第6章

多態や隠ぺいの実装を解説

オブジェクトやクラスをC言語で表現する

本章では，オブジェクト指向のさまざまな概念をC言語プログラムにどう落としていくかについて解説する．まず，オブジェクトやクラスという表現をもっていないC言語でオブジェクトとやクラスを表す方法を説明する．さらに，隠ぺい，多態を実装する方法を，サンプル・コードを紹介しながら解説する．（編集部）

「オブジェクト指向をやるのは，オブジェクト指向言語でなければならない」と思われているかもしれませんが，そんなことはありません．オブジェクト指向でない言語でオブジェクト指向をするのは，できないのではなく，めんどうなだけです．

クラスを表すにはどうしたら良いの？　継承を表すにはどうするの？　そのようなことが，オブジェクト指向言語では言語仕様で決まっていて，便利なようになっているだけで，それでなければできないということではありません．

たとえば，アセンブリ言語には引き数や返り値という概念がないはずです．また，サブルーチンは多くのアセンブリ言語にあるでしょうが，関数はありません．自動変数もありません．

それでも，アセンブリ言語で関数は作れるし，引き数も返り値も作れます．レジスタを引き数だと定めて使ったり，サブルーチンを関数として使ったり，保護したレジスタで自動変数を作ることすらあります．

C言語には関数や変数があります．したがって，C言語のほうが関数を作ったり，引き数や返り値を作ったりするのが楽なのです．しかし，アセンブリ言語でもできるのです．

同じことがオブジェクト指向にもいえます．結局，使ってしまえば，C言語だろうとアセンブリ言語だろうとオブジェクト指向ができるわけです．C++やJavaよりめんどうですが[注1]．

ということで，C言語でどうやってオブジェクト指向の概念をプログラムに落としていくか？ということを紹介していきます．サブタイトルには「実装」という文字がありますが，ここで紹介するものは，じつはメカニズム設計で考えるべきことです．

6.1 オブジェクトやクラスという表現

オブジェクトやクラスという表現をもっていないC言語で，オブジェクトやクラスを表すことはできません．あの関数とこの関数は，そのオブジェクトまたはクラスのメソッドであるということを，C言語で直接表現する方法がないからです．なければ何か別の方法を考えなければなりません．

しかし，これは簡単です．関数などにそのような名前を付けてしまえばよいのです．たとえば，オブジェクトFooのメソッドには，最後に必ず_Fooという名前を付けるなどというやり方です．

```
void init_Foo();
int  getCount_Foo();
void term_Foo();
```

このような名前を付けてしまえば，呼び出す側も呼ばれる側もこれらがオブジェクトFooのメソッドで，これを呼び出せばオブジェクトFooを操作できると考えることができます．

注1：C++やJavaを覚えるのがめんどうだからという理由でC言語でオブジェクト指向をやろうとしているなら，それはやめたほうがいい．

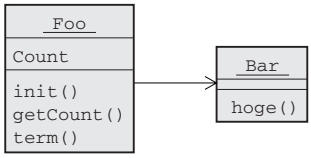

図6.1
FooがBarを使う
オブジェクト図

リスト6.1　FooがBarを使う

```
#if !defined FOO_H_ /* ダブルインクルード防止 */
#define FOO_H_

/* オブジェクトFooの公開操作 */
void init_Foo(void);
int getCount_Foo(void);
void term_Foo(void);

#endif
```

（**a**）ヘッダ（foo.h）

```
#if !defined BAR_H_
#define BAR_H_

/* オブジェクトBarの公開操作 */
void hoge_Bar(void);

#endif
```

（**b**）ヘッダ（bar.h）

```
#include "foo.h"
#include "bar.h"  /* 関連するオブジェクトBarのヘッダをインクルードする */

static int count;

void init_Foo(void){
  count=0;
  hoge_Bar();   /* Barのhoge操作を呼び出す */
}

... 以下略
```

（**c**）ソース本体（foo.c）

リスト6.2　隠ぺい

```
#if !defined FOO_H_ /* ダブルインクルード防止 */
#define FOO_H_

/** オブジェクトFooの使い方など記述 **/

/* オブジェクトFooのインターフェース */
void init_Foo(void);
int getCount_Foo(void);

#endif /* FOO_H_ */
```

