

# Deep C++(V17.02)

autor:ichiroprogrammer

## インデックス

### SOLID

单一責任の原則(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)  
まとめ

### デザインパターン

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

### テンプレートメタプログラミング

ログ取得ライブラリの開発  
Nstdライブラリの開発  
メタ関数のテクニック  
Nstdライブラリの開発2  
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  
注意点まとめ

### ダイナミックメモリアロケーション

malloc/freeの問題点  
グローバルnew/deleteのオーバーロード  
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

### 用語解説

型とインスタンス  
オブジェクトと生成  
オブジェクトのコピー  
name lookupと名前空間

[expressionと値カテゴリ](#)

[リファレンス](#)

[エクセプション安全性の保証](#)

[シンタックス、セマンティクス](#)

[C++コンパイラー](#)

[C++その他](#)

[ソフトウェア一般](#)

[非ソフトウェア用語](#)

[Sample Code](#)

[C++](#)

# 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。

- [单一責任の原則\(SRP\)](#)
- [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\(DIP\)](#)

## 单一責任の原則(SRP)

单一責任の原則(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());
```

## 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);
        ...
    }
}

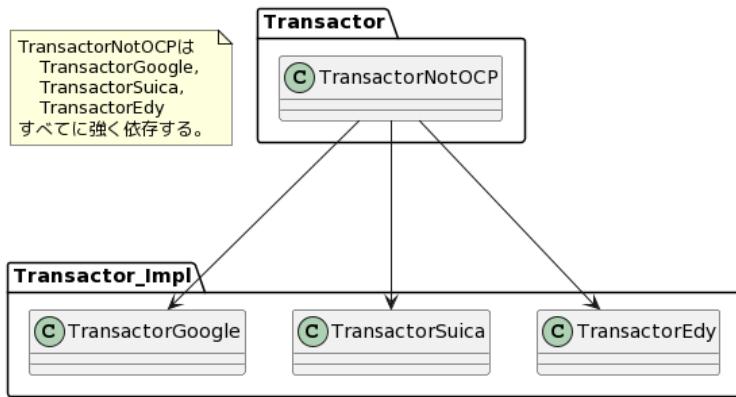
bool Pay(Yen price) noexcept
{
    switch (pay_method_) {
        case TransactionMethod::Google:
            return TransactorGoogle::Pay(price);
        case TransactionMethod::Suica:
            return TransactorSuica::Pay(price);
        ...
    }
}
...
};


```

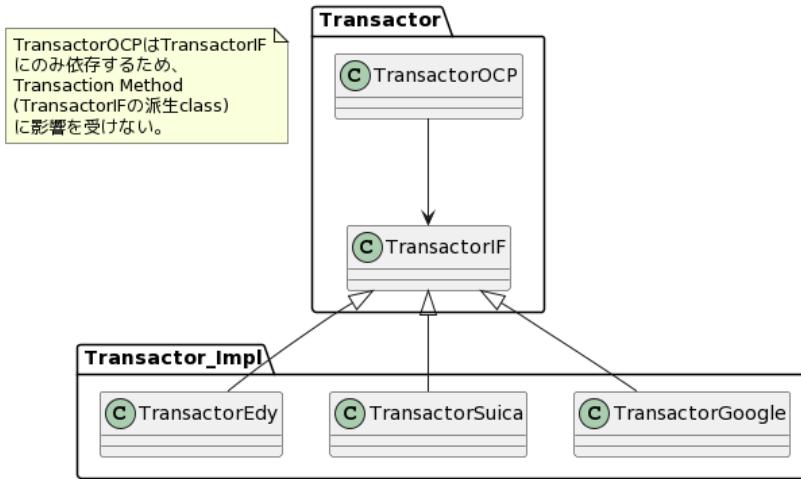
Transaction Method(enum TransactionMethod)が増えた場合、少なくとも3か所に手を入れなければならなくなる(修正に対してclosedでない)。従って、下記のTransactorNotOCP::Charge()や、TransactorNotOCP::Pay()は Transaction Methodの追加、変更に対して、脆弱な構造だと言える。

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。

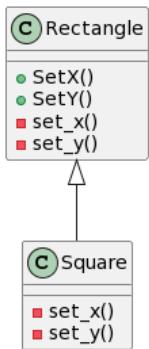
## リスコフの置換原則(LSP)

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ってはならない。

- ・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且つ、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す。

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、エクゼプション記述子は、関数のエクゼ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)

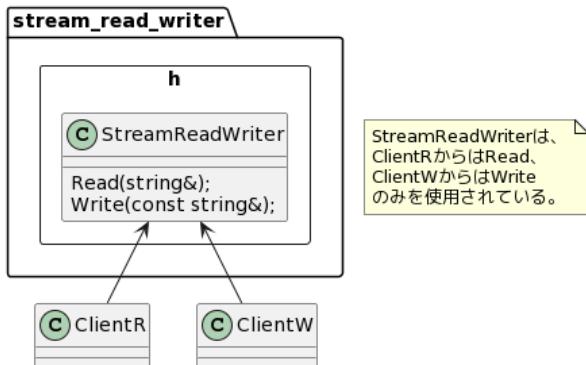
インターフェース分離の原則 (ISP, Interface Segregation Principle)とは、

- 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。

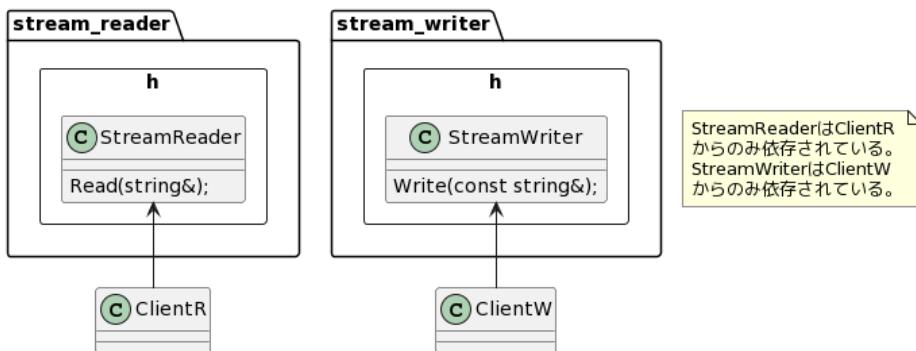
- 一つのヘッダファイルに互いが密接な関係を持たない複数のクラスを定義、宣言すべきでない。
- 一つのヘッダファイルにそのファイルのコンパイルに不要なヘッダファイルをインクルードすべきでない。

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、このような逡巡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。

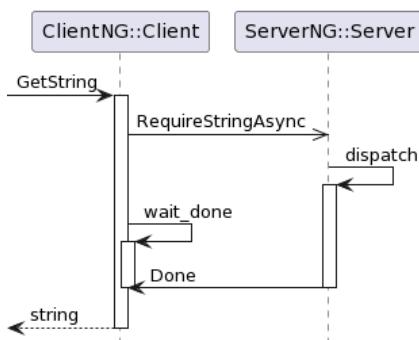
## 依存関係逆転の原則(DIP)

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;
    ...
};

} // namespace ServerNG
```

```
// @@@ 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));
}
... // namespace ServerNG
```

```
// @@@ 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};

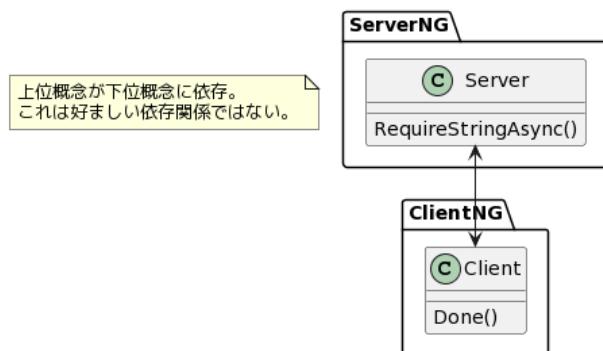
    auto actual = client.GetString(1);
    ASSERT_EQ("hello", actual);

    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

```

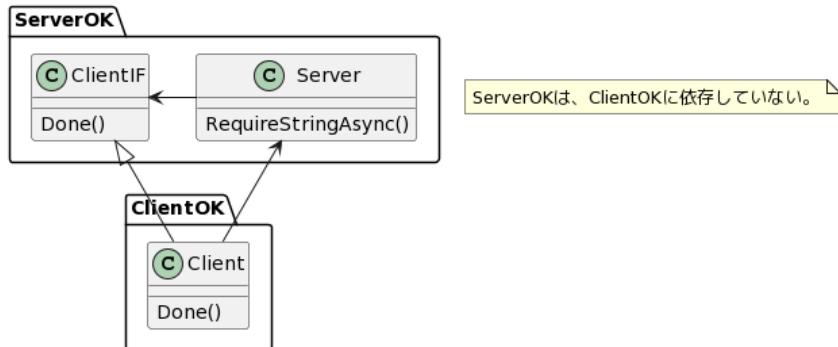
```
// @@@ example/solid/dip_ut.cpp 28
// 使用方法は、ServerNG, ClientNGと同じ。
TEST(DIP, ok_pattern)
{
    auto server = ServerOK::Server{};
    auto client = ClientOK::Client{server};

