C++とJavaの設計思想と機能の比較
～筑波EDP-skillup:システム開発技術を考える～

久野　靖

2007.9.26

0　はじめに

□ 久野の専門分野…

- プログラミング言語 --- オブジェクト指向言語、並行/並列言語、教育用言語
- システムソフトウェア ---「コンピュータを使うためのソフト」→「何をどう作ったら役に立つ?」(言語処理系、コンパイラもこれに含まれる)
- ユーザインタフェース --- コンピュータと人間の接点、対話方式など
- 情報教育、情報技術教育 --- ICTの何を、どこまで、どのように教えるか? (中学・高校、大学、社会人…)

□ 本日の内容構成

- 1章 --- プログラミング言語の体系とC++、Javaの位置づけ
- 2章 --- C++言語の由来とあらまし
- 3章 --- Java言語の由来とあらまし
- 4章 --- C++とJavaの違うところ←機能←設計思想
- ----- 以下は時間的余裕と皆様の希望に応じて -----
- 5章 --- 実行時の動的処理と注記 (Javaが持つ特色)
- 6章 --- 総称的プログラミング (主にC++、一部Java)
- 7章 --- テンプレートメタプログラミング (C++)
- 8章 --- テンプレートパラメタの仕様記述機構 (C++、研究レベル)

1　プログラミング言語の体系

1.1　ソフトウェア開発とプログラミング言語

□ ソフトウェア開発のあり方とプログラミング言語は密接に関係

- COBOL: データ中心、定型的事務処理
- FORTRAN: 科学技術計算
- C: ネットワークなどシステム機能を用いる処理

□ 近年のソフトウェア開発の主流の1つ→Webアプリケーション

- サーバ側開発言語→PHP、Perlなど
- クライアント側言語→JavaScript (連携:Ajax)

□ スクリプト言語の利点→弱い型、少量の記述で多くの機能、短い開発サイクル

- その反面、大規模な開発には記述面、性能面で不利

□ 大規模ソフトウェア開発に必要な機能

- 強い型検査→コンパイラによる整合性検査
- モジュール→名前の衝突等を防ぐ、部品化を可能にする
- オブジェクト指向→オブジェクト指向はなぜ必要?

1.2　オブジェクト指向言語

□ オブジェクト指向とは…

- プログラムが扱う対象を「もの」(オブジェクト) と捉える「考え方」
- オブジェクトの種類ごとに「属性」「操作 (動作)」が付随
- 人間は「もの」の世界に生きている→「もの」中心が考えやすい

- 考えやすい (把握しやすい) ことはプログラミング言語にとって重要 (「考えやすい」言語の方が同じ努力であればより複雑なものまで作れる)

□ 動的分配…オブジェクト指向言語の重要な機能

- 例:　道路交通「自動車. 前進 ()」「自転車. 前進 ()」等の操作があるとする
- 「x. 前進 ()」で「x に現在入っているオブジェクトの前進」が呼ばれる (実行時に定まる) →動的分配の機能
- OO 以前:「if(x が自動車) 自動車の前進 (x); elsif(x が自転車) 自転車の前進 (x); elsif...」をやっていた (コードが長く読みづらい)
- 動的分配のため x が「何であっても」同じ 1 つのコードでよい (総称的)

□ クラス方式のオブジェクト指向言語…オブジェクトの種別毎の性質を「クラス」単位で記述

- クラス方式の他にプロトタイプ方式があるがクラス方式が主流
- クラス間の「継承」により短い記述で類似したクラスが作れる (差分プログラミング) ←現在ではあまり推奨されない
- クラスを「型」として扱うことで強い型検査を適用してコードの誤り「の一部」をコンパイル時検査できる

□ クラスで記述するもの (細かいものは略)

- 各オブジェクトが持つデータ (=インスタンス変数)
- 各オブジェクトが持つ操作 (=メソッド)

□ 例:　Point (2 次元平面上の点) の場合…

- データ:　double x, y; ← X 座標と Y 座標
- 操作:　たとえば移動と XY 座標の取得--- p1.moveBy(dx, dy)、p1.getX()、p1.getY()

1.3　オブジェクト指向言語の歴史

□ Simula (1960's) --- 最初のオブジェクト指向言語

- イベントシミュレーションの記述のため、「もの」を言語上で扱うためにオブジェクト指向が発明された
- クラス、インスタンス、継承、動的分配など基本的な「道具」はこの言語で既に揃っていた
- ヨーロッパ発の言語であり、あまり普及しないまま忘れられる

□ Smalltalk-80 (1980) --- オブジェクト指向の再発見

- 世界初のワークステーション (個人用計算機) である Alto 上で稼働し、ウィンドウ、マウス、メニュー、GUI 部品を駆使 →センセーショナルな登場
- 当時のプログラミング言語は「モジュール」「抽象データ型」の段階→オブジェクト指向を取り入れることの可能性を提示
- 言語自体は Lisp などに近い型の無い言語、実用のソフト開発にはあまり使われなかった
- この後、「実用言語にオブジェクト指向導入」の開発競争 → Objective-C (NeXT)、COB (日本 IBM) など

□ C++

- Bjane Stroustroup (ベル研究所) が「C with class」として開発
- 効率が良い (C 言語に劣らない性能が目標) →支持を集める→ C 言語に基づいたオブジェクト指向言語として支配的な立場

□ Java

- C++(高速だが危険なこともできる) のアンチテーゼとして Sun Microsystems 社で開発
- 安全、互換性、標準ライブラリの充実などで支持を集める

□ その他… Eiffel など (ヨーロッパ系はどうしてもマイナー)

□ C++/Java のコンパイル文化とは別に、スクリプト言語文化も

- Self --- 研究用言語、クラスを使わない (プロトタイプ方式)
- JavaScript --- ブラウザ上のスクリプト言語として開発され普及した。これもプロトタイプ方式
- Perl --- 現代的スクリプト言語の元祖。Perl はもともとはオブジェクト指向言語でないが、Perl5 でオブジェクト指向を導入 (PHP なども同様)
- Python --- Perl の「後付けオブジェクト指向」に対抗して最初からオブジェクト指向を組み込んだスクリプト言語として普及
- Ruby --- Python と同様だが「書きやすさ」指向のカルチャーに特色。日本人まつもとゆきひろ氏が開

発。日本初で世界に広まった言語というのはこれが始めて。

1.4　この節のまとめ

□ ソフトウェア開発のあり方とプログラミング言語は関連

- スクリプト言語系とコンパイル言語系

□ 今日ではオブジェクト指向言語が広く使われている

- モジュール (部品化)、継承、動的分配 (総称性)

□ オブジェクト指向言語の歴史

- Simula → Smalltalk →コンパイル系 (C++、Java)、スクリプト系 (Perl、PHP、Python、Ruby、JavaScript)

2　C++言語

2.1　C++の由来とあらまし

□ Bjane Stroustrup によって、「C with class」として始まる

- C からスムーズに移行できるオブジェクト指向言語として普及
- C に (ほぼ) 上位互換な言語。型検査とかは厳しくなっている
- +α されているもの…「オブジェクト指向」「例外」「テンプレート」など
- C の「裸のマシンがそのまま使える」に「よい構造化ができる」を追加

□ C++の設計思想…

- フルスピードで動く (C に負けない)
- そのために足枷となるものは言語に導入しない
- どのようなプログラミングスタイルでもサポートする
- プログラマが分かっていてやることは禁止しない
- ユーザ定義型も組み込み型と同様な見た目で使える

□ 例題: 2 つの数を読み込んで足す

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

int main(void) {
  int x; cout << "x? "; cin >> x;
  int y; cout << "y? "; cin >> y;
  cout << "x+y = " << (x+y) << '\n';
}
```

- 「cin」「cout」は入出力ストリーム
- 「>>」「<<」(もともとはシフト演算子) を演算子定義により入出力演算として使っている
- 「>>」の結果はまたストリームなので連続して使える
- 多重定義により、「<<」を文字列用、整数用、実数用と多数用意している

2.2　C++のオブジェクト指向機能

□ オブジェクト指向機能の基本要件 --- インスタンス変数とメソッド

- 構造体 (struct) の拡張として class という構文を作った。
- class ではメソッド定義もできるようにした。

```
struct point {
  double x, y;
};            ←ここに「;」が必要!

class Point {
  double x, y;
public:  ←以下「クラス外から見える」
  Point(double x0,y0) { ←初期化
    x = x0; y = y0;
  }
  void moveBy(double dx,dy) {
    x += dx; y += dy;
  }
  double getX() { return x; }
  double getY() { return y; }
};            ←ここに「;」が必要!
```

- ここではクラスの宣言に定義も入れてあるが、実際には宣言と定義を分離して記述し、宣言をヘッダファイルに置く
- C++では「virtual」と指定していないメソッドは動的分配を行わない (その分効率がよい)
- 実際にはこのような使い方 (抽象データ型) で済む場合も多い

2.3　例題:分数電卓 (C++)

□ 分数をあらわすクラスを用意する。方針としては 2 つの数 a、b で分数 a/b を表す。まず宣言部。

```
class Rational { // a/b
  int a, b;
  int gcd(int x, int y);
public:
  Rational(int x);
  Rational(int x, int y);
  Rational operator+(const Rational& r) const;
  Rational operator-(const Rational& r) const;
```

```
  Rational operator*(const Rational& r) const;
  Rational operator/(const Rational& r) const;
  friend ostream& operator<<(ostream& o, Rational& r);
  friend istream& operator>>(istream& i, Rational& r);
};
```

- private 部分には変数と補助関数 GCD(最大公約数)。
- コンストラクタは値 1 つ (分母 1) と 2 つ。
- 四則は演算子。
- 分母 0 になる場合は NaN (Not a Number)。
- 入出力は単独関数として定義した演算子「>>」と「<<」。(なぜ単独関数かというと、クラス istream や ostream に後からメソッドの追加はできないから。) 第 1 引数は入出力ストリーム。firend 宣言は「この関数は特別にこのクラス定義の中身 (private 部分) にアクセスできる」という意味。

□ 実装部分。

```
int Rational::gcd(int x, int y) {
  while(x != y) if(x > y) x -= y; else y -= x;
  return x;
}
Rational::Rational(int x) { a = x; b = 1; }
Rational::Rational(int x, int y) {
  if(y == 0) { a = b = 0; return; }
  if(x == 0) { a = 0; b = 1; return; }
  if(y < 0) { x = -x; y = -y; }
  if(x == 0) {
    a = x; b = y;
  } else if(x > 0) {
    a = x / gcd(x,y); b = y / gcd(x,y);
  } else {
    a = x / gcd(-x,y); b = y / gcd(-x,y);
  }
}
Rational Rational::operator+(const Rational& r) const {
  return Rational(a*r.b + r.a*b, r.b*b);
}
Rational Rational::operator-(const Rational& r) const {
  return Rational(a*r.b - r.a*b, r.b*b);
}
Rational Rational::operator*(const Rational& r) const {
  return Rational(a*r.a, r.b*b);
}
Rational Rational::operator/(const Rational& r) const {
  return Rational(a*r.b, r.a*b);
}
ostream& operator<<(ostream& o, Rational& r) {
  if(r.b == 0) o << "NaN";
  else o << r.a << '/' << r.b;
  return o;
}
istream& operator>>(istream& i, Rational& r) {
  int a, b; i >> a >> b;
  r = Rational(a, b); return i;
}
```

- 面倒な「約分」の計算はコンストラクタで。

□ メイン部分。

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

// ここにヘッダ部分
// ここに実装部分

int main(void) {
  Rational r(1), x(0);
  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;
case '/': cin>>x; r = r / x; break;
default: continue;
    }
    cout << r << '\n';
  }
}
```