（**a**）ヘッダ（foo.h）

```
/** オブジェクトFooの実装について記述 **/

#include "foo.h"

/** 属性 隠蔽されているし,staticなのでクラス名を付けなくてもOk **/
static int count;

/** プライベートな(公開していない)メソッド **/
/* 隠蔽されているがメソッドはオブジェクト名を付けた方がわかりやすい */
static void algo_Foo(void);


/* 初期化メソッド */
void init_Foo(void){
  count=0;
}

/* 通常のメソッド */
int getCount_Foo(void){
  return count;
}

/* プライベートなメソッド */
static void algo_Foo(void){
  ...
}
```

（**b**）ソース本体（foo.c）

また，これを実装する場合には，オブジェクトFooのメソッド（_Fooの付いた関数）は一つのソース・ファイルに書きます．すると，オブジェクトFooの実装はそのファイルで行っているということで，わかりやすくなります．オブジェクトFooのインターフェース部分（public部分）であるプロトタイプも，一つのヘッダで表現します．このヘッダをfoo.h，ソースファイル本体をfoo.cとして，オブジェクトFooを使ったり操作する場合には，foo.hをインクルードすればよいわけです．foo.c自身もfoo.hをインクルードします．

たとえば，**図6.1**に示すようなオブジェクト図を考えます．

これに対し，**リスト6.1**のようなコーディングを行います．

1）一つのオブジェクトは一つのヘッダと一つのソース・ファイルで表現する

2）オブジェクトのメソッドはオブジェクト名を含んだ名前を付ける

これだけで，オブジェクト指向の第1歩目をC言語で踏み出すことができます．

6.2 隠ぺいするには

オブジェクト指向では，隠すべき「どうやって？」と，公開すべき「何ができるの？」を切り分けて考えます．これが隠ぺいの原則です．隠ぺいするのは属性，公開するのは操作です．

これをC言語ではどう表すかということですが，これ自身はすでにおなじみのテクニックである，ヘッダとソース本体がその関係になります．

公開すべきものはヘッダに書き，隠すべきものはソース本体に書くわけです．公開したくない属性などの変数はソースのほうに書き，公開したい操作（の宣言）はヘッダに書きます．隠しておきたいものはソース側に書くだ

けでなく，さらにstatic変数やstatic関数にしておくと良いでしょう（**リスト6.2**）．

6.3 クラスの表現

前項で示した方法では，オブジェクトに直接メソッドや属性を記述していました．そのため，どのオブジェクトも独特で，同じようなオブジェクトがほかに存在しないならよいのですが，同じようなオブジェクトがたくさんあると，同じようなことをたくさん書かなければならなくなるのでめんどうです．操作もそれぞれのオブジェクト固有のものなので，扱うときも同じように扱うことができません．同じ種類のオブジェクトのためには1度だけその記述を行えばよく，同じ種類のオブジェクトを扱うときは同じように扱えるようにするためにクラスという概念があります．

クラスはオブジェクトを種類ごとに分けたもので，同じクラスに属するオブジェクトは同じ性質をもちます．よって，クラスをプログラムで表してしまえば，オブジェクトは「あのクラスに属する」というだけで，オブジェクトのためのプログラムを省略することができます．

たとえば，同じクラスに属するオブジェクトAとオブジェクトBがあるとします．これらは何が違うから，別々のオブジェクトなのでしょうか？　同じ性質をもっているはずなので，操作もメソッドも同じはずです．同じようなことを覚えなければならないはずなので，属性も同じ…….

