TSSI 基盤技術研修コース
— プログラミング言語 — ＃1

久野　靖*

2011.5.13

はじめに

□ 本講座のタイトル→「プログラミング言語」

□ 本講座の目的→

- (a) プログラミング言語の基本機能やメカニズム→「言語の機能/仕様」
- (b) それを用いてソフトウェアを組み立てる戦略・考え方→「パラダイム」

□ とくに「オブジェクト指向」に重点を置いている。

- 今日のソフトウェア開発において主流となっているパラダイム
- 道具だて、言語仕様が複雑、込み入っている
- 「なぜそうなっているか」という理屈はちゃんとある
- 「どういう理由で」「どうなっている」を学ぶ→使いこなし方が分かる

□ ソフトウェア開発は結局「自分の頭でどれだけ考えられるか」の限界までしか作れない

- 同じことをするのに「考えやすい道具 (言語)」を使うと→
- 同じ能力でより高度なものまで考える (作る) ことが可能になる
- ではどうすれば「考えやすい」のか→言語の設計思想を学ぶ

□ 構成:

- 第 1 回→プログラミング言語の諸概念+オブジェクト指向 (主として ADT) の導入
- 第 2 回→オブジェクト指向言語の主要メカニズムと用法
- 第 3 回→オブジェクト指向のさまざまなバリエーション (予定)

□ 参考書は毎回持参される必要はないです。講座の性質上、やや高度ですがきちんと解説されている本を選んでいます。疑問に思った点があれば参考書の該当個所を調べて読むという形で利用してください。

- (Java) Ken Arnold, James GoslingDevid Holmes 著, 柴田訳, プログラミング言語 Java 第 4 版, ピアソン, 2007.
- (C++) Stenley B. Lippman, JOsee Lajoie, Barbara E. Moo 著, 玉井訳, C++プライマー第 4 版, 翔泳社, 2006.

□ 上記でしんどい人は次のものなどどうでしょうか。ただし東芝様ではそんなに冊数は出せないそうなので自費ということになりますが。これらはざっと通読するのに向いています。

- 久野 靖、久野禎子、Java によるプログラミング入門 第 2 版、共立、2011.
- 高橋麻奈、やさしい Java 第 3 版、ソフトバンク、2005.
- 高橋麻奈、やさしい C++第 3 版、ソフトバンク、2007.

1　計算機言語とは…

□ まず計算機言語とは何かというお話から。

- では計算機とは何をするもの?

□ プログラミング言語…計算機言語の一種。

□ 計算機言語とはどういう言語?　なぜ存在する?

1.1　計算機と人間の情報伝達

□ 計算機…情報を取り扱うための装置

□ 人間と計算機はどうやって情報をやりとりするか?

□ ハードウェア…入出力装置

- 入力→キーボード、マウス、マイク、…
- 出力→ディスプレイ、プリンタ、スピーカ、…

□ 具体的にどのような形で情報をやりとり?

□ 計算機に自然言語 (日本語) で命令できたら嬉しいと思うか?　(Yes/No)

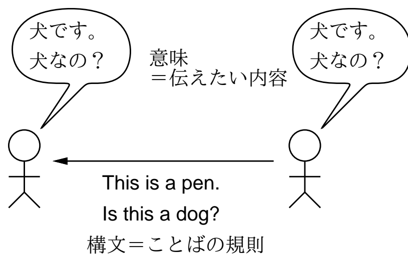
*筑波大学大学院経営システム科学専攻

1.2 人工言語

- □ 人間どうしの情報交換→自然言語 (日本語、英語、…)
  - 計算機とのやりとりには不向き
  - あいまいさ、処理が複雑
  - 人間を相手にするのと計算機を相手にするのでは違って当然 (?)
- □ 簡単な規則に基づく人工言語→計算機の処理に向く
  - 計算機で取り扱うために作った人工言語→計算機言語

1.3 構文と意味

- □ 構文→言語に現われる語の並び方の規則性
- □ 意味→どのような構文であれば何を意味するか



- □ 言語の定義→構文と意味のペア (計算機言語も同様)
  - ほかに語彙 (単語) も必要だけど。

1.4 プログラミング言語

- □ プログラムを書くための計算機言語
  - プログラムとは?
- □ プログラミング言語の用途?
  - 計算機に読み込ませる
  - 人間が読む (娯楽、仕事、…)
  - 自分が考えるための手段→でも動かした方が嬉しい

1.5 その他の計算機言語

- □ プログラミング言語でない計算機言語も多数ある
- □ 楽譜記述言語
- □ 図形記述言語
  - プリンタ→ページ記述言語→ PostScript…プログラミング言語でもある。
- □ 文書記述言語 (マークアップ言語)、データ記述言語
  - SGML → HTML → XML、XHTML
- □ アプリケーションを動かし画面で見れば十分?
  - 「処理」したいときは言語になっている方がよい。

1.6 この節のまとめ

- □ 計算機言語とは、計算機で扱うための人工言語
  - でも人間が読み書きしてもよい。
  - 人間が考えるための「手段」でもある。
- □ 代表はプログラミング言語だが、他の計算機言語もいろいろある。

2 プログラミング言語の諸側面

- □ プログラム→「人間が書くもの」だが、プログラム以前には無かったさまざまな特徴/特性がある
  - 個別のプログラミング言語についても、その特徴/特性についていろいろな選択肢がある→以下で検討

