基礎プログラミングおよび演習 2018

久野　靖
電気通信大学

2019.2.28 改訂

目 次

＃1　プログラム入門+様々な誤差

今回は初回ですが、次のことが目標となります。

- プログラムとは何であるか理解し、簡単な Ruby のプログラムを動かせるようになる。
- コンピュータによる実数計算の特性を理解し、さまざまな原因の誤差について判別できる。

ただしその前にガイダンスから始めて、その後本題に入ります。

1.0　ガイダンス

1.0.1　本科目の主題・目標・運用

本科目の主題ならびに達成目標は次のようになっています。

主題: コンピュータは、ソフトウェア (プログラム) によっていろいろな機能を実現している。将来、自分でプログラムを作ることがないとしても、コンピュータに関わることは避けられない。プログラムがどのように作られているかを知っていることは大変重要である。本授業では、新たな機能を実現するための方法論として、プログラミングの基礎を学ぶ.

達成目標: プログラミングに必要な基礎知識を Ruby 言語および C 言語を用いて習得し、簡単なプログラムの作成と読解ができるようになること、および、基礎的なアルゴリズムの理解や、ソフトウェアの開発方法を理解し、問題解決の基盤となる思考能力を身に付けることを目標とする。

本学では授業 1 単位について 45 時間の学修を必要とすることとなっています。本科目は 2 単位ですから 90 時間となります。これを 15 週で割ると週あたり 6 時間となります。授業そのものは 90 分 (1.5 時間) であるので、時間外に 4.5 時間の学修が必要です。課題等もこのことを前提に用意されていますので、留意してください。

本科目の運用ですが、各時間の内容は前もって Web で公開しますので、予習してくるようにお願いします。授業時間中は予習時に分からなかったことの質問を受けて捕捉説明を行い、あとはコンピュータ上での演習が中心となります。

LMS の「授業開始 5 分前以後の」演習室でのログインを用いて出席を把握します。授業を十分に受けていない方は個別判断の上、成績評価を行わない可能性があります。

各時の終了 10 分後までに、LMS を使用して当該時間の演習内容をレポートとして提出していただきますが (A 課題)、これは演習内容を振り返って頂くためのもので、成績には関係しません。

そして、上記とは別の課題に対する解答レポートを、次回授業の前日までに提出頂きます (B 課題)。B 課題は担当教員が次の 3 段階 (未提出は別扱い) で評価します (大文字:通常回、小文字:総合課題回)。

C/c — 課された課題に対して不十分な内容である (ないし遅刻提出)

B/b — 課された課題に対する解答として適切である

A/a — 課された課題に対する解答として優れている点がある

評価は「B/b」が通常ですが、担当教員の判断によって C/c、A/a(極少数) をつけることもあります。成績評価は課題点と試験点を 1:1 で合わせたものとしますが、課題点はすべて「B/b」のとき満点とし、「A/a」の場合は上積み (ないし C/c の減点を相殺) となります。期限に遅れた場合も一定期間は遅刻提出できますが (期末を除く)、その場合 C/c の採点となります。

成績評価は課題点と試験点を 1:1 で合わせたものとしますが、課題点はすべて「B/b」のとき満点とし、それより良い採点の場合は上積み (ないし C/c の減点を相殺) となります。

1.0.2　担当教員・情報部会との連絡

この科目の連絡は LMS 上の「全クラス共通アナウンスメント」「クラス内アナウンスメント」で行ないます。また質問等は「クラス用フォーラム」に書き込んでください。授業期間中は、これらを少なくとも 2 日に 1 回程度チェック願います (休講などがありそうな場合は 1 日に 2 回くらいお願いします)。**掲示を見ていなかったことによる不利益は救済しませんから**、十分注意願います。

1.0.3　プログラミングを学ぶ理由・学び方・使用言語

本科目ではプログラミングを学びますが、なぜこのことが必要なのでしょうか。それは、プログラミングを学ぶことではじめて、コンピュータとは何であり、何ができるかが分かるからです。

世の中の多くの人は、「特定のソフトウェアの機能や操作方法」を個別に学びますが、それだと別のソフトウェアに対面した時や、悪くすると同じソフトウェアの新バージョンに対面した時に、これまでの知識が通用しないことになります。これに対し、プログラミングから学ぶことによって、ソフトウェアに対するより一般的な「古くならない」理解力がつきます。

ただし、プログラミングをきちんとマスターするには、それなりに集中して学ぶ必要があります。今週の講義/演習をやって、それから 1 週間ほっておいて、翌週忘れたころに続きをやる、というのでは駄目なのです。このため本科目では、資料は予習していただき、時間中に演習したものを出席課題として提出していただき、さらに次回授業前日までに追加の演習をおこなっていただきます。

プログラミング言語としては、前半で Ruby 言語、後半で C 言語を使用します。このようにした理由は、初めてプログラミングを学ぶ人は Ruby のような「簡潔に書ける言語」が望ましく、そしてその後 C 言語を学ぶことで多様な言語に対する展望が得られるためです。

Ruby 処理系に関する情報は `http://www.ruby-lang.org/ja/`にあります。自宅などの Windows 上で動かしたい場合はここの「ダウンロード」ページから Windows 用バイナリを取って来て入れるとよいでしょう (macOS では最初から使えるようになっています)。

1.0.4　ペアプログラミング

本科目では「ペアプログラミング」を採用します。これは次のようなものです。

- 1 つの画面の前に 2 人で座り、一人がキーボードを持ちプログラムを打つ。もう 1 人はそれを一緒に眺めて意見やコメントや考えを述べる。キーボードの担当者は適宜交替してもよい。

このような方法がよい理由としては、次のものがあげられます。

- プログラムを作って動かすのには多くの緻密で細かい注意が必要ですが、1 人でやるより 2 人でやる方がこれらの点が行き届き、無用なトラブルによる時間の空費が避けられます。
- プログラミングではたまたま「簡単な知識」が足りなくて、それを調べて使うまでにすごく時間が掛かることがありますが、2 人いればそのような知識を「どちらかは知っている」可能性が高まり、時間の無駄が省けます。
- プログラミング的な考え方を身に付けるには、さまざまな方面から思考したり、それを身体的な活動と結びつけることが有効です。2 人で互いに議論することで、思考が活発になり、多方面にわたるアイデアが出やすくなるため、上記のことがらに貢献します。

課題提出に際してはもちろん、2 人で作ったものですから、その 2 人については同一のプログラムを出して頂いて構いません。ただし次の条件があります。

- 提出するレポートにおいて、互いに「誰がペアであるか (相手の学籍番号)」を明示する。個人的な好みや都合よりペアを組まずに作業することも認めますので、そのときは「個人作業」と記してください。

- 「当日課題 (A 課題)」と「翌週までの課題 (B 課題)」でペアを変更したり、1 人で作業との間で変更しても構わない。ただし課題の「途中で」は変更しないこと。たとえば B 課題をペアでやる場合は、時間外もプログラミングについてはすべて 2 人で時間を合わせて作業すること。

レポートはあくまでも個人単位で出して頂き、個人単位で採点します (試験も)。ペアで複数のプログラムを作った場合、どれを提出するかは各自で選んで構いません。では、よろしくお願いします。

1.1 プログラムとモデル化

1.1.1 モデル化とコンピュータ

モデル (model) とは、何らかの扱いたい対象があって、その対象全体をそのまま扱うのが難しい場合に、その特定の側面 (扱いたい側面) だけを取り出したものを言います。

たとえば、プラモデルであれば飛行機や自動車などの「大きさ」「重さ」「機能」などは捨てて「形」「色」だけを取り出したもの、と言えます。ファッションモデルであれば、さまざまな人が服を着る、その「様々さ」を捨てて特定の場面で服を見せる、という仕事だと言えます。

コンピュータで計算をするのに、なぜモデルの話をしているのでしょう? それは、コンピュータによる計算自体がある意味で「モデル」だからです。たとえば、「三角形の面積を求める」という計算を考えてみましょう。底辺が 10cm、高さが 8cm であれば

$$\frac{10 \times 8}{2} = 40(cm^2)$$

ですし、底辺が 6cm、高さが 5cm であれば

$$\frac{6 \times 5}{2} = 15(cm^2)$$

です。「電卓」で計算するのなら、実際にこれらを計算するようにキーを叩けばよいわけです。

[1] [0] [×] [8] [÷] [2] [=]

しかし、コンピュータでの計算はこれとは違っています。なぜなら、コンピュータは非常に高速に計算するためのものなので、いちいち人間が「計算ボタン」を押していたら人間の速度でしか計算が進まず意味がないからです。

そこで、「どういうふうに計算をするか」という**手順** (procedure) を予め用意しておき、実際に計算するときは**データ** (data) を与えてそれからその手順を実行させるとあっという間に計算ができる、というふうにします。そしてこの手順が**プログラム** (program) です。

たとえば面積の計算だったら、手順は

[☆] [×] [◇] [÷] [2] [=]

みたいに書いてあり、あとで「☆は 10、◇は 8」というデータを与えて一気に計算します (もちろん、「☆は 6、◇は 5」とすれば別の三角形の計算ができます)。これを捉え直すと、「個々の三角形の面積の計算」から「具体的なデータ」を取り除いた「計算のモデル」が手順だ、ということになります。[1]

コンピュータでの計算はモデル、と言うのには別の意味もあります。三角形は 3 つの直線 (正確に言えば線分) から成りますが、世の中には完璧な直線など存在しませんし、まして鉛筆で紙の上に引いた線は明らかに「幅」を持っていて縁はギザギザ曲がっています。また、10cm とか 8cm とか「きっかり」の長さも世の中には存在しません。でも、そういう細かいことは捨てて「理想的な三角形」に抽象化してその面積を考えて計算しているわけです。

逆に言えば、コンピュータでの計算は常に、現実世界をそのまま扱うのではなく、必要な部分をモデルとして取り出し、それを計算している、ということです。この意味での抽象化やモデル化には、皆様はこれまで数学を通して多く接してきたと思いますが、これからはコンピュータでプログラムを扱う時にもこのようなモデル化を多く扱っていきます。

[1] モデルを作る時の「不要な側面を捨てる」という作業を**抽象化** (abstraction) と言います。つまり、具体的な計算を抽象化したものが手順、という言い方をしてもよいわけです。

1.1.2　アルゴリズムとその記述方法

「三角形の面積の計算方法」のような、計算 (や情報の加工) の手順のことを**アルゴリズム** (algorithm) と言います。ある手順がアルゴリズムであるためには、次の条件を満たす必要があります。

- 有限の記述でできている。
- 手順の各段階に曖昧さがない。
- 手順を実行すると常に停止して求める答えを出す。[2]

1 番目は、「無限に長い」記述は書くこともコンピュータに読み込ませることも不可能だからです。2 番目は、曖昧さがあるとそれをコンピュータで実行させられないからです。3 番目はどうでしょうか。実際にコンピュータのプログラムを書いてみると、手順に問題があって実行が止まらなくなることも頻繁に経験しますが、そのようなものはアルゴリズムとは言えないのです。[3]

アルゴリズムを考えたり検討するためには、それを何らかの方法で記述する必要があります。その記述方法には色々なものがありますが、ここでは手順や枝分かれ等をステップに分けて日本語で記述する、**擬似コード** (pseudocode) と呼ばれる方法を使います。**コード** (code) とは「プログラムの断片」という意味で、「擬似」というのはプログラミング言語ではなく日本語を使うから、ということです。

三角形の面積計算のアルゴリズムを擬似コードで書いてみます。[4]

- triarea: 底辺 w、高さ h の三角形の面積を返す
- 　s ← $\frac{w \times h}{2}$。
- 　面積 s を返す

1.1.3　変数と代入/手続き型計算モデル

上のアルゴリズム中で次のところをもう少しよく考えてみましょう。

- s ← $\frac{w \times h}{2}$。

この「←」は**代入** (assignment) を表します。代入とは、右辺の**式** (expression) [5] で表された値を計算し、その結果を左辺に書かれている**変数** (variable — コンピュータ内部の記憶場所を表すもの) に「格納する」「しまう」ことを言います。つまり、「w と h を掛けて、2 で割って、結果を s のところに書き込む」という「動作」を表していて、数式のような定性的な記述とは別物なのです。

数式であれば $s = \frac{w \times h}{2}$ ならば $h = \frac{2s}{w}$ のように変形できるわけですが、アルゴリズムの場合は式は「この順番で計算する」というだけの意味、代入は「結果をここに書き込む」というだけの意味ですから、そのような変形はできないので注意してください (困ったことに、多くのプログラミング言語では代入を表すのに文字「=」を使うので、普通の数式であるかのような混乱を招きやすいのです)。

モデルという立場からとらえると、式は「コンピュータ内の演算回路による演算」を抽象化したもの、変数は「コンピュータ内部の**主記憶**ないし**メモリ** (memory) 上のデータ格納場所」を抽象化したもの、そして代入は「格納場所へのデータの格納動作」を抽象化したもの、と考えることができます。

このような、式による演算とその結果の変数への代入によって計算が進んでいく計算のモデルを**手続き型計算モデル**と呼び、そのようなモデルに基づくプログラミング言語を**命令型言語** (imperative

[2]実は、計算の理論の中に「答えを出すかどうか分からないが、出したときはその答えが正しい」という手順を扱う部分もありますが、ここでは扱いません。

[3]停止することを条件にしておかないと、アルゴリズムの正しさについて論じることが難しくなります。たとえば、「このプログラムは永遠に計算を続けるかもしれませんが、停止したときは億万長者になる方法を出力してくれます」と言われて、それを実行していつまでも止まらない (ように思える) とき、上の記述が正しいかどうか確かめようがありません。

[4]以下ではこのように、何を受け取って何を行う手順 (アルゴリズム) かを明示するようにします。上の例で「返す」というのは、底辺と高さを渡されて計算を開始し、求まった結果 (面積) を渡されたところに答えとして引き渡す、というふうに考えてください。

[5]プログラミングで言う式とは、計算のしかたを数式に似た形で記述したものを言います。先に説明した、電卓で計算する手順を記したようなものと思ってください。

language) ないし**手続き型言語** (procedural language) と呼びます。手続き型計算モデルは、今日のコンピュータとその動作をそのまま素直に抽象化したものになっています。このため手続き型計算モデルは、最も古くからある計算モデルでもあるのです。

コンピュータによる計算を表すモデルとしては他に、関数とその評価に土台を置く関数型モデルや、論理に土台を置く論理型モデルなどもあるのですが、上記のような理由から、手続き型モデルが今のところもっとも広く使われています。

1.2 アルゴリズムとプログラミング言語

1.2.1 プログラミング言語

プログラムとは、アルゴリズムを実際にコンピュータに与えられる形で表現したものであり、その具体的な「書き表し方」ないし「規則」のことを**プログラミング言語** (programming language) と呼びます。これはちょうど、人間が会話をする時の「話し方」として日本語、英語など多くの言語があるのと同様です。ただし、**自然言語** (natural language — 日本語や英語など、人間どうしが会話したり文章を書くのに使う言語) とは違い、プログラミング言語はあくまでもコンピュータに読み込ませて処理するための人工言語であり、書き方も杓子定規です。

ひとくちにプログラミング言語といっても、実際にはさまざまな特徴を持つ多くのものが使われています。ここでは、プログラムが簡潔に書けて簡単に試してみられるという特徴を持つ、**Ruby 言語** (Ruby language) を用います。

1.2.2 Ruby 言語による記述 exam

では、三角形の面積計算アルゴリズムを Ruby プログラムに直してみましょう。本クラスでは入力と出力は基本的に **irb** コマンド[6] の機能を使わせてもらって楽をするので、計算部分だけを Ruby の**メソッド** (method)[7] として書くことにします。先にアルゴリズムを示した、三角形の面積計算を行うメソッドは次のようになります。

```
def triarea(w, h)
  s = (w * h) / 2.0
  return s
end
```

詳細を説明しましょう。

1. 「`def` メソッド名」～「`end`」の範囲が 1 つのメソッド定義になる。「メソッド名」は自由につけてよい (ここでは Triangle の Area なので `triarea` としている)。
2. メソッド名の後に丸かっこで囲んだ名前の並びがあれば、それらは**パラメタ** (parameter) ないし**引数**[8] の名前となる。メソッドを呼び出す時、各パラメタに対応する値を指定する。
3. メソッド内には**文** (statement — プログラムの中の個々の命令のこと) を任意個並べられる。各々の文は行を分けて書くが、1 行に書く場合は「`;`」で区切る。たとえば上の例のメソッド本体を 1 行に書きたければ「`s = (w * h) / 2.0; return s`」とする。
4. 文は原則として先頭から順に 1 つずつ実行される。
5. **return 文**「`return` 式」を実行するとメソッド実行は終了 (式の値がメソッドの結果となる)。

[6] Ruby の実行系に備わっているコマンドの 1 つで、さまざまな値をキーボードから入力し、それを用いてプログラムを動かす機能を提供してくれます。

[7] メソッドは他の言語での**手続き・サブルーチン** (subroutine) に相当し、一連の処理に名前をつけたもののことです。

[8] メソッドを使用するごとに、毎回異なる値を引き渡して、それに基づいて処理を行わせるための仕組みです。

ここで return についてもう少し説明しておきましょう。メソッドというのはパラメタ (w や h) を指定して呼び出すことができ、呼び出されると上から順番に指示通りの計算をしますが、「return 値」があるとそこで「呼び出されたところに値を持って帰る」動作をします (図 1.1)。ですから、return の後ろにさらに動作を書いてもそれは実行されませんし、2 つ値を返そうと思っても 2 回 return することはできません (1 つの return で複数の値を並べて返すことはできます)。

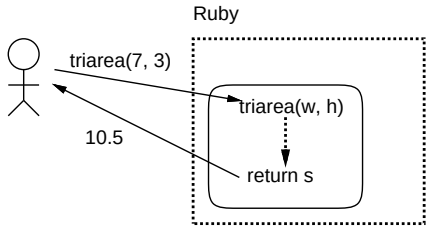


図 1.1: メソッドの呼び出しと return

ところで、上では擬似コードに合わせて、面積の計算結果を変数 s に入れ、次にそれを return していましたが、次のように return の後ろに計算式を直接書いても同じです。

```
def triarea(w, h)
  return (w * h) / 2.0
end
```

このように、たったこれだけのコードでも、大変細かい規則に従って書き方が決まっていることが分かります。要は、プログラミング言語というのはコンピュータに対して実際にアルゴリズムを実行する際のありとあらゆる細かい所まで指示できるように決めた形式なのです。

そのため、プログラムのどこか少しでも変更すると、コンピュータの動作もそれに応じて変わるか、(よくあることですが) そういうふうには変えられないよ、と怒られます。いくら怒られても偉いのは人間であってコンピュータではないので、そういうものだと思って許してやってください。

1.2.3　プログラムを動かす exam

では、このコードを動かしてみましょう。まず、Emacs 等のエディタで上と同じ内容を sample1.rb というファイルに打ち込んで保存してください。この、人間が打ち込んだプログラムを (プログラムを動かす「源」という意味で) ソースないし**ソースコード** (source code) と呼びます。Ruby のソースファイルは最後を「.rb」にするのが通例です。そして、先に進む前に ls を使って、作成したファイルがあることを確認してください。ディレクトリが違っている場合はソースファイルのあるディレクトリに移動しておくこと。

上記が済んだらいよいよ **irb** コマンドを実行して Ruby 実行系を起動してください (「%」はコマンドプロンプトのつもりなので打ち込まないでください)。

```
% irb
irb(main):001:0>
```

この「irb なんとか>」というのは irb の**プロンプト** (prompt — 入力をどうぞ、という意味の表示) で、ここの状態で Ruby のコードを打ち込めます。

プロンプトの読み方を説明すると、main というのは現在打ち込んでいる状態がメインプログラム (最初に実行される部分) に相当することを意味しています。次の数字は何行目の入力かを表しています。最後の数字はプログラムの**入れ子** (nesting — 「はじめ」と「おわり」で囲む構造の部分) の中に入るごとに 1 ずつ増え、出ると 1 ずつ減ります。とりあえずあまり気にしなくてよいでしょう。以後の実行例では見た目がごちゃごちゃしないように「irb>」だけを示すことにします。

次に load(ファイルからプログラムを読み込んでくる、という意味です) で sample1.rb を読み込ませます。ファイル名は**文字列** (string) として渡すので'' または""で囲んでください。[9]

```
irb> load 'sample1.rb'
=> true
irb>
```

true が表示されたら読み込みは成功で、ファイルに書かれているメソッド triarea が使える状態になります。成功しなかった場合は、ファイルの置き場所やファイル名の間違い、ファイル内容の打ち間違いが原因と思われるので、よく調べて再度 load をやり直してください。

なぜわざわざ 3〜4 行程度の内容を別のファイルに入れて面倒なことをしているのでしょうか? それは、メソッド定義の中に間違いがあった時、定義を毎回 irb に向かって打ち直すのでは大変すぎるからです。このため、以下でもメソッド定義はファイルに入れて必要に応じて直し、irb では load とメソッドを呼び出して実行させるところだけを行う、という分担にします。

load が成功したら triarea が使えるはずなので、それを実行します。

```
irb> triarea 8, 5
=> 20.0
irb> triarea 7, 3
=> 10.5
irb>
```

確かに実行できているようです。irb は quit で終わらせられます。

```
irb> quit
%
```

苦労の割に大した結果ではない感じですが、初心者の第 1 歩として、着実に進んでいきましょう。

演習 1 例題の三角形の面積計算メソッドをそのまま打ち込み、irb で実行させてみよ。数字でないものを与えたりするとどうなるかも試せ。

演習 2 三角形の面積計算で、割る数の指定を「2.0」でなくただの「2」にした場合に何か違いがあるか試せ。

演習 3 次のような計算をするメソッドを作って動かせ。[10]

a. 2 つの実数を与え、その和を返す (ついでに、差、商、積も)。何か気づいたことがあれば述べよ。
b. 「%」という演算子は**剰余** (remainder) を求める演算である。上と同様に剰余もやってみよ。何か気づいたことがあれば述べよ。
c. 数値 x を与え、逆数 $\frac{1}{x}$ を出力する (分子は「1.0」という書き方にした方がいいかもしれない)。
d. 数値 x を与え、その 8 乗を返す。ついでに 6 乗、7 乗もやるとなおよい。なお、Ruby のべき乗演算「**」は使わず、なおかつ乗除算が少ないことが望ましい。
e. 円錐の底面の半径と高さを与え、体積を返す。
f. 実数 x を与え、x の平方根を出力する。さまざまな値について計算し、精度がどれくらいあるか検討せよ。[11]
g. その他、自分が面白いと思う計算を行うメソッドを作って動かせ。

[9] 本来ならメソッドに渡すパラメタは丸かっこで囲むのですが、Ruby では曖昧さが生じない範囲でパラメタを囲む丸かっこを省略できます。本資料ではプログラム例の丸かっこは省略しませんが、irb コマンドに打ち込む時は見た目がすっきりするので丸かっこを適宜省略します。

[10] 1 つのファイルにメソッド定義 (def ... end) はいくつ入れても構わないので、ファイルが長くなりすぎない範囲でまとめて入れておいた方が扱いやすいと思います。

[11] x の**平方根** (square root) は Math.sqrt(x) で計算できます。

1.3　コンピュータ上での実数の扱い

1.3.1　printf による表示の制御 exam

コンピュータ上での正負の整数が 2 進法を用いて表現されていることはすでに学んできましたが、それでは小数点付きの数値はどうでしょうか。数学の世界では整数は実数の特別な場合ですが、コンピュータ上の数の表現の場合は**整数型** (integral type) と**実数型** (real type) はまったく違った性質を持っていて、プログラムの上でもきっぱり区別されます。なお、型ないし**データ型** (data type) とはデータの種類を意味する用語です。

たとえば、先の三角形の面積のプログラムで割る数を「2.0」としたのと「2」としたのでは挙動が違うのに気づいた人もいるかと思います。「10 を 3 で割る」例で確認してみましょう。実は irb では、いちいちプログラムを書かなくても式を直接計算できます。

```
irb> 1 / 3                          ←両方とも整数だと
=> 0                                ←整数の (切捨ての) 割り算
irb> 1.0 / 3                        ←片方が実数なら
=> 0.3333333333333333               ←小数点つきの結果
```

つまり「1」や「3」は整数を表し、「1.0」のように小数点がついていると実数、という区別があり、さらに整数の割り算と実数の割り算は現然と違っているのです。

ところで、小数点つきの方は本当は無限に「3」が続くはずですが、途中で打ち切られていますね。この「表示の桁数」を自分で指定するには次のようにします。

```
irb> printf("value = %.20g\n", 1.0 / 3) ←小数点以下 20 桁指定
value = 0.33333333333333331483          ← printf の出力
=> nil                              ←結果は「なし」
```

printf というのは出力の命令で、ただしそのときに「桁数や出力の幅などを指定」できるので、それを使って 20 桁表示させています。もう少し詳しく説明しましょう。

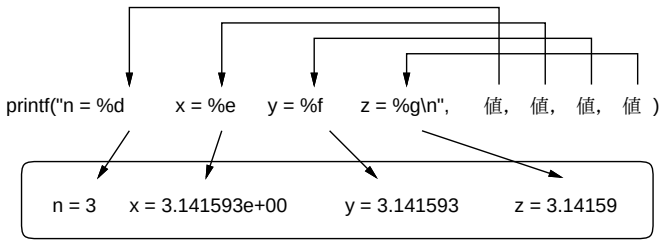


図 1.2: printf の機能と書式文字列

printf の書き方は「printf("書式文字列", 値, …)」のような形であり、書式文字列の中に「%○」という形の出力指定が複数埋め込めます (改行はされないので、改行したければ改行文字「\n」を含める必要があります)。そして、基本的には書式文字列が表示されるのですが、それぞれの出力指定は、後に出て来る「値」1 つずつと順番に対応し、その指定に応じた形に整形されて埋め込まれます (図 1.2)。主要な出力指定を表 1.1 に示します。

注意! コンピュータでは「小さい字」が使えないので、伝統的に指数部分を「e ±指数」で表します (e は exponent の e)。たとえば「3.0×10^{22}」であれば「3.0e+22」です。このような表示は「エラー」とかではないのでそのつもりで。

そしてさらに、「%」と指定文字の間に追加の指定を入れられます。具体的には、「%w ○」により出力する幅 w が指定でき、「%.d ○」(実数のみ) により小数点以下の桁数 d が指定できます。ということで、先の例では「小数点以下 20 桁をおまかせで出力」していたわけです。

表 1.1: printf の主要な書式指定

%d	%e	%f	%g	%s
整数として出力	実数を指数形式で出力	実数を小数点表現で出力	実数をおまかせで出力	文字列を出力

「0.333…」のところに戻りますが, こうしてみると最後が「1483」となっています。つまりコンピュータでは有限の桁数で計算するため、精度に限界があります。そして「おまかせ」で表示させているときはその限界より長くは (どうせ誤差なので) 表示しないわけです。

1.3.2 実数の表現と浮動小数点 exam

では具体的には、有限のビット数で実数を表すのにはどうしたらよいでしょうか? たとえば、十進表現で 8 桁ぶんの整数を表す方法があるのなら、そのうちの下から 4 桁が小数点以下、その上が小数点以上、のように考えればそれで小数点付きの数が表せる、という考えもあります。

□□□□. □□□□

このような考え方を、小数点が決まった位置に固定されていることから**固定小数点** (fixed point) による実数表現と呼びます。しかし実際には、この方法はあまりうまくいきません。というのは、科学技術計算では頻繁に「30,000,000」だとか「0.0000001」のような数値が出てくるので、この方法ではすぐに限界になってしまうからです。

ではどうしましょう? たとえば理科では、上のような数値の表現ではなく、「3×10^{8}」とか「1×10^{-6}」のような記法が使われますね。つまり、1 つの数値を**指数** (exponent — 桁取り) と**仮数** (mantissa — 有効数字) に分けて扱うことで、広い範囲の数値を柔軟に扱うことができます。この方法は、指数によって小数点の位置を動かすものと考えて**浮動小数点** (floating point) と呼ばれます。

たとえば、同じ十進表現で 8 桁ぶんでも、6 桁の有効数字と 2 桁の指数に分けた浮動小数点表現を扱うとすれば、表せる絶対値のもっとも大きい数は「$\pm\ 9.99999\times10^{99}$」、0 でない絶対値のもっとも小さい数は「$0.00001\times10^{-99}$」ということになり、ずっと広い範囲の数が扱えることになるわけです。

コンピュータでは 2 進法を使うため、上と同様のことを 2 進表現で行います。多くの環境では、符号 1 ビット、仮数部 52 ビット、指数部 (符号含む)11 ビット、合計 64 ビットの浮動小数点表現が使われています。(このビットの割り当ては、**IEEE754** と呼ばれる標準に従ったものです。)

1.3.3 浮動小数点計算の誤差 exam

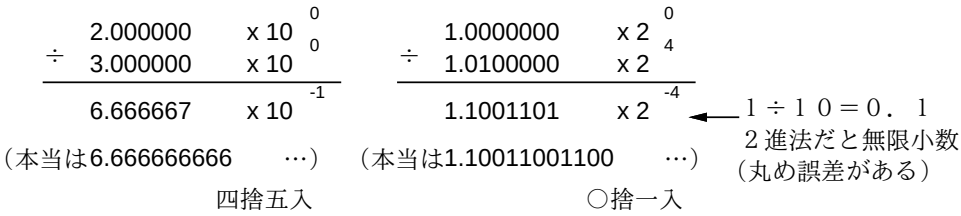


図 1.3: 丸め誤差

浮動小数点を用いた実数表現には、整数の表現とは異なる注意点があります。まず、有効数字は当然ながら有限なので、その範囲で表せない結果の細かい部分は**丸め** (rouding — 十進表現で言えば四捨五入) が行われて、**丸め誤差** (roundoff error) となります。言い替えれば、コンピュータによる実数計算は基本的に近似値による計算を行っているものと考えるべきなのです。

また、絶対値が大きく異る 2 つの数を足したり引いたりすると、絶対値が小さいほうの数値の下の桁は (演算のための桁揃えの結果) 捨てられてしまうので、これも誤差の原因となります。これを**情**

報落ち (loss of information) と言います。極端な例として、演算した結果が元の (絶対値が大きいほうの) 数のまま、ということも起こります。これは、たとえば図 1.4 左のような例を思い浮かべてみれば分かると思います。

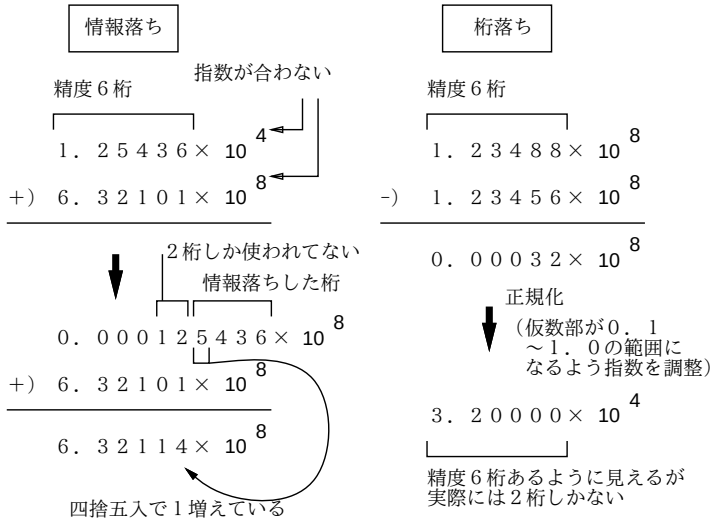


図 1.4: 情報落ちと桁落ち

逆に、非常に値が近い数値どうしを引き算する場合も、上のほうの桁がすべて 0 になるため、結果は元の数の下の部分だけから得られたものとなり、やはり誤差が大きくなります。これを**桁落ち** (cancellation) と言います。素朴に計算すると桁落ちが問題になる例として、次のものを考えてみます。

$$\sqrt{x+1}-1$$

x が 0 に近いとき、$\sqrt{x+1}$ も 1 に近いので桁落ちが起きます。これを避けるためには、次のように変形します。

$$\sqrt{x+1}-1=\frac{\sqrt{x+1}-1}{1}=\frac{(\sqrt{x+1}-1)(\sqrt{x+1}+1)}{\sqrt{x+1}+1}=\frac{x}{\sqrt{x+1}+1}$$

難しくしただけみたいに見えるかも知れませんが、最後の式であれば引き算をしないので桁落ちから逃れられるわけです。ところで、こう変形してみると、$x\sim 0$ のときにはこの式はおよそ $\frac{x}{2}$ だと分かりますね。実際に両方のやり方で計算してみて確認しましょう。

```
def calc1(x)
  return Math.sqrt(x + 1.0) - 1.0
end
def calc2(x)
  return x / (Math.sqrt(x + 1.0) + 1.0)
end
```

最初の素朴版から見てみます。

```
irb> calc1 0.00000000001
=> 5.000000413701855e-12
irb> calc1 0.000000000001
=> 5.000444502911705e-13
irb> calc1 0.0000000000001
=> 4.9960036108132044e-14
```

```
irb> calc1 0.00000000000001
=> 4.884981308350689e-15
irb> calc1 0.000000000000001
=> 4.440892098500626e-16
```

x が小さくなると、どんどん $\frac{x}{2}$ から外れて行きます。では修正版ではどうでしょうか。

```
irb> calc2 0.00000000001
=> 4.9999999999875e-12
irb> calc2 0.000000000001
=> 4.99999999999875e-13
irb> calc2 0.0000000000001
=> 4.999999999999876e-14
irb> calc2 0.00000000000001
=> 4.999999999999988e-15
irb> calc2 0.000000000000001
=> 4.999999999999999e-16
```

確かにこちらは大丈夫です。

最後にあと 1 つだけ、浮動小数点表現に関する注意があります。整数では全てのビットのパターンを数値の表現として使っていましたが、浮動小数点では指数部と仮数部の組み合わせ方に制約があるので (たとえば仮数部が 0 であれば値が 0 なので指数部には意味がなく、この時は指数部も 0 にしておくのが普通)、これを利用して正負の**無限大** (infinity — $\pm\infty$) や**非数** (NaN — Not a Number) などの特別な値を用意しています。また、0 にも「+0」と「−0」があったりします。だから、演算の結果としてこれらのヘンな値が表示されても驚かないようにしてください。

演習 4 2 次方程式の解の公式「$\frac{-b\pm\sqrt{b^2-4ac}}{2a}$」について、次のことをやりなさい。

a. 係数 a, b, c の値を引数として渡すと、2 つの解を打ち出す (または「[値, 値]」の形で返す) メソッドを作成しなさい。いくつかの値で実行例を示すこと。

b. 上記に加えて、$|b|$ と $|\sqrt{b^2-4ac}|$ が非常に近い場合を解かせてみて、桁落ちによる誤差が現れることを観察しなさい。いくつかの実行例を示すこと。

   ヒント: $(x+d)(x+1)$ で d が非常に 0 に近い値 (たとえば 0.000000000012345 とか) はそうなるでしょう。この式を展開して a, b, c を決めればいいわけです。

c. 仮に $b \geq 0$ とする (負なら a, b, c すべてに -1 を掛ければ解は同じで $b \geq 0$ とできる)。すると、$\pm$ のうち $-$ については両方の符号が同じなので桁落ちなしに解が求まる。これを α とおき、解と係数の関係 $\alpha\beta = \frac{c}{a}$ を利用して他方の解 β を求めることができる。この方法で 2 つの解を求めて打ち出す (または「[値, 値]」の形で返す) メソッドを作成しなさい。いくつかの値で実行例を示すこと。

d. 上記の設問 b で桁落ち誤差のあった実行例が c のプログラムでは問題なく計算できることを観察しなさい。いくつかの実行例を示すこと。

演習 5 実数の演算で誤差が現れるような、次のような計算をプログラムにおこなわせて、確かに誤差が現れることを確認しなさい。いずれも複数の実行例を示すこと。必要なら `printf` で表示桁数を増やしてみること。次のようなプログラムの形が想定されます。

```
def kadai5a
  printf("%.20g\n", 1.12345 - 1.0);
  (同様の行がいくつも…)
end
```

a. (桁落ち): 「1.12345 - 1.0」は「0.12345」になるでしょうか。「1.1234512345 - 1.12345」はどうでしょうか。「12345」をさらに増やすとどうなるでしょうか。
b. (情報落ち);「1.0 + 0.0012345」は「1.0012345」になるでしょうか。「1.0 + 0.00000000012345」はどうでしょうか。「0」をさらに増やすとどうなるでしょうか。
c. (丸め誤差): 「ある数に 0.1 を掛ける」場合と「ある数を 10.0 で割る (10 ではなく 10.0 にすること!)」場合では結果が違うような数があります。ところが、「0.125 を掛ける」のと「8.0 で割る (8 ではなく 8.0 にすること!)」の場合は違いはないかも知れません。どんな数がそうなるかとかその理由とかを探究してみてください。
d. その他、コンピュータの実数計算が数学と違っている具体例を示すようなプログラムを好きなように探究してみてください。

本日の課題 1A

「演習 3」で動かしたプログラム (どれか 1 つでよい) を含むレポートを提出しなさい。プログラムと、簡単な説明が含まれること。アンケートの回答もおこなうこと。

Q1. プログラム、って恐そうですか? 第 2 外国語と比べてどう?

Q2. Ruby 言語のプログラムを打ち込んで実行してみて、どのような感想を持ちましたか?

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 1B

「演習 3」(ただし 1A で提出したものは除外、以後も同様)「演習 4」「演習 5」の小課題全体から選択して 1 つ以上プログラムを作り、レポートを提出しなさい。プログラムと、課題に対する報告・考察 (やってみた結果・そこから分かったことの記述) が含まれること。アンケートの回答もおこなうこと。

Q1. プログラムを作るという課題はどれくらい大変でしたか?

Q2. コンピュータでの数値の計算に対する数学とは違う挙動についてどう思いましたか?

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

＃2　分岐と反復+数値積分

前回はプログラミング入門なので一直線のコードで計算するだけでしたが、今回はいよいよ制御構造 (分岐や反復) を使っていただきます。今回の目標は次の通り。

- 基本的な制御構造 (分岐・反復) を理解し、これらを使ったプログラムが書けるようになる。
- 基本的な制御構造を用いたアルゴリズムやプログラムについて考えられるようになる。

以後毎回、前回の演習問題から抜粋して解説します。自分で課題をやってから読むことを勧めます。

2.1　前回演習問題の解説

2.1.1　演習 3a — 四則演算を試す

演習 3a は和の計算でした。メソッド内の計算式を取り替えればいいだけなので簡単です。

```
def add(x, y)
  return x + y
end

irb> add 3.5, 6.8
=> 10.3
```

和、差、商、積の場合も同様でいいのですが、4 つメソッドを作る代わりに 1 つで済ませる方法を考えてみます (半分くらいは新しい内容の紹介を兼ねています)。まず先に説明したように、メソッドの最後に値を返す代わりに、puts などで順次画面に書き出す方法があります。

```
def shisoku0(x, y)
  puts(x+y)
  puts(x-y)
  puts(x*y)
  puts(x/y)
end

irb> shisoku0 3.3, 4.7
8.0
-1.4
15.51
0.702127659574468
=> nil
```

4 つの値が打ち出され、shisoku0 の結果としては **nil**(何もないことを示す値) が返されています。

上の方法だと「1 つの結果が返る」のでないのがちょっと、という気がするかもしれません。そこで次に、1 つの文字列を返し、その中に 4 つの数値が埋め込まれている、というふうにしてみましょう。

Ruby では**文字列** (string — 文字が並んだデータ) は「'...'」または「"..."」のように**シングルクォート**または**ダブルクォート**で囲んで表しますが、ダブルクォートのほうは内部に色々なものを埋

め込む機能がついています。[1] 具体的には、文字列の中に「#{...}」という形のものがあると、中カッコ内の式を**評価** (evaluation — 値を計算すること) して、結果をそこに埋め込んでくれます。

これを利用した「四則演算」のメソッドを示します。

```
def shisoku1(x, y)
  return "#{x+y} #{x-y} #{x*y} #{x/y}"
end

irb> shisoku1 3.3, 4.7
=> "8.0 -1.4 15.51 0.702127659574468"
```

確かに、「文字列」が打ち出されていて、その中に 4 つの数値が埋め込まれています。

もう 1 つ、Ruby など多くの言語では値の並んだものを**配列** (array) という機能で扱います。Ruby では [...] の中に値をカンマで区切って並べることで配列を直接書けるので、これを使って 4 つの数値をまとめて返すことができます。[2]

```
def shisoku2(x, y)
  return [x+y, x-y, x*y, x/y]
end

irb> shisoku2 3.3, 4.7
=> [8.0, -1.4, 15.51, 0.702127659574468]
```

ここでは文字列の場合とあまり変わらない感じがするかもしれませんが、配列では返された値の中から「0 番目」「1 番目」など番号を指定して特定の要素を取り出せるので、より便利に使えます。

2.1.2　演習 3b — 剰余演算

演習 3b は剰余演算「%」を試すというものでした。演算子を取り換えるだけなので、プログラムは簡単ですね。

```
def jouyo(x, y)
  return x % y
end
```

実行してみましょう。

```
irb> jouyo 8, 5
=> 3
irb> jouyo 20, 5
=> 0
irb> jouyo -8, 5
=> 2
irb> jouyo -21, 5
=> 4
```

マイナスの時も試しましたか? 「ここでマイナスだとどうだろう」と気付くようになってください。で、分母がマイナスだとどうでしょう?

---

[1] ダブルクォートは埋め込み機能等のために**特殊文字** (special character — 英数字以外の文字) を様々に解釈します。そのようなことをせずに文字列をそのまま表示させたい場合はシングルクォートを使ってください。

[2] return の後だと囲んでいる [ ] を省略できますが、場所によっては省略できないので常に書くことを薦めます。

演習 3c — 逆数

演習 3c は逆数ですが、これは簡単ですね。分子を「1」と整数にした場合は、パラメタを整数で与えると「整数割る整数の切捨て割算」になってしまうことに注意が必要です。

```
def inverse(x)
  return 1.0 / x
end
```

演習 3d — 8 乗、6 乗、7 乗

演習 3d は 8 乗、6 乗、7 乗です。Ruby のべき乗演算子「**」を使えば次のように簡単です。

```
def x8a(x)
  return x**8
end
```

しかしこの演算を使わないとするとどうでしょうか。

```
def x8b(x)
  return x*x*x*x*x*x*x*x
end
```

もちろんこれでもいいのですが、乗算の数を減らす方法があります (「;」は 1 行に複数の文を書くときに区切りとして入れる必要があります)。

```
def x8c(x)
  x2 = x*x; x4 = x2*x2; return x4*x4
end
```

6 乗は「return x4*x2」、7 乗は「return x4*x2*x」ですね。「return x4*x4 / x」はどうでしょう? 除算は乗算より遅いので、無理のない範囲で少なくしておいた方がよいです。

演習 3e — 円錐の体積

演習 3e は円錐の体積でした。底面の半径 r、高さ h として、まず円錐の底面の面積は πr^2。体積はこれに高さを掛けて 3 で割ればできます。

```
def cornvol(r, h)
  return (r**2*3.1416*h) / 3.0
end
```

ちなみに「**」はべき乗の演算子です。もちろん 2 乗は「r*r」と書いても構いません。

```
irb> cornvol 3.0, 4.0
=> 37.6992
```

ところで「円周率が 3.1416 というのは不正確だ」と思う人もいそうですね。しかし、コンピュータ上の計算は「電卓での計算」と同様、有限の桁数でしか行えないのであり、自分で必要と思う適当な桁数を決めてその範囲でやるしかないので、有効数字 5 桁くらいでと思うならこれでよいわけです。[3]

[3] 3.141592653589793 くらいまでは扱える精度があるので、この定数をいちいち書くのは嫌だという人のために Math::PI と書いてもよいようになっています。同様に、自然対数の底 e は Math::E で表せます。

2.1.3　演習 3f — 平方根

平方根は Math.sqrt(x) で計算できるので、要するに何桁くらい精度があるか調べるわけです。

```
def sqrts
  printf("%.20g\n", Math.sqrt(2));
  printf("%.20g\n", Math.sqrt(3));
  printf("%.20g\n", Math.sqrt(5));
end
```

実行してみます。

```
irb> sqrts
1.4142135623730951455
1.7320508075688771932
2.2360679774997898051
=> nil
```

正確な平方根の値を掲げておきます (見比べると、精度としては 16〜17 桁程度であると言えます)。

$$\sqrt{2} \approx 1.41421356237309504880168872420969807856 96$$
$$\sqrt{3} \approx 1.73205080756887729352744634150587236694 28$$
$$\sqrt{5} \approx 2.23606797749978969640917366873127623544 06$$

2.1.4　演習 4 — 2 次方程式の解の公式

まず素直に計算式通りコードを書きます。$\sqrt{D}$ を変数 rd に保存し、それを使って 2 つの解を計算しました。最後にその 2 つを配列として返すようにしました。

```
def solve1(a, b, c)
  rd = Math.sqrt(b**2 - 4*a*c)
  x1 = (-b - rd) / (2.0 * a)
  x2 = (-b + rd) / (2.0 * a)
  return [x1, x2]
end
```

では実行させてみます。

```
irb> solve1 1, -2, 1
=> [1.0, 1.0]
irb> solve1 1, 5, 6
=> [-3.0, -2.0]
```

とくに問題なさげです。

では設問 b に進んで、b と $\sqrt{D}$ が非常に近いものをやってみます。ヒントに書いたように、$(x+d)(x+1)$ で $d = 0.000000000012345$ をやってみましょう。

```
irb> solve1 1, 1.000000000012345, 0.000000000012345
=> [-1.0, -1.2345013900016966e-11]
```

「e-11」は「$\times 10^{-11}$」の意味でした。誤差はありますが、まあ計算できてます。ゼロを増やすと?

```
irb>  solve1 1, 1.000000000000012345, 0.000000000000012345
=> [-1.0, -1.2378986724570495e-14]
```

x2 の有効数字が 2 桁くらいに減ってしまった感じです。問題を克服するため、x2 の計算を解の公式ではなく c と x1 から求めるようにします (x1 については、b の符号が正なので $-b-rd$ は内容が足し算であり桁落ちは起きません)。

```
def solve2(a, b, c)
  rd = Math.sqrt(b**2 - 4*a*c)
  x1 = (-b - rd) / (2.0 * a)
  x2 = c / (a * x1)
  return [x1, x2]
end
```

先の値について実行してみると、完全にぴったりになると分かります。

```
irb> > solve2 1, 1.000000000000012345, 0.000000000000012345
=> [-1.0, -1.2345e-14]
```

2.1.5 演習 5 — 実数計算の誤差

実数計算の誤差を観察する問題なので、printf を用いて十分な桁数を表示します。まず a. から。

```
def kadai5a
  printf("%.20g\n", 1.12345 - 1.0)
  printf("%.20g\n", 1.1234512345 - 1.12345)
  printf("%.20g\n", 1.123451234512345 - 1.1234512345)
  printf("%.20g\n", 1.12345123451234512345 - 1.123451234512345)
end
```

```
irb> kadai5a
0.12345000000000005969      ←まあまあ
1.2345000000024697329e-06   ←誤差が
1.2345013900016965636e-11   ←増えて来て
0                           ←最後は近すぎるので 0
=> nil
```

値が近くなるにつれて桁落ちが現れてくることが分かります。次は b. です。

```
def kadai5b
  printf("%.20g\n", 1.0 + 0.0012345);
  printf("%.20g\n", 1.0 + 0.000000012345);
  printf("%.20g\n", 1.0 + 0.00000000000012345);
  printf("%.20g\n", 1.0 + 0.0000000000000000012345);
end
```

```
irb> kadai5b
1.0012345123449999384  ← OK
1.000000012345123368   ←誤差が
1.0000000000001234568  ←増えて来て
1                      ←小さすぎると 1+ε =1 となる
=> nil
```

足す値が小さくなるほど下の桁は失われていき、最後はまったく値が増えなくなります。設問 c. はいろいろ試したいので irb で直接計算してみましょう。

```
irb> printf "%.20g\n", 1.0/3.0
0.33333333333333331483  ←有効桁数は 10 進で 16 桁程度
=> nil
irb> printf "%.20g\n", 1.0/3.0 * 3.0
1           ← 3 倍すると最後の桁が丸められて元に戻る
=> nil
irb> printf "%.20g\n", 7.0 / 10.0
0.69999999999999995559 ← 0.7 も 2 進で切りよくない
=> nil
irb> printf "%.20g\n", 7.0 * 0.1
0.70000000000000006661 ←値 0.1 も誤差を含むので
=> nil
```

このように、有限桁数の計算なので微妙に誤差が現れます。しかし一方で、2 進表現を使っているということは、2^N とか $\frac{1}{2^N}$ とかは非常に「切りのよい数」となり、あふれない限りは誤差が出ません。

```
irb> printf "%.40g\n", 7.0 / 16.0
0.4375
=> nil
irb> printf "%.40g\n", 7.0 * 0.0625
0.4375
=> nil
irb> printf "%.40g\n", 7.0 / (2**16)
0.0001068115234375
=> nil
irb> printf "%.40g\n", 7.0 * (0.5**16)
0.0001068115234375
=> nil
irb> printf "%.40g\n", 7.0 / (2**32)
1.629814505577087402343750e-09
=> nil
irb> printf "%.40g\n", 7.0 * (0.5**32)
1.629814505577087402343750e-09
=> nil
```

2.2　基本的な制御構造

2.2.1　実行の流れと制御構造

ここまでに出てきたアルゴリズムおよびプログラムはすべて「1 本道」、つまり上から順番に実行して一番下まで来たらおしまい、というものでした。単純な計算ならそれでも問題ありませんが、手順が複雑になってくると、実行の流れをさまざまに切り換えていくことが必要になります。この、実行の流れを切り換える仕組みのことを、一般に**制御構造** (control structure) と呼びます。

制御構造の表現方法の 1 つに**流れ図** (flowchart) があります。流れ図では、図 2.1 にあるような「処理を示す箱」「条件による枝分かれを示す箱」などを矢線でつなげて実行の流れを表現します。流れ図は一見分かりやすそうですが、作成に手間が掛かる、場所をとる、ごちゃごちゃの構造を作ってしまいがち、という弱点のため、今日のソフトウェア開発ではあまり使われません (このため本資料でも、流れ図の代わりに擬似コードを主に用います)。

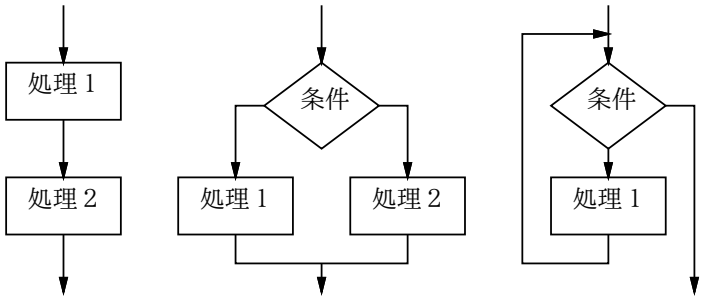


図 2.1: 3 つの基本的な制御構造

アルゴリズムを記述する時にはさまざまな実行の流れを組み立てますが、今日ではそれらの実行の流れは、図 2.1 に示す 3 つの制御構造を組み合せる形で作り出していくのが普通です。

- **順次実行**ないし連接— 動作を順番に実行していくこと。
- **枝分かれ**ないし分岐 — 条件に応じて 2 群の動作のうちから一方を選んで実行すること。
- **繰り返し**ないし反復 — 条件が成り立つ限り一群の動作を繰り返し実行すること。[4]

なぜこの 3 つが基かというと、「どんなにごちゃごちゃの流れ図でも、それと同等の動作を、この 3 つの組み合わせによって作り出せる」という定理があり、そのためにこの 3 つさえあればどのような処理の流れでも表現可能だからです。連接については単に動作を並べて書いたものは並べた順番に実行される、というだけなので、以下では残りの 2 つの制御構造をコード上で表現するやり方と、それらを組み合わせてアルゴリズムを組み立てていくやり方を学びます。

2.2.2 枝分かれと if 文 exam

上述のように、枝分かれとは、条件に応じて 2 群の動作のうちから一方を選んで実行するものです。擬似コードでは枝分かれを次のように書き表すものとします (「動作 2」が不要なら「そうでなければ」も書かなくてもかまいません)。

- もし ～ ならば、
- 　　動作 1。
- そうでなければ、
- 　　動作 2。
- 枝分かれ終わり。

Ruby ではこれを **if 文** (if statement) を使って表します (右側は「動作 2」のない場合です)。

```
if 条件 then
  ... 動作 1 ...
else
  ... 動作 2 ...
end
```

```
if 条件 then
   ... 動作 1 ...
end
```

`then` は Ruby では省略できますが、ただし「動作 1」を条件と同じ行に書く場合には省略できません。

「条件」については、当面は次の形のものがあると思っておいてください。

- 比較演算 — 「`x > 10`」等、2 値を比べるもの。**比較演算子**としては次の 6 種類がある。[5]

[4] 実行の流れを図示すると環状になるので、**ループ** (loop) とも呼びます。

[5] Ruby では「`!`」は「否定」を表すのに使っています。階乗の記号ではないので注意してください。

>	>=	<	<=	==	!=
より大	以上	より小	以下	等しい	等しくない

- 条件の組み合わせとして次が使える。[6] これらを「()」でくくったり複数組み合わせられる。

かつ (両方が成立)	または (最低限一方が成立)	否定 (〜でない)
条件1 && 条件2	条件1 || 条件2	! 条件1

では例として、「入力 x の絶対値を計算する」ことを考えます。擬似コードを示しましょう。

- abs1: 数値 x の絶対値を返す
- 　もし $x < 0$ ならば、
- 　　$result \leftarrow -x$。
- 　そうでなければ、
- 　　$result \leftarrow x$。
- 　枝分かれ終わり。
- 　$result$ を返す。

考え方としては簡単ですね? これを Ruby にしてみましょう。

```
def abs1(x)
  if x < 0
    result = -x
  else
    result = x
  end
  return result
end
```

実行の様子も示しておきます (0 もテストしていることに注意。作成したコードをテストするときには系統的に洩れなく試してみることが大切です)。

```
irb> abs1 8        ←正の数の絶対値は
=> 8               ←元のまま
irb> abs1 -3       ←負の数であれば
=> 3               ←正の数になる
irb> abs1 0        ← 0 の場合も
=> 0               ←元のまま
irb>
```

ところで、同じ絶対値のプログラムを次のように書いたらどうでしょうか?

- abs2: 数値 x の絶対値を返す
- 　もし $x < 0$ ならば、
- 　　$-x$ を返す。
- 　そうでなければ、
- 　　x を返す。
- 　枝分かれ終わり。

[6] Ruby ではさらに演算子として and、or、not も使えますが、結合の強さが記号版と違っていて混乱しやすいので、本資料では使っていません。

Ruby 版は次のようになります。

```
def abs2(x)
  if x < 0
    return -x
  else
    return x
  end
end
```

先のとどちらが好みでしょうか? また、別のバージョンとして次のものはどうでしょうか?

- abs3: 数値 x の絶対値を返す
- $result \leftarrow x$。
- もし $x < 0$ ならば、
- $result \leftarrow -x$。
- 枝分かれ終わり。
- $result$ を返す。

Ruby プログラムも示します (if を 1 行に書いてみました。このような時は then が必須)。

```
def abs3(x)
  result = x
  if x < 0 then result = -x end
  return result
end
```

3 つのプログラムについて、あなたはどれが好みだったでしょうか?

一般に、プログラムの書き方は「どれが絶対正解」ということはなく、場面ごとに何がよいかが違ってきますし、人によっても基準が違うところがあります。ですから、皆様がこれからプログラミングを学習するに当たっては、自分なりの「よいと思う書き方」を発見していく、という側面が大いにあります。そのことを心に留めておいてください。

演習 1 絶対値計算プログラムの好きなバージョンを打ち込んで動かせ。

演習 2 枝分かれを用いて、次の動作をする Ruby プログラムを作成せよ。

a. 2 つの異なる実数 a、b を受け取り、より大きいほうを返す。
b. 3 つの異なる実数 a、b、c を受け取り、最大のものを返す。(やる気があったら 4 つでやってみてもよいでしょう。)
c. 実数を 1 つ受け取り、それが正なら「'positive'」、負なら「'negative'」、零なら「'zero'」という文字列を返す。(注意! 文字列は'…' または"…"で囲んで指定します。)

2.2.3 繰り返しと while 文 exam

ここまでは、プログラム上に書かれた命令はせいぜい 1 回実行されるだけでしたから、プログラムが行う計算の量はプログラムの長さ程度しかありませんでした。しかし、繰り返しがあれば、その範囲内の命令は何回も反復して実行されますから、短いプログラムでも大量の計算を行わせられます。

まず、繰り返しの最も一般的な形である、条件を指定した繰り返しの擬似コードは次のように書き表すものとします。[7]

[7]「〜」のところには条件を記述しますが、ここに書けるものは if 文の条件とまったく同じです。

- ～ である間繰り返し、
- 　　動作 1。
- 繰り返し終わり。

この形の繰り返しは、Ruby では while 文 (while statement) として記述します。

```
while 条件 do
  ... 動作 1 ...
end
```

条件の次にある do も、Ruby では省略することができます。ただし、「動作 1」を条件と同じ行に書く場合は省略できません。本資料では do は省略しないことにします。

多くのプログラミング言語では、このような条件を指定した繰り返しは while というキーワードを用いて表すので、**while ループ**と呼びます。while ループは形だけなら if 文より簡単ですが、慣れるまではどのように実行されるかイメージが湧かない人が多いと思います。while ループの実行のされ方は、次のようなものだと考えてください。

- 「～」を調べる (成立)。
- 動作 1 を実行。
- 「～」を調べる (成立)。
- 動作 1 を実行。
- 「～」を調べる (成立)。
- 動作 1 を実行。
- …
- 「～」を調べる (不成立)。
- 繰り返しを終わる。

つまり、条件を調べ、成り立てば動作 1 を実行し、また条件を調べ、…のように繰り返していき、条件が成り立たなくなると繰り返しを終わります。

while を使った簡単な例として、カウントダウンをしてみましょう。

```
def countdown(n)
  while n > 0 do
    puts(n)
    n = n - 1
    sleep(1)
  end
end
```

puts は値を出力します。n = n - 1 は「n-1 を計算し、それを n に代入」するので、つまり n を 1 減らします。「sleep(数値)」は指定した秒数だけ実行を待ちます (Unix コマンドの sleep と同じ)。これを反復しますが、n が 0 になったら (0 は 0 より大きくないので) 終わります。

```
irb> countdown 5
5
4
3
2
1
=> nil
```

0 も表示して欲しい? それは直すのは簡単ですのでやってみてください。

2.3 数値積分

2.3.1 数値的に定積分を求める exam

もっと有用な繰り返しの具体的な題材として、**数値積分** (numerical integration — 定積分の値を数値を計算する方法で求めること) を取り上げてみましょう。皆様はこれまで、定積分を求めるのに、積分の公式を覚えたり、公式のあてはめや変形に苦労したりされてきたと思います。しかしプログラムを使えば、元の関数から直接定積分の値を計算してしまえるのです。

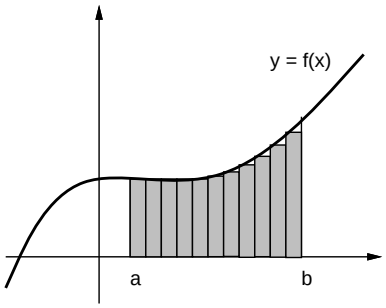


図 2.2: 数値積分の原理

関数 $y = f(x)$ の $x = a$ から $x = b$ までの定積分は図 2.2 のように、その関数のグラフを描くと、区間 $[a, b]$ の範囲における関数の下側の面積でした。そこで、図 2.2 にあるように、その部分に多数の細長い長方形を詰めて、その面積を合計すれば知りたい面積の値、つまり定積分の値が求まります。各長方形の幅は区間を n 等分した値 dx、高さは $f(x)$ の値なので、面積は容易に計算できます。

この方法であれば、$f(x)$ が数式として不定積分が求められなくても、定積分が計算できます。数式の形で一般的に解を求めることを**解析的** (analytical) に解くと言い、これと対比して数値で計算して特定の問題の解を求めることを**数値的** (numerical) に解くと言います。

とはいえ、今は「正しい」値が求まるかどうかチェックしたいので、簡単な関数 $y = x^2$ でやってみます。不定積分は $\frac{1}{3}x^3$ ですから、区間 $[a, b]$ の定積分は $[\frac{1}{3}x^3]_a^b$ ということになります。たとえば $[1, 10]$ だったら $\frac{1000}{3} - \frac{1}{3} = \frac{999}{3} = 333$ となります。ではアルゴリズムを作ってみましょう。

- integ1: 関数 x^2 の区間 $[a, b]$ の定積分を区間数 n で計算
- $dx \leftarrow \frac{b-a}{n}$。
- $s \leftarrow 0$。
- $x \leftarrow a$。
- $x < b$ が成り立つ間繰り返し、
- $y \leftarrow x^2$。 # 関数 $f(x)$ の計算
- $s \leftarrow s + y \times dx$。
- $x \leftarrow x + dx$。
- 繰り返し終わり。
- s を返す。

すなわち、x にまず a を格納しておき、繰り返しの中で $x \leftarrow x + dx$、つまり x に dx を足した値を作ってそれを x に入れ直すことで x を徐々に (dx きざみで) 動かしていき、b まで来たら繰り返しを終わります。このように、繰り返しでは「こういう条件で変数を動かしていき、こうなったら終わる」という考え方が必要なのです。面積のほうは、s を最初 0 にしておき、繰り返しの中で細長い長方形の面積を繰り返し加えていくことで、合計を求めます。

では Ruby プログラムを示しましょう。「#」の右側に書かれている部分は注記ないし**コメント** (comment) と呼ばれ、Ruby ではこの書き方でプログラム中に覚え書きを入れておくことができます。コー

ドの意味が分かりづらい (何のためにこのような計算をしているのか読み取りにくい) 箇所には、その意図を注記しておくようにしてください。また、一時的に命令を実行しないようにするのにも、コメントが便利に使えます。これを**コメントアウト** (comment out) と呼びます。この例でも後で使うコードをコメントアウトしてあります。

```
def integ1(a, b, n)
  dx = (b - a).to_f / n
  s = 0.0
  x = a
# count = 0
  while x < b do
    y = x**2        # 関数 f(x) の計算
    s = s + y * dx
    x = x + dx
#   count = count + 1
#   puts("count=#{count} x=#{x}")
  end
  return s
end
```

2 行目の「(b - a).to_f」というのは、b - a を計算した後、その結果を実数に変換するメソッドです。a も b も n も整数で指定された場合、切り捨て除算されると dx が正しくならないので、このようにしました。それも含め、やっていることは先の擬似コードそのままだと分かるはずです。さて、333 が求まるでしょうか? 実行させてみます。

```
irb> integ1 1, 10, 100
=> 337.557149999999            ←ふーん?
irb> integ1 1, 10, 1000
=> 332.554621500007            ←小さい
irb> integ1 1, 10, 10000
=> 333.045451214912            ←大きい…
```

なんだかヘンですね。そこで、繰り返しの回数がいくつになっているかをチェックすることにして、上の行頭の「#」を削って動かし直してみました。[8]

```
irb> integ1 1.0, 10.0, 100
...
count=98 x=9.81999999999999  ←誤差が…
count=99 x=9.90999999999999
count=100 x=9.99999999999999
count=101 x=10.09            ← 101 回目が…
=> 337.557149999999          ←このため多い
```

区間数 100 個なのに長方形を 1 個余計に加えたため、値が大きすぎるわけです。なぜこんなことが起きるのでしょう? それは「$x \leftarrow x + dx$ で x を増やしていき b になったらやめる」というアルゴリズムの問題なのです。そもそもコンピュータでの浮動小数点計算は近似値の計算なので、dx を区間長の $\frac{1}{100}$ にしたとしても、そこに誤差があります。このため 100 回足してもわずかに b より小さい場合があり、その時は余分に繰り返しを実行してしまいます。

[8] つまり、変数 count に回数を数えつつ x を表示するようにするわけです。このように、コメントアウトしてあったコードを活かして動かすことを「コメントアウトを外す」とも言います。

2.3.2 計数ループ exam

ではどうすればよいでしょうか。繰り返し回数を 100 回と決めているのですから、回数を数えるのは整数型で行い、[9] それをもとに各回の x を計算するのがよいのです。つまり、次のようなループを書くことになります。(**カウンタ** (counter) とは「数を数える」ために使う変数のことです。)

```
i = 0           # i はカウンタ
while i < n do # 「n 未満の間」繰り返し
  ...           #  ここでループ内側の動作
  i = i + 1     # カウンタを 1 増やす
end
```

このように指定した上限まで数えながら反復する繰り返しを**計数ループ** (counting loop) と呼びます。計数ループはプログラムで頻繁に使われるため、ほとんどのプログラミング言語は計数ループのための専用の機能や構文を持っています (while 文でも計数ループは書けますが、専用の構文のほうが書きやすくて読みやすいからです)。

Ruby では計数ループ用の構文として for 文 (for statement) を用意しています。これを使って上の while 文による計数ループと同等のものを書くと次のようになります。

```
for i in 0..n-1 do
  ...
end
```

これは、カウンタ変数 `i` を 0 から初めて 1 つずつ増やしながら `n-1` まで繰り返していくループとなります (多くのプログラミング言語では、計数ループを表すのに for というキーワードを使うので、計数ループのことを **for ループ**と呼ぶこともあります)。

せっかく for 文を説明しておきながら恐縮ですが、以下では計数ループを整数値が持つメソッド `times` を使って書くことにします。[10] これはたとえば次のようになります。

```
100.times do
  ...
end
```

この `times` も先の `to_f` などのように「値 x に対して何かをする」メソッドですが、さらにブロック (コードの並び、`do`〜`end` の部分) を受け取るようになっています。そしてそのブロックを数値の回数 (上の例では 100 回) 実行します。ブロックの指定のための `do` は省略できません。[11]

ところで、計数ループの中でカウンタの値 $(0, 1, 2, \ldots)$ を使いたいこともあります。このため、`times` は各繰り返しごとにカウンタ値をブロックにパラメタとして渡してくれます。上の例ではそれを受け取っていませんでしたが、ブロックの冒頭に「`|`名前`|`」という書き方でパラメタ (の列) を指定することで、このパラメタを受け取ることができます。複数ある場合は`|x, y|`のようにカンマで区切って並べます。たとえば次のようにすると、0 から 99 までの数を次々に出力することができます。

```
100.times do |i|
  puts(i)
end
```

いろいろありましたが、元に戻って擬似コードでは、計数ループを次のように記します。[12]

---

[9] 整数ならば、あふれない限り誤差はありません。

[10] なぜ for 文でなく `times` を使うかというと、ブロックを受け取るメソッドは Ruby でさまざまな用途に使える便利な仕組みなので、そちらに慣れたほうがよいと思うからです。

[11] それで混乱しやすいので、`while` でも `do` を省略しないことにしたわけです。

[12] 擬似コードはあくまでも「擬似」コードであり、Ruby に直した時に for 文か `times` かは特に指定しません。

- 変数 i を 0 から n の手前まで変えながら繰り返し、
- 　... # ループ内の動作
- 繰り返し終わり。

2.3.3　計数ループを用いた数値積分 exam

余談が終わったので、先の積分プログラムを計数ループを使うように直してみましょう。

- integ2: 関数 x^2 の区間 $[a, b]$ の定積分を区間数 n で計算
- 　$dx \leftarrow \frac{b-a}{n}$。
- 　$s \leftarrow 0$。
- 　変数 i を 0 から n の手前まで変えながら繰り返し、
- 　　$x \leftarrow a + i \times dx$。
- 　　$y \leftarrow x^2$。 # 関数 $f(x)$ の計算
- 　　$s \leftarrow s + y \times dx$。
- 　繰り返し終わり。
- 　s を返す。

先の例との違いは、毎回 x を i から計算する点です。これを Ruby にしたものも示します。

```
def integ2(a, b, n)
  dx = (b - a).to_f / n
  s = 0.0
  n.times do |i|
    x = a + i * dx
    y = x**2        # 関数 f(x) の計算
    s = s + y * dx
  end
  return s
end
```

これを動かしてみましょう。

```
irb> integ2 1.0, 10.0, 100
=> 328.55715
irb> integ2 1.0, 10.0, 1000
=> 332.5546215
irb> integ2 1.0, 10.0, 10000
=> 332.955451215
```

こんどはきざみを小さくすると順当に誤差が減少していきます。

しかし、常に正しい面積である 333 より小さいようですが、これはなぜでしょうか? それは、長方形の面積を計算するのに微小区間の左端の x を使って高さを決めるため、増大関数では図 2.3 のように微小な三角形の分だけ面積が小さめに計算されるからです (逆に減少関数だと大きめに計算されます)。これをもうちょっと何とかする方法については、演習問題にしたので、やってみてください。

演習 3　上の演習問題のプログラムを打ち込んで動かせ。動いたら「減少する関数だと値が大き目に出る」ことも確認せよ。できれば、左端ではなく右端で計算するのもやってみるとよい。その後、次のような考え方で誤差が減少できるかどうか、実際にプログラムを書いて試してみよ。

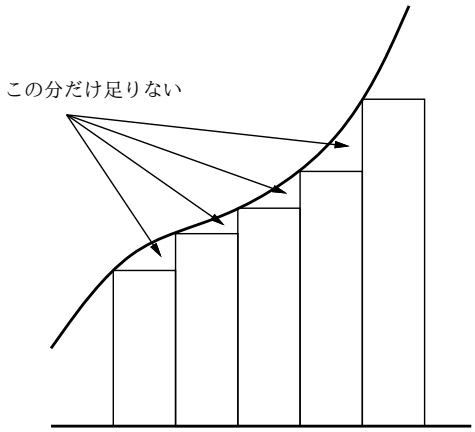


図 2.3: 区間の左端を使う場合の誤差

a. 左端の x だけでも右端の x だけでも弱点があるので、両方で計算して平均を取る。
b. 左端や右端だからよくないので、区間の中央の x を使う。
c. 上記 a と b をうまく組み合わせてみる。

演習 4 次のような、繰り返しを使ったプログラムを作成せよ。

a. 非負整数 n を受け取り、2^n を計算する。
b. 非負整数 n を受け取り、$n! = n \times (n-1) \times \cdots \times 2 \times 1$ を計算する。
c. 非負整数 n と $r(\leq n)$ を受け取り、${}_nC_r$ を計算する。

$$ {}_nC_r = \frac{n \times (n-1) \times \cdots \times (n-r+1)}{r \times (r-1) \times \cdots \times 1} $$

d. x と計算する項の数 n を与えて、次のテイラー展開を計算する。

$$ sinx = \frac{x^1}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots $$

$$ cosx = \frac{x^0}{0!} - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots $$

実際に値の分かる x を入れて精度を確認してみること。$\pm 10\pi$ とかだとどうか? n はいくつくらいが適切か?