いや，属性の型や名前，数は同じでしょうが，属性の値は違います．同じ犬クラスのオブジェクトである「ポチ」と「ジョン」は，「名前」という属性の値が違います．つまり，オブジェクトの違いは属性の値の違いということになります（属性の値が同じなら同じオブジェクトなのか？というと，そうでもない．同じ名前の別の犬というのも考えられる）．

- 属性の型，つまり変数の型が同じで値が違う
- 一つのクラスには複数の属性があるが，その属性の型はばらばら

この二つのことからC言語では，クラスは構造体型，オブジェクトは構造体変数で表現すればよさそうです[注2]．それぞれの属性はクラスを表す構造体型のメンバ変数として表現します．

クラスを構造体型として表現する場合，その構造体名にクラスの名前を付けてしまいましょう．構造体だということもあまり意識したくないですし，またよく使うので打鍵数[注3]を少なくするためにもtypedefを使って型として定めてしまえば楽です．

属性はこれでよいとして，メソッドはどうすればよいでしょうか．メソッドはどのオブジェクトも共通に使えるため，「隠ぺいするには」の項で示した方法と同じ……いや，じつはそれでは困るのです．

「隠ぺいするには」で示した方法では，メソッドはそれぞれのオブジェクトに固有なものでした．だから，メソッドの中で属性を読み書きする場合でも，どのオブジェクトの属性を読み書きするのか？と悩むことはありません．しかし，今度は同じメソッドで，複数のオブジェクトを扱わなければなりません．どのオブジェクトの属性を読み書きすればよいのか？という問題があります．

呼び出す側としても，どのオブジェクトに対する操作なのか？ということを指定する方法が必要です．

では，指定する方法を設けましょう．いろいろな方法があると思いますが，簡単に操作の第1引き数でどのオブジェクトに対する操作なのか？ということを指定してしまえばよいわけです．指定方法は，オブジェクトへのポインタが便利です．操作の第1引き数はそのままメソッドの第1引き数となります．メソッドは，オブジェクトへのポインタを引き数としてもらうことになるので，それを使って属性を読み書きできます．オブジェクトは構造体変数で表現することにしていたので，その構造体変数のアドレスを操作の第1引き数に指定してもらいましょう．

どのオブジェクトに対する操作なのか？ということを表す第1引き数は，JavaやC++風ならthis，Smalltalk

注2：ほかにも方法はあるが，ここでは構造体を使う方法を紹介する．
注3：実際には単語数だろうか．

やDelphi，Ruby風ならself，Visual Basic風にはmeと表現します．CコンパイラはC++コンパイラと兼ねていることがあり，C言語だからといってうかつにC++の予約語を使うと困ることがあります．そのため，selfを使うことをおすすめします（VisualBasicが好きな人はmeでもかまわない）．以降，selfといえば，この引き数のことをさすことにします．

```
typedef struct {
  ...
} Foo;

void init_Foo(Foo* self);
int getCount_Foo(Foo* self);
```

さて，ここでスコープの話をしておきます．操作やメソッド，属性には**オブジェクト・スコープ**と**クラス・スコープ**という二つのスコープがあります．オブジェクト・スコープというのは，いままで説明したような普通の操作やメソッドや属性です．

クラス・スコープの属性や操作，メソッドは「オブジェクトが一つも存在しない状態でも意味がある」ものです．

たとえば，オブジェクトの生成という操作を考えると，「オブジェクトが一つも存在しない状態でも意味がある」ものです．第一，最初の一つを作り出すときには，オブジェクトが一つも存在しないはずです．

逆にオブジェクトの破壊（消滅，解放，破棄）という操作は，「オブジェクトが一つも存在しない状態でも意味がある」わけではありません．「このオブジェクトを」と指定するわけなので，一つもなければ破壊する意味はありません．

ということで，「オブジェクトの生成」という操作は「クラス・スコープ」，「オブジェクトの破壊」という操作は「オブジェクト・スコープ」の操作です．クラス・スコープの操作は，オブジェクトが存在しなくても呼び出せるので，self引き数をもちません．オブジェクト・スコープの操作は，オブジェクトが存在しているはずで，どのオブジェクトに対するものなのかということを明示するために，self引き数をもちます．属性に関していえば，オブジェクトごとに存在するのがオブジェクト・スコープの属性で，オブジェクトが一つも存在しなくても，またはオブジェクトが複数あっても，一つだけ存在するものがクラス・スコープの属性です．