    // 以下、ng_patern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、クライ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。

## 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。

# 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。

## 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、以下のようない点がある。

- 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つた
- 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。

## 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 PhisicalAbility { // 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PhisicalAbility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 : phisical_ability_{pa} {}

    uint32_t GetPhisicalAbility() const noexcept { return phisical_ability_; }

    ...
};
```

```
// @@@ example/design_pattern/enum_operator_ut.cpp 13

Animal dolphin{Animal::PhisicalAbility::Swim}; // OK
ASSERT_EQ(Animal::PhisicalAbility::Swim, dolphin.GetPhisicalAbility());

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 PhisicalAbility : uint32_t {
        Run = 0b0001,
        Fly = 0b0010,
        Swim = 0b0100,
    };

    explicit Animal(PhisicalAbility pa) noexcept : phisical_ability_{pa} {}

    PhisicalAbility GetPhisicalAbility() const noexcept { return phisical_ability_; }

private:
    PhisicalAbility const phisical_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::PhysicalAbility{0xff}};

auto dolphin = Animal{Animal::PhysicalAbility::Swim};
ASSERT_EQ(Animal::PhysicalAbility::Swim, dolphin.GetPhysicalAbility());

auto pa = Animal::PhysicalAbility{Animal::PhysicalAbility::Run};
pa |= Animal::PhysicalAbility::Swim;

auto human = Animal{pa};
ASSERT_TRUE(IsTrue(Animal::PhysicalAbility::Run & human.GetPhysicalAbility()));

```

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::GetPhysicalAbility()の戻り値もenumになり、これも誤用が難しくなった。

## 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ライブラリのパースは多くの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ヘッダファイルはパースされる。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トは遅くなる。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>()} {}

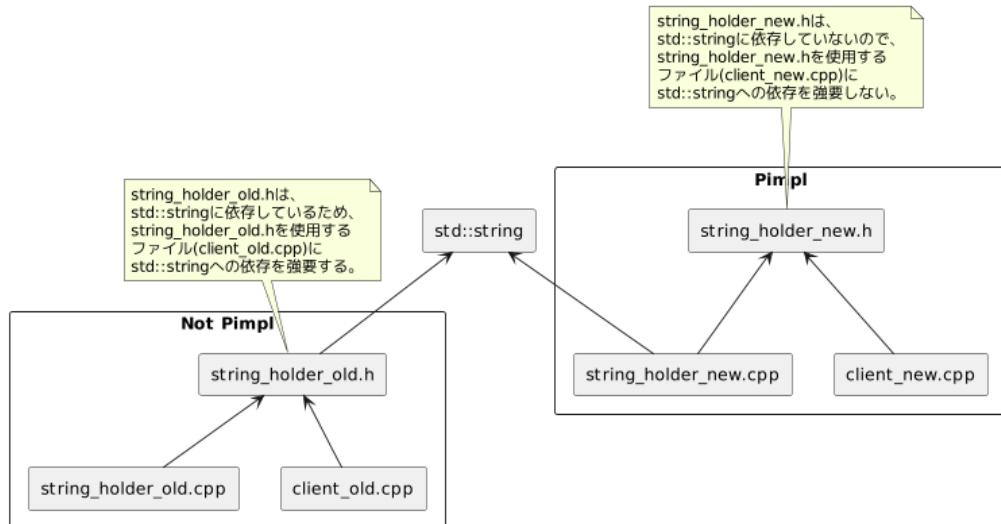
void StringHolderNew::Add(char const* str) { core_->Add(str); }

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());
}

```

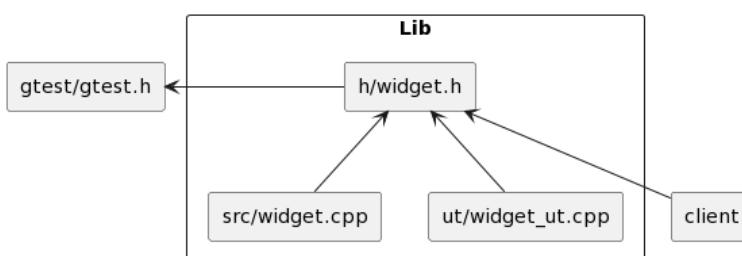
何らかの事情により、単体テストでprivateなメンバにアクセスする必要があったため、 単体テストクラスをテスト対象クラスのfriendすることで、それを実現している。

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;

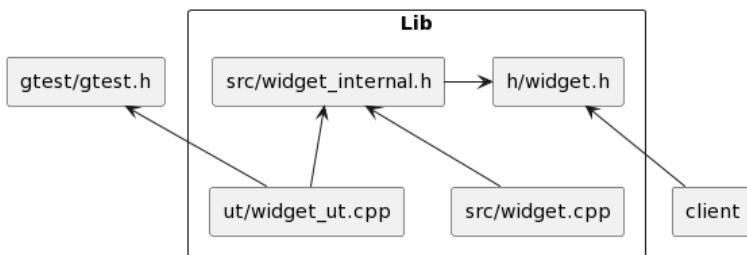
    ASSERT_EQ(1, wi.xxx_data_);
    wi.DoSomething();
    ASSERT_EQ(3, wi.xxx_data_);
    ASSERT_EQ(9, wi.gen_xxx_data(3));

    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。



## 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かの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。

## 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_(は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。

## 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。

## 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になった。

## 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()への依存が集中し、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。

## 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; // この行以前に函数を抜けると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

// RAIIf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_RAIIfor_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 RAIIf未使用
        // Do something to recover
        ...
    }

    return;
}
...
close(fd); // NG RAIIf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 RAIIf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

// RAIIf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_RAIIfor_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。