2.4 制御構造の組み合わせ

少し込み入ったプログラムになると、ある制御構造 (枝分かれ、繰り返し) の内側にさらに制御構造を入れることになります。たとえば、

- もし～であれば、
- 　条件～が成り立つ間繰り返し、
- 　　○○をする
- 　以上を繰り返し。
- 枝分かれ終わり。

だと次のようになるわけです。

```
if ...
    while
        ...
    end
end
```

このように規則に従って要素を組み合わせて行くことで (単に並べるのも組み合わせ方のうち)、いくらでも複雑なプログラムが作成できます。これはちょうど、簡単な規則と単語からいくらでも複雑な文章が (日本語や英語で) 作れるのと同じです。

演習 5 a と b の最大公約数を $gcd(a,b)$ と記す。正の整数 x、y の $gcd(x,y)$ を求めることを考える。

- $x = y$ のとき、$gcd(x,y) = x = y$。
- $x > y$ のとき、$gcd(x,y) = gcd(x-y,y)$。
- $x < y$ のとき、$gcd(x,y) = gcd(x,y-x)$。

これを利用して、2 つの正の整数 x、y に対してその最大公約数を求めるアルゴリズムの疑似コードを書き、Ruby プログラムを作成せよ (なぜこれで求まるかも説明すること)。

演習 6 「正の整数 N を受け取り、N が素数なら `true`、そうでなければ `false` を返す Ruby プログラム」を書け。[13] まず疑似コードを書き、次に Ruby に直すこと。(ヒント: N が素数ということは、N を $2 \sim N-1$ のいずれで割っても余りが出るということ。剰余は演算子「`%`」で計算できる)。

演習 7 「正の整数 N を受け取り、N 以下の素数をすべて打ち出す Ruby プログラム」を書け。待ち時間 10 秒以内でいくつの N まで処理できるか調べて報告せよ。(もちろん N が大きくなるように工夫してくれるとなおよい。)

本日の課題 2A

「演習 2」で動かしたプログラム (どれか 1 つでよい) を含むレポートを提出しなさい。プログラムと、簡単な説明が含まれること。アンケートの回答もおこなうこと。

Q1. プログラムを打ち込んで動かすのに慣れましたか?

Q2. 自分にとって次の「難しいポイント」は何だと思いますか?

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 2B

「演習 2〜演習 7」の (小) 課題から選択して 1 つ以上プログラムを作り、レポートを提出しなさい。プログラムと、課題に対する報告・考察 (やってみた結果・そこから分かったことの記述) が含まれること。アンケートの回答もおこなうこと。

Q1. 枝分かれや繰り返しの動き方が納得できましたか?

Q2. 枝分かれと繰り返しのどっちが難しいですか? それはなぜ?

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

[13] `true`/`false` は「はい/いいえ」を表す値である。

＃3　制御構造＋配列とその利用

今回はまず、前回の課題の解説と併せて、数値積分や制御構造などで追加すべき点を説明します。その後の本日の内容としては、次のものを取り上げます。

- 制御構造の組み合わせについて (再)
- データ構造と配列

3.1　前回演習問題の解説

3.1.1　演習 2a — 枝分かれの復習

演習 2a は例題とほとんど同じです。まず擬似コードを見てみましょう。

- max2: 数 a、b の大きいほうを返す
- 　もし a > b であれば、
- 　　$result \leftarrow a$。
- 　そうでなければ、
- 　　$result \leftarrow b$。
- 　枝分かれ終わり。
- 　$result$ を返す。

Ruby では次のとおり。

```
def max2(a, b)
  if a > b
    result = a
  else
    result = b
  end
  return result
end
```

これも、次のような「別解」があり得ます。

- max2x: 数 a、b の大きい方を返す
- 　$result \leftarrow a$。
- 　もし $b > result$ であれば、
- 　　$result \leftarrow b$。
- 　枝分かれ終わり。
- 　$result$ を返す。

これの Ruby 版は次のとおり。

```
def max2x(a, b)
  result = a
  if b > result then result = b end
  return result
end
```

どちらが好みですか? これもどちらが正解ということはありません。

ところで、「2 数が等しい場合はどうするのか」について皆様の中には迷った人がいると思います。問題には「異なる数」と書いてあるので考えなくてもよいのですが、仮にそれが書いていなかったとします。そうなると、等しい場合について何らかの指示が本来あるべきですよね。たとえば次のものがあり得ます。

- 「等しい場合はその等しい数を返す」
- 「等しい場合は何が返るかは分からない」
- 「等しい数を渡してはならない」

上 2 つの場合は例解のままで OK です (2 番目では何が返ってもよいので、等しい数でもよい)。最後の場合はどうでしょう。次の考え方があり得ます。

(a) 「渡してはならない」以上、渡されることはないのだから、例解のままでよい

(b) 「渡してはならない」値が渡されたのだから、エラーを表示するなどして警告するべき

どちらにも (互いに裏返しの) 利点と弱点があります。(a) の方が簡潔で短く間違いが起きにくいですが、(b) の方が起きるべきでないことが起きていることが分かるので対処が必要な場合には有用です。

で、あなたは発注者 (教員) の注文を受けてこの課題をやっているわけですから、正解は発注者に「どうしますか」と確認することです。そうすれば、どちらにするかは決められるでしょう。勝手に (b) を選んでプログラムを複雑で間違いやすいものにするのはいかがかと思いますし、発注者が「等しい場合はその等しい数を返す」と書き忘れただけだったら目もあてられませんね。

3.1.2 演習 2b — 枝分かれの入れ子

演習 2b はもう少し複雑です。まず考えつくのは、a と b の大きいほうはどちらかを判断し、それぞれの場合についてそれを c と比べるというものでしょうか。

- max3: 数 a、b、c で最大のものを返す
- 　もし $a > b$ であれば、
- 　　もし $a > c$ であれば、
- 　　$result \leftarrow a$。
- 　　そうでなければ、
- 　　$result \leftarrow c$。
- 　　枝分かれ終わり。
- 　そうでなければ、
- 　　もし $b > c$ であれば、
- 　　$result \leftarrow b$。
- 　　そうでなければ、
- 　　$result \leftarrow c$。
- 　　枝分かれ終わり。
- 　枝分かれ終わり。
- 　$result$ を返す。

かなり大変ですね。これを Ruby にしたものは次のとおり。

```
def max3(a, b, c)
  if a > b
    if a > c
      result = a
    else
      result = c
    end
  else
    if b > c
      result = b
    else
      result = c
    end
  end
  return result
end
```

こうなると字下げしてないとごちゃごちゃになるでしょう? しかし字下げしてあってもこれはかなり苦しいですね。一般に、if の中に if を入れると非常に分かりづらくなるので、できるだけ避けたほうがよいのです。

ところで、先の別解から発展させるとどうなるでしょう?

- max3x: 数 a、b、c で最大のものを返す
- result ← a
- もし $b > result$ であれば、$result \leftarrow b$。
- もし $c > result$ であれば、$result \leftarrow c$。
- $result$ を返す。

「もし」の擬似コードが 1 行に書かれていますが、この場合はこちらののほうが見やすいと思ったのでそうしてみました。Ruby でも次のとおり (こんどはどちらが好みですか?)。

```
def max3x(a, b, c)
  result = a
  if b > result then result = b end
  if c > result then result = c end
  return result
end
```

一般には、枝分かれの中に枝分かれを入れるよりは、枝分かれを並べるだけで済ませられればそのほうが分かりやすいと言えます。また、この方法では入力の数 N がいくつになっても簡単に対処できるという利点があります。

実は、さらなる別解があります。それは、既に `max2` を作ったわけですから、それを利用するというものです。

```
def max3xx(a, b, c)
  return max2(a, max2(b, c))
end
```

このように、一度作って完成したものは後から別のものを作る時の「部品」として使える、というのは重要な考え方です。このことも覚えておいてください。

3.1.3 演習 2c — 多方向の枝分かれ

演習 2c は 3 通りに分かれるので、if の中にまた if が入るのはやむをえないはずです。Ruby コードを見てみましょう。

```
def sign1(x)
  if x > 0
    return "positive."
  else
    if x < 0
      return "negative."
    else
      return "zero."
    end
  end
end
```

このような「複数の条件判断」はよく使うので、実はこれは `if` の入れ子にしなくても書けるようになっています。具体的には、`if` 文には「`elsif` 条件 `then` 動作」という部分を途中に何回でも入れられ、それを使うと次のようになります。

```
def sign2(x)
  if x > 0
    return "positive."
  elsif x < 0
    return "negative."
  else
    return "zero."
  end
end
```

順序が前後しましたが、擬似コードだと次のようになります。

- sign2: 数 x の正/負/零に応じて positive/negative/zero を返す
- もし x $>$ 0 ならば、
- 　　「positive.」を返す。
- そうでなくて x $<$ 0 ならば、
- 　　「negative.」を返す。
- そうでなければ、
- 　　「zero.」を返す。
- 枝分かれ終わり。

「そうでなくて〜ならば、」は何回現われても構いません。また、そのどれもが成り立たない場合は「そうでなければ」に来るわけですが、この部分は不要なら無くても構いません。

ところで、最大値の問題にちょっと戻ると、複合条件を使えば「`a > b && a > c`」なら `a` が最大だと分かりますから、これを利用した 3 方向枝分かれで書くこともできます (変数を使わず値を返すスタイルにしてみました)。

```
def max3y(a, b, c)
  if a > b && a > c
```

```
    return a
  elsif b > c
    return b
  else
    return c
  end
end
```

ただし、この方法でも N が 4、5 と増えてくると条件の中の比較演算が増えて、一般に N^2 に比例してしまいます。だからいけないというわけではなく、N の個数が多くなければ、このやり方を使ってもよいかも知れません。

3.1.4 演習 3a〜3c — 数値積分

長方形の高さとして区間の左端の $f(x)$ を使うと増大関数で値が小さくなり、右端の $f(x)$ を使うと大きくなるので、「左端と右端の平均を取って」みるという課題でした (減少関数だと逆に大きく/小さくなります)。これは考えてみると、面積を計算するのにその区間の関数を直線で補間した「台形」を考え、その面積を求めているのと同等です。このため、これを数値積分の**台形公式** (trapezoid rule) と呼びます。

台形公式の計算内容は次のようになります (区間の幅を d で表す)。

$$s = \sum \frac{1}{2}\{f(x) + f(x+d)\}d$$

これを計算する Ruby プログラムを示しておきます。

```
def integ3(a, b, n)
  dx = (b - a).to_f / n
  s = 0.0
  n.times do |i|
    x = a + i * dx
    y0 = x**2
    y1 = (x+dx)**2
    s = s + 0.5*(y0+y1) * dx
  end
  return s
end
```

台形公式は直線による補間なので、曲線が上に凸だと値は小さく、下に凸だと値は大きくなります。一方、これも演習にありましたが、区間の中央の x を使って長方形で計算すると (これを**中点公式**と言います)、逆に上に凸だと大きく、下に凸だと小さくなります (図 3.1)。だからこれをちょうどよく混ぜたらよいのでは、というのが演習 3c になっていたわけです。実は、左端:中央:右端を 1:4:1 で混ぜると (つまり台形:中点を 1:2 で混ぜると) よい結果が得られます。プログラムも示しておきます。

```
def integ4(a, b, n)
  dx = (b - a).to_f / n
  s = 0.0
  n.times do |i|
    x = a + i * dx
    y0 = x**2
    y1 = (x+0.5*dx)**2
    y2 = (x+dx)**2
```

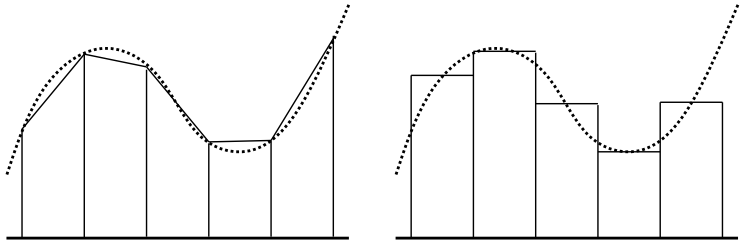


図 3.1: 台形公式と中点公式

```
    s = s + (y0+4*y1+y2) * dx / 6.0
  end
  return s
end
```

実際に計算させてみましょう (「正解」は 333 だったことに注意)。

```
irb> integ4 1, 10, 100
=> 333.0                ←ぴったり? 本当?
irb> printf "%.20g\n", integ4(1, 10, 100)
332.99999999999994316  ←確かにすごくよい
=> nil
irb> printf "%.20g\n", integ4(1, 10, 10)
333          ←分割数を減らしたら逆にぴったり?
=> nil
```

この計算式は**シンプソンの公式** (Sympson rule) と言われ、数値積分では標準的な方法です。[1]

$$s = \sum \frac{1}{3}\{f(x) + 4f(x+d) + f(x+2d)\}d$$

なぜこれがよいかというと、当該区間を 2 次曲線で補間することになるからです。だから積分しようとしている関数が 2 次以下の多項式だと「ぴったし」になり、そのため上の例では区間数が少ないほど (誤差が出ないため) よかったわけです。実際、分割数 1 でもぴったりなので、もはや数値積分と言えないような…

2 次式の補間になる理由を示しておきます。当該区間の曲線を 2 次式

$$y = ax^2 + bx + c$$

で表せるものとします。また区間の幅を $2d$、左端を x_0、中央を $x_1 = x_0 + d$、右端を $x_0 + 2d$、対応する関数値を y_0、y_1、y_2 とおきます。上の 2 次式の不定積分は $\frac{1}{3}ax^3 + \frac{1}{2}bx^2 + cx$ ですから、面積 (定積分) は次のようになります。

$$s = \left[\frac{1}{3}ax^3 + \frac{1}{2}bx^2 + cx\right]_{x_0}^{x_0+2d}$$

これを整理すると次のようになります。

$$3s = \{a(6{x_0}^2 + 12x_0d + 8d^2) + b(6x_0 + 6d) + 6c\}d$$

ところで

$$y_0 = a{x_0}^2 + bx_0 + c$$

[1] この式では見やすくするため区間の半分を d としていて、そのためプログラムの 6 で割る代わりに 3 で割っています

$$y_1 = a(x_0 + d)^2 + b(x_0 + d) + c$$

$$y_2 = a(x_0 + 2d)^2 + b(x_0 + 2d) + c$$

なので、見比べると次の式が成り立つと分かります。

$$3s = (y_0 + 4y_1 + y_2)d$$

というわけで、上の式が出て来るわけです。

数値積分にはシンプソンの公式が一番よいのかというと、必ずしもそうとは言えません。たとえば、ある細かさで積分を計算して、もっと細かくするために d を半分にしたいと思ったとすると、台形公式では既に計算した値をとっておいて、新たに加えた半分ずつの点についての計算を追加すれば済みます。このような計算方法を**漸近的**と言います。漸近的に計算していき、値の変化がなくなったらこれ以上細かさを増やしても意味がないと判断してやめるというのは 1 つの方法です。

3.1.5 演習 4a〜4c — 繰り返し

この辺は簡単なのでプログラムだけ示します (べき乗は計算するだけなら `2**n` でよいのですが、繰り返しを使うという前提なのでループを使います)。

```
def pow2(n)
  result = 1
  n.times do result = result * 2 end
  return result
end

def fact(n)
  result = 1
  n.times do |i| result = result * (i+1) end
  return result
end
```

階乗の方は「1 × 2 ×…× N」を計算したいわけですが、`times` が渡して来るカウント値は「0, 1, …, N-1」なので、全部 1 足してから掛けています。しかしそれはちょっと分かりにくいですね。

実は、N`.times` の代わりに N`.step(`M`,` d`)` という別のメソッドを使うと、初項 N、終項 M、増分 d を指定して計数ループを作ることができます (d は指定しないと「1」が使われます)。これを使えば、階乗は次のようにもっと分かりやすくなります。

```
def factx(n)
  result = 1
  1.step(n) do |i| result = result * i end
  return result
end
```

上の `step` は「`i` を 1 から `n` まで 1 ずつ増やしながら」という擬似コードに対応します。

組み合わせの数を整数で計算できるようにするためには「小さい側から」掛けて・割って・掛けて・割ってのようにしないとうまくいきません。$\frac{4\times5\times6\times7}{1\times2\times3\times4}$ のように並べて左から 1 列ずつ乗算・除算の順で計算するわけです。この順序でやれば、除算が常に割り切れるので、誤差なしで計算できます (浮動小数点で計算してしまうと、誤差が現れるのでいまいちだと思います)。

```
def comb(n, r)
  result = 1
  1.step(r) do |i|
    result = result * (i + (n-r)) / i
  end
  return result
end
```

3.1.6 演習 4d — テイラー級数で sin と cos を計算

これは「階乗や x^n を計算しつつ」足していくのでちょっと面倒ですね。しかも交互に+/-が変わることも扱う必要があります。

```
def sincos(x, n)
  sign = 1; pow = 1.0; fact = 1; sin = 0.0; cos = 0.0
  n.times do |i|
    cos = cos + sign * pow / fact
    pow = pow * x
    fact = fact * (2*i+1)
    sin = sin + sign * pow / fact
    pow = pow * x
    fact = fact * (2*i+2)
    sign = -sign
  end
  return [sin, cos]
end
```

このプログラムでは、べき乗の数を 2 ずつ増やしながら sin と cos のテイラー展開を並行して計算しています。計算式を再録しておきましょう。

$$sinx = \frac{x^1}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

$$cosx = \frac{x^0}{0!} - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

「何項まで計算するか」によって精度が変わって来ますが、言い換えれば実用のプログラムでは項を無限に計算することはできず、どこかで打ち切る必要があります。ということは、打ち切ったその先の項の値のぶんは無視されて誤差となわけです。これを**打ち切り誤差** (cutoff error) といい、既に学んだ丸め誤差、情報落ち、桁落ちと並んで数値計算における誤差の要因の 1 つです。

では実際に計算してみます。

```
irb> Math::PI / 3
  => 1.0471975511966  ← π/3 (60 度) はこの値)
irb> sincos 1.0471975511966, 5   ←π/3 の sin,cos
  => [0.866025445099782, 0.500000433432913] ←微妙
irb> sincos 1.0471975511966, 10  ←項を増やすと
  => [0.86602540378444, 0.499999999999998] ←まあ OK
irb> sincos 3.141592653589, 10   ←π だと
=> [-5.2812499185062e-10, -1.00000000352908] ←微妙
irb> sincos 31.41592653589, 10   ← 10 π
=> [-167876715320.415, -104528895953.392] ←破綻
```

何が問題なのでしょう? それは、x が大きくなるほどテイラー級数の収束が遅くなるためです。これに対処するため、sin とか cos が周期関数であることを利用し、この方法で計算するのは絶対値の小さい $0 \leq x \leq \frac{\pi}{4}$ の範囲だけにすべきでしょう (この範囲の sin と cos があれば残りの範囲は全部これらをもとに計算できますから)。

そして、x の範囲をこのように限定するなら、テイラー級数の項の数は 8 つくらいあれば十分と分かります (その先の項は分子の絶対値が 1 より小さく、分母は 10^{10} 以上になるので、そこで打ち切っても精度は十分です)。その場合、加えていく順序をテイラー級数の後ろの項から順にしたほうがよいのです。と言うのは、後ろのほうほど絶対値が小さくなるので、前から順に足すと情報落ちしやすくなります。項の数を決めておけば、後ろから足すように書くのも簡単です。

3.2 制御構造の組み合わせ (再) exam

簡単なプログラムでは制御構造として「枝分かれ」「繰り返し」のどちらかを 1 つだけを使えば済みますが、もう少し込み入ったプログラムになると、ある制御構造 (枝分かれ、繰り返し) の内側にさらに制御構造を入れることになります。たとえば、「0〜99 の数を順に打ち出すが、ただし 3 の倍数の時だけは fizz と打ち出す」という例を考えてみます。[2]

- fizz1: 3 の倍数の時だけ fizz
- 　変数 i を 0 から 100 の手前まで変えながら繰り返し、
- 　　もし i が 3 の倍数ならば、
- 　　「fizz」と出力。
- 　　そうでなければ、
- 　　i を出力。
- 　　枝分かれ終わり。
- 　以上を繰り返し。

これを Ruby に直したものは次のようになります。

```
def fizz1
  100.times do |i|
    if i % 3 == 0
      puts('fizz')
    else
      puts(i)
    end
  end
end
```

動かしてみましょう。

```
irb> fizz1
fizz
1
2
fizz
```

[2]海外で古くからある言葉遊びに **fizzbuzz** というのがあります。これは輪になって「1, 2, . . .」と順に数を唱えますが、ただし数が 3 の倍数なら「fizz」、5 の倍数なら「buzz」、3 と 5 の公倍数なら「fizzbuzz」と (数の代わりに) 言わなければならず、間違えたりつっかえたりしたら負けで輪から抜ける、というものです。日本で有名なのは世界のナベアツの「3 の倍数と 3 がつく数字の時だけアホになります」というネタですが、ナベアツも fizzbuzz をヒントにこのネタを考案したという説があります。

```
(途中略)
97
98
fizz
=> 100
irb>
```

このように、基本的な制御構造を組み合わせていけば、いくらでも複雑なプログラムが作成できます。これはちょうど、簡単な規則と単語からいくらでも複雑な文章が (日本語や英語で) 作れるのと同じだと考えてください。

演習 1 上の fizz プログラムを打ち込んでそのまま動かせ。動いたら、繰り返しと枝分かれを組み合わせて次の動作をする Ruby プログラムを作成せよ。

a. 0 から 99 までの数のうち、2 の倍数でも 3 の倍数でもないものだけを順に打ち出す。
b. 0 から 99 までの数を順に打ち出すが、ただし 3 の倍数の時は fizz、5 の倍数の時は buzz、3 の倍数かつ 5 の倍数の時は fizzbuzz と (数値の代わりに) 打ち出す (fizzbuzz 問題)。[3]
c. 0 から 99 までの数を順に打ち出すが、ただし 3 の倍数と 3 がつく数字の時は数値の代わりに aho と打ち出す。

以下の 3 問は前回の演習 5〜7 と (ほぼ) 同じなので、やってしまった人はご勘弁ください。というか、今回解説する時間がないので次回解説するため、ここに再録しています。

演習 2 2 数 a、b の**最大公約数** (greatest common divisor、GCD) を求めるアルゴリズムを次に示す。

- gcd1: 整数 x、y の最大公約数を返す
- 　x ≠ y である間繰り返し、
- 　　x ＞ y なら、
- 　　　x ← x - y。
- 　　そうでなければ、
- 　　　y ← y - x。
- 　　枝分かれ終わり。
- 　繰り返し終わり。
- 　x を返す。

これを Ruby プログラムにして動かせ。これで最大公約数が求まる理由も併せて説明すること。(ヒント: $x > y$ ならば $gcd(x, y) = gcd(x - y,\ y)$ 等のこと (つまり x と y の大きいほうから小さいほうを引いても 2 数の最大公約数は変化しないこと) を示せばよいわけですね。)

演習 3 「正の整数 N を受け取り、N が素数か否かを (`true`/`false` で) 返す Ruby プログラム」を書け。まず擬似コードを書き、それから Ruby に直すこと。(ヒント: N が素数ということは、N を 2〜$N-1$ のいずれで割っても割り切れない、つまり剰余が 0 でないということ。剰余は演算子`%`で計算できるのでしたね。)

演習 4 「正の整数 N を受け取り、N 以下の素数をすべて打ち出す Ruby プログラム」を書け。待ち時間 10 秒以内でいくつの N まで処理できるか調べて報告せよ。N が大きくなるように工夫してくれるとなおよい。(ヒント: 処理を速くするためには、(1) 割ってみる数をできるだけ少なくとどめる、(2) 素数の候補とする数をできるだけ少なくとどめる、という 2 点を工夫するとよいでしょう。たとえば 2 は別扱いして奇数だけ扱うなど。)

[3] fizzbuzz 問題については、「(米国で) プログラマを募集して応募者にこの問題のプログラムを書かせたら書けない人だらけだったので、応募者のふるい分けに使っている」という都市伝説 (?) があります。

3.3 配列とその利用

3.3.1 データ構造の概念と配列 exam

ここまででではプログラムが扱うデータは個々の「値」であり、1 つの変数に 1 つの値が入っていました。しかしこのやり方では、大量のデータを扱うのが困難なのは明らかです。ではどうするかというと、複数のデータを組にしたり、列として並べるなどの「構造」を持たせて扱う、というのが答えです。この、データに持たせる構造のことを**データ構造** (data structure) と言います。

プログラミング言語の用語では、データの種類のことを**データ型** (data type)、その中で「整数」「実数」など単一の値から成るものを**基本型** (primitive data type) と呼びます。それと対比して、組や列など複数の値が集まったデータのことは**複合型** (compound data type) と呼びます。実は文字列は、中に複数の文字が含まれているので複合型だといえます。

今回は複合型のうちでもよく使われる**配列** (array) を取り上げます。配列は既に「`[1, 2, 3]` のように値を並べたもの」として言及したことがありますが、要するに値が一列に並んだものです。図 3.2 のように、整数であれば 1 つの変数に 1 つの値しか入れられませんが、配列を使うことで 1 つの変数に一連の値を入れることができます。

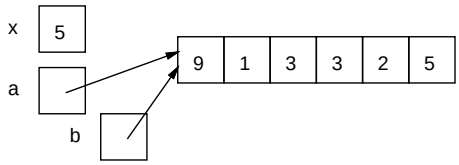


図 3.2: 配列の概念

図 3.2 を見て不思議に思ったことはないでしょうか。具体的には、基本型では変数の位置に「箱」が書かれていてそこに値が入っていますが、複合型では少し離れたところにデータを入れる場所があって、変数からはそこに矢印が出ています。

実はこの矢印はデータのありかを示す**参照** (reference — ありかを指す値で、実体はメモリ上の番地だと思ってよい) です。そして、変数に配列を入れると、配列本体はどこか別の場所に置かれ、変数にはその場所への参照が入ります。そして、「`b = a`」のように変数間で代入をした時、基本型では値 (箱の中身) がコピーされますが、複合型では参照 (矢印) がコピーされるだけで、本体は 1 つのまま (単に 2 つの変数が同じ場所を指すだけ) です。[4]

さらに、「2 つの変数が同じ場所を指している」状態でその複合データの中身を書き換えると、複合データは 1 つだけなので、どちらの変数から見た複合データも同じように変化していることになります。このあたりの挙動は間違えやすいので注意が必要です。

3.3.2 配列の生成 exam

配列を使うには、まず配列を作り出す必要があります。その方法が色々ありますので、ここではそれらについて説明しておきます。

```
a = [1, 2, 3]                # 直接指定
a = Array.new(10, 0)         # 要素数と初期値
a = Array.new(10) do 0 end   # 要素数とブロック
a = Array.new(10) do |i| 2*i end  # 〃
```

1 番目の方法はこれまでにも使ってきた、各要素を直接指定する方法です。この方法は、比較的少数の値を用意する場合に使います。

[4] Ruby の場合。C 言語はまた違います。

2 番目は、要素数と初期値を指定する方法で、要素数の多い配列を用意するときにはこの方法が一番単純です。[5]

3・4 番目も、要素数と初期値を指定しますが、初期値として値を計算するブロック (do〜end) を指定します (この場合はブロックの中で式を直接指定し、メソッドではないので return は書けません)。0 などと定数を指定した場合は 2 番目と変わりませんが、ブロックは (times などと同様)「何番目」というパラメタを受け取ることができ、それを用いて計算により初期値を決められます。

配列は後からメソッド `push` で要素を追加できます。上の例の 4 番目と次は同じ結果になります。

```
a = []                                  # 0 要素の配列を作り
10.times do |i| a.push(2*i) end  # 0〜18 を追加
```

現在の**配列の長さ** (要素数) は、メソッド `length` で取得できます。上の例では `a.length` は 10 です。

3.3.3　配列の利用 exam

いちど用意してしまえば、配列の個々の要素は 1 つの変数と同様に扱えます。ここで「どの要素か」を指定するのに `[...]` の中に式を書いて指定します。これを**添字** (index) と呼びます。たとえば上の例だと `a[0]`〜`a[9]` という要素があることになります (0 番目から数えることは慣れないと忘れやすいので注意)。

また、Ruby ではまだ用意していない添字番号 (たとえば上で「10 番」とか) の要素を参照すると `nil` が返ります。飛び離れた添字番号 (たとえば上で「100 番」とか) に値を格納すると、そこまでの途中の要素は全部 `nil` で埋められます。

では、配列を与えてその合計を求めるというのをやってみましょう (合計は積分とかで散々やったので簡単ですね)。

- arraysum：配列 a の数値の合計を求める
- 　sum ← 0。
- 　i を 0 から配列要素数の手前まで変えながら繰り返し、
- 　　sum ← sum + a[i]。
- 　繰り返し終わり。
- 　sum を返す。

ループの初回では `i` は 0 なので、`sum` と `a[0]` を足し、それを `sum` に入れます。次は `i` は 1 なので、`sum` と `a[1]` を足し…のように続きます。Ruby コードは次のとおり。

```
def arraysum(a)
  sum = 0
  a.length.times do |i|
    sum = sum + a[i]
  end
  return sum
end
```

動かした様子を示します。また、動いている途中の変数の変化を図 3.3 左に示します。このように、繰り返しの中で添字に用いる変数 `i` を 1 つずつ変化させるというのが定石です。

```
irb> arraysum([1,3,5,7,9])
=> 25
```

実は Ruby では「配列の各要素を取りながら周回するループ」というのもあって、そのほうが少し簡単になります。コードだけ示しておきます (変数の変化は図 3.3 右)。

[5] 初期値を指定しないと各要素の初期値は `nil` になります。

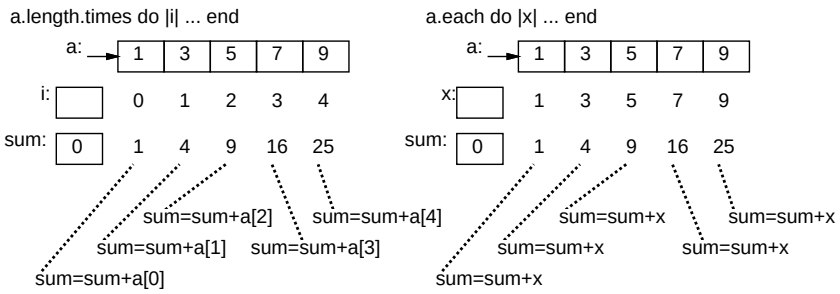


図 3.3: 配列の合計が動く様子

```
def arraysum1(a)
  sum = 0
  a.each do |x|      # x に配列の各要素が順次入る
    sum = sum + x
  end
  return sum
end
```

合計ならこのほうが少し簡単ですが、「何番目」を必要とする場合もあるので、その場合には計数ループを使うことになるでしょう。[6]

演習 5 上記の配列合計プログラムの好きな方をそのまま打ち込んで動かせ。動いたらこれを参考に下記のような Ruby プログラムを作れ。[7]

a. 数の配列を受け取り、その最大値を返す。
b. 数の配列を受け取り、最大値が何番目かを返す。なお先頭を 0 番目とし、最大値が複数あればその最初の番号が答えであるとする。
c. 数の配列を受け取り、最大値が何番目かを出力する。なお先頭を 0 番目とし、最大値が複数あればそれらをすべて出力する。
d. 数の配列を受け取り、その平均より小さい要素を出力する (例: 1、4、5、11 → 1、4、5)。
e. 数の配列を受け取り、その内容を「小さい順」に並べて出力する (例: 4, 11, 5, 1 → 1, 4, 5, 11)。

演習 6 「素数列挙」の問題は、配列を使うとより高速にできる可能性がある。次の 2 つの方針を用いたプログラムを作成し、これまでに作ったものと速度を比較せよ。

a. 素数は値の大きいところではまばらにしかないので、これまでに見つかった素数を配列に覚えておき、新たな素数の候補をチェックする時に「これまで見つかった素数で割ってみて割り切れなければ素数」という方針にすれば、チェックする回数がかなり少なくできる。
b. 別の考え方として、N 未満の素数を打ち出すのに次の方針を用いるのはどうだろう。[8][9]
   - 論理値が並んだ要素数 N の配列を作り、全部「真」に初期化。
   - 2 から始めて順次、その番号が「真」の値は素数として出力。
   - 2、4、6、…と、2 の倍数番目の部分を「偽」に変更。
   - 3、6、9、…と、3 の倍数番目の部分を「偽」に変更。
   - 同様に、素数を出力するごとにその倍数番目を「偽」に変更。

[6] メソッド `each_index` で配列の添字を順次取り出してループすることもできます。
[7] 「返す」の場合は上の例と同様に `return` を使い、「出力する」の場合は `puts` を使って画面に直接 (その場で) 出力させてください。`return` は使った瞬間にそのメソッド呼び出しは終ってしまうので、複数回 `return` を使うことはできません。
[8] これは「方針」であって、まだ擬似コードでもないことに注意してください。
[9] この方法を考案したのはギリシャの哲学者エラトステネス (Eratosthenes) であり、この方法を彼の名前を冠して**エラトステネスのふるい** (sieve of Eratosthenes) と呼びます。なぜ「ふるい」かというと、素数でないもの (各数の倍数) をふるい落としてしまうと、残ったものは素数だ、という方針でできているからです。

本日の課題 3A

「演習 1」で動かしたプログラム (どれか 1 つでよい) を含むレポートを提出しなさい。プログラムと、簡単な説明が含まれること。アンケートの回答もおこなうこと。

Q1. 制御構造の組み合わせができるようになりましたか。

Q2. 配列について学びましたが、使えそうですか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 3B

「演習 1」〜「演習 6」の (小) 課題から選択して 1 つ以上プログラムを作り、レポートを提出しなさい。配列の内容を含むことを強く勧めます。プログラムと、課題に対する報告・考察 (やってみた結果・そこから分かったことの記述) が含まれること。アンケートの回答もおこなうこと。

Q1. 配列が使いこなせるようになりましたか。

Q2. 疑似コードを書くのと、Ruby に直すのと、打ち込んで動かすのとで掛かった手間の比率を教えてください。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

3.4 付録: ruby コマンドによる実行

ここまで Ruby プログラムの実行に `irb` を使って来ましたが (この先もそうです)、プログラムが完成して実用に使う時は **ruby コマンド**を使うのが普通です。この場合、「`ruby なんとか.rb`」で直接プログラムが実行されます。そうすると、これまでのように `load` した後実際に動かす部分が入力できませんが、それもファイルの末尾に次のように書いておきます。

```
def triarea(w, h)
  return (w * h) / 2.0
end
w = ARGV[0].to_f; h = ARGV[1].to_f; puts(triarea(w, h))
```

しかしこの **ARGV** とは? これは「**コマンド引数配列**」で、`ruby` コマンドでファイル名の後ろに指定したパラメタが「文字列として」並んだ配列です (文字列なので数値として扱うためには `to_f` や `to_i` が必要)。そして普通にメソッドを呼びますが、結果も表示するためには `puts` や `printf` などを呼ぶ必要があります。実行の様子を見ましょう。

```
% ruby triarea.rb 7 5
17.5
```

なお、さらに 1 行目にインタプリタ指定「`#!/usr/bin/ruby`」(`ruby` コマンドの絶対パスはシステムごとに違うので「`which ruby`」で確認してください) を追加し、ファイルを実行可能にしておけば、ファイルを直接コマンドとして扱えます。

```
% ./triarea.rb 9 3
13.5
```

このように、Ruby でプログラムを書いても C などのマシン命令を生成するコンパイラと変わらない使い方ができるようになるわけです。

＃4　手続きと抽象化+再帰呼び出し

今回の主な内容は次の通りです。

- 手続き (メソッド) による抽象化
- 再帰的な呼び出し

手続きが使えるとプログラムを見通しよく書けるようになります。また、「再帰」の考え方を使うと、複雑な問題がこなせるようになります。ぜひマスターしてください。

4.1　前回演習問題の解説

4.1.1　演習 1 — fizzbuzz

演習 1 は繰り返しと枝分かれの基本的な組み合わせなので、さっさと Ruby コードを示します。 まず 2 の倍数と 3 の倍数以外を打ち出すものから。

```
def fizz2
  100.times do |i|
    if i % 2 != 0 && i % 3 != 0
      puts(i)
    end
  end
end
```

条件が読みにくいかもしれませんが、「2 の倍数でなく、3 の倍数でもないもの」を打ち出すと考えれば、これでよいと分かります。別案として、条件を変換するかわりに「素直に」枝分かれしてしまい、打ち出さないのは「何もしない」という案もあります。「何も書いてない」のは不安なので、「何もしない」というコメントを書いてあります。この方が分かりやすいでしょうか。

```
def fizz2
  100.times do |i|
    if i % 2 == 0
      # do nothing
    elsif i % 3 == 0
      # do nothing
    else
      puts(i)
    end
  end
end
```

演習 1b は if-else の連鎖で「3 の倍数」「5 の倍数」「3 と 5 の公倍数 (15 の倍数)」「それ以外」の 4 つに場合分けするのが一番素直です。

```
def fizzbuzz1
  100.times do |i|
```

```
    if i % 15 == 0
      puts('fizzbuzz')
    elsif i % 3 == 0
      puts('fizz')
    elsif i % 5 == 0
      puts('buzz')
    else
      puts(i)
    end
  end
end
```

なぜ15の倍数を最初に調べるのでしょう。それは、else-if の連鎖では上から順に条件を調べるで、先に「3の倍数」や「5の倍数」を調べると、15の倍数は3と5の倍数でもあるのでその枝が選ばれて、15の倍数の枝には行かなくなるからです。

せっかく「3の倍数なら fizz」「5の倍数なら buzz」「両方の倍数なら fizzbuzz」なので、両者をうまく組み合わせたいと思うかもしれません。やってみましょう。[1]

```
def fizzbuzz2
  100.times do |i|
    num = i
    if i % 3 == 0
      print('fizz'); num = ''
    end
    if i % 5 == 0
      print('buzz'); num = ''
    end
    print(num); puts
  end
end
```

変数 num に打ち出す数を入れますが、3の倍数なら fizz、5の倍数なら buzz を打ち出してから num に空文字列を入れ直すので、最後の print で何も出力しなくなります。3や5の倍数でないなら num に入れた i がそのまま出力されます。パラメタなしの puts は改行のためです。

このコードはよくできていると思いますか? 個人的には、このコードは先のコードより分かりづらいと思います。2つの if の境界で合流があり、数の出力を抑制するために変数 num も使いますから、流れを追うのが難しくなります。それよりも、4方向に枝分かれしてそれぞれの場合を明快に処理する先のコードのほうが読みやすくてスマートだと思いませんか?

最後は「世界のナベアツ」ですが、「3がつく数」はどうしましょうか。それは、対象とする数が1桁または2桁なので、「3がつく」というのは1桁目が3か、2桁目が3という意味になります。1桁目が3というのは、10で割った余りが3ということですし、2桁目が3というのは、10で切捨て除算した結果が3ということですね。ここまで分かればあとは書くだけです。

```
def fizz3
  100.times do |i|
    if i % 3 == 0 || i % 10 == 3 || i / 10 == 3
      puts('aho')
```

[1] print は puts と同様に文字列や数値を打ち出すけれども、行換えはしないメソッドです。なぜこれを使うかというと、fizz と buzz をくっつけて1行に打ち出すためです。

```
      else
        puts(i)
      end
    end
  end
```

4.1.2 演習 2 — 最大公約数

課題の擬似コードを Ruby に直したものは次のとおり。

```
def gcd1(x, y)
  while x != y
    if x > y
      x = x - y
    else
      y = y - x
    end
  end
  return x
end
```

なぜこれで最大公約数が求まるのでしょう? 次のように考えます (x と y は正の整数とします)。

- $x = y$ であれば、$gcd(x, y)$ は x そのもの。当然ですね。
- $x > y$ であれば、$gcd(x, y)$ は $gcd(x - y, y)$ に等しい。[2]
- したがって、$x - y$ を改めて x と置いて $gcd(x, y)$ を求めればよい。
- $x < y$ の場合も同様。
- 反復ごとに x または y の一方は減少するが、大きい方から小さい方を引くので負にはならない。
- ということは、この反復は有限回で止まる。
- ということは、そのとき $x = y$ が成り立ち、x が一番最初の x と y の最大公約数に等しい。

繰り返しを使うときは「必ず止まって、止まった時には求める状況が成り立っている」ように設計する、という感じがお分かりになりましたか?

4.1.3 演習 3 — 素数判定

素数の判定ですが、擬似コードは次のとおり。変数 `sosu` は先に「はい」を表す `true` を入れておきます。そして素数でないと思ったら「いいえ」を表す `false` 入れ、最後に結果がどちらか見ます。このような使い方の変数のことを**旗** (flag) と呼びます。

- isprime1: N が素数か否かを返す
- 　sosu ← 「真」。
- 　i を 2 から $N - 1$ まで変化させながら繰り返し、
- 　　もし N が i で割り切れるならば、sosu ←「偽」
- 　繰り返し終わり。

[2] 証明: 最大公約数を G と置くと、x も y も G の整数倍なので、$x - y$ も G の整数倍です。つまり、G は $x - y$ と y の公約数です。最大かどうかはまだ不明ですが、もし最大公約数で「なかった」なら、最大公約数 $H(> G)$ が別にあり、H は y の約数かつ $x - y$ の約数になります。ということは、H は $x - y + y = x$ の約数でもあります。これは G が x と y の最大公約数であるということに矛盾します。したがって G は $x - y$ と y の最大公約数でもあります。

- sosu を返す。

最初「旗」が立っていて、後で見たら「旗」が降りていたとすれば、誰が降ろしたかは分からなくても、誰かが旗を降ろしたことは確実に分かるわけです。では Ruby コードを見てみましょう。

```
def isprime1(n)
  sosu = true
  2.step(n-1) do |i|
    if n % i == 0 then sosu = false end
  end
  return sosu
end
```

4.1.4 演習 4 — 素数列挙とその改良

もっとも素朴な素数列挙プログラムは、上の素数判定を利用すれば簡単にできます。Ruby のコードを直接示しましょう。

```
def primes1(n)
  2.step(n-1) do |i|
    if isprime1(i) then puts(i) end
  end
end
```

これを手もとのマシンで動かしてみましたところ、10 秒間でおよそ 17,000 まで調べられました。これはあまり速くはないです。ところで、先の素数判定 `isprime` は「割り切れる」と分かってもそこでやめないで `n` の手前までずっと割っていきますから、早い段階で割り切れた数に対しては非常に無駄が大きいと思われます。そこで、改良版を作ってみました。

```
def isprime2(n)
  2.step(n-1) do |i|
    if n % i == 0 then return false end
  end
  return true
end
```

こちらは割り切れると分かったら直ちに `return` で「いいえ」を返すので、無駄な割り算をしなくてすみます。`primes1` をこちらを使うように直したら、10 秒間で 50,000 くらいまで調べられました。速度が 3 倍くらいになったわけです。さらに考えると、割り算は $N-1$ までやる必要はなく、$\sqrt{N}$ まで調べれば十分です ($\sqrt{N}$ よりも大きい因数があるなら、小さい因数もあるはずですから)。そこで素数判定を次のように改良します。

```
def isprime3(n)
  2.step(Math.sqrt(n)) do |i|
    if n % i == 0 then return false end
  end
  return true
end
```

これで試してみると、10 秒間で 800,000 くらいまで調べられました。次に、2 は素数であり、2 の倍数は素数でないことが分かり切っているので調べる必要がない、ということを利用しましょう。このため、3 以上の奇数だけで割ってみる「改編版」の素数判定を作って使います。

```
def isprime4(n)
  3.step(Math.sqrt(n), 2) do |i|
    if n % i == 0 then return false end
  end
  return true
end
def primes4(n)
  puts(2)
  3.step(n-1, 2) do |i|
    if isprime4(i) then puts(i) end
  end
end
```

2 は「別建てで」出力します (これでも仕様としては OK です)。こんどは 10 秒間で 1,000,000 くらいまで調べられました。もう少し頑張って、2 と 3 より大きい素数は 6 の倍数± 1 だけ (それ以外は 2 と 3 の倍数になる)、ということを利用して調べる数を減らしてみます。

```
def isprime5(n)
  6.step(Math.sqrt(n)+1, 6) do |i|
    if n % (i-1) == 0 then return false end
    if n % (i+1) == 0 then return false end
  end
  return true
end
def primes5(n)
  puts(2)
  puts(3)
  6.step(n-1, 6) do |i|
    if isprime5(i-1) then puts(i-1) end
    if isprime5(i+1) then puts(i+1) end
  end
end
```

こんどは 10 秒間で 1,400,000 くらいまで調べられました。コンピュータが高速だといっても、大量に計算する場合にはやはり工夫する価値はあるわけです。

演習 5 — 配列の演習

演習 5 も擬似コード略して Ruby のコードだけ記します。まず最大。

```
def arraymax(a)
  max = a[0]
  a.each do |x|
    if x > max then max = x end
  end
  return max
end
```

このような形で配列を使う場合は、「とりあえず max に最初の値を入れておき、より大きい値が出てきたら入れ換える」方法になります。each は配列の各要素を順に取り出して来るメソッドでした。

次は最大の値が何番目に出てくるかなので、普通の計数ループにします。また、「何番目か」も変数に記録し、最大を更新した時に同時に更新します。

```
def arraymaxno(a)
  max = a[0]
  pos = 0
  a.each_index do |i|
    if a[i] > max then max = a[i]; pos = i end
  end
  return pos
end
```

配列の各添字を列挙するには a.length.times を使えばよいのですが、ここに示したように配列のメソッド a.each_index を使うこともできます。

最大値の位置 1 箇所を記録するのは変数で OK ですが、最大が複数あった時に位置を全部打ち出すには、(1) 最大を求め、(2) 最大と等しいものの位置を打ち出す、という形で 2 回ループを使います。

```
def arraymaxno2(a)
  max = a[0]
  a.each_index do |i|
    if a[i] > max then max = a[i] end
  end
  a.each_index do |i|
    if a[i] == max then puts(i) end
  end
end
```

平均より小さい値を打ち出すのもこれと同様です。

```
def arrayavgsmaller(a)
  sum = 0.0
  a.each do |x| sum = sum + x end
  avg = sum / a.length
  a.each do |x|
    if x < avg then puts(x) end
  end
end
```

sum の初期値を 0.0 にしていますが、こうすれば sum の内容は常に実数なので、最後の割り算も実数の割り算になります。これが 0 だと、配列の中身がすべて整数のときは切捨て除算になり、平均の計算が正しくできません。

演習 6 — 配列を使った素数列挙

まず最初は、これまでに見つかった素数を配列に覚えておく方法です。[3]

```
def isprime8(a, n)
  limit = Math.sqrt(n).to_i + 1
  a.each do |i|
    if i >= limit then a.push(n); return true end
    if n % i == 0 then return false end
  end
  a.push(n); return true
```

[3]配列のメソッド push は配列の末尾に新たな値を追加します。

```
end
def primes8(n)
  a = []
  2.step(n-1) do |i|
    if isprime8(a, i) then puts(i) end
  end
end
```

素数判定メソッドは、素数の入った配列を受け取り、順に素数を取り出して候補の数を割り切るかどうか調べます。ただし、候補の数の平方根まで来たらもう割り切れないことが分かるので「素数である」という答えを返しますが、後に備えて配列にその素数を追加しておきます。この方法だと、「2 や 3 の倍数を除外」などのワザを使っていないのにもかかわらず、手元のマシンで 10 秒間で 2,000,000 くらいまでの素数が調べられましした。

ただし、このあたりをやっていると分かりますが (もっと早く気づいた人もいることでしょう)、実は数値を表示するという処理にもかなり手間が掛かっています。計測するという観点からは、表示を省略して内部のチェックだけの時間を計った方がいいでしょう。でも、速さの違いを実感していただくには、画面に出力が出た方が分かりやすいので、課題としては画面に出力するようにしてあります。

もう 1 つ (エラトステネスのふるい) はこれまでと大幅に違う方法です。

```
def primes9(n)
  a = Array.new(n, true);
  2.step(n-1) do |i|
    if a[i] then
      puts(i)
      i.step(n-1, i) do |k| a[k] = false end
    end
  end
end
```

最初に添字が $0 \sim N-1$ の配列 a を作り、各要素の値を true とします。次に、2 から始めて各候補の数値 i について、a[i] が true ならそれは素数なので打ち出し、その素数の倍数 k について a[k] を false にします。これで、調べて行ってまだ true の要素が素数として順に拾えます。

この方法は非常に高速で、10 秒間で 10,000,000 以上の素数がチェックできました。最初の素朴版が 10000 レベルだったので、1000 倍!!!も速くなりました。日常的には「1000 倍の差」はそうはありません。我々の歩く速度がおよそ 4km/h ですが、4000km/h はジェット機の速度です。これに対し、プログラムの動作速度では簡単に「ものすごい差」が生まれてしまうのです。

4.2 手続き/関数と抽象化

4.2.1 手続き/関数が持つ意味

手続きないし**サブルーチン**とは、ひとまとまりの動作に名前をつけ、他の箇所からの**呼び出し** (call) により実行できるようなものを言います。多くのプログラミング言語では、手続きから値を「返す」ことができ、このために手続きのことを**関数** (function) と呼ぶ言語もあります。既に学んできたように、Ruby では**メソッド**が手続きに相当します。そして既に使ってきたように、手続き呼び出し時に**パラメタ**を渡すことで、その渡した値に応じた動作や処理を行わせることができます。

たとえば前節では「整数 n が素数かどうかを調べる」というメソッドを作り、「素数を列挙する」メソッドからはそれを呼び出していました。そのコードを少し手直して再掲します。

```
def isprime(n)
  2.step(Math.sqrt(n)) do |i|
    if n % i == 0 then return false end
  end
  return true
end
def primes(n)
  2.step(n-1) do |i|
    if isprime(i) then puts(i) end
  end
end
```

2 つのメソッドに分けると、何がよいのでしょうか? それは、2 番目のメソッド中で「もし i が素数なら」とひとことで書けるようになることです。別の例として「n 未満の数で、2 つ違いの素数になっているものを打ち出す」作業を考えます。これも上のメソッドがあれば、次のように書けます。

```
def adjacentprimes(n)
  2.step(n-3) do |i|
   if isprime(i) && isprime(i+2) then puts "#{i} #{i+2}" end
  end
end
```

つまり「もし i が素数で、かつ、i+2 が素数 なら」と簡単に言えます。中では複雑な計算が必要な手順でも、まとめて名前をつけることで、必要なら何箇所からでも呼び出せ、コードも分かりやすくなるのです。言い替えれば、手続きによって**抽象化**が行えます。

抽象化とは、不要な細部を省いて問題の検討に必要なことがらだけを残すことです。たとえば、「n が素数かどうか」を調べる方法が一度分かれば、あとはそれを参照すればよいのであって、その中でどのように処理しているかは「不要な細部」として見ないで済むことが利点なのです。

4.2.2 手続き/関数と副作用 exam

関数という言葉は数学でも使われますが、数学で言う関数は「入力空間ないし定義域から出力空間ないし値域への写像」であって、同じ入力 (パラメタ) を与えた場合は同じ結果を返します。

たとえば $f(x) = x^2$ であれば、$f(2)$ の値は 4 であり、計算するたびに違うということはあり得ません。ですから、関数の値を 1 回計算して取っておき、2 回目は取っておいた値を利用するのでも、2 回とも計算するのとで、結果は一緒です。これに対し、プログラムにおける関数は「単なる計算手順」であり、計算のやり方によっては、毎回違う値を返したり、どこかに観測できる変化を残したりします。これを**副作用** (side effect) と呼びます。

一番簡単な例として、puts は呼び出すたびに画面に文字が出力されるので、1 回呼び出すのと 2 回呼び出すのでは結果が違います。つまり、**入出力** (input/output — キーボードや画面やファイルなどとの間でのデータのやりとり) は副作用だと言えます。また、関数や手続きの中で外部の (関数や手続きの外で定義された) 変数を書き換える場合も副作用です。

これまで使って来た変数は**局所変数** (local variable) と呼ばれ、そのメソッドが実行されている間だけ存在していて、実行が終わると消滅します (メソッドのパラメタも局所変数の一種と考えられます)。これに対し、プログラムの実行中ずっと存在し続け、さまざまなメソッド中から参照できる変数を**広域変数** (global variable) と呼びます。Ruby では先頭に$のついた名前の変数が広域変数です。広域変数は通常、複数のメソッド呼び出しをまたがって値を共有するのに使います。たとえば、次に示すようなやり方でいくつもの値を合計することを考えます。

```
irb> sum 1.5  ←次々に指定した値の
```

```
=> 1.5          ←合計が返される
irb> sum 2
=> 3.5
irb> sum 0.8
=> 4.3
irb> reset    ←ご破算もできる
=> 0
irb> sum 2
=> 2
irb> sum 0.7
=> 2.7
```

この実現のためにメソッド sum と reset を作りますが、両者の間で (および複数の sum 呼び出し間で) 値を保持するのに広域変数を使います。

```
$x = 0
def sum(v)
  $x = $x + v; return $x
end
def reset
  $x = 0
end
```

sum や reset は広域変数$x を変更するという副作用を持ちます。手続きが副作用を持つのは、広域変数に対する書き換えだけではありません。たとえば、配列をパラメタとして受け取り、その配列を書き換えた場合、変更は配列を渡した側にも影響します。これも副作用になります。

4.2.3 例題: RPN 電卓

上述の sum と reset は合計という簡単な計算だけでしたが、もっと込み入った計算もできる仕組みとして、**逆ポーランド記法** (RPN、Reverse Polish Notation) 電卓を作ってみます。私たちが普段書いている数式の書き方は**中置記法** (infix notation) と呼び、演算子が被演算子の間に書かれます。

$$8 + 5 \times 3 \rightarrow 23$$
$$(8 + 5) \times 3 \rightarrow 39$$

中置記法は「演算子の強さ (乗除算を優先)」「かっこの中を優先」などの規則を持ち、実は複雑です。これに対し、(1) 演算子は被演算子の後に書き、(2) 被演算子は演算子の直近にある「残っている値」とする、という規則を持つのが RPN です。上の 2 つの例を RPN で書くと次のようになります。[4]

$$8\ 5\ 3 \times +\ \rightarrow 8\ 15 +\ \rightarrow 23$$
$$8\ 5 + 3 \times\ \rightarrow 13\ 3 \times\ \rightarrow 39$$

上の例からも分かるように、RPN を使って式を記述する場合は、かっこが不要です。

RPN の計算は、図 4.1 のように値の並びを内部で保持し、数値が現れたら値を並びの末尾 (上が末尾です) につけ加え、演算子が出て来たら最後の 2 つを取って演算し、結果を末尾につけ加えるようにします。そして式の最後まで来た時に並びに残っている値が結果となります。[5]

さて、先の合計と同様、この RPN 電卓を Ruby で実現してみます。数値の入力は「e」というメソッドで実行し、演算は「add」「mul」をとりあえず用意しました。動かした様子を見ましょう。

[4]演算子を「後に」置くことから、**後置記法** (postfixnotation) とも呼びます。
[5]Mac の「電卓」は「Command-R」で RPN モードにできるので、少し計算してみると様子が分かります。

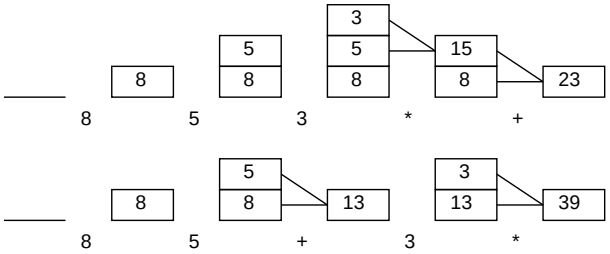


図 4.1: RPN 電卓による計算

```
irb> e 8
=> [8]
irb> e 5
=> [8, 5]
irb> e 3
=> [8, 5, 3]
irb> mul
=> [8, 15]
irb> add
=> [23]
irb> clear
=> []
irb> e 8
=> [8]
irb> e 5
=> [8, 5]
irb> add
=> [13]
irb> e 3
=> [13, 3]
irb> mul
=> [39]
```

プログラムですが、並びには配列を使用します。配列には末尾に値を追加するメソッド「a.push(値)」と末尾から結果を取り除いて返すメソッド「a.pop」が用意されているので好都合です。

```
$vals = []
def e(x)
  $vals.push(x); return $vals
end
def add
  x = $vals.pop; $vals.push($vals.pop + x); return $vals
end
def mul
  x = $vals.pop; $vals.push($vals.pop * x); return $vals
end
def clear
 $vals = []; return $vals
end
```

演習 1 「合計を求める」例題をそのまま打ち込んで動かしなさい。動いたら、さらに次の機能を実現するメソッドを追加しなさい。

a. 加える代わりに指定した値を引く機能 `dec(`x`)`。
b. うっかり間違って `reset` した時にそれを取り消せる機能 `undo`(undo の undo はできなくてもよいが、できるようにしてもよい)。
c. これまでに加えた (そして引いた) 値の一覧を表示した上で合計を表示する機能 `list`(reset はできた方がよい。reset の undo もできるとなおよい)。

演習 2 「RPN 電卓」の例題をそのまま打ち込んで動かしなさい。動いたら、さらに次の機能を実現するメソッドを追加しなさい。

a. 加算と乗算に加えて減算 (sub)、除算 (div)、剰余 (mod) を追加。
b. 現在の演算結果の符号を反転する操作 `inv`。たとえば「1 2 add inv → -3」となる。
c. 最後の結果と 1 つ前の結果を交換する操作 `exch`。たとえば「1 3 exch sub → 2」となる。
d. ご破産の機能 `clear` と、開始またはご破産から現在までの操作をすべて横に並べて (つまり RPN で) 表示する機能 `show`。[6]
e. その他、RPN 電卓にあったらよいと思う任意の機能。

演習 3 2 要素の配列を 2 つ並べた配列を 2 × 2 の行列として扱うことを考える。たとえば「`[[1.0, 0.0], [0.0, 1.0]]`」は単位行列であり、一般に「`[[`a`, `b`], [`c`, `d`]]`」は次の行列を表す。

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

a. 2 × 2 行列の RPN 電卓を作れ。加減算、乗算は作ること。
b. さらに、転置行列、逆行列の演算も作ってみよ。
c. 2 × 2 より大きな 3 × 3、できれば一般の $N \times N$ 行列の RPN 電卓を作ってみよ。

4.3 再帰呼び出し

4.3.1 再帰手続き・再帰関数の考え方 exam

関数や手続きの興味深い用法として、ある関数の中から直接または間接に自分自身を呼び出す、というものがあります。これを**再帰** (recursion) と呼びます。たとえば、前章でやった内容から、正の整数 x、y について、その最大公約数は次のように定義できます。

$$gcd(x,y) = \begin{cases} x & (x = y) \\ gcd(x-y,\ y) & (x > y) \\ gcd(x,\ y-x) & (x < y) \end{cases}$$