クラスを表す構造体のメンバは，オブジェクト・スコープの属性だけです．クラス・スコープの属性は，クラスを表す構造体のメンバには入れません．では，どうやって表現するのでしょうか？　クラス・スコープの属性は，普通にソース・ファイルの中でstatic変数として表現すればよいのです．そうすればオブジェクトがあろうとなかろうと，たくさんあろうとも，たった一つだけ存在することができます．

どのようにクラスを表す構造体を公開するかという方法は，2通りあります．それぞれの方法には一長一短があり，どちらを使うべきともいえません．クラスごとにその使われ方を考えて選んでください．

● オブジェクトの変数を呼び出し側で用意する場合

オブジェクトの変数を呼び出し側で用意する場合は，残念ながらうまく属性を隠ぺいすることができません．ということは，隠ぺいはプログラマの良識で「そのクラスのメソッドからしかアクセスしない」と決めただけのものにならざるを得ません．

オブジェクトの変数を使う側で用意する（たとえば自動変数としてなど）場合，構造体として完全に呼び出し側から見えなければなりません．ヘッダにメンバとして属性まで正しく定義した構造体を定義します（**リスト6.3**）．

● オブジェクトの変数をクラス側で用意する場合

▶基本パターン

オブジェクトの変数を呼び出し側が用意せずに，クラス側で用意する場合は，構造体の不完全宣言を使って属性を隠ぺいすることができます．

不完全宣言とは，構造体のタグ名だけ宣言してメンバなどを宣言しない方法です．この場合，呼び出し側は，オブジェクトをポインタでしか扱えません．

オブジェクトの変数をどのようにクラス側で用意するかといった方法には，いくつかの手段が考えられますが，

リスト6.3　オブジェクトの変数を呼び出し側で用意する場合

```
#if !defined FOO_H_
#define FOO_H_

/* Fooのメンバは,Fooのメソッドからしかアクセスしないこと!! */
typedef struct {
  int count;  /* 属性 */
  ...
}Foo;

void init_Foo(Foo* self);
void getCount_Foo(Foo* self);
...その他操作...

#endif /* FOO_H_ */
```

（**a**）ヘッダ（foo.h）

```
#include "foo.h"

void init_Foo(Foo* self){
  self->count=0;
  ...
}

...
```

（**b**）ソース本体（foo.c）

```
#include "foo.h"
void bar(){
  Foo foos[2];
  int i,count;

  for(i=0;i<2;++i){
    init_Foo(&foos[i]);
  }
  ...
  for(i=0;i<2;++i){
    count=getCount_Foo(&foos[i]);
                        /* 複数のオブジェクトを同じように操作できる */
    ...
  }
}
```

（**c**）Fooクラスのオブジェクトを使うソース（bar.c）

リスト6.4　基本パターン

```
#if !defined FOO_H_
#define FOO_H_

typedef struct Foo Foo; /* 構造体の不完全宣言.ポインタとしてだけ扱える */

Foo* new_Foo(void);  /* オブジェクトをクラスに要求する */
void delete_Foo(Foo* self); /* オブジェクトを解放する */
int getCount_Foo(Foo* self);

/** Tips:
  delete_Fooに関しては,2重解放防止にこんなトリックも
  delete_Fooを呼び出すと,自動的にそのポインタもNULLに

void delete_Foo__(Foo* self);
#define deleteFoo( self )¥
   do{ delete_Foo__(self); (self)=NULL; }while( FALSE )

**/

#endif /* FOO_H_ */
```

（**a**）ヘッダ（foo.h）

```
#include "foo.h"
void bar(){
  Foo* pFoos[3];
  int i,count;

  for(i=0;i<3;++i){
    pFoos[i]=new_Foo(); /* オブジェクトを要求 */
  }
  ...
  for(i=0;i<3;++i){
    count=getCount_Foo(pFoos[i]); /* オブジェクトを操作する */
    ...
  }
  ...
  for(i=0;i<3;++i){
    delete_Foo(pFoos[i]); /* オブジェクトを解放 */
  }
}
```