2.1 プログラムが持つ特徴

- □ 簡単な C プログラムを題材に…

```
int main() {
  int x, y;
  printf("input x> "); scanf("%d", &x);
  printf("input y> "); scanf("%d", &y);
  if(x > y)
    printf("%d is larger.\n", x);
  else
    printf("%d is larger.\n", y);
  return 0;
}
```

  - Q. このプログラムは何をするプログラムですか?
  - Q. このプログラムにはどんな問題がありますか?
  - Q. あなたが認識した「問題」を解決する方法は?
- □ 問題かどうかはあくまで目的 (仕様) に照らさなければ分からない
  - 仕様: 「x と y が等しければ何が出力されてもよい」だったら?
  - 仕様: 「x と y に等しい数を入れてはならない」だったら?
  - 仕様: 「x と y に等しい数が入ることはあり得ない」だったら?
- □ 動作はきっちり決まっている
  - 言語仕様としてきちんと決まっている部分

- 言語仕様で決まっていなくても、動かせば分かる (???)

□ それが「プログラマの意図」さらには「ソフトの仕様」と一致しているかどうかはまた全然別の問題
□ 1 つの動作を行うプログラムは何通りも書ける

```
/* A */
if(x > y)
  printf("%d is larger.\n", x);
else
  printf("%d is larger.\n", y);

/* B */
if(x > y) z = x; else z = y;
printf("%d is larger.\n", z);

/* C */
z = (x > y) ? x : y;
printf("%d is larger.\n", z);

/* D */
printf("%d is larger.\n", (x>y)?x:y));

/* E */
z = x;
if(y > z) z = y;
printf("%d is larger.\n", z);
```

- どれがいいと思うか? それはなぜか?

□ もし 3 つの値 x、y、z の最大だったら?

```
/* D' */
printf("%d is larger.\n",
       (x>y)?((x>z)?x:z):((y>z)?y:z));

/* E' */
max = x;
if(y > max) max = y;
if(z > max) max = z;
printf("%d is larger.\n", max);
```

- こんどはどれがいいか?

□ 1 つの動作を行うプログラムは何通りも書ける

- そのどれがいいかは様々な側面を総合して決めるしかない
- プログラミング言語の機能やデザインもまさにそう!

2.2 プログラミング言語が持つ「要素」「概念」「機能」

□ また同じ C プログラムを題材に…

```
int main() {
  int x, y;
  printf("input x> "); scanf("%d", &x);
  printf("input y> "); scanf("%d", &y);
  if(x > y)
    printf("%d is larger.\n", x);
  else
    printf("%d is larger.\n", y);
  return 0;
}
```

- これにどのような「要素」「概念」「機能」が含まれている?

□ たとえば…

- 順次実行 (A; B; C)
- 制御構造 (if、while、…)
- 変数
- 型 (値の種別)
- アドレス (ポインタ)
- 関数 (サブルーチン)
- ...

□ これらがどういう意味を持っているか考えて見よう…

2.3 実行順序

□ 「文が順次実行される」という考え方は「当然」か?

```
x = y * z;
y = 3.1416 + 1;
z = 2.7183 + 2;
```

- どういう順番で実行されると「感じられる」?

□ 方程式であれば、「参照関係に従って」計算する。
□ 関数型言語などでも同様。

```
double x() { return y() + z(); }
double y() { return 3.1416 + 1; }
double z() { return 2.7183 + 2; }
```

□ 再帰と組み合わせることで複雑な計算でも可能

```
double fact(n) {
  return (n < 1) ? 1 : n * fact(n-1);
}
```

□ ではなぜ現在の多くの言語は「順次実行」なのか?

- 手続き型言語のモデル←現在の計算機のモデル
- 現在の CPU は「命令を順番に実行して行く」モデルに従っている (実際には全然順番に実行していないことが多いが…)
- プログラム言語でもこれにならった方が自然 (?)

2.4 制御構造

□ 「枝分かれ」「繰り返し」などの*実行順序*を表す機構

```
if(x > y) {
  z = x; x = y; y = z;
}

while(n > 0) {
  fact = fact * n; n = n - 1;
}
```

- 実はこのように「制御構造の中に文の集まりが入る」という考え方が一般的になったのは 1970 年代の「構造化プログラミング運動」の結果。
- それ以前はこんな便利な (?) ものはなかった。

□ Fotran 60 (JIS Fortran 5000/7000)

```
   IF(X .LE. Y) GOTO 10
     Z = X
     X = Y
     Y = Z
10 CONTINUE

30 IF(N .LE. 0) GOTO 20
     FACT = FACT * N
     N = N * 1
     GOTO 30
20 CONTINUE
```

□ JIS 3000

```
   IF(X - Y) 20, 10, 10
20   Z = X
     X = Y
     Y = Z
10 CONTINUE
```

□ 構造化プログラミング運動

- GOTO はスパゲティプログラムになるからやめよう
- 構造化構文 (今の if や while) の入れ子を使おう

□ しかし「入れ子」がいいかどうかも疑問はある

```
if(a[low] < a[high]) {
  for(i = 0; i < high-low; ++i)
    for(j = low; j < high; ++j)
      if(a[j] > a[j+1]) {
        z = a[j]; a[j] = a[j+1]; a[j+1] = z;
      }
} else {
  for(i = 0; i < high-low; ++i)
    for(j = high; j > low; --j)
      if(a[j-1] < a[j]) {
        z = a[j-1]; a[j-1] = a[j]; a[j] = z;
      }
}
```

- このコードが何をしているか読めるか?
- 読めるようになったとすれば「どう考えた」から?

