2007 年度 北海道大学 工学部 情報エレクトロニクス学科

計算機プログラミングI・同演習

2007年度 講義ノート

井上 純一

北海道大学 大学院情報科学研究科 複合情報学専攻

URL: http://chaosweb.complex.eng.hokudai.ac.jp/~j_inoue/index.html

目 次

第1回講義・演習 — 平成19年4月6日 —

1.1 はじめに — この講義ではいったい何を学ぶのか? —

この講義では皆さんにおなじみの Windows Vista/XP/2000 とは少し使い勝手が異なる基本ソフト (オペレーティング・システム, OS) である **Linux (リナックス) [RedHat9]** の操作法とこの基本ソフト上で動作するプログラミングソフト **gcc** を用いた **C 言語**による計算機プログラミングの基礎を学びます. ここで学ぶことはプログラミング初心者はもちろんのこと, プログラミングにある程度慣れている者も 2 年次後期の「計算機プログラミング II」, 3 年次以降の実験, 研究室に配属してからの研究においても必要となる最小限の知識/技術であるので, この機会に必ずマスターしておくことが大切です.

また, この講義では演習課題の提出/採点を情報基盤センター赤間清先生が開発した学習管理システム (LMS: Learning Manegement System) を用いて行います. 従って, 従来のような**レポート用紙に記入した上での課題の提出は不要になります**. 後に詳しく説明がありますが, この LMS の機能としては

- 出席と進度の管理
- 問題提示と課題レポートの投稿
- レポート提出状況の確認
- アナウンス・掲示板機能

等が挙げられます. このシステムのアカウントは後に配布します. なお, インターネットに接続できる環境にあるならば時間外でもアクセスでき, レポート課題を提出できるなど, 従来の課題提出方法に比べてかなり便利になるものと予想されます.

1.1.1 教科書について

この講義では C 言語による計算機プログラミングに関する標準的な教科書として下記を用います. 各自が生協書籍部で購入し, この教科書をもって講義/演習に臨んでください.

「C によるプログラミング演習」岡田 稔 著, 近代科学社 (1993) 定価 2,300 円 (税抜)

なお, 演習では上記教科書から類似問題が出される可能性があることに注意してください. 講義・演習の時間は限られているため, 当然, 上記の教科書に載っている全ての事項をフォローすることはできません.

一方, 本講義では補足事項や演習課題を説明するための資料 (お手元にあるこの資料のようなもの) も各回に配布します. 配布資料で説明される事項に関しては全て講義・演習中に確認します. 従って, 学習の進め方としては, **配布資料をベースに資料中の事項は各回のうちに確実にマスターし, 教科書は適時辞書的に使い, C 言語についての知識を増やして行く**というやり方が現実的かつ適切だと思います.

これらのプリントおよび講義で用いたスライドなどは下記 URL からダウンロード可能です.

http://chaosweb.complex.eng.hokudai.ac.jp/~j_inoue/PROG2007/PROG2007.html

講義に欠席した場合は上記サイトからダウンロードしてください (※ 翌週に前回の資料を受け取りに来てももらえない可能性が高いと思っておいてください).

なお, 昨年度はポインタまでの内容を学習しましたが, 今年度よりポインタ/構造体は後期開講の**「計算機プログラミング II」**の内容となりました. 従って, この講義・演習ではより C 言語らしいプログラミング技法に関しては学習しないのですが, これは逆に言えば, この講義・演習で学ぶ事項は C 言語にかかわらず, 他の言語にも共通するものであるわけなので, プログラミング初心者も経験者もここでの内容を確実にマスターしておいてください.

1.1.2 講義/演習の進め方

各回はじめに, その回の演習で必要となる事項, Linux OS, 計算機システムに関する概念, C 言語プログラミングに関する文法等の全般的説明を **M151 講義室**にて行い, 後に**計算機室 (M152)** に移り, 皆さんに端末の前で実習を行ってもらいます. 実習 (演習) ではプリントに書かれた課題を考え, 実際にプログラミングして頂きます. 実習中には TA(ティーチング・アシスタント) の院生, 教員が皆さんのまわりを巡回しますので, わからない点は遠慮なく尋ねること. 各回に学ぶ事項はその日のうちにマスターしていくことが肝心です.

1.1.3 成績について

まず, **計算機プログラミング I と計算機プログラミング演習 I では同じ成績がつく**ことに注意してください. また, 各回の理解度を確かめるため, レポート課題を考えてもらい, 解答を LMS からの入力により提出してもらいます (※ LMS の使用方法は後ほど説明します). また, 期末試験を行い, 最終成績は LMS からのレポート提出, 期末試験の成績を考慮し, 総合的に判定されます.

※ とても重要なことについて (必読のこと)

この計算機プログラミング I・同演習の単位を落とした場合, 3 年次以降に再履修をしてもらうことになりますが, その際, 3 年次カリキュラムの編成上, **皆さんの 3 年次の履修申告上の時間割の中に, この計算機プログラミング I・同演習の講義を再履修科目として組み込むことがとても難しくなります. (実質的にほぼ不可能と思います. それだけ 3 年次に履修しなければならない科目が多いということです.)** 従って, この科目を 4 年次以降に再履修して頂くことになる可能性が高く, その結果として, この計算機プログラミングの単位を落とすことが, 皆さんの卒業・進級に対し, 極めて高いリスクを持つことになってしまいます. ですから, 賢明な皆さんは**「数ある科目の中の一つ」程度に考えるのではなく, この単位を落とした場合, 留年に直結する可能性が高いことを肝に銘じて当科目を受講してください**.

1.2　計算機室システムのアカウントとシステムへのログイン

計算機室のコンピュータシステムを使用するための手続きを以下で説明して行きます.

(1) まず, 計算機室の入り口に掲示されている名簿の中から自分の名前を探し, 自分の ID 番号を確認します. ※ 皆さんは金曜日の A グループであることに注意してください.
(2) 指定された端末の前に着席し, ログイン画面から自分の ID 番号を入力し, システムに入ります.
(3) 無事にログインに成功し, 計算機システムに入ると図 1.1 のような画面が現れることを確認してください (ただし, 背景の壁紙はデフォルトだと無地のものになっているはずです). 図 1.1 の黒矢印で示され

図 1.1: ログインしてみると, このような画面が現れる. 背景 (壁紙) はデフォルトでは無地. 矢印の部分をクリックして Netscape ブラウザを立ち上げる.

た「地球儀」の部分をクリックし, Netscape ブラウザを立ち上げる.
(4) 計算機室のホームページを立ち上げ, **スルー PASS 4.2** の画面を表示させる.
(5) (4) でパスワードの変更が済んだら, Netscape を終了し, 図 1.1 の右隅の赤い帽子 (RedHat) の部分をマウスでクリックし, コンピュータからログアウトし, 再度ログインし直してみましょう. **password** の部分には先ほどスルー PASS4.2 を用いて設定したパスワードを用いること.

以上がスルー PASS4.2 を用いたパスワード設定の流れです.

1.2.1　パスワードに関する注意事項

以下にパスワードの取り扱いに関して注意すべきことを列記しておきましょう.

- パスワードは他人に教えない (例え親しい友人に対しても, である).
- パスワードの変更はまめに行う (少なくとも数週間に一度は変更するように). このときのパスワードの変更の仕方は上で説明したスルー PASS4.2 を用いたパスワードの設定と同じ手順で行えます.

図 1.2: Netscape ブラウザで計算機室のホームページを立ち上げ, 矢印の部分をクリックして, パスワード変更ツール : スルー PASS4.2 の画面に入る.

- 他人に容易にわかってしまうようなパスワードは用いない (生年月日を含むもの, 著名人などの名前などは全くダメな例). 「いくつかの単語の一部分を間に数字を挟みながら並べ, かつ, それを逆から読む」等, 可能な限りデタラメに近い, 自分が読み直してもデタラメに思えるようなパスワードを作る. ただし, この場合には自分もすぐには覚えられないかもしれないので, 当初は小さな紙に書いておき, 覚えた段階でその紙を破棄するなどすると良いと思う.

- パスワードを忘れてしまった場合には早急に教員に申し出ること. 特に危ないのが, ゴールデンウィーク明けあたりである.

この講義で用いる計算機のシステムが皆さんの普段用いているであろう PC と違うのは, 基本ソフトが Windows Vista/XP/2000 の代わりに Linux であるというだけではなく, この講義で用いるシステムでは **CPU(計算機の演算装置) やハードディスクをユーザ全員で共有する**ということです. 従って, 後に詳しく見ていきますが, 皆さんの作成したプログラムや電子メールのテキストファイル等は設定によっては全てのユーザが読め, 書き直すことが可能となってしまいます. このようなシステムでは個人のパスワードの管理のまずさが同じシステムを用いている全てのユーザに降りかかってくることがありえます (例えば, 本演習に関係のない第 3 者がそのパスワードを悪用することにより, システムに悪意をもってダメージを与えることも起こりうる). 自分の PC 以上にパスワードに関しては慎重に取り扱ってください. また, 同様の理由で, ネットワークから大量の大容量のファイルをダウンロードすることはネットワークに負荷を与えるだけではなく, 全てのユーザが用いるハードディスクの容量を必要以上に占有してしまうことになりかねません. 従って, 本システムでは自分の行為が他人にも何らかの影響を与えてしまうという意識を常に持ちながら演習を行ってください.

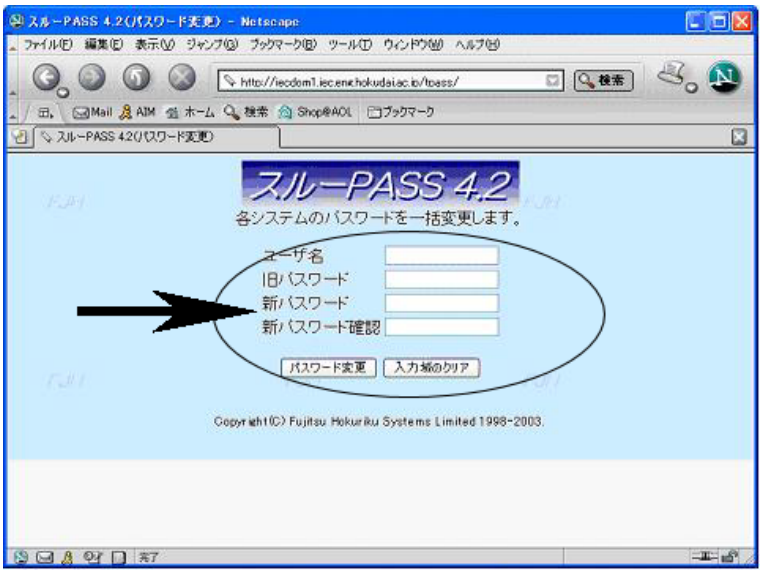

図 1.3: Netscape ブラウザでスルー PASS 4.2 のページ. ここから自分のパスワードを設定/変更しよう.

1.3　ターミナルウインドウの起動

この演習では計算機上での多くの操作は Windows 上でのマウスによる「視覚的」なものではなく, 計算機への命令をキーボードから文字列として入力することにより行うことになります. そのような命令を与える場所として**ターミナルウインドウ**を用います. このターミナルウインドウを起動するには, 図 1.1 の画面左隅にある「赤帽 (RedHat)」をマウスでクリックし, 「システムツール」「ターミナルセッション」の順で開き, さらにその中の「シェル」というところをクリックしてみます. すると, 図 1.4(左) に示すよう

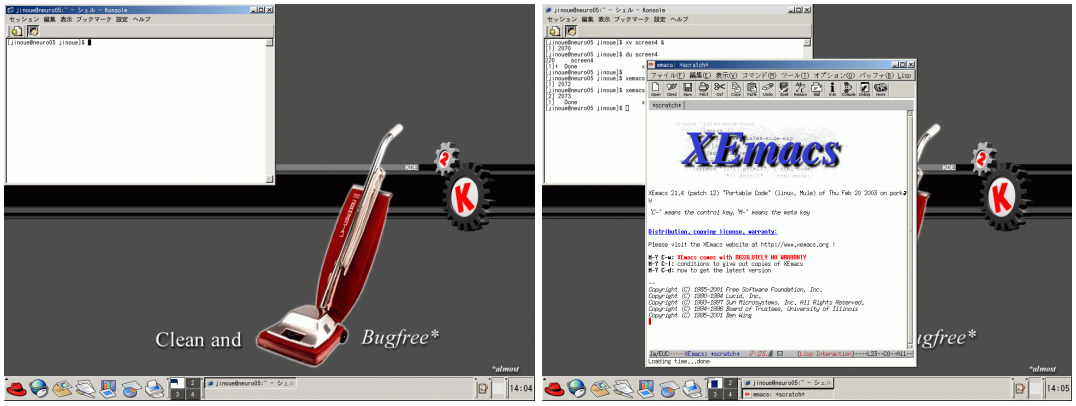

図 1.4: 起動したターミナルウインドウ (左) とターミナルウインドウから起動した xemacs(右).

な新しい「ウインドウ」が立ち上がります. 本演習で行う作業のほとんどはこのウインドウから計算機への命令 (コマンド) を文字列でキーボードより入力することで行われます.

なお, 一度開いたウインドウを終了するためには, このウインドウの中で

```
exit      [Enter]
```

と行います[1] ．ターミナルウインドウは複数 (原理的にはいくつでも) 同時に起動することができますが, ターミナルウインドウを複数起動すればするほどシステムに負荷をかけ, 従って, その時に自分が行っている他の作業が遅くなったりすることもあるので, せいぜい 3 つくらいにとどめておくことが望ましいです.

1.4 Xemacs によるテキストファイルの編集 : 必要最小限

この講義ではプログラミングをしてもらうわけですが, そのとき, プログラム自体を書き, テキスト形式のファイルとして保存する場合には **Xemacs** と呼ばれる編集ソフトを用います. このソフトを起動するには前節で説明したターミナルウインドウ内で

```
xemacs & [Enter]
```

とします. すると図 1.4(右) のように Xemacs が起動します. この際, 初期設定としていくつかのメッセージが出ますが, その際には全て **Yes** と答えて先に進んでください.

この Xemacs を終了するには [Ctrl-X] [Ctrl-C] ([Ctrl] を押しながら X と C を順番に押す) と押します. すると保存する名前を尋ねてきますので, そこでは **test.txt** と打ち込んで [**Enter**] を押してください. Xemacs の便利な使い方は回を追うごとに見ていくことになります.

1.4.1 日本語の入力方法

日本語を入力するには [Shift][Space] としますと入力できます. ローマ字入力に戻したい場合には再度, [Shift][Space] とします.

1.5 電子メールの設定と使い方

この講義では netscape を用いてメールを読み書きしてもらいます. その設定は図 1.5 のように netecape を立ち上げてから, 赤○で囲った “Mail” と書かれたアイコンをクリックします. 初回にこのメイラ (メー

図 1.5: メールを使用するには図のように netscape ブラウザの○で囲まれた部分にある “Mail” アイコンをクリックする.

[1] 今後, いくつかのアプリケーション・ソフトの使用方法を学びますが, その際には必ず, 実際にソフトを使う前に, その終了方法を確認しておくこと. 当たり前に思われるかもしれませんが, 重要なことです.

ルソフト) を起動すると, メールの送受信に必要な設定を行うためのダイアログが開きますので, 設定を以下の順序で行ってください.

(1) 「電子メールアカウント」を選択して「次へ」をクリック.
(2) 名前欄に自分の名前を, 電子メールアドレスはそのまま (ユーザ名) にして「次へ」.
(3) 受信のサーバタイプは「POP」を選択し, 受信サーバの名前は「mailsrv.iec.eng.hokudai.ac.jp」と入力する. 送信サーバの名前は「mlvirus.iec.eng.hokudai.ac.jp」と入力して「次へ」.
(4) ユーザ名に自分のアカウントを入力して「次へ」.
(5) アカウント名を自由に入力して「次へ」.
(6) 入力した情報に間違いが無ければ「完了」をクリック.

新着メールを受信するには受信ボタンをクリック. メールを送信するには「作成」ボタンをクリックし, 「宛先」「件名」「本文」を所定の箇所に記入し, 「送信」ボタンをクリックする.

※ 各ユーザのメールアカウントは

`XXXXXXX@iec.eng.hokudai.ac.jp`

で与えられます (XXXXXXX は自分のユーザアカウント). 確認のため自分自身のメールアドレスにメールを送信し, 無事に受信できるかを各自確認すること.

※ 重要なこと
当講義で取得したメールアドレスからの電子メール送受信は当講義に関するものだけにしてください. 私用で使わないこと.

1.6 Learninig Manegement System の使い方 — まずは簡単に —

ここでは, 皆さんに課題を Web 投稿してもらうためのシステムである LMS (Learning Manegement System) に関し, 現段階で必要最小限の事項を確認してもらうことにします. 他のアプリケーション・ソフトと同様, より細かな機能に関しては追って見ていくことにします.

まずは皆さんに使ってもらう LMS のホームページアドレス

`http://133.87.128.133/compro/index.html`

を先ほど使い方を確認した netscape ブラウザを用いて表示すると図 1.6 のような待ちうけ画面が現れます. このとき, 皆さんは「A グループ」ですから「Group A」の四角で囲まれた部分の右下にある「矢印」をクリックしてみると, [**お知らせ**], [**次回の講義**], [**提出課題**] 等の各情報を得ることができます. 特に期末試験/中間試験, 休講情報などの重要情報はこのページから発信する場合が多いですから, 日頃から頻繁に上記 URL を訪れて, この情報をこまめにチェックしてください.