- 1 文字読んでコマンド文字によって動作を切替え。例:

```
? = 1 3   ← 1/3 を入れる
1/3       ←現在値
? + 1 6   ← 1/6 を足す
1/2       ←現在値
q         ←おしまい
```

□ ここまでで分かる特徴…

- C 言語を拡張してオブジェクト指向としている
- 効率が悪くなる機能は使わないようにできる (さらにメソッドに inline と指定するとその場展開→メソッド呼び出しのコストも削減可能)
- 「*」などの演算子も定義して差し替えられる

2.4 この節のまとめ

□ C++ --- C 言語に上位互換なオブジェクト指向言語

□ フルスピードで動く (C に負けない) ことが目標

□ ユーザ定義型も見た目で既存の型と同等に見える

3 Java 言語

3.1 Java の由来とあらまし

☐ Java --- 「C++よりも安全なオブジェクト指向言語」として SunMicrosystems 社で開発された

- 当初 Web ブラウザの上で動く「アプレット」のための言語として普及
- この場合の「安全」…プログラムが悪さをできないような防壁がある (例: ファイルの読み書きができない、勝手にネットワーク接続ができない、など)
- 現在では通常のソフトウェア開発用言語として使われている
- C(や C++) とは制御構造の構文が似ているだけであとは別物

☐ Java の設計思想…

- C++の危険さ、複雑さを減少させ単純さを求める
- C や C++との互換性は捨てたので言語仕様は単純化できた
- 安全さを第一とし、危険なことはどうやってもできない
- すべてのオブジェクトはヒープ上に割り当て、ごみ集めを行う
- Smalltalk 的、伝統的なオブジェクト指向言語ふう
- これらの代償として見た目は長く、動作は遅くなりがち

☐ 例題: 2 つの数を読み込んで足す

```
import java.util.*;

public class Sample31 {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("x? ");
    int x = sc.nextInt();
    System.out.print("y? ");
    int y = sc.nextInt();
    System.out.println("x+y = " + (x+y));
  }
}
```

- C++のような「演算子の駆使」はない
- すべて「オブジェクト. メソッド (引数…)」で普通に書く

3.2 Java のオブジェクト指向機能

☐ オブジェクト指向言語の基本要件 --- インスタンス変数とメソッド

- Java は最初から新しい言語として設計→ class という構造を作り、その中にインスタンス変数定義、メソッド定義を入れる

```
class Point {
  double x, y;
  public Point(double x0,y0) {
    x = x0; y = y0;
  }
  public void moveBy(double dx,dy) {
    x += dx; y += dy;
  }
  public double getX() { return x; }
  public double getY() { return y; }
}                   ←ここに「;」不要
```

- public/private 等は各宣言に付加
- このレベルだと C++とほどんど変わらない

3.3 例題:分数電卓 (Java)

☐ C++版と同じ方針で実装。こちらはメイン側から。

```
import java.util.*;

public class Sample32 {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    Rational r = new Rational(1);
    while(true) {
      System.out.print("? ");
      String s = sc.next();
      if(s.equals("q")) return;
      int a = sc.nextInt(), b = sc.nextInt();
      Rational x = new Rational(a, b);
      switch(s.charAt(0)) {
case '=': r = x; break;
case '+': r = r.add(x); break;
case '-': r = r.sub(x); break;
case '*': r = r.mul(x); break;
case '/': r = r.div(x); break;
default: continue;
      }
      System.out.println(": " + r);
    }
  }
}
```

- Scanner の next() は「次の空白等で区切られた 1 かたまり」を返す。
- 動かし方は C++版と同じ。

☐ Rational クラスは内容的にはほぼ同じ。

```
class Rational {
  int a = 0, b = 0; // b == 0 -> Not a Number
  public Rational(int x) { a = x; b = 1; }
  public Rational(int x, int y) {
    if(y == 0) { return; }
    if(x == 0) { a = 0; b = 1; return; }
    if(y < 0) { x = -x; y = -y; }
    if(x == 0) {
      a = x; b = y;
    } else if(x > 0) {
      a = x / gcd(x,y); b = y / gcd(x,y);
    } else {
      a = x / gcd(-x,y); b = y / gcd(-x,y);
    }
  }
  public Rational add(Rational r) {
    return new Rational(a*r.b + r.a*b, r.b*b);
  }
  public Rational sub(Rational r) {
    return new Rational(a*r.b - r.a*b, r.b*b);
  }
  public Rational mul(Rational r) {
    return new Rational(a*r.a, r.b*b);
  }
  public Rational div(Rational r) {
    return new Rational(a*r.b, r.a*b);
  }
  public String toString() {
    if(b == 0) return "Nan"; else return a + "/" + b;
  }
  private int gcd(int x, int y) {
    while(x != y) if(x > y) x -= y; else y -= x;
    return x;
  }
}
```

- 演算子定義はないので適当なメソッド名をつける。
- 出力も演算子はないので toString() で文字列への変換方法を定める (文字列が必要なときは自動的にこれが呼ばれる)。

3.4 この節のまとめ

□ Java --- C++の問題点である、「安全でない操作もプログラマが分かっていてやるなら許す」ことをやめるために新規に設計

□ 新しい言語として作ったため機能を整理できた (今はその後追加した機能があって複雑になっている面も)

□ 書き方は標準的で平穏。その分長くなる面も

4 C++と Java の違い

4.1 プログラム構造

□ C++はクラスと単独関数 (と struct と union と) がある。Java はクラスのみ (と、インタフェース)。

□ 分割コンパイル→ C++はやはりちょっと古い。Java はよりスマート。

- C++はクラス定義+各メンバの定義。クラス定義はデータの定義+メソッドインタフェースの定義。ヘッダファイルに入れる (そうしないとオブジェクトの大きさが分からないので使う側がコンパイルできない)。テンプレートだとさらにそういう感じ。
- Java は 1 つのクラス定義に全部書く。インタフェース情報は.class ファイルに書き出され、分割コンパイル時はコンパイラがこれを参照する。
- 名前空間の分割 (同じ名前のクラスが衝突することを避ける) ← Java はパッケージ機能、C++は名前空間機能。扱いは違うができることは同程度

□ Java の入れ子クラス --- クラスの中にクラスが書ける

- static な入れ子クラス --- 普通のクラスが「外から見えない」ように作れる (外から見せてもいい)
- static でない入れ子クラス (内部クラス) --- 外側のクラスのインスタンス変数群を参照できる
- 無名内部クラス --- 内部クラスを使うのが 1 箇所だけの場合に、いちいち名前をつけなくても済む仕組み
- 全体として… Java では「どんどんクラスを作って利用」→オブジェクト指向的?

4.2 値の扱い

□ 値、オブジェクト、ポインタ

- C++も Java も「値」と「オブジェクト」を区別している。
- 値→数値、文字、論理値。単純なビット表現。
- オブジェクト→ C++でも Java でもクラスにより定義し、内部的に複雑な構造を持つ (持てる)
- ポインタ型/参照型→ C++にのみ存在。Java ではオブジェクト値は全部ポインタのようなもの (後でまた扱う)
- C++では「==」などの演算子を定義できる→オブジェクトでも演算子を使って書く。

- Java では「値→演算子」「オブジェクト→メソッド」ときっぱり区分されている。「v1 == v2」(値)、「o1.equals(o2)」。もしオブジェクトに「o1 == o1」を使うと「ポインタ比較 (同一のオブジェクトかどうかを調べる)」になる。

□ Java の包囲クラス

- C++では文字列→数値変換などは (C に由来する)atoi などの関数利用。
- Java ではそのようなものの置き場所として「Integer」「Double」などのクラス (包囲クラス、wrapper class) がある

```
int x = Integer.parseInt("123");
```

- これらのクラスは int や double の値を「オブジェクトにくるんで扱う」ためにも使われる (包囲クラスという名前の由来)

```
Integer i = new Integer(123);
int k = i.intValue();
```

- なぜクラスにするかというと、Java では「オブジェクトでないと不便なこと」が色々ある (例: Object 型の変数にはオブジェクトなら何でも入れられるが、値はオブジェクトでないので入れられない
- C++は (テンプレート機構を使って) オブジェクトでも値でも見た目は変わり無く扱えるようにするので包囲クラスは不要

4.3 型情報を利用した自動型変換

□ すべての式や変数に型がある→それらに「不一致」があることも

- 数値型など基本型の間では一定範囲で「自動変換」が提供されている
- オブジェクト型どうしでは「上位の変数に下位のオブジェクトが入れられる」という規則が「原則」
- オブジェクトと基本型→相互変換行わないのが「原則」
- 便利さのためのこの「原則」をやめて変換を可能とする場合がある

□ Java の AutoBoxing/Unboxing

- Java では「1」(値) と「new Integer(1)」のように基本型とそれに対応するオブジェクト型の両方を用意→相互に変換することが多い
- Boxing --- 「箱に入れる」: 値をオブジェクトにする
- Unboxing --- 「箱から出す」: オブジェクトから値を取り出す

```
Integer i1 = new Integer(1);   ← boxing
int j1 = i1.intValue();       ← unboxing
```

- JDK 1.5 以降でこれらを「自動化」:boxing / unboxing が必要になったときは自動的に行う (演算などの場合も)

```
Integer i1 = 100, i2 = 200; ← auto boxing
int j1 = i1, j2 = i1+i2;  ← auto unboxing
int j3 = i2++;  ← auto unboxing+boxing
```

- しかし「==」「!=」はオブジェクトどうしが等しいという意味がもともとあるので auto unbox されないことに注意
- メソッド呼び出しのために autoboxing されることはない。「3.toString()」はダメ
- 整数だと思っているのが実は Integer オブジェクトだったりすると、見た目で感じるのとまったく違うオーバヘッドが掛かることがある→「どの型なのか」をきちんと把握しておかないと危険

```
public class Sample41 {
  public static void main(String[] args) {
//  int k = 0;
    Integer k = 0;
    for(int i = 0; i < 1000000000; ++i)
      k = k + i;
  }
}
```

□ C++の自動変換機能

- 多くの場合「変換」しなくても演算子オーバロードで済む
- コンストラクタを定義することで自動型変換が可能になる

```
class C1 {
public:
  C1(&int i) { ... }
  ...
}

C1 x = 5; // C1 x = C1(5) と同じ
```

