プログラミング言語論2012 # 4 — オブジェクト指向とジェネリクス

久野　靖*

2012.4.26

1　手続き型から抽象データ型まで

1.1　手続き型からモジュールへ

初回にプログラミング言語が提供する基本的な概念/機能として、次のものを挙げました。

- 式による値の計算、変数による値の保持
- 制御構造による実行順序の制御 (while、if、…)
- 手続き (とパラメタ) による抽象化

これらが確立して定着したのはおよそ 1970 年くらいで、このころ設計された言語である C、Pascal などが「代表的な言語」である時代はかなり長く続きました (1990 年くらいまで)。しかし、計算機の能力増大により、開発されるプログラムの規模も大きくなってきた結果、さまざまな限界が明らかになってきました。(具体的には何?)

まず、手続きによる抽象化はあくまでも「一連の手順 (操作)」をまとめたものであり、「データは別」だったことが挙げられます。つまり、複数の手続きが共通のデータにアクセスする必要がある場合は、次のどちらかを採用する必要があります (図 1)。

- データをそのつどパラメタとして受け渡す
- データをグローバル変数にして、どこからでもアクセス可能にする

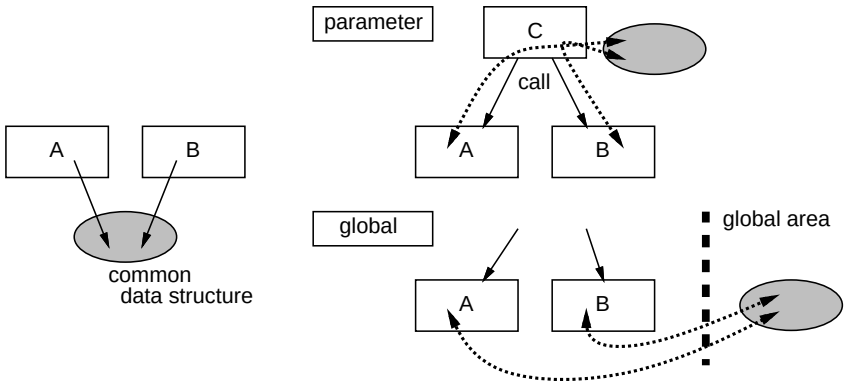

図 1: 共有データの参照方法

*経営システム科学専攻

前者はデータが多くなってくると大変繁雑です。一方、グローバル変数にしてあると、それは「どこからでも」読み書きできるため、どこかでそのデータを「壊して」しまうような操作があった場合、それがどこなのかが大変分かりづらくなります。

このような問題は比較的早期から認識されていて、これを解決するために「モジュール」の考え方が取り入れられるようになりました。この考え方では、複数の手続きとそれらが共通に使用するデータを「モジュール」としてひとまとめにパッケージします。モジュール外からできることは、モジュールに属する手続きのどれかを呼び出すことだけです (図 2)。

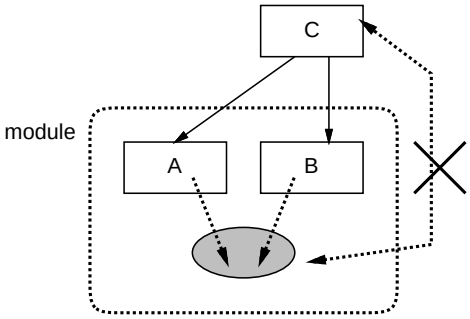

図 2: モジュールの考え方

これにより、このデータは外部からは勝手にアクセスされることがなくなり、もしデータに不整合などが見つかったら、それをアクセスするモジュール内の手続きだけをチェックすれば問題が見つけられるようになります。

モジュール化の考え方が広まった時代には、この機能を取り入れた言語として Modula、Modula-2 などが作られました。しかし実は、C 言語もそのための機構は持たないものの、モジュールを作ることができました。つまり、C ではファイル単位でコンパイルを行いますが、ファイル内で関数の外側にあたる位置に定義した変数は、通常ならグローバル変数になりますが、`static` キーワードをつけるとその変数が「ファイル内だけで使える」変数になります。

ということは、モジュールをファイルに対応させ、モジュール内に閉じ込めたい変数は `static` つきにしておけば、それでモジュール化ができるわけです。このような「たまたま」の設計がうまかったことが、C が長く使われて来ている 1 つの要因だと言えるでしょう。

1.2 抽象データ型 (ADT)

さて、ここまででではモジュールではその内部に閉じ込めたデータ構造は「1 組」だけですが、次第に込み入ったソフトウェアが作られるようになってくると、そのような閉じ込めたデータを「N 組」保持したいというニーズが現れて来るようになりました。

