にはstd::unique_ptr<>を使用する。 std::unique_ptr<>では機能が足りない場合のみ、 std::shared_ptr<>を使用する。

配列系コンテナクラスの使用制限

- 配列系のコンテナを使用する場合、コンパイル時に要素数の上限が
 - 定まるのであれば、 std::arrayを使用する。
 - 未定ならば、 std::vectorを使用する。
- std::vector<bool>は、 std::vectorの特殊化であり、通常のstd::vectorと同じように扱えない。 std::vector<bool>を使用する場合、その要素へのハンドルがbool&やbool*でないことに注意する。
- std::arrayを除くコンテナクラスは、それ自体でメモリリソースのRAII(scoped_guard)を実現しているため、 newしない。

std::stringの使用制限

- std::stringは、それ自体でメモリリソースのRAII(scoped_guard)を実現しているため、 newしない。
- std::stringの添字演算子[]は領域外アクセスを通知しない(std::out_of_rangeエクセプションを発生させない)ため、 std::string::at()を使用する(「安全な配列型コンテナ」参照)。
- std::string::data()は、C++のバージョンによってはNULLターミネイトが保証されていないため、 std::string::c_str()を使用する。

std::string_viewの使用制限

- std::string_viewが保持するポインタが指す文字列の所有権は、他のオブジェクトが保持しているため、 std::string_viewの初期化や、 copy代入の右辺にrvalueを使用しない。

```
// @@@ example/programming_convention/string_view_ut.cpp 11

auto str = std::string{"abc"};
auto sv = std::string_view{str}; // OK lvalueからの初期化

ASSERT_EQ(sv, std::string_view{"abc"});
```

```
// @@@ example/programming_convention/string_view_ut.cpp 31

std::string_view sv = std::string("abc"); // NG rvalueからの初期化           // この行でstd::string{"abc"}が解放
                                         // svは無効なポインタを保持

ASSERT_EQ(sv, std::string_view{"abc"}); // svは無効なポインタを保持
```

- 文字列リテラルで初期化されたstd::string_viewと文字列リテラルとの微妙な違いに気を付ける。

```
// @@@ example/programming_convention/string_view_ut.cpp 43

{ // 文字列リテラルを範囲として使用すると、ヌル文字が要素に含まれる
    auto oss = std::ostringstream{};

    for (char c : "abc") {
        oss << c;
    }

    ASSERT_EQ((std::string{'a', 'b', 'c', '\0'}), oss.str()); // ヌル文字が入る
}

{ // string_viewを使用すると、ヌル文字が要素に含まれない
    auto oss = std::ostringstream{};

    for (char c : std::string_view{"abc"}) {
        oss << c;
    }

    ASSERT_EQ((std::string{'a', 'b', 'c'}), oss.str()); // ヌル文字は入らない
}
```

```
// @@@ example/programming_convention/string_view_ut.cpp 65

char const a[]{"123"};
auto b = std::string_view{"01234"}.substr(1, 3); // インデックス1 - 3

ASSERT_EQ(a, b); // a == bが成り立つ
```

```

auto oss_a = std::ostringstream{};
oss_a << a;

auto oss_b = std::ostringstream{};
oss_b << b;

ASSERT_EQ(oss_a.str(), oss_b.str()); // ここまででは予想通り

// bをインデックスアクセスすると以下のようになる。
ASSERT_EQ('0', b[-1]); ASSERT_EQ('1', b[0]); ASSERT_EQ('2', b[1]);
ASSERT_EQ('3', b[2]); ASSERT_EQ('4', b[3]); ASSERT_EQ('\0', b[4]);

// 上記の結果から、以下の結果になることには注意が必要
auto oss_b_cstr = std::ostringstream{};
oss_b_cstr << b.data(); // data()は文字列リテラルへのポインタを指す。

ASSERT_NE(oss_a.str(), oss_b_cstr.str());
ASSERT_EQ("123", oss_a.str());
ASSERT_EQ("1234", oss_b_cstr.str());

```

POSIX系関数

使用禁止関数一覧

禁止関数	代替え
alloca()	コンテナ
asctime()	strftime()
asctime_r()	strftime()
bcmp()	
bcopy()	
brk()	
bzero()	
ctermid()	
ctime()	strftime()
ctime_r()	strftime()
cuserid()	
ecvt()	
execl()	execle(), execve()
execlp()	execle(), execve()
execv()	execle(), execve()
execvp()	execle(), execve()
fattach()	
fcvt()	
fdetach()	
ftw()	
gcvt()	
getc()	
getchar()	
getgrgid()	getgrgir_r()
getgrnam()	getgnam_r()
getitimer()	
getlogin()	getlogin_r()
getmsg()	
getopt()	
getpmsg()	
getpwuid()	getpwuid_r()
getpwnam()	getpwnam_r()
gets()	fgets()
getitimer()	timer_gettime()
gettimeofday()	clock_gettime() (戻り値を確認すること)

禁止関数	代替え
getw()	
getwd()	
gmtime()	gmtime_r()
index()	
ioctl() (stropts.hに定義されているもの)	
isascii()	
isastream()	
localtime()	localtime_r()
_longjmp()	
mktemp()	
popen()	execle(), execve()
pthread_getconcurrency()	
pthread_setconcurrency()	
putc()	
putchar()	
putenv() に autoな変数のポインタ	setenv()
putmsg()	
putpmsg()	
rand()	rand()
rand_r()	rand()
readdir()	readdir_r()
rindex()	
sbrk()	
scanf()	sscanf()
_setjmp()	
setpgrp()	
settimer()	timer_settimer()
sighold()	pthread_sigmask() または sigprocmask()
sigignore()	
siginterrupt()	
signal()	signalfd()
sigpause()	sigsuspend()
sigrelse()	pthread_sigmask() または sigprocmask()
sigset()	sigaction()
sigstack()	
strcpy()	strncpy()
strcat()	strncat()
strlen()	strnlen()
strtok()	
sprintf()	snprintf()
system()	execle(), execve()
tmpnam()	tmpfile(), mkdtemp(), mkstemp()
tmpnam()	tmpfile(), mkdtemp(), mkstemp()
toascii()	
_tolower()	tolower()
_toupper()	toupper()
ttyname()	ttyname_r()
ttyslot()	
ulimit()	getrlimit(), setrlimit()
utime()	utimensat()
utimes()	
valloc()	
vfork()	fork()

禁止関数	代替え
vfprintf()	vsnprintf()
wcsat()	wcsncat()
wcsncpy()	wcsncpy()

使用禁止関数の理由や注意点

バッファオーバーランを引き起こしやすい関数

- 以下の関数は、バッファオーバーフロー等のバグを引き起こしやすい。

`gets(), scanf(), strcpy(), strcat(), sprintf(), vsprintf(), wcsat(), wcsncpy()`

コマンドインジェクション防止

- 以下の関数は、外部コマンドの実行時に環境変数に依存してしまう。

`exec(), execp(), execv(), execvp(), popen(), system()`

*obsolete*関数

- 以下の関数は、すでにメンテナンスがされなくなった(obsolete)。

```
asc_time(), asctime_r(), ctime(), ctime_r(), fattach(), fdetach(), ftw(), getitimer(),
getmsg(), getpmsg(), gets(), settimer(), gettimeofday(), ioctl() in stropts.h for stream,
isascii(), isastream(), _longjmp(),
pthread_getconcurrency(), pthread_setconcurrency(), putmsg(), putpmsg(), rand_r(),
_setjmp(), settimer(),
setpgroup(), sighold(), sigignore(), siginterrupt(), sigpause(), sigrelse(), sigset(),
strlen(), _tolower(), _toupper(), tempnam(), tmpnam(), toascii(), ulimit(), utime()
```

LEGACY関数

- 以下の関数は、すでに役目を終えた。

`sigstack(), cuserid(), getopt(), getw(), ttyslot(), valloc(), ecvt(), fcvt(),
gcvt(), mkttemp(), bcmp(), bcopy(), bzero(), index(), rindex(), utimes(), getwd(),
brk(), sbrk(), rand()`

スレッドセーフでない関数

- 以下の関数は、スレッドセーフでない。

`asctime(), ctime(), getgrgid(), getgrnam(), getlogin(), getpwuid(), getpwnam(), gmtime(),
localtime(), ttyname(),
ctermid(), tmpnam() (引数がNULLのとき、非リエントラントになる)`

標準外関数等

- 可変長配列や、alloca()は、標準外である。

扱いが難しい関数

- signalの扱いは極めて難しく、安定動作をさせるのは困難である。「シグナルのリエントラント問題を解決でき、使用できる関数に制限がない」という利点があるため、signal()の代わりに、signalfd()を使用する。
- 排他的にファイルをオープンできないため、tmpfile()を使用しない。代わりにmkstemp()を使用する。

典型的な注意点

リソースリークを引き起こしやすい関数

- open()/close()、fopen()/fclose()はリソースリークを引き起こしやすい。「[RAII\(scoped_guard\)](#)」で例示したコードやstd::fstreamを使うことでその問題を回避する。

シンボリックリンクの検査

- シンボリックリンクはlstat()のみで検査せず、以下のように検査する。

1. ファイル名をlstat()
2. ファイルをopen()
3. 2で取得したファイル記述子に対してfstat()
4. 1, 3の情報を照合して同一ファイルであることを確認

strncpy(), strncat()の終端

- 下記のような問題を回避するために文字列操作にはstd::stringを使用する。
 - sizeof(dst) <= strlen(src) の場合、strncpy(dst, src, sizeof(dst) - 1)の呼び出しは、dstの文字列を'\0'終端しない。
 - コピーすべきデータが無くなると、dstの残りを'\0'で埋めるので性能上の問題がある。
 - strncpy(), strncat()ともに、sizeof(dst) < strlen(src) のときにsrcの文字列が切り捨てられたことを判別できない。

TOCTOU (Time Of Check, Time Of Use)

- open()前にaccess()でファイルの存在を確認する等、チェックして使用するパターンでは、この動作がアトミックに行われないため問題が発生する。この問題回避の一般解はないが「ファイルの存在確認後、read-open」のような場合には、「いきなりread-openし、エラーした場合に対処」することでアトミックな処理にできる。

メモリアロケーション

- new/deleteとmalloc/freeの混在を避けるため下記の使用をしない。newしたオブジェクトのポインタをfreeした場合、そのオブジェクトのデストラクタが呼び出されず、リソースリークしてしまうことがある。

`malloc(), realloc(), free()`

非同期シグナル

- プロセス監視のSIGCHLDや、accept()でのブロッキングの中断等、シグナルでしか処理できない場合を除き、非同期シグナルを使用しない。
- 使用してもよいシグナルは、以下に限られる。
 - SIGHUP : デーモン制御
 - SIGINT : 端末の割り込みキー
 - SIGILL : 不正なハードウェア命令
 - SIGBUS : ハードウェアFAULT
 - SIGFPE : 算術演算例外
 - SIGSEGV : 不正なメモリ参照
 - SIGALRM : タイムアウト検知
 - SIGTERM : killで送られるデフォルト終了シグナル
 - SIGCHLD : プロセス監視
- 上記シグナルを扱う場合であっても、シグナル処理専用スレッドでsigwait()を用いることで、非同期シグナルを同期的に扱うようとする。

その他

assertion

- 論理的にありえない状態(特に論理的に到達しないはずの条件文への到達)を検出するために、assert()を使用する(「switch文」、「if文」参照)。
- assert()はコンパイルオプションにより無効化されることがあるため、assert()の引数に副作用のある式を入れない。
- ランタイムでなく、コンパイル時に判断できる論理矛盾や使用制限には、static_assertを使用する。

```
// @@@ example/programming_convention/etc.cpp 12

template <uint32_t SIZE>
struct POD {
    POD() noexcept
    { // 何らかの理由で、10を超えるSIZEをサポートしたくない。
        static_assert(SIZE < 10, "too big");
    }

    uint32_t mem[SIZE];
};
```

```

void f() noexcept
{
    POD<3> p3;           // コンパイル可能
    auto p4 = POD<4>{}; // コンパイル可能
    // POD<10> p10;      // static assertion failed: too big でコンパイルエラー
    // POD<11> p11;       // static assertion failed: too big でコンパイルエラー
}

```

- static_assert、assert両方が使える場合には、static_assertを優先して使用する。

- 演習-アサーションの選択

- 演習-assert/static_assert

アセンブラー

- アセンブラー関数は、.asm等で定義し、ヘッダファイルでCの関数として宣言する。
- アセンブラー関数も、関数/メンバ関数のルールに従う(「非メンバ関数/メンバ関数」参照)。
- インラインアセンブラーや、それを含む関数型マクロがソースコード全域に広がらないようにする。

言語拡張機能

- #pragma once以外で、且つそれ以外に実装方法がない場合を除き、コンパイラ独自の言語拡張機能を使用しない。
- オブジェクトのアライメントが必要な場合、
 - alignas、alignofを使用する(「固定長メモリプール」参照)。
 - コンパイラ独自のアライメント機能(#pragma等)の使用を避ける。
- 繰り返し使用する#pragmasに関しては、_Pragma演算子とマクロを組み合わせて使用する。コンパイラの警告には従うべきであるが、ごく稀に無視せざるを得ない場合がある。そういう場合は下記例のような方法で抑止する。

```

// @@@ example/programming_convention/etc.cpp 38

#if defined(__clang__)
#define SUPPRESS_WARN_CLANG_UNUSED_PRIVATE_FIELD \
    _Pragma("clang diagnostic ignored \"-Wunused-private-field\"")
#else
#define SUPPRESS_WARN_CLANG_UNUSED_PRIVATE_FIELD
#endif

#define SUPPRESS_WARN_GCC_BEGIN _Pragma("GCC diagnostic push")
#define SUPPRESS_WARN_GCC_END _Pragma("GCC diagnostic pop")
#define SUPPRESS_WARN_GCC_NOT_EFF_CPP _Pragma("GCC diagnostic ignored \"-Weffc++\"")
#define SUPPRESS_WARN_GCC_UNUSED_VAR _Pragma("GCC diagnostic ignored \"-Wunused-variable\"")

// ...
// ...

SUPPRESS_WARN_GCC_BEGIN;
SUPPRESS_WARN_GCC_UNUSED_VAR;
SUPPRESS_WARN_GCC_NOT_EFF_CPP;
SUPPRESS_WARN_CLANG_UNUSED_PRIVATE_FIELD;

class A {
public:
    A() noexcept
    {
        // 警告: 'PragmaSample::A::b' should be initialized in
        //       the member initialization list [-Weffc++]
        // 警告: unused variable 'c' [-Wunused-variable]
        // のようなワーニングが出力される。

        int32_t c;
        b_ = 0;
    }

private:
    int32_t a_{0};
    int32_t b_;
};

SUPPRESS_WARN_GCC_END;

```

特に重要なプログラミング規約

本章で取り上げた規約は、重要度という観点で様々なレベルのものが混在するため量も多く、すぐに実践することが難しいかもしれない。そういう場合は、まずは特に重要な下記リストを守ることから始めるのが良いだろう。

- 浮動小数点型となるべく使わない([浮動小数点型](#))。
- const/constexprを積極的に使用する([const/constexprインスタンス](#), [メンバ関数](#))。
- すべてのインスタンスは定義と同時に初期化する([インスタンスの初期化](#))。
- クラスのpublicメンバ関数は最大7個([メンバの数](#))。
- クラスのメンバ変数は最大4個([メンバの数](#))。
- クラスのメンバ変数はprivateのみ([アクセスレベルと隠蔽化](#))。
- クラスのメンバ変数はコンストラクタ終了時までに初期化する([非静的なメンバ変数/定数の初期化](#))。
- friendは使用しない([アクセスレベルと隠蔽化](#))。
- 派生は最大2回([継承/派生](#))。
- 関数は小さくする([サイクロマティック複雑度](#))。
- 関数の仮引数は最大4個([実引数/仮引数](#))。
- グローバルなインスタンスは使わない([スコープ](#))。
- throw, try-catchは控えめに使用する([エクセプション処理](#))。
- 構文に関しては以下に気を付ける。
 - if, else, for, while, do後には{}を使う([複合文](#))。
 - switchでのフォールスルーをしない([switch文](#))。
 - switchにはdefaultラベルを入れる([switch文](#))。
 - 範囲for文を積極的に使う([範囲for文](#))。
 - gotoを使用しない([goto文](#))。
- オブジェクトのダイナミックな生成にはstd::make_unique<>を使用する ([メモリアロケーション](#))。
- Cタイプのキャストは使用しない([キャスト](#)、[暗黙の型変換](#))。

コード解析

本ドキュメントでは、ソースコードの品質を向上するために下記のような様々な方法を推奨する。

- 自動単体テスト
- 自動統合テスト
- コードインスペクション
- ツールによるコード解析

ツールによるコード解析(以下、単にコード解析と呼ぶ)とは、開発対象のソフトウェアの仕様とは無関係に発見できるバグやその類、セキュリティホール、コーディングルール不順守、その他のソースコード記述の問題を発見するための検出手段であり、動的、静的に分類される。静的コード解析には、以下のようなものがある。

1. ソースコードの記述自体の解析
2. 実行形式バイナリコードの解析
3. 各種メトリクスやリバースエンジニアリングツールを用いた解析

本章では、無償のオープンソースで実施できる静的解析¹と動的解析について解説を行う。

この章の構成

[コンパイラによる静的解析](#)

[scan-buildによる静的解析](#)

[cppcheck静的解析](#)

[sanitizerによる動的解析](#)

[まとめ](#)

コンパイラによる静的解析

コンパイラによる静的解析とは、コンパイラの警告出力を使用する解析である。静的解析の中で最も手軽に実施することができるが、意外なほど多くのソフトウェア開発でおざなりにされている。

多くのコンパイラでは警告をエラーとして扱うオプションが用意されているため、それをオンにしたビルドをすることで、多くのバグやバグの元となり得るコードを排除できる。

g++の警告機能

本ドキュメントのサンプルコードは、以下のようなg++/clang++の警告機能を使用してビルドを行っている。

```
-Werror -Wall -Wextra -Weffc++
```

また、演習で使用するコードに関しては、あえて問題のあるコードを記述するため、下記のようなオプションを使用し一部の警告を抑止している。

```
// @@@ exercise/programming_convention_q/Makefile 4  
  
SUPPRESS_WARN=-Wno-effc++ -Wno-unused-variable -Wno-delete-incomplete -Wno-unused-function \  
-Wno-sizeof-array-argument -Wno-unused-parameter -Wno-conversion-null \  
-Wno-literal-conversion
```

実際のコードによりこの効果を例示する。

```
// @@@ example/code_analysis/code_analysis.cpp 24  
  
int32_t x{-1};  
uint32_t y{1};  
  
bool b0{x < y};  
ASSERT_FALSE(b0); // 数学では成立する x < y が成立しない  
  
++x, ++y;  
bool b1{x < y};  
ASSERT_TRUE(b1); // x、yが正になれば x < y が成立する
```

上記コードでは、`int32_t`である`x`と`uint32_t`である`y`を比較することにより、`x`が`uint32_t`に型変換されるため、数学的には自明な `x < y` が成立しない。「整数型」で述べたルールに違反したために発生する問題であるが、その検出はg++により下記のように行うことができる。

```
// @@@ example/code_analysis/warnings/GCC.txt 2

code_analysis.cpp:29:15: warning: comparison of integer expressions of different signedness: ‘int32_t’ {aka ‘int’} and ‘uint32_t’
  {aka ‘unsigned int’} [-Wsign-compare]
  29 |     bool b0{x < y};
  |     ~~^~~
code_analysis.cpp:33:15: warning: comparison of integer expressions of different signedness: ‘int32_t’ {aka ‘int’} and ‘uint32_t’
  {aka ‘unsigned int’} [-Wsign-compare]
  33 |     bool b1{x < y};
  |     ~~^~~
```

次のコードは、Pimplパターンの誤った実装によってメモリリークを引き起こす（「delete」参照）。

```
// @@@ example/code_analysis/code_analysis.cpp 46

class Pimpl {
public:
    Pimpl();
    ~Pimpl() { delete core_; } // 不完全型のdelete
private:
    class PimplCore;
    PimplCore* core_;
};

class Pimpl::PimplCore {
public:
    PimplCore() : x_{new X} {}
    ~PimplCore() { delete x_; } // ~PimplCore()から呼び出されない

private:
    X* x_;
};

Pimpl::Pimpl() : core_{new PimplCore} {}

void incomplete_class()
{
    // ~Pimpl()では、クラスPimplCoreが不完全型なので~PimplCore()が呼び出されないため、
    // x_の解放がされずメモリリークする
    auto pimpl = Pimpl{};
}
```

このように連続的に記述されている場合は、コードインスペクションで発見できるかもしれないが、クラスの規模がある程度大きくなれば、このような問題を目視で発見することは容易ではない。一方でg++は下記のように、いとも簡単にそれを指摘する。

```
// @@@ example/code_analysis/warnings/GCC.txt 20

code_analysis.cpp: In destructor ‘Pimpl::~Pimpl()’:
code_analysis.cpp:51:16: warning: possible problem detected in invocation of ‘operator delete’ [-Wdelete-incomplete]
  51 |     ~Pimpl() { delete core_; } // 不完全型のdelete
  |     ~~~~~
code_analysis.cpp:51:23: warning: invalid use of incomplete type ‘class Pimpl::PimplCore’
  51 |     ~Pimpl() { delete core_; } // 不完全型のdelete
  |     ~~~
code_analysis.cpp:53:11: note: forward declaration of ‘class Pimpl::PimplCore’
  53 |     class PimplCore;
  |     ~~~~~
```

また、「move処理」で触れたようなパフォーマンスに悪影響のある下記のようなコードに対しても、g++は適切な指摘をすることができる。

```
// @@@ example/code_analysis/code_analysis.cpp 76

std::string prevent_copy_elision()
{
    auto ret = std::string("prevent copy elision");

    return std::move(ret); // std::moveのためにRVOが阻害される
}

void rvo_inhibition()
{
    // RVOが機能すればstd::stringのコンストラクタは一度だけ呼び出される
    std::string a = prevent_copy_elision();
```

```
// @@@ example/code_analysis/warnings/GCC.txt 45

code_analysis.cpp: In function ‘std::string prevent_copy_elision()’:
code_analysis.cpp:82:21: warning: moving a local object in a return statement prevents copy elision [-Wpessimizing-move]
  82 |     return std::move(ret); // std::moveのためにRVOが阻害される
     |     ~~~~~^~~~~~^~~~~~
code_analysis.cpp:82:21: note: remove ‘std::move’ call
```

clang++の警告機能

clang++にもg++と同様の優れた警告機能が備わっているが、それらは実装が異なるため、下記のような混乱を引き起こすコードに対して（「[オーバーライド](#)」参照）、clang++は、g++ができない問題点の指摘を行うことができる。

```
// @@@ example/code_analysis/code_analysis.cpp 92

class OverloadVirtualBase {
public:
    OverloadVirtualBase() = default;
    virtual ~OverloadVirtualBase() = default;
    virtual void DoSomething(int32_t) noexcept // 派生クラスがオーバーライドできる。
{
    ...
}
};

class OverloadVirtualDerived : public OverloadVirtualBase {
public:
    // シグネチャが異なるためOverloadVirtualBase::DoSomethingのオーバーライドではない。
    virtual void DoSomething(uint32_t) noexcept
{
    ...
}
};
```

```
// @@@ example/code_analysis/warnings/CLANG.txt 19

code_analysis.cpp:108:18: warning: 'OverloadVirtualDerived::DoSomething' hides overloaded virtual function [-Woverloaded-virtual]
    virtual void DoSomething(uint32_t) noexcept
    ^
code_analysis.cpp:98:18: note: hidden overloaded virtual function 'OverloadVirtualBase::DoSomething' declared here: type mismatch
    at 1st parameter ('int32_t' (aka 'int') vs 'uint32_t' (aka 'unsigned int'))
    virtual void DoSomething(int32_t) noexcept // 派生クラスがオーバーライドできる。
```

こういった問題があるため、両コンパイラによるコンパイルを薦める。

scan-buildによる静的解析

scan-buildはclang++をベースにした静的解析ツールであり、コンパイラの警告機能では指摘できないバグやバグの元となり得るコードを指摘できる。

まずは、問題のあるコードを以下に示す。

```
// @@@ example/code_analysis/code_analysis.cpp 115

class IllegalShallowCopy {
public:
    IllegalShallowCopy() : x_{new X} {}
    ~IllegalShallowCopy() { delete x_; }

private:
    X* x_;

void illegal_shallow_copy()
{
    auto a = IllegalShallowCopy{};
    auto b = IllegalShallowCopy{};

    a = b; // a.x_ = b.x_が行われるため、代入前のa.x_は解放されず、
           // 代入後のb.x_は2度deleteされる
}
```

上記クラスIllegalShallowCopyは、オブジェクトをnewにより生成し、そのポインタをメンバ変数として持つ。このようなクラスに対してはディープコピーを実装するか、オブジェクトのコピーを禁止すべきであることは、「[コンストラクタ](#)」で述べた通りである。

こういったコードに対して、g++/clang++はその問題を発見できないが、scan-buildは下記のように適切な指摘を行うことができる。

```
// @@@ example/code_analysis/warnings/scan-build.txt 1

code_analysis.cpp:120:29: warning: Attempt to free released memory [cplusplus.NewDelete]
~IllegalShallowCopy() { delete x_; }
^~~~~~
```

次に示すのは、「RAII(scoped guard)」に従わなかつたために発生した潜在的バグを含んだコードである。

```
// @@@ example/code_analysis/code_analysis.cpp 138

void potential_leak()
{
    X* x{new X};

    if (global == 2) { // globalが2ならメモリリーク
        return;
    }

    delete x;
}
```

こういったコードに対しても以下に示す通り適切なメッセージを出力する。

```
// @@@ example/code_analysis/warnings/scan-build.txt 7

code_analysis.cpp:145:9: warning: Potential leak of memory pointed to by 'x' [cplusplus.NewDeleteLeaks]
    return;
^~~~~~
```

scan-buildは以下のような方法で簡単に使用できるため、C++でのソフトウェア開発における必須アイテムの一つであるといえる。

```
> scan-build make
```

cppcheck静的解析

cppcheckはscan-buildと同様な静的解析ツールであり、コンパイラの警告機能では指摘できないバグやバグの元となり得るコードを指摘できる。

まずは、問題のあるコードを以下に示す。

```
// @@@ example/code_analysis/code_analysis.cpp 152

int32_t array_access(int32_t index)
{
    uint32_t array[8]{};

    return array[index];
}

void array_stash_read_overflow()
{
    array_access(8); // off-by-1 このようなコードは意外なほど多い
}
```

問題は、配列への不正アクセスであり、これは未定義動作につながる典型的なバグであるが、scan-buildでは発見できない。

cppcheckはこういったコードに対して以下に示す通り適切なメッセージを出力する。

```
// @@@ example/code_analysis/warnings/cppcheck.txt 1

[1mcode_analysis.cpp:158:17: [31error:[39m Array 'array[8]' accessed at index 8, which is out of bounds.
[arrayIndexOutOfBounds][0m
    return array[index];
^

[1mcode_analysis.cpp:163:18: [2mnote:[0m Calling function 'array_access', 1st argument '8' value is 8
    array_access(8); // off-by-1 このようなコードは意外なほど多い
^

[1mcode_analysis.cpp:158:17: [2mnote:[0m Array index out of bounds
    return array[index];
^
```

このバグは、後述するsanitizerで発見できるものの、静的解析で発見できた方が当然ながら好ましい。

cppcheckは以下のような方法で簡単に使用できるため、C++でのソフトウェア開発における必須アイテムの一つであるといえる。

```
> bear make --always-make      # compile_commands.jsonの生成
> cppcheck --project=compile_commands.json 2> cppcheck_bugs.txt
```

sanitizerによる動的解析

本ドキュメントで扱うsanitizerとは、無償で利用できるC/C++動的解析ツールである。sanitizerオプションをオンにしたg++/clang++でテスト対象をビルドし、生成された実行形式バイナリを駆動することで使用することができる。

サンプルコードをsanitizerで解析するために使用したg++/clang++のコンパイルオプションを以下に示す。

```
-fsanitize=address,leak,undefined,float-divide-by-zero,float-cast-overflow
```

これらのオプションは、g++/clang++共通である。

このオプションを使用した実行形式コードがどのように動作するのかを例示するために、まずは、動的解析対象のコードを下記する。

```
// @@@ example/code_analysis/code_analysis.cpp 167

class NonVirtualDestructorBase {
public:
    NonVirtualDestructorBase() noexcept {}
    ~NonVirtualDestructorBase() { std::cout << __func__ << std::endl; }
};

class NonVirtualDestructorDerived : public NonVirtualDestructorBase {
public:
    NonVirtualDestructorDerived(char const* str) : str_(std::make_unique<std::string>(str)) {}
    ~NonVirtualDestructorDerived() { std::cout << __func__ << std::endl; }
    std::string const& Get() const noexcept { return *str_; }

private:
    std::unique_ptr<std::string> str_;
};

void non_virtual_destructor()
{
    // ~NonVirtualDestructorBase()がvirtualであるため、aの解放時に~NonVirtualDestructorDerived()
    // が呼び出されないことによってNonVirtualDestructorDerived::str_はリークする。
    std::unique_ptr<NonVirtualDestructorBase> a{std::make_unique<NonVirtualDestructorDerived>("D")};
}
```

上記コードは、「継承/派生」で説明した内容(基底クラスのデストラクタはvirtual)に反するため、メモリ管理にstd::unique_ptr<>を使用しているにもかかわらずメモリリークを引き起こす。g++/clang++/scan-build/cppcheckはこの問題を指摘できないが、sanitizerは以下のような出力によりメモリリークを指摘することができる。

```
// @@@ example/code_analysis/warnings/sanitizer.txt 4

==4691==ERROR: AddressSanitizer: new-delete-type-mismatch on 0x602000000410 in thread T0:
object passed to delete has wrong type:
size of the allocated type: 8 bytes;
size of the deallocated type: 1 bytes.
#0 0x7f79c7b5422f in operator delete(void*, unsigned long) ../../../../../../src/libasan/asan/asan_new_delete.cpp:172
#1 0x55aae7e44f4f in std::default_delete<NonVirtualDestructorBase>::operator()(NonVirtualDestructorBase*) const
/usr/include/c++/11/bits/unique_ptr.h:85
#2 0x55aae7e42954 in std::unique_ptr<NonVirtualDestructorBase, std::default_delete<NonVirtualDestructorBase> >::unique_ptr()
/usr/include/c++/11/bits/unique_ptr.h:361
#3 0x55aae7e37129 in non_virtual_destructor()
/home/ichiro/ichiroprogrammer/comprehensive_cpp/example/code_analysis/code_analysis.cpp:190
#4 0x55aae7e3b7fd in exec_background(void (*))
/home/ichiro/ichiroprogrammer/comprehensive_cpp/example/code_analysis/code_analysis.cpp:366
#5 0x55aae7e3ba66 in all() /home/ichiro/ichiroprogrammer/comprehensive_cpp/example/code_analysis/code_analysis.cpp:378
#6 0x55aae7e3bafe in CodeAnalysis_others_Test::TestBody()
/home/ichiro/ichiroprogrammer/comprehensive_cpp/example/code_analysis/code_analysis.cpp:382
#7 0x55aae7e919e0 in void testing::internal::HandleSehExceptionsInMethodIfSupported<testing::Test, void>(testing::Test*, void
(testing::Test::*)(), char const*)
(/home/ichiro/ichiroprogrammer/comprehensive_cpp/example/code_analysis/sanitizer/example.exe+0x969e0)
#8 0x55aae7e8986c in void testing::internal::HandleExceptionsInMethodIfSupported<testing::Test, void>(testing::Test*, void
(testing::Test::*)(), char const*)
(/home/ichiro/ichiroprogrammer/comprehensive_cpp/example/code_analysis/sanitizer/example.exe+0x8e86c)
#9 0x55aae7e63623 in testing::Test::Run()
(/home/ichiro/ichiroprogrammer/comprehensive_cpp/example/code_analysis/sanitizer/example.exe+0x68623)
#10 0x55aae7e64050 in testing::TestInfo::Run()
(/home/ichiro/ichiroprogrammer/comprehensive_cpp/example/code_analysis/sanitizer/example.exe+0x69050)
#11 0x55aae7e649af in testing::TestSuite::Run()
(/home/ichiro/ichiroprogrammer/comprehensive_cpp/example/code_analysis/sanitizer/example.exe+0x699af)
#12 0x55aae7e74162 in testing::internal::UnitTestImpl::RunAllTests()
(/home/ichiro/ichiroprogrammer/comprehensive_cpp/example/code_analysis/sanitizer/example.exe+0x79162)
```

```

#13 0x55aae7e92b03 in bool testing::internal::HandleSehExceptionsInMethodIfSupported<testing::internal::UnitTestImpl, bool>
(testing::internal::UnitTestImpl*, bool (testing::internal::UnitTestImpl::*)(), char const*)
(/home/ichiro/ichiroprogrammer/comprehensive_cpp/example/code_analysis/sanitizer/example.exe+0x97b03)
#14 0x55aae7e8ab12 in bool testing::internal::HandleExceptionsInMethodIfSupported<testing::internal::UnitTestImpl, bool>
(testing::internal::UnitTestImpl*, bool (testing::internal::UnitTestImpl::*)(), char const*)
(/home/ichiro/ichiroprogrammer/comprehensive_cpp/example/code_analysis/sanitizer/example.exe+0x8fb12)
#15 0x55aae7e72859 in testing::UnitTest::Run()
(/home/ichiro/ichiroprogrammer/comprehensive_cpp/example/code_analysis/sanitizer/example.exe+0x77859)
#16 0x55aae7ea90aa in RUN_ALL_TESTS()
(/home/ichiro/ichiroprogrammer/comprehensive_cpp/example/code_analysis/sanitizer/example.exe+0xae0aa)
#17 0x55aae7ea9023 in main
(/home/ichiro/ichiroprogrammer/comprehensive_cpp/example/code_analysis/sanitizer/example.exe+0xae023)
#18 0x7f79c6f38d8f (/lib/x86_64-linux-gnu/libc.so.6+0x29d8f)
#19 0x7f79c6f38e3f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x29e3f)
#20 0x55aae7e35204 in _start
(/home/ichiro/ichiroprogrammer/comprehensive_cpp/example/code_analysis/sanitizer/example.exe+0x3a204)

...
SUMMARY: AddressSanitizer: new-delete-type-mismatch .../.../.../src/libsanitizer/asan/asan_new_delete.cpp:172 in operator
delete(void*, unsigned long)

```

次に示すコードは、型が違うインスタンスへの代入によりオーバーフローを起こしてしまう例である。先に示したコード同様、`g++/clang++/scan-build/cppcheck`では指摘されない問題がsanitizerにより指摘される。

```

// @@@ example/code_analysis/code_analysis.cpp 386

float x{0x1'0000'0000};
// int32_t y{x}; // yのレンジを超えるため、コンパイルエラー
int32_t y = x;

// @@@ example/code_analysis/warnings/sanitizer.txt 585

code_analysis.cpp:390:17: runtime error: 4.29497e+09 is outside the range of representable values of type 'int'

```

sanitizerは以上に示した通り極めて優れたバグ検出能力を持つが、動的解析の特性からソースコードに上記のようなバグがあってもそれが実行されなければ、そのバグは指摘されない。

また、このような指摘をするためのコードは実行形式バイナリの中にコンパイル時に組み込まれるため、以下のような問題が発生する。

- ランタイム動作が遅くなる。
- ビルド時間が延びる。
- 実行形式バイナリのファイルサイズが巨大になる。

このためプログラミングの最中に行われる動作確認や手作業でのシステムテスト(特に組み込みソフトウェア)に用いる実行形式バイナリにsanitizerを適用することは難しい。

従って、[CI\(継続的インテグレーション\)](#)の一環で行われる[自動単体テスト](#)や[自動統合テスト](#)でのsanitizerの使用を薦める。

まとめ

以上で述べてきたようにコード解析ツールにはそれぞれ得手不得手があり、完璧なものは存在しないため、これらを組み合わせてコード品質の向上に努める必要があるが、これらの実施が各プログラマによって個別に行われるのであれば、コンパイル時間の増大等による新たなロスが発生する。

上記や「[CI項目の例](#)」で述べたように自動化によって、こういったロスを回避しつつ、様々なコード解析ツールを組み合わせて使用することが効率的なプロセスの要件となる。

コーディングスタイル

スタイルが統一されていないソースコードは、それだけで可読性に劣るため、スタイルの統一は重要であるが、それにこだわりすぎれば、不毛な宗教論争が発生してしまう。そのような口論を避けスタイルを定めたとしても、その遵守が目視、手作業によって行われるならば、これもまた新たな口論になる。

こういった状況に陥ることなくソースコードの記述スタイルを統一するために、本ドキュメントでは「clang-formatを使い、デフォルトのスタイルを適切に定め、必要なら多少のカスタマイズを行い、それに従う」ことを推奨する。

以下は、clang-formatを使えない場合のスタイルの指針である。

この章の構成

インデント

インデント用文字

if、for、while、do-whileのインデント

ブロックのインデント

case、defaultのインデント

ブロック(波括弧({}))

関数シグネチャ内の'()'

クラスのアクセスレベル

スペース

文の後

コンマの後

単項演算子、二項演算子、三項演算子の前後

不要なブランク文字

三項演算子のスタイル

ポインタ型やリファレンス型インスタンスの宣言、定義の*や&の場所

行数・桁数

関数の行数

行のカラム数

ブロックの論理レベル

名前空間

clang-format

インデント

インデント用文字

- 1インデントは、4つのスペース文字で表す(ハードタブはビューアにより見え方が変わるために使用しない)。

```
// @@@ example/etc/coding_style.cpp 13

int32_t f0() noexcept
{
    auto var_x = 0;           // NG インデントが2スペース
    return var_x;
}

int32_t f1() noexcept
{
    auto var_x = 0;           // NG インデントが8スペース
    return var_x;
}

int32_t f2() noexcept
{
    int32_t var_x{0};         // NG インデントがハードタブ
    return var_x;
}
```

```

int32_t f3() noexcept
{
    auto var_x = 0;           // OK
    return var_x;
}

```

if、for、while、do-whileのインデント

- if、for、while、do-whileに付随する文は、if、for、while、do-whileより1インデント下げる(「ブロック(波括弧({}))」等を参照)。

ブロックのインデント

- ブロック内部の文はブロック開始の'{'より1インデント下げる。

case、defaultのインデント

- case、defaultのインデントはswitchと合わせる。
- case、defaultに続く文のインデントは一つ下げる。
- case、defaultの次の文は改行後に書く。
- case内をブロック化するために{...}で囲む場合、「{」は:の後に1スペースを置き、その直後に配置する。'}」はcaseと同じカラムに置く。

```

// @@@ example/etc/coding_style.cpp 47

switch (var_a) {
case 1: {                      // OK
    var_b = 1;
    break;
}
case 2: {                      // NG caseのインデントはswitchと同じカラム
    var_b = 2;
    break;
}                                // NG caseと同じカラム
case 3: {                      // NG caseから1インデント下げる
    var_b = 3;
    break;
}
case 4: var_b = 4;             // NG caseの行にそのまま処理を続けない
    break;
default: {                     // OK
    break;
}
}

```

ブロック(波括弧({}))

- 文に続く'{'は、文の後に1スペースを置き、その直後に配置する。
- '}'の後に続く文は'}'の直後に改行を入れ、同じカラムから書き始める。
- 関数宣言に続く'{'は、関数宣言の直後に改行し、直前の行の先頭と同じカラムに'{'を配置する。

```

// @@@ example/etc/coding_style.cpp 76

void f0(int32_t var_x, int32_t* var_y) noexcept { // NG {の前に改行
    if (var_x == 0)
    {                                // NG
        *var_y = 0;
    }
    else
    {                                // NG
        *var_y = 10;
    }
    return;
}

void f1(int32_t var_x, int32_t* var_y) noexcept
{
    if (var_x == 0) {                // OK
        *var_y = 0;
    }
    else {                          // OK
        *var_y = 1;
    }
    return;
}

```

```

void f2(int32_t var_x, int32_t* var_y) noexcept
{
    if (var_x > 0) // OK
    {
        *var_y = 0;
    }

    if (var_x == 0) // NG の後の改行不要
    {
        *var_y = 0;
    }
    else { // OK
        var_x = 3;
    }

    if (var_x == 0) // NG
    {
        *var_y = 0;
    } else // NG else前に改行。後ろは1スペース開けて{
    {
        var_x = 3;
    }

    if (var_x == 0) // NG if文と同じ行に{
    {
        *var_y = 0;
    }
    else // NG
    {
        *var_y = 3;
    }

    return;
}

```

関数シグネチャ内の'()'

- 関数シグネチャ内の'()'は関数名の直後に置く。
- 行が長すぎる等より関数を複数行で宣言する場合、'()'は最後の仮引数の直後か、独立の行に置く。
- 全体が一行に収まるときは、そのまま一行に書く。

```

// @@@ example/etc/coding_style.cpp 140

void function(); // OK
void function(int32_t foo, // OK
              int32_t bar);
void funcction(); // OK
int32_t foo,
int32_t bar
);
void function(int32_t hoge
              ); // NG
void function // NG (は関数の直後
(
    int32_t hoge,
    char  foo
);

```

クラスのアクセスレベル

- メンバ関数等のネストが深くなりすぎるため、クラスのアクセスレベルはインデントしない。

```

// @@@ example/etc/coding_style.cpp 161

class A {
public: // NG
    void B();
private: // OK
    void C();
};

```

スペース

文の後

- [定義] ステートメントキーワードとは、for、while、do-while、switch、try、if、else等を指す。
- ステートメントキーワードの後には1スペースを入れる。
- 関数名の直後にはスペースを入れない。

```
// @@@ example/etc/coding_style.cpp 177

for (;;) {                                // OK forの後ろにはスペース
    ...
}

for(;;){                                // NG forの後ろにはスペース
    ...
}

try {                                    // OK tryの後ろにはスペース
    ...
}
catch (std::exception const& e) {        // OK catchの後ろと{の前にはスペース
    ...
}

try{                                     // NG tryの後ろにはスペース
    ...
}
catch(std::exception const& e){          // NG catchの後ろと{の前にはスペース
    ...
}

g();                                      // OK 関数の後ろにはスペース無し

g ();                                     // NG
```

コンマの後

- 行最後の文字でないコンマ(,)の後には1スペースを入れる。

```
// @@@ example/etc/coding_style.cpp 220

for (int32_t i{0}, j{0}; i + j < 10; ++i, ++j) { // OK
    ...
}

for (int32_t i{0},j{0}; i + j < 10; ++i,++j) {    // NG ,の後ろにはスペース
    ...
}

g("%d print toooooooooooooooooooooo many characters.",
  a);                                              // OK ,の直後、スペース無し ↑
```

単項演算子、二項演算子、三項演算子の前後

- 単項演算子とオペランドの間にはスペースを入れない。
- []、->、ピリオド(.)、コンマ(,)は除き、二項演算子、三項演算子の前後には1スペースを入れる。

```
// @@@ example/etc/coding_style.cpp 241

var_a=0;                                  // NG
var_a = 0;                                 // OK

var_b[ 1 ] = 1;                            // NG
var_b[2] = 1;                             // OK

var_c+=3;                                  // NG
var_c += 3;                               // OK

if(var_a == *var_b) {                      // OK
    return var_d .c_str();                // NG
}
else {
    return var_d.c_str() + 1;   // OK
}
```

不要なブランク文字

- スペース文字は、セパレータとして適切に使用するためのものである。従って、行末に不要なブランクキャラクタを置かない。
- ファイル末に不要な改行を入れない。

三項演算子のスタイル

- 三項演算子は以下のように書く。

```
// @@@ example/etc/coding_style.cpp 265

auto ret = condition ? x : y; // ワンライナーが基本

auto ret2 = (a > b) ? x      // 行が長すぎる場合
               : y;
```

- 以下のような表記方法も認められる。この表記は、switch文やif-else-if文と同様に使える。

```
// @@@ example/etc/coding_style.cpp 278

auto max = (a > b) ? a :
           (b > c) ? b :
           (c > d) ? c :
           ...
           x;
```

ポインタ型やリファレンス型インスタンスの宣言、定義の*や&の場所

- ポインタ型やリファレンス型インスタンスの宣言、定義の*や&は型の直後に配置する。

```
// @@@ example/etc/coding_style.cpp 299

char* a; // OK
char *b; // NG 型の直後に*

int32_t& j{i}; // OK
```

- 一つの文で複数の変数の定義をしない。

```
// @@@ example/etc/coding_style.cpp 306

char* c, d; // NG dはchar*ではない
int32_t e, f; // NG
```

行数・桁数

関数の行数

- 関数の行数({から}の間)は30行までに収める。
- 「テストシーケンスを一つの関数に押し込める」という方法は一般的なため、単体テストのための関数に対しては、この制限を適用しない。

行のカラム数

- 1行の最長はコメントを含めて100カラムとする。100カラムを超える行は、適切な位置に改行を入れる。
- 以下に100カラムを超える行のスタイルを例示する(縦にそろえることを重要視する)。
 - 関数の宣言

```
// @@@ example/etc/coding_style.cpp 317

int32_t f(int32_t arg1, // OK
           int32_t arg2,
           int32_t arg3) noexcept;

int32_t g( // OK
           int32_t arg1,
```

```
    int32_t arg2,
    int32_t arg3) noexcept;
```

- 定義を伴う関数の宣言

```
// @@@ example/etc/coding_style.cpp 328

int32_t f(int32_t arg1, // OK
           int32_t arg2,
           int32_t arg3) noexcept
{
    return arg1 + arg2 + arg3;
}

int32_t g( // OK
           int32_t arg1,
           int32_t arg2,
           int32_t arg3) noexcept
{
    return arg1 + arg2 + arg3;
}
```

- 関数呼び出し

```
// @@@ example/etc/coding_style.cpp 348

int32_t ret{f(arg1,
               arg2,
               arg3)};
```

- 論理演算子は行末ではなく、行頭に配置(行中にあってもよい)

```
// @@@ example/etc/coding_style.cpp 354

if (((arg1 == arg2) && (arg2 == arg3))
    || (arg3 == 3)) {
    ret = 0;
}
```

- 代入の'='の前で改行し、'='を行頭に持ってくる。

```
// @@@ example/etc/coding_style.cpp 362

auto some_loooooooooooooog_variable // 式の最後までが100カラムに入らない場合
    = arg1 + 1;
```

- 長い文字列

```
// @@@ example/etc/coding_style.cpp 368

std::cout << "foobarfubarhoge"
        "hugahogehoge"
        "1234567890"; // 長い文字列は分割
```

ブロックの論理レベル

- 各ブロックの抽象度を揃える。

```
// @@@ example/etc/coding_style.cpp 394

void DoSomethingNG() noexcept // NG
{
    Buffer_t* buff{new Buffer_t}; // NG 抽象度が低すぎる
                                //
    buff->len = 1024;          //
    buff->buff = new uint8_t[buf->len]; //
    std::memset(buff->buff, 0, buff->len); //

    ReadFromStream(*buff);
    WriteToStorage(*buff);

    DestroyBuffer(buff);
}

void DoSomethingOK() noexcept // OK
{
    Buffer_t* buff = PrepareBuffer();
```

```
    ReadFromStream(*buff);
    WriteToStorage(*buff);

    DestroyBuffer(buff);
}
```

名前空間

- ・一般に名前空間定義の区間は縦に長いため、最後に必ず名前空間の終わりであることを示すためのコメントを記述する。
- ・ネストが深くなりすぎるため、名前空間用のインデントはしない。

```
// @@@ example/etc/coding_style.cpp 423
namespace event {

class A { // インデントなし
    ...
};

...

namespace {

void f() noexcept; // インデントなし
...

} // namespace

...
} // namespace event
```

clang-format

参考のために、サンプルソースコードに適用している clang-format を例示する。

```
// @@@ .clang-format 1

# default
BasedOnStyle: Google

# indents
AccessModifierOffset: -4
IndentCaseLabels: false
IndentWidth: 4
NamespaceIndentation: None

# alignment
AlignConsecutiveAssignments: true
AlignConsecutiveDeclarations: true
AlignOperands: true
AlignTrailingComments: true

#includeBlocks: Regroup
IncludeCategories:
- Regex:      '^<.*\.h>'
  Priority:   -10
- Regex:      '^<'
  Priority:   -9
- Regex:      'gtest_wrapper\.h'
  Priority:   -8
- Regex:      'h/.*'
  Priority:   -7
- Regex:      '.*'
  Priority:   -6
SortIncludes: true

# new line
AllowShortCaseLabelsOnASingleLine: false
AllowShortFunctionsOnASingleLine: All
AllowShortBlocksOnASingleLine: false
BreakBeforeBinaryOperators: All
BreakBeforeBraces: Custom
BraceWrapping:
  AfterClass: false
  AfterControlStatement: false
  AfterEnum: false
  AfterFunction: true
```

```
AfterNamespace: false
AfterObjCDeclaration: false
AfterStruct: false
AfterUnion: false
AfterExternBlock: false
BeforeCatch: true
BeforeElse: true
ColumnLimit: 100

# space
DerivePointerAlignment: false
PointerAlignment: Left
```

命名規則

平凡なプログラマでもソースコードを介してコンパイラにその意図を伝えることはできる。優れたプログラマはそれに加えて、ソースコードを「他のプログラマに自分の意図を伝えるためのコミュニケーション手段」と位置づけ、「ソフトウェア構成物(識別子、ファイル等)への適切な命名」を重要なプログラミング技法の一つであると考える。

適切な命名とは、少なくとも下記のようなものである。

- ・ 「Name and Conquer」に書かれている内容を守っている。
- ・ ソフトウェア構成物の意味や責務、性質を容易に想起できる。
 - 一般に使用されている、もしくはプロジェクトで定義(「略語リスト」参照)されている略語以外の略語は使用しない(特に母音の省略は悪習慣である)。
 - 対称的な概念には、対称的な名前を付ける(「言葉の対称性」参照)。
 - 変数名、クラス名等は「人(er)」や「もの」にする(と適切な場合が多い)。また、単数、複数形に気をつける。
 - メンバ関数名は動詞から始める(と適切な場合が多い)。
- ・ 非常識、時代遅れのルールに従っていないハンガリアン記法による命名は時代遅れ)。
- ・ 違う概念の構成物に同じような(紛らわしい)名前を付けない。

このような技法は、簡単に習得できることではないが、すべてのプログラマは、このことに細心の注意を払い、可読性の高いソースコードを作り出すための努力を怠ってはならない。

なお、これ以降この章では、主に名前の形式についてのルールを定める。

この章の構成

禁止事項

略語リスト

言葉の対称性

ファイル、ディレクトリの命名

ファイル名

ディレクトリ名

識別子の命名

型名

定数名

ローカル変数名(自動変数名、仮引数名、関数内static変数名)

メンバ変数名

メンバ関数名

Accessorメンバ関数名

関数名

名前空間名

テンプレート名

テンプレートの仮引数名

マクロ名

その他の命名則

禁止事項

- ・ [定義] 以下のような識別子は、C++の予約語であるため使用しない。
 - グローバルスコープを持ち、'_'で始まる識別子
 - '_'で始まり、その次が大文字の識別子
 - '____'を含む識別子(Cでは、'__'で始まる識別子)
- ・ C++の予約語と紛らわしいため、'_'で始まる名前は使用しない。

```
// @@@ example/etc/naming.cpp 12

int32_t      _max;           // NG
int32_t      max;            // OK
void         _func();        // NG
```

```

void      func();          // OK
#define _FOO_BAR_H_    // NG
#define FOO_BAR_H_     // OK

```

- システムハンガリアン的な命名は、不要な変更を発生させるため使用しない。

```

// @@@ example/etc/naming.cpp 25

int32_t s32_read_counter; // NG
int32_t read_counter;    // OK

```

- エイリアス(typedef)を除き、同じ物に複数の名前をつけない。特に、同じ構造の型を複数定義することは不要な混乱を招くだけでなく、コードクローンの原因になる(こういったソースコードを意図的に書くプログラマは、「型とインスタンスを明確に区別する」、「委譲や継承を使う」といった基本的スキルへの理解が足りていない可能性が高い)。
- 名前に列挙のための数字をつけない。

```

// @@@ example/etc/naming.cpp 32

int32_t name1;          // NG
int32_t name2;          // NG
int32_t utf16code;      // OK
int32_t storage1K;      // OK
bool   image1_onoff;    // image1が正式な名前ならOK

```

- 母音等を省略しない。

```

// @@@ example/etc/naming.cpp 42

int32_t acnt;           // NG
int32_t account;        // OK
int32_t sig_handler;    // NG
int32_t signal_handler; // OK

// その他やってしまいがちな例
// ctrl    -> control
// no      -> number
// ttl     -> total
// cb      -> callback
// fn      -> function, func

```

- 英文字の大小で識別子を区別しない。ただし、「NVI(non virtual interface)」を使用するメンバ関数名は例外とする。

```

// @@@ example/etc/naming.cpp 59

// NGの例 一見同じに見える
class XxxCallBack {
    ...
};

class XxxCallback {
    ...
};

```

- 一般的でない略語を使わない。一般的でない略語をプロジェクト横断的に定義したい場合には、略語リストを作り、それに定義する(「略語リスト」参照)。
- 対照的概念に非対称な名前(startとend、topとdown等)を付けない(「言葉の対称性」参照)。

略語リスト

プロジェクト独自の略語については下記のようなテーブルを作り、ソースコードと同じリポジトリで管理する。

略語	正式名称
cmd	command
fd	file descriptor
lhs	left-hand side
num	number
prev	previous
proc	process
ptr	pointer
rhs	right-hand side

略語	正式名称
str	std::stringのオブジェクト名
thd	std::threadのオブジェクト名
uniq	unique

言葉の対称性

「始まり」に対しては「終わり」であり、「開始」に対しては「終了」である。これと同様に対称的な意味をもつ識別子に対して、正確な対義語を使うことは重要であり、良い命名をするための一つの方法である。

プログラミングにおいてよく使われる対義語リストを下記する。

対応語	意味
add/subtract	加減算
add/delete	追加／削除
attach/detach	オブサーバ登録／解除
begin/end	位置
first/last	順番
front/back	位置
get/set	Accessor
lock/unlock	
new/old	
open/close	
previous(prev)/next	
push/pop	スタック処理
send/receive(recv)	
set up/tear down	
source/destination(src/dst)	「先」と「元」
start/stop	開始／停止
under/over	
up/down	
upper/lower	
lhs/rhs	左右

ファイル、ディレクトリの命名

ファイル名、ディレクトリ名は、ソフトウェアを構成するパッケージや、それらの構造から強い影響を受けるため、「パッケージとその構成ファイル」で定めたルールが守られていなければ、ファイル、ディレクトリへの適切な命名は困難である。従って、ここで定める命名ルールに従っていないソースコードをベースとした開発、保守を行っているプログラマが、そのソースコードをここでの名称ルールに従わせる場合、まずは「パッケージとその構成ファイル」の順守から始めなければならない。

[注意] ほとんどのバージョン管理システムはファイル名、ディレクトリ名の大文字小文字を区別するが、OSによっては、その区別がないものがあるため、この違いがトラブルを発生させることがある。それを避けるために、ファイル名、ディレクトリ名には大文字を使わない方が良い。

ファイル名

- ・ ファイル名は、以下のような形式の文字列にするか、それらを「_」によって連結した文字列にする。
 - 正規表現[a-z][a-z0-9]+で表される文字列
- ・ ファイル名は、下記のようにクラス名から生成する「ファイルの使用方法」参照)。
- ・ 単体テストのソースコードのファイル名はテスト対象のファイル名(.cpp)のベース名末尾に"_ut"を挿入することで生成する。

```
// クラス名
class XxxController;
```

```
// ファイル名
xxx_controller.cpp      // OK XxxControllerの定義
xxx_controller.h        // OK XxxControllerの宣言、定義
XxxController.cpp       // NG
XxxController.h         // NG

xxx_controller_ut.cpp   // OK xxx_controller.cppの単体テストのソースコードのファイル名
```

ディレクトリ名

- クラス名からファイル名を生成する方法と同様に、パッケージ名(=名前空間名)からディレクトリ名を生成する(パッケージ名がXxxYyyであれば、そのディレクトリ名はxxx_yyyである)。

識別子の命名

型名

- 型(class、struct、enum、それらのエイリアス)の名前は、以下のような形式の文字列にするか、それらを連結した文字列にする。ただし、文字列の連結部が大文字の連続になる場合はそれらを'_'によって区切る。
 - 正規表現[A-Z][a-zA-Z0-9]+で表される文字列
 - 正規表現[A-Z][A-Z0-9]+で表される文字列(慣習的に小文字を使わない文字列のみ。TCP等)

```
// @@@ example/etc/naming.cpp 73

class XxxController;           // OK
class xxx_control;             // NG
struct SomeStruct;             // OK
enum class SomeEnumeration;    // OK
enum class some_enumeration;   // OK
using Container = std::vector<int32_t>; // OK
class TCP_IP;                  // OK
```

- 責務がわかる名前をつける。クラスには「xxxする人」、「yyyするもの」のような命名が適している場合が多い。

定数名

- クラスや構造体のpublicな定数や、外部リンクエージを持つ定数、定数テンプレートの名前は、型名と同じルールで生成する。

```
// @@@ example/etc/naming.cpp 86

constexpr int SOME_CONSTANT{0}; // NG 定数
constexpr int SomeConstant{0};  // OK 定数

struct Xxx {
    static constexpr int SOME_CONSTANT{0}; // NG 定数
    static constexpr int SomeConstant{0};  // OK 定数
};

template <typename T>
constexpr bool IsSameAsXxxV = std::is_same_v<Xxx, T>; // OK 定数テンプレート
```

ローカル変数名(自動変数名、仮引数名、関数内static変数名)

- ローカル変数名は、以下のような形式の文字列にするか、それらを'_'によって連結した文字列にする。
 - 正規表現[a-z][a-zA-Z0-9]+で表される文字列
 - 正規表現[a-z]で表される1文字(デカルト座標でのxやy、ループ変数iやj、std::string::c_str()のc等のみ)
 - 正規表現[A-Z][A-Z0-9]+で表される文字列(慣習的に小文字を使わない文字列のみ。TCP等)
- 数字と見分けが難しい文字は単独では使用しない(o、l、_o、_l等は0や1と区別が困難)。

メンバ変数名

- クラスのメンバ変数(static const、static constexprメンバ定数を含む)の名前は、ローカル変数名と同じルールで生成した文字列の末尾に'_'を加えることにより生成する(visual studioで開発を行うプロジェクトを除きメンバ変数をm'_'で始めことはしない)。
- 構造体のメンバ変数の名前は、ローカル変数名と同じルールで生成する(末尾に'_'はつけない)。

```
// @@@ example/etc/naming.cpp 102

class Foo {
```

```

private:
    int32_t read_counter; // NG
    int32_t read_counter_; // OK
    int32_t m_ReadCounter; // NG
};

struct Hoo {
    int32_t bar; // OK
    int32_t bar_; // NG
};

```

- enumのメンバ名は、型名と同じルールで生成する。

```

// @@@ example/etc/naming.cpp 117

enum class SignalType {
    RED, // NG
    Green, // OK
    yellow // NG
};

```

メンバ関数名

- メンバ関数はオブジェクトの振る舞いを定義するためのものであるため、名前は動詞から始める(と良い場合が多いが、`std::string::c_str()`のように例外も多い)。
- 非同期なサービスを駆動するためのメンバ関数の名前には末尾に`async`をつける。
- `public`メンバ関数の名前は、型名と同じルールで生成する。
- `private`、`protected`メンバ関数の名前は、ローカル変数名と同じルールで生成する。

```

// @@@ example/etc/naming.cpp 125

class Fee {
public:
    void PrintMessage(); // OK
    void Print_Message(); // NG _は不要
    bool WatchdogExit(); // NG ExitByWatchdog等の方が良い
    bool DoSomething(); // OK
    bool doSomething(); // NG

private:
    bool print_system_message_async(); // OK privateで非同期
    void do_something(); // OK
};

```

Accessorメンバ関数名

- 慣習に従うと機能が類推しやすいため、一般的なAccessorメンバ関数には、以下のような接頭語をつける。

接頭語	意味	例
is	xxxか？	signal.IsRed()
get	xxxを返せ	signal.GetSignal()
set	xxxを設定せよ	signal.SetSignal(signal_color)
can	xxxはできるか？	signal.CanTurnOn()
has	xxxを持っているか？	signal.HasLed()

- `IsXXX()`, `CanXXX()`, `HasXXX()`等は、戻り型`bool`の`const`メンバ関数と宣言、定義する。
- 戻り値の意味を明示できないため、`bool CheckXxx()`のような名前は使用しない。
- 変更対象であるメンバ変数を明示するためにsetterメンバ関数の仮引数は、変更対象の変数名の末尾から'_'を除いた名前にする。

```

// @@@ example/etc/naming.cpp 140

class Fii {
public:
    int32_t GetFoo() const noexcept; // OK 変数の値の取得
    void SetFoo(int32_t foo) noexcept // OK 変数に値を設定
    {
        foo_ = foo; // OK setterの仮引数はメンバ名から'_'を除いたもの
    }

    bool IsCurrency() const noexcept; // OK bool値の問い合わせ
    bool IsGettableMemory() const noexcept; // OK
};

```

```

    bool HasHoge() const noexcept;           // OK
    bool CanGet() const noexcept;           // OK
    bool CheckHoge() const noexcept;         // NG true/falseの意味が明確でない

private:
    int32_t foo_;
};
```

関数名

- 非メンバ関数の名前は、型名と同じルールで生成する。

名前空間名

- 名前空間名は、型名と同じルールで生成する。
- ディレクトリが階層を持つならば、名前空間もそれと同様にする。パッケージXxx(パッケージを構成するディレクトリ名はxxx)が、サブパッケージYyy(ディレクトリ名はyyy、相対パスはxxx/yyy)を内包するのであれば、それらパッケージの名前空間名は、それぞれXxx、Xxx::Yyyである。

テンプレート名

- ファイル外部から参照されるテンプレートの名前は、型名と同じルールで生成する。
- ファイル外部から参照されないテンプレート(「名前空間」参照)の名前は、ローカル変数名と同じルールで生成する。

テンプレートの仮引数名

- templateの仮引数名は、以下のような形式の文字列にするか、定数名と同じルールで生成する。
 - 正規表現[A-Z][0-9]?で表される文字列(慣習的にはTやその周辺の大文字SやU等)

マクロ名

- 二重読み込みの防御用以外のマクロの名前は、定数名と同じルールで生成する。
- 二重読み込みの防御用のマクロの名前は、「二重読み込みの防御」で定めたルールで生成する。

その他の命名則

- コレクションクラスやコンテナクラス、それらのインスタンスに対しては、その名前に以下のような工夫をすることにより、物の集まりであることを示す。
 - 複数形にする。
 - 集合、群等を名前に取り入れる。

```
// @@@ example/etc/naming.cpp 160

class StringCollector { // OK 文字列の集合を扱うことを示す
public:
    std::vector<std::string> const& GetStrings() const; // OK 戻り値が集合であることを示す
    std::vector<std::string> const& GetString() const; // NG 複数形にしてvectorであることを
                                                       // 示すべき
private:
    std::vector<std::string> strings_; // OK インスタンスが集合であることを示す
    std::vector<std::string> string_; // NG
};
```

- 名前を構成する単語は、概念が大きい順番に左から並べる。

```
// @@@ example/etc/naming.cpp 175

// 下記例のXXX_YYY_R0は、XXX回路のYYYサブ回路のR0レジスタを表す。
volatile auto& XXX_YYY_R0 = *reinterpret_cast<uint32_t*>(0x1300'0000);
volatile auto& XXX_YYY_R1 = *reinterpret_cast<uint32_t*>(0x1300'0100);
volatile auto& XXX_YYY_Q0 = *reinterpret_cast<uint32_t*>(0x1300'0200);
volatile auto& XXX_ZZZ_R0 = *reinterpret_cast<uint32_t*>(0x1400'0200);
```

```
// @@@ example/etc/naming.cpp 183

// OK DataFormatterという概念のインターフェース クラス
class DataFormatterIF {
    ...
};
```

```
// OK DataFormatterという概念の具象クラスDataFormatterXml。XmlDataFormatterとしない。
class DataFormatterXml final : public DataFormatterIF {
    ...
};

// OK DataFormatterという概念の具象クラスDataFormatterCsv。CsvDataFormatterとしない。
class DataFormatterCsv final : public DataFormatterIF {
    ...
};
```

コメント

コメントの目的は、複雑怪奇なソースコードのエクスキューズでない。ソースコードから読み取れない情報や、ソースコードのサマリーを書くべきである。

この章の構成

[情報を付加しないコメント](#)

[コメントのスタイル](#)

[クラスのコメント](#)

[関数のコメント](#)

[enumのコメント](#)

[型エイリアスのコメント](#)

[template仮引数のコメント](#)

情報を付加しないコメント

以下のような情報を付加しないコメントは無駄であるだけではなく、可読性に悪影響を与える場合もあるため、避けるべきである。

1. 「ファイルの最後に、ファイルの最後を示すEnd of file」、「sleep(1)に対しての、1秒待つ」のようなコメントは、見れば明らかであるため不要である。
2. ソースコードのコメントアウト(#if 0等を含む)をしない。
3. リポジトリが保持している情報(annotation等)をコメントに含めない。

特に2、3のコメントを書くプログラマは、バージョン管理システムに未習熟である可能性が高い。そういうプログラマには、バージョン管理システムの書籍を読ませることを推奨する。

コメントのスタイル

有償、無名もしくは、日本でしか使われていないコメントフォーマットやそのツールを使うべきではない。本ドキュメントでは、doxygenを推奨する。

doxygenフォーマットの各要素に対する書き方を例示する。

クラスのコメント

```
// @@@ example/etc/comment.cpp 9

/// @class FileFinder
/// @brief ディレクトリ配下の特定のファイルをリカーシブに探して、その一覧を返すクラス
class FileFinder {
public:
```

関数のコメント

```
// @@@ example/etc/comment.cpp 45

/// @fn std::vector<std::string> FindFileRecursively(IsMatch is_match)
/// @brief 条件にマッチしたファイルをリカーシブに探して返す
/// @param is_match どのようなファイルかをラムダ式で指定する
/// @return 条件にマッチしたファイル名をvector<string>で返す
std::vector<std::string> FindFileRecursively(IsMatch is_match);
```

enumのコメント

```
// @@@ example/etc/comment.cpp 22

/// @enum FileSort
/// FindFileRecursivelyの条件
enum class FileSort {
    File,           ///< pathがファイル
    Dir,            ///< pathがディレクトリ
```

```
    FileNameHeadIs_f, //< pathがファイル且つ、そのファイル名の先頭が"f"
};
```

型エイリアスのコメント

```
// @@@ example/etc/comment.cpp 39

/// @typedef IsMatch
/// @brief FindFileRecursivelyの仮引数の型
using IsMatch = std::function<bool(std::filesystem::path const&);
```

template仮引数のコメント

```
// @@@ example/etc/comment.cpp 66

/// @class FixedPoint
/// @brief BASIC_TYPEで指定する基本型のビット長を持つ固定小数点を扱うためのクラス
/// @tparam BASIC_TYPE 全体のビット長や、符号を指定するための整数型
/// @tparam FRACTION_BIT_NUM 小数点保持のためのビット長
template <typename BASIC_TYPE, uint32_t FRACTION_BIT_NUM>
class FixedPoint {
public:
    FixedPoint(BASIC_TYPE integer = 0,
               typename std::make_unsigned_t<BASIC_TYPE> fraction = 0) noexcept
        : value_{get_init_value(integer, fraction)}
    {
        ...
    }
```

SOLID

SOLIDとは、オブジェクト指向(OOD/OOP)プログラミングにおいて特に重要な下記の5つの原則である。

- 単一責任の原則(SRP)
- オープン・クローズドの原則(OCP)
- リスコフの置換原則(LSP)
- インターフェース分離の原則(ISP)
- 依存関係逆転の原則(DIP)

単一責任の原則(SRP)

単一責任の原則(SRP, Single Responsibility Principle)とは、

- 一つのクラスは、ただ一つの責任(機能)を持つようにしなければならない
- 一つのクラスは、ただ一つの理由で変更されるように作られなければならない

というクラスデザイン上の制約である。

下記クラスSentenceHolderNotSRPは、一見問題ないように見えるが、std::stringの保持と、その出力という二つの責務を持つため、SRP違反である。

```
// @@@ example/solid/srp_ut.cpp 27

class SentenceHolderNotSRP {
public:
    SentenceHolderNotSRP() = default;
    ~SentenceHolderNotSRP() = default;

    void Add(std::string const& sentence) { sentence_ += sentence; }

    std::string const& Get() const noexcept { return sentence_; }

    void Save(std::string const& file)
    {
        std::ofstream o{file};
        o << sentence_;
    }

    void Display() { std::cout << sentence_; }

private:
    std::string sentence_{};
};
```

実践的にはこの程度の単純なクラスでのSRP違反が問題になることは少ないが、下記のコメントで示す通り、単体テストの実施が困難になる。

```
// @@@ example/solid/srp_ut.cpp 53

auto not_srp = SentenceHolderNotSRP{};

not_srp.Add("haha\n");
not_srp.Add("hihi\n");
not_srp.Add("huhu\n");

// SRPに従っていないため、テストが面倒
not_srp.Save(not_srp_text_); // not_srp_text_への書き込み

auto ifs      = std::ifstream{not_srp_text_};
auto ifs_begin = std::istreambuf_iterator<char>{ifs};
auto ifs_end   = std::istreambuf_iterator<char>{};
auto act       = std::string{ifs_begin, ifs_end}; // not_srp_text_ファイルの読み出し

ASSERT_EQ("haha\nhihi\nhuhu\n", act);

// SRPに従っていないため、テストできない
not_srp.Display();
```

クラスSentenceHolderNotSRPの二つの責務をクラスSentenceHolderSRPと、Output()に分離したコード実装例を下記する。

```
// @@@ example/solid/srp_ut.cpp 75

class SentenceHolderSRP {
public:
    SentenceHolderSRP() = default;
    ~SentenceHolderSRP() = default;

    void Add(std::string const& sentence) { sentence_ += sentence; }

    std::string const& Get() const noexcept { return sentence_; }

private:
    std::string sentence_{};
};

// SRPに従うために、
// SentenceHolderNotSRP::Save(), SentenceHolderNotSRP::Display()
// をクラスの外に出し、さらに仮引数に出力先(std::ostream&)を追加してこの2関数を統一。
void Output(SentenceHolderSRP const& sentence_holder, std::ostream& o)
{
    o << sentence_holder.Get();
}
```

下記のコードで示したように、この分離の効果で単体テストの実施が容易になった。

```
// @@@ example/solid/srp_ut.cpp 101

auto srp = SentenceHolderSRP{};

srp.Add("haha\n");
srp.Add("hihi\n");
srp.Add("huhu\n");

// SRPに従ったことで、ファイル操作やstd::coutへの操作が不要になり、単体テストの実施が容易
auto act = std::ostringstream{};
Output(srp, act);

ASSERT_EQ("haha\nhihi\nhuhu\n", act.str());
```

演習-SRP

オープン・クローズドの原則(OCP)

オープン・クローズドの原則(OCP, Open-Closed Principle)とは、

- クラスは拡張に対して開いて(open)いなければならず、
- クラスは修正に対して閉じて(closed)いなければならない

というクラスデザイン上の制約である。

まずは、アンチパターンから示す。

```
// @@@ example/solid/ocp_ut.cpp 14

class TransactorGoogle {
public:
    static bool Pay(Yen price) noexcept
    {
        ...
    }

    static bool Charge(Yen price) noexcept
    {
        ...
    }
};

class TransactorSuica {
    ...
};

class TransactorEdy {
public:
    ...
};

class TransactorNotOCP {
```

```

public:
    enum class TransactionMethod { Google, Suica, Edy };

    explicit TransactorNotOCP(TransactionMethod pay_method) noexcept : pay_method_{pay_method} {}

    ...
    bool Charge(Yen price) noexcept
    {
        switch (pay_method_) {
        case TransactionMethod::Google:
            return TransactorGoogle::Charge(price);
        case TransactionMethod::Suica:
            return TransactorSuica::Charge(price);
        ...
        }
    }

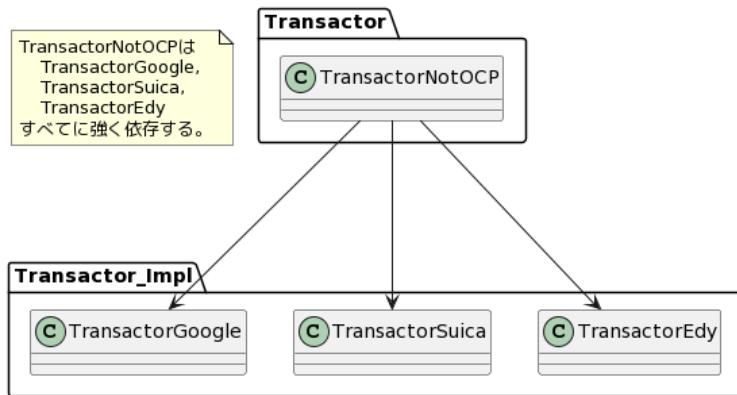
    bool Pay(Yen price) noexcept
    {
        switch (pay_method_) {
        case TransactionMethod::Google:
            return TransactorGoogle::Pay(price);
        case TransactionMethod::Suica:
            return TransactorSuica::Pay(price);
        ...
        }
    }
    ...
};


```

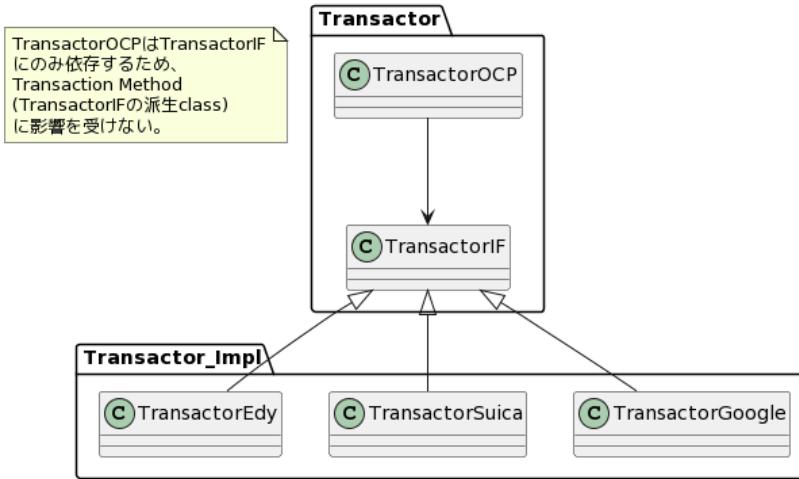
Transaction Method(enum TransactionMethod)が増えた場合、少なくとも3か所に手を入れなければならなくなる(修正に対してclosedでない)。従って、下記のTransactorNotOCP::Charge()や、TransactorNotOCP::Pay()は Transaction Methodの追加、変更に対して、脆弱な構造だと言える。

次に上記ソースコードのクラス図を下記する。

クラス図が示す通り、TransactorNotOCPは、TransactorGoogle, TransactorSuica, TransactorEdy (Transaction Methodに対応した具体的なクラス)に強く依存する。したがって、新たなTransactor Methodが追加されれば、Transaction Methodを使用しているTransactorNotOCPのすべてのメンバ関数は影響を受ける。この構造は、上位概念が下位概念に依存しているとも言えるため、後述する「[依存関係逆転の原則\(DIP\)](#)」にも反している。



下記は、TransactorIFを導入することによって、上例をOCPに沿うように改善したクラス図と実装である。TransactorOCPは、TransactorIFの効果によりTransaction Methodの追加に対して全く影響を受けなくなった(実際には、TransactorIFから派生する具象クラスの生成用Factory関数(「[Factory](#)」参照)が必要になるため全く影響がないわけではないが、そのような箇所はソースコード全体でただ一つにすることができるため、Transaction Methodの追加に対して強固な構造になったと言える)。



下記にこのクラス図に従ったコードを示す。

```

// @@@ example/solid/ocp_ut.cpp 122

class TransactorIF {
public:
    ...
    bool Charge(Yen price) noexcept { return charge(price); }
    bool Pay(Yen price) noexcept { return pay(price); }

private:
    virtual bool charge(Yen price) = 0;
    virtual bool pay(Yen price) = 0;
};

class TransactorGoogle : public TransactorIF {
    ...
};

class TransactorSuica : public TransactorIF {
    ...
};

class TransactorEdy : public TransactorIF {
    ...
};

class TransactorOCP {
public:
    explicit TransactorOCP(std::unique_ptr<TransactorIF>&& transactor) noexcept
        : transactor_{std::move(transactor)}
    {
    }

    bool Charge(Yen price) noexcept { return transactor_->Charge(price); }

    bool Pay(Yen price) noexcept { return transactor_->Pay(price); }

private:
    std::unique_ptr<TransactorIF> transactor_;
};

```

ここでは、この原則に沿う実装方法としてポリモーフィズムを使うパターンを紹介したが、[Pimpl](#)のようにラッピングを使用するパターンも有用である。

[演習-OCP](#)

リスコフの置換原則(LSP)

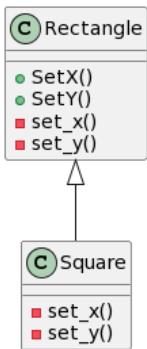
リスコフの置換原則(LSP, Liskov Substitution Principle)とは、

- 基底クラスを使っているX(関数もしくはクラス)に、基底クラスの代わりにその派生クラスを渡した場合でも、Xはその実際の型を知ること無しに正常動作できなければならない

というクラスデザイン上の制約であり、この制約を守るために下記のような契約プログラミングを行うことが求められる。

- ・事前条件を派生クラスで強めることはできない。つまり、基底クラスよりも強い事前条件を持つ派生クラスを作つてはならない。
- ・事後条件を派生クラスで弱めることはできない。つまり、基底クラスよりも弱い事後条件を持つ派生クラスを作つてはならない。

この原則に従わない実装例を示すために、以下のようなクラスRectangleとその派生クラスSquareを定義する。



```

// @@@ example/solid/lsp.h 5

/// @class Rectangle
/// @brief (0, 0) からの矩形を表す
class Rectangle {
public:
    explicit Rectangle(int x, int y) noexcept : x_{x}, y_{y} {}
    ...
    void SetX(int x) noexcept
    {
        auto temp = y_;
        set_x(x);
        assert(temp == y_); // 「set_xはy_に影響を与えない」が事後条件
    }
    ...

protected:
    virtual void set_x(int x) noexcept { x_ = x; }

    ...

private:
    int x_;
    int y_;
};

/// @class Rectangle
/// @brief (0, 0) からの正方形を表す
class Square : public Rectangle {
public:
    explicit Square(int x) noexcept : Rectangle{x, x} {}

protected:
    virtual void set_x(int x) noexcept override
    {
        Rectangle::set_x(x);
        Rectangle::set_y(x);
    }

    virtual void set_y(int y) noexcept override { set_x(y); }

};
  
```

Rectangleのリファレンスを受け取るSetX()とその単体テストを以下のようにすると、 Rectangleのテストでは問題は起ららないが、同じことをSquareに行うとアボートしてしまう（下記例ではASSERT_DEATHを使用しアボートすることを確認している）。

```

// @@@ example/solid/lsp_ut.cpp 13

void SetX(Rectangle& rect, int x) noexcept { rect.SetX(x); }

TEST(LSP_Opt, violation_abort)
{
    // Rectangleのテスト
    auto rect = Rectangle{0, 0};
    SetX(rect, 3);
    ASSERT_EQ(3, rect.GetX());

    // Squareのテスト
    auto square = Square{0};
  
```

```
    ASSERT_DEATH(SetX(square, 3), ""); // ここでRectangle::SetX()の中のassert()がfailする。
}
```

上記コードがアボート(assertion fail)してしまったのは

- Rectangle::SetX()は、この実行によるy_の値が不変であることを表明している
- この表明は、Rectangle::set_x()の事後条件となる
- Square::set_x()は、この事後条件を守らず、y_の値を変えてしまった

が原因である。このデザイン上の問題には目をつぶり(Rectangle、Squareを修正せずに)、しかもアボートしないSetX()の実装を考えてみよう。

SetX()は仮引数で渡されたオブジェクトの実際の型がわからなければアボートを避けることはできない。従って、新しいSetX()のコード実装例は以下のようになる。

```
// @@ example/solid/lsp_ut.cpp 32

void SetX(Rectangle& rect, int x) noexcept
{
    if (dynamic_cast<Square*>(&rect) != nullptr) {
        rect = Square(x);
    }
    else {
        // rectの型は、Rectangle
        rect.SetX(x);
    }
}

TEST(LSP, violation_not_abort)
{
    // Rectangleのテスト
    auto rect = Rectangle{0, 0};
    SetX(rect, 3);
    ASSERT_EQ(3, rect.GetX());

    // Squareのテスト
    auto square = Square{0};
    SetX(square, 3); // assert()はfailしない。
    ASSERT_EQ(3, square.GetX());
}
```

上記の新たなSetX()は、アボートはしないがきわめて醜悪目つ、Rectangleの全派生クラスに依存した、変更に弱い関数となる。

なお、リスコフの置換原則とは関係しないが、上記のdynamic_castを含むSetX()は、下記のように修正することができる。

```
// @@ example/solid/lsp_ut.cpp 61

void SetX(Rectangle& rect, int x) noexcept
{
    auto y = rect.GetY();
    rect = Rectangle(x, y);
}
```

このSetX()は、Rectangleからの派生クラスに依存していないため、良い解法に見える。ところが実際にはオブジェクトのスライシングという別の問題を引き起こそ。

例示した問題は結局のところデザインの誤りが原因であり、それを修正しない限り、問題の回避は容易ではない。

一般に、継承関係は、IS-Aの関係と呼ばれる。数学の世界では「正方形 is a 長方形」であるため、この関係を継承で表したのだが、「Rectangle::SetX()の性質より導き出されたRectangle::set_x()の事後条件」により、「クラスSquare is NOT a クラスRectangle」となり、SquareとRectangleは継承関係ではないため問題が発生した。

継承を用いなければこのような問題は発生しないため、public継承を使用する際には、「本当にその関係は継承で表すべきか(それが最もシンプルな方法か)?」について熟慮する必要がある。

なお、エクセプション記述子は、関数のエクセプション仕様を強制的にLSPに従わせる仕組みであるが、C++11から非推奨になり、C++17では規格から削除された。その理由は、「非推奨だった古い例外仕様を削除」の説明の通り、これを使用し場合、OCPに違反する可能性が高いからである。従って、原則に従うのみでなく、その他の原則とのバランスも考慮する必要がある。

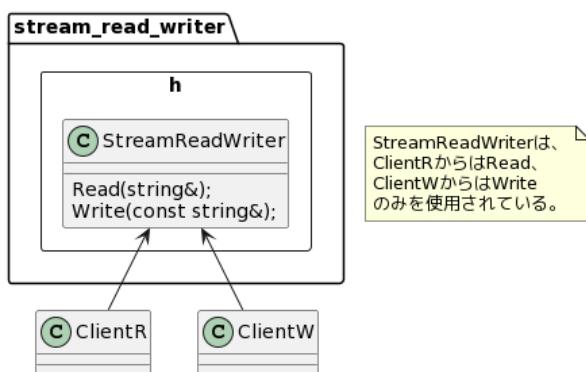
インターフェース分離の原則(ISP)

インターフェース分離の原則 (ISP, Interface Segregation Principle)とは、

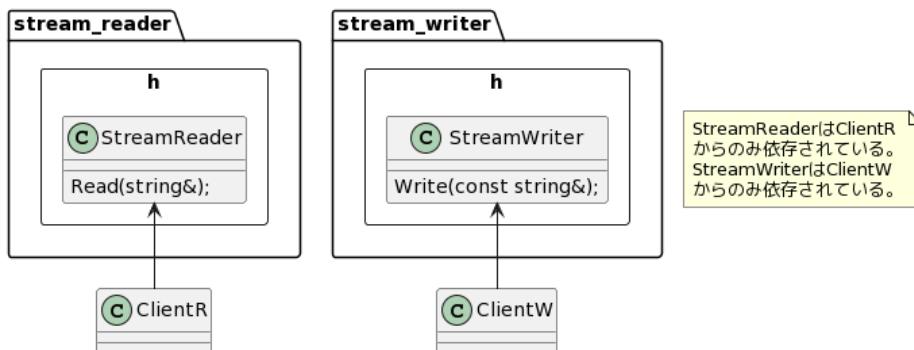
- クラスは、そのクライアントが使用しないメソッドへの依存を、そのクライアントに強制するべきではない。
 - クラスのインターフェースを巨大にしない。
 - 一つのヘッダファイルに互いが密接な関係を持たない複数のクラスを定義、宣言すべきでない。
 - 一つのヘッダファイルにそのファイルのコンパイルに不要なヘッダファイルをインクルードすべきでない。

というクラスデザイン上の制約である。

まずは、ISPに従っていない例を示す。下記のStreamReadWriteは、ClientRからはStreamWriter::Read()のみが、ClientWからはStreamWriter::Write()のみが使用されている。



ほとんどのStreamWriter使用ファイルでこのような依存関係がある場合、このクラスは下記のようにStreamReaderとStreamWriterに分割した方が良い(依存関係が小さくなる)。



クラスの設計時に統合か分割かで悩むことは多いが、一度統合してしまえば分割は困難であり、逆に分割されたものを統合することは容易である。このことを考慮すれば、このような迂回に解を与えることは簡単である。言うまでもないが、「まずは分割」が原則である。

演習-ISP

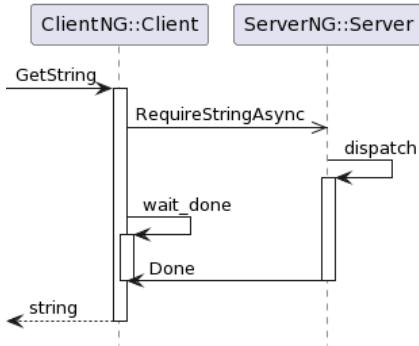
依存関係逆転の原則(DIP)

依存関係逆転の原則 (DIP, Dependency Inversion Principle)とは、

- 上位レベルのモジュールは下位レベルのモジュールに依存すべきではない。
- 抽象は具象に依存すべきではない。

というクラスデザイン上の制約である。

下記ServerNG::Serverは、ClientNG::Clientに非同期サービスを提供する(従って、ServerNG::ServerはClientNG::Clientに対して上位概念である)。



非同期サービスであるServerNG::Server::RequireStringAsync()の完了は ServerNG::ServerがClientNG::Client::Done()を呼び出すことにより通知される。

その実装、使用例を下記に示す。

```

// @@@ example/solid/dip_server_ng.h 10

namespace ServerNG {
class Server {
public:
    Server();
    void RequireStringAsync(ClientNG ::Client& client) noexcept;
    ...
};

// @@@ example/solid/dip_server_ng.cpp 6

namespace ServerNG {
namespace {
void dispatch(ClientNG::Client& client) // コマンドのディスパッチ
{
    switch (client.GetNum()) {
    case 1:
        client.Done(new std::string{"hello"});
        break;
    case 2:
        client.Done(new std::string{"good bye"});
        break;
    ...
    }
}

void thread_entry(Pipe& pipe) // サーバーのスレッド関数
{
    for (;;) {
        ClientNG::Client* client=nullptr;
        auto const      ret = pipe.Read(&client, sizeof(client));
        assert(ret == sizeof(client));

        if (client == nullptr) { // nullptr受信でサーバー終了
            break;
        }

        dispatch(*client);
    }
}
} // namespace
...

void Server::RequireStringAsync(ClientNG::Client& client) noexcept
{
    void const* const buff=&client;

    auto ret = pipe_.Write(&buff, sizeof(buff));
    assert(ret == sizeof(&client));
}
}

// @@@ example/solid/dip_client_ng.h 10

```

```

namespace ClientNG {
class Client {
public:

```

```

    explicit Client(ServerNG::Server& server) noexcept : server_{server}, pipe_{}, num_{0} {}

    std::string GetString(uint32_t num);

    void Done(std::string* str) noexcept // サーバーからクライアントへのコマンド終了通知
    {
        auto const ret = pipe_.Write(&str, sizeof(str));
        assert(ret == sizeof(str));
    }

    ...
};

} // namespace ClientNG

```

```

// @@@ example/solid/dip_client_ng.cpp 3

namespace ClientNG {
std::string Client::GetString(uint32_t num)
{
    set_num(num);
    server_.RequireStringAsync(*this);

    return *wait_done(); // 非同期通知待ち
}

std::unique_ptr<std::string> Client::wait_done()
{
    std::string* str=nullptr;
    auto const ret = pipe_.Read(&str, sizeof(str));
    assert(ret == sizeof(str));

    return std::unique_ptr<std::string>{str};
}
} // namespace ClientNG

```

```
// @@@ example/solid/dip_ut.cpp 11
```

```

TEST(DIP, ng_pattern)
{
    auto server = ServerNG::Server{};
    auto client = ClientNG::Client{server};

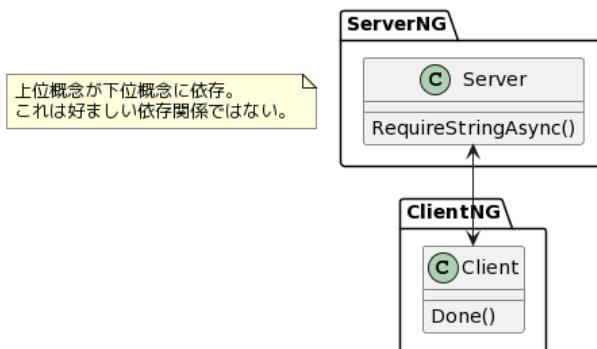
    auto actual = client.GetString(1);
    ASSERT_EQ("hello", actual);

    actual = client.GetString(2);
    ASSERT_EQ("good bye", actual);

    actual = client.GetString(3);
    ASSERT_EQ("unknown", actual);
}

```

上記ソースコードから明らかのようにServerNG::ServerとClientNG::Clientは相互に依存している。このうちの一つはサーバがクライアントに依存(上位概念が下位概念に依存)する問題のある構造となっている。



このため、クライアントのバリエーションが増えた場合、容易にServerNG::Serverのコードは肥大化する。また、ServerNG::Serverを介して各クライアント間にも(暗黙、明示両方の)依存関係が生まれやすいため、ServerNG::Serverのコード修正は非常に困難になることが予想される。

次にDIPに従い上記コードを改善した例を示す。

```
// @@@ example/solid/dip_server_ok.h 7
```

```

namespace ServerOK {
class ClientIF {
public:
    ClientIF() noexcept : num_{0} {}
    void Done(std::string* str) { done(str); } // サーバーからクライアントへのコマンド終了通知
    ...
private:
    virtual void done(std::string* str) = 0;
    ...
};

class Server {
public:
    Server();
    void RequireStringAsync(ClientIF& client) noexcept;
    ...
};
} // namespace ServerOK

```

```

// @@@ example/solid/dip_server_ok.cpp 5

namespace ServerOK {
namespace {
void dispatch(ClientIF& client) // コマンドのディスパッチ
{
    switch (client.GetNum()) {
    case 1:
        client.Done(new std::string{"hello"});
        break;
    case 2:
        client.Done(new std::string{"good bye"});
        break;
    ...
    }
}

void thread_entry(Pipe& pipe) // サーバーのスレッド関数
{
    for (;;) {
        ClientIF* client=nullptr;
        auto const ret = pipe.Read(&client, sizeof(client));
        assert(ret == sizeof(client));

        if (client == nullptr) { // nullptr受信でサーバー終了
            break;
        }

        dispatch(*client);
    }
}
} // namespace
...

void Server::RequireStringAsync(ClientIF& client) noexcept
{
    void const* const buff=&client;

    auto ret = pipe_.Write(&buff, sizeof(buff));
    assert(ret == sizeof(&client));
}
...
} // namespace ServerOK

```

```

// @@@ example/solid/dip_client_ok.h 10

namespace ClientOK {
class Client : public ServerOK::ClientIF {
public:
    explicit Client(ServerOK::Server& server) noexcept : ClientIF{}, server_{server}, pipe_{} {}

    std::string GetString(uint32_t num);

    ...
};

} // namespace ClientOK

```

```

// @@@ example/solid/dip_client_ok.cpp 3

namespace ClientOK {
std::string Client::GetString(uint32_t num)
{

```

```

    SetNum(num);
    server_.RequireStringAsync(*this);

    return *wait_done(); // 非同期通知待ち
}

std::unique_ptr<std::string> Client::wait_done()
{
    std::string* str{nullptr};
    auto const ret = pipe_.Read(&str, sizeof(str));
    assert(ret == sizeof(str));

    return std::unique_ptr<std::string>{str};
} // namespace ClientOK

```

```

// @@@ example/solid/dip_ut.cpp 28

// 使用方法は、ServerNG, ClientNGと同じ。
TEST(DIP, ok_pattern)
{
    auto server = ServerOK::Server{};
    auto client = ClientOK::Client{server};

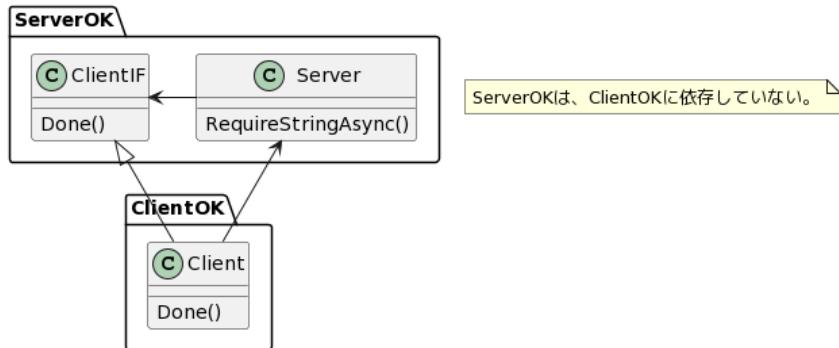
    // 以下、ng_patternと同じ
    ...
}

```

修正後のコードは、

- ServerOK::ServerはServerOK::ClientIFに依存する。
- ClientOK::ClientはServerOK::ClientIFから派生する。

このクラス図を以下に示す。



ServerNGとClientNGの双方向依存関係は、ClientOKからServerOKへの単方向依存関係へと改善され、サーバに影響を与えることなく、クラスアントの機能変更やバリエーション追加を行うことが可能となった。

[演習-DIP](#)

まとめ

以上で述べたように、SOLIDはオブジェクト指向(OOD/OOP)プログラミングにおいて極めて重要な原則である。この逸脱はソースコードを劣化させ、ソフトウェアの品質低下や開発費増大に直結するため、厳守することが求められる。

[演習-SOLIDの定義](#)

デザインパターン

ソースコードを劣化させるアンチパターンには、

- ・大きすぎる関数、クラス、ファイル等のソフトウェア構成物
- ・複雑怪奇な依存関係
- ・コードクローン

等があるだろう。こういった問題は、ひどいソースコードを書かないという強い意志を持ったプログラマの不斷の努力と、そのプログラマを支えるソフトウェア工学に基づいた知識によって回避可能である。本章ではその知識の一翼をになうデザインパターン、イデオム等を解説、例示する。

なお、ここに挙げるデザインパターン、イデオム等は「適切な場所に適用される場合、ソースコードをよりシンプルに記述できる」というメリットがある一方で、「不適切な場所に適用される場合、ソースコードの複雑度を不要に上げてしまう」という負的一面を持つ。

また、デザインパターン、イデオム等を覚えたてのプログラマは、自分のスキルが上がったという一種の高揚感や顯示欲を持つため、それをむやみやたらに多用してしまう状態に陥ることある。このようなプログラマの状態を

- ・パターン病に罹患した
- ・パターン猿になった、もしくは単に、猿になった

と呼ぶ。猿になり不要に複雑なソースコードを書かないとするために、デザインパターン、イデオム等を使用する場合、本当にそれが必要か吟味し、不要な場所への適用を避けなければならない。

この章の構成

ガード節
BitmaskType
Pimpl
Accessor
Copy-And-Swap
Immutable
Clone(仮想コンストラクタ)
NVI(non virtual interface)
RAII(scoped guard)
Future
DI(dependency injection)
Singleton
State
Null Object
Templateメソッド
Factory
Named Constructor
Proxy
Strategy
Visitor
CRTTP(curiously recurring template pattern)
Observer
MVC
Cでのクラス表現

ガード節

ガード節とは、「可能な場合、処理を早期に打ち切るために関数やループの先頭に配置される短い条件文(通常はif文)」であり、以下のような利点がある。

- ・処理の打ち切り条件が明確になる。
- ・関数やループのネストが少なくなる。

まずは、ガード節を使っていない例を上げる。

```
// @@@ example/design_pattern/guard_ut.cpp 24
/// @fn int32_t SequentialA(char const (&a)[3])
/// @brief a(配列へのリファレンス)の要素について、先頭から'a'が続く数を返す
/// @param 配列へのリファレンス
```

```

int32_t SequentialA(char const (&a)[3]) noexcept
{
    if (a[0] == 'a') {
        if (a[1] == 'a') {
            if (a[2] == 'a') {
                return 3;
            }
            else {
                return 2;
            }
        }
        else {
            return 1;
        }
    }
    else {
        return 0;
    }
}

```

上記の例を読んで一目で何が行われているか、理解できる人は稀である。一方で、上記と同じロジックである下記関数を一目で理解できない人も稀である。

```

// @@@ example/design_pattern/guard_ut.cpp 78

int32_t SequentialA(char const (&a)[3]) noexcept
{
    if (a[0] != 'a') { // ガード節
        return 0;
    }
    if (a[1] != 'a') { // ガード節
        return 1;
    }
    if (a[2] != 'a') { // ガード節
        return 2;
    }

    return 3;
}

```

ここまで効果的な例はあまりない。

もう一例、(ガード節導入の効果が前例ほど明確でない)ガード節を使っていないコードを示す。

```

// @@@ example/design_pattern/guard_ut.cpp 49

std::optional<std::vector<uint32_t>> PrimeNumbers(uint32_t max_num)
{
    auto result = std::vector<uint32_t>{};

    if (max_num < 65536) { // 演算コストが高いためエラーにする
        if (max_num >= 2) {
            auto is_num_prime = std::vector<bool>(max_num + 1, true); // falseなら素数でない
            is_num_prime[0] = is_num_prime[1] = false;
            auto prime_num = 2U; // 最初の素数

            do {
                result.emplace_back(prime_num);
                prime_num = next_prime_num(prime_num, is_num_prime);
            } while (prime_num < is_num_prime.size());
        }
    }

    return result;
}

return std::nullopt;
}

```

上記にガード節を適用した例を下記する。

```

// @@@ example/design_pattern/guard_ut.cpp 95

std::optional<std::vector<uint32_t>> PrimeNumbers(uint32_t max_num)
{
    if (max_num >= 65536) { // ガード節。演算コストが高いためエラーにする。
        return std::nullopt;
    }

    auto result = std::vector<uint32_t>{};

```

```

if (max_num < 2) { // ガード節。2未満の素数はない。
    return result;
}

auto is_num_prime = std::vector<bool>(max_num + 1, true); // falseなら素数でない。
is_num_prime[0] = is_num_prime[1] = false;
auto prime_num = 2U; // 最初の素数

do {
    result.emplace_back(prime_num);
    prime_num = next_prime_num(prime_num, is_num_prime);
} while (prime_num < is_num_prime.size());

return result;
}

```

ガード節を使っていない例に比べて、

- ネストが減って読みやすくなった
- max_numが1, 2, 65535, 65536である場合がロジックの境界値であることが一目でわかるようになった

といった改善はされたものの、最初の例ほどのレベル差はない。しかし、ソースコードの改善やリファクタリングのほとんどは、このようなものであり、この少しのレベルアップが数か月後、数年後に大きな差を生み出すことを忘れてはならない。

演習-ガード節

BitmaskType

下記のようなビットマスク表現は誤用しやすいインターフェースである。修正や拡張等に関しても脆弱であるため、避けるべきである。

```

// @@@ example/design_pattern/enum_operator.h 6

class Animal {
public:
    struct PhysicalAbility { // オブジェクトの状態を表すためのビットマスク
        static constexpr auto Run = 0b0001U;
        static constexpr auto Fly = 0b0010U;
        static constexpr auto Swim = 0b0100U;
    };

    // paにはPhysicalAbilityのみを受け入れたいが、実際にはすべてのuint32_tを受け入れる。
    explicit Animal(uint32_t pa) noexcept : physical_ability_{pa} {}

    uint32_t GetPhysicalAbility() const noexcept { return physical_ability_; }
    ...
};

// @@@ example/design_pattern/enum_operator_ut.cpp 13

Animal dolphin{Animal::PhysicalAbility::Swim}; // OK
ASSERT_EQ(Animal::PhysicalAbility::Swim, dolphin.GetPhysicalAbility());

Animal uma{0xff}; // NG 誤用だが、コンストラクタの仮引数の型がuint32_tなのでコンパイル可能

```

上記のような誤用を防ぐために、enumによるビットマスク表現を使用して型チェックを強化した例を以下に示す。このテクニックは、STLのインターフェースとしても使用されている強力なイデオムである。

```

// @@@ example/design_pattern/enum_operator.h 30

class Animal {
public:
    enum class PhysicalAbility : uint32_t {
        Run = 0b0001,
        Fly = 0b0010,
        Swim = 0b0100,
    };

    explicit Animal(PhysicalAbility pa) noexcept : physical_ability_{pa} {}

    PhysicalAbility GetPhysicalAbility() const noexcept { return physical_ability_; }

private:
    PhysicalAbility const physical_ability_;
};

// &, | &=, |=, IsTrue, IsFalseの定義

```

```

constexpr Animal::PhisicalAbility operator&(Animal::PhisicalAbility x,
                                              Animal::PhisicalAbility y) noexcept
{
    return static_cast<Animal::PhisicalAbility>(static_cast<uint32_t>(x)
                                                & static_cast<uint32_t>(y));
}

constexpr Animal::PhisicalAbility operator|(Animal::PhisicalAbility x,
                                              Animal::PhisicalAbility y) noexcept
{
    return static_cast<Animal::PhisicalAbility>(static_cast<uint32_t>(x)
                                                | static_cast<uint32_t>(y));
}

inline Animal::PhisicalAbility& operator&=(Animal::PhisicalAbility& x,
                                              Animal::PhisicalAbility y) noexcept
{
    return x = x & y;
}

...

```

```

// @@@ example/design_pattern/enum_operator_ut.cpp 28

// コンストラクタの仮引数の型が厳密になったためコンパイル不可
// これにより誤用を防ぐ
// Animal uma{0xff};

// C++17から下記はコンパイル可能となったが、アクシデントでこのようなミスはしないだろう
auto uma = Animal{Animal::PhisicalAbility{0xff}};

auto dolphin = Animal{Animal::PhisicalAbility::Swim};
ASSERT_EQ(Animal::PhisicalAbility::Swim, dolphin.GetPhisicalAbility());

auto pa = Animal::PhisicalAbility{Animal::PhisicalAbility::Run};
pa |= Animal::PhisicalAbility::Swim;

auto human = Animal{pa};
ASSERT_TRUE(IsTrue(Animal::PhisicalAbility::Run & human.GetPhisicalAbility()));

```

この改善により、Animalのコンストラクタに域値外の値を渡すことは困難になった（少なくとも不注意で間違うことはないだろう）。この修正の延長で、Animal::GetPhisicalAbility()の戻り値もenumになり、これも誤用が難しくなった。

演習-BitmaskType

Pimpl

このパターンは、「クラスA(a.cpp、a.hで宣言、定義)を使用するクラスにAの実装の詳細を伝搬させたくない」ような場合に使用する。そのためオープン・クローズドの原則(OCP)の実装方法としても有用である。

一般的に、STLライブラリのベースは多くのCPUタイムを消費する。クラスAがSTLクラスをメンバに使用し、a.hにそのSTLヘッダファイルがインクルードされた場合、a.hをインクルードするファイルをコンパイルする度にそのSTLヘッダファイルはベースされる。これはさらに多くのCPUタイムの消費につながり、ソースコード全体のビルドは遅くなる。こういった問題をあらかじめ避けるためにも有効な手段ではあるが、そのトレードオフとして実行速度は若干遅くなる。

下記は、Pimplイデオム未使用の、std::stringに依存したクラスStringHolderOldの例である。

```

// @@@ example/design_pattern/string_holder_old.h 3
// このファイルには<string>が必要

#include <memory>
#include <string>

class StringHolderOld final {
public:
    StringHolderOld();

    void      Add(char const* str);
    char const* GetStr() const;

private:
    std::unique_ptr<std::string> str_;
};

// @@@ example/design_pattern/string_holder_old.cpp 1

```

```
#include "string_holder_old.h"

StringHolderOld::StringHolderOld() : str_{std::make_unique<std::string>()} {}

void StringHolderOld::Add(char const* str) { *str_ += str; }

char const* StringHolderOld::GetStr() const { return str_->c_str(); }
```

下記は、上記クラスStringHolderOldにPimplイデオムを適用したクラスStringHolderNewの例である。

```
// @@ example/design_pattern/string_holder_new.h 3
// このファイルには<string>は不要

#include <memory>

class StringHolderNew final {
public:
    StringHolderNew();

    void Add(char const* str);
    char const* GetStr() const;
    ~StringHolderNew(); // デストラクタは.cppで=defaultで定義

private:
    class StringHolderNewCore; // StringHolderNewの振る舞いは、StringHolderNewCoreに移譲
    std::unique_ptr<StringHolderNewCore> core_;
};

// @@ example/design_pattern/string_holder_new.cpp 1
// このファイルには<string>が必要

#include <string>

#include "string_holder_new.h"

class StringHolderNew::StringHolderNewCore final {
public:
    StringHolderNewCore() = default;
    void Add(char const* str) { str_ += str; }

    char const* GetStr() const noexcept { return str_.c_str(); }

private:
    std::string str_{};
};

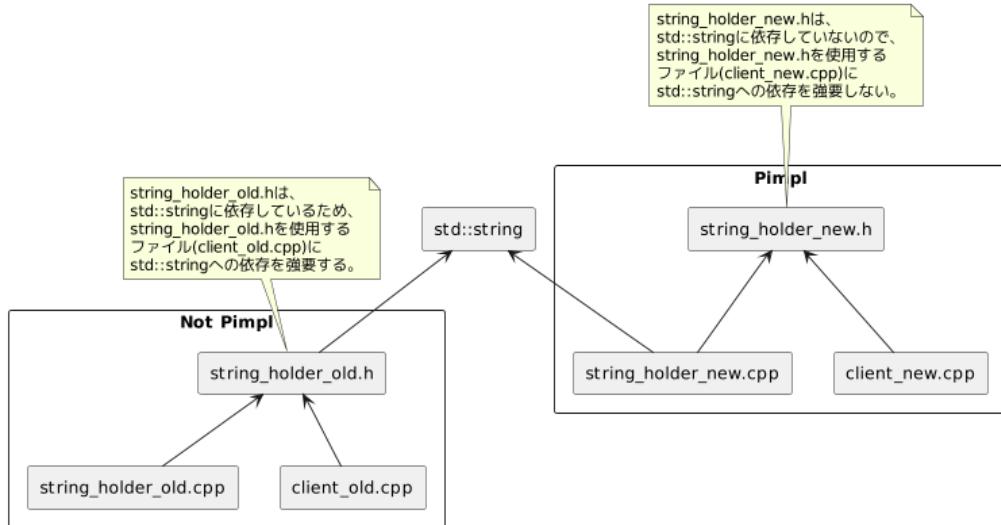
StringHolderNew::StringHolderNew() : core_{std::make_unique<StringHolderNewCore>()} {}

void StringHolderNew::Add(char const* str) { core_->Add(str); }

char const* StringHolderNew::GetStr() const { return core_->GetStr(); }

// この宣言、定義をしないと、StringHolderNewをインスタンス化した場所では、
// StringHolderNewCoreが不完全型であるため、std::unique_ptrが実体化できず、コンパイルエラーとなる。
// この場所であれば、StringHolderNewCoreは完全型であるためstd::unique_ptrが実体化できる。
StringHolderNew::~StringHolderNew() = default;
```

下記図は、上記ファイルやそれらを使用するファイルの依存関係である。string_holder_old.hは、std::stringに依存しているが、string_holder_new.hは、std::stringに依存していないこと、それによってStringHolderNewを使用するファイルから、std::stringへの依存を排除できていることがわかる。



このパターンを使用して問題のある依存関係をリファクタリングする例を示す。

まずは、リファクタリング前のコードを下記する。

```
// in lib/h/widget.h

#include "gtest/gtest.h"

class Widget {
public:
    void DoSomething();
    uint32_t GetValue() const;
    // 何らかの宣言

private:
    uint32_t gen_xxx_data(uint32_t a);
    uint32_t xxx_data_{1};
    FRIEND_TEST(Pimpl, widget_ng); // 単体テストをfriendにする
};
```

```
// in lib/src/widget.cpp

#include "widget.h"

void Widget::DoSomething()
{
    // 何らかの処理
    xxx_data_ = gen_xxx_data(xxx_data_);
}

uint32_t Widget::GetValue() const { return xxx_data_; }

uint32_t WidgetNG::Widget::gen_xxx_data(uint32_t a) { return a * 3; }
```

```
// in lib/ut/widget_ut.cpp

#include "widget.h"

TEST(Pimpl, widget_ng)
{
    Widget w;

    ASSERT_EQ(1, w.xxx_data_); // privateのテスト
    w.DoSomething();
    ASSERT_EQ(3, w.xxx_data_); // privateのテスト
    ASSERT_EQ(9, w.gen_xxx_data(3)); // privateのテスト

    ASSERT_EQ(3, w.GetValue());
}
```

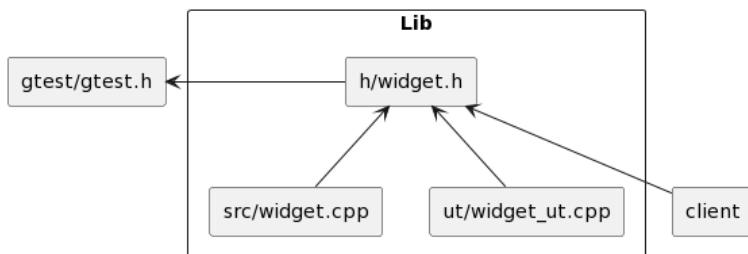
何らかの事情により、単体テストでprivateなメンバにアクセスする必要があったため、単体テストクラスをテスト対象クラスのfriendすることで、それを実現している。

単体テストクラスをテスト対象クラスのfriendにするためには、上記コードの抜粋である下記を記述する必要がある。

```
FRIEND_TEST(Pimpl, widget_ng); // 単体テストをfriendにする
```

このマクロは、gtest.h内で定義されているため、widget.hからgtest.hをインクルードしている。

このため、ファイルの依存関係は下記のようになる。



この依存関係は、Widgetのクライアントに不要な依存関係を強要してしまう問題のある構造を作り出す。

この問題をPimplによるリファクタリングで解決したコードを以下に示す(コンパイラのインクルードパスにはlib/hのみが入っていることを前提とする)。

```
// in lib/h/widget.h

#include <memory>

class Widget {
public:
    Widget(); // widget_pimplは不完全型であるため、コンストラクタ、
    ~Widget(); // デストラクタはインラインにできない
    void DoSomething();
    uint32_t GetValue() const;
    // 何らかの宣言

    struct widget_pimpl; // 単体テストのため、publicとするが、実装はsrc/の下に置くため、
    // 単体テスト以外の外部からのアクセスはできない

private:
    std::unique_ptr<widget_pimpl> widget_pimpl_;
};
```

```
// in lib/src/widget.cpp

#include "widget_internal.h"

// widget_pimpl
void Widget::widget_pimpl::DoSomething()
{
    // 何らかの処理
    xxx_data_ = gen_xxx_data(xxx_data_);
}

uint32_t Widget::widget_pimpl::gen_xxx_data(uint32_t a) { return a * 3; }

// Widget
void Widget::DoSomething() { widget_pimpl_->DoSomething(); }
uint32_t Widget::GetValue() const { return widget_pimpl_->xxx_data_; }

// ヘッダファイルの中では、widget_pimplは不完全型であるため、コンストラクタ、
// デストラクタは下記に定義する
Widget::Widget() : widget_pimpl_{std::make_unique<Widget::widget_pimpl>()} {}
Widget::~Widget() = default;
```

```
// in lib/src/widget_internal.h

#include "widget.h"

struct Widget::widget_pimpl {
    void DoSomething();
    uint32_t gen_xxx_data(uint32_t a);
    uint32_t xxx_data_{1};
};
```

```
// in lib/ut/widget_ut.cpp

#include "../src/widget_internal.h" // 単体テストのみに、このようなインクルードを認める
#include "gtest/gtest.h"

TEST(Pimpl, widget_ok)
{
    Widget::widget_pimpl wi;
```

```

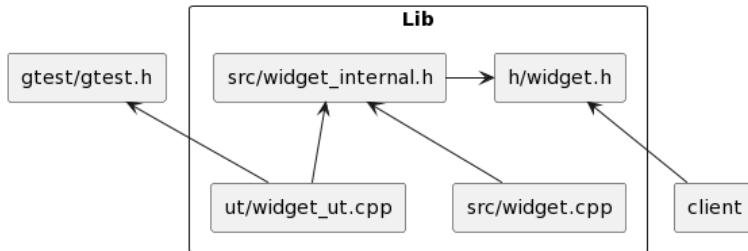
    ASSERT_EQ(1, wi.xxx_data_);
    wi.DoSomething();
    ASSERT_EQ(3, wi.xxx_data_);
    ASSERT_EQ(9, wi.gen_xxx_data(3));

    Widget w;

    w.DoSomething();
    ASSERT_EQ(3, w.GetValue());
}

```

このリファクタリングにより、ファイルの依存は下記のようになり、問題のある構造は解消された。



演習-Pimpl

Accessor

publicメンバ変数とそれにアクセスするソースコードは典型的なアンチパターンであるため、このようなコードを禁じるのが一般的なプラクティスである。

```

// @@@ example/design_pattern/Accessor_ut.cpp 8

class A { // アンチパターン
public:
    int32_t a_{0};
};

void f(A& a) noexcept
{
    a.a_ = 3;

    // Do something
    ...
}

```

とはいっても、ソフトウェアのプラクティスには必ずといってほど例外があり、製品開発の現場において、オブジェクトのメンバ変数にアクセスせざるを得ないような場面は、稀にではあるが発生する。このような場合に適用するがこのイデオムである。

```

// @@@ example/design_pattern/Accessor_ut.cpp 28

class A { // Accessorの実装例
public:
    void SetA(int32_t a) noexcept // setter
    {
        a_ = a;
    }

    int32_t GetA() const noexcept // getter
    {
        return a_;
    }
private:
    int32_t a_{0};
    ...
};

void f(A& a) noexcept
{
    a.SetA(3);

    // Do something
    ...
}

```

メンバ変数への直接のアクセスに比べ、以下のようなメリットがある。

- アクセスのログを入れることができる。
- メンバ変数へのアクセスをデバッガで捕捉しやすくなる。
- setterに都合の悪い値が渡された場合、何らかの手段を取ることができる(assertや、エラー処理)。
- リファクタリングや機能修正により対象のメンバ変数がなくなった場合においても、クラスのインターフェースの変更を回避できる(修正箇所を局所化できる)。

一方で、クラスに対するこのような細かい制御は、カプセル化に対して問題を起こしやすい。下記はその典型的なアンチパターンである。

```
// @@@ example/design_pattern/Accessor_ut.cpp 62

class A { // Accessorを使用して細かすぎる制御をしてしまうアンチパターン
public:
    void SetA(int32_t a) noexcept // setter
    {
        a_ = a;
    }

    int32_t GetA() const noexcept // getter
    {
        return a_;
    }

    void Change(bool is_changed) noexcept // setter
    {
        is_changed_ = is_changed;
    }

    bool IsChanged() const noexcept // getter
    {
        return is_changed_;
    }

    void DoSomething() noexcept // is_changed_がtrueの時に、呼び出してほしい
    {
        // Do something
        ...
    }
    ...
};

void f(A& a) noexcept
{
    if (a.GetA() != 3) {
        a.SetA(3);
        a.Change(true);
    }

    ...
}

void g(A& a) noexcept
{
    if (!a.IsChanged()) {
        return;
    }

    a.Change(false);
    a.DoSomething(); // a.IsChanged()がtrueの時に実行する。
    ...
}
```

上記ソースコードは、オブジェクトaのA::a_が変更された場合、その後、それをもとに何らかの動作を行うこと(a.DoSomething)を表しているが、本来オブジェクトaの状態が変わったかどうかはオブジェクトa自体が判断すべきであり、a.DoSomething()の実行においても、それが必要かどうかはオブジェクトaが判断すべきである。この考えに基づいた修正ソースコードを下記に示す。

```
// @@@ example/design_pattern/Accessor_ut.cpp 130

class A { // 上記アンチパターンからChange()とIsChanged()を削除し、状態の隠蔽レベルを強化
public:
    void SetA(int32_t a) noexcept // setter
    {
        if (a_ == a) {
            return;
        }
```

```

        a_      = a;
        is_changed_ = true;
    }

    void DoSomething() noexcept
    {
        if (!is_changed_) {
            return;
        }

        // Do something
        ...

        is_changed_ = false; // 状態変更の取り消し
    }
    ...
};

void f(A& a) noexcept
{
    a.SetA(3);

    ...
}

void g(A& a) noexcept
{
    a.DoSomething(); // DoSomethingは無条件で呼び出す。
                      // 実際に何かをするかどうかは、オブジェクトaが決める。
    ...
}

```

setterを使用する場合、上記のように処理の隠蔽化には特に気を付ける必要がある。

演習-Accesstorの副作用

演習-Accesstor

Copy-And-Swap

メンバ変数にポインタやスマートポインタを持つクラスに

- copyコンストラクタ
- copy代入演算子
- moveコンストラクタ
- move代入演算子

が必要になった場合、コンパイラが生成するデフォルトの特殊メンバ関数では機能が不十分であることが多い。

下記に示すコードは、そのような場合の上記4関数の実装例である。

```

// @@@ example/design_pattern/no_copy_and_swap_ut.cpp 8

class NoCopyAndSwap final {
public:
    explicit NoCopyAndSwap(char const* name0, char const* name1)
        : name0_{name0 == nullptr ? "" : name0}, name1_{name1 == nullptr ? "" : name1}
    {
    }

    NoCopyAndSwap(NoCopyAndSwap const& rhs) : name0_{rhs.name0_}, name1_{rhs.name1_} {}

    NoCopyAndSwap(NoCopyAndSwap&& rhs) noexcept
        : name0_{std::exchange(rhs.name0_, nullptr)}, name1_{std::move(rhs.name1_)}
    {
        // move後には、
        // * name0_はnullptr
        // * name1_(nullptrを保持したunique_ptr
        // となる。
    }

    NoCopyAndSwap& operator=(NoCopyAndSwap const& rhs)
    {
        if (this == &rhs) {
            return *this;
        }
    }
}

```

```

// copyコンストラクタのコードクローン
name0_ = rhs.name0_;
name1_ = rhs.name1_; // ここでエクセプションが発生すると*thisが壊れる

return *this;
}

NoCopyAndSwap& operator=(NoCopyAndSwap&& rhs) noexcept
{
    if (this == &rhs) {
        return *this;
    }

    // moveコンストラクタのコードクローン
    name0_ = std::exchange(rhs.name0_, nullptr);
    name1_ = std::string{}; // これがないと、name1_の値がrhs.name1_にスワップされる
    name1_ = std::move(rhs.name1_);

    return *this;
}

char const* GetName0() const noexcept { return name0_; }
std::string const& GetName1() const noexcept { return name1_; }
~NoCopyAndSwap() = default;

private:
    char const* name0_; // 問題やその改善を明示するために、敢えてname0_をchar const*としたが、
                        // 本来ならば、std::stringかstd::string_viewを使うべき
    std::string name1_;
};

```

コード内のコメントで示したように、このコードには以下のような問題がある。

- copy代入演算子には、エクセプション安全性の保証がない。
- 上記4関数は似ているにも関わらず、微妙な違いがあるためコードクローンとなっている。

ここで紹介するCopy-And-Swapはこのような問題を解決するためのイデオムである。

実装例を以下に示す。

```

// @@@ example/design_pattern/copy_and_swap_ut.cpp 6

class CopyAndSwap final {
public:
    explicit CopyAndSwap(char const* name0, char const* name1)
        : name0_{name0 == nullptr ? "" : name0}, name1_{name1 == nullptr ? "" : name1}
    {}

    CopyAndSwap(CopyAndSwap const& rhs) : name0_{rhs.name0_}, name1_{rhs.name1_} {}

    CopyAndSwap(CopyAndSwap&& rhs) noexcept
        : name0_{std::exchange(rhs.name0_, nullptr)}, name1_{std::move(rhs.name1_)}
    {
        // move後には、
        // * name0_はnullptr
        // * name1_は""を保持したstd::string
        // となる。
    }

    CopyAndSwap& operator=(CopyAndSwap const& rhs)
    {
        if (this == &rhs) {
            return *this;
        }

        // copyコンストラクタの使用
        CopyAndSwap tmp{rhs}; // ここでエクセプションが発生しても、tmp以外、壊れない

        Swap(tmp);

        return *this;
    }

    CopyAndSwap& operator=(CopyAndSwap&& rhs) noexcept
    {
        if (this == &rhs) {
            return *this;
        }
    }

```

```

CopyAndSwap tmp{std::move(rhs)}; // moveコンストラクタ

Swap(tmp);

return *this;
}

void Swap(CopyAndSwap& rhs) noexcept
{
    std::swap(name0_, rhs.name0_);
    std::swap(name1_, rhs.name1_);
}

char const* GetName0() const noexcept { return name0_; }
std::string const& GetName1() const noexcept { return name1_; }

~CopyAndSwap() = default;

private:
    char const* name0_; // 問題やその改善を明示するために、敢えてname0_をchar const*としたが、
                        // 本来ならば、std::stringかstd::string_viewを使うべき
    std::string name1_;
};

```

上記CopyAndSwapのcopyコンストラクタ、moveコンストラクタに変更はない。また、CopyAndSwap::Swapに関しててもstd::vector等が持つswapと同様のものである。このイデオムの特徴は、copy代入演算子、move代入演算子が各コンストラクタとSwap関数により実装されている所にある。これによりエクセプション安全性の保証を持つ4関数をコードクローンすることなく実装できる。

演習-Copy-And-Swap

Immutable

クラスに対するimmutable、immutabilityの定義を以下のように定める。

- immutable(不变な)なクラスとは、初期化後、状態の変更ができないクラスを指す。
- immutability(不变性)が高いクラスとは、状態を変更するメンバ関数(非constなメンバ関数)が少ないクラスを指す。

immutabilityが高いほど、そのクラスの使用方法は制限される。これにより、そのクラスやそのクラスを使用しているソースコードの可読性やデバッグ容易性が向上する。また、クラスがimmutableでなくても、そのクラスのオブジェクトをconstハンドル経由でアクセスすることで、immutableとして扱うことができる。

一方で、「Accessor」で紹介したsetterは、クラスのimmutabilityを下げる。いつでも状態が変更できるため、ソースコードの可読性やデバッグ容易性が低下する。また、マルチスレッド環境においてはこのことが競合問題や、それを回避するためのロックがパフォーマンス問題やデッドロックを引き起こしてしまう。

従って、クラスを宣言、定義する場合、immutabilityを出来るだけ高くするべきであり、そのクラスのオブジェクトを使う側は、可能な限りimmutableオブジェクト(constオブジェクト)として扱うべきである。

演習-Immutable

Clone(仮想コンストラクタ)

オブジェクトコピーによるスライシングを回避するためのイデオムである。

下記は、オブジェクトコピーによるスライシングを起こしてしまう例である。

```

// @@@ example/design_pattern/clone_ut.cpp 8

class BaseSlicing {
public:
    ...
    virtual char const* Name() const noexcept { return "BaseSlicing"; }
};

class DerivedSlicing final : public BaseSlicing {
public:
    ...
    virtual char const* Name() const noexcept override { return "DerivedSlicing"; }
};

TEST(Clone, object_slicing)
{

```

```

auto b = BaseSlicing{};
auto d = DerivedSlicing{};

BaseSlicing* b_ptr = &b;
BaseSlicing* b_ptr_d = &d;

ASSERT_STREQ("BaseSlicing", b_ptr->Name());
ASSERT_STREQ("DerivedSlicing", b_ptr_d->Name());

*b_ptr = *b_ptr_d; // コピーしたつもりだがスライシングにより、*b_ptrは、
// DerivedSlicingのインスタンスではなく、BaseSlicingのインスタンス

#if 0
    ASSERT_STREQ("DerivedSlicing", b_ptr->Name());
#else
    ASSERT_STREQ("BaseSlicing", b_ptr->Name()); // "DerivedSlicing"が返るはずだが、
                                                // スライシングにより"BaseSlicing"が返る
#endif
}

```

下記は、上記にcloneイデオムを適用した例である。

```

// @@@ example/design_pattern/clone_ut.cpp 50

// スライシングを起こさないようにコピー演算子の代わりにClone()を実装。
class BaseNoSlicing {
public:
    ...
    virtual char const* Name() const noexcept { return "BaseNoSlicing"; }

    virtual std::unique_ptr<BaseNoSlicing> Clone() { return std::make_unique<BaseNoSlicing>(); }

    BaseNoSlicing(BaseNoSlicing const&) = delete; // copy生成の禁止
    BaseNoSlicing& operator=(BaseNoSlicing const&) = delete; // copy代入の禁止
};

class DerivedNoSlicing final : public BaseNoSlicing {
public:
    ...
    virtual char const* Name() const noexcept override { return "DerivedNoSlicing"; }

    std::unique_ptr<DerivedNoSlicing> CloneOwn() { return std::make_unique<DerivedNoSlicing>(); }

    // DerivedNoSlicingはBaseNoSlicingの派生クラスであるため、
    // std::unique_ptr<DerivedNoSlicing>オブジェクトから
    // std::unique_ptr<BaseNoSlicing>オブジェクトへのmove代入可能
    virtual std::unique_ptr<BaseNoSlicing> Clone() override { return CloneOwn(); }
};

TEST(Clone, object_slicing_avoidance)
{
    auto b = BaseNoSlicing{};
    auto d = DerivedNoSlicing{};

    BaseNoSlicing* b_ptr = &b;
    BaseNoSlicing* b_ptr_d = &d;

    ASSERT_STREQ("BaseNoSlicing", b_ptr->Name());
    ASSERT_STREQ("DerivedNoSlicing", b_ptr_d->Name());

    #if 0
        *b_ptr = *b_ptr_d; // コピー演算子をdeleteしたのでコンパイルエラー
    #else
        auto b_uptr = b_ptr_d->Clone(); // コピー演算子の代わりにClone()を使う。
    #endif

    ASSERT_STREQ("DerivedNoSlicing", b_uptr->Name()); // 意図通り"DerivedNoSlicing"が返る。
}

```

B1::Clone()やそのオーバーライドであるD1::Clone()を使うことで、スライシングを起こすことなくオブジェクトのコピーを行うことができるようになった。

演習-Clone

NVI(non virtual interface)

NVIとは、「virtualなメンバ関数をpublicにしない」という実装上の制約である。

下記のようにクラスBaseが定義されているとする。

```
// @@@ example/design_pattern/nvi_ut.cpp 7

class Base {
public:
    virtual bool DoSomething(int something) const noexcept
    {
        ...
    }

    virtual ~Base() = default;

private:
    ...
};
```

これを使うクラスはBase::DoSomething()に依存する。また、このクラスから派生した下記のクラスDerivedもBase::DoSomething()に依存する。

```
// @@@ example/design_pattern/nvi_ut.cpp 26

class Derived : public Base {
public:
    virtual bool DoSomething(int something) const noexcept override
    {
        ...
    }

private:
    ...
};
```

この条件下ではBase::DoSomething()へ依存が集中し、この関数の修正や機能追加の作業コストが高くなる。このイデオムは、この問題を軽減する。

これを用いた上記2クラスのリファクタリング例を以下に示す。

```
// @@@ example/design_pattern/nvi_ut.cpp 57

class Base {
public:
    bool DoSomething(int something) const noexcept { return do_something(something); }
    virtual ~Base() = default;

private:
    virtual bool do_something(int something) const noexcept
    {
        ...
    }

    ...
};

class Derived : public Base {
private:
    virtual bool do_something(int something) const noexcept override
    {
        ...
    }

    ...
};
```

オーバーライド元の関数とそのオーバーライドのデフォルト引数の値は一致させる必要がある。

それに従わない下記のようなクラスとその派生クラス

```
// @@@ example/design_pattern/nvi_ut.cpp 105

class NotNviBase {
public:
    virtual std::string Name(bool mangled = false) const
    {
        return mangled ? typeid(*this).name() : "NotNviBase";
    }

    virtual ~NotNviBase() = default;
};
```

```

class NotNviDerived : public NotNviBase {
public:
    virtual std::string Name(bool mangled = true) const override // NG デフォルト値が違う
    {
        return mangled ? typeid(*this).name() : "NotNviDerived";
    }
};

```

には下記の単体テストで示したような、メンバ関数の振る舞いがその表層型に依存してしまう問題を持つことになる。

```

// @@@ example/design_pattern/nvi_ut.cpp 129

NotNviDerived const d;
NotNviBase const& d_ref = d;

ASSERT_EQ("NotNviDerived", d.Name(false)); // OK
ASSERT_EQ("13NotNviDerived", d.Name(true)); // OK

ASSERT_EQ("NotNviDerived", d_ref.Name(false)); // OK
ASSERT_EQ("13NotNviDerived", d_ref.Name(true)); // OK

ASSERT_EQ("13NotNviDerived", d.Name()); // mangled == false
ASSERT_EQ("NotNviDerived", d.ref.Name()); // mangled == true

ASSERT_NE(d.Name(), d_ref.Name()); // NG d_refの実態はdであるが、d.Name()と動きが違う

```

この例のように継承階層が浅く、デフォルト引数の数も少ない場合、この値を一致させることは難しくないが、これよりも遙かに複雑な実際のコードではこの一致の維持は困難になる。

下記のようにNVIに従わせることでこのような問題に対処できる。

```

// @@@ example/design_pattern/nvi_ut.cpp 148
class NviBase {
public:
    std::string Name(bool mangled = false) const { return name(mangled); }
    virtual ~NviBase() = default;

private:
    virtual std::string name(bool mangled) const
    {
        return mangled ? typeid(*this).name() : "NviBase";
    }
};

class NviDerived : public NviBase {
private:
    virtual std::string name(bool mangled) const override // OK デフォルト値を持たない
    {
        return mangled ? typeid(*this).name() : "NviDerived";
    }
};

```

下記の単体テストにより、この問題の解消が確認できる。

```

// @@@ example/design_pattern/nvi_ut.cpp 173

NviBase const b;
NviDerived const d;
NviBase const& d_ref = d;

ASSERT_EQ("NviDerived", d.Name(false)); // OK
ASSERT_EQ("10NviDerived", d.Name(true)); // OK

ASSERT_EQ("NviDerived", d_ref.Name(false)); // OK
ASSERT_EQ("10NviDerived", d_ref.Name(true)); // OK

ASSERT_EQ("NviDerived", d.Name()); // mangled == false
ASSERT_EQ("NviDerived", d.ref.Name()); // mangled == false

ASSERT_EQ(d.Name(), d_ref.Name()); // OK

```

なお、メンバ関数のデフォルト引数は、そのクラス外部からのメンバ関数呼び出しを簡潔に記述するための記法であるため、`private`なメンバ関数はデフォルト引数を持つべきではない。

演習-NVI

RAII(scoped guard)

RAIIとは、「Resource Acquisition Is Initialization」の略語であり、リソースの確保と解放をオブジェクトの初期化と破棄処理に結びつけるパターンもしくはイデオムである。特にダイナミックにオブジェクトを生成する場合、RAIIに従わないとメモリリークを防ぐことは困難である。

下記は、関数終了付近でdeleteする素朴なコードである。

```
// @@@ example/design_pattern/raii_ut.cpp 18

// Aは外部の変数をリファレンスcounter_として保持し、
// * コンストラクタ呼び出し時に++counter_
// * デストラクタ呼び出し時に--counter_
// とするため、生成と解放が同じだけ行われれば外部の変数の値は0となる
class A {
public:
    A(uint32_t& counter) noexcept : counter_{++counter} {}
    ~A() { --counter_; }

private:
    uint32_t& counter_;
};

char not_use_RAIIfor_memory(size_t index, uint32_t& object_counter)
{
    auto a = new A{object_counter}; // RAIIfでない例
    auto s = std::string{"hehe"};

    auto ret = s.at(index); // index >= 5でエクセプション発生

    // 何らかの処理

    delete a; // この行以前に関数を抜けるとaはメモリリーク

    return ret;
}
```

このコードは下記の単体テストが示す通り、第1パラメータが5以上の場合、エクセプションが発生しメモリリークしてしまう。

```
// @@@ example/design_pattern/raii_ut.cpp 71

auto object_counter = 0U;

// 第1引数が5なのでエクセプション発生
ASSERT_THROW(not_use_RAIIfor_memory(5, object_counter), std::exception);

// 上記のnot_use_RAIIfor_memoryではエクセプションが発生し、メモリリークする
ASSERT_EQ(1, object_counter);
```

以下は、std::unique_ptrによってRAIIを導入し、この問題に対処した例である。

```
// @@@ example/design_pattern/raii_ut.cpp 83

char use_RAIIfor_memory(size_t index, uint32_t& object_counter)
{
    auto a = std::make_unique<A>(object_counter);
    auto s = std::string{"hehe"};

    auto ret = s.at(index); // index >= 5でエクセプション発生

    // 何らかの処理

    return ret; // aは自動解放される
}
```

下記単体テストで確認できるように、エクセプション発生時にもstd::unique_ptrによる自動解放によりメモリリークは発生しない。

```
// @@@ example/design_pattern/raii_ut.cpp 100

auto object_counter = 0U;

// 第1引数が5なのでエクセプション発生
ASSERT_THROW(use_RAIIfor_memory(5, object_counter), std::exception);

// 上記のuse_RAIIfor_memoryではエクセプションが発生するがメモリリークはしない
ASSERT_EQ(0, object_counter);
```

RAIIのテクニックはメモリ管理のみでなく、ファイルディスクリプタ(open-close、socket-close)等のリソース管理においても有効であるという例を示す。

下記は、生成したソケットを関数終了付近でcloseする素朴なコードである。

```
// @@@ example/design_pattern/raii_ut.cpp 111

// RAIをしない例
// 複数のclose()を書くような関数は、リソースリークを起こしやすい。
void not_use_RAI_for_socket()
{
    auto fd = socket(AF_INET, SOCK_STREAM, 0);

    try {
        // Do something
        ...
    }
    catch (std::exception const& e) { // エクセプションはconstリファレンスで受ける。
        close(fd); // NG RAI未使用
        // Do something to recover
        ...

        return;
    }
    ...
    close(fd); // NG RAI未使用
}
```

エクセプションを扱うために関数の2か所でソケットをcloseしている。この程度であれば大きな問題にはならないだろうが、実際には様々な条件が重なるため、リソースの解放コードは醜悪にならざるを得ない。

このような場合には、下記するようなリソース解放用クラス

```
// @@@ h/scoped_guard.h 4

/// @class ScopedGuard
/// @brief RAIのためのクラス。
///       コンストラクタ引数の関数オブジェクトをデストラクタから呼び出す。
template <typename F>
class ScopedGuard {
public:
    explicit ScopedGuard(F&& f) noexcept : f_{f}
    {
        // f()がill-formedにならず、その戻りがvoidでなければならぬ
        static_assert(std::is_invocable_r_v<void, F>, "F must be callable and return void");
    }

    ~ScopedGuard() { f_(); }
    ScopedGuard(ScopedGuard const&) = delete; // copyは禁止
    ScopedGuard& operator=(ScopedGuard const&) = delete; // copyは禁止

private:
    F f_;
};
```

を使用し、下記のようにすることで安全なコードをすっきりと書くことができる。

```
// @@@ example/design_pattern/raii_ut.cpp 138

// RAIをScopedGuardで行った例。
// close()が自動実行されるためにリソース解放を忘れない。
void use_RAI_for_socket()
{
    auto fd = socket(AF_INET, SOCK_STREAM, 0);
    auto guard = ScopedGuard{[fd] { close(fd); }}; // 関数終了時に自動実行

    try {
        // Do something
    }
    catch (...) {
        // Do something to recover

        return;
    }
    ...
}
```

クリティカルセクションの保護をlock/unlockで行うstd::mutex等を使う場合にも、 std::lock_guard<>によってunlockを行うことで、同様の効果が得られる。

演習-RAIIの効果

演習-RAII

Future

Futureとは、並行処理のためのデザインパターンであり、別スレッドに何らかの処理をさせる際、その結果の取得を、必要になるまで後回しにする手法である。

C++11では、std::future, std::promise, std::asyncによって実現できる。

まずは、C++03以前のスタイルから示す。

```
// @@@ example/design_pattern/future_ut.cpp 11

int do_something(std::string_view str0, std::string_view str1) noexcept
{
    ...
    return ret0 + ret1;
}

TEST(Future, old_style)
{
    auto str0 = std::string{};
    auto th0  = std::thread{[&str0]() noexcept { str0 = do_heavy_algorithm("thread 0"); }};
    auto str1 = std::string{};
    auto th1  = std::thread{[&str1]() noexcept { str1 = do_heavy_algorithm("thread 1"); }};

    // このスレッドで行うべき何らかの処理
    //

    th0.join();
    th1.join();

    ASSERT_EQ("THREAD 0", str0);
    ASSERT_EQ("THREAD 1", str1);

    ASSERT_EQ(16, do_something(str0, str1));
}
```

上記は、

1. 時間かかる処理を並行して行うために、スレッドを二つ作る。
2. それぞれの完了をthread::join()で待ち合わせる。
3. その結果を参照キャプチャによって受け取る。
4. その2つの結果を別の関数に渡す。

という処理を行っている。

この程度の単純なコードでは特に問題にはならないが、目的外の処理が多いことがわかるだろう。

次にFutureパターンによって上記をリファクタリングした例を示す。

```
// @@@ example/design_pattern/future_ut.cpp 45

TEST(Future, new_style)
{
    std::future<std::string> result0
        = std::async(std::launch::async, []() noexcept { return do_heavy_algorithm("thread 0"); });

    std::future<std::string> result1
        = std::async(std::launch::async, []() noexcept { return do_heavy_algorithm("thread 1"); });

    // future::get()は処理の待ち合わせと値の取り出しを行う。
    auto str0 = result0.get();
    auto str1 = result1.get();
    ASSERT_EQ(16, do_something(str0, str1));

    ASSERT_EQ("THREAD 0", str0);
}
```

```
    ASSERT_EQ("THREAD 1", str1);
}
```

リファクタリングした例では、時間のかかる処理をstd::future型のオブジェクトにし、その結果を必要とする関数に渡すことができるため、目的をよりダイレクトに表すことができる。

なお、

```
std::async(関数オブジェクト)
```

という形式を使った場合、関数オブジェクトは、

```
std::launch::async | std::launch::deferred
```

が指定されたとして実行される。この場合、

```
std::launch::deferred
```

の効果により、関数オブジェクトは、並行に実行されるとは限らない（この仕様はランタイム系に依存しており、std::future::get()のコンテキストで実行されることもあり得る）。従って、並行実行が必要な場合、上記例のように

```
std::launch::async
```

のみを明示的に指定するべきである。

演習-Future

DI(dependency injection)

メンバ関数内でクラスDependedのオブジェクトを直接、生成する（もしくはSingletonオブジェクトや静的オブジェクト（std::coutやstd::cin等）に直接アクセスする）クラスNotDIがあるとする。この場合、クラスNotDIはクラスDependedのインスタンスに依存してしまう。このような依存関係はクラスNotDIの可用性とテスト容易性を下げる。これは、「仮にクラスDependedがデータベースをラップするクラスだった場合、クラスNotDIの単体テストにデータベースが必要になる」ことからも容易に理解できる。

```
// @@@ example/design_pattern/di_ut.cpp 8

/// @class Depended
/// @brief NotDIや、DIから依存されるクラス
class Depended {
    ...
};

/// @class NotDI
/// @brief NotDIを使わない例。そのため、NotDIは、Dependedのインスタンスに依存している。
class NotDI {
public:
    NotDI() : not_di_depended_{std::make_unique<Depended>()} {}

    void DoSomething() { not_di_depended_->DoSomething(); }

private:
    std::unique_ptr<Depended> not_di_depended_;
};
```

下記は上記NotDIにDIパターンを適用した例である。この場合、クラスDIは、クラスDependedの型にのみ依存する。

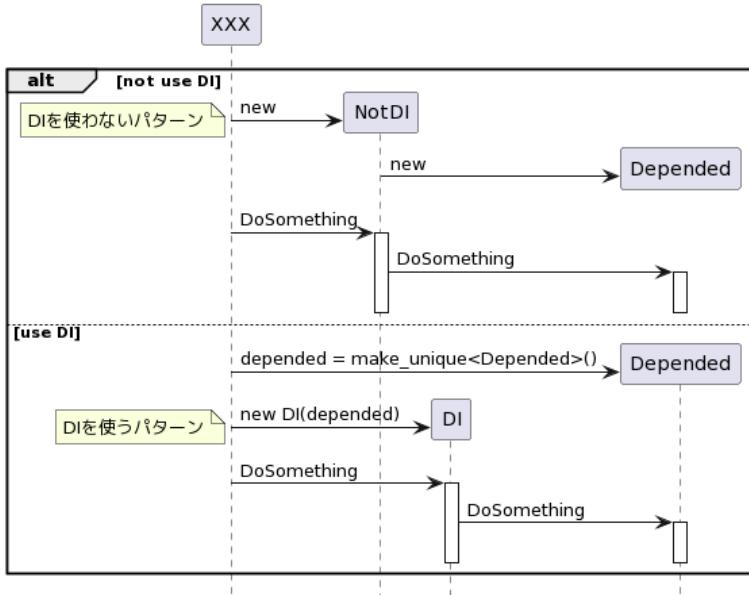
```
// @@@ example/design_pattern/di_ut.cpp 39

/// @class DI
/// @brief DIを使う例。そのため、DIは、Dependedの型に依存している。
class DI {
public:
    explicit DI(std::unique_ptr<Depended>&& di_depended) noexcept
        : di_depended_{std::move(di_depended)} {}

    void DoSomething() { di_depended_->DoSomething(); }

private:
    std::unique_ptr<Depended> di_depended_;
};
```

下記は、クラスNotDIとクラスDIがそれぞれのDoSomething()を呼び出すまでのシーケンス図である。



このパターンの効果により、DIオブジェクトにはDependedかその派生クラスのオブジェクトを渡すことができるようになった。これによりクラスDIは拡張性に対して柔軟になっただけでなく、テスト容易性も向上した。

次に示すのは、このパターンを使用して問題のある単体テストを修正した例である。

まずは、問題があるクラスとその単体テストを下記する。

```
// in device_io.h

class DeviceIO {
public:
    uint8_t read()
    {
        // ハードウェアに依存した何らかの処理
    }

    void write(uint8_t a)
    {
        // ハードウェアに依存した何らかの処理
    }

private:
    // 何らかの宣言
};

#ifndef UNIT_TEST      // 単体テストビルトでは定義されるマクロ
class DeviceIO_Mock { // 単体テスト用のモック
public:
    uint8_t read()
    {
        // ハードウェアに依存しない何らかの処理
    }

    void write(uint8_t a)
    {
        // ハードウェアに依存しない何らかの処理
    }

private:
    // 何らかの宣言
};
#endif
```

```
// in widget.h

#include "device_io.h"

class Widget {
public:
    void DoSomething()
    {
        // io_を使った何らかの処理
    }
}
```

```

    uint8_t GetResp()
    {
        // io_を使った何らかの処理
    }

private:
#ifndef UNIT_TEST
    DeviceIO_Mock io_;
#else
    DeviceIO io_;
#endif
};

// in widget_ut.cpp

// UNIT_TESTマクロが定義されたWidgetの単体テスト
Widget w;

w.DoSomething();
ASSERT_EQ(0, w.GetResp());

```

当然であるが、この単体テストは、UNIT_TESTマクロを定義している場合のWidgetの評価であり、UNIT_TESTを定義しない実際のコードの評価にはならない。

以下では、DIを用い、この問題を回避する。

```

// in device_io.h

class DeviceIO {
public:
    virtual uint8_t read() // モックでオーバーライドするためvirtual
    {
        // ハードウェアに依存した何らかの処理
    }

    virtual void write(uint8_t a) // モックでオーバーライドするためvirtual
    {
        // ハードウェアに依存した何らかの処理
    }
    virtual ~DeviceIO() = default;

private:
    // 何らかの宣言
};

// in widget.h

class Widget {
public:
    Widget(std::unique_ptr<DeviceIO> io = std::make_unique<DeviceIO>()) : io_{std::move(io)} {}

    void DoSomething()
    {
        // io_を使った何らかの処理
    }

    uint8_t GetResp()
    {
        // io_を使った何らかの処理
    }

private:
    std::unique_ptr<DeviceIO> io_;
};

// in widget_ut.cpp

class DeviceIO_Mock : public DeviceIO { // 単体テスト用のモック
public:
    uint8_t read() override
    {
        // ハードウェアに依存しない何らかの処理
    }

    void write(uint8_t a) override
    {
        // ハードウェアに依存しない何らかの処理
    }
}

```

```

private:
    // 何らかの宣言
};

// 上記DeviceIO_Mockと同様に、in widget_ut.cpp

Widget w{std::unique_ptr<DeviceIO>(new DeviceIO_Mock)}; // モックのインジェクション

// Widgetの単体テスト
w.DoSomething();
ASSERT_EQ(1, w.GetResp());

```

この例では、単体テストのためだけに仮想関数を導入しているため、多少やりすぎの感がある。そのような場合、下記のようにテンプレートを用いればよい。

```

// in device_io.h

class DeviceIO {
public:
    uint8_t read() // Widgetがテンプレートであるため非virtualで良い
    {
        // ハードウェアに依存した何らかの処理
    }

    void write(uint8_t a) // Widgetがテンプレートであるため非virtualで良い
    {
        // ハードウェアに依存した何らかの処理
    }
    virtual ~DeviceIO() = default;

private:
    // 何らかの宣言
};

```

```

// in widget.h

template <class T = DeviceIO>
class Widget {
public:
    void DoSomething()
    {
        // io_を使った何らかの処理
    }

    uint8_t GetResp()
    {
        // io_を使った何らかの処理
    }

private:
    T io_;
};

```

```

// in widget_ut.cpp

class DeviceIO_Mock { // 単体テスト用のモック
public:
    uint8_t read() // Widgetがテンプレートであるため非virtualで良い
    {
        // ハードウェアに依存しない何らかの処理
    }

    void write(uint8_t a) // Widgetがテンプレートであるため非virtualで良い
    {
        // ハードウェアに依存しない何らかの処理
    }

private:
    // 何らかの宣言
};

```

```

// 上記DeviceIO_Mockと同様に、in widget_ut.cpp

Widget<DeviceIO_Mock> w;

// Widget<>の単体テスト
w.DoSomething();
ASSERT_EQ(2, w.GetResp());

```

以上からわかるように、ここで紹介したDIは単体テストを容易にするクラス設計のために非常に有用なパターンである。

Singleton

このパターンにより、特定のクラスのインスタンスをシステム全体で唯一にすることができる。これにより、グローバルオブジェクトを規律正しく使用しやすくなる。

以下は、Singletonの実装例である。

```
// @@@ example/design_pattern/singleton_ut.cpp 7

class Singleton final {
public:
    static Singleton& Inst();
    static Singleton const& InstConst() noexcept // constインスタンスを返す
    {
        return Inst();
    }
    ...
private:
    Singleton() noexcept {} // コンストラクタをprivateにすることで、
                            // Inst()以外ではこのオブジェクトを生成できない。
    ...
};

Singleton& Singleton::Inst()
{
    static Singleton inst; // instの初期化が同時に行われることはない。

    return inst;
}

TEST(Singleton, how_to_use)
{
    auto& inst      = Singleton::Inst();
    auto const& inst_const = Singleton::InstConst();

    ASSERT_EQ(0, inst.GetXxx());
    ASSERT_EQ(0, inst_const.GetXxx());
#ifndef NDEBUG
    inst_const.SetXxx(10); // inst_constはconstオブジェクトなのでコンパイルエラー
#else
    inst.SetXxx(10);
#endif
    ASSERT_EQ(10, inst.GetXxx());
    ASSERT_EQ(10, inst_const.GetXxx());

    inst.SetXxx(0);
    ASSERT_EQ(0, inst.GetXxx());
    ASSERT_EQ(0, inst_const.GetXxx());
}
} // namespace
```

このパターンを使用する場合、以下に注意する。

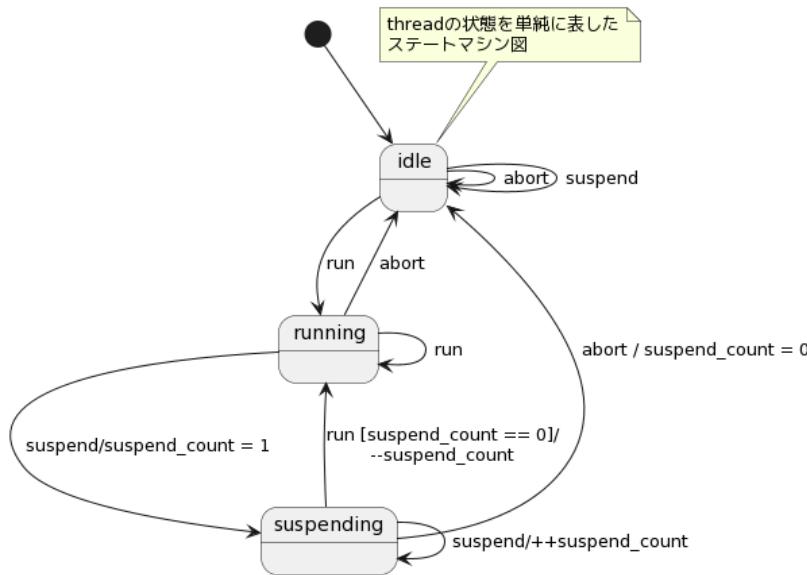
- Singletonはデザインパターンの中でも、特にパターン猿病を発生しやすい。 Singletonは「ほとんどグローバル変数である」ことを理解した上で、控えめに使用する。
- Singletonを定義する場合、以下の二つを定義する。
 - インスタンスを返すstaticメンバ関数Inst()
 - constインスタンスを返すstaticメンバ関数InstConst()
- InstConst()は、Inst()が返すオブジェクトと同じオブジェクトを返すようにする。
- Singletonには、可能な限りInstConst()経由でアクセスする。

Singletonオブジェクトの初期化(最初のコンストラクタ呼び出し)は、C++03以前はスレッドセーフでなかったため、「Double Checked Lockingを使って競合を避ける」か、「他のスレッドを起動する前にメインスレッドから各SingletonのInstConst()を呼び出す」ことが必要であった。 C++11から上記例のようなSingletonオブジェクトのコンストラクタ呼び出しがスレッドセーフとなったため、このような黒魔術が不要になった。

なお、Inst()のような関数を複数定義する場合、そのパターンはNamed Constructor (「[Named Constructor](#)」参照)と呼ばれる。

State

Stateは、オブジェクトの状態と、それに伴う振る舞いを分離して記述するためのパターンである。これにより状態の追加、削減、変更に伴う修正範囲が限定される（「[オープン・クローズドの原則\(OCP\)](#)」参照）。またオブジェクトのインターフェース変更（パブリックメンバ関数の変更）に関しても、修正箇所が明確になる。



上記ステートマシン図の「オールドスタイルによる実装」と、「stateパターンによる実装」、それぞれを例示する。

まずは、下記にオールドスタイルな実装例を示す。この実装では、状態を静的なenum変数`thread_old_style_state`で管理するため、`ThreadOldStyleStateStr()`、`ThreadOldStyleRun()`、`ThreadOldStyleAbort()`、`ThreadOldStyleSuspend()`には、`thread_old_style_state`に対する同型のswitch文が入ることになる（下記例では一部省略）。これは醜悪で、バグを起こしやすい構造である。ただし、要求される状態遷移がこの例程度であり、状態ごとに決められた振る舞いの数が少なければ、この構造でも問題ないともいえる。

```
// @@@ example/design_pattern/state_machine_old.h 4

extern std::string_view ThreadOldStyleStateStr() noexcept;
extern void ThreadOldStyleRun();
extern void ThreadOldStyleAbort();
extern void ThreadOldStyleSuspend();

// @@@ example/design_pattern/state_machine_old.cpp 6

namespace {
enum class ThreadOldStyleState {
    Idle,
    Running,
    Suspending,
};

ThreadOldStyleState thread_old_style_state;
...

} // namespace

std::string_view ThreadOldStyleStateStr() noexcept
{
    switch (thread_old_style_state) { // このswitch文と同型switch文が何度も記述される
        case ThreadOldStyleState::Idle:
            return "Idle";
        case ThreadOldStyleState::Running:
            return "Running";
        case ThreadOldStyleState::Suspending:
            return "Suspending";
        default:
            assert(false);
            return "";
    }
}

void ThreadOldStyleRun()
{
    switch (thread_old_style_state) {
        case ThreadOldStyleState::Idle:
```

```

case ThreadOldStyleState::Running:
    thread_old_style_state = ThreadOldStyleState::Running;
    break;
case ThreadOldStyleState::SUSPENDING:
    --thread_old_style_suspend_count;
    if (thread_old_style_suspend_count == 0) {
        thread_old_style_state = ThreadOldStyleState::Running;
    }
    break;
default:
    assert(false);
}
}

void ThreadOldStyleAbort()
{
    ...
}

void ThreadOldStyleSuspend()
{
    ...
}

```

```

// @@@ example/design_pattern/state_machine_ut.cpp 15

// ステートのテスト。仕様書よりも単体テストでその仕様や使用法を記述したほうが正確に理解できる。
TEST(StateMachine, old_style)
{
    ASSERT_EQ("Idle", ThreadOldStyleStateStr());

    ThreadOldStyleAbort();
    ASSERT_EQ("Idle", ThreadOldStyleStateStr());

    ThreadOldStyleRun();
    ASSERT_EQ("Running", ThreadOldStyleStateStr());

    ThreadOldStyleRun();
    ASSERT_EQ("Running", ThreadOldStyleStateStr());

    ThreadOldStyleSuspend();
    ASSERT_EQ("SUSPENDING", ThreadOldStyleStateStr()); // suspend_count_ == 1

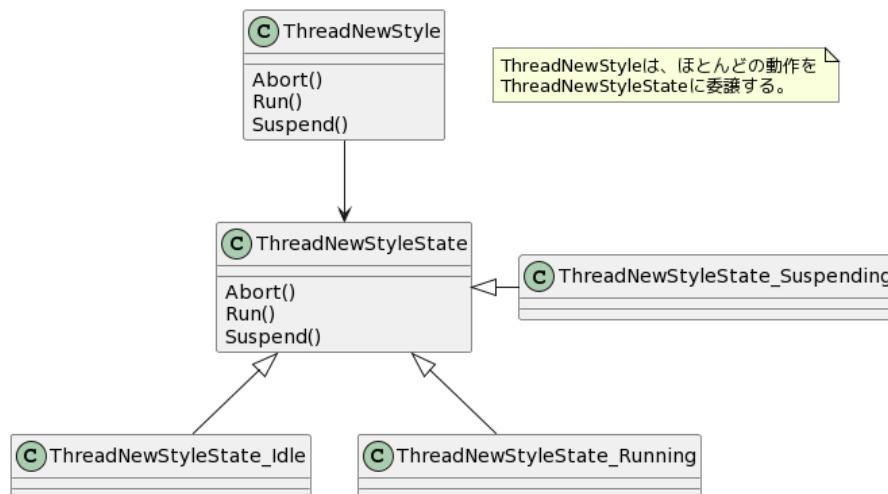
    ThreadOldStyleSuspend();
    ASSERT_EQ("SUSPENDING", ThreadOldStyleStateStr()); // suspend_count_ == 2

    ThreadOldStyleRun();
    ASSERT_EQ("SUSPENDING", ThreadOldStyleStateStr()); // suspend_count_ == 1

    ...
}

```

下記は、上記例へstateパターンを適用した例である。まずは、stateパターンを形成するクラスの関係をクラス図で示す。



次に上記クラス図の実装例を示す。

```
// @@@ example/design_pattern/state_machine_new.h 6
```

```

/// @class ThreadNewStyleState
/// @brief ThreadNewStyleのステートを表す基底クラス
class ThreadNewStyleState {
public:
    ThreadNewStyleState() = default;
    virtual ~ThreadNewStyleState() = default;

    std::unique_ptr<ThreadNewStyleState> Abort() // NVI
    {
        return abort_thread();
    }

    std::unique_ptr<ThreadNewStyleState> Run() // NVI
    {
        return run_thread();
    }

    std::unique_ptr<ThreadNewStyleState> Suspend() // NVI
    {
        return suspend_thread();
    }

    std::string_view GetStateStr() const noexcept { return get_state_str(); }

private:
    virtual std::unique_ptr<ThreadNewStyleState> abort_thread()
    {
        return {}; // デフォルトでは何もしない。
    }

    virtual std::unique_ptr<ThreadNewStyleState> run_thread()
    {
        return {}; // デフォルトでは何もしない。
    }

    virtual std::unique_ptr<ThreadNewStyleState> suspend_thread()
    {
        return {}; // デフォルトでは何もしない。
    }

    virtual std::string_view get_state_str() const noexcept = 0;
};

```

```

// @@@ example/design_pattern/state_machine_new.h 52

class ThreadNewStyle final {
public:
    ThreadNewStyle();

    void Abort() { change_state(state_->Abort()); }

    void Run() { change_state(state_->Run()); }

    void Suspend() { change_state(state_->Suspend()); }

    std::string_view GetStateStr() const noexcept { return state_->GetStateStr(); }

private:
    std::unique_ptr<ThreadNewStyleState> state_;

    void change_state(std::unique_ptr<ThreadNewStyleState>&& new_state) noexcept
    {
        if (new_state) {
            state_ = std::move(new_state);
        }
    }
};


```

```

// @@@ example/design_pattern/state_machine_new.cpp 10

class ThreadNewStyleState_Idle final : public ThreadNewStyleState {
    ...
};

class ThreadNewStyleState_Running final : public ThreadNewStyleState {
    ...
};

class ThreadNewStyleState_Suspending final : public ThreadNewStyleState {
public:
    ...
};


```

```

private:
    virtual std::unique_ptr<ThreadNewStyleState> abort_thread() override
    {
        // do something to abort
        ...

        return std::make_unique<ThreadNewStyleState_Idle>();
    }

    virtual std::unique_ptr<ThreadNewStyleState> run_thread() override
    {
        --suspend_count_;

        if (suspend_count_ == 0) {
            // do something to resume
            ...
            return std::make_unique<ThreadNewStyleState_Running>();
        }
        else {
            return {};
        }
    }

    virtual std::unique_ptr<ThreadNewStyleState> suspend_thread() override
    {
        ++suspend_count_;

        return {};
    }
    ...
};

```

```

// @@@ example/design_pattern/state_machine_ut.cpp 57

TEST(StateMachine, new_style)
{
    auto tns = ThreadNewStyle{};

    ASSERT_EQ("Idle", tns.GetStateStr());

    tns.Abort();
    ASSERT_EQ("Idle", tns.GetStateStr());

    tns.Run();
    ASSERT_EQ("Running", tns.GetStateStr());

    tns.Run();
    ASSERT_EQ("Running", tns.GetStateStr());

    tns.Suspend();
    ASSERT_EQ("Suspending", tns.GetStateStr()); // suspend_count_ == 1

    tns.Suspend();
    ASSERT_EQ("Suspending", tns.GetStateStr()); // suspend_count_ == 2

    tns.Run();
    ASSERT_EQ("Suspending", tns.GetStateStr()); // suspend_count_ == 1

    ...
}

```

オールドスタイルな構造に比べると一見複雑に見えるが同型のswitch構造がないため、状態の増減や振る舞いの変更等への対応が容易である。一方で、前述したとおり、この例程度の要求であれば、シンプルさという意味においてオールドスタイルのソースコードの方が優れているともいえる。従って、オールドスタイルとstateパターンの選択は、その要求の複雑さと安定度によって決定されるべきものである。

なお、C++でのstateパターンの実装には、下記に示すようなメンバ関数を使う方法もある。多くのクラスを作る必要はないが、各状態での状態管理変数を別の状態のものと分けて管理することができないため、複雑な状態管理が必要な場合には使えないが、単純な状態管理で十分な場合には便利なパターンである。

```

// @@@ example/design_pattern/state_machine_new.h 77

class ThreadNewStyle2 final {
public:
    ThreadNewStyle2() noexcept {}

    void Abort() { (this->*abort_)(); }
    void Run() { (this->*run_())(); }
    void Suspend() { (this->*suspend_())(); }

    std::string_view GetStateStr() const noexcept { return state_str_; }
}

```

```

private:
    void (ThreadNewStyle2::*abort_())() = &ThreadNewStyle2::abort_idle;
    void (ThreadNewStyle2::*run_())() = &ThreadNewStyle2::run_idle;
    void (ThreadNewStyle2::*suspend_())() = &ThreadNewStyle2::suspend_idle;
    std::string_view state_str_{state_str_idle_};

    void abort_idle() {} // do nothing
    void run_idle();
    void suspend_idle() {} // do nothing
    static inline std::string_view const state_str_idle_{"Idle"};

    void abort_running();
    void run_running() {} // do nothing
    void suspend_running();
    static inline std::string_view const state_str_running_{"Running"};

    void abort_suspending();
    void run_suspending();
    void suspend_suspending() {} // do nothing
    static inline std::string_view const state_str_suspending_{"Suspending"};
};

// @@@ example/design_pattern/state_machine_new.cpp 106

```

```

void ThreadNewStyle2::run_idle()
{
    // スレッドの始動処理
    ...

    // ステートの切り替え
    run_ = &ThreadNewStyle2::run_running;
    suspend_ = &ThreadNewStyle2::suspend_running;
    state_str_ = state_str_running_;
}

void ThreadNewStyle2::abort_running()
{
    // スレッドのアボート処理
    ...

    // ステートの切り替え
    run_ = &ThreadNewStyle2::run_idle;
    suspend_ = &ThreadNewStyle2::suspend_running;
    state_str_ = state_str_idle_;
}

void ThreadNewStyle2::suspend_running()
{
    // スレッドのサスPEND処理
    ...

    // ステートの切り替え
    run_ = &ThreadNewStyle2::run_suspending;
    suspend_ = &ThreadNewStyle2::suspend_running;
    state_str_ = state_str_suspending_;
}

void ThreadNewStyle2::run_suspending()
{
    // スレッドのリジューム処理
    ...

    // ステートの切り替え
    run_ = &ThreadNewStyle2::run_running;
    suspend_ = &ThreadNewStyle2::suspend_running;
    state_str_ = state_str_running_;
}

```

```

// @@@ example/design_pattern/state_machine_ut.cpp 95
TEST(StateMachine, new_style2)
{
    auto tns = ThreadNewStyle2{};

    ASSERT_EQ("Idle", tns.GetStateStr());

    tns.Run();
    ASSERT_EQ("Running", tns.GetStateStr());

    tns.Suspend();
}

```

```

    ASSERT_EQ("Suspending", tns.GetStateStr());
    tns.Suspend();
    ASSERT_EQ("Suspending", tns.GetStateStr());
}

```

演習-State

Null Object

オブジェクトへのポインタを受け取った関数が「そのポインタがnullptrでない場合、そのポインタが指すオブジェクトに何かをさせる」というような典型的な条件文を削減するためのパターンである。

```

// @@@ example/design_pattern/null_object_ut.cpp 7

class A {
public:
    ...
    bool Action() noexcept
    {
        // do something
        ...
        return result;
    }
    ...
};

bool ActionOldStyle(A* a) noexcept
{
    if (a != nullptr) { // ←このif文を消すためのパターン。
        return a->Action();
    }
    else {
        return false;
    }
}

```

上記例にNull Objectパターンを適用した例を下記する。

```

// @@@ example/design_pattern/null_object_ut.cpp 41

class A {
public:
    ...
    bool Action() noexcept { return action(); }

private:
    virtual bool action() noexcept
    {
        // do something
        ...
        return result;
    }
    ...
};

class ANull final : public A {
    ...
private:
    virtual bool action() noexcept override { return false; }
};

bool ActionNewStyle(A& a) noexcept
{
    return a.Action(); // ←Null Objectによりif文が消えた。
}

```

この単純な例では、逆にソースコードが複雑化したように見えるが、

```
if(a != nullptr)
```

を頻繁に使うような関数、クラスではソースコードの単純化やnullptrチェック漏れの防止に非常に有効である。

演習-Null Object

Templateメソッド

Templateメソッドは、雛形の形式(書式等)を定めるメンバ関数(templateメソッド)と、それを埋めるための振る舞いやデータを定めるメンバ関数を分離するときに用いるパターンである。

以下に実装例を示す。

```
// @@@ example/design_pattern/template_method.h 6

/// @class XxxData
/// @brief 何かのデータを入れる箱
struct XxxData {
    int a;
    int b;
    int c;
};

/// @class XxxDataFormatterIF
/// @brief data_storer_if.cppに定義すべきだが、サンプルであるため便宜上同じファイルで定義する
/// データフォーマットを行なうクラスのインターフェースクラス
class XxxDataFormatterIF {
public:
    explicit XxxDataFormatterIF(std::string_view formatter_name) noexcept
        : formatter_name_(formatter_name)
    {
    }
    virtual ~XxxDataFormatterIF() = default;

    std::string ToString(XxxData const& xxx_data) const
    {
        return header() + body(xxx_data) + footer();
    }

    std::string ToString(std::vector<XxxData> const& xxx_datas) const
    {
        std::string ret{header()};
        for (auto const& xxx_data : xxx_datas) {
            ret += body(xxx_data);
        }
        return ret + footer();
    }
    ...
private:
    virtual std::string const& header() const = 0;
    virtual std::string const& footer() const = 0;
    virtual std::string body(XxxData const& xxx_data) const = 0;
    ...
};
```

上記XxxDataFormatterIFでは、以下のようなメンバ関数を宣言、定義している。

メンバ関数		振る舞い
header()	private pure-virtual	ヘッダをstd::stringオブジェクトとして生成
footer()	private pure-virtual	フッタをstd::stringオブジェクトとして生成
body()	private pure-virtual	XxxDataからボディをstd::stringオブジェクトとして生成
ToString()	public normal	header(),body(),footer()の出力を組み合わせた全体像を生成

この構造により、XxxDataFormatterIFは、

- ・ 全体の書式を定義している。
- ・ 各行の生成をXxxDataFormatterIFから派生した具象クラスに委譲している。

下記XxxDataFormatterXml、XxxDataFormatterCsv、XxxDataFormatterTableでは、header()、body()、footer()をオーバーライドすることで、それぞれの機能を実現している。

```
// @@@ example/design_pattern/template_method.cpp 8

/// @class XxxDataFormatterXml
/// @brief XxxDataをXmlに変換
class XxxDataFormatterXml final : public XxxDataFormatterIF {
```

```

...
private:
    virtual std::string const& header() const noexcept final { return header_; }
    virtual std::string const& footer() const noexcept final { return footer_; }
    virtual std::string body(XxxData const& xxx_data) const override
    {
        auto content = std::string("<Item>\n");

        content += "    <XxxData a=\"" + std::to_string(xxx_data.a) + "\">\n";
        content += "    <XxxData b=\"" + std::to_string(xxx_data.b) + "\">\n";
        content += "    <XxxData c=\"" + std::to_string(xxx_data.c) + "\">\n";

        return content + "</Item>\n";
    }

    static inline std::string const header_{
        "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n<XxxDataFormatterXml>\n";
    static inline std::string const footer_{"</XxxDataFormatterXml>\n";
};

/// @class XxxDataFormatterCsv
/// @brief XxxDataをCsvに変換
class XxxDataFormatterCsv final : public XxxDataFormatterIF {
    ...
private:
    virtual std::string const& header() const noexcept final { return header_; }
    virtual std::string const& footer() const noexcept final { return footer_; }
    virtual std::string body(XxxData const& xxx_data) const override
    {
        return std::string{std::to_string(xxx_data.a) + ", " + std::to_string(xxx_data.b) + ", "
                           + std::to_string(xxx_data.b) + "\n"};
    }

    static inline std::string const header_("a, b, c\n");
    static inline std::string const footer_{};
};

/// @class XxxDataFormatterTable
/// @brief XxxDataをTableに変換
class XxxDataFormatterTable final : public XxxDataFormatterIF {
    ...
private:
    virtual std::string const& header() const noexcept final { return header_; }
    virtual std::string const& footer() const noexcept final { return footer_; }
    virtual std::string body(XxxData const& xxx_data) const override
    {
        auto a = std::string{std::string("| ") + std::to_string(xxx_data.a)};
        auto b = std::string{std::string("| ") + std::to_string(xxx_data.b)};
        auto c = std::string{std::string("| ") + std::to_string(xxx_data.c)};

        a += std::string(colomun_ - a.size() + 1, ' ');
        b += std::string(colomun_ - b.size() + 1, ' ');
        c += std::string(colomun_ - c.size() + 1, ' ');

        return a + b + c + "| \n" + border_;
    }
    ...
};

```

以下の単体テストで、これらのクラスの振る舞いを示す。

```

// @@@ example/design_pattern/template_method_ut.cpp 6

TEST(TemplateMethod, xml)
{
    auto xml = XxxDataFormatterXml{};

    {
        auto const xd      = XxxData{1, 100, 10};
        auto const expect = std::string_view{
            "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n"
            "<XxxDataFormatterXml>\n"
            "<Item>\n"
            "    <XxxData a=\"1\">\n"
            "    <XxxData b=\"100\">\n"
            "    <XxxData c=\"10\">\n"
            "</Item>\n"
            "</XxxDataFormatterXml>\n";
        auto const actual = xml.ToString(xd);

```

```

        ASSERT_EQ(expect, actual);
    }

    {
        auto const xds    = std::vector<XxxData>{{1, 100, 10}, {2, 200, 20}};
        auto const expect = std::string_view{
            "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n"
            "<XxxDataFormatterXml>\n"
            "  <Item>\n"
            "    <XxxData a=\"1\">\n"
            "    <XxxData b=\"100\">\n"
            "    <XxxData c=\"10\">\n"
            "  </Item>\n"
            "  <Item>\n"
            "    <XxxData a=\"2\">\n"
            "    <XxxData b=\"200\">\n"
            "    <XxxData c=\"20\">\n"
            "  </Item>\n"
            "</XxxDataFormatterXml>\n"};
        auto const actual = xml.ToString(xds);

        ASSERT_EQ(expect, actual);
    }
}

TEST(TemplateMethod, csv)
{
    auto csv = XxxDataFormatterCsv{};

    {
        auto const xd    = XxxData{1, 100, 10};
        auto const expect = std::string_view{
            "a, b, c\n"
            "1, 100, 100\n"};
        auto const actual = csv.ToString(xd);

        ASSERT_EQ(expect, actual);
    }
    {
        auto const xds    = std::vector<XxxData>{{1, 100, 10}, {2, 200, 20}};
        auto const expect = std::string_view{
            "a, b, c\n"
            "1, 100, 100\n"
            "2, 200, 200\n"};
        auto const actual = csv.ToString(xds);

        ASSERT_EQ(expect, actual);
    }
}

TEST(TemplateMethod, table)
{
    auto table = XxxDataFormatterTable{};

    ...
}

```

上記で示した実装例は、public継承による動的ポリモーフィズムを使用したため、XxxDataFormatterXml、XxxDataFormatterCsv、XxxDataFormatterTableのインスタンスやそのポインタは、XxxDataFormatterIFのリファレンスやポインタとして表現できる。この性質は、FactoryやNamed Constructorの実装には不可欠であるが、逆にこのようなポリモーフィズムが不要な場合、この柔軟性も不要である。

そういう場合は、private継承を用いるか、テンプレートを用いた静的ポリモーフィズムを用いることでこの柔軟性を排除できる。

下記のコードはそのような実装例である。

```

// @@@ example/design_pattern/template_method_ut.cpp 111

template <typename T> // Tは下記のXxxDataFormatterXmlのようなクラス
class XxxDataFormatter : private T {
public:
    std::string ToString(XxxData const& xxx_data) const
    {
        return T::Header() + T::Body(xxx_data) + T::Footer();
    }

    std::string ToString(std::vector<XxxData> const& xxx_datas) const
    {
        auto ret = std::string{T::Header()};
        for (auto const& xxx_data : xxx_datas) {

```

```

        ret += T::Body(xxx_data);
    }

    return ret + T::Footer();
};

class XxxDataFormatterXml_Impl {
public:
    std::string const& Header() const noexcept { return header_; }
    std::string const& Footer() const noexcept { return footer_; }
    std::string Body(XxxData const& xxx_data) const
    {
        auto content = std::string("<Item>\n");

        content += "    <XxxData a=\"" + std::to_string(xxx_data.a) + "\">\n";
        content += "    <XxxData b=\"" + std::to_string(xxx_data.b) + "\">\n";
        content += "    <XxxData c=\"" + std::to_string(xxx_data.c) + "\">\n";

        return content + "</Item>\n";
    }

private:
    inline static std::string const header_{
        "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n<XxxDataFormatterXml>\n";
    }
    inline static std::string const footer_{"</XxxDataFormatterXml>\n"};
};

using XxxDataFormatterXml = XxxDataFormatter<XxxDataFormatterXml_Impl>;

```

上記の単体テストは下記のようになる。

```

// @@@ example/design_pattern/template_method_ut.cpp 159

auto xml = XxxDataFormatterXml{};

{
    auto const xd      = XxxData{1, 100, 10};
    auto const expect = std::string{
        "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n"
        "<XxxDataFormatterXml>\n"
        "<Item>\n"
        "    <XxxData a=\"1\"\n"
        "    <XxxData b=\"100\"\n"
        "    <XxxData c=\"10\"\n"
        "</Item>\n"
        "</XxxDataFormatterXml>\n";
    };
    auto const actual = xml.ToString(xd);

    ASSERT_EQ(expect, actual);
}

{
    auto const xds     = std::vector<XxxData>{{1, 100, 10}, {2, 200, 20}};
    auto const expect = std::string{
        "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n"
        "<XxxDataFormatterXml>\n"
        "<Item>\n"
        "    <XxxData a=\"1\"\n"
        "    <XxxData b=\"100\"\n"
        "    <XxxData c=\"10\"\n"
        "</Item>\n"
        "<Item>\n"
        "    <XxxData a=\"2\"\n"
        "    <XxxData b=\"200\"\n"
        "    <XxxData c=\"20\"\n"
        "</Item>\n"
        "</XxxDataFormatterXml>\n";
    };
    auto const actual = xml.ToString(xds);

    ASSERT_EQ(expect, actual);
}

```

演習-Templateメソッド

Factory

Factoryは、専用関数(Factory関数)にオブジェクト生成をさせるためのパターンである。オブジェクトを生成するクラスや関数をそのオブジェクトの生成方法に依存させたくない場合や、オブジェクトの生成に統一されたルールを適用したい場合等に用いられる。DI(「[DI\(dependency injection\)](#)」参照)と組み合わせて使われることが多い。

「[Templateメソッド](#)」の例にFactoryを適用したソースコードを下記する。

下記のXxxDataFormatterFactory関数により、

- XxxDataFormatterIFオブジェクトはstd::unique_ptrで保持されることを強制できる
- XxxDataFormatterIFから派生したクラスはtemplate_method.cppの無名名前空間で宣言できるため、これらのクラスは他のクラスから直接依存されることがない

といった効果がある。

```
// @@@ example/design_pattern/template_method.h 73

enum class XxxDataFormatterMethod {
    Xml,
    Csv,
    Table,
};

/// @fn XxxDataFormatterFactory
/// @brief std::unique_ptrで保持されたXxxDataFormatterIFオブジェクトを生成するFactory関数
/// @param method XxxDataFormatterMethodのいずれか
/// @return std::unique_ptr<const XxxDataFormatterIF>
///         XxxDataFormatterIFはconstメンバ関数のみを持つため、戻り値もconstオブジェクト
std::unique_ptr<XxxDataFormatterIF const> XxxDataFormatterFactory(XxxDataFormatterMethod method);

// @@@ example/design_pattern/template_method.cpp 109

std::unique_ptr<XxxDataFormatterIF const> XxxDataFormatterFactory(XxxDataFormatterMethod method)
{
    switch (method) {
        case XxxDataFormatterMethod::Xml:
            return std::unique_ptr<XxxDataFormatterIF const>{new XxxDataFormatterXml}; // C++11
        case XxxDataFormatterMethod::Csv:
            return std::make_unique<XxxDataFormatterCsv const>(); // C++14 make_uniqueもFactory
        case XxxDataFormatterMethod::Table:
            return std::make_unique<XxxDataFormatterTable const>();
        default:
            assert(false);
            return {};
    }
}
```

以下に上記クラスの単体テストを示す。

```
// @@@ example/design_pattern/template_method_factory_ut.cpp 7

TEST(Factory, xml)
{
    auto xml = XxxDataFormatterFactory(XxxDataFormatterMethod::Xml);

    ...
}

TEST(Factory, csv)
{
    auto csv = XxxDataFormatterFactory(XxxDataFormatterMethod::Csv);

    ...
}

TEST(Factory, table)
{
    auto table = XxxDataFormatterFactory(XxxDataFormatterMethod::Table);

    {
        auto const xd      = XxxData{1, 100, 10};
        auto const expect = std::string_view{
            "+-----+-----+-----+\n"
            "| a     | b     | c     |\n"
            "+-----+-----+-----+\n"
            "| 1     | 100   | 10    |\n"
        };
    }
}
```

```

"-----|-----+\\n"};
auto const actual = table->ToString(xd);

ASSERT_EQ(expect, actual);
}

{
    auto const xds    = std::vector<XxxData>{{1, 100, 10}, {2, 200, 20}};
    auto const expect = std::string_view{
        "-----|-----+\\n"
        "| a    | b      | c      |\\n"
        "+-----+-----+-----+\\n"
        "| 1    | 100    | 10    |\\n"
        "+-----+-----+-----+\\n"
        "| 2    | 200    | 20    |\\n"
        "+-----+-----+-----+\\n";
    };
    auto const actual = table->ToString(xds);

    ASSERT_EQ(expect, actual);
}
}

```

一般にFactory関数はヒープを使用してオブジェクトを生成する場合が多いため、それを例示する目的でXxxDataFormatterFactoryもヒープを使用している。

この例ではその必要はないため、ヒープを使用しないFactory関数の例を下記する。

```

// @@@ example/design_pattern/template_method.cpp 126

XxxDataFormatterIF const& XxxDataFormatterFactory2(XxxDataFormatterMethod method) noexcept
{
    static auto xml    = XxxDataFormatterXml{};
    static auto csv   = XxxDataFormatterCsv{};
    static auto table = XxxDataFormatterTable{};

    switch (method) {
    case XxxDataFormatterMethod::Xml:
        return xml;
    case XxxDataFormatterMethod::Csv:
        return csv;
    case XxxDataFormatterMethod::Table:
        return table;
    default:
        assert(false);
        return xml;
    }
}

```

次に示すのは、このパターンを使用して、プリプロセッサ命令を排除するリファクタリングの例である。

まずは、出荷仕分け向けのプリプロセッサ命令をロジックの内部に記述している問題のあるコードを示す。このようなオールドスタイルなコードは様々な開発障害要因になるため、避けるべきである。

```

// in shipping.h

#define SHIP_TO_JAPAN 1
#define SHIP_TO_US 2
#define SHIP_TO_EU 3

class ShippingOp {
public:
    virtual int32_t DoSomething() = 0;
    virtual ~ShippingOp() = default;
};

```

```

// in shipping_japan.h

class ShippingOp_Japan : public ShippingOp {
public:
    ShippingOp_Japan();
    int32_t DoSomething() override;
    ~ShippingOp_Japan() override;

private:
    // 何らかの宣言
};

```

```

// in xxx.cpp 仕分けに依存した処理

// SHIPPINGはmake等のビルドツールから渡される

```

```

#ifndef SHIPPING
#define SHIPPING == SHIP_TO_JAPAN
    auto shipping = ShippingOp_Japan();
#elif SHIPPING == SHIP_TO_US
    auto shipping = ShippingOp_US();
#elif SHIPPING == SHIP_TO_EU
    auto shipping = ShippingOp_EU();
#else
#error "SHIPPING must be defined"
#endif

    shipping.DoSomething();

```

このコードは、関数テンプレートの特殊化を利用したFactoryを以下のように定義することで改善することができる。

```

// in shipping.h

// ShippingOpクラスは改善前のコードと同じ

enum class ShippingRegion { Japan, US, EU };

template <ShippingRegion>
std::unique_ptr<ShippingOp> ShippingOpFactory(); // ShippingOpFactory特殊化のための宣言

template <>
std::unique_ptr<ShippingOp> ShippingOpFactory<ShippingRegion::Japan>(); // 特殊化関数の宣言

template <>
std::unique_ptr<ShippingOp> ShippingOpFactory<ShippingRegion::US>(); // 特殊化関数の宣言

template <>
std::unique_ptr<ShippingOp> ShippingOpFactory<ShippingRegion::EU>(); // 特殊化関数の宣言

```

```

// in shipping_japan.cpp
// ファクトリーの効果で、ShippingOp_Japanは外部への公開が不要

class ShippingOp_Japan : public ShippingOp {
public:
    ShippingOp_Japan();
    int32_t DoSomething() override;
    ~ShippingOp_Japan() override;

private:
    // 何らかの宣言
};

template <>
std::unique_ptr<ShippingOp> ShippingOpFactory<ShippingRegion::Japan>()
{
    return std::unique_ptr<ShippingOp>{new ShippingOp_Japan};
}

```

```

// in xxx.cpp 仕分けに依存した処理

// SHIPPINGはmake等のビルドツールからShippingRegionのいづれかとして渡される
auto shipping = ShippingOpFactory<SHIPPING>();

shipping->DoSomething();

```

もしくは、関数オーバーロードを利用したFactoryを以下のように定義することで改善することもできる。

```

// in shipping.h

// ShippingOpクラスは改善前のコードと同じ

enum class ShippingRegion { Japan, US, EU };

template <ShippingRegion R>
class ShippingRegion2Type : std::integral_constant<ShippingRegion, R> {

};

using ShippingRegionType_Japan = ShippingRegion2Type<ShippingRegion::Japan>;
using ShippingRegionType_US   = ShippingRegion2Type<ShippingRegion::US>;
using ShippingRegionType_EU   = ShippingRegion2Type<ShippingRegion::EU>;

std::unique_ptr<ShippingOp> ShippingOpFactory(ShippingRegionType_Japan);
std::unique_ptr<ShippingOp> ShippingOpFactory(ShippingRegionType_US);
std::unique_ptr<ShippingOp> ShippingOpFactory(ShippingRegionType_EU);

```

```
// in shipping_japan.cpp
// ファクトリーの効果で、ShippingOp_Japanは外部への公開が不要

class ShippingOp_Japan : public ShippingOp {
public:
    ShippingOp_Japan();
    int32_t DoSomething() override;
    ~ShippingOp_Japan() override;

private:
    // 何らかの宣言
};

std::unique_ptr<ShippingOp> ShippingOpFactory(ShippingRegionType_Japan)
{
    return std::unique_ptr<ShippingOp>{new ShippingOp_Japan};
}
```

```
// in xxx.cpp 仕分けに依存した処理

// SHIPPINGはmake等のビルドツールからShippingRegionのいづれかとして渡される
auto shipping = ShippingOpFactory(ShippingRegion2Type<SHIPPING>{});

shipping->DoSomething();
```

演習-Factory

Named Constructor

Named Connstructorは、Singletonのようなオブジェクトを複数、生成するためのパターンである。

```
// @@@ example/design_pattern/enum_operator.h 82

class Mammals : public Animal { // 哺乳類
public:
    static Mammals& Human() noexcept
    {
        static auto inst = Mammals{PhysicalAbility::Run | PhysicalAbility::Swim};
        return inst;
    }

    static Mammals& Bat() noexcept
    {
        static auto inst = Mammals{PhysicalAbility::Run | PhysicalAbility::Fly};
        return inst;
    }

    static Mammals& Whale() noexcept
    {
        static auto inst = Mammals{PhysicalAbility::Swim};
        return inst;
    }

    bool Act();

private:
    Mammals(PhysicalAbility pa) noexcept : Animal{pa} {}
};
```

上記例のHuman()、Bat()、Whale()は、人、コウモリ、クジラに対応するクラスMammalsオブジェクトを返す。

次に示したのは「Factory」の例にこのパターンを適応したコードである。

```
// @@@ example/design_pattern/template_method.h 16

/// @class XxxDataFormatterIF
/// @brief data_storer_if.cppに定義すべきだが、サンプルであるため便宜上同じファイルで定義する
///       データフォーマットを行うクラスのインターフェースクラス
class XxxDataFormatterIF {
public:
    explicit XxxDataFormatterIF(std::string_view formatter_name) noexcept
        : formatter_name_{formatter_name}
    {
    }
    virtual ~XxxDataFormatterIF() = default;

    static XxxDataFormatterIF const& Xml() noexcept;
    static XxxDataFormatterIF const& Csv() noexcept;
```

```

    static XxxDataFormatterIF const& Table() noexcept;

    ...
};

// @@@ example/design_pattern/template_method.cpp 147

XxxDataFormatterIF const& XxxDataFormatterIF::Xml() noexcept
{
    static auto xml = XxxDataFormatterXml{};

    return xml;
}

XxxDataFormatterIF const& XxxDataFormatterIF::Csv() noexcept
{
    static auto csv = XxxDataFormatterCsv{};

    return csv;
}

XxxDataFormatterIF const& XxxDataFormatterIF::Table() noexcept
{
    static auto table = XxxDataFormatterTable{};

    return table;
}

```

これまでにXxxDataFormatterIFオブジェクトを取得するパターンを以下のように3つ示した。

1. Factory関数によってstd::unique_ptr<XxxDataFormatterIF>オブジェクトを返す。
2. Factory関数によってstaticなXxxDataFormatterIFオブジェクトを返す。
3. Named ConstructorによってstaticなXxxDataFormatterIFオブジェクトを返す。

最も汎用的な方法はパターン1であるが、上記例のようにオブジェクトが状態を持たない場合、これは過剰な方法であり、パターン3が最適であるように思える。このような考察からわかるように、(単にnewする場合も含めて)オブジェクトの取得にどのような方法を用いるかは、クラスの性質に依存する。

演習-Named Constructor

Proxy

Proxyとは代理人という意味で、本物のクラスに代わり代理クラス(Proxy)が処理を受け取る(実際は、処理自体は本物クラスに委譲されることがある)パターンである。

以下の順番で例を示すことで、Proxyパターンの説明を行う。

1. 内部構造を外部公開しているサーバクラス
2. そのサーバをラッピングして、使いやすくしたサーバクラス(Facadeパターン)
3. サーバをラップしたクラスのProxyクラス

まずは、内部構造を外部公開しているの醜悪なサーバの実装例である。

```

// @@@ example/design_pattern/bare_server.h 5

enum class Cmd {
    SayHello,
    SayGoodbye,
    Shutdown,
};

struct Packet {
    Cmd cmd;
};

class BareServer final {
public:
    BareServer() noexcept;
    ~BareServer();
    int GetPipeW() const noexcept // クライアントのwrite用
    {
        return to_server_[1];
    }

    int GetPipeR() const noexcept // クライアントのread用
};

```

```

    {
        return to_client_[0];
    }

    void Start();
    void Wait() noexcept;

private:
    int      to_server_[2]; // サーバへの通信用
    int      to_client_[2]; // クライアントへの通信用
    std::thread thread_;
};

// @@@ example/design_pattern/bare_server.cpp 9

namespace {
bool cmd_dispatch(int wfd, Cmd cmd) noexcept
{
    static char const hello[] = "Hello";
    static char const goodbye[] = "Goodbye";

    switch (cmd) {
    case Cmd::SayHello:
        write(wfd, hello, sizeof(hello));
        break;
    case Cmd::SayGoodbye:
        write(wfd, goodbye, sizeof(goodbye));
        break;
    case Cmd::Shutdown:
    default:
        std::cout << "Shutdown" << std::endl;
        return false;
    }

    return true;
}

void thread_entry(int rfd, int wfd) noexcept
{
    for (;;) {
        auto packet = Packet{};

        if (read(rfd, &packet, sizeof(packet)) < 0) {
            continue;
        }

        if (!cmd_dispatch(wfd, packet.cmd)) {
            break;
        }
    }
}
} // namespace

BareServer::BareServer() noexcept : to_server_{-1, -1}, to_client_{-1, -1}, thread_={}
{
    auto ret = pipe(to_server_);
    assert(ret >= 0);

    ret = pipe(to_client_);
    assert(ret >= 0);
}

BareServer::~BareServer()
{
    close(to_server_[0]);
    close(to_server_[1]);
    close(to_client_[0]);
    close(to_client_[1]);
}

void BareServer::Start()
{
    thread_ = std::thread{thread_entry, to_server_[0], to_client_[1]};
    std::cout << "thread started !!!" << std::endl;
}

void BareServer::Wait() noexcept { thread_.join(); }

```

下記は、上記BareServerを使用するクライアントの実装例である。通信がpipe()によって行われ、その中身がPacket{}であること等、不要な依存関係をbare_client()に強いていることがわかる。このような構造は、機能追加、保守作業を非効率、困難にするアンチパターンである。

```

// @@@ example/design_pattern/proxy_ut.cpp 17

/// @fn bare_client
/// @brief 非同期サービスを隠蔽していないBareServerを使用したときのクライアントの例
std::vector<std::string> bare_client(BareServer& bs)
{
    auto const wfd = bs.GetPipeW();
    auto const rfd = bs.GetPipeR();
    auto      ret = std::vector<std::string>{};

    bs.Start();

    auto packet = Packet{};
    char buffer[30];

    packet.cmd = Cmd::SayHello;
    write(wfd, &packet, sizeof(packet));

    auto read_ret = read(rfd, buffer, sizeof(buffer));
    assert(read_ret > 0);

    ret.emplace_back(buffer);

    packet.cmd = Cmd::SayGoodbye;
    write(wfd, &packet, sizeof(packet));

    read_ret = read(rfd, buffer, sizeof(buffer));
    assert(read_ret > 0);

    ret.emplace_back(buffer);

    packet.cmd = Cmd::Shutdown;
    write(wfd, &packet, sizeof(packet));

    bs.Wait();

    return ret;
}

```

次に、この書き出しの構造をラッピングする例を示す(このようなラッピングをFacadeパターンと呼ぶ)。

```

// @@@ example/design_pattern/bare_server_wrapper.h 6

enum class Cmd; // C++11からenumは前方宣言できる。
class BareServer;

class BareServerWrapper final {
public:
    BareServerWrapper();

    void      Start();
    std::string SayHello();
    std::string SayGoodbye();
    void      Shutdown() noexcept;

private:
    void          send_message(enum Cmd cmd) noexcept;
    std::unique_ptr<BareServer> bare_server_;
};

// @@@ example/design_pattern/bare_server_wrapper.cpp 8

BareServerWrapper::BareServerWrapper() : bare_server_{std::make_unique<BareServer>()} {}

void BareServerWrapper::Start() { bare_server_->Start(); }

void BareServerWrapper::send_message(enum Cmd cmd) noexcept
{
    auto packet = Packet{cmd};

    write(bare_server_->GetPipeW(), &packet, sizeof(packet));
}

std::string BareServerWrapper::SayHello()
{
    char buffer[30];

    send_message(Cmd::SayHello);
    read(bare_server_->GetPipeR(), buffer, sizeof(buffer));
}

```

```

    return buffer;
}

std::string BareServerWrapper::SayGoodbye()
{
    char buffer[30];

    send_message(Cmd::SayGoodbye);
    read(bare_server_->GetPipeR(), buffer, sizeof(buffer));

    return buffer;
}

void BareServerWrapper::Shutdown() noexcept
{
    send_message(Cmd::Shutdown);

    bare_server_->Wait();
}

```

下記は、上記BareServerWrapperのクライアントの実装例である。 BareServerWrapperがむき出しの通信をラップしたことで、bare_wrapper_client()は、bare_client()に比べてシンプルになったことがわかる。

```

// @@@ example/design_pattern/proxy_ut.cpp 57

/// @fn bare_wrapper_client
/// @brief BareServerを使いやくらップしたBareServerWrapperを使用したときのクライアントの例
std::vector<std::string> bare_wrapper_client(BareServerWrapper& bsw)
{
    auto ret = std::vector<std::string>{};

    bsw.Start();

    ret.emplace_back(bsw.SayHello());

    ret.emplace_back(bsw.SayGoodbye());

    bsw.Shutdown();

    return ret;
}

```

次の例は、BareServerとBareServerWrapperを統合し、さらに全体をシンプルにリファクタリングしたWrappedServerである。Packet{}やpipe等の通信の詳細がwrapped_server.cppの無名名前空間に閉じ込められ、クラスの隠蔽性が強化されたことで、より機能追加、保守が容易になった。

```

// @@@ example/design_pattern/wrapped_server.h 5

class WrappedServer {
public:
    WrappedServer() noexcept;
    virtual ~WrappedServer();

    void Start();
    std::string SayHello() { return say_hello(); }
    std::string SayGoodbye() { return say_goodbye(); }
    void Shutdown() noexcept;

protected:
    virtual std::string say_hello(); // 後で拡張するためにvirtual
    virtual std::string say_goodbye(); // 同上

private:
    int to_server_[2];
    int to_client_[2];
    std::thread thread_;
};

```

```

// @@@ example/design_pattern/wrapped_server.cpp 8

namespace {
enum class Cmd {
    ...
};

struct Packet {
    Cmd cmd;
};

} // namespace

```

```
// 以下、bare_server_wrapper.cppのコードとほぼ同じであるため省略。
```

```
...
```

WrappedServerの使用例を下記する。当然ながらbare_wrapper_client()とほぼ同様になる。

```
// @@@ example/design_pattern/proxy_ut.cpp 7

/// @fn wrapped_client
/// @brief 非同期サービスを隠蔽しているWrappedServerを使用したときのクライアントの例
std::vector<std::string> wrapped_client(WrappedServer& ws)
{
    auto ret = std::vector<std::string>{};

    ws.Start();

    ret.emplace_back(ws.SayHello());

    ret.emplace_back(ws.SayGoodbye());

    wsShutdown();

    return ret;
}
```

WrappedServerが提供する機能はスレッド間通信を含むため処理コストが高い。その対策として、サーバから送られてきた文字列をキャッシュするクラス(Proxyパターン)の導入により、そのコストを削減する例を下記する。

```
// @@@ example/design_pattern/wrapped_server_proxy.h 7

class WrappedServerProxy final : public WrappedServer {
public:
    WrappedServerProxy() = default;

private:
    std::string hello_cashe_{};
    virtual std::string say_hello() override;
    virtual std::string say_goodbye() override;
};

// @@@ example/design_pattern/wrapped_server_proxy.cpp 7

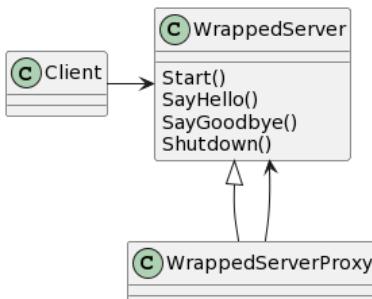
std::string WrappedServerProxy::say_hello()
{
    if (hello_cashe_.size() == 0) {
        hello_cashe_ = WrappedServer::say_hello(); // キャッシュとし保存
    }

    return hello_cashe_;
}

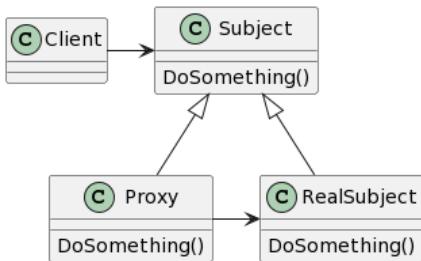
std::string WrappedServerProxy::say_goodbye()
{
    hello_cashe_ = std::string{}; // helloキャッシュをクリア

    return WrappedServer::say_goodbye();
}
```

下記図のようにWrappedServerProxyはWrappedServerからのパブリック継承であるため、WrappedServerのクライアントは、そのままWrappedServerProxyのクライアントとして利用できる。



なお、正確には下記のようなクラス構造をProxyパターンと呼ぶことが多いが、ここでは単純さを優先した。



演習-Proxy

Strategy

関数f(args)の振る舞いが、

- ・全体の制御
- ・部分的な振る舞い(何らかの条件を探す等)

に分けられるような場合、関数fを

- ・「全体の制御」を行う関数
- ・「部分的な振る舞い」を規定するStrategyオブジェクト(関数へのポインタ、関数オブジェクト、ラムダ式)

に分割し、下記のように、Strategyオブジェクトをgの引数として外部から渡せるようにしたパターンである(std::sort()のようなパターン)。

g(args, Strategyオブジェクト)

Strategyオブジェクトにいろいろなバリエーションがある場合、このパターンを使うと良い。なお、このパターンの対象はクラスになる場合もある。

「ディレクトリをリカーシブに追跡し、引数で指定された属性にマッチしたファイルの一覧を返す関数」を開発することを要求されたとする。

まずは、拡張性のない実装例を示す。

```

// @@@ example/design_pattern/find_files_old_style.h 4

/// @enum FindCondition
/// find_files_recursivelyの条件
enum class FindCondition {
    File,           ///< pathがファイル
    Dir,            ///< pathがディレクトリ
    FileNameHeadIs_f,  ///< pathがファイル且つ、そのファイル名の先頭が"f"
};

// @@@ example/design_pattern/find_files_old_style.cpp 9

/// @fn std::vector<std::string> find_files_recursively(std::string const& path,
///                                                       FindCondition condition)
/// @brief 条件にマッチしたファイルをリカーシブに探して返す
/// @param path リカーシブにディレクトリをたどるための起点となるパス
/// @param condition どのようなファイルかを指定する
/// @return 条件にマッチしたファイルをstd::vector<std::string>で返す
std::vector<std::string> find_files_recursively(std::string const& path, FindCondition condition)
{
    auto files = std::vector<fs::path>{};

    // recursive_directory_iteratorはファイルシステム依存するため、その依存を排除する他の処理
    std::copy(fs::recursive_directory_iterator{path}, fs::recursive_directory_iterator{}, 
              std::back_inserter(files));

    std::sort(files.begin(), files.end());

    auto ret = std::vector<std::string>{};

    std::for_each(files.begin(), files.end(), [&](fs::path const& p) noexcept {
        auto is_match = false;

        switch (condition) {
        case FindCondition::File:
            if (fs::is_regular_file(p)) {
                is_match = true;
            }
        }
    });
}

```

```

        }
        break;
    case FindCondition::Dir:
        if (fs::is_directory(p)) {
            is_match = true;
        }
        break;
    ...
}

if (is_match) {
    ret.emplace_back(p.generic_string());
}
});

return ret;
}

```

```

// @@@ example/design_pattern/find_files_ut.cpp 29

TEST(Strategy, old_style)
{
    assure_test_files_exist(); // test用のファイルがあることの確認

    auto const files_actual = find_files_recursively(test_dir, FindCondition::File);
    auto const files_expect = sort(std::vector{
        test_dir + "dir0/file2",
        test_dir + "dir0/gile3",
        test_dir + "dir1/dir2/file4",
        test_dir + "file0",
        test_dir + "gile1"
    });
    ASSERT_EQ(files_expect, files_actual);

    auto const dirs_actual = find_files_recursively(test_dir, FindCondition::Dir);
    auto const dirs_expect = sort(std::vector{
        test_dir + "dir0",
        test_dir + "dir1",
        test_dir + "dir1/dir2"
    });
    ASSERT_EQ(dirs_expect, dirs_actual);

    auto const f_actual = find_files_recursively(test_dir, FindCondition::FileNameHeadIs_f);
    auto const f_expect = sort(std::vector{
        test_dir + "dir0/file2",
        test_dir + "dir1/dir2/file4",
        test_dir + "file0"
    });
    ASSERT_EQ(f_expect, f_actual);
}

```

この関数は、見つかったファイルが「引数で指定された属性」にマッチするかどうかを検査する。検査は、「引数で指定された属性」に対するswitch文によって行われる。これにより、この関数は「引数で指定された属性」の変更に強く影響を受ける。

下記は、この関数にStrategyパターンを適用したものである。

```

// @@@ example/design_pattern/find_files_strategy.h 7

/// @typedef find_condition
/// @brief find_files_recursively返引数conditionの型(関数オブジェクトの型)
using find_condition = std::function<bool(std::filesystem::path const&)>

// Strategyパターン
/// @fn std::vector<std::string> find_files_recursively(std::string const& path,
///                                                       find_condition condition);
/// @brief 条件にマッチしたファイルをリカーシブに探索して返す
/// @param path リカーシブにディレクトリを辿るための起点となるパス
/// @param condition 探索するファイルの条件
/// @return 条件にマッチしたファイルをstd::vector<std::string>で返す
extern std::vector<std::string> find_files_recursively(std::string const& path,
                                                       find_condition condition);

// @@@ example/design_pattern/find_files_strategy.cpp 6

std::vector<std::string> find_files_recursively(std::string const& path, find_condition condition)
{
    namespace fs = std::filesystem;

    auto files = std::vector<fs::path>{};