これにそのまま従って Ruby のメソッドを書くことができます。

```
def gcd(x, y)
  if x == y
    return x
  elsif x > y
    return gcd(x-y, y)
  else
    return gcd(x, y-x)
  end
end
```

[6] show を実現するためには、すべての演算にその操作内容を記録するコードを追加する必要がある。

プログラムそのものは大変分かりやすいですが、なぜ「堂々めぐり」にならずに計算が終わるのでしょうか。それは、図 4.2 を見れば分かります。

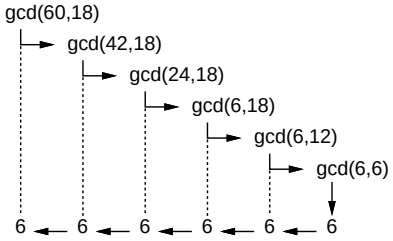


図 4.2: 再帰関数による最大公約数の計算

再帰関数 (再帰手続き) を作る時は、必ず次の原則に従います。

- 問題の「簡単な場合」は、すぐに答えを返す (上の例では $x = y$ の場合)。
- それ以外は問題を「少し簡単な問題に変形した上で」自分自身を呼び出す (上の例では、少し小さい数の最大公約数問題に変形)。

これがうまくできていれば、堂々めぐりにならずに正しく実行できるわけです。

演習 4 上の例題をそのまま打ち込んで動かせ。うまく動いたら、次のような再帰的定義に従った計算を再帰関数として書いて動かせ。また、典型的な実行の様子を表す、図 4.2 のような図を描いてみよ。

a. 階乗の計算。

$$fact(n) = \begin{cases} 1 & (n = 0) \\ n \times fact(n-1) & (otherwise) \end{cases}$$

b. フィボナッチ数。

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

c. 組み合わせの数の計算。

$$comb(n, r) = \begin{cases} 1 & (r = 0 \ or \ r = n) \\ comb(n-1,\ r) + comb(n-1,\ r-1) & (otherwise) \end{cases}$$

d. 正の整数 n の 2 進表現。[7]

$$binary(n) = \begin{cases} \text{“0”} & (n = 0) \\ \text{“1”} & (n = 1) \\ binary(n \div 2) + \text{“0”} & (n \text{ が } 2 \text{ 以上の偶数}) \\ binary(n \div 2) + \text{“1”} & (n \text{ が } 2 \text{ 以上の奇数}) \end{cases}$$

4.3.2　再帰呼び出しの興味深い特性

再帰呼び出しの興味深い特性として、「現在実行しているコードと、再帰的に呼び出した自分とは、動作は同一だが (同じプログラムだから当然!)、人格としては別人」だということがあります。たとえば ${}_nC_r$ の計算の様子を図 4.3 に示します。${}_5C_3$ を計算するとして、その「私」は「手下」として ${}_4C_2$

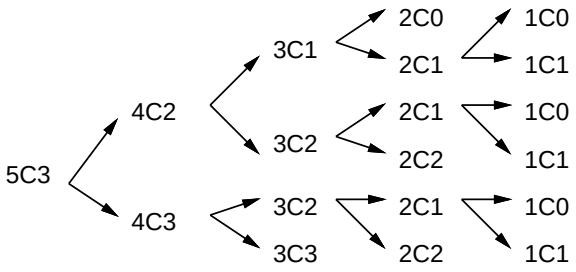


図 4.3: 再帰関数による組合せの数の計算

を計算する人と ${}_4C_3$ を計算する人に作業を依頼します。これらの「人」はデータ (n とか r) は「私」とは違っているので別人ですが、動作は「私」と同じわけです。

このような別人格を利用すると、興味深い処理が可能になります。たとえば、1〜3 を打ち出す場合、次のように 1 重ループを使えばできますね。

```
1.step(3) do |i| puts(i) end
```

では、「1〜3 が 2 つ並んだ全ての組合せ」だと…ループを 2 重にします (to_s は数値を文字列に変換するメソッドで、文字列どうしの＋は「連結」になります)。

```
1.step(3) do |i| 1.step(3) do |j| puts(i.to_s + j.to_s) end end
```

では「3 つ」だと 3 重…一般に n を指定して「1〜3 が n 並んだ全ての組合せ」が作れるでしょうか? プログラムでループを n 個書く方法では、プログラムを生成しない限り無理そうですよね? ところが、次のようにすればできるのです。

```
def nest3(n, s)  # 呼び方:nest3(3,"") ←空文字列渡す
  if n <= 0 then
    puts(s)
  else
    1.step(3) do |i| nest3(n-1, s + i.to_s) end
  end
end
```

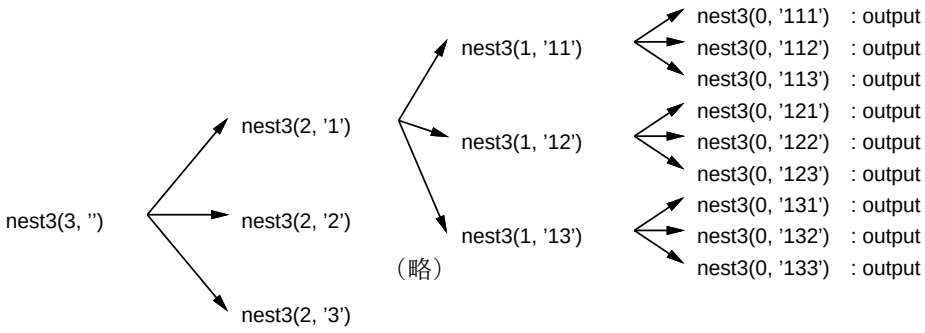


図 4.4: 1〜3 が n 個並んだすべての場合を出力

つまり、それぞれの「私」は自分の担当として 1〜3 を順番に生成し、「親の私」から渡された文字列にそれをくっつけ、「子の私」を呼び出します。入れ子になる (内側の) ループはその「子の私」の中で実行されるわけです。そして並べる数 n が 0 の場合は…「文字列を打ち出す」のが仕事になります。この呼び出しの様子を図 4.4 に示します。

[7] この場合、関数の返す値は文字列であることと、+ は文字列の連結演算、÷は整数の除算 (切捨て除算) を表していることに注意してください。Ruby では整数どうしの「/」は自動的に切捨て除算になるのでしたね。

演習 5 この例題では結果に同じ数字が何回も出て来ていたが、1 つの数字は 1 回しか使わないようにすると**順列** (permutaiton) を生成したことになるので、やってみよう。できれば、配列を渡すとその要素のすべての順列を次々に生成するのがよい。

ヒント: 渡された配列から 1 つ列に追加したら、その要素は配列に無いことにすればよい (たとえばそこに `nil` を入れるなどして)。子供の処理が終わったら元に戻すことを忘れないように。

演習 6 与えられた配列の全ての並び替えを生成できるということは、その中から昇順に並んだものを選ぶことで、元の配列を昇順に整列するアルゴリズムができることになる。実際にそのようなプログラムを作ってみよ。また、この方法の弱点を検討し、できれば改良する方法についても検討せよ。

演習 7 自分の名前のローマ字表記を与えると、そのアナグラム (さまざまな順で文字を入れ替えたもの) を表示するプログラムを作りなさい。ただしローマ字として成立しないものは表示しないように工夫すること。

演習 8 再帰を利用して自分の興味のあるプログラムを作りなさい。

本日の課題 4A

「演習 1」または「演習 4」で動かしたプログラム (どれか 1 つでよい) を含むレポートを提出しなさい。プログラムと、簡単な説明が含まれること。アンケートの回答もおこなうこと。

Q1. 手続き/関数/広域変数について学びましたが、納得しましたか。

Q2. 再帰的な呼び出しについてはどうですか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 4B

「演習 1」〜「演習 8」の (小) 課題から選択して 1 つ以上プログラムを作り、レポートを提出しなさい。プログラムと、課題に対する報告・考察 (やってみた結果・そこから分かったことの記述) が含まれること。アンケートの回答もおこなうこと。

Q1. 手続きが使いこなせるようになりましたか。

Q2. 再帰的な呼び出しについてはどうですか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

＃５　２次元配列＋レコード＋画像

今回の主な内容は次の通りです。

- ２次元配列・レコード型と画像の表現
- さまざまな形の描画

ここまでは数値の題材ばかりでしたが、コンピュータでは画像や音など任意のディジタル情報が扱えます。ここでは画像を通じて「自分が構想したものを作る」という経験を持って頂きます。

5.1　前回演習問題の解説

5.1.1　演習１ — 合計

一通り作りました (短いメソッドは１行に書いています)。引き算は簡単ですが、少しひねって負の数加えました。これまでの数値を覚えるためには$list という配列を用意し、数値をこの後ろに追加して覚えて行きます。reset の時は現在の値とこの$list を別の変数に退避しておき、undo では退避しておいたものを元に戻します。

```
$x = 0; $sx = 0; $list = []; $slist = [];
def sum(v) $x = $x + v; $list.push(v); return $x end
def dec(v) sum(-v) end
def list() p($list, $x) end
def reset() $sx = $x; $x = 0; $slist = $list; $list = [] end
def undo() $x,$sx = $sx,$x; $list,$slist = $slist,$list end
```

実行例を示します。

```
irb> sum 1
  => 1
irb> sum 2.5
  => 3.5       ←合計 3.5
irb> dec 1.2   ← 1.2 を引く
  => 2.3
irb> list      ←履歴表示
[1, 2.5, -1.2] ←履歴
2.3            ←現在値
  => nil
irb> reset     ←リセット
  => []
irb> sum 1
  => 1         ←また 0 からの和
irb> undo      ←リセットを戻す
  => [[1, 2.5, -1.2], [1]]
```

undo したときは過去の結果/リストと現在のものを交換するので、２回 undo すると元に戻ります。

5.1.2　演習 2 — RPN 電卓

これも一通り作りました。演算は add と同様で計算だけ変えます。交換は 2 つの値を取り出して逆に入れます。演算内容を覚えるために s というメソッドを用意し、この中で渡された値を文字列に変換して広域変数$str の後ろに連結して行きます。show はこの変数の内容を打ち出せばよいだけです (ついでに演算結果も打ち出します)。

```
$vals = []; $str = ''
def clear() $vals = []; $str = '' end
def s(x) $str = $str + ' ' + x.to_s end
def show() p($str, $vals[$vals.length-1]) end
def e(x) $vals.push(x); s(x); return $vals end
def add
  x = $vals.pop; $vals.push($vals.pop + x); s('+')
  return $vals
end
def sub
  x = $vals.pop; $vals.push($vals.pop - x); s('-')
  return $vals
end

def mul
  x = $vals.pop; $vals.push($vals.pop * x); s('*')
  return $vals
end
def div
  x = $vals.pop; $vals.push($vals.pop / x); s('/')
  return $vals
end
def exch
  x = $vals.pop; y = $vals.pop; s('x')
  $vals.push(x); $vals.push(y); return $vals
end
```

実行のようすを示します。

```
irb> e 1
  => [1]
irb> e 2
  => [1, 2]
irb> e 3
  => [1, 2, 3]  ←1、2、3を入れたところ
irb> mul        ←掛けたら6
  => [1, 6]
irb> add        ←足したら7
  => [7]
irb> show
" 1 2 3 * +"    ←履歴表示
7
```

```
  => nil
irb> e 4          ←さらに 7 を入れ
  => [7, 4]
irb> exch         ←交換
  => [4, 7]
irb> sub          ←引き算
  => [-3]
```

作ってみると、スタックを使った計算のようすがよく分かると思います。

5.1.3 演習 3 — 行列電卓

2 × 2 行列の電卓ですが、加減算は要素ごとに演算すればよいので簡単ですね。乗算とか逆行列とかはちょっとごちゃごちゃしますが、まあこれらも 2 × 2 であればひたすら書けばできるでしょう。

```
$vals = []
def e(m) $vals.push(m) end
def add
  m = $vals.pop; n = $vals.pop
  $vals.push([[n[0][0]+m[0][0], n[0][1]+m[0][1]],
              [n[1][0]+m[1][0], n[1][1]+m[1][1]]])
  return $vals
end
def sub
  m = $vals.pop; n = $vals.pop
  $vals.push([[n[0][0]-m[0][0], n[0][1]-m[0][1]],
              [n[1][0]-m[1][0], n[1][1]-m[1][1]]])
  return $vals
end
def mul
  m = $vals.pop; n = $vals.pop
  $vals.push([[n[0][0]*m[0][0] + n[0][1]*m[1][0],
               n[0][0]*m[0][1] + n[0][1]*m[1][1]],
              [n[1][0]*m[0][0] + n[1][1]*m[1][0],
               n[1][0]*m[0][1] + n[1][1]*m[1][1]]])
  return $vals
end
def trans
  m = $vals.pop;
  $vals.push([[m[0][0], m[1][0]],
              [m[0][1], m[1][1]]])
  return $vals
end
def inv
  m = $vals.pop
  d = (m[0][0]*m[1][1] - m[0][1]*m[1][0]).to_f
  $vals.push([[m[1][1]/d, -m[0][1]/d],
              [-m[1][0]/d, m[0][0]/d]])
  return $vals
end
```

3 × 3 以上になると、直接書くのではなくループを使った方が楽になりますが、そのあたりはこの科目では含みません (いずれどこかで習うと思いますが)。実行例をみてみましょう。

```
irb> e [[1, 2], [3, 4]]
  => [[[1, 2], [3, 4]]]
irb> e [[1, 1], [1, 1]]
  => [[[1, 2], [3, 4]], [[1, 1], [1, 1]]]
irb> sub
  => [[[0, 1], [2, 3]]]
```

引き算とかは問題ないですね。逆行列はどうでしょうか。

```
irb> e [[2, 1], [1, -1]]
  => [[[2, 1], [1, -1]]]
irb> inv
  => [[[0.3333333, 0.3333333], [0.3333333, -0.6666667]]]
irb> e [[1, 0], [5, 0]]
  => [[[0.3333333, 0.3333333], [0.3333333, -0.6666667]], [[1, 0], [5, 0]]]
irb> mul
  => [[[2.0, 0.0], [-3.0, 0.0]]]
```

これは何を計算しているかというと、次の連立方程式を解いています。

$$\begin{cases} 2x & + & y & = & 1 \\ x & - & y & = & 5 \end{cases}$$

これを行列の形に書くと次のようになります。

$$\begin{pmatrix} 2 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} x & 0 \\ y & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 5 & 0 \end{pmatrix}$$

係数行列 $A =$`[[2, 1], [1, -1]]` の逆行列を A^{-1} を上式両辺に左から掛けます。

$$A^{-1}\, A \begin{pmatrix} x & 0 \\ y & 0 \end{pmatrix} = A^{-1} \begin{pmatrix} 1 & 0 \\ 5 & 0 \end{pmatrix}$$

$A^{-1}A$ は単位行列なので消えますから、つまり右辺を計算すると x と y が求まるわけです。実際、$x = 2$、$y = -3$ を代入してみると元の連立方程式が成り立っていることが確認できます。

5.1.4　演習 4 — 再帰関数

これらは定義のとおり再帰関数にすればできるので、まずはコードを示します。

```
def fact(n)
  if n == 0 then return 1
  else           return n * fact(n-1)
  end
end
def fib(n)
  if n < 2 then return 1
  else          return fib(n-1) + fib(n-2)
  end
end
```

```
def comb(n, r)
  if r == 0 || r == n then return 1
  else                     return comb(n-1, r) + comb(n-1, r-1)
  end
end
def binary(n)
  if n == 0 then        return "0"
  elsif n == 1 then     return "1"
  elsif n % 2 == 0 then return binary(n / 2) + "0"
  else                  return binary((n-1) / 2) + "1"
  end
end
```

実行例も一応示しておきます。

```
irb> fact 4
  => 24
irb> fact 5
  => 120
irb> fib 2
  => 2
irb> fib 3
  => 3
irb> fib 4
  => 5
irb> comb 5, 2
  => 10
irb> comb 6, 2
  => 15
irb> binary 5
  => "101"
irb> binary 7
  => "111"
irb> binary 8
  => "1000"
```

5.1.5 演習 5 — 順列

問題のヒントに書いたように、配列から 1 つずつ要素を取って出力用の列に入れますが、その際取った要素の位置に nil を入れることで重複して取らないようにします。呼び出し方を覚えなくて済むように perm には配列だけ渡し、そこから再帰用メソッド perm1 を「残った長さ、元の配列、空の配列」をパラメタとして呼びます。

```
def perm(a) perm1(a.length, a, []) end
def perm1(n, a, b)
  if n == 0 then p(b); return end
  a.each_index do |i|
    if a[i] == nil then next end
    b.push(a[i]); a[i] = nil; perm1(n-1, a, b); a[i] = b.pop
```

```
    end
  end
```

perm1では、残った長さが0なら出力して終わります。そうでない場合はaの各要素を順番に見ますが、その際入っていたのがnilなら「ループの次に進み」ます(nextの機能)。そうでない場合は、配列bにその要素を追加し、配列aのその位置にnilを入れて自分自身を呼びます(もちろんnは1減らす)。戻ってきたらbの最後を取り除いてそれをa[i]に戻します。このように、それぞれが自分が変更したものを元に戻すことで、全体としてうまく動きます。実行例は次の通り。

```
irb> perm [1, 2, 3]
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
[3, 2, 1]
 => [1, 2, 3]
irb>
```

5.2　2次元配列と画像の表現

5.2.1　2次元配列の生成 exam

2次元配列(配列の配列)を用いた前回の演習問題では、直接すべての値を「[[a, b], [c, d]]」のように指定することで2次元配列を生成していました。しかしこの方法は大きさが大きくなるととても大変です。1次元(1列)の配列を作る時に、ブロックを使って初期値を設定する方法がありました。

```
a = Array.new(100) do |i| 2*i end
```

これを応用することで大きな2次元配列を作ることができます。つまり、ブロックの中にさらにArray.new(...)を入れれば、「配列が並んだ配列」つまり2次元配列ができるからです。

```
a = Array.new(10) do Array.new(10, 1) end
  # 10 × 10 ですべて「1」の行列
a = Array.new(10) do |i| Array.new(10) do |j| i*j end end
  # 「九九の表」
```

「2次元配列」は実際には図5.1のように配列の各要素が配列という構造です。でも普段は「縦横2次元に要素が並んでいる」とイメージして問題ありません。

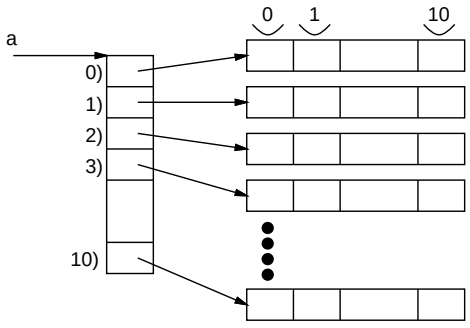


図 5.1: 2次元配列

5.2.2 レコード型の利用 exam

配列が「同じ型 (種類) の値の並びで、添字 (番号) により要素を指定する」のに対し、**レコード**は「様々な型 (種類) の複数の値が並んだもので、どの値 (フィールド) かは名前で指定する」ものです。Ruby ではレコード型はまず `Struct.new` によりレコードクラスを定義し、その後レコードクラスを使って個々のレコード (データ) を作ります。具体的には、レコードクラスの定義は次のようにします (レコード名は大文字で始まる必要があります)。

```
レコード名 = Struct.new(:名前, :名前, ...)
```

ここで「:名前」は**記号型** (symbol type) の定数で、これによりフィールドの名前が指定できます。個々のレコードを作るのは次によります。

```
p = レコード名.new(値, 値, ...)
```

これによりレコード型の値が作られ、指定した値が各フィールドの初期値になります (順番はレコード定義の時に指定した順になります)。上の例ではそのレコードを変数 `p` に入れています。

コンピュータ上では画像は**ピクセル** (pixel、画面上の小さな点) の集まりとして扱い、各ピクセルの色は赤 (R)、緑 (G)、青 (B) の強さを 0〜255 の範囲の整数で表すことが普通です。ピクセルの情報をレコードとして定義し、それを用いてピクセルを生成します。

```
Pixel = Struct.new(:r, :g, :b)
p = Pixel.new(255, 255, 255)  # RGB とも 255 の値
```

Ruby では配列と同様、レコードも new を使って作り出す必要があります。作り出したあとは、「`p.r`」「`p.g`」「`p.b`」等、変数名の後にフィールド名を加えたものが通常の変数と同様に使えます。配列と似ていますが、レコードの場合はフィールドは「名前」である点が違います。

5.2.3 ピクセルの 2 次元配列による画像の表現 exam

さて、ピクセル 1 個の説明が終わったので、今度はこれを「2 次元に (縦横に) 並べて」画像を作ることを考えます (図 5.2 左)。これを Ruby で表現する場合、各 Pixel を上で説明したように Ruby のレコード型で表現し、それを縦横に並べるわけです。たとえば縦方向 (高さ) が 200 ピクセル、横方向 (幅) が 300 ピクセルの画像を作るとします。そのためには、先に学んだ 2 次元配列の初期化のとき、個々の要素をピクセルにすればよいのです。

```
$img = Array.new(200) do
  Array.new(300) do Pixel.new(255,255,255) end
end
```

これによって作られるデータ構造は図 5.2 右のようになります。

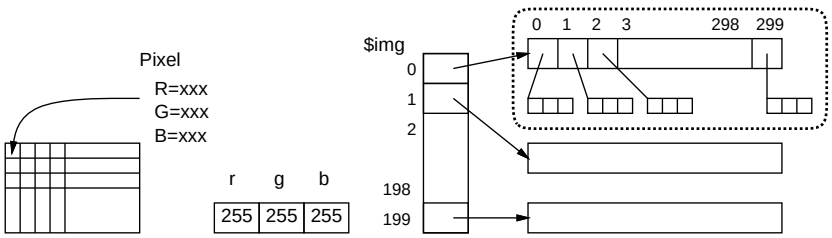


図 5.2: 画像のデータ構造とレコードの 2 次元配列

5.2.4　画像中の点の設定と書き出し

先に生成した画像はRGB値が全て「255」なので「真っ白」です。そこで次に、座標(x, y)のピクセルを指定したRGB値に書き換えるメソッドを作ります。

```
def pset(x, y, r, g, b)
  if 0 <= x && x < 300 && 0 <= y && y < 200
    $img[y][x].r = r; $img[y][x].g = g; $img[y][x].b = b
  end
end
```

なぜif文があるかというと、X座標とY座標が画像の範囲内 (ここでは0〜299、0〜199) のときだけ書き込むためです。こうしておくと、呼び出す側で間違って (または簡単のため) 画像の範囲外に書き込もうとしても単に無視できます。

さて次に、こうして画像中に書き込むことができるようになりましたが、画像を実際に「見る」ためには何らかのファイル形式で書き出す必要があります。ここではできるだけ簡単な形式として**PPM形式**を選び、その形式でファイルに書き出すメソッド`writeimage`を作りました。[1]

```
def writeimage(name)
  open(name, "wb") do |f|
    f.puts("P6\n300 200\n255")
    $img.each do |a|
      a.each do |p| f.write(p.to_a.pack("ccc")) end
    end
  end
end
```

メソッドの説明は次の通りです。

- `writeimage`は画像のファイル名 (文字列) を指定して呼び出す。
- `open`は指定した名前のファイルをバイナリ (binary) 形式で書き出す (write) 準備をして、その出力チャネル (データの通り道) をブロックに渡して呼び出す。
- ここではブロックでチャネルを`f`という名前で受け取る。
- まず、ファイルに「P6 300 200 255」と出力する。これは「PPM画像のカラー形式で、幅300×高さ200、RGB値の最大は255」を表す指定になっている。[2]
- 続いて、画像の2次元配列の各行について、さらにその行の中の各ピクセルについて、(1) ピクセルを配列に変換し (`p.to_a`)、その配列を3バイトのバイナリデータに変換し (`.pack("ccc")`)、ファイルに書き込む (`f.write(…)`)。

5.2.5　例題: 画像を生成し書き出す

ではいよいよ、画像を作って書き出すメインのメソッドまで含めた全体像を見て頂きましょう。

```
Pixel = Struct.new(:r, :g, :b)
$img = Array.new(200) do
```

[1] 普段私たちがWebなどで見ている画像形式はGIF、JPEG、PNGなどで、ブラウザもこれらの画像を表示するようにできていますが、これらのファイル形式は圧縮などの機能が備わっているため、そんなに簡単なコードで書き出すことができないのです。

[2] 画像ファイルの先頭にはだいたい、このような形で画像の種別やサイズを記述したデータが置かれています。この部分のことを**ヘッダ** (header) などと呼びます。

```
    Array.new(300) do Pixel.new(255,255,255) end
  end
  def pset(x, y, r, g, b)
    if 0 <= x && x < 300 && 0 <= y && y < 200
      $img[y][x].r = r; $img[y][x].g = g; $img[y][x].b = b
    end
  end
  def writeimage(name)
    open(name, "wb") do |f|
      f.puts("P6\n300 200\n255")
      $img.each do |a|
        a.each do |p| f.write(p.to_a.pack("ccc")) end
      end
    end
  end
  def mypicture
    pset(100, 80, 255, 0, 0)
    writeimage("t.ppm")
  end
```

mypicture がメインになりますが、ここでは $(100, 80)$ の位置に真っ赤な点 (RGB のうち R だけが最大なので) を打ち、t.ppm というファイルに書き出します。実際にこれを動かすには、ターミナルの窓を「2 つ」開いて次のようにします。

- 片方の窓ではこれまで通り irb を動かし、mypicture を実行させる。
- もう片方の窓では、できあがった t.ppm を gimp など表示できるツールを使って表示する (変換ツールで PNG など普通に見られる画像形式にしてもよいです)。

図 5.3: 赤い点が 1 個

生成された画像を図 5.3 にお見せします (赤い点が小さすぎてほとんど分からないと思いますが…)。ではもうちょっとそれらしい図形はどうすればいいでしょうか? 点 $(100, 80)$ が 1 つの点だとすれば、点 $(80, 80), (81, 80), (82, 80) \cdots (120, 80) = \{(x, 80) | 80 \leq x \leq 120\}$ のようなものが線分になりますね。それはどうやって作ればいいかというと、ループを使えばいいわけです。

```
  def mypicture2
    80.step(120) do |x| pset(x, 80, 0, 0, 255) end
    writeimage("t.ppm")
  end
```

「M.step(N) do |x| ... end」は整数 M から始めて N まで 1 ずつ値を変化させながらブロックのパラメタ (ここでは x) に渡してブロックを繰り返してくれます。色は同じではつまらないので、今度は「真っ青」にしてみました。できた絵を図 5.4 に示します。

図 5.4: 青い線分

演習 1 上の例題の好きなものを打ち込み、そのまま動かしなさい (色の RGB 値は 0〜255 の範囲で適宜変えてみるとよいでしょう)。動いたら、次のように変更してみなさい。

a. 水平はもうわかったので、垂直または斜め (右上がり) に線を引くようにしてみる。
b. 線が細いと弱っちいので、幅 3 の線を引くようにしてみる (ヒント: $(x, 79), (x, 80), (x, 81)$ に点を打つなどする)。
c. 線を複数使って、長方形とか正方形を描く。
d. 長方形とか正方形の中を塗りつぶす (ヒント: x 方向だけに繰り返したら水平線だが、それをさらに y 方向に繰り返すと四角い領域の中全部が塗れる)。
e. 三角形を描いてみる (塗りつぶすのは多少工夫が必要かと)。
f. その他、好きな図形や模様や色を表現してみる。

上記の演習では、mypicture の中にコードを追加して pset を呼び出すことを想定していますが、場合によってはさらにメソッドを追加する方が作りやすいかも知れません。また、先に示した pset は「Y 座標が大きいほど下」に点を打ちます。コンピュータ上の画像は伝統的にそうしていますが、皆様は「Y 軸が上向き」に慣れているので、pset をそのように直すことを勧めます。

5.2.6　計算により図形を塗りつぶす exam

先の演習はどうでしたか。グラフのように「x を変えながら y=f(x) を計算して pset(x, y, ...)」と考える人が多いと思いますが、実はこの方法で輪郭線を描くと細かったり途切れたりしてあんまりよくありません。[3] むしろ図 5.5 のように「塗りつぶす」方がきれいにできやすいです。

このような塗りつぶしをおこなうメソッド fillcircle を見てみましょう。

```
def fillcircle(x0, y0, rad, r, g, b)
  200.times do |y|
    300.times do |x|
      if (x-x0)**2 + (y-y0)**2 <= rad**2 then pset(x, y, r, g, b) end
    end
  end
end
```

[3] 途切れないように工夫しても、「1 ピクセル幅」というのは今日の画面解像度では「極めて細い」線になりますから。

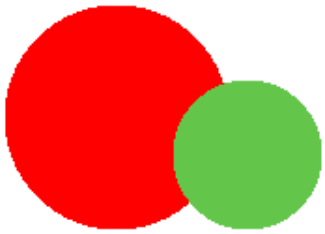


図 5.5: 2 つの内部まで色を塗った円

このメソッド内では、times の中にさらに times、つまりループの中にさらにループがあるので、このようなものを **2 重ループ**と呼びます。この内側での 2 つの変数の進み方は、次のようになります。

```
0,0  0,1  0,2,  0,3  0,4  ....  0,288 0,299
1,0  1,1  1,2,  1,3  1,4  ....  1,288 1,299
2,0  2,1  2,2,  2,3  2,4  ....  2,288 2,299
...
...
198,0  198,1  198,2,  198,3  198,4  ....  198,288 198,299
199,0  199,1  199,2,  199,3  199,4  ....  199,288 199,299
```

縦横に揃えて書いてありますが、要は外側ループで y の値を 0〜199 まで変化させ、**そのそれぞれの値について**内側の x の値を 0〜299 まで変化させます。そして、これで画像上のすべての点 (座標) を洩れなく列挙しているわけです。つまり、ここで列挙される (x, y) の集合は次のようになるわけです (これが画像の全範囲)。

$$\{\ (x, y)\ |\ 0 \leq x < 300,\ 0 \leq y < 200\ \}$$

円というのは中心 (x_0, y_0) からの距離が rad 以内の点の集合ですから、次のように表せます。

$$\{\ (x, y)\ |\ |(x, y) - (x_0, y_0)| \leq rad\ \}$$

これをプログラムで扱いやすいように、距離の 2 乗を使う形に直します。

$$\{\ (x, y)\ |\ (x - x_0)^2 + (y - y_0)^2\ \leq\ rad^2\ \}$$

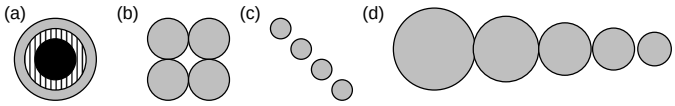
さて、先のコードでは 2 重ループの内側に if 文がありますが、その条件がまさにこの条件であり、従って「円に含まれるすべての点 (x, y) に対して色を設定する (塗る)」ことになるわけです。

実際にはこれを呼び出す必要があるので、円を 2 つ描き、ファイルに画像を書き出すメソッドを mypicture1 という名前で用意しました (その結果が図 5.5 なのでした)。

```
def mypicture1
  fillcircle(110, 100, 60, 255, 0, 0)
  fillcircle(180, 120, 40, 100, 200, 80)
  writeimage("t.ppm")
end
```

長方形などさまざまな図形についても、このように「すべての点のうち、図形内部に含まれるという条件を満たす点のみに色を設定する」という形でコードを作ることができます

演習 2 円を塗るメソッドを先のプログラムに追加して動かせ。動いたら、`fillcircle`の呼び出し方を調節して、次の図のように円を配置してみよ。



演習 3 次のような手続きを追加して円以外の図形を塗ってみよ。

a. ドーナツ型を塗るメソッド。[4]
b. 長方形または楕円を塗るメソッド。
c. 三角形を塗るメソッド。
d. その他、自分の好きな形を塗るメソッド。

演習 4 どの図形でもよいが、色を塗る際に、単色で単純に塗るのでなく、次のような塗り方ができるようにしてみよ。

a. 2 色を指定して、ストライプ、ボーダー、チェックなどで塗れるようにする。
b. 色を塗る際に、「重ね塗り」できるようにしする。つまり**透明度** (transparency)$0 \leq t < 1$ を指定し、各 R/G/B 値について単に新しい値で上書きする代わりに $t \times c_{old} + (1-t) \times c_{new}$ のように混ぜ合わせた値にする。
c. 徐々に色調が変わっていくようにする。(注意: RGB 値は 0〜255 の「整数」でなければならない! 実数で計算した場合はその値が x の場合、「`x.to_i`」で**小数部分を切り捨てて**整数にできる。)
d. ぼやけた形、ふわっとした形などを表現してみよ。
e. その他、美しい絵を描くのにあるとよい機能を工夫してみよ。

演習 5 何か好きな絵を生成してみなさい。

本日の課題 5A

「演習 1」または「演習 2」で動かしたプログラム (どれか 1 つでよい) を含むレポートを提出しなさい。プログラムと、簡単な説明が含まれること。アンケートの回答もおこなうこと。

Q1. 画像のデータ構造について学びましたが、納得しましたか。

Q2. どのような画像を生成してみたいと考えていますか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 5B

「演習 1」〜「演習 5」の (小) 課題から選択して 1 つ以上プログラムを作り、レポートを提出しなさい。プログラムと、課題に対する報告・考察 (やってみた結果・そこから分かったことの記述) が含まれること。アンケートの回答もおこなうこと。

Q1. 簡単なものなら自分が思った画像が作れますか。

Q2. うまく画像を作り出すコツは何だと思いますか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

[4]ヒント: ドーナツ上の点であることは次のような条件で表せる: $\{\ (x,y) \mid r_0 \leq |(x,y)-(x_0,y_0)| \leq r_1\ \}$

＃6　画像の生成 (総合実習)

今回は「総合実習」であり、「美しいと考える画像」をプログラムで生成して頂きます。したがって課題は「B 課題」のみです。ペアも OK ですが、ただし**ペアであっても今回に限り「全く同じ絵ではないようにする」こと**。同一図柄の色違いは認めますので、それほど厳しい条件ではないと思います。今回の目標は次のことがらです。

- 自分が生成したいと思う画像のために何が必要かを考える
- 画像の内容に合わせてプログラム構造を実現していく

以下にはまず画像生成で役立つかも知れない補足説明があり、続いて前回演習問題の解説をします。

6.1　画像の生成に関連する補足

6.1.1　三角形や凸多角形を塗るには

前回は円の中を塗りつぶすことだけやりましたが、「図形内という条件を記述する」方法についてもう少し具体例を示しましょう。たとえば、XY 軸と平行な長方形だったら、その XY 座標の小さい側の角を (x_0, y_0)、大きい側の角を (x_1, y_1) とした場合、条件は次のように表せますから簡単ですね。

$$\{\ (x, y) \mid x_0 \leq x \leq x_1 \wedge y_0 \leq y \leq y_1\ \}$$

しかし、XY 軸に対して傾けたい場合は、もっと一般的に考える必要があります。例として三角形を取り上げましょう。三角形は 3 つの辺で囲まれた領域ですよね (当り前)。1 つの直線は、平面を 2 つの半平面に分けます。直線だと指定しにくいので、直線に含まれる線分を $(x_0, y_0) - (x_1, y_1)$ で指定することにして、この半平面の点の集合は次の式で表されます。

$$\{\ (x, y) \mid (x_1 - x_0)(y - y_0) - (y_1 - y_0)(x - x_0) \geq 0\ \}$$

なぜそうなるかというと、上の条件式は、ベクトル $\overrightarrow{(x_0, y_0) - (x_1, y_1)}$ と $\overrightarrow{(x_0, y_0) - (x, y)}$ の外積が正であるという条件であり、一般に起点を共有する位置ベクトル $\vec{\mathrm{a}}$ と $\vec{\mathrm{b}}$ の外積 $\vec{\mathrm{a}} \times \vec{\mathrm{b}}$ の符号は $\vec{\mathrm{a}}$ から見て $\vec{\mathrm{b}}$ が左にある場合には正、右にある場合には負になるからです。

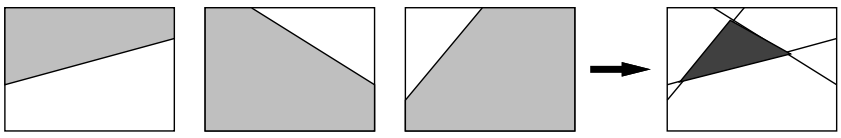


図 6.1: 三角形は 3 つの半平面の共通集合

そして、半平面が定義できたら、3 つの半平面に「ともに」含まれる点 (つまり共通集合) が三角形となります (図 6.1)。ということは、条件で言えば上に記したような半平面の条件を 3 つ「すべて」満たす点、ということになります。

三角形に限らず、凸な (へこみの無い) 多角形についても同様の考え方で定義することができます。そして傾いた長方形 (太さのある線分は細長い長方形だと考えることができます) も、こちらの方法で定義することができます。

6.1.2　おまけ 2: 楕円を塗るには

図形によっては、上述のように直接 2 次元座標上での条件を記述するのは厄介なものがあります。たとえば楕円を考えてみましょう。皆様は楕円の一般式を言えますか?

しかし、よい方法があります。楕円は円を「引き延ばして作る」ことができますね。ですから、原点を中心とした、たとえば横の半径が 3、縦の半径が 2 の楕円があったとして、それを「横に $\frac{1}{3}$ 倍、縦に $\frac{1}{2}$ 倍に縮めると」半径 1 の円になるはずです。そして、半径 1 の円内にあるかどうかは簡単に判定できます。つまり、$p(x, y)$ を $p'(\frac{x}{3}, \frac{y}{2})$ に写像してから円内の判定をすればよいわけです (図 6.2)。原点にない楕円や軸の回転した楕円は? これらも、原点の移動や座標の回転をしてから判断すればいいわけですね。[1]

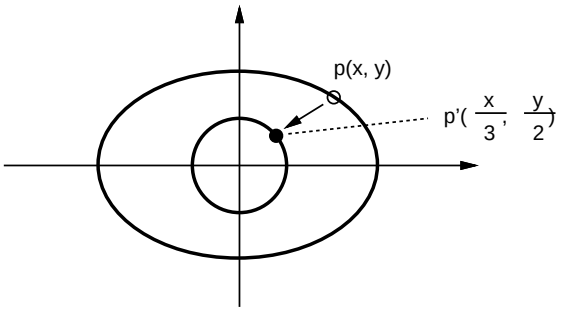


図 6.2: 楕円の内部かどうかの判定

6.1.3　おまけ 3: フラクタル

フラクタルな図形というのは、再帰性のある図形 (その図形の一部分が、全体と相似になっているような図形) を言います。たとえば、図 6.3 を見ると、「正方形の 4 隅に小さい正方形がくっついている」という構造が繰り返されています。

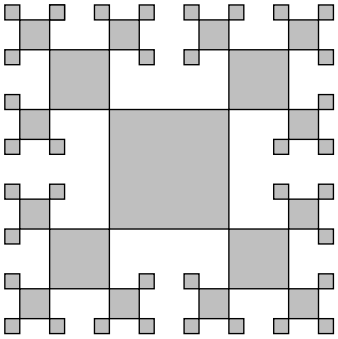


図 6.3: フラクタルな図形の例

こういうのは、再帰的なメソッドで作れます。どのみち、図形が小さくなりすぎたら描けないので、それが終了条件となります。たとえば次のような擬似コードを考えてみてください (正方形を描くメソッド fillsquare は別にあるものとします。)

- squares: 中心 x, y, 1 辺 2 × len の正方形フラクタルを描く
- 　　もし `len < 1` ならば 戻る。
- 　　fillsquare(x, y, len)。

[1] こういう変換のことを総称してアフィン変換とか呼びますが、まあ用語はそれとして、適切な行列を作って掛けるとかでできます。

- half ← len / 2。
- squares(x+len+half, y+len+half, half)。
- squares(x+len+half, y-len-half, half)。
- squares(x-len-half, y+len+half, half)。
- squares(x-len-half, y-len-half, half)。

なお、このようにきっちり機械的にやると人工物ぽくなりますが、乱数を使って「大きさがランダムに変動したり」「子供が確率的にできたりできなかったり」すると、自然物ぽくなります (自然界はフラクタルだと言われている)。Ruby では乱数は次の 2 つの方法で使えます。

- `rand()` — 0 以上 1 未満の実数値の一様乱数が得られる。
- `rand(`N`)` — N は整数として、0 から $N-1$ までの整数の一様乱数が得られる。

6.2 前回演習問題の解説

6.2.1 演習 1 — 線分と塗りつぶし

演習 1 のものはだいたい簡単なのでプログラムだけ示します。

```
def verticalline
  80.step(120) do |y| pset(100, y, 255, 0, 0) end
  writeimage("t.ppm")
end

def slantline
  0.step(50) do |i| pset(100+i, 150-i, 255, 0, 0) end
  writeimage("t.ppm")
end

def thickline
  80.step(120) do |x|
    pset(x,80,255,0,0); pset(x,79,255,0,0); pset(x,81,255,0,0)
  end
  writeimage("t.ppm")
end

def square
  80.step(120) do |v|
    pset(v, 80, 255, 0, 0); pset(v, 120, 255, 0, 0)
    pset(80, v, 255, 0, 0); pset(120, v, 255, 0, 0)
  end
  writeimage("t.ppm")
end

def fillsquare
  80.step(120) do |y|
    80.step(120) do |x| pset(x, y, 255, 0, 0) end
  end
  writeimage("t.ppm")
end
```

6.2.2 演習 2 — 円を配置する

これは本当に配置すればいいだけなので、fillcircle を呼ぶところだけを示します。

```
fillcircle(100, 100, 80, 255, 0, 0)
fillcircle(100, 100, 60, 0, 255, 0)
fillcircle(100, 100, 40, 0, 0, 255)

fillcircle(100,100,50,255,0,0); fillcircle(200,100,50,255,0,0)
fillcircle(100,200,50,255,0,0); fillcircle(200,200,50,255,0,0)

fillcircle(100,100,25,255,0,0); fillcircle(150,150,25,255,0,0);
fillcircle(200,200,25,255,0,0); fillcircle(250,250,25,255,0,0);

fillcircle(100,100,40,255,0,0); fillcircle(170,100,30,255,0,0);
fillcircle(220,100,20,255,0,0); fillcircle(250,100,10,255,0,0);
```

6.2.3 演習 3〜4 — 様々な図形

細かい演習問題は省略して、今回は図 6.4 のようなさまざまな図形を描くプログラムを説明します (このために、透明度の機能も追加しました)。

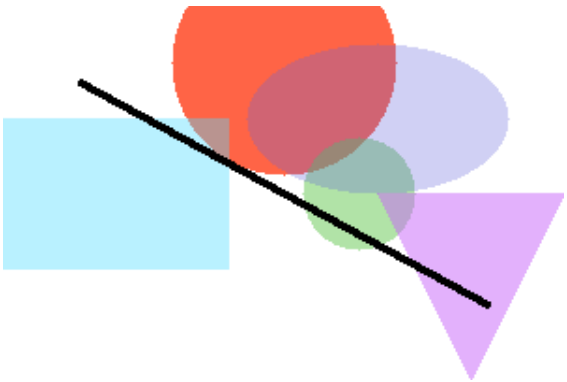


図 6.4: 生成されたさまざまな図形

レコード定義、画像の書き出しについては変更はありません。色に「透明度」をつけて塗れるようにするには、現在の色と塗りたい色を α (透明度) に応じて比例配分すればよいでしょう。それを行うように pset を変更し、あとはすべてこれを使っています。

```
def pset(x, y, r, g, b, a = 0.0)
  if x < 0 || x >= 300 || y < 0 || y >= 200 then return end
  $img[y][x].r = ($img[y][x].r * a + r * (1.0 - a)).to_i
  $img[y][x].g = ($img[y][x].g * a + g * (1.0 - a)).to_i
  $img[y][x].b = ($img[y][x].b * a + b * (1.0 - a)).to_i
end
```

ところでパラメタの指定で「a = 0.0」とは? これはデフォルト引数といい、呼ぶときにそのパラメタを指定しなければ 0.0 つまり「不透明、べた塗り」として扱うので、これまで通りに使えるということです。以下の図形もすべて、そのようにしました。

次に、円を描く (正確には円の形に色を塗る) メソッド fillcircle を示します。ただし、前回は「画像全部の点」に対して判定していましたが、今回は処理を軽くするため、必要にしてできるだ

け少ない範囲の点だけを列挙して判定します。それには、円に含まれ得る点の X 座標、Y 座標の範囲 (中心 (x_c, y_c) 半径 r として $x_c \pm r$ と $y_c \pm r$) をまず考え、その範囲内の各点 (x, y) について $(x - x_c)^2 + (y - y_c)^2 \leq r^2$ を満たすなら円内にあるものとしてその点の色を設定します:

```
def fillcircle(x, y, rad, r, g, b, a = 0.0)
  j0 = (y-rad).to_i; j1 = (y+rad).to_i
  i0 = (x-rad).to_i; i1 = (x+rad).to_i
  j0.step(j1) do |j|
    i0.step(i1) do |i|
      if (i-x)**2+(j-y)**2<rad**2 then pset(i,j,r,g,b,a) end
    end
  end
end
```

XY 座標や半径に小数点つきの値が入れられても動作するように、調べる範囲を計算した時に結果をメソッド to_i で整数にしています。

長方形を描く fillrect は、円よりもっと簡単で、単にその範囲全部を pset するだけです。[2]

```
def fillrect(x, y, w, h, r, g, b, a = 0.0)
  j0 = (y-0.5*h).to_i; j1 = (y+0.5*h).to_i
  i0 = (x-0.5*w).to_i; i1 = (x+0.5*w).to_i
  j0.step(j1) do |j|
    i0.step(i1) do |i| pset(i, j, r, g, b, a) end
  end
end
```

楕円を描く fillellipse は、円と同様で、ただし縦横をそれぞれ縦横の半径で割ってから半径 1 の円に入っているかどうかで判定すればよいでしょう:

```
def fillellipse(x, y, rx, ry, r, g, b, a = 0.0)
  j0 = (y-ry).to_i; j1 = (y+ry).to_i
  i0 = (x-rx).to_i; i1 = (x+rx).to_i
  j0.step(j1) do |j|
    i0.step(i1) do |i|
      if (i-x).to_f**2/rx**2 + (j-y).to_f**2/ry**2 < 1.0
        pset(i, j, r, g, b, a)
      end
    end
  end
end
```

三角形を描く filltriangle は、凸多角形を塗る fillconvex というのを作ってそれを呼ぶようにしました。

```
def filltriangle(x0, y0, x1, y1, x2, y2, r, g, b, a = 0.0)
  fillconvex([x0, x1, x2, x0], [y0, y1, y2, y0], r, g, b, a)
  fillconvex([x0, x2, x1, x0], [y0, y2, y1, y0], r, g, b, a)
end
```

[2] 回転させたい場合は後の「直線」を援用してください。

fillconvex はX座標、Y座標をそれぞれ配列で渡し、最後には最初と同じ要素を重複して入れておくことにします。また、点を指定する順序は「左回り」である必要があります (これらの理由は後述します)。しかし三角形で向きを考えるのも面倒なので、ここでは2頂点の順序を入れ換えて「右回りと左回り」で fillconvex を呼ぶようにしました (逆回りの方は何も塗れないのでとくに害はない)。

fillconvex では、まず座標の範囲は配列に入っているX座標やY座標の最大と最小を求め (最大と最小は前に演習でやったようなものですが、実は配列にはメソッド max と min があって最大と最小を計算してくれるのでそれを使っています)、その後各点についてそれが図形の内側にあるなら塗ります:

```
def fillconvex(ax, ay, r, g, b, a = 0.0)
  xmax = ax.max.to_i; xmin = ax.min.to_i
  ymax = ay.max.to_i; ymin = ay.min.to_i
  ymin.step(ymax) do |j|
    xmin.step(xmax) do |i|
     if isinside(i, j, ax, ay) then pset(i, j, r, g, b, a) end
    end
  end
end
```

図形の内側にあるかどうかは isinside で判定します。isinside は、与えられた点が「いずれかの辺の右側にある」なら図形の外にある、そうでなければ内側にあるか線上にある、と判断します。

```
def isinside(x, y, ax, ay)
  (ax.length-1).times do |i|
    if oprod(ax[i+1]-ax[i],ay[i+1]-ay[i],x-ax[i],y-ay[i])<0
      return false
    end
  end
  return true
end
```

右側にあるかどうかは、辺の線分の**ベクトル** (vector) と、線分の起点から調べたい点までのベクトルの**外積** (outer product) を計算して、負なら右側と判定します (このために左回りで周囲を指定するという条件が必要なのでした)。[3]

```
def oprod(a, b, c, d)
  return a*d - b*c;
end
```

このほか、線分が直交かどうか調べるにはベクトルの**内積** (inner product) が0かどうか調べればよいなど、図形処理においてベクトルの考え方はさまざまに活用できます。このような、プログラムで幾何学的な図形の計算を行うものを一般に**計算幾何学** (computational geometry) と呼びます。

線を描く fillline ですが、2点のXY座標と「線の幅」を指定します:

```
def fillline(x0, y0, x1, y1, w, r, g, b, a = 0.0)
  dx = y1-y0; dy = x0-x1; n = 0.5*w / Math.sqrt(dx**2 + dy**2)
  dx = dx * n; dy = dy * n
  fillconvex([x0-dx, x0+dx, x1+dx, x1-dx, x0-dx],
             [y0-dy, y0+dy, y1+dy, y1-dy, y0-dy], r, g, b, a)
end
```

[3]今回扱っているプログラムではY軸が下向き (つまり通常の座標系に対し鏡像) なので「右周り」になる。あまり気にせず通常の座標系だと思って左周りということでよい。

線分のベクトルからそれと直交するベクトルを計算し、その長さが線の幅の半分になるようにします。あとは線分の両端点と幅ベクトルを加減することで細長い長方形ができますから、それを `fillconvex` で塗ればよいわけです。

では最後に、さまざまな絵を描くメソッドを示します:

```
def mypicture
  fillcircle(150, 30, 60, 255, 100, 70, 0.0)
  fillcircle(190, 100, 30, 100, 200, 80, 0.5)
  fillrect(60, 100, 120, 80, 80, 220, 255, 0.6)
  fillellipse(200, 60, 70, 40, 100, 100, 220, 0.7)
  filltriangle(200, 100, 300, 100, 250, 200, 200, 100, 250, 0.5)
  fillline(40, 40, 260, 160, 4, 0, 0, 0, 0.0)
  writeimage("t.ppm")
end
```

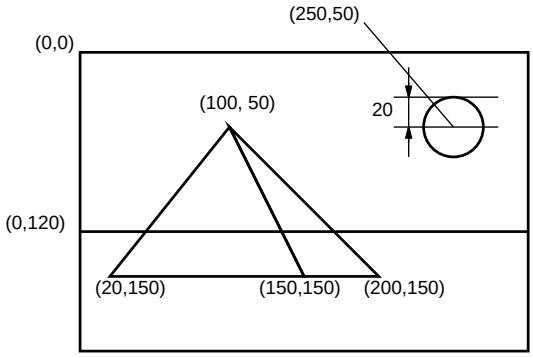


図 6.5: ピラミッドの絵の設計図

課題の指定にある「美しい」については皆様にお任せしているのですが、たとえば風景みたいに構図のある絵を描くとしたら、やっぱり何らかの設計が必要ではと思います。たとえば「海にうかぶピラミッドと太陽」という絵を描くものとします。まず、図 6.5 のように方眼紙などで構図の設計をして、それからそれぞれの図形を色指定して入れていく、みたいにすればそれらしくなるのではないでしょうか (図 6.6)。

```
def mypicture1
  fillrect(150, 60, 300, 120, 180, 240, 250)
  fillrect(150, 160, 300, 80, 20, 90, 200)
  filltriangle(100, 50, 150, 150, 20, 150, 120, 70, 20)
  filltriangle(100, 50, 200, 150, 150, 150, 160, 90, 80)
  fillcircle(250, 50, 20, 255, 0, 0)
  writeimage("t1.ppm")
end
```

なかなか大変でしたが、このように手続きを次々に作っていくことで、大きなプログラムでも見通しよく組み立てて行けることが納得いただけたかと思います。

報告課題 6A

今回は総合実習のため当日は「報告課題」(時間中にやったことの報告) です (プログラムの提出は不要)。簡単にまとめてください。

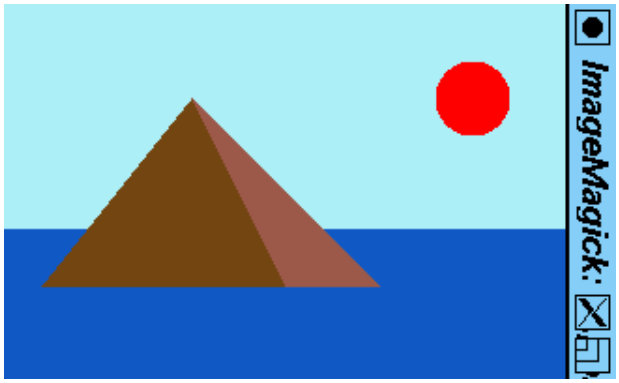


図 6.6: ピラミッドの絵の画像

Q1. 前回から今回までの間にどんな準備をしましたか。

Q2. 最終的にどのような絵を課題として作成する計画ですか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

総合実習課題 6B

B 課題については、ペアでやるかどうかはいつも通り選択可能です。**ただし、ペアであっても今回に限り「全く同じ絵ではないようにする」こと。**同一図柄の色違いは認めますので、それほど厳しい条件ではないと思います。総合実習課題は平常のレポートより配点を高くしますので頑張ってください。課題は次のものです。

課題 X　自分 (達) の「美しい」と思う絵を生成するプログラムを作成しなさい。

「美しい」の定義は各自にお任せしますので、自分のレベルに合った内容で結構です。レポートを重視するので、きちんとどのように美しいか書いてください。レポートは次の順で記述してください。

0. 表紙 — 学籍番号＋氏名、ペア学籍番号＋氏名 (あれば)、提出日付。
1. 構想・計画・設計 — どのような構想で絵を生成したか、具体的にどのように計画し、プログラムはどのように設計したか。
2. プログラムコード — 必ず動作するものを提出してください。また絵を生成するために呼び出す Ruby 命令を最後の行に追加してください (「`ruby` ファイル名」で実行できるようにするため)。
3. プログラムの説明 — プログラムのどの部分が何をしているかの説明をお願いします。
4. 生成された絵 — アップロードで提出してください。プログラムコードと絵が一致していること。レポートには生成された絵がどのようなものであるという説明を記してください。
5. 考察 — 課題をやって分かったことや感想など。
6. 以下のアンケートの解答。

    Q1. 画像が自由に生成できるようになりましたか。

    Q2. 画像をうまく生成する「コツ」は何だと思いましたか。

    Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

期限は次回授業前日一杯 (この点はいつもと同じ) ですが、出題が前回 (＃ 5) の授業時だったはずなので、通常よりは期間が長くなっています。生成する画像についてはクレジットつきでネットや会合等で紹介することがありますので、公序良俗に反する (ネット等に掲示できない) 画像を生成することはやめてください。

＃7　整列アルゴリズム+計算量

今回は「整列」のさまざまな手法を取り上げ、それに関連して、アルゴリズム解析において重要な「計算量」の概念について学びます。「整列」が題材なのは、1つの作業に対してさまざまなアルゴリズムがあるという点でぴったりだからです。というわけで今回の内容は次の通り。

- さまざまな整列アルゴリズムと
- 時間計算量の考え方

7.1　さまざまな整列アルゴリズム

7.1.1　整列アルゴリズムを考える

配列を扱うアルゴリズムの代表例として、数値の並びを受け取り、**昇順** (ascending order — 小さいもの順) に並べ換えることを考えます。これを**整列** (sorting) と言います。[1]

アルゴリズムの前に、皆様がふだん整列を行うとき (たとえば数字を書いたカードを順に並べる際)、どうするかやってみてください。ただしコンピュータに移すことを前提に、次の制限を設けます。

- カードは列にきっちり (間をあけずに左から詰めて) 並べること (配列に対応)。
- 2本の人差指だけを使ってカードを指して動かす (コンピュータでは本当は操作できるデータは一時には1個だけれど、さすがに不自由すぎるので2本にします)。
- カードの数値を読んだり比較するときは、2本の指のどちらかでそのカードを指す (これもコンピュータが操作できるデータは一時には1個だけだから)。

演習 0 数字のカード (10枚くらい) をよく切って机上に1列に並べ、上の条件を守って昇順に並べ替えなさい。ペアのところはペアで互いに観察し、相手のアルゴリズムを推察しなさい。

7.1.2　基本的な整列アルゴリズム: バブルソート exam

では、一番基本的な整列アルゴリズムを1つお見せしましょう。次の擬似コードを見てください。[2]

- bubsort(a): 配列 a を昇順に整列
- 　done ← 偽。
- 　done でない間繰り返し、
- 　　done ← 真。
- 　　i を 0 から a.length-2 まで変えながら繰り返し、
- 　　　もし a[i] と a[i+1] の順番が逆なら、
- 　　　　{a[i] と a[i+1] の値を交換。}
- 　　　　done ← 偽。
- 　　枝分かれ終わり。
- 　　繰り返し終わり。
- 　繰り返し終わり。

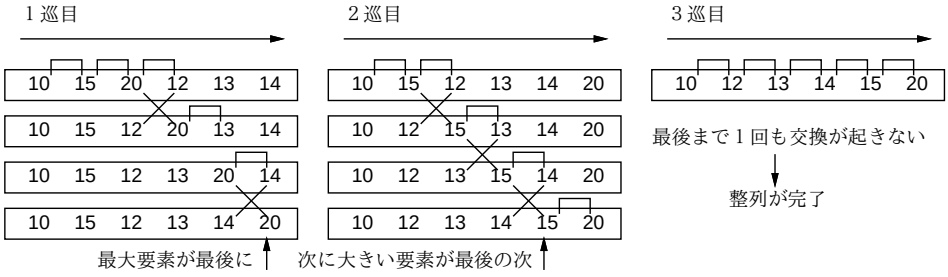


図 7.1: バブルソートによる整列

擬似コードでは手続きを呼ぶところは「{…}」で囲みました。このコードの肝となるところは、内側のループで隣り合う要素を順に見ていき、逆順になっているところがあれば交換する、というところです。これを次々におこなっていくと、大きい要素が右のほうに移動していきます (図 7.1)。この処理を繰り返していくと、最後は全要素が昇順で並び、交換が起きなくなります。

繰り返しを終わってよいかどうか判断するために、done(終了) という旗を用意し、まず立ててから上記の比較交換を行います。交換を行ったら、そのことを示すために旗を降ろします。最後まで旗が立ったままだったら、1 回も交換しなかった、つまり 1 箇所も逆順になっているところがなかったということなので、整列が完了しています。

この整列方法を**バブルソート** (bubblesort) と呼びます。各要素が移動する様子が水中から泡が浮かんでくるのに似ているためにこう呼ばれるとされています。

ところで、「a[i] と a[i+1] を**交換** (swap)」という命令は言語には直接ないので、これを手続きとして記述します。

```
def swap(a, i, j)
  x = a[i]; a[i] = a[j]; a[j] = x
end
```

これで配列 a の i 番目と j 番目の要素を入れ換えることができます (実際にそうなっていることをコードを追って確認しておいてください)。[3]

バブルソート本体は次のようになります。

```
def bubblesort(a)
  done = false
  while !done do
    done = true
    0.step(a.length-2) do |i|
      if a[i] > a[i+1] then swap(a, i, i+1); done = false end
    end
  end
end
```

では実際に動かしてみましょう。

```
irb> a = [1, 9, 5, 4, 2]
```

[1]逆に大きい順番の場合は**降順** (descending order) と呼びます。正確には列を昇順や降順に並べ換える処理が整列です。

[2]変数 done は交換が無くなったことを表す「**旗** (flag)」として使われています。配列をスキャンする前に旗を立て、交換が起きたらまだ完了でないので旗を降ろします。while 文の条件が「done でない間」なので、最初に「done ←偽」にしてから始めます (そうしないと while 文の中が実行されない)。

[3]この程度のコードなら、いちいちメソッドにせず、直接書いてもよいです。とくに Ruby では「a[i], a[j] = a[j], a[i]」という書き方ができますから。ここでは「交換」するところに「swap」と書いてある分かりやすいと思ったのでメソッドを作り、その中では他の言語でもできる「作業変数を介して交換する」書き方を使いました。

```
=> [1, 9, 5, 4, 2]
irb> bubblesort(a)
=> nil
irb> a
=> [1, 2, 4, 5, 9]
irb>
```

bubblesort 自体は値を返さず、配列 a の中身を書き換えるだけなのに注意。このため、まず配列 a を用意し、それを bubblesort で整列させ、最後に a を打ち出して並びを確認しています。

7.1.3 基本的な整列アルゴリズム: 選択ソート exam

バブルソートのように、「求める状態が成り立っていない間、少しでもその状態に近付けることをずっと繰り返す」というのはコンピュータではよくありますが、人間はあんまりそんなやり方はしない気もします。もうちょっと自然な考え方のものとして、次のものはどうでしょうか。

> 数の並びから最小値を取り出してはその並びからは取り除くことを繰り返していく。その取り出したものを順に並べると昇順の整列結果になっている。

これは、最小の要素を次々に選ぶことから**選択ソート** (selection sort) ないし**単純選択法**と呼びます。

これを作るに当たっては、「配列 a の i 番目から j 番目までの間で最も小さい要素が何番目にあるかを返す」操作を下請けメソッドとして用意するのがいいでしょう。それがあったとして、アルゴリズムは次のようになります。

- selectionsort(a): 配列 a を単純選択法で整列
- 　i を 0 から a.length-2 まで変化させながら繰り返し、
- 　　k ← {a の i 番以後の最小要素の番号。}
- 　　{a[i] と a[k] の内容を交換。}
- 　繰り返し終わり。

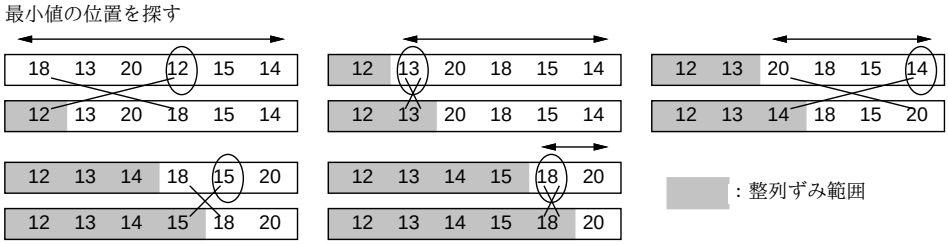


図 7.2: 単純選択法による整列

なぜ「交換」を使っているのかというと、まず選んだ最小の要素を先頭に置くには、先頭にある要素と最小の要素とを交換するのが合理的だからです。その後も、残っているものの中から最も小さい要素を選んではその先頭位置と交換することで、1 つの配列だけですべての作業が行えます (図 7.2)。

演習 1 次の手順で単純選択法のプログラムを作成しなさい。

a. まず、「配列 a の i 番〜j 番の中での最小要素の番号を返す」メソッド arrayminrange(a, i, j) を作成する。正しく動作することを確認すること。

b. 続いて、それと swap(a, i, j) を呼び出しながら配列を整列する単純選択法のメソッド selectionsort(a) を作成する。正しく動作することを確認すること。

7.1.4　基本的な整列アルゴリズム: 挿入ソート exam

単純選択法は、数を 1 つずつ処理しますが、それらを「取り出す」時に正しい順になるようにするというものでした。人間にとって自然なもう 1 つのやり方は、取り出すのは「最初に並んでいる順」で、入れる時に正しい位置に入れる、というものです。

> 数の並びから順に数を取り出し、それを新しい列に加えるが、その際に「順番として正しい」位置に挿入する (その後ろにある要素は全て 1 個ぶんずつずらす)。

これは、各要素を順次あるべき位置に挿入していくことから、**挿入ソート** (insertion sort) ないし**単純挿入法**と呼びます。これを作るに当たっては、「配列の a の i 番目が空いているものとして、x より大きい要素を 1 つずつ詰めていって、最終的な空きの位置を返す操作」を下請けメソッドとして作るのがいいでしょう。それがあったとして、単純挿入法のアルゴリズムは次のようになります。

- insertionsort(a): 配列 a を挿入ソートで整列
- 　i を 1 から a.length-1 まで変化させながら繰り返し、
- 　　x ← a[i]。
- 　　k ← {a の i 番目以前で x より大な値をずらしていき、最終的に x が入る位置を返す。}
- 　　a[k] ← x。
- 　繰り返し終わり。