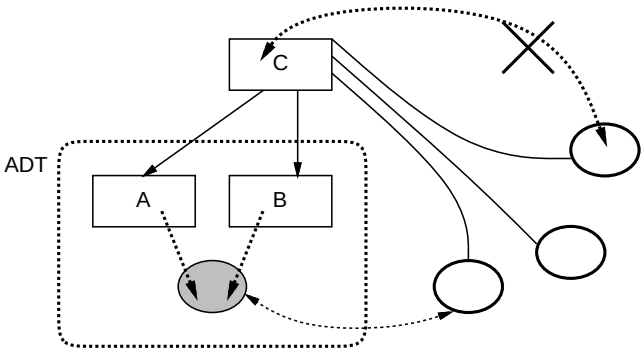

図 3: 抽象データ型

つまり、図 3 でいうと、モジュール外の手続き C にとっては当該データはただの「かたまり」であって、それを N 個保持しておくことはできますが、中身にアクセスすることはできません。しかし、そのデータをモジュールの手続きに渡すと、そこでは中身をあけてアクセスできるわけです。

これを「抽象データ型 (Abstract Data Types、ADT)」と呼びます。なぜそう呼ぶかというと、まず「ある種類のデータ」なので「データ型」なのですが、その中身は一般のコードからは「隠され (抽象化され)」ていて、当該モジュール (抽象データ型モジュール) の中に入らないとアクセスできないからです。これにより、中身のデータの整合性についてはモジュールの場合と同様、少数の手続きだけをきちんと検証すれば済みます。

別の見方をすると、ADT の「値」というのは、中がどうなっているかは外からは分からなくて、ただ単に「その値に対する操作 (手続き) を呼び出したときにどのように振る舞うか」だけによって定義されている、というふうにも見ることができます。ADT を取り入れた言語の代表は CLU ですが、今日のオブジェクト指向言語もその多くは ADT の「上に」オブジェクトの機能が追加されていると見ることができ、そして実際には ADT までの部分が多く使われている、とも見ることができます。

ADT に関しては、この振る舞いを数学的に定義することでモジュールの機能を定義し、それらを組み合わせてプログラムの正しさを検証する、という研究も多く行われてきました (まだ実用まで至っていませんが)。

ともあれ、ADT は「中身」と「それを利用する側」をきっちり分けて独立に開発することを可能にするため、複雑で大きなソフトウェアを開発する上で非常に重要な考え方だと言えます。さまざまな ADT を用意し、それらを組み合わせて「下請け」として利用することで、必要とするプログラムを迅速に低コストで開発しよう、という動きは現在主流になっています。

1.3 抽象データ型の C 言語による実装

さて、抽象データ型もまた、C 言語で実装することができます。それには、次のようにします。

- 抽象データ型内部のデータ構造は struct などにより定義する。
- 抽象データ型の値としては、その struct へのポインタを用いる。
- ヘッダファイルを使い分け、抽象データ型を実装するファイルでは内部の構造が記述された宣言を使うが、外部からはポインタのみ記述された宣言を使う。

実際に見てみましょう。ここで例として取り上げるのは、1 個ずつカウントアップする機能と、途中で「覚える」指示をしておくと、あとでその値に「戻る」ことができるようなカウンタです。まず、`main()` が参照するヘッダファイルを見ます。これを見ると、`counter` という抽象データ型は `void` へのポインタとして定義されています。

```
/* counter.h */
typedef void *counter;
counter counter_new();
void counter_incr(counter p);
void counter_mark(counter p);
void counter_reset(counter p);
int counter_get(counter p);
```

では、これを参照する `main()` とその実行例を見てみます。2 回 `mark` を読んでいますが、今のところ 1 個までしか覚えられないので最後の 2 回は同じ値になります。

```
/* counter.c */

#include "counter.h"
```

```
main() {
  counter c = counter_new();
  printf("%d\n", counter_get(c));
  counter_incr(c);
  counter_mark(c);
  counter_incr(c);
  counter_mark(c);
  printf("%d\n", counter_get(c));
  counter_incr(c);
  counter_incr(c);
  printf("%d\n", counter_get(c));
  counter_reset(c);
  printf("%d\n", counter_get(c));
  counter_reset(c);
  printf("%d\n", counter_get(c));
}
```

```
% a.out
0
2
4
2
2
%
```

それはともかく、こういう「特別な機能を持ったカウンタ」が抽象データ型として実現されていることは納得できたかと思います。では、その中身を見てみましょう。まずヘッダファイルから。こちらのヘッダファイルの中では、coutner_impl というレコードが定義されていて、counter はこのレコードへのポインタ型になっています。

```
/* counter_impl.h */

struct counter_impl {
  int count;
  int markval;
};

typedef struct counter_impl *counter;
```