（**c**）Fooクラスのオブジェクトを使うソース（bar.c）

```
#include <stdlib.h>
#include "foo.h"

/** 属性の定義 **/
struct Foo {
  int count;
  ...
};

/* 初期化関数: プライベートで十分 */
static void init_Foo(Foo* self);

/** オブジェクトを要求された **/
Foo* new_Foo(void){
  Foo* newObj=(Foo*)malloc(sizeof(Foo));
  if(newObj!=NULL){
    init_Foo(newObj);
  }
  return newObj;
}

/** オブジェクトを解放する */
void delete_Foo(Foo* self){
  free(self);
}

/** プライベートになった初期化関数 */
static void init_Foo(Foo* self){
  self->count=0;
  ...
}

/* 通常のメソッド */
int getCount_Foo(Foo* self){
  return self->count;
}
```

（**b**）ソース本体（foo.c）

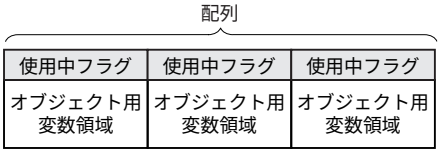

図6.2
mallocが使えない場合にあらかじめ配列で変数領域を確保しておく

ここではmallocを使うことにします(**リスト6.4**)．mallocは，ライブラリやOSが管理するメモリ領域(メモリ・プールまたはヒープ領域と呼ぶ)から指定サイズだけ切り売りしてもらう標準関数です．不要になったら，freeという標準関数を使ってメモリ・プールへ返却しなければなりません．

▶mallocが使えない場合

先ほどの基本パターンでは，オブジェクトの生成にmallocを使いました．しかし，もともとmallocが用意されていない場合や，処理時間の関係でmallocを使うべきではないなど，組み込みではmallocが使えない環境も多いことでしょう．

だからといってあきらめることはありません．たとえば，同時にオブジェクトはいくつまで存在するのか？ということを読めれば，とくにmallocに頼る必要もありません．mallocのような可変長のメモリ管理を使わなくても，同じクラスに属するオブジェクトは，同じサイズのメモリを使用するはずなので，固定長のメモリ管理で十分です．つまり，あらかじめ配列でオブジェクトの使う変数領域を確保しておき(**図6.2**)，要求されるごとにここから変数を切り出していけばよいのです．

ここでは，オブジェクトは同時に二つまで存在する可能性があり，とくに処理時間も気にならず，違うタスク(スレッド)から呼び出されないという条件で考えてみます．この条件は簡単に説明するための方便なので，実際には排他制御を追加したり，処理速度を向上させるために空きブロック・リストを作るべきです．また，エラー・チェックも厳しくすると良いでしょう．

少しトリッキーなコードかもしれませんが，順番に追いかけていけば理解できるかと思います(**リスト6.5**)．

▶初期化できるところがない

先ほどの「クラスを初期化する」メソッド，init_FooClassをプライベートにして隠しておくこともできます(**リスト6.6**)．クラスの初期化を呼び出したくない(適当な初期化できるところがないなど)場合に便利です．クラスを初期化しなければならないのは，最初にオブジェクトを作るときです．それならば，そういう判断とinit_FooClassの呼び出しを「オブジェクトの生成」メソッドに入れてしまえばよいわけです．

● クラスの実装のまとめ

クラスは，構造体とそれを扱う関数群として定義してしまえば良いでしょう．構造体を隠すかどうかは，オブジェクトをどうやって作るかという点で変わります．

mallocを使ったり，自前でメモリ管理を行う場合は構造体を隠すことができるのでそうしましょう．その場合，メモリの解放を忘れないようにしなければなりません．

自動変数やstatic変数として作る場合は，構造体を隠すことができません．構造体を直接さわらないように注意しましょう．

じつに簡単なものです．これで十分という場合も多いので，次の「多態を実装する」前にこれで済まないかどうかを考えてみたほうがよいでしょう．

6.4 多態を実装する

さて，実装するには最大の難関，さらにトリッキなコードが出てくる**多態**です．

多態は次のような場合に便利な代物です．

- 違うクラスのオブジェクトがある．当然それぞれにそれぞれのメソッドがある
- しかし，違うクラスのオブジェクトでも，同じように操作したい
- 同じように操作しても，オブジェクトごとに違うメソッドが呼び出されてほしい