さて, まずは講義中の指示に従って, LMS のアカウントを作成します (※ これは計算機システムのアカウントとは別物であることに注意). 無事にアカウントが生成され, 自分のパスワードも決めたならば早速, このページ左上にある**ユーザ ID**, **パスワード**に各自の情報を正確に打ち込み, LMS にログインしてみましょう.
無事にログインできると, 図 1.7 のような画面が現れます. このとき, 左の一番上に例えば「井上 純一 ○○」のように自分の名前が現れることを確認してください.

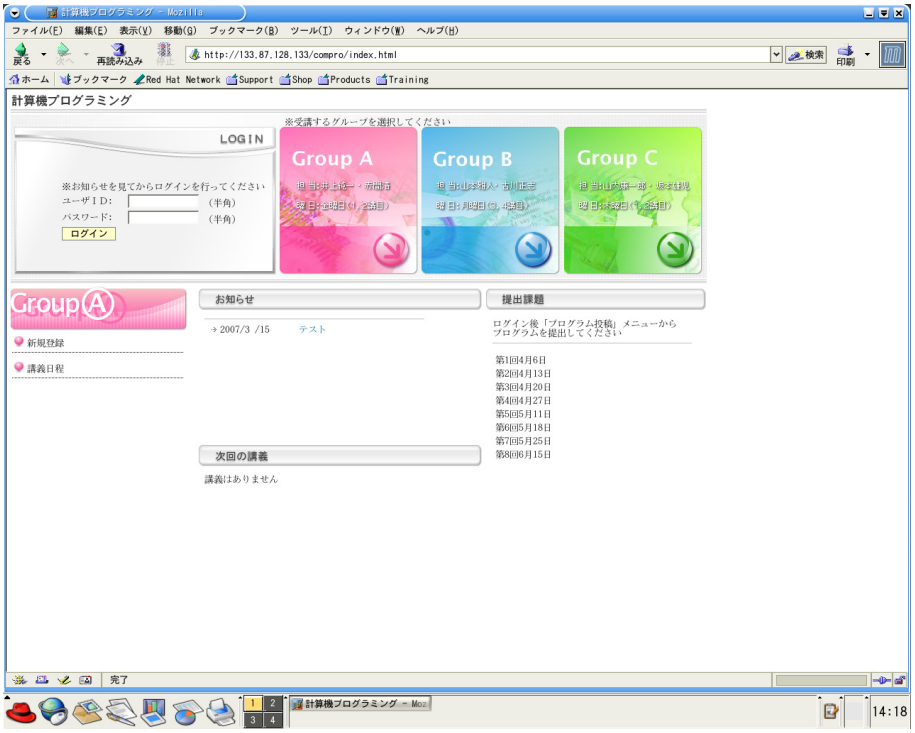

図 1.6: LMS の待ちうけ画面. 左上の**ユーザ ID**, **パスワード**を正確に打ち込み, LMS にログインしてみよう.

また, 左端のメニューには個人設定として [**パスワードの変更**], [**ノート**], また, 利用可能なテキスト類として [**ルーズリーフの表示**], [**プログラム例題の表示**], また, 課題として, [**課題の投稿**], [**投稿状況の確認**], **クラスの投稿状況**], 掲示板として [**掲示板**], また, ログアウトのために [**ログアウト**] なる項目が並んでいることを確認してください.

1.6.1　プログラムの投稿

ここではプログラムの投稿方法について簡単に確認しておきます. まず, 左端のメニューから [**課題の投稿**] を選ぶと [**投稿可能な問題シリーズ一覧**] が出ますので, この場合に利用可能なものを [**表示**] させます. すると問題リストが週別に表示されますので, 該当する問題を選びます. このとき, 図 1.7 のように選択された問題文が表示されますので, この問題画面の右上にある [**投稿**] をクリックして次に進みましょう.
すると, 図 1.8 のような画面が現れますので, ここの [**ハンドルネーム**] に自分の好きな名前を書いてプログラムを記入していきます. 必要ならば [**一言コメント**] にコメントを記入し (例えば, 込み入った質問など), [**投稿**] ボタンをクリックしてください. 投稿が正しく受理されると, [**投稿完了**] **書き込みは正常に終了しました**というメッセージが表示されるはずです.

1.6.2　投稿状況の確認

この LMS では自分の投稿状況が逐次確認できます. ログイン画面の左端のメニューの中から [**投稿状況の確認**] を選択し, その部分をクリックします. すると, 図 1.9 のような画面が現れます. ここで, 各問題の投稿状況, 採点結果などが確認できます. 採点結果が×になっているものは問題の要求する出力を返さない

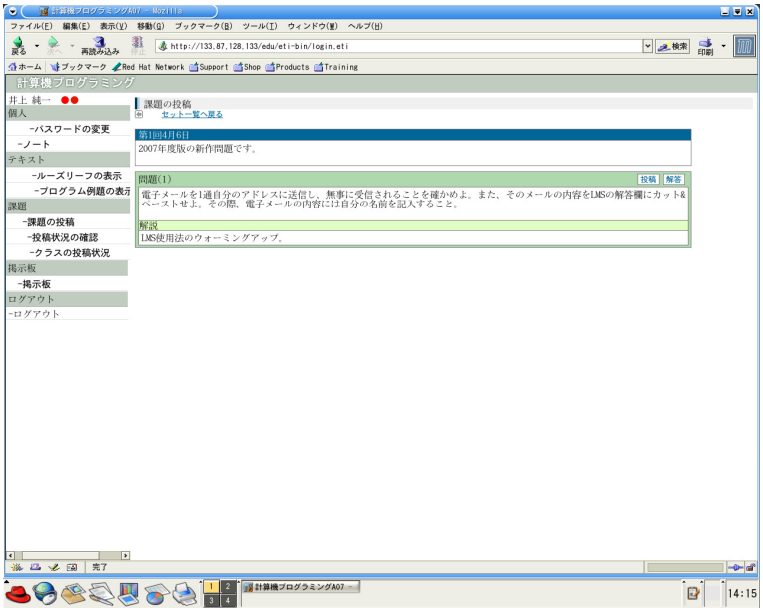

図 1.7: ログインした後の画面.

プログラムだということです．再投稿は何回でも可能なので，○になるまで繰り返しトライしてください[2]

1.6.3　ログアウト

作業を終え，LMS から抜ける場合には [**ログアウト**] をクリックすることを忘れないように．また，パスワードはこまめに変更してください．

今週の LMS 入力課題

電子メールを 1 通，自分のメールアドレスに送信し，無事に受信されることを確かめよ．また，そのメールの内容を LMS の解答欄にカット & ペーストせよ．その際，電子メールの内容には自分の名前を記入すること．

これで計算機システムを使う際の基本的な設定は終わりです．次回からいよいよプログラミングを行っていきます．

[2] 基本的に入力期限はありませんが，出題以降，2 週間以内を目安としてください．

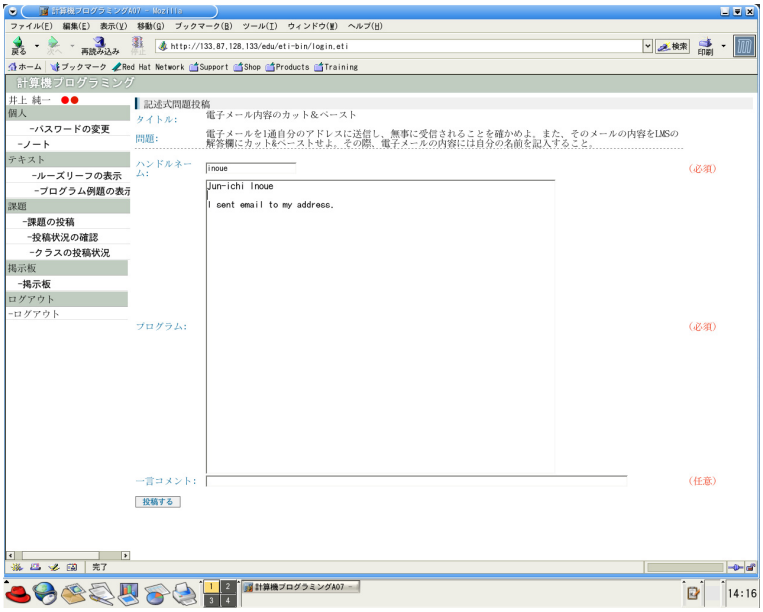

図 1.8: 投稿画面.

図 1.9: 投稿確認画面.

第2回講義・演習 — 平成19年4月13日 —

2.1 ファイルシステムとディレクトリ構造

今回は前回の講義で皆さんがアカウントを取得し, これから半年間使っていくことになる計算機室の Linux システムにおける [ファイル] と [ディレクトリ] の概念を詳しく学んで行きます. また, これらを操作するコマンド (命令) を端末からキーボード入力してもらうことで, 計算機がそれらコマンドに対してどのような応答をするのかを実際に確認してもらいます. ある程度コンピュータに慣れた人であれば, この配布資料を読むだけでも何となくわかった気になりますが, はじめは面倒がらず, 必ず資料に書かれたことを逐次入力し, その動作を自分で確かめながら進んでください. 途中, 頻繁に出てくる **練習問題 *.*** は, そこまでに学んだ事項を実際に手を動かしながら理解してもらうために設けたものあり, 学習支援システム LMS (Learninig Manegement System) へ入力して頂く課題 (「※ LMS 入力課題」と断り書きが書いてあります) とは別に作ってあります. また, 各配布資料の最後にあげられている **今週の LMS 入力課題** は, 各回の知識を使ってある程度まとまった作業を行ってもらうための問題であり, 翌週に詳しい解説を行う可能性の高い問題です. 学習ペースと理解度は, 各回, この問題が解けるかどうかを確認することでつかめるようになっています.

2.1.1 ファイルとその操作

まずはプログラミングにおいて恒常的に必要となるファイルの作成, 表示, コピー (複製), そしてファイル名の変更等について順を追って見ていきましょう.

- **ファイルの作成**
  前回の講義で少し見たように, (テキスト) ファイルを作成するにはテキストエディタ Xemacs を起動させ, データを書き込んだ後に Ctrl-x Ctrl-s を行い, エコー領域に自分の好きな名前を書き込めば良かったわけです. しかし, こうしなくとも Xemacs を起動する段階で自分が付けたい名前を指定して

  ```
  xemacs test.c &
  ```

  とすればよく[3] , こうして開かれた Xemacs の画面の中に文字を書き込み, その後に Ctrl-x Ctrl-s とすれば, 自動的に test.c という名前でキーボードから入力した内容がファイル test.c に保存されることになります. ここで, 入力行の最後につけた & マークの意味は後に詳しく説明しますので, ここでは特に気にしないでください.

  これを確認するために次の練習問題を実際にやってみましょう.

  練習問題 6.1
  test.c という名前で Xemacs を起動し, 下記の文章を打ち込んだ後に保存せよ.

  ```
  #include<stdio.h>
  main()
  ```

[3] いちいち書きませんが, 各コマンドを打ち込んだ後には [Enter] を押すこと

```
{
    printf("Today is 13th April 2007 and Friday! ");
}
```

※ このプログラムの意味については後ほど詳しく見ていくことになります.

- **ファイルの表示**

ファイルの内容を確認するためわざわざテキストエディタ Xemacs を起動させなくても, **less** というコマンドによって, 現在使用しているターミナルにファイルの内容を表示させることもできます. 具体的には

```
less [ファイル名]
```

とすれば指定した [ファイル名] の内容がターミナル上に表示されることになります. less を終了させるには q とすれば OK です. 既に述べたように, この less はファイルの内容を表示させるためだけのコマンドであり, 編集機能はありません. 従って, ファイルの内容を参照するだけでなく, 書き変えたい, 一部を削除したい場合には今までどおりに Xemacs を使ってください.

それではどのような場合に編集機能を持たない less を使うべきなのかが気になりますが, 例えば, システム管理などで, ファイルの一部を (何らかのアクシデントで) うかつに書き換えてしまうと, その後のシステム動作に悪影響を与えかねないような種類のファイル内容を一時的に参照したい場合, 編集機能を持つエディタよりも less が安全であり, 好ましいことになります.

練習問題 6.2

less コマンドを用いて test.c の内容を確認せよ.

- **ファイルのコピー**

ときとして同一の内容のファイルをもう一つ複製 (コピー) したい場合があります. このような場合には **cp** コマンド (**c**o**p**y の略) が有用です. 例えば, test.c と全く同じ内容のファイル test2.c を作りたいのであれば

```
cp test.c test2.c
```

とすれば OK です. つまり, より一般的に書けば

```
cp [コピー元のファイル名] [コピー先のファイル名]
```

となります. 実際に上記のコマンドでファイルが複製されることを次の練習問題で確かめてください.

練習問題 6.3

cp コマンドを用いて, test.c の複製を作り, その内容を less コマンドで確認せよ.

- **ファイル名の変更**

ファイル名を変えたい場合には mv コマンド (**m**o**v**e の略) が便利です. 例えば, test2.c というファイル名を test3.c に変更したいのであれば

```
mv test2.c test3.c
```

とすれば OK です. 一般的に書けば

```
mv [変更前のファイル名] [変更後のファイル名]
```

となります. これを次の練習問題で確かめてください.

練習問題 6.4
mv コマンドにより, test2.c の名前を test3.c に変更せよ.

- **ファイルの削除**
  不要なファイルは, こまめに削除しておきたいものです. そのような場合には **rm** コマンド (**rm**ove の略) が有効です. 例えば, test3.c を削除したいような場合には

```
rm test3.c
```

とすれば OK です.

ところで, 当たり前ですが, 一端削除してしまったファイルは復活しません. しかし, 削除してしまったファイルが, それまでに何度か編集を繰り返し, その都度「上書き保存」を実行していたファイルであれば, backup (old) file (ヴァージョンの一つ古いファイル) として, 例えば, オリジナルファイルが text.txt というファイル名であれば, text.txt~ というファイル名で, かろうじて (保存する前の古いヴァージョンですが) ファイルが生き残っている場合があります. その際は, 前に学んだ mv コマンドを用いて, その名前 text.txt~ を text.txt へと変えてください. こうすれば最新のヴァージョンは復活できないにしても, 一つ古いヴァージョンを元に, 引き続き text.txt の編集作業を続けることができます[4] .

練習問題 6.5
rm コマンドで test3.c を削除せよ.

2.1.2 ディレクトリ構造

ディレクトリとは一言で言うと「データの記憶 (ファイルの保管), 参照や作業を行う場所」のことです. これは Linux のシステムでは次のような「木」の構造をとります. この図の最上部にある root は**ルートディレクトリ**と呼ばれます. 前回の講義では Xemacs に自分の名前を書き込んでもらい, それを name.txt という名前のファイルに保存してもらいました. このファイルはログインした時に自分がいるディレクトリにあるはずであり, **ホームディレクトリ**と呼ばれます. つまり, 各ユーザにはそれぞれ一つのホームディレクトリが割り当てられているわけです. また, 木構造の構成要素的な観点から言えば, (自分のホームディレクトリに何も新しいディレクトリを作っていない状況下では) 各個人のホームディレクトリはこの木構造の末端に位置します図 2.10 参照.

[4] この意味でも Xemacs でプログラムの編集作業をする際には, こまめにファイルを「上書き保存」すること. Xemacs での「上書き保存」のキー操作は Ctrl-x Ctrl-s です.

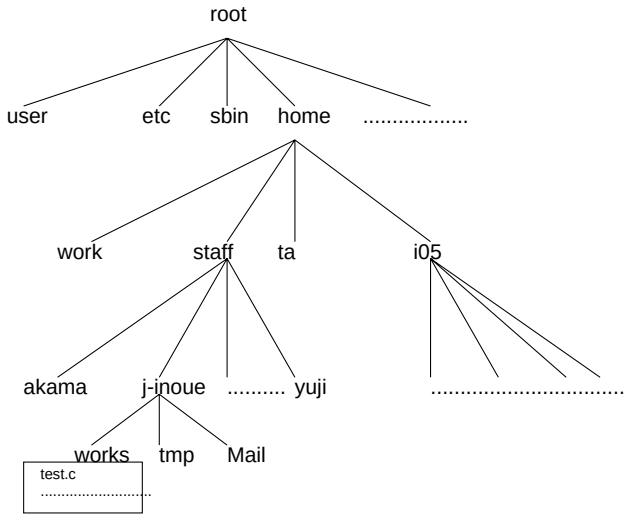

図 2.10: Linux のディレクトリ構造.

絶対パスと相対パス

さて, Linux システムのディレクトリ構造は上述のような木構造をとるわけですから, その性質上, **ルートディレクトリから任意のディレクトリへの経路は一意に定まります**. 従って, 一貫したルールを設ければ, 全てのディレクトリは一意に指定されることになるわけです. その規則とは, 例えば, 上図で私 (井上) のホームディレクトリ j-inoue を指定する場合には

```
/home/staff/j-inoue/
```

とすれば良いし, 私のホームディレクトリにある works というディレクトリを指定したいならば

```
/home/staff/j-inoue/works/
```

とすれば良いわけです. ここで, いずれの場合にも先頭の「/」はルートディレクトリを表す記号であると思ってください. このように, Linux の計算機システムでは任意のディレクトリは「/」から出発して「ディレクトリ名/」を順々に並べていけばよいことになります.