では最後に、実装コードを見てみましょう。

```
/* counter_impl.c */

#include "counter_impl.h"

counter counter_new() {
  counter p = (counter)malloc(sizeof(struct counter_impl));
  p->count = p->markval = 0;
  return p;
}
```

```
void counter_incr(counter p) {
  p->count += 1;
}

void counter_mark(counter p) {
  p->markval = p->count;
}

void counter_reset(counter p) {
  p->count = p->markval;
}

int counter_get(counter p) {
  return p->count;
}
```

これを動かす場合は次のようにします。

```
% gcc -c counter_impl.c  ←オブジェクトファイル counter_impl.o 生成
% gcc counter.c counter_impl.o  ←ライブラリはオブジェクトを参照
% a.out
...
```

演習 1 この例題をそのまま動かしてみなさい。動いたら、次のような拡張をしてみなさい。

a. この実装は「1 つまでしか」mark できないが、「2 つまで」mark できるようにしてみよ。それで上の `main()` を実行すると 2 つ前まで戻れるので最後は「1」になるはず。
b. 「100 まで」mark できるようにしてみよ。
c. その他、自分で面白いと思う拡張を考えて実装せよ。

さて、このようにして C で抽象データ型は作れますが、これによってどのような利点が得られたでしょうか? また逆にこれって完璧ですか? または弱点がありますか? 弱点があるとすれば何でしょう?

1.4 抽象データ型の例: C++による「整数の集合」

前節末尾の質問に対する答えですが、C 言語でヘッダファイル等を駆使して抽象データ型を定義することはできるし実際に多く行われてはいますが、次のような弱点があると言えるでしょう。

- 必ずポインタ型にする必要がある (データ構造のサイズが分からないため)。このため記憶管理が必要になったりする。
- 操作の名前に型名を前置するなどして、名前がごっちゃにならないようにする必要がある。
- 引数としてポインタ型を必ず渡す必要がある。
- 適切なヘッダファイルの使い分けを行う必要がある。

全体として、言語そのものが持たない機能を「規約 (コンベンション)」によって実現することは、その規約を逸脱しないように「人間が」気をつける必要があって繁雑ですし、できたコードも規約を頭に置いて読まなければならないので十分すっきりした形にならないことが多いわけです。

では次の例として、「整数の集合」型を C++言語で書いてみます。こちらでは C++言語が持つクラスの機能を使うことで、上に書いた繁雑さの一部が克服できています (一方で C++の設計から来る繁雑さが加わっているけれど…後述)。

まず、ヘッダファイルに相当するクラス宣言部分から見てみます。C++ではクラス定義の中に内部のみ使用の宣言と外部から参照する宣言の両方を書くことができます (両者の区別を public 指定でおこなう)。ここでは集合の要素は配列 arr に順番につめて保持しますが、このような宣言は外部からは「無かったもの」として扱われます。にもかかわらず、この情報は外部に見せる必要がありますが、それはこの型の変数を宣言したときにどれだけの領域を取るかをコンパイラが分かるようにするためです。

また、操作名は size などのように短いものを使うことができます。これは、クラス名が前置されるので、他のクラスと同じ操作名でも衝突することはないからです。さらに、operator+のような名前で操作を定義することで、その操作を「+」など演算子を用いて呼び出すこともできます。さらに friend 指定の操作を定義することで、既にあるクラスに対する演算子の動作も定義できます (ここでは入出力に使っている)。

```
#include <iostream>
#include <string>
#include <cstdlib>
using namespace std;

static const int maxsize = 100;

class Intset { // set of ints
  int count;
  int arr[maxsize];
  void add1(int x);
public:
  Intset();
  int size() const;
  bool is_in(const int i) const;
  Intset operator+(const Intset &s) const;
//  Intset operator-(const Intset &s) const;
  Intset operator*(const Intset &s) const;
  class Overflow { };
  friend ostream& operator<<(ostream& o, Intset& s);
  friend istream& operator>>(istream& i, Intset& s);
};
```

実装部分はメソッドごとに個別に記述します。下請けメソッド add1 を利用することで個別の操作を簡単にできています。

```
Intset::Intset() { count = 0; }
void Intset::add1(int x) {
  if(count+1 >= maxsize) throw Overflow();
  arr[count++] = x;
}
int Intset::size() const { return count; }
bool Intset::is_in(const int i) const {
  for(int k = 0; k < count; ++k)
    if(arr[k] == i) return true;
  return false;
}
Intset Intset::operator+(const Intset &s) const {
```

```
  Intset r = s;
  for(int k = 0; k < count; ++k)
    if(!s.is_in(arr[k])) r.add1(arr[k]);
  return r;
}
Intset Intset::operator*(const Intset &s) const {
  Intset r;
  for(int k = 0; k < count; ++k)
    if(s.is_in(arr[k])) r.add1(arr[k]);
  return r;
}
ostream& operator<<(ostream& o, Intset& r) {
  o << "{ ";
  for(int k = 0; k < r.count; ++k) o << r.arr[k] << ' ';
  o << '}'; return o;
}
istream& operator>>(istream& i, Intset& r) {
  r.count = 0;
  while(i.peek() != '\n') {
    int x; i >> x; r.add1(x);
  }
  i.get(); return i;
}
```