- 代入時も別扱いしたければ代入演算子も定義
- 基本型や既にあるクラスを「変換先」にしたい…コンストラクタは定義できない→その場合は「型変換演算子」を定義 (「operator T()」…自分クラスの値を T 型に変換)

```
int i = x; // x は C1 型
...
class C1 {
public:
  operator int() { return this.size; }
  ...
}
```

- これらの複数の機能のどれを使うか→言語仕様で規定 (複雑)

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

class Test {
  int val;
public:
  Test() { val = 9; }
//Test(int i) { val = i+1; }
//void operator=(const int& i) { val = i+2; }
//operator int() const { return val+3; }
  int value() { return val; }
};

int main() {
  Test x;
//Test x = 10;
//x = 20;
  cout << x.value() << "\n";
//cout << (x+0) << "\n";
}
```

4.4 動的な型の扱い

□ たとえば、「何でも出力しちゃう関数」を作る:

- Javaの場合→「Object型の変数には任意のオブジェクトが入れられる」(Objectはすべてのクラスの祖先だから)ことを利用 + AutoBoxing

```
public class Sample42 {
  public static void main(String[] args) {
    p(1);
    p("abc");
    p(new int[]{1, 2, 3});
  }
  public static void p(Object o) {
    System.out.println(o);   ←文字列変換は動的分配
  }
}
```

- C++の場合→テンプレート機能を使って任意の型Tを受け取り、それを総称的に扱うことができることを利用

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

template<typename T> void p(T x) {
  cout << x << "\n";   ←「<<」の多重定義を利用
}
int main() {
  p(1);
  p("abc");
  int a[] = {1, 2, 3};
  p(a);
}
```

□ 一見似ているようだが全然違う

- Javaでは実行時の動的分配を用いて各オブジェクト毎の処理をする→コンパイル時に存在していないクラスでもOK(柔軟性)
- C++ではコンパイル時に多重定義のどれが使われるかを決定→コンパイル時にすべて処理される(高速)
- この「実行時のJava」「コンパイル時のC++」という違いは言語の設計思想の違いによっている

□ 親クラスの変数に入れたものを「元に戻す」必要性

```
Animal a = new Dog(...) // or Cat
...
Dog d = (Dog)a;

Animal *a = new Dog(...) // or Cat
...
Dog *d = dynamic_cast<Dog>(a);
```

- aに入っているものがDogならDogに戻せるが、CatならDogに戻せてはいけない→エラーになる
- このために型の情報が実行時に保持されている必要

□ Javaの場合(上の例)

- すべてのオブジェクトにはクラス情報が付随
- オブジェクトに対するキャストはクラス情報によりチェック
- ほかに「if(a instanceof Cat) ...」などの演算子もある
- もっと自由に型の情報そのものを扱う→自己反映機能(後述)

□ C++の場合(下の例)

- クラス情報が付随するのは「virtualメソッド(動的分配を行うメソッド)を持つクラスのオブジェクト」のみ←効率のため
- そのようなオブジェクトに限りdynamic_castが使える
- それ以外のキャスト(型変換)はコンパイル時の扱いのみ
- 実行時に情報がない場合のキャスト→static_cast<T>(値)
- constのある型からconstを外す→const_cast<T>(値)
- 何を何にでも変換できるキャスト→reinterpret_cast<T>(値)←これまでの「(型)値」というキャストと同等。これまでのキャストは危険な上に目立たないのでやめさせたいという考え

4.5　多重継承とインタフェース

□ 多重継承 --- あるクラスが複数の親クラスを持つこと

- 用途 1:　1 つのオブジェクトがいくつかの側面を併せ持つ場合。たとえば「馬」は「交通手段」の一種であるし「動物」の一種でもある。
- 用途 2:　あるオブジェクトを実装するのに複数のオブジェクトの機能を取り込んで来たい場合。←干渉などの問題がある…

□ 継承そのものが「総称的に扱う手段を提供 (用途 1)」と「実装を取り込んで来て短く記述 (用途 2)」の 2 側面を持つ

□ C++ --- 多重継承を実装

- 1 つのクラスに複数の親クラスを指定できる。そのクラスの実体は複数の親クラスのデータを連結したものに独自のデータを付加した形になる (一部例外あり)
- 干渉の問題は解決されていない (プログラマの責任)

□ Java --- 多重継承は除外し、代わりにインタフェース機能

- インタフェース --- オブジェクトが持つメソッド名とその引数や返値の情報 (メソッドシグニチャ) を集めたもの

```
interface Drawable {
  public void draw(Graphics g); ←画面に描ける
}
```

- 1 つのクラスは任意個数のインタフェースに従うことができる
- インタフェース型の変数にはそのインタフェースに従うクラスのインスタンスであればどれでも入れることができる (メソッド呼び出し時に動的分配)

```
class Cirlce implements Drawable { ... }
Drawable d = new Circle(...);
...
d.draw(g);
```

□ 両者を比較すると…

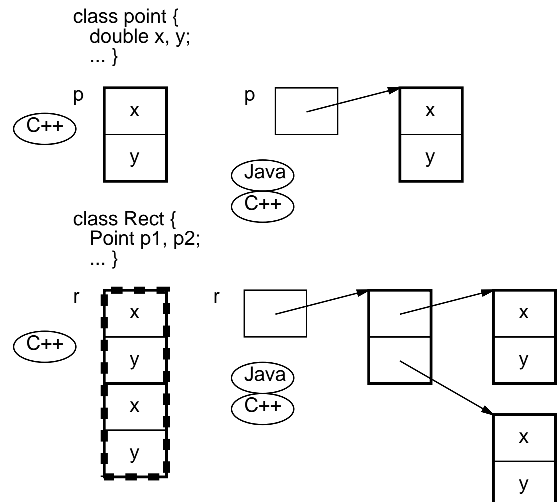
- Java は「用途 1」が主な目的、C++は「用途 2」が (ある程度) 目的
- プログラムの構造化という点では「用途 1」が重要。少なくとも「用途 1」と「用途 2」が分離できることが重要→ Java の方が新しい言語だからそれなり
- C++でインタフェースと同じことをする場合は、「変数もメソッド内容も定義しない抽象クラス」を作成する

```
abstract class Drawable {
public:
  void draw(Graphics g) = 0;
};
```

4.6　記憶域の割り当て

□ C++と Java で一番違うところは:

- C++ではオブジェクトは「配列や実行スタック上に直接その記憶域を配置できる。外に配置するときはポインタを使う」
- Java ではオブジェクトは「すべてヒープ上のオブジェクト。オブジェクトはすべてポインタで扱う」(Java の用語では「参照」) →最も重要な単純化。動的データ構造や多態のためにはどのみち参照が必要。そのため、すべてポインタに統一している。そのために遅い面も。



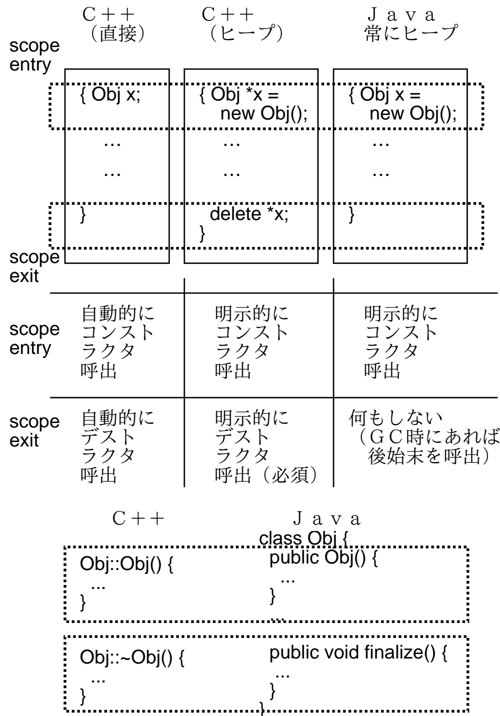
- そのため、Java ではごみ集め (GC、後述) が必須。C++ではごみ集めはオーバヘッドがあるので言語本体としては提供しない→手動メモリ管理が前提 (うまくやれば効率はよいが面倒、間違えるとメモリリークしたり落ちたりする)。

□ C++でのオブジェクトの初期設定と後始末は…

- コンストラクタとデストラクタ (名前が「〜クラス名」) による
- ローカル変数のオブジェクトはスコープに入った時にコンストラクタが呼ばれ、出るときにデストラクタが呼ばれる。
- ヒープ上のものは「new クラス名 (…)」でメモリ割り当て+コンストラクタ、「delete オブジェクト」でデストラクタ+メモリ開放

- 配列の場合「new クラス名 [大きさ]」で割り当て+デフォルトコンストラクタ (引数なし)、「delete[] ポインタ」でデストラクタ+メモリ開放。(ポインタが単一要素か配列参照か分からないという弱点は C 言語から引き継いでいる。)

□ Java の場合は「new クラス名 (···)」でコンストラクタというのは C++と同じ。配列はポインタ配列なので初期値 null。領域回収は GC なのでずっと簡単。後始末用にはメソッド finalize()、ただし GC が領域を回収するときに呼ばれるので、呼ばれない場合も。



□ C++ではこれだけ「スタック上に直接」のために努力しているにもかかわらず、継承 (とオブジェクト指向) を使う場合はヒープ割り当てしてポインタアクセスするしかない!

- 理由: スタック上のオブジェクトは「大きさ固定」だからサブクラスでインスタンス変数を増やしたりしても対応できない。無理にスタック上に取っても格納するときに「余分な変数が切捨て」変換されてしまい役に立たない。
- だったら Java 方式でいいじゃん!

4.7 Java とごみ集め

□ Java のコード→どんどん必要なオブジェクトを生成し、使わないものはごみ集め (Garbage Collection、GC) で回収、というスタイル。

- GC のオーバヘッドは織り込み済みという割り切りが必要。
- 自分で領域開放するよりずっと楽。自前でやるとコードが複雑化し、また重大な誤りの原因になりやすい。
- GC は Lisp あたりでは古くから使われているが、普通の手続き型言語で標準としたのは Java がはじめて

□ GC の原理: 変数から参照できるオブジェクト群を順次たどって行き、たどったという印をつける。印がつかなかった領域はごみとして回収

- その他、使っているオブジェクトだけコピーする方式、オブジェクトがそれぞれ何個所から指されているか常に数えておく方式などもある。
- GC で自動的に回収するためには「不要なデータ構造は指さないようにする」(指している所に null を入れるなどしてつながりを切る) ことが必要
- 「もう要らないからこれ回収して」とは言えない (安全のため)

4.8 C++のコピーコンストラクタと代入演算子

□ 「整数の集合」を Java で実装→実際の並びは配列に格納→値を追加していくと最初に用意した配列では不足→「配列が足りなくなったら大きくする」

```
class IntSet {
  int[] arr;
  int count = 0;
  public IntSet() { arr = new int[0]; }
  private IntSet(int c) { arr = new int[c]; }
  private void add1(int x) {
    if(count+1 >= arr.length) {
      int[] a = new int[arr.length*2 + 1];
      for(int i=0; i<count; ++i) a[i] = arr[i];
      arr = a;
    }
    arr[count++] = x;
  }
  (以下略)
}
```

- Java では配列は常に「別オブジェクトへの参照」になる
- 必要な容量を見積もってその大きさで用意し、容量が不足したら大きいものを用意して差し替え。
- 使わなくなった古い配列はごみ集めにより回収

□ 同じものを C++で実装するとどうか?