```

```

// recursive_directory_iteratorはファイルシステム依存するため、その依存を排除する他の処理
std::copy(fs::recursive_directory_iterator{path}, fs::recursive_directory_iterator{}, std::back_inserter(files));

std::sort(files.begin(), files.end());

auto ret = std::vector<std::string>{};

std::for_each(files.cbegin(), files.cend(), [&](fs::path const& p) {
    if (condition(p)) {
        ret.emplace_back(p.generic_string());
    }
});

return ret;
}

```

```
// @@@ example/design_pattern/find_files_ut.cpp 69
```

```

TEST(Strategy, strategy_lambda)
{
    namespace fs = std::filesystem;

    assure_test_files_exist(); // test用のファイルがあることの確認

    // ラムダ式で実装
    auto const files_actual = find_files_recursively(
        test_dir, [] (fs::path const& p) noexcept { return fs::is_regular_file(p); });

    auto const files_expect = sort(std::vector{
        test_dir + "dir0/file2",
        test_dir + "dir0/gile3",
        test_dir + "dir1/dir2/file4",
        test_dir + "file0",
        test_dir + "gile1"
    });
    ASSERT_EQ(files_expect, files_actual);

    auto const dirs_actual = find_files_recursively(
        test_dir, [] (fs::path const& p) noexcept { return fs::is_directory(p); });
    auto const dirs_expect = sort(std::vector{
        test_dir + "dir0",
        test_dir + "dir1",
        test_dir + "dir1/dir2"
    });
    ASSERT_EQ(dirs_expect, dirs_actual);

    auto const f_actual = find_files_recursively(test_dir, [] (fs::path const& p) noexcept {
        return p.filename().generic_string()[0] == 'f';
    });

    auto const f_expect = sort(std::vector{
        test_dir + "dir0/file2",
        test_dir + "dir1/dir2/file4",
        test_dir + "file0"
    });
    ASSERT_EQ(f_expect, f_actual);
}

/// @fn bool condition_func(std::filesystem::path const& path)
/// @brief find_files_recursivelyの第2引数に渡すためのファイル属性を決める関数
bool condition_func(std::filesystem::path const& path)
{
    return path.filename().generic_string().at(0) == 'f';
}

TEST(Strategy, strategy_func_pointer)
{
    assure_test_files_exist(); // test用のファイルがあることの確認

    // FindCondition::FileNameHeadIs_fで行ったことを関数ポインタで実装。
    auto const f_actual = find_files_recursively(test_dir, condition_func);
    auto const f_expect = sort(std::vector{
        test_dir + "dir0/file2",
        test_dir + "dir1/dir2/file4",
        test_dir + "file0"
    });
    ASSERT_EQ(f_expect, f_actual);
}

```

```

/// @class ConditionFunctor
/// @brief
/// find_files_recursivelyの第2引数に渡すためのファイル属性を決める関数オブジェクトクラス。
/// 検索条件に状態が必要な場合、関数オブジェクトを使うとよい。
class ConditionFunctor {
public:
    ConditionFunctor() = default;
    ~ConditionFunctor() = default;

    /// @fn bool operator()(std::filesystem::path const& path)
    /// @brief 先頭が'f'のファイルを最大2つまで探す
    bool operator()(std::filesystem::path const& path)
    {
        if (path.filename().generic_string().at(0) != 'f') {
            return false;
        }

        return ++count_ < 3;
    }

private:
    int32_t count_{0};
};

TEST(Strategy, strategy_func_obj)
{
    // 条件に状態が必要な場合(この例では最大2つまでを判断するのに状態が必要)、
    // 関数ポインタより、ファンクタの方が便利。
    auto const f_actual = find_files_recursively(test_dir, ConditionFunctor{});
    auto const f_expect = sort(std::vector{
        test_dir + "dir0/file2",
        test_dir + "dir1/dir2/file4",
    });
    ASSERT_EQ(f_expect, f_actual);
}

```

検査対象のファイル属性の指定をfind_files_recursively()の外に出したため、その属性の追加に対して「[オープン・クローズドの原則\(OCP\)](#)」に対応した構造となった。

なお、上記find_files_recursivelyの第2パラメータをテンプレートパラメータとしてすることで、

```

// @@@ example/design_pattern/find_files_strategy.h 23

template <typename F> // Fはファンクタ
auto find_files_recursively2(std::string const& path, F condition)
    -> std::enable_if_t<std::is_invocable_r_v<bool, F, std::filesystem::path const&, std::vector<std::string>>
{
    namespace fs = std::filesystem;

    auto files = std::vector<fs::path>{};

    // recursive_directory_iteratorはファイルシステム依存するため、その依存を排除する他の処理
    std::copy(fs::recursive_directory_iterator{path}, fs::recursive_directory_iterator{}, std::back_inserter(files));

    std::sort(files.begin(), files.end());

    auto ret = std::vector<std::string>{};

    std::for_each(files.cbegin(), files.cend(), [&](fs::path const& p) {
        if (condition(p)) {
            ret.emplace_back(p.generic_string());
        }
    });

    return ret;
}

```

のように書くこともできる。

次に示すのは、このパターンを使用して、プリプロセッサ命令を排除するリファクタリングの例である。

まずは、出荷仕分け向けのプリプロセッサ命令をロジックの内部に記述している問題のあるコードを示す。このようなオールドスタイルなコードは様々な開発障害要因になるため、避けるべきである。

```

// @@@ example/design_pattern/strategy_shipping_ut.cpp 11

class X {

```

```

public:
    X() = default;

    int32_t DoSomething()
    {
        int32_t ret{0};

#If SHIPPING == SHIP_TO_JAPAN
        // 日本向けの何らかの処理
#elif SHIPPING == SHIP_TO_US
        // US向けの何らかの処理
#elif SHIPPING == SHIP_TO_JAPAN
        // EU向けの何らかの処理
#else
#error "SHIPPING must be defined"
#endif
        return ret;
    }

private:
    // 何らかの宣言
};

```

```

// @@@ example/design_pattern/strategy_shipping_ut.cpp 43

X x;

x.DoSomething();

```

このコードは、Strategyを使用し以下のようにすることで、改善することができる。

```

// @@@ example/design_pattern/strategy_shipping_ut.cpp 56

class ShippingOp {
public:
    virtual int32_t DoSomething() = 0;
    virtual ~ShippingOp() = default;
};

class X {
public:
    X() = default;

    int32_t DoSomething(ShippingOp& shipping)
    {
        int32_t ret = shipping.DoSomething();

        // 何らかの処理

        return ret;
    }

private:
    // 何らかの宣言
};

```

```

// @@@ example/design_pattern/strategy_shipping_ut.cpp 81

class ShippingOp_Japan : public ShippingOp {
public:
    ShippingOp_Japan();
    int32_t DoSomething() override;
    ~ShippingOp_Japan() override;

private:
    // 何らかの宣言
};

// @@@ example/design_pattern/strategy_shipping_ut.cpp 100

X x;
ShippingOp_Japan sj;

x.DoSomething(sj);

```

あるいは、[DI\(dependency injection\)](#)と組み合わせて、下記のような改善も有用である。

```

// @@@ example/design_pattern/strategy_shipping_ut.cpp 112

class ShippingOp {

```

```

public:
    virtual int32_t DoSomething() = 0;
    virtual ~ShippingOp() = default;
};

class X {
public:
    explicit X(std::unique_ptr<ShippingOp> shipping) : shipping_{std::move(shipping)} {}

    int32_t DoSomething()
    {
        int32_t ret = shipping_->DoSomething();

        // 何らかの処理

        return ret;
    }

private:
    std::unique_ptr<ShippingOp> shipping_;
    // 何らかの宣言
};

```

```

// @@@ example/design_pattern/strategy_shipping_ut.cpp 138

class ShippingOp_Japan : public ShippingOp {
public:
    ShippingOp_Japan();
    int32_t DoSomething() override;
    ~ShippingOp_Japan() override;

private:
    // 何らかの宣言
};

```

```

// @@@ example/design_pattern/strategy_shipping_ut.cpp 157

X x{std::unique_ptr<ShippingOp>(new ShippingOp_Japan)};

x.DoSomething();

```

演習-Strategy

Visitor

このパターンは、クラス構造とそれに関連するアルゴリズムを分離するためのものである。

最初に「クラス構造とそれに関連するアルゴリズムは分離できているが、それ以前にオブジェクト指向の原則に反している」例を示す。

```

// @@@ example/design_pattern/visitor.cpp 42

/// @class FileEntity
/// @brief ファイルシステムの構成物(ファイル、ディレクトリ等)を表すクラスの基底クラス
class FileEntity {
public:
    explicit FileEntity(std::string pathname) : pathname_{std::move(pathname)} {}
    virtual ~FileEntity() {}
    std::string const& Pathname() const { return pathname_; }

    ...

private:
    std::string const pathname_;
};

class File final : public FileEntity {
    ...
};

class Dir final : public FileEntity {
    ...
};

class OtherEntity final : public FileEntity {
    ...
};

```

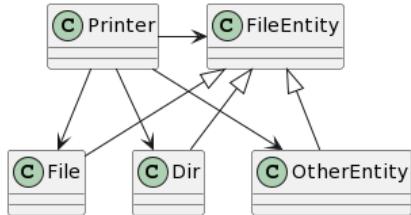
```

class Printer {
public:
    static void PrintPathname1(FileEntity const& file_entity)
    {
        if (typeid(File) == typeid(file_entity)) {
            std::cout << file_entity.Pathname();
        }
        else if (typeid(Dir) == typeid(file_entity)) {
            std::cout << file_entity.Pathname() + "/";
        }
        else if (typeid(OtherEntity) == typeid(file_entity)) {
            std::cout << file_entity.Pathname() + "(o1)";
        }
        else {
            assert(false);
        }
    }

    static void PrintPathname2(FileEntity const& file_entity)
    {
        if (typeid(File) == typeid(file_entity)) {
            std::cout << file_entity.Pathname();
        }
        else if (typeid(Dir) == typeid(file_entity)) {
            std::cout << find_files(file_entity.Pathname());
        }
        else if (typeid(OtherEntity) == typeid(file_entity)) {
            std::cout << file_entity.Pathname() + "(o2)";
        }
        else {
            assert(false);
        }
    }
};

```

下記クラス図からもわかる通り、ポリモーフィズムに反したこのような構造は複雑な依存関係を作り出す。このアンチパターンにより同型の条件文が2度出てきてしまうため、Printerのアルゴリズム関数が増えれば、この繰り返しはそれに比例して増える。またFileEntityの派生が増えれば、それら条件文はすべて影響を受ける。このようなソースコードは、このようにして等比級数的に複雑化する。



これをポリモーフィズムの導入で解決した例を示す。

```

// @@@ example/design_pattern/visitor.cpp 143

class FileEntity {
public:
    explicit FileEntity(std::string pathname) : pathname_(std::move(pathname)) {}
    ...
    virtual void PrintPathname1() const = 0;
    virtual void PrintPathname2() const = 0;

private:
    std::string const pathname_;
};

class File final : public FileEntity {
public:
    ...
    virtual void PrintPathname1() const override { std::cout << Pathname(); }
    virtual void PrintPathname2() const override { std::cout << Pathname(); }
};

class Dir final : public FileEntity {
public:
    ...
    virtual void PrintPathname1() const override { std::cout << Pathname() + "/"; }
    virtual void PrintPathname2() const override { std::cout << find_files(Pathname()); }
};

class OtherEntity final : public FileEntity {

```

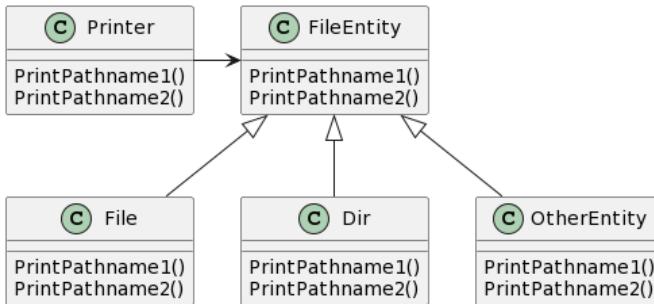
```

public:
    ...
    virtual void PrintPathname1() const override { std::cout << Pathname() + "(o1)"; }
    virtual void PrintPathname2() const override { std::cout << Pathname() + "(o2)"; }
};

class Printer {
public:
    static void PrintPathname1(FileEntity const& file_entity) { file_entity.PrintPathname1(); }
    static void PrintPathname2(FileEntity const& file_entity) { file_entity.PrintPathname2(); }
};

```

上記例では、PrinterのアルゴリズムをFileEntityの各派生クラスのメンバ関数で実装することで、Printerの各関数は単純化された。



これはポリモーフィズムによるリファクタリングの良い例と言えるが、SRP(「单一責任の原則(SRP)」)に反するため、Printerの関数が増えるたびにPrintPathname1、PrintPathname2のようなFileEntityのインターフェースが増えてしまう。

このようなインターフェースの肥大化に対処するパターンがVisitorである。

上記例にVisitorを適用してリファクタリングした例を示す。

```

// @@@ example/design_pattern/visitor.h 9

class FileEntityVisitor {
public:
    virtual void Visit(File const&) = 0;
    virtual void Visit(Dir const&) = 0;
    virtual void Visit(OtherEntity const&) = 0;
    ...
};

class FileEntity {
public:
    explicit FileEntity(std::string pathname) : pathname_{std::move(pathname)} {}
    ...
    std::string const& Pathname() const { return pathname_; }

    virtual void Accept(FileEntityVisitor&) const = 0; // Acceptの仕様は安定しているので
                                                       // NVIは使わない。
private:
    std::string const pathname_;
};

class File final : public FileEntity {
public:
    using FileEntity::FileEntity;
    virtual void Accept(FileEntityVisitor& visitor) const override { visitor.Visit(*this); }
};

class Dir final : public FileEntity {
public:
    using FileEntity::FileEntity;
    virtual void Accept(FileEntityVisitor& visitor) const override { visitor.Visit(*this); }
};

class OtherEntity final : public FileEntity {
public:
    using FileEntity::FileEntity;
    virtual void Accept(FileEntityVisitor& visitor) const override { visitor.Visit(*this); }
};

class PathnamePrinter1 final : public FileEntityVisitor {
public:
    virtual void Visit(File const&) override;
    virtual void Visit(Dir const&) override;
}

```

```

    virtual void Visit(OtherEntity const&) override;
};

class PathnamePrinter2 final : public FileEntityVisitor {
public:
    virtual void Visit(File const&) override;
    virtual void Visit(Dir const&) override;
    virtual void Visit(OtherEntity const&) override;
};

// @@@ example/design_pattern/visitor.cpp 219

void PathnamePrinter1::Visit(File const& file) { std::cout << file.Pathname(); }
void PathnamePrinter1::Visit(Dir const& dir) { std::cout << dir.Pathname() + "/"; }
void PathnamePrinter1::Visit(OtherEntity const& other) { std::cout << other.Pathname() + "(o1)"; }

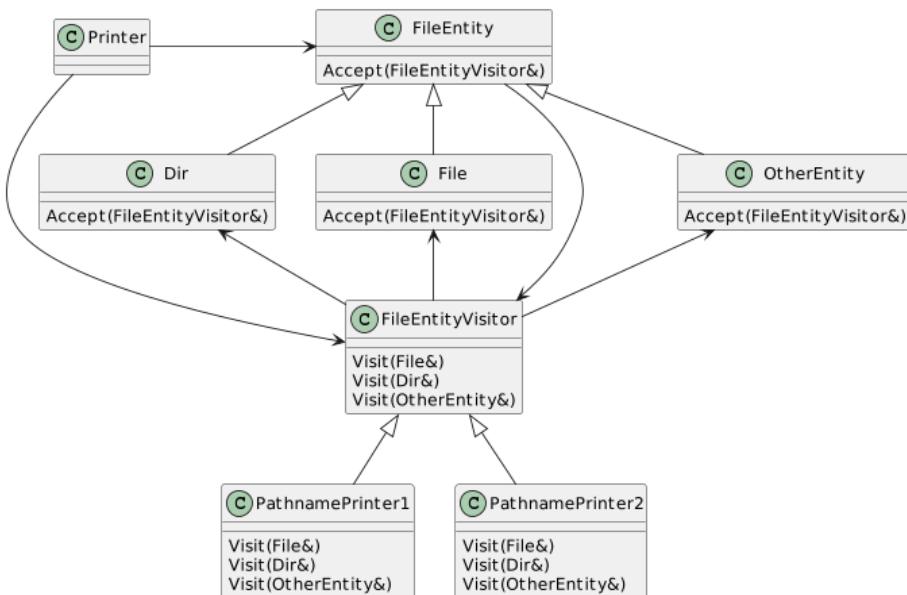
void PathnamePrinter2::Visit(File const& file) { std::cout << file.Pathname(); }
void PathnamePrinter2::Visit(Dir const& dir) { std::cout << find_files(dir.Pathname()); }
void PathnamePrinter2::Visit(OtherEntity const& other) { std::cout << other.Pathname() + "(o2)"; }

class Printer {
public:
    static void PrintPathname1(FileEntity const& file_entity)
    {
        auto visitor = PathnamePrinter1{};
        file_entity.Accept(visitor);
    }

    static void PrintPathname2(FileEntity const& file_entity)
    {
        auto visitor = PathnamePrinter2{};
        file_entity.Accept(visitor);
    }
};

```

上記クラスの関係は下記のようになる。



このリファクタリングには、

- FileEntityのインターフェースを小さくできる
- FileEntityVisitorから派生できるアルゴリズムについては、 FileEntityのインターフェースに影響を与えずに追加できる（「オープン・クローズドの原則(OCP)」参照）

という利点がある。一方で、この程度の複雑さの(単純な)例では、Visitorの適用によって以前よりも構造が複雑になり、改悪してしまった可能性があるため、デザインパターンを使用する場合には注意が必要である。

なお、上記の抜粋である下記コード

```

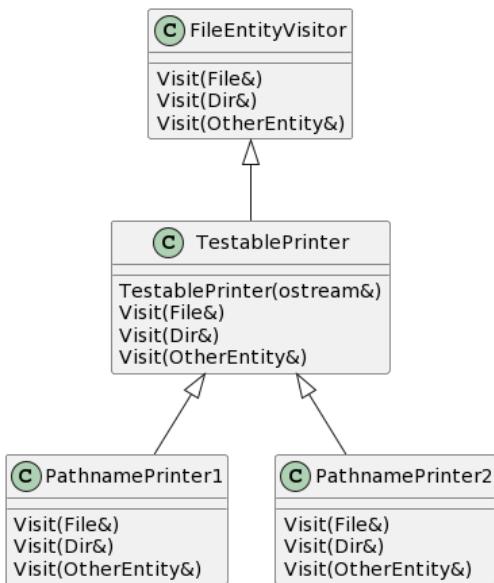
// @@@ example/design_pattern/visitor.h 39

virtual void Accept(FileEntityVisitor& visitor) const override { visitor.Visit(*this); }

```

はコードクローンだが、thisの型が違うため、各Acceptが呼び出すFileEntityVisitor::Visit()も異り、単純に統一することはできない。これを改めるためには、「[CRTP\(curiously recurring template pattern\)](#)」が必要になる。

次に示すソースコードはVisitorとは関係がないが、FileEntityVisitorから派生するクラスを下記クラス図が示すように改善することで、単体テストが容易になる例である(「[DI\(dependency injection\)](#)」参照)。



```

// @@@ example/design_pattern/visitor.h 72

class TestablePrinter : public FileEntityVisitor {
public:
    explicit TestablePrinter(std::ostream& os) : ostream_{os} {}

protected:
    std::ostream& ostream_;
};

class TestablePathnamePrinter1 final : public TestablePrinter {
public:
    explicit TestablePathnamePrinter1(std::ostream& os) : TestablePrinter{os} {}
    virtual void Visit(File const& file) override;
    virtual void Visit(Dir const& dir) override;
    virtual void Visit(OtherEntity const& other) override;
};

class TestablePathnamePrinter2 final : public TestablePrinter {
public:
    explicit TestablePathnamePrinter2(std::ostream& os) : TestablePrinter{os} {}
    virtual void Visit(File const& file) override;
    virtual void Visit(Dir const& dir) override;
    virtual void Visit(OtherEntity const& other) override;
};

// @@@ example/design_pattern/visitor.cpp 246

void TestablePathnamePrinter1::Visit(File const& file) { ostream_ << file.Pathname(); }
void TestablePathnamePrinter1::Visit(Dir const& dir) { ostream_ << dir.Pathname() + "/"; }
void TestablePathnamePrinter1::Visit(OtherEntity const& other)
{
    ostream_ << other.Pathname() + "(o1)";
}

void TestablePathnamePrinter2::Visit(File const& file) { ostream_ << file.Pathname(); }

void TestablePathnamePrinter2::Visit(Dir const& dir) { ostream_ << find_files(dir.Pathname()); }

void TestablePathnamePrinter2::Visit(OtherEntity const& other)
{
    ostream_ << other.Pathname() + "(o2)";
}

// @@@ example/design_pattern/visitor_ut.cpp 28

TEST(Visitor, testable_visitor)
{
}
  
```

```

auto oss = std::ostringstream{};

// 出力をキャプチャするため、std::coutに代えてossを使う
auto visitor1 = TestablePathnamePrinter1{oss};
auto visitor2 = TestablePathnamePrinter2{oss};

auto file = File{"visitor.cpp"};
{
    file.Accept(visitor1);
    ASSERT_EQ("visitor.cpp", oss.str());
    oss = {};
}
{
    file.Accept(visitor2);
    ASSERT_EQ("visitor.cpp", oss.str());
    oss = {};
}

auto dir = Dir{"find_files_ut_dir/dir0"};
{
    dir.Accept(visitor1);
    ASSERT_EQ("find_files_ut_dir/dir0/", oss.str());
    oss = {};
}
{
    dir.Accept(visitor2);
    ASSERT_EQ("find_files_ut_dir/dir0/file2,find_files_ut_dir/dir0/file3", oss.str());
    oss = {};
}
}

```

演習-Visitor

CRTP(curiously recurring template pattern)

CRTPとは、

```

// @@@ example/design_pattern/crtpt_ut.cpp 8

template <typename T>
class Base {
    ...
};

class Derived : public Base<Derived> {
    ...
};

```

のようなテンプレートによる再帰構造を用いて、静的ポリモーフィズムを実現するためのパターンである。

このパターンを用いて、「Visitor」のFileEntityの3つの派生クラスが持つコードクローン

```

// @@@ example/design_pattern/visitor.h 39

virtual void Accept(FileEntityVisitor& visitor) const override { visitor.Visit(*this); }

```

を解消した例を以下に示す。

```

// @@@ example/design_pattern/crtpt.h 31

class FileEntity { // VisitorのFileEntityと同じ
public:
    explicit FileEntity(std::string&& pathname) : pathname_{std::move(pathname)} {}
    virtual ~FileEntity() {}
    std::string const& Pathname() const { return pathname_; }

    virtual void Accept(FileEntityVisitor&) const = 0; // Acceptの仕様は安定しているので
                                                       // NVIは使わない。
private:
    std::string const pathname_;
};

template <typename T>
class AcceptableFileEntity : public FileEntity { // CRTP
public:
    virtual void Accept(FileEntityVisitor& visitor) const override
    {
        visitor.Visit(*static_cast<T const*>(this));
    }
};

```

```

    }

private:
    // T : public AcceptableFileEntity<T> { ... };
    // 以外の使い方をコンパイルエラーにする
    AcceptableFileEntity(std::string&& pathname) : FileEntity{std::move(pathname)} {}
        friend T;
};

class File final : public AcceptableFileEntity<File> { // CRTPでクローンを解消
public:
    File(std::string pathname) : AcceptableFileEntity{std::move(pathname)} {}
};

class Dir final : public AcceptableFileEntity<Dir> { // CRTPでクローンを解消
public:
    Dir(std::string pathname) : AcceptableFileEntity{std::move(pathname)} {}
};

class OtherEntity final : public AcceptableFileEntity<OtherEntity> { // CRTPでクローンを解消
public:
    OtherEntity(std::string pathname) : AcceptableFileEntity{std::move(pathname)} {}
};

```

Observer

Observerは、クラスSubjectと複数のクラスObserverN(N = 0, 1, 2 ...)があり、この関係が下記の条件を満たさなければならない場合に使用されるパターンである。

- ObserverNオブジェクトはSubjectオブジェクトが変更された際、その変更通知を受け取る。
- SubjectはObserverNへ依存してはならない。

GUIアプリケーションをMVCで実装する場合のModelがSubjectであり、ViewがObserverNである。

まずは、このパターンを使用しない実装例を示す。

```

// @@@ example/design_pattern/observer_ng.h 6

/// @class ObserverNG_N
/// @brief SubjectNGからの変更通知をUpdate()で受け取る。
///     Observerパターンを使用しない例。
class ObserverNG_0 {
public:
    ObserverNG_0() = default;

    virtual void Update(SubjectNG const& subject) // テストのためにvirtual
    {
        // 何らかの処理
    }

    virtual ~ObserverNG_0() = default;
    // 何らかの定義、宣言
};

class ObserverNG_1 {
public:
    ...
};

class ObserverNG_2 {
public:
    ...
};

```

```

// @@@ example/design_pattern/observer_ng.cpp 6

void ObserverNG_1::Update(SubjectNG const& subject)
{
    ...
}

void ObserverNG_2::Update(SubjectNG const& subject)
{
    ...
}

```

```
// @@@ example/design_pattern/subject_ng.h 9
```

```

/// @class SubjectNG
/// @brief 監視されるクラス。SetNumでの状態変更をObserverNG_Nに通知する。
///         Observerパターンを使用しない例。
class SubjectNG final {
public:
    explicit SubjectNG(ObserverNG_0& ng_0, ObserverNG_1& ng_1, ObserverNG_2& ng_2) noexcept
        : num_{0}, ng_0_{ng_0}, ng_1_{ng_1}, ng_2_{ng_2}
    {}

    void SetNum(uint32_t num);
    ...
};

```

```

// @@@ example/design_pattern/subject_ng.cpp 4

void SubjectNG::SetNum(uint32_t num)
{
    if (num_ == num) {
        return;
    }

    num_ = num;

    notify(); // subjectが変更されたことをobserverへ通知
}

void SubjectNG::notify()
{
    ng_0_.Update(*this);
    ng_1_.Update(*this);
    ng_2_.Update(*this);
}

```

```

// @@@ example/design_pattern/observer_ut.cpp 15

struct ObserverNG_0_Test : ObserverNG_0 { // テスト用クラス
    virtual void Update(SubjectNG const& subject) final
    {
        ++call_count;
        num = subject.GetNum();
    }

    uint32_t call_count{0};
    std::optional<uint32_t> num{};
};

auto ng0 = ObserverNG_0_Test{};
auto ng1 = ObserverNG_1{};
auto ng2 = ObserverNG_2{};

auto subject = SubjectNG{ng0, ng1, ng2};

ASSERT_EQ(0, ng0.call_count); // まだ何もしていない
ASSERT_FALSE(ng0.num);

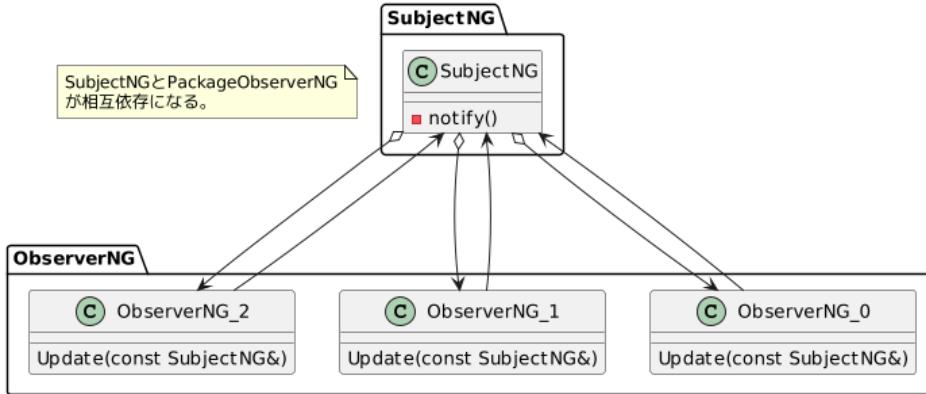
subject.SetNum(1);
subject.SetNum(2);

ASSERT_EQ(2, ng0.call_count);
ASSERT_EQ(2, *ng0.num);

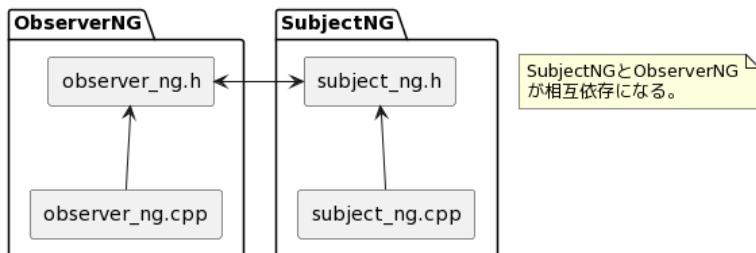
subject.SetNum(2); // 同じ値をセットしたため、Updateは呼ばれないはず
ASSERT_EQ(2, ng0.call_count);
ASSERT_EQ(2, *ng0.num);

```

上記実装例のクラス図を下記する。これを見ればわかるように、クラスSubjectNGとクラスObserverNG_Nは相互依存しており、機能追加、修正が難しいだけではなく、この図の通りにパッケージを分割した場合（パッケージがライブラリとなると前提）、リンクすら難しくなる。



このようなクラス間の依存関係は下記のようにファイル間の依存関係に反映される。このような相互依存は、差分ビルドの長時間化等の問題も引き起こす。



次に、上記にObserverパターンを適用した実装例 (Subjectを抽象クラスにすることもあるが、下記例ではSubjectを具象クラスにしている)を示す。

```

// @@@ example/design_pattern/observer_ok.h 3

/// @class ObserverOK_0
/// @brief SubjectOKからの変更通知をUpdate()で受け取る。
///     Observerパターンの使用例。
class ObserverOK_0 : public Observer {
    ...
private:
    virtual void update(SubjectOK const& subject) override;
};

class ObserverOK_1 : public Observer {
    ...
private:
    virtual void update(SubjectOK const& subject) override;
};

class ObserverOK_2 : public Observer {
    ...
private:
    virtual void update(SubjectOK const& subject) override;
};

```

```

// @@@ example/design_pattern/observer_ok.cpp 5

void ObserverOK_0::update(SubjectOK const& subject)
{
    ...
}

void ObserverOK_1::update(SubjectOK const& subject)
{
    ...
}

void ObserverOK_2::update(SubjectOK const& subject)
{
    ...
}

```

```

// @@@ example/design_pattern/subject_ok.h 8

/// @class SubjectOK
/// @brief 監視されるクラス。SetNumでの状態変更をObserverOK_Nに通知する。

```

```

///          Observerパターンの使用例。
class SubjectOK final {
public:
    SubjectOK() : observers_{}, num_{0} {}

    void SetNum(uint32_t num)
    {
        if (num_ == num) {
            return;
        }

        num_ = num;

        notify(); // subjectが変更されたことをobserverへ通知
    }

    void Attach(Observer& observer);           // Observerの登録
    void Detach(Observer& observer) noexcept; // Observerの登録解除
    uint32_t GetNum() const noexcept { return num_; }

private:
    void notify() const;

    std::list<Observer*> observers_;
    ...
};

/// @class Observer
/// @brief SubjectOKを監視するクラスの基底クラス
class Observer {
public:
    Observer() = default;
    void Update(SubjectOK const& subject) { update(subject); }

    ...
private:
    virtual void update(SubjectOK const& subject) = 0;
    ...
};

```

```

// @@@ example/design_pattern/subject_ok.cpp 3

void SubjectOK::Attach(Observer& observer_to_attach) { observers_.push_back(&observer_to_attach); }

void SubjectOK::Detach(Observer& observer_to_detach) noexcept
{
    observers_.remove_if(
        [&observer_to_detach](Observer* observer) { return &observer_to_detach == observer; });
}

void SubjectOK::notify() const
{
    for (auto observer : observers_) {
        observer->Update(*this);
    }
}

```

```

// @@@ example/design_pattern/observer_ut.cpp 51

struct ObserverOK_Test : Observer { // テスト用クラス
    virtual void update(SubjectOK const& subject) final
    {
        ++call_count;
        num = subject.GetNum();
    }

    uint32_t call_count{0};
    std::optional<uint32_t> num{};

};

auto ok0 = ObserverOK_Test{};
auto ok1 = ObserverOK_1{};
auto ok2 = ObserverOK_2{};

auto subject = SubjectOK{};

subject.Attach(ok0);
subject.Attach(ok1);
subject.Attach(ok2);

ASSERT_EQ(0, ok0.call_count); // まだ何もしていない

```

```

ASSERT_FALSE(ok0.num);

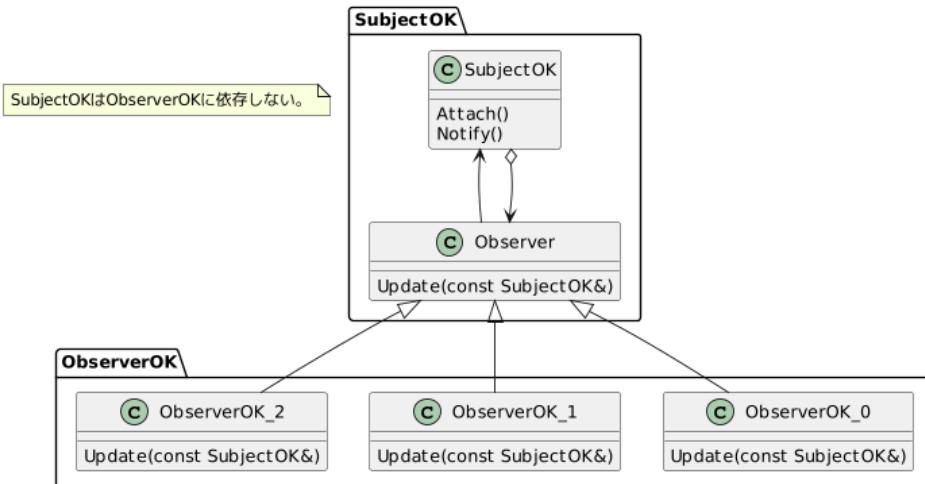
subject.SetNum(1);
subject.SetNum(2);

ASSERT_EQ(2, ok0.call_count);
ASSERT_EQ(2, *ok0.num);

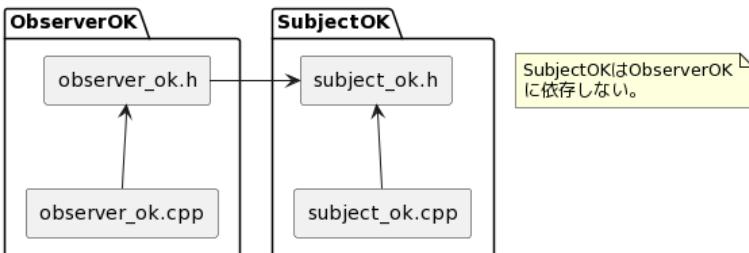
subject.SetNum(2); // 同じ値をセットしたため、Updateは呼ばれないはず
ASSERT_EQ(2, ok0.call_count);
ASSERT_EQ(2, *ok0.num);

```

上記実装例のクラス図を下記する。Observerパターンを使用しない例と比べると、クラスSubjectOKとクラスObserverOK_Nとの相互依存が消えたことがわかる。



最後に、上記のファイルの依存関係を示す。ファイル(パッケージ)の依存関係においてもSubjectOKはObserverOKに依存していないことがわかる(MVCに置き換えると、ModelはViewに依存していない状態であるといえる)。



演習-Observer

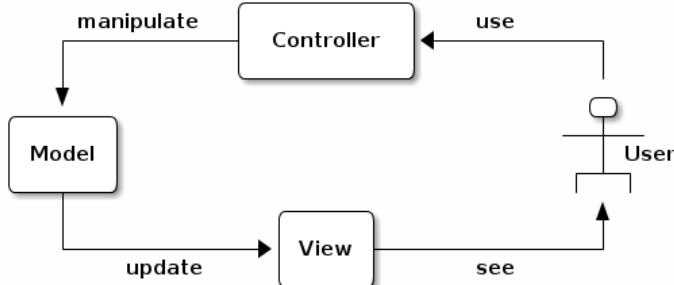
MVC

MVCはデザインパターンと言うよりもアーキテクチャパターンである。一般にGUIアプリケーションのアーキテクチャに使用されるが、外部からの非同期要求を処理するアプリケーションのアーキテクチャにも相性が良い。

MVCのそれぞれのアルファベットの意味は、下記テーブルの通りである。

	MVC	主な役割
M	Model	ビジネスロジックの処理
V	View	UIへの出力
C	Controller	入力をModelへ送信

下記はMVCの概念モデルである(矢印は制御の流れであって、依存関係ではない)。



制御の流れは、

1. ユーザの入力に応じてControllerのメソッドが呼び出される。
2. Controllerのメソッドは、ユーザの入力に応じた引数とともにModelのメソッドを呼び出す。
3. Modelは、それに対応するビジネスロジック等の処理を(通常、非同期に行い、自分自身の状態を変える(変わらないこともある)。
4. Modelの状態変化は、そのModelのオブザーバーとして登録されているViewに通知される。
5. Viewは関連するデータをModelから取得し、それを出力(UIに表示)する。

ViewはModelのObserverであるため、ModelはViewへ依存しない。多々あるMVC派生パターンすべてで、そのような依存関係は存在しない(具体的なパターンの選択はプロジェクトで使用するGUIフレームワークに強く依存する)。

そのようにする理由は下記の通りで、極めて重要な規則である。

- GUIのテストは目で見る必要がある(ことが多い)ため、Viewに自動単体テストを実施することは困難である。一方、ViewがModelに依存しないのであれば、Modelは自動単体テストをすることが可能である。
- 通常、Viewの仕様は不安定で、Modelの仕様は安定しているため、Modelのソースコード変更はViewのそれよりもかなり少ない。しかし、ModelがViewに依存してしまうと、Viewに影響されModelのソースコード変更も多くなる。

演習-デザインパターン選択1

演習-デザインパターン選択2

演習-デザインパターン選択3

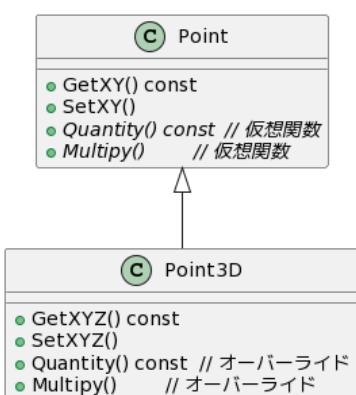
Cでのクラス表現

このドキュメントは、C++でのソフトウェア開発を前提としているため、ここで示したコードもC++で書いているが、

- 何らかの事情でCを使わざるを得ないプログラマがデザインパターンを使用できるようにする
- クラスの理解が曖昧なC++プログラマの理解を助ける(「クラスのレイアウト」参照)

のような目的のためにCでのクラスの実現方法を例示する。

下記のような基底クラスPointとその派生クラスPoint3Dがあった場合、



C++では、Pointのコードは下記のように表すことが一般的である。

```
// @@@ example/design_pattern/class_ut.cpp 7
class Point {
```

```

public:
    explicit Point(int x, int y) noexcept : x_{x}, y_{y} {}
    virtual ~Point() = default;

    void SetXY(int x, int y) noexcept
    {
        x_ = x;
        y_ = y;
    }

    void GetXY(int& x, int& y) const noexcept
    {
        x = x_;
        y = y_;
    }

    virtual int Quantity() const noexcept { return x_ * y_; }

    virtual void Multipy(int m) noexcept
    {
        x_ *= m;
        y_ *= m;
    }

private:
    int x_;
    int y_;
};

```

この単体テストは、下記のようになる。

```

// @@@ example/design_pattern/class_ut.cpp 42

Point a{1, 2};

int x;
int y;
a.GetXY(x, y);
ASSERT_EQ(x, 1);
ASSERT_EQ(y, 2);

a.SetXY(3, 4);

a.GetXY(x, y);
ASSERT_EQ(x, 3);
ASSERT_EQ(y, 4);

ASSERT_EQ(a.Quantity(), 12);

a.Multipy(2);
ASSERT_EQ(a.Quantity(), 48);

```

これをCで表した場合、下記のようになる。

```

// @@@ example/design_pattern/class_ut.cpp 124

struct Point {
    int x;
    int y;

    int (*const Quantity)(Point const* self);
    void (*const Multipy)(Point* self, int m);
};

static int point_quantity(Point const* self) { return self->x * self->y; }

static void point_multipy(Point* self, int m)
{
    self->x *= m;
    self->y *= m;
}

Point Point_Construct(int x, int y)
{
    Point ret = {x, y, point_quantity, point_multipy}; // C言語のつもり
    return ret;
}

void Point_SetXY(Point* self, int x, int y)

```

```

{
    self->x = x;
    self->y = y;
}

void Point_GetXY(Point* self, int* x, int* y)
{
    *x = self->x;
    *y = self->y;
}

```

C++のメンバ関数はプログラマから見えない引数thisを持つ。これを表したもののが各関数の第1引数selfである。また、ポリモーフィックな関数は関数ポインタで、非ポリモーフィックな関数は通常の関数で表される。

この単体テストは、下記のようになる。

```

// @@@ example/design_pattern/class_ut.cpp 164

Point a = Point_Construct(1, 2);

int x;
int y;

Point_GetXY(&a, &x, &y);
ASSERT_EQ(x, 1);
ASSERT_EQ(y, 2);

Point_SetXY(&a, 3, 4);

Point_GetXY(&a, &x, &y);
ASSERT_EQ(x, 3);
ASSERT_EQ(y, 4);

ASSERT_EQ(a.Quantity(&a), 12);

a.Multiply(&a, 2);
ASSERT_EQ(a.Quantity(&a), 48);

```

Pointから派生したクラスPoint3DのC++での実装を以下に示す。

```

// @@@ example/design_pattern/class_ut.cpp 65

class Point3D : public Point {
public:
    explicit Point3D(int x, int y, int z) noexcept : Point{x, y}, z_{z} {}

    void SetXYZ(int x, int y, int z) noexcept
    {
        SetXY(x, y);
        z_ = z;
    }

    void GetXYZ(int& x, int& y, int& z) const noexcept
    {
        GetXY(x, y);
        z = z_;
    }

    virtual int Quantity() const noexcept override { return Point::Quantity() * z_; }

    virtual void Multiply(int m) noexcept override
    {
        Point::Multiply(m);
        z_ *= m;
    }

private:
    int z_;
};

```

この単体テストは、下記のようになる。

```

// @@@ example/design_pattern/class_ut.cpp 98

auto a = Point3D{1, 2, 3};
auto& b = a;

auto x = int{};
auto y = int{};

```

```

b.GetXY(x, y);
ASSERT_EQ(x, 1);
ASSERT_EQ(y, 2);

b.SetXY(3, 4);

b.GetXY(x, y);
ASSERT_EQ(x, 3);
ASSERT_EQ(y, 4);

ASSERT_EQ(b.Quantity(), 36);

b.Multipy(2);
ASSERT_EQ(b.Quantity(), 288);

```

これをCで実装したものが下記である。

```

// @@@ example/design_pattern/class_ut.cpp 188

struct Point3D {
    Point point;
    int z;
};

static int point3d_quantity(Point const* self)
{
    Point3D const* self_derived = (Point3D const*)self;

    return point_quantity(self) * self_derived->z;
}

static void point3d_multipy(Point* self, int m)
{
    point_multipy(self, m);

    Point3D* self_derived = (Point3D*)self;

    self_derived->z *= m;
}

Point3D Point3D_Construct(int x, int y, int z)
{
    Point3D ret{{x, y, point3d_quantity, point3d_multipy}, z};

    return ret;
}

```

この単体テストは、下記のようになる。

```

// @@@ example/design_pattern/class_ut.cpp 221

Point3D a = Point3D_Construct(1, 2, 3);
Point* b = &a.point;

int x;
int y;

Point_GetXY(b, &x, &y);
ASSERT_EQ(x, 1);
ASSERT_EQ(y, 2);

Point_SetXY(b, 3, 4);

Point_GetXY(b, &x, &y);
ASSERT_EQ(x, 3);
ASSERT_EQ(y, 4);

ASSERT_EQ(b->Quantity(b), 36);

b->Multipy(b, 2);
ASSERT_EQ(b->Quantity(b), 288);

```

以上からわかる通り、Cでのクラス実装はC++のものに比べ、

- 記述が多い
- キャストを使わざるを得ない
- リファレンスが使えないため、NULLにならないハンドル変数をポインタにせざるを得ない

等といった問題があるため、「何らかの事情でC++が使えない」チームは、なるべく早い時期にその障害を乗り越えることをお勧めする。

どうしてもその障害を超えない場合は、モダンC言語プログラミングが役に立つだろう。

アーキテクチャ

この章ではソフトウェアアーキテクチャについて考察する。

以下のwikipediaからの引用にあるように、

ソフトウェアアーキテクチャ（英：Software Architecture）は、
ソフトウェアコンポーネント、それらの外部特性、またそれらの相互関係から構成される。
また、この用語はシステムのソフトウェアアーキテクチャの文書化を意味することもある。

… 中略 …

ただし、今までのところ、「ソフトウェアアーキテクチャ」という用語に関して、
万人が合意した厳密な定義は存在しない。

プログラマは、ソフトウェアアーキテクチャ(以下、単にアーキテクチャ)について、ある一定のイメージを持っているが、一元的な概念としては捉えていないと思われる。したがって、アーキテクチャについて語る前に、ここで定義を明確にしておく必要がある。

このドキュメントの内容はアカデミックなものではなく、日々行われるC++での開発の実践に即したものになるように努めてきたため、アーキテクチャの定義もそのようになるべきだろう。

商業的に成功したソフトウェアは開発が継続され、その期間は10年を超えることが珍しくない。そういう事情から、多くのプログラマには開発初期からソフトウェアを作った経験はなく、その結果、開発前段で行われるアーキテクチャへの考察を行った経験もない。

このように考えると、彼らにとってのアーキテクチャは、目の前のソースコードが作り出す構造となるだろう。目の前のソースコードが作り出す構造になるだろう。

この章の構成

アーキテクチャの定義

アーキテクチャの設計

パッケージ図例

シーケンス図例

アーキテクチャとファイル構造

Modelの非同期処理

Viewの非同期処理

アーキテクチャの見直し

パッケージが大きくなりすぎる

当初、想定していない依存関係が必要になる

コードクローンが避けられない

当初、想定していない非同期処理が必要になる

アーキテクチャの再構築

アーキテクチャ再構築の準備

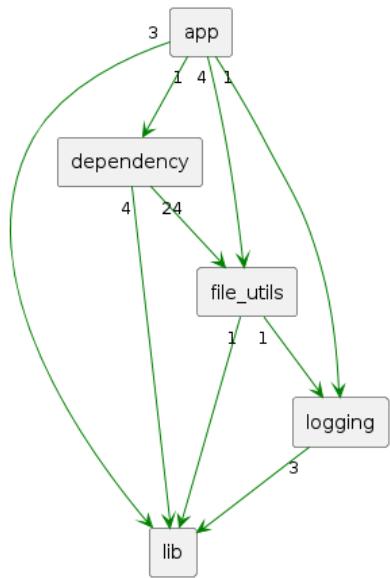
アーキテクチャ再構築のチーム編成

アーキテクチャ再構築の手順

アーキテクチャの定義

以上の考察からここでのアーキテクチャやそれにまつわる概念を以下のように定義する。

- ・アーキテクチャとは「ソースコードをいくつかに分割したパッケージと、それらの依存関係」を指す。
- ・パッケージとは、ソフトウェア構成要素（型、関数、enum、定数、変数など）の集まりを指し、通常、ディレクトリ単位で管理される。パッケージはライブラリ（*.lib、*.dll、*.a、*.so等）を生成することが一般的である。また、サブパッケージを持つこともある。
- ・パッケージの依存関係とは、あるパッケージが別のパッケージの構成要素を使用していること、もしくは使用していないことを指す。従って「パッケージAがパッケージBに依存する」とは、パッケージAがパッケージBの構成要素を使用していることを指す。UMLでは、この依存関係を下記のように表す。



この定義により、アーキテクチャはUMLのパッケージ図/クラス図を使って視覚化できる。

アーキテクチャの設計

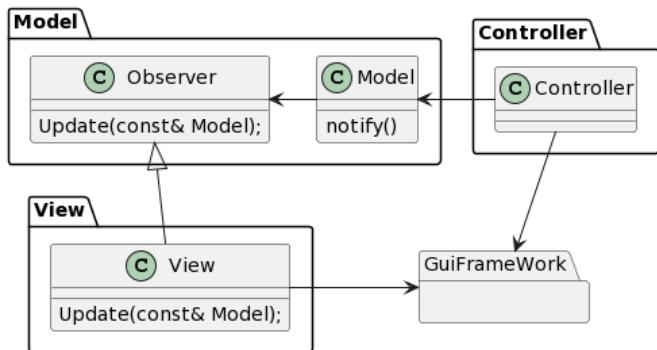
アーキテクチャ設計は、以下のような思考プロセスを繰り返すことで進められる。

1. 要求仕様を理解する。
2. 要求仕様からアーキテクチャに強い影響を与えると予想される下記のような要素について考察する。
 - OS(windows、linux、RTOS、OSを使わない等)は何か?
 - 外部のコンポーネント(GUI等のフレームワークや、データベース等)を使用するか?
 - 並行・並列処理は必要か?
 - 非同期処理はあるか(GUIを含む非同期処理インターフェースを持つアプリケーションにはMVC系のアーキテクチャが適していることが多い)?
 - その非同期処理にアボートや、サスPEND/レジュームは必要か?
3. 非機能要件を掘り起こす。
 - 拡張性、セキュリティのレベル、トラブル解析機能、性能
 - 開発の効率化のためのログの取得機能やモニタリング
 - テストの自動化(「アジャイル系プロセスのプラクティスとインフラ」参照)
4. 下記を考慮したパッケージを定義し、上記の内容を分割して、パッケージに割り当てる。
 - 各パッケージの責務
 - パッケージ間の依存関係
5. 下記のようなシナリオのプロトタイピングをする。
 - なるべく多くのパッケージを使用する含むシナリオ
 - アーキテクチャに強い影響を与える(非同期処理やそのアボート等)シナリオ

この繰り返しの中で、新たな要求仕様や非機能要件が見つかるることはよくあることである。その場合、当然その新規要件も要求仕様書に書き加える。

上記4の成果物として、下記のようなパッケージ図/クラス図を作る。この場合のクラスは、概念を表すためのものであるため、各パッケージの代表的なもののみを記述すればよい。

パッケージ図例



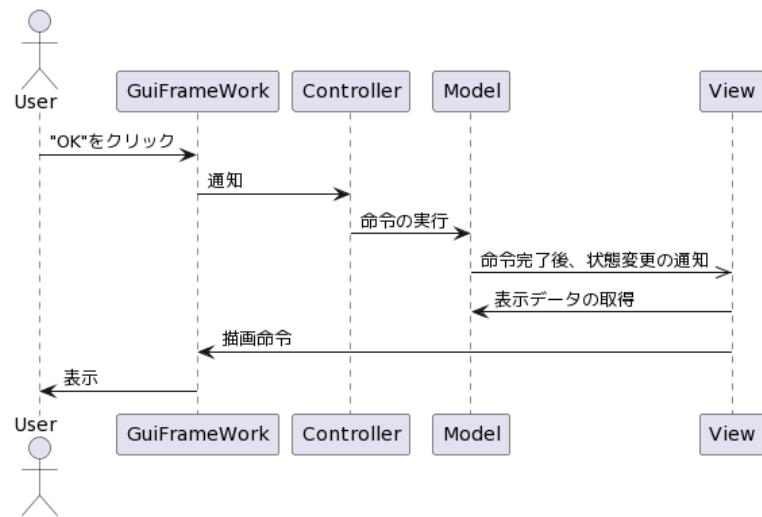
この時点で、パッケージ間に相互依存や循環依存があれば、まず間違いなくそのアーキテクチャは使い物にならないため再考する。

コンウェイの法則で述べたように、「組織に属する者はその組織構造を投射したアーキテクチャが正しいと思ってしまうバイアスを持つ」ことに注意することも必要である。

このフェーズでパッケージの名前が決定されるが、適切な名前を選ぶことは大変難しい。最適と思えるものが思い浮かばなければ、後から修正することを前提に適当な名前を付けることがベストな戦略となり得るが、パッケージの概念が固まっていない証拠ともなり得るため、難しい判断が求められる。

上記5の成果物として、プロトタイピングで使用したシナリオを表す下記のような概念的なシーケンス図やアクティビティ図等を作る。

シーケンス図例



上記のようなダイアグラムは、

- 繰り返し修正する
- 概念を表すことが目的である

ため、

- 詳細な物を作るのは時間の無駄である。
- 修正前後の違いが簡単にわかるものでなければならぬ(修正前後の違いがdiffで簡単に表示できるため、当ドキュメントでのダイアグラムの記述には、[plant uml](#)を使用している)。

このようにしなければ、ダイアグラムのドローイング作業は無限に工数を吸収する沼となるだろう。

上記のステップを複数回、試行することで、いくつかのプロトタイプコードができる。この試行はアジャイル系プロセスと相性が良い。一方で、ウォーターフォールモデル、V字モデルや、プロセスを決めない試行は、沼にはまり込み大きく時間をロスする可能性が高い。

[deps](#)等のリバースエンジニアリングツールを使用し、プロトタイプコードから生成したパッケージ図/クラス図が、上記5の最終版のパッケージ図/クラス図と一致するのであれば、一旦アーキテクチャの設計は終了し次のフェーズに進む。

アーキテクチャとファイル構造

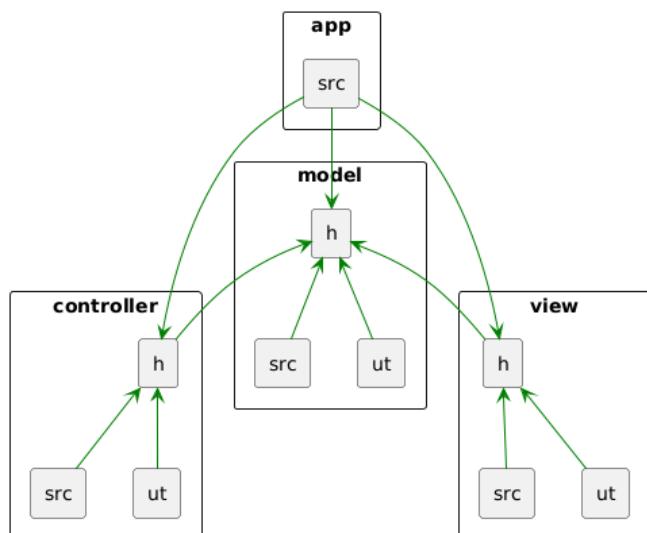
プロトタイピングで開発したコードやビルドツールの設定を開発の起点にするためには、下記のような、もうひと手間が必要である。

- プロトタイプコードが「パッケージとその構成ファイル」で述べた規則に沿うように修正する。
- パッケージをライブラリ(*.lib、*.dll、*.a、*.so等)としてビルドできるように、make等のビルドツールを修正する。
- パッケージから生成されたライブラリに対する単体テストを作る。単体テストは各パッケージごとに実行形式ファイルを生成できるようビルドツールを修正する。これにより自動単体テストが実行できるようになる。

この作業の完了時、パッケージ図は下記のようになっているだろう。

```
architecture
├── CMakeLists.txt
└── app
    └── main.cpp
└── controller
    ├── CMakeLists.txt          # controller.aのビルドcmake
    ├── h
    │   └── controller
    │       └── controller.h    # controller.aの機能の公開ヘッダ
    ├── src
    │   └── controller.cpp      # controller.aの実装ファイル(*.h *.cpp)
    └── ut
        └── controller_ut.cpp  # controller.aの単体テスト
└── model
    ├── CMakeLists.txt          # model.aのビルドcmake
    ├── h
    │   └── model
    │       └── model.h         # model.aの機能の公開ヘッダ
    ├── src
    │   └── model.cpp           # model.aの実装ファイル(*.h *.cpp)
    └── ut
        └── model_ut.cpp        # model.aの単体テスト
└── view
    ├── CMakeLists.txt          # view.aのビルドcmake
    ├── h
    │   └── view
    │       └── view.h          # view.aの機能の公開ヘッダ
    ├── src
    │   └── view.cpp             # view.aの実装ファイル(*.h *.cpp)
    └── ut
        └── view_ut.cpp          # view.aの単体テスト
```

h</パッケージ名>に配置されたヘッダファイルは、パッケージの外部からアクセスできるソフトウェア構成物を宣言、定義する。その他に配置されたヘッダファイルは、パッケージ自身の実装用か、単体テスト用である。ここで例示したアプリケーションはMVC構造であるため、ディレクトリの依存関係は、下記の様になるはずである。



循環や相互依存が残ってしまう場合、「SOLID」に記載したコードのパターンや「デザインパターン」が役立つはずである。

#includeで指定するパス名でのルールに従うことで、パッケージの依存関係は、

```
// @@@ example/architecture/model/src/model.cpp 6

#include <iostream> // stdの使用

#include "./x.h" // ローカルヘッダの使用
#include "logging/logger.h" // logger.aの使用
#include "model/model.h" // model.aの使用
```

のようにコードに投影されるため、メンテナンス性や可読性が向上する。

従って、各ライブラリのビルド毎にインクルードパスを設定できないようなビルドツールやIDEを使うべきではない（「[エディタ/IDE](#)」参照）。

Modelの非同期処理

前記したパッケージ図例、シーケンス図例で示した通り、GUIのボタン押下などによるControllerの呼び出しから、呼び出されるModelオブジェクトのメンバ関数は非同期処理となることが求められることが多い。このため、Modelクラスの構造はアクティブオブジェクトを生成できるようにスレッドを内包することになる。こういった構造は定型となるため、そのコードを以下に例示する。

```
// @@@ example/architecture/model/h/model/model.h 15

class Model {
public:
    class Observer {
    public:
        virtual void Update(Model const& model) = 0;
        virtual ~Observer() = default;
    };

    struct msg_t {
        msg_t() : exec([] {}) {}
        msg_t(std::function<void()> exec) : exec{std::move(exec)} {}
        std::function<void()> exec;
    };

    Model() : worker_{&Model::worker_function, this} {}
    ~Model();

    bool ExecAsync(std::function<void()> exec); // 非同期リクエスト
    bool IsBusy() const noexcept { return busy_; }

    void Sync(); // 非同期要求の完了待ち
    void Attach(std::unique_ptr<Observer>&& observer);

private:
    std::thread worker_; // 非同期処理を実現するためのワーカスレッド
    std::atomic<bool> busy_ = false; // ExecAsyncを受け付けるか否か
    std::atomic<bool> stop_ = false; // worker_functionの終了変数
    void worker_function(); // スレッドのメイン関数。msg_cv_でウェイト
    void notify(); // observer::Updateの呼び出し

    std::list<Model::msg_t> msgs_{}; // 非同期要求はmsg_tとしてリスト化される
    std::mutex msg_mtx_{}; // リスト処理の競合の保護
    std::condition_variable msg_cv_{}; // msgs_に追加されたことの通知

    std::list<std::unique_ptr<Model::Observer>> observers_{};
};
```

Modelクラスに[Pimpl](#)を適用することでこのクラスの内部構造を隠蔽した方が良い場合もあるが、ここでは例示するコードを単純にすることを優先する。

非同期処理のためのコードを以下に示す。

```
// @@@ example/architecture/model/src/model.cpp 34

bool Model::ExecAsync(std::function<void()> exec) // Modelに対する非同期要求
{
    // 非同期要求のキューイングはしないが、キューイング可能に変更は容易
    if (busy_) {
        return false;
    }

    {
        std::unique_lock<std::mutex> lock{msg_mtx_};
        msgs_.emplace_back(std::move(exec));
    }
}
```

```

        busy_ = true;
    }
    msg_cv_.notify_one();

    return true;
}

// @@@ example/architecture/model/src/model.cpp 53

void Model::worker_function() // スレッドのメイン関数
{
    for (;;) {
        msg_t msg;
        {
            std::unique_lock<std::mutex> lock{msg_mtx_};
            msg_cv_.wait(lock, [&msgs_ = msgs_, &stop_ = stop_] { return !msgs_.empty() || stop; });
            if (stop_ && msgs_.empty()) {
                return;
            }

            msg = std::move(msgs_.front());
            msgs_.pop_front();
        }

        msg.exec(); // ExecAsync(exec)で渡された関数オブジェクトの非同期呼び出し
        busy_ = false;

        notify(); // オブザーバーへの通知処理
    }
}

```

```

// @@@ example/architecture/model/src/model.cpp 77

void Model::Attach(std::unique_ptr<Observer>&& observer) // オブザーバーのアタッチ
{
    observers_.emplace_back(std::move(observer));
}

void Model::notify()
{
    for (auto const& observer : observers_) {
        observer->Update(*this);
    }
}

```

以下に、単体テストによりModelの動作を示す。

```

// @@@ example/architecture/model/ut/model_ut.cpp 17

class TestObserver : public Model::Observer { // テスト用オブザーバー
public:
    void Update(const Model& model) override { ++update_counter_; }
    std::uint32_t update_counter_ = 0;
};

// @@@ example/architecture/model/ut/model_ut.cpp 28

Model model{};
int exec_counter{};
TestObserver* to = new TestObserver; // 下のunique_ptrで管理

model.Attach(std::unique_ptr<TestObserver>{to}); // オブザーバの登録

ASSERT_FALSE(model.IsBusy()); // ビジーでないことの確認

model.ExecAsync([&exec_counter]{}); // 非同期要求のテスト開始
++exec_counter;
std::this_thread::sleep_for(std::chrono::milliseconds(100));
LOGGER("in ExecAsync");
});

ASSERT_TRUE(model.IsBusy()); // ラムダ内で100ms待つため、ビジーとなる
ASSERT_EQ(to->update_counter_, 0); // まだラムダが実行されていないはず

ASSERT_FALSE(model.ExecAsync([&exec_counter]{})); // まだラムダが実行されていないはず
++exec_counter;
std::this_thread::sleep_for(std::chrono::milliseconds(100));
LOGGER("in ExecAsync");
));

model.Sync(); // 非同期要求の完了待ち

```

```

ASSERT_EQ(exec_counter, 1);
ASSERT_EQ(to->update_counter_, 1); // オブザーバーのUpdateが呼ばれたことの確認

model.ExecAsync([&exec_counter](){
    ++exec_counter;
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    LOGGER("in ExecAsync");
});

```

以上からわかる通り、Modelはインスタンスごとにスレッドを持つため、Observer::Updateはメインスレッドとは別々の様々なスレッドから呼び出されることになる。

Viewの非同期処理

Observerデザインパターンでは、Observerが監視しているSubjectの状態に変更があった場合、Subject::notifyが呼び出され、その延長でObserver::Updateが呼び出されるような構造を持つ。Subjectの状態変更や、Subjectのメンバ関数の戻り値をObserver::Updateの処理の一部として、GUIや標準出力の更新をすることが一般的である。

MVCのような構造を持つアプリケーションでは、ModelオブジェクトやViewオブジェクトを複数必要とする。通常のViewオブジェクトは、GUIや標準出力はアプリケーション毎に唯一存在するため、これら表示用リソースは複数のViewオブジェクトに共有されることになる。

以上の考察から明らかになったViewオブジェクトの制約を以下にまとめる。

- Viewオブジェクト複数、生成される。
- Viewオブジェクトは出力用リソースを共有する。
- View::Updateは様々なスレッドから呼び出される。

この結果、Viewオブジェクトに共有される出力用リソースへのアクセスは、それを防ぐための特別な構造を持たないと競合を起こしてしまう。競合を防ぐためにロックを多用するとコードが複雑になってしまふため、出力用リソースへのアクセスは、1つのスレッドにすることが競合を防ぐための最もシンプルな解になることが多い。

このようなクラス構造は、Viewの非同期処理の例で示したクラス内と同じような構造となることが多いが、また、出力リソースがアプリケーションに唯一であることから、出力を受け持つクラスを、Singletonにすることが理にかなっているだろう。

これまでの考察から明らかになったViewの典型的なコードを以下に例示する。

```

// @@@ example/architecture/view/h/view/view.h 11

class ViewCore { // すべてのviewから保持される非同期出力オブジェクトを生成するためのシングルトン
public:
    static ViewCore& Inst()
    {
        static ViewCore inst;
        return inst;
    }

    void ShowAsync(std::string&& msg); // 非同期出力
    void Sync(); // 非同期出力の同期待ち

    void SetOStream(std::ostream& ostream) { ostream_ = &ostream; } // テスト用出力切り替え

private:
    ViewCore(const ViewCore&) = delete;
    ViewCore(ViewCore&&) = delete;
    ViewCore() : ostream_{&std::cout}, worker_{&ViewCore::worker_function, this} {}
    ~ViewCore();

    std::thread worker_; // 非出力を実現するためのワーカスレッド
    bool busy_ = false; // 非出力完了待ちに使用
    std::atomic<bool> stop_ = false; // worker_functionの終了変数

    void show_msg(std::string const& msg) // 非同期にmsgを出力
    {
        *ostream_ << msg;
        busy_ = false;
    }
    void worker_function();

    std::list<std::string> msgs_{};
    std::condition_variable msg_cv_{};
    std::mutex msg_mtx_{};

```

```

        std::ostream* ostream_;
};

// @@@ example/architecture/view/h/view/view.h 51

class View : public Model::Observer {
public:
    View() : view_core_{ViewCore::Inst()} {}
    ~View() = default;
    void ShowAsync(std::string&& msg) { view_core_.ShowAsync(std::move(msg)); }
    void Sync() { view_core_.Sync(); }
    void Update(Model const& model) override { view_core_.ShowAsync("View updated"); }

private:
    ViewCore& view_core_; // すべての出力をViewCoreに委譲
};

// @@@ example/architecture/view/src/view.cpp 16

void ViewCore::ShowAsync(std::string&& msg) // 非同期出力
{
{
    std::unique_lock<std::mutex> lock{msg_mtx_};
    msgs_.push_back(std::move(msg));
    busy_ = true;
}
msg_cv_.notify_one();
}

void ViewCore::worker_function()
{
    for (;;) {
    {
        std::unique_lock<std::mutex> lock{msg_mtx_};
        msg_cv_.wait(lock, [&msgs_ = msgs_, &stop_ = stop_] { return !msgs_.empty() || stop; });
        if (stop_ && msgs_.empty()) {
            return;
        }
        std::string msg = std::move(msgs_.front());

        LOGGER("Processing message", msg, ":busy", b2str(busy_));
        msgs_.pop_front();
        show_msg(msg);
    }
}
}

void ViewCore::Sync()
{
    for (;;) {
        if (!busy_) { // busy_のポーリング
            return;
        }

        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
}

ViewCore::~ViewCore()
{
    stop_ = true;
    msg_cv_.notify_one();
    worker_.join();
}

```

以下の単体テストでviewの使用例を以下に示す。

```

// @@@ example/architecture/view/ut/view_ut.cpp 27

std::ostringstream out;
View         view{};

ViewCore::Inst().SetOStream(out); // 出力の切り替え
const auto* str = "Output string to View";

view.ShowAsync(str); // 非同期出力
view.Sync();        // 出力待ち
ASSERT_EQ(out.str(), str);

```

アーキテクチャの見直し

ソフトウェアの成長に伴いアーキテクチャに下記のような綻びが発見されることは珍しいことではない。このような綻びがプロジェクトの破綻を引き起こす前に手を入れなければならないことは言うまでもない。

- ・パッケージが大きくなりすぎる
- ・当初、想定していない依存関係が必要になる
- ・コードクローンが避けられない
- ・当初、想定していない非同期処理が必要になる

パッケージが大きくなりすぎる

初期のアーキテクチャのパッケージ分割が不十分であることはよくあることである。その場合、大きくなつた、もしくは間違いなく大きくなると想定されるパッケージに対しては、「アーキテクチャの設計」の手順を適用し、そのパッケージを分割する。

分割されたパッケージは、一旦、元のパッケージのサブパッケージとするが、そのサブパッケージの中で、外部のパッケージから使用される機会の多いものや、元のパッケージとの関係が少ないものは、元のパッケージから出し、単独のパッケージをするべきだろう。

こういったリファクタリングをその効果に見合った工数で行うためには、「開発プロセスとインフラ」で述べたように自動単体テストや自動統合テストが必要となる。

自動単体テストや自動統合テストがない場合、このような修正にはデグレードが多発するため多くの工数ロスは避けがたいが、放置すれば状態は、より悪化する。

当初、想定していない依存関係が必要になる

初期のアーキテクチャの依存関係は、その設計に用いたシナリオが必要とするもののみとなっているため、ソフトウェアの成長に伴い新たな依存関係が必要になることは当然である。このような場合、下記のようなことに気を付けて新たな依存関係を追加すべきだろう。

- ・不要な依存関係を作らない。
- ・循環、相互依存を作らない。
- ・依存関係逆転の原則(DIP)に反した依存関係を作らない。

すでに述べたように、循環や相互依存の解消には、「SOLID」に記載したコードのパターンや「デザインパターン」が有用である。

コードクローンが避けられない

コードクローンの原因はいくつも存在するが、その対処方法は常に、一連のコードクローンを一つの関数やクラスにして、適切な場所に配置することである。

例えばある基底クラスから派生したクラス群のいくつかがほぼ処理の等しいメンバ関数を持つのであれば、そのメンバ関数を統一して、基底クラスに移動すれば良い。

この方法と同様に、ほぼ処理の等しい関数やクラスが複数のパッケージに存在し、それらが「Nstdライブラリの開発」で述べたような汎用的なものであれば、プロジェクト全域からアクセスを許可するパッケージを用意し、そこに配置すればよい。

汎用的ではない場合、おそらくパッケージの分割が不十分であったために、具現化されていないパッケージの処理が複数のパッケージに散乱していることが考えられる。具現化されていないパッケージを具現化するために、「アーキテクチャの設計」の手順を再実行すべきだろう。

当初、想定していない非同期処理が必要になる

「リファクタリングの例」で述べたように、同期処理を前提としたソフトウェアに非同期処理を追加すると、ソースコードは腐敗を始める。

以下のような事項が腐敗の原因となる。

- ・非同期処理のプログラミングは、同期処理のプログラミングよりもかなり難しい。
 - マルチスレッド化が必要になる。
 - マルチスレッド化には競合回避が必要になる。
 - マルチスレッドのデバッグは難しいため、リファクタリングをしなくなる。
 - プロジェクトで使用していなかった新たなシステムコールを使用しなければならなくなる。
- ・今までとは逆向きの依存関係が必要になる。

これらに関しては、

- マルチスレッドで必要なシステムコールを熟知する
- MVC系の構造を取り入れる

ここで対処できる(「並行処理」参照)が、レベルの高いプログラマの工数をかなり消費するため、避けるべきアーキテクチャ変更である。初期のアーキテクチャの設計時に、非同期処理が不要であることに対して、絶対の自信がないのであれば、非同期処理が必要であるという前提で設計すべきだろう。

アーキテクチャの再構築

多くのプログラマが、日々、スパゲティコードに向かい合い戦闘している。そうなった理由は以下のようなことが原因となっている。

1. ソフトウェア開発プロセス(「開発プロセスとインフラ」参照)の未成熟
2. アーキテクチャの不備
3. プログラマのスキル不足

このような状況を改善・改革するための活動をここでは「アーキテクチャの再構築」と呼ぶことにする。

このような環境の中でのプログラマは、以下のいずれかの行動を選択する。

- その組織から離れる。
- その状態を受け入れる。
- 「アーキテクチャの再構築」に向けた活動を始める。

ここで議論は、最後の選択をするプログラマのためのものである。以下では、このようなプログラマを改善プログラマと呼ぶ。

改善プログラマは、その行動様式から原因1、2、3の改善を行おうとするだろうが、このプログラマがチームの運営方針や予算に影響を与えることができる立場でないのであれば、原因1、2の対処は出来ないため、原因3の対処のみに注力する以外の改善策はない。

アーキテクチャ再構築の準備

上記のような議論から改善プログラマは以下のようなことをすることになる。

- アーキテクチャの再構築に必要と思われる下記のようなスキルの獲得や向上
 - プログラミング
 - UML、オブジェクト指向、デザインパターン
 - CI(継続的インテグレーション)のための知識や言語
- 周りのプログラマのスキル向上
- 今よりもコード品質が悪化しないための施策
例えば、「機能追加時にその機能のみに単体テストを導入する」ことや「jenkinsによる自動ビルドを行う」こと等は、コードのレベルを上げる効果のみにとどまらず、アーキテクチャの再構築のための知識習得に役立つ。

このような改善活動を行いながら、ソフトウェアを大きく変えざるを得ないようなプロジェクトが企画されることを待つことになるが、時間が来た時に組織に影響を与えられる立場にいなければならぬ。従って、改善プログラマにはそのような立場へのプロモーションも必要となる。

以上をまとめると、スパゲティコードの改善には以下のようないくつかの条件が必要になる。

- 改善プログラマがアーキテクチャの再構築に必要なスキルを獲得する。
- 改善プログラマが組織に影響を与えられる立場を獲得する(ここでの立場とは、公式なもののみではなく、権限者の意思決定に影響を与える非公式なものでもよい)。
- 何人かのプログラマが改善プログラマのフォロワーとなり、スキルの改善をする。
- ソフトウェアを大きく変えざるを得ないようなプロジェクトが企画される。

アーキテクチャ再構築のチーム編成

ソフトウェア開発プロセス、アーキテクチャ設計、プログラミングに対する知識は、良書を読み、それを実践することで始めて身につく類のものであるため、残念なことであるが、日々行うプログラミングを除いて、アーキテクチャ再構築のための十分な知識をチームが身に着けることは難しい。ソフトウェア開発プロセスやアーキテクチャ設計の知識は外部から調達せざるを得ないため、スパゲティコードを立て直すプロジェクトが開始できる目途が付けば、そのようなコンサルティング業者を探さなければならない。こういったことを行える業者の数は多くはないが、見つけることは不可能ではない。運よく見つかった場合でも、その業者が派遣するコンサルタントを何の疑いもなく受け入れてはならない。コンサルタントのスキルレベルを問い合わせてみる必要がある。間違いなく必要なのは、ソフトウェア開発プロセス導入や

アーキテクチャ設計の実績である。多くのエビデンスはないが、ウォーターフォールモデル、V字モデルのみを提案するコンサルタントや、アジャイル系プロセスを薦めながらテストの自動化を提案しないコンサルタントは、十分な知識を持っていないと判断してよいと思う。

良いコンサルタントが見つからない場合、改善プログラマが中心になり、ソフトウェア開発プロセスの導入やアーキテクチャの再設計を行うことになる。良いコンサルタントが見つかった場合でも、そのコンサルタントがこれから再構築するソフトウェアの要求仕様を熟知していることはないので、アーキテクチャ設計を完全に委託することはできず、結局、アーキテクチャ設計は自力で行わなければならない。ということで、仮にコンサルタントが見つからなかったとしても、レビューアーが足りない程度のインパクトである(と思う他ない)。

様々なな障害があるだろうが、兎に角それらを乗り越えて、改善プロジェクトが開始できたとしよう。コンサルタントが見つかった場合の初期のチーム構成は以下の様になる。

人員	ロール
改善プログラマ	リーダ、アーキテクト
数人のフォロワープログラマ	UMLダイヤグラムやプロトタイプコード等の開発者
コンサルタント	知見提供、UMLダイヤグラムやプロトタイプコード等のレビューアー

コンサルタントが見つからなかった場合、他のメンバーにこのロールを振り分けることになる。助っ人なしではやや不安だろうが、時には奮勇も必要である。改善に向けて踏み出そう。

アーキテクチャ再構築の手順

アーキテクチャの再構築は、

1. 現在のソースコードをベースに再構築するかどうかを判断する
2. パッケージ間の依存関係を整理する
3. 依存関係が整理されたパッケージ毎に単体テストを作る
4. 統合テストプログラムを作る
5. 単体テストや統合テストを自動実行できる環境を整える
6. リファクタリングを行う

のような手順で行うことになる。

この一連の活動は、実際には上から順次実行できるわけではなく、反復する必要がある。また、この活動後のチーム全体へのプロセスの導入にもスムーズに移行できるため、アジャイル系プロセスを選択するべきである。

このプロセスの定義には、「開発プロセスとインフラ」や、「プロセス・プラクティス」で紹介した書籍が参考になるだろう(改善プログラマならば、この時点でのドキュメントには精通しているはずである)。

具体的なプロセスを選択した後は、上記の作業を細分化し、プロセスに合わせこむ必要があるが、この作業は一回のミーティングで完了できる。後から多少の不備が見つかるだろうが、その都度、見直せばよい。多くのアジャイル系プロセスには、そのための振り返りミーティングが設けられている。

現在のソースコードをベースに再構築するかどうかを判断する

この活動の目的は、今のソースコードと同じ動作をする、きれいなソースコードを作ることであり、リファクタリングがキーファクターとなる。リファクタリングを行うためには、単体テストが不可欠であるため、新アーキテクチャには、そのための構造が必要になる。

従って、現在のソースコードをベースに再構築するかどうかを判断するためには、現在のパッケージの依存関係が循環していないか、循環していた場合、現実的な工数とデグレードリスクで「アーキテクチャとファイル構造」で示したクラス図のように修正できるかを調査する必要がある。

循環が修正できない場合、現在のソースコードをベースにすることはできない。この場合、古いコードの修正はあきらめ、新しいコードをスクラッチから作ることになるため、手順は「要求仕様を理解する」ことが不要な「アーキテクチャの設計」となる。

パッケージ間の依存関係を整理する

パッケージの依存関係が循環していないかった場合、このフェーズでやることはない。

循環を修正する場合、「SOLIDで示したコードパターン」や「デザインパターン」が役に立つだろう。

修正が完了した時点で、手作業による簡単な統合テストを行い、クリティカルなデグレードを修正する。

依存関係が整理されたパッケージ毎に単体テストを作る

「アーキテクチャとファイル構造」で述べたようにビルド環境を整える。

これで単体テストを記述するための環境はできることになる。これを使い、各パッケージのいくつかのクラスや関数の単体テストを書く。

この時点で、プロジェクトの起点となるソースコードが出来ているはずである。

メンバを増やし、彼らに単体テストのカバレッジを上げる仕事をさせることになる。新規メンバには、このプロジェクトのプロセスに従つてもらい、「TDD」で述べたようなワークフローで作業してもらうのが理にかなっている。もし、コンサルタントが雇えていないのであれば、このタイミングでアジャイル系プロセスの導入をサポートしてくれるコンサルタントを迎えて入れても良い。

独力で行うにしても、助成を頼むにしても、増員後のチームへのプロセスの導入には細心の注意が必要である。細かく規定すぎると、「鳥網、精緻にして一鳥もかからず」の例えにある通り、誰もそのルールには従わなくなる。逆に、自然に任せれば、せっかく作り上げたアーキテクチャは破壊される。

統合テストプログラムを作る

上記の手順により、ある程度整理されたソースコードに対し、自動統合テスト用プログラムを開発する。GUI系のアプリケーションでは、GUIオブジェクト(ボタンやテキストボックス等)を直接操作する統合テストの開発は困難であるが、その場合は、「アーキテクチャとファイル構造」で示したmodel部分を評価する統合テストを開発する。

統合テストの開発のために、アーキテクチャ再構築中のソフトウェアに新機能の追加(「自動統合テストのための仕様追加」参照)や、さらなるアーキテクチャ変更が発生することがあるが、多くの場合ここで妥協すべきではない。

単体テストや統合テストを自動実行できる環境を整える

上記までで用意した単体テストと統合テストをバージョン管理システムと連動させて自動実行するための環境を整える。

典型的には「CI(継続的インテグレーション)」で述べたように、

- テストの自動実行を行うツールはJenkins
- バージョン管理システムはgitかgitのクラウドサービス

を使うことになるだろう。この環境により、例えばgit pushが行われるたびに単体テストと統合テストが自動実行される。これに合わせて「コード解析」で述べたような解析ツールを導入すると、この後行うリファクタリングが捲る。

リファクタリングを行う

上記までの開発でリファクタリングを行う環境は整ったことになるため、リファクタリングを開始する。

初期のリファクタリングは、

- 「パッケージ間の依存関係を整理する」フェーズで修正しきれなかった循環依存の修正
- 巨大なファイルの分割
- サイクロマティック複雑度の値が高い関数の分割
- 巨大なクラスや、凝集度の低いクラス分割

になる。依存関係の整理(循環依存や不要インクルード)は、コンパイル時間短縮にも効果があるため、なるべく早い時期に取り掛かるのが良いだろう。

まとめ

アーキテクチャ(もしくはソースコード)とプロセスは共生関係にあり、どちらか一方のみを改善することはできないため、アーキテクチャの再構築には、これまで述べたように

- コードの整理
- プロセスの整備
- 開発インフラの整備、開発

等、多岐にわたる知識が必要なため難易度が高い。また、これによりチーム活動にも多大なインパクトを与えるため苦渋に満ちたものになるだろう。従って、このような活動はプロダクトライフタイムで最多でも一回に留めるべきであることは言うまでもない。

開発プロセスとインフラ

ソフトウェア開発を効率よく行うためには、以下の三要素をレベル高く保つことが重要である。

- プログラマ
- 開発プロセス(以下、単にプロセスと呼ぶ)
- 開発をサポートするためのインフラ(以下、単にインフラと呼ぶ)

この章では後ろ2つの要素(プロセスとそれを支えるインフラ)について、アジャイル、CIに軸足を置いて説明する。

この章の構成

プロセス

ウォーターフォールモデル、V字モデル

アジャイル系プロセス

ウォーターフォール vs アジャイル

アジャイル系プロセスのプラクティスとインフラ

自動単体テスト

リファクタリング

自動統合テスト

TDD

CI(継続的インテグレーション)

まとめ

プロセス

本ドキュメントでは、プロセスを下記の3つに分類する。

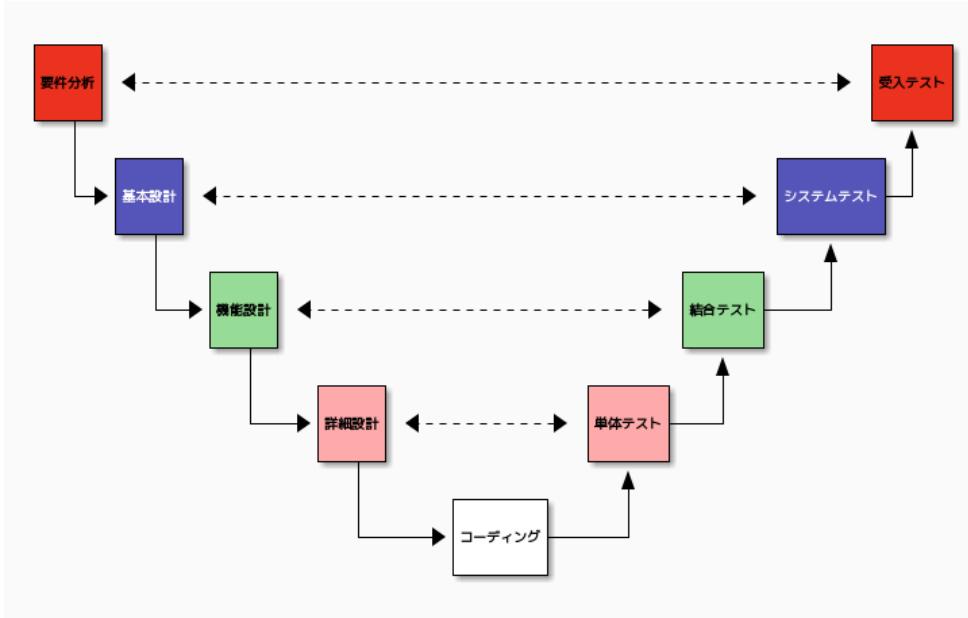
- ウォーターフォールモデル、V字モデル
- 反復型
- アジャイル系プロセス

上から順に初期計画順守的であり、逆に下から順に状況適応的である。状況適応的であることは、無計画であることを意味しない。ただ単にプライオリティの問題として、計画に従うことより状況に適応・対処することを選択するということである。

ほとんどのアジャイル系のプロセスは、繰り返し構造を持つため、「アジャイル ⊑ 反復型」と分類されることもある(が、本ドキュメントではそうしない)。

ウォーターフォールモデル、V字モデル

ウォーターフォールモデルもしくはV字モデルと呼ばれるプロセスでは、「要件分析」→「基本設計」→「機能設計」→「詳細設計」→「プログラミング」といった工程でソフトウェアを作り、その後「単体テスト(UT)」→「結合テスト(IT)」→「システムテスト」→「受入テスト(運用テスト)」といった工程でテストを行う。



設計・開発の各フェーズ(上図の左側)は、同じ高さにあるテストの各フェーズ(上図の右側)にそれぞれ対応する。ソフトウェアの機能はそれが定義された設計・開発フェーズに対するテストフェーズで評価される。

設計・開発フェーズではトップダウンで作業を行うことで実装漏れや手戻りを防ぎ、テストフェーズではその逆にボトムアップで作業を行うことにより、細かいバグによる全体進捗の妨げを防ぐことを意図している。

長い歴史を持った手法であるため、安定したプロセスであるが、下記するような問題を持っている。

- プロダクトへの学習が進んでないプロジェクト初期に、下記のようなリスクの高い意思決定をせざるを得ない。
 - プロジェクトの計画
 - アーキテクチャの設計
- 設計・開発フェーズでの手戻りを無くすために多くのレビューを繰り返すが、多くの仕様上のバグは実装時に発見される(実装することで仕様バグは発見しやすくなる)。
- 設計・開発の各フェーズで一定以上のバグが見つかった場合、そのフェーズを中止し、前のフェーズに戻らなければならない。これにはコストがかかりすぎるため、このルールの順守は容易ではない。
- 自然言語やコンピュータ言語、数式等を使って以下のようなソフトウェアを仕様化することや、その仕様書をレビューすることは困難である。
 - ルック&フィールが重要なリッチなGUI
 - トライ&エラーを繰り返しながら開発するアルゴリズム
- ソフトウェア開発は短くても数か月必要であるため、その間に要求仕様が不变であることは稀であり、開発途中での仕様変更は避けがたいが、このプロセスに従った要求仕様の変更、追加はコストがかかりすぎるため現実的には不可能である。
- 要求仕様の変更、追加が困難であるため、プロジェクトの初期に必要性の不確かな機能が大量に要求される。
- 実装が終わるのは通常全体日程の7割～8割を消化した頃である。その時期まで開発の進捗や品質はわからない。わかるのは「消費した工数」と、「希望的観測により作られた進捗率」である。

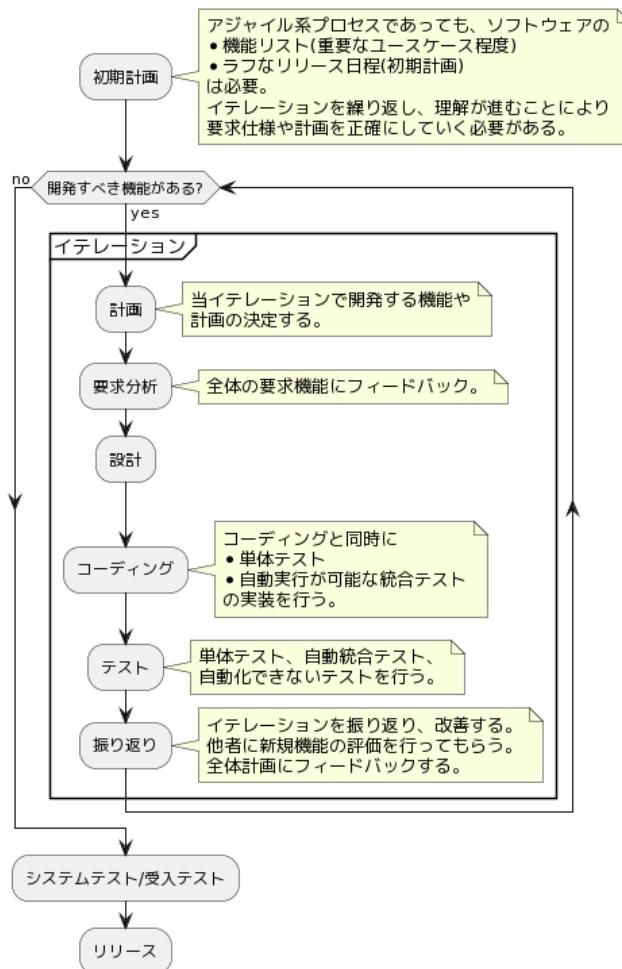
アジャイル系プロセス

アジャイル系プロセスとは、敏捷(=agile)かつ適応的にソフトウェア開発を行う軽量な開発手法群の総称である。

以前、アジャイルが意味するものは誤用・乱用され、今もってその状態が解消されたとは言い難いため、あえて下記の通り注意喚起する。

- 無計画なソフトウェア開発はアジャイルではない。
- カウボイコーディングはアジャイルでない。
- 要求変更を受け入れるだけではアジャイルではない。
- アジャイルは、フレデリック・ブルックスがいうところの「銀の弾」ではない。

英単語の「agile(アジャイル)」には「繰り返し」という意味は含まれていないが、実際には、その代表格であるスクラムやXPを含め、ほとんどのアジャイル系プロセスは、下記のような繰り返し(イテレーション)構造を持つ。



こういったプロセスは、ウォーターフォールが持つ欠点への反省と克服のために作られた(と言って良い)ため、多くのウォーターフォールの欠点を軽減、回避しているが、一方でウォーターフォールが持っていない下記のような欠点を持っている。

- 比較的小さいプロジェクトに向いている(大規模プロジェクトでは難しい)。
- 要求仕様の全項目の完了時期を約束しない(ウォーターフォールでは約束するが、約束が守られるとは限らない。また、無理に守ろうとするため、しばしばデスマーチが発生し、返って完成が遅れる)。
- ほとんどのアジャイル系プロセスは、チームが様々なサブプロジェクトやプラクティスを実行することを前提としている。
- プログラマが多能工であることを前提している。
 - 要求分析→設計→コーディング→単体テスト(UT)→統合テスト(IT)を各人ができなければならぬ。
 - 各人は、プロダクト開発言語のみではなく、単体テストフレームワークや各種自動化用言語を理解する必要がある。

ウォーターフォール vs アジャイル

ウォーターフォールとアジャイルの対比を下記する。

	ウォーターフォール	アジャイル
計画	無謬が前提なので硬直的に従う	誤りが前提なので修正しながら進める
進捗計測	ほぼ不可能	実測ベース
要求技能	OOD/C++	OOD/C++/TDD etc
自動化	通常は未実施	自動化前提
プロジェクト規模	規模とは関係小	大規模は難しい

アジャイル系プロセスは開発チーム全員による議論を要求するので、1チーム10人程度以下でなければ運営が効率的ではない。それ以上の人数が必要な場合、10人以下のチームを複数個作り、「各チームのリーダーが参加するイテレーション毎の計画ミーティング」でソフトウェア開発全体の計画作りと各チームへのタスクの割り振りを行うことになる。先に述べた理由から、このミーティングの参加者も10人程度以下が望ましい。以上のような考察から、10人を超えるような大規模開発にはアジャイル系プロセスは向きであるというのが常識的な結論である。逆にそれほどの規模でないプロジェクトでウォーターフォールを選ぶ理由はないと思われる。

アジャイル系プロセスのプラクティスとインフラ

多くのアジャイル系プロセスは、下記のようなサブプロセスやプラクティスの実施を前提とする。

- 自動単体テスト
- リファクタリング
- 自動統合テスト
- TDD
- CI(継続的インテグレーション)
- コードインスペクション

このプロセスでは前イテレーションまでに作られたテスト済のソースコードを何度も修正することになる。これにより、

- ソースコードを修正すると再帰テストが必要になるが、これを手作業で行えば、プロジェクトの工数が圧迫されるため、テストの自動化はアジャイル系プロセスにとって必須である
- 汚いソースコードへの機能追加は困難・非効率であるため、リファクタリングが必要であり、リファクタリングには単体テストが必須である

という理由から、自動単体テストとリファクタリングは特に重要なプラクティスである。

自動単体テスト

自動単体テストとは？

一般に、単体テスト(UT)とは、個々のクラスや関数といったソフトウェア構成要素の機能が正確に動作することを検証するためのテストを指す。原理的には、デバッガ等を利用して手作業で単体テストを実行することは可能であるが、

- 工数が膨大になる
- テストの再現性が低い
- 高速な操作のテストが困難である

等の問題がある(V字モデルであれば可能かもしれないが)ため、現実的ではない。

自動単体テスト(以下単に単体テストやUTと呼ぶこともある)とは、この問題に対処するためのもので、ワンコマンド(もしくはワンクリック)でクラスや関数の単体テストを行うプログラムである。

下記にstd::vectorの単体テストを例示する。

```
// @@@ example/etc/ut.cpp 7

TEST(UT, std_vector)
{
    auto v0 = std::vector{3, 2, 1};

    ASSERT_EQ(3, v0.size());
    ASSERT_EQ(3, v0[0]);
    ASSERT_EQ(2, v0[1]);
    ASSERT_EQ(1, v0[2]);
    ASSERT_THROW(v0.at(3), std::out_of_range); // エクセプション発生

    // sortのテスト
    std::sort(v0.begin(), v0.end());
    ASSERT_EQ({1, 2, 3}, v0);

    // transformのテスト
    auto v1 = std::vector<int>{};
    std::transform(v0.begin(), v0.end(), std::back_inserter(v1), [](auto x) { return x * 2; });
    ASSERT_EQ({2, 4, 6}, v1);
}
```

このようなプログラムを書くことを「単体テストを書く(を作る)」、このようなプログラムを実行する(実行してバグを取り除く)ことを「単体テストを行う(をする)」という。通常、単体テストソースコードのビルドや単体テストの実行は、その対象ソースコードと同じビルドシステム(makeやVisual Studioのソリューション等)に組み込まれる。単体テストを書き、それをビルドシステムから実行できるようにすることで、ほとんど工数をかけることなく何度も単体テストを繰り返し実行できる(つまり単体テストを持つクラスの回帰テストのコストをほぼ0にできる)。

単体テストのメリット

単体テストを行うメリットは、

- ・バグが単体テストで検出可能である場合、そのバグを統合テストで検出・デバッグするよりも、単体テストで検出・デバッグする方が効率的である。
- ・自動単体テストは工数をほとんどロスすることなしに何度でも実行できるため、機能追加、バグ修正、リファクタリング等のソースコード修正後の回帰テストが容易になる。
- ・ソースコードカバレッジを計測できるため、テストの網羅性を定量化できる。
- ・統合テスト以降では実施が難しいテスト(エラーハンドリング等)であっても、単体テストであれば比較的容易に実施できる。

一方で単体テストを行うデメリットは、

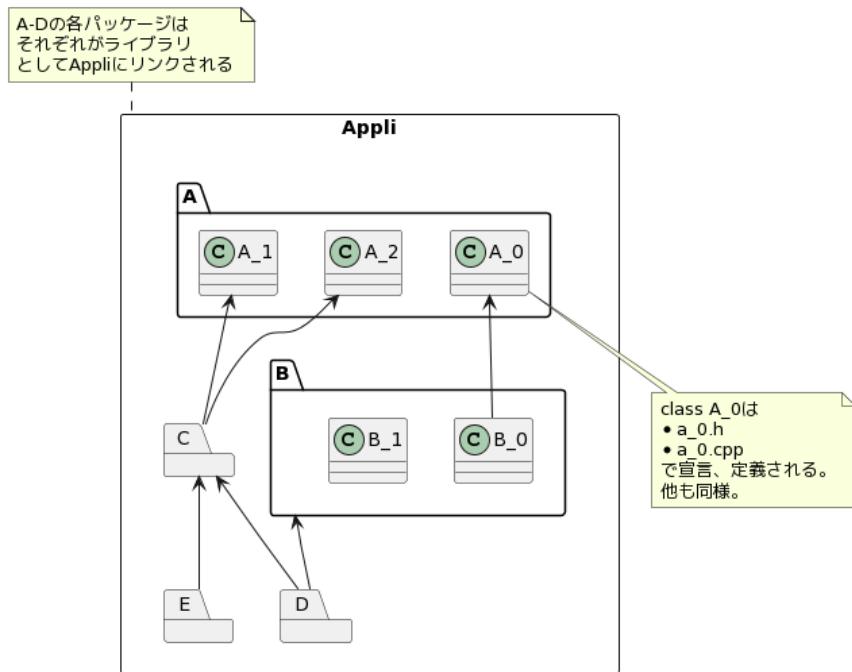
- ・単体テストが可能なクラス設計には、人員のスキルの向上が必要(OOD、デザインパターン等)である。
- ・単体テストのコーディングに時間がかかる(一見そう見えるが、単体テストにより統合テストやシステムテストの時間が短縮されるので、期間トータルでは問題ないことが多い)。

単体テストのメリット、デメリットを比べれば明らかな通り、単体テストを行わない合理的な理由はない(そもそも一般的な意味での単体テストを行わないプロセスはおそらく存在しない)。

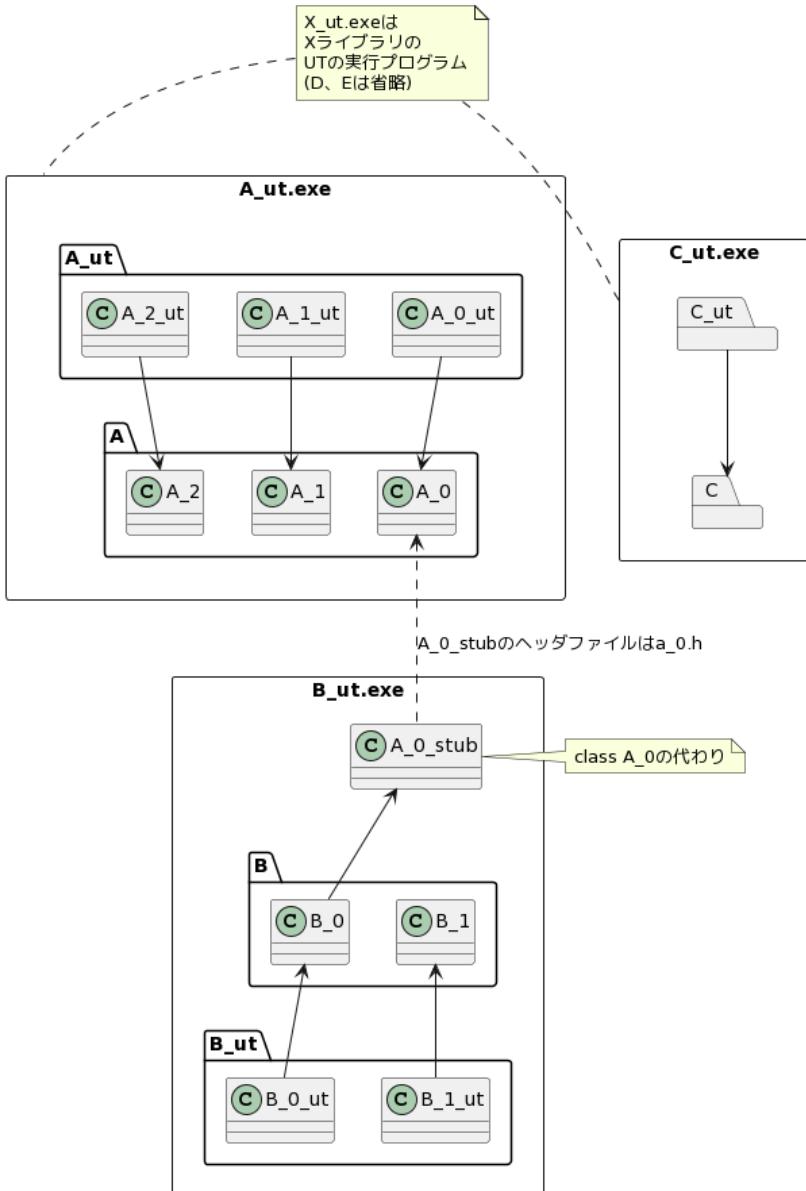
アーキテクチャと単体テスト

プログラムとその単体テストのパッケージの構造(「アーキテクチャ」参照)を説明するために、以下のような特徴を持つAppliというプログラムを想定する。

- ・「パッケージとその構成ファイル」で定めたルールに従っている。
- ・各パッケージはライブラリとして実装され、それらをリンクすることによりAppliが生成される。
- ・下記パッケージ図のような構造を持つ。



この場合、下記図のように各パッケージ(ライブラリ)毎に単体テスト実行プログラムを作るのが一般的である(「ファイル名」で述べたように、*.cppに対しては*_ut.cppとする)。



パッケージ間に無駄な依存関係や相互依存、循環依存があると、単体テスト用のモックやスタブを大量に作らざるを得なくなる場合が多く、最悪の場合、単体テスト用のリンクができないこともある。

単体テストの開発を行うかどうかに関係なく、効率の良いソフトウェア開発を行うためにはこのように整理されたパッケージ構造を持つことが好ましいが、単体テストの開発を行う場合、このような構造は特に重要となる（「[アーキテクチャとファイル構造](#)」参照）。

単体テストのサポートツール

C++をサポートする単体テストフレームワークとしては、

- cUnit
- [google test\(gtest\)](#)
- MSTest
- C++Test

等がある。

単体テスト用の実行形式バイナリはビルドを行うOS上で実行するため、組み込みソフトウェア開発のようにクロスコンパイラを使用している場合、単体テストのビルドにそのクロスコンパイラを使用することはできない。そのような場合、単体テスト用にはg++やclang++を使用することが一般的である。ネイティブなコンパイラを使用しているプロジェクトでは、単体テストのビルドにもそのコンパイラを使用する。

多くのビルド環境では下記のようなテストカバレッジ(g++/clang++とlcov)を出力できる。単体テストが十分かどうかは、それを見て判断できる。

LCOV - code coverage report

Current view: [top level](#) - sample

Test: cov.info

Date: 2020-06-05 20:22:52

	Hit	Total	Coverage
Lines:	230	256	89.8 %
Functions:	105	112	93.8 %

Filename	Line Coverage	Functions
find_files_old_style.cpp	92.0 %	23 / 25
find_files_straceexec.cpp	100.0 %	9 / 9
fixed_point.h	100.0 %	61 / 61
lsm.h	65.5 %	19 / 29
state_machine_new.cpp	95.8 %	46 / 48
state_machine_new.h	93.9 %	31 / 33
state_machine_old.cpp	80.4 %	41 / 51

Generated by: LCOV version 1.14

```

17     9 : std::vector<std::string> find_files_recursively(const std::string& path, FindCondition condition)
18     9 : {
19       9 : std::vector<std::string> files;
20       9 :
21       9 : std::for_each(fs::recursive_directory_iterator(fs::path(path.c_str())),
22       18 :           fs::recursive_directory_iterator()),
23       9 :
24      72 :           [&](const fs::path& p) {
25      72 :             bool is_match = false;
26      9 :
27      72 :             switch (condition) {
28      24 :               case FindCondition::File:
29      24 :                 if (fs::is_regular_file(p)) {
30      15 :                   is_match = true;
31      9 :                 }
32      24 :                 break;
33      24 :               case FindCondition::Dir:
34      24 :                 if (fs::is_directory(p)) {
35      9 :                   is_match = true;
36      9 :                 }
37      24 :                 break;
38      9 :               // @@ ignore begin
39      24 :               case FindCondition::FileNameBeginIs_f:
40      24 :                 if (p.filename().generic_string()[0] == 'f') {
41      9 :                   is_match = true;
42      9 :                 }
43      24 :                 break;
44      0 :               default:
45      0 :                 assert(false);
46      9 :                 // @@ ignore end
47      9 :
48      72 :               if (is_match) {
49      33 :                 files.push_back(p.generic_string());
50      9 :               }
51      81 :             }
52      9 :         }
53      9 :
54      9 :         return files;
55      9 :

```

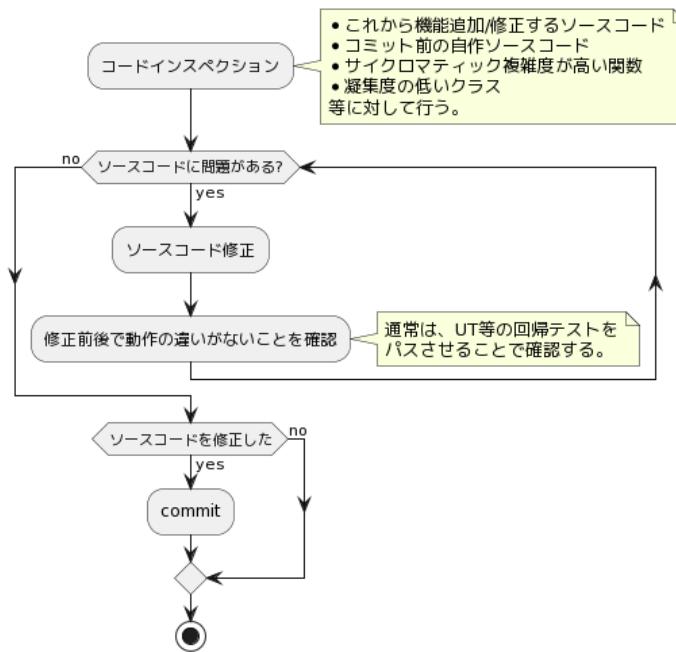
リファクタリング

リファクタリングとは、ソフトウェア(や、その構成物であるクラスや関数等)の外部に対する振る舞いを変えることなしに、その内部構造を改善することである。従って、リファクタリングには、

- リファクタリングの前後で動作の違ひがないことを確認できる
- ソースコードの問題点に気づき、それをより良いソースコードに改善できる

ことが必要である。

通常、リファクタリングは下図に示すようなワークフローとしてプロセスに組み込まれ、日々の開発業務の一環として行われる。



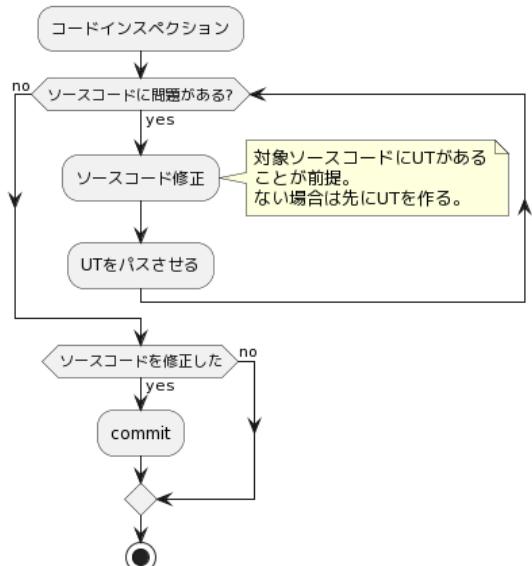
リファクタリングのための回帰テスト

すでに述べたように、リファクタリング後には回帰テストが必要である。通常、この回帰テストは、改善したソースコードをカバーする自動実行可能な単体テストや統合テスト等の、エンジニアの工数をほとんどロスしない方法で行われる。もし、多くの回帰テストが手作業で行われるならば、プロジェクトがその工数負担に耐えられなくなり、以下のいずれかが発生する。

1. リファクタリングがほとんど行われなくなる。
2. ソースコード改善後、十分な回帰テストが行われなくなる。
3. 一度に大量のソースコード改善を行うようになる(回帰テストの回数を減らす)。

ソースコードの構造を改善し、自動テストを組み込めるようにするために、上記3の方法は効果的である場合もある(すなわち、方法3はそのソースコードのライフサイクルの中で一度だけ施行が許される)。他の方法ではリファクタリングをプロセスに組み込むことはできない。

以上の考察から、リファクタリングのための回帰テストは、自動単体テストか自動統合テストのいずれか、もしくは両方にならざるを得ない。プロジェクトの進捗とともに自動統合テストは数時間～数日を要するようになるので、実践的に考えると、自動統合テストはリファクタリングのための回帰テストには不向きである。その結果、リファクタリング後の回帰テストには、「修正したソースコードやその周辺をカバーする自動単体テスト」以外の選択肢はないと考えられる。



ソースコードの改善

良いプログラマが書いたソースコードは、無駄がなく機能的である。それが機能美として目に映ることもあるため、自分や自分のチームにはそのようなソースコードは書けないと思う人がいる。他方で、良いソースコードとはどのようなものかも知らずに、自信満々にリファクタリングをやりたいという人もいる。

どちらの考え方も間違っている。「コードインスペクション」で述べた観点に沿ってソースコードを読み、その違反を確実に修正できることが、ソースコード改善の第一歩である。

これは前者が思うほど難しくもなく、後者が思うほど簡単でもない。

リファクタリングの例

ソースコードに多くの問題があった場合でも、一度に多くの視点からの修正をしてはならない。ステップバイステップで少しづつ改善させることを心掛けるべきである。

例えば大きすぎる関数を分割するリファクタリングを行う場合は、その分割のみに集中すべきで、その最中に別の問題を見つけても、その修正を行ってはならない。まずは、仕掛け中のリファクタリングを終了させ、その後(コミットした後)、次の問題点に取り掛かるべきである。

以下では、そういったステップバイステップのリファクタリングを例示する。

オリジナルのソースコード

まずはリファクタリング前のオリジナルのソースコードを示す。

```
// @@@ example/ref_org/main.cpp 8

int main(int, char**)
{
    auto strings = std::vector<std::string>{};

    for (;;) {
        auto buffer = std::string{};

        if (std::getline(std::cin, buffer)) {
            switch (buffer.at(0)) {
                case '+': {
                    auto result = std::string{do_heavy_algorithm(buffer.substr(1))};
                    strings.emplace_back(result);
                    for (auto i = 0U; i < strings.size(); i++) {
                        std::cout << strings[i] << std::endl;
                    }
                    break;
                }
                case '.':
                    // do nothing
                    break;
                case '=':
                    return 0;
                default:
                    return 1; // error exit
            }
        }
        return 0;
    }
}
```

上記プログラムは、

1. 標準入力から文字列を受け取り、
2. 文字列をパースし、コマンドと引数文字列に分離し、
3. 引数文字列を、時間のかかるアルゴリズムで別の文字列に変換し、
4. 変換文字列を保存し、
5. 保存した全ての変換文字列をstd::coutに出力する。

ソースコードの品質は高くないので、すくなくとも機能追加するタイミングではリファクタリングが必要になる。

機能追加によるソースコード品質劣化

時間のかかるアルゴリズムがプログラムをブロックしてしまうので、下記のように、この処理を(不十分ながら)非同期化した。

```
// @@@ example/ref_async_org/main.cpp 15

int main(int, char**)
{
    auto strings = std::vector<std::string>{};
    std::thread* thd = nullptr;

    for (;;) {
        auto buffer = std::string{};

        if (std::getline(std::cin, buffer)) {
            switch (buffer[0]) {
                case '+': {
                    if (thd != nullptr) {
                        thd->join();
                        delete thd;
                    }

                    thd = new std::thread{[&strings, input = buffer.substr(1)] {
                        auto result = std::string{do_heavy_algorithm(input)};
                        strings.emplace_back(result);

                        for (auto i = 0U; i < strings.size(); i++) {
                            std::cout << strings[i] << std::endl;
                        }
                    }};
                    break;
                }
                case '.': {
                    ...
                }
            }
        }
    }

    return 0;
}
```

一般に、同期処理をそのまま非同期に変更するとソースコードは腐敗を始める。上記ソースコードもその例に漏れず、かなり醜悪になった。

小規模なリファクタリング

Null Object/パターンやRAIIの導入で肥大化したmain関数を改善する小規模なリファクタリングを行う。

```
// @@@ example/ref_async_r0/main.cpp 13

int main(int, char**)
{
    auto strings = std::vector<std::string>{};
    auto thd = std::make_unique<std::thread>([] {}); // Null Object & RAII
    auto sg = ScopedGuard{[&thd] { thd->join(); }}; // RAII

    for (;;) {
        auto buffer = std::string{};

        if (std::getline(std::cin, buffer)) {
            switch (buffer[0]) {
                case '+': {
                    thd->join();
                    thd = std::make_unique<std::thread>([&strings, input = buffer.substr(1)] {
                        auto result = std::string{do_heavy_algorithm(input)};
                        strings.emplace_back(result);
                        for (auto const& str : strings) { // 範囲for文
                            std::cout << str << std::endl;
                        }
                    });
                    break;
                }
                case '.': {
                    ...
                }
            }
        }
    }

    return 0;
}
```

上記例のScopedGuardは、汎用性が高くプロジェクト全体で使用できるため、別のファイルとして、下記のように宣言、定義する(汎用性が高いクラスや関数をプロジェクト全体で共有することは良い習慣である)。

```
// @@@ deep/h/scoped_guard.h 4

/// @class ScopedGuard
/// @brief RAIIのためのクラス。
///       コンストラクタ引数の関数オブジェクトをデストラクタから呼び出す。
template <typename F>
class ScopedGuard {
public:
    explicit ScopedGuard(F&& f) noexcept : f_{f}
    {
        // f()がill-formedにならず、その戻りがvoidでなければならぬ
        static_assert(std::is_invocable_r<void, F>, "F must be callable and return void");
    }

    ~ScopedGuard() { f_(); }
    ScopedGuard(ScopedGuard const&) = delete; // copyは禁止
    ScopedGuard& operator=(ScopedGuard const&) = delete; // copyは禁止

private:
    F f_;
};
```

構造のリファクタリング

前記レベルでは不十分であるため、ブロックを関数化するリファクタリングを行う。

```
// @@@ example/ref_async_r1/main.cpp 9

namespace {
int main_loop()
{
    auto strings = std::vector<std::string>{};
    auto thd     = std::thread{[] {}}; // NullObject & RAII
    auto sg      = ScopedGuard{[&thd] { thd.join(); }}; // RAII

    for (;;) {
        auto buffer = std::string{};

        if (std::getline(std::cin, buffer)) {
            if (auto exit_code = dispatch(thd, strings, buffer)) {
                return *exit_code;
            }
        }
    }

    assert(false);
    return 0;
}
} // namespace

int main(int, char**) { return main_loop(); }
```

メインループ関数は、main()と同じファイルに残すが、他の関数は下記のように他のファイルで定義する。これにより、部分的だが単体テストが導入できる。

```

{
    switch (command[0]) {
    case '+':
        convert_store_async(thd, strings, command.substr(1));
        return std::nullopt;
    case '.':
        // do nothing
        return std::nullopt;
    case '=':
        return 0;
    default:
        return 1;
    }
}

```

単体テストの開発

前述したように、関数やファイルを分割したことにより、不十分なレベルではあるが単体テストを開発、実行できるようになった。

```

// @@@ example/ref_async_r1/lib_ut.cpp 9

TEST(RefAsyncR1, dispatch)
{
    auto actual = std::vector<std::string>{};

    {
        auto thd = std::thread{[] {}};
                                // NullObject & RAI
        auto sg  = ScopedGuard{[&thd] { thd.join(); }}; // RAI

        {
            auto exit_code = dispatch(thd, actual, "+abc");
            ASSERT_FALSE(exit_code);
        }

        {
            auto exit_code = dispatch(thd, actual, "+defg");
            ASSERT_FALSE(exit_code);
        }

        {
            auto exit_code = dispatch(thd, actual, ".");
            ASSERT_FALSE(exit_code);
        }

        {
            auto exit_code = dispatch(thd, actual, "+hijkl");
            ASSERT_FALSE(exit_code);
        }

        {
            auto exit_code = dispatch(thd, actual, "=");
            ASSERT_TRUE(exit_code);
            ASSERT_EQ(0, *exit_code);
        }

        {
            auto exit_code = dispatch(thd, actual, "?");
            ASSERT_TRUE(exit_code);
            ASSERT_NE(0, *exit_code);
        }
    }

    ASSERT_EQ(std::vector<std::string>{"ABC", "DEFG", "HIJKL"}, actual);
}

```

scoped_guard.hに関しても、以下のように単体テストを追加する。バグが発生しそうにないこのようないいクラスに対しても単体テストを行うことは一見無駄なように見えるが、単体テストカバレッジの管理、コードクローンの撲滅、「割れ窓理論」等の観点から重要である。

```

// @@@ example/programming_convention/scoped_guard_ut.cpp 7

TEST(ScopedGuard, scoped_guard)
{
    auto a = 0;

    {
        auto sg = ScopedGuard{[&a]() noexcept { a = 99; }};
        ASSERT_NE(99, a); // ~ScopedGuardは呼ばれていない
    }
    ASSERT_EQ(99, a); // ~ScopedGuardは呼ばれた
}

```

クラスの導入

このプログラムは非同期処理が必要であるため、 そういったアプリケーションとの相性が良いMVCの導入によるリファクタリングを行う（この程度の規模のソフトウェアにMVCを導入する必要はないが、 その導入を例示するためリファクタリングを行う）。

まずは、コマンドの非同期処理を行うクラス（＝ビジネスロジック＝Model）を導入する（例としての分かりやすさを優先するためにクラス名もModelとする）。

```
// @@@ example/ref_async_r2/main.cpp 6

namespace {
int main_loop()
{
    auto model = Model{};

    for (;;) {
        auto buffer = std::string{};

        if (std::getline(std::cin, buffer)) {
            if (auto exit_code = dispatch(model, buffer)) {
                return *exit_code;
            }
        }
    }

    assert(false);
    return 0;
}
} // namespace

int main(int, char**) { return main_loop(); }
```

以下のファイルで、Modelの宣言、定義を行う。

```
// @@@ example/ref_async_r2/lib.h 8

class Model {
public:
    Model() : thd_{[] {}}, strings_{ {} } {}
    ~Model() { thd_.join(); }
    void ConvertStoreAsync(std::string const& input);

private:
    std::thread thd_;
    std::vector<std::string> strings_;
};
```



```
// @@@ example/ref_async_r2/lib.cpp 6

void Model::ConvertStoreAsync(std::string const& input)
{
    thd_.join();

    thd_ = std::thread{[&sv = strings_, input = input] {
        sv.emplace_back(do_heavy_algorithm(input));

        for (auto const& str : sv) {
            std::cout << str << std::endl;
        }
    }};
}

std::optional<int> dispatch(Model& model, std::string const& command)
{
    ...
}
```

単体テストの変更

このリファクタリングにより単体テストは以下のようになる（以下のModelのデザインは不十分であるため、 妥当な単体テストはできない）。

```
// @@@ example/ref_async_r2/lib_ut.cpp 7

TEST(RefAsyncR2, Model)
{
    auto model = Model{}; // Modelのデザインが悪いために適切な単体テストは書けない

    model.ConvertStoreAsync("hehe");
}
```

```

TEST(RefAsyncR2, dispatch)
{
    auto model = Model{};

    {
        auto exit_code = dispatch(model, "+abc");
        ASSERT_FALSE(exit_code);
    }
    {
        auto exit_code = dispatch(model, "+defg");
        ASSERT_FALSE(exit_code);
    }
    {
        auto exit_code = dispatch(model, ".");
        ASSERT_FALSE(exit_code);
    }
    {
        auto exit_code = dispatch(model, "+hijkl");
        ASSERT_FALSE(exit_code);
    }
    {
        auto exit_code = dispatch(model, "=");
        ASSERT_TRUE(exit_code);
        ASSERT_EQ(0, *exit_code);
    }
    {
        auto exit_code = dispatch(model, "?");
        ASSERT_TRUE(exit_code);
        ASSERT_NE(0, *exit_code);
    }
}

```

MVCの導入

非同期関数であるModel::ConvertStoreAsync()の完了がクラス外から捕捉できないことを一因として、上記例のModelへの十分な単体テストができなかった。この問題を解決し、単体テストのカバレッジを上げるために「MVC」の構造を導入する。

```

// @@@ example/ref_async_r3/main.cpp 5

int main(int, char**)
{
    auto view = View{};
    auto model = Model{};
    model.Attach(view);
    auto controller = Controller{model};

    return controller.WatchInput();
}

```

以下のファイルで、Modelの宣言、定義を行う。

```

// @@@ example/ref_async_r3/model.h 10

class Observer {
public:
    Observer() = default;
    void Update(Model const& model) { update(model); }
    virtual ~Observer() = default;

private:
    virtual void update(Model const& model) = 0;
};

class Model {
public:
    Model() : thd_{{}}, strings_{{}}, observers_{{}} {}
    ~Model() { thd_.join(); }
    void ConvertStoreAsync(std::string const& input);
    void Attach(Observer& observer);
    void Detach(Observer& observer);
    std::vector<std::string> const& GetStrings() const { return strings_; }

private:
    void notify() const;

    std::thread thd_;
    std::vector<std::string> strings_;
};

```

```

        std::list<Observer*> observers_;
};

// @@@ example/ref_async_r3/model.cpp 4

void Model::ConvertStoreAsync(std::string const& input)
{
    thd_.join();

    thd_ = std::thread{[this, input = input] {
        strings_.emplace_back(do_heavy_algorithm(input));
        notify();
    }};
}

void Model::Attach(Observer& observer) { observers_.emplace_back(&observer); }
void Model::Detach(Observer& detach)
{
    observers_.remove_if([&detach](Observer* observer) { return &detach == observer; });
}

void Model::notify() const
{
    for (auto* observer : observers_) {
        observer->Update(*this);
    }
}

```

以下のファイルで、Viewの宣言、定義を行う。

```

// @@@ example/ref_async_r3/view.h 7

class View : public Observer {
private:
    virtual void update(Model const& model) override;
};

// @@@ example/ref_async_r3/view.cpp 3

void View::update(Model const& model)
{
    for (auto const& str : model.GetStrings()) {
        std::cout << str << std::endl;
    }
}

```

以下のファイルで、Controllerの宣言、定義を行う。

```

// @@@ example/ref_async_r3/controller.h 7

class Controller {
public:
    explicit Controller(Model& model) : model_{model} {}

    int WatchInput();

private:
    std::optional<int> dispatch(std::string const& command);

    Model& model_;
};

// @@@ example/ref_async_r3/controller.cpp 6

int Controller::WatchInput()
{
    for (;;) {
        auto buffer = std::string{};

        if (std::getline(std::cin, buffer)) {
            if (auto exit_code = dispatch(buffer)) {
                return *exit_code;
            }
        }
    }

    assert(false);
    return 0;
}

```

```

    std::optional<int> Controller::dispatch(std::string const& command)
{
    ...
}

```

単体テストの変更(単体テスト用クラス導入)

Observerから派生した下記のテスト用クラスViewTestを使うことにより、ConvertStoreAsync()の完了が捕捉できるようになった。

```

// @@@ example/h/ref_async_mock.h 10

class ViewTest : public Observer {
public:
    void WaitUpdate(uint32_t num) noexcept // num回、updateが呼び出されるまでブロック
    {
        while (update_counter_ != num) { // ポーリングは避けるべきだが、単体テストなら問題ない
            org_msec_sleep(100);
        }
    }

    uint32_t GetCount() const noexcept { return update_counter_; }

private:
    virtual void update(Model const&) noexcept override { ++update_counter_; }

    std::atomic<uint32_t> update_counter_{0};
};

```

これにより、Modelの単体テストは十分なレベルになったが、下記の通り、ControllerやViewがstd::coutやstd::cinに依存しているために、これらの単体テストの開発は困難である。

```

// @@@ example/ref_async_r3/model_ut.cpp 10

TEST(RefAsyncR3, Model)
{
    auto view_test = ViewTest{};

    auto model = Model{};
    model.Attach(view_test);

    auto const input_model = std::vector<std::string>{"abc", "defg", "hijkl"};

    for (auto const& s : input_model) {
        model.ConvertStoreAsync(s);
    }

    view_test.WaitUpdate(input_model.size());

    ASSERT_EQ((std::vector<std::string>{"ABC", "DEFG", "HIJKL"}), model.GetStrings());
}

```

```

// @@@ example/ref_async_r3/view_ut.cpp 6

TEST(RefAsyncR3, View)
{
    auto view = View{}; // オブジェクト生成程度の単体テストしかできない
}

```

```

// @@@ example/ref_async_r3/controller_ut.cpp 8

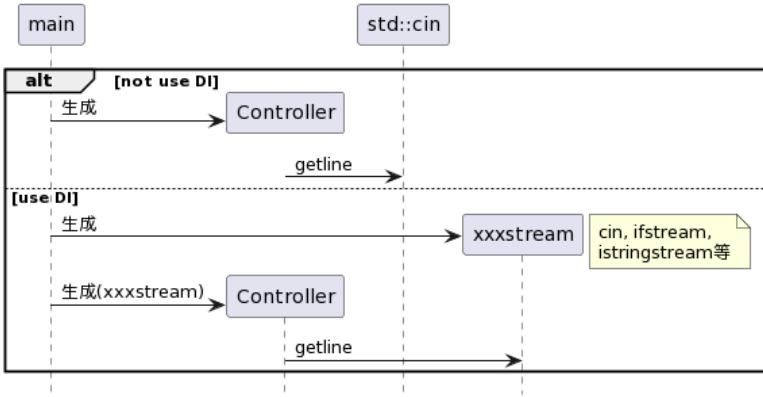
TEST(RefAsyncR3, Controller)
{
    auto model = Model{};
    auto controller = Controller{model}; // オブジェクト生成程度の単体テストしかできない
}

```

DIの導入

DI(dependency injection)を導入することで、Controller、Viewのstd::coutやstd::cinへの直接の依存を回避する。

下図は、DI導入前後でControllerによる「std::cinからの文字列読み込み」がどのように変更されたかを示す。



DIの導入により`main()`は以下のようにになる。

```
// @@@ example/ref_async_r4/main.cpp 5

int main(int, char**)
{
    auto view = View{std::cout}; // 修正前のソースコード
    auto model = Model{}; // > auto view = View{};
    model.Attach(view); // > auto model = Model{};
    auto controller = Controller{model, std::cin}; // > auto controller = Controller{model};

    return controller.WatchInput();
}
```

`Model`については、`std::cout`、`std::cin`への依存はないので変更しない。`View`については、以下のように`std::cout`への依存を削除する。

```
// @@@ example/ref_async_r4/view.h 7

class View : public Observer {
public:
    explicit View(std::ostream& os) : Observer{}, os_{os} {}

private:
    virtual void update(Model const& model) override;
    std::ostream& os_;
};

// @@@ example/ref_async_r4/view.cpp 3

void View::update(Model const& model) // 修正前のソースコード
{
    for (auto const& str : model.GetStrings()) { // > for (auto const& str : model.GetStrings()) {
        os_ << str << std::endl; // > std::cout << str << std::endl;
    } // }
}
```

`Controller`についても、以下のように`std::cin`への依存を削除する。

```
// @@@ example/ref_async_r4/controller.h 7

class Controller {
public:
    explicit Controller(Model& model, std::istream& is) : model_{model}, is_{is} {}

    int WatchInput();

private:
    std::optional<int> dispatch(std::string const& command);

    Model& model_;
    std::istream& is_;
};

// @@@ example/ref_async_r4/controller.cpp 6
```

```
int Controller::WatchInput()
{
    for (;;) {
        auto buffer = std::string{};

        // 修正前のソースコード
```

```

        if (std::getline(is_, buffer)) { // if (std::getline(std::cin, buffer)) {
            if (auto exit_code = dispatch(buffer)) {
                return *exit_code;
            }
        }

        assert(false);
        return 0;
    }

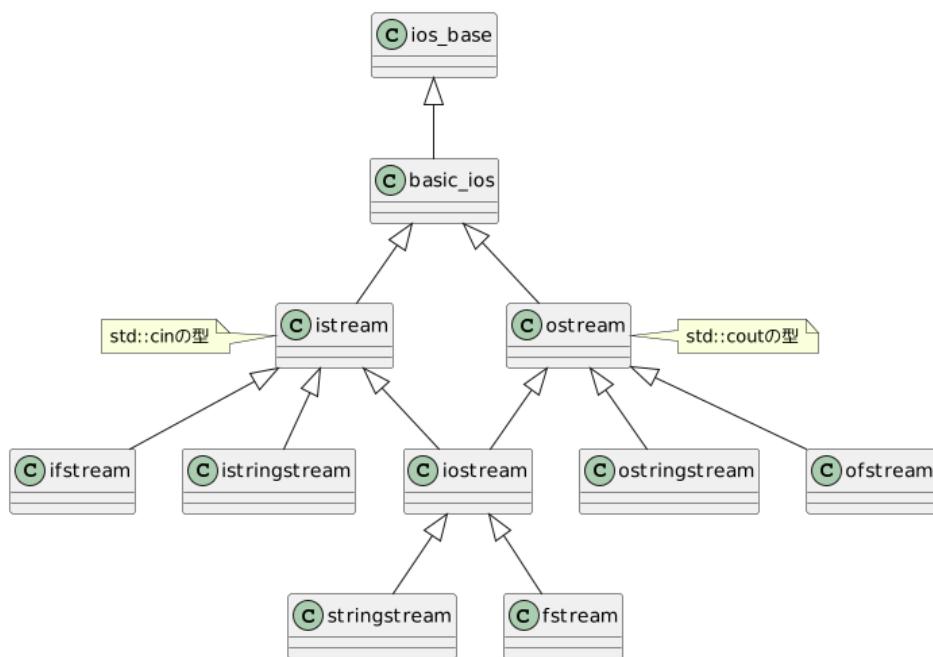
    std::optional<int> Controller::dispatch(std::string const& command)
{
    ...
}

```

以上のように、

- Controllerは、インスタンスstd::cinへの依存から、std::istream型への依存へ
- Viewは、インスタンスstd::coutへの依存から、std::ostream型への依存へ

と改善された(一般にグローバルオブジェクトへの依存よりも型への依存の方が柔軟性に勝る)。なお、std::cin、std::cout、std::istream、std::ostream等の継承関係は以下の通りであるため、このような変更が可能となった。



全クラスの単体テスト

これまでの改善によりModel、View、Controllerすべてに妥当な単体テストをすることが可能となった。

```

// @@@ example/ref_async_r4/model_ut.cpp 11

TEST(RefAsyncR4, Model)
{
    auto view_test = ViewTest{};
    auto ss_view = std::ostringstream{};
    auto view_normal = View{ss_view};
    auto model = Model{};

    model.Attach(view_normal);
    model.Attach(view_test);

    auto const input_model = std::vector<std::string>{"abc", "defg", "hijkl"};

    for (auto const& s : input_model) {
        model.ConvertStoreAsync(s);
    }

    view_test.WaitUpdate(input_model.size());

    ASSERT_EQ(std::vector<std::string>{"ABC", "DEFG", "HIJKL"}, model.GetStrings());
}

```

```

    auto ss_expect = std::ostringstream{}; // viewのテスト
    ss_expect << "ABC" << std::endl;
    ss_expect << "ABC" << std::endl << "DEFG" << std::endl;
    ss_expect << "ABC" << std::endl << "DEFG" << std::endl << "HIJKL" << std::endl;

    ASSERT_EQ(ss_expect.str(), ss_view.str());
}

```

```

// @@@ example/ref_async_r4/view_ut.cpp 4

TEST(RefAsyncR4, View)
{
    // model/controllerの単体テストで代用
}

```

```

// @@@ example/ref_async_r4/controller_ut.cpp 11

TEST(RefAsyncR4, Controller)
{
    auto view_test = ViewTest{};
    auto ss_view = std::ostringstream{};
    auto view_normal = View{ss_view};
    auto model = Model{};

    model.Attach(view_test);
    model.Attach(view_normal);

    auto ss = std::stringstream{};

    ss << "+abc" << std::endl;
    ss << "+defg" << std::endl;
    ss << "." << std::endl;
    ss << "+hijkl" << std::endl;
    ss << "=" << std::endl;

    auto controller = Controller{model, ss};

    ASSERT_EQ(0, controller.WatchInput());

    auto const exp_model = std::vector<std::string>{"ABC", "DEFG", "HIJKL"};
    view_test.WaitUpdate(exp_model.size());

    ASSERT_EQ(exp_model, model.GetStrings());

    auto ss_expect = std::ostringstream{};
    ss_expect << "ABC" << std::endl;
    ss_expect << "ABC" << std::endl << "DEFG" << std::endl;
    ss_expect << "ABC" << std::endl << "DEFG" << std::endl << "HIJKL" << std::endl;

    ASSERT_EQ(ss_expect.str(), ss_view.str());

    // エラー入力テスト
    ss << "?" << std::endl;
    ASSERT_NE(0, controller.WatchInput());
}

```

自動統合テスト

自動統合テストとは？

一般に、統合テスト(IT)とは、クラスや関数、ライブラリ等のソフトウェア構成要素すべてを結合したプログラムの動作が、要求仕様に沿っていることを検証するためのテストである。統合テストはテストエンジニアやプログラマの手作業で行われることが慣例になっているが、プログラムに少しの工夫を加えることで、その多くを自動化することができる(プログラムにもよるが100%自動化は困難である)。この自動化された統合テストを本ドキュメントでは自動統合テストと呼ぶ。

以下のようなテストを手作業で行うことにはほぼ不可能であるが、これらの項目を統合テストから外すこともできないため、すべてのソフトウェア開発において自動統合テストは必須である。

- ・長時間オペレーションで不具合(メモリリーク等)が出ないことの検証
- ・手作業では難しい高速入力

自動統合テストのための仕様追加

リファクタリングの説明に使用したソースコード(「[リファクタリング](#)」参照)は、

- ・複数の小さなファイルに分割され、
- ・MVC構造を導入され、
- ・すべてのクラスは十分にコンパクトで、
- ・すべてのクラスは単体テストを実行できる

ように改善された。単体テストのラインカバレッジは100%に近く、動作品質という観点からも改善したが、未だに統合テストは手作業で行わなければならない(現在の仕様では統合テストを自動化することは難しい)。

このプログラムの統合テストの自動化を難しくさせている原因是、「標準入出力を利用し、ユーザとインタラクティブなやり取りを行う」からである。この問題を解決するために、「コマンド引数によりダイナミックに、標準入出力をファイル入出力へ切り替える」ように変更を行う(このようなテスト機能の実現のために、#if/#endif等のプリプロセッサ命令を利用することは誤りである)。

コマンド引数の仕様(このプログラム名はref_async_r5)は下記のとおりである。

```
ref_async_r5 [OPTIONS]
-i <input-file>    std::cinの代わりに、<input-file>を使用する。
-o <output-file>   std::coutの代わりに、<output-file>を使用する。
```

自動統合テストのためのソースコード変更

前述したオプションを備えた実装は、以下のようになる。

```
// @@@ example/ref_async_r5/main.cpp 10

namespace {

void how_to_use(std::string_view program)
{
    std::cerr << program << " [OPTIONS]" << std::endl;
    std::cerr << " -i <input-file>" << std::endl;
    std::cerr << " -o <output-file>" << std::endl;
}

} // namespace

int main(int argc, char** argv)
{
    auto ret = getopt(argc, argv);
    if (!ret) {
        how_to_use(argv[0]);
        return __LINE__;
    }

    auto ios = IOStreamSelector{std::move(ret->ifile), std::move(ret->ofile)};

    if (!ios.Open()) {
        how_to_use(argv[0]);
        return __LINE__;
    }
    //                                         // 修正前のソースコード
    auto view  = View{ios.GetOStream()};           // > auto view  = View{std::cout};
    auto model = Model{};                         // > auto model = {};
    model.Attach(view);                          //   model.Attach(view);
    auto controller = Controller{model, ios.GetIStream()}; // > auto controller
                                                //   = Controller{model, std::cin};

    return controller.WatchInput();
}
```

```
// @@@ example/ref_async_r5/arg.h 5

struct opt_result {
    std::string ifile;
    std::string ofile;
};

std::optional<opt_result> getopt(int argc, char* const* argv);
```

```
// @@@ example/ref_async_r5/arg.cpp 7

std::optional<opt_result> getopt(int argc, char* const* argv)
{
    auto opt  = int{};
    auto ifile = std::string{};
    auto ofile = std::string{};

    ...
```

```
    return opt_result{std::move(ifile), std::move(ofile)};
}
```

IOStreamSelectorと単体テスト用に導入した「[DI\(dependency injection\)](#)」構造により、 std::istream、 std::ostreamのインスタンスを選択できるようになった。

IOStreamSelectorは以下のようになる。

```
// @@@ example/ref_async_r5/arg.h 14

class IOStreamSelector {
public:
    IOStreamSelector(std::string ifile, std::string ofile)
        : ifile_{std::move(ifile)},
          ifs_{},
          is_{nullptr},
          ofile_{std::move(ofile)},
          ofs_{},
          os_{nullptr}
    {
    }

    bool Open();
    std::istream& GetIStream();
    std::ostream& GetOStream();

private:
    ...
};

// @@@ example/ref_async_r5/arg.cpp 36

namespace {
template <typename FSTREAM, typename IOSTREAM>
bool select_iostream(std::string const& filename, FSTREAM& fs, IOSTREAM& cin_cout,
                     IOSTREAM*& output)
{
    output = &cin_cout;

    if (filename.size() != 0) {
        fs.open(filename);
        if (!fs) {
            return false;
        }
        output = &fs;
    }

    return true;
}
} // namespace

bool IOStreamSelector::Open()
{
    if (!select_iostream(ifile_, ifs_, std::cin, is_)) {
        return false;
    }

    return select_iostream(ofile_, ofs_, std::cout, os_);
}

std::istream& IOStreamSelector::GetIStream()
{
    assert(is_ != nullptr);

    return *is_;
}

std::ostream& IOStreamSelector::GetOStream()
{
    assert(os_ != nullptr);

    return *os_;
}
```

自動統合テスト用ソースコード変更のための単体テスト

追加機能に対する単体テストを以下に示す。なお、新規単体テストはファイルの生成、書き込み、読み込みを行うため、

- ・単体テストの実行前に、結果に影響を与えるファイルを削除する
- ・単体テストの実行後に、単体テストで生成したファイルを削除する

を行う必要がある。

- ・前者を行うのがSetUp()
- ・後者を行うのがTearDown()

である。

```
// @@@ example/ref_async_r5/arg_ut.cpp 18

class Refactorin_5 : public ::testing::Test { // google testクラス
protected:
    virtual void SetUp() noexcept override
    {
        remove_file(ofilename); // ファイルがあると、テストがエラーするので。
        remove_file(ifilename);
    }

    virtual void TearDown() noexcept override
    {
        remove_file(ofilename); // ローカルリポジトリにゴミを残さない。
        remove_file(ifilename);
    }

    static void remove_file(std::string const& filename) noexcept
    {
        if (std::filesystem::exists(filename)) {
            std::filesystem::remove(filename);
        }
    }
};

TEST_F(Refactorin_5, IOStreamSelector)
{
    {
        auto ios = IOStreamSelector{"", ofilename};

        ASSERT_TRUE(ios.Open());
        ASSERT_TRUE(&std::cin == &ios.GetIStream());
    }
    ASSERT_TRUE(std::filesystem::exists(ofilename));

    {
        // テスト用ファイルの作成
        auto ofs = std::ofstream{};

        ofs.open(ifilename);
        ASSERT_TRUE(ofs);

        ofs << "test";
    }
    ASSERT_TRUE(std::filesystem::exists(ifilename));

    auto ios = IOStreamSelector{ifilename, ""};

    ASSERT_TRUE(ios.Open());
    ASSERT_TRUE(&std::cout == &ios.GetOStream());

    auto file_contents = std::string{};

    ios.GetIStream() >> file_contents;
    ASSERT_EQ("test", file_contents);
    }
}

TEST_F(Refactorin_5, getopt)
{
    {
        char p[] = "p";
        char* const argv[] = {p};

        auto ret = getopt(array_length(argv), argv);

        ASSERT_TRUE(ret);
        ASSERT_EQ("", ret.value().ifile);
        ASSERT_EQ("", ret.value().ofile);
    }
}
```

```
...  
}
```

自動統合テストの実装

ref_async_r5に追加された機能をスクリプト言語(下記例ではbash)等から利用することで、下記のような自動統合テスト(非手作業統合テスト)を開発することができる。入力文字列や入力タイミングのバリエーションを増やすこと等によってこの方法を発展させれば、堅固な自動統合テストにすることも可能である。

```
// @@@ example/ref_it/it.sh 45

# $IFILEの作成
cat << IFILE_END > $IFILE
+abcdef
+ddd
+fffff
=
IFILE_END

# expectの生成
gen_expect "ABCDEF" "DDD" "FFFF" > $OFILE_EXP

$TARGET -i $IFILE -o $OFILE_ACT

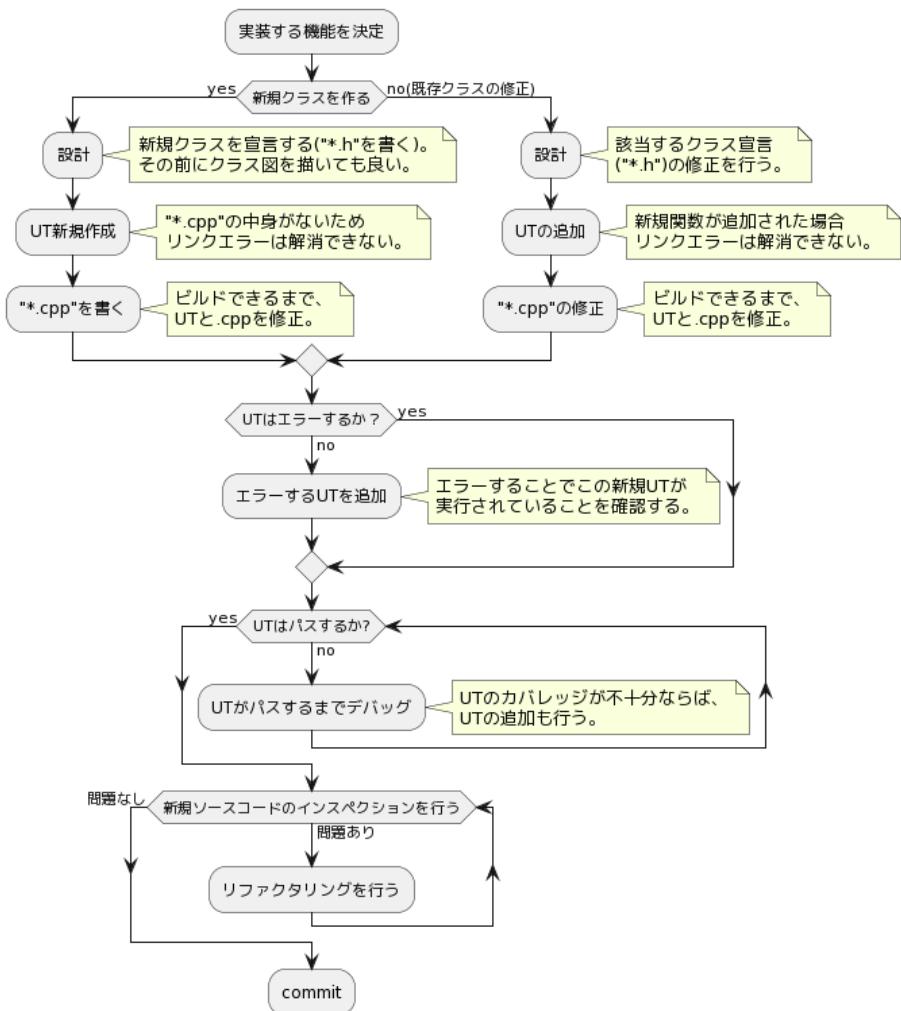
declare -r diff_result1=$(diff $OFILE_EXP $OFILE_ACT)

# $TARGETが正常動作すれば、文字列diff_result1の長さは0
[[ -n "$diff_result1" ]] && exit $LINENO
```

TDD

この章の考察に従って、単体テストをプロセスに組み込むのであれば、ほとんどのクラス、関数は統合テスト前までに単体テストを実施されデバッグされることになる。もしそうであるならば、この作業の流れ(クラスや関数の開発→単体テスト)はTest driven development(TDD)を使うことでさらに効率的になる。

TDDとは、下図に示すようなプログラマのワークフローである。



これは極めて強力なプラクティスであり、TDDを習慣化することで生産性の大幅な向上が見込める。

CI(継続的インテグレーション)

CI(continuous integration == 継続的インテグレーション)は、バージョン管理システムやそのウェブサービス、ブランチ運用等と密接な関係を持つため、まずはこれらの説明を行う。

バージョン管理システム

バージョン管理システムは、ソフトウェア開発にとって最も重要なインフラの一つである。従って、多彩なバージョン管理システムから何を選ぶかは、プロジェクトの成否に大きく影響する重大な意思決定である。

機能、性能、今後の発展、情報入手の容易さ等を総合的に考えると、

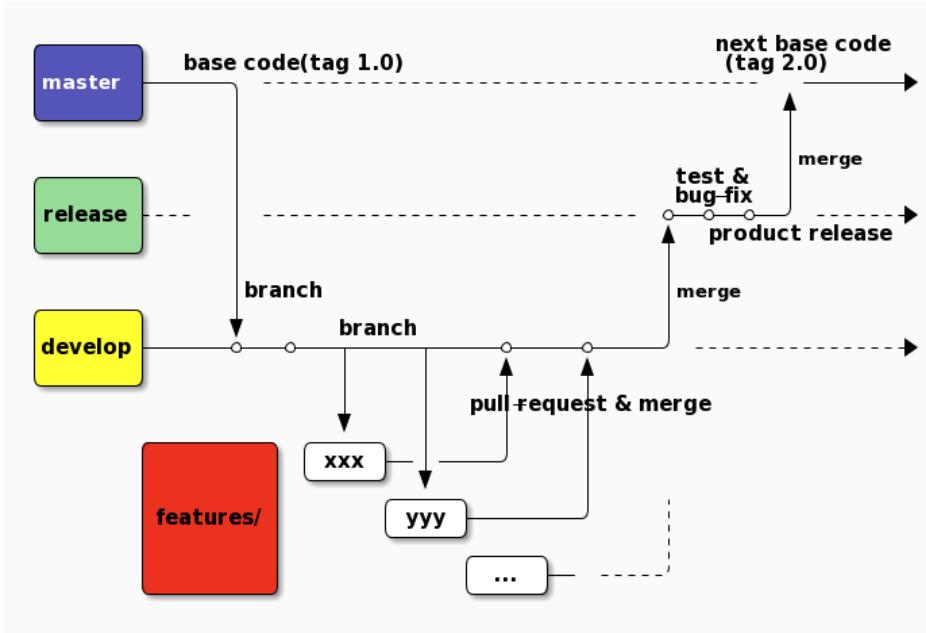
- ・バージョン管理システムにはgit
- ・そのウェブサービスにはGitHub

がベストな選択であると思われる。社内ルール等によりGitHubが使えない場合、GitHubと同等なgitウェブサービスをインターネット内に構築することを推奨する。

gitのブランチモデル

gitは極めて自由度の高いバージョン管理システムであるため、ブランチ運用については細心の注意が必要である。特別な理由がない限りgit-flow (A successful Git branching model) 等の世界的に評価の高い運用モデルを使用すべきである。

単純化したgit-flowを下記する。



このモデルにおける特に大切なポイントは、プロジェクト全体でシェアされる開発用ブランチdevelopと、個別の機能開発用ブランチ(features/xxx, yyy等)を分けたことである。これにより、

- 中途半端な機能の追加によりdevelopブランチの動作が不安定になることを軽減できる。
- 上記の問題回避のために、プログラマのローカルブランチにソースコードが滞留し続けることを回避できる。
- featureブランチからdevelopブランチへのマージ前にpull-requestすることで、コードインスペクションの実施やその履歴管理が容易になる。

等のメリットを享受できる。

コードインスペクション

GitHubのようなシステムを前提とするプロセスでは、コードインスペクションはpull-requestをトリガーとして行われる。pull-requestとは、featureブランチをdevelopブランチへマージする直前に行われる、ブランチ開発者からインスペクタへの依頼である。インスペクタはこの依頼を受けると、対象コミットのコードインスペクションを行い、

- 問題ない品質であると判断した場合、pull-requestを承認する。
- 問題がある場合は、その部分を指摘し、pull-requestを却下する。

pull-requestが承認されれば、ブランチ開発者はfeatureブランチをdevelopブランチにマージする。

ブランチ開発者は、pull-request前にその対象のコミットが、以下のコミットクライテリアを満たしていることを保証しなければならない。

- 最新のdevelopブランチはそのfeatureブランチにマージされている（マージされてないと、この後のfeatureブランチからdevelopブランチへのマージで多くのコンフリクトが発生し、インスペクションが無駄になることがある）。
- 新規のクラスや関数の単体テスト、新規機能の統合テストは作られていて、パスしている。
- コミットは十分に小さい（様々な目的のソースコードを一度のコミットに混在させるべきではない）。

コードインスペクションは、以下のような観点で行われる。

- コミットクライテリアをクリアしているか（単体テストや自動統合テストが作られているかどうかの確認）？
- 設計上の問題点はないか？
 - SOLID等の原則に従っているか？
 - デザインパターンの使用は適切か（AccessorやSingletonの多用は認められない等）？
- プログラミング規約に従っているか？
 - 不要な依存関係はないか？依存関係の方向は問題ないか？
 - クラス、関数は大きすぎないか？
 - コードクローンや同型のcase文はないか？
 - 識別子やファイル名等の名前は適切か？

CIとは？

内容を具体化、単純化するためこれまで述べてきたように、今後の説明も下記項目を前提とする（この前提でなければCIができないという意味ではない。当然ながらsubversion等でもCIの運用は可能である）。

- ・バージョン管理システムにgitを使用する。
- ・gitウェブサービスにはGitHubか、それと同等のものを使用する。
- ・gitの運用は、git-flowに従う（「[gitのブランチモデル](#)」参照）。

CIとは、「すべてのプログラマは、1日1回以上の頻度で、featureブランチをdevelopブランチへマージしなければならない」というプラクティスである。developブランチが更新され次第、ビルド、単体テスト、統合テスト等を自動で行うシステムとの併用が前提となるため、本ドキュメントのCIとは、本来の意味に加えてこの自動システムの運用も含めた概念であると定義する（一般にも、そのように定義していると思われる）。また、そのシステムをCIサーバ、CIサーバが実行するジョブ項目を単にCI項目と呼ぶ。CIサーバにはクラウドサービスを含めて様々なものがあるが、代表的なものはJenkinsである。

CIの技術的優位性は、「枝分かれしてから長時間が経過したブランチは統合(マージ)が困難である」という前提から発生している。この前提は明らかに正しいが、こうしないプロジェクトにはこうしない理由がある。マージするとdevelopブランチの動作品質が下がり、チーム全体の作業が滞るからである。この問題の対抗策がビルド、単体テスト、統合テスト等の自動実行であるため、これらの自動化ができていないチームが、featureブランチからdevelopブランチへのマージを頻繁に行うと悲惨な結果になる。一方で、developブランチへのマージを頻繁に行わないチームは、いずれ困難なマージを行わざるを得なくなる。この作業は多くのデグレードを引き起こすため、これも悲惨な結果となる。

以上をまとめると、

- ・developブランチへのマージを頻繁に行った方が、効率よくソフトウェア開発ができる
- ・そのためには、ビルド、単体テスト、統合テストの自動化ができなければならない

ということになる。従って、

- ・CIは効率よいソフトウェア開発を行うための重要なファクターである

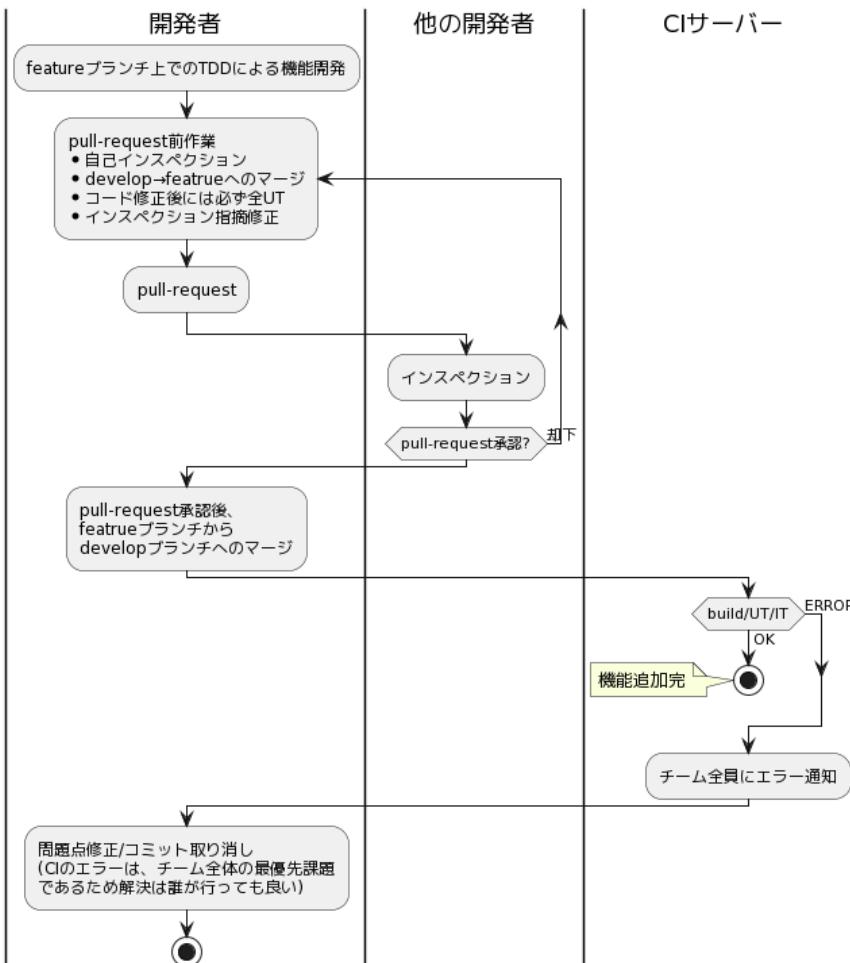
というのが結論である。

CIとワークフロー

アジャイル系プロセスに下記プラクティス

- ・TDD
- ・git-flow
- ・コードインスペクション(pull-request)
- ・CI

を組み込んだ場合、プログラマの典型的なワークフローは下図のようになる。



多くのアジャイル系プロセスは、このように洗練されたワークフローを前提としている。逆にこのようなワークフローを行えないチームのアジャイル系プロセスは機能しない。アジャイル導入の失敗例のほとんどは、こう言った問題が原因となっている。

このようなプラクティス、特に自動単体テストや自動統合テストを実践できていないチームが、いきなりこのワークフローを身に着けることは極めて困難である。イテレーション毎に決定される開発項目、改善項目の中にワークフローの向上に必要な項目を入れ、ステップバイステップで改善し続けることが重要である。

CI項目実行の長時間化と分割

CI項目には前述した

- ビルド
- 単体テスト
- 統合テスト

に加えて、

- ソースコードの静的解析(「コード解析」参照)
- リリースパッケージの作成

等がある。

developブランチの更新をトリガーとして行われるCI項目は、長くても30分程度で完了できるようにするのが理想的である。一方で、機能開発の進捗とともに、この時間制限を超える項目が出てくる(フルビルドですら30分を超えることはめずらしくない)。

こういった場合に行われるのが、CI項目の分割である。以下のテーブルのように、実行タイミング毎にCI項目を分けることで効率の良い運用ができる。

実行タイミング	develop更新後	深夜	週末
実行時間	30分程度	深夜～翌朝	金曜日深夜～月曜日朝
ビルト	差分	フル	同左

実行タイミング	develop更新後	深夜	週末
単体テスト	差分	あり	同左
統合テスト	なし/20分以内程度	あり	あり(長期動作系)
静的解析	なし/20分以内程度	あり	同左
pgk作成	なし	あり	同左

CI項目の例

CIの環境として、

- CIサーバとしてJenkins
- Jenkinsのジョブ記述にbash
- コンパイラにg++
- ビルドツールにmake

を使用すると前提とする。この場合、

- 差分ビルド

```
> make -j
```

- フルビルド

```
> make clean  
> make -j
```

- 単体テスト

```
> make ut      # sanitizerをオンにしてビルドするとより効果的
```

- 統合テスト

```
> make it      # sanitizerをオンにしてビルドするとより効果的
```

- 静的解析

```
> make clang    # gccの他にclangでコンパイルすることで、clangの警告機能を使う  
> scan-build make # clangベースの静的解析ツール
```

をJenkinsジョブ記述用テキストボックスに記述すればよい(「[コード解析](#)」参照)。つまり、CIで重要なことはテスト等の項目をコマンド化することである(従って、ビルドや単体テストをコマンドによって駆動できないIDEを使用してはならない)。

まとめ

産業は、労働集約型と知識集約型に二分できる。一般に労働集約型産業の就労者の生産性には大差がなく、知識集約型産業の就労者の生産性には大きな差がある。ソフトウェア産業は知識集約型であるにもかかわらず、プログラマの月単価のような労働集約的な基準で生産性が語られることが慣行となっているが、これはプログラマの生産性に大差がないという考え方から発した誤りである。

ただし、以下のように前提することで、この誤りにも、ある程度の正当性が与えられる。

- プログラマの生産性の違いが発揮されるフェーズはプログラミングのみである。
- V字モデルでは、このフェーズの工数は全工数から見れば少ない。
- テストは、マウスをひたすらクリックするような手作業で行われるため、これは労働集約的な作業である。
- V字モデルでのこのフェーズの工数は、プログラミングの工数と同程度になる。

この章で解説した「ほとんどのテストはプログラミングにより自動化できる」ことを理解すれば、この前提が成り立たないことは明らかだろう。

このことに気づいている組織は知識集約型のアプローチで、気づかない組織は労働集約型のアプローチでソフトウェア開発を行うことになる。どちらの生産性が高いかを議論する余地はない。

こういったことをマネージャ、リーダはよく理解するべきだろう。

[演習-プロセス分類](#)

[演習-V字モデル](#)

[演習-アジャイル](#)

[演習-自動化](#)

[演習-単体テスト](#)

[演習-リファクタリングに付随する活動](#)

[演習-リファクタリング対象コード](#)

[演習-CI](#)

並行処理

03以前のC++は、並行処理に関して十分な機能を備えていなかったため、多くのソフトウェア開発ではOSネイティブなAPIを使用せざるを得なかった。11以降のC++は、下記するような言語拡張、STLの大幅な機能追加等により、この問題の軽減に成功している。

- ラムダ式
- std::future、std::async、std::promiseによるFutureパターンのサポート
- std::unique_lock、std::condition_variableによるイベント通知
- ロック機構のRAIIのサポート(「RAII(scoped guard)」参照)
- std::atomic等によるアトミック処理の簡易化

本章では、「自動統合テスト」で開発したref_async_r5を改善することによって、これらを用いたC++11での並行プログラミングを例示する。

なお、本来であればこの章は「デザインパターン」に含めるべきだろうが、サンプルコードの説明の都合上ここに掲載することとした。

この章の構成

ref_async_r5改善プログラムの要件

Controller
View
Model

非同期処理とその管理

TwoPhaseTaskIF(TwoPhaseTaskPtr)
Dispatcher
TwoPhaseTaskPtrキュー管理機構

ref_async_r6の構造

まとめ

ref_async_r5改善プログラムの要件

「自動統合テスト」で開発したref_async_r5は、少なくとも以下のような並行処理にまつわる問題を持っている。

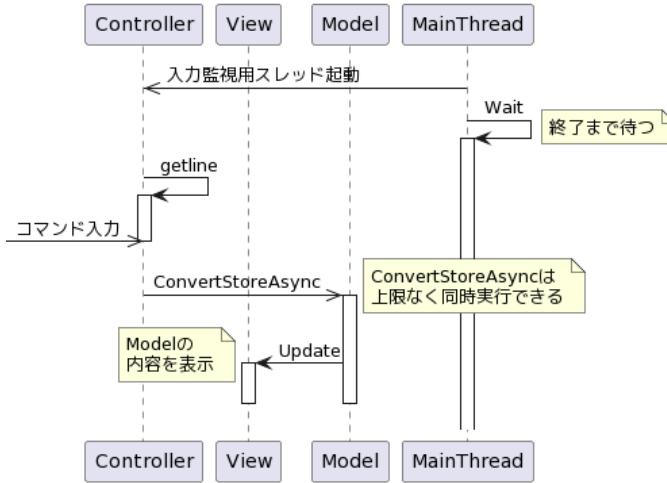
1. メインスレッドのメインループがstd::getline()でブロックしてしまうため、ネットワーク等の複数種類の同時入力には対応できない。
2. ref_async_r5のModel(ref_async_r5のModelはref_async_r3のModelと同じ)は2つ以上のスレッドを同時実行できない。

問題1に関しては、複数種類の同時入力に対応できる構造への改善のみを行う(本章の目的から逸脱するため、複数種類の入力への実際の対応は行わない)。問題2に関しては、Modelが開始できるスレッドの上限を持たないような改善を行う。この変更により、ref_async_r5では起こらなかつたViewからの出力が競合する問題が発生するため、その対処も行う。

以上を踏まえ、ref_async_r5改善プログラムは以下のようなものになる。

- プログラム名をref_async_r6(不適切だが、名前の継承とわかり易さを優先する)。
- ref_async_r6のModelは、同時に実行できるスレッド数の上限を持たない。
- ref_async_r5と同じ統合テスト項目(「自動統合テストのための仕様追加」参考)をパスしなければならない。
 - コマンド処理が終了したときの出力が競合してはならない。
 - ref_async_r5と同様にコマンド処理の順番はファースト・イン・ファースト・アウトとする。
- 複数種類の同時入力に対応できる構造にする(メインスレッドがstd::getline()でブロックしない構造)。

以上の条件を満たすref_async_r6の概念的なシーケンス図を下記する。



Controller

上記したように`ref_async_r6`は、メインスレッドで`std::getline()`を実行することができないため、Controllerがその内部スレッドから`std::getline()`を呼び出すことで、コマンド入力に対応することにする。

`std::getline()`により受信したコマンドのエラーコードは、`ref_async_r6`のexitコードになるため、そのエラーコードをController外部から取り出せる必要がある。その取り出しが簡単に実現できるため、スレッド生成には`std::thread()`ではなく、`std::async()`を使用する。

なお、`std::async()`の戻り値である`std::future<T>`のデストラクタは、管理対象スレッドの終了を待つため、そのスレッドが終了しない場合ハングアップを引き起こす。これに対処するためには、プログラムが終了手続きに入る前に(典型的には`exit()`を呼び出すか、`main()`が`return`する前に)スレッドを終了させる必要がある。このことは、旧来の`std::thread`を使用する場合、スレッドが終了しないことによって`thread::join()`が永久にリターンしないことと同様である。

以上のような条件から、Controllerは以下のようになる。

```
// @@@ example/ref_async_r6/controller.h 8

class Controller {
public:
    explicit Controller(Model& model, std::istream& is) : model_{model}, is_{is}, future_{()} {}

    /// @fn WatchInput
    /// @brief 入力を受け取るスレッドを起動する
    void WatchInput();

    /// @fn GetExitCode
    /// @brief 上記スレッドが終わるまで待ち合わせて、スレッドのリターンコードを返す。
    int GetExitCode();

private:
    std::optional<int> dispatch(std::string const& command);
    int watch_input(); // WatchInputが開始するスレッドの中身

    Model& model_;
    std::istream& is_;
    std::future<int> future_;
};
```

```
// @@@ example/ref_async_r6/controller.cpp 7

void Controller::WatchInput()
{
    future_ = std::async(std::launch::async, [this] { return watch_input(); });

    /// @fn watch_input
    /// @brief Controllerのメインループ。std::istreamからコマンド文字列を受け取る。
    /// @return ループ終了の原因がエラーであるならば非0。そうでなければ0。
    /// この値は、GetExitCode()により取得できる。
    int Controller::watch_input()
    {
        for (;;) {
            auto buffer = std::string{};

            if (std::getline(is_, buffer)) {
                if (auto exit_code = dispatch(buffer)) {
```

```

        return *exit_code;
    }
}

assert(false);
return 0;
}

/// @fn dispatch
/// @brief 受け取った文字列をパースして、ディスパッチする
/// @param std::istreamから受け取った文字列
/// @return プログラムを終了する場合、exitコードを返す
std::optional<int> Controller::dispatch(std::string const& command)
{
    switch (command.at(0)) {
    case '+':
        model_.ConvertStoreAsync(command.substr(1));
        return std::nullopt;
    case '.':
        // do nothing
        return std::nullopt;
    case '=':
        Dispatcher::Inst().Stop(); // プログラム終了処理
        return 0;
    ...
    }
}

int Controller::GetExitCode() { return future_.get(); }

```

上記ソースコードで使用しているDispatcherについては「[Dispatcher](#)」で解説する。

View

「出力が競合してはならない」という要件は、以下のどちらかの方法で対処できる。

1. std::istreamに対する一連の出力をロック(std::mutex)により排他制御する。
2. std::istreamへの出力をを行う一連の処理を特定の单ースレッドで行う(これにより処理が直列になる)。

`ref_async_r6`程度の単純なソフトウェアであれば上記1がシンプルな解となるが、GUI APIを使用するような、ある程度複雑なアプリケーションの実装では、上記2を選択するのが一般的である(ちなみにWindowsアプリケーションの処理系であるWPFでは、GUI操作はメインスレッドのみに限定されている)。

`ref_async_r6`では、上記1を選択し、「スレッドセーフでない処理(Viewの出力)を特定の单ースレッドで行うことにより、その競合を回避する」例を示す。

[MVC](#)の構造により、Viewの出力は、Modelの状態変更(メンバー変数の変更)をトリガーとして(通常はそのコンテキスト上で)行われるため、Viewの出力が競合するのであれば、その前段のModelのメンバ変数変更も競合する。Viewの改善では、この両競合を同時に回避することはできないため、「スレッドセーフでない処理を特定の单ースレッドで行うことにより、その競合を回避する」機構はView以外の場所で作るべきであると判断できる。よって、Viewには、ほぼ変更する必要がない。

```

// @@@ example/ref_async_r6/view.h 7

class View : public Observer {
public:
    explicit View(std::ostream& os) noexcept : os_{os} {}

private:
    virtual void update(Model const& model) override;

    std::ostream& os_;
};

// @@@ example/ref_async_r6/view.cpp 3

void View::update(Model const& model)
{
    for (auto const& str : model.GetStrings()) { // コメントを除き
        os_ << str << std::endl; // ref_async_r4/view.cppと同じ
    }
}

```

Model

Modelの要件は、以下のようなものである。

1. Controllerから渡された文字列をModel::ConvertStoreAsync()により非同期に処理する。
2. Model::ConvertStoreAsync()は渡された文字列をdo_heavy_algorithm()によって別の文字列に変換する。
3. 変換された文字列をModel::strings_に保存し、Model::notify()を呼び出す。
4. Model::notify()は、Observerを経由してView::Update()を呼び出す。

この動作の制約条件として、

1. Modelは、同時実行されるModel::ConvertStoreAsync()の個数の上限値を持たない(複数個のスレッドを管理できなければならない)。
2. Model::ConvertStoreAsync()の並行呼び出しにより生成されたそれぞれの文字列は、それに対応するModel::ConvertStoreAsync()の呼び出し順序のまま保存されなければならない。
3. ViewからのModelの状態表示は、別の状態表示と競合してはならない。

制約条件1から、Modelは複数個のスレッドを管理する必要がある(他のクラスに委譲しても良い)。制約条件2、3からModel::ConvertStoreAsync()で起動されたスレッドのコンテキスト上で、変換された文字列をModel::strings_に保存することや、View::Update()を呼び出すことはできない(「[View](#)」参照)。こうしなければ競合が発生してしまう。

このような制約を回避するため、スレッド管理を行う以下の2つのクラスを導入する。

- [TwoPhaseTaskIF\(TwoPhaseTaskPtr\)](#)
- [Dispatcher](#)

これらは「スレッドセーフでない処理を特定の单一スレッドで行うことにより、その競合を回避する」機構も内包している。

まずは、この2つのクラスを使用して改善したModelのソースコードを以下に示す。

```
// @@@ example/ref_async_r6/model.h 10

class Observer {
public:
    Observer() = default;
    void Update(Model const& model) { update(model); }
    virtual ~Observer() = default;

private:
    virtual void update(Model const& model) = 0;
};

class Model {
public:
    Model() = default;
    ~Model() = default;
    void ConvertStoreAsync(std::string&& input);
    void Attach(Observer& observer);
    void Detach(Observer& observer);
    std::vector<std::string> const& GetStrings() const noexcept { return strings_; }

private:
    void notify() const;

    std::vector<std::string> strings_{};
    std::list<Observer*> observers_{};
};
```

```
// @@@ example/ref_async_r6/model.cpp 5

void Model::ConvertStoreAsync(std::string&& input)
{
    TwoPhaseTaskPtr task
        = MakeTwoPhaseTaskPtr([str = std::move(input)] { return do_heavy_algorithm(str); },
                               [this](auto&& str) {
                                   strings_.emplace_back(std::move(str));
                                   notify();
                               });

    Dispatcher::Inst().Invoke(std::move(task));
}

void Model::Attach(Observer& observer) { observers_.emplace_back(&observer); }
void Model::Detach(Observer& detach)
{
    observers_.remove_if([&detach](Observer* observer) { return &detach == observer; });
}
```

```

void Model::notify() const
{
    for (auto* observer : observers_) {
        observer->Update(*this);
    }
}

```

Model::ConvertStoreAsync()を除いて、ref_async_r5で使用したModelとほぼ同じである。なお、上記TwoPhaseTaskPtrは、

```

// @@@ example/ref_async_r6/lib.h 13

/// @typedef TwoPhaseTaskPtr
/// @brief std::unique_ptr<TwoPhaseTaskIF>オブジェクトを便利に使うためのエイリアス。
using TwoPhaseTaskPtr = std::unique_ptr<TwoPhaseTaskIF>;

```

と定義されている。

非同期処理とその管理

Model::ConvertStoreAsync()の処理は、以下の2つから成立している。

1. Model::ConvertStoreAsync()で行う処理を2つのラムダ式にして、それらを引数にしてMakeTwoPhaseTaskPtr()を呼び出し、TwoPhaseTaskPtrオブジェクトを生成する。
2. そのTwoPhaseTaskPtrオブジェクトを引数にしてDispatcher::Invoke()を呼び出す。

そのソースコードから推測できるように、Model::ConvertStoreAsync()は、

- その処理の実行をTwoPhaseTaskPtrオブジェクトに委譲する。
- TwoPhaseTaskPtrオブジェクトの管理等をDispatcherに委譲する。

以下にその詳細を説明する。

TwoPhaseTaskIF(TwoPhaseTaskPtr)

ModelからDispatcherへ委譲されるラムダ式を管理するための基底クラスとして、以下のようなTwoPhaseTaskIFを定義する。

```

// @@@ example/ref_async_r6/lib.h 20

/// @class TwoPhaseTaskIF
/// @brief Dispatcherに代理実行するためのタスクを定義するためのクラスのインターフェース。
/// TwoPhaseTaskのベースクラス。
class TwoPhaseTaskIF {
public:
    TwoPhaseTaskIF() = default;

    void DoPreTask() { do_pre_task(); }
    bool IsPreTaskDone() const { return is_pre_task_done_; }
    void PreTaskDone() { is_pre_task_done_ = true; }
    bool DoPostTask() { return do_post_task(); }
    virtual ~TwoPhaseTaskIF() = default;

    ...

private:
    bool is_pre_task_done_{false};

    virtual void do_pre_task() = 0;
    virtual bool do_post_task() = 0;
};

```

TwoPhaseTaskIFから派生する以下のような具象クラスTwoPhaseTask<PRE, POST>を定義する。

```

// @@@ example/ref_async_r6/lib.h 99

/// @class TwoPhaseTask
/// @brief Dispatcherが管理するタスクを管理するTwoPhaseTaskIFの具象クラス
/// @tparam PRE 非同期に行われる処理を記述したラムダ式の型。重い処理を行う。
/// @tparam POST PREの実行の結果をModelに保存するラムダ式の型。
/// この処理は、スレッドセーフでないため、単一のスレッドで行う。
/// ブロックしたり、重い処理を行ってはならない。
template <typename PRE, typename POST>
class TwoPhaseTask final : public TwoPhaseTaskIF {
public:
    using Result = decltype(std::declval<PRE>());
    // PRE()の戻り値型

```

```

// POSTは、bool post(T)のような関数型でなければならない。
static_assert(std::is_same_v<bool, decltype(std::declval<POST>()(Result()))>);

TwoPhaseTask(PRE pre_task, POST post_task)
    : pre_task_{pre_task}, post_task_{post_task}, result_={}
{
}

virtual ~TwoPhaseTask() override = default;

private:
    /// @fn do_pre_task
    /// @brief PREを非同期実行し、終了をDispatcherに通知
    virtual void do_pre_task() override
    {
        result_ = std::async(std::launch::async, [this] {
            auto ret = Result{pre_task_()};
            Dispatcher::Inst().Notify(*this);
            return ret;
        });
    }

    /// @fn do_post_task
    /// @brief PREの戻り値を添えてPOSTを同期実行
    virtual bool do_post_task() override { return post_task_(result_.get()); }

    PRE          pre_task_;
    POST         post_task_;
    std::future<Result> result_;
};

```

すでに掲載したが、念のため再度TwoPhaseTaskPtrの定義を示す。

```

// @@@ example/ref_async_r6/lib.h 13

/// @typedef TwoPhaseTaskPtr
/// @brief std::unique_ptr<TwoPhaseTaskIF>オブジェクトを便利に使うためのエイリアス。
using TwoPhaseTaskPtr = std::unique_ptr<TwoPhaseTaskIF>;

```

TwoPhaseTaskPtrオブジェクト(TwoPhaseTaskIFオブジェクトとほぼ等価)の生成を行うファクトリ関数を以下のように定義する。

```

// @@@ example/ref_async_r6/lib.h 144

/// @fn MakeTwoPhaseTaskPtr
/// @brief TwoPhaseTaskIFオブジェクトを生成するファクトリ関数
/// @tparam PRE      TwoPhaseTaskのPRE
/// @tparam POST_BODY TwoPhaseTaskのPOSTの中身で、戻り値はvoid
template <typename PRE, typename POST_BODY>
TwoPhaseTaskPtr MakeTwoPhaseTaskPtr(PRE pre_task, POST_BODY post_body)
{
    using T      = decltype(std::declval<PRE>());
    auto post_task = [post_body](T& pre_result) {
        post_body(std::move(pre_result));
        return true;
    };

    return std::make_unique<TwoPhaseTask<PRE, decltype(post_task)>>(pre_task, post_task);
}

```

TwoPhaseTask<>は以下のような前提を持つ。

- コンストラクタの第1引数(pre_task)で渡されたラムダ式は、ある程度の期間、ブロックしてしまうような時間のかかる処理である可能性がある。
- コンストラクタの第2引数(post_task)で渡されたラムダ式は、短時間で処理できる。

DispatcherやTwoPhaseTaskPtrオブジェクトは以上の前提から、以下のような処理を行う。

- TwoPhaseTaskPtrオブジェクトのDoPreTask()が呼び出されると、新しいスレッドを起動し、そのコンテキスト上でpre_taskを実行し、その完了をDispatcherに通知する。
- 通知を受けたDispatcherは、メインスレッド上で呼び出されたDispatcher::ExecUntilStop()のコンテキスト上で、通知をしたTwoPhaseTaskPtrオブジェクトのDoPostTask()を呼び出す。
- TwoPhaseTaskPtrオブジェクトのDoPostTask()は、そのTwoPhaseTaskPtrオブジェクトのDoPreTask()で生成された結果を引数としてpost_taskを実行する(この処理はメインスレッドのコンテキスト上で順次処理されるため排他の必要がない)。

Dispatcher

Dispatcherは、[「TwoPhaseTaskIF\(TwoPhaseTaskPtr\)」](#)で説明した動作に加え、以下のような特徴、前提を持つ。

- Dispatcherは、[Singleton](#)として実装され、`ref_async_r6`のどこからでもそのpublic関数を呼び出すことができる。
- Dispatcher::ExecUntilStop()は、メインスレッド上でイベント待ちループを形成し、Dispatcher::Stop()が呼び出されると、そのループを抜ける。
- Dispatcher::Invoke()は、TwoPhaseTaskPtrオブジェクトを自分のキューにプッシュし、そのオブジェクトのDoPreTask()を呼び出す。
- Dispatcherは、TwoPhaseTaskPtrオブジェクトからのDoPreTask()完了イベント通知を受け、Dispatcher::ExecUntilStop()のコンテキスト上で、そのオブジェクトのDoPostTask()を呼び出す。この処理は特定の単一スレッド(メインスレッド)で実行される(直列化される)ため、スレッドセーフでない処理はここで処理されることを前提としている。

Dispatcherのクラス宣言、定義を下記する。

```
// @@@ example/ref_async_r6/lib.h 50

/// @class Dispatcher
/// @brief InvokeされたTwoPhaseTaskPtrの
///        * PreTaskを非同期に呼び出し、終了時自分に通知する。
///        * PreTask終了通知をトリガーに、PreTaskの結果を引数にしてPostTaskを
///          ExecUntilStop()のコンテキスト上で呼び出す。
///        ExecUntilStop()は、Stop()が呼び出されるまでリターンしない。
class Dispatcher {
public:
    static Dispatcher& Inst();

    /// @fn Invoke
    /// @brief TwoPhaseTaskPtrを登録してPreTaskを非同期実行。
    /// @param TwoPhaseTaskPtrオブジェクトのrvalue。
    void Invoke(TwoPhaseTaskPtr&&);

    /// @fn Stop
    /// @brief 登録されているTaskの処理が終わったら、ExecUntilStop()がリターンする。
    void Stop();

    /// @fn Notify
    /// @brief PreTaskがその終了を通知する
    void Notify(TwoPhaseTaskIF& task);

    /// @fn ExecUntilStop
    /// @brief 終了したPreTaskの対のPostTaskをPreTaskの戻り値を添えて呼び出す。
    void ExecUntilStop();

    ...

private:
    Dispatcher() = default;
    TwoPhaseTaskPtr pop_task();
    void push_task(TwoPhaseTaskPtr&& task, bool stop);

    std::mutex mutex_{};

    std::queue<TwoPhaseTaskPtr> two_phase_tasks_{};
    std::condition_variable pre_task_done_{};
    bool stopped_{false};
};
```

Dispatcherのメンバ関数定義は、次節に掲載する。

TwoPhaseTaskPtrキュー管理機構

これまでの説明やソースコードからわかるように、`ref_async_r6`の並行処理は、TwoPhaseTaskPtrのキュー管理機構によって実現されている。TwoPhaseTaskPtrのキュー管理機構は、下記表に示したDispatcher内部のSTLクラスにより実装されている。ここでは、その詳細について説明を行う。

型	インスタンス	役割
std::queue<TwoPhaseTaskPtr>	Dispatcher::two_phase_tasks_	TwoPhaseTaskPtrキュー
std::condition_variable	Dispatcher::pre_task_done_	イベント待ち、イベント通知
std::mutex	Dispatcher::mutex_	TwoPhaseTaskPtrキューの排他
std::unique_lock<std::mutex>	関数ローカル	イベント待ち解除時の排他

型	インスタンス	役割
std::lock_guard<std::mutex>	関数ローカル	mutex_のRAII

なお、std::condition_variableを用いたイベント待ちに、

```
std::condition_variable::wait(lock)
```

を使用する場合、「Spurious Wakeup」への対処が必要になるが、

```
std::condition_variable::wait(lock, 関数オブジェクト)
```

にはその対処が含まれているため、前者を非推奨とし、ここでは後者を使用する。

TwoPhaseTaskPtrキーー管理機構はDispatcher::Invoke()が呼び出されることを起点にして、以下のようなプッシュ処理を行う。

1. MakeTwoPhaseTaskPtr()により生成されたTwoPhaseTaskPtrオブジェクトを引数にして、Dispatcher::Invoke()が呼び出される。
2. Dispatcher::Invoke()はその仮引数であるTwoPhaseTaskPtrオブジェクトを引数にして、Dispatcher::push_task()を呼び出す。
3. Dispatcher::push_task()は、Dispatcher::mutex_によって排他されたセクションで(pop_task()との競合があり得る)、TwoPhaseTaskPtrオブジェクトをDispatcher::two_phase_tasks_へプッシュする(その後、そのオブジェクトのDoPreTask()を実行し、非同期処理を開始させる)。

```
// @@@ example/ref_async_r6/lib.cpp 13

void Dispatcher::Invoke(TwoPhaseTaskPtr&& task) { push_task(std::move(task), false); }

void Dispatcher::push_task(TwoPhaseTaskPtr&& task, bool stop)
{
    auto* st = task.get();
    assert(st != nullptr);
    {
        auto lock = std::lock_guard{mutex_};
        if (stoped_) {
            return;
        }
        stoped_ = stop;
        two_phase_tasks_.push(std::move(task));
    }
    st->DoPreTask();
}
```

TwoPhaseTaskPtrキーー管理機構は以下のようにしてTwoPhaseTaskPtrオブジェクトのポップ処理を行う。

1. Dispatcher::ExecUntilStop()は、Dispatcher::pop_task()の中で、Dispatcher::pre_task_done_.wait()によりイベント待ちを行う。
2. イベント待ち解除はDispatcher::Notify()中から、pre_task_done_.notify_all()が呼び出されることによって行われる。
3. 「pre_task_done_.notify_all()が呼び出され」且つ、「Dispatcher::two_phase_tasks_の先頭TwoPhaseTaskPtrオブジェクトの非同期処理が完了」していれば、Dispatcher::pop_task()のイベント待ち状態は解除される。
4. Dispatcher::pop_task()のイベント待ち解除時に発生するクリティカルセクションは、unique_lock<mutex>によって保護される(push_task()との競合があり得る)。
5. このクリティカルセクションの中で、Dispatcher::two_phase_tasks_からTwoPhaseTaskPtrオブジェクトがポップされることで、そのTwoPhaseTaskPtrオブジェクトはキーーの管理対象外となる(その後、そのTwoPhaseTaskPtrオブジェクトがスコープアウトした時点で自動deleteされる)。

```
// @@@ example/ref_async_r6/lib.cpp 43

void Dispatcher::ExecUntilStop()
{
    for (;;) {
        auto task = TwoPhaseTaskPtr{pop_task()};

        if (!task->DoPostTask()) {
            assert(two_phase_tasks_.empty());
            stoped_ = false;
            break;
        }
    }
}

void Dispatcher::Notify(TwoPhaseTaskIF& task)
{
    auto lock = std::lock_guard{mutex_};

    task.PreTaskDone();
    pre_task_done_.notify_all(); // pop_taskでのイベント待ち解除
}
```

```

TwoPhaseTaskPtr Dispatcher::pop_task()
{
    auto lock = std::unique_lock{mutex_};

    pre_task_done_.wait(lock, [this](){ noexcept { // イベント待ち
        return !two_phase_tasks_.empty() && two_phase_tasks_.front()->IsPreTaskDone();
    });

    // キューからのTwoPhaseTaskPtrオブジェクトのポップ
    auto task = TwoPhaseTaskPtr{std::move(two_phase_tasks_.front())};
    two_phase_tasks_.pop();

    return task;
}

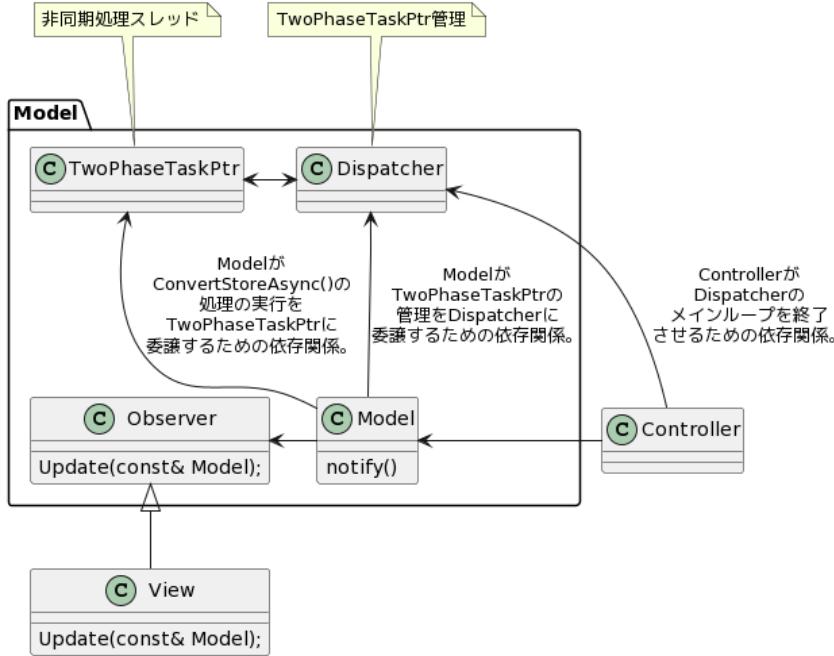
```

ref_async_r6の構造

ここまでで説明したように主に以下の5つのクラスがref_async_r6を構成する。

- Controller
- View
- Model
- TwoPhaseTaskIF(TwoPhaseTaskPtr)
- Dispatcher

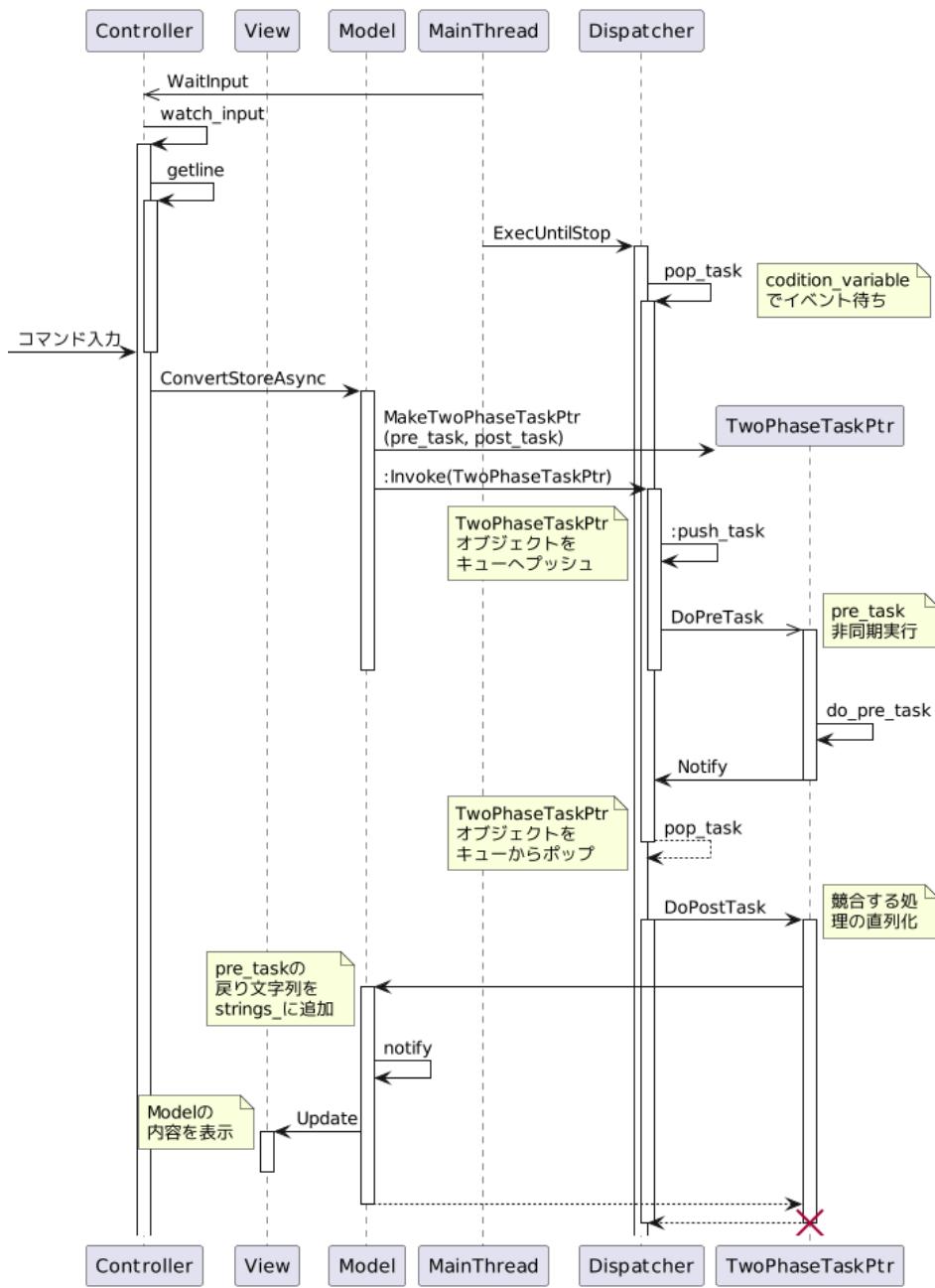
まずは、ref_async_r6のクラス図によりそれらの依存関係を示す。



すでに定義したref_async_r6の動作仕様を改めて下記する。

1. 標準入力から文字列を受け取り、
2. 文字列をパースし、コマンドと引数文字列に分離し、
3. 引数文字列を、時間のかかるアルゴリズムで別の文字列に変換し、
4. 変換文字列を記録し、
5. 記録した全ての変換文字列をstd::coutに出力する。

上記クラスを使い、この動作仕様を実現するシーケンス図を下記する。



まとめ

本章では、C++03から大幅に改善されたC++11以降の機能を使い、並行処理プログラミングについて解説したが、文脈に沿わなかったことにより、説明を割愛した事項もあるため、以下に改めて注意点をまとめる。

- C++11以降では、以下のような方法で競合回避のための排他制御ができる。
 1. `std::atomic`により、基本型の演算をアトミックに行う。
 2. `std::mutex`により、クリティカルセクションを保護する。
 3. スレッドセーフでない処理を特定の単一スレッドで行う。
- 一般に、組み込み型演算の競合回避には排他制御1が、オブジェクト操作等の競合回避には排他制御2が、それより複雑な競合回避(GUI API内部での競合回避等)には排他制御3が向いている。
- [注意] `volatile`はメモリアクセスの最適化を防ぐためのものであり、`std::atomic`は異なるコンテキストからのメモリのアクセス競合を防ぐものである。従って、全く用途が異なる。
- 排他制御1を使用し、スピンドロックを実装できる(「[固定長メモリプール](#)」参照)。
- 排他制御2を使用する場合、`RAII(scoped_guard)`を使用する。

- lock()/unlock()を直接ソースコードに書かない。代わりに std::lock_guard<std::mutex>を使用する。
- 排他制御3を使用する場合、アーキテクチャに大きな影響を与えるため、それが必要になるのであれば、できるだけ早期に対応する。
- より直感的な記述ができるため、std::thread(とstd::promiseやstd::packaged_task)よりstd::asyncを優先して使用する(「Future」参照)。
 - std::asyncを使用する場合、第1引数にはstd::launch::asyncを指定する。
 - std::thread、std::asyncから起動されるスレッドのエントリー関数オブジェクトは、プログラムが終了手続きに入る前にreturnさせる。そうしなければ、終了時にプログラムがクラッシュするか、ハングアップする。
 - std::asyncは処理を非同期に行うための機構であり、従ってその内部処理が排他制御のロック(std::mutex等)を行わないようにする。
- OSネイティブなAPIよりも、C++STLを優先して使用する。
 - イベント待ちには、std::condition_variableを優先して使用する(ビギループは、よほどの理由がない限り使用しない)。

並行処理はそれ自体が複雑であり、その実装やデバッグは難しい。一方でハードウェア性能を十分に引き出すソフトウェアの開発には絶大な効果を発揮する方法であるため、多くのプログラマにとって、避けて通ることはできない、時間を投資するに値する技術である。そのことを良く心得て、実践に当たってほしい。

テンプレートメタプログラミング

本章でのテンプレートメタプログラミングとは、下記の2つを指す。

- ジェネリックプログラミング
- メタプログラミング

C++においては、この2つはテンプレートを用いたプログラミングとなる。

ジェネリックプログラミングとは、具体的なデータ型に依存しない抽象的プログラミングであり、その代表的な成果物はSTLのコンテナやそれらを扱うアルゴリズム関数テンプレートである。

この利点は、

- i種の型
- j種のコンテナ
- k種のアルゴリズム

の開発を行うことを考えれば明らかである。

ジェネリックプログラミングが無ければ、コンテナの種類は $i \times j$ 個必要になり、それらに適用するアルゴリズム関数は、 $i \times j \times k$ 個必要になる。また、サポートする型の増加に伴いコンテナやアルゴリズム関数は指数関数的に増えて行く。C言語のqsort()のように強引なキャストを使い、この増加がある程度食い止めることはできるが、それによりコンパイラによる型チェックは無効化され、静的な型付け言語を使うメリットの多くを失うことになる。

メタプログラミングとは、

- ジェネリックのサポート
- 実行時コードの最適化
- 関数やクラスを生成するコードのプログラミング

のような目的で行われるテンプレートプログラミングの総称である。

ジェネリックプログラミングとメタプログラミングに明確な境界はない、また明確にしたところで大きなメリットはと思われるため、本章では、これらをまとめた概念であるテンプレートメタプログラミングとして扱い、ログ取得ライブラリやSTLを応用したNstdライブラリの実装を通して、これらのテクニックや、使用上の注意点について解説する。

この章の構成

ログ取得ライブラリの開発

要件

ログ取得ライブラリのインターフェース

パラメータパック

Loggerの実装

ユーザ定義型とそのoperator<<のname lookup

Ints_tのログ登録

Nstdライブラリの開発

Nstdライブラリを使用したリファクタリング

安全なvector

安全な配列型コンテナ

初期化子リストの副作用

メタ関数のテクニック

STLのtype_traits

is_void_xxxの実装

is_same_xxxの実装

AreConvertibleXxxの実装

関数の存在の診断

[Nstdライブラリの開発2](#)
[SafeArray2の開発](#)
[Nstd::SafeIndexの開発](#)
[Nstd::SafeIndexのoperator<<の開発](#)
[コンテナ用Nstd::operator<<の開発](#)

[ログ取得ライブラリの開発2](#)
[その他のテンプレートテクニック](#)
[ユニバーサルリファレンスとstd::forward](#)
[ジェネリックラムダ](#)
[クラステンプレートと継承の再帰構造](#)
[constexpr if文](#)
[意図しないname lookupの防止](#)
[Nstd::Type2Strの開発](#)
[静的な文字列オブジェクト](#)
[関数型をテンプレートパラメータで使う](#)

[注意点まとめ](#)

ログ取得ライブラリの開発

ここではログ取得ライブラリの開発を行う。

要件

ログ取得ライブラリの要件は、

- ソースコードの場所とそこで指示されたオブジェクトの値を文字列で保持する
- 後からそれらを取り出せる

ことのみとする。下記はその文字列を取り出した例である。

```
app/src/main.cpp: 96:Options
    cmd      : GenPkg
    in       :
    out      :
    recursive : true
    src_as_pkg: false
    ...

app/src/main.cpp: 51:start GenPkg

file_utils/ut/path_utils.cpp: 38:1
file_utils/ut/path_utils.cpp: 48:ut_data/app1
    ut_data/app1/mod1
    ut_data/app1/mod2

...
app/src/main.cpp:100:Exit:0
```

単純化のためログの番号やタイムスタンプのサポートはしない。また、実行速度や仕様メモリ量の制限等も本章の趣旨とは離れるため考慮しない。

ログ取得ライブラリのインターフェース

ログ取得コードにより、コードクローンが増えたり、主なロジックの可読性が下がったのでは、本末転倒であるため、下記のようにワントライナーで記述できるべきだろう。

```
LOGGER("start GenPkg", objA, objB, objC);
```

また、要件で述べた通り、ソースコード位置を特定できなければならないため、上記LOGGERは下記のような関数型マクロにならざるを得ない。

```
#define LOGGER(...) CppLoggerFunc(__FILE__, __LINE__, __VA_ARGS__)
```

CppClassLoggerFuncをクラス外の関数として実装した場合、ログ保持のための静的なオブジェクトが必要になる。これは避けるべきなので、「Singleton」で述べた構造を導入すると、

```
#define LOGGER(...) Logger::Inst().Set(__FILE__, __LINE__, __VA_ARGS__)
```

のような関数型マクロとなる。これがログ取得ライブラリの主なインターフェースとなる。

C言語プログラミングばかりをやりすぎて、視神経と手の運動神経が直結してしまった大脳レス・プログラマーは、

```
__VA_ARGS__
```

を見るとprintf(...)のような可変長引数を取る関数を思い浮かべる。「人は一昨日も行ったことを昨日も行ったという理由で、今日もそれを行う」という諺を思い出すと気持ちは分からなくもないが、C++ではprintf(...)のような危険な可変長引数を取る関数を作ってはならない。パラメータパックを使って実装するべきである。

パラメータパック

C++11で導入されたパラメータパックはやや複雑なシンタックスを持つため、まずは単純な例から説明する。

次のような単体テストをパスする関数テンプレートsumをパラメータパックで実装することを考える。

```
// @@@ example/template/parameter_pack_ut.cpp 26

ASSERT_EQ(1, sum(1));
ASSERT_EQ(3, sum(1, 2));
ASSERT_EQ(6, sum(1, 2, 3));
ASSERT_FLOAT_EQ(6.0, sum(1, 2.0, 3.0));
ASSERT_EQ(10, sum(1, 2, 3, 4));

...
ASSERT_EQ(55, sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10));
...
```

sumの要件は、

- 可変長引数を持つ
- 算術型の引数と戻り値を持つ
- すべての引数の和を返す

のようなものになるため、関数テンプレートsumは下記のように書ける。

```
// @@@ example/template/parameter_pack_ut.cpp 9

template <typename HEAD>
int sum(HEAD head)
{
    return head;
}

template <typename HEAD, typename... TAIL>
int sum(HEAD head, TAIL... tails)
{
    return head + sum(tails...);
}
```

1つ目の関数テンプレートsumは引数が一つの場合に対応する。2つ目の関数テンプレートsumは引数が2つ以上の場合に対応する。

2つ目の関数テンプレートsumのテンプレートパラメータ

```
typename... TAIL
```

がパラメータパックであり、0個以上の型が指定されることを意味する。これを関数の引数として表すシンタックスが

```
TAIL... tails
```

であり、同様に0個以上のインスタンスが指定されることを表している。

HEADとTAILより、2つ目のsumは1個以上の引数を取れることになるため、引数が1つの場合、どちらのsumを呼び出すかが曖昧になるよう思えるが、ベストマッチの観点から1つ目のsumが呼び出される。

sum(1, 2, 3)の呼び出し時のsumの展開を見てみることでパラメータパックの振る舞いを解説する。

この呼び出しへは、2つ目のsumにマッチする。従って下記のように展開される。

```
return 1 + sum(2, 3);
```

sum(2, 3)も同様に展開されるため、上記コードは下記のようになる。

```
return 1 + 2 + sum(3);
```

sum(3)は1つ目のsumにマッチするため、最終的には下記のように展開される。

```
return 1 + 2 + 3;
```

これで基本的な要件は満たしたが、このsumでは下記のようなコードもコンパイルできてしまう。

```
// @@@ example/template/parameter_pack_ut.cpp 43
```

```
ASSERT_EQ(2, sum(1, true, false));
```

これを認めるかどうかはsumの仕様次第だが、ここではこれらを認めないようにしたい。また、引数に浮動小数が与えられた場合でも、sumの戻り値の型がintなる仕様には問題がある。合わせてそれも修正する。

```
// @@@ example/template/parameter_pack_ut.cpp 53
```

```
template <typename HEAD>
auto sum(HEAD head)
{
    // std::is_sameの2パラメータが同一であれば、std::is_same<>::value == true
    static_assert(!std::is_same<HEAD, bool>::value, "arguemnt type must not be bool.");
    return head;
}

template <typename HEAD, typename... TAIL>
auto sum(HEAD head, TAIL... tails)
{
    // std::is_sameの2パラメータが同一であれば、std::is_same<>::value == true
    static_assert(!std::is_same<HEAD, bool>::value, "arguemnt type must not be bool.");
    return head + sum(tails...);
}
```

```
// @@@ example/template/parameter_pack_ut.cpp 83
```

```
// boolを除く算術型のみ認めるため、下記はコンパイルできない。
// ASSERT_EQ(2, sum(1, true, false));

auto i1 = sum(1);
auto i2 = sum(1, 2);

static_assert(std::is_same<int, decltype(i1)>::value); // 1の型はint
static_assert(std::is_same<int, decltype(i2)>::value); // 1 + 2の型はint

auto u1 = sum(1U);
auto u2 = sum(1U, 2);

static_assert(std::is_same<unsigned int, decltype(u1)>::value); // 1Uの型はunsigned int
static_assert(std::is_same<unsigned int, decltype(u2)>::value); // 1U + 2の型はunsigned int

auto f0 = sum(1.0, 1.2);
static_assert(std::is_same<double, decltype(f0)>::value);

// ただし、戻り型をautoにしたため、下記も認められるようになった。
// これに対しての対処は別の関数で行う。
auto str = sum(std::string{"1"}, std::string{"2"});

ASSERT_EQ(str, "12");
static_assert(std::is_same<std::string, decltype(str)>::value);
```

以上で示したようにパラメータパックにより、C言語での可変長引数関数では不可能だった引数の型チェックができるようになったため、C言語でのランタイムエラーがコンパイルエラーにできるようになった。

なお、上記コードで使用した`std::is_same`は、与えられた2つのテンプレートパラメータが同じ型であった場合、`value`を`true`で初期化するクラステンプレートであり、`type_traits`で定義されている（後ほど使用する`std::is_same_v`は`std::is_same<>::value`と等価な定数テンプレート）。この実装については、後ほど説明する。

[演習-パラメータパック](#)