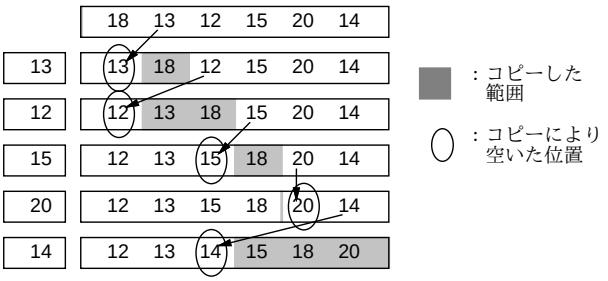


図 7.3: 単純挿入法による整列

これも、元の配列からデータを取り除きながらそれをもとに先頭部分に整列されている部分を作っていくので、配列は 1 つだけで済みます (図 7.3)。なお、配列を「後ろにずらす」時に後ろから順にやらないとまずいことに注意してください (図 7.4)。

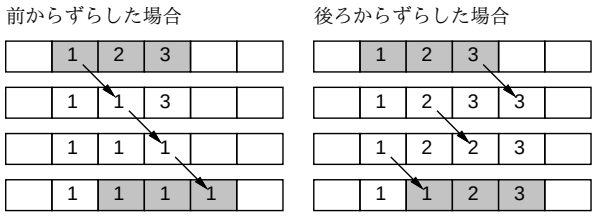


図 7.4: 配列をずらす

演習 2 次の手順で単純挿入法のプログラムを作成しなさい。

a. 配列 a の位置 i が空いているものとして、位置 i より前にある x より大きい要素を (位置 i を埋めるように)1 つずつ後ろに詰めて行き、最終的な空きの位置を返すメソッド `shiftlarger(a, i, x)` を作りなさい (すべての値が x より大きければ全部詰めて位置 0 番が空くので 0 を返す)。正しく動作することを確認すること。

b. 上記を呼び出しながら単純挿入法で配列を整列するメソッド `insertionsort(a)` を書きなさい。正しく動作することを確認すること。

7.1.5 整列アルゴリズムの計測 exam

次の段階として、「整列プログラムの時間を計測する」問題に取り組んでみましょう。「バブルソート」「単純選択法」「単純挿入法」の 3 つの整列アルゴリズムについて、データ量を変化させて**時間計測**を行ってみます。データは次のコードにより個数を指定して**乱数**によりランダムに生成します。

```
def randarray(n)
  return Array.new(n) do rand(10000) end
end
```

`rand` は乱数を生成するメソッドで、パラメタを指定しないと区間 $[0, 1)$ の**一様乱数** (uniform random number) を (実数値で) 返し、パラメタとして整数 N を指定すると、0 以上 N 未満の整数値の一様乱数を返します。ちょっと試してみましょうか。

```
irb> randarray 10
=> [9257, 4988, 6894, 8064, 329, 4362, 1868, 472, 1527, 6317]
```

次に、時間計測に役立つメソッドを用意します。これは、実行回数 (省略した場合は 1) とブロックを受け取って動作します。使い方は「`bench do 測りたい処理 end`」です。

```
def bench
  t1 = Process.times.utime    # (1)
  yield                       # (2)
  t2 = Process.times.utime    # (3)
  return t2-t1                # (4)
end
```

(1) Process.times.utime は、Ruby 処理系が現時点で消費した CPU の時間 (秒) を取得します。

(2) 「yield」は特別な命令で「このメソッドに渡されて来た do...end の中身を実行」します。

(3) そのあとで再度 Process.times.utime で CPU 使用時間を調べ、

(4) 両者の差を返せば、それは do...end の中身を実行するのに要する CPU 時間であるはずです。

さて、これらの材料を使って、整列の速度を測ります。randarray で多めの配列を生成し、bench の中で整列を行い、時間を計測します。これを 1 行にまとめて書くとして、次のようになります。

```
irb> a = randarray(1000); bench do bubblesort(a) end
=> 1.21875    ←計測結果が表示される
```

演習 3 ここまでに動かした整列アルゴリズムアルゴリズムについて、データ数 N を変化させて「N と所要時間の関係」を検討しなさい。自分で作成したプログラムが 1 つは含まれること。

7.1.6 基本的な整列アルゴリズムの計測

表 7.1 に筆者の手元のマシンでのバブルソート、単純選択法、単純挿入法の計測結果を示します。これを見ると、バブルソートが圧倒的に遅く、残りの 2 つはそれほど差はありません。これは、バブルソートはすべての要素を移すのに隣と 1 個ぶんずつ交換してゆくので手間が多くなるのに対し、他の 2 つは「1 個データを選んで、それを適切な位置に置く」ことの反復なので、それだけ手間が少ないからと言えるでしょう。

表 7.1: バブルソート/単純選択法/単純挿入法の所要時間 (msec)

データ数	1,000	2,000	3,000
バブルソート	1,219	4,945	11,117
単純選択法	305	1,242	2,766
単純挿入法	375	1,531	3,445

では次に、データの量が 2 倍、3 倍になった時の所要時間を見てみると、こんどはどのアルゴリズムでも所要時間がほぼ 4 倍、9 倍になっていることが分かります。$4 = 2^2$、$9 = 3^2$ ですから、どのアルゴリズムでも「所要時間はデータ量の 2 乗に比例している」と言ってよいでしょう。ということは、データ数が 100,000(100 倍) になった時の所要時間は単純選択法でも $0.3 \times 10,000 = 3000$ 秒 $= 50$ 分 (!) となってしまい、終わるまで待つのはあまり嬉しいものではないと分かります。

7.2 より高速な整列アルゴリズム

7.2.1 マージソート exam

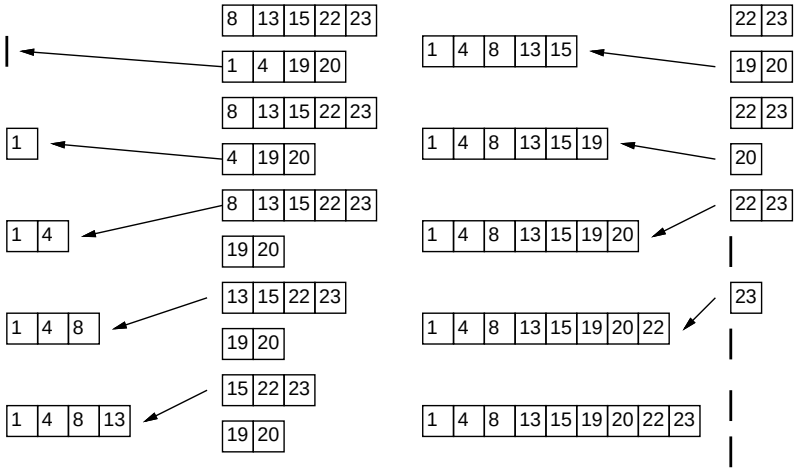


図 7.5: マージの処理

では、整列アルゴリズムでもっと速いものはないのでしょうか。ここで**マージソート** (merge sort) と呼ばれるアルゴリズムを見てみましょう。**マージ** (merge) とは**併合**とも呼ばれ、図 7.5 のように 2 つの整列ずみの列を「あわせて」1 つの整列ずみの列にすることを言います。この操作は、2 つの列それぞれの先頭だけ調べればできるので効率よく実行できるのです。

マージソートの手続きには「`mergesort(a, i, j)`」のように呼び出すと、配列 `a` の `i` 番から `j` 番の範囲だけを整列します。Ruby コードを示します。パラメタのところが代入になっていますが、これは**デフォルト引数**で、`i` と `j` を指定しない場合は 0 番目から `a.length-1` 番目、つまり配列全体が整列されます。

```
def mergesort(a, i = 0, j = a.length-1)
  if j <= i
    # do nothing
  else
    k = (i + j) / 2
    mergesort(a, i, k); mergesort(a, k+1, j)
    b = merge(a, i, k, a, k+1, j)
    b.length.times do |m| a[i+m] = b[m] end
```

```
    end
  end
```

短いですが、再帰を使っているので読み方にコツが必要ですね。

- まず、整列する範囲の長さが 1 以下なら「もう並んでいる」ことになるので何もしない。
- そうでないなら、範囲の中間位置 k を計算して (分割)、
- 中間位置までと、それより後の部分をそれぞれ整列する (自分自身を再帰呼び出しすると整列できる「はず」)。
- 整列できたら、その 2 つの範囲をマージする。
- 最後に、マージ結果の配列 b から元の配列 a の i 番から j 番までの範囲に値をコピーし戻す。

再帰が進み戻って来る様子を図 7.6 に示します。点線の囲みが大きさ 2 以上での呼び出しです。

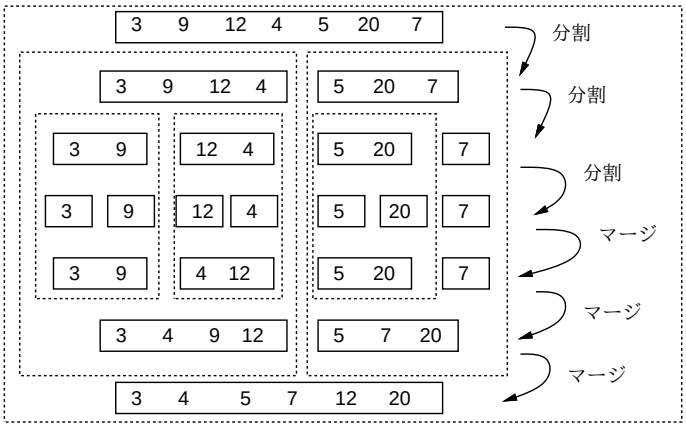


図 7.6: マージソートによる整列

下請けとなるマージも見てみます。どちらかの列に値が残っている間繰り返し、「片方しか残っていなければそちらから、両方残っているなら先頭の値の小さい方から」1 つ取り出して配列 b に追加し、取り出した列の先頭位置を進める (つまりその列が 1 つ短くなる)、わけです。

```
def merge(a1, i1, j1, a2, i2, j2)
  b = []
  while i1 <= j1 || i2 <= j2 do
    if i1 > j1 then            b.push(a2[i2]); i2 = i2 + 1
    elsif i2 > j2 then         b.push(a1[i1]); i1 = i1 + 1
    elsif a1[i1] > a2[i2] then b.push(a2[i2]); i2 = i2 + 1
    else                       b.push(a1[i1]); i1 = i1 + 1
    end
  end
  return b
end
```

7.2.2 クイックソート exam

もう 1 つ別の、**クイックソート** (quicksort) というアルゴリズムを直接 Ruby プログラムで示します。

```
def quicksort(a, i = 0, j = a.length-1)
  if j <= i
```

```
    # do nothing
  else
    pivot = a[j]; s = i
    i.step(j-1) do |k|
      if a[k] <= pivot then swap(a, s, k); s = s + 1 end
    end
    swap(a, j, s); quicksort(a, i, s-1); quicksort(a, s+1, j)
  end
end
```

非常に短いですが、説明されないと分かりませんね。まず長さ 1 以下なら何もしないのはマージソートと同様です。次に、マージソートと同じく列を 2 つに分けますが、こちらは**ピボット** (pivot) と呼ぶある値 p を選び、「左半分は p 以下、続いて p の値、右半分は p より大きい」という状態にしてから、左半分と右半分をそれぞれ自分自身を再帰呼び出しして整列します。そうすると、「p 以下の整列された列」「p」「p より大きい整列された列」になるのでこれで整列が完了するわけです (図 7.7)。

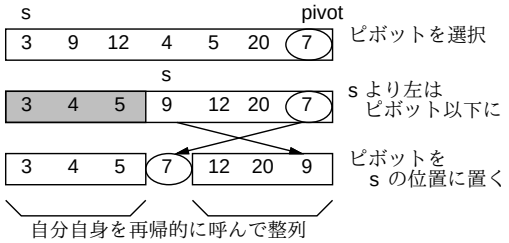


図 7.7: クイックソートによる整列

p としては「ちょうど列を半分ずつに分ける値」を使えるとベストですが、そんなものは分からないのでランダムに選ぶこととし、上のコードでは右端 (j 番目) の値を p にしています。変数 s は「この番号の 1 つ手前までは p 以下のものを詰めてあるので、次に p 以下のものが見つかったらこの位置に入れる」番号を表しています。そこで、k を i から j-1 まで左から順に調べて、[4]a[k] が p 以下ならそれを s 番目の要素と交換して s を増やすことで、左半分と右半分に分けられます。分け終わったら、最後に j 番目と s 番目を交換することで、保留してあったピボットの値をあるべき位置に置きます。その後、自分自身を再帰的に呼ぶわけですが、s 番目のピボットの位置はこれで合っているので、i〜s-1 と s+1〜j の範囲について自分自身を呼びます。

演習 4 mergesort または quicksort を動かしてみなさい。データ数 N を何通りかに変化させて時間も測ること (N を大きくしないと 0.0 となり測れません)。N と所要時間の関係を検討すること。

7.3 整数値のための整列アルゴリズム

7.3.1 ビンソート

ここまでの方法とは考え方がまったく違う整列アルゴリズムである、**ビンソート** (bin sort) ないし**バケツソート** (bucket sort) を紹介しましょう。このアルゴリズムは、整列する値が整数であり、かつ範囲があまり広くない場合に利用できます。たとえば、整列する値の範囲が 0〜3 の整数だけだったとします (もちろん、そのデータの個数は 1 万も 2 万もあるかもしれません)。

そこで図 7.8 のように、まず 0、1、2、3 それぞれの値について「何回現れるか」を数えます。続いて、「0 が 2 回、1 が 3 回、…」のように数えた個数ずつその値を繰り返せば、元のデータを並べ換え

[4] j 番はピボットが入っているので保留します。

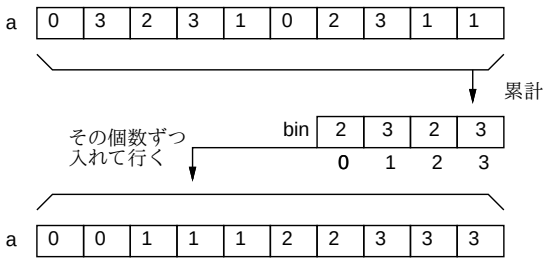


図 7.8: ビンソートによる整列

たのと同じことになるわけです。0〜3 ではあまり役に立たないしょうが、実際にはコンピュータのメモリは沢山あるので、0〜9999 とかでも全く問題ありません。[5]

演習 5 次の段階を踏んでビンソートのプログラムを作成しなさい。動くことを確認すること。そのあと、N を何通りかに変化させて時間も測ること (データ数を少し大きくしないと 0.0 となり測れません)。N と所要時間の関係を検討すること。

a. 配列 a の中に 0〜9999 の値がそれぞれ何回現れるかを集計するメソッド `makebin(a)` を作る。その中ではまず 10000 要素の初期値 0 の配列 b を作る。[6] 続いて、a に入っている値 x を順に取り出し、それぞれについて `b[x]` の値を 1 増やす。最後に値として b を返す。

b. 配列 a を受け取り、ビンソートにより整列するメソッド `binsort(a)` を作る。中ではまず `makebin(a)` を呼び出し、結果を配列 b として受け取る。次に、i を 0〜9999 の範囲で変化させながら、a の先頭から「0 を `b[0]` 個、1 を `b[1]` 個、…、i を `b[i]` 個入れてゆく。[7]

7.3.2 基数ソート

ビンソートの弱点は、現れる値の範囲があまりに広いと (100 万とか 1000 万とか) 巨大な配列を必要とし、効率も悪くなることです。そこで、やはり値が整数である必要があるものの、ビンソートよりも値の範囲に対する許容度が高い整列アルゴリズムである**基数ソート** (radix sort) を紹介しましょう。ここでは簡単のため、負の値はないものとして説明します。

基数ソートでは、整列する値を 2 進表現した時に「下から i ビット目が 1 であるか否か」を調べる必要があります。これを Ruby でどう書くかを説明しておきます。

Ruby では<<という演算子は**左シフト** (left shift) つまりビット列である整数値を 1 ビットぶん左にずらす働きがあります。だから `1 << 2` は $100_{(2)}$ だから 4 だし、一般に `1 << i` で i 番目のビットだけが 1 になった数値をえることができます。[8]

次に、&という演算子は**ビット毎 and**(bitwise and) 演算つまり 2 つの数の 2 進表現で「両方とも 1」の位置だけが 1、それ以外は 0 であるような 2 進表現に対応する数が得られます。[9] たとえば図 7.9 のように、`52 & 29` の結果は 20 ということになります。

```
  110100 — 52
&) 011101 — 29
  ------
  010100 — 20
```

図 7.9: ビット毎 and 演算

[5] 先の `randarray` が生成するデータもこの範囲の整数であることに注意。もちろんわざとそうしたのですが。

[6] N 要素の配列で初期値を 0 とするのは `Array.new(`N`, 0)` でできるのでしたね。

[7] まず変数 k を 0 に初期化してからループに入り、それぞれの i に対して「`a[k]` に i を入れてから k を 1 増やす」ことを `b[i]` 回行なえばよい。

[8] そして今回は使いませんが、もちろん>>は**右シフト** (right shift) 演算子です。

[9] 条件の「かつ」は&&でしたが、アンド記号が 1 個の場合はまったく別の意味になるわけです。ちなみに、|は**ビット毎 or**(bitwise or) 演算、~は**ビット毎反転** (bitwise inversion) 演算です。

では、基数ソートの説明に入ります。たとえば、変数 mask に 1 ビットだけが「1」になっている値を入れ、その 1 の位置を一番右 (下位) から順に左に移していきます。そして、その mask との&の結果が 1 か 0 かで、データを右半分と左半分に分割します (図 7.10)。そうするとあらふしぎ、一番上のビット (ここでは 4 ビットとしました) までやったときには、すべての数は小さい順に並んでいます。

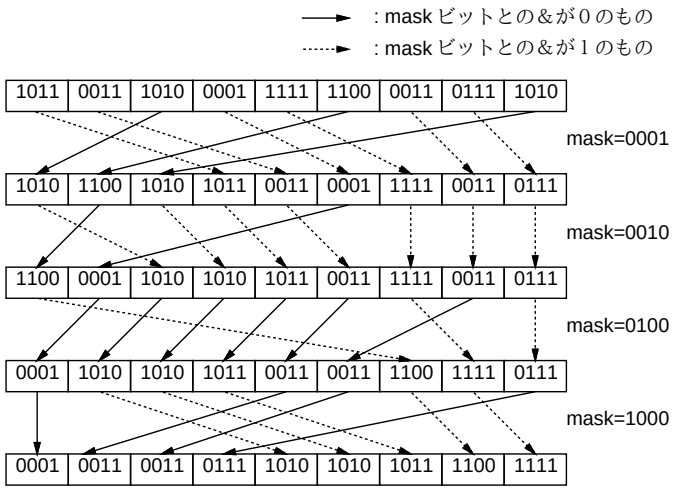


図 7.10: 基数ソートによる整列

なぜかというと、1 回目では一番下のビットが 1 のものが左、0 のものが右になるように振り分け、それについて 2 回目に下から 2 ビット目が 0 のものが左、1 のものが右になるように振り分けるわけですが、2 回目の振り分けをしても 1 回目の振り分けの順序は崩れないので、2 ビット目が 0 のものの中や、1 のものの中ではそれぞれ、まず 1 ビット目が 0 のもの、続いて 1 のものという順序が維持されています。3 ビット目、4 ビット目でも同様にそれより下のビットについては順序が維持されているので、結局最後まで来たときには順番が完全に並んだ状態となるわけです。

演習 6 基数ソートのプログラムを作成しなさい。データ量と所要時間の関係を検討すること。

7.4　時間計算量

7.4.1　時間計算量の考え方 exam

ここまででさまざまな整列アルゴリズムを実現したプログラムの所要時間の計測について話題にしてきましたが、本節ではアルゴリズムの**性能** (performance) を評価する指針の 1 つである**計算の複雑さ** (computational complexity) ないし**計算量** (complexity) について取り上げます。complexity だと日本語は「複雑さ」になりそうですが、「複雑さ」という日本語では一般的すぎて何のことか分かりにくいので、日本語では「計算量」と呼ぶわけです。なお、計算量には「どれくらいメモリが必要になるか」を表す**領域計算量** (space complexity) もありますが、ここではとりあえず「所要時間」に着目する**時間計算量** (time complexity) を取り上げます。

バブルソートを例題にして、これがどれくらいの時間を要するかかを見積もってみましょう。その前に、まず次の前提を置きますが、これはいいですね?

コンピュータは、1 つの決まった動作はその動作に応じた決まった時間で実行する。

では、バブルソートでどれくらいの「動作」を行なうでしょうか。バブルソートの「一番沢山処理する部分」というのは、「配列を順番に調べて逆順なら交換する」ところです。そこで「比較し、交換する」のに掛かる処理をおよそ T_1 であるものとします。配列の長さが N であれば、配列全体を処理するのに掛かる時間は $(N-1)T_1 \sim NT_1$ ということになります (N は大きいので N と $N-1$ はほぼ等しいでしょう)。

問題はこれを「何回」やるかです。よく観察すれば分かる通り、1 回スキャン (端から端まで比較交換) すると「最大の値が右端に」来ます。2 回目は「最大の次の値が右から 2 番目に」来ます。ですから、運が悪いと (完全に逆順に並んでいる時)、全部の値が並び終わるまでにスキャンする回数は N 回になります。運が良いと (最初から完全に並んでいる時) は 1 回です。平均で $\frac{N}{2}$ 回とすると、所要時間は次のようになります。

$$\frac{N^2 T_1}{2}$$

ここまで T_1 が何 (マイクロ?) 秒かとか一切考えないで来ましたが、実はこの定数は「コンピュータが違うと」「プログラミング言語の処理系が違うと」大きく変わってしまうので、測ってもあまり意味はありません。

じゃあ何の意味があるのかというと、「所要時間は N^2 に比例する」(これを $O(N^2)$ と書き「オーダー N^2」と呼びます) ということはとても役に立ちます。それは先にやったように、あるアルゴリズム (例: バブルソート) が $O(N^2)$ で、1000 個でやって 1 秒だったら、10,0000 個でやったら 1 万秒掛かる (それはやりたくない)、と分かるからです。

この「$O(f(N))$」の形で計算時間の「次数、オーダーを」表現したものを時間計算量と呼ぶのです。

そういうわけで、時間計算量とは「最も高次の次数だけを問題にする」考え方なわけですが、「ではアルゴリズムが違う時は定数 T_1 が問題になるんじゃないか」と思うかも知れません。はい、正解で、間違いです。

まず、同じ時間計算量 ($O(N^2)$ とか) の場合は、定数の違いがそのまま時間の違いになります。そういう意味では正解です。しかし、別のアルゴリズム/プログラムの所要時間が NT_2 つまり時間計算量 $O(N)$ だとしたらどうでしょう? たとえ T_2 が T_1 の 1000 倍だったとしても、データ数 N が 10000 になったら $N^2 T_1' > NT_2$ になってしまいますね? ですから、オーダーが違うときはそのオーダーだけでほぼ決まる、というのが時間計算量の考え方なわけです (ちなみに $T_1' = \frac{T_1}{2}$ とします)。

7.4.2 整列アルゴリズムの時間計算量

バブルソートは分かりましたが、では単純選択法の時間計算量はどうでしょう。単純選択法では、最初は N 個の要素から最小を探し、左に置きます。2 回目は $N-1$ から最小を探し、…と続けて、1 個から最小を探すところまでやります。探すというのは、「a[i] と min と比較し、小さければ min ← a[i] とする」処理ですから、これを T_3 とします。そうすると、処理全体は $(N+(N-1)+(N-2)+\cdots+1)T_3 = \frac{N(N+1)T_3}{2} \sim \frac{N^2 T_3}{2}$ で、これも $O(N^2)$ です。

単純挿入法の時間計算量はどうでしょうか。外側のループで i を 1〜N まで変えながらその番号の要素を適切な位置に挿入していきます。挿入位置を探索するのに平均して $\frac{i}{2}$ 個の要素を比較し、挿入位置が見つかったら平均して $\frac{i}{2}$ の要素を後ろにずらす必要があります。なのでこれも $1+2+\cdots+(N-1)$ の定数倍、つまり $O(N^2)$ になります。

ここで表 7.1 の結果を振り返ると、単純選択法もバブルソートも N が 2 倍、3 倍になった時所用時間が 4 倍、9 倍になっているので、この計測結果はこれらが確かに $O(N^2)$ の時間計算量であることを裏付けています。

ではマージソートの計算量はどうでしょうか。1 つの `mergesort` の呼び出しを見ると、単純な場合 (長さが 1 以下) は一定時間で済みます。長さ N の場合は、それを前半と後半に分けて、それぞれ自分自身を再帰的に呼び出して整列し、最後にマージします。自分自身に掛かる時間は分けて考えるとして、マージは両方の列の先頭を見て小さいほうを取ることを繰り返せばいいので、$O(N)$ で済みます。さて、再帰呼び出しのほうはどうでしょうか。長さ N の列を半分にしてそれぞれ `mergesort` を呼ぶのですから、2 段目の呼び出しは $O(\frac{N}{2}) + O(\frac{N}{2}) = O(N)$ 。3 段目は 4 分の 1 の列について 4 つ呼ぶのでやはり $O(N)$ 、となります。これが合計何段あるかというと、「N を何回半分にしたら 1 になるか」だから $\log_2 N$ となります。なので、全体では $O(N \log N)$ の計算量となります (計算量の議論では log の底が何かも省略するのが通例です)。

では、クイックソートの計算量はどうでしょうか。1 回ぶんの処理はやはり $O(N)$ で、再帰の段数はピボットの選択が完璧なら $\log_2 N$ 回ですが、ランダムに選んでいるのでその定数倍と考えてよいでしょう。すると定数倍は無視するので、これも計算量は $O(N \log N)$ になります。

ただし、極めて運が悪い場合、つまりピボットの選択が悪くて毎回列の最大か最小の値をピボットにしてしまうと、段数が N になってしまうので、最悪の計算量は $O(N^2)$ ということになります。そんな運が悪いことはないだろうと思うかもしれませんが、既に整列ずみの値を渡されるとまさにそうなってしまうのです。

演習 7 クイックソートに既に並んでいる配列を与えると計算量が $O(N \log N)$ から $O(N^2)$ になってしまうことを計測で確認しなさい。また、この弱点を解消する工夫を考えて実現しなさい。

本日の課題 7A

「演習 1」「演習 2」または「演習 5」のプログラムを作り、時間計測もおこなって結果を提出しなさい。プログラム、簡単な説明、複数のサイズでの計測結果が含まれること。アンケートの回答もおこなうこと。

Q1. 整列アルゴリズムを少なくとも 1 つは理解しましたか。

Q2. 時間計算量という考え方についてどう思いましたか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 7B

「演習 1」〜「演習 7」の (小) 課題から選択して 1 つ以上プログラムを作って動かし、レポートを提出しなさい。プログラムと、課題に対する報告・考察 (やってみた結果・そこから分かったことの記述) が含まれること。アンケートの回答もおこなうこと。

Q1. 整列アルゴリズムをいくつ理解しましたか。それぞれの時間計算量についてはどうですか。

Q2. 自分で作れる程度のプログラムについてなら、その時間計算量が求められそうですか?

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

＃8　計算量 (2)+乱数とランダム性

時間計算量は難しいところだったので、今回はとりあえず演習問題解説を復習を兼ねてやった後、改めて時間計算量を求める演習を用意しました。その後で以下内容が本題となります。

- 既出アルゴリズムの別解法と計算量
- 乱数とランダムアルゴリズム

8.1　前回演習問題の解説

8.1.1　演習 1 — 単純選択法

単純選択法の整列プログラムと、その下請けメソッドを示します。arrayminrange は、以前扱った「最小値を求める」コードと似ていますが、最小値のほかにその位置 p も覚えておいてそれを返すということです。

```
def selectionsort(a)
  0.step(a.length-2) do |i|
    k = arrayminrange(a, i, a.length-1); swap(a, i, k)
  end
end
def arrayminrange(a, i, j)
  p = i; min = a[p]
  i.step(j) do |k|
    if min > a[k] then p = k; min = a[k] end
  end
  return p
end
def swap(a, i, j)
  x = a[i]; a[i] = a[j]; a[j] = x
end
```

8.1.2　演習 2 — 単純挿入法

単純挿入法の整列プログラムと、その下請けメソッドを示します。shiftlarger は、渡された i から始めて、繰り返し「a[i] = a[i-1]; i = i - 1」を i を減らしながら行ないますが、i が 0 になったときと、次にコピーする値が x 以下になったときはそこで終わります。どちらでも返す位置は「最後にコピーをやめた位置」です。

```
def insertionsort(a)
  1.step(a.length-1) do |i|
    x = a[i]; k = shiftlarger(a, i, x); a[k] = x
  end
end
def shiftlarger(a, i, x)
```

```
  while i > 0 && a[i-1] > x do a[i] = a[i-1]; i = i - 1 end
  return i
end
```

8.1.3 演習 5 — ビンソート

ビンソートを行なう binsort とその下請け makebin を示します。makebin は添字が 0-9999 の範囲で初期値 0 の配列 b を用意してから、a の各要素 x について b[x] を 1 増やします。ですから、たとえば a 中に「321」が 10 回現れたら、合計で 10 回 1 増やされるわけです。本体では、今度は 0 から順にその番号が現れた回数だけ a の中にその番号の数を並べて行きます。

```
def binsort(a)
  b = makebin(a); k = 0
  b.each_index do |i|
    b[i].times do a[k] = i; k = k + 1 end
  end
end
def makebin(a)
  b = Array.new(10000, 0)
  a.each do |x| b[x] = b[x] + 1 end
  return b
end
```

8.1.4 演習 6 — 基数ソート

基数ソートを十進で説明し直します (図 8.1)。たとえば 3 桁の場合、最初は下から 1 桁目に着目して「0」〜「9」の箱に分類し、次にそのままの順で取り出した後、下から 2 桁目に着目して「0」〜「9」の箱に分類します。再度そのままの順で取り出した後、下から 3 桁目で分類すると全部並んでいます。

なぜこうなるかというと、ある桁について並べ終わった後はその順を崩さないで分類・移動を行っているので、下から p 桁目まで来た時には「それ以降の桁については順番になっている」状態であり、以後もその順が崩れないから全部の桁が終わったら完全に整列できるわけです。ここでは十進で説明しましたが、2 進であれば「箱」は 2 つでよいわけです。

次に 2 進による基数ソートのコードを示します。整列する値のビット数を指定します (10000 までの値だと 14 ビットで済むので 14 を指定)。各周回ごとに当該ビットの値に応じてデータを配列 b と c に振り分け、終わったらこの順で a にコピーし戻しています。

```
def radixsort(a, bits)
  b = Array.new(a.length); c = Array.new(a.length)
  bits.times do |pos|
    mask = 2**pos; bc = 0; cc = 0
    a.length.times do |i|
      if (a[i] & mask) == 0
        b[bc] = a[i]; bc = bc + 1
      else
        c[cc] = a[i]; cc = cc + 1
      end
    end
    bc.times do |i| a[i] = b[i] end
```

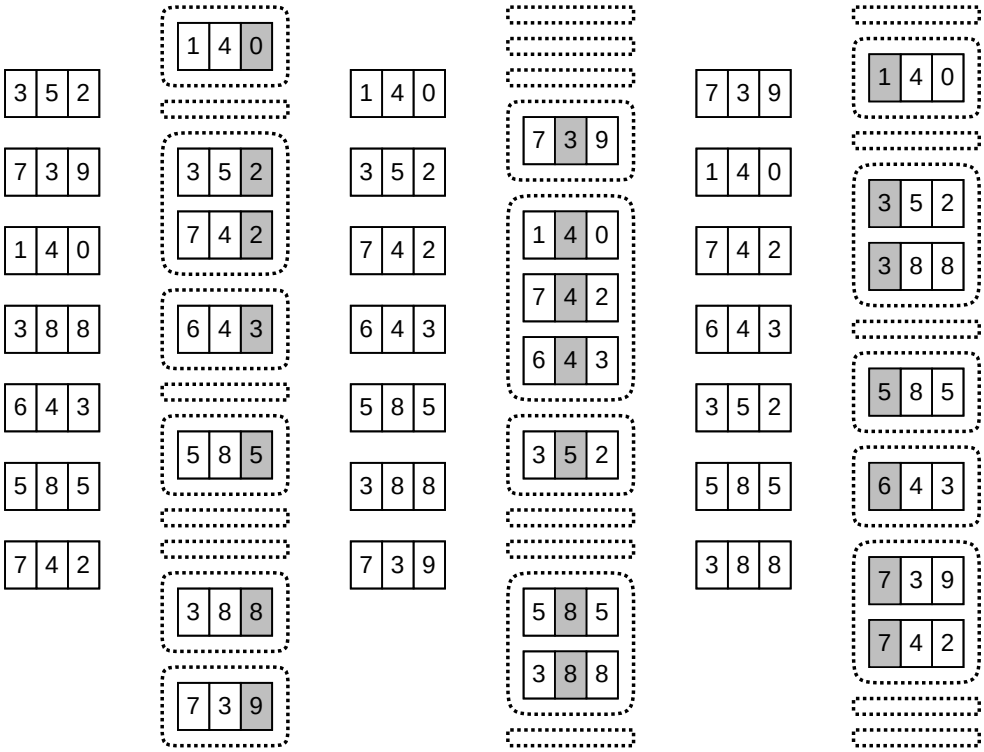


図 8.1: 十進法の場合の基数ソートの例

```
    cc.times do |i| a[bc+i] = c[i] end
  end
end
```

8.1.5 演習 3〜4 — 整列アルゴリズムの時間計測

各種整列アルゴリズムの計算時間を N を変えて手元のマシンで計測した結果を 8.1 に示します。

表 8.1: さまざまな整列アルゴリズムの時間計測

データ数	1,000	2,000	3,000	5,000	10,000	20,000	30,000	50,000
バブルソート	1,219	4,945	11,117	-	-	-	-	-
単純選択法	305	1,242	2,766	-	-	-	-	-
単純挿入法	375	1,531	3,445	-	-	-	-	-
マージソート	23	62	94	164	351	758	1,172	2,055
クイックソート	15	31	55	109	219	508	789	1,492
ビンソート	8	8	11	14	20	34	47	74
基数ソート	27	57	86	141	280	566	838	1,402
$N+E$	11,000	12,000	13,000	15,000	20,000	30,000	40,000	60,000

$O(N^2)$ のアルゴリズムであるバブルソート、単純選択法、単純挿入法は、N が大きくなると急激に遅くなって役に立たなくなります。一方、$O(N \log N)$ のマージソートとクイックソートは十万くらいのデータであれば十分実用になると言えます。

ビンソートは極めて速いことは分かりますね。では、このアルゴリズムの時間計算量はどうでしょう。まず、すべてのデータを順に走査するという点では $O(N)$ だと言えますが、それだけではありません。値の数を E とすると、最後に大きさ E の配列を全部調べながら値を生成するので、このため

の時間も E が大きいと問題になります。なので、時間計算量は $O(N+E)$ になるわけです。今回は E が 10,000 なので、途中まではこちらの方が主に問題になります。表 8.1 の一番下に $N+E$ を示しましたが、所要時間がだいたいこれに比例していることが読み取れます。

そして、ビンソートは E 個ぶんの配列を必要とすることも忘れてはいけません。アルゴリズムによっては、大量のメモリを使うことで時間を速くすることができますが、ビンソートはまさにその例です。ビンソートが要する記憶領域は元のデータ数 N と数えるための配列の数 E を併せたものだから、これを「**領域計算量**が $O(N+E)$ である」のように言います。つまり、値の範囲が広くなると、ビンソートは領域計算量の点でも不利になるわけでです。

なお、これまでに出て来たアルゴリズムのほとんどは領域計算量 $O(N)$ ですが、マージソートと基数ソートは「別の場所に移してから戻す」ので $O(2N)$ になって います。[1]

最後に基数ソートの時間計算量ですが、キーのビット数ぶんだけ振り分け処理をおこなうので、時間計算量は $O(N \log E)$ ということになります。これを $\log E$ が今回は定数 (14) と考えれば、時間計算量は**線形時間**ということになります。実際、表 8.1 をチェックすると所要時間が N にほぼ比例していることが分かります。

ただし、時間そのものは半分くらいのところまで、クイックソートよりも劣っています。これはつまり、データが非常に多くなると、先に説明したようにオーダーの差がすべてを支配しますが、それほどでもない場合には定数項の差が無視できず、オーダーの大きいアルゴリズムでも処理時間が短くて済む場合がある、ということを意味しています。たとえば、データが数個しかないのであれば、クイックソートを使うよりも単純選択方を使うほうが適切なわけです。

8.1.6　演習 7 — クイックソートとその弱点

再度クイックソートの説明をしてから、その時間計算量を考えます。まずコードを示します。

```
def quicksort(a, i, j)
  if j <= i
    # do nothing
  else
    pivot = a[j]; s = i
    i.step(j-1) do |k|
      if a[k] <= pivot then swap(a, s, k); s = s + 1 end
    end
    swap(a, j, s); quicksort(a, i, s-1); quicksort(a, s+1, j)
   end
end
```

長さ 1 以下なら何もしないのはマージソートと同様です。次に、ピボットと呼ぶある値 p を選び、「左半分は p 以下、続いて p の値、右半分は p より大きい」という状態にしてから、左半分と右半分をそれぞれ自分自身を再帰呼び出しして整列します。そうすると、「p 以下の整列された列」「p」「p より大きい整列された列」になり、整列が完了します (図 7.7)。

p としては「ちょうど列を半分ずつに分ける値」を使えるとベストですが、そんなものは分からないので適当に選ぶこととし、上では右端 (j 番目) の値を p にしています。

時間計算量はどうでしょうか。理想的な場合、つまり毎回列がおよそ半分ずつになるとすると、再帰呼び出しの深さは $\log_2 N$ になります (たとえば 16 なら 4 段、64 なら 6 段という感じ)。そして、各深さにおいて、その深さの呼び出しを全部合わせると、N 個のデータ全部をピボットと比較して振り分けることになります。これを合計すると、N 個のデータを振り分けることを $\log N$ 段繰り返しておこなうので、計算量は $O(N \log N)$ となります。

[1] 基数ソートでは b と c を a と同じサイズで取るので 3 倍と思うかもしれませんが、ケチるなら b と c を 1 つの配列にして「前から」と「後ろから」データを詰めてゆけばよいのです。

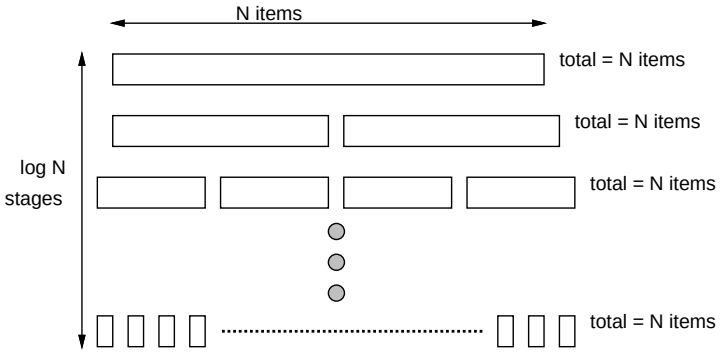


図 8.2: クイックソートのコード実行回数の検討

quicksort のコードが整列ずみの配列に対しては遅いという弱点を実際に示すには、次のように 2 回ずつ整列してみればよいでしょう。

```
def test
  a = randarray(1000)
  puts bench do quicksort(a) end; puts bench do quicksort(a) end
  a = randarray(2000)
  puts bench do quicksort(a) end; puts bench do quicksort(a) end
  a = randarray(3000)
  puts bench do quicksort(a) end; puts bench do quicksort(a) end
end
```

これを実行してみると、結果は次のようになりました。

データ数	1,000	2,000	3,000
1 回目	16	39	63
2 回目	984	3960	8859

1 回目は $O(N)$ に近い (実際には $O(N \log N)$ のはず) けれど、2 回目はずっと遅く $O(N^2)$ に近いようです。これを改良するにはどうしたらいいでしょう? それには、ピボット値を取る時にいつも「端っこ」から取っていたからまずいので、代わりにランダムに取るようにすればよいでしょう。次のコードは先のものに***の 1 行を追加しています。ここでは、i〜j の範囲の整数 p を 1 つランダムに選び、a[j] と a[p] を交換します。あとはこれまでと同様に処理するだけです。

```
def quicksort(a, i, j)
  if j <= i then
    # do nothing
  else
    p = i + rand(j-i+1); swap(a, p, j) # ***
    pivot = a[j]; s = i
    i.step(j-1) do |k|
      if a[k] <= pivot then swap(a, s, k); s = s + 1 end
    end
    swap(a, j, s); quicksort(a, i, s-1); quicksort(a, s+1, j)
   end
end
```

これの計測は上の表と異なり、1 回目も 2 回目もほぼ同じ時間で整列が終わります。

8.2 時間計算量ふたたび exam

ここまでで整列アルゴリズムを題材に時間計算量を考えて来ましたが、他のアルゴリズムについても考えましょう。時間計算量の求め方を簡単にまとめると、次のようになります。

> 入力の値 n に対して、プログラム中の「最も多く実行される箇所」の実行回数を求め、n の式で表し、$O(f(n))$ の形で記す。

極端な例ですが、n が出て来なければ $O(1)$(定数計算量) つまり n の値に関わらず一定時間で終わることを意味します。速い方から順に典型的なものを挙げておきます。

ア $O(1)$ — 定数時間

イ $O(\log n)$ — 対数

ウ $O(\sqrt{n})$ — 平方根

エ $O(n)$ — 線形計算量、n に比例

オ $O(n \log n)$ — よい整列アルゴリズム

カ $O(n^2)$、$O(n^3)$ — 一般に多項式計算量と呼ぶ

キ $O(2^n)$ — 指数計算量

ク $O(n!)$ — 階乗計算量

実際に動かす時の感覚としては、$O(n)$ までは「すごく速い」、$O(n \log n)$ は「まあまあ速い」、$O(n^2)$ は「遅い」、$O(2^n)$ は「ひどく遅くて小さい n しか役立たない」という感じです。

演習 1 以下に示すメソッドにさまざまな n を与えた際の時間計算量を見積もりなさい (上記ア〜クのどれに相当するかを選べという意味)。また、実際に bench で掛かる時間を計測して、選択が合っていたかを確認しなさい。今回使う bench のソースコードを示します。前回と違うのは「渡されたブロックを指定回数繰り返す (回数を指定しなければ 1 回なので前回と同じ)」点です。

```
def bench(count = 1)          ← count は指定しなければ 1
  t1 = Process.times.utime
  count.times do yield end    ← count 回繰り返し yield を実行
  t2 = Process.times.utime
  return t2-t1
end
```

bench の計測値はあまり時間が短いと誤差が大きいので、0.1〜1 秒くらいになるように回数を調節して計測してください。1 回あたりの所要時間は計測した時間を繰り返し回数で割ればよいです。

a. n^2 を計算するメソッドその 1

```
def square1(n)
  return n*n
end
```

b. n^2 を計算するメソッドその 2

```
def square2(n)
  result = 0
  n.times do result = result + n end
  return result
end
```

c. n^2 を計算するメソッドその 3

```
def square3(n)
  result = 0
  n.times do n.times do result = result + 1 end end
  return result
end
```

d. 1.0000000001^n を計算するメソッドその 1

```
def near1pow1(n)
  result = 1.0
  n.times do result = result * 1.0000000001 end
  return result
end
```

e. 1.0000000001^n を計算するメソッドその 2

```
def near1pow2(n)
  if n == 0
    return 1.0
  elsif n == 1
    return 1.0000000001
  elsif n % 2 > 0
    return near1pow2(n-1) * 1.0000000001
  else
    return near1pow2(n/2)**2
  end
end
```

f. 1.0000000001^n を計算するメソッドその 3 [2]

```
def near1pow3(n)
  return Math.exp(n*Math.log(1.0000000001))
end
```

g. 1〜3 の値が n 個並んだ全組み合わせを生成する (印刷は省略)

```
def nest3n(n) nest3(n, "") end
def nest3(n, s)
  if n <= 0 then
    # puts(s)
  else
    1.step(3) do |i| nest3(n-1, s + i.to_s) end
  end
end
```

h. 1〜n の値のすべての順列を生成する (印刷は省略)

```
def perm(n)
  a = Array.new(n) do |i| i+1 end
  perm1(a, [])
```

[2] Math.log は $\ln x$、Math.exp は e^x を計算するメソッドである。

```
end
def perm1(a, b)
  if a.length == b.length
    # p(b)
  else
    a.each_index do |i|
      if a[i] != nil
        x = a[i]; a[i] = nil; b.push(x)
        perm1(a, b)
        a[i] = x; b.pop
      end
    end
  end
end
```

8.3 既出アルゴリズムの別バージョン

8.3.1 最大公約数

以下では、これまでに出てきたアルゴリズムについて、計算量の違う別バージョンをお見せして、計測の題材にしていただきましょう。まず、前に出てきた最大公約数のプログラムは引き算を使っていましたが、代わりに剰余演算を使えば演算回数はずっと少なくなります (こちらが一般に**ユークリッドの互除法** (Euclid's algorithm) として知られているものです。もっとも、最初にユークリッドが考案したのは引き算を使う方だったそうですが)。

逆に、もっとベタなアルゴリズムとして、次のようなものも考えられます。

- gcd3(x, y) — x と y の最大公約数を求める
- 　i を $min(x, y)$ から 1 まで 1 ずつ減らしながら繰り返し、
- 　　x も y も i で割り切れるなら、i を返す。
- 　繰り返し終わり。

8.3.2 フィボナッチ数

やってみればすぐ分かりますが、再帰的定義そのままのフィボナッチ数の計算はすごく遅いです。別の方法として、たとえば `x0` と `x1` に 1 を入れておき、それからループで `x0` にはこれまでの `x1`、`x1` にはこれまでの `x0+x1` を入れることを繰り返して計算することが考えられます (図 8.3)。

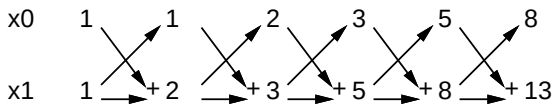


図 8.3: ループによるフィボナッチ数

もう 1 つ、こういうのはどうでしょうか。

$$\begin{pmatrix} x_{i+1} \\ x_i \end{pmatrix} = \begin{pmatrix} x_{i-1} + x_i \\ x_i \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x_i \\ x_{i-1} \end{pmatrix}$$

だから、$x_0 = x_1 = 1$ とおいてあとは上の漸化式で x_i を計算すれば i 番目のフィボナッチ数が求まります。漸化式といっても次々に同じ行列を掛けるだけですから、次の Q、v について $Q^n v$ を求めればよいのです。

$$Q = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix},\ v = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

そして、Q^n を求めるときに次の漸化式を活用すると、まじめに n 回行列の掛け算をやるよりも速く結果を求められます。[3]

$$Q^n = \begin{cases} E & (n = 0) \\ QQ^{n-1} & (n \text{ が奇数}) \\ (Q^{\frac{n}{2}})^2 & (n \text{ が偶数}) \end{cases}$$

8.3.3 組み合わせの数

組合せの数も再帰的定義そのままでは非常に遅いです。別の方法として、以前にやった掛け算を使う方法がまずあります。また、**パスカルの三角形** (Pascal's triangle) を作る方法があります (図 8.4)。

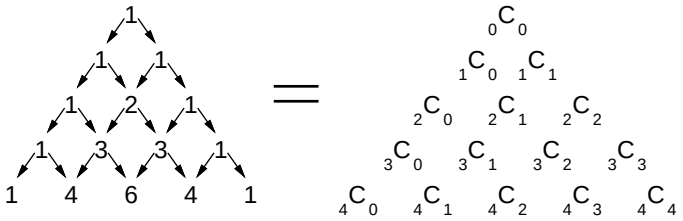


図 8.4: パスカルの三角形

演習 2 ここに挙げたものまたはそれ以外のものについて、計算量の異なる複数 (少なくとも 2 つ) のアルゴリズムを用いたプログラムを作成し、それらの答えが一致することを確認した上で、実行時間を比較しなさい。

8.4 乱数とランダムアルゴリズム

8.4.1 乱数とは

乱数 (random number) とは、簡単に言えば「ランダムな数」です。より正確に言うと、「ある分布に従う、互いに独立な事象を表す、確率変数の実現値の列」のことを乱数列 (randam sequence) と言い、その中の個々の値が乱数です。互いに独立ということは、ある点までの乱数列が分かったからといって、次の乱数がいくつであるかは予測できないことを意味します。

また、**分布** (distribution) とは、どの範囲の値がどのくらい出現しやすいかを表すものです (図 8.5)。たとえば、区間 $[0, 1)$ の一様分布 (uniform distribution) であれば、乱数の範囲は 0 以上 1 未満で、その間のどの数も同じくらいの確からしさで出現します。これを**一様乱数** (uniform random number) と言います。また、偏りのないサイコロを振って出る目の数は 1 以上 6 以下の整数値ですが、どの数も同じ確率で出現するため、これも一様乱数です。この他によく使われる乱数としては、正規分布 (normal distribution) に従う**正規乱数** (normal random numbers) があります。[4]

[3] 先の 1.0000000001^n の計算もこれと同じ方法のものがあります。

[4] 正規分布は中央が一番高く両側にすそを引いたツリガネ形の分布であり、試験の**偏差値** (standard score) などでおなじみです。試験が受験者の集団に対して易しすぎたり難しすぎたりヘンな問題だったりすると、分布が綺麗な正規分布でなくなるので偏差値による順位推定が役に立たなくて問題になったりします。

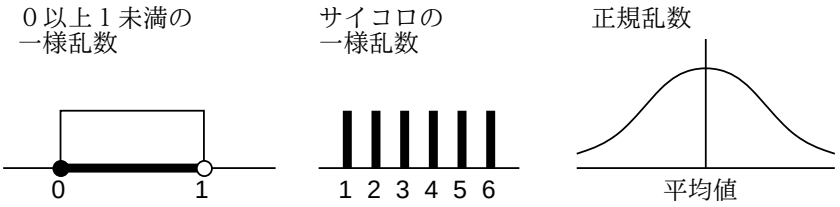


図 8.5: 乱数と分布

8.4.2 擬似乱数 exam

擬似乱数 (pseudorandom number) とは、プログラムで順次計算を行うことで生成される数値の列で、乱数列のように思えるものを言います。さんざん学んできたように、プログラムの動作は完全に決定的なものなので、「でたらめな」数を生成するのは思いのほか難しいものです。

擬似乱数のアルゴリズムの代表的なものとして、次のものがあります。

- **自乗採中法** (middle-square method) — w ビットの数を 2 乗すると $2w$ ビットになるので、そこから中央付近の w ビットを取り出して「次の数」とする。これを繰り返すことで w ビットの乱数列を生成する。[5]
- **線形合同法** (linear congruential method)— $x_{i+1} = (x_i \times a + c)\ mod\ m$ により次々に値を生成していく。[6]
- **メルセンヌツイスター** (Mersenne twister)、MT — 1997 年に松本　眞、西村拓士が開発した乱数アルゴリズム。

どのようなアルゴリズムでも、計算方法が決まっている以上、順次 x_i を生成していくうちに前に出てきたものが再度現れたら、それ以降の列は前と同じものの繰り返しになります。これを**周期** (period) といい、もちろん周期が長いものが望まれます。MT は 32 ビットで $2^{19937}-1$ という長い周期を保証できるという性質を持つという点で画期的でした。さらに周期の他に統計的独立性の検定などもクリアしています。

Ruby では、`rand(`N`)` で 0 以上 N 未満の整数の一様乱数、引数なしの `rand` で区間 $[0,1)$ の実数一様乱数が得られます。[7]

このほか、最近では**オペレーティングシステム** (operating system, OS) が乱数機能を提供してくれている場合があります。OS の乱数機能は、ユーザのキーボード入力やディスクの動作などの外部割り込みに基づいた「ランダムさ」を活用するので、擬似乱数のような周期の問題がありませんが、速い速度で乱数を生成消費すると「ランダムさ」の供給が追い付かなくなることがあります。また、最近では CPU チップ自体に物理的な乱数発生装置を持つものもあります。

8.4.3 ランダムアルゴリズム exam

ランダムアルゴリズム (randomized algorithm) とは、(擬似) 乱数を活用して、ランダムな振舞いを持たせたアルゴリズムを言います。これに対し、通常の決定的な動作を行うアルゴリズムは**決定的アルゴリズム** (deterministic algorithm) と呼ばれます。

ランダムアルゴリズムは、設計によっては決定的アルゴリズムよりすぐれた性能を持たせることができます。たとえば、1 億要素の配列があるとして、「その半分の 5 千万要素には値 a が入っているが、それがどことどこか所は分からない」場合と「まったく値 a が入っていない」場合とがあり、そのどちらであるかを判断する必要があるものとしましょう。

[5]自乗採中法は古くからあるアルゴリズムですが、あまりよい乱数列は生成できないことが知られています。

[6]線形合同法は MT の発明以前は主流のアルゴリズムでした。ただし、よい擬似乱数とするためには、パラメタ a, c, m の選定に注意が必要です。

[7]Ruby の `rand` の乱数アルゴリズムにも MT が使われています (バージョン 1.8 以降)。

決定的アルゴリズムでは、どのような調べ方をしてもその「裏をかかれる」可能性があって半分は調べなければ確実な判断ができません。たとえば先頭から順番に値 a があるかどうか見ていくとすると、値 a が全部後半に詰まっているかもしれないので、最悪で 5 千万要素を見る必要があります。では後ろから順に見ればいいかというと、値 a が全部前半に詰まっているかもしれないので同じことです。1 つおきでも何でも同様です。

ここで、乱数を用いて 1 億の位置からランダムに 1 つ選び、その値が a かどうかを判断することを 1 万回繰り返したとしましょう。その結果 1 回も値 a に遭遇しなければ、「値 a はない」と判断してまず問題ありません。というのは、この判断が間違っている確率は $\frac{1}{2^{10000}}$ であり、それはこの計算をするコンピュータが故障する確率よりはるかに小さいからです。

このような、微小だが 0 でない「間違う」確率を持ったアルゴリズムを**モンテカルロアルゴリズム** (Monte Carlo algorithm) と言います。[8] これに対し、間違うことはないが運が悪い場合に性能が低下するアルゴリズムを**ラスベガスアルゴリズム** (Las Vegas algorithm) と言います。[9]

たとえば、クイックソートでどの要素をピボットとして用いるかを乱数で決めるようにすると、これはラスベガスアルゴリズムとなります。なぜなら、乱数がすべて「悪い要素 (その区間の最大や最小の要素)」を選び続ける確率は非常に小さいので、よほど運が悪くない限り高速に整列が行え、そして運が悪い場合は実行時間が長く掛かることになるものの、整列そのものはやはり正しく行えるからです。

8.4.4 モンテカルロ法 exam

モンテカルロ法 (Monte Carlo method) とは、シミュレーションや数値計算などにおいて乱数を活用する手法を言います。

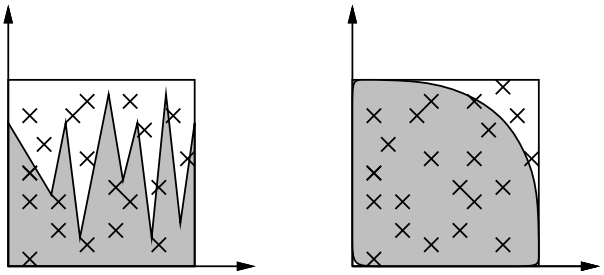


図 8.6: モンテカルロ法による数値積分

ここではモンテカルロ法を用いた数値積分を見てみましょう (図 8.6)。具体的には、積分する範囲の関数値の最大より大きい値を選んで長方形の領域を考え、その範囲内に乱数で多数の点を打ち、関数値より下にある点の比率を求めます。積分とは「その関数の下側の面積を求める」ことですから、長方形の面積にその比率を掛けたものが積分値 (の近似値) として使えます。

そんな面倒なことをするよりシンプソンのアルゴリズムでよいのでは? しかし、シンプソンのアルゴリズムなどは、対象とする関数が連続かつ微分可能 (なめらか) でないと使えません。そのような性質が期待できないような分野では、モンテカルロ法が有力な手法の 1 つとなるのです。

たとえば半径 1 の四分の一円の面積を求めて (それを 4 倍することで) π の近似値を計算してみましょう。[10]

```
def pirandom(n)
  count = 0
  n.times do
```

[8] モンテカルロはヨーロッパにあるカジノで有名な都市の名前です。
[9] ラスベガスは米国にあるカジノで有名な都市の名前です。
[10] もちろん円周は十分連続かつ微分可能ですが、それは置いておいて。

```
    x = rand(); y = rand()
    if x**2 + y**2 < 1.0 then count = count + 1 end
  end
  return 4.0 * count / n
end
```

実行させてみると次のとおり。

```
irb> pirandom 10000
=> 3.13
irb> pirandom 100000
=> 3.14444
irb> pirandom 1000000
=> 3.141604
```

有効数字 3〜4 桁では使えない、と思いますか? 実際には、3〜4 桁の有効数字が得られれば十分な場合は結構あります。たとえば、来年の GDP の成長率が 5.11 だろうと 5.12 だろうと、3 桁目はさして重要ではないでしょう?