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

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

- 初期化時に配列サイズを決めて割り当てる。
- デストラクタが必要に (このオブジェクトが不要になったら指している配列も不要になるから。
- `add1()` で配列が満杯になったら増やす。

□ クラス定義部分のコードを示す:

```
class Intset { // set of ints
  int count, limit;
  int *arr;
  Intset(int c);
  void add1(int x);
  static int min(int x, int y) { return (x<y)?x:y; }
public:
  Intset();                    ←空の並びを作る
  Intset(const Intset &s); ←???
  ~Intset();                   ←デストラクタ
  Intset& operator=(const Intset& s); ←???
  int size() const;            ←サイズ
  bool is_in(const int i) const; ←包含判定
  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);
};
```

- ヘンなコンストラクタと演算子が増えている?

□ QUIZ: あるクラス `Someclass` の変数 `x`、`y` があるとする。

□ 次の 2 つは同等だと思う?　YES/NO

```
(A) Someclass x = y;  (B) Somclass x;
                          x = y;
```

□ 前問の答え:　「初期化」と「代入」は別物

- 初期化は「ゼロから内部データ構造を作る」
- 代入は「できているデータ構造に格納する」

□ 次の場合は初期化か代入か?

```
(A) int f(Someclass x) (B) Someclass f() {
    ...                        ...
    f(y) ←呼び出し             return y; ←返値
                             }
```

□ 前問の答え:　いずれも「初期化」。引数は実引数のコピーで初期化される。返値は返値用のテンポラリが `return` の式で初期化される。

□ 「初期化」も「代入」も内部データ構造に応じたものを用意しないとまずい*場合がある*→「コピーコンストラクタ」と「代入演算子」

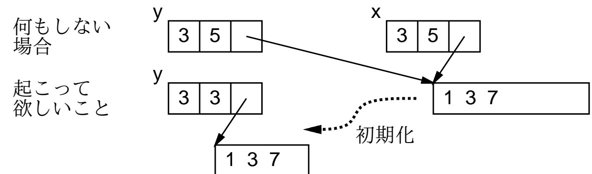
```
Intset(const Intset &s);
Intset& operator=(const Intset& s);
```

- 指定しない場合は「各メンバごとのコピー」が用意される。それだとどういう場合にまずいか分かりますか?

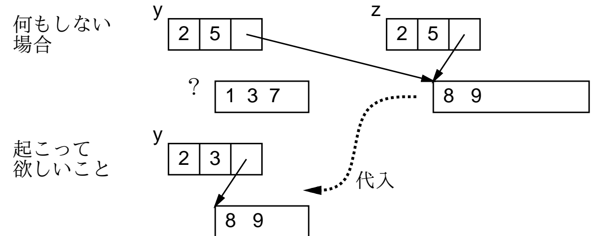
□ 通常のコピー (引数渡し等)、代入動作→オブジェクトのそれぞれのインスタンス変数をコピー。

- 変数 `arr` をコピーする→ポインタのコピーであって配列はコピーされない (共有された状態になる) →それでは困る (副作用とか)。
- このため、「コピーコンストラクタ」と「代入演算子」を作成して、その中で `arr` が指す配列をコピーする。

Intset y = x;



y = z;



□ 具体的なコード (この種のものとしては典型的)

```
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];
}
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;
}
```

- 代入の場合は「自分への代入は何もしない」を入れておくのが通例。

4.9 本節のまとめ

- □ 設計思想として、Javaはオブジェクトを多数使って、動的に(実行時に)さまざまな処理を行う。C++はできるだけコンパイル時に解決することで、効率良く実行できることを重視。
- □ Javaは実行時に何でもできるようにする分、遅い面も。
- □ C++は実行時の効率重視であるため、実行時にできることは少ない。
- □ C++は効率重視のため色々選べるがそのため複雑になっている面も。
- □ コンパイル時の参照方式やインタフェースなど、Javaの方が新しい分スマート。また、Javaの方が安全。
- □ C++は「プログラマが分かっていて使う」言語。分かっていれば何でもやっていいが、相応の責任。
- □ Javaは「安全側にチェックしつつ動く」言語。危ないことはできなくしてあり、安心だがのろい。
- □ あなたなら、どちらが好みですか?

5 実行時の動的処理と注記

5.1 Javaのリフレクション機能

- □ 自己反映(reflection): 実行中のコードが、実行系の情報にアクセスしたり、実行系の状態/動作を変更したりできるような機能
  - 狭い意味では前者のみ(後者をreificationと呼んで区別することも)
- □ 何のためにそんなことをするのか?
  - 例: デバッガ→実行系の内部状態を調べたり変更する必要
  - 例: システムの拡張→「任意の手続き呼び出しを遠隔メッセージに変換」など
  - 例: 拡張可能言語(構文や意味づけ等)
- □ Javaの自己反映機能→処理系そのもを変更する、という部分はない。
  - 内部の状態をのぞく
  - のぞいた情報を利用して、その場でメソッド呼び出し等を組み立てて実行させられる
- □ 強い型の言語は「型が合わなければ扱えない」→リフレクションのような自由自在なことは表しにくい
  - Javaではこれらをきちんと型を割り当てた上で可能にしている
  - ある意味では、Lisp等の「eval」(任意のプログラムを合成してその場で走らせる」を強い型の言語上で可能にしたといえる
- □ 任意のオブジェクトはgetClass()でそのClassオブジェクトを取得できる
  - または、「型名.class」でもClassオブジェクトを指定できる
- □ Classオブジェクトはそのクラスに関する情報を取得するメソッドを持つ
  - 例: getConstractors(), getMethods(), getMembers()
- □ ConstructorオブジェクトのnewInstance()を呼ぶとオブジェクトが生成される
- □ Methodオブジェクトのinvoke()を呼ぶとメソッドが実行できる
- □ たとえば、任意のクラスを1つもってきてオブジェクトを生成しメソッドを呼ぶ(ただし引数はすべて空)というプログラム

```
import java.util.*;
import java.lang.reflect.*;

public class Sample51 {
  public static void main(String args[]) {
    Scanner sc = new Scanner(System.in);
    while(true) {
      try {
        System.out.print("Class Name? ");
        String cname = sc.nextLine();
        if(cname.equals("q")) break;
        Class cls = Class.forName(cname);
        Constructor[] cons = cls.getConstructors();
        for(int i = 0; i < cons.length; ++i)
          System.out.println(""+i+": "+cons[i]);
        System.out.print("Constructor Number? ");
        int cno = sc.nextInt();
        Object obj =
          cons[cno].newInstance(new Object[]{});
        Method[] meths = cls.getMethods();
        for(int i = 0; i < meths.length; ++i)
          System.out.println(""+i+": "+meths[i]);
        System.out.print("Method Number? ");
        int mno = sc.nextInt(); sc.nextLine();
        Object res = meths[mno].invoke(obj, new Object[]{});
        System.out.println("Result Class:"+ (res.getClass()));
        System.out.println("Result: "+res);
      } catch(Exception e) { e.printStackTrace(); }
    }
  }
}
```

□ これをたとえば次のクラスに対して使ってみる…

```
public class Sample51Test {
  int val;
  public Sample51Test() { val = 1; }
  public Sample51Test(int i) { val = i; }
  public Sample51Test add() {
    return new Sample51Test(val+1);
  }
  public Sample51Test sub() {
    return new Sample51Test(val-1);
  }
  public String toString() {
    return "Sample51Test("+val+")";
  }
}
```

□ なお、この方法で通常取れるのは public なものだけ (セキュリティ上の制約)

5.2 リフレクション機能の使用例

□ 例: コンポーネントツールや GUI ビルダ

- 画面で直接部品を配置したり動かしてみたりしたい
- 動かせるためには「部品そのもの」を操作したい
- しかし、部品オブジェクトは部品ごとに「使い方」が異なる
- →自己反映機能を用いて調べつつ「呼び出せば」よい

□ Java による GUI 部品配置の例

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;

public class Sample52 extends JFrame {
  int x1, y1, x2, y2, x0, y0, width, height;
  Component c0 = null;
  JPanel p0 = new JPanel() {
    public void paint(Graphics g) {
      g.drawRect(x0, y0, width, height); } };
  Label l0 = new Label("Component:");
  TextField t0 = new TextField();
  Button b0 = new Button("Create");
  Button b1 = new Button("Move");
  Label l1 = new Label();
  Choice c2 = new Choice();
  Label l2 = new Label("String:");
  TextField t2 = new TextField();
  Button b2 = new Button("Call");
  Method[] meths; int[] maps;
  public Sample52() {
    getContentPane().add(p0); pack();
    p0.setLayout(null);
    p0.add(l0); l0.setBounds(10, 10, 60, 30);
    p0.add(t0); t0.setBounds(80, 10, 200, 30);
    p0.add(b0); b0.setBounds(290, 10, 60, 30);
    p0.add(b1); b1.setBounds(360, 10, 40, 30);
    p0.add(l1); l1.setBounds(10, 40, 380, 30);
    p0.add(c2); c2.setBounds(10, 70, 400, 30);
    p0.add(l2); l2.setBounds(10, 110, 60, 30);
    p0.add(t2); t2.setBounds(80, 110, 200, 30);
    p0.add(b2); b2.setBounds(290, 110, 40, 30);
    b0.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        try {
          c0 = (Component)Class.forName(t0.getText()).newInstance
          p0.add(c0);
          c0.setBounds(x0, y0, width, height);
          t0.setText("");
          meths = c0.getClass().getMethods();
          maps = new int[meths.length];
          c2.removeAll();
          for(int i=0,k=0; i < meths.length; ++i) {
            Class[] arg = meths[i].getParameterTypes();
            if(arg.length == 1 && arg[0] == String.class) {
              c2.add(meths[i].toString());
              maps[k++] = i;
            }
          }
        } catch(Exception ex) {
          l1.setText(ex.toString());
        }
      }
    });
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        if(c0 != null) {
          c0.setBounds(x0, y0, width, height);
        }
      }
    });
    b2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        try {
          Method m = meths[maps[c2.getSelectedIndex()]];
          m.invoke(c0, new Object[]{t2.getText()});
          t2.setText("");
        } catch(Exception ex) {
           l1.setText(ex.toString());
        }
      }
    });
    p0.addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
        x1 = x2 = e.getX(); y1 = y2 = e.getY();
        calcbox(); repaint();
      }
      public void mouseReleased(MouseEvent e) {
        x2 = e.getX(); y2 = e.getY();
        calcbox(); repaint();
      }
    });
    p0.addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent e) {
        x2 = e.getX(); y2 = e.getY();
        calcbox(); repaint();
      }
```