## 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、目的的に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。

## 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)} {}

    ...
};

```

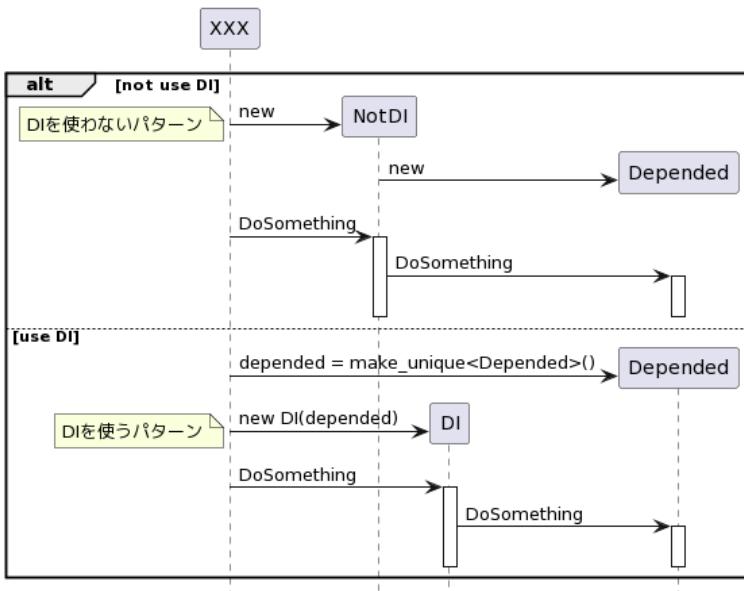
```

        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           // 単体テストビルド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());
}
```

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。

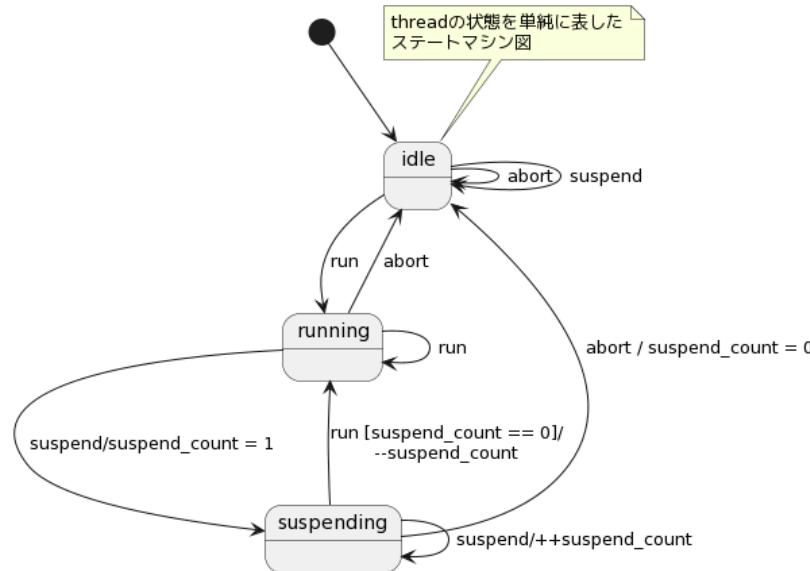
Singletonオブジェクトの初期化(最初のコンストラクタ呼び出し)は、C++03以前はスレッドセーフでなかったため、「Double Checked Lockingを使って競合を避ける」か、「他のスレッドを起動する前にメインスレッドから各SingletonのInstConst()を呼び出す」ことが必要であった。 C++11から上記例のようなSingletonオブジェクトのコンストラクタ呼び出しがスレッドセーフとなったため、このような黒魔術が

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::Suspended:
            --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("Suspended", ThreadOldStyleStateStr()); // suspend_count_ == 1

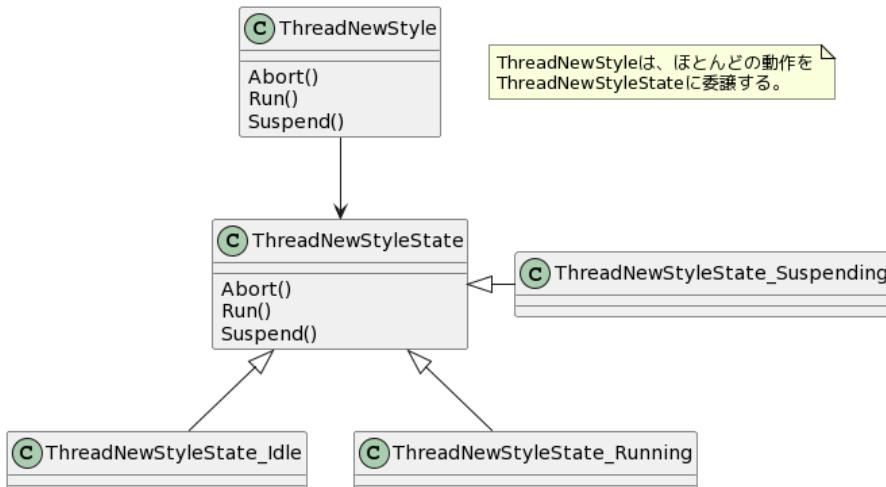
    ThreadOldStyleSuspend();
    ASSERT_EQ("Suspended", ThreadOldStyleStateStr()); // suspend_count_ == 2

    ThreadOldStyleRun();
    ASSERT_EQ("Suspended", 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_idle;
    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_suspending;
    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());
}

```

## 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。

## 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>";
}
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、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);
}

```

## 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

#if 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();

```

## 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;

    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。

## 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ークラス
2. そのサーバーをラッピングして、使いやすくしたサーバークラス(Facadeパターン)
3. サーバー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ーの実装例である。

```

// @@@ 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、このむ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 5

/// @fn bare_wrapper_client
/// @brief BareServerを使いやすくラップした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 77

/// @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
    }

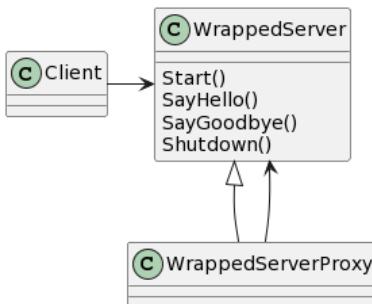
    return hello_cashe_;
}

std::string WrappedServerProxy::say_goodbye()
{
    hello_cashe_ = std::string{}; // helloキャッシュをクリア

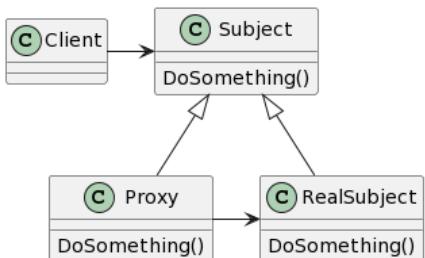
    return WrappedServer::say_goodbye();
}

```

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。



## 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g
- ・「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仮引数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に起点となるパス
/// @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);
}

```

検索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};

#define SHIPPING == SHIP_TO_JAPAN
        // 日本向けの何らかの処理
#define SHIPPING == SHIP_TO_US
        // US向けの何らかの処理
#define SHIPPING == SHIP_TO_JAPAN
        // EU向けの何らかの処理
    }
}

```

```

#ifndef "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();

```

## 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);
        }
    }
}

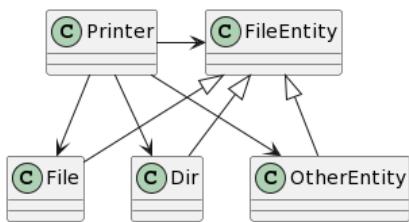
```

```

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(); }
};

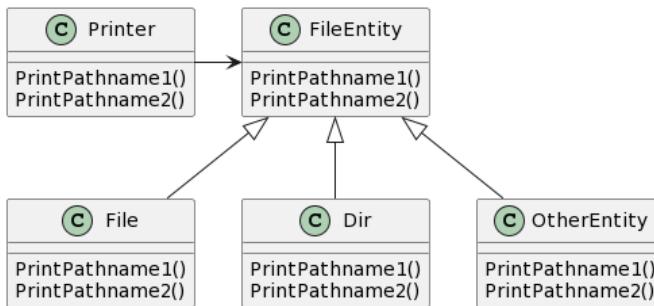
class Dir final : public FileEntity {
public:
    ...
    virtual void PrintPathname1() const override { std::cout << Pathname() + "/"; }
    virtual void PrintPathname2() const override { std::cout << find_files(Pathname()); }
};

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)"; }

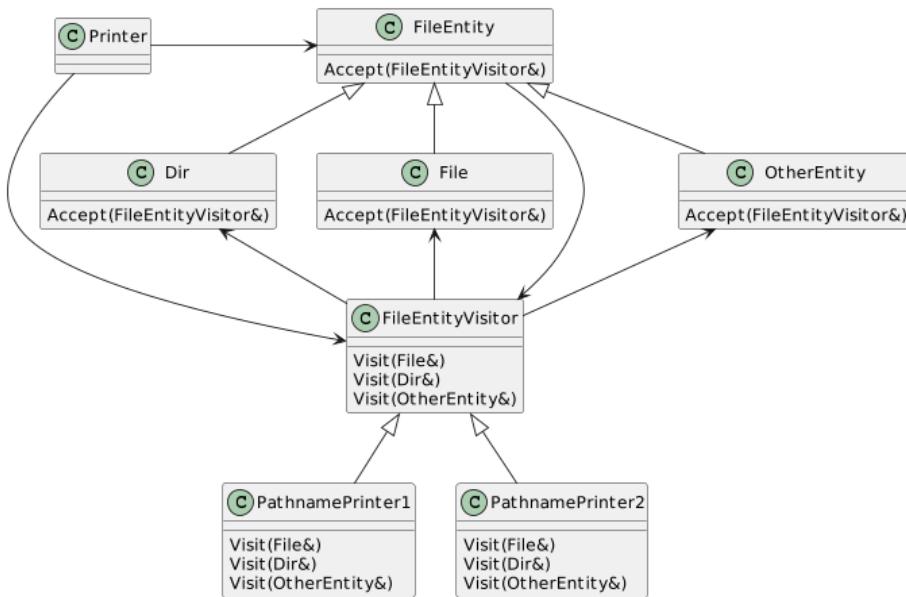
void PathnamePrinter2::Visit(File const& file) { std::cout << file.Pathname(); }
void PathnamePrinter2::Visit(Dir const& dir) { std::cout << find_files(dir.Pathname()); }
void PathnamePrinter2::Visit(OtherEntity const& other) { std::cout << other.Pathname() + "(o2)"; }

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

```

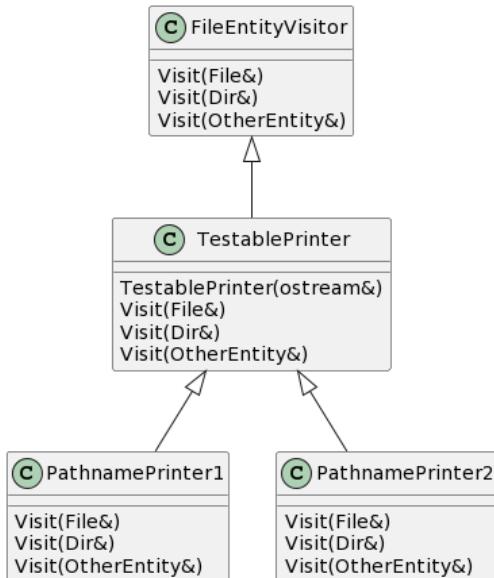
// @@@ example/design_pattern/visitor.h 39

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/gile3", oss.str());
        oss = {};
    }
}

