第11章

C++の概要と仮想関数について

C++の使い方（前編）

本章と次章では，C++がどんな言語であるか，どのような使い方がなされるかを，具体的なコードを示しながらていねいに解説する．前編たる本章では，C++の概要と，C言語との大きな違いである仮想関数についてくわしく説明する．（編集部）

11.1 C++について

この章は，C++の文法について書いたものではありません．この章は，C++の文法がわかって，ある程度プログラムが書けるという人を対象にしています．そのため，この章を読んだだけでは，決してC++でプログラムが書けるようになるとわけではありません．

ここでは，プログラムを作るときにどのようにC++の言語機能を使うのかということについて解説するわけですが，その前にC++という言語について説明しておきたいと思います．

● そもそもどういう経緯で？

C++といえばオブジェクト指向言語であり，「C言語そっくり」と評する人がいます．その一方，Cとはまるで違う言語だという人もいます．さて，これはいったいどういうことでしょうか．

C++は，Bjarne Stroustrup[注1]という人が1985年に開発した言語であり，その経緯を昔話風にごく乱暴に言い切ってしまうと，次のような感じです[注2]．

この人は，仕事で複雑なプログラムを開発しなければなりませんでした．ちなみに，それはネットワークに散らばったプログラムの管理用だったそうです．複雑な問題を含むプログラムですが，C言語が得意とするような低レベルの操作が必要です．彼が「エレガントだ」と思う言語はSimulaというシミュレーション用の言語[注3]です．

複雑な問題をシミュレーションで解くプログラムを作るのは得意です．だから，この複雑な問題をシミュレーションで解きたいわけです．ということで，C言語でシミュレーション・プログラムを作ろうとしたのですが……．

「やってられない！」ということになってしまったのです．C言語は，シミュレーションの要素を素直に表現するための文法をもっていません．もともとC言語は，OSを記述するために作られた言語です．シミュレーション要素を表現するような文法をもっているはずがありません[注4]．Stroustrupさんは考えました．C言語にシミュレーション要素を素直に表すための文法を足してやれば良いのではないかと．

文法を足した後，実際に自分の仕事に役立てるには，それを処理するものがいります．が，コンパイラを自作するのはめんどうです．なので，そのちょこっと足した部分だけC言語に変換するプログラムを作ることにしました．C言語のままで良い部分はそのままで，C言語ではない部分だけC言語に変換するというプログラムなら，コンパイラを一から作るより簡単です．

それがcfrontといわれるプログラムでした．これを通すと，C言語にシミュレーション用の文法を足した新し

注1：ビアルネ・ストラストラップと読むらしい．

注2：この段落はとても乱暴な説明になっている．説明のためのお話だと思ってください．

注3：当時からSimulaはシミュレーション用言語ではなく，汎用オブジェクト指向言語であったとのこと．しかし，生い立ちというのは怖いもので……．

注4：これはかなり乱暴な言い切りである．OSを記述するために作られた言語が，シミュレーション要素を表現するような文法をもっていても不思議はない．しかし，必然もないので，このような言い切りになっている．

い言語をC言語に変換することができます．これで，シミュレーション・プログラムが楽に開発できて，めでたしめでたしとなりました．

このとき作られた，C言語にちょっとだけシミュレーション用のものを追加した言語である「C with Classes」が，C++の前身です．シミュレーション分野におけるシミュレーションの要素は，オブジェクトといいます．シミュレーションを行う際には，同じようなオブジェクトがたくさん必要になります．たとえば，雑踏での人の流れをシミュレーションするなら，人というオブジェクトがたくさん必要です．そのため，オブジェクトを一つの単位としてプログラムを動かしたいのですが，オブジェクトを単位としたプログラムを作るのはたいへんです．

同じようなオブジェクトのためのプログラムは，できたら一つで済ませたいものです．そして同じようなオブジェクトであるということは，同じ種類のオブジェクトであるということです．このオブジェクトの種類をクラスと言います．