また, 任意のファイルも指定可能です. 例えば, 私のホームディレクトリの下にある works ディレクトリの内の test.c という名前のファイルを指定したいのであれば

```
/home/staff/j-inoue/works/test.c
```

とすればよいことになります. このような形でのディレクトリ/ファイル指定を**フルパス指定 (絶対パス指定)** と呼びます.

ところで, ここですぐに気になるのは, このような方法で全てのディレクトリやファイルを指定する場合, 末端のファイルが木構造の深部へ行けばいくほど長くなり, 非常に使いにくくなる点です. このような不便さを回避するために, 現在自分がいるディレクトリ (**カレントディレクトリ**と呼ぶ) を基準に, 目標とするディレクトリやファイルを指定する**相対パス指定**も利用できます. 例えば, 自分が現在, ディレクトリ j-inoue にいるとすれば, このカレントディレクトリから, ディレクトリ works 内のファイル test.c を指定する場合には

```
./works/test.c
```

とすれば良いわけです. ここで, 「.」はカレントディレクトリを表します. なお, これは省略が可能であり

```
works/test.c
```

としても OK です. また, ホームディレクトリの所有者がホームディレクトリ (j-inoue) からこのファイルを指定する場合には

```
~/works/test.c
```

とします. この「~/」は自分のホームディレクトリを表します.
以上のことを念頭に, 以下でディレクトリとファイルの操作法を具体的に見ていくことにしましょう.

ディレクトリとファイルの操作法

ここからはディレクトリ, ファイルを操作する際のコマンドのいくつかを見ていきます. これらはいずれも比較的使用頻度の高いものなので, この場で覚えて, 今すぐにでも使えるようにしておきましょう.

- **カレントディレクトリに存在するファイルやディレクトリの参照法**
  既に前回使っていますが

  ```
  ls
  ```

  とすれば, カレントディレクトリに存在するファイルの一覧を参照することができます. (**list** の略).

- **カレントディレクトリのフルパス指定の参照**
  現在自分のいるディレクトリのフルパス指定はコマンド pwd (**p**resent **w**orking **d**irectory の略) で参照できます.

  ```
  pwd
  ```

  実際に次の練習問題で動作を確かめてください.

  練習問題 6.6
  カレントディレクトリで pwd コマンドを実行し, 画面への出力結果を確認せよ.

- **ディレクトリの作成**
  name.txt の他にも多数のファイルを作成した場合 (例えば, test.c, tmp.dat etc.), それらのファイルの中で互いに関連するものどうしをまとめて保存した方がよい場合があります. こうしたときにはホームディレクトリ, あるいは, カレントディレクトリに新たなディレクトリを作成し, そこに互いに関連するファイルを保管することになります. この方法を以下で説明することにしましょう.

  例えば, カレントディレクトリに tmp という名前のディレクトリを作成したいのであれば

  ```
  mkdir tmp
  ```

  とすれば OK です. (ここで mkdir は **ma**k**e** **dir**ectory の略). このとき, ls コマンドを用いて確認すれば, カレントディレクトリに新しいディレクトリ tmp が存在するのがわかります.

- **ディレクトリの削除**
  ファイルの場合と同様に, 不要になったディレクトリは削除し, 親ディレクトリを整理したくなります. そのときには rmdir (**re**m**o**ve **dir**ectory の略) コマンドが使えます. 実際に上で作った tmp ディレクトリを削除したいのであれば

```
rmdir tmp
```

  とすれば OK です. ここで注意しなければならないことは, **このコマンドが機能するためには, 削除対象のディレクトリ内にファイルが存在せず, 空の状態であることが必要になる**という点です.

- **ディレクトリへのファイルの移動**
  これで新規ディレクトリが作成できましたから, このファイルを格納する「箱」の中にファイルを入れることを考えます. つまり, 新規のディレクトリの中に既存のファイルを入れることを考えましょう. このときには, 「ファイル名の更新」のところで既にみた, mv コマンドが役に立ちます. カレントディレクトリにある test.c をカレントディレクトリの下に作成したディレクトリ tmp に移動したいのであれば

```
mv test.c tmp/
```

  とすればよいことになります. この最後の「/」は tmp がディレクトリであることを明示するためであり, 付けなくても機能します. 次の練習問題をやってみてください.

練習問題 6.7
自分のホームディレクトリに tmp という名前のディレクトリを新たに作成し, そこに test.c, name.txt の 2 つのファイルを移動せよ.

- **ディレクトリ間の移動**
  複数のディレクトリ間を行ったり来たりするには cd (**c**hange **d**irctory の略) を用います. この場合, 行き先のディレクトリの指定に関しては前出の「絶対パス指定」「相対パス指定」の双方が可能です. 例えば, 前出の図 1.1 にある私 (井上) のホームディレクトリ j-inoue に行きたい場合には, 絶対パス指定で

```
cd /home/staff/j-inoue/
```

  とすれば良いし, 現在自分のホームディレクトリにいる私がディレクトリ staff の直下にある赤間先生のディレクトリ akama に行きたい場合には相対パス指定で

```
cd ../akama
```

  とすればよいことになります. ここで, 「..」は「/ディレクトリ名」の一つ手前のディレクトリを意味します. これを「/ディレクトリ名」の**親ディレクトリ**と呼びます. 具体例を用いてこれを確認するならば, 例えば, ディレクトリ j-inoue で

```
cd works
```

とした後に

```
cd ../
```

と行うと, この時点でカレントディレクトリは j-inoue ということになります. つまり, 「j-inoue は works にとっての親ディレクトリ」ということになります.

ところで上出の赤間先生のホームディレクトリに絶対パス指定で行きたい場合にはもちろん

```
cd /home/staff/akama/
```

とすればよいことは, もうおわかりでしょう. 次の練習問題をやってみてください.

練習問題 6.8
私のホームディレクトリ (j-inoue) の下の tmp というディレクトリに check.txt というファイルを置いてある. これを各自が自分のホームディレクトリに作った新規ディレクトリ tmp にコピーし, このファイルを Xemacs で開くことにより, このファイルに書かれた内容を確認せよ.

2.1.3 ファイルへのアクセス権限とその変更

上の **練習問題 6.8** をやってみた段階で次のような疑問を持たれた方もいるでしょう. すなわち, 「他人のディレクトリにある任意のファイルがコピーできて, それが読めるということになると, プライバシーの保護, 個人情報の取り扱いという観点から言って, Linux というシステムは好ましくないのではないか ?」と. これに関しては, 各ファイルの所有権, アクセス権限を変更することで, 他人には見せたくないファイルのアクセス・モードを読み込み (書き込み) 禁止にすることができます.

これをみるために, まずは ls コマンドを

```
ls -l
```

のようにオプション「–l」付きで実行してみましょう. すると画面には次のような表示が現れるはずです.

```
-rw-r--r-- 1 j-inoue staff 17 4 月 13 日 11:13 check.txt
```

この表示で先頭の文字「–」を除く (この「–」に d が立っていれば, それはディレクトリであることを意味します) はじめの 9 文字は 3 つずつ 1 組で意味を持ち, 左から「所有者」「グループ所属者」「第 3 者」を意味し, そこに表記されている「r」は「読み込み可」を「w」は「書き込み可」を, また上の例では該当しませんが, 「x」は「実行可」を意味します. 従って, 上の例で言えば, check.txt の所有者である j-inoue は書き込みも読み込みも可能であり 1, グループ内の人たち (staff ディレクトリにホームディレクトリのある人たち) は読み込みにみ可能であり, 第 3 者の人たちに対しても読み込みのみが可能であることを意味しています.

従って, **練習問題 6.8** で皆さんがこのファイル check.txt を読むことができたのはこのファイルが上記のようなアクセス権限であったためであったわけです.

さて, ここで私がこのファイル check.txt を他人に見せたくないと思った場合, どのようにファイルのアクセス権限を変えたらよいのでしょうか ? この場合にはコマンド chmod を用います. (**ch**ange **mod**e の略). 具体的には

```
chmod o-r check.txt
```

とすれば OK です. このモード指定の先頭の「o」は他人 (others) を意味し, 真ん中の「−」は禁止の意味です. 最後の「r」は「読み込み」を意味しますので, これらを 3 つ組み合わせた「o−r」は「第 3 者の読み込みを禁止する」ということになります. また, このファイルを再び第 3 者が読み込み可能にするのであれば

```
chmod o+r check.txt
```

とモード指定の真ん中の「−」を「+」に変更すれば OK です. 「o」の部分は「a」(all, 全て), 「u」(usr, 所有者), 「g」(group members) に変更が可能です. また, 「r」の部分も「w」(書き込み), 「x」(実行) に変更可能です. 実際にこれらを次の練習問題で確かめてください.

練習問題 6.9
自分の tmp ディレクトリにある test.c のアクセス権限モードを確認し, コマンド

```
chmod g+w,o-r test.c
```

を実行した後のアクセス権限モードと比較せよ.

2.1.4　オンライン・マニュアルの活用

以上確認した各種コマンドの詳細はコマンド man (**man**ual の略) で確認できます. 例えば, less の使い方に関する詳細が知りたければ

```
man less
```

とすれば, less の活用法に関する詳細が閲覧できます. この講義ではフォローしきれない, しかし, 使い勝手が良いコマンドもいくつかありますので, この man でこまめに確認することが Linux を上達するための近道かもしれません. なお, man を停止するには q を押してください[5] .

2.2　プログラムの開発手順 : まずは簡単に説明

ここからは簡単なプログラムを Xemacs により書き, それをコンパイル・リンクすることにより実行形式ファイルに変換し, それを計算機上で実行するという一連の流れを確認していくことにします. なお, 今回学ぶ事項は教科書の 8 ページまでに書かれた内容です.

2.2.1　計算機に仕事をさせるための流れ

何かの仕事を計算機にさせる場合, 下に示した一連の作業を行うことになります.

プログラムファイル (*.c) ⇒ **コンパイル, リンク** ⇒ **実行形式 (a.out)**

[5] man は調べたいコマンドと共に man *** と呼ばれるごとに, システム上に置いてある該当するコマンド *** の説明が書かれたテキストファイルを読み込みに行き, そのファイルを less コマンドで画面表示する働きをします. 従って, 画面のスクロール法も less と同じであり, 終了方法も less の停止方法と同じ q となります.

以下で各々のプロセスを簡単に見ていくことにしましょう.

- **Step1 : Xemacs による C プログラムの作成**
  このステップではデータ構造を定義し, そのデータに対してどのような操作をどのような順序で行うかを, 一貫した文法の下に書き下すことを行います. 例えば, 先に書いてもらった test.c を思い出していただくと

```
#include<stdio.h>
main()
{
   printf("Today is 13th April 2007 and Friday!");
}
```

  でしたが, これは C 言語による最も簡単な典型です. このプログラムの意味は「Today is 13th April 2007 and Friday!を画面に表示する」という意味です.

  この例に見るように, C 言語によるプログラムは一般的に

```
main()
 {
     文 1;
     文 2;
         ..............
         ............

.....................
  }
```

  のような形をしています. ここで, 先頭の main() というのは main という名前の**関数**の始まりを意味します. この関数の働きは「{」から「}」までの文によって定義され, **関数**はデータの入出力, 計算, 計算順序の制御などといった基本的な動作を表す文を並べることにより定義されます. また, この main 関数はプログラムの中にただ一つだけ存在しなければなりません.

  さて, もう一度 test.c を見てみると, このプログラムの場合の main 関数の定義範囲は 3 行目の「{」から 5 行目の「}」までです. 次に「{」から「}」に挟まれた「文」に関してですが, この文 (statement) は数値や文字に対して演算を行ったり, 関数の実行を指示したり, 条件によって次の処理を選択したり, あるいは一定の条件を満たしている間, ある処理を繰り返すといった動作を表します. 文は 1 行である必要はなく, セミコロン「;」で終わる文字列です. 以上のことをふまえて再度 test.c を見てみると, 先頭の

```
#include<stdio.h>
```

  ですが, これはプログラムで共通に用いる各種関数や変数の定義を一つのファイルにまとめて stdio.h という名前をつけておき, それをこのプログラム (test.c) の中に取り込んで用いることを表しています. このような形式でプログラムの中に取り込む「.h」という**拡張子** (ファイルの種類・属性を識別するためのタグ) のついたファイルを**ヘッダファイル**と呼びます. この一行は必要な場合に書けば良

いのですが, 必要の無い場合に書いたとしても何も問題は無いので, 通常はこの 1 行を必ず書き込むようにしましょう.

プログラム test.c で次に問題になるのが

```
printf("Today is 13th April 2007 and Friday!");
```

ですが, これは関数 printf() であり, 一般に

```
printf("画面に表示させたい文字列");
```

のような形式をとり, 「"...."」で囲まれた部分を画面上に表示させる関数です. 従って, main 関数の中でこの printf 関数を呼ぶことにより, プログラム全体としては「Today is 13th April 2007 and Friday!を画面に表示する」という働きをすることになるのです.

さて, これを実行するためには, このプログラムをコンパイル (およびリンク) し, **実行可能ファイル (executable file)** に変換しなければなりません.

- **Step 2 : コンパイル・リンクの方法**
  test.c をコンパイル・リンクして実行可能形式に変換するには

  ```
  gcc test.c -lm
  ```

  とします. ここで, 最後のオプション「−lm」はサインやコサインなどの数学関数を用いるときには付けなければなりません. 数学関数を用いない場合にこのオプションをつけても問題は無いので, 通常はこれをつけるようにしましょう. このコンパイルが成功するとカレントディレクトリに

  ```
  a.out
  ```

  という名前のファイルができているはずです. 次の練習問題で確かめましょう.

  練習問題 7.1
  test.c に関して各自がコンパイルを実行してみよ. 次いで, ls コマンドを用いてカレントディレクトリに a.out が存在するか確認せよ. また, test.c の printf 関数の最後のセミコロン (;) を削除したプログラムを test2.c として保存し, この test.c を同様にコンパイルしたとき, どのようなエラー・メッセージが画面に表示されるかを確認せよ.

- **Step 3 : 実行**
  コンパイルで実行形式ファイル a.out が作成された場合, この実行形式のファイルを実行するには

  ```
  ./a.out
  ```

  とすれば OK です. より一般的には

  ```
  ./[実行可能形式のファイル名]
  ```

  となります. 早速, 次の練習問題をやってみてください.

練習問題 7.2

test.c をコンパイルすることにより作成された a.out を実行し, 結果を確認せよ.

練習問題 7.3

次のプログラムを test2.c という名前で作成し, コンパイルしてできる実行可能形式 a.out を実行してみることでこのプログラムの意味を考えよ.

```
#include<stdio.h>
main()
{
  double a,b,x,y;
  a=5;
  b=3;
  x = a+b;
  y = a*b;
   printf("sum=%lf prod=%lf\n",x,y);
}
```

今週の LMS 入力課題

1[s] 間に自由落下する距離 [m] を計算し, その答えを画面上に表示するプログラムを作成せよ. ただし, 重力の加速度を $9.8[\mathrm{m/s^2}]$ とすること.

第3回講義・演習 — 平成19年4月20日 —

3.1 表ジョブと裏ジョブ

第 2 回講義でターミナル上で Xemacs を起動する際

```
xemacs &
```

のように「&」を最後につけて起動しました. この「&」は Xemacs というジョブ (計算機上での作業) を **裏ジョブ (background job)** として実行することを意味します. 一方, この「&」をつけないで Xemacs を起動する場合, Xemacs を**表ジョブ (foreground job)** として実行したことになります. ここで, 両者の違いは, 裏ジョブの場合にはコマンド・タスクの終了を待たずに次のジョブを実行できるという点です. 言い方を変えると, **表ジョブは一つのターミナルで常に一つであるのに対し, 裏ジョブは複数個同時に実行することができる**という点です.

3.1.1 ジョブの監視

さて, 裏ジョブはターミナルの次の入力コマンドに依存しない形で実行しているわけですから, このジョブは何らかの形で監視できなければなりません (正常に動作しているか, 現時点で何秒間動作しているか, など). そこで, この目的のためにいくつかのコマンドが用意されており, その中の一つは jobs コマンドであり, これは

```
jobs
```

とします. すると, もし Xemacs が裏ジョブとして起動しているのであれば, ターミナル上に

```
+Running xemacs
```

と表示されます. 従って, この jobs コマンドで現時点でどの作業が計算機上で実行されているのかを確認することができるわけです.

もう一つのコマンドは ps であり

```
ps
```

あるいは

```
ps -u [loginID] -l
```

とすれば OK です. 最後の「–l」オプションは「リスト形式で表示せよ」という意味です[6] . 次の練習問

[6] ps コマンドについてもう少し詳しく知りたいのであれば, 先週学んだオンラインマニュアルを早速使って man ps で調べてみよう.

題でこれを確認してみましょう.

練習問題 8.1

Xemacs を用いて, test.c と test2.c を裏ジョブとして起動することで開き, jobs コマンド, ps コマンドの各々を用いて, それらのジョブの状態を確認せよ.