パラメータパックの畳み込み式

上記したsumは、パラメータパックの展開に汎用的な再帰構造を用いたが、C++17で導入された畳み込み式を用い、以下の様に簡潔に記述することもできる。

```
// @@@ example/template/parameter_pack_ut.cpp 123

template <typename... ARGS>
auto sum(ARGS... args)
{
    return (args + ...); // 畳み込み式は()で囲まなければならない。
}
```

```
// @@@ example/template/parameter_pack_ut.cpp 134

ASSERT_EQ(1, sum(1));
ASSERT_EQ(3, sum(1, 2));
ASSERT_EQ(6, sum(1, 2, 3));
ASSERT_EQ(6.0, sum(1, 2.0, 3.0));
ASSERT_EQ(10, sum(1, 2, 3, 4));
ASSERT_EQ(55, sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10));
```

畳み込み式で使用できる演算子を以下に示す。

```
+ - * / % ^ & | = < > << >> += -= *= /= %= ^= &= |= <<= >>= == != <= >= && || , .* ->*
```

これらの演算子がオーバーロードである場合でも、畳み込み式は利用できる。

前から演算するパラメータパック

パラメータパックを使うプログラミングでは、上記したHEADとTAILによるリカーシブコールがよく使われるパターンであるが、これには後ろから処理されるという、微妙な問題点がある。

これまでのsumに代えて下記のようなproduct(掛け算)を考える。

```
// @@@ example/template/parameter_pack_ut.cpp 149

template <typename HEAD>
auto product(HEAD head)
{
    static_assert(!std::is_same_v<HEAD, bool>, "argument type must not be bool.");
    static_assert(std::is_arithmetic_v<HEAD>, "argument type must be arithmetic.");

    return head;
}

template <typename HEAD, typename... TAIL>
auto product(HEAD head, TAIL... tails)
{
    static_assert(!std::is_same_v<HEAD, bool>, "argument type must not be bool.");
    static_assert(std::is_arithmetic_v<HEAD>, "argument type must be arithmetic.");

    return head * product(tails...);
}
```

このコードの単体テストは、

```
// @@@ example/template/parameter_pack_ut.cpp 173

ASSERT_EQ(1, product(100, 0.1, 0.1));
```

のようになるだろうが、`std::numeric_limits<>::epsilon`を使用していないため（「浮動小数点型」参照）、このテストはパスしない。一方で、以下のテストはパスする。

```
// @@@ example/template/parameter_pack_ut.cpp 178

ASSERT_EQ(1, product(0.1, 0.1, 100));
```

一般に0.01の2進数表現は無限小数になるため、これを含む演算にはepsilon以下の演算誤差が発生する。前者単体テストでは、後ろから演算されるために処理の途中に0.01が現れるが、後者では現れないため、この誤差の有無が結果の差になる。

このような演算順序による微妙な誤差が問題になるような関数を開発する場合、演算は見た目の順序通りに行われた方が良いだろう。ということで、productを前から演算するように修正する。

```
// @@@ example/template/parameter_pack_ut.cpp 196
```

```

template <typename HEAD>
auto product(HEAD head)
{
    static_assert(!std::is_same_v<HEAD, bool>, "argument type must not be bool.");
    static_assert(std::is_arithmetic_v<HEAD>, "argument type must be arithmetic.");

    return head;
}

template <typename HEAD, typename HEAD2, typename... TAIL>
auto product(HEAD head, HEAD2 head2, TAIL... tails)
{
    static_assert(!std::is_same_v<HEAD, bool>, "argument type must not be bool.");
    static_assert(std::is_arithmetic_v<HEAD>, "argument type must be arithmetic.");

    return product(head * head2, tails...);
}

```

HEAD、TAILに加えHEAD2を導入することで、前からの演算を実装できる（引数が一つのproductに変更はない）。当然ながら、これにより、

```

// @@@ example/template/parameter_pack_ut.cpp 220

ASSERT_EQ(1, product(100, 0.1, 0.1));

```

はパスし、下記はパスしなくなる。

```

// @@@ example/template/parameter_pack_ut.cpp 225

ASSERT_EQ(1, product(0.1, 0.1, 100));

```

Loggerの実装

パラメータパックを使用したログ取得コードは以下のようになる。

```

// @@@ example/template/logger_0.h 47

#define LOGGER_P(...) Logging::Logger::Inst().Set(__FILE__, __LINE__)
#define LOGGER(...) Logging::Logger::Inst().Set(__FILE__, __LINE__, __VA_ARGS__)

```

予定していたものと若干違う理由は、`__VA_ARGS__`が1個以上の識別子を表しているからである。従って、通過ポイントのみをロギングしたい場合、`LOGGER_P()`を使うことになる。gcc拡張を使えば、`LOGGER_P`と`LOGGER`を統一できるが、そのようなことをすると別のコンパイラや、静的解析ツールが使用できなくなることがあるため、残念だが上記のように実装するべきである。

Loggerクラスの実装は、下記のようになる。

```

// @@@ example/template/logger_0.h 5

namespace Logging {
class Logger {
public:
    static Logger& Inst();
    static Logger const& InstConst() { return Inst(); }

    std::string Get() const; // ログデータの取得
    void Clear(); // ログデータの消去

    template <typename... ARGS> // ログの登録
    void Set(char const* filename, uint32_t line_no, ARGS const&... args)
    {
        oss_.width(32);
        oss_ << filename << ":";

        oss_.width(3);
        oss_ << line_no;

        set_inner(args...);
    }

    Logger(Logger const&) = delete;
    Logger& operator=(Logger const&) = delete;
};

private:
    void set_inner() { oss_ << std::endl; }

    template <typename HEAD, typename... TAIL>
    void set_inner(HEAD const& head, TAIL const&... tails)
    {
        oss_ << ":" << head;
    }
}

```

```

        set_inner(tails...);
    }

    Logger() {}
    std::ostringstream oss_{};
};

} // namespace Logging

```

すでに述べた通り、

- クラスはシングルトンにする
- パラメータパックにより可変長引数を実現する

ようにした。また、識別子の衝突を避けるために、名前空間Loggingを導入し、Loggerはその中で宣言した。

次に、どのように動作するのかを単体テストで示す。

```

// @@@ example/template/logger_0_ut.cpp 16

auto a = 1;
auto b = std::string{"b"};

LOGGER_P();           // (1)
LOGGER(5, "hehe", a, b); // (2)
auto line_num = __LINE__;

auto s = Logging::Logger::InstConst().Get();

auto exp = std::string{log_str_exp(__FILE__, line_num - 2, "\n")}; // (1)のログ
exp += log_str_exp(__FILE__, line_num - 1, ":5:hehe:1:b\n");      // (2)のログ
ASSERT_EQ(exp, s);

Logging::Logger::Inst().Clear(); // クリアの確認
ASSERT_EQ("", Logging::Logger::InstConst().Get());

```

行を含む出力の期待値をソースコードに直接書くと行増減のたびにそれらを修正する必要ある。期待値の一部を自動計算する下記コード(上記コードで使用)を単体テストに導入することで、そういう修正を避けている。

```

// @@@ example/template/logger_ut.h 4

inline std::string line_to_str(uint32_t line)
{
    if (line < 10) {
        return ":" ;
    }
    else if (line < 100) {
        return ":" ;
    }
    else if (line < 1000) {
        return ":" ;
    }
    else {
        assert(false); // 1000行を超える単体テストファイルを認めない
        return "" ;
    }
}

inline std::string log_str_exp(char const* filename_cstr, uint32_t line, char const* str)
{
    auto const filename = std::string(filename_cstr);
    auto const len     = 32 > filename.size() ? 32 - filename.size() : 0;
    auto       ret     = std::string(len, ' ');

    ret += filename;
    ret += line_to_str(line);
    ret += std::to_string(line);
    ret += str;

    return ret;
}

```

アプリケーションの開発では、下記のようなユーザが定義した名前空間とクラスを用いることがほとんどである。

```

// @@@ example/template/app_ints.h 12

namespace App {

class X {
public:

```

```

    X(std::string str, int num) : str_{std::move(str)}, num_{num} {}
    std::string ToString() const { return str_ + "/" + std::to_string(num_); }
    ...
};

} // namespace App

```

このApp::Xのインスタンスのログを取得できることも、当然Logging::Loggerの要件となる。従って、下記の単体テストはコンパイルでき、且つパスすることが必要になる。

```

// @@@ example/template/logger_0_ut.cpp 42

auto x = App::X{"name", 3};

LOGGER(1, x);
auto line_num = __LINE__;

auto s = Logging::Logger::InstConst().Get();

auto exp = std::string{log_str_exp(__FILE__, line_num - 1, ":1:name/3\n")};
ASSERT_EQ(exp, s);

```

Logging::Loggerのコードからオブジェクトのログを登録するためには、Logging::Logger::set_innerがコンパイルできなければならない。つまり、

```
std::ostream& operator<<(std::ostream&, ログ登録オブジェクトの型)
```

の実装が必要条件となる。App::Xでは下記のようなコードになる。

```

// @@@ example/template/app_ints.h 28

namespace App {

inline std::ostream& operator<<(std::ostream& os, X const& x) { return os << x.ToString(); }
} // namespace App

```

他の任意のユーザ定義型に対しても、このようにすることでログ登録が可能になる。

なお、ヒューマンリーダブルな文字列でその状態を表示できる関数をユーザ定義型に与えることは、デバッガを使用したデバッグ時にも有用である。

ユーザ定義型とそのoperator<<のname lookup

ここで、一旦Logging::Loggerの開発を止め、Logging::Logger::set_innerでのApp::operator<<のname lookupについて考えてみることにする。

ここまでで紹介したログ取得ライブラリやそれを使うユーザ定義型等の定義、宣言の順番は、

1. Logging::Logger
2. App::X
3. App::operator<<
4. 単体テスト(Logger::set_innerのインスタンス化される場所)

となっている。name lookupの原則に従い、App::Xの宣言は、App::operator<<よりも前に行われている。これを逆にするとコンパイルできない。しかし、Logging::Loggerは、後から宣言されたApp::operator<<を使うことができる。多くのプログラマは、これについて気づいていないか、その理由を間違っての認識している。

その認識とは、「テンプレート内の識別子のname lookupは、それがインスタンス化される時に行われる」というものであり、これにより「Logging::Loggerのname lookupは単体テスト内で行われる。それはApp::operator<<宣言後であるためコンパイルできる」と考えることができるが、two phase name lookupで行われるプロセスと反するため誤りである。

まずは、この認識の誤りを下記のコードで説明する。

```

// @@@ example/template/logger_0_ut.cpp 68

namespace App2 {
class X {
public:
    explicit X(std::string str, int num) : str_{std::move(str)}, num_{num} {}
    std::string ToString() const { return str_ + "/" + std::to_string(num_); }
    ...
};
} // namespace App2

namespace App3 { // App3をApp2にすればコンパイルできる
std::ostream& operator<<(std::ostream& os, App2::X const& x) { return os << x.ToString(); }
} // namespace App3

```

```
namespace {

TEST(Template, logger_0_X_in_AppX)
{
    Logging::Logger::Inst().Clear();

    auto x = App2::X{"name", 3};

    using namespace App3; // この記述は下記のエラーに効果がない

    LOGGER(1, x); // ここがコンパイルエラーとなる
    auto line_num = __LINE__;

    auto s = Logging::Logger::InstConst().Get();

    auto exp = std::string{log_str_exp(__FILE__, line_num - 1, ":1:name/3\n")};
    ASSERT_EQ(exp, s);
}

} // namespace
```

このコードは、もともとのコードの名前空間名をApp2とApp3にしただけのものである。もし、前記した「認識」の内容が正しいのであれば、このコードもコンパイルできるはずであるが（実際にApp3と書いた部分をApp2に書き換えればコンパイルできる）、実際には下記のようなエラーが発生する。

```
logger_0.h:37:21: error: no match for 'operator<<'  
(operand types are 'std::basic_ostream<char>' and 'const App2::X')  
37 |         oss_ << ":" << head;  
| ~~~~~^~~~~~
```

エラー内容からoperator<<が発見できることは明らかである。単体テスト内でのusing namespace App3はLogging::Logger::set_innerの宣言よりも後に書かれているため、このエラーを防ぐ効果はない。

Logging::Logger::set_innerの中でusing namespace App3とした上で、two phase name lookupの原則に従い、App2::XとApp3::operator<<をLogging::Loggerの宣言より前に宣言することで、ようやくコンパイルすることができる。

名前空間Appの例と名前空間App2、App3の例での本質的な違いは、「型Xとそのoperator<<が同じ名前空間で宣言されているかどうか」である。

名前空間Appの例の場合、型Xとそのoperator<<が同じ名前空間で宣言されているため、ADL(実引数依存探索)が働く。また、Logging::Logger::set_inner(x)はテンプレートであるため、`two_phase_name_lookup`が使用される。その結果、Logging::Logger::set_inner(x)でのname lookupの対象には、「Logging::Logger::set_inner(x)がインスタンス化される場所(単体テスト内でのLOGGER_PやLOGGERが使われている場所)より前方で宣言された名前空間App」も含まれる。こういったメカニズムにより、Logging::Logger::set_inner定義位置の後方で宣言されたApp::operator<<も発見できることになる。

一方で、名前空間App2、App3の例では、型XがApp2で宣言されているため、Logging::Logger::set_inner(x)でのname lookupの対象にApp3は含まれず、App3::operator<<は発見されない繰り返すが、インスタン化の場所直前でのusing nameには効果がない。

型Xとそのoperator<<を同じ名前空間で宣言することは本質的に重要なことであるが、名前空間を使用する場合、自然にそのような構造になるため、その重要性の理由を知る必要はないように思われる。しかし、次の例で示すようにこのメカニズムを知らずに解決することができないケースが存在する。

Ints tのログ登録

話題はログ取得ライブラリの開発に属る。アプリケーションの開発では、下記のように宣言された型エイリアスを使うことは珍しくない。

```
// @@@ example/template/app_ints.h 6

namespace App {
using Ints_t = std::vector<int>;
}
```

そのoperator \ll を下記のように定義したとする。

```
// @@@ example/template/logger_0_ut.cpp 109

namespace App {
std::ostream& operator<<(std::ostream& os, Ints_t const& ints)
{
    auto first = true;

    for (auto const i : ints) {
        if (!std::exchange(first, false)) {
            os << " ";
        }
        os << i;
    }
}
```

```

        }
        os << i;
    }

    return os;
}
} // namespace App

```

単体テストは下記のように書けるが、残念ながらコンパイルエラーになり、

```

// @@@ example/template/logger_0_ut.cpp 134

auto ints = App::Ints_t{1, 2, 3};

auto oss = std::ostringstream{};

oss << ints;
ASSERT_EQ("1, 2, 3", oss.str());

```

下記のようなエラーメッセージが表示される。

```

logger_0_ut.cpp:140:9: error: no match for ‘operator<<’
  (operand types are ‘std::ostringstream’ {aka ‘std::basic_ostringstream<char>’}
   and ‘App::Ints_t’ {aka ‘std::vector<int>’})
140 |     oss << ints;
|     ~~~ ~ ~~~~ ~~~
|     |     |
|     |     App::Ints_t {aka std::vector<int>}
|     std::ostringstream {aka std::basic_ostringstream<char>}

```

Ints_tはAppで定義されているが、実際の型はstdで定義されているため、intsの関連名前空間もstdであり、Appではない。その結果App::operator<<は発見できず、このようなエラーになった。

LOGGERからApp::operator<<を使う場合の単体テストは下記のようになるが、ADLによってLogging::Logger::set_inner(ints)内に導入される名前空間はstdのみであり、前記単体テスト同様にコンパイルできない。

```

// @@@ example/template/logger_0_ints_ut.h 8

auto ints = App::Ints_t{1, 2, 3};

LOGGER("Ints", ints);
auto line_num = __LINE__;

auto s = Logging::Logger::InstConst().Get();

auto exp = std::string{log_str_exp(__FILE__, line_num - 1, ":Ints:1, 2, 3\n")};
ASSERT_EQ(exp, s);

```

この解決方法は、

- operator<<をstd内で宣言する
- operator<<をグローバル名前空間内で宣言する
- operator<<をLogging内で宣言する
- Logging::Logger::set_inner(ints)内でusing namespace Appを行う
- Ints_tを構造体としてApp内に宣言する
- operator<<を使わない

のようにいくつか考えられる。以下では、順を追ってこれらの問題点について解説を行う。

operator<<をstd内で宣言する

ここで解決したい問題は、すでに示した通り、「ADLによってLogging::Logger::set_inner(ints)内に導入される名前空間はstdである」ことについて発生する。であれば、App内のoperator<<の宣言をstdで行えばコンパイルできるはずである。下記はその変更を行ったコードである。

```

// @@@ example/template/app_ints.h 6

namespace App {
using Ints_t = std::vector<int>;
}

// @@@ example/template/logger_0_std_ut.cpp 11

namespace std { // operator<<の定義をstdで行う
std::ostream& operator<<(std::ostream& os, App::Ints_t const& ints)

```

```

{
    auto first = true;

    for (auto const i : ints) {
        if (!std::exchange(first, false)) {
            os << ", ";
        }
        os << i;
    }

    return os;
}
} // namespace std

```

上記コードはtwo phase name lookup等の効果により、想定通りコンパイルできるが、`std`をユーザが拡張することは一部の例外を除き未定義動作を引き起こす可能性があり、たとえこのコードがうまく動作したとしても（実際、このコードはこのドキュメント作成時には正常動作している）、未来においてその保証はなく、このようなプログラミングは厳に避けるべきである。

operator<<をグローバル名前空間内で宣言する

すでに述べた通り、「ADLによって`Logging::Logger::set_inner(ints)`内に導入される名前空間は`std`のみである」ため、この関数の中での`name lookup`に使用される名前空間は、`std`、グローバル名前空間、`Logger`を宣言している`Logging`の3つである。

ここでは、下記のコードのようにグローバル名前空間内のoperator<<の宣言を試す。

```

// @@@ example/template/app_ints.h 6

namespace App {
using Ints_t = std::vector<int>;
}

// @@@ example/template/logger_0_global_ut.cpp 10

// グローバル名前空間
std::ostream& operator<<(std::ostream& os, App::Ints_t const& ints)
{
    auto first = true;

    for (auto const i : ints) {
        if (!std::exchange(first, false)) {
            os << ", ";
        }
        os << i;
    }

    return os;
}

```

このドキュメントで使用しているg++ではこのコードはコンパイルでき、動作も問題ないように思われるが、clang++では以下のエラーが発生し、コンパイルできない。

```

./logger_0.h:37:21: error: call to function 'operator<<' that is neither
visible in the template definition nor found by argument-dependent lookup
    oss_ << ":" << head;

```

この理由は「two phase name lookup」の後半で詳しく解説したので、ここでは繰り返さないが、このようなコードを使うと、コード解析ツール等が使用できなくなることがあるため、避けるべきである（「[scan-buildによる静的解析](#)」参照）。

多くのプログラマは、コードに問題があるとしても、それが意図通りに動くように見えるのであればその問題を無視する。今回のような難題に対しては、なおさらそのような邪悪な欲求に負けやすい。そのような観点でclang++が吐き出したエラーメッセージを眺めると、上記したメッセージの後に、下記のような出力を見つけるかもしれない。

```

logger_0_global_ut.cpp:13:15: note: 'operator<<' should be declared prior to the call site
std::ostream& operator<<(std::ostream& os, App::Ints_t const& ints)

```

clang++は「LOGGERの前にoperator<<を宣言せよ」と言っている。実際そうすれば、clang++でのコンパイルも通り、単体テストもパスする。しかし、それには下記のような問題がある。

- `operator<<(std::ostream& os, App::Ints_t const& ints)`という名前空間App口一カルな宣言をグローバル名前空間で行うことによって、グローバル名前空間を汚染してしまう（このコードは名前空間を正しく使うことに対しての割れ窓（「割れ窓理論」参照）になってしまいかもしれない）。
- 例示したコードでの`operator<<(std::ostream& os, App::Ints_t const& ints)`の定義は、単体テストファイル内にあったが、実際には何らかのヘッダファイル内で定義されることになる。その場合、ロガーのヘッダファイルよりも、そのヘッダファイルを先にインクルードしな

ければならなくなる。これは大した問題ではないように見えるが、ヘッダファイル間の暗黙の依存関係を生み出し将来の保守作業を難しくさせる。

以上述べた理由からこのアイデアを選択するべきではない。

operator<<をLogging内で宣言する

前節でのグローバル名前空間内のoperator<<の宣言はうまく行かなかったので、同様のことをLoggingで試す。

```
// @@@ example/template/app_ints.h 6

namespace App {
using Ints_t = std::vector<int>;
}

// @@@ example/template/logger_0_logging_ut.cpp 10

namespace Logging { // operator<<の定義をLoggingで行う
std::ostream& operator<<(std::ostream& os, App::Ints_t const& ints)
{
    auto first = true;

    for (auto const i : ints) {
        if (!std::exchange(first, false)) {
            os << ", ";
        }
        os << i;
    }

    return os;
}
} // namespace Logging
```

動作はするものの、当然ながら結果は「operator<<をグローバル名前空間内で宣言する」で述べた状況とほぼ同様であるため、このアイデアを採用することはできない。

Logging::Logger::set_inner(ints)内でusing namespace Appを行う

Logging::Logger::set_inner(ints)内でusing namespace Appを行えば、意図通りに動作させることができるが、App内のロギングは名前空間Loggingに依存するため、AppとLoggingが循環した依存関係を持つてしまう。また、LoggingはAppに対して上位概念であるため、依存関係逆転の原則(DIP)にも反する。よって、このアイデアを採用することはできない。

Ints_tを構造体としてApp内に宣言する

App::Ints_t用のoperator<<がLogging::Logger::set_inner内でname lookup出来ない理由は、これまで述べてきたようにApp::Inst_tの関連名前空間がAppではなく、stdになってしまふからである。

これを回避するためにはその原因を取り払えばよく、つまり、App::Inst_tの関連名前空間がAppになるようにすればよい。これを実現するために、次のコードを試してみる。

```
// @@@ example/template/logger_0_struct_ut.cpp 10

namespace App { // Ints_tの宣言はApp
struct Ints_t : std::vector<int> { // エイリアスではなく、継承を使う
    using vector::vector; // 継承コンストラクタ
};

// App内
std::ostream& operator<<(std::ostream& os, Ints_t const& ints)
{
    auto first = true;

    for (auto const i : ints) {
        if (!std::exchange(first, false)) {
            os << ", ";
        }
        os << i;
    }

    return os;
}
} // namespace App
```

上記のコードでは、

- App::Ints_tをstd::vectorからpublic継承
- using宣言によりstd::vectorのすべてのコンストラクタをApp::Ints_tに導入（「継承コンストラクタ」参照）

としているため、エイリアスで宣言されたInts_tと等価である。C++03では、継承コンストラクタが使えなかつたため、上記のような構造体を定義するためには、std::vectorのすべてのコンストラクタと等価なコンストラクタをApp::Ints_t内に定義することが必要で、実践的にはこのようなアイデアは使い物にならなかつたが、C++11での改善により、実践的なアイデアとして使用できるようになった。

実際、名前空間の問題もなく、すでに示した単体テストもパスするので有力な候補となるが、若干の「やりすぎ感」は否めない。

operator<<を使わない

色々なアイデアを試してみたが、これまでの議論ではこれといった解決方法を見つけることができなかつた。「バーニーの祈り」が言つてゐる通り、時にはどうにもならないことを受け入れることも重要である。LOGGERの中でname lookupできる、エイリアスApp::Ints_tのoperator<<の開発をあきらめ、ここでは一旦、下記のような受け入れがたいコードを受け入れることにする。

```
// @@@ example/template/app_ints.h 6

namespace App {
    using Ints_t = std::vector<int>;
}

// @@@ example/template/logger_0_no_put_to_ut.cpp 10

namespace App { // App::Ints_tのoperator<<とToStringをApp内で定義
    namespace { // operator<<は外部から使わない
        std::ostream& operator<<(std::ostream& os, Ints_t const& ints)
        {
            auto first = true;

            for (auto const i : ints) {
                if (!std::exchange(first, false)) {
                    os << ", ";
                }
                os << i;
            }

            return os;
        }
    } // namespace
}

// Ints_tオブジェクトをstd::stringに変換する
// この変換によりロガーに渡すことができる
std::string ToString(Ints_t const& inst)
{
    auto oss = std::ostringstream{};
    oss << inst;
    return oss.str();
} // namespace App
```

当然だが、恥を忍んで受け入れたコードにも単体テストは必要である。

```
// @@@ example/template/logger_0_no_put_to_ut.cpp 47

auto ints = App::Ints_t{1, 2, 3};

// ToStringのテスト
ASSERT_EQ("1, 2, 3", App::ToString(ints));

// LOGGERのテスト
LOGGER("Ints", App::ToString(ints));
auto line_num = __LINE__;

auto s = Logging::Logger::InstConst().Get();

auto exp = std::string{log_str_exp(__FILE__, line_num - 1, ":Ints:1, 2, 3\n")};
ASSERT_EQ(exp, s);
```

上記コードから明らかな通り、App::Ints_tのインスタンスをログ登録する場合、App::ToString()によりstd::stringへ変換する必要があり、残念なインターフェースとなつている。

Ints_tのログ登録のまとめ

製品開発では、満足できる仕様の関数やクラスが作れず、妥協せざるを得ないことはよくあることである。このような場合、将来、良いアイデアが見つかった時に備えて、妥協コードを簡単に修正できるようなレベルにした後、捲土重来を期してさっさと退却するのがベストである。ただし、漫然と過ごしても良いアイデアは浮かばない。時間を作り、関連書籍やウェブドキュメント等を読み、学習を継続する必要があることは言うまでもない。

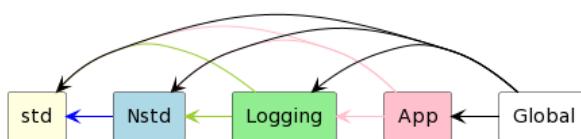
演習-名前空間による修飾不要なoperator<<

Nstdライブラリの開発

「operator<<を使わない」で導入したコードは、短いながらも汎用性が高い。このようなコードをローカルなファイルに閉じ込めてしまうと、コードクローンや、車輪の再発明による開発効率の低下につながることがある。

通常、プロジェクトの全ファイルから参照可能で且つ、プロジェクトの他のパッケージに非依存なパッケージを用意することで、このような問題を回避できる。

ここでは、そのようなパッケージをNstd(not standard library)とし、名前空間も同様に宣言する。そうした場合、この章の例題で使用している名前空間の依存関係は下記のようになる。



このように整理された依存関係は、「パッケージとその構成ファイル」でも述べた通り、大規模ソフトウェア開発においては特に重要であり、決して循環しないように維持しなければならない。

Nstdライブラリを使用したリファクタリング

すでに述べた通り、「operator<<を使わない」で導入したコードは、Nstdで定義するべきである。その場合、下記のようにさらに一般化するのが良いだろう。

```
// @@@ example/template/nstd_0.h 4

namespace Nstd {

    template <typename T>
    std::ostream& operator<<(std::ostream& os, std::vector<T> const& vec)
    {
        auto first = true;

        for (auto const& i : vec) {
            if (!std::exchange(first, false)) {
                os << ", ";
            }
            os << i;
        }

        return os;
    }

    template <typename T>
    std::string ToString(std::vector<T> const& vec)
    {
        auto oss = std::ostringstream{};

        oss << vec;

        return oss.str();
    }
} // namespace Nstd
```

その単体テストは下記のようになる。

```
// @@@ example/template/nstd_0_ut.cpp 13

auto const ints = App::Ints_t{1, 2, 3};

{
```

```

auto oss = std::ostringstream{};

using namespace Nstd;
oss << ints << 4;
ASSERT_EQ("1, 2, 34", oss.str());
}

{
    auto oss = std::ostringstream{};

    Nstd::operator<<(oss, ints) << 4; // 念のためこの形式でもテスト
    ASSERT_EQ("1, 2, 34", oss.str());
}

ASSERT_EQ("1, 2, 3", Nstd::ToString(ints));

```

勘のいい読者なら、このコードをLOGGERから利用することで、App::Ints_tのログ登録問題を解消できると思うかもしれない。実際その通りなのであるが、そうした場合、std::list等の他のコンテナや配列には対応できないという問題が残るため、以降もしばらくNstdの開発を続ける。

安全なvector

std::vector、std::basic_string、std::array等の配列型コンテナは、

- operator[]経由でのメンバアクセスについて範囲の妥当性をチェックしない
- 範囲のチェックが必要ならばat()を使用する

という仕様になっているが、ここではoperator[]にも範囲のチェックを行う配列型コンテナが必要になった場合について考える。

手始めにoperator[]にも範囲のチェックを行うstd::vector相当のコンテナSafeVectorを作ると、下記のコードのようになる。

```

// @@@ example/template/safe_vector_ut.cpp 9

namespace Nstd {

template <typename T>
struct SafeVector : std::vector<T> {
    using std::vector<T>::vector; // 継承コンストラクタ

    using base_type = std::vector<T>;
    using size_type = typename base_type::size_type;

    typename base_type::reference operator[](size_type i) { return this->at(i); }
    typename base_type::const_reference operator[](size_type i) const { return this->at(i); }
};

} // namespace Nstd

```

このコードで行ったことは、

- std::vectorからSafeVectorをpublic継承する
- 継承コンストラクタの機能を使い、std::vectorのコンストラクタをSafeVectorで宣言する
- std::vector::atを使い、SafeVector::operator[]を定義する

である。単体テストは下記のようになる。

```

// @@@ example/template/safe_vector_ut.cpp 29

{
    auto v = Nstd::SafeVector<int>(10); // ()での初期化
    ASSERT_EQ(10, v.size());
}
{
    auto const v = Nstd::SafeVector<int>{10};

    ASSERT_EQ(1, v.size());
    ASSERT_EQ(10, v[0]);
    ASSERT_THROW(v[1], std::out_of_range); // エクセプションの発生
}
{
    auto v = Nstd::SafeVector<std::string>{"1", "2", "3"};

    ASSERT_EQ(3, v.size());
    ASSERT_EQ(std::vector<std::string>{"1", "2", "3"}, v);
    ASSERT_THROW(v[3], std::out_of_range); // エクセプションの発生
}

```

```

auto const v = Nstd::SafeVector<std::string>{"1", "2", "3"};

ASSERT_EQ(3, v.size());
ASSERT_EQ(std::vector<std::string>{"1", "2", "3"}, v);
ASSERT_THROW(v[3], std::out_of_range); // エクセプションの発生
}

```

演習-std::arrayの継承

安全な配列型コンテナ

配列型コンテナはすでに述べたようにstd::vectorの他にすくなともstd::basic_string、std::arrayがあるため、それらにも範囲チェックを導入する。

std::basic_stringはstd::vectorとほぼ同様に下記のようになる。

```

// @@@ example/template/safe_vector_ut.cpp 61
namespace Nstd {

struct SafeString : std::string {
    using std::string; // 継承コンストラクタ

    using base_type = std::string;
    using size_type = typename base_type::size_type;

    typename base_type::reference operator[](size_type i) { return this->at(i); }
    typename base_type::const_reference operator[](size_type i) const { return this->at(i); }
};

} // namespace Nstd

```

std::stringはstd::basic_string<char>のエイリアスであるため、上記では、通常使われる形式であるstd::stringを継承したSafeStringを定義した。

この単体テストはSafeVectorの場合と同様に下記のようになる。

```

// @@@ example/template/safe_vector_ut.cpp 79
{
    auto s = Nstd::SafeString("0123456789");

    ASSERT_EQ(10, s.size());
    ASSERT_EQ("0123456789", s);
    ASSERT_THROW(s[10], std::out_of_range);
}

{
    auto const s = Nstd::SafeString(3, 'c'); // ()での初期化が必要

    ASSERT_EQ(3, s.size());
    ASSERT_EQ("ccc", s);
}

```

std::arrayでは少々事情が異なるが、std::vectorのコードパターンをそのまま適用すると下記のようになる。

```

// @@@ example/template/safe_vector_ut.cpp 100
namespace Nstd {

template <typename T, size_t N>
struct SafeArray : std::array<T, N> {
    using std::array<T, N>::array; // 継承コンストラクタ

    using base_type = std::array<T, N>;
    using size_type = typename base_type::size_type;

    typename base_type::reference operator[](size_type i) { return this->at(i); }
    typename base_type::const_reference operator[](size_type i) const { return this->at(i); }
};

} // namespace Nstd

```

ただし、この実装には問題がある。

```

// @@@ example/template/safe_vector_ut.cpp 121
auto sa_not_init = Nstd::SafeArray<int, 3>{};

ASSERT_EQ(3, sa_not_init.size());
ASSERT_THROW(sa_not_init[3], std::out_of_range);

```

上記コードでは、その問題が露見することはないが、以下のコードはコンパイルできない。

```
// @@@ example/template/safe_vector_ut.cpp 131

// std::initializer_listを引数とするコンストラクタが未定義
auto sa_init = Nstd::SafeArray<int, 3>{1, 2, 3};

// デフォルトコンストラクタがないため、未初期化
Nstd::SafeArray<int, 3> const sa_const;
```

std::arrayにはコンストラクタが明示的に定義されていないため、std::arrayにはデフォルトで自動生成される

- デフォルトコンストラクタ
- copyコンストラクタ
- moveコンストラクタ

以外のコンストラクタがないことが原因である。従って、SafeArray(std::initializer_list)が定義されず前述したようにコンパイルエラーとなる。

この問題に対処したのが以下のコードである。

```
// @@@ example/template/safe_vector_ut.cpp 145

namespace Nstd {

template <typename T, size_t N>
struct SafeArray : std::array<T, N> {
    using std::array<T, N>::array; // 繙承コンストラクタ
    using base_type = std::array<T, N>;

    template <typename... ARGS> // コンストラクタを定義
    SafeArray(ARGS... args) : base_type{args...}
    {

    }

    using size_type = typename base_type::size_type;

    typename base_type::reference operator[](size_type i) { return this->at(i); }
    typename base_type::const_reference operator[](size_type i) const { return this->at(i); }
};

} // namespace Nstd
```

上記コードで注目すべきは、パラメータパックをテンプレートパラメータとしたコンストラクタである。これにより、前例ではコンパイルすらできなかった下記のような初期化子リストを用いた単体テストが、このコンストラクタによりパスするようになった。

```
// @@@ example/template/safe_vector_ut.cpp 180
{
    auto sa_init = Nstd::SafeArray<int, 3>{1, 2, 3};

    ASSERT_EQ(3, sa_init.size());
    ASSERT_EQ(1, sa_init[0]);
    ASSERT_EQ(2, sa_init[1]);
    ASSERT_EQ(3, sa_init[2]);
    ASSERT_THROW(sa_init[3], std::out_of_range);
}

{
    auto const sa_string_const = Nstd::SafeArray<std::string, 5>{"1", "2", "3"};

    ASSERT_EQ(5, sa_string_const.size());
    ASSERT_EQ("1", sa_string_const[0]);
    ASSERT_EQ("2", sa_string_const[1]);
    ASSERT_EQ("3", sa_string_const[2]);
    ASSERT_EQ("", sa_string_const[3]);
    ASSERT_EQ("", sa_string_const[4]);
    ASSERT_THROW(sa_string_const[5], std::out_of_range);
}
```

この効果を生み出した上記を抜粋した下記のコードには解説が必要だろう。

```
// @@@ example/template/safe_vector_ut.cpp 154

template <typename... ARGS> // コンストラクタを定義
SafeArray(ARGS... args) : base_type{args...}
{}
```

一般にコンストラクタには「メンバ変数の初期化」と「基底クラスの初期化」が求められるが、SafeArrayにはメンバ変数が存在しないため、このコンストラクタの役割は「基底クラスの初期化」のみとなる。基底クラスstd::array(上記例ではbase_typeにエイリアスしている)には名前が非規定の配列メンバのみを持つため、これを初期化するためには初期化子リスト(「初期化子リストコンストラクタ」、「一様初

期化」参照)を用いるのが良い。

ということは、SafeArrayの初期化子リストコンストラクタには、「基底クラスstd::arrayに初期子リストを与えて初期化する」形式が必要になる。値を持つパラメータパックは初期化子リストに展開できるため、ここで必要な形式はパラメータパックとなる。これを実現したのが上記に抜粋したわずか数行のコードである。

初期化子リストの副作用

上記SafeArrayの初期化子リストコンストラクタは以下のようなコードを許可しない。

```
// @@ example/template/safe_vector_ut.cpp 212
{
    auto sa_init = Nstd::SafeArray<int, 3>{1.0, 2, 3};

    ASSERT_EQ(3, sa_init.size());
    ASSERT_EQ(1, sa_init[0]);
    ASSERT_EQ(2, sa_init[1]);
    ASSERT_EQ(3, sa_init[2]);
    ASSERT_THROW(sa_init[3], std::out_of_range);
}
```

このコードをコンパイルすると、

```
safe_vector_ut.cpp:147:41: error: narrowing conversion of
      'args#0' from 'double' to 'int' -Werror=narrowing]
147 |     SafeArray(ARGs... args) : base_type{args...}
      |           ^~~~
```

のようなエラーが出力されるが、

- double(上記例では1.0)をintに変換する際に縮小変換(narrowing conversion)が起こる
- 初期化子リストでの縮小変換は許可されない

が原因である。これは意図しない縮小変換によるバグを防ぐ良い機能だと思うが、ここではテンプレートメタプログラミングのテクニックを解説するため、あえてこのコンパイルエラーを起こさないSafeArray2を開発する(言うまでもないが、通常のソフトウェア開発では、縮小変換によるコンパイルエラーを回避するようなコードを書いてはならない)。

SafeArray2のコードは、

- STLのtype_traitsの使用
- テンプレートの特殊化
- メンバ関数テンプレートとオーバーロードによる静的ディスパッチ(コンパイル時ディスパッチ)
- SFINAE

等のメタ関数系のテクニックが必要になるため、まずはこれらを含めたテンプレートのテクニックについて解説し、その後SafeArray2を見していくことにする。

メタ関数のテクニック

本章で扱うメタ関数とは、型、定数、クラステンプレート等からなるテンプレート引数から、型、エイリアス、定数等を宣言、定義するようなクラステンプレート、関数テンプレート、定数テンプレート、エイリアステンプレートを指す(本章ではこれらをまとめて単にテンプレート呼ぶことがある)。

演習-エイリアステンプレート

STLのtype_traits

メタ関数ライブラリの代表的実装例はSTLのtype_traitsである。

ここでは、よく使ういくつかのtype_traitsテンプレートの使用例や解説を示す。

std::true_type/std::false_type

std::true_type/std::false_typeは真/偽を返すSTLメタ関数群の戻り型となる型エイリアスであるため、最も使われるテンプレートの一つである。

これらは、下記で確かめられる通り、後述するstd::integral_constantを使い定義されている。

```
// @@@ example/template/type_traits_ut.cpp 13

// std::is_same_vの2パラメータが同一であれば、std::is_same_v<> == true
static_assert(std::is_same_v<std::integral_constant<bool, true>, std::true_type>);
static_assert(std::is_same_v<std::integral_constant<bool, false>, std::false_type>);
```

それぞれの型が持つvalue定数は、下記のように定義されている。

```
// @@@ example/template/type_traits_ut.cpp 20

static_assert(std::true_type::value, "must be true");
static_assert(!std::false_type::value, "must be false");
```

これらが何の役に立つか直ちに理解することは難しいが、true/falseのメタ関数版と考えれば、追々理解できるだろう。

以下に簡単な使用例を示す。

```
// @@@ example/template/type_traits_ut.cpp 29

// 引数の型がintに変換できるかどうかを判定する関数
// decltypeの中でのみ使用されるため、定義は不要
constexpr std::true_type IsConvertibleToInt(int); // intに変換できる型はこちら
constexpr std::false_type IsConvertibleToInt(...); // それ以外はこちら
```

上記の単体テストは下記のようになる。

```
// @@@ example/template/type_traits_ut.cpp 40

static_assert(decltype(IsConvertibleToInt(1))::value);
static_assert(decltype(IsConvertibleToInt(1u))::value);
static_assert(!decltype(IsConvertibleToInt(""))::value); // ポインタはintに変換不可

struct ConvertibleToInt {
    operator int();
};

struct NotConvertibleToInt {};

static_assert(decltype(IsConvertibleToInt(ConvertibleToInt{}))::value);
static_assert(!decltype(IsConvertibleToInt(NotConvertibleToInt{}))::value);

// なお、IsConvertibleToInt()やConvertibleToInt::operator int()は実際に呼び出されるわけでは
// ないため、定義は必要なく宣言のみがあれば良い。
```

IsConvertibleToIntの呼び出しをdecltypeのオペランドにすることで、std::true_typeかstd::false_typeを受け取ることができる。

std::integral_constant

std::integral_constantは「テンプレートパラメータとして与えられた型とその定数から新たな型を定義する」クラステンプレートである。

以下に簡単な使用例を示す。

```
// @@@ example/template/type_traits_ut.cpp 62

using int3 = std::integral_constant<int, 3>

// std::is_same_vの2パラメータが同一であれば、std::is_same_v<> == true
static_assert(std::is_same_v<int, int3::value_type>);
static_assert(std::is_same_v<std::integral_constant<int, 3>, int3::type>);
static_assert(int3::value == 3);

using bool_true = std::integral_constant<bool, true>

static_assert(std::is_same_v<bool, bool_true::value_type>);
static_assert(std::is_same_v<std::integral_constant<bool, true>, bool_true::type>);
static_assert(bool_true::value == true);
```

また、すでに示したようにstd::true_type/std::false_typeを実装するためのクラステンプレートもある。

std::is_same

すでに上記の例でも使用したが、std::is_sameは2つのテンプレートパラメータが

- 同じ型である場合、std::true_type
- 違う型である場合、std::false_type

から派生した型となる。

以下に簡単な使用例を示す。

```
// @@@ example/template/type_traits_ut.cpp 99

static_assert(std::is_same<int, int>::value);
static_assert(std::is_same<int, int32_t>::value); // 64ビットg++/clang++
static_assert(!std::is_same<int, int64_t>::value); // 64ビットg++/clang++
static_assert(std::is_same<std::string, std::basic_string<char>>::value);
static_assert(std::is_same<typename std::vector<int>::reference, int&>::value);
```

また、C++17で導入されたstd::is_same_vは、定数テンプレートを使用し、下記のように定義されている。

```
// @@@ example/template/type_traits_ut.cpp 90

template <typename T, typename U>
constexpr bool is_same_v{std::is_same<T, U>::value};

// @@@ example/template/type_traits_ut.cpp 108

static_assert(is_same_v<int, int>);
static_assert(is_same_v<int, int32_t>); // 64ビットg++/clang++
static_assert(!is_same_v<int, int64_t>); // 64ビットg++/clang++
static_assert(is_same_v<std::string, std::basic_string<char>>);
static_assert(is_same_v<typename std::vector<int>::reference, int&>);
```

このような簡潔な記述の一般形式は、

```
T::value -> T_v
T::type -> T_t
```

のように定義されている(このドキュメントのほとんど場所では、簡潔な形式を用いる)。

第1テンプレートパラメータが第2テンプレートパラメータの基底クラスかどうかを判断する std::is_base_of を使うことで下記のように std::is_same の基底クラス確認することもできる。

```
// @@@ example/template/type_traits_ut.cpp 117

static_assert(std::is_base_of_v<std::true_type, std::is_same<int, int>>);
static_assert(std::is_base_of_v<std::false_type, std::is_same<int, char>>);
```

std::enable_if

std::enable_ifは、bool値である第1テンプレートパラメータが

- trueである場合、型である第2テンプレートパラメータをメンバ型typeとして宣言する。
- falseである場合、メンバ型typeを持たない。

下記のコードはクラステンプレートの特殊化を用いたstd::enable_ifの実装例である。

```
// @@@ example/template/type_traits_ut.cpp 124

template <bool T_F, typename T = void>
struct enable_if;

template <typename T>
struct enable_if<true, T> {
    using type = T;
};

template <typename T>
struct enable_if<false, T> { // メンバエイリアスtypeを持たない
};

template <bool COND, typename T = void>
using enable_if_t = typename enable_if<COND, T>::type;
```

std::enable_ifの使用例を下記に示す。

```
// @@@ example/template/type_traits_ut.cpp 148

static_assert(std::is_same_v<void, std::enable_if_t<true>>);
static_assert(std::is_same_v<int, std::enable_if_t<true, int>>);
```

実装例から明らかのように

- std::enable_if<true>::typeはwell-formed
- std::enable_if<false>::typeはill-formed

となるため、下記のコードはコンパイルできない。

```
// @@@ example/template/type_traits_ut.cpp 155
// 下記はill-formedとなるため、コンパイルできない。
static_assert(std::is_same_v<void, std::enable_if_t<false>>);
static_assert(std::is_same_v<int, std::enable_if_t<false, int>>);
```

std::enable_ifのこの特性と後述する[SFINAE](#)により、様々な静的ディスパッチを行うことができる。

std::conditional

std::conditionalは、bool値である第1テンプレートパラメータが

- trueである場合、第2テンプレートパラメータ
- falseである場合、第3テンプレートパラメータ

をメンバ型typeとして宣言する。

下記のコードはクラステンプレートの特殊化を用いたstd::conditionalの実装例である。

```
// @@@ example/template/type_traits_ut.cpp 164
template <bool T_F, typename, typename>
struct conditional;

template <typename T, typename U>
struct conditional<true, T, U> {
    using type = T;
};

template <typename T, typename U>
struct conditional<false, T, U> {
    using type = U;
};

template <bool COND, typename T, typename U>
using conditional_t = typename conditional<COND, T, U>::type;
```

std::conditionalの使用例を下記に示す。

```
// @@@ example/template/type_traits_ut.cpp 189
static_assert(std::is_same_v<int, std::conditional_t<true, int, char>>);
static_assert(std::is_same_v<char, std::conditional_t<false, int, char>>);
```

std::is_void

std::is_voidはテンプレートパラメータの型が

- voidである場合、std::true_type
- voidでない場合、std::false_type

から派生した型となる。

以下に簡単な使用例を示す。

```
// @@@ example/template/type_traits_ut.cpp 82
static_assert(std::is_void<void>::value);
static_assert(!std::is_void<int>::value);
static_assert(!std::is_void<std::string>::value);
```

is_void_xxxの実装

ここではstd::is_voidに似た以下のような仕様を持ついくつかのテンプレートis_void_xxxの実装を考える。

テンプレートパラメータ	戻り値
void	true
非void	false

それぞれのis_void_xxxは下記テーブルで示した言語機能を使用して実装する。

is_void_xxx	実装方法
is_void_f	関数テンプレートの特殊化
is_void_s	クラステンプレートの特殊化
is_void_sfinae_f	SFINAEと関数テンプレートのオーバーロード
is_void_sfinae_s	SFINAEとクラステンプレートの特殊化
is_void_ena_s	std::enable_ifによるSFINAEとクラステンプレートの特殊化
is_void_cond_s	std::conditionalと関数テンプレートの特殊化

なお、実装例をシンプルに保つため、理解の妨げとなり得る下記のような正確性(例外条件の対応)等のためのコードを最低限に留めた。

- テンプレートパラメータの型のチェック
- テンプレートパラメータの型からのポインタ/リファレンス/const/volatileの削除
- 戻り型からのconst/volatileの削除

これは、「テンプレートプログラミングでの有用なテクニックの解説」というここでの目的を見失わないとめの措置である。

is_void_fの実装

関数テンプレートの特殊化を使用したis_void_fの実装は以下のようにになる。

```
// @@@ example/template/is_void_ut.cpp 7

template <typename T>
constexpr bool is_void_f() noexcept
{
    return false;
}

template <>
constexpr bool is_void_f<void>() noexcept
{
    return true;
}

template <typename T>
constexpr bool is_void_f_v<is_void_f<T>();
```

単純なので解説は不要だろう。これらの単体テストは下記のようになる。

```
// @@@ example/template/is_void_ut.cpp 27

static_assert(!is_void_f_v<int>);
static_assert(!is_void_f_v<std::string>);
static_assert(is_void_f_v<void>);
```

関数テンプレートの特殊化には、

- 特殊化された関数テンプレートとそのプライマリテンプレートのシグネチャ、戻り値は一致しなければならない
- クラステンプレートのような部分特殊化は許可されない

のような制限があるため用途は限られるが、関数テンプレートはオーバーロードすることが可能である。

演習-SFINAEを利用しない関数テンプレートの特殊化によるis_void

is_void_sの実装

クラステンプレートの特殊化を使用したis_void_sの実装は以下のようになる。

```
// @@@ example/template/is_void_ut.cpp 35

template <typename T>
struct is_void_s {
    static constexpr bool value{false};
};

template <>
struct is_void_s<void> {
    static constexpr bool value{true};
};
```

```
template <typename T>
constexpr bool is_void_s_v{is_void_s<T>::value};
```

`is_void_f`と同様に単純なので解説は不要だろう。これらの単体テストは下記のようになる。

```
// @@@ example/template/is_void_ut.cpp 53

static_assert(!is_void_s_v<int>);
static_assert(!is_void_s_v<std::string>);
static_assert(is_void_s_v<void>);
```

演習-SFINAEを利用しないクラステンプレートの特殊化による`is_void`

`is_void_sfinae_f`の実装

SFINAEを使用した関数テンプレート`is_void_sfinae_f`の実装は以下のようになる。

```
// @@@ example/template/is_void_ut.cpp 61

namespace Inner_ {

// T == void
template <typename T>
constexpr auto is_void_sfinae_f_detector(void const* v, T const* t) noexcept
    -> decltype(t == v, bool{}) // T != voidの場合、t == vはill-formed
                                // T == voidの場合、well-formedでbool型生成
{
    return true;
}

// T != void
template <typename T>
constexpr auto is_void_sfinae_f_detector(void const*, T const*) noexcept
    -> decltype(sizeof(T), bool{}) // T != voidの場合、well-formedでbool型生成
                                // T == voidの場合、sizeof(T)はill-formed
{
    return false;
}
} // namespace Inner_

template <typename T>
constexpr bool is_void_sfinae_f() noexcept
{
    return Inner_::is_void_sfinae_f_detector(nullptr, static_cast<T*>(nullptr));
}

template <typename T>
constexpr bool is_void_sfinae_f_v{is_void_sfinae_f<T>()};
```

関数テンプレートである2つの`is_void_sfinae_f_detector`のオーバーロードにSFINAEを使用している。

1つ目の`is_void_sfinae_f_detector`では、

T	t = v の診断(コンパイル)
== void	well-formed
!= void	ill-formed

そのため、Tがvoidの時のみ`name lookup`の対象になる。

2つ目の`is_void_sfinae_f_detector`では、

T	sizeof(T)の診断(コンパイル)
== void	ill-formed
!= void	well-formed

そのため、Tが非voidの時のみ`name lookup`の対象になる。

`is_void_sfinae_f`はこの性質を利用し、

- T == voidの場合、1つ目の`is_void_sfinae_f_detector`が選択され、戻り値はtrue
- T != voidの場合、2つ目の`is_void_sfinae_f_detector`が選択され、戻り値はfalse

となる。念のため単体テストを示すと下記のようになる。

```
// @@@ example/template/is_void_ut.cpp 96

static_assert(!is_void_sfinae_f_v<int>);
static_assert(!is_void_sfinae_f_v<std::string>);
static_assert(is_void_sfinae_f_v<void>);
```

一般にファイル外部に公開するテンプレートは、コンパイルの都合上ヘッダファイルにその全実装を記述することになる。これは、本来外部公開すべきでない実装の詳細である `is_void_sfinae_f_detector` のようなテンプレートに関しては大変都合が悪い。というのは、外部から使用されたくない実装の詳細が使われてしまうことがあり得るからである。上記の例では、こういうことに備え「これは外部非公開である」ということを示す名前空間 `Inner_`（「名前空間」参照）を導入した。

関数テンプレートはクラステンプレート内にも定義することができるため、`is_void_sfinae_f` は下記のように実装することも可能である。この場合、名前空間 `Inner_` は不要になる。

```
// @@@ example/template/is_void_ut.cpp 105

template <typename T>
class is_void_sfinae_f {
    // U == void
    template <typename U>
    static constexpr auto detector(void const* v, U const* u) noexcept
        -> decltype(u = v, bool{}) // U != voidの場合、t = vはill-formed
                                    // U == voidの場合、well-formedでbool型生成
    {
        return true;
    }

    // U != void
    template <typename U>
    static constexpr auto detector(void const*, U const*) noexcept
        -> decltype(sizeof(U), bool{}) // U != voidの場合、well-formedでbool型生成
                                    // U == voidの場合、ill-formed
    {
        return false;
    }
public:
    static constexpr bool value{is_void_sfinae_f::detector(nullptr, static_cast<T*>(nullptr))};
};

template <typename T>
constexpr bool is_void_sfinae_f_v{is_void_sfinae_f<T>::value};
```

```
// @@@ example/template/is_void_ut.cpp 137
```

```
static_assert(!is_void_sfinae_f_v<int>);
static_assert(!is_void_sfinae_f_v<std::string>);
static_assert(is_void_sfinae_f_v<void>);
```

演習-SFINAEを利用した関数テンプレートの特殊化による`is_void`

is_void_sfinae_sの実装

SFINAEを使用したクラステンプレート `is_void_sfinae_s` の実装は以下のようになる。

```
// @@@ example/template/is_void_ut.cpp 146

namespace Inner_ {
template <typename T>
T*& t2ptr(); // 定義は不要
} // namespace Inner_

template <typename T, typename = void*&>
struct is_void_sfinae_s : std::false_type {
};

template <typename T>
struct is_void_sfinae_s<
    T,
    // T != voidの場合、ill-formed
    // T == voidの場合、well-formedでvoid*&生成
    decltype(Inner_::t2ptr<T>() = Inner_::t2ptr<void>())
> : std::true_type {
};

template <typename T>
constexpr bool is_void_sfinae_s_v{is_void_sfinae_s<T>::value};
```

1つ目のis_void_sfinae_sはプライマリテンプレートである。is_void_sfinae_sの特殊化がname lookupの対象の中に見つからなかった場合、これが使われる。

2つ目のis_void_sfinae_sは、上記を抜粋した下記のコード

```
// @@@ example/template/is_void_ut.cpp 162
// T != voidの場合、ill-formed
// T == voidの場合、well-formedでvoid*&生成
decltype(Inner_::t2ptr<T>()) = Inner_::t2ptr<void>()
```

がT == voidの時のみ、well-formedになり、このテンプレートは下記のようにインスタンス化される。

```
struct is_void_sfinae_s<void, void*&>
```

この形状はプライマリテンプレートの

- 第1パラメータにvoidを与える
- 第2パラメータには何も与えない(デフォルトのまま)

とした場合の、つまりプライマリテンプレートを

```
struct is_void_sfinae_s<void> // プライマリテンプレート
```

としてインスタンス化した場合と一致する。プライマリと特殊化が一致した場合、特殊化されたものがname lookupで選択される。

T != voidの場合、2つ目のis_void_sfinae_sはill-formedになり、name lookupの対象から外れるため、プライマリが選択される。

以上をまとめると、

T	is_void_sfinae_sの基底クラス
== void	std::true_type
!= void	std::false_type

となる。以下の単体テストによって、このことを確かめることができる。

```
// @@@ example/template/is_void_ut.cpp 179
static_assert(!is_void_sfinae_s_v<int>);
static_assert(std::is_base_of_v<std::false_type, is_void_sfinae_s<int>>);

static_assert(!is_void_sfinae_s_v<std::string>);
static_assert(std::is_base_of_v<std::false_type, is_void_sfinae_s<std::string>>);

static_assert(is_void_sfinae_s_v<void>);
static_assert(std::is_base_of_v<std::true_type, is_void_sfinae_s<void>>);
```

上記コードのように「プライマリテンプレートのデフォルトパラメータ」と、

```
// @@@ example/template/is_void_ut.cpp 162
// T != voidの場合、ill-formed
// T == voidの場合、well-formedでvoid*&生成
decltype(Inner_::t2ptr<T>()) = Inner_::t2ptr<void>()
```

が「well-formedであった場合に生成される型」が一致することを利用した静的ディスパッチは、SFINAEとクラステンプレートの特殊化を組み合わせたメタ関数の典型的な実装パターンである。ただし、一般にはill-formedを起こすためにstd::enable_ifを使うことが多いため、「is_void_ena_sの実装」でその例を示す。

演習-SFINAEを利用したクラステンプレートの特殊化によるis_void

is_void_ena_sの実装

std::enable_ifによるSFINAEとクラステンプレートの特殊化を使用したis_void_ena_sの実装は以下のようになる。

```
// @@@ example/template/is_void_ut.cpp 192
template <typename T, typename = void>
struct is_void_ena_s : std::false_type {
};

template <typename T>
struct is_void_ena_s<
    T,
    typename std::enable_if_t<is_void_f<T>()>
```

```

    > : std::true_type {
};

template <typename T>
constexpr bool is_void_ena_s_v{is_void_ena_s<T>::value};

```

この例では、「is_void_sfinae_sの実装」の

```

// @@@ example/template/is_void_ut.cpp 162

// T != voidの場合、ill-formed
// T == voidの場合、well-formedでvoid*&生成
decltype(Inner_::t2ptr<T>()) = Inner_::t2ptr<void>()

```

で示したSFINAEの処理を上記を抜粋した下記のコード

```

// @@@ example/template/is_void_ut.cpp 202

typename std::enable_if_t<is_void_f<T>()>

```

で行っている。std::enable_ifの値パラメータis_void_f<T>()は、「is_void_fの実装」で示したものである。

単体テストは、「is_void_sfinae_sの実装」で示したものとほぼ同様で、以下のようになる。

```

// @@@ example/template/is_void_ut.cpp 216

static_assert(!is_void_ena_s_v<int>);
static_assert(std::is_base_of_v<std::false_type, is_void_ena_s<int>>);

static_assert(!is_void_ena_s_v<std::string>);
static_assert(std::is_base_of_v<std::false_type, is_void_ena_s<std::string>>);

static_assert(is_void_ena_s_v<void>);
static_assert(std::is_base_of_v<std::true_type, is_void_ena_s<void>>);

```

is_void_cond_sの実装

std::conditionalと関数テンプレートの特殊化を使用したis_void_cond_sの実装は以下のようになる。

```

// @@@ example/template/is_void_ut.cpp 229
template <typename T>
struct is_void_cond_s : std::conditional_t<is_void_f<T>(), std::true_type, std::false_type> {
};

template <typename T>
constexpr bool is_void_cond_s_v{is_void_cond_s<T>::value};

```

std::conditionalの値パラメータis_void_f<T>()は、「is_void_fの実装」で示したものである。この例では、SFINAEもクラステンプレートの特殊化も使用していないが、下記単体テストからわかる通り、「is_void_sfinae_sの実装」と同じ機能を備えている。

```

// @@@ example/template/is_void_ut.cpp 240

static_assert(!is_void_cond_s_v<int>);
static_assert(std::is_base_of_v<std::false_type, is_void_cond_s<int>>);

static_assert(!is_void_cond_s_v<std::string>);
static_assert(std::is_base_of_v<std::false_type, is_void_cond_s<std::string>>);

static_assert(is_void_cond_s_v<void>);
static_assert(std::is_base_of_v<std::true_type, is_void_cond_s<void>>);

```

is_same_xxxの実装

ここではstd::is_same<T, U>に似た、以下のような仕様を持ついくつかのテンプレートis_same_xxxの実装を考える。

テンプレートパラメータ	戻り値
T == U	true
T != U	false

それぞれのis_same_xxxは下記テーブルで示された言語機能を使用して実装する。

is_same_xxx	実装方法
is_same_f	関数テンプレートのオーバーロード

is_same_xxx	実装方法
is_same_v	定数テンプレートの特殊化
is_same_s	クラステンプレートの特殊化
is_same_sfinae_f	SFINAEと関数テンプレート/関数のオーバーロード
is_same_sfinae_s	SFINAEとクラステンプレートの特殊化
is_same_templ	テンプレートテンプレートパラメータ
IsSameSomeOf	パラメータパックと再帰

is_same_fの実装

関数テンプレートのオーバーロードを用いたis_same_fの実装は以下のようになる。

```
// @@@ example/template/is_same_ut.cpp 9

template <typename T, typename U>
constexpr bool is_same_f_helper(T const*, U const*) noexcept
{
    return false;
}

template <typename T>
constexpr bool is_same_f_helper(T const*, T const*) noexcept
{
    return true;
}

template <typename T, typename U>
constexpr bool is_same_f() noexcept
{
    return is_same_f_helper(static_cast<T*>(nullptr), static_cast<U*>(nullptr));
}

template <typename T, typename U>
constexpr bool is_same_f_v{is_same_f<T, U>();}
```

すでに述べたように関数テンプレートの部分特殊化は言語仕様として認められておらず、

```
// @@@ example/template/is_same_ut.cpp 34

template <typename T, typename U>
constexpr bool is_same_f()
{
    return true;
}

template <typename T>
constexpr bool is_same_f<T, T>()
{
    return true;
}
```

上記のようなコードは、以下のようなコンパイルエラーになる(g++/clang++のような優れたコンパイラーを使えば、以下のメッセージのように簡単に問題点が理解できることもある)。

```
is_same_ut.cpp:35:32: error: non-class, non-variable partial specialization ‘
is_same_f<T, T>’ is not allowed
35 | constexpr bool is_same_f<T, T>()
```

関数テンプレートは部分特殊化が出来ない代わりに、同じ識別子を持つ関数や関数テンプレートとのオーバーロードができる。関数とのオーバーロードの場合、is_same_f_helper<T>()のようなテンプレートパラメータを直接使用した静的ディスパッチが出来ないため、常に型推測によるディスパッチが必要になる。

単体テストは以下のようになる。

```
// @@@ example/template/is_same_ut.cpp 52

static_assert(!is_same_f_v<int, void>);
static_assert(is_same_f_v<int, int>);
static_assert(!is_same_f_v<int, uint32_t>);
static_assert(is_same_f_v<std::string, std::basic_string<char>>);
```

is_same_vの実装

定数テンプレートの特殊化を用いたis_same_vの実装は以下のようになる。

```
// @@@ example/template/is_same_ut.cpp 61

template <typename T, typename U>
constexpr bool is_same_v{false};

template <typename T>
constexpr bool is_same_v<T, T>{true};
```

単純であるため、解説は不要だろう。単体テストは以下のようになる。

```
// @@@ example/template/is_same_ut.cpp 72

static_assert(!is_same_v<int, void>);
static_assert(is_same_v<int, int>);
static_assert(!is_same_v<int, uint32_t>);
static_assert(is_same_v<std::string, std::basic_string<char>>);
```

is_same_sの実装

クラステンプレートの特殊化を用いたis_same_sの実装は以下のようになる。

```
// @@@ example/template/is_same_ut.cpp 81

template <class T, class U>
struct is_same_s {
    static constexpr bool value{false};
};

template <class T>
struct is_same_s<T, T> {
    static constexpr bool value{true};
};

template <typename T, typename U>
constexpr bool is_same_s_v{is_same_s<T, U>::value};
```

「is_same_vの実装」と同様に単純であるため、解説は不要だろう。単体テストは以下のようになる。

```
// @@@ example/template/is_same_ut.cpp 99

static_assert(!is_same_s_v<int, void>);
static_assert(is_same_s_v<int, int>);
static_assert(!is_same_s_v<int, uint32_t>);
static_assert(is_same_s_v<std::string, std::basic_string<char>>);
```

is_same_sfinae_fの実装

SFINAEと関数テンプレート/関数のオーバーロードを用いたis_same_sfinae_f実装は以下のようになる。

```
// @@@ example/template/is_same_ut.cpp 108

namespace Inner_ {
template <typename T, typename U>
constexpr auto is_same_sfinae_f_detector(T const* t, U const* u) noexcept
    -> decltype(t = u, u = t, bool{}) // T != Uの場合、t = u, u = tはill-formed
                                         // T == Uの場合、well-formedでbool型生成
{
    return true;
}

constexpr bool is_same_sfinae_f_detector(...) noexcept { return false; }
} // namespace Inner_

template <typename T, typename U>
constexpr bool is_same_sfinae_f() noexcept
{
    return Inner_::is_same_sfinae_f_detector(static_cast<T*>(nullptr), static_cast<U*>(nullptr));
}

template <typename T, typename U>
constexpr bool is_same_sfinae_f_v{is_same_sfinae_f<T, U>()};
```

上記の抜粋である下記コードのコメントで示したように、

```
// @@@ example/template/is_same_ut.cpp 114

-> decltype(t = u, u = t, bool{}) // T != Uの場合、t = u, u = tはill-formed
// T == Uの場合、well-formedでbool型生成
```

T != Uの場合、この関数テンプレートはill-formedとなりname lookupの対象ではなくなる。その結果、関数is_same_sfinae_f_detectorが選択される。省略記号”...”(ellipsis)を引数とする関数は、そのオーバーロード群の中での最後の選択となるため、T == Uの場合は、関数テンプレートis_same_sfinae_f_detectorが選択される。

単体テストは以下のようにになる。

```
// @@@ example/template/is_same_ut.cpp 138

static_assert(!is_same_sfinae_f_v<int, void>);
static_assert(is_same_sfinae_f_v<int, int>);
static_assert(!is_same_sfinae_f_v<int, uint32_t>);
static_assert(is_same_sfinae_f_v<std::string, std::basic_string<char>>);
```

is_same_sfinae_sの実装

SFINAEとクラステンプレートの特殊化を用いたis_same_sfinae_sの実装は以下のようにになる。

```
// @@@ example/template/is_same_ut.cpp 147

namespace Inner_ {
template <typename T>
T*& t2ptr();
}

template <typename T, typename U, typename = T*&>
struct is_same_sfinae_s : std::false_type {
};

template <typename T, typename U>
struct is_same_sfinae_s<
    T, U,
    // T != Uの場合、ill-formed
    // T == Uの場合、well-formedでT*&生成
    decltype(Inner_::t2ptr<T>() = Inner_::t2ptr<U>(), Inner_::t2ptr<U>() = Inner_::t2ptr<T>())
> : std::true_type {
};

template <typename T, typename U>
constexpr bool is_same_sfinae_s_v{is_same_sfinae_s<T, U>::value};
```

「is_void_sfinae_sの実装」とほぼ同様であるため、解説は不要だろう。単体テストは以下のようにになる。

```
// @@@ example/template/is_same_ut.cpp 176

static_assert(!is_same_sfinae_s_v<int, void>);
static_assert(is_same_sfinae_s_v<int, int>);
static_assert(!is_same_sfinae_s_v<int, uint32_t>);
static_assert(is_same_sfinae_s_v<std::string, std::basic_string<char>>);
```

is_same_tempの実装

例えば、std::stringとstd::basic_string<T>が同じもしくは違う型であることを確認するためには、すでに示したis_same_sを使用し、

```
// @@@ example/template/is_same_ut.cpp 197

static_assert(is_same_s_v<std::string, std::basic_string<char>>);
static_assert(!is_same_s_v<std::string, std::basic_string<signed char>>);
```

のようにすればよいが、以下に示したコードのようにテンプレートテンプレートパラメータを使うことでも実装できる。

```
// @@@ example/template/is_same_ut.cpp 185

template <typename T, template <class...> class TEMPL, typename... ARGS>
struct is_same_temp : is_same_sfinae_s<T, TEMPL<ARGS...>> {
};

template <typename T, template <class...> class TEMPL, typename... ARGS>
constexpr bool is_same_temp_v{is_same_temp_v<T, TEMPL, ARGS...>::value};
```

上記のis_same_tempは、第2引数にクラステンプレート、第3引数以降にそのクラステンプレートの1個以上の引数を取ることができる。使用例を兼ねた単体テストは以下のようにになる。

```
// @@@ example/template/is_same_ut.cpp 202
static_assert(is_same_temp1_v<std::string, std::basic_string, char>);
static_assert(!is_same_temp1_v<std::string, std::basic_string, signed char>);
```

これを応用したエイリアステンプレート

```
// @@@ example/template/is_same_ut.cpp 209
template <typename T>
using gen_std_string = is_same_temp1_v<std::string, std::basic_string, T>;

template <typename T>
constexpr bool gen_std_string_v{gen_std_string<T>::value};
```

は与えられたテンプレートパラメータがstd::stringを生成するかどうかを判定することができる。

```
// @@@ example/template/is_same_ut.cpp 220
static_assert(gen_std_string_v<char>);
static_assert(!gen_std_string_v<signed char>);
```

演習-テンプレートテンプレートパラメータ

IsSameSomeOfの実装

IsSameSomeOfはこれまでの例とは少々異なり、

- 第1パラメータが第2パラメータ以降で指定された型の
 - どれかと同じであれば、std::true_typeから派生する
 - どれとも違えば、std::false_typeから派生する
- 2つの型の同一性の判定にはstd::is_sameを使用する
- 汎用性が高いため名前空間Nstdで定義し、命名はキャメルにする

のような特徴のを持つ。このようなIsSameSomeOfをパラメータパックと再帰を使用して実装すると以下のようになる。

```
// @@@ example/template/nstd_type_traits.h 10
namespace Nstd {
namespace Inner_ {

template <typename T, typename U, typename... Us>
struct is_same_some_of {
    static constexpr bool value{std::is_same_v<T, U> ? true : is_same_some_of<T, Us...>::value};
};

template <typename T, typename U>
struct is_same_some_of<T, U> {
    static constexpr bool value{std::is_same_v<T, U>};
};

template <typename T, typename... Us>
constexpr bool is_same_some_of_v{is_same_some_of<T, Us...>::value};
} // namespace Inner_

template <typename T, typename... Us>
struct IsSameSomeOf
: std::conditional_t<Inner_::is_same_some_of_v<T, Us...>, std::true_type, std::false_type> {};

template <typename T, typename... Us>
constexpr bool IsSameSomeOfV{IsSameSomeOf<T, Us...>::value};
} // namespace Nstd
```

IsSameSomeOfは、TがUsのいずれかと一致するかどうかの処理をInner_::is_same_some_ofに移譲する。

Usが1つだった場合、特殊化されたInner_::is_same_some_ofのvalueがstd::is_same::valueで初期化される。Usが複数だった場合、プライマリのInner_::is_same_some_ofは、IsSameSomeOfから渡されたパラメータパックUsを、UとパラメータパックUsに分割後、TとUをstd::is_sameで比較し、

- 同じ場合、valueはtrueで初期化される
- 違う場合、valueは再帰的に読み出されたInner_::is_same_some_of<T, Us...>::valueで初期化される

再帰的なInner_::is_same_some_of::valueの読み出しあは、IsSameSomeOfが受け取ったパラメータパックをひとつずつ左シフトしながら、それが1つになるまで(特殊化されたInner_::is_same_some_ofが使われるまで)、続けられる。

単体テストは以下のようになる。

```
// @@@ example/template/nstd_type_traits_ut.cpp 14

static_assert(!Nstd::IsSameSomeOfV<int, int8_t, int16_t, uint16_t>);
static_assert(Nstd::IsSameSomeOfV<int, int8_t, int16_t, uint16_t, int32_t>);
static_assert(Nstd::IsSameSomeOfV<int&, int8_t, int16_t, int32_t&, int32_t>);
static_assert(!Nstd::IsSameSomeOfV<int&, int8_t, int16_t, uint32_t&, int32_t>);
static_assert(Nstd::IsSameSomeOfV<std::string, int, char*, std::string>);
static_assert(!Nstd::IsSameSomeOfV<std::string, int, char*>);
```

演習-テンプレートパラメータを可変長にしたstd::is_same

AreConvertibleXxxの実装

std::is_convertible<FROM, TO>は、

- 型FROMが型TOに変換できる場合、std::true_typeから派生する
- 型FROMが型TOに変換できない場合、std::false_typeから派生する

のような仕様を持つテンプレートである。

ここでは、

- std::is_convertibleを複数のFROMが指定できるように拡張したNstd::AreConvertible
- 縮小無しでの型変換ができるかどうかを判定するAreConvertibleWithoutNarrowConv

の実装を考える。

AreConvertibleの実装

AreConvertibleの実装は以下のようになる。

```
// @@@ example/template/nstd_type_traits.h 42

namespace Nstd {
namespace Inner_ {

template <typename TO, typename FROM, typename... FROMs>
struct are_convertible {
    static constexpr bool value{
        std::is_convertible_v<FROM, TO> && are_convertible<TO, FROMs...>::value};
};

template <typename TO, typename FROM>
struct are_convertible<TO, FROM> {
    static constexpr bool value{std::is_convertible_v<FROM, TO>};
};

template <typename TO, typename... FROMs>
constexpr bool are_convertible_v{are_convertible<TO, FROMs...>::value};
} // namespace Inner_

template <typename TO, typename... FROMs>
struct AreConvertible
: std::conditional_t<Inner_::are_convertible_v<TO, FROMs...>, std::true_type, std::false_type> {};

template <typename TO, typename... FROMs>
constexpr bool AreConvertibleV{AreConvertible<TO, FROMs...>::value};
} // namespace Nstd
```

「IsSameSomeOfの実装」のコードパターンとほぼ同様であるため、解説は不要だろうが、

- パラメータパックの都合上、TOとFROMのパラメータの位置がstd::is_convertibleとは逆になる
- IsSameSomeOfでは条件の一つがtrueであればIsSameSomeOf::valueがtrueとなるが、AreConvertibleでは全条件がtrueとならない限り、AreConvertible::valueがtrueとならない

ので注意が必要である。

単体テストは以下のようになる。

```
// @@@ example/template/nstd_type_traits_ut.cpp 27
```

```

static_assert(Nstd::AreConvertibleV<int, int8_t, int16_t, int>);
static_assert(Nstd::AreConvertibleV<int, char, int, int>);
static_assert(!Nstd::AreConvertibleV<int, char*, int, int>);
static_assert(Nstd::AreConvertibleV<std::string, std::string, char*, char[3]>);
static_assert(!Nstd::AreConvertibleV<std::string, std::string, char*, int>);

```

AreConvertibleWithoutNarrowConvの実装

縮小無しの型変換ができるかどうかを判定するAreConvertibleWithoutNarrowConvは、 AreConvertibleと同じように実装できるが、 その場合、 AreConvertibleに対してstd::is_convertibleが必要になったように、 AreConvertibleWithoutNarrowConvに対しis_convertible_without_narrow_convが必要になる。

縮小無しでFROMからTOへの型変換ができるかどうかを判定するis_convertible_without_narrow_convは、 SFINAEと関数テンプレート/関数のオーバーライドを使用し以下のように実装できる。

```

// @@@ example/template/nstd_type_traits.h 75

namespace Nstd {
namespace Inner_ {

template <typename TO, typename FROM>
class is_convertible_without_narrow_conv {
    template <typename T = TO, typename U = FROM>
    static constexpr auto detector(T* t, U* u) noexcept
        // 縮小無しでFROMからTOへ変換可能な場合、 *t = T{*u}はwell-formed
        // 上記ではない場合、 *t = T{*u}はill-formed
        -> decltype(*t = T{*u}, bool{});
    {
        return true;
    }

    static constexpr bool detector(...) noexcept { return false; }
};

public:
    static constexpr bool value{is_convertible_without_narrow_conv::detector(
        static_cast<TO*>(nullptr), static_cast<FROM*>(nullptr))};
};

template <typename TO, typename FROM>
constexpr bool is_convertible_without_narrow_conv_v{
    is_convertible_without_narrow_conv<TO, FROM>::value};
} // namespace Inner_
} // namespace Nstd

```

AreConvertibleWithoutNarrowConvはNstdで定義するため、 その内部のみで用いる is_convertible_without_narrow_convはNstd::Inner_で定義している。

上記を抜粋した下記のコードは「縮小型変換を発生さる{}による初期化はill-formedになる」ことをSFINAEに利用している。

```

// @@@ example/template/nstd_type_traits.h 85

// 縮小無しでFROMからTOへ変換可能な場合、 *t = T{*u}はwell-formed
// 上記ではない場合、 *t = T{*u}はill-formed
-> decltype(*t = T{*u}, bool{})
```

単体テストは以下のようになる。

```

// @@@ example/template/nstd_type_traits_ut.cpp 39

static_assert(Nstd::Inner_::is_convertible_without_narrow_conv_v<int, int>);
static_assert(Nstd::Inner_::is_convertible_without_narrow_conv_v<int, int16_t>);
static_assert(!Nstd::Inner_::is_convertible_without_narrow_conv_v<int16_t, int>);
static_assert(Nstd::Inner_::is_convertible_without_narrow_conv_v<std::string, char*>);
static_assert(!Nstd::Inner_::is_convertible_without_narrow_conv_v<char*, std::string>);
```

is_convertible_without_narrow_convを利用したAreConvertibleWithoutNarrowConv の実装は以下のようになる。

```

// @@@ example/template/nstd_type_traits.h 108

namespace Nstd {
namespace Inner_ {

template <typename TO, typename FROM, typename... FROMs>
struct are_convertible_without_narrow_conv {
    static constexpr bool value{
        is_convertible_without_narrow_conv_v<TO, FROM>
        && are_convertible_without_narrow_conv_v<TO, FROMs...>::value};
```

```

};

template <typename TO, typename FROM>
struct are_convertible_without_narrow_conv<TO, FROM> {
    static constexpr bool value{is_convertible_without_narrow_conv_v<TO, FROM>};
};

template <typename TO, typename FROM, typename... FROMs>
constexpr bool are_convertible_without_narrow_conv_v{
    are_convertible_without_narrow_conv<TO, FROM, FROMs...>::value;
} // namespace Inner_


template <typename TO, typename FROM, typename... FROMs>
struct AreConvertibleWithoutNarrowConv
: std::conditional_t<Inner_::are_convertible_without_narrow_conv_v<TO, FROM, FROMs...>,
                     std::true_type, std::false_type> {
};

template <typename TO, typename FROM, typename... FROMs>
constexpr bool AreConvertibleWithoutNarrowConvV{
    AreConvertibleWithoutNarrowConv<TO, FROM, FROMs...>::value;
} // namespace Nstd

```

単体テストは以下のようにになる。

```

// @@@ example/template/nstd_type_traits_ut.cpp 47

static_assert(Nstd::AreConvertibleWithoutNarrowConvV<int, char, int16_t, uint16_t>);
static_assert(!Nstd::AreConvertibleWithoutNarrowConvV<int, char, int16_t, uint32_t>);
static_assert(Nstd::AreConvertibleWithoutNarrowConvV<std::string, char[5], char*>);
static_assert(Nstd::AreConvertibleWithoutNarrowConvV<double, float>);

// int8_t -> doubleは縮小型変換
static_assert(!Nstd::AreConvertibleWithoutNarrowConvV<double, float, int8_t>);

```

関数の存在の診断

Nstdライブラリの開発には関数の存在の診断が欠かせない。例えば、

- テンプレートパラメータに特定のメンバ関数がある場合、特殊化を作る
- テンプレートパラメータに範囲for文が適用できる場合にのみoperator<<を適用する
- テンプレートパラメータに適用できるoperator<<がすでにあった場合、自作operator<<を不活性化する

等、応用範囲は多岐にわたる。ここでは、上記の場合分けを可能とするようなメタ関数に必要なテクニックや、それらを使用したNstdのメタ関数の実装を下記のように示す。

- テンプレートパラメータである型が、メンバ関数void func()を持つかどうかの診断について、次の表のように実装を示す。

メタ関数名	メタ関数の目的
exists_void_func_sfinae_f	メンバ関数void func()を持つかどうかの判断
exists_void_func_sfinae_s	同上
exists_void_func_sfinae_s2	同上

- テンプレートパラメータに範囲for文ができるかどうかの診断について、次の表のように実装を示す。

メタ関数名	メタ関数の目的
exists_begin	std::begin(T)が存在するか否かの診断
exists_end	std::end(T)が存在するか否かの診断
IsRange	T const& tの時に、for(auto const& : t)ができるかどうかの診断

- テンプレートパラメータにoperator<<(put toと発音する)ができるかどうかの診断について、次の表のように実装を示す。

メタ関数名	メタ関数の目的
exists_put_to_as_member	std::ostream::operator<<(T)が存在するか否かの診断
exists_put_to_as_non_member	operator<<(std::ostream&, T)が存在するか否かの診断
ExistsPutTo	std::ostream & << Tができるかどうかの診断

- テンプレートパラメータがT[N]やC<T>の形式である時のTに、operator<<が適用できるかの診断については、Tの型を取り出す必要がある。そのようなメタ関数ValueTypeの実装を示す。

exists_void_func_sfinae_fの実装

「テンプレートパラメータである型が、メンバ関数void func()を持つかどうかを診断する」 exists_void_func_sfinae_f のSFINAЕと関数テンプレート/関数のオーバーロードを用いた実装は以下のようになる。

```
// @@@ example/template/exists_func_ut.cpp 13

namespace Inner_ {

template <typename T>
using exists_void_func_void =
    // メンバvoid func(void)があれば、voidを生成
    // メンバvoid func(void)がなければ、ill-formed
    typename std::enable_if_t<std::is_same_v<decltype(std::declval<T>().func()), void>>;
} // namespace Inner_

template <typename T, typename = Inner_::exists_void_func_void<T>>
constexpr bool exists_void_func_sfinae_f(T) noexcept
{
    return true;
}

constexpr bool exists_void_func_sfinae_f(...) noexcept { return false; }
```

decltypeの中での関数呼び出しは、実際には呼び出されず関数の戻り値の型になる。上記の抜粋である下記のコードはこの性質を利用してSFINAЕによる静的ディスパッチを行っている。

```
// @@@ example/template/exists_func_ut.cpp 20

// メンバvoid func(void)があれば、voidを生成
// メンバvoid func(void)がなければ、ill-formed
typename std::enable_if_t<std::is_same_v<decltype(std::declval<T>().func()), void>>;
```

単体テストは以下のようになる。

```
// @@@ example/template/exists_func_ut.cpp 40

// テスト用クラス
struct X {
    void func();
};

struct Y {
    int func();
};

struct Z {
private:
    void func(); // privateなvoid func()は外部からは呼び出せない
};
```



```
// @@@ example/template/exists_func_ut.cpp 60

static_assert(!exists_void_func_sfinae_f(int{}));
static_assert(exists_void_func_sfinae_f(X{}));
static_assert(!exists_void_func_sfinae_f(Y{}));
static_assert(!exists_void_func_sfinae_f(Z{}));
```

exists_void_func_sfinae_sの実装

「テンプレートパラメータである型が、メンバ関数void func()を持つかどうかを診断」する exists_void_func_sfinae_s のSFINAЕとクラステンプレートの特殊化を用いた実装は以下のようになる。

```
// @@@ example/template/exists_func_ut.cpp 69

template <typename T, typename U = void>
struct exists_void_func_sfinae_s : std::false_type {
};

template <typename T>
struct exists_void_func_sfinae_s<T>,
    // メンバvoid func()が呼び出せれば、voidを生成
    // メンバvoid func()が呼び出せなければ、ill-formed
    decltype(std::declval<T>().func())
```

```

    > : std::true_type {
};

template <typename T>
constexpr bool exists_void_func_sfinae_s_v{exists_void_func_sfinae_s<T>::value};

```

exists_void_func_sfinae_fとほぼ等しいSFINAEを利用したクラステンプレートの特殊化により、静的ディスパッチを行っている。

単体テストは以下のようになる。

```

// @@@ example/template/exists_func_ut.cpp 91

static_assert(!exists_void_func_sfinae_s_v<int>);
static_assert(exists_void_func_sfinae_s_v<X>);
static_assert(!exists_void_func_sfinae_s_v<Y>);
static_assert(!exists_void_func_sfinae_s_v<Z>);

```

exists_void_func_sfinae_s2の実装

exists_void_func_sfinae_sとほぼ同様の仕様を持つexists_void_func_sfinae_s2の

- SFINAE
- メンバ関数テンプレート/メンバ関数のオーバーロード
- メンバ関数へのポインタ

を用いた実装は以下のようになる。

```

// @@@ example/template/exists_func_ut.cpp 100

template <typename T>
class exists_void_func_sfinae_s2 {

    // メンバvoid func()が呼び出せれば、メンバ関数テンプレートはtrueを返す
    // メンバvoid func()が呼び出せなければ、ill-formed
    template <typename U, void (U::*())() = &U::func>
    static constexpr bool detector(U*) noexcept
    {
        return true;
    }

    static constexpr bool detector(...) noexcept { return false; }

public:
    static constexpr bool value{exists_void_func_sfinae_s2::detector(static_cast<T*>(nullptr))};
};

template <typename T>
constexpr bool exists_void_func_sfinae_s2_v{exists_void_func_sfinae_s2<T>::value};

```

前2例とは異なり、上記の抜粋である下記コードのように、メンバ関数へのポインタを使用しSFINAEを実装している。

```

// @@@ example/template/exists_func_ut.cpp 105

// メンバvoid func()が呼び出せれば、メンバ関数テンプレートはtrueを返す
// メンバvoid func()が呼び出せなければ、ill-formed
template <typename U, void (U::*())() = &U::func>
static constexpr bool detector(U*) noexcept
{
    return true;
}

```

あまり応用範囲が広くない方法ではあるが、decltypeを使っていないのでC++03コンパイラにも受け入れられるメリットがある。

exists_void_func_sfinae_fと同じテスト用クラスを用いた単体テストは以下のようになる。

```

// @@@ example/template/exists_func_ut.cpp 129

static_assert(!exists_void_func_sfinae_s2_v<int>);
static_assert(exists_void_func_sfinae_s2_v<X>);
static_assert(!exists_void_func_sfinae_s2_v<Y>);
static_assert(!exists_void_func_sfinae_s2_v<Z>);

```

演習-メンバ関数の存在の診断

exists_begin/exsits_endの実装

「テンプレートパラメータTに対して、`std::begin(T)`が存在するか否かの診断」をする`exists_begin`の実装は、「[exists_void_func_sfinae_sの実装](#)」で用いたパターンのメンバ関数を非メンバ関数に置き換えて使えば以下のようにになる。

```
// @@@ example/template/exists_func_ut.cpp 140

template <typename, typename = void>
struct exists_begin : std::false_type {
};

template <typename T>
struct exists_begin<T, std::void_t<decltype(std::begin(std::declval<T>()))>> : std::true_type {
};

template <typename T>
constexpr bool exists_begin_v{exists_begin<T>::value};
```

上記で使用した`std::void_t`は、テンプレートパラメータが

- ill-formedならばill-formedになる
- well-formedならvoidを生成する

テンプレートである。

下記単体テストでは問題ないように見えるが、

```
// @@@ example/template/exists_func_ut.cpp 156

static_assert(exists_begin_v<std::string>);
static_assert(!exists_begin_v<int>);
static_assert(exists_begin_v<int const[3]>);
```

下記の単体テストは`static_assert`がフェールするためコンパイルできない。

```
// @@@ example/template/exists_func_ut.cpp 166

// 以下が問題
static_assert(exists_begin_v<int[3]>);
```

理由は、

```
std::declval<int[3]>()
```

の戻り型が配列型のlvalueである”`int (&) [3]`“となり、これに対応する`std::begin`が定義されていないためである。

これに対処する方法方はいくつかあるが、すべての配列は常に`std::begin`の引数になれるために気づけば、テンプレートパラメータが配列か否かで場合分けしたクラステンプレートの特殊化を使い、下記のように実装できることにも気付けるだろう。

```
// @@@ example/template/exists_func_ut.cpp 183

template <typename, typename = void>
struct exists_begin : std::false_type {
};

// Tが非配列の場合の特殊化
template <typename T>
struct exists_begin<T,
    typename std::enable_if_t<!std::is_array_v<T>,
    std::void_t<decltype(std::begin(std::declval<T>()))>>>
: std::true_type {
};

// Tが配列の場合の特殊化
template <typename T>
struct exists_begin<T, typename std::enable_if_t<std::is_array_v<T>>> : std::true_type {
};

template <typename T>
constexpr bool exists_begin_v{exists_begin<T>::value};
```

2個目の`exists_begin`はTが配列でない場合、3個目の`exists_begin`はTが配列ある場合にそれぞれが対応しているが、複雑すぎて何とも醜い。ということで、このコードは却下して、別のアイデアを試そう。

テンプレートパラメータが配列である場合でも、そのオブジェクトがlvalue(この例では`int (&) [3]`)であれば、`std::begin`はそのオブジェクトを使用できるので、`decltype`内で使用できる`lvalue`のT型オブジェクトを生成できれば、と考えれば下記のような実装を思いつくだろう。

```
// @@@ example/template/nstd_type_traits.h 150

template <typename, typename = void>
struct exists_begin : std::false_type {
};

template <typename T>
struct exists_begin<T, std::void_t<decltype(std::begin(std::declval<T&>()))>> : std::true_type {
};

template <typename T>
constexpr bool exists_begin_v{exists_begin<T>::value};
```

十分にシンプルなのでこれを採用し、exists_endも同様に実装する。

```
// @@@ example/template/nstd_type_traits.h 163

template <typename, typename = void>
struct exists_end : std::false_type {
};

template <typename T>
struct exists_end<T, std::void_t<decltype(std::end(std::declval<T&>()))>> : std::true_type {
};

template <typename T>
constexpr bool exists_end_v{exists_end<T>::value};
```

単体テストは下記のようになる。

```
// @@@ example/template/nstd_type_traits_ut.cpp 81

static_assert(exists_begin_v<std::string>);
static_assert(!exists_begin_v<int>);
static_assert(exists_begin_v<int const[3]>);
static_assert(exists_begin_v<int[3]>); // 問題が解決

static_assert(exists_end_v<std::string>);
static_assert(!exists_end_v<int>);
static_assert(exists_end_v<int const[3]>);
static_assert(exists_end_v<int[3]>);
```

IsRangeの実装

範囲for文 文の”:“の後にT型オブジェクトが指定できる要件は、

- std::begin(T)、std::end(T)がTのイテレータであるITOR型のオブジェクトを返す
- std::begin(T)が返すITORオブジェクトはTが保持する先頭の要素を指す
- std::end(T)が返すITORオブジェクトはTが保持する最後の要素の次を指す
- ++ITORによりTが保持する全要素にアクセスできる

ようなことである。多くの要件はセマンティクス的なものであり、メタ関数で診断できることは前項で見たようなstd::begin(T)、std::end(T)の可否のみであると考えれば、IsRangeの実装は以下のようになる。

```
// @@@ example/template/nstd_type_traits.h 177

template <typename T>
struct IsRange : std::conditional_t<Inner_::exists_begin_v<T> && Inner_::exists_end_v<T>,
                std::true_type, std::false_type> {};
```

```
template <typename T>
constexpr bool IsRangeV{IsRange<T>::value};
```

なお、上記のコードでは、exists_begin/exsits_endは、IsRangeの実装の詳細であるため、名前空間Inner_で宣言している。

```
// @@@ example/template/nstd_type_traits_ut.cpp 100

static_assert(IsRangeV<std::string>);
static_assert(!IsRangeV<int>);
static_assert(IsRangeV<int const[3]>);
static_assert(IsRangeV<int[3]>);
```

演習-範囲for文のオペランドになれるかどうかの診断

exists_put_to_as_memberの実装

std::ostreamのメンバ関数operator<<の戻り型はstd::ostream&であるため、exists_put_to_as_memberの実装は以下のようになる(“<<”は英語で“put to”と発音する)。

```
// @@@ example/template/exists_func_ut.cpp 219

template <typename, typename = std::ostream&>
struct exists_put_to_as_member : std::false_type {
};

template <typename T>
struct exists_put_to_as_member<T, decltype(std::declval<std::ostream&>().operator<<(std::declval<T>()))> : std::true_type {
};

template <typename T>
constexpr bool exists_put_to_as_member_v{exists_put_to_as_member<T>::value};
```

「[exists_void_func_sfinae_fの実装](#)」と同様のパターンを使用したので解説は不要だろう。

単体テストは以下のようにになる。

```
// @@@ example/template/test_class.h 3

class test_class_exits_put_to {
public:
    test_class_exits_put_to(int i = 0) noexcept : i_{i} {}
    int get() const noexcept { return i_; }

private:
    int i_;
};

inline std::ostream& operator<<(std::ostream& os, test_class_exits_put_to const& p)
{
    return os << p.get();
}

class test_class_not_exits_put_to {};
```



```
// @@@ example/template/exists_func_ut.cpp 236

static_assert(exists_put_to_as_member_v<bool>);
static_assert(!exists_put_to_as_member_v<std::string>);
static_assert(!exists_put_to_as_member_v<std::vector<int>>);
static_assert(exists_put_to_as_member_v<std::vector<int*>>);
static_assert(!exists_put_to_as_member_v<test_class_exits_put_to>);
static_assert(!exists_put_to_as_member_v<test_class_not_exits_put_to>);
static_assert(exists_put_to_as_member_v<test_class_not_exits_put_to[3]>); // 驚き!
```

やや驚きなのは、上記の抜粋である下記コードがコンパイルできることである。

```
// @@@ example/template/exists_func_ut.cpp 245

static_assert(exists_put_to_as_member_v<test_class_not_exits_put_to[3]>); // 驚き!
```

これは、

```
std::ostream& std::ostream::operator<<(void const*)
```

が定義されているため、配列がポインタに変換されてこのメンバ関数にバインドした結果である。

exists_put_to_as_non_memberの実装

exists_put_to_as_non_memberの実装は以下になる。

```
// @@@ example/template/exists_func_ut.cpp 254

template <typename, typename = std::ostream&>
struct exists_put_to_as_non_member : std::false_type {
};

template <typename T>
struct exists_put_to_as_non_member<T, decltype(operator<<(std::declval<std::ostream&>(),
    std::declval<T>()))> : std::true_type {
};
```

```
template <typename T>
constexpr bool exists_put_to_as_non_member_v{exists_put_to_as_non_member<T>::value};
```

「exists_begin/exists_endの実装」と「exists_put_to_as_memberの実装」で使用したパターンを混合しただけなので解説や単体テストは省略する。

ExistsPutToの実装

テンプレートパラメータT、T型オブジェクトtに対して、`std::ostream << t`ができるかどうかを判断するExistsPutToの実装は以下のようになる。

```
// @@@ example/template/exists_func_ut.cpp 283

template <typename T>
struct ExistsPutTo
: std::conditional_t<
    Inner_::exists_put_to_as_member_v<T> || Inner_::exists_put_to_as_non_member_v<T>,
    std::true_type, std::false_type> {
};

template <typename T>
constexpr bool ExistsPutToV{ExistsPutTo<T>::value};
```

「IsRangeの実装」に影響されて、一旦このように実装したが、先に書いた通り、そもそもExistsPutToの役割は`std::ostream << t`ができるかどうかの診断であることを思い出せば、下記のように、もっとシンプルに実装できることに気づくだろう。

```
// @@@ example/template/nstd_type_traits.h 192

namespace Nstd {

template <typename, typename = std::ostream&>
struct ExistsPutTo : std::false_type {
};

template <typename T>
struct ExistsPutTo<T, decltype(std::declval<std::ostream&>() << std::declval<T>())>
: std::true_type {
};

template <typename T>
constexpr bool ExistsPutToV{ExistsPutTo<T>::value};
} // namespace Nstd
```

単体テストは下記のようになる。

```
// @@@ example/template/nstd_type_traits_ut.cpp 111

static_assert(Nstd::ExistsPutToV<bool>);
static_assert(Nstd::ExistsPutToV<std::string>);
static_assert(!Nstd::ExistsPutToV<std::vector<int>>);
static_assert(Nstd::ExistsPutToV<std::vector<int*>>);
static_assert(Nstd::ExistsPutToV<test_class_exists_put_to>);
static_assert(!Nstd::ExistsPutToV<test_class_not_exists_put_to>);
static_assert(Nstd::ExistsPutToV<test_class_not_exists_put_to[3]>);
```

ValueTypeの実装

下記で示す通り、

```
// @@@ example/template/nstd_type_traits_ut.cpp 129

struct T {};

std::ostream& operator<<(std::ostream& os, std::vector<T> const& x)
{
    return os << "T:" << x.size();
}

std::ostream& operator<<(std::ostream&, T const&) = delete;

static_assert(!Nstd::ExistsPutToV<T>);           // std::cout << T{} はできない
static_assert(Nstd::ExistsPutToV<std::vector<T>>); // std::cout << std::vector<T>{} はできる
static_assert(Nstd::ExistsPutToV<T[3]>);          // std::cout << T[3]{} はできる
```

型Xが与えられ、その形式が、

- クラステンプレートCとその型パラメータTにより、C<T>
- 型Tと定数整数Nにより、T[N]

のような場合、ExistsPutToV<X>がtrueであっても、ExistsPutToV<T>の真偽はわからない。従って上記のようなTに対して、ExistsPutToV<T>がtrueかどうかを診断するためには、XからTを導出することが必要になる。ここでは、そのようなメタ関数ValueTypeの実装を考える。このValueTypeは上記のX、Tに対して、

```
std::is_same<ValueType<X>::type, T>::value == true
```

となるような機能を持たなければならることは明らかだろう。その他の機能については実装しながら決定していく。

一見、難しそうなテンプレートを作るコツは、条件を絞って少しづつ作っていくことである。いきなり大量のテンプレートを書いてしまうと、その何十倍ものコンパイルエラーに打ちのめされること必至である。

ということで、まずは、1次元の配列に対してのみ動作するValueTypeの実装を示す(下記で使用するstd::remove_extent_t<T>は、テンプレートパラメータが配列だった場合に、その次元を一つだけ除去するメタ関数である)。

```
// @@@ example/template/value_type_ut.cpp 14

template <typename T, typename = void>
struct ValueType {
    using type = void;
};

template <typename T>
struct ValueType<T, typename std::enable_if_t<std::is_array_v<T>>> {
    using type = typename std::remove_extent_t<T>;
};

template <typename T>
using ValueTypeT = typename ValueType<T>::type;
```

このコードは問題なく動作するが、下記の通り、2次元配列に対するValueType::typeは1次元配列となる。

```
// @@@ example/template/value_type_ut.cpp 32

static_assert(std::is_same_v<int, ValueTypeT<int[1]>>);
static_assert(std::is_same_v<void, ValueTypeT<int>>);
static_assert(std::is_same_v<int[2], ValueTypeT<int[1][2]>>);
```

これを多次元配列に拡張する前に、配列の次元をValueType::Nestで返す機能を追加することにすると、コードは下記のようになるだろう。

```
// @@@ example/template/value_type_ut.cpp 44

template <typename T, typename = void>
struct ValueType {
    using type = void;
    static constexpr size_t Nest{0};
};

template <typename T>
struct ValueType<T, typename std::enable_if_t<std::is_array_v<T>>> {
    using type = typename std::remove_extent_t<T>;

    static constexpr size_t Nest{ValueType<type>::Nest + 1};
};

template <typename T>
using ValueTypeT = typename ValueType<T>::type;
```

動作は下記のようになる。

```
// @@@ example/template/value_type_ut.cpp 69

static_assert(0 == ValueType<int>::Nest);
static_assert(1 == ValueType<int[1]>::Nest);
static_assert(2 == ValueType<int[1][2]>::Nest);
```

ここで、下記のような仕様をもつValueType::type_n<N>を考える。

```
ValueType<int[1][2][3]>::type_n<0>が表す型は、int[1][2][3]
ValueType<int[1][2][3]>::type_n<1>が表す型は、int[2][3]
ValueType<int[1][2][3]>::type_n<2>が表す型は、int[3]
ValueType<int[1][2][3]>::type_n<3>が表す型は、int
```

ValueType::type_n<N>は玉ねぎの皮を一枚ずつむくようなメンバエイリアステンプレートになる。プライマリの実装は以下のようになる。

```
// @@@ example/template/value_type_ut.cpp 82

template <typename T, typename = void>
struct ValueType {
    using type = void;
    static constexpr size_t Nest{0};

    template <size_t N>
    using type_n = typename std::conditional_t<N == 0, T, void>;
};
```

Nが非0の場合、ValueType::type_n<N>はvoidになる仕様にした。

配列に対する特殊化は以下のようになる。

```
// @@@ example/template/value_type_ut.cpp 94

template <typename T, size_t N>
struct ConditionalValueTypeN {
    using type = typename std::conditional_t<
        ValueType<T>::Nest != 0,
        typename ValueType<typename ValueType<T>::type>::template type_n<N - 1>, T>;
};

template <typename T>
struct ConditionalValueTypeN<T, 0> {
    using type = T;
};

template <typename T>
struct ValueType<T, typename std::enable_if_t<std::is_array_v<T>>> {
    using type = typename std::remove_extent_t<T>;

    static constexpr size_t Nest{ValueType<type>::Nest + 1};

    template <size_t N>
    using type_n = typename ConditionalValueTypeN<T, N>::type;
};

template <typename T>
using ValueTypeT = typename ValueType<T>::type;

template <typename T, size_t N>
using ValueTypeT_n = typename ValueType<T>::template type_n<N>;
```

下記コードのコンパイル時の展開を説明することで、上記の解説を行う。

```
// @@@ example/template/value_type_ut.cpp 128

using T = ValueTypeT_n<int[1][2][3], 3>;
```

1. ValueTypeのテンプレートパラメータが配列であるため、配列への特殊化であるValueTypeが選択され、下記の疑似コードのように展開される。

```
ValueType<int[1][2][3], void> {
    using type = int[2][3];
    static constexpr size_t Nest = ...
    using type_n = ConditionalValueTypeN<int[1][2][3], 3>::type;
};
```

2. ConditionalValueTypeNは下記のように展開される。なお、下記のコードの中のtype_n前で使われているキーワードtemplateは、「外部のクラステンプレートのメンバテンプレートにアクセスする」際に必要になる記法である。

```
struct ConditionalValueTypeN<int[1][2][3], 3> {
    using type = typename std::conditional_t<
        true, // ValueType<int[1][2][3]>::Nest == 3であるためtrue
        ValueType<ValueType<int[1][2][3]>::type>::template type_n<3 - 1>,
        int[1][2][3]>::type;
};
```

3. ValueType<int[1][2][3]>::typeは一枚皮をむいたint[2][3]なので、上記はさらに下記のように展開される。

```
struct ConditionalValueTypeN<int[1][2][3], 3> {
    using type = ValueType<int[2][3]>::template type_n<2>;
};
```

4. ConditionalValueTypeN<int[1][2][3], 3>::typeを展開するため、ValueType<int[2][3]>::template type_n<2>の展開が上記1 - 3のように繰り返される。この繰り返しはN == 0になるまで続く。

5. 3回の皮むきによりN == 0となる。この時点で、下記の特殊化が選択されるため再帰は終了し、ConditionalValueTypeN<::type>はintとなる。

```
struct ConditionalValueTypeN<int, 0> {
    using type = int;
};
```

6. 1 - 5により最終的には下記のように展開される。

```
ValueType<int[1][2][3]> {
    using type = int[2][3];
    static constexpr size_t Nest = 3;
    using type_n = int;
};
```

単体テストは下記のようになる。

```
// @@@ example/template/value_type_ut.cpp 136

using T = int[1][2][3];

static_assert(ValueType<T>::Nest == 3);
static_assert(std::is_same_v<int[1][2][3], ValueTypeT_n<T, 0>>);
static_assert(std::is_same_v<int[2][3], ValueTypeT_n<T, 1>>);
static_assert(std::is_same_v<int[3], ValueTypeT_n<T, 2>>);
static_assert(std::is_same_v<int, ValueTypeT_n<T, 3>>);
static_assert(std::is_same_v<void, ValueTypeT_n<T, 4>>);
static_assert(std::is_same_v<void, ValueTypeT_n<T, 5>>);
```

また、ValueType::NestとValueType::type_n<::type>の関係に注目すれば、上記エイリアスTに対して下記が成立する。

```
// @@@ example/template/value_type_ut.cpp 148

static_assert(std::is_same_v<int, ValueTypeT_n<T, ValueType<T>::Nest>>);
```

このテンプレートにコンテナが渡された時の特殊化を与えることができればValueTypeは完成するが、その前に名前の整理をした方が良いため、下記のような変更を行う。

- この例では、typeは配列が直接保持する型を表すが、この名前は慣例的にメタ関数の戻り型を表すことが多いため、現在の仕様では混乱を招く。また名は体を表す方が良いため、typeを改めtype_directとする。
- ValueTypeの結果は、上記のようにNestとtype_nの組み合わせで得られるが、このままでは使い勝手が悪い。慣例に従いこれをtypeとする。
- ConditionalValueTypeNは実装の詳細であるため外部から使われたくない。これまで通り、名前空間Inner_で定義し、名前を小文字と_で生成する。

これによりValueTypeは下記のようになる。

```
// @@@ example/template/value_type_ut.cpp 182

template <typename T, typename = void>
struct ValueType {
    using type_direct = void;

    static constexpr size_t Nest{0};

    template <size_t N>
    using type_n = typename std::conditional_t<N == 0, T, void>;

    using type = type_n<Nest>;
};

namespace Inner_ {

template <typename T, size_t N>
struct conditional_value_type_n {
    using type = typename std::conditional_t<
        ValueType<T>::Nest != 0,
        ValueType<T>::type_n<N - 1>, T>;
};

template <typename T>
struct conditional_value_type_n<T, 0> {
    using type = T;
};

} // namespace Inner_

template <typename T>
```

```

struct ValueType<T, typename std::enable_if_t<std::is_array_v<T>>> {
    using type_direct = typename std::remove_extent_t<T>;

    static constexpr size_t Nest{ValueType<type_direct>::Nest + 1};

    template <size_t N>
    using type_n = typename Inner_::conditional_value_type_n<T, N>::type;

    using type = type_n<Nest>;
};

template <typename T>
using ValueTypeT = typename ValueType<T>::type;

template <typename T, size_t N>
using ValueTypeT_n = typename ValueType<T>::template type_n<N>;

```

準備は整ったので上記のValueTypeに下記のようなコンテナ用特殊化を追加する。

```

// @@@ example/template/value_type_ut.cpp 276

namespace Inner_ {

// Tが配列でなく、且つIsRangeV<T>が真ならばコンテナと診断する
template <typename T>
constexpr bool is_container_v{std::is_range_v<T> && !std::is_array_v<T>};
} // namespace Inner_

template <typename T>
struct ValueType<T, typename std::enable_if_t<Inner_::is_container_v<T>>> {
    using type_direct = typename T::value_type;

    static constexpr size_t Nest{ValueType<type_direct>::Nest + 1};

    template <size_t N>
    using type_n = typename Inner_::conditional_value_type_n<T, N>::type;

    using type = type_n<Nest>;
};

template <typename T>
using ValueTypeT = typename ValueType<T>::type;

template <typename T, size_t N>
using ValueTypeT_n = typename ValueType<T>::template type_n<N>;

```

単体テストは下記のようになる。

```

// @@@ example/template/value_type_ut.cpp 307

using T = std::vector<std::list<int*>[3]>;

static_assert(std::is_same_v<int*, ValueTypeT<T>>);

static_assert(std::is_same_v<T, ValueTypeT_n<T, 0>>);
static_assert(std::is_same_v<std::list<int*>[3], ValueTypeT_n<T, 1>>);
static_assert(std::is_same_v<std::list<int*>, ValueTypeT_n<T, 2>>);
static_assert(std::is_same_v<int*, ValueTypeT_n<T, 3>>);
static_assert(std::is_same_v<void, ValueTypeT_n<T, 4>>);

```

最初のValueTypeには、その単純さとは不釣り合aina、やや複雑なSFINAE用のコードを記述をしたが、ここまで来ればその理由は明らかだろう。今回のように限定的な機能を作ってから、一般化して行く開発スタイルでも、静的ディスパッチにはSFINAE等の汎用的手法を使った方が後の修正が少なく済むことが多い。一方で、完成時にその静的ディスパッチが不要に複雑であると気づいた場合は、リファクタリングを行い、コードを程よいレベルに留めなければならないことは言うまでもない。

ValueTypeの開発はまだ終わらない。静的ディスパッチは最初のカンガ当り修正の必要はないと思うが、「配列用特殊化とコンテナ用特殊化のほとんどがコードクローンになっている」という問題がある。この程度のクローンは問題のないレベルであるとも言えるが、演習のため修正する。また、合わせてTが配列かどうかを示すための定数IsBuiltInArrayも追加すると下記のようなコードになる。

```

// @@@ example/template/nstd_type_traits.h 213

namespace Nstd {

template <typename T, typename = void> // ValueTypeのプライマリ
struct ValueType {
    using type_direct = void;

    static constexpr bool IsBuiltInArray{false};
};

```

```

static constexpr size_t Nest{0};

template <size_t N>
using type_n = typename std::conditional_t<N == 0, T, void>;

using type = type_n<Nest>;
};

namespace Inner_ {

template <typename T, size_t N>
struct conditional_value_type_n {
    using type = typename std::conditional_t<
        ValueTpe<T>::Nest != 0,
        typename ValueTpe<typename ValueTpe<T>::type_direct>::template type_n<N - 1>, T>;
};

template <typename T>
struct conditional_value_type_n<T, 0> {
    using type = T;
};

template <typename T, typename = void>
struct array_or_container : std::false_type {
};

template <typename T>
struct array_or_container<T, typename std::enable_if_t<std::is_array_v<T>>> : std::true_type {
    using type = typename std::remove_extent_t<T>;
};

// Tが配列でなく、且つT型インスタンスに範囲for文が適用できるならばstdコンテナと診断する
template <typename T>
constexpr bool is_container_v{Nstd::IsRange<T>::value && !std::is_array_v<T>} ;

template <typename T>
struct array_or_container<T, typename std::enable_if_t<is_container_v<T>>> : std::true_type {
    using type = typename T::value_type;
};

template <typename T>
constexpr bool array_or_container_v{array_or_container<T>::value};
} // namespace Inner_

template <typename T> // ValueTypeの特殊化
struct ValueTpe<T, typename std::enable_if_t<Inner_::array_or_container_v<T>>> {
    using type_direct = typename Inner_::array_or_container<T>::type;

    static constexpr bool IsBuiltinArray{std::is_array_v<T>};
    static constexpr size_t Nest{ValueTpe<type_direct>::Nest + 1};

    template <size_t N>
    using type_n = typename Inner_::conditional_value_type_n<T, N>::type;

    using type = type_n<Nest>;
};

template <typename T>
using ValueTpeT = typename ValueTpe<T>::type;

template <typename T, size_t N>
using ValueTpeT_n = typename ValueTpe<T>::template type_n<N>;
} // namespace Nstd

```

演習-配列の長さの取り出し 演習-配列の次元の取り出し

Nstdライブラリの開発2

ここでは予定していた通りSafeArray2を開発し、その後Nstdに必要なライブラリの開発を続ける。

SafeArray2の開発

「安全な配列型コンテナ」で断念したSafeArray2の開発を再開する前に、 SafeArray2の要件をまとめると、

- std::arrayを基底クラスとする
- operator[]に範囲チェックを行う

- SafeArrayでのパラメータパックによる初期化機能はそのまま残す
- SafeArrayではできなかった縮小型変換が起こる初期化にも対応する
- 新規要件として、縮小型変換により初期化されたかどうかを示すメンバ関数InitializedWithNarrowConv()を持つ。

となる。この要件を満たすためには、SafeArrayが

```
// @@@ example/template/safe_vector_ut.cpp 154

template <typename... ARGS> // コンストラクタを定義
SafeArray(ARGS... args) : base_type{args...}
{}
```

で行っていた初期化を、SafeArray2では、「縮小型変換が起こるか否かによる場合分けを行い、それぞれの場合に対応するコンストラクタテンプレートによって初期化」するようにすれば良いことがわかる。

パラメータパックによるコンストラクタのシグネチャは上記した一種類しかないので、関数のシグネチャの差異によるオーバーロードは使えない。とすれば、テンプレートパラメータの型の差異によるオーバーロードを使うしか方法がない。縮小型変換が起こるか否かの場合分けはSFINAEで実現させることができる。という風な思考の変遷により以下のコードにたどり着く。

```
// @@@ example/template/safe_vector_ut.cpp 227
namespace Nstd {

template <typename T, size_t N>
struct SafeArray2 : std::array<T, N> {
    using std::array<T, N>::array; // 繙承コンストラクタ
    using base_type = std::array<T, N>;

    // 縮小型変換した場合には、ill-formedになるコンストラクタ
    template <typename... ARGS,
              typename =
              typename std::enable_if_t<
                  AreConvertibleWithoutNarrowConvV<T, ARGS...>>>
    SafeArray2(ARGS... args) : base_type{args...} // 初期化子リストによるarrayの初期化
    {
    }

    // 縮小型変換しない場合には、ill-formedになるコンストラクタ
    template <typename... ARGS,
              typename std::enable_if_t<
                  !AreConvertibleWithoutNarrowConvV<T, ARGS...>>* = nullptr>
    SafeArray2(ARGS... args) :
        base_type{T(args)...}, // 縮小型変換を抑止するため、T(args)が必要
        is_with_narrow_conv_{true}
    {
    }

    using size_type = typename base_type::size_type;

    typename base_type::reference operator[](size_type i) { return this->at(i); }
    typename base_type::const_reference operator[](size_type i) const { return this->at(i); }

    bool InitializedWithNarrowConv() const noexcept { return is_with_narrow_conv_; }

private:
    bool const is_with_narrow_conv_{false};
};
```

下記のようなコードでのコンストラクタ呼び出しには、

```
// @@@ example/template/safe_vector_ut.cpp 290
auto sa_init = Nstd::SafeArray2<int, 3>{1, 2, 3};
```

上記の抜粋である下記のコンストラクタが置換失敗により排除される(SFINAE)。

```
// @@@ example/template/safe_vector_ut.cpp 247
// 縮小型変換しない場合には、ill-formedになるコンストラクタ
template <typename... ARGS,
          typename std::enable_if_t<
              !AreConvertibleWithoutNarrowConvV<T, ARGS...>>* = nullptr>
SafeArray2(ARGS... args) :
    base_type{T(args)...}, // 縮小型変換を抑止するため、T(args)が必要
    is_with_narrow_conv_{true}
{}
```

従って、マッチするコンストラクタは

```
// @@@ example/template/safe_vector_ut.cpp 236

// 縮小型変換した場合には、ill-formedになるコンストラクタ
template <typename... ARGS,
          typename =
          typename std::enable_if_t<
              AreConvertibleWithoutNarrowConvV<T, ARGS...>>>
SafeArray2(ARGS... args) : base_type{args...} // 初期化子リストによるarrayの初期化
{
}
```

のみとなり、無事にコンパイルが成功し、下記の単体テストもパスする。

```
// @@@ example/template/safe_vector_ut.cpp 290

auto sa_init = Nstd::SafeArray2<int, 3>{1, 2, 3};

ASSERT_FALSE(sa_init.InitializedWithNarrowConv()); // 縮小型変換なし
ASSERT_EQ(3, sa_init.size());
ASSERT_EQ(1, sa_init[0]);
ASSERT_EQ(2, sa_init[1]);
ASSERT_EQ(3, sa_init[2]);
ASSERT_THROW(sa_init[3], std::out_of_range);
```

下記の単体テストの場合、SFINAEにより、先述の例とは逆のコンストラクタが選択され、コンパイルも単体テストもパスする。

```
// @@@ example/template/safe_vector_ut.cpp 305
auto const sa_init = Nstd::SafeArray2<int, 3>{10, 20, 30.0}; // 30.0はintに縮小型変換される

ASSERT_TRUE(sa_init.InitializedWithNarrowConv()); // 縮小型変換あり
ASSERT_EQ(3, sa_init.size());
ASSERT_EQ(10, sa_init[0]);
ASSERT_EQ(20, sa_init[1]);
ASSERT_EQ(30, sa_init[2]);
ASSERT_THROW(sa_init[3], std::out_of_range);
```

ここで紹介した2つのコンストラクタテンプレートの最後のパラメータには、かなりの違和感があるだろうが、引数や戻り値に制限の多いコンストラクタテンプレートでSFINAEを起こすためには、このような記述が必要になる。

なお、2つ目のコンストラクタテンプレートの中で使用した下記のコードは、パラメータパックで与えられた全引数をそれぞれにT型オブジェクトに変換するための記法である。

```
// @@@ example/template/safe_vector_ut.cpp 255
base_type{T(args)...}, // 縮小型変換を抑止するため、T(args)が必要
```

これにより、`std::array<T, N>`の`std::initializer_list`による初期化が縮小変換を検出しなくなる。

Nstd::SafeIndexの開発

「安全なvector」、「安全な配列型コンテナ」等の中で、

- Nstd::SafeVector
- Nstd::SafeString
- Nstd::SafeArray

を定義した。これらは少しだけランタイム速度を犠牲にすることで、安全な(未定義動作を起こさない)インデックスアクセスを保障するため、一般的なソフトウェア開発にも有用であると思われるが、コードクローンして作ったため、リファクタリングを行う必要がある。

まずは、Nstd::SafeVectorとNstd::SafeStringの統一を考える。

`std::string`は、実際には`std::basic_string<char>`のエイリアスであることに注目すれば、Nstd::SafeStringの基底クラスは`std::basic_string<char>`であることがわかる。この形式は、`std::vector<T>`と同形であるため、Nstd::SafeVectorとNstd::SafeStringの共通コードはテンプレートテンプレートパラメータ(「is_same_templateの実装」参照)を使用し下記のように書ける。

```
// @@@ example/template/nstd_safe_index.h 8

namespace Nstd {

template <template <class...> class C, typename... Ts>
struct SafeIndex : C<Ts...> {
    using C<Ts...>::C;

    using base_type = C<Ts...>;
```

```

    using size_type = typename base_type::size_type;

    typename base_type::reference operator[](size_type i) { return this->at(i); }
    typename base_type::const_reference operator[](size_type i) const { return this->at(i); }
};

} // namespace Nstd

```

このコードの使用例を兼ねた単体テストは下記のようになる。

```

// @@@ example/template/nstd_safe_index_ut.cpp 8

auto v_i = Nstd::SafeIndex<std::vector, int>{1, 2};

static_assert(std::is_same_v<int&, decltype(v_i[0])>);
static_assert(std::is_base_of_v<std::vector<int>, decltype(v_i)>);
ASSERT_EQ(1, v_i[0]);
ASSERT_EQ(2, v_i[1]);
ASSERT_THROW(v_i[2], std::out_of_range);

auto str = Nstd::SafeIndex<std::basic_string, char>{"123"};

static_assert(std::is_same_v<char&, decltype(str[0])>);
static_assert(std::is_base_of_v<std::string, decltype(str)>);
ASSERT_EQ(3, str.size());
ASSERT_EQ("123", str);
ASSERT_THROW(str[3], std::out_of_range);

```

このままでは使いづらいので下記のようにエイリアスを使い、元のテンプレートと同じ名前を与える。

```

// @@@ example/template/nstd_safe_index.h 24

namespace Nstd {

template <typename T>
using SafeVector = Nstd::SafeIndex<std::vector, T>;

using SafeString = Nstd::SafeIndex<std::basic_string, char>;
} // namespace Nstd

```

このコードの単体テストは下記のようになる。

```

// @@@ example/template/nstd_safe_index_ut.cpp 54

auto v_i = Nstd::SafeVector<int>{1, 2};

static_assert(std::is_same_v<int&, decltype(v_i[0])>);
static_assert(std::is_base_of_v<std::vector<int>, decltype(v_i)>);
ASSERT_EQ(1, v_i[0]);
ASSERT_EQ(2, v_i[1]);
ASSERT_THROW(v_i[2], std::out_of_range);

auto str = Nstd::SafeString{"123"};

static_assert(std::is_same_v<char&, decltype(str[0])>);
static_assert(std::is_base_of_v<std::string, decltype(str)>);
ASSERT_EQ(3, str.size());
ASSERT_EQ("123", str);
ASSERT_THROW(str[3], std::out_of_range);

```

これで、Nstd::SafeVectorとNstd::SafeStringは統一できたので、Nstd::SafeIndexにNstd::SafeArrayの実装が取り込めれば、リファクタリングは終了となるが、残念ながら、下記のコードはコンパイルできない。

```

// @@@ example/template/nstd_safe_index_ut.cpp 44

// 下記のように書きたいが、パラメータパックは型と値を混在できないのでコンパイルエラー
auto a_i = Nstd::SafeIndex<std::array, int, 5>{};

```

理由は、パラメータパックにはそのすべてに型を指定するか、そのすべてに値を指定しなければならず、上記のコードのような型と値の混在が許されていないからである。

値を型に変換する`std::integral_constant`を使用し、この問題を解決できる。`std::array`から派生した下記の`StdArrayLike`は、`std::integral_constant::value`から値を取り出し、基底クラス`std::array`の第2テンプレートパラメータとする。この仕組みにより、`StdArrayLike`は、`Nstd::SafeIndex`のテンプレートテンプレートパラメータとして使用できるようになる。

```

// @@@ example/template/nstd_safe_index.h 34

namespace Nstd {
namespace Inner_ {

```

```

template <typename T, typename U>
struct std_array_like : std::array<T, U::value> {
    using std::array<T, U::value>::array;

    template <typename... ARGS>
    std_array_like(ARGS... args) noexcept(std::is_nothrow_constructible_v<T, ARGS...>)
        : std::array<T, U::value>{args...}
    {
        static_assert(AreConvertibleV<T, ARGS...>, "arguemnt error");
    }
};

} // namespace Inner_
} // namespace Nstd

```

まずは、このコードの使用例を兼ねた単体テストを下記に示す。

```

// @@@ example/template/nstd_safe_index_ut.cpp 134

auto sal = Nstd::Inner_::std_array_like<int, std::integral_constant<size_t, 3>>{1, 2, 3};

static_assert(std::is_nothrow_constructible_v<decltype(sal), int>); // エクセプション無し生成
static_assert(std::is_same_v<int&, decltype(sal[0])>);
static_assert(std::is_base_of_v<std::array<int, 3>, decltype(sal)>);

ASSERT_EQ(1, sal[0]);
ASSERT_EQ(2, sal[1]);
ASSERT_EQ(3, sal[2]);

using T = Nstd::Inner_::std_array_like<std::string, std::integral_constant<size_t, 3>>;
auto sal2 = T{"1", "2", "3"};

static_assert(!std::is_nothrow_constructible_v<std::string, char const*>);
static_assert(!std::is_nothrow_constructible_v<T, char const*>); // エクセプション有り生成
static_assert(std::is_same_v<std::string&, decltype(sal2[0])>);
static_assert(std::is_base_of_v<std::array<std::string, 3>, decltype(sal2)>);

ASSERT_EQ("1", sal2[0]);
ASSERT_EQ("2", sal2[1]);
ASSERT_EQ("3", sal2[2]);

```

これを使えば、下記のような記述が可能となる。

```

// @@@ example/template/nstd_safe_index_ut.cpp 157

using T2 = Nstd::SafeIndex<Nstd::Inner_::std_array_like, std::string,
                           std::integral_constant<size_t, 4>>;
auto sal_s = T2{"1", "2", "3"};

static_assert(!std::is_nothrow_constructible_v<T2, char const*>); // エクセプション有り生成
static_assert(std::is_same_v<std::string&, decltype(sal_s[0])>);
static_assert(std::is_base_of_v<std::array<std::string, 4>, decltype(sal_s)>);

ASSERT_EQ("1", sal_s[0]);
ASSERT_EQ("2", sal_s[1]);
ASSERT_EQ("3", sal_s[2]);
ASSERT_EQ("", sal_s[3]);
ASSERT_THROW(sal_s[4], std::out_of_range);

```

このままでは使いづらいのでNstd::SafeVector、Nstd::Stringと同様にエイリアスを使えば、下記のようになる。

```

// @@@ example/template/nstd_safe_index.h 53

namespace Nstd {

template <typename T, size_t N>
using SafeArray
    = Nstd::SafeIndex<Nstd::Inner_::std_array_like, T, std::integral_constant<size_t, N>>;
} // namespace Nstd

```

このコードの単体テストは下記のようになる。

```

// @@@ example/template/nstd_safe_index_ut.cpp 89

auto sal_s = Nstd::SafeArray<std::string, 4>{"1", "2", "3"};

static_assert(std::is_same_v<std::string&, decltype(sal_s[0])>);
static_assert(std::is_base_of_v<std::array<std::string, 4>, decltype(sal_s)>);

ASSERT_EQ("1", sal_s[0]);
ASSERT_EQ("2", sal_s[1]);
ASSERT_EQ("3", sal_s[2]);
ASSERT_EQ("", sal_s[3]);
ASSERT_THROW(sal_s[4], std::out_of_range);

```

これにより、当初の目的であったコードクローンの除去が完了した。この効果により、下記に示したような拡張もコードクローンせずに簡単に実行できるようになった。

```
// @@@ example/template/nstd_safe_index.h 62

namespace Nstd {

using SafeStringU16 = Nstd::SafeIndex<std::basic_string, char16_t>;
using SafeStringU32 = Nstd::SafeIndex<std::basic_string, char32_t>;
} // namespace Nstd

// @@@ example/template/nstd_safe_index_ut.cpp 112

auto u16str = Nstd::SafeStringU16{u"あいうえお"};

static_assert(std::is_same_v<char16_t&, decltype(u16str[0])>);
static_assert(std::is_base_of_v<std::u16string, decltype(u16str)>);
ASSERT_EQ(5, u16str.size());
ASSERT_EQ(u"あいうえお", u16str);
ASSERT_THROW(u16str[5], std::out_of_range);

auto u32str = Nstd::SafeStringU32{u"かきくけご"};

static_assert(std::is_same_v<char32_t&, decltype(u32str[0])>);
static_assert(std::is_base_of_v<std::u32string, decltype(u32str)>);
ASSERT_EQ(5, u32str.size());
ASSERT_EQ(u"かきくけご", u32str);
ASSERT_THROW(u32str[5], std::out_of_range);
```

Nstd::SafeIndexのoperator<<の開発

ここでは、Nstd::SafeIndexのoperator<<の開発を行う。

他のoperator<<との間で定義が曖昧にならないようにするために、テンプレートテンプレートパラメータを使って以下のようにすることが考えられる。

```
// @@@ example/template/safe_index_put_to_ut.cpp 8

template <template <class...> class C, typename... Ts>
std::ostream& operator<<(std::ostream& os, Nstd::SafeIndex<C, Ts...> const& safe_index)
{
    auto first = true;

    for (auto const& i : safe_index) {
        if (!std::exchange(first, false)) {
            os << ", ";
        }
        os << i;
    }

    return os;
}
```

以下の単体テストで動作確認する。

```
// @@@ example/template/safe_index_put_to_ut.cpp 28

{
    auto v_i = Nstd::SafeVector<int>{1, 2};

    auto oss = std::ostringstream{};
    oss << v_i;
    ASSERT_EQ("1, 2", oss.str());
}

{
    auto sal_s = Nstd::SafeArray<std::string, 4>{"1", "2", "3"};
    auto oss = std::ostringstream{};
    oss << sal_s;
    ASSERT_EQ("1, 2, 3, ", oss.str()); // 4番目には何も入っていない
}
```

ここまでうまく行っているが、以下の単体テストによりバグが発覚する。

```
// @@@ example/template/safe_index_put_to_ut.cpp 43

{
    auto s_str = Nstd::SafeString{"hello"};
    auto oss = std::ostringstream{};
    oss << s_str;
```

```

    // ASSERT_EQ("hello", oss.s_str());      // これがパス出来たらよいが、
    ASSERT_EQ("h, e, l, l, o", oss.str());  // 実際にはこのようになる。
}
{
    auto str = std::string{"hello"}; // 上記と比較のためのstd::stringでのoperator<<

    auto oss = std::ostringstream{};
    oss << str;
    ASSERT_EQ("hello", oss.str());
}

```

この原因は、`Nstd::SafeString`オブジェクトに対して、`std::operator<<`が使用されなかったからである。

「[メタ関数のテクニック](#)」で紹介したSFINAEにより、この問題は下記のように対処できる。

```

// @@@ example/template/safe_index_put_to_ut.cpp 102

template <template <class...> class C, typename... Ts>
auto operator<<(std::ostream& os, Nstd::SafeIndex<C, Ts...> const& safe_index) ->
    typename std::enable_if_t< // safe_indexがSafeString型ならば、SFINAEにより非活性化
        !std::is_same_v<Nstd::SafeIndex<C, Ts...>, Nstd::SafeString>, std::ostream&>
{
    auto first = true;

    for (auto const& i : safe_index) {
        if (!std::exchange(first, false)) {
            os << ", ";
        }
        os << i;
    }

    return os;
}

```

これにより先ほど問題が発生した単体テストも下記のようにパスする。

```

// @@@ example/template/safe_index_put_to_ut.cpp 138

auto str = Nstd::SafeString{"hello"};
auto oss = std::ostringstream{};
oss << str;
ASSERT_EQ("hello", oss.str()); // std::operator<<が使われる
// ASSERT_EQ("h, e, l, l, o", oss.str());

```

コンテナ用`Nstd::operator<<`の開発

「[Nstd::SafeIndexのoperator<<の開発](#)」で定義した`Nstd::operator<<`の構造は、範囲for文に適用できる配列やstdコンテナにも使えるため、ここではその拡張を考える。

すでに述べたように注意すべきは、

- 使い勝手の良い`std::operator<<`(例えば`char[N]`や`std::string`の`operator<<`)はそのまま使う
- ほとんど使い物にならない`std::operator<<`(例えば、`int[N]`のような配列に対する`operator<<(void*)`)の代わりに、ここで拡張する`Nstd::operator<<`を使う

そのため、型Tが新しい`Nstd::operator<<`を使用できる条件は、

- Tの型が、以下の条件を満たす
 - T == U[N]であった場合、Uは`char`ではない
 - `std::string`およびその派生型ではない
- `Nstd::ValueType<T>::type`が`operator<<`を持つ

となるだろう。この条件を診断するためのメタ関数は以下のようになる。

```

// @@@ example/template/nstd_put_to.h 16

namespace Nstd {
namespace Inner_ {

template <typename T>
constexpr bool enable_range_put_to() noexcept
{
    if constexpr (Nstd::ValueType<T>::IsBuiltInArray) { // Tは配列
        if constexpr (std::is_same_v<char,
            typename Nstd::ValueType<T>::type_direct>) { // Tはchar配列

```

```

        return false;
    }
    else {
        return Nstd::ExistsPutToV<typename Nstd::ValueTypeT<T>>;
    }
}
else { // Tは配列ではない
#ifndef __clang__
    if constexpr (Nstd::ExistsPutToV<T>) { // operator<<を持つ(std::string等)
#else
    if (Nstd::ExistsPutToV<T>) { // operator<<を持つ(std::string等)
#endif
        return false;
    }
    else {
        if constexpr (Nstd::IsRangeV<T>) { // 範囲for文に適用できる
            return Nstd::ExistsPutToV<typename Nstd::ValueTypeT<T>>;
        }
        else {
            return false;
        }
    }
}
}

template <typename T>
constexpr bool enable_range_put_to_v{enable_range_put_to<T>()};
} // namespace Inner_
} // namespace Nstd

```

ただし、このようなコードはコンパイラのバグによりコンパイルできないことがある。実際、現在使用中のg++ではこのコードはコンパイルできず、上記コードではそのワークアラウンドを行っている。

このような場合、条件分岐に三項演算子を使うことで回避できることが多いが、ここではg++の問題を明示するためにプリプロセッサ命令を用いた。

このような複雑なメタ関数には単体テストは必須である。

```

// @@@ example/template/test_class.h 3

class test_class_exits_put_to {
public:
    test_class_exits_put_to(int i = 0) noexcept : i_{i} {}
    int get() const noexcept { return i_; }

private:
    int i_;
};

inline std::ostream& operator<<(std::ostream& os, test_class_exits_put_to const& p)
{
    return os << p.get();
}

class test_class_not_exits_put_to {};

// @@@ example/template/nstd_put_to_ut.cpp 31

static_assert(enable_range_put_to_v<int[3]>); // Nstd::operator<<
static_assert(!enable_range_put_to_v<char[3]>); // std::operator<<
static_assert(!enable_range_put_to_v<int>); // std::operator<<
static_assert(enable_range_put_to_v<std::vector<int>>); // Nstd::operator<<
static_assert(enable_range_put_to_v<std::vector<std::vector<int>>>); // Nstd::operator<<
static_assert(!enable_range_put_to_v<std::string>); // std::operator<<
static_assert(enable_range_put_to_v<std::vector<std::string>>); // Nstd::operator<<

static_assert(!enable_range_put_to_v<test_class_not_exits_put_to>); // operator<<無し
static_assert(!enable_range_put_to_v<test_class_exits_put_to>); // ユーザ定義operator<<
static_assert(
    !enable_range_put_to_v<std::vector<test_class_not_exits_put_to>>); // operator<<無し
static_assert(enable_range_put_to_v<std::vector<test_class_exits_put_to>>); // Nstd::operator<<
static_assert(
    !enable_range_put_to_v<std::list<test_class_not_exits_put_to>>); // operator<<無し
static_assert(enable_range_put_to_v<std::list<test_class_exits_put_to>>); // Nstd::operator<<

```

以上によりstd::enable_ifの第1引数に渡す値(enable_range_put_to_vはconstexpr)が用意できたので、Nstd::operator<<は下記のように定義できる。

```

// @@@ example/template/nstd_put_to.h 58

namespace Nstd {
namespace Inner_ {

template <size_t N>
constexpr std::string_view range_put_to_sep() noexcept
{
    static_assert(N != 0);
    switch (N) {
    case 1:
        return ", ";
    case 2:
        return " | ";
    case 3:
    default:
        return " # ";
    }
}
} // namespace Inner_

template <typename T>
auto operator<<(std::ostream& os, T const& t) ->
#if defined(__clang__)
    typename std::enable_if_t<Inner_::enable_range_put_to_v<T>, std::ostream&>
#else // g++でのワークアラウンド
    typename std::enable_if_t<Inner_::enable_range_put_to<T>(), std::ostream&>
#endif
{
    auto first = true;
    constexpr auto s = Inner_::range_put_to_sep<ValueType<T>::Nest>();

    for (auto const& i : t) {
        if (!std::exchange(first, false)) {
            os << s;
        }
        os << i;
    }

    return os;
}
} // namespace Nstd

```

値表示用のセパレータに”,”のみを用いるとコンテナや配列が多次元(ValueType::Nest > 2)の場合、各次元でのデータの判別が難しくなるため、ValueType::Nestの値によってセパレータの種類を変える range_put_to_sep<>()を用意した。下記単体テストでわかる通り、この効果により値の構造が見やすくなっている。

まずは、配列の単体テストを示す。

```

// @@@ example/template/nstd_put_to_ut.cpp 61

using namespace Nstd;
{
    char str[] = "abcdef";
    auto oss = std::ostringstream{};

    oss << str; // std::operator<<
    ASSERT_EQ(str, oss.str());
}

{
    char str[2][4] = {"abc", "def"};
    auto oss = std::ostringstream{};

    oss << str; // Nstd::operator<<
    ASSERT_EQ("abc | def", oss.str());
}

{
    test_class_exits_put_to p1[3]{1, 2, 3};
    auto oss = std::ostringstream{};

    oss << p1; // Nstd::operator<<
    ASSERT_EQ("1, 2, 3", oss.str());
}

{
    char const* str[] = {"abc", "def", "ghi"};
    auto oss = std::ostringstream{};

    oss << str; // Nstd::operator<<
    ASSERT_EQ("abc, def, ghi", oss.str());
}

```

```

}
{
    int v[2][3][2]{{{0, 1}, {2, 3}, {4, 5}}, {{6, 7}, {8, 9}, {10, 11}}};
    auto oss = std::ostringstream{};

    oss << v;
    ASSERT_EQ("0, 1 | 2, 3 | 4, 5 # 6, 7 | 8, 9 | 10, 11", oss.str());
}

```

次に、コンテナの単体テストを示す。

```

// @@@ example/template/nstd_put_to_ut.cpp 118

using namespace Nstd;
{
    auto v    = std::vector<int>{1, 2, 3};
    auto oss = std::ostringstream{};

    oss << v;
    ASSERT_EQ("1, 2, 3", oss.str());
}
{
    auto p1 = std::list<test_class_exits_put_to>{1, 2, 3, 4};
    auto oss = std::ostringstream{};

    oss << p1;
    ASSERT_EQ("1, 2, 3, 4", oss.str());
}
{
    std::vector<int> v[2]{{1, 2}, {3, 4, 5}}; // std::vectorの配列
    auto          oss = std::ostringstream{};

    oss << v;
    ASSERT_EQ("1, 2 | 3, 4, 5", oss.str());
}

```

最後に、Nstd::SafeIndexの単体テストを示す。

```

// @@@ example/template/nstd_put_to_ut.cpp 168

{
    auto sal_s = Nstd::SafeArray<std::string, 4>{"1", "2", "3"};
    auto oss   = std::ostringstream{};

    oss << sal_s;
    ASSERT_EQ("1 | 2 | 3 | ", oss.str());
}
{
    auto sv
        = Nstd::SafeVector<Nstd::SafeArray<Nstd::SafeString, 2>>{{"ab", "cd"}, {"ef", "gh"}};
    auto oss = std::ostringstream{};

    oss << sv;
    ASSERT_EQ("ab | cd # ef | gh", oss.str());
}

```

ログ取得ライブラリの開発2

ログ取得ライブラリでの問題は「Logging名前空間が依存してよい名前空間」に

```

// @@@ example/template/app_ints.h 6

namespace App {
    using Ints_t = std::vector<int>;
}

```

のようなコンテナに共通したoperator<<を定義することで解決する。それは「[コンテナ用Nstd::operator<<の開発](#)」で示したコードそのものであるため、これを使い、問題を解決したログ取得ライブラリを以下に示す。

```

// @@@ example/template/logger.h 7

namespace Logging {
    class Logger {
    public:
        static Logger&      Inst();
        static Logger const& InstConst() { return Inst(); }
    }
}

```

```

    std::string Get() const; // ログデータの取得
    void Clear(); // ログデータの消去

    template <typename... ARGS> // ログの登録
    void Set(char const* filename, uint32_t line_no, ARGS const&... args)
    {
        oss_.width(32);
        oss_ << filename << ":";

        oss_.width(3);
        oss_ << line_no;

        set_inner(args...);
    }

    Logger(Logger const&) = delete;
    Logger& operator=(Logger const&) = delete;

private:
    void set_inner() { oss_ << std::endl; }

    template <typename HEAD, typename... TAIL>
    void set_inner(HEAD const& head, TAIL const&... tails)
    {
        using Nstd::operator<<; // Nstd::operator<<もname lookupの対象にする

        oss_ << ":" << head;
        set_inner(tails...);
    }

    Logger() {}
    std::ostringstream oss_{};
};

} // namespace Logging

#define LOGGER_P(...) Logging::Logger::Inst().Set(__FILE__, __LINE__)
#define LOGGER(...) Logging::Logger::Inst(__FILE__, __LINE__, __VA_ARGS__)

```

問題のあったコードとの差分は、メンバ関数テンプレートset_innerの

```

// @@@ example/template/logger.h 40

using Nstd::operator<<; // Nstd::operator<<もname lookupの対象にする

```

のみである。実際に解決できたことを以下の単体テストで示す。

```

// @@@ example/template/logger_0_ints_ut.h 8

auto ints = App::Ints_t{1, 2, 3};

LOGGER("Ints", ints);
auto line_num = __LINE__;

auto s = Logging::Logger::InstConst().Get();

auto exp = std::string{log_str_exp(__FILE__, line_num - 1, ":Ints:1, 2, 3\n")};
ASSERT_EQ(exp, s);

```

また、

```

// @@@ example/template/app_ints.h 12

namespace App {

class X {
public:
    X(std::string str, int num) : str_{std::move(str)}, num_{num} {}
    std::string ToString() const { return str_ + "/" + std::to_string(num_); }
    ...
};

} // namespace App

```

のように定義されたクラスも、

```

// @@@ example/template/app_ints.h 28

namespace App {

inline std::ostream& operator<<(std::ostream& os, X const& x) { return os << x.ToString(); }

} // namespace App

```

のような型専用のoperator<<があれば、そのオブジェクトのみではなく、コンテナや配列に対しても下記のようにログ取得が可能となる。

```
// @@@ example/template/logger_ut.cpp 37

using namespace Nstd;

auto      x  = App::X{"name", 3};
auto      lx = std::list<App::X>{{"lx3", 3}, {"lx4", 1}};
App::X const x3[3]{{"x0", 0}, {"x1", 1}, {"x2", 2}};

LOGGER(1, x, x3, lx);
auto line_num = __LINE__;

auto s = Logging::Logger::InstConst().Get();

auto const exp
= log_str_exp(__FILE__, line_num - 1, ":1:name/3:x0/0, x1/1, x2/2:lx3/3, lx4/1\n");
ASSERT_EQ(exp, s);
```

「Nstdライブラリの開発」で示した依存関係も維持されており、これでログ取得ライブラリは完成したと言って良いだろう。

その他のテンプレートテクニック

ここでは、これまでの議論の対象にならなかったテンプレートのテクニックや注意点について記述する。

ユニバーサルリファレンスとstd::forward

2個の文字列からstd::vector<std::string>を生成する下記のような関数について考える。

```
// @@@ example/template/universal_ref_ut.cpp 9

std::vector<std::string> gen_vector(std::string const& s0, std::string const& s1)
{
    auto ret = std::vector<std::string>{};

    ret.push_back(s0);
    ret.push_back(s1);

    return ret;
}
```

これは下記のように動作する。

```
// @@@ example/template/universal_ref_ut.cpp 25

auto a = std::string("a");
auto b = std::string("b");

auto v = gen_vector(a, std::move(b));

ASSERT_EQ((std::vector<std::string>{"a", "b"}), v);
ASSERT_EQ("a", a);
ASSERT_EQ("b", b); // bはmoveされない
```

このコードは正しく動作するものの、move代入できず、パフォーマンス問題を引き起こす可能性があるため、ユニバーサルリファレンスを使って下記のように書き直した。

```
// @@@ example/template/universal_ref_ut.cpp 41

template <typename STR0, typename STR1>
std::vector<std::string> gen_vector(STR0&& s0, STR1&& s1)
{
    auto ret = std::vector<std::string>{};

    ret.push_back(s0);
    ret.push_back(s1);

    return ret;
}
```

残念ながら、このコードは意図したようには動作せず、下記に示した通り相変わらずmove代入ができない。

```
// @@@ example/template/universal_ref_ut.cpp 58

auto a = std::string("a");
auto b = std::string("b");
```

```

auto v = gen_vector(a, std::move(b));
ASSERT_EQ(std::vector<std::string>{"a", "b"}, v);
ASSERT_EQ("a", a);
ASSERT_EQ("b", b); // bはmoveされない

```

この原因是、「関数が受け取ったrvalueリファレンスは、その関数から別の関数に受け渡される時にlvalueリファレンスとして扱われる」からである。

この現象について下記の関数テンプレートを用いて解説を行う。

```

// @@@ example/template/universal_ref_ut.cpp 71

enum class ExpressionType { Lvalue, Rvalue };

template <typename T>
constexpr ExpressionType universal_ref2(T&& t)
{
    return std::is_lvalue_reference_v<decltype(t)> ? ExpressionType::Lvalue
                                                   : ExpressionType::Rvalue;
}

// std::pair<>::first : universal_refの中のtのExpressionType
// std::pair<>::second : universal_ref2の中でtのExpressionType
template <typename T>
constexpr std::pair<ExpressionType, ExpressionType> universal_ref(T&& t)
{
    return std::make_pair(
        std::is_lvalue_reference_v<decltype(t)> ? ExpressionType::Lvalue : ExpressionType::Rvalue,
        universal_ref2(t));
}

```

下記に示した通り、universal_refとuniversal_ref2のパラメータが同じ型であるとは限らない。

```

// @@@ example/template/universal_ref_ut.cpp 95

auto i = 0;

constexpr auto p = universal_ref(i);

static_assert(universal_ref2(i) == ExpressionType::Lvalue);
static_assert(p.first == ExpressionType::Lvalue);
static_assert(p.second == ExpressionType::Lvalue);

constexpr auto pm = universal_ref(std::move(i));

static_assert(universal_ref2(std::move(i)) == ExpressionType::Rvalue);
static_assert(pm.first == ExpressionType::Rvalue);
static_assert(pm.second == ExpressionType::Lvalue);

constexpr auto pm2 = universal_ref(int{});

static_assert(universal_ref2(int{}) == ExpressionType::Rvalue);
static_assert(pm2.first == ExpressionType::Rvalue);
static_assert(pm2.second == ExpressionType::Lvalue);

```

この問題はstd::forwardにより対処できる。これによって改良されたコードを下記に示す。

```

// @@@ example/template/universal_ref_ut.cpp 124

template <typename STR0, typename STR1>
std::vector<std::string> gen_vector(STR0&& s0, STR1&& s1)
{
    auto ret = std::vector<std::string>{};

    ret.push_back(std::forward<STR0>(s0));
    ret.push_back(std::forward<STR1>(s1));

    return ret;
}

```

下記単体テストが示す通り、rvalueリファレンスはmove代入され、lvalueリファレンスはcopy代入されている。

```

// @@@ example/template/universal_ref_ut.cpp 142

auto a = std::string("a");
auto b = std::string("b");

```

```

auto v = gen_vector(a, std::move(b));
ASSERT_EQ((std::vector<std::string>{"a", "b"}), v);
ASSERT_EQ("a", a);
ASSERT_EQ("", b); // bはmoveされた

```

しかし残念ながら、このコードにも改良すべき点がある。

```

// @@@ example/template/universal_ref_ut.cpp 155

auto a = std::string{"a"};
auto v = gen_vector(a, "b");
ASSERT_EQ((std::vector<std::string>{"a", "b"}), v);
ASSERT_EQ("a", a);

```

上記の”b”のような文字列リテラルを引数にした場合、それをstd::vector<std::string>::push_backに渡した時に、“b”はテンポラリオブジェクトstd::string(“b”)に変換されてしまう。std::vector<std::string>へのオブジェクトの挿入は、文字列リテラルから行うことが出来るため、このテンポラリオブジェクトの生成は明らかに不要な処理である。

下記は、この対策を施すとともに任意の数の引数を受け取れるように改良したコードである。

```

// @@@ example/template/universal_ref_ut.cpp 170

void emplace_back(std::vector<std::string>&) noexcept {}

template <typename HEAD, typename... TAIL>
void emplace_back(std::vector<std::string>& strs, HEAD&& head, TAIL&&... tails)
{
    strs.emplace_back(std::forward<HEAD>(head));

    if constexpr (sizeof...(tails) != 0) {
        emplace_back(strs, std::forward<TAIL>(tails)...);
    }
}

template <typename... STR>
std::vector<std::string> gen_vector(STR&&... ss)
{
    auto ret = std::vector<std::string>{};

    emplace_back(ret, std::forward<STR>(ss)...);

    return ret;
}

```

上記の

```
sizeof...(tails)
```

はパラメータパックの個数を受け取るための記法である。従ってこのコードではすべてのパラメータパック変数を消費するまでリカーシブコールを続けることになる（が、このリカーシブコールはコンパイル時に行われるため、実行時の速度低下は起こさない）。

上記の

```
std::forward<TAIL>(tails)...
```

は、それぞれのパラメータパック変数をstd::forwardに渡した戻り値を、再びパラメータパックにするための記法である。

このコードは下記の単体テストが示すように正しく動作する（が、残念ながらテンポラリオブジェクトが生成されていないことを単体テストで証明することはできない）。

```

// @@@ example/template/universal_ref_ut.cpp 198

auto a = std::string{"a"};
auto b = std::string{"b"};

auto v = gen_vector(a, std::move(b), "c");

ASSERT_EQ((std::vector<std::string>{"a", "b", "c"}), v);
ASSERT_EQ("a", a);
ASSERT_EQ("", b); // bはmoveされた

```

ユニバーサルリファレンスはconstにすることができないが（T const&&はconstなlvalueリファレンスである）、ユニバーサルリファレンスがlvalueリファレンスであった場合は、constなlvalueリファレンスとして扱うべきである。

従って、下記のようなコードは書くべきではない。

```
// @@@ example/template/universal_ref_ut.cpp 215

template <typename STR0, typename STR1>
std::vector<std::string> gen_vector(STR0&& s0, STR1&& s1)
{
    auto ret = std::vector<std::string>{};

    ret.push_back(std::move(s0));
    ret.push_back(std::move(s1));

    return ret;
}
```

もしそのようにしてしまえば、下記単体テストが示すように非constな実引数はmoveされてしまうことになる。

```
// @@@ example/template/universal_ref_ut.cpp 232

auto a = std::string{"a"};
auto const b = std::string{"b"};

auto v = gen_vector(a, std::move(b));

ASSERT_EQ((std::vector<std::string>{"a", "b"}), v);
ASSERT_EQ("", a); // aはmoveされてしまう
ASSERT_EQ("b", b); // bはconstなのでmoveされない
```

任意の型Tのrvalueのみを引数に取る関数テンプレートを下記のように記述した場合、すでに述べたように引数はユニバーサルリファレンスとなってしまうため、lvalueにもバインドしてしまう。

```
// @@@ example/template/universal_ref_ut.cpp 248

template <typename T>
void f(T&& t) noexcept
{
    ...
}
```

このような場合、下記の記述が必要になる。

```
// @@@ example/template/universal_ref_ut.cpp 267

template <typename T>
void f(T&) = delete;
```

この効果により、下記に示した通りlvalueにはバインドできなくなり、当初の目的通り、rvalueのみを引数に取る関数テンプレートが定義できたことになる。

```
// @@@ example/template/universal_ref_ut.cpp 275

auto s = std::string{};

// f(s);           // f(std::string&)はdeleteされたため、コンパイルエラー
f(std::string{}); // f(std::string&&)にはバインドできる
```

なお、ユニバーサルリファレンスは、[リファレンスcollapsing](#)の一機能としても理解できる。

ジェネリックラムダ

下記のようなクラスとoperator<<があった場合を考える。

```
// @@@ example/template/generic_lambda_ut.cpp 13

struct XYZ {
    XYZ(int ax, int ay, int az) noexcept : x{ax}, y{ay}, z{az} {}
    int x;
    int y;
    int z;
};

std::ostream& operator<<(std::ostream& os, XYZ const& xyz)
{
    return os << std::to_string(xyz.x) + "/" + std::to_string(xyz.y) + "/" + std::to_string(xyz.z);
}
```

「[Nstd::SafeIndexの開発](#)」や「[コンテナ用Nstd::operator<<の開発](#)」の成果物との組み合わせの単体テストは下記のように書けるだろう。

```
// @@@ example/template/generic_lambda_ut.cpp 31

auto v = Nstd::SafeVector<XYZ>{{9, 8, 7}, {6, 5, 4}, {3, 2, 1}, {0, 1, 2}};
auto oss = std::ostringstream{};

oss << v;
ASSERT_EQ("9/8/7, 6/5/4, 3/2/1, 0/1/2", oss.str());
```

std::sortによるソートができるかどうかのテストは、C++11までは、

```
// @@@ example/template/generic_lambda_ut.cpp 41

auto v = Nstd::SafeVector<XYZ>{{9, 8, 7}, {6, 5, 4}, {3, 2, 1}, {0, 1, 2}};

// C++11 lambda          型の明示が必要
//                      ↓      ↓
std::sort(v.begin(), v.end(), [] (XYZ const& lhs, XYZ const& rhs) noexcept {
    return std::tie(lhs.x, lhs.y, lhs.z) < std::tie(rhs.x, rhs.y, rhs.z);
});
auto oss = std::ostringstream{};

oss << v;
ASSERT_EQ("0/1/2, 3/2/1, 6/5/4, 9/8/7", oss.str());
```

のように書くのが一般的だろう。ラムダ式の引数の型を指定しなければならないのは、範囲for文でautoが使用出来ること等と比べると見劣りがするが、C++14からは下記のコードで示した通り引数にautoが使えるようになった。

```
// @@@ example/template/generic_lambda_ut.cpp 57

auto v = Nstd::SafeVector<XYZ>{{9, 8, 7}, {6, 5, 4}, {3, 2, 1}, {0, 1, 2}};

// C++14 generic lambda      型の明示が不要
//                      ↓      ↓
std::sort(v.begin(), v.end(), [] (auto const& lhs, auto const& rhs) noexcept {
    return std::tie(lhs.x, lhs.y, lhs.z) < std::tie(rhs.x, rhs.y, rhs.z);
});
auto oss = std::ostringstream{};

oss << v;
ASSERT_EQ("0/1/2, 3/2/1, 6/5/4, 9/8/7", oss.str());
```

この記法はジェネリックラムダと呼ばれる。この機能により関数の中で関数テンプレートと同等のものが定義できるようになった。

ジェネリックラムダの内部構造

ジェネリックラムダは下記のように使用することができる。

```
// @@@ example/template/generic_lambda_ut.cpp 73

template <typename PUTTO>
void f(PUTTO&& p)
{
    p(1);        // ラムダの引数elemの型はint
    p(2.71);     // ラムダの引数elemの型はdouble
    p("hehe");   // ラムダの引数elemの型はchar [5]
}

TEST(Template, generic_lambda)
{
    auto oss = std::ostringstream{};

    f([&oss](auto const& elem) { oss << elem << std::endl; });

    ASSERT_EQ("1\n2.71\nhehe\n", oss.str());
}
```

この例で使用しているクロージャは一見、型をダイナミックに扱っているように見えるが、下記のような「テンプレートoperator()を持つ関数型」オブジェクトとして展開されていると考えれば、理解できる。

```
// @@@ example/template/generic_lambda_ut.cpp 92

class Closure {
public:
    Closure(std::ostream& os) : os_{os} {}

    template <typename T>
    void operator()(T& t)
    {
```

```

        os_ << t << std::endl;
    }

private:
    std::ostream& os_;
};

TEST(Template, generic_lambda_like)
{
    auto oss = std::ostringstream{};

    auto closure = Closure{oss};
    f(closure);

    ASSERT_EQ("1\n2.71\nhehe\n", oss.str());
}

```

std::variantとジェネリックラムダ

unionは、オブジェクトを全く無関係な複数の型に切り替えることができるため、これが必要な場面では有用な機能であるが、未定義動作を誘発してしまう問題がある。この対策としてC++17で導入されたものが、std::variantである。

まずは、std::variantの使用例を下記する。

```

// @@@ example/template/variant_ut.cpp 13

auto v = std::variant<int, std::string, double>{}; // 3つの型を切り替える

// std::get<N>()の戻り値型は、下記の通り。
// N == 0, 1, 2 は、それぞれint, std::string, doubleに対応
static_assert(std::is_same_v<decltype(std::get<0>(v)), int>);
static_assert(std::is_same_v<decltype(std::get<1>(v)), std::string>);
static_assert(std::is_same_v<decltype(std::get<2>(v)), double>);

v = int{3}; // int型の3を代入

ASSERT_EQ(v.index(), 0); // intを保持
ASSERT_EQ(std::get<0>(v), 3); // intなので問題なくアクセス
ASSERT_THROW(std::get<1>(v), std::bad_variant_access); // std::stringではないのでエクセプション
ASSERT_THROW(std::get<2>(v), std::bad_variant_access); // doubleではないのでエクセプション

v = std::string{"str"}; // std::stringオブジェクトを代入

ASSERT_EQ(v.index(), 1); // std::stringを保持
ASSERT_THROW(std::get<0>(v), std::bad_variant_access); // intではないのでエクセプション
ASSERT_EQ(std::get<1>(v), std::string{"str"}); // std::stringなので問題なくアクセス
ASSERT_THROW(std::get<2>(v), std::bad_variant_access); // doubleではないのでエクセプション

```

上記からわかる通り、std::variantオブジェクトは、直前に代入されたオブジェクトの型以外で、値を読み出した場合、問題なく読み出せるが、それ以外ではエクセプションを発生させる。

このstd::variantオブジェクトの保持する型とその値を文字列として取り出すラムダ式は、下記のように書ける。

```

// @@@ example/template/variant_ut.cpp 37

auto oss = std::ostringstream{};

// type_valueはvが保持する型をその値を文字列で返す
auto type_value = [&oss](auto const& v) { // ジェネリックラムダでなくとも実装可能
    if (v.index() == 0) {
        auto a = std::get<0>(v);
        using T = std::remove_const_t<std::remove_reference_t<decltype(a)>>;
        oss << Nstd::Type2Str<T>() << " : " << a; // Nstd::Type2Str<T>はTの型を文字列にする
    }
    else if (v.index() == 1) {
        auto a = std::get<1>(v);
        using T = std::remove_const_t<std::remove_reference_t<decltype(a)>>;
        oss << Nstd::Type2Str<T>() << " : " << a; // Nstd::Type2Str<T>はTの型を文字列にする
    }
    else if (v.index() == 2) {
        auto a = std::get<2>(v);
        using T = std::remove_const_t<std::remove_reference_t<decltype(a)>>;
        oss << Nstd::Type2Str<T>() << " : " << a; // Nstd::Type2Str<T>はTの型を文字列にする
    }
    else {
        assert(false); // ここには来ないはず
    }
};

```

```

v = 3; // int型の3を代入
type_value(v);
ASSERT_EQ("int : 3", oss.str());
oss = std::ostringstream{}; // ossのリセット

v = std::string{"str"}; // std::stringオブジェクトを代入
type_value(v);
ASSERT_EQ("std::string : str", oss.str());
oss = std::ostringstream{}; // ossのリセット

v = 1.1; // double型の1.1を代入
type_value(v);
ASSERT_EQ("double : 1.1", oss.str());

```

このラムダは、3つの型をテンプレートパラメータとするstd::variantオブジェクト以外には適用できないため、型の個数に制限のない方法を考える。

この実装は、

- 保持する型が何番目かを見つけるための関数テンプレート
- 関数テンプレートの引数となるジェネリックラムダ

の2つによって下記のように行うことができる。

```

// @@@ example/template/variant_ut.cpp 79

template <typename VARIANT, typename F, size_t INDEX = 0>
void org_visit(const F& f, const VARIANT& v)
{
    constexpr auto n = std::variant_size_v<VARIANT>;

    if constexpr (INDEX < n) {
        if (v.index() == INDEX) { // 保持する型が見つかった
            f(std::get<INDEX>(v));
            return;
        }
        else { // 保持する型が見つかるまでリカーシブ
            org_visit<VARIANT, F, INDEX + 1>(f, v);
        }
    }
    else {
        assert(false); // ここには来ないはず
    }
}

// @@@ example/template/variant_ut.cpp 103

auto oss = std::ostringstream{};

// 文字列を返すためのジェネリックラムダ
auto type_value = [&oss](auto const& a) {
    using T = std::remove_const_t<std::remove_reference_t<decltype(a)>>;
    oss << Nstd::Type2Str<T>() << " : " << a;
};

```

単体テストは、以下のようになる。

```

// @@@ example/template/variant_ut.cpp 113
{
    auto v = std::variant<int, std::string, double>{}; // 3つの型を切り替える

    v = 3;
    org_visit(type_value, v);
    ASSERT_EQ("int : 3", oss.str());
    oss = std::ostringstream{}; // ossのリセット

    ...
}

auto v = std::variant<char, int, std::string, double>{}; // 4つの型を切り替える

v = 'c';
org_visit(type_value, v);
ASSERT_EQ("int : 3", oss.str());
oss = std::ostringstream{}; // ossのリセット

v = 'c';
org_visit(type_value, v);

```

```

    ASSERT_EQ("char : c", oss.str());
    oss = std::ostringstream{}; // ossのリセット
}

...
}

```

下記のように継承関係のない複数のクラスが同じシグネチャのメンバ関数を持つ場合、

```

// @@@ example/template/variant_ut.cpp 177

class A {
public:
    char f() const noexcept { return 'A'; }
};

class B {
public:
    char f() const noexcept { return 'B'; }
};

class C {
public:
    char f() const noexcept { return 'C'; }
};

```

`std::variant`、上に示した関数テンプレート、ジェネリックラムダを使い、下記に示したような疑似的なポリモーフィズムを実現できる。

```

// @@@ example/template/variant_ut.cpp 197

char ret{};
auto call_f = [&ret](auto const& a) { ret = a.f(); };

auto v = std::variant<A, B, C>{};

org_visit(call_f, v); // 疑似的なポリモーフィズム
ASSERT_EQ('A', ret);

v = B{};
org_visit(call_f, v); // 疑似的なポリモーフィズム
ASSERT_EQ('B', ret);

v = C{};
org_visit(call_f, v); // 疑似的なポリモーフィズム
ASSERT_EQ('C', ret);

```

ここで示した関数テンプレートは、デザインパターンVisitorの例であり、ほぼこれと同様のものが`std::visit`として定義されている。

```

// @@@ example/template/variant_ut.cpp 215

v = A{};
std::visit(call_f, v); // 疑似的なポリモーフィズム
ASSERT_EQ('A', ret);

v = B{};
std::visit(call_f, v); // 疑似的なポリモーフィズム
ASSERT_EQ('B', ret);

v = C{};
std::visit(call_f, v); // 疑似的なポリモーフィズム
ASSERT_EQ('C', ret);

```

クラステンプレートと継承の再帰構造

クラステンプレートと継承の再帰構造はCRTPと呼ばれる。このコードパターンについては、「[CRTP\(curiously recurring template pattern\)](#)」で説明している。

constexpr if文

C++17で導入された`constexpr if`文とは、文を条件付きコンパイルすることができるようにするための制御構文である。

まずは、この構文を使用しない例を示す。

```

// @@@ example/template/constexpr_if_ut.cpp 9

// 配列のサイズ
template <typename T>
auto Length(T const&) -> std::enable_if_t<std::is_array_v<T>, size_t>

```

```

    {
        return std::extent_v<T>;
    }

    // コンテナのサイズ
    template <typename T>
    auto Length(T const& t) -> decltype(t.size())
    {
        return t.size();
    }

    // その他のサイズ
    size_t Length(...) { return 0; }

```

```

// @@@ example/template/constexpr_if_ut.cpp 31

uint32_t a[5];
auto v = std::vector{0, 1, 2};
struct SizeTest {
} t;

ASSERT_EQ(5, Length(a));
ASSERT_EQ(3, Length(v));
ASSERT_EQ(0, Length(t));

// C++17で、Lengthと同様の機能の関数テンプレートがSTLに追加された
ASSERT_EQ(std::size(a), Length(a));
ASSERT_EQ(std::size(v), Length(v));

```

このような場合、SFINAEによるオーバーロードが必要であったが、この文を使用することで、下記のようにオーバーロードを使用せずに記述できるため、条件分岐の可読性の向上が見込める。

```

// @@@ example/template/constexpr_if_ut.cpp 52

struct helper {
    template <typename T>
    auto operator()(T const& t) -> decltype(t.size());
};

template <typename T>
size_t Length(T const& t)
{
    if constexpr (std::is_array_v<T>) { // Tが配列の場合
        // Tが配列でない場合、他の条件のブロックはコンパイル対象外
        return std::extent_v<T>;
    }
    else if constexpr (std::is_invocable_v<helper, T>) { // T::Lengthが呼び出せる場合
        // T::Lengthが呼び出せない場合、他の条件のブロックはコンパイル対象外
        return t.size();
    }
    else { // それ以外
        // Tが配列でなく且つ、T::Lengthが呼び出しない場合、他の条件のブロックはコンパイル対象外
        return 0;
    }
}

```

この構文はパラメータパックの展開においても有用な場合がある。

```

// @@@ example/template/constexpr_if_ut.cpp 93

// テンプレートパラメータで与えられた型のsizeofの値が最も大きな値を返す。
template <typename HEAD>
constexpr size_t MaxSizeof()
{
    return sizeof(HEAD);
}

template <typename HEAD, typename T, typename... TAILS>
constexpr size_t MaxSizeof()
{
    return std::max(sizeof(HEAD), MaxSizeof<T, TAILS...>());
}

// @@@ example/template/constexpr_if_ut.cpp 111

static_assert(4 == (MaxSizeof<int8_t, int16_t, int32_t>()));
static_assert(4 == (MaxSizeof<int32_t, int16_t, int8_t>()));
static_assert(sizeof(std::string) == MaxSizeof<int32_t, int16_t, int8_t, std::string>());

```

C++14までの構文を使用する場合、上記のようなオーバーロードとリカーシブコールの組み合わせが必要であったが、`constexpr if`を使用することで、やや単純に記述できる。

```
// @@@ example/template/constexpr_if_ut.cpp 123

// テンプレートパラメータで与えられた型のsizeofの値が最も大きな値を返す。
template <typename HEAD, typename... TAILS>
constexpr size_t MaxSizeof()
{
    if constexpr (sizeof...(TAILS) == 0) { // TAILSが存在しない場合
        return sizeof(HEAD);
    }
    else {
        return std::max(sizeof(HEAD), MaxSizeof<TAILS...>());
    }
}
```

意図しないname lookupの防止

下記のようにクラスや関数テンプレートが定義されている場合を考える。

```
// @@@ example/template/suppress_adl_ut.cpp 11

namespace App {

struct XY {
    int x;
    int y;
};

// このような関数テンプレートは適用範囲が広すぎる所以定義すべきではないが、
// 危険な例を示すためあえて定義している
template <typename T, typename U>
inline auto is_equal(T const& lhs, U const& rhs) noexcept
    -> decltype(lhs.x == rhs.x, lhs.y == rhs.y)
{
    return lhs.x == rhs.x && lhs.y == rhs.y;
}
} // namespace App
```

これに対しての単体テストは下記のようになる。

```
// @@@ example/template/suppress_adl_ut.cpp 37

auto xy0 = App::XY{0, 1};
auto xy1 = App::XY{0, 2};
auto xy2 = App::XY{0, 1};

ASSERT_FALSE(is_equal(xy0, xy1));
ASSERT_TRUE(is_equal(xy0, xy2));

struct point {
    int x;
    int y;
};
auto p0 = point{0, 1};

// 下記のような比較ができるようにするためにis_equalはテンプレートで実装している
ASSERT_TRUE(is_equal(p0, xy0));
ASSERT_FALSE(is_equal(p0, xy1));
```

上記の抜粋である

```
// @@@ example/template/suppress_adl_ut.cpp 43

ASSERT_FALSE(is_equal(xy0, xy1));
ASSERT_TRUE(is_equal(xy0, xy2));
```

が名前空間Appの指定なしでコンパイルできる理由は、[ADL\(実引数依存探索\)](#)により、Appもis_equalのname lookupの対象になるからである。これは便利な機能であるが、その副作用として意図しないname lookupによるバグの混入を起こしてしまうことがある。

上記の名前空間での定義が可視である状態で、下記のようなコードを書いた場合を考える。

```
// @@@ example/template/suppress_adl_ut.cpp 63

namespace App2 {
struct XYZ {
    int x;
```

```

        int y;
        int z;
    };

    inline bool is_equal(XYZ const& lhs, XYZ const& rhs) noexcept
    {
        return lhs.x == rhs.x && lhs.y == rhs.y && lhs.z == rhs.z;
    }
} // namespace App2

```

この単体テストは、やはりADLを使い下記のように書ける。

```

// @@@ example/template/suppress_adl_ut.cpp 84
auto xyz0 = App2::XYZ{0, 2, 2};
auto xyz1 = App2::XYZ{0, 1, 2};

ASSERT_TRUE(is_equal(xyz0, xyz0));
ASSERT_FALSE(is_equal(xyz0, xyz1));

```

これに問題はないが、下記のテストもコンパイルでき、且つテストもパスしてしまうことには問題がある。

```

// @@@ example/template/suppress_adl_ut.cpp 93
auto xyz0 = App2::XYZ{0, 2, 2};
auto xyz1 = App2::XYZ{0, 1, 2};
auto xy0   = App::XY{0, 1};

ASSERT_FALSE(is_equal(xy0, xyz0)); // これがコンパイルできてしまう
ASSERT_TRUE(is_equal(xy0, xyz1)); // このis_equalはAppで定義されたもの

```

このセマンティクス的に無意味な(もしくは混乱を引き起こしてしまうであろう)コードは、

- is_equalの引数の型XY、XYZはそれぞれ名前空間App、App2で定義されている
- 従って、ADLによりis_equalのname lookupには名前空間App、App2も使われる
- 引数の型XY、XYZを取り得るis_equalはAppで定義されたもののみである

というメカニズムによりコンパイルできてしまう。

こういったname lookup、特にADLの問題に対処する方法は、

- ジェネリックすぎるテンプレートを書かない
- ADLが本当に必要でない限り名前を修飾する
- ADL Firewallを使う

のようにいくつか考えられる。これらについて以下で説明を行う。

ジェネリックすぎるテンプレートを書かない

ここで「ジェネリックすぎるテンプレート」とは、シンタックス的には適用範囲が広いにもかかわらず、セマンティクス的な適用範囲は限られているものを指す。従って下記のような関数テンプレートを指す概念ではない。

```

// @@@ example/template/suppress_adl_ut.cpp 108

template <typename T, size_t N>
constexpr auto array_length(T const (&)[N]) noexcept
{
    return N;
}

```

前記で問題を起こした関数テンプレート

```

// @@@ example/template/suppress_adl_ut.cpp 20

// このような関数テンプレートは適用範囲が広すぎるので定義すべきではないが、
// 危険な例を示すためあえて定義している
template <typename T, typename U>
inline auto is_equal(T const& lhs, U const& rhs) noexcept
    -> decltype(lhs.x == rhs.x, lhs.y == rhs.y)
{
    return lhs.x == rhs.x && lhs.y == rhs.y;
}
} // namespace App

```

が典型的な「ジェネリックすぎるテンプレート」である。これに対する最も安全な対処は下記コードで示す通りテンプレートを使わないことである。

```
// @@@ example/template/suppress_adl_ut.cpp 126

namespace App {

struct XY {
    int x;
    int y;
};

inline bool is_equal(XY const& lhs, XY const& rhs) noexcept
{
    return lhs.x == rhs.x && lhs.y == rhs.y;
} // namespace App
```

ジェネリックな`is_equal`が必要であれば下記単体テストのように ジェネリックラムダを使えばよい。こうすることでその適用範囲はそれを定義した関数内に留まる。

```
// @@@ example/template/suppress_adl_ut.cpp 153

// 下記のpointのようなクラスが他にもいくつかあった場合、
// このジェネリックラムダでコードの被りは回避できる
auto is_equal = [](auto const& lhs, auto const& rhs) noexcept {
    return lhs.x == rhs.x && lhs.y == rhs.y;
};

struct point {
    int x;
    int y;
};
auto p0 = point{0, 1};

ASSERT_TRUE(is_equal(p0, xy0));
ASSERT_FALSE(is_equal(p0, xy1));
```

上記で示した

- テンプレートを使わない
- 適用範囲の広いテンプレート(ジェネリック)に対してはアクセスできる箇所を局所化する

といった方法の他にも、「コンテナ用Nstd::operator<<の開発」で示した

- `std::enable_if`等を使用してテンプレートに適用できる型を制限する

ことも考えられる。ベストな方法は状況に大きく依存するため一概には決められない。その状況でのもっとも単純は方法を選ぶべきだろう(が、何が単純かも一概に決ることは難しい)。

ADLが本当に必要でない限り名前を修飾する

下記のコードについて考える。

```
// @@@ example/template/suppress_adl_ut.cpp 176

struct A {
    int f(int i) noexcept { return i * 3; }
};

int f(int i) noexcept { return i * 2; }

namespace App {

template <typename T>
class ExecF : public T {
public:
    int operator()(int i) noexcept
    {
        return f(i); // T::fの呼び出しにも見えるが、::fの呼び出し
    }

    // Tを使ったコード
    ...
};

} // namespace App
```

基底クラスのメンバ関数を呼び出す場合は、`T::f()`、もしくは、`this->f()`と書く必要があるため、下記コードで呼び出した関数`f`は外部関数`f`の呼び出しになる(two phase name lookupの一回目のname lookupで`f`がバインドされるため)。

```
// @@@ example/template/suppress_adl_ut.cpp 203

auto ef = App::ExecF<A>{};

ASSERT_EQ(4, ef(2)); // ::fの呼び出しなので、2 * 2 == 4となる
```

これだけでも十分わかりづらいが、 ExecFのテンプレートパラメータにはクラスAしか使われないことがわかったので、 下記のようにリファクタリングしたとしよう。

```
// @@@ example/template/suppress_adl_ut.cpp 213

struct A {
    int f(int i) noexcept { return i * 3; }
};

int f(int i) noexcept { return i * 2; }

namespace App {

class ExecF : public A {
public:
    int operator()(int i) noexcept { return f(i); }

    // Tを使ったコード
    ...
};

} // namespace App
```

すると、 fのname lookupの対象が変わってしまい、元の単体テストはパスしなくなる。

```
// @@@ example/template/suppress_adl_ut.cpp 236

auto ef = App::ExecF{};

// ASSERT_EQ(4, ef(2));
ASSERT_EQ(6, ef(2)); // リファクタリングでname lookupの対象が変わり、A::fが呼ばれる
```

こういった場合に備え単体テストを実行すべきなのだが、この程度の問題はコンパイルで検出したい。[ADL](#)や[two_phase_name_lookup](#)が絡む場合ならなおさらである。

こういう意図しないname lookupに備えるためには、修飾されていない識別子を使わないこと、つまり、識別子には、名前空間、クラス名、this->等による修飾を施すことが重要である。

ただし、「[コンテナ用Nstd::operator<<の開発](#)」で示したコード等にはADLが欠かせないため、修飾することをルール化することはできない。場合に合わせた運用が唯一の解となる。

ADL Firewallを使う

下記のコードについて考える。

```
// @@@ example/template/adl_firewall_0_ut.cpp 10

namespace App {

template <typename T>
std::string ToString(std::vector<T> const& t)
{
    auto oss = std::ostringstream{};

    using Nstd::operator<<;
    oss << t; // Nstd::operator<<もname lookupの対象に含める

    return oss.str();
}
} // namespace App

...

namespace App {
struct XY {
    XY(int ax, int ay) noexcept : x{ax}, y{ay} {}
    int x;
    int y;
};

std::ostream& operator<<(std::ostream& os, XY const& xyz)
{
```

```

        return os << std::to_string(xyz.x) + "/" + std::to_string(xyz.y);
    }
} // namespace App

```

上記のApp::ToStringは「[コンテナ用Nstd::operator<<の開発](#)」のコードを使用し、 std::vectorオブジェクトをstd::stringに変換する。

これに対しての単体テストは下記のようになる。

```

// @@@ example/template/adl_firewall_0_ut.cpp 47

auto xys = std::vector<App::XY>{{9, 8}, {7, 6}};

ASSERT_EQ("9/8, 7/6", App::ToString(xys));

```

これは想定通りの動作だが、上記のAppの後に下記のコードを追加するとApp::ToStringは影響を受ける。

```

// @@@ example/template/adl_firewall_1_ut.cpp 40

// Appに下記を追加
namespace App {
template <typename T>
std::ostream& operator<<(std::ostream& os, std::vector<T> const& t)
{
    return os << "size:" << t.size();
}
} // namespace App

```

これにより元の単体テストはエラーとなり、新しい単体テストは下記のようになる。

```

// @@@ example/template/adl_firewall_1_ut.cpp 56

auto xys = std::vector<App::XY>{{9, 8}, {7, 6}};

// App::operator<<の追加で、App::ToStringの出力が影響を受ける
// ASSERT_EQ("9/8, 7/6", App::ToString(xys));
ASSERT_EQ("size:2", App::ToString(xys));

```

これが意図通りなら問題ないが、ここでは「新たに追加した関数テンプレートApp::operator<<はstd::vector<App::XY>用ではなかった」としよう。その場合、これは意図しないADLによるバグの混入となる。「[ジェネリックすぎるテンプレートを書かない](#)」で述べたように追加した関数テンプレートの適用範囲が広すぎることが原因であるが、XY型から生成されたオブジェクト(std::vector<App::XY>も含む)によるADLのため、Appの宣言がname lookupの対象になったことにも原因がある。

下記のコードは後者の原因を解消する。

```

// @@@ example/template/adl_firewall_2_ut.cpp 23

// Appの中の新たな名前空間XY_Firewall_でstruct XYとoperator<<を宣言
namespace App {
namespace XY_Firewall_ {

struct XY {
    XY(int ax, int ay) noexcept : x{ax}, y{ay} {}
    int x;
    int y;
};

std::ostream& operator<<(std::ostream& os, XY const& xyz)
{
    return os << std::to_string(xyz.x) + "/" + std::to_string(xyz.y);
}
} // namespace XY_Firewall_

using XY = XY_Firewall_::XY;

} // namespace App

```

XY型オブジェクトを引数にした関数呼び出しによる関連名前空間は、極小なApp::XY_Firewall_であるため、意図しないADLは起こりづらく、起こっても発見しやすい。また、XY型用operator<<もApp::XY_Firewall_で定義し、App内でusing XYを宣言したことでのこれまで通りApp::XYが使える。

このようなテクニックをADL firewallと呼ぶ。

Nstd::Type2Strの開発

「[Nstdライブラリの開発](#)」等で行ったメタ関数の実装は、

- 入り組んだ<>や()の対応漏れ
- &や&&のつけ忘れ
- typenameやtemplateキーワードの漏れ
- メタ関数メンバー::valueや::typeの漏れ

等によるコンパイルエラーとの戦いである。また、これをクリアしてもtwo phase name lookupやADLが次の関門になる。これには、デバッガのステップ実行が強力な武器となるが、型を文字列に変換する関数があればこれもまた強力な武器となる。

以下に示すNstd::Type2Strは、「[Nstdライブラリの開発](#)」等で実際に使用したそのような関数である。

```
// @@@ h/nstd_type2str.h 9

namespace Nstd {
namespace Inner_ {

inline std::string demangle(char const* to_demangle)
{
    int status;

    std::unique_ptr<char, decltype(&std::free)> demangled{
        abi::__cxa_demangle(to_demangle, 0, 0, &status), &std::free};

    return demangled.get();
}

template <typename> // typenameを取り出すためだけのクラステンプレート
struct type_capture {
};
} // namespace Inner_

template <typename T>
std::string Type2Str()
{
    // typeid(T)とした場合、const/volatile/&の情報が捨てられるため、
    // typeid(type_capture<T>)とし、それを防ぐ。
    auto str = std::string{Inner_::demangle(typeid(Inner_::type_capture<T>).name())};

    // T == const int ならば、
    // str == Nstd::Inner_::type_capture<int const>
    //           <----- 27 -----><-- x --> 下記ではxを切り出す
    constexpr auto beg = 27; // 先頭の不要な文字列数
    auto name = str.substr(beg, str.size() - beg - 1); // 最後の文字は>なので不要

    while (name.back() == ' ') { // 無駄なスペースを消す
        auto last = --name.end();
        name.erase(last);
    }

    return name;
}
} // namespace Nstd
```

typeid::name()が返す文字列リテラルは引数の型の文字列表現を持つが、マングリングされているためヒューマンリーダブルではない。それをデマングルするのがabi::__cxa_demangleであるが、残念なことにこの関数は非標準であるため、それを使っているNstd::Inner_::demangleはg++/clang++でなければコンパイルできないだろう。

それを除けば、複雑なシンタックスを持つ型を文字列で表現できるNstd::Type2Strは、テンプレートプログラミングにおける有効なデバッグツールであると言える。

下記単体テストは、そのことを示している。

```
// @@@ example/template/nstd_type2str_ut.cpp 11

ASSERT_EQ("int const", Nstd::Type2Str<int const>());
ASSERT_EQ("std::string", Nstd::Type2Str<std::string>());
ASSERT_EQ("std::vector<int, std::allocator<int>>", Nstd::Type2Str<std::vector<int>>());

extern void f(int);
ASSERT_EQ("void (int)", Nstd::Type2Str<decltype(f)>()); // 関数の型

auto lambda = []() noexcept {};
ASSERT_NE("", Nstd::Type2Str<decltype(lambda)>()); // XXX::{lambda()#1}な感じになる

ASSERT_EQ("std::ostream& (std::ostream&, int const (&) [3])",
Nstd::Type2Str<decltype(Nstd::operator<< <int[3]>>())>();

// std::declvalはrvalueリファレンスを返す
```

```

ASSERT_EQ("int (&&) [3]", Nstd::Type2Str<decltype(std::declval<int[3]>())>());
int i3[3];
ASSERT_EQ("int [3]", Nstd::Type2Str<decltype(i3)>());
ASSERT_EQ("int (&) [3]", Nstd::Type2Str<decltype((i3))>()); // (i3)はlvalueリファレンス
auto& r = i3;
ASSERT_EQ("int (&) [3]", Nstd::Type2Str<decltype(r)>());

```

静的な文字列オブジェクト

`std::string`は文字列を扱うことにおいて、非常に有益なクラスではあるが、コンパイル時に文字列が決定できる場合でも、動的にメモリを確保する。

この振る舞いは、

- ・ ランタイム時に`new/delete`を行うため、処理の遅さにつながる。
- ・ 下記のようにエクゼプションオブジェクトにファイル位置を埋め込むことは、デバッグに便利であるが、メモリ確保失敗を通知するような場面ではこの方法は使えない。

```

// @@@ example/template/nstd_exception_ut.cpp 6

class Exception : std::exception {
public:
    Exception(char const* filename, uint32_t line_num, char const* msg)
        : what_str_{std::string{filename} + ":" + std::to_string(line_num) + ":" + msg}
    {
    }

    char const* what() const noexcept override { return what_str_.c_str(); }

private:
    std::string what_str_;
};

int32_t div(int32_t a, int32_t b)
{
    if (b == 0) {
        throw Exception{__FILE__, __LINE__, "divided by 0"}; // 24行目
    }

    return a / b;
}

```

```

// @@@ example/template/nstd_exception_ut.cpp 34

auto caught = false;
try {
    div(1, 0);
}
catch (Exception const& e) {
    ASSERT_STREQ("nstd_exception_ut.cpp:24:divided by 0", e.what());
    caught = true;
}
ASSERT_TRUE(caught);

```

このような問題を回避するために、ここでは静的に文字列を扱うためのクラス`StaticString`を開発する。

StaticStringのヘルパークラスの開発

`StaticString`オブジェクトは、`char`配列をメンバとして持つが、コンパイル時に解決できる配列の初期化にはパラメータパックが利用できる。そのパラメータパック生成クラスを下記のように定義する。

```

// @@@ example/template/nstd_seq.h 4

// パラメータパック展開ヘルパークラス
template <size_t... Ns>
struct index_sequence {

    // index_sequence<0, 1, 2, ...>を作るためのクラステンプレート
    // make_index_sequence<3>
    // -> make_index_sequence<2, 2>
    // -> make_index_sequence<1, 1, 2>
    // -> make_index_sequence<0, 0, 1, 2>
    // -> index_sequence<0, 1, 2>
};

```

```

template <size_t N, size_t... Ns>
struct make_index_sequence : make_index_sequence<N - 1, N - 1, Ns...> {
};

template <size_t... Ns>
struct make_index_sequence<0, Ns...> : index_sequence<Ns...> {
};

```

このクラスにより、下記のような配列メンバの初期ができるようになる。

```

// @@@ example/template/nstd_seq_ut.cpp 7

template <size_t N>
struct seq_test {
    template <size_t... S>
    constexpr seq_test(index_sequence<S...>) noexcept : data{S...}
    {
    }
    int const data[N];
};

```

```

// @@@ example/template/nstd_seq_ut.cpp 24

constexpr auto st = seq_test<3>{index_sequence<1, 2, 3>()};
ASSERT_EQ(1, st.data[0]);
ASSERT_EQ(2, st.data[1]);
ASSERT_EQ(3, st.data[2]);

```

これを下記のように使うことで、メンバである文字列配列のコンパイル時初期化ができるようになる。

```

// @@@ example/template/nstd_seq_ut.cpp 33

template <size_t N>
class seq_test2 {
public:
    template <size_t... S>
    constexpr seq_test2(char const (&str)[N], index_sequence<S...>) noexcept : string_{str[S]...}
    {
    }

    constexpr char const (&String() const noexcept)[N] { return string_; }

private:
    char const string_[N];
};

```

```

// @@@ example/template/nstd_seq_ut.cpp 52

constexpr char const str[]{"123"};

constexpr auto st = seq_test2<4>{str, index_sequence<0, 1, 2>()};
ASSERT_STREQ("123", st.String());

constexpr auto st2 = seq_test2<4>{str, make_index_sequence<sizeof(str) - 1>()};
ASSERT_STREQ("123", st2.String());

```

上記とほぼ同様のクラステンプレートstd::index_sequence、std::make_index_sequenceが、utilityで定義されているため、以下ではこれらを使用する。

StaticStringの開発

StaticStringはすでに示したテクニックを使い、下記のように定義できる。

```

// @@@ example/h/nstd_static_string.h 8

template <size_t N>
class StaticString {
public:
    constexpr StaticString(char const (&str)[N]) noexcept
        : StaticString{str, std::make_index_sequence<N - 1>{}}
    {
    }

    constexpr StaticString(std::initializer_list<char> args) noexcept
        : StaticString{args, std::make_index_sequence<N - 1>{}}
    {
    }

    constexpr char const (&String() const noexcept)[N] { return string_; }
};

```

```

    constexpr size_t Size() const noexcept { return N; }

private:
    char const string_[N];

    template <typename T, size_t... I>
    constexpr StaticString(T& t, std::index_sequence<I...>) noexcept : string_{std::begin(t)[I]...}
    {
        static_assert(
            std::is_same_v<T, std::initializer_list<char>> || std::is_same_v<T, char const[N]>);
        static_assert(N - 1 == sizeof...(I));
    }
};

```

文字列リテラルからStaticStringを生成する単体テストは下記のようになる。

```

// @@@ example/template/nstd_static_string_ut.cpp 11

const auto fs = StaticString{"abc"}; // C++17からのNの指定は不要

static_assert(sizeof(4) == fs.Size());
ASSERT_STREQ("abc", fs.String());

// 文字列不足であるため、下記はコンパイルさせない
// constexpr StaticString<4> fs2{"ab"};

```

また、std::initializer_list<char>による初期化の単体テストは下記のようになる。

```

// @@@ example/template/nstd_static_string_ut.cpp 23

const auto fs = StaticString<4>{'a', 'b', 'c'}; // C++17でもNの指定は必要

static_assert(sizeof(4) == fs.Size());
ASSERT_STREQ("abc", fs.String());

// 文字列不足であるため、下記はコンパイルさせない
// constexpr StaticString<4> fs2{'a', 'b'};

```

次にこのクラスにoperator==を追加する。

```

// @@@ example/h/nstd_static_string.h 38

namespace Inner_ {
template <size_t N>
constexpr bool equal_n(size_t n, StaticString<N> const& lhs, StaticString<N> const& rhs) noexcept
{
    if (n == N) {
        return true;
    }
    else {
        return lhs.String()[n] != rhs.String()[n] ? false : equal_n(n + 1, lhs, rhs);
    }
} // namespace Inner_

template <size_t N1, size_t N2>
constexpr bool operator==(StaticString<N1> const&, StaticString<N2> const&) noexcept
{
    return false;
}

template <size_t N1, size_t N2>
constexpr bool operator!=(StaticString<N1> const& lhs, StaticString<N2> const& rhs) noexcept
{
    return !(lhs == rhs);
}

template <size_t N>
constexpr bool operator==(StaticString<N> const& lhs, StaticString<N> const& rhs) noexcept
{
    return Inner_::equal_n(0, lhs, rhs);
}

template <size_t N>
constexpr bool operator!=(StaticString<N> const& lhs, StaticString<N> const& rhs) noexcept
{
    return !(lhs == rhs);
}

template <size_t N1, size_t N2>

```

```

constexpr bool operator==(StaticString<N1> const& lhs, char const (&rhs)[N2]) noexcept
{
    return lhs == StaticString{rhs};
}

template <size_t N1, size_t N2>
constexpr bool operator!=(StaticString<N1> const& lhs, char const (&rhs)[N2]) noexcept
{
    return !(lhs == rhs);
}

template <size_t N1, size_t N2>
constexpr bool operator==(char const (&lhs)[N1], StaticString<N2> const& rhs) noexcept
{
    return StaticString{lhs} == rhs;
}

template <size_t N1, size_t N2>
constexpr bool operator!=(char const (&lhs)[N1], StaticString<N2> const& rhs) noexcept
{
    return !(lhs == rhs);
}

```

単体テストは下記のようになる。

```

// @@@ example/template/nstd_static_string_ut.cpp 38

static_assert(StaticString{"123"} == StaticString{"123"});
static_assert(StaticString{"123"} != StaticString{"1234"});
static_assert(StaticString{"123"} == "123");
static_assert("123" == StaticString{"123"});
static_assert(StaticString{"123"} != "1234");
static_assert("1234" != StaticString{"123"});

```

非explicitなコンストラクタによる暗黙の型変換を利用した文字列リテラルからStaticStringオブジェクトへの変換は、StaticStringがテンプレートであるため機能せず、上記のように書く必要がある。

同様にoperator+を追加する。

```

// @@@ example/h/nstd_static_string.h 101

namespace Inner_ {
template <size_t N1, size_t... I1, size_t N2, size_t... I2>
constexpr StaticString<N1 + N2 - 1> concat(char const (&str1)[N1], std::index_sequence<I1...>,
                                             char const (&str2)[N2],
                                             std::index_sequence<I2...>) noexcept
{
    return {str1[I1]..., str2[I2]...};
} // namespace Inner_

template <size_t N1, size_t N2>
constexpr auto operator+(StaticString<N1> const& lhs, StaticString<N2> const& rhs) noexcept
{
    return Inner_::concat(lhs.String(), std::make_index_sequence<N1 - 1>{}, rhs.String(),
                         std::make_index_sequence<N2>{});
}

template <size_t N1, size_t N2>
constexpr auto operator+(StaticString<N1> const& lhs, char const (&rhs)[N2]) noexcept
{
    return lhs + StaticString{rhs};
}

template <size_t N1, size_t N2>
constexpr auto operator+(char const (&lhs)[N1], StaticString<N2> const& rhs) noexcept
{
    return StaticString{lhs} + rhs;
}

// @@@ example/template/nstd_static_string_ut.cpp 51

constexpr auto fs0 = StaticString{"1234"} + StaticString{"567"};
static_assert(std::is_same_v<StaticString<fs0.Size()>, decltype(fs0)>);
static_assert("1234567" == fs0);

constexpr auto fs1 = StaticString{"1234"} + ":";  

static_assert(std::is_same_v<StaticString<fs1.Size()>, decltype(fs1)>);
static_assert("1234:" == fs1);

```

```

constexpr auto fs2 = ":" + StaticString{"567"};
static_assert(std::is_same_v<StaticString<fs2.Size()> const, decltype(fs2)>);
static_assert(":567" == fs2);

constexpr auto fs3 = StaticString("1234") + ":" + StaticString{"567"};
static_assert(std::is_same_v<StaticString<fs3.Size()> const, decltype(fs3)>);
static_assert("1234:567" == fs3);

```

整数をStaticStringに変換する関数の開発

コンパイル時に`__LINE__`をStaticStringに変換できれば、ファイル位置をStaticStringで表現できるため、ここではその変換関数`Int2StaticString<>()`の実装を行う。

行番号を10進数での文字列で表現するため、いくつかのヘルパー関数を下記のように定義する。

```

// @@@ example/h/nstd_static_string_num.h 8

namespace Inner_ {

// 10進数桁数を返す
constexpr size_t num_of_digits(size_t n) noexcept { return n > 0 ? 1 + num_of_digits(n / 10) : 0; }

// 10のn乗を返す
constexpr uint32_t ten_to_nth_power(uint32_t n) noexcept
{
    return n == 0 ? 1 : 10 * ten_to_nth_power(n - 1);
}

// 10進数の桁の若い順番に左から並べなおす(12345 -> 54321)
constexpr uint32_t reverse_num(uint32_t num) noexcept
{
    return num != 0 ? (num % 10) * ten_to_nth_power(num_of_digits(num) - 1) + reverse_num(num / 10)
                    : 0;
}

// 10進数一行をascii文字に変換
constexpr char digit_to_char(uint32_t num, uint32_t n_th) noexcept
{
    return '0' + (num % (ten_to_nth_power(n_th + 1))) / ten_to_nth_power(n_th);
}

// Int2StaticStringのヘルパー関数
template <size_t N, size_t... Cs>
constexpr StaticString<num_of_digits(N) + 1> make_static_string(std::index_sequence<Cs...>) noexcept
{
    return {digit_to_char(reverse_num(N), Cs)...};
}
} // namespace Inner_

```

単体テストは下記のようになる。

```

// @@@ example/template/nstd_static_string_num_ut.cpp 47

constexpr auto ns
= make_static_string<__LINE__>(std::make_index_sequence<Inner_::num_of_digits(__LINE__)>());
auto line_num = __LINE__ - 1;

ASSERT_EQ(std::to_string(line_num), ns.String());

```

このままで使いづらいため、これをラッピングした関数を下記のように定義することで、`Int2StaticString<>()`が得られる。

```

// @@@ example/h/nstd_static_string_num.h 42

template <size_t N>
constexpr StaticString<Inner_::num_of_digits(N) + 1> Int2StaticString() noexcept
{
    return Inner_::make_static_string<N>(std::make_index_sequence<Inner_::num_of_digits(N)>());
}

```

単体テストは下記のようになる。

```

// @@@ example/template/nstd_static_string_num_ut.cpp 66

constexpr auto ns      = Int2StaticString<__LINE__>();
auto       line_num = __LINE__ - 1;

ASSERT_EQ(std::to_string(line_num), ns.String());

```

ファイル位置を静的に保持したエクセプションクラスの開発

「静的な文字列オブジェクト」で見たように、ファイル位置を動的に保持するエクセプションクラスは使い勝手が悪い。ここでは、その問題を解決するためのExceptionクラスの実装を示す。

```
// @@@ example/h/nstd_exception.h 10

/// @class Exception
/// @brief StaticString<>を使ったエクセプションクラス
/// 下記のMAKE_EXCEPTIONを使い生成
/// @tparam E std::exceptionから派生したエクセプションクラス
/// @tparam N StaticString<N>
template <class E, size_t N>
class Exception : public E {
public:
    static_assert(std::is_base_of_v<std::exception, E>);

    Exception(StaticString<N> const& what_str) noexcept : what_str_{what_str} {}
    char const* what() const noexcept override { return what_str_.String(); }

private:
    StaticString<N> const what_str_;
};
```

StaticStringと同様に、このままで不便であるため、下記の関数を定義する。

```
// @@@ example/h/nstd_exception.h 29

namespace Inner_ {
template <class E, template <size_t> class STATIC_STR, size_t N>
auto make_exception(STATIC_STR<N> exception_str) noexcept
{
    return Exception<E, N>{exception_str};
}
} // namespace Inner_

template <class E, size_t LINE_NUM, size_t F_N, size_t M_N>
auto MakeException(char const (&filename)[F_N], char const (&msg)[M_N]) noexcept
{
    return Inner_::make_exception<E>(StaticString{filename} + ":" + Int2StaticString<LINE_NUM>()
        + ":" + msg);
}
```

単体テストは下記のようになる。

```
// @@@ example/template/nstd_exception_ut.cpp 89

auto caught = false;
auto line_num = __LINE__ + 2; // 2行下の行番号
try {
    throw MakeException<std::exception, __LINE__>(__FILE__, "some error message");
}
catch (std::exception const& e) {
    auto oss = std::ostringstream{};
    oss << __FILE__ << ":" << line_num << ":some error message";

    ASSERT_EQ(oss.str(), e.what());
    caught = true;
}

ASSERT_TRUE(caught);
```

Exceptionクラスの利便性をさらに高めるため、下記の定義を行う。

```
// @@@ example/h/nstd_exception.h 48

#define MAKE_EXCEPTION(E__, msg__) Nstd::MakeException<E__, __LINE__>(__FILE__, msg__)
```

上記は、関数型マクロの数少ない使いどころである。

単体テストは下記のようになる。

```
// @@@ example/template/nstd_exception_ut.cpp 109

uint32_t line_num_div; // エクセプション行を指定

int32_t div(int32_t a, int32_t b)
{
    if (b == 0) {
```

```

        line_num_div = __LINE__ + 1; // 次の行番号
        throw MAKE_EXCEPTION(std::exception, "divided by 0");
    }

    return a / b;
}

// @@@ example/template/nstd_exception_ut.cpp 126

auto caught = false;

try {
    div(1, 0);
}
catch (std::exception const& e) { // リファレンスでcatchしなければならない
    auto oss = std::ostringstream{};
    oss << __FILE__ << ":" << line_num_div << ":divided by 0";
    ASSERT_EQ(oss.str(), e.what());
    caught = true;
}

ASSERT_TRUE(caught);

```

関数型をテンプレートパラメータで使う

ここで使う「関数型」とは、

- 関数へのポインタの型
- クロージャの型、もしくはラムダ式の型
- 関数オブジェクトの型

の総称を指す。

`std::unique_ptr`は、

- 第1パラメータにポインタの型
- 第2パラメータにそのポインタの解放用の関数ポインタの型

を取ることができるが、通常は第2パラメータは省略される。省略時には`std::default_delete`が割り当てられ、そのオブジェクトによって、第1パラメータに対応するポインタが`delete`される。

下記コードではこの第2パラメータに`std::free`のポインタの型を与え、それから生成される`std::unique_ptr`オブジェクトを、

- `abi::__cxa_demangle`が`std::malloc`で取得した`char`型ポインタ
- `std::free`のポインタ

で初期化することでメモリの解放を行っている。

```

// @@@ h/nstd_type2str.h 18

std::unique_ptr<char, decltype(&std::free)> demangled{
    abi::__cxa_demangle(to_demangle, 0, 0, &status), &std::free};

```

`std::unique_ptr`の第2パラメータには、上記のような関数へのポインタのみではなく、関数型を取ることができます。

そのことを順を追って示す。まずは、`std::unique_ptr`の動作を確かめるためのクラスを下記のように定義する。

```

// @@@ example/template/func_type_ut.cpp 10

// デストラクタが呼び出された時に、外部から渡されたフラグをtrueにする
struct A {
    explicit A(bool& destructor_called) noexcept : destructor_called{destructor_called} {}
    ~A() { destructor_called = true; }

    bool& destructor_called;
};

```

次に示すのは、第2パラメータに何も指定しないパターンである。テスト用クラスAの動作確認ができるはずである。

```

// @@@ example/template/func_type_ut.cpp 27

{ // 第2パラメータに何も指定しない
    auto is_called = false;
    {
        auto ua = std::unique_ptr<A>{new A{is_called}};

```

```

        ASSERT_FALSE(is_called); // uaのデストラクタは呼ばれていない
    }
    ASSERT_TRUE(is_called); // uaのデストラクタは呼ばれた
}

```

次に示すのは、

```

// @@@ example/template/func_type_ut.cpp 20

void delete_func(A* a) noexcept { delete a; }

```

のポインタをstd::unique_ptrの第2パラメータに与えた例である。

```

// @@@ example/template/func_type_ut.cpp 38

{ // 第2パラメータに関数ポインタを与える
    auto is_called = false;
    {
        auto ua = std::unique_ptr<A, void (*)(A*)>{new A{is_called}, &delete_func};
        ASSERT_FALSE(is_called); // uaのデストラクタは呼ばれていない
    }
    ASSERT_TRUE(is_called); // uaのデストラクタは呼ばれた
}

```

次に示すのは、std::unique_ptrの第2パラメータにラムダを与えた例である。

```

// @@@ example/template/func_type_ut.cpp 49

{ // 第2パラメータにラムダを与える
    auto is_called = false;
    {
        auto delete_lambda = [](A* a) noexcept { delete a; };

        // ラムダ式の型はインスタンス毎に異なるため、
        // ラムダ式の型を取得するためには下記のように decltypeを使う必要がある
        auto ua = std::unique_ptr<A, decltype(delete_lambda)>{new A{is_called}, delete_lambda};
        ASSERT_FALSE(is_called); // uaのデストラクタは呼ばれていない
    }
    ASSERT_TRUE(is_called); // uaのデストラクタは呼ばれた
}

```

次に示すのは、std::unique_ptrの第2パラメータに関数型オブジェクトの型(std::function)を与えた例である。

```

// @@@ example/template/func_type_ut.cpp 64

{ // 第2パラメータにstd::function型オブジェクトを与える
    auto is_called = false;
    {
        auto delete_obj = std::function<void(A*)>{[](A* a) noexcept { delete a; }};
        auto ua = std::unique_ptr<A, std::function<void(A*)>&>{new A{is_called}, delete_obj};
        ASSERT_FALSE(is_called); // uaのデストラクタは呼ばれていない
    }
    ASSERT_TRUE(is_called); // uaのデストラクタは呼ばれた
}

```

以上で見てきたようにstd::unique_ptrの第2パラメータには、第1パラメータのポインタを引数に取る関数型であれば指定できる。

このようなテンプレートパラメータを持つクラステンプレートの実装例を示すため、「[RAII\(scoped_guard\)](#)」でも示したScopedGuardの実装を下記する。

やや意外だが、このようなテンプレートパラメータに特別な記法はなく、以下のようにすれば良い。

```

// @@@ h/scoped_guard.h 4

/// @class ScopedGuard
/// @brief RAIのためのクラス。
///       コンストラクタ引数の関数オブジェクトをデストラクタから呼び出す。
template <typename F>
class ScopedGuard {
public:
    explicit ScopedGuard(F&& f) noexcept : f_{f}
    {
        // f()がill-formedにならず、その戻りがvoidでなければならぬ
        static_assert(std::is_invocable_r_v<void, F>, "F must be callable and return void");
    }

    ~ScopedGuard() { f_(); }
    ScopedGuard(ScopedGuard const&) = delete; // copyは禁止
    ScopedGuard& operator=(ScopedGuard const&) = delete; // copyは禁止
}

```

```
private:  
    F f_;  
};
```

上記コードの抜粋である下記は、テンプレートパラメータである関数型を規定するものである。

```
// @@@ h/scoped_guard.h 15  
  
// f()がill-formedにならず、その戻りがvoidでなければならぬ  
static_assert(std::is_invocable_r_v<void, F>, "F must be callable and return void");
```

これがなければ、誤った型の関数型をテンプレートパラメータに指定できてしまう。

以下にこのクラステンプレートの単体テストを示す。

まずは、以下の関数と静的変数の組み合わせ

```
// @@@ example/template/func_type_ut.cpp 80  
  
bool is_caleded_in_static{false};  
void caleded_by_destructor() noexcept { is_caleded_in_static = true; }
```