□ 「入れ子」構造は、ある部分 (たとえば「文」が入る部分) に、複雑な構造であっても 1 つの単位として互換性があれば入れられる、という思想によっている。

- 計算機による処理は容易 (やり方が分っていれば)。
- しかし人間の頭はそういう風にはできていない。
- →プログラムの書き手としては、「人間に扱えるように」書く方がよい。
- →具体的にはどうする?

□ 「if」と「switch」と「while/for」と「do-while」でいいのか? もっとあった方がいいのか? 減らした方がいいのか? (yes/no)

□ 現在あるもので「一応」足りているが「まだ」あった方がよいかも。

- 定理: すべての (含むスパゲティ) プログラムは連接、分岐 (if)、反復 (while) の組合せに書き換えることができる。
- しかし「言語としての使いやすさ」が問題だからもっとあってもよい。
- 後述する「例外」などは新しい制御構造のバリエーション。

2.5 変数

□ 変数とは、何ですか? (知らない人に説明するとしたら?)

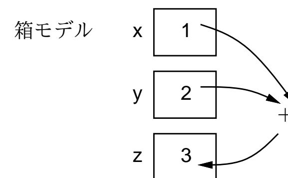
□ 変数の 2 つのモデル (どっちがいい?)

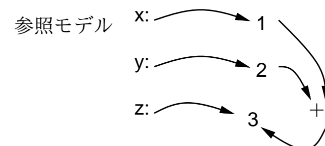
- 説明 A: 値を入れておく「箱」のようなもの。
- 説明 B: 値を表す名前 (値に名前をつけたもの)。

□ 「箱」のモデル…手続き型言語のモデル

- 手続き型言語…Fotran、COBOL、Pascal、C、C++、Java、…
- 変数に代入して値を書き換えて行く
- 計算機ハードウェア (メモリ) の自然な抽象化になっている
- 副作用ベース→分かりにくいバグの原因となることがある

```
while(i > 0) {
  fact = fact * i; i = i = 1;
}
```





□ 「値を表す名前」のモデル…数学に近い感じ

- 関数型/論理型言語…ML、Haskell、Prolog、…
- 変数には値が「束縛」される
- 一度束縛された値が書き変わることはない (単一代入とも言う)

- ただし変数は複数のインカーネーション (実体) を持つことも…

```
int fact(int n) {
  return (n < 1) ? 1 : n * fact(n-1);
}

fact(n:5)
↓ 5 * fact(n:4)
↓ 5 * 4 * fact(n:3)
↓ 5 * 4 * 3 * fact(n:2)
↓ 5 * 4 * 3 * 2 * fact(n:1)
↓ 5 * 4 * 3 * 2 * 1 * fact(n:0)
↓ 5 * 4 * 3 * 2 * 1 * 1
```

□ 結局: どちらのモデルも言語として成り立つ

- 優劣は一概に言えない (言語どうしで優劣が言えないのと同様)

2.6 型 (= 値の種別)

□ 「int x」「double x」「char *x」

- 何のために「型」があるのか?

□ さまざまな「型がある理由」(A)

- 入れる「箱」の大きさが種別によって変わるから。
- 種別ごとに演算 (演算命令) が違うから
- →これらは言語の設計次第でどうにでもなる。
- →ただし型がある方が効率はよい
- 型のない言語 (弱い型の言語) → Lisp、Smalltalk、Perl、…

□ さまざまな「型がある理由」(B)

- プログラマに「ここはどういう種類の値」という情報を書かせたい
- おかしなデータの使い型をチェックして教えたい
- →「よりよくプログラムを書くための道具」としての型
- 強い型の言語 → Pascal、C、C++、Java、ML、Haskell、…
- 型の明示 (変数宣言に必ず型を書く) と型推論 (「x = 10」なら x は int、のように使用関係に基づいて決定) とがある。

□ その他の区分

- 型には単純な型 (「整数」「実数」「文字」…基本型) と込み入った型 (「レコード」「配列」…複合型) がある (後述)。
- 「標準で用意されている型」「ユーザ定義の型」という区別があることもある。
- できるだけどんな型でも同じように区別なく使える方が望ましい。が。
- 例: C++の演算子定義→ユーザ定義の型でも演算子が使える。が、そのために言語的には複雑になっている。

2.7 さまざまな型

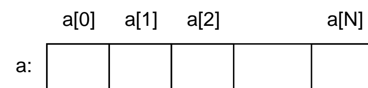
□ 基本型…数値 (整数、実数)、論理値、文字値など。

- 定義? 「直接 CPU が演算できるような値の型」「マシンレジスタに載るような値の型」「それ以上分解できない値の型」(どれもよくない)
- 結局、言語仕様として「基本型に決めたから基本型」みたいな…
- 基本型の値を書く手段として、リテラル (定数) が用意されている。(基本型でない場合も一部用意されていることはある。)

□ 複合型…基本型でない型すべて。複数の値を含む…かどうか?