3.2 変数の宣言と代入

ここからは C 言語の各論を学んでいきます. まずは, C 言語で用いる変数について見ていくことにしましょう. その準備のため先週の復習を兼ねて次の練習問題をやってみましょう.

練習問題 9.1

次のプログラムを Xemacs を用いて書き, hensu.c という名前で保存し, 次いでコンパイル, 実行し, 結果を確認せよ.

```
#include<stdio.h>
main()
{
        int add, sub, pro, div, mod, x, y;  /* 変数の宣言 */

        x = 8; /* x に 8 を代入 */
        y = 4; /* y に 4 を代入 */

        add = x + y; /* x と y を足したものを add に代入 */
        sub = x - y; /* x から y を引いたものを sub に代入 */
        pro = x*y; /* x と y を掛け合わせたものを pro に代入 */
        div = x/y; /* x を y で割ったものを div に代入 */
        mod = x%y; /* x を y で割った余りを mod に代入 */

        printf("%d %d\n", x,y);
        printf("%d %d %d %d %d\n", add,sub,pro,div,mod);
 }
```

上記のプログラムで「/*」と「*/」で囲まれた部分は「コメント」であり, コンパイル時には無視される部分です. 従って, この部分には何を書き込んでも良く, その行がどのような処理を行う (行わせようとしている) のかをこまめに書き込んでおくと良いでしょう. 時間が経った後では自分の書いたプログラムでさえも, どのようなものであったかを思い出すことは容易ではないことが結構あります. このような状況を回避するためにも, 自分の書くプログラムにはなるべくコメントを書く習慣をつけると良いと思います.

さて, hensu.c の main() 関数の先頭にある

```
int add, sub, pro, div, mod, x, y;  /* 変数の宣言 */
```

は「add, sub, pro, div, mod, x, y の 7 つの変数は整数型の変数として扱う」ということの宣言です. ここで用いた int 型の他に下表のように変数の性質により型が異なります.

指定子	データタイプ	表現範囲
char	文字	ASCI 文字
int	整数	$-2^{31} \sim 2^{31}-1$
float	単精度浮動小数点	$\pm 3.4 \times 10^{28} \sim \pm 3.4 \times 10^{38}$
double	倍精度浮動小数点	$\pm 1.7 \times 10^{-308} \sim \pm 1.7 \times 10^{308}$

また, hensu.c の printf() 関数ですが, これは前回も見たように, 一般的に printf(".....") のような形式をとり, "..."を表示する機能を持ちましたが, この関数はまた

```
printf("%d\n",a);
```

のような形式をとり, **変数 a を 10 進数で画面に表示せよ**という作業をさせることもできます. この %d は printf() 関数の**変換指定子**と呼ばれ, ここでの d は 10 進数を表します. 従って, もう少しこの手の printf() 関数の使い方を一般的に書けば

```
printf("%[変換指定子]\n", 変数);
```

となり, 表示したい変数が複数ある場合には

```
printf("%[変換指定子] %[変換指定子] .....[変換指定子]\n", 変数, 変数, ..., 変数);
```

とすれば良いわけです.

変換指定子と表現形式の対応は次の通りです.

変換指定子	表現形式
d,i	10 進数
u	符号無し 10 進数
o	符号無し 8 進数
x,X	符号無し 16 進数
c	文字
s	文字列
f	通常の小数形式 (例: 123.4567)
e,E	指数形式 (例: 1.234567E+02)
g,G	指数部の桁数により, f 形式と g 形式を自動振り分け

練習問題 9.2 (※ LMS 入力課題) [7]
A,B,C さんの英語, 数学, 国語, 理科, 社会の点数は以下の表のようであった.

名前	英語	数学	国語	理科	社会
A	70	60	53	67	82
B	40	93	48	81	30
C	94	30	87	36	85

このとき, A, B, C のそれぞれに対し, 5 教科平均を算出し

[7] 「※ LMS 入力課題」と書かれた練習問題は端末で実行するだけではなく, LMS の課題として投稿してください. 詳しくは講義中に指示します.

```
score_A=*** score_B=*** score_C=***
```

の形式で表示するプログラムを作成せよ.

3.2.1 変数の名前

C 言語における変数の命名規則を次に列記しておきます.

(1) 最初の文字は英字か下線 (_) でなければならない.
(2) 2 番目以降の文字は英字か下線か数字でなければならない.
(3) 大文字と小文字は区別される.
(4) 変数名の長さには制限はないが, 最初の 31 文字までが有効である.
(5) 途中に英字, 下線, 数字以外の文字があってはいけない.
(6) 途中にスペース, タブ, 改行記号があってはいけない.

また, int, printf, main 等, C 言語の文法上で明確に定義され, 予め意味を持ち, 予約されたもの (所謂「予約語」) は変数として用いることができません. このような変数を用いた場合にはコンパイル時エラーが出るはずですから, その際にはもう一度変数名を考え直し, 適切な変数名を用いるようにします.

3.2.2 代入による型変形

ここでは, ある型で宣言した変数に, その宣言とは異なる型のデータが代入された場合にどのようなことが生じるのかを調べてみましょう. まず, 次のケースを考えます.

```
float x; /* x という単精度浮動小数点型変数を宣言する */
x = 12; /* 浮動小数点型変数 x に整数型データを代入する */
```

この場合には, **整数型が型変形され, 浮動小数点型になって x に代入される**ことになります.
また, 次のケースはどうでしょうか.

```
int k; /* k という整数型の変数を宣言する */
k=12.3456; /* 実数データを整数型の変数 k に代入する */
```

この場合には**浮動小数点データが整数型に型変換される**ことになります. 次の練習問題を見てみましょう.

練習問題 9.3
次のプログラムを書き, katahenkei.c という名前で保存し, コンパイル・実行せよ. そこで得られた結果について考察せよ.

```
#include<stdio.h>
main()
{
     int i;
     double x,y;
     x=i=y=3.141592;
     printf("x=%lf y=%lf i=%d",x,y,i);
}
```

3.2.3　キャスト演算

一般的に次の形:

```
(型名) 式;
```

によって, まずは「式」が評価され, その値が型名で指定された型に変換されます. このような変換を**キャスト (cast)** と呼び, 型名を () でくくったものを**キャスト演算子 (cast operator)** と言います. **キャスト演算は剰余算よりも先に行われる**ことを注意しておきましょう. 実際の使い方は次のプログラムのようになります.

```
#include<stdio.h>
main()
{
     int sum,N;
     float f;
     sum=314;
     N=100;

     f = (float)sum/N;
     printf("f=%f\n",f);
}
```

このプログラムでは整数型の変数 sum をキャスト演算子で単精度浮動小数点に直し, その値を整数 N で割った値を浮動小数点型の変数 f に代入し, それを表示させています. これをふまえて, 次の練習問題をやってみましょう.

練習問題 9.4 (※ LMS 入力問題)
次のプログラムを書き, 答えが正しいかどうかを確認せよ. また, 正しくない場合にはキャスト演算子を用いて正しい結果が出力されるようにプログラムを変更せよ.

```
#include<stdio.h>
main()
{
    int i,j;
    double f;
    i=7;
    j=2;
    f=i/j;
    printf("f=%lf\n",f);
}
```

3.2.4　混合演算による型変形

異なる型どうしの演算 (混合演算) の場合, 精度の高い方への型変換が行われます. 次の例で確認しておきましょう.

```
#include<stdio.h>
main()
{
        int k;
        float x;
        double y;

        k+1; /* 型変形無し */
        k+x; /* k が float に型変換される */
        k+y; /* k が double に型変換される */
        x+y; /* x が y に型変換される */
        ..................
        .........
        ......
}
```

3.2.5 様々な演算子とその使い方

教科書の pp. 21-27 にリストアップされている各種演算子の中でさしあたり比較的使用頻度の高いものをピックアップして説明します.

3.3 関数 scanf() の使い方

変数に関して学んだところで, scanf() という関数の使い方について見ておきましょう. scanf() 関数は

```
scanf(書式, &変数名 1, &変数 2, ... ,&変数 n);
```

のようにプログラム中で用います. この機能としては, **キーボードからの文字列として与えられるデータが, 書式に記されている各変数指定に従って, 計算機内部の値に変換され, 変数 1, 変数 2, ... , 変数 n に順次格納される**ということです.

具体的には次のような使い方をします.

```
#include<stdio.h>
main()
{
            int data1,data2,add;
            scanf("%d %d",&data1,&data2);
            add=data1+data2;
            printf("add=%d\n",add);
}
```

これを例えば, examScanf.c という名前で保存し

```
gcc examScanf.c -o b.out -lm
```

としてコンパイルし, その後

```
./b.out
```

として実行すると, 画面は入力待ちの状態になるので, そこに

```
12 34
```

と打ち込むと, 画面には

```
add=46
```

と出力されるはずです. つまり, キーボードから打ち込まれた 12 と 34 はそれぞれ変数 data1, data2 に格納され, これを足したものが add に代入された後, printf() 関数により画面に表示されるわけです.

3.4 流れの制御

計算機に何かの作業をさせたい場合, ある条件に合致するときのみ, それを実行させたいとか, いくつかの場合を想定し, その場合に応じて異なる処理を行わせたいというようなことがたびたびあります. ここからは C 言語でそのような「条件分岐」を含む処理をどのようにして行わせるのかについて詳しく見ていきます.

3.4.1 if 文

まず, 最も基本的な **if 文**と呼ばれる制御法の使い方はプログラム中で次のように書きます.

```
if(式) 文;
```

ここで, この if 文の機能は () 内が評価され, その値が 0 でなければ (真ならば), () に続く文が実行されます. もし, () 内が 0 ならば (偽ならば), () に続く文は実行されず, この if 文に続く次の文が処理の対象になり, 実行されます.

実際には次のようにして使うことになります.

```
#include<stdio.h>
main()
{
            float h,w,s;
            scanf("%f",&h);  /* キーボードから身長を読み込む */
            scanf("%f",&w); /* キーボードから体重を読み込む */
            s=(h-100)*0.9;  /* 標準体重を算出 */

    /* もし, 体重から標準体重を引いた値が 10 以上であれば警告を促す */
            if(w-s >=10.0)
            printf("Be careful ! You are overweight! \n");
}
```

この例で見るように, if 文は () 内の関係式 (あるいは等価式) が成立するときのみ, それに続く文を実行する働きを持ちます. 関係式, 等価式に用いる**関係演算子**, **等価演算子**は以下の通りです.

関係演算子	一般形	意味
$<$	e1 $<$ e2	$e_1 < e_2$ なら 1, 違えば 0
$>$	e1 $>$ e2	$e_1 > e_2$ なら 1, 違えば 0
$<=$	e1 $<=$ e2	$e_1 \leq e_2$ なら 1, 違えば 0
$>=$	e1 $>=$ e2	$e_1 \geq e_2$ なら 1, 違えば 0
等価演算子	**一般形**	**意味**
$==$	e1 $==$ e2	$e_1 = e_2$ ならば 1, 違えば 0
$!=$	e1 $!=$ e2	$e_1 \neq e_2$ ならば 1, 違えば 0

※ 特に等価演算子の== と代入を意味する= を間違うケースが非常に多いですから注意してください.

3.4.2 if else 文

条件分岐をさせるには次のような **if else 文**を用いることもできます.

```
#include<stdio.h>
main()
{
            float h,w,s;
            scanf("%f",&h);  /* キーボードから身長を読み込む */
            scanf("%f",&w); /* キーボードから体重を読み込む */
            s=(h-100)*0.9;  /* 標準体重を算出 */

    /* もし, 体重から標準体重を引いた値が 10 以上であれば警告を促す */
           if(w-s >=10.0)
           printf("Be careful ! You are overweight! \n");

    /* そうでなければダイエットの必要はないと表示する */
            else
            printf("OK. You do not need diet. \n");

}
```

if else 文は次のような「入れ子」の形で用いることもできます.

```
if (式 1) {文 1;
}else if (式 2){文 2;
}else if (式 3){文 3;
.....................
.....................
}else if (式 n){文 n;}
```

今週の LMS 入力課題

2 次方程式 $x^2 + ax + b = 0$ の係数 a, b を尋ねられ, キーボード入力により, a, b の値を入力すると, この 2 次方程式の解が 2 つの実根を持つ場合には 2 つの解を出力し, 重根の場合には 1 つの解のみを表示し, 虚数解を持つ場合には “No Solution” と表示するプログラムを作成せよ.

※ 注意 : ルートを使う場合には, 例えば, 実数 a のルートを実数 y に代入したければ

```
y=sqrt(a);
```

とします. また, この際には

```
#include<math.h>
```

の一行を先頭に加えてください.

第4回講義・演習 — 平成19年4月27日 —

3.4.3 switch case 文

流れの処理を行わせるには, 先週学んだ if 文, if else 文の他に **switch case 文**があります. 今回はまずこの使い方について詳しくみていきましょう.

switch case 文の使い方は以下の通りです.

```
switch(式){
        case 定数1: 文1.................
                break;
        case 定数2: 文2.................
                break;
        case 定数3: 文3.................
                break;
                ..............
                ..............
                ..............
        case 定数n: 文n.................
                break;
        default: 文n+1...................
                break;
}
```

このとき, 処理の流れはまず, (式) が評価され, その値と各 case の後に書かれた定数を順次に比較します. その再, もし, (式) と一致する定数 k が存在すれば, その後の文 k 以降を実行されます. その実行中に break; に出会うと, switch case 文全体の実行を終了します. もし, break; が無ければ switch 文の終わりまでの文を順次に実行することになります. この switch case 文の使い方に慣れるために次の **練習問題 11.1** をやってみましょう.

練習問題 11.1

次のプログラムを作成, コンパイルし, 実行せよ. 特に, break 文の位置による違いを確認せよ.

```
#include<stdio.h>
main()
{
        int x;
        printf("x=");
        scanf("%d",&x);
```

```
        printf("******* break in all three lines *******\n");
    switch(x)
    {
                case 1: printf("1\n"); break;
                case 2: printf("2\n"); break;
                case 3: printf("3\n"); break;
    }

     printf("****** no break in all three lines *******\n");
    switch(x)
    {
                case 1: printf("1\n");
                case 2: printf("2\n");
                case 3: printf("3\n");
     }

     printf("****** break in the second line *****\n");
    switch(x)
    {
                 case 1: printf("1\n");
                 case 2: printf("2\n"); break;
                 case 3: printf("3\n");
     }
}
```

この練習問題での結果を念頭に次の問題を LMS 入力問題として考えてみてください.

練習問題 11.2 (※ LMS 入力問題)

各種運動を一定時間行った場合の消費カロリーが次表で与えられている.

運動種目番号	運動種目	消費カロリー (K cal/分)
1	縄跳び	8.60
2	ジョギング	6.25
3	エアロビクス	4.41
4	散歩	2.25

このとき, 運動種目番号と運動時間 (分) をキーボード入力から与えると消費カロリーを計算して画面表示するプログラムを作成せよ.
※ 「運動種目番号」を switch case 文の case の番号に対応させ, 分岐を行わせます.

3.5 数学関数

科学計算, 工学計算では $\sin x, \cos x$ などの数学関数を用いるのが普通です. C 言語には予め用意されている数学関数があり, これは math.h というファイルの中で定義されています. これらの数学関数を用いるためには, プログラムに

```
#include<math.h>
```

という一行を書き込んでおく必要があります. また, 既に注意を促していましたが, コンパイルを実行する際には

```
gcc -o b.out test.c -lm
```

のように –lm オプションをつけることを忘れないようにしてください.

以下に主だった数学関数を列記しておきます.

関数	説明
sin(x)	x の sine(正弦), 単位はラジアン
cos(x)	x の cosine(余弦), 単位はラジアン
tan(x)	x の tangent(正接), 単位はラジアン
asin(x)	$\sin^{-1}(x)$, x の範囲 $[-1, 1]$, 関数値の範囲 $[-\pi/2, \pi/2]$
acos(x)	$\cos^{-1}(x)$, x の範囲 $[-1, 1]$, 関数値の範囲 $[-\pi/2, \pi/2]$
atan(x)	$\tan^{-1}(x)$, 関数値の範囲 $[-\pi/2, \pi/2]$
atan2(x,y)	$\tan^{-1}(y/x)$, 関数値の範囲 $[-\pi, \pi]$
log(x)	x の自然対数 $\ln(x)$, $(x > 0)$
log10(x)	x の常用対数 $\log(x)$, $(x > 0)$
pow(x,y)	x の y 乗. x^y
fabs(x)	x の絶対値
sqrt(x)	x の平方根. $\sqrt{x}$ $(x > 0)$
floor(x)	x より大きくない最小の整数
ceil(x)	x より小さくない最小の整数

※ このリストにない関数でも, 後にみるように, 自分で必要な関数を作成し, それをメイン関数の中に呼び込んで用いることもできます. 次の練習問題をやってみましょう.

練習問題 12.1 (※ LMS 入力問題)
三角形の 3 つの辺の長さ a, b, c をキーボードより入力すると, その面積 s を次の公式:

$$s = \sqrt{t(t-a)(t-b)(t-c)},\ t = (a+b+c)/2$$

によって計算し,

```
s=*****
```

の形式で画面に表示させるプログラムを作成せよ. ここで, a, b, c **が三角形をなすためにはどの辺も他の 2 辺の和よりも小さくなくてはいけないことに注意し (三角不等式)**, もし, この条件を満たさない場合には impossible と表示するようにプログラムを書くこと.