さいごに main() を示します。これは集合の値を計算する「電卓」になっています。

```
int main(void) {
  Intset r, x;
  while(true) {
    char c; cout << "? "; cin >> c;
    if(c == 'q') return 0;
    switch(c) {
case 'q': return 0;
case '=': cin>>x; r = x; break;
case '+': cin>>x; r = r + x; break;
//case '-': cin>>x; r = r - x; break;
case '*': cin>>x; r = r * x; break;
default: continue;
    }
    cout << r << '\n';
  }
}
```

演習 2 この例題をそのまま動かせ。動いたら次のような拡張をしてみよ。

a. 「集合の引き算」演算は現在、宣言や利用部分がコメントアウトされている。引き算機能を追加実装して動かせ。

b. 現在の版では配列のサイズが固定なので、大きい集合を作るとあふれる。この弱点を改良してみよ。

c. その他、自分で面白いと思う拡張をおこなえ。

1.5 C++言語の面倒なところ

配列のサイズを固定でなくするには、配列をオブジェクト本体とは別の場所に取り、一杯になったら取り替えればいいわけです。

```
class Intset { // set of ints
  int count, limit;
  int *arr;
  ...
```

こうした上で、add1 で要素を追加するときにあふれたら倍の大きさに取り直してコピーすればよいわけです。

```
void Intset::add1(int x) {
  if(count+1 >= limit) {
    int *a = new int[limit*2+1];
    for(int i = 0; i < count; ++i)
      a[i] = arr[i];
    delete[] arr; arr = a;
    limit = limit*2+1;
  }
  arr[count++] = x;
}
```

しかし、これだけで OK でしょうか? もちろん、最初に Intset を初期化するときに、小さめの配列を入れておく必要があります。

```
Intset::Intset(int c) {
  count = 0; limit = c; arr = new int[c];
}
Intset::Intset() {
  count = limit = 0; arr = new int[0];
}
```

大きさが「0」でもいいというのはちょっと面白いところです。それはいいのですが、この配列の領域はいつ解放するのでしょう? それは、Intset を使わなくなったところです。これを表現するために、C++ではデストラクタ (後しまつメソッド) というのが用意されています。

```
Intset::~Intset() { delete[] arr; }
```

これが無いと、確保した領域が返却されないままになり、使用可能な領域がどんどん減少していきます (メモリリーク)。C++が自動ごみ集め (GC) を採用しなかったことの弱点がこういうところに現れるのですね。

さて、問題はこれだけではありません。たとえば、次のようなコードを見てください。

```
Intset s1, s2;
...
s2 = s1;
```

このような代入は、C でも C++でも「領域のそのままコピー」によって実現される、という言語仕様です。しかしそのような実装だと、図 4 下のような状況になってしまい、片方の配列がリークし、もう片方の配列は共有されてしまって正しく動作しなくなります。

これを防ぐには、次のように「代入演算子」を自前で記述する必要があるのです。これは、自分のサイズをまず 0 にして、それから右辺の集合の各内容を自分に追加します。あと、自分で自分に代入

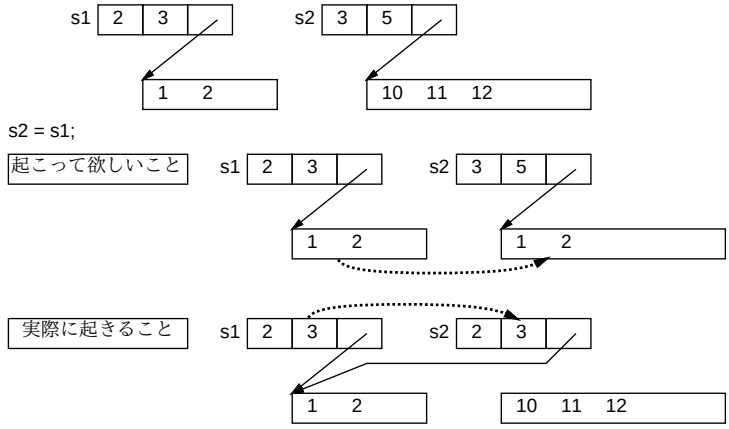

図 4: C++でのコピーの起こり方

するときにコピーしはじめたらまずいのでその回避と、「a = b = c」みたいな多重代入のために最後に自分自身を返します。

```
Intset& Intset::operator=(const Intset& s) {
  if(this != &s) {
    count = 0;
    for(int i = 0; i < s.count; ++i)
      add1(s.arr[i]);
  }
  return *this;
}
```