を使った例である。

```
// @@@ example/template/func_type_ut.cpp 88  
  
{ // Fに関数ポインタを与える  
    is_caleded_in_static = false;  
    {  
        auto sg = ScopedGuard{&caleded_by_destructor};  
        ASSERT_FALSE(is_caleded_in_static); // sgのデストラクタは呼ばれていない  
    }  
    ASSERT_TRUE(is_caleded_in_static); // sgのデストラクタは呼ばれた  
}
```

次に示すのは、それぞれにラムダ式とstd::functionを使った2例である。

```
// @@@ example/template/func_type_ut.cpp 103  
  
{ // Fにラムダ式を与える  
    auto is_called = false;  
    {  
        auto gs = ScopedGuard{[&is_called]() noexcept { is_called = true; }};  
        ASSERT_FALSE(is_called); // sgのデストラクタは呼ばれていない  
    }  
    ASSERT_TRUE(is_called); // sgのデストラクタは呼ばれた  
}  
{ // Fにstd::function型オブジェクトを与える  
    auto is_called = false;  
    {  
        auto f = std::function<void(void)>{[&is_called]() noexcept { is_called = true; }};  
        auto gs = ScopedGuard{std::move(f)}; // sgのデストラクタは呼ばれていない  
        ASSERT_FALSE(is_called); // sgのデストラクタは呼ばれた  
    }  
    ASSERT_TRUE(is_called);  
}
```

次に示すのは関数型オブジェクト

```
// @@@ example/template/func_type_ut.cpp 125  
  
struct TestFunctor {  
    explicit TestFunctor(bool& is_called) : is_called_{is_called} {}  
    void operator()() noexcept { is_called_ = true; }  
    bool& is_called_;  
};
```

を使った例である。

```
// @@@ example/template/func_type_ut.cpp 136  
  
{ // Fに関数型オブジェクトを与える  
    auto is_called = false;  
    auto tf = TestFunctor{is_called};  
    {  
        auto sg = ScopedGuard{std::move(tf)}; // C++17以降の記法  
        ASSERT_FALSE(is_called); // sgのデストラクタは呼ばれていない  
    }
```

```
    ASSERT_TRUE(is_called); // sgのデストラクタは呼ばれた
}
```

C++17からサポートされた「クラステンプレートのテンプレートパラメータ推論」が使えないC++14以前では、下記に示すように ScopedGuardのテンプレートラメータ型を指定しなければならない煩雑さがある。

```
// @@@ example/template/func_type_ut.cpp 148
{ // Fに関数型オブジェクトを与える
    auto is_called = false;
    auto tf      = TestFunctor{is_called};
    {
        auto sg = ScopedGuard<TestFunctor>{std::move(tf)}; // C++14以前の記法
        ASSERT_FALSE(is_called); // sgのデストラクタは呼ばれていない
    }
    ASSERT_TRUE(is_called); // sgのデストラクタは呼ばれた
}
```

これを回避するためには下記のような関数テンプレートを用意すればよい。

```
// @@@ example/template/func_type_ut.cpp 161
template <typename F>
ScopedGuard<F> MakeScopedGuard(F&& f) noexcept
{
    return ScopedGuard<F>(std::move(f));
}
```

下記に示した単体テストから明らかな通り、関数テンプレートの型推測の機能により、テンプレートパラメータを指定する必要がなくなる。

```
// @@@ example/template/func_type_ut.cpp 172
{ // Fに関数ポインタを与える
    is_caleded_in_static = false;
    {
        auto sg = MakeScopedGuard(&caleded_by_destructor);
        ASSERT_FALSE(is_caleded_in_static); // sgのデストラクタは呼ばれていない
    }
    ASSERT_TRUE(is_caleded_in_static); // sgのデストラクタは呼ばれた
}
{ // Fにラムダ式を与える
    auto is_called = false;
    {
        auto sg = MakeScopedGuard([&is_called]{} noexcept { is_called = true; });
        ASSERT_FALSE(is_called); // sgのデストラクタは呼ばれていない
    }
    ASSERT_TRUE(is_called); // sgのデストラクタは呼ばれた
}
{ // Fにstd::function型オブジェクトを与える
    auto is_called = false;
    {
        auto f  = std::function<void(void)>{[&is_called]{} noexcept { is_called = true; }};
        auto sg = MakeScopedGuard(std::move(f));
        ASSERT_FALSE(is_called); // sgのデストラクタは呼ばれていない
    }
    ASSERT_TRUE(is_called); // sgのデストラクタは呼ばれた
}
{ // Fに関数型オブジェクトを与える
    auto is_called = false;
    auto tf      = TestFunctor{is_called};
    {
        auto sg = MakeScopedGuard(std::ref(tf)); // std::refが必要
        ASSERT_FALSE(is_called); // sgのデストラクタは呼ばれていない
    }
    ASSERT_TRUE(is_called); // sgのデストラクタは呼ばれた
}
```

このような便利なテンプレートは、Nstdのようなライブラリで定義、宣言し、ソースコード全域からアクセスできるようにするとプロジェクトの開発効率が少しだけ高まる。

演習-関数型のテンプレートパラメータを持つクラステンプレート

注意点まとめ

本章では、テンプレートメタプログラミングのテクニックや注意点について解説したが、本章の情報量は多く、また他の章で行ったものもあるため以下にそれらをまとめる。

- name lookupには複雑なルールが適用されるため、非直感的なバインドが行われる場合がある。従って、テンプレートライブラリの開発には単体テストは必須である。
- 使用しているコンパイラがtwo_phase name lookupをサポートしているか否かに気を付ける。それがオプションである場合は、two phase name lookupを活性化させる。
- 関数型マクロはそれ以外に実装方法がない時のみに使用する（「関数型マクロ」参照）。
- 可変長引数を持つ関数の実装にはパラメータパックを使う。
- 処理速度や関数のリターンの型に影響する場合があるため、パラメータパックの処理の順番に気を付ける（「前から演算するパラメータパック」参照）。
- ADLを利用しない場合、テンプレートで使う識別子は名前空間名やthis->等で修飾する（「意図しないname lookupの防止」参照）。
- テンプレートのインターフェースではないが、実装の都合上ヘッダファイルに記述する定義は、“namespace Inner_”を使用し、非公開であることを明示する。また、“namespace Inner_”で宣言、定義されている宣言、定義は単体テストを除き、外部から参照しない（「is_void_sfinae_fの実装」参照）。
- ユニバーサルリファレンスの実際の型がlvalueリファレンスであるならば、constなlvalueリファレンスとして扱う（「実引数/仮引数」参照）。
- ユニバーサルリファレンス引数を他の関数に渡すのであれば、std::forwardを使う（「ユニバーサルリファレンス」、「ユニバーサルリフアレンスとstd::forward」参照）。
- 関数テンプレートとその特殊化はソースコード上なるべく近い位置で定義する（「two_phase name lookup」参照）。
- two_phase name lookupにより意図しない副作用が発生する可能性があるため、STLが特殊化を想定しているstd::hash等を除き、STLの拡張は行わない。
- ユーザが定義するテンプレートは適切に定義された名前空間内で定義する（「スコープの定義と原則」参照）。
- 型とその2項演算子オーバーロードは同じ名前空間で定義する（「two_phase name lookup」参照）。
- 関数テンプレートのオーバーロードと特殊化のname lookupの優先度に気を付ける。オーバーロードのベストマッチ選択後に特殊化された関数テンプレートがname lookupの対象になるため、下記コードが示すように直感に反する関数が選択される場合がある。

```
// @@@ example/template/etc_ut.cpp 7

template <typename T> constexpr int32_t f(T) noexcept { return 0; } // f-0
template <typename T> constexpr int32_t f(T*) noexcept { return 1; } // f-1
template <> constexpr int32_t f<int32_t*>(int32_t*) noexcept { return 2; } // f-2
// f-2はf-1の特殊化のように見えるが、T == int32_t*の場合のf-0の特殊化である。

// @@@ example/template/etc_ut.cpp 18

// 以下、f-0/f-1/f-2のテスト
auto c    = char{0};
auto i32 = 0;

// 以下はおそらく直感通り
static_assert(f(0) == 0); // f-0が呼ばれる
static_assert(f(&c) == 1); // f-1が呼ばれる
static_assert(f<int32_t*>(&i32) == 2); // f-2が呼ばれる

// 以下はおそらく直感に反する
static_assert(f(nullptr) == 0); // f-1ではなく、f-0が呼ばれる
static_assert(f(&i32) == 1); // f-2ではなく、f-1が呼ばれる
```

- ユニバーサルリファレンスを持つ関数テンプレートをオーバーロードしない。「ユニバーサルリファレンスとstd::forward」で述べたように、ユニバーサルリファレンスはオーバーロードするためのものではなく、lvalue、rvalue両方を受け取ることができる関数テンプレートを、オーバーロードを使わずに実現するための記法である。
- テンプレートに関数型オブジェクトを渡す場合、リファレンスの付け忘れに気を付ける（「関数型をテンプレートパラメータで使う」、「現象6-STLのバグ？」参照）。
- 意図しないテンプレートパラメータによるインスタンス化の防止や、コンパイルエラーを解読しやすくするために、適切に static_assert（「exists_begin/exists_endの実装」、「assertion」参照）を使う。
- ランタイム時の処理を削減する、static_assertを適切に用いる等の目的のために、関数テンプレートには適切にconstexprを付けて宣言する（「コンテナ用Nstd::operator<<の開発」、「constexpr関数」参照）。

ダイナミックメモリアロケーション

本章で扱うダイナミックメモリアロケーションとは、new/delete、malloc/freeによるメモリ確保/解放のことである。

malloc/freeは、

- ・最長処理時間を規定できない(リアルタイム性の欠如)
- ・メモリのフラグメントを起こす

等の問題(「[malloc/freeの問題点](#)」参照)を持っている。new/deleteは通常malloc/freeを使って実装されているため同じ問題を持っているが、これらが汎用OS上でアプリケーションで実際の不具合につながることはほとんどない。一方で、

- ・リアルタイムな応答が要求される
- ・メモリの使用制限が厳しい(ページングと2次記憶がない)

のような組み込みソフトでは、上記の2点は致命的な不具合につながる。

本章では、この問題を回避するための技法を紹介する。

この章の構成

[malloc/freeの問題点](#)

[グローバルnew/deleteのオーバーロード](#)

[固定長メモリプール](#)

[グローバルnew/deleteのオーバーロードの実装](#)

[プレースメントnew](#)

[デバッグ用イテレータ](#)

[クラスnew/deleteのオーバーロード](#)

[STLコンテナのアロケーター](#)

[STLコンテナ用アロケーター](#)

[可変長メモリプール](#)

[デバッグ用イテレータ](#)

[エクセプション処理機構の変更](#)

malloc/freeの問題点

UNIX系のOSでの典型的なmalloc/freeの実装例の一部を以下に示す(この実装は長いため、全体は巻末の[example/dynamic_memory_allocation/malloc_ut.cpp](#)に掲載する)。

```
// @@@ example/dynamic_memory_allocation/malloc_ut.cpp 19

namespace {

struct header_t {
    header_t* next;
    size_t n_nuits; // header_tが何個あるか
};

header_t* header{nullptr};
SpinLock spin_lock{};
constexpr size_t unit_size{sizeof(header_t)};

inline bool sprit(header_t* header, size_t n_nuits, header_t*& next) noexcept
{
    ...
}

inline void concat(header_t* front, header_t* rear) noexcept
{
    ...
}

header_t* set_back(void* mem) noexcept { return static_cast<header_t*>(mem) - 1; }

static_assert(sizeof(header_t) == alignof(std::max_align_t));
```

```

void* malloc_inner(size_t size) noexcept
{
    ...
}

// @@@ example/dynamic_memory_allocation/malloc_ut.cpp 105

void free(void* mem) noexcept
{
    header_t* mem_to_free = set_back(mem);

    mem_to_free->next = nullptr;

    auto lock = std::lock_guard{spin_lock};

    if (header == nullptr) {
        header = mem_to_free;
        return;
    }

    if (mem_to_free < header) {
        concat(mem_to_free, header);
        header = mem_to_free;
        return;
    }

    auto curr = header;
    for (; curr->next != nullptr; curr = curr->next) {
        if (mem_to_free < curr->next) { // 常に curr < mem_to_free
            concat(mem_to_free, curr->next);
            concat(curr, mem_to_free);
            return;
        }
    }

    concat(curr, mem_to_free);
}

void* malloc(size_t size) noexcept
{
    void* mem = malloc_inner(size);

    if (mem == nullptr) {
        auto const add_size = Roundup(unit_size, 1024 * 1024 + size); // 1MB追加

        header_t* add = static_cast<header_t*>(sbrk(add_size));
        add->n_nuits = add_size / unit_size;
        free(++add);
        mem = malloc_inner(size);
    }

    return mem;
}

```

上記で示したようにmalloc/freeで使用されるメモリはHeader_t型のheaderで管理され、このアクセスの競合はspin_lockによって回避される。headerが管理するメモリ用域からのメモリの切り出しはmalloc_innerによって行われるが、下のコメントの説明でも示す通り、headerで管理されたメモリは長さの上限が単純には決まらないリスト構造になるため、このリストをなぞるmalloc/freeにリアルタイム性の保証をすることは困難である。

アプリケーションが実行する最初のmallocから呼び出されるmalloc_innerは、headerがnullptrであるため必ずnullptrを返すことになる。

上記の抜粋である下記のコードによりmalloc_innerの戻りがnullptrであった場合、sbrkが呼び出される。

```

// @@@ example/dynamic_memory_allocation/malloc_ut.cpp 148

if (mem == nullptr) {
    auto const add_size = Roundup(unit_size, 1024 * 1024 + size); // 1MB追加

    header_t* add = static_cast<header_t*>(sbrk(add_size));
    add->n_nuits = add_size / unit_size;
    free(++add);
    mem = malloc_inner(size);
}

```

sbrkとはOSからメモリを新たに取得するための下記のようなシステムコールである。

```
// @@@ example/dynamic_memory_allocation/malloc_ut.cpp 13
extern "C" void* sbrk(ptrdiff_t __incr);
```

OSがアプリケーションに割り当てるための十分なメモリを持っていない場合、`sbrk`はページングによるメモリ確保のトリガーとなる。これはOSのファイルシステムの動作を含む処理であるため、やはりリアルタイム性の保証は困難である。

フリースタンディング環境では、`sbrk`のようなシステムコールは存在しないため、アプリケーションの未使用領域や静的に確保した領域を上記コードで示したようなリスト構造で管理し、`malloc`で使用することになる。このような環境では、`sbrk`によるリアルタイム性の阻害は発生しないものの、メモリ管理ためのリスト構造があるため、やはりリアルタイム性の保証は難しい。

次にもう一つの問題である「メモリのフラグメントを起こす」ことについて見て行く。

```
// @@@ example/dynamic_memory_allocation/malloc_ut.cpp 190
void* mem[1024];

for (auto& m : mem) { // 32バイト x 1024個のメモリ確保
    m = malloc(32);
}

// memを使用した何らかの処理
...

for (auto i = 0U; i < ArrayLength(mem); i += 2) { // 512個のメモリを解放
    free(mem[i]);
}
```

上記のような処理の後、解放されたメモリは、32バイト(メモリヘッダがあるため、実際はもう少し大きい)の断片が512個ある状態になるため、このサイズを超える新たな`malloc`の呼び出しには使えない。このため、ページングが行えないようなOS上のアプリケーションでは、メモリは十分にあるにもかかわらず`malloc`が失敗してしまうことが起こり得る。

また、上記`free`の実装例の抜粋である下記のコードからわかるように、このように断片化されたメモリは、そのアドレス順にソートされた単方向リストによって管理される。

```
// @@@ example/dynamic_memory_allocation/malloc_ut.cpp 120
if (mem_to_free < header) {
    concat(mem_to_free, header);
    header = mem_to_free;
    return;
}

auto curr = header;
for (; curr->next != nullptr; curr = curr->next) {
    if (mem_to_free < curr->next) { // 常に curr < mem_to_free
        concat(mem_to_free, curr->next);
        concat(curr, mem_to_free);
        return;
    }
}

concat(curr, mem_to_free);
```

この状態でさらにメモリ解放が行われた場合、`free`はこのリストを辿りメモリを最適な場所に戻す必要がある。戻したメモリがリスト前後のメモリと隣接していれば、それらは結合される。この処理は断片化への対策であるが、ページングの無いOS上のアプリケーションにとっては不十分であるばかりでなく、

- `free`の排他ロックする期間が長い
- `free`の処理が遅い

といったリアルタイム処理を阻害する別の問題も発生させる(繰り返しになるが、windows/linuxのような通常のOS上のアプリケーションでは、このような仕様が問題になることはほとんどない)。

グローバルnew/deleteのオーバーロード

すでに述べたように、組み込みソフトには`malloc/free`を使用した`new/delete`は使えない可能性が高い。そのような場合に備えC++11ではグローバルな`new/delete`のオーバーロードをサポートする。ここでは、そのような`new/delete`の実装例を示すが、その前に`new/delete`の内部実装用メモリ管理用ライブラリを実装する。

固定長メモリプール

malloc/freeにリアルタイム性がない原因は、

- リアルタイム性がないOSのシステムコールを使用している
- メモリを可変長で管理しているため処理が重いにもかかわらず、この処理中にグローバルロックを行う。

ためである。従って、この問題に対処するためのメモリ管理システムは、

- 初期に静的なメモリを確保
- メモリを固定長で管理

する必要がある。これを含めこの章で開発するメモリ管理システムをメモリプールと呼ぶことにする。

「グローバルnew/deleteのオーバーロードの実装」で示すように、このメモリプールは管理する固定長のメモリブロックのサイズごとに複数必要になる一方で、これらを統合的に扱う必要も出てくる。

そのため、固定長のメモリプールは、

- 複数個のメモリプールを統合的に扱うインターフェースクラスMPool
- MPoolを基底クラスとし、固定長メモリブロックを管理するクラステンプレートMPoolFixed

によって実装することにする。

まずは、MPoolを下記に示す（「ファイル位置を静的に保持したエクゼプションクラスの開発」参照）。

```
// @@@ example/dynamic_memory_allocation/mpool.h 12

class MPool {
public:
    explicit MPool(size_t max_size) : max_size_{max_size} {}

    void* Alloc(size_t size)
    {
        if (size > max_size_) {
            throw MAKE_EXCEPTION(MPoolBadAlloc, "MPF : memory size too big");
        }

        void* mem = alloc(size);

        if (mem == nullptr) {
            throw MAKE_EXCEPTION(MPoolBadAlloc, "MPF : out of memory");
        }

        return mem;
    }

    void* AllocNoExcept(size_t size) noexcept { return alloc(size); }

    void Free(void* area) noexcept { free(area); }

    size_t GetSize() const noexcept { return get_size(); } // メモリ最小単位
    size_t GetCount() const noexcept { return get_count(); } // メモリ最小単位が何個取れるか
    size_t GetCountMin() const noexcept { return get_count_min(); } // GetCount()の最小値
    bool IsValid(void const* area) const noexcept { return is_valid(area); }

protected:
    ~MPool() = default;

private:
    size_t const max_size_;

    virtual void* alloc(size_t size) noexcept = 0;
    virtual void free(void* area) noexcept = 0;
    virtual size_t get_size() const noexcept = 0;
    virtual size_t get_count() const noexcept = 0;
    virtual size_t get_count_min() const noexcept = 0;
    virtual bool is_valid(void const* area) const noexcept = 0;
};
```

次に、MPoolFixedを下記に示す。

```
// @@@ example/dynamic_memory_allocation/mpool_fixed.h 25

template <uint32_t MEM_SIZE, uint32_t MEM_COUNT>
class MPoolFixed final : public MPool {
public:
```

```

MPoolFixed() noexcept : MPool{mem_chunk_size_} {}

private:
    using chunk_t = Inner_::mem_chunk<MEM_SIZE>;
    static constexpr size_t mem_chunk_size_{sizeof(chunk_t)};

    size_t          mem_count_{MEM_COUNT};
    size_t          mem_count_min_{MEM_COUNT};
    chunk_t         mem_chunk_[MEM_COUNT]{};
    chunk_t*        mem_head_{setup_mem()};
    mutable SpinLock spin_lock_{};

    chunk_t* setup_mem() noexcept
    {
        for (auto i = 0U; i < MEM_COUNT - 1; ++i) {
            mem_chunk_[i].next = &mem_chunk_[i + 1];
        }

        mem_chunk_[MEM_COUNT - 1].next = nullptr;
    }

    return mem_chunk_;
}

virtual void* alloc(size_t size) noexcept override
{
    assert(size <= mem_chunk_size_);

    auto lock = std::lock_guard{spin_lock_};

    auto mem = mem_head_;

    if (mem != nullptr) {
        mem_head_     = mem_head_->next;
        mem_count_min_ = std::min(--mem_count_, mem_count_min_);
    }

    return mem;
}

virtual void free(void* mem) noexcept override
{
    assert(is_valid(mem));

    auto lock = std::lock_guard{spin_lock_};

    chunk_t* curr_head = static_cast<chunk_t*>(mem);
    curr_head->next   = mem_head_;
    mem_head_          = curr_head;

    mem_count_min_ = std::min(++mem_count_, mem_count_min_);
}

virtual size_t get_size() const noexcept override { return mem_chunk_size_; }
virtual size_t get_count() const noexcept override { return mem_count_; }
virtual size_t get_count_min() const noexcept override { return mem_count_min_; }

virtual bool is_valid(void const* mem) const noexcept override
{
    return (&mem_chunk_[0] <= mem) && (mem <= &mem_chunk_[MEM_COUNT - 1]);
}
};

```

上記コードからわかる通り、MPoolFixedは初期化直後、 サイズMEM_SIZEのメモリブロックをMEM_COUNT個、保持する。個々のメモリブロックは、下記のコードのalignas/alignofでアライメントされた領域となる。

```

// @@@ example/dynamic_memory_allocation/mpool_fixed.h 11

constexpr size_t MPoolFixed_MinSize{32};

namespace Inner_ {
template <uint32_t MEM_SIZE>
union mem_chunk {
    mem_chunk* next;

    // MPoolFixed_MinSizeの整数倍のエリアを、最大アラインメントが必要な基本型にアライン
    alignas(alignof(std::max_align_t)) uint8_t mem[Roundup(MPoolFixed_MinSize, MEM_SIZE)];
};

} // namespace Inner_

```

MPoolFixedに限らずメモリアロケータが返すメモリは、どのようなアライメントにも対応できなければならないため、このようにする必要がある。

MPoolFixed::alloc/MPoolFixed::freeを見ればわかる通り、malloc/freeの実装に比べ格段にシンプルであり、これによりリアルタイム性の保障は容易である。

なお、この実装ではmalloc/freeと同様に下記のSpinLockを使用したが、このロックは、ラウンドロビンでスケジューリングされるスレッドの競合を防ぐためのものであり、固定プライオリティでのスケジューリングが前提となるような組み込みソフトで使用した場合、デッドロックを引き起こす可能性がある。組み込みソフトでは、割り込みディセブル/イネーブルを使ってロックすることを推奨する。

```
// @@@ example/dynamic_memory_allocation/spin_lock.h 3

#include <atomic>

class SpinLock {
public:
    void lock() noexcept
    {
        while (state_.exchange(state::locked, std::memory_order_acquire) == state::locked) {
            ; // busy wait
        }
    }

    void unlock() noexcept { state_.store(state::unlocked, std::memory_order_release); }

private:
    enum class state { locked, unlocked };
    std::atomic<state> state_{state::unlocked};
};
```

MPoolFixedの単体テストは、下記のようになる。

```
// @@@ example/dynamic_memory_allocation/mpool_fixed_ut.cpp 10
Inner_::mem_chunk<5> mc5[3];
static_assert(32 == sizeof(mc5[0]));
static_assert(96 == sizeof(mc5));

auto mc33 = Inner_::mem_chunk<33>{};
static_assert(64 == sizeof(mc33));

// @@@ example/dynamic_memory_allocation/mpool_fixed_ut.cpp 106

auto mpf = MPoolFixed<33, 2>{};

ASSERT_EQ(64, mpf.GetSize());
ASSERT_EQ(2, mpf.GetCount());
ASSERT_EQ(2, mpf.GetCountMin());
ASSERT_FALSE(mpf.IsValid(&mpf)); // mpfの管理外のアドレス

auto m0 = mpf.Alloc(1);
ASSERT_TRUE(mpf.IsValid(m0)); // mpfの管理のアドレス
ASSERT_NE(nullptr, m0);
ASSERT_EQ(1, mpf.GetCount());
ASSERT_EQ(1, mpf.GetCountMin());

auto m1 = mpf.Alloc(1);
ASSERT_TRUE(mpf.IsValid(m1)); // mpfの管理のアドレス
ASSERT_NE(nullptr, m1);
ASSERT_EQ(0, mpf.GetCount());
ASSERT_EQ(0, mpf.GetCountMin());

// mpfが空の場合のテスト
ASSERT_THROW(mpf.Alloc(1), MPoolBadAlloc); // MPoolBadAlloc例外が発生するはず
auto m2 = mpf.AllocNoExcept(1);
ASSERT_EQ(nullptr, m2);
ASSERT_EQ(0, mpf.GetCount());

mpf.Free(m0);
ASSERT_EQ(1, mpf.GetCount());
ASSERT_EQ(0, mpf.GetCountMin());

mpf.Free(m1);
ASSERT_EQ(2, mpf.GetCount());
ASSERT_EQ(0, mpf.GetCountMin());

ASSERT_THROW(mpf.Alloc(65), MPoolBadAlloc); // MPoolBadAlloc例外が発生するはず
```

上記テストで使用したMPoolBadAllocは下記のように定義されたクラスであり(「[ファイル位置を静的に保持したエクセプションクラスの開発](#)」参照)、

```
// @@@ example/h/nstd_exception.h 10

/// @class Exception
/// @brief StaticString<>を使ったエクセプションクラス
/// 下記のMAKE_EXCEPTIONを使い生成
/// @tparam E std::exceptionから派生したエクセプションクラス
/// @tparam N StaticString<N>
template <class E, size_t N>
class Exception : public E {
public:
    static_assert(std::is_base_of_v<std::exception, E>);

    Exception(StaticString<N> const& what_str) noexcept : what_str_{what_str} {}
    char const* what() const noexcept override { return what_str_.String(); }

private:
    StaticString<N> const what_str_;
};

#define MAKE_EXCEPTION(E__, msg__) Nstd::MakeException<E__, __LINE__>(__FILE__, msg__)

// @@@ example/dynamic_memory_allocation/mpool.h 7

class MPoolBadAlloc : public std::bad_alloc { // Nstd::Exceptionの基底クラス
};
```

MPoolから派生したクラスが、

- メモリブロックを保持していない状態でのMPool::alloc(size, true)の呼び出し
- MEM_SIZEを超えたsizeでのMPool::alloc(size, true)の呼び出し

のような処理の継続ができない場合に用いるエクセプション用クラスである。

グローバルnew/deleteのオーバーロードの実装

[固定長メモリプール](#)を使用したoperator newのオーバーロードの実装例を以下に示す。

```
// @@@ example/dynamic_memory_allocation/global_new_delete.cpp 31

namespace {
MPool* mpool_table[32];

// mainの前に呼ばれるため、mpool_tableを初期化するには下記のような方法が必要
bool is_setup{false};

void setup() noexcept
{
    is_setup = true;

    mpool_table[0] = gen_mpool<1, 128>(); // 32
    mpool_table[1] = gen_mpool<2, 128>(); // 64
    mpool_table[2] = gen_mpool<3, 128>(); // 96

    ...
    mpool_table[29] = gen_mpool<30, 128>(); // 960
    mpool_table[30] = gen_mpool<31, 128>(); // 992
    mpool_table[31] = gen_mpool<32, 128>(); // 1024
}

size_t size2index(size_t v) noexcept
{
    return (((v + (min_unit - 1)) & ~(min_unit - 1)) / min_unit) - 1;
}
// namespace

[[nodiscard]] void* operator new(std::size_t size)
{
    if (!is_setup) {
        setup();
    }

    for (auto i = size2index(size); i < ArrayLength(mpool_table); ++i) {
        void* mem = mpool_table[i]->AllocNoExcept(size);
        if (mem != nullptr) {

```

```

        return mem;
    }

    throw std::bad_alloc{};

    static char fake[0];

    return fake;
}

```

上記で定義されたoperator newは、

- 32の整数倍のサイズを持つ32個のメモリプールを持つ
- 各メモリープールは128個のメモリブロックを持つ
- メモリブロックの最大長は1024バイト

のような仕様を持つため、実際に使う場合は、メモリのサイズや個数の調整が必要だろうが、後で詳しく見るようリアルタイム性の阻害となるようなコードはないため、リアルタイム性が必要なソフトウェアでも使用可能である。

静的オブジェクトを含まないアプリケーションでは、上記のコードのsetupで行っているmpool_tableの初期化は一様初期化で行った方が良いが、例で用いたアプリケーションにはnewを行う静的オブジェクトが存在するため(google testは静的オブジェクトを利用する)、setupで行っているような方法以外では、最初のoperator newの呼び出しそり前にmpool_tableの初期化をすることはできない。

mpool_tableはMPoolポインタを保持するが、そのポインタが指すオブジェクトの実態は、gen_mpool<>が生成したMPoolFixed<>オブジェクトである。gen_mpool<>については、「[プレースメントnew](#)」で説明する。

size2indexは、要求されたサイズから、それに対応するMPoolポインタを保持するmpool_tableのインデックスを導出する関数である。

この実装では対応するMPoolが空であった場合、それよりも大きいメモリブロックを持つMPoolからメモリを返す仕様としたが、その時点でアサーションフェールさせ(つまり、対応するMPoolが空である状態でのAllocの呼び出しをバグとして扱う)、MEM_COUNTの値を見直した方が、より少ないメモリで動作する組み込みソフトを作りやすいだろう。

operator deleteについては、下記の2種類が必要となる。

```

// @@@ example/dynamic_memory_allocation/global_new_delete.cpp 109

void operator delete(void* mem) noexcept
{
    for (MPool* mp : mpool_table) {
        if (mp->IsValid(mem)) {
            mp->Free(mem);
            return;
        }
    }
    assert(false);
}

void operator delete(void* mem, std::size_t size) noexcept
{
    for (auto i = size2index(size); i < ArrayLength(mpool_table); ++i) {
        if (mpool_table[i]->IsValid(mem)) {
            mpool_table[i]->Free(mem);
            return;
        }
    }
    assert(false);
}

```

operator delete(void* mem, std::size_t size)は、完全型のオブジェクトのメモリ解放に使用され、operator delete(void* mem)は、それ以外のメモリ解放に使用される。

コードから明らかな通り、size付きのoperator deleteの方がループの回転数が少なくなるため、高速に動作するが、malloc/freeの実装(「[malloc/freeの問題点](#)」参照)で使用したHeader_tを導入することでこの実行コストはほとんど排除できる。そのトレードオフとしてメモリコストが増えるため、ここでは例示した仕様にした。

プレースメントnew

「[グローバルnew/deleteのオーバーロードの実装](#)」で使用したgen_mpool<>は、下記のように定義されている。

```

// @@@ example/dynamic_memory_allocation/global_new_delete.cpp 8

namespace {

```

```

constexpr size_t min_unit{MPoolFixed_MinSize};

template <uint32_t N_UNITS, uint32_t MEM_COUNT>
[[nodiscard]] MPool* gen_mpool() noexcept
{
    using mp_t = MPoolFixed<min_unit * N_UNITS, MEM_COUNT>;

    static union {
        std::max_align_t max_align;
        uint8_t mem[sizeof(mp_t)];
    } mem;

    static_assert(static_cast<void*>(&mem.max_align) == static_cast<void*>(mem.mem));
    static_assert(sizeof(mem) >= sizeof(mp_t));

    return new (mem.mem) mp_t; // プレースメントnew
}
} // namespace

```

この関数テンプレートは、MPoolFixed<>オブジェクトを生成し、それをMPool型のポインタとして返す。MPoolFixedの生成は、上記で示したようにプレースメントnewを使用して行っている。

gen_mpool<>内でMPoolFixedのstaticなインスタンスを定義した方がシンプルに実装できるが、その場合、main()終了後、そのインスタンスは解放され(デストラクタが呼び出され)、その後、他の静的オブジェクトの解放が行われると、その延長でoperator deleteが呼び出され、ライフタイム終了後のMPoolFixedのstaticなインスタンスが使われてしまう。

現在のMPoolFixedの実装ではこの操作で不具合は発生しないが、解放済のオブジェクトを操作することは避けるべきであるため、MPoolFixedの生成にプレースメントnewを用いている。

プレースメントnewで生成したオブジェクトをdeleteすることはできず、デストラクタはユーザが明示的に呼び出さない限り、呼び出されない。ここでは、プレースメントnewのこの特性を利用したが、逆に、この特性があるため、ここで実装のような特殊な事情がある場合を除き、プレースメントnewを使うべきではない(デストラクタの明示的な呼び出しを忘れるリソースリークしてしまう)。

デバッグ用イテレータ

この章で例示したグローバルnew/deleteは、すでに述べたように適切なメモリの量を調整する必要がある。そのためには、これを使用するアプリケーションをある程度動作させた後、グローバルnew/deleteのメモリの消費量を計測しなければならない。

下記のコードは、そのためのインターフェースを提供する。

```

// @@@ example/dynamic_memory_allocation/global_new_delete.h 4

class GlobalNewDeleteMonitor {
public:
    MPool const* const* cbegin() const noexcept;
    MPool const* const* cend() const noexcept;
    MPool const* const* begin() const noexcept;
    MPool const* const* end() const noexcept;
};

// @@@ example/dynamic_memory_allocation/global_new_delete.cpp 135

MPool const* const* GlobalNewDeleteMonitor::begin() const noexcept { return &mpool_table[0]; }
MPool const* const* GlobalNewDeleteMonitor::end() const noexcept
{
    return &mpool_table[ArrayLength(mpool_table)];
}

MPool const* const* GlobalNewDeleteMonitor::cbegin() const noexcept { return begin(); }
MPool const* const* GlobalNewDeleteMonitor::cend() const noexcept { return end(); }

```

このインターフェースを下記のように使用することで、

```

// @@@ example/dynamic_memory_allocation/global_new_delete_ut.cpp 124

auto gm = GlobalNewDeleteMonitor{};

std::cout << " size current min" << std::endl;
std::cout << " -----" << std::endl;

for (MPool const* mp : gm) {
    std::cout << std::setw(6) << mp->GetSize() << std::setw(8) << mp->GetCount() << std::setw(6)
        << mp->GetCountMin() << std::endl;
}

```

下記のようにメモリの現在の状態や使用履歴を見ることができる。

```
size current min
-----
 32      90    0
 64      78   74
 96     127  125

...
992     128  128
1024    128    0
```

実際の組み込みソフトの開発では、デバッグ用入出力機能からこのようなコードを実行できるようにすることで、グローバルnew/deleteが使用するそれぞれのMPoolFixedインスタンスのメモリの調整ができるだろう。

クラスnew/deleteのオーバーロード

「[グローバルnew/deleteのオーバーロードの実装](#)」で示したコードのロックを、「割り込みディセブル/イネーブル」に置き換えることで、リアルタイム性を保障することができるが、この機構はある程度多くのメモリを必要とするため、極めてメモリ制限の厳しいシステムでは使用が困難である場合もあるだろう。

そのような場合、非スタック上のオブジェクト生成には、

- 限定的なクラスのみ、newによる動的な方法を用いる
- その他のクラスに対しては、[Singleton](#)や[Named Constructor](#)と同様な静的な方法を用いる

とし、グローバルnewを使用しないことが、より良いメモリ使用方法となり得る。

グローバルnewを使わずに動的にオブジェクトを生成するためには、

- プレースメントnewを使う
- クラス毎にnew/deleteをオーバーロードする

という2つの選択肢が考えられるが、すでに述べた理由によりプレースメントnewの使用は避けるべきである。従って、その方法はクラス毎のnew/deleteのオーバーロードになる。

メモリ管理に「[固定長メモリプール](#)」で示したMPoolFixedを利用した実装例を以下に示す。

```
// @@@ example/dynamic_memory_allocation/class_new_delete_ut.cpp 14

struct A {
    A() noexcept : name0{"A"} {}
    char const* name0;

    [[nodiscard]] static void* operator new(size_t size);
    static void          operator delete(void* mem) noexcept;
    static void          operator delete(void* mem, std::size_t size) noexcept;

    [[nodiscard]] static void* operator new[](size_t size)           = delete;
    static void            operator delete[](void* mem) noexcept   = delete;
    static void            operator delete[](void* mem, std::size_t size) noexcept = delete;
};

MPoolFixed<sizeof(A), 3> mpf_A;

void* A::operator new(size_t size) { return mpf_A.Alloc(size); }
void A::operator delete(void* mem) noexcept { mpf_A.Free(mem); }
void A::operator delete(void* mem, std::size_t) noexcept { mpf_A.Free(mem); }
```

以下の単体テストが示す通り、静的に定義したMPoolFixedインスタンスがオーバーロードしたnew/deleteから使われていることがわかる（従ってグローバルnew/deleteは使われていないこともわかる）。

```
// @@@ example/dynamic_memory_allocation/class_new_delete_ut.cpp 43

ASSERT_EQ(3, mpf_A.GetCount());

{
    auto a = std::make_unique<A>();
    ASSERT_STREQ("A", a->name0);
    ASSERT_EQ(2, mpf_A.GetCount());
}

ASSERT_EQ(3, mpf_A.GetCount());
```

```

{
    auto a = std::make_unique<A>();
    ASSERT_STREQ("A", a->name0);
    ASSERT_EQ(2, mpf_A.GetCount());

    auto b = std::make_unique<A>();
    ASSERT_STREQ("A", b->name0);
    ASSERT_EQ(1, mpf_A.GetCount());

    auto c = std::make_unique<A>();
    ASSERT_STREQ("A", c->name0);
    ASSERT_EQ(0, mpf_A.GetCount());

    ASSERT_THROW(std::make_unique<A>(), MPoolBadAlloc);
}
ASSERT_EQ(3, mpf_A.GetCount());

```

しかし、この方法ではnewのオーバーロードを行うクラス毎に、

```

// @@@ example/dynamic_memory_allocation/class_new_delete_ut.cpp 20

[[nodiscard]] static void* operator new(size_t size);
static void           operator delete(void* mem) noexcept;
static void           operator delete(void* mem, std::size_t size) noexcept;

[[nodiscard]] static void* operator new[](size_t size)          = delete;
static void            operator delete[](void* mem) noexcept = delete;
static void            operator delete[](void* mem, std::size_t size) noexcept = delete;

```

を記述しなければならず、コードクローンの温床となってしまう。これを避けるためには、[CRTP\(curiously recurring template pattern\)](#) を利用した下記のようなクラステンプレートを導入すれば良い。

```

// @@@ example/dynamic_memory_allocation/op_new.h 5

template <typename T>
class OpNew {
public:
    [[nodiscard]] static void* operator new(size_t size) { return mpool_.Alloc(size); }
    static void           operator delete(void* mem) noexcept { mpool_.Free(mem); }
    static void operator delete(void* mem, std::size_t) noexcept { mpool_.Free(mem); }

    [[nodiscard]] static void* operator new[](size_t size)          = delete;
    static void            operator delete[](void* mem) noexcept = delete;
    static void            operator delete[](void* mem, std::size_t size) noexcept = delete;

private:
    static MPool& mpool_;
};

```

このOpNewを使用した「new/deleteのオーバーロードを持つ基底クラスとその一連の派生クラス」の実装例を以下に示す。

```

// @@@ example/dynamic_memory_allocation/class_new_delete_ut.cpp 77

struct A : public OpNew<A> {
    A() noexcept : name0{"A"} {}
    char const* name0;
};

struct B : A {
    B() noexcept : name1{"B"} {}
    char const* name1;
};

struct C : A {
    C() noexcept : name1{"C"} {}
    char const* name1;
};

struct D : C {
    D() noexcept : name2{"D"} {}
    char const* name2;
};

MPoolFixed<MaxSizeof<A, B, C, D>(), 10> mpf_ABCD;

template <>
MPool& OpNew<A>::mpool_ = mpf_ABCD;

```

OpNewをクラステンプレートとし、内部で利用しないテンプレートパラメータを宣言した理由は、別のクラスからはOpNewの別インスタンスを使用できるようにするためである。

この方法は、コードが若干複雑にすることを除けば、「[グローバルnew/deleteのオーバーロード](#)」に比べ、優れているように見えてしまうかもしれないが、下記のように、さらに派生クラスを定義してしまうとnewが失敗してしまうことがあるので注意が必要である。

```
// @@@ example/dynamic_memory_allocation/class_new_delete_ut.cpp 135

struct Large : A {
    uint8_t buff[1024]; // mpf_ABCDのメモリブロックのサイズを超える
};

TEST(NewDelete_Opt, class_new_delete_fixed_derived_large)
{
    ASSERT_EQ(10, mpf_ABCD.GetCount());
    ASSERT_THROW(auto large = std::make_unique<Large>(), MPoolBadAlloc); // サイズが大きすぎる
}
```

なお、下記のようなクラスをnew/deleteをオーバーロードしないすべてのクラスの基底クラスとしてすることで、偶発的にグローバルnewを使ってしまわないようにすることもできる。

```
// @@@ example/dynamic_memory_allocation/op_new_deleted.h 3

class OpNewDeleted {
    static void* operator new(size_t size) = delete;
    static void operator delete(void* mem) noexcept = delete;
    static void operator delete(void* mem, std::size_t size) noexcept = delete;
};

// @@@ example/dynamic_memory_allocation/class_new_delete_ut.cpp 150

class DeletedNew : OpNewDeleted { // プライベート継承
};

class DelivedDeletedNew : DeletedNew { // プライベート継承
};

// DeletedNew* ptr0 { new DeletedNew }; // OpNewDeletedの効果でコンパイルエラー
// DelivedDeletedNew* ptr1 { new DelivedDeletedNew }; // 同上
```

この記述方法は、コードインスペクションの省力化にも繋がるため、OpNewを使うプロジェクトには導入するべきだろう。

STLコンテナのアロケーター

ここまで前提として来たような組み込みソフトにおいても、その大部分のコードにリアルタイム性は不要であり、このような部分のコードにSTLコンテナが使用できれば、

- 開発効率が向上する
- 開発コード量が少なくなる

等のポジティブな影響を期待できることは多い。STLコンテナはこういった状況に備えて、ユーザ定義のアロケータを使用できるように定義されている。ここでは、アロケータの定義例や、その使い方を示す。

STLコンテナ用アロケータ

アロケータの定義例を以下に示す。

```
// @@@ example/dynamic_memory_allocation/mpool_allocator.h 7

template <typename T>
class MPoolBasedAllocator {
public:
    using pointer = T*;
    using const_pointer = T const*;
    using value_type = T;
    using propagate_on_container_move_assignment = std::true_type;
    using is_always_equal = std::true_type;
    using size_type = size_t;
    using difference_type = size_t;

    template <class U>
    struct rebind {
        using other = MPoolBasedAllocator<U>;
    };
}
```

```

T*    allocate(size_type count) { return static_cast<pointer>(mpool_.Alloc(count * sizeof(T))); }
void deallocate(T* mem, size_type) noexcept { mpool_.Free(mem); }

private:
    static MPool& mpool_;
};

template <class T> // T型のMPoolBasedAllocatorはシステムに唯一
bool operator==(MPoolBasedAllocator<T> const&, MPoolBasedAllocator<T> const&) noexcept
{
    return true;
}

template <class T, class U>
bool operator==(MPoolBasedAllocator<T> const&, MPoolBasedAllocator<U> const&) noexcept
{
    return false;
}

template <class T, class U>
bool operator!=(MPoolBasedAllocator<T> const&, MPoolBasedAllocator<U> const& rhs) noexcept
{
    return !(lhs == rhs);
}

```

アロケータのパブリックなメンバやoperator==、operator!=は、STLに従い定義している ([STL allocator参照](#))。

上記コードからわかるようにメモリの実際のアロケーションには、これまでと同様にMPoolから派生したクラスを使用するが、リアルタイム性は不要であるためメモリ効率が悪いMPoolFixedは使わない。代わりに、可変長メモリを扱うためメモリ効率がよいMPoolVariable (「[可変長メモリプール](#)」参照)を使う。

可変長メモリプール

可変長メモリプールを生成するMPoolVariableの実装は下記のようになる (全体は巻末の[example/dynamic_memory_allocation/mpool_variable.h](#)に掲載する)。

```

// @@@ example/dynamic_memory_allocation/mpool_variable.h 59

template <uint32_t MEM_SIZE>
class MPoolVariable final : public MPool {
public:
    MPoolVariable() noexcept : MPool{MEM_SIZE}
    {
        header_>next     = nullptr;
        header_>n_nuits = sizeof(buff_) / Inner_::unit_size;
    }

    // 中略
    ...

private:
    using header_t = Inner_::header_t;

    Inner_::buffer_t<MEM_SIZE> buff_{};
    header_t*          header_{reinterpret_cast<header_t*>(buff_.buffer)};
    mutable SpinLock    spin_lock_{};
    size_t              unit_count_{sizeof(buff_) / Inner_::unit_size};
    size_t              unit_count_min_{sizeof(buff_) / Inner_::unit_size};

    virtual void* alloc(size_t size) noexcept override
    {
        ...
    }

    virtual void free(void* mem) noexcept override
    {
        ...
    }

    virtual size_t get_size() const noexcept override { return 1; }
    virtual size_t get_count() const noexcept override { return unit_count_ * Inner_::unit_size; }
    virtual size_t get_count_min() const noexcept override
    {
        return unit_count_min_ * Inner_::unit_size;
    }

    virtual bool is_valid(void const* mem) const noexcept override

```

```

    {
        return (&buff_ < mem) && (mem < &buff_.buffer[ArrayLength(buff_.buffer)]);
    }
};

```

下記のようにMPoolVariable、 MPoolBasedAllocatorを使うことでnew char[]に対応するアロケータが定義できる。

```

// @@@ example/dynamic_memory_allocation/mpool_allocator_ut.cpp 11

namespace {
MPoolVariable<1024 * 64> mpv_allocator;
}

template <>
MPool& MPoolBasedAllocator<char>::mpool_ = mpv_allocator;

```

下記の単体テストは、このアロケータを使うstd::stringオブジェクトの宣言方法と、その振る舞いを示している。

```

// @@@ example/dynamic_memory_allocation/mpool_allocator_ut.cpp 24

auto rest = mpv_allocator.GetCount();
auto str = std::basic_string<char, std::char_traits<char>, MPoolBasedAllocator<char>>{"hehe"};

ASSERT_TRUE(mpv_allocator.IsValid(str.c_str())); // mpv_allocatorを使用してメモリ確保
ASSERT_GT(rest, mpv_allocator.GetCount()); // mpv_allocatorのメモリが減っていることの確認

```

この長い宣言は、下記のようにして簡潔に記述できるようになる。

```

// @@@ example/dynamic_memory_allocation/mpool_allocator_ut.cpp 34

using mpv_string = std::basic_string<char, std::char_traits<char>, MPoolBasedAllocator<char>>;

```

下記のように宣言、定義することで、

```

// @@@ example/dynamic_memory_allocation/mpool_allocator_ut.cpp 50

template <>
MPool& MPoolBasedAllocator<int>::mpool_ = mpv_allocator;

using mpv_vector_int = std::vector<int, MPoolBasedAllocator<int>>;

```

下記の単体テストが示す通り、std::vector<int>にこのアロケータを使わせることもできる。

```

// @@@ example/dynamic_memory_allocation/mpool_allocator_ut.cpp 61

auto rest = mpv_allocator.GetCount();
auto ints = mpv_vector_int{1, 2, 3};

ASSERT_TRUE(mpv_allocator.IsValid(&ints[0])); // mpv_allocatorのメモリであることの確認
ASSERT_GT(rest, mpv_allocator.GetCount()); // mpv_allocatorのメモリが減っていることの確認

```

これまでの手法を組み合わせ下記のようにして、std::stringと同等のオブジェクトを保持するstd::vectorを宣言することもできる。

```

// @@@ example/dynamic_memory_allocation/mpool_allocator_ut.cpp 72

using mpv_vector_str = std::vector<mpv_string, MPoolBasedAllocator<mpv_string>>;

template <>
MPool& MPoolBasedAllocator<mpv_string>::mpool_ = mpv_allocator;

// @@@ example/dynamic_memory_allocation/mpool_allocator_ut.cpp 94

auto strs = mpv_vector_str{"1", "2", "3"};

ASSERT_GT(rest, mpv_allocator.GetCount());

for (auto const& s : strs) {
    ASSERT_TRUE(mpv_allocator.IsValid(&s)); // mpv_allocatorのメモリであることの確認
    ASSERT_TRUE(mpv_allocator.IsValid(s.c_str())); // mpv_allocatorのメモリであることの確認
}

```

しかし、下記に示すように、これまでの定義、宣言のみではmpv_stringのnewにこのアロケータを使わせることはできない。

```

// @@@ example/dynamic_memory_allocation/mpool_allocator_ut.cpp 121

auto rest = mpv_allocator.GetCount();

auto str0 = std::make_unique<mpv_string>(); // グローバルnewが使われる

// mpv_stringのnewにはmpv_allocatorは使われない

```

```
ASSERT_FALSE(mpv_allocator.IsValid(str0.get()));
ASSERT_EQ(rest, mpv_allocator.GetCount());
```

そうするためには、さらに下記のような定義が必要になる。

```
// @@@ example/dynamic_memory_allocation/mpool_allocator_ut.cpp 108

struct mpv_string_op_new : OpNew<mpv_string_op_new>, mpv_string {
    using mpv_string::basic_string;
};

template <>
MPool& OpNew<mpv_string_op_new>::mpool_ = mpv_allocator;
```

このようにすることで、下記に示すように期待した動きになる。

```
// @@@ example/dynamic_memory_allocation/mpool_allocator_ut.cpp 133

rest = mpv_allocator.GetCount();

auto str1 = std::make_unique<mpv_string_op_new>();

// mpv_string_op_newのnewにmpv_allocatorが使われる
ASSERT_TRUE(mpv_allocator.IsValid(str1.get()));
ASSERT_GT(rest, mpv_allocator.GetCount());
```

ただし、`std::make_shared`を使用した場合、この関数のメモリアロケーションの最適化により、下記に示すように期待した結果にならないため、注意が必要である。

```
// @@@ example/dynamic_memory_allocation/mpool_allocator_ut.cpp 145

rest = mpv_allocator.GetCount();

auto str2 = std::make_shared<mpv_string_op_new>();

// mpv_string_op_newのnewにmpv_allocatorが使われない!!!
ASSERT_FALSE(mpv_allocator.IsValid(str2.get()));
ASSERT_EQ(rest, mpv_allocator.GetCount());
```

`new`をオーバーロードしたクラスを`std::shared_ptr`で管理する場合、下記のようにしなければならない。

```
// @@@ example/dynamic_memory_allocation/mpool_allocator_ut.cpp 157

rest = mpv_allocator.GetCount();

auto str3 = std::shared_ptr<mpv_string_op_new>{new mpv_string_op_new};

// mpv_string_op_newのnewにmpv_allocatorが使われる
ASSERT_TRUE(mpv_allocator.IsValid(str3.get()));
ASSERT_GT(rest, mpv_allocator.GetCount());
```

デバッグ用イテレータ

可変長メモリプールを使用すると、メモリのフラグメントによりアロケーションが失敗することがあり得る。このような事態が発生している可能性がある場合、アロケータが保持しているメモリの状態を表示させることができるのがデバッグの第一歩となる。

下記のコードは、そのためのインターフェースを提供する。

```
// @@@ example/dynamic_memory_allocation/mpool_variable.h 59

template <uint32_t MEM_SIZE>
class MPoolVariable final : public MPool {
public:

    // 中略
    ...

    class const_iterator {
public:
    explicit const_iterator(Inner_::header_t const* header) noexcept : header_{header} {}
    const_iterator(const_iterator const&) = default;
    const_iterator(const_iterator&&) = default;

    const_iterator& operator++() noexcept // 前置++のみ実装
    {
        assert(header_ != nullptr);
        header_ = header_->next;
    }
};
```

```

        return *this;
    }

    Inner_::header_t const* operator*() noexcept { return header_; }
    bool operator==(const_iterator const& rhs) noexcept { return header_ == rhs.header_; }
    bool operator!=(const_iterator const& rhs) noexcept { return !(*this == rhs); }

private:
    Inner_::header_t const* header_;
};

const_iterator begin() const noexcept { return const_iterator{header_}; }
const_iterator end() const noexcept { return const_iterator{nullptr}; }
const_iterator cbegin() const noexcept { return const_iterator{header_}; }
const_iterator cend() const noexcept { return const_iterator{nullptr}; }

// 中略
...
};

```

このインターフェースを下記のように使用することで、

```

// @@@ example/dynamic_memory_allocation/mpool_allocator_ut.cpp 213

for (auto mem : mpv_allocator) {
    std::cout << std::setw(16) << mem->next << ":" << mem->n_nuits << std::endl;
}

```

下記のようにmpv_allocator.header_が保持するメモリの現在の状態を見ることができる(これによるとmpv_allocatorが保持するメモリの先頭付近では多少フラグメントを起こしているが、最後に大きなメモリブロックがあるため、全体としては問題ないレベルである)。

```

0x7f073afe59d0:3
0x7f073afe5a60:3
0x7f073afe5ac0:3
0x7f073afe5b70:3
0x7f073afe5c50:11
0x7f073afe5cb0:3
0x7f073afe5e50:13
0:4018

```

「[グローバルnew/deleteのオーバーロードの実装](#)」でも述べたように、デバッグ用入出力機能からこのような出力を得られるようにしておかべきである。

エクセプション処理機構の変更

多くのコンパイラのエクセプション処理機構にはnew/deleteやmalloc/freeが使われているため、リアルタイム性が必要な個所でエクセプション処理を行ってはならない。そういうたった規制でプログラミングを行っていると、リアルタイム性が不要な処理であるため使用しているSTLコンテナにすら、既存のエクセプション処理機構を使わせたく無くなるものである。

コンパイラにg++やclang++を使っている場合、下記関数を置き換えることでそういった要望を叶えることができる。

関数	機能
__cxa_allocate_exception(size_t thrown_size)	エクセプション処理用のメモリ確保
__cxa_free_exception(void* thrown_exception)	上記で確保したメモリの解放

オープンソースである[static exception](#)を使うことで、上記2関数を置き換えることもできるが、この実装が複雑すぎると思うのであれば、下記に示すような、これまで使用したMPoolFixedによる単純な実装を使うこともできる。

```

// @@@ example/dynamic_memory_allocation/exception_allocator_ut.cpp 12

// https://github.com/hjl-tools/gcc/blob/master/libstdc%2B%2B-v3/libsupc%2B%2B/unwind-cxx.h
// の抜粋
namespace __cxxabiv1 {
struct __cxa_exception {
    ...
};
} // namespace __cxxabiv1

namespace {

constexpr size_t          offset{sizeof(__cxxabiv1::__cxa_exception)};
MPoolFixed<offset + 128, 50> mpf_exception;
} // namespace

```

```

extern "C" {

void* __cxa_allocate_exception(size_t thrown_size)
{
    auto alloc_size = thrown_size + offset; // メモリの実際の必要量はthrown_size+offset
    auto mem        = mpf_exception.AllocNoExcept(alloc_size);

    assert(mem != nullptr);

    memset(mem, 0, alloc_size);
    auto* ret = static_cast<uint8_t*>(mem);

    ret += offset;

    return ret;
}

void __cxa_free_exception(void* thrown_exception)
{
    auto* ret = static_cast<uint8_t*>(thrown_exception);

    ret -= offset;
    mpf_exception.Free(ret);
}

```

以下に単体テストを示す。

```

// @@@ example/dynamic_memory_allocation/exception_allocator_ut.cpp 100

auto count           = mpf_exception.GetCount();
auto exception_occured = false;

try {
    throw std::exception{};
}
catch (std::exception const& e) {
    ASSERT_EQ(count - 1, mpf_exception.GetCount()); // 1個消費
    exception_occured = true;
}

ASSERT_TRUE(exception_occured);
ASSERT_EQ(count, mpf_exception.GetCount()); // 1個解放

```

すでに述べたが、残念なことに、この方法はC++の標準外であるため、これを適用できるコンパイラは限られている。しかし、多くのコンパイラはこれと同様の拡張方法を備えているため、安易にエクセプションやSTLコンテナを使用禁止することなく、安全に使用する方法を探るべきだろう。

デバッグ

ソフトウェア開発にバグは付き物であり、プログラマの工数の多くはそのバグの修正に費やされる。したがってデバッグの効率化はソフト開発効率改善の大きなポーションを占める。このドキュメントの提言の多くはそのためのものであるが、色々な文脈に分散しており、またそのときの論旨とは違ったため解説が不十分であったりする。

この問題を幾分緩和するためにこの章を設ける。

バグには以下のような様々な発生パターンがある。

- ある操作で100%発生するバグ
- ある操作の繰り返しによって発生するバグ
 - メモリやファイルハンドルのリークのように少しづつ劣化していくようなバグ
- ランダムに発生するように見えるバグ
 - マルチスレッドの競合問題のような確率論的にしか発生しないバグ
 - メモリ二重解放によるクラッシュのような発生パターンの特定が難しいバグ
 - デッドロック

当然ながら発生パターンの違いはデバッグ手法の有効性に大きく影響する。

共有リポジトリ内のソースコードにバグが多ければ、プログラマは各人の開発に集中できず、それ自体が新たなバグの発生源となる。また、共有リポジトリ内のソースコードのデバッグを行った為に、そのプログラマの新規機能開発に充てられる工数は不足し、これによるテスト不足は共有リポジトリ内のソースコードの新たなバグにつながる。と考えれば、共有リポジトリ内のソースコードをクリーンに保つことも、デバッグの効率化への重要な戦略であるといえる。

ここまで議論からわかるように、デバッグの効率化とは、単なるデバッグ手法にとどまらず、共有リポジトリ内のソースコードをクリーンに保つようなプロセス的な手法も包含する広範囲な内容となる。以下に、そのための様々なアイテムを説明する。

この章の構成

[デバッガの使用](#)

[gitの使用](#)

[TATの短縮](#)

[ビルド時間の短縮](#)

[依存関係の整理](#)

[自動単体テストの実施](#)

[自動統合テストの実施](#)

[事後的デバッグ](#)

[ログの取得](#)

[モニター](#)

[潜在的バグの削減](#)

[難しいバグの対処](#)

[助成の依頼](#)

[現象1 - 不思議なハングアップ](#)

[現象2 - グローバル変数の破壊](#)

[現象3 - プロセスのスローダウン](#)

[現象4 - バックトレースが見れない](#)

[現象5 - newしたオブジェクトの破壊](#)

[現象6 - STLのバグ?](#)

[現象7 - 解放後のrvalueへのアクセス](#)

[まとめ](#)

デバッガの使用

言うまでもなく特定操作で100%発生するバグのデバッグには、デバッガの使用が適している。windowsアプリケーションの開発では、Visual Studioが素晴らしいデバッガを提供しているため、それを使い、ステップ実行すればまず間違いなく短時間でバグを特定できる。linux系のアプリケーション開発においてはgdb/lldbで同様のことが言える。

問題は組み込みソフト開発の現場である。printfデバッグが唯一のデバッグ手段となっているエンジニアをよく見かける。こういった現場ではjtagデバッガを使えば大きな改善が期待できる。

また、「自動単体テストの実施」に従えば、組み込みソフト開発においても、先に述べたような優れたデバッガを使用できる。

ランダムに発生するようなバグにおいても、デバッガは有効であることがある。このようなバグではステップ実行でその原因を特定できないが、例えばデッドロックが起こった場合は、

- デッドロックが起こったプロセスをデバッガにアタッチさせてスレッドの状態を見る
- メモリダンプを取り、デバッガでスレッドの状態を見る(linuxでもwindowsでも同様のことはできる)

のような手法が有効である。

gitの使用

言うまでもないが、新規機能を追加した後にすでに動いていた機能が動かなくなつたのであれば、新規コードが原因であると考えるのが常識である。このようなデグレードの原因を効率的に発見するためには、コード追加前後での差分を見るのが理にかなつていて。バージョン管理システムを使えば、これを正確かつ簡単に行うことができる。

複数人でソフトウェア開発を行う場合、頻繁に各人の作り出すソースコードをマージする必要がある。マージにはある一定確率でのバグ混入は避けられず、バグが発生すればこれを直ちに修正するのが合理的なチーム運営である。バージョン管理システムを使わずに、これを行うことは難しい。

リリースしたソフトウェアの異常動作が報告されたときに、そのバイナリコードを生成したソースコードに完全に一致するソースコードを使用せずに、その原因を特定することは、ほぼ不可能である。バージョン管理システムを使わずにこれを行うことも上記同様に難しい。

このような理由(他にも多数の理由はある)から、バージョン管理システムの導入には議論の余地はないが、数あるバージョン管理システムの中から何を選ぶのかは意見の分かれるところである。

本ドキュメントでは以下のような観点からgitを推奨する。

- 現在も活発に開発されている。
- ウェブドキュメントや書籍が豊富である。
- クラウドサービスがある。
- ライセンス費用がない。
- 有名なオープンソース管理に使用されている。
- 多くのオープンソース管理に使用されているため、gitの知識は現在のプログラマにとって常識となっている。

gitには以下のような基本的なコマンドに加えて、

- add/commit - ソースコードのリポジトリへの登録
- branch/checkout/reset - ブランチ操作
- merge - ブランチ間のマージ
- log - コミットログ

リポジトリの分散をサポートする下記のようなコマンドがある。

- fetch/pull - リモートブランチからの差分取り込み
- push - リモートブランチへの差分転送
- rebase - カレントブランチをリモートブランチに同期

他にも多数のコマンドがあるが、中でも有用なのが、

- bisect - デバッガサポート

である。以下にbisectの使い方を例示する。

本ドキュメントでは、「開発プロセスとインフラ」で述べたように、単体テスト、統合テストを自動化することを推奨している。

本ドキュメントで使用するディレクトリやファイル間の依存関係を解析するためのソフトウェアdepsの開発においても、当然これを実践している。

depsの開発においては、単体テスト、統合テストは下記のようなdeps/build.shによって行われる。

```
// @@@ example/deps/build.sh 3
readonly BASE_DIR=$(cd $(dirname $0); pwd)
```

```

readonly BASENAME=$(basename $0)

$BASE_DIR/../../deep/build/build_core.sh $BASE_DIR $@ -i
# build_core.shによって下記が行われ、何らかのエラーが発生した場合、非0でexitする。
# * ビルド & UT & IT
# * 静的解析
# * 動的解析

```

depsが、

- xxxリビジョンまでは、build.shが成功(値0でexit)
- yyyリビジョンでは、build.shが失敗(非0でexit)

のような状態に陥ったとする(CI(継続的インテグレーション)を導入しているソフトウェア開発では、このような状態は珍しいことない)。この場合、バグが混入したリビジョンを確定することがデバッグの第一歩となる。bisectはこれをダイレクトにサポートする。

```

git bisect start xxx yyy
git bisect run deps/build.sh

```

上記コマンドは、xxx～yyyのリビジョン毎にbuild.shを実行し、build.shが非0でexitした場合、そこで停止する。これによりバグが混入したリビジョンをほぼ工数をロスすることなしに確定できる。

TATの短縮

TATとはターンアラウンドタイムの略語であり、一度の試行に要する時間を指す。デバッグのTATが短ければ短いほどデバッグ効率が良くなることは明らかだろう(これとは対をなす概念が「事後のデバッグ」である)。

再現性の悪いバグには、様々なテストパターンを試す、環境を変えてみる、等の任せの手法も時には必要になるが、以下のような戦略的アプローチが最も有効である。

- printfデバッグを使わない(「デバッガの使用」参照)
- ビルド時間の短縮
- 自動単体テストの実施(ターゲットハードウェアへのプログラム転送が不要なため、組み込みソフト開発には特に有効である)

ビルド時間の短縮

ビルド時間を短くするためには、下記等が施策となる。

- コンパイルの並列化を行う(make -j等)
- ライブラリに細かく分割する(コンパイルの並列化がされやすくなる)
- コア数/スレッド数が多く、SSD等のストレージアクセスが早く、メモリが多いコンパイルマシンを使う
- 差分ビルドが有効に働くように、依存関係の整理を行う
- 不要なインクルードを行わない(静的解析ツールによっては、不要なインクルードを指摘してくれる)

ビルドツールが並列コンパイルをサポートしているのであれば(「コンパイラ」で述べたしたメーカーのIDEは並列コンパイルができなかった)、上記したような高スペックのビルドマシンを導入することで、工数負担なしにコンパイル時間の改善ができる。多少費用は増えるが、人件費とは比較にならないほど安価である(「開発ツール」参照)。

各プログラマがフルビルトを一日に何度も行うことはないので、実践的には差分ビルドが有効になるように依存関係を見直すのが良い。ただし、これには「依存関係の整理」で説明したような努力が必要である。

依存関係の整理

「パッケージとその構成ファイル」で述べたようにソースコードをパッケージ分割し、そのパッケージをライブラリとしてまとめ、ライブラリ間の循環依存がないようにすることで、差分ビルドは有効に働くことが多い(その場合、当然無駄なインクルードもしない必要がある)。

このようなライブラリ構成は、

- 単体テストが容易になる(「自動単体テスト」参照)
- 修正後の影響範囲を明確にしやすい
- ライブラリの転用がしやすい

等他にも良い影響があるため、この構造を維持すべきである。そのためには、

- 静的解析ツール等を使った依存関係の監視(「deps」参照)

- ・ビルドツールの使い方の工夫(「[makeによる依存関係の維持](#)」参照)

等が必要になる。

自動単体テストの実施

自動単体テストは、

- ・デバッグのTATを短縮できる(「[TATの短縮](#)」参照)
- ・統合テストでは実施困難なテストを行うことができる
- ・ほぼ工数をかけることなく、回帰テストを実施できるため、共有リポジトリ内のソースコードをクリーンに保てる
- ・カバレッジを取ることで、テストが不十分なコードを特定できる

等の効果を期待できるため、現在のソフトウェア開発において欠かすことができない手法である。

自動統合テストの実施

自動統合テストは、

- ・ほぼ工数をかけることなく、回帰テストを実施できるため、共有リポジトリ内のソースコードをクリーンに保てる
- ・長時間操作が必要なテスト(メモリーリークや、発生率の悪いタイミング問題等)を、人手をかけずに実施できる
- ・git bisect(「[gitの使用](#)」参照)と組み合わせれば、デグレードが起こった共有リポジトリのリビジョンを特定できる

等の効果を期待できるため、現在のソフトウェア開発において欠かすことができない手法である。

depsに関しても、以下のシェルスクリプトにより自動統合テストを実現している。

```
// @@@ example/deps/it.sh 6
function help(){
    echo "$BASENAME [option]"
    echo " -a BIN : BIN is test target(default:./g++/deps)"
    echo " -d     : debug mode"
    echo " -h     : show this message"

    exit $1
}

...
function test_a2pu() {
    local -r exp=$1
    local -r in=$2
    local -r act=${exp}.act
    rm -f $act

    $APP a2pu -i $in -o $act $dir
    diff $exp $act

    rm $act
}
...
echo test_a 0      && test_a it_data/a_ut_data.txt it_data/p2p_ut_data.txt
echo test_a2pu 0   && test_a2pu it_data/ut_data.pu it_data/p2p_ut_data.txt
echo test_a2pu 1   && test_a2pu it_data/ut_data_s.pu it_data/p2p_ut_data_s.txt
echo test_deps_fake && test_deps_fake
echo test_deps_dep  && test_deps_dep
echo test_deps_dep2 && test_deps_dep2
```

上記でも使ったunix系のコマンドは、

- ・統合テストのみならず、様々な自動化に関して非常に強力なツールである
- ・[cygwin](#)によりwindows上でもunixでの使用感とほぼ同等に使える

ため、[The Art of Command Line](#) 等を参考に学習するべきである。

事後的デバッグ

多くのバグは、組織の公式なテストやユーザによって発見されるため(ユーザには発見されたくないが)、その解析にデバッガが使えないことはよくあることである。このような場合でも、ホスト上のソフトウェアであれば、メモリダンプを取り、事後的にデバッガを利用するることはできるが、このアプローチが必ずしも効果的とは限らない。また、組み込みソフトの場合、メモリダンプの取得ですら簡単ではない。

ここでは、そのような場合に有効な手法について解説する。

ログの取得

第三者によってバグが報告された場合、デバッグのファーストステップは、そのバグを手元で再現させることになるが、その第三者からバグ発生時の正確な手順を聞き出すことは困難である。また、聞き出せたとしても、もし、そのバグが簡単に再現しない場合、その手順を完全に信じることは難しい。

こういったことに備えるために、ソースコードにあらかじめログを仕込んでおくことは昔から行われてきており、今なお有効な手法である。

`deps`の開発にもログは有効であり、

- ログ取得用のクラスと関数テンプレート(「[ログ取得ライブラリの開発](#)」、「[ログ取得ライブラリの開発2](#)」参照)

```
// @@ exampledeps/logging/h/logging/logger.h 11

namespace Logging {
class Logger {
public:
    static Logger& Inst(char const* filename = nullptr);

    template <typename HEAD, typename... TAIL>
    void Set(char const* filename, uint32_t line_no, HEAD const& head, TAIL... tails)
    {
        auto path     = std::string_view{filename};
        size_t npos   = path.find_last_of('/');
        auto basename = (npos != std::string_view::npos) ? path.substr(npos + 1) : path;

        os_.width(12);
        os_ << basename << ":";

        os_.width(3);
        os_ << line_no;

        set_inner(head, tails...);
    }

    ...

private:
    void set_inner() { os_ << std::endl; }

    template <typename HEAD, typename... TAIL>
    void set_inner(HEAD const& head, TAIL... tails)
    {
        using Nstd::operator<<;
        os_ << ":" << head;
        set_inner(tails...);
    }

    template <typename HEAD, typename... TAIL>
    void set_inner(char sep, HEAD const& head, TAIL... tails)
    {
        using Nstd::operator<<;
        os_ << sep << head;
        set_inner(tails...);
    }

    ...
};

} // namespace Logging

#define LOGGER_INIT(filename) Logging::Logger::Inst(filename)
#define LOGGER(...) Logging::Logger::Inst().Set(__FILE__, __LINE__, __VA_ARGS__)
```

- オブジェクトを文字列変換するため関数

```
// @@@ exampledeps/app/src/deps_opts.h 53
std::string ToStringDepsOpts(DepsOpts const& deps_opts, std::string_view indent = "");
inline std::ostream& operator<<(std::ostream& os, DepsOpts const& opts)
{
    return os << ToStringDepsOpts(opts);
}
```

- ログの取得方法

```
// @@@ exampledeps/app/src/main.cpp 43
std::unique_ptr<Dependency::ScenarioGenerator> gen_scenario(App::DepsOpts const& opt)
try {
    using namespace Dependency;

    switch (opt.GetCmd()) {
        case App::DepsOpts::Cmd::GenPkg:
            LOGGER("start GenPkg");
            return std::make_unique<PkgGenerator>(opt.In(), opt.IsRecursive(), opt.Dirs(),
                                                    opt.Exclude());
        case App::DepsOpts::Cmd::GenSrc:
            LOGGER("start GenPkg");
            return std::make_unique<SrcsGenerator>(opt.In(), opt.IsRecursive(), opt.Dirs(),
                                                    opt.Exclude());
        ...
    }
    catch (std::runtime_error const& e) {
        LOGGER("error occurred:", e.what());

        std::cerr << e.what() << std::endl;
    }
    return std::make_unique<ScenarioGeneratorNop>(false);
}

...
int main(int argc, char* argv[])
{
    App::DepsOpts d_opt{argc, argv};

    LOGGER_INIT(d_opt.Log() == "-" ? nullptr : d_opt.Log().c_str());

    LOGGER("Options", '\n', d_opt);

    auto out_sel = OStreamSelector{d_opt.Out()};
    auto exit_code = gen_scenario(d_opt)->Output(out_sel.OStream()) ? 0 : -1;

    LOGGER("Exit", exit_code);

    return exit_code;
}
```

によりログを取得している。これにより以下のような内容のログファイルが取得できる。

```
app/src/main.cpp:109:Options
cmd      : GenPkg
in       :
out      :
recursive : true
src_as_pkg: false
log      : it_data/log_exp.txt.act
dirs     : ut_data
exclude  :
parsed   : true
app/src/main.cpp: 51:start GenPkg
app/src/main.cpp:114:Exit:0
```

なお、組み込みソフトの場合、ファイルが使えないこともよくあるため、ログをメモリ上にストアし、モニターの機能によりそれを吸い上げる必要がある(ファイルが使える場合にもこれは同様である)。

モニター

先に書いた通り、組み込みソフトの場合、バグ発生後にメモリダンプを取り、後からデバッグで調査することは難しい。また、ログを取っていた場合でもそれを吸い上げる方法が無ければ、それを利用できない。このような場合には、あらかじめデバッグ用の機能を入れることが有効である。本ドキュメントではこういった機能をモニタープログラム、あるいは単にモニターと呼ぶ。

モニターには以下のような機能が必要である。

- ホスト上のターミナルソフト(Tera Term等)と接続するため入出力(通常はuartを利用)
- 下記のコマンドを実装するため簡易シェル(ntshell等)
- メモリに保存されたログの出力
- デバッガの基本機能
 - メモリダンプ
 - メモリ(回路のレジスタ等)への書き込み
 - レジスタダンプ(各種レジスタやCPUの状態の表示)
 - リアルタイムOSのリソースの表示
 - スレッドの状態の表示やスタック消費量の計測
 - メモリ保護機能のON/OFF
 - グローバルなnew/deleteの使用状況(「グローバルnew/deleteのオーバーロード」参照)

レジスタダンプは、CPU例外場所の特定やコールスタックの構築には欠かせない機能であるが、使用しているCPUやコンパイラの仕様を理解しなければ実装できない。ARMを使っているのであれば、<https://developer.arm.com/>等のネットドキュメントに十分情報が掲載されている。他のランタイム系にも同様のドキュメントは提供されているはずである。

実装難度が高い、もしくは時間がない等の理由でこういった機能の実装をおろそかにすれば、必ずと言っていいほど極度に解析困難なバグに悩まされることになる。

メモリ保護機能

組み込みソフトは、仮想アドレスを使わず、直接ハードウェアアドレスを使って動作することが一般的である。その場合、CPUのデフォルトの設定では、リードオンリー領域(インストラクション、文字列リテラル、constなPOD等)に対してもライトアクセスできるため、例えば、クラスへのポインタにnullptrが代入された状態で、そのポインタをデリファレンスし、ライト系のメンバ関数を呼び出せば(ptr->f)のような呼び出し)、0アドレス付近のコードが破壊される。

多くのCPUでは、この領域には割り込みベクタが配置されているため、CPU例外が発生する。それを捕捉する機能が実装されていなければソフトは暴走する。CPU例外を捕捉する機能が実装されている場合でも、その例外コードは「不正なインストラクションの実行」を示すことがほとんどであるため、真の原因特定は難しい。

使用しているCPUがメモリ保護機能(MMUやMPUと呼ばれることが多い)をサポートしているのであれば、それを使いリードオンリー領域を保護することで、このような状況を回避することができる。

適切にメモリ保護機能を活性化した状態でも、リードオンリー領域にライトアクセスした場合、やはりCPU例外が発生するが、CPU例外を捕捉し、その時のレジスタをダンプすれば、「リードオンリー領域にライトアクセスしたインストラクションが配置されているアドレス」を特定することができる。後は、ツールチェーンに付属するaddr2line、readelf、objdump、nmのようなバイナリ解析ツールを使い、アドレスからソースコードを特定すればよい。

ただし、jtagデバッガを使用する場合、メモリ保護機能の設定には注意が必要である。

jtagデバッガがブレークポイントを設定する方法は2つある。一つはハードウェアブレークであり、他方はソフトウェアブレークである。ソフトウェアブレークを設定する場合、デバッガは、一時的に「ブレークを指定されたソースコードに対応するインストラクション」に「ソフトウェアブレークするためのインストラクション」を上書きする。

メモリ保護をしている場合、jtagデバッガによるこの動作はCPU例外を発生させことがある。これを避けるためには、一時にメモリ保護機能を不活性にすればよい。これをモニターから実行できるコマンドとして実装すると便利である。

スタック消費量の計測

強固なメモリ保護を持つOS上のソフトウェアがスタックのオーバーフローを起こした場合、それを示す例外コードが、コアダンプ等の何らかの方法で通知されるため、多くの場合、デバッガはそれほど困難ではない。

組み込みソフトの場合でのスタックオーバーフローは、「スタックに割り当てられたメモリのすぐ上にある領域が破壊される」というデバッガ困難な現象を引き起こす(これにより、メモリ破壊だけではなくスレッドの暴走等も起こり得る)。

こういったことを防ぐには、スタックがある特定のバイト列(0xdeadbeaf等)でフィルし、スタックの消費量を計測できるようにすることが重要である(0xdeadbeafでないデータが書き込まれている最上位アドレスまでが使われたと判断できる)。当然ながらスタックの消費量を表示するコマンドはモニターから実行できれば便利である。

潜在的バグの削減

すでに述べたように共有リポジトリ内のソースコードをクリーンに保つことは、デバッグの効率化に繋がる。そのための手段として以下の三点が有効である。

- コードインスペクション

- コードの静的解析

- コンパイラによる静的解析

- 静的解析ツールの使用

静的解析ツールを選択する最も重要な基準は、S/N比である(S(シグナル)とはバグやバグに近いコードであり、N(ノイズ)とは修正の必要のないコードである)。S/N比が悪いと、早晚プログラマはこのツールの指摘を無視し始める。ツールの指摘を確認するようにプロセス的に強制しても、おざなりな対応が行われるだけなので、S/N比の良い下記のようなツールを選ぶべきである。

- scan-build(「scan-buildによる静的解析」参照、ライセンス費フリー)

- coverity

- コードの動的解析

- コードにassert()を入れる。これは外部ツールではなく、標準ヘッダcassertをインクルードすれば使用できる(「assertion」参照)。

- g++/clang++を使用しているのであれば、マクロ GLIBCXX_DEBUGを定義してビルドを行うことで、STLの誤使用に関するバグが指摘される。多くのコンパイラがこのような機能を持っている(Visual Studioでは、コンパイラのMDdオプションを指定する)。

- 動的解析ツールの使用

sanitizer(「sanitizerによる動的解析」参照)やvalgrind等が優れている。LeakTracer等のメモリリーク専用のツールもある。

静的解析ルールと違い動的解析ツールは、プログラムを動作させなければバグを発見できないが、CI(継続的インテグレーション)と組み合わせれば、効率的に運用できる(「自動統合テストの実施」参照)。

なお、depsに関しては、以下のようにすることで単体テスト、統合テスト時にsanitizerによる動的解析を行っている。

```
// @@@ example/deps/Makefile 173
ut: $(EXES_UT_DONE) # 単体テスト用ターゲット
it: $(EXE_IT_DONE) # 統合テスト用ターゲット

SANITIZER_DIR=sanitizer/
.PHONY : sanitizer
san: # g++ sanitizerビルド
$(MAKE) O=$(SANITIZER_DIR) \
    SANITIZER_OPT="-fsanitize=address,leak,undefined,floating-point-exception,overflow"
san-ut: sanitizer # g++ sanitizerバイナリでの単体テスト
make O=$(SANITIZER_DIR) ut

san-it: sanitizer # g++ sanitizerバイナリでの統合テスト
make O=$(SANITIZER_DIR) it
```

難しいバグの対処

一般に下記のような現象の原因特定は難しい。その再現性が悪いとなおさらそうなる。

- ハングアップ

- メモリ、ファイルハンドル等のリソースの枯渇

- 不正メモリアクセス

- 不正命令の実行

本ドキュメントの提言に従ってソフトウェア開発を進めれば、そのリスクを低く保てるが、それでもこののようなバグは発生する。

ハングアップについては、ログの取得、デバッガの使用が有効なデバッグ方法となることが多い(もちろん組み込みの場合は、エニターが必要である)。また、バージョンが特定できるのであればgitのdiff/bisect等(「デバッガの使用」参照)が役に立つ。

リソースの枯渇については、scan-buildやcoverity等の静的解析ツール(「潜在的バグの削減」参照)を使うことで、ピンポイントでリーク箇所を特定できることがあるため、まずはこれらを使ってみることを推奨する。これで発見できない場合、動的解析ツールの使用を検討すべきだろう。

恒久的には、「RAII(scoped guard)」で紹介したようなリソースの自動解放のパターンを使用し、そもそもこのような問題が発生しないよう改善すべきだろう。

バグの性質上、再現/回帰テストには長時間のアプリケーション動作が必要になることがある。テスターにひたすらクリックさせるようなテストは非効率であるだけでなく、人道的にも問題になるかもしれないため、自動統合テストの実施も必須になる。

不正メモリアクセス、不正な命令の実行に関しては、下記のようなメモリ破壊を疑うべきだろう。

- スタックの破壊
- 静的なオブジェクトの破壊
- newしたオブジェクトの破壊やダブルデリート

これらに関しても、上記したような静的/動的解析ツールの活用が有効になる場合が多い。これで問題が発見できない場合、以下のような知識が必要になる。

- コンパイラのスタックの生成方法、CPUレジスタの使用方法、アセンブラー
- コンパイラに付属するバイナリツール
- CPU例外に対するランタイム系の処理方法

これらの知識はC++プログラマの基本スキルセットの一部だと思うが、実際にはこれらの知識を持たないプログラマも少なくない。そのようなプログラマにはこれを良い機会と捉え、新知識の習得に励んでほしいが、目の前に迫った納期の前に、そのような悠長なことを言つていられないこともあるだろう。そのような場合は、上級なプログラマに助成(「助成の依頼」参照)を頼むこと以外、建設的なデバッグ方法はない(やみくもにソースコードを眺めたり、ログやprintfを仕込んで、よほどの幸運でもない限りバグは見つからない)。

助成の依頼

ここでデバッグの「助成の依頼」とは以下のいずれかを指す。

- 動作説明を聞いてもらう
- 一緒にデバッグしてもらう

「動作説明を聞いてもらう」相手は、同プロジェクトのプログラマがベストであるが、実際には難しいことが多い。こういった時(ほとんどの場合そうなるが)には、相手は誰でもいい。それすらできない場合、ペットでも無生物でも良い。他者に説明することで、自分の理解を確認することができ、エアポケットに落ち込んでしまった見落としを発見できる可能性が高くなる。

「一緒にデバッグしてもらう」相手は、「難しいバグの対処」で述べたような知識豊富なプログラマで無ければならない。これができるか否かは、日頃の人付き合いに依存するが、知識豊富なプログラマの多くは、人から頼られることを拒否しない(当然状況によるが、通常は喜んで手伝ってくれるはず)。

下記に筆者が他者から助成を頼まれた事例を取り上げる。もしこのような現象を独力で解決できる自信がないのであれば、まずは、思い切って依頼してみるとことだ。

現象1 - 不思議なハングアップ

この現象は、モニター機能を実装した組み込みソフトウェアのテストで起こった。あるプログラマが、自分の書いた機能を長時間テストするために、Tera Termをターミナルボードに接続し、そのソフトウェアに組み入れたテストをTera Termから起動して帰宅した。翌日出社するとそのテストは途中でハングアップしてるように見える状態で止まっていた。彼はなんとなく、Tera Termにリターンキーを入力した。すると、途中で止まっていたテストが再び走り出し、数時間後に完了した。この状態でできることはほとんどないので、彼は何もなかつたことにして、その日の帰宅前にまた同様のテストを走らせた。次の朝、また同じことが起きたため、この現象のデバッグに取り掛かろうとしたが、やり方が全く分からなかった。

この日の朝会はすでに終わっていたため、次の日の朝会でこの現象を報告し、助成を求めた。

この報告を聞いて難問題であることを確信した私は、すぐにでもデバッグを手伝いたかったが、後2、3日かかるプログラミングの最中であったため、彼に「その現象のTATがもっと短いテストパターンを探る」ように指示し、自分の作業をつづけた。2日後、TATが30分までに短縮されたこの現象のデバッグに参加した。

このデバッグの難しさは、この現象が起こっている時にモニターから何か文字を打ち込むと、ハングアップ状態が解けてしまうことであった。

現象からRTOSのバグであると推測して、jtagデバッガでRTOSのリソースを観察して、以下のことが分かった(RTOSのリソースはより便利にモニターから見れたが、モニターを使うとこの現象が霧散してしまうため、この機能は使えない)。

- カレントスレッドポインタがNULLである
- テストを実行するスレッドはレディーキューに繋がれている

RTOSの仕様では、

- 各スレッドには、その情報を保持するスレッド構造体が存在する
- カレントスレッドポインタは、実行中のスレッドのスレッド構造体を指す
- 実行可能(待っていたイベントが起こった等)なスレッドのスレッド構造体はレディーキューに繋がれる
- スレッドのディスパッチャは、RTOSのシステムコールか割り込みルーチンから実行される。
- ディスパッチャは、一旦カレントスレッドポインタをNULLにして、レディーキューに繋がれたスレッドの中で最高プライオリティを持つものをそこから取り出し、カレントスレッドポインタに代入した後、そのスレッドを走らせる
- カレントスレッドポインタがNULLの際に発生したチック割込みからは、ディスパッチャは呼び出されない(プリエンプション対象のスレッドがないため)。

となっていた(このような詳細仕様は、通常ドキュメントには書かれていない。また、書かれてもそもそもRTOSを疑っているのだからその情報は信じられないため、RTOSのコードハック以外の方法はない)。

以上から、

- 「カレントスレッドポインタがNULLである」ことは「現在実行中のスレッドは存在しない」
- 「レディーキューが空ではない」ことは「実行可能なスレッドが存在する」

を意味する。このような中途半端な状態はディスパッチャ内でしか発生しないので、以下のことが分かったことになる。

- この現象はハングアップである
- カレントスレッドポインタは静的変数であるため、「現象2 - グローバル変数の破壊」のようなことがない限り、RTOSのバグである

「現象2 - グローバル変数の破壊」で書いたような方法により、この現象が変数破壊でないことを確信した我々はさらにRTOSのコードハックを続けた。

ディスパッチャは、ディスパッチャ自身に割り込まれることがあるため、カレントスレッドポインタやレディーキューにアクセスするときに割り込みを禁止し、その処理の終了後に割り込みを許可する(これをクリティカルセクションの保護と言う)。もしクリティカルセクションの保護をしなければこの現象が起きたことが分かったが、クリティカルセクションを保護しているソースコード内のインラインアセンブリに問題がないことも分かり、デバッグは行き詰った。とはいっても放置できる問題でもないため、コンパイル後のバイナリを逆アセンブルして、動作を確かめることにした。これは苦痛に満ちた作業だが、他の方法を思いつかなかった。

そして、ようやく原因を特定できた。「割り込み許可を行なうインストラクション」と「カレントスレッドポインタへNULLを代入するインストラクション」の順番が、ソースコードに書かれている順番と逆になっており、このためカレントスレッドポインタの操作が割り込みから保護されていなかったのだ。

結果はRTOSのバグではなく、コンパイラのバグ(最適化に伴うインストラクションの不適切な入れ替え)であった。

一旦ソースコードにこのバグのワークアラウンドを入れ、コンパイラメーカーにこの現象を報告した。2か月後にこのバグが修正されたコンパイラがリリースされたため、ソースコードを元に戻し、この問題が再現しないことを確認してこの問題をクローズした。

現象2 - グローバル変数の破壊

新人プログラマから以下のような現象の助言を求められた。

- 静的オブジェクトのメンバ変数一つが0を設定していないにもかかわらず、0になる。
- デバッガでアドレスが近い静的オブジェクトを見てみたが、他のオブジェクトには異常がないように見える。
- この問題は1日一回程度起る(起せる)

この現象を聞いて、まず疑ったのは「そのメンバ変数に0を代入していないつもりになっているだけで、実際には0を代入している」ことであったため、ソースコードレビューをしたのだが、問題は見つからなかった(ソースコードが汚いことは分かったが)。

次に行すべきは、「この現象が静的メモリの破壊である」ことの確定である。壊されたと推定されたオブジェクトの前後に以下のようないオブジェクトを挿入して、再現テストをすることにした。

```
uint32_t pad0[] = { 0xdeadbeaf, 0xdeadbeaf, 0xdeadbeaf, ... };
Collaps collaps_obj; // 破壊を疑っているオブジェクト。実際はシングルトンだった。
uint32_t pad1[] = { 0xdeadbeaf, 0xdeadbeaf, 0xdeadbeaf, ... };
```

ここで注意すべきは、この変更で静的オブジェクトの配置が大きく変わっていないことを確かめることである。linuxの場合、それにはnmコマンドが最適である。

参考のため、「deps」の実行形式バイナリにnmを適用した例を示す。

```
> nm --demangle g++/deps
000000000008b000 D __data_start
00000000000005b0 t __do_global_dtors_aux
0000000000089ff8 d __do_global_dtors_aux_fini_array_entry
```

```

000000000008b008 D __dso_handle
    U __dynamic_cast@@CXXABI_1.3
    U __errno_location@@GLIBC_2.2.5
0000000000089fa8 d __frame_dummy_init_array_entry
    w __gmon_start__

...
U abort@@GLIBC_2.2.5
000000000008b440 B optarg@@GLIBC_2.2.5
000000000008b590 B optarg@@GLIBC_2.2.5
000000000008570 t register_tm_clones
    U strchr@@GLIBC_2.2.5
    U strcmp@@GLIBC_2.2.5
    U strtol@@GLIBC_2.2.5
    U tolower@@GLIBC_2.2.5

```

nmにより挿入前後でpad0、pad1以外に並びが変わっていないことを確認した後、再現テストを行い(ASLRを行うOS上のアプリケーションの場合、それを非活性化することも必要である)、pad0の1ワードだけが0になっていることを確認できた。これでメモリ破壊であることが確定した(さらに運のいいことに、0が書かれていたアドレスはpad0、pad1挿入前後で同じであった)。

もし、「デバッガの使用」で説明したようなことが可能ならば、pad0への書き込みアドレスにライトトラップをかけて再現テストを行えば、不正なデータの書き込み個所を特定できる。この例の場合もデバッガを使用しバグを特定した。

デバッガが使用でない場合でもやり方はいくつかあるが、最も単純なのはassert()を使う方法である。

このバグの場合、疑うべきは静的な配列のオーバーランである。nmによりリストアップした静的な配列のライトアクセス箇所のインデックスに下記のようなコードを挿入して、再現テストを行えばよい。

```
assert(0 <= index && index < array_length(global_array));
```

静的な配列へのライトアクセス箇所が多い場合、苦痛な作業となるが、典型的なアンチパターンである静的な変数を多用した罰である考え方、その苦痛を受け入れるべきだろう。

現象3 - プロセスのスローダウン

組み込みlinuxでの開発で起こったこの現象は、usbストレージに大量のデータをコピーするプロセスがスローダウンするというものであった。この問題を最初に聞いたのは、この対処に私が加わる3か月ほど前であった。この時、難問であるとは思ったが、リリースまでまだ日数があったことと、担当が海外のチームだったため、スルーした。

3ヶ月後、このチームサイトに出張した折、この問題がまだ解決できていないことを知った。彼らがこの3ヵ月でこの問題に対して行ったことは、

- 1時間程度で再現させられるshellスクリプトを開発した(当初は数日に1回程度の頻度)
- 以下のことを発見した
 - この現象が起こった時、プロセスのnice値が最低プライオリティになっている
 - nice値をコマンドから変更しプライオリティを上げると問題はなくなる
- 最新のusbストレージのドライバを変更せずにそのまま使っており、ネットにもバグ等は報告されいない

等であった。ここから推測されることは、

- usbストレージの最新の標準ドライバにこれほど高い再現率のバグがあるとは思えないため、この現象はusbとは関係ない
- nice値が不正になるということはチック生成と密接な関係がある

ということである。

チックの生成がどうなっているかを聞いたが誰も答えられなかつたため、ソースコードからそのドライバを探し出した。subversionの履歴から、このコードはこのチームが一年ほど前に作ったことが分かったため、担当者を呼び出しソースコードレビューを行い、以下のような怪しいコードを見つけた。

```

#define CHICK_REG_ADDR 0x00.....
#define CHICK_REG_VALUE ((volatile ulonglong*)CHICK_REG_ADDR)

...
ulonglong chick = CHICK_REG_VALUE;

if(chick_previous > chick) { // chick_previous: 前回ロードしたchick
    ... // 柄上り対策
}

```

この辺りのハードウェアの仕様は良く知らなかつたが、チックを作り出すチップとCPUはおそらく32ビットバスで繋がっていたはずである。すぐにハードウェアの担当者に連絡を取り、32ビットバスであることを確認した。(経験の長いエンジニアやチームリーダーはチームをまたがる情報に簡単にアクセスできるため、彼らに助成を依頼すべき理由の一つになる)。

この開発で使用していたgccでのulonglongは64ビットであるため、

```
ulonglong chick = CHICK_REG_VALUE;
```

は以下のように段階的に読み込まれ、それぞれの値を合わせることでchickの値が生成される。

1. chick0 = (チックレジスタ63-32ビットの値)
2. chick1 = (チックレジスタ31-0ビットの値)
3. chick = (chick0 << 32) | chick1

チックレジスタの値が0x0000'0000'ffff'ffffのような場合、上記1でchick0には0がロードされるが、上記2はそれより少し遅れて実行されるため桁上がりが起こりほとんど0の値がchick1にロードされる。上記3によりchickもほとんど0のような値になるため、前回ロードしたchickの値よりも小さい値となる。これは長い時間が経過したことを表してしまうため、現在実行中のプロセスがきわめて多くのCPUタイムを使用したことになり、linuxのスケジューラはこのプロセスのプライオリティを最低に引き下げた。

これがこの問題のメカニズムだった。メカニズムさえわかってしまえば対処は簡単である。担当者はすぐにコードを修正し、私がそれをレビューした後、1時間でこの問題を再現させられるスクリプトによりテストを開始した。2日後、問題がないことを確かめてこの問題をクローズした。

現象4 - バックトレースが見れない

関数Aが関数Bを呼び出した場合、Bの処理の完了後、「AがBを呼び出したアドレスの次のアドレス」に戻る必要がある。このアドレスを記録するためのレジスタをリターンアドレスレジスタと呼ぶことにする。さらにBが関数Cを呼び出す場合、リターンアドレスレジスタは「BがCを呼び出したアドレスの次のアドレス」で上書きされる。こうなるとBからAに戻れなくなるため、BはCを呼び出す前にリターンアドレスレジスタをスタックにプッシュする。

デバッガのバックトレース機能を実行して表示される関数一覧は、スタックに保存されたものを含むリターンアドレスレジスタから生成される。従って、ソフトウェアがクラッシュした時、バックトレースが見れない原因は、

- スタックが破壊された
- スタックポインタが破壊された

のいずれかであると思って良いが、C/C++からスタックポインタを破壊することは難しいため、OSの開発でない限り、スタック破壊を疑うべきである。

スタックが破壊された影響でバックトレースが見れない場合でも、ほとんどの場合、多少は表示されるため、スタック破壊が発生する関数を絞り込める。そのような関数群に下記のようなログを仕込むことでバグの個所をさらに絞り込める。

```
// どこかの.cpp内で以下の定義
char const* global_last_func;

// 絞り込んだ関数の先頭行
extern char const* global_last_func;
global_last_func = __func__;
```

該当する関数群が複数のスレッドから呼び出される場合、thread_localを使った工夫が必要になるが、概ね同様のアプローチでバグが発生した関数を特定できる。

基本的なデバッグ方法であるため、わざわざ説明する必要はないとも思ったが、「バックトレースが見れないこと」と「その原因のほとんどがスタック破壊であること」の論理的な繋がりを知らないプログラマが多いため、この節を書いた。

現象5 - newしたオブジェクトの破壊

「開発中のソフトウェアが毎回違う場所でクラッシュする」現象について助力を求められたことがある。クラッシュ時にデバッガからthisを表示すると毎回違うクラスであるという。こういった場合、下記のようなコードが原因であると推測できる。

- newした配列やstd::vector等のオーバーラン
- delete済のポインタを使ったオブジェクトへの書き込み操作
- 二重delete

newした配列(「[メモリアロケーション](#)」参照)やstd::vector等のオーバーランに関しては、多くの場合、「潜在的バグの削減」で触れたようなコード解析で発見できるが、このチームはすでにこの方法を試して、効果がないことを確認していた。この現象につながるような静的解析での指摘もないことから、再現したらメモリダンプを取るように指示して現場を後にした。

数日後、メモリダンプを解析するよう依頼を受けたため、デバッガを使用して壊されたオブジェクトを観察し、オブジェクトの破壊を確認した。thisをいくら見ても壊れていること以外何もわからないので、thisアドレス辺りを16進数表示してみた。

アドレスからその型を特定することは極めて難しいが、この場合は幸運だった。16進数表示の中にヒープのアドレスらしきバイト列があつたので、そこをさらに16進数表示して、ASCII文字列のようなバイト列を発見した。このバイト列をソースコードから検索すると、下記のようなコードを見つかった。

```
// f.cpp
void f(char* s) {
    // sの後ろへの書き込み
    ...
}

// g.cpp
g() {
    std::string s = "xxx";
    ...
    f((char*)s.c_str());
    ...
}
```

最近CプログラマからC++プログラマに鞍替えした人が書いてしまったコードだそうだ(このようなコードはCでもダメだが)。

現象6 - STLのバグ?

あるプロジェクトの朝会でのこと。プログラマの一人がSTLのバグを発見したということで、他のメンバーに注意を喚起した。このプロジェクトで使用していたコンパイラは広く使われているものだったので、そんなことはないだろうと思いつつ、念のために確認するとコードは下記のようなものだった。

```
// @@@ example/etc/debug.cpp 13
class Pred { // 0を最大3個まで見つける
public:
    Pred() noexcept {}
    bool operator()(int32_t i) noexcept
    {
        if (found_ > 2) {
            return false;
        }

        if (i == 0) {
            ++found_;
            return true;
        }

        return false;
    }

private:
    size_t found_{0};
};

TEST(Debug, no_ref)
{
    auto v = std::vector{0, 0, 0, 0, 1, 2, 3, 4, 5};

    v.erase(std::remove_if(v.begin(), v.end(), Pred{}), v.end());

#if 0 // 本来ならば下記のテストがパスするはずだが。
    ASSERT_EQ((std::vector<int32_t>{0, 1, 2, 3, 4, 5}), v);
#else
    ASSERT_EQ((std::vector<int32_t>{1, 2, 3, 4, 5}), v);
#endif
}
```

なるほど、初心者がよくやるミスである。

STLの関数は引数オブジェクトをコピーすることがあるため、下記のように書くべきだ。

```
// @@@ example/etc/debug.cpp 49
TEST(Debug, ref)
{
    auto v     = std::vector{0, 0, 0, 0, 1, 2, 3, 4, 5};
```

```

    auto pred = Pred{};

    v.erase(std::remove_if(v.begin(), v.end(), std::ref(pred)), v.end());
}

ASSERT_EQ((std::vector<int32_t>{0, 1, 2, 3, 4, 5}), v);
}

```

この程度の述語であればラムダ式を使い下記のように書いても良いだろう。

```

// @@@ example/etc/debug.cpp 61
TEST(Debug, lambda)
{
    auto v = std::vector{0, 0, 0, 0, 1, 2, 3, 4, 5};

    auto found = 0;
    auto it     = std::remove_if(v.begin(), v.end(), [&found](int32_t i) noexcept {
        if (found > 2) {
            return false;
        }

        if (i == 0) {
            ++found;
            return true;
        }

        return false;
    });
    v.erase(it, v.end());

    ASSERT_EQ((std::vector<int32_t>{0, 1, 2, 3, 4, 5}), v);
}

```

このちょっとした事件で、このチームは、ありがちなSTLの誤用パターンを認知できた。また、このバグはcopyコンストラクタを=deleteすれば防げたため、改めて「[特殊メンバ関数](#)」で指摘したことの重要性を確認することもできた。と考えれば、この程度の知識で全員に注意を喚起した彼の勇気を称えるべきだろう。

現象7 - 解放後のrvalueへのアクセス

下記の関数gen_strは、

```

// @@@ example/etc/debug.cpp 86
std::string gen_str(std::string const& str)
{
    return do_heavy_algorithm(str); // 何らかの重い処理
}

```

下記のように文字列リテラルから変換されたされたstd::stringオブジェクトである rvalueを引数に取ることができる。

```

// @@@ example/etc/debug.cpp 96
auto str = gen_str("haha"); // "haha"は、std::string{"haha"}に変換される

```

このテンポラリオブジェクトは、それをバインドしたリファレンスがスコープアウトするまで存在するため、gen_strの仮引数リファレンスでバインドされたstd::stringオブジェクトであるrvalueは、この関数がリターンするまで存在し続ける。この仕様により、gen_str内ではこの仮引数に安全にアクセスすることができる。

この関数の処理が重すぎたため、下記のように並列化したが、これにより動作が不安定になった。

```

// @@@ example/etc/debug.cpp 104
std::future<std::string> gen_future(std::string const& str)
{
    return std::async(std::launch::async, [&str] { return do_heavy_algorithm(str); });
}

```

この関数は、下記のような呼び出しでクラッシュしてしまうことがある。

```

// @@@ example/etc/debug.cpp 129
auto f = gen_future("haha");

```

上記関数gen_futureの中で、

1. ラムダ式が生成され、そのラムダ式はrvalueのリファレンスをメンバとして保持する。
2. そのラムダ式を引数としたstd::asyncによりスレッドが生成される。

3. それを保持したstd::futureが生成され、関数がリターンする。

のようなことが行われ、この後、

4. gen_futureの呼び出しにより生成されたstd::stringオブジェクトであるrvalueは破棄される。

これはgen_futureを実行するコンテキスト上で行われるが、

5. それとは別のコンテキスト(スレッド)上でラムダ式が実行される。

上記4,5のどちらが先に実行されるかは、その状況に依存するが、4が5より先に実行された場合、このラムダ式は、すでに破棄されたrvalueへアクセスしてしまい、クラッシュを引き起こす。

この修正は、下記のようにラムダ式のリファレンスキャプチャをコピーキャプチャにすれば良い。

```
// @@@ example/etc/debug.cpp 139

std::future<std::string> gen_future(std::string const& str)
{
    return std::async(std::launch::async, [str] { return do_heavy_algorithm(str); });
    // ^^^ コピーキャプチャ
}
```

gen_futureと同様の仕様を持つ以下の関数

```
// @@@ example/etc/debug.cpp 166

std::thread gen_thread_lambda(std::string const& str, std::string& str_out) // str_outに結果出力
{
    return std::thread{[&str, &str_out] { str_out = do_heavy_algorithm(str); }};
}
```

にも、下記のように呼び出すことで、ほぼ同様の問題が発生する。

```
// @@@ example/etc/debug.cpp 193

auto str_out = std::string{};
auto t        = gen_thread_lambda("haha", str_out);

t.join();
```

この修正も、下記のようにラムダ式のリファレンスキャプチャをコピーキャプチャにすれば良い。

```
// @@@ example/etc/debug.cpp 206

std::thread gen_thread_lambda(std::string const& str, std::string& str_out) // str_outに結果出力
{
    return std::thread{[str, &str_out] { str_out = do_heavy_algorithm(str); }};
    // ^^^ コピーキャプチャ
}
```

gen_thread_lambdaのC++03のスタイルである以下の関数

```
// @@@ example/etc/debug.cpp 236

void thread_entry(std::string const& str, std::string& str_out)
{
    str_out = do_heavy_algorithm(str);
}

std::thread gen_thread_func(std::string const& str, std::string& str_out) // str_outに結果出力
{
    return std::thread{thread_entry, std::ref(str), std::ref(str_out)};
}
```

にも、やはり同様の問題が発生する。

上記修正とほぼ同様に、下記のようにリファレンス渡しをコピー渡しにすれば修正できる。

```
// @@@ example/etc/debug.cpp 278

void thread_entry(std::string str, std::string& str_out)
// ^^^ コピー渡し
{
    str_out = do_heavy_algorithm(str);
}

std::thread gen_thread_func(std::string const& str, std::string& str_out) // str_outに結果出力
```

```
{  
    return std::thread{thread_entry, str, std::ref(str_out)};  
    //  
    ^^^ コピー渡し  
}
```

以上で見てきたように、C++ではリファレンスがわかりづらいバグを生み出してしまったことがある。リファレンスを使わなければこのような問題を避けることができるが、実行速度が遅くなり、C++を使用する意味がなくなる。

従って、C++で効率よくプログラミングするためには、このような微妙な問題を的確に記述できなければならない。

まとめ

おぞましいことだが、修正の難しいバグをその場しのぎ手法(グローバル変数を使う等)で回避したり、周知されていない再現性の低いバグをなかつたことに対する等の、プロとしてあるまじき行為への誘惑にかられることがある。ガッツはそういう悪魔のささやきにあらがうための重要な要素だが、そのような精神論のみならず、このドキュメントで示したような手法を自在に使えるようになることが、おぞましい行為から自身を守る手段となる。

一旦ダークサイドに落ちてしまえば、それは習慣となり、そこでプログラマとしての成長は終わる。そうならないために学習あるのみである。

開発ツール

この章で触れる開発ツールとは、以下の3つである。

- コンパイラ
- デバッガ
- エディタ/IDE

開発ツールはプログラマの生産性に大きく影響を及ぼすため、ソフトウェア開発プロジェクトにとって極めて重要なファクターである。

この章の構成

コンパイラ

デバッガ

printfデバッグ

エディタ/IDE

筆者の開発ツール環境

vim/neovimの設定

費用

コンパイラ

コンパイラの要件とは、以下のようなものである。

- C++11以上の規約に準拠している。
- バグが少なく、その修正も早い。
- 静的、動的検査機能が豊富である(「コード解析」参照)。
 - 静的検査とはコンパイル時の警告や、clangのscan-buildのようなものを指す。
 - 動的検査とはg++/clang++に置けるsanitizerのようなものを指す。
- コンパイルが速い。
- カバレッジの計測ができる(「単体テストのサポートツール」参照)。

当然の事柄のように思えるかもしれないが、ある大手半導体メーカー純正のコンパイラ(IDE)は、言語規約に準拠しておらず、下記のようなコードが警告すらなくコンパイルできた。

```
int f(char const* arg)
{
    char* arg_local = arg; // <- これがコンパイルできてしまう。
    ...
}
```

さらに驚くことに、組み込みソフトウェア専用のコンパイラであるにもかかわらず、上記constをvolatileとしても同様にコンパイルできた。

これらバグは指摘しても修正されず、少なくとも数年にわたって放置された。筆者がこのコンパイラを使う機会をなくしたため、修正されたかどうかは未確認である。プログラマのちょっとしたミスを指摘できない、このようなコンパイラを使ってはならない。

私がお勧めするのは以下のコンパイラである。

- gcc/g++
- clang/clang++
- Visual Studioコンパイラ(少々納得できない仕様もあるが)

他にもたくさんあるはずである。コンパイラの選定は重要な事項と心得て慎重に選んでほしい。

なお、windowsアプリケーションの開発にもclang/clang++は適用でき、その場合、sanitizerを使用できる。Visual Studio 19でもaddress sanitizerは使用できるようなので、windowsアプリケーションの開発にはsanitizerを用いることなどを強く推奨する。

デバッガ

当然ながらデバッガは下記を選択するのが良いだろう。

- gcc/g++に対してはgdb
- clang/clang++に対してはlldb
- Visual Studioではそれに同梱されてるもの

組み込みソフトウェアに関しては少々事情が異なる。本ドキュメントでは、組み込みソフトウェアに関して、PC上のコンパイラを使用した単体テストを推奨している(「単体テストのサポートツール」参照)。これに従うのであれば、単体テストのコンパイル、デバッグには、通常のアプリケーション開発と同様に上記の3つの組み合わせのいずれかを使うのが良いだろう。

組み込みソフトウェアのターゲットのデバッグに関しては、gdb等のリモート機能を使える場合もあるが、多くの場合jtagデバッガを使う必要がある。専用のハードウェアが必要なため多少高価になるが、工数の方がよほど高価であることを忘れてはならない(ただし、半導体メーカー純正品よりもサードパーティ品の方が、数分の一の費用である場合があるので注意)。

printfデバッグ

printfデバッグとは、printf()やstd::coutを使ったデバッグ手法のことである。これ自体はそれなりに有効な手段であり、完全に否定はできないが、これ以外にデバッグ手法を知らないプログラマに出会うと、現在に蘇ったネアンデルタールに見えてしまう。ホモ・サピエンスの名に恥じないプログラマのために最新のデバッガを導入しよう。導入でロスした時間はデバッグの効率化でお釣りが来る。

エディタ/IDE

エディタ/IDEの良し悪しについて語ることには、宗教論争のような危険が伴う。「宗論はどちら負けても釈迦の恥」というように、不毛な戦いは避けるべきであるが、とはいっても各プログラマにその選択の全権を与えるのも問題である。

開発プロジェクトでは、下記の要件を満たすようなエディタ/IDEを使い、それらの設定ファイル等もバージョン管理システムで管理し、同一エディタ/IDEを使うメンバが共有できるようにするべきである。

以下にエディタ/IDEに求められる要件を上げる。

- 他の開発ツールとの連動
 - ビルドができる(「アーキテクチャとファイル構造」参照)。
 - デバッガを起動し、エディタ/IDEソースコード画面上でのデバッグができる。
 - バージョン管理システムとの連携ができる。
 - 単体テストを実行できる。
 - エディタ/IDE画面上でshell/ターミナルを使用できる。
 - OSの任意のコマンドを実行できる。
- 実行環境等
 - windows/linuxを含む複数のOSで動作する。
 - 動作が軽い。
 - プラグイン等の拡張機能をサポートしている。また、追加費用なしで使えるプラグインが豊富である。
 - 多言語対応、UTF-8対応している。
- プログラム編集
 - コード補完機能やタグジャンプができる。
 - ショートカットキー等が豊富にあり、習熟すれば効率化できる。また、マウスを使わなくても、ほぼすべての機能が使える。
 - 多プログラミング言語対応(カラーリング等)をしている。
 - UMLやマークダウン等の設計ドキュメントの編集もサポートしている。
- その他
 - 世界的に評価が高い、もしくは評価がある。
 - 盛んに開発が行われている。
 - ライセンス費がフリーか、安価である。
 - help機能、ネットドキュメント、書籍が充実している。
 - 設定ファイルの管理が簡単である。windows版のエディタ/IDEによっては、
 - 設定をレジストリに保存しエクスポート機能もないため、環境の再現や共有に時間がかかるものがある。
 - 設定ファイルから絶対パスの記述を排除できないものがある。このようなIDEを使用すると、作業ディレクトリが固定化されてしまう。

多くの要件を上げたが、このような条件をほぼ満たすエディタ/IDEは少なくない。例を上げる。

- Visual Studio

- Visual Studio Code
- Eclipse
- Sublime Text
- Atom
- vim/neovim
- emacs

少なく見積もってもこれだけの選択肢があるのであるから、間違っても日本国内でのみで流通しているようなローカルなエディタ/IDEを使ってはならない。

筆者の開発ツール環境

筆者は、windows、cygwin、linuxでソフトウェアを開発する必要があるため、一般的なプログラマと比べ、少々事情は異なると思うが、参考のためC++での開発ツールを紹介する。

- windowsアプリケーションの開発にはVisual Studioを使用している。
- Macでの本格的な開発経験がないので推奨できる開発ツールはないが、おそらくlinuxと同じで問題ない。
- linuxアプリケーションの開発には下記を使用している。
 - コンパイラ
 - gcc/g++もしくはclang/clang++
 - デバッガ
 - gdbもしくはlldb
 - エディタ/IDE
 - Visual Studio Code
 - vim/neovim
- 組み込みアプリケーションの開発では下記のようにしている。
 - 単体テスト(「単体テストのサポートツール」参照)に関してはlinuxと同じ。
 - ターゲットは様々であるためクロスコンパイラの推奨は難しいが、オプションや警告機能のレベルがgcc/g++と近いものを選ぶと良い。
 - デバッガも同様にこれと決まったものはないが、jtagデバッガは必須である。
- このドキュメントの作成にはコードも合わせて下記を使用している
 - g++/clang++
 - gdb
 - vim/neovim

vim/neovimの設定

参考のため、このドキュメントを作成するために筆者が使用しているvim/neovimの設定やプラグイン、その管理方法を紹介する。
vim/neovimの設定について興味がなければ、「費用」まで飛んでほしい。

設定ファイルと管理

このドキュメント作成時、cygwinパッケージにはneovimは存在せず、優れたコード補完機能はneovimでなければ動作しない。このドキュメントのコードはcygwin、linux両方でビルドでき、少なくともg++でのバイナリはそれぞれの環境で正常動作できるようにしなければならない。開発とは関係ないが、windows上での複雑なテキスト編集にはwindows版gvim(gui vim)を使用したい。また、「DRYの原則」の通り、それぞれの環境でそれぞれの設定をしたくない。

これから条件を満たすためのvim/neovimの設定ファイルは、

- cygwin上のvim
- windows上のgvim
- linux(実際にはWSL ubuntu)上のneovim

を正しく初期化できる必要がある。また、これらが別々に進化しないように一つのリポジトリで管理できるのが良い。筆者の場合は、クラウド上のgitでvim_configリポジトリとして管理している。

以下、順を追ってvim_configリポジトリとして管理しているファイルについて簡単に説明する。詳細については「vim」を参照してほしい。

ドキュメント

ファイル	機能
vim_config/README.md	リポジトリの説明、インストール手順
vim_config/nvim/cheatsheet.md	自分用ヘルプやtodo

初期化

ファイル	機能
<code>vim_config/nvim/init.vim</code>	nvim初期化
<code>vim_config/nvim/org.vim</code>	基本設定
<code>vim_config/nvim/package.vim</code>	外部パッケージ初期化
<code>vim_config/vim/gvimrc</code>	gvim初期化
<code>vim_config/vim/vimrc</code>	vim初期化
<code>vim_config/vim.sh</code>	エイリアス等の設定
<code>vim_config/inputrc</code>	gdb設定

独自スクリプト

c/c++編集

ファイル	機能
<code>vim_config/nvim/autoload/next_file.vim</code>	xxx.cpp xxx.h xxx_ut.cpp切り替え
<code>vim_config/nvim/plugin/next_file.vim</code>	同上
<code>vim_config/nvim/plugin/ctags_ext.vim</code>	ctags作成
<code>vim_config/nvim/plugin/dev_env.vim</code>	開発用画面設定
<code>vim_config/nvim/autoload/termdbg.vim</code>	gdb連動
<code>vim_config/nvim/plugin/termdbg.vim</code>	同上

bash起動

ファイル	機能
<code>vim_config/nvim/autoload/term.vim</code>	:terminal設定
<code>vim_config/nvim/plugin/term.vim</code>	同上

git運動

ファイル	機能
<code>vim_config/nvim/autoload/git_diff.vim</code>	git diffをvimdiffで表示
<code>vim_config/nvim/plugin/git_diff.vim</code>	同上
<code>vim_config/nvim/autoload/git_session.vim</code>	mksessionやpath設定
<code>vim_config/nvim/plugin/git_session.vim</code>	同上

ファイルタイプ設定

ファイル	機能
<code>vim_config/nvim/ftplugin/c.vim</code>	c/c++ファイルのインデント等
<code>vim_config/nvim/ftplugin/python.vim</code>	pythonファイルのインデント等
<code>vim_config/nvim/ftplugin/ruby.vim</code>	rubyファイルのインデント等

その他

ファイル	機能
<code>vim_config/nvim/autoload/buffers.vim</code>	バッファエクスプローラー
<code>vim_config/nvim/plugin/buffers.vim</code>	同上

ファイル	機能
<code>vim_config/nvim/autoload/grep.vim</code>	grep
<code>vim_config/nvim/plugin/grep.vim</code>	同上
<code>vim_config/nvim/autoload/cd.vim</code>	カレントディレクトリの変更
<code>vim_config/nvim/plugin/cd.vim</code>	同上
<code>vim_config/nvim/autoload/multi_hl.vim</code>	マルチハイライト
<code>vim_config/nvim/plugin/multi_hl.vim</code>	同上
<code>vim_config/nvim/plugin/clear_undo.vim</code>	undo履歴クリア
<code>vim_config/nvim/plugin/keybind.vim</code>	キーバインドの設定
<code>vim_config/nvim/plugin/scratch.vim</code>	スクラッチウインド
<code>vim_config/nvim/plugin/path_set.vim</code>	pathの追加
<code>vim_config/nvim/rplugin/python3/next_file.py</code>	pythonでのプラグイン例

外部パッケージ

パッケージ	機能
Shougo/dein.vim	外部パッケージ管理
roxma/nvim-yarp	補完
roxma/vim-hug-neovim-rpc	補完
Shougo/deoplete.nvim	補完
zchee/deoplete-clang	C++補完
Shougo/neoinclude.vim	補完
tpope/vim-fugitive	Git運動
reireias/vim-cheatsheet	自分用ヘルプ
mattn/vim-maketable	MDテーブル
aklt/plantuml-syntax	Plant Umlサポート

費用

コンパイラ、デバッガ、エディタ/IDEの費用は、ライセンス形態によっては総額が単価×プログラマの人数となるため、プロジェクトやそのプロジェクトを所有する事業体にとっては重荷になる。

それでも人件費に比べれば一桁以上安いはずなので、金を出し惜しんでプログラマの生産性を落としては本末転倒であるが、その金を管理する人々にはそのことが理解できないため、予算申請が却下されることは珍しくない。

そうならないためには、プログラマといえども会計の知識が必要である(とはいっても、基本的知識のみで十分である)。

一般にプロジェクトに必要なお金は2つに分けられる。

- 経費
- 固定資産の購入費

プロジェクトの経費には、

- 人件費
- 少額物品(10万円未満)の購入費
- 固定資産の償却費

等が含まれる。固定資産の償却費とは、長く使用するもの(通常は高額)を年割で費用化したものである。これだけでは、初見の人には何のことか全くわからないので例を上げる。

あるプロジェクトのために固定資産の購入費を使用し100万円のコンパイラを買ったとする。話を単純にするため、会社では定額償却とした場合(実際には定額と定率がある)、コンパイラの耐用年数は5年(税法で決まっている)であるため、100万円÷5年で、一年間20万円の費用が掛かると考える。これが固定資産の償却費である。最初に100万円払ったのにまた払うのかと思うかもしれないが、最初の100万円は

同じ価値の固定資産(コンパイラ)になっただけであるため、経費会計上、費用は掛かっていないと考えることになっているので矛盾はない。

実際にはコンパイラの代金100万円はこの会社からコンパイラ販売会社に支払われるため、会社が持っているキャッシュは100万円減る。が、上記したようにプロジェクトの初年度には20万円しか使っていないことになる(キャッシュが無くなれば会社は倒産するのだから、これが黒字倒産の仕組みである)。

このような仕組みがあるため、キャッシュが少ない会社では固定資産の申請は却下されやすい(キャッシュリッチな会社でも上場している場合は、固定資産の増加を嫌う傾向がある)。

一方で、このコンパイラをサブスクリプションで月額16,667円の契約した場合(5年で100万円)、この費用はプロジェクトの経費に組み込まれる。

ここまでツール購入の会計知識は十分である。

ここで考えてほしいのは、一括購入100万円とサブスク16,667円/月のどちらが予算申請を通りやすいのか、ということである。通常の会社では100万円の決済は課長レベルではできないが、16,667円/月の決済ならば、チームリーダーレベルでできるはずである。自分たちが使うツールはサブスクができないとの意見もあるだろうが、リース会社を通すことでほとんどのツールはサブスク化可能である(この場合少し費用が高くなるが、万事うまく行く案等そうそうない)。

費用レスで十分な機能をもつツールが多いのだから、なるべくそれらを使うべきだと思うが、どうしても必要な高額ツールの購入には、こういった方法を駆使してプログラマの生産性向上を図ってほしい。

deps

本ドキュメントでは、いくつかの場所でパッケージ間の依存関係の重要性について説明したため、これに従って開発を行うのであれば、依存関係の維持、監視が必要になる。

そのための市販のツールを購入することもできるが、やりたいことと完全にマッチしたものがあるわけではないため、このドキュメント専用のツールdepsを開発した。

このようなツールの開発にはpythonやrubyが適しているが、このドキュメントの目的に合わせて、depsは下記のようにC++で書かれている。

この章の構成

ディレクトリ、ファイル構成

depsの使い方

ユースケース-循環依存を発見した場合、非0でexitする

ユースケース-C++のソースコードを含むディレクトリを探す

ユースケース-ディレクトリをパッケージとみなして、パッケージとソースコードの関係を示す

ユースケース-パッケージ間の依存関係を示す

ユースケース-パッケージ間の依存関係を構造的に表す

ユースケース-パッケージ間の依存関係をplant umlで表す

ユースケース-ソースコード間の依存関係をplant umlで表す

ユースケース-パッケージでないディレクトリをそれとみなさない

makeによる依存関係の維持

ディレクトリ、ファイル構成

- app:main.cppを含むパッケージ
 - [example/deps/CMakeLists.txt](#) — メインのCMakeLists.txt
 - [example/deps/app/src/main.cpp](#) — depsのmain関数を含むファイル
 - [example/deps/app/src/deps_opts.cpp](#) — depsのオプション処理
 - [example/deps/app/src/deps_opts.h](#)
 - [example/deps/app/ut/deps_opts_ut.cpp](#) — appパッケージの単体テスト
- dependency:依存関係を導き出すアルゴリズムライブラリdependency.a用のパッケージ
 - [example/deps/dependency/CMakeLists.txt](#) — dependencyのCMakeLists.txt
 - [example/deps/dependency/src/arch_pkg.cpp](#) — パッケージの依存関係の導出
 - [example/deps/dependency/src/arch_pkg.h](#) — arch_pkg.cppの非公開ヘッダ
 - [example/deps/dependency/src/cpp_deps.cpp](#) — ファイル間依存関係の依存関係の導出
 - [example/deps/dependency/src/cpp_deps.h](#) — cpp_deps.cppの非公開ヘッダ
 - [example/deps/dependency/src/cpp_dir.cpp](#) — C++ファイルを含むディレクトリ抽出
 - [example/deps/dependency/src/cpp_dir.h](#) — cpp_dir.cppの非公開ヘッダ
 - [example/deps/dependency/src/cpp_src.cpp](#) — C++ファイルの抽出
 - [example/deps/dependency/src/cpp_src.h](#) — cpp_src.cppの非公開ヘッダ
 - [example/deps/dependency/h/dependency/deps_scenario.h](#) — 依存関係表示のシナリオの公開ヘッダ
 - [example/deps/dependency/src/deps_scenario.cpp](#) — 依存関係表示のユースケースシナリオ
 - [example/deps/dependency/src/load_store_format.cpp](#) — deps生成ファイルのロード/ストア
 - [example/deps/dependency/src/load_store_format.h](#) — load_store_format.cppの非公開ヘッダ
 - [example/deps/dependency/ut/arch_pkg_ut.cpp](#) — arch_pkg.cppの単体テスト
 - [example/deps/dependency/ut/cpp_deps_ut.cpp](#) — cpp_deps.cppの単体テスト
 - [example/deps/dependency/ut/cpp_dir_ut.cpp](#) — cpp_dir.cppの単体テスト
 - [example/deps/dependency/ut/cpp_src_ut.cpp](#) — cpp_src.cppの単体テスト
 - [example/deps/dependency/ut/deps_scenario_ut.cpp](#) — deps_scenario.cppの単体テスト
 - [example/deps/dependency/ut/load_store_format_ut.cpp](#) — load_store_format.cppの単体テスト
- file_utils:file_utils.a用のディレクトリ
 - [example/deps/file_utils/CMakeLists.txt](#) — file_utilsのCMakeLists.txt
 - [example/deps/file_utils/h/file_utils/load_store.h](#) — ファイルのロード/ストア
 - [example/deps/file_utils/h/file_utils/load_store_row.h](#) — load_store_row.cppのヘッダ
 - [example/deps/file_utils/h/file_utils/path_utils.h](#) — path_utils.cppのヘッダ

- [example/deps/file_utils/src/load_store_row.cpp](#) — ファイルのロード/ストア
- [example/deps/file_utils/src/path_utils.cpp](#) — ファイル操作
- [example/deps/file_utils/ut/load_store_ut.cpp](#) — load_store_row.cppの単体テスト
- [example/deps/file_utils/ut/path_utils_ut.cpp](#) — path_utils.cppの単体テスト
- lib: 全域からアクセス可能なテンプレートライブラリ
 - [example/deps/lib/CMakeLists.txt](#) — libのCMakeLists.txt
 - [example/deps/lib/h/lib/nstd.h](#) — テンプレートライブラリ
 - [example/deps/lib/ut/nstd_ut.cpp](#) — nstd.hの単体テスト
- logging: logging.a用のディレクトリ
 - [example/deps/logging/CMakeLists.txt](#) — loggingのCMakeLists.txt
 - [example/deps/logging/h/logging/logger.h](#) — logger.cppのヘッダ
 - [example/deps/logging/src/logger.cpp](#) — ログの取得
 - [example/deps/logging/ut/logger_ut.cpp](#) — logger.cppの単体テスト

下記のをファイルツリーは上記を表す。

```

deps
├── makefile           # makeでもビルドできる
├── CMakeLists.txt     # cmakeのルートCMakeLists.txt
├── app
│   ├── CMakeLists.txt
│   ├── src
│   │   ├── deps_opts.cpp
│   │   ├── deps_opts.h
│   │   └── main.cpp
│   └── ut
│       └── deps_opts_ut.cpp    # utはsrcにアクセスできる
└── dependency
    ├── CMakeLists.txt
    ├── h
    │   └── dependency        # このディレクトリにエクスポートするヘッダを配置
    │       └── deps_scenario.h
    └── src
        ├── arch_pkg.cpp
        ├── arch_pkg.h
        ├── cpp_deps.cpp
        ├── cpp_deps.h
        ├── cpp_dir.cpp
        ├── cpp_dir.h
        ├── cpp_src.cpp
        ├── cpp_src.h
        ├── deps_scenario.cpp
        ├── load_store_format.cpp
        └── load_store_format.h
            └── ut              # utはh、srcにアクセスできる
                ├── arch_pkg_ut.cpp
                ├── cpp_deps_ut.cpp
                ├── cpp_dir_ut.cpp
                ├── cpp_src_ut.cpp
                ├── deps_scenario_ut.cpp
                └── load_store_format_ut.cpp
    └── file_utils
        ├── CMakeLists.txt
        ├── h
        │   └── file_utils      # このディレクトリにエクスポートするヘッダを配置
        │       ├── load_store.h
        │       ├── load_store_row.h
        │       └── path_utils.h
        ├── src
        │   ├── load_store_row.cpp
        │   └── path_utils.cpp
        └── ut
            ├── load_store_row_ut.cpp
            └── path_utils_ut.cpp
    └── lib
        ├── CMakeLists.txt
        ├── h
        │   └── lib             # このディレクトリにエクスポートするヘッダを配置
        │       └── nstd.h
        └── ut
            └── nstd_ut.cpp    # utはh、srcにアクセスできる
└── logging
    ├── CMakeLists.txt
    ├── h
    │   └── logging         # このディレクトリにエクスポートするヘッダを配置
    └── ut

```

```

|   └── logger.h
+-- src
|   └── logger.cpp
+-- ut
    └── logger_ut.cpp

```

utはh、srcにアクセスできる

例えば、dependencyの外部公開ヘッダを配置するためのディレクトリ

dependency/h/dependency

は助長に見える。コンパイラオプションのインクルードパスにdependency/h指定することにより、dependencyをインポートするソースコードのインクルードディレクトリは下記のように記述することになる。

```

// @@@ exampledeps/dependency/src/deps_scenario.cpp 7

#include "cpp_deps.h"           // 実装用ヘッダファイル
#include "cpp_dir.h"             // 実装用ヘッダファイル
#include "cpp_src.h"             // 実装用ヘッダファイル
#include "dependency/deps_scenario.h" // dependencyパッケージからのインポート
#include "file_utils/load_store.h" // file_utilsパッケージからのインポート
#include "lib/nstd.h"             // libパッケージからのインポート

```

上記から明らかな通り、このソースコードの外部パッケージとの依存関係が明確になる。このようなインクルードディレクトリを下記のように指定することでこのような記述が可能になる。

```

// @@@ exampledeps/dependency/CMakeLists.txt 19

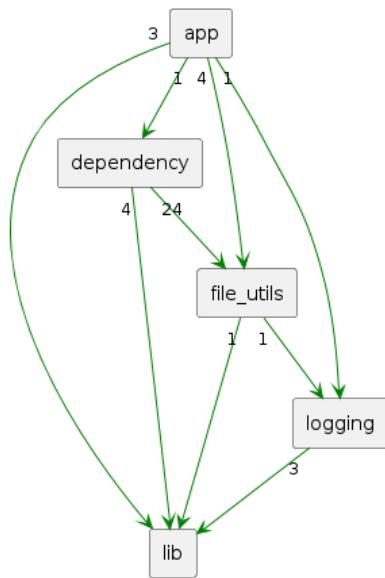
# dependency.aをリンクするファイルに
# ..../dependency/h ../file_utils/h ../lib/h
# のヘッダファイルを公開する

target_include_directories(dependency PUBLIC ..../dependency/h ..../file_utils/h ..../lib/h)

```

CMakeの公式ガイドラインや一般的な慣習に沿ったこの構造とインクルードディレクトリの記述様式は、プロジェクトの可読性と保守性を向上させるために推奨される方法である。冗長に見えるディレクトリ名も、プロジェクト全体の理解を容易にするために有効である。

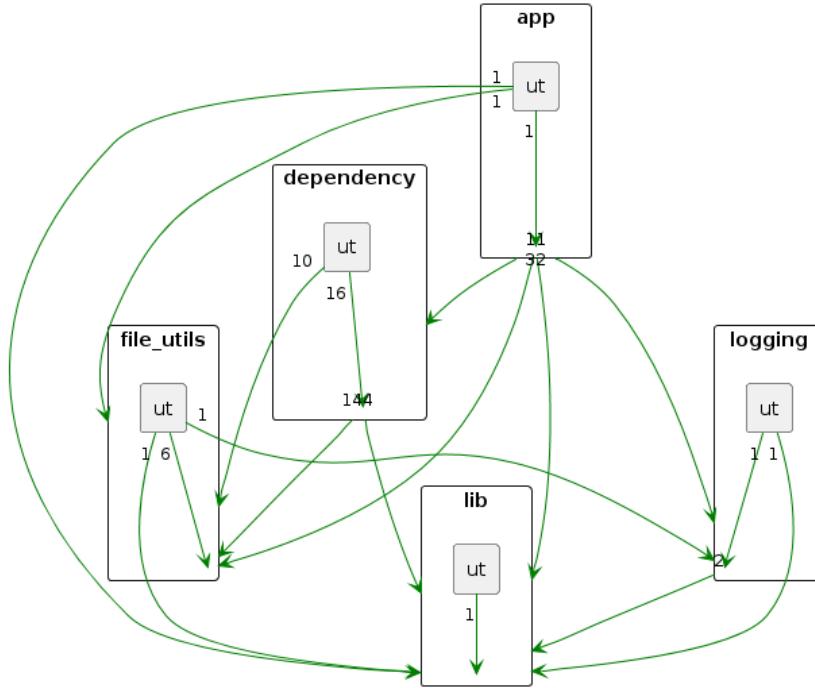
depsの各パッケージの依存関係は、



のようになっている。

当然ながら、「パッケージとその構成ファイル」で述べた構造と相似である。

なお、utディレクトリを各パッケージ内のサブパッケージとした場合の依存関係は、



のようになっており、整理された依存関係であるといえる。

パッケージとその単体テスト用ソースコードへのヘッダファイル公開は、必要以上に公開範囲を広げないようにするために下記のように行われる。

```
// @@@ exampledeps/dependency/CMakeLists.txt 38

# dependency_ut_exeはdependency.aの単体テスト
# dependency_ut_exeが使用するライブラリのヘッダは下記の記述で公開される
target_link_libraries(dependency_ut_exe dependency file_utils logging gtest gtest_main)

# dependency_ut_exeに上記では公開範囲が不十分である場合、
# dependency_ut_exeが使用するライブラリのヘッダは下記の記述で限定的に公開される
# dependency_ut_exeにはdependency/src/*.hへのアクセスが必要
target_include_directories(dependency_ut_exe PRIVATE ../../deep/h src)
```

ソースコードの構成をdepsのようになりますことを推奨する。

depsの使い方

depsのコマンドオプションを以下に示す。

```
deps CMD [option] [DIRS] ...
CMD:
  p   : generate package to OUT.
  s   : generate srcs with incs to OUT.
  p2s : generate package and srcs pairs to OUT.
  p2p : generate packages' dependencies to OUT.
  a   : generate structure to OUT from p2p output.
  a2pu: generate plant uml package to OUT from p2p output.
  cyc : exit !0 if found cyclic dependencies.
  help : show help message.
  h   : same as help(-h, --help).

options:
  --in IN      : use IN to execute CMD.
  --out OUT    : CMD outputs to OUT.
  --recursive : search dir as package from DIRS or IN contents.
  -R          : same as --recursive.
  --src_as_pkg: every src is as a package.
  -s          : same as --src_as_pkg.
  --log LOG   : loggin to LOG(if LOG is "-", using STDOUT).
  --exclude PTN : exclude dirs which matchs to PTN(JS regex).
  -e PTN     : same as --exclude.

DIRS: use DIRS to execute CMD.
IN  : 1st line in this file must be
```

```
#dir2srcs for pkg-srcs file  
or  
#dir for pkg file.
```

各ユースケース

- ユースケース-循環依存を発見した場合、非0でexitする
- ユースケース-C++のソースコードを含むディレクトリを探す
- ユースケース-ディレクトリをパッケージとみなして、パッケージとソースコードの関係を示す
- ユースケース-パッケージ間の依存関係を示す
- ユースケース-パッケージ間の依存関係を構造的に表す
- ユースケース-パッケージ間の依存関係をplant umlで表す
- ユースケース-ソースコード間の依存関係をplant umlで表す
- ユースケース-パッケージでないディレクトリをそれとみなさない

におけるdepsの使い方や出力等を示す。

ユースケース-循環依存を発見した場合、非0でexitする

このツールの主な目的は、パッケージ間の循環依存を検出することである。このユースケースは、これを実現する方法を提示する。

ソースコードを含むディレクトリut_data/で下記のようにすれば、ディレクトリをパッケージとみなした依存関係が循環した場合、depsは非0でexitする。

```
> ./g++/deps p2p -R -s --out p2p.txt ut_data/  
> ./g++/deps cyc --in p2p.txt
```

CIのチェック項目(「CI(継続的インテグレーション)」参照)に上記を導入することで、循環の無い依存関係を維持することができる。

下記に、「ディレクトリが必ずしもパッケージに対応するわけではない」場合の対処法を掲載する。

ユースケース-C++のソースコードを含むディレクトリを探す

以下のコマンドは、CMD pによりut_data/配下のソースコードを含むディレクトリを探す。

```
> ./g++/deps p -R ut_data/
```

アウトプットは以下のようになる。

```
#dir  
ut_data/app1  
ut_data/app1/mod1  
ut_data/app1/mod2  
ut_data/app1/mod2/mod2_1  
ut_data/app1/mod2/mod2_2  
ut_data/app2
```

-out OUT-FILEを指定すれば、上記出力はOUT-FILEに書き出される。OUT-FILEを適切に編集し、他のCMDの入力(-in IN-FILE)に使用することもできる。

ユースケース-ディレクトリをパッケージとみなして、パッケージとソースコードの関係を示す

以下のコマンドは、CMD p2sによりut_data配下のディレクトリをパッケージとみなして、パッケージとソースコードの関係を出力する。

```
> ./g++/deps p2s -R ut_data/
```

アウトプットは以下のようになる。

```
#dir2srcs  
ut_data/app1  
    ut_data/app1/a_1_c.c  
    ut_data/app1/a_1_c.h  
    ut_data/app1/a_1_cpp.cpp  
    ut_data/app1/a_1_cpp.h  
  
... 中略 ...  
  
ut_data/app1/mod2/mod2_2  
    ut_data/app1/mod2/mod2_2/mod2_2_1.cpp  
    ut_data/app1/mod2/mod2_2/mod2_2_1.h  
  
ut_data/app2
```

```
ut_data/app2/b_1.cpp  
ut_data/app2/b_1.h
```

ユースケース-パッケージ間の依存関係を示す

以下のコマンドは、CMD p2pによりdeps/ut_data配下のパッケージとの依存関係をp2p.txtに出力する。

```
> cd ./deps  
> ./g++/deps p2p -R --out p2p.txt ut_data/
```

アウトプットは以下のようになる。

```
#deps  
ut_data/app1 -> ut_data/app1/mod1 : 2 ut_data/app1/mod1/mod1_1.hpp ut_data/app1/mod1/mod1_2.hpp  
ut_data/app1/mod1 -> ut_data/app1 : 0  
  
ut_data/app1 -> ut_data/app1/mod2 : 0  
ut_data/app1/mod2 -> ut_data/app1 : 0  
  
... 中略 ...  
  
ut_data/app1/mod2 -> ut_data/app2 : 0  
ut_data/app2 -> ut_data/app1/mod2 : 0  
  
ut_data/app1/mod2/mod2_1 -> ut_data/app1/mod2/mod2_2 : 1 ut_data/app1/mod2/mod2_2/mod2_2_1.h  
ut_data/app1/mod2/mod2_2 -> ut_data/app1/mod2/mod2_1 : 2 ut_data/app1/mod2/mod2_1/mod2_1_1.h  
  
ut_data/app1/mod2/mod2_1 -> ut_data/app2 : 0  
ut_data/app2 -> ut_data/app1/mod2/mod2_1 : 0  
  
ut_data/app1/mod2/mod2_2 -> ut_data/app2 : 0  
ut_data/app2 -> ut_data/app1/mod2/mod2_2 : 0
```

例を上げて、上記の意味を説明する。

[例]

```
ut_data/app1/mod2/mod2_1 -> ut_data/app1/mod2/mod2_2 : 1 ut_data/app1/mod2/mod2_2/mod2_2_1.h  
ut_data/app1/mod2/mod2_2 -> ut_data/app1/mod2/mod2_1 : 2 ut_data/app1/mod2/mod2_1/mod2_1_1.h
```

[例の意味]

- ut_data/app1/mod2/mod2_1からut_data/app1/mod2/mod2_2への依存ファイルはut_data/app1/mod2/mod2_2/mod2_2_1.hで、1箇所includeされている。
- ut_data/app1/mod2/mod2_2からut_data/app1/mod2/mod2_1の依存ファイルはut_data/app1/mod2/mod2_1/mod2_1_1.hで、2箇所includeされている。

ユースケース-パッケージ間の依存関係を構造的に表す

以下のコマンドは、CMD aにより上記p2p.txtを構造的に出力する。

```
> ./g++/deps a --in p2p.txt
```

アウトプットは以下のようになる。

```
#arch  
package :app1:CYCLIC  
parent :TOP  
depend_on: {  
    mod1 : CYCLIC  
}  
children : {  
    package :mod1:CYCLIC  
    parent :app1  
    depend_on: {  
        mod2 : STRAIGHT  
        mod2_2 : CYCLIC  
    }  
    children : {}  
  
    package :mod2  
    parent :app1  
    depend_on: {}  
    children : {  
        package :mod2_1:CYCLIC
```

```

parent :mod2
depend_on: {
    mod2_2 : CYCLIC
}
children : { }

package :mod2_2:CYCLIC
parent :mod2
depend_on: {
    app1 : CYCLIC
    mod2_1 : CYCLIC
}
children : { }
}

package :app2
parent :TOP
depend_on: {
    app1 : STRAIGHT
    mod1 : STRAIGHT
}
children : { }

```

ディレクトリ名の後の

- STRAIGHTは依存関係が循環していない
- CYCLICは依存関係が循環している

ことを示している。

ユースケース-パッケージ間の依存関係をplant umlで表す

以下のコマンドは、CMD a2puにより上記p2p.txtをplant uml形式で出力する。

```
> ./g++/deps a2pu --in p2p.txt
```

アウトプットは以下のようになる。

```

@startuml
scale max 730 width
rectangle "app1" as ut_data__app1 {
    rectangle "mod1" as ut_data__app1__mod1
    rectangle "mod2" as ut_data__app1__mod2 {
        rectangle "mod2_1" as ut_data__app1__mod2__mod2_1
        rectangle "mod2_2" as ut_data__app1__mod2__mod2_2
    }
}
rectangle "app2" as ut_data__app2

ut_data__app1 "6" <-[#red]-> "1" ut_data__app1__mod1
ut_data__app1 "3" <-[#red]-> "1" ut_data__app1__mod2__mod2_1
ut_data__app1 "3" <-[#red]-> "2" ut_data__app1__mod2__mod2_2
ut_data__app2 "3" -[#green]-> ut_data__app1
ut_data__app1__mod1 "1" -[#green]-> ut_data__app1__mod2
ut_data__app1__mod1 "1" <-[#red]-> "2" ut_data__app1__mod2__mod2_1
ut_data__app1__mod1 "1" <-[#red]-> "4" ut_data__app1__mod2__mod2_2
ut_data__app2 "4" -[#green]-> ut_data__app1__mod1
ut_data__app1__mod2__mod2_1 "1" <-[#red]-> "2" ut_data__app1__mod2__mod2_2
ut_data__app2 "2" -[#green]-> ut_data__app1__mod2__mod2_1
ut_data__app2 "2" -[#green]-> ut_data__app1__mod2__mod2_2

@enduml

```

このアウトプットを[plant umlオンラインジェネレータ](#)のテキストボックスに貼り付ければpngファイルが得られ、視覚的に依存関係を把握できる。

ユースケース-ソースコード間の依存関係をplant umlで表す

ソースコードをパッケージとみなすオプション(-sもしくは--src_as_pkg)を付加して、これまでの説明と同様のことを行うと、

```
> ./g++/deps p2p -s --out p2p.txt ut_data/
> ./g++/deps a2pu --in p2p.txt --out p2p.pu
```

ソースコードの依存関係がplant uml形式で得られる。

ユースケース-パッケージでないディレクトリをそれとみなさない

depsのソースコードを含むdependency/hは、 dependencyパッケージのインターフェースを外部公開するためのものであり、 dependencyのサブパッケージではない。このような場合、 dependency/h/deps_scenario.hのようなファイルは、 dependencyに直接属するように扱うべきであるが、 このツールがファイル構造からそれを読み解くことは不可能である。

このような場合に対処する方法は下記の3通りある。

- -Rを指定せず、 パッケージとなる全ディレクトリをINに記述する。
- pコマンドで候補ディレクトリを全てファイルに出力し、 パッケージでないディレクトリをそのファイルから削除した後、 そのファイルをINファイルとしてp2pコマンド等を使う。
- --exclude PTNでパッケージ対象でないディレクトリを指定しp2pコマンド等を使う。 なお、 例えばhディレクトリを排除する場合のPTNの指定は下記のようになる(PTNはC++の正規表現)。

```
--exclude \".*\/*\/*"
```

makeによる依存関係の維持

ビルドツールにmake、 コンパイラにg++やclang++を使うのであれば、

- 下記のMakefileのように、 コンパイラに指定するインクルードパスを制限し、 パッケージごとにライブラリを作る

```
// @@@ example/deps/Makefile 85
### logging
INC_LOGGER=-Ilib/h -Ilogging/h # インクルードパスの指定

# 指定されたインクルードパスを使用したコンパイル
$(O)logging/src/%.o : logging/src/%.cpp
    $(CXX) $(INC_LOGGER) $(SANITIZER_OPT) $(CCFLAGS) -c -o $@ $<

# 指定されたインクルードパスを使用したUTのコンパイル
$(O)logging/ut/%.o : logging/ut/%.cpp
    $(CXX) $(INC_UT) $(INC_LOGGER) $(SANITIZER_OPT) $(CCFLAGS) -c -o $@ $<

# ライブラリの生成
$(LOGGER_A) : $(LOGGER_OBJS)
    ar cr $@ $^

# UT実行バイナリの生成
$(O)logging_ut : $(LOGGER_UT_OBJS) $(FILE_UTILS_A) $(LOGGER_A) $(DEPENDENCY_A) $(GTEST_LIB)
    $(CXX) -o $@ $^ -lpthread -lstdc++fs $(SANITIZER_OPT)

### file_utils
INC_FILE_UTILS:=-Ilib/h -Ilogging/h -Ifile_utils/h # インクルードパスの指定

# 指定されたインクルードパスを使用したコンパイル
$(O)file_utils/src/%.o : file_utils/src/%.cpp
    $(CXX) $(INC_FILE_UTILS) $(SANITIZER_OPT) $(CCFLAGS) -c -o $@ $<

# 指定されたインクルードパスを使用したUTのコンパイル
$(O)file_utils/ut/%.o : file_utils/ut/%.cpp
    $(CXX) $(INC_UT) $(INC_FILE_UTILS) $(SANITIZER_OPT) $(CCFLAGS) -c -o $@ $<

# ライブラリの生成
$(FILE_UTILS_A) : $(FILE_UTILS_OBJS)
    ar cr $@ $^

# UT実行バイナリの生成
$(O)file_utils_ut : $(FILE_UTILS_UT_OBJS) $(FILE_UTILS_A) $(LOGGER_A) $(GTEST_LIB)
    $(CXX) -o $@ $^ -lpthread -lstdc++fs $(SANITIZER_OPT)
```

- #includeディレクトイブでのパスに上方向のディレクトリ指定("../")を使わない(「[#includeで指定するパス名](#)」参照)

とすることで、 ビルド時に循環依存を作らないことを担保することができる(「[アーキテクチャの設計](#)」参照)。

CMakeやVisual Studioを含むほとんどのビルドツールでも同様のことは可能である(逆に言えば、 このようなことができないビルドツールを使うべきではない)。

用語解説

この章では、このドキュメントで使用する用語の解説をする。

この章の構成

型とインスタンス

算術型

汎整数型

整数型

算術変換

汎整数拡張

POD

標準レイアウト型

トリビアル型

`underlying_type`

不完全型

完全型

ポリモーフィックなクラス

インターフェースクラス

`const`インスタンス

`constexpr`インスタンスと関数

ユーザ定義リテラル演算子

`std::string`型リテラル

オブジェクトと生成

特殊メンバ関数

初期化子リストコンストラクタ

継承コンストラクタ

委譲コンストラクタ

非`explicit`なコンストラクタによる暗黙の型変換

NSDMI

二様初期化

AAAスタイル

オブジェクトの所有権

オブジェクトのライフタイム

クラスのレイアウト

オブジェクトのコピー

シャローコピー

ディープコピー

スライシング

name lookupと名前空間

ルックアップ

`name lookup`

`two_phase_name_lookup`

実引数依存探索

ADL

関連名前空間

SFINAE

`name-hiding`

ドミナンス

ダイヤモンド継承

仮想継承

仮想基底

`using`宣言

`using`ディレクティブ

expressionと値カテゴリ

expression
lvalue
rvalue
rvalue修飾
lvalue修飾
リファレンス修飾
decltype

リファレンス

ユニバーサルリファレンス
forwardingリファレンス
perfect forwarding
リファレンスcollapsing
danglingリファレンス
danglingポインタ

エクセプション安全性の保証

no-fail保証
強い保証
基本保証

シンタックス、セマンティクス

等価性のセマンティクス
copyセマンティクス
moveセマンティクス

C++コンパイラ

g++
clang++

C++その他

オーバーライドとオーバーロードの違い
実引数/仮引数
範囲for文
ラムダ式
ジェネリックラムダ
関数tryブロック
単純代入
ill-formed
well-formed
one-definition rule
ODR
RVO(Return Value Optimization)
SSO(Small String Optimization)
Most Vexing Parse
RTTI
Run-time Type Information
simple-declaration
typeid
トライグラフ
フリースタンディング環境

ソフトウェア一般

凝集度
サイクロマティック複雑度
Spurious Wakeup
副作用
is-a
has-a
is-implemented-in-terms-of

非ソフトウェア用語

割れ窓理論
車輪の再発明

型とインスタンス

算術型

算術型とは下記の型の総称である。

- 汎整数型(bool, char, int, unsigned int, long long等)
- 浮動小数点型(float、double、long double)

算術型のサイズは下記のように規定されている。

- 1 == sizeof(bool) == sizeof(char)
- sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)
- 4 <= sizeof(long)
- 8 <= sizeof(long long)
- 4 == sizeof(float)
- 8 == sizeof(double) <= sizeof(long double)

汎整数型

汎整数型とは下記の型の総称である。

- 論理型(bool)
- 文字型(char、wchar_t等)
- 整数型(int、unsigned int、long等)

整数型

整数型とは下記の型の総称である。

- char
- signed char
- unsigned char
- short
- unsigned short
- int
- unsigned int
- long
- unsigned long
- long long
- unsigned long long

算術変換

C++における算術変換とは、算術演算の1つのオペランドが他のオペランドと同じ型でない場合、1つのオペランドを他のオペランドと同じ型に変換するプロセスのことを目指す。

算術変換は、汎整数拡張と通常算術変換に分けられる。

```
// @@@ example/term_explanation/integral_promotion_ut.cpp 11

bool      bval{};
char      cval{};
short     sval{};
unsigned short usval{};
int       ival{};
unsigned int  uival{};
long      lval{};
unsigned long ulval{};
float     fval{};
double    dval{};

auto ret_0 = 3.14159 + 'a'; // 'a'は汎整数拡張でintになった後、さらに通常算術変換でdoubleに
static_assert(std::is_same<decltype(ret_0), double>::value, "");

auto ret_1 = dval + ival; // ivalは通常算術変換でdoubleに
static_assert(std::is_same<decltype(ret_1), double>::value, "");

auto ret_2 = dval + fval; // fvalは通常算術変換でdoubleに
static_assert(std::is_same<decltype(ret_2), double>::value, "");

auto ret_3 = ival = dval; // dvalは通常算術変換でintに
static_assert(std::is_same<decltype(ret_3), int>::value, "")
```

```

static_assert(std::is_same<decltype(ret_3), int>::value, "");

bval = dval; // dvalは通常算術変換でboolに
ASSERT_FALSE(bval);

auto ret_4 = cval + fval; // cvalは汎整数拡張でintになった後、さらに通常算術変換でfloatに
static_assert(std::is_same<decltype(ret_4), float>::value, "");

auto ret_5 = sval + cval; // svalとcvalは汎整数拡張でintに
static_assert(std::is_same<decltype(ret_5), int>::value, "");

auto ret_6 = cval + lval; // cvalは汎整数拡張でintになった後、通常算術変換でlongに
static_assert(std::is_same<decltype(ret_6), long>::value, "");

auto ret_7 = ival + ulval; // ivalは通常算術変換でunsigned longに
static_assert(std::is_same<decltype(ret_7), unsigned long>::value, "");

auto ret_8 = usval + ival; // usvalは汎整数拡張でintに
// ただし、この変換はunsigned shortとintのサイズに依存する
static_assert(std::is_same<decltype(ret_8), int>::value, "");

auto ret_9 = uival + lval; // uivalは通常算術変換でlongに
// ただし、この変換はunsigned intとlongのサイズに依存する
static_assert(std::is_same<decltype(ret_9), long>::value, "");

```

一様初期を使用することで、変数定義時の算術変換による意図しない値の変換(縮小型変換)を防ぐことができる。

```

// @@@ example/term_explanation/integral_promotion_ut.cpp 62

int i{-1};
// int8_t i8 {i}; 縮小型変換によりコンパイル不可
int8_t i8 = i; // intからint8_tへの型変換
// これには問題ないが

ASSERT_EQ(-1, i8);

// uint8_t ui8 {i}; 縮小型変換によりコンパイル不可
uint8_t ui8 = i; // intからuint8_tへの型変換
// おそらく意図通りではない

ASSERT_EQ(255, ui8);

```

以下に示すように、算術変換の結果は直感に反することがあるため、注意が必要である。

```

// @@@ example/term_explanation/integral_promotion_ut.cpp 81

int          i{-1};
unsigned int ui{1};

// ASSERT_TRUE(i < ui);
ASSERT_TRUE(i > ui); // 算術変換の影響で、-1 < 1が成立しない

signed short  s{-1};
unsigned short us{1};

ASSERT_TRUE(s < us); // 汎整数拡張により、-1 < 1が成立

```

汎整数拡張

bool、char、signed char、unsigned char、short、unsigned short型の変数が、算術のオペランドとして使用される場合、

- その変数の型の取り得る値全てがintで表現できるのならば、int型に変換される。
- そうでなければ、その変数はunsigned int型に変換される。

この変換を汎整数拡張と呼ぶ。

従って、`sizof(short) < sizeof(int)`である処理系では、bool、char、signed char、unsigned char、short、unsigned short型の変数は、下記のようにintに変換される。

```

// @@@ example/term_explanation/integral_promotion_ut.cpp 100

bool bval;
static_assert(std::is_same<int, decltype(bval + bval)>::value, "");

char cval;
static_assert(std::is_same<int, decltype(cval + cval)>::value, "");

unsigned char ucval = 128;

```

```

static_assert(std::is_same<int, decltype(ucval + ucval)>::value, "");
ASSERT_EQ(256, ucval + ucval); // 況整数拡張により256になる

static_assert(std::is_same<int, decltype(cval + ucval)>::value, "");

short sval;
static_assert(std::is_same<int, decltype(sval + sval)>::value, "");

unsigned short usval;
static_assert(std::is_same<int, decltype(usval + usval)>::value, "");

static_assert(std::is_same<int, decltype(sval + usval)>::value, "");

```

POD

PODとは、 Plain Old Dataの略語であり、

```
std::is_pod<T>::value
```

がtrueとなる型Tを指す。下記のコードはその使用例である。

```

// @@@ example/term_explanation/pod_ut.cpp 7

static_assert(std::is_pod<int>::value, "");
static_assert(std::is_pod<int const>::value, "");
static_assert(std::is_pod<int*>::value, "");
static_assert(std::is_pod<int[3]>::value, "");
static_assert(!std::is_pod<int&>::value, ""); // リファレンスはPODではない

struct Pod {};

static_assert(std::is_pod<Pod>::value, "");
static_assert(std::is_pod<Pod const>::value, "");
static_assert(std::is_pod<Pod*>::value, "");
static_assert(std::is_pod<Pod[3]>::value, "");
static_assert(!std::is_pod<Pod&>::value, "");

struct NonPod { // コンストラクタがあるためPODではない
    NonPod();
};

static_assert(!std::is_pod<NonPod>::value, "");

```

概ね、C言語と互換性のある型を指すと思って良い。

「型がトリビアル型且つ標準レイアウト型であること」と「型がPODであること」は等価であるため、C++20では、PODという用語は非推奨となった。

標準レイアウト型

標準レイアウト型とは、

```
std::is_standard_layout<T>::value
```

がtrueとなる型Tを指す。下記のコードはその使用例である。

```

// @@@ example/term_explanation/pod_ut.cpp 31

static_assert(std::is_standard_layout<int>::value, "");
static_assert(std::is_standard_layout<int*>::value, "");
static_assert(std::is_standard_layout<int[1]>::value, "");
static_assert(!std::is_standard_layout<int&>::value, "");

enum class SizeUndefined { su_0, su_1 };

struct StandardLayout { // 標準レイアウトだがトリビアルではない
    StandardLayout() : a{0}, b{SizeUndefined::su_0} {}
    int a;
    SizeUndefined b;
};

static_assert(std::is_standard_layout<StandardLayout>::value, "");
static_assert(!std::is_trivial<StandardLayout>::value, "");
static_assert(!std::is_pod<StandardLayout>::value, "");

```

型がPODである場合、その型は標準レイアウト型である。

トリビアル型

トリビアル型とは、

```
std::is_trivial<T>::value
```

がtrueとなる型Tを指す。下記のコードはその使用例である。

```
// @@@ example/term_explanation/pod_ut.cpp 52

static_assert(std::is_trivial<int>::value, "");
static_assert(std::is_trivial<int*>::value, "");
static_assert(std::is_trivial<int[1]>::value, "");
static_assert(!std::is_trivial<int&>::value, "");

enum class SizeUndefined { su_0, su_1 };

struct Trivial {      // トリビアルだが標準レイアウトではない
    int&           a; // リファレンスは標準レイアウトではない
    SizeUndefined b;
};

static_assert(!std::is_standard_layout<Trivial>::value, "");
static_assert(std::is_trivial<Trivial>::value, "");
static_assert(!std::is_pod<Trivial>::value, "");
```

型がPODである場合、その型はトリビアル型である。

underlying type

underlying typeとは、enumやenum classの汎整数表現を指定できるようにするために、C++11で導入されたシンタックスである。

```
// @@@ example/term_explanation/underlying_type_ut.cpp 9

// 従来のenum
enum NormalEnum {
    ...
};

// enum underlying typeがint8_tに指定された従来のenum
enum NormalEnumWithUnderlyingType : int8_t {
    ...
};

// enum class
enum class EnumClass {
    ...
};

// enum underlying typeがint64_tに指定されたenum class
enum class EnumClassWithUnderlyingType : int64_t {
    ...
};

// @@@ example/term_explanation/underlying_type_ut.cpp 38

ASSERT_EQ(4, sizeof(NormalEnum)); // 列挙子の値を表現するのに十分なサイズの整数型で処理系依存

// NormalEnumWithUnderlyingTypeのunderlying typeはint8_t
static_assert(std::is_same_v<int8_t, std::underlying_type_t<NormalEnumWithUnderlyingType>>);
ASSERT_EQ(sizeof(int8_t), sizeof(NormalEnumWithUnderlyingType));

ASSERT_EQ(4, sizeof(EnumClass)); // 列挙子の値を表現するのに十分なサイズの整数型で処理系依存

// EnumClassWithUnderlyingTypeのunderlying typeはint64_t
static_assert(std::is_same_v<int64_t, std::underlying_type_t<EnumClassWithUnderlyingType>>);
ASSERT_EQ(sizeof(int64_t), sizeof(EnumClassWithUnderlyingType));
```

不完全型

不完全型とは、型のサイズや構造が不明な型を指す。下記のような場合、不完全型となる。

```
// @@@ example/term_explanation/incomplete_type_ut.cpp 6

class A; // Aの前方宣言
// これ以降、Aは不完全型となる
```

```
// auto a = sizeof(A); Aが不完全型であるため、コンパイルエラー

class A { // この宣言により、この行以降はAは完全型になる
public:
    // 何らかの宣言
};

auto a = sizeof(A); // Aが完全型であるため、コンパイル可能
```

完全型

不完全型ではない型を指す。

ポリモーフィックなクラス

ポリモーフィックなクラスとは、仮想関数を持つクラスを指す。なお、純粋仮想関数を持つクラスは、仮想クラスと呼ばれる。

インターフェースクラス

インターフェースクラスとは、純粋仮想関数のみを持つ抽象クラスのことを指す。インターフェースクラスは、クラスの実装を提供することなく、クラスのインターフェースを定義するために使用される。インターフェースクラスは、クラスの仕様を定義するために使用されるため、多くの場合、抽象基底クラスとして使用される。

```
// @@@ example/term_explanation/interface_class.cpp 8

class InterfaceClass { // インターフェースクラス
public:
    virtual void DoSomething(int32_t) = 0;
    virtual bool IsXxx() const      = 0;
    virtual ~InterfaceClass()       = 0;
};

class NotInterfaceClass { // メンバ変数があるためインターフェースクラスではない
public:
    NotInterfaceClass();
    virtual void DoSomething(int32_t) = 0;
    virtual bool IsXxx() const      = 0;
    virtual ~NotInterfaceClass()     = 0;
};

private:
    int32_t num_;
};
```

constインスタンス

constインスタンスとはランタイムの初期化時に値が確定し、その後、状態が不变であるインスタンスである。

constexprインスタンスと関数

constexprインスタンスとはコンパイル時に値が確定するインスタンスである。当然、ランタイム時でも不变である。

```
// @@@ example/term_explanation/constexpr_ut.cpp 6

constexpr double PI{3.14159265358979323846}; // PIはconstexpr
```

constexprとして宣言された関数の戻り値がコンパイル時に確定する場合、その関数の呼び出し式はconstexprと扱われる。従って、この値はテンプレートパラメータやstatic_assertのオペランドに使用することができる。

```
// @@@ example/term_explanation/constexpr_ut.cpp 10

constexpr int f(int a) noexcept { return a * 3; } // aがconstexprならばf(a)もconstexpr

int g(int a) noexcept { return a * 3; } // aがconstexprであってもg(a)は非constexpr

template <int N>
struct Templ {
    static constexpr auto value = N;
};

// @@@ example/term_explanation/constexpr_ut.cpp 25

auto x = int{0};
```

```

constexpr auto a = f(3); // f(3)はconstexprなのでaの初期化が可能
// constexpr auto b = f(x); // xは非constexprなのでbの初期化はコンパイルエラー
auto const c = f(3); // cはconstexprとすべき
// constexpr auto d = g(3); // g(3)は非constexprなのでdの初期化はコンパイルエラー
auto const e = g(x); // eはここで初期化して、この後不变

constexpr auto pi = PI; // PIもconstexprなので初期化が可能

auto templ_a = Temp1<a>{}; // aはconstexprなのでaの初期化が可能
auto templ_f = Temp1<f(a)>{}; // f(a)はconstexprなのでaの初期化が可能
// auto templ_x = Temp1<x>{}; // xは非constexprなのでテンプレートパラメータに指定できない

static_assert(templ_a.value == 9);
// static_assert(x == 0); // xは非constexprなのでstatic_assertで使用できない

```

下記のようなリカーシブな関数でも場合によってはconstexprにできる。これによりこの関数の実行速度は劇的に向上する。

```

// @@@ example/term_explanation/constexpr_ut.cpp 48

inline constexpr uint32_t BitMask(uint32_t bit_len) noexcept
{
    if (bit_len == 0) {
        return 0x0;
    }

    return BitMask(bit_len - 1) | (0x01 << (bit_len - 1));
}

```

下記の単体テストが示すように、

- 引数がconstexprである場合、上記constexpr関数の戻り値はconstexprになるため、static_assertのオペランド式としても利用できる。
- 引数がconstexprでない場合、上記constexpr関数は通常の関数として振舞うため、戻り値はconstexprとならない。

```

// @@@ example/term_explanation/constexpr_ut.cpp 63

constexpr auto b_0x00000000 = BitMask(0);
constexpr auto b_0x000000ff = BitMask(8);

static_assert(b_0x00000000 == 0x00000000);
static_assert(b_0x000000ff == 0x000000ff);
static_assert(BitMask(16) == 0x0000'ffff);

constexpr auto bit_len_constexpr = 24U;
static_assert(BitMask(bit_len_constexpr) == 0x00ff'ffff);

auto bit_len = 24U;

// bit_lenがconstexprでないことによりBitMask(bit_len)もconstexprでないため、
// コンパイルできない
// constexpr auto b_0x00ffffffff = BitMask(bit_len);

// b_0x00ffffffffの定義からconstexprを外せばコンパイル可能
// ただし、コンパイル時でなくランタイム時動作になるため動作が遅い
auto b_0x00ffffffff = BitMask(bit_len);

ASSERT_EQ(b_0x00ffffffff, 0x00ff'ffff);

```

下記のようにクラスのコンストラクタをconstexprとすることで、コンパイル時にリテラルとして振る舞うクラスを定義することができる。

```

// @@@ example/term_explanation/constexpr_ut.cpp 90

class Integer {
public:
    constexpr Integer(int32_t integer) noexcept : integer_{integer} {}
    constexpr int32_t Get() const noexcept { return integer_; } // constexprメンバ関数はconst
    constexpr int32_t Always2() const noexcept { return 2; } // constexprメンバ関数はconst
    static constexpr int32_t Always3() noexcept { return 3; } // static関数のconstexpr化

private:
    int32_t integer_;
};

// @@@ example/term_explanation/constexpr_ut.cpp 107

constexpr auto int3      = 3; // int3はconstexpr
constexpr auto integer3 = Integer{int3}; // integer3自体がconstexpr
static_assert(integer3.Get() == 3, "wrong number"); // integer3.Get()もconstexpr

auto integer4 = Integer{4};
// integer4は非constexprであるため、integer4.Get()も非constexprとなり、コンパイル不可

```

```
// static_assert(integer4.Get() == 4, "wrong number");

// integer4は非constexprだが、integer4.Allway2()はconstexprであるため、コンパイル可能
static_assert(integer4.Allways2() == 2, "wrong number");
```

ユーザ定義リテラル演算子

ユーザ定義リテラル演算子とは以下のようなものである。

```
// @@@ example/term_explanation/user_defined_literal_ut.cpp 4

constexpr int32_t one_km = 1000;

// ユーザ定義リテラル演算子の定義
constexpr int32_t operator""_kilo_meter(unsigned long long num_by_mk) { return num_by_mk * one_km; }
constexpr int32_t operator""_meter(unsigned long long num_by_m) { return num_by_m; }
```

```
// @@@ example/term_explanation/user_defined_literal_ut.cpp 15

int32_t km = 3_kilo_meter; // ユーザ定義リテラル演算子の利用
int32_t m = 3000_meter; // ユーザ定義リテラル演算子の利用

ASSERT_EQ(m, km);
```

std::string型リテラル

“xxx”sとすることで、std::string型のリテラルを作ることができる。

```
// @@@ example/term_explanation/user_defined_literal_ut.cpp 26

using namespace std::literals::string_literals;

auto a = "str"s; // aはstd::string
auto b = "str"; // bはconst char*

static_assert(std::is_same_v<decltype(a), std::string>);
ASSERT_EQ(std::string{"str"}, a);

static_assert(std::is_same_v<decltype(b), char const*>);
ASSERT_STREQ("str", b);
```

オブジェクトと生成

特殊メンバ関数

特殊メンバ関数とは下記の関数を指す。

- デフォルトコンストラクタ
- copyコンストラクタ
- copy代入演算子
- moveコンストラクタ
- move代入演算子
- デストラクタ

ユーザがこれらを一切定義しない場合、または一部のみを定義する場合、コンパイラは、下記のテーブル等で示すルールに従い、特殊関数メンバの宣言、定義の状態を定める。

左1列目がユーザによる各関数の宣言を表し、2列目以降はユーザ宣言の影響による各関数の宣言の状態を表す。

下記表において、

- 「= default」とは、「コンパイラによってその関数が`= default`と宣言された」状態であることを表す。
- 「`= default`」とは、`= default`と同じであるが、バグが発生しやすいので推奨されない。
- 「宣言無し」とは、「コンパイラによってその関数が`= default`と宣言された状態ではない」ことを表す。
 - 「moveコンストラクタが`= default`と宣言された状態ではない」且つ「copyコンストラクタが宣言されている」場合、`rvalue`を使用したオブジェクトの初期化には、moveコンストラクタの代わりにcopyコンストラクタが使われる。
 - 「move代入演算子が`= default`と宣言された状態ではない」且つ「copy代入演算子が宣言されている」場合、`rvalue`を使用したオブジェクトの代入には、move代入演算子の代わりにcopy代入演算子が使われる。
- 「`= delete`」とは「コンパイラによってその関数が`= delete`と宣言された」状態であることを表す。

ユーザによる特殊関数の宣言	デフォルト コンストラクタ	デストラクタ	copy コンストラクタ	copy 代入演算子	move コンストラ クタ	move 代入演算子
宣言無し	= default	= default	= default	= default	= default	= default
非デフォルトコンストラクタ	宣言なし	= default	= default	= default	= default	= default
デフォルトコンストラクタ	-	= default	= default	= default	= default	= default
デストラクタ	= default	-	= default	= default	宣言なし	宣言なし
copyコンストラクタ	宣言なし	= default	-	= default	宣言なし	宣言なし
copy代入演算子	= default	= default	= default	-	宣言なし	宣言なし
moveコンストラクタ	宣言なし	= default	= delete	= delete	-	宣言なし
move代入演算子	= default	= default	= delete	= delete	宣言なし	-

上記表より、下記のようなことがわかる。

- ユーザが上記6メンバ関数を一切宣言しない場合、それらはコンパイラにより暗黙に宣言、定義される。
- ユーザがcopyコンストラクタを宣言した場合、デフォルトコンストラクタは暗黙に宣言、定義されない。
- ユーザがcopyコンストラクタを宣言した場合、copy代入演算子はコンパイラにより暗黙に宣言、定義されるが、そのことは推奨されない(= defaultは非推奨のdefault宣言を指す)。
- moveコンストラクタ、move代入演算子は、以下のいずれもが明示的に宣言されていない場合にのみ暗黙に宣言、定義される。
 - copyコンストラクタ
 - copy代入演算子(operator =)
 - moveコンストラクタ
 - move代入演算子
 - デストラクタ
- ユーザがmoveコンストラクタまたはmove代入演算子を宣言した場合、copyコンストラクタ、copy代入演算子は= deleteされる。

初期化子リストコンストラクタ

初期化子リストコンストラクタ(リスト初期化用のコンストラクタ)とは、{}によるリスト初期化をサポートするためのコンストラクタである。下記コードでは、E::E(std::initializer_list<uint32_t>)が初期化子リストコンストラクタである。

```
// @@@ example/term_explanation/constructor_ut.cpp 6

class E {
public:
    E() : str_{"default constructor"} {}

    // 初期化子リストコンストラクタ
    explicit E(std::initializer_list<uint32_t>) : str_{"initializer list constructor"} {}

    explicit E(uint32_t, uint32_t) : str_{"uint32_t uint32_t constructor"} {}

    std::string const& GetString() const { return str_; }

private:
    std::string const str_;
};

TEST(Constructor, initializer_list_constructor)
{
    E const e0;
    ASSERT_EQ("default constructor", e0.GetString());

    E const e1{};
    ASSERT_EQ("default constructor", e1.GetString());

    E const e2{3, 4}; // E::E(uint32_t, uint32_t)の呼び出しと区別が困難
    ASSERT_EQ("initializer list constructor", e2.GetString());

    E const e3{3, 4}; // E::E(std::initializer_list<uint32_t>)の呼び出しと区別が困難
    ASSERT_EQ("uint32_t uint32_t constructor", e3.GetString());
}
```

デフォルトコンストラクタと初期化子リストコンストラクタが、それぞれに定義されているクラスの初期化時に空の初期化子リストが渡された場合、デフォルトコンストラクタが呼び出される。

初期化子リストコンストラクタと、「その初期化子リストの要素型と同じ型の仮引数のみを受け取るコンストラクタ(上記コードのE::E(uint32_t, uint32_t))」の両方を持つクラスの初期化時にどちらでも呼び出せる初期化子リストが渡された場合({}を使った呼び出し)、初期化子コンストラクタが呼び出される。

継承コンストラクタ

継承コンストラクタとは、基底クラスで定義したコンストラクタ群を、派生クラスのインターフェースとしても使用できるようにするための機能である。下記コードのように、継承コンストラクタは派生クラス内でusingを用いて宣言される。

```
// @@@ example/term_explanation/constructor_ut.cpp 40

class Base {
public:
    explicit Base(int32_t b) noexcept : b_{b} {}
    virtual ~Base() = default;
    ...
};

class Derived : public Base {
public:
    using Base::Base; // 継承コンストラクタ
#if 0
    Derived(int32_t b) : Base{b} {} // オールドスタイル
#endif
};

void f() noexcept
{
    Derived d{1}; // Derived::Derived(int32_t)が使える
    ...
}
```

委譲コンストラクタ

委譲コンストラクタとは、コンストラクタから同じクラスの他のコンストラクタに処理を委譲する機能である。以下のコード中では、委譲コンストラクタを使い、A::A(uint32_t)の処理をA::A(std::string const&)へ委譲している。

```
// @@@ example/term_explanation/constructor_ut.cpp 72

class A {
public:
    explicit A(std::string str) : str_{std::move(str)}
    {
        ...
    }

    explicit A(uint32_t num) : A{std::to_string(num)} // 委譲コンストラクタ
    {
    }

private:
    std::string str_;
};
```

非explicitなコンストラクタによる暗黙の型変換

非explicitなコンストラクタによる暗黙の型変換とは、

```
// @@@ example/term_explanation/implicit_conversion_ut.cpp 6

class Person {
public:
    Person(char const* name, uint32_t age = 0) : name_{name}, age_{age} {}
    Person(Person const&) = default;
    Person& operator=(Person const&) = default;

    std::string const& GetName() const noexcept { return name_; }
    uint32_t GetAge() const noexcept { return age_; }

private:
    std::string name_; // コピーをするため非const
    uint32_t age_;
};

bool operator==(Person const& lhs, Person const& rhs) noexcept
```

```
{  
    return (lhs.GetName() == rhs.GetName()) && (lhs.GetAge() == rhs.GetAge());  
}
```

上記のクラスPersonを使用して、下記のようなコードをコンパイルできるようにする機能である。

```
// @@@ example/term_explanation/implicit_conversion_ut.cpp 27  
  
void f(Person const& person) noexcept  
{  
    ...  
  
    void using_implicit_coversion()  
    {  
        f("Ohtani"); // "Ohtani"はPerson型ではないが、コンパイル可能  
    }  
}
```

この記法は下記コードの短縮形であり、コードの見た目をシンプルに保つ効果がある。

```
// @@@ example/term_explanation/implicit_conversion_ut.cpp 41  
  
void not_using_implicit_coversion()  
{  
    f(Person{"Ohtani"}); // 本来は、fの引数はPerson型  
}
```

この記法は下記のようにstd::string等のSTLでも多用され、その効果は十分に発揮されているものの、

```
// @@@ example/term_explanation/implicit_conversion_ut.cpp 53  
  
auto otani = std::string("Ohtani");  
  
...  
  
if (otani == "Ohtani") { // 暗黙の型変換によりコンパイルできる  
    ...  
}
```

以下のようなコードがコンパイルできてしまうため、わかりづらいバグの元にもなる。

```
// @@@ example/term_explanation/implicit_conversion_ut.cpp 67  
  
auto otani = Person{"Ohtani", 26};  
  
...  
  
if (otani == "Otani") { // このコードがコンパイルされる。  
    ...  
}
```

下記のようにコンストラクタにexplicitを付けて宣言することにより、この問題を防ぐことができる。

```
// @@@ example/term_explanation/implicit_conversion_ut.cpp 94  
  
class Person {  
public:  
    explicit Person(char const* name, uint32_t age = 0) : name_{name}, age_{age} {}  
    Person(Person const&) = default;  
    Person& operator=(Person const&) = default;  
  
    ...  
};  
  
void prohibit_implicit_coversion()  
{  
    #if 0 // explicit付きのコンストラクタを持つPersonと違い、コンパイルできない。  
        f("Ohtani");  
    #else  
        f(Person{"Ohtani"});  
    #endif  
  
    auto otani = Person{"Ohtani", 26};  
  
    ...  
  
    #if 0  
        if (otani == "Otani") { // このコードもコンパイルできない。  
            ...  
    }
```

```

#else
    if (otani == Person{"Otani", 26}) { // この記述を強制できる。
        ...
    }
#endif
}

```

std::stringは暗黙の型変換を許して良く、(多くの場合)Personには暗黙の型変換をしない方が良い理由は、

- std::stringの役割は文字列の管理と演算のみであるため、std::stringを文字列リテラルと等価なもののように扱っても違和感がない
- Personは、明らかに文字列リテラルと等価なものではない

といったセマンティクス的観点(「シンタックス、セマンティクス」参照)によるものである。

クラスPersonと同様に、ほとんどのユーザ定義クラスには非explicitなコンストラクタによる暗黙の型変換は必要ない。

NSDMI

NSDMIとは、non-static data member initializerの略語であり、下記のような非静的なメンバ変数の初期化子を指す。

```

// @@@ example/term_explanation/nsdmi.cpp 8

class A {
public:
    A() : a_{1} // NSDMIではなく、非静的なメンバ初期化子による初期化
    {
    }

private:
    int32_t    a_;
    int32_t    b_ = 0;           // NSDMI
    std::string str_{"init"};   // NSDMI
};

```

一様初期化

一様初期化(uniform initialization)とは、C++11で導入された、コンストラクタの呼び出しをリスト初期化と合わせて{}で記述する構文である。

```

// @@@ example/term_explanation/uniform_initialization_ut.cpp 12

struct X {
    X(int) {}
};

X x0{0}; // 通常従来のコンストラクタ呼び出し
X x1 = 0; // 暗黙の型変換を使用した従来のコンストラクタ呼び出し

X x2{0}; // 一様初期化
X x3 = {0}; // 暗黙の型変換を使用した一様初期化

struct Y {
    Y(int, double, std::string) {}
};

auto lamda = [](int, double, std::string) -> Y {
    return {1, 3.14, "hello"}; // 暗黙の型変換を使用した一様初期化でのYの生成
};

```

変数による一様初期化が縮小型変換を起こす場合や、リテラルによる一様初期化がその値を変更する場合、コンパイルエラーとなるため、この機能を積極的に使用することで、縮小型変換による初期化のバグを未然に防ぐことができる。

```

// @@@ example/term_explanation/uniform_initialization_ut.cpp 34

int i{0}; // 一様初期化

bool b0 = 7; // 縮小型変換のため、b0の値はtrue(通常は1)となる
ASSERT_EQ(b0, 1);

// bool b1{7}; // 縮小型変換のため、コンパイルエラー
// bool b2{i}; // 縮小型変換のため、コンパイルエラー

uint8_t u8_0 = 256; // 縮小型変換のためu8_0は0となる
ASSERT_EQ(u8_0, 0);

// uint8_t u8_1{256}; // 縮小型変換のため、コンパイルエラー

```

```

// uint8_t u8_2[i];    // 縮小型変換のため、コンパイルエラー

uint8_t array0[3]{1, 2, 255}; // 一様初期化
// uint8_t array1[3] = {1, 2, 256}; // 縮小型変換のため、コンパイルエラー
// uint8_t array2[3]{1, 2, 256};    // 縮小型変換のため、コンパイルエラー
// uint8_t array2[3]{1, 2, 1};      // 縮小型変換のため、コンパイルエラー

int i0 = 1.0; // 縮小型変換のため、i0の値は1
ASSERT_EQ(i0, 1);

// int ii{1.0}; // 縮小型変換のため、コンパイルエラー

double d{1}; // 縮小型変換は起こらないのでコンパイル可能
// int i2{d}; // 縮小型変換のため、コンパイルエラー

```

AAAスタイル

このドキュメントでのAAAとは、単体テストのパターンarrange-act-assertではなく、almost always autoを指し、AAAスタイルとは、「可能な場合、型を左辺に明示して変数を宣言する代わりに、autoを使用する」というコーディングスタイルである。この用語は、Andrei Alexandrescuによって造られ、Herb Sutterによって広く推奨されている。

特定の型を明示して使用する必要がない場合、下記のように書く。

```

// @@@ example/term_explanation/aaa.cpp 11

auto i = 1;
auto ui = 1U;
auto d = 1.0;
auto s = "str";
auto v = {0, 1, 2};

for (auto i : v) {
    // 何らかの処理
}

auto add = [](auto lhs, auto rhs) { // -> return_typeのような記述は不要
    return lhs + rhs;           // addの型もautoで良い
};

// 上記変数の型の確認
static_assert(std::is_same_v<decltype(i), int>);
static_assert(std::is_same_v<decltype(ui), unsigned int>);
static_assert(std::is_same_v<decltype(d), double>);
static_assert(std::is_same_v<decltype(s), char const*>);
static_assert(std::is_same_v<decltype(v), std::initializer_list<int>>);

char s2[] = "str"; // 配列の宣言には、AAAは使えない
static_assert(std::is_same_v<decltype(s2), char[4]>);

int* p0 = nullptr; // 初期値がnullptrであるポインタの初期化には、AAAは使うべきではない
auto p1 = static_cast<int*>(nullptr); // NG
auto p2 = p0;           // OK
auto p3 = nullptr;      // NG 通常、想定通りにならない
static_assert(std::is_same_v<decltype(p3), std::nullptr_t>);

```

特定の型を明示して使用する必要がある場合、下記のように書く。

```

// @@@ example/term_explanation/aaa.cpp 51

auto b = new char[10]{0};
auto v = std::vector<int>{0, 1, 2};
auto s = std::string{"str"};
auto sv = std::string_view{"str"};

static_assert(std::is_same_v<decltype(b), char*>);
static_assert(std::is_same_v<decltype(v), std::vector<int>>);
static_assert(std::is_same_v<decltype(s), std::string>);
static_assert(std::is_same_v<decltype(sv), std::string_view>);

// 大量のstd::stringオブジェクトを定義する場合
using std::literals::string_literals::operator""s;

auto s_0 = "222"s; // OK
// ...
auto s_N = "222"s; // OK

static_assert(std::is_same_v<decltype(s_0), std::string>);
static_assert(std::is_same_v<decltype(s_N), std::string>);

```

```
// 大量のstd::string_viewオブジェクトを定義する場合
using std::literals::string_view_literals::operator""sv;

auto sv_0 = "222"sv; // OK
// ...
auto sv_N = "222"sv; // OK

static_assert(std::is_same_v<decltype(sv_0), std::string_view>);
static_assert(std::is_same_v<decltype(sv_N), std::string_view>);

std::mutex mtx; // std::mutexはmove出来ないので、AAAスタイル不可
auto lock = std::lock_guard{mtx};

static_assert(std::is_same_v<decltype(lock), std::lock_guard<std::mutex>>);
```

関数の戻り値を受け取る変数を宣言する場合、下記のように書く。

```
// @@@ example/term_explanation/aaa.cpp 94

auto v = std::vector<int>{0, 1, 2};

// AAAを使わない例
std::vector<int>::size_type t0{v.size()}; // 正確に書くとこうなる
std::vector<int>::iterator itr0 = v.begin(); // 正確に書くとこうなる

std::unique_ptr<int> p0 = std::make_unique<int>(3);

// 上記をAAAにした例
auto t1 = v.size(); // size()の戻りは算術型であると推測できる
auto itr1 = v.begin(); // begin()の戻りはイテレータであると推測できる

auto p1 = std::make_unique<int>(3); // make_uniqueの戻りはstd::unique_ptrであると推測できる
```

ただし、関数の戻り値型が容易に推測しがたい下記のような場合、型を明示しないAAAスタイルは使うべきではない。

```
// @@@ example/term_explanation/aaa.cpp 121

extern std::map<std::string, int> gen_map();

// 上記のような複雑な型を戻す関数の場合、AAAを使うと可読性が落ちる
auto map0 = gen_map();

for (auto [str, i] : gen_map()) {
    // 何らかの処理
}

// 上記のような複雑な型を戻す関数の場合、AAAを使うと可読性が落ちるため、AAAにしない
std::map<std::string, int> map1 = gen_map(); // 型がコメントとして役に立つ

for (std::pair<std::string, int> str_i : gen_map()) {
    // 何らかの処理
}

// 型を明示したAAAスタイルでも良い
auto map2 = std::map<std::string, int>{gen_map()}; // 型がコメントとして役に立つ
```

インライン関数や関数テンプレートの宣言は、下記のように書く。

```
// @@@ example/term_explanation/aaa.cpp 148

template <typename F, typename T>
auto apply_0(F&& f, T value)
{
    return f(value);
}
```

ただし、インライン関数や関数テンプレートが複雑な下記のような場合、AAAスタイルは出来る限り避けるべきである。

```
// @@@ example/term_explanation/aaa.cpp 156

template <typename F, typename T>
auto apply_1(F&& f, T value) -> decltype(f(std::declval<T>())) // autoを使用しているが、AAAではない
{
    auto cond = false;
    auto param = value;

    // 複雑な処理

    if (cond) {
```

```

        return f(param);
    }
    else {
        return f(value);
    }
}

```

このスタイルには下記のような狙いがある。

- コードの安全性の向上

`auto`で宣言された変数は未初期化にすることができないため、未初期化変数によるバグを防げる。また、下記のように縮小型変換(下記では、`unsigned`から`signed`の変換)を防ぐこともできる。

```
// @@@ example/term_explanation/aaa.cpp 183

auto v = std::vector<int>{0, 1, 2};

int t0 = v.size(); // 縮小型変換されるため、バグが発生する可能性がある
// int t1{v.size()}; 縮小型変換のため、コンパイルエラー
auto t2 = v.size(); // t2は正確な型
```

- コードの可読性の向上

冗長なコードを排除することで、可読性の向上が見込める。

- コードの保守性の向上

「変数宣言時での左辺と右辺を同一の型にする」非AAAスタイルはDRYの原則に反するが、この観点において、AAAスタイルはDRYの原則に沿うため、コード修正時に型の変更があった場合でも、それに付随したコード修正を最小限に留められる。

AAAスタイルでは、以下のような場合に注意が必要である。

- 関数の戻り値を`auto`で宣言された変数で受ける場合

上記で述べた通り、AAAの過剰な仕様は、可読性を下げてしまう。

- `auto`で推論された型が直感に反する場合

下記のような型推論は、直感に反する場合があるため、`auto`の使い方に対する習熟が必要である。

```
// @@@ example/term_explanation/aaa.cpp 197

auto str0 = "str";
static_assert(std::is_same_v<char const*, decltype(str0)>); // str0はchar[4]ではない

// char[]が必要ならば、AAAを使わずに下記のように書く
char str1[] = "str";
static_assert(std::is_same_v<char[4], decltype(str1)>);

// &が必要になるパターン
class X {
public:
    explicit X(int32_t a) : a_{a} {}
    int32_t& Get() { return a_; }

private:
    int32_t a_;
};

X x{3};

auto a0 = x.Get();
ASSERT_EQ(3, a0);

a0 = 4;
ASSERT_EQ(4, a0);
ASSERT_EQ(3, x.Get()); // a0はリファレンスではないため、このような結果になる

// X::a_のリファレンスが必要ならば、下記のように書く
auto& a1 = x.Get();
a1      = 4;
ASSERT_EQ(4, a1);
ASSERT_EQ(4, x.Get()); // a1はリファレンスであるため、このような結果になる

// constが必要になるパターン
class Y {
public:
    std::string& Name() { return name_; }
    std::string const& Name() const { return name_; }

private:
```

```

        std::string name_{"str"};
    };

auto const y = Y{};

auto      name0 = y.Name(); // std::stringがコピーされる
auto&     name1 = y.Name(); // name1はconstに見えない
auto const& name2 = y.Name(); // このように書くべき

static_assert(std::is_same_v<std::string, decltype(name0)>);
static_assert(std::is_same_v<std::string const&, decltype(name1)>);
static_assert(std::is_same_v<std::string const&, decltype(name2)>);

// 範囲for文でのauto const&
auto const v = std::vector<std::string>{"0", "1", "2"};

for (auto s : v) { // sはコピー生成される
    static_assert(std::is_same_v<std::string, decltype(s)>);
}

for (auto& s : v) { // sはconstに見えない
    static_assert(std::is_same_v<std::string const&, decltype(s)>);
}

for (auto const& s : v) { // このように書くべき
    static_assert(std::is_same_v<std::string const&, decltype(s)>);
}

```

オブジェクトの所有権

オブジェクトxがオブジェクトaの解放責務を持つ場合、xはaの所有権を持つ(もしくは、所有する)という。

定義から明らかな通り、ダイナミックに生成されたaをxが所有する場合、xはaへのポインタをdeleteする責務を持つ。

xがaを所有し、且つxがaを他のオブジェクトと共有しない場合、「xはaを排他所有する」という。

オブジェクト群x0、x1、...、xNがaを所有する場合、「x0、x1、...、xNはaを共有所有する」という。

x0、x1、...、xNがaを共有所有する場合、x0、x1、...、xN全体で、aへのポインタをdeleteする責務を持つ。

下記で示したような状況では、ダイナミックに生成されたオブジェクトの所有権の所在をコードから直ちに読み取ることは困難であり、その解放責務も曖昧となる。

```

// @@@ example/term_explanation/ambiguous_ownership_ut.cpp 11

class A {
    // 何らかの宣言
};

class X {
public:
    explicit X(A* a) : a_{a} {}
    A* GetA() { return a_; }

private:
    A* a_;
};

auto* a = new A;
auto x = X{a};
// aがxに排他所有されているのか否かの判断は難しい

auto x0 = X{new A};
auto x1 = X{x0.GetA()};
// x0生成時にnewされたオブジェクトがx0とx1に共有所有されているのか否かの判断は難しい

```

こういった問題に対処するためのプログラミングパターンを以下の「[オブジェクトの排他所有](#)」と「[オブジェクトの共有所有](#)」で解説する。

オブジェクトの排他所有

オブジェクトの排他所有や、それを容易に実現するための[`std::unique_ptr`](#)の仕様をを説明するために、下記のようにクラスA、Xを定義する。

```
// @@@ example/term_explanation/unique_ptr_ownership_ut.cpp 7
```

```

class A final {
public:
    explicit A(int32_t n) noexcept : num_{n} { last_constructed_num_ = num_; }
    ~A() { last_destructed_num_ = num_; }

    int32_t GetNum() const noexcept { return num_; }

    static int32_t LastConstructedNum() noexcept { return last_constructed_num_; }
    static int32_t LastDestructedNum() noexcept { return last_destructed_num_; }

private:
    int32_t const num_;
    static int32_t last_constructed_num_;
    static int32_t last_destructed_num_;
};

int32_t A::last_constructed_num_ = -1;
int32_t A::last_destructed_num_ = -1;

class X final {
public:
    // Xオブジェクトの生成と、ptrからptr_へ所有権の移動
    explicit X(std::unique_ptr<A>&& ptr) : ptr_{std::move(ptr)} {}

    // ptrからptr_へ所有権の移動
    void Move(std::unique_ptr<A>&& ptr) noexcept { ptr_ = std::move(ptr); }

    // ptr_から外部への所有権の移動
    std::unique_ptr<A> Release() noexcept { return std::move(ptr_); }

    A const* GetA() const noexcept { return ptr_ ? ptr_.get() : nullptr; }

private:
    std::unique_ptr<A> ptr_{};
};

```

下記に示した上記クラスの単体テストにより、オブジェクトの所有権やその移動、`std::unique_ptr`、`std::move()`、`rvalue`の関係を解説する。

```

// @@@ example/term_explanation/unique_ptr_ownership_ut.cpp 48

// ステップ0
// まだ、クラスAオブジェクトは生成されていないため、
// A::LastConstructedNum()、A::LastDestructedNum()は初期値である-1である。
ASSERT_EQ(-1, A::LastConstructedNum());           // まだ、A::A()は呼ばれてない
ASSERT_EQ(-1, A::LastDestructedNum());            // まだ、A::~A()は呼ばれてない

```

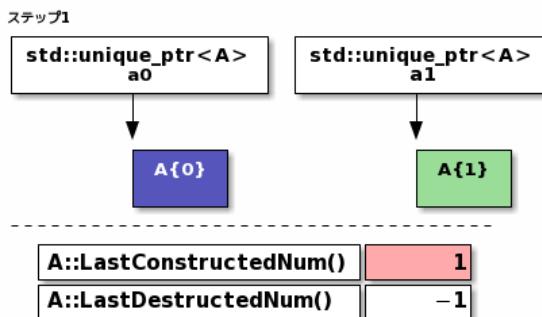
```

// @@@ example/term_explanation/unique_ptr_ownership_ut.cpp 57

// ステップ1
// a0、a1がそれぞれ初期化される。
auto a0 = std::make_unique<A>(0);                // a0はA{0}を所有
auto a1 = std::make_unique<A>(1);                // a1はA{1}を所有

ASSERT_EQ(1, A::LastConstructedNum());           // A{1}は生成された
ASSERT_EQ(-1, A::LastDestructedNum());            // まだ、A::~A()は呼ばれてない

```



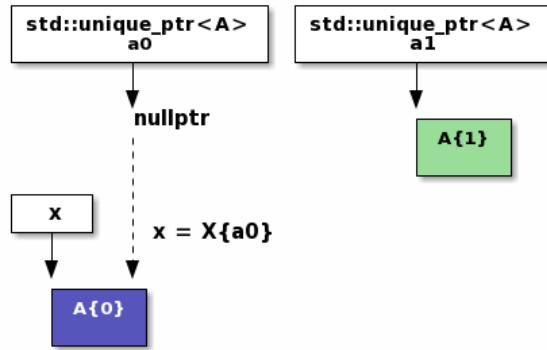
```

// @@@ example/term_explanation/unique_ptr_ownership_ut.cpp 67

// ステップ2
// xが生成され、オブジェクトA{0}の所有がa0からxへ移動する。
ASSERT_EQ(0, a0->GetNum());                      // a0はA{0}を所有
auto x = X{std::move(a0)};                         // xの生成と、a0からxへA{0}の所有権の移動
ASSERT_FALSE(a0);                                  // a0は何も所有していない

```

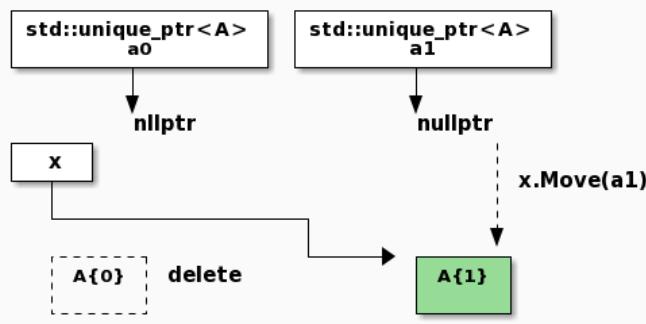
ステップ2



```
// @@@ example/term_explanation/unique_ptr_ownership_ut.cpp 75
```

```
// ステップ3
// オブジェクトA{1}の所有がa1からxへ移動する。
// xは以前保持していたA{0}オブジェクトへのポインタをdeleteするため
// (std::unique_ptrによる自動delete)、A::LastDestructedNum()の値が0になる。
ASSERT_EQ(1, a1->GetNum()); // a1はA{1}を所有
x.Move(std::move(a1)); // xによるA{0}の解放
// a1からxへA{1}の所有権の移動
ASSERT_EQ(0, A::LastDestructedNum()); // A{0}は解放された
ASSERT_FALSE(a1); // a1は何も所有していない
ASSERT_EQ(1, x.GetA()->GetNum()); // xはA{1}を所有
```

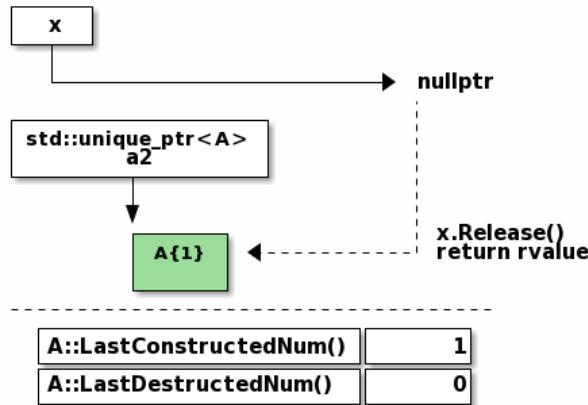
ステップ3



```
// @@@ example/term_explanation/unique_ptr_ownership_ut.cpp 88
```

```
// ステップ4
// x.ptr_はstd::unique_ptr<A>であるため、ステップ3の状態では、
// x.ptr_はA{1}オブジェクトのポインタを保持しているが、
// x.Release()はそれをrvalueに変換し戻り値にする。
// その戻り値をa2で受け取るため、A{1}の所有はxからa2に移動する。
std::unique_ptr<A> a2{x.Release()}; // xからa2へA{1}の所有権の移動
ASSERT_EQ(nullptr, x.GetA()); // xは何も所有していない
ASSERT_EQ(1, a2->GetNum()); // a2はA{1}を所有
```

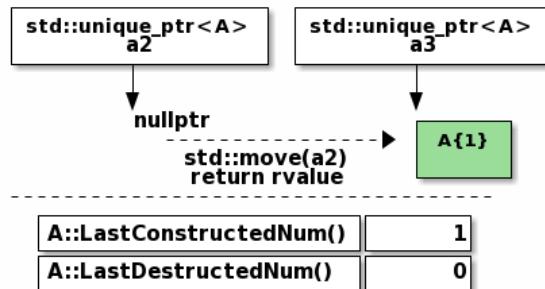
ステップ4



A::LastConstructedNum()	1
A::LastDestructedNum()	0

```
// @@@ example/term_explanation/unique_ptr_ownership_ut.cpp 99
// ステップ5
// a2をstd::move()によりrvalueに変換し、ブロック内のa3に渡すことで、
// A{1}の所有はa2からa3に移動する。
{
    std::unique_ptr<A> a3{std::move(a2)};
    ASSERT_FALSE(a2); // a2は何も所有していない
    ASSERT_EQ(1, a3->GetNum()); // a3はA{1}を所有
}
```

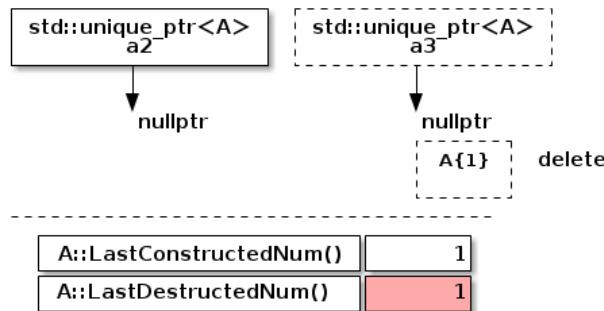
ステップ5



A::LastConstructedNum()	1
A::LastDestructedNum()	0

```
// @@@ example/term_explanation/unique_ptr_ownership_ut.cpp 109
// ステップ6
// このブロックが終了することで、std::unique_ptrであるa3のデストラクタが呼び出される。
// これはA{1}オブジェクトへのポインタをdeleteする。
}
ASSERT_EQ(1, A::LastDestructedNum()); // A{1}が解放されたことの確認
```

ステップ6



A::LastConstructedNum()	1
A::LastDestructedNum()	1

また、以下に見るように**std::unique_ptr**はcopy生成やcopy代入を許可しない。

```
// @@@ example/term_explanation/unique_ptr_ownership_ut.cpp 123
auto a0 = std::make_unique<A>(0);
```

```

// auto a1 = a0;                                // 下記のようなメッセージでコンパイルエラー
//     unique_ptr_ownership_ut.cpp:125:15: error: use of deleted function 'std::unique_ptr ...
auto a1 = std::move(a0);                      // すでに示したようにmove生成は可能

auto a2 = std::unique_ptr<A>{};

// a2 = a1;                                     // 下記のようなメッセージでコンパイルエラー
//     unique_ptr_ownership_ut.cpp:131:10: error: use of deleted function 'std::unique_ptr ...
a2 = std::move(a1);                          // すでに示したようにmove代入は可能

//
auto x0 = X{std::make_unique<A>(0)};

// auto x1 = x0;                                // Xはstd::unique_ptrをメンバとするため、
//                                                 // デフォルトのcopyコンストラクタによる生成は
//                                                 // コンパイルエラー
auto x1 = std::move(x0);                      // デフォルトのmove生成は可能

auto x2 = X{std::make_unique<A>(0)};

// x2 = x1;                                     // Xはstd::unique_ptrをメンバとするため、
//                                                 // デフォルトのcopy代入子の呼び出しは
//                                                 // コンパイルエラー
x2 = std::move(x1);                          // デフォルトのmove代入は可能

```

以上で示したstd::unique_ptrの仕様の要点をまとめると、以下のようになる。

- std::unique_ptrはダイナミックに生成されたオブジェクトを保持する。
- ダイナミックに生成されたオブジェクトを保持するstd::unique_ptrがスコープアウトすると、保持中のオブジェクトは自動的にdeleteされる。
- 保持中のオブジェクトを他のstd::unique_ptrにmoveすることはできるが、copyすることはできない。このため、下記に示すような不正な方法以外で、複数のstd::unique_ptrが1つのオブジェクトを共有することはできない。

```

// @@@ example/term_explanation/unique_ptr_ownership_ut.cpp 161

// 以下のようなコードを書いてはならない

auto a0 = std::make_unique<A>(0);
auto a1 = std::unique_ptr<A>{a0.get()}; // a1もa0が保持するオブジェクトを保持するが、
// 保持されたオブジェクトは二重解放される

auto a_ptr = new A{0};

auto a2 = std::unique_ptr<A>{a_ptr};
auto a3 = std::unique_ptr<A>{a_ptr}; // a3もa2が保持するオブジェクトを保持するが、
// 保持されたオブジェクトは二重解放される

```

こういった機能によりstd::unique_ptrはオブジェクトの排他所有を実現している。

オブジェクトの共有可能

オブジェクトの共有可能や、それを容易に実現するためのstd::shared_ptrの仕様をを説明するために、下記のようにクラスA、Xを定義する。

```

// @@@ example/term_explanation/shared_ptr_ownership_ut.cpp 7

class A final {
public:
    explicit A(int32_t n) noexcept : num_{n} { last_constructed_num_ = num_; }
    ~A() { last_destructed_num_ = num_; }

    int32_t GetNum() const noexcept { return num_; }

    static int32_t LastConstructedNum() noexcept { return last_constructed_num_; }
    static int32_t LastDestructedNum() noexcept { return last_destructed_num_; }

private:
    int32_t const num_;
    static int32_t last_constructed_num_;
    static int32_t last_destructed_num_;
};

int32_t A::last_constructed_num_ = -1;
int32_t A::last_destructed_num_ = -1;

```

```

class X final {
public:
    // Xオブジェクトの生成と、ptrからptr_へ所有権の移動もしくは共有
    explicit X(std::shared_ptr<A> ptr) : ptr_{std::move(ptr)} {}

    // ptrからptr_へ所有権の移動
    void Move(std::shared_ptr<A>&& ptr) noexcept { ptr_ = std::move(ptr); }

    int32_t UseCount() const noexcept { return ptr_.use_count(); }

    A const* GetA() const noexcept { return ptr_ ? ptr_.get() : nullptr; }

private:
    std::shared_ptr<A> ptr_{};
};