□ 配列型…もっとも古くからある複合型 (ベクトルや行列演算のため)

- 同じ型の値が連続して並んでいる
- 添字により「何番目」を指定してアクセスする
- 実現上は単にメモリ上に必要なだけの領域を並べるのが基本 (だった)



- しかしそれでは不便だと分かってきたので…(どう不便?) →連想配列
- 型としては「T 型の配列」という型→ T は「パラメタ」になっている。
- さらに要素数 (ないし、添字の上限/下限) も型のパラメタとする言語もある (Pascal とか)

□ レコード/可変レコード/ユニオン型

- こちらは COBOL が元祖。複数の「違う型の」値を組み合わせたもの
- 非常に基本的な型だった…はずが、オブジェクト指向のせいで影が薄く…
- Java にはレコード型はない。C++ではクラスの特別なカタチ。
- 可変レコード…レコードで「途中からフィールドを取り換える」ようにした← Pascal
- これを「取り換えずに並べるだけ」と「取り換えるだけ」に分離→ユニオン型← C

2.8 ポインタ型

- □ ポインタ型…とは? C言語の式「&x」の結果とは?
  - 答A: 変数xの主記憶上の番地(アドレス)
  - 答B: 変数xを「指す」(参照する)何らかの値
  - 答C: その他…
- □ 言語仕様的には「B」。
  - 言語仕様に「番地」は出てこない(実装に依存しすぎる)。
  - int *p = …; … p + 1 … ←実際に足されるのは「1」ではない
  - それはいいが、なぜこういうもの(ポインタないし参照)が必要なのか?
- □ ポインタは何のために必要?
  - 変数に値を書き込んで戻してもらうため←言語に参照渡しがない場合
  - 動的データ構造(連結リスト、2分木、グラフ、…)
  - 大きなデータを持ち回る非効率を避けたい場合
  - データを複数個所で共有したい場合(cf. 広域変数)
- □ データの共有は副作用を可能にする→注意が必要
- □ 言語設計上の問題点
  - 任意の変数のポインタを取れるのは危険(なくなった後も…)
  - C/C++→配列の添字式がポインタ演算(「a[i]」は「*(a+i)」と同じ意味)
  - aが配列、iが整数として「a[i]」でも「i[a]」でも同じって知ってます?
  - 非常に特異な言語設計、問題山積み(添字検査困難とか)。
- □ 型という点では「T型へのポインタ」→パラメタつきの型。
  - しかし値そのものは「レジスタに載るような値」
  - 型Tと型*Tを使い分けるのは混乱のもと(C++だとさらに&Tもある)
  - すべての値をポインタとするのも1つの選択肢(Javaがこれに近い)
- □ 古い言語には「ポインタ型」が1種類しかない言語もあった(PL/I)。
  - どんな型の変数でもアドレスを取ると同じ型→型安全でない。
  - しかしCでもキャストすれば何型のポインタにでもできる→安全でないことに変わりはない。ただ、キャストを書くことで「分かっていてやっている」という意志表示にはなる。

2.9 関数型/関数へのポインタ型

- □ 関数を指すポインタ→複数の関数のどれを呼ぶか選択可能に
  - 関数(サブルーチン)は呼ぶ以外の使い方はできない→わざわざ「〜のポインタ」としなくてもいい→そういう言語もある。
  - Cでは「関数名を自動的にその関数を参照するポインタ値に変換」

```
int myfunc(int x) { return x*x; }
int (*f)(int) = &myfunc;   ←○
int (*g)(int) = myfunc;   ←これも○
```

  - Cでは型名や変数宣言の書き方がすごく分かりづらい…
- □ 引数の個数と型、および関数が返す型も関数型の一部(パラメタ)
  - 「proctype(int,int)returns(int)」← CLU流の記法
  - 「(int(*)(int,int))」←Cのキャスト等で書く型指定…分かりづらい
- □ ただしCでは「int (*f)()」のようにパラメタ部分を書かないことで「引数の個数と型を指定しない」ことが可能→安全でない
  - または引数のどこかから後を「...」と指定できる(可変引数)
  - 引数を「持たない」場合は「int (*f)(void)」と指定 ← 注記: C++では「int (*f)()」でも同じ。Cでは「int (*f)()」だと「パラメタは何でもよい」になってしまう。

2.10 強い型と弱い型

- □ ここまでおもに「強い型の言語」(=コンパイル時にチェックする)について考えて来た。
  - 弱い型の言語(=実行時にチェックする)→ Lisp、Smalltalk、Perl、Ruby他スクリプト言語
  - 弱い型の言語はチェックが抜けることはない(実行時に逐一チェックするから)→安全ではあるが、速度は犠牲になる
  - 型のエラーがあってもそこを実行してみるまで間違いがあることが分からない→製品を作る上では弱点
  - 強い型であれば、コンパイル時にチェックできる→実行時には安全であることが保証される(ただし最近のオブジェクト指向言語では実行時のチェックにたよる部分もある)。

2.11 この節のまとめ

□ プログラム言語/プログラムが持つ特徴

- 厳密だが「何が正しいか」は難しい (仕様の問題)
- 1 つのことを何通りにも書ける

□ プログラム言語が持つ多くの概念

- 順次実行、制御構造、変数、型、…

□ 型→データの種別、対応する演算の種別

- 型安全性→間違った操作が起こらない (突然死なない)
- 強い型→コンパイル時にチェックできるようにする

3 命令型からオブジェクト指向へ

□ 命令型言語 (imperative language) → C のような、順次実行と変数の書き換えに基づくプログラミング言語

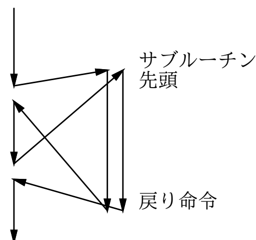
- オブジェクト指向言語はその発展形と考えてよい
- どのような経緯でオブジェクト指向言語ができたか?
- 細かい話題は後まわしにして、ここでは全体的な概観

3.1 関数 (サブルーチン)

□ 関数 (サブルーチン) とは、何ですか?