C言語の文法を借りてきて，クラスをプログラミングし，オブジェクトを動かす「C言語にクラスを追加した言語」というのが，「C with Classes」です．これがC++の生い立ちになります．

OS記述用言語のC[注5]と同じような文法や表記法でありながら，シミュレーション用プログラム言語であるC++[注6]があります．プログラムの基本単位が「関数であり機能」であるのがC言語，プログラムの基本単位が「オブジェクト，つまりシミュレーションの要素」であるのがC++です．そもそものプログラムの分け方が違っているのです．

C++でプログラムを記述するなら，問題をいったんシミュレーションの形に直し，シミュレーションの要素に分解する必要があります．C言語が問題を機能に直し，機能を分解するのとは違うのです．

● しくみは？

C++には，オブジェクト指向周りの言葉がたくさん出てきます．しかし，もともとはC言語のフロント・エンド・プロセッサです．つもりはどうあれ，C言語の構造を借りてきたものにすぎません(「たいしたことはない」といっているのではない．すごいことを簡単に作ってしまったという，「たいしたこと」なのである)．ということで，C++のしくみを理解するために，オブジェクト指向周りの言葉を説明しましょう．たとえ話に使うのはC言語です．

▶クラス&オブジェクト

クラスというのは，構造体のようなものです．オブジェクトは構造体変数です．プログラムの作り方として想定しているのは，構造体を中心としたものです．構造体を作ったら，その構造体を扱う関数を作ります．一つのファイルには，一つの構造体を扱う関数だけを収めます．

C言語を例に出すと，入出力用標準ライブラリでFILE*型あたりが近いでしょうか．fopen，fread，fwrite，fclose，fprintf，fscanf，……．どれもFILE*をキーワードにした関数です．これらを一つのファイルに収めてしまい，構造体宣言とこれらの関数のプロトタイプ宣言を一つのヘッダで行うようにします．

プロトタイプ宣言や構造体宣言がクラスです．この構造体の変数がオブジェクトになります．構造体を扱う関数をC++用語ではメンバ関数，オブジェクト指向用語ではメソッドと呼びます．

▶隠ぺい

構造体のメンバ変数にアクセスするのは，ただ一つのファイルであるべし，というプログラム上のヒントがあります[注7]．もともとその構造体を扱う関数は，専用のファイルに入れたはずです(前項参照)．しかし，メンバ変

注5：C言語の生い立ちはOS記述用だが，別にOS記述用に専用化された言語というわけではない．しかし，OS記述用として生まれたC言語は，OSを記述するのに必要である危険な操作が簡単にできるようになっている．そこが「高級アセンブラ」ともいわれるゆえんである．「三つ子の魂百まで」とはよく言ったものだ．

注6：実際にはC++は別にシミュレーション用言語ではないのだが，これも「三つ子の魂百まで」かもしれない．初期のオブジェクト指向は「現実世界をそのままにプログラミングする」だったため，これは結局シミュレーション・プログラムだった．なので，C++にかぎらずオブジェクト指向言語でプログラムを行う際には，シミュレーション・プログラムを作るつもりであれば，かなりうまくいく．

注7：上級者であれば，メンバ変数に複数のファイルからアクセスしているのすら隠ぺいしているということがあり得る．しかし，上級者であればそんなことはしないだろう(継承のことを言っているわけではない)．

数にアクセスするようなコードがほかにもあるなら，それもそれ専用のファイルに入れるべきです．よくある説明に「get/set関数を作ること」というのがありますが，これでは本末転倒です．メンバ変数にアクセスしているだけという関数があっていけないわけではないのですが，メンバ変数にアクセスするのが目的という関数であっては困ります．

C言語のFILE*関数群の一つfeof関数は，実際にはFILE構造体のeofメンバ変数を読み出しているだけです(そういう実装もありえると言うだけで，すべての実装でeofメンバ変数を読み出しているわけではない)．しかし，FILE構造体のeofメンバ変数を読み出すためにfeof関数を呼び出す人はいません．あくまで，ファイルを読み出すとき，最後まで読んだかどうかを調べるためにfeof関数を呼び出すのです．