3.6 反復制御

ここからは, ある作業を計算機に繰り返し行わせるための方法 — ループ処理 (繰り返し処理) — のいくつかを学んでいきます.

3.6.1 while 文

ループ処理の中でも while 文と呼ばれる反復制御法は次のような構造を持っています.

```
while(条件式){
        文 1;
        文 2;
        ..................
        ..................
        文 n;
 }
```

この while 文は「条件式」の値が 0(偽) になるまで次の手順でループ本体を繰り返します.

(1) **繰り返しの判定**: 「条件式」を評価し, s の値が 0(偽) ならば, この while 文を終了し, {....} に続く文の実行に移る.
(2) **ループ本体の実行**: ループ本体を実行し, (1) 戻る.

具体的な使い方を見るために, 次のような練習問題を見ておきましょう.

練習問題 13.1
次のプログラムをファイルに打ち込み, コンパイル・実行し, while 文の動作を確認せよ.

```
/* while 文の動作の確認 */
/* s=1,....,10 に対し，それぞれの逆数，2 乗，平方根を表示させる */
#include<stdio.h>
#include<math.h>

main()
{
        int s;
        double a,b,c,value;
        s = 1;
        printf("Number Inverse Squre Root\n");
        printf("-----------\n");

        while(s <= 10)
        {
                 value = s;
                 a = 1/value;
                 b = pow(value,2);
                 c = sqrt(value);

                printf("%f %f %f %f\n", value, a,b,c);
                s = s + 1;
        }
```

```
}
```

※ いくつかの注意事項

(1) while 文の処理がただ一つの文だけの場合には中括弧 {} を省略できる.
(2) 次のような while 文を**無限ループ**と呼ぶ.

```
/* 無限ループの例 */
while(1)
{
        printf("beautiful life!\n");
}
```

プログラミングの際にこのような無限ループを作ってしまうと, 永遠にジョブが終了しないので, 絶対に避けなければならない. 自分でも気づかないうちに無限ループを作ってしまっている場合もあるので, なかなかジョブが終了しない場合には Ctrl-C でジョブを強制的に終了し, 無限ループができていないかどうかをチェックすると良い.

3.6.2 for 文

繰り返し処理を行わせるには次に挙げる **for 文**を使うこともできます.

```
for(式 1; 式 2; 式 3)
{
        文 1;
        文 2;
        ..............
        ..............
        文 n;
}
```

for 文では, 式 2 の値が 0(偽) になるまで, 次のようにループ本体が繰り返し実行されます.

(1) **ループの初期化**: 式 1 を評価し, 繰り返し用いる変数に必要な初期値を代入する.
(2) **繰り返しの判定**: 式 2 を評価し, その値が 0(偽) ならば for 文の実行を終了し, そうでなければ次のステップへ移る.
(3) **ループ本体の実行**: ループ本体の文を実行する.
(4) **繰り返し後の演算**: 式 3 を評価する. 式 3 は繰り返しを制御する変数の値を増加, あるいは減少させる. その後 (2) へ移る.

ここでも for 文の使い方を実際に見てもらうため, 次のような練習問題をやってみましょう.

練習問題 12.2
次のプログラムをファイルに書き込み, コンパイル・実行し, for 文の動作を確かめよ.

```
#include<stdio.h>
#include<math.h>
```

```
main()
{
        int i,n;
        n=0;
        for(i=1; i<=10; i=i+1){
                n=n+i;
                printf("sum is %d\n",n);
        }
}
```

※ 注意

for 文の場合にも無限ループができるので注意が必要です. 次のように書くと無限ループができます.

```
for( ; ; ){
        printf("beautiful life!\n");
        }
```

今週の LMS 入力問題

(1) $s = \sum_{k=1}^{10000}(6/k^2)$ を計算し

```
s=*****
```

の形式で表示するプログラムを作成せよ.

(2) 漸化式:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{s}{x_n}\right), \ x_1 = 3$$

を考えよう. この漸化式の $n = 100$ の項を

```
x=*****
```

の形式で表示させるプログラムを作成せよ. ただし, この漸化式の中に現れる s の値は (1) で求めたものであるとする.

※ 参考 1: C 言語において a=b とは b の値を a に「代入する」ことであったことを思い出すと良い.
※ 参考 2: どのようにプログラムを組んだらよいのかは, 具体的に漸化式のはじめの数項を紙に書き出してみて「感じ」をつかむと良い.「面倒だな」と思った部分を計算機に渡して処理させる.

(3) (2) での漸化式が収束する場合にその収束値を求めたい. 今回学習した for 文を用いて繰り返しの最大値を 10000 程度とし, 条件: $|x_{n+1} - x_n| < 10^{-5}$ を満たすまで反復を繰り返し, この条件が満たされた時点での x_n の値を

```
x=*****
```

の形式で表示させるプログラムを作成せよ. このとき, うまくプログラムが動作していれば, この収束値は円周率 π の良い近似値になっているはずである.

※ 注: 今週の問題の LMS への入力期限は 5/10(月) 午後 5 時までとします.

第5回講義・演習 — 平成19年5月11日 —

3.6.3 do-while 文

繰り返し制御の最後に **do-while 文**を学びます. do-while 文の一般形は次のようになっています.

```
do
{
        文 1;
        文 2;
        ............
        ............
        ......
        文 n;
}while(条件式);
```

そして, 具体的には次のように実行されることになります.

(1) **ループ本体の実行：** ループ本体を実行し, 次のステップへ移る.
(2) **繰り返しの判定：** while 文の「条件式」を評価し, 0(偽) であれば do-while 文全体の実行を終了し, 0 でなければ (1) へもどる.

これも次の練習問題にあたることで実際に動作を確かめてみましょう.

練習問題 13.3

次のプログラムをファイルに書き込み, コンパイル・実行することによって動作を確かめよ.

```
/* do-while 文の動作を確認するためのプログラム */
/* 最大公約数を求める */
#include<stdio.h>
#include<math.h>

main()
{
        int a,b,m,n,r;
        scanf("%d %d",&a,&b);
        m=a;
        n=b;

        do{
             r=m%n; /* m を n で割った余りを求め，その値を変数 r に代入 */
```

```
            m=n; /* n の値を m に代入 */
            n=r; /* r の値を n に代入 */
        }while(r!=0);

        printf("G.C.D. of %d and %d is %d\n",a,b,m);
}
```

※注意事項：

(1) 条件式の値がはじめから 0(偽) であっても, **ループ本体の文は少なくとも 1 回は実行される**.
(2) while 文, for 文はループ本体の実行前に条件式の評価を行うが, do-while 文は**最後**に行う. 従って, 最低でも 1 回はループ本体を実行する必要のある場合には do-while 文を用いるべきである.
(3) 次のような記述は無限ループを生成する. 注意されたし.

```
/* 無限ループ */
do
 {
        printf("beautilful life !\n");
 }while(1);
}
```

3.7　プログラムを書く際の注意点

ここからは C 言語プログラミングの本題をやや外れて, 良いプログラムとはどんなものか, ということを少し考えてみたいと思います. まず, 良いプログラムかどうかというのは様々な観点から評価されるべきであり, 例えば次のような見方があげられます.

- 読みやすいプログラム
- (自分もしくは他人が) 修正しやすいプログラム
- 効率の良いプログラム
- 実行時のエラーが少ないプログラム

これらのそれぞれを以下で詳しくみて行くことにしましょう.

3.7.1　読みやすいプログラム

プログラムとしての「読みやすさ」とは何を指すのかを見ていきます.

段付け (indent)

次の例に見るように適切な**段付け (indent)** をつけるとプログラムが格段に読みやすくなります.

```
/* 段付けの例 */
for(i=0;i<=9; i=i+1){
       n = n + 1;
```

```
        printf("%d\n",n);
 }
```

しかし, 次のようにしてしまうと段付け自体が本質的な間違いを犯してしまいますので, 段付けにも細心の注意が必要です.

```
/* 段付けが問題を引き起こす例 */
#include<stdio.h>
#include<math.h>
main()
{
        int t;
        t=0;
        while(t < 10)
        printf("t=%d\n",t);
        t = t + 2;
}
```

練習問題 14.1 (※ LMS 入力問題)
上記の「段付けが問題を引き起こす例」にあげたプログラムは無限ループを生成してしまう. このプログラムを修正し, 無限ループを回避せよ.
※ 万が一, 無限ループが生じてジョブが止まらなくなった場合には, Ctrl-C で強制終了させること.

簡潔な記述

C 言語には**特殊代入演算子**や**インクリメント演算子**, **デクリメント演算子**があり, これらを用いることで簡潔な記述が可能になります. まず, 以下に特殊代入演算子について表にまとめておきましょう.

特殊演算子	説明	意味
$+=$	加算する	$x = x + \text{value}$
$-=$	減算する	$x = x - \text{value}$
$*=$	乗算する	$x = x * \text{value}$
$/=$	除算する	$x = x/\text{value}$

また, インクリメント, デクリメントに関しては以下の通りです.

○ インクリメント :
意味 : 整数変数に 1 を加えることを意味する.

表現例 1 : count + +;
区別 : count の値を使用した後にインクリメントを行う.
表現例 2 : + + count;
区別 : count の値を使用する前にインクリメントを行う.

○ デクリメント :
意味 : 整数変数から 1 を引くことを意味する.

表現例 1：count − −;
区別：count の値を使用した後にデクリメントを行う.
表現例 2：− − count;
区別：count の値を使用する前にデクリメントを行う.

次の練習問題で確認しておきましょう.

練習問題 14.2
インクリメント演算子の使用法を確認するために, 各自が次の 2 つのプログラムを書き, コンパイル・実行することによって出力結果を比較せよ.

```
/* インクリメント (区別 1) */
#include<stdio.h>
#include<math.h>
main()
{
        int x=6,y;
        y=x++;
        printf("%d\n",y);
}
```

```
/* インクリメント (区別 2) */
#include<stdio.h>
#include<math.h>
main()
{
        int x=6,y;
        y=++x;
        printf("%d\n",y);
}
```

コメントの活用, 変数名の工夫

コメントに関しては既に述べてますのでここでは繰り返しません. 変数名に関してですが, 基本的にプログラムの中に現れる全ての変数は何らかの意味を持っているはずです. 例えば, 既に見てきましたが, 2 次方程式の解, 三角形の辺などを変数を使って表す場合に, x,a,b,c などを用いました. 簡単なプログラムで変数の数も少ない場合にはプログラムを読めば大体の変数の意味までわかってしまいますが, プログラムが長くなり, 変数の及ぶ範囲も広い場合, その変数の名前を見ただけで何を表しているのか, がある程度わかるものが望まれることになります. 例えば, 2 次方程式の解の場合には kai1,kai2 あるいは, solution1, solution2 を用いたり, 辺の場合には side1,side2,side3 などです. また, これ以外にループなどをまわす際の変数は通常, アルファベットの小文字の一文字を使う場合が多いようです. つまり

```
for(i=0; i < N; i++){
```

```
        ..........  ;
}
```

として i あるいは j を用いるように. ただし, 小文字の l (エル) は数字の 1 と間違いやすいので, l は単独で用いずに何か別のアルファベットと一緒に組み合わせたりして, 数字の 1 と明確に区別できるようにすると良いかも知れません.

3.7.2　修正しやすいプログラム

プログラムは未知の修正に対しても, 可能な限り修正ができるように作成することが必要です. 例えば, 次のプログラムでは **define** を用いて変数の値を変更しています.

```
/* #define を用いた変数の変更 */
/* まずは #define を使わない場合で書いてみましょう */
#include<stdio.h>
#include<math.h>
main()
{
        int i;
        for(i=1; i <= 100; i++){
                n = n + i;
               printf("%d\n",n);
        }
}
/* 続いて #define でループの上限を与える書き方 */
#include<stdio.h>
#include<math.h>
#define N 100
main()
{
        int i;
        for(i=1; i <=N; i++){
               n = n + i;
               printf("%d\n",n);
        }
}
```

3.7.3　効率の良いプログラム

一口に「効率」と言っても様々な観点があります.

記憶効率

言うまでもなく, 記憶効率を高めるにはあまりメモリを使わないことですが, 例えば用いる変数にしても次の表のような違いがあります.

変数型	必要バイト数	表現範囲
int	4 (UNIX 高速)	$-2^{31} \sim 2^{31}-1$
short in	2 (UNIX 低速)	$-2^{15} \sim 2^{15}-1$
float	4 (UNIX 低速)	$\pm 3.4 \times 10^{-38} \sim \pm 3.4 \times 10^{38}$
double	8 (UNIX 高速)	$\pm 1.7 \times 10^{-308} \sim \pm 1.4 \times 10^{308}$
long double	10 (UNIX 低速)	$\pm 3.4 \times 10^{-4932} \sim \pm 3.4 \times 10^{4932}$

一般的に言って, メモリを低く抑えると表現精度は落ちることになります.

実行効率

変数の選択では高速演算のできる変数型を選択することが重要になります (上の表を参照). UNIX(LINUX) のプログラムを書く際, 大量メモリが必要な場合には整数型は short int, 浮動小数は float を用いるけれども, 一般的には精度と演算効率とを考慮に入れて int と double を用いることが適切です. また, 整数演算と実数演算は使用される回路が異なるので演算速度も異なります.

また, これ以外にも計算アルゴリズムを工夫することで実行効率を高めることもできます. 例えば, 変数 y の値に x の多項式を代入するような場合, 具体的には

$$y \quad = \quad 3x^3 + 2x^2 - 4x + 0.5 \tag{3.1}$$

を C 言語で書くとなると

```
y=3*x*x*x+2*x*x-4*x+0.5 /* (式 1) */
```

となりますが, これは $y = \{(3x+2)x - 4\}x + 0.5$ のように式変形できますから, 次のように書くこともできます.

```
y=((3*x+2)*x-4)*x+0.5 /* (式 2) */
```

この (式 1) と (式 2) の乗算回数と加算回数を勘定してみると次のようになります.

	乗算回数	加減算回数	計
式 1	6	3	9
式 2	3	3	6

従って, どちらとも加算回数は同じだけれども, (式 2) のように式変形を施してから計算機にジョブを渡した方が乗算回数が少なくて済み, 従って実行効率も上がるというわけです. このように, 計算回数を考えて効率の良い方の演算順序を選ぶことでも実行効率は向上します. 前に述べたように, 一般的に言って, 記憶効率と実行効率はトレードオフのようなところがありますが, この演算順序の交換や式変形で計算回数を減らす工夫は記憶効率にほとんど影響しませんから, とりわけ大容量のメモリを使う計算機シミュレーションなどではとても有効になってきます.

実行時エラーの少ないプログラム

実行時にエラーが生じると, 多くの場合, 「エラー・コード」と呼ばれる記号と簡単なエラー・メッセージが出力され, エラーの発生したソースプログラムの行数は表示されません. 従って, コンパイル時のエラーと比べ, その原因を探るのが格段と厄介になりますから, この実行時エラーを少なくするようなプログラムの書き方をすることが望まれます.

それでもやはり実行時にエラーが出る場合にはどうするかですが, 例えば, 変数の中にとても大きな数が代入されてしまったり, あるいはゼロで割ってしまうことに気づかないでプログラムを実行してしまう場合があります. その際には printf 文を用いてその変数の値を逐次表示させ, 途中で変数に何かおかしな値を代入していないかどうかをチェックすることはできます. この手のデバックの方法に関しては回を改めて後日見ていくことにしましょう.

3.8 数値計算 I

ここまで学んだ事柄を用いるとある程度までのプログラミングはできるようになっていると思いますので, ここでは実際の科学技術計算で用いる数値計算上の技法のいくつかを学んでいくことにします. その中でも, ここでは

$$f(x) \quad = \quad 0 \tag{3.2}$$

のタイプの代数方程式を数値的に解くにはどうするのかということを考えてみたいと思います.

そのような代数方程式として, 例えば $f(x) = x^2 - a^2$ の場合には, $x = \pm\sqrt{a}$ のようにその解が手で求まってしまいますが, 特殊なケースとして因数分解ができない限り, 多項式 : $f(x) = a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$ に関する $f(x) = 0$ の解は求まりません. こういう場合に解の候補 (近似解) を探すにはどうするのかを, ここでは具体的に見て行こうというわけです.

3.8.1 素朴な 2 分法で解く

ここでは簡単のため, $f(x) = x^2 - a^2$ に選び, この解 (もちろん, a が正の実数なら $x = \pm\sqrt{a}$) を数値的に求めることを考えます. この際, $f(x) = 0$ の解が範囲 : (x_1, x_2) の間に含まれていることがわかっているとしましょう. これは $f(x)$ の形があまり複雑でなければ割と難なく見積もることができるでしょう. このとき, $x = x_1$ と $x = x_2$ の中点を

$$x_c \quad = \quad \frac{1}{2}(x_1 + x_2) \tag{3.3}$$

と置きます. これをグラフで表すと図 3.11 のようになります. $f(x_c)$ の値の符号が $f(x_1)$ と同じであれば, $f(x) = 0$ の解は区間 (x_c, x_2) の中のどこかにあるわけですから, 区間 (x_1, x_2) の左端 x_1 を x_c で置き換えます. 逆に, $f(x_c)$ の符号が $f(x_2)$ と同じであれば, 区間の右端 x_2 を x_c で置きなおします. このようにして得られる新しい区間に対して, 上記の操作を繰り返して行くわけです. すると, この繰り返し回数が十分であれば, 最終的には両端 x_1, x_2 がともに $f(x) = 0$ の解 x に近づいて行くことになり, 結果として, この方程式 $f(x) = 0$ の良い近似値が得られるはずです. この手の数値解法を **2 分法**と呼んでいます.

