プログラミング言語論 2012 # 5 — オブジェクト指向とジェネリクス (2)

久野　靖*

2012.4.26

3　型とジェネリック性

3.1　プログラミング言語と型

次の質問: 「型 (type)」とは何でしょうか?、また、なぜ「型」があるのでしょうか? 「型」にはどのような属性が備わっていますか (ないし、備わっているべきですか)?

基本的には、型とは「値の種類」だと考えます。「値の集合」という説もありますが、なぜ集合を分けるか (異る型があるか)、ということになると、分けた以上それぞれが「種類」になるので同じことですよね。

では、なぜプログラミング言語には型があるのでしょうか。

(a) データの種類ごとに記憶領域やその構造や適用できる演算 (ないし CPU 命令) が異るので、その区別を記述する (図 1 左)。

(b) プログラマに「ここで扱う値はどのような種類」かを記述させることで、整合性をチェックしたり読む際の手がかりとする。

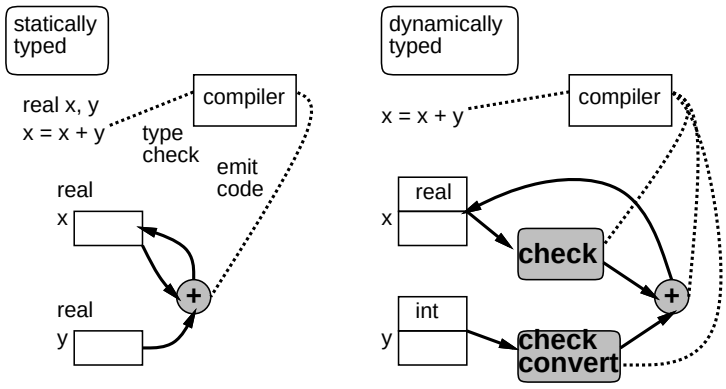

図 1: 静的型検査と動的型検査

これもまた、(a) が実現面、(b) が仕様面という区分にあります。歴史的には、FORTRAN など初期の言語では記憶領域サイズや出力する演算命令を整数と実数で区分するために型ができた、ということは言えると思います (なにしろ FORTRAN では「`INTEGER*4`」とか「`REAL*8`」とか書いてデータのサイズも一緒に型の一部になっていました)。これが正しくないと正しい演算が行えなかったわけです。

*経営システム科学専攻

しかし、そのような理由からプログラム上で型を指定することが必須かというと、必ずしもそうではありません。動的な型の言語 (図 1 右) ではデータと一緒に型を表すタグを格納することで、プログラマが型を指定しなくても、実行時に型をチェックして正しい演算を行うことができます。CPU が遅かった時代にはこのような方式は遅いという弱点が目だったのですが、現在は CPU も高速になったのであまり問題はなさそうです。

一方で、(b) の仕様面という部分は今日、非常に大きくなっています。とくに抽象データ型以降では、「内部は見せないけれど、種別どうしを互いに区分し、またどのような操作が可能かを示すことで型検査を行う」という面が強くなっています。現在では型とは「その名前と、メソッドシグニチャ (名前およびパラメタと返値の型) の集合」という意識が強いと考えます。

演習 8 ここにあげた例のほかに、自分の知っているプログラミング言語におけるジェネリック性の発現箇所をできるだけ挙げてみなさい。

3.2 型パラメタとジェネリック性

プログラミング言語においてジェネリック (generic、汎用的) であるとは、あるコードが複数の型に対して適用できることを言います。たとえば、乗算演算子「`*`」は、「`1 * 2`」では整数乗算、「`1.0 * 2.0`」では実数乗算を行うので、ジェネリックだと言えます。FORTRAN では `max()` などの組み込み関数も整数引数なら整数値、実数引数なら実数値を返すようになっていて、総称関数と呼ばれていました。