```

下記に示した上記クラスの単体テストにより、オブジェクトの所有権やその移動、共有、`std::shared_ptr`、`std::move()`、`rvalue`の関係を解説する。

```

// @@@ example/term_explanation/shared_ptr_ownership_ut.cpp 47

// ステップ0
// まだ、クラスAオブジェクトは生成されていないため、
// A::LastConstructedNum()、A::LastDestructedNum()は初期値である-1である。
ASSERT_EQ(-1, A::LastConstructedNum());           // まだ、A::A()は呼ばれてない
ASSERT_EQ(-1, A::LastDestructedNum());            // まだ、A::~A()は呼ばれてない

```

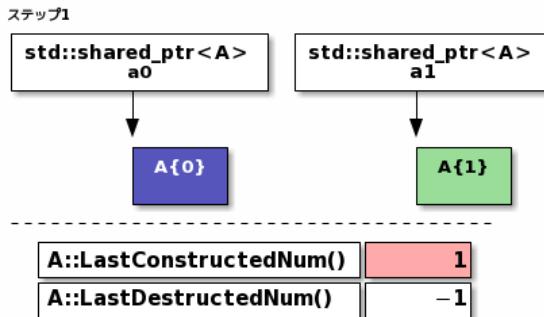
```

// @@@ example/term_explanation/shared_ptr_ownership_ut.cpp 56

// ステップ1
// a0、a1がそれぞれ初期化される。
auto a0 = std::make_shared<A>(0);                // a0はA{0}を所有
auto a1 = std::make_shared<A>(1);                // a1はA{1}を所有
ASSERT_EQ(1, a0.use_count());                     // A{0}の共有所有カウント数は1
ASSERT_EQ(1, a1.use_count());                     // A{1}の共有所有カウント数は1

ASSERT_EQ(1, A::LastConstructedNum());           // A{1}は生成された
ASSERT_EQ(-1, A::LastDestructedNum());            // まだ、A::~A()は呼ばれてない

```



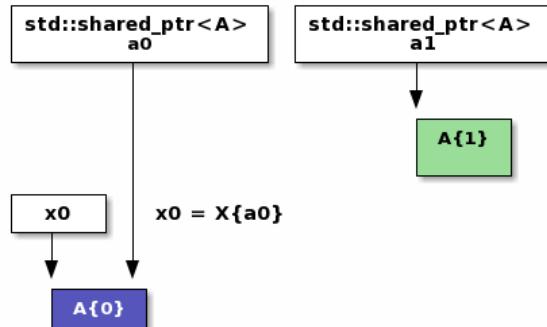
```

// @@@ example/term_explanation/shared_ptr_ownership_ut.cpp 68

// ステップ2
// x0が生成され、オブジェクトA{0}がa0とx0に共同所有される。
ASSERT_EQ(0, a0->GetNum());                      // a0はA{0}を所有
ASSERT_EQ(1, a0.use_count());                      // A{0}の共有所有カウントは1
auto x0 = X{a0};                                    // x0の生成と、a0とx0によるA{0}の共有所有
ASSERT_EQ(2, a0.use_count());                      // A{0}の共有所有カウント数は2
ASSERT_EQ(2, x0.UseCount());
ASSERT_EQ(x0.GetA(), a0.get());

```

ステップ2

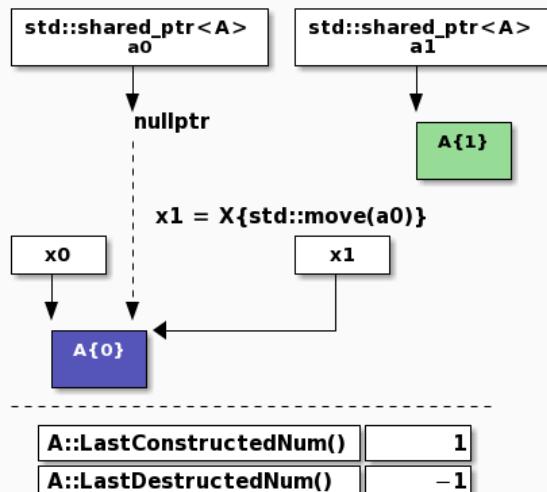


<code>A::LastConstructedNum()</code>	<code>1</code>
<code>A::LastDestructedNum()</code>	<code>-1</code>

```
// @@@ example/term_explanation/shared_ptr_ownership_ut.cpp 79
```

```
// ステップ3
// x1が生成され、オブジェクトA{0}の所有がa0からx1へ移動する。
auto x1 = X{std::move(a0)}; // x1の生成と、a0からx1へA{0}の所有権の移動
ASSERT_EQ(x1.GetA(), x1.GetA()); // x0、x1がA{0}を共有所有
ASSERT_EQ(2, x0.UseCount()); // A{0}の共有所有カウント数は2
ASSERT_EQ(2, x1.UseCount()); // A{0}は何も所有していない
ASSERT_FALSE(a0); // a0は何も所有していない
```

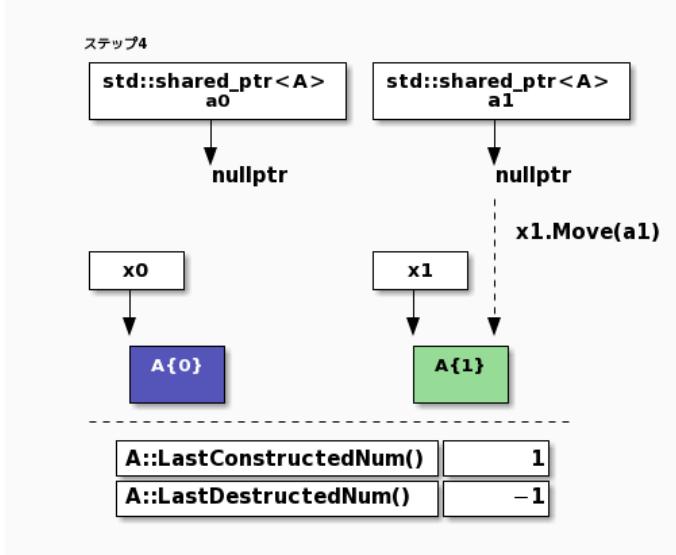
ステップ3



<code>A::LastConstructedNum()</code>	<code>1</code>
<code>A::LastDestructedNum()</code>	<code>-1</code>

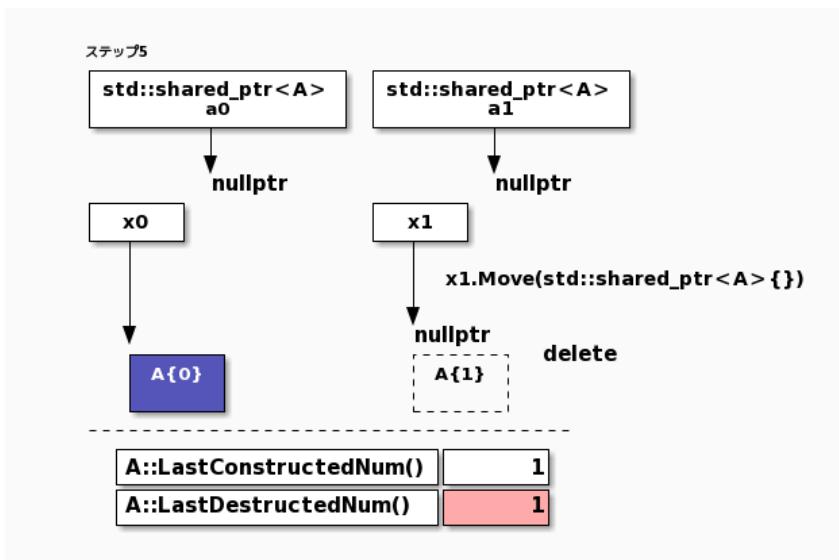
```
// @@@ example/term_explanation/shared_ptr_ownership_ut.cpp 89
```

```
// ステップ4
// オブジェクトA{1}の所有がa1からx1へ移動する。
// この時、x1::ptr_は下記のような手順で以前保持していたA{0}オブジェクトへの所有を放棄する。
// 1. x1::ptr_の共有所有カウント(ptr_.use_count()の戻り値)をデクリメント
// 2. 共有所有カウントが0ならば、ptr_で保持しているオブジェクト(この場合、A{0})をdelete
// 3. x1::ptr_の管理対象をに新規オブジェクト(この場合、A{1})に変更
//
// ここでは、x0::ptr_がA{0}を所有しているため、共有所有カウントは1であり、
// 従って、A{0}はdeleteされず、A::LastDestructedNum()の値は-1のまま。
ASSERT_EQ(1, a1->GetNum()); // a1はA{1}を所有
ASSERT_EQ(0, x1.GetA()->GetNum()); // x1はA{0}を所有
ASSERT_EQ(2, x1.UseCount()); // A{0}の共有所有カウント数は2
x1.Move(std::move(a1)); // x1はA{0}の代わりに、A{1}を所有
// a1からx1へA{1}の所有権の移動
ASSERT_EQ(-1, A::LastDestructedNum()); // x0がA{0}を所有するため、A{0}は未解放
ASSERT_FALSE(a1); // a1は何も所有していない
ASSERT_EQ(1, x1.GetA()->GetNum()); // x1はA{1}を所有
ASSERT_EQ(1, x1.UseCount()); // A{0}の共有所有カウント数は1
```



```
// @@@ example/term_explanation/shared_ptr_ownership_ut.cpp 110
```

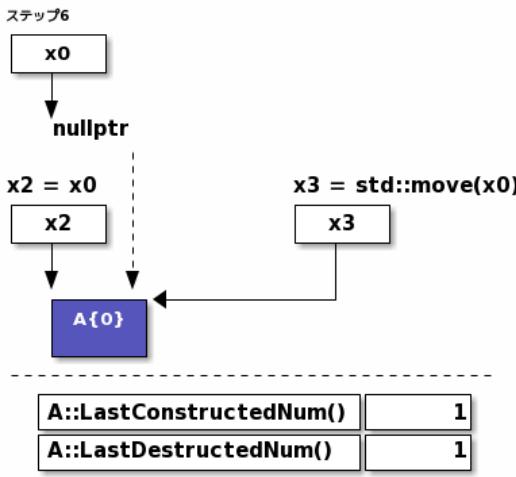
```
// ステップ5
// 現時点でx1はA{1}オブジェクトを保持している。
// x1::Moveに空のstd::shared_ptrを渡すことにより、A{1}を解放する。
x1.Move(std::shared_ptr<A>{}); // x1に空のstd::shared_ptr<A>を代入することで、
// A{1}を解放
ASSERT_EQ(nullptr, x1.GetA()); // x1は何も保持していない
ASSERT_EQ(1, A::LastDestructedNum()); // A{1}が解放された
```



```
// @@@ example/term_explanation/shared_ptr_ownership_ut.cpp 120
```

```
// ステップ6
// 現時点でx0はA{0}オブジェクトを保持している。
//
// ここでは、x0からx2、x3をそれぞれcopy、move生成し、
// この次のステップ7では、x2、x3がスコープアウトすることでA{0}を解放する。
{
    ASSERT_EQ(0, x0.GetA()->GetNum()); // x0はA{0}を所有
    ASSERT_EQ(1, x0.UseCount()); // A{0}の共有所有カウント数は1
    auto x2 = x0; // x0からx2をcopy生成
    ASSERT_EQ(x0.GetA(), x2.GetA()); // A{0}の共有所有カウント数は2
    ASSERT_EQ(2, x0.UseCount()); // A{0}の共有所有カウント数は2

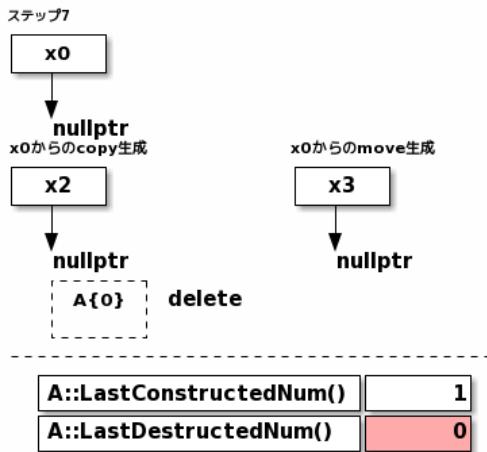
    auto x3 = std::move(x0); // x0からx2をmove生成、x0はA{0}の所有を放棄
    ASSERT_EQ(nullptr, x0.GetA()); // x2はA{0}を保有
    ASSERT_EQ(0, x2.GetA()->GetNum()); // x2、x3はA{0}を共有保有
    ASSERT_EQ(x2.GetA(), x3.GetA()); // A{1}の共有所有カウント数は2
    ASSERT_EQ(2, x2.UseCount()); // A{1}の共有所有カウント数は2
```



```

// @@@ example/term_explanation/shared_ptr_ownership_ut.cpp 142

// ステップ7
// このブロックが終了することで、x2、x3はスコープアウトする。
// デストラクタ呼び出しの順序はコンストラクタ呼び出しの逆になるため、
// 最初にx3::~X()が呼び出され、この延長でx3::ptr_のデストラクタが呼び出される。
// これによりA{0}の共有所有カウントは1になる。
// 次にx2::~X()が呼び出され、この延長でx2::ptr_のデストラクタが呼び出される。
// これによりA{0}の共有所有カウントは0になり、A{0}はdeleteされる。
//
} // x2、x3のスコープアウト
ASSERT_EQ(0, A::LastDestructedNum()); // A{0}が解放された
  
```



以上で示したstd::shared_ptrの仕様の要点をまとめると、以下のようになる。

- std::shared_ptrはダイナミックに生成されたオブジェクトを保持する。
- ダイナミックに生成されたオブジェクトを保持するstd::shared_ptrがスコープアウトすると、共有所有カウントはデクリメントされ、その値が0ならば保持しているオブジェクトはdeleteされる。
- std::shared_ptrを他のstd::shared_ptrに、
 - moveすることことで、保持中のオブジェクトの所有権を移動できる。
 - copyすることことで、保持中のオブジェクトの所有権を共有できる。
- 下記のようなコードはstd::shared_ptrの仕様が想定するセマンティクスに沿っておらず、未定義動作に繋がる。

```

// @@@ example/term_explanation/shared_ptr_ownership_ut.cpp 162

// 以下のようなコードを書いてはならない

auto a0 = std::make_shared<A>(0);
auto a1 = std::shared_ptr<A>{a0.get()};
  
```

// a1もa0が保持するオブジェクトを保持するが、
// 保持されたオブジェクトは二重解放される

```

auto a_ptr = new A{0};

auto a2 = std::shared_ptr<A>{a_ptr};
auto a3 = std::shared_ptr<A>{a_ptr}; // a3もa2が保持するオブジェクトを保持するが、
// 保持されたオブジェクトは二重解放される

```

こういった機能によりstd::shared_ptrはオブジェクトの共有所有を実現している。

オブジェクトのライフタイム

オブジェクトは、以下のような種類のライフタイムを持つ。

- 静的に生成されたオブジェクトのライフタイム
- thread_localに生成されたオブジェクトのライフタイム
- newで生成されたオブジェクトのライフタイム
- スタック上に生成されたオブジェクトのライフタイム
- prvalue(「rvalue」参照)のライフタイム

なお、リファレンスの初期化をrvalueで行った場合、そのrvalueはリファレンスがスコープを抜けるまで存続し続ける。

クラスのレイアウト

クラス(やそのクラスが継承した基底クラス)が仮想関数を持たない場合、そのクラスは、非静的なメンバ変数が定義された順にメモリ上に配置されたレイアウトを持つ(CPUアーキテクチャに依存したパディング領域が変数間に挿入されることもある)。このようなクラスはPOD(C++20では、PODという用語は非推奨となり、トリビアル型とスタンダードレイアウト型に用語が分割された)とも呼ばれ、C言語の構造体のレイアウトと互換性を持つことが一般的である。

クラス(やそのクラスが継承したクラス)が仮想関数を持つ場合、仮想関数呼び出しを行う(「オーバーライドとオーバーロードの違い」参照)ためのメモリレイアウトが必要になる。それを示すために、まずは下記のようにクラスX、Y、Zを定義する。

```

// @@@ example/term_explanation/class_layout_ut.cpp 4

class X {
public:
    virtual int64_t GetX() { return x_; }
    virtual ~X() {}

private:
    int64_t x_{1};
};

class Y : public X {
public:
    virtual int64_t GetX() override { return X::GetX() + y_; }
    virtual int64_t GetY() { return y_; }
    virtual ~Y() override {}

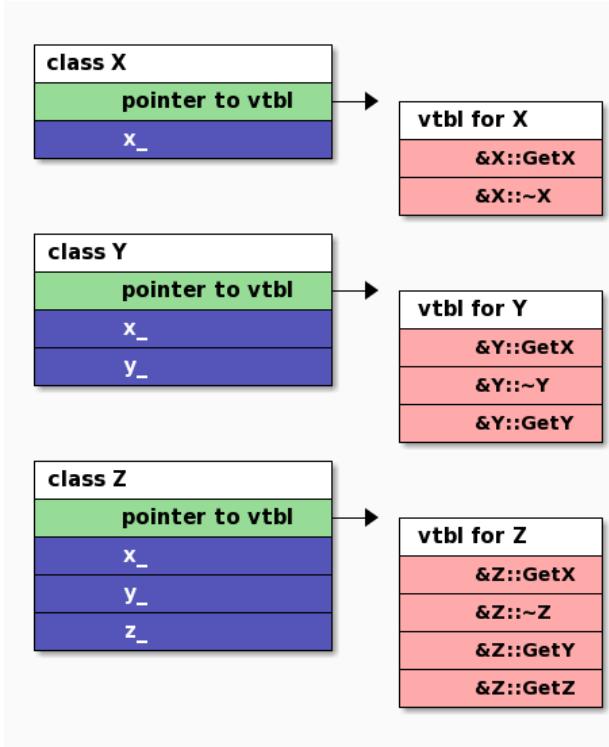
private:
    int64_t y_{2};
};

class Z : public Y {
public:
    virtual int64_t GetX() override { return Y::GetX() + z_; }
    virtual int64_t GetY() override { return Y::GetY() + z_; }
    virtual int64_t GetZ() { return z_; }
    virtual ~Z() override {}

private:
    int64_t z_{3};
};

```

通常のC++コンパイラが作り出すX、Y、Zの概念的なメモリレイアウトは下記のようになる。



各クラスがvtblへのポインタを保持するため、このドキュメントで使用しているg++では、`sizeof(X)`は8ではなく16、`sizeof(Y)`は16ではなく24、`sizeof(Z)`は24ではなく32となる。

g++の場合、以下のオプションを使用し、クラスのメモリレイアウトをファイルに出力することができる。

```
// @@ example/term_explanation/Makefile 19
CCFLAGS_ADD:=-fdump-lang-class
```

X、Y、Zのメモリレイアウトは以下の様に出力される。

```
Vtable for X
X::_ZTV1X: 5 entries
0  (int (*)(...))0
8   (int (*)(...))(& _ZTI1X)
16  (int (*)(...))X::GetX
24  (int (*)(...))X::~X
32  (int (*)(...))X::~X
```

```
Class X
size=16 align=8
base size=16 base align=8
X (0x0x7f54bbc23a80) 0
vptr=((& X::_ZTV1X) + 16)
```

```
Vtable for Y
Y::_ZTV1Y: 6 entries
0  (int (*)(...))0
8   (int (*)(...))(& _ZTI1Y)
16  (int (*)(...))Y::GetX
24  (int (*)(...))Y::~Y
32  (int (*)(...))Y::~Y
40  (int (*)(...))Y::GetY
```

```
Class Y
size=24 align=8
base size=24 base align=8
Y (0x0x7f54bbc3f000) 0
vptr=((& Y::_ZTV1Y) + 16)
X (0x0x7f54bbc23d20) 0
primary-for Y (0x0x7f54bbc3f000)
```

```
Vtable for Z
Z::_ZTV1Z: 7 entries
0  (int (*)(...))0
8   (int (*)(...))(& _ZTI1Z)
16  (int (*)(...))Z::GetX
```

```

24     (int (*)(...))Z::~Z
32     (int (*)(...))Z::~Z
40     (int (*)(...))Z::GetY
48     (int (*)(...))Z::GetZ

Class Z
    size=32 align=8
    base size=32 base align=8
Z (0x0x7f54bbc3f068) 0
    vptr=((& Z::ZTV1Z) + 16)
Y (0x0x7f54bbc3f0d0) 0
    primary-for Z (0x0x7f54bbc3f068)
X (0x0x7f54bbc43060) 0
    primary-for Y (0x0x7f54bbc3f0d0)

```

このようなメモリレイアウトは、

```

// @@@ example/term_explanation/class_layout_ut.cpp 40

auto z_ptr = new Z;

```

のようなオブジェクト生成に密接に関係する。その手順を下記の疑似コードにより示す。

```

// ステップ1 メモリアロケーション
void* ptr = malloc(sizeof(Z));

// ステップ2 ZオブジェクトのX部分の初期化
X* x_ptr = (X*)ptr;
x_ptr->vtbl = &vtbl_for_X           // Xのコンストラクタ呼び出し処理
x_ptr->x_ = 1;                      // Xのコンストラクタ呼び出し処理

// ステップ3 ZオブジェクトのY部分の初期化
Y* y_ptr = (Y*)ptr;
y_ptr->vtbl = &vtbl_for_Y           // Yのコンストラクタ呼び出し処理
y_ptr->y_ = 2;                      // Yのコンストラクタ呼び出し処理

// ステップ4 ZオブジェクトのZ部分の初期化
Z* z_ptr = (Z*)ptr;
z_ptr->vtbl = &vtbl_for_Z           // Zのコンストラクタ呼び出し処理
z_ptr->z_ = 3;                      // Zのコンストラクタ呼び出し処理

```

オブジェクトの生成がこのように行われるため、Xのコンストラクタ内で仮想関数GetX()を呼び出した場合、その時のvtblへのポインタはXのvtblを指しており(上記ステップ2)、X::GetX()の呼び出しどとなる(Z::GetX()の呼び出しどとはならない)。

なお、オブジェクトの解放は生成とは逆の順番で行われる。

オブジェクトのコピー

シャローコピー

シャローコピー(浅いコピー)とは、暗黙的、もしくは=defaultによってコンパイラが生成するようなcopyコンストラクタ、copy代入演算子が行うコピーであり、ディープコピーと対比的に使われる概念である。

以下のクラスShallowOKには、コンパイラが生成するcopyコンストラクタ、copy代入演算子と同等なものを定義したが、これは問題のないシャローコピーである(が、正しく自動生成される関数を実装すると、メンバ変数が増えた際にバグを生み出すことがあるため、実践的にはこのようなことはすべきではない)。

```

// @@@ example/term_explanation/deep_shallow_copy_ut.cpp 7

class ShallowOK {
public:
    explicit ShallowOK(char const* str = "") : str_{std::string(str)} {}
    std::string const& GetString() const noexcept { return str_; }

    // 下記2関数を定義しなければ、以下と同等なcopyコンストラクタ、copy代入演算子が定義される。
    ShallowOK(ShallowOK const& rhs) : str_{rhs.str_} {}

    ShallowOK& operator=(ShallowOK const& rhs)
    {
        str_ = rhs.str_;
        return *this;
    }

private:

```

```
    std::string str_;  
};
```

コンストラクタでポインタのようなリソースを確保し、デストラクタでそれらを解放するようなクラスの場合、シャローコピーは良く知られた問題を起こす。

下記のShallowNGはその例である。

```
// @@@ example/term_explanation/deep_shallow_copy_ut.cpp 43  
  
class ShallowNG {  
public:  
    explicit ShallowNG(char const* str = "") : str_(new std::string{str}) {}  
    ~ShallowNG() { delete str_; }  
    std::string const& GetString() const noexcept { return *str_; }  
  
private:  
    std::string* str_;  
};
```

シャローコピーにより、メンバで保持していたポインタ(ポインタが指しているオブジェクトではない)がコピーされてしまうため、下記のコード内のコメントで示した通り、メモリリークや2重解放を起こしてしまう。

```
// @@@ example/term_explanation/deep_shallow_copy_ut.cpp 60  
  
auto const s0 = ShallowNG{"s0"};  
  
// NG s0.str_とs1.str_は同じメモリを指すため~ShallowNG()に2重解放される。  
auto const s1 = ShallowNG{s0};  
  
auto s2 = ShallowNG{"s2"};  
  
// NG s2.str_が元々保持していたメモリは、解放できなくなる。  
s2 = s0;  
  
// NG s0.str_とs2.str_は同じメモリを指すため、  
//     s0、s2のスコープアウト時に、~ShallowNG()により、2重解放される。
```

ディープコピー

ディープコピーとは、[シャローコピー](#)が発生させる問題を回避したコピーである。

以下に例を示す。

```
// @@@ example/term_explanation/deep_shallow_copy_ut.cpp 79  
  
class Deep {  
public:  
    explicit Deep(char const* str = "") : str_(new std::string{str}) {}  
    ~Deep() { delete str_; }  
    std::string const& GetString() const noexcept { return *str_; }  
  
    // copyコンストラクタの実装例  
    Deep(Deep const& rhs) : str_{new std::string{*rhs.str_}} {}  
  
    // copy代入演算子の実装例  
    Deep& operator=(Deep const& rhs)  
    {  
        *str_ = *(rhs.str_);  
        return *this;  
    }  
  
private:  
    std::string* str_;  
};  
  
class Deep2 { // std::unique_ptrを使いDeepをリファクタリング  
public:  
    explicit Deep2(char const* str = "") : str_{std::make_unique<std::string>(str)} {}  
    std::string const& GetString() const { return *str_; }  
  
    // copyコンストラクタの実装例  
    Deep2(Deep2 const& rhs) : str_{std::make_unique<std::string>(*rhs.str_)} {}  
  
    // copy代入演算子の実装例  
    Deep2& operator=(Deep2 const& rhs)  
    {  
        *str_ = *(rhs.str_);  
    }
```

```

        return *this;
    }

private:
    std::unique_ptr<std::string> str_;
};
```

上記クラスのDeepは、copyコンストラクタ、copy代入演算子でポインタをコピーするのではなく、ポインタが指しているオブジェクトを複製することにより、シャローコピーの問題を防ぐ。

スライシング

オブジェクトのスライシングとは、

- クラスBaseとその派生クラスDerived
- クラスDerivedのインスタンスd1、d2(解説のために下記例ではd0も定義)
- d2により初期化されたBase&型のd2_ref(クラスBase型のリファレンス)

が宣言されたとした場合、

```
d2_ref = d1; // オブジェクトの代入
```

を実行した時に発生するようなオブジェクトの部分コピーのことである(この問題はリファレンスをポインタに代えた場合にも起こる)。

以下のクラスと単体テストはこの現象を表している。

```
// @@@ example/term_explanation/slice_ut.cpp 10

class Base {
public:
    explicit Base(char const* name) noexcept : name0_{name} {}
    char const* Name0() const noexcept { return name0_; }

    ...
private:
    char const* name0_;
};

class Derived final : public Base {
public:
    Derived(char const* name0, char const* name1) noexcept : Base{name0}, name1_{name1} {}
    char const* Name1() const noexcept { return name1_; }

    ...
private:
    char const* name1_;
};

TEST(Slicing, reference)
{
    auto const d0      = Derived{"d0", "d0"};
    auto const d1      = Derived{"d1", "d1"};
    auto      d2      = Derived{"d2", "d2"};
    Base&    d2_ref = d2;

    ASSERT_STREQ("d2", d2.Name0()); // OK
    ASSERT_STREQ("d2", d2.Name1()); // OK

    d2 = d0;
    ASSERT_STREQ("d0", d2.Name0()); // OK
    ASSERT_STREQ("d0", d2.Name1()); // OK

    d2_ref = d1; // d2_refはBase&型で、d2へのリファレンス
    ASSERT_STREQ("d1", d2.Name0()); // OK
    ASSERT_STREQ("d0", d2.Name1()); // OK

#ifndef NDEBUG
    ASSERT_STREQ("d1", d2.Name1()); // 本来ならこうなってほしいが、
#else
    ASSERT_STREQ("d0", d2.Name1()); // スライシングの影響でDerived::name1_はコピーされない
#endif
}
```

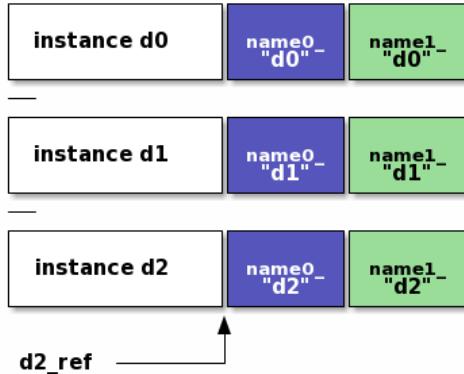
copy代入演算子(=)によりコピーが行われた場合、=の両辺のオブジェクトは等価になるべきだが(copy代入演算子をオーバーロードした場合も、そうなるように定義すべきである)、スライシングが起った場合、そうならないことが問題である(「[等価性のセマンティクス](#)」参照)。

下記にこの現象の発生メカニズムについて解説する。

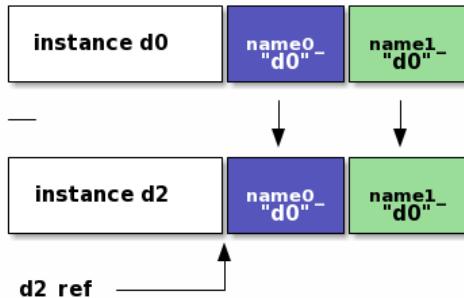
1. 上記クラスBase、Derivedのメモリ上のレイアウトは下記のようになる。



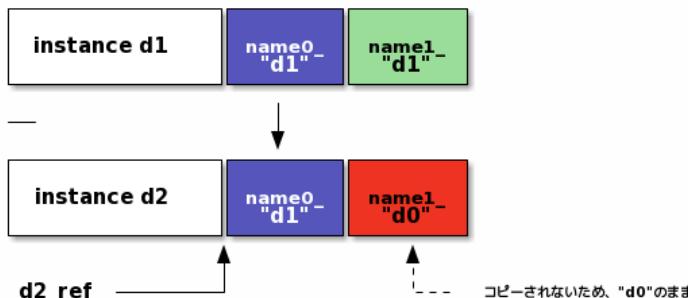
2. 上記インスタンスd0、d1、d2、d2_refのメモリ上のレイアウトは下記のようになる。



3. d2 = d0をした場合の状態は下記のようになる。



4. 上記の状態でd2_ref = d1をした場合の状態は下記のようになる。



d2.name1_ の値が元のままであるが(これがスライシングである)、その理由は下記の疑似コードが示す通り、「d2_refの表層型がクラスBaseであるためd1もクラスBase(正確にはBase型へのリファレンス)へ変換された後、d2_refが指しているオブジェクト(d2)へコピーされた」からである。

```
d2_ref.Base::operator=(d1); // Base::operator=(Base const&)が呼び出される
```

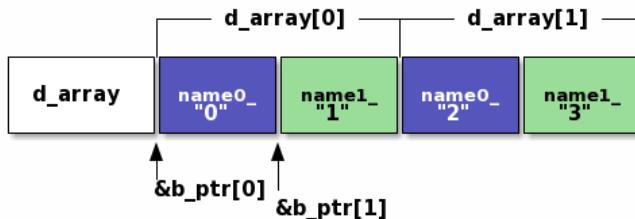
次に示すのは、「オブジェクトの配列をその基底クラスへのポインタに代入し、そのポインタを配列のように使用した場合に発生する」スライシングと類似の現象である。

```
// @@@ example/term_explanation/slice_ut.cpp 61

TEST(Slicing, array)
{
    Derived d_array[]{{"0", "1"}, {"2", "3"}};
    Base* b_ptr = d_array; // この代入までは問題ないが、b_ptr[1]でのアクセスで問題が起こる

    ASSERT_STREQ("0", d_array[0].Name0()); // OK
    ASSERT_STREQ("0", b_ptr[0].Name0()); // OK

    ASSERT_STREQ("2", d_array[1].Name0()); // OK
#ifndef 0 // スライシングに類似した問題で、以下のテストは失敗する。
    ASSERT_STREQ("2", b_ptr[1].Name0()); // NG
#else // こうすればテストは通るが、、、
    ASSERT_STREQ("1", b_ptr[1].Name0()); // NG
#endif
}
```



name lookupと名前空間

ここではname lookupとそれに影響を与える名前空間について解説する。

ルックアップ

このドキュメントでのルックアップとはname lookupを指す。

name lookup

name lookupとはソースコードで名前が検出された時に、その名前をその宣言と関連付けることである。以下、name lookupの例を上げる。

下記のようなコードがあった場合、

```
// @@@ example/term_explanation/name_lookup_ut.cpp 5

namespace NS_LU {
    int f() noexcept { return 0; }
} // namespace NS_LU
```

以下のコードでの関数呼び出しf()のname lookupは、

```
// @@@ example/term_explanation/name_lookup_ut.cpp 29

NS_LU::f();
```

1. NS_LUをその前方で宣言された名前空間と関連付けする
2. f()呼び出しをその前方の名前空間NS_LUで宣言された関数fと関連付ける

という手順で行われる。

下記のようなコードがあった場合、

```
// @@@ example/term_explanation/name_lookup_ut.cpp 11

namespace NS_LU {
    bool g(int i) noexcept { return i < 0; }

    char g(std::string_view str) noexcept { return str[0]; }
}
```

```
template <typename T, size_t N>
size_t g(T const (&)[N]) noexcept
{
    return N;
}
```

以下のコードでの関数呼び出し`g()`のname lookupは、

```
// @@@ example/term_explanation/name_lookup_ut.cpp 37
int a[3]{1, 2, 3};
NS_LU::g(a);
```

1. NS_LUをその前方で宣言された名前空間と関連付けする
2. 名前空間NS_LU内で宣言された複数の`g`を見つける
3. `g()`呼び出しを、すでに見つけた`g`の中からベストマッチした`g(T const (&)[N])`と関連付ける

という手順で行われる。

下記記のようなコードがあった場合、

```
// @@@ example/term_explanation/name_lookup_ut.cpp 44

// グローバル名前空間
std::string ToString(int i) { return std::to_string(i) + " in Global"; }

namespace NS_LU {
struct X {
    int i;
};

std::string ToString(X const& x) { return std::to_string(x.i) + " in NS_LU"; }
} // namespace NS_LU

namespace NS2 {
std::string ToString(NS_LU::X const& x) { return std::to_string(x.i) + " in NS2"; }
} // namespace NS2
```

以下のコードでの関数呼び出し`ToString()`のname lookupは、

```
// @@@ example/term_explanation/name_lookup_ut.cpp 65

auto x = NS_LU::X{1};

ASSERT_EQ("1 in NS_LU", ToString(x));
```

1. `ToString()`呼び出しの引数`x`の型`X`が名前空間`NS_LU`で定義されているため、`ToString`を探索する名前空間に`NS_LU`を組み入れる(「関連名前空間」参照)
2. `ToString()`呼び出しより前方で宣言されたグローバル名前空間と`NS_LU`の中から、複数の`ToString`の定義を見つける
3. `ToString()`呼び出しを、すでに見つけた`ToString`の中からベストマッチした`NS_LU::ToString`と関連付ける

という手順で行われる。

two phase name lookup

two phase name lookupとはテンプレートをインスタンス化するときに使用される、下記のような2段階でのname lookupである。

1. テンプレート定義内でname lookupを行う(通常のname lookupと同じ)。この時、テンプレートパラメータに依存した名前(dependent name)はname lookupの対象外となる(name lookupの対象が確定しないため)。
2. 1の後、テンプレートパラメータを展開した関数内で、関連名前空間の宣言も含めたname lookupを行う。

以下の議論では、

- 上記1のname lookupを1st name lookup
- 上記2のname lookupを2nd name lookup

と呼ぶことにする。

下記記のようなコードがあった場合、

```
// @@@ example/term_explanation/two_phase_name_lookup_ut.cpp 5

namespace NS_TPLU {
struct X {
    int i;
```

```

};

} // namespace NS_TPLU

// グローバル名前空間
inline std::string ToType(NS_TPLU::X const&) { return "X in global"; }
inline std::string ToType(int const&) { return "int in global"; }

// 再びNS_TPLU
namespace NS_TPLU {

std::string Header(long) { return "type:"; } // 下記にもオーバーロードあり

template <typename T>
std::string ToType(T const&) // 下記にもオーバーロードあり
{
    return "unknown";
}

template <typename T>
std::string TypeName(T const& t) // オーバーロードなし
{
    return Header(int{}) + ToType(t);
}

std::string Header(int) { return "TYPE:"; } // 上記にもオーバーロードあり

std::string ToType(X const&) { return "X"; } // 上記にもオーバーロードあり
std::string ToType(int const&) { return "int"; } // 上記にもオーバーロードあり
} // namespace NS_TPLU

```

以下のコードでのTypeNameのインスタンス化に伴うname lookupは、

```

// @@@ example/term_explanation/two_phase_name_lookup_ut.cpp 44

auto x = NS_TPLU::X{1};

ASSERT_EQ("type:X", TypeName(x));

```

1. TypeName()呼び出しの引数xの型Xが名前空間NS_TPLUで宣言されているため、 NS_TPLUをTypeNameを探索する関連名前空間にする。
2. TypeName()呼び出しそれ前方で宣言されたグローバル名前空間とNS_TPLUの中からTypeNameを見つける。
3. TypeNameは関数テンプレートであるためtwo phase lookupが以下のように行われる。
 1. TypeName内でのHeader(int{})の呼び出しは、1st name lookupにより、 Header(long)の宣言と関連付けられる。 Header(int)は Header(long)よりもマッチ率が高い、 TypeNameの定義より後方で宣言されているため、 name lookupの対象外となる。
 2. TypeName内でのToType(t)の呼び出しに対しては、2nd name lookupが行われる。このためTypeName定義より前方で宣言されたグローバル名前空間と、tの型がNS_TPLU::Xであるため関連名前空間となったNS_TPLUがname lookupの対象となるが、グローバル名前空間内のToTypeは、NS_TPLU内でTypeNameより前に宣言されたtemplate< T > ToTypeによってname-hidingが起こり、 TypeNameからは非可視となるためname lookupの対象から外れる。このため、 ToType(t)の呼び出しは、 NS_TPLU::ToType(X const&)の宣言と関連付けられる。

という手順で行われる。

上と同じ定義、宣言がある場合の以下のコードでのTypeNameのインスタンス化に伴うname lookupは、

```

// @@@ example/term_explanation/two_phase_name_lookup_ut.cpp 50

ASSERT_EQ("type:unknown", NS_TPLU::TypeName(int{}));

```

1. NS_TPLUを名前空間と関連付けする(引数の型がintなのでNS_TPLUは関連名前空間とならず、 NS_TPLUを明示する必要がある)。
2. TypeName()呼び出しそれ前方で宣言されたNS_TPLUの中からTypeNameを見つける。
3. TypeNameは関数テンプレートであるためtwo phase lookupが以下のように行われる。
 1. TypeName内でのHeader(int{})の呼び出しは、1st name lookupにより、前例と同じ理由で、 Header(long)の宣言と関連付けられる。
 2. TypeName内でのToType(t)の呼び出しに対しては、2nd name lookupが行われる。tの型がintであるためNS_TPLUは関連名前空間とならず、通常のname lookupと同様に ToType(t)の呼び出し前方のグローバル名前空間とNS_TPLUがname lookupの対象になるが、グローバル名前空間内のToTypeは、NS_TPLU内でTypeNameより前に宣言されたtemplate< T > ToTypeによってname-hidingが起こり、 TypeNameからは非可視となるためname lookupの対象から外れる。また、 ToType(int const&)は、 TypeNameの定義より後方で宣言されているため、 name lookupの対象外となり、その結果、 ToType(t)の呼び出しは、 NS_TPLU内のtemplate< T > ToTypeの宣言と関連付けられる。

という手順で行われる。

以上の理由から、先に示した例でのToTypeの戻り値は”X”となり、後に示した例でのToTypeの戻り値は”unknown”となる。これはtwo phase lookupの結果であり、two phase lookupが実装されていないコンパイラ(こういったコンパイラは存在する)では、結果が異なるため注意が必要である(本ドキュメントではこのような問題をできる限り避けるために、サンプルコードをg++とclang++でコンパイルしている)。

以下に、two phase lookupにまつわるさらに驚くべきコード例を紹介する。上と同じ定義、宣言がある場合の以下のコードの動作を考える。

```
// @@@ example/term_explanation/two_phase_name_lookup_ut.cpp 54
ASSERT_EQ("type:long", NS_TPLU::TypeName(long{}));
```

NS_TPLU::TypeName(int{})のintをlongにしただけなので、この単体テストはパスしないが、この単体テストコードの後(実際にはこのファイルのコンパイル単位の中のNS_TPLU内で、且つtemplate<‐> ToTypeの宣言の後方であればどこでもよい)に以下のコードを追加するとパスしてしまう。

```
// @@@ example/term_explanation/two_phase_name_lookup_ut.cpp 61
namespace NS_TPLU {
template <‐>
std::string ToType<long>(long const&)
{
    return "long";
} // namespace NS_TPLU
```

この理由は、関数テンプレート内での2nd name lookupで選択された名前が関数テンプレートであった場合、その特殊化の検索範囲はコンパイル単位内になることがあるからである([template specialization](#)によるとこの動作は未定義のようだが、g++/clang++両方ともこのコードを警告なしでコンパイルする)。

TypeName(long{})内でのtwo phase name lookupは、TypeName(int{})とほぼ同様に進み、template<‐> ToTypeの宣言を探し出すが、さらに前述したようにこのコンパイル単位のNS_TPLU内からその特殊化も探し出す。その結果、ToType(t)の呼び出しが、NS_TPLU内のtemplate<‐> ToType<long>の定義と関連付けられる。

以上の議論からわかる通り、関数テンプレートとその特殊化の組み合わせは、そのインスタンス化箇所(この場合単体テストコード内)の後方から、name lookupでバインドされる関数を変更することができるため、極めて分かりづらいコードを生み出す。ここから、

- 関数テンプレートとその特殊化はソースコード上なるべく近い位置で宣言するべきである
- STL関数テンプレートの特殊化は行うべきではない

という教訓が得られる。

なお、関数とその関数オーバーロードのname lookupの対象は、呼び出し箇所前方の宣言のみであるため、関数テンプレートToType(T const& t)の代わりに、関数ToType(...)を使うことで、上記問題は回避可能である。

次に示す例は、一見2nd name lookupで関連付けされるように見える関数ToType(NS_TPLU2::Y const&)が、実際には関連付けされないコードである。

```
// @@@ example/term_explanation/two_phase_name_lookup_ut.cpp 71
namespace NS_TPLU2 {
struct Y {
    int i;
};
} // namespace NS_TPLU2
```

```
// @@@ example/term_explanation/two_phase_name_lookup_ut.cpp 79
// global名前空間
template <typename T>
std::string ToType(T const&)
{
    return "unknown";
}

template <typename T>
std::string TypeName(T const& t)
{
    return "type:" + ToType(t);
}

std::string ToType(NS_TPLU2::Y const&) { return "Y"; }
```

これは先に示したNS_TPLU::Xの例と極めて似ている。本質的な違いは、TypeNameやToTypeがグローバル名前空間で宣言されていることのみである。だが、下記の単体テストで示す通り、TypeName内でのname lookupで関数オーバーライドToType(NS_TPLU2::Y const&)が選択されないのである。

```
// @@@ example/term_explanation/two_phase_name_lookup_ut.cpp 100
auto y = NS_TPLU2::Y{1};

// ASSERT_EQ("type:Y", TypeName(y));
ASSERT_EQ("type:unknown", TypeName(y)); // ToType(NS_TPLU2::Y const&)は使われない
```

ここまで現象を正確に理解するには、「two phase lookupの対象となる宣言」を下記のように、より厳密に認識する必要がある。

- TypeNameの中で行われる1st name lookupの対象となる宣言は下記の積集合である。
 - TypeNameと同じ名前空間内かグローバル名前空間内の宣言
 - TypeName定義位置より前方の宣言
- TypeNameの中で行われる2nd name lookupの対象となる宣言は下記の和集合である。
 - 1st name lookupで使われた宣言
 - TypeName呼び出しより前方にある関連名前空間内の宣言

この認識に基づくNS_TPLU2::Yに対するグローバルなTypeName内でtwo phase name lookupは、

1. TypeName内に1st name lookupの対象がないため何もしない。
2. TypeName内の2nd name lookupに使用される関連名前空間NS_TPLU2は、ToType(NS_TPLU2::Y const&)の宣言を含まないため、この宣言は2nd name lookupの対象とならない。その結果、ToType(t)の呼び出しが関数テンプレートToType(T const&)と関連付けられる。

という手順で行われる。

以上が、TypeNameからToType(NS_TPLU2::Y const&)が使われない理由である。

ここまで示したようにtwo phase name lookupは理解しがたく、理解したとしてもその使いこなしはさらに難しい。

次のコードは、この難解さに翻弄されるのが現場のプログラマのみではないことを示す。

```
// @@@ example/term_explanation/two_phase_name_lookup_ut.cpp 71
namespace NS_TPLU2 {
struct Y {
    int i;
};
} // namespace NS_TPLU2

// @@@ example/term_explanation/two_phase_name_lookup_ut.cpp 110
// global名前空間
template <typename T>
int operator+(T const&, int i)
{
    return i;
}

template <typename T>
int TypeNum(T const& t)
{
    return t + 0;
}

int operator+(NS_TPLU2::Y const& y, int i) { return y.i + i; }
```

上記の宣言、定義があった場合、operator+の単体テストは以下のようになる。

```
// @@@ example/term_explanation/two_phase_name_lookup_ut.cpp 132
auto y = NS_TPLU2::Y{1};

ASSERT_EQ(1, y + 0); // 2つ目のoperator+が選択される
```

このテストは当然パスするが、次はどうだろう？

```
// @@@ example/term_explanation/two_phase_name_lookup_ut.cpp 142
auto y = NS_TPLU2::Y{1};

ASSERT_EQ(1, TypeNum(y)); // g++ではoperator+(NS_TPLU2::Y const&, int i)がname lookupされる
```

これまでのtwo phase name lookupの説明では、operator+(NS_TPLU2::Y const&, int i)はTypeNum内でname lookupの対象にはならないため、このテストはエラーとならなければならないが、g++ではパスしてしまう。2nd name lookupのロジックにバグがあるようである。

有難いことに、clang++では仕様通りこのテストはエラーとなり、当然ながら以下のテストはパスする(つまり、g++ではエラーする)。

```
// @@@ example/term_explanation/two_phase_name_lookup_ut.cpp 151
auto y = NS_TPLU2::Y{1};

ASSERT_EQ(0, TypeNum(y)); // clang++ではoperator+(T const&, int i)がname lookupされる
```

なお、TypeNum内のコードである

```
return t + 0;
```

を下記のように変更することで

```
return operator+(t, 0);
```

g++のname lookupはclang++と同じように動作するため、記法に違和感があるものの、この方法はg++のバグのワークアランドとして使用できる。

また、operator+(NS_TPLU2::Y const& y, int i)をNS_TPLU2で宣言することで、g++ではパスしたテストをclang++でもパスさせられるようになる(これは正しい動作)。これにより、型とその2項演算子オーバーロードは同じ名前空間で宣言するべきである、という教訓が得られる。

以上で見てきたようにtwo phase name lookupは、現場プログラマのみではなく、コンパイラを開発するプログラマをも混乱させるほど難解ではあるが、STLを含むテンプレートメタプログラミングを支える重要な機能であるため、C++プログラマには、最低でもこれを理解し、出来れば使いこなせるようになってほしい。

実引数依存探索

実引数依存探索とは、argument-dependent lookupの和訛語であり、通常はその略語であるADLと呼ばれる。

ADL

ADLとは、関数の実引数の型が宣言されている名前空間(これを関連名前空間と呼ぶ)内の宣言が、その関数のname lookupの対象になることがある。

下記のようなコードがあった場合、

```
// @@@ example/term_explanation/name_lookup_adl_ut.cpp 5
namespace NS_ADL {
struct A {
    int i;
};

std::string ToString(A const& a) { return std::string("A:") + std::to_string(a.i); }
} // namespace NS_ADL
```

以下のコードでのToStringの呼び出しに対するname lookupは、

```
// @@@ example/term_explanation/name_lookup_adl_ut.cpp 18
auto a = NS_ADL::A{0};

ASSERT_EQ("A:0", ToString(a)); // ADLの効果により、ToStringはNS_ADLを指定しなくても見つかる
```

- ToStringの呼び出しより前方で行われているグローバル名前空間内の宣言
- ToStringの呼び出しより前方で行われているNS_ADL内の宣言

の両方を対象として行われる。NS_ADL内の宣言がToStringの呼び出しに対するname lookupの対象になる理由は、ToStringの呼び出しに使われている実引数aの型AがNS_ADLで宣言されているからである。すでに述べたようにこれをADLと呼び、この場合のNS_ADLを関連名前空間と呼ぶ。

ADLは思わずname lookupによるバグを誘発することもあるが、下記コードを見れば明らかのように、また、多くのプログラマはそれと気づかずを使っていることからもわかる通り、コードをより自然に、より簡潔に記述するための重要な機能となっている。

```
// @@@ example/term_explanation/name_lookup_adl_ut.cpp 28
// 下記operator <<は、std::operator<<(ostream&, string const&)であり、
// namespace stdで定義されている。
// ADLがあるため、operator <<は名前空間修飾無しで呼び出せる。
std::cout << std::string{_func_};

// ADLが無いと下記のような呼び出しになる。
std::operator<<(std::cout, std::string{_func_});
```

関連名前空間

関連名前空間(associated namespace)とは、[ADL\(実引数依存探索\)](#)によってname lookupの対象になった宣言を含む名前空間のことである。

SFINAE

SFINAE (Substitution Failure Is Not An Errorの略称、スフィネエと読む)とは、「テンプレートのパラメータ置き換えに失敗した(ill-formedになつた)際に、即時にコンパイルエラーとはせず、置き換えに失敗したテンプレートを name lookupの候補から除外する」という言語機能である。

name-hiding

name-hidingとは「前方の識別子が、その後方に同一の名前をもつ識別子があるために、name lookupの対象外になる」現象一般を指す通称である([namespace](#)参照)。

まずは、クラスとその派生クラスでのname-hidingの例を示す。

```
// @@@ example/term_explanation/name_hiding.cpp 4

struct Base {
    void f() noexcept {};
};

struct Derived : Base {
    // void f(int) { f(); }      // f()では、Baseのf()をname lookupできないため、
    void f(int) noexcept { Base::f(); } // Base::f()を修飾した
};
```

上記の関数は一見オーバーロードに見えるが、そうではない。下記のコードで示したように、Base::f()には、修飾しない形式でのDerivedクラス経由のアクセスはできない。

```
// @@@ example/term_explanation/name_hiding.cpp 18

{
    auto d = Derived{};
#if 0
    d.f(); // コンパイルできない
#else
    d.Base::f(); // Base::での修飾が必要
#endif
}
```

これは前述したように、Base::f()がその後方にあるDerived::f(int)によりname-hidingされたために起こる現象である(name lookupによる探索には識別子が使われるため、シグネチャの違いはname-hidingに影響しない)。

下記のように[using宣言](#)を使用することで、修飾しない形式でのDerivedクラス経由のBase::f()へのアクセスが可能となる。

```
// @@@ example/term_explanation/name_hiding.cpp 34

struct Derived : Base {
    using Base::f; // using宣言によりDerivedにBase::fを導入
    void f(int) noexcept { Base::f(); }
};

// @@@ example/term_explanation/name_hiding.cpp 45

auto d = Derived{};
d.f(); // using宣言によりコンパイルできる
```

下記コードは、名前空間でも似たような現象が起こることを示している。

```
// @@@ example/term_explanation/name_hiding.cpp 54

// global名前空間
void f() noexcept {}

namespace NS_A {
void f(int) noexcept {}

void g() noexcept
{
#if 0
    f(); // NS_A::fによりname-hidingされたため、コンパイルできない
#endif
}
```

```
}
```

```
// namespace NS_A
```

この問題に対しては、下記のようにf(int)の定義位置を後方に移動することで回避できる。

```
// @@@ example/term_explanation/name_hiding.cpp 70
```

```
namespace NS_A_fixed_0 {
void g() noexcept
{
    // グローバルなfの呼び出し
    f(); // NS_A::fは後方に移動されたためコンパイルできる
}

void f(int) noexcept {}
} // namespace NS_A_fixed_0
```

また、先述のクラスでの方法と同様にusing宣言を使い、下記のようにすることもできる。

```
// @@@ example/term_explanation/name_hiding.cpp 82
```

```
namespace NS_A_fixed_1 {
void f(int) noexcept {}

void g() noexcept
{
    using ::f;

    // グローバルなfの呼び出し
    f(); // using宣言によりコンパイルできる
}
} // namespace NS_A_fixed_1
```

当然ながら、下記のようにf()の呼び出しを::で修飾することもできる。

```
// @@@ example/term_explanation/name_hiding.cpp 96
```

```
namespace NS_A_fixed_2 {
void f(int) noexcept {}

void g() noexcept
{
    // グローバルなfの呼び出し
    ::f(); // ::で修飾すればコンパイルできる
}
} // namespace NS_A_fixed_2
```

修飾の副作用として「[two phase name lookup](#)」の例で示したような ADLを利用した高度な静的ディスパッチが使用できなくなるが、通常のソフトウェア開発では、ADLが必要な場面は限られているため、デフォルトでは名前空間を使用して修飾を行うことにするのが、無用の混乱をさけるための安全な記法であると言えるだろう。

次に、そういうた混乱を引き起こすであろうコードを示す。

```
// @@@ example/term_explanation/name_hiding.cpp 108
```

```
namespace NS_B {
struct S_in_B {};

void f(S_in_B) noexcept {}
void f(int) noexcept {}

namespace NS_B_Inner {
void g() noexcept
{
    f(int{}); // コンパイルでき、NS_B::f(int)が呼ばれる
}

void f() noexcept {}

void h() noexcept
{
    // f(int{}); // コンパイルできない
    NS_B::f(int{}); // 名前空間で修飾することでコンパイルできる

    f(S_in_B{}); // ADLによりコンパイルできる
}
} // namespace NS_B_Inner
} // namespace NS_B
```

NS_B_Inner::g()内のf(int)の呼び出しはコンパイルできるが、name-hidingが原因で、NS_B_Inner::h()内のf(int)の呼び出しはコンパイルできず、名前空間で修飾することが必要になる。一方で、ADLの効果で名前空間での修飾をしていないf(S_in_B)の呼び出しはコンパイルできる。

全チームメンバーがこういったname lookupを正しく扱えると確信できないのであれば、前述の通り、デフォルトでは名前空間を使用して修飾を行うのが良いだろう。

ドミナンス

ドミナンス(Dominance、支配性)とは、「探索対称の名前が継承の中にも存在するような場合のname lookupの仕様の一部」を指す慣用句である。

以下に

- ダイヤモンド継承を含まない場合
- ダイヤモンド継承かつそれが仮想継承でない場合
- ダイヤモンド継承かつそれが仮想継承である場合

のドミナンスについてのコードを例示する。

この例で示したように、ダイヤモンド継承を通常の継承で行うか、仮想継承で行うかでは結果が全く異なるため、注意が必要である。

ダイヤモンド継承を含まない場合

```
// @@@ example/term_explanation/dominance_ut.cpp 9

int32_t f(double) noexcept { return 0; }

struct Base {
    int32_t f(int32_t) const noexcept { return 1; }
    int32_t f(double) const noexcept { return 2; }
};

struct Derived : Base {
    int32_t f(int32_t) const noexcept { return 3; } // Base::fを隠蔽する(name-hiding)
};

struct DerivedDerived : Derived {
    int32_t g() const noexcept { return f(2.14); }
};
```

```
// @@@ example/term_explanation/dominance_ut.cpp 29

Base b;

ASSERT_EQ(2, b.f(2.14)); // オーバーロード解決により、B::f(double)が呼ばれる

DerivedDerived dd;

// Derivedのドミナンスにより、B::fは、DerivedDerived::gでのfのname lookupの対象にならず、
// DerivedDerived::gはDerived::fを呼び出す。
ASSERT_EQ(3, dd.g());
```

このname lookupについては、name-hidingで説明した通りである。

ダイヤモンド継承かつそれが仮想継承でない場合

```
// @@@ example/term_explanation/dominance_ut.cpp 45

struct Base {
    int32_t f(int32_t) const noexcept { return 1; }
    int32_t f(double) const noexcept { return 2; }
};

struct Derived_0 : Base {
    int32_t f(int32_t) const noexcept { return 3; } // Base::fを隠蔽する(name-hiding)
};

struct Derived_1 : Base {};

struct DerivedDerived : Derived_0, Derived_1 {
    int32_t g() const noexcept { return f(2.14); } // Derived_0::f or Derived_1::f ?
};
```

```
// dominance_ut.cpp:58:41: error: reference to 'f' is ambiguous
//   58 |     int32_t g() const noexcept { return f(2.14); } // Derived_0::f or Derived_1::f ?
//   |
```

上記コードはコードブロック内のコメントのようなメッセージが原因でコンパイルできない。

Derived_0のドミナンスにより、DerivedDerived::gはDerived_0::fを呼び出すように見えるが、もう一つの継承元であるDerived_1が導入したDerived_1::f(実際には、Derived_1::Base::f)があるため、Derived_1によるドミナンスも働き、その結果として、呼び出しが曖昧(ambiguous)になることで、このような結果となる。

ダイヤモンド継承かつそれが仮想継承である場合

```
// @@@ example/term_explanation/dominance_ut.cpp 71

struct Base {
    int32_t f(int32_t) const noexcept { return 1; }
    int32_t f(double) const noexcept { return 2; }
};

struct Derived_0 : virtual Base {
    int32_t f(int32_t) const noexcept { return 3; } // Base::fを隠蔽する(name-hiding)
};

struct Derived_1 : virtual Base {};

struct DerivedDerived : Derived_0, Derived_1 {
    int32_t g() const noexcept { return f(2.14); }
};
```

```
// @@@ example/term_explanation/dominance_ut.cpp 92
```

```
DerivedDerived dd;

// Derived_0のドミナンスと仮想継承の効果により、
// B::fは、DerivedDerived::gでのfのname lookupの対象にならず、
// DerivedDerived::gはDerived_0::fを呼び出す。
ASSERT_EQ(3, dd.g());
```

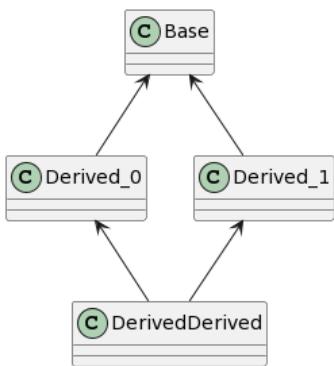
これまでと同様にDerived_0のドミナンスによりBase::fはname-hidingされることになる。この時、Derived_0、Derived_1がBaseから仮想継承した効果により、この継承ヒエラルキーの中でBaseは1つのみ存在することになるため、Derived_1により導入されたBase::fも併せてname-hidingされる。結果として、曖昧性は排除され、コンパイルエラーにはならず、このような結果となる。

ダイヤモンド継承

ダイヤモンド継承(Diamond Inheritance)とは、以下のような構造のクラス継承を指す。

- 基底クラス(Base)が一つ存在し、その基底クラスから二つのクラス(Derived_0、Derived_1)が派生する。
- Derived_0とDerived_1からさらに一つのクラス(DerivedDerived)が派生する。したがって、DerivedDerivedはBaseの孫クラスとなる。

この継承は、多重継承の一形態であり、クラス図で表すと下記のようになるため、ダイヤモンド継承と呼ばれる。



ダイヤモンド継承は、仮想継承(virtual inheritance)を使ったものと、使わないものに分類できる。

仮想継承を使わないダイヤモンド継承のコードを以下に示す。

```
// @@@ example/term_explanation/diamond_inheritance_ut.cpp 6
```

```
class Base {
public:
```

```

int32_t get() const noexcept { return x_; }
void set(int32_t x) noexcept { x_ = x; }

private:
    int32_t x_ = 0;
};

class Derived_0 : public Base {};
class Derived_1 : public Base {};

class DerivedDerived : public Derived_0, public Derived_1 {};

// @@@ example/term_explanation/diamond_inheritance_ut.cpp 26

auto dd = DerivedDerived{};

Base& b0 = static_cast<Derived_0&>(dd); // Derived_0::Baseのリファレンス
Base& b1 = static_cast<Derived_1&>(dd); // Derived_1::Baseのリファレンス

ASSERT_NE(&b0, &b1); // ddの中には、Baseインスタンスが2つできる

```

これからわかるように、DerivedDerivedインスタンスの中に2つのBaseインスタンスが存在する。

下記コードは、それが原因で名前解決が曖昧になりコンパイルできない。

```

// @@@ example/term_explanation/diamond_inheritance_ut.cpp 36

Base& b = dd; // Derived_0::Base or Derived_1::Base ?

dd.get(); // Derived_0::get or Derived_1::get ?

// 下記のようなエラーが発生する
// diamond_inheritance_ut.cpp:37:15: error: 'Base' is an ambiguous base of 'DerivedDerived'
//   37 |     Base& b = dd; // Derived_0::Base or Derived_1::Base ?
//   |           ^
// diamond_inheritance_ut.cpp:39:8: error: request for member 'get' is ambiguous
//   39 |     dd.get(); // Derived_0::get or Derived_1::get ?
//   |           ^

```

この問題に対処するには、クラス名による修飾が必要になるが、Baseインスタンスが2つ存在するため、下記に示すようなわかりづらいバグの温床となる。

```

// @@@ example/term_explanation/diamond_inheritance_ut.cpp 53

ASSERT_EQ(0, dd.Derived_0::get()); // クラス名による名前修飾
ASSERT_EQ(0, dd.Derived_1::get());

dd.Derived_0::set(1);
ASSERT_EQ(1, dd.Derived_0::get()); // Derived_0::Base::x_(は1に変更
ASSERT_EQ(0, dd.Derived_1::get()); // Derived_1::Base::x_(は0のまま

dd.Derived_1::set(2);
ASSERT_EQ(1, dd.Derived_0::get()); // Derived_0::Base::x_(は1のまま
ASSERT_EQ(2, dd.Derived_1::get()); // Derived_1::Base::x_(は2に変更

```

次に示すのは、仮想継承を使用したダイヤモンド継承の例である。

```

// @@@ example/term_explanation/diamond_inheritance_ut.cpp 70

class Base {
public:
    int32_t get() const noexcept { return x_; }
    void set(int32_t x) noexcept { x_ = x; }

private:
    int32_t x_ = 0;
};

class Derived_0 : public virtual Base {} // 仮想継承

class Derived_1 : public virtual Base {} // 仮想継承

class DerivedDerived : public Derived_0, public Derived_1 {};

// @@@ example/term_explanation/diamond_inheritance_ut.cpp 90

auto dd = DerivedDerived{};

Base& b0 = static_cast<Derived_0&>(dd); // Derived_0::Baseのリファレンス

```

```
Base& b1 = static_cast<Derived_1>(dd); // Derived_1::Baseのリファレンス
ASSERT_EQ(&b0, &b1); // ddの中には、Baseインスタンスが1つできる
```

仮想継承の効果で、DerivedDerivedインスタンスの中に存在するBaseインスタンスは1つになるため、上で示した仮想継承を使わないダイヤモンド継承での問題は解消される(が、仮想継承による別の問題が発生する)。

```
// @@@ example/term_explanation/diamond_inheritance_ut.cpp 99
Base& b = dd; // Baseインスタンスは1つであるため、コンパイルできる
dd.get(); // Baseインスタンスは1つであるため、コンパイルできる
dd.Derived_0::set(1); // クラス名による修飾
ASSERT_EQ(1, dd.Derived_1::get()); // Derived_1::BaseとDerived_1::Baseは同一であるため
dd.set(2);
ASSERT_EQ(2, dd.get());
```

仮想継承

下記に示した継承方法を仮想継承、仮想継承の基底クラスを仮想基底クラスと呼ぶ。

```
// @@@ example/term_explanation/virtual_inheritance_ut.cpp 9
class Base {
public:
    explicit Base(int32_t x = 0) noexcept : x_{x} {}
    int32_t get() const noexcept { return x_; }

private:
    int32_t x_;
};

class DerivedVirtual : public virtual Base { // 仮想継承
public:
    explicit DerivedVirtual(int32_t x) noexcept : Base{x} {}
};
```

仮想継承は、ダイヤモンド継承の基底クラスのインスタンスを、その継承ヒエラルキーの中で1つのみにするための言語機能である。

仮想継承の独特的動作を示すため、上記コードに加え、仮想継承クラス、通常の継承クラス、それぞれを通常の継承したクラスを下記のように定義する。

```
// @@@ example/term_explanation/virtual_inheritance_ut.cpp 25
class DerivedDerivedVirtual : public DerivedVirtual { // 仮想継承を通常の継承
public:
    explicit DerivedDerivedVirtual(int32_t x) noexcept : DerivedVirtual{x} {}

class DerivedNormal : public Base { // 通常の継承
public:
    explicit DerivedNormal(int32_t x) noexcept : Base{x} {}

class DerivedDerivedNormal : public DerivedNormal { // 通常の継承を通常の継承
public:
    explicit DerivedDerivedNormal(int32_t x) noexcept : DerivedNormal{x} {}
```

この場合、継承ヒエラルキーに仮想継承を含むクラスと、含まないクラスでは、以下に示したような違いが発生する。

```
// @@@ example/term_explanation/virtual_inheritance_ut.cpp 45
auto dv = DerivedVirtual{1}; // 仮想継承クラス
auto dn = DerivedNormal{1}; // 通常の継承クラス

ASSERT_EQ(1, dv.get()); // これは非仮想継承と同じ動作
ASSERT_EQ(1, dn.get());

auto ddv = DerivedDerivedVirtual{1}; // 仮想継承クラスを継承したクラス
auto ddn = DerivedDerivedNormal{1}; // 通常の継承クラスを継承したクラス

ASSERT_EQ(0, ddv.get()); // Baseのデフォルトコンストラクタが呼ばれる
ASSERT_EQ(1, ddn.get());
```

これは、「仮想継承クラスを継承したクラスが、仮想継承クラスの基底クラスのコンストラクタを明示的に呼び出さない場合、引数なしで呼び出せる基底クラスのコンストラクタが呼ばれる」仕様に起因している（引数なしで呼び出せる基底クラスのコンストラクタがない場合はコンパイルエラー）。以下では、これを「仮想継承のコンストラクタ呼び出し」仕様と呼ぶことにする。

仮想継承クラスが、基底クラスのコンストラクタを呼び出したとしても、この仕様が優先されるため、上記コードのような動作となる。

これを通常の継承クラスと同様な動作にするには、下記のようにしなければならない。

```
// @@@ example/term_explanation/virtual_inheritance_ut.cpp 61

class DerivedDerivedVirtualFixed : public DerivedVirtual { // DerivedDerivedNormalと同じように動作
public:
    explicit DerivedDerivedVirtualFixed(int32_t x) noexcept : Base{x}, DerivedVirtual{x} {}
    //                                     基底クラスのコンストラクタ呼び出し ^^^^^^
};

// @@@ example/term_explanation/virtual_inheritance_ut.cpp 72

DerivedDerivedVirtual ddv{1}; // 仮想継承クラスを継承したクラス
DerivedDerivedVirtualFixed ddvf{1}; // 上記クラスのコンストラクタを修正したクラス
DerivedDerivedNormal ddn{1}; // 通常の継承クラスを継承したクラス

ASSERT_EQ(0, ddv.get()); // 仮想継承独特の動作
ASSERT_EQ(1, ddvf.get());
ASSERT_EQ(1, ddn.get());
```

「仮想継承のコンストラクタ呼び出し」仕様は、ダイヤモンド継承での基底クラスのコンストラクタ呼び出しを一度にするために存在する。

もし、この機能がなければ、下記のコードでの基底クラスのコンストラクタ呼び出しは2度になるため、デバッグ困難なバグが発生してしまうことは容易に想像できるだろう。

```
// @@@ example/term_explanation/virtual_inheritance_ut.cpp 87

int32_t base_called;

class Base {
public:
    explicit Base(int32_t x = 0) noexcept : x_{x} { ++base_called; }
    int32_t get() const noexcept { return x_; }

private:
    int32_t x_;
};

class Derived_0 : public virtual Base { // 仮想継承
public:
    explicit Derived_0(int32_t x) noexcept : Base{x} { assert(base_called == 1); }
};

class Derived_1 : public virtual Base { // 仮想継承
public:
    explicit Derived_1(int32_t x) noexcept : Base{x} { assert(base_called == 1); }
};

class DerivedDerived : public Derived_0, public Derived_1 {
public:
    DerivedDerived(int32_t x0, int32_t x1) noexcept : Derived_0{x0}, Derived_1{x1} {}
    // 「仮想継承のコンストラクタ呼び出し」仕様がなければ、このコンストラクタは、
    //   Base::Base -> Derived_0::Derived_0 ->
    //   Base::Base -> Derived_0::Derived_0 ->
    //   DerivedDerived::DerivedDerived
    // という呼び出しになるため、Base::Baseが2度呼び出されてしまう。
};
```

```
// @@@ example/term_explanation/virtual_inheritance_ut.cpp 123

ASSERT_EQ(0, base_called);

auto dd = DerivedDerived{2, 3}; // Base::Baseが最初に呼ばないとassertion failする

ASSERT_EQ(1, base_called); // 「仮想継承のコンストラクタ呼び出し」仕様のため
ASSERT_EQ(0, dd.get()); // Baseのデフォルトコンストラクタは、x_を0にする
```

基底クラスのコンストラクタ呼び出しは、下記のコードのようにした場合でも、単体テストが示すように、一番最初に行われる。

```
// @@@ example/term_explanation/virtual_inheritance_ut.cpp 138
```

```

class DerivedDerived : public Derived_0, public Derived_1 {
public:
    DerivedDerived(int32_t x0, int32_t x1) noexcept : Derived_0{x0}, Derived_1{x1}, Base{1} {}
};

// @@@ example/term_explanation/virtual_inheritance_ut.cpp 150

ASSERT_EQ(0, base_called);

auto dd = DerivedDerived{2, 3}; // Base::Baseが最初に呼ばれないとassertion failする

ASSERT_EQ(1, base_called); // 「仮想継承のコンストラクタ呼び出し」仕様のため
ASSERT_EQ(1, dd.get()); // Base{1}呼び出しの効果

```

このため、基底クラスのコンストラクタ呼び出しは下記のような順番で行うべきである。

```

// @@@ example/term_explanation/virtual_inheritance_ut.cpp 163

class DerivedDerived : public Derived_0, public Derived_1 {
public:
    DerivedDerived(int32_t x0, int32_t x1) noexcept : Base{1}, Derived_0{x0}, Derived_1{x1} {}
};

```

仮想基底

仮想基底(クラス)とは、仮想継承の基底クラス指す。

using宣言

using宣言とは、“using XXX::func”のような記述である。この記述が行われたスコープでは、この記述後の行から名前空間XXXでの修飾をすることなく、funcが使用できる。

```

// @@@ example/term_explanation/namespace_ut.cpp 6
namespace XXX {
void func() noexcept {}
void gunc() noexcept {}
} // namespace XXX

// @@@ example/term_explanation/namespace_ut.cpp 12

// global namespace
void using_declaration() noexcept
{
    using XXX::func; // using宣言

    func(); // XXX::不要
    XXX::gunc(); // XXX::必要
}

```

usingディレクティブ

usingディレクティブとは、“using namespace XXX”のような記述である。この記述が行われたスコープでは、下記例のように、この記述後から名前空間XXXでの修飾をすることなく、XXXの識別子が使用できる。

```

// @@@ example/term_explanation/namespace_ut.cpp 6
namespace XXX {
void func() noexcept {}
void gunc() noexcept {}
} // namespace XXX

// @@@ example/term_explanation/namespace_ut.cpp 24

// global namespace
void using_directive() noexcept
{
    using namespace XXX; // usingディレクティブ

    func(); // XXX::不要
    gunc(); // XXX::不要
}

```

より多くの識別子が名前空間の修飾無しで使えるようになる点において、using宣言よりも危険であり、また、下記のようにname-hidingされた識別子の導入には効果がない。

```

// @@@ example/term_explanation/namespace_ut.cpp 6
namespace XXX {

```

```

void func() noexcept {}
void gunc() noexcept {}
} // namespace XXX

// @@@ example/term_explanation/namespace_ut.cpp 35

namespace XXX_Inner {
void func(int) noexcept {}
void using_declaration() noexcept
{
#if 0
    using namespace XXX; // name-hidingのため効果がない
#else
    using XXX::func; // using宣言
#endif

    func(); // XXX::不要
}

```

従って、usingディレクティブの使用は避けるべきである。

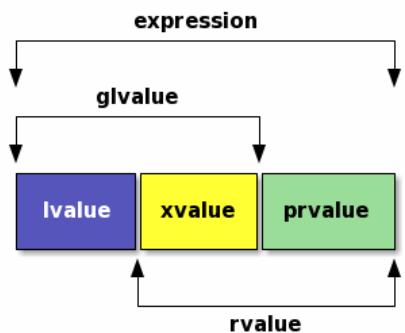
expressionと値カテゴリ

ここでは、expression(式)の値カテゴリや、それに付随した機能についての解説を行う。

expression

C++においてexpression、lvalue、rvalue、xvalue、glvalue、prvalueは以下のように定められている。

- expression(式)とは「演算子とそのオペランドの並び」である(オペランドのみの記述も式である)。演算子とは以下のようなものである。
 - 四則演算、代入(a = b、 a += b …)、インクリメント、比較、論理式
 - 明示的キャストや型変換
 - メンバアクセス(a.b、 a->b、 a[x]、 *a、 &a …)
 - 関数呼び出し演算子(f(...))、sizeof、decltype等
- expressionは、以下のいずれかに分類される。lvalueでないexpressionがrvalueである。
 - lvalue
 - rvalue
- lvalueとは、関数もしくはオブジェクトを指す。
- rvalueは、以下のいずれかに分類される。
 - xvalue
 - prvalue
- xvalueとは以下のようなものである。
 - 戻り値の型がT&&(Tは任意の型)である関数の呼び出し式(std::move(x))
 - オブジェクトへのT&&へのキャスト式(static_cast<char&&>(x))
 - aを配列のrvalueとした場合のa[N]や、cをクラス型のrvalueとした場合のc.m(mはaの非staticメンバ)等
- prvalueとは、オブジェクトやビットフィールドを初期化する、もしくはオペランドの値を計算する式であり、以下のようなものである。
 - 文字列リテラルを除くリテラル
 - 戻り値の型が非リファレンスである関数呼び出し式、または前置++と前置-を除くオーバーロードされた演算子式(path.string()、str1 + str2、it++)…)
 - 組み込み型インスタンスaのa++、a--(a++、a--はlvalue)
 - 組み込み型インスタンスa、bに対するa + b、a % b、a & b、a && b、a || b、!a、a < b、a == b等
- glvalueは、以下のいずれかに分類される。
 - lvalue
 - xvalue



ざっくりと言えば、lvalueとは代入式の左辺になり得る式、rvalueとは代入式の左辺にはなり得ない式である。T const&は左辺にはなり得ないが、lvalueである。rvalueリファレンス(T&&)もlvalueであるため、rvalueであることとrvalueリファレンスであることとは全く異なる。

xvalueとは、多くの場合、「std::move()の呼び出し式のことである」と考へても差し支えない。

prvalueとは、いわゆるテンポラリオブジェクトのことであるため(下記のstd::string()で作られるようなオブジェクト)、名前はない。また、アドレス演算子(&)のオペランドになれない。

```
// @@@ example/term_explanation/rvalue_lvalue_ut.cpp 8

{
    // sを初期化するためにstd::string{}により生成されるオブジェクトはprvalue
    // sはlvalue

    auto s = std::string{};

#ifndef 0
    // 下記はコンパイルエラー

    auto* sp = &std::string{};
    // 下記はg++のエラーメッセージ
    // programming_convention_type.cpp|709 col 29| error: taking address of rvalue [-fpermissive]
    // || 709 |     auto* sp = &std::string{};

#else
    // 下記のようにすればアドレスを取得できるが、このようなことはすべきではない。
    auto&& rvalue_ref = std::string{};
    auto    sp        = &rvalue_ref;
#endif
    static_assert(std::is_same_v<std::string*, decltype(sp)>);
}
```

C++11でrvalueの概念の整理やstd::move()の導入が行われた目的はプログラム実行速度の向上である。

- lvalueからの代入
- rvalueからの代入
- std::move(lvalue)からの代入

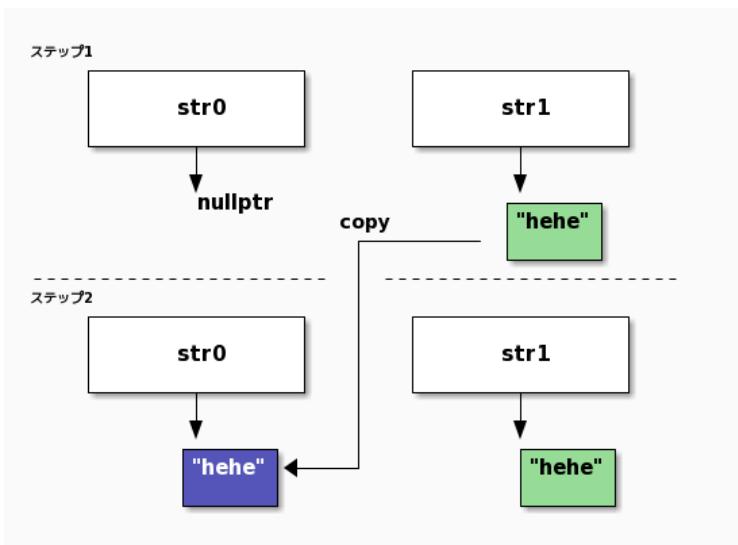
の処理がどのように違うのかを見ることでrvalueの効果について説明する。

1. 下記コードにより「lvalueからの代入」を説明する。

```
// @@@ example/term_explanation/rvalue_lvalue_ut.cpp 35

auto str0 = std::string{};           // str0はlvalue
auto str1 = std::string{"hehe"};    // str1もlvalue
str0     = str1;                   // lvalueからの代入
```

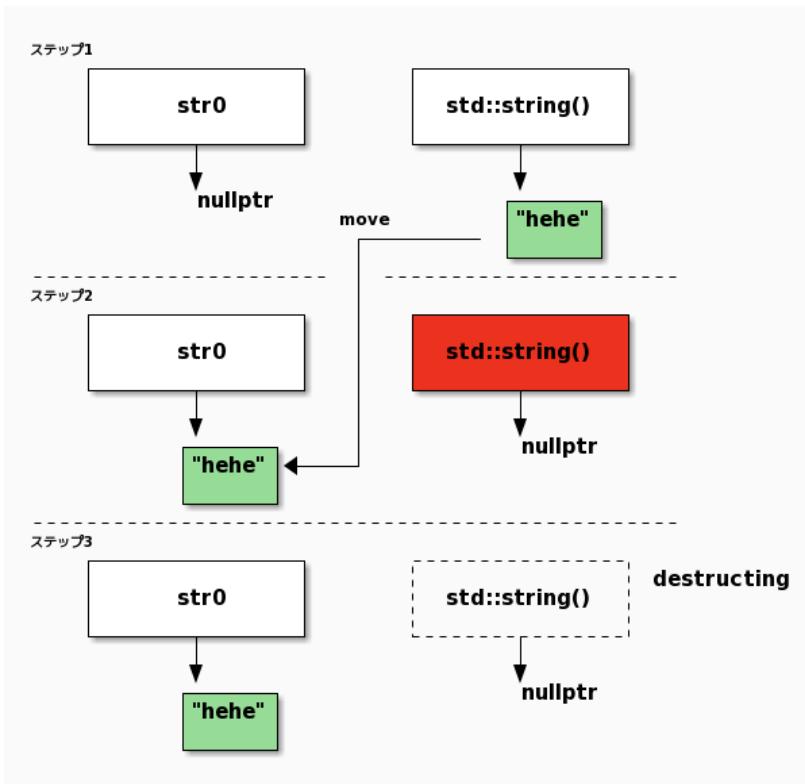
- ステップ1。str0、str1がそれぞれ初期化される("hehe"を保持するバッファが生成され、それをstr1オブジェクトが所有する)。
- ステップ2。str1が所有している文字列バッファと等価のバッファが作られ(文字列バッファ用のメモリをnewし、文字列を代入)、str0がそれを所有する。従って、"hehe"を保持するバッファが2つできる。この代入をcopy代入と呼ぶ。



2. 下記コードにより「rvalueからの代入」を説明する。

```
// @@@ example/term_explanation/rvalue_lvalue_ut.cpp 46
auto str0 = std::string{};           // str0はlvalue
str0    = std::string{"hehe"}; // rvalueからの代入
```

- ステップ1。`str0`、`std::string()`により作られたテンポラリオブジェクトがそれぞれ初期化される（`"hehe"`を保持するバッファが生成され、それをテンポラリオブジェクトが所有する）。
- ステップ2。“`hehe`”を保持する文字列バッファをもう1つ作る代わりに、テンポラリオブジェクトが所有している文字列バッファを`str0`の所有にする。この代入をmove代入と呼ぶ。
- ステップ3。テンポラリオブジェクトが解体されるが、文字列バッファは`str0`の所有であるため`delete`する必要がなく、実際には何もしない。move代入によって、文字列バッファの生成と破棄の回数がそれぞれ1回少なくなったため、実行速度は向上する（通常、`new/delete`の処理コストは高い）。



3. 下記コードにより「`std::move(lvalue)`からの代入」を説明する。

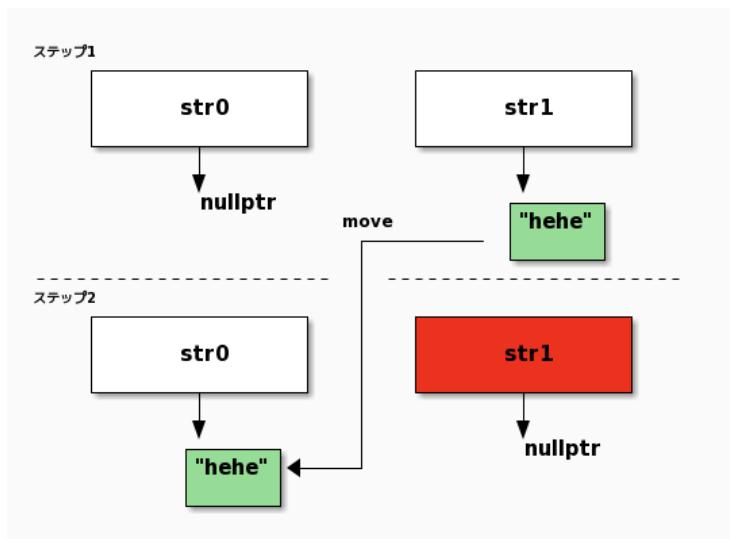
```
// @@@ example/term_explanation/rvalue_lvalue_ut.cpp 56
```

```

auto str0 = std::string{};           // str0はlvalue
auto str1 = std::string{"hehe"};    // str1もlvalue
str0     = std::move(str1);        // str1はこれ以降使われないとする

```

- ステップ1。「lvalueからの代入」のステップ1と同じである。
- ステップ2。`std::move()`の効果により(実際にはrvalueリファレンスへのキャストが行われるだけなので、実行時速度に影響はない)、“hehe”を保持する文字列バッファをもう1つ作る代わりに、`str1`が所有している文字列バッファを`str0`の所有にする。この代入もmove代入と呼ぶ。この動作は「rvalueからの代入」と同じであり、同様に速度が向上するが、その副作用として、`str1.size() == 0`となる。



エッセンシャルタイプがTであるlvalue、xvalue、prvalueに対して(例えば、`std::string const&`のエッセンシャルタイプは`std::string`である)、`decltype`の算出結果は下表のようになる。

decltype	算出された型
<code>decltype(lvalue)</code>	T
<code>decltype((lvalue))</code>	T&
<code>decltype(xvalue)</code>	T&&
<code>decltype((xvalue))</code>	T&&
<code>decltype(prvalue)</code>	T
<code>decltype((prvalue))</code>	T

この表の結果を使用した下記の関数型マクロ群により式を分類できる。定義から明らかな通り、これらは[テンプレートメタプログラミング](#)に有効に活用できる。

```

// @@@ example/term_explanation/rvalue_lvalue_ut.cpp 65

#define IS_LVALUE(EXPR_) std::is_lvalue_reference_v<decltype((EXPR_))>
#define IS_XVALUE(EXPR_) std::is_rvalue_reference_v<decltype((EXPR_))>
#define IS_PRVALUE(EXPR_) !std::is_reference_v<decltype((EXPR_))>
#define IS_RVALUE(EXPR_) (IS_PRVALUE(EXPR_) || IS_XVALUE(EXPR_))

TEST(Expression, rvalue)
{
    auto str = std::string{};

    static_assert(IS_LVALUE(str), "EXPR_ must be lvalue");
    static_assert(!IS_RVALUE(str), "EXPR_ must NOT be rvalue");

    static_assert(IS_XVALUE(std::move(str)), "EXPR_ must be xvalue");
    static_assert(!IS_PRVALUE(std::move(str)), "EXPR_ must NOT be prvalue");

    static_assert(IS_PRVALUE(std::string{}), "EXPR_ must be prvalue");
    static_assert(IS_RVALUE(std::string{}), "EXPR_ must be rvalue");
    static_assert(!IS_LVALUE(std::string{}), "EXPR_ must NOT be lvalue");
}

```

lvalue

「expression」を参照せよ。

rvalue

「expression」を参照せよ。

xvalue

「expression」を参照せよ。

prvalue

「expression」を参照せよ。

rvalue修飾

下記GetString0()のような関数が返すオブジェクトの内部メンバに対するハンドルは、 オブジェクトのライフタイム終了後にもアクセスすることができるため、 そのハンドルを通じて、 ライフタイム終了後のオブジェクトのメンバオブジェクトにもアクセスできてしまう。

ライフタイム終了後のオブジェクトにアクセスすることは未定義動作であり、 特にそのオブジェクトがrvalueであった場合、 さらにその危険性は高まる。

こういったコードに対処するためのシンタックスが、 lvalue修飾、 rvalue修飾である。

下記GetString1()、 GetString3()、 GetString4()のようにメンバ関数をlvalue修飾やrvalue修飾することで、 rvalueの内部ハンドルを返さないようになることが可能となり、 上記の危険性を緩和することができる。

```
// @@@ example/term_explanation/rvalue_lvalue_ut.cpp 91

class C {
public:
    explicit C(char const* str) : str_{str} {}

    // lvalue修飾なし、 rvalue修飾なし
    std::string& GetString0() noexcept { return str_; }

    // lvalue修飾
    std::string const& GetString1() const& noexcept { return str_; }

    // rvalue修飾
    // *thisがrvalueの場合でのGetString1()の呼び出しは、 この関数を呼び出すため、
    // class内部のハンドルを返してはならない。
    // また、 それによりstd::stringを生成するため、 noexcept指定してはならない。
    std::string GetString1() const&& { return str_; }

    // lvalue修飾だが、 const関数はrvalueからでも呼び出せる。
    // rvalueに対しての呼び出しを禁止したい場合には、 GetString4のようにする。
    std::string const& GetString2() const& noexcept { return str_; }

    // lvalue修飾
    // 非constなのでrvalueからは呼び出せない。
    std::string const& GetString3() & noexcept { return str_; }

    // lvalue修飾
    std::string const& GetString4() const& noexcept { return str_; }

    // rvalue修飾
    // rvalueからこの関数を呼び出されるとrvalueオブジェクトの内部ハンドルを返してしまい、
    // 危険なので=deleteすべき。
    std::string const& GetString4() const&& = delete;

private:
    std::string str_;
};

// @@@ example/term_explanation/rvalue_lvalue_ut.cpp 132

auto      c      = C{"c0"};
auto const& s0_0 = c.GetString0();           // OK cが解放されるまでs0_0は有効
auto      s0_1 = C{"c1"}.GetString0(); // NG 危険なコード
// s0_1が指すオブジェクトは、 次の行で無効になる

auto const& s1_0 = c.GetString1();           // OK GetString1()&が呼び出される
auto const& s1_1 = C{"c1"}.GetString1(); // OK GetString1()&&が呼び出される
// s1_0が指すrvalueはs1_0がスコープアウトするまで有効

auto const& s2_0 = c.GetString2();           // OK GetString2()&が呼び出される
```

```

auto const& s2_1 = C{"c1"}.GetString2(); // NG const関数はlvalue修飾しても呼び出し可能
// s2_1が指すオブジェクトは、次の行で無効になる

auto const& s3_0 = c.GetString3(); // OK GetString3()&が呼び出される
// auto const& s3_1 = C{"c1"}.GetString3(); // 危険なのでコンパイルさせない

auto const& s4_0 = c.GetString4(); // OK GetString4()&が呼び出される
// auto const& s4_1 = C{"c1"}.GetString4(); // 危険なのでコンパイルさせない

```

lvalue修飾

rvalue修飾を参照せよ。

リファレンス修飾

rvalue修飾とlvalue修飾とを併せて、リファレンス修飾と呼ぶ。

decltype

decltypeはオペランドにexpressionを取り、その型を算出する機能である。 decltype(auto)はそのオペランドの省略形である。 autoとdecltypeでは、以下に示す通りリファレンスの扱いが異なることに注意する必要がある。

```

// @@@ example/term_explanation/decltype_ut.cpp 7

int32_t x{3};
int32_t& r{x};

auto a = r; // aの型はint32_t
decltype(r) b = r; // bの型はint32_t&
decltype(auto) c = r; // cの型はint32_t& C++14からサポート
                     // decltype(auto)は、decltypeに右辺の式を与えるための構文

// std::is_sameはオペランドの型が同じか否かを返すメタ関数
static_assert(std::is_same_v<decltype(a), int>);
static_assert(std::is_same_v<decltype(b), int&>);
static_assert(std::is_same_v<decltype(c), int&>);

```

decltypeは、テンプレートプログラミングに多用されるが、クロージャ型(「ラムダ式」参照)のような記述不可能な型をオブジェクトから算出できるため、下記例のような場合にも有用である。

```

// @@@ example/term_explanation/decltype_ut.cpp 26

// 本来ならばA::dataは、
//      * A::Aでメモリ割り当てる
//      * A::~Aでメモリ解放
// すべきだが、何らかの理由でそれが出来ないとする
struct A {
    size_t len;
    uint8_t* data;
};

void do_something(size_t len)
{
    auto deallocate = [] (A* p) {
        delete[] (p->data);
        delete p;
    };

    auto a_ptr = std::unique_ptr<A, decltype(deallocate)>{new A, deallocate};

    a_ptr->len = len;
    a_ptr->data = new uint8_t[10];

    ...
    // do something for a_ptr
    ...

    // a_ptrによるメモリの自動解放
}

```

リファレンス

ここでは、C++11から導入された

- ユニバーサルリファレンス
- リファレンスcollapsing

について解説する。

ユニバーサルリファレンス

関数テンプレートの型パラメータや型推論autoに&&をつけて宣言された変数を、ユニバーサルリファレンスと呼ぶ(C++17から「forwardingリファレンス」という正式名称が与えられた)。ユニバーサルリファレンスは一見rvalueリファレンスのように見えるが、下記に示す通り、lvalueにもrvalueにもバインドできる。

```
// @@@ example/term_explanation/universal_ref_ut.cpp 8

template <typename T>
void f(T&& t) noexcept // tはユニバーサルリファレンス
{
    ...
}

template <typename T>
void g(std::vector<T>&& t) noexcept // tはrvalueリファレンス
{
    ...
}

// @@@ example/term_explanation/universal_ref_ut.cpp 29

auto      vec = std::vector<std::string>{"lvalue"}; // vecはlvalue
auto const cvec = std::vector<std::string>{"clvalue"}; // cvecはconstなlvalue

f(vec);           // 引数はlvalue
f(cvec);          // 引数はconstなlvalue
f(std::vector<std::string>{"rvalue"}); // 引数はrvalue

// g(vec); // 引数がlvalueなのでコンパイルエラー
// g(cvec); // 引数がconst lvalueなのでコンパイルエラー
g(std::vector<std::string>{"rvalue"}); // 引数はrvalue
```

下記のコードはジェネリックラムダ(「ラムダ式」参照)の引数をユニバーサルリファレンスにした例である。

```
// @@@ example/term_explanation/universal_ref_ut.cpp 47

// sはユニバーサルリファレンス
auto value_type = [] (auto&& s) noexcept {
    if (std::is_same_v<std::string&, decltype(s)>) {
        return 0;
    }
    if (std::is_same_v<std::string const&, decltype(s)>) {
        return 1;
    }
    if (std::is_same_v<std::string&&, decltype(s)>) {
        return 2;
    }
    return 3;
};

auto      str = std::string{"lvalue"};
auto const cstr = std::string{"const lvalue"};

ASSERT_EQ(0, value_type(str));
ASSERT_EQ(1, value_type(cstr));
ASSERT_EQ(2, value_type(std::string{"rvalue"}));
```

通常、ユニバーサルリファレンスはstd::forwardと組み合わせて使用される。

forwardingリファレンス

「ユニバーサルリファレンス」を参照せよ。

perfect forwarding

perfect forwardingとは、引数のrvalue性やlvalue性を損失することなく、その引数を別の関数に転送する技術のことを指す。通常は、ユニバーサルリファレンスである関数の仮引数をstd::forwardを用いて、他の関数に渡すことで実現される。

リファレンスcollapsing

Tを任意の型とし、TRを下記のように宣言した場合、

```
using TR = T&;
```

下記のようなコードは、C++03ではコンパイルエラーとなったが、C++11からはエラーとならず、TRRはT&となる。

```
using TRR = TR&;
```

2つの&を1つに折り畳む、このような機能をリファレンスcollapsingと呼ぶ。

下記はTをintとした場合のリファレンスcollapsingの動きを示している。

```
// @@@ example/term_explanation/ref_collapsing_ut.cpp 7

int i;

using IR = int&;
using IRR = IR&; // IRRはint& &となり、int&&に変換される

IR ir = i;
IRR irr = ir;

static_assert(std::is_same_v<int&, decltype(ir)>); // lvalueリファレンス
static_assert(std::is_same_v<int&, decltype(irr)>); // lvalue!リファレンス
```

リファレンスcollapsingは、型エイリアス、型であるテンプレートパラメータ、decltypeに対して行われる。詳細な変換則は、下記のようになる。

```
T& & -> T&
T& && -> T&
T&& & -> T&
T&& && -> T&&
```

下記のようなクラステンプレートを定義した場合、

```
// @@@ example/term_explanation/ref_collapsing_ut.cpp 26

template <typename T>
struct Ref {
    T& t;
    T&& u;
};
```

下記のコードにより、テンプレートパラメータに対するこの変換則を確かめることができる。

```
// @@@ example/term_explanation/ref_collapsing_ut.cpp 38

static_assert(std::is_same_v<int&, decltype(Ref<int>::t)>); // lvalueリファレンス
static_assert(std::is_same_v<int&&, decltype(Ref<int>::u)>); // rvalueリファレンス

static_assert(std::is_same_v<int&, decltype(Ref<int&>::t)>); // lvalueリファレンス
static_assert(std::is_same_v<int&, decltype(Ref<int&>::u)>); // lvalueリファレンス

static_assert(std::is_same_v<int&, decltype(Ref<int&&>::t)>); // lvalueリファレンス
static_assert(std::is_same_v<int&&, decltype(Ref<int&&>::u)>); // rvalueリファレンス
```

この機能がないC++03では、

```
// @@@ example/term_explanation/ref_collapsing_ut.cpp 52

template <typename T>
struct AddRef {
    using type = T&;
};
```

ようなクラステンプレートに下記コードのようにリファレンス型を渡すとコンパイルエラーとなる。

```
// @@@ example/term_explanation/ref_collapsing_ut.cpp 69

static_assert(std::is_same_v<int&, AddRef<int>::type>);
```

この問題を回避するためには下記のようなテンプレートの特殊化が必要になる。

```
// @@@ example/term_explanation/ref_collapsing_ut.cpp 59

template <typename T>
```

```
struct AddRef<T&> {
    using type = T&;
};
```

上記したようなクラステンプレートでのメンバエイリアスの宣言は、[テンプレートメタプログラミング](#)で頻繁に使用されるため、このようなテンプレートの特殊化を不要にするリファレンスcollapsingは、有用な機能拡張であると言える。

danglingリファレンス

Dangling リファレンスとは、破棄後のオブジェクトを指しているリファレンスを指す。このようなリファレンスにアクセスすると、未定義動作動作に繋がるに繋がる。

```
// @@@ example/term_explanation/dangling_ut.cpp 9

bool X_destructed;
class X {
public:
    X() { X_destructed = false; }
    ~X() { X_destructed = true; }
};

bool A_destructed;
class A {
public:
    A() { A_destructed = false; }
    ~A() { A_destructed = true; }

    X const& GetX() const noexcept { return x_; }

private:
    X x_;
};

// @@@ example/term_explanation/dangling_ut.cpp 34

auto a = A{};

auto const& x_safe = a.GetX(); // x_safeはダングリングリファレンスではない
ASSERT_FALSE(A_destructed || X_destructed);

auto const& x_dangling = A{}.GetX(); // 次の行でxが指すオブジェクトは解放される
// この行ではxはdangling リファレンスになる。
ASSERT_TRUE(A_destructed && X_destructed);

auto const* x_ptr_dangling = &A{}.GetX(); // 次の行でxが指すオブジェクトは解放される
// この行ではxはdangling ポインタになる。
ASSERT_TRUE(A_destructed && X_destructed);
```

danglingポインタ

danglingポインタとは、[danglingリファレンス](#)と同じような状態になったポインタを指す。

エクセプション安全性の保証

関数のエクセプション発生時の安全性の保証には以下の3つのレベルが規定されている。

no-fail保証

「no-fail保証」を満たす関数はエクセプションをthrowしない。

強い保証

「強い保証」を満たす関数は、この関数がエクセプションによりスコープから外れた場合でも、この関数が呼ばれなかった状態と同じ(プログラムカウンタ以外の状態は同じ)であることを保証する。従って、この関数呼び出しは成功したか、完全な無効だったかのどちらかになる。

基本保証

「基本保障」を満たす関数は、この関数がエクセプションによりスコープから外れた場合でも、メモリ等のリソースリークは起こさず、オブジェクトは(変更されたかもしれないが)引き続き使えることを保証する。

シンタックス、セマンティクス

直訳すれば、シンタックスとは構文論のことであり、セマンティクスとは意味論のことである。この二つの概念の違いをはつきりと際立たせる有名な文を例示する。

```
Colorless green ideas sleep furiously(直訳:無色の緑の考えが猛烈に眠る)
```

この文は構文的には正しい(シンタックスは問題ない)が、意味不明である(セマンティクスは誤り)。

C++プログラミングにおいては、コンパイルできることがシンタックス的な正しさであり、例えば

- クラスや関数がその名前から想起できる責務を持っている
 - 「単一責任の原則(SRP)」を満たしている
 - Accessorを実装する関数の名前は、GetXxxやSetXxxになっている
 - コンテナクラスのメンバ関数beginやendは、そのクラスが保持する値集合の先頭や最後尾の次を指すイテレータを返す等
- 「等価性のセマンティクス」を守ってる
- 「copyセマンティクス」を守ってる
- 「moveセマンティクス」を守っている

等がセマンティクス的な正しさである。

セマンティクス的に正しいソースコードは読みやすく、保守性、拡張性に優れている。

等価性のセマンティクス

純粋数学での実数の等号(=)は、任意の実数x、y、zに対して、

律	意味
反射律	$x = x$
対称律	$x = y$ ならば $y = x$
推移律	$x = y$ 且 $y = z$ ならば $x = z$

を満たしている。 $x = y$ が成立する場合、「 x は y と等しい」もしくは「 x は y と同一」であると言う。

C++における組み込みの==も純粋数学の等号と同じ性質を満たしている。下記のコードは、その性質を表している。

```
// @@@ example/term_explanation/semantics_ut.cpp 12

auto a = 0;
auto& b = a;

ASSERT_TRUE(a == b);
ASSERT_TRUE(&a == &b); // aとbは同一
```

しかし、下記のコード内のa、bは同じ値を持つが、アドレスが異なるため同一のオブジェクトではないにもかかわらず、組み込みの==の値はtrueとなる。

```
// @@@ example/term_explanation/semantics_ut.cpp 22

auto a = 0;
auto b = 0;

ASSERT_TRUE(a == b);
ASSERT_FALSE(&a == &b); // aとbは同一ではない
```

このような場合、aとbは等価であるという。同一ならば等価であるが、等価であっても同一とは限らない。

ポインタや配列をオペランドとする場合を除き、C++における組み込みの==は、数学の等号とは違い、等価を表していると考えられるが、上記した3つの律を守っている。従ってオーバーロードoperator==も同じ性質を守る必要がある。

組み込みの==やオーバーロードoperator==のこののような性質をここでは「等価性のセマンティクス」と呼ぶ。

クラスAを下記のように定義し、

```
// @@@ example/term_explanation/semantics_ut.cpp 33

class A {
public:
    explicit A(int num, char const* name) noexcept : num_{num}, name_{name} {}

    int      GetNum() const noexcept { return num_; }
    char const* GetName() const noexcept { return name_; }

private:
    int const  num_;
    char const* name_;
};
```

そのoperator==を下記のように定義した場合、

```
// @@@ example/term_explanation/semantics_ut.cpp 50

inline bool operator==(A const& lhs, A const& rhs) noexcept
{
    return lhs.GetNum() == rhs.GetNum() && lhs.GetName() == rhs.GetName();
}
```

単体テストは下記のように書けるだろう。

```
// @@@ example/term_explanation/semantics_ut.cpp 61

auto a0 = A{0, "a"};
auto a1 = A{0, "a"};

ASSERT_TRUE(a0 == a1);
```

これは、一応パスするが(処理系定義の動作を前提とするため、必ず動作する保証はない)、下記のようにすると、パスしなくなる。

```
// @@@ example/term_explanation/semantics_ut.cpp 71

char a0_name[] = "a";
auto a0        = A{0, a0_name};

char a1_name[] = "a";
auto a1        = A{0, a1_name};

ASSERT_TRUE(a0 == a1); // テストが失敗する
```

一般にポインタの等価性は、その値の同一性ではなく、そのポインタが指すオブジェクトの等価性で判断されるべきであるが、先に示したoperator==はその考慮をしていないため、このような結果になった。

次に、これを修正した例を示す。

```
// @@@ example/term_explanation/semantics_ut.cpp 90

inline bool operator==(A const& lhs, A const& rhs) noexcept
{
    return lhs.GetNum() == rhs.GetNum()
        && std::string_view{lhs.GetName()} == std::string_view{rhs.GetName()};
}
```

ポインタをメンバに持つクラスのoperator==については、上記したような処理が必要となる。

次に示す例は、基底クラスBaseとそのoperator==である。

```
// @@@ example/term_explanation/semantics_ut.cpp 113

class Base {
public:
    explicit Base(int b) noexcept : b_{b} {}
    virtual ~Base() = default;
    int GetB() const noexcept { return b_; }

private:
    int b_;
};

inline bool operator==(Base const& lhs, Base const& rhs) noexcept
{
    return lhs.GetB() == rhs.GetB();
}
```

次の単体テストが示す通り、これ自体には問題がないように見える。

```
// @@@ example/term_explanation/semantics_ut.cpp 133

auto b0 = Base{0};
auto b1 = Base{0};
auto b2 = Base{1};

ASSERT_TRUE(b0 == b0);
ASSERT_TRUE(b0 == b1);
ASSERT_FALSE(b0 == b2);
```

しかし、Baseから派生したクラスDerivedを

```
// @@@ example/term_explanation/semantics_ut.cpp 145

class Derived : public Base {
public:
    explicit Derived(int d) noexcept : Base{0}, d_{d} {} 
    int GetD() const noexcept { return d_; }

private:
    int d_;
};
```

のように定義すると、下記の単体テストで示す通り、等価性のセマンティクスが破壊される。

```
// @@@ example/term_explanation/semantics_ut.cpp 159

{
    auto b = Base{0};
    auto d = Derived{1};

    ASSERT_TRUE(b == d); // NG bとdは明らかに等価でない
}
{
    auto d0 = Derived{0};
    auto d1 = Derived{1};

    ASSERT_TRUE(d0 == d1); // NG d0とd1は明らかに等価ではない
}
```

Derived用のoperator==を

```
// @@@ example/term_explanation/semantics_ut.cpp 176

bool operator==(Derived const& lhs, Derived const& rhs) noexcept
{
    return lhs.GetB() == rhs.GetB() && lhs.GetD() == rhs.GetD();
```

と定義しても、下記に示す通り部分的な効果しかない。

```
// @@@ example/term_explanation/semantics_ut.cpp 186

auto d0 = Derived{0};
auto d1 = Derived{1};

ASSERT_FALSE(d0 == d1); // OK operator==(Derived const&, Derived const&)の効果で正しい判定

Base& d0_b_ref = d0;

ASSERT_TRUE(d0_b_ref == d1); // NG d0_b_refの実態はd0なのでd1と等価でない
```

この問題は、RTTIを使った下記のようなコードで対処できる。

```
// @@@ example/term_explanation/semantics_ut.cpp 202

class Base {
public:
    explicit Base(int b) noexcept : b_{b} {}
    virtual ~Base() = default;
    int GetB() const noexcept { return b_; }

protected:
    virtual bool is_equal(Base const& rhs) const noexcept { return b_ == rhs.b_; }

private:
    int b_;
```

```

friend inline bool operator==(Base const& lhs, Base const& rhs) noexcept
{
    if (typeid(lhs) != typeid(rhs)) {
        return false;
    }

    return lhs.is_equal(rhs);
};

class Derived : public Base {
public:
    explicit Derived(int d) : Base{0}, d_{d} {}

    int GetD() const noexcept { return d_; }

protected:
    virtual bool is_equal(Base const& rhs) const noexcept
    {
        // operator==によりrhsの型はDerivedであるため、下記のキャストは安全
        auto const& rhs_d = static_cast<Derived const&>(rhs);

        return Base::is_equal(rhs) && d_ == rhs_d.d_;
    }

private:
    int d_;
};

```

下記に示す通り、このコードは、オープン・クローズドの原則(OCP)にも対応した柔軟な構造を実現している。

```

// @@@ example/term_explanation/semantics_ut.cpp 269

class DerivedDerived : public Derived {
public:
    explicit DerivedDerived(int dd) noexcept : Derived{0}, dd_{dd} {}

    int GetDD() const noexcept { return dd_; }

protected:
    virtual bool is_equal(Base const& rhs) const noexcept
    {
        // operator==によりrhsの型はDerivedDerivedであるため、下記のキャストは安全
        auto const& rhs_dd = static_cast<DerivedDerived const&>(rhs);

        return Derived::is_equal(rhs) && dd_ == rhs_dd.dd_;
    }

private:
    int dd_;
};

```

前例では「両辺の型が等しいこと」が「等価であること」の必要条件となるが、この要件が、すべてのoperator==に求められるわけではない。

次に示すのは、一見すると両辺の型が違うにもかかわらず、等価性のセマンティクスを満たしている例である。

```

// @@@ example/term_explanation/semantics_ut.cpp 319

auto abc = std::string("abc");

ASSERT_TRUE("abc" == abc);
ASSERT_TRUE(abc == "abc");

```

これは、文字列リテラルを第1引数に取るstd::stringのコンストラクタが非explicitであることによって、文字列リテラルからstd::stringへの暗黙の型変換が起こるために成立する（「非explicitなコンストラクタによる暗黙の型変換」参照）。

以上で見てきたように、等価性のセマンティクスを守ったoperator==の実装には多くの観点が必要になる。

copyセマンティクス

copyセマンティクスとは以下を満たすようなセマンティクスである。

- $a = b$ が行われた後に、 a と b が等価である。
- $a = b$ が行われた前後で b の値が変わっていない。

従って、これらのオブジェクトに対して等価性のセマンティクスを満たすoperator==が定義されている場合、以下を満たすようなセマンティクスであると言い換えることができる。

- $a = b$ が行われた後に、 $a == b$ がtrueになる。
- $b == b_pre$ がtrueの時に、 $a = b$ が行われた後でも $b == b_pre$ がtrueとなる。

下記に示す通り、std::stringはcopyセマンティクスを満たしている。

```
// @@@ example/term_explanation/semantics_ut.cpp 333

auto c_str = "string";
auto str   = std::string{};

str = c_str;
ASSERT_TRUE(c_str == str);      // = 後には == が成立している
ASSERT_STREQ("string", c_str); // c_strの値は変わっていない
```

一方で、std::auto_ptrはcopyセマンティクスを満たしていない。

```
// @@@ example/term_explanation/semantics_ut.cpp 346

std::auto_ptr<std::string> str0{new std::string("string")};
std::auto_ptr<std::string> str0_pre{new std::string("string")};

ASSERT_TRUE(*str0 == *str0_pre); // 前提は成立

std::auto_ptr<std::string> str1;

str1 = str0;

// ASSERT_TRUE(*str0 == *str0_pre); // これをするとクラッシュする
ASSERT_TRUE(str0.get() == nullptr); // str0の値がoperator ==で変わってしまった

ASSERT_TRUE(*str1 == *str0_pre); // これは成立
```

この仕様は極めて不自然であり、std::auto_ptrはC++11で非推奨となり、C++17で規格から排除された。

下記の単体テストから明らかな通り、「等価性のセマンティクス」で示した最後の例も、copyセマンティクスを満たしていない。

```
// @@@ example/term_explanation/semantics_ut.cpp 366

auto b = Base{1};
auto d = Derived{1};

b = d; // スライシングが起こる

ASSERT_FALSE(b == d); // copyセマンティクスを満たしていない
```

原因是、copy代入でスライシングが起こるためである。

moveセマンティクス

moveセマンティクスとは以下を満たすようなセマンティクスである(operator==が定義されていると前提)。

- copy代入の実行コスト \geq move代入の実行コスト
- $a == b$ がtrueの時に、 $c = \text{std::move}(a)$ が行われた場合、
 - $b == c$ がtrueになる。
 - $a == c$ はtrueにならなくても良い(a はmove代入により破壊されるかもしれない)。

必須ではないが、「 a がポインタ等のリソースを保有している場合、move代入後には、そのリソースは c に移動している」ことが一般的である(「rvalue」参照)。

- no-fail保証をする(noexceptと宣言し、エクセプションをthrowしない)。

moveセマンティクスはcopy代入後に使用されなくなるオブジェクト(主にrvalue)からのcopy代入の実行コストを下げるために導入されたため、下記のようなコードは推奨されない。

```
// @@@ example/term_explanation/semantics_ut.cpp 381

class NotRecommended {
public:
    NotRecommended(char const* name) : name_{name} {}
    std::string const& Name() const noexcept { return name_; }

    NotRecommended& operator=(NotRecommended&& rhs) // move代入、非no-fail保証
    {
```

```

        name_ = rhs.name_; // rhs.name_からname_へのcopy代入。パフォーマンス問題になるかも。
    }

private:
    std::string name_;
};

bool operator==(NotRecommended const& lhs, NotRecommended const& rhs) noexcept
{
    return lhs.Name() == rhs.Name();
}

TEST(Semantics, move1)
{
    auto a = NotRecommended{"a"};
    auto b = NotRecommended{"a"};

    ASSERT_EQ("a", a.Name());
    ASSERT_TRUE(a == b);

    auto c = NotRecommended{"c"};
    ASSERT_EQ("c", c.Name());

    c = std::move(a);
    ASSERT_TRUE(b == c); // 一応、moveセマンティクスは守っているが・・・
}

```

下記のコードのようにメンバの代入もできる限りmove代入を使うことで、パフォーマンスの良い代入ができる。

```

// @@@ example/term_explanation/semantics_ut.cpp 419

class Recommended {
public:
    Recommended(char const* name) : name_{name} {}
    std::string const& Name() const noexcept { return name_; }

    Recommended& operator=(Recommended&& rhs) noexcept // move代入、no-fail保証
    {
        name_ = std::move(rhs.name_); // rhs.name_からname_へのmove代入
        return *this;
    }

private:
    std::string name_;
};

bool operator==(Recommended const& lhs, Recommended const& rhs) noexcept
{
    return lhs.Name() == rhs.Name();
}

TEST(Semantics, move2)
{
    auto a = Recommended{"a"};
    auto b = Recommended{"a"};

    ASSERT_EQ("a", a.Name());
    ASSERT_TRUE(a == b);

    auto c = Recommended{"c"};
    ASSERT_EQ("c", c.Name());

    c = std::move(a); // これ以降aは使ってはならない
    ASSERT_TRUE(b == c); // moveセマンティクスを正しく守っている
}

```

C++コンパイラー

本ドキュメントで使用するg++/clang++のバージョンは以下のとおりである。

g++

```

g++ (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

```

clang++

```
Ubuntu clang version 14.0.0-1ubuntu1
Target: x86_64-pc-linux-gnu
Thread model: posix
InstalledDir: /usr/bin
```

C++その他

オーバーライドとオーバーロードの違い

下記例では、Base::g()がオーバーロードで、Derived::f()がオーバーライドである (Derived::g()はオーバーロードでもオーバーライドでもない (「name-hiding」参照))。

```
// @@@ example/term_explanation/override_overload_ut.cpp 5

class Base {
public:
    virtual ~Base() = default;
    virtual std::string f() { return "Base::f"; }
    std::string g() { return "Base::g"; }

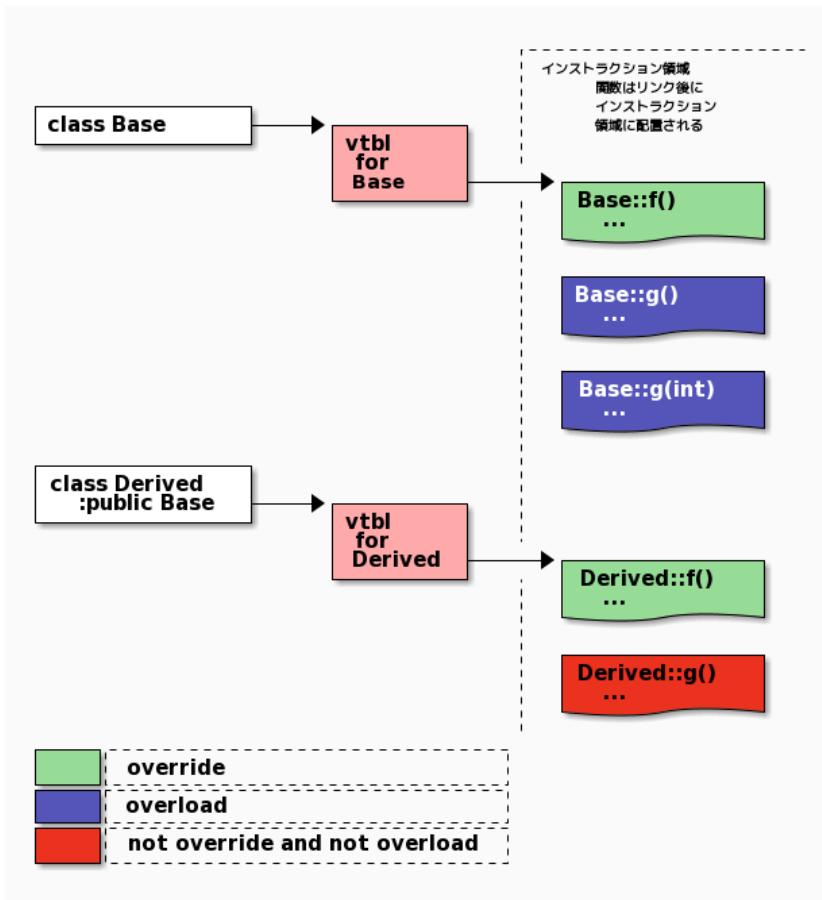
    // g()のオーバーロード
    std::string g(int) { return "Base::g(int)"; }
};

class Derived : public Base {
public:
    // Base::fのオーバーライド
    virtual std::string f() override { return "Derived::f"; }

    // Base::gのname-hiding
    std::string g() { return "Derived::g"; }
};
```

下記図の通り、

- BaseのインスタンスはBase用のvtblへのポインタを内部に持ち、そのvtblでBase::f()のアドレスを保持する。
- DerivedのインスタンスはDerived用のvtblへのポインタを内部に持ち、そのvtblでDerived::f()のアドレスを保持する。
- Base::g()、Base::g(int)、Derived::g()のアドレスはBaseやDerivedのインスタンスから辿ることはできない。



vtblとは仮想関数テーブルとも呼ばれる、仮想関数ポインタを保持するための上記のようなテーブルである（「[クラスのレイアウト](#)」参照）。

Base::f()、Derived::f()の呼び出し選択は、オブジェクトの表層の型ではなく、実際の型により決定される。Base::g()、Derived::g()の呼び出し選択は、オブジェクトの表層の型により決定される。

```
// @@@ example/term_explanation/override_overload_ut.cpp 29

auto ret = std::string{};
auto b = Base{};
auto d = Derived{};
Base& d_ref = d;

ret = b.f(); // Base::f()呼び出し
ASSERT_EQ("Base::f", ret);

ret = d.f(); // Derived::f()呼び出し
ASSERT_EQ("Derived::f", ret);

ret = b.g(); // Base::g()呼び出し
ASSERT_EQ("Base::g", ret);

ret = d.g(); // Derived::g()呼び出し
ASSERT_EQ("Derived::g", ret);
// ret = d.g(int{}); // Derived::gによって、Base::gが隠されるのでコンパイルエラー

ret = d_ref.f(); // Base::fはDerived::fによってオーバーライドされたので、Derived::f()呼び出し
ASSERT_EQ("Derived::f", ret);

ret = d_ref.g(); // d_refの表層型はBaseなので、Base::g()呼び出し
ASSERT_EQ("Base::g", ret);

ret = d_ref.g(int{}); // d_refの表層型はBaseなので、Base::g(int)呼び出し
ASSERT_EQ("Base::g(int)", ret);
```

上記のメンバ関数呼び出し

```
d_ref.f()
```

がどのように解釈され、Derived::f()が選択されるかを以下に疑似コードで例示する。

```

vtbl = d_ref.vtbl           // d_refの実態はDerivedなのでvtblはDerivedのvtbl
member_func = vtbl->f      // vtbl->fはDerived::f()のアドレス
(d_ref.*member_func)(&d_ref) // member_func()の呼び出し

```

このようなメカニズムにより仮想関数呼び出しが行われる。

実引数/仮引数

引数(もしくは実引数、 argument)、 仮引数(parameter)とは下記のように定義される。

```

// @@@ example/term_explanation/argument.cpp 2

int f0(int a, int& b) noexcept // a, bは仮引数
{
    ...
}

void f1() noexcept
{
    ...
    f0(x, y); // x, yは実引数
}

```

範囲for文

範囲for文は下記例のコメントで示されたようなfor文であり、 begin()、 end()によって表される範囲内のすべての要素に対して付属するブロックの処理を行う。

```

// @@@ example/term_explanation/range_for_ut.cpp 9

auto list = std::list{1, 2, 3};

for (auto const a : list) { // 範囲for文
    std::cout << a << std::endl;
}

// 上記for文は下記for文のシンタックスシュガード
for (std::list<int32_t>::iterator it = std::begin(list); it != std::end(list); ++it) {
    std::cout << *it << std::endl;
}

```

```

// @@@ example/term_explanation/range_for_ut.cpp 25

int32_t array[3]{1, 2, 3};

for (auto const a : array) { // 範囲for文
    std::cout << a << std::endl;
}

// 上記for文は下記for文のシンタックスシュガード
for (int32_t* it = std::begin(array); it != std::end(array); ++it) {
    std::cout << *it << std::endl;
}

```

ラムダ式

ラムダ式に関する言葉の定義と例を示す。

- ・ ラムダ式とは、 その場で関数オブジェクトを定義する式。
- ・ クロージャ(オブジェクト)とは、 ラムダ式から生成された関数オブジェクト。
- ・ クロージャ型とは、 クロージャオブジェクトの型。
- ・ キャプチャとは、 ラムダ式外部の変数をラムダ式内にコピーかりファレンスとして定義する機能。
- ・ ラムダ式からキャプチャできるのは、 ラムダ式から可視である自動変数と仮引数(thisを含む)。
- ・ ジェネリックラムダとは、 C++11のラムダ式を拡張して、 パラメータにautoを使用(型推測)できるようにした機能。

```

// @@@ example/term_explanation/lambda.cpp 10

auto a = 0;

// closureがクロージャ。それを初期化する式がラムダ式
// [a = a]がキャプチャ

```

```

// [a = a]内の右辺aは上記で定義されたa
// [a = a]内の左辺aは右辺aで初期化された変数。ラムダ式内で使用されるaは左辺a。
auto closure = [a = a](int32_t b) noexcept { return a + b; };

auto ret = closure(3); // クロージャの実行

// g_closureはジェネリックラムダ
auto g_closure = [](auto t0, auto t1) { return t0 + t1; };

auto i = g_closure(1, 2); // t0, t1はint
auto s = g_closure(std::string("1"), std::string("2")); // t0, t1はstd::string

```

ジェネリックラムダ

ジェネリックラムダとは、C++11のラムダ式のパラメータの型にautoを指定できるようにした機能で、C++14で導入された。

この機能により関数の中で関数テンプレートと同等のものが定義できるようになった。

ジェネリックラムダで定義されたクロージャは、通常のラムダと同様にオブジェクトであるため、下記のように使用することもできる便利な記法である。

```

// @@@ example/term_explanation/generic_lambda_ut.cpp 4

template <typename PUTTO>
void f(PUTTO& p)
{
    p(1);
    p(2.71);
    p("str");
}

TEST(Template, generic_lambda)
{
    std::ostringstream oss;

    f([&oss](auto const& elem) { oss << elem << std::endl; });

    ASSERT_EQ("1\n2.71\nstr\n", oss.str());
}

```

なお、上記のジェネリックラムダは下記クラスのインスタンスの動きと同じである。

```

// @@@ example/term_explanation/generic_lambda_ut.cpp 23

class Closure {
public:
    Closure(std::ostream& os) : os_(os) {}

    template <typename T>
    void operator()(T& t)
    {
        os_ << t << std::endl;
    }

private:
    std::ostream& os_;
};

TEST(Template, generic_lambda_like)
{
    std::ostringstream oss;

    Closure closure(oss);
    f(closure);

    ASSERT_EQ("1\n2.71\nstr\n", oss.str());
}

```

関数tryブロック

関数tryブロックとはtry-catchを本体とした下記のような関数のブロックを指す。

```

// @@@ example/term_explanation/func_try_block.cpp 8

void function_try_block()
try { // 関数tryブロック
    // 何らかの処理
}

```

```
    ...
}
catch (std::length_error const& e) { // 関数tryブロックのエクセプションハンドラ
    ...
}
catch (std::logic_error const& e) { // 関数tryブロックのエクセプションハンドラ
    ...
}
```

単純代入

代入は下記のように分類される。

- 単純代入(=)
- 複合代入(+=, ++ 等)

ill-formed

標準規格と処理系に詳しい解説があるが、

- well-formed(適格)とはプログラムが全ての構文規則・診断対象の意味規則・單一定義規則を満たすことである。
- ill-formed(不適格)とはプログラムが適格でないことである。

プログラムがwell-formedになった場合、そのプログラムはコンパイルできる。プログラムがill-formedになった場合、通常はコンパイルエラーになるが、対象がテンプレートの場合、事情は少々異なり、SFINAEによりコンパイルできることもある。

well-formed

「ill-formed」を参照せよ。

one-definition rule

「ODR」を参照せよ。

ODR

ODRとは、One Definition Ruleの略語であり、下記のようなことを定めている。

- どの翻訳単位でも、テンプレート、型、関数、またはオブジェクトは、複数の定義を持つことができない。
- プログラム全体で、オブジェクトまたは非インライン関数は複数の定義を持つことはできない。
- 型、テンプレート、外部インライン関数等、いくつかのものは複数の翻訳単位で定義することができる。

より詳しい内容が知りたい場合は、<https://en.cppreference.com/w/cpp/language/definition>が参考になる。

RVO(Return Value Optimization)

関数の戻り値がオブジェクトである場合、戻り値オブジェクトは、その関数の呼び出し元のオブジェクトにコピーされた後、すぐに破棄される。この「オブジェクトをコピーして、その後すぐにそのオブジェクトを破棄する」動作は、「関数の戻り値オブジェクトをそのままその関数の呼び出し元で使用する」ことで効率的になる。RVOとはこのような最適化を指す。

なお、このような最適化は、C++17から規格化された。

SSO(Small String Optimization)

一般にstd::stringで文字列を保持する場合、newしたメモリが使用される。64ビット環境であれば、newしたメモリのアドレスを保持する領域は8バイトになる。std::stringで保持する文字列が終端の'\0'も含め8バイト以下である場合、アドレスを保持する領域をその文字列の格納に使用すれば、newする必要がない(当然deleteも不要)。こうすることで、短い文字列を保持するstd::stringオブジェクトは効率的に動作できる。

SSOとはこのような最適化を指す。

Most Vexing Parse

Most Vexing Parse(最も困惑させる構文解析)とは、C++の文法に関連する問題で、Scott Meyersが彼の著書"Effective STL"の中でこの現象に名前をつけたことに由来する。

この問題はC++の文法が関数の宣言と変数の定義とを曖昧に扱うことによって生じる。特にオブジェクトの初期化の文脈で発生し、意図に反して、その行は関数宣言になってしまう。

```
// @@@ example/term_explanation/most_vexing_parse_ut.cpp 6

class Vexing {
public:
    Vexing(int) {}
};

// @@@ example/term_explanation/most_vexing_parse_ut.cpp 19

Vexing obj1();          // はローカルオブジェクトobj1の宣言ではない
Vexing obj2(Vexing);   // ローカルオブジェクトobj2の宣言ではない

ASSERT_EQ("Vexing()", Nstd::Type2Str<decltype(obj1)>());
ASSERT_EQ("Vexing (Vexing)", Nstd::Type2Str<decltype(obj2)>());
// 上記単体テストが示すように、
// * obj1はVexingを返す関数
// * obj2はVexingを引数に取りVexingを返す関数
// となる。
```

初期化子リストコンストラクタの呼び出しでオブジェクトの初期化を行うことで、このような問題を回避できる。

RTTI

RTTI(Run-time Type Information)とは、プログラム実行中のオブジェクトの型を導出するための機能であり、具体的には下記の3つの要素を指す。

- dynamic_cast
- typeid
- std::type_info

下記のようなポリモーフィックな(virtual関数を持った)クラスに対しては、

```
// @@@ example/term_explanation/rtti_ut.cpp 7

class Polymorphic_Base { // ポリモーフィックな基底クラス
public:
    virtual ~Polymorphic_Base() = default;
};

class Polymorphic_Derived : public Polymorphic_Base { // ポリモーフィックな派生クラス
};
```

dynamic_cast、typeidやその戻り値であるstd::type_infoは、下記のように振舞う。

```
// @@@ example/term_explanation/rtti_ut.cpp 21

auto b = Polymorphic_Base{};
auto d = Polymorphic_Derived{};

Polymorphic_Base& b_ref_d = d;
Polymorphic_Base& b_ref_b = b;

// std::type_infoの比較
ASSERT_EQ(typeid(b_ref_d), typeid(d));
ASSERT_EQ(typeid(b_ref_b), typeid(b));

// ポインタへのdynamic_cast
auto* d_ptr = dynamic_cast<Polymorphic_Derived*>(&b_ref_d);
ASSERT_EQ(d_ptr, &d);

auto* d_ptr2 = dynamic_cast<Polymorphic_Derived*>(&b_ref_b);
ASSERT_EQ(d_ptr2, nullptr); // キャストできない場合、nullptrが返る

// リファレンスへのdynamic_cast
auto& d_ref = dynamic_cast<Polymorphic_Derived&>(b_ref_d);
ASSERT_EQ(&d_ref, &d);

// キャストできない場合、エクセプションのが発生する
ASSERT_THROW(dynamic_cast<Polymorphic_Derived&>(b_ref_b), std::bad_cast);
```

下記のような非ポリモーフィックな(virtual関数を持たない)クラスに対しては、

```
// @@@ example/term_explanation/rtti_ut.cpp 53

class NonPolymorphic_Base { // 非ポリモーフィックな基底クラス
```

```

};

class NonPolymorphic_Derived : public NonPolymorphic_Base { // 非ポリモーフィックな派生クラス
};

```

`dynamic_cast`、`typeid`やその戻り値である`std::type_info`は、下記のように振舞う。

```

// @@@ example/term_explanation/rtti_ut.cpp 65

auto b = NonPolymorphic_Base{};
auto d = NonPolymorphic_Derived{};

NonPolymorphic_Base& b_ref_d = d;
NonPolymorphic_Base& b_ref_b = b;

// std::type_infoの比較
ASSERT_EQ(typeid(b_ref_d), typeid(b)); // 実際の型ではなく、表層型のtype_infoが返る
ASSERT_EQ(typeid(b_ref_b), typeid(b));

// virtual関数を持たないため、ポインタへのdynamic_castはコンパイルできない
// auto* d_ptr = dynamic_cast<NonPolymorphic_Derived*>(&b_ref_d);
// auto* d_ptr2 = dynamic_cast<NonPolymorphic_Derived*>(&b_ref_b);

// virtual関数を持たないため、リファレンスへのdynamic_castはコンパイルできない
// auto& d_ref = dynamic_cast<NonPolymorphic_Derived&>(b_ref_d);
// ASSERT_THROW(dynamic_cast<NonPolymorphic_Derived&>(b_ref_b), std::bad_cast);

```

Run-time Type Information

「[RTTI](#)」を参照せよ。

simple-declaration

このための記述が [simple-declaration](#) とは、C++17から導入された「従来for文しか使用できなかった初期化をif文とswitch文でも使えるようにする」ための記述方法である。

```

// @@@ example/term_explanation/simple_declaration_ut.cpp 9
int32_t f();
int32_t g1()
{
    if (auto ret = f(); ret != 0) { // retがsimple-declaration
        return ret;
    }
    else {
        return 0;
    }
}

int32_t g2()
{
    switch (auto ret = f()) { // retがsimple-declaration
    case 0:
        return 0;
    case 1:
        return ret * 5;
    case 2:
        return ret + 3;
    default:
        return -1;
    }
}

```

typeid

「[RTTI](#)」を参照せよ。

トライグラフ

トライグラフとは、2つの疑問符とその後に続く1文字によって表される、下記の文字列である。

```
?=? ??/ ??' ??( ??) ??! ??< ??> ??-
```

フリースタンディング環境

フリースタンディング環境 とは、組み込みソフトウェアやOSのように、その実行にOSの補助を受けられないソフトウェアを指す。

ソフトウェア一般

凝集度

凝集度 とはクラス設計の妥当性を表す尺度の一種であり、 Lack of Cohesion in Methods というメトリクスで計測される。

- Lack of Cohesion in Methodsの値が1に近ければ凝集度は低く、この値が0に近ければ凝集度は高い。
- 凝集度とは結合度とも呼ばれ、メンバ(メンバ変数、メンバ関数等)間の結びつきを表す。メンバ間の結びつきが強いほど、良い設計とされる。
- メンバ変数やメンバ関数が多くなれば、凝集度は低くなりやすい。
- 「単一責任の原則(SRP)」を守ると凝集度は高くなりやすい。
- 「Accessor」を多用すれば凝集度は低くなる。従って、下記のようなクラスは凝集度が低い。言い換えれば、凝集度を下げることなく、より小さいクラスに分割できる。

```
// @@@ example/term_explanation/lack_of_cohesion_ut.cpp 7

class ABC {
public:
    explicit ABC(int32_t a, int32_t b, int32_t c) noexcept : a_{a}, b_{b}, c_{c} {}

    int32_t GetA() const noexcept { return a_; }
    int32_t GetB() const noexcept { return b_; }
    int32_t GetC() const noexcept { return c_; }
    void SetA(int32_t a) noexcept { a_ = a; }
    void SetB(int32_t b) noexcept { b_ = b; }
    void SetC(int32_t c) noexcept { c_ = c; }

private:
    int32_t a_;
    int32_t b_;
    int32_t c_;
};
```

良く設計されたクラスは、下記のようにメンバが結合しあっているため凝集度が高い(ただし、「Immutable」の観点からは、QuadraticEquation::Set()がない方が良い)。言い換えれば、凝集度を落とさずにクラスを分割することは難しい。

```
// @@@ example/term_explanation/lack_of_cohesion_ut.cpp 26

class QuadraticEquation { // 2次方程式
public:
    explicit QuadraticEquation(int32_t a, int32_t b, int32_t c) noexcept : a_{a}, b_{b}, c_{c} {}

    void Set(int32_t a, int32_t b, int32_t c) noexcept
    {
        a_ = a;
        b_ = b;
        c_ = c;
    }

    int32_t Discriminant() const noexcept // 判定式
    {
        return b_ * b_ - 4 * a_ * c_;
    }

    bool HasRealNumberSolution() const noexcept { return 0 <= Discriminant(); }

    std::pair<int32_t, int32_t> Solution() const;

private:
    int32_t a_;
    int32_t b_;
    int32_t c_;
};

std::pair<int32_t, int32_t> QuadraticEquation::Solution() const
{
    if (!HasRealNumberSolution()) {
        throw std::invalid_argument("solution is an imaginary number");
    }
}
```

```

auto a0 = static_cast<int32_t>((-b_ - std::sqrt(Discriminant())) / 2);
auto a1 = static_cast<int32_t>((-b_ + std::sqrt(Discriminant())) / 2);

return {a0, a1};
}

```

サイクロマティック複雑度

サイクロマティック複雑度とは関数の複雑さを表すメトリクスである。このメトリクスの解釈は諸説あるものの、概ね以下のテーブルのようなものである。

サイクロマティック複雑度(CC)	複雑さの状態
CC <= 10	非常に良い構造
11 < CC < 30	やや複雑
31 < CC < 50	構造的なリスクあり
51 < CC	テスト不可能、デグレードリスクが非常に高い

Spurious Wakeup

Spurious Wakeupとは、条件変数に対する通知待ちの状態であるスレッドが、その通知がされていないにもかかわらず、起き上がってしまう現象のことである。

下記のようなstd::condition_variableの使用で起こり得る。

```

// @@@ example/term_explanation/spurious_wakeup_ut.cpp 8

namespace {
std::mutex           mutex;
std::condition_variable cond_var;
} // namespace

void notify_wrong() // 通知を行うスレッドが呼び出す関数
{
    auto lock = std::lock_guard{mutex};

    cond_var.notify_all(); // wait()で待ち状態のスレッドを起こす。
}

void wait_wrong() // 通知待ちスレッドが呼び出す関数
{
    auto lock = std::unique_lock{mutex};

    // notifyされるのを待つ。
    cond_var.wait(lock); // notify_allされなくとも起き上がってしまうことがある。

    // do something
}

```

std::condition_variable::wait()の第2引数を下記のようにすることでこの現象を回避できる。

```

// @@@ example/term_explanation/spurious_wakeup_ut.cpp 34

namespace {
bool           event_occurred{false};
std::mutex     mutex;
std::condition_variable cond_var;
} // namespace

void notify_right() // 通知を行うスレッドが呼び出す関数
{
    auto lock = std::lock_guard{mutex};

    event_occurred = true;

    cond_var.notify_all(); // wait()で待ち状態のスレッドを起こす。
}

void wait_right() // 通知待ちスレッドが呼び出す関数
{
    auto lock = std::unique_lock{mutex};

    // notifyされるのを待つ。
    cond_var.wait(lock, []() noexcept { return event_occurred; }); // Spurious Wakeup対策
}

```

```

    event_occurred = false;

    // do something
}

```

副作用

プログラミングにおいて、式の評価による作用には、主たる作用とそれ以外の副作用(side effect)がある。式は、評価値を得ること(関数では「引数を受け取り値を返す」と表現する)が主たる作用とされ、それ以外のコンピュータの論理的状態(ローカル環境以外の状態変数の値)を変化させる作用を副作用という。副作用の例としては、グローバル変数や静的ローカル変数の変更、ファイルの読み書き等のI/O実行、等がある。

is-a

「is-a」の関係は、オブジェクト指向プログラミング(OOP)においてクラス間の継承関係を説明する際に使われる概念である。クラス DerivedとBaseが「is-a」の関係である場合、DerivedがBaseの派生クラスであり、Baseの特性をDerivedが引き継いでいることを意味する。C++でのOOPでは、DerivedはBaseのpublic継承として定義される。通常DerivedやBaseは以下の条件を満たす必要がある。

- Baseはvirtualメンバ関数(Base::f)を持つ。
- DerivedはBase::fのオーバーライド関数を持つ。
- DerivedはBaseに対して リスコフの置換原則を守る必要がある。この原則を簡単に説明すると、「派生クラスのオブジェクトは、いつでもその基底クラスのオブジェクトと置き換えて、プログラムの動作に悪影響を与えることなく問題が発生してはならない」という設計の制約である。

「is-a」の関係とは「一種の～」と言い換えることができることが多い。ペンギンや九官鳥は一種の鳥であるため、この関係を使用したコード例を次に示す。

```

// @@@ example/term_explanation/class_relation_ut.cpp 11

class bird {
public:
    // 事前条件: altitude > 0 でなければならない
    // 事後条件: 呼び出しが成功した場合、is_flyingがtrueを返すことである
    virtual void fly(int altitude)
    {
        if (not(altitude > 0)) { // 高度(altitude)は0より大きくなければ、飛べない
            throw std::invalid_argument("altitude error");
        }
        altitude_ = altitude;
    }

    bool is_flying() const noexcept
    {
        return altitude_ != 0; // 高度が0でなければ、飛んでいると判断
    }

    virtual ~bird() = default;
};

private:
    int altitude_ = 0;
};

class kyukancho : public bird {
public:
    void speak()
    {
        // しゃべるため処理
    }

    // このクラスにget_nameを追加した理由はこの後を読めばわかる
    virtual std::string get_name() const // その個体の名前を返す
    {
        return "no name";
    }
};

```

bird::flyのオーバーライド(penguin::fly)について、リスコフの置換原則に反した例を下記する。

```

// @@@ example/term_explanation/class_relation_ut.cpp 50

class penguin : public bird {
public:
    void fly(int altitude) override

```

```

    {
        if (altitude != 0) {
            throw std::invalid_argument("altitude error");
        }
    };
}

// ...

auto let_it_fly = [] (bird& b, int altitude) {
    try {
        b.fly(altitude);
    }
    catch (std::exception const&) {
        return 0; // エクセプションが発生した
    }

    return b.is_flying() ? 2 : 1; // is_flyingがfalseなら1を返す
};

bird b;
penguin p;
ASSERT_EQ(let_it_fly(p, 0), 1); // パスする
// birdからpenguinへの派生がリスコフ置換の原則を満たすのであれば、
// 上記のテストのpをbで置き換えたテストがパスしなければならないが、
// 実際には逆に下記テストがパスしてしまう
ASSERT_NE(let_it_fly(b, 0), 1);
// このことからpenguinへの派生はリスコフ置換の原則を満たさない

```

birdからpenguinへの派生がリスコフ置換の原則に反してしまった原因は以下のように考えることができる。

- bird::flyの事前条件penguin::flyが強めた
- bird::flyの事後条件をpenguin::flyが弱めた

penguinとbirdの関係はis-aの関係ではあるが、上記コードの問題によって不適切なis-aの関係と言わざるを得ない。

上記の例では鳥全般と鳥の種類のis-a関係をpublic継承を使用して表した(一部不適切であるもの)。さらにis-aの誤った適用例を示す。自身が飼っている九官鳥に”キューちゃん”と名付けることはよくあることである。キューちゃんという名前の九官鳥は一種の九官鳥であることは間違いないことであるが、このis-aの関係を表すためにpublic継承を使用するのは、is-aの関係の誤用になることが多い。実際のコード例を以下に示す。この場合、型とインスタンスの概念の混乱が原因だと思われる。

```

// @@@ example/term_explanation/class_relation_ut.cpp 92

class q_chan : public kyukancho {
public:
    std::string get_name() const override { return "キューちゃん"; }
};

```

この誤用を改めた例を以下に示す。

```

// @@@ example/term_explanation/class_relation_ut.cpp 114

class kyukancho {
public:
    kyukancho(std::string name) : name_{std::move(name)} {}

    std::string const& get_name() const // 名称をメンバ変数で保持するため、virtualである必要はない
    {
        return name_;
    }

    virtual ~kyukancho() = default;

private:
    std::string const name_; // 名称の保持
};

// ...

kyukancho q{"キューちゃん"};

ASSERT_EQ("キューちゃん", q.get_name());

```

修正されたKyukanchoはstd::stringインスタンスをメンバ変数として持ち、kyukanchoとstd::stringの関係をhas-aの関係と呼ぶ。

has-a

「has-a」の関係は、あるクラスのインスタンスが別のクラスのインスタンスを構成要素として含む関係を指す。つまり、あるクラスのオブジェクトが別のクラスのオブジェクトを保持している関係である。

例えば、CarクラスとEngineクラスがあるとする。CarクラスはEngineクラスのインスタンスを含むので、CarはEngineを「has-a」の関係にあると言える。通常、has-aの関係はクラス内でメンバ変数またはメンバオブジェクトとして実装される。Carクラスの例ではCarクラスにはEngine型のメンバ変数が存在する。

```
// @@@ example/term_explanation/class_relation_ut.cpp 145

class Engine {
public:
    void start() {} // エンジンを始動するための処理
    void stop() {} // エンジンを停止するための処理

private:
    // ...
};

class Car {
public:
    Car() : engine_{} {}
    void start() { engine_.start(); }
    void stop() { engine_.stop(); }

private:
    Engine engine_; // Car は Engine を持っている (has-a)
};
```

is-implemented-in-terms-of

「is-implemented-in-terms-of」の関係は、オブジェクト指向プログラミング（OOP）において、あるクラスが別のクラスの機能を内部的に利用して実装されていることを示す概念である。これは、あるクラスが他のクラスのインターフェースやメソッドを用いて、自身の機能を提供する場合に使われる。has-aの関係は、is-implemented-in-terms-ofの関係の一種である。

is-implemented-in-terms-ofは下記の手段1-3に示した方法がある。

手段1. public継承によるis-implemented-in-terms-of

手段2. private継承によるis-implemented-in-terms-of

手段3. コンポジションによる(has-a)is-implemented-in-terms-of

手段1-3にはそれぞれ、長所、短所があるため、必要に応じて手段を選択する必要がある。以下の議論を単純にするため、下記のようにクラスS、C、CCを定める。

- S(サーバー): 実装を提供するクラス
- C(クライアント): Sの実装を利用するクラス
- CC(クライアントのクライアント): Cのメンバを使用するクラス

コード量の観点から考えた場合、手段1が最も優れていることが多い。依存関係の複雑さから考えた場合、CはSに強く依存する。場合によっては、この依存はCCからSへの依存間にも影響を及ぼす。従って、手段3が依存関係を単純にしやすい。手段1はis-aに見え、以下に示すような問題も考慮する必要があるため、可読性、保守性を劣化させる可能性がある。

```
// @@@ example/term_explanation/class_relation_ut.cpp 261

class MyString : public std::string { // 手段1
};

// ...
std::string* m_str = new MyString("str");

// このようなpublic継承を行う場合、基底クラスのデストラクタは非virtualであるため、
// 以下のコードでは~my_stringのデストラクタは呼び出されない。
// この問題はリソースリークを発生させる場合がある。
delete m_str;
```

以上述べたように問題の多い手段1であるが、実践的には有用なパターンであり、CRTTP(curiously recurring template pattern)の実現手段でもあるため、一概にコーディング規約などで排除することもできない。

public継承によるis-implemented-in-terms-of

public継承によるis-implemented-in-terms-ofの実装例を以下に示す。

```
// @@@ example/term_explanation/class_relation_ut.cpp 283

class MyString : public std::string {};

// ...
MyString str{"str"};

ASSERT_EQ(str[0], 's');
ASSERT_STREQ(str.c_str(), "str");

str.clear();
ASSERT_EQ(str.size(), 0);
```

すでに述べたようにこの方法は、private継承によるis-implemented-in-terms-ofや、コンポジションによる(has-a)is-implemented-in-terms-ofと比べコードがシンプルになる。

private継承によるis-implemented-in-terms-of

private継承によるis-implemented-in-terms-ofの実装例を以下に示す。

```
// @@@ example/term_explanation/class_relation_ut.cpp 180

class MyString : std::string {
public:
    using std::string::string;
    using std::string::operator[];
    using std::string::c_str;
    using std::string::clear;
    using std::string::size;
};

// ...
MyString str{"str"};

ASSERT_EQ(str[0], 's');
ASSERT_STREQ(str.c_str(), "str");

str.clear();
ASSERT_EQ(str.size(), 0);
```

この方法は、public継承によるis-implemented-in-terms-ofが持つデストラクタ問題は発生せず、is-aと誤解してしまう問題も発生しない。

コンポジションによる(has-a)is-implemented-in-terms-of

コンポジションによる(has-a)is-implemented-in-terms-ofの実装例を示す。

```
// @@@ example/term_explanation/class_relation_ut.cpp 208

namespace is_implemented_in_terms_of_1 {
class MyString {
public:
    // コンストラクタ
    MyString() = default;
    MyString(const std::string& str) : str_(str) {}
    MyString(const char* cstr) : str_(cstr) {}

    // 文字列へのアクセス
    const char* c_str() const { return str_.c_str(); }

    using reference = std::string::reference;
    using size_type = std::string::size_type;

    reference operator[](size_type pos) { return str_[pos]; }

    // 他のメソッドも必要に応じて追加する
    // 以下は例
    std::size_t size() const { return str_.size(); }

    void clear() { str_.clear(); }

    MyString& operator+=(const MyString& rhs)
    {
        str_ += rhs.str_;
        return *this;
    }
private:
    std::string str_;
};
```

```
};

// ...
MyString str{"str"};

ASSERT_EQ(str[0], 's');
ASSERT_STREQ(str.c_str(), "str");

str.clear();
ASSERT_EQ(str.size(), 0);
```

この方は実装を利用するクラストの依存関係を他の2つに比べるとシンプルにできるが、逆に実装例から昭などおり、コード量が増えてしまう。

非ソフトウェア用語

割れ窓理論

割れ窓理論とは、軽微な犯罪も徹底的に取り締まることで、凶悪犯罪を含めた犯罪を抑止できるとする環境犯罪学上の理論。アメリカの犯罪学者ジョージ・ケリングが考案した。「建物の窓が壊れているのを放置すると、誰も注意を払っていないという象徴になり、やがて他の窓もまもなく全て壊される」との考え方からこの名がある。

ソフトウェア開発での割れ窓とは、「朝会に数分遅刻する」、「プログラミング規約を守らない」等の軽微なルール違反を指し、この理論の実践には、このような問題を放置しないことによって、

- チームのモラルハザードを防ぐ
- コードの品質を高く保つ

等の重要な狙いがある。

車輪の再発明

車輪の再発明とは、広く受け入れられ確立されている技術や解決法を（知らずに、または意図的に無視して）再び一から作ること」を指すための慣用句である。ソフトウェア開発では、STLのような優れたライブラリを使わずに、それと同様なライブラリを自分たちで実装するような非効率な様を指すことが多い。

演習

本章では、これまで解説した内容の復習やより深い理解のために、解選択問題とプログラミング記述問題を提供する。

多くのプログラミング記述問題は、google testを含むC++のソースコードと以下のようなコードコメントで与えられる。

```
// [Q]
// 以下のxxxをせよ。
```

プログラミング規約(型)

演習-汎整数型の選択

- 問題

汎整数型には、特に理由がない限りに、int32_tを使用するべきであるが、値が負にならない場合は、(A)を使用するべきである。
(A)に相応しいものを下記より選べ。

- 選択肢

1. int32_t
2. unsigned long
3. uint32_t
4. uint16_t

- 解答-汎整数型の選択

演習-汎整数型の演算

- 問題

```
// @@@ exercise/programming_convention_q/type.cpp 7
TEST(ProgrammingConventionTypeQ, GeneralInteger)
{
    // [Q]
    // 以下の組み込み型の使用方法は、その下のテストコードを(環境依存で)パスするが、
    // 適切であるとは言えない。適切な型に修正せよ。
    auto b = true;
    int i{b};
    char c{-1};

    ASSERT_EQ(i * c, -1);
}
```

- 参照 算術型
- 解答例-汎整数型の演算

演習-浮動小数点型

- 問題

```
// @@@ exercise/programming_convention_q/type.cpp 21
double f(double a) noexcept { return 1 / a; }

TEST(ProgrammingConventionTypeQ, Float)
{
    // [Q]
    // 以下の両辺を同一と判定するための関数を作り、その関数の単体テストを行え。
    ASSERT_FALSE(1.0 == 1 + 0.001 - 0.001);

    // [Q]
    // 以下の0除算を捕捉するためのコードを書け。
    f(0.0);
}
```

- 参照 浮動小数点型
- 解答例-浮動小数点型

演習-定数列挙

- 問題
特に理由がない限り、一連の定数の列挙には(A)を定義して使用する。
(A)に相応しいものを下記より選べ。
- 選択肢
 - マクロ定数
 - static const定数
 - 非スコープenum
 - スコープenum
- 解答-定数列挙

演習-enum

- 問題

```
// @@@ exercise/programming_convention_q/type.cpp 36
// [Q]
// 以下のマクロ引数を型安全なenumに修正せよ

#define COLOR_RED 0
#define COLOR_GREEN 1
#define COLOR_BLUE 2

std::string GetString(int color)
{
    switch (color) {
        case COLOR_RED:
            return "Red";
        case COLOR_GREEN:
            return "Green";
        case COLOR_BLUE:
            return "Blue";
        default:
            assert(false);
            return "";
    }
}

TEST(ProgrammingConventionTypeQ, Enum)
{
    ASSERT_EQ(std::string("Red"), GetString(COLOR_RED));
    ASSERT_EQ(std::string("Green"), GetString(COLOR_GREEN));
    ASSERT_EQ(std::string("Blue"), GetString(COLOR_BLUE));
}
```

- 参照 [enum](#)
- 解答例-enum

演習-配列の範囲for文

- 問題

```
// @@@ exercise/programming_convention_q/type.cpp 67
int32_t array_value() noexcept
{
    static int32_t i;

    return i++;
}

TEST(ProgrammingConventionTypeQ, Array)
{
    // [Q]
    // 以下の配列の値の設定を範囲for文を使って書き直せ
    int32_t array[10];

    for (auto i = 0U; i < sizeof(array) / sizeof(array[0]); ++i) {
        array[i] = array_value();
    }
}
```

```
    ASSERT_EQ(0, array[0]);
    ASSERT_EQ(3, array[3]);
    ASSERT_EQ(9, array[9]);
}
```

- 参照配列
- 解答例-配列の範囲for文

演習-エイリアス

- 問題

```
// @@@ exercise/programming_convention_q/type.cpp 91
// [Q]
// 以下のtypedefをC++11から導入された新しい形式のエイリアスに直せ。
typedef unsigned char uchar;
typedef bool (*func_type)(int32_t);

// [Q]
// template引数で与えられた型のオブジェクトをstd::vectorで保持するエイリアスtemplateを
// 定義し、その単体テストを行え。
```

- 参照型エイリアス
- 解答例-エイリアス

演習-constの意味

- 問題

初期化後に変更されない変数はconstと宣言すべきであるが、以下のコードの2つのconstの意味にふさわしいものを選択せよ。

```
char const* const country = "japan";
```

- 選択肢
 - 左側のconstはcountryが指す先が不変。右側のconstは、country自体が不変。
 - 左側のconstはcountry自体が不変。右側のconstは、countryが指す先が不変。
 - 左右ともconstはcountry自体が不変。
 - 上記のどれでもない。
- 解答-constの意味

演習-const/constexpr

- 問題

```
// @@@ exercise/programming_convention_q/type.cpp 103
// [Q]
// 下記のStringHolderに「const/constexprを付加する」等を行い、より良いコードに修正せよ。
```

```
class StringHolder {
public:
    StringHolder() = default;
    void Add(std::string str)
    {
        if (vector_len_max_ > strings_.size()) {
            strings_.push_back(str);
        }
    }

    std::vector<std::string> GetStrings() const { return strings_; }

private:
    size_t vector_len_max_{3};
    std::vector<std::string> strings_{};
};

TEST(ProgrammingConventionTypeQ, ConstConstexpr)
{
    auto sh = StringHolder{};

    ASSERT_EQ(std::vector<std::string>{}, sh.GetStrings());

    sh.Add("a");
    sh.Add(std::string("bc"));
    ASSERT_EQ((std::vector<std::string>{"a", "bc"}), sh.GetStrings());
}
```

```

    sh.Add("def");
    sh.Add(std::string("g"));
    ASSERT_EQ(std::vector<std::string> {"a", "bc", "def"}, sh.GetStrings());
}

```

- 参照 [const/constexprインスタンス](#)
- [解答例-const/constexpr](#)

演習-危険なconst_cast

- 問題

```

// @@@ exercise/programming_convention_q/type.cpp 140
// [Q]
// 下記の"DISABLED_"を削除し、何が起こるのか、なぜそうなるのかを確かめた上で、
// nameの型やその初期化を行っているコードを修正せよ。
TEST(DISABLED_ProgrammingConventionQ, ConstConstexpr2)
{
    char* name = const_cast<char*>("abcdef");

    for (auto i = 0U; name[i] != '\0'; ++i) {
        name[i] = std::toupper(name[i]);
    }

    ASSERT_STREQ("ABCDEF", name);
    ASSERT_EQ("ABCDEF", std::string{name});
}

```

- 参照 [const/constexprインスタンス](#)
- [解答例-危険なconst_cast](#)

演習-リテラル

- 問題

```

// @@@ exercise/programming_convention_q/type.cpp 157
int32_t literal_test(int64_t) noexcept { return 0; }
int32_t literal_test(int32_t*) noexcept { return 1; }

// [Q]
// 下記変数の初期化コードをコメントに基づき適切に修正せよ。
TEST(ProgrammingConventionTypeQ, Literal)
{
    int32_t* p{NULL}; // NULLは使用不可
    uint64_t a{0x1234567890abcdef}; // 適切なセパレータを挿入
    int32_t b{0x715}; // ビット表現に修正

    // [Q]
    // 下記resultはfalseになるが、その理由を述べ、trueになるようにコードを修正せよ。
    bool const result{(literal_test(NULL) == literal_test(p))};
    ASSERT_FALSE(result);

    ASSERT_EQ(0x1234567890abcdef, a);
    ASSERT_EQ(b, 0x715);
}

```

- 参照 [リテラル](#)
- [解答例-リテラル](#)

演習-適切なautoの使い方

- 問題

以下のコードのautoの中で、使い方が好ましくないものを選べ。

- 選択肢

1. auto s0 = std::string("xxx");
2. auto s1(std::string("xxx"));
3. auto s2 = s0; // s0は上記に定義されたもの
4. auto s3 = get_name(); // get_name()は離れた場所に宣言

- [解答-適切なautoの使い方](#)

演習-ポインタの初期化

- 問題

初期値が定まらないポインタ変数の初期化方法にふさわしいものを選べ。

- 選択肢

- 0を代入する。
- NULLを代入する。
- nullptrを代入する。
- 何もしない。

- 解答-ポインタの初期化

演習-vector初期化

- 問題

以下のコード実行後のvecの状態を選べ。

```
vector<int32_t> vec{10};
```

- 選択肢

- vec.size()が1で、vec[0]は10
- vec.size()が10で、vec[0]～vec[9]は不定
- vec.size()が1で、vec[0]は不定
- vec.size()が10で、vec[0]～vec[9]は0

- 解答-vector初期化

演習-インスタンスの初期化

- 問題

```
// @@@ exercise/programming_convention_q/type.cpp 179
TEST(ProgrammingConventionTypeQ, Initialization)
{
    // [Q]
    // 変数a、b、v、wの定義と初期化を1文で行え。
    {
        int32_t a[3];

        for (auto& r : a) {
            r = 1;
        }

        ASSERT_EQ(1, a[0]);
        ASSERT_EQ(1, a[1]);
        ASSERT_EQ(1, a[2]);
    }

    {
        int32_t b[3];

        for (auto& r : b) {
            r = 0;
        }

        ASSERT_EQ(0, b[0]);
        ASSERT_EQ(0, b[1]);
        ASSERT_EQ(0, b[2]);
    }

    auto v = std::vector<std::string>{3};

    for (auto& r : v) {
        r = "1";
    }

    ASSERT_EQ("1", v[0]);
    ASSERT_EQ("1", v[1]);
    ASSERT_EQ("1", v[2]);
}

{
    auto w = std::vector<std::string>{};
}
```

```

        for (auto i = 0; i < 3; ++i) {
            w.emplace_back(std::to_string(i));
        }

        ASSERT_EQ("0", w[0]);
        ASSERT_EQ("1", w[1]);
        ASSERT_EQ("2", w[2]);
    }
}

```

- 参照 インスタンスの初期化、一様初期化
- 解答例-インスタンスの初期化

プログラミング規約(クラス)

演習-凝集度の意味

• 問題

クラスの凝集度とはLack of Cohesion in Methodsというメトリクスで計測される。この凝集度やメトリクスの説明として正しくないものを選べ。

• 選択肢

1. クラスは凝集度を高くするように設計すべきである。
2. このメトリクス値が1に近ければ凝集度は低く、この値が0に近ければ凝集度は高い。
3. メンバ変数やメンバ関数が多くなれば、凝集度は低くなりやすい。
4. setterやgetterを使用することで、凝集度を高く保つことができる。

• 解答-凝集度の意味

演習-凝集度の向上

• 問題

```

// @@@ exercise/programming_convention_q/class.cpp 7
// [Q]
// 以下のクラスABCの凝集度が高くなるように、ABC、HasRealNumberSolutionをリファクタリングせよ。
// その時に、他の問題があれば併せて修正せよ。

class ABC { // 2次方程式のパラメータ保持
public:
    ABC(int32_t a, int32_t b, int32_t c) noexcept : a_{a}, b_{b}, c_{c} {}

    int32_t GetA() const { return a_; }
    int32_t GetB() const { return b_; }
    int32_t GetC() const { return c_; }
    void SetA(int32_t a) { a_ = a; }
    void SetB(int32_t b) { b_ = b; }
    void SetC(int32_t c) { c_ = c; }

private:
    int32_t a_;
    int32_t b_;
    int32_t c_;
};

bool HasRealNumberSolution(ABC abc)
{
    auto const discriminant = abc.GetB() * abc.GetB() - 4 * abc.GetA() * abc.GetC(); // 判定式

    return 0 <= discriminant;
}

TEST(ProgrammingConventionClassQ, Cohesion)
{
    {
        auto abc = ABC{1, 2, 1};

        ASSERT_TRUE(HasRealNumberSolution(abc));
    }
    {
        auto abc = ABC{2, 0, 1};
    }
}

```

```
        ASSERT_FALSE(HasRealNumberSolution(abc));
    }
}
```

- 参照 [凝集度](#)
- [解答例-凝集度の向上](#)

演習-メンバ変数の初期化方法の選択

- 問題
非静的なメンバ変数の初期化には下記の3つの方法がある。これらの説明として誤っているものを下記選択肢より選べ。
 - 初期化方法 0: 非静的メンバ変数の初期化子による初期化
 - 初期化方法 1: コンストラクタの非静的メンバ初期化子による初期化
 - 初期化方法 2: コンストラクタ内での非静的メンバ変数の初期化
- 選択肢
 1. 初期化方法 0 を優先して使う。
 2. どの方法も優劣はないので、任意に選択してよい。
 3. 初期化方法 0 が使えない場合は、なるべく初期化方法 1 を使う。
 4. 初期化方法 0、1 が使えない場合のみ初期化方法 2 を使う。
- [解答-メンバ変数の初期化方法の選択](#)

演習-メンバの型

- 問題
以下のA::strの意味を表すものを選べ。

```
class A {
    ...
private:
    std::string str(); // <- この意味
};
```

- 選択肢
 1. デフォルト初期化されたstd::stringオブジェクト
 2. 初期化されていないstd::stringオブジェクト
 3. std::stringオブジェクトを返す関数
 4. 以上のいずれでもない
- [解答-メンバの型](#)

演習-メンバ変数の初期化

- 問題

```
// @@@ exercise/programming_convention_q/class.cpp 51
// [Q]
// 以下のMemberInitのメンバ変数を適切な方法で初期化せよ。
```

```
class MemberInit {
public:
    MemberInit() noexcept
    {
        a_ = 0;
        b_[0] = 1;
        b_[1] = 1;
        c_ = 2;
    }

    explicit MemberInit(int a) noexcept
    {
        a_ = a;
        b_[0] = a;
        b_[1] = 99;
        c_ = 2;
    }

    int32_t GetA() noexcept { return a_; }
    int32_t* GetB() noexcept { return b_; }
```

```

int32_t      GetC() noexcept { return c_; }
static size_t b_len;

private:
    int32_t a_;
    int32_t b_[2];
    int32_t c_;
};

size_t MemberInit::b_len = 2;

TEST(ProgrammingConventionClassQ, MemberInit)
{
{
    auto mi = MemberInit{};

    ASSERT_EQ(0, mi.GetA());
    ASSERT_EQ(1, mi.GetB()[0]);
    ASSERT_EQ(1, mi.GetB()[1]);
    ASSERT_EQ(2, mi.GetC());
}
{
    auto mi = MemberInit{1};

    ASSERT_EQ(1, mi.GetA());
    ASSERT_EQ(1, mi.GetB()[0]);
    ASSERT_EQ(99, mi.GetB()[1]);
    ASSERT_EQ(2, mi.GetC());
}
}
}

```

- 参照 非静的なメンバ変数/定数の初期化
- 解答例-メンバ変数の初期化

演習-スライシング

- 問題

```

// @@@ exercise/programming_convention_q/class.cpp 106
// [Q]
// 以下のクラスBaseはオブジェクトのスライシングを引き起こす。
// このような誤用を起こさないようにするために、Baseオブジェクトのコピーを禁止せよ。
// 合わせてクラスDerivedも含め、不十分な記述を修正せよ。

class Base {
public:
    Base(char const* name = nullptr) noexcept : name_{name == nullptr ? "Base" : name} {}
    ~Base() = default;

    virtual char const* Name0() { return "Base"; }
    char const*       Name1() { return name_; }

private:
    char const* name_;
};

class Derived final : public Base {
public:
    Derived() noexcept : Base{"Derived"} {}

    char const* Name0() { return "Derived"; }
};

TEST(ProgrammingConventionClassQ, Slicing)
{
    auto b      = Base{};
    auto d      = Derived{};
    Base& d_ref = d;

    // 以下はBase、Derivedの単純なテスト
    ASSERT_STREQ("Base", b.Name0());
    ASSERT_STREQ("Base", b.Name1());
    ASSERT_STREQ("Derived", d.ref.Name0());
    ASSERT_STREQ("Derived", d.ref.Name1());

    // 以下はbがスライスされたオブジェクトであることのテスト
    // こういった誤用を防ぐためにBaseのコピーを禁止せよ。
    b = d.ref;
    ASSERT_STREQ("Base", b.Name0()); // vtblはBaseになるから
}

```

```
    ASSERT_STREQ("Derived", b.Name1()); // name_はコピーされるから
}
```

- 参照スライシング
- 解答例-スライシング

演習-オブジェクトの所有権

- 問題

```
// @@@ exercise/programming_convention_q/class.cpp 151
class A {
public:
    explicit A(int32_t n) noexcept : num_{n} { last_constructed_num_ = num_; }
    ~A() { last_destructed_num_ = num_; }

    int32_t GetNum() const noexcept { return num_; }

    static int32_t LastConstructedNum() noexcept { return last_constructed_num_; }
    static int32_t LastDestructedNum() noexcept { return last_destructed_num_; }

private:
    int32_t         num_;
    static int32_t last_constructed_num_;
    static int32_t last_destructed_num_;
};

int32_t A::last_constructed_num_ = -1;
int32_t A::last_destructed_num_ = -1;

class X final {
public:
    void Move(std::unique_ptr<A>&& ptr) noexcept
    {
        ptr_ = std::move(ptr); // ptr->ptr_へ所有権の移動
    }

    std::unique_ptr<A> Release() noexcept
    {
        return std::move(ptr_); // ptr_から外部への所有権の移動
    }

    A const* GetA() const noexcept { return ptr_.get(); }
    X() = default;
    ~X() = default;

private:
    std::unique_ptr<A> ptr_{};
};

TEST(ProgrammingConventionClassQ, Ownership)
{
    // [Q]
    // 以下の単体テストを完成させよ。
    // ?はインスタンスが入り、?????はTRUEかFALSEが入る。

    ASSERT_EQ(-1, A::LastConstructedNum()); // まだ、A::A()は呼ばれてない
    ASSERT_EQ(-1, A::LastDestructedNum()); // まだ、A::~A()は呼ばれてない

    auto a0 = std::make_unique<A>(0); // a0はA(0)を所有
    auto a1 = std::make_unique<A>(1); // a1はA(1)を所有
    auto x = X{};

    // ASSERT_EQ(? , A::LastConstructedNum()); // A(1)は生成された
    // ASSERT_EQ(? , A::LastDestructedNum()); // まだ、A::~A()は呼ばれてない
    // ASSERT_EQ(? , a0->GetNum()); // a0はA(0)を所有
    // x.Move(std::move(a0)); // a0からxへA(0)の所有権の移動
    // ASSERT_EQ(? , a0); // a0は何も所有していない

    // ASSERT_EQ(? , a1->GetNum()); // a1はA(1)を所有
    // x.Move(std::move(a1)); // xによるA(0)の解放
    // // a1からxへA(1)の所有権の移動

    // ASSERT_EQ(? , A::LastDestructedNum()); // A(0)は解放された
    // ASSERT_EQ(? , a1); // a1は何も所有していない
    // ASSERT_EQ(? , x.GetA()->GetNum()); // xはA(1)を所有

    std::unique_ptr<A> a2{x.Release()}; // xからa2へA(1)の所有権の移動
    ASSERT_EQ(? , x.GetA()); // xは何も所有していない
    ASSERT_EQ(? , a2->GetNum()); // a2はA(1)を所有
}
```

```

    {
        std::unique_ptr<A> a3{std::move(a2)};
        // ASSERT_?????(a2); // a2は何も所有していない
        // ASSERT_EQ(?, a3->GetNum()); // a3はA(1)を所有
    }
    // ASSERT_EQ(?, A::LastDestructedNum());
}

```

- 参照 オブジェクトの所有権
- 解答例-オブジェクトの所有権

プログラミング規約(関数)

演習-非メンバ関数の宣言

- 問題

```

// @@@ exercise/programming_convention_q/func.cpp 5
extern "C" double cos(double x);

TEST(ProgrammingConventionFuncQ, NonMemberFunc)
{
    // [Q]
    // 適切な#includeを追加し、上記のextern宣言がなくとも下記がコンパイルできるようにせよ。
    ASSERT_EQ(1, cos(0));
}

```

- 参照 非メンバ関数
- 解答例-非メンバ関数の宣言

演習-メンバ関数の修飾

- 問題

```

// @@@ exercise/programming_convention_q/func.cpp 17
// [Q]
// 下記のクラスAのメンバ関数の不正確な記述を修正せよ。
// また、単体テストを同様に修正せよ。

class A {
public:
    A() : strings_{GetStringsDefault()} {}

    void SetStrings(size_t index, std::string str)
    {
        if (index < max_len) {
            strings_[index] = str;
        }
    }

    std::vector<std::string>& GetStrings() { return strings_; }
    std::vector<std::string> const GetStrings() const { return strings_; }
    size_t MaxLen() { return max_len; }

    std::vector<std::string>& GetStringsDefault()
    {
        static auto strings_default = std::vector<std::string>{max_len, ""};
        return strings_default;
    }

private:
    std::vector<std::string> strings_;
    static constexpr size_t max_len{3};
};

TEST(ProgrammingConventionFuncQ, MemberFunc)
{
    auto a = A{};

    auto strings_default = a.GetStringsDefault();
    ASSERT_EQ("", strings_default[0]);
    ASSERT_EQ("", strings_default[1]);
    ASSERT_EQ("", strings_default[2]);
    ASSERT_EQ(3, a.MaxLen());
}

```

```

auto strings = a.GetStrings();

ASSERT_EQ("", strings[0]);
ASSERT_EQ("", strings[1]);
ASSERT_EQ("", strings[2]);

a.SetStrings(1, "TEST");
ASSERT_EQ("", strings[0]);

// [Q]
// このテストをASSERT_EQでパスできるようにせよ
ASSERT_NE("TEST", strings[1]);

ASSERT_EQ("", strings[2]);
}

```

- 参照 メンバ関数
- 解答例-メンバ関数の修飾

演習-特殊メンバ関数の削除

- 問題

```

// @@@ exercise/programming_convention_q/func.cpp 75
// [Q]
// 下記クラスAutoGenのコンパイラが自動生成するメンバ関数を生成しないようにせよ。

class AutoGen {};

```

- 参照 特殊メンバ関数
- 解答例-特殊メンバ関数の削除

演習-委譲コンストラクタ

- 問題

```

// @@@ exercise/programming_convention_q/func.cpp 82
// [Q]
// 下記クラスDelConstructorの2つのコンストラクタのコードクローンができるだけ排除せよ。
// また、不正確な記述を修正せよ。

class DelConstructor {
public:
    DelConstructor(std::string const& str) : str0_{str + "0"}, str1_{str + "1"}, str2_{str + "2"} {}

    DelConstructor(int32_t num)
        : str0_{std::to_string(num) + "_0"},
          str1_{std::to_string(num) + "_1"},
          str2_{std::to_string(num) + "_2"}
    {}

    std::string& GetString0() { return str0_; }
    std::string& GetString1() { return str1_; }
    std::string& GetString2() { return str2_; }

private:
    std::string str0_;
    std::string str1_;
    std::string str2_;
};

TEST(ProgrammingConventionFuncQ, Constructor)
{
    {
        auto dc = DelConstructor{"hehe"};

        ASSERT_EQ("hehe0", dc.GetString0());
        ASSERT_EQ("hehe1", dc.GetString1());
        ASSERT_EQ("hehe2", dc.GetString2());
    }
    {
        auto dc = DelConstructor{123};

        ASSERT_EQ("123_0", dc.GetString0());
        ASSERT_EQ("123_1", dc.GetString1());
        ASSERT_EQ("123_2", dc.GetString2());
    }
}

```

```
}
```

- 参照 [コンストラクタ](#)
- [解答例-委譲コンストラクタ](#)

演習-copyコンストラクタ

- 問題

```
// @@@ exercise/programming_convention_q/func.cpp 127
// [Q]
// 下記クラスInteger、IntegerHolderに適切にcopyコンストラクタ、copy代入演算子を追加して、
// 単体テストを行え(DISABLED_削除)。
class Integer {
public:
    explicit Integer(int32_t i) noexcept : i_{i} {}

    int32_t GetValue() const noexcept { return i_; }

private:
    int32_t i_;
};

class IntegerHolder {
public:
    explicit IntegerHolder(int32_t i) : integer_{new Integer{i}} {}

    int32_t GetValue() const noexcept { return integer_->GetValue(); }

    ~IntegerHolder() { delete integer_; }

private:
    Integer* integer_;
};

#ifndef __clang_analyzer__
TEST(DISABLED_ProgrammingConventionFuncQ, Constructor2)
{
    {
        auto i = Integer{3};
        ASSERT_EQ(3, i.GetValue());

        auto j = Integer{i};
        ASSERT_EQ(3, j.GetValue());

        auto k = Integer{0};
        ASSERT_EQ(0, k.GetValue());

        k = i;
        ASSERT_EQ(3, k.GetValue());
    }
    {
        auto i = IntegerHolder{3};
        ASSERT_EQ(3, i.GetValue());

        auto j = IntegerHolder{i};
        ASSERT_EQ(3, j.GetValue());

        auto k = IntegerHolder{0};
        ASSERT_EQ(0, k.GetValue());

        k = i;
        ASSERT_EQ(3, k.GetValue());
    }
}
#endif
```

- 参照 [コンストラクタ](#)
- [解答例-copyコンストラクタ](#)

演習-moveコンストラクタ

- 問題

```
// @@@ exercise/programming_convention_q/func.cpp 186
// [Q]
```

```
// 上記問題を解決したIntegerHolderにmoveコンストラクタ、move演算子を追加した  
// クラスIntegerHolder2を作成し、単体テストを行え。
```

- 参照 [moveコンストラクタ、move代入演算子](#)
- [解答例-moveコンストラクタ](#)

演習-関数分割

- 問題

```
// @@@ exercise/programming_convention_q/func.cpp 192  
// [Q]  
// 下記PrimeNumbersは引数で与えられた整数以下の素数を返す関数である。  
// PrimeNumbersの単体テストを作成し、その後、行数を短くする等のリファクタリングを行え。  
  
std::vector<uint32_t> PrimeNumbers(uint32_t max_number)  
{  
    auto result = std::vector<uint32_t>{};  
  
    if (max_number >= 2) {  
        auto prime_num = 2U; // 最初の素数  
        auto is_num_prime = std::vector<bool>(max_number + 1, true); // falseなら素数でない。  
        is_num_prime[0] = is_num_prime[1] = false;  
  
        do {  
            result.emplace_back(prime_num);  
  
            for (auto i = 2 * prime_num; i < is_num_prime.size(); i += prime_num) {  
                is_num_prime[i] = false;  
            }  
  
            do { // 次の素数の探索  
                prime_num++;  
            } while (!is_num_prime[prime_num] && (prime_num < is_num_prime.size()));  
        } while (prime_num < is_num_prime.size());  
  
        return result;  
    }  
    else {  
        return result;  
    }  
}  
  
TEST(ProgrammingConventionFuncQ, Lines)  
{  
    ASSERT_EQ((std::vector<uint32_t>{}), PrimeNumbers(0));  
    // 以下に単体テストを追加  
}
```

- 参照 [行数](#)
- [解答例-関数分割](#)

演習-オーバーライド関数の修飾

- 問題
- オーバーライドしたメンバ関数は、virtualと(A)を使用して宣言する。
(A)に相応しいものを下記より選べ。

- 選択肢

1. final
2. override
3. overload
4. const

- [解答-オーバーライド関数の修飾](#)

演習-オーバーライド/オーバーロード

- 問題

```
// @@@ exercise/programming_convention_q/func.cpp 233  
// [Q]
```

```

// 下記クラスBase、Derivedの単体テストを完成せよ。

class Base {
public:
    virtual ~Base() = default;
    int32_t f() noexcept { return 0; }

    virtual int32_t g() noexcept { return 0; }
};

class Derived : public Base {
public:
    int32_t f() noexcept { return 1; }

    virtual int32_t g() noexcept override { return 1; }
};

TEST(ProgrammingConventionFuncQ, Overload)
{
    auto b      = Base{};
    auto d      = Derived{};
    Base& d_ref = d;

#ifndef NDEBUG
    // 下記?は0もしくは1が入る
    ASSERT_EQ(0, b.f());
    ASSERT_EQ(0, b.g());

    ASSERT_EQ(1, d.f());
    ASSERT_EQ(1, d.g());

    ASSERT_EQ(1, d_ref.f());
    ASSERT_EQ(1, d_ref.g());
#endif
}

```

- 参照 [オーバーライド](#)
- 解答例-[オーバーライド/オーバーロード](#)

演習-オーバーロードによる誤用防止

- 問題

```

// @@@ exercise/programming_convention_q/func.cpp 272
// [Q]
// 下記関数Squareは、引数が浮動小数点となることを想定していない。
// 誤用を防ぐために、引数に浮動小数点を指定された場合、コンパイルできないようにせよ。
int32_t Square(int32_t a) noexcept { return a * a; }

TEST(ProgrammingConventionFuncQ, Overload2)
{
    ASSERT_EQ(9, Square(3));
    ASSERT_EQ(4, Square(2.5)); // この誤用を防ぐためにコンパイルエラーにせよ。
}

```

- 参照 [オーバーロード](#)
- 解答例-[オーバーロードによる誤用防止](#)

演習-仮引数の修飾

- 問題

```

// @@@ exercise/programming_convention_q/func.cpp 285
// [Q]
// 下記AddStringsの仮引数等を適切に修正せよ。
using Strings = std::list<std::string>;
void AddStrings(Strings a, Strings* b, Strings* ret)
{
    *ret = a;

    if (b == nullptr) {
        return;
    }

    ret->insert(ret->end(), b->begin(), b->end());
}

```

```

TEST(ProgrammingConventionFuncQ, Parameter)
{
    auto a = Strings{"abc", "d"};
    auto b = Strings{"e", "fgh", "i"};

    auto ret = Strings{};
    AddStrings(a, nullptr, &ret);
    ASSERT_EQ(ret, (Strings{"abc", "d"}));

    AddStrings(a, &b, &ret);
    ASSERT_EQ(ret, (Strings{"abc", "d", "e", "fgh", "i"}));
}

```

- 参照 [実引数/仮引数](#)
- [解答例-仮引数の修飾](#)

演習-constexpr関数

- 問題

```

// @@@ exercise/programming_convention_q/func.cpp 314
// [Q]
// 下記Factorialをconstexpr関数にせよ。
uint32_t Factorial(uint32_t a) noexcept
{
    if (a == 0 || a == 1) {
        return 1;
    }

    auto fact = 1U;

    for (auto i = 2U; i <= a; ++i) {
        fact *= i;
    }

    return fact;
}

TEST(ProgrammingConventionFuncQ, ConstexprFunc)
{
#if 0 // Factorialはconstexpr関数でないため、下記はコンパイルできない。
    static_assert(1 == Factorial(0), "Factorial fail");
#endif
    ASSERT_EQ(1, Factorial(0));
    ASSERT_EQ(6, Factorial(3));
    ASSERT_EQ(120, Factorial(5));
    ASSERT_EQ(3628800, Factorial(10));
}

```

- 参照 [constexpr関数](#)
- [解答例-constexpr関数](#)

演習-エクセプションの型

- 問題

try-catchでエクセプションを捕捉する場合、catchの中で宣言するエクセプション補足用の変数は(A)として定義する。(A)に相応しいものを下記より選べ。

- 選択肢

1. constポインタ
2. constリファレンス
3. 非constオブジェクト
4. constオブジェクト

- [解答-エクセプションの型](#)

プログラミング規約(構文)

演習-コンテナの範囲for文

- 問題

```
// @@@ exercise/programming_convention_q/syntax.cpp 8
// [Q]
// 下記Accumulateのfor文を
// * イテレータを使ったfor文を使用したAccumlate2
// * 範囲for文を使用したAccumlate3
// を作り、それらの単体テストを行え。また、その時にその他の不具合があれば合わせて修正せよ。

std::string Accumulate(std::vector<std::string> strings) noexcept
{
    auto ret = std::string{};

    for (auto i = 0U; i < strings.size(); ++i) {
        ret += strings[i];
    }

    return ret;
}

TEST(ProgrammingConventionSyntaxQ, RangeFor)
{
    ASSERT_EQ("abcd", Accumulate(std::vector<std::string>{"a", "b", "cd"}));
}
```

- 参照 [範囲for文](#)
- 解答例-[コンテナの範囲for文](#)

演習-ラムダ式

- 問題

```
// @@@ exercise/programming_convention_q/syntax.cpp 32
// [Q]
// 下記のcopy_ifの第4引数をラムダ式を使って書き直せ。
bool is_not_size0(std::string const& s) noexcept { return s.size() != 0; }

TEST(ProgrammingConventionSyntaxQ, Lambda)
{
    auto data = std::vector<std::string>{"", "abc", "", "d"};

    auto ret = std::vector<std::string>{};

    std::copy_if(data.cbegin(), data.cend(), std::back_inserter(ret), is_not_size0);
    ASSERT_EQ((std::vector<std::string>{"abc", "d"}), ret);
}
```

- 参照 [ラムダ式](#)
- 解答例-[ラムダ式](#)

演習-ラムダ式のキャプチャ

- 問題

```
// @@@ exercise/programming_convention_q/syntax.cpp 48
// [Q]
// 下記Lambda::GetNameLessThan()のラムダ式の問題点を修正し、単体テストを行え。
class Lambda {
public:
    explicit Lambda(std::vector<std::string>&& strs) : strs_{std::move(strs)} {}
    std::vector<std::string> GetNameLessThan(uint32_t length) const
    {
        auto ret = std::vector<std::string>{};

        std::copy_if(strs_.cbegin(), strs_.cend(), std::back_inserter(ret),
                    [=](auto const& str) noexcept { return (strs_.size() < length); });

        return ret;
    }
}
```

```

private:
    std::vector<std::string> strs_;
};

TEST(ProgrammingConventionSyntaxQ, Lambda2)
{
    auto lambda = Lambda{{"abc", "abcdef", "a"}};

    // 以下に単体テストを追加。
}

```

- 参照 [ラムダ式](#)
- [解答例-ラムダ式のキャプチャ](#)

プログラミング規約(演算子)

演習-三項演算子

- 問題

```

// @@@ exercise/programming_convention_q/operator.cpp 6
// [Q]
// 下記whichのif文を三項演算子を使用して書き直せ。
int32_t which(bool left, int32_t lhs, int32_t rhs) noexcept
{
    if (left) {
        return lhs;
    }
    else {
        return rhs;
    }
}

TEST(ProgrammingConventionOperatorQ, OoOperator)
{
    ASSERT_EQ(3, which(true, 3, 4));
    ASSERT_EQ(4, which(false, 3, 4));
}

```

- 参照 [三項演算子](#)
- [解答例-三項演算子](#)

演習-delete

- 問題

```

// @@@ exercise/programming_convention_q/operator.cpp 26
// [Q]
// 下記DeleteProblemのメモリ管理の問題を修正し、単体テストを行え(DISABLED_削除)。
// また、他の問題があれば、合わせて修正せよ。
class DeleteProblem {
public:
    DeleteProblem(char const* str0 = nullptr, char const* str1 = nullptr)
    {
        if (str0 != nullptr) {
            str0_ = new std::string{str0};
        }
        if (str1 != nullptr) {
            str1_ = new std::string{str1};
        }
    }

    std::string const* GetStr0() { return str0_; }
    std::string const* GetStr1() { return str1_; }

    ~DeleteProblem()
    {
        delete_str(str0_);
        delete_str(str1_);
    }
};

private:
    static void delete_str(void* str)
    {
        if (str != nullptr) {

```

```

        delete str;
    }

    std::string* str0_ = nullptr;
    std::string* str1_ = nullptr;
};

TEST(DISABLED_ProgrammingConventionOperatorQ, Delete)
{
    // この単体テストはメモリリークを起こす
    // このメモリリークはmake san-utで検出される
    {
        auto dp = DeleteProblem{};

        ASSERT_EQ(nullptr, dp.GetStr0());
        ASSERT_EQ(nullptr, dp.GetStr1());
    }
    {
        auto dp = DeleteProblem{"abc"};

        ASSERT_EQ("abc", *dp.GetStr0());
        ASSERT_EQ(nullptr, dp.GetStr1());
    }
    {
        auto dp = DeleteProblem{"abc", "de"};

        ASSERT_EQ("abc", *dp.GetStr0());
        ASSERT_EQ("de", *dp.GetStr1());
    }
}

```

- 参照 [delete](#)
- [解答例-delete](#)

演習-sizeof

- 問題

```

// @@@ exercise/programming_convention_q/operator.cpp 88
// [Q]
// 下記Size1() - Size4()の単体テストを作れ。
size_t Size0(int32_t a) noexcept { return sizeof(a); }

size_t Size1(int32_t a[10]) noexcept { return sizeof(a); }

size_t Size2(int32_t a[]) noexcept { return sizeof(a); }

size_t Size3(int32_t* a) noexcept { return sizeof(a); }

size_t Size4(int32_t (&a)[10]) noexcept { return sizeof(a); }

TEST(ProgrammingConventionOperatorQ, Sizeof)
{
    int32_t array[10]{};

    // Size1() - Size4()の単体テスト
    ASSERT_EQ(sizeof(void*), Size3(array));
}

```

- 参照 [sizeof](#)
- [解答例-sizeof](#)

演習-dynamic_castの削除

- 問題

```

// @@@ exercise/programming_convention_q/operator.cpp 110
// [Q]
// 下記クラスX、Y、ZとGetNameをdynamic_castを使わずに書き直せ。

class X {
public:
    virtual ~X() = default;
};

class Y : public X {};

```

```

class Z : public X {};

std::string GetName(X* x)
{
    if (dynamic_cast<Y*>(x) != nullptr) {
        return "Y";
    }
    if (dynamic_cast<Z*>(x) != nullptr) {
        return "Z";
    }
    if (dynamic_cast<X*>(x) != nullptr) {
        return "X";
    }

    assert(false);
    return "UnKnown";
}

TEST(ProgrammingConventionOperatorQ, Cast)
{
    auto x = X{};
    auto y = Y{};
    auto z = Z{};

    ASSERT_EQ("X", GetName(&x));
    ASSERT_EQ("Y", GetName(&y));
    ASSERT_EQ("Z", GetName(&z));
}

```

- 参照 キャスト、暗黙の型変換
- 解答例-dynamic_castの削除

演習-キャスト

- 問題
キャストは避けるべきだが、やむを得ず使用する場合であっても、Cタイプキャスト、(A)、dynamic_castは使用しない。
(A)に相応しいものを下記より選べ。
- 選択肢
 1. static_cast
 2. reinterpret_cast
 3. const_cast
 4. 該当なし
- 解答-キャスト

プログラミング規約(スコープ)

演習-usingディレクティブ

- 問題
usingディレクティブ(using namespace NS)の使用上の注意として相応しいものを選べ。
- 選択肢
 1. usingディレクティブは、関数先頭でのみ使用してよい。
 2. ヘッダファイルでusingディレクティブをすると便利である。
 3. ファイルスコープでusingディレクティブをすると便利である。
 4. using宣言よりusingディレクティブを優先するべきである。
- 解答-usingディレクティブ

プログラミング規約(その他)

演習-アサーションの選択

- 問題
論理的にありえない(switchでそのcaseやdefaultを通過することはあり得ない等) 状態を検知するために積極的に(A)を使用する。
(A)に相応しいものを下記より選べ。
- 選択肢
 - assert()
 - static assert
 - while(1);
 - abort()

- 解答-アサーションの選択

演習-assert/static_assert

- 問題

```
// @@@ exercise/programming_convention_q/etc.cpp 4
// [Q]
// 下記FloatingPointは、Tが浮動小数点型、Tのインスタンスは非0であることを前提としている。
// 適切にアサーションを挿入して誤用を防げ。

template <typename T>
class FloatingPoint {
public:
    FloatingPoint(T num) noexcept : num_(num) {}
    T Get() const noexcept { return num_; }
    T Reciprocal() const noexcept { return 1 / num_; }

private:
    T num_;
};
```

- 参照 [assertion](#)
- 解答例-assert/static_assert

SOLID

以下の演習問題の単体テストで使用される

```
TEST_F(Xxx, Yyy)
```

のような記述のXxyは、以下のように宣言・定義されている。

```
// @@@ exercise/h/solid_ut.h 7

class SolidFixture : public ::testing::Test {
protected:
    std::string const test_score_org_ = "../ut_data2/test_score_org.csv";
    std::string const test_score_org_f_ = "../ut_data2/test_score_org_f.csv";
    std::string const test_score_act_ = "../ut_data2/test_score_act.csv";
    std::string const test_score_exp_ = "../ut_data2/test_score_exp.csv";
    std::string const test_score_exp_err_ = "../ut_data2/test_score_exp_err.csv";

    virtual void SetUp() noexcept override { remove_file(test_score_act_); }
    virtual void TearDown() noexcept override { remove_file(test_score_act_); }

    static void remove_file(std::string const& filename) noexcept
    {
        if (std::filesystem::exists(filename)) {
            std::filesystem::remove(filename);
        }
    }
};

class SolidSRP_Q : public SolidFixture {};
class SolidOCP_Q : public SolidFixture {};
class SolidLSP_Q : public SolidFixture {};
class SolidISP_Q : public SolidFixture {};
```

```

class SolidDIP_Q : public SolidFixture {};

class SolidSRP_A : public SolidFixture {};
class SolidOCP_A : public SolidFixture {};
class SolidLSP_A : public SolidFixture {};
class SolidISP_A : public SolidFixture {};
class SolidDIP_A : public SolidFixture {};

```

演習-SRP

- 問題

```

// @@@ exercise/solid_q/srp_test_score.h 8
// [Q]
// 下記クラスTestScoreはメンバにする必要のない関数までメンバにしてるため、
// インターフェースが肥大化してしまい、少なくともSRPに反している。
// メンバにする必要のないStoreCSVを外部関数にせよ。
// また、受験者の平均点を求める
//     TestScore::ScoreOne_t Average(TestScore const& test_score);
// を同様の方法で作り、単体テストを行え。

class TestScore {
public:
    TestScore() = default;
    TestScore(TestScore const&) = default;
    TestScore& operator=(TestScore const&) = delete;

    using ScoreAll_t = std::map<std::string, std::vector<int32_t>>;
    using ScoreOne_t = std::vector<std::pair<std::string, int32_t>>;

    void AddScore(ScoreOne_t const& one_test_score);
    std::vector<int32_t> const& GetScore(std::string const& name) const
    {
        return test_score_row_.at(name);
    }

    void StoreCSV(std::string const& filename) const;
    void LoadCSV(std::string const& filename);

    ScoreAll_t::const_iterator begin() const noexcept { return test_score_row_.begin(); }
    ScoreAll_t::const_iterator end() const noexcept { return test_score_row_.end(); }

private:
    // int32_t score: 0~100はスコア、-1は未受験、それ以外は不正データ
    void validate_score(int32_t score) const;
    ScoreAll_t test_score_row_{};
};

std::string ToString(TestScore const& ts);

// @@@ exercise/solid_q/srp_test_score.cpp 10

namespace {
std::pair<std::string, std::vector<int32_t>> parse_line(std::string const& line)
{
    auto const csv_sep = std::regex{R"(\s+)"};
    auto name = std::string{};
    auto score = std::vector<int32_t>{};

    auto end = std::sregex_token_iterator{};
    for (auto it = std::sregex_token_iterator{line.begin(), line.end(), csv_sep, -1}; it != end;
         ++it) {
        if (name.length() == 0) {
            name = *it;
        }
        else {
            auto s = std::stoi(*it);
            score.emplace_back(s);
        }
    }

    return {std::move(name), std::move(score)};
}
} // namespace

void TestScore::validate_score(int32_t score) const
{
    auto highest = 100;
    auto lowest = 0;
}

```

```

auto invalid = -1;

if (lowest <= score && score <= highest) {
    ; // do nothing
}
else if (invalid == score) {
    ; // do nothing
}
else {
    throw std::out_of_range("Invalid Score");
}

void TestScore::AddScore(TestScore::ScoreOne_t const& one_test_score)
{
    for (auto const& pair : one_test_score) {
        validate_score(pair.second);
    }

    if (test_score_row_.size() == 0) {
        test_score_row_[one_test_score[0].first] = std::vector<int32_t>{};
    }

    for (auto& pair : test_score_row_) {
        pair.second.push_back(-1);
    }

    auto curr_test_count = test_score_row_.begin()[-second.size()];

    for (auto const& pair : one_test_score) {
        if (test_score_row_.find(pair.first) == test_score_row_.end()) {
            test_score_row_[pair.first] = std::vector<int32_t>(curr_test_count, -1);
        }

        test_score_row_[pair.first].back() = pair.second;
    }
}

void TestScore::LoadCSV(std::string const& filename)
{
    auto data = std::ifstream{filename};

    auto test_score_raw = ScoreAll_t();
    auto line          = std::string{};
    while (std::getline(data, line)) {
        // std::pair<TestScore::ScoreAll_t::iterator, bool>
        auto ret = test_score_raw.insert(parse_line(line));
        assert(ret.second);
    }

    for (auto const& pair : test_score_raw) {
        for (auto const s : pair.second) {
            validate_score(s);
        }
    }

    test_score_row_.swap(test_score_raw);
}

void TestScore::StoreCSV(std::string const& filename) const
{
    auto data = std::ofstream{filename};
    auto ss   = std::ostringstream{};

    for (auto const& pair : test_score_row_) {
        ss << pair.first;
        for (auto const s : pair.second) {
            ss << ", " << s;
        }
        ss << std::endl;
    }

    data << ss.str();
}

std::string ToString(TestScore const& ts)
{
    auto ss = std::ostringstream{};

    for (auto const& pair : ts) {

```

```

        ss << pair.first << ':';
        for (auto const& s : pair.second) {
            ss << ' ' << s;
        }
        ss << std::endl;
    }

    return ss.str();
}

// @@@ exercise/solid_q/srp_test_score_ut.cpp 13

namespace {

TEST_F(SolidSRP_Q, TestScore_LoadCSV)
{
    auto ts = TestScore{};
    ts.LoadCSV(test_score_org_);

    auto const exp = TestScore::ScoreAll_t{
        TestScore::ScoreAll_t::value_type("堂林", {-1, 50, 80}),
        TestScore::ScoreAll_t::value_type("広輔", {40, 30, 10}),
        TestScore::ScoreAll_t::value_type("會澤", {30, 60, 70}),
        TestScore::ScoreAll_t::value_type("松山", {80, 90, 50}),
        TestScore::ScoreAll_t::value_type("菊池", {50, 20, 80}),
        TestScore::ScoreAll_t::value_type("鈴木", {0, 80, 100}),
    };

    ASSERT_TRUE(std::equal(ts.begin(), ts.end(), exp.begin()));

    auto const exp_str = std::string{
        "堂林: -1 50 80\n広輔: 40 30 10\n會澤: 30 60 70\n"
        "松山: 80 90 50\n菊池: 50 20 80\n鈴木: 0 80 100";
    }
    ASSERT_EQ(ToString(ts), exp_str);
}

std::string whole_file(std::string const& filename)
{
    auto ifs = std::ifstream{filename};

    return std::string{std::istreambuf_iterator<char>{ifs}, std::istreambuf_iterator<char>{}};
}

TEST_F(SolidSRP_Q, TestScore_AddScore)
{
    auto ts = TestScore{};
    ts.LoadCSV(test_score_org_);

    auto const one_score = TestScore::ScoreOne_t{
        TestScore::ScoreOne_t::value_type("堂林", 50),
        TestScore::ScoreOne_t::value_type("広輔", 40),
        TestScore::ScoreOne_t::value_type("會澤", 70),
        TestScore::ScoreOne_t::value_type("松山", 1),
        TestScore::ScoreOne_t::value_type("菊池", -1),
        TestScore::ScoreOne_t::value_type("鈴木", 5),
        TestScore::ScoreOne_t::value_type("田中", 100),
        TestScore::ScoreOne_t::value_type("西川", 90),
    };

    ts.AddScore(one_score);

    auto const exp = TestScore::ScoreAll_t{
        TestScore::ScoreAll_t::value_type("堂林", {-1, 50, 80, 50}),
        TestScore::ScoreAll_t::value_type("広輔", {40, 30, 10, 40}),
        TestScore::ScoreAll_t::value_type("會澤", {30, 60, 70, 70}),
        TestScore::ScoreAll_t::value_type("松山", {80, 90, 50, 1}),
        TestScore::ScoreAll_t::value_type("菊池", {50, 20, 80, -1}),
        TestScore::ScoreAll_t::value_type("鈴木", {0, 80, 100, 5}),
        TestScore::ScoreAll_t::value_type("田中", {-1, -1, -1, 100}),
        TestScore::ScoreAll_t::value_type("西川", {-1, -1, -1, 90}),
    };

    ASSERT_TRUE(std::equal(ts.begin(), ts.end(), exp.begin()));

    auto const one_score_err = TestScore::ScoreOne_t{
        TestScore::ScoreOne_t::value_type("野村", -2),
        TestScore::ScoreOne_t::value_type("衣笠", 40),
    };
}

// 不正データロード

```

```

auto ts2 = ts;
ASSERT_THROW(ts.AddScore(one_score_err), std::out_of_range);

// エクセプション 強い保証
ASSERT_TRUE(std::equal(ts.begin(), ts.end(), ts2.begin()));
}

TEST_F(SolidSRP_Q, TestScore_GetScore)
{
    auto ts = TestScore{};
    ts.LoadCSV(test_score_org_);

    auto const& score_0 = ts.GetScore("堂林");
    ASSERT_EQ({-1, 50, 80}, score_0);

    auto const& score_1 = ts.GetScore("広輔");
    ASSERT_EQ({40, 30, 10}, score_1);

    auto const& score_2 = ts.GetScore("會澤");
    ASSERT_EQ({30, 60, 70}, score_2);

    auto const& score_3 = ts.GetScore("松山");
    ASSERT_EQ({80, 90, 50}, score_3);

    auto const& score_4 = ts.GetScore("菊池");
    ASSERT_EQ({50, 20, 80}, score_4);

    auto const& score_5 = ts.GetScore("鈴木");
    ASSERT_EQ({0, 80, 100}, score_5);

    ASSERT_THROW(ts.GetScore("西川"), std::out_of_range);
}

TEST_F(SolidSRP_Q, TestScore_StoreCSV)
{
    auto ts = TestScore{};
    ts.LoadCSV(test_score_org_);
    ts.StoreCSV(test_score_act_);

    auto content_act = whole_file(test_score_act_);
    auto content_exp = whole_file(test_score_exp_);

    ASSERT_EQ(content_exp, content_act);

    // 不正ファイルロード
    auto ts2 = ts;
    ASSERT_THROW(ts.LoadCSV(test_score_exp_err_), std::out_of_range);

    // エクセプション 強い保証
    ASSERT_TRUE(std::equal(ts.begin(), ts.end(), ts2.begin()));
}
} // namespace

```

- 参照 [単一責任の原則\(SRP\)](#)
- [解答例-SRP](#)

演習-OCP

- 問題

```

// @@@ exercise/solid_q/ocp_test_score.h 8
// [Q]
// 下記クラスTestScoreは、
//   * テスト受講者とその点数を保持/提供する。
//   * テスト受講者とその点数をCSVファイルからロードする。
// 責任を持つ。サポートするファイル形式が増えた場合、このクラスを修正せざるを得ないため、
// 機能拡張に対して開いていない。つまり、OCPに反していると言える
// (実際にはこの程度の違反が問題になることは稀である)。
//
// サポートしているファイル形式はCSVのみであったが、TSVを追加することになった。
// 今後もサポートするファイル形式を増やす必要があるため、OCPに従った方が良いと判断し、
// TestScoreの責務から「ファイルのロード」を外し、その機能を外部関数として定義することにした。
// これに従い、下記クラスTestScoreを修正し、外部関数
//   void LoadCSV(std::string const& filename, TestScore& test_score);
// を作り、単体テストを行え。
//
class TestScore {
public:
    TestScore() = default;

```

```

TestScore(TestScore const&) = default;
TestScore& operator=(TestScore const&) = delete;

using ScoreAll_t = std::map<std::string, std::vector<int32_t>>;
using ScoreOne_t = std::vector<std::pair<std::string, int32_t>>;

void AddScore(ScoreOne_t const& one_test_score);
std::vector<int32_t> const& GetScore(std::string const& name) const
{
    return test_score_row_.at(name);
}

void LoadCSV(std::string const& filename);

ScoreAll_t::const_iterator begin() const noexcept { return test_score_row_.begin(); }
ScoreAll_t::const_iterator end() const noexcept { return test_score_row_.end(); }

private:
    // int32_t score: 0~100はスコア、-1は未受験、それ以外は不正データ
    void validate_score(int32_t score) const;
    ScoreAll_t test_score_row_{};
};

std::string ToString(TestScore const& ts);
void StoreCSV(TestScore const& test_score, std::string const& filename);
TestScore::ScoreOne_t Average(TestScore const& test_score);

```

// @@@ exercise/solid_q/ocp_test_score.cpp 10

```

namespace {
std::pair<std::string, std::vector<int32_t>> parse_line(std::string const& line)
{
    auto const csv_sep = std::regex{R"( *, *)"};
    auto name = std::string{};
    auto score = std::vector<int32_t>{};

    auto end = std::sregex_token_iterator{};
    for (auto it = std::sregex_token_iterator{line.begin(), line.end(), csv_sep, -1}; it != end;
        ++it) {
        if (name.length() == 0) {
            name = *it;
        }
        else {
            auto s = std::stoi(*it);
            score.emplace_back(s);
        }
    }
    return {std::move(name), std::move(score)};
}

bool is_valid_score(int32_t score) noexcept { return 0 <= score && score <= 100; }

bool not_score(int32_t score) noexcept { return score == -1; }
} // namespace

void TestScore::validate_score(int32_t score) const
{
    if (is_valid_score(score) || not_score(score)) {
        return;
    }

    throw std::out_of_range{"Invalid Score"};
}

void TestScore::AddScore(TestScore::ScoreOne_t const& one_test_score)
{
    for (auto const& pair : one_test_score) {
        validate_score(pair.second);
    }

    if (test_score_row_.size() == 0) {
        test_score_row_[one_test_score[0].first] = std::vector<int32_t>{};
    }

    for (auto& pair : test_score_row_) {
        pair.second.push_back(-1);
    }

    auto curr_test_count = test_score_row_.begin()->second.size();