リスト6.5　mallocが使えない場合

```
#if !defined BOOL_H_
#define BOOL_H_

/** TRUEとFALSEの定義 **/
#if !defined FALSE
  #define FALSE (0!=0)
#endif
#if !defined TRUE
  #define TRUE (0==0)
#endif

#endif /* BOOL_H_ */
```

（**a**）共通ヘッダ（bool.h）

```
#if !defined FOO_H_
#define FOO_H_

typedef struct Foo Foo; /* 構造体の不完全定義.ポインタとしてだけ扱える */

/* クラス自体の初期化(メモリ管理部の初期化) */
/* このクラスのオブジェクトを生成する前に1度だけ呼び出しておく必要がある.*/
void init_FooClass(void);

Foo* new_Foo(void);  /* オブジェクトをクラスに要求する */
void delete_Foo(Foo* self); /* オブジェクトを解放する */
int getCount_Foo(Foo* self);

#endif /* FOO_H_ */
```

（**b**）ヘッダ（foo.h）

```
#include <stdio.h>
#include "bool.h"
#include "foo.h"

/** 属性などを定義するところ **/
struct Foo {
  int count;
  ...
};

/* objAreaはMemBlockとアドレスが同じという事がトリックのミソ */
typedef struct {
  Foo objArea;
  int use;    /* 変数使用中フラグ */
} MemBlock;

/* オブジェクトの領域は二つ */
/* ここから変数を切り出していく */
static MemBlock memBlock[2];

/** 配列の要素数を調べる **/
#define elementof( x ) ( sizeof(x) / sizeof((x)[0]) )

/* 初期化関数: プライベートで十分 */
static void init_Foo(Foo* self);

/* クラス自体の初期化(メモリの初期化) */
/* すべての要素を「使用していない」状態にする */
void init_FooClass(void){
  int i;
  for(i=0;i<elementof(memBlock);++i){
    memBlock[i].use=FALSE;
  }
}

/** オブジェクトを要求された **/
Foo* new_Foo(void){
  int i;
  for(i=0;i<elementof(memBlock);++i){
    if(!memBlock[i].use){
      memBlock[i].use=TRUE;           /** 使用中にする **/
      init_Foo(&memBlock[i].objArea); /** 初期化する **/
      return &memBlock[i].objArea;
                                /** オブジェクト用の変数を返す */
    }
  }
  /** 見つからなければNULL, 実際にはエラー処理を行ったほうが良い **/
  return NULL;
}

/** オブジェクトを解放する */
void delete_Foo(Foo* self){
  if(self!=NULL){
    MemBlock* pBlock=(MemBlock*)self;

  /** Tips:
    ここで,2重解放のチェックや,メモリのアドレスが正しいか等
    デバッグ用のチェックを行っても良い.
   **/

    pBlock->use=FALSE; /** 未使用にする **/
  }
}

/** プライベートになった初期化関数 */
static void init_Foo(Foo* self){
  self->count=0;
  ...
}

/* 通常のメソッド */
int getCount_Foo(Foo* self){
  return self->count;
}
```

（**c**）ソース本体（foo.c）

```
#include "foo.h"
void bar(){
  Foo* pFoos[2];
  int i,count;

  for(i=0;i<2;++i){
    pFoos[i]=new_Foo(); /* オブジェクトを要求 */
  }
  ...
  for(i=0;i<2;++i){
    count=getCount_Foo(pFoos[i]); /* オブジェクトを操作する */
    ...
  }
  ...
  for(i=0;i<2;++i){
    delete_Foo(pFoos[i]); /* オブジェクトを解放 */
  }
}
```

（**d**）Fooクラスのオブジェクトを使うソース（bar.c）

```
#include "foo.h"
int main(int ac,char* av[]){
  init_FooClass();
  ...
}
```