▶継承

新しく構造体を作るとき，その第1メンバ変数に別の構造体をもってきます．

```
struct Base {
  int member1;
  char* member2;
};

struct NewStruct {
  struct Base base;
  int newMember1;
};
```

新しく作った構造体へのポインタと，第1メンバ変数の構造体のポインタは同じ値になるはずです．そのため，第1メンバ変数の構造体を扱う関数は，そのまま新しく作った構造体を扱える(当然第1メンバ変数の部分しか扱えないが)はずです．

```
void baseMethod(struct Base* self);

struct Base base;
baseMethod(&base);

struct NewStruct newStruct;
baseMethod((struct Base*)&newStruct); /* C言語の場合，キャストが必要になる */
```

だから，新しい構造体を扱う関数としては，新しいメンバ変数を扱わなければならないものだけを書けば良いのです．

```
/* baseMethodに相当するものは作らなくて良い */
void newStructMethod(struct NewStruct* self);
                                    /* newMember1を扱わなければならないものだけ書けば良い */
```

これを**差分継承**と呼びます．

C言語では，新しく作った構造体と第1メンバ変数の構造体では型が違うため，キャストが必要になります．しかし，C++では第1メンバ変数と書かず，代わりに「継承の親である」と書けばキャストが不要です．

```
// C++の場合
class Base {
private:
  int member1;
  char* member2;
public:
  void baseMethod();  // Base* は暗黙の引き数として渡される */
```

```
};

class NewStruct : public Base {
private:
  int newMember1;
public:
  void newStructMethod();
};

Base base;
base.baseMethod();      // &baseを第1引き数として渡したのとほぼ同じ

NewStruct newStruct;
newStruct.baseMethod(); // &newStructを第1引き数として渡したのとほぼ同じで，さらにキャスト不要
newStruct.newMethod();
```

▶仮想関数

構造体を扱う関数のうち，関数ポインタ・テーブルを使って呼び出されるものを指します．この関数ポインタ・テーブルは，対象となる構造体の中に自動的に作られます[注8]．

```
//C++での記述
class Foo {
private:
  int member1;
public:
  virtual void method1(); //仮想関数
};

void Foo::method1(){
  //何らかの処理
}

Foo foo;
Foo* pFoo=&foo;

pFoo->method1();
```

これをC言語で書くと，このようになります．

```
/* C言語での記述 */
strcut Foo {
  int member1;
  void (*method1)(struct Foo* self); /*関数ポインタ宣言 */
};

void method1(struct Foo* self){
```

注8：実際には，関数ポインタ・テーブルへのポインタが構造体の中に作られる．

```
    /* 何らかの処理 */
}

struct Foo foo;
foo.method1=method1;  /* C言語では自分で設定する必要がある */

strict Foo* pFoo;
pFoo=&foo;

pFoo->method1(pFoo);
```

構造体の中の関数ポインタ・テーブルの部分は，コンパイラが自動的に初期化します(コードを埋め込む)．引き数の構造体に入っている関数ポインタ・テーブルを使って呼び出すため，引き数の構造体が違えば別の関数が呼び出されます(かもしれない)．つまり，引き数の構造体を扱うのに最適な関数が，あたかも自動的に選択されたように見えるというわけです．

▶メッセージ・パッシング

仮想関数は，結局のところ関数ポインタ・テーブルを使って呼び出される関数ですが，いつもこのように考えるのはめんどうな話です．そのため，OSがあるように考えて，「オブジェクトどうしはメッセージを送っている」と考えることにしようということです[注9]．

同じコードでも，関数ポインタ・テーブルが差し換えられれば，別の関数を呼び出してしまいます．これは「同じメッセージを送っても受け取るプロセス/タスク/Windowが違えば別の関数が動き出す」のになぞらえているわけです．