やれやれ、でしょうか? 実はまだ終わっていません。そもそも、次の 2 つのコードは同じだと思いますか、違うものだと思いますか?

```
(A) Intset s2 = s1;   (B) Intset s2; s2 = s1;
```

同じであれば、これまでに用意したコンストラクタとコピー演算子で OK ですが、それだと「まず使いもしない配列を用意して、それからそれを 0 から拡張する」ことになりますよね。サイズが最初から分かっているのに無駄だと思いませんか。そのような無駄を避けるために、C++ではこれらは「別のもの」ということになっています。そして、(A) に対応するのは「自分と同じ型の値を渡されて自分を初期化する」コンストラクタ (コピーコンストラクタ) の呼び出しになります。

```
Intset::Intset(const Intset &s) {
  arr = new int[s.count];
  limit = count = s.count;
  for(int i = 0; i < count; ++i)
    arr[i] = s.arr[i];
}
```

これらを併せると、この「配列分離版」の宣言は次のようになるのです。C++の「その場にデータ構造が埋め込める」「GC を採用しない」「フルスピードで動く」という方針を徹底した結果が、この面倒くささだ、というわけです。

```
class Intset { // set of ints
  int count, limit;
  int *arr;
```

```
  Intset(int c);    //←あった方がよい
  void add1(int x);
public:
  Intset();
  Intset(const Intset &s);  //←必要
  ~Intset();                //←必要
  Intset& operator=(const Intset& s);  //←必要
  int size() const;
  bool is_in(const int i) const;
  Intset operator+(const Intset &s) const;
  Intset operator-(const Intset &s) const;
  Intset operator*(const Intset &s) const;
  friend ostream& operator<<(ostream& o, Intset& s);
  friend istream& operator>>(istream& i, Intset& s);
};
```

演習 3 配列分離版の Intset を完成させ、テストせよ。どのようなテストをすればここまでに説明した問題が克服されていることが確認できるか考えること。

2 オブジェクト指向

2.1 オブジェクト指向とは

「オブジェクト指向とは何でしょうか?」この質問はオブジェクト指向言語の黎明期にさんざん投げかけられたけれど、今でも答えるのが結構難しい質問だと思います。あなたならどのように答えますか?

私の回答は、次のようになります。

> オブジェクト指向とは、プログラム (やソフトウェア) が扱う対象のそれぞれを自立した「もの」として扱うような「考え方」である。

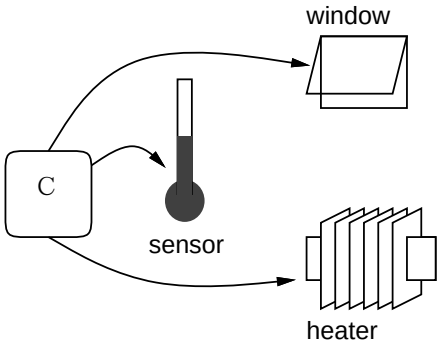

図 5: オブジェクト指向以前の考え方

少し具体例で考えてみましょう。たとえば、「温室の温度調節システム」を考えてみます。旧来の考え方であれば、システムは手続きと受動的なデータの集まりだと考え、手続きとして「センサーを見て、気温が下がって来たらヒータを通電し窓を閉じるが、気温が上がって来たら逆にヒータを止めて窓をあける」など機能を中心に考えます。これはコンピュータの機能から見れば自然ですが、制御する要素や条件が複雑になるとごちゃごちゃになりやすいという問題があります (ちょうど抽象データ型以前の手続きコードのようなもの)。

これに対し、オブジェクト指向的な考え方では、「センサー」「ヒータ」「窓開閉装置」などの「もの」を中心に考え、センサーは観測した温度に応じて他の装置にメッセージを送り、それを受けたヒータや窓開閉装置はそれらのメッセージに呼応して自分の判断で通電制御や窓開閉をおこなう、というふうになります (図 6)。