演習 3 モンテカルロ法で数値積分を行うときの、精度 (有効桁数) と試行の数との関係について考察せよ。円周率の例題を活用してもよいが、できれば別の関数を積分するプログラムを作って検討することが望ましい。

8.4.5 乱数によるシミュレーション exam

シミュレーション (simulation) とは、様々な事項を実際に実験して見るかわりにコンピュータの上で計算のみにより行なってみて結果を調べる手法で、今日では広く使われています。たとえば、交通の流れを実際に観察する代わりに、乱数を用いてランダムに車を (プログラム内で) 発生させ、それらの車がどのように流れていくかを見ることで、さまざまな方法で交通信号を制御した場合の状況を簡単に試すことができ、それに基づいてよい制御方式を出すことができます。

簡単な例として「コインを投げて表だったら 1 円もらうのを 10 回やる」とします。乱数で「半々」を作り出せばいいわけです。たとえば実数乱数を使って「`if rand < 0.5 ...`」でもいいですが、ここでは 0 または 1 が出て来る整数乱数「`rand(2)`」を使うとそのまま加算できて便利そうです。

```
def toss10
  sum = 0
  10.times do sum = sum + rand(2) end
  return sum
end
```

動かしてみましょう。

```
irb> toss10
=> 3
irb> toss10
=> 8
```

なるほど。では、たとえば「ちょうど 3 円」もうかる確率はどれくらいでしょうか? もちろん数学が得意ならちょっと計算すればいいのですが、シミュレーションでもできます。1000 回やってもうかった金額がいくらだったかを集計してみます。

```
def toss10hist
  a = Array.new(11, 0)
  1000.times do n = toss10; a[n] = a[n] + 1 end
  return a
end
```

なぜ配列サイズが 11 かというと「0」の場合から「10」の場合まであるからですね。では実行します。

```
irb> toss10hist
=> [1, 11, 37, 148, 194, 240, 187, 128, 39, 15, 0]
irb> toss10hist
=> [3, 12, 48, 109, 219, 240, 203, 115, 44, 7, 0]
irb> toss10hist
=> [2, 11, 43, 106, 225, 255, 183, 125, 43, 7, 0]
```

こうして見ると「ちょうど 3 円」は $\frac{100}{1000} \sim \frac{150}{1000}$ くらいの確率、ということになるでしょうか。もちろん 2 項分布ですので、普通に計算でもできます。

$$ {}_{10}C_3 \cdot (0.5)^3 \cdot (1-0.5)^7 = \frac{120}{1024} $$

ですから、まあ合っているということですね。

解析的にすぐ求まるのだと面白くないですから、今度は「すごろく」をやってみましょう。40 ますのすごろくで、40 ます目が「あがり」とします。そして 10、20、30 のますに止まったら「振り出しに戻る」ものとします (これでこそすごろくです)。何回であがるでしょうか?

```
def sugo40
  pos = 0; count = 0
  while pos < 40 do
    n = rand(6) + 1; pos = pos + n; count = count + 1
    if pos < 40 && pos % 10 == 0 then pos = 0 end
  end
  return count
end
```

位置とサイコロを振った回数を 0 にし、位置があがり (40) より小さい間繰り返します。`rand(6)` で 0〜5 の一様乱数ができ、それに 1 を足すことでサイコロの 1〜6 にします。サイコロにしたがって位置を進め、サイコロを振った回数を増やします。そして、位置があがりでなく 10 の倍数なら振り出しなので位置を 0 にします。あがったらループを抜けるので、振った回数を返します。やってみましょう。

```
irb> sugo40
=> 28
irb> sugo40
=> 13
irb> sugo40
=> 18
```

かなり運に左右されるようですね (当然ですが)。では分布を調べてみましょう。アルゴリズムは先のとほぼ同じですが、ただしこんどは「最大何回」と分からないので、配列を必要に応じて増やすようにしています。

```
def sugo40hist
  a = []
  1000.times do
    n = sugo40
    while a.length < n + 1 do a.push(0) end
    a[n] = a[n] + 1
  end
  return a
end
```

ではやってみましょう。

```
irb> sugo40hist
=> [0, 0, 0, 0, 0, 0, 0, 0, 2, 24, 67, 99, 77, 67, 60, 46, 52,
39, 36, 43, 38, 25, 35, 16, 19, 22, 18, 16, 17, 18, 7, 15, 12,
6, 4, 12, 12, 4, 11, 10, 6, 6, 4, 7, 3, 3, 6, 2, 6, 4, 3, 2, 0,
2, 0, 3, 1, 2, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0,
0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
```

なるほど、運が悪いときは相当悪いようです…

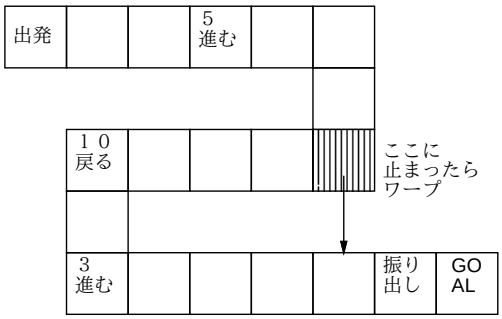
演習 4 次のようなシミュレーションをやってみよ。

a. サイコロを 2 個振った目の合計の分布を調べる。

b. 60%の確率で表がでるイカサマコインで「10 回投げて表が出た回数の金額だけもらう」場合の金額の分布を調べる。

c. サイコロを 3 個振ってうち 2 個が同じ目 (もう 1 個は違う目) である確率がどれくらいか調べる。

d. 次のようなすごろくをあがるのにサイコロを何回振るか分布を調べる。



e. その他自分の好きなシミュレーションを行なえ。

8.4.6 配列のシャッフル

ゲームやシミュレーションを行なうときに「独立にランダムな値を次々取る」なら乱数そのままでよいのですが、「値の集まりが決っていて、そこからランダムな順で取る」ことも多くあります。たとえばカードを**シャッフル** (shuffle) してから順に取って行く、みたいなものですね。

これをする場合は、値の配列があるとして、最初はそこからランダムに 1 つ取るとして、次はさっき出たものは除外する必要があります。取り出すのは同じままで、ただし出たものを覚えておいて同じなら「やり直す」のでいいのでは、と思うかも知れませんが、そうすると取り進むにつれてどんどん「やり直し」だらけになって遅くなります。

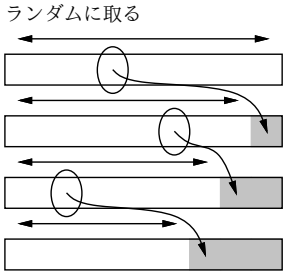


図 8.7: シャッフルのアルゴリズム

そうではなく、選択ソートでやったように、1 つランダムに取るごとに、その取ったものを「配列の端に置いて (実際にはそこにあった値と交換する)」ランダムに取る範囲を狭めて行けばよいのです (図 8.7)。このやりかたなら、最初に全部そのようにしてランダムな列を作ってしまい、そのあとは端から順に使えばちゃんとランダムに 1 つずつ取れます。これがシャッフルの標準的な方法です。コードは次の通り (乱数の範囲が 0〜N なので選択ソートとは逆に配列の後ろから順に詰めていきます)。

```
def swap(a, i, j)
  x = a[i]; a[i] = a[j]; a[j] = x
end
def shuffle(a)
  (a.length-1).step(1, -1) do |i| swap(a, i, rand(i+1)) end
end
```

やってみましょう。

```
irb> a = [1,2,3,4,5]; suffle(a); a
=> [3, 4, 2, 5, 1]
```

8.4.7 乱数とゲーム

最後にお楽しみの話題としてゲームに言及しておきましょう。ゲームの中には、将棋や囲碁のように (先手後手を決める以外は) ランダム性を使わないものもありますが、多くのゲームでは (サイコロやカードのシャッフルなどを通じて) ランダム性を採り入れています。これは、ランダム性を採り入れることで、ゲームの「場面」が毎回違ったものになり新鮮さが保たれ、また複数プレーヤで行う場合に「上手下手」以外の要因が入って下手な人にも勝つチャンスが生まれ、勝負の行方に興味が持てるようになるからです。

簡単なゲームとして「数当て」を作ってみます。そのルールは次のとおり。

- プログラムは内部で 4 桁の数を「思い浮かべ」る (4 桁の中に重複はない。また 0 もない)。
- プレーヤはその 4 桁の数を「当てる」ことをめざして、自分も 4 桁の数を入力する。
- プログラムは 2 つの 4 桁の数を照合して、「同じ位置に同じ数がある (これをヒットと呼ぶ)」個数と、「同じ数があるがただし違う位置にある (これをブローと呼ぶ) 個数とを数えて知らせる。
- プレーヤはその情報を見て再度チャレンジする。
- 10 回以内のチャレンジで当たればプレーヤの勝ち、さもなければプレーヤの負けとする。

Ruby プログラムを次に示します。最初の行で 4 桁のランダムな数 (実際には 4 文字の文字列) を作っています。文字列も配列と同じに文字単位で (添字を指定して) アクセスできるので、まず `a` に `"123456789"` を入れてから先の `shuffle` を呼んでシャッフルし、それから先頭の 4 文字を `a[0..3]` で取り出しています。

```
def kazuate
  a = "123456789"; shuffle(a); a = a[0..3]; count = 0
  while true do
    print("your guess? ")
    s = gets; hit = 0; blow = 0
    4.times do |i|
      4.times do |j|
        if s[i] == a[j] then
          if i == j then hit +=  1 else blow += 1 end
        end
      end
    end
    if hit == 4 then puts "you win!"; return end
    count += 1
    if count > 9 then puts "you lose! answer = #{a}."; return end
    printf("hit = %d, blow = %d.\n", hit, blow)
  end
end
```

本体ループではゲームの「やりとり」を行うために、キーボードから 1 行入力するメソッド gets を使っています。また、puts は改行してしまうので、プロンプトを出力するのに print を使いました。ヒット/ブローの計算はシャッフルの時と同様、文字列の添字参照を使っています。

演習 5 上の例題プログラムを打ち込んで遊んでみよ。納得したら、乱数を使ったゲームを何か作ってみよ。上の例題の改良 (改造) 版でもよい。

本日の課題 8A

「演習 3」または「演習 4」から 1 つ以上選び、動かしたプログラムを含むレポートを提出しなさい。プログラムの簡単な説明が含まれること。アンケートの回答もおこなうこと。

Q1. 乱数を使ったアルゴリズムの利点を納得しましたか。

Q2. シミュレーションは書けるようになりそうですか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 8B

「演習 1」〜「演習 5」の (小) 課題から選択して 1 つ以上プログラムを作り、レポートを提出しなさい。プログラムと、課題に対する報告・考察 (やってみた結果・そこから分かったことの記述) が含まれること。アンケートの回答もおこなうこと。

Q1. 乱数を使ったアルゴリズムを自分なりにどのように考えますか。

Q2. シミュレーションを構成するときのコツは何だと思いますか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

＃9　オブジェクト指向

今回は現在のプログラミングにおいて重要な概念となっているオブジェクト指向です。

- オブジェクト指向の考え方について知る。
- クラス方式のオブジェクト指向言語による記述を学ぶ。

9.1　前回の演習問題解説

9.1.1　演習 1 — さまざまなメソッドの計算量

これは簡単に答えだけ書きましょう。

a: $O(1)$, b: $O(n)$, c: $O(n^2)$, d: $O(n)$, e: $O(\log n)$, f: $O(1)$, g: $O(2^n)$, h: $O(n!)$

9.1.2　演習 2a — 最大公約数

2 つの数 $M, N (M < N$ とする) の最大公約数のベタなバージョン (M からカウンタを 1 ずつ減らしてゆき、それで両者が割り切れることをチェックする方法) は当然 $O(M)$ になります。

```
def gcdenumerate(x, y)
  min = x; if min > y then min = y end
  (min-1).step(1, -1) do |i|
    if x % i == 0 && y % i == 0 then return i end
  end
end
```

では「大きい方から小さい方を引いてゆく」バージョンはどうなのでしょうか?

```
def gcd(x, y)
  while x != y do
    if x > y
      x = x - y
    else
      y = y - x
    end
  end
  return x
end
```

最善の場合は $M = N$ の時で、すぐ終わります。最悪の場合は $M = 1$ のときで、$N - 1$ 回引き算をしないと終わりません。平均は…? そこで、乱数を使って実験してみました。

```
bench(10000) do gcd(rand(100)+1, rand(100)+1) end → 0.0859375
bench(10000) do gcd(rand(1000)+1, rand(1000)+1) end → 0.1640625
bench(10000) do gcd(rand(10000)+1, rand(10000)+1) end → 0.2734375
bench(10000) do gcd(rand(100000)+1, rand(100000)+1) end → 0.4453125
```

これを見ると、値が 10 倍ずつ大きくなる時に一定くらいずつ (やや多いですが) 値が増えて行きます。引き算ごとに M や N が一定比率くらいで減少して行くと考えれば $O(\log M)$ ということになるわけです。しかし M と N の比率が違うとどうでしょうか。

```
bench(10000) do gcd(rand(100)+1, rand(100)+1) end → 0.203125
bench(10000) do gcd(rand(1000)+1, rand(100)+1) end → 1.328125
bench(10000) do gcd(rand(10000)+1, rand(100)+1) end → 12.1171875
bench(10000) do gcd(rand(100000)+1, rand(100)+1) end → 130.546875
```

M の方が大きいとして、M が 10 倍になると時間も 10 倍になります (これは、M が N の 1000 倍なら 1000 回引く必要があるわけですから当り前ですね)。そうすると、計算量は全体として $O(M \log M)$ ということになるでしょうか。

では、引き算の代わりに剰余演算を使うユークリッドの互除法ではどうでしょう?

```
def gcd3(x, y)
  while true do
    if x > y
      x = x % y; if x == 0 then return y end
    else
      y = y % x; if y == 0 then return x end
    end
  end
end
```

これでまずアンバランスな方から測ってみましょう。

```
bench(10000) do gcd3(rand(100)+1, rand(100)+1) end → 0.0390625
bench(10000) do gcd3(rand(1000)+1, rand(100)+1) end → 0.046875
bench(10000) do gcd3(rand(10000)+1, rand(100)+1) end → 0.046875
bench(10000) do gcd3(rand(100000)+1, rand(100)+1) end → 0.0390625
```

まったく変わりませんね。それは、CPU の割算命令の時間は値が変わってもほとんど一定時間で実行されるからです (ただし Ruby で多倍長演算が必要なくらい大きい値になるとそれは 1 命令ではできなくなるのでこのようには行きません)。では値の大きさの影響はどうでしょうか?

```
bench(10000) do gcd3(rand(100)+1, rand(100)+1) end → 0.0390625
bench(10000) do gcd3(rand(1000)+1, rand(1000)+1) end → 0.0546875
bench(10000) do gcd3(rand(10000)+1, rand(10000)+1) end → 0.0625
bench(10000) do gcd3(rand(100000)+1, rand(100000)+1) end → 0.078125
bench(10000) do gcd3(rand(1000000)+1, rand(1000000)+1) end → 0.0859375
```

こちらは 10 倍になるごとにおよそ一定ずつ増えてゆくようです。それは先と同じで、ループを 1 回まわるごとにだいたい一定比率で 2 つの値が小さくなるからでしょう。これを総合すると、このアルゴリズムの時間計算量は $O(\log M)$ ということになります。[1]

9.1.3　演習 2b — フィボナッチ数

再帰的定義そのままの**フィボナッチ数**の計算は、`fib`(N) の計算に `fib`($N-1$) と `fib`($N-2$) を実行し、それらが `fib`($N-2$) と `fib`($N-3$)、`fib`($N-3$) と `fib`($N-4$) を呼ぶというふうに「倍々」に

[1] 数が大きくなり 1 語に入らなくなると、除算の実行が一定時間と見なせなくなります。その場合、除算の計算に数の桁数に比例する時間、すなわち $O(\log M)$ を要するので、全体の時間計算量は $O(\log^2 M)$ となります。

なるので、時間計算量は $O(2^N)$ になります (指数時間)。これに対し、ループで計算する場合は $O(N)$ になります。

```
def fibloop(n)
  x0 = 1; x1 = 1
  (n-1).times do
    t = x0 + x1; x0 = x1; x1 = t
  end
  return x1
end

bench(10000) do fibloop(10) end → 0.0625
bench(10000) do fibloop(100) end → 0.8359375
bench(10000) do fibloop(1000) end → 11.9140625
```

「10 倍になるごとに時間も 10 倍」から外れますが、これは `fib(50)` くらいから多倍長計算が必要になるためでしょう。

では、行列計算で N 乗の計算を工夫する方法はどうでしょう。「工夫」の漸化式を再掲します。

$$Q^n = \begin{cases} E & (n=0) \\ QQ^{n-1} & (n \text{ が正の奇数}) \\ (Q^{\frac{n}{2}})^2 & (n \text{ が正の偶数}) \end{cases}$$

これを用いるプログラムは次のとおり。[2]

```
def mat22multvec(a, v)
  c = [0, 0]   # 結果用の 1 次元配列
  c[0] = a[0][0]*v[0] + a[0][1]*v[1]
  c[1] = a[1][0]*v[0] + a[1][1]*v[1]
  return c
end
def mat22mult(a, b)
  c = [[0,0],[0,0]]   # 結果用の 2 次元配列
  c[0][0] = a[0][0]*b[0][0] + a[0][1]*b[1][0]
  c[0][1] = a[0][0]*b[0][1] + a[0][1]*b[1][1]
  c[1][0] = a[1][0]*b[0][0] + a[1][1]*b[1][0]
  c[1][1] = a[1][0]*b[0][1] + a[1][1]*b[1][1]
  return c
end
def mat22power(a, n)
  if n < 0
    return [[1,0],[0,1]]  # 2x2 単位行列
  elsif n % 2 == 1
    return mat22mult(mat22power(a, n-1), a)
  else
    b = mat22power(a, n/2); return mat22mult(b, b)
  end
```

[2] 2×2 行列の積、行列とベクトルとの積を下請けに用意して、それを用いて上の漸化式を使った N 乗を定義し、最後にそれを用いてフィボナッチ数を計算しています。

```
end
def fibmat(n)
  return mat22multvec(mat22power([[1,1],[1,0]], n-1), [1,1])[0]
end
```

実験してみましょう。

```
irb(main):101:0> bench(10000) do fibmat(10) end → 0.703125
irb(main):102:0> bench(10000) do fibmat(100) end → 1.46875
irb(main):103:0> bench(10000) do fibmat(1000) end → 2.921875
```

10 倍ごとにほぼ一定ずつ時間が増えています (最後のほうは多倍長になるため外れています)。

計算量はどうでしょうか。「正の奇数」が選ばれるのは N を 2 進表現した時に「1」が現れる回数と等しくなり、「正の偶数」が選ばれるのは N を (奇数なら 1 を引きながら) 半分ずつにしていき 0 になるまでやるので、2 進表現の桁数つまり $\log N$ が回数となります。上記「1」の数は平均すると桁数の半分くらいだから $\frac{\log N}{2}$ となります。これらを合わせると、全体として $O(\log N)$ となります。

9.1.4 演習 2c — 組み合わせの数

再帰的定義はフィボナッチと同様、倍々の呼び出しのため $O(2^N)$ です。「パスカルの三角形」の場合はどうでしょうか。図 9.1 を N 段目まで作るとなると、要素数が $\frac{N(N+1)}{2}$ ありますから、計算量としては $O(N^2)$ ということになるはずです。コードを書いてみると次のようになります。

```
def combarray(n, r)
  a = Array.new(n+1, 1)
  1.step(n) do |i|
    (i-1).step(1, -1) do |k| a[k] = a[k-1] + a[k] end
    # p(a)
  end
  return a[r]
end
```

これは、N 段までのパスカルの三角形を作るために、要素数 $N+1$ の「1」ばかりが詰まった配列を用意し、それを図 9.1 のように隣の要素どうし足すことを繰り返していくものです。内側ループを添字が大きい方から順に処理しているのは、そうしないと「1 つ前の値」を使うことができないからです。

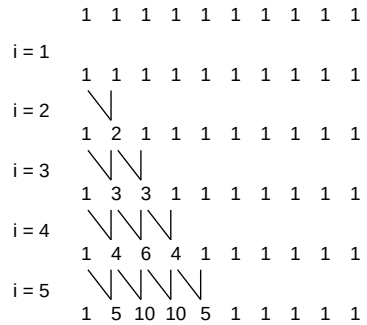


図 9.1: パスカルの三角形の計算

では時間を計測してみます。

```
bench(10000) do combarray(10,5) end → 0.609375
bench(10000) do combarray(20,10) end → 2.2578125
bench(10000) do combarray(30,15) end → 4.9765625
```

確かに $O(N^2)$ のようです。ところで、前にやった「普通に掛け算する」バージョンはどうでしょうか?

```
def combloop(n, r)
  result = 1
  1.step(r) do |i|
    result = result * (n - r + i) / i
  end
  return result
end
```

これならループが r 回だから ($r \sim N$ として) $O(N)$ となります。

```
bench(10000) do combloop(10,5) end → 0.0703125
bench(10000) do combloop(20,10) end → 0.1171875
bench(10000) do combloop(30,15) end → 0.2265625
```

たしかにずっと速いので、結局、ループで普通に計算するのがよいというオチでした。ただし、「何回もさまざな値を」使うのであれば、パスカルの三角形を「2 次元配列」の上で作成しておいて、そこから値を取り出すようにするのが良さそうです。

9.1.5 演習 3 — モンテカルロ法の誤差

関数 $y = x$ の区間 $[0, 1)$ における積分を求めてみましょう。答えは両辺が 1 の直角 2 等辺三角形の面積ですから、0.5 であることはすぐ分かります。プログラムは次のとおり。

```
def integrandom(n)
  count = 0
  n.times do
    x = rand(); y = rand()
    if y < x then count = count + 1 end
  end
  return count / n.to_f
end

irb> integrandom 100
=> 0.55              ←誤差 0.05
irb> integrandom 1000
=> 0.475             ←誤差 0.025
irb> integrandom 10000
=> 0.4933            ←誤差 0.0067
irb> integrandom 100000
=> 0.50127           ←誤差 0.00127
irb> integrandom 1000000
=> 0.500674          ←誤差 0.000674
```

試行数が 100 倍になると誤差が $\frac{1}{10}$ になるように見えます。これはなぜでしょうか。

なぜこの方法で面積が求まるのかに立ち帰って考えて見ます。このプログラムでは 1 回の**試行** (trial — サイコロを振ること) で得られるのは「打った点が関数 f の上か下か」つまり「0 か 1 か」の情報です。そして上であれば count は増やさず (つまり 0 を足し)、下であれば 1 を足し、最後に N で割るので、この「0 か 1 か」の確率変数の平均を求めています。この確率変数が 1 である確率は関数の面積と等しいので、N を増やしていけば**大数の法則** (law of large number) により、観測される平均

値は理論的平均値 (この場合は関数の面積) に近づいていきます。そして「どれくらい近づく」かは**中心極限定理** (central limit theorem) が教えてくれます。観測される平均値を $\overline{X}$、真の平均を μ とすると、次の式は $N(0,1)$ つまり平均 0、分散 1 の正規分布に収束します。

$$\sqrt{N}(\overline{X} - \mu)$$

言い換えれば、誤差を $\sqrt{N}$ 倍したものが同じ分布なのですから、試行数を N 倍にすると誤差は $\frac{1}{\sqrt{N}}$ 倍になるわけです。これは上の結果と合致しています。

9.1.6　演習 4

シミュレーションやその結果の分布を調べるのは設定通りに書けばよいだけです。ここでは簡単な最初の 3 つをやりましょう。まずサイコロ 2 つを投げるもの。

```
def twodicehist
  a = Array.new(13, 0)
  1000.times do
    n = 1+rand(6) + 1+rand(6); a[n] += 1
  end
  return a
end

irb> twodicehist
=> [0, 0, 21, 47, 75, 101, 125, 195, 145, 114, 87, 67, 23]
```

つぎに「60%が表のコイン」。今度は半々でないので、実数を返すように `rand` を呼び出し、0.6 未満なら表とします。1000 回やると、やっぱり「6 円もらえる」のが最も多くなります (当然)。

```
def unfaircoin10
  sum = 0
  10.times do
    if rand < 0.6 then sum += 1 end
  end
  return sum
end
def unfaircoinhist
  a = Array.new(11, 0)
  1000.times do n = unfaircoin10; a[n] += 1 end
  return a
end

irb> unfaircoinhist
=> [0, 0, 11, 41, 112, 171, 261, 224, 128, 50, 2]
```

最後の「サイコロ 3 個振ってちょうど 2 つ同じ」ですが、その条件をチェックするのが面倒なだけですね。

```
def threediceequal
  sum = 0
  1000.times do
    a = 1+rand(6); b = 1+rand(6); c = 1+rand(6);
```

```
      if a == b
        if b != c then sum += 1 end
      elsif b == c || a == c
        sum += 1
      end
    end
    return sum
  end

  irb> threediceequal
  => 399
```

期待値はどれくらいでしょうか。a のダイスがどれであるにしろ、b がそれと等しく ($\frac{1}{6}$ の確率) かつ c はそれ以外 ($\frac{5}{6}$ の確率) であるか、または a と b が等しくなく ($\frac{5}{6}$ の確率) かつ c は a か b いずれかと等しい ($\frac{2}{6}$ の確率) ですね。計算してみると上の結果とだいたい合っているようです。

$$\frac{1}{6} \times \frac{5}{6} + \frac{5}{6} \times \frac{2}{6} = \frac{15}{36} = 0.417$$

9.2 オブジェクト指向

9.2.1 オブジェクト指向とは

これまで、配列やレコード等のデータ構造を作り、それを操作するメソッドを組み合わせてアルゴリズムを実現する、という形のプログラムを作ってきました (図 9.2 左)。このようなモデルを**手続き型計算モデル**、このモデルに基づくプログラミング言語を**手続き型言語**と呼ぶのでした。

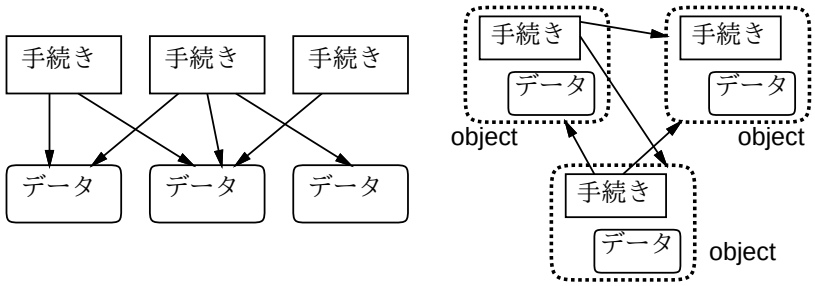


図 9.2: 手続き型モデルとオブジェクト指向

このプログラミングスタイルは長い間主流として使われてきましたが、近年のようにプログラムが大きくなり複雑化してくると、次のような弱点が問題になってきました。

- データ構造と手続きが分離していて、両者の対応が取りにくい。
- 手続きが複雑になると、どの部分が何を行っているかの把握が困難になる。
- 各データ構造がどの手続きからでも (原理的には) アクセスできるため、本来アクセスすべきでないデータ構造に触ってしまうことによるトラブルが起きやすい。

オブジェクト指向 object-orientation は、上記の点を克服すべく手続き型モデルを拡張した概念で (図 9.2 右)、プログラムが扱う対象を多様な**もの**、ないし**オブジェクト** (object) として捉える考え方です。

我々が日常扱っている「もの」にはそれぞれ固有の機能や特性があり、我々は「内部構造」には関わらなくてもこれらの「もの」の機能や特性を活用できます。たとえばイスであれば「座る」「高さを調節」「移動させる」などの操作ができますし、ペンであれば「キャップをつける/外す」「描く」な

どの操作ができ、それぞれ固有の色などもあります。しかし、これらを利用したり参照するのに、イスやペンの内部構造を理解している必要はありません。プログラミングもこれと同様にできれば人間にとってずっと扱いやすくなる、というのがオブジェクト指向の基本的なアイデアです。

今日では多くのプログラミング言語がオブジェクト指向を取り入れています。それらの言語 (オブジェクト指向言語) では、手続きとそれが扱うデータが組になるため対応がつけ易く、個々の手続きは自分の担当するデータのみを直接扱うため簡潔に保ちやすく、データは対応する手続き以外からは操作されないため、不用意に壊される心配が減ります。データを外部から直接アクセスされないようにすることを**カプセル化** (encapsulation) ないし**情報隠蔽** (information hiding) と呼びます。

9.2.2　クラスとインスタンス exam

前節で述べたような「もの」を言語上でどのように表すかを考えてみましょう。ものには種類ないし**クラス** (class) があるものと考え、その種類ごとに「どんな性質を持つか」「どのような操作ができるか」を定義していく、というのが 1 つの方法です。このような考え方に従うオブジェクト指向言語を**クラス方式** (class based) のオブジェクト指向言語と呼びます。Ruby、Java、C++などの言語はクラス方式のオブジェクト指向言語です。[3]

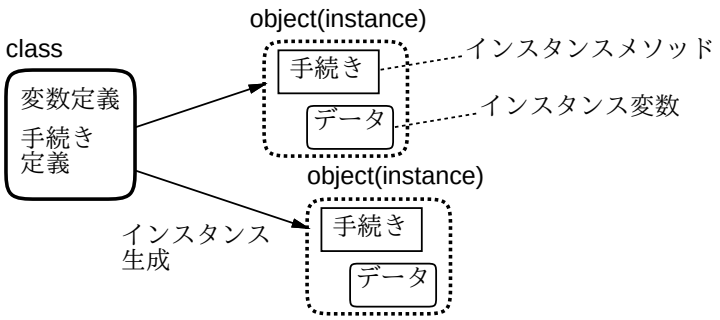


図 9.3: クラスからインスタンスを生成

では、ものの種類の定義、つまりクラス定義 (class definition) には何が含まれるべきでしょうか (図 9.3)。上述のように、それぞれの「もの」には固有のデータと固有の操作があるので、それを**変数**と**手続き**ないし**メソッド**で表すのが自然な方法です。これらをそれぞれ、**インスタンス変数** (instance variable)、**インスタンスメソッド** (instance method) と呼びます。

ただし、ここで言う変数とメソッドは、これまで使ってきた変数やメソッドとは少し違っています。つまり、あるクラスを定義し、それをもとに「そのクラスに所属するもの」 — オブジェクト指向言語の言葉で言えば**インスタンス** (instance — 実体と呼ぶこともあります) を生成したとすると、クラス内で定義した変数やメソッドは「そのクラスのインスタンスに付随した」ものとなります。

たとえば図 9.4 では、クラス定義 `Xyz` を「ひな型」として 2 つのインスタンスを生成し、変数 `x1` と `x2` に入れています。この時、この 2 つのインスタンスは内部に持っているインスタンス変数群も使用できるメソッド群も同じですが、インスタンスとしては別個、つまりインスタンス変数群はそれぞれ別個になっています。つまりクラス定義に書かれているとおりのインスタンス変数群とインスタンスメソッド群を持つということです。

インスタンスメソッドを呼び出す時は、メソッド名だけでは「どのインスタンスに付随する」メソッドか特定できないので、インスタンス x に対して「x. メソッド名」の形で指定します。これを**メッセージ送信記法** (message sending) と呼びます。この記法は、既に配列などさまざまな (Ruby が提供してくれている) オブジェクトの機能を呼び出す時に用いてきました。

[3]なお、別の方式として**プロトタイプ方式** (prototype based) のオブジェクト指向言語があります。これは、「お手本」となるオブジェクトを (概念的に) コピーして類似したオブジェクトを用意する方式で、JavaScript などが採用しています。

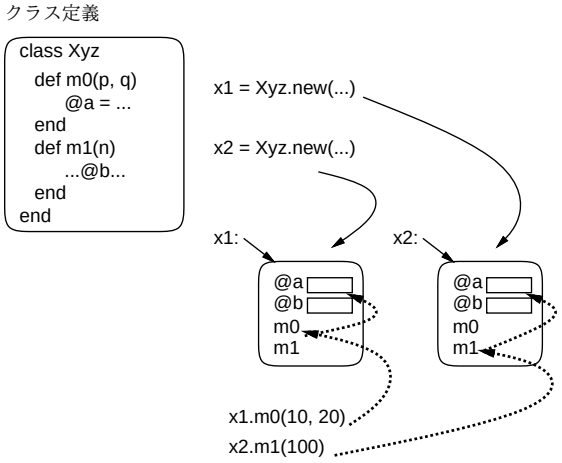


図 9.4: クラスとインスタンス

そして、メッセージ送信記法で「`x1.m0(...)`」「`x2.m1(...)`」のようにインスタンスメソッドを呼び出すと、それらのインスタンスメソッド中でインスタンス変数を参照した時は、それぞれ `x1`、`x2` のインスタンス変数が使われます。

たとえば、「犬」というクラスを作って、そこで「名前」「走っている速さ」というインスタンス変数を持たせたとすると、どの犬もこれら 2 つのインスタンス変数を持っているという点は同じですが、そこに格納されている値、つまりそれぞれの犬の名前やそれぞれの犬の走っている速さは、どの犬かによって、つまりインスタンスによって違う、というわけです。[4]

9.2.3 Ruby による簡単なクラスの定義 exam

Ruby の場合についてクラス定義の方法を説明します。クラスは次の構文により定義します。

```
class クラス名
  ...
end
```

クラス名は必ず英大文字で始めることになっています。そして、この中にメソッド定義を書くと、自動的にインスタンスメソッドになり、メッセージ送信記法で呼び出せるようになります。また、インスタンス変数はこれまでの変数と異なり、名前の最初が「`@`」で始まります。

そして最後に、クラスからインスタンスを作るには「クラス名`.new(...)`」という特別なメソッド呼び出しを使います。この時、もしインスタンスメソッドの中に `initialize` という名前のものがあればそれが呼び出され、その時 `new` に渡したパラメタがそっくりそのまま渡されてきます。つまり名前どおり、初期化のためにこのような仕組みになっているわけです。[5]

では、クラス定義の例を見てみます (先に説明で使った「犬」をクラスとして定義しました)。

```
class Dog
  def initialize(name)
    @name = name; @speed = 0.0
  end
  def talk
```

[4]Ruby ではクラスもオブジェクトなので、クラスに直接付随するメソッドである**クラスメソッド** (class method)、クラスに直接付随する変数である**クラス変数** (class variable) も存在します。クラスメソッドを呼び出す場合はメッセージ送信記法のオブジェクトのところにクラスの名前を指定します。

[5]ここでは使いませんが、クラスメソッドを定義する場合は「`def` クラス名`.` メソッド名 … `end`」のような `def` をクラスの外側に書いて定義します。また変数名を「`@@`」で始めるとその変数はクラス変数になります。

```
    puts('my name is ' + @name)
  end
  def addspeed(d)
    @speed = @speed + d
    puts('speed = ' + @speed.to_s)
  end
end
```

initialize では名前を受け取り、その値でインスタンス変数@name を初期化します。インスタンス変数@speed は 0 に設定します。メソッド talk では自分の名前を喋ります (喋る犬?)。addspeed では渡された値だけスピードを増して、それを表示します。動かしてみましょう。

```
irb> a = Dog.new('pochi')
=> #<Dog:0x81b5b2c @name="pochi", @speed=0.0>
irb> b = Dog.new('tama')
=> #<Dog:0x81b049c @name="tama", @speed=0.0>
irb> a.talk
my name is pochi
=> nil
irb> b.talk
my name is tama
=> nil
irb> a.addspeed(5.0)
speed = 5.0
=> nil
irb> b.addspeed(8.0)
speed = 8.0
=> nil
irb> a.addspeed(10.0)
speed = 15.0
=> nil
```

ポチのインスタンスとタマのインスタンスは別であり、名前や速度を別に持つことが分かると思います。このように「もの」単位で扱えるところが、オブジェクト指向の特徴なのです。

演習 1 この例題を打ち込み動かせ。次に「ほえる」メソッド bark(引数無) と、「ほえる回数」を設定するメソッド setcount(回数を渡す) を追加せよ。最初は 3 回ほえるものとする。[6]

演習 2 次のような機能と使い方を持つクラスを作成せよ。使用例の通りに使えることを確認すること。

a. 「覚える」機能を持つクラス Memory。put(x) で与えた容を記憶し、get で取り出す。

```
irb> m1 = Memory.new            # 作る
=> #<Memory:0x81d59e0 @mem=nil>
irb> m1.put(5)                  # 5 を覚えさせる
=> 5                            # put の返値は任意
irb> m1.get                     # 取り出す
=> 5                            # 5
irb> m1.get                     # 再度取り出す
```

[6]もちろん、ほえる回数を憶えるインスタンス変数を追加する必要があるはずです。

```
=> 5                          # やはり 5
irb> m1.put(10)               # 10 を覚えさせる
=> 10
irb> m1.get                   # 取り出す
=> 10                         # 10
```

b. 「文字列を連結していく」クラス Concat。add(s) で文字列 s を今まで覚えているものに連結する (最初は空文字列)。get で現在覚えている文字列を返す。reset で覚えている文字列を空文字列にリセット。(文字列どうしを連結するのは「+」でできます。)

```
irb> c = Concat.new           # 作る
=> #<Concat:0x81c7e94 @str="">
irb> c.add("This")            # 追加
=> "This"
irb> c.add("is")              # 追加
=> "Thisis"
irb> c.get                    # 取り出す
=> "Thisis"
irb> c.add("a")               # 追加
=> "Thisisa"
irb> c.reset                  # リセット
=> ""
irb> c.add("pen")             # 追加
=> "pen"
irb> c.get                    # 取り出し
=> "pen"
```

c. 「最大 2 つ覚える」機能を持つクラス Memory2。put(x) で新しい内容を記憶させ、get で取り出す。2 回取り出すと 2 回目はより古い内容が出てくる。取り出した値は忘れる。覚えている以上に取り出すと nil が返る (興味があれば「最大 N 個覚える」をやってもよい)。

```
irb> m2 = Memory2.new  # 作る
=> #<Memory2:0x80fdab8 @mem2=nil, @mem1=nil>
irb> m2.put(1)    # 1 を入れる
=> 1
irb> m2.put(3)    # 3 を入れる
=> 3
irb> m2.put(5)    # 5 を入れる
=> 5
irb> m2.get       # 取り出す → 5
=> 5
irb> m2.get       # 取り出す → 3
=> 3
irb> m2.get       # 取り出す → nil (2 個が限界)
=> nil
irb> m2.put(7)    # 7 を入れる
=> 7
irb> m2.put(9)    # 9 を入れる
=> 9
irb> m2.get       # 取り出す → 9
```

```
=> 9
irb> m2.put(11)   # 11 を入れる
=> 11
irb> m2.get       # 取り出す → 11
=> 11
irb> m2.get       # 取り出す → 7
=> 7
```

9.2.4 例題: 有理数クラス

今度は、もう少し有用なものを作ってみます。これまで、実数の計算には誤差がつきものだという説明をしてきましたね。具体的には、浮動小数点計算では割り切れない除算は循環小数になるので、必ず誤差が生じます。

そこで代わりに、数値を $\frac{\text{分子}}{\text{分母}}$ という形で保持すれば誤差なく除算結果を保持できるはずです (もちろん、加減算の時は通分して計算し、最後に約分します ($\sqrt{2}$ や π などの無理数は扱えません)。そのような**有理数** (rational number) クラスを作ってみましょう。このクラスでは、インスタンス変数`@a`と`@b`に分子と分母をそれぞれ保持するようにしています。

```
class Ratio
  def initialize(a, b = 1)
    @a = a; @b = b
    if b == 0 then @a = 1; return end
    if a == 0 then @b = 1; return end
    if b < 0 then @a = -a; @b = -b end
    g = gcd(a.abs, b.abs); @a = @a/g; @b = @b/g
  end
  def getDivisor
    return @b
   end
  def getDividend
     return @a
  end
  def to_s
    return "#{@a}/#{@b}"
  end
  def +(r)
    c = r.getDividend; d = r.getDivisor
    return Ratio.new(@a*d + @b*c, @b*d)  # a/b+c/d = (ad+bc)/bd
  end

  def gcd(x, y)
    while true do
      if x > y then x = x % y; if x == 0 then return y end
      else          y = y % x; if y == 0 then return x end
      end
    end
  end
end
```

initializeのパラメタに代入が書いてありますが、これは**デフォルト値** (default value) つまりそのパラメタを省略した場合はこの値を使ってねという意味になります。ですから、「Rational.new(3)」で $\frac{3}{1}$ になるわけです。initializeの中身がごちゃごちゃしていますが、これは (1) 分母が 0 の時 (不定) は分子を 1 とする、(2) 分母は 0 でないなら常に正とする (負の数は分子が負)、(3) 値ゼロは $\frac{0}{1}$ で表す、(4) 必ず既約分数にする、という**正規化** (normalization — なるべく形を揃えること) を行っているからです。

分母だけ、または分子だけを取り出したい場合のために、メソッド getDivisor、getDividend を用意しました。このような、インスタンス変数をアクセスするだけのメソッドのことを**アクセサ** (accessor) と呼びます。Ruby ではアクセサがなければインスタンス変数の内容は外部からは参照できません。これにより、カプセル化が実現され、また後で内部のデータ表現を変えた時にも外部に影響が及ばないで済みます。

また、文字列への変換メソッド to_s も用意しました。これは puts などによる打ち出しなどの時に自動的に「a/b」という形の文字列を生成できるので、用意しておくと便利です。そして、演算としてはとりあえず加算だけを用意しました。加算は「+」で表したいので、メソッド名を「+」にしてあります。このようにして、演算子を定義できるのは Ruby の特徴の 1 つです (C++などでも演算子定義は可能です)。

では動かしてみましょう。

```
irb> a = Ratio.new(3,5)
=> #<Ratio:0x81f978c @b=5, @a=3>
irb> puts a
3/5
=> nil
irb> b = Ratio.new(8,7)
=> #<Ratio:0x81f00d8 @b=7, @a=8>
irb> puts b
8/7
=> nil
irb> puts a+b
61/35
=> nil
```

確かに、通分して計算してくれていますね。なお、なぜ puts で打ち出しているかというと、puts は引数を文字列に変換して出力するため to_s を呼んでくれるからです。単に irb の機能で打ち出させるのだと、オブジェクトを表す「#<Ratio ....>」というのが表示されてしまいます。

演習 3 有理数クラスをそのまま打ち込んで動かせ。動いたら、四則の他の演算も追加し、動作を確認せよ。できれば、これを用いて浮動小数点では正確に行えない「実用的な」計算が正確にできることを確認してみよ。

演習 4 **複素数** (complex number) を表すクラス Comp を定義し、動作を確認せよ。これを用いて何らかの役に立つ計算をしてみられるとなおよい。[7]

演習 5 クラス定義を活用した「面白い」Ruby プログラムを作って動かせ。面白さの定義は各自に任されるものとする。

[7]名前を Complex にしたくなるかも知れませんが、標準ライブラリの名前と衝突するのでやめておきましょう。

本日の課題 **9A**

「演習 1」～「演習 2」で動かしたプログラム (どれか 1 つでよい) を含むレポートを提出しなさい。プログラムと、簡単な説明が含まれること。アンケートの回答もおこなうこと。

Q1. クラスの概念やその機能について納得しましたか。

Q2. オブジェクト指向というものにどのような感想を持ちましたか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 **9B**

「演習 2」～「演習 5」の (小) 課題から選択して 1 つ以上プログラムを作り、レポートを提出しなさい。プログラムと、課題に対する報告・考察 (やってみた結果・そこから分かったことの記述) が含まれること。アンケートの回答もおこなうこと。

Q1. クラス定義が書けるようになりましたか。

Q2. オブジェクト指向について納得しましたか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

＃10　動的データ構造＋情報隠蔽

今回は新たな題材として動的データ構造を扱いますが、これをオブジェクト指向による情報隠蔽の具体例としても捉えます。

- 動的データ構造の概念と考え方、単連結リストの操作
- 情報隠蔽 (カプセル化) の考え方

10.1　前回の演習問題の解説

10.1.1　演習 1 — クラス定義の練習

この演習はメソッドとインスタンス変数が追加できればよいということで、コードだけ掲載しておきましょう。

```
class Dog
  def initialize(name)
    @name = name; @speed = 0.0; @count = 3
  end
  def talk
    puts('my name is ' + @name)
  end
  def addspeed(d)
    @speed = @speed + d
    puts('speed = ' + @speed.to_s)
  end
  def setcount(c)
    @count = c
  end
  def bark
    @count.times do puts('Wan! ') end
  end
end
```

10.1.2　演習 2 — 簡単なクラスを書いてみる

この演習はクラスの書き方の練習みたいなものなので、見ていただけば十分でしょう。Memory2 はちょっと頭を使う必要がありますかね。

```
class Memory
  def initialize()
    @mem = nil
  end
  def put(x)
    @mem = x
```

```
    end
    def get()
      return @mem
    end
  end

  class Memory2
    def initialize()
      @mem2 = @mem1 = nil
    end
    def put(x)
      @mem2 = @mem1; @mem1 = x
    end
    def get()
      x = @mem1; @mem1 = @mem2; @mem2 = nil; return x
    end
  end

  class Concat
    def initialize
      @str = ""
    end
    def add(s)
      @str = @str + s
    end
    def get()
      return @str
    end
    def reset()
      @str = ""
    end
  end
```

10.1.3 演習 3 — 有理数クラス

この演習も Ratio クラスのメソッドを「同様に」増やせばよいだけなので、難しくはありません。追加するメソッドだけ掲載します。

```
    def -(r)
      c = r.getDividend; d = r.get Divisor
      return Ratio.new(@a*d - @b*c, @b*d)  # a/b-c/d = (ad-bc)/bd
    end
    def *(r)
      return Ratio.new(@a*r.getDividend, @b*r.getDivisor)
    end
    def /(r)
      return Ratio.new(@a*r.getDivisor, @b*r.getDividend)
    end
```

要は、引き算は足し算と同様、乗算は分母どうし掛け、除算はひっくり返して掛けるということですね。

10.1.4 演習 4 — 複素数クラス

複素数も 2 つの値 (実部、虚部) の組なので、有理数によく似ています。ただし個々の値として整数でなく実数を使います。演算はちょっと面倒 (とくに除算) ですが、作る時に約分とか分母が 0 とか考えなくてよい部分は簡単になります。

```
class Comp
  def initialize(r = 1.0, i = 0.0)
    @re = r; @im = i
  end
  def getRe
    return @re
   end
  def getIm
     return @im
  end
  def to_s
    if @im < 0 then
      return "#{@re}#{@im}i"
    else
      return "#{@re}+#{@im}i"
    end
  end

  def +(r)
    return Comp.new(@re + r.getRe, @im + r.getIm)
  end
  def -(r)
    return Comp.new(@re - r.getRe, @im - r.getIm)
  end
  def *(r)
    return Comp.new(@re*r.getRe - @im*r.getIm,
                    @im*r.getRe + @re*r.getIm)
  end
  def /(r)
    norm = (r.getRe**2 + r.getIm**2).to_f
    return Comp.new((@re*r.getRe + @im*r.getIm) / norm,
                    (@im*r.getRe - @re*r.getIm) / norm)
  end
end
```

to_s でヘンなことをやっているのは、「$a + bi$」の形で表示させようとした時、虚数部が負の場合には「$a - bi$」にしたいためです。

10.2 動的データ構造/再帰的データ構造

10.2.1 動的データ構造とその特徴 exam

データ構造 (data structure) とは「プログラムが扱うデータのかたち」を言います。ここまでに扱ってきたプログラムではおおむね、各変数には決まった形のデータが入り、それらはプログラムの実行が進んでも同じままでした。これを**静的**データ構造 (static data structure) と呼びます。以下では、プログラムの実行につれて構造を自在に変化させられる、**動的データ構造** (dynamic data structure) について学びます。[1]

動的データ構造は、プログラム言語が持つ「データのありかを指す」機能を用いて作ります。Ruby では、複合型 (配列、レコード等) や一般のオブジェクトの値は実際は、それらのありかを指す**参照**になっているので、これを用います。

以下ではレコード型を使って動的データ構造を作ります (レコード型は複数のフィールドを並べた構造でしたね)。たとえば、次のレコードを見てみましょう。

```
Cell = Struct.new(:data, :next)
```

これは 2 つのフィールド `data` と `next` を持つ `Cell` という名前のレコードを定義していますが、ここで各セルのフィールド `next` に「次」のセルへの参照を入れることで、「数珠つなぎ」の動的データ構造を作ることができます (図 10.1(a))。[2] このような「数珠つなぎ」の構造のことを**単連結リスト** (single linked list) ないし単リストと呼びます。

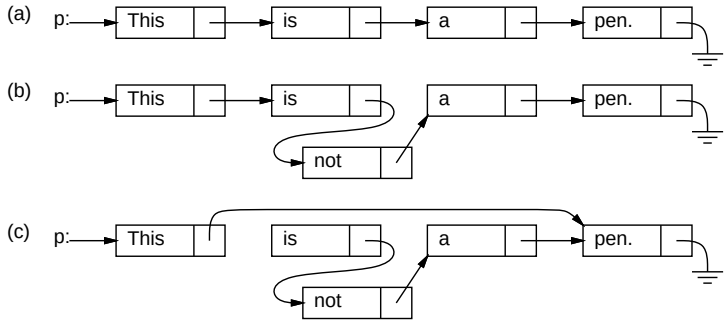


図 10.1: 単連結リストの動的データ構造

この `Cell` の使い方は、各 `Cell` の `next` がまた `Cell` になっていて、自分の中に自分が入っているように思えます。これは再帰関数と同様で、このようにデータ型 (構造) の中に自分自身と同じデータ型 (構造) への参照を含むものを**再帰的データ構造** (recursive data structure) と呼びます。実際には自分自身が入っているわけではなく、図 10.1 のように「同種のデータへの参照」が入っているだけですから、何ら問題はありません。

一番最後のところ (アース記号で表している) は「何も入っていない」という印である **nil** が入っています。このあたりも、「簡単な場合は自分自身を呼ばずにすぐ値が決まる」再帰関数と似ています。

動的データ構造だと何がよいのでしょうか? たとえば、図 10.1(a) で途中に単語「not」を入れたくなったとします。文字列の配列であれば、途中に挿入するためには後ろの要素を 1 個ずつずらして空いた場所に入れる必要があります。しかし、単連結リストでは、矢線 (参照) を (b) のようにつけ替えるだけで挿入ができます。逆に、数単語削除したいような場合も、(c) のように参照のつけ換えで行えます。このように、動的データ構造は柔軟な構造の変更が行えるという特徴を持ちます。

---

[1] 一般に、静的は「プログラム記述時に決まる」、動的は「プログラム実行時に決まる」という意味になります。

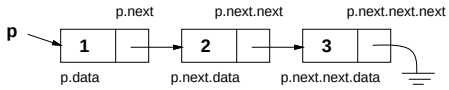
[2] 本当はフィールド `data` も文字列オブジェクトを参照しているので、文字列を箱の外に描いて矢線で指させるべきなのですが、ごちゃごちゃして見づらくなるのでここでは箱の中に直接描いていています。

10.2.2 単連結リストを操作してみる exam

では、プログラムを書く前に irb で直接試してみましょう。まず、以下のことをやってみてください。

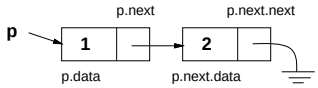
```
irb> Cell = Struct.new(:data, :next)
irb> p = Cell.new(1, Cell.new(2, Cell.new(3, nil)))
irb> p
```

表示をよく見てください。次の図のような構造ができているはずです。図に出て来る「p.data」、「p.next」等を irb に打ち込んで表示を確認すること (以下同様)。



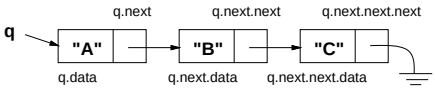
では次に、矢線をつけかえてみましょう。

```
irb> p.next.next = nil
irb> p
```



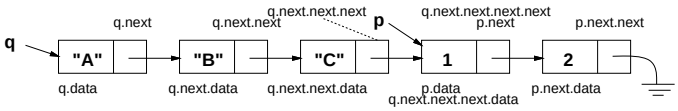
今度は文字列が値になっているリストを作ります。

```
irb> q = Cell.new("A", Cell.new("B", Cell.new("C", nil)))
irb> q
```



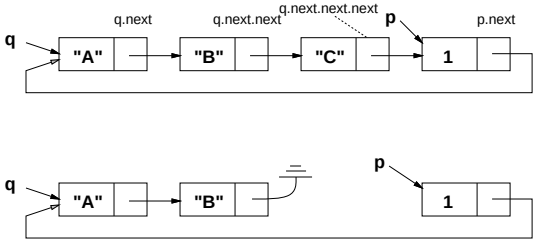
さっきのリストとくっつけてみましょう。

```
irb> q.next.next.next = p
irb> q
```



このように、参照を代入するだけで構造をつなげたり切り離したりできます。これが動的データ構造の利点になります。

演習 0 上の続きとして、次のような形を作れるかやってみなさい。



10.2.3 単連結リストのループと再帰による操作 exam

ではコードを書くことにして、毎回先のように打ち込んでリストを作るのでは大変なので、まずは配列を渡すとリストに変換してくれるメソッドを書きます。

```
Cell = Struct.new(:data, :next)   # 1 回定義すれば十分
def atol(a)
  p = nil
  (a.length-1).step(0, -1) do |i| p = Cell.new(a[i], p) end
  return p
end
```

リストは後ろの方から作る必要があるので、次のように配列の末尾から先頭に向かって 1 つずつセルを作っていきます。p は最初 nil を入れておいてループ周回ごとに書き換えていますが、それまでの p の値がリストの next に入ることにより、つながったリストができます (図 10.2)。

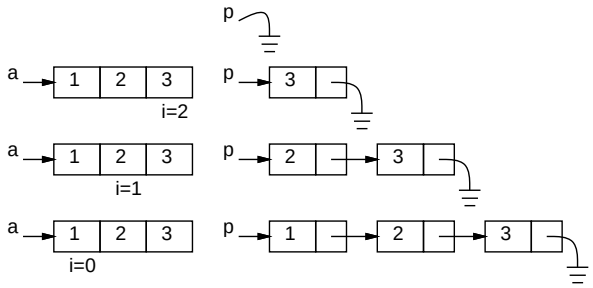


図 10.2: ループで配列から単連結リストを作る

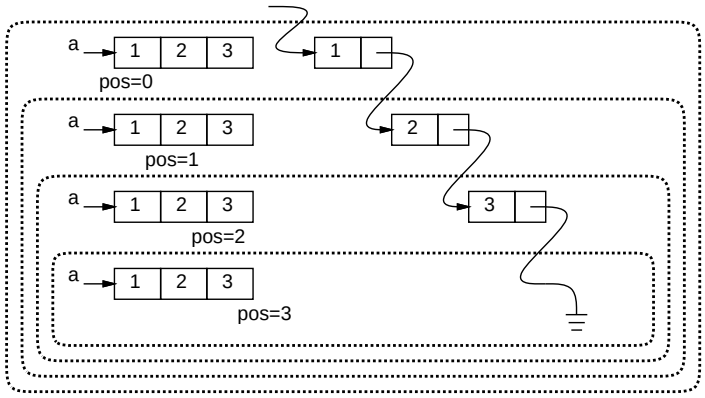


図 10.3: 再帰で配列から単連結リストを作る

ところで、同じことを「再帰呼び出しを使って」やるバージョンを見てみましょう (再帰版は recursive の r を末尾につけた名前にします)。このメソッドは「配列の指定した位置 pos から後ろをリストにして返す」のですが、pos は指定しない場合は 0 になります (デフォルト引数の機能)。

```
def atolr(a, pos = 0)
  if a.length <= pos
    return nil
  else
    return Cell.new(a[pos], atolr(a, pos+1))
  end
end
```

それで、どう読めばいいのでしょう？「配列が指定位置より短いなら、データは無いので nil を返し、そうでなければ指定位置のセルの後ろに、1 つ先の位置以降のリストをくっつけたもの を返せばよい」ということです。そして、下線の部分は自分自身を再帰呼び出しすることで実現できます。この呼び出しのようすを図 10.3 に示しました。

動かしてみましょう。当然ながら、どちらでもできるものは一緒です。

```
irb> atol [1, 2, 3]
=> #<struct Cell data=1, next=#<struct Cell data=2,  next=#<struct Cell
data=3, next=nil>>>
irb> atolr ["a", "b", "c"]
=> #<struct Cell data="a", next=#<struct Cell data="b", next=#<struct
Cell data="c", next=nil>>>
```

再帰的データ構造は名前から分かるように再帰的メソッドと相性がよいです。たとえば、単連結リストに何個セルがつながっているかを調べるとします。ループ版と再帰版の両方を示します。

```
def listlen(p)
  len = 0
  while p != nil do p = p.next; len = len + 1 end
  return len
end
def listlenr(p)
  if p == nil
    return 0
  else
    return listlenr(p.next) + 1
  end
end
```

ループ版では p が nil になるまで次のセルに進みながら、進むごとにカウンタを 1 増やすことで長さを求めています。では再帰版は？「渡された p が nil なら、長さは 0、そうでなければ、次のセル以降の長さに 1 足したものが長さ」ということですね。

たどりながら出力する方が動作が分かりやすいかも知れません。各データを順に出力してみましょう。

```
def printlist(p)
  while p != nil do puts(p.data); p = p.next end
end
def printlistr(p)
  if p != nil then puts(p.data); printlistr(p.next) end
end
```

考え方はこれまでと同じです。ループの方が分かりやすいですか？ では次の実行例を見てください。

```
irb> p = atol ["A", "B", "C"]
...
irb> printlistr p
A
B
C
=> nil
```

```
irb> revprintlistr p
C
B
A
=> nil
```

しかし revprintlistrって? それは次のものです。まず残りを打ち出してもらい、最後に自分の担当を打ち出す、というふうに逆にしただけでこうなるわけです。

```
def revprintlistr(p)
  if p != nil then revprintlistr(p.next); puts(p.data) end
end
```

ループでこれをやろうと思うと、データを全部まず配列などに保存してから逆順で出力することになり、大変です。このように、場合によっては再帰でないと書きにくいものもあるわけです。

演習 1 ここまでの例 (構築、長さ、プリント) を打ち込んで動かせ。動いたら、次のメソッドを作ってみよ (ループでも再帰でも好きな方でよい)。

a. data に数値が入っている単連結リストに対して、その数値の合計を求める listsum。
b. 各セルの data(文字列) を連結した 1 つの文字列を返す listcat。文字列連結は「p.data.to_s + …」のように to_s で文字列にしてから「+」を使えばよい。
c. 上と同様だがただし逆順に連結する listcatrev。
d. printlist と同様だが、1 行目は 1 回、2 行目は 2 回、3 行目は 4 回、…と倍倍で打ち出す回数が増える printmany。打ち出す順番は任意の順番で (ごちゃまぜで) よい。
e. listsum と同様だが、ただし奇数番目のセルの値だけ合計する listoddsum。
ヒント: ループでも再帰でも「次をたどる」代わりに「次の次をたどる」ようにすれば 1 つおきになります。ただし「次が nil」になることもあるのに注意。再帰では別の方法として、自分自身を再帰呼び出しする代わりに、奇数番目用 (加算をする) は偶数番目用 (加算しない) を呼び、偶数番目用は奇数番目用を呼ぶ、というふうに交互にやる方法もあります。これを**相互再帰**と呼びます。
f. 単リストの並び順を逆向きにした単リストを返す listrev。これには元のリストを変更してしまう版と変更しない版とが考えられるが、どちらでもよいことにする。

10.3 情報隠蔽

10.3.1 例題: 単連結リストを使ったエディタ

ではここで、単連結リストを使った例題として、簡単な**テキストエディタ** (text editor) を作ってみましょう。「簡単」なので、編集に使うコマンドは次のものしかありません。

- 「i 文字列」 — 文字列を新しい行として現在位置の直前に挿入する。
- 「d」 — 現在位置の行を削除する。
- 「t」 — 先頭行を表示し、そこを現在位置とする。
- 「p」 — 現在位置の内容を表示する。
- 「n」または改行 — 現在位置を次の行へ移しその行を表示する。
- 「q」 — 終了する。

実際にこれを使っている様子を示します (すごく面倒そうですが、実際にこういうプログラムを使ってファイルの編集をしていた時代は実在しました)。

```
>iThis is a pen.       ←挿入
>iThis is not a book. ←挿入
>iHow are you?         ←挿入
>t                     ←先頭へ
  This is a pen.
>                      ←次の行
  This is not a book.
>                      ←次の行
  How are you?
>                      ←次の行
  EOF                  ←おしまい
>t                     ←再度先頭へ
  This is a pen.
>iI am a boy.          ←挿入
>                      ←次の行
  This is not a book.
>iWho are you?         ←挿入

>t                     ←再度先頭へ行き全部見る
  I am a boy.
>
  This is a pen.
>
  Who are you?
>
  This is not a book.
>
  How are you?
>
 EOF
>q                     ←おしまい
```

これをこれから実現してみましょう。

10.3.2 エディタバッファ

以下では、単連結リストのデータ構造を先頭や現在位置などの各変数も含めてクラスとしてパッケージします。

```
class Buffer
  Cell = Struct.new(:data, :next)
  def initialize
    @tail = @cur = Cell.new("EOF", nil)
    @head = @prev = Cell.new("", @cur)
  end
  def atend
    return @cur == @tail
  end
  def top
```

```
    @prev = @head; @cur = @head.next
  end
  def forward
    if atend then return end
    @prev = @cur; @cur = @cur.next
  end
  def insert(s)
    @prev.next = Cell.new(s, @cur); @prev = @prev.next
  end
  def print
    puts("  " + @cur.data)
  end
end
```

レコード定義もクラスに入れることにしました。また、「1 つの値を複数箇所に代入する」のに=を連続して書いてみました。もちろん、2 つの代入に分けても一向に構いません。

このクラスでは、単連結リストのセルを上記の Cell レコードであらわし、これを指すための変数として次の 4 つを使っています。

- @head — 一番先頭に「ダミーの」セルを置き、そのセルを常にこの変数で指しておく (ダミーがあると、先頭行を削除するのを特別扱いしないで済ませられるため、プログラムの作成が楽になります)。
- @cur — 「現在行」のセルを指しておく。
- @prev — 「現在行の 1 つ前」のセルを指しておく (挿入や削除の時にこの変数があるとコードを書くのが楽です)。
- @tail — 一番最後にも「ダミーの」セルを置き、そのセルをこの変数で指しておく (表示することがあるので内容は「EOF」(end of file) としてあります)。

initialize では 2 つのダミーセルと上記 4 変数を用意します。head の次が tail であるように Cell.new にパラメタを渡していることにも注意。

では次に、メソッドを見てみましょう。図 10.4 に、適当なバッファの状態で操作を行った例を示します (最後の delete は課題の参考用です)。atend は現在行が末尾にあるか (@tail と等しいか) を調べます。top は@prev と@cur を先頭に設定します。forward は@prev と@cur を 1 つ先に進めますが、現在行が@tail の時は「果て」なので何もしません。print は現在行の文字列を表示します。insert は新しいセルが@prev となり、元の@prev のセルの次が新しいセル、新しい@prev の次が@cur のセルとなります。これを動かした様子を見てみましょう。[3]

```
irb> e = Buffer.new
=> ...
irb> e.insert('abc')
=> ...
irb> e.insert('def')
=> ...
irb> e.insert('ghi')
=> ...
irb> e.top
=> nil
```

[3] irb の結果表示はうるさいので省略しています。

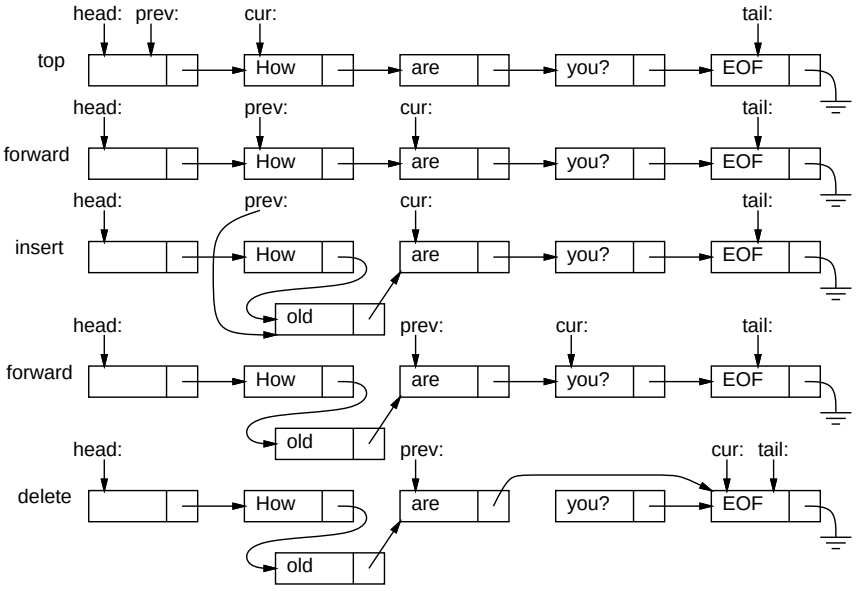


図 10.4: エディタバッファに対する操作

```
irb> e.print
   abc
=> ...
irb> e.forward
=> ...
irb> e.print
   def
=> nil
```

確かに文字列が順序どおり挿入でき、それをたどることができています。

このクラスは「行の挿入や削除が自在にできる機能を持ったオブジェクト」を作り出しています。内部では込み入ったデータ構造を管理していますが、その様子はクラスの外側からは見えません。このように内部構造を外から見せないようにして外部に機能を提供することを**情報隠蔽** (information hiding) または**カプセル化** (encapsulation) と呼びます。その利点は、内部のデータにアクセスするのはそのクラスのコードだけなので、データ構造の整合性が保て、またプログラムの正しさの確信が持ちやすいことです。

また、カプセル化を用いて、操作だけが外から呼び出せ、それを用いて整合性のある汎用的な機能を提供するものを、**抽象データ型** (abstract data type, ADT) と呼びます。クラス方式のオブジェクト指向言語では、抽象データ型はクラスによって定義するのが自然です。前章に出てきた有理数クラスや複素数クラスも抽象データ型の例だといえます。

演習 2 図 10.5 は「How」という行と「are」という行の間に「old」という行を挿入する様子 (A → B)、および、「old」「are」「you?」という 3 行のうちから「are」を削除する様子 (B → C) を示しています。資料 (ないし同じ図を描き写したもの) の上に赤ペンで次のものを記入しなさい。

a. (A) の図の上に、(A) から (B) につながりが変化するための矢線のつけ替えを、(1)、(2) のようにつけ替えを行う順番つきで記入しなさい。ただし、矢印のつけ替えを行う時には、その出発点がどれかの変数そのものであるか、またはどれかの変数から矢線でたどれる箱であることが必要である。

b. (B) の図の上に、(B) から (C) につながりが変化するための矢線のつけ替えを、(1)、(2) のようにつけ替えを行う順番つきで記入しなさい。ただし、矢印のつけ替えを行う時には、その出発点がどれかの変数そのものであるか、またはどれかの変数から矢線でたどれる箱であることが必要である。

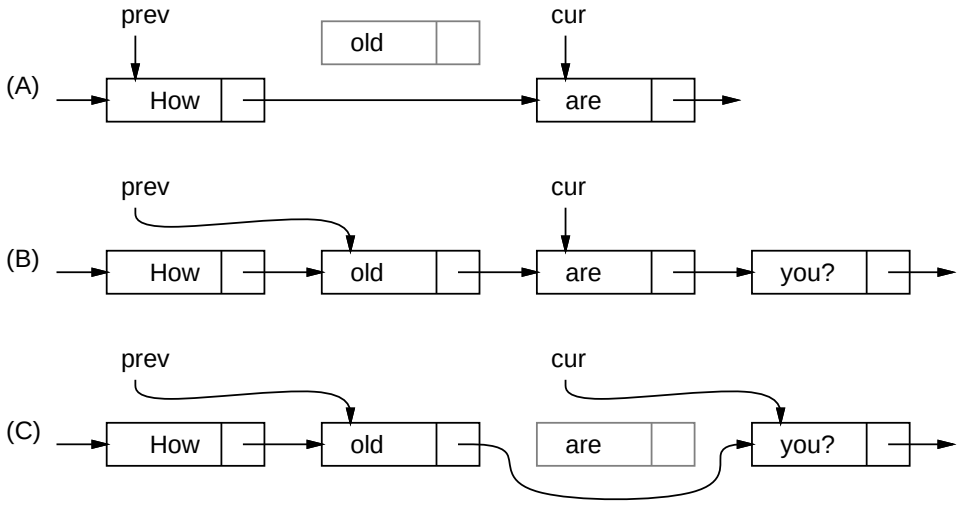


図 10.5: 挿入と削除のようす

演習 3 クラス `Buffer` を打ち込み、動作を確認せよ。動いたら、以下の操作 (メソッド) を追加してみよ。

a. 現在行を削除する (EOF 行は削除しないように注意…)
b. 現在行と次の行の順序を交換する (EOF は交換しないように…)
c. 1 つ前の行に戻る (実は大変かも)
d. すべての行の順番を逆順にする (かなり過激)

演習 4 単連結リストでは各セルが「次」の要素への参照だけを保持していたが、各セルが「次」と「前」2 つの参照を持つようなリストもある。これを**双連結リスト** (double linked list) と呼ぶ。編集バッファの双連結リスト版を作り、その得失を検討せよ。[4]

10.3.3 エディタドライバ

バッファのメソッドを呼ぶだけでも編集はできますが、面倒です。先にお見せしたように「コマンド (+パラメタ)」ですらすら編集ができるように、エディタとして動作するコードも作ってみました。内容はとても簡単で、バッファを生成し、その後無限ループでプロンプトを出し、1 行読んでは先頭の 1 文字でどのコマンドを実行するか枝分かれします (コメントにしてあるのはあなたが作るか、後で機能を追加するためのものです)。[5]

```
def edit
  e = Buffer.new
  while true do
    printf(">")
```

[4] 双連結リストでは単連結リストでの「頭」と「最後」を 1 つで兼ねることもできます (無理に兼ねなくてもよい)。
[5] コマンド「n」の記述がありませんが、「n」は else の節へ来るので、結果として所定の動作になります。

```
      line = gets; c = line[0..0]; s = line[1..-2]
      if c == "q"    then return
      elsif c == "t" then e.top; e.print
      elsif c == "p" then e.print
      elsif c == "i" then e.insert(s)
  #   elsif c == "r" then e.read(s)
  #   elsif c == "w" then e.save(s)
  #   elsif c == "s" then e.subst(s); e.print
  #   elsif c == "d" then e.delete
      else                e.forward; e.print
      end
    end
  end
```

文字列の一部を取り出すには「[位置..位置]」という添字指定を使います。1 文字だけの場合でも文字列として取り出したい場合は位置を 2 つ指定するので、先頭の文字は line[0..0] で取り出しているわけです。

i(insert) コマンド等では、2 文字目から最後の文字の手前まで (最後の文字は改行文字) も必要なので、これも取り出しています。Ruby では文字列や配列中の位置として負の整数を指定すると末尾からの位置指定になります。

どのコマンドでもない場合 (や改行だけの場合) はいちばんよく使う「1 行進んで表示」にしました。

演習 5 エディタドライバを打ち込んで先のクラスと組み合わせて動作を確認せよ。動いたら以下のような改良を試みよ (クラス側を併せて改良しても、このメソッドだけを改良しても、どちらでも構いません。文字列を数値にする必要が生じたら、メソッド to_i を使ってください)。

a. 演習 3 で追加した機能が使えるようにコマンドを増やす。
b. 現在行の「次に」新しい行を追加するコマンド「a」を作る (追加した行が新たな現在行になるようにしてください)。
c. 現在行の内容をまるごと置き換えるコマンド「c」を作る。
d. 「g 行数」で指定した行へ行くコマンド「g」を作る。
e. コマンド「p」を「p 行数」でその行数ぶん打ち出すように改良 (その際、できれば現在位置は変更しないほうが望ましいです)。
f. その他、自分が使うのに便利だと思うコマンドを作る。

10.3.4 文字列置換とファイル入出力

せっかくエディタができたのに、行内の置き換えとかファイルの読み書きができないと実用になりませんから、これらを一応解説しておきます。

まず、行内の置き換は「s/α/β/」により現在行中の部分文字列 α を β に置き換えるというコマンドにしました。エディタドライバからはバッファのメソッド subst を呼ぶだけとしたので、こちらの中身を示します。

```
  def subst(str)
    if atend then return end
    a = str.split('/')
    @cur.data[Regexp.new(a[1])] = a[2]
  end
```