さらに、型についてもジェネリックなものがあります。その代表的なものが配列で、`array[int]` は整数の配列、`array[double]` は実数の配列というふうに与える型によって 1 要素のサイズがさまざまな配列ができますが、「並んでいる」という構造は共通です (図 2)。この例のように、さまざまな型を与えるということは、型がパラメタ (type parameter) になっている、というふうに見ることができます。そして、型パラメタを指定することでさまざまな具体的な型ができることから、`array[]` は型生成子 (type generator) である、というふうに言います。

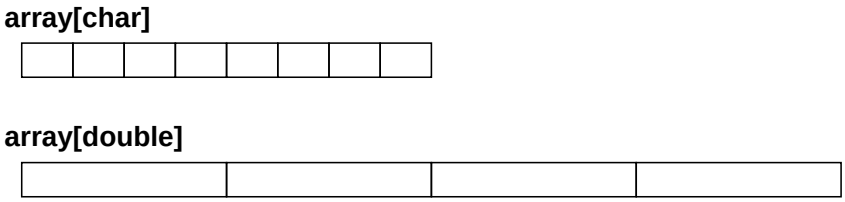

図 2: 配列=型生成子

ここまで見て来た、演算子、総称関数、配列などはすべて言語に組み込みの機能として始まりました。つまり、初期の言語では型は「具体的に固定されたもの」であり、さまざまな型に適用可能なコードを書くということはその言語の枠内ではできず、処理系が予めサポートしているものだけが使用可能だったわけです。

しかし、プログラミング言語では「直交性」が重視されることから、組み込みの型生成子がある以上、ユーザも型生成子を書けるようになるべきだ、という考え方が生まれ、そのような言語機構が模索されました。CLU などはユーザによる型生成子をサポートした最初期の言語です。

直交性のためというだけでなく、コードを書く側から見ても型生成子のニーズはあります。たとえば、配列と同様にさまざまな型の値を格納できるものとして、連結リストや B 木などのデータ構造が考えられます。しかし、従来の言語では「文字列の B 木」「実数の B 木」などはすべて別の型なので、そのつど別のモジュール (クラスや抽象データ型) を書く必要があります。しかしそれらのモジュールのコードは実質的に同一です。これは面倒であり望ましくありません。

ここでオブジェクト指向言語においては、型の包含関係があるので、すべてのオブジェクトの親クラスとして `Object` があるような言語 (Java が代表的) では、`Object` を保持するような連結リストや

B 木を作っておけば、任意のオブジェクトが入れられます。実際、JDK 1.4 までは Java ではそのようなやり方が標準でした。しかし、この方法には次のような弱点があります。

- オブジェクト型で格納しておいた値を取り出した後、元の型に戻す (ダウンキャスト) 必要があり繁雑
- 任意のオブジェクトが入れられてしまうということは、格納する値についての型検査が行えないことを意味する
- 格納する型の種別に応じた最適化が行えない☆

このようなことから、JDK 1.5 以降では型パラメタ機構が導入されています。ただし、「JVM を変更しない」という方針の実装であるため、実行時には型パラメタの情報が保持されず、また実現としてはこれまでと同様にパラメタ型を `Object` 型として扱うコードが動くようになっています (均質な実装)。このため、☆の速度上の利点はなく、またパラメタ型に関して次のような制約があります (とっても不便!)。