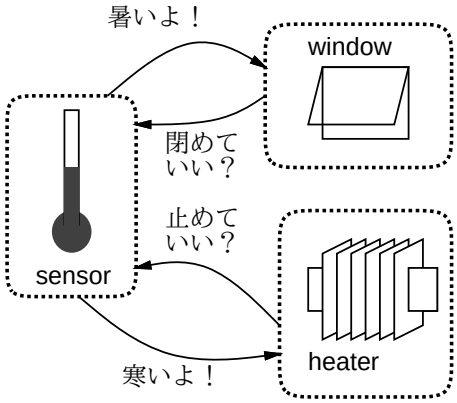

図 6: オブジェクト指向での考え方

さて、この説明を聞いて「なるほどよさそうだ」と思いますか、または「結局やっていることは同じでは」と思いますか? プログラムの動作そのものは、目指すところが同じなのですから当然同じになるでしょう。ただ、そこまでに至る道筋として、オブジェクト指向では「窓」「ヒータ」などの「もの」に着目してこれらがプログラム上の単位となるので、人間にとって構造が分かりやすく、考えやすい、ということが利点になります。まあ、その分言語には道具だて (概念) が増えてややこしくはなるのですが…

2.2 オブジェクト指向言語の中心概念

では、考え方については「もの」を中心に考える、でよいとして、オブジェクト指向言語はここまでに見て来た「抽象データ型」の機能を持つ言語とはどう違っているのでしょうか?

これにはいくつか考え方があります。1 つは、抽象データ型は結局、それぞれのデータ型の値がそれぞれ「固有のもの」なのだから、オブジェクト指向におけるオブジェクトと変わらない、とする考え方です。

まあそれでもよいのですが、私が抽象データ型言語 (CLU とか) で結構たくさんプログラムを組んでからオブジェクト指向に進んだ時に思った違いについてここで説明しておきましょう。

抽象データ型言語でも、上に書いたように個々のオブジェクトを定義することは問題ありません。たとえば、図形を扱うプログラムだったら、長方形、円、三角形などの形に対応する抽象データ型を作ることができます。ですが、それを画面に描画する段階になると、次のように種別ごとに枝分かれする必要があるのです。

```
if typeof(f) in 三角形 then
  三角形$描画 (convert_図形_to_三角形 (f))
elsif typeof(f) in 長方形 then
  長方形$描画 (convert_図形_to_長方形 (f))
elsif typeof(f) in 円 then
  円$描画 (convert_図形_to_円 (f))
...
```

これは非常に繁雑ですね。これに対し、オブジェクト指向言語であればこれは次のように書けば済むのが普通です。

```
f.描画()
```

こうすると、言語の実行系はまず f がどの種別のオブジェクトであるかを調べて、中身が三角形なら三角形の、円なら円の「描画」を読んでくれます。つまり、「具体的にどのメソッドが呼ばれるかは、呼び出す瞬間に (実行時に)f に入っているオブジェクトの種類に応じて変わる」わけです。これを動的分配 (dynamic dispatch)、このように f という 1 つの変数が同時にさまざまな種別のものを表現することを多相性 (polymorphism) と呼びます。[1]

なぜ多相性がありがたいかというと、上のような長いコードが下のように簡潔になるということがまずありますが、それと同時にこのような「簡単化」が人間にとって自然だ、ということもあります。たとえば筆記用具にはボールペンも万年筆も鉛筆もありますが、人間が「書く」ときにはどれでも同じようにすれば済みます。つまり現実世界そのものが多相性を持っていて、「具体的に何であるか」はあまり考えなくても用が済むようになっているのです。これと同じことをプログラミング言語に持ち込むことで人間にとって「考えやすくなる」「細かいことを考えなくて済む」という恩恵は非常に大きいものと考えます。

このほかに、たとえばある種別のオブジェクトの定義 (クラス記述) を「土台にして」他の種別を定義できること (継承、inheritance) を重視する人もいるかも知れませんが、継承機能のないオブジェクト指向言語も (初期には) それなりにあったりしたので、これが「要件」とは考えないことが多いようです。

2.3 C 言語による動的分配の実現

では、動的分配の実現方法を見るため、わざと C 言語で実装してみます。基本的には次のような方針になります。

- 動的分配により呼ばれるメソッドの表 (vtable) をデータ構造として持つ。その「何番目」はオブジェクトの種類が違っても共通になるようにしておく。
- 各オブジェクトは決まった場所 (通常は先頭) に自分用の vtable の場所を保持しておく。
- メソッドび出し時には「決まった場所→ vtble →その決まった位置」とたどってメソッドのポインタを取り出し、そこを呼び出す。

これを C 言語で実装してみたものを示します。ものの種類ごとにそのもののデータを保持する構造体を定義し、その先頭位置が vtable へのポインタになっています。

```
#include <stdio.h>

struct vtbl {
  int (*pname)();
  int (*bark)();
};
struct object {
  struct vtbl *vtbl;
};
struct cat {
  struct vtbl *vtbl;
  char *name;
  double weight;
};
int cat_pname(struct cat *obj) {
```