文字列のメソッド split は、渡されたパラメタ「/」のところで文字列を分割した配列を返します。その 1 番目を Regexp(パターン) オブジェクトに変換して文字列に添字アクセスすると、そのパターンの箇所があれば、代入によりそこを別の文字列に置き換えられます。

ファイルの読み書きみは、# 5 で学んだ open でファイルを開き、読む場合は付属ブロック内でそのファイルの各行を insert、書く場合は逆にバッファの各行をファイルに puts で書き出します。

```
def read(file)
  open(file, "r") do |f|
    f.each do |s| insert(s) end
  end
end
def save(file)
  top
  open(file, "w") do |f|
    while not atend do f.puts(@cur.data); forward end
  end
end
```

演習 6 自作のエディタでどれか 1 課題ぶんの編集を行い、体験を述べよ。エディタの機能は何があれば必要十分か、使いやすさは何で決まるかについて考察すること。

演習 7 動的データ構造を活用した、何か面白いプログラムを作れ。面白さの定義は各自に任されます。

本日の課題 10A

「演習 1」「演習 3」で動かしたプログラム (どれか 1 つでよい) を含むレポートを提出しなさい。プログラムと、簡単な説明が含まれること。アンケートの回答もおこなうこと。

Q1. 動的データ構造とはどのようなものか理解しましたか。

Q2. 連結リストの操作ができるようになりましたか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 10B

「演習 1」「演習 3」〜「演習 7」の (小) 課題から選択して 1 つ以上プログラムを作り、レポートを提出しなさい。プログラムと、課題に対する報告・考察 (やってみた結果・そこから分かったことの記述) が含まれること。アンケートの回答もおこなうこと。

Q1. 何らかの動的データ構造が扱えるようになりましたか。

Q2. 複雑な構造をクラスの中にパッケージ化する利点について納得しましたか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

＃11　型と宣言+$f(x)=0$の求解

今回からC言語の内容に入ります。今回は次のことを取り上げます。

- 強い型の概念とC言語の基本、変数宣言、制御構造
- 1変数方程式 $f(x)=0$ の求解 (経験者向け)

「経験者向け」ですが、C言語は皆様の中にもやったことがある人がいると思います。そのような人に入門の内容だけやっても興味が持てないと思うので、以後各回とも後半は「経験者向け」とし、入門で物足りない人にやってもらうことを意図しています。試験範囲外であり、初心者はやらなくて良いです。ただし読んでおいてください (「付録」も読んでおいてください)。

11.1　演習問題解説

11.1.1　演習1 — 単連結リストを扱うメソッド

ループ版と再帰版を両方掲載していると長いので再帰版のみ示します。それほど難しくないと思うのでメソッドを一通り示しましょう。まずlistsumはnilのとき0を返し、それ以外は再帰で次のセル以降の合計を求めたものに自分の値を足します。

```
def listsum(p)
  if p == nil then return 0
              else return p.data + listsum(p.next) end
end
```

listcatはnilのとき返すものが空文字列なだけで、あとはヒント通りです。左右反対にするのには連結の左右を入れ換えればよいです。

```
def listcat(p)
  if p == nil then return ''
              else return p.data.to_s + listcat(p.next) end
                   # or listcat(p.next) + p.data.to_s
end
```

printmanyは再帰を2回呼ぶだけです。後で実行例を見てもらいます。

```
def printmany(p)
  if p != nil then puts(p.data); printmany(p.next); printmany(p.next) end
end
```

奇数番目のみ加算は、ヒントの通り2つのメソッドの相互再帰で奇数番目のだけ足し算します。

```
def listoddsum(p)
  if p == nil then return 0
              else return p.data + listoddsum2(p.next) end
end
def listoddsum2(p)
  if p == nil then return 0
              else return listoddsum(p.next) end
end
```

実行例は次の通り (数値のリストの課題と文字列のリストの 2 種類を用意しました)。

```
irb> Cell = Struct.new(:data, :next)
=> Cell
irb> listsum(atol([1, 3, 5]))
=> 9
irb> listcat(atol(["A", "B", "C"]))
=> "ABC"
irb> printmany(atol(["A", "B", "C"]))
A
B
C
C
B
C
C
=> nil
irb> listoddsum(atol([1, 3, 5]))
=> 6
```

リストの逆転は説明すると長くなるのでコードと実行の様子の図 11.1 だけ示します。`listrev1` は元のセルを書き換える方法 (図の左)、`listrev2` は元のセルを書き換ずに新しい逆転リストを作る方法 (図の右) です。いずれも 2 つ目の引数があり、指定しないときは `nil` が初期値になります。

```
def listrev1(p, n = nil)
  if p  == nil then return n end
  q = p.next; p.next = n; return listrev1(q, p)
end
def listrev2(p, q = nil)
  if p == nil then return q end
  return listrev2(p.next, Cell.new(p.data, q))
end
```

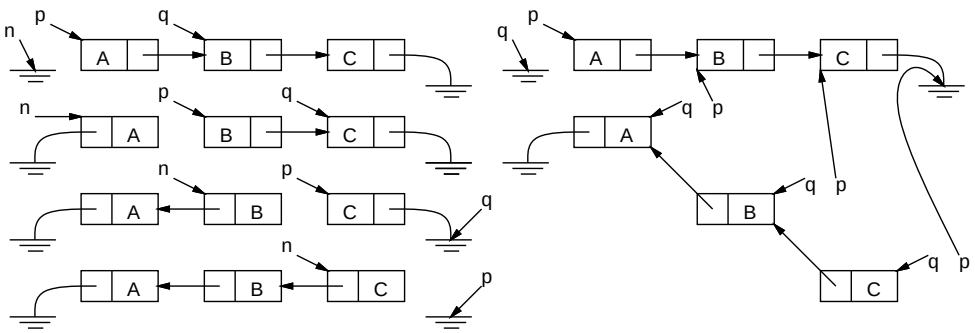


図 11.1: 2 通りのリスト逆転

11.1.2 演習 3 — エディタバッファのメソッド追加

この演習については、メソッドのみだけ掲載します。まず削除です。

```
def delete
  if atend then return end
  @cur = @prev.next = @cur.next
end
```

これは前回もかなり説明しましたが、要は (1) 最後の EOF は消さないようにする、(2)@cur は現在行の 1 つ先にする、(3)@prev の「次」も同じく現在行の 1 つ先にする、ということですね。どれかが足りないとおかしくなるので注意。次に交換です。

```
def exch
  if atend || @cur.next == @tail then return end
  a = @prev; b = @cur.next; c = @cur; d = @cur.next.next
  a.next = b; b.next = c; c.next = d; @cur = b
end
```

このように込み入ったつなぎ換えは、作業変数を使った方が間違えないで済みます。まず現在行か次の行が「おしまい」だったら交換できないのでそれを除外し、あとは最終的に並ぶ 4 つのセル (中央の 2 つが交換) を変数 a、b、c、d に入れて、つなぎ直し、@cur を変更します。

次は 1 つ戻るですが、せっかくある程度メソッドが作ってあるわけですから、「@prev を覚えておき、先頭に行ってから現在行が覚えておいた行になるまで 1 行ずつ進む」方法で作ってみました。

```
def backward
  if @prev == @head then return end
  a = @prev; top; while @cur != a do forward end
end
```

全部反転はやや大変ですが、先頭から順にたどりながら、今まで「前→後」の対だったものを「後←前」の順になるように参照をつなぎ換える (ただし先頭と末尾はそれなりに対処)、という方針です。

```
def invert
  top; if atend then return end
  a = @cur; b = @cur.next; a.next = @tail
  while b != @tail do c = b.next; b.next = a; a = b; b = c end
  @head.next = a; top
end
```

先頭と末尾の対処はまず最初に先頭の次を@tail にし、最後にループを抜けてきた時のセルを先頭 (@head の次) にする、ということです。なお、バッファ内に 1 行しかない時はループ周回数が 0 で、その時もちゃんと動作することに注意。

11.1.3 演習 5 — エディタの機能強化

「指定した行へ行く」機能はバッファ側で「何行目」を管理するのがよいので、エディタバッファ全体を示します。インスタンス変数@lineno を追加し、現在行が変化するメソッドでこれを更新します。invert のように大幅に直す場合は最後に top を呼び、ここで 1 にリセットされるので問題ありません。そして goto があれば backward はずっと簡単です。行番号を間違いなく維持するのはきわどそうに見えますが、@lineno をアクセスするのもバッファ内容や現在位置を変更するのも Buffer 内だけなので、この中できちんと処理すれば大丈夫です。つまり、オブジェクト指向の持つ**カプセル化**の機能によって、プログラムが正しく構成し易くなるのです。

```
class Buffer
  Cell = Struct.new(:data, :next)
```

```
    def initialize
      @tail = @cur = Cell.new("EOF", nil)
      @head = @prev = Cell.new("", @cur)
      @lineno = 1
    end
    def getlineno
      return @lineno
    end
    def goto(n)
      top; (n-1).times do forward end
    end
    def atend
      return @cur == @tail
    end

    def top
      @prev = @head; @cur = @head.next; @lineno = 1
    end
    def forward
      if atend then return end
      @prev = @cur; @cur = @cur.next; @lineno = @lineno + 1
    end
    def insert(s)
      @prev.next = Cell.new(s, @cur)
      @prev = @prev.next; @lineno = @lineno + 1
    end
    def print
      puts("  " + @cur.data)
    end
# delete、exch は上掲のとおり。backward は以下のように変更
    def backward
      goto(@lineno - 1)
    end
    def invert
      top; if atend then return end
      a = @cur; b = @cur.next; a.next = @tail
      while b != @tail do
        c = b.next; b.next = a; a = b; b = c
      end
      @head.next = a; top
    end
# subst, read, write は前回資料掲載
  end
```

エディタドライバ側も一応示します。「位置変更しない指定行数プリント」も、最初に行番号を覚え、印刷し終わったらそこに戻ればよいので簡単です。

```
  def edit
```

```
    e = Buffer.new
    while true do
      printf(">")
      line = gets; c = line[0..0]; s = line[1..-2]
      if c == "q"    then return
      elsif c == "t" then e.top; e.print
      elsif c == "p" then
        e.print; l = e.getlineno;
        s.to_i.times do e.forward; e.print end; e.goto(l)
      elsif c == "i" then e.insert(s)
      elsif c == "r" then e.read(s)
      elsif c == "w" then e.save(s)
      elsif c == "s" then e.subst(s); e.print
      elsif c == "d" then e.delete
      elsif c == "x" then e.exch
      elsif c == "b" then e.backward
      elsif c == "v" then e.invert
      elsif c == "a" then e.forward; e.insert(s); e.backward
      elsif c == "c" then e.delete; e.insert(s); e.backward
      elsif c == "g" then e.goto(s.to_i)
      else                e.forward; e.print
      end
    end
  end
```

11.2 C 言語入門

11.2.1 弱い型と強い型

これまで使ってきた Ruby では、変数は使ったときに自動的に用意され、どのような種類の値でも格納できました。それは確かに便利なのですが、変数名を間違ったり入れる値の種類を間違えても処理系は教えてくれないという弱点がありました。どのみちプログラムは実際に動かして間違いを調べる必要があるから、と思うかも知れませんが、苦労して動かしながら調べるよりも処理系が「これは違います」と言ってくれる方がずっと簡単に直せるのも確かです。

そのため世の中には、次のような設計のプログラミング言語も多くあります。

- 変数は使う前に**宣言** (declaration — これからこの変数を使うという指定) を書く必要がある。
- 宣言時に変数に**データ型** (値の種別) を明示し、それと異なる値を入れることを許さない。

なお、ここでは「変数」とだけ書きましたが、関数のパラメタや関数や返す値についても同様です (そうしないと検査がきちんとできない)。このような言語を、型を厳密に検査することから**強い型の言語** (strongly-typed language) と呼びます。[1] 繁雑で不自由なようですが、その方が結局プログラムの誤りを速やかに修正できる、というのが、この方式を支持する人の主張です。

また、強い型の言語のほうが CPU 命令への変換がやりやすく、結果としてプログラムが高速に実行できる場合が多い、という点もあります。さらに、CPU の動作との対応がはっきりしていて、**組み込みシステム** (embedded system — 様々な機器に CPU が搭載されていてそこでプログラムが動くもの) で使いやすい、という性質もあります。この 2 つの利点はこれから取り上げる C 言語にとくに

[1] Ruby のようにどの変数にどの種類の値を入れてもよい言語は**弱い型の言語** (weakly-typed language) です。

あてはまります。つまり、皆様がこれから研究や仕事でハードウェアに近いプログラミングをやる場合、C 系列の言語を使う可能性が高いと言えます。

そして本科目の立場としては、Ruby で弱い型をやったので、それと異なる強い型の言語も知って欲しいということと、2 つ異なる言語を体験することで「さまざまな言語といっても似たところも多い」ことを学んで頂きたいということから、**C 言語** (C language) を取り上げています。

11.2.2 C 言語のバージョンについて

本題に入る前に、C 言語のバージョンについて説明しておきます。C には現在おおよそ「K&R」「C89」「C99」「C11」の 4 つの版があります。K&R は最初に C 言語を解説した本「The C Programming Langauge」の著者 Kernighan、Ritche の名前からそう呼ぶもので、古い版です。そのあと標準化活動により C89、C99、C11 ができました (数字はそれぞれの規格ができた年の西暦の下 2 桁です)。

C99 が C89 の使いにくいところを改良していて処理系も普及しているので、本科目では C99 にしたがって説明しています。ただし今日でも C89 の処理系を使う場面があるかも知れないので、C89 と互換性のないところはその旨を説明し、C89 に書き換える方法も注記するようにします。

11.2.3 C 言語の実行環境と本科目でのスタイル

これまで Ruby では irb コマンドにプログラムをロードし、メソッド名を指定して実行開始させる、というやり方をしてきました。これは、irb を使うと配列やレコードなど様々なテストデータを直接打ち込んで指定でき、いろいろ試しやすいからです (Ruby で irb を使わない実行方法もあります)。また、結果の表示も irb が受け持ってくれていました。

しかし、C 言語では irb に相当するプログラムはなく、プログラムの中に `main` という関数 (Ruby でいうメソッド) が必ず必要で、実行はそこから開始されることに決っています。また、テストのためのデータの入力も、結果の表示も、「自分のプログラムで」書かなければなりません。

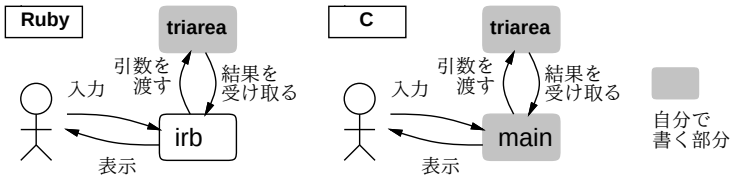


図 11.2: Ruby と C での実行環境の違い

そこで、以下の例題では当面次のような形で (`main` が irb の代わりをするような形で) プログラムを構成します (図 11.2)。

- Ruby と同じように、自分がやりたい動作を好きな名前の関数として作成する。
- `main` は「データを受け取って、本体の関数を呼び出し、結果を表示する」ことを主におこなう。

C 言語では `main` で本体の計算を書いてしまってもいいのですが、皆様は Ruby のやりかたに慣れていますし、入出力と計算を分けた方がきれいでもあるので、このような形をとります。

11.2.4 最初の C プログラム exam

本科目で最初にやった Ruby プログラムが三角形の面積だったので、C 言語でもそうします。復習のため、Ruby 版を再掲しておきます。

```
def triarea(w, h)   # Ruby 版の triarea
  s = (w * h) / 2.0
  return s
end
```

では次に、C 版を見ていただきます。ずっと長いですが、それは主に irb に相当するものが無くて入力も自分で扱う必要があるためです。

```
// triarea --- area of triangle
#include <stdio.h>

double triarea(double w, double h) {
  double s;
  s = (w * h) / 2.0;
  return s;
}

int main(void) {
  double w, h, s;
  printf("w> "); scanf("%lf", &w);
  printf("h> "); scanf("%lf", &h);
  s = triarea(w, h);
  printf("triarea = %g\n", s);
  return 0;
}
```

見比べれば分かるように、C 版の tirearea は中央にはさまっていて、上と下に他のものがくっついています。では最初なので丁寧に説明しましょう。

1. 「`// ...`」はそこから行末までがコメントになります。C では最初に実行する関数は `main` という名前に決められているので、何をするプログラムかのコメントは書いた方がいいでしょう。なお、このほかに「`/* ... */`」というコメントの書き方もできます。[2]
2. 「`#include <ファイル名>`」はシステムの決まった場所にあるファイルを取り込む指示です。[3] 最後が「`.h`」で終わるファイルは**ヘッダファイル**と呼ばれ、関数等の宣言を含んでいます。C では型検査のため、使用する各関数についてパラメタと結果の型を記述する必要があり、入出力など誰もが使う関数も同様です。しかしそれを毎回手で書き込むのは大変なので、宣言を集めたファイルがあるわけです。`stdio.h` は出力関数 `printf` や入力関数 `scanf` の宣言を含みます。この先、別の機能を使うための関数を呼ぶとき、必要に応じて include を追加します。
3. C 言語での関数は次のような形をしています。

   ```
   型指定 関数名 (パラメタの宣言…) {
       本体…
   }
   ```

   「型指定」はこの関数が返す値はどの型であるかを示します。`double` は「実数型」、すぐ後で出て来る `int` は「整数型」です。そしてパラメタも 1 つずつ、それがどの型かを記述します。ここでは三角形の面積なので、2 つのパラメタとも実数で、返す値も実数です (当然)。あと、C 言語では関数本体は「{ … }」で囲むことになっています (Ruby の def…end に相当)。
4. 関数内のローカル変数もすべて、宣言が必要です。ここでは面積の計算結果を入れる変数 `s` がそうで、これも実数なので実数として宣言します。宣言の形は「型名 変数, 変数, ...;」です。変数を宣言した直後に「`= 式`」で初期値を入れることもできます (したがって、上の例でいえば「`double s = (w * h) / 2;`」とも書けます)。

[2] C89 まででは「`/* ... */`」型のコメントしか使えないので注意。ここでは見やすいので「`// ...`」を主に使います。
[3] 「`#`」で始まる命令は行の先頭文字が「`#`」である必要があるので注意。

5. 次は計算ですが、C の式の書き方は Ruby とおおむね同様です (「**」演算子は無い)。ただし、Ruby では 1 行に複数の文を書く時だけ間に「;」を入れて区切っていましたが、C では個々の文の終わりに「;」を書くことになっています。
6. return 文の動作は Ruby と同じです。式も書けるので、s を使わず次のようにもできます。

```
double triarea(double w, double h) {
  return (w * h) / 2.0;
}
```

7. その先が main になります。実行開始はここからになりますが、main の中から triarea を呼び、その型検査を行なうために、triarea の定義が先になされている必要があります。このため、当面は main を最後に書いてください (順番を任意にする方法は後で説明)。
8. main は int を返し、そして「void」(無という意味の英語) はパラメタ無しを示します。
9. 次の行は、main で使うローカル変数の宣言で、w, h, s を実数で宣言します。
10. 次は w と h に実数を読み込みます。何を入力するか分かるように、printf でプロンプトを出力します。printf の機能は Ruby と同じです。次に、値を読み込むのに scanf を使いますが、その使い方は当面次のように覚えてください。「&」の機能については次回説明します。

```
scanf("%lf", &実数型変数);   ←実数の場合
scanf("%d", &整数型変数);    ←整数の場合
```

11. 次の行で、w と h を渡して triarea を呼び、結果を s に格納します。これは Ruby と同じです。
12. 出力の printf も Ruby と同じです。なお、変数 s を使わずに printf のパラメタに triarea の呼び出しを書いて「printf("trirarea = %g\n", triarea(w, h));」としてもよいです。
13. return 0; は main からシステムに正常終了の合図として 0 を返しています。C のプログラムでは原則として main から 0 を返してください (正しく処理ができない場合は 0 以外を返す)。[4]

続いて動かし方を説明します。プログラムのファイルは C では「.c」で終わる慣例なので、Emacs で triarea.c に打ち込んだとします (C では必ず main から動くので、1 つのファイルに 1 つのプログラムしか入れられません。ですから、プログラムに対応するファイル名をつけましょう)。次に C では**コンパイラ** (compiler) と呼ばれるプログラムでソースプログラムを翻訳し、CPU 命令の列に変換します。ここでは GCC(GNU C Compiler) というコンパイラを使用し、gcc というコマンドで翻訳します。GCC はエラーがなければ a.out というファイルに実行可能形式を出力するので、次のようにそのファイル名を指定して起動します (%はプロンプトのつもりなので打ち込まないこと)。

```
% gcc triarea.c  ←コンパイル。何も出力が無いなら OK
% ./a.out  ←実行
w> 7        ←入力
h> 5        ←入力
17.5        ←出力
```

上の例は実数の計算でしたがもう 1 つ「整数」で「if による枝分かれ」の例として絶対値を計算しましょう。if 文の書き方は、Ruby と書き方を対比して見て頂くのが簡単なのでそうします (図 11.3)。

```
// iabs1 --- absolute value (of integer)
#include <stdio.h>
```

[4] この 0 が返ったかどうかの情報はシェルが受け取り、その後正常かどうかに応じて処理を違えること「も」できます。ただ、普通はユーザが出力を見てどうか判断すれば済むので、この情報を使わないことの方が多いです。

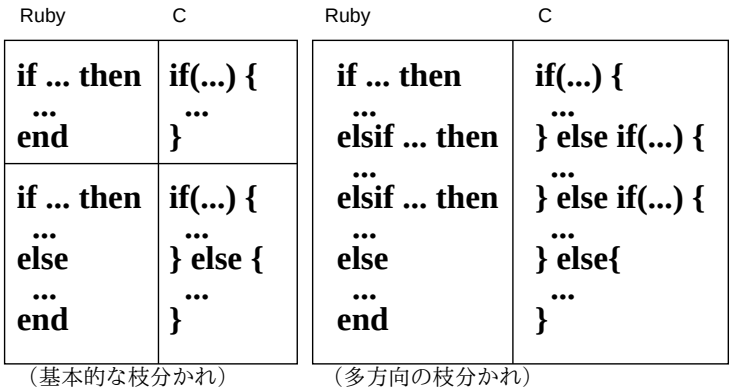


図 11.3: Ruby と C の if 文の対比

```
int iabs1(int a) {
  if(a < 0) {
    return -a;
  } else {
    return a;
  }
}
```

```
int iabs1(int a) {
  if(a < 0) {
    return -a;
  }
  return a;
}
```

```
int iabs1(int a) {
  return (a<0) ? -a : a;
}
```

```
int main(void) {
  int x, v;
  printf("x> "); scanf("%d", &x);
  v = iabs1(x);
  printf("abs = %d\n", v);
  return 0;
}
```

iabs1 は整数を 1 個受け取り、整数を返します。3 通りの書き方を例示しました (3 番目は Ruby の if-then-else 式に相当。すぐ下の 3 項演算子のところを参照のこと)。実行のようすを示します。

```
% gcc iabs1.c
% ./a.out
x> -3
3
% ./a.out
x> 5
5
```

説明が長くなったので、以下の演習で C のプログラムを書くときの「テンプレ」を掲載します。

```
// 説明…              ←コメント行で何のプログラムか書く
#include <stdio.h>  ←あと「#include <stdbool.h>」の行も必要かも
本題の関数 (Ruby でプログラムとして書いていたもの)
int main(void) {
  変数の宣言…
  変数に値を入力…
  本題の関数を呼び出す…
```

```
    結果を printf で出力…
    return 0;
}
```

演習 1 例題のうち好きな方をそのまま打ち込み実行しなさい。実行できたら、プログラムの一部をわざと色々に壊してコンパイルし、エラーの出かたを体験しなさい。OK なら次の課題をやりなさい。その場合、「`cp triarea.c max2.c`」のように課題ごとに先に作ったプログラムのファイルをコピーしてから修正すると (`main` はほぼ同じなので) 楽だと思います。

a. 円錐の底面の半径と高さを与え、体積を出力する。
b. 実数 x を与え、その平方根を出力する。
c. 実数 x を与え、その 8 乗 (または 7 乗または 6 乗) を出力する。
d. 整数を 2 つ与え、その大きい方を出力する。
e. 整数を 2 つ与え、その小さい方を出力する。
f. 整数を 3 つ与え、その最大値を出力する (または 4 つの最大でもよい)。
g. 整数を 3 つ与え、その最小値を出力する (または 4 つの最小でもよい)。
h. 整数を 3 つ与え (すべて異なる値とする)、中央値を出力する。
i. 正の整数 n を与え、n の階乗を出力する。
j. 正の整数 n を与え、2^n を出力する。
k. 正の整数 $n, r(n \geq r)$ を与え、${}_nC_r$ を出力する。
l. 正の整数 a, b を与え、それらの最大公約数を出力する。
m. その他、自分が面白いと思う計算を行う関数を作って行なえ。

わざと多くを Ruby の初期の演習と同じにしましたが、どうでしょうか。階乗、べき乗、組合せの数、最大公約数については、再帰を想定していますが、この後出て来るループを使ってもよいです。

あと、**平方根** (square root) は **sqrt**(x) で計算できますが、これを使うには冒頭に「`#include <math.h>`」を追加する必要があり、また次の実行例のように `gcc` コマンドの末尾にオプション `-lm` を追加してください (ちなみに桁数を増やすため `printf` の書式文字列は「`"%.20g\n"`」にしています)。

```
% gcc sqrt1.c -lm
% ./a.out
x> 2
1.4142135623730951455
```

なお、`math.h` 取り込みと `-lm` 指定を行うことで、次の**数学関数**が使えるようになります。

- `sqrt(`x`)` — 平方根, `cbrt(`x`)` — 立方根、`pow(`x`,`y`)` — べき乗 x^y (実数)
- `abs(`x`)` — 絶対値 (整数)、`fabs(`x`)` — 絶対値 (実数)
- `exp(`x`)` — e^x、`log(`x`)` — $\ln x$、`log10(`x`)` — $\log_{10} x$
- `sin(`x`)`、 `cos(`x`)`、`tan(`x`)` — $\sin, \cos, \tan$
- `asin(`x`)`, `acos(`x`)`, `atan(`x`)`, `atan2(`y`,`x`)` — 逆三角関数で、最後のは $\arctan \frac{y}{x}$ を計算し、x が 0 のとき y の正負に応じ $\pm\frac{\pi}{4}$(± 90 度) を返すので便利。

11.2.5 C 言語の演算子 exam

Ruby については演算子をまとめて説明しませんでしたが、C についてはここで主なものをまとめて説明してしまいます (説明がややこしいものは後で必要なところで追加します)。演算子には結び付

表 11.1: C 言語の主要な演算子 (優先順位順)

種別	演算子
単項演算子	++ (増加)、-- (減少)、+ (プラス符号)、- (符号反転)、~ (ビット反転)、! (論理否定)
2 項演算子	*、/、% (乗算、除算、剰余)
2 項演算子	+、- (加算、減算)
2 項演算子	<<、>> (左シフト、右シフト)
2 項演算子	>、>=、<、<= (比較演算子)
2 項演算子	==、!= (等しい/等しくない)
2 項演算子	& (ビット毎 and)
2 項演算子	| (ビット毎 or)
2 項演算子	&& (論理 and)
2 項演算子	|| (論理 or)
3 項演算子	x?y:z (if x then y else z end)
2 項演算子	= (代入)、+=、-=、*=、/=、%=、&=、|= (複合代入)
2 項演算子	, (順次評価)

きの強さがあります (たとえば a * b + c は (a * b) + c ですから、* は-より**優先順位**が高い、と言います。表 11.1 に、主要な演算子を優先順位順に記します。

増加/減少演算子は C 言語における発明の 1 つで、「値を 1 増やす/減らす」専用の演算です。たとえば「++i」とすると、変数 i の値が 1 増えます。プログラムで 1 増やす/減らすことは多いので結構役に立ちます。さらに特異な点として、これらの演算子は後置と前置の両方で使えます。その違いは「増減する前の値/増減した後の値」が式の値となることです。

```
int i, j, k;
i = 10; // i は 10
j = i++; // i は 11 になるが、j は増やす前の 10 が入る
k = ++i; // i は 12 になり、k は増やした後の 12 が入る
```

次に、~、&、|はビット毎演算です。多くの環境では int は 32 ビットの 2 の補数表現で整数を表しますが、それをビットの列とみなしてビット毎に NOT、AND、OR をとります。さらに<<、>>はシフト演算で、左側の項をビットの列とみなして右側で指定した値だけずらします (空いた場所には 0 のビット[5] が入ってきます)。これらは整数の値専用です。

そして論理演算や比較演算は Ruby と同様です。なお、C では「はい」は 1、「いいえ」は 0 で表すので、これらの演算は 0 または 1 を返します。さらに、&&は左の項が 0 なら右の項は計算せずに 0 を返しますし、||も左の項が 1 なら右の項は計算せずに 1 を返します。

しかし「0」「1」では分かりづらいため、多くのプログラマが型名 bool、「はい」の値 true、「いいえ」の値 false を自己流で定義してきました。それもよくないということで、C99 からは「#include <stdbool.h>」でヘッダファイルを取り込むことで true/false(論理値)、bool(型名) が使えます。[6]

次に代入=は、「右辺の値を左辺の変数に入れる」演算であり、入れた値が結果となります。代入には+=(x += 1 は x = x + 1 と同等) など演算と代入がくっついたものが一通りあります。

最後に「,」は文が書けないところで順番に実行したいときに使われます。たとえば「x = 1, y = 2」は変数 x に 1、y に 2 を入れ、右の項の値 2 が全体の値となります。

[5]符号つき整数では符号ビットのコピー

[6]C89 まででは自分で「#define bool int」「#define true 1」「#define false 0」などと定義してください。

11.2.6 繰り返しの構文 exam

if 文については最初の例題のついでに説明してしまいましたが、繰り返しの構文についてここで説明しておきます。まず **while ループ**については Ruby と同等で、書き方が少し違うだけです。

```
while(条件) {
  ...
}
```

問題は **for ループ**で、C 言語はこれがとっつきにくいので定評 (?) があります。そして、Ruby の `times` や `step` のようなメソッドも無いので、**計数ループ**にはこれを使うしかありません。具体的には、C の for ループは図 11.4 のように while ループを書き換えたものになっています。

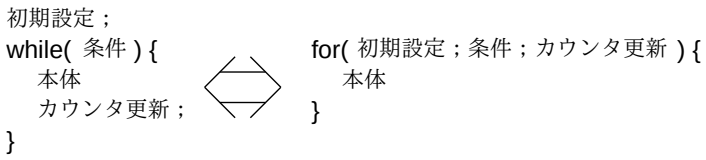


図 11.4: C 言語における for 文の意味づけ

なんだか分かりにくいですが、通常は計数ループとして「`i` に 0(または 1) を入れ」「`i` がいくつ未満 (以下) の間」「`i` を 1 増やす」という形なので「`for(i = 0; i < n; ++i) …`」のようにします。これを使った例題を示しましょう。1 から指定した整数までの数を打ち出すというものです。

```
// times1 --- count from 1 to specified number.
#include <stdio.h>
#include <math.h>
void times1(int n) {
  int i;
  for(i = 1; i <= n; ++i) {
    printf("%3d", i);
    if(i % 20 == 0 || i == n) { printf("\n"); }
  }
}
int main(void) {
  int n;
  printf("n> "); scanf("%d", &n);
  times1(n);
  return 0;
}
```

返値の型に `void` を指定すると、返す値が無いことを表します。なぜ無いかというと、関数 `times` は「数を打ち出す動作」が目的で、返す値は無いですから。そして、呼び出すところでも受け取る値は無いので、単に関数呼び出しだけを書いています。

では実行のようすを見ます。`printf` で改行をしないので横に並んで出力されていますが、くっつくと見にくいので「`"%3d"`」という書式指定を使って 3 文字幅にそろえています。また、そのままだとずっと横に長くなってしまうので、if 文を使って「20 の倍数または最後」の時に改行しています。

```
% gcc times1.c
% ./a.out
n> 50
```

```
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50
```

このように for は書き方が面倒ですが、逆に任意の条件やカウンタ更新が書けるので、色々な反復が行なえます。少し例を示します。

```
for(i = 1; i < 20; i += 2) --- 1, 3, 5, 7, ..., 19
for(i = 10; i > 0; --i) --- 10, 9, 8, ..., 1
for(i = 1; i < 1000; i *= 2) --- 1, 2, 4, 8, ..., 512
```

while、for とも途中で繰り返しを抜けるのに「`break;`」という文が使えます (Ruby と同じ)。途中で繰り返しの残りを飛ばして次の周回に進むには「`continue;`」です (Ruby の `next`)。[7]

演習 2 上のループの例題を動かしなさい。動いたら、次のものをやってみなさい。

a. 1 から 99 までの数を順に打ち出すが、3 の倍数または 3 がつく数のときはかわりに aho と打ち出す (ナベアツ)。99 の代わりに最大の値を指定できるようにしてもよい。
b. 1 から 99 までの数を順に打ち出すが、3 の倍数なら fizz、5 の倍数なら buzz、両方の倍数なら fizzbuzz を数の代わりに打ち出す (fizzbuzz 問題)。
c. 正の整数 n を受け取り、n 以下の素数を順に打ち出す (論理値を返したり変数に入れるときは`#include <stdbool.h>`を指定する。出力は整数の 0 か 1 なので`%d` で)。
d. 正の整数 n を受け取り、n から 1 までを 1 つずつ小さくなる順に打ち出す (幅をそろえたり適切に改行できるとなおよい。以下同様)。
e. 正の整数 n を受け取り、0 1 0 1…と交互に n 個打ち出す。(ヒント: n を 2 で割った余りは 0 または 1)。
f. 正の整数 n を受け取り、1 0 1 0…と交互に n 個打ち出す。(ヒント: n を 2 で割った余りは 0 または 1)。
g. 正の整数 n を受け取り、1, 3, 9, 27, … 3^n を打ち出す。
h. 九九の表を打ち出す。
i. その他、自分の面白いと思う計算をループを使って行なえ。

11.3 $f(x) = 0$ の求解

11.3.1 数え上げによる求解

C の書き方の話ばかりではつまらないので、こんどは関数 $f(x)$ について、$f(x) = 0$ を満たす x を求めるという問題、つまり **1 変数方程式の求解**を取り上げます。これも解析的に解けなくても、次の条件が満たされていればプログラムで解を求めることができます。

> ある区間 $[a, b]$ において、$f(x)$ が単調増大、連続、かつ $f(a) < 0$、$f(b) > 0$ となるような a、b が分かっている

a でマイナス、そこからなめらかに増えていって、b でプラスになっているのなら、その間のどこかに解があるわけですから、それを求めればよいわけです。

たとえば「$N(> 1)$ の平方根を求める」ことを考えます。$f(x) = x^2 - N$ とすれば、$f(0) < 0$、$f(N) > 0$ なのでここで説明する方法で解を求められます (そしてそれが N の平方根なわけです)。

では具体的にどうやったら $f(x) = 0$ の解が求まるでしょうか? たとえば、次の方針はどうでしょう?

[7] for 文ではこの場合、カウンタ更新に進みます (次の周回でカウンタが増えないと不便)。この点で、図 11.4 に示した「while と for の書き換え」は完全に同等ではありません (while で残りをスキップした場合すぐ次の条件テストに進む)。

小さい値 d を決めて、a から初めて $f(a+d)$、$f(a+2d)$、$f(a+3d)$、$\cdots$ を求めて行く。
はじめてその値が 0 以上になったところが解である。

これで確かに「誤差 d で」解が求まります。これを**数え上げ** (enumeration) 法と呼びます。

演習 3 数え上げ法によって平方根を求める C プログラムを作成しなさい。精度をあげた時にどれくらいまで実用になるか検討しなさい。

11.3.2 区間 2 分法

数え上げ法はコンピュータらしいとは言えますが、いかにも効率が悪そうです。そこで端からちょっとずつ計算するかわりに、a と b の中間の値を計算するように、次の方針を考えます。

$c=\frac{a+b}{2}$ を求め、$f(c)$ を計算する。もしも $f(c)<0$ であれば、解がある範囲は区間 $(c,b]$。
そうでなければ、解がある範囲は区間 $[a,c]$ とわかる。そこでこのどちらかに応じ、a、b いずれかを c で置き換え、同様に繰り返すことを、$|b-a|$ が十分小さくなるまで行なう。

これは 1 ステップごとに区間を 2 つに分けるため、**区間 2 分法** (binary search method) と呼びます。

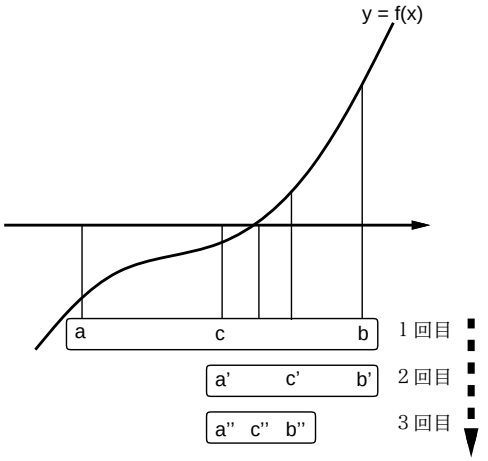


図 11.5: 区間 2 分法による求根

$2^{10}=1024$ ですから、区間を半分にすることを 10 回繰り返すと区間の幅はおよそ 1000 分の 1、40 回繰り返すとおよそ 1 兆分の 1 になります。言い換えれば、その精度で解が求まるわけです。

演習 4 区間 2 分法によって平方根を求める C プログラムを作成しなさい。必要と思われる精度にしたとき、繰り返し回数がいくつになるか検討しなさい。

11.3.3 ニュートン法

ニュートン法 (Newton's method) は、万有引力の発見者ニュートンに由来する方法で、[8] 適当な近似値 r から始め、その近似値を改良していくことで解に到達します。具体的には、$f(x)$ の $x=r$ における接線を求め、接線と X 軸が交わる点の X 座標を新たな r とし、これを反復していきます。その i 回目の値を r_i と書き、各回の計算内容を**漸化式** (recurrence formula) として表しましょう。

具体的にやってみましょう。$x=r_i$ の時の接点の座標は $(r_i, f(r_i))$、そこでの接線の傾きは $f'(r_i)$ (もちろん関数は微分可能でないといけません)。

$$\frac{f(r_i)}{r_i - r_{i+1}} = f'(r_i)$$

[8] 彼は微積分学の発明者の一人でもあります

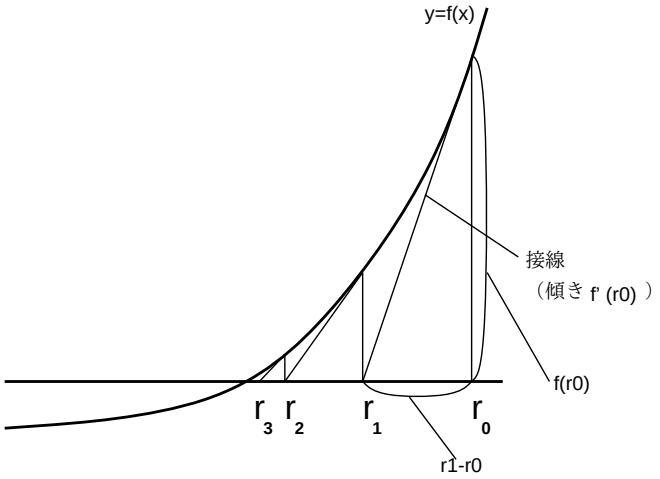


図 11.6: ニュートン法による求解

より、

$$r_{i+1} = r_i - \frac{f(r_i)}{f'(r_i)}$$

となります。$f(x) = x^2 - n$ の場合、$f'(x) = 2x$ より、

$$r_{i+1} = r_i - \frac{{r_i}^2 - n}{2r_i} = \frac{r_i}{2} + \frac{n}{2r_i}$$

となります。そこで $r_0 = N$ とおき、$r_1, r_2, \cdots$ を計算していくと、その値は $\sqrt{N}$ に**収束** (converge) していくわけです。一般にこのような、近似値を反復によって改良していく方法を**反復解法** (iterative method) と呼びます。反復の結果、値がほとんど変化しなくなったら、つまり $|r_{i+1} - r_i| < \epsilon$ となったら収束したこととし、そこでの近似値を解とするわけです。

なお、収束する値が大きい値であるような計算をする場合は、**絶対誤差** (absolute error) ではなく**相対誤差** (relative error) に基づいて収束を判定するのがよいかもしれません。その場合は反復をやめる条件は次のようになります。

$$\left| \frac{r_{i+1} - r_i}{r_i} \right| < \epsilon$$

ニュートン法は収束すれば高速なことで知られていますが、収束しない場合もあります。平方根の計算の場合は、最初の近似値として N から始めれば問題ありません。

演習 5 ニュートン法によって平方根を求める C プログラムを作成しなさい。必要と思われる精度にしたとき、繰り返し回数がいくつになるか検討しなさい。(ヒント: 繰り返しごとに現在の近似値を書き出すのでもよいですね。)

演習 6 C 言語で書いてみたいと思う自分にとって興味深い題材をプログラムとして作成しなさい。

本日の課題 11A

「演習 1」または「演習 2」で動かしたプログラム (どれか 1 つでよい) を含むレポートを提出しなさい。プログラムと、簡単な説明が含まれること。アンケートの回答もおこなうこと。

Q1. 強い型の言語、とくに C 言語についてどう思いましたか。

Q2. 言語が異なるとプログラミングの方法も異なると思いますか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 11B

「演習 1」〜「演習 6」の (小) 課題から「10 個以上」選択してプログラムを作り、レポートを提出しなさい。ただし、演習 3 以降が含まれる場合は「1 個以上」でよい。プログラムと、課題に対する報告・考察 (やってみた結果・そこから分かったことの記述) が含まれること。アンケートの回答もおこなうこと。

Q1. C 言語でプログラムが書けるようになりましたか。

Q2. C と Ruby はどのように違うと感じていますか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

11.4 付録: C 言語の文法

C 言語の書き方の規則をまとめておきます。ここで用いている記法は **BNF** (Backus Normal Form) と呼ばれるもので、次のような規則によっています。

- A ::= B は「左辺を右辺で定義する」
- A | B は「A または B のいずれかである」
- [A] は「A があってもなくてもよい」
- A... は「A が 0 個以上並んでいる」
- A,.. は「A が 1 個以上カンマで区切られて並んでいる」

以下の文法は読みやすさのため、かなり簡略化してあります。

```
C プログラム ::= ( 変数定義 | 構造体定義 | 関数宣言 | 関数定義 )...
変数定義 ::= 型指定 ( 宣言子 [ = 式] ),..  ;
宣言子 ::= 変数名 | * 宣言子 | 宣言子 [] | ( 宣言子 )
構造体定義 ::= struct 名前 { フィールド...  } ;
フィールド ::= 型指定 宣言子,..  ;
型指定 ::= 型修飾...  型名 | 型修飾...  struct 名前
型修飾 ::= extern | static | unsigned | long | short
関数宣言 ::= 型指定 関数名 ( [ ( 型指定 宣言子 ),..  ] ) ;
関数定義 ::= 型指定 関数名 ( [ ( 型指定 宣言子 ),..  ] ) { 文...  }
文 ::= 変数定義 | 式 ; | if 文 | while 文 | for 文 | switch 文
    | return [ 式 ] ; | break ; | continue ; | { 文...  }
if 文 ::= if ( 式 ) 文 [ else 文 ]
while 文 ::= while ( 式 ) 文
for 文 ::= for ( ( 変数定義 | 式 ) ; 式 ; 式 ) 文
switch 文 ::= switch ( 式 ) { ラベル文… }
ラベル文 ::= case ( 整数定数 | 文字定数 ) :  | default :  | 文
式 ::= 因子 | 式 演算子 式 | 演算子 式 | 式 演算子 | 式 ?  式 :  式
因子 ::= 変数名 | 定数 | 因子 [ 式 ] | 因子 .  変数名 | 因子 -> 因子 | ( 式 )
定数 ::= 整数定数 | 実数定数 | 文字列定数 | 文字定数
```

代入はないの? と思ったかも知れませんが、既に述べたように、代入は演算子の 1 つになっています。演算子については表 11.1 を参照のこと。

＃12　さまざまな型+動的計画法

今回は次の内容を取り上げます。

- C のさまざまな型、ポインタ型、配列型
- 動的計画法 (経験者向け)

12.1　前回演習問題の解説

12.1.1　演習 1 — 簡単な計算

最初の「円錐の体積」のみプログラム全体を示します。「`#define` 名前 文字列」は、指定した名前を別の文字列に置き換えられてコンパイルする機能で、これで円周率に PI という名前をつけました。

```
#include <stdio.h>
#include <stdlib.h>
#define PI 3.14159265358979
double cornvol(double r, double h) {
  return r*r*PI*h / 3.0;
}
int main(void) {
  double r, h, v;
  printf("r> "); scanf("%lf", &r);
  printf("h> "); scanf("%lf", &h);
  v = cornvol(r, h);
  printf("corn volume = %g\n", v);
  return 0;
}
```

中身ですが、`cornvol` は実数 `r`、`h` を受け取り、$\frac{1}{3}\pi r^2 h$ を計算して返します。`main` は実数 `r` と `h` を読み込み、`cornvol` にこれらを渡して計算結果を `v` に受け取り、最後に `printf` で出力します。

`main` はすべて同様のパターンなので、以後は関数本体のみ示します。解答例のように、「型指定 変数名 = 式, …;」の形で変数宣言の後ろに初期値を指定して値を入れることもできます。数学関数を使う場合は`#include <math.h>`とコンパイル時の`-lm` 指定が必要です。また、後半のものは再帰版とループ版の両方を掲載しています。

```
double calcsqrt(double x) {
  return sqrt(x);
}
double power8(double x) {
  double x2 = x*x, x4 = x2*x2, x8 = x4*x4;
  return x8;
}
int imax2(int a, int b) {
  if(a > b) { return a; }
```

```
  return b;
}
int imax2(int a, int b) {
  if(a > b) { return a; }
  return b;
}
int imin2(int a, int b) {
  if(a < b) { return a; }
  return b;
}
int imax3(int a, int b, int c) {
  return imax2(a, imax2(b, c));
}
int imin3(int a, int b, int c) {
  return imin2(a, imax2(b, c));
}
int imid3(int a, int b, int c) {
  int max = imax3(a, b, c), min = imin3(a, b, c);
  if(min < a && a < max) { return a; }
  if(min < b && b < max) { return b; }
  return c;
}
int fact(int n) {  // recursive version
  if(n == 0) { return 1; }
  else { return n * fact(n-1); }
}
int fact(int n) {  // loop version
  int i, result = 1;
  for(i = 1; i <= n; ++i) { result *= i; }
  return result;
}
int pow2(int n) {  // recursive version
  if(n == 0) { return 1; }
  else { return 2 * pow2(n - 1); }
}
int pow2(int n) {  // loop version
  int i, result = 1;
  for(i = 0; i < n; ++i) { result *= 2; }
  return result;
}
int comb(int n, int r) {  // recursive version
  if(r == 0 || r == n) { return 1; }
  else { return comb(n-1, r) + comb(n-1, r-1); }
}
int comb(int n, int r) {  // loop version
  int i, result = 1;
  for(i = 1; i <= r; ++i) { result = result * (n-r+i) / i; }
```

```
  return result;
}
int gcd(int x, int y) {  // recursive version
  if(x == y)     { return x; }
  else if(x > y) { return gcd(x-y, y); }
  else           { return gcd(x, y-x); }
}
int gcd(int x, int y) {  // loop version
  while(x != y) {
    if(x > y) { x -= y; } else { y -= x; }
  }
  return x;
}
```

12.1.2 演習 2

true、false、bool を使うものは#include <stdbool.h>が必要です。

```
void nabeatsu(void) {
  int i;
  for(i = 1; i <= 99; ++i) {
    if(i%3 == 0 || i/10 == 3 || i%10 == 3) { printf("aho\n"); }
    else { printf("%d\n", i); }
  }
}
void fizzbuzz(void) {
  int i;
  for(i = 1; i <= 99; ++i) {
    if(i % 15 == 0)     { printf("fizzbuzz\n"); }
    else if(i % 3 == 0) { printf("fizz\n"); }
    else if(i % 5 == 0) { printf("buzz\n"); }
    else                { printf("%d\n", i); }
  }
}
bool isprime(int n) {
  int i;
  for(i = 2; i < n; ++i) {
    if(n % i == 0) { return false; }
  }
  return true;
}
void primes(int n) {
  int i;
  for(i = 2; i <= n; ++i) {
    if(isprime(i)) { printf("%d\n", i); }
  }
}
void countdown(int n) {
```

```
  int i;
  for(i = n; i > 0; --i) {
    printf("%4d", i);
    if((n-i+1) % 10 == 0 || i == 1) { printf("\n"); }
  }
}
void alt01(int n) {
  int i;
  for(i = 0; i < n; ++i) {
    printf("%4d", i%2);
    if((i+1) % 10 == 0 || i == n-1) { printf("\n"); }
  }
}
void alt10(int n) {
  int i;
  for(i = 1; i <= n; ++i) {
    printf("%4d", i%2);
    if(i % 10 == 0 || i == n) { printf("\n"); }
  }
}
void printpow3(int n) {
  int i, v = 1;
  for(i = 0; i <= n; ++i) {
    printf("%4d", v); v *= 3;
    if(i+1 % 10 == 0 || i == n) { printf("\n"); }
  }
}
void print99(void) {
  int i, j;
  for(i = 1; i <= 9; ++i) {
    for(j = 1; j <= 9; ++j) { printf("%4d", i * j); }
    printf("\n");
  }
}
```

12.1.3 演習 3〜5 — 3 つの方法による平方根の計算

いずれも上にある平方根の関数に置き換えられるべきなので、必要な定義と関数本体のみ示します。まず数え上げ。

```
// enum1 --- calculate square root using enumeration
#include <stdio.h>
#include <stdlib.h>
#define N 1000000
double enumsqrt(double x) {
  double dx = x / N;
  int i;
  for(i = 0; i < N; ++i) {
```

```
      double t = i * dx;
      double y = t*t - x;
      if(y >= 0) { return t; }
    }
    return x;
  }
  int main(void) {
    double x;
    printf("x> "); scanf("%lf", &x);
    printf("sqrt = %.20g\n", enumsqrt(x));
    return 0;
  }
```

実行例を示しますが、1000000 くらいではあまり精度よくないです。

```
% ./a.out
x> 2
sqrt = 1.4142139999999998601
```

以下では同じなので include と main を省略します。区間 2 分法ですが、回数が分かるように途中経過を印字することもできるようにします (コメントアウトしてあります)。

```
#define EPSILON 1e-10
double binsqrt(double x) {
  double a = 0.0, b = x;
  while(b - a > EPSILON) {
    double c = 0.5 * (a + b);
    double y = c*c - x;
//  printf("%.20g\n", c);
    if(y >= 0) { b = c; } else { a = c; }
  }
  return a;
}
(ここに main)
```

コメントアウトを外して実行すると次のようになります。遅くはないですが、それなりに反復が必要です (35 回程度)。

```
% ./a.out
x> 2
1
1.5
1.25
1.375
1.4375
1.40625
1.421875
1.4140625
1.41796875
(途中略)
```

```
1.4142135621514171362
1.414213562267832458
1.4142135623260401189
1.4142135623260401189
%
```

最後はニュートン法ですが、x0 が 1 つ前の r_i、x1 が次の r_{i+1} で、ループ内で計算につれて順ぐりにコピーしています。これらの値が減少していくことを前提に while の条件を書いているので、ループに入る前で x0 を n、x1 を $2n$ にしています。

```
#define EPSILON 1e-10
double ntnsqrt(double n) {
  double x0 = n, x1 = 2*n;
  while(x1 - x0 > EPSILON) {
    x1 = x0; x0 = 0.5*x1 + 0.5*n/x1;
//  printf("%.20g\n", x0);
  }
  return x0;
}
```

実行してみると、ずっと周回数が少なくて済むことがわかります。

```
% ./a.out
x> 2
1.5
1.4166666666666665186
1.4142156862745096646
1.4142135623746898698
1.4142135623730949234
1.4142135623730949234
%
```

12.2 C 言語のさまざまな型

12.2.1 アドレスとポインタ型 exam

ここまでではC言語の整数型 (int) と実数型 (double) について説明しました (そして論理型は無いことも)。あと文字型などもあるのですが、その前にここで、C言語の特徴的な機能の 1 つである、変数の**アドレス** (address、メモリ上の番地) を取得する機能について取り上げます。変数の場所を「指す」ことからアドレス値のことを**ポインタ** (pointer) とも呼びます。ポインタを扱うためには、次の 2 つの演算子を使います。

- `&`x — 変数 x のアドレスを取得して返す
- `*`p — アドレス値 p にある変数をアクセス (**参照たどり**、dereference)

実際には変数には int 型のもの、double 型のものなど色々ありますから、ポインタ値も「int 型の変数を指すポインタ」など指す先の型で区分しないといけません。変数宣言において、変数名の直前に「*」を付けることでその変数はポインタ型であることを示します。例として、値を 2 つ、変数のアドレスを 2 つ渡すと、その 2 つの変数に値が「小さい順に」入るメソッド sort2 というのを作って呼び出してみます。

```
// sort2 --- sort 2 numbers
#include <stdio.h>
#include <stdlib.h>
void sort2(double a, double b, double *p, double *q) {
  if(a < b) { *p = a; *q = b; }
  else      { *p = b; *q = a; }
}
int main(void) {
  double a, b, x, y;
  printf("a> "); scanf("%lf", &a);
  printf("b> "); scanf("%lf", &b);
  sort2(a, b, &x, &y);
  printf("smaller = %g, larger = %g\n", x, y);
  return 0;
}
```

main から見ましょう。まず数値を 2 つ入力しますが、その 2 つと、変数 x、y のアドレスを渡して sort2 を呼びます。すると小さい順に値が格納されるので、それを出力します。

```
% ./a.out
a> 6.1
b> 3.2
smaller = 3.2, larger = 6.1
% ./a.out
a> 7.7
b> 1.5
smaller = 1.5, larger = 7.7
```

この様子を図 12.1 に示しました。コンピュータ内ではすべてのデータは**主記憶**に格納されていますが、主記憶のすべての場所には、**アドレス** (番地) がつけられています。実数値は 8 バイトの大きさなので、実数を入れる変数が並んでいた場合、それらの番地も 8 バイトきざみになっています。そこで、変数 x が仮に 1F80 番地、y が 1F88 番地 (いずれも 16 進) だったとしましょう。main から sort2 を呼ぶ時に、a や b はその変数に入っている値 (6.1 や 3.2) が渡されてパラメタに入りますが、&x や &y はこれらの変数のアドレスが取られて渡され、パラメタ p や q に入ります。そして、sort2 の中で「*p = ...」「*q = ...」の代入を行なうと、p や q に入っている番地、つまり 1F80 番地や 1F88 番地に値が入ります。というわけで、main 側の x や y が書き換えられるわけです。

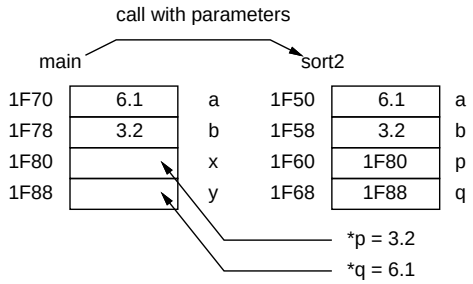


図 12.1: アドレスをパラメタとして渡す

このように、「&」を使って番地を渡し、「*」を使ってその番地にアクセスすることで、関数から複数の変数を書き換えられます (return だと 1 つしか値が返せませんでした)。

さて、お待たせしました。scanf で値を読み込み、その値を変数に入れてもらうのにも、この機能を使っていたわけです。scanf は printf とペアになった関数で、その第 1 引数として「どんな値を読み込むか」を指定します。そして、その読み込む先は「変数のアドレス」を指定するのでした。

しかし、整数を読み込む指定が「"%d"なのは出力と同じなので分かりますが、実数の方の書式はなぜ"%lf"なのでしょう? それは、変数に複数のビット数のものがある関係です。たとえば int は 32 ビットですが、もっとビット数が必要な場合は long という整数型も使えます (64 ビットになります)。これの入力や出 力は"%ld"という書式指定を使います (long の l)。次に実数 double は 64 ビットですが、メモリを節約したい場合用にビット数の少ない実数型である float も使えます (32 ビットになります)。その入力に"%f"を使うので、それより長い double は"%lf"なのです。だったら出力は? と言われるでしょうけれど、C ではパラメタに渡す実数は標準で double を使うので、float の式を渡しても double に変換して渡されます。なので出力はどちらも"%f"なのです。

12.2.2 配列型とポインタ演算 exam

数値変数とポインタまで来ましたが、やはりプログラミングには配列がないと不便です。C では配列変数の宣言は次のような形で行ないます。

```
int arr[100]; double vec[1000];
```

一般には「型名 変数名 [要素数];」という形です。また、「;」の前に「= { 値, 値, … }」という形で初期値を指定することができます。そして添字を指定したアクセスのしかたは「一見」Ruby と同じです。例を見てみましょう。main で初期値を指定した配列を用意しますが、それを piarray というメソッドに渡して中身を打ち出します。

```
// array1 --- array demonstration
#include <stdio.h>
void piarray(int n, int a[]) {
  int i;
  for(i = 0; i < n; ++i) {
    printf(" %2d", a[i]);
    if(i % 10 == 9 || i == n-1) { printf("\n"); }
  }
}
int main(void) {
  int a[24] = {1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8};
  piarray(24, a);
  return 0;
}
```

main から見ましょう。配列 a を要素数 24 で宣言し、24 個の値を初期値として入れています。なお、配列 a の大きさは右辺の値の数から分かるので「int a[] = { ... };」でも OK です。そして、その要素数 24 と配列を渡して piarray を呼び出します。

piarray の中では、個数と配列を受け取り、順次打ち出します。printf の書式では、空白 1 個と、あと整数を最低 2 文字幅で出力しています。また前回やったのと似ていますが、添字が 9, 19, 29, … のときと「最後」は改行します (こんどは添字なので 0 から始まるのに注意)。では実行のようす。

```
% ./a.out
  1  2  3  4  5  6  7  8  1  2
  3  4  5  6  7  8  1  2  3  4
  5  6  7  8
```

Ruby と全然おんなじで問題ないと思ったかも知れませんが、そうではないのです。C では基本的に、変数は存在範囲に入ったところで領域ができますが、上の中かっこで囲んだリストはそのときの「初期化」のための値です。ですから、実行の途中で好きな配列値を入れ直す等はできません (個別の要素への代入はできます)。[1]

第 2 に、`a[i]` という**添字**つきの式が問題です。C では「T `*p;`」であるようなポインタ値に対して `p[i]`(`i` は整数の式) という書き方ができ、その意味は`*(p + i)` と等しい、と定めています。

そして `p + i` とは? これは**ポインタ演算**と呼ばれ、`p` の指している要素から `i` 個ぶん先 (マイナスなら手前) の要素のアドレスとなります。1 つの要素のサイズは型 T によって違いますから、アドレスでいうと `p` に `sizeof(`T`)*i` を加えた値になります。ちなみに `sizeof(`T`)` は T 型の変数の占めるバイト数で、`sizeof(int) == 4`、`sizeof(double) == 8` 等となります。

そして、上の例での `a` のような配列名はその要素型のポインタ型であり、先頭要素のアドレスを表します。なので、上の例の `piarray` に渡していたのは、実は整数へのポインタです。ただ、配列らしく書きたいので、C では「`int *a`」と書くかわりに「`int a[]`」と書いてもよいのです。`piarray` の中で `a[0], a[1], ...` を打ち出していたのは、実は`*(a+0), *(a+1), ...` を打ち出していたのですが、これはつまり配列の先頭要素、次の要素、…なので意図した通りなわけです。

ポインタ演算の使用例として、`piarray` 呼び出しを変更し、配列の「一部分を」出力してみます。

```
piarray(5, a+2); piarray(8, a+8);
```

対応する出力は次のようになります。

```
3  4  5  6  7
1  2  3  4  5  6  7  8
```

これは図 12.2 のように、配列 a の「2 つ先の要素から 5 個」「8 つ先の要素から 8 個」を出力させるようになっているわけです。

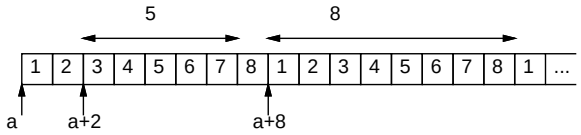


図 12.2: ポインタ演算により配列の途中を指定

演習 1 上の例題を打ち込んで実行せよ (一部を出力するところも追加して動かしてみること)。うまく動いたら、次のような関数を追加してみよ。

a. 整数配列を「後ろから順に」打ち出す関数 `void piarrayrev(int n, int a[])`。
b. 整数配列と整数値を渡し、指定した整数値が配列の何番に入っているかを返す (入っていなければ-1 を返す) 関数 `int iindex(int n, int a[], int x)`。
c. 整数配列の最大値を返す関数 `int maxiarray(int n, int a[])`。
d. 整数配列の最小値を返す関数 `int miniarray(int n, int a[])`。
e. 整数配列の合計値を返す関数 `int sumiarray(int n, int a[])`。
f. 整数配列の平均値を返す関数 `double avgiarray(int n, int a[])`。
g. 実数配列の打ち出し/後ろから順に打ち出し/最大値/最小値/ 合計値/平均値を返す関数。
h. 好きな方法で整数配列を整列する関数。テスト用に乱数が必要なら付録を参照のこと。
i. その他配列を受け取り好きな処理をする関数。

[1] C99 では配列リテラルの機能が加わったのですが、そのリテラルの存在範囲も関数内なので結局あまり便利ではありませんし、C89 までと互換性がないのでここで説明している「初期化」のみ扱います。

12.2.3 配列への入力 exam

配列をプリントする関数と組になる、配列を入力する関数も作ってみます。入力する個数と整数配列 (正確には整数へのポインタ) を渡すと、その場所以下指定個数ぶんの場所に値を入力します。

```
// arrayread --- array input
#include <stdio.h>
void piarray(int n, int a[]) {
  int i;
  for(i = 0; i < n; ++i) {
    printf(" %2d", a[i]);
    if(i % 10 == 9 || i == n-1) { printf("\n"); }
  }
}
void riarray(int n, int a[]) {
  int i;
  for(i = 0; i < n; ++i) {
    printf("%d> ", i+1); scanf("%d", a+i);  // &a[i] でも OK
  }
}
int main(void) {
  int n, a[100];
  printf("n> "); scanf("%d", &n);
  riarray(n, a); piarray(n, a);
  return 0;
}
```

riarray では、i を 0〜n-1 の範囲で変えながら、プロンプトを出力し (分かりやすさのため番号を表示しています)、a[i] に値を読み込みます。しかし「a+i」とは? これはポインタ演算で、「a から i 個ぶん先の場所のアドレス」が計算できるので、これで合っています。または「&a[i]」と書いても同じことです。実行してみましょう。

```
./a.out
n> 3
1> 11
2> 12
3> 13
 11 12 13
```

演習 2 上の例題を打ち込んで動かし、動作を確認しなさい。OK なら、以下のような配列入力プログラムを作りなさい。入力結果を表示すること。

a. 最初に個数を指定してその数だけ入力する代わりに、順番に数値を入力して最後に 0 を入れると終わり、入力した個数 (0 は含まない) を返す関数 int riarrayz(int lim, int a[])。lim は最大個数で、それより多く入力してはいけない。動かすと次のようになる。

```
1> 21
2> 31
3> 0   → 21 と 31 が 0 番目と 1 番目に入り「2」を返す
```

b. 上記と同様だが、上記では「0」が入れられないので、終わりの印になる数をパラメタで渡す int riarrayz2(int lim, int a[], int endval)。たとえば-1 を渡すと次のように。

```
input integer (-1 for end) 1> 41
input integer (-1 for end) 2> 0
input integer (-1 for end) 3> -1  → 41 と 0 が入力され「2」を返す
```

c. riarray、riarrayz、riarrayz2 の実数入力版。

d. その他自分があったらいいと思う入力関数。

e. その他ポインタやアドレスを使った面白いと思う関数。

12.3 動的計画法

12.3.1 動的計画法とは

先に**フィボナッチ数**の計算を取り上げた時、次のような再帰的定義を示し、それをそのまま再帰関数にしたのでは遅すぎる、という説明をしました。遅すぎる理由は、この定義どおりだと 1 段階の再帰ごとに自分自身を 2 回呼び出し、同じパラメタに対する値を何回も重複して実行するからです。