```
    });
  }
  private void calcbox() {
    x0 = Math.min(x1, x2); y0 = Math.min(y1, y2);
    width = Math.abs(x1-x2); height = Math.abs(y1-y2);
  }
  public static void main(String[] args) {
    Sample52 app = new Sample52();
    app.setDefaultCloseOperation(EXIT_ON_CLOSE);
    app.setSize(600, 400); app.setVisible(true);
  }
}
```

5.3 注記 (アノテーション)

□ 動的にクラスやオブジェクトを取り出して操作 (リフレクション) → それらのクラスや要素 (メソッド等) に関する情報も参照したい

- リフレクションで取れる情報: メソッド名やシグニチャ情報などに限定
- 「このメソッドについてはこれこれ」というツール/アプリ固有の情報を付属させたい
- クラスについては、「何もメソッド定義しないインタフェース」を使って「印つけ」できる --- 例: class X implements Serializable
- それ以外はこれまでは「コメントに書く」「それをツールで処理」くらいしか方法がなかった→繁雑、標準化されてない、チェックされない

□ JDK 1.5 から注記 (アノテーション) 機能を導入

- クラス、メソッド、変数の前に「これはこうね」という付加情報を書ける
- 付加情報の内容や扱い (実行時まで残すか等) をユーザが制御可能
- 標準の注記もいくつか。@Deprecated (もう古いから使わないこと)、@Override など。コンパイラが処理。

```
@Override
public void paint(Graphics g)...
```

- とくに @Override (このメソッドは他のメソッドを差し替えてます) はこれまでの「メソッド名をミスタイプしたため差し替えが効いてない」というバグの防止に有効

```
@Override
public void panit(Graphics g)... ←警告!
```

□ アノテーションは自分で定義できる。

- インタフェース定義の形をしているが「@interface」ではじまる
- メソッドを定義すると、そのメソッド名で値を保持できる

```
@interface Marked {
  String name();  ←name という文字列と
  int value(); ←value という数値を保持
}
```

- アノテーション定義の前につけるアノテーション (メタアノテーション) で「いつまで残すか」などを指定できる (コンパイル時、class ファイルに残す、実行時まで残す)

```
@Retention(RetentionPolicy.RUNTIME)
```

□ クラス、メソッド、変数の前にアノテーションを指定

- 上記種別の「どれに」指定できるかもメタアノテーションで指定可能

□ 指定方法…

- 何もメソッドがないアノテーションは「@注記名」だけ指定
- メソッドがある場合は「@注記名 (名前=値, 名前=値, ...)」で指定

□ 利用方法…

- クラスオブジェクト、メソッドオブジェクト等に getAnnotation() というメソッドがあり、これにアノテーション型 (インタフェースに対応したクラスオブジェクト) を渡すと対応する型のアノテーションが (付加されていれば) 取得できる

```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface Marked {
  String name();
  int value();
}

class Test {
  String str;
  public Test(String n) { name = n; }
  @Marked(name = "abc", value = 123)
  public void m1() { System.out.println("m1:" + str); }
  public void m2() { System.out.println("m2:" + str); }
  public void m3() { System.out.println("m3:" + str); }
  @Marked(name = "xyz", value = 456)
  public void m4() { System.out.println("m4:" + str); }
}

public class Sample53 {
  public static void main(String[] args) throws Exception {
    Test t = new Test("Me");
    Class cls = t.getClass();
    Method[] meths = cls.getMethods();
```

```
    for(Method m: meths) {
      Marked a = m.getAnnotation(Marked.class);
      if(a != null) {
        System.out.println(a.name() + a.value());
        m.invoke(t, new Object[]{});
      }
    }
  }
}
```

5.4 本節のまとめ

- □ 自己反映機能→「プログラムの中で、プログラムそのものを操作できる」→より柔軟なソフトウェア構成が可能になるが、その反面、プログラムが分かりにくくなる
- □ どこまで「静的」(コンパイル時)、どこから「動的」(実行時) の線引きをきちんと考える必要はあるが、動的にもできることは Java の利点
- □ アノテーション機能→動的に扱う時に付加情報をアドホックでない方法でコードそのものに入れておける→さまざまな用途に利用可能

6 型パラメタと総称的プログラミング

6.1 型パラメタ

□ 配列型→「int の配列」「char の配列」など型をパラメタに持てる

- 見かたを変えれば…1 つの「もの」が複数の型に対し利用できる←「総称的」(generic) という
- 組み込み演算子なども総称的。5 / 2 == 2, 5.0 / 2.0 = 2.5。
- 総称関数 … max(int,int) → int, max(double,double) → double 等。← オーバローディングである程度はできるが、有限個数
- array[int]、array[double] …型がパラメタになっている (型パラメタ) →無限の場合に対応。

□ 「さまざまな型の配列」は古くからあるが…

- 今日の言語では、後から作成するライブラリでも同様のことがやりたい (予め用意されているものとの間の「区別」をなくしたい)
- パラメタつきの型「自前で」(ユーザ定義で) 作れるようにするには?
- C++ → テンプレート機能 (わりと前からある)
- Java → Generics 機能 (JDK 1.5 から)

6.2 コンテナクラス

□ C++でも Java でも配列は「個数を最初に指定」→途中で大きさが伸び縮みできるものが欲しい→ライブラリで用意。

□ JDK 1.4 までの Java では → 例: ArrayList クラス。任意のオブジェクトを格納できる配列。

```
import java.util.*;

public class Sample61 {
  public static void main(String[] args) {
    ArrayList a = new ArrayList(); //サイズ 0
    for(int i = 0; i < 10; ++i)
      a.add("X" + i);  // 文字列を追加
    for(int i = 0; i < a.size(); ++i) {
      String s = (String)a.get(i); // 取出し
      System.out.println(s);
    }
  }
}
```

- 何か疑問な点はありますか?
- この方法の弱点は何だと思う?

□ ダウンキャスト … 「元の型にキャストし戻す」こと。

- a.add("abc") --- Object 型のパラメタに String を渡す→ OK
- String s = a.get(i) --- Object 型は String 型に入れられない。×。
- String s = (Object)a.get(i) --- ダウンキャスト:親クラスの型から子クラスの型へのキャスト。このとき「元々なに型だったか」がチェックされる。String でなかったものは String にはキャストできない (ClassCastException が投げられる)。

□ 弱点 (まずいこと) のまとめ

- a. ダウンキャストは繁雑。
- b. 実行時にチェックが必要→オーバヘッドになる (Java はそれを気にしない言語だということはあるが)。
- c. オブジェクトでないと入れられない。たとえば int は Integer クラスのインスタンスにして入れなければならない (これもオーバヘッド)。
- d. 配列だから「均一なものの並び」にしたいのに、何でも入ってしまう。
- e. 間違ってヘンなものを入れてもコンパイラは怒ってくれない。
- f. そのヘンなものが取り出されて来たところで実行時にエラー。

- □ 実行してみないとエラーが出ないというのはコンパイルする言語にとっての敗北。
- □ 根本的な問題…本来「型パラメタ」によって複数の型 (クラス) を使用するべきところを 1 個の ArrayList クラスで済ませていること。
- □ JDK 1.5 から「型パラメタ」が使えるようになった (Java Generics)。ArrayList<T>のようにパラメタは「<>」の中に指定する。T のところには任意の型 (ただしクラス) を入れてよい。
- □ 動かすときは…当然、JDK 1.5 以降の javac を使う。逆に 1.5 の javac で 1.4 のソースを動かすときはオプション指定必要
  - javac -source 1.4 Sample31.java ← JDK1.4 ソース
  - javac Sample31b.java ← JDK1.5 ソース (Generics)

```
import java.util.*;

public class Sample61b {
  public static void main(String[] args) {
    ArrayList<String> a = new ArrayList<String>();
    for(int i = 0; i < 10; ++i)
      a.add("X" + i);
    for(int i = 0; i < a.size(); ++i) {
      String s = a.get(i);
      System.out.println(s);
    }
  }
}
```

  - これで先の弱点のうち a、b、e、f が解消。残る c はダメ。
  - 実はこのコードを実行するとき内部的には先と同様のコードが使われている。つまり ArrayList の実装は 1 つ (均一な実装、homogeneous implementation)。だから c の制約が残っている。
  - c の緩和策として、JDK 1.5 で int と Integer、double と Double 等の自動変換 (AutoBoxing/Unboxing) を追加。

```
ArrayList<Integer> a = ...
a.add(3); // JDK1.5 で○
...
int i = a.get(0); // 〃
```

- □ C++では→言語設計上のポリシーとして、c は受け入れられない。また、C++では配列は「直接そのオブジェクトが埋め込まれた」ものになるので、コンテナクラスも同様でなければ受け入れられない。←フルスピードで動く C に負けないことが原則。

```
//sample61
#include <vector>
#include <iostream>

int main() {
  vector<int> a;
  for(int i = 0; i < 10; ++i)
    a.push_back(i+9);
  for(int i = 0; i < a.size(); ++i)
    cout << a[i] << '\n'; // operator[]
}
```

  - C++では型パラメタをソースのその位置に埋め込んで全体をコンパイルするという実装 (非均一な実装、heterogenous implementation)。だから型パラメタごとに別のコードができる。その代わり、その場展開ができる→極めて高速。

6.3 型パラメタと継承関係

- □ X ⊃ Y で継承関係を表すものとする。たとえば Object ⊃ String。
- □ パラメタつきクラス C<T>を考える。
- □ クイズ: X ⊃ Y ならば、C<X>⊃ C<Y> だといえるか?
  - たとえば、ArrayList<Object> ⊃ ArrayList<String> か? YES/NO?
- □ 次のように考えるべき
  - X ⊃ Y とは、X 型オブジェクトの代わりに Y 型オブジェクトを使っても OK であること (互換性があること)
  - 例: 「自動車」クラスの変数に「乗用車」を格納しても OK(互換性がある)
- □ ArrayList<Object> a = new ArrayList<String>; は OK?
  - a.get(0) ... String が取り出されるから OK。
  - a.set(0, Z) ... Object が入れられないと困るが String しか駄目!!!
- □ よって先の質問の答えは「NO」であるべきなんだけど…
- □ 実は配列でも同じことが起こる。
- □ しかし! Java では X ⊃ Y ならば X[] ⊃ Y[] になっている
  - 「X[] a = new Y[10];」が可能。そのあとで「a[0] = new X();」→ ArrayStoreException 発生
- □ 一般にこういうのを contravariance/covariance problem というらしい

6.4 Java Generics の制約

□ Java Generics では実行時には型パラメタ情報がなくなっている→そのため、型検査の抜け道が起きる可能性→それを防ぐための制約

- たとえば、「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」