[1] オブジェクト指向では操作のことを「メソッド」と呼ぶので、ここからはこちらの用語を遣います。

```
  printf("I am cat, my name is %s, weight %lg\n", obj->name, obj->weight);
}
int cat_bark(struct cat *obj) {
  printf("Meaw\n");
}
struct vtbl cat_vtbl = { cat_pname, cat_bark };
struct object *new_cat(char *name, double weight) {
  struct cat *c = (struct cat*)malloc(sizeof(struct cat));
  c->vtbl = &cat_vtbl; c->name = name; c->weight = weight;
  return (struct object*)c;
}
struct dog {
  struct vtbl *vtbl;
  char *name;
  double speed;
};
int dog_pname(struct dog *obj) {
  printf("I amd dog, my name is %s, speed %lg\n", obj->name, obj->speed);
}
int dog_bark(struct dog *obj) {
  printf("Vow!\n");
}
struct vtbl dog_vtbl = { dog_pname, dog_bark };
struct object *new_dog(char *name, double speed) {
  struct dog *c = (struct dog*)malloc(sizeof(struct dog));
  c->vtbl = &dog_vtbl; c->name = name; c->speed = speed;
  return (struct object*)c;
}
main() {
  int i;
  struct object *o[3];
  o[0] = new_cat("tama", 8.0);
  o[1] = new_dog("pochi", 12.0);
  o[2] = new_cat("mike", 20.0);
  for(i = 0; i < 3; ++i) {
    (o[i]->vtbl->pname)(o[i]); (o[i]->vtbl->bark)(o[i]);
  }
}
```

確かにできているが、正しくデータ構造を合わせたり呼び出したりするためにかなり規約を意識して書かなければならないことが分かる。ちなみに、これを C++で書き直すと次のようになる。

```
#include <cstdio>

class object {
public:
  virtual int pname() = 0;
  virtual int bark() = 0;
};
```

```
class cat: public object {
  char *name;
  double weight;
public:
  cat(char *n, double w) { name = n; weight = w; }
  virtual int pname() {
    printf("I am cat, my name is %s, weight %lg\n", name, weight);
  }
  virtual int bark() { printf("Meaw\n"); }
};

class dog: public object {
  char *name;
  double speed;
public:
  dog(char *n, double s) { name = n; speed = s; }
  virtual int pname() {
    printf("I amd dog, my name is %s, speed %lg\n", name, speed);
  }
  virtual int bark() { printf("Vow!\n"); }
};

int main(void) {
  object *o[3];
  o[0] = new cat("tama", 8.0);
  o[1] = new dog("pochi", 12.0);
  o[2] = new cat("mike", 20.0);
  for(int i = 0; i < 3; ++i) {
    o[i]->pname(); o[i]->bark();
  }
}
```

演習 4 C 版と C++版の違いについてできるだけ多く列挙し、なぜそのような違いが現れているのかも説明しなさい。

演習 5 C 版と C++版に新しいメソッドを追加しなさい。また新しいものの種類も追加しなさい。その作業から分かったことを整理して示しなさい。

2.4 継承とその実装

次の質問は「継承とは何ですか」になりますが、どうでしょうか。

まずここで、ここまでに説明してきたオブジェクト指向言語は「クラス方式」つまりものの種類に対応した記述をまとめた構文単位であるクラスがあるような言語である、ということを注意しておきます。Java、C++はじめ、現在主流であるオブジェクト指向言語はクラス方式になっています。

その上で継承について定義するなら、「既にあるクラスを土台として、新たなクラスを (まったくゼロから書くよりも容易に) 定義できるような機能」と言えばよいでしょうか。なお、クラス方式でない言語まで含めて通用するように直すなら、上記の「クラス」を「オブジェクトの種類」に置き換えればよいかと思います。

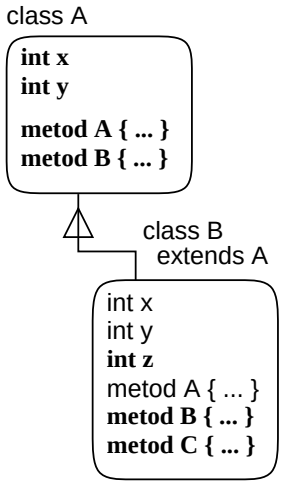

図 7: 継承の概念

具体的には図 7 にあるように、子クラスは親クラスから「変数定義」と「メソッド定義」を継承してきます。さらにそれに加えて子クラスでは、(1) 変数定義やメソッド定義を追加すること、および (2) 親クラスから継承してきているメソッド定義を別のものに差し替えること (オーバライド) ができます。[2]

では、これによってどのような「効果」が得られるのでしょうか?