（**e**）どこか初期化できるところ

リスト6.6　init_FooClassをプライベートにして隠しておく――ソース本体(foo.c)〔リスト5(c)からトピックになる部分だけ記述〕

```
#include <stdio.h>
#include "bool.h"
#include "foo.h"

....(略)....

/* 初期化済みのフラグがミソ */
static int isClassInitialized=FALSE;
                              /* クラスの初期化を行ったか? */
static MemBlock memBlock[2];

static void init_FooClass(void);
                            /* クラス初期化関数をプライベートに */
static void init_Foo(Foo* self);

/* クラス自体の初期化(メモリ管理部の初期化) */
/* ここのコードはまったく同じ(プライベートになっただけ) */
static void init_FooClass(void){
  ...(略)...
}
```

```
/** オブジェクトを要求された **/
Foo* new_Foo(void){
  int i;

/** クラスが初期化されていないならする **/
  if(!isClassInitialized){
    init_FooClass();
    isClassInitialized=TRUE;  /** クラスは初期化された */
  }

  for(i=0;i<elementof(memBlock);++i){
    ...(略)...
  }
  return NULL;
}

..(略 以下同じ)...

*ヘッダ(foo.h)は**リスト5(a)**と同じ(init_FooClassがプライベートだから)
 Fooクラスのオブジェクトを使うソース(bar.c)は,**リスト5(d)**と同じ
 「どこか初期化できるところ」は不要.init_FooClassは自動的に呼び出される)
```

これができれば，いろいろなクラスのオブジェクトを統一的な操作で扱うことができます．たとえば，プリンタ・クラスのオブジェクトがあったとして，これは自分と接続されるオブジェクトに対し「データをよこせ」と命令を出したいとします．このとき，ファイル・クラスのオブジェクトとイメージ・センサ・クラスのオブジェクトそれぞれに「データをよこせ」という操作があったとしても，多態が使えなければ，クラスが違うのでそれぞれ別の「データをよこせ」操作として扱わなければなりません．特定の操作に対応するのは特定のメソッドになってしまうからです．つまり，getData_File()とgetData_Image()が別の関数なので，同じ関数呼び出しにできないわけです．

ここで多態が使えれば，別々のクラスのオブジェクトであったとしても同じ操作を適用することができます．多態を使って，getData()と呼び出せば，実際につながっているオブジェクトにしたがってgetData_File()とgetData_Image()が呼び出されるのです．

これによって，プリンタ・クラスのオブジェクトはイメージ・センサ・クラスのオブジェクトにも，ファイル・クラスのオブジェクトにも，さらにはネットワーク通信クラスのオブジェクトにも統一的に「データをよこせ」と命令することができ，オブジェクトの組み合わせを自由に選ぶことができるようになります．

さて，この多態をどのようにして実現すればよいのでしょうか？

いちばん簡単なのは，小さなオブジェクトを考えないことです．

たとえば，すべてのオブジェクトはタスクのようなものとしてしまいます．メソッドの呼び出しにはOSの機能を使ってメールやメッセージ，イベント・フラグを用います．それぞれのオブジェクト(タスク)で，自動的に多態を行ってくれるでしょう．操作をメッセージで表現するわけです．呼び出すほうからすると，同じメッセージを送っても，それぞれのオブジェクトがそれにふさわしいふるまい，動きをするわけですから，多態そのものです．

タスクにするには小さすぎるオブジェクトを考えるなら，トリックが必要になります．このトリックとしていろいろと手段を考えることができます．ヒントもあちこちから拾ってこれます．

- C++コンパイラはどのように仮想関数[注4]を実現しているのか?
- 状態遷移ドライバはどのように箱処理を呼び出すのか？
- Windowsプログラミングにおいて，MFC(Microsoft Foundation Class library)で使われるメッセージ・マップはどうなっているのか？
- WindowsプログラミングのCOM(Component Object Model)で，インターフェースとはどのような実現方法か？

これらは，どれも同じような呼び出し方で，別々の動きをします．このあたりをヒントに多態の実現の仕方を考えてみましょう．紹介するのは次の三つです．

注4：多態を行う操作/メソッドのことをC++では仮想関数と呼ぶ．