```

```

        for (auto const& pair : one_test_score) {
            if (test_score_row_.find(pair.first) == test_score_row_.end()) {
                test_score_row_[pair.first] = std::vector<int32_t>(curr_test_count, -1);
            }

            test_score_row_[pair.first].back() = pair.second;
        }
    }

void TestScore::LoadCSV(std::string const& filename)
{
    auto data = std::ifstream{filename};

    auto test_score_raw = TestScore::ScoreAll_t{};
    auto line          = std::string{};
    while (std::getline(data, line)) {
        // std::pair<TestScore::ScoreAll_t::iterator, bool>
        auto ret = test_score_raw.insert(parse_line(line));
        assert(ret.second);
    }

    for (auto const& pair : test_score_raw) {
        for (auto const s : pair.second) {
            validate_score(s);
        }
    }

    test_score_row_.swap(test_score_raw);
}

void StoreCSV(TestScore const& test_score, std::string const& filename)
{
    auto data = std::ofstream{filename};
    auto ss   = std::ostringstream{};

    for (auto const& pair : test_score) {
        ss << pair.first;
        for (auto const s : pair.second) {
            ss << ", " << s;
        }
        ss << std::endl;
    }

    data << ss.str();
}

std::string ToString(TestScore const& ts)
{
    auto ss = std::ostringstream{};

    for (auto const& pair : ts) {
        ss << pair.first << ':';
        for (auto const s : pair.second) {
            ss << ' ' << s;
        }
        ss << std::endl;
    }

    return ss.str();
}

TestScore::ScoreOne_t Average(TestScore const& test_score)
{
    auto ret = TestScore::ScoreOne_t{};

    for (auto const& pair : test_score) {
        auto sum      = 0;
        auto valid_count = 0U;
        for (auto const s : pair.second) {
            if (is_valid_score(s)) {
                sum += s;
                ++valid_count;
            }
        }
        ret.emplace_back(std::make_pair(pair.first, valid_count == 0 ? -1 : sum / valid_count));
    }

    return ret;
}

```

```

// @@@ exercise/solid_q/ocp_test_score_ut.cpp 13

namespace {

TEST_F(SolidOCP_Q, TestScore_LoadCSV)
{
    auto ts = TestScore{};
    ts.LoadCSV(test_score_org_);

    auto const exp = TestScore::ScoreAll_t{
        TestScore::ScoreAll_t::value_type("堂林", {-1, 50, 80}),
        TestScore::ScoreAll_t::value_type("広輔", {40, 30, 10}),
        TestScore::ScoreAll_t::value_type("會澤", {30, 60, 70}),
        TestScore::ScoreAll_t::value_type("松山", {80, 90, 50}),
        TestScore::ScoreAll_t::value_type("菊池", {50, 20, 80}),
        TestScore::ScoreAll_t::value_type("鈴木", {0, 80, 100}),
    };

    ASSERT_TRUE(std::equal(ts.begin(), ts.end(), exp.begin()));

    auto const exp_str = std::string{
        "堂林: -1 50 80\n広輔: 40 30 10\n會澤: 30 60 70\n"
        "松山: 80 90 50\n菊池: 50 20 80\n鈴木: 0 80 100\n";
    }
    ASSERT_EQ(ToString(ts), exp_str);
}

std::string whole_file(std::string const& filename)
{
    auto ifs = std::ifstream{filename};

    return std::string{std::istreambuf_iterator<char>{ifs}, std::istreambuf_iterator<char>{}};
}

TEST_F(SolidOCP_Q, TestScore_AddScore)
{
    auto ts = TestScore{};
    ts.LoadCSV(test_score_org_);

    auto const one_score = TestScore::ScoreOne_t{
        TestScore::ScoreOne_t::value_type("堂林", 50),
        TestScore::ScoreOne_t::value_type("広輔", 40),
        TestScore::ScoreOne_t::value_type("會澤", 70),
        TestScore::ScoreOne_t::value_type("松山", 1),
        TestScore::ScoreOne_t::value_type("菊池", -1),
        TestScore::ScoreOne_t::value_type("鈴木", 5),
        TestScore::ScoreOne_t::value_type("田中", 100),
        TestScore::ScoreOne_t::value_type("西川", 90),
    };

    ts.AddScore(one_score);

    auto const exp = TestScore::ScoreAll_t{
        TestScore::ScoreAll_t::value_type("堂林", {-1, 50, 80, 50}),
        TestScore::ScoreAll_t::value_type("広輔", {40, 30, 10, 40}),
        TestScore::ScoreAll_t::value_type("會澤", {30, 60, 70, 70}),
        TestScore::ScoreAll_t::value_type("松山", {80, 90, 50, 1}),
        TestScore::ScoreAll_t::value_type("菊池", {50, 20, 80, -1}),
        TestScore::ScoreAll_t::value_type("鈴木", {0, 80, 100, 5}),
        TestScore::ScoreAll_t::value_type("田中", {-1, -1, -1, 100}),
        TestScore::ScoreAll_t::value_type("西川", {-1, -1, -1, 90}),
    };

    ASSERT_TRUE(std::equal(ts.begin(), ts.end(), exp.begin()));

    auto const one_score_err = TestScore::ScoreOne_t{
        TestScore::ScoreOne_t::value_type("野村", -2),
        TestScore::ScoreOne_t::value_type("衣笠", 40),
    };

    // 不正データロード
    auto ts2 = ts;
    ASSERT_THROW(ts.AddScore(one_score_err), std::out_of_range);

    // エクセプション 強い保証
    ASSERT_TRUE(std::equal(ts.begin(), ts.end(), ts2.begin()));
}

TEST_F(SolidOCP_Q, TestScore_GetScore)
{
    auto ts = TestScore{};

```

```

    ts.LoadCSV(test_score_org_);

    auto const& score_0 = ts.GetScore("堂林");
    ASSERT_EQ((std::vector{-1, 50, 80}), score_0);

    auto const& score_1 = ts.GetScore("広輔");
    ASSERT_EQ((std::vector{40, 30, 10}), score_1);

    auto const& score_2 = ts.GetScore("會澤");
    ASSERT_EQ((std::vector{30, 60, 70}), score_2);

    auto const& score_3 = ts.GetScore("松山");
    ASSERT_EQ((std::vector{80, 90, 50}), score_3);

    auto const& score_4 = ts.GetScore("菊池");
    ASSERT_EQ((std::vector{50, 20, 80}), score_4);

    auto const& score_5 = ts.GetScore("鈴木");
    ASSERT_EQ((std::vector{0, 80, 100}), score_5);

    ASSERT_THROW(ts.GetScore("西川"), std::out_of_range);
}

TEST_F(SolidOCP_Q, TestScore_StoreCSV)
{
    auto ts = TestScore{};
    ts.LoadCSV(test_score_org_);
    StoreCSV(ts, test_score_act_);

    auto content_act = whole_file(test_score_act_);
    auto content_exp = whole_file(test_score_exp_);

    ASSERT_EQ(content_exp, content_act);

    // 不正ファイルロード
    auto ts2 = ts;
    ASSERT_THROW(ts.LoadCSV(test_score_exp_err_), std::out_of_range);

    // エクセプション 強い保証
    ASSERT_TRUE(std::equal(ts.begin(), ts.end(), ts2.begin()));
}

TEST_F(SolidOCP_Q, TestScore_Average)
{
    auto ts = TestScore{};
    ts.LoadCSV(test_score_org_);

    auto const exp = TestScore::ScoreOne_t{
        TestScore::ScoreOne_t::value_type("堂林", 65),
        TestScore::ScoreOne_t::value_type("広輔", 26),
        TestScore::ScoreOne_t::value_type("會澤", 53),
        TestScore::ScoreOne_t::value_type("松山", 73),
        TestScore::ScoreOne_t::value_type("菊池", 50),
        TestScore::ScoreOne_t::value_type("鈴木", 60),
    };
    auto act = Average(ts);

    ASSERT_EQ(act, exp);
}
} // namespace

```

- 参照 オープン・クローズドの原則(OCP)
- 解答例-OCP

演習-LSP

- 問題

```

// @@@ exercise/solid_q/lsp_test_score.h 8
// [Q]
// 下記クラスTestScoreが管理するテストのスコアの値は、
//   * 0~100 テストのスコア
//   * -1 未受験
//   * それ以外 不正値であるため、このデータを入力すると
//         std::out_of_rangeエクセプションが発生する。
// を表すが、未受講を許可しない仕様(受験できない場合のスコアは0点)の
// TestScoreForceも必要になったため下記のように定義した。
//   * TestScoreForceが管理するテストのスコアの値は
//         * 0~100 テストのスコア

```

```

//      * それ以外 不正値であるため、このデータを入力すると
//          std::out_of_rangeエクセプションが発生する。
//      * それ以外の動作はTestScoreと同じ。
// これは、事前条件(「-1~100を受け入れる」から「0~100を受け入れる」)の強化であるため、
// LSPに反する。
// これにより起こる問題点を単体テストを用いて指摘せよ。
//
// [Q]
// 上記問題を解決するため、クラスTestScoreForceFixedを作り単体テストを行え。

class TestScore {
public:
    TestScore() = default;
    virtual ~TestScore() = default;
    TestScore(TestScore const&) = default;
    TestScore& operator=(TestScore const&) = delete;
    TestScore& operator=(TestScore&&) = default;

    using ScoreAll_t = std::map<std::string, std::vector<int32_t>>;
    using ScoreOne_t = std::vector<std::pair<std::string, int32_t>>;

    void AddScore(ScoreOne_t const& one_test_score);
    std::vector<int32_t> GetScore(std::string const& name) const
    {
        return test_score_row_.at(name);
    }

    ScoreAll_t::const_iterator begin() const noexcept { return test_score_row_.begin(); }
    ScoreAll_t::const_iterator end() const noexcept { return test_score_row_.end(); }

private:
    // int32_t score: 0~100はスコア、-1は未受験、それ以外は不正データ
    virtual void validate_score(int32_t score) const;
    ScoreAll_t test_score_row_{};
};

class TestScoreForce : public TestScore {
private:
    // int32_t score: 0~100はスコア、それ以外は不正データ
    virtual void validate_score(int32_t score) const override;
};

std::string ToString(TestScore const& ts);
void LoadCSV(std::string const& filename, TestScore& test_score);
void StoreCSV(TestScore const& test_score, std::string const& filename);
TestScore::ScoreOne_t Average(TestScore const& test_score);

```

```

// @@@ exercise/solid_q/lsp_test_score.cpp 10

namespace {
std::pair<std::string, std::vector<int32_t>> parse_line(std::string const& line)
{
    auto const csv_sep = std::regex{R"( *, *)"};
    auto name = std::string{};
    auto score = std::vector<int32_t>{};

    auto end = std::sregex_token_iterator{};
    for (auto it = std::sregex_token_iterator{line.begin(), line.end(), csv_sep, -1}; it != end;
         ++it) {
        if (name.length() == 0) {
            name = *it;
        } else {
            auto s = std::stoi(*it);
            score.emplace_back(s);
        }
    }

    return {std::move(name), std::move(score)};
}

bool is_valid_score(int32_t score) noexcept { return 0 <= score && score <= 100; }

bool not_score(int32_t score) noexcept { return score == -1; }
} // namespace

void TestScore::validate_score(int32_t score) const
{
    if (is_valid_score(score) || not_score(score)) {
        return;
    }
}

```

```

    }

    throw std::out_of_range{"Invalid Score"};
}

void TestScoreForce::validate_score(int32_t score) const
{
    if (is_valid_score(score)) {
        return;
    }

    throw std::out_of_range{"Invalid Score"};
}

void TestScore::AddScore(TestScore::ScoreOne_t const& one_test_score)
{
    for (auto const& pair : one_test_score) {
        validate_score(pair.second);
    }

    if (test_score_row_.size() == 0) {
        test_score_row_[one_test_score[0].first] = std::vector<int32_t>{};
    }

    for (auto& pair : test_score_row_) {
        pair.second.push_back(-1);
    }

    auto curr_test_count = test_score_row_.begin()->second.size();

    for (auto const& pair : one_test_score) {
        if (test_score_row_.find(pair.first) == test_score_row_.end()) {
            test_score_row_[pair.first] = std::vector<int32_t>(curr_test_count, -1);
        }

        test_score_row_[pair.first].back() = pair.second;
    }
}

void LoadCSV(std::string const& filename, TestScore& test_score)
{
    auto data = std::ifstream{filename};

    auto test_score_raw = TestScore::ScoreAll_t{};
    auto line          = std::string{};
    while (std::getline(data, line)) {
        // std::pair<TestScore::ScoreAll_t::iterator, bool>
        auto ret = test_score_raw.insert(parse_line(line));
        assert(ret.second);
    }

    auto one_test = TestScore::ScoreOne_t{};
    for (auto const& pair : test_score_raw) {
        one_test.emplace_back(std::make_pair(pair.first, 0));
    }

    auto const score_count = test_score_raw.begin()->second.size();
    auto      ts          = TestScore{};

    for (auto i = 0U; i < score_count; ++i) {
        for (auto& pair : one_test) {
            pair.second = test_score_raw[pair.first][i];
        }
        ts.AddScore(one_test);
    }

    test_score = std::move(ts);
}

void StoreCSV(TestScore const& test_score, std::string const& filename)
{
    auto data = std::ofstream{filename};
    auto ss   = std::ostringstream{};

    for (auto const& pair : test_score) {
        ss << pair.first;
        for (auto const s : pair.second) {
            ss << ", " << s;
        }
        ss << std::endl;
    }
}

```

```

    }

    data << ss.str();
}

std::string ToString(TestScore const& ts)
{
    auto ss = std::ostringstream{};

    for (auto const& pair : ts) {
        ss << pair.first << ':';
        for (auto const s : pair.second) {
            ss << ' ' << s;
        }
        ss << std::endl;
    }

    return ss.str();
}

TestScore::ScoreOne_t Average(TestScore const& test_score)
{
    auto ret = TestScore::ScoreOne_t{};

    for (auto const& pair : test_score) {
        auto sum         = 0;
        auto valid_count = 0U;
        for (auto const s : pair.second) {
            if (is_valid_score(s)) {
                sum += s;
                ++valid_count;
            }
        }
        ret.emplace_back(std::make_pair(pair.first, valid_count == 0 ? -1 : sum / valid_count));
    }

    return ret;
}

```

```

// @@@ exercise/solid_q/lsp_test_score_ut.cpp 13

namespace {

TEST_F(SolidLSP_Q, TestScore_LoadCSV)
{
    auto ts = TestScore{};
    LoadCSV(test_score_org_, ts);

    auto const exp = TestScore::ScoreAll_t{
        TestScore::ScoreAll_t::value_type("堂林", {-1, 50, 80}),
        TestScore::ScoreAll_t::value_type("広輔", {40, 30, 10}),
        TestScore::ScoreAll_t::value_type("會澤", {30, 60, 70}),
        TestScore::ScoreAll_t::value_type("松山", {80, 90, 50}),
        TestScore::ScoreAll_t::value_type("菊池", {50, 20, 80}),
        TestScore::ScoreAll_t::value_type("鈴木", {0, 80, 100}),
    };

    ASSERT_TRUE(std::equal(ts.begin(), ts.end(), exp.begin()));

    auto exp_str = std::string{
        "堂林: -1 50 80\n広輔: 40 30 10\n會澤: 30 60 70\n"
        "松山: 80 90 50\n菊池: 50 20 80\n鈴木: 0 80 100\n";
    };
    ASSERT_EQ(ToString(ts), exp_str);
}

std::string whole_file(std::string const& filename)
{
    auto ifs = std::ifstream{filename};

    return std::string{std::istreambuf_iterator<char>{ifs}, std::istreambuf_iterator<char>{}};
}

TEST_F(SolidLSP_Q, TestScore_AddScore)
{
    auto ts = TestScore{};
    LoadCSV(test_score_org_, ts);

    auto const one_score = TestScore::ScoreOne_t{
        TestScore::ScoreOne_t::value_type("堂林", 50),
        TestScore::ScoreOne_t::value_type("広輔", 40),

```

```

    TestScore::ScoreOne_t::value_type("會澤", 70),
    TestScore::ScoreOne_t::value_type("松山", 1),
    TestScore::ScoreOne_t::value_type("菊池", -1),
    TestScore::ScoreOne_t::value_type("鈴木", 5),
    TestScore::ScoreOne_t::value_type("田中", 100),
    TestScore::ScoreOne_t::value_type("西川", 90),
};

ts.AddScore(one_score);

auto const exp = TestScore::ScoreAll_t{
    TestScore::ScoreAll_t::value_type("堂林", {-1, 50, 80, 50}),
    TestScore::ScoreAll_t::value_type("広輔", {40, 30, 10, 40}),
    TestScore::ScoreAll_t::value_type("會澤", {30, 60, 70, 70}),
    TestScore::ScoreAll_t::value_type("松山", {80, 90, 50, 1}),
    TestScore::ScoreAll_t::value_type("菊池", {50, 20, 80, -1}),
    TestScore::ScoreAll_t::value_type("鈴木", {0, 80, 100, 5}),
    TestScore::ScoreAll_t::value_type("田中", {-1, -1, -1, 100}),
    TestScore::ScoreAll_t::value_type("西川", {-1, -1, -1, 90}),
};

ASSERT_TRUE(std::equal(ts.begin(), ts.end(), exp.begin()));

auto const one_score_err = TestScore::ScoreOne_t{
    TestScore::ScoreOne_t::value_type("野村", -2),
    TestScore::ScoreOne_t::value_type("衣笠", 40),
};

// 不正データロード
auto ts2 = ts;
ASSERT_THROW(ts.AddScore(one_score_err), std::out_of_range);

// エクセプション 強い保証
ASSERT_TRUE(std::equal(ts.begin(), ts.end(), ts2.begin()));
}

TEST_F(SolidLSP_Q, TestScore_GetScore)
{
    auto ts = TestScore{};
    LoadCSV(test_score_org_, ts);

    auto const& score_0 = ts.GetScore("堂林");
    ASSERT_EQ({-1, 50, 80}, score_0);

    auto const& score_1 = ts.GetScore("広輔");
    ASSERT_EQ({40, 30, 10}, score_1);

    auto const& score_2 = ts.GetScore("會澤");
    ASSERT_EQ({30, 60, 70}, score_2);

    auto const& score_3 = ts.GetScore("松山");
    ASSERT_EQ({80, 90, 50}, score_3);

    auto const& score_4 = ts.GetScore("菊池");
    ASSERT_EQ({50, 20, 80}, score_4);

    auto const& score_5 = ts.GetScore("鈴木");
    ASSERT_EQ({0, 80, 100}, score_5);

    ASSERT_THROW(ts.GetScore("西川"), std::out_of_range);
}

TEST_F(SolidLSP_Q, TestScore_StoreCSV)
{
    auto ts = TestScore{};
    LoadCSV(test_score_org_, ts);
    StoreCSV(ts, test_score_act_);

    auto content_act = whole_file(test_score_act_);
    auto content_exp = whole_file(test_score_exp_);

    ASSERT_EQ(content_exp, content_act);

    // 不正ファイルロード
    auto ts2 = ts;
    ASSERT_THROW(StoreCSV(test_score_exp_err_, ts2), std::out_of_range);

    // エクセプション 強い保証
    ASSERT_TRUE(std::equal(ts.begin(), ts.end(), ts2.begin()));
}

```

```

TEST_F(SolidLSP_Q, TestScore_Average)
{
    auto ts = TestScore{};
    LoadCSV(test_score_org_, ts);

    auto const exp = TestScore::ScoreOne_t{
        TestScore::ScoreOne_t::value_type("堂林", 65),
        TestScore::ScoreOne_t::value_type("広輔", 26),
        TestScore::ScoreOne_t::value_type("會澤", 53),
        TestScore::ScoreOne_t::value_type("松山", 73),
        TestScore::ScoreOne_t::value_type("菊池", 50),
        TestScore::ScoreOne_t::value_type("鈴木", 60),
    };
    auto act = Average(ts);

    ASSERT_EQ(act, exp);
}
} // namespace

```

- 参照 [リスコフの置換原則\(LSP\)](#)
- [解答例-LSP](#)

演習-ISp

- 問題

```

// @@@ exercise/solid_q/isp_test_score.h 8
// [Q]
// 下記クラスTestScoreの管理データの内、受験者とその平均スコア、
// 平均スコアの高い順でソートされた受験者リストを扱うクラスが必要になったため、
// 下記のようにイミュータブルなクラスTestScoreAverageを作成した。
//
// 現在のファイル構成では、TestScoreAverageのみを使うクラスや関数にも、
// このファイル全体への依存を強いる(つまり、TestScoreやLoadCSV等に依存させる)ため、
// ISPに反する。
// TestScoreAverageを使うクラスや関数に余計な依存関係が発生しないようにリファクタリングを
// 行え。

```

```

class TestScore {
public:
    TestScore() = default;
    TestScore(TestScore const&) = default;
    TestScore& operator=(TestScore const&) = delete;
    TestScore& operator=(TestScore&&) = default;

    using ScoreAll_t = std::map<std::string, std::vector<int32_t>>;
    using ScoreOne_t = std::vector<std::pair<std::string, int32_t>>;

    void AddScore(ScoreOne_t const& one_test_score);
    std::vector<int32_t> const& GetScore(std::string const& name) const
    {
        return test_score_row_.at(name);
    }

    ScoreAll_t::const_iterator begin() const noexcept { return test_score_row_.begin(); }
    ScoreAll_t::const_iterator end() const noexcept { return test_score_row_.end(); }

private:
    // int32_t score: 0～100はスコア、-1は未受験、それ以外は不正データ
    void validate_score(int32_t score) const;
    ScoreAll_t test_score_row_{};
};

std::string ToString(TestScore const& ts);
TestScore LoadCSV(std::string const& filename);
void StoreCSV(TestScore const& test_score, std::string const& filename);
TestScore::ScoreOne_t Average(TestScore const& test_score);

class TestScoreAverage {
public:
    explicit TestScoreAverage(std::string const& filename);
    uint32_t GetAverage(std::string const& name) const;
    std::vector<std::string> const& DescendingOrder() const;

private:
    TestScore::ScoreOne_t const average_;
    mutable std::vector<std::string> desending_order_{};
};

```

```

// @@@ exercise/solid_q/isp_test_score.cpp 10

namespace {
std::pair<std::string, std::vector<int32_t>> parse_line(std::string const& line)
{
    auto const csv_sep = std::regex{R"(*, *)"};
    auto name      = std::string{};
    auto score     = std::vector<int32_t>{};

    auto end = std::sregex_token_iterator{};
    for (auto it = std::sregex_token_iterator{line.begin(), line.end(), csv_sep, -1}; it != end;
         ++it) {
        if (name.length() == 0) {
            name = *it;
        }
        else {
            auto s = std::stoi(*it);
            score.emplace_back(s);
        }
    }

    return {std::move(name), std::move(score)};
}

bool is_valid_score(int32_t score) noexcept { return 0 <= score && score <= 100; }

bool not_score(int32_t score) noexcept { return score == -1; }
} // namespace

void TestScore::validate_score(int32_t score) const
{
    if (is_valid_score(score) || not_score(score)) {
        return;
    }

    throw std::out_of_range{"Invalid Score"};
}

void TestScore::AddScore(TestScore::ScoreOne_t const& one_test_score)
{
    for (auto const& pair : one_test_score) {
        validate_score(pair.second);
    }

    if (test_score_row_.size() == 0) {
        test_score_row_[one_test_score[0].first] = std::vector<int32_t>{};
    }

    for (auto& pair : test_score_row_) {
        pair.second.push_back(-1);
    }

    auto curr_test_count = test_score_row_.begin()->second.size();

    for (auto const& pair : one_test_score) {
        if (test_score_row_.find(pair.first) == test_score_row_.end()) {
            test_score_row_[pair.first] = std::vector<int32_t>(curr_test_count, -1);
        }

        test_score_row_[pair.first].back() = pair.second;
    }
}

TestScore LoadCSV(std::string const& filename)
{
    auto data = std::ifstream{filename};

    auto test_score_raw = TestScore::ScoreAll_t{};
    auto line          = std::string{};
    while (std::getline(data, line)) {
        // std::pair<TestScore::ScoreAll_t::iterator, bool>
        auto ret = test_score_raw.insert(parse_line(line));
        assert(ret.second);
    }

    auto one_test = TestScore::ScoreOne_t{};
    for (auto const& pair : test_score_raw) {
        one_test.emplace_back(std::make_pair(pair.first, 0));
    }
}

```

```

auto const score_count = test_score_raw.begin()->second.size();
auto ts = TestScore{};

for (auto i = 0U; i < score_count; ++i) {
    for (auto& pair : one_test) {
        pair.second = test_score_raw[pair.first][i];
    }
    ts.AddScore(one_test);
}

return ts;
}

void StoreCSV(TestScore const& test_score, std::string const& filename)
{
    auto data = std::ofstream{filename};
    auto ss = std::ostringstream{};

    for (auto const& pair : test_score) {
        ss << pair.first;
        for (auto const s : pair.second) {
            ss << ", " << s;
        }
        ss << std::endl;
    }

    data << ss.str();
}

std::string ToString(TestScore const& ts)
{
    auto ss = std::ostringstream{};

    for (auto const& pair : ts) {
        ss << pair.first << ':';
        for (auto const s : pair.second) {
            ss << ' ' << s;
        }
        ss << std::endl;
    }

    return ss.str();
}

TestScore::ScoreOne_t Average(TestScore const& test_score)
{
    auto ret = TestScore::ScoreOne_t{};

    for (auto const& pair : test_score) {
        auto sum = 0;
        auto valid_count = 0U;
        for (auto const s : pair.second) {
            if (is_valid_score(s)) {
                sum += s;
                ++valid_count;
            }
        }
        ret.emplace_back(std::make_pair(pair.first, valid_count == 0 ? -1 : sum / valid_count));
    }

    return ret;
}

namespace {
TestScore::ScoreOne_t get_average(std::string const& filename)
{
    TestScore ts = LoadCSV(filename);

    return Average(ts);
}
} // namespace

TestScoreAverage::TestScoreAverage(std::string const& filename) : average_{get_average(filename)} {}

uint32_t TestScoreAverage::GetAverage(std::string const& name) const
{
    auto pos = std::find_if(average_.cbegin(), average_.cend(),
                           [&name](std::pair<std::string, int32_t> const& pair) noexcept {
                               return name == pair.first;
                           });
}

```

```

    if (pos == average_.cend()) {
        throw std::out_of_range{"no member"};
    }

    return pos->second;
}

std::vector<std::string> const& TestScoreAverage::DescendingOrder() const
{
    if (desending_order_.size() != 0) {
        return desending_order_;
    }

    auto ave = average_;
    std::sort(ave.begin(), ave.end(),
              [] (std::pair<std::string, int32_t> const& lhs, auto const& rhs) noexcept {
                  return lhs.second > rhs.second;
              });

    for (auto& pair : ave) {
        desending_order_.emplace_back(std::move(pair.first));
    }

    return desending_order_;
}

```

```

// @@@ exercise/solid_q/isp_test_score_ut.cpp 13

namespace {

TEST_F(SolidISP_Q, TestScore_LoadCSV)
{
    TestScore ts = LoadCSV(test_score_org_);

    auto const exp = TestScore::ScoreAll_t{
        TestScore::ScoreAll_t::value_type("堂林", {-1, 50, 80}),
        TestScore::ScoreAll_t::value_type("広輔", {40, 30, 10}),
        TestScore::ScoreAll_t::value_type("會澤", {30, 60, 70}),
        TestScore::ScoreAll_t::value_type("松山", {80, 90, 50}),
        TestScore::ScoreAll_t::value_type("菊池", {50, 20, 80}),
        TestScore::ScoreAll_t::value_type("鈴木", {0, 80, 100}),
    };

    ASSERT_TRUE(std::equal(ts.begin(), ts.end(), exp.begin()));

    auto exp_str = std::string{
        "堂林: -1 50 80\n広輔: 40 30 10\n會澤: 30 60 70\n",
        "松山: 80 90 50\n菊池: 50 20 80\n鈴木: 0 80 100\n";
    };
    ASSERT_EQ(ToString(ts), exp_str);
}

std::string whole_file(std::string const& filename)
{
    auto ifs = std::ifstream{filename};

    return std::string{std::istreambuf_iterator<char>{ifs}, std::istreambuf_iterator<char>{}};
}

TEST_F(SolidISP_Q, TestScore_AddScore)
{
    TestScore ts = LoadCSV(test_score_org_);

    auto const one_score = TestScore::ScoreOne_t{
        TestScore::ScoreOne_t::value_type("堂林", 50),
        TestScore::ScoreOne_t::value_type("広輔", 40),
        TestScore::ScoreOne_t::value_type("會澤", 70),
        TestScore::ScoreOne_t::value_type("松山", 1),
        TestScore::ScoreOne_t::value_type("菊池", -1),
        TestScore::ScoreOne_t::value_type("鈴木", 5),
        TestScore::ScoreOne_t::value_type("田中", 100),
        TestScore::ScoreOne_t::value_type("西川", 90),
    };

    ts.AddScore(one_score);

    auto const exp = TestScore::ScoreAll_t{
        TestScore::ScoreAll_t::value_type("堂林", {-1, 50, 80, 50}),
        TestScore::ScoreAll_t::value_type("広輔", {40, 30, 10, 40}),
        TestScore::ScoreAll_t::value_type("會澤", {30, 60, 70, 70}),
    };
}
```

```

    TestScore::ScoreAll_t::value_type("松山", {80, 90, 50, 1}),
    TestScore::ScoreAll_t::value_type("菊池", {50, 20, 80, -1}),
    TestScore::ScoreAll_t::value_type("鈴木", {0, 80, 100, 5}),
    TestScore::ScoreAll_t::value_type("田中", {-1, -1, -1, 100}),
    TestScore::ScoreAll_t::value_type("西川", {-1, -1, -1, 90}),
};

ASSERT_TRUE(std::equal(ts.begin(), ts.end(), exp.begin()));

auto const one_score_err = TestScore::ScoreOne_t{
    TestScore::ScoreOne_t::value_type("野村", -2),
    TestScore::ScoreOne_t::value_type("衣笠", 40),
};

// 不正データロード
auto ts2 = ts;
ASSERT_THROW(ts.AddScore(one_score_err), std::out_of_range);

// エクセプション 強い保証
ASSERT_TRUE(std::equal(ts.begin(), ts.end(), ts2.begin()));
}

TEST_F(SolidISP_Q, TestScore_GetScore)
{
    TestScore ts = LoadCSV(test_score_org_);

    auto const& score_0 = ts.GetScore("堂林");
    ASSERT_EQ({-1, 50, 80}, score_0);

    auto const& score_1 = ts.GetScore("広輔");
    ASSERT_EQ({40, 30, 10}, score_1);

    auto const& score_2 = ts.GetScore("會澤");
    ASSERT_EQ({30, 60, 70}, score_2);

    auto const& score_3 = ts.GetScore("松山");
    ASSERT_EQ({80, 90, 50}, score_3);

    auto const& score_4 = ts.GetScore("菊池");
    ASSERT_EQ({50, 20, 80}, score_4);

    auto const& score_5 = ts.GetScore("鈴木");
    ASSERT_EQ({0, 80, 100}, score_5);

    ASSERT_THROW(ts.GetScore("西川"), std::out_of_range);
}

TEST_F(SolidISP_Q, TestScore_StoreCSV)
{
    TestScore ts = LoadCSV(test_score_org_);
    StoreCSV(ts, test_score_act_);

    auto content_act = whole_file(test_score_act_);
    auto content_exp = whole_file(test_score_exp_);

    ASSERT_EQ(content_exp, content_act);

    // 不正ファイルロード
    auto ts2 = ts;
    ASSERT_THROW(ts2 = LoadCSV(test_score_exp_err_), std::out_of_range);

    // エクセプション 強い保証
    ASSERT_TRUE(std::equal(ts.begin(), ts.end(), ts2.begin()));
}

TEST_F(SolidISP_Q, TestScore_Average)
{
    TestScore ts = LoadCSV(test_score_org_);

    auto const exp = TestScore::ScoreOne_t{
        TestScore::ScoreOne_t::value_type("堂林", 65),
        TestScore::ScoreOne_t::value_type("広輔", 26),
        TestScore::ScoreOne_t::value_type("會澤", 53),
        TestScore::ScoreOne_t::value_type("松山", 73),
        TestScore::ScoreOne_t::value_type("菊池", 50),
        TestScore::ScoreOne_t::value_type("鈴木", 60),
    };
    auto act = Average(ts);

    ASSERT_EQ(act, exp);
}

```

```

}

TEST_F(SolidISP_Q, TestScoreAverage)
{
    auto tsa = TestScoreAverage{test_score_org_};

    ASSERT_EQ(tsa.GetAverage("堂林"), 65);
    ASSERT_EQ(tsa.GetAverage("広輔"), 26);
    ASSERT_EQ(tsa.GetAverage("會澤"), 53);
    ASSERT_EQ(tsa.GetAverage("松山"), 73);
    ASSERT_EQ(tsa.GetAverage("菊池"), 50);
    ASSERT_EQ(tsa.GetAverage("鈴木"), 60);

    ASSERT_THROW(tsa.GetAverage("野村"), std::out_of_range);

    auto const exp = std::vector<std::string>{
        "松山", "堂林", "鈴木", "會澤", "菊池", "広輔",
    };

    ASSERT_EQ(tsa.DescendingOrder(), exp);
    ASSERT_EQ(tsa.DescendingOrder(), exp); // キャッシュのテスト
}
} // namespace

```

- 参照 [インターフェース分離の原則\(ISP\)](#)
- [解答例-ISP](#)

演習-DIP

- 問題

```

// @@@ exercise/solid_q/dip_test_score.h 9
// [Q]
// クラスTestScoreClientは、
//     * dip_test_score_client.h
//     * dip_test_score_client.cpp
// で宣言・定義され。
// クラスTestScoreLoaderは、
//     * dip_test_score.h(このファイル)
//     * dip_test_score.cpp
// で宣言・定義されされている。
// TestScoreLoaderは宣言・定義の中にTestScoreClientを使用しているため、
//     * dip_test_score.cpp -> dip_test_score_client.h
// の依存関係が発生してる(dip_test_score.h -> dip_test_score_client.hの依存関係は、
// dip_test_score.h内のTestScoreClientの前方宣言で回避)。
// クラスの名前からもわかる通り、
//     * TestScoreClientはTestScoreLoaderのクライアント
//     * TestScoreLoaderはTestScoreClientのサーバ
// であるため、この依存関係
//     * TestScoreLoader -> TestScoreClient(逆の依存関係もあるため、双方向依存)
//     * dip_test_score.cpp -> dip_test_score_client.h
// はDIPに反し、機能拡張(や、場合によっては単体テスト可能なパッケージ構成維持)
// に多大な悪影響がある(TestScoreLoaderを使うTestScoreClient2を新たに定義したときに
// TestScoreLoaderがどのように修正されるかを考えればこの問題に気づくだろう)。
// この問題に対処せよ。

class TestScore {
public:
    TestScore() = default;
    TestScore(TestScore const&) = default;
    TestScore& operator=(TestScore const&) = delete;
    TestScore& operator=(TestScore&&) = default;

    using ScoreAll_t = std::map<std::string, std::vector<int32_t>>;
    using ScoreOne_t = std::vector<std::pair<std::string, int32_t>>;

    void AddScore(ScoreOne_t const& one_test_score);
    std::vector<int32_t> const& GetScore(std::string const& name) const
    {
        return test_score_row_.at(name);
    }

    ScoreAll_t::const_iterator begin() const noexcept { return test_score_row_.begin(); }
    ScoreAll_t::const_iterator end() const noexcept { return test_score_row_.end(); }

private:
    // int32_t score: 0~100はスコア、-1は未受験、それ以外は不正データ
    void validate_score(int32_t score) const;
    ScoreAll_t test_score_row_{};
}

```

```

};

std::string      ToString(TestScore const& ts);
TestScore        LoadCSV(std::string const& filename);
void             StoreCSV(TestScore const& test_score, std::string const& filename);
TestScore::ScoreOne_t Average(TestScore const& test_score);

class TestScoreClient;
class TestScoreLoader {
public:
    TestScoreLoader() {}
    ~TestScoreLoader();
    void    LoadCSV_Async(std::string&& filename, TestScoreClient& client);
    TestScore LoadCSV_Get() { return future_.get(); }

private:
    std::future<TestScore> future_{};
};

// @@@ exercise/solid_q/dip_test_score.cpp 11

```

```

namespace {
std::pair<std::string, std::vector<int32_t>> parse_line(std::string const& line)
{
    auto const csv_sep = std::regex{R"( *, *)"};
    auto name     = std::string{};
    auto score    = std::vector<int32_t>{};

    auto end = std::sregex_token_iterator{};
    for (auto it = std::sregex_token_iterator{line.begin(), line.end(), csv_sep, -1}; it != end;
         ++it) {
        if (name.length() == 0) {
            name = *it;
        }
        else {
            auto s = std::stoi(*it);
            score.emplace_back(s);
        }
    }
    return {std::move(name), std::move(score)};
}

bool is_valid_score(int32_t score) noexcept { return 0 <= score && score <= 100; }

bool not_score(int32_t score) noexcept { return score == -1; }
} // namespace

void TestScore::validate_score(int32_t score) const
{
    if (is_valid_score(score) || not_score(score)) {
        return;
    }

    throw std::out_of_range{"Invalid Score"};
}

void TestScore::AddScore(TestScore::ScoreOne_t const& one_test_score)
{
    for (auto const& pair : one_test_score) {
        validate_score(pair.second);
    }

    if (test_score_row_.size() == 0) {
        test_score_row_[one_test_score[0].first] = std::vector<int32_t>{};
    }

    for (auto& pair : test_score_row_) {
        pair.second.push_back(-1);
    }

    auto curr_test_count = test_score_row_.begin()->second.size();

    for (auto const& pair : one_test_score) {
        if (test_score_row_.find(pair.first) == test_score_row_.end()) {
            test_score_row_[pair.first] = std::vector<int32_t>(curr_test_count, -1);
        }

        test_score_row_[pair.first].back() = pair.second;
    }
}

```

```

}

TestScore LoadCSV(std::string const& filename)
{
    auto data = std::ifstream{filename};

    auto test_score_raw = TestScore::ScoreAll_t{};
    auto line          = std::string{};
    while (std::getline(data, line)) {
        // std::pair<TestScore::ScoreAll_t::iterator, bool>
        auto ret = test_score_raw.insert(parse_line(line));
        assert(ret.second);
    }

    auto one_test = TestScore::ScoreOne_t{};
    for (auto const& pair : test_score_raw) {
        one_test.emplace_back(std::make_pair(pair.first, 0));
    }

    auto const score_count = test_score_raw.begin()->second.size();
    auto ts               = TestScore{};

    for (auto i = 0U; i < score_count; ++i) {
        for (auto& pair : one_test) {
            pair.second = test_score_raw[pair.first][i];
        }
        ts.AddScore(one_test);
    }

    return ts;
}

void StoreCSV(TestScore const& test_score, std::string const& filename)
{
    auto data = std::ofstream{filename};
    auto ss   = std::ostringstream{};

    for (auto const& pair : test_score) {
        ss << pair.first;
        for (auto const s : pair.second) {
            ss << ", " << s;
        }
        ss << std::endl;
    }

    data << ss.str();
}

std::string ToString(TestScore const& ts)
{
    auto ss = std::ostringstream{};

    for (auto const& pair : ts) {
        ss << pair.first << ':';
        for (auto const s : pair.second) {
            ss << ' ' << s;
        }
        ss << std::endl;
    }

    return ss.str();
}

TestScore::ScoreOne_t Average(TestScore const& test_score)
{
    auto ret = TestScore::ScoreOne_t{};

    for (auto const& pair : test_score) {
        auto sum      = 0;
        auto valid_count = 0U;
        for (auto const s : pair.second) {
            if (is_valid_score(s)) {
                sum += s;
                ++valid_count;
            }
        }
        ret.emplace_back(std::make_pair(pair.first, valid_count == 0 ? -1 : sum / valid_count));
    }

    return ret;
}

```

```

}

TestScoreLoader::~TestScoreLoader()
{
    if (future_.valid()) {
        future_.get();
    }
}

void TestScoreLoader::LoadCSV_Async(std::string&& filename, TestScoreClient& client)
{
    if (future_.valid()) {
        future_.get();
    }

    future_ = std::async(std::launch::async, [&client, filename = std::move(filename)]() {
        auto test_score = LoadCSV(filename);
        client.Done();
        return test_score;
    });
}

```

```

// @@@ exercise/solid_q/dip_test_score_ut.cpp 13

namespace {

TEST_F(SolidDIP_Q, TestScore_LoadCSV)
{
    TestScore ts = LoadCSV(test_score_org_);

    auto const exp = TestScore::ScoreAll_t{
        TestScore::ScoreAll_t::value_type("堂林", {-1, 50, 80}),
        TestScore::ScoreAll_t::value_type("広輔", {40, 30, 10}),
        TestScore::ScoreAll_t::value_type("會澤", {30, 60, 70}),
        TestScore::ScoreAll_t::value_type("松山", {80, 90, 50}),
        TestScore::ScoreAll_t::value_type("菊池", {50, 20, 80}),
        TestScore::ScoreAll_t::value_type("鈴木", {0, 80, 100}),
    };

    ASSERT_TRUE(std::equal(ts.begin(), ts.end(), exp.begin()));

    auto exp_str = std::string{
        "堂林: -1 50 80\n広輔: 40 30 10\n會澤: 30 60 70\n"
        "松山: 80 90 50\n菊池: 50 20 80\n鈴木: 0 80 100\n"};
    ASSERT_EQ(ToString(ts), exp_str);
}

std::string whole_file(std::string const& filename)
{
    auto ifs = std::ifstream{filename};

    return std::string{std::istreambuf_iterator<char>{ifs}, std::istreambuf_iterator<char>{}};
}

TEST_F(SolidDIP_Q, TestScore_AddScore)
{
    TestScore ts = LoadCSV(test_score_org_);

    auto const one_score = TestScore::ScoreOne_t{
        TestScore::ScoreOne_t::value_type("堂林", 50),
        TestScore::ScoreOne_t::value_type("広輔", 40),
        TestScore::ScoreOne_t::value_type("會澤", 70),
        TestScore::ScoreOne_t::value_type("松山", 1),
        TestScore::ScoreOne_t::value_type("菊池", -1),
        TestScore::ScoreOne_t::value_type("鈴木", 5),
        TestScore::ScoreOne_t::value_type("田中", 100),
        TestScore::ScoreOne_t::value_type("西川", 90),
    };

    ts.AddScore(one_score);

    auto const exp = TestScore::ScoreAll_t{
        TestScore::ScoreAll_t::value_type("堂林", {-1, 50, 80, 50}),
        TestScore::ScoreAll_t::value_type("広輔", {40, 30, 10, 40}),
        TestScore::ScoreAll_t::value_type("會澤", {30, 60, 70, 70}),
        TestScore::ScoreAll_t::value_type("松山", {80, 90, 50, 1}),
        TestScore::ScoreAll_t::value_type("菊池", {50, 20, 80, -1}),
        TestScore::ScoreAll_t::value_type("鈴木", {0, 80, 100, 5}),
        TestScore::ScoreAll_t::value_type("田中", {-1, -1, -1, 100}),
        TestScore::ScoreAll_t::value_type("西川", {-1, -1, -1, 90}),
    };
}
```

```

};

ASSERT_TRUE(std::equal(ts.begin(), ts.end(), exp.begin()));

auto const one_score_err = TestScore::ScoreOne_t{
    TestScore::ScoreOne_t::value_type("野村", -2),
    TestScore::ScoreOne_t::value_type("衣笠", 40),
};

// 不正データロード
auto ts2 = ts;
ASSERT_THROW(ts.AddScore(one_score_err), std::out_of_range);

// エクセプション 強い保証
ASSERT_TRUE(std::equal(ts.begin(), ts.end(), ts2.begin())));
}

TEST_F(SolidDIP_Q, TestScore_GetScore)
{
    TestScore ts = LoadCSV(test_score_org_);

    auto const& score_0 = ts.GetScore("堂林");
    ASSERT_EQ((std::vector{-1, 50, 80}), score_0);

    auto const& score_1 = ts.GetScore("広輔");
    ASSERT_EQ((std::vector{40, 30, 10}), score_1);

    auto const& score_2 = ts.GetScore("會澤");
    ASSERT_EQ((std::vector{30, 60, 70}), score_2);

    auto const& score_3 = ts.GetScore("松山");
    ASSERT_EQ((std::vector{80, 90, 50}), score_3);

    auto const& score_4 = ts.GetScore("菊池");
    ASSERT_EQ((std::vector{50, 20, 80}), score_4);

    auto const& score_5 = ts.GetScore("鈴木");
    ASSERT_EQ((std::vector{0, 80, 100}), score_5);

    ASSERT_THROW(ts.GetScore("西川"), std::out_of_range);
}

TEST_F(SolidDIP_Q, TestScore_StoreCSV)
{
    TestScore ts = LoadCSV(test_score_org_);
    StoreCSV(ts, test_score_act_);

    auto content_act = whole_file(test_score_act_);
    auto content_exp = whole_file(test_score_exp_);

    ASSERT_EQ(content_exp, content_act);

    // 不正ファイルロード
    auto ts2 = ts;
    ASSERT_THROW(ts2 = LoadCSV(test_score_exp_err_), std::out_of_range);

    // エクセプション 強い保証
    ASSERT_TRUE(std::equal(ts.begin(), ts.end(), ts2.begin())));
}

TEST_F(SolidDIP_Q, TestScore_Average)
{
    TestScore ts = LoadCSV(test_score_org_);

    auto const exp = TestScore::ScoreOne_t{
        TestScore::ScoreOne_t::value_type("堂林", 65),
        TestScore::ScoreOne_t::value_type("広輔", 26),
        TestScore::ScoreOne_t::value_type("會澤", 53),
        TestScore::ScoreOne_t::value_type("松山", 73),
        TestScore::ScoreOne_t::value_type("菊池", 50),
        TestScore::ScoreOne_t::value_type("鈴木", 60),
    };
    auto act = Average(ts);

    ASSERT_EQ(act, exp);
}
} // namespace

```

```
// @@@ exercise/solid_q/dip_test_score_client.h 11
```

```

class TestScoreClient {
public:
    void LoadAsync(std::string&& filename);
    void Done();
    void Wait();
    TestScore const& GetTestScore() const noexcept { return test_score_; }

private:
    std::condition_variable condition_{};
    std::mutex mutex_{};
    TestScore test_score_{};
    TestScoreLoader loader_{};
    bool loaded_{false};
};


```

```

// @@@ exercise/solid_q/dip_test_score_client.cpp 5

void TestScoreClient::LoadAsync(std::string&& filename)
{
    loader_.LoadCSV_Async(std::move(filename), *this);
}

void TestScoreClient::Done()
{
{
    auto lock = std::lock_guard{mutex_};
    loaded_ = true;
}

    condition_.notify_all();
}

void TestScoreClient::Wait()
{
    auto lock = std::unique_lock{mutex_};

    condition_.wait(lock, [&loaded = loaded_] { return loaded; });

    test_score_ = loader_.LoadCSV_Get();
}

```

```

// @@@ exercise/solid_q/dip_test_score_client_ut.cpp 9

namespace {

TEST_F(SolidDIP_Q, TestScoreClient_LoadAsync)
{
    auto tsc = TestScoreClient{};

    tsc.LoadAsync("../ut_data2/test_score_org.csv");

    auto const exp = TestScore::ScoreAll_t{
        TestScore::ScoreAll_t::value_type("豈林", {-1, 50, 80}),
        TestScore::ScoreAll_t::value_type("広輔", {40, 30, 10}),
        TestScore::ScoreAll_t::value_type("會澤", {30, 60, 70}),
        TestScore::ScoreAll_t::value_type("松山", {80, 90, 50}),
        TestScore::ScoreAll_t::value_type("菊池", {50, 20, 80}),
        TestScore::ScoreAll_t::value_type("鈴木", {0, 80, 100}),
    };

    // ここで何か別のことをして終わったら
    tsc.Wait();

    auto const& ts = tsc.GetTestScore();

    ASSERT_TRUE(std::equal(ts.begin(), ts.end(), exp.begin()));

    auto const exp_str = std::string{
        "豈林: -1 50 80\n広輔: 40 30 10\n會澤: 30 60 70\n"
        "松山: 80 90 50\n菊池: 50 20 80\n鈴木: 0 80 100\n";
    };
    ASSERT_EQ(ToString(ts), exp_str);
}
} // namespace

```

- 参照 依存関係逆転の原則(DIP)
- 解答例-DIP

演習-SOLIDの定義

- 問題

下記原則群と説明群を結びつけよ。

- 原則群

- SRP
- OCP
- LIP
- ISP
- DIP

- 説明群

1. 一つのクラスは、ただ一つの責任(機能)を持つようにしなければならない。
2. クラスは拡張に対して開いていて、クラスは修正に対して閉じていなければならない。
3. 事前条件を派生クラスで強めることはできず、事後条件を派生クラスで弱めることはできない。
4. クラスは、そのクライアントが使用しないメソッドへの依存をそのクライアントに強制するべきではない。
5. 上位レベルのモジュールは下位レベルのモジュールに依存すべきではない。

- 解答-SOLIDの定義

デザインパターン

演習-ガード節

- 問題

```
// @@@ exercise/design_pattern_q/guard.cpp 7
// [Q]
// 以下の関数PrimeNumbersをガード節や、関数の括りだし等によってリファクタリングせよ。

std::optional<std::vector<uint32_t>> PrimeNumbers(uint32_t max_number)
{
    if (max_number < 65536) { // 演算コストが高いためエラーにする
        auto result = std::vector<uint32_t>{};

        if (max_number >= 2) {
            auto prime_num     = 2U;                                // 最初の素数
            auto is_num_prime = std::vector<bool>(max_number + 1, true); // falseなら素数でない
            is_num_prime[0] = is_num_prime[1] = false;

            do {
                result.emplace_back(prime_num);

                for (auto i = 2 * prime_num; i < is_num_prime.size(); i += prime_num) {
                    is_num_prime[i] = false; // 次の倍数は素数ではない
                }

                do { // 次の素数の探索
                    ++prime_num;
                } while (!is_num_prime[prime_num] && (prime_num < is_num_prime.size()));

                } while (prime_num < is_num_prime.size());
        }
        return result;
    }

    return std::nullopt;
}

TEST(DesignPatternQ, Guard)
{
    auto result = PrimeNumbers(1);
    ASSERT_TRUE(result);
    ASSERT_EQ((std::vector<uint32_t>{}), *result);

    result = PrimeNumbers(2);
    ASSERT_TRUE(result);
    ASSERT_EQ((std::vector<uint32_t>{2}), *result);

    result = PrimeNumbers(30);
    ASSERT_TRUE(result);
}
```

```

    ASSERT_EQ((std::vector<uint32_t>{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}), *result);
    ASSERT_FALSE(PrimeNumbers(65536));
}

```

- 参照 ガード節
- 解答例-ガード節

演習-BitmaskType

- 問題

```

// @@@ exercise/design_pattern_q/enum_bitmask.cpp 5
// [Q]
// 下記関数ColorMask2Strはuint32_t型のビットマスクを引数に取る。
// これはユーザが使用間違いを起こしやすい脆弱なインターフェースである。
// enumによるビットマスク表現を使用しこの問題に対処せよ。

constexpr auto COLOR_RED    = 0b0001U;
constexpr auto COLOR_YELLOW = 0b0010U;
constexpr auto COLOR_GREEN   = 0b0100U;
constexpr auto COLOR_BLUE    = 0b1000U;

std::string ColorMask2Str(uint32_t color)
{
    auto ret = std::string{};

    if (COLOR_RED & color) {
        ret += "RED";
    }
    if (COLOR_YELLOW & color) {
        if (ret.size() != 0) {
            ret += ',';
        }
        ret += "YELLOW";
    }
    if (COLOR_GREEN & color) {
        if (ret.size() != 0) {
            ret += ',';
        }
        ret += "GREEN";
    }
    if (COLOR_BLUE & color) {
        if (ret.size() != 0) {
            ret += ',';
        }
        ret += "BLUE";
    }

    return ret;
}

TEST(DesignPatternQ, EnumBitmask)
{
    ASSERT_EQ("RED", ColorMask2Str(COLOR_RED));
    ASSERT_EQ("RED,YELLOW", ColorMask2Str(COLOR_RED | COLOR_YELLOW));
    ASSERT_EQ("YELLOW", ColorMask2Str(COLOR_YELLOW));
    ASSERT_EQ("YELLOW,GREEN,BLUE", ColorMask2Str(COLOR_YELLOW | COLOR_GREEN | COLOR_BLUE));

    ASSERT_EQ("", ColorMask2Str(0b1000)); // 想定していない使用法
}

```

- 参照 BitmaskType
- 解答例-BitmaskType

演習-Pimpl

- 問題

```

// @@@ exercise/design_pattern_q/pimpl.cpp 5
// [Q] 下記クラスCollectionの宣言はWidgetの宣言に依存している。
// Pimplパターンを使用し、Collectionの宣言がWidgetの宣言に依存しないようにせよ。

class Widget {
public:
    explicit Widget(char const* name) : name_{name} {}
    char const* Name() const noexcept { return name_; }

```

```

private:
    char const* name_;
};

class Collection {
public:
    char const* Name(size_t i) const { return widgets_.at(i).Name(); }
    void AddName(char const* name) { widgets_.emplace_back(name); }

    size_t Count() const noexcept { return widgets_.size(); }

private:
    std::vector<Widget> widgets_{};
};

TEST(DesignPatternQ, Pimpl)
{
    auto c = Collection{};

    ASSERT_EQ(0, c.Count());
    ASSERT_THROW(c.Name(0), std::out_of_range);

    c.AddName("n0");
    c.AddName("n1");
    c.AddName("n2");

    ASSERT_EQ(3, c.Count());
    ASSERT_STREQ("n0", c.Name(0));
    ASSERT_STREQ("n1", c.Name(1));
    ASSERT_STREQ("n2", c.Name(2));
    ASSERT_THROW(c.Name(4), std::out_of_range);
}

```

- 参照 [Pimpl](#)
- [解答例-Pimpl](#)

演習-Accessorの副作用

- 問題

Accessor(特にセッター)には重大な副作用がある。その副作用を下記から選択せよ。
- 選択肢
 1. クラスのカプセル化の破壊
 2. SRPへの違反
 3. 関数の巨大化
 4. クラスの巨大化
- [解答-Accessorの副作用](#)

演習-Accessor

- 問題

```

// @@@ exercise/design_pattern_q/Accessor.cpp 5
// [Q]
// 下記クラスPrimeNumbersはAccessorの多用により、クラスのカプセル化が破壊されている例である。
// これにより、このクラスは凝集性が低く、誤用を誘発しやすい。
// この問題を解決するため、クラスPrimeNumbersや関数GetPrimeNumbersを修正せよ。
// また、別の問題があれば合わせて修正せよ。

class PrimeNumbers {
public:
    uint32_t GetMaxNumber() const { return max_number_; }
    void SetMaxNumber(uint32_t max_number) { max_number_ = max_number; }
    bool HasCache() const { return cached_; }
    void Cashed(bool cached) { cached_ = cached; }

    std::vector<uint32_t>& GetPrimeNumbers() { return prime_numbers_; }

private:
    uint32_t max_number_;
    bool cached_;
    std::vector<uint32_t> prime_numbers_;
};

```

```

};

inline uint32_t next_prime_num(uint32_t curr_prime_num, std::vector<bool>& is_num_prime)
{
    for (auto i = 2 * curr_prime_num; i < is_num_prime.size(); i += curr_prime_num) {
        is_num_prime[i] = false; // 次の倍数は素数ではない
    }

    auto prime_num = curr_prime_num;

    do { // 次の素数の探索
        ++prime_num;
    } while (!is_num_prime[prime_num] && (prime_num < is_num_prime.size()));

    return prime_num;
}

inline std::vector<uint32_t> get_prime_numbers(uint32_t max_number)
{
    auto result      = std::vector<uint32_t>{};
    auto prime_num   = 2U;                                // 最初の素数
    auto is_num_prime = std::vector<bool>(max_number + 1, true); // falseなら素数でない。
    is_num_prime[0] = is_num_prime[1] = false;

    do {
        result.emplace_back(prime_num);
        prime_num = next_prime_num(prime_num, is_num_prime);
    } while (prime_num < is_num_prime.size());

    return result;
}

void GetPrimeNumbers(PrimeNumbers& pm)
{
    if (pm.HasCache()) {
        return;
    }

    if (pm.GetMaxNumber() < 2) { // ガード節。2未満の素数はない。
        pm.GetPrimeNumbers().clear();
        return;
    }

    pm.GetPrimeNumbers() = get_prime_numbers(pm.GetMaxNumber());
}

TEST(DesignPatternQ, Accessor)
{
    auto pm = PrimeNumbers{};

    pm.SetMaxNumber(1);
    pm.Cashed(false);
    GetPrimeNumbers(pm);
    pm.Cashed(true);

    ASSERT_EQ((std::vector<uint32_t>{}), pm.GetPrimeNumbers());

    pm.SetMaxNumber(3);
    pm.Cashed(false);
    GetPrimeNumbers(pm);
    pm.Cashed(true);

    ASSERT_EQ((std::vector<uint32_t>{2, 3}), pm.GetPrimeNumbers());

    pm.SetMaxNumber(30);
    pm.Cashed(false);
    GetPrimeNumbers(pm);
    pm.Cashed(true);

    ASSERT_EQ((std::vector<uint32_t>{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}), pm.GetPrimeNumbers());

    pm.SetMaxNumber(3);
    GetPrimeNumbers(pm); // pm.Cashed(false);しないので前のまま。
                        // このような用途は考えづらいので、おそらく仕様のバグ。

    ASSERT_EQ((std::vector<uint32_t>{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}), pm.GetPrimeNumbers());
}

```

- 参照 [Accessor](#)
- 解答例-[Accessor](#)

演習-Copy-And-Swap

- 問題

```
// @@@ exercise/design_pattern_q/copy_and_swap.cpp 5
// [Q]
// 以下のクラスCopyAndSwapの
// * copyコンストラクタ
// * copy代入演算子
// * moveコンストラクタ
// * move代入演算子
// をCopy-And-Swapイデオムを使用して実装し、単体テストを行え。

class CopyAndSwap final {
public:
    explicit CopyAndSwap(char const* name0, char const* name1)
        : name0_{name0 == nullptr ? "" : name0}, name1_{name1 == nullptr ? "" : name1}
    {}

    CopyAndSwap(CopyAndSwap const& rhs)
    {
        // この関数の実装
    }

    CopyAndSwap(CopyAndSwap&& rhs) noexcept
    {
        // この関数の実装
    }

    CopyAndSwap& operator=(CopyAndSwap const& rhs)
    {
        // この関数の実装
        return *this;
    }

    CopyAndSwap& operator=(CopyAndSwap&& rhs) noexcept
    {
        // この関数の実装
        return *this;
    }

    void Swap(CopyAndSwap& rhs) noexcept
    {
        // この関数の実装
    }

    char const* GetName0() const noexcept { return name0_; }

    std::string const& GetName1() const noexcept { return name1_; }

    ~CopyAndSwap() = default;

private:
    char const* name0_;
    std::string name1_;
};

TEST(DesignPatternQ, CopyAndSwap)
{
    // test for explicit CopyAndSwap(char const* name0, char const* name1)
    auto n = CopyAndSwap(nullptr, nullptr);
    ASSERT_STREQ("", n.GetName0());
    ASSERT_EQ("", n.GetName1());

    auto a = CopyAndSwap{"a0", "a1"};
    ASSERT_STREQ("a0", a.GetName0());
    ASSERT_EQ("a1", a.GetName1());
}
```

- 参照 [Copy-And-Swap](#)
- 解答例 [Copy-And-Swap](#)

演習-Immutable

- 問題

```

// @@@ exercise/design_pattern_q/immutable.cpp 5
// [Q]
// 下記クラスPrimeNumbersはSetMaxNumberにより状態が変わってしまうことがある。
// 状態変更が必要ない場合、こういった仕様はない方が良い。
// PrimeNumbersからSetMaxNumberを削除し、このクラスをimmutableにせよ。

class PrimeNumbers {
public:
    PrimeNumbers() = default;

    PrimeNumbers(PrimeNumbers const&) = default;
    PrimeNumbers& operator=(PrimeNumbers const&) = default;

    uint32_t GetMaxNumber() const noexcept { return max_number_; }
    void SetMaxNumber(uint32_t max_number) noexcept
    {
        if (max_number != max_number_) {
            cached_ = false;
            max_number_ = max_number;
        }
    }

    std::vector<uint32_t> const& GeneratePrimeNumbers();

private:
    uint32_t max_number_{0};
    bool cached_{false};
    std::vector<uint32_t> prime_numbers_{};

    static uint32_t next_prime_num(uint32_t curr_prime_num,
                                  std::vector<bool>& is_num_prime) noexcept;
    static std::vector<uint32_t> get_prime_numbers(uint32_t max_number);
};

uint32_t PrimeNumbers::next_prime_num(uint32_t curr_prime_num,
                                     std::vector<bool>& is_num_prime) noexcept
{
    for (auto i = 2 * curr_prime_num; i < is_num_prime.size(); i += curr_prime_num) {
        is_num_prime[i] = false; // 次の倍数は素数ではない
    }

    auto prime_num = curr_prime_num;

    do { // 次の素数の探索
        ++prime_num;
    } while (!is_num_prime[prime_num] && (prime_num < is_num_prime.size()));

    return prime_num;
}

std::vector<uint32_t> PrimeNumbers::get_prime_numbers(uint32_t max_number)
{
    auto result = std::vector<uint32_t>{};
    auto prime_num = 2U; // 最初の素数
    auto is_num_prime = std::vector<bool>(max_number + 1, true); // falseなら素数でない。
    is_num_prime[0] = is_num_prime[1] = false;

    do {
        result.emplace_back(prime_num);
        prime_num = next_prime_num(prime_num, is_num_prime);
    } while (prime_num < is_num_prime.size());

    return result;
}

std::vector<uint32_t> const& PrimeNumbers::GeneratePrimeNumbers()
{
    if (cached_) {
        return prime_numbers_;
    }

    if (max_number_ < 2) { // ガード節。2未満の素数はない。
        prime_numbers_.clear();
    }
    else {
        prime_numbers_ = get_prime_numbers(max_number_);
    }

    cached_ = true;
    return prime_numbers_;
}

```

```

}

TEST(DesignPatternQ, Immutable)
{
    auto pm = PrimeNumbers{};

    pm.SetMaxNumber(1);
    ASSERT_EQ((std::vector<uint32_t>{}), pm.GeneratePrimeNumbers());

    pm.SetMaxNumber(3);
    ASSERT_EQ((std::vector<uint32_t>{2, 3}), pm.GeneratePrimeNumbers());

    pm.SetMaxNumber(30);
    ASSERT_EQ((std::vector<uint32_t>{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}),
              pm.GeneratePrimeNumbers());

    ASSERT_EQ((std::vector<uint32_t>{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}),
              pm.GeneratePrimeNumbers());

    pm.SetMaxNumber(3);
    ASSERT_EQ((std::vector<uint32_t>{2, 3}), pm.GeneratePrimeNumbers());

    auto pm3_copy = PrimeNumbers{pm};
    ASSERT_EQ((std::vector<uint32_t>{2, 3}), pm3_copy.GeneratePrimeNumbers());

    auto pm5 = PrimeNumbers{};
    pm5.SetMaxNumber(5);
    pm = pm5;
    ASSERT_EQ((std::vector<uint32_t>{2, 3, 5}), pm.GeneratePrimeNumbers());
}

```

- 参照 [Immutable](#)
- [解答例-Immutable](#)

演習-Clone

- 問題

```

// @@@ exercise/design_pattern_q/clone.cpp 5
// [Q]
// TEST(DesignPatternQ, Clone)に記述したように、オブジェクトのスライシングによる影響で、
// Base型ポインタに代入されたDerivedインスタンスへのコピーは部分的にしか行われない。
// Cloneパターンを使用してこの問題を修正せよ。
// また、その他の問題があれば合わせて修正せよ。

class Base {
public:
    explicit Base(std::string name) : name1_{name} {}
    virtual ~Base() = default;
    virtual std::string const& GetName() { return name1_; }

private:
    std::string name1_;
};

class Derived final : public Base {
public:
    explicit Derived(std::string name1 = "", std::string name2 = "") : Base{name1}, name2_{name2} {}
    virtual ~Derived() = default;
    virtual std::string const& GetName() { return name2_; }

private:
    std::string name2_;
};

TEST(DesignPatternQ, Clone)
{
    Derived d1{"name1", "name2"};

    ASSERT_EQ("name1", d1.Base::GetName());
    ASSERT_EQ("name2", d1.GetName());

    Derived d2{d1};
    ASSERT_EQ("name1", d2.Base::GetName());
    ASSERT_EQ("name2", d2.GetName());

    Derived d3;
    Base* b3 = &d3;

```

```

*b3 = d1; // d1からd3へコピーしたつもりだが、スライスによりうまく行かない。

ASSERT_EQ("name1", b3->Base::GetName());
#if 0
    ASSERT_EQ("name2", b3->GetName()); // スライスの影響でname2_がコピーされていない。
#else
    ASSERT_EQ("", d3.GetName());
#endif
}

```

- 参照 [Clone\(仮想コンストラクタ\)](#)
- [解答例-Clone](#)

演習-NVI

- 問題

```

// @@@ exercise/design_pattern_q/nvi.cpp 7
// [Q]
// 下記クラスBase、Derived、DerivedDerivedの前処理はクローンコードになっている。
// NVIを用いて、この問題に対処せよ。

class Base {
public:
    explicit Base(std::string name) : name1_{name} {}
    virtual ~Base() = default;
    std::string const& GetName1() const noexcept { return name1_; }

    virtual bool IsEqual(Base const& rhs) const noexcept
    {
        if (this == &rhs) {
            return true;
        }

        if (typeid(*this) != typeid(rhs)) {
            return false;
        }

        return name1_ == rhs.name1_;
    }
private:
    std::string name1_;
};

class Derived : public Base {
public:
    explicit Derived(std::string name1 = "", std::string name2 = "") : Base{name1}, name2_{name2} {}
    virtual ~Derived() override = default;
    std::string const& GetName2() const noexcept { return name2_; }

    virtual bool IsEqual(Base const& rhs) const noexcept override
    {
        if (this == &rhs) {
            return true;
        }

        if (!Base::IsEqual(rhs)) {
            return false;
        }

        auto rhs_d = dynamic_cast<Derived const*>(&rhs);

        return (rhs_d != nullptr) && (name2_ == rhs_d->name2_);
    }
private:
    std::string name2_;
};

class DerivedDerived : public Derived {
public:
    explicit DerivedDerived(std::string name1 = "", std::string name2 = "", std::string name3 = "")
        : Derived{name1, name2}, name3_{name3}
    {
    }
    virtual ~DerivedDerived() override = default;
    std::string const& GetName3() const noexcept { return name3_; }
}

```

```

virtual bool IsEqual(Base const& rhs) const noexcept override
{
    if (this == &rhs) {
        return true;
    }

    if (!Derived::IsEqual(rhs)) {
        return false;
    }

    auto rhs_d = dynamic_cast<DerivedDerived const*>(&rhs);

    return (rhs_d != nullptr) && (name3_ == rhs_d->name3_);
}

private:
    std::string name3_;
};

TEST(DesignPatternQ, NVI)
{
    auto b1 = Base{"b1"};

    ASSERT_TRUE(b1.IsEqual(Base{b1}));
    ASSERT_TRUE(b1.IsEqual(Base{"b1"}));
    ASSERT_FALSE(b1.IsEqual(Base{"b2"}));
    ASSERT_FALSE(b1.IsEqual(Derived{"b1", "d1"}));

    auto d1 = Derived{"b1", "d1"};

    ASSERT_FALSE(d1.IsEqual(Base{"b1"}));
    ASSERT_TRUE(d1.IsEqual(d1));
    ASSERT_TRUE(d1.IsEqual(Derived{"b1", "d1"}));
    ASSERT_FALSE(d1.IsEqual(Derived{"b1", "d2"}));
    ASSERT_FALSE(d1.IsEqual(DerivedDerived{"b1", "d1", "dd2"}));

    auto dd1 = DerivedDerived{"b1", "d1", "dd1"};

    ASSERT_FALSE(dd1.IsEqual(Base{"b1"}));
    ASSERT_FALSE(dd1.IsEqual(Derived{"b1", "d1"}));
    ASSERT_TRUE(dd1.IsEqual(dd1));
    ASSERT_TRUE(dd1.IsEqual(DerivedDerived{"b1", "d1", "dd1"}));
    ASSERT_FALSE(dd1.IsEqual(DerivedDerived{"b1", "d1", "dd2"}));
}

```

- 参照 [NVI\(non virtual interface\)](#)
- [解答例-NVI](#)

演習-RAIIの効果

- 問題
RAIIにはどのような効果があるか？ 下記より相応しいものを選択せよ。
- 選択肢
 1. リソースリークの防止
 2. 使用リソースの削減
 3. クラスの実装を隠す
 4. LIPの順守
- [解答-RAIIの効果](#)

演習-RAII

- 問題

```

// @@@ exercise/design_pattern_q/raii.cpp 5
// [Q]
// 下記クラスBase、Derivedはクローンパターンをしているが、Clone関数はnewしたオブジェクトであるため、
// メモリリークを起こしやすい。std::unique_ptrを使用してこの問題に対処せよ。

class Base {
public:
    explicit Base(std::string name) : name1_{std::move(name)} {}
    virtual ~Base() = default;

```

```

virtual std::string const& GetName() const noexcept { return name1_; }

virtual Base* Clone() const { return new Base(name1_); }

Base(Base const&) = delete;
Base& operator=(Base const&) = delete;

private:
    std::string name1_;
};

class Derived final : public Base {
public:
    explicit Derived(std::string name1 = "", std::string name2 = "")
        : Base{std::move(name1)}, name2_{std::move(name2)}
    {
    }
    virtual ~Derived() override = default;
    virtual std::string const& GetName() const noexcept override { return name2_; }

    virtual Derived* Clone() const override { return new Derived{Base::GetName(), name2_}; }

private:
    std::string name2_;
};

TEST(DesignPatternQ, RAI)
{
    Derived d1{"name1", "name2"};
    Derived* d2{d1.Clone()};

    ASSERT_EQ("name1", d2->Base::GetName());
    ASSERT_EQ("name2", d2->GetName());

    delete d2;

    Base* b3 = d1.Clone();

    ASSERT_EQ("name1", b3->Base::GetName());
    ASSERT_EQ("name2", b3->GetName());

    delete b3;
}

```

- 参照 [RAII\(scoped_guard\)](#)
- [解答例-RAII](#)

演習-Future

- 問題

```

// @@@ exercise/design_pattern_q/future.cpp 24
// [Q]
// 下記のfind_files_concurrentlyはスレッドの出力の結果をキャプチャリファレンスで受け取るため、
// 入出力の関係が明確でない。Futureパターンを使用しそれを明確にするリファクタリングを行え。

std::vector<std::string> find_files_concurrently()
{
    auto pca = std::vector<std::string>{};
    auto pcq = std::vector<std::string>{};

    auto th0 = std::thread{[&pca] { pca = find_files("../programming_convention_a/"); }};
    auto th1 = std::thread{[&pcq] { pcq = find_files("../programming_convention_q/"); }};

    th0.join();
    th1.join();

    pca.insert(pca.end(), pcq.begin(), pcq.end());

    return pca;
}

TEST(DesignPatternQ, Future)
{
    auto files = find_files_concurrently();

    ASSERT_GT(files.size(), 10);
}

```

- 参照 Future
 - 解答例-Future

演習-DI

- ## • 問題

```
    ASSERT_EQ(exp, act);
}
```

- 参照 [DI\(dependency injection\)](#)
- [解答例-DI](#)

演習-Singleton

- 問題

```
// @@@ exercise/design_pattern_q/singleton.cpp 5
// [Q]
// 下記 AppConfig はアプリケーション全体の設定を管理するためのクラスである。
// 目的上、そのインスタンス AppConfig は広域のアクセスが必要であり、
// グローバルインスタンスとして実装している。
// グローバルインスタンスは、初期化の順番が標準化されておらず、
// 多くの処理系ではリンクの順番に依存しているため、
// アプリケーション立ち上げ時に様々な問題を起こすことがある。
// こういった問題を回避するため、AppConfig を Singleton 化せよ。
// また他の問題があれば合わせて修正せよ。

class AppConfig {
public:
    enum BaseColor { Red, Green, Black };

    void SetBaseColor(BaseColor color) noexcept { color_ = color; }
    BaseColor GetBaseColor() { return color_; }

    void SetUserName(std::string_view username) { username_ = username; }
    std::string& GetUserName() { return username_; }

    void Logging(bool is_logging) { is_logging_ = is_logging; }
    bool IsLoggin() { return is_logging_; }

    // 他の設定値は省略

    void SetDefault()
    {
        SetBaseColor(Red);
        SetUserName("No Name");
        Logging(false);
    }

private:
    BaseColor color_{Red};
    std::string username_{ "No Name" };
    bool is_logging_{false};
};

AppConfig AppConfig;

class DesignPatternQ_F : public ::testing::Test {
protected:
    virtual void SetUp() override { AppConfig.SetDefault(); }

    virtual void TearDown() override { AppConfig.SetDefault(); }
};

TEST_F(DesignPatternQ_F, Singleton)
{
    ASSERT_EQ(AppConfig::Red, AppConfig.GetBaseColor());
    ASSERT_EQ("No Name", AppConfig.GetUserName());
    ASSERT_FALSE(AppConfig.IsLoggin());

    AppConfig.SetBaseColor(AppConfig::Green);
    ASSERT_EQ(AppConfig::Green, AppConfig.GetBaseColor());

    AppConfig.SetUserName("Stroustrup");
    ASSERT_EQ("Stroustrup", AppConfig.GetUserName());

    AppConfig.Logging(true);
    ASSERT_TRUE(AppConfig.IsLoggin());

    AppConfig.SetDefault();
    ASSERT_EQ(AppConfig::Red, AppConfig.GetBaseColor());
    ASSERT_EQ("No Name", AppConfig.GetUserName());
    ASSERT_FALSE(AppConfig.IsLoggin());
}
```

- 参照 [Singleton](#)
- 解答例-Singleton

演習-State

- 問題

```
// @@@ exercise/design_pattern_q/state.cpp 5
// [Q]
// 下記クラスGreetingにはlang_に対する同型のswitch文が3個ある。
// これは機能追加時にバグが混入しやすいアンチパターンであるため、
// Stateパターンを用いリファクタリングせよ。
// また、他の問題があれば合わせて修正せよ。

enum class Language { English, Japanese, French };

class Greeting {
public:
    explicit Greeting(Language lang = Language::English) noexcept : lang_{lang} {}
    void SetLanguage(Language lang) noexcept { lang_ = lang; }

    std::string GoodMorning()
    {
        switch (lang_) {
        case Language::Japanese:
            return u8"おはよう";
        case Language::French:
            return u8"Bonjour";
        case Language::English:
        default:
            return u8"good morning";
        }
    }

    std::string Hello()
    {
        switch (lang_) {
        case Language::Japanese:
            return u8"こんにちは";
        case Language::French:
            return u8"Bonjour";
        case Language::English:
        default:
            return u8"hello";
        }
    }

    std::string GoodEvening()
    {
        switch (lang_) {
        case Language::Japanese:
            return u8"こんばんは";
        case Language::French:
            return u8"bonne soirée";
        case Language::English:
        default:
            return u8"good evening";
        }
    }
}

private:
    Language lang_;
};

TEST(DesignPatternQ, State)
{
    auto greeting = Greeting{};

    ASSERT_EQ(u8"good morning", greeting.GoodMorning());
    ASSERT_EQ(u8"hello", greeting.Hello());
    ASSERT_EQ(u8"good evening", greeting.GoodEvening());

    greeting.SetLanguage(Language::Japanese);
    ASSERT_EQ(u8"おはよう", greeting.GoodMorning());
    ASSERT_EQ(u8"こんにちは", greeting.Hello());
    ASSERT_EQ(u8"こんばんは", greeting.GoodEvening());

    greeting.SetLanguage(Language::French);
}
```

```

ASSERT_EQ(u8"Bonjour", greeting.GoodMorning());
ASSERT_EQ(u8"Bonjoun", greeting.Hello());
ASSERT_EQ(u8"bonne soirée", greeting.GoodEvening());

greeting.SetLanguage(Language::English);
ASSERT_EQ(u8"good morning", greeting.GoodMorning());
ASSERT_EQ(u8"hello", greeting.Hello());
ASSERT_EQ(u8"good evening", greeting.GoodEvening());
}

```

- 参照 [State](#)
- [解答例-State](#)

演習-Null Object

- 問題

```

// @@@ exercise/design_pattern_q/null_object.cpp 38
// [Q]
// 下記クラスPersonにはgreeting_のヌルチェックを行う三項演算子が3つある。
// これはヌルポインタアクセスを起こしやすいアンチパターンであるため、
// Null Objectパターンを用いリファクタリングせよ。
// また、他の問題があれば合わせて修正せよ。

class Greeting {
public:
    explicit Greeting(Language lang = Language::English) : state_{new_state(lang)} {}
    void SetLanguage(Language lang) { state_ = new_state(lang); }

    std::string GoodMorning() const { return state_->GoodMorning(); }
    std::string Hello() const { return state_->Hello(); }
    std::string GoodEvening() const { return state_->GoodEvening(); }

private:
    std::unique_ptr<GreetingState> state_;

    static std::unique_ptr<GreetingState> new_state(Language lang)
    {
        switch (lang) {
        case Language::Japanese:
            return std::make_unique<GreetingState_Japanese>();
        case Language::French:
            return std::make_unique<GreetingState_French>();
        case Language::English:
        default:
            return std::make_unique<GreetingState_English>();
        }
    }
};

class Person {
public:
    explicit Person(Language lang, bool silent = false)
        : greeting_{silent ? std::unique_ptr<Greeting>{} : std::make_unique<Greeting>(lang)}
    {}

    std::string GoodMorning() { return greeting_ ? greeting_->GoodMorning() : ""; }
    std::string Hello() { return greeting_ ? greeting_->Hello() : ""; }
    std::string GoodEvening() { return greeting_ ? greeting_->GoodEvening() : ""; }

private:
    std::unique_ptr<Greeting> greeting_;
};

TEST(DesignPatternQ, NullObject)
{
    auto e = Person{Language::English};

    ASSERT_EQ(u8"good morning", e.GoodMorning());
    ASSERT_EQ(u8"hello", e.Hello());
    ASSERT_EQ(u8"good evening", e.GoodEvening());

    auto j = Person{Language::Japanese};
    ASSERT_EQ(u8"おはよう", j.GoodMorning());
    ASSERT_EQ(u8"こんにちは", j.Hello());
    ASSERT_EQ(u8"こんばんは", j.GoodEvening());

    auto f = Person{Language::French};

```

```

ASSERT_EQ(u8"Bonjour", f.GoodMorning());
ASSERT_EQ(u8"Bonjoun", f.Hello());
ASSERT_EQ(u8"bonne soirée", f.GoodEvening());

auto e_s = Person{Language::English, true};

ASSERT_EQ(u8"", e_s.GoodMorning());
ASSERT_EQ(u8"", e_s.Hello());
ASSERT_EQ(u8"", e_s.GoodEvening());
}

```

- 参照 [Null Object](#)
- [解答例-Null Object](#)

演習-Templatedメソッド

- 問題

```

// @@@ exercise/design_pattern_q/template_method.cpp 5
// [Q]
// 下記クラスXxxDataFormatterXml、XxxDataFormatterCsvは同様の処理を行い、
// それぞれのフォーマットで文字列を出力する。このような処理のクローンはTemplate Method
// パターンにより排除できる。
// このパターンを用い、下記2クラスをリファクタリングせよ。
// また、他の問題があれば合わせて修正せよ。

struct XxxData {
    int a;
    int b;
    int c;
};

class XxxDataFormatterXml {
public:
    XxxDataFormatterXml() = default;

    std::string ToString(XxxData const& xxx_data) const
    {
        auto body = std::string("<Item>\n");

        body += "    <XxxData a=\"" + std::to_string(xxx_data.a) + "\">\n";
        body += "    <XxxData b=\"" + std::to_string(xxx_data.b) + "\">\n";
        body += "    <XxxData c=\"" + std::to_string(xxx_data.c) + "\">\n";
        body += "</Item>\n";

        return header_ + body + footer_;
    }

    std::string ToString(std::vector<XxxData> const& xxx_datas) const
    {
        auto ret = std::string(header_);

        for (auto const& xxx_data : xxx_datas) {
            ret += "<Item>\n";
            ret += "    <XxxData a=\"" + std::to_string(xxx_data.a) + "\">\n";
            ret += "    <XxxData b=\"" + std::to_string(xxx_data.b) + "\">\n";
            ret += "    <XxxData c=\"" + std::to_string(xxx_data.c) + "\">\n";
            ret += "</Item>\n";
        }

        return ret + footer_;
    }

private:
    std::string header_ = "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n<XxxDataFormatterXml>\n";
    std::string footer_ = "</XxxDataFormatterXml>\n";
};

class XxxDataFormatterCsv {
public:
    XxxDataFormatterCsv() = default;

    std::string ToString(XxxData const& xxx_data) const
    {
        auto body = std::string(std::to_string(xxx_data.a) + ", " + std::to_string(xxx_data.b)
                               + ", " + std::to_string(xxx_data.b) + "\n");

        return header_ + body;
    }
}

```

```

        std::string ToString(std::vector<XxxData> const& xxx_datas) const
    {
        auto ret = std::string{header_};

        for (auto const& xxx_data : xxx_datas) {
            ret += std::string{std::to_string(xxx_data.a) + ", " + std::to_string(xxx_data.b) + ", "
                + std::to_string(xxx_data.b) + "\n"};
        }

        return ret;
    }

private:
    std::string const header_ = "a, b, c\n";
};

TEST(DesignPatternQ, TemplateMethod)
{
    auto xml = XxxDataFormatterXml{};
    {
        auto const expect_scalar = std::string{
            "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n"
            "<XxxDataFormatterXml>\n"
            "<Item>\n"
            "    <XxxData a=\"1\"\n"
            "    <XxxData b=\"100\"\n"
            "    <XxxData c=\"10\"\n"
            "</Item>\n"
            "</XxxDataFormatterXml>\n"};
        auto const actual_scalar = xml.ToString({1, 100, 10});
        ASSERT_EQ(expect_scalar, actual_scalar);

        auto const expect_array = std::string{
            "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n"
            "<XxxDataFormatterXml>\n"
            "<Item>\n"
            "    <XxxData a=\"1\"\n"
            "    <XxxData b=\"100\"\n"
            "    <XxxData c=\"10\"\n"
            "</Item>\n"
            "<Item>\n"
            "    <XxxData a=\"2\"\n"
            "    <XxxData b=\"200\"\n"
            "    <XxxData c=\"20\"\n"
            "</Item>\n"
            "</XxxDataFormatterXml>\n"};
        auto const actual_array = xml.ToString({{1, 100, 10}, {2, 200, 20}});
        ASSERT_EQ(expect_array, actual_array);
    }

    auto csv = XxxDataFormatterCsv{};
    {
        auto expect_scalar = std::string{
            "a, b, c\n"
            "1, 100, 100\n"};
        auto const actual_scalar = csv.ToString({1, 100, 10});
        ASSERT_EQ(expect_scalar, actual_scalar);

        auto const expect_array = std::string{
            "a, b, c\n"
            "1, 100, 100\n"
            "2, 200, 200\n"};
        auto const actual_array = csv.ToString({{1, 100, 10}, {2, 200, 20}});
        ASSERT_EQ(expect_array, actual_array);
    }
}

```

- 参照 [Templateメソッド](#)
- [解答例-Templateメソッド](#)

演習-Factory

- 問題

```

// @@@ exercise/design_pattern_q/factory_lib.h 6
// [Q]
// 下記クラスXxxDataFormatterXml、XxxDataFormatterCsvはヘッダファイルで宣言・定義を行ったために
// 他の.cppファイルから直接アクセスできてしまう。

```

```

// Factoryパターンを用いて、XxxDataFormatterXml、XxxDataFormatterCsvを他の.cppファイルから
// 直接アクセスできないようにせよ。

struct XxxData {
    int a;
    int b;
    int c;
};

class XxxDataFormatterIF {
public:
    XxxDataFormatterIF() noexcept = default;
    virtual ~XxxDataFormatterIF() = default;
    XxxDataFormatterIF(XxxDataFormatterIF const&) = delete;
    XxxDataFormatterIF& operator=(XxxDataFormatterIF const&) = delete;

    std::string ToString(XxxData const& xxx_data) const
    {
        return header() + body(xxx_data) + footer();
    }

    std::string ToString(std::vector<XxxData> const& xxx_datas) const
    {
        auto ret = header();

        for (auto const& xxx_data : xxx_datas) {
            ret += body(xxx_data);
        }

        return ret + footer();
    }

private:
    virtual std::string const& header() const = 0;
    virtual std::string const& footer() const = 0;
    virtual std::string body(XxxData const& xxx_data) const = 0;
};

class XxxDataFormatterXml final : public XxxDataFormatterIF {
public:
    XxxDataFormatterXml() = default;
    virtual ~XxxDataFormatterXml() override = default;

private:
    virtual std::string const& header() const override;
    virtual std::string const& footer() const override;
    virtual std::string body(XxxData const& xxx_data) const override;
};

class XxxDataFormatterCsv final : public XxxDataFormatterIF {
public:
    XxxDataFormatterCsv() = default;
    virtual ~XxxDataFormatterCsv() override = default;

private:
    virtual std::string const& header() const override;
    virtual std::string const& footer() const override;
    virtual std::string body(XxxData const& xxx_data) const override;
};

```

```

// @@@ exercise/design_pattern_q/factory_lib.cpp 5

std::string const& XxxDataFormatterXml::header() const
{
    static auto const header
        = std::string("<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n<XxxDataFormatterXml>\n");

    return header;
}

std::string const& XxxDataFormatterXml::footer() const
{
    static auto const footer = std::string("</XxxDataFormatterXml>\n");

    return footer;
}

std::string XxxDataFormatterXml::body(XxxData const& xxx_data) const
{
    auto content = std::string("<Item>\n");

```

```

        content += "    <XxxData a=\"" + std::to_string(xxx_data.a) + "\">\n";
        content += "    <XxxData b=\"" + std::to_string(xxx_data.b) + "\">\n";
        content += "    <XxxData c=\"" + std::to_string(xxx_data.c) + "\">\n";

    return content + "</Item>\n";
}

std::string const& XxxDataFormatterCsv::header() const
{
    static auto const header = std::string{"a, b, c\n"};
    return header;
}

std::string const& XxxDataFormatterCsv::footer() const
{
    static auto const footer = std::string{};

    return footer;
}

std::string XxxDataFormatterCsv::body(XxxData const& xxx_data) const
{
    return std::string{std::to_string(xxx_data.a) + ", " + std::to_string(xxx_data.b) + ", "
                      + std::to_string(xxx_data.b) + "\n"};
}

```

```

// @@@ exercise/design_pattern_q/factory.cpp 9

TEST(DesignPatternQ, Factory)
{
    auto xml = XxxDataFormatterXml{};
    {
        auto const expect_scalar = std::string{
            "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n"
            "<XxxDataFormatterXml>\n"
            "<Item>\n"
            "    <XxxData a=\"1\"\n"
            "    <XxxData b=\"100\"\n"
            "    <XxxData c=\"10\"\n"
            "</Item>\n"
            "</XxxDataFormatterXml>\n"};
        auto const actual_scalar = xml.ToString({1, 100, 10});
        ASSERT_EQ(expect_scalar, actual_scalar);
    }

    auto csv = XxxDataFormatterCsv{};
    {
        auto const expect_array = std::string{
            "a, b, c\n"
            "1, 100, 100\n"
            "2, 200, 200\n"};
        auto const actual_array = csv.ToString({{1, 100, 10}, {2, 200, 20}});
        ASSERT_EQ(expect_array, actual_array);
    }
}

```

- 参照 [Factory](#)
- [解答例-Factory](#)

演習-Named Constructor

- 問題

```

// @@@ exercise/design_pattern_q/named_constructor_lib.h 14
// [Q]
// 下記関数XxxDataFormatterFactoryはインターフェースクラスXxxDataFormatterIFのファクトリ関数
// である。これをnamed constructorパターンで実装しなおせ。

class XxxDataFormatterIF {
public:
    XxxDataFormatterIF() noexcept = default;
    virtual ~XxxDataFormatterIF() = default;
    XxxDataFormatterIF(XxxDataFormatterIF const&) = delete;
    XxxDataFormatterIF& operator=(XxxDataFormatterIF const&) = delete;

    std::string ToString(XxxData const& xxx_data) const
    {

```

```

        return header() + body(xxx_data) + footer();
    }

    std::string ToString(std::vector<XxxData> const& xxx_datas) const
    {
        auto ret = header();

        for (auto const& xxx_data : xxx_datas) {
            ret += body(xxx_data);
        }

        return ret + footer();
    }

private:
    virtual std::string const& header() const = 0;
    virtual std::string const& footer() const = 0;
    virtual std::string body(XxxData const& xxx_data) const = 0;
};

enum class XxxDataFormatterType { Xml, Csv, Table };

XxxDataFormatterIF const& XxxDataFormatterFactory(XxxDataFormatterType type) noexcept;

```

// @@@ exercise/design_pattern_q/named_constructor.lib.cpp 114

```

XxxDataFormatterIF const& XxxDataFormatterFactory(XxxDataFormatterType type) noexcept
{
    static auto const xml = XxxDataFormatterXml{};
    static auto const csv = XxxDataFormatterCsv{};
    static auto const table = XxxDataFormatterTable{};

    switch (type) {
    case XxxDataFormatterType::Xml:
        return xml;
    case XxxDataFormatterType::Csv:
        return csv;
    case XxxDataFormatterType::Table:
        return table;
    default:
        assert("unknown type");
        return csv;
    }
}

```

// @@@ exercise/design_pattern_q/named_constructor.cpp 9

```

TEST(DesignPatternQ, NamedConstructor)
{
    auto const& xml = XxxDataFormatterFactory(XxxDataFormatterType::Xml);
    {
        auto const expect_scalar = std::string{
            "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n"
            "<XxxDataFormatterXml>\n"
            "  <Item>\n"
            "    <XxxData a=\"1\">\n"
            "    <XxxData b=\"100\">\n"
            "    <XxxData c=\"10\">\n"
            "  </Item>\n"
            "</XxxDataFormatterXml>\n"};
        auto const actual_scalar = xml.ToString({1, 100, 10});

        ASSERT_EQ(expect_scalar, actual_scalar);
    }

    auto const& csv = XxxDataFormatterFactory(XxxDataFormatterType::Csv);
    {
        auto const expect_array = std::string{
            "a, b, c\n"
            "1, 100, 100\n"
            "2, 200, 200\n"};
        auto const actual_array = csv.ToString({{1, 100, 10}, {2, 200, 20}});

        ASSERT_EQ(expect_array, actual_array);
    }

    auto const& table = XxxDataFormatterFactory(XxxDataFormatterType::Table);
    {
        auto const expect_array = std::string{
            "+-----+-----+-----+\n"
            "| a     | b     | c     |\n"
            "+-----+-----+-----+"};

```

```

    "-----+-----+-----+\n"
    "| 3    | 300   | 30    |\n"
    "-----+-----+-----+\n"
    "| 4    | 400   | 40    |\n"
    "-----+-----+-----+\n";
auto const actual_array = table.ToString({{3, 300, 30}, {4, 400, 40}});

ASSERT_EQ(expect_array, actual_array);
}
}

```

- 参照 [Named Constructor](#)
- [解答例-Named Constructor](#)

演習-Proxy

- 問題

```

// @@@ exercise/design_pattern_q/proxy.cpp 7
// [Q]
// 下記クラスLsDirのFileListはlsコマンドをpopenにより実行し、その戻り値をstd::stringで返す。
// popenはコストの高いコールなので、パフォーマンスを上げるためにlsの戻り値をキャッシュしたいが、
// 現行のLsDirも必要である。
// Proxyパターンを使い、この問題に対処するためのLsDirCachedを作れ。

class LsDir {
public:
    LsDir() = default;
    ~LsDir() = default;

    void SetArgs(std::string_view args) { args_ = args; }
    std::string const& GetArgs() const noexcept { return args_; }

    std::string fileList() const
    {
        auto cmd      = std::string("ls ") + GetArgs();
        auto to_close = [](FILE* f) { fclose(f); };
        auto stream = std::unique_ptr<FILE, decltype(to_close)>{popen(cmd.c_str(), "r"), to_close};

        auto files = std::string{};
        char buff[256];

        while (fgets(buff, sizeof(buff) - 1, stream.get()) != NULL) {
            files += buff;
        }

        return files;
    }

private:
    std::string args_{};
};

TEST(DesignPatternQ, Proxy)
{
    auto ld = LsDir{};

    {
        ld.SetArgs("../ut_data/");
        auto exp = std::string{"a.cpp\na.h\nabc.cpp\nabc.h\nnd\nefghij.cpp\nefghij.h\nlib\no\n"};
        auto act = ld.fileList();

        ASSERT_EQ(exp, act);
        ASSERT_EQ(act, ld.fileList());
    }
    {
        ld.SetArgs("../ut_data/lib/");

        auto exp = std::string{"lib.cpp\nlib.h\n"};
        auto act = ld.fileList();

        ASSERT_EQ(exp, act);
        ASSERT_EQ(act, ld.fileList());
    }
}

```

- 参照 [Proxy](#)

- 解答例-Proxy

演習-Strategy

- 問題

```
// @@@ exercise/design_pattern_q/strategy.cpp 11
// [Q]
// 下記find_filesは醜悪であるだけでなく、拡張性もない。
// Strategyパターンを用い、この問題に対処せよ。

enum class FindCondition {
    File,
    Dir,
    FileCpp,
};

std::vector<std::string> find_files(std::string const& path, FindCondition condition)
{
    namespace fs = std::filesystem;

    auto files = std::vector<fs::path>{};

    // recursive_directory_iteratorはファイルシステム依存するため、その依存を排除する他の処理
    std::copy(fs::recursive_directory_iterator{path}, fs::recursive_directory_iterator{},
              std::back_inserter(files));

    std::sort(files.begin(), files.end());

    auto ret = std::vector<std::string>{};

    for (fs::path const& p : files) {
        auto is_match = false;

        switch (condition) {
            case FindCondition::File:
                if (fs::is_regular_file(p)) {
                    is_match = true;
                }
                break;
            case FindCondition::Dir:
                if (fs::is_directory(p)) {
                    is_match = true;
                }
                break;
            case FindCondition::FileCpp:
                auto const filename = p.filename().generic_string();
                auto const cpp_file = std::string{".cpp"};

                if (filename.length() > cpp_file.length()
                    && (filename.substr(filename.length() - cpp_file.length()) == cpp_file)) {
                    is_match = true;
                }
                break;
            default:
                assert(false);
        }

        if (is_match) {
            ret.emplace_back(p.generic_string());
        }
    }
}

return ret;
}

TEST(DesignPatternQ, Strategy)
{
    auto sort = [](&auto& v) {
        std::sort(v.begin(), v.end());
        return v;
    };

    {
        auto exp = sort(std::vector<std::string>{
            "../ut_data/a.cpp", "../ut_data/a.h", "../ut_data/abc.cpp", "../ut_data/abc.h",
            "../ut_data/d/a.d", "../ut_data/efghij.cpp", "../ut_data/efghij.h",
            "../ut_data/lib/lib.cpp", "../ut_data/lib/lib.h", "../ut_data/o/a.o"});
    }
}
```

```

        auto act = find_files("../ut_data", FindCondition::File);
        ASSERT_EQ(exp, act);
    }
    {
        auto exp = sort(std::vector<std::string>{"../ut_data/d", "../ut_data/lib", "../ut_data/o"});
        auto act = find_files("../ut_data", FindCondition::Dir);
        ASSERT_EQ(exp, act);
    }
    {
        auto exp
            = sort(std::vector<std::string>{"../ut_data/a.cpp", "../ut_data/abc.cpp",
                "../ut_data/efghij.cpp", "../ut_data/lib/lib.cpp"});
        auto act = find_files("../ut_data", FindCondition::FileCpp);
        ASSERT_EQ(exp, act);
    }
}

```

- 参照 [Strategy](#)
- [解答例-Strategy](#)

演習-Visitor

- 問題

```

// @@@ exercise/design_pattern_q/visitor.cpp 9
// [Q]
// 下記クラスFile、Dir、OtherEntityはクラスFileEntityから派生し、
// それぞれが自身をstd::stringに変換するアルゴリズム関数
//     * to_string_normal()
//     * to_string_with_char()
//     * to_string_with_children()
// をオーバーライドしている。これはポリモーフィズムの使用方法としては正しいが、
// to_string_xxx系のインターフェースが大量に増えた場合に、
// FileEntityのインターフェースがそれに比例して増えてしまう問題を持っている。
// Visitorパターンを使用しこれに対処せよ。

class FileEntity {
public:
    explicit FileEntity(std::string const& pathname) : pathname_{strip(pathname)} {}
    virtual ~FileEntity() = default;
    std::string const& Pathname() const noexcept { return pathname_; }

    std::string ToStringNormal() const { return to_string_normal(); }
    std::string ToStringWithChar() const { return to_string_with_char(); }
    std::string ToStringWithChildren() const { return to_string_with_children(); }

private:
    std::string const pathname_;

    virtual std::string to_string_normal() const      = 0;
    virtual std::string to_string_with_char() const   = 0;
    virtual std::string to_string_with_children() const = 0;

    static std::string strip(std::string const& pathname)
    {
        return std::regex_replace(pathname, std::regex{R"(^/+)"}, "");
    }
};

class File final : public FileEntity {
public:
    explicit File(std::string const& pathname) : FileEntity{pathname} {}

private:
    virtual std::string to_string_normal() const override { return Pathname(); }
    virtual std::string to_string_with_char() const override { return Pathname(); }
    virtual std::string to_string_with_children() const override { return Pathname(); }
};

class Dir final : public FileEntity {
public:
    explicit Dir(std::string const& pathname) : FileEntity{pathname} {}

private:
    virtual std::string to_string_normal() const override { return Pathname() + '/'; }
    virtual std::string to_string_with_char() const override { return to_string_normal(); }
};

```

```

virtual std::string to_string_with_children() const override { return find_files(Pathname()); };

static std::string find_files(std::string const& dir)
{
    namespace fs = std::filesystem;

    auto files = std::vector<std::string>{};

    std::for_each(fs::recursive_directory_iterator{dir}, fs::recursive_directory_iterator{},
                 [&files](fs::path const& p) { files.emplace_back(p.generic_string()); });

    std::sort(files.begin(), files.end());

    auto ret = std::string{dir};

    for (auto f : files) {
        ret += ' ' + f;
    }

    return ret;
}

class OtherEntity final : public FileEntity {
public:
    explicit OtherEntity(std::string const& pathname) : FileEntity{pathname} {}

private:
    virtual std::string to_string_normal() const override { return Pathname(); }
    virtual std::string to_string_with_char() const override { return Pathname() + '+'; }
    virtual std::string to_string_with_children() const override { return Pathname(); };
};

TEST(DesignPatternQ, Visitor)
{
    auto const f0 = File{"../ut_data/a.cpp"};
    auto const f1 = File{"../ut_data/a.cpp///"};

    ASSERT_EQ("../ut_data/a.cpp", f0.Pathname());
    ASSERT_EQ("../ut_data/a.cpp", f0.Pathname());
    ASSERT_EQ("../ut_data/a.cpp", f0.ToStringNormal());
    ASSERT_EQ("../ut_data/a.cpp", f0.ToStringWithChar());
    ASSERT_EQ("../ut_data/a.cpp", f0.ToStringWithChildren());

    auto const dir = Dir{"../ut_data/lib/"};

    ASSERT_EQ("../ut_data/lib", dir.Pathname());
    ASSERT_EQ("../ut_data/lib/", dir.ToStringNormal());
    ASSERT_EQ("../ut_data/lib/", dir.ToStringWithChar());
    ASSERT_EQ("../ut_data/lib ../ut_data/lib/lib.cpp ../ut_data/lib/lib.h",
              dir.ToStringWithChildren());

    auto const other = OtherEntity{"symbolic_link"};

    ASSERT_EQ("symbolic_link", other.Pathname());
    ASSERT_EQ("symbolic_link", other.ToStringNormal());
    ASSERT_EQ("symbolic_link+", other.ToStringWithChar());
    ASSERT_EQ("symbolic_link", other.ToStringWithChildren());
}

```

- 参照 [Visitor](#)
- [解答例-Visitor](#)

演習-CRTP

- 問題

```

// @@@ exercise/design_pattern_q/crtp.cpp 9
// [Q]
// 下記クラスFileEntityから派生しクラスFile、Dir、OtherEntityは、
// Visitorパターンを利用しているため、そのすべてで下記のコードクローンを持つ。
//
//     virtual std::string to_string(Visitor const& to_s) const { return to_s.Visit(*this); }
//
// このコードクローンのthisの型は、それぞれFile、Dir、OtherEntityとなるため、
// この関数をFileEntityで定義すると動作が変わってしまい、単純には統一できない。
// CRTPを用い、このクローンを削除せよ。
//
class Visitor;

```

```

class FileEntity {
public:
    explicit FileEntity(std::string const& pathname) : pathname_{strip(pathname)} {}
    virtual ~FileEntity() = default;
    std::string const& Pathname() const { return pathname_; }
    std::string ToString(Visitor const& to_s) const { return to_string(to_s); }

private:
    std::string const pathname_;

    virtual std::string to_string(Visitor const& to_s) const = 0;
    static std::string strip(std::string const& pathname)
    {
        return std::regex_replace(pathname, std::regex{R"(/+$)"}, "");
    }
};

class File;
class Dir;
class OtherEntity;

class Visitor {
public:
    virtual ~Visitor() = default;
    std::string Visit(File const& file) const { return visit(file); }
    std::string Visit(Dir const& dir) const { return visit(dir); }
    std::string Visit(OtherEntity const& other) const { return visit(other); }

private:
    virtual std::string visit(File const& file) const = 0;
    virtual std::string visit(Dir const& dir) const = 0;
    virtual std::string visit(OtherEntity const& f) const = 0;
};

class File final : public FileEntity {
public:
    explicit File(std::string const& pathname) : FileEntity{pathname} {}

private:
    virtual std::string to_string(Visitor const& to_s) const { return to_s.Visit(*this); }
};

class Dir final : public FileEntity {
public:
    explicit Dir(std::string const& pathname) : FileEntity{pathname} {}

private:
    virtual std::string to_string(Visitor const& to_s) const { return to_s.Visit(*this); }
};

class OtherEntity final : public FileEntity {
public:
    explicit OtherEntity(std::string const& pathname) : FileEntity{pathname} {}

private:
    virtual std::string to_string(Visitor const& to_s) const { return to_s.Visit(*this); }
};

class ToStringNormal : public Visitor {
private:
    virtual std::string visit(File const& file) const override { return file.Pathname(); }
    virtual std::string visit(Dir const& dir) const override { return dir.Pathname() + '/'; }
    virtual std::string visit(OtherEntity const& other) const override { return other.Pathname(); }
};

class ToStringWithChar : public Visitor {
private:
    virtual std::string visit(File const& file) const override { return file.Pathname(); }
    virtual std::string visit(Dir const& dir) const override { return dir.Pathname() + '/'; }
    virtual std::string visit(OtherEntity const& other) const override
    {
        return other.Pathname() + '+';
    }
};

class ToStringWithChildren : public Visitor {
private:
    virtual std::string visit(File const& file) const override { return file.Pathname(); }
    virtual std::string visit(Dir const& dir) const override { return find_files(dir.Pathname()); }
};

```

```

virtual std::string visit(OtherEntity const& other) const override { return other.Pathname(); }

static std::string find_files(std::string const& dir)
{
    namespace fs = std::filesystem;

    auto files = std::vector<std::string>{};

    std::for_each(fs::recursive_directory_iterator{dir}, fs::recursive_directory_iterator{},
                 [&files](fs::path const& p) { files.emplace_back(p.generic_string()); });

    std::sort(files.begin(), files.end());

    auto ret = std::string{dir};

    for (auto f : files) {
        ret += ' ' + f;
    }

    return ret;
}
};

TEST(DesignPatternQ, CRTP)
{
    auto ts_normal = ToStringNormal{};
    auto ts_char = ToStringWithChar{};
    auto ts_children = ToStringWithChildren{};

    auto const f0 = File{"../ut_data/a.cpp"};
    auto const f1 = File{"../ut_data/a.cpp///"};

    ASSERT_EQ("../ut_data/a.cpp", f0.Pathname());
    ASSERT_EQ("../ut_data/a.cpp", f0.Pathname());

    ASSERT_EQ("../ut_data/a.cpp", f0.ToString(ts_normal));
    ASSERT_EQ("../ut_data/a.cpp", f0.ToString(ts_char));
    ASSERT_EQ("../ut_data/a.cpp", f0.ToString(ts_children));

    auto const dir = Dir{"../ut_data/lib/"};

    ASSERT_EQ("../ut_data/lib", dir.Pathname());
    ASSERT_EQ("../ut_data/lib/", dir.ToString(ts_normal));
    ASSERT_EQ("../ut_data/lib/", dir.ToString(ts_char));
    ASSERT_EQ("../ut_data/lib ../ut_data/lib/lib.cpp ../ut_data/lib/lib.h",
              dir.ToString(ts_children));

    auto const other = OtherEntity{"symbolic_link"};

    ASSERT_EQ("symbolic_link", other.Pathname());
    ASSERT_EQ("symbolic_link", other.ToString(ts_normal));
    ASSERT_EQ("symbolic_link+", other.ToString(ts_char));
    ASSERT_EQ("symbolic_link", other.ToString(ts_children));
}
}

```

- 参照 [CRTP\(curiously recurring template pattern\)](#).
- 解答例-CRTP

演習-Observer

- 問題

```

// @@@ exercise/design_pattern_q/observer.cpp 7
// [Q]
// 下記クラスはそれぞれが
//   * ViewX, ViewY : GUIへの出力(描画)
//   * Model        : 何らかのビジネスロジックの演算
//   * Controller    : OKボタンクリックイベントをModelへ通知
// 行うことを探している。
// 依存関係Model->ViewX, ViewYはMVCに逆行しているため下記のような問題を持つ。
//   * ViewX, ViewYの変更がModelに伝搬してしまう。
//   * この例は単純であるためViewX, ViewY->Modelへの依存関係は存在しないが、
//     実際のアプリケーションではそのような依存関係が存在するため、依存関係が循環してしまう。
//   * ModelがダイレクトにViewX, ViewYへ出力するため、単体テストの実施は困難である。
//   * この依存関係が直接の原因ではないが、このような依存関係を持つアプリケーションのクラスは
//     巨大になる。
// アプリケーションが小規模である時には、このような問題がバグや開発効率悪化の原因となることは稀
// であり放置されることが多いが、大規模化に伴いこのような潜在的問題が表出する。
// ModelにObserverパターンを適用する等をしてこの問題に対処するとともに、Modelの単体テストを行え。

```

```

class ViewX {
public:
    void DisplaySomething(std::string const&) noexcept {}
};

ViewX g_ViewX;

class ViewY {
public:
    void DisplaySomething(std::string const&) noexcept {}
};

ViewY g_ViewY;

class Model {
public:
    Model() = default;
    ~Model() { wait_future(); }

    void DoSomething()
    {
        wait_future();

        future_ = std::async(std::launch::async, [] {
            // 本来は非同期処理が必要な重い処理
            auto result = std::string("result of doing something");

            g_ViewX.DisplaySomething(result);
            g_ViewY.DisplaySomething(result);
        });
    }
}

private:
    std::future<void> future_;

    void wait_future() noexcept
    {
        if (future_.valid()) {
            future_.wait();
        }
    }
};

class Controller {
public:
    Controller(Model& model) noexcept : model_{model} {}

    void OK_Clicked() { model_.DoSomething(); }

    Model& model_;
};

TEST(DesignPatternQ, Observer)
{
    auto model      = Model{};
    auto controller = Controller{model};

    controller.OK_Clicked();
    controller.OK_Clicked();
    controller.OK_Clicked();
}

```

- 参照 [Observer](#)
- [解答例-Observer](#)

演習-デザインパターン選択1

- 問題
オブジェクトの状態と、それに伴う振る舞いを分離して記述する場合に使用されるデザインパターンとは何か？下記から選択せよ。
- 選択肢
 1. Singleton
 2. State
 3. Observer
 4. Null Object
- [解答-デザインパターン選択1](#)

演習-デザインパターン選択2

- 問題
オブジェクトへのポインタがヌルかどうかを確かめるif文が頻繁に出てくる場合、そのif文を無くすために使われるデザインパターンは何か？下記から選択せよ。
- 選択肢
 1. Singleton
 2. State
 3. Observer
 4. Null Object
- 解答-デザインパターン選択2

演習-デザインパターン選択3

- 問題
MVCアーキテクチャの実現のためによく使われるデザインパターンは何か？以下から選択せよ。
- 選択肢
 1. Singleton
 2. State
 3. Observer
 4. Null Object
- 解答-デザインパターン選択3

開発プロセスとインフラ(全般)

演習-プロセス分類

- 問題
ソフトウェア開発プロセスは(A)、(B)、(C)の3つに分類できる。(A)から順に初期計画順守的であり、(C)から逆順に状況適応的である。状況適応的であることは、無計画であることを意味しない。ただ単にプライオリティの問題として、計画に従うことより状況に適応、対処することを選択するということである。
(A)、(B)、(C)それぞれにふさわしいものを下記から選べ。
- 選択肢
 1. ウォーターフォール
 2. 反復型
 3. アジャイル
 4. 無手順
- 解答-プロセス分類

演習-V字モデル

- 問題
ウォーターフォールモデルもしくはV字モデルと呼ばれるプロセスでは、「フェーズA」→「フェーズB」→「フェーズC」→「フェーズD」→「プログラミング」といった工程でソフトウェアを作り、その後「単体テスト(UT)」→「結合テスト(IT)」→「システムテスト」→「受入テスト(運用テスト)」といった工程でテストを行う。
フェーズA、B、C、Dそれぞれにふさわしいものを下記から選べ。
- 選択肢
 1. 詳細設計
 2. 基本設計
 3. 機能設計
 4. 要件分析

- 解答-V字モデル

演習-アジャイル

- 問題

アジャイル系プロセスの説明にふさわしくないものを選べ。

- 選択肢

1. アジャイル系プロセスとは、敏捷かつ適応的にソフトウェア開発を行う軽量な開発手法群の総称である。
2. アジャイル系プロセスには計画は必要ない。
3. ほとんどのアジャイル系プロセスでは、イテレーションを繰り返すことにより開発を進める。
4. アジャイル系プロセスにはスクラムやXPがある。

- 解答-アジャイル

演習-自動化

- 問題

一般に、(A)とは、個々のクラスや関数といったソフトウェア構成要素の機能が正確に動作することを検証するためのテストを指す。

原理的には、デバッグ等を利用して(B)で(A)を実行することは可能であるが、

「工数が膨大になる」、「テストの(C)が低い」

等の問題があるため、現実的ではない。自動(A)とは、この問題に対処するためのもので、ワンコマンドもしくはワンクリックで(A)を実行するよう開発されるプログラムである。

(A)、(B)、(C)それぞれにふさわしいものを下記から選べ。

- 選択肢

1. 単体テスト
2. 統合テスト
3. 自動
4. 手作業
5. 再現性
6. 可塑性

- 解答-自動化

演習-単体テスト

- 問題

単体テストの説明としてふさわしくないものを選べ。

- 選択肢

1. 単体テストで検出可能なバグを、統合テストで検出・デバッグすることは非効率である。
2. 単体テストが可能なクラス設計には、プログラマのスキルの向上が必要である。
3. 自動単体テストは工数をほとんどロスすることなしに何度でも実行できるため、機能追加、バグ修正、リファクタリング等のソースコード修正後の回帰テストが容易になる。
4. 単体テストはアジャイル系プロセスのみで使われるテスト手法である。

- 解答-単体テスト

演習-リファクタリングに付随する活動

- 問題

リファクタリングに付随する活動ではないものを選べ。

- 選択肢

1. ソースコードインスペクション
2. 受入テスト
3. 回帰テスト
4. クラス分割

- 解答-リファクタリングに付随する活動

演習-リファクタリング対象コード

- 問題
リファクタリングの対象とならないソースコードの問題点を下記から選べ。
- 選択肢

1. 巨大なクラス
2. コードクローン
3. メモリリーク
4. 間違った依存関係

- 解答-リファクタリング対象コード

演習-CI

- 問題
CI(継続的インテグレーション)を前提とするプロセスに特徴的な活動すべてを下記から選べ。
- 選択肢

1. (svnやgitリポジトリへの)コミット前にコードインスペクションを行う。
2. リポジトリの最新ソースコードを自動的にビルド、単体テストを行うプログラムの開発や設定を行う。
3. コミット前に回帰テストを行う。
4. 開発者全員に共有されているブランチになるべく頻繁にコミットする。

- 解答-CI

テンプレートメタプログラミング

演習-パラメータパック

- 問題

```
// @@@ exercise/template_q/parameter_pack.cpp 5
// [Q]
// 下記の関数Maxは、単体テストが示す通り、2つのパラメータの大きい方を返す。
// 任意の個数の引数を取れるようにMaxを修正せよ。

template <typename T>
T Max(T const& t0, T const& t1) noexcept
{
    return t0 > t1 ? t0 : t1;
}

TEST(TemplateMetaProgrammingQ, parameter_pack)
{
    ASSERT_EQ(2, Max(1, 2));
    ASSERT_EQ("bcd", Max(std::string("abc"), std::string("bcd")));
}
```

- 参照 [パラメータパック](#)
- 解答例-パラメータパック

演習-エイリアステンプレート

- 問題

```
// @@@ exercise/template_q/template_alias.cpp 5
// [Q]
// 下記の単体テストでしているstd::vector<std::vector<XXX>>を、
// テンプレートエイリアスによって簡潔に記述せよ。

TEST(TemplateMetaProgrammingQ, template_alias)
{
    {
```

```

        auto vv = std::vector<std::vector<int>>{{1, 2, 3}, {3, 4, 5}};
        ASSERT_EQ(2, vv.size());
        ASSERT_EQ((std::vector<int>{1, 2, 3}), vv[0]);
        ASSERT_EQ((std::vector<int>{3, 4, 5}), vv[1]);
        ASSERT_EQ(5, vv[1][2]);
    }
    {
        auto vv = std::vector<std::vector<float>>{{1, 2, 3}, {3, 4, 5}};
        ASSERT_EQ(2, vv.size());
        ASSERT_EQ((std::vector<float>{1, 2, 3}), vv[0]);
        ASSERT_EQ((std::vector<float>{3, 4, 5}), vv[1]);
        ASSERT_EQ(5, vv[1][2]);
    }
    {
        auto vv = std::vector<std::vector<std::string>>{{"1", "2", "3"}, {"3", "4", "5"}};
        ASSERT_EQ(2, vv.size());
        ASSERT_EQ((std::vector<std::string>{"1", "2", "3"}), vv[0]);
        ASSERT_EQ((std::vector<std::string>{"3", "4", "5"}), vv[1]);
        ASSERT_EQ("5", vv[1][2]);
    }
}

```

- 解答例-エイリアステンプレート

演習-名前空間による修飾不要なoperator<<

- 問題

```

// @@@ exercise/template_q/put_to.cpp 3
// [Q]
// 下記のように名前空間TemplateMP、エイリアスInts_tとそのoperator<<が定義されている場合、
// 単体テストで示した通り、Ints_tのoperator<<を使用するためには、
// 名前空間による修飾やusing宣言/ディレクティブの記述が必要になる。
// Ints_tをstd::vectorから継承したクラスとして定義することにより、このような記述を不要にせよ。

```

```

namespace TemplateMP {

using Ints_t = std::vector<int>

std::ostream& operator<<(std::ostream& os, Ints_t const& ints)
{
    auto first = true;
    for (auto i : ints) {
        if (!std::exchange(first, false)) {
            os << " : ";
        }
        os << i;
    }

    return os;
}
} // namespace TemplateMP

```

```

namespace {
TEST(TemplateMetaProgrammingQ, put_to)
{
    {
        auto oss = std::ostringstream{};
        auto ints = TemplateMP::Ints_t{1, 2, 3};

        // oss << ints;
        TemplateMP::operator<<(oss, ints); // 名前空間による修飾

        ASSERT_EQ("1 : 2 : 3", oss.str());
    }

    {
        auto oss = std::ostringstream{};
        auto ints = TemplateMP::Ints_t{1, 2, 3};

        using TemplateMP::operator<<; // using宣言
        oss << ints;

        ASSERT_EQ("1 : 2 : 3", oss.str());
    }

    {
        auto oss = std::ostringstream{};
        auto ints = TemplateMP::Ints_t{1, 2, 3};

        using namespace TemplateMP; // usingディレクティブ

```

```

        oss << ints;
        ASSERT_EQ("1 : 2 : 3", oss.str());
    }
} // namespace

```

- 参照 Ints_tのログ登録、Ints_tを構造体としてApp内に宣言する
- 解答例-名前空間による修飾不要なoperator<<

演習-std::arrayの継承

- 問題

```

// @@@ exercise/template_q/safe_array.cpp 3
// [Q]
// std::array、std::vector、std::string等のSTLの配列型コンテナはインデックスアクセスに対して、
// レンジのチェックをしないため、不正なメモリアクセスをしてしまうことがある。
// std::arrayを使用して、このような問題のないSafeArrayを作り、単体テストを行え。

namespace {
TEST(TemplateMetaProgrammingQ, safe_array)
{
    // SafeArrayの単体テスト
}
} // namespace

```

- 参照 安全な配列型コンテナ
- 解答例-std::arrayの継承

演習-SFINAEを利用しない関数テンプレートの特殊化によるis_void

- 問題

```

// @@@ exercise/template_q/is_void.cpp 3
// [Q]
// 下記の仕様を満たす関数テンプレートis_void_f<T>と定数テンプレートis_void_f_v<T>を作れ。
//   * 与えられたテンプレートパラメータがvoidの場合、trueを返す
//   * 与えられたテンプレートパラメータがvoidでない場合、falseを返す
//   * std::is_sameを使わない
//   * 下記の単体テストをパスする(#if 0を削除してもコンパイルできる)

namespace IsVoidTest {
void test_func_0() noexcept {};
std::string test_func_1() { return "test"; };
} // namespace IsVoidTest

namespace {

TEST(TemplateMetaProgrammingQ, is_void_f)
{
#ifndef 0
    static_assert(!is_void_f_v<int>);
    static_assert(is_void_f_v<void>);
    static_assert(is_void_f_v<decltype(IsVoidTest::test_func_0())>);
    static_assert(!is_void_f_v<decltype(IsVoidTest::test_func_1())>);
#endif
}
} // namespace

```

- 参照 is_void_fの実装
- 解答例-SFINAEを利用しない関数テンプレートの特殊化によるis_void

演習-SFINAEを利用しないクラステンプレートの特殊化によるis_void

- 問題 // @@@@ exercise/template_q/is_void.cpp 30 // [Q] // 下記の仕様を満たすクラステンプレートis_void_sと定数テンプレートis_void_s_vを作れ。 /* 与えられたテンプレートパラメータがvoidの場合、メンバvalueがtrueになる */ 与えられたテンプレートパラメータがvoidでない場合、メンバvalueがtrueになる /* std::is_sameを使わない */ /* std::true_type/std::false_typeを利用する */ 下記の単体テストをパスする (#if 0を削除してもコンパイルできる)

```
namespace {
```

```
TEST(TemplateMetaProgrammingQ, is_void_s) { #if 0 static_assert(!is_void_s_v); static_assert(is_void_s_v);  
static_assert(is_void_s_v< decltype(IsVoidTest::test_func_0())>); static_assert(!is_void_s_v< decltype(IsVoidTest::test_func_1())>); #endif } //  
namespace
```

- 参照 [is_void_sの実装](#)
- 解答例-SFINAEを利用しないクラステンプレートの特殊化によるis_void

演習-SFINAEを利用した関数テンプレートの特殊化によるis_void

- 問題

```
// @@@ exercise/template_q/is_void.cpp 53  
// [Q]  
// 下記の仕様を満たす関数テンプレートis_void_sfinae_f<T>と  
// 定数テンプレートis_void_sfinae_f<T>を作れ。  
//   * 与えられたテンプレートパラメータがvoidの場合、trueを返す  
//   * 与えられたテンプレートパラメータがvoidでない場合、falseを返す  
//   * std::is_sameを使わない  
//   * SFINAEを利用する  
//   * 下記の単体テストをパスする(#if 0を削除してもコンパイルできる)  
  
namespace {  
  
TEST(TemplateMetaProgrammingQ, is_void_sfinae_f)  
{  
#if 0  
    static_assert(!is_void_sfinae_f_v<int>);  
    static_assert(is_void_sfinae_f_v<void>);  
    static_assert(is_void_sfinae_f_v< decltype(IsVoidTest::test_func_0())>);  
    static_assert(!is_void_sfinae_f_v< decltype(IsVoidTest::test_func_1())>);  
#endif  
}  
} // namespace
```

- 参照 [is_void_sfinae_fの実装](#)
- 解答例-SFINAEを利用した関数テンプレートの特殊化によるis_void

演習-SFINAEを利用したクラステンプレートの特殊化によるis_void

- 問題

```
// @@@ exercise/template_q/is_void.cpp 77  
// [Q]  
// 下記の仕様を満たすクラステンプレートis_void_sfinae_s<T>と  
// 定数テンプレートis_void_sfinae_s<T>を作れ。  
//   * 与えられたテンプレートパラメータがvoidの場合、メンバvalueがtrueになる  
//   * 与えられたテンプレートパラメータがvoidでない場合、メンバvalueがtrueになる  
//   * std::is_sameを使わない  
//   * std::true_type/std::false_typeを利用する  
//   * SFINAEを利用する  
//   * 下記の単体テストをパスする(#if 0を削除してもコンパイルできる)  
  
namespace {  
  
TEST(TemplateMetaProgrammingQ, is_void_sfinae_s)  
{  
#if 0  
    static_assert(!is_void_sfinae_s_v<int>);  
    static_assert(is_void_sfinae_s_v<void>);  
    static_assert(is_void_sfinae_s_v< decltype(IsVoidTest::test_func_0())>);  
    static_assert(!is_void_sfinae_s_v< decltype(IsVoidTest::test_func_1())>);  
#endif  
}  
} // namespace
```

- 参照 [is_void_sfinae_sの実装](#)
- 解答例-SFINAEを利用したクラステンプレートの特殊化によるis_void

演習-テンプレートテンプレートパラメータ

- 問題

```
// @@@ exercise/template_q/template_template.cpp 3  
// [Q]
```

```

// 以下の使用を満たすクラステンプレートを作れ。
// * 任意のSTLコンテナを唯一のテンプレートパラメータとする
// * そのコンテナを使用しint型のデータを格納する

namespace {

TEST(TemplateMetaProgrammingQ, template_template)
{
    //
}
} // namespace

```

- 参照 [is_same_tempルの実装](#)
- [解答例-テンプレートテンプレートパラメータ](#)

演習-テンプレートパラメータを可変長にしたstd::is_same

- 問題

```

// @@@ exercise/template_q/is_same.cpp 3
// [Q]
// 以下の仕様を満たすクラステンプレートis_same_some_of<T, U...>と
// 定数テンプレートis_same_some_of_v<T, U...>を作れ。
// * 2個以上のテンプレートパラメータを持つ
// * 第1パラメータと他のパラメータの何れかが同一の型であった場合、メンバvalueがtrueになる
// * 前行の条件が成立しなかった場合、メンバvalueがfalseになる
// * 型の同一性はstd::is_sameを使って判定する

namespace {

TEST(TemplateMetaProgrammingQ, is_same_some_of)
{
#ifndef 0
    static_assert(!is_same_some_of_v<int, int8_t, int16_t, uint16_t>);
    static_assert(is_same_some_of_v<int, int8_t, int16_t, uint16_t, int32_t>);
    static_assert(is_same_some_of_v<int&, int8_t, int16_t, int32_t&, int32_t>);
    static_assert(!is_same_some_of_v<int&, int8_t, int16_t, uint32_t&, int32_t>);
    static_assert(is_same_some_of_v<std::string, int, char*, std::string>);
    static_assert(!is_same_some_of_v<std::string, int, char*>);
#endif
}
} // namespace

```

- 参照 [IsSameSomeOfの実装](#)
- [解答例-テンプレートパラメータを可変長にしたstd::is_same](#)

演習-メンバ関数の存在の診断

- 問題

```

// @@@ exercise/template_q/exists_func.cpp 3
// [Q]
// テンプレートパラメータの型がメンバ関数c_str()を持つか否かを判定する
// クラステンプレートhas_c_str<T>と定数テンプレートhas_c_str_v<T>を作れ。

namespace {

TEST(TemplateMetaProgrammingQ, has_c_str)
{
#ifndef 0
    static_assert(has_c_str_v<std::string>);
    static_assert(!has_c_str_v<std::vector<int>>);
#endif
}
} // namespace

```

- 参照 関数の存在の診断
- [解答例-メンバ関数の存在の診断](#)

演習-範囲for文のオペランドになれるかどうかの診断

- 問題

```

// @@@ exercise/template_q/exists_func.cpp 20
// [Q]

```

```

// 範囲for文は、
//   for(auto a : obj ) { ... }
// のような形式で表現される。
// テンプレートパラメータから生成されたオブジェクトが、
// このobjに指定できるか否かを判定するクラステンプレートis_range<T>
// と定数テンプレートis_range_v<T>を作れ。
namespace {

TEST(TemplateMetaProgrammingQ, is_range)
{
#ifndef __
    static_assert(is_range_v<std::string>);
    static_assert(is_range_v<std::vector<int>>);
    static_assert(!is_range_v<std::mutex>);
    static_assert(!is_range_v<std::lock_guard<std::mutex>>);
#endif
}
} // namespace

```

- 参照 [関数の存在の診断](#)
- [解答例-範囲for文のオペランドになれるかどうかの診断](#)

演習-配列の長さの取り出し

- 問題

```

// @@@ exercise/template_q/array_op.cpp 3
// [Q]
// 配列を引数に取り、その長さを返す関数テンプレートarray_lengthを作れ。

namespace {

TEST(TemplateMetaProgrammingQ, array_length)
{
#ifndef __
    int i[5];
    std::string str[]{"a", "b", "c"};

    static_assert(array_length(i) == 5);
    static_assert(array_length(str) == 3);
#endif
}
} // namespace

```

- 参照 [関数の存在の診断](#)
- [解答例-配列の長さの取り出し](#)

演習-配列の次元の取り出し

- 問題

```

// @@@ exercise/template_q/array_op.cpp 22
// [Q]
// 配列を引数に取り、その次元を返す関数テンプレートarray_dimensionを作れ。

namespace {

TEST(TemplateMetaProgrammingQ, array_dimension)
{
#ifndef __
    constexpr int i1[5]{};
    constexpr int i2[5][2]{};
    constexpr int i3[5][2][3]{};

    static_assert(array_dimension(i1) == 1);
    static_assert(array_dimension(i2) == 2);
    static_assert(array_dimension(i3) == 3);
#endif
}
} // namespace

```

- 参照 [ValueTypeの実装](#)
- [解答例-配列の次元の取り出し](#)

演習-関数型のテンプレートパラメータを持つクラステンプレート

- 問題

```
// @@@ exercise/template_q/scoped_guard.cpp 8
// [Q]
// RAIIを行うための下記クラスscoped_guardをstd::functionを使わずに再実装せよ。

class scoped_guard {
public:
    explicit scoped_guard(std::function<void()> f) noexcept : f_{f} {}
    ~scoped_guard() { f_(); }

    scoped_guard(scoped_guard const&) = delete;
    void operator=(scoped_guard const&) = delete;

private:
    std::function<void()> f_;
};

namespace {

TEST(TemplateMetaProgrammingQ, scoped_guard)
{
    {
        auto demangled = abi::__cxa_demangle(typeid(std::vector<int>).name(), 0, 0, nullptr);
        auto sg       = scoped_guard{[demangled]() noexcept { free(demangled); }};

        ASSERT_STREQ("std::vector<int, std::allocator<int>>", demangled);
    }
    {
        auto stream = popen("ls " __FILE__, "r");
        auto sg     = scoped_guard{[stream]() noexcept { fclose(stream); }};

        char buff[256]{};
        fgets(buff, sizeof(buff) - 1, stream);

        ASSERT_STREQ("scoped_guard.cpp\n", buff);
    }
}
} // namespace
```

- 参照 関数型をテンプレートパラメータで使う
- 解答例-関数型のテンプレートパラメータを持つクラステンプレート

解答

プログラミング規約(型)

解答-汎整数型の選択

- 選択肢3
- 参照 算術型
- 解説
代入する小さい整数に合わせて8ビット型や16ビット型を使うと、それら同士の演算時にint昇格が起こり、わかりづらいバグを生むことがある。
int32_tを使うことは、ほとんどのコンパイラでint32_tの実際の型がintであることを利用しているため、その前提を避けるべきと考えるのであれば、int32_tの代わりにint、uint32_tの代わりにunsigned intを使用すればよい。

- 演習-汎整数型の選択へ戻る。

解答例-汎整数型の演算

```
// @@@ exercise/programming_convention_a/type.cpp 14
TEST(ProgrammingConventionTypeA, GeneralInteger)
{
    // [A]
    // 以下の組み込み型の使用方法は、以下のテストコードを(環境依存で)パスするが、
    // 適切であるとは言えない。適切な型に修正せよ。
    int32_t b{1};
    int32_t i{b};
    int32_t c{-1};

    ASSERT_EQ(i * c, -1);
}
```

- 演習-汎整数型の演算へ戻る。

解答例-浮動小数点型

```
// @@@ exercise/programming_convention_a/type.cpp 28
double f(double a) noexcept { return 1 / a; }

template <typename FLOAT_0, typename FLOAT_1>
bool is_equal(FLOAT_0 lhs, FLOAT_1 rhs)
{
    static_assert(std::is_floating_point_v<FLOAT_0>, "FLOAT_0 shoud be float or double.");
    static_assert(std::is_same_v<FLOAT_0, FLOAT_1>, "FLOAT_0 and FLOAT_1 shoud be a same type.");

    return std::abs(lhs - rhs) <= std::numeric_limits<FLOAT_0>::epsilon();
}

TEST(ProgrammingConventionTypeA, Float)
{
    // [A]
    // 以下の両辺を同一と判定するための関数を作り、その関数の単体テストを行え。
    ASSERT_TRUE(is_equal(1.0, 1 + 0.001 - 0.001));

    // [A]
    // 以下の0除算を捕捉するためのコードを書け。
    std::feclearexcept(FE_ALL_EXCEPT); // エラーをクリア
    f(0.0);
    ASSERT_TRUE(std::fetestexcept(FE_ALL_EXCEPT) & FE_DIVBYZERO);
    std::feclearexcept(FE_ALL_EXCEPT); // エラーをクリア
}
```

- 演習-浮動小数点型へ戻る。

解答-定数列挙

- 選択肢4
- 参照 enum
- 演習-定数列挙へ戻る。

解答例-enum

```
// @@@ exercise/programming_convention_a/type.cpp 55
// [A]
// 以下のマクロ引数を型安全なenumに修正せよ

enum class Color { Red, Green, Blue };

std::string GetString(Color color)
{
    switch (color) {
        case Color::Red:
            return "Red";
        case Color::Green:
            return "Green";
        case Color::Blue:
            return "Blue";
        default:
            assert(false);
            return "";
    }
}

TEST(ProgrammingConventionTypeA, Enum)
{
    ASSERT_EQ(std::string{"Red"}, GetString(Color::Red));
    ASSERT_EQ(std::string{"Green"}, GetString(Color::Green));
    ASSERT_EQ(std::string{"Blue"}, GetString(Color::Blue));
}
```

- 演習-enumへ戻る。

解答例-配列の範囲for文

```
// @@@ exercise/programming_convention_a/type.cpp 84
int32_t array_value() noexcept
{
    static int32_t i;

    return i++;
}

TEST(ProgrammingConventionTypeA, Array)
{
    // [A]
    // 以下の配列の値の設定を範囲for文を使って書き直せ
    int32_t array[10];

    for (auto& a : array) {
        a = array_value();
    }

    ASSERT_EQ(0, array[0]);
    ASSERT_EQ(3, array[3]);
    ASSERT_EQ(9, array[9]);
}
```

- 演習-配列の範囲for文へ戻る。

解答例-エイリアス

```
// @@@ exercise/programming_convention_a/type.cpp 108
// [A]
// 以下のtypedefをC++11から導入された新しい形式のエイリアスに直せ。
using uchar     = unsigned char;
using func_type = bool (*)(int32_t);

// [A]
// template引数で与えられた型のオブジェクトをstd::vectorで保持するエイリアストラックを
```

```
// 定義し、その単体テストを行え。
template <class T>
using TypeVector = std::vector<T>

TEST(ProgrammingConventionTypeA, Alias)
{
    auto a = TypeVector<std::string>{"abc", "de", "f"};
    ASSERT_EQ(std::vector<std::string>{"abc", "de", "f"}, a);
    ASSERT_EQ(3, a.size());
}
```

- 演習-エイリアスへ戻る。

解答-constの意味

- 選択肢1
- 参照 const/constexprインスタンス
- 演習-constの意味へ戻る。

解答例-const/constexpr

```
// @@@ exercise/programming_convention_a/type.cpp 129
// [A]
// 下記のStringHolderに「const/constexprを付加する」等を行い、より良いコードに修正せよ。

class StringHolder {
public:
    StringHolder() = default;
    void Add(std::string const& str)
    {
        if (vector_len_max_ > strings_.size()) {
            strings_.push_back(str);
        }
    }

    std::vector<std::string> const& GetStrings() const noexcept { return strings_; }

private:
    static constexpr size_t vector_len_max_{3};
    std::vector<std::string> strings_{};
};

TEST(ProgrammingConventionTypeA, ConstConstexpr)
{
    auto sh = StringHolder{};

    ASSERT_EQ(std::vector<std::string>{}, sh.GetStrings());

    sh.Add("a");
    sh.Add(std::string("bc"));
    ASSERT_EQ((std::vector<std::string>{"a", "bc"}), sh.GetStrings());

    sh.Add("def");
    sh.Add(std::string("g"));
    ASSERT_EQ((std::vector<std::string>{"a", "bc", "def"}), sh.GetStrings());
}
```

- 演習-const/constexprへ戻る。

解答例-危険なconst_cast

```
// @@@ exercise/programming_convention_a/type.cpp 166
// [A]
// 下記の"DISABLED_"を削除し、何が起こるのか、なぜそうなるのかを確かめた上で、
// nameの型やその初期化を行っているコードを修正せよ。
TEST(ProgrammingConventionTypeA, ConstConstexpr2)
{
    char name[] = "abcdef";

    for (auto& n : name) {
        n = std::toupper(n);
    }

    ASSERT_STREQ("ABCDEF", name);
```

```
    ASSERT_EQ("ABCDEF", std::string{name});
}
```

- 演習-危険なconst_castへ戻る。

解答例-リテラル

```
// @@@ exercise/programming_convention_a/type.cpp 186
int32_t literal_test(int64_t) noexcept { return 0; }
int32_t literal_test(int32_t*) noexcept { return 1; }

// [A]
// 下記変数の初期化コードをコメントに基づき適切に修正せよ。
TEST(ProgrammingConventionTypeA, Literal)
{
    int32_t* p{nullptr};           // NULLは使用不可
    uint64_t a{0x1234'5678'90ab'cdef}; // 適切なセパレータを挿入
    int32_t b{0b0111'0001'0101};    // ビット表現に修正

    // [A]
    // 下記resultはfalseになるが、その理由を述べ、trueになるようにコードを修正せよ。
    bool const result{!(literal_test(nullptr) == literal_test(p))};
    ASSERT_TRUE(result);

    ASSERT_EQ(0x1234567890abcdef, a);
    ASSERT_EQ(b, 0x715);
}
```

- 演習-リテラルへ戻る。

解答-適切なautoの使い方

- 選択肢4
- 参照 auto
- 演習-適切なautoの使い方へ戻る。

解答-ポインタの初期化

- 選択肢3
- 参照 リテラル
- 演習-ポインタの初期化へ戻る。

解答-vector初期化

- 選択肢1
- 参照 インスタンスの初期化
- 演習-vector初期化へ戻る。

解答例-インスタンスの初期化

```
// @@@ exercise/programming_convention_a/type.cpp 209
TEST(ProgrammingConventionTypeA, Initialization)
{
    // [A]
    // 変数a、b、v、wの定義と初期化を1文で行え。
    {
        int32_t a[3]{1, 1, 1};

        ASSERT_EQ(1, a[0]);
        ASSERT_EQ(1, a[1]);
        ASSERT_EQ(1, a[2]);
    }
    {
        int32_t b[3]{};

        ASSERT_EQ(0, b[0]);
        ASSERT_EQ(0, b[1]);
        ASSERT_EQ(0, b[2]);
    }
    {
        auto v = std::vector<std::string>{3, std::string{"1"}};
    }
}
```

```

        ASSERT_EQ("1", v[0]);
        ASSERT_EQ("1", v[1]);
        ASSERT_EQ("1", v[2]);
    }
{
    auto w = std::vector<std::string>{"0", "1", "2"};
    ASSERT_EQ("0", w[0]);
    ASSERT_EQ("1", w[1]);
    ASSERT_EQ("2", w[2]);
}

```

- 演習-インスタンスの初期化へ戻る。

プログラミング規約(クラス)

解答-凝集度の意味

- 選択肢4
- 参照 凝集度
- 演習-凝集度の意味へ戻る。

解答例-凝集度の向上

```

// @@@ exercise/programming_convention_a/class.cpp 7
// [A]
// 以下のクラスABCの凝集度が高くなるように、ABC、HasRealNumberSolutionをリファクタリングせよ。
// その時に、他の問題があれば併せて修正せよ。

class ABC { // 2次方程式のパラメータ保持
public:
    explicit ABC(int32_t a, int32_t b, int32_t c) noexcept : a_{a}, b_{b}, c_{c} {}

    bool HasRealNumberSolution() const noexcept { return 0 <= discriminant(); }

private:
    int32_t const a_;
    int32_t const b_;
    int32_t const c_;

    int32_t discriminant() const noexcept // 判定式
    {
        return b_ * b_ - 4 * a_ * c_;
    }
};

bool HasRealNumberSolution(ABC const& abc) noexcept { return abc.HasRealNumberSolution(); }

TEST(ProgrammingConventionClassA, Cohesion)
{
    {
        auto const abc = ABC{1, 2, 1};

        ASSERT_TRUE(HasRealNumberSolution(abc));
    }
    {
        auto const abc = ABC{2, 0, 1};

        ASSERT_FALSE(HasRealNumberSolution(abc));
    }
}

```

- 演習-凝集度の向上へ戻る。

解答-メンバ変数の初期化方法の選択

- 選択肢2
- 参照 非静的なメンバ変数/定数の初期化
- 演習-メンバ変数の初期化方法の選択へ戻る。

解答-メンバの型

- 選択肢3
- 参照 [インスタンスの初期化](#)
- 解説
関数の宣言と、クラスのデフォルトコンストラクタ呼び出しによるオブジェクトの生成は、プログラマを混乱させることがあるので注意が必要である。
- [演習-メンバの型](#)へ戻る。

解答例-メンバ変数の初期化

```
// @@@ exercise/programming_convention_a/class.cpp 46
// [A]
// 以下のMemberInitのメンバ変数を適切な方法で初期化せよ。

class MemberInit {
public:
    MemberInit() noexcept {}

    explicit MemberInit(int a) noexcept : a_{a}, b_{a, 99} {}

    int32_t GetA() const noexcept { return a_; }

    static constexpr size_t b_len{2};
    int32_t const (&GetB() const noexcept)[b_len] { return b_; }
    int32_t GetC() const noexcept { return c_; }

private:
    int32_t const a_{0};
    int32_t const b_[b_len]{1, 1};
    int32_t const c_{2};
};

TEST(ProgrammingConventionClassA, MemberInit)
{
    {
        auto mi = MemberInit{};

        ASSERT_EQ(0, mi.GetA());
        ASSERT_EQ(1, mi.GetB()[0]);
        ASSERT_EQ(1, mi.GetB()[1]);
        ASSERT_EQ(2, mi.GetC());
    }
    {
        auto mi = MemberInit{1};

        ASSERT_EQ(1, mi.GetA());
        ASSERT_EQ(1, mi.GetB()[0]);
        ASSERT_EQ(99, mi.GetB()[1]);
        ASSERT_EQ(2, mi.GetC());
    }
}
```

- [演習-メンバ変数の初期化](#)へ戻る。

解答例-スライシング

```
// @@@ exercise/programming_convention_a/class.cpp 89
// [A]
// 以下のクラスBaseはオブジェクトのスライシングを引き起こす。
// このような誤用を起こさないようにするために、Baseオブジェクトのコピーを禁止せよ。
// 合わせてクラスDerivedも含め、不十分な記述を修正せよ。

class Base {
public:
    explicit Base(char const* name = nullptr) noexcept : name_{name == nullptr ? "Base" : name} {}
    virtual ~Base() = default;

    virtual char const* Name0() const noexcept { return "Base"; }
    char const* Name1() const noexcept { return name_; }

    Base& operator=(Base const& rhs) = delete;
    Base(Base const&) = delete;

private:
```

```

    char const* name_;
};

class Derived final : public Base {
public:
    Derived() noexcept : Base{"Derived"} {}

    virtual char const* Name0() const noexcept override { return "Derived"; }
};

TEST(ProgrammingConventionClassA, Slicing)
{
    auto b      = Base{};
    auto d      = Derived{};
    Base& d_ref = d;

    // 以下はBase、Derivedの単純なテスト
    ASSERT_STREQ("Base", b.Name0());
    ASSERT_STREQ("Base", b.Name1());
    ASSERT_STREQ("Derived", d_ref.Name0());
    ASSERT_STREQ("Derived", d_ref.Name1());

#ifndef 0
    // Base::operator=(Base const& rhs)をdeleteしたためにこのような誤用はコンパイルエラーになる。

    // 以下はbがスライスされたオブジェクトであることのテスト
    // こういった誤用を防ぐためにBaseのコピーを禁止せよ。
    b = d_ref;
    ASSERT_STREQ("Base", b.Name0()); // vtblはBaseになるから
    ASSERT_STREQ("Derived", b.Name1()); // name_はコピーされるから
#else
    // 意図的に上記のようなことがしたい場合、下記のようにするべき。
    auto b_copy = Base{d_ref.Name0()};
    ASSERT_STREQ("Base", b_copy.Name0()); // vtblはBaseになるから
    ASSERT_STREQ("Derived", b_copy.Name1()); // name_はコピーされるから
#endif
}

```

- 演習-スライシングへ戻る。

解答例-オブジェクトの所有権

```

// @@@ exercise/programming_convention_a/class.cpp 146
class A {
public:
    explicit A(int32_t n) noexcept : num_{n} { last_constructed_num_ = num_; }
    ~A() { last_destructed_num_ = num_; }

    int32_t GetNum() const noexcept { return num_; }

    static int32_t LastConstructedNum() noexcept { return last_constructed_num_; }
    static int32_t LastDestructedNum() noexcept { return last_destructed_num_; }

private:
    int32_t        num_;
    static int32_t last_constructed_num_;
    static int32_t last_destructed_num_;
};

int32_t A::last_constructed_num_ = -1;
int32_t A::last_destructed_num_ = -1;

class X final {
public:
    void Move(std::unique_ptr<A>&& ptr) noexcept
    {
        ptr_ = std::move(ptr); // ptr->ptr_へ所有権の移動
    }

    std::unique_ptr<A> Release() noexcept
    {
        return std::move(ptr_); // ptr_から外部への所有権の移動
    }

    A const* GetA() const noexcept { return ptr_.get(); }
    X() = default;
    ~X() = default;

private:

```

```

        std::unique_ptr<A> ptr_{}; }

TEST(ProgrammingConventionClassA, Ownership)
{
    // [A]
    // 以下の単体テストを完成させよ。

    ASSERT_EQ(-1, A::LastConstructedNum());           // まだ、A::A()は呼ばれてない
    ASSERT_EQ(-1, A::LastDestructedNum());             // まだ、A::~A()は呼ばれてない

    auto a0 = std::make_unique<A>(0);                 // a0はA(0)を所有
    auto a1 = std::make_unique<A>(1);                 // a1はA(1)を所有
    auto x = X {};

    ASSERT_EQ(1, A::LastConstructedNum());             // A(1)は生成された
    ASSERT_EQ(-1, A::LastDestructedNum());             // まだ、A::~A()は呼ばれてない
    ASSERT_EQ(0, a0->GetNum());                      // a0はA(0)を所有
    x.Move(std::move(a0));                            // a0からxへA(0)の所有権の移動
    ASSERT_FALSE(a0);                                // a0は何も所有していない

    ASSERT_EQ(1, a1->GetNum());                      // a1はA(1)を所有
    x.Move(std::move(a1));                            // xによるA(0)の解放
    // a1からxへA(1)の所有権の移動
    ASSERT_EQ(0, A::LastDestructedNum());             // A(0)は解放された
    ASSERT_FALSE(a1);                                // a1は何も所有していない
    ASSERT_EQ(1, x.GetA()->GetNum());                // xはA(1)を所有

    std::unique_ptr<A> a2{x.Release()};
    ASSERT_EQ(nullptr, x.GetA());                     // xは何も所有していない
    ASSERT_EQ(1, a2->GetNum());                      // a2はA(1)を所有
    {
        std::unique_ptr<A> a3{std::move(a2)};
        ASSERT_FALSE(a2);                            // a2は何も所有していない
        ASSERT_EQ(1, a3->GetNum());                // a3はA(1)を所有
        // a3によるA(1)の解放
    }
    ASSERT_EQ(1, A::LastDestructedNum());
}

```

- 演習-オブジェクトの所有権へ戻る。

プログラミング規約(関数)

解答例-非メンバ関数の宣言

```

// @@@ exercise/programming_convention_a/func.cpp 11
TEST(ProgrammingConventionFuncA, NonMemberFunc)
{
    // [A]
    // 適切な#includeを追加し、上記のextern宣言がなくとも下記がコンパイルできるようにせよ。

    // このファイルの先頭付近に
    // #include <cmath>
    // を追加した。

    ASSERT_EQ(1, cos(0));
}

```

- 演習-非メンバ関数の宣言へ戻る。

解答例-メンバ関数の修飾

```

// @@@ exercise/programming_convention_a/func.cpp 25
// [A]
// 下記のクラスAのメンバ関数の不正確な記述を修正せよ。
// また、単体テストを同様に修正せよ。

class A {
public:
    A() : strings_{GetStringsDefault()} {}

    void SetStrings(size_t index, std::string str)
    {
        if (index < max_len) {
            strings_[index] = str;
        }
    }
}

```

```

    }

    std::vector<std::string>& GetStrings() noexcept { return strings_; }
    std::vector<std::string> const& GetStrings() const noexcept { return strings_; }
    constexpr size_t MaxLen() const noexcept { return max_len; }

    static std::vector<std::string> const& GetStringsDefault()
    {
        static const std::vector<std::string> strings_default{max_len, ""};
        return strings_default;
    }

private:
    std::vector<std::string> strings_;
    static constexpr size_t max_len{3};
};

TEST(ProgrammingConventionFuncA, MemberFunc)
{
    auto a = A{};

    auto const& strings_default = a.GetStringsDefault();
    ASSERT_EQ((std::vector<std::string>{a.MaxLen(), ""}), strings_default);

    auto const& strings = a.GetStrings();

    ASSERT_EQ((std::vector<std::string>{a.MaxLen(), ""}), strings);

    a.SetStrings(1, "TEST");
    ASSERT_EQ("", strings[0]);

    // [A]
    // このテストをASSERT_EQでパスできるようにせよ
    ASSERT_EQ("TEST", strings[1]);

    ASSERT_EQ("", strings[2]);

    // 上記は下記のように書くべき
    ASSERT_EQ((std::vector<std::string>{"", "TEST", ""}), strings);
}

```

- 演習-メンバ関数の修飾へ戻る。

解答例-特殊メンバ関数の削除

```

// @@@ exercise/programming_convention_a/func.cpp 81
// [A]
// 下記クラスAutoGenのコンパイラが自動生成するメンバ関数を生成しないようにせよ。

class AutoGen {
public:
    AutoGen() = delete;
    ~AutoGen() = delete;

    AutoGen(AutoGen const&) = delete;
    AutoGen& operator=(AutoGen const&) = delete;
    AutoGen(AutoGen&&) noexcept = delete;
    AutoGen& operator=(AutoGen&&) noexcept = delete;
};

```

- 演習-特殊メンバ関数の削除へ戻る。

解答例-委譲コンストラクタ

```

// @@@ exercise/programming_convention_a/func.cpp 97
// [A]
// 下記クラスDelConstructorの2つのコンストラクタのコードクローンができるだけ排除せよ。

class DelConstructor {
public:
    explicit DelConstructor(std::string const& str)
        : str0_{str + "0"}, str1_{str + "1"}, str2_{str + "2"}
    {}

    explicit DelConstructor(int32_t num) : DelConstructor{std::to_string(num) + "_"} {}

    std::string const& GetString0() const { return str0_; }

```

```

    std::string const& GetString1() const { return str1_; }
    std::string const& GetString2() const { return str2_; }

private:
    std::string const str0_;
    std::string const str1_;
    std::string const str2_;
};

TEST(ProgrammingConventionFuncA, Constructor)
{
{
    auto const dc = DelConstructor{"hehe"};
    ASSERT_EQ("hehe0", dc.GetString0());
    ASSERT_EQ("hehe1", dc.GetString1());
    ASSERT_EQ("hehe2", dc.GetString2());
}
{
    auto const dc = DelConstructor{123};
    ASSERT_EQ("123_0", dc.GetString0());
    ASSERT_EQ("123_1", dc.GetString1());
    ASSERT_EQ("123_2", dc.GetString2());
}
}

```

- 演習-委譲コンストラクタへ戻る。

解答例-copyコンストラクタ

```

// @@@ exercise/programming_convention_a/func.cpp 139
// [A]
// 下記クラスInteger、IntegerHolderに適切にcopyコンストラクタ、copy代入演算子を追加して、
// 単体テストを行え。
class Integer {
public:
    explicit Integer(int32_t i) noexcept : i_{i} {}

    Integer(Integer const& rhs) noexcept = default;

    Integer& operator=(Integer const& rhs) noexcept = default;

    int32_t GetValue() const noexcept { return i_; }

private:
    int32_t i_;
};

class IntegerHolder {
public:
    explicit IntegerHolder(int32_t i) : integer_{std::make_unique<Integer>(i)} {}

    IntegerHolder(IntegerHolder const& rhs) : integer_{std::make_unique<Integer>(*rhs.integer_)} {}

    IntegerHolder& operator=(IntegerHolder const& rhs)
    {
        *integer_ = *(rhs.integer_);
        return *this;
    }

    int32_t GetValue() const noexcept { return integer_->GetValue(); }

private:
    std::unique_ptr<Integer> integer_;
};

TEST(ProgrammingConventionFuncA, Constructor2)
{
{
    auto i = Integer{3};
    ASSERT_EQ(3, i.GetValue());

    auto j = Integer{i};
    ASSERT_EQ(3, j.GetValue());

    auto k = Integer{0};
    ASSERT_EQ(0, k.GetValue());
}

```

```

        k = i;
        ASSERT_EQ(3, k.GetValue());
    }
{
    auto i = IntegerHolder{3};
    ASSERT_EQ(3, i.GetValue());

    auto j = IntegerHolder{i};
    ASSERT_EQ(3, j.GetValue());

    auto k = IntegerHolder{0};
    ASSERT_EQ(0, k.GetValue());

    k = i;
    ASSERT_EQ(3, k.GetValue());
}
}

```

- 演習-copyコンストラクタへ戻る。

解答例-moveコンストラクタ

```

// @@@ exercise/programming_convention_a/func.cpp 206
// [A]
// 上記問題を解決したIntegerHolderにmoveコンストラクタ、move演算子を追加した
// クラスIntegerHolder2を作成し、単体テストを行え。

class IntegerHolder2 {
public:
    explicit IntegerHolder2(int32_t i) : integer_{std::make_unique<Integer>(i)} {}

    IntegerHolder2(IntegerHolder2 const& rhs) : integer_{std::make_unique<Integer>(*rhs.integer_)}
    {
    }

    IntegerHolder2(IntegerHolder2&& rhs) noexcept : integer_{std::move(rhs.integer_)} {}

    IntegerHolder2& operator=(IntegerHolder2 const& rhs) noexcept
    {
        *integer_ = *(rhs.integer_);
        return *this;
    }

    IntegerHolder2& operator=(IntegerHolder2&& rhs) noexcept
    {
        integer_ = std::move(rhs.integer_);
        return *this;
    }

    int32_t GetValue() const
    {
        if (!integer_) {
            throw std::bad_exception{};
        }
        return integer_->GetValue();
    }

private:
    std::unique_ptr<Integer> integer_;
};

#ifndef __clang_analyzer__
TEST(ProgrammingConventionFuncA, Move)
{
    {
        auto i = IntegerHolder2{3};
        auto j = IntegerHolder2{std::move(i)};

        ASSERT_EQ(3, j.GetValue());

        ASSERT_THROW(i.GetValue(), std::bad_exception);
    }
    {
        auto i = IntegerHolder2{30};
        auto j = IntegerHolder2{0};

        ASSERT_EQ(0, j.GetValue());

        j = std::move(i);
    }
}
#endif

```

```

        ASSERT_EQ(30, j.GetValue());
    }
}
#endif

```

- 演習-moveコンストラクタへ戻る。

解答例-関数分割

```

// @@@ exercise/programming_convention_a/func.cpp 272
// [A]
// 下記PrimeNumbersは引数で与えられた整数以下の素数を返す関数である。
// PrimeNumbersの単体テストを作成し、その後、行数を短くする等のリファクタリングを行え。

namespace {
uint32_t next_prime_num(uint32_t curr_prime_num, std::vector<bool>& is_num_prime) noexcept
{
    for (auto i = 2 * curr_prime_num; i < is_num_prime.size(); i += curr_prime_num) {
        is_num_prime[i] = false; // 次の倍数は素数ではない
    }

    auto prime_num = curr_prime_num;

    do { // 次の素数の探索
        ++prime_num;
    } while (!is_num_prime[prime_num] && (prime_num < is_num_prime.size()));

    return prime_num;
}
} // namespace

std::vector<uint32_t> PrimeNumbers(uint32_t max_number)
{
    auto result = std::vector<uint32_t>{};

    if (max_number < 2) { // ガード節。2未満の素数はない。
        return result;
    }

    auto prime_num = 2U; // 最初の素数
    auto is_num_prime = std::vector<bool>(max_number + 1, true); // falseなら素数でない。
    is_num_prime[0] = is_num_prime[1] = false;

    do {
        result.emplace_back(prime_num);
        prime_num = next_prime_num(prime_num, is_num_prime);
    } while (prime_num < is_num_prime.size());

    return result;
}

TEST(ProgrammingConventionFuncA, Lines)
{
    ASSERT_EQ((std::vector<uint32_t>{}), PrimeNumbers(0));
    ASSERT_EQ((std::vector<uint32_t>{}), PrimeNumbers(1));
    ASSERT_EQ((std::vector<uint32_t>{2}), PrimeNumbers(2));
    ASSERT_EQ((std::vector<uint32_t>{2, 3}), PrimeNumbers(3));
    ASSERT_EQ((std::vector<uint32_t>{2, 3, 5, 7}), PrimeNumbers(8));
    ASSERT_EQ((std::vector<uint32_t>{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}), PrimeNumbers(30));
}

```

- 演習-関数分割へ戻る。

解答-オーバーライド関数の修飾

- 選択肢2
- 参照 [オーバーライド](#)
- 演習-オーバーライド関数の修飾へ戻る。

解答例-オーバーライド/オーバーロード

```

// @@@ exercise/programming_convention_a/func.cpp 325
// [A]

```

```

// 下記クラスBase、Derivedの単体テストを完成せよ。

class Base {
public:
    virtual ~Base() = default;
    int32_t f() noexcept { return 0; }

    virtual int32_t g() noexcept { return 0; }
};

class Derived : public Base {
public:
    int32_t f() noexcept { return 1; }

    virtual int32_t g() noexcept override { return 1; }
};

TEST(ProgrammingConventionFuncA, Overload)
{
    auto b      = Base{};
    auto d      = Derived{};
    Base& d_ref = d;

    ASSERT_EQ(0, b.f());
    ASSERT_EQ(0, b.g());

    ASSERT_EQ(1, d.f());
    ASSERT_EQ(1, d.g());

    ASSERT_EQ(0, d_ref.f());
    ASSERT_EQ(1, d_ref.g());
}

```

- 演習-オーバーライド/オーバーロードへ戻る。

解答例-オーバーロードによる誤用防止

```

// @@@ exercise/programming_convention_a/func.cpp 361
// [A]
// 下記関数Squareは、引数が浮動小数点となることを想定していない。
// 誤用を防ぐために、引数に浮動小数点を指定された場合、コンパイルできないようにせよ。
int32_t Square(int32_t a) noexcept { return a * a; }
int32_t Square(double a) noexcept = delete;

TEST(ProgrammingConventionFuncA, Overload2)
{
    ASSERT_EQ(9, Square(3));
    #if 0 // int32_t Square(double a) = delete;により下記はコンパイルできない。
    ASSERT_EQ(4, Square(2.5));
    #endif
}

```

- 演習-オーバーロードによる誤用防止へ戻る。

解答例-仮引数の修飾

```

// @@@ exercise/programming_convention_a/func.cpp 377
// [A]
// 下記AddStringsの仮引数等を適切に修正せよ。
using Strings = std::list<std::string>;
Strings AddStrings(Strings const& a, Strings const* b)
{
    auto ret = a;

    if (b == nullptr) {
        return ret;
    }

    ret.insert(ret.end(), b->begin(), b->end());

    return ret;
}

TEST(ProgrammingConventionFuncA, Parameter)
{
    auto a = Strings{"abc", "d"};
    auto b = Strings{"e", "fgh", "i"};

```

```

    auto ret = AddStrings(a, nullptr);
    ASSERT_EQ(ret, (Strings{"abc", "d"}));

    ret = AddStrings(a, &b);
    ASSERT_EQ(ret, (Strings{"abc", "d", "e", "fgh", "i"}));

    ASSERT_EQ(AddStrings(a, nullptr), (Strings{"abc", "d"}));
    ASSERT_EQ(AddStrings(a, &b), (Strings{"abc", "d", "e", "fgh", "i"}));
}

```

- 演習-仮引数の修飾へ戻る。

解答例-constexpr関数

```

// @@@ exercise/programming_convention_a/func.cpp 411
// [A]
// 下記Factorialをconstexpr関数にせよ。
constexpr uint32_t Factorial(uint32_t a) noexcept
{
    if (a == 0 || a == 1) {
        return 1;
    }

    return Factorial(a - 1) * a;
}

TEST(ProgrammingConventionFuncA, ConstexprFunc)
{
    static_assert(1 == Factorial(0), "Factorial fail");
    ASSERT_EQ(1, Factorial(0));
    ASSERT_EQ(6, Factorial(3));
    ASSERT_EQ(120, Factorial(5));
    ASSERT_EQ(3628800, Factorial(10));
}

```

- 演習-constexpr関数へ戻る。

解答-エクセプションの型

- 選択肢2
- 参照 エクセプション処理
- 解説

下記3つを統合して考えれば、必然的に「選択肢2」であることがわかる。

- エクセプションでthrowされるオブジェクトのポインタがnullptrになることはない。
- throwされたオブジェクトのスライスは当然避けるべきである。
- throwされたオブジェクトを直接修正すべきでない。

- 演習-エクセプションの型へ戻る。

プログラミング規約(構文)

解答例-コンテナの範囲for文

```

// @@@ exercise/programming_convention_a/syntax.cpp 8
// [A]
// 下記Accumulateのfor文を
// * イテレータを使ったfor文を使用したAccumulate2
// * 範囲for文を使用したAccumulate3
// を作り、それらの単体テストを行え。また、その時に他の不具合があれば合わせて修正せよ。

std::string Accumulate(std::vector<std::string> const& strings) noexcept
{
    auto ret = std::string{};

    for (auto i = 0U; i < strings.size(); ++i) {
        ret += strings[i];
    }

    return ret;
}

std::string Accumulate2(std::vector<std::string> const& strings) noexcept

```

```

{
    auto ret = std::string{};

#ifndef _0 // old style
    for (std::vector<std::string>::const_iterator it = strings.cbegin(); it != strings.cend(); ++it) {
        ret += *it;
    }
#else
    for (auto it = strings.cbegin(); it != strings.cend(); ++it) {
        ret += *it;
    }
#endif

    return ret;
}

std::string Accumulate3(std::vector<std::string> const& strings) noexcept
{
    auto ret = std::string{};

    for (auto const& s : strings) {
        ret += s;
    }

    return ret;
}

TEST(ProgrammingConventionSyntaxA, RangeFor)
{
    ASSERT_EQ("abcd", Accumulate(std::vector<std::string>{"a", "b", "cd"}));
    ASSERT_EQ("ABCD", Accumulate2(std::vector<std::string>{"A", "B", "CD"}));
    ASSERT_EQ("AbCd", Accumulate3(std::vector<std::string>{"A", "b", "Cd"}));
}

```

- 演習-コンテナの範囲for文へ戻る。

解答例-ラムダ式

```

// @@@ exercise/programming_convention_a/syntax.cpp 62
// [A]
// 下記のcopy_ifの第4引数をラムダ式を使って書き直せ。

TEST(ProgrammingConventionSyntaxA, Lambda)
{
    auto data = std::vector<std::string>("", "abc", "", "d");

    auto ret = std::vector<std::string>{};

    std::copy_if(data.cbegin(), data.cend(), std::back_inserter(ret),
                [] (auto const& s) noexcept { return s.size() != 0; });
    ASSERT_EQ((std::vector<std::string> {"abc", "d"}), ret);
}

```

- 演習-ラムダ式へ戻る。

解答例-ラムダ式のキャプチャ

```

// @@@ exercise/programming_convention_a/syntax.cpp 78
// [A]
// 下記Lambda::GetNameLessThan()のラムダ式の問題点を修正し、単体テストを行え。
class Lambda {
public:
    explicit Lambda(std::vector<std::string>&& strs) : strs_{std::move(strs)} {}
    std::vector<std::string> GetNameLessThan(uint32_t length) const
    {
        auto ret = std::vector<std::string>{};

        std::copy_if(strs_.cbegin(), strs_.cend(), std::back_inserter(ret),
                    [length = length] (auto const& str) noexcept { return (str.size() < length); });

        return ret;
    }
private:
    std::vector<std::string> strs_;
};

TEST(ProgrammingConventionSyntaxA, Lambda2)

```

```

{
    auto lambda = Lambda{{"abc", "abcdef", "a"}};

    ASSERT_EQ(lambda.GetNameLessThan(4), (std::vector<std::string>{"abc", "a"}));
    ASSERT_EQ(lambda.GetNameLessThan(2), (std::vector<std::string>{"a"}));
    ASSERT_EQ(lambda.GetNameLessThan(1), (std::vector<std::string>{}));
}

```

- 演習-ラムダ式のキャプチャへ戻る。

プログラミング規約(演算子)

解答例-三項演算子

```

// @@@ exercise/programming_convention_a/operator.cpp 8
// [A]
// 下記whichのif文を三項演算子を使用して書き直せ。
int32_t which(bool left, int32_t lhs, int32_t rhs) noexcept { return left ? lhs : rhs; }

TEST(ProgrammingConventionOperatorA, OoOperator)
{
    ASSERT_EQ(3, which(true, 3, 4));
    ASSERT_EQ(4, which(false, 3, 4));
}

```

- 演習-三項演算子へ戻る。

解答例-delete

```

// @@@ exercise/programming_convention_a/operator.cpp 20
// [A]
// 下記DeleteProblemのメモリ管理の問題を修正せよ。
// また、他の問題があれば、合わせて修正せよ。
class DeleteProblem {
public:
    DeleteProblem(char const* str0 = nullptr, char const* str1 = nullptr)
        : str0_{(str0 == nullptr) ? std::unique_ptr<std::string>{} :
                                         std::make_unique<std::string>(str0)},
          str1_{(str1 == nullptr) ? std::unique_ptr<std::string>{} :
                                         std::make_unique<std::string>(str1)}
    {}

    DeleteProblem(DeleteProblem const&)           = delete;
    DeleteProblem& operator=(DeleteProblem const&) = delete;

    std::string const* GetStr0() const noexcept { return str0_.get(); }
    std::string const* GetStr1() const noexcept { return str1_.get(); }

private:
    std::unique_ptr<std::string> str0_;
    std::unique_ptr<std::string> str1_;
};

TEST(ProgrammingConventionOperatorA, Delete)
{
    {
        auto const dp = DeleteProblem{};

        ASSERT_EQ(nullptr, dp.GetStr0());
        ASSERT_EQ(nullptr, dp.GetStr1());
    }

    {
        auto const dp = DeleteProblem{"abc"};

        ASSERT_EQ("abc", *dp.GetStr0());
        ASSERT_EQ(nullptr, dp.GetStr1());
    }

    {
        auto const dp = DeleteProblem{"abc", "de"};

        ASSERT_EQ("abc", *dp.GetStr0());
        ASSERT_EQ("de", *dp.GetStr1());
    }
}

```

- 演習-deleteへ戻る。

解答例-sizeof

```
// @@@ exercise/programming_convention_a/operator.cpp 71
// [A]
// 下記Size1() - Size4()の単体テストを作れ。
size_t Size0(int32_t a) noexcept { return sizeof(a); }

size_t Size1(int32_t a[10]) noexcept { return sizeof(a); }

size_t Size2(int32_t a[]) noexcept { return sizeof(a); }

size_t Size3(int32_t* a) noexcept { return sizeof(a); }

size_t Size4(int32_t (&a)[10]) noexcept { return sizeof(a); }

TEST(ProgrammingConventionOperatorA, Sizeof)
{
    int32_t array[10]{};

    ASSERT_EQ(4, Size0(array[0]));
    ASSERT_EQ(sizeof(void*), Size1(array));
    ASSERT_EQ(sizeof(void*), Size2(array));
    ASSERT_EQ(sizeof(void*), Size3(array));
    ASSERT_EQ(sizeof(int32_t) * 10, Size4(array));
}
```

- 演習-sizeofへ戻る。

解答例-dynamic_castの削除

```
// @@@ exercise/programming_convention_a/operator.cpp 97
// [A]
// 下記クラスX、Y、ZとGetNameをdynamic_castを使わずに書き直せ。

class X {
public:
    virtual std::string GetName() const { return "X"; }

    virtual ~X() = default;
};

class Y : public X {
public:
    virtual std::string GetName() const override { return "Y"; }
};

class Z : public X {
public:
    virtual std::string GetName() const override { return "Z"; }
};

std::string GetName(X const& x) { return x.GetName(); }

TEST(ProgrammingConventionOperatorA, Cast)
{
    auto x = X{};
    auto y = Y{};
    auto z = Z{};

    ASSERT_EQ("X", GetName(x));
    ASSERT_EQ("Y", GetName(y));
    ASSERT_EQ("Z", GetName(z));
}
```

- 演習-dynamic_castの削除へ戻る。

解答-キャスト

- 選択肢3
- 参照 キャスト、暗黙の型変換
- 解説

reinterpret_castも避けるべきであるが、組み込みソフト等でのハードウエアアドレスの記述等には、使わざるを得ない。
- 演習-キャストへ戻る。

プログラミング規約(スコープ)

解答-usingディレクティブ

- 選択肢1
- 参照 using宣言/usingディレクティブ
- 演習-usingディレクティブへ戻る。

プログラミング規約(その他)

解答-アサーションの選択

- 選択肢1
- 参照 assertion
- 演習-アサーションの選択へ戻る。

解答例-assert/static_assert

```
// @@@ exercise/programming_convention_a/etc.cpp 10
// [A]
// 下記FloatingPointは、Tが浮動小数点型、Tのインスタンスは非0であることを前提としている。
// 適切にアサーションを挿入して誤用を防げ。

template <typename T>
class FloatingPoint {
public:
    static_assert(std::is_floating_point_v<T>, "T must be floating point type");
    FloatingPoint(T num) noexcept : num_{num} { assert(num != 0); }

    T Get() const noexcept { return num_; }
    T Reciprocal() const noexcept { return 1 / num_; }

private:
    T const num_;
};

TEST(ProgrammingConventionFuncA_Opt, Assertion)
{
    auto f1 = FloatingPoint<float>{1.0F};
    auto d1 = FloatingPoint<double>{1.0};

    ASSERT_EQ(f1.Get(), 1.0F);
    ASSERT_EQ(d1.Get(), 1.0);
    ASSERT_DEATH(FloatingPoint<float>{0}, "num != 0");
    ASSERT_DEATH(FloatingPoint<double>{0}, "num != 0");

#ifdef _MSC_VER // コンパイルできない。
    auto i = FloatingPoint<int32_t>{1};
#endif
}
```

- 演習-assert/static_assertへ戻る。

SOLID

解答例-SRP

```
// @@@ exercise/solid_a/srp_test_score.h 8
// [A]
// 下記クラスTestScoreはメンバにする必要のない関数までメンバにしてるため、
// インターフェースが肥大化てしまい、少なくともSRPに反している。
// メンバにする必要のないStoreCSVを外部関数にせよ。
// また、受験者の平均点を求める
//     TestScore::ScoreOne_t Average(TestScore const& test_score);
// を同様の方法で作り、単体テストを行え。

class TestScore {
public:
    ...
}
```

```

// void StoreCSV(std::string const& filename) const;
// は外部関数にした。
void LoadCSV(std::string const& filename);

...
};

std::string ToString(TestScore const& ts);
void StoreCSV(TestScore const& test_score, std::string const& filename);
TestScore::ScoreOne_t Average(TestScore const& test_score);

// @@@ exercise/solid_a/srp_test_score.cpp 10

namespace {

...

bool is_valid_score(int32_t score) noexcept { return 0 <= score && score <= 100; }

bool not_score(int32_t score) noexcept { return score == -1; }
} // namespace

void TestScore::validate_score(int32_t score) const
{
    if (is_valid_score(score) || not_score(score)) {
        return;
    }

    throw std::out_of_range{"Invalid Score"};
}

void TestScore::AddScore(TestScore::ScoreOne_t const& one_test_score)
{
    for (auto const& pair : one_test_score) {
        validate_score(pair.second);
    }

    if (test_score_row_.size() == 0) {
        test_score_row_[one_test_score[0].first] = std::vector<int32_t>{};
    }

    for (auto& pair : test_score_row_) {
        pair.second.push_back(-1);
    }

    auto curr_test_count = test_score_row_.begin()->second.size();

    for (auto const& pair : one_test_score) {
        if (test_score_row_.find(pair.first) == test_score_row_.end()) {
            test_score_row_[pair.first] = std::vector<int32_t>(curr_test_count, -1);
        }

        test_score_row_[pair.first].back() = pair.second;
    }
}

void TestScore::LoadCSV(std::string const& filename)
{
    auto data = std::ifstream{filename};

    auto test_score_raw = ScoreAll_t{};
    auto line          = std::string{};
    while (std::getline(data, line)) {
        // std::pair<TestScore::ScoreAll_t::iterator, bool>
        auto ret = test_score_raw.insert(parse_line(line));
        assert(ret.second);
    }

    for (auto const& pair : test_score_raw) {
        for (auto const s : pair.second) {
            validate_score(s);
        }
    }

    test_score_row_.swap(test_score_raw);
}

void StoreCSV(TestScore const& test_score, std::string const& filename)
{
    auto data = std::ofstream{filename};
}

```

```

auto ss = std::ostringstream{};

for (auto const& pair : test_score) {
    ss << pair.first;
    for (auto const s : pair.second) {
        ss << ", " << s;
    }
    ss << std::endl;
}

data << ss.str();
}

...

```

TestScore::ScoreOne_t Average(TestScore const& test_score)

```

{
    auto ret = TestScore::ScoreOne_t{};

    for (auto const& pair : test_score) {
        auto sum = 0;
        auto valid_count = 0U;
        for (auto const s : pair.second) {
            if (is_valid_score(s)) {
                sum += s;
                ++valid_count;
            }
        }
        ret.emplace_back(std::make_pair(pair.first, valid_count == 0 ? -1 : sum / valid_count));
    }

    return ret;
}

```

```

// @@@ exercise/solid_a/srp_test_score_ut.cpp 13

namespace {

...

```

TEST_F(SolidSRP_A, TestScore_StoreCSV)

```

{
    auto ts = TestScore{};
    ts.LoadCSV(test_score_org_);
    StoreCSV(ts, test_score_act_);

    auto content_act = whole_file(test_score_act_);
    auto content_exp = whole_file(test_score_exp_);

    ASSERT_EQ(content_exp, content_act);

    // 不正ファイルロード
    auto ts2 = ts;
    ASSERT_THROW(ts.LoadCSV(test_score_exp_err_), std::out_of_range);

    // エクセプション 強い保証
    ASSERT_TRUE(std::equal(ts.begin(), ts.end(), ts2.begin()));
}

```

TEST_F(SolidSRP_A, TestScore_Average)

```

{
    auto ts = TestScore{};
    ts.LoadCSV(test_score_org_);

    auto const exp = TestScore::ScoreOne_t{
        TestScore::ScoreOne_t::value_type("堂林", 65),
        TestScore::ScoreOne_t::value_type("広輔", 26),
        TestScore::ScoreOne_t::value_type("會澤", 53),
        TestScore::ScoreOne_t::value_type("松山", 73),
        TestScore::ScoreOne_t::value_type("菊池", 50),
        TestScore::ScoreOne_t::value_type("鈴木", 60),
    };
    auto act = Average(ts);

    ASSERT_EQ(act, exp);
}
} // namespace

```

- 演習-SRPへ戻る。

解答例-OCP

```
// @@@ exercise/solid_a/ocp_test_score.h 8
// [A]
// 下記クラスTestScoreは、
//   * テスト受講者とその点数を保持/提供する。
//   * テスト受講者とその点数をCSVファイルからロードする。
// 責任を持つ。サポートするファイル形式が増えた場合、このクラスを修正せざるを得ないため、
// 機能拡張に対して開いていない。つまり、OCPに反していると言える
// (実際にはこの程度の違反が問題になることは稀である)。
//
// サポートしているファイル形式はCSVのみであったが、TSVを追加することになった。
// 今後もサポートするファイル形式を増やす必要があるため、OCPに従った方が良いと判断し、
// TestScoreの責務から「ファイルのロード」を外し、その機能を外部関数として定義することにした。
// これに従い、下記クラスTestScoreを修正し、外部関数
//   void LoadCSV(std::string const& filename, TestScore& test_score);
// を作り、単体テストを行え。
class TestScore {
public:
    TestScore() = default;
    TestScore(TestScore const&) = default;
    TestScore& operator=(TestScore const&) = delete;
    TestScore& operator=(TestScore&&) = default; // moveが必要になった

    ...

    // void LoadCSV(std::string const& filename);
    // は外部関数にした。

    ScoreAll_t::const_iterator begin() const noexcept { return test_score_row_.begin(); }
    ScoreAll_t::const_iterator end() const noexcept { return test_score_row_.end(); }

private:
    // int32_t score: 0～100はスコア、-1は未受験、それ以外は不正データ
    void validate_score(int32_t score) const;
    ScoreAll_t test_score_row_{};
};

std::string ToString(TestScore const& ts);
void LoadCSV(std::string const& filename, TestScore& test_score);
void StoreCSV(TestScore const& test_score, std::string const& filename);
TestScore::ScoreOne_t Average(TestScore const& test_score);

// @@@ exercise/solid_a/ocp_test_score.cpp 10
```

```
...

void LoadCSV(std::string const& filename, TestScore& test_score)
{
    auto data = std::ifstream{filename};

    auto test_score_raw = TestScore::ScoreAll_t{};
    auto line = std::string{};
    while (std::getline(data, line)) {
        // std::pair<TestScore::ScoreAll_t::iterator, bool>
        auto ret = test_score_raw.insert(parse_line(line));
        assert(ret.second);
    }

    auto one_test = TestScore::ScoreOne_t{};
    for (auto const& pair : test_score_raw) {
        one_test.emplace_back(std::make_pair(pair.first, 0));
    }

    auto const score_count = test_score_raw.begin() ->second.size();
    auto ts = TestScore{};

    for (auto i = 0U; i < score_count; ++i) {
        for (auto& pair : one_test) {
            pair.second = test_score_raw[pair.first][i];
        }
        ts.AddScore(one_test);
    }

    test_score = std::move(ts);
}

void StoreCSV(TestScore const& test_score, std::string const& filename)
{
```

```

auto data = std::ofstream{filename};
auto ss   = std::ostringstream{};

for (auto const& pair : test_score) {
    ss << pair.first;
    for (auto const s : pair.second) {
        ss << ", " << s;
    }
    ss << std::endl;
}

data << ss.str();
}

std::string ToString(TestScore const& ts)
{
    auto ss = std::ostringstream{};

    for (auto const& pair : ts) {
        ss << pair.first << ':';
        for (auto const s : pair.second) {
            ss << ' ' << s;
        }
        ss << std::endl;
    }

    return ss.str();
}
}

TestScore::ScoreOne_t Average(TestScore const& test_score)
{
    auto ret = TestScore::ScoreOne_t{};

    for (auto const& pair : test_score) {
        auto sum      = 0;
        auto valid_count = 0U;
        for (auto const s : pair.second) {
            if (is_valid_score(s)) {
                sum += s;
                ++valid_count;
            }
        }
        ret.emplace_back(std::make_pair(pair.first, valid_count == 0 ? -1 : sum / valid_count));
    }

    return ret;
}

```

// @@@ exercise/solid_a/ocp_test_score_ut.cpp 13

```

namespace {

TEST_F(SolidOCP_A, TestScore_LoadCSV)
{
    auto ts = TestScore{};
    LoadCSV(test_score_org_, ts);

    ...
}

...

TEST_F(SolidOCP_A, TestScore_AddScore)
{
    auto ts = TestScore{};
    LoadCSV(test_score_org_, ts);

    ...
}

TEST_F(SolidOCP_A, TestScore_GetScore)
{
    auto ts = TestScore{};
    LoadCSV(test_score_org_, ts);

    ...
}

TEST_F(SolidOCP_A, TestScore_StoreCSV)
{

```

```

auto ts = TestScore{};
LoadCSV(test_score_org_, ts);
StoreCSV(ts, test_score_act_);

auto content_act = whole_file(test_score_act_);
auto content_exp = whole_file(test_score_exp_);

ASSERT_EQ(content_exp, content_act);

// 不正ファイルロード
auto ts2 = ts;
ASSERT_THROW(LoadCSV(test_score_exp_err_, ts2), std::out_of_range);

// エクセプション 強い保証
ASSERT_TRUE(std::equal(ts.begin(), ts.end(), ts2.begin())));
}

TEST_F(SolidOCP_A, TestScore_Average)
{
    auto ts = TestScore{};
    LoadCSV(test_score_org_, ts);

    ...
}
} // namespace

```

- 演習-OCPへ戻る。

解答例-LSP

```

// @@@ exercise/solid_a/lsp_test_score.h 8
// [A]
// 下記クラスTestScoreが管理するテストのスコアの値は、
//     * 0~100 テストのスコア
//     * -1 未受験
//     * それ以外 不正値であるため、このデータを入力すると
//             std::out_of_rangeエクセプションが発生する。
// を表すが、未受講を許可しない仕様(受験できない場合のスコアは0点)の
// TestScoreForceも必要になったため下記のように定義した。
//     * TestScoreForceが管理するテストのスコアの値は
//         * 0~100 テストのスコア
//         * それ以外 不正値であるため、このデータを入力すると
//             std::out_of_rangeエクセプションが発生する。
//     * それ以外の動作はTestScoreと同じ。
// これは、事前条件(「-1~100を受け入れる」から「0~100を受け入れる」)の強化であるため、
// LSPに反する。
// これにより起こる問題点を単体テストを用いて指摘せよ。
//
// [A]
// 上記問題を解決するため、クラスTestScoreForceFixedを作り単体テストを行え。
//
// [解説]
// TestScoreForceFixedをTestScoreからprivate継承することで、
// TestScoreForceFixedとTestScoreの関係がis-aの関係ではなくなるためLSPに適合する。
// 一方で、private継承の影響で、TestScoreForceFixedは、
//     * void LoadCSV(std::string const& filename, TestScore& test_score);
//     * void StoreCSV(TestScore const& test_score, std::string const& filename);
// 等TestScoreオブジェクトのリファレンスやポインタを受け取る関数が使えなくなる
// (そもそもそれが目的でprivate継承にした)。
// これに対処するために、オリジナルのコードをほとんどクローンした
//     * void LoadCSV(std::string const& filename, TestScoreForceFixed& test_score);
//     * void StoreCSV(TestScoreForceFixed const& test_score, std::string const& filename);
// を作ることは当然、誤りである。
// クローンを作らずに対処するためのコードを単体テストに記述したので参照してほしい。
// また、TestScoreForceを作ったことが根本的な誤りであった可能性もある。
// 「未受講データ-1を入力された場合、それをテストスコア0と解釈する」ようなオプションを
// TestScoreに持たせることも考慮すべきであった。

class TestScore {
    ...
};

class TestScoreForce : public TestScore {
    ...
};

class TestScoreForceFixed : TestScore {
public:
    TestScoreForceFixed() = default;

```

```

virtual ~TestScoreForceFixed() = default;
TestScoreForceFixed(TestScoreForceFixed const&) = default;
TestScoreForceFixed& operator=(TestScoreForceFixed const&) = delete;
TestScoreForceFixed& operator=(TestScoreForceFixed&&) = default;

using ScoreAll_t = TestScore::ScoreAll_t;
using ScoreOne_t = TestScore::ScoreOne_t;

using TestScore::AddScore;
using TestScore::GetScore;

using TestScore::begin;
using TestScore::end;

private:
    // int32_t score: 0~100はスコア、それ以外は不正データ
    virtual void validate_score(int32_t score) const override;
};

std::string ToString(TestScore const& ts);
void LoadCSV(std::string const& filename, TestScore& test_score);
void StoreCSV(TestScore const& test_score, std::string const& filename);
TestScore::ScoreOne_t Average(TestScore const& test_score);

```

```

// @@@ exercise/solid_a/lsp_test_score.cpp 10

...
void TestScoreForceFixed::validate_score(int32_t score) const
{
    if (is_valid_score(score)) {
        return;
    }

    throw std::out_of_range{"Invalid Score"};
}
...
```

```

// @@@ exercise/solid_a/lsp_test_score_ut.cpp 15

namespace {

...
// [A]
// ファイルtest_score_org_には、TestScoreForceの不正値が含まれているため、
// 下記の単体テストでの、
//     LoadCSV(test_score_org_, ts_f);
// はエクセプションが発生するべきだが、実際にはバシしてしまう。
// 一方で、エクセプションが発生するようにLoadCSVを変更するには、LoadCVSの第2引数の
// ランタイム時の実際の型が必要になってしまうため、この解決手段にも問題がある。
```

```

TEST_F(SolidLSP_A, TestScoreForce_LoadCSV)
{
    auto ts_f = TestScoreForce{};
    LoadCSV(test_score_org_, ts_f); // 本来はエクセプションが発生すべき。

    auto const exp = TestScore::ScoreAll_t{
        TestScore::ScoreAll_t::value_type("豈林", {-1, 50, 80}),
        TestScore::ScoreAll_t::value_type("広輔", {40, 30, 10}),
        TestScore::ScoreAll_t::value_type("會澤", {30, 60, 70}),
        TestScore::ScoreAll_t::value_type("松山", {80, 90, 50}),
        TestScore::ScoreAll_t::value_type("菊池", {50, 20, 80}),
        TestScore::ScoreAll_t::value_type("鈴木", {0, 80, 100}),
    };

    ASSERT_TRUE(std::equal(ts_f.begin(), ts_f.end(), exp.begin()));

    auto const one_score = TestScore::ScoreOne_t{
        TestScore::ScoreOne_t::value_type("豈林", 50),
        TestScore::ScoreOne_t::value_type("広輔", 40),
        TestScore::ScoreOne_t::value_type("會澤", 70),
        TestScore::ScoreOne_t::value_type("松山", 1),
        TestScore::ScoreOne_t::value_type("菊池", -1),
        TestScore::ScoreOne_t::value_type("鈴木", 5),
        TestScore::ScoreOne_t::value_type("田中", 100),
        TestScore::ScoreOne_t::value_type("西川", 90),
    };
}
```

```

std::pair<std::string, std::vector<int32_t>> parse_line(std::string const& line)
{
    auto const csv_sep = std::regex{R"( *, *)"};
    auto name      = std::string{};
    auto score     = std::vector<int32_t>{};

    auto end = std::sregex_token_iterator{};
    for (auto it = std::sregex_token_iterator{line.begin(), line.end(), csv_sep, -1}; it != end;
         ++it) {
        if (name.length() == 0) {
            name = *it;
        }
        else {
            auto s = std::stoi(*it);
            score.emplace_back(s);
        }
    }
}

return {std::move(name), std::move(score)};
}

template <typename TEST_SCORE>
TEST_SCORE LoadCSV(std::string const& filename)
{
    static_assert(
        std::is_same_v<TEST_SCORE, TestScore> || std::is_same_v<TEST_SCORE, TestScoreForceFixed>);

    auto data = std::ifstream{filename};

    typename TEST_SCORE::ScoreAll_t test_score_raw;
    auto                      line = std::string{};
    while (std::getline(data, line)) {
        // std::pair<TestScore::ScoreAll_t::iterator, bool>
        auto ret = test_score_raw.insert(parse_line(line));
        assert(ret.second);
    }

    typename TEST_SCORE::ScoreOne_t one_test;
    for (auto const& pair : test_score_raw) {
        one_test.emplace_back(std::make_pair(pair.first, 0));
    }

    auto const score_count = test_score_raw.begin()->second.size();
    auto      ts          = TEST_SCORE{};

    for (auto i = 0U; i < score_count; ++i) {
        for (auto& pair : one_test) {
            pair.second = test_score_raw[pair.first][i];
        }
        ts.AddScore(one_test);
    }

    return ts;
}

TEST_F(SolidLSP_A, TestScoreForceFixed_LoadCSV)
{
    // 下で使用しているLoadCSVはtemplateで実装し直したもの(上記template <>LoadCSV)。
    // 従来のLoadCSVでは型違いでコンパイルできない。
    // また、従来のLoadCSVは第2引数をTestScore&をしたため、
    // ここで指摘したような問題を引き起こしやすい。この問題を解決するため、
    // template <>LoadCSVは引数での値戻しをやめ、リターンでの値戻しに改めた。
    ASSERT_THROW(LoadCSV<TestScoreForceFixed>(test_score_org_), std::out_of_range);

    TestScoreForceFixed ts_f_f = LoadCSV<TestScoreForceFixed>(test_score_org_f_);

    auto const exp = TestScore::ScoreAll_t{
        TestScore::ScoreAll_t::value_type("堂林", {0, 50, 80}),
        TestScore::ScoreAll_t::value_type("広輔", {40, 30, 10}),
        TestScore::ScoreAll_t::value_type("會澤", {30, 60, 70}),
        TestScore::ScoreAll_t::value_type("松山", {80, 90, 50}),
        TestScore::ScoreAll_t::value_type("菊池", {50, 20, 80}),
        TestScore::ScoreAll_t::value_type("鈴木", {0, 80, 100}),
    };

    ASSERT_TRUE(std::equal(ts_f_f.begin(), ts_f_f.end(), exp.begin()));
}
} // namespace

```

- 演習-LSPへ戻る。

解答例-ISP

```
// @@@ exercise/solid_a/isp_test_score_average.h 8
// [A]
// 下記クラスTestScoreの管理データの内、受験者とその平均スコア、
// 平均スコアの高い順でソートされた受験者リストを扱うクラスが必要になったため、
// 下記のようにイミュータブルなクラスTestScoreAverageを作成した。
//
// 現在のファイル構成では、TestScoreAverageのみを使うクラスや関数にも、
// このファイル全体への依存を強いる(つまり、TestScoreやLoadCSV等に依存させる)ため、
// ISPに反する。
// TestScoreAverageを使うクラスや関数に余計な依存関係が発生しないようにリファクタリングを
// 行え。
//
// [解説]
// TestScoreAverageの宣言・定義をsolid_isp_test_score_a.hからこのファイルに移動し、
// TestScoreAverageのTestScore依存部分をTestScoreAverageDataで隠蔽することで、
// このファイルからのsolid_isp_test_score_a.hの依存を消した。
// また、TestScoreAverageのTestScore依存部はすべてsolid_isp_test_score_average_a.cppに移動した。
// これにより、TestScoreAverageの利用者はTestScoreに依存しなくなった。
//

class TestScoreAverage {
public:
    explicit TestScoreAverage(std::string const& filename);
    ~TestScoreAverage();
    uint32_t GetAverage(std::string const& name) const;
    std::vector<std::string> const& DescendingOrder() const;

private:
    struct TestScoreAverageData;
    std::unique_ptr<TestScoreAverageData> const data_;
};
```

```
// @@@ exercise/solid_a/isp_test_score.h 8

class TestScore {
    ...
};

std::string ToString(TestScore const& ts);
TestScore LoadCSV(std::string const& filename);
void StoreCSV(TestScore const& test_score, std::string const& filename);
TestScore::ScoreOne_t Average(TestScore const& test_score);
```

```
// @@@ exercise/solid_a/isp_test_score_average.cpp 10

namespace {

TestScore::ScoreOne_t get_average(std::string const& filename)
{
    TestScore ts = LoadCSV(filename);

    return Average(ts);
}
} // namespace

struct TestScoreAverage::TestScoreAverageData {
    TestScoreAverageData(TestScore::ScoreOne_t& average) : average{std::move(average)} {}

    TestScore::ScoreOne_t const average;
    std::vector<std::string> desending_order{};
};

TestScoreAverage::TestScoreAverage(std::string const& filename)
: data_{std::make_unique<TestScoreAverage::TestScoreAverageData>(get_average(filename))}

TestScoreAverage::~TestScoreAverage() = default; // これはヘッダには書けない

uint32_t TestScoreAverage::GetAverage(std::string const& name) const
{
    auto pos = std::find_if(data_->average.cbegin(), data_->average.cend(),
                           [&name](std::pair<std::string, int32_t> const& pair) noexcept {
                               return name == pair.first;
                           });

    if (pos == data_->average.cend()) {
        throw std::out_of_range{"no member"};
    }
}
```

```

    }

    return pos->second;
}

std::vector<std::string> const& TestScoreAverage::DescendingOrder() const
{
    if (data_->desending_order.size() != 0) {
        return data_->desending_order;
    }

    auto ave = data_->average;
    std::sort(ave.begin(), ave.end(),
              [] (std::pair<std::string, int32_t> const& lhs, auto const& rhs) noexcept {
                  return lhs.second > rhs.second;
              });

    for (auto& pair : ave) {
        data_->desending_order.emplace_back(std::move(pair.first));
    }

    return data_->desending_order;
}

```

```
// @@@ exercise/solid_a/isp_test_score.cpp 10
```

```
// 演習コードと同一であるため省略
```

```
...
```

```
// @@@ exercise/solid_a/isp_test_score_average_ut.cpp 13
```

```
namespace {

TEST_F(SolidISP_A, TestScoreAverage)
{
    auto tsa = TestScoreAverage{test_score_org_};

    ASSERT_EQ(tsa.GetAverage("堂林"), 65);
    ASSERT_EQ(tsa.GetAverage("広輔"), 26);
    ASSERT_EQ(tsa.GetAverage("會澤"), 53);
    ASSERT_EQ(tsa.GetAverage("松山"), 73);
    ASSERT_EQ(tsa.GetAverage("菊池"), 50);
    ASSERT_EQ(tsa.GetAverage("鈴木"), 60);

    ASSERT_THROW(tsa.GetAverage("野村"), std::out_of_range);

    auto const exp = std::vector<std::string>{
        "松山", "堂林", "鈴木", "會澤", "菊池", "広輔",
    };

    ASSERT_EQ(tsa.DescendingOrder(), exp);
    ASSERT_EQ(tsa.DescendingOrder(), exp); // キャッシュのテスト
}
} // namespace
```

```
// @@@ exercise/solid_a/isp_test_score_ut.cpp 13
```

```
// 演習コードと同一であるため省略
```

```
...
```

- 演習-ISPへ戻る。

解答例-DIP

```
// @@@ exercise/solid_a/dip_test_score.h 9
// [A]
// クラスTestScoreClientは、
//     * dip_test_score_client.h
//     * dip_test_score_client.cpp
// で宣言・定義され、
// クラスTestScoreLoaderは、
//     * dip_test_score.h(このファイル)
//     * dip_test_score.cpp
// で宣言・定義されされている。
// TestScoreLoaderは宣言・定義の中にTestScoreClientを使用しているため、
//     * dip_test_score.cpp -> dip_test_score_client.h
// の依存関係が発生してる(dip_test_score.h -> dip_test_score_client.hの依存関係は、
// dip_test_score.h内のTestScoreClientの前方宣言で回避)。
// クラスの名前からもわかる通り、
```

```

//      * TestScoreClientはTestScoreLoaderのクライアント
//      * TestScoreLoaderはTestScoreClientのサーバ
// であるため、この依存関係
//      * TestScoreLoader -> TestScoreClient(逆の依存関係もあるため、双向依存)
//      * dip_test_score.cpp -> dip_test_score_client.h
// (はDIPに反し、機能拡張(や、場合によっては単体テスト可能なパッケージ構成維持)
// に多大な悪影響がある(TestScoreLoaderを使うTestScoreClient2を新たに定義したときに
// TestScoreLoaderがどのように修正されるかを考えればこの問題に気づくだろう)。
// この問題に対処せよ。

// [解説]
// TestScoreLoaderを「TestScoreClientへの依存」から「TestScoreClientIFへの依存」に変更し、
// TestScoreClientIFをTestScoreLoaderと同じファイル(このファイル)で宣言・定義したことにより、
//      * TestScoreLoader -> TestScoreClient
// の依存関係は解消された。
// TestScoreClientは、TestScoreClientIFを継承することでTestScoreLoaderのサービスを使用できる。
// この依存関係は、
//      * TestScoreClient -> TestScoreClientIF
// であり、クライアントからサーバへの依存であるため問題にならない。

class TestScore {
    ...
};

...
class TestScoreClientIF {
public:
    TestScoreClientIF() = default;
    virtual ~TestScoreClientIF() = default;
    virtual void Done() = 0;
};

class TestScoreLoader {
public:
    TestScoreLoader() {}
    ~TestScoreLoader();
    void LoadCSV_Async(std::string&& filename, TestScoreClientIF& client);
    TestScore LoadCSV_Get() { return future_.get(); }

private:
    std::future<TestScore> future_{};
};

```

```

// @@@ exercise/solid_a/dip_test_score.cpp 10

...
void TestScoreLoader::LoadCSV_Async(std::string&& filename, TestScoreClientIF& client)
{
    if (future_.valid()) {
        future_.get();
    }

    future_ = std::async(std::launch::async, [&client, filename = std::move(filename)]() {
        auto test_score = LoadCSV(filename);
        client.Done();
        return test_score;
    });
}

```

```

// @@@ exercise/solid_a/dip_test_score_ut.cpp 13

// 演習コードと同一であるため省略
...

```

```

// @@@ exercise/solid_a/dip_test_score_client.h 11

class TestScoreClient : public TestScoreClientIF {
public:
    void LoadAsync(std::string&& filename);
    virtual void Done() override;
    void Wait();
    TestScore const& GetTestScore() const noexcept { return test_score_; }

private:
    std::condition_variable condition_{};
    std::mutex mutex_{};
    TestScore test_score_{};
    TestScoreLoader loader_{};
};

```

```

        bool           loaded_{false};
};

// @@@ exercise/solid_a/dip_test_score_client.cpp 5
// 演習コードと同一であるため省略
...

// @@@ exercise/solid_a/dip_test_score_client_ut.cpp 9
// 演習コードと同一であるため省略
...