```
ArrayList<String>[] lsa = new ArrayList<String>[5];
// ↑これは本来は許されない
Object[] oa = (Object[])lsa; // Java では OK
oa[0] = new ArrayList<Integer>(); // !!!
...
String s = lsa[0].get(0); // 実行時例外!!!
```

6.5 イテレータ

□ コンテナの種類→可変長配列 (ArrayList<T>、vector<T>)、連結リスト、B 木、ハッシュ表、などなど。

- どれでも「順番に要素を処理する」などの操作は必要→それをどれでも同等に扱えるようにしたい (後でコンテナを取り替えたりしても利用コードは変えないで済むように)。
- 「イテレータ (iterator、反復子)」→次々に要素にアクセスしていくためのオブジェクト。

□ Java では共通の Iterator インタフェースとして規定。Iterator<T>のように各要素の型が型パラメタになる。

```
bool hasNext() --- 終わりかどうか調べる
T next() --- 次の値 (T:パラメタ型) を取り出す
void remove() --- 現在値を削除 (今回は使わない)
```

```
import java.util.*;

public class Sample61c {
  public static void main(String[] args) {
    ArrayList<String> a = new ArrayList<String>();
    for(int i = 0; i < 10; ++i)
      a.add("X" + i);
    for(Iterator<String>i = a.iterator(); i.hasNext(); ) {
      String s = i.next();
      System.out.println(s);
    }
  }
}
```

□ JDK 1.5 から、上のような長い for 文を書かなくても済むように。

- 式 a の型がインタフェース Iterable<T> に従っているなら、次の 2 つは同等。

```
for(Iterator<T>i = a.iterator(); i.hasNext(); ) {
  T x = i.next();
  ...
}

for(T x : a) {
  ...
}
```

□ C++では共通のインタフェースはない (各コンテナごとに違う) が形としては次のもの。

```
*p --- 現在の要素を参照
++p、p++ --- 次の要素に進める
p == q --- 同じ要素を指すかどうか判定
```

```
//sample61b
#include <vector>
#include <iostream>

int main() {
  vector<int> a;
  for(int i = 0; i < 10; ++i)
    a.push_back(i+9);
  for(vector<int>::iterator i = a.begin(); i != a.end(); )
    cout << *i++ << '\n';
}
```

- 「vectior<int>::iterator」→ vector<int>クラスにおいて typedef で定義されている iterator という型名。
- 普通の「配列要素を指すポインタ」もイテレータとして使える
- コンテナにもよるが単なるポインタではないイテレータが大半

6.6 型パラメタつきクラスを作る

□ 例: リングバッファ。次々に値を追加できるが、最近の N 個だけが記憶されている。(実際には 2 のべき乗個を記憶→計算が簡単だから)

□ C++版 (クラス定義の中に直接コード本体も書いている。こうするとインライン展開が可能に。または従来通り分けて書いてもいいが、その場合は inline と指定。)

```
//sample62
#include <iostream>

template<typename T>  class ring_iter {
  const int mask;
  const T *arr;
  int index;
public:
  ring_iter(const int m, const T *const a, const int i)
    : mask(m), arr(a), index(i) { }
  const T& operator*() const {
    return arr[index];
```

```
  }
  bool operator!=(const ring_iter &i) const {
    return index != i.index;
  }
  ring_iter operator++(int dummy) {
    index = (index-1)&mask; return *this;
  }
  int geti() const { return index; }
};

template<typename T> class ring {
  static const int mask = 0x0f;
  static const int size = 10;
  T arr[mask+1];
  int count;
public:
  typedef ring_iter<T> iterator;
  ring() { count = 0; }
  void push_back(T v) {
    arr[mask & (count++)] = v;
  }
  iterator begin() {
    return ring_iter<T>(mask, arr, count&mask);
  }
  iterator end() {
    return ring_iter<T>(mask, arr, (count-size)&mask);
  }
};

int main() {
  ring<int> a;
  for(int i = 0; i < 100; ++i)
    a.push_back(i+9);
  for(ring<int>::iterator i = a.begin(); i != a.end();)
    cout << *i++ << '\n';
}
```

□ Java 版。パラメタ型の配列を生成することはできないという制約があるため、ArrayList<T>を使っている。

- イテレータをコンテナの内部クラスにすることでいちいち内部データ構造を渡して初期化しなくて済む(直接参照できる)

```
import java.util.*;

public class Sample62 {
  public static void main(String[] args) {
    Ring<Integer> a = new Ring<Integer>(0x0f, 10);
    for(int i = 0; i < 100; ++i)
      a.add(i+9);
    for(int v : a)
      System.out.println(v);
  }
}

class Ring<T> implements Iterable<T> {
  final int mask, size;
  ArrayList<T> arr;
  int count = 0;

  public Ring(int m, int s) {
    mask = m; size = s;
    arr = new ArrayList<T>(mask+1);
    for(int i = 0; i < mask+1; ++i) arr.add(null);
  }
  public void add(T v) {
    arr.set(mask & (count++), v);
  }
  public Iterator<T> iterator() {
    return new MyIter(count&mask, (count-size)&mask);
  }
  class MyIter implements Iterator<T> {
    int index, end;
    public MyIter(int i, int e) {
     index = i; end = e;
    }
    public boolean hasNext() {
      return index != end;
    }
    public T next() {
      index = (index-1) & mask;
      return arr.get(index);
    }
    public void remove() {
      throw new UnsupportedOperationException();
    }
  }
}
```

□ C++と Java の比較…

- C++はテンプレートによる型パラメタの実装が古くからあるのでこなれている。高速。ただし美しくない部分も。
- Java は Generics を後づけで入れたのでこなれていない部分もある。一方で新しい設計ならではの見やすさもある (内部クラスや for)。

6.7 Standard Template Library (STL)

□ C++の標準ライブラリの土台となった、テンプレートを駆使したライブラリ群→現在では C++標準ライブラリのコンテナクラス部分のことをそう呼ぶ。重要なアイデア:

- テンプレートを使った効率のよいコンテナクラス群
- 要素アクセスはコンテナクラスのメソッドではなくイテレータ経由 (その方が高速化しやすい)
- イテレータに対して働くアルゴリズム群。find (線形探索)、sort (整列)、count(計数)、などなど。
- アルゴリズム群はテンプレート関数 (型パラメタつき関数) →高速。

□ 例: 普通の swap(C 版)

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

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

- 関数呼び出しのオーバヘッド、ポインタ経由のアクセス→遅い。しかも型ごとに用意する必要。

□ 関数テンプレート版の swap(C++版)

```
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]) ...
```

- 関数テンプレートの型パラメタは推定される (a[i] 等が float だから T は float) →繁雑な指定が不要
- 「float z = a[i]; a[i] = a[j]; a[j] = z;」がこの場所に埋められているのと同様→高速

□ アルゴリズム関数を利用すれば制御構造をこちら側で書かなくてもよくなる。たとえば for_each を使うとループが不要になる。

```
//sample63
#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>);
}
```

- これで for_each の中で「f(値)」が繰り返し呼ばれる→「ベクタの各要素を出力」ができる。ループは不要。
- しかし「合計を取る」だとどうするか?

□ 関数オブジェクト… operator() を持つようなオブジェクト。先の「f(値)」というのは関数でなく operator() の呼び出しでもよいので。

```
//sample33b
#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';
}
```

- ループ周回を通じて保存されて欲しいものは関数オブジェクトのインスタンス変数にすればよい。→ ループ構造と処理の分離

6.8 テンプレートの特殊化

□ テンプレートで一般向けの実装のほかに「特殊ケース」を用意したいことがある。

- たとえば bool(0/1) の ring だったら 1 ワードにシフトでデータを入れて行くとか…
- コンテナクラスの場合、「ポインタ用」はそれなりのポインタ用のものを使う方がよさそう… (「ポインタ用」のクラスを 1 つ用意して残りはすべてそれを呼ぶ→コードを何回も展開しなくて済む→コード量の爆発を防ぐ上で重要なテクニック)

□ そのような「特殊ケース」を別途用意できる→特殊化

```
template <typename T> class ring {
  //一般
}
template <typename T> class ring<T*> {
  // ポインタ用
}
template <> class <bool> {
  // bool 用
}
```

6.9 型パラメタに対する制約

□ 例: maxbuf<T>というコンテナを作る。

- 次々に put(T x) で値を入れていく。
- T get() でその時点での最大値を取得できる。

```
//sample34
#include <iostream>

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';
}
```

□ 型Tには演算子「<」が必要。これを満たさないと…

```
//sample34b
#include <iostream>

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) { }
};

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]);
}
```

□ 上記のをコンパイルすると…

```
In method 'void maxbuf<point>::put(point)':
22:   instantiated from here
7: no match for 'point & < point &'
```

□ 確かに「<」がない、と言われる… 次のを入れればよくなるが。

```
  bool operator<(point &p) {
    return x*x+y*y < p.x*p.x+p.y*p.y;
  }
```

- しかし「コンパイルしてみたら無かった」では実装の中に立ち入っていてエラーメッセージとしてあまり良くないのでは…
- 実際、極めて分かりづらいエラーメッセージに遭遇することがよくある
- 本来だったら「この型パラメタにはこういう型しかあてはめてはいけませんよ」という指定ができた方がいいのでは? → 賛否両論。否定派: 指定が繁雑になり自由度が減る。ともあれ、Java Genericsではこちらの立場。

□ 上記のJava Generics版。整数はIntegerクラスを使う必要。

```
import java.util.*;

public class Sample33 {
  public static void main(String[] args) {
    int[] a1 = { 5, 3, 8, 2, 4 };
    Maxbuf<Integer> m1 = new Maxbuf<Integer>(a1[0]);
    for(int i = 0; i < 5; ++i) m1.put(a1[i]);
    System.out.println(m1.get());
    String[] a2 = { "this", "by", "foo", "z", "x" };
    Maxbuf<String> m2 = new Maxbuf<String>(a2[0]);
    for(int i = 0; i < 5; ++i) m2.put(a2[i]);
    System.out.println(m2.get());
  }
}

class Maxbuf<T extends Comparable<T>> {
  T max;
  public Maxbuf(T x) { max = x; }
  public void put(T x) {
    if(max.compareTo(x) < 0) max = x;
  }
  public T get() { return max; }
}
```

□ Comparable<T>というのはx.compareTo(T y)というメソッドを持ち、このメソッドがxとyの大小関係に応じて正/負/零を返す。

□ この方が確かにあらかじめチェックはできるが、窮屈かも…

□ あらかじめComparable<T>をimplementsしているクラスにしか使えない→自分がソースを持っていればいいが、そうでないと…

- 実はC++版にも同様の問題がある。たとえばchar*のmaxbufを取ろうとすると「アドレスが最大の」ものが取れてしまう…求めていることと違う。

□ C++の場合は、単独関数テンプレートと特殊化を使えばOK。

```
//sample34c
#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';
}
```

□ 特殊化の応用

- 一般の T →「<」が使われる
- char* → strcmp が使われる
- その他特殊ケースはいくらでも対応可能 … C++の実用性に 1 日の長があるかも?

6.10 テンプレートによる mixin クラス

□ 前回やったように、mixin とは「他のクラスに混ぜるためのクラス」

- それ自体単独でインスタンスを生成しないことから「抽象サブクラス」と呼ぶことも
- 前回の話では flavors などの多重継承を使って混ぜていた
- しかし C++のような静的検査な言語では多重継承ではうまく行かない

□ たとえば、次のようなクラスを考える。

```
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; }
};
```

□ 「update の回数を数える」

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

□ 「更新履歴を記録する」

```
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';
  }
};
```

□ これを次のようにして使いたい。

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

□ flavors ではこれでできる (どっちの update が呼ばれるかの規則があって OK)。しかし C++ではそもそも getValue() や update() の宣言がないと型検査できなくてアウト。

□ ではどうするか…

- count や history で親クラスを指定すればもちろん動くようになる。
- しかしさまざまなものにこれらの機能を組み込もうと思って作ったのに、決め打ちで親クラスを書いてしまうのでは 1 つにしか使えない (または繰り返しコピーになる) のでよろしくな。
- ではどうすればいい?

□ 答: その親クラスをテンプレートパラメタで指定する!!