```

## 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);

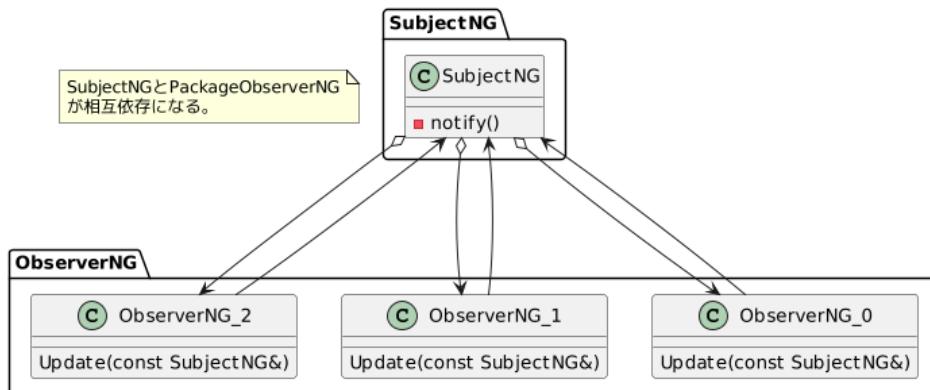
subject.SetNum(1);
subject.SetNum(2);

ASSERT_EQ(2, ng0.call_count);
ASSERT_EQ(2, *ng0.num);

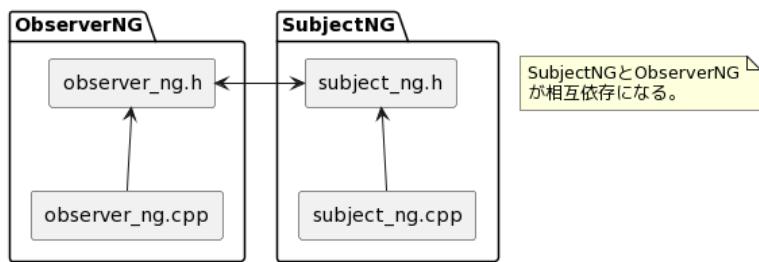
subject.SetNum(2); // 同じ値をセットしたため、Updateは呼ばれないはず
ASSERT_EQ(2, ng0.call_count);
ASSERT_EQ(2, *ng0.num);

```

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);

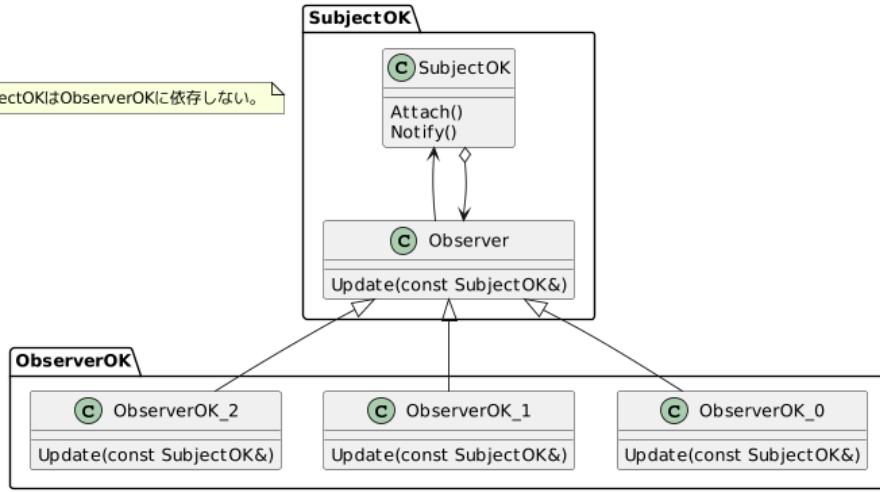
subject.SetNum(1);
subject.SetNum(2);

ASSERT_EQ(2, ok0.call_count);
ASSERT_EQ(2, *ok0.num);

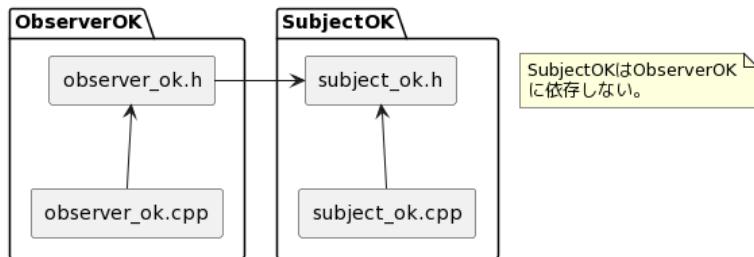
subject.SetNum(2); // 同じ値をセットしたため、Updateは呼ばれないはず
ASSERT_EQ(2, ok0.call_count);
ASSERT_EQ(2, *ok0.num);

```

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)。



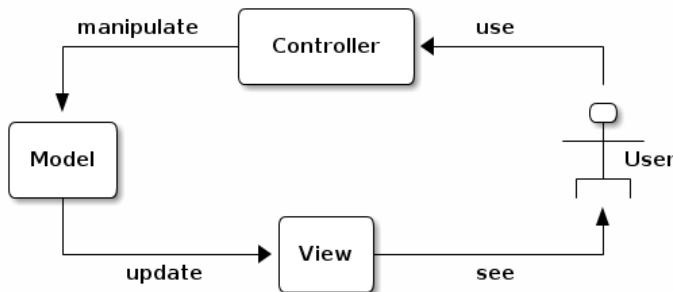
## MVC

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。

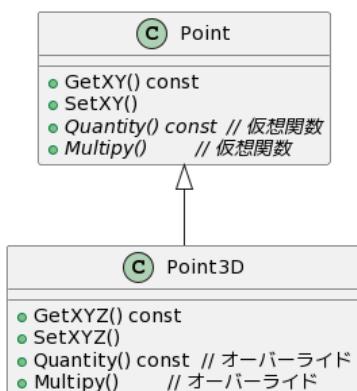
## Cでのクラス表現

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)

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 Multiply(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.Multiply(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 Multiply)(Point* self, int m);
};

static int point_quantity(Point const* self) { return self->x * self->y; }

static void point_multiply(Point* self, int m)
{
    self->x *= m;
    self->y *= m;
}

Point Point_Construct(int x, int y)
{
    Point ret = {x, y, point_quantity, point_multiply}; // 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.Multiply(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_multiply(Point* self, int m)
{
    point_multiply(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_multiply}, 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->Multiply(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。

# 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、この増加を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。

## 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ぬため、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、今日もそれを行なう」という諺を思い出すと気持ちは分からぬもないが、C++ではprintf(...)のような危険な可変長引数を取る関数を作つてはならない。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, "argument 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, "argument 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`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。

### 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、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はその中に宣言した。

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。単体テストでのusing namespace App3はLogging::Logger::set\_innerの宣言より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戻る。 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<<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;
    }

    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

ここで解決したい问题是、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。

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ロー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）になってしまふかも知れない）。
- 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ったが、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発見できなかった。「バーニーの祈り」が言っている通り、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っている。

### Ints\_tのログ登録のまとめ

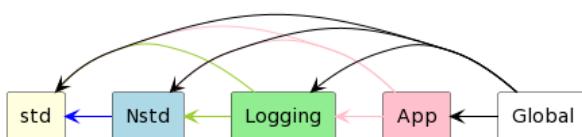
製品開発では、満足できる仕様の関数やクラスが作れず、妥協せざるを得ないことはよくあることである。このような場合、将来、良いアイデアが見つかった時に備えて、妥協コードを簡単に修正できるようなレベルにした後、捲土重来を期してさっさと退却するのがベストである。ただし、漫然と過ごしても良いアイデアは浮かばない。時間を作り、関連書籍やウェブドキュメント等を読み、学習を継続する必要があることは言うまでもない。

## 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、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[]経由でのメンバー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
}

```

## 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の他にすぐなとも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::string; // 繙承コンストラクタ

    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; // 繙承コンストラクタ

    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 '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)。

## 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(std::is_convertible_v<int, int>);
static_assert(std::is_convertible_v<int, unsigned>);
static_assert(!std::is_convertible_v<"", int>); // ポインタはintに変換不可

struct ConvertibleToInt {
    operator int();
};

struct NotConvertibleToInt {};

static_assert(std::is_convertible_v<ConvertibleToInt, ConvertibleToInt>);
static_assert(!std::is_convertible_v<NotConvertibleToInt, ConvertibleToInt>);

// 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でもある。

### 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ための措置である。

## 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。

## 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>);
```

## 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()};
```

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

であるため、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\_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>);
```

### 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を使うことが多いいため、

「is\_void\_ena\_sの実装」でその例を示す。

## 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> {
    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_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

```

$T \neq U$ 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なになる。

```

// @@@ 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なになる。

```

// @@@ 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_tmpl : is_same_sfinae_s<T, TEMPL<ARGS...>> {  
};  
  
template <typename T, template <class...> class TEMPL, typename... ARGS>  
constexpr bool is_same_tmpl_v{is_same_tmpl<T, TEMPL, ARGS...>::value};
```