それでは実際に上記のアルゴリズムを C 言語でコーディングし, 結果を見てみるために次の練習問題をやってもらいましょう.

練習問題 15.1 (※ LMS 入力問題)
$f(x) = x^2 - 2 = 0$ の解のうち, 正の値を持つものを 2 分法により求めたい. このとき次の 2 点に注意してプログラムを作成し, 実行せよ.

(1) 区間の両端の差の絶対値 $|x_1 - x_2|$ が 10^{-5} 以下になったときを収束点を x とし, 解 x を

```
x=******
```

の形式で表示せよ. ただし, はじめの区間を $(0, 2006)$ に選ぶこと.
(2) 繰り返し回数 i とそのときの x_1 の値 (区間の左端) と真の解 $\sqrt{2}$ とのズレ : $d = |\sqrt{2} - x_1|$ を

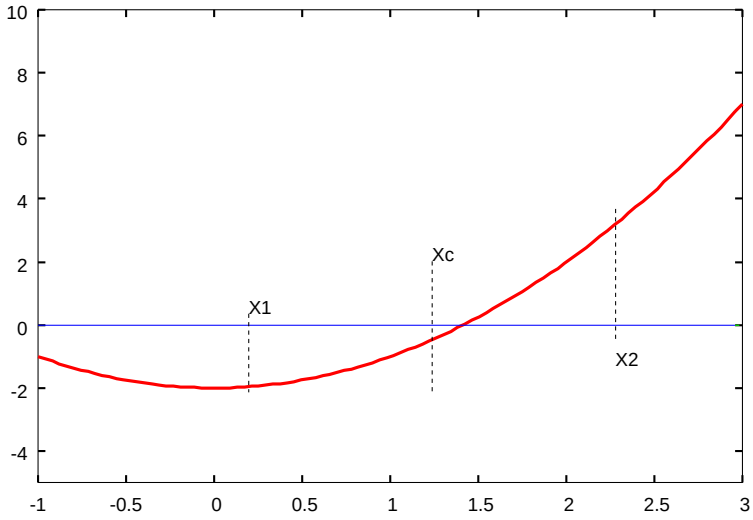

図 3.11: ここで考える 2 分法の概念図.

```
i=1 d=******
i=2 d=******
i=3 d=******
.................
............
..............
```

のような形式で表示し, 繰り返し回数とともに, この「誤差」がどのように減少していくのかを確かめよ.

※ LMS に投稿するプログラムは (1) の要求を満たすものだけでよい.
※ この手の漸化式の繰り返しに関しては前回 (第 4 回) の **LMS 入力問題** を思い出すと良い.

3.8.2　ニュートン法

この手の方程式の数値解を求めたい場合, **ニュートン法**と呼ばれる手法を用いることもできます.
この方法は方程式 $f(x) = 0$ の解の候補 $x = x_0$ で曲線 $y = f(x)$ に接する接線の方程式:

$$y \quad = \quad f(x_0) + f^{'}(x_0)(x - x_0) \tag{3.4}$$

の x 軸との交点 :

$$x_1 \quad \equiv \quad x_0 - \frac{f(x_0)}{f'(x_0)} \tag{3.5}$$

を解候補 x_0 の改良版とするもので, これを推し進めて n-番目の解候補と $n+1$-番目の解候補の間に成り立つ漸化式

$$x_{n+1} \quad = \quad x_n - \frac{f(x_n)}{f'(x_n)} \tag{3.6}$$

を反復的に解き, この収束点を $f(x) = 0$ の解とするものです.

$f(x) = x^2 - a^2$ に対する具体的なプログラムが教科書の 47 ページに載っています. これを参考にして次の練習問題をやってもらいましょう.

練習問題 15.2 (※ LMS 入力問題)
$f(x) = x^2 - 2 = 0$ の解のうち, 正の値を持つものをニュートン法により求めたい. このとき次の 2 点に注意してプログラムを作成し, 実行せよ.

(1) x_n, x_{n+1} の差の絶対値 $|x_n - x_{n+1}|$ が 10^{-5} 以下になったときを収束点 x とし, 解 x を

```
x=******
```

の形式で表示せよ. ただし, x の初期値を 2006.0 に選ぶこと.

(2) 繰り返し回数が i のときの x_i と真の解 $\sqrt{2}$ とのズレ : $d = |\sqrt{2} - x_i|$ を

```
i=1 d=******
i=2 d=******
i=3 d=******
..............
............
..............
```

のような形式で表示し, 繰り返し回数とともに, この「誤差」がどのように減少するのかを確かめよ. また, 前問 **練習問題 15.1** (2) で調べた 2 分法の場合と比べて, この誤差の減少の振る舞い方に違いが出るか否かを確認せよ.

※ LMS に投稿するプログラムは (1) の要求を満たすものだけでよい.

今週の LMS 入力問題

方程式

$$f(x) \quad = \quad x^2 - 4x + 1 = 0 \tag{3.7}$$

の小さな方の解をニュートン法で求めるプログラムを作成し, 結果を

```
x=*****
```

として表示するプログラムを作成せよ.

第6回講義・演習 — 平成19年5月18日 —

3.9 関数

計算機プログラミングを行う際, 決まりきった手続きは**関数 (function)** として定義して使用すると便利です. 既にこの講義・演習では「数学関数」として sin() や log() を用いてきましたが, これらは標準ライブラリ関数としてヘッダファイル math.h に定義されており, これを使うためにプログラムの先頭で

```
#include<math.h>
```

のように宣言してから使いました. しかし, 名の知れた数学関数ならまだしも, 複数の関数が組み合わされた関数や, 関数をある条件の下で級数展開近似してから数値的に扱いたい場合, あるいは積分形で定義された特殊関数などはプログラマ (研究者) 自身が用途に合わせて自分で作らなければならないことが多々あります.

例えば, 指数関数 e^x のを C 言語プログラミングでは exp(x) のように使うことを既に見ましたが, もし, この exp() の存在を知らない場合, 変数 x の冪での展開 :

$$\mathrm{e}^x \quad = \quad \sum_{k=0}^{n} \frac{x^k}{k!}$$

を用いたいところです. このとき, x の値そのものと共に, この展開を何次までで打ち切るか (どの程度までの精度が欲しいか), という n の値とをペアで指定すると, そのときの e^x の値を出力するような関数 : Exp(x,n) を作っておいて, 必要に応じて, sin() や log() を使ったようにプログラムの中で使えることができれば便利でしょう.

また, このようないわゆる数学関数でなくとも, あるまとまった手続きを関数として定義しておいて, それをメイン関数の中で呼ぶことで作業を行わせたい場合も今後増えていくことでしょう. このようにして別途関数を用意し, それを必要に応じてメイン関数の中に呼び込んで用いることは, 実用上必要となるばかりではなく, こうすることでメイン関数の記述を簡潔にし, 「構造化された」プログラムを書くことができるようになります. そこで, 今回はそうした関数の作り方と使い方に関して学んでいくことにしましょう.

3.9.1 関数定義の一般形

プログラムの中で関数を定義する際には次のように書きます.

```
関数型 関数名 (引数宣言)
{
        (関数を計算する手続きの並び);
        return (関数値);
    }
```

ここで「引数宣言」とは数学関数の「変数」に対応するものであり, **関数を使用する側から関数に値を引き渡すのに用いられ**, 次の形式を持ちます.

○ 引数宣言

型 変数名，型 変数名，.....，型 変数名

また, **return 文**は関数内の処理を終了させるとともに, 関数の値を決定します[8] . 関数値を持つ関数を終了させるときには

```
return（関数値);
```

の形式を用います. 一方, 関数値を持たない関数の場合には

```
return;
```

の形式を用いることになります.
　具体的に次の練習問題で確認しておきましょう.

練習問題 16.1

次のプログラムをファイルに書き, コンパイル・実行せよ.

```
#include<stdio.h>
#include<math.h>

int add(int a, int b);  /* プロトタイプ宣言（後で説明します）*/

main()
{
        int x=1,y=2,c;
        c = add(x,y);
        printf("%d + %d = %d\n", x,y,c);
}
int add(int a, int b)
{
        int c;
        c =  a+b;
        return (c);
}
```

※ 関数を使用するにあたっては次のようなことに注意する必要があります.

- 使うより先に宣言すること. 宣言する前に使ってはいけない.
- 関数実体より先に使いたい場合には**プロトタイプ宣言**を用いる. (上の**練習問題 16.1**を参照)
- 関数型は省略すると int 型となる.
- 関数値の無い関数は **void 型**として宣言する.

[8] あとで詳しく述べますが, C 言語で用いられる関数には関数値のあるものと無いものがあります. 関数値を持たないものも, 関数内で定義された処理は行われることに注意すべきです (Fortran 言語を知っている人なら, 「サブ・ルーチン」と言えばわかるでしょうか).

```
void printint(int a)
{
        printf("%d\n",a);
}
```

以上に注意して, 次の練習問題を LMS への入力課題としてやってもらいましょう.

練習問題 16.2 (※ LMS 入力問題)

(1) 実数 a, b がキーボードから入力されると, 大きな方の数字を

```
maxi=******
```

の形式で表示するプログラムを作成せよ.

(2) 実数 a と b を引数とし, それらの大きいほうの数を関数値とする関数定義 :

```
double maxi(double a, double b);
```

を作成し, その動作をテストするためのメイン関数を作成せよ. つまり, main() 関数の中で a,b の値がキーボードから入力されると

```
maxi=*****
```

と大きい方の数字を表示するプログラムを作成せよ.

練習問題 16.3 (※ LMS 入力問題)

前回 (5/11) の **今週の LMS 入力問題** で扱ったニュートン法のプログラムを用いて正の実数 a の平方根を求めるプログラムを作成したい. このとき, a の値とニュートン法の初期値 $x0$ を与えると a の平方根を出力するような関数 :

```
double Newton(double a, double x0)
```

を作り, これをメイン関数の中で呼ぶことで $\sqrt{5}$ を

```
x=*****
```

の形式で出力するプログラムを作成せよ.

3.9.2 返り値の無い関数 (void 関数)

上の課題等で学んだ関数は全て本体の中に return (...); が存在し, 何らかの値を返す関数でした. しかし, C 言語での関数の中にはこうした「返り値」を持たない関数も存在し, それを用いてプログラムを作成することもできます. そのような関数は「型」を持たないので, 宣言の仕方としては

```
void 関数名（引数宣言）
```

のように **void 型**として宣言します. こうした返り値の無い関数の例を以下に載せておきましょう.

```
#include<stdio.h>
void printint(a);  /* プロトタイプ宣言 */

main()
```

```
{
        int x;
        scanf("%d",&x);
        printint(x);
}
/*  整数型の変数 a を表示する関数．返り値が無いので関数型は void */
void printint(int a)
{
        printf("%d\n",a);
}
```

void 型関数の使い方, 変数の**記憶クラス**, **有効範囲 (スコープ, scope)** を確認するために, 次の練習問題をやってもらいます.

練習問題 16.4

次のプログラムをファイルに書き, コンパイル・実行せよ.

```
#include<stdio.h>
void sub(void); /* void 型関数 sub のプロトタイプ宣言 */
int extern_var=0; /* 整数型外部変数の宣言と初期化 */

main()
{
        int i;
        for(i=1;i<=10;i++)
        {
        sub(); /* 関数 sub を 10 回呼び出す */
        }
}

void sub(void)
{
        auto int auto_var=0; /* 自動変数 auto_var を整数型で宣言し，初期化 */
        static int static_var=0; /* 静的変数 static_var を整数型で宣言し，初期化 */
        extern int extern_var; /* 外部変数 extern_var を int 型で宣言 */

        /* それぞれインクリメントする */
        auto_var++;
        static_var++;
        extern_var++;

        /* 値を表示させる */
        printf("auto=%d, static=%d, extern=%d\n",auto_var,static_var,extern_var);
}
```

上のプログラムに見るように, 各変数は**記憶クラス**を指定することにより, その振る舞いに違いが出ます. 記憶クラスには次の 3 つがあります.

- **自動変数 (auto)**:
  その変数が使われているとき (その変数が宣言されているブロック中) のみに作成される変数. 複数回ブロックが実行された場合でも毎回新たに作成されるので, その値は保持されない.
- **静的変数 (static)**:
  プログラムの実行に先立って作成され, 実行中はその値を保持する. 初期化をする場合にも, プログラムの実行に先立って 1 回のみ行われる.
- **外部変数 (extern)**:
  注目しているブロックの外側で宣言されている変数をそのまま用いることを表すクラスである. よって, extern によって宣言された変数の実体はそのブロックの外部, あるいはプログラムの上方にあることになる.

また, 変数は宣言されている「場所」によっても有効範囲が変化します. あるブロック内で宣言された有効範囲はそのブロックのみです.

3.9.3 再帰的関数定義

関数の中で「その関数自身を用いること」を関数の**再帰的呼び出し**と言います. C 言語ではこの再帰的呼び出しが可能です. この再帰的関数定義は級数展開を用いて数学関数を近似するときなどに便利です. 次の練習問題では $n!$ を次の 2 通りの定義により求め, 後者を再帰的関数定義によって求めるものです.

$$n! = 1 \cdot 2 \cdots (n-1) \cdot n,\ 0! = 1 \tag{3.8}$$

$$n! = n \cdot (n-1)!,\ 0! = 1 \tag{3.9}$$

練習問題 16.5

次のプログラムをファイルに書き, コンパイル・実行せよ.

```
#include<stdio.h>
#include<math.h>

double fact1(int n);
double fact2(int n);

main()
{
        int a;
        printf("a=");
        scanf("%d",&a);
        printf("%d!=%f (def.1)\n", a,fact1(a));
        printf("%d!=%f (def.2)\n", a,fact2(a));
}
```

```
double fact1(int n) /* (1) 式による階乗の関数定義 */
{
        int i;
        double ans=1.0;
        for(i=1; i<=n; i++)
        {
                ans *= i;
        }
return (ans);
}

double fact2(int n) /* (2) 式による階乗の関数定義 */
{
        if(n==0){
        return (1);
        }else{
        return (n*fact2(n-1)); /* 再帰的関数定義 */
        }
}
```

今週の LMS 入力問題

$\sin x$ のマクローリン展開は

$$\sin x \quad = \quad \sum_{l=1}^{\infty} (-1)^{l-1} \frac{x^{2l-1}}{(2l-1)!}$$

で与えられる. これと教科書 p. 67 ページのプログラムを参考にして, 再帰的関数定義を用いて展開のはじめの 10 項で $\sin x$ の近似値を出力するプログラムを作成せよ. なお, 投稿するプログラムは x の値をキーボード入力すると

```
sin_10 = ******
```

の形式で出力するものとすること.

第7回講義・演習 — 平成19年5月25日 —

3.10 マクロ定義とヘッダファイル

※ 説明は教科書 pp. 69-82 を用いて行います.

練習問題 17.1 (※ LMS 入力問題)
3 個の引数のうち, 最小のものを返す関数型引数マクロ min3(a,b,c) を作成し, これを確認するメイン関数を書け. メイン関数内では a,b,c の値がキーボード入力されると, その最小値を

```
min=*****
```

の形式で出力するようにせよ.
[**ヒント**] 教科書 p.24 の「3 項演算子」を用いると良い.

練習問題 17.2
練習問題 17.1 で作成した関数マクロを user.h という名前のヘッダファイルに書きこみ, これを読み込んで **練習問題 17.1** と同じ動作をするプログラムを作成せよ.

3.11 配列

これまでプログラムを書く際には各データを名前をつけた変数に入れて扱ってきました. このような扱い方はデータ数が少ないうちは対応できますが, データ数が多くなると, 変数の定義部だけでも十分に長くなり, 見通しの悪いプログラムになってしまうことがあります. さらに, そのようにして定義した変数の操作も不便です.

この点を実際に見てみるために, 復習になりますが, 次の練習問題をやってもらいましょう.

練習問題 18.1
次のプログラムをファイルに打ち込み, コンパイル・実行せよ.

```
#include<stdio.h>
#include<math.h>

main()
{
        double a1,a2,a3,a4,a5,max;
        scanf("%lf %lf %lf %lf %lf",&a1,&a2,&a3,&a4,&a5);

        /* 最大値を求める*/
```

```
        max=a1;
        if(a2 > max){
           max=a2;
           }else{
           max=max;
           }
        if(a3 > max){
           max=a3;
           }else{
           max=max;
           }
        if(a4 > max){
           max=a4;
           }else{
           max=max;
           }
        if(a5 > max){
           max=a5;
           }else{
           max=max;
           }

        printf("max=%lf\n",max);
}
```

練習問題 18.2 (※ LMS 入力問題)
変数 a1,a2,a3,a4,a5 に対して, これらの値をキーボードから入力すると, これらの変数の最小値, 平均値, 標準偏差を計算し, 表示させるプログラム作成せよ. ただし, 平均値 m, 標準偏差 d はそれぞれ

$$m = \frac{1}{5}\sum_{i=1}^{5} a_i,\ \ d = \sqrt{\frac{1}{5}\sum_{i=1}^{5}(a_i - m)^2}$$

で定義されることに注意し, 計算結果を

```
min=***** m=***** d=*****
```

の形式で出力させること.