- 基本型は型パラメタに入れられない
- パラメタ型の配列は作れない
- パラメタつき型の配列は作れない

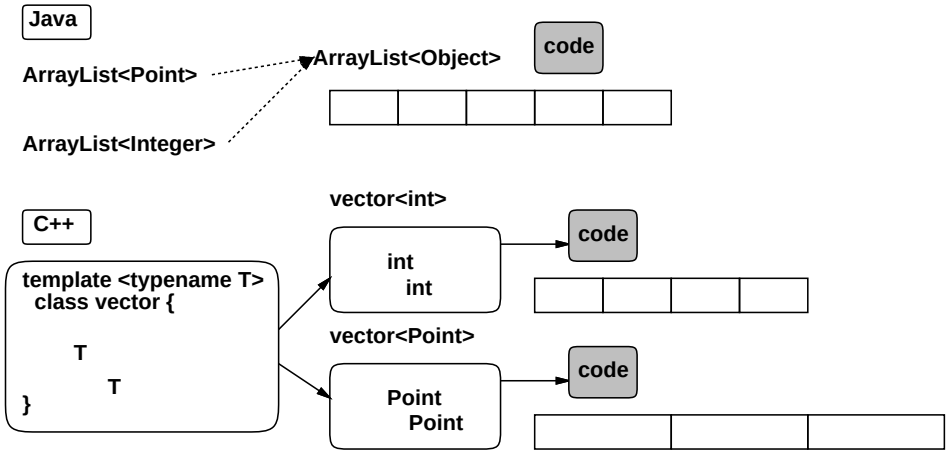

図 3: 型パラメタの均質な実装と非均質な実装

一方 C++は、比較的初期からテンプレート (template) 機能として型パラメタの機能を提供して来ました。C++では Java と異り、別個の型パラメタを指定するごとにパラメタの位置に実際のパラメタ型を埋め込んだコードをコンパイルする実装 (非均質な実装) を採用しています。この方法はコード量が多くなりがちですが、個別の型に応じた最適化が行いやすいという利点があります。

実は、C++の方がその実装もあいまって「変わった」ことが色々できるので、以下ではこちらを中心に取り上げます。

演習 9 Java の Generics 機能と C++のテンプレート機能の比較表を作成してみなさい。

3.3 C++で型パラメタつきクラスを作る

C++では型パラメタはテンプレートクラスでパラメタとして「`typename` 名前」を指定することで作ることができます。具体例として、次々に値を格納できるが、最近の 2 つだけを覚えているクラス」を実現してみます。

```
//last2.cc
#include <iostream>
using namespace std;

template<typename T> class last2 {
  T v1, v2;
public:
  last2(T x, T y) { v1 = x; v2 = y; }
  void put(T v) {
    v2 = v1; v1 = v;
  }
  T get() {
    T x = v1; v1 = v2; return x;
  }
};

int main() {
  last2<int> a(0, 0);
  a.put(1); a.put(2); a.put(3);
  cout << a.get() << '\n';
  cout << a.get() << '\n';
  cout << a.get() << '\n';
  last2<char*> b("", "");
  b.put("ab"); b.put("cd"), b.put("ef");
  cout << b.get() << '\n';
  cout << b.get() << '\n';
  cout << b.get() << '\n';
}
```

演習 10 上の例題を動かしなさい。動いたら、「最後の 3 つのものを覚える」ように拡張しなさい。また、これとは別の機能を持つ「覚える」クラスを作ってみなさい (スタックとキューが定番でしょう)。

3.4 Standard Template Library (STL)

STL とは C++の標準ライブラリの中に入っている、テンプレートを駆使したライブラリ群であり、さまざまなアイデアが盛り込まれています。その特徴として、次のものがあげられます。

- コンテナクラス群 (さまざまな型のデータを格納する機能を提供する抽象データ型)
- コンテナクラスへのアクセスは、コンテナクラス自体のメソッドではなくイテレータと呼ばれるアクセス用オブジェクトを通じておこなう (その方が高速化しやすい)
- イテレータに対して働く汎用のアルゴリズム群の提供
- アルゴリズム群はテンプレート関数であり、高速化しやすい

なぜ、テンプレートが高速化しやすいかを考えてみましょう。たとえば、配列の 2 要素を交換する関数 `swap()` を C で書くと次のようになるでしょう。

```
void swap(int *x, int *y) {
  int z = *x; *x = *y; *y = z;
```

```
}
...
int a[100]; ... swap(&a[i], &a[j]); ...
```

この場合、swap() の呼び出し箇所からは 2 つのメモリアドレスを渡し、swap() の中からはポインタ経由で読み出しと書き換えを 2 回ずつ行います。それ以上速くできそうな気がしないでしょうか? これをテンプレート関数にした版を見てみましょう。

```
template<typename T> inline void swap(T& x, T& y) {
  T z = x; x = y; y = z;
}
...
float a[100]; ... swap(a[i], a[j]); ...
```