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

遅くなる理由である再計算を防ぐために、たとえば配列 `fib[`i`]` を用意して一度計算した値はそこに蓄え、2 回目からは計算しないでそれを持ってくる、という方法があります。一般に、関数を計算する時に、一度計算した結果を引数と一緒に覚えておいて、同じ引数に対しては覚えておいた値を返すようにすることを**メモ化** (memoization) と呼びます。

しかしそもそも、配列を使うのだったら、いちいち計算する代わりに、最大 30 番目までのフィボナッチ数だったら最初に順番に計算してしまい、それを参照するだけの方が分かりやすいはずです:

```
int i, fib[31] = { 1, 1 };
for(i = 2; i <= 30; ++i) { fib[i] = fib[i-1]+fib[i-2] }
```

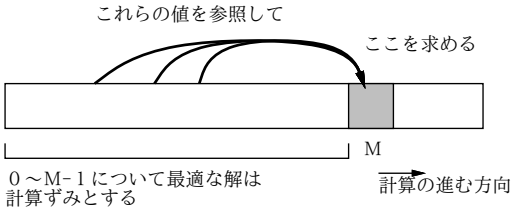


図 12.3: 動的計画法の考え方

これは図 12.3 のように、「値 $0 \sim M-1$ までについて解が求まっていれば値 M についての解もすぐ求まる」問題に対し、配列を用いて 0 から順に答えを埋めていくことで値 M に対する答えを求めていると言えます。一般にある問題に対して、その問題だけを解く代わりに小さい問題から順に全ての問題を答えを記録しつつ解くことで必要な解を求める手法のことを、**動的計画法** (dynamic programming, DP) と呼びます。なお、これは単なる 1 つの手法であり、特別に動的でも特別にプログラミングでも何でもありません。この手法を考案した人がそういう名前をつけた、というだけです。他の方法では計算量が多すぎて扱えない問題が動的計画法によって効率よく扱える場合も多くあります。

演習 3 実際に上に説明した方法で「0 番目から 30 番目までのフィボナッチ数列を打ち出す」プログラムを作りなさい。さらに、「0〜30 までのいずれかの番号を入力すると、その番号のフィボナッチ数列を出力する」プログラムも作れるとなおよい。

12.3.2 部屋割り問題

動的計画法の適用例として、次のような問題を考えてみます。

> 合宿で 1 泊料金が「1 人部屋:5,000 円、3 人部屋:12,000 円、7 人部屋:20,000 円」というホテルに泊まる。[2] 合計宿泊人数 n 人に対し、最も安い宿泊金額総計を求めよ。

この問題では、7 人部屋が非常に割安なので、7 人より少ない人数で泊まっても 7 人部屋を選んだほうがよい場合があり、最適な割り当てを求めるのは簡単ではありません。この問題に限って言えば、できるだけ多く 7 人部屋を使って、残った 1〜6 人の場合について全部の場合を検討すれば済みますが、17 人部屋とか 31 人部屋とかもあったとすると大変すぎます。

そこで動的計画法を用いる準備として、人数 n に対して最も安い値段を計算する関数 $roomprice$ を次のように定義します。

$$roomprice(n) = minimumof \begin{cases} roomprice(n-1) + 5000 \\ roomprice(n-3) + 12000 \\ roomprice(n-7) + 20000 \end{cases}$$

$minimumof$ というのは聞いたことがないと思いますが (今発明したものなので当然です)、右側の選択肢のうち一番小さい値を取る、という意味のつもりです。なお、$roomprice(n)$ は $n \leq 0$ のときは 0 であるものとします (泊まる人数が 0 以下ならお金は掛かりませんから)。

なぜこれでいいかというと、n 人で泊まる時の最も安い方法は、「$n-1$ 人で泊まる時の最も安い場合に 1 人部屋を追加する」「$n-3$ 人で泊まる時の最も安い場合に 3 人部屋を追加する」「$n-7$ 人で泊まる時の最も安い場合に 7 人部屋を追加する」のうちのどれかではあるに決まっているからです (どれであるかは計算してみないと分かりませんが)。

では、これを C プログラムにしてみましょう。大きさ `RMAX` の配列 `roomprice` を作りますが、この配列は何回も使うのでグローバル変数にします (Ruby と同様、関数の外に置けばそうなります。ただし Ruby と異なり、`$`はつけません)。そしてその配列に対し、関数 `initialize` において、1〜$RMAX-1$ までの `i` について順次、3 つの場合の最小値を求めて入れて行きます。

```
// rooms1 --- room pricing with DP
#include <stdio.h>
#include <stdbool.h>
#define RMAX 1000
int roomprice[RMAX] = { 0, };
int room1(int i) {
  return (i < 0) ? 0 : roomprice[i];
}
void initialize(void) {
  int i;
  for(i = 1; i < RMAX; ++i) {
    int min = room1(i-1) + 5000;
    if(min > room1(i-3) + 12000) { min = room1(i-3) + 12000; }
    if(min > room1(i-7) + 20000) { min = room1(i-7) + 20000; }
    roomprice[i] = min;
  }
}
int main(void) {
  int n;
```

[2] 参加者は全員同性とし、各部屋には収容人数より少ない人数でも泊まれます。どの部屋も数は十分あるものとします。

```
  initialize();
  while(true) {
    printf("input number (0 for end)> "); scanf("%d", &n);
    if(n == 0) { return 0; }
    if(n<0 || n>=RMAX-1) { printf("%d: invalid\n", n); continue; }
    printf("room price for %d => %d\n", n, roomprice[n]);
  }
}
```

initialize では先のアルゴリズムによって roomprice を順に初期化していますが、値を読み出す時には room1 という下請け関数を呼びます。この関数は、人数 i が負のときは 0 を返し、それ以外は roomprice[i] を返します。そうしないと配列の負の場所を参照してしまいますから。return の後ろにあるのは C 言語の 3 項演算子「条件 ? 式 : 式」で、機能は Ruby の if-then-else 式と同じです。

main ではまず最初に initialize を呼び、配列 roomprice を初期化します。そのあと「while(true)」は無限ループですが、その中で n に整数を読み込みます。次にそれが 0 なら return 0; により main を終わります。負または RMAX-1 以上なら範囲外なのでエラーを出力して次の周回に進みます (continue 文の働き)。いずれでもなければ人数とそのその人数のときのコストを表示し、また次の周回に進みます。では動かしてみましょう。

```
% ./a.out
input number (0 for end)> 10
room price for 10 => 32000
input number (0 for end)> 11
room price for 11 => 37000
input number (0 for end)> 12
room price for 12 => 40000
input number (0 for end)> 0
%
```

なるほど、12 人だと 7 人部屋が 2 つの方が安いわけです。

しかし、何人部屋がいくつなのかも知りたいですよね? そのためには、次の定義による値 $roomsel(n)$ も一緒に計算すればよいのです。

$$roomsel(n) = \begin{cases} 1 & (roomprice(n-1) + 5000 \ is \ the \ smallest) \\ 3 & (roomprice(n-3) + 12000 \ is \ the \ smallest) \\ 7 & (roomprice(n-7) + 20000 \ is \ the \ smallest) \end{cases}$$

この関数は、n 人のときに「最後に選んだ最適な部屋人数」を返します。選んだ部屋のリストを得るには、たとえば $roomsel(10)$ が 3 だったら、さらに $roomsel(7)$ を調べ、というふうに次々に「逆向きに」たどって行く必要があります。このため、こちらの情報のことを「トレースバック情報」と呼びます。では、先のメソッドを改造してトレースバックを記録し、金額に続いて部屋のリストを (1 つの配列として) 並べて返すようにしてみます。

```
// rooms2 --- room pricing with DP w/ traceback
#include <stdio.h>
#define RMAX 1000
int roomprice[RMAX] = { 0, };
int roomsel[RMAX] = { 0, };
int room1(int i) {
  return (i < 0) ? 0 : roomprice[i];
```

```
}
void initialize(void) {
  int i;
  for(i = 1; i < RMAX; ++i) {
    int min = room1(i-1) + 5000, sel = 1;
    if(min > room1(i-3) + 12000) { min = room1(i-3) + 12000; sel = 3; }
    if(min > room1(i-7) + 20000) { min = room1(i-7) + 20000; sel = 7; }
    roomprice[i] = min; roomsel[i] = sel;
  }
}
int main(void) {
  int n;
  initialize();
  while(true) {
    printf("input number (0 for end)> "); scanf("%d", &n);
    if(n == 0) { return 0; }
    if(n<0 || n>=RMAX-1) { printf("%d: invalid\n", n); continue; }
    printf("room price for %d => %d;", n, roomprice[n]);
    while(n > 0) { printf(" %d", roomsel[n]); n -= roomsel[n]; }
    printf("\n");
  }
}
```

これを動かすと、今度はちゃんと部屋の選択が分かります。

```
% ./a.out
input number (0 for end)> 10
room price for 10 => 32000; 3 7
input number (0 for end)> 11
room price for 11 => 37000; 1 3 7
input number (0 for end)> 12
room price for 12 => 40000; 7 7
input number (0 for end)> 0
%
```

演習 4 上の例題を打ち込んでそのまま動かしなさい (最初はトレースバック無しの簡単な方を動かし、動いてからトレースバックを追加した方が楽だと思います)。動いたら、「13 人部屋 3 万円」「17 人部屋 4 万円」の選択肢を追加して動かしてみなさい。

演習 5 「釣り銭問題」も動的計画法が使える典型的な問題です。たとえば、米国だとコインの額面が「1 ¢」「5 ¢」「10 ¢」「25 ¢」の 4 種類なので、ある金額 (¢) を与えられたとき「何枚」コインがあれば済むかを決めるのはちょっと面倒です。これも、次のように考えると動的計画法で解けます (ただし $coins(0) = 0$ と定義します)。

$$coins(c) = minimumof \begin{cases} coins(c-1)+1 & (c \geq 1) \\ coins(c-5)+1 & (c \geq 5) \\ coins(c-10)+1 & (c \geq 10) \\ coins(c-25)+1 & (c \geq 25) \end{cases}$$

これに基づき、¢ 金額を与えたとき最小コイン枚数を答えるプログラムを作りなさい。できればさらにトレースバックを追加して、具体的なコインの組み合わせも答えるようにしなさい。

演習 6 整数の列が与えられた時、その中から (とびとびに) 後の方ほど値が大きくなるような部分列を選ぶとする。そのような部分列で最も長いもの (最長増加部分列、longest increasing subsequence) の長さ (できれば列自体も) を表示するプログラムを書きなさい。たとえば列が「1, 5, 7, 2, 6, 3, 4, 9」であれば、長さ「5」、列としては「1、2、3、4、9」が解となる。

ヒント: 動的計画法の場合配列を用意し、その i 番には i 番の値が最後にある部分列の最大長を入れていきます。列自体を表示するにはトレースバックが必要になります。

演習 7 動的計画法を使って何か面白いと思うプログラムを作りなさい。

本日の課題 12A

「演習 1」または「演習 2」で動かしたプログラム (どれか 1 つでよい) を含むレポートを提出しなさい。プログラムと、簡単な説明が含まれること。アンケートの回答もおこなうこと。

Q1. C 言語のアドレスとポインタについてどう思いましたか。

Q2. C 言語の配列機能についてどう思いましたか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 12B

「演習 1」〜「演習 7」の (小) 課題から 1 つ以上を選択してプログラムを作り、レポートを提出しなさい。プログラムと、課題に対する報告・考察 (やってみた結果・そこから分かったことの記述) が含まれること。アンケートの回答もおこなうこと。

Q1. C 言語で配列を取り扱えるようになりましたか。

Q2. 動的計画法を理解しましたか。またどのように思いましたか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

12.4 付録: いくつかの補足説明

12.4.1 printf のまとめ

`printf` の使い方については Ruby の初回と同じと前回説明しましたが、再度整理しましょう。`printf` で第 1 引数として渡す文字列は**書式文字列** (format string) と呼ばれ、基本的にはそのまま出力されますが、その中の「特別な」指定があるとその箇所で指定に応じた埋め込みが行われます。

- `%%` — 「%」1 文字を出力する。
- `%d` — 対応する引数 (整数) を十進表現で出力する。
- `%x` — 対応する引数 (整数) を 16 進表現で出力する。
- `%o` — 対応する引数 (整数) を 8 進表現で出力する。
- `%f` — 対応する引数 (実数) を小数点表現で出力する。
- `%e` — 対応する引数 (実数) を仮数+指数表現で出力する。
- `%g` — 対応する引数 (実数) を値に応じ小数点か仮数+指数表現で出力。

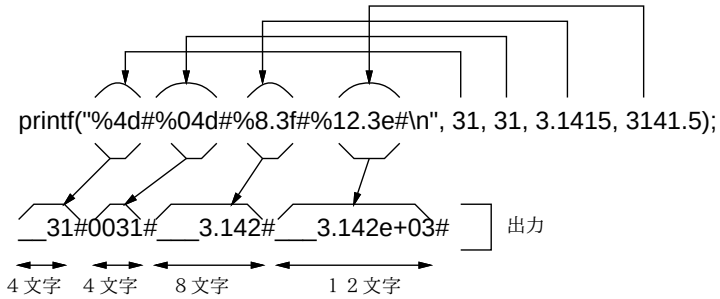


図 12.4: 書式文字列の機能

「対応する引数」というのは、書式文字列の後に指定した引数を 1 つずつ対応させます (図 12.4)。最後の g は、基本的には小数点表現したいけれど大きすぎる/小さすぎる値は指数表現に切替えてね、という指定になります。

さらに、「%」と指定文字 (d とか f とか) の間に「w」「$w.v$」「$.v$」という形の指定が追加できます。w はその値を出力する幅を指定し、出力をきれいに揃えるために使います。数値の表示が w より小さい場合は左に空白を詰めて指定幅にしますが、w の先頭に数字の「0」をつけると空白の代わりに 0 が詰められます。一方 v は実数のみで「小数点以下の表示桁数」です。そいううわけで、「"%.20g"」ではおまかせで小数点以下は 20 桁表示、ということになります。

12.4.2 変数の存在期間と可視範囲

プログラムの構造について学んだので、それと関連する重要な話題である変数の存在期間ないし**エクステント** (extent) と可視範囲ないし**スコープ** (scope) について説明しておきます。前者は「その変数がどの範囲で存在しているか」、後者は「その変数がどの範囲からアクセスできるか」を意味します。たとえば次のようなプログラムの断片を考えてください。

```
int globx;
...
int sub(int a) {
  int b = 10;
  ...
  if(...) {
    ...
    int a = 20;
    ...
  }
  return a;
}
```

グローバル変数 globx の存在期間はプログラムの実行開始から終了までずっとです。いつでも使える変数なので当然ですね。これに対し、関数 sub のパラメタ a の存在期間は sub の実行開始から終了までの間です。また、その中の局所変数 b についても (先頭で定義していますから) 同じです。ということは、これらは sub が 2 回呼び出されたとすれば、2 回「存在を開始し」「存在を終了する」ことになります。ですから、2 回目に b に前回の値が残っていることは期待できません。if の内側の a はどうでしょう。これはブロックの途中で宣言されているので、「その箇所を実行した時点からブロックを出るまで」が存在期間です。

次に可視範囲ですが、globx は (ファイルの先頭で定義したとして) プログラムのどこからでもアクセスできますから、プログラム全体が可視範囲です。ただし、もし sub のパラメタ a か局所変数 b の名前が globx だったとしたら?! そのときは、sub の範囲内で globx と書くとそちらを意味しますか

ら、sub の範囲内は可視範囲外になります。これを隠蔽ないし**シャドウ** (shadow) する、と言います。同様に、if の中の a の定義箇所以降ではパラメタ a はシャドウされています。ところが同じブロック内でも局所変数 a の定義より手前ではまだその局所変数は存在していないので、シャドウはなく、パラメタ a がアクセスできます。このシャドウの規則は整合性はあるのですが、実際にこういうプログラムを書くと勘違いを犯しやすいので、シャドウはできるだけ避けるのがよいでしょう。

12.4.3 型変換とキャスト

Ruby で整数と実数を混ぜて計算すると実数に自動変換されていましたが、C でも同様に整数は実数に自動変換が行われます。さらに C では、複数のビット数のものが混ざった場合は長いビット数に合わせられます。

では逆はどうでしょう? Ruby では実数を整数にするのに .to_i を使っていましたが、C ではメソッドではなく**キャスト** (cast) と呼ばれる専用の構文があります。その形は次の通りです。

(型名) 式

では例題を見てみましょう。

```
// cast1 --- cast demonstaration
#include <stdio.h>
int main(void) {
  double x, y = 3.1416;
  printf("x? "); scanf("%lf", &x);
  printf("%f cast to int => %d\n", x, (int)x);
  printf("address of x, y => %lx, %lx\n", (long)&x, (long)&y);
  double *p =(double*)((long)&x - 8);
  (*p) = 2.71828;
  printf("y = %f\n", y);
  return 0;
}
```

実行結果を示します。

```
% ./a.out
x? 3.1416
3.141600 cast to int => 3
address of x, y => 7ffe0923c7b0, 7ffe0923c7a8
y = 2.718280
```

まず実数 x を読み込み、整数にキャストして出力します。次に、変数 x と y のアドレスを取得し、long にキャストして 16 進で出力します (多くの実習環境ではアドレスは 64 ビットあるので long を使用する必要があります)。次に、double を指すポインタ変数 p を用意し、そこには「x のアドレスを long に変換し、8 引いて、double のポインタに再度変換して格納します。そして p の指す先を 2.71828 にすると、確かに y がその値になっています (最初 8 足していましたが、やってみると後に書いた変数の方がアドレスが小さいので修正しました)。

ただし、このようにアドレスを整数型に変換して計算というのは C 言語としては保証されていませんし、計算を間違って不正なアドレスをアクセスすると「Bus Error」「Segmententation Fault」などのエラーで強制終了されますから、注意してください (でも別に他人に迷惑を掛けることはないので、色々試すことは問題ないです)。

あと、型名の指定がよく分からなかったかも知れません。「p が整数を指すポインタ型」のとき、その宣言は「int *p」ですね。C では、その具体的な変数名 p を取り除いたもの、つまり「int*」が「整数を指すポインタ型」を意味する、という形で型名を指定するようになっています。

12.4.4 擬似乱数の使用

C 言語で擬似乱数を使う場合は「#include <stdlib.h>」「#include <time.h>」した上で次の呼び出しを使ってください。

- srand(time(NULL)) — main の冒頭で 1 回呼ぶ。これは乱数の「種」を時刻に基づいて設定するもので、やらなくても以下の関数は使えますが、その場合毎回同じ擬似乱数列となります。
- rand() — 0 から RAND_MAX-1 (これも stdlib.h で定義) の範囲の整数一様乱数を返す。

これをもとに、0 以上 N 未満の整数乱数が欲しければ N で剰余を取り、また区間 $[0, 1)$ の実数乱数が欲しければ RAND_MAX で実数割り算して使います。

12.4.5 コマンド引数

配列について学んだところで、「実は main にはパラメタが渡されていた」という説明をします。たとえば、次のようなコマンド行でプログラムを起動したとします。

```
./a.out  3.5  4.2  6.8
```

これまでは最初の「./a.out」しか打っていませんでしたが… 実は main は int main(int argc, char *argv[]) のように定義することもでき、これにより 2 つの引数 argc、argv を受け取れます。argc はコマンド行で指定した文字列の個数 (上の例では 4)、そして argv は…「char*」というのは、次回扱いますが、「文字列」を意味しているので、argv は「文字列の配列」です。そして、内容はコマンド行に打ち込んだ 4 つの文字列"./a.out"、"3.5"、"4.2"、"6.8"が並んでいます。そこで、今回はとりあえず文字列を整数・実数に変換する次の関数を使います。

- int atoi(char *s) — 文字列に対応する整数値を返す。
- double atof(char *s) — 文字列に対応する実数値を返す。

これらを使うときは「#include <stdlib.h>」の指定が必要です。では、渡された数値の合計を計算してみましょう。

```
// argdemo.c --- argc/argv demonstration
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
  double sum = 0.0;
  int i;
  for(i = 1; i < argc; ++i) { sum += atof(argv[i]); }
  printf("sum = %g\n", sum);
  return 0;
}
```

argv[0] は"./a.out"なので、argv[1] 以降を実数に変換して加算しています。実行例は次の通り。

```
% ./a.out 3.5 4.2 6.8
sum = 14.5
```

＃13　文字列操作＋パターン探索

今回は次の内容を取り上げます。

- C 言語の文字・文字列の扱いと文字列の操作
- パターンマッチの考え方と実装 (経験者向け)
- 2 次元配列とポインタの配列 (経験者向け)

13.1　前回演習問題の解説

13.1.1　演習 1 — 配列の基本的な操作

これは関数本体だけ示します。説明は不要でしょう。

```
void piarrayrev(int n, int a[]) {
  int i;
  for(i = n-1; i >= 0; --i) {
    printf(" %2d", a[i]);
    if(i % 10 == 9 || i == 0) { printf("\n"); }
  }
}
int iindex(int n, int a[], int x) {
  int i;
  for(i = 0; i < n; ++i) {
    if(a[i] == x) { return i; }
  }
  return -1;
}
int maxiarray(int n, int a[]) {
  int i, max = a[0];
  for(i = 1; i < n; ++i) {
    if(max < a[i]) { max = a[i]; }
  }
  return max;
}

int miniarray(int n, int a[]) {
  int i, min = a[0];
  for(i = 1; i < n; ++i) {
    if(min > a[i]) { min = a[i]; }
  }
  return min;
}
int sumiarray(int n, int a[]) {
```

```
  int i, sum = 0;
  for(i = 0; i < n; ++i) { sum += a[i]; }
  return sum;
}
double avgarray(int n, int a[]) {
  int i, sum = 0;
  for(i = 0; i < n; ++i) { sum += a[i]; }
  return (double)sum / n;
}
void pdarray(int n, double a[]) {
  int i;
  for(i = 0; i < n; ++i) {
    printf(" %7g", a[i]);
    if(i % 5 == 4 || i == n-1) { printf("\n"); }
  }
}
```

avgarray で (double)sum とキャストしているのは、整数除算 (切捨て除算) を避けるためです。

13.1.2 演習 2 — 終わりの印の数値を用いた入力

代表例として riarray2 を示します。

```
int riarray2(int lim, int a[], int endval) {
  int d, i = 0;
  while(i < lim) {
    printf("input integer (%d for end) %d> ", endval, i);
    scanf("%d", &d);
    if(d == endval) { return i; } else { a[i++] = d; }
  }
  return i;
}
```

ループ内の最後は「a の i 番目に d の値を代入し、その後 i を 1 増やす」と読みます。また、データ数の上限に達した時はすぐ i を返せばよいです。i の値はデータの入っている個数と常に一致していることに注意。

13.1.3 演習 3 — フィボナッチ数

いちおう、フィボナッチ数の計算と打ち出しを main と別にしましたが、一緒でもよかったかも知れません。パラメタの無い呼び出しでも C 言語では「()」が必要です。

```
// printfib --- print fibonacchi numberts.
#include <stdio.h>
void printfib(void) {
  int i, fib[31] = {1, 1};
  for(i = 2; i <= 30; ++i) { fib[i] = fib[i-1]+fib[i-2]; }
  for(i = 0; i <= 30; ++i) { printf("%2d: %8d\n", i, fib[i]); }
}
int main(void) { printfib(); return 0; }
```

13.1.4 演習 5 — 動的計画法による釣り銭問題

この問題は部屋割り問題とほぼ同様のやり型でできます。ただし、おつりを「多め」に渡したら損ですから、選択肢を考えるときに「残額がそのコインの額面以上」という条件をつけています。

```
// coins1 --- coin problem DP
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define CMAX 1000
int coincnt[CMAX] = { 0, }, coinsel[CMAX] = { 0, };
void initialize(int *a, int *b, int n) {
  int i;
  for(i = 1; i < n; ++i) {
    int x = a[i-1] + 1, s = 1;
    if(i >= 5 && a[i-5]+1 < x) { x = a[i-5]+1; s = 5; }
    if(i >= 10 && a[i-10]+1 < x) { x = a[i-10]+1; s = 10; }
    if(i >= 25 && a[i-25]+1 < x) { x = a[i-25]+1; s = 25; }
    a[i] = x; b[i] = s;
  }
}
int main(void) {
  initialize(coincnt, coinsel, CMAX);
  while(true) {
    int n;
    printf("\ninput integer (0 for end)> "); scanf("%d", &n);
    if(n == 0) { return 0; }
    printf("coin count %d => %d;", n, coincnt[n]);
    while(n > 0) { printf(" %d", coinsel[n]); n -= coinsel[n]; }
  }
}
```

広域変数の配列とその大きさを `initialize` にパラメタで渡していますが、それは `initialize` の中では配列を 1 文字で指定したい (その方が読みやすいと思う) からです。枚数は `coincnt` から読むだけですが、トレースバックは「その金額のときの選択肢のコインの金額を引く」ことを金額が 0 になるまで繰り返します (だから 25 ¢ が 3 枚なら 3 回 25 と出ますがまあいいでしょう)。

```
% ./a.out
input integer (0 for end)> 80
coin count 80 => 4; 5 25 25 25
input integer (0 for end)> 115
coin count 115 => 6; 5 10 25 25 25 25
input integer (0 for end)> 40
coin count 40 => 3; 5 10 25
input integer (0 for end)> 0
%
```

13.1.5 演習 6 — 最大増加部分列

最大増加部分列はまず列の与え方を考える必要がありますが、ここではコマンド引数で与えることにします。それを整数に変換して配列 `seq` に格納します。そして `len` が動的計画法のための長さを入

れる配列、pre がトレースバック用配列です。

```
// lis1 --- longest increasing sequence DP
#include <stdio.h>
#include <stdlib.h>
#define SMAX 1000

(riarray をここに)
int main(void) {
  int seq[SMAX], len[SMAX], pre[SMAX];
  int i, n, maxi, maxl = 0;
  printf("n> "); scanf("%d", &n); riarrai(n, seq);
  for(i = 0; i < n; ++i) {
    int j, l = 1, p = -1;
    for(j = 0; j < i; ++j) {
      if(seq[j] < seq[i] && len[j] >= l) { l = len[j]+1; p = j; }
    }
    len[i] = l; pre[i] = p;
    if(len[i] > maxl) { maxl = len[i]; maxi = i; }
  }
  while(maxi >= 0) {
    printf(" %d", seq[maxi]); maxi = pre[maxi];
  }
  printf("\n");
  return 0;
}
```

l と p は自分が属する列の最大長とその場合の「1 つ前の番号」を表します。どの数字もそれ単独で長さ 1 の列になるので、これを初期値とし、1 つ前は「ない」のでこれを -1 で表します。続いてループで自分の番号より手前の各要素を調べ、自分より小さい値で、なおかつ現在自分が属している最大列の長さと同じ以上であれば、その後ろに自分がつながることでこれまでより長くなるので、長さと手前の番号を更新します。調べ終わったら現在の l と p を自分の位置に格納します。さらに、全体の最大列が必要なのでそれは maxl に長さ、maxi に位置を覚えます。最後まで格納し終わったら、見つかった最大長の列についてトレースバックしながら出力します (なので表示は逆順になります)。

```
% ./a.out
n> 8
(1 5 7 2 6 3 4 9 を順次入力)
 9 4 3 2 1
%
```

13.2 C 言語の文字型と文字列

13.2.1 基本型の整理と文字型 exam

ここで、C 言語の**基本型** (primitive type、これ以上分解できないような値の型) について整理します。まず、整数を扱う**整数型** (integral type) からです。整数型として int とビット数の多い long があること、**論理型** (Boolean type) はなく、整数型の 0 と 1 で代用することは既に説明しました。また、定数の表記方法に 8 進、16 進もありますが、すぐ後で文字型と一緒に述べます。

long の定数を指定したい場合は「1L」のように最後にエルをつけます (小文字も使えるが 1 と紛らわしい)。さらに文字型を含む整数型では、unsigned int、unsigned long のように冒頭に**符号なし** (unsigned) と指定することで、2 の補数の代わりに符号なし整数として扱えます。[1]

実数型 (real type) にも、double とビット数の少ない float があることは既に説明ました。実数の定数は十進のみで、数字列の中に小数点または指数表記があれば実数、それ以外は整数となります。「10000.0」、「1e5」「1.0e+05」はどれも 1 万を表す実数定数で型は double、float の定数を指定したい場合は最後に「F」をつけます (小文字も使えます)。

C 言語にはさらに、文字を表す**文字型**である char があります。しかし実際には、この型は「8 ビット幅の整数型」であり、しかも符号つきなので「-128〜127」の範囲の整数が入れられます (unsigned char とすれば符号なしになり「0〜255」の範囲になります)。

文字を表す定数は「'a'」、「'!'」のようにシングルクォートで囲み、中は 1 文字だけで。[2] そして整数型なので、'a' は 97、'!' は 33 と同等です (これらは各文字の ASCII コード値)。[3]

表 13.1: 文字定数と整数定数の記法の例

	文字定数			整数定数		
文字	文字・記号	16 進	8 進	16 進	8 進	十進
タブ (TAB)	'\t'	'\0x09'	'\011'	0x09	011	9
改行 (NL)	'\n'	'\0x0a'	'\012'	0x0a	012	10
復帰 (CR)	'\r'	'\0x0d'	'\015'	0x0d	015	13
ナル文字	'\0'	'\0x0'	'\000'	0x0	00	0
通常文字	'a'	'\0x61'	'\141'	0x61	0141	97

文字定数には改行などの制御文字や任意の文字コードを指定する記法がありますが、整数定数と一緒に説明します (図 13.1)。整数では「0x」ではじまり 16 進法の数字を並べた 16 進定数、「0」ではじまり 8 進法の数字 (0〜7) を並べた 8 進定数が書けます。そして文字定数でも「'...'」の中に「\」に続けて同様に書くことができます (8 進は常に 0〜7 の数字 3 個)。さらに、改行、復帰、タブなどよく使う制御文字はこれらを 1 文字で表す記法があります。このように「\」で始まる列は通常の文字と違う特別な意味を持つことから**エスケープシーケンス** (escape sequence) と呼ばれます。「\」(エスケープ文字) そのものを書きたい場合は「'\\'」、シングルクォートを書きたい場合は「'\''」です。

13.2.2 文字列の扱い exam

文字型の説明に続いてようやく、**文字列** (string) の説明ができます。C 言語では文字列は「文字の配列」です。そして、文字の配列の初期化に文字列を指定できます。簡単な例を見てみましょう。泥縄ですが、書式文字列において char の入出力は「%c」、文字列の入出力は「%s」で指定します。

```
// str1.c --- string demonstration 1.
#include <stdio.h>
void showstr(int n, char *s) {
  int i;
  for(i = 0; i < n; ++i) { printf("  %c", s[i]); }
  putchar('\n');
  for(i = 0; i < n; ++i) { printf(" %02x", s[i]); }
  putchar('\n');
```

[1] このように様々な整数型があり、ビット幅もシステムによって異るので、正しい整数型を選ぶのは結構大変です。このため標準ヘッダファイルで size_t、time_t などの型が定義されていて、標準ライブラリではこれらの型を使っています。

[2] Ruby では"..."も'...' も文字列でしたが、C では前者は文字例、後者は文字で全く違っています。

[3] 日本語はどうなのという疑問が湧くでしょうけれど、C 言語は歴史的経緯により日本語の扱いが面倒なので、日本語を扱いたければ Ruby などを使うことを進めます。本資料では C 言語で扱う文字はすべて ASCII の範囲とします。

```
  printf("string = '%s'\n", s);
}
int main(void) {
  char s[] = "abcde";
  showstr(sizeof(s), s);
  return 0;
}
```

showstr は文字列配列 (実際には文字型へポインタ) と文字列の長さを渡して中身を表示させます。まず最初のループで、それぞれの文字をそのまま (下と揃えるために空白をあけて) 打ちます。次の putchar とは? これは標準出力に「1 文字」出力する関数で、改行 1 文字だけ出力するのなら「printf("\n");」よりこの方が明解です。2 番目のループでは、各文字の文字コードを 16 進表現で出力し、再度改行します。最後に、printf の「%s」書式で文字列を打ち出します。ですが、変だと思いませんか? 文字列は配列なのに、長さを渡しません。なぜそれで大丈夫かはすぐに説明します。

main ですが、文字配列 s を"abcde"で初期化して用意し (長さは文字列から自動決定)、sizeof(s) で配列 s のバイト数を取得し、先頭アドレスと一緒に showstr を呼びます。では実行例です。

```
% ./a.out
  a  b  c  d  e
 61 62 63 64 65 00
string = 'abcde'
```

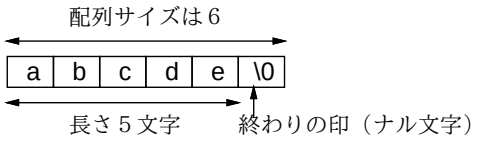


図 13.1: C の文字列とナル文字

予想外のことがいつかあったかと思います。まず、文字列は"abcde"の 5 文字に見えるのに、配列 s のサイズが 6 です。これは、C では文字列は最後に**ナル文字** (null character)「'\0'」がくっつくため、1 文字多くなるのです (図 13.1)。考えてみてください。文字列はいろいろな長さのものを扱いたいでしょうから、その長さを表す方法が必要です。常に長さを別の変数で管理するより、「最後にナル文字があってそこまでが文字列」とするのが単純でよい、というのが C 言語の設計方針です。ナル文字を%c で出力 16 進で表示するとコードは 00 です。そして、printf の%s で文字列の長さを渡さないで済んでいるのは、「ナル文字の手前まで出力」するようになっているからです。

13.2.3 文字列の入力 exam

文字列を入力するとき、普通に考えると scanf で%s を指定すればよさそうですが、それだと空白などがあるとそこで入力が終わってしまうように作られていて、あまりよくありません。そこで、1 行ぶん (改行まで) 入力する、という関数 getl を自前で作りました。この関数には文字列を入れる配列と、最大何文字まで入れてよいかの値 (つまり配列のサイズ) を渡します。

この中で文字を読むのに、先に出てきた putchar の相方である getchar という関数を使います。この関数は入力の文字を 1 文字ずつ返しますが、ただし入力の終わりになったら EOF という特別な値 (stdio.h の中で定義) を返します。そこで、入力終わりになったときは false、それ以外の普通に読めた場合は true を返すことにして、getl の戻り値は bool としてあります。

その中身の処理ですが、まず i は 0 としてから、for ループを使って 1 行を読んで行きます。見慣れない形ですが、まず 1 文字読んでから終わり (EOF か改行) をチェックし、以後も 1 文字読んではチェッ

クするので、初期設定と更新がともに「c = getchar()」になっていて、条件は「EOF でなく改行でもない間」繰り返すとしています。

ループ内では s の i 文字目に読めた文字を入れ、i は 1 増やします。ここで、最大文字数の 1 つ手前 (ナル文字まで入れたら満杯) に到達していたら、これ以上入れられないのでこの場合もループを抜けます。抜けたところで最後にナル文字を入れ、終わりかどうかは最後に読んだ文字が EOF かどうかで分かるのでその論理値を返します。

```
// str2.c --- string demonstration 2.
#include <stdio.h>
#include <stdbool.h>

bool getl(char s[], int lim) {
  int c, i = 0;
  for(c = getchar(); c != EOF && c != '\n'; c = getchar()) {
    s[i++] = c; if(i+1 >= lim) { break; }
  }
  s[i] = '\0'; return c != EOF;
}
void printtriangle(char s[]) {
  int i = 0;
  while(s[i] != '\0') { printf("%s\n", s+i); ++i; }
}
int main(void) {
  char buf[100];
  printf("s> "); getl(buf, 100);
  printtriangle(buf);
  return 0;
}
```

printtriangle は文字列を「先頭から」「先頭の 1 つ先から」「2 つ先から」…と繰り返し出力しますが、ナル文字まで来たら終わり、という関数です。s+i というのがポインタ演算で、配列 s の先頭から i 文字ぶん先のアドレスを計算しています。main は 1 行入力して printtriangle を呼ぶだけです。実行例は次の通り。

```
% ./a.out
s> abcd
abcd
bcd
cd
d
```

あと 2 つほど補足です。"abcd"のような文字列リテラルの中にも文字リテラルと同じエスケープシーケンスが使えます。そして、文字列リテラルは配列と同じものなので「char *p = "abcd";」のように書くことができます。つまり文字列リテラルの型はその文字列が入った配列の先頭要素のポインタ値ですが、ただし文字列リテラルの中身を直接書き換えてはいけません (先の例の「char s[] = "abcde";」では配列の初期値なので変更もできます)。

演習 1 2 番目の (getl を含む方の) 例題を打ち込んで動かしなさい。OK なら、次のものをやってみなさい。関数を作ったら、その動作が確認できるように呼び出してみること。

a. getl で入力した文字列は毎回長さが違っているので、その長さを調べたい。文字列の長さを調べて返す関数 int mystrlen(char s[]) を作れ。(ヒント: printtriangle の中で [i] == '\0' が成立したときの i は文字列の長さの値になっている。)

b. printtriangle では1行ごとに先頭の文字が削られて行が短くなっていたが、それと似ているが1行ごとに末尾の文字が削られて短くなっていく関数 void printtriangletail(char s[]) を作れ。(ヒント: 文字列のどの位置でも、ナル文字を代入したらそこが文字列の終わりになる。)

c. 文字列の中に現れる文字 c1 を文字 c2 に置き換える (たとえば空白を*に置き換えたりできる) 関数 void mapchar(char s[], char c1, char c2) を作れ。

d. 文字列中の指定した文字 c1 をすべて削除して詰める関数 void deletechar(char s[], char c1) を作れ。

e. 文字列を左右ひっくり返す関数 void reverse(char s[]) を作れ。

f. 文字列 s2 の内容を別の文字配列 s1 にコピーする関数 void mystrcpy(char s1[], char s2[]) を作れ。

g. 文字列 s2 の内容を別の文字列 s1 の末尾に追加する (くっつける) 関数 void mystrcat(char s1[], char s2[]) を作れ。

h. 文字列 s1 と s2 を比較して等しければ 0、1 番目のものがコード順で後なら 1、前なら-1 を返す関数 int mystrcmp(char s1[], char s2[]) を作れ。たとえば mystrcmp("abcd", "abcd") → 0、mystrcmp("abcd", "abaa") → 1、mystrcmp("abcd", "abcz") → -1 となる。(ヒント: 先頭から両方の文字列を比べていき、最後まで (ナル文字まで) 同じなら等しい。そうでなければ、違いがあったところの対応する文字の大小関係で 1 か-1 を返せばよい。)

13.2.4 文字列ライブラリ exam

上で演習問題にしたもののうち「my…」となっているものは、実はライブラリで実装されているものです (混同しないように my をつけました)。文字列ライブラリを使うためには、「#include <string.h>」を指定する必要がありますが、いちいち自分で定義せずに使えるので便利です。

- size_t strlen(s), size_t strnlen(s, L) — 領域 s にある文字列の長さを返す。
- char *strcpy(s, t), char *strncpy(s, t, L) — 領域 t から s に文字列をコピー。
- char *strcat(s, t), char *strncat(s, t, L) — t の文字列を領域 s の文字列末尾に連結。
- int strcmp(s, t), int strncmp(s, t, L) — 領域 s と t の文字列を比較し、結果として正の数、0、負の数のいずれかを返す。

文字 n がついているものは長さ上限 L を渡すもので、ついていないものは長さ上限を渡しません。長さ上限を渡さないと、用意されている領域より先まで書き込んでしまう恐れがあるので、配列への書き込みをするものは長さ上限つきを使う方が安全です。読み取るだけのもの (strlen や strcmp) については、領域に書き込まないので、渡す文字列を間違えない限りはあまり危険はありません。

なお、戻り値の型が見慣れないと思ったかも知れません。size_t は領域の長さなどの表現に使う標準ライブラリの型ですが、int に代入できると思って差し支えありません。また、コピー系のものは char*を返しますが、これは何も返さないよりはたまたま使える場合は利用できるように操作した文字列のアドレスを返しているもので、本資料では利用していません。

13.2.5 文字列から数値への変換 exam

次の例題は「整数と文字列を読み込み、指定回数だけ文字列を打ち出す」です。

```
// str3.c --- string demonstration 3 --- wrong version.
#include <stdio.h>
#include <stdbool.h>
(ここに getl)
int main(void) {
  int i, n;
  char buf[100];
  printf("n> "); scanf("%d", &n);
  printf("s> "); getl(buf, 100);
  for(i = 0; i < n; ++i) { printf("%s\n", buf); }
  return 0;
}
```

動かしてみたところ、思ったように動作しません。「3」を入力して改行したとたん、空文字列が入力されたことになり、3 行空行が打ち出されてしまいます。

```
% ./a.out
n> 3
s>




%
```

なぜこうなるかというと、scanf は数字「3」を読み取ると処理を終わるので、その直後に入力した「改行」は入力として残っているためです。そのため、getl が呼ばれるとその改行が読まれ、空っぽの行が入ったことになってしまいます。このような問題があるので、scanf での数値入力と文字列入力はまぜない方がよいのです。ではどうするかというと、すべて文字列入力で扱い、前回付録でも出て来た次の関数 (#include <stdlib.h>が必要です) を用いて文字列を整数や実数に変換します。

- int atoi(char *s) — 文字列が表している整数値を返す。
- double atof(char *s) — 文字列が表している実数値を返す。

これを用いた改良版を示しましょう。

```
// str4.c --- string demonstration 4 --- correct version
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
(getl をここに)
int main(void) {
  int i, n;
  char buf[100];
  printf("n> "); getl(buf, 100); n = atoi(buf);
  printf("s> "); getl(buf, 100);
  for(i = 0; i < n; ++i) { printf("%s\n", buf); }
  return 0;
}
```

こんどは次のように予定通り動きます。

```
% ./a.out
n> 3
s> abcd
abcd
abcd
abcd
%
```

13.2.6 switch 文

文字列の話題ではないのですが、文字列と一緒に使うことが多い switch という制御構文をここで紹介しておきます。switch 文の一般形は次のようなものです。

```
  switch(式) {
case 値: case 値: 文…; break;
case 値: case 値: 文…; break;
default: 文…;
  }
```

まず「式」は整数型の式、「値」は整数の定数である必要があります (文字リテラルも整数の定数です)。そして、最初の式がどれかの「値」に一致したら、その場所にある文に実行が移ります (どれにもあてはまらなければ `default:`の次にある文に移ります。`default:`が書かれていなければ switch 文の中は実行せず次の文に進みます)。switch 文の中の文に実行が移ったあとは普通の実行ですが、`break;` はこの switch 文から外に出ることを意味します (もしうっかり `break;` を書き忘れると次にある `case` ラベルや `default:`の後の文に「合流」するので注意。はまりやすいです)。

では例題として、先の `atoi` のそっくりさんを作ってみます。

```
// switch1.c --- demonstaration of switch stat.
#include <stdio.h>
#include <stdbool.h>

int myatoi(char *s) {
  int sign = 1, val = 0;
  switch (*s) {
case '-': sign = -1;
case '+': ++s;
  }
  while(true) {
    char c = *s++;
    switch(c) {
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
      val = val * 10 + (c - '0'); break;
default:
      return sign * val;
    }
  }
}
int main(int argc, char *argv[]) {
```

```
    int i = myatoi(argv[1]);
    printf("value = %d\n", i);
    return 0;
  }
```

このプログラムでは前回付録でやった「コマンド引数」を使っています。具体的には「./a.out 123」のようにプログラムを起動するときに一緒に入力を渡します。この"123"という文字列は argv[1] として渡されて来るので、それに対して myatoi を呼び、返された整数を打ち出しています。

myatoi の内容ですが、まず符号の変数 sign に 1、値の変数 val に 0 を入れます。次に文字列の先頭の文字によって分岐し、'-' だった場合は sign に −1 を入れ直し、そして s を次の文字に進め…ますが、それは'+' の場合もやることなので、わざと break; を書かずに'+' の処理に合流しています。符号でない場合はこの部分の処理は何も起きません。次は無限ループで、その中で、s が指している箇所の文字を取り出して c に入れ、「その後で」s を次の文字に進めます。こうやって 1 文字ずつ処理していくわけです。そして、'0'〜'9' の数字のどれかだった場合はこれまでの値を 10 倍して、c から'0' の値を引いたものを加えます。数字の文字コードは連続しているので、この引き算で'1' なら 1、'3' なら 3 等が得られるのです。そして、それ以外の文字 (文字列末尾の'\0' も含む) だったら、これまでに作って来た sign と val を乗じたものを返せばいいのです。ということは、途中に数字以外のものが出てきたらそこで終わりますが、atoi もそういうふうにできているのでした。さて、いかがでしたか。switch 文、便利でしょうか? しかし…次のも見てください。

```
int myatoi(char *s) {
  int sign = 1, val = 0;
  if(*s == '-') { sign = -1; ++s; } else if(*s == '+') { ++s; }
  while(true) {
    char c = *s++;
    if(c < '0' || c > '9') { return sign * val; }
    val = val * 10 + (c - '0');
  }
}
```

別に if 文でいいような気もしますね。まあ用途として合っていそうなことがあったときに、switch 文のことも思い出してあげてください。

演習 2 上の例題の好きな方を打ち込んで動かしなさい。動いたら次のいずれかをやってみなさい。

a. 数字列の先頭が 0 ではじまったら 8 進として受け取るようにする。
b. 数字列の先頭が 0x ではじまったら 16 進として受け取るようにする。
c. atoi の代わりに自分流 atof(文字列を実数に変換) を (指数記法「e ±数字列」は不要)。
d. 指数記法も扱える自分流 atof を作る。
e. switch 文を使った何か面白いプログラムを作る。

13.3 パターンマッチング

13.3.1 部分文字列の検索

次のテーマは、ある (長い) 文字列 str 中に指定した (短い) 文字列 pat が含まれているか、含まれているとしたらどこに含まれているかを調べるという問題です。図 13.2 のように、「'abbabbab'」の中で「'bbab'」がどこにあるかを探すと、(先頭が 0 なので)1 文字目と 4 文字目にあることがわかる、という具合です。完成したプログラムを動かすと次のように先頭位置に印を表示します。

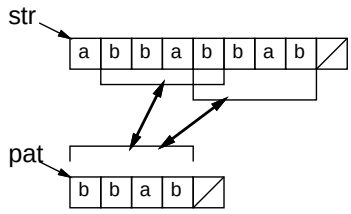


図 13.2: 部分文字列の検索

```
% ./a.out 'abbabbab' 'bbab'
abbabbab
 ^
abbabbab
    ^
%
```

では、いきなり全部示すのでなく、下請けの関数から順に読んでいきます。まず、文字列 str と pat の先頭 len 文字が一致していれば true、そうでなければ false を返す関数 nmatch を見てみます。なお、bool、true、false を使っていますので、#include <stdbool.h>が必要です。

```
bool nmatch(char *str, char *pat, int len) {
  while(len-- > 0) {
    if(*str++ != *pat++) { return false; }
  }
  return true;
}
```

「なんだこれは」と思うかも知れませんが、このようなのが C 言語流です。まず、len 文字比べるのですから、len が 0 より大きくなければ比べません。つまり len > 0 を条件とする while を書いています。普通なら while の末尾で len を 1 減らすのですが (len はこの関数内ではローカル変数と同じなので他に使わなければ変更して行って構いません)、len を使っているのはここだけなので、条件を調べるため参照したあとすぐ減らしてしまいます。これが後置演算子 len--の働きでした。[4]

次に文字列 str と pat の先頭文字が等しいか調べますから、if の条件は*str == *pat でこれが「いいえ」なら false を返します。そうでなければ str も pat も次の文字の場所に増やしますが、それを別に書く代わりに後置演算子の++で行えます。指定文字ぶんだけ調べてループが終わったら true を返せばよいですね。

さて、では次はこれを利用して「どこか途中にある」場合にその位置を探す関数 findstr を作ります。どこにあるかは「何文字目」という整数を返せば良さそうですが、ここではその代わりに「その見つかった位置のポインタを返す」ことにします。というのは、ポインタ演算 p + i で「p の i 要素ぶん先のポインタ」が得られるのと逆に、「q - p」というポインタ同士の演算で「位置が何要素ぶん離れているか」が計算できるので同じことだからです。なお、見つからなかった場合は NULL という値を返します (実態は 0 ですが C 言語では stdio.h や string.h で定義されていて「ない」ことを表すのに使います)。それでは見てみましょう。

```
char *findstr(char *str, char *pat, int len) {
  if(len == 0) { return NULL; }
  int l;
  for(l = strlen(pat); l <= len; ++str, --len) {
```

[4]復習: --i は i を 1 減らし減らした後の i の値を返します。i--は i を 1 減らすが減らす前の i の値を返します。

```
    if(nmatch(str, pat, l)) { return str; }
  }
  return NULL;
}
```

len は str の長さで、これが 0 のときは見つかりようがないので NULL をすぐ返します。次はいきなり for 文ですが、初期設定として変数 l(小文字のエル — 1 と間違いやすいですが長さにはエルを使いたいので) に pat が何文字ぶんかを入れ、ループの条件としては l が len 以下の間繰り返します。この条件はヘンに思えるかも知れませんが、ある位置で試してあてはまらなければ、str を 1 進めて (つまり先頭の文字は除外して)、ということは長さは減らす必要があるので len は 1 減らす、というのがループ周回ごとの処理です。この 2 つを一緒に書くためにカンマ演算子 , で並べています (for のこの場所にセミコロンは書けないので)。

こういうコンパクトに詰め込めるところが C 言語らしいところです。で、ループ本体では現在位置にあてはまるかを nmatch で調べ、OK なら str を返します。最後まであてはまらずにループを抜けたら NULL を返します。

では main を見てみましょう。コマンド引数で受け取った 2 つの文字列を str および pat として findstr を呼びます。

```
// findstr1.c --- search substring occurence in longer string.
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
void putrepl(char c, int count) {
  while(count-- > 0) { putchar(c); }
}
(ここに nmatch)
(ここに findstr)
int main(int argc, char *argv[]) {
  if(argc != 3) { fprintf(stderr, "needs 2 args.\n"); return 1; }
  char *str = argv[1];
  int len = strlen(str);
  while(true) {
    str = findstr(str, argv[2], len - (str - argv[1]));
    if(str == NULL) { return 0; }
    printf("%s\n", argv[1]);
    putrepl(' ', str - argv[1]); printf("^\n"); ++str;
  }
}
```

長さのかわりにへんなものを渡していますが何でしょうか? それに while ループ (それも条件が true なので無限ループ) ですが…まず最初は、str と argv[1] は同じなので (str - argv[1]) は 0 で、長さ len がそのまま渡ります。そして、findstr であてはまり位置が返されます。これが NULL ならあてはまりは無いので 0 を返して終わりです。そうでない場合は、もともとの文字列を表示し、次に「str が argv[1] より何文字進んでいるか」計算してそのぶんだけ空白を出力し (putrepl はすぐ読めますね — putchar は 1 文字を出力する関数です)、そのあと目印の記号と改行を出力します。ループ本体の最後で str を 1 文字ぶん先に進めるので、次は今の場所より先にあるあてはまり位置を探すことになります。というわけで、findstr に渡す長さは len から先に進んだぶんだけ差し引く必要があるのでした。実行結果は先に示した通り。

演習 3 このプログラムをそのまま動かしなさい。OK なら、目印を先頭位置に 1 文字表示するのでなく、pat のあてはまる範囲に連続して表示するように直してみなさい。

13.3.2 正規表現のマッチ

上では「指定した文字列ぴったり」があてはまる位置を調べましたが、Unix の**正規表現** (regular expression) のように「パターン」が指定できるとずっと便利です。正規表現には多くの機能があり全部は大変なので、とりあえず「+」つまり「直前の文字を 1 回以上繰り返す」機能だけ作ってみます。実行例を先に見ましょう。

```
% ./a.out 'aababbaaabaaabbbaa' 'abb+aa+'
aababbaaabaaabbbaa
   ^^^^^^
aababbaaabaaabbbaa
            ^^^^^^
%
```

パターンは abb+aa+つまり「a が 1 個、b が 2 個以上、a が 2 個以上」で、実際そのようなところだけに印がついています。

ではコードを見ます。main は先のものと似ていますが、大きく違うのは matchstr(先の findstr の代わり) は「どこからどこまでがあてはまったか」返す必要があるという点です。2 つ値を返す際、Ruby では配列が使えますが、C では配列はアドレスしか返せず、このような用途に不向きです。そこで先と同じに先頭位置は返値で返し「どこまで」の方は scanf のように変数のアドレスを渡し、その変数に格納してもらうことで受け取ります。受け取る変数の型は文字へのポインタ「char*」ですから、その変数へのポインタは「char**」つまりポインタへのポインタ、になります。

```
// matchstr1.c --- match pattern occurence in a string.
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
(ここに putrepl)
(ここに matchstr)
(ここに pmatch)

int main(int argc, char *argv[]) {
  if(argc != 3) { fprintf(stderr, "needs 2 args.\n"); return 1; }
  char *str = argv[1];
  int len = strlen(str);
  while(true) {
    char *tail;
    str = matchstr(str, argv[2], len - (str - argv[1]), &tail);
    if(str == NULL) { return 0; }
    printf("%s\n", argv[1]);
    putrepl(' ', str-argv[1]); putrepl('^', tail-str);
    putchar('\n'); ++str;
  }
}
```

「どこまで」が受け取れれば、その範囲全体に印をつけるのも簡単です (putrepl は先の例と同じ)。

次に matchstr ですが、長さ 0 なら NULL を返すのは同じ。次に、探される文字列の末尾位置 lim を計算し、str がそれより手前にある間 str を 1 つずつ進めながら調べる for ループに入ります。その中では pmatch を読んで現在の str のその位置にあてはまるかどうか調べます。pmatch はあてはまるならその終わりの位置、あてはまらなければ NULL を返すようにしたので、NULL でなければ終わりの位置を渡されて来たポインタの場所に格納して先頭位置を返します。for ループを終わってもあてはまりが無ければ自分も NULL を返します。

```
char *matchstr(char *str, char *pat, int len, char **tail) {
  if(len == 0) { return NULL; }
  char *lim;
  for(lim = str + len; str < lim; ++str) {
    char *t = pmatch(str, pat, lim);
    if(t != NULL) { *tail = t; return str; }
  }
  return NULL;
}
```

pmatch では終端位置 lim が渡されてくるので、現在位置 str がそれより先ならあてはまらないので NULL を返します。そうでなくてパターンの最後まで来たら、あてはまり終わったということなので成功として現在位置を返します。その次の printf はデバッグ用ですが動きが分からないときはコメントアウトして動かしてみてください。

```
char *pmatch(char *str, char *pat, char *lim) {
  if(str > lim) { return NULL; }
  if(*pat == '\0') { return str; }
//printf("pmatch: '%s' '%s'\n", str, pat);
  if(pat[0] == str[0] && pat[1] == '+') {
    int i = 1;
    while(pat[0] == str[i]) { ++i; }
    for( ; i > 0; --i) {
      char *t = pmatch(str+i, pat+2, lim);
      if(t != NULL) { return t; }
    }
    return NULL;
  } else {
    if(*pat != *str) { return NULL; }
    return pmatch(str+1, pat+1, lim);
  }
}
```

さて次ですが、現在位置があてはまって次が「+」のときは繰り返しのパターンです (最初の条件は*pat == *str でもいいのですが、この後*(pat + 1) なども必要になるので代わりに短く書ける配列添字記法に揃えています)。

その処理の内容ですが、まず変数 i を用意し、str[i] が pat[0] と等しい間 i を増やすことで、pat[0] の文字の並びが「最大で」いくつあるかを i に求めます。ただしこの i は「最大の」繰り返し数であり、実際にあてはまるのはそれより少ない回数かも知れません。図 13.3 を見てください。文字列が'aaaabb'、パターンが'a+ab' のとき、先頭からマッチさせるとしてパターン'a+' を最大の 'aaaa' とマッチさせると、文字列の次は'b' ですからマッチしません。1 つ減らして'aaa' にすれ

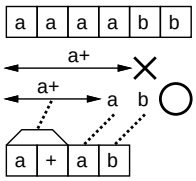


図 13.3: パターンのあてはまりの探索

ば、残りが'ab' になるのでうまくマッチします。ですから次に、for ループで i を 1 つずつ減らしながらこれで OK かどうかを調べます。[5]

なお、この for ループでは初期化は空です。このように for 文では初期化も条件も更新も空っぽでも大丈夫です。そこでは何もしないし、条件は成り立っているものと見なされます。つまり「for(; ;) {...}」は無限ループつまり「while(true) {...}」と同等です。

話を戻して、i を変化させながら、その反復数の状態で文字列の残り、パターンの残り (+の次の文字)、文字列の終端 (これはずっと同じ) を渡して自分自身を再帰的に呼びます。そこから返された値が NULL でなければ OK なので、その値をそのまま返します (これがあてはまりの終端)。ループを全部試し終わっても成功しなければ NULL を返します。

以上が「+」パターンの処理で、残りは普通の場合です。次の文字どうしが一致しなければ NULL を返し、一致していれば文字列もパターンも 1 文字進めた状態で pmatch を再帰呼び出しします。

この pmatch は 1 つ処理したら自分を再帰的に呼び出しますが、理由がお分かりでしょうか。それは、たとえば「+」パターンが複数あった場合、最初のもので長さを変化させながら、さらにそれぞれについて 2 番目でも長さを変化させ調べる必要があるからです。そのためには普通は 2 重ループが必要ですが、再帰を使えば何重のループでも実行時に作り出せます。

演習 4 上のコードを打ち込み、パターンが正しく処理されていることを確認しなさい。OK なら、次のことをやってみなさい。

a. 「+」(1 回以上の繰り返し) に加えて「*」(0 回以上の繰り返し) も記述できるようにしてみなさい。
b. 「?」(直前の文字があってもなくてもよい) を実現してみなさい。
c. ^(先頭に固定) と$(末尾に固定) を実現してみなさい。
d. 文字クラス [...](... の文字のいずれかならあてはまる) を実現してみなさい。[^...](... のいずれでもなければ) も実現できるとなおよいです。
e. ここまでに出て来た特殊文字の機能をなくすエスケープ記号「\」を実現しなさい (この文字に続いて特殊文字があった場合通常の文字として扱う)。
f. その他、パターンマッチにおいてあると面白いと思う好きな機能を選び実現しなさい。

13.4 ポインタ配列と多次元配列

13.4.1 ポインタ配列

ここまで C 言語の配列として 1 次元のものばかり扱って来ましたが、2 次元以上の配列ももちろん使いたいです。しかしその前に、ポインタの配列を見てみましょう。その代表例が argv です。このパラメタはコマンド行で指定した文字列の並びを受け取ります。つまり文字列の配列ですが、文字列というのは C 言語では文字の配列でしたね。そして配列の値というのは先頭要素へのポインタなわ

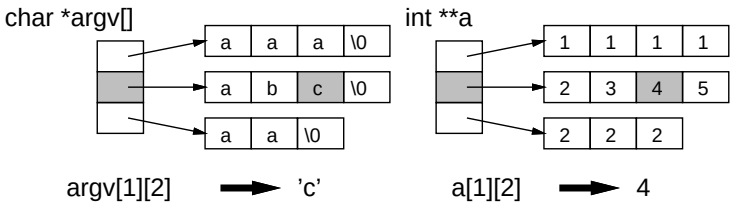


図 13.4: ポインタの配列の図解

けで、ですから argv はポインタの配列なのです。文字でなく整数や実数のポインタの配列も当然あり得ます (図 13.4)。[6]

ところで、char *argv[] って何でしょう? C では配列の名前はその先頭要素のポインタ値と同じであり、配列を関数に渡すときはパラメタの型はポインタというふうに説明してきました。しかし配列を渡すことを想定しているのなら、パラメタも配列と書きたいですよね? そこで C 言語では、パラメタに型を書く時、パラメタ名の直後に [] を書くことができ、これを先頭の*と同じに扱います。ですから実際には char **argv でもよかったのです。[7]

13.4.2 多次元配列

さて、C では 2 次元以上の配列も普通に要素数を指定することで作り出せます。int c[10][10]; であれば 100 要素、int d[10][10][10]; であれば 1000 要素の領域が確保されて使えます。参照も c[i][j] とか d[j][k][l] とか普通です。

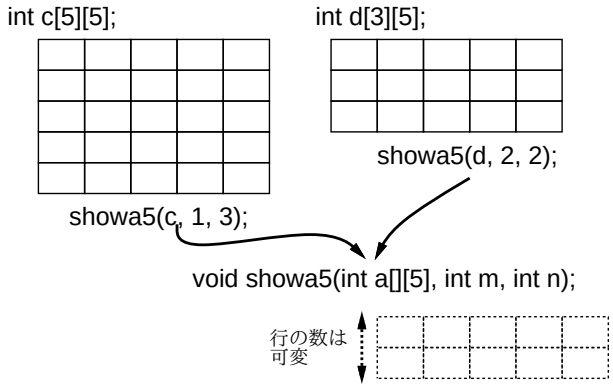


図 13.5: 2 次元配列の図解

ただし、2 次元以上の配列をパラメタとして渡すときはちょっと注意が必要です。1 次元目についてはこれまでと同じ考えでポインタとして渡され、実際の個数はいくつでもよいのですが、2 次元目以降の要素数は定数を書かないといけません。そうしないと呼ばれる側で大きさが分からないからです。このため、たとえば int c[5][5] や int d[3][5] を受け取るパラメタは int a[][5] のように書く必要があります (図 13.5)。例を見てみます。

```
// array2dim1.c --- demonstorate of 2dim array.
#include <stdio.h>
void showa5(int a[][5], int m, int n) {
  int i, j;
  for(i = m; i <= n; ++i) {
```

[5]なぜ長い方から順に調べるかということですが、それはパターンを書いたらそれに対するなるべく長いあてはまりを優先するのが人間にとって自然だからです。

[6]Ruby では「配列の配列」等はすべてポインタの配列でした。配列オブジェクトはすべて参照値として扱うので。

[7]2 個以上 [] を連続させることはできません。その説明はすぐ後で。

```
    for(j = 0; j < 5; ++j) { printf("%3d", a[i][j]); }
    putchar('\n');
  }
}
int main(int argc, char argv) {
  int c[5][5] = { {1, 2, 3, 4, 5}, {2, 3, 4, 5, 6},
    {3, 4, 5, 6, 7}, {4, 5, 6, 7, 8}, {5, 6, 7, 8, 9} };
  int d[][5] = {{5,4,3,2,1},{6,5,4,3,2},{7,6,5,4,3}};
  showa5(c, 1, 3); showa5(d, 2, 2); return 0;
}
```

実行例は次の通り。

```
% ./a.out
  2  3  4  5  6
  3  4  5  6  7
  4  5  6  7  8
  7  6  5  4  3
%
```

では、2 次元目以上の要素数が様々なものを受け取る関数はどうすればいいの? それは C99 でのみ使える機能があります。具体的には、C99 ではパラメタの配列の要素数に一緒にパラメタとして受け取る整数型のパラメタ名を書けます (ただし配列より前に書かれている必要あり)。つまり、次のようにできるのです。

```
void showx(int w, int a[][w], ...) { ... }
```

ただし、この機能は C11 ではオプションになるなど、今後とも使えるとは限らないので、説明はしましたが基本的に使わない方がいいでしょう。

演習 5 2 次元以上の配列またはポインタ配列を使った自分の面白いと思うプログラムを作りなさい。

本日の課題 13A

「演習 1」または「演習 2」で動かしたプログラム (どれか 1 つでよい) を含むレポートを提出しなさい。プログラムと、簡単な説明が含まれること。アンケートの回答もおこなうこと。

Q1. C 言語の文字列機能についてどのように思いましたか。

Q2. パターンマッチや 2 次元配列についてはどうですか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 13B

「演習 1」〜「演習 5」の (小) 課題から 1 つ以上を選択してプログラムを作り、レポートを提出しなさい。プログラムと、課題に対する報告・考察 (やってみた結果・そこから分かったことの記述) が含まれること。アンケートの回答もおこなうこと。

Q1. 文字列の基本的な操作ができるようになりましたか。

Q2. 文字列から整数や実数を作り出す原理が分かりましたか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

＃14　構造体+表と探索

今回は次の内容を取り上げます。

- C 言語の構造体機能
- 構造体・配列・ポインタによる表の実現 (経験者向け)

14.1　前回の演習問題解説

14.1.1　演習 1 — 文字列の基本的な演習

細かい課題が沢山あるのでいちいち`#include` や `main` は示さず、該当する関数だけ示すようにします。まず文字列の長さです。`len` を最初に 0 とし、`s[len]` がナル文字でない間 1 つずつ増やし、ナル文字だったらループを終わって `len` を返します。ナル文字の位置 (添字値) と文字列の長さは一致しているため、これでよいわけです。

```
int mystrlen(char s[]) {
  int len = 0;
  while(s[len] != '\0') { ++len; }
  return len;
}
```