上記の練習問題のような場合, 次に定義される**配列**を用いると便利です.

3.11.1 配列の宣言 — 1 次元の場合 —

変数型 配列変数名 [配列要素数];

(例)

```
int a[10]; /* int 型の要素数 10 の配列に a という名前をつける */
double Input[100]; /* double 型の要素数 100 の配列に Input という名前をつける*/
```

これを用いると, 先に見た**練習問題 18.1** は次のように簡潔に書けます.

```
#include<stdio.h>
#include<math.h>

main()
{
        double a[5],max;
        int i;

        for(i=0; i<5; i++)
        {
        scanf("%lf",&a[i]);
        }

        max=a[0];

        for(i=1;i<5;i++){
                if(a[i]>max){
                  max=a[i];
                  }else{
                  max=max;
                  }
                }
        printf("max=%lf\n",max);
}
```

当然のことながら, **練習問題 18.2** で行った最小値, 平均値, 標準偏差を求めるプログラムも配列を用いて書き直すことができます. これを次の練習問題としてやってみましょう.

練習問題 18.3 (※ LMS 入力問題)
練習問題 18.2 で作成したプログラムを配列を用いて書き直せ.

配列を用いる際には次の事項に注意すべきです.

[**注意事項**]

- 要素数は確定されている必要あり. 要素数は数値定数, あるいは

  ```
  #define N 100
  ....
  ....
  main()
  {
  int a[N];
  ```

```
.....
.......
}
```

のようにして指定する.

- 要素数が N 配列では, 要素番号は 0 から始まり, $N-1$ が最終要素番号である.
- 要素番号の範囲を越えてしまうと, そのデータは無意味であるばかりでなく, プログラムの動作自体も保障されなくなる. これは常に注意を払うべきである.

3.11.2 配列の初期化 — 1 次元の場合 —

配列変数を初期化するには例えば次のようにします.

```
int a[3]={10,11,12}; /* 要素数は 3 個 */
```

このように初期化すると, a[0]=10,a[1]=11,a[2]=12 のように, 配列の各成分が初期化されます. ここで [] 内に書いた要素数が列挙された要素数よりも多い場合には, 不足している要素は初期化されないことに注意すべきです[9]. 逆に, [] 内に書いた要素数が実際に列挙された要素数よりも少ない場合には文法エラーとなります.

なお, 初期化の方法としては次のようなやり方もあります.

```
int a[]={5,6,7,8};
```

この場合には, **実際に列挙された要素数によって配列の要素数が決定する**ことになります.

練習問題 18.4

次のプログラムをファイルに書き, コンパイル・実行することで動作を確認せよ.

```
#include<stdio.h>
#include<math.h>

main()
{
        int a[]={1,2,3,4,5,6,7,8,9,10};
        int n=10,sum,i;

        sum=0;
        for(i=0; i<n; i++)
        {
                sum += a[i];
        }

        printf("sum: %d\n",sum);
}
```

[9] 一見, ゼロで初期化されても良さそうに見えるかもしれないが, 初期化されない.

今週の LMS 入力問題

サイコロを用いた賭けを行いたいのだが, 当のサイコロが無いことに気がついた. そこで, この賭けには何も利害関係を持たない A さんをつかまえて別室に待機させ, サイコロを振る必要が生じたときに, A さんにその瞬間に頭にひらめいた 1〜6 までの数字の中で好きな数字を言わせ, その数字をもって「サイコロの目の代用」とすることを考えたい. しかし, 例えば A さんはとりわけ「3」という数字が好きで, 3 を他の 5 つの数字よりも頻繁に言う傾向があるかもしれない.

(1) これを確かめるための予備実験として 1〜6 までの数字 120 個を可能な限りでたらめにキーボードから入力すると, 120 回の試行のうち 1 が出た回数, 2 が出た回数, …, 6 が出た回数, 及び打ち込んだ数の総和を表示するプログラムを作成せよ. ただし, そのときの結果を

```
number_1=***
number_2=***
number_3=***
number_4=***
number_5=***
number_6=***
sum_number=*****
```

の形式で表示させること.

(2) キーボードの数字の配置では「1」の隣に「2」が来るが, 120 回の試行のうち 12 あるいは 21 の並びが何回程度現れるかも確かめておきたい. このためのプログラムを作成せよ. 結果は

```
number_12=***
number_21=***
```

の形式で表示させること.

第8回講義・演習 — 平成19年6月15日 —

3.11.3　2次元配列の宣言と初期化

先週学んだ配列は要素数を1列に並べた1次元配列でしたが, この配列は一般次元に拡張することができます. 例えば, 2次元の配列は次のように宣言します.

```
/* 第1軸が3，第2軸が2の要素数3 x 2 = 6の2次元配列 */

int a[2][3];
```

この初期化は次のように行います.

```
int a[2][3]={10,11,12,13,14,15}; /* 表現1 */
```

これは次のように書くこともできます.

```
int a[2][3]={{10,11,12},{13,14,15}}; /* 表現2 */
```

または

```
int a[][3]={10,11,12,13,14,15}; /* 表現3 */
```

と書けば, 具体的に

```
a[0][0] ← 10
a[0][1] ← 11
a[0][2] ← 12
a[1][0] ← 13
a[1][1] ← 14
a[1][2] ← 15
```

のように初期化されることになります.

表現2のように記述すると, 直観的にわかりやすくなると思います. また, 表現3のように2次元配列 (すぐ後にみる多次元配列でも) では最終軸 (この場合には第2軸) のみが省略可能です. 以上をふまえて次の練習問題をやってもらいましょう.

練習問題 18.5

次のプログラムは2行2列の行列の和を求めるプログラムである. 各自がファイルに入力し, コンパイル・実行することで, その動作を確認せよ.

```
#include<stdio.h>
#include<math.h>
```

```
main(){
        double a[2][2]={{4,3},{5,1}},b[2][2]={{1,0},{0,1}},c[2][2];
        int i,j;

        for(i=0;i<2;i++){
                for(j=0;j<2;j++){
                c[i][j]=a[i][j]+b[i][j];
                }
        }
        printf("%lf %lf\n %lf %lf\n",c[0][0],c[0][1],c[1][0],c[1][1]);
 }
```

練習問題 18.6 (※ LMS 入力問題)

次の 2×2 の正方行列:

$$\boldsymbol{A} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, \quad \boldsymbol{B} = \begin{pmatrix} 2 & 3 \\ 4 & 5 \end{pmatrix}$$

の積 $\boldsymbol{C} = \boldsymbol{AB}$ を計算し, $\boldsymbol{C}$ の成分を

```
c[0][0]=*** c[0][1]=*** c[1][0]=*** c[1][1]=***
```

の形式で出力するプログラムを作成せよ.

3.11.4　多次元配列の宣言

配列の次元が例え何次元になろうが, 常に定義でき, 次のように宣言することができます.

```
/* このように何次元でも可能 */

int a[10][23][6].....[8];
```

3.11.5　配列のアドレス

配列の宣言として, 例えばサイズが 10 であり, その成分が整数型をもつ 1 次元配列 COUNT は

```
int COUNT[10];
```

としましたが, この COUNT という表記は「配列の名前」であると同時にこの配列の先頭, つまり, COUNT[0] のアドレス (記憶媒体上のデータの格納場所) を表しています. 実際に既にみたようにアドレス演算子を変数名の前につけると, 該当するその変数のアドレスを表記するのでしたから

```
&COUNT[0]
```

というのは COUNT[0] のアドレスですが, これは上で説明したように COUNT の値と同じ値を持ちます.

これは多次元の配列の場合も同じであって, 例えば, COUNT2[10][10] において, &COUNT2[0][0] の値と COUNT2 の値は同じになります. このことを確認するための練習問題をやってもらうことにしましょう.

演習問題 18.7

次のプログラムをファイルに書き込み, コンパイル・実行することで結果を確認せよ.

```
#include<stdio.h>

main(){
   int array1[10];
   int array2[10][10];

  /* 配列の先頭アドレスを 16 進数表示する */
    printf("array1=%x, &array1[0]=%x\n",array1,&array1[0]);
    printf("array2=%x, array2[0]=%x, &array2[0][0]=%x\n",array2,array2[0],&array2[0][0]);

}
```

3.11.6　バブルソートとクイックソート

与えられた数列をその小さい順（あるいは大きい順) に並べ替えるアルゴリズム (ソート) とその C 言語によるプログラミングを学んでいきます. ここでは代表的なソーティングのアルゴリズムである**バブルソート**と**クイックソート**を教科書 pp. 96-99 を参考にして説明していきます.

練習問題 18.8 (※ LMS 入力問題)

バブルソートを実現するプログラムを作成せよ. 具体的には, 0 から 9 までの 10 個の数字をキーボードから入力すると, それを小さい順に並べ替えて

```
**********
```

のように表示せよ.

※ この際, 必要ならば 2 つの変数 a,b を交換する手続き型引数マクロ :

```
#define swap(a,b){int t=(a);(a)=(b);(b)=t;}
```

を用いても良い. この関数マクロは以降頻繁に用いることになる.

練習問題 18.9

クイックソートを実現する次のプログラムをファイルに書き込み, コンパイル・実行し, 動作を確認せよ.
※ プログラムの各関数・各行でどのような処理を行わせているのかをコメントとして書き込むこと.

```
#include<stdio.h>
```

```
#define NMAX 10
#define swap(a,b){int t=(a);(a)=(b);(b)=t;}

void print(int array[NMAX], int N)
{
        int i;
        for (i=0; i<N; i++){
             printf("%d", array[i]);
            }
        putchar('\n');
}


void quicksort(int array[NMAX], int lower, int upper)
{
      int bound=array[lower];
      int l=lower;
      int u=upper;

    do{
        while(array[l]<bound) l++;
        while(array[u]>bound) u--;
      if(l<=u){
             swap(array[l],array[u]);
             l++;
             u--;
            }
      } while(l<u);
       if(lower<u) quicksort(array,lower,u);
       if(l<upper) quicksort(array,l,upper);
}


main()
{
      int array[NMAX];
      int N=0;
        while(1){
            printf("data> ");
            scanf("%d",&array[N]);
          if((array[N]<0) || (++N==NMAX)) break;
}

      quicksort(array,0,N-1);
      print(array,N);
```

```
}
```

今週の LMS 入力問題

3×3 の奇数陣を表示するプログラムを作成せよ. 奇数陣とは 1 辺が奇数 n の正方行列に 1 から n^2 までを埋めたとき, 縦, 横, 斜めの n 個の和が全て等しいようなものを言う (ここで考えるのは $n = 3$ の場合).
※ 以下のアルゴリズムを参考にしてもよい (教科書 95 ページの図 6.1 も同時に参照せよ).

奇数陣作成アルゴリズム

(1) n 次正方行列を用意する. 「注目要素 a」を最下段中央とし, $a = 1$ とする.
(2) 注目要素に数 a を置き, それを以下のルールで変更する.

> **注目要素の更新規則 :**
> 斜め右下に既に数が埋まっている場合には注目要素の上の要素を, また, 空いている場合には斜め右下の要素を新しい注目要素とする. 注目要素が行列外の場合には, 周期的に配置されているものと考える.

(3) a を 1 だけ増やし, 行列が埋まるまで (2) を繰り返す.

ただし, 出力結果を

```
* * *
* * *
* * *
```

の形式で表示せよ.

(参考までに)
皆さんの中には得られる奇数陣がユニークかどうかが気になっている人もいるかもしれません. 少し調べてみると出来上がる 3×3 の奇数陣としては

4	9	2
3	5	7
8	1	6

か, この回転対称の配置しかありえないことがわかります. この事実は次のようにして示すことができます. まず, 各マスに入る数字を形式的に

$X_{0,0}$	$X_{1,0}$	$X_{2,0}$
$X_{0,1}$	$X_{1,1}$	$X_{1,2}$
$X_{0,2}$	$X_{1,2}$	$X_{2,2}$

のように行列成分として表すことにすると, 奇数陣の要請より

$$X_{0,1} + X_{1,0} + X_{2,0} = a \tag{3.10}$$
$$X_{0,1} + X_{1,1} + X_{2,1} = a \tag{3.11}$$
$$X_{0,2} + X_{1,2} + X_{2,2} = a \tag{3.12}$$

となることが必要であり, 全ての数字の和は 45 になるべきですから

$$\sum_{j=0}^{2}\sum_{i=0}^{2} X_{i,j} = \sum_{i=1}^{9} i = 45 \tag{3.13}$$

であり, (3.10)-(3.12) を (3.13) 式に代入すると

$$a = 15 \tag{3.14}$$

が得られます. 従って, 縦, 横, 斜めの数字の和はいずれも 15 でなければならないことがわかります. よって, これからはこの条件下で各数字を配置していくことを考えます.

まずは数字の 1 をどこに置くかを考えます. 縦, 横, 斜めの 3 つの数字を足しあげると 15 にならなければならないわけですから, 1 が置かれた同じ縦, 横, 斜めの列に 2, 3, 4 が来ることはできません. 例えば, 1 と 4 が同じ列に来たとすると, $15 - (1 + 4) = 10$ となりますから, 残る一つの場所に今存在しない 10 を置かなければならなくなるからです. 従って, 次のような箇所に 1 を置くことはできないことになります.

●	●	●
●	1	●
●	●	●

●	●	1
○	●	●
●	○	●

上の左側の例では 1 が中央にありますから, 2, 3, 4 をどのように配置しょうが, 必ず 1 と同じ列になってしまいます. 一方, 右側の例では, 1 と同じ列にならない場所は○の 2 箇所のみであり, 2, 3, 4 のいずれか 1 つの置き場所がなくなります. 従って, 結局, 1 の置ける場所としては

●	◎	●
◎	●	◎
●	◎	●

●	●	●
●	●	●
●	1	●

上図の左側に示したように◎の箇所のみとなり, 以下では対称性を考えて, 上の図の右側に示した場所に 1 を置くことにします.

次に考えるのは 2, 3 の置く場所ですが, 2, 3 が同じ列に来ることもできません. なぜならば, 同じ列に 2, 3 があれば, 同じ列の残りの 1 箇所には $15 - (2 + 3) = 10$ を置かなくてはならないからです. このことを考えると 2, 3 の配置の仕方としては次のものとその回転対称なものに限られることがわかります.

B	●	2
3	●	A
●	1	●

次に 4 を置くことを考えると, 4 は 1 と同じ列に来てはいけませんでしたから, 上図の A,B の 2 箇所のみとなります. まず, A に 4 が来たとすると, 各列の数字の和が 15 となることから, 残りの数字の置き方は自然に決まって行き, 順調に最後まで行きそうに思えるのですが

7	6	2
3	8	9
5	1	9

が最終的に得られてしまい, 斜めの 1 列が $7+8+9=24$ となり, 15 を超えてしまいアウトです. 従って, 4 の置き方としては前図の B の場所しかなく, この場合に他の数字を並べていけば, 結局

4	9	2
3	5	7
8	1	6

が得られます. 従って, 可能な 9 つの数字の配置はこれか, あるいは, この回転対称なものに限られることがわかります.

第9回講義・演習 — 平成19年6月22日 —

※ 注意：今回は練習問題のプログラムと結果を紙に印刷したものを期日までに提出していただきます.

3.12 数値計算II：連立1次方程式の解法 — Gauss-Jordan 法 —

ここでは配列を用いた数値計算の一例として連立方程式の数値的な解法を取り上げます. 具体的には

$$\begin{aligned} a_{11}x + a_{12}y + a_{13}z &= b_1 \\ a_{21}x + a_{22}y + a_{23}z &= b_2 \\ a_{31}x + a_{32}y + a_{33}z &= b_3 \end{aligned}$$

のような方程式を解くことを考えます. ここで, 上記の方程式は次のように行列を用いて書き直せることに注意しましょう.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \tag{3.15}$$

ここで, 正方行列

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \tag{3.16}$$

と 1×3 行列 (ベクトル)

$$\boldsymbol{b} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \tag{3.17}$$

を合わせた次の**拡大係数行列** $(\boldsymbol{A} : \boldsymbol{b})$:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{pmatrix} \tag{3.18}$$

に対して基本変形を行い, 正方行列 $\boldsymbol{A}$ の部分を単位行列にすることができれば, そのときの $\boldsymbol{b}$ がその答えを与えるというのがこの解法 — **Gauss-Jordan 法 (掃き出し法)** — での基本的な指針となります.