(a) 子クラスは親クラスのメソッド一式を引き継いでいるので、子クラスのオブジェクトは親クラスのオブジェクトの「代わりに」用いることができる。

(b) 子クラスでは変数やメソッドの追加により、親クラスに機能を「追加」できる。また、オーバライドにより親クラスのメソッドの機能も「調整/変更」できる。[3]

この (a) と (b) はかなり違う性質のものだということがお分かりでしょうか。(b) は「少ない労力で新しいクラスを作れる」という「実装面」の特性であるのに対し、(a) は「上位互換なクラスができる」という「インタフェース面」の特性だからです。では、どちらが重要でしょう?

実際のところ、Smalltalk-80 でオブジェクト指向が広まった時には (b) の部分がクローズアップされたこともあるのですが、今日のプログラミング実践においては重要なのは明らかに (a) です。つまりデザインパターンでも動的分配でも、(a) の「互換性のある複数のオブジェクトを使い分ける」ことが基盤になっています。そして、Java や C++をはじめ型検査のあるオブジェクト指向言語では、親クラスと子クラスの間に包含関係を持たせることで、親クラス型の変数に子クラスのオブジェクトを入れることができるようにしています。

こうしてみると、継承という 1 つのメカニズムに (a) と (b) 両方の機能が混ざっている、というのはあまりよろしくないことだと見ることもできます。実際、Java などでは (a) の部分だけを取り出したものをインタフェース機能として提供しています。Java ではインタフェースも包含関係を持つ型であり、動的分配に使用できますが、実装が付随していないので、継承よりもずっと自由に (柔軟に) 使うことができます。とはいえ、Java でも継承は残してあるので、そこでは (a) と (b) はやはり一緒です。今後は (a) と (b) が完全分離した言語が望まれるのでは、と個人的にはずっと思っているのですが…

では、継承はどのように実装されるのでしょう? ここでは、どのクラスも 1 つの親クラスだけを持つ (単一継承)、という制限のもとで効率よく実装できる方法を説明しておきます。

単一継承の枠内では、子クラスのインスタンスは親クラスのインスタンスの「後ろに」自分で追加したインスタンス変数を持つようなレコードで表現できます。また同様に、子クラスの vtable は親

[2] さらに、差し替えたメソッド内から元の (親クラスの) メソッドを呼び出すこともできるようになっています。
[3] 元のメソッドも呼び出せるので、調整のためのコードも最小限で済ませることができます。

クラスの vtable の「後ろに」自分で追加したメソッドのスロットを持つように配置できます (そして親部分のスロットに入れるメソッドポインタは親と基本的に同じですが、ただしオーバライドしたメソッドについては代わりにそちらのポインタが入ります)。これを図 8 に示します。

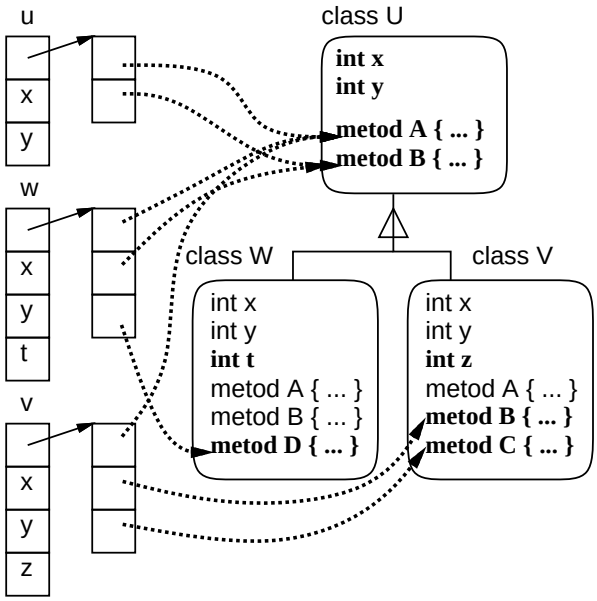

図 8: 単一継承の素直な実装

この方法では、子クラスのインスタンスや vtable の「先頭部分は」親と同じ構造になるので、子クラスのインスタンスをそのまま親クラス用のコードに渡しても正しく動作します。もちろん、あるクラスの複数の子クラス (つまり兄弟クラス) 間では互換性はありませんが、それらは本来別のものであり (強い型の言語では互換性の無い型になる)、まぜて使われることはないので、これでよいのです。

多重継承 (複数の親クラスを許す) の場合や、Java のインタフェースのように 1 つの型が多数のインタフェースに所属する場合には、このように綺麗に互換性のあるデータ構造は作れないので、実行時に変数名やメソッド名で検索するなどの方法が使われます。

演習 6 先の C 言語版の動的分配の例題をもとに、図 8 の方式で「継承」を実現した例題を作ってみなさい。