ところで，こういうしくみを使って何を行うかなのですが，これは概念的な話になります．

オブジェクトはモデルを表現するために
クラスはオブジェクトの種類を表現するために
継承はis_a関係を表現するために
……

しかし，それらはここで説明したようなしくみで実現していることばかりです．逆に，このようなしくみは以前から自分で作っていませんでしたか？ そうだとすると，そのしくみは何のために作ったものだったのでしょう？ 結局それがオブジェクト指向云々という話なのでは？

● プログラム単位の大きさ

C++では，プログラムをクラスに分割してコーディングしていきます．モニタやOSを使ってC言語でコーディングする場合は，タスクに分割すると思いますが，このタスクとクラスは基本的にサイズが異なっています[注10]．タスクの大きさはそれぞれの箇所で違うとは思いますが，小さくても大体数千行ほどではないでしょうか．C++におけるクラスの大きさは，大きくても数百行ほどです．ざっと関数数個分だと思ってください．

つまり，クラスのほうが圧倒的に小さいのです．タスクは一つ作るごとに資源やオーバヘッドが必要で，あまり小さいタスクを作るのは効率が悪いということもあります．一方，クラスでは必要な資源やオーバヘッドが圧倒的に少ないのです．このようなことから，分割の単位としてはクラスのほうが小さくなります．プロジェクトなどの開発において一つのタスクに一人担当者を割り当てるということが多いと思いますが，このような割り当ては乱暴です．そうなると担当者一人当たり，クラスを10個以上作り出すことになるはずだからです．

ところで，現在C言語を使う開発で，あらかじめすべての関数を設計しておくことは少ないと思います．機能ブロックのインターフェースとなる関数のプロトタイプぐらいまでを設計し，あとはコーディングに任せるのが大

注9：実際は話が逆で，メッセージを送っているのと同じ効果を出すために，C++では仮想関数をいうしくみを作り出した．
注10：もちろん，タスクやクラスは大きさで分割するものではない．

部分だと思います．そして，コーディング時点でかなりの数の関数が作られることでしょう．

クラスも似たようなものだと思ってください．あらかじめすべてのクラスを設計することはありません．おおまかな機能ブロックに相当するクラスまで設計し，あとはコーディングに任せます．そして，コーディング時点でかなりの数のクラスが作られるのです．

あらかじめすべてのクラスを設計しようとします．設計できたクラスしか存在できないため，一つのクラスがものすごく大きくなります．大きなクラスというのは，大きな関数と同じようにわかりにくい代物です．だからこそ，クラスを小さくする必要があるのです．

また，仮想関数をはじめとするC++の特有の言語機能は，主にクラスが単位となります．デザインパターンなどもそうです．それぞれのパターンはクラスが単位となっています．ちまたで噂のCPPUnitは，クラスを「単体テスト」するためのものです．クラスが単位なのです．これらの言語機能やデザインパターン，CPPUnitは，ちょっとしたコーディングをするのに役立ちます．ちょっとしたコーディングをするために，クラスを単位としたこれらのしくみを活用するということは，かなり小さいクラスがたくさん作られるということになります．

クラスは，それが含んでいるメンバ関数より小さくはなれません．クラスがある程度以上小さくなると，必然的にメンバ関数はもっと小さくなります．だから1行しかないメンバ関数というものも，けっこう多いものです．

11.2 仮想関数を使う

C++は，「一応ながら」オブジェクト指向プログラミング言語ということになっています[注11]．C言語と何が違うかといえば，まずクラスの存在でしょう．クラスに関連した，C言語が真似しにくい機能として「仮想関数」と「デストラクタ」があります．どちらもC言語レベルでは「プログラマがコーディングしなかった」ものをいわば埋め込む機能です．仮想関数はコーディングしなかった関数ポインタ・テーブルを，デストラクタはコーディングしなかった関数呼び出しを，それぞれ埋め込みます．

この章では，仮想関数を使ったC++らしいプログラムについて解説します．

● 仮想関数って何物？

まずは，仮想関数が何を実現しようとして生まれてきたのか，そしてそれをどのように実現しているのかというあたりから解説しましょう．

▶多態，メッセージ・パッシング

オブジェクト指向について勉強していると，三つのキーワードが出てきます．

1. 隠ぺい(カプセル化)
2. 継承(派生)
3. 多態(多相，ポリモーフィズム)