この場合、inline と指定されているので、次のように書いたのと同じことになります。

```
float a[100]; ... { float z = a[i]; a[i] = a[j]; a[j] = z; } ...
```

この範囲の外側で a[i] や a[j] がどのように使われているかにもよりますが、大抵は (交換すると判断する以上)、a[i] や a[j] はレジスタの上に載っており、それを格納する (またはレジスタ上の値を置き換える) だけなので、このコードは C 版よりずっと高速なコードにコンパイルできるわけです。

次に、イテレータについて説明しましょう。イテレータとは、次の性質を持つようなオブジェクト x です。

- *x でコンテナ内の個々の値を参照できる
- x++でコンテナ内の「次の要素」を指すように進められる
- <で大小を比較できる。

そして、コンテナオブジェクト c は、先頭要素を指すイテレータを返すメソッド c.begin()、末尾要素の「次」を指すイテレータを返すメソッド c.end() があります。これらを使うと、配列の STL 版 (伸び縮みできる配列) である vector に 10 個の値を入れて打ち出す例は次のようになります。

```
//foreach0.cc
#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;

int main() {
  vector<int> a;
  for(int j = 0; j < 10; ++j) a.push_back(j);
  for(vector<int>::iterator i = a.begin(); i < a.end(); ) {
    cout << *i++ << endl;
  }
}
```

なんか C 言語で見慣れたような書き方ですが…実は、配列を指す要素ポインタとイテレータは「同じ形になる」ように設計されています。この、同じ形に使えれば動く、というのは C++のテンプレートの実装 (埋め込んでからコンパイルする) ゆえに許されていることで、非常に強力です。

さて、先のようなコードを見慣れているということは、いつも同じようなことを書いているということでもあります。そこで、このループを STL の関数 foreach() に置き換えることができます。

```
//foreach1.cc
#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;

template<typename T> void f(T x) {
  cout << x << '\n';
}

int main() {
  vector<int> a;
  for(int i = 0; i < 10; ++i) a.push_back(i);
  for_each(a.begin(), a.end(), &f<int>);
}
```

foreach には「各要素に対してやりたいこと」を記述した関数を渡します。ここでは、任意の型パラメタ T について、それを標準出力に打ち出すだけのテンプレート関数 f() を用意して渡しています。

しかしこれはこれでいいのですが、たとえば単に打ち出すだけでなく、「合計を取る」こともしたいものとします。となると、f() の中でずっと値を覚えて置きたいことになります。どうしましょうか。たとえば、集計値をグローバル変数にするというのは…最低ですね。そこで、次のようにします。

```
//foreach2.cc
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

template<typename T> class sum {
  T res;
public:
  sum(T init) { res = init; }
  void operator()(T x) { res += x; cout << res << "\n"; }
  T result() const { return res; }
};

int main() {
  vector<int> a;
  for(int i = 0; i < 10; ++i) a.push_back(i);
  sum<int> s(0);
  s = for_each(a.begin(), a.end(), s);
  cout << s.result() << '\n';
}
```

つまり、foreach の中では渡された関数を「f(値)」のように呼び出すところがあるわけですが、この「f(値)」は実は関数呼び出しでなくて、「オーバライドされた関数呼び出し演算子」であってもよいわけです。C++コンパイラは「形があっていれば」ちゃんとそのように翻訳してくれます。それにより、クラス sum の operator() が呼び出され、そこで合計を累計していきます (ここではついでに値も表示していますが)。