```
//sample65
#include <iostream>

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

template<class 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();
}
```

□ mixin クラスを使うことで、1 つのクラスに盛り込まれている複数の側面を分離して記述し、使う時に組み立てることができる。

□ しかし、実際には 1 つの「側面」が複数のクラスにまたがって存在していることが普通→このような「一群のクラス」を 1 つの「層」として、それを積み重ねてシステムが構成されるというイメージ。

□ そのようなものを実現するには→ C++でもクラスの中にクラスが書けることを利用し、外側クラスをパラメタつきとする。

```
template<class T> class ThisLayer : public T {
public:
  class First : public T::First { ... };
  class Second : public T::Second { ... };
  ...
};
```

□ こうすれば、一群のクラスで Fist は First どうし、Second は Second どうし、…で縦に合成が起きる。

- ThisLayer の中にはこの層が実現する機能のためのデータやコードをまとめて入れることができる。
- このような構成を「mixin layers」と呼ぶ。
- このような「構成法」の代替案→言語自体の拡張 (AOP 言語)

6.11 本節のまとめ

□ 総称的プログラミング (型パラメタ、パラメタつき型) → C++ テンプレート、Java Genrics で実現 (内容はかなり違っている)。主な用途: コンテナクラス

□ STL(C++) → イテレータによるアクセス、汎用アルゴリズム

□ 型パラメタの制約 → 賛否分かれるところ。Java ではインタフェース/サブクラスによる制約指定可能 (だが不自由でもある) → C++に対する制約機構 (concepts) は設計途上 → 後で少し紹介

□ mixin クラス (C++) → 親クラスを拡張するための抽象サブクラス

□ 全体として C++テンプレートは強力、しかし難しい。

7 テンプレートメタプログラミング

7.1 テンプレートメタプログラミングとは

□ メタプログラミング --- 「プログラム自身を参照/変更するようなプログラミング」

□ 先にメタオブジェクトプロトコル (MOP) について紹介した

- (復習) オブジェクト指向的に言えば → 「クラス」というオブジェクトが存在し、それに対するメソッドが呼べる。そのメソッドによって、クラスに関する情報を取り出したり、クラスの内容を変更 (!) →

クラス定義を変更するということは、そのクラスに属するオブジェクトの動作が変更されるということ(!!)

- 実際には、コンパイルする言語では「動作を変える」のは難しい(実行中に動作を変更しようとしても、変更後の動作のコードをコンパイルできない…もうコンパイル段階は終わっているから) → インタプリタ系の言語(ないし、コンパイラが実行系に入っているような言語)でのみ実用化

□ コンパイル時MOP --- 上記とは別のアプローチとして、コンパイル時に動作するプログラム(メタプログラム)を分けて記述

- これなら、コンパイル時にメタプログラムが動き、プログラムを加工してから、コンパイルに進めばよい。そのかわり、メタプログラムはコンパイラと一緒に動く必要
- Open C++などの研究用言語があったが普及せず
- C++のテンプレート機構を使えばコンパイル時に計算できるじゃん! (誰が発見したのだろう?) → テンプレートメタプログラミング

□ テンプレートの実体化について整理すると…

- 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。
- パラメタには整数値も書けて、整数の計算はコンパイル時に行われる
- 特殊化を応用すると「パラメタがこの値ならこちら」という分岐が可能

□ →全体として、テンプレート機能を使って「コンパイル時の計算」ができる→「計算の結果によりプログラムを作る」ことができる→ テンプレートメタプログラミング。

7.2 基本的な例

□ 例: 階乗を「コンパイル時に」計算

```
//sample71.cc
#include <iostream>

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

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

- (std::coutとしているのはusing namespace stdを使わないため)
- テンプレートから値を返すにはenum型で定数を定義すればよい→その定数を「型::名前」で参照可能
- 数値が0かどうかの判定→テンプレートの特殊化で実装
- この「120」はコンパイル時に計算が終わっている→速い(といってもこの場合はたいしたことはないが)

□ 類似例: 組み合わせの数を「コンパイル時に」計算

```
//sample71b
#include <iostream>
#include <ctime>

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() {
  std::cout << Comb<10,4>::RET << '\n';
  std::cout << comb(10,4) << '\n';
  int i, v1 = 0, v2 = 0;
  clock_t t1 = clock();
  for(i=0;i<1000000;++i) v1 += Comb<10,4>::RET;
  clock_t t2 = clock();
  for(i=0;i<1000000;++i) v2 += comb(10,4);
  clock_t t3 = clock();
  std::cout << double(t2-t1)/CLOCKS_PER_SEC << '\n';
  std::cout << double(t3-t2)/CLOCKS_PER_SEC << '\n';
}
```

- さっきのと同様だが計測してみる…

```
210
210
0.0234375 ←はるかに速い (アタリマエ)
1.27344
```

7.3 テンプレートメタプログラミングの制御構造

□ もっと一般的な「IF 文」を作ることもできる。

```
//sample72
#include <iostream>

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

- IF の Then 部、Else 部はまた別のクラス (=テンプレートのインスタンス) を渡すようになっている。条件は bool 値。
- 「> >」のところは間をあけないと「>>」(シフト) になってしまい構文エラーになるので注意 (すごく嫌!)
- ただし、コンパイラによっては「テンプレートの部分特殊化」がサポートされていなくてこれでは通らないこともある (らしい)。(部分特殊化: `< false, Then, Else >` のように最初のパラメタだけ特殊化することをいう)

□ 部分特殊化を使わないように書き直すこともできる (が面倒)。

```
//sample72b
#include <iostream>

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

- 要するに 1 引数の Select で SelThen か SelElse かどっちかのテンプレートを返す。Select は 1 引数だから特殊化しても部分特殊化にならない。
- 以下の部分特殊化を使うものも全部同様に書き直せるが略。

7.4 大量コードの生成

□ 単一の値を計算するだけではあまり面白くないが、もっと長いコードを生成させることもできる。例: `pow<3>(x)` → `x*x*x`。

```
//sample38
#include <iostream>

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;
  std::cout << pow<10>(x) << '\n';
}
```

- ただし、この方法で pow<100>(x) とかやらせようとしても、テンプレート再帰呼び出しの深さが深くなりすぎるといって止められてしまう。

□ とりあえず、2 のべき乗個の展開を、区間を半分ずつにして再帰呼び出しする方法で書いてみた。

```
//sample73b
#include <iostream>
#include <ctime>

template<int n, int b> class pow0 {
public:
  static double calc(const double& x) {
    return pow0<n/2,b>::calc(x) *
           pow0<n/2,b+n/2>::calc(x);
  }
};
template<int b> class pow0<1,b> {
public:
  static double calc(const double& x) {
    return x;
  }
};

double pow1(int n, double x) {
  double r = 1.0;
  for(int i = 0; i < n; ++i) r *= x;
  return r;
}

int main() {
  double x = 1.00001;
  std::cout << pow0<1024,0>::calc(x) << '\n';
  std::cout << pow1(1024, x) << '\n';
  int i; double v1 = 0, v2 = 0;
  clock_t t1 = clock();
  for(i=0;i<1000000;++i) v1 += pow0<1024,0>::calc(x);
  clock_t t2 = clock();
  for(i=0;i<1000000;++i) v2 += pow1(1024, x);
  clock_t t3 = clock();
  std::cout << v1 << '\n';
  std::cout << v2 << '\n';
  std::cout << double(t2-t1)/CLOCKS_PER_SEC << '\n';
  std::cout << double(t3-t2)/CLOCKS_PER_SEC << '\n';
}
```

- この場合、テンプレート側 (pow0) では本当に 1023 個のかけ算命令が並ぶことになる→ループ制御が不要、コンパイルはのろい
- コードも大きくなるので、キャッシュを考えると得かどうはか場合による。
- 最適化を書けてインライン展開させないと無意味。

```
1.01029
1.01029
1.01029e+06
1.01029e+06
2.70312  ←倍くらい速い (思ったほどでもない?)
4.5625
```

□ 普通に「i=0,1,2,... とループする」があった方が便利→作ってみた。

```
//sample74
#include <iostream>
using namespace std;

template<int n, int b, template<int i>class Stat>
  class DO1 {
public:
  template<class Env>
    static void exec(Env& e) {
      DO1<n/2,b,Stat>::exec(e);
      DO1<n/2,b+n/2,Stat>::exec(e);
  }
};
template<int b, template<int i>class Stat>
  class DO1<1,b,Stat> {
public:
  template<class Env>
    static void exec(Env& e) { Stat<b>::exec(e); }
};

struct PowEnv { double r, x; };
template<int i> class PowStat {
public:
  static void exec(PowEnv &e) { e.r *= e.x; }
};

int main() {
  PowEnv e; e.r = 1.0; e.x = 1.00001;
  DO1<1024,0,PowStat>::exec(e);
  cout << e.r << '\n';
}
```

- 今度は変数は「環境 (enviroment)」に格納し、exec を呼ぶところで引数として渡す。結果も環境に格納。

□ もっと実用ぽく、「ベクトルの加算」をやってみた。

```
//sample39b
#include <iostream>
#include <ctime>

template<int n, int b, template<int i>class Stat>
  class DO1 {
public:
  template<class Env>
    static void exec(Env& e) {
      DO1<n/2,b,Stat>::exec(e);
      DO1<n/2,b+n/2,Stat>::exec(e);
  }
};
```

```
template<int b, template<int i>class Stat>
  class DO1<1,b,Stat> {
public:
  template<class Env>
    static void exec(Env& e) { Stat<b>::exec(e); }
};

struct VecEnv {
  double r, a[1024], b[1024];
};
template<int i> class VecStat {
public:
  static void exec(VecEnv &e) { e.r += e.a[i]*e.b[i]; }
};

double iprod(double a[1024], double b[1024]) {
  double r = 0.0;
  for(int i = 0; i < 1024; ++i) r += a[i]*b[i];
  return r;
}

int main() {
  VecEnv e; e.r = 0.0; int i; double v;
  for(i = 0; i < 1024; ++i) e.a[i] = e.b[i] = i;
  DO1<1024,0,VecStat>::exec(e);
  std::cout << e.r << '\n';
  std::cout << iprod(e.a, e.b) << '\n';
  clock_t t1 = clock();
  for(i=0;i<1000000;++i) DO1<1024,0,VecStat>::exec(e);
  clock_t t2 = clock();
  for(i=0;i<1000000;++i) v = iprod(e.a, e.b);
  clock_t t3 = clock();
  std::cout << double(t2-t1)/CLOCKS_PER_SEC << '\n';
  std::cout << double(t3-t2)/CLOCKS_PER_SEC << '\n';
}
```