□ 現象的には、「ある範囲のコードの先頭にジャンプし、そのコードを実行し、終わったら元の (ジャンプする前の) ところに戻って来る」

- 実装的には戻り番地 (ジャンプした命令の次の命令の番地) を保存しておいてそこへ間接ジャンプで戻る
- しかしそれが関数 (サブルーチン) だと言われても釈然としないでしょう?



□ プログラマから見ると→

- ひとまとまりのコードに名前をつけたもの
- その名前を指定して呼び出せる
- それぞれが完結した何らかの仕事をしてくれる (その細部は呼び側では知らなくてもよい)

□ 要するに「抽象化の手段」→最も重要な機能

- さっきの「入れ子をどうするか」の答えも第 1 義的にはここにある。
- 「いちど関数を書いてしまえば、言語にそういう命令が増えたものとして扱える」

□ 関数の利用が普及したのは構造化プログラミングの時期以降 (1970〜)

- それ以前だと「1 万行のメインプログラムだけ」とか珍しくなかった

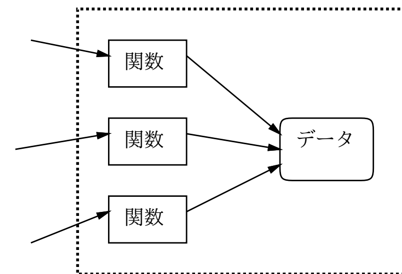
□ 関数の弱点→かたまりになるのはあくまでも「コード」だけという点→データの共有 (呼び側と呼ばれ側、呼ばれ側どうし) は?

- パラメタとして逐一渡す→繁雑、渡す側でもいじれてしまう
- グローバル変数として共有→どこでもいじれるから一層危険
- →結局、関数だけを「部品」として用いると複雑さに限界。← 1 レベルであること、データの共有問題

3.2 モジュール

□ モジュール以前の言語→手続きが単位

- 手続きが単位だと「カタマリ」が小さすぎる
- 複数の手続きが 1 つのデータ構造を共有する→よくあるパターン



□ そのデータ構造は「グローバル変数」じゃいけないの?

□ グローバルだと「よそからいじられる」可能性

- データ構造→通常、「こういう性質を維持」という条件がある (整合性条件) →これが壊れるとそれを前提としているコードも破綻
- グローバルだとそれを分からずにいじられてしまう

□ モジュール機構があれば、データ構造は特定のコードからだけしかいじれなくできる

- 整合性条件の維持→そのデータをいじるコードが限定されていると、そこだけきちんとすれば確実に大丈夫

- さらに、外から直接呼ばれたくない関数も同様にして限定できるとよい。

□ どうやって実装するか?　言語にモジュールがあれば簡単だが、C 言語でも「ファイル内の static」を使ってある程度できる

□ 例:　「図形描画できるウィンドウ」

```
---GraphWin.c---
#include <なんとか.h>
static Window *win;
static struct { int x1,y1,x2,y2 } line[1000];
static int linecount;
static int initialized = 0;
static int init() {
  win = ...
  linecount = 0;
}
int addline(int x1, int y1, int x2, int y2) {
  if(!initialized) init();
  line[lincount].x1 = x1; line[lincount].y1 = y1;
  line[lincount].x2 = x2; line[lincount].y2 = y2;
  ++linecount;
}
int update() {
  int i;
  if(!initialized) init();
  for(i = 0; i < linecount; ++i) 線 i を描く;
}
...
```

□ このコードでは「どういう整合条件」があると思うか?

□ こうしておくと他にどういういいことがあるか?

□ 結局、モジュールの機能は「何もしなければアクセスできるものを隠す」こと→カプセル化 (encapsulation)

- 「わざわざ不自由にすること」が実は重要

3.3　抽象データ型

□ モジュールでは隠す対象は「データ一式」

□ では窓を沢山作りたければどうする?　→抽象データ型

```
---GraphWin.c---
#include <なんとか.h>
typedef struct {
  Window *win;
  struct { int x1,y1,x2,y2 } line[1000];
  int linecount; } GraphWin;

GrappWin* create_gw() {
  GraphWin *gw = (GraphWin*)malloc(siezof(GraphWin));
  gw->win = ...
  gw->linecount = 0;
  return gw;
}
int gw_addline(GraphWin *gw,
      int x1, int y1, int x2, int y2) {
  gw->line[lincount].x1 = x1;
  gw->line[lincount].y1 = y1;
  gw->line[lincount].x2 = x2;
  gw->line[lincount].y2 = y2;
  ++gw->linecount;
}
int gw_update(GraphWin *gw) {
  int i;
  for(i = 0; i < gw->linecount; ++i) 線 i を描く;
}
...
```