```

- 演習-DIPへ戻る。

解答-SOLIDの定義

- 選択肢対応
 - SRP - 1
 - OCP - 2
 - LIP - 3
 - ISP - 4
 - DIP - 5
- 参照 SOLID
- 演習-SOLIDの定義へ戻る。

デザインパターン

解答例-ガード節

```

// @@@ exercise/design_pattern_a/guard.cpp 7
// [A]
// 以下の関数PrimeNumbersをガード節や、関数の括りだし等によってリファクタリングせよ。

inline uint32_t next_prime_num(uint32_t curr_prime_num, std::vector<bool>& is_num_prime) noexcept
{
    for (auto i = 2 * curr_prime_num; i < is_num_prime.size(); i += curr_prime_num) {
        is_num_prime[i] = false; // 次の倍数は素数ではない
    }

    auto prime_num = curr_prime_num;

    do { // 次の素数の探索
        ++prime_num;
    } while (!is_num_prime[prime_num] && (prime_num < is_num_prime.size()));

    return prime_num;
}

inline std::vector<uint32_t> prime_numbers(uint32_t max_number)
{
    auto result      = std::vector<uint32_t>{};
    auto prime_num   = 2U;                                // 最初の素数
    auto is_num_prime = std::vector<bool>(max_number + 1, true); // falseなら素数でない
    is_num_prime[0] = is_num_prime[1] = false;

    do {
        result.emplace_back(prime_num);
        prime_num = next_prime_num(prime_num, is_num_prime);
    } while (prime_num < is_num_prime.size());

    return result;
}

std::optional<std::vector<uint32_t>> PrimeNumbers(uint32_t max_number)
{
    if (max_number >= 65536) { // ガード節。演算コストが高いためエラーにする
        return std::nullopt;
    }

    if (max_number < 2) { // ガード節。2未満の素数はない
        return std::vector<uint32_t>{};
    }

    return prime_numbers(max_number);
}

```

```

}

TEST(DesignPatternA, Guard)
{
    auto result = PrimeNumbers(1);
    ASSERT_TRUE(result);
    ASSERT_EQ((std::vector<uint32_t>{}), *result);

    result = PrimeNumbers(2);
    ASSERT_TRUE(result);
    ASSERT_EQ((std::vector<uint32_t>{2}), *result);

    result = PrimeNumbers(30);
    ASSERT_TRUE(result);
    ASSERT_EQ((std::vector<uint32_t>{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}), *result);

    ASSERT_FALSE(PrimeNumbers(65536));
}

```

- 演習-ガード節へ戻る。

解答例-BitmaskType

```

// @@@ exercise/design_pattern_a/enum_bitmask.cpp 5
// [A]
// 下記関数ColorMask2Strはuint32_t型のビットマスクを引数に取る。
// これはユーザが使用間違いを起こしやすい脆弱なインターフェースである。
// enumによるビットマスク表現を使用しこの問題に対処せよ。

enum class Color : uint32_t {
    RED      = 0b0001,
    YELLOW   = 0b0010,
    GREEN    = 0b0100,
    BLUE     = 0b1000,
};

constexpr Color operator&(Color x, Color y) noexcept
{
    return static_cast<Color>(static_cast<uint32_t>(x) & static_cast<uint32_t>(y));
}

constexpr Color operator|(Color x, Color y) noexcept
{
    return static_cast<Color>(static_cast<uint32_t>(x) | static_cast<uint32_t>(y));
}

Color& operator&=(Color& x, Color y) noexcept { return x = x & y; }
Color& operator|=(Color& x, Color y) noexcept { return x = x | y; }

constexpr bool IsTrue(Color x) noexcept { return static_cast<bool>(x); }

std::string ColorMask2Str(Color color)
{
    auto ret = std::string{};

    if (IsTrue(Color::RED & color)) {
        ret += "RED";
    }
    if (IsTrue(Color::YELLOW & color)) {
        if (ret.size() != 0) {
            ret += ',';
        }
        ret += "YELLOW";
    }
    if (IsTrue(Color::GREEN & color)) {
        if (ret.size() != 0) {
            ret += ',';
        }
        ret += "GREEN";
    }
    if (IsTrue(Color::BLUE & color)) {
        if (ret.size() != 0) {
            ret += ',';
        }
        ret += "BLUE";
    }

    return ret;
}

```

```

TEST(DesignPatternA, EnumBitmask)
{
    ASSERT_EQ("RED", ColorMask2Str(Color::RED));
    ASSERT_EQ("RED,YELLOW", ColorMask2Str(Color::RED | Color::YELLOW));
    ASSERT_EQ("YELLOW", ColorMask2Str(Color::YELLOW));
    ASSERT_EQ("YELLOW,GREEN,BLUE", ColorMask2Str(Color::YELLOW | Color::GREEN | Color::BLUE));

    auto c = Color::GREEN;
    ASSERT_EQ("GREEN", ColorMask2Str(c));

    c |= Color::RED;
    ASSERT_EQ("RED,GREEN", ColorMask2Str(c));

    c &= Color::RED;
    ASSERT_EQ("RED", ColorMask2Str(c));

#ifndef 0 // 間違った使い方はコンパイルさせない
    ASSERT_EQ("", ColorMask2Str(0b1000)); // 想定していない使用法
#endif
}

```

- 演習-BitmaskTypeへ戻る。

解答例-Pimpl

```

// @@@ exercise/design_pattern_a/pimpl.cpp 5
// [A] 下記クラスCollectionの宣言はクラスWidgetの宣言に依存している。
// Pimplパターンを使用し、Collectionの宣言がWidgetの宣言に依存しないようにせよ。

class Widget;
class Collection {
public:
    Collection();
    char const* Name(size_t i) const;
    void AddName(char const* name);
    size_t Count() const noexcept;

private:
    class Pimpl;
    std::unique_ptr<Pimpl> pimpl_{};

};

TEST(DesignPatternA, Pimpl)
{
    auto c = Collection{};

    ASSERT_EQ(0, c.Count());
    ASSERT_THROW(c.Name(0), std::out_of_range);

    c.AddName("n0");
    c.AddName("n1");
    c.AddName("n2");

    ASSERT_EQ(3, c.Count());
    ASSERT_STREQ("n0", c.Name(0));
    ASSERT_STREQ("n1", c.Name(1));
    ASSERT_STREQ("n2", c.Name(2));
    ASSERT_THROW(c.Name(4), std::out_of_range);
}

class Widget {
public:
    explicit Widget(char const* name) : name_{name} {}
    char const* Name() const noexcept { return name_; }

private:
    char const* name_;
};

class Collection::Pimpl {
public:
    char const* Name(size_t i) const { return widgets_.at(i).Name(); }
    void AddName(char const* name) { widgets_.emplace_back(name); }

    size_t Count() const noexcept { return widgets_.size(); }

private:
    std::vector<Widget> widgets_{};
}

```

```

};

Collection::Collection() : pimpl_{std::make_unique<Collection::Pimpl>()} {}
char const* Collection::Name(size_t i) const { return pimpl_->Name(i); }
void Collection::AddName(char const* name) { pimpl_->AddName(name); }
size_t Collection::Count() const noexcept { return pimpl_->Count(); }

```

- 演習-Pimplへ戻る。

解答-Accessorの副作用

- 選択肢1
- 参照 Accessor
- 演習-Accessorの副作用へ戻る。

解答例-Accessor

```

// @@@ exercise/design_pattern_a/Accessor.cpp 5
// [A]
// 下記クラスPrimeNumbersはAccessorの多用により、クラスのカプセル化が破壊されている例である。
// これにより、このクラスは凝集性が低く、誤用を誘発しやすい。
// この問題を解決するため、クラスPrimeNumbersや関数GetPrimeNumbersを修正せよ。
// また、別の問題があれば合わせて修正せよ。

// [解説]
// * Accessorについて
//   * HasCache, Cashedを廃止。
//   値(この場合は素数列prime_numbers_)をキャッシュしているかどうかの判断は、
//   クラスに考えさせるべき。
//   * SetMaxNumberの変更。
//     SetMaxNumberで素数列を作つても良いが、一般に重い計算はなるべく遅延実行させた方が良い。
// * その他の変更について
//   * GetPrimeNumbersのGeneratePrimeNumbersへの名前変更。
//     GetXXXはconstメンバ関数にすべきなので。
//   * コンストラクタやcopy代入演算子等の自動生成関数は、何らかの定義・宣言をした方が良い。
//   * メンバ変数は必ず初期化する。

class PrimeNumbers {
public:
    PrimeNumbers() = default;

    PrimeNumbers(PrimeNumbers const&) = default;
    PrimeNumbers& operator=(PrimeNumbers const&) = default;

    uint32_t GetMaxNumber() const noexcept { return max_number_; }
    void SetMaxNumber(uint32_t max_number) noexcept
    {
        if (max_number != max_number_) {
            cached_ = false;
            max_number_ = max_number;
        }
    }

    std::vector<uint32_t> const& GeneratePrimeNumbers();

private:
    uint32_t max_number_{0};
    bool cached_{false};
    std::vector<uint32_t> prime_numbers_{};

    static uint32_t next_prime_num(uint32_t curr_prime_num,
                                  std::vector<bool>& is_num_prime) noexcept;
    static std::vector<uint32_t> get_prime_numbers(uint32_t max_number);
};

uint32_t PrimeNumbers::next_prime_num(uint32_t curr_prime_num,
                                      std::vector<bool>& is_num_prime) noexcept
{
    for (auto i = 2 * curr_prime_num; i < is_num_prime.size(); i += curr_prime_num) {
        is_num_prime[i] = false; // 次の倍数は素数ではない
    }

    auto prime_num = curr_prime_num;

    do { // 次の素数の探索
        ++prime_num;
    } while (!is_num_prime[prime_num] && (prime_num < is_num_prime.size()));
}

```

```

        return prime_num;
    }

    inline std::vector<uint32_t> PrimeNumbers::get_prime_numbers(uint32_t max_number)
    {
        auto result      = std::vector<uint32_t>{};
        auto prime_num   = 2U;                                // 最初の素数
        auto is_num_prime = std::vector<bool>(max_number + 1, true); // falseなら素数でない。
        is_num_prime[0] = is_num_prime[1] = false;

        do {
            result.emplace_back(prime_num);
            prime_num = next_prime_num(prime_num, is_num_prime);
        } while (prime_num < is_num_prime.size());

        return result;
    }

    std::vector<uint32_t> const& PrimeNumbers::GeneratePrimeNumbers()
    {
        if (cached_) {
            return prime_numbers_;
        }

        if (max_number_ < 2) { // ガード節。2未満の素数はない。
            prime_numbers_.clear();
        }
        else {
            prime_numbers_ = get_prime_numbers(max_number_);
        }

        cached_ = true;
        return prime_numbers_;
    }

TEST(DesignPatternA, Accessor)
{
    auto pm = PrimeNumbers{};

    pm.SetMaxNumber(1);
    ASSERT_EQ((std::vector<uint32_t>{}), pm.GeneratePrimeNumbers());

    pm.SetMaxNumber(3);
    ASSERT_EQ((std::vector<uint32_t>{2, 3}), pm.GeneratePrimeNumbers());

    pm.SetMaxNumber(30);
    ASSERT_EQ((std::vector<uint32_t>{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}),
              pm.GeneratePrimeNumbers());

#if 0 // このテストパターンは廃止した
    pm.SetMaxNumber(3);
    GeneratePrimeNumbers(pm); // pm.Cashed(false);しないので前のまま。
                            // このような用途は考えづらいので、おそらく仕様のバグ。

    ASSERT_EQ((std::vector<uint32_t>{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}), pm.GeneratePrimeNumbers());
#else
    ASSERT_EQ((std::vector<uint32_t>{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}),
              pm.GeneratePrimeNumbers());
#endif

    pm.SetMaxNumber(3);
    ASSERT_EQ((std::vector<uint32_t>{2, 3}), pm.GeneratePrimeNumbers());

    auto pm3_copy = PrimeNumbers{pm};
    ASSERT_EQ((std::vector<uint32_t>{2, 3}), pm3_copy.GeneratePrimeNumbers());

    auto pm5 = PrimeNumbers{};
    pm5.SetMaxNumber(5);
    pm = pm5;
    ASSERT_EQ((std::vector<uint32_t>{2, 3, 5}), pm.GeneratePrimeNumbers());
}

```

- 演習-Accessorへ戻る。

解答例-Copy-And-Swap

```
// @@@ exercise/design_pattern_a/copy_and_swap.cpp 5
// [A]
// 以下のクラスCopyAndSwapの
// * copyコンストラクタ
// * copy代入演算子
// * moveコンストラクタ
// * move代入演算子
// をCopy-And-Swapイデオムを使用して実装し、単体テストを行え。

class CopyAndSwap final {
public:
    explicit CopyAndSwap(char const* name0, char const* name1)
        : name0_{name0 == nullptr ? "" : name0}, name1_{name1 == nullptr ? "" : name1}
    {
    }

    CopyAndSwap(CopyAndSwap const& rhs) : name0_{rhs.name0_}, name1_{rhs.name1_} {}

    CopyAndSwap(CopyAndSwap&& rhs) noexcept
        : name0_{std::exchange(rhs.name0_, nullptr)}, name1_{std::move(rhs.name1_)}
    {
    }

    CopyAndSwap& operator=(CopyAndSwap const& rhs)
    {
        if (this == &rhs) {
            return *this;
        }

        // copyコンストラクタの使用
        auto tmp = CopyAndSwap{rhs}; // ここでエクセプションが発生しても、tmp以外、壊れない

        Swap(tmp);

        return *this;
    }

    CopyAndSwap& operator=(CopyAndSwap&& rhs) noexcept
    {
        if (this == &rhs) {
            return *this;
        }

        auto tmp = CopyAndSwap{std::move(rhs)}; // moveコンストラクタ

        Swap(tmp);

        return *this;
    }

    void Swap(CopyAndSwap& rhs) noexcept
    {
        std::swap(name0_, rhs.name0_);
        std::swap(name1_, rhs.name1_);
    }

    char const* GetName0() const noexcept { return name0_; }

    std::string const& GetName1() const noexcept { return name1_; }

    ~CopyAndSwap() = default;

private:
    char const* name0_;
    std::string name1_;
};

#if defined(__clang__)
    // clangコンパイルでの警告抑止
#define SUPPRESS_WARN_CLANG_BEGIN _Pragma("clang diagnostic push")
#define SUPPRESS_WARN_CLANG_SELF_ASSIGN_OVERLOADED \
    _Pragma("clang diagnostic ignored \"-Wself-assign-overloaded\"")
#define SUPPRESS_WARN_CLANG_SELF_MOVE _Pragma("clang diagnostic ignored \"-Wself-move\"")
#define SUPPRESS_WARN_CLANG_END _Pragma("clang diagnostic pop")
#else
#define SUPPRESS_WARN_CLANG_BEGIN
#define SUPPRESS_WARN_CLANG_SELF_ASSIGN_OVERLOADED
#define SUPPRESS_WARN_CLANG_SELF_MOVE
#define SUPPRESS_WARN_CLANG_END

```

```

#endif

// 本来は下記単体テストは分割すべきだが、紙面の都合上一つにまとめる。
TEST(DesignPatternA, CopyAndSwap)
{
    // test for explicit CopyAndSwap(char const* name0, char const* name1)
    auto n = CopyAndSwap(nullptr, nullptr);
    ASSERT_STREQ("", n.GetName0());
    ASSERT_EQ("", n.GetName1());

    auto a = CopyAndSwap("a0", "a1");
    ASSERT_STREQ("a0", a.GetName0());
    ASSERT_EQ("a1", a.GetName1());

    // test for void Swap(CopyAndSwap& rhs) noexcept
    auto b = CopyAndSwap("b0", "b1");

    a.Swap(b);
    ASSERT_STREQ("b0", a.GetName0());
    ASSERT_EQ("b1", a.GetName1());
    ASSERT_STREQ("a0", b.GetName0());
    ASSERT_EQ("a1", b.GetName1());

    a.Swap(a);
    ASSERT_STREQ("b0", a.GetName0());
    ASSERT_EQ("b1", a.GetName1());

    // test for CopyAndSwap(CopyAndSwap const& rhs)
    auto const const_a = CopyAndSwap("const_a0", "const_a1");

    auto b_copy = CopyAndSwap(const_a);

    ASSERT_STREQ("const_a0", b_copy.GetName0());
    ASSERT_EQ("const_a1", b_copy.GetName1());

    // test for CopyAndSwap& operator=(CopyAndSwap const& rhs)
    auto const c = CopyAndSwap("c0", "c1");

    b_copy = c;
    ASSERT_STREQ("c0", b_copy.GetName0());
    ASSERT_EQ("c1", b_copy.GetName1());

    SUPPRESS_WARN_CLANG_BEGIN;
    SUPPRESS_WARN_CLANG_SELF_ASSIGN_OVERLOADED;

    b_copy = b_copy;

    SUPPRESS_WARN_CLANG_END;

    ASSERT_STREQ("c0", b_copy.GetName0());
    ASSERT_EQ("c1", b_copy.GetName1());

    // test for CopyAndSwap(CopyAndSwap&& rhs) noexcept
    auto b_move = CopyAndSwap(std::move(b));

    ASSERT_STREQ("a0", b_move.GetName0());
    ASSERT_EQ("a1", b_move.GetName1());

#ifndef __clang_analyzer__ // move後のオブジェクトにリードアクセスするとscan-buildでエラー
    ASSERT_EQ(nullptr, b.GetName0());
    ASSERT_EQ("", b.GetName1());
#endif

    auto c_move = CopyAndSwap(std::move(const_a)); // moveに見えるが実はコピー

    ASSERT_STREQ("const_a0", const_a.GetName0());
    ASSERT_EQ("const_a1", const_a.GetName1());

    ASSERT_STREQ("const_a0", c_move.GetName0());
    ASSERT_EQ("const_a1", c_move.GetName1());

    // test for CopyAndSwap& operator=(CopyAndSwap&& rhs) noexcept
    c_move = std::move(b_move);
    ASSERT_STREQ("a0", c_move.GetName0());
    ASSERT_EQ("a1", c_move.GetName1());

#ifndef __clang_analyzer__ // move後のオブジェクトにリードアクセスするとscan-buildでエラー
    ASSERT_EQ(nullptr, b_move.GetName0());
    ASSERT_EQ("", b_move.GetName1());
#endif
}

```

```

SUPPRESS_WARN_CLANG_BEGIN;
SUPPRESS_WARN_CLANG_SELF_MOVE;

c_move = std::move(c_move);

SUPPRESS_WARN_CLANG_END;

ASSERT_STREQ("a0", c_move.GetName0());
ASSERT_EQ("a1", c_move.GetName1());
}

```

- 演習-Copy-And-Swapへ戻る。

解答例-Immutable

```

// @@@ exercise/design_pattern_a/immutable.cpp 5
// [A]
// 下記クラスPrimeNumbersはSetMaxNumberにより状態が変わってしまうことがある。
// 状態変更が必要ない場合、こういった仕様はない方が良い。
// PrimeNumbersからSetMaxNumberを削除し、このクラスをimmutableにせよ。

// [解説]
// * Immutableなクラスとは、生成後状態が変えられないクラスである。
// * PrimeNumbersをImmutableにするには
//   * SetMaxNumberを廃止し、メンバ変数をconstにする。
//   * GeneratePrimeNumbersをconstメンバ関数にして、GetPrimeNumbersに変更する。
//   * copy代入演算子を =deleteする(これがなくてもconstメンバならばコピーはできないが)

class PrimeNumbers {
public:
    explicit PrimeNumbers(uint32_t max_number = 2);

    PrimeNumbers(PrimeNumbers const&) = default;
    PrimeNumbers& operator=(PrimeNumbers const&) = delete; // constメンバはコピーできない。

    uint32_t GetMaxNumber() const noexcept { return max_number_; }
    std::vector<uint32_t> const& GetPrimeNumbers() const noexcept { return prime_numbers_; }

private:
    uint32_t const max_number_;
    std::vector<uint32_t> const prime_numbers_{};

    static uint32_t next_prime_num(uint32_t curr_prime_num,
                                  std::vector<bool>& is_num_prime) noexcept;
    static std::vector<uint32_t> get_prime_numbers(uint32_t max_number);
};

uint32_t PrimeNumbers::next_prime_num(uint32_t curr_prime_num,
                                      std::vector<bool>& is_num_prime) noexcept
{
    for (auto i = 2 * curr_prime_num; i < is_num_prime.size(); i += curr_prime_num) {
        is_num_prime[i] = false; // 次の倍数は素数ではない
    }

    auto prime_num = curr_prime_num;

    do { // 次の素数の探索
        ++prime_num;
    } while (!is_num_prime[prime_num] && (prime_num < is_num_prime.size()));

    return prime_num;
}

std::vector<uint32_t> PrimeNumbers::get_prime_numbers(uint32_t max_number)
{
    auto result = std::vector<uint32_t>{};

    if (max_number < 2) { // ガード節。2未満の素数はない。
        return result;
    }

    auto prime_num = 2U; // 最初の素数
    auto is_num_prime = std::vector<bool>(max_number + 1, true); // falseなら素数でない。
    is_num_prime[0] = is_num_prime[1] = false;

    do {
        result.emplace_back(prime_num);
        prime_num = next_prime_num(prime_num, is_num_prime);
    }
}

```

```

    } while (prime_num < is_num_prime.size());

    return result;
}

PrimeNumbers::PrimeNumbers(uint32_t max_number)
: max_number_{max_number}, prime_numbers_{get_prime_numbers(max_number)}
{
}

TEST(DesignPatternA, Immutable)
{
    auto pm1 = PrimeNumbers{1};
    ASSERT_EQ((std::vector<uint32_t>{}), pm1.GetPrimeNumbers());

    auto pm3 = PrimeNumbers{3};
    ASSERT_EQ((std::vector<uint32_t>{2, 3}), pm3.GetPrimeNumbers());

    auto pm30 = PrimeNumbers{30};
    ASSERT_EQ((std::vector<uint32_t>{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}), pm30.GetPrimeNumbers());

    ASSERT_EQ((std::vector<uint32_t>{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}), pm30.GetPrimeNumbers());

    auto pm3_copy = PrimeNumbers{pm3};
    ASSERT_EQ((std::vector<uint32_t>{2, 3}), pm3_copy.GetPrimeNumbers());

#if 0 // immutableなのでコピーはできない。
    auto pm = PrimeNumbers{1};
    auto pm5 = PrimeNumbers{5};
    pm = pm5;
    ASSERT_EQ((std::vector<uint32_t>{2, 3, 5}), pm.GetPrimeNumbers());
#endif
}

```

- 演習-Immutableへ戻る。

解答例-Clone

```

// @@@ exercise/design_pattern_a/clone.cpp 7
// [A]
// TEST(DesignPatternQ, Clone)に記述したように、オブジェクトのスライシングによる影響で、
// Base型ポインタに代入されたDerivedインスタンスへのコピーは部分的にしか行われない。
// Cloneパターンを使用してこの問題を修正せよ。
// また、その他の問題があれば合わせて修正せよ。

class Base {
public:
    explicit Base(std::string name) : name1_{std::move(name)} {}
    virtual ~Base() = default;
    virtual std::string const& GetName() const noexcept { return name1_; }

    virtual std::unique_ptr<Base> Clone() const { return std::make_unique<Base>(name1_); }

    Base(Base const&) = delete;
    Base& operator=(Base const&) = delete;

private:
    std::string name1_;
};

class Derived final : public Base {
public:
    explicit Derived(std::string name1 = "", std::string name2 = "")
        : Base{std::move(name1)}, name2_{std::move(name2)}
    {
    }
    virtual ~Derived() override = default;
    virtual std::string const& GetName() const noexcept override { return name2_; }

    virtual std::unique_ptr<Base> Clone() const override { return CloneOwn(); }

    std::unique_ptr<Derived> CloneOwn() const
    {
        return std::make_unique<Derived>(Base::GetName(), name2_);
    }

private:
    std::string name2_;
};

```

```

TEST(DesignPatternA, Clone)
{
    Derived d1{"name1", "name2"};

    ASSERT_EQ("name1", d1.Base::GetName());
    ASSERT_EQ("name2", d1.GetName());

    std::unique_ptr<Derived> d2 = d1.CloneOwn();
    ASSERT_EQ("name1", d2->Base::GetName());
    ASSERT_EQ("name2", d2->GetName());

    std::unique_ptr<Base> b3 = d1.Clone(); // コピーの代わりにクローン

    ASSERT_EQ("name1", b3->Base::GetName());
    ASSERT_EQ("name2", b3->GetName()); // ちゃんとコピーされた。
}

```

- 演習(Clone)へ戻る。

解答例-NVI

```

// @@@ exercise/design_pattern_a/nvi.cpp 7
// [A]
// 下記クラスBase、Derived、DerivedDerivedの前処理はクローンコードになっている。
// NVIを用いて、この問題に対処せよ。

class Base {
public:
    explicit Base(std::string name) : name1_{name} {}
    virtual ~Base() = default;
    std::string const& GetName1() const noexcept { return name1_; }

    bool IsEqual(Base const& rhs) const noexcept
    {
        if (this == &rhs) {
            return true;
        }

        if (typeid(*this) != typeid(rhs)) {
            return false;
        }

        return is_equal(rhs);
    }

protected:
    virtual bool is_equal(Base const& rhs) const noexcept { return name1_ == rhs.name1_; }

private:
    std::string name1_;
};

class Derived : public Base {
public:
    explicit Derived(std::string name1 = "", std::string name2 = "") : Base{name1}, name2_{name2} {}

    virtual ~Derived() override = default;
    std::string const& GetName2() const noexcept { return name2_; }

protected:
    virtual bool is_equal(Base const& rhs) const noexcept override
    {
        if (!Base::is_equal(rhs)) {
            return false;
        }

        auto rhs_d = dynamic_cast<Derived const*>(&rhs);

        return (rhs_d != nullptr) && (name2_ == rhs_d->name2_);
    }

private:
    std::string name2_;
};

class DerivedDerived : public Derived {
public:
    explicit DerivedDerived(std::string name1 = "", std::string name2 = "", std::string name3 = "")
        : Derived{name1, name2}, name3_{name3}
}

```

```

{
}

virtual ~DerivedDerived() override = default;
std::string const& GetName3() const noexcept { return name3_; }

protected:
    virtual bool is_equal(Base const& rhs) const noexcept override
    {
        if (!Derived::is_equal(rhs))
            return false;
    }

    auto rhs_d = dynamic_cast<DerivedDerived const*>(&rhs);

    return (rhs_d != nullptr) && (name3_ == rhs_d->name3_);
}

private:
    std::string name3_;
};

TEST(DesignPatternA, NVI)
{
    auto b1 = Base{"b1"};

    ASSERT_TRUE(b1.IsEqual(Base{b1}));
    ASSERT_TRUE(b1.IsEqual(Base{"b1"}));
    ASSERT_FALSE(b1.IsEqual(Base{"b2"}));
    ASSERT_FALSE(b1.IsEqual(Derived{"b1", "d1"}));

    auto d1 = Derived{"b1", "d1"};

    ASSERT_FALSE(d1.IsEqual(Base{"b1"}));
    ASSERT_TRUE(d1.IsEqual(d1));
    ASSERT_TRUE(d1.IsEqual(Derived{"b1", "d1"}));
    ASSERT_FALSE(d1.IsEqual(Derived{"b1", "d2"}));
    ASSERT_FALSE(d1.IsEqual(DerivedDerived{"b1", "d1", "dd2"}));

    auto dd1 = DerivedDerived{"b1", "d1", "dd1"};

    ASSERT_FALSE(dd1.IsEqual(Base{"b1"}));
    ASSERT_FALSE(dd1.IsEqual(Derived{"b1", "d1"}));
    ASSERT_TRUE(dd1.IsEqual(dd1));
    ASSERT_TRUE(dd1.IsEqual(DerivedDerived{"b1", "d1", "dd1"}));
    ASSERT_FALSE(dd1.IsEqual(DerivedDerived{"b1", "d1", "dd2"}));
}

```

- 演習-NVIへ戻る。

解答-Raiiの効果

- 選択肢1
- 参照 [RAII\(scoped_guard\)](#)
- 演習-RAIIの効果へ戻る。

解答例-RAII

```

// @@@ exercise/design_pattern_a/raii.cpp 5
// [A]
// 下記クラスBase、Derivedはクローンパターンをしているが、Clone関数はnewしたオブジェクトであるため、
// メモリーリークを起こしやすい。std::unique_ptrを使用してこの問題に対処せよ。

class Base {
public:
    explicit Base(std::string name) : name1_{std::move(name)} {}
    virtual ~Base() = default;
    virtual std::string const& GetName() const noexcept { return name1_; }

    virtual std::unique_ptr<Base> Clone() const { return std::make_unique<Base>(name1_); }

    Base(Base const&) = delete;
    Base& operator=(Base const&) = delete;

private:
    std::string name1_;
};

```

```

class Derived final : public Base {
public:
    explicit Derived(std::string name1 = "", std::string name2 = "")
        : Base{std::move(name1)}, name2_{std::move(name2)}
    {
    }
    virtual ~Derived() override = default;
    virtual std::string const& GetName() const noexcept override { return name2_; }

    virtual std::unique_ptr<Base> Clone() const override { return CloneOwn(); }

    std::unique_ptr<Derived> CloneOwn() const
    {
        return std::make_unique<Derived>(Base::GetName(), name2_);
    }

private:
    std::string name2_;
};

TEST(DesignPatternA, RAI)
{
    Derived d1{"name1", "name2"};
    std::unique_ptr<Derived> d2{d1.CloneOwn()};

    ASSERT_EQ("name1", d2->Base::GetName());
    ASSERT_EQ("name2", d2->GetName());

    std::unique_ptr<Base> b3 = d1.Clone();

    ASSERT_EQ("name1", b3->Base::GetName());
    ASSERT_EQ("name2", b3->GetName());
}

```

- 演習-RAIIへ戻る。

解答例-Future

```

// @@@ exercise/design_pattern_a/future.cpp 25
// [A]
// 下記のfind_files_concurrentlyはスレッドの出力の結果をキャプチャリファレンスで受け取るため、
// 入出力の関係が明確でない。Futureパターンを使用しそれを明確にするリファクタリングを行え。

std::vector<std::string> find_files_concurrently()
{
    std::future<std::vector<std::string>> result0
        = std::async(std::launch::async, [] { return find_files("../programming_convention_a/"); });

    std::future<std::vector<std::string>> result1
        = std::async(std::launch::async, [] { return find_files("../programming_convention_q/"); });

    auto pca = result0.get();
    auto pcq = result1.get();

    pca.insert(pca.end(), pcq.begin(), pcq.end());

    return pca;
}

TEST(DesignPatternA, Future)
{
    auto files = find_files_concurrently();

    ASSERT_GT(files.size(), 10);
}

```

- 演習-Futureへ戻る。

解答例-DI

```

// @@@ exercise/design_pattern_a/di.cpp 10
// [A]
// CppFilesはLsCppを直に生成するため、LsCpp::FileList()がエラーした場合の単体テスト実施が
// 困難である。CppFiles[DI]パターンを適用するとともに、LsCppを適切に変更することによって、
// LsCpp::FileList()がエラーした場合のCppFilesの単体テストを行え。

class LsCpp {
public:

```

```

virtual ~LsCpp() {}

std::string const& FileList() { return file_list(); }

private:
    std::string files_{};

virtual std::string const& file_list() // 単体テストのためにvirtual
{
    if (files_.size() != 0) { // キャッシュを使う
        return files_;
    }

    auto stream
        = std::unique_ptr<FILE, decltype(&fclose)>{popen("ls ..//ut_data/*.cpp", "r"), fclose};

    if (stream.get() == NULL) {
        throw std::exception{};
    }

    char buff[256];
    while (fgets(buff, sizeof(buff) - 1, stream.get()) != NULL) {
        files_ += buff;
    }
}

return files_;
};

class CppFiles {
public:
    explicit CppFiles(std::unique_ptr<LsCpp>&& ls_cpp = std::make_unique<LsCpp>())
        : ls_cpp_{std::move(ls_cpp)}
    {
    }

    std::vector<std::string> FileList() const
    {
        auto files = std::string{};

        try {
            files = ls_cpp_->fileList();
        }
        catch (...) {
            ; // 例外発生時には空のベクタを返すので何もしない。
        }

        return split_cr(files);
    }
}

private:
    std::unique_ptr<LsCpp> ls_cpp_{};

static std::vector<std::string> split_cr(std::string const& str)
{
    auto ss = std::stringstream{str};
    auto ret = std::vector<std::string>{};

    for (std::string line; std::getline(ss, line);) {
        ret.emplace_back(line);
    }

    return ret;
};
};

class LsCppError : public LsCpp {
public:
    LsCppError() noexcept {}
    virtual ~LsCppError() override {}

private:
    [[noreturn]] virtual std::string const& file_list() override { throw std::exception{}; }
};

TEST(DesignPatternA, DI)
{
    auto      files = CppFiles{};
    auto const& act   = files.FileList();
    auto      exp   = std::vector<std::string>{"..//ut_data/a.cpp", "..//ut_data/abc.cpp",

```

```

        ".../ut_data/efghij.cpp");

ASSERT_EQ(exp, act);

// エラー系のテスト
auto files2 = CppFiles{std::make_unique<LsCppError>()};

ASSERT_EQ(0, files2.FileList().size());
}

```

- 演習-DIへ戻る。

解答例-Singleton

```

// @@@ exercise/design_pattern_a/singleton.cpp 5
// [A]
// 下記 AppConfig はアプリケーション全体の設定を管理するためのクラスである。
// 目的に、そのインスタンス AppConfig は広域のアクセスが必要であり、
// グローバルインスタンスとして実装している。
// グローバルインスタンスは、初期化の順番が標準化されておらず、
// 多くの処理系ではリンクの順番に依存しているため、
// アプリケーション立ち上げ時に様々な問題を起こすことがある。
// こういった問題を回避するため、 AppConfig を Singleton 化せよ。
// また他の問題があれば合わせて修正せよ。

// [解説]
// * AppConfig を Singleton にした。
//   * インスタンスを返す Inst() と同じインスタンスを const 修飾したものを返す InstConst() を追加。
//   * コンストラクタを private にした。
//   * copy コンストラクタを = delete した（こうすれば move コンストラクタも = delete される）。
// * リファクタリング
//   * BaseColor をスコープ enum にした。
//   * GetXXX を const 関数にした。
//   * GetUserNames の戻りを const リファレンスにした。
//   * コピー演算子を使用し SetDefault をシンプルにした。

class AppConfig {
public:
    static AppConfig& Inst()
    {
        static auto inst = AppConfig{};
        return inst;
    }

    static AppConfig const& InstConst() { return Inst(); }

    enum class BaseColor { Red, Green, Black };

    void SetBaseColor(BaseColor color) noexcept { color_ = color; }
    BaseColor GetBaseColor() const noexcept { return color_; }

    void SetUserName(std::string_view username) { username_ = username; }
    std::string const& GetUserName() const noexcept { return username_; }

    void Logging(bool is_logging) noexcept { is_logging_ = is_logging; }
    bool IsLogging() const noexcept { return is_logging_; }

    // 他の設定値は省略

    void SetDefault() { *this = AppConfig{}; }

    // これがないと copy コンストラクタや move コンストラクタで別のインスタンスが作れる。
    // AppConfig app{AppConfig::Inst()};
    // AppConfig app{std::move(AppConfig::Inst())};
    AppConfig(AppConfig const&) = delete;

private:
    BaseColor color_{BaseColor::Red};
    std::string username_{ "No Name" };
    bool is_logging_{ false };

    AppConfig() = default;
    AppConfig& operator=(AppConfig const&) = default;
};

class DesignPatternA_F : public ::testing::Test {
protected:
    virtual void SetUp() override { AppConfig::Inst().SetDefault(); }
}

```

```

    virtual void TearDown() override { AppConfig::Inst().SetDefault(); }
};

TEST_F(DesignPatternA_F, Singleton)
{
    ASSERT_EQ(AppConfig::BaseColor::Red, AppConfig::InstConst().GetBaseColor());
    ASSERT_EQ("No Name", AppConfig::InstConst().GetUserName());
    ASSERT_FALSE(AppConfig::InstConst().IsLoggin());

    AppConfig::Inst().SetBaseColor(AppConfig::BaseColor::Green);
    ASSERT_EQ(AppConfig::BaseColor::Green, AppConfig::InstConst().GetBaseColor());

    AppConfig::Inst().SetUserName("Stroustrup");
    ASSERT_EQ("Stroustrup", AppConfig::InstConst().GetUserName());

    AppConfig::Inst().Logging(true);
    ASSERT_TRUE(AppConfig::InstConst().IsLoggin());

    AppConfig::Inst().SetDefault();
    ASSERT_EQ(AppConfig::BaseColor::Red, AppConfig::InstConst().GetBaseColor());
    ASSERT_EQ("No Name", AppConfig::InstConst().GetUserName());
    ASSERT_FALSE(AppConfig::InstConst().IsLoggin());
}

```

- 演習-Singletonへ戻る。

解答例-State

```

// @@@ exercise/design_pattern_a/state.cpp 5
// [A]
// 下記クラスGreetingにはlang_に対する同型のswitch文が3個ある。
// これは機能追加時にバグが混入しやすいアンチパターンであるため、
// Stateパターンを用いリファクタリングせよ。
// また、他の問題があれば合わせて修正せよ。

enum class Language { English, Japanese, French };

class GreetingState {
public:
    virtual ~GreetingState() = default;
    std::string GoodMorning() { return good_morning(); }
    std::string Hello() { return hello(); }
    std::string GoodEvening() { return good_evening(); }

private:
    virtual std::string good_morning() const = 0;
    virtual std::string hello() const = 0;
    virtual std::string good_evening() const = 0;
};

class GreetingState_English : public GreetingState {
    virtual std::string good_morning() const override { return u8"good morning"; }
    virtual std::string hello() const override { return u8"hello"; }
    virtual std::string good_evening() const override { return u8"good evening"; }
};

class GreetingState_Japanese : public GreetingState {
    virtual std::string good_morning() const override { return u8"おはよう"; }
    virtual std::string hello() const override { return u8"こんにちは"; }
    virtual std::string good_evening() const override { return u8"こんばんは"; }
};

class GreetingState_French : public GreetingState {
    virtual std::string good_morning() const override { return u8"Bonjour"; }
    virtual std::string hello() const override { return u8"Bonjour"; }
    virtual std::string good_evening() const override { return u8"bonne soirée"; }
};

class Greeting {
public:
    explicit Greeting(Language lang = Language::English) : state_{new_state(lang)} {}
    void SetLanguage(Language lang) { state_ = new_state(lang); }

    std::string GoodMorning() const { return state_->GoodMorning(); }
    std::string Hello() const { return state_->Hello(); }
    std::string GoodEvening() const { return state_->GoodEvening(); }

private:

```

```

    std::unique_ptr<GreetingState> state_;
```

```

    static std::unique_ptr<GreetingState> new_state(Language lang)
    {
        switch (lang) {
        case Language::Japanese:
            return std::make_unique<GreetingState_Japanese>();
        case Language::French:
            return std::make_unique<GreetingState_French>();
        case Language::English:
        default:
            return std::make_unique<GreetingState_English>();
        }
    }
};

TEST(DesignPatternA, State)
{
    auto greeting = Greeting{};

    ASSERT_EQ(u8"good morning", greeting.GoodMorning());
    ASSERT_EQ(u8"hello", greeting.Hello());
    ASSERT_EQ(u8"good evening", greeting.GoodEvening());

    greeting.SetLanguage(Language::Japanese);
    ASSERT_EQ(u8"おはよう", greeting.GoodMorning());
    ASSERT_EQ(u8"こんにちは", greeting.Hello());
    ASSERT_EQ(u8"こんばんは", greeting.GoodEvening());

    greeting.SetLanguage(Language::French);
    ASSERT_EQ(u8"Bonjour", greeting.GoodMorning());
    ASSERT_EQ(u8"Bonjour", greeting.Hello());
    ASSERT_EQ(u8"bonne soirée", greeting.GoodEvening());

    greeting.SetLanguage(Language::English);
    ASSERT_EQ(u8"good morning", greeting.GoodMorning());
    ASSERT_EQ(u8"hello", greeting.Hello());
    ASSERT_EQ(u8"good evening", greeting.GoodEvening());
}

```

- 演習-Stateへ戻る。

解答例-Null Object

```

// @@@ exercise/design_pattern_a/null_object.cpp 38
// [A]
// 下記クラスPersonにはgreeting_のヌルチェックを行う三項演算子が3つある。
// これはヌルポインタアクセスを起こしやすいアンチパターンであるため、
// Null Objectパターンを用いリファクタリングせよ。
// また、他の問題があれば合わせて修正せよ。

// [解説]
// * 通常Null Objectパターンは
//     if(object_ptr != nullptr) { ... }
//   のようなコードが頻繁に存在する場合にそのコードの繰り返しを無くすためのものであるが、
//   本例では対象が生のポインタでなくスマートポインタに適用した。
// * 本例では、GoodMorning等が単純であるためGreetingにNVIを適用していないが、NVIを適用しても良い。
// * 本例では、ヌルかどうかの同型条件分岐が3個しかないコードにNull Objectパターンを適用した。
//   例題のためそうしたが、この程度の単純なコードにこのパターンを適用するのはやりすぎである。
//   この程度のコードクローンであれば一つのヘルパー関数にまとめた方が実践的である。

class Greeting {
public:
    explicit Greeting(Language lang = Language::English) : state_{new_state(lang)} {}
    virtual ~Greeting() = default;
    void SetLanguage(Language lang) { state_ = new_state(lang); }

    virtual std::string GoodMorning() const { return state_->GoodMorning(); }
    virtual std::string Hello() const { return state_->Hello(); }
    virtual std::string GoodEvening() const { return state_->GoodEvening(); }

private:
    std::unique_ptr<GreetingState> state_;
```

```

    static std::unique_ptr<GreetingState> new_state(Language lang)
    {
        switch (lang) {
        case Language::Japanese:
            return std::make_unique<GreetingState_Japanese>();
        
```

```

        case Language::French:
            return std::make_unique<GreetingState_French>();
        case Language::English:
        default:
            return std::make_unique<GreetingState_English>();
    }
}

class GreetingSilent : public Greeting {
public:
    explicit GreetingSilent() = default;
    virtual ~GreetingSilent() override = default;

    virtual std::string GoodMorning() const override { return ""; }
    virtual std::string Hello() const override { return ""; }
    virtual std::string GoodEvening() const override { return ""; }
};

class Person {
public:
    explicit Person(Language lang, bool silent = false)
        : greeting_{silent ? std::make_unique<GreetingSilent>() : std::make_unique<Greeting>(lang)}
    {}

    std::string GoodMorning() const { return greeting_->GoodMorning(); }
    std::string Hello() const { return greeting_->Hello(); }
    std::string GoodEvening() const { return greeting_->GoodEvening(); }

private:
    std::unique_ptr<Greeting> greeting_;
};

TEST(DesignPatternA, NullObject)
{
    auto e = Person{Language::English};

    ASSERT_EQ(u8"good morning", e.GoodMorning());
    ASSERT_EQ(u8"hello", e.Hello());
    ASSERT_EQ(u8"good evening", e.GoodEvening());

    auto j = Person{Language::Japanese};
    ASSERT_EQ(u8"おはよう", j.GoodMorning());
    ASSERT_EQ(u8"こんにちは", j.Hello());
    ASSERT_EQ(u8"こんばんは", j.GoodEvening());

    auto f = Person{Language::French};
    ASSERT_EQ(u8"Bonjour", f.GoodMorning());
    ASSERT_EQ(u8"Bonjour", f.Hello());
    ASSERT_EQ(u8"bonne soirée", f.GoodEvening());

    auto e_s = Person{Language::English, true};

    ASSERT_EQ(u8"", e_s.GoodMorning());
    ASSERT_EQ(u8"", e_s.Hello());
    ASSERT_EQ(u8"", e_s.GoodEvening());
}

```

- 演習-Null Objectへ戻る。

解答例-Templated Methodメソッド

```

// @@@ exercise/design_pattern_a/template_method.cpp 5
// [A]
// 下記クラスXxxDataFormatterXml、XxxDataFormatterCsvは同様の処理を行い、
// それぞれのフォーマットで文字列を出力する。このような処理のクローンはTemplate Method
// パターンにより排除できる。
// このパターンを用い、下記2クラスをリファクタリングせよ。
// また、他の問題があれば合わせて修正せよ。

// [解説]
// * Template Methodのインターフェースクラスとして、XxxDataFormatterIFを定義した。
//     * header()、footer()はstd::stringのリファレンスを返すが、
//     body()は返すstd::stringが引数に依存して変わるために、実態を返す。
// * その他の修正
//     * header_やfooter_はそれぞれのクラスで同じオブジェクトであるため、
//     static constインスタンスとして、それぞれheader()、footer()の内部で定義した。
//     * XxxDataFormatterXml、XxxDataFormatterCsvはそれ以上派生する必要がないためfinalとした。

```

```

struct XxxData {
    int a;
    int b;
    int c;
};

class XxxDataFormatterIF {
public:
    XxxDataFormatterIF() noexcept = default;
    virtual ~XxxDataFormatterIF() = default;
    XxxDataFormatterIF(XxxDataFormatterIF const&) = delete;
    XxxDataFormatterIF& operator=(XxxDataFormatterIF const&) = delete;

    std::string ToString(XxxData const& xxx_data) const
    {
        return header() + body(xxx_data) + footer();
    }

    std::string ToString(std::vector<XxxData> const& xxx_datas) const
    {
        auto ret = std::string{header()};

        for (auto const& xxx_data : xxx_datas) {
            ret += body(xxx_data);
        }

        return ret + footer();
    }

private:
    virtual std::string const& header() const = 0;
    virtual std::string const& footer() const = 0;
    virtual std::string body(XxxData const& xxx_data) const = 0;
};

class XxxDataFormatterXml final : public XxxDataFormatterIF {
public:
    XxxDataFormatterXml() = default;
    virtual ~XxxDataFormatterXml() override = default;

private:
    virtual std::string const& header() const override
    {
        static auto const header
            = std::string("xml version=\"1.0\" encoding=\"UTF-8\" ?\n<XxxDataFormatterXml>\n");

        return header;
    }

    virtual std::string const& footer() const override
    {
        static auto const footer = std::string("</XxxDataFormatterXml>\n");

        return footer;
    }

    virtual std::string body(XxxData const& xxx_data) const override
    {
        auto content = std::string("<Item>\n");

        content += "    <XxxData a=\"" + std::to_string(xxx_data.a) + "\">\n";
        content += "    <XxxData b=\"" + std::to_string(xxx_data.b) + "\">\n";
        content += "    <XxxData c=\"" + std::to_string(xxx_data.c) + "\">\n";

        return content + "</Item>\n";
    }
};

class XxxDataFormatterCsv final : public XxxDataFormatterIF {
public:
    XxxDataFormatterCsv() = default;
    virtual ~XxxDataFormatterCsv() override = default;

private:
    virtual std::string const& header() const override
    {
        static auto const header = std::string("a, b, c\n");

        return header;
    }
};

```

```

}

virtual std::string const& footer() const override
{
    static auto const footer = std::string{};

    return footer;
}

virtual std::string body(XxxData const& xxx_data) const override
{
    return std::string{std::to_string(xxx_data.a) + ", " + std::to_string(xxx_data.b) + ", "
                      + std::to_string(xxx_data.b) + "\n"};
}
};

TEST(DesignPatternA, TemplateMethod)
{
    auto xml = XxxDataFormatterXml{};
    {
        auto const expect_scalar = std::string{
            "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n"
            "<XxxDataFormatterXml>\n"
            "<Item>\n"
            "    <XxxData a=\"1\"\n"
            "    <XxxData b=\"100\"\n"
            "    <XxxData c=\"10\"\n"
            "</Item>\n"
            "</XxxDataFormatterXml>\n"};
        auto const actual_scalar = xml.ToString({1, 100, 10});
        ASSERT_EQ(expect_scalar, actual_scalar);

        auto const expect_array = std::string{
            "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n"
            "<XxxDataFormatterXml>\n"
            "<Item>\n"
            "    <XxxData a=\"1\"\n"
            "    <XxxData b=\"100\"\n"
            "    <XxxData c=\"10\"\n"
            "</Item>\n"
            "<Item>\n"
            "    <XxxData a=\"2\"\n"
            "    <XxxData b=\"200\"\n"
            "    <XxxData c=\"20\"\n"
            "</Item>\n"
            "</XxxDataFormatterXml>\n"};
        auto const actual_array = xml.ToString({{1, 100, 10}, {2, 200, 20}});
        ASSERT_EQ(expect_array, actual_array);
    }

    auto csv = XxxDataFormatterCsv{};
    {
        auto expect_scalar = std::string{
            "a, b, c\n"
            "1, 100, 100\n"};
        auto const actual_scalar = csv.ToString({1, 100, 10});
        ASSERT_EQ(expect_scalar, actual_scalar);

        auto const expect_array = std::string{
            "a, b, c\n"
            "1, 100, 100\n"
            "2, 200, 200\n"};
        auto const actual_array = csv.ToString({{1, 100, 10}, {2, 200, 20}});
        ASSERT_EQ(expect_array, actual_array);
    }
}

```

- 演習-Templateメソッドへ戻る。

解答例-Factory

```

// @@@ exercise/design_pattern_a/factory_lib.h 43
// [A]
// 下記クラスXxxDataFormatterXml、XxxDataFormatterCsvはヘッダファイルで宣言・定義を行ったために
// 他の.cppファイルから直接アクセスできてしまう。
// Factoryパターンを用いて、XxxDataFormatterXml、XxxDataFormatterCsvを他の.cppファイルから
// 直接アクセスできないようにせよ。
// [解説]

```

```

// 一般的には、Factory関数は
//     std::unique_ptr<XxxDataFormatterIF> XxxDataFormatterFactory(XxxDataFormatterType type);
// のような形状になるが、今回の例では生成オブジェクトの提供するサービスがconst関数のみであるため、
// constなunique_ptrを返している。
// また、さらにこの考え方を進め、newしてオブジェクトの生成をする必要はないことに気づけば、
// XxxDataFormatterFactory2のようにconstリファレンスを返すこともできる。

enum class XxxDataFormatterType { Xml, Csv };

std::unique_ptr<XxxDataFormatterIF const> XxxDataFormatterFactory(XxxDataFormatterType type);
XxxDataFormatterIF const& XxxDataFormatterFactory2(XxxDataFormatterType type) noexcept;

```

```

// @@@ exercise/design_pattern_a/factory_lib.cpp 77

std::unique_ptr<XxxDataFormatterIF const> XxxDataFormatterFactory(XxxDataFormatterType type)
{
    switch (type) {
        case XxxDataFormatterType::Xml:
            return std::make_unique<XxxDataFormatterXml>();
        case XxxDataFormatterType::Csv:
            return std::make_unique<XxxDataFormatterCsv>();
        default:
            assert("unknown type");
            return std::unique_ptr<XxxDataFormatterIF const>{};
    }
}

XxxDataFormatterIF const& XxxDataFormatterFactory2(XxxDataFormatterType type) noexcept
{
    static auto const xml = XxxDataFormatterXml{};
    static auto const csv = XxxDataFormatterCsv{};

    switch (type) {
        case XxxDataFormatterType::Xml:
            return xml;
        case XxxDataFormatterType::Csv:
            return csv;
        default:
            assert("unknown type");
            return csv;
    }
}

```

```

// @@@ exercise/design_pattern_a/factory.cpp 9

TEST(DesignPatternA, Factory)
{
    auto xml = XxxDataFormatterFactory(XxxDataFormatterType::Xml);
    {
        auto const expect_scalar = std::string{
            "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n"
            "<XxxDataFormatterXml>\n"
            "<Item>\n"
            "  <XxxData a=\"1\">\n"
            "  <XxxData b=\"100\">\n"
            "  <XxxData c=\"10\">\n"
            "</Item>\n"
            "</XxxDataFormatterXml>\n"};
        auto const actual_scalar = xml->ToString({1, 100, 10});

        ASSERT_EQ(expect_scalar, actual_scalar);
    }

    auto csv = XxxDataFormatterFactory(XxxDataFormatterType::Csv);
    {
        auto const expect_array = std::string{
            "a, b, c\n"
            "1, 100, 100\n"
            "2, 200, 200\n"};
        auto const actual_array = csv->ToString({{1, 100, 10}, {2, 200, 20}});

        ASSERT_EQ(expect_array, actual_array);
    }
}

TEST(DesignPatternA, Factory2)
{
    auto const& xml = XxxDataFormatterFactory2(XxxDataFormatterType::Xml);
    {
        auto const expect_scalar = std::string{
            "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n"

```

```

    "<XxxDataFormatterXml>\n"
    "<Item>\n"
    "  <XxxData a=\"1\"\n"
    "  <XxxData b=\"100\"\n"
    "  <XxxData c=\"10\"\n"
    "</Item>\n"
    "</XxxDataFormatterXml>\n";
auto const actual_scalar = xml.ToString({1, 100, 10});

ASSERT_EQ(expect_scalar, actual_scalar);
}

auto const& csv = XxxDataFormatterFactory2(XxxDataFormatterType::Csv);
{
    auto const expect_array = std::string{
        "a, b, c\n"
        "1, 100, 100\n"
        "2, 200, 200\n"};
    auto const actual_array = csv.ToString({{1, 100, 10}, {2, 200, 20}});

    ASSERT_EQ(expect_array, actual_array);
}
}

```

- 演習-Factoryへ戻る。

解答例-Named Constructor

```

// @@@ exercise/design_pattern_a/named_constructor_lib.h 14
// [A]
// 下記関数XxxDataFormatterFactoryはインターフェースクラスXxxDataFormatterIFのファクトリ関数
// である。これをnamed constructorパターンで実装しなおせ。

// [解説]
// * XxxDataFormatterIFの特性から、Named ConstructorはXxxDataFormatterIFのconstリファレンスを返す
//   仕様としたが、戻すオブジェクトの特性より戻り値型は以下のようにすべき。
//
//   const/非const | 静的/動的 | 戻す型
//   -----+-----+-----
//   const       | 静的      | XxxDataFormatterIFのconstリファレンス
//   | 動的      | std::unique_ptr<const XxxDataFormatterIF>
//   非const     | 静的      | XxxDataFormatterIFのリファレンス
//   | 動的      | std::unique_ptr<XxxDataFormatterIF>
//
// * FactoryとNamed Constructorはほぼ等価であり、どちらを使っても派生型の隠蔽という効果は等しい。
// * 筆者は、今回の例のように静的オブジェクトを返す場合、Named Constructor、
//   動的オブジェクトを返す場合、Factoryを使用している。

class XxxDataFormatterIF {
public:
    XxxDataFormatterIF() = default;
    static XxxDataFormatterIF const& Xml() noexcept;
    static XxxDataFormatterIF const& Csv() noexcept;
    static XxxDataFormatterIF const& Table() noexcept;

    virtual ~XxxDataFormatterIF() = default;
    XxxDataFormatterIF(XxxDataFormatterIF const&) = delete;
    XxxDataFormatterIF& operator=(XxxDataFormatterIF const&) = delete;

    std::string ToString(XxxData const& xxx_data) const
    {
        return header() + body(xxx_data) + footer();
    }

    std::string ToString(std::vector<XxxData> const& xxx_datas) const
    {
        auto ret = header();

        for (auto const& xxx_data : xxx_datas) {
            ret += body(xxx_data);
        }

        return ret + footer();
    }

private:
    virtual std::string const& header() const = 0;
    virtual std::string const& footer() const = 0;
}

```

```

        virtual std::string      body(XxxData const& xxx_data) const = 0;
    };

// @@@ exercise/design_pattern_a/named_constructor_lib.cpp 114

XxxDataFormatterIF const& XxxDataFormatterIF::Xml() noexcept
{
    static auto const inst = XxxDataFormatterXml{};
    return inst;
}

XxxDataFormatterIF const& XxxDataFormatterIF::Csv() noexcept
{
    static auto const inst = XxxDataFormatterCsv{};
    return inst;
}

XxxDataFormatterIF const& XxxDataFormatterIF::Table() noexcept
{
    static auto const inst = XxxDataFormatterTable{};
    return inst;
}

```

```

// @@@ exercise/design_pattern_a/named_constructor.cpp 9

TEST(DesignPatternA, NamedConstructor)
{
    auto const& xml = XxxDataFormatterIF::Xml();
    {
        auto const expect_scalar = std::string{
            "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n"
            "<XxxDataFormatterXml>\n"
            "<Item>\n"
            "  <XxxData a=\"1\">\n"
            "  <XxxData b=\"100\">\n"
            "  <XxxData c=\"10\">\n"
            "</Item>\n"
            "</XxxDataFormatterXml>\n"};
        auto const actual_scalar = xml.ToString({1, 100, 10});

        ASSERT_EQ(expect_scalar, actual_scalar);
    }

    auto const& csv = XxxDataFormatterIF::Csv();
    {
        auto const expect_array = std::string{
            "a, b, c\n"
            "1, 100, 100\n"
            "2, 200, 200\n"};
        auto const actual_array = csv.ToString({{1, 100, 10}, {2, 200, 20}});

        ASSERT_EQ(expect_array, actual_array);
    }

    auto const& table = XxxDataFormatterIF::Table();
    {
        auto const expect_array = std::string{
            "+---+---+---+\n"
            "| a | b | c |\n"
            "+---+---+---+\n"
            "| 3 | 300 | 30 |\n"
            "+---+---+---+\n"
            "| 4 | 400 | 40 |\n"
            "+---+---+---+"};
        auto const actual_array = table.ToString({{3, 300, 30}, {4, 400, 40}});

        ASSERT_EQ(expect_array, actual_array);
    }
}

```

- 演習-Named Constructorへ戻る。

解答例-Proxy

```

// @@@ exercise/design_pattern_a/proxy.cpp 8
// [A]
// 下記クラスLsDirのFileListはlsコマンドをpopenにより実行し、その戻り値をstd::stringで返す。
// popenはコストの高いコードなので、パフォーマンスを上げるためにlsの戻り値をキャッシュしたいが、
// 現行のLsDirも必要である。

```

```

// Proxyパターンを使い、この問題に対処するためのLsDirCachedを作れ。
#define USE_ACCURATE_PROXY
#ifndef USE_ACCURATE_PROXY // 本来#endifは問題を発生させるため使うべきではないが、例なので。
class LsDirIF {
public:
    LsDirIF() = default;
    virtual ~LsDirIF() = default;

    void SetArgs(std::string_view args) { args_ = args; }
    std::string const& GetArgs() const noexcept { return args_; }
    std::string const& FileList() const { return file_list(); }

private:
    std::string args_{};
    virtual std::string file_list() const = 0;
};

class LsDir : public LsDirIF {
public:
    LsDir() = default;
    virtual ~LsDir() = default;

private:
    virtual std::string file_list() const override
    {
        auto cmd = std::string("ls ") + GetArgs();
        auto to_close = [](FILE* f) { fclose(f); };
        auto stream = std::unique_ptr<FILE, decltype(to_close)>{popen(cmd.c_str(), "r"), to_close};

        auto files = std::string{};
        char buff[256];

        while (fgets(buff, sizeof(buff) - 1, stream.get()) != NULL) {
            files += buff;
        }

        return files;
    }
};

class LsDirCached : public LsDirIF {
public:
    LsDirCached() = default;
    virtual ~LsDirCached() override = default;

private:
    mutable std::string latest_ls_{};
    mutable LsDir ld_no_cache_{};

    virtual std::string file_list() const override
    {
        if (GetArgs() == ld_no_cache_.GetArgs()) {
            return latest_ls_;
        }

        ld_no_cache_.SetArgs(GetArgs());
        latest_ls_ = ld_no_cache_.FileList();

        return latest_ls_;
    }
};

#else // not USE_ACCURATE_PROXY
class LsDir {
public:
    LsDir() = default;
    virtual ~LsDir() = default;

    void SetArgs(std::string_view args) { args_ = args; }
    std::string const& GetArgs() const { return args_; }
    std::string const& FileList() const { return file_list(); }

protected:
    virtual std::string file_list() const
    {
        auto cmd = std::string("ls ") + GetArgs();
        auto to_close = [](FILE* f) { fclose(f); };
        auto stream = std::unique_ptr<FILE, decltype(to_close)>{popen(cmd.c_str(), "r"), to_close};

        auto files = std::string{};

```

```

char buff[256];

    while (fgets(buff, sizeof(buff) - 1, stream.get()) != NULL) {
        files += buff;
    }

    return files;
}

private:
    std::string args_{};

class LsDirCached : public LsDir {
public:
    LsDirCached() = default;
    virtual ~LsDirCached() override = default;

protected:
    virtual std::string file_list() const override
    {
        if (GetArgs() == latest_args_) {
            return latest_ls_;
        }

        latest_args_ = GetArgs();
        latest_ls_ = LsDir::file_list();

        return latest_ls_;
    }

private:
    mutable std::string latest_ls_{};
    mutable std::string latest_args_{};
};

#endif

TEST(DesignPatternA, Proxy)
{
    auto ld = LsDir{};

    {
        ld.SetArgs("../ut_data/");

        auto exp = std::string{"a.cpp\na.h\nabc.cpp\nabc.h\n\nd\nefghij.cpp\nefghij.h\nlib\no\n"};
        auto act = ld.FileList();

        ASSERT_EQ(exp, act);
        ASSERT_EQ(act, ld.FileList());
    }

    {
        ld.SetArgs("../ut_data/lib/");

        auto exp = std::string{"lib.cpp\nlib.h\n"};
        auto act = ld.FileList();

        ASSERT_EQ(exp, act);
        ASSERT_EQ(act, ld.FileList());
    }
}

TEST(DesignPatternA, Proxy2)
{
    auto ld = LsDirCached{};

    {
        ld.SetArgs("../ut_data/");

        auto exp = std::string{"a.cpp\na.h\nabc.cpp\nabc.h\n\nd\nefghij.cpp\nefghij.h\nlib\no\n"};
        auto act = ld.FileList();

        ASSERT_EQ(exp, act);
        ASSERT_EQ(act, ld.FileList());
    }

    {
        ld.SetArgs("../ut_data/lib/");

        auto exp = std::string{"lib.cpp\nlib.h\n"};
        auto act = ld.FileList();
    }
}

```

```

        ASSERT_EQ(exp, act);
        ASSERT_EQ(act, ld.FileList());
    }

template <typename LSDIR>
uint32_t measure_performance(LSDIR const& ls_dir, uint32_t count) noexcept
{
    auto const start = std::chrono::system_clock::now();
    {
        for (decltype(count) i = 0; i < count; ++i) {
            volatile auto const list = ls_dir.FileList();
        }
    }

    auto const stop = std::chrono::system_clock::now();

    return std::chrono::duration_cast<std::chrono::microseconds>(stop - start).count();
}

TEST(DesignPatternA, ProxyPerformance)
{
    auto ld          = LsDir{};
    auto elapsed_no_cache = uint32_t{measure_performance(ld, 10)};
    std::cout << "No Cache Elapse:" << elapsed_no_cache << " usec" << std::endl;

    auto ldc          = LsDirCached{};
    auto elapsed_cache = uint32_t{measure_performance(ldc, 10)};
    std::cout << "Cached Elapse:" << elapsed_cache << " usec" << std::endl;

    ASSERT_LT(30 * elapsed_cache, elapsed_no_cache); // 30倍に理由はない。
}

```

- 演習_Proxyへ戻る。

解答例-Strategy

```

// @@@ exercise/design_pattern_a/strategy.cpp 12
// [A]
// 下記find_filesは醜悪であるだけでなく、拡張性もない。
// Strategyパターンを用い、この問題に対処せよ。

using FindCondition = std::function<bool(std::filesystem::path const&)>;

std::vector<std::string> find_files(std::string const& path, FindCondition condition)
{
    auto files = std::vector<fs::path>{};

    // recursive_directory_iteratorはファイルシステム依存するため、その依存を排除する他の処理
    std::copy(fs::recursive_directory_iterator{path}, fs::recursive_directory_iterator{},
              std::back_inserter(files));

    std::sort(files.begin(), files.end());

    auto ret = std::vector<std::string>{};

    std::for_each(files.cbegin(), files.cend(), [&](fs::path const& p) {
        if (condition(p)) {
            ret.emplace_back(p.generic_string());
        }
    });

    return ret;
}

bool is_cpp_file(std::filesystem::path const& path)
{
    auto const filename = path.filename().generic_string();
    static auto const cpp_file = std::string{".cpp"};

    return (filename.length() > cpp_file.length())
        && (filename.substr(filename.length() - cpp_file.length()) == cpp_file);
}

TEST(DesignPatternA, Strategy)
{
    auto sort = [](&auto& v) {
        std::sort(v.begin(), v.end());
        return v;
}

```

```

};

{
    auto exp = sort(std::vector<std::string>{
        "../ut_data/a.cpp", "../ut_data/a.h", "../ut_data/abc.cpp", "../ut_data/abc.h",
        "../ut_data/d/a.d", "../ut_data/efghij.cpp", "../ut_data/efghij.h",
        "../ut_data/lib/lib.cpp", "../ut_data/lib/lib.h", "../ut_data/o/a.o"});
    auto act = find_files("../ut_data",
        [] (fs::path const& p) noexcept { return fs::is_regular_file(p); });

    ASSERT_EQ(exp, act);
}

{
    auto exp = sort(std::vector<std::string>{"../ut_data/d", "../ut_data/lib", "../ut_data/o"});
    auto act = find_files("../ut_data",
        [] (fs::path const& p) noexcept { return fs::is_directory(p); });

    ASSERT_EQ(exp, act);
}

{
    auto exp
        = sort(std::vector<std::string>{"../ut_data/a.cpp", "../ut_data/abc.cpp",
            "../ut_data/efghij.cpp", "../ut_data/lib/lib.cpp"});
    auto act = find_files("../ut_data", is_cpp_file);

    ASSERT_EQ(exp, act);
}
}

```

- 演習-Strategyへ戻る。

解答例-Visitor

```

// @@@ exercise/design_pattern_a/visitor.cpp 9
// [A]
// 下記クラスFile、Dir、OtherEntityはクラスFileEntityから派生し、
// それぞれが自身をstd::stringに変換するアルゴリズム関数
//     * to_string_normal()
//     * to_string_with_char()
//     * to_string_with_children()
// をオーバーライドしている。これはポリモーフィズムの使用方法としては正しいが、
// to_string_xxx系のインターフェースが大量に増えた場合に、
// FileEntityのインターフェースがそれに比例して増えてしまう問題を持っている。
// Visitorパターンを使用しこれに対処せよ。

// [解説]
// * 通常の例では、Visitor::Visit()の戻り値はvoidになっていることが多いが、この例では
// std::stringにした。Visitorパターンは戻り値が同じでなければ適用できない。
// * Visitorパターンは静的型付け言語のダブルディスパッチと呼ばれるテクニックを使っている。
//     * 1目目のディスパッチは、Visitor::Visitのオーバーロードによって行われる。
//     * 2目目のディスパッチは、Visitorの派生クラスのオーバーライドによって行われる。
// * デザインパターンとはそういうものであるが、この例でもVisitorの導入によって返って複雑になった。
// しかし、to_string_xxxのようなアルゴリズムが10個、20個となるような場合には、
// クラスの肥大化を防ぐ有用な手段となる。

class Visitor;

class FileEntity {
public:
    explicit FileEntity(std::string const& pathname) : pathname_{strip(pathname)} {}
    virtual ~FileEntity() = default;
    std::string const& Pathname() const noexcept { return pathname_; }
    std::string ToString(Visitor const& to_s) const { return to_string(to_s); }

private:
    std::string const pathname_;

    virtual std::string to_string(Visitor const& to_s) const = 0;
    static std::string strip(std::string const& pathname)
    {
        return std::regex_replace(pathname, std::regex(R"(^/+)"), "");
    }
};

class File;
class Dir;
class OtherEntity;

class Visitor {

```

```

public:
    virtual ~Visitor() = default;
    std::string Visit(File const& file) const { return visit(file); }
    std::string Visit(Dir const& dir) const { return visit(dir); }
    std::string Visit(OtherEntity const& other) const { return visit(other); }

private:
    virtual std::string visit(File const& file) const = 0;
    virtual std::string visit(Dir const& dir) const = 0;
    virtual std::string visit(OtherEntity const& f) const = 0;
};

class File final : public FileEntity {
public:
    explicit File(std::string const& pathname) : FileEntity{pathname} {}

private:
    virtual std::string to_string(Visitor const& to_s) const { return to_s.Visit(*this); }
};

class Dir final : public FileEntity {
public:
    explicit Dir(std::string const& pathname) : FileEntity{pathname} {}

private:
    virtual std::string to_string(Visitor const& to_s) const { return to_s.Visit(*this); }
};

class OtherEntity final : public FileEntity {
public:
    explicit OtherEntity(std::string const& pathname) : FileEntity{pathname} {}

private:
    virtual std::string to_string(Visitor const& to_s) const { return to_s.Visit(*this); }
};

class ToStringNormal : public Visitor {
private:
    virtual std::string visit(File const& file) const override { return file.Pathname(); }
    virtual std::string visit(Dir const& dir) const override { return dir.Pathname() + '/'; }
    virtual std::string visit(OtherEntity const& other) const override { return other.Pathname(); }
};

class ToStringWithChar : public Visitor {
private:
    virtual std::string visit(File const& file) const override { return file.Pathname(); }
    virtual std::string visit(Dir const& dir) const override { return dir.Pathname() + '/'; }
    virtual std::string visit(OtherEntity const& other) const override
    {
        return other.Pathname() + '+';
    }
};

class ToStringWithChildren : public Visitor {
private:
    virtual std::string visit(File const& file) const override { return file.Pathname(); }
    virtual std::string visit(Dir const& dir) const override { return find_files(dir.Pathname()); }
    virtual std::string visit(OtherEntity const& other) const override { return other.Pathname(); }

    static std::string find_files(std::string const& dir)
    {
        namespace fs = std::filesystem;

        auto files = std::vector<std::string>{};

        std::for_each(fs::recursive_directory_iterator{dir}, fs::recursive_directory_iterator{},
                     [&files](fs::path const& p) { files.emplace_back(p.generic_string()); });

        std::sort(files.begin(), files.end());
    }

    auto ret = std::string{dir};

    for (auto f : files) {
        ret += ' ' + f;
    }

    return ret;
}
};

```

```

TEST(DesignPatternA, Visitor)
{
    auto ts_normal = ToStringNormal{};
    auto ts_char = ToStringWithChar{};
    auto ts_children = ToStringWithChildren{};

    auto const f0 = File{"../ut_data/a.cpp"};
    auto const f1 = File{"../ut_data/a.cpp///"};

    ASSERT_EQ("../ut_data/a.cpp", f0.Pathname());
    ASSERT_EQ("../ut_data/a.cpp", f0.Pathname());

    ASSERT_EQ("../ut_data/a.cpp", f0.ToString(ts_normal));
    ASSERT_EQ("../ut_data/a.cpp", f0.ToString(ts_char));
    ASSERT_EQ("../ut_data/a.cpp", f0.ToString(ts_children));

    auto const dir = Dir{"../ut_data/lib/"};

    ASSERT_EQ("../ut_data/lib", dir.Pathname());
    ASSERT_EQ("../ut_data/lib/", dir.ToString(ts_normal));
    ASSERT_EQ("../ut_data/lib/", dir.ToString(ts_char));
    ASSERT_EQ("../ut_data/lib ../ut_data/lib/lib.cpp ../ut_data/lib/lib.h",
              dir.ToString(ts_children));

    auto const other = OtherEntity{"symbolic_link"};

    ASSERT_EQ("symbolic_link", other.Pathname());
    ASSERT_EQ("symbolic_link", other.ToString(ts_normal));
    ASSERT_EQ("symbolic_link+", other.ToString(ts_char));
    ASSERT_EQ("symbolic_link", other.ToString(ts_children));
}

```

- 演習-Visitorへ戻る。

解答例-CRTP

```

// @@@ exercise/design_pattern_a/crtpp.cpp 9
// [A]
// 下記クラスFileEntityから派生しクラスFile、Dir、OtherEntityは、
// Visitorパターンを利用しているため、そのすべてで下記のコードクローンを持つ。
//
//     virtual std::string to_string(Visitor const& to_s) const { return to_s.Visit(*this); }
//
// このコードクローンのthisの型は、それぞれFile、Dir、OtherEntityとなるため、
// この関数をFileEntityで定義すると動作が変わってしまい、単純には統一できない。
// CRTPを用い、このクローンを削除せよ。
//
// [解説]
// 下記のクラステンプレートAcceptableFileEntityと、
// それから派生したFile、Dir、OtherEntityがCRTPを実装し、コードクローンを排除した。

class Visitor;

class FileEntity {
public:
    explicit FileEntity(std::string const& pathname) : pathname_{strip(pathname)} {}
    virtual ~FileEntity() = default;
    std::string const& Pathname() const { return pathname_; }
    std::string ToString(Visitor const& to_s) const { return to_s.to_string(to_s); }

private:
    std::string const pathname_;

    virtual std::string to_string(Visitor const& to_s) const = 0;
    static std::string strip(std::string const& pathname)
    {
        return std::regex_replace(pathname, std::regex{R"(/+$)"}, "");
    }
};

class File;
class Dir;
class OtherEntity;

class Visitor {
public:
    virtual ~Visitor() = default;
    std::string Visit(File const& file) const { return visit(file); }
    std::string Visit(Dir const& dir) const { return visit(dir); }
}

```

```

        std::string Visit(OtherEntity const& other) const { return visit(other); }

private:
    virtual std::string visit(File const& file) const = 0;
    virtual std::string visit(Dir const& dir) const = 0;
    virtual std::string visit(OtherEntity const& f) const = 0;
};

template <typename T>
class AcceptableFileEntity : public FileEntity { // CRTP
private:
    virtual std::string to_string(Visitor const& to_s) const
    {
        return to_s.Visit(*static_cast<T const*>(this));
    }

// T : public AcceptableFileEntity<T> { ... };
// 以外の使い方をコンパイルエラーにする
AcceptableFileEntity(std::string const& pathname) : FileEntity{pathname} {}
friend T;
};

class File final : public AcceptableFileEntity<File> {
public:
    File(std::string const& pathname) : AcceptableFileEntity{pathname} {}
};

class Dir final : public AcceptableFileEntity<Dir> {
public:
    Dir(std::string const& pathname) : AcceptableFileEntity{pathname} {}
};

class OtherEntity final : public AcceptableFileEntity<OtherEntity> {
public:
    OtherEntity(std::string const& pathname) : AcceptableFileEntity{pathname} {}
};

class ToStringNormal : public Visitor {
private:
    virtual std::string visit(File const& file) const override { return file.Pathname(); }
    virtual std::string visit(Dir const& dir) const override { return dir.Pathname() + '/'; }
    virtual std::string visit(OtherEntity const& other) const override { return other.Pathname(); }
};

class ToStringWithChar : public Visitor {
private:
    virtual std::string visit(File const& file) const override { return file.Pathname(); }
    virtual std::string visit(Dir const& dir) const override { return dir.Pathname() + '/'; }
    virtual std::string visit(OtherEntity const& other) const override
    {
        return other.Pathname() + '+';
    }
};

class ToStringWithChildren : public Visitor {
private:
    virtual std::string visit(File const& file) const override { return file.Pathname(); }
    virtual std::string visit(Dir const& dir) const override { return find_files(dir.Pathname()); }
    virtual std::string visit(OtherEntity const& other) const override { return other.Pathname(); }

    static std::string find_files(std::string const& dir)
    {
        namespace fs = std::filesystem;

        auto files = std::vector<std::string>{};

        std::for_each(fs::recursive_directory_iterator{dir}, fs::recursive_directory_iterator{},
                     [&files](fs::path const& p) { files.emplace_back(p.generic_string()); });

        std::sort(files.begin(), files.end());
    }

    auto ret = std::string{dir};

    for (auto f : files) {
        ret += ' ' + f;
    }

    return ret;
}
};

```

```

TEST(DesignPatternA, CRTP)
{
    auto ts_normal = ToStringNormal{};
    auto ts_char = ToStringWithChar{};
    auto ts_children = ToStringWithChildren{};

    auto const f0 = File{"../ut_data/a.cpp"};
    auto const f1 = File{"../ut_data/a.cpp///"};

    ASSERT_EQ("../ut_data/a.cpp", f0.Pathname());
    ASSERT_EQ("../ut_data/a.cpp", f0.Pathname());

    ASSERT_EQ("../ut_data/a.cpp", f0.ToString(ts_normal));
    ASSERT_EQ("../ut_data/a.cpp", f0.ToString(ts_char));
    ASSERT_EQ("../ut_data/a.cpp", f0.ToString(ts_children));

    auto const dir = Dir{"../ut_data/lib/"};

    ASSERT_EQ("../ut_data/lib", dir.Pathname());
    ASSERT_EQ("../ut_data/lib/", dir.ToString(ts_normal));
    ASSERT_EQ("../ut_data/lib/", dir.ToString(ts_char));
    ASSERT_EQ("../ut_data/lib ../ut_data/lib/lib.cpp ../ut_data/lib/lib.h",
              dir.ToString(ts_children));

    auto const other = OtherEntity{"symbolic_link"};

    ASSERT_EQ("symbolic_link", other.Pathname());
    ASSERT_EQ("symbolic_link", other.ToString(ts_normal));
    ASSERT_EQ("symbolic_link+", other.ToString(ts_char));
    ASSERT_EQ("symbolic_link", other.ToString(ts_children));
}

```

- 演習-CRTPへ戻る。

解答例-Observer

```

// @@@ exercise/design_pattern_a/observer.cpp 9
// [A]
// 下記クラスはそれぞれが
//   * ViewX、ViewY : GUIへの出力(描画)
//   * Model       : 何らかのビジネスロジックの演算
//   * Controller  : OKボタンクリックイベントをModelへ通知
// 行うことを模擬している。
// 依存関係Model->ViewX、ViewYはMVCに逆行しているため下記のような問題を持つ。
//   * ViewX、ViewYの変更がModelに伝搬してしまう。
//   * この例は単純であるためViewX、ViewY->Modelへの依存関係は存在しないが、
//     実際のアプリケーションではそのような依存関係が存在するため、依存関係が循環してしまう。
//   * ModelがダイレクトにViewX、ViewYへ出力するため、単体テストの実施は困難である。
//   * この依存関係が直接の原因ではないが、このような依存関係を持つアプリケーションのクラスは
//     巨大になる。
// アプリケーションが小規模である時には、このような問題がバグや開発効率悪化の原因となることは稀
// であり放置されることが多いが、大規模化に伴いこのような潜在的問題が表出する。
// ModelにObserverパターンを適用する等をしてこの問題に対処するとともに、Modelの単体テストを行え。

// [解説]
// * Observerパターンの使用について
//   * ObserverIFはModelを宣言、定義しているヘッダファイルに定義する。
//   * 何らかの理由でそうしない場合は、ObserverIFはModelを含むパッケージ内に定義する。
// * Observerパターンの導入により、
//   * ModelはViewX、ViewYに依存しなくなったが、代わりにObserverIFに依存する。
//   * この依存関係はObserverIFがModelヘッダに含まれることで問題にならない。
// * その他
//   * 通常は、Attachに対してDetachも定義するが、AttachされたオブジェクトをDetachしない
//     場合は、今回の例のようにAttachされたオブジェクトの廃棄をModelにさせた方が良い。
//   * こういった動作の確認にもForTestクラスを使用した。

class ObserverIF {
public:
    void DisplaySomething(std::string const& result) { display_something(result); }
    virtual ~ObserverIF() = default;
};

private:
    virtual void display_something(std::string const&) = 0;
};

class Model {
public:
    Model() = default;
}

```

```

~Model() { wait_future(); }

// Detachできない仕様にする。その代わりにobserverの廃棄もModelに任せることができる。
void Attach(std::unique_ptr<ObserverIF> observer)
{
    observers_.emplace_back(std::move(observer));
}

void DoSomething()
{
    wait_future();

    future_ = std::async(std::launch::async, [this] {
        // 本来は非同期処理が必要な重い処理
        auto result = std::string("result of doing something");

        notify(result);
    });
}

private:
    std::future<void> future_{};
    std::vector<std::unique_ptr<ObserverIF>> observers_{};

    void notify(std::string const& result) const
    {
        for (auto& observer : observers_) {
            observer->DisplaySomething(result);
        }
    }

    void wait_future()
    {
        if (future_.valid()) {
            future_.wait();
        }
    }
};

class ViewX : public ObserverIF {
private:
    virtual void display_something(std::string const&) override {}
};

class ViewY : public ObserverIF {
private:
    virtual void display_something(std::string const&) override {}
};

class Controller {
public:
    Controller(Model& model) : model_{model} {}
    void OK_Clicked() { model_.DoSomething(); }

    Model& model_;
};

class ForTest : public ObserverIF {
public:
    explicit ForTest(std::string& result, uint32_t& called, bool& destructed)
        : result_{result}, called_{called}, destructed_{destructed}
    {
    }
    virtual ~ForTest() override { destructed_ = true; }

private:
    virtual void display_something(std::string const& result) noexcept override
    {
        result_ = result;
        ++called_;
    }

    std::string& result_;
    uint32_t& called_;
    bool& destructed_;
};

TEST(DesignPatternA, Observer)
{
    auto result      = std::string{};


```

```

    auto called      = 0U;
    auto destructed = false;

    {
        auto model      = Model{};
        auto controller = Controller{model};

        model.Attach(std::make_unique<ViewX>());
        model.Attach(std::make_unique<ViewY>());
        model.Attach(std::make_unique<ForTest>(result, called, destructed));

        controller.OK_Clicked();
        controller.OK_Clicked();
        controller.OK_Clicked();
    }

    ASSERT_EQ("result of doing something", result);
    ASSERT_EQ(3, called);
    ASSERT_TRUE(destructed);
}

```

- 演習-Observerへ戻る。

解答-デザインパターン選択1

- 選択肢2
- 参照 State
- 演習-デザインパターン選択1へ戻る。

解答-デザインパターン選択2

- 選択肢4
- 参照 Null Object
- 演習-デザインパターン選択2へ戻る。

解答-デザインパターン選択3

- 選択肢3
- 参照 Observer
- 演習-デザインパターン選択3へ戻る。

開発プロセスとインフラ(全般)

解答-プロセス分類

- 選択肢対応
 - A - 1
 - B - 2
 - C - 3
- 参照 プロセス
- 演習-プロセス分類へ戻る。

解答-V字モデル

- 選択肢対応
 - フェーズA - 4
 - フェーズB - 2
 - フェーズC - 3
 - フェーズD - 1
- 参照 ウォーターフォールモデル、V字モデル
- 演習-V字モデルへ戻る。

解答-アジャイル

- 選択肢2

- 参照 [アジャイル系プロセス](#)
- 演習-アジャイルへ戻る。

解答-自動化

- 選択肢対応
 - A - 1
 - B - 4
 - C - 5
- 参照 [自動単体テスト](#)
- 演習-自動化へ戻る。

解答-単体テスト

- 選択肢4
- 参照 [自動単体テスト](#)
- 演習-単体テストへ戻る。

解答-リファクタリングに付随する活動

- 選択肢2
- 参照 [リファクタリング](#)
- 演習-リファクタリングに付随する活動へ戻る。

解答-リファクタリング対象コード

- 選択肢3
- 参照 [リファクタリング](#)
- 演習-リファクタリング対象コードへ戻る。

解答-CI

- 選択肢2, 4
- 参照 [CI\(継続的インテグレーション\)](#)
- 演習-CIへ戻る。

テンプレートメタプログラミング

解答例-パラメータパック

```
// @@@ exercise/template_a/parameter_pack.cpp 7
// [A]
// 下記の関数Maxは、単体テストが示す通り、2つのパラメータの大きい方を返す。
// 任意の個数の引数を取れるようにMaxを修正せよ。

// 解答例1
// パラメータパックを使用しMaxを修正した例
template <typename T>
T Max(T const& t0, T const& t1) noexcept
{
    return t0 > t1 ? t0 : t1;
}

template <typename HEAD, typename... ARGS>
auto Max(HEAD const& head, ARGS const&... args) noexcept
{
    auto args_max = Max(args...);

    return head > args_max ? head : args_max;
}

TEST(TemplateMetaProgrammingA, parameter_pack)
{
    ASSERT_EQ(2, Max(1, 2));
    ASSERT_EQ("bcd", Max(std::string("abc"), std::string("bcd")));
}
```

```

    ASSERT_EQ(3, Max(1, 2, 3));
    ASSERT_EQ("efg", Max(std::string("abc"), std::string("bcd"), std::string("efg")));
}

// 解答例2
// std::initializer_listを使用しMaxを修正した例
template <typename T>
T Max(std::initializer_list<T> t_list) noexcept
{
    auto ret = T{};
    auto first = true;

    for (auto const& t : t_list) {
        if (std::exchange(first, false)) {
            ret = t;
        }
        else {
            ret = Max(ret, t);
        }
    }

    return ret;
}

TEST(TemplateMetaProgrammingA, initializer_list)
{
    ASSERT_EQ(2, Max({1, 2}));
    ASSERT_EQ("bcd", Max({std::string("abc"), std::string("bcd")}));

    ASSERT_EQ(3, Max({1, 2, 3}));
    ASSERT_EQ("efg", Max({std::string("abc"), std::string("bcd"), std::string("efg")}));
}

```

- 演習-パラメータパックへ戻る。

解答例-エイリアステンプレート

```

// @@@ exercise/template_a/template_alias.cpp 5
// [A]
// 下記の単体テストでしているstd::vector<std::vector<XXX>>を、
// テンプレートエイリアスによって簡潔に記述せよ。

// 解説
// 下記のVect1Dはstd::vectorに対して簡潔な記述方法を提供しているとは言えないが、
// Vect2Dと同じような場面で使用することが明示されるため、ソースコードに一貫性を与える。

template <typename T>
using Vect2D = std::vector<std::vector<T>>;

template <typename T>
using Vect1D = std::vector<T>;

TEST(TemplateMetaProgrammingA, template_alias)
{
    {
        auto vv = Vect2D<int>{{1, 2, 3}, {3, 4, 5}};
        ASSERT_EQ(2, vv.size());
        ASSERT_EQ((Vect1D<int>{1, 2, 3}), vv[0]);
        ASSERT_EQ((Vect1D<int>{3, 4, 5}), vv[1]);
        ASSERT_EQ(5, vv[1][2]);
    }
    {
        auto vv = Vect2D<float>{{1, 2, 3}, {3, 4, 5}};
        ASSERT_EQ(2, vv.size());
        ASSERT_EQ((Vect1D<float>{1, 2, 3}), vv[0]);
        ASSERT_EQ((Vect1D<float>{3, 4, 5}), vv[1]);
        ASSERT_EQ(5, vv[1][2]);
    }
    {
        auto vv = Vect2D<std::string>{{"1", "2", "3"}, {"3", "4", "5"}};
        ASSERT_EQ(2, vv.size());
        ASSERT_EQ((Vect1D<std::string>{"1", "2", "3"}), vv[0]);
        ASSERT_EQ((Vect1D<std::string>{"3", "4", "5"}), vv[1]);
        ASSERT_EQ("5", vv[1][2]);
    }
}

```

- 演習-エイリアステンプレートへ戻る。

解答例-名前空間による修飾不要なoperator<<

```
// @@@ exercise/template_a/put_to.cpp 3
// [A]
// 下記のように名前空間TemplateMP、エイリアスInts_tとそのoperator<<が定義されている場合、
// 単体テストで示した通り、Ints_tのoperator<<を使用するためには、
// 名前空間による修飾やusing宣言/ディレクティブの記述が必要になる。
// Ints_tをstd::vectorから継承したクラスとして定義することにより、このような記述を不要にせよ。

namespace TemplateMP {

    struct Ints_t : std::vector<int> {
        using std::vector<int>::vector; // 継承コンストラクタ
    };

    std::ostream& operator<<(std::ostream& os, Ints_t const& ints)
    {
        auto first = true;
        for (auto i : ints) {
            if (!std::exchange(first, false)) {
                os << " : ";
            }
            os << i;
        }

        return os;
    }
} // namespace TemplateMP

namespace {
TEST(TemplateMetaProgrammingA, put_to)
{
    {
        auto oss = std::ostringstream{};
        auto ints = TemplateMP::Ints_t{1, 2, 3};

        oss << ints; // ADLによるname lookup

        ASSERT_EQ("1 : 2 : 3", oss.str());
    }
}
} // namespace
```

- 演習-名前空間による修飾不要なoperator<<へ戻る。

解答例-std::arrayの継承

```
// @@@ exercise/template_a/safe_array.cpp 5
// [A]
// std::array、std::vector、std::string等のSTLの配列型コンテナはインデックスアクセスに対して、
// レンジのチェックをしないため、不正なメモリアクセスをしてしまうことがある。
// std::arrayを使用して、このような問題のないSafeArrayを作り、単体テストを行え。

template <typename T, std::size_t N>
struct SafeArray : std::array<T, N> {
    using std::array<T, N>::array; // 継承コンストラクタ

    template <typename... ARGS> // コンストラクタを定義
    SafeArray(ARGS... args) noexcept(std::is_nothrow_constructible_v<T, ARGS...>)
        : base_type{args...}
    {

    }

    using base_type = std::array<T, N>;
    using size_type = typename base_type::size_type;

    typename base_type::reference operator[](size_type i) { return this->at(i); }
    typename base_type::const_reference operator[](size_type i) const { return this->at(i); }
};

namespace {
TEST(TemplateMetaProgrammingA, safe_array)
{
    auto sa = SafeArray<int, 3>{1, 2, 3};

    static_assert(std::is_nothrow_constructible_v<decltype(sa), int>);
    ASSERT_EQ(3, sa.size());
    ASSERT_EQ(1, sa[0]);
}
```

```

    ASSERT_EQ(2, sa[1]);
    ASSERT_EQ(3, sa[2]);
    ASSERT_THROW(sa[3], std::out_of_range);

    auto sa2 = SafeArray<std::string, 2>{"1", "2"};

    static_assert(std::is_nothrow_constructible_v<decltype(sa2), char const*>);
    ASSERT_EQ(2, sa2.size());
    ASSERT_EQ("1", sa2[0]);
    ASSERT_EQ("2", sa2[1]);
    ASSERT_THROW(sa2[2], std::out_of_range);
}
} // namespace

```

- 演習-`std::array`の継承へ戻る。

解答例-SFINAEを利用しない関数テンプレートの特殊化による`is_void`

```

// @@@ exercise/template_a/is_void.cpp 3
// [A]
// 下記の仕様を満たす関数テンプレートis_void_f<T>と定数テンプレートis_void_f_v<T>を作れ。
//   * 与えられたテンプレートパラメータがvoidの場合、trueを返す
//   * 与えられたテンプレートパラメータがvoidでない場合、falseを返す
//   * std::is_sameを使わない
//   * 下記の単体テストをパスする(#if 0を削除してもコンパイルできる)

template <typename T>
constexpr bool is_void_f() noexcept
{
    return false;
}

template <>
constexpr bool is_void_f<void>() noexcept
{
    return true;
}

template <typename T>
constexpr bool is_void_f_v{is_void_f<T>}();

namespace IsVoidTest {
void test_func_0() noexcept {};
std::string test_func_1() { return "test"; };
} // namespace IsVoidTest

namespace {

TEST(TemplateMetaProgrammingA, is_void_f)
{
    static_assert(!is_void_f_v<int>());
    static_assert(is_void_f_v<void>());
    static_assert(is_void_f_v<decltype(IsVoidTest::test_func_0())>());
    static_assert(!is_void_f_v<decltype(IsVoidTest::test_func_1())>());
}
} // namespace

```

- 演習-SFINAEを利用しない関数テンプレートの特殊化による`is_void`へ戻る。

解答例-SFINAEを利用しないクラステンプレートの特殊化による`is_void`

```

// @@@ exercise/template_a/is_void.cpp 43
// [A]
// 下記の仕様を満たすクラステンプレートis_void_s<T>と定数テンプレートis_void_s_v<T>を作れ。
//   * 与えられたテンプレートパラメータがvoidの場合、メンバvalueがtrueになる
//   * 与えられたテンプレートパラメータがvoidでない場合、メンバvalueがfalseになる
//   * std::is_sameを使わない
//   * std::true_type/std::false_typeを利用する
//   * 下記の単体テストをパスする(#if 0を削除してもコンパイルできる)

template <typename T>
struct is_void_s : std::false_type {
};

template <>
struct is_void_s<void> : std::true_type {
};

```

```

template <typename T>
constexpr bool is_void_s_v{is_void_s<T>::value};

namespace {
TEST(TemplateMetaProgrammingA, is_void_s)
{
    static_assert(!is_void_s_v<int>);
    static_assert(is_void_s_v<void>);
    static_assert(is_void_s_v<decltype(IsVoidTest::test_func_0())>);
    static_assert(!is_void_s_v<decltype(IsVoidTest::test_func_1())>);
}
} // namespace

```

- 演習-SFINAEを利用しないクラステンプレートの特殊化によるis_voidへ戻る。

解答例-SFINAEを利用した関数テンプレートの特殊化によるis_void

```

// @@@ exercise/template_a/is_void.cpp 75
// [A]
// 下記の仕様を満たす関数テンプレートis_void_sfinae_f<T>と
// 定数テンプレートis_void_sfinae_f<T>を作れ。
//   * 与えられたテンプレートパラメータがvoidの場合、trueを返す
//   * 与えられたテンプレートパラメータがvoidでない場合、falseを返す
//   * std::is_sameを使わない
//   * SFINAEを利用する
//   * 下記の単体テストをパスする(#if 0を削除してもコンパイルできる)

namespace Inner_ {

// T == void
template <typename T>
constexpr auto is_void_sfinae_f_detector(void const* v, T const* t) noexcept
    -> decltype(t = v, bool{}) // T != voidの場合、t = vはill-formed
                                // T == voidの場合、well-formedでbool型生成
{
    return true;
}

constexpr auto is_void_sfinae_f_detector(...) noexcept // name lookupの順位は最低
{
    return false;
}
} // namespace Inner_

template <typename T>
constexpr bool is_void_sfinae_f() noexcept
{
    return Inner_::is_void_sfinae_f_detector(nullptr, static_cast<T*>(nullptr));
}

template <typename T>
constexpr bool is_void_sfinae_f_v{is_void_sfinae_f<T>()};

namespace {

TEST(TemplateMetaProgrammingA, is_void_sfinae_f)
{
    static_assert(!is_void_sfinae_f_v<int>);
    static_assert(is_void_sfinae_f_v<void>);
    static_assert(is_void_sfinae_f_v<decltype(IsVoidTest::test_func_0())>);
    static_assert(!is_void_sfinae_f_v<decltype(IsVoidTest::test_func_1())>);
}
} // namespace

```

- 演習-SFINAEを利用した関数テンプレートの特殊化によるis_voidへ戻る。

解答例-SFINAEを利用したクラステンプレートの特殊化によるis_void

```

// @@@ exercise/template_a/is_void.cpp 123
// [A]
// 下記の仕様を満たすクラステンプレートis_void_sfinae_s<T>と
// 定数テンプレートis_void_sfinae_s_v<T>を作れ。
//   * 与えられたテンプレートパラメータがvoidの場合、メンバvalueがtrueになる
//   * 与えられたテンプレートパラメータがvoidでない場合、メンバvalueがtrueになる
//   * std::is_sameを使わない
//   * std::true_type/std::false_typeを利用する
//   * SFINAEを利用する

```

```

//     * 下記の単体テストをパスする(#if 0を削除してもコンパイルできる)
namespace Inner_ {

template <typename T>
T*& t2ptr();

} // namespace Inner_

template <typename T, typename = void*&>
struct is_void_sfinae_s : std::false_type {
};

template <typename T>
struct is_void_sfinae_s<T, decltype(Inner_::t2ptr<T>() = Inner_::t2ptr<void>())> : std::true_type {
};

template <typename T>
constexpr bool is_void_sfinae_s_v{is_void_sfinae_s<T>::value};

namespace {

TEST(TemplateMetaProgrammingA, is_void_sfinae_s)
{
    static_assert(!is_void_sfinae_s_v<int>);
    static_assert(is_void_sfinae_s_v<void>);
    static_assert(is_void_sfinae_s_v<decltype(IsVoidTest::test_func_0())>);
    static_assert(!is_void_sfinae_s_v<decltype(IsVoidTest::test_func_1())>);

}
} // namespace

```

- 演習-SFINAEを利用したクラステンプレートの特殊化によるis_voidへ戻る。

解答例-テンプレートテンプレートパラメータ

```

// @@@ exercise/template_a/template_template.cpp 5
// [A]
// 以下の仕様を満たすクラステンプレートを作れ。
// * 任意のSTLコンテナを唯一のテンプレートパラメータとする
// * そのコンテナを使用しint型のデータを格納する

template <template <class...> class STL_CONTAINER>
struct IntContainer : STL_CONTAINER<int> {
    using STL_CONTAINER<int>::STL_CONTAINER;
};

namespace {

TEST(TemplateMetaProgrammingA, template_template)
{
    auto vi = IntContainer<std::vector>{1, 2, 3};
    auto vl = IntContainer<std::list>{1, 2, 3};
    auto vs = IntContainer<std::basic_string>{1, 2, 3}; // 意味は不明だがこれも可能

    ASSERT_EQ((std::vector<int>{1, 2, 3}), vi);
    ASSERT_EQ((std::list<int>{1, 2, 3}), vl);
    ASSERT_EQ((std::basic_string<int>{1, 2, 3}), vs);
}
} // namespace

```

- 演習-テンプレートテンプレートパラメータへ戻る。

解答例-テンプレートパラメータを可変長にしたstd::is_same

```

// @@@ exercise/template_a/is_same.cpp 3
// [A]
// 以下の仕様を満たすクラステンプレートis_same_some_of<T, U...>と
// 定数テンプレートis_same_same_of_v<T, U...>を作れ。
// * 2個以上のテンプレートパラメータを持つ
// * 第1パラメータと他のパラメータの何れかが同一の型であった場合、メンバvalueがtrueになる
// * 前行の条件が成立しなかった場合、メンバvalueがfalseになる
// * 型の同一性はstd::is_sameを使って判定する

template <typename T, typename U, typename... Us>
struct is_same_some_of {
    static constexpr bool value{std::is_same_v<T, U> ? true : is_same_some_of<T, Us...>::value};
};

```

```

template <typename T, typename U>
struct is_same_some_of<T, U> {
    static constexpr bool value{std::is_same_v<T, U>};
};

template <typename T, typename U, typename... Us>
constexpr bool is_same_some_of_v{is_same_some_of<T, U, Us...>::value};

namespace {

TEST(TemplateMetaProgrammingA, is_same_some_of)
{
    static_assert(!is_same_some_of_v<int, int8_t, int16_t, uint16_t>);
    static_assert(is_same_some_of_v<int, int8_t, int16_t, uint16_t, int32_t>);
    static_assert(is_same_some_of_v<int&, int8_t, int16_t, int32_t&, int32_t>);
    static_assert(!is_same_some_of_v<int&, int8_t, int16_t, uint32_t&, int32_t>);
    static_assert(is_same_some_of_v<std::string, int, char*, std::string>);
    static_assert(!is_same_some_of_v<std::string, int, char*>);
}
} // namespace

```

- 演習-テンプレートパラメータを可変長にしたstd::is_sameへ戻る。

解答例-メンバ関数の存在の診断

```

// @@@ exercise/template_a/exists_func.cpp 5
// [A]
// テンプレートパラメータの型がメンバ関数c_str()を持つか否かを判定する
// クラステンプレートhas_c_str<T>と定数テンプレートhas_c_str_v<T>を作れ。

template <typename T, typename U = bool>
struct has_c_str : std::false_type {
};

template <typename T>
struct has_c_str<T, decltype(std::declval<T>().c_str(), bool{})> : std::true_type {
};

template <typename T>
constexpr bool has_c_str_v{has_c_str<T>::value};

namespace {

TEST(TemplateMetaProgrammingA, has_c_str)
{
    static_assert(has_c_str_v<std::string>);
    static_assert(!has_c_str_v<std::vector<int>>);
}
} // namespace

```

- 演習-メンバ関数の存在の診断へ戻る。

解答例-範囲for文のオペランドになれるかどうかの診断

```

// @@@ exercise/template_a/exists_func.cpp 31
// [A]
// 範囲for文は、
//     for(auto a : obj) { ... }
// のような形式で表現される。
// テンプレートパラメータから生成されたオブジェクトが、
// このobjに指定できるか否かを判定するクラステンプレートis_range<T>
// と定数テンプレートis_range_v<T>を作れ。

// 解説
// 上記objに指定できるための条件は、std::begin()、std::end()の引数になれるとした。
// セマンティクス的に正しいstd::begin()、std::end()は、それぞれが最初と最後を表す
// イテレータ(もしくはポインタ)でなければならないが、それはテンプレートでの判定の範囲外である。

template <typename, typename = bool>
struct exists_begin : std::false_type {
};

template <typename T>
struct exists_begin<T, decltype(std::begin(std::declval<T&>())>, bool{})> : std::true_type {
};

template <typename T>
constexpr bool exists_begin_v{exists_begin<T>::value};

```

```

template <typename, typename = bool>
struct exists_end : std::false_type {
};

template <typename T>
struct exists_end<T, decltype(std::end(std::declval<T&>()), std::bool{})> : std::true_type {
};

template <typename T>
constexpr bool exists_end_v{exists_end<T>::value};

template <typename T>
struct is_range
: std::conditional_t<exists_begin_v<T> && exists_end_v<T>, std::true_type, std::false_type> {};

template <typename T>
constexpr bool is_range_v{is_range<T>::value};

namespace {

TEST(TemplateMetaProgrammingA, is_range)
{
    static_assert(is_range_v<std::string>);
    static_assert(is_range_v<std::vector<int>>);
    static_assert(!is_range_v<std::mutex>);
    static_assert(!is_range_v<std::lock_guard<std::mutex>>);
}
} // namespace

```

- 演習範囲for文のオペランドになれるかどうかの診断へ戻る。

解答例-配列の長さの取り出し

```

// @@@ exercise/template_a/array_op.cpp 3
// [A]
// 配列を引数に取り、その長さを返す関数テンプレートarray_lengthを作れ。

template <typename T, size_t N>
constexpr size_t array_length(T const (&)[]N) noexcept
{
    return N;
}

namespace {

TEST(TemplateMetaProgrammingA, array_length)
{
    int i[5];
    std::string str[]{"a", "b", "c"};

    static_assert(array_length(i) == 5);
    static_assert(array_length(str) == 3);
}
} // namespace

```

- 演習配列の長さの取り出しへ戻る。

解答例-配列の次元の取り出し

```

// @@@ exercise/template_a/array_op.cpp 26
// [A]
// 配列を引数に取り、その次元を返す関数テンプレートarray_dimensionを作れ。

constexpr size_t array_dimension(...) noexcept { return 0; }

template <typename T, size_t N>
constexpr size_t array_dimension(T const (&t)[N]) noexcept
{
    return 1 + array_dimension(t[0]);
}

namespace {

TEST(TemplateMetaProgrammingA, array_dimension)
{
    constexpr int i1[5]{};

```

```

constexpr int i2[5][2]{};
constexpr int i3[5][2][3]{};

static_assert(array_dimension(i1) == 1);
static_assert(array_dimension(i2) == 2);
static_assert(array_dimension(i3) == 3);
}
} // namespace

```

- 演習-配列の次元の取り出しへ戻る。

解答例-関数型のテンプレートパラメータを持つクラステンプレート

```

// @@@ exercise/template_a/scoped_guard.cpp 8
// [A]
// RAIIを行うための下記クラスscoped_guardをstd::functionを使わずに再実装せよ。

template <typename FUNC>
class scoped_guard {
public:
    explicit scoped_guard(FUNC&& f) noexcept : f_{f}
    {
        static_assert(std::is_nothrow_invocable_r_v<void, FUNC>, "FUNC()() must return void");
    }
    ~scoped_guard() { f_(); }

    scoped_guard(scoped_guard const&) = delete; // copy禁止
    scoped_guard(scoped_guard&&) = default; // move
    void operator=(scoped_guard const&) = delete; // copy代入禁止
    void operator=(scoped_guard&&) = delete; // move代入禁止

private:
    FUNC f_;
};

namespace {

TEST(TemplateMetaProgrammingA, scoped_guard)
{
    {
        auto demangled = abi::__cxa_demangle(typeid(std::vector<int>).name(), 0, 0, nullptr);
        auto f         = [demangled](){ noexcept { free(demangled); }};
        auto sg        = scoped_guard<decltype(f)>{std::move(f)}; // C++14までの記法

        ASSERT_STREQ("std::vector<int, std::allocator<int>>", demangled);
    }
    {
        auto demangled = abi::__cxa_demangle(typeid(std::vector<int>).name(), 0, 0, nullptr);
        auto gs = scoped_guard{[demangled](){ noexcept { free(demangled); }}}; // C++17からの記法

        ASSERT_STREQ("std::vector<int, std::allocator<int>>", demangled);
    }
    {
        auto stream = fopen("__FILE__", "r");
        auto f      = [stream](){ fclose(stream); };
        auto sg     = scoped_guard<decltype(f)>{std::move(f)}; // C++14までの記法

        char buff[256]{};
        fgets(buff, sizeof(buff) - 1, stream);

        ASSERT_STREQ("scoped_guard.cpp\n", buff);
    }
    {
        auto stream = fopen("__FILE__", "r");
        auto gs = scoped_guard{[stream](){ fclose(stream); }}; // C++17からの記法

        char buff[256]{};
        fgets(buff, sizeof(buff) - 1, stream);

        ASSERT_STREQ("scoped_guard.cpp\n", buff);
    }
}
} // namespace

```

- 演習-関数型のテンプレートパラメータを持つクラステンプレートへ戻る。

参考文献

このドキュメントを書くにあたり参考にした書籍や、さらに学習を進めたい読者に薦める書籍を下記する。出版後10年以上が経過したものも少なくないが、その理由はソフトウェア開発の肝要は急激に変化することがないことと、これら書籍のほとんどが不朽の必読書と言って良いレベルであることによる。

プロセス・プラクティス

- 達人プログラマー
- アジャイルな見積りと計画づくり
- アジャイルプロジェクトマネジメント
- アート・オブ・プロジェクトマネジメント

設計・デザイン

- オブジェクト指向における再利用のためのデザインパターン
- デザインパターンとともに学ぶオブジェクト指向のこころ
- アジャイルソフトウェア開発の奥義
- UMLモデリングのエッセンス
- リファクタリング—プログラムの体質改善テクニック

C++・C

- C++プライマー
- プログラミング言語C++
- cpprefjp - C++日本語リファレンス
- C++リファレンス
- Effective C++
- Effective Modern C++
- C++テンプレートテクニック
- プログラミング言語C
- プログラミング作法
- モダンC言語プログラミング
- テスト駆動開発による組み込みプログラミング

あとがき

すでに述べたように組織のソフトウェア開発能力は、以下の三要素に基づいている。

- ・インフラ
- ・プロセス
- ・プログラマ(人材)

それぞれの観点から、ソフトウェア開発に未熟な組織について考察してみたい。

インフラ

ソフトウェア開発においてインフラとは、

- ・コンパイラ
- ・デバッガ
- ・IDE/エディタ
- ・単体テストフレームワーク

のようなプログラマが直接使用するツールに加えて、

- ・バージョン管理システムやそのウェブサービス
- ・CIサーバー(Jenkinsのようなもの)
- ・静的解析ツール

のようなプログラマを陰で支えるツール等がある。概ねそれらは安価もしくは無料で入手できるため、もしそれらを使っていない(もしくはそれらの性能を十分に引き出していない)のであれば、その理由は予算がないことではない。

多くのソフトウェア開発は、何もないところから始まらない。前リリースのソースコードを起点として、次期開発を行うことがほとんどである。とすれば、開発の起点となるソースコードは上記インフラよりもはるかに重要なインフラであることに気づく。

自明のことであるが、次期ソフトウェアの開発効率はこのソースコードの品質に強く依存する。このことはよく知られているはずなのに、品質の悪いソースコードを捨てることも、改善することもなく次期開発の起点にしてしまう理由は何なのだろうか?

「便利なツールを使っていない(使いこなしていない)」前者にしても、「品質の悪いソースコードを使い続ける」後者にしても、「あるべき姿になっていない」ことは共通している。そして、その言い訳として「工数がない」のである。残念ながらこの分析は間違っている。工数不足は原因ではなく結果である。

プロセス

組み立て工場において、製造工程に工員の高度な技量や知識は不要である。そのため、極端に言えば誰がやってもほぼ同じものを作ることができる。製品の組み立て工程が、高度な技量や熟練を必要としないようにデザインされているからであり、これが可能になる理由は、組み立て行為が単純作業であるためである。

ソフトウェア開発には、複雑で高度な思考、意思決定が必要となるため、誰がやっても同じ成果物を作り出せるようなプロセスのデザインは不可能である。にもかかわらず、ソフトウェア開発に組み立て工場のアナロジーを持ち込み、誰がやっても同じ成果物を作り出せるようなプロセスをデザインしようとする人々がいる。

それとは逆に、ソフトウェア開発は融通無碍であるべきとばかりに、何のルールもなくカウボーイスタイルのプログラミングが横行するチームもある。主張の強いプログラマがこのような思想を持ち、マネージメントが機能しないとこのようなカオス状態のソフトウェア開発が行われる。

前者をプロセス万能主義、後者をプロセス無力主義と呼ぶことにする。万能主義、無力主義どちらの場合も、プロセスに対する無知がその根底にある。この二つは対極的な思想に思えるが、現場の混乱、非効率な開発、バグだらけのソフトウェア等の同様な現象を引き起こす。

万能主義は、さらに大きな問題を引き起こす。効果が疑わしいプロセス(解読困難なソースコードのレビュー等)を押しつけられたプログラマはいずれやってもいいことをやったことにしてしまう。これによって引き起こされたモラルハザードはさらにソフトウェア開発を難しくする。

この両チームより少しだけ気の利いたチームは、万能主義にも無力主義に陥ることなくアジャイル系プロセスを導入することで改善を始める。ただし、ハードルの高いキープラクティス(単体テストや統合テストの自動化、CI等)の導入は無視してしまう。そして数か月後、無視したプラクティス無しではそのプロセスは役に立たないことを証明してしまう。

このようなチームのリーダーやマネージャに「なぜこんなことになったのか？」と問えば、改善する「工数がなかった」と答えるだろう。残念ながらこの分析も間違っている。工数不足は原因ではなく結果である。

プログラマ(人材)

前述したとおり、

- 工数不足が原因となって、不適切なインフラやプロセスを使うという結果になった。

という主張は、ほとんどの場合、因果関係が逆である。

- 不適切なインフラやプロセスを使うことが原因となって、工数不足という結果に陥った。

というのが実状に近い。では、適切なインフラやプロセスを使えない原因は何だろうか？

良いソースコードは良いプログラマにしか書けず、知識豊富なプログラマのみが開発用ツールを使いこなす。プロセスについても同様で、比較的容易に導入できるといわれているSCRUM等の軽量プロセスであっても、高いレベルのプログラマ無しで導入・運用することは不可能である。

つまり、優れたプログラマ無くして適切なインフラ構築やプロセス運用はできないのである。従って、不適切なインフラやプロセスを使い続ける原因是、その組織の人材不足である。

ソフトウェア開発に必要な知識を、単なる経験のみから得ることはほぼ不可能である。実際に10年以上の経験を持ち、且つほとんど素人といってよいプログラマは珍しくない。彼らの問題は、先人の知恵に学ばない(書籍やドキュメントをほとんど読まない)ことである。このため、長い経験の中で行ったトライ&エラーから得られる断片知識だけが彼らのよりどころである。

エディタ、コンパイラといった基礎的なツールですらマニュアルを熟読せずに、その使用方法を習得するこはできない。にもかかわらず、彼らはそれをしない。その代償として、経験だけは長い素人が出来上がる。

ソフトウェア開発の問題の根本原因是これである。従って、この問題に着手せずに、他の問題は解決しない。

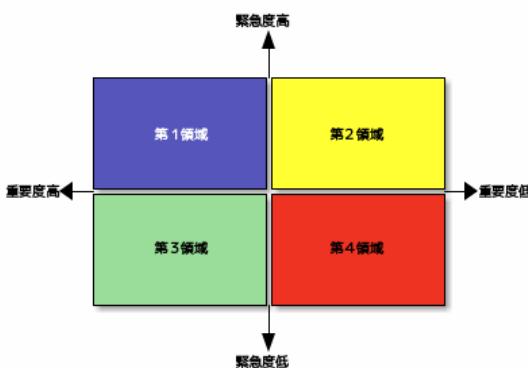
まとめ

ここまで結論は、

- 不足しているのは工数ではなく知識である。
- 知識不足の解決がソフトウェア開発能力改善の必要条件である。

したがって、ソフトウェア開発能力の改善を行いたいのであれば知識の習得(=学習)が不可避である。「学習する工数等ない」との主張が聞こえてきそうだが、本当にその主張は正しいだろうか？

まずは、「7つの習慣」から拝借した下図を見てほしい(「ノコギリの刃を研ぐ」も参照)。



この図は、組織(や個人)が行う仕事を

- 横軸：重要度
- 縦軸：緊急度

の観点から4つに分類するためのものである。

どんな組織だろうが第1領域(緊急かつ重要、定常業務)の仕事から手を付ける。ソフトウェア開発を主な業務とする組織の第1領域の仕事とは、ソフトウェア開発そのものやそれにまつわるソフトウェアのトラブル対応等だろう。未熟な組織においてもこの選択を間違えることはない。

問題はその次の選択である。第2領域(緊急だが、重要でない)の仕事に手を付けてしまうのである(ミーティング、電話、メール等の雑務で、本当に必要なものは全体の何パーセントだろうか?)。

これらに工数を投入してしまうと、第3領域(緊急でないが重要。学習はその代表)に割り当てる工数はなくなる。逆に言えば第2領域よりも第3領域のプライオリティを高くすることのみが、第3領域に対処する方法である。

これは言うよりも簡単でないことは確かだが、これをしない限り第3領域(学習)への時間割り当てができず、したがって組織の知識不足も解決しない。知識不足が解決できないのであれば、工数不足も解決しない(知識不足によって発生した工数不足に見える状態に、増員で対処しようとしてもほとんど効果はない)。

学習が必要であることに賛成でも、組織が今抱えている問題の対策にはあまりにも遅く迂遠に感じる人も多いだろう。そういう人々は、起死回生を図りプロセス改善を始めてしまうこともあるだろうが、すでに述べた通り、このような拙速な対策が大きな効果を發揮することはまずない(例えばソースコードレビューはレビュアのスキルが低ければ効果がない)。さらに悪いことに、この仕事により工数はますます逼迫し、その組織はさらに学習から遠ざかる。

以上の議論をまとめると、ソフトウェア開発能力不足の解決に王道はなく、その唯一の方法は学習なのである。組織改善における格言「着眼大局、着手小局」が教えるところに従い、まずはこのドキュメントをチームで理解することから始めるのが良いのではないかと思う。

本ドキュメントが、多くのプログラマとその組織が学習に向かう一助になることを願う。

Sample Code

vim

vim_config/README.md

```
1 # このリポジトリの目的
2 wsl-ubuntu vim/nvim、cygwin vim、windows gvim/nvim/nvim-qtの設定方法や設定ファイルを保存する。
3
4
5 # このリポジトリのファイル
6 ## README.md
7 このファイル。
8
9 ## ./nvimディレクトリ
10
11 |ファイル名      |内容
12 |:-----|:-----|
13 |nvim/init.vim  |nvimから最初に読み込まれる
14 |nvim/org.vim   |オリジナルの設定
15 |nvim/package.vim|deinを使用した外部パッケージの設定|
16 |nvim/plugin/    |自作プラグイン
17 |nvim/autoload/  |自作プラグインのオートロード
18 |nvim/rplugin/   |自作プラグイン
19 |nvim/ftplugin/  |filetype別プラグイン
20 |nvim/cheatsheet.md|自分用ヘルプ。:Cheatで表示
21
22 ## ./vimディレクトリ
23 |ファイル名|内容
24 |:-----|:-----|
25 |vim/vimrc  |~/.vimrcにコピーかシンボリックリンクを張って使用|
26 |vim/gvimrc |~/.gvimrcにコピーかシンボリックリンクを張って使用|
27
28 ## その他
29 |ファイル名|内容
30 |:-----|:-----|
31 |vim.sh     |bashrc等からsource
32 |inputrc   |readline(gdb)を使用するアプリケーションのrcファイル。~/.inputrcにコピーして使用する。|
33
34
35 # 環境変数
36 ここで説明する環境変数は下記のvim系プログラムを初期設定をするためのものである。
37 * ubuntu上のvim、nvim
38 * cygwin上のvim
39 * windows上のgvim、nvim、nvim-qt
40 * cygwinターミナルから起動されたwindows上のgvim、nvim、nvim-qt
41
42 下記の環境変数XDG_CONFIG_HOMEはすべて同一の実態(ディレクトリ)を指している。
43
44 環境変数VIMやVIMRUNTIMEは設定しない。設定しない場合、
45 * windows版vimでは、
46   * VIMは、'$HOME/.vim'がセットされる。
47   * VIMRUNTIMEは、'$HOME/.vim/vimNN'がセットされる。
48
49 * windows版nvim-qでは、Neovimにバンドルされているパッケージが入っているディレクトリが
50   VIMやVIMRUNTIMEにセットされる。
51
52 ## wsl-ubuntu
53 * .bashrcに下記を追加する。
54
55     export XDG_CONFIG_HOME=~/config
56     source $XDG_CONFIG_HOME/vim.sh
57
58 ## cygwin
59 * .bashrcに下記を追加する。
60
61     # LINUX_HOMEはubuntuのホームディレクトリを指定
62     export LINUX_HOME=/cygdrive/c/XXX
63     export XDG_CONFIG_HOME=$LINUX_HOME/.config
64     source $XDG_CONFIG_HOME/vim.sh
```

```

65
66 ## windows
67 * 環境変数に以下を追加する
68
69     # XXXは上記と同じものを指定
70     # YYYはcygwinのホームディレクトリを指定
71     XDG_CONFIG_HOME C:\XXX\.config
72     HOME C:\cygwin64\home\YYY
73
74
75 # インストール
76 1. 上記「環境変数」に従って環境変数、bashrcを設定
77 2. このリポジトリを'$LINUX_HOME/.config'にgit clone
78 3. https://github.com/shougo/dein.vim
79 に従って dein を'$LINUX_HOME/.config/nvim_pkg/dein' にインストール
80 4. 外部パッケージのインストールのために、wsl-ubuntu nvim立ち上げて:call dein#install()を実行
81 5. cygwin vimの設定として、'$LINUX_HOME/.config/vim/vimrc'をcygwinのホームディレクトリの
82     .vimrcにコピーするかシンボリックリンクを張る。
83 6. windows gvimの設定として、'$LINUX_HOME/.config/vim/gvimrc'をcygwinのホームディレクトリの
84     .gvimrcにコピーするかシンボリックリンクを張る。
85
86
87 # 現在の問題点
88 * パス形式の違いでwindows nvim/nvim-qtから動作しないプラグインがある。
89 * cygwin vimはpython3対応でコンパイルされていないため、pythonが必要なプラグインは動作しない。
90 * deinから設定したvim-fugitiveはcygwin vimでは動作しないため、
91   https://github.com/tptope/vim-fugitive
92   に書いてある設定が必要である。
93
94
95 # スクリプト読み込みのデバッグ
96 |コマンド          |意味
97 |:-----|:-----|
98 |:echo $MYVIMRC      |最初に読み込まれる設定ファイルの表示
99 |:scriptnames        |現在読み込まれているスクリプトファイル一覧|
100 |:let @a=execute('scriptnames')|scriptnamesの出力をレジスタaに取り込む
101 |:checkhealth        |足りないパッケージや設定等のチェック

```

vim_config/inputrc

```

1 set editing-mode vi
2 set completion-ignore-case on
3 set show-all-if-unesaved on

```

vim_config/nvim/autoload/buffers.vim

```

1 let s:buffers_buffer = 'Buffers'
2
3 function! s:line(num, buff_name)
4     if a:buff_name == s:buffers_buffer
5         return printf("%3d [%s]\n", a:num, a:buff_name)
6     else
7         return printf("%3d %s\n", a:num, a:buff_name)
8     endif
9 endfunction
10
11 function! s:load_buffers_list()
12     let l:line = line('.')
13     let l:col = col('.')
14
15     let buffs = split(execute('buffers'), "\n")
16
17     setlocal modifiable
18
19     silent execute ':%delete'
20     for b in buffs
21         let new_b = substitute(b, '^ *\\(\\d+\\).*(\\.*\\)\\\".*', { m -> s:line(m[1], m[2])}, 'g')
22         silent execute ':normal i' . new_b
23     endfor
24     silent execute '$:delete'
25
26     setlocal modifiable
27     call cursor(l:line, l:col)
28 endfunction
29
30 function! buffers#close_buff(force)

```

```

31     let word = expand("<cword>")
32
33     if word =~ '\d\+'
34         let to_delete = bufname(str2nr(word))
35     elseif word =~ '.\+'
36         let to_delete = word
37     else
38         echo 'no file'
39         return
40     endif
41
42     if to_delete == s:buffers_buffer
43         execute 'bdelete!' . word
44     else
45         let bwipeout = 'bwipeout' . (a:force == 0 ? ' ' : '! ')
46         execute bwipeout . word
47         call s:load_buffers_list()
48     endif
49 endfunction
50
51 function! buffers#open_buff()
52     let word = expand("<cword>")
53
54     if word =~ '\d\+'
55         execute 'buffer ' . word
56     elseif word =~ '.\+'
57         execute 'buffer ' . word
58     else
59         echo 'no file'
60     endif
61 endfunction
62
63 function! s:key_map() abort
64     nnoremap <silent> <buffer> <Plug>(buffers-delete) :call buffers#close_buff(0)<CR>
65     nmap <buffer> d <Plug>(buffers-delete)
66
67     nnoremap <silent> <buffer> <Plug>(buffers-delete-force) :call buffers#close_buff(1)<CR>
68     nmap <buffer> D <Plug>(buffers-delete-force)
69
70     nnoremap <silent> <buffer> <Plug>(buffers-open) :call buffers#open_buff()<CR>
71     nmap <buffer> o <Plug>(buffers-open)
72
73     nnoremap <silent> <buffer> <Plug>(buffers-reload) :call buffers#begin()<CR>
74     nmap <buffer> r <Plug>(buffers-reload)
75 endfunction
76
77 function! buffers#begin()
78     let winid = bufwinid(s:buffers_buffer)
79     if winid != -1
80         call win_gotoid(winid)
81     else
82         execute 'new' s:buffers_buffer
83         set buftype=nofile
84         set noswapfile
85         call s:key_map()
86     endif
87     call s:load_buffers_list()
88 endfunction

```

vim_config/nvim/autoload/cd.vim

```

1 function! s:is_term()
2     let bn = bufname("%")
3     if bn =~ 'term://'
4         return 1
5     else
6         return 0
7     endif
8 endfunction
9
10 function! cd#find_dir_candidate()
11     for i in range(50)
12         silent execute 'normal "uyy'
13         let reg=@u
14         if reg =~ 'ichiro@[^ ]\+ \~.*'
15             return substitute(reg, 'ichiro@[^ ]\+', '', '')
16         endif
17         silent execute 'normal k'
18     endfor
19

```

```

20     return ''
21 endfunction
22
23
24 function! cd#change_dir()
25     if s:is_term() == 1
26         let dir_candidate=cd#find_dir_candidate()
27
28     if dir_candidate == ''
29         echo "no dir candidate"
30     else
31         silent execute ':cd ' . dir_candidate
32     endif
33 else
34     silent execute ':cd %:h'
35 endif
36 endfunction

```

vim_config/nvim/autoload/git_diff.vim

```

1 let s:git_diff_buffer = 'GIT DIFF'
2 let s:git_diff_buffer_cur = ''
3 let s:git_diff_buffer_new = ''    ".newと表示されるが実際には改定前ファイル
4
5 set patchexpr=git_diff#patch()
6
7 function! git_diff#patch() abort
8     silent execute '!patch -R -o ' . v:fname_out . ' ' . v:fname_in . ' < ' . v:fname_diff
9 endfunction
10
11 function! s:echo_err(msg) abort
12     echohl ErrorMsg
13     echomsg 'git_diff.vim:' a:msg
14     echohl None
15 endfunction
16
17 " pattern : 0      just changed
18 "           : 1      change filename
19 function! s:modified_pattern(pattern) abort
20     if a:pattern == 0
21         return '^[[ADMU? ][DMU? ] \+'
22     elseif a:pattern == 1
23         return 'R[M ] \+.* -> '
24     endif
25     call s:echo_err('modified_pattern wrong')
26 endfunction
27
28 function! s:get_target_name() abort
29     let l:line=getline(".")
30
31     if match(l:line, s:modified_pattern(0)) isnot# -1
32         return substitute(l:line, s:modified_pattern(0), "", "")
33     elseif match(l:line, s:modified_pattern(1)) isnot# -1
34         return substitute(l:line, s:modified_pattern(1), "", "")
35     else
36         call s:echo_err(l:line . ' :this file is not modefied')
37         return ''
38     endif
39 endfunction
40
41 function! s:git_diff(patch_target, patch_file_name, from_head) abort
42     if a:from_head == 0
43         silent execute '!git diff ' . a:patch_target . ' > ' . a:patch_file_name
44     else
45         silent execute '!git diff HEAD ' . a:patch_target . ' > ' . a:patch_file_name
46     endif
47 endfunction
48
49 function! s:each(line1, line2, func_ref) abort
50     let l:col = col('.')
51
52     for i in range(a:line1, a:line2)
53         call cursor(i, l:col)
54         call a:func_ref()
55     endfor
56 endfunction
57
58 function! s:launch_file() abort
59     let l:target=s:get_target_name()
60

```

```

61      if g:os == 'linux'
62          silent execute '!wslstart' . l:target
63      else
64          if g:os == 'cygwin'
65              silent execute '!wslstart' . l:target
66          else
67              echo 'not support launch'
68          endif
69      endif
70 endfunction
71
72 function! git_diff#launch_file(line1, line2) abort
73     call s:each(a:line1, a:line2, function('s:launch_file'))
74 endfunction
75
76
77 function! s:git_add() abort
78     let l:target=s:get_target_name()
79
80     silent execute '!git add ' . l:target
81     call git_diff#make_list_load()
82 endfunction
83
84 function! git_diff#git_add(line1, line2) abort
85     call s:each(a:line1, a:line2, function('s:git_add'))
86 endfunction
87
88 function! s:git_reset() abort
89     let l:target=s:get_target_name()
90
91     if len(l:target) == 0
92         return
93     endif
94
95     silent execute '!git reset ' . l:target
96     call git_diff#make_list_load()
97 endfunction
98
99 function! git_diff#git_reset(line1, line2) abort
100    call s:each(a:line1, a:line2, function('s:git_reset'))
101 endfunction
102
103 function! git_diff#make_list_load() abort
104     setlocal modifiable
105
106     let l:line = line('.')
107     let l:col = col('.')
108     silent execute ':%delete'
109     silent execute 'r! git status -s'
110     silent execute ':idelete'
111     call cursor(l:line, l:col)
112
113     setlocal nomodifiable
114 endfunction
115
116 function! s:key_map_end() abort
117     delcommand GitAdd
118     delcommand GitReset
119 endfunction
120
121 function! s:key_map_begin() abort
122     command! -range GitAdd :call git_diff#git_add(<line1>, <line2>)
123     map <silent> <buffer> a :GitAdd<CR>
124
125     command! -range LaunchFile :call git_diff#launch_file(<line1>, <line2>)
126     map <silent> <buffer> L :LaunchFile<CR>
127
128     command! -range GitReset :call git_diff#git_reset(<line1>, <line2>)
129     map <silent> <buffer> r :GitReset<CR>
130
131     nnoremap <silent> <buffer> <Plug>(diff-open) :<C-u>call git_diff#show_diff(0, 0)<CR>
132     nmap <buffer> d <Plug>(diff-open)
133
134     nnoremap <silent> <buffer> <Plug>(diff-open-V) :<C-u>call git_diff#show_diff(1, 0)<CR>
135     nmap <buffer> vd <Plug>(diff-open-V)
136
137     nnoremap <silent> <buffer> <Plug>(diff-open-h) :<C-u>call git_diff#show_diff(0, 1)<CR>
138     nmap <buffer> h <Plug>(diff-open-h)
139
140     nnoremap <silent> <buffer> <Plug>(diff-open-Vh) :<C-u>call git_diff#show_diff(1, 1)<CR>

```

```

141 nmap <buffer> vh <Plug>(diff-open-Vh)
142
143 nnoremap <silent> <buffer> <Plug>(diff-list-close) :<C-u>call git_diff#end()<CR>
144 nmap <buffer> <C-Q> <Plug>(diff-list-close)
145
146 nnoremap <silent> <buffer> <Plug>(diff-reload) :<C-u>call git_diff#make_list_load()<CR>
147 nmap <buffer> s <Plug>(diff-reload)
148
149 nnoremap <silent> <buffer> <Plug>(git-commit) :!git commit -m
150 nmap <buffer> c <Plug>(git-commit)
151 endfunction
152
153 "vertical      : 0 normal split to show diff buffers
154 "              : 1 virtual split to show diff buffers
155 "from_head     : 0 git diff
156 "              : 1 git diff HEAD
157 function! git_diff#show_diff(vertical, from_head) abort
158   call git_diff#show_diff_off()
159
160   let l:target=s:get_target_name()
161
162   if gettimeofday(l:target) == -1
163     s:echo_err('file not exists')
164   endif
165
166   let s:git_diff_buffer_cur = l:target
167   let s:git_diff_buffer_new = l:target . '.new'
168
169   let l:temp_file = tempname()
170   call s:git_diff(l:target, l:temp_file, a:from_head)
171
172   execute ':sp'
173   execute ':e ' . l:target
174
175   if a:vertical == 0
176     execute ':diffpatch ' . l:temp_file
177
178     " 上バッファがオリジナル。下バッファが変更後ファイル
179     execute "normal \<C-w>j"
180   else
181     execute ':vert diffpatch ' . l:temp_file
182
183     " 左バッファがオリジナル。右バッファが変更後ファイル
184     execute "normal \<C-w>l"
185   endif
186
187   call delete(l:temp_file)
188 endfunction
189
190 function! git_diff#show_diff_off()
191   execute ':diffoff!'
192
193   if s:git_diff_buffer_cur != ''
194     let winid = bufwinid(s:git_diff_buffer_cur)
195
196     if winid isnot# -1
197       call win_gotoid(winid)
198       :quit
199     endif
200
201     let s:git_diff_buffer_cur = ''
202   endif
203
204   if s:git_diff_buffer_new != ''
205     let winid = bufwinid(s:git_diff_buffer_new)
206
207     if winid isnot# -1
208       call win_gotoid(winid)
209       execute 'bwipeout! ' . s:git_diff_buffer_new
210     endif
211
212     let s:git_diff_buffer_new = ''
213   endif
214 endfunction
215
216 function! git_diff#begin(resize) abort
217   let winid = bufwinid(s:git_diff_buffer)
218   if winid isnot# -1
219     call win_gotoid(winid)
220   else

```

```

221     execute 'edit' s:git_diff_buffer
222     set buftype=nofile
223     set noswapfile
224     setlocal nomodifiable
225
226     call s:key_map_begin()
227     call git_diff#make_list_load()
228   endif
229
230   if a:resize != 0
231     execute ":only"
232     let l:lines=&lines
233     echo l:lines
234     if l:lines < 50
235       execute ":set lines=50"
236     endif
237     execute ":set columns=201"
238   endif
239
240 endfunction
241
242 function! git_diff#end()
243   call git_diff#show_diff_off()
244   call s:key_map_end()
245   bwipeout!
246 endfunction
247

```

vim_config/nvim/autoload/git_session.vim

```

1 function! s:get_git_top()
2   let top = system('git rev-parse --show-toplevel')
3   return substitute(top, "\n", '/', 'g')
4 endfunction
5
6 let s:git_session_top = s:get_git_top()
7 let s:git_session_file = s:git_session_top . 'Session.vim'
8
9 function! s:get_dirs() abort
10   " ディレクトリのみを抽出
11   let dir_filter = '|sed -e /^[^/]\+\$/d -e s@/[^\/]\+\$@\@g | sort | uniq'
12   let dirs_str = system("git ls-files --full-name " . s:git_session_top . dir_filter)
13
14   let dirs_list = split(dirs_str, "\n")
15
16   let full_dirs = map(dirs_list, 's:git_session_top . v:val')
17
18   call add(full_dirs, s:git_session_top)
19
20   return full_dirs
21 endfunction
22
23 function! git_session#set_path()
24   let dirs = s:get_dirs()
25   let &path='./,/usr/include/c++/9/' . join(dirs, ',')
26
27   let tag_dirs = map(dirs, 'v:val . "tags"')
28   let &tags ='tags,.tags,' . join(tag_dirs, ',')
29 endfunction
30
31 function! git_session#begin() abort
32   let top = s:get_git_top()
33
34   if top != s:git_session_top
35     echo 'must be on the dir under ' . s:git_session_top
36     return
37   endif
38
39   try
40     execute 'source ' . s:git_session_file
41   catch
42     echo 'no session file'
43   endtry
44   call git_session#set_path()
45 endfunction
46
47 function! git_session#dir() abort
48   echo s:git_session_top
49 endfunction
50

```

```

51 function! git_session#make() abort
52     let top = s:get_git_top()
53
54     if top != s:git_session_top
55         echo 'must be on the dir under ' . s:git_session_top
56         return
57     endif
58
59     execute 'mksession! ' . s:git_session_file
60 endfunction
61
62 function! git_session#new_session() abort
63     let s:git_session_top = s:get_git_top()
64     let s:git_session_file = s:git_session_top . 'Session.vim'
65     call git_session#begin()
66 endfunction

```

vim_config/nvim/autoload/grep.vim

```

1 function s:grep(target, args_for_dir)
2     let ignore_case=&ic
3     if l:ignore_case == 0
4         let ic_str = ''
5     else
6         let ic_str = '-i '
7     end
8     execute ':grep -R ' . l:ic_str . a:args_for_dir . ' "' . a:target . '"'
9 endfunction
10
11 function grep#grep(...)
12     let args_for_dir = s:make_args_for_dir(a:000)
13     let l:target=@/
14     call s:grep(l:target, args_for_dir)
15 endfunction
16
17 function grep#grepp(pattern, ...)
18     let args_for_dir = s:make_args_for_dir(a:000)
19     call s:grep(a:pattern, args_for_dir)
20 endfunction
21
22 function s:make_include(array)
23     let beg='--include="*.'
24     let end=''
25     let inc = map(a:array, 'l:beg . v:val . l:end')
26     return join(inc)
27 endfunction
28
29 function! s:make_args_for_dir(args)
30     let res = []
31     call extend(res, a:args)
32
33     if len(res) == 0
34         return s:make_include(['[chCH]', 'cpp', 'sh', 'rb', 'py', 'vim', 'md', 'pu'])
35     else
36         if len(res) == 1 && res[0] == '-'
37             return '--exclude-dir=".git"'
38         else
39             return s:make_include(res)
40         endif
41     end
42 endfunction
43

```

vim_config/nvim/autoload/multi_hl.vim

```

1 function! multi_hl#add_word(word, row)
2
3     let curr = @/
4     if a:row
5         let new_word = a:word
6     else
7         let new_word = '\<' . a:word . '\>'
8     endif
9
10    if len(curr) != 0
11        let search = curr . '\|' . new_word
12    else
13        let search = new_word
14    endif

```

```

15
16     let @/ = search
17
18 endfunction
19
20 function! multi_hl#clear()
21     let @/ = ''
22 endfunction

```

vim_config/nvim/autoload/next_file.vim

```

1 " カレントファイルがxxx_yyy_zzz.cppだったとすると、
2 "   del_num      :0      エクステンションを取ったxxx_yyy_zzzに*を付けてglob
3 "                 :5      xxx_yyy_z*をglob
4
5 function! s:make_stem(curr_file, del_num) abort
6     if a:del_num == 0
7         let stem = expand('%:r')
8     else
9         let name_len = len(a:curr_file)
10
11        if name_len > a:del_num
12            let name_len -= a:del_num
13        else
14            let name_len = 1
15        endif
16
17        let stem = expand('%')[0 : name_len]
18    endif
19
20 " ファイル名がxxx_ut.(cpp|h)だった場合の調整
21 return substitute(stem, "_ut$", "", "")
22 endfunction
23
24 function! s:get_next_file(curr_file, stem) abort
25     let candidates = glob(a:stem . "*", 1, 1)
26     let candi_num = len(candidates)
27
28     if candi_num < 2
29         return ''
30     endif
31
32     for i in range(candi_num)
33         if a:curr_file == candidates[i]
34             let i = i + 1
35             if i < candi_num
36                 return candidates[i]
37             else
38                 return candidates[0]
39             endif
40         endif
41     endfor
42
43     echomsg 'mybe bug found in next_file.vim:'
44 endfunction
45
46 let s:last_file = ''
47 let s:last_del_num = 0
48
49 function! next_file#change(del_num) abort
50     let curr_file = expand('%')
51
52     if s:last_file == curr_file && a:del_num == 0
53         let del_num = s:last_del_num
54     else
55         let del_num = a:del_num
56     endif
57
58     let stem = s:make_stem(curr_file, del_num)
59     echo stem . '*' . del_num
60
61     let next_file = s:get_next_file(curr_file, stem)
62     if next_file == ''
63         let s:last_file = curr_file
64         let s:last_del_num = del_num < 6 ? 6 : del_num + 1
65     else
66         let s:last_file = next_file
67         let s:last_del_num = del_num
68         execute 'edit' next_file

```

```
69      endif
70 endfunction
71
```

vim_config/nvim/autoload/term.vim

```
1 let s:term_num = -1
2
3 function! s:start_term() abort
4 "    silent execute 'new'
5     silent execute 'terminal'
6
7     let s:term_num = bufnr()
8 endfunction
9
10 function! term#start() abort
11     if s:term_num == -1
12         call s:start_term()
13     else
14         try
15             execute "buffer " . s:term_num
16         catch
17             " bufferが閉じられていた
18             call s:start_term()
19         endtry
20     endif
21 endfunction
22
23
```

vim_config/nvim/autoload/termdbg.vim

```
1 ":Run [args]      [args] または以前の引数でプログラムを実行する
2 ":Arguments      {args}  次の :Run のために引数を設定する
3 ":Break          カーソル位置にブレークポイントを設定する。
4 ":Break          {position}
5 "               指定位置にブレークポイントを設定する。
6 ":Clear          カーソル位置のブレークポイントを削除する
7 ":Step           gdb の "step" コマンドを実行する
8 ":Over           gdb の "next" コマンドを実行する
9 "               (:Next だと Vim のコマンドとかぶるので)
10 ":Finish         gdb の "finish" コマンドを実行する
11 ":Continue       gdb の "continue" コマンドを実行する
12 ":Stop           プログラムを中断する
13
14
15 " todo
16 "   b   をトグルにしたい
17 "
18 function! termdbg#key_map() abort
19     nnoremap <silent> <buffer> <Plug>(gdb-run) :Run<CR>
20     nmap <buffer> R <Plug>(gdb-run)
21
22     nnoremap <silent> <buffer> <Plug>(gdb-break) :Break<CR>
23     nmap <buffer> B <Plug>(gdb-break)
24
25     nnoremap <silent> <buffer> <Plug>(gdb-continue) :Continue<CR>
26     nmap <buffer> C <Plug>(gdb-continue)
27
28     nnoremap <silent> <buffer> <Plug>(gdb-break-clear) :Clear<CR>
29     nmap <buffer> D <Plug>(gdb-break-clear)
30
31     nnoremap <silent> <buffer> <Plug>(gdb-finish) :Finish<CR>
32     nmap <buffer> F <Plug>(gdb-finish)
33
34     nnoremap <silent> <buffer> <Plug>(gdb-next) :Over<CR>
35     nmap <buffer> N <Plug>(gdb-next)
36
37     nnoremap <silent> <buffer> <Plug>(gdb-step) :Step<CR>
38     nmap <buffer> S <Plug>(gdb-step)
39
40 endfunction
41
42 function! termdbg#start(program)
43     silent execute ':packadd termdebug'
44     silent execute ':Termdebug ' . a:program
```

```

45 endfunction
46

vim_config/nvim/cheatsheet.md

1 # My CheatSheet
2
3 ## dein
4 * https://github.com/Shougo/dein.vimからdeinとダウンロードしてセットアップ
5   * curl ... > installer.sh # インストーラのダウンロード
6   * sh ./installer.sh ~/$XDG_CONFIG_HOME/nvim_pkg/dein # deinのダウンロード
7 * :call dein#install() # 他のパッケージインストール
8
9 ## terminal(windows app)操作
10 * Alt-space -> x 最大化
11 * Alt-space -> r 元のサイズに戻す
12 * Alt-space -> n 最小化
13 * Alt-space -> m ウィンドウの移動
14 * Alt-space -> s ウィンドウのサイズ変更
15 * window-m 全アプリの最小化
16 * window-d デスクトップの表示/全アプリの表示
17
18
19 ## 正規表現
20 [正規表現アトム](#pattern-atoms)
21
22 \_. 改行含むすべての文字にマッチ
23
24 ## NextFile
25 <M-q> 現在のファイル名を@qに代入
26 <Num><C-q> 現在のファイル名を<Num>文字消して、"その文字列.\*"とマッチしたファイルを開く
27 Numが省略された場合<Num>は0として処理する
28 Numが0であった場合、"サフィックスを削除した文字列.\*"とマッチしたファイルを開く
29
30 ## MultiHl
31 :MultiHlAdd 現在のカーソルの下のwordを'\\<word\>'にして、@/に追加。
32 :MultiHlAddR 現在のカーソルの下のwordをそのまま@/に追加。
33 :MultiHlAddI <WORD> WORDをそのまま@/に追加。
34 :MultiHlClear @/に""を入力。
35
36 ## git diff
37 :GitDiff : gitリポジトリのdiffリスト
38 :GitDiffResize : 画面をリサイズして、gitリポジトリのdiffリスト
39 :GitDiffOff : \*.newをbwipeoutしてdiffモード終了
40 {Visual}a : git add from '< to \'>
41 c" : git commit -m "までコマンドラインに入力
42 d : 水平分割diff表示
43 vd : 垂直分割diff表示
44 h : HEADとの水平分割diff表示
45 vh : HEADとの垂直分割diff表示
46 <C-Q> : git diffモード終了
47 {Visual}L : cygstart/wslstart from '< to \'>
48 {Visual}r : git reset from '< to \'>
49 s : diffリストの再ロード
50
51 ## grep
52 :Grep [suffix ...] : grep -R --include=\*.suffix ... @/
53 :GrepP pattern [suffix ...] : grep -R --include=\*.suffix ... pattern
54
55 suffixを省略した場合、cpp c h rb py vimがsuffixになる。
56 全体をgrepしたい場合は - を指定する。
57 :Grep、:Grepp実行後はquick fixウインドが開いて結果を見ることができる。
58 <C-g><C-n>で次へ移動、<C-g><C-p>で前に移動できる。
59
60 :set ignorecaseが行われていた場合には、grepのオプションに-iが追加される。
61 :Grepが行われた場合に実際に起動されるgrepコマンドは、:grepである。
62 :grepが行われた場合に実際に起動されるgrepコマンドは、:set grepprg=...で設定されている。
63
64
65 ## :terminalの使い方
66 * <C-q> vimモード
67 * a, i等の通常のinsertモード移行で terminalモード
68
69 でソースコード全体からの補完ができるらしいので調べる。
70

```

```

71 ## termdebug
72 * nvim-gdbはやめて、こちらにした。
73
74 :DbgStart <prog>      termdebugをロードして、gdb <prog>
75 :DbgKey                 下記mapをバッファローカルでmap
76
77 オリジナルコマンド map
78 :Run [args]          R  [args] または以前の引数でプログラムを実行する
79 :Arguments           {args} 次の :Run のために引数を設定する
80 :Break               B  カーソル位置にブレークポイントを設定する。
81 :Clear               D  カーソル位置のブレークポイントを削除する
82 :Step                S  gdb の "step" コマンドを実行する
83 :Over                N  gdb の "next" コマンドを実行する
84 :Finish              F  gdb の "finish" コマンドを実行する
85 :Continue            C  gdb の "continue" コマンドを実行する
86 :Stop                プログラムを中断する
87
88 ## Session
89 * gitプロジェクトの全ディレクトリを使用してpathやtabsをセットする。
90 * mksessionでgitリポジトリのトップにSession.vimを作る。
91
92 :ProjectBegin       Session.vimをsourceして、path、tabsをセットする。
93 :ProjectDir         現在のセッションが使用しているgitリポジトリの
94                   トップディレクトリを表示。
95 :ProjectMake        Session.vimを作る。
96 :ProjectNew         現在のセッションをカレントディレクトリを含む
97                   gitリポジトリにする。
98
99 ## PathSet
100 * pathにカレントディレクトリ以下のディレクトリをセットする。
101
102 :PathAdd            pathにディレクトリを追加する
103 :PathClear          pathを./のみにする
104 :PathSet            PathClearしてからPathAddする
105
106 ## Buff
107 軽いのが取り柄のバッファエクスプローラ。
108
109 :Buff                :buffersの情報からBuffers問う名前のバッファを開く。
110 o                  プロンプトの下のバッファを開く。
111 d                  プロンプトの下のバッファをスワイプアウトする。
112 D                  プロンプトの下のバッファを強制スワイプアウトする。
113
114 ## チップス
115 * 現在のrcファイルの確認
116     :echo $MYGVIMRC
117
118 * マップの定義位置
119     :verbose map <C-Q>
120
121 * vimコマンドの出力の取り込み
122     :let @a=execute('scriptnames')  "scriptnamesの出力をレジスタaに入れる。
123
124 * globalで行に移動し、そこでコマンド実行
125     :global/^</normal AHEHE    "先頭が"<"である行の末尾に"HEHE"を追加する。
126
127 * コマンドの繰り返し数の注意
128     :map <C-A> 3w
129 とした場合、
130     2<C-A>
131 は
132     23w
133 となるため、このコマンドは23ワードの移動になるが、それはおそらく意図したものではない。
134 <Num><C-A>を<C-A>の<Num>回の繰り返しにしたい場合、式レジスタを使い以下のように書く。
135     :map <C-A> @"'3w'<CR>
136
137 ### todo
138 * 言語サポート
139     * gdbのpでSTLコンテナのきれいな表示がしたい。
140     * rtag.vim
141     * Tagbar: a class outline viewer for Vim
142
143 * そのうち調べる
144     * コメントフォーマット。'formatoptions' の設定
145     * 'showcmd'、'backspace'

```

```
146     * 関数escape(@", '\\\\/')
147     * packadd! matchit 「と」のマッチング
148     * colorscheme evening
149     * :mksession、:wviminfo、:rviminfo
150     * terminal 端末通信 call/drop
```

vim_config/nvim/ftplugin/c.vim

```
1 setlocal tabstop=4
2 setlocal shiftwidth=4
3 setlocal expandtab
4 setlocal tags=tags,./tags
5
6 " paren and etc matching
7 set matchpairs+=<:>
8 hi MatchParen ctermfg=0 cterm=bold,reverse
9
```

vim_config/nvim/ftplugin/python.vim

```
1 setlocal tabstop=4
2 setlocal shiftwidth=4
3 setlocal expandtab
4 setlocal tags=tags,./tags
5
```

vim_config/nvim/ftplugin/ruby.vim

```
1 setlocal tabstop=2
2 setlocal shiftwidth=2
3 setlocal expandtab
4 setlocal tags=tags,./tags
```

vim_config/nvim/init.vim

```
1 if exists('s:loaded')
2     finish
3 else
4     let s:loaded = 1
5 endif
6
7 function! s:get_os()
8     if isdirectory('c:/')
9         if isdirectory('/usr')
10            return 'cygwin'
11        else
12            return 'windows'
13        endif
14    endif
15
16    let old=&ignorecase
17    let uname=system('uname')
18
19    let &ignorecase = 1
20
21    if uname =~ ".*linux.*"
22        let ret = 'linux'
23    else
24        let ret = 'unknown'
25    endif
26
27    let &ignorecase = old
28    return ret
29 endfunction
30
31 let g:os=s:get_os()
32
33 let s:dir=expand('<sfile>:p:h')
34
35 execute 'source ' . s:dir . '/' . 'org.vim'
36 execute 'source ' . s:dir . '/' . 'package.vim'
37
38 if !has('nvim')
39     execute 'set runtimepath+=' . s:dir
40     if g:os == 'linux'
41         set runtimepath+=/usr/share/vim/
42     elseif g:os == 'cygwin'
```

```
43      set runtimepath+=/usr/share/vim/vim82
44  elseif g:os == 'windows'
45
46  endif
47 endif
```

vim_config/nvim/org.vim

```
1 " misc
2 set hidden
3 set ruler
4 set hlsearch
5 set nowrapscan
6 set laststatus=2
7 set noequalalways
8 set backspace=indent,eol,start
9 set incsearch
10 set background=dark
11 set statusline=%<%f[%n]%h%m%r%=%l,%c%V\ %P
12 set fileformats=unix,dos,mac
13 set printoptions=number:y
14
15 set virtualedit=block
16
17 " Makefileやbashスクリプトでgfを効かせるため
18 set isfname-=:
19 set isfname-=,
20 set isfname-=\=
21
22 " encoding
23 "set encoding=utf-8
24 set fileencoding=utf-8
25 set fileencodings=iso-2022-jp,euc-jp,sjis,utf-8
26
27 "
28 syntax on
29
30 " to use quick fix
31 "make setting
32 set makeprg=/usr/bin/make
33 set errorformat+=In\ file\ included\ from\ %f:%l:%m      "gcc
34 set errorformat+=%f:%l:%m          "gcc
35 set errorformat+=%f\|%l\|%m          "grep from vim
36 set errorformat+=%t\\,%f\\,%l\\,%c\\,%m
37
38 "grep setting
39 set grepprg=/usr/bin/grep\ -nH
40
41 let &makeef = expand('<sfile>:p:h') . '/' . getpid()
42
43 " default text
44 set tabstop=4
45 set shiftwidth=4
46 set expandtab
47 set nowrap
48 set cindent
49 "set iskeyword+=-
```

vim_config/nvim/package.vim

```
1 if &compatible
2   set nocompatible " Be improved
3 endif
4
5 " このファイルのあるディレクトリ/../nvim_pkg/deinにパッケージを入れる。
6 let g:pkg_dir=simplify(expand('<sfile>:p:h') . '/../nvim_pkg/dein')
7
8 " Required:
9 " Add the dein installation directory into runtimepath
10 execute 'set runtimepath+=' . g:pkg_dir . '/repos/github.com/Shougo/dein.vim'
11 execute 'set runtimepath+=' . g:pkg_dir . '/repos/github.com/'
12
13 " Required:
14 call dein#begin(g:pkg_dir)
15
16 " Let dein manage dein
17 " Required:
18 call dein#add(g:pkg_dir . '/repos/github.com/Shougo/dein.vim')
19 if !has('nvim')
```

```

20     call dein#add('roxma/nvim-yarp')
21     call dein#add('roxma/vim-hug-neovim-rpc')
22 endif
23
24 call dein#add('Shougo/deoplete.nvim')
25 call dein#add('zchee/deoplete-clang')
26
27 " Add or remove your plugins here like this:
28 call dein#add('Shougo/neosnippet.vim')
29 call dein#add('Shougo/neosnippet-snippets')
30 call dein#add('Shougo/neoinclude.vim')
31
32 call dein#add('tpope/vim-fugitive')
33 call dein#add('reireias/vim-cheatsheet')
34 call dein#add('mattn/vim-maketable')
35 call dein#add('aklt/plantuml-syntax')
36
37 " Required:
38 call dein#end()
39
40 " Required:
41 filetype plugin indent on
42 syntax enable
43
44 let g:cheatsheet#cheat_file = expand('<filename>:p:h') . '/cheatsheet.md'
45 let g:table_mode_corner = '|'
46
47 if g:os == 'windows'
48     let g:python_host_prog = 'C:\cygwin64\bin\python2.7.exe'
49     let g:python3_host_prog = 'C:\cygwin64\bin\python3.8.exe'
50 elseif g:os == 'cygwin'
51
52 elseif g:os == 'linux'
53     let g:ruby_host_prog = '/usr/local/bin/neovim-ruby-host'
54     let g:python_host_prog = '/usr/bin/python2'
55     let g:python3_host_prog = '/usr/bin/python3'
56 else
57     echo 'unkown os'
58 endif
59
60 " deoplete
61 if has('python3')
62     let g:deoplete#enable_at_startup = 1
63     let g:deoplete#sources#clang#libclang_path = '/usr/lib/llvm-10/lib/libclang.so'
64     let g:deoplete#sources#clang#clang_header = '/usr/lib/llvm-10/lib/clang/'
65     let g:deoplete#sources#clang#clang_complete_database = './'
66     "g:deoplete#sources#clang#flags See this section No
67     "g:deoplete#sources#clang#sort_algo '' No
68     "g:deoplete#sources#clang#include_default_arguments False No
69     "g:deoplete#sources#clang#filter_availability_kinds
70 endif

```

vim_config/nvim/plugin/buffers.vim

```
1 command! -nargs=0 Buff      call buffers#begin()
```

vim_config/nvim/plugin/cd.vim

```
1 command! -nargs=0 Cd      call cd#change_dir()
2 nmap <C-c>          :Cd<CR>
3
4 nmap <S-c>          :cd ..<CR>:pwd<CR>
5
```

vim_config/nvim/plugin/clear_undo.vim

```

1 function! ClearUndo()
2     let old_undolevels = &undolevels
3     set undolevels=-1
4     exe "normal a \<BS>\<Esc>"
5     let &undolevels = old_undolevels
6     unlet old_undolevels
7 endfunction
8
9 command! -nargs=0 ClearUndo call ClearUndo()
10
11
```

vim_config/nvim/plugin/ctags_ext.vim

```
1 function! CTags(...)
2     if a:0 == 0
3         let l:dir="."
4     else
5         let l:dir=a:1
6     end
7     silent execute '!ctags -R --extras+=q ' . l:dir
8 endfunction
9
10 command! -nargs=? -complete=dir Ctags call CTags(<f-args>)
11
```

vim_config/nvim/plugin/dev_env.vim

```
1 function! dev_env#setup()
2     execute ":only"
3     execute ":set lines=62"
4     execute ":set columns=201"
5
6     "少し待たないとカラムが増える前にvsplitしてしまう
7     execute ":sleep 300m"
8     execute ":vsplit"
9     execute ":split"
10    execute "normal \<C-W>l"
11    execute ":split"
12    execute "normal \<C-W>j"
13    execute ":Term"
14    execute "normal \<C-W>h"
15    execute "normal \<C-W>k"
16    set textwidth=100
17 endfunction
18
19 command! -nargs=0 DevEnv      call dev_env#setup()
```

vim_config/nvim/plugin/git_diff.vim

```
1 command! -nargs=0 GitDiff      call git_diff#begin(0)
2 command! -nargs=0 GitDiffResize call git_diff#begin(1)
3 command! -nargs=0 GitDiffOff   call git_diff#show_diff_off()
```

vim_config/nvim/plugin/git_session.vim

```
1 command! -nargs=0 SessionBegin call git_session#begin()
2 command! -nargs=0 SessionDir   call git_session#dir()
3 command! -nargs=0 SessionMake  call git_session#make()
4 command! -nargs=0 SessionNew   call git_session#new_session()
5 command! -nargs=0 SessionPath  call git_session#set_path()
```

vim_config/nvim/plugin/grep.vim

```
1 command! -nargs=? Grep call grep#grep(<f-args>)
2 command! -nargs+= Grepp call grep#grep(<f-args>)
3
```

vim_config/nvim/plugin/keybind.vim

```
1 "Set up key binding
2
3 " window control
4 nmap ;  4<C-W>+
5 nmap -  4<C-W>-
6 noremap L  <C-L>
7 nmap >  <C-W>>
8 nmap <  <C-W><
9 nmap <C-k>  <C-W>k
10 nmap <C-j> <C-W>j
11 nmap <C-h> <C-W>h
12 nmap <C-l> <C-W>l
13 nmap <C-p> <C-o>
14
15 "next/prev file
16 nmap <C-n>  :bn<CR>
17 nmap <C-p>  :bp<CR>
18
```

```
19 "next/prev quick fix
20 nmap <C-g><C-n> :cn<CR>
21 nmap <C-g><C-p> :cp<CR>
22
```

vim_config/nvim/plugin/multi_hl.vim

```
1 command! -nargs=0 MultiHlAdd call multi_hl#add_word(expand('<cword>'), 0)
2 command! -nargs=0 MultiHlAddR call multi_hl#add_word(expand('<cword>'), 1)
3 command! -nargs=1 MultiHlAddI call multi_hl#add_word(<q-args>, 1)
4 command! -nargs=0 MultiHlClear call multi_hl#clear()
```

vim_config/nvim/plugin/next_file.vim

```
1 if g:os == 'linux'
2   nmap <M-q>      :let @q=expand('%')<CR>
3 else
4   nmap <S-C-q>    :let @q=expand('%')<CR>
5 endif
6
7 command! -count NF call next_file#change(<count>)
8 nmap <C-q>      :NF<CR>
```

vim_config/nvim/plugin/path_set.vim

```
1 function! path_set#add()
2   let temp = system('find $(pwd) -type d')
3   let dirs = split(temp, "\n")
4   let paths = &path . ',' . join(dirs, ',')
5   let &path = paths
6 endfunction
7
8 function! path_set#clear()
9   let &path = './'
10 endfunction
11
12 function! path_set#set()
13   call path_set#clear()
14   call path_set#add()
15 endfunction
16
17 command! -nargs=0 PathAdd call path_set#add()
18 command! -nargs=0 PathClear call path_set#clear()
19 command! -nargs=0 PathSet call path_set#set()
```

vim_config/nvim/plugin/scratch.vim

```
1 let s:scratch_buffer = 'Scratch'
2
3 function! scratch#begin()
4   let winid = bufwinid(s:scratch_buffer)
5   if winid isnotequal -1
6     call win_gotoid(winid)
7   else
8     execute 'new' s:scratch_buffer
9     set buftype=nofile
10    set noswapfile
11  endif
12 endfunction
13
14 command! -nargs=0 Scratch call scratch#begin()
15
```

vim_config/nvim/plugin/term.vim

```
1 if g:os == 'windows'
2   set shell=C:/cygwin64/bin/bash.exe
3 else
4   set shell=/bin/bash
5 endif
6
7 set shellpipe=\|&\ tee
8 set shellcmdflag=-c
9 set shellslash
10
11 tnoremap <silent> <C-q> <C-\><C-n>
```

```
12 command! -nargs=0 Term      call term#start()
13
```

vim_config/nvim/plugin/termdbg.vim

```
1 command! -nargs=1 -complete=file DbgStart  call termdbg#start(<q-args>)
2 command! -nargs=0 DbgKey      call termdbg#key_map()
3
4
```

vim_config/nvim/rplugin/python3/next_file.py

```
1 import pynvim
2 import glob
3
4 @pynvim.plugin
5 class NextFile(object):
6
7     def __init__(self, nvim):
8         self.nvim = nvim
9         self.delete_len = 0
10        self.latest_file = None
11        self.file_list = []
12
13    @pynvim.command('NextFile', nargs='?', range='', sync = True)
14    def next_file(self, args, range):
15        if len(args) != 0 :
16            try :
17                self.delete_len = int(args[0])
18            except:
19                self.delete_len = 0
20
21        curr_filename = self.nvim.eval("expand('%')")
22        next_filename = self._next_filename(curr_filename)
23        self.nvim.command(f"edit {next_filename}")
24
25    def _next_filename(self, filename) :
26
27        if self.delete_len == 0 :
28            body, *_ = filename.split(".")
29        else:
30            body = filename[0 : len(filename) - self.delete_len]
31            self.latest_file = None
32
33        if self.latest_file != filename :
34            self.file_list = glob.glob(body + "*")
35
36        for i, f in enumerate(self.file_list) :
37            if filename == f :
38                self.latest_file = self.file_list[(i + 1) % len(self.file_list)]
39
40        return self.latest_file
41
```

vim_config/vim.sh

```
1 function os_name()
2 {
3     local -r uname=$(uname)
4
5     if [[ $uname =~ .*[lL]inux.* ]]; then
6         echo linux
7     elif [[ $uname =~ .*CYGWIN_NT.* ]]; then
8         echo cygwin
9     else
10        echo unknown
11    fi
12 }
13
14 readonly OS=$(os_name)
15
16 if [[ $OS == 'cygwin' ]]; then
17     function gvim()
18     {
19         (
20             gvim_prog=$(which gvim)
21             export XDG_CONFIG_HOME=$(cygpath -w $XDG_CONFIG_HOME)
22             $gvim_prog "$@"
23     )
24
25     fi
26 }
27
28 if [[ $OS == 'linux' ]]; then
29     function gvim()
30     {
31         (
32             gvim_prog=$(which gvim)
33             export XDG_CONFIG_HOME=$HOME/.config
34             $gvim_prog "$@"
35     )
36
37     fi
38 }
```

```

23      )
24  }
25
26  function _nvim_qt()
27  {
28      (
29          export XDG_CONFIG_HOME=$(cygpath -w $XDG_CONFIG_HOME)
30          nvim-qt "$@"
31      )
32  }
33
34  function _nvim()
35  {
36      (
37          export XDG_CONFIG_HOME=$(cygpath -w $XDG_CONFIG_HOME)
38          nvim "$@"
39      )
40  }
41  alias nvim-qt=_nvim_qt
42  alias nvim=_nvim
43
44 elif [[ $OS == 'linux' ]]; then
45     function tnvim()
46     {
47         /mnt/c/Users/ichiro.inoue/AppData/Local/wsltty/bin/mintty.exe \
48             --WSL= \
49             --configdir='C:\Users\ichiro.inoue\AppData\Roaming\wsltty' \
50             nvim "$@"
51     }
52
53     alias tn=tnvim
54 fi

```

vim_config/vim/gvimrc

```

1 "このファイルは$HOME/.gvimrcにコピーするかシンボリックリンクを張る。
2
3 set guifont=M S _ゴシック:h10:cSHIFTJIS
4
5 let g:bg_color='black'
6 highlight Normal guibg='black' guifg='white'
7

```

vim_config/vim/vimrc

```

1 "このファイルは$HOME/.vimrcにコピーするかシンボリックリンクを張る。
2
3 source $XDG_CONFIG_HOME/nvim/init.vim

```

C++

example/deps/app/srcdeps_opts.cpp

```

1 #include <getopt.h>
2
3 #include <cassert>
4 #include <sstream>
5
6 #include "deps_opts.h"
7 #include "lib/nstd.h"
8
9 namespace App {
10 std::string DepsOpts::Help()
11 {
12     auto ss = std::ostringstream{};
13
14     ss << "deps CMD [option] [DIRS] ..." << std::endl;
15     ss << "    CMD:" << std::endl;
16     ss << "        p   : generate package to OUT." << std::endl;
17     ss << "        s   : generate srcs with incs to OUT." << std::endl;
18     ss << "        p2s : generate package and srcs pairs to OUT." << std::endl;
19     ss << "        p2p : generate packages' dependencies to OUT." << std::endl;
20     ss << "        a   : generate structure to OUT from p2p output." << std::endl;
21     ss << "        a2pu : generate plant uml package to OUT from p2p output." << std::endl;
22     ss << "        cyc  : exit !0 if found cyclic dependencies." << std::endl;
23     ss << "        help : show help message." << std::endl;

```

```

24     ss << "      h    : same as help(-h, --help)." << std::endl;
25     ss << std::endl;
26     ss << "      opptions:" << std::endl;
27     ss << "      --in IN   : use IN to execute CMD." << std::endl;
28     ss << "      --out OUT  : CMD outputs to OUT." << std::endl;
29     ss << "      --recursive : search dir as package from DIRS or IN contents." << std::endl;
30     ss << "      -R       : same as --recursive." << std::endl;
31     ss << "      --src_as_pkg: every src is as a package." << std::endl;
32     ss << "      -s       : same as --src_as_pkg." << std::endl;
33     ss << "      --log LOG   : log to LOG(if LOG is \"-\\", using STDOUT)." << std::endl;
34     ss << "      --exclude PTN : exclude dirs which matchs to PTN(JS regex)." << std::endl;
35     ss << "      -e PTN    : same as --exclude." << std::endl;
36     ss << std::endl;
37     ss << "      DIRS: use DIRS to execute CMD." << std::endl;
38     ss << "      IN   : 1st line in this file must be" << std::endl;
39     ss << "          #dir2srcs for pkg-srcs file" << std::endl;
40     ss << "          or" << std::endl;
41     ss << "          #dir for pkg file." << std::endl << std::endl;
42
43     return ss.str();
44 }
45
46 DepsOpts::Cmd DepsOpts::parse_command(int argc, char* const* argv)
47 {
48     if (argc < 2) {
49         return Cmd::NotCmd;
50     }
51
52     auto command = std::string{argv[1]};
53
54     if (command == "p") {
55         return Cmd::GenPkg;
56     }
57     if (command == "s") {
58         return Cmd::GenSrc;
59     }
60     else if (command == "p2s") {
61         return Cmd::GenPkg2Srcs;
62     }
63     else if (command == "p2p") {
64         return Cmd::GenPkg2Pkg;
65     }
66     else if (command == "a") {
67         return Cmd::GenArch;
68     }
69     else if (command == "a2pu") {
70         return Cmd::GenPlantUml;
71     }
72     else if (command == "cyc") {
73         return Cmd::GenCyclic;
74     }
75     else if (command == "h" || command == "help" || command == "-h" || command == "--help") {
76         return Cmd::Help;
77     }
78
79     return Cmd::NotCmd;
80 }
81
82 bool D depsOpts::parse_opt(int opt_char, D depsOpts::DepsOptsData& data) noexcept
83 {
84     switch (opt_char) {
85     case 'i':
86         data.in = optarg;
87         return true;
88     case 'e':
89         data.exclude = optarg;
90         return true;
91     case 'o':
92         data.out = optarg;
93         return true;
94     case 'l':
95         data.log = optarg;
96         return true;
97     case 'R':
98         data.recursive = true;
99         return true;
100    case 's':
101        data.src_as_pkg = true;
102        return true;
103    case 'h':

```

```

104         data.cmd = Cmd::Help;
105     return false;
106 default:
107     return false;
108 }
109 }
110
111 DepsOpts::DepsOptsData DepsOpts::parse(int argc, char* const* argv)
112 {
113     DepsOptsData data{parse_command(argc, argv)};
114
115     if (data.cmd == Cmd::NotCmd || data.cmd == Cmd::Help) {
116         return data;
117     }
118
119     optind = 2;
120     static struct option const opts[]
121     = {{"in", required_argument, 0, 'i'}, {"out", required_argument, 0, 'o'},
122         {"exclude", required_argument, 0, 'e'}, {"recursive", no_argument, 0, 'R'},
123         {"src_as_pkg", no_argument, 0, 's'}, {"log", required_argument, 0, 'l'},
124         {"help", no_argument, 0, 'h'}, {0, 0, 0, 0}};
125
126     for (;;) {
127         auto opt_char = getopt_long(argc, argv, "i:o:e:l:Rsh", opts, nullptr);
128
129         if (!parse_opt(opt_char, data)) {
130             break;
131         }
132     }
133
134     if (optind < argc) {
135         while (optind < argc) {
136             data.dirs.emplace_back(FileUtils::NormalizeLexically(argv[optind++]));
137         }
138     }
139
140     return data;
141 }
142
143 namespace {
144 std::string to_string_cmd(DepsOpts::Cmd cmd)
145 {
146     switch (cmd) {
147     case DepsOpts::Cmd::GenPkg:
148         return "GenPkg";
149     case DepsOpts::Cmd::GenSrc:
150         return "GenSrc";
151     case DepsOpts::Cmd::GenPkg2Srcs:
152         return "GenPkg2Srcs";
153     case DepsOpts::Cmd::GenPkg2Pkg:
154         return "GenPkg2Pkg";
155     case DepsOpts::Cmd::GenPlantUml:
156         return "GenPlantUml";
157     case DepsOpts::Cmd::GenCyclic:
158         return "GenCyclic";
159     case DepsOpts::Cmd::Help:
160         return "Help";
161     case DepsOpts::Cmd::NotCmd:
162     default:
163         return "NotCmd";
164     }
165 }
166 } // namespace
167
168 std::string ToStringDepsOpts(DepsOpts const& deps_opts, std::string_view indent)
169 {
170     auto ss      = std::ostringstream{};
171     char const cmd[] = "cmd      : ";
172     auto const indent2 = std::string(Nstd::ArrayLength(cmd) - 1, ' ') + std::string{indent};
173
174     ss << std::boolalpha;
175
176     ss << indent << cmd << to_string_cmd(deps_opts.GetCmd()) << std::endl;
177     ss << indent << "in      : " << deps_opts.In() << std::endl;
178     ss << indent << "out     : " << deps_opts.Out() << std::endl;
179     ss << indent << "recursive : " << deps_opts.IsRecursive() << std::endl;
180     ss << indent << "src_as_pkg: " << deps_opts.IsSrcPkg() << std::endl;
181     ss << indent << "log      : " << deps_opts.Log() << std::endl;
182     ss << indent << "dirs     : " << FileUtils::ToStringPaths(deps_opts.Dirs(), "\n" + indent2)
183         << std::endl;

```

```

184     ss << indent << "exclude   : " << deps_opts.Exclude() << std::endl;
185     ss << indent << "parsed    : " << !deps_opts;
186
187     return ss.str();
188 }
189 } // namespace App

```

example/deps/app/src/deps_opts.h

```

1 #pragma once
2 #include <iostream>
3
4 #include "file_utils/path_utils.h"
5
6 namespace App {
7 class DepsOpts {
8 public:
9     enum class Cmd {
10         GenPkg,
11         GenSrc,
12         GenPkg2Srcs,
13         GenPkg2Pkg,
14         GenArch,
15         GenPlantUml,
16         GenCyclic,
17         Help,
18         NotCmd,
19     };
20     explicit DepsOpts(int argc, char* const* argv) : data_{parse(argc, argv)} {}
21     static std::string Help();
22
23     Cmd GetCmd() const noexcept { return data_.cmd; }
24     std::string const& In() const noexcept { return data_.in; }
25     std::string const& Out() const noexcept { return data_.out; }
26     std::string const& Log() const noexcept { return data_.log; }
27     bool IsRecursive() const noexcept { return data_.recursive; }
28     bool IsSrcPkg() const noexcept { return data_.src_as_pkg; }
29     FileUtils::Paths_t const& Dirs() const noexcept { return data_.dirs; }
30     std::string const& Exclude() const noexcept { return data_.exclude; }
31
32     explicit operator bool() const { return data_.cmd != Cmd::NotCmd; }
33
34 private:
35     struct DepsOptsData {
36         DepsOptsData(Cmd cmd_arg) noexcept : cmd{cmd_arg} {}
37         Cmd cmd;
38         std::string in{};
39         std::string out{};
40         std::string log{};
41         FileUtils::Paths_t dirs{};
42         std::string exclude{};
43         bool recursive{false};
44         bool src_as_pkg{false};
45     };
46     DepsOptsData const data_;
47
48     static DepsOptsData parse(int argc, char* const* argv);
49     static Cmd parse_command(int argc, char* const* argv);
50     static bool parse_opt(int opt_char, DepsOptsData& data) noexcept;
51 };
52
53 // @@@ sample begin 0:0
54
55 std::string ToStringDepsOpts(DepsOpts const& deps_opts, std::string_view indent = "");
56 inline std::ostream& operator<<(std::ostream& os, DepsOpts const& opts)
57 {
58     return os << ToStringDepsOpts(opts);
59 }
60 // @@@ sample end
61 } // namespace App

```

example/deps/app/src/main.cpp

```

1 #include <cassert>
2 #include <fstream>
3 #include <iostream>
4 #include <stdexcept>
5
6 #include "dependency/deps_scenario.h"

```

```

7 #include "deps_opts.h"
8 #include "logging/logger.h"
9
10 namespace {
11
12 class OStreamSelector {
13 public:
14     explicit OStreamSelector(std::string const& out) : os_{select(out, out_f_)} {}
15     std::ostream& OStream() noexcept { return os_; }
16
17 private:
18     std::ofstream out_f_{};
19     std::ostream& os_;
20
21     static std::ostream& select(std::string const& out, std::ofstream& out_f)
22     {
23         if (out.size()) {
24             out_f.open(out);
25             assert(out_f);
26             return out_f;
27         }
28         else {
29             return std::cout;
30         }
31     }
32 };
33
34 class ScenarioGeneratorNop : public Dependency::ScenarioGenerator {
35 public:
36     explicit ScenarioGeneratorNop(bool no_error) : no_error_{no_error} {}
37     virtual bool Output(std::ostream&) const noexcept override { return no_error_; }
38
39 private:
40     bool no_error_;
41 };
42
43 // @@@ sample begin 0:0
44 std::unique_ptr<Dependency::ScenarioGenerator> gen_scenario(App::DepsOpts const& opt)
45 try {
46     using namespace Dependency;
47
48     switch (opt.GetCmd()) {
49     case App::DepsOpts::Cmd::GenPkg:
50         LOGGER("start GenPkg");
51         return std::make_unique<PkgGenerator>(opt.In(), opt.IsRecursive(), opt.Dirs(),
52                                              opt.Exclude());
53     case App::DepsOpts::Cmd::GenSrc:
54         LOGGER("start GenPkg");
55         return std::make_unique<SrcsGenerator>(opt.In(), opt.IsRecursive(), opt.Dirs(),
56                                              opt.Exclude());
57     // @@@ ignore begin
58     case App::DepsOpts::Cmd::GenPkg2Srcs:
59         LOGGER("start GenPkg2Srcs");
60         return std::make_unique<Pkg2SrcsGenerator>(opt.In(), opt.IsRecursive(), opt.IsSrcPkg(),
61                                              opt.Dirs(), opt.Exclude());
62     case App::DepsOpts::Cmd::GenPkg2Pkg:
63         LOGGER("start GenPkg2Pkg");
64         return std::make_unique<Pkg2PkgGenerator>(opt.In(), opt.IsRecursive(), opt.IsSrcPkg(),
65                                              opt.Dirs(), opt.Exclude());
66     case App::DepsOpts::Cmd::GenArch:
67         LOGGER("start GenArch");
68         return std::make_unique<ArchGenerator>(opt.In());
69     case App::DepsOpts::Cmd::GenPlantUml:
70         LOGGER("start GenPlantUml");
71         return std::make_unique<Arch2PUmlGenerator>(opt.In());
72     case App::DepsOpts::Cmd::GenCyclic:
73         LOGGER("start GenCyclic");
74         return std::make_unique<CyclicGenerator>(opt.In());
75     case App::DepsOpts::Cmd::Help:
76         std::cout << App::DepsOpts::Help() << std::endl;
77         return std::make_unique<ScenarioGeneratorNop>(true);
78     case App::DepsOpts::Cmd::NotCmd:
79     default:
80         std::cout << App::DepsOpts::Help() << std::endl;
81         return std::make_unique<ScenarioGeneratorNop>(false);
82     // @@@ ignore end
83     }
84 }
85 catch (std::runtime_error const& e) {
86     LOGGER("error occurred:", e.what());

```

```

87     std::cerr << e.what() << std::endl;
88
89     return std::make_unique<ScenarioGeneratorNop>(false);
90 }
91
92 // @@@ ignore begin
93 catch (...) {
94     LOGGER("unknown error occurred:");
95
96     return std::make_unique<ScenarioGeneratorNop>(false);
97 }
98 } // namespace
99 // @@@ ignore end
100
101
102 int main(int argc, char* argv[])
103 {
104     App::DepsOpts d_opt{argc, argv};
105
106     LOGGER_INIT(d_opt.Log() == "-" ? nullptr : d_opt.Log().c_str());
107
108     LOGGER("Options", '\n', d_opt);
109
110     auto out_sel = OStreamSelector{d_opt.Out()};
111     auto exit_code = gen_scenario(d_opt)->Output(out_sel.OStream()) ? 0 : -1;
112
113     LOGGER("Exit", exit_code);
114
115     return exit_code;
116 }
117 // @@@ sample end

```

exampledeps/app/ut/deps_opts_ut.cpp

```

1 #include "gtest_wrapper.h"
2
3 #include "deps_opts.h"
4 #include "lib/nstd.h"
5
6 namespace App {
7 namespace {
8
9 TEST(deps_args, DepsOpts)
10 {
11     using FileUtils::Paths_t;
12
13     char prog[] = "prog";
14     char cmd_p[] = "p";
15     char cmd_s[] = "s";
16     char cmd_p2s[] = "p2s";
17     char cmd_p2p[] = "p2p";
18     char cmd_a[] = "a";
19     char cmd_a2pu[] = "a2pu";
20     char cmd_help[] = "help";
21     char cmd_dd_help[] = "--help";
22     char cmd_h[] = "h";
23     char cmd_d_h[] = "-h";
24     char cmd_unknown[] = "unknown";
25     char opt_in[] = "--in";
26     char opt_in_arg[] = "in-file";
27     char opt_out[] = "--out";
28     char opt_out_arg[] = "out-file";
29     char opt_e[] = "-e";
30     char opt_exclude[] = "--exclude";
31     char opt_e_arg[] = "pattern.*";
32     char opt_recursive[] = "--recursive";
33     char opt_src_pkg[] = "--src_as_pkg";
34     char opt_log[] = "--log";
35     char opt_log_arg[] = "log-file";
36     char opt_log_dash[] = "-";
37     char opt_R[] = "-R";
38     char opt_s[] = "-s";
39     char opt_help[] = "--help";
40     char opt_h[] = "-h";
41     char dir0[] = "dir0";
42     char dir1[] = "dir1";
43     char dir2[] = "dir2";
44
45     {
46         char* const argv[] {prog, cmd_p, opt_recursive, opt_out, opt_out_arg, dir0, dir1, dir2};

```

```

47     auto d_opt = DepsOpts{Nstd::ArrayLength(argv), argv};
48
49     ASSERT_EQ(DepsOpts::Cmd::GenPkg, d_opt.GetCmd());
50     ASSERT_EQ("", d_opt.In());
51     ASSERT_EQ(opt_out_arg, d_opt.Out());
52     ASSERT_TRUE(d_opt.IsRecursive());
53     ASSERT_EQ((Paths_t{dir0, dir1, dir2}), d_opt.Dirs());
54     ASSERT_TRUE(d_opt);
55 }
56 {
57     char* const argv[] = {prog, cmd_p, opt_src_pkg, opt_out, opt_out_arg, dir0, dir1, dir2};
58
59     auto d_opt = DepsOpts{Nstd::ArrayLength(argv), argv};
60
61     ASSERT_EQ(DepsOpts::Cmd::GenPkg, d_opt.GetCmd());
62     ASSERT_EQ("", d_opt.In());
63     ASSERT_EQ(opt_out_arg, d_opt.Out());
64     ASSERT_FALSE(d_opt.IsRecursive());
65     ASSERT_TRUE(d_opt.IsSrcPkg());
66     ASSERT_EQ((Paths_t{dir0, dir1, dir2}), d_opt.Dirs());
67     ASSERT_TRUE(d_opt);
68 }
69 {
70     char* const argv[] {prog, cmd_s, opt_src_pkg, opt_out, opt_out_arg, dir0, dir1, dir2};
71
72     auto d_opt = DepsOpts{Nstd::ArrayLength(argv), argv};
73
74     ASSERT_EQ(DepsOpts::Cmd::GenSrc, d_opt.GetCmd());
75     ASSERT_EQ("", d_opt.In());
76     ASSERT_EQ(opt_out_arg, d_opt.Out());
77     ASSERT_FALSE(d_opt.IsRecursive());
78     ASSERT_TRUE(d_opt.IsSrcPkg());
79     ASSERT_EQ((Paths_t{dir0, dir1, dir2}), d_opt.Dirs());
80     ASSERT_TRUE(d_opt);
81 }
82 {
83     char* const argv[] {prog, cmd_p2s, opt_R, opt_in, opt_in_arg, opt_exclude, opt_e_arg, dir0};
84
85     auto d_opt = DepsOpts{Nstd::ArrayLength(argv), argv};
86
87     ASSERT_EQ(DepsOpts::Cmd::GenPkg2Srcs, d_opt.GetCmd());
88     ASSERT_EQ(opt_in_arg, d_opt.In());
89     ASSERT_EQ("", d_opt.Out());
90     ASSERT_TRUE(d_opt.IsRecursive());
91     ASSERT_EQ((Paths_t{dir0}), d_opt.Dirs());
92     ASSERT_TRUE(d_opt);
93     ASSERT_EQ(opt_e_arg, d_opt.Exclude());
94 }
95 {
96     char* const argv[] {prog, cmd_p2s, opt_s, opt_in, opt_in_arg, opt_e, opt_e_arg, dir0};
97
98     auto d_opt = DepsOpts{Nstd::ArrayLength(argv), argv};
99
100    ASSERT_EQ(DepsOpts::Cmd::GenPkg2Srcs, d_opt.GetCmd());
101    ASSERT_EQ(opt_in_arg, d_opt.In());
102    ASSERT_EQ("", d_opt.Out());
103    ASSERT_FALSE(d_opt.IsRecursive());
104    ASSERT_TRUE(d_opt.IsSrcPkg());
105    ASSERT_EQ((Paths_t{dir0}), d_opt.Dirs());
106    ASSERT_TRUE(d_opt);
107    ASSERT_EQ(opt_e_arg, d_opt.Exclude());
108 }
109 {
110     char* const argv[] {prog, cmd_p2p, opt_in, opt_in_arg, opt_out, opt_out_arg};
111
112     auto d_opt = DepsOpts{Nstd::ArrayLength(argv), argv};
113
114     ASSERT_EQ(DepsOpts::Cmd::GenPkg2Pkg, d_opt.GetCmd());
115     ASSERT_EQ(opt_in_arg, d_opt.In());
116     ASSERT_EQ(opt_out_arg, d_opt.Out());
117     ASSERT_FALSE(d_opt.IsRecursive());
118     ASSERT_EQ(Paths_t{}, d_opt.Dirs());
119     ASSERT_TRUE(d_opt);
120 }
121 {
122     char* const argv[] {prog, cmd_a, opt_in, opt_in_arg, opt_out, opt_out_arg};
123
124     auto d_opt = DepsOpts{Nstd::ArrayLength(argv), argv};
125
126 }
```

```

127     ASSERT_EQ(DepsOpts::Cmd::GenArch, d_opt.GetCmd());
128     ASSERT_EQ(opt_in_arg, d_opt.In());
129     ASSERT_EQ(opt_out_arg, d_opt.Out());
130     ASSERT_FALSE(d_opt.IsRecursive());
131     ASSERT_EQ(Paths_t{}, d_opt.Dirs());
132     ASSERT_TRUE(d_opt);
133 }
134 {
135     char* const argv[] {prog, cmd_a2pu, opt_in, opt_in_arg, opt_out, opt_out_arg};
136
137     auto d_opt = DepsOpts{Nstd::ArrayLength(argv), argv};
138
139     ASSERT_EQ(DepsOpts::Cmd::GenPlantUml, d_opt.GetCmd());
140     ASSERT_EQ(opt_in_arg, d_opt.In());
141     ASSERT_EQ(opt_out_arg, d_opt.Out());
142     ASSERT_FALSE(d_opt.IsRecursive());
143     ASSERT_EQ(Paths_t{}, d_opt.Dirs());
144     ASSERT_TRUE(d_opt);
145 }
146 {
147     char* const argv[] {prog, cmd_help};
148
149     auto d_opt = DepsOpts{Nstd::ArrayLength(argv), argv};
150
151     ASSERT_EQ(DepsOpts::Cmd::Help, d_opt.GetCmd());
152     ASSERT_TRUE(d_opt);
153 }
154 {
155     char* const argv[] {prog, cmd_dd_help};
156
157     auto d_opt = DepsOpts{Nstd::ArrayLength(argv), argv};
158
159     ASSERT_EQ(DepsOpts::Cmd::Help, d_opt.GetCmd());
160     ASSERT_TRUE(d_opt);
161 }
162 {
163     char* const argv[] {prog, cmd_h};
164
165     auto d_opt = DepsOpts{Nstd::ArrayLength(argv), argv};
166
167     ASSERT_EQ(DepsOpts::Cmd::Help, d_opt.GetCmd());
168     ASSERT_TRUE(d_opt);
169 }
170 {
171     char* const argv[] {prog, cmd_d_h};
172
173     auto d_opt = DepsOpts{Nstd::ArrayLength(argv), argv};
174
175     ASSERT_EQ(DepsOpts::Cmd::Help, d_opt.GetCmd());
176     ASSERT_TRUE(d_opt);
177 }
178 {
179     char* const argv[] {prog, cmd_unknown};
180
181     auto d_opt = DepsOpts{Nstd::ArrayLength(argv), argv};
182
183     ASSERT_EQ(DepsOpts::Cmd::NotCmd, d_opt.GetCmd());
184     ASSERT_FALSE(d_opt);
185 }
186 {
187     char* const argv[] {prog, cmd_p, opt_recursive, opt_out, opt_out_arg, opt_help};
188
189     auto d_opt = DepsOpts{Nstd::ArrayLength(argv), argv};
190
191     ASSERT_EQ(DepsOpts::Cmd::Help, d_opt.GetCmd());
192     ASSERT_TRUE(d_opt);
193 }
194 {
195     char* const argv[] {prog, cmd_p, opt_recursive, opt_out, opt_out_arg, opt_h};
196
197     auto d_opt = DepsOpts{Nstd::ArrayLength(argv), argv};
198
199     ASSERT_EQ(DepsOpts::Cmd::Help, d_opt.GetCmd());
200     ASSERT_TRUE(d_opt);
201 }
202 {
203     char* const argv[] {prog, cmd_p, opt_log, opt_log_arg, opt_out, opt_out_arg};
204
205     auto d_opt = DepsOpts{Nstd::ArrayLength(argv), argv};

```

```

207     ASSERT_EQ(DepsOpts::Cmd::GenPkg, d_opt.GetCmd());
208     ASSERT_EQ(opt_log_arg, d_opt.Log());
209     ASSERT_EQ(opt_out_arg, d_opt.Out());
210     ASSERT_TRUE(d_opt);
211 }
212 {
213     char* const argv[] {prog, cmd_p, opt_log, opt_log_dash, opt_in, opt_in_arg};
214
215     auto d_opt = DepsOpts{Nstd::ArrayLength(argv), argv};
216
217     ASSERT_EQ(DepsOpts::Cmd::GenPkg, d_opt.GetCmd());
218     ASSERT_EQ(opt_log_dash, d_opt.Log());
219     ASSERT_EQ(opt_in_arg, d_opt.In());
220     ASSERT_TRUE(d_opt);
221 }
222 }
223 } // namespace
224 } // namespace App

```

exampledeps/dependency/h/dependency/deps_scenario.h

```

1 #pragma once
2 #include <memory>
3 #include <string>
4 #include <vector>
5
6 #include "file_utils/path_utils.h"
7
8 namespace Dependency {
9
10 class ScenarioGenerator {
11 public:
12     virtual bool Output(std::ostream& os) const = 0;
13     virtual ~ScenarioGenerator() {}
14 };
15
16 class PkgGenerator : public ScenarioGenerator {
17 public:
18     explicit PkgGenerator(std::string const& in, bool recursive, FileUtils::Paths_t const& dirs_opt,
19                           std::string const& pattern);
20     virtual bool Output(std::ostream& os) const override;
21
22 private:
23     FileUtils::Paths_t const dirs_;
24 };
25
26 class SrcsGenerator : public ScenarioGenerator {
27 public:
28     explicit SrcsGenerator(std::string const& in, bool recursive,
29                           FileUtils::Paths_t const& dirs_opt, std::string const& pattern);
30     virtual bool Output(std::ostream& os) const override;
31
32 private:
33     FileUtils::Paths_t const dirs_;
34 };
35
36 class Pkg2SrcsGenerator : public ScenarioGenerator {
37 public:
38     explicit Pkg2SrcsGenerator(std::string const& in, bool recursive, bool src_as_pkg,
39                               FileUtils::Paths_t const& dirs_opt, std::string const& pattern);
40     virtual bool Output(std::ostream& os) const override;
41
42 private:
43     FileUtils::Dirs2Srcs_t const dirs2srcs_;
44 };
45
46 class Pkg2PkgGenerator : public ScenarioGenerator {
47 public:
48     explicit Pkg2PkgGenerator(std::string const& in, bool recursive, bool src_as_pkg,
49                               FileUtils::Paths_t const& dirs_opt, std::string const& pattern);
50     virtual bool Output(std::ostream& os) const override;
51
52 private:
53     FileUtils::Dirs2Srcs_t const dirs2srcs_;
54 };
55
56 class ArchGenerator : public ScenarioGenerator {
57 public:
58     explicit ArchGenerator(std::string const& in);
59     virtual bool Output(std::ostream& os) const override;

```

```

60     ~ArchGenerator();
61
62 protected:
63     struct Impl;
64     std::unique_ptr<Impl> impl_;
65 };
66
67 class Arch2PUmlGenerator : public ArchGenerator {
68 public:
69     explicit Arch2PUmlGenerator(std::string const& in);
70     virtual bool Output(std::ostream& os) const override;
71 };
72
73 class CyclicGenerator : public ArchGenerator {
74 public:
75     explicit CyclicGenerator(std::string const& in);
76     virtual bool Output(std::ostream& os) const override;
77
78 private:
79     bool has_cyclic_dep_;
80 };
81 } // namespace Dependency

```

exampledepsdependencysrcarch_pkg.cpp

```

1 #include <cassert>
2 #include <sstream>
3
4 #include "arch_pkg.h"
5 #include "lib/nstd.h"
6
7 namespace Dependency {
8
9 void ArchPkg::set_cyclic(ArchPkg const* pkg, bool is_cyclic) const
10 {
11     assert(std::count(depend_on_.cbegin(), depend_on_.cend(), pkg) != 0);
12     assert(cyclic_.count(pkg) == 0 || cyclic_[pkg] == is_cyclic);
13
14     cyclic_.insert(std::make_pair(pkg, is_cyclic));
15 }
16
17 bool ArchPkg::is_cyclic(ArchPkgs_t& history, size_t depth) const
18 {
19     if (++depth > max_depth_) {
20         std::cerr << "too deep dependency:" << name_ << std::endl;
21         return true;
22     }
23
24     auto const it = find(history.cbegin(), history.cend(), this);
25
26     if (it != history.cend()) { // 循環検出
27         for (auto it2 = it; it2 != history.cend(); ++it2) {
28             auto next = (std::next(it2) == history.cend()) ? it : std::next(it2);
29             (*it2)->set_cyclic(*next, true);
30         }
31
32         // it == history.cbegin()ならば、一番上からの循環 A->B->C->...->A
33         // it != history.cbegin()ならば、上記以外の循環 A->B->C->...->B
34         return it == history.cbegin();
35     }
36
37     auto gs = Nstd::ScopedGuard{[&history] { history.pop_back(); }};
38     history.push_back(this);
39
40     for (ArchPkg const* pkg : depend_on_) {
41         if (pkg->is_cyclic(history, depth)) {
42             return true;
43         }
44     }
45
46     return false;
47 }
48
49 bool ArchPkg::IsCyclic(ArchPkg const& pkg) const
50 {
51     if (std::count(depend_on_.cbegin(), depend_on_.cend(), &pkg) == 0) {
52         return false;
53     }
54
55     if (cyclic_.count(&pkg) == 0) {

```

```

56     ArchPkgs_t history{this};
57     set_cyclic(&pkg, pkg.is_cyclic(history, 0));
58 }
59
60 assert(cyclic_.count(&pkg) != 0);
61
62 return cyclic_[&pkg];
63 }
64
65 bool ArchPkg::IsCyclic() const noexcept
66 {
67     for (ArchPkg const* pkg : DependOn()) {
68         if (IsCyclic(*pkg)) {
69             return true;
70         }
71     }
72
73     return false;
74 }
75
76 ArchPkg::Map_Path_ArchPkg_t ArchPkg::build_depend_on(DepRelation const& dep_rel,
77                                                       Map_Path_ArchPkg_t&& pkg_all)
78 {
79     auto const a_path = FileUtils::Path_t(dep_rel.PackageA);
80     if (pkg_all.count(a_path) == 0) {
81         pkg_all.insert(std::make_pair(a_path, std::make_unique<ArchPkg>(a_path)));
82     }
83
84     auto const b_path = FileUtils::Path_t(dep_rel.PackageB);
85     if (pkg_all.count(b_path) == 0) {
86         pkg_all.insert(std::make_pair(b_path, std::make_unique<ArchPkg>(b_path)));
87     }
88
89     ArchPkgPtr_t& a_ptr = pkg_all.at(a_path);
90     ArchPkgPtr_t& b_ptr = pkg_all.at(b_path);
91
92     if (dep_rel.CountAtoB != 0) {
93         a_ptr->depend_on_.push_back(b_ptr.get());
94     }
95     if (dep_rel.CountBtoA != 0) {
96         b_ptr->depend_on_.push_back(a_ptr.get());
97     }
98
99     return std::move(pkg_all);
100 }
101
102 Arch_t ArchPkg::build_children(Map_Path_ArchPkg_t&& pkg_all)
103 {
104     auto cache = std::map<FileUtils::Path_t, ArchPkg*>{};
105     auto top   = Arch_t{};
106
107     for (auto& [path, pkg] : pkg_all) { // C++17 style
108
109         auto const parent_name = path.parent_path();
110         cache.insert(std::make_pair(path, pkg.get()));
111
112         if (pkg_all.count(parent_name) == 0) {
113             top.emplace_back(std::move(pkg));
114         }
115         else {
116             ArchPkg* parent = cache.count(parent_name) != 0 ? cache.at(parent_name)
117                                                       : pkg_all.at(parent_name).get();
118
119             pkg->parent_ = parent;
120             parent->children_.emplace_back(std::move(pkg));
121         }
122     }
123
124     return top;
125 }
126
127 Arch_t ArchPkg::GenArch(DepRelPs_t const& dep_rels)
128 {
129     auto pkg_all = std::map<FileUtils::Path_t, ArchPkgPtr_t>{};
130
131     for (auto const& d : dep_rels) {
132         pkg_all = build_depend_on(d, std::move(pkg_all));
133     }
134
135     auto top = Arch_t{build_children(std::move(pkg_all))};

```

```

136
137     return top;
138 }
139
140 std::string ArchPkg::make_full_name(ArchPkg const& pkg)
141 {
142     if (pkg.Parent()) {
143         return make_full_name(*pkg.Parent()) + "/" + pkg.Name();
144     }
145     else {
146         return pkg.Name();
147     }
148 }
149
150 ArchPkg const* FindArchPkgByName(Arch_t const& arch, std::string_view pkg_name) noexcept
151 {
152     for (ArchPkgPtr_t const& pkg_ptr : arch) {
153         if (pkg_ptr->Name() == pkg_name) {
154             return pkg_ptr.get();
155         }
156         else {
157             ArchPkg const* pkg_found = FindArchPkgByName(pkg_ptr->Children(), pkg_name);
158             if (pkg_found) {
159                 return pkg_found;
160             }
161         }
162     }
163     return nullptr;
164 }
165
166 ArchPkg const* FindArchPkgByFullName(Arch_t const& arch, std::string_view full_name) noexcept
167 {
168     for (ArchPkgPtr_t const& pkg_ptr : arch) {
169         if (pkg_ptr->FullName() == full_name) {
170             return pkg_ptr.get();
171         }
172         else {
173             ArchPkg const* pkg_found = FindArchPkgByFullName(pkg_ptr->Children(), full_name);
174             if (pkg_found) {
175                 return pkg_found;
176             }
177         }
178     }
179     return nullptr;
180 }
181
182 namespace {
183
184 std::string unique_str_name(std::string const& full_name)
185 {
186     auto ret = Nstd::Replace(full_name, "/", "__");
187     return Nstd::Replace(ret, "-", "_");
188 }
189
190 std::string_view cyclic_str(ArchPkg const& pkg) noexcept
191 {
192     if (pkg.IsCyclic()) {
193         return ":CYCLIC";
194     }
195
196     return "";
197 }
198
199 std::string to_string_depend_on(ArchPkg const& pkg_top, uint32_t indent)
200 {
201     auto ss      = std::ostringstream{};
202     auto indent_str = std::string(indent, ' ');
203
204     auto first = true;
205
206     for (ArchPkg const* pkg : pkg_top.DependOn()) {
207         if (!std::exchange(first, false)) {
208             ss << std::endl;
209         }
210
211         ss << indent_str << pkg->Name();
212
213         if (pkg_top.IsCyclic(*pkg)) {
214             ss << " : CYCLIC";
215         }

```

```

216     else {
217         ss << " : STRAIGHT";
218     }
219 }
220
221 return ss.str();
222 }
223
224 std::string to_string_pkg(ArchPkg const& arch_pkg, uint32_t indent)
225 {
226     static auto const top      = std::string("TOP");
227     auto          ss        = std::ostringstream{};
228     auto          indent_str = std::string(indent, ' ');
229
230     auto package   = "package :";
231     auto full      = "fullname :";
232     auto parent    = "parent :";
233     auto children  = "children : {";
234     auto depend_on = "depend_on: {";
235
236     constexpr auto next_indent = 4U;
237
238     ss << indent_str << package << arch_pkg.Name() << cyclic_str(arch_pkg) << std::endl;
239     ss << indent_str << full << arch_pkg.FullName() << std::endl;
240     ss << indent_str << parent << (arch_pkg.Parent() ? arch_pkg.Parent()->Name() : top)
241         << std::endl;
242
243     ss << indent_str << depend_on;
244     if (arch_pkg.DependOn().size() != 0) {
245         ss << std::endl;
246         ss << to_string_depend_on(arch_pkg, indent + next_indent) << std::endl;
247         ss << indent_str << "}" << std::endl;
248     }
249     else {
250         ss << " }" << std::endl;
251     }
252
253     ss << indent_str << children;
254     if (arch_pkg.Children().size() != 0) {
255         ss << std::endl;
256         ss << ToStringArch(arch_pkg.Children(), indent + next_indent) << std::endl;
257         ss << indent_str << "}";
258     }
259     else {
260         ss << " }";
261     }
262
263     return ss.str();
264 }
265 } // namespace
266
267 std::string ToStringArch(Arch_t const& arch, uint32_t indent)
268 {
269     auto ss    = std::ostringstream{};
270     auto first = true;
271
272     for (auto const& pkg : arch) {
273         if (!std::exchange(first, false)) {
274             ss << std::endl << std::endl;
275         }
276         ss << to_string_pkg(*pkg, indent);
277     }
278
279     return ss.str();
280 }
281
282 namespace {
283 std::string to_pu_rectangle(ArchPkg const& pkg, uint32_t indent)
284 {
285     auto ss        = std::ostringstream{};
286     auto indent_str = std::string(indent, ' ');
287
288     ss << indent_str << "rectangle \\" << pkg.Name() << "\\" as " << unique_str_name(pkg.FullName());
289
290     if (pkg.Children().size() != 0) {
291         ss << " {" << std::endl;
292         ss << ToPlantUML_Rectangle(pkg.Children(), indent + 4);
293         ss << std::endl << indent_str << "}";
294     }
295 }
```

```

296     return ss.str();
297 }
298 } // namespace
299
300 std::string ToPlantUML_Rectangle(Arch_t const& arch, uint32_t indent)
301 {
302     auto ss    = std::ostringstream{};
303     auto first = true;
304
305     for (auto const& pkg : arch) {
306         if (!std::exchange(first, false)) {
307             ss << std::endl;
308         }
309         ss << to_pu_rectangle(*pkg, indent);
310     }
311
312     return ss.str();
313 }
314
315 namespace {
316
317 // 単方向依存のみ
318 bool dep_is_cyclic(std::string const& from, std::string const& to, Arch_t const& arch) noexcept
319 {
320     ArchPkg const* pkg_from = FindArchPkgByFullName(arch, from);
321     ArchPkg const* pkg_to   = FindArchPkgByFullName(arch, to);
322
323     assert(pkg_from != nullptr);
324     assert(pkg_to != nullptr);
325
326     return pkg_from->IsCyclic(*pkg_to);
327 }
328
329 std::string_view pu_link_color(std::string const& from, std::string const& to,
330                               Arch_t const& arch) noexcept
331 {
332     return dep_is_cyclic(from, to, arch) ? "orange" : "green";
333 }
334
335 std::string to_pu_rectangle(Arch_t const& arch, DepRelation const& dep_rel)
336 {
337     auto ss = std::ostringstream{};
338     auto a  = unique_str_name(dep_rel.PackageA);
339     auto b  = unique_str_name(dep_rel.PackageB);
340
341     if (dep_rel.CountAtoB != 0) {
342         ss << a << " \\" << dep_rel.CountAtoB << "\\" ";
343         if (dep_rel.CountBtoA != 0) {
344             ss << "<-[#red]-> \\" << dep_rel.CountBtoA << "\\" " << b;
345         }
346         else {
347             ss << "-[#" << pu_link_color(dep_rel.PackageA, dep_rel.PackageB, arch) << "]-> " << b;
348         }
349     }
350     else if (dep_rel.CountBtoA != 0) {
351         ss << b << " \\" << dep_rel.CountBtoA << "\\" -[#"
352             << pu_link_color(dep_rel.PackageB, dep_rel.PackageA, arch) << "]-> " << a;
353     }
354
355     return ss.str();
356 }
357 } // namespace
358
359 bool HasCyclicDeps(Arch_t const& arch, DepRels_t const& dep_rels) noexcept
360 {
361     for (auto const& dep : dep_rels) {
362         if (dep.CountAtoB != 0) {
363             if (dep.CountBtoA != 0) {
364                 return true;
365             }
366             else {
367                 if (dep_is_cyclic(dep.PackageA, dep.PackageB, arch)) {
368                     return true;
369                 }
370             }
371         }
372         else if (dep.CountBtoA != 0) {
373             if (dep_is_cyclic(dep.PackageB, dep.PackageA, arch)) {
374                 return true;
375             }

```

```

376         }
377     }
378
379     return false;
380 }
381
382 std::string ToPlantUML_Rectangle(Arch_t const& arch, DepRels_t const& dep_rels)
383 {
384     auto ss = std::ostringstream{};
385
386     auto first = true;
387     for (auto const& d : dep_rels) {
388         auto rel_s = to_pu_rectangle(arch, d);
389
390         if (rel_s.size() != 0) {
391             if (!std::exchange(first, false)) {
392                 ss << std::endl;
393             }
394             ss << rel_s;
395         }
396     }
397     return ss.str();
398 }
399 } // namespace Dependency

```

exampledeps/dependency/src/arch_pkg.h

```

1 #pragma once
2
3 #include "cpp_deps.h"
4 #include "file_utils/path_utils.h"
5
6 namespace Dependency {
7
8 class ArchPkg;
9 using ArchPkgPtr_t = std::unique_ptr<ArchPkg>;
10 using Arch_t          = std::list<ArchPkgPtr_t>;
11
12 using ArchPkgs_t = std::vector<ArchPkg const*>;
13
14 class ArchPkg {
15 public:
16     explicit ArchPkg(FileUtils::Path_t const& full_name)
17         : name_{full_name.filename()}, full_name_{full_name}
18     {
19     }
20
21     std::string const& Name() const noexcept { return name_; }
22     ArchPkg const* Parent() const noexcept { return parent_; }
23     Arch_t const& Children() const noexcept { return children_; }
24     ArchPkgs_t const& DependOn() const noexcept { return depend_on_; }
25     bool IsCyclic() const noexcept;
26     bool IsCyclic(ArchPkg const& pkg) const;
27     std::string const& FullName() const noexcept { return full_name_; }
28
29     ArchPkg(ArchPkg const&) = delete;
30     ArchPkg& operator=(ArchPkg const&) = delete;
31
32     static Arch_t GenArch(DepRels_t const& deps);
33
34 private:
35     std::string const           name_;
36     std::string const           full_name_{};
37     ArchPkg const*             parent_{};
38     Arch_t                     children_{};
39     ArchPkgs_t                depend_on_{};
40     mutable std::map<ArchPkg const*, bool> cyclic_{};
41     static constexpr size_t      max_depth_{12};
42
43     void set_cyclic(ArchPkg const* pkg, bool is_cyclic) const;
44
45     using Map_Path_ArchPkg_t = std::map<FileUtils::Path_t, ArchPkgPtr_t>;
46     static Map_Path_ArchPkg_t build_depend_on(DepRelation const& dep_rel,
47                                              Map_Path_ArchPkg_t& pkg_all);
48     static Arch_t               build_children(Map_Path_ArchPkg_t& pkg_all);
49     static std::string          make_full_name(ArchPkg const& pkg);
50     bool                      is_cyclic(ArchPkgs_t& history, size_t depth) const;
51 };
52
53 std::string      ToStringArch(Arch_t const& arch, uint32_t indent = 0);

```

```

54 inline std::ostream& operator<<(std::ostream& os, Arch_t const& arch)
55 {
56     return os << ToStringArch(arch);
57 }
58
59 std::string ToPlantUML_Rectangle(Arch_t const& arch, uint32_t indent = 0);
60 std::string ToPlantUML_Rectangle(Arch_t const& arch, DepRels_t const& dep_rels);
61 bool      HasCyclicDeps(Arch_t const& arch, DepRels_t const& dep_rels) noexcept;
62
63 ArchPkg const* FindArchPkgByName(Arch_t const& arch, std::string_view pkg_name) noexcept;
64 ArchPkg const* FindArchPkgByFullName(Arch_t const& arch, std::string_view full_name) noexcept;
65 } // namespace Dependency

```

exampledeps/dependency/src/cpp_deps.cpp

```

1 #include <algorithm>
2 #include <cassert>
3 #include <memory>
4 #include <iostream>
5 #include <tuple>
6
7 #include "cpp_deps.h"
8 #include "cpp_dir.h"
9 #include "cpp_src.h"
10
11 namespace Dependency {
12
13 bool operator==(DepRelation const& lhs, DepRelation const& rhs) noexcept
14 {
15     return std::tie(lhs.PackageA, lhs.CountAtoB, lhs.IncsAtoB, lhs.PackageB, lhs.CountBtoA,
16                     lhs.IncsBtoA)
17         == std::tie(rhs.PackageA, rhs.CountAtoB, rhs.IncsAtoB, rhs.PackageB, rhs.CountBtoA,
18                     rhs.IncsBtoA);
19 }
20
21 std::string ToStringDepRel(DepRelation const& rep_rel)
22 {
23     auto ss = std::ostringstream{};
24
25     ss << FileUtils::ToStringPath(rep_rel.PackageA) << " -> "
26         << FileUtils::ToStringPath(rep_rel.PackageB) << " : " << rep_rel.CountAtoB << " "
27         << FileUtils::ToStringPaths(rep_rel.IncsAtoB, " ") << std::endl;
28
29     ss << FileUtils::ToStringPath(rep_rel.PackageB) << " -> "
30         << FileUtils::ToStringPath(rep_rel.PackageA) << " : " << rep_rel.CountBtoA << " "
31         << FileUtils::ToStringPaths(rep_rel.IncsBtoA, " ");
32
33     return ss.str();
34 }
35
36 std::string ToStringDepRels(DepRels_t const& dep_rels)
37 {
38     auto ss = std::ostringstream{};
39
40     auto first = true;
41     for (auto const& dep : dep_rels) {
42         if (!std::exchange(first, false)) {
43             ss << std::endl;
44         }
45         ss << ToStringDepRel(dep) << std::endl;
46     }
47
48     return ss.str();
49 }
50
51 namespace {
52 DepRelation gen_DepRelation(CppDir const& dirA, CppDir const& dirB)
53 {
54     auto a_dep      = std::pair<uint32_t, FileUtils::Paths_t>{dirA.DependsOn(dirB)};
55     auto count_from_a = a_dep.first;
56     auto incs_from_a = std::move(a_dep.second);
57
58     auto b_dep      = std::pair<uint32_t, FileUtils::Paths_t>{dirB.DependsOn(dirA)};
59     auto count_from_b = b_dep.first;
60     auto incs_from_b = std::move(b_dep.second);
61
62     if (dirA < dirB) {
63         return DepRelation{dirA.Path(), count_from_a, std::move(incs_from_a),
64                            dirB.Path(), count_from_b, std::move(incs_from_b)};
65     }