上記のis\_same\_tmpl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なる。

```
// @@@ example/template/is_same_ut.cpp 202  
  
static_assert(is_same_tmpl_v<std::string, std::basic_string, char>);  
static_assert(!is_same_tmpl_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_tmpl<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>);
```

### 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
```

IsSameSomeOfV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、IsSameSomeOfV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の読み出しは、IsSameSomeOfV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*>);
```

## 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せる{}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<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_sfae_f	メンバ関数void func()を持つかどうかの判断
exists_void_func_sfae_s	同上
exists_void_func_sfae_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

メタ関数名	メタ関数の目的
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SFINAE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SFINAE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SFINAE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>);
```

### 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rvalue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の引数になれることに気づけば、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”“の後ろに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]>);
```

### 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の実装は以下のようになる。

```
// @@@ 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_exits_put_to>);
static_assert(!Nstd::ExistsPutToV<test_class_not_exits_put_to>);
static_assert(Nstd::ExistsPutToV<test_class_not_exits_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ない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<>`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<typename ValueType<typename ValueType<T>::type_direct>::template 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::IsRangeV<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 = 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いな、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つてから、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つ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が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<
        ValueType<T>::Nest != 0,
        typename ValueType<typename ValueType<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_v<array_or_container<T>::value>};
} // namespace Inner_

template <typename T> // ValueTypeの特殊化
struct ValueType<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{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>;
} // namespace Nstd

```

## 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};
};

} // namespace Nstd
```

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\_temp」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(→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行えるようになった。

```

// @@@ 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には、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

であるため、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
#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());
}

```

```

    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().Set(__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。実際に解決できる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。

## 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;
}

```

```

        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

```

この原因は、「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、残念ならが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rvalue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ても実装可能
    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
    }
}
```

```

        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 = 3;
    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が呼び出せない場合、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な方法を選ぶべきだろう(が、何が単純かも一概に決めるのは難しい)。

### 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std::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<<(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、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が次の閑門になる。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 = 27U; // 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。
- ・ 下記のようにエクセ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]);

```

これを下記のようを使うことで、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()> const, decltype(fs0)>);
static_assert("1234567" == fs0);

constexpr auto fs1 = StaticString{"1234"} + ":";  

static_assert(std::is_same_v<StaticString<fs1.Size()> const, 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進数一桁を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る。

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 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_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 calded_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のテンプレートパ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。

## 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に使用する。
- 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。
- 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」、「ユニバーサルリファレンスと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。
- 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的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](#)」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](#)」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](#)」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上の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。

## 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
{
    ...
}
} // namespace
```

```
// @@@ 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、下のフラグ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セ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(nullptrptr, 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(nullptrptr, 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(nullptrptr, 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) / 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(=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、「割り込みディセイブル/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 size) 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にある。

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、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& lhs, 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、下記のようにすることで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することで、`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で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で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で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くべきである。

## 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いつた要望を叶えることができる。

関数	機能
<code>_cxa_allocate_exception(size_t thrown_size)</code>	エクセプション処理用のメモリ確保
<code>_cxa_free_exception(void* thrown_exception)</code>	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、安易にエクゼプションや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。

# 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。

## 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つのオペラントが他のオペラントと同じ型でない場合、1つのオペラントを他のオペラントと同じ型に変換するプロセスのことを指す。

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, "");

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。

従つて、`sizeof(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); // 沢整数拡張により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 = Templ<a>{}; // aはconstexprなのでaの初期化が可能
auto templ_f = Templ<f(a)>{}; // f(a)はconstexprなのでaの初期化が可能
// auto templ_x = Templ<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 Allways2() const noexcept { return 2; } // constexprメンバ関数はconst
static constexpr int32_t Allways3() 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	<code>= default</code>	<code>= default</code>	<code>= default</code>	<code>= default</code>	<code>= default</code>	<code>= default</code>
非デフォルトコンストラクタ	宣言なし	<code>= default</code>	<code>= default</code>	<code>= default</code>	<code>= default</code>	<code>= default</code>
デフォルトコンストラクタ	-	<code>= default</code>	<code>= default</code>	<code>= default</code>	<code>= default</code>	<code>= default</code>
デストラクタ	<code>= default</code>	-	<del><code>= default</code></del>	<del><code>= default</code></del>	宣言なし	宣言なし
copyコンストラクタ	宣言なし	<code>= default</code>	-	<del><code>= default</code></del>	宣言なし	宣言なし
copy代入演算子	<code>= default</code>	<code>= default</code>	<del><code>= default</code></del>	-	宣言なし	宣言なし
moveコンストラクタ	宣言なし	<code>= default</code>	<code>= delete</code>	<code>= delete</code>	-	宣言なし
move代入演算子	<code>= default</code>	<code>= default</code>	<code>= delete</code>	<code>= delete</code>	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()

```

```

{
#ifndef 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};

    ...

#ifndef 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 i1{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 LastDeconstructedNum() 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

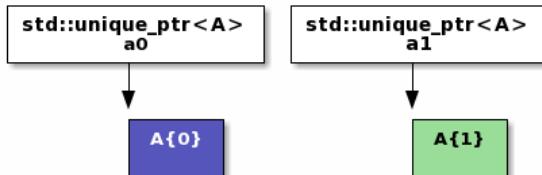
// ステップ0
// まだ、クラスAオブジェクトは生成されていないため、
// A::LastConstructedNum()、A::LastDeconstructedNum()は初期値である-1である。
ASSERT_EQ(-1, A::LastConstructedNum());           // まだ、A::A()は呼ばれてない
ASSERT_EQ(-1, A::LastDeconstructedNum());          // まだ、A::~A()は呼ばれてない
```

```
// @@@ 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::LastDeconstructedNum());          // まだ、A::~A()は呼ばれてない
```

ステップ1

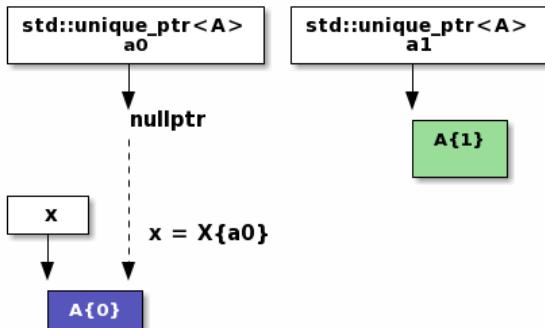


<code>A::LastConstructedNum()</code>	1
<code>A::LastDestructedNum()</code>	-1

```
// @@@ 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
```

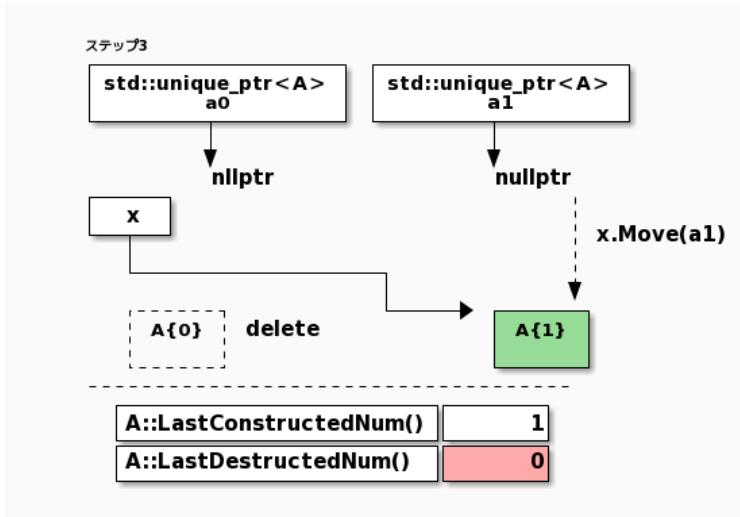
ステップ2



<code>A::LastConstructedNum()</code>	1
<code>A::LastDestructedNum()</code>	-1

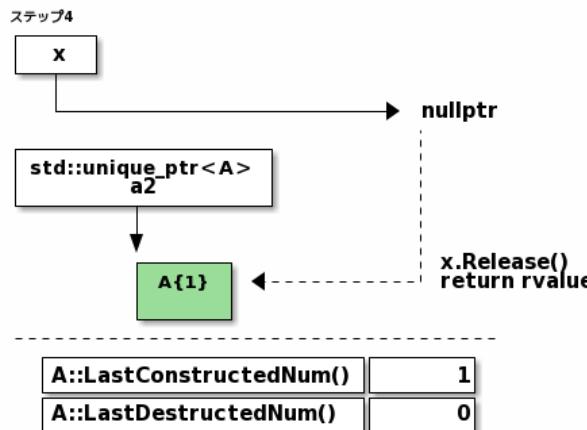
```
// @@@ example/term_explanation/unique_ptr_ownership_ut.cpp 75
```