□ なぜこれが「抽象データ型」かというと…GraphWin 型というデータ型を新しく作っていると考えられる。たとえば利用側は:

```
---GraphWin.h---
typedef void *GraphWin;
GraphWin create_gw();
int gw_addline(GraphWin gw, int x1, int y1, int x2, int y2);
int gw_update(GraphpWin gw);
...

---アプリケーション.c---
include <GraphWin.h>
int main() {
  GraphWin gw = create_gw();
  ...
  gw_addline(gw, ...);
  ...
  gw_update(gw);
  ...
}
```

□ この「2 回ずつ言う」あたりを何とかしたのがオブジェクト指向の「メッセージ送信記法」と考えてもらえればよい。

3.4　オブジェクト指向

□ いちいち上のようにヘッダファイルを使い分けたり、第 1 引数でデータ構造のポインタを渡したりすると煩雑→間違い→破綻。

□ これを言語の方で自動的にやってくれると嬉しい

```
class GraphWin {
  Window win;
  Line[] line = new Line[100];
  int linecount;
  public GraphWin() {
    win = ...;
    linecount = 0;
  }
  public addLine(int x1, int y1, int x2, int y2) {
    line[linecount++] = new Line(x1,y1,x2,y2);
  }
  public upate() {
    for(int i = 0; i < linecount; ++i) 線を描く…
  }
  ...
  class Line {
    int x1, y1, x2, y2;
    public Line(int px1, py1, px2, py2) {
      x1 = px1; y1 = py1; x2 = px2; y2 = py2;
    }
```

```
    public int getX1() { return x1; }
    public int getY1() { return y1; }
    public int getX2() { return x2; }
    public int getY2() { return y2; }
  }
}
```

□ 使う側では次のようになる

```
GraphWin gw = new GraphWin();
...
gw.addLine(...);
...
gw.update();
```

- 初期設定とかポインタとかに煩わされなくなった
- メッセージ送信記法→「馬から落馬」がなくなった

□ ところで、上の例は C++ですか? Java ですか?

□ 答: 上の例は Java。

- C++では new はポインタを返すが、こういう場合はポインタを使う必要はない
- C++では「その場に」(ローカル変数ならスタック上などに) オブジェクトが作れるように工夫されている

  Line[] line = new Line[100]; //Java
  Line line[100]; //C++、ただし vector<Line>とかの方がよい

  ```
  GraphWin gw = new GraphWin(); //Java
  GraphWin gw = GraphWin(); //C++
  ```

- 少し後で C++と Java を題材にもう少し具体的にやりますがその前に…

3.5 抽象データ型再訪

□ 抽象データ型 (Abstract Data Types, ADT) とは…

- 「型」とそれが持つ「操作群」を定義する。
- その型の「内部構造」や「操作の実現」は外部からは隠蔽
- たとえば「+」や「- 」などの演算を持つ「整数型」のようなもの (例:ベクトル型とか) を自前で定義できる。
- 初期の ADT 言語: CLU、Alphard、…

□ ADT の何が重要か?

- 内部構造や実現を外部から見せない (カプセル化) →モジュール間相互の依存関係を軽減。
- オブジェクト指向言語も「単純な」クラス (上に挙げた例のようなもの) は単なる ADT を実現している。ADT 的な考え方は重要 (それなりに慣れが必要)。

3.6 この節のまとめ

□ プログラム言語の重要な機能→「抽象化」

- ひらたくいえば「かたまり」の作り方
- 関数 (サブルーチン) →データはかたまりにできないのでいまいち
- モジュール→関数+データのパッケージ。より大きなかたまり
- 抽象データ型 (ADT) →データ「型」として沢山使える
- オブジェクト指向→ ADT+メッセージ送信記法+ α

4 オブジェクト指向言語入門

□ 本講座では実際にオブジェクト指向言語を使ってレポートをやって頂きたい→具体的な言語の紹介も必要

- 言語として「C++」「Java」の両方を取り上げる
- 例題・課題ともできるだけ両方を対比させて用意する
- ただし題材によってはどちらか片方だけになるものもある

4.1 C++入門

□ C++ --- Bjane Stroustrup によって、「C with class」として始まった

- 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

□ C++の処理系は…

- Windows 用…Borland C++、MS Visulal C++など色々ある
- フリーなもの…GNU C++ (G++)。Unix (Linux) には普通ついている。Windows なら Cygwin を入れればそこに一緒にいれて動かせる (資料末尾も参照)

4.2 Java 入門

□ 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 Sample1 {
  public static void main(String[] args) throws Exception {
    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));
  }
}
```

4.3 オブジェクト指向言語

□ オブジェクト指向言語とは?→いろいろな定義があると思うがここでは一応次のように考える。

- 「オブジェクト指向」→プログラムが扱う対象をそれぞれ自律的な/完結した「もの」(オブジェクト) であるとする「考え方」
- 「オブジェクト指向言語」→オブジェクト指向の考え方をサポートするようなプログラミング言語

□ 抽象的でよく分からない?　まあそうだと思いますが…

