オブジェクト指向と C++

– オブジェクト指向言語としての C++とは –

2014 年 2 月 10 日 初版

板垣 衛

■ 改訂履歴

版	リリース	担当	改訂内容
初版	2014 年 2 月 10 日	板垣 衛	（初版）

■ 目次

■ 概略

他のオブジェクト指向言語と C++との用語の相違を解説。

■ 目的

本書は、C++プログラマーに向けて、他のオブジェクト指向言語や一般的なオブジェクト指向用語によって解説されている技術文書を理解できるようにすることを目的として、ごく初歩的な知識を提供する。

■ オブジェクト指向用語と C++の用語

後発の Java や C#、Ruby などの言語は、基本的にオブジェクト指向言語として一般的な用語を用いている。

C++の用語がとりわけ異質であることを前置きし、一般的なオブジェクト指向用語と C++の用語との対応を示す。

なお、「クラス」などの、C++でも普通に用いられている一般的なオブジェクト指向用語は扱わない。

▼ C++：メンバー関数

対応する一般的なオブジェクト指向用語：**メソッド**

「メソッド」は、C++でも普通に使われている。C++では本来「メンバー関数」と呼ぶのが正しい。

両者の用語は全く同じ意味という理解で問題ない。

▼ C++：メンバー変数

対応する一般的なオブジェクト指向用語：**フィールド**

これも C++でも普通に使われる用語。

▼ C++：アクセッサ

対応する一般的なオブジェクト指向用語：**プロパティ**

「プロパティ」は、一般的と言うより、VB や.Net Framework でサポートされている機構。多くの場合、フィールドへのアクセス方法を提供する。

主な利用目的は、フィールドへのアクセスを Read/Write、ReadOnly、非公開に選別すること。

C++ や Javba の場合は「x = o.getMember()」「o.setMember(x)」といった getter, setter メソッドを用いるのが一般的だが、「プロパティ」は一見してフィールドへの直接アクセスと同じに見える「o = x.member」「o.member = x」という形でアクセスする。

なお、C++でも operator() を利用した「プロパティクラス」を作れば、プロパティと全く同じアクセス方法を提供することが可能。

▼ C++：仮想クラス／virtual

対応する一般的なオブジェクト指向用語：**抽象クラス／abstract**

クラスのメンバーに virtual 関数（抽象メソッド）を含んだクラスのこと。
C++で言うところの「仮想化」は、一般的には「抽象化」と言う。

一般的に、抽象クラスは必ず「継承」してメソッドを「具象化」しないとインスタンス化できないが、C++は可能。

C++の virtual 関数は厳密には抽象化ではないため、virtual 宣言しつつ関数の中身を書くことができ、インスタンス化も問題ない。これがは「オーバーライドが許可されたメンバー関数」という意味で扱われる。また、「virtual int memberFunc() = 0;」のように、「= 0」を付けて宣言すると抽象関数となり、継承が必須となる。

なお、C#の場合は、「abstract」と「virtual」の両方の宣言があり、前者が「抽象化」、後者が「オーバーライド許可」を表す。更に、virtual 関数をオーバーライドする際は、override 宣言も必要となり、言語の設計として非常に安全性が高い。

C++の場合は、意図通りのオーバーライドがされていなくてもコンパイルエラーにならないという問題がある。なお、C++11 仕様から override 宣言が追加されている。

▼ C++：純粋仮想クラス／virtual

対応する一般的なオブジェクト指向用語：**インターフェースクラス／interface**

フィールドがなく、100%抽象メンバーのみで構成されたクラスのこと。

C++では 100%の抽象クラスは意外と使いにくく、わずかに仮想関数の実体を含めることが多いのではないかと思う。

● 多態性（ポリモーフィズム）

「インターフェース」は、オブジェクト指向プログラミングを支える「最も重要な要素」といっても過言ではない。

共通インターフェースに対して様々な振る舞いが実装可能な「性質」のことを「多態性」（ポリモーフィズム）と呼ぶ。この用語はオブジェクト指向の技術処理に頻出する。

C++では、具体的に下記のようなプログラミングで「多態性」を実現できる。

例：

```
//インターフェースクラス
class ISample
{
public:
	virtual const char* getMessage() const = 0;
};
//具象クラス①
class CTest1 : public ISample
{
public:
	virtual const char* getMessage() const { return "This is CTest1!" ; }
};
//具象クラス②
class CTest2 : public ISample
{
public:
	virtual const char* getMessage() const { return "This is CTest2!!!" ; }
};
//共通処理
void testCommon(ISample& obj)
{
	printf( "[ %s ]¥n" , obj.getMessage());	//←ここで「多態性」が反映されている
}
//テスト
void testMain()
{
	CTest1 o1;
	CTest2 o2;
	testCommon(o1);
	testCommon(o2);
}
```

↓（実行結果）

```
[This is CTest1!]
[This is CTest2!!!]
```

● インターフェースのインプリメントと多重継承

Java や VB、C#などは、クラスの単一継承しか許可されていない。しかし、クラスの継承とは別に、複数のインターフェースを実装することができる。インターフェースを実装することを「インプリメント」と呼ぶ。