- しかしこうやって全部展開したものが速いとは限らない

```
3.5739e+08
3.5739e+08
8.49219    ←あれ?
2.74219
```

- ループ最適化はコンパイラ業界では研究し尽くされている→単純に展開するより速い
- 実際には「数個ずつ展開しつつループで回る」(部分ループ展開) がよいとされている。

□ もちろん、2 次元や 3 次元も必要なら作れる。

```
//sample39c
#include <iostream>

template<int ir, int jr, int iv, int jv,
  template<int i, int j>class Stat> class DO2 {
public:
  template<class Env>
    static void exec(Env& e) {
      DO2<ir/2,jr,iv,jv,Stat>::exec(e);
      DO2<ir/2,jr,iv+ir/2,jv,Stat>::exec(e);
  }
};
template<int iv, int jr, int jv,
  template<int i, int j>class Stat>
    class DO2<1,jr,iv,jv,Stat> {
public:
  template<class Env>
    static void exec(Env& e) {
      DO2<1,jr/2,iv,jv,Stat>::exec(e);
      DO2<1,jr/2,iv,jv+jr/2,Stat>::exec(e);
  }
};
template<int iv, int jv,
  template<int i, int j>class Stat>
    class DO2<1,1,iv,jv,Stat> {
public:
  template<class Env>
    static void exec(Env& e) { Stat<iv,jv>::exec(e); }
};

struct TestEnv {
};
template<int i,int j> class TestStat {
public:
  static void exec(TestEnv &e) {
    std::cout << i << ',' << j << '\n';
  }
};

int main() {
  TestEnv e;
  DO2<16,16,0,0,TestStat>::exec(e);
}
```

7.5 本節のまとめ

□ テンプレート展開機構を用いてプログラムを「組み立てる」

- テンプレート展開機構の動作や記述方法は C++言語の通常の動作やその記述方法とはまったく違っている (むしろ関数型に近い)
- コンパイル時に計算してしまうので、効率的に有利(な場合)
- 弱点: テンプレートのデバッグは大変

□ テンプレートメタプログラミングの将来…

- C++ライブラリ実装者にとっては、効率を向上させる手段として有力
- 普通のプログラマがこれを駆使するようになるのかどうかは?
- Boost MPL (http://www.boost.org/libs/mpl/doc/) → Boost (C++の標準ライブラリに追加提案するものを集めたライブラリ群) に含まれる、テンプレー

トメタプログラミング向けライブラリ→ 普通の人もこれを書きましょうという方向?

8 テンプレートパラメタの仕様記述

8.1 はじめに

□ 以下の内容はまだ C++言語に入っているものではない…研究段階

- 次の C++規格では「このようなもの」が入ることが期待されている (が公式には何とも解らないし、次の規格まで何年か掛かる)
- 元ネタの論文は以下のもの
- Gabriel Dos Reis, Bjane Stroustroup, Specifying C++ Concepts, Proceedings on Principles of Programing Languages 2006 (POPL'06), pp. 295-308, 2006.
- Douglas Gregor, Jaakko Jarvi, Jeremy Siek, Concepts: Linguistic Support for Generic Programming in C++, Proceeginds of Object Oriented Systems, Languages and Applications 2006 (OOPSLA'06), pp. 291-310, 2006.

8.2 テンプレートパラメタの仕様記述問題

□ テンプレートはパラメタが「ある条件を満たしている」ことを前提に書かれている

- 現在はそのことはソースプログラム中に表現されていない
- 「条件を満たしていない」パラメタを与えた場合、コンパイルエラーになるがその箇所/内容はテンプレートのソース内になる
- テンプレートを「利用するだけの」プログラムには理解できない
- テンプレートを作る時のこの「条件」に関する間違いも作る時には解らない

□ 例: テンプレート関数 `fiill` --- 指定範囲を指定値で埋める

```
template<typename Iter, typename T>
void fill(Iter first, Iter last, const T& t) {
  for(Iter cur = first; cur != last; ++cur )
    *cur = t;
}
```

使い方:

```
vector<int> vec(100);
fill(vec.begin(), vec.end(), 0);
double a[100];
fill(a, a+100, 0.0);
```

- 型パラメタ `Iter` と `T` は「どのような条件」を満たす必要がある?

□ 従来の「一般的な」考えとしては、各型についてそれが持つメソッド名とその引数、返値の型を指定する

- しかし、C++の場合はあらゆる演算子がオーバロードでき自動型変換もあるので、これを全部指定するのは非現実的 (大変すぎる)
- テンプレート中に「`x = y + z;`」と書いてあったとして、この「`+`」も「`=`」も C++のもともとの演算なのか「`operator+`」等なのかは解らない (テンプレートはどちらでも対応可能)
- もっと別な方法がいいのでは…

8.3 concept 記述

□ ある型 (ないし型の並び) が満たすべき条件を指定するもの。たとえば先の `fill()` に使うための簡単化した concept の例を示す。

```
concept Mutable_fwd(typename Iter, typename T) {
  Var<Iter> p;
  Var<const T> v;
  Iter p = q;
  bool b = (p != q);
  ++p;
  *p = v;
};
```

- 実際にテンプレート中で使うコードと同等のものを書いてみせることで「こういうことができないといけませんよ」と指定
- `p` や `v` はそれぞれ `Iter` 型、`const T` 型の変数だとする。「`Iter p;`」としてしまうとデフォルトの引数なしコンストラクタで初期化可能であるという条件になってしまう。これを避けるため「`Var<...>`」という形を使う。

□ 上の concept が規定している条件は次のようになる

- `Iter` 型の値はコピー初期化できる
- `Iter` 型の値は「`!=`」で比較でき結果は論理値に変換できる
- `Iter` 型の値は「`++`」前置演算できる

- Iter 型を dereference した先に T の値を入れられる (そのとき T の値を取り出す変数は書き換えられることはない)

□ 上のような concept 記述を用いて、fill() を次のように書く

```
template<typename Iter, typename T>
  where Mutable_fwd<Iter, T>
void fill(Iter first, Iter last, const T& t) {
  for(Iter cur = first; cur != last; ++cur )
    *cur = t;
}
```

- テンプレート定義時に、concept の中身を用いてテンプレートのコードをチェックできる
- テンプレート利用時に、concept の中身を用いてテンプレートパラメタが条件を満たしていることをチェックする (満たしていなければこれこれの concept を満たしていないというエラーメッセージ)
- 強すぎず、弱すぎない concept 条件を指定することが大切

8.4 本節のまとめ

□ concept --- テンプレートパラメタが満たす条件を規定するもの

- 「どのように使うか」のコードを書けばいいというアイデア →記述しやすく解りやすい (代替案もあるがこの方がよさそう)
- concept 条件はテンプレートを作る側と使う側の「契約事項」。これを明示することにより、作る側も使う側もやりやすくなる
- まだ C++の標準として入るまでには時間が掛かりそう
- テンプレートパラメタをきちんとするというのは永年の課題なので次の改訂で入って欲しい…

9 全体のまとめ

□ C++と Java の違い…

- 誰でも答えられそうな質問だけれど、突き詰めるとすごーく奥深い
- 設計思想の違い: Java →動的、安全、のろい。C++ →静的 (コンパイル時)、速い、プログラマの責任。
- 久野の思っていること: 言語は人間が使うものだから、人間にとって分かりやすく書きやすいことが一番大切。その点で Java は平易だけれどコードが長々しすぎる。C++はコンパクトに書けるけれど裏側で何が起きているか分からなさすぎる。

□ 結局、理想の言語というのはありませんね (当り前)。ではあなたが作りますか? (半分本気)

10 より進んだ学習に向けて

□ プログラミング言語分野の研究について…

- 過去の EDP 参加者のアンケート中で「講座の後自分で勉強を進められるように参考書籍などを紹介して欲しい」というご意見がありましたので、お役に立つか分かりませんが一応。
- プログラミング言語の設計とか機能とかはかなり深くて狭い分野なので、あまり適当な本がありません (先端の研究は論文を読むしかない)。
- そもそも本数冊読んで済むのだと我々研究者の仕事は何なのかということになるわけで…
- C++と Java も入門書・解説書は山のようにありますが、今回喋ったような細かい話がきちんと「議論されて」いる本はあまりないです。
- とはいっても、何も無いではつまらないので、いくつか参考となる本を挙げておきます。

□ C++

- ビョーン・ストラウストラップ著, 岩谷 宏訳, C++の設計と進化, ソフトバンク, 2005. --- C++言語の設計者ストラウストラップが C++ 言語の進化の過程やそこで出て来たさまざまな設計上の選択などを議論。
- ビョーン・ストラウストラップ著, 長尾高弘訳, プログラミング言語 C++ 第 3 版, アジソンウェスレイ, 1998. --- C++言語の詳細について書いてあって本講座のような「楽しみ」がある C++の解説書はやっぱりストラウストラップの本かなという感じ。
- アンドレイ・アレキサンドレスク著, 村上雅章訳, Modern C++ Design – ジェネリック・プログラミングおよびデザイン・パターンを利用するための究極のテンプレート活用術 – , ピアソン, 2001. --- C++のテンプレートを駆使するマニアックな本ということで。

☐ Java

- ケン・アーノルド，ジェームズ・ゴスリン，デビッド・ホームズ著，柴田芳樹訳，プログラミング言語 Java 第4版，ピアソン，2007． --- これも Java 言語の設計が執筆陣に入っている本で、各種機能について詳細に説明。
- ブレット・マクラフリン，デイビッド・フラナガン著，菅野良二訳，Java 5.0 Tiger，オライリー，2007． --- JDK 1.5 で新しく入った機能 (Generics、アノテーション等) に絞って解説。

☐ 以上、和書を挙げましたが、英語でも構わないという人は… 本を読むのもいいですが、英語論文を読むのもいいです。学会としては ACM(Association for Computing Machinely，米国計算機学会) がメジャー。ACM の言語関係の代表的な国際会議:

- ACM Principles of Programming Languages (POPL)
- ACM Object-Oriented Systems, Languages and Applications (OOPSLA)
- ACM Programming Languaged Design and Implementation (PLDI)
- ACM (http://www.acm.org/) から会議録が買えます。またはオンラインで読める選択もあります。

☐ そして、プログラミング言語研究に興味があるという人はぜひ GSSM へ…