```
outstream ostream = ...;
outstream_println(ostream, "Hello, World.\n");
↑関数名が長い    ↑操作対象も引数 : 従来のスタイル

outstream ostream = ...;
ostream.println("Hello, World.\n");
↑オブジェクト. メソッド (引数…) : オブジェクト指向ぽい
```

- 同じことを 2 度言わなくて済む感じ
- 同じことには同じ名前が使える
- 名前空間の節約

□ あと、とりあえず関連する用語を紹介しておく

□ 「クラス」→オブジェクトの種類に対応する構文要素。

- 新しい種類のオブジェクトを定義したければ、クラスを書く。

□ クラス定義に含まれるもの…

- オブジェクトはどういう動作 (メソッド、メンバ関数) を持っているか
- オブジェクトはどういう属性を持っているか←インスタンス変数、メンバ変数
- そのほかクラスはモジュールとしての役割りも←クラスメソッド変数、static メソッド/変数

□ 「インスタンス」→クラス定義に基づいて作り出された「もの」(オブジェクト)

□ 逆に見ると→「新しいモノ」を定義できるのがオブジェクト指向

- 世の中はさまざまなモノから成り立っている→それをそのまま言語に移せる (プログラムだから現実世界よりは杓子定規だけど…) →考えやすい (考えやすさは重要)

□ どういうモノを作るか考える→オブジェクト指向における設計

- (1) どんなモノを作るか
- (2) それぞれのモノはどういう操作/インタフェースを持つか
- (3) その実装はどうするか←ある意味どうでもいい/別建て
- (上記はADTの考え方と言った方が正確← ADTは重要)

4.4 整数を出し入れするバッファ

□ 非常に簡潔な「もの」を例として取り上げて具体例を見てみましょう。

□ 数を「ある決まった規則で」保持するバッファというものを作ります。

- 数を入れる→「buf.put(値)」
- 数を取り出す→「x = buf.get()」

□ Mybuf1は次の規則により保持するものとします。

- 「最近に入れた2つの数を保持する (初期値は0)。1回取り出すと、一番最後に入れた値が出て来る。もう1回以上取り出すと、それより1つ前に入れた値が出て来る」。

□ C++版のMybuf1

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

class Mybuf1 { // memorize last two
  int val1, val2;
public:
  Mybuf1();
  void put(int x);
  int get();
};

Mybuf1::Mybuf1() {
  val1 = val2 = 0;
}
void Mybuf1::put(int x) {
  val2 = val1; val1 = x;
}
int Mybuf1::get() {
  int x = val1; val1 = val2;
  return x;
}

int main(void) {
  Mybuf1 buf;
  while(true) {
    char c; int v;
    cout << "? "; cin >> c;
    switch(c) {
case 'q': return 0;
case 'p': cin >> v; buf.put(v); break;
case 'g': cout << buf.get() << '\n'; break;
    }
  }
}
```

□ クラス定義部分

- 「class クラス名 { ⋯ } ;」がクラスを定義 (最後に「;」があることに注意!)
- public: より前はprivate部分→クラス外からは見えない (コンパイラは見るけれど) →カプセル化、ADTの保護
- 変数:「この型のデータは何と何を組みにしたものか」を表す
- public: 以下にメソッド (メンバ関数) のインタフェースを書く→これをヘッダファイルとして取り込むことで型検査が可能に
- クラス名と同名のメソッド→「コンストラクタ」(初期化用の特別なメソッド)

□ メソッド定義部分

- 実際に各メソッド (メンバ関数) 本体のコードを書く。
- 各クラスに属する名前 (変数、メソッド) は「クラス名::名前」が正式な (他と厳密に区別できる) 名前になる ← 複数のクラスが同名のメソッドや変数を持てるから当然
- 各クラスのコード中では「クラス名::」は省略できる

□ メイン部分

- 1文字読み取ってその値 (コマンド) に従って put、get、終了のいずれかの動作を行う

□ Java版のMybuf1

```
import java.util.*;

public class MybufTest1 {
  public static void main(String[] args) {
    Mybuf1 buf = new Mybuf1();
    Scanner sc = new Scanner(System.in);
    while(true) {
      System.out.print("? ");
      char c = sc.next().charAt(0);
      switch(c) {
case 'q': return;
case 'p': buf.put(sc.nextInt()); break;
case 'g': System.out.println(buf.get());
      }
    }
  }
}

class Mybuf1 {
  int val1, val2;
  public Mybuf1() {
    val1 = val2 = 0;
  }
```

```
  public void put(int x) {
    val2 = val1; val1 = x;
  }
  public int get() {
    int x = val1; val1 = val2;
    return x;
  }
}
```

□ メイン部分

- Scanner オブジェクトは入力時に便利な機能を提供
- sc.next() は「1 単語」読む。その先頭の文字を charAt(0) で取り出し、使用する。(Scanner には 1 文字だけ読むというメソッドがないためちょっと不便。)

□ クラス定義部分

- Java ではヘッダと本体を分けるみたいな面倒くささはないし、前方参照も OK(やはり設計が新しい…)
- 「class クラス名 { … }」の内側に変数もメソッドも書いてそれでおしまい
- コンストラクタがクラスと同名というのは C++と同じ
- クラス外から取れるものはそれぞれ public 修飾子をつける

□ では練習問題をやってみましょう。C++でも Java でも好きな方で (ただしできるだけ普段使っていない言語で) 作成してください。

- 練習 1: 「最近 3 つの値を覚えるバッファ」を Mybuf2 という名前で作りなさい。
- 練習 2: 「最後に入れた値を覚えるが、取り出すごとに覚えている値が 1 つずつ減るバッファ」を Mybuf3 という名前で作りなさい。
- 練習 3: 「入れた値が累計され、取り出すとその累計値が取れるバッファ」を Mybuf4 という名前で作りなさい。