```
// 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
```



```
// @@@ example/term_explanation/unique_ptr_ownership_ut.cpp 88
```

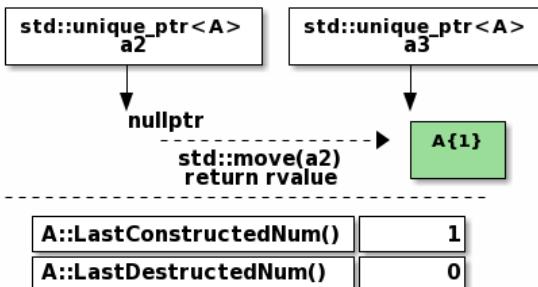
```
// ステップ4
// x.ptr_はstd::unique_ptr<A>であるため、ステップ3の状態では、
// x.ptr_はA{1}オブジェクトのポインタを保持しているが、
// x.Release()はそれをrvalueに変換し戻り値にする。
// その戻り値をa2で受け取るため、A{1}の所有はxからa2に移動する。
std::unique_ptr<A> a2{x.Release()};           // xからa2へA{1}の所有権の移動
ASSERT_EQ(nullptr, x.GetA());                  // xは何も所有していない
ASSERT_EQ(1, a2->GetNum());                  // a2はA{1}を所有
```



```
// @@@ 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
```

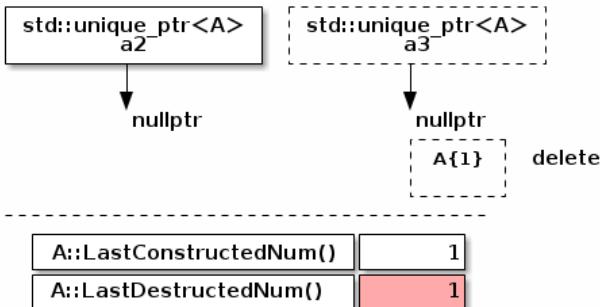
ステップ5



```
// @@@ 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



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所有

オブジェクトの共有所有や、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
```

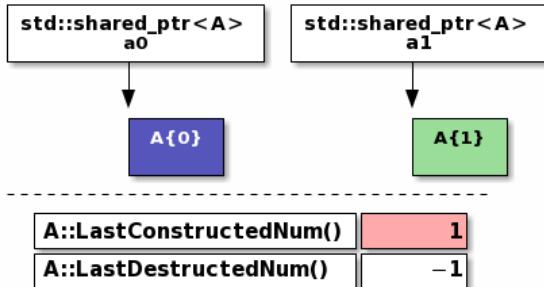
```

// 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

```

ステップ1



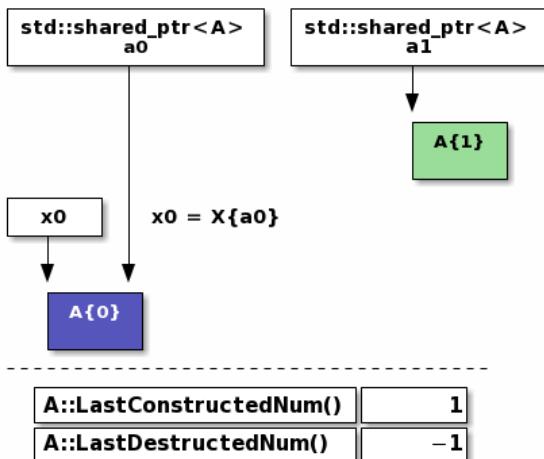
```
// @@@ example/term_explanation/shared_ptr_ownership_ut.cpp 68
```

```

// 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



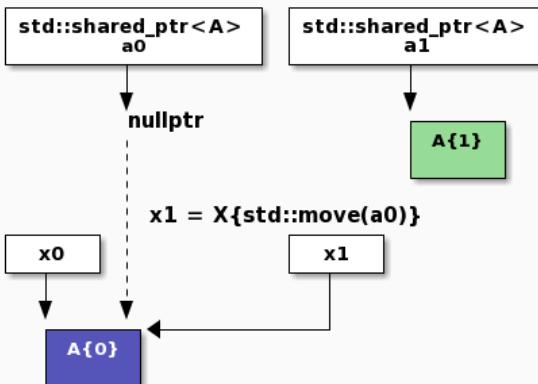
```
// @@@ example/term_explanation/shared_ptr_ownership_ut.cpp 79
```

```

// 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());          // a0は何も所有していない
ASSERT_FALSE(a0);

```

ステップ3

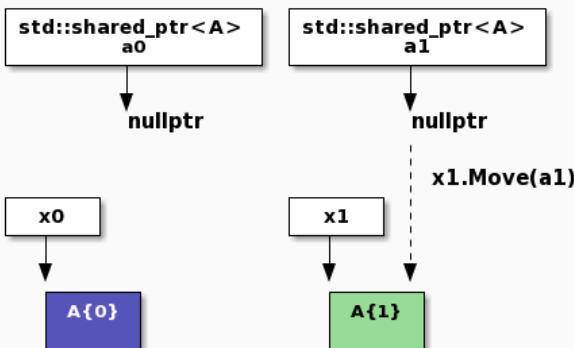


<code>A::LastConstructedNum()</code>	1
<code>A::LastDestructedNum()</code>	-1

```
// @@@ 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
```

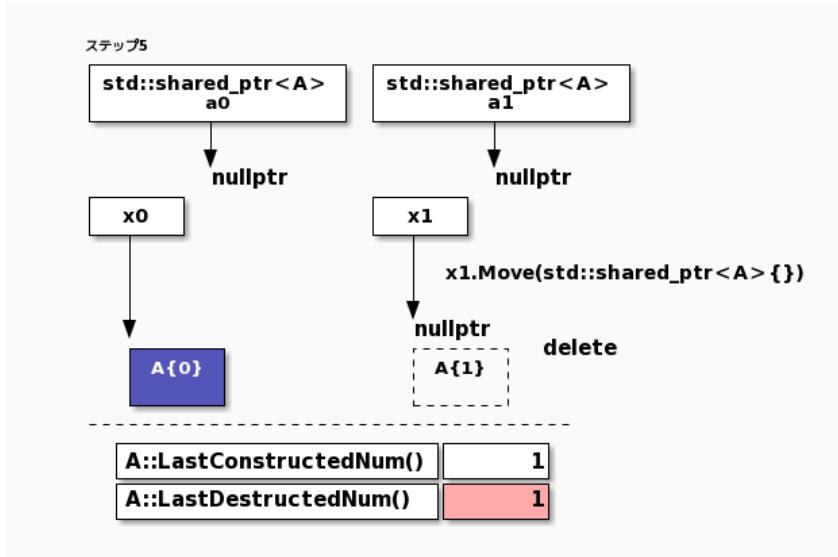
ステップ4



<code>A::LastConstructedNum()</code>	1
<code>A::LastDestructedNum()</code>	-1

```
// @@@ example/term_explanation/shared_ptr_ownership_ut.cpp 110
```