この三つのキーワードのうち，多態がもっとも難解だと言われています．しかし多態自身は，それほど難しい概念ではありません．つまり，こう考えてくださいWindowsのアプリケーションにはいろいろなボタンがあります．それぞれのボタンにはそれぞれの役割があり，別のボタンを押せば別のことが行われます．それぞれのボタンごとに別の処理がくっついていて，「押す/クリックする」と，ボタンごとにくっついているそれぞれの処理が動き出すのです．

多態とは，同じ「押す/クリックする」という呼び出し方で，ボタンごとに違う処理が行われる，単にそれだけのことです．「多態がどのように実現されているのか」という簡単なメカニズムが，メッセージ・パッシングです．「処理を呼び出すのではなく，メッセージを投げるのだ．メッセージを解釈してどのような処理を行うのかは，メッセージを受信した側の問題だ．」――これがメッセージ・パッシングと呼ばれる考え方です．

Windowsのプログラムやモニタ，OSを使ったプログラムを書いたことがある方なら，この考え方は別に不思

注11：C++を純粋なオブジェクト指向言語とはいいにくい．ほかのパラダイムもいろいろサポートしているので．オブジェクト指向プログラミングもサポートしている言語というぐらいだろうか．

議な考え方でもなんでもないでしょう.

たとえば，モニタやOSを使ったプログラムでは，多くの場合に次のようになります.

```
TaskA{
  ...
  SendMessage(TaskB,MESSAGE_1);
  ...
}

TaskB{
  for(;;){
    id=ReceiveMessage();
    switch(id){
    case MESSAGE_1:
      TaskB_MESSAGE1();
      break;
    case MESSAGE_2:
      TaskB_MESSAGE2();
      break;
    ...
    }
  }
}
```

TaskAが，TaskBにMESSAGE_1というメッセージを送っています．その結果，TaskB_MESSAGE1()という処理が動きます.

もしTaskAが，TaskCにMESSAGE_1というメッセージを送ると，どういう結果になるでしょうか？ まさかTaskB_MESSAGE1()が動くとは思わないでしょう．TaskCがどのようなコードになっているかはわかりませんが，おそらくTaskC_MESSAGE1()という処理があって，それが動くのではないでしょうか.

「同じメッセージを送っても，別のタスクが受け取るのであれば，別の処理が起動されるだろう.」――これがメッセージ・パッシングであり，多態です.

例はモニタを使ったプログラムで，タスクをプログラムの分割単位としました．しかしタスクやスレッドに話を限定する必要はありません．たとえばWindowsプログラミングにおいて，OSが想定するプログラムの分割単位はWindowです．このため，メッセージを受け取るのはWindowということになります.「同じメッセージを送っても，別のWindowが受け取るのであれば，別の処理が起動される」はずです．だからこそ，同じように押しても，別のボタンを押したら，別の処理が実行されます.

クラス・ライブラリなどで隠されていなければ，Windowsのプログラムも，先のタスクの例と同じようになります．ましてや，Windowに届けられるメッセージの種類はたくさんあるのですから，巨大なswitch文がそこに存在するはずです．そうです．昔のWindowsのアプリケーションには，その中心に巨大なswitch文があったのです.

▶ C++では？

C++では，プログラムをクラスやオブジェクトに分割します．そして，このオブジェクトどうしでメッセージ・パッシングを行うと考えます．つまり，オブジェクトは，メソッドを呼び合うのではなく，メッセージを投げ合い，メッセージを受け取るとメソッドが起動されると考えるのです.

しかし，毎度毎度メッセージを受け取るごとにどのメソッドを起動すれば良いのかと「プログラマ」が考えるのはめんどうです．通常のプログラムにはたくさんのメッセージ，クラスが存在します．そのたくさんのメッセー