このようにC++のテンプレートでは、上記のイテレータも含め「同じ形に見える」ならそのまま使えることをうまく活用してコードの再利用を可能にしているわけです。

演習 11 上の例をそのまま動かしなさい。動いたら、「最大値を見つける」「最大値の次に大きい値を見つける」版も同様に作りなさい。

3.5 型パラメタに対する制約

では次の例として、次々に値を入れて行くと、最大値を常に覚えている、という例を作ってみます。

```
//maxbuf1.cc
#include <iostream>
using namespace std;

template<typename T> class maxbuf {
  T max;
public:
  maxbuf(T x) : max(x) { }
  void put(T x) { if(max < x) max = x; }
  T get() { return max; }
};

int main() {
  int a1[] = { 5, 9, 2, 7, 6 };
  maxbuf<int> m1(a1[0]);
  for(int i = 1; i < 5; ++i) m1.put(a1[i]);
  cout << "a1: " << m1.get() << '\n';
  double a2[] = { 3.0, -2.7, 6e2, 4e1, 0.5 };
  maxbuf<double> m2(a2[0]);
  for(int i = 1; i < 5; ++i) m2.put(a2[i]);
  cout << "a2: " << m2.get() << '\n';
}
```

別に問題はなさそうですが、パラメタに入れる値の種類によってはうまく行きません。

```
//maxbuf2.cc
#include <iostream>
using namespace std;

template<typename T> class maxbuf {
  T max;
public:
  maxbuf(T x) : max(x) { }
  void put(T x) { if(max < x) max = x; }
  T get() { return max; }
};

class point {
  double x, y;
public:
```

```
  point(double a, double b)
    : x(a), y(b) { }
//bool operator<(point &p) {
//  return x*x+y*y < p.x*p.x+p.y*p.y;
//}
};

int main() {
  point a3[] = { point(0,0), point(1,1), point(2,2), point(3,3), point(4,4) };
  maxbuf<point> m3(a3[0]);
  for(int i = 1; i < 5; ++i) m3.put(a3[i]);
}
```

これをコンパイルした結果を示します。

```
maxbuf2.cc: In member function ‘void maxbuf<T>::put(T) [with T = point]’:
maxbuf2.cc:27:   instantiated from here
maxbuf2.cc:9: error: no match for ’operator<’ in
  ’((maxbuf<point>*)this)->maxbuf<point>::max < x’
```

このように、maxbufの中で「operator<がない」というエラーが出ています。それはそうで、point型どうしを大小比較しようとしても、その演算子が無ければ比較できません。そこでコメントアウトしてある箇所を外せば大小比較ができるようになるので通ります。

このように、パラメタつきの型について、その型の値を操作する (その型のメソッドを呼ぶ) 必要があるときは、どのようなメソッドがあるか、というチェックが必要なわけです。C++は上で見たように、「実際に埋め込んでコンパイルしてみて通るかどうか」で決めていますが、これはちょっと場当たり的すぎます。そこで、あらかじめ型に対して「こういう操作を持っている」という宣言をおこなう機能 (concept 機能) を言語仕様に追加しようとしていますが、提案からだいぶたっているものの、まだ議論が収まっておらず、最新版の C++0x でも入っていません。

ちなみに Java Generics の方は、パラメタ型に対して「このような型/インタフェースに従う」という指定ができるようになっていて、これを使ってパラメタ型の操作をチェックできます。しかし、あらかじめパラメタに渡す型に決まったインタフェースを実装させないといけない、というのはかなり不自由なのでこれがよいとも言い切れない状況です。

3.6 テンプレートの特殊化とその応用

テンプレートはパラメタとしてさまざまな型を受け取る汎用性が基本だが、型によっては特別な実装を用いた方がよいこともあります。そのような場合は、別途「専用の」実装を用意することもできます。

```
template<typename T> class last2 {
  ... // 一般用
};

template<> class last2<void*> {  // template<> が特殊化を意味する
  ... // void*専用
};

template<typename T> class last2<T*> { // 部分特殊化: T*のみ該当
  ... // T*専用
};
```

vector のようなコンテナクラスでは、ポインタを格納するコンテナは上のように別バージョンを用意して、T*版も中では全部 void*用を使用するようにしています。これにより、さまざまな T ごとに全部別バージョンがコンパイルされてコードが巨大になることが回避されます。