C++は継承とインターフェースのインプリメントを区別しないので、多重継承でそれを実現する。

▼ C++：テンプレート／template

対応する一般的なオブジェクト指向用語：ジェネリック／generic

ジェネリックプログラミングは、「汎用プログラミング」とも呼ばれる。

プログラムの一般的な構造を用意して、コンパイル時に型や定数を与えてプログラムを実体化する手法のことである。

テンプレートには、関数をテンプレート化する「テンプレート関数」と、クラス全体をテンプレート化する「テンプレートクラス」がある。クラスのメンバー関数を「テンプレート関数」にすることも可能。

● メタプログラミング

コンパイル時にプログラムが実体化される際、C++の場合、可能な限り事前計算して最適化が行われ、時には定数化までされてしまう。

このように、コンパイル時にプログラムを実行させる手法を、「メタプログラミング」と呼ぶ。

【注意】「ジェネリック」と「メタプログラミング」は本来別物なので混同しないように。「ジェネリック」は、プログラミングの時点で「型」を特定しない「汎用＝ジェネリック」のプログラミングのことである。C++言語における「メタプログラミング」とは、C++言語が「コンパイル時に」ジェネリックなプログラムを実体化する性質を最大限に利用する、いわば副次的な効果を狙った手法である。「メタプログラミング」の本来の意味としては、大まかには「プログラムがプログラムを生成すること」を指す。

例：メタプロミングのサンプル

```
//テンプレート関数
template<class T>
inline T max(T a, T b){ return a > b ? a : b;}
//テスト
void testMain()
{
	int a = max(10, 20);
}
```

↓（コンパイル結果）

```
void testMain()
{
	int a = 20; //メタプログラミングにより、関数が消滅して定数化されている
}
```

「メタプログラミング」では、if 文やループ文は実行されず、単純な計算しか行われないが、「特殊化」というテンプレートクラス（テンプレート関数は不可）の仕組みを利用することで、実質的に if 文やループ文に相当する処理をコンパイル時にさせることも可能。

▼ 進化し続ける C++

C++は今もなお仕様拡張が進んでいる。ゲームプログラミングの現場でも、新しい仕様に目を向け、最適化や安全性の向上を目指し続けるべきである。

以下、現在主流の C++の仕様を簡単に列挙する。

- **C++03**

  現在最も普及している C++の仕様は、「C++03」という仕様である。STL などのライブラリも標準化されている。

- **Boost C++**

  「C++03」から漏れながら、「自機標準化候補」として開発されていたライブラリが「Boost C++」である。かなり巨大なライブラリで、STL のようなテンプレートライブラリが多い。C++11 が世に出始めた現在も、Boost C++ が統合されることも無く、独自に進化を続けている。

- **C++11（旧称：C++0x）**

  「C++03」の次の標準仕様が既にリリースされている。「C+11」である。これは、旧称「C++0X」と呼ばれていたものである。ただし、「C++11」の仕様はまだまだ普及しておらず、最新の「Visual C++ 2013」も部分的な実装しかできていない。

- **C++14**

  「C++11」の実装がままならない現在、早くも次期仕様の策定が進んでいる。それが「C++14」である。「C++14」は「C++11」の問題点を補う点や、「C++11」に間に合わなかったものの追加要素的なイメージが強く、今後は両者の仕様が混ざった形で実装されていくと思われる。

● C++11 について

C++11 には、注目すべき追加仕様が多数存在する。細かい説明は省き、ゲームプログラミングの処理最適化、安全性向上のために、是非とも活用したい仕様を以下に列挙する。

- 右辺値参照 ..... 無駄な値渡しを減らし、スマートポインタ処理の最適化などに活用。
- constexpr ..... 強力なメタプログラミング。(VC++2013 未だ未対応)
- ユーザー定義リテラル ..... 「1.2f」の「f」のようなリテラル値のサフィックスを独自実装。constexpr と組み合わせてメタプログラミングを強化。
- 外部テンプレート ..... テンプレートの extern。コンパイル最適化。
- 型推論型 ..... auto 型。濫用は危険？テストコードを書くのが少し楽に。
- 範囲に基づく for ループ ... 固定配列の処理ループなどが簡潔に。
- ラムダ式 ..... 関数内のちょっとした共通処理や、ソートなどのアルゴリズムに渡す条件判定式などを、わざわざ共通関数にせずに対応可能。
- 明示的なオーバーライド .... C#と同様の override キーワードの実装。書き方は C#と異なる。
- nullptr ..... 数字の 0（NULL） ではない、ポインター用の null。区別できるようになった。
- 強い片付けの列挙型 ..... int 型以外の enum。
- 別名テンプレート ..... テンプレートクラス／関数の typedef。uing キーワードを用いる。
- 可変長引数テンプレート .... 関数型プログラミング言語のようなメタプログラミングがより可能に。
- 文字コード指定の文字列リテラル（unicode サポート強化）
- コピーコンストラクタなどの delete 指定。
- ハッシュテーブルライブラリ
- 正規表現ライブラリ
- スマートポインタ
- 乱数エンジン

■■以上■■

■ 索引

し

た

ち

て

は

ふ

ほ

め

オブジェクト指向と C++

以　上