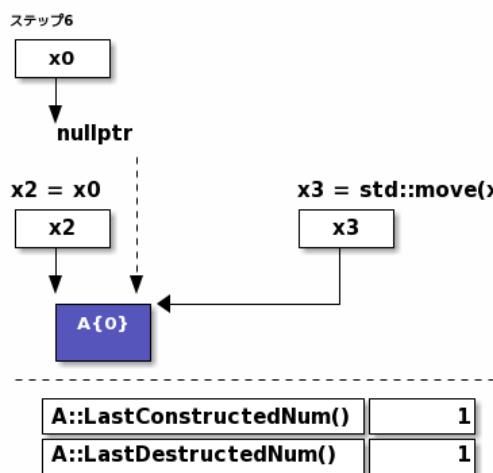
```
// ステップ5
// 現時点でx1はA{1}オブジェクトを保持している。
// x1::Moveに空のstd::shared_ptrを渡すことにより、A{1}を解放する。
x1.Move(std::shared_ptr<A>{});           // x1に空のstd::shared_ptr<A>を代入することで、
                                         // A{1}を解放
ASSERT_EQ(nullptr, x1.GetA());            // x1は何も保持していない
ASSERT_EQ(1, A::LastDestructedNum());     // 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

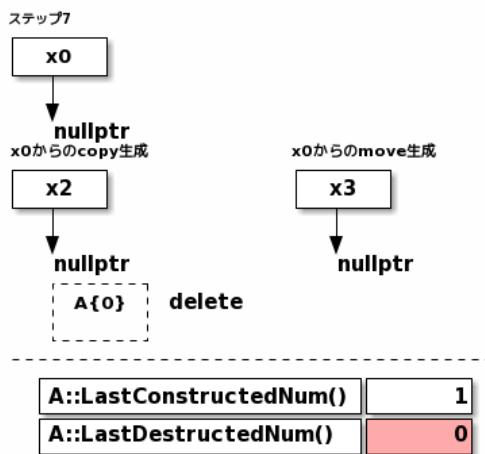
    auto x3 = std::move(x0);                      // x0からx2をmove生成、x0はA{0}の所有を放棄
    ASSERT_EQ(nullptr, x0.GetA());
    ASSERT_EQ(0, x2.GetA()->GetNum());            // x2はA{0}を保有
    ASSERT_EQ(x2.GetA(), x3.GetA());              // x2、x3はA{0}を共有保有
    ASSERT_EQ(2, x2.UseCount());                  // A{1}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>();
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};

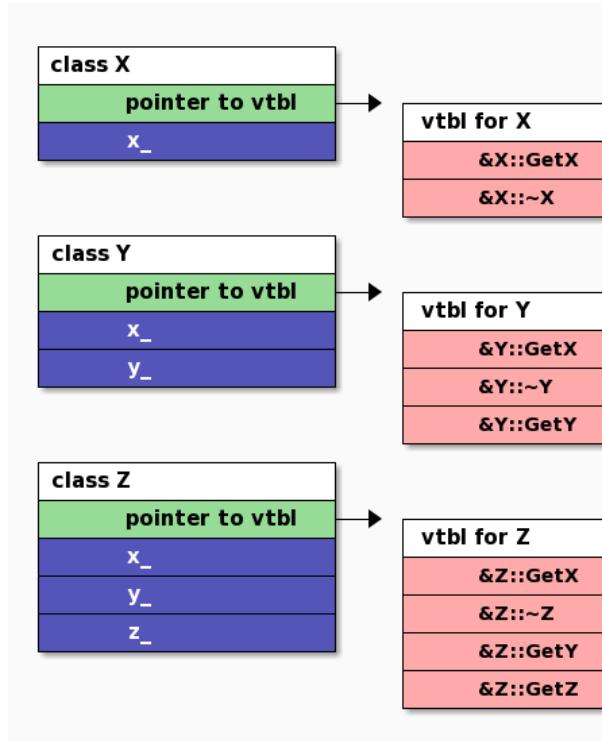
};

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のコンストラクタ呼び出し処理
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のコンストラクタ呼び出し処理
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のコンストラクタ呼び出し処理
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()の呼び出しとなる(Z::GetX()の呼び出しとはならない)。

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()); // 本来ならこうなってほしいが、
#ifndef NDEBUG
    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
}

```

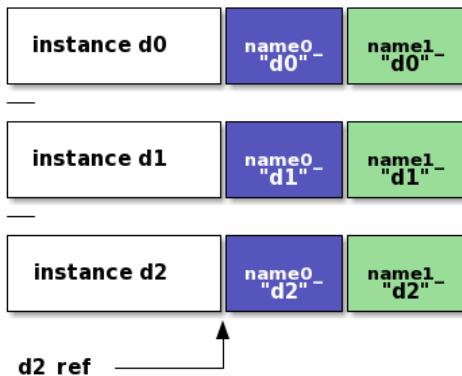
copy代入演算子(=)によりコピーが行われた場合、=の両辺のオブジェクトは等価になるべきだが (copy代入演算子をオーバーロードした場合も、そうなるように定義すべきである)、スライシングが起こった場合、そうならないことが問題である(「等価性のセマンティクス」参照)。

下記にこの現象の発生メカニズムについて解説する。

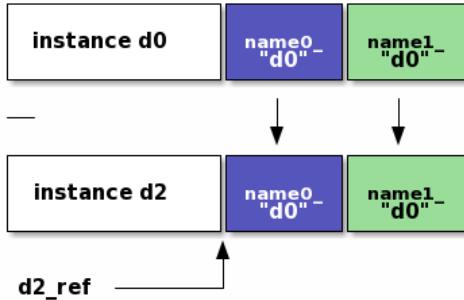
- 上記クラスBase、Derivedのメモリ上のレイアウトは下記のようになる。



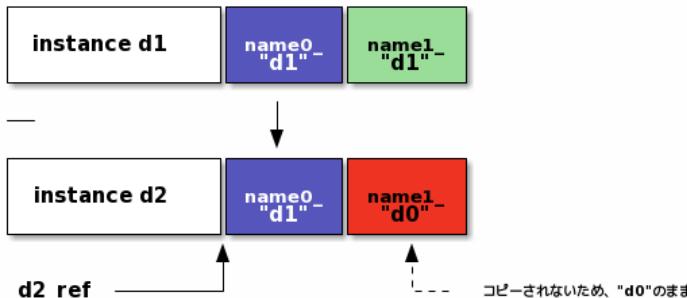
- 上記インスタンスd0、d1、d2、d2\_refのメモリ上のレイアウトは下記のようになる。



- 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。

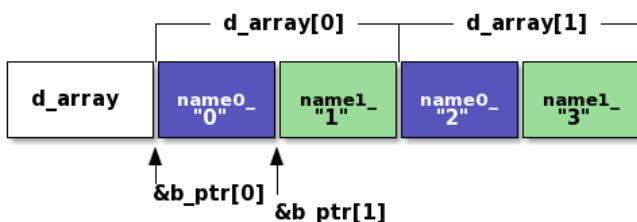
```
d2_ref.Base::operator=(d1); // Base::operator=(Base const&)が呼び出される
```

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

と呼ぶこととする。

下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()呼び出しより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< 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()呼び出しより前方で宣言された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< 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< 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)の呼び出しは、 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\_TPLU2::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の対象となるない。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& y, int i)はTypeNum内での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を指定しなくとも見つかる
```

- 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は思ぬ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なように、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になった)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f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{};
#ifndef
    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
{
#ifndef
    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)の呼び出しがコンパイルできるが、name-hidingが原因で、NS\_B\_Inner::h()内のf(int)の呼び出しがコンパイルできず、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)の呼び出しがコンパイルできる。

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());

```

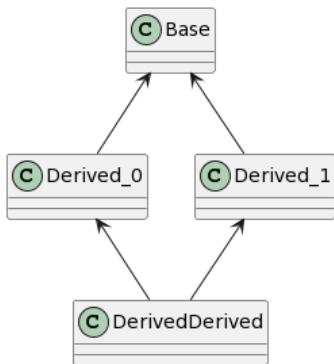
これまでと同様にDerived\_0のドミナンスによりBase::fはname-hidingされることになる。この時、Derived\_0、Derived\_1がBaseから仮想継承した効果により、この継承ヒエラルキーの中でBaseは1つのみ存在することになるため、Derived\_1により導入されたBase::fも併せてname-hidingされる。結果として、曖昧性は排除され、コンパイルエラーにはならず、このような結果となる。

## 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」仕様と呼ぶこととする。

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
{
#ifndef 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

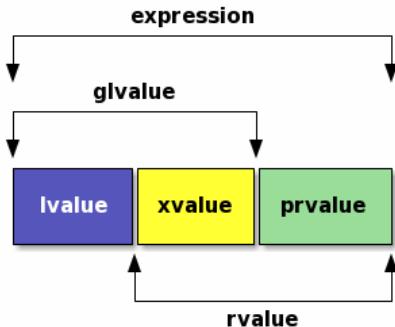
ここでは、expression(式)の値カテゴリや、それに付随した機能についての解説を行う。

### 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
  - メンバーアクセス(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」と考えても差し支えない。

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 NDEBUG
    // 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

```

```

// 下記のようす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。

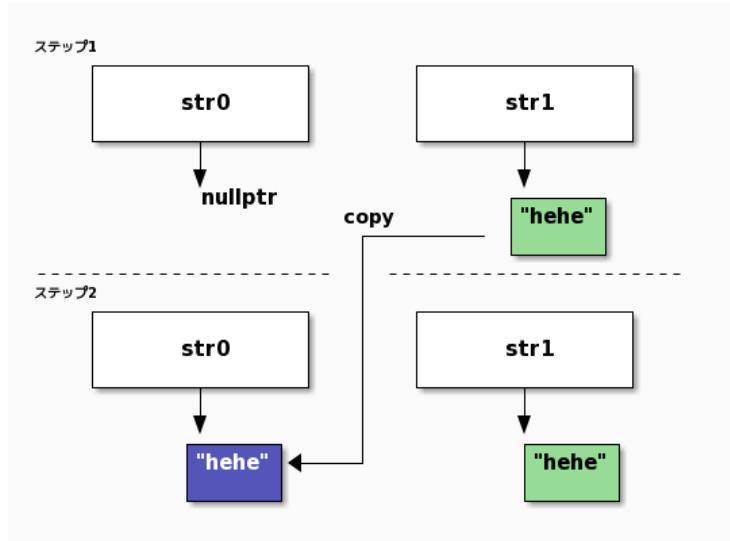
```

// @@@ 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。

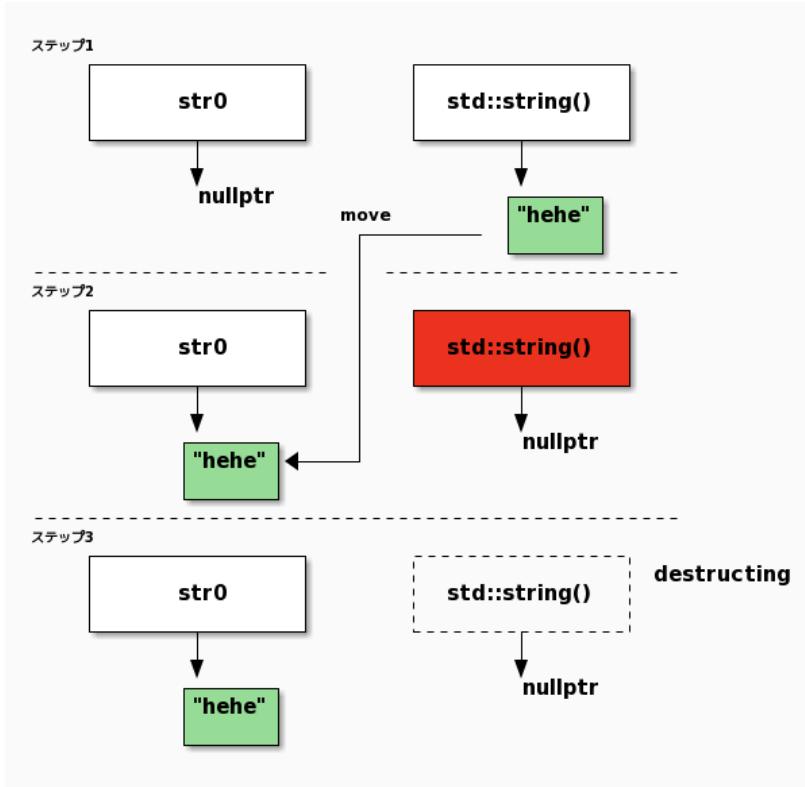
```