これは、標準の演算子が場合によってはまずい、などの状況を回避するためにも使えます。たとえば、上記の maxbuf で char*を渡すと、<がアドレスの大小比較になるため、「文字列値が最大のものを得る」という意図した結果が得られません。そこで次のように、最大には別の比較関数 lt を使用することにして、これをテンプレート特殊化により切り替えることで、文字列比較には strcmp を使うようにさせられます。

```
//maxbuf3.cc
#include <iostream>
#include <cstring>
using namespace std;

template<typename T> bool lt(T x, T y) {
  return x < y;
}
template<> bool lt(char* x, char* y) {
  return strcmp(x, y) < 0;
}

template<typename T> class maxbuf {
  T max;
public:
  maxbuf(T x) : max(x) { }
  void put(T x) { if(lt(max, x)) max = x; }
  T get() { return max; }
};

int main() {
  int a1[] = { 5, 9, 2, 7, 6 };
  maxbuf<int> m1(a1[0]);
  for(int i = 1; i < 5; ++i) m1.put(a1[i]);
  cout << "a1: " << m1.get() << '\n';
  char* a2[] = { "this", "by", "foo", "z", "x" };
  maxbuf<char*> m2(a2[0]);
  for(int i = 1; i < 5; ++i) m2.put(a2[i]);
  cout << "a2: " << m2.get() << '\n';
}
```

演習 12 上の例題をそのまま動かせ。できたら、T*用が void*用を用いて実装されるような自前のベクターを実装してみよ。

3.7 テンプレートによる mixin クラス

mixin クラスとは、それ自体はインスタンスを生成する能力を持たず、他のクラスに継承を用いて「混ぜる」ことで機能を付加することを目的としたクラスを言い、「抽象サブクラス」と呼ばれることもあります。歴史的には Lisp Machine Lisp のオブジェクト指向機能 Flavors において多用されたことから広まりました。Flavors では、ウィンドウシステムにおいて基本的なウィンドウに「メニュー

を付加する」「タイトルバーを付加する」などの mixin を混ぜることでさまざまな種類の窓を作るようになっていました。

mixin ではメインのクラスに機能を追加して混ぜるため、多重継承 (親が複数ある継承) が前提となります。たとえば C++には多重継承がありますから、C++でなら mixin ができそうです。しかし、C++で単純に多重継承を使っても駄目なのです。たとえば、次のようなクラスを考えてみてください。前者は銀行口座のようなもの、後者は最大値バッファです。

```
class balance {
  int value;
public:
  balance() : value(0) { }
  void update(int v) { value += v; }
  int getValue() { return value; }
};

class maxbuf {
  int max;
public:
  maxbuf() : max(0) { }
  void update(int v) { if(v>max) max=v; }
  int getValue() { return max; }
};
```

これらに追加する機能として、update() の呼ばれた回数を数える count、更新履歴を記録する history という 2 つの mixin を考えます。

```
class count {
  int num;
public:
  count() : num(0) { }
  void update(int v) { ++num; update(v); }
  int getCount() { return num; }
};

class history {
  int record[1000], nrecs;
public:
  history() : nrecs(0) { }
  void update(int v) {
    record[nrecs++] = getValue(); update(v);
  }
  void showHistory() {
    for(int i = 0; i < nrecs; ++i)
      cout << ' ' << record[i];
    cout << '\n';
  }
};
```

これらを組み合わせて、たとえば次のように書くことで「計数と履歴の機能を持った口座」を作りたいわけです。

```
class MyClass : public balance, count, history {
};
```