```

```

66     else {
67         return DepRelation{dirB.Path(), count_from_b, std::move(incs_from_b),
68                             dirA.Path(), count_from_a, std::move(incs_from_a)};
69     }
70 }
71 } // namespace
72
73 Dir2Dir_t GenDir2Dir(std::string dirA, std::string dirB)
74 {
75     return dirA < dirB ? std::make_pair(std::move(dirA), std::move(dirB))
76                         : std::make_pair(std::move(dirB), std::move(dirA));
77 }
78
79 DepRels_t GenDepRels(CppDirs_t const& cpp_dirs)
80 {
81     auto ret = DepRels_t{};
82
83     for (auto const& dirA : cpp_dirs) {
84         for (auto const& dirB : cpp_dirs) {
85             if (dirA <= dirB) {
86                 continue;
87             }
88             ret.emplace_back(gen_DepRelation(dirA, dirB));
89         }
90     }
91     ret.sort();
92
93     return ret;
94 }
95 }
96
97 DepRels_t::const_iterator FindDepRels(DepRels_t const& dep_rels, std::string const& dirA,
98                                         std::string const& dirB) noexcept
99 {
100    assert(dirA != dirB);
101
102    auto dirs = std::minmax(dirA, dirB);
103
104    return std::find_if(dep_rels.cbegin(), dep_rels.cend(), [&dirs](auto const& d) noexcept {
105        return d.PackageA == dirs.first && d.PackageB == dirs.second;
106    });
107 }
108 } // namespace Dependency

```

exampledepsdependencysrccpp_deps.h

```

1 #pragma once
2 #include <vector>
3
4 #include "cpp_deps.h"
5 #include "cpp_dir.h"
6 #include "file_utils/path_utils.h"
7
8 namespace Dependency {
9
10 struct DepRelation {
11     explicit DepRelation(std::string package_a, uint32_t count_a2b, FileUtils::Paths_t&& incs_a2b,
12                          std::string package_b, uint32_t count_b2a, FileUtils::Paths_t&& incs_b2a)
13         : PackageA{std::move(package_a)},
14           CountAtoB{count_a2b},
15           IncsAtoB{std::move(incs_a2b)},
16           PackageB{std::move(package_b)},
17           CountBtoA{count_b2a},
18           IncsBtoA{std::move(incs_b2a)}
19     {
20     }
21
22     std::string const      PackageA;
23     uint32_t const        CountAtoB;
24     FileUtils::Paths_t const IncsAtoB;
25
26     std::string const      PackageB;
27     uint32_t const        CountBtoA;
28     FileUtils::Paths_t const IncsBtoA;
29   };
30
31 using Dir2Dir_t = std::pair<std::string, std::string>;
32 using DepRels_t = std::list<DepRelation>;
33
34 std::string ToStringDepRel(DepRelation const& rep_rel);

```

```

35
36 bool operator==(DepRelation const& lhs, DepRelation const& rhs) noexcept;
37 inline bool operator!=(DepRelation const& lhs, DepRelation const& rhs) noexcept
38 {
39     return !(lhs == rhs);
40 }
41 inline bool operator<(DepRelation const& lhs, DepRelation const& rhs) noexcept
42 {
43     return lhs.PackageA != rhs.PackageA ? lhs.PackageA < rhs.PackageA : lhs.PackageB < rhs.PackageB;
44 }
45
46 inline bool operator>(DepRelation const& lhs, DepRelation const& rhs) noexcept { return rhs < lhs; }
47
48 inline std::ostream& operator<<(std::ostream& os, DepRelation const& dep_rel)
49 {
50     return os << ToStringDepRel(dep_rel);
51 }
52
53 Dir2Dir_t GenDir2Dir(std::string const& dirA, std::string const& dirB);
54
55 std::string ToStringDepRels(DepRels_t const& dep_rels);
56 inline std::ostream& operator<<(std::ostream& os, DepRels_t const& dep_rels)
57 {
58     return os << ToStringDepRels(dep_rels);
59 }
60
61 DepRels_t GenDepRels(CppDirs_t const& dirs);
62 DepRels_t::const_iterator FindDepRels(DepRels_t const& dep_rels, std::string const& dirA,
63                                         std::string const& dirB) noexcept;
64 } // namespace Dependency

```

exampledeps/dependency/src/cpp_dir.cpp

```

1 #include <cassert>
2 #include <sstream>
3 #include <tuple>
4
5 #include "cpp_dir.h"
6 #include "cpp_src.h"
7 #include "lib/nstd.h"
8
9 namespace Dependency {
10
11 bool CppDir::Contains(FileUtils::Path_t const& inc_path) const noexcept
12 {
13     for (auto const& src : srcs_) {
14         if (src.Path() == inc_path) {
15             return true;
16         }
17     }
18
19     return false;
20 }
21
22 std::pair<uint32_t, FileUtils::Paths_t> CppDir::DependsOn(CppDir const& cpp_pack) const
23 {
24     auto count = 0U;
25     auto incs = FileUtils::Paths_t{};
26
27     for (auto const& src : srcs_) {
28         for (auto const& inc : src.GetIncs()) {
29             if (cpp_pack.Contains(inc)) {
30                 incs.push_back(inc);
31                 ++count;
32             }
33         }
34     }
35
36     Nstd::SortUnique(incs);
37
38     return {count, std::move(incs)};
39 }
40
41 bool operator==(CppDir const& lhs, CppDir const& rhs) noexcept
42 {
43     return std::tie(lhs.path_, lhs.srcs_) == std::tie(rhs.path_, rhs.srcs_);
44 }
45
46 bool operator<(CppDir const& lhs, CppDir const& rhs) noexcept
47 {

```

```

48     return std::tie(lhs.path_, lhs.srcts_) < std::tie(rhs.path_, rhs.srcts_);
49 }
50
51 CppDirs_t GenCppDirs(FileUtils::Paths_t const& srcts, FileUtils::Filename2Path_t const& db)
52 {
53     auto ret = CppDirs_t{};
54
55     for (auto const& src : srcts) {
56         auto cpp_src = CppSrc{src, db};
57         ret.emplace_back(CppDir{cpp_src.Filename(), {cpp_src}});
58     }
59
60     return ret;
61 }
62
63 std::string ToStringCppDir(CppDir const& cpp_pack)
64 {
65     auto ss = std::ostringstream{};
66
67     ss << FileUtils::ToStringPath(cpp_pack.Path()) << std::endl;
68
69     auto first = true;
70     for (auto const& src : cpp_pack.GetSrcts()) {
71         if (first) {
72             first = false;
73         }
74         else {
75             ss << std::endl;
76         }
77         ss << ToStringCppSrc(src);
78     }
79
80     return ss.str();
81 }
82 } // namespace Dependency

```

exampledeps/dependency/src/cpp_dir.h

```

1 #pragma once
2 #include <iostream>
3 #include <string>
4 #include <utility>
5
6 #include "cpp_src.h"
7 #include "file_utils/path_utils.h"
8
9 namespace Dependency {
10
11 class CppDir {
12 public:
13     explicit CppDir(FileUtils::Path_t const& path, CppSrcts_t&& srcts)
14         : path_{path}, srcts_{std::move(srcts)}
15     {
16     }
17
18     FileUtils::Path_t const& Path() const noexcept { return path_; }
19     bool Contains(FileUtils::Path_t const& inc_path) const noexcept;
20
21     // first 依存するヘッダファイルのインクルード数
22     // second 依存するヘッダファイル
23     std::pair<uint32_t, FileUtils::Paths_t> DependsOn(CppDir const& cpp_pack) const;
24     CppSrcts_t const& GetSrcts() const noexcept { return srcts_; }
25
26 private:
27     FileUtils::Path_t const path_;
28     CppSrcts_t const srcts_;
29
30     friend bool operator==(CppDir const& lhs, CppDir const& rhs) noexcept;
31     friend bool operator<(CppDir const& lhs, CppDir const& rhs) noexcept;
32 };
33
34 inline bool operator<=(CppDir const& lhs, CppDir const& rhs) noexcept
35 {
36     if (lhs == rhs) {
37         return true;
38     }
39
40     return lhs < rhs;
41 }
42

```

```

43 inline bool operator!=(CppDir const& lhs, CppDir const& rhs) noexcept { return !(lhs == rhs); }
44 inline bool operator>(CppDir const& lhs, CppDir const& rhs) noexcept { return rhs < lhs; }
45 inline bool operator>=(CppDir const& lhs, CppDir const& rhs) noexcept { return rhs <= lhs; }
46
47 using CppDirs_t = std::vector<CppDir>;
48
49 CppDirs_t GenCppDirs(FileUtils::Paths_t const& srcs, FileUtils::Filename2Path_t const& db);
50
51 std::string ToStringCppDir(CppDir const& cpp_pack);
52 inline std::ostream& operator<<(std::ostream& os, CppDir const& dir)
53 {
54     return os << ToStringCppDir(dir);
55 }
56 } // namespace Dependency

```

exampledepsdependency/src/cpp_src.cpp

```

1 #include <cassert>
2 #include <fstream>
3 #include <regex>
4 #include <sstream>
5 #include <tuple>
6
7 #include "cpp_src.h"
8 #include "lib/nstd.h"
9
10 namespace {
11
12 FileUtils::Paths_t get_incs(FileUtils::Path_t const& src)
13 {
14     static auto const include_line = std::regex{R"(^\s*#include\s+["<]([\w/.]+)[>](.*))"};
15
16     auto ret = FileUtils::Paths_t{};
17     auto f = std::ifstream{src};
18     auto line = std::string{};
19
20     while (std::getline(f, line)) {
21         if (line.size() > 0) { // CRLF対策
22             auto last = --line.end();
23             if (*last == '\xa' || *last == '\xd') {
24                 line.erase(last);
25             }
26         }
27
28         if (auto results = std::smatch{}; std::regex_match(line, results, include_line)) {
29             ret.emplace_back(FileUtils::Path_t(results[1].str()).filename());
30         }
31     }
32
33     return ret;
34 }
35
36 void get_incs_full(FileUtils::Filename2Path_t const& db, FileUtils::Path_t const& src,
37                     FileUtils::Paths_t& incs, FileUtils::Paths_t& not_found, bool sort_uniq)
38 {
39     auto const inc_files = get_incs(src);
40
41     for (auto const& f : inc_files) {
42         if (db.count(f) == 0) {
43             not_found.push_back(f);
44         }
45         else {
46             auto full_path = db.at(f);
47             if (!any_of(
48                 incs.cbegin(), incs.cend(),
49                 [&full_path](FileUtils::Path_t const& p) noexcept { return p == full_path; })) {
50                 incs.emplace_back(full_path);
51                 get_incs_full(db, full_path, incs, not_found, false);
52             }
53         }
54     }
55
56     if (sort_uniq) {
57         Nstd::SortUnique(incs);
58         Nstd::SortUnique(not_found);
59     }
60 }
61 } // namespace Dependency
62
63 namespace Dependency {

```

```

64 CppSrc::CppSrc(FileUtils::Path_t const& pathname, FileUtils::Filename2Path_t const& db)
65   : path_{FileUtils::NormalizeLexically(pathname)},
66   filename_{path_.filename()},
67   incs_{},
68   not_found_{}
69 {
70     get_incs_full(db, pathname, incs_, not_found_, true);
71 }
72
73
74 bool operator==(CppSrc const& lhs, CppSrc const& rhs) noexcept
75 {
76   return std::tie(lhs.path_, lhs.filename_, lhs.incs_, lhs.not_found_)
77     == std::tie(rhs.path_, rhs.filename_, rhs.incs_, rhs.not_found_);
78 }
79
80 bool operator<(CppSrc const& lhs, CppSrc const& rhs) noexcept
81 {
82   return std::tie(lhs.path_, lhs.filename_, lhs.incs_, lhs.not_found_)
83     < std::tie(rhs.path_, rhs.filename_, rhs.incs_, rhs.not_found_);
84 }
85
86 CppSrcs_t GenCppSrc(FileUtils::Paths_t const& srcs, FileUtils::Filename2Path_t const& db)
87 {
88   auto ret = CppSrcs_t{};
89
90   for (auto const& src : srcs) {
91     ret.emplace_back(CppSrc{src, db});
92   }
93
94   return ret;
95 }
96
97 std::string ToStringCppSrc(CppSrc const& cpp_src)
98 {
99   auto ss = std::ostringstream{};
100
101  ss << "file" : " << FileUtils::ToStringPath(cpp_src.Filename()) << std::endl;
102  ss << "path" : " << FileUtils::ToStringPath(cpp_src.Path()) << std::endl;
103  ss << "include" : " << FileUtils::ToStringPaths(cpp_src.GetIncs(), " ") << std::endl;
104  ss << "include not found" : " << FileUtils::ToStringPaths(cpp_src.GetIncsNotFound(), " ")
105    << std::endl;
106
107  return ss.str();
108 }
109
110 namespace {
111 constexpr std::string_view target_ext[]{"c", ".h", ".cpp", ".cxx", ".cc", ".hpp", "..hxx", ".tcc"};
112
113 bool is_c_or_cpp(std::string ext)
114 {
115   std::transform(ext.begin(), ext.end(), ext.begin(), ::tolower);
116
117   if (std::any_of(std::begin(target_ext), std::end(target_ext),
118                  [&ext](std::string_view s) noexcept { return s == ext; })) {
119     return true;
120   }
121
122   return false;
123 }
124
125 FileUtils::Paths_t gen_dirs(FileUtils::Path_t const& top_dir, FileUtils::Paths_t const& srcs)
126 {
127   auto dirs = FileUtils::Paths_t{top_dir};
128   auto const top_dir2 = FileUtils::Path_t{""}; // top_dirが""の場合、parent_path()は""になる}。
129
130   for (auto const& src : srcs) {
131     for (auto dir = src.parent_path(); dir != top_dir && dir != top_dir2;
132           dir = dir.parent_path()) {
133       dirs.push_back(dir);
134     }
135   }
136
137   return dirs;
138 }
139
140 FileUtils::Paths_t find_c_or_cpp_srcs(FileUtils::Path_t const& top_path)
141 {
142   auto srcs = FileUtils::Paths_t{};
143

```

```

144     namespace fs = std::filesystem;
145     for (fs::path const& p : fs::recursive_directory_iterator{top_path}) {
146         if (fs::is_regular_file(p) && is_c_or_cpp(p.extension())) {
147             srcs.emplace_back(FileUtils::NormalizeLexically(p));
148         }
149     }
150
151     return srcs;
152 }
153 } // namespace
154
155 std::pair<FileUtils::Paths_t, FileUtils::Paths_t> GetCppDirsSrcs(FileUtils::Paths_t const& dirs)
156 {
157     auto dirs_srcs = FileUtils::Paths_t{};
158     auto srcs      = FileUtils::Paths_t{};
159
160     for (auto const& dir : dirs) {
161         FileUtils::Path_t const top_path      = FileUtils::NormalizeLexically(dir);
162         auto                 sub_srcs       = find_c_or_cpp_srcs(top_path);
163         auto                 sub_dirs_srcs = gen_dirs(top_path, sub_srcs);
164
165         Nstd::Concatenate(srcs, std::move(sub_srcs));
166         Nstd::Concatenate(dirs_srcs, std::move(sub_dirs_srcs));
167     }
168
169     Nstd::SortUnique(srcs);
170     Nstd::SortUnique(dirs_srcs);
171
172     return {std::move(dirs_srcs), std::move(srcs)};
173 }
174 } // namespace Dependency

```

exampledeps/dependency/src/cpp_src.h

```

1 #pragma once
2 #include <string>
3 #include <utility>
4 #include <vector>
5
6 #include "file_utils/path_utils.h"
7
8 namespace Dependency {
9
10 class CppSrc {
11 public:
12     explicit CppSrc(FileUtils::Path_t const& pathname, FileUtils::Filename2Path_t const& db);
13     FileUtils::Paths_t const& GetIncs() const noexcept { return incs_; }
14     FileUtils::Paths_t const& GetIncsNotFound() const noexcept { return not_found_; }
15     FileUtils::Path_t const& Filename() const noexcept { return filename_; }
16     FileUtils::Path_t const& Path() const noexcept { return path_; }
17
18 private:
19     FileUtils::Path_t const path_;
20     FileUtils::Path_t const filename_;
21     FileUtils::Paths_t      incs_;
22     FileUtils::Paths_t      not_found_;
23
24     friend bool operator==(CppSrc const& lhs, CppSrc const& rhs) noexcept;
25     friend bool operator<(CppSrc const& lhs, CppSrc const& rhs) noexcept;
26 };
27
28 inline bool operator!=(CppSrc const& lhs, CppSrc const& rhs) noexcept { return !(lhs == rhs); }
29 inline bool operator>(CppSrc const& lhs, CppSrc const& rhs) noexcept
30 {
31     if (lhs < rhs) {
32         return false;
33     }
34
35     return lhs != rhs;
36 }
37
38 using CppSrcs_t = std::vector<CppSrc>;
39 CppSrcs_t    GenCppSrc(FileUtils::Paths_t const& srcs, FileUtils::Filename2Path_t const& db);
40 std::string ToStringCppSrc(CppSrc const& cpp_src);
41 inline std::ostream& operator<<(std::ostream& os, CppSrc const& cpp_src)
42 {
43     return os << ToStringCppSrc(cpp_src);
44 }
45
46 // first dirs配下のソースファイルを含むディレクトリ

```

```
47 // second dirs配下のソースファイル
48 std::pair<FileUtils::Paths_t, FileUtils::Paths_t> GetCppDirsSrcs(FileUtils::Paths_t const& dirs);
49 } // namespace Dependency
```

exampledepsdependencysrcdeps_scenario.cpp

```
1 #include <cassert>
2 #include <iostream>
3 #include <regex>
4 #include <stdexcept>
5
6 #include "arch_pkg.h" // 実装用ヘッダファイル
7 // @@@ sample begin 0:0
8
9 #include "cpp_deps.h" // 実装用ヘッダファイル
10 #include "cpp_dir.h" // 実装用ヘッダファイル
11 #include "cpp_src.h" // 実装用ヘッダファイル
12 #include "dependency/deps_scenario.h" // dependencyパッケージからのインポート
13 #include "file_utils/load_store.h" // file_utilsパッケージからのインポート
14 #include "lib/nstd.h" // libパッケージからのインポート
15 // @@@ sample end
16 #include "load_store_format.h"
17
18 namespace Dependency {
19 namespace {
20
21 bool has_error_for_dir(FileUtils::Paths_t const& dirs)
22 {
23     if (dirs.size() == 0) {
24         throw std::runtime_error{"need directories to generate package"};
25     }
26
27     auto not_dirs = FileUtils::NotDirs(dirs);
28
29     if (not_dirs.size() != 0) {
30         throw std::runtime_error{FileUtils::ToStringPaths(not_dirs) + " not directory"};
31     }
32
33     return false;
34 }
35
36 FileUtils::Paths_t remove_dirs_match_pattern(FileUtils::Paths_t& dirs, std::string const& pattern)
37 {
38     if (pattern.size() == 0) {
39         return std::move(dirs);
40     }
41
42     auto const re_pattern = std::regex{pattern};
43
44     dirs.remove_if([&re_pattern](auto const& d) {
45         auto results = std::smatch{};
46         auto d_str = d.string();
47         return std::regex_match(d_str, results, re_pattern);
48    });
49
50     return std::move(dirs);
51 }
52
53 // first dirs配下のソースファイルを含むディレクトリ(パッケージ)
54 // second 上記パッケージに含まれるソースファイル
55 std::pair<FileUtils::Paths_t, FileUtils::Dirs2Srcs_t> gen_dirs_and_dirs2srcs(
56     FileUtils::Paths_t const& dirs, bool recursive, std::string const& pattern)
57 {
58     auto ret = std::pair<FileUtils::Paths_t, FileUtils::Paths_t>{GetCppDirsSrcs(dirs)};
59     auto srcs = FileUtils::Paths_t{std::move(ret.second)};
60     auto dirs_pkg = FileUtils::Paths_t{recursive ? std::move(ret.first) : dirs};
61
62     dirs_pkg = remove_dirs_match_pattern(std::move(dirs_pkg), pattern);
63
64     auto dirs2srcs = FileUtils::Dirs2Srcs_t{FileUtils::AssginSrcsToDirs(dirs_pkg, srcs)};
65
66     return {std::move(dirs_pkg), std::move(dirs2srcs)};
67 }
68
69 FileUtils::Paths_t gen_dirs(FileUtils::Paths_t const& dirs, bool recursive,
70                             std::string const& pattern)
71 {
72     auto dirs2srcs = std::pair<FileUtils::Paths_t, FileUtils::Dirs2Srcs_t>{
73         gen_dirs_and_dirs2srcs(dirs, recursive, pattern)};
74 }
```

```

75     auto dirs_pkg = FileUtils::Paths_t{std::move(dirs2srcs.first)};
76     auto assign   = FileUtils::Dirs2Srcs_t{std::move(dirs2srcs.second)};
77
78     auto ret = FileUtils::Paths_t{};
79     for (auto& dir : dirs_pkg) {
80         if (assign.count(dir) == 0) {
81             std::cout << dir << " not including C++ files" << std::endl;
82         }
83         else {
84             ret.emplace_back(std::move(dir));
85         }
86     }
87
88     return ret;
89 }
90
91 FileUtils::Paths_t gen_dirs(std::string const& in, bool recursive,
92                             FileUtils::Paths_t const& dirs_opt, std::string const& pattern)
93 {
94     auto dirs = FileUtils::Paths_t{};
95
96     if (in.size() != 0) {
97         auto ret = std::optional<FileUtils::Paths_t>{FileUtils::LoadFromFile(in, Load_Paths)};
98         if (!ret) {
99             throw std::runtime_error{in + " is illegal"};
100        }
101        dirs = std::move(*ret);
102    }
103
104    Nstd::Concatenate(dirs, FileUtils::Paths_t(dirs_opt));
105
106    if (has_error_for_dir(dirs)) {
107        return dirs;
108    }
109
110    return gen_dirs(dirs, recursive, pattern);
111 }
112
113 bool includes(FileUtils::Paths_t const& dirs, FileUtils::Path_t const& dir) noexcept
114 {
115     auto const count
116         = std::count_if(dirs.cbegin(), dirs.cend(),
117                         [&dir](auto const& dir_in_dirs) noexcept { return dir_in_dirs == dir; });
118
119     return count != 0;
120 }
121
122 FileUtils::Dirs2Srcs_t dirs2srcs_to_src2src(FileUtils::Paths_t const& dirs_opt,
123                                              FileUtils::Dirs2Srcs_t const dirs2srcs, bool recursive)
124 {
125     auto ret = FileUtils::Dirs2Srcs_t{};
126
127     for (auto const& pair : dirs2srcs) {
128         for (auto const& src : pair.second) {
129             if (recursive) {
130                 ret.insert(std::make_pair(src.filename(), FileUtils::Paths_t{src}));
131             }
132             else {
133                 if (includes(dirs_opt, pair.first)) {
134                     auto dir = FileUtils::NormalizeLexically(src.parent_path());
135
136                     if (dir == pair.first) {
137                         ret.insert(std::make_pair(src.filename(), FileUtils::Paths_t{src}));
138                     }
139                 }
140             }
141         }
142     }
143
144     return ret;
145 }
146
147 FileUtils::Dirs2Srcs_t gen_dirs2srcs(std::string const& in, bool recursive, bool src_as_pkg,
148                                     FileUtils::Paths_t const& dirs_opt, std::string const& pattern)
149 {
150     auto dirs2srcs = FileUtils::Dirs2Srcs_t{};
151     auto dirs      = FileUtils::Paths_t{};
152
153     if (in.size() != 0) {
154         using FileUtils::LoadFromFile;

```

```

155     auto ret = std::optional<FileUtils::Dirs2Srcs_t>{LoadFromFile(in, Load_Dirs2Srcs)};
156
157     if (ret) {
158         if (dirs_opt.size() != 0) {
159             std::cout << "DIRS ignored." << std::endl;
160         }
161
162         if (recursive) {
163             std::cout << "option \"recursive\" ignored." << std::endl;
164         }
165         return std::move(*ret);
166     }
167     else {
168         auto ret = std::optional<FileUtils::Paths_t>{LoadFromFile(in, Load_Paths)};
169
170         if (!ret) {
171             throw std::runtime_error{in + " is illegal"};
172         }
173         dirs = std::move(*ret);
174     }
175 }
176
177 Nstd::Concatenate(dirs, FileUtils::Paths_t(dirs_opt));
178
179 if (has_error_for_dir(dirs)) {
180     return dirs2srcs;
181 }
182
183 std::pair<FileUtils::Paths_t, FileUtils::Dirs2Srcs_t> ret
184     = gen_dirs_and_dirs2srcs(dirs, recursive, pattern);
185
186 auto dirs_pkg = FileUtils::Paths_t{std::move(ret.first)};
187 auto assign   = FileUtils::Dirs2Srcs_t{std::move(ret.second)};
188
189 return src_as_pkg ? dirs2srcs_to_src2src(dirs_opt, assign, recursive) : assign;
190 }
191
192 FileUtils::Filename2Path_t gen_src_db(FileUtils::Dirs2Srcs_t const& dir2srcs)
193 {
194     auto srcs = FileUtils::Paths_t{};
195
196     for (auto const& pair : dir2srcs) {
197         auto s = pair.second;
198         Nstd::Concatenate(srcs, std::move(s));
199     }
200
201     return FileUtils::GenFilename2Path(srcs);
202 }
203 } // namespace
204
205 PkgGenerator::PkgGenerator(std::string const& in, bool recursive,
206                             FileUtils::Paths_t const& dirs_opt, std::string const& pattern)
207     : dirs_{gen_dirs(in, recursive, dirs_opt, pattern)}
208 {
209 }
210
211 bool PkgGenerator::Output(std::ostream& os) const
212 {
213     StoreToStream(os, dirs_);
214
215     return true;
216 }
217
218 SrcsGenerator::SrcsGenerator(std::string const& in, bool recursive,
219                               FileUtils::Paths_t const& dirs_opt, std::string const& pattern)
220     : dirs_{gen_dirs(in, recursive, dirs_opt, pattern)}
221 {
222 }
223
224 bool SrcsGenerator::Output(std::ostream& os) const
225 {
226     auto      ret  = std::pair<FileUtils::Paths_t, FileUtils::Paths_t>{GetCppDirsSrcs(dirs_)};
227     auto      dirs = FileUtils::Paths_t{std::move(ret.first)};
228     auto      srcs = FileUtils::Paths_t{std::move(ret.second)};
229     auto const db   = FileUtils::GenFilename2Path(srcs);
230
231     auto cpp_dirs = CppDirs_t{GenCppDirs(srcs, db)};
232
233     for (auto const& d : cpp_dirs) {
234         os << "---" << std::endl;

```

```

235     os << d << std::endl;
236 }
237
238     return true;
239 }
240
241 Pkg2SrcsGenerator::Pkg2SrcsGenerator(std::string const& in, bool recursive, bool src_as_pkg,
242                                     FileUtils::Paths_t const& dirs_opt, std::string const& pattern)
243 : dirs2srcs_{gen_dirs2srcs(in, recursive, src_as_pkg, dirs_opt, pattern)}
244 {
245 }
246
247 bool Pkg2SrcsGenerator::Output(std::ostream& os) const
248 {
249     StoreToStream(os, dirs2srcs_);
250
251     return true;
252 }
253
254 Pkg2PkgGenerator::Pkg2PkgGenerator(std::string const& in, bool recursive, bool src_as_pkg,
255                                     FileUtils::Paths_t const& dirs_opt, std::string const& pattern)
256 : dirs2srcs_{gen_dirs2srcs(in, recursive, src_as_pkg, dirs_opt, pattern)}
257 {
258 }
259
260 bool Pkg2PkgGenerator::Output(std::ostream& os) const
261 {
262     auto cpp_dirs = CppDirs_t{};
263
264     auto const db = gen_src_db(dirs2srcs_);
265
266     for (auto const& pair : dirs2srcs_) {
267         cpp_dirs.emplace_back(CppDir{pair.first, GenCppSrc(pair.second, db)});
268     }
269
270     DepRels_t const dep_rels = GenDepRels(cpp_dirs);
271
272     StoreToStream(os, dep_rels);
273
274     return true;
275 }
276
277 namespace {
278 DepRels_t gen_dep_rel(std::string const& in)
279 {
280     if (in.size() == 0) {
281         throw std::runtime_error("IN-file needed");
282     }
283
284     auto ret = std::optional<DepRels_t>{FileUtils::LoadFromFile(in, Load_DepRels)};
285
286     if (!ret) {
287         throw std::runtime_error("IN-file load error");
288     }
289
290     return *ret;
291 }
292 } // namespace
293
294 struct ArchGenerator::Impl {
295     Impl(DepRels_t&& a_dep_rels) : dep_rels(std::move(a_dep_rels)), arch(ArchPkg::GenArch(dep_rels))
296     {
297     }
298     DepRels_t const dep_rels;
299     Arch_t const      arch;
300 };
301
302 ArchGenerator::ArchGenerator(std::string const& in)
303 : impl_{std::make_unique<ArchGenerator::Impl>(gen_dep_rel(in))}
304 {
305 }
306
307 bool ArchGenerator::Output(std::ostream& os) const
308 {
309     StoreToStream(os, impl_->arch);
310
311     return true;
312 }
313 ArchGenerator::~ArchGenerator() {}
314

```

```

315 Arch2PUmlGenerator::Arch2PUmlGenerator(std::string const& in) : ArchGenerator{in} {}
316
317 bool Arch2PUmlGenerator::Output(std::ostream& os) const
318 {
319     os << "@startuml" << std::endl;
320     os << "scale max 730 width" << std::endl; // これ以上大きいとpdfにした時に右端が切れる
321
322     os << ToPlantUML_Rectangle(impl_->arch) << std::endl;
323     os << std::endl;
324
325     os << ToPlantUML_Rectangle(impl_->arch, impl_->dep_rels) << std::endl;
326     os << std::endl;
327
328     os << "@enduml" << std::endl;
329
330     return true;
331 }
332
333 CyclicGenerator::CyclicGenerator(std::string const& in)
334     : ArchGenerator{in}, has_cyclic_dep_{HasCyclicDeps(impl_->arch, impl_->dep_rels)}
335 {
336 }
337
338 bool CyclicGenerator::Output(std::ostream& os) const
339 {
340     os << "cyclic dependencies " << (has_cyclic_dep_ ? "" : "not ") << "found" << std::endl;
341
342     return !has_cyclic_dep_;
343 }
344 } // namespace Dependency

```

exampledeps/dependency/src/load_store_format.cpp

```

1 #include <cassert>
2 #include <iostream>
3 #include <regex>
4
5 #include "file_utils/load_store.h"
6 #include "load_store_format.h"
7
8 namespace Dependency {
9 namespace {
10 auto const file_format_dir2srcs = std::string_view{"#dir2srcs"};
11 auto const file_format_dir      = std::string_view{"#dir"};
12 auto const file_format_deps    = std::string_view{"#deps"};
13 auto const file_format_arch   = std::string_view{"#arch"};
14 } // namespace
15
16 bool StoreToStream(std::ostream& os, FileUtils::Paths_t const& paths)
17 {
18     os << file_format_dir << std::endl;
19
20     using FileUtils::operator<<;
21     os << paths << std::endl;
22
23     return true;
24 }
25
26 bool StoreToStream(std::ostream& os, FileUtils::Dirs2Srcs_t const& dirs2srcs)
27 {
28     os << file_format_dir2srcs << std::endl;
29
30     using FileUtils::operator<<;
31     os << dirs2srcs << std::endl;
32
33     return true;
34 }
35
36 namespace {
37
38 bool is_format_dirs2srcs(std::istream& is)
39 {
40     auto line = std::string{};
41
42     if (std::getline(is, line)) {
43         if (line == file_format_dir2srcs) {
44             return true;
45         }
46     }
47

```

```

48     return false;
49 }
50
51 FileUtils::Dirs2Srcs_t load_Dirs2Srcs_t(std::istream& is)
52 {
53     static auto const line_sep = std::regex{R"(^\s*$)"};
54     static auto const line_dir = std::regex{R"^(?([^\w/.]+)$)"};
55     static auto const line_src = std::regex{R"^(?(\s+([\w/.]+))$)"};
56
57     auto line      = std::string{};
58     auto dir       = FileUtils::Path_t{};
59     auto srcs      = FileUtils::Paths_t{};
60     auto dirs2srcs = FileUtils::Dirs2Srcs_t{};
61
62     while (std::getline(is, line)) {
63         if (auto results = std::smatch{}; std::regex_match(line, results, line_sep)) {
64             dirs2srcs[dir].swap(srcs);
65         }
66         else if (std::regex_match(line, results, line_dir)) {
67             dir = results[1].str();
68         }
69         else if (std::regex_match(line, results, line_src)) {
70             srcs.push_back(results[1].str());
71         }
72         else {
73             std::cout << line << std::endl;
74             assert(false);
75         }
76     }
77
78     return dirs2srcs;
79 }
80 } // namespace
81
82 std::optional<FileUtils::Dirs2Srcs_t> Load_Dirs2Srcs(std::istream& is)
83 {
84     auto dirs2srcs = FileUtils::Dirs2Srcs_t{};
85
86     if (!is) {
87         return std::nullopt;
88     }
89
90     if (!is_format_dirs2srcs(is)) {
91         return std::nullopt;
92     }
93
94     return load_Dirs2Srcs_t(is);
95 }
96
97 namespace {
98
99 bool is_format_dirs(std::istream& is)
100 {
101     auto line = std::string{};
102
103     if (std::getline(is, line)) {
104         if (line == file_format_dir) {
105             return true;
106         }
107     }
108
109     return false;
110 }
111 } // namespace
112
113 std::optional<FileUtils::Paths_t> Load_Paths(std::istream& is)
114 {
115     auto paths = FileUtils::Paths_t{};
116
117     if (!is_format_dirs(is)) {
118         return std::nullopt;
119     }
120
121     auto line = std::string{};
122     while (std::getline(is, line)) {
123         paths.emplace_back(FileUtils::Path_t(line));
124     }
125
126     return paths;
127 }

```

```

128
129 bool StoreToStream(std::ostream& os, DepRel_t const& dep_rels)
130 {
131     os << file_format_deps << std::endl;
132     os << dep_rels << std::endl;
133
134     return true;
135 }
136
137 namespace {
138
139 bool is_format_deps(std::istream& is)
140 {
141     auto line = std::string{};
142
143     if (std::getline(is, line)) {
144         if (line == file_format_deps) {
145             return true;
146         }
147     }
148
149     return false;
150 }
151
152 struct dep_half_t {
153     bool           valid{false};
154     std::string    from{};
155     std::string    to{};
156     uint32_t       count{0};
157     FileUtils::Paths_t headers{};
158 };
159
160 FileUtils::Paths_t gen_paths(std::string const& paths_str)
161 {
162     auto const sep = std::regex{R"( +)"};
163     auto      ret = FileUtils::Paths_t{};
164
165     if (paths_str.size() != 0) {
166         auto end = std::sregex_token_iterator{};
167         for (auto it = std::sregex_token_iterator{paths_str.begin(), paths_str.end(), sep, -1};
168              it != end; ++it) {
169             ret.emplace_back(it->str());
170         }
171     }
172
173     return ret;
174 }
175
176 dep_half_t get_dep_half(std::smatch const& results)
177 {
178     auto dep_half = dep_half_t{};
179
180     dep_half.valid = true;
181     dep_half.from = results[1].str();
182     dep_half.to = results[2].str();
183     dep_half.count = std::stoi(results[3].str());
184     dep_half.headers = gen_paths(results[4].str());
185
186     return dep_half;
187 }
188
189 DepRelation gen_dep_rel(dep_half_t& first, dep_half_t& second)
190 {
191     assert(first.valid);
192     assert(second.valid);
193     assert(first.from < second.from);
194
195     return DepRelation{first.from, first.count, std::move(first.headers),
196                       second.from, second.count, std::move(second.headers)};
197 }
198
199 DepRel_t load_DepRelations_t(std::istream& is)
200 {
201     static auto const line_sep = std::regex{R"(\s+$)"};
202     static auto const line_dep = std::regex{R"(([\w\.-]+) -> ([\w\.-]+) : ([\d]+) \.*(.*)$)"};
203
204     auto line = std::string{};
205     auto first = dep_half_t{};
206     auto second = dep_half_t{};
207

```

```

208     auto dep_rels = DepRels_t{};
209
210     while (std::getline(is, line)) {
211         if (auto results = std::smatch{}; std::regex_match(line, results, line_sep)) {
212             dep_rels.emplace_back(gen_dep_rel(std::move(first), std::move(second)));
213
214             first.valid = false;
215             second.valid = false;
216         }
217         else if (std::regex_match(line, results, line_dep)) {
218             (!first.valid ? first : second) = get_dep_half(results);
219         }
220         else {
221             assert(false);
222         }
223     }
224
225     return dep_rels;
226 }
227 } // namespace
228
229 std::optional<DepRels_t> Load_DepRels(std::istream& is)
230 {
231     if (!is) {
232         return std::nullopt;
233     }
234
235     if (!is_format_deps(is)) {
236         return std::nullopt;
237     }
238
239     return load_DepRelations_t(is);
240 }
241
242 bool StoreToStream(std::ostream& os, Arch_t const& arch)
243 {
244     os << file_format_arch << std::endl;
245     os << arch << std::endl;
246
247     return true;
248 }
249 } // namespace Dependency

```

exampledeps/dependency/src/load_store_format.h

```

1 #pragma once
2 #include <optional>
3 #include <utility>
4
5 #include "arch_pkg.h"
6 #include "cpp_deps.h"
7 #include "file_utils/path_utils.h"
8
9 namespace Dependency {
10
11 // LoadStore
12 bool StoreToStream(std::ostream& os, FileUtils::Paths_t const& paths);
13 std::optional<FileUtils::Paths_t> Load_Paths(std::istream& is);
14
15 // Dirs2Srcs_t
16 bool StoreToStream(std::ostream& os, FileUtils::Dirs2Srcs_t const& dirs2srcs);
17 std::optional<FileUtils::Dirs2Srcs_t> Load_Dirs2Srcs(std::istream& is);
18
19 // DepRels_t
20 bool StoreToStream(std::ostream& os, DepRels_t const& dep_rels);
21 std::optional<DepRels_t> Load_DepRels(std::istream& is);
22
23 // Arch_t
24 bool StoreToStream(std::ostream& os, Arch_t const& arch);
25 } // namespace Dependency

```

exampledeps/dependency/ut/arch_pkg_ut.cpp

```

1 #include "gtest_wrapper.h"
2
3 #include "arch_pkg.h"
4
5 namespace Dependency {
6 namespace {

```

```

7
8 using FileUtils::Paths_t;
9
10 DepRels_t const dep_rels_simple{
11     {DepRelation{"A", 1, Paths_t{"b.h"}, "B", 0, Paths_t{}},,
12 };
13
14 DepRels_t const dep_rels_simple2{
15     {DepRelation{"X", 1, Paths_t{"b"}, "X/A", 0, Paths_t{}},,
16     {DepRelation{"X", 1, Paths_t{"c"}, "X/B", 0, Paths_t{}},,
17     {DepRelation{"X", 1, Paths_t{"d"}, "X/C", 0, Paths_t{}},,
18     {DepRelation{"X", 0, Paths_t{}, "X/D", 0, Paths_t{}},,
19     {DepRelation{"X", 0, Paths_t{}, "X/E", 0, Paths_t{}},,
20
21     {DepRelation{"X/A", 1, Paths_t{"b"}, "X/B", 0, Paths_t{}},,
22     {DepRelation{"X/B", 1, Paths_t{"c"}, "X/C", 0, Paths_t{}},,
23     {DepRelation{"X/C", 1, Paths_t{"d"}, "X/D", 0, Paths_t{}},,
24     {DepRelation{"X/A", 0, Paths_t{"a"}, "X/D", 1, Paths_t{}},,
25     {DepRelation{"X/A", 1, Paths_t{"a"}, "X/E", 1, Paths_t{"d"}},,
26 };
27
28 DepRels_t const dep_rels_simple3{
29     // A -> B
30     // A -> C -> D -> A
31     //           C -> B
32     {DepRelation{"A", 1, Paths_t{}, "B", 0, Paths_t{}},,
33     {DepRelation{"A", 1, Paths_t{}, "C", 0, Paths_t{"a"}},,
34     {DepRelation{"A", 0, Paths_t{}, "D", 1, Paths_t{"a"}},,
35
36     {DepRelation{"B", 0, Paths_t{}, "C", 1, Paths_t{"b"}},,
37     {DepRelation{"B", 0, Paths_t{}, "D", 0, Paths_t{}},,
38     {DepRelation{"C", 1, Paths_t{"d"}, "D", 0, Paths_t{}},,
39 };
40
41 DepRels_t const dep_rels_middle{
42     {DepRelation{"ut_data/app1/mod2", 0, Paths_t{}, "ut_data/app1/mod2/mod2_1", 0, Paths_t{}},,
43     {DepRelation{"ut_data/app1/mod2", 0, Paths_t{}, "ut_data/app1/mod2/mod2_2", 0, Paths_t{}},,
44     {DepRelation{"ut_data/app1/mod2/mod2_1", 1, Paths_t{"ut_data/app1/mod2/mod2_2/mod2_2_1.h"},,
45         "ut_data/app1/mod2/mod2_2", 2, Paths_t{"ut_data/app1/mod2/mod2_1/mod2_1_1.h"}},,
46 };
47
48 DepRels_t const dep_rels_complex{
49     {DepRelation{"ut_data/app1", 2,
50         Paths_t{"ut_data/app1/mod1/mod1_1.hpp", "ut_data/app1/mod1/mod1_2.hpp"},,
51         "ut_data/app1/mod1", 0, Paths_t{}},,
52     {DepRelation{"ut_data/app1", 0, Paths_t{}, "ut_data/app1/mod2", 0, Paths_t{}},,
53     {DepRelation{"ut_data/app1", 0, Paths_t{}, "ut_data/app1/mod2/mod2_1", 0, Paths_t{}},,
54     {DepRelation{"ut_data/app1", 0, Paths_t{}, "ut_data/app1/mod2/mod2_2", 1,
55         Paths_t{"ut_data/app1/a_1.cpp.h"}},,
56     {DepRelation{"ut_data/app1", 0, Paths_t{}, "ut_data/app2", 1,
57         Paths_t{"ut_data/app1/a_2.cpp.hpp"}},,
58     {DepRelation{"ut_data/app1/mod1", 1, Paths_t{"ut_data/app1/mod2/mod2_1.hpp"},,
59         "ut_data/app1/mod2", 0, Paths_t{}},,
60     {DepRelation{"ut_data/app1/mod1", 0, Paths_t{}, "ut_data/app1/mod2/mod2_1", 0, Paths_t{}},,
61     {DepRelation{"ut_data/app1/mod1", 1, Paths_t{"ut_data/app1/mod2/mod2_2/mod2_2_1.h"},,
62         "ut_data/app1/mod2/mod2_2", 0, Paths_t{}},,
63     {DepRelation{"ut_data/app1/mod1", 0, Paths_t{}, "ut_data/app2", 2,
64         Paths_t{"ut_data/app1/mod1_1.hpp", "ut_data/app1/mod1/mod1_2.hpp"}},,
65     {DepRelation{"ut_data/app1/mod2", 0, Paths_t{}, "ut_data/app1/mod2/mod2_1", 0, Paths_t{}},,
66     {DepRelation{"ut_data/app1/mod2", 0, Paths_t{}, "ut_data/app1/mod2/mod2_2", 0, Paths_t{}},,
67     {DepRelation{"ut_data/app1/mod2", 0, Paths_t{}, "ut_data/app2", 0, Paths_t{}},,
68     {DepRelation{"ut_data/app1/mod2/mod2_1", 1, Paths_t{"ut_data/app1/mod2/mod2_2/mod2_2_1.h"},,
69         "ut_data/app1/mod2/mod2_2", 2, Paths_t{"ut_data/app1/mod2/mod2_1/mod2_1_1.h"}},,
70     {DepRelation{"ut_data/app1/mod2/mod2_1", 0, Paths_t{}, "ut_data/app2", 0, Paths_t{}},,
71     {DepRelation{"ut_data/app1/mod2/mod2_2", 0, Paths_t{}, "ut_data/app2", 0, Paths_t{}},,
72 };
73
74 TEST(arch_pkg, ArchPkgSimple)
75 {
76     auto const arch = ArchPkg::GenArch(dep_rels_simple);
77
78     ASSERT_EQ(2, arch.size());
79
80     auto const a = *arch.cbegin();
81     ASSERT_EQ("A", a->Name());
82     ASSERT_EQ(nullptr, a->Parent());
83     ASSERT_EQ("B", a->DependOn().front()->Name());
84     ASSERT_FALSE(a->IsCyclic());
85     ASSERT_FALSE(a->IsCyclic(*a->DependOn().front()));
86

```

```

87     auto const& children = a->Children();
88     ASSERT_EQ(0, children.size());
89
90     auto const& b = *std::next(arch.cbegin());
91     ASSERT_EQ("B", b->Name());
92     ASSERT_EQ(nullptr, b->Parent());
93
94     ASSERT_EQ(0, b->DependOn().size());
95     ASSERT_FALSE(b->IsCyclic());
96 }
97
98 TEST(arch_pkg, ArchPkgSimple2)
99 {
100     auto const arch = ArchPkg::GenArch(deps_simple2);
101
102     auto exp = std::string{
103         "package :X\n"
104         "fullname :X\n"
105         "parent :TOP\n"
106         "depend_on: {\n"
107             "    A : STRAIGHT\n"
108             "    B : STRAIGHT\n"
109             "    C : STRAIGHT\n"
110         "}\n"
111         "children : {\n"
112             "    package :A:CYCLIC\n"
113             "    fullname :X/A\n"
114             "    parent :X\n"
115             "    depend_on: {\n"
116                 "        B : CYCLIC\n"
117                 "        E : CYCLIC\n"
118             "}\n"
119             "    children : { }\n"
120         "}\n"
121         "    package :B:CYCLIC\n"
122         "    fullname :X/B\n"
123         "    parent :X\n"
124         "    depend_on: {\n"
125             "        C : CYCLIC\n"
126         "}\n"
127         "    children : { }\n"
128         "\n"
129         "    package :C:CYCLIC\n"
130         "    fullname :X/C\n"
131         "    parent :X\n"
132         "    depend_on: {\n"
133             "        D : CYCLIC\n"
134         "}\n"
135         "    children : { }\n"
136         "\n"
137         "    package :D:CYCLIC\n"
138         "    fullname :X/D\n"
139         "    parent :X\n"
140         "    depend_on: {\n"
141             "        A : CYCLIC\n"
142         "}\n"
143         "    children : { }\n"
144         "\n"
145         "    package :E:CYCLIC\n"
146         "    fullname :X/E\n"
147         "    parent :X\n"
148         "    depend_on: {\n"
149             "        A : CYCLIC\n"
150         "}\n"
151         "    children : { }\n"
152     "}";
153
154     ASSERT_EQ(exp, ToStringArch(arch));
155 }
156
157 TEST(arch_pkg, ArchPkgSimple3)
158 {
159     auto const arch = ArchPkg::GenArch(deps_simple3);
160
161     auto exp = std::string{
162         "package :A:CYCLIC\n"
163         "fullname :A\n"
164         "parent :TOP\n"
165         "depend_on: {\n"
166             "    B : STRAIGHT\n"

```

```

167     "      C : CYCLIC\n"
168     "}\n"
169     "children : { }\n"
170     "\n"
171     "package :B\n"
172     "fullname :B\n"
173     "parent :TOP\n"
174     "depend_on: { }\n"
175     "children : { }\n"
176     "\n"
177     "package :C:CYCLIC\n"
178     "fullname :C\n"
179     "parent :TOP\n"
180     "depend_on: {\n"
181     "    B : STRAIGHT\n"
182     "    D : CYCLIC\n"
183     "}\n"
184     "children : { }\n"
185     "\n"
186     "package :D:CYCLIC\n"
187     "fullname :D\n"
188     "parent :TOP\n"
189     "depend_on: {\n"
190     "    A : CYCLIC\n"
191     "}\n"
192     "children : { }";
193
194     ASSERT_EQ(exp, ToStringArch(arch));
195 }
196
197 TEST(arch_pkg, ArchPkg2)
198 {
199     auto const arch = ArchPkg::GenArch(dep_rels_middle);
200
201     ASSERT_EQ(1, arch.size());
202
203     Arch_t const* mod2_children(nullptr);
204     {
205         auto const& mod2 = *arch.cbegin();
206
207         ASSERT_EQ("mod2", mod2->Name());
208         ASSERT_EQ(nullptr, mod2->Parent());
209         ASSERT_EQ(0, mod2->DependOn().size());
210         ASSERT_FALSE(mod2->IsCyclic());
211
212         mod2_children = &mod2->Children();
213         ASSERT_EQ(2, mod2_children->size());
214     }
215     {
216         auto const& mod2_1 = *mod2_children->cbegin();
217         ASSERT_EQ("mod2_1", mod2_1->Name());
218         ASSERT_EQ("mod2", mod2_1->Parent()->Name());
219         ASSERT_EQ("mod2_2", mod2_1->DependOn().front()->Name());
220         ASSERT_TRUE(mod2_1->IsCyclic());
221         ASSERT_TRUE(mod2_1->IsCyclic(*mod2_1->DependOn().front()));
222
223         auto const& children = mod2_1->Children();
224         ASSERT_EQ(0, children.size());
225     }
226     {
227         auto const& mod2_2 = *std::next(mod2_children->cbegin());
228         ASSERT_EQ("mod2_2", mod2_2->Name());
229         ASSERT_EQ("mod2", mod2_2->Parent()->Name());
230         ASSERT_EQ("mod2_1", mod2_2->DependOn().front()->Name());
231         ASSERT_TRUE(mod2_2->IsCyclic());
232         ASSERT_TRUE(mod2_2->IsCyclic(*mod2_2->DependOn().front()));
233
234         auto const& children = mod2_2->Children();
235         ASSERT_EQ(0, children.size());
236     }
237 }
238
239 TEST(arch_pkg, ArchPkg3)
240 {
241     auto const arch = ArchPkg::GenArch(dep_rels_complex);
242
243     /* std::cout << ToStringArch(arch) << std::endl;
244
245     package :app1:CYCLIC
246     parent :TOP

```

```

247     depend_on: {
248         mod1
249     }
250     children : {
251         package :mod1:CYCLIC
252         parent :app1
253         depend_on: {
254             mod2
255             mod2_2
256         }
257
258         package :mod2
259         parent :app1
260         children : {
261             package :mod2_1:CYCLIC
262             parent :mod2
263             depend_on: {
264                 mod2_2
265             }
266
267             package :mod2_2:CYCLIC
268             parent :mod2
269             depend_on: {
270                 app1
271                 mod2_1
272             }
273         }
274     }
275     package :app2
276     parent :TOP
277     depend_on: {
278         app1
279         mod1
280     }
281 */
282
283 {
284     Arch_t const* app1_children(nullptr);
285     {
286         auto const& app1 = *arch.cbegin();
287
288         ASSERT_EQ("app1", app1->Name());
289         ASSERT_EQ(nullptr, app1->Parent());
290         ASSERT_EQ(1, app1->DependOn().size());
291         {
292             auto const& depend = app1->DependOn();
293
294             ASSERT_EQ("mod1", (*depend.cbegin())->Name());
295             ASSERT_TRUE(app1->IsCyclic(*(*depend.cbegin())));
296         }
297
298         ASSERT_TRUE(app1->IsCyclic());
299
300         app1_children = &app1->Children();
301         ASSERT_EQ(2, app1_children->size());
302     }
303     {
304         {
305             auto const& mod1 = *app1_children->cbegin();
306             ASSERT_EQ("mod1", mod1->Name());
307             ASSERT_EQ("app1", mod1->Parent()->Name());
308             ASSERT_EQ(2, mod1->DependOn().size());
309             {
310                 auto const& depend = mod1->DependOn();
311
312                 ASSERT_EQ("mod2", (*depend.cbegin())->Name());
313                 ASSERT_FALSE(mod1->IsCyclic(*(*depend.cbegin())));
314
315                 auto const next = *std::next(depend.cbegin());
316                 ASSERT_EQ("mod2_2", next->Name());
317                 ASSERT_TRUE(mod1->IsCyclic(*next));
318             }
319             ASSERT_TRUE(mod1->IsCyclic());
320         }
321         Arch_t const* mod2_children(nullptr);
322         {
323             auto const& mod2 = *std::next(app1_children->cbegin());
324             ASSERT_EQ("mod2", mod2->Name());
325             ASSERT_EQ("app1", mod2->Parent()->Name());
326             ASSERT_EQ(0, mod2->DependOn().size());

```

```

327         mod2_children = &mod2->Children();
328         ASSERT_EQ(2, mod2_children->size());
329
330         ASSERT_FALSE(mod2->IsCyclic());
331     }
332     {
333         {
334             auto const& mod2_1 = *mod2_children->cbegin();
335             ASSERT_EQ("mod2_1", mod2_1->Name());
336
337             ASSERT_EQ("mod2", mod2_1->Parent()->Name());
338             ASSERT_EQ(1, mod2_1->DependOn().size());
339             {
340                 auto const& depend = mod2_1->DependOn();
341                 ASSERT_EQ("mod2_2", (*depend.cbegin())->Name());
342                 ASSERT_TRUE(mod2_1->IsCyclic(*depend));
343             }
344
345             ASSERT_TRUE(mod2_1->IsCyclic());
346             ASSERT_EQ(0, mod2_1->Children().size());
347         }
348         {
349             auto const& mod2_2 = *std::next(mod2_children->cbegin());
350             ASSERT_EQ("mod2_2", mod2_2->Name());
351
352             ASSERT_EQ("mod2", mod2_2->Parent()->Name());
353             ASSERT_EQ(2, mod2_2->DependOn().size());
354             {
355                 auto const& depend = mod2_2->DependOn();
356                 ASSERT_EQ("app1", (*depend.cbegin())->Name());
357                 ASSERT_TRUE(mod2_2->IsCyclic(*depend));
358
359                 auto const next = *std::next(depend.cbegin());
360                 ASSERT_EQ("mod2_1", (*std::next(depend.cbegin())->Name()));
361                 ASSERT_TRUE(mod2_2->IsCyclic(*next));
362             }
363
364             ASSERT_TRUE(mod2_2->IsCyclic());
365             ASSERT_EQ(0, mod2_2->Children().size());
366         }
367     }
368 }
369 }
370 }
371 {
372     auto const& app2 = *std::next(arch.cbegin());
373
374     ASSERT_EQ("app2", app2->Name());
375     ASSERT_EQ(nullptr, app2->Parent());
376     ASSERT_EQ(2, app2->DependOn().size());
377     {
378         auto const& depend = app2->DependOn();
379
380         ASSERT_EQ("app1", (*depend.cbegin())->Name());
381         ASSERT_EQ("mod1", (*std::next(depend.cbegin())->Name()));
382     }
383
384     ASSERT_FALSE(app2->IsCyclic());
385     ASSERT_EQ(0, app2->Children().size());
386 }
387 }
388
389 TEST(arch_pkg, ToPlantUML_Rectangle)
390 {
391     {
392         auto const arch = ArchPkg::GenArch(dep_rels_simple);
393         auto const exp = std::string{
394             "rectangle \"A\" as A\n"
395             "rectangle \"B\" as B";
396         ASSERT_EQ(exp, ToPlantUML_Rectangle(arch));
397     }
398     {
399         auto const arch = ArchPkg::GenArch(dep_rels_middle);
400         auto const exp = std::string{
401             "rectangle \"mod2\" as ut_data_app1_mod2 {\n"
402                 "    rectangle \"mod2_1\" as ut_data_app1_mod2_mod2_1\n"
403                 "    rectangle \"mod2_2\" as ut_data_app1_mod2_mod2_2\n"
404             "}";
405         ASSERT_EQ(exp, ToPlantUML_Rectangle(arch));
406     }

```

```

407 {
408     auto const arch = ArchPkg::GenArch(dep_rels_complex);
409     auto const exp = std::string{
410         "rectangle \"app1\" as ut_data__app1 {\n"
411             "    rectangle \"mod1\" as ut_data__app1__mod1\n"
412             "    rectangle \"mod2\" as ut_data__app1__mod2 {\n"
413                 "        rectangle \"mod2_1\" as ut_data__app1__mod2__mod2_1\n"
414                 "        rectangle \"mod2_2\" as ut_data__app1__mod2__mod2_2\n"
415             "}\n"
416         "}\n"
417         "rectangle \"app2\" as ut_data__app2"};
418     ASSERT_EQ(exp, ToPlantUML_Rectangle(arch));
419 }
420 }
421
422 TEST(arch_pkg, ToPlantUML_Rectangle2)
423 {
424     auto const arch = ArchPkg::GenArch(dep_rels_complex);
425     auto const exp = std::string{
426         "ut_data__app1 \"2\" -[#orange]-> ut_data__app1__mod1\n"
427         "ut_data__app1__mod2__mod2_2 \"1\" -[#orange]-> ut_data__app1\n"
428         "ut_data__app2 \"1\" -[#green]-> ut_data__app1\n"
429         "ut_data__app1__mod1 \"1\" -[#green]-> ut_data__app1__mod2\n"
430         "ut_data__app1__mod1 \"1\" -[#orange]-> ut_data__app1__mod2__mod2_2\n"
431         "ut_data__app2 \"2\" -[#green]-> ut_data__app1__mod1\n"
432         "ut_data__app1__mod2__mod2_1 \"1\" <-[#red]-> \"2\" ut_data__app1__mod2__mod2_2"};
433
434     ASSERT_EQ(exp, ToPlantUML_Rectangle(arch, dep_rels_complex));
435 }
436
437 TEST(arch_pkg, HasCyclicDeps)
438 {
439 {
440     auto const arch = ArchPkg::GenArch(dep_rels_simple);
441     ASSERT_FALSE(HasCyclicDeps(arch, dep_rels_simple));
442 }
443 {
444     auto const arch = ArchPkg::GenArch(dep_rels_middle);
445     ASSERT_TRUE(HasCyclicDeps(arch, dep_rels_middle));
446 }
447 {
448     auto const arch = ArchPkg::GenArch(dep_rels_complex);
449     ASSERT_TRUE(HasCyclicDeps(arch, dep_rels_complex));
450 }
451 }
452
453 TEST(arch_pkg, FindArchPkg)
454 {
455     auto const arch = ArchPkg::GenArch(dep_rels_simple);
456
457 {
458     ArchPkg const* pkg_a = FindArchPkgByName(arch, "A");
459     ASSERT_NE(nullptr, pkg_a);
460     ASSERT_EQ("A", pkg_a->Name());
461 }
462 {
463     ArchPkg const* pkg_a_f = FindArchPkgByFullName(arch, "A");
464     ASSERT_NE(nullptr, pkg_a_f);
465     ASSERT_EQ("A", pkg_a_f->FullName());
466 }
467 {
468     ArchPkg const* pkg_b = FindArchPkgByName(arch, "B");
469     ASSERT_NE(nullptr, pkg_b);
470     ASSERT_EQ("B", pkg_b->Name());
471 }
472 {
473     ArchPkg const* pkg_b_f = FindArchPkgByName(arch, "B");
474
475     ASSERT_NE(nullptr, pkg_b_f);
476     ASSERT_EQ("B", pkg_b_f->FullName());
477 }
478 }
479
480 TEST(arch_pkg, FindArchPkg2)
481 {
482     auto const arch = ArchPkg::GenArch(dep_rels_simple2);
483
484 {
485     ArchPkg const* pkg_x = FindArchPkgByName(arch, "X");
486     ASSERT_NE(nullptr, pkg_x);

```

```

487     ASSERT_EQ("X", pkg_x->Name());
488 }
489 {
490     ArchPkg const* pkg_x_f = FindArchPkgByFullName(arch, "X");
491     ASSERT_NE(nullptr, pkg_x_f);
492     ASSERT_EQ("X", pkg_x_f->FullName());
493 }
494 {
495     ArchPkg const* pkg_a = FindArchPkgByName(arch, "A");
496     ASSERT_NE(nullptr, pkg_a);
497     ASSERT_EQ("A", pkg_a->Name());
498 }
499 {
500     ArchPkg const* pkg_a_f = FindArchPkgByFullName(arch, "X/A");
501     ASSERT_NE(nullptr, pkg_a_f);
502     ASSERT_EQ("X/A", pkg_a_f->FullName());
503 }
504 {
505     ArchPkg const* pkg_y = FindArchPkgByName(arch, "Y");
506     ASSERT_EQ(nullptr, pkg_y);
507 }
508 {
509     ArchPkg const* pkg_y_f = FindArchPkgByFullName(arch, "Y");
510     ASSERT_EQ(nullptr, pkg_y_f);
511 }
512 }
513
514 TEST(arch_pkg, FindArchPkg3)
515 {
516     auto const arch = ArchPkg::GenArch(dep_rels_complex);
517
518 {
519     ArchPkg const* pkg_app1 = FindArchPkgByName(arch, "app1");
520     ASSERT_NE(nullptr, pkg_app1);
521     ASSERT_EQ("app1", pkg_app1->Name());
522     ASSERT_EQ("ut_data/app1", pkg_app1->FullName());
523 }
524 {
525     ArchPkg const* pkg_app1_f = FindArchPkgByFullName(arch, "ut_data/app1");
526     ASSERT_NE(nullptr, pkg_app1_f);
527     ASSERT_EQ("app1", pkg_app1_f->Name());
528     ASSERT_EQ("ut_data/app1", pkg_app1_f->FullName());
529 }
530 {
531     ArchPkg const* pkg_mod2_1 = FindArchPkgByName(arch, "mod2_1");
532     ASSERT_NE(nullptr, pkg_mod2_1);
533     ASSERT_EQ("mod2_1", pkg_mod2_1->Name());
534     ASSERT_EQ("ut_data/app1/mod2/mod2_1", pkg_mod2_1->FullName());
535 }
536 {
537     ArchPkg const* pkg_mod2_1_f = FindArchPkgByFullName(arch, "ut_data/app1/mod2/mod2_1");
538     ASSERT_NE(nullptr, pkg_mod2_1_f);
539     ASSERT_EQ("mod2_1", pkg_mod2_1_f->Name());
540     ASSERT_EQ("ut_data/app1/mod2/mod2_1", pkg_mod2_1_f->FullName());
541 }
542 }
543 } // namespace
544 } // namespace Dependency

```

exampledeps/dependency/ut/cpp_deps_ut.cpp

```

1 #include "gtest_wrapper.h"
2
3 #include "cpp_deps.h"
4 #include "cpp_dir.h"
5 #include "cpp_src.h"
6
7 namespace Dependency {
8 namespace {
9
10 TEST(cpp_deps, GenDepRels)
11 {
12     using FileUtils::Paths_t;
13
14     auto const [dirs, srcs] = GetCppDirsSrcs({"ut_data/"});
15     auto const assign      = FileUtils::AssignSrcsToDirs(dirs, srcs);
16     auto const srcs_db    = FileUtils::GenFilename2Path(srcs);
17
18     auto cpp_dirs = CppDirs_t{};
19

```

```

20     for (auto const& pair : assign) {
21         cpp_dirs.emplace_back(CppDir{pair.first, GenCppSrc(pair.second, srcs_db)});
22     }
23
24     auto dep_all = GenDepRels(cpp_dirs);
25
26     auto const app1    = std::string("ut_data/app1");
27     auto const mod1    = std::string("ut_data/app1/mod1");
28     auto const mod2_2  = std::string("ut_data/app1/mod2/mod2_2");
29
30     {
31         auto const app1_mod1 = FindDepRels(dep_all, app1, mod1);
32         ASSERT_EQ("ut_data/app1", app1_mod1->PackageA);
33         ASSERT_EQ("ut_data/app1/mod1", app1_mod1->PackageB);
34
35         ASSERT_EQ(6, app1_mod1->CountAtob);
36         ASSERT_EQ(1, app1_mod1->CountBtoA);
37
38         auto const app1_mod1_IncsAtob
39             = Paths_t{"ut_data/app1/mod1/mod1_1.hpp", "ut_data/app1/mod1/mod1_2.hpp"};
40
41         ASSERT_EQ(app1_mod1_IncsAtob, app1_mod1->IncsAtob);
42         ASSERT_EQ(Paths_t{"ut_data/app1/a_1.cpp.h"}, app1_mod1->IncsBtoA);
43     }
44     {
45         auto const app1_mod1 = FindDepRels(dep_all, mod1, app1);
46         ASSERT_EQ("ut_data/app1", app1_mod1->PackageA);
47         ASSERT_EQ("ut_data/app1/mod1", app1_mod1->PackageB);
48
49         ASSERT_EQ(6, app1_mod1->CountAtob);
50         ASSERT_EQ(1, app1_mod1->CountBtoA);
51
52         auto const app1_mod1_IncsAtob
53             = Paths_t{"ut_data/app1/mod1/mod1_1.hpp", "ut_data/app1/mod1/mod1_2.hpp"};
54
55         ASSERT_EQ(app1_mod1_IncsAtob, app1_mod1->IncsAtob);
56         ASSERT_EQ(Paths_t{"ut_data/app1/a_1.cpp.h"}, app1_mod1->IncsBtoA);
57     }
58     {
59         auto const mod1_mod2_2 = FindDepRels(dep_all, mod1, mod2_2);
60         ASSERT_EQ("ut_data/app1/mod1", mod1_mod2_2->PackageA);
61         ASSERT_EQ("ut_data/app1/mod2/mod2_2", mod1_mod2_2->PackageB);
62
63         ASSERT_EQ(1, mod1_mod2_2->CountAtob);
64         ASSERT_EQ(4, mod1_mod2_2->CountBtoA);
65
66         auto const mod1_mod2_2_IncsAtob = Paths_t{"ut_data/app1/mod2/mod2_2/mod2_2_1.h"};
67
68         ASSERT_EQ(mod1_mod2_2_IncsAtob, mod1_mod2_2->IncsAtob);
69         ASSERT_EQ((Paths_t{"ut_data/app1/mod1/mod1_1.hpp", "ut_data/app1/mod1/mod1_2.hpp"}),
70                   mod1_mod2_2->IncsBtoA);
71     }
72     {
73         auto const app1_mod2_2 = FindDepRels(dep_all, app1, mod2_2);
74         ASSERT_EQ("ut_data/app1", app1_mod2_2->PackageA);
75         ASSERT_EQ("ut_data/app1/mod2/mod2_2", app1_mod2_2->PackageB);
76
77         ASSERT_EQ(3, app1_mod2_2->CountAtob);
78         ASSERT_EQ(2, app1_mod2_2->CountBtoA);
79
80         ASSERT_EQ(Paths_t{"ut_data/app1/mod2/mod2_2/mod2_2_1.h"}, app1_mod2_2->IncsAtob);
81
82         auto const app1_mod2_2_IncsAtob = Paths_t{"ut_data/app1/a_1.cpp.h"};
83         ASSERT_EQ(app1_mod2_2_IncsAtob, app1_mod2_2->IncsBtoA);
84     }
85 }
86 } // namespace
87 } // namespace Dependency

```

exampledeps/dependency/ut/cpp_dir_ut.cpp

```

1 #include "gtest_wrapper.h"
2
3 #include "cpp_dir.h"
4 #include "cpp_src.h"
5
6 namespace Dependency {
7     namespace {
8
9     TEST(cpp_dir, GenCppDirs)

```

```

10 {
11     using FileUtils::Paths_t;
12
13     auto const [dirs, srcts] = GetCppDirsSrcs({"ut_data/app1", "ut_data/app2///"});
14     auto const db           = FileUtils::GenFilename2Path(srcts);
15     auto const cpp_dirs     = CppDirs_t{GenCppDirs(srcts, db)};
16
17     auto a_1_cpp = std::find_if(cpp_dirs.begin(), cpp_dirs.end(),
18                                [] (CppDir const& pkg) { return pkg.Path() == "a_1_cpp.cpp"; });
19     ASSERT_NE(a_1_cpp, cpp_dirs.end());
20
21     auto a_1_cpp_h = std::find_if(cpp_dirs.begin(), cpp_dirs.end(),
22                                   [] (CppDir const& pkg) { return pkg.Path() == "a_1_cpp.h"; });
23     ASSERT_NE(a_1_cpp_h, cpp_dirs.end());
24
25     auto mod2_2_1_h = std::find_if(cpp_dirs.begin(), cpp_dirs.end(),
26                                   [] (CppDir const& pkg) { return pkg.Path() == "mod2_2_1.h"; });
27     ASSERT_NE(mod2_2_1_h, cpp_dirs.end());
28
29     auto ret_a_1_cpp = std::pair<uint32_t, Paths_t>{a_1_cpp->DependsOn(*a_1_cpp_h)};
30     ASSERT_EQ(0, ret_a_1_cpp.first);
31
32     auto ret_mod2_2_1_h = std::pair<uint32_t, Paths_t>{mod2_2_1_h->DependsOn(*a_1_cpp_h)};
33     ASSERT_EQ(1, ret_mod2_2_1_h.first);
34 }
35
36 TEST(cpp_dir, CppDir)
37 {
38     using FileUtils::Paths_t;
39
40     auto const [dirs, srcts] = GetCppDirsSrcs({"ut_data/app1", "ut_data/app2///"});
41     auto const package_srcs = FileUtils::AssginSrcsToDirs(dirs, srcts);
42     auto const db           = FileUtils::GenFilename2Path(srcts);
43
44     auto mod1 = CppDir{"ut_data/app1/mod1", GenCppSrc(package_srcs.at("ut_data/app1/mod1"), db)};
45     auto app2 = CppDir{"ut_data/app2", GenCppSrc(package_srcs.at("ut_data/app2"), db)};
46
47     ASSERT_TRUE(mod1.Contains("ut_data/app1/mod1/mod1_1.cpp"));
48     ASSERT_TRUE(mod1.Contains("ut_data/app1/mod1/mod1_1.hpp"));
49     ASSERT_TRUE(mod1.Contains("ut_data/app1/mod1/mod1_2.hpp"));
50     ASSERT_FALSE(mod1.Contains("ut_data/app1/mod2/mod2_1.cpp"));
51
52     auto ret_mod1 = std::pair<uint32_t, Paths_t>{mod1.DependsOn(app2)};
53     ASSERT_EQ(0, ret_mod1.first);
54     ASSERT_EQ(0, ret_mod1.second.size());
55
56     auto ret_app2 = std::pair<uint32_t, Paths_t>{app2.DependsOn(mod1)};
57     ASSERT_EQ(4, ret_app2.first);
58     ASSERT_EQ((Paths_t{"ut_data/app1/mod1/mod1_1.hpp", "ut_data/app1/mod1/mod1_2.hpp"}),
59               ret_app2.second);
60 }
61
62 TEST(cpp_dir, operator_eq_tl)
63 {
64     auto const [dirs, srcts] = GetCppDirsSrcs({"ut_data/app1", "ut_data/app2///"});
65     auto const package_srcs = FileUtils::AssginSrcsToDirs(dirs, srcts);
66     auto const db           = FileUtils::GenFilename2Path(srcts);
67
68     auto mod1_0 = CppDir{"ut_data/app1/mod1", GenCppSrc(package_srcs.at("ut_data/app1/mod1"), db)};
69     auto mod1_1 = CppDir{"ut_data/app1/mod1", GenCppSrc(package_srcs.at("ut_data/app1/mod1"), db)};
70     auto app2   = CppDir{"ut_data/app2", GenCppSrc(package_srcs.at("ut_data/app2"), db)};
71
72     ASSERT_EQ(mod1_0, mod1_0);
73     ASSERT_EQ(mod1_0, mod1_1);
74     ASSERT_EQ(mod1_1, mod1_0);
75
76     ASSERT_NE(mod1_0, app2);
77     ASSERT_LT(mod1_0, app2);
78     ASSERT_GT(app2, mod1_0);
79 }
80 } // namespace
81 } // namespace Dependency

```

exampledeps/dependency/ut/cpp_src_ut.cpp

```

1 #include "gtest_wrapper.h"
2
3 #include "cpp_src.h"
4
5 namespace Dependency {

```

```

6 namespace {
7
8 TEST(cpp_src, CppSrc)
9 {
10    using FileUtils::Paths_t;
11
12    auto const [act_dirs, act_srcs] = GetCppDirsSrcs({"ut_data/app1"});
13    auto const db
14        = FileUtils::GenFilename2Path(act_srcs);
15    auto const cpp_src
16        = CppSrc{"ut_data/app1/a_1_c.c", db};
17
18    auto const exp_incs = Paths_t{"ut_data/app1/a_1_c.h",
19                                "ut_data/app1/a_1_cpp.h",
20                                "ut_data/app1/mod1/mod1_1.hpp",
21                                "ut_data/app1/mod1/mod1_2.hpp",
22                                "ut_data/app1/mod2/mod2_1/mod2_1_1.h",
23                                "ut_data/app1/mod2/mod2_2/mod2_2_1.h"};
24
25    ASSERT_EQ(cpp_src.GetIncs(), exp_incs);
26
27    auto const exp_not_found = Paths_t{"stdio.h", "string.h"};
28    ASSERT_EQ(cpp_src.GetIncsNotFound(), exp_not_found);
29
30    auto const cpp_src2 = CppSrc{"ut_data/app1/a_1_cpp.h", db};
31
32    auto const exp_incs2 = Paths_t{
33        "ut_data/app1/a_1_cpp.h", "ut_data/app1/mod1/mod1_1.hpp", "ut_data/app1/mod1/mod1_2.hpp",
34        "ut_data/app1/mod2/mod2_1/mod2_1_1.h", "ut_data/app1/mod2/mod2_2/mod2_2_1.h"};
35    ASSERT_EQ(cpp_src2.GetIncs(), exp_incs2);
36
37    ASSERT_EQ(cpp_src2.GetIncsNotFound(), Paths_t{});
38
39    auto const cpp_src3 = CppSrc{"ut_data/app1/mod1/mod1_2.hpp", db};
40
41    ASSERT_EQ(cpp_src3.GetIncs(), Paths_t{});
42
43 TEST(cpp_src, GenCppSrc)
44 {
45    using FileUtils::Paths_t;
46
47    auto const [act_dirs, act_srcs] = GetCppDirsSrcs({"ut_data/app1"});
48    auto const db
49        = FileUtils::GenFilename2Path(act_srcs);
50    auto const srcs
51        = Paths_t{"ut_data/app1/a_1_c.c", "ut_data/app1/a_1_c.h", "ut_data/app1/a_1_cpp.cpp"};
52    auto const cpp_srcs_act = GenCppSrc(srcs, db);
53
54    ASSERT_EQ(cpp_srcs_act.size(), 3);
55
56    Paths_t const exp_incs[]{};
57    {"ut_data/app1/a_1_c.h", "ut_data/app1/a_1_cpp.h", "ut_data/app1/mod1/mod1_1.hpp",
58    "ut_data/app1/mod1/mod1_2.hpp", "ut_data/app1/mod2/mod2_1/mod2_1_1.h",
59    "ut_data/app1/mod2/mod2_2/mod2_2_1.h"}, {"ut_data/app1/a_1_cpp.h", "ut_data/app1/mod1/mod1_1.hpp", "ut_data/app1/mod1/mod1_2.hpp",
60    "ut_data/app1/mod2/mod2_1/mod2_1_1.h", "ut_data/app1/mod2/mod2_2/mod2_2_1.h"}, {},
61    {};
62 };
63 Paths_t const exp_not_found[]{};
64 {"stdio.h", "string.h"}, {"stdio.h", "string.h"}, {}, {};
65
66
67
68
69    auto it_exp_srcs
70        = srcs.cbegin();
71    auto it_exp_incs
72        = std::cbegin(exp_incs);
73    auto it_exp_not_found
74        = std::cbegin(exp_not_found);
75
76    for (auto it_act = cpp_srcs_act.cbegin(); it_act != cpp_srcs_act.cend(); ++it_act) {
77        ASSERT_EQ(*it_exp_srcs, it_act->Path());
78        ASSERT_EQ(*it_exp_incs, it_act->GetIncs());
79        ASSERT_EQ(*it_exp_not_found, it_act->GetIncsNotFound());
80
81        ++it_exp_srcs;
82        ++it_exp_incs;
83        ++it_exp_not_found;
84    }
85
86    ASSERT_EQ(it_exp_srcs, srcs.cend());
87    ASSERT_EQ(it_exp_incs, std::cend(exp_incs));
88    ASSERT_EQ(it_exp_not_found, std::cend(exp_not_found));

```

```

86 }
87
88 TEST(cpp_src, operator_equal)
89 {
90     using FileUtils::Paths_t;
91
92     auto const [act_dirs, act_srcs] = GetCppDirsSrcs({"ut_data/app1"});
93     auto const db
94                     = FileUtils::GenFilename2Path(act_srcs);
95
96     auto const cpp_src_0 = CppSrc{"ut_data/app1/a_1_c.c", db};
97     auto const cpp_src_1 = CppSrc{"ut_data/app1/a_1_c.c", db};
98     auto const cpp_src_2 = CppSrc{"ut_data/app1/a_1_c.h", db};
99
100    ASSERT_EQ(cpp_src_0, cpp_src_0);
101    ASSERT_EQ(cpp_src_0, cpp_src_1);
102    ASSERT_EQ(cpp_src_1, cpp_src_0);
103    ASSERT_NE(cpp_src_0, cpp_src_2);
104
105 TEST(cpp_src, operator_lt)
106 {
107     using FileUtils::Paths_t;
108
109     auto const [act_dirs, act_srcs] = GetCppDirsSrcs({"ut_data/app1"});
110     auto const db
111                     = FileUtils::GenFilename2Path(act_srcs);
112
113     auto const cpp_src_0 = CppSrc{"ut_data/app1/a_1_c.c", db};
114     auto const cpp_src_1 = CppSrc{"ut_data/app1/a_1_c.h", db};
115
116     ASSERT_LT(cpp_src_0, cpp_src_1);
117     ASSERT_GT(cpp_src_1, cpp_src_0);
118
119 TEST(cpp_src, ToString)
120 {
121     using FileUtils::Paths_t;
122
123     auto const [act_dirs, act_srcs] = GetCppDirsSrcs({"ut_data/app1"});
124     auto const db
125                     = FileUtils::GenFilename2Path(act_srcs);
126     auto const cpp_src
127                     = CppSrc{"ut_data/app1/a_1_c.c", db};
128
129     auto const exp = std::string_view{
130         "file           : a_1_c.c\n"
131         "path           : ut_data/app1/a_1_c.c\n"
132         "include        : ut_data/app1/a_1_c.h ut_data/app1/a_1_cpp.h "
133         "ut_data/app1/mod1/mod1_1.hpp ut_data/app1/mod1/mod1_2.hpp "
134         "ut_data/app1/mod2/mod2_1/mod2_1.h ut_data/app1/mod2/mod2_2/mod2_2_1.h\n"
135         "include not found : stdio.h string.h\n"};
136
137
138 TEST(cpp_src, GetCppDirsSrcs)
139 {
140     using FileUtils::Paths_t;
141
142     {
143         auto const exp_dirs = Paths_t{"ut_data/app1",
144                                     "ut_data/app1/mod1",
145                                     "ut_data/app1/mod2",
146                                     "ut_data/app1/mod2/mod2_1",
147                                     "ut_data/app1/mod2/mod2_2",
148                                     "ut_data/app2"};
149
150         auto const exp_srcs = Paths_t{"ut_data/app1/a_1_c.c",
151                                     "ut_data/app1/a_1_c.h",
152                                     "ut_data/app1/a_1_cpp.cpp",
153                                     "ut_data/app1/a_1_cpp.h",
154                                     "ut_data/app1/a_2_c.C",
155                                     "ut_data/app1/a_2_c.H",
156                                     "ut_data/app1/a_2_cpp.cxx",
157                                     "ut_data/app1/a_2_cpp.hpp",
158                                     "ut_data/app1/a_3_cpp.cc",
159                                     "ut_data/app1/mod1/mod1_1.cpp",
160                                     "ut_data/app1/mod1/mod1_1.hpp",
161                                     "ut_data/app1/mod1/mod1_2.hpp",
162                                     "ut_data/app1/mod2/mod2_1/mod2_1_1.cpp",
163                                     "ut_data/app1/mod2/mod2_1/mod2_1_1.h",
164                                     "ut_data/app1/mod2/mod2_1.cpp",
165                                     "ut_data/app1/mod2/mod2_1.hpp",
166     };

```

```

166                               "ut_data/app1/mod2/mod2_2/mod2_2_1.cpp",
167                               "ut_data/app1/mod2/mod2_2/mod2_2_1.h",
168                               "ut_data/app2/b_1.cpp",
169                               "ut_data/app2/b_1.h"};
170
171     auto const [act_dirs, act_srcs] = GetCppDirsSrcs({"./ut_data/app1", "ut_data/app2///"});
172
173     ASSERT_EQ(act_dirs, exp_dirs);
174     ASSERT_EQ(act_srcs, exp_srcs);
175 }
176 {
177     auto const exp_dirs = Paths_t{"ut_data",
178                                 "ut_data/app1",
179                                 "ut_data/app1/mod1",
180                                 "ut_data/app1/mod2",
181                                 "ut_data/app1/mod2/mod2_1",
182                                 "ut_data/app1/mod2/mod2_2",
183                                 "ut_data/app2"};
184
185     auto const exp_srcs
186         = Paths_t{"ut_data/app1/a_1_c.c",
187                 "ut_data/app1/a_1_c.h",
188                 "ut_data/app1/a_1_cpp.cpp",
189                 "ut_data/app1/a_1_cpp.h",
190                 "ut_data/app1/a_2_c.C",
191                 "ut_data/app1/a_2_c.H",
192                 "ut_data/app1/a_2_cpp.cxx",
193                 "ut_data/app1/a_2_cpp.hpp",
194                 "ut_data/app1/a_3_cpp.cc",
195                 "ut_data/app1/mod1/mod1_1.cpp",
196                 "ut_data/app1/mod1/mod1_1.hpp",
197                 "ut_data/app1/mod1/mod1_2.hpp",
198                 "ut_data/app1/mod2/mod2_1/mod2_1_1.cpp",
199                 "ut_data/app1/mod2/mod2_1/mod2_1_1.h",
200                 "ut_data/app1/mod2/mod2_1.cpp",
201                 "ut_data/app1/mod2/mod2_1.hpp",
202                 "ut_data/app1/mod2/mod2_2/mod2_2_1.cpp",
203                 "ut_data/app1/mod2/mod2_2/mod2_2_1.h",
204                 "ut_data/app2/b_1.cpp",
205                 "ut_data/app2/b_1.h"};
206
207     auto const [act_dirs, act_srcs] = GetCppDirsSrcs("././ut_data");
208
209     ASSERT_EQ(act_dirs, exp_dirs);
210     ASSERT_EQ(act_srcs, exp_srcs);
211 }
212 } // namespace Dependency

```

exampledeps/dependency/ut/deps_scenario_ut.cpp

```

1 #include <iostream>
2
3 #include "gtest_wrapper.h"
4
5 #include "dependency/deps_scenario.h"
6 #include "file_utils/load_store.h"
7 #include "file_utils/load_store_row.h"
8
9 namespace Dependency {
10 namespace {
11
12 TEST(deps_scenario, PkgGenerator)
13 {
14     using FileUtils::Paths_t;
15
16     {
17         auto pg = PkgGenerator{"ut_data/load_store/pkg_org", true, Paths_t{"ut_data/app3/"}, ""};
18         auto exp = std::string{
19             "#dir\n"
20             "ut_data/app1\n"
21             "ut_data/app1/mod1\n"
22             "ut_data/app1/mod2\n"
23             "ut_data/app1/mod2/mod2_1\n"
24             "ut_data/app1/mod2/mod2_2\n"
25             "ut_data/app2\n"};
26
27         auto ss = std::ostringstream{};
28
29         pg.Output(ss);
30         ASSERT_EQ(exp, ss.str());

```

```

31     }
32     {
33         auto pg = PkgGenerator{"ut_data/load_store/pkg_org", false, Paths_t{}, ""};
34         auto exp = std::string{
35             "#dir\n"
36             "ut_data/app1\n"
37             "ut_data/app1/mod1\n"
38             "ut_data/app1/mod2/mod2_1\n"
39             "ut_data/app2\n";
40
41         auto ss = std::ostringstream{};
42
43         pg.Output(ss);
44         ASSERT_EQ(exp, ss.str());
45     }
46 }
47
48 TEST(deps_scenario, PkgGenerator2)
49 {
50     using FileUtils::Paths_t;
51
52     {
53         auto pg = PkgGenerator:"", false, Paths_t{"ut_data/app1", "ut_data/app2"}, "";
54         auto exp = std::string{
55             "#dir\n"
56             "ut_data/app1\n"
57             "ut_data/app2\n";
58
59         auto ss = std::ostringstream{};
60
61         pg.Output(ss);
62         ASSERT_EQ(exp, ss.str());
63     }
64     {
65         auto pg = PkgGenerator:"", false, Paths_t{"ut_data/app1", "ut_data/app2"}, "hehe";
66         auto exp = std::string{
67             "#dir\n"
68             "ut_data/app1\n"
69             "ut_data/app2\n";
70
71         auto ss = std::ostringstream{};
72
73         pg.Output(ss);
74         ASSERT_EQ(exp, ss.str());
75     }
76     {
77         auto pg = PkgGenerator:"", false, Paths_t{"ut_data/app1", "ut_data/app2"}, "./app2";
78         auto exp = std::string{
79             "#dir\n"
80             "ut_data/app1\n";
81
82         auto ss = std::ostringstream{};
83
84         pg.Output(ss);
85         ASSERT_EQ(exp, ss.str());
86     }
87     {
88         auto pg = PkgGenerator:"", true, Paths_t{"ut_data/app1", "ut_data/app2"}, "";
89         auto exp = std::string{
90             "#dir\n"
91             "ut_data/app1\n"
92             "ut_data/app1/mod1\n"
93             "ut_data/app1/mod2\n"
94             "ut_data/app1/mod2/mod2_1\n"
95             "ut_data/app1/mod2/mod2_2\n"
96             "ut_data/app2\n";
97
98         auto ss = std::ostringstream{};
99
100        pg.Output(ss);
101        ASSERT_EQ(exp, ss.str());
102    }
103    {
104        auto pg = PkgGenerator:"", true, Paths_t{"ut_data/app1", "ut_data/app2"}, "./mod2/*";
105        auto exp = std::string{
106            "#dir\n"
107            "ut_data/app1\n"
108            "ut_data/app1/mod1\n"
109            "ut_data/app1/mod2\n"
110            "ut_data/app2\n";

```

```

111
112     auto ss = std::ostringstream{};
113
114     pg.Output(ss);
115     ASSERT_EQ(exp, ss.str());
116 }
117
118 auto exception_occured = false;
119 try {
120     auto pg = PkgGenerator{ "", true, Paths_t{"ut_data/app1/a_1_c.c", "ut_data/app2"}, ""};
121 }
122 catch (std::runtime_error const& e) {
123     exception_occured = true;
124     ASSERT_STREQ("ut_data/app1/a_1_c.c not directory", e.what());
125 }
126 ASSERT_TRUE(exception_occured);
127 }
128
129 TEST(deps_scenario, SrcsGenerator)
130 {
131     using FileUtils::Paths_t;
132
133 {
134     auto sg = SrcsGenerator{
135         "", true, Paths_t{"ut_data/app1/mod2/mod2_1", "ut_data/app1/mod2/mod2_2"}, ""};
136
137 // clang-format off
138     auto exp = std::string{
139         "\n"
140         "mod2_1_1.cpp\n"
141         "file           : mod2_1_1.cpp\n"
142         "path          : ut_data/app1/mod2/mod2_1/mod2_1_1.cpp\n"
143         "include        : \n"
144         "include not found : \n"
145         "\n"
146         "mod2_1_1.h\n"
147         "file           : mod2_1_1.h\n"
148         "path          : ut_data/app1/mod2/mod2_1/mod2_1_1.h\n"
149         "include        : ut_data/app1/mod2/mod2_1/mod2_1_1.h ut_data/app1/mod2/mod2_2/mod2_2_1.h\n"
150         "include not found : a_1_cpp.h\n"
151         "\n"
152         "mod2_2_1.cpp\n"
153         "file           : mod2_2_1.cpp\n"
154         "path          : ut_data/app1/mod2/mod2_2/mod2_2_1.cpp\n"
155         "include        : ut_data/app1/mod2/mod2_1/mod2_1_1.h ut_data/app1/mod2/mod2_2/mod2_2_1.h\n"
156         "include not found : a_1_cpp.h\n"
157         "\n"
158         "mod2_2_1.h\n"
159         "file           : mod2_2_1.h\n"
160         "path          : ut_data/app1/mod2/mod2_2/mod2_2_1.h\n"
161         "include        : ut_data/app1/mod2/mod2_1/mod2_1_1.h ut_data/app1/mod2/mod2_2/mod2_2_1.h\n"
162         "include not found : a_1_cpp.h\n"
163         "\n"};
164     // clang-format on
165
166     auto ss = std::ostringstream{};
167
168     sg.Output(ss);
169     ASSERT_EQ(exp, ss.str());
170 }
171
172 {
173     auto sg = SrcsGenerator{
174         "", true, Paths_t{"ut_data/app1/mod2/mod2_1", "ut_data/app1/mod2/mod2_2"}, "./mod2_2"};
175
176     auto exp = std::string{
177         "\n"
178         "mod2_1_1.cpp\n"
179         "file           : mod2_1_1.cpp\n"
180         "path          : ut_data/app1/mod2/mod2_1/mod2_1_1.cpp\n"
181         "include        : \n"
182         "include not found : \n"
183         "\n"
184         "mod2_1_1.h\n"
185         "file           : mod2_1_1.h\n"
186         "path          : ut_data/app1/mod2/mod2_1/mod2_1_1.h\n"
187         "include        : \n"
188

```

```

191         "include not found : mod2_2_1.h\n"
192         "\n"};
193
194     auto ss = std::ostringstream{};
195
196     sg.Output(ss);
197     ASSERT_EQ(exp, ss.str());
198 }
199 }
200
201 TEST(deps_scenario, Pkg2SrcsGenerator)
202 {
203     using FileUtils::Paths_t;
204
205     auto exception_occured = false;
206
207     try {
208         auto p2sg
209             = Pkg2SrcsGenerator{"ut_data/app1/a_1_c.c", false, false, Paths_t{"ut_data/app3/"}, ""};
210     }
211     catch (std::runtime_error const& e) {
212         exception_occured = true;
213         ASSERT_STREQ("ut_data/app1/a_1_c.c is illegal", e.what());
214     }
215     ASSERT_TRUE(exception_occured);
216
217     {
218         auto p2sg = Pkg2SrcsGenerator{"ut_data/load_store/pkg_org", true, false,
219                                         Paths_t{"ut_data/app3/"}, ""};
220
221         auto exp = std::string{
222             "#dir2srcs\n"
223             "ut_data/app1\n"
224             "    ut_data/app1/a_1_c.c\n"
225             "    ut_data/app1/a_1_c.h\n"
226             "    ut_data/app1/a_1_cpp.cpp\n"
227             "    ut_data/app1/a_1_cpp.h\n"
228             "    ut_data/app1/a_2_c.C\n"
229             "    ut_data/app1/a_2_c.H\n"
230             "    ut_data/app1/a_2_cpp.cxx\n"
231             "    ut_data/app1/a_2_cpp.hpp\n"
232             "    ut_data/app1/a_3_cpp.cc\n"
233             "\n"
234             "ut_data/app1/mod1\n"
235             "    ut_data/app1/mod1/mod1_1.cpp\n"
236             "    ut_data/app1/mod1/mod1_1.hpp\n"
237             "    ut_data/app1/mod1/mod1_2.hpp\n"
238             "\n"
239             "ut_data/app1/mod2\n"
240             "    ut_data/app1/mod2/mod2_1.cpp\n"
241             "    ut_data/app1/mod2/mod2_1.hpp\n"
242             "\n"
243             "ut_data/app1/mod2/mod2_1\n"
244             "    ut_data/app1/mod2/mod2_1/mod2_1_1.cpp\n"
245             "    ut_data/app1/mod2/mod2_1/mod2_1_1.h\n"
246             "\n"
247             "ut_data/app1/mod2/mod2_2\n"
248             "    ut_data/app1/mod2/mod2_2/mod2_2_1.cpp\n"
249             "    ut_data/app1/mod2/mod2_2/mod2_2_1.h\n"
250             "\n"
251             "ut_data/app2\n"
252             "    ut_data/app2/b_1.cpp\n"
253             "    ut_data/app2/b_1.h\n"
254             "\n"};
255
256         auto ss = std::ostringstream{};
257
258         p2sg.Output(ss);
259         ASSERT_EQ(exp, ss.str());
260     }
261     {
262         auto p2sg = Pkg2SrcsGenerator{"ut_data/load_store/pkg_org", true, false,
263                                         Paths_t{"ut_data/app3/"}, "./mod2\\b.*"};
264
265         auto exp = std::string{
266             "#dir2srcs\n"
267             "ut_data/app1\n"
268             "    ut_data/app1/a_1_c.c\n"
269             "    ut_data/app1/a_1_c.h\n"
270             "    ut_data/app1/a_1_cpp.cpp\n"

```

```

271     "    ut_data/app1/a_1_cpp.h\n"
272     "    ut_data/app1/a_2_c.C\n"
273     "    ut_data/app1/a_2_c.H\n"
274     "    ut_data/app1/a_2_cpp.cxx\n"
275     "    ut_data/app1/a_2_cpp.hpp\n"
276     "    ut_data/app1/a_3_cpp.cc\n"
277     "    ut_data/app1/mod2/mod2_1/mod2_1_1.cpp\n"
278     "    ut_data/app1/mod2/mod2_1/mod2_1_1.h\n"
279     "    ut_data/app1/mod2/mod2_1.cpp\n"
280     "    ut_data/app1/mod2/mod2_1.hpp\n"
281     "    ut_data/app1/mod2/mod2_2/mod2_2_1.cpp\n"
282     "    ut_data/app1/mod2/mod2_2/mod2_2_1.h\n"
283     "\n"
284     "ut_data/app1/mod1\n"
285     "    ut_data/app1/mod1/mod1_1.cpp\n"
286     "    ut_data/app1/mod1/mod1_1.hpp\n"
287     "    ut_data/app1/mod1/mod1_2.hpp\n"
288     "\n"
289     "ut_data/app2\n"
290     "    ut_data/app2/b_1.cpp\n"
291     "    ut_data/app2/b_1.h\n"
292     "\n"};
```

293

```

294     auto ss = std::ostringstream{};

295     p2sg.Output(ss);
296     ASSERT_EQ(exp, ss.str());
297 }
298 {
```

300 auto p2sg = Pkg2SrcsGenerator{"ut_data/load_store/pkg_org", false, false,
301 Paths_t{"ut_data/app3/"}, ""};

302

```

303     auto exp = std::string{
304         "#dir2srcs\n"
305         "ut_data/app1\n"
306         "    ut_data/app1/a_1_c.c\n"
307         "    ut_data/app1/a_1_c.h\n"
308         "    ut_data/app1/a_1_cpp.cpp\n"
309         "    ut_data/app1/a_1_cpp.h\n"
310         "    ut_data/app1/a_2_c.C\n"
311         "    ut_data/app1/a_2_c.H\n"
312         "    ut_data/app1/a_2_cpp.cxx\n"
313         "    ut_data/app1/a_2_cpp.hpp\n"
314         "    ut_data/app1/a_3_cpp.cc\n"
315         "    ut_data/app1/mod2/mod2_1.cpp\n"
316         "    ut_data/app1/mod2/mod2_1.hpp\n"
317         "    ut_data/app1/mod2/mod2_2/mod2_2_1.cpp\n"
318         "    ut_data/app1/mod2/mod2_2/mod2_2_1.h\n"
319         "\n"
320         "ut_data/app1/mod1\n"
321         "    ut_data/app1/mod1/mod1_1.cpp\n"
322         "    ut_data/app1/mod1/mod1_1.hpp\n"
323         "    ut_data/app1/mod1/mod1_2.hpp\n"
324         "\n"
325         "ut_data/app1/mod2\n"
326         "\n"
327         "\n"
328         "ut_data/app1/mod2/mod2_1\n"
329         "    ut_data/app1/mod2/mod2_1/mod2_1_1.cpp\n"
330         "    ut_data/app1/mod2/mod2_1/mod2_1_1.h\n"
331         "\n"
332         "ut_data/app2\n"
333         "    ut_data/app2/b_1.cpp\n"
334         "    ut_data/app2/b_1.h\n"
335         "\n"};
```

336

```

337     auto ss = std::ostringstream{};

338     p2sg.Output(ss);
339     ASSERT_EQ(exp, ss.str());
340 }
```

341 {

343 auto p2sg = Pkg2SrcsGenerator{"ut_data/load_store/pkg_org", false, false,
344 Paths_t{"ut_data/app3/"}, "ut_data/app2"};

345

```

346     auto exp = std::string{
347         "#dir2srcs\n"
348         "no_package\n"
349         "    ut_data/app2/b_1.cpp\n"
350         "    ut_data/app2/b_1.h\n"
```

```

351         "\n"
352         "ut_data/app1\n"
353         "    ut_data/app1/a_1_c.c\n"
354         "    ut_data/app1/a_1_c.h\n"
355         "    ut_data/app1/a_1_cpp.cpp\n"
356         "    ut_data/app1/a_1_cpp.h\n"
357         "    ut_data/app1/a_2_c.C\n"
358         "    ut_data/app1/a_2_c.H\n"
359         "    ut_data/app1/a_2_cpp.cxx\n"
360         "    ut_data/app1/a_2_cpp.hpp\n"
361         "    ut_data/app1/a_3_cpp.cc\n"
362         "    ut_data/app1/mod2/mod2_1.cpp\n"
363         "    ut_data/app1/mod2/mod2_1.hpp\n"
364         "    ut_data/app1/mod2/mod2_2/mod2_2_1.cpp\n"
365         "    ut_data/app1/mod2/mod2_2/mod2_2_1.h\n"
366         "\n"
367         "ut_data/app1/mod1\n"
368         "    ut_data/app1/mod1/mod1_1.cpp\n"
369         "    ut_data/app1/mod1/mod1_1.hpp\n"
370         "    ut_data/app1/mod1/mod1_2.hpp\n"
371         "\n"
372         "ut_data/app1/mod2\n"
373         "\n"
374         "\n"
375         "ut_data/app1/mod2/mod2_1\n"
376         "    ut_data/app1/mod2/mod2_1/mod2_1_1.cpp\n"
377         "    ut_data/app1/mod2/mod2_1/mod2_1_1.h\n"
378         "\n";
379
380     auto ss = std::ostringstream{};
381
382     p2sg.Output(ss);
383     ASSERT_EQ(exp, ss.str());
384 }
385 }
386
387 TEST(deps_scenario, Pkg2SrcsGenerator2)
388 {
389     using FileUtils::Paths_t;
390
391     {
392         auto dirs2srcts_org_str = std::string{
393             "#dir2srcts\n"
394             "ut_data\n"
395             "    ut_data/app1/a_1_c.c\n"
396             "    ut_data/app1/a_1_c.h\n"
397             "    ut_data/app1/a_1_cpp.cpp\n"
398             "    ut_data/app1/a_1_cpp.h\n"
399             "    ut_data/app1/a_2_c.C\n"
400             "    ut_data/app1/a_2_c.H\n"
401             "    ut_data/app1/a_2_cpp.cxx\n"
402             "    ut_data/app1/a_2_cpp.hpp\n"
403             "    ut_data/app1/a_3_cpp.cc\n"
404             "    ut_data/app1/mod1/mod1_1.cpp\n"
405             "    ut_data/app1/mod1/mod1_1.hpp\n"
406             "    ut_data/app1/mod1/mod1_2.hpp\n"
407             "\n"
408             "ut_data/app1/mod2\n"
409             "    ut_data/app1/mod2/mod2_1.cpp\n"
410             "    ut_data/app1/mod2/mod2_1.hpp\n"
411             "\n"
412             "ut_data/app1/mod2/mod2_1\n"
413             "    ut_data/app1/mod2/mod2_1/mod2_1_1.cpp\n"
414             "    ut_data/app1/mod2/mod2_1/mod2_1_1.h\n"
415             "\n"
416             "ut_data/app1/mod2/mod2_2\n"
417             "    ut_data/app1/mod2/mod2_2/mod2_2_1.cpp\n"
418             "    ut_data/app1/mod2/mod2_2/mod2_2_1.h\n"
419             "\n"
420             "ut_data/app2\n"
421             "    ut_data/app2/b_1.cpp\n"
422             "    ut_data/app2/b_1.h\n"
423             "\n"};
424
425     auto p2sg
426         = Pkg2SrcsGenerator{"ut_data/load_store/dirs2srcts_org", true, false, Paths_t{}, ""};
427
428     auto ss = std::ostringstream{};
429
430     p2sg.Output(ss);

```

```

431     ASSERT_EQ(dirs2srcs_org_str, ss.str());
432 }
433 {
434     auto dirs2srcs_org_str = std::string{
435         "#dir2srcs\n"
436         "ut_data\n"
437         "    ut_data/app1/a_1_c.c\n"
438         "    ut_data/app1/a_1_c.h\n"
439         "    ut_data/app1/a_1_cpp.cpp\n"
440         "    ut_data/app1/a_1_cpp.h\n"
441         "    ut_data/app1/a_2_c.C\n"
442         "    ut_data/app1/a_2_c.H\n"
443         "    ut_data/app1/a_2_cpp.cxx\n"
444         "    ut_data/app1/a_2_cpp.hpp\n"
445         "    ut_data/app1/a_3_cpp.cc\n"
446         "    ut_data/app1/mod1/mod1_1.cpp\n"
447         "    ut_data/app1/mod1/mod1_1.hpp\n"
448         "    ut_data/app1/mod1/mod1_2.hpp\n"
449         "\n"
450         "ut_data/app1\n"
451         "\n"
452         "\n"
453         "ut_data/app1/mod2\n"
454         "    ut_data/app1/mod2/mod2_1.cpp\n"
455         "    ut_data/app1/mod2/mod2_1.hpp\n"
456         "\n"
457         "ut_data/app1/mod2/mod2_1\n"
458         "    ut_data/app1/mod2/mod2_1/mod2_1_1.cpp\n"
459         "    ut_data/app1/mod2/mod2_1/mod2_1_1.h\n"
460         "\n"
461         "ut_data/app1/mod2/mod2_2\n"
462         "    ut_data/app1/mod2/mod2_2/mod2_2_1.cpp\n"
463         "    ut_data/app1/mod2/mod2_2/mod2_2_1.h\n"
464         "\n"
465         "ut_data/app2\n"
466         "    ut_data/app2/b_1.cpp\n"
467         "    ut_data/app2/b_1.h\n"
468         "\n"};
469
470     auto const dirs = Paths_t{"ut_data", "ut_data/app1/mod2", "ut_data/app1/mod2/mod2_1",
471                           "ut_data/app1/mod2/mod2_2", "ut_data/app2"};
472
473     auto p2sg = Pkg2SrcsGenerator{ "", false, false, dirs, "" };
474
475     auto ss = std::ostringstream{};
476
477     p2sg.Output(ss);
478     ASSERT_EQ(dirs2srcs_org_str, ss.str());
479 }
480 {
481     auto p2sg = Pkg2SrcsGenerator{ "", true, false, Paths_t{"ut_data"}, "" };
482
483     auto const exp = std::string{
484         "#dir2srcs\n"
485         "ut_data/app1\n"
486         "    ut_data/app1/a_1_c.c\n"
487         "    ut_data/app1/a_1_c.h\n"
488         "    ut_data/app1/a_1_cpp.cpp\n"
489         "    ut_data/app1/a_1_cpp.h\n"
490         "    ut_data/app1/a_2_c.C\n"
491         "    ut_data/app1/a_2_c.H\n"
492         "    ut_data/app1/a_2_cpp.cxx\n"
493         "    ut_data/app1/a_2_cpp.hpp\n"
494         "    ut_data/app1/a_3_cpp.cc\n"
495         "\n"
496         "ut_data/app1/mod1\n"
497         "    ut_data/app1/mod1/mod1_1.cpp\n"
498         "    ut_data/app1/mod1/mod1_1.hpp\n"
499         "    ut_data/app1/mod1/mod1_2.hpp\n"
500         "\n"
501         "ut_data/app1/mod2\n"
502         "    ut_data/app1/mod2/mod2_1.cpp\n"
503         "    ut_data/app1/mod2/mod2_1.hpp\n"
504         "\n"
505         "ut_data/app1/mod2/mod2_1\n"
506         "    ut_data/app1/mod2/mod2_1/mod2_1_1.cpp\n"
507         "    ut_data/app1/mod2/mod2_1/mod2_1_1.h\n"
508         "\n"
509         "ut_data/app1/mod2/mod2_2\n"
510         "    ut_data/app1/mod2/mod2_2/mod2_2_1.cpp\n"

```

```

511         "    ut_data/app1/mod2/mod2_2/mod2_2_1.h\n"
512         "\n"
513         "ut_data/app2\n"
514         "    ut_data/app2/b_1.cpp\n"
515         "    ut_data/app2/b_1.h\n"
516         "\n"};;
517
518     auto ss = std::ostringstream{};
519
520     p2sg.Output(ss);
521     ASSERT_EQ(exp, ss.str());
522 }
523 }
524
525 TEST(deps_scenario, Pkg2PkgGenerator)
526 {
527     using FileUtils::Paths_t;
528
529     auto exception_occured = false;
530     try {
531         auto p2pg
532             = Pkg2PkgGenerator{"ut_data/app1/a_1.c.c", false, false, Paths_t{"ut_data/app3/"}, ""};
533     }
534     catch (std::runtime_error const& e) {
535         exception_occured = true;
536         ASSERT_STREQ("ut_data/app1/a_1.c.c is illegal", e.what());
537     }
538     ASSERT_TRUE(exception_occured);
539
540 {
541     auto p2pg = Pkg2PkgGenerator{"ut_data/load_store/pkg_org", true, false,
542                                 Paths_t{"ut_data/app3/"}, ""};
543
544     // clang-format off
545     auto exp = std::string {
546         "#deps\n"
547         "ut_data/app1 -> ut_data/app1/mod1 : 6 ut_data/app1/mod1/mod1_1.hpp ut_data/app1/mod1/mod1_2.hpp\n"
548         "ut_data/app1/mod1 -> ut_data/app1 : 1 ut_data/app1/a_1_cpp.h\n"
549         "\n"
550         "ut_data/app1 -> ut_data/app1/mod2 : 0 \n"
551         "ut_data/app1/mod2 -> ut_data/app1 : 0 \n"
552         "\n"
553         "ut_data/app1 -> ut_data/app1/mod2/mod2_1 : 3 ut_data/app1/mod2/mod2_1/mod2_1_1.h\n"
554         "ut_data/app1/mod2/mod2_1 -> ut_data/app1 : 1 ut_data/app1/a_1_cpp.h\n"
555         "\n"
556         "ut_data/app1 -> ut_data/app1/mod2/mod2_2 : 3 ut_data/app1/mod2/mod2_2/mod2_2_1.h\n"
557         "ut_data/app1/mod2/mod2_2 -> ut_data/app1 : 2 ut_data/app1/a_1_cpp.h\n"
558         "\n"
559         "ut_data/app1 -> ut_data/app2 : 0 \n"
560         "ut_data/app2 -> ut_data/app1 : 3 ut_data/app1/a_1_cpp.h ut_data/app1/a_2_cpp.hpp\n"
561         "\n"
562         "ut_data/app1/mod1 -> ut_data/app1/mod2 : 1 ut_data/app1/mod2/mod2_1.hpp\n"
563         "ut_data/app1/mod2 -> ut_data/app1/mod1 : 0 \n"
564         "\n"
565         "ut_data/app1/mod1 -> ut_data/app1/mod2/mod2_1 : 1 ut_data/app1/mod2/mod2_1/mod2_1_1.h\n"
566         "ut_data/app1/mod2/mod2_1 -> ut_data/app1/mod1 : 2 ut_data/app1/mod1/mod1_1.hpp
567         ut_data/app1/mod1/mod1_2.hpp\n"
568         "\n"
569         "ut_data/app1/mod1 -> ut_data/app1/mod2/mod2_2 : 1 ut_data/app1/mod2/mod2_2/mod2_2_1.h\n"
570         "ut_data/app1/mod2/mod2_2 -> ut_data/app1/mod1 : 4 ut_data/app1/mod1/mod1_1.hpp
571         ut_data/app1/mod1/mod1_2.hpp\n"
572         "\n"
573         "ut_data/app1/mod1 -> ut_data/app2 : 0 \n"
574         "ut_data/app2 -> ut_data/app1/mod1 : 4 ut_data/app1/mod1/mod1_1.hpp ut_data/app1/mod1/mod1_2.hpp\n"
575         "\n"
576         "ut_data/app1/mod2 -> ut_data/app1/mod2/mod2_1 : 0 \n"
577         "ut_data/app1/mod2/mod2_1 -> ut_data/app1/mod2 : 0 \n"
578         "\n"
579         "ut_data/app1/mod2 -> ut_data/app2 : 0 \n"
580         "ut_data/app2 -> ut_data/app1/mod2 : 0 \n"
581         "\n"
582         "ut_data/app1/mod2/mod2_1 -> ut_data/app1/mod2/mod2_2 : 1 ut_data/app1/mod2/mod2_2/mod2_2_1.h\n"
583         "ut_data/app1/mod2/mod2_2 -> ut_data/app1/mod2/mod2_1 : 2 ut_data/app1/mod2/mod2_1/mod2_1_1.h\n"
584         "\n"
585         "ut_data/app1/mod2/mod2_1 -> ut_data/app2 : 0 \n"
586         "ut_data/app2 -> ut_data/app1/mod2/mod2_1 : 2 ut_data/app1/mod2/mod2_1/mod2_1_1.h\n"
587         "\n"
588         "ut_data/app1/mod2/mod2_1 -> ut_data/app2 : 0 \n"
589         "ut_data/app2 -> ut_data/app1/mod2/mod2_1 : 2 ut_data/app1/mod2/mod2_1/mod2_1_1.h\n"
590         "\n"

```

```

589         "ut_data/app1/mod2/mod2_2 -> ut_data/app2 : 0 \n"
590         "ut_data/app2 -> ut_data/app1/mod2/mod2_2 : 2 ut_data/app1/mod2/mod2_2/mod2_2_1.h\n"
591         "\n"};
592     // clang-format on
593
594     auto ss = std::ostringstream{};
595
596     p2pg.Output(ss);
597     ASSERT_EQ(exp, ss.str());
598 }
599 {
600     auto p2pg = Pkg2PkgGenerator{"ut_data/load_store/pkg_org", true, false,
601                                 Paths_t{"ut_data/app3/"}, "./mod2\\b.*"};
602
603     // clang-format off
604     auto exp = std::string {
605         "#deps\n"
606         "ut_data/app1 -> ut_data/app1/mod1 : 12 ut_data/app1/mod1/mod1_1.hpp ut_data/app1/mod1/mod1_2.hpp\n"
607         "ut_data/app1/mod1 -> ut_data/app1 : 4 ut_data/app1/a_1.cpp.h ut_data/app1/mod2/mod2_1/mod2_1_1.h
608         ut_data/app1/mod2/mod2_1.hpp ut_data/app1/mod2/mod2_2/mod2_2_1.h\n"
609         "\n"
610         "ut_data/app1 -> ut_data/app2 : 0 \n"
611         "ut_data/app2 -> ut_data/app1 : 7 ut_data/app1/a_1.cpp.h ut_data/app1/a_2.cpp.hpp
612         ut_data/app1/mod2/mod2_1/mod2_1_1.h ut_data/app1/mod2/mod2_2/mod2_2_1.h\n"
613         "\n"
614         "ut_data/app1/mod1 -> ut_data/app2 : 0 \n"
615         "ut_data/app2 -> ut_data/app1/mod1 : 4 ut_data/app1/mod1/mod1_1.hpp ut_data/app1/mod1/mod1_2.hpp\n"
616         "\n";
617     // clang-format on
618
619     auto ss = std::ostringstream{};
620
621     p2pg.Output(ss);
622     ASSERT_EQ(exp, ss.str());
623 }
624 {
625     auto p2pg = Pkg2PkgGenerator{"ut_data/load_store/pkg_org", false, false,
626                                 Paths_t{"ut_data/app3/"}, ""};
627
628     // clang-format off
629     auto exp = std::string {
630         "#deps\n"
631         "ut_data/app1 -> ut_data/app1/mod1 : 10 ut_data/app1/mod1/mod1_1.hpp ut_data/app1/mod1/mod1_2.hpp\n"
632         "ut_data/app1/mod1 -> ut_data/app1 : 3 ut_data/app1/a_1.cpp.h ut_data/app1/mod2/mod2_1.hpp
633         ut_data/app1/mod2/mod2_2/mod2_2_1.h\n"
634         "\n"
635         "ut_data/app1 -> ut_data/app1/mod2 : 0 \n"
636         "ut_data/app1/mod2 -> ut_data/app1 : 0 \n"
637         "\n"
638         "ut_data/app1 -> ut_data/app2/mod2_1 : 5 ut_data/app1/mod2/mod2_1/mod2_1_1.h\n"
639         "ut_data/app1/mod2/mod2_1 -> ut_data/app1 : 2 ut_data/app1/a_1.cpp.h ut_data/app1/mod2/mod2_2/mod2_2_1.h\n"
640         "\n"
641         "ut_data/app1/mod1 -> ut_data/app1/mod2 : 0 \n"
642         "ut_data/app1/mod2 -> ut_data/app1/mod1 : 0 \n"
643         "\n"
644         "ut_data/app1/mod1 -> ut_data/app1/mod2/mod2_1 : 1 ut_data/app1/mod2/mod2_1/mod2_1_1.h\n"
645         "ut_data/app1/mod2/mod2_1 -> ut_data/app1/mod1 : 2 ut_data/app1/mod1/mod1_1.hpp
646         ut_data/app1/mod1/mod1_2.hpp\n"
647         "\n"
648         "ut_data/app1/mod1 -> ut_data/app2 : 0 \n"
649         "ut_data/app2 -> ut_data/app1/mod1 : 4 ut_data/app1/mod1_1.hpp ut_data/app1/mod1/mod1_2.hpp\n"
650         "\n"
651         "ut_data/app1/mod2 -> ut_data/app1/mod2/mod2_1 : 0 \n"
652         "ut_data/app1/mod2/mod2_1 -> ut_data/app1/mod2 : 0 \n"
653         "\n"
654         "ut_data/app1/mod2 -> ut_data/app2 : 0 \n"
655         "ut_data/app2 -> ut_data/app1/mod2 : 0 \n"
656         "\n"
657         "ut_data/app1/mod2/mod2_1 -> ut_data/app2 : 0 \n"
658         "ut_data/app2 -> ut_data/app1/mod2/mod2_1 : 2 ut_data/app1/mod2/mod2_1/mod2_1_1.h\n"
659         "\n";
660     // clang-format on
661
662     auto ss = std::ostringstream{};
663
664     p2pg.Output(ss);
665     ASSERT_EQ(exp, ss.str());

```

```

665     }
666     {
667         auto p2pg = Pkg2PkgGenerator{"ut_data/load_store/pkg_org", false, false,
668                                     Paths_t{"ut_data/app3/"}, "ut_data/app2"};
669
670         // clang-format off
671         auto exp = std::string {
672             "#deps\n"
673             "no_package -> ut_data/app1 : 5 ut_data/app1/a_1_cpp.h ut_data/app1/a_2_cpp.hpp
674             ut_data/app1/mod2/mod2_2/mod2_2_1.h\n"
675             "ut_data/app1 -> no_package : 0 \n"
676             "\n"
677             "no_package -> ut_data/app1/mod1 : 4 ut_data/app1/mod1/mod1_1.hpp ut_data/app1/mod1/mod1_2.hpp\n"
678             "ut_data/app1/mod1 -> no_package : 0 \n"
679             "\n"
680             "no_package -> ut_data/app1/mod2 : 0 \n"
681             "ut_data/app1/mod2 -> no_package : 0 \n"
682             "\n"
683             "no_package -> ut_data/app1/mod2/mod2_1 : 2 ut_data/app1/mod2/mod2_1/mod2_1_1.h\n"
684             "ut_data/app1/mod2/mod2_1 -> no_package : 0 \n"
685             "\n"
686             "ut_data/app1 -> ut_data/app1/mod1 : 10 ut_data/app1/mod1/mod1_1.hpp ut_data/app1/mod1/mod1_2.hpp\n"
687             "ut_data/app1/mod1 -> ut_data/app1 : 3 ut_data/app1/a_1_cpp.h ut_data/app1/mod2/mod2_1.hpp
688             ut_data/app1/mod2/mod2_2/mod2_2_1.h\n"
689             "\n"
690             "ut_data/app1 -> ut_data/app1/mod2 : 0 \n"
691             "ut_data/app1/mod2 -> ut_data/app1 : 0 \n"
692             "\n"
693             "ut_data/app1 -> ut_data/app1/mod2/mod2_1 : 5 ut_data/app1/mod2/mod2_1/mod2_1_1.h\n"
694             "ut_data/app1/mod2/mod2_1 -> ut_data/app1 : 2 ut_data/app1/a_1_cpp.h ut_data/app1/mod2/mod2_2/mod2_2_1.h\n"
695             "\n"
696             "ut_data/app1/mod1 -> ut_data/app1/mod2 : 0 \n"
697             "ut_data/app1/mod2 -> ut_data/app1/mod1 : 0 \n"
698             "\n"
699             "ut_data/app1/mod1 -> ut_data/app1/mod2/mod2_1 : 1 ut_data/app1/mod2/mod2_1/mod2_1_1.h\n"
700             "ut_data/app1/mod2/mod2_1 -> ut_data/app1/mod1 : 2 ut_data/app1/mod1/mod1_1.hpp
701             ut_data/app1/mod1_2.hpp\n"
702             "\n"
703             "ut_data/app1/mod2 -> ut_data/app1/mod2/mod2_1 : 0 \n"
704             "ut_data/app1/mod2_1 -> ut_data/app1/mod2 : 0 \n"
705             "\n";
706             // clang-format on
707
708             auto ss = std::ostringstream{};
709
710             p2pg.Output(ss);
711             ASSERT_EQ(exp, ss.str());
712         }
713     }
714     TEST(deps_scenario, ArchGenerator)
715     {
716         using FileUtils::Paths_t;
717
718         auto exception_occured = false;
719         try {
720             auto ag = ArchGenerator{"ut_data/load_store/arch_org"};
721         }
722         catch (std::runtime_error const& e) {
723             exception_occured = true;
724             ASSERT_STREQ("IN-file load error", e.what());
725         }
726         ASSERT_TRUE(exception_occured);
727
728         {
729             auto ag = ArchGenerator{"ut_data/load_store/deps_org"};
730             auto ss = std::stringstream{};
731
732             ag.Output(ss);
733
734             auto act = std::optional<std::vector<std::string>>{FileUtils::Load.Strings(ss)};
735             auto exp = std::optional<std::vector<std::string>>{
736                 FileUtils::LoadFromFile("ut_data/load_store/arch_org", FileUtils::Load.Strings)};
737
738             ASSERT_TRUE(exp);
739             ASSERT_EQ(*act, *exp);
740         }
741     }
742 }
```

```

743 TEST(deps_scenario, Arch2PUmlGenerator)
744 {
745     auto exception_occured = false;
746     try {
747         auto ag = Arch2PUmlGenerator{"ut_data/load_store/arch_org"};
748     }
749     catch (std::runtime_error const& e) {
750         exception_occured = true;
751         ASSERT_STREQ("IN-file load error", e.what());
752     }
753     ASSERT_TRUE(exception_occured);
754
755     {
756         auto ag = Arch2PUmlGenerator{"ut_data/load_store/deps_org"};
757         auto ss = std::stringstream{};
758
759         ag.Output(ss);
760
761         auto exp = std::string{
762             "@startuml\n"
763             "scale max 730 width\n"
764             "rectangle \"app1\" as ut_data__app1 {\n"
765                 "    rectangle \"mod1\" as ut_data__app1__mod1\n"
766                 "    rectangle \"mod2\" as ut_data__app1__mod2 {\n"
767                     "        rectangle \"mod2_1\" as ut_data__app1__mod2__mod2_1\n"
768                     "        rectangle \"mod2_2\" as ut_data__app1__mod2__mod2_2\n"
769                 }\n"
770             "}\n"
771             "rectangle \"app2\" as ut_data__app2\n"
772             "\n"
773             "ut_data__app1 \"2\" -[#orange]-> ut_data__app1__mod1\n"
774             "ut_data__app1__mod2__mod2_2 \"1\" -[#orange]-> ut_data__app1\n"
775             "ut_data__app2 \"1\" -[#green]-> ut_data__app1\n"
776             "ut_data__app1__mod1 \"1\" -[#green]-> ut_data__app1__mod2\n"
777             "ut_data__app1__mod1 \"1\" -[#orange]-> ut_data__app1__mod2__mod2_2\n"
778             "ut_data__app2 \"2\" -[#green]-> ut_data__app1__mod1\n"
779             "ut_data__app1__mod2__mod2_1 \"1\" <-[#red]-> \"2\" ut_data__app1__mod2__mod2_2\n"
780             "\n"
781             "@enduml\n"};
782
783         ASSERT_EQ(exp, ss.str());
784     }
785 }
786
787 TEST(deps_scenario, CyclicGenerator)
788 {
789     auto exception_occured = false;
790     try {
791         auto cg = CyclicGenerator{"ut_data/load_store/arch_org"};
792     }
793     catch (std::runtime_error const& e) {
794         exception_occured = true;
795         ASSERT_STREQ("IN-file load error", e.what());
796     }
797     ASSERT_TRUE(exception_occured);
798
799     {
800         auto cg = CyclicGenerator{"ut_data/load_store/deps_org"};
801         auto ss = std::stringstream{};
802
803         ASSERT_FALSE(cg.Output(ss));
804
805         ASSERT_EQ("cyclic dependencies found\n", ss.str());
806     }
807     {
808         auto cg = CyclicGenerator{"ut_data/load_store/deps_org2"};
809         auto ss = std::stringstream{};
810
811         ASSERT_TRUE(cg.Output(ss));
812
813         ASSERT_EQ("cyclic dependencies not found\n", ss.str());
814     }
815 }
816 } // namespace
817 } // namespace Dependency

```

exampledepsdependencyutload_store_format_ut.cpp

```

1 #include "gtest_wrapper.h"
2

```

```

3 #include "file_utils/load_store.h"
4 #include "file_utils/load_store_row.h"
5 #include "load_store_format.h"
6
7 namespace Dependency {
8 namespace {
9
10 TEST(load_store_format, Paths_t)
11 {
12     using FileUtils::Paths_t;
13
14     auto const pkg_org = std::string("ut_data/load_store/pkg_org");
15     auto const pkg_act = std::string("ut_data/load_store/pkg_act");
16
17     FileUtils::RemoveFile(pkg_act);
18
19     auto const dir_in
20         = Paths_t{"ut_data/app1", "ut_data/app1/mod1", "ut_data/app1/mod2/mod2_1", "ut_data/app2"};
21
22     // ディレクトリなのでエラーなはず
23     ASSERT_FALSE(FileUtils::StoreToFile("ut_data/app1", dir_in, StoreToStream));
24
25     ASSERT_TRUE(FileUtils::StoreToFile(pkg_act, dir_in, StoreToStream));
26
27     auto dir_out0 = std::optional<Paths_t>{FileUtils::LoadFromFile(pkg_org, Load_Paths)};
28     ASSERT_TRUE(dir_out0);
29
30     auto dir_out1 = std::optional<Paths_t>{FileUtils::LoadFromFile(pkg_act, Load_Paths)};
31     ASSERT_TRUE(dir_out1);
32
33     ASSERT_EQ(dir_in, *dir_out0);
34     ASSERT_EQ(dir_in, *dir_out1);
35
36     FileUtils::RemoveFile(pkg_act);
37 }
38
39 TEST(load_store_format, Dirs2Srcs_t)
40 {
41     auto const dirs2srcs_org = std::string("ut_data/load_store/dirs2srcs_org");
42     auto const dirs2srcs_act = std::string("ut_data/load_store/dirs2srcs_act");
43
44     FileUtils::RemoveFile(dirs2srcs_act);
45
46     const auto dir2srcs_in = FileUtils::Dirs2Srcs_t{
47         {"ut_data",
48             {"ut_data/app1/a_1_c.c", "ut_data/app1/a_1_c.h", "ut_data/app1/a_1_cpp.cpp",
49                 "ut_data/app1/a_1_cpp.h", "ut_data/app1/a_2_c.C", "ut_data/app1/a_2_C.H",
50                     "ut_data/app1/a_2_cpp.cxx", "ut_data/app1/a_2_cpp.hpp", "ut_data/app1/a_3_cpp.cc",
51                         "ut_data/app1/mod1/mod1_1.cpp", "ut_data/app1/mod1/mod1_1.hpp",
52                             "ut_data/app1/mod1/mod1_2.hpp"}},
53             {"ut_data/app1/mod2", {"ut_data/app1/mod2/mod2_1.cpp", "ut_data/app1/mod2/mod2_1.hpp"}},
54             {"ut_data/app1/mod2/mod2_1",
55                 {"ut_data/app1/mod2/mod2_1/mod2_1_1.cpp", "ut_data/app1/mod2/mod2_1/mod2_1_1.h"}},
56                 {"ut_data/app1/mod2/mod2_2",
57                     {"ut_data/app1/mod2/mod2_2/mod2_2_1.cpp", "ut_data/app1/mod2/mod2_2/mod2_2_1.h"}},
58                     {"ut_data/app2", {"ut_data/app2/b_1.cpp", "ut_data/app2/b_1.h}}},
59     };
60
61     // ディレクトリなのでエラーなはず
62     ASSERT_FALSE(FileUtils::StoreToFile("ut_data/app1", dir2srcs_in, StoreToStream));
63
64     ASSERT_TRUE(FileUtils::StoreToFile(dirs2srcs_act, dir2srcs_in, StoreToStream));
65
66     auto dir2srcs_out0 = std::optional<FileUtils::Dirs2Srcs_t>{
67         FileUtils::LoadFromFile(dirs2srcs_org, Load_Dirs2Srcs)};
68     ASSERT_TRUE(dir2srcs_out0);
69
70     auto dir2srcs_out1 = std::optional<FileUtils::Dirs2Srcs_t>{
71         FileUtils::LoadFromFile(dirs2srcs_act, Load_Dirs2Srcs)};
72     ASSERT_TRUE(dir2srcs_out1);
73
74     ASSERT_EQ(dir2srcs_in, *dir2srcs_out0);
75     ASSERT_EQ(dir2srcs_in, *dir2srcs_out1);
76
77     FileUtils::RemoveFile(dirs2srcs_act);
78 }
79
80 namespace {
81 using FileUtils::Paths_t;
82

```

```

83 DepRels_t const dep_rels{
84     {DepRelation{"ut_data/app1", 2,
85         Paths_t{"ut_data/app1/mod1/mod1_1.hpp", "ut_data/app1/mod1/mod1_2.hpp"}, 
86         "ut_data/app1/mod1", 0, Paths_t{}}, 
87     {DepRelation{"ut_data/app1", 0, Paths_t{}, "ut_data/app1/mod2", 0, Paths_t{}}, 
88     {DepRelation{"ut_data/app1", 0, Paths_t{}, "ut_data/app1/mod2/mod2_1", 0, Paths_t{}}, 
89     {DepRelation{"ut_data/app1", 0, Paths_t{}, "ut_data/app1/mod2/mod2_2", 1,
90         Paths_t{"ut_data/app1/a_1.cpp.h"}}, 
91     {DepRelation{"ut_data/app1", 0, Paths_t{}, "ut_data/app2", 1,
92         Paths_t{"ut_data/app1/a_2.cpp.hpp"}}, 
93     {DepRelation{"ut_data/app1/mod1", 1, Paths_t{"ut_data/app1/mod2/mod2_1.hpp"}, 
94         "ut_data/app1/mod2", 0, Paths_t{}}, 
95     {DepRelation{"ut_data/app1/mod1", 0, Paths_t{}, "ut_data/app1/mod2/mod2_1", 0, Paths_t{}}, 
96     {DepRelation{"ut_data/app1/mod1", 1, Paths_t{"ut_data/app1/mod2/mod2_2/mod2_2_1.h"}, 
97         "ut_data/app1/mod2/mod2_2", 0, Paths_t{}}, 
98     {DepRelation{"ut_data/app1/mod1", 0, Paths_t{}, "ut_data/app2", 2,
99         Paths_t{"ut_data/app1/mod1_1.hpp", "ut_data/app1/mod1/mod1_2.hpp"}}, 
100    {DepRelation{"ut_data/app1/mod2", 0, Paths_t{}, "ut_data/app1/mod2/mod2_1", 0, Paths_t{}}, 
101    {DepRelation{"ut_data/app1/mod2", 0, Paths_t{}, "ut_data/app1/mod2/mod2_2", 0, Paths_t{}}, 
102    {DepRelation{"ut_data/app1/mod2", 0, Paths_t{}, "ut_data/app2", 0, Paths_t{}}, 
103    {DepRelation{"ut_data/app1/mod2/mod2_1", 1, Paths_t{"ut_data/app1/mod2/mod2_2/mod2_2_1.h"}, 
104        "ut_data/app1/mod2/mod2_2", 2, Paths_t{"ut_data/app1/mod2/mod2_1/mod2_1.h"}}, 
105    {DepRelation{"ut_data/app1/mod2/mod2_1", 0, Paths_t{}, "ut_data/app2", 0, Paths_t{}}, 
106    {DepRelation{"ut_data/app1/mod2/mod2_2", 0, Paths_t{}, "ut_data/app2", 0, Paths_t{}}, 
107 }; 
108 } // namespace 
109 
110 TEST(load_store_format, DepRels_t) 
111 { 
112     auto const deps_org = std::string("ut_data/load_store/deps_org"); 
113     auto const deps_act = std::string("ut_data/load_store/deps_act"); 
114 
115     FileUtils::RemoveFile(deps_act); 
116 
117     // ディレクトリなのでエラーなはず 
118     ASSERT_FALSE(FileUtils::StoreToFile("ut_data/app1", dep_rels, StoreToString)); 
119 
120     ASSERT_TRUE(FileUtils::StoreToFile(deps_act, dep_rels, StoreToString)); 
121 
122     auto deps_out0 = std::optional<DepRels_t>{FileUtils::LoadFromFile(deps_org, Load_DepRels)}; 
123     ASSERT_TRUE(deps_out0); 
124 
125     auto exp_it      = dep_rels.cbegin(); 
126     auto exp_it_end = dep_rels.cend(); 
127     auto act_it      = deps_out0->cbegin(); 
128 
129     while (exp_it != exp_it_end) { 
130         auto exp_str = ToStringDepRel(*exp_it); 
131         auto act_str = ToStringDepRel(*act_it); 
132         ASSERT_EQ(exp_str, act_str); 
133         ASSERT_EQ(exp_it->PackageA, act_it->PackageA); 
134         ASSERT_EQ(exp_it->CountAtoB, act_it->CountAtoB); 
135         ASSERT_EQ(exp_it->IncsAtoB, act_it->IncsAtoB); 
136         ASSERT_EQ(exp_it->PackageB, act_it->PackageB); 
137         ASSERT_EQ(exp_it->CountBtoA, act_it->CountBtoA); 
138         ASSERT_EQ(exp_it->IncsBtoA, act_it->IncsBtoA); 
139         ASSERT_EQ(*exp_it, *act_it); 
140 
141         ++exp_it; 
142         ++act_it; 
143     } 
144     ASSERT_EQ(dep_rels, *deps_out0); 
145 
146     auto deps_out1 = std::optional<DepRels_t>{FileUtils::LoadFromFile(deps_act, Load_DepRels)}; 
147     ASSERT_TRUE(deps_out1); 
148 
149     ASSERT_EQ(dep_rels, *deps_out1); 
150 
151     FileUtils::RemoveFile(deps_act); 
152 } 
153 
154 TEST(load_store_format, Arch_t) 
155 { 
156     auto const arch_exp = std::string("ut_data/load_store/arch_org"); 
157     auto const arch_act = std::string("ut_data/load_store/arch_act"); 
158 
159     FileUtils::RemoveFile(arch_act); 
160 
161     auto row_exp = std::optional<std::vector<std::string>>{ 
162         FileUtils::LoadFromFile(arch_exp, FileUtils::Load_Strings)}; 

```

```

163     ASSERT_TRUE(row_exp);
164
165     auto const arch = ArchPkg::GenArch(dep_rels);
166     ASSERT_TRUE(FileUtils::StoreToFile(arch_act, arch, StoreToStream));
167
168     auto row_act = std::optional<std::vector<std::string>>{
169         FileUtils::LoadFromFile(arch_act, FileUtils::Load_Strings)};
170     ASSERT_TRUE(row_act);
171
172     ASSERT_EQ(row_exp, *row_act);
173
174     FileUtils::RemoveFile(arch_act);
175 }
176 } // namespace
177 } // namespace Dependency

```

exampledeps/file_utils/h/file_utils/load_store.h

```

1 #pragma once
2 #include <fstream>
3 #include <optional>
4 #include <string>
5
6 namespace FileUtils {
7
8 template <typename T>
9 bool StoreToFile(std::string_view filename, T const& t, bool (*ss)(std::ostream& os, T const&))
10 {
11     auto fout = std::ofstream{filename.data()};
12
13     if (!fout) {
14         return false;
15     }
16
17     return (*ss)(fout, t);
18 }
19
20 template <typename T>
21 std::optional<T> LoadFromFile(std::string_view filename, std::optional<T> (*ls)(std::istream& os))
22 {
23     auto fin = std::ifstream{filename.data()};
24
25     if (!fin) {
26         return std::nullopt;
27     }
28
29     return (*ls)(fin);
30 }
31 } // namespace FileUtils

```

exampledeps/file_utils/h/file_utils/load_store_row.h

```

1 #pragma once
2 #include <fstream>
3 #include <optional>
4 #include <utility>
5 #include <vector>
6
7 namespace FileUtils {
8
9 bool StoreToString(std::ostream& os, std::vector<std::string> const& lines);
10 std::optional<std::vector<std::string>> Load_Strings(std::istream& is);
11 } // namespace FileUtils

```

exampledeps/file_utils/h/file_utils/path_utils.h

```

1 #pragma once
2 #include <filesystem>
3 #include <fstream>
4 #include <list>
5 #include <map>
6 #include <string>
7
8 namespace FileUtils {
9
10 using Path_t = std::filesystem::path;
11 std::string ToStringPath(Path_t const& paths);
12

```

```

13 using Paths_t = std::list<std::filesystem::path>;
14
15 Paths_t           NotDirs(Paths_t const& dirs);
16 std::string        ToStringPaths(Paths_t const& paths, std::string_view sep = "\n",
17                                std::string_view indent = "");
18 inline std::ostream& operator<<(std::ostream& os, Paths_t const& paths)
19 {
20     return os << ToStringPaths(paths);
21 }
22
23 // first path: filename
24 // second path: pathname
25 using Filename2Path_t = std::map<Path_t, Path_t>;
26 Filename2Path_t GenFilename2Path(Paths_t const& paths);
27
28 // first : package name(directory name)
29 // second : srcs assigned to package
30 using Dirs2Srcs_t = std::map<Path_t, Paths_t>;
31
32 Dirs2Srcs_t       AssginSrcsToDirs(Paths_t const& dirs, Paths_t const& srcs);
33 std::string        ToStringDirs2Srcs(Dirs2Srcs_t const& dirs2srcs);
34 inline std::ostream& operator<<(std::ostream& os, Dirs2Srcs_t const& dirs2srcs)
35 {
36     return os << ToStringDirs2Srcs(dirs2srcs);
37 }
38
39 Path_t NormalizeLexically(Path_t const& path);
40
41 void RemoveFile(Path_t const& filename);
42 } // namespace FileUtils

```

exampledeps/file_utils/src/load_store.cpp

```

1 #include <cassert>
2 #include <iostream>
3 #include <optional>
4 #include <regex>
5
6 #include "file_utils/load_store.h"
7 #include "file_utils/load_store_row.h"
8
9 namespace FileUtils {
10
11 bool StoreToStream(std::ostream& os, std::vector<std::string> const& lines)
12 {
13     for (auto const& line : lines) {
14         os << line;
15     }
16
17     return true;
18 }
19
20 std::optional<std::vector<std::string>> Load_Strings(std::istream& is)
21 {
22     auto content = std::vector<std::string>{};
23     auto line    = std::string{};
24
25     while (std::getline(is, line)) {
26         auto ss = std::ostringstream{};
27
28         ss << line << std::endl;
29         content.emplace_back(ss.str());
30     }
31
32     return content;
33 }
34 } // namespace FileUtils

```

exampledeps/file_utils/src/path_utils.cpp

```

1 #include <algorithm>
2 #include <sstream>
3 #include <utility>
4
5 #include "file_utils/path_utils.h"
6
7 namespace FileUtils {
8
9 std::string ToStringPath(Path_t const& path)

```

```

10 {
11     auto pn = path.string();
12
13     if (pn.size() == 0) {
14         pn = "\"\"";
15     }
16
17     return pn;
18 }
19
20 std::string ToStringPaths(Paths_t const& paths, std::string_view sep, std::string_view indent)
21 {
22     auto ss    = std::ostringstream{};
23     auto first = true;
24
25     for (auto const& p : paths) {
26         if (!std::exchange(first, false)) {
27             ss << sep;
28         }
29
30         ss << indent << ToStringPath(p);
31     }
32
33     return ss.str();
34 }
35
36 std::string ToStringDirs2Srcs(Dirs2Srcs_t const& dirs2srcs)
37 {
38     auto ss    = std::ostringstream{};
39     auto first = bool{true};
40
41     for (auto const& pair : dirs2srcs) {
42         if (first) {
43             first = false;
44         }
45         else {
46             ss << std::endl;
47         }
48
49         ss << ToStringPath(pair.first) << std::endl;
50         ss << ToStringPaths(pair.second, "\n", "    ") << std::endl;
51     }
52
53     return ss.str();
54 }
55
56 Paths_t NotDirs(Paths_t const& dirs)
57 {
58     auto ret = Paths_t{};
59
60     std::copy_if(dirs.cbegin(), dirs.cend(), std::back_inserter(ret),
61                  [] (auto const& dir) noexcept { return !std::filesystem::is_directory(dir); });
62
63     return ret;
64 }
65
66 Filename2Path_t GenFilename2Path(Paths_t const& paths)
67 {
68     auto ret = Filename2Path_t {};
69
70     for (auto const& p : paths) {
71         ret[p.filename()] = p;
72     }
73
74     return ret;
75 }
76
77 namespace {
78 Path_t const current_dir{".";
79
80 size_t match_count(Path_t const& dir, Path_t const& src)
81 {
82     auto const dir_str = dir.string();
83     auto const src_str = dir == current_dir ? "./" + src.string() : src.string();
84
85     if (dir_str.size() >= src_str.size()) {
86         return 0;
87     }
88
89     auto count      = 0U;

```

```

90     auto count_max = dir_str.size();
91
92     for (; count < count_max; ++count) {
93         if (dir_str[count] != src_str[count]) {
94             break;
95         }
96     }
97
98     if (count == count_max && src_str[count] == '/') {
99         return count;
100    }
101
102    return 0;
103 }
104
105 Path_t select_package(Path_t const& src, Paths_t const& dirs)
106 {
107     Path_t const* best_match=nullptr;
108     auto count_max = 0U;
109
110     for (auto const& dir : dirs) {
111         auto count = match_count(dir, src);
112         if (count_max < count) {
113             best_match = &dir;
114             count_max = count;
115         }
116     }
117
118     if (best_match == nullptr) {
119         return Path_t("no_package");
120     }
121     else {
122         return *best_match;
123     }
124 }
125
126 Paths_t gen_parent_dirs(Path_t const dir)
127 {
128     auto ret = Paths_t{};
129
130     for (auto p = dir.parent_path(), pp = p.parent_path(); !p.empty() && p != pp;
131         p = pp, pp = p.parent_path()) {
132         ret.push_front(p);
133     }
134
135     return ret;
136 }
137
138 // a/
139 //   a0.c
140 //     b/
141 //       c/
142 //         d/
143 //           d.c
144 // のようなファイル構造があった場合、
145 // d2sには a、a/b/c/d が登録され、a/b、a/b/cは登録されていない。
146 // a/b、a/b/cを埋めるのがpad_parent_dirsである。
147 void pad_parent_dirs(Paths_t const& dirs, Dirs2Srcs_t& d2s)
148 {
149     for (auto const& dir : dirs) {
150         auto parent_found = false;
151
152         for (auto const& p : gen_parent_dirs(dir)) {
153             if (!parent_found && d2s.count(p) != 0) {
154                 parent_found = true;
155             }
156             else if (parent_found && d2s.count(p) == 0) {
157                 d2s[p] = Paths_t();
158             }
159         }
160     }
161 }
162 } // namespace
163
164 Dirs2Srcs_t AssginSrcsToDirs(Paths_t const& dirs, Paths_t const& srcs)
165 {
166     auto ret      = Dirs2Srcs_t{};
167     auto add_dirs = Paths_t{};
168
169     for (auto const& src : srcs) {

```

```

170     auto dir = select_package(src, dirs);
171
172     if (ret.count(dir) == 0) {
173         ret[dir] = Paths_t();
174         add_dirs.push_back(dir);
175     }
176     ret[dir].push_back(src);
177 }
178
179 pad_parent_dirs(add_dirs, ret);
180
181 return ret;
182 }
183
184 Path_t NormalizeLexically(Path_t const& path)
185 {
186     // lexically_normalは"a/..../b"を"b"にする
187     // 最後の'/'を削除
188     auto path_lex = Path_t(path.string() + '/').lexically_normal().string();
189     path_lex.pop_back();
190
191     if (path_lex.size() == 0) {
192         return Path_t(".");
193     }
194     return path_lex;
195 }
196
197 void RemoveFile(Path_t const& filename)
198 {
199     if (std::filesystem::exists(filename)) {
200         std::filesystem::remove(filename);
201     }
202 }
203 } // namespace FileUtils

```

exampledeps/file_utils/ut/load_store_ut.cpp

```

1 #include "gtest_wrapper.h"
2
3 #include "file_utils/load_store.h"
4 #include "file_utils/load_store_row.h"
5 #include "file_utils/path_utils.h"
6
7 namespace FileUtils {
8 namespace {
9
10 TEST(load_store, Row)
11 {
12     auto const row_exp = std::string{"ut_data/load_store/pkg_org"};
13     auto const row_act = std::string{"ut_data/load_store/pkg_act"};
14
15     RemoveFile(row_act);
16
17     auto row_data0 = std::optional<std::vector<std::string>>{LoadFromFile(row_act, Load_Strings)};
18
19     // row_actはないのでエラーなはず
20     ASSERT_FALSE(row_data0);
21
22     // ディレクトリなのでエラーなはず
23     ASSERT_FALSE(StoreToFile("ut_data/app1", *row_data0, StoreToStream));
24
25     row_data0 = LoadFromFile(row_exp, Load_Strings);
26     ASSERT_TRUE(row_data0);
27     ASSERT_TRUE(StoreToFile(row_act, *row_data0, StoreToStream));
28
29     auto row_data1 = std::optional<std::vector<std::string>>{LoadFromFile(row_act, Load_Strings)};
30     ASSERT_TRUE(row_data1);
31
32     ASSERT_EQ(*row_data0, *row_data1);
33
34     RemoveFile(row_act);
35 }
36 } // namespace
37 } // namespace FileUtils

```

exampledeps/file_utils/ut/path_utils_ut.cpp

```

1 #include "gtest_wrapper.h"
2

```

```

3 #include "file_utils/load_store.h"
4 #include "file_utils/load_store_row.h"
5 #include "file_utils/path_utils.h"
6 #include "logging/logger.h"
7
8 #define SCAN_BUILD_ERROR 0
9
10 #if SCAN_BUILD_ERROR == 1
11 struct X {};
12 void potential_leak(int a)
13 {
14     X* x{new X};
15
16     if (a == 2) { // aが2ならメモリリーク
17         return;
18     }
19
20     delete x;
21 }
22 #endif
23
24 namespace FileUtils {
25 namespace {
26
27 TEST(path_utils, Logger)
28 {
29     auto log_file_org = "ut_data/load_store/logger_org";
30     auto log_file_act = "ut_data/load_store/logger_act";
31
32     RemoveFile(log_file_act);
33
34     LOGGER_INIT(log_file_act);
35     LOGGER(1);
36     LOGGER("xyz", 3, 5);
37
38     auto const dirs = Paths_t{"ut_data/app1",
39                             "ut_data/app1/mod1",
40                             "ut_data/app1/mod2",
41                             "ut_data/app1/mod2/mod2_1",
42                             "ut_data/app1/mod2/mod2_2",
43                             "ut_data/app2"};
44
45     LOGGER(ToStringPaths(dirs));
46     LOGGER(dirs);
47
48     Logging::Logger::Inst().Close();
49
50     auto exp = std::optional<std::vector<std::string>>{LoadFromFile(log_file_org, Load_Strings)};
51     ASSERT_TRUE(exp);
52
53     auto act = std::optional<std::vector<std::string>>{LoadFromFile(log_file_act, Load_Strings)};
54     ASSERT_TRUE(act);
55
56     ASSERT_EQ(*exp, *act);
57
58     RemoveFile(log_file_act);
59 }
60
61 TEST(path_utils, NotDirs)
62 {
63     {
64         auto const dir_in = Paths_t{"ut_data/app1",
65                             "ut_data/app1/mod1",
66                             "ut_data/app1/mod2",
67                             "ut_data/app1/mod2/mod2_1",
68                             "ut_data/app1/mod2/mod2_2",
69                             "ut_data/app2"};
70         auto const dir_act = Paths_t{NotDirs(dir_in)};
71
72         ASSERT_EQ(Paths_t{}, dir_act);
73     }
74     {
75         auto const dir_in = Paths_t{"ut_data/app1",
76                             "ut_data/app1/notdir",
77                             "ut_data/notdir2",
78                             "ut_data/app1/mod2/mod2_1",
79                             "ut_data/app1/mod2/mod2_2",
80                             "ut_data/app2"};
81         auto const dir_act = Paths_t{NotDirs(dir_in)};
82         auto const dir_exp = Paths_t{

```

```

83         "ut_data/app1/notdir",
84         "ut_data/notdir2",
85     );
86
87     ASSERT_EQ(dir_exp, dir_act);
88 }
89
90
91 TEST(path_utils, NormalizeLexically)
92 {
93     // こうなるのでNormalizeLexicallyが必要
94     ASSERT_EQ(Path_t("a"), Path_t("a"));
95     ASSERT_NE(Path_t("a/"), Path_t("a"));
96     ASSERT_EQ("a/", Path_t("x...a/").lexically_normal().string());
97     ASSERT_EQ("a", Path_t("x.../a").lexically_normal().string());
98
99     // テストはここから
100    ASSERT_EQ("a", NormalizeLexically("x.../a/").string());
101    ASSERT_EQ("a", NormalizeLexically("./x.../a/").string());
102    ASSERT_EQ("../a", NormalizeLexically(".././x.../a/").string());
103    ASSERT_EQ("../a", NormalizeLexically(".././x.../a///").string());
104    ASSERT_EQ("../a", NormalizeLexically(".././x.../a././").string());
105
106    ASSERT_EQ("a", NormalizeLexically(Path_t("x.../a/")).string());
107
108    ASSERT_EQ(".", NormalizeLexically(Path_t("./")).string());
109    ASSERT_EQ(".", NormalizeLexically(Path_t(".")).string());
110 }
111
112 TEST(path_utils, GenFilename2Path)
113 {
114     auto const act_srcs
115         = Paths_t{"ut_data/app1/a_1_c.c", "ut_data/app1/a_1_c.h", "ut_data/app1/a_1_cpp.cpp",
116             "ut_data/app1/a_1_cpp.h", "ut_data/app1/a_2_c.C"};
117
118     auto const act = GenFilename2Path(act_srcs);
119
120     auto const exp = Filename2Path_t{
121         {"a_1_c.c", "ut_data/app1/a_1_c.c"}, {"a_1_c.h", "ut_data/app1/a_1_c.h"},
122         {"a_1_cpp.cpp", "ut_data/app1/a_1_cpp.cpp"}, {"a_1_cpp.h", "ut_data/app1/a_1_cpp.h"},
123         {"a_2_c.C", "ut_data/app1/a_2_c.C"}, };
124 }
125
126     ASSERT_EQ(act, exp);
127 }
128
129 TEST(path_utils, AssginSrcsToDirs)
130 {
131     {
132         auto const exp_dirs = Paths_t{"ut_data/app1",
133             "ut_data/app1/mod1",
134             "ut_data/app1/mod2",
135             "ut_data/app1/mod2/mod2_1",
136             "ut_data/app1/mod2/mod2_2",
137             "ut_data/app2"};
138
139         auto const exp_srcs = Paths_t{"ut_data/app1/a_1_c.c",
140             "ut_data/app1/a_1_c.h",
141             "ut_data/app1/a_1_cpp.cpp",
142             "ut_data/app1/a_1_cpp.h",
143             "ut_data/app1/a_2_c.C",
144             "ut_data/app1/a_2_c.H",
145             "ut_data/app1/a_2_cpp.cxx",
146             "ut_data/app1/a_2_cpp.hpp",
147             "ut_data/app1/a_3_cpp.cc",
148             "ut_data/app1/mod1/mod1_1.cpp",
149             "ut_data/app1/mod1/mod1_1.hpp",
150             "ut_data/app1/mod1/mod1_2.hpp",
151             "ut_data/app1/mod2/mod2_1.cpp",
152             "ut_data/app1/mod2/mod2_1.hpp",
153             "ut_data/app1/mod2/mod2_1/mod2_1_1.cpp",
154             "ut_data/app1/mod2/mod2_1/mod2_1_1.h",
155             "ut_data/app1/mod2/mod2_2/mod2_2_1.cpp",
156             "ut_data/app1/mod2/mod2_2/mod2_2_1.h",
157             "ut_data/app2/b_1.cpp",
158             "ut_data/app2/b_1.h"};
159
160         auto const act = AssginSrcsToDirs(exp_dirs, exp_srcs);
161
162         auto const exp = Dirs2Srcs_t{
```

```

163     {"ut_data/app1",
164      {"ut_data/app1/a_1_c.c", "ut_data/app1/a_1_c.h", "ut_data/app1/a_1_cpp.cpp",
165       "ut_data/app1/a_1_cpp.h", "ut_data/app1/a_2_c.C", "ut_data/app1/a_2_c.H",
166       "ut_data/app1/a_2_cpp.cxx", "ut_data/app1/a_2_cpp.hpp", "ut_data/app1/a_3_cpp.cc"}},
167     {"ut_data/app1/mod1",
168      {"ut_data/app1/mod1/mod1_1.cpp", "ut_data/app1/mod1/mod1_1.hpp",
169       "ut_data/app1/mod1/mod1_2.hpp"}},
170     {"ut_data/app1/mod2", {"ut_data/app1/mod2/mod2_1.cpp", "ut_data/app1/mod2/mod2_1.hpp"}},
171     {"ut_data/app1/mod2/mod2_1",
172      {"ut_data/app1/mod2/mod2_1/mod2_1_1.cpp", "ut_data/app1/mod2/mod2_1/mod2_1_1.h"}},
173     {"ut_data/app1/mod2/mod2_2",
174      {"ut_data/app1/mod2/mod2_2/mod2_2_1.cpp", "ut_data/app1/mod2/mod2_2/mod2_2_1.h"}},
175     {"ut_data/app2", {"ut_data/app2/b_1.cpp", "ut_data/app2/b_1.h"}},
176   };
177
178   ASSERT_EQ(act, exp);
179 }
180 {
181   auto const exp_dirs = Paths_t{".", "ut_data/app1/mod1"};
182   auto const exp_srcs = Paths_t{"path_utils.cpp", "ut_data/app1/mod1/mod1_1.cpp",
183                                "ut_data/app1/mod1/mod1_1.hpp"};
184
185   auto const act = AssginSrcsToDirs(exp_dirs, exp_srcs);
186
187   auto const exp = Dirs2Srcs_t{
188     {".", {"path_utils.cpp"}},
189     {"ut_data/app1/mod1", {"ut_data/app1/mod1/mod1_1.cpp", "ut_data/app1/mod1/mod1_1.hpp"}},
190   };
191
192   ASSERT_EQ(act, exp);
193 }
194 }
195
196 TEST(path_utils, PackageSrcMatcher2)
197 {
198   auto const exp_dirs = Paths_t{"ut_data", "ut_data/app1/mod2", "ut_data/app1/mod2/mod2_1",
199                                "ut_data/app1/mod2/mod2_2", "ut_data/app2"};
200
201   auto const exp_srcs = Paths_t{"ut_data/app1/a_1_c.c",
202                               "ut_data/app1/a_1_c.h",
203                               "ut_data/app1/a_1_cpp.cpp",
204                               "ut_data/app1/a_1_cpp.h",
205                               "ut_data/app1/a_2_c.C",
206                               "ut_data/app1/a_2_c.H",
207                               "ut_data/app1/a_2_cpp.cxx",
208                               "ut_data/app1/a_2_cpp.hpp",
209                               "ut_data/app1/a_3_cpp.cc",
210                               "ut_data/app1/mod1/mod1_1.cpp",
211                               "ut_data/app1/mod1/mod1_1.hpp",
212                               "ut_data/app1/mod1/mod1_2.hpp",
213                               "ut_data/app1/mod2/mod2_1.cpp",
214                               "ut_data/app1/mod2/mod2_1.hpp",
215                               "ut_data/app1/mod2/mod2_1/mod2_1_1.cpp",
216                               "ut_data/app1/mod2/mod2_1/mod2_1_1.h",
217                               "ut_data/app1/mod2/mod2_2/mod2_2_1.cpp",
218                               "ut_data/app1/mod2/mod2_2/mod2_2_1.h",
219                               "ut_data/app2/b_1.cpp",
220                               "ut_data/app2/b_1.h"};
221
222   auto const act = AssginSrcsToDirs(exp_dirs, exp_srcs);
223
224   auto const exp = Dirs2Srcs_t{
225     {"ut_data",
226      {"ut_data/app1/a_1_c.c", "ut_data/app1/a_1_c.h", "ut_data/app1/a_1_cpp.cpp",
227       "ut_data/app1/a_1_cpp.h", "ut_data/app1/a_2_c.C", "ut_data/app1/a_2_c.H",
228       "ut_data/app1/a_2_cpp.cxx", "ut_data/app1/a_2_cpp.hpp", "ut_data/app1/a_3_cpp.cc",
229       "ut_data/app1/mod1/mod1_1.cpp", "ut_data/app1/mod1/mod1_1.hpp",
230       "ut_data/app1/mod1/mod1_2.hpp"}},
231     {"ut_data/app1", {}},
232     {"ut_data/app1/mod2", {"ut_data/app1/mod2/mod2_1.cpp", "ut_data/app1/mod2/mod2_1.hpp"}},
233     {"ut_data/app1/mod2/mod2_1",
234      {"ut_data/app1/mod2/mod2_1/mod2_1_1.cpp", "ut_data/app1/mod2/mod2_1/mod2_1_1.h"}},
235     {"ut_data/app1/mod2/mod2_2",
236      {"ut_data/app1/mod2/mod2_2/mod2_2_1.cpp", "ut_data/app1/mod2/mod2_2/mod2_2_1.h"}},
237     {"ut_data/app2", {"ut_data/app2/b_1.cpp", "ut_data/app2/b_1.h"}},
238   };
239
240   ASSERT_EQ(act, exp);
241 }
242

```

```

243 TEST(path_utils, PackageSrcMatcher3)
244 {
245     auto const exp_dirs
246         = Paths_t{"ut_data/app1/mod2/mod2_1", "ut_data/app1/mod2/mod2_2", "ut_data/app2"};
247
248     auto const exp_srcs = Paths_t{"ut_data/app1/a_1_c.c",
249                                 "ut_data/app1/mod1/mod1_1.cpp",
250                                 "ut_data/app1/mod1/mod1_1.hpp",
251                                 "ut_data/app1/mod1/mod1_2.hpp",
252                                 "ut_data/app1/mod2/mod2_1.cpp",
253                                 "ut_data/app1/mod2/mod2_1.hpp",
254                                 "ut_data/app1/mod2/mod2_1/mod2_1_1.cpp",
255                                 "ut_data/app1/mod2/mod2_1/mod2_1_1.h",
256                                 "ut_data/app1/mod2/mod2_2/mod2_2_1.cpp",
257                                 "ut_data/app1/mod2/mod2_2/mod2_2_1.hpp",
258                                 "ut_data/app2/b_1.cpp",
259                                 "ut_data/app2/b_1.h"};
260
261     auto const act = AssginSrcsToDirs(exp_dirs, exp_srcs);
262
263     auto const exp = Dirs2Srcs_t{
264         {"ut_data/app1/mod2/mod2_1",
265          {"ut_data/app1/mod2/mod2_1/mod2_1_1.cpp", "ut_data/app1/mod2/mod2_1/mod2_1_1.h"}},
266         {"ut_data/app1/mod2/mod2_2",
267          {"ut_data/app1/mod2/mod2_2/mod2_2_1.cpp", "ut_data/app1/mod2/mod2_2/mod2_2_1.h"}},
268         {"ut_data/app2", {"ut_data/app2/b_1.cpp", "ut_data/app2/b_1.h"}},
269         {"no_package",
270          {"ut_data/app1/a_1_c.c", "ut_data/app1/mod1/mod1_1.cpp", "ut_data/app1/mod1/mod1_1.hpp",
271           "ut_data/app1/mod1/mod1_2.hpp", "ut_data/app1/mod2/mod2_1.cpp",
272           "ut_data/app1/mod2/mod2_1.hpp"}},
273     };
274
275     ASSERT_EQ(act, exp);
276 }
277
278 TEST(path_utils, ToString_Path)
279 {
280     {
281         auto const exp_path = Path_t{"ut_data/app1/a_1_c.c"};
282         auto const exp      = std::string{"ut_data/app1/a_1_c.c"};
283         auto const act      = ToStringPath(exp_path);
284
285         ASSERT_EQ(act, exp);
286     }
287     {
288         auto const exp_path = Path_t{""};
289         auto const exp      = std::string{"\"\""};
290         auto const act      = ToStringPath(exp_path);
291
292         ASSERT_EQ(act, exp);
293     }
294 }
295
296 TEST(path_utils, ToString_Paths)
297 {
298     auto const exp_srcs = Paths_t{"ut_data/app1/a_1_c.c", "ut_data/app1/mod1/mod1_1.cpp",
299                                 "ut_data/app1/mod2/mod2_1/mod2_1_1.cpp",
300                                 "ut_data/app1/mod2/mod2_1/mod2_1_1.h", "ut_data/app2/b_1.h"};
301
302     auto const exp = std::string{
303         "ut_data/app1/a_1_c.c "
304         "ut_data/app1/mod1/mod1_1.cpp "
305         "ut_data/app1/mod2/mod2_1/mod2_1_1.cpp "
306         "ut_data/app1/mod2/mod2_1/mod2_1_1.h "
307         "ut_data/app2/b_1.h"};
308     auto const act = ToStringPaths(exp_srcs, " ");
309
310     ASSERT_EQ(act, exp);
311 }
312 } // namespace
313 } // namespace FileUtils

```

example/deps/lib/h/lib/nstd.h

```

1 #pragma once
2 #include <algorithm>
3 #include <fstream>
4 #include <list>
5 #include <string>
6 #include <utility>

```

```

7 #include <vector>
8
9 namespace Nstd {
10 template <typename T, size_t N>
11 constexpr size_t ArrayLength(T const (&)[N]) noexcept
12 {
13     return N;
14 }
15
16 template <typename T>
17 void SortUnique(std::vector<T>& v)
18 {
19     std::sort(v.begin(), v.end());
20     auto result = std::unique(v.begin(), v.end());
21     v.erase(result, v.end());
22 }
23
24 template <typename T>
25 void SortUnique(std::list<T>& v)
26 {
27     v.sort();
28     v.unique();
29 }
30
31 template <typename T>
32 void Concatenate(std::vector<T>& v0, std::vector<T>&& v1)
33 {
34     for (auto& v1_elem : v1) {
35         v0.insert(v0.end(), std::move(v1_elem));
36     }
37 }
38
39 template <typename T>
40 void Concatenate(std::list<T>& v0, std::list<T>&& v1)
41 {
42     v0.splice(v0.end(), std::move(v1));
43 }
44
45 template <typename F>
46 class ScopedGuard {
47 public:
48     explicit ScopedGuard(F&& f) noexcept : f_{f} {}
49     ~ScopedGuard() { f_(); }
50     ScopedGuard(ScopedGuard const&) = delete;
51     ScopedGuard& operator=(ScopedGuard const&) = delete;
52
53 private:
54     F f_;
55 };
56
57 inline std::string Replace(std::string in, std::string_view from, std::string_view to)
58 {
59     auto pos = in.find(from);
60
61     while (pos != std::string::npos) {
62         in.replace(pos, from.length(), to);
63         pos = in.find(from, pos + to.length());
64     }
65
66     return in;
67 }
68
69 //
70 // operator<< for range
71 //
72 namespace Inner_ {
73 //
74 // exists_put_to_as_member
75 //
76 template <typename, typename = std::ostream&>
77 struct exists_put_to_as_member : std::false_type {
78 };
79
80 template <typename T>
81 struct exists_put_to_as_member<T, decltype(std::declval<std::ostream&>().operator<<(std::declval<T>()))> : std::true_type {
82 };
83
84
85 template <typename T>
86 constexpr bool exists_put_to_as_member_v{exists_put_to_as_member<T>::value};

```

```

87
88 // 
89 // exists_put_to_as_non_member
90 //
91 template <typename, typename = std::ostream&>
92 struct exists_put_to_as_non_member : std::false_type {
93 };
94
95 template <typename T>
96 struct exists_put_to_as_non_member<T, decltype(operator<<(std::declval<std::ostream&>(),
97                                         std::declval<T>()))> : std::true_type {
98 };
99
100 template <typename T>
101 constexpr bool exists_put_to_as_non_member_v{exists_put_to_as_non_member<T>::value};
102
103 //
104 // exists_put_to_v
105 //
106 template <typename T>
107 constexpr bool exists_put_to_v{
108     Nstd::Inner_::exists_put_to_as_member_v<T> || Nstd::Inner_::exists_put_to_as_non_member_v<T>};
109
110 //
111 // is_range
112 //
113 template <typename, typename = bool>
114 struct is_range : std::false_type {
115 };
116
117 template <typename T>
118 struct is_range<T, typename std::enable_if_t<!std::is_array_v<T>,
119                                         decltype(std::begin(std::declval<T>())), bool{}>>
120     : std::true_type {
121 };
122
123 template <typename T>
124 struct is_range<T, typename std::enable_if_t<std::is_array_v<T>, bool>> : std::true_type {
125 };
126
127 //
128 // is_range_v
129 //
130 template <typename T>
131 constexpr bool is_range_v{is_range<T>::value};
132
133 } // namespace Inner_
134
135 //
136 // operator<< for range
137 //
138 template <typename T>
139 auto operator<<(std::ostream& os, T const& t) ->
140     typename std::enable_if_t<Inner_::is_range_v<T> && !Inner_::exists_put_to_v<T>, std::ostream&>
141 {
142     auto first = true;
143
144     for (auto const& i : t) {
145         if (!std::exchange(first, false)) {
146             os << ", ";
147         }
148         os << i;
149     }
150
151     return os;
152 }
153 } // namespace Nstd

```

exampledeps/lib/ut/nstd_ut.cpp

```

1 #include <filesystem>
2 #include <list>
3 #include <iostream>
4 #include <regex>
5 #include <string>
6
7 #include "gtest_wrapper.h"
8
9 #include "lib/nstd.h"
10

```

```

11 namespace Nstd {
12 namespace {
13
14 TEST(Nstd, ArrayLength)
15 {
16     {
17         char const* act[] = {"d", "a", "ab", "bcd"};
18
19         ASSERT_EQ(4, ArrayLength(act));
20     }
21     {
22         std::string const act[] = {"d", "a", "Ab"};
23
24         ASSERT_EQ(3, ArrayLength(act));
25     }
26 }
27
28 TEST(Nstd, SortUnique)
29 {
30     {
31         auto act = std::vector<std::string>{"d", "a", "ab", "bcd"};
32
33         SortUnique(act);
34
35         ASSERT_EQ((std::vector<std::string>{"a", "ab", "bcd", "d"}), act);
36     }
37     {
38         auto act = std::list<std::filesystem::path>{"d", "a", "Ab", "bcd"};
39
40         SortUnique(act);
41
42         ASSERT_EQ((std::list<std::filesystem::path>{"Ab", "a", "bcd", "d"}), act);
43     }
44 }
45
46 TEST(Nstd, Concatenate)
47 {
48     {
49         auto act0 = std::vector<std::string>{"d", "a", "ab"};
50         auto act1 = std::vector<std::string>{"bcd", "ef"};
51
52         Concatenate(act0, std::move(act1));
53         ASSERT_EQ((std::vector<std::string>{"d", "a", "ab", "bcd", "ef"}), act0);
54     }
55     {
56         auto act0 = std::list<std::filesystem::path>{"d", "a", "ab"};
57         auto act1 = std::list<std::filesystem::path>{"bcd", "ef"};
58
59         Concatenate(act0, std::move(act1));
60         ASSERT_EQ((std::list<std::filesystem::path>{"d", "a", "ab", "bcd", "ef"}), act0);
61     }
62 }
63
64 TEST(Nstd, ScopedGuard)
65 {
66     auto s = std::string("not called");
67
68     {
69         auto sg = ScopedGuard{&s}() noexcept { s = "called"; };
70         ASSERT_EQ(s, "not called");
71     }
72
73     ASSERT_EQ(s, "called");
74 }
75
76 TEST(Nstd, Replace)
77 {
78     {
79         auto in = std::string("a-b-c-d");
80         auto act = Replace(in, "-", "/");
81         ASSERT_EQ(act, "a/b/c/d");
82     }
83     {
84         auto in = std::string("a-b-c-d");
85         auto act = Replace(in, "-", "/////////");
86         ASSERT_EQ(act, "a////////b////////c////////d");
87     }
88     {
89         auto in = std::string("a-b-c-d");
90         auto act = Replace(in, "-", "");

```

```

91     ASSERT_EQ(act, "abcd");
92 }
93 }
94
95 TEST(stl_try, exclude)
96 {
97     auto dirs = std::list<std::string>{"A", "B", "A/e", "A/e/f", "B/xxx/ef"};
98     auto exclude = std::string{R"(.*/e\b.*)"};
99     auto const pattern = std::regex(exclude);
100
101    dirs.remove_if([&pattern](auto const& d) {
102        auto results = std::smatch{};
103        return std::regex_match(d, results, pattern);
104    });
105
106    ASSERT_EQ(dirs, (std::list<std::string>{"A", "B", "B/xxx/ef"}));
107 }
108 } // namespace
109
110 namespace Inner_ {
111 namespace {
112 class test_class_exits_put_to {};
113
114 std::ostream& operator<<(std::ostream& os, test_class_exits_put_to) { return os; }
115
116 class test_class_not_exits_put_to {};
117
118 TEST(Nstd, exists_put_to_as_member)
119 {
120     static_assert(exists_put_to_as_member_v<bool>);
121     static_assert(exists_put_to_as_member_v<char[3]>);
122     static_assert(exists_put_to_as_member_v<std::string>);
123     static_assert(!exists_put_to_as_member_v<std::vector<int>>);
124     static_assert(exists_put_to_as_member_v<std::vector<int*>>);
125     static_assert(!exists_put_to_as_member_v<test_class_exits_put_to>);
126     static_assert(exists_put_to_as_member_v<test_class_not_exits_put_to>);
127     static_assert(exists_put_to_as_member_v<test_class_not_exits_put_to[3]>);
128     auto oss = std::ostringstream{};
129     oss << test_class_exits_put_to{};
130 }
131
132 TEST(Template, exists_put_to_as_non_member)
133 {
134     static_assert(!exists_put_to_as_non_member_v<bool>);
135     static_assert(exists_put_to_as_non_member_v<std::string>);
136     static_assert(exists_put_to_as_non_member_v<std::vector<int>>);
137     static_assert(exists_put_to_as_non_member_v<std::vector<int*>>);
138     static_assert(exists_put_to_as_non_member_v<test_class_exits_put_to>);
139     static_assert(!exists_put_to_as_non_member_v<test_class_not_exits_put_to>);
140     static_assert(!exists_put_to_as_non_member_v<test_class_not_exits_put_to[3]>);
141 }
142
143 TEST(Template, exists_put_to_v)
144 {
145     static_assert(exists_put_to_v<bool>);
146     static_assert(exists_put_to_v<std::string>);
147     static_assert(exists_put_to_v<std::vector<int>>);
148     static_assert(exists_put_to_v<std::vector<int*>>);
149     static_assert(exists_put_to_v<test_class_exits_put_to>);
150     static_assert(!exists_put_to_v<test_class_not_exits_put_to>);
151     static_assert(exists_put_to_v<test_class_not_exits_put_to[3]>);
152 }
153
154 TEST(Template, is_range)
155 {
156     static_assert(is_range_v<std::string>);
157     static_assert(!is_range_v<int>);
158     static_assert(is_range_v<int const[3]>);
159     static_assert(is_range_v<int[3]>);
160 }
161 } // namespace
162 } // namespace Inner_
163
164 namespace {
165 TEST(Template, PutTo)
166 {
167     {
168         auto oss = std::ostringstream{};
169         char c[] = "c3";
170

```

```

171     oss << c;
172     ASSERT_EQ("c3", oss.str());
173 }
174 {
175     auto oss = std::ostringstream{};
176     auto str = std::vector<std::string>{"1", "2", "3"};
177
178     oss << str;
179     ASSERT_EQ("1, 2, 3", oss.str());
180 }
181 {
182     auto oss = std::ostringstream{};
183     auto p   = std::list<std::filesystem::path>{"1", "2", "3"};
184
185     oss << p;
186     ASSERT_EQ("\\"1\\", \\"2\\", \\"3\\\"", oss.str());
187 }
188 }
189 } // namespace
190 } // namespace Nstd

```

exampledeps/logging/h/logging/logger.h

```

1 #pragma once
2
3 #include <fstream>
4 #include <iostream>
5 #include <sstream>
6 #include <string>
7 #include <string_view>
8
9 #include "lib/nstd.h"
10
11 // @@@ sample begin 0:0
12
13 namespace Logging {
14 class Logger {
15 public:
16     static Logger& Inst(char const* filename = nullptr);
17
18     template <typename HEAD, typename... TAIL>
19     void Set(char const* filename, uint32_t line_no, HEAD const& head, TAIL... tails)
20     {
21         auto path    = std::string_view{filename};
22         size_t npos   = path.find_last_of('/');
23         auto basename = (npos != std::string_view::npos) ? path.substr(npos + 1) : path;
24
25         os_.width(12);
26         os_ << basename << ":";
27
28         os_.width(3);
29         os_ << line_no;
30
31         set_inner(head, tails...);
32     }
33
34 // @@@ ignore begin
35 void Close();
36 Logger(Logger const&)           = delete;
37 Logger& operator=(Logger const&) = delete;
38 // @@@ ignore end
39
40 private:
41     void set_inner() { os_ << std::endl; }
42
43     template <typename HEAD, typename... TAIL>
44     void set_inner(HEAD const& head, TAIL... tails)
45     {
46         using Nstd::operator<<;
47         os_ << ":" << head;
48         set_inner(tails...);
49     }
50
51     template <typename HEAD, typename... TAIL>
52     void set_inner(char sep, HEAD const& head, TAIL... tails)
53     {
54         using Nstd::operator<<;
55         os_ << sep << head;
56         set_inner(tails...);
57     }

```

```

58 // @@@ ignore begin
59     explicit Logger(char const* filename);
60
61     std::ofstream ofs_{};
62     std::ostream& os_{};
63
64     // @@@ ignore end
65 };
66 } // namespace Logging
67
68 #define LOGGER_INIT(filename) Logging::Logger::Inst(filename)
69 #define LOGGER(...) Logging::Logger::Inst().Set(__FILE__, __LINE__, __VA_ARGS__)
70
71 // @@@ sample end

```

exampledeps/logging/src/logger.cpp

```

1 #include "logging/logger.h"
2
3 namespace {
4 class null_ostream : private std::streambuf, public std::ostream {
5 public:
6     static null_ostream& Inst()
7     {
8         static null_ostream inst;
9         return inst;
10    }
11
12 protected:
13     virtual int overflow(int c)
14     {
15         setup(buf_, buf_ + sizeof(buf_));
16         return (c == eof() ? '\0' : c);
17     }
18
19 private:
20     null_ostream() : std::ostream{this} {}
21     char buf_[128];
22 };
23
24 std::ostream& init_os(char const* filename, std::ofstream& ofs)
25 {
26     if (filename == nullptr) {
27         return std::cout;
28     }
29     else {
30         if (std::string{filename}.size() == 0) {
31             return null_ostream::Inst();
32         }
33         else {
34             ofs.open(filename);
35             return ofs;
36         }
37     }
38 }
39 } // namespace
40
41 namespace Logging {
42 Logger::Logger(char const* filename) : os_{init_os(filename, ofs_)} {}
43
44 Logger& Logger::Inst(char const* filename)
45 {
46     static auto inst = Logger{filename};
47
48     return inst;
49 }
50
51 void Logger::Close()
52 {
53     if (&std::cout != &os_) {
54         ofs_.close();
55     }
56 }
57 } // namespace Logging

```

exampledeps/logging/ut/logger_ut.cpp

```

1 #include <filesystem>
2

```

```

3 #include "gtest_wrapper.h"
4
5 #include "logging/logger.h"
6
7 namespace {
8
9 TEST(log, Logger)
10 {
11     // loggingのテストは他のライブラリで行う。
12     // ここではコンパイルできるとの確認のみ。
13
14     LOGGER_INIT(nullptr);
15     LOGGER(1);
16     LOGGER("xyz", 3, 5);
17
18     auto file = std::filesystem::path{"hehe"};
19     LOGGER(file);
20 }
21 } // namespace

```

example/dynamic_memory_allocation/malloc_ut.cpp

```

1 #include <sys/unistd.h>
2
3 #include <cassert>
4 #include <cstdint>
5 #include <mutex>
6
7 #include "gtest_wrapper.h"
8
9 #include "dynamic_memory_allocation_ut.h"
10 #include "spin_lock.h"
11 #include "utils.h"
12
13 // @@@ sample begin 0:0
14
15 extern "C" void* sbrk(ptrdiff_t __incr);
16 // @@@ sample end
17
18 namespace MallocFree {
19 // @@@ sample begin 1:0
20
21 namespace {
22
23 struct header_t {
24     header_t* next;
25     size_t n_nuits; // header_tが何個あるか
26 };
27
28 header_t* header{nullptr};
29 SpinLock spin_lock{};
30 constexpr size_t unit_size{sizeof(header_t)};
31
32 inline bool sprit(header_t* header, size_t n_nuits, header_t*& next) noexcept
33 {
34     // @@@ ignore begin
35     assert(n_nuits > 1); // ヘッダとバッファなので最低でも2
36
37     next = nullptr;
38
39     if (header->n_nuits == n_nuits || header->n_nuits == n_nuits + 1) {
40         next = header->next;
41         return true;
42     }
43     else if (header->n_nuits > n_nuits) {
44         next = header + n_nuits;
45         next->n_nuits = header->n_nuits - n_nuits;
46         next->next = header->next;
47         header->n_nuits = n_nuits;
48         return true;
49     }
50
51     return false;
52     // @@@ ignore end
53 }
54
55 inline void concat(header_t* front, header_t* rear) noexcept
56 {
57     // @@@ ignore begin
58     if (front + front->n_nuits == rear) { // 1枚のメモリになる

```

```

59         front->n_nuits += rear->n_nuits;
60         front->next = rear->next;
61     }
62     else {
63         front->next = rear;
64     }
65     // @@@ ignore end
66 }
67
68 header_t* set_back(void* mem) noexcept { return static_cast<header_t*>(mem) - 1; }
69
70 static_assert(sizeof(header_t) == alignof(std::max_align_t));
71
72 void* malloc_inner(size_t size) noexcept
73 {
74     // @@@ ignore begin
75     // size分のメモリとヘッダ
76     auto n_nuits = (Roundup(unit_size, size) / unit_size) + 1;
77     auto lock    = std::lock_guard{spin_lock};
78
79     auto curr = header;
80     for (header_t* prev = nullptr; curr != nullptr; prev = curr, curr = curr->next) {
81         header_t* next;
82
83         if (!split(curr, n_nuits, next)) {
84             continue;
85         }
86
87         if (prev == nullptr) {
88             header = next;
89         }
90         else {
91             prev->next = next;
92         }
93         break;
94     }
95
96     if (curr != nullptr) {
97         ++curr;
98     }
99
100    return curr;
101    // @@@ ignore end
102 }
103 } // namespace
104 // @@@ sample end
105 // @@@ sample begin 2:0
106
107 void free(void* mem) noexcept
108 {
109     header_t* mem_to_free = set_back(mem);
110
111     mem_to_free->next = nullptr;
112
113     auto lock = std::lock_guard{spin_lock};
114
115     if (header == nullptr) {
116         header = mem_to_free;
117         return;
118     }
119     // @@@ sample end
120     // @@@ sample begin 2:1
121
122     if (mem_to_free < header) {
123         concat(mem_to_free, header);
124         header = mem_to_free;
125         return;
126     }
127
128     auto curr = header;
129     for (; curr->next != nullptr; curr = curr->next) {
130         if (mem_to_free < curr->next) { // 常に curr < mem_to_free
131             concat(mem_to_free, curr->next);
132             concat(curr, mem_to_free);
133             return;
134         }
135     }
136
137     concat(curr, mem_to_free);
138     // @@@ sample end

```

```

139     // @@@ sample begin 2:2
140 }
141 // @@@ sample end
142 // @@@ sample begin 3:0
143
144 void* malloc(size_t size) noexcept
145 {
146     void* mem = malloc_inner(size);
147     // @@@ sample end
148     // @@@ sample begin 3:1
149
150     if (mem == nullptr) {
151         auto const add_size = Roundup(unit_size, 1024 * 1024 + size); // 1MB追加
152
153         header_t* add = static_cast<header_t*>(sbrk(add_size));
154         add->n_nuits = add_size / unit_size;
155         free(++add);
156         mem = malloc_inner(size);
157     }
158     // @@@ sample end
159     // @@@ sample begin 3:2
160
161     return mem;
162 }
163 // @@@ sample end
164
165 namespace {
166 TEST(NewDelete_Opt, malloc)
167 {
168     {
169         void* mem = malloc(1024);
170
171         ASSERT_NE(nullptr, mem);
172         free(mem);
173
174         void* ints[8]{};
175
176         constexpr auto n_nuits = Roundup(unit_size, unit_size + sizeof(int)) / unit_size;
177
178         for (auto& i : ints) {
179             i = malloc(sizeof(int));
180
181             header_t* h = set_back(i);
182             ASSERT_EQ(h->n_nuits, n_nuits);
183         }
184
185         for (auto& i : ints) {
186             free(i);
187         }
188     }
189
190     // @@@ sample begin 4:0
191
192     void* mem[1024];
193
194     for (auto& m : mem) { // 32バイト x 1024個のメモリ確保
195         m = malloc(32);
196     }
197
198     // memを使用した何らかの処理
199     // @@@ ignore begin
200     // @@@ ignore end
201
202     for (auto i = 0U; i < ArrayLength(mem); i += 2) { // 512個のメモリを解放
203         free(mem[i]);
204     }
205     // @@@ sample end
206
207     for (auto i = 1U; i < ArrayLength(mem); i += 2) {
208         free(mem[i]);
209     }
210 }
211 } // namespace
212 } // namespace MallocFree

```

example/dynamic_memory_allocation/mpool_variable.h

```

1 #pragma once
2 #include <cassert>
3 #include <cstdint>

```

```

4 #include <mutex>
5 #include <optional>
6
7 #include "mpool.h"
8 #include "spin_lock.h"
9 #include "utils.h"
10
11 namespace Inner_ {
12
13 struct header_t {
14     header_t* next;
15     size_t    n_nuits; // header_tが何個あるか
16 };
17
18 constexpr auto unit_size = sizeof(header_t);
19
20 inline std::optional<header_t*> sprit(header_t* header, size_t n_nuits) noexcept
21 {
22     assert(n_nuits > 1); // ヘッダとバッファなので最低でも2
23
24     if (header->n_nuits == n_nuits || header->n_nuits == n_nuits + 1) {
25         return header->next;
26     }
27     else if (header->n_nuits > n_nuits) {
28         auto next      = header + n_nuits;
29         next->n_nuits = header->n_nuits - n_nuits;
30         next->next    = header->next;
31         header->n_nuits = n_nuits;
32         return next;
33     }
34
35     return std::nullopt;
36 }
37
38 inline void concat(header_t* front, header_t* rear) noexcept
39 {
40     if (front + front->n_nuits == rear) { // 1枚のメモリになる
41         front->n_nuits += rear->n_nuits;
42         front->next = rear->next;
43     }
44     else {
45         front->next = rear;
46     }
47 }
48
49 inline header_t* set_back(void* mem) noexcept { return static_cast<header_t*>(mem) - 1; }
50
51 static_assert(sizeof(header_t) == alignof(std::max_align_t));
52
53 template <uint32_t MEM_SIZE>
54 struct buffer_t {
55     alignas(alignof(std::max_align_t)) uint8_t buffer[Roundup(sizeof(header_t), MEM_SIZE)];
56 };
57 } // namespace Inner_
58
59 // @@@ sample begin 0:0
60
61 template <uint32_t MEM_SIZE>
62 class MPoolVariable final : public MPool {
63 public:
64     // @@@ sample end
65     // @@@ sample begin 0:1
66     MPoolVariable() noexcept : MPool{MEM_SIZE}
67     {
68         header_->next      = nullptr;
69         header_->n_nuits = sizeof(buff_) / Inner_::unit_size;
70     }
71     // @@@ sample end
72     // @@@ sample begin 0:2
73
74     class const_iterator {
75     public:
76         explicit const_iterator(Inner_::header_t const* header) noexcept : header_{header} {}
77         const_iterator(const_iterator const&) = default;
78         const_iterator(const_iterator&&)      = default;
79
80         const_iterator& operator++() noexcept // 前置++のみ実装
81         {
82             assert(header_ != nullptr);
83             header_ = header_->next;

```

```

84         return *this;
85     }
86
87     Inner_::header_t const* operator*() noexcept { return header_; }
88     bool operator==(const_iterator const& rhs) noexcept { return header_ == rhs.header_; }
89     bool operator!=(const_iterator const& rhs) noexcept { return !(*this == rhs); }
90
91 private:
92     Inner_::header_t const* header_;
93 };
94
95 const_iterator begin() const noexcept { return const_iterator{header_}; }
96 const_iterator end() const noexcept { return const_iterator{nullptr}; }
97 const_iterator cbegin() const noexcept { return const_iterator{header_}; }
98 const_iterator cend() const noexcept { return const_iterator{nullptr}; }
99
100 // @@@ sample end
101 // @@@ sample begin 0:3
102
103 private:
104     using header_t = Inner_::header_t;
105
106     Inner_::buffer_t<MEM_SIZE> buff_{};
107     header_t* header_{reinterpret_cast<header_t*>(buff_.buffer)};
108     mutable SpinLock spin_lock_{};
109     size_t unit_count_{sizeof(buff_) / Inner_::unit_size};
110     size_t unit_count_min_{sizeof(buff_) / Inner_::unit_size};
111
112     virtual void* alloc(size_t size) noexcept override
113 {
114     // @@@ ignore begin
115     // size分のメモリとヘッダ
116     auto n_nuits = (Roundup(Inner_::unit_size, size) / Inner_::unit_size) + 1;
117
118     auto lock = std::lock_guard{spin_lock_};
119
120     auto curr = header_;
121
122     for (header_t* prev=nullptr; curr != nullptr; prev = curr, curr = curr->next) {
123         auto opt_next = std::optional<header_t*>{sprit(curr, n_nuits)};
124
125         if (!opt_next) {
126             continue;
127         }
128
129         auto next = *opt_next;
130         if (prev == nullptr) {
131             header_ = next;
132         }
133         else {
134             prev->next = next;
135         }
136         break;
137     }
138
139     if (curr != nullptr) {
140         unit_count_ -= curr->n_nuits;
141         unit_count_min_ = std::min(unit_count_, unit_count_min_);
142         ++curr;
143     }
144
145     return curr;
146     // @@@ ignore end
147 }
148
149     virtual void free(void* mem) noexcept override
150 {
151     // @@@ ignore begin
152     header_t* to_free = Inner_::set_back(mem);
153
154     to_free->next = nullptr;
155
156     auto lock = std::lock_guard{spin_lock_};
157
158     unit_count_ += to_free->n_nuits;
159     unit_count_min_ = std::min(unit_count_, unit_count_min_);
160
161     if (header_ == nullptr) {
162         header_ = to_free;
163         return;

```

```

164     }
165
166     if (to_free < header_) {
167         concat(to_free, header_);
168         header_ = to_free;
169         return;
170     }
171
172     header_t* curr = header_;
173
174     for (; curr->next != nullptr; curr = curr->next) {
175         if (to_free < curr->next) { // 常に curr < to_free
176             concat(to_free, curr->next);
177             concat(curr, to_free);
178             return;
179         }
180     }
181
182     concat(curr, to_free);
183     // @@@ ignore end
184 }
185
186 virtual size_t get_size() const noexcept override { return 1; }
187 virtual size_t get_count() const noexcept override { return unit_count_ * Inner_::unit_size; }
188 virtual size_t get_count_min() const noexcept override
189 {
190     return unit_count_min_ * Inner_::unit_size;
191 }
192
193 virtual bool is_valid(void const* mem) const noexcept override
194 {
195     return (&buff_ < mem) && (mem < &buff_.buffer[ArrayLength(buff_.buffer)]);
196 }
197 // @@@ sample end
198 // @@@ sample begin 0:4
199 };
200 // @@@ sample end

```

example/programming_convention/fixed_point.h

```

1 #pragma once
2
3 #include <cstdint>
4 #include <type_traits>
5
6 // @@@ sample begin 0:0
7
8 /// @class FixedPoint
9 /// @brief BASIC_TYPEで指定する基本型のビット長を持つ固定小数点を扱うためのクラス
10 /// @tparam BASIC_TYPE 全体のビット長や、符号を指定するための整数型
11 /// @tparam FRACTION_BIT_NUM 小数点保持のためのビット長
12 template <typename BASIC_TYPE, uint32_t FRACTION_BIT_NUM>
13 class FixedPoint {
14 public:
15     FixedPoint(BASIC_TYPE integer = 0,
16                 typename std::make_unsigned_t<BASIC_TYPE> fraction = 0) noexcept
17     : value_{get_init_value(integer, fraction)}
18     {
19         // @@@ ignore begin
20         // signedに対する右ビットシフトの仕様が、算術右ビットシフトでないと
21         // このクラスは成立しない。下記のstatic_assertはその確認。
22         static_assert(IsSigned() ? (-1 >> 1 == -1) : true, "need logical right bit shift");
23
24         // BASIC_TYPEをcharにすることは認めない。
25         static_assert(!std::is_same_v<BASIC_TYPE, char>, "BASIC_TYPE should not be char");
26         // @@@ ignore end
27     }
28 // @@@ sample end
29 // @@@ sample begin 0:1
30
31 // @@@ ignore begin
32     ~FixedPoint() = default;
33     FixedPoint(FixedPoint const&) = default;
34     FixedPoint& operator=(FixedPoint const&) = default;
35     FixedPoint(FixedPoint&&) noexcept = default;
36     FixedPoint& operator=(FixedPoint&&) noexcept = default;
37
38     BASIC_TYPE GetValue() const noexcept { return value_; }
39
40     BASIC_TYPE GetInteger() const noexcept { return value_ >> fraction_bit_num_; }

```

```

41     BASIC_TYPE GetFraction() const noexcept { return value_ & fraction_bit_mask_; }
42
43     double ToFloatPoint() const noexcept
44     {
45         return GetInteger() + (static_cast<double>(GetFraction()) / (fraction_bit_mask_ + 1));
46     }
47
48     constexpr typename std::make_unsigned_t<BASIC_TYPE> GetFractionMask() const noexcept
49     {
50         return fraction_bit_mask_;
51     }
52
53     constexpr typename std::make_unsigned_t<BASIC_TYPE> GetIntegerMask() const noexcept
54     {
55         return integer_bit_mask_;
56     }
57
58     static constexpr bool IsSigned() noexcept { return std::is_signed_v<BASIC_TYPE>; }
59
60     static constexpr bool IsUnsigned() noexcept { return std::is_unsigned_v<BASIC_TYPE>; }
61
62     // @@@ ignore end
63     FixedPoint& operator+=(FixedPoint rhs) noexcept
64     // @@@ ignore begin
65     {
66         value_ += rhs.value_;
67         return *this;
68     }
69
70     // @@@ ignore end
71     FixedPoint& operator-=(FixedPoint rhs) noexcept
72     // @@@ ignore begin
73     {
74         value_ -= rhs.value_;
75         return *this;
76     }
77
78     // @@@ ignore end
79     FixedPoint& operator*=(FixedPoint rhs) noexcept
80     // @@@ ignore begin
81     {
82         value_ *= rhs.value_ >> fraction_bit_num_;
83         return *this;
84     }
85
86     // @@@ ignore end
87     FixedPoint& operator/=(FixedPoint rhs) noexcept
88     // @@@ ignore begin
89     {
90         using T = std::conditional_t<IsSigned(), int64_t, uint64_t>;
91
92         value_ = (static_cast<T>(value_) << fraction_bit_num_) / rhs.value_;
93
94         return *this;
95     }
96
97     // @@@ ignore end
98 private:
99     BASIC_TYPE value_;
100    // @@@ ignore begin
101
102    static constexpr uint32_t bit_mask(uint32_t bit_len) noexcept
103    {
104        if (bit_len == 0) {
105            return 0x0;
106        }
107
108        return bit_mask(bit_len - 1) | (0x01 << (bit_len - 1));
109    }
110
111    static constexpr uint32_t fraction_bit_num_{FRACTION_BIT_NUM};
112    static constexpr uint32_t fraction_bit_mask_{bit_mask(fraction_bit_num_)};
113    static constexpr uint32_t integer_bit_num_{sizeof(BASIC_TYPE) * 8 - FRACTION_BIT_NUM};
114    static constexpr uint32_t integer_bit_mask_{bit_mask(integer_bit_num_) << fraction_bit_num_};
115
116    static constexpr BASIC_TYPE get_init_value(BASIC_TYPE integer, BASIC_TYPE fraction) noexcept
117    {
118        #if 0 // 本来は左シフト<<を使いたいが、signedに対しての<<は良くないので
119        return (integer << fraction_bit_num_) | fraction;
120    }

```

```

121 #else
122     return (integer * (fraction_bit_mask_ + 1)) | fraction;
123 #endif
124 }
125 // @@@ ignore end
126
127 friend bool operator==(FixedPoint lhs, FixedPoint rhs) noexcept
128 // @@@ ignore begin
129 {
130     return lhs.value_ == rhs.value_;
131 }
132
133 friend bool operator!=(FixedPoint lhs, FixedPoint rhs) noexcept { return !(lhs == rhs); }
134
135 friend bool operator>(FixedPoint lhs, FixedPoint rhs) noexcept
136 {
137     return lhs.value_ > rhs.value_;
138 }
139
140 friend bool operator>=(FixedPoint lhs, FixedPoint rhs) noexcept
141 {
142     return (lhs > rhs) || (lhs == rhs);
143 }
144
145 friend bool operator<(FixedPoint lhs, FixedPoint rhs) noexcept { return (rhs > lhs); }
146
147 friend bool operator<=(FixedPoint lhs, FixedPoint rhs) noexcept
148 {
149     return (lhs < rhs) || (lhs == rhs);
150 }
151 // @@@ ignore end
152
153 // FixedPoint() + intのようなオーバーロードを作るためにあえてfriend
154 friend FixedPoint operator+(FixedPoint lhs, FixedPoint rhs) noexcept
155 // @@@ ignore begin
156 {
157     lhs += rhs;
158     return lhs;
159 }
160
161 friend FixedPoint operator-(FixedPoint lhs, FixedPoint rhs) noexcept
162 {
163     lhs -= rhs;
164     return lhs;
165 }
166
167 friend FixedPoint operator*(FixedPoint lhs, FixedPoint rhs) noexcept
168 {
169     lhs *= rhs;
170     return lhs;
171 }
172
173 friend FixedPoint operator/(FixedPoint lhs, FixedPoint rhs) noexcept
174 {
175     lhs /= rhs;
176     return lhs;
177 }
178 // @@@ ignore end
179 };
180 // @@@ sample end

```

etc

exampledeps/CMakeLists.txt

```

1 cmake_minimum_required(VERSION 3.10)
2
3 project(main_project)
4
5 set(CMAKE_CXX_STANDARD 17)
6 set(CMAKE_CXX_STANDARD_REQUIRED True)
7
8 # CMakeオプションを定義
9 option(USE_SANITIZERS "Enable sanitizers" OFF)
10
11 # USE_SANITIZERS オプションをチェック
12 if(USE_SANITIZERS)
13     message(STATUS "Sanitizers are enabled")

```

```

14     set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fsanitize=address,leak,undefined,floating-point-exception,overflow")
15     set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -fsanitize=address,leak,undefined,floating-point-exception,overflow")
16   else()
17     message(STATUS "Sanitizers are disabled")
18   endif()
19
20
21 set(GTEST_DIR "../../googletest")
22 set(CMAKE_EXPORT_COMPILE_COMMANDS ON)
23
24 include_directories("${GTEST_DIR}/googletest/include")
25
26 # googletestサブディレクトリを追加
27 add_subdirectory(${GTEST_DIR} ${CMAKE_BINARY_DIR}/googletest EXCLUDE_FROM_ALL)
28
29 add_subdirectory(lib)
30 add_subdirectory(logging)
31 add_subdirectory(file_utils)
32 add_subdirectory(dependency)
33 add_subdirectory(app)
34
35 # すべてのテストを実行するカスタムターゲットを追加
36 add_custom_target(tests
37   DEPENDS app_ut dependency_ut file_utils_ut lib_ut logging_ut deps_it
38   WORKING_DIRECTORY ${CMAKE_BINARY_DIR}
39 )
40

```

exampledeps/dependency/CMakeLists.txt

```

1 cmake_minimum_required(VERSION 3.10)
2
3 project(dependency VERSION 1.0)
4
5 # C++の標準を設定
6 set(CMAKE_CXX_STANDARD 17)
7 set(CMAKE_EXPORT_COMPILE_COMMANDS ON)
8
9 # ライブラリのソースファイルを追加
10 add_library(dependency STATIC
11   src/arch_pkg.cpp
12   src/cpp_deps.cpp
13   src/cpp_dir.cpp
14   src/cpp_src.cpp
15   src/deps_scenario.cpp
16   src/load_store_format.cpp
17 )
18
19 # @@@ sample begin 0:0
20
21 # dependency.aをリンクするファイルに
22 # ..../dependency/h ..../file_utils/h ..../lib/h
23 # のヘッダファイルを公開する
24
25 target_include_directories(dependency PUBLIC ..../dependency/h ..../file_utils/h ..../lib/h)
26 # @@@ sample end
27
28 # テスト用のソースファイルを追加して単一の実行ファイルを生成
29 add_executable(dependency_ut_exe
30   ut/arch_pkg_ut.cpp
31   ut/cpp_deps_ut.cpp
32   ut/cpp_dir_ut.cpp
33   ut/cpp_src_ut.cpp
34   ut/deps_scenario_ut.cpp
35   ut/load_store_format_ut.cpp
36 )
37
38 # @@@ sample begin 1:0
39
40 # dependency_ut_exeはdependency.aの単体テスト
41 # dependency_ut_exeが使用するライブラリのヘッダは下記の記述で公開される
42 target_link_libraries(dependency_ut_exe dependency file_utils logging gtest gtest_main)
43
44 # dependency_ut_exeに上記では公開範囲が不十分である場合、
45 # dependency_ut_exeが使用するライブラリのヘッダは下記の記述で限定的に公開される
46 # dependency_ut_exeにはdependency/src/*.hへのアクセスが必要
47 target_include_directories(dependency_ut_exe PRIVATE ../../../../deep/h src)

```

```

48 # @@@ sample end
49
50 # テストを追加
51 enable_testing()
52 add_test(NAME dependency_ut COMMAND dependency_ut_exe)
53
54 add_custom_target(dependency_ut_copy_test_data
55   COMMAND ${CMAKE_COMMAND} -E copy_directory
56   ${CMAKE_SOURCE_DIR}/ut_data ${TARGET_FILE_DIR}:dependency_ut_exe>/ut_data
57 )
58
59 # カスタムターゲットを追加して、ビルド後にテストを実行
60 add_custom_target(dependency_ut
61   COMMAND ${CMAKE_CTEST_COMMAND} --output-on-failure
62   DEPENDS dependency_ut_exe dependency_ut_copy_test_data
63 )
64

```

exampledeps/file_utils/CMakeLists.txt

```

1 cmake_minimum_required(VERSION 3.10)
2
3 project(file_utils VERSION 1.0)
4
5 # C++の標準を設定
6 set(CMAKE_CXX_STANDARD 17)
7 set(CMAKE_EXPORT_COMPILE_COMMANDS ON)
8
9 add_library(file_utils STATIC
10   src/load_store_row.cpp
11   src/path_utils.cpp
12 )
13
14 target_include_directories(file_utils PUBLIC ..//file_utils/h)
15
16 add_executable(file_utils_ut_exe ut/load_store_row_ut.cpp ut/path_utils_ut.cpp)
17
18 target_link_libraries(file_utils_ut_exe file_utils logging gtest gtest_main)
19
20 target_include_directories(file_utils_ut_exe PRIVATE h ..//..//deep/h ..//logging/h ..//lib/h)
21
22 add_custom_command(TARGET file_utils_ut_exe POST_BUILD
23   COMMAND ${CMAKE_COMMAND} -E copy_directory
24   ${CMAKE_SOURCE_DIR}/ut_data ${TARGET_FILE_DIR}:file_utils_ut_exe>/ut_data
25 )
26
27 enable_testing()
28 add_test(NAME file_utils_ut COMMAND file_utils_ut_exe)
29
30 add_custom_target(file_utils_ut_copy_test_data
31   COMMAND ${CMAKE_COMMAND} -E copy_directory
32   ${CMAKE_SOURCE_DIR}/ut_data ${TARGET_FILE_DIR}:file_utils_ut_exe>/ut_data
33 )
34
35 # カスタムターゲットを追加して、ビルド後にテストを実行
36 add_custom_target(file_utils_ut
37   COMMAND ${CMAKE_CTEST_COMMAND} --output-on-failure
38   DEPENDS file_utils_ut_exe file_utils_ut_copy_test_data
39 )
40

```

exampledeps/lib/CMakeLists.txt

```

1 cmake_minimum_required(VERSION 3.10)
2
3 project(lib VERSION 1.0)
4
5 # C++の標準を設定
6 set(CMAKE_CXX_STANDARD 17)
7 set(CMAKE_CXX_STANDARD_REQUIRED True)
8 set(CMAKE_EXPORT_COMPILE_COMMANDS ON)
9
10 add_executable(lib_ut_exe ut/nstd_ut.cpp)
11
12 if(NOT TARGET gtest)
13   message(FATAL_ERROR "gtest target not found. Make sure googletest is added at the top level CMakeLists.txt")
14 endif()
15

```

```
16 target_include_directories(lib_ut_exe PRIVATE h ../../h/ ../../deep/h)
17 target_link_libraries(lib_ut_exe gtest gtest_main)
18
19 enable_testing()
20 add_test(NAME lib_ut COMMAND lib_ut_exe)
21
22 add_custom_target(lib_ut
23     COMMAND ${CMAKE_CTEST_COMMAND} --output-on-failure
24     DEPENDS lib_ut_exe
25 )
26
```

exampledeps/logging/CMakeLists.txt

```
1 #logging/CMakeLists.txt
2
3 cmake_minimum_required(VERSION 3.10)
4
5 project(logging VERSION 1.0)
6
7 # C++の標準を設定
8 set(CMAKE_CXX_STANDARD 17)
9 set(CMAKE_CXX_STANDARD_REQUIRED True)
10 set(CMAKE_EXPORT_COMPILE_COMMANDS ON)
11
12 add_library(logging STATIC src/logger.cpp)
13
14 target_include_directories(logging PUBLIC h ./lib/h)
15
16 add_executable(logging_ut_exe ut/logger_ut.cpp)
17
18 target_include_directories(logging_ut_exe PRIVATE ../../../../deep/h ./lib/h)
19 target_link_libraries(logging_ut_exe logging gtest gtest_main)
20
21 enable_testing()
22 add_test(NAME logging_ut COMMAND logging_ut_exe)
23
24 add_custom_target(logging_ut
25     COMMAND ${CMAKE_CTEST_COMMAND} --output-on-failure
26     DEPENDS logging_ut_exe
27 )
28
```