// @@@ 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

```

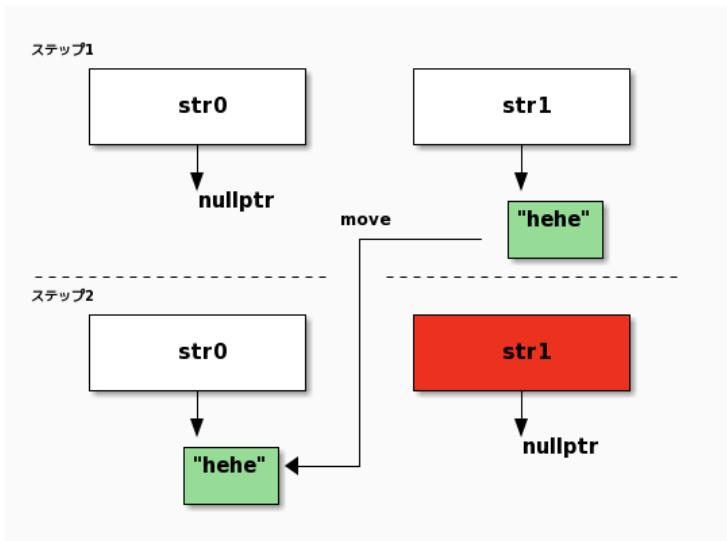
- ステップ1。 str0、 「`std::string()`により作られたテンポラリオブジェクト」がそれぞれ初期化される (“hehe”を保持するバッファが生成され、それをテンポラリオブジェクトが所有する)。
- ステップ2。“hehe”を保持する文字列バッファをもう1つ作る代わりに、 テンポラリオブジェクトが所有している文字列バッファを `str0` の所有にする。この代入をmove代入と呼ぶ。
- ステップ3。 テンポラリオブジェクトが解体されるが、 文字列バッファは `str0` の所有であるため `delete` する必要がなく、 実際には何もしない。 move代入によって、 文字列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
decltype(lvalue)	T
decltype((lvalue))	T&
decltype(xvalue)	T&&
decltype((xvalue))	T&&
decltype(prvalue)	T
decltype((prvalue))	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
//      * 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"};
auto const rstr = std::string{"rvalue"};

ASSERT_EQ(0, value_type(str));
ASSERT_EQ(1, value_type(cstr));
ASSERT_EQ(2, value_type(rstr));
```

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つたが、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っ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
  - 「單一責任の原則(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	$x = y$ 且つ $y = z$ 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{d}, 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_d = static_cast<DerivedDerived const&>(rhs);

    return Derived::is_equal(rhs) && dd_ == rhs_d.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、 $c = std::move(a)$ 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。
        return *this;
    }

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

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"; }

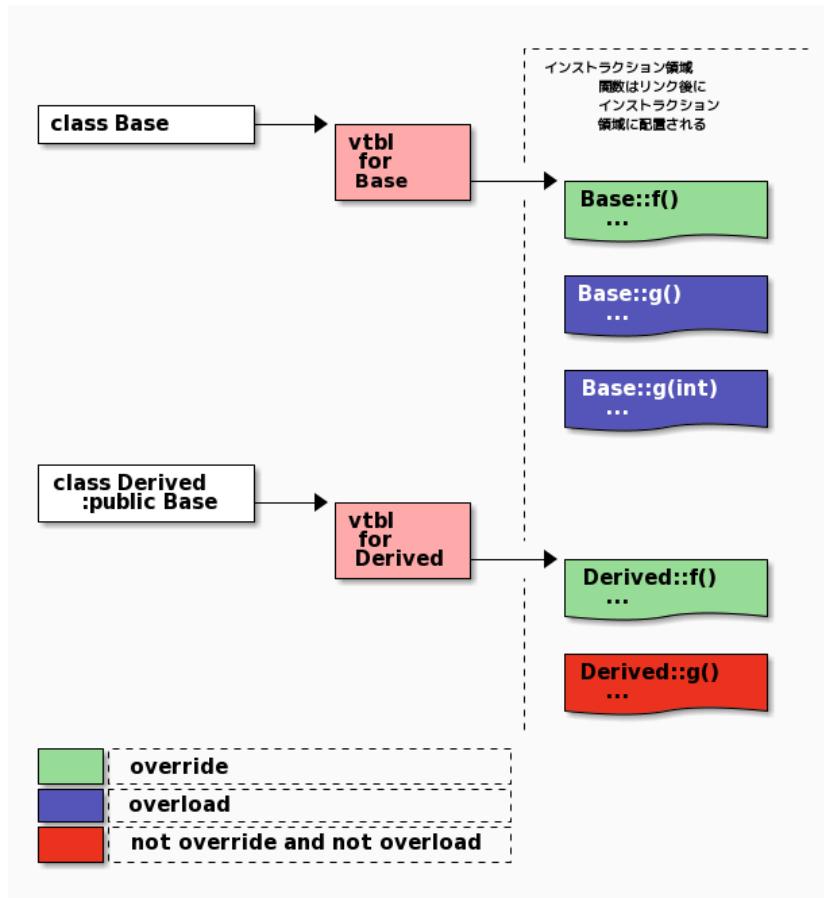
    // Base::gのname-hiding

```

```
    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・リ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・単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。

SOO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); // キャストできない場合、nullptrが返る

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と指す。

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あたえる。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、 [CRTP(curiously recurring template pattern)] ([https://ja.wikibooks.org/wiki/More\\_C%2B%2B\\_Idioms/%E5%A5%87%E5%A6%99%E3%81%AB%E5%86%8D%E5%B8%B0%E3%81%97%E3%81%9F%E3%83%86%E3%83%BC%E3%83%97%E3%83%AC%E3%83%BC%E3%83%88%E3%83%91%E3%82%BF%E3%83%83%BC%E3%83%BC](https://ja.wikibooks.org/wiki/More_C%2B%2B_Idioms/%E5%A5%87%E5%A6%99%E3%81%AB%E5%86%8D%E5%B8%B0%E3%81%97%E3%81%9F%E3%83%86%E3%83%BC%E3%83%97%E3%83%AC%E3%83%BC%E3%83%88%E3%83%91%E3%82%BF%E3%83%83%BC%E3%83%BC)(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]; }

    // その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。

# Sample Code

C++

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 split(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 (!sprit(curr, n_nuits, next)) {
84             continue;
85         }
86
87         if (prev == nullptr) {
88             header = next;
89         } else {
90             prev->next = next;
91         }
92         break;
93     }
94
95     if (curr != nullptr) {
96         ++curr;
97     }
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
85             return *this;
86         }
87
88         Inner_::header_t const* operator*() noexcept { return header_; }
89         bool operator==(const_iterator const& rhs) noexcept { return header_ == rhs.header_; }
90         bool operator!=(const_iterator const& rhs) noexcept { return !(*this == rhs); }
91
92     private:
93         Inner_::header_t const* header_;
94     };

```

```

95
96     const_iterator begin() const noexcept { return const_iterator{header_}; }
97     const_iterator end() const noexcept { return const_iterator{nullptr}; }
98     const_iterator cbegin() const noexcept { return const_iterator{header_}; }
99     const_iterator cend() const noexcept { return const_iterator{nullptr}; }
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
```