Flavors は型が無い言語なのでこれで OK でしたが、C++ではメソッドの宣言を書かないといけないので、このように「空っぽ」にはできません。そもそも、count や history の側で親クラスを指定しないと、親クラスのメソッドのオーバライドもできません。こうして見ると全くダメそうに思えますが、実はできるのです。次のを見てください。

```
//mixin1.cc
#include <iostream>
using namespace std;

class balance {
  int value;
public:
  balance() : value(0) { }
  void update(int v) { value += v; }
  int getValue() { return value; }
};

class maxbuf {
  int max;
public:
  maxbuf() : max(0) { }
  void update(int v) { if(v>max) max=v; }
  int getValue() { return max; }
};

template<typename T> class count : public T {
  int num;
public:
  count() : num(0) { }
  void update(int v) { ++num; T::update(v); }
  int getCount() { return num; }
};

template<typename T> class history : public T {
  int record[1000], nrecs;
public:
  history() : nrecs(0) { }
  void update(int v) {
    record[nrecs++] = T::getValue();
    T::update(v);
  }
  void showHistory() {
    for(int i = 0; i < nrecs; ++i)
      cout << ' ' << record[i];
    cout << '\n';
  }
```

```
};

int main() {
  int a[] = { 5, 9, 2, 7, 6 };
  history< count< balance > > c1;
  count< history < maxbuf > > c2;
  for(int i = 0; i < 5; ++i) {
    c1.update(a[i]); c2.update(a[i]);
  }
  cout << "c1: " << c1.getCount(); c1.showHistory();
  cout << "c2: " << c2.getCount(); c2.showHistory();
}
```

つまり、mixin クラスはテンプレートとして、親クラスを型パラメタで受け取り、そのクラスのサブクラスにすればいいわけです。「置き換えてコンパイルする」方式の強力さが分かりますね。

演習 13 上記の例をそのまま打ち込んで動かせ。動いたら、自分でも新たな mixin を考案して追加してみよ。

3.8 テンプレートメタプログラミング

メタプログラミングとはひとこで言えば、「通常実行するようなコードを加工するようなプログラミング」のことです。Lisp のような eval を持つ言語ではこれはごく簡単なことですが (そして Lisp にはそれを積極活用するマクロ機能がついていますが)、型検査をする言語では普通はできません。

しかしここまで見て来たように、C++ではテンプレートの展開はかなり強力かつ自由なコード加工が可能になっています。それを用いてメタプログラミングを行うのがテンプレートメタプログラミングです。テンプレートの実体化機能とメタプログラミングの関係を整理すると、次のようになります。

- テンプレートの展開はコンパイル時に「必要になったところで」行われることになっている (遅延評価)。このため、無限に展開されそうに見えるものでも必要な範囲までしか展開されないで済む。
- ここまでテンプレートパラメタには型ばかり書いてきたが、整数もパラメタに書け、整数の計算はテンプレート実体化のときに計算される。これによって整数で数えながら展開することが可能。
- テンプレートの特殊化を利用することで、「パラメタが 0 ならこちら」のように基本ケースを書いて枝分かれさせることができる。これを使うことで、再帰的な展開が可能になる。

では基本的な例として、階乗を「コンパイル時に」計算してみましょう。

```
//factorial1.cc
#include <iostream>
using namespace std;

template<int n> class Fact {
public:
  enum { RET = Fact<n-1>::RET * n };
};
template<> class Fact<0> {
public:
  enum { RET = 1 };
```

```
};

int main() {
  cout << Fact<5>::RET << '\n';
}
```

つまり、Fact というテンプレートクラスは RET という数値定数を持っていて、Fact<1>::RET は 1、Fact<2>::RET は 2、Fact<3>::RET は 6、等々となっているわけです。テンプレートは直接は値を返せないけれど、数値定数は定義できることを利用して値を返させているわけです。

もっと込み入った例として、組合せの数を定義してみましょう。さらに、計算時間も計測してみます。

```
//factorial2.cc
#include <iostream>
#include <ctime>
using namespace std;

template<int n> class Fact {
public:
  enum { RET = Fact<n-1>::RET * n };
};
template<> class Fact<0> {
public:
  enum { RET = 1 };
};

template<int n, int k> class Comb {
  enum { a = Fact<n>::RET,
         b = Fact<k>::RET * Fact<n-k>::RET };
public:
  enum { RET = a / b };
};

int fact(int n) {
  return (n<=1) ? 1 : n*fact(n-1);
}
int comb(int n, int k) {
  return fact(n)/(fact(k)*fact(n-k));
}

int main() {
  cout << Comb<10,4>::RET << '\n';
  cout << comb(10,4) << '\n';
  int i, v1 = 0, v2 = 0;
  clock_t t1 = clock();
  for(i=0;i<10000000;++i) v1 = Comb<10,4>::RET;
  clock_t t2 = clock();
  for(i=0;i<10000000;++i) v2 = comb(10,4);
  clock_t t3 = clock();
```

```
  cout << double(t2-t1)/CLOCKS_PER_SEC << '\n';
  cout << double(t3-t2)/CLOCKS_PER_SEC << '\n';
}
```