次は文字列を連続して打つけれど、後ろから順番に削って行くものです。上で作ったナル文字の位置に最初はナル文字を入れ (つまり何も変化なし)、出力し、次に 1 つ手前にナル文字を入れ、出力し、…と繰り返して行きます。入れる位置が 0 番になったら空文字列は出力してもしかたないので終わります。`len--`は、`len` の値をもってくるけど、その後副作用として `len` を 1 減らすという演算子でしたね。

```
void printtriangletail(char s[]) {
  int len = mystrlen(s);
  while(len > 0) { s[len--] = '\0'; printf("%s\n", s); }
}
```

次は文字 c1 を文字 c2 に置き換えます。こんどは for 文を使っていますが、for の条件部に「カウンタがいくつまで」のような条件ではなく、「i 番目がナル文字でない間繰り返し i を増やして行く」となっています。ループ内では普通に、i 番目が c1 ならそれを c2 に変更しています。

```
void mapchar(char s[], char c1, char c2) {
  int i;
  for(i = 0; s[i] != '\0'; ++i) {
    if(s[i] == c1) { s[i] = c2; }
  }
}
```

次は指定文字の削除です。変数 `i` を 1 文字ずつ進めながらその位置の文字を変数 `j` の位置にコピーして `j` を進めますが、削除する文字のときはコピーしません。先の例と同様、ナル文字に遭遇したら終わりですが、そうするとナル文字はコピーされないので、ループを出てから `j` の位置にナル文字を入れています。

```
void delchar(char *s, char del, int lim) {
  int i, j = 0;
  for(i = 0; s[i] != '\0'; ++i) {
    if(s[i] != del) { s[j++] = s[i]; }
  }
  s[j] = '\0';
}
```

次は文字列の反転です。下請けの交換関数を作り、文字列の前半それぞれについて、後半の対応する位置と交換します。

```
void cswap(char *s, int i, int j) {
  char c = s[i]; s[i] = s[j]; s[j] = c;
}
void reverse(char *s) {
  int i, len = strlen(s);
  for(i = 0; i < len/2; ++i) { cswap(s, i, len-i-1); }
}
```

次は文字列のコピーと連結ですが、コピーは長さとよく似ていて、調べるだけでなく、代わりに新しい文字列にコピーします。ナル文字は最後に追加する必要があります。連結は最初にコピー先の長さを調べて j に入れ、そこはナル文字の位置ですから、そこから先にコピーします。

```
void mystrcpy(char *s, char *t) {
  int i = 0;
  while(t[i] != '\0') { s[i] = t[i]; ++i; }
  s[i] = '\0';
}
void mystrcat(char *s, char *t) {
  int i = strlen(s); mysctcpy(s+i, t);
}
```

次は文字列比較ですが、だいたいヒントの通りです。文字列を先頭から各文字が等しい間比較していき、その等しい文字がナル文字であれば最後まで等しかったので 0 を返します。ループから抜けた時は等しくなかったので、両者の大小に応じて 1 か-1 を返します。

```
int mystrcmp(char *s, char *t) {
  int i;
  for(i = 0; s[i] == t[i]; ++i) {
    if(s[i] == '\0') { return 0; }
  }
  return (s[i] > t[i]) ? 1 : -1;
}
```

14.1.2 演習 2 — atoi と atof の実装

8 進と 16 進の両方に対応した myatoi を示します。switch 文は長くなるので個人的には if の方が好きですが、どちらでも書くことはできます。符号は前と同じ、その後「0x」ならその後ろの文字列を 16 進として解釈して累計していきます。数字が 0〜9 と a〜f に分かれているので条件が複雑です。そうでなくて 0 から始まる場合は 8 進で、こちらは単純です。それ以外は十進で前と同じですが、いちいち変数にいれないで直接扱い、for 文で反復を指定しています。

```
#include <stdio.h>
int myatoi(char *s) {
  int sign = 1, val = 0;
  if(*s == '-') { sign = -1; ++s; } else if(*s == '+') { ++s; }
  if(*s == '0' && s[1] == 'x') { // hex
    for(s += 2; *s>='0' && *s<='9' || *s>='a' && *s<='f'; ++s) {
      if(*s >= '0' && *s <= '9') { val = val*16 + (*s - '0'); }
      else { val = val*16 + 10 + (*s - 'a'); }
    }
  } else if(*s == '0') { // ocatl
    for(++s; *s>='0' && *s<='7'; ++s) { val = val*8 + (*s - '0'); }
  } else { // decimal
    for( ; *s>='0' && *s<='9'; ++s) { val = val*10 + (*s - '0'); }
  }
  return sign * val;
}
int main(int argc, char *argv[]) {
  int i;
  for(i = 1; i < argc; ++i) { printf("%d\n", myatoi(argv[i])); }
}
```

main ではコマンド引数 1 つずつを変換して表示するようにしました。実行のようすを見ましょう。

```
% ./a.out 10 012 -0x1f
10
10
-31
%
```

確かに大丈夫そうです。

atof の方は数字かどうかを判定する下請け関数を使うようにしてみました。また、指数部を取り出すのに myatoi を呼んでいます (なので指数が 8 進や 16 進で書けますがそれがいやなら前回の例題のままのものを使えばよいです)。

```
#include <stdio.h>
#include <stdbool.h>
(ここに myatoi をいれる)
bool digit(char c) { return c >= '0' && c <= '9'; }
double myatof(char *s) {
  int sign = 1, scale = 0;
  double val = 0.0;
  if(*s == '-') { sign = -1; ++s; } else if(*s == '+') { ++s; }
  for( ; digit(*s); ++s) { val = val*10 + (*s - '0'); }
  if(*s == '.') {
    ++s;
    for( ; digit(*s); ++s) { val = val*10 + (*s - '0'); --scale; }
  }
  if(*s == 'e') { scale += myatoi(s+1); }
  for( ; scale > 0; --scale) { val *= 10; }
```

```
  for( ; scale < 0; ++scale) { val /= 10; }
  return sign * val;
}
int main(int argc, char *argv[]) {
  int i;
  for(i = 1; i < argc; ++i) { printf("%g\n", myatof(argv[i])); }
}
```

scale は全体に 10 の何乗を掛けるかを保持します (指数部とも言えます) 符号はこれまでと同じ、そのあと小数点の手前部分もこれまでと同じですが、小数点があった場合はその先は val はこれまでと同様に値を累積していきますが、同じだけ scale を減らしていきます。つまり 1.234 は 1234×10^{-3} なのでその -3 を数えるわけです。ここまで終わって次の文字が指数なら指数部ですが、その整数値の取り込みは上述のように myatoi を使い、現在の scale の値に足し込みます。そのあと、scale が正ならその値ぶん繰り返し 10 倍し、負ならばその値ぶん繰り返し 10 で割れば求める値になり、最後に符号を掛けて返します。実行のようすを示します。

```
% ./a.out 1.234 1.5e5 1e-3
1.234
150000
0.001
```

14.1.3 演習 4 — パターンマッチの拡張

この演習は全部は大変なので分かりやすいもののみ、変更の要点だけ示します。まずパターンを先頭に固定する^ですが、matchstr 中で先頭位置をずらして調べるのを^のときだけやめます。

```
char *matchstr(char *str, char *pat, int len, char **tail) {
  if(len == 0) { return NULL; }
//printf("matchstr: '%s' '%s'\n", str, pat);
  if(*pat == '^') {
    char *t = pmatch(str, pat+1, str + len);
    if(t != NULL) { *tail = t; return str; }
    return NULL;
  }
  for(char *lim = str + len; str < lim; ++str) {
    char *t = pmatch(str, pat, lim);
    if(t != NULL) { *tail = t; return str; }
  }
  return NULL;
}
```

残りの機能は基本的に pmatch の中の if-else による分岐を増やして実現します (文字クラスはずっと大変なので扱っていません)。まず$はパターン側が$のとき、文字列側が最後の'\0' であれば成功でその場所を返し、それ以外は失敗とすればよいです。

```
  } else if(pat[0] == '$') {
    return (*str == '\0')? str : NULL;
```

次に*は+とよく似ていますが、0 回のマッチもありなところが違います。

```
  } else if(pat[1] == '*') {
    int i = 0;
    while(pat[0] == str[i]) { ++i; }
    for( ; i >= 0; --i) {
      char *t = pmatch(str+i, pat+2, lim);
      if(t != NULL) { return t; }
    }
    return NULL;
```

つまり、i を「0 から」探し、縮める方も「0 まで」縮めて探します。最後に?ですが、これは「0 回か 1 回」と考えて最初にのばすところで while の代わりに 1 回だけ i を増やすかどうかテストし、あとは同じにします。

```
  } else if(pat[1] == '?') {
    int i = 0;
    if(pat[0] == str[i]) { ++i; }
    for( ; i >= 0; --i) {
      char *t = pmatch(str+i, pat+2, lim);
      if(t != NULL) { return t; }
    }
    return NULL;
```

動かしてみます。

```
% ./a.out 'abcabccaba' 'abc*a'
abcabccaba
^^^^
abcabccaba
   ^^^^^
abcabccaba
       ^^^
% ./a.out 'abcabccaba' '^abc*a'
abcabccaba
^^^^
% ./a.out 'abcabccaba' 'abc*a$'
abcabccaba
       ^^^
% ./a.out 'abcabccaba' 'abc?a'
abcabccaba
^^^^
abcabccaba
       ^^^
```

14.2 C 言語の構造体機能

14.2.1 構造体の概念と定義 exam

構造体 (structure) ないし**レコード** (record) とは、複数の**フィールド** (field) が集まったデータ構造です。複数のデータという点では配列に似ていますが、配列が「同種の」データの並びで、実行時に

「何番目」という番号で要素を指定するのに対し、構造体では集まったデータそれぞれが違う型であってよく、そのためフィールドごとに別の名前をつけて扱います。

1 つのプログラムで様々な構造体型を使うこともあるので、どの構造体かを区別するために**タグ** (tag) と呼ばれる名前をつけます。タグの定義と構造体型を用いた変数宣言は分けた方が分かりやすいので、本資料ではそのようにします。その場合、タグの定義は次の形になります。

```
struct タグ名 { 変数定義… } ;
```

最後に「;」があるのに注意してください。次に変数宣言するときはこのタグを指定して次のようにします (つまり「struct タグ名」が型名になるわけです)。

```
struct タグ名 変数名 [= 初期値 ],..  ;
```

初期値を指定する場合、配列と同様、{...} で囲んだ値の並びをフィールドを定義した順番に指定します。気をつける必要があるのは、変数を宣言するときにその中身が分からないと困るので、プログラムの中でタグの定義を先に置かなければならないという点です [1] そして、構造体型の変数を定義した後は、その個々のフィールドは「変数. フィールド名」で読んだり書いたりできます。

では、色の RGB 値を扱う例題を見ていただきます。始めのほうに `struct color` という構造体の定義があります。この構造体は `unsigned char` つまり 0〜255 の整数を保持する `r`、`g`、`b` という 3 つのフィールドから成ります。

```
// color1.c --- handle color struct.
#include <stdio.h>
struct color { unsigned char r, g, b; };
void showcolor(struct color c) {
  printf("%02x%02x%02x\n", c.r, c.g, c.b);
}
struct color mixcolor(struct color c, struct color d) {
  struct color ret = { (c.r+d.r)/2, (c.g+d.g)/2, (c.b+d.b)/2 };
  return ret;
}
int main(void) {
  struct color white = { 255, 255, 255 };
  struct color c1 = { 10, 100, 120 };
  showcolor(c1);
  showcolor(mixcolor(white, c1));
  return 0;
}
```

関数 `showcolor` は、受け取った色の RGB 値をそれぞれ 16 進 2 桁ずつで表示します。この 16 進 6 桁はカラーコードとして色々な場面で使えます。関数 `mixcolor` はこの構造体を 2 つ受け取り 1 つ返します。中では、`struct color` 型の変数 `ret` を定義し、その RGB 値をパラメタとして受け取った 2 つの色の RGB 値それぞれの平均として初期化し、そして `ret` の値を返します。

`main` ですが、変数 `while` は RGB とも 255 の値の構造体になります。`c1` はもうちょっと暗めの中間色です。そして次の 2 行で `c1` および `c1` と `whilte` を混合した色を 16 進表示しています。

C 言語では配列の名前はポインタを意味するため配列をそっくり値として渡したり代入したり返すことができないのですが、構造体についてはこのようにそっくり値として渡したり代入したり返すことができます。ただし大きいデータに対してやるとそのぶんだけ実行が遅くなるので注意も必要です。では実行例を見ましょう。

[1] ポインタ変数はアドレスのビット数が分かればよいので、ポインタ変数だけはタグの定義前でも宣言できます。

```
% gcc color1.c
% ./a.out
0a6478
84b1bb
```

あと、さまざまな色を試したいときにプログラムを修正するのでは手間ですから、色を入力する関数を用意しておきますので、適宜使ってください。[2]

```
struct color readcolor(void) {
  struct color ret;
  printf("r(0-255)> "); scanf("%d", &ret.r);
  printf("g(0-255)> "); scanf("%d", &ret.g);
  printf("b(0-255)> "); scanf("%d", &ret.b);
  return ret;
}
```

このように、構造体のフィールドも「`&`」でアドレスを取ることができます。呼び出す時は「`struct color c1 = readcolor();`」などのようにします。

演習 1 上の例題をそのまま打ち込んで実行しなさい。c1 の色は別のものにしてよいです。LMS 上に 16 進 6 桁を入力してその色を表示するページを用意してあるので、それを利用してどんな色か確認すること。OK なら次のような関数を作ってみなさい。

a. 渡された色と白の平均を取って返す関数 `struct color brighter(struct color c)`。
b. 渡された色と黒の平均を取って返す関数 `struct color darker(struct color c)`。
c. RGB 値は 0〜255 なので、それぞれ「255 からその値を引く」と 0 は 255 に、255 は 0 になる。これを利用して、明るい色は暗く、暗い色は明るい色にして返す関数 `struct color reversecolor(struct color c)`。
d. R の値を G に、G の値を B に、B の値を R にコピーすることで、もとと明るさが同じくらいだけど色調が違う色ができるはずである。これをおこなう関数 `struct color rot1color(struct color c)`。ついでに R を B に、G を R に、B を G にコピーする関数 `struct color rot2color(struct color c)` も作ってみるとよい。
e. 2 つの色と 0.0〜1.0 の値を渡すとその 2 色を指定した比率で混ぜた色を返す関数 `struct color linearmix(struct color c, struct color d, double ratio)`。`ratio` が 0.5 のときは平均になるので `mixcolor` と同じになる。
f. パラメタは何も受け取らず、中で擬似乱数でランダムな色を生成し返す関数 `struct color randomcolor(void)`(擬似乱数は#12 の付録参照)。
g. その他、色を計算する何か面白い関数。

なお、実数計算をする問題では、最後にフィールドに入れるとき整数へのキャスト演算「`(int)` 式」を使って整数に変換する必要があることに注意。

14.2.2 構造体のポインタ exam

先の例題では関数にパラメタとして色の構造体値を渡し、別の構造体値を受け取っていました。これでもいいのですが、時には構造体の「アドレスを」渡して、関数の中で副作用としてその渡した構造体を書き換えてもらいたい場合もあります。そのような例を見てみましょう。

[2]演習室環境では入力書式「`%d`」で動作しますが、8 ビットの unsigned の入力なので本来は「`%hhu`」とすべきです。ただしこの書式は C99 以降で使えるものなので、普通に整数に読み込んでレコードのフィールドに代入する方がよいかも知れません。

```
// color2.c --- handle color struct with pointer.
#include <stdio.h>
struct color { unsigned char r, g, b; };
(showcolor をここに)
(readcolor をここに)
void makedarker(struct color *p) {
  p->r = p->r / 2; p->g = p->g / 2; p->b = p->b / 2;
}
int main(void) {
  struct color c1 = readcolor(); showcolor(c1);
  makedarker(&c1); showcolor(c1);
  return 0;
}
```

main の方から見ると、readcolor で色を読み込んで c1 に入れ、その色を表示し、次に c1 のアドレスを渡して makedarker を呼びます。最後に再度表示しますが、そのときは c1 は暗い色に変わっているはずです。

では makedarker ですが、struct color のポインタ p がパラメタで…その先は何でしょうか? ポインタの参照たどり演算は「*p」ですから、渡された構造体のたとえば r フィールドをアクセスしたければ「(*p).r」となるはずです。実はそのように書いてもまったくよいのですが、C 言語では「構造体のポインタ」を良く使うため、「(*p).r」を「p->r」と書いてもよいようになっています。これを**アロー演算子**と呼びます。どちらでもよいので、makedarker の本体の 1 行はこう書いてもよいわけです。

```
(*p).r = (*p).r / 2; (*p).g = (*p).g / 2; (*p).b = (*p).b / 2;
```

好みの問題もありますが、だいたいはアロー演算子の方が読みやすいといえるでしょう。そしてこの行の動作ですが、RGB とも明るさを半分にしているので、暗い色に変化するというわけです。

演習 2 上の例題をそのまま動かし、暗い色ができることを確認しなさい。OK なら次のような関数を作ってみなさい。

a. 色を明るく変化させる関数 void makebrighter(struct color *p)。
b. 先の演習の reversecolor と同じ変化を施す関数 void makereverse(struct color *p)。
c. 先の演習の rot1color、rot2color と同様の変化をおこなう関数 void makerot1(struct color *p)、void makerot2(struct color *p)。
d. RGB 値の増分 (マイナスでもよい) を受け取り、その分だけそれぞれの成分を増やす関数 void addtocolor(struct color *p, int dr, int dg, int db)。[3]
e. RGB 値それぞれに-10〜10 の範囲のランダムな値を足すことで元とちょっとだけ違う色にする関数 void varcolor(struct color *p)。
f. その他色の構造体のアドレスを受け取り、好きな変化を施す関数。

14.3 表と探索

14.3.1 構造体の配列による表

構造体を使う例として**表** (table) と**表の探索** (table lookup) を取り上げます。プログラミングの分野では表とは「**鍵** (key) となる値を指定して**値** (value) を読み書きできする」ようなものを言います。

[3] unsinde char の値は 0〜255 に固定されているので、この範囲を超えたら 256 の剰余が取られてこの範囲に入れられる。このため、範囲を超えることは心配しなくてよい。

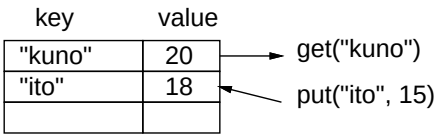


図 14.1: 表とその概念

たとえば図 14.1 は、鍵が文字列、値が整数であるような表を示しています。文字列の鍵"kuno"を指定しして取り出すと、対応する値「20」が取れます (表に指定した鍵が入っていない場合は「ない」ことを示す何らかの値が返されることにします)。また鍵"sato" を指定して値「18」を書き込むと、これまでの値の代わりにこの「18」が記録されます (表に指定した鍵が入っていない場合は新たにその鍵と値が追加されますが、表が満杯で追加に失敗することもあります)。

さて、この表はどのようにして実現したらいいでしょうか。すぐ思い付くのが、テーブルの 1 項目を構造体の値とし、それを並べた配列で表すものです。実際に見てみましょう。

```
// tbllinear1 --- table with linear array.
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include "tbl.h"
#define MAXTBL 1000000
struct ent { char *key; int val; };
struct ent tbl[MAXTBL];
int tblsize = 0;

int tbl_get(char *k) {
  int i;
  for(i = 0; i < tblsize; ++i) {
    if(strcmp(tbl[i].key, k) == 0) { return tbl[i].val; }
  }
  return -1;
}
bool tbl_put(char *k, int v) {
  int i;
  for(i = 0; i < tblsize; ++i) {
    if(strcmp(tbl[i].key, k) == 0) { tbl[i].val = v; return true; }
  }
  if(tblsize+1 >= MAXTBL) { return false; }
  char *s = (char*)malloc(strlen(k)+1);
  if(s == NULL) { return false; }
  strcpy(s, k); tbl[tblsize].key = s; tbl[tblsize].val = v;
  ++tblsize; return true;
}
```

見慣れない#include があって main がないですが、そこは保留して#define から読みます。1 項目は文字列の key、整数の val の 2 つのフィールドを持つ struct ent で、それが MAXTBL 個並ぶ配列が表です。入っている個数を表すのに変数 tblsize を使用します (初期値は 0)。

表を検索するには、値の入っている範囲内を順に指定された鍵と等しいかどうか調べて、等しい項目があれば対応する値を返します。最後まで一致しなければ、「ない」印として -1 を返します (この

表には 0 以上だけ入れるつもり)。

登録時は同様に調べて行き、鍵が等しい項目があればそこの val に値を書き込み、「はい」を返します。最後まで一致しなかった場合は、新たな項目を追加し、鍵と値を覚えます (ただし tblesize を増やした時に最大個数を超えるなら追加できないので「いいえ」を返して終わります)。追加する際は、値は整数だから代入でよいですが、鍵が文字列なので注意が必要です。文字列は実際は配列の先頭を指していて、その配列は入力用の領域だったりしますね。そうすると、その場所を覚えていても、次の入力時に中身が書き変わってしまいます。そこで、**malloc**(memory allocate) という新しい関数が出てきます。この関数は「指定サイズのあき領域を確保して領域の先頭を返す」関数です。これに文字列の長さ+1(末尾のナル文字のぶん) を渡して、戻された値を文字列へのポインタ型にキャストし、変数 s に入れます (図 14.2)。

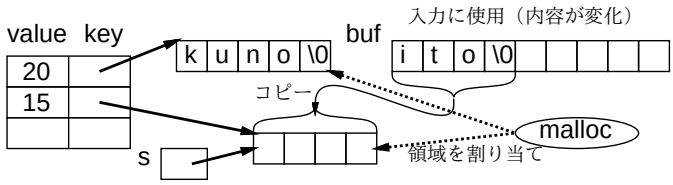


図 14.2: malloc による領域の割り当て

もし malloc がメモリ不足で失敗した場合は NULL が返ってくるので、その場合は「いいえ」を返します。OK なら次に sctcpy でパラメタ k にある文字列をここへコピーします。そして、表の key フィールドにはこのポインタ、val には渡された値を入れて、tblsize は増やして「はい」を返します。

この実現のように、項目を端から順に一致を探すやり方を**線形探索**と呼びます。データが N 個入っていた場合、平均して半分の位置で見つかるとして、見つかる際合の比較数が $\frac{N}{2}$、見つからない際は N ですから、見つかる比率がいくらでも、線形探索では 1 項目の get/put にかかる時間計算量は $O(N)$ になります。

14.3.2 ファイルの分割とヘッダファイル

さて、main はどこでしょう。唐突ですがここで、上で呼んだ表の機能 (get と put で使える) と、main やその下請け関数とを、別ファイルに分けることにします。両方で合わせなければならないのは、関数 get と put の呼び方だけです。そこで、**ヘッダファイル** tbl.h に次の 2 行を入れます。

```
bool tbl_put(char *k, int v);
int tbl_get(char *k);
```

これらは**プロトタイプ宣言**と呼び、関数定義の先頭部分 (定義本体の「{…}」を「;」に変更したもの) です。そして#include "tbl.h"はファイル内容をそこに取り込みます。[4] なぜこれが必要なのでしょう。これまで、main で呼び出す関数は main より前に書いていました。これにより、main 中の呼ぶ箇所では関数の引数や返値の型が分かっていて、型検査できます。しかしそうしない場合…定義を main より下に書いたり、(この例のように) 別ファイルに分けると、型情報がありません。この問題を回避するため、関数の先頭部分を書いてパラメタや返値の情報を教えるのが、プロトタイプ宣言の役割なのです。[5]

```
// tbltest1 --- test table functions
#include <stdio.h>
#include <string.h>
```

[4] #include においてダブルクォートでファイル名を囲んだ場合は、現在位置にあるヘッダファイルを取り込めます (相対パス名や絶対パス名も指定可能)。

[5] なので、別に#include を使わなくても上の 2 行を main の上に書けばそれでもよいです。しかし tbl.c でも同じプロトタイプ宣言を取り込むことで間違いのチェックができるので、ヘッダファイルに分離してそれぞれ取り込むのが通例です。

```
#include <stdbool.h>
#include <stdlib.h>
#include "tbl.h"
(getl をここに)
int main(void) {
  char b1[100], b2[100];
  int val;
  while(true) {
    printf("key (empty for quit)> ");
    if(!getl(b1, 100) || strlen(b1) == 0) { return 0; }
    printf("val (-1 for query)> "); getl(b2, 100); val = atoi(b2);
    if(val != -1) { tbl_put(b1, val); }
    else          { printf("tbl[%s] == %d\n", b1, tbl_get(b1)); }
  }
}
```

main ですが、無限ループで繰り返しテストできるようにしています。ループの先頭でプロンプトを出して getl で 1 行読みます。読めない場合と、読めたけれど長さが 0 だった場合は終わります (! は論理否定演算子)。そうでない場合はプロンプトを出して記録する値を読みますが、−1 のときは問い合わせて結果を表示、そうでない場合は値の登録を行います。

では動かしてみましょう。コンパイル時に 2 つのファイルを指定する点がこれまでとちょっと違います。

```
% gcc tbltest1.c tbllinear1.c  ←2 つ指定してコンパイル
% ./a.out
key (empty for quit)> kuno
val (-1 for query)> 20       ← kuno に 20 を登録
key (empty for quit)> kuno
val (-1 for query)> -1       ← kuno を検索
tbl[kuno] == 20
key (empty for quit)> ito    ← ito を検索
val (-1 for query)> -1
tbl[ito] == -1               ←未登録
key (empty for quit)> ito
val (-1 for query)> 18       ← ito に 18 を登録
key (empty for quit)> ito
val (-1 for query)> -1       ← ito を検索
tbl[ito] == 18
key (empty for quit)> ito
val (-1 for query)> 15       ← ito を変更
key (empty for quit)> ito
val (-1 for query)> -1
tbl[sato] == 15
key (empty for quit)>         ← [RET] で終わる
%
```

演習 3 上の例題をそのまま打ち込んで動かしなさい。動いたら次の変更をしてみなさい。

a. 登録できる値を整数 1 個から変更しなさい (整数 2 個とか文字列とか)。

b. 今は表は追加と書き換えしかできないが、削除機能をつけてみなさい。

c. 表の中身を全部まとめて表示する機能をつけてみなさい。
(ヒント: この機能そのものは tbllinear1.c の中に置くのが自然で、main からそれを呼び出す。どういう場合にこの機能が呼ばれることにするかは好きに決めてかまいません。)

d. そのほか、面白いと思う機能をつけてみなさい。

14.3.3 C 言語における時間計測

先に述べたように、線形探索による表は 1 項目あたりの get/put の時間計算量が $O(N)$ です。これを計測によって確認してみましょう。そのために次のような計測プログラムを書きました。表の実現部分のファイルは変更する必要がないことに注意。

```
// tblbench1 --- benchmark table performance
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include "tbl.h"

int main(int argc, char *argv[]) {
  if(argc != 2) { fprintf(stderr, "need count.\n"); return 1; }
  int i, count = atoi(argv[1]);
  struct timespec t1, t2;
  clock_gettime(CLOCK_REALTIME, &t1);
  for(i = 0; i < count; ++i) {
    char buf[100];
    sprintf(buf, "s%d", (int)(drand48()*10000000));
    tbl_put(buf, i+1);
    int k = tbl_get(buf);
  }
  clock_gettime(CLOCK_REALTIME, &t2);
  int msec = 1000*(t2.tv_sec-t1.tv_sec) +
             (t2.tv_nsec-t1.tv_nsec)/1000000;
  printf("%d\n", msec);
}
```

このプログラムは指定した回数だけ乱数で生成した文字列データと数値データを表に put してすぐ同じものを get します。乱数は前に学んだ drand48() を使い、これに百万をかけて整数にしたものを先頭に「s」という文字をつけて文字配列 buf に生成します。**sprintf** というのは printf とそっくりで、ただし整形出力をファイルに出力するかわりに第 1 引数の文字配列に書き込みます。これを鍵として (書く場合は値はその整数値に 1 足したものとして)、get/put を呼びます。

そして、上記の繰り返し処理全体にかかる時間を測ります。そのために、ヘッダファイル<time.h>で定義されているライブラリ関数 clock_gettime を使います。

この関数は第 1 引数として定数 CLOCK_REALTIME(前記ヘッダファイルで定義) を渡すと、その呼んだ時点での 1970.1.1 の 0:00:00 からの経過時間を第 2 引数の構造体で受け取れます。構造体の定義 (これも前記ヘッダファイルにある) は次のようになります。

```
struct timespec {
```

```
            time_t  tv_sec;         /* seconds */
            long    tv_nsec;        /* and nanoseconds */
    };
```

つまり、非常に細かい値も計測できる場合に備えて、秒数と秒に満たないナノ秒数とを別々に受け取ります。clock_gettime を上記のループの前と後で呼び、両方の時間を引き算して (ナノ秒では細かすぎるので) ミリ秒数に直し、それを表示しています。

では実際に計測してみましょう。

```
% gcc tblbench1.c tbllinear1.c
% ./a.out 1000
9
% ./a.out 2000
36
% ./a.out 3000
85
%
```

このように、データ数が 2 倍、3 倍になると所要時間が 4 倍、9 倍になり、時間計算量が $O(N^2)$ であると分かります。それは当然で、データ数が 2 倍になると、1 回の所要時間が 2 倍になり、その反復回数も 2 倍になるから、掛け算して 4 倍なわけです。

ところでここに、同じ呼び出し方で使える表の別の実装があります。それを計測してみましょう。

```
% ./a.out 1000
0
% ./a.out 10000
5
% ./a.out 20000
10
% ./a.out 30000
15
%
```

このように 1000 では速すぎて計測できず、1 万から 2 倍、3 倍としたときに時間も 2 倍、3 倍となっています。つまり、要素 1 個の get/put にかかる時間は定数 $O(1)$ ということですね。その仕組みはすぐ後で説明します。

演習 4 自分でも線形探索の表を時間計測し、$O(N^2)$ の時間計算量であることを確認しなさい。

14.3.4 ハッシュ表と動的データ構造

1 回の get/put が $O(1)$ の表はどうやって作れるのでしょうか。1 つのヒントは、普通の配列のアクセス a[i] は読むときも書くときも一定の時間でよい、ということです。ですから、たとえば鍵が 0〜9999 の整数であれば、その大きさの配列を取れば $O(1)$ の表になります。

しかし今回は鍵が文字列ですから、この方法は使えません。また、実数値や配列が確保できなような大きい整数を鍵にする場合も同様です。そこで、「ある規則にしたがって、文字列 s から一定範囲の整数に変換する」関数を作ります。これを**ハッシュ関数** (hash function) と呼びます。たとえば、文字列"kuno"でこの関数を計算して場所を決めて格納したとすれば、後でまた"kuno" で検索したときも同じ関数で計算することで場所が分かり、すぐ取り出せるはずです。これが**ハッシュ表** (hash table) の基本的な考えです。

ただし、運が悪いと別の文字列でも関数の計算結果が同じ値になるかも知れません。これを**衝突** (collision) と言います。衝突の対処方法はいくつかありますが、ここでは衝突したときに同じ場所に複数の値が入れられるように単連結リストを使います (図 14.3)。

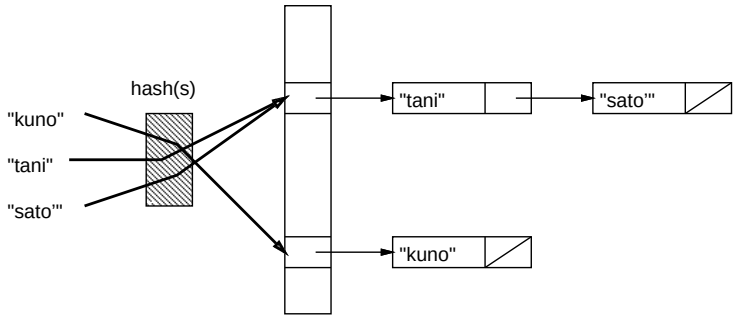


図 14.3: 単連結リストを使うハッシュ表

ではコードを見てみましょう。ハッシュ表の配列サイズが 9973 となっていますが、これは 1 万を超えない素数を選んだものです。ハッシュ表ではハッシュ関数の値がなるべく「バラバラに」ばらけて衝突が少ないことが重要なので、最後に表の範囲に入れるために表サイズで剰余を取りますが、そのとき素数の剰余の方がばらけやすいと考えているからです。

そして、構造体は単連結リストにするのでフィールドとして鍵、値に加えて次のセルを指す `next` があります。表そのものはセルを指すポインタの配列です。本来ならこの各要素に `NULL` を入れるべきですが、C 言語ではグローバル変数には数値なら 0、ポインタなら `NULL` が入っていることになっているので、初期化は省略しています。

次にハッシュ関数ですが、先頭に `static` とついています。これはこのファイル内だけで有効な関数という意味で、外部の関数と名前が衝突しないので短い名前を安心して使うことができます。あと、返す値も作業変数 `v` も符号なし整数です (マイナスの数が出て来ると扱いずらいため)。`v` の初期値は 1 で、文字列の 1 文字ごとにその文字コードを 11 倍して 1 を足したものを掛け算します。これは、素数倍がばらけやすいからとか、次々に掛けていくときたまたま 0 になって以後 0 のままになるのを防ぐため 1 足すなどの工夫をしたためです。最後まで掛け算したら上記のようにサイズで剰余を取って返します。

```
// tblchash --- table impl. with chained hash.
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include "tbl.h"
#define MAXTBL 9973
struct ent { char *key; int val; struct ent *next; };
struct ent *tbl[MAXTBL];

static unsigned int hash(char *s) {
  unsigned int v = 1;
  while(*s) { v *= 11 * (*s++) + 1; }
  return v % MAXTBL;
}
static struct ent *lookup(struct ent *p, char *k) {
  for( ; p != NULL; p = p->next) {
    if(strcmp(p->key, k) == 0) { return p; }
  }
```

```
    return NULL;
  }
  static bool get1(struct ent *p, char *k) {
    struct ent *q = lookup(p, k);
    return (q == NULL) ? -1 : q->val;
  }

  static bool put1(struct ent **p, char *k, int v) {
    struct ent *q = lookup(*p, k);
    if(q != NULL) { q->val = v; return true; }
    q = (struct ent*)malloc(sizeof(struct ent));
    if(q == NULL) { return false; }
    int len = strlen(k);
    q->key = (char*)malloc(len+1);
    if(q->key == NULL) { return false; }
    strcpy(q->key, k);
    q->val = v; q->next = *p; *p = q; return true;
  }
  int tbl_get(char *k) { return get1(tbl[hash(k)], k); }
  bool tbl_put(char *k, int v) { return put1(&tbl[hash(k)], k, v); }
```

外部から呼ぶ getr/put は一番下の 2 行です。これらは、配列中のハッシュ関数で計算した場所の値 (put では場所のアドレス) と鍵 (と put では値) を持って下請け関数を呼びます。

次は `lookup` を読みましょう。for 文ですが、初期化のところは使わないので空になっていて、ポインタ値 `p` が `NULL` でない間くりかえし、次のセルをたどって行きます。たどっていく中で `key` のフィールドが渡された `k` と等しい文字列なら、そのセルのポインタを返します。最後まで見つからなかったら `NULL` を返します。

では次に下請け関数 `get1` ですが、これは `looup` を呼んで結果が `NULL` なら -1、そうでなければ返されたセルの `val` フィールドを返すだけです。

最後に `put1` ですが、これは第 1 引数がポインタのポインタになっています。なぜかというと、新たなセルを作るためにその場所に値を入れる必要が生じるかも知れないためです。`lookup` にも `*p` を渡し、返った結果が `NULL` でなければその値のセルがあったので、`val` フィールドに値を入れて「はい」で返ります。見つからなかったのなら、新たなセルが必要です。`malloc` でセルの領域を割り当て、変数 `q` に入れます。もし `NULL` なら失敗で返ります。OK なら今度は文字列の領域を割り当てて `q->key` に入れます。これも `NULL` なら失敗で返ります。OK なら文字列をコピーし、値を入れます。そして `next` にはこれまでの`*p` を入れ、最後に`*p` にこの `q` を入れれば、新たなセルが単連結リストの先頭に挿入されたことになります。

演習 5 このハッシュ表の実装を入力し、先の計測プログラムと一緒にして計測してみなさい。回数が多くなると $O(1)$ でなくなりますが、その理由を検討し、その問題を解消するような変更を行ってみなさい。

(ヒント: 多く登録しすぎるとそうなるので、登録数を調べて一定以上」登録しようとしたら満杯ということにするのが方法の 1 つです。)

演習 6 連結リストを使わないでハッシュ表を作る次の方法があります (図 14.4)。

- エントリは「値が入っている」「入っていない」を区分できるようにする (文字列が鍵なら NULL ポインタのとき入っていないということにすればよい)。
- ハッシュ関数を 2 つ用意する。

- 登録時は 1 つ目のハッシュ関数で選んだ位置 i に衝突があったら、2 つ目のハッシュ関数で値 d を計算し、$i+d$、$i+2d$、…を調べていって空いている位置にいれる。検索時は同様にして空いている位置に来たら登録されていないことが分かる。

なお、ハッシュ表の端まで来たら先頭に戻ることにします。また、ハッシュ表のサイズを素数にしておくことで、同じ $i+d$ に戻って来てしまうことがなくせます。表が満杯だと検索が止まらなくなるので、いくつ入れたか数えて管理することは必須です。この方法を**ランダムリハッシュ**(random rehash) と呼びます。この方法のハッシュ表を実現してみなさい。性能計測もすること。

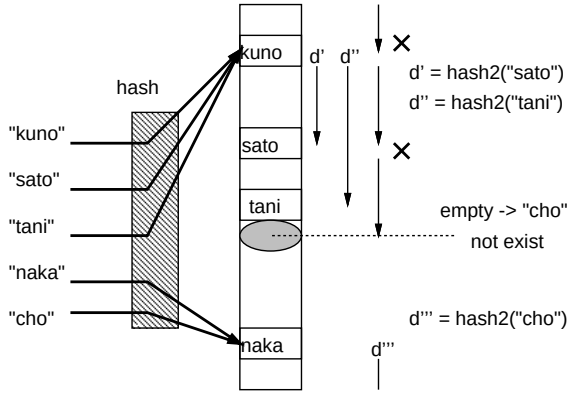


図 14.4: ランダムリハッシュ法

本日の課題 14A

「演習 1」または「演習 2」で動かしたプログラム (どれか 1 つでよい) を含むレポートを提出しなさい。プログラムと、簡単な説明が含まれること。アンケートの回答もおこなうこと。

Q1. C 言語の構造体機能についてどのように思いましたか。

Q2. C 言語でファイルを複数に分ける方法が分かりましたか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 14B

「演習 1」〜「演習 6」の (小) 課題から 1 つ以上を選択してプログラムを作り、レポートを提出しなさい。プログラムと、課題に対する報告・考察 (やってみた結果・そこから分かったことの記述) が含まれること。アンケートの回答もおこなうこと。

Q1. 構造体を使ったプログラムが書けるようになりましたか。

Q2. 表と検索とはどういうことか理解しましたか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

＃15　チームによるソフトウェア開発 (総合実習)

今回の内容は「総合実習」であり、2〜3 名のグループで協力して「動画を生成する整った構造のプログラム」を開発して頂きます。したがって課題は「B 課題」のみです。今回の目標は次のことがらです。

- チームでソフトウェアを開発する際に注意すべきことを知る。
- C 言語の機能を活用して分担してプログラムを開発する。

15.1　前回演習問題の解説

15.1.1　演習 1 — 色の構造体

色の構造体の演習は簡単だと思います。実数計算する場合は、最後に整数に戻すため `int` へのキャストが必要なことに注意してください。また、`randomcolor()` では `rand()` に対し 256 で剰余を取り 0〜255 の乱数を得ます。通常の `rand()` の使用と同様、`main` で `srand(time(NULL))` が必要です。

```
struct color brighter(struct color c) {
  struct color white = { 255, 255, 255 }; return mixcolor(c, white);
}
struct color darker(struct color c) {
  struct color black = { 0, 0, 0 }; return mixcolor(c, black);
}
struct color reversecolor(struct color c) {
  struct color ret = { 255-c.r, 255-c.g, 255.c.b }; return ret;
}
struct color rot1color(struct color c) {
  struct color ret = { c.b, c.r, c.g }; return ret;
}
struct color rot2color(struct color c) {
  struct color ret = { c.g, c.b, c.r }; return ret;
}
struct color linearmix(struct color c, struct color d, double p) {
  double q = 1.0 - p;
  struct color c1 = {
    (int)(c.r*p+d.r*q), (int)(c.g*p+d.g*q), (int)(c.b*p+d.b*q) };
  return c1;
}
struct color randomcolor(void) {
  struct color c1 = { rand()%256, rand()%256, rand()%256 };
  return c1;
}
```

15.1.2 演習 2 — 構造体のポインタ

個別に計算してもよいのですが、上で定義したものを使ってポインタ参照の先に書き込むこともできるので、そのような版も示します。

```
void makebrighter(struct color *p) {
  p->r = (255+p->r)/2; p->g = (255+p->g)/2; p->b = (255+p->b)/2;
}
void makebrighter2(struct color *p) { *p = brighter(*p); }
void makereverse(struct color *p) {
  p->r = 255-p->r; p->g = 255-p->g; p->b = 255-p->b;
}
void makerevese2(struct color *p) { *p = reversecolor(*p); }
void makerot1(struct color *p) {
  unsigned char x = p->r; p->r = p->b; p->b = p->g; p->g = x;
}
void makerot12(struct color *p) { *p = rot1color(*p); }
void addtocolor(struct color *p, int dr, int dg, int db) {
  p->r += dr; p->g += dg; p->b += db;
}
void varcolor(struct color *p) {
  p->r += (int)(21*drand48()-10);
  p->g += (int)(21*drand48()-10);
  p->b += (int)(21*drand48()-10);
}
```

15.1.3 演習 3 — 線形探索の表

演習 3 は線形探索の表に機能を追加するものでした。ここでは削除と全部表示の 2 つを示します。まず tbl.h にこれらの関数の宣言を追加します。

```
bool tbl_delete(char *k);
void tbl_show(void);
```

tbl_delete が bool を返すのは、見つかって削除したかそのキーの項目は無かったかの区別を返すというつもりです。そして削除ですが、これまでと同様に探して見つかったときは削除します。削除するときは表の最後の項目をその位置にコピーしてきて表のサイズを 1 つ減らせばよいですが、サイズが 1 のときはコピーしてくるものがないのでコピーしません。あと、malloc で割り当てた文字列領域は不要になったら free で返却するべきなのでそうしています。

```
bool tbl_delete(char *k) {
  int i;
  for(i = 0; i < tblsize; ++i) {
    if(strcmp(tbl[i].key, k) == 0) {
      free(tbl[i].key);
      if(tblsize > 1) { tbl[i] = tbl[tblsize-1]; }
      --tblsize;  return true;  // found and deleted
    }
  }
  return false; // not found
}
```

そして全部表示はむしろ簡単ですね。

```
void tbl_show(void) {
  int i;
  for(i = 0; i < tblsize; ++i) {
    printf("%s: %d\n", tbl[i].key, tbl[i].val);
  }
}
```

これらをテストする main の方も変更しました。-2 を入力したとき削除、そして終わるときに全部表示します。

```
int main(void) {
  char buf[MAXBUF];
  int val;
  while(true) {
    printf("key (empty for quit)> ");
    if(fgets(buf, MAXBUF, stdin) == NULL) { return 1; }
    chopnl(buf, MAXBUF);
    if(strlen(buf) == 0) { break; }
    printf("val (-1 for query, -2 for del)> ");
    scanf("%d", &val);
    if(val == -1) {
      printf("tbl[%s] == %d\n", buf, tbl_get(buf));
    } else if(val == -2) {
      tbl_delete(buf);
    } else {
      tbl_put(buf, val);
    }
    if(fgets(buf, MAXBUF, stdin) == NULL) { return  1; }
  }
  tbl_show(); return 0;
}
```

では実行例です。

```
% ./a.out
key (empty for quit)> kuno
val (-1 for query, -2 for del)> 5
key (empty for quit)> nakano
val (-1 for query, -2 for del)> 10
key (empty for quit)> sasaki
val (-1 for query, -2 for del)> 15
key (empty for quit)> kuno
val (-1 for query, -2 for del)> -2
key (empty for quit)>
sasaki: 15
nakano: 10
%
```

削除するときに最後の要素をコピーしてくるので、最後に全表示すると `sasaki` の方が前になっているのがわかります。

15.2 チームによるソフトウェア開発

15.2.1 ソフトウェア開発の難しさ

ここまで様々なプログラムについて扱ってきましたが、世の中では「ソフトウェア」という用語の方が多く使われます。一般に**ソフトウェア** (software) とは、プログラムとそれを動かすのに必要なデータ等を合わせたものを言います。

世の中のソフトウェアは数十万行以上のプログラムコードを含むものも珍しくなく、そのようなものは一人では開発できないので、チームで開発することになります (もちろん、もっと小さい規模でも必要があればチーム開発が行なわれます)。

一人でただプログラムを作るのと比較して、チームによるソフトウェア開発では次のような追加の作業が必要です。

- 仕様策定 — どのようなソフトウェアを作るかを決める。
- 設計 — プログラムやデータの構成や形を決める。
- 開発管理 — 分担やスケジュールを決めて進捗を管理しながら開発する。
- テスト・デバッグ — 作成したソフトが仕様通り動くか検査し不具合があれば修正する。
- 運用・保守 — ソフトを動かしつつ改訂や不具合修正を行なう。
- 文書化・記録 — 上記すべての作業について記録して残す。

実際には 1 人であっても、ソフトウェアを「きちんと」作るのであればこれらの作業はすべて必要なことです。

さらに、ソフトウェアが大規模で関係する人数が多くなると、これらの作業内容を複数人で打ち合せて調整する手間が非常に大きくなります。このため、ただプログラムを書くのであれば 1 日に何百行も書けるようなソフトウェア開発者でも、仕事としてプロジェクトでソフトウェア開発を行なう場合は、平均すると 1 人あたり 1 日数行程度しか書いていない計算になると言われています。

15.2.2 ソフトウェア工学とソフトウェア開発プロセス

過去においても現在においても、実際にソフトウェア開発をおこなうと、様々なトラブルが発生しています。典型的なものをいくつか挙げます。

- どのようなソフトウェアを作るのかの仕様策定がいつまでも終わらずに開発に入れない。
- 仕様策定して開発したものができあがってみると発注側からこれでは使えないと言われる。
- 仕様を決定して開始したはずなのに途中で変更が次々に現れてつぎはぎだらけのソフトウェアになる。
- プログラムの品質が悪くバグだらけでいつまでも開発が終わらない。
- プログラムが完成して運用に入るが重大なバグが残っていてトラブルが発生する。

このような問題の多くは、ソフトウェア開発が非常に緻密な作業であり 1 箇所の間違いでも重大な障害につながる可能性があるということに由来しています。また、最後にソフトができあがって動かしてみないとどのようなものか分からないから、という面もあります。

このような多くの問題を何とかしようとする研究の分野を**ソフトウェア工学** (software engineering) と呼びます。その名前からすると「プログラムを作ることの研究」みたいですが、実際には発注者にきちんと確認するとか進捗を管理するとか、人間にまつわる事柄の多い分野です。

その中に、どのような流れでソフトウェアを開発するか、という分野があり、そこでは具体的な開発の進め方のことを**ソフトウェア開発プロセス** (software development process) と呼んでいます。

過去においては「分析→設計→製作→テスト→保守」のように段階を経てソフトウェアを開発していくプロセス (ウォーターフォール型) が主流でしたが、このやり方だと「分析・設計の結果が後になって違っていると分かって手戻りになる」問題が大きいと分かってきました。

そのため今日では、作成する機能の優先順位つきリストを作り、優先度の高い機能をとりあえず実現して動かし、動作を確認してから次の機能を追加することを反復していくプロセス (アジャイル型) が広まって来ています。

皆様がソフトウェアを開発するときも、後者のやり方を強く勧めます。最初に大きな完成形を描いてしまうと、そこまでの道のりが遠くて挫折しやすいですし、完成するまで動かないので組み立てて動かそうとしたときにはコードが大量になっていて、どこが間違っているか分からず困る、ということになるからです。

本資料でもいくつか、やや大きなプログラムの例がありましたが、いずれも「まず小さく作って動かす」「動いたら徐々に機能を増やして行く」というやり方で説明されています。この方法だと、機能を追加したときにトラブルが起きたら、その追加したあたりを見ればよいと分かっているので、はるかに問題を解決しやすいのです。

15.2.3 C 言語の機能と共同作業

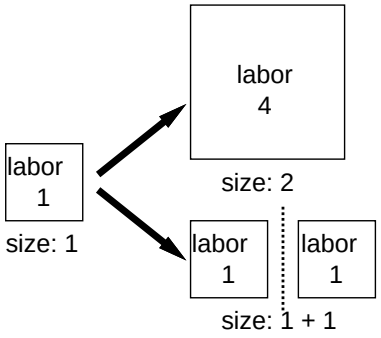


図 15.1: プログラムを独立した部分に分ける必要性

大きなプログラムを設計するときに重要な指針として、「部分ごとの独立性を高める」ということがあります。一般に、サイズ S のプログラムに対して、その 2 倍、$2S$ のサイズのプログラムは作ったり理解したりする労力が 4 倍くらいかかる、という面があります。それは、プログラムのどの箇所でもほかの様々な箇所と相互作用する可能性があるから、サイズが 2 倍に掛け算してそれぞれの部分の相互作用が 2 倍になる、と考えればよいでしょう (図 15.1 上)。

ここでプログラムの設計を見直し、そのプログラムを互いにほとんど関係しない (数箇所で呼び出すだけの)2 つの部分に分けることができたら、$S + S$ でもとの 2 倍の労力で開発できます (図 15.1 下)。すなわち、プログラムの部分どうしができるだけ関係しないようにすることが大切なのです。

C 言語でそのような分離を実現するには、「特定用途の機能の集まり」を 1 つのファイルに入れる形で行なうのが定石です。既に見てきたように、C 言語ではグローバル変数に `static` を指定することで、その変数がファイル内だけで参照できるものになります。それぞれのファイルをそのようにすることで、個々のファイル内のコードは他のファイルからほとんど独立したものとできます。

あるファイル内から別のファイルで実現されている機能を利用するには、もちろんそのファイルの関数を呼び出します。そのためにはプロトタイプ宣言が必要ですが、それはファイルごとに対応するヘッダファイルに用意すればよいのです。そして各ファイルにおいて、よそのファイルから呼び出されることを想定しない関数は、やはり `static` を指定してよそから参照できなくしておきます (図 15.2)。

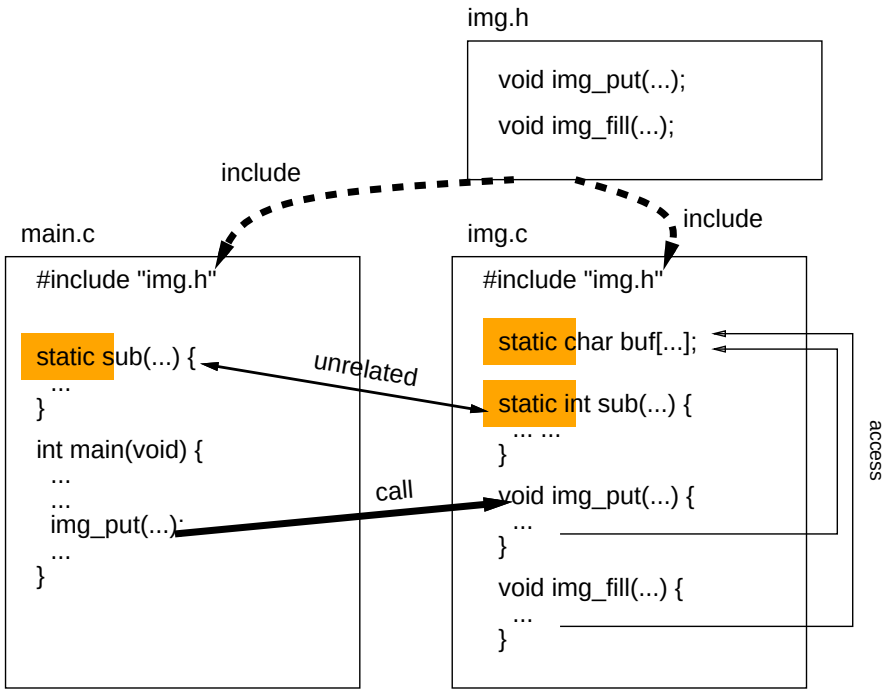


図 15.2: C 言語でファイルを分けて扱う

一般に、ひとまとまりの機能に対して、その機能を外部から (たとえばよそのファイルから) 呼び出すときに使う関数の集まりのことを **API**(application programming interface) と呼びます。プログラムをうまくいくつかの機能に分け、それぞれの機能ごとにうまく設計された API を定義し、それらを呼ぶことでプログラム全体が動作する、というのが整ったプログラムの形だと考えてよいでしょう。

15.3 動画ファイルの API を作る

15.3.1 API の設計

それでは今回の例題として、Ruby でも扱った PPM 画像ファイルの出力を取り上げます。ただし今回は、動画を生成するために多数の PPM 画像ファイルを出力することを想定します。そこで、API を定義するヘッダファイル `img.h` を見てみましょう。

```
#define WIDTH 300
#define HEIGHT 200
struct color { unsigned char r, g, b; };
void img_clear(void);
void img_write(void);
void img_putpixel(struct color c, int x, int y);
void img_fillcircle(struct color c, double x, double y, double r);
```

まず画像の幅と高さはここで定義しています。次に色については前にやったように構造体 `struct color` で定義しています。そして残りの関数は次のようにします。

- `img_clear` — 画像を真っ白に初期化する。
- `img_write` — 現在の画像を PPM 形式でファイルに書き出す。ファイル名は `img`*dddd*`.ppm` に固定で、*dddd* のところは 0001, 0002, ... と書くごとに番号が進んで行くものとする。
- `img_putpixel` — 指定した色で指定した (x, y) 位置に点を打つ。
- `img_fillcircle` — 指定した色で指定した (x, y) を中心とし半径 r の円を塗りつぶす。

ここでなぜ円の方だけ座標や半径が実数なのかと思ったかも知れませんが、アニメーションをやるということは座標や半径を連続的に変化させたいので実数が便利なのです。putpixel の方は直接呼ぶことはあまりなさそうなので整数のままにしました。

15.3.2 API の実装

では上で設計した API の実装を見ます。画像データを入れる配列 buf は構造体の 2 次元配列ではなく、文字の 3 次元配列にしました。その理由は、C 言語では 3 バイトの大きさの構造体を配列にするとアクセスを高速にするため 1 バイトの「詰めもの」をして 4 バイトにするという機能がくっついていて、そうすると画像としてそのまま書き出せないからです (難しいと思いますがそういうものだと思ってください)。文字だけの配列ならこのようなことは起きません。変数 filecnt はファイルにつける連番、fname はファイル名生成用の領域です。

クリアは簡単で、すべてのピクセルの RGB 値をすべて 255(真っ白) にします。書くときは、まずファイル名を fname に生成し、次に fopen でその名前のファイルを書き出しモードで準備します。ファイル生成が失敗すると NULL が返されるのでそのときはエラーメッセージを出して終わります。OK なら、そのファイルにまず PPM 画像のヘッダ部分「P6、幅 高さ、255」を書き、続いて buf 全体をいっきに出力します。fwrite は指定したポインタ値の場所から指定したバイト数のかたまりを N 個ぶん (今回は 1 個を指定) 書き出します。

```
#include <stdio.h>
#include <stdlib.h>
#include "img.h"
static unsigned char buf[HEIGHT][WIDTH][3];
static int filecnt = 0;
static char fname[100];
void img_clear(void) {
  int i, j;
  for(j = 0; j < HEIGHT; ++j) {
    for(i = 0; i < WIDTH; ++i) {
      buf[j][i][0] = buf[j][i][1] = buf[j][i][2] = 255;
    }
  }
}
void img_write(void) {
  sprintf(fname, "img%04d.ppm", ++filecnt);
  FILE *f = fopen(fname, "wb");
  if(f == NULL) { fprintf(stderr, "can't open %s\n", fname); exit(1); }
  fprintf(f, "P6\n%d %d\n255\n", WIDTH, HEIGHT);
  fwrite(buf, sizeof(buf), 1, f);
  fclose(f);
}
void img_putpixel(struct color c, int x, int y) {
  if(x < 0 || x >= WIDTH || y < 0 || y >= HEIGHT) { return; }
  buf[HEIGHT-y-1][x][0] = c.r;
  buf[HEIGHT-y-1][x][1] = c.g;
  buf[HEIGHT-y-1][x][2] = c.b;
}

void img_fillcircle(struct color c, double x, double y, double r) {
```

```
  int imin = (int)(x - r - 1), imax = (int)(x + r + 1);
  int jmin = (int)(y - r - 1), jmax = (int)(y + r + 1);
  int i, j;
  for(j = jmin; j <= jmax; ++j) {
    for(i = imin; i <= imax; ++i) {
      if((x-i)*(x-i) + (y-j)*(y-j) <= r*r) { img_putpixel(c, i, j); }
    }
  }
}
```

putpixel は指定したピクセル位置にレコードから RGB 値をコピーしますが、ただし画像の上の方が Y 軸で正の向きにしたいので、0 が画像の一番下の行になるように引き算を使っています。

fillcircle は画像上の円が含まれる範囲をまず整数で計算し、その範囲すべての点について、円の中に入っていたら putpixel で点を打ちます。

15.3.3 動画を作り出す

動画の原理は「少しずつ違う画像を次々に表示すると動いて見える」ということはご存じですよね。では、動画を作り出してみましょう。非常に簡単なプログラムです。まず色を 2 つ用意し、1 番目の色で 20 フレームぶん、だんだん位置を横に動かしながら円を描きます。そのあとさらに 20 フレームぶん、こんどは 2 番目の色でだんだん上の位置に動きながら、半径が小さくなるように円を動かします (図 15.3)。

```
// animate1 --- create animation using img lib.
#include "img.h"

int main(void) {
  struct color c1 = { 30, 255, 0 };
  struct color c2 = { 255, 0, 0 };
  int i;
  for(i = 0; i < 20; ++i) {
    img_clear(); img_fillcircle(c1, 20+i*8, 100, 20); img_write();
  }
  for(i = 0; i < 20; ++i) {
    img_clear(); img_fillcircle(c2, 180, 100+i*5, 20-i); img_write();
  }
}
```

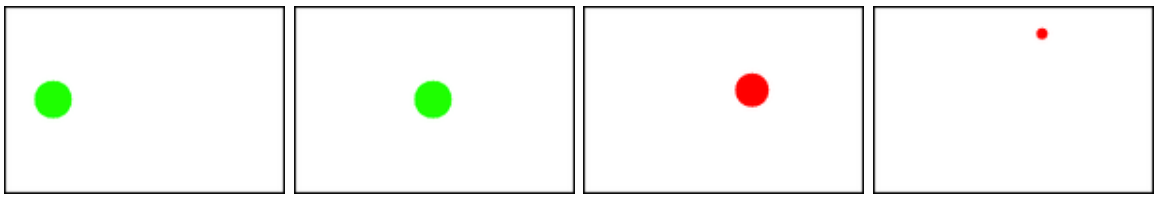


図 15.3: 動画のコマの例

実際にこれを動いて見えるようにするためには、**アニメーション GIF**(animation GIF) 形式に変換してください。次のようにすればよいのです。

```
% gcc animate1.c img.c -std=c99  ←普通にコンパイル
```

```
% ./a.out                                    ←実行
% animate img*.ppm          ←アニメーション表示
% convert -delay 5 img*.ppm out.gif  ←アニメ GIF に変換
```

convert で生成した GIF ファイルはブラウザで開いてください (普通のブラウザにはアニメーション GIF 再生機能がついています)。animate というコマンドで直接アニメーション表示もできます。テスト中はこちらが便利ですが、後で色々な人に見せるときはアニメーション GIF の方がよいでしょう。

15.3.4 課題のためのヒント

課題は動画を生成するプログラムなので、もちろん上の例題を利用して頂いて構いません (独自に設計したければそうされても構いません)。上の例題を利用するとして、「整った構造」についてはどのように考えたらいいでしょうか。

まず、例題ではほとんど円しか描けないので、ほかの図形を追加したいですね。もちろん main の方で直接点を打って図形を描くことはできますが、プログラムを複数の部分にきれいに分けるという点では、img.c の中に三角形とか長方形などを描く関数を追加する方が整っていると思われます。

さらに、複雑なシーンを持つ絵であれば単純な図形ではなく「家」とか「車」とか図形の組み合わさった形が現れると思われます。このとき、毎回 main から三角形や長方形や円の関数を呼び出すより、「家」や「車」という関数があってそれを呼び出す方がよさそうですね。

その「家」「車」という関数はどこに入れるのがいいでしょうか。main と一緒に入れるとか、img.c に入れてしまうとか、別のファイル parts.c を作ってそこに入れるとか、複数の選択肢があると思います (これはどれが正解ということはないですが、どのようにするかはきちんと考えて決めてレポートに書いて頂きたいです)。

次に動画なのでどのように動かすかも問題です。例題では直接フレームという単位でフレームごとにどれだけ動かす/小さくするなどを扱っていましたが、複雑な絵や複雑な動きだとごちゃごちゃになります。

そもそも動くということは、時間とともに位置や大きさが変化するということですから、時間に関する関数 $(x, y) = f(t)$ のようなものを定義して使うと「整って」いるかも知れません (t は秒数で与えたいですが、たとえば 20 フレームで 1 秒とか適当に決めればいいと思われます)。

または、複数の場面から構成される演劇のような動画も考えると、「この円は時刻 $t1$ から $t2$ の間存在し、その間に $(x1, y1)$ から $(x2, y2)$ までなめらかに移動し、かつ大きさは $r1$ から $r2$ に変化する」のような指定ができることが望ましいかも知れません。そうすると、そのようなものは当然 main で直接やることではなく、また img.c でやることでもなく、その中間にある anim.c のようなファイルが img.c を呼び出しつつ「動く円」「動く長方形」など必要なものを提供し、main はこれを利用して動画を組み立てて行く、というふうになるかも知れません。

なお、これらはすべてヒントなので、どのくらい何を工夫するかはそれぞれのチームで相談して決めてください。元の例題の構造のままで整っているということでもいっこうに構いません。

また、さまざまなファイルの分け方の話もしましたが、課題をチームでやる場合には、ファイルごとに担当するとか、1 人が設計をして他方が作るとか、誰かは記録だけするとか、これもそれぞれのチームにお任せします。ただし、何も仕事をしない人が出ることは避けてください。

報告課題 15A

今回は総合実習のため当日は「報告課題」(時間中にやったことの報告) です (プログラムの提出は不要)。簡単にまとめてください。

Q1. どのような分担で課題プログラムを構成する計画ですか。

Q2. 複数で分担して 1 つのプログラムを作成することをどう思いますか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

総合実習課題 15B

B 課題は必ず 2〜3 名のグループで実施し、かつ、ペアプログラミングではなく「各自が自分の担当を書く」形にしてください (どうしてもメンバーが見つからない時は担当教員に相談のこと)。課題は次のものです。

課題 Y　「動画を生成するプログラム」で「整った構造を持つ」プログラムを開発しなさい。

「整った構造」の定義は各自にお任せします (自分たちのレベルに合った内容でよい)。レポートを重視するので、どのように整っているかをしっかり書いてください。プログラムは当然グループ内で同一となりますが、レポートは各自でお願いします。レポートは次の順で記述してください。

0. 表紙 — 学籍番号＋氏名、グループメンバーの学籍番号＋氏名 (1〜2 名)、提出日付。
1. 構想・計画・設計 — どのような構想でプログラムを企画したか、プログラムはどのように設計したか。
2. プログラムコード — 必ず動作するものを提出してください。
3. プログラムの説明 — プログラムのどの部分が何をしているかの説明をお願いします。
4. 生成された動画 — アップロードで提出してください。プログラムコードと動画が一致していること。レポートにはどのような動画という説明を書いてください。
5. 開発過程の説明 — 誰が何を分担し、どのような過程を経てプログラムが完成したか。各作業の日時と担当者の記録があるとよい。
6. 考察 — 課題をやって分かったことや感想など。
7. 以下のアンケートの解答。

    Q1. うまく分担して課題プログラムを開発できましたか。
    Q2. 複数で分担する際に注意すべきことは何だと思いましたか。
    Q3. ここまでの科目全体を通して、学べたこと、学びたかったけど学べなかったことは何ですか。その他感想や、この科目の今後改善した方がよいこと、今後も維持したことがよいことの指摘もどうぞ。

生成する動画についてはクレジットつきでネットや会合等で紹介することがありますので、公序良俗に反する (ネット等に掲示できない) 動画を生成することはやめてください。

索 引

基礎プログラミングおよび演習 2018