□ C++はこういう感じ

```
(冒頭省略)

class Mybuf2 {
  (インスタンス変数をここで定義)
public:
  Mybuf2();
  void put(int x);
  int get();
};

Mybuf2::Mybuf2() {
  (初期化動作を記述)
}
void Mybuf2::put(int x) {
  (入れる動作を記述)
}
int Mybuf2::get() {
  (x に取り出す動作を記述)
  return x;
}

(main は同様なので省略)
```

□ Java はこういう感じ

```
(冒頭+main 部分省略)

class Mybuf2 {
  (インスタンス変数の定義)

  public Mybuf1() {
    (初期化動作を記述)
  }
  public void put(int x) {
    (入れる動作を記述)
  }
  public int get() {
    (x に取り出す動作を記述)
    return x;
  }
}
```

4.5 分数電卓:C++版

□ 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。「operator △」というメソッドを定義するとそれは「△」という演算子として呼べる。
- 今度は個々のオブジェクトは「有理数」というものを表すので、その状態は変化しない→演算においてパラメタもオブジェクト自身も変化しないことを const で表している
- 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
```

4.6 分数:Java 版

□ 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 Sample3 {
  public static void main(String[] args)
      throws Exception {
    Scanner sc = new Scanner(System.in);
    Rational r = new Rational(1);
    while(true) {
      System.out.print("? ");
      String s = sc.nextLine();
      if(s.charAt(0) == 'q') return;
      Rational x = new Rational(s.substring(1));
      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);
    }
  }
}
```

- こちらはコンストラクタに文字列を渡せるように作った。文字列から 1 つまたは 2 つの数値を読み取るのに Scanner を使用
- 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(String s) {
    try {
      Scanner sc = new Scanner(s);
      a = sc.nextInt();
      if(sc.hasNextInt()) b = sc.nextInt();
      else                b = 1;
    } catch(Exception ex) { }
  }
  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)。

5 第1回レポート課題

□ 下記 (1)～(4) のうちから 1 つ以上課題を選んで製作や実験を行い、結果をレポートせよ。やったことの説明や書いたコードだけでなく、計測の場合は方法まで説明し、きちんと考察まで書くこと。言語は特に指定していない場合、C++でも Java でもよい (両方でやって比べてみるとなおよい)。期限はリーダから指定。

□ (1) Mybuf の練習問題について、さらに次のものから興味を持ったものを 1 つ以上作ってみなさい。

- a. 最近 10 個のものを覚えている。取り出すごとに 1 つ前の値に遡ることができる (10 回戻るとそれ以上は同じ値になる)。
- b. 入れた値のうち最近 10 個の値の合計を覚えている。取り出すごとに 1 つ前の時点での 10 個の合計に戻ることができる (10 回戻るとそれ以上は同じ値になる)。
- c-d. 上の a と b の「10」それぞれの代わりにサイズ N をコンストラクタで指定可能にしたもの。

□ (2) 課題 (1) で作成した容量 10 ないし N のバージョンについて、演習問題でやった単純なものと比べてどれだけ速度が遅いか検討しなさい。それより速くできる可能性があるかも検討すること。(実際に速くしてみられるとなおよいです。)

- ヒント: C++で時間計測をするには標準関数 clock() を利用するとよい。

```
#include <ctime>
...
clock_t t1 = clock();
計測したい処理
clock_t t2 = clock();
double sec = double(t2-t1)/CLOCKS_PER_SEC;
        ↑秒単位に換算
```

- ヒント: Java の時間計測には System クラスのクラスメソッド System.currentTimeMillis() を利用するとよい。

```
long t1 = System.currentTimeMillis();
計測したい処理
long t2 = System.currentTimeMillis();
long dt = t2 - t1; ←所要時間 (msec)
```

□ (3) Rational オブジェクトの例題を動かし、その「足し算」などの演算が int や double などの数値の演算の何倍遅いか計測してみよ。計測する前に予測し、予測がどれくらい合っていたか検討すること。(予測はあてずっぽうではなく、何らかの根拠が必要)。

□ (4) Rational オブジェクトの実装にはいろいろと「無駄」がある。どこに無駄があるかを検討し、そこを改良して速くなったかどうか計測し確認してみなさい。どれだけ速くなるかきちんと予測すること。

6 実習のための言語処理系

□ Java --- Sun からフリーでダウンロードできます。

http://java.sun.com/javase/ja/6/download.html

ここから「JDK 6 Update 20」を取得してください。Windows、Solaris、Linux 用があります。NetBeans は別にいらないです (統合開発環境なので入れてみたければどうぞ)。

□ C++ --- Visual C++とかは売りものですが、Linux や Mac OS-X なら g++が入っています。また、Windows でも cygwin を入れて、その時「Devel」(developer) パッケージを選択しておけば入ります。入れる時の操作の解説はここが分かりやすいです。

http://sohda.net/cygwin/setup.html

□ JavaScript --- IE に「互換品」である JScript が付属していますが、プログラムを動かすときは FireFox の JavaScript コンソール (Error コンソール) があると便利ですので、ぜひこの際入れてみてください。

http://www.mozilla.org/