これを実行したところ、次のようになりました。値はどちらも同じですが (当り前)、実行時間は 100 倍以上違うわけです。テンプレートメタプログラミングの結果は実行時にはただの定数になっているので当り前ではありますが。

```
% a.out
210
210
0.015625
2.32812
```

演習 14 上の例を動かして確認せよ。次に、次の公式通りにフィボナッチ数を計算するコードを通常の再帰関数とテンプレートメタプログラミングの両方で作成し、時間を比較せよ。

$$fib(n) = \begin{cases} 1 & (n = 0 \; or \; n = 1) \\ fib(n-1) + fib(n-2) & (otherwise) \end{cases}$$

実は、テンプレートを使って「IF 文」や「WHILE 文」を作ることもできます。それを利用すると、階乗のテンプレートも素直 (?) に書けます。ハテナがついているのは、テンプレートでは「末尾再帰に」書かなければならないため、そのような変形が必要ということです。

```
//factorial3.cc
#include <iostream>
using namespace std;

template<bool cond, class Then, class Else> class IF {
  public:
    typedef Then RET;
};
template<class Then, class Else> class IF<false,Then,Else> {
  public:
    typedef Else RET;
};
template<int m> class FactThen {
public:
  enum { RET = m };
};
template<int n, int m> class Fact {
  typedef typename
    IF< (n<=0),FactThen<m>,Fact<n-1,m*n> >::RET result;
public:
  enum { RET = result::RET };
};

int main() {
  cout << Fact<5,1>::RET << '\n';
}
```

ただし C++の処理系によってはテンプレートの部分特殊化 (条件部だけの特殊化) をサポートしていないものもあるそうで、その場合は次のような分かりにくい書き直しが必要になります。

```
//factorial4.cc
#include <iostream>
using namespace std;

class SelThen {
public:
  template<class Then, class Else> class Res {
  public:
    typedef Then RET;
  };
};
class SelElse {
public:
  template<class Then, class Else> class Res {
  public:
    typedef Else RET;
  };
};
template<bool cond> class Select {
public:
  typedef SelThen RET;
};
template<> class Select<false> {
public:
  typedef SelElse RET;
};
template<bool cond, class Then, class Else> class IF {
  typedef typename Select<cond>::RET which;
public:
  typedef typename
    which::template Res<Then,Else>::RET RET;
};
template<int m> class FactThen {
public:
  enum { RET = m };
};
template<int n, int m> class Fact {
  typedef typename
    IF< (n<=0),FactThen<m>,Fact<n-1,m*n> >::RET result;
public:
  enum { RET = result::RET };
};
int main() {
  cout << Fact<5,1>::RET << '\n';
}
```

ここまでは単一の値を計算するテンプレートメタプログラミングでしたが、もっと大量のコードを生成することもできます。たとえば、「x*x*x*…*x」のように沢山掛け算を必要とする場合、それを正しく数えるのは大変ですよね。かといってループにしたらループ制御のオーバヘッドが掛かります。そこで、次のようにして掛け算の列をテンプレートから生成させることもできます。

```
//multiply1.cc
#include <iostream>
using namespace std;

template<int n> double pow(const double& x) {
    return pow<n-1>(x) * x;
}
template<> double pow<1>(const double& x) {
    return x;
}

int main() {
  double x = 2.0;
  cout << pow<10>(x) << '\n';
}
```

ただし、テンプレートの再帰数には制限があるので、これであまり大量の掛け算は生成できません(再帰が深くならないように工夫すればできる)。あと、コードが肥大するので、この方法が速いかどうかは必ずしも分かりません。

演習 15 上の方法で生成した掛け算とループによる掛け算の速度比較をしてみよ。