例えば, 上の拡大係数行列 $(\boldsymbol{A} : \boldsymbol{b})$ に対し, 第 1 行を a_{11} で割ると (以下の操作で各行の成分を割る係数 a_{11} 等を**ピポット**と呼びます)

$$\begin{pmatrix} 1 & a_{12}/a_{11} & a_{13}/a_{11} & b_1/a_{11} \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{pmatrix} \tag{3.19}$$

が得られます. 次にこの行列 (3.19) の第 1 行を a_{21} 倍したものを第 2 行から引くことで

$$\begin{pmatrix} 1 & a_{12}/a_{11} & a_{13}/a_{11} & b_1/a_{11} \\ 0 & a_{22}-(a_{12}a_{21}/a_{11}) & a_{23}-(a_{13}a_{21}/a_{11}) & b_2-(b_1a_{21}/a_{11}) \\ a_{31} & a_{32} & a_{33} & b_3 \end{pmatrix} \tag{3.20}$$

となります. 最後にこの行列 (3.20) の第 1 行を a_{31} 倍したものを第 3 行から引くことで

$$\begin{pmatrix} 1 & a_{12}/a_{11} & a_{13}/a_{11} & b_1/a_{11} \\ 0 & a_{22}-(a_{12}a_{21}/a_{11}) & a_{23}-(a_{13}a_{21}/a_{11}) & b_2-(b_1a_{21}/a_{11}) \\ 0 & a_{32}-(a_{12}a_{31}/a_{11}) & a_{33}-(a_{13}a_{31}/a_{11}) & b_3-(b_1a_{31}/a_{11}) \end{pmatrix} \tag{3.21}$$

が得られることになり, 無事に第 1 列に $1, 0, 0$ が現れることになるわけです.

従って, 正方行列 $\boldsymbol{A}$ を単位行列にするためには, 行列 (3.21) においてピポットを $\{a_{22}-(a_{12}a_{21}/a_{11})\}$ に選び, これで第 2 行を割り, ... という操作を繰り返していけばよいことになります.

この操作によって, 上記の拡大係数行列の右端に現れるベクトル $\boldsymbol{b}$ は $\boldsymbol{b} \to \boldsymbol{b}' \to \cdots$ のように変化していきますが, 正方行列 $\boldsymbol{A}$ が単位行列になった時点での $\boldsymbol{b}$ の値がこの連立方程式の解を与えることになるわけです.

練習問題 19.1

教科書に与えられた連立方程式:

$$\begin{aligned} 2x+y-z &= 5 \\ -3x+3y+2z &= 1 \\ x-2y-2z &= -1 \end{aligned}$$

に対し, 上記に説明した操作を繰り返して解を求めよ. (※ これはプログラミングをするのではなく, 紙に鉛筆で計算すること).

練習問題 19.2

次のプログラムをファイルに書き込み, コンパイル・実行することで動作を確かめよ. (※ 教科書 100 ページの図 6.7 に載っている PAD も合わせて参照すること).

```
#include<stdio.h>
#define IMAX 3
#define JMAX 4

double array[IMAX][JMAX]={
  { 2,1,-1,5 },
  { -3,3,2,1 },
  { 1,-2,-2,-1 },
};

void PrintMat(void){
  int i,j;
  for(i=0; i<IMAX; i++){
```

```
    for(j=0; j<JMAX; j++){
      printf("%5.2f ",array[i][j]);
    }
    putchar('\n');
  }
  putchar('\n');
}

void SweepOut(void){
  int i,j,axis;
  double pivot,aik;

 for(axis=0; axis<IMAX; axis++){
    pivot=array[axis][axis];

    for(j=axis;j<JMAX;j++){
      array[axis][j]/=pivot;
    }

    for(i=0;i<IMAX;i++){
      if(i!=axis){
aik=array[i][axis];
for(j=0; j<JMAX;j++){
  array[i][j]-=array[axis][j]*aik;
}
      }
    }
  }
}

main()
{
  PrintMat();
  SweepOut();
  PrintMat();
}
```

3.13　リダイレクション

Linux 上で動作する C 言語プログラミングでは, プログラミング中に現れた scanf 関数, printf 関数に対し, リダイレクションという操作を行うことで, キーボード入力をファイルからの入力へ, ディスプレイへの出力をファイルへの出力として与えることができます. 例えば, input.dat というファイルに書かれた数字の列をプログラム test.c に現れる scanf 関数の入力として与えたい場合には, test.c をコンパイルして得られる実行形式 a.out に対して

```
./a.out < input.dat
```

とすればよいことになります.

逆に test.c の中に現れる printf 関数の出力を output.dat に出力したい場合には

```
./a.out > output.dat
```

とします. この両者を組み合わせる, すなわち, input.dat というファイルに書かれた数字の列をプログラム test.c に現れる scanf 関数の入力として与え, かつ, 逆に test.c の中に現れる printf 関数の出力を output.dat に出力したい場合には

```
./a.out < input.dat > output.dat
```

とすることになります. 以上をふまえて次の練習問題をやっていただきます.

練習問題 20.1

1 から 9 までの数字を書き込んだ input.dat というファイルを予め用意しておき, これら数字の和を計算し,

```
sum=*****
```

という形式の出力を output.dat に出力させるようなプログラムを作成し, 実行結果を確かめよ.

3.14　ファイル操作

前節でリダイレクションによるファイルからのデータの読み込みとファイルへの書き込みを学びましたが, ここでは, より柔軟なファイルの操作法を学びます. また, データの関連性を視覚化するものとして, gnuplot と呼ばれるグラフ作成ツールの使い方についても学習します.

3.14.1　ファイル操作の基本

以下に ASCII ファイルの**読み込み**と**書き込み**の典型的なプログラムを載せておきます. 教科書の第 8 章を合わせて確認してください.

```
/* ファイルの読み込みと書き込みの典型例 */
#include<stdio.h>
#include<stdlib.h>

main()
{
  FILE *pt,*pt2;
  double data, sum=0;
  int n;
        /* データを ASCII ファイル ascii.txt から読み込む */
       if((pt=fopen("ascii.txt", "rt")) != NULL){
     /* データを一つ一つ読み込み, その総和を算出 */
  while(fscanf(pt,"%lf", &data) != EOF){
       sum += data;
          n++;
```

```
  }
fclose(pt);
/* ascii.txt のデータ数，平均値を算出し，mean.txt に書き込む */
if((pt2=fopen("mean.txt", "wt")) != NULL){
fprintf(pt2, "n = %d \n sum = %lf", n, sum);
           }
fclose(pt2);
}
```

3.14.2 gnuplot によるグラフの作成

数値データが数値計算により求まったら, 今度はそのデータをグラフとして見てみたいと思うのが人情というものです. ここでは gnuplot というフリーソフトを用いて, データをグラフ化 (可視化) する方法を簡単に見て行くことにします.

gnuplot の起動とデータのプロット

コマンドラインから

```
iecsv{your_account}2% gnuplot [リターン]
```

としてみましょう. すると gnuplot が立ち上がり

gnuplot >

という入力待ちの状態になります. そこで例えば

```
0.100000 1.000000
0.500000 1.000000
1.000000 0.999282
1.500000 0.986609
.................
.................
```

という内容のデータファイル test.dat に対して, 第 1 列を x 軸に, 第 2 列を y 軸として描きたいのであれば

gnuplot > plot “test.dat” [リターン]

とすれば, test.dat が描写されます. 「線」で描きたい場合には

gnuplot > plot “test.dat” with line [リターン]

とすればよく, gnuplot を終了したい場合には

gnuplot > quit [リターン]

とします. データが

```
0.100000 1.000000 -4.000000
0.500000 1.000000 -4.000000
1.000000 0.999282 -3.994238
1.500000 0.986609 -3.894168
...................
...................
```

のように 3 行に渡っていて, この第 1 行を x 軸, 第 3 行を y 軸に描きたいのであれば

gnuplot > plot “test.dat” using 1:3 with line [リターン]

とすれば OK です.

gnuplot で描いたグラフを印刷形式で保存する方法

gnuplot で描いた図をレポートや論文として印刷できる形式 (ポストスクリプト形式, PS ファイル) に変換するにはコマンドラインから

```
iecsv{your_account}3%  gnuplot [リターン]
```

として gnuplot に入り, そのあと

gnuplot > set term postscript [リターン]
gnuplot > set output “out.ps” [リターン]
gnuolot > plot “test.dat” [リターン]

とすれば, ここでは何も表示されないですが

gnuolot > quit [リターン]

として gnuplot を抜けて

```
iecsv{your_account}4% ls  [リターン]
```

で見てみると, 確に out.ps ができています. この out.ps をディスプレイ上で見るには ghostview を用います.

```
iecsv{your_account}5% ghostview out.ps [リターン]
```

また, プリントアウトしたければ

```
iecsv{your_account}6% lpr out.ps  [リターン]
```

とすれば OK です.

3.14.3　写像力学系とカオス

ここでは物理学, 数学, 生物学や経済学などの社会科学, 工学など非常に広範な分野で研究されているカオスについて, その性質や基礎的概念に簡単に触れる目的で写像力学系を用いた数値実験を行なうことにしてみましょう. そして gnuplot による「データの可視化」を実際に行なってみましょう.

写像力学系

カオスが他の運動と異なるものであるという意味で, カオスとは

比較的簡単な規則に支配された不規則な運動

とであると言われます. こう書いてしまうと, 何のことだかさっぱりわからないかもしれません. そこで, 実際にそのような運動を生じさせる規則 (力学系) をとりあげてこの言葉の意味を理解することから始めてみましょう. ここで, 力学系というと, 微分方程式としてのニュートンの運動方程式を連想されるかもしれませんが, ここでは簡単のため, 写像 (マップ) と呼ばれる次の繰り返し演算を扱うことにします.

$$x_{n+1} \quad = \quad L(x_n) \quad (n = 1, 2 \cdots) \tag{3.22}$$

これは初期値を x_1 とした数列 $x_1, x_2, \cdots, x_n, \cdots$ が規則 (3.22) に従って生成されるということを意味します. この n を「時間」と考えれば, この写像 (3.22) を一種の時間発展とみなすこともできます. もちろん, 自然界の現象の中で, このような単純な写像で記述されるものはほとんどないですが, その現象のある側面のみを注目して観察し, かつ運が良ければこの写像も, その現象の, その注目する側面に関しては良い近似を与える場合があります. 従って, 写像を勉強すること自体は決して無意味なことではないわけです.

さて, この写像を特徴つけるのは関数 $L(x)$ です. これは有限区間で定義されるものが望ましいでしょう. また, できることならば, 何らかの可変なパラメータを有し (例えば a としょう), そのパラメータ a を調節することにより, 単なる固定点への漸近挙動だけでなく, 周期運動, そしてカオスとそこから生まれる運動にバラエティを持たせられるものがよいわけです. そこで, まずは次のような関数を使った写像力学系を調べて行くことにします.

$$x_{n+1} = L(x_n; a) = a x_n (1 - x_n) \tag{3.23}$$

ここで, $a = 4$ の場合は**ロジスティック写像**と呼ばれる有名な写像です. さて, 実際に (3.23) がどのような数列を生成するかを確認するために, 次のような課題を行なってもらうことにしょう.

今週の提出課題

写像力学系 (3.23) より初期値を $x_1 = 0.1$ として数列 $x_1, x_2, \cdots, x_n$ (n は 100 程度でよい.) を求めるプログラムを作成し, 横軸を n 縦軸を x_n として, gnuplot でプロットし, その挙動を考察せよ. a の値は $a = 1.0, 3.5, 4.0$ の 3 つの場合について行なうこと.

(注)
得られた結果をグラフとして見る場合, 得られるデータをファイルに

```
1.000000 0.004670
1.000000 0.004649
1.000000 0.004627
1.000000 0.004606
1.000000 0.004584
```

```
1.000000 0.004563
1.000000 0.004542
1.000000 0.004522
1.000000 0.004501
1.000000 0.004481
.................
.........
```

のような形式で格納する必要がある.

※ 提出するのは得られたグラフをプリントアウトしたものと作成しプログラムです.
なお, プログラムファイル (テキストファイル) のプリンアウト方法は

```
iecsv{your_account}16%  a2ps prog.c | lpr [リターン]
```

ポストスクリプト・ファイルのプリントアウトは

```
iecsv{your_account}17% lpr sample.ps  [リターン]
```

です.

第10回講義・演習 — 平成19年7月6日 —

この講義: 計算機プログラミングIで学習する内容は, 「ポインタ」「構造体」が今年度から後期開講の計算機プログラミングIIにまわされた関係で, 前回で無事に全て終了しました. 従って, 「ポインタ」「構造体」を多用するような, よりC言語らしいプログラミング技法は後期になってから学習していくことになります. しかし, 現在の知識だけでもある程度のプログラムは書けるようになっているはずですので, 今回と次回 (7/20: 最終回) の2回にわたり, 今までの知識を使った応用問題をやってもらうことにします. 題材は自然界に数多く現れる**フラクタル**に関してです. 前回その使い方を学んだ gnuplot を用いて, そのフラクタル図形を描いてもらうことにします.

3.15 フラクタル図形の計算機による作成

下の図 3.12 を見てください. この図形は「フラクタル」と呼ばれる構造をもっています. この種の図形の特徴は, 部分的な構造の積み重ねで全体が構成され, 描かれている点です. 言い方を変えれば, **スケールを変えて図形を眺めても, 常に同じ図形として見える**ということです.

図 3.12 は私がフリーハンドで描いたものではなく, また, 木の影の写真でもありません. **これは簡単な**

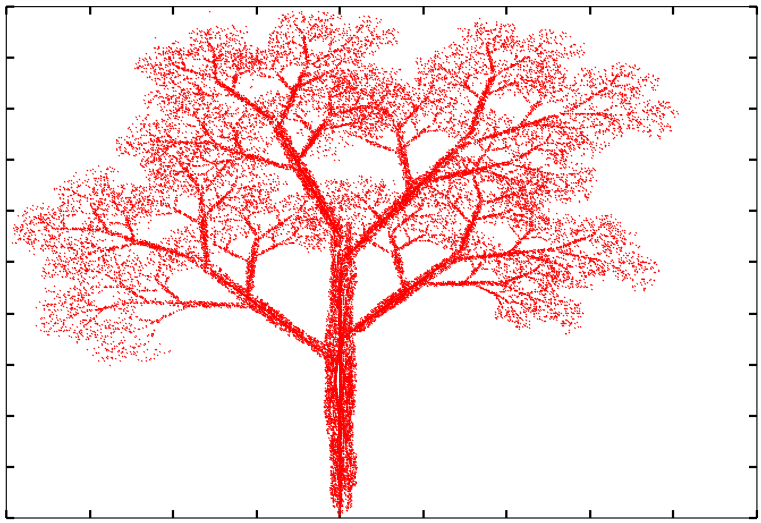

図 3.12: フラクタルな木.

ルールに基づき, コンピュータに描かせたものです. 今回の計算機プログラミングIの講義・演習では皆さんにこのようなフラクタル図形を実際に描いてもらうことにします.

3.15.1　複素数で記述される力学系 : 決定論的なフラクタル

まずは「決定論的」なルールに従って描かれるフラクタルについて見て行きましょう.

ロジスティック写像の複素数への拡張

まずは前回の講義で提出して頂いた課題であった**ロジスティック写像**を思い出してみましょう.

$$x_{n+1} \quad = \quad ax_n(1-x_n) \tag{3.24}$$

写像 (漸化式)(3.24) に含まれるパラメータ a を変えていくと $x_0 \to x_1 \to x_2 \to \cdots \to x_n \cdots$ という時系列は「固定点」「周期軌道」「カオス」というような様々な振る舞いを見せましたが, a の変化に対してその周期軌道の分岐は図 3.13 のようになります. 今回のこの講義では写像の変数 x を複素数 z に拡張した場合に

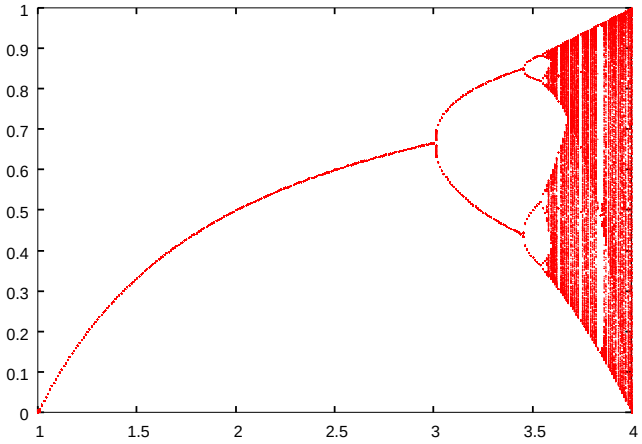

図 3.13: ロジスティック写像の分岐図

得られる複素数版のロジスティック写像からは何がわかるのかを調べてみましょう. 複素数 z の実部を x, 虚部を y で表せば, z は $z = x + iy$ と書けるので, これをロジスティック写像の式 : $z_{n+1} = az_n(1-z_n)$ に代入すれば

$$\begin{aligned} x_{n+1} + iy_{n+1} \quad &= \quad a(x_n + iy_n)(1 - x_n - iy_n) \\ &= \quad a(x_n - x_n^2 + y_n^2) + iay_n(1-2x_n) \end{aligned} \tag{3.25}$$

が得られます. ここでパラメータ a は $a = a_R + ia_I$ として複素数に選ぶこともできますが, まずは簡単のため実数であるとして話を進めましょう.

さて, (3.25) 式の両辺の実部, 虚部をそれぞれ等しいと置くことにより

$$x_{n+1} \quad = \quad a(x_n - x_n^2 + y_n^2) \tag{3.26}$$

$$y_{n+1} \quad = \quad ay_n(1-2x_n) \tag{3.27}$$

が得られます. この連立漸化式 (3.26)(3.27) をある初期条件 $\boldsymbol{x}_0 \equiv (x_0, y_0)$ からスタートさせると, この初期条件の選び方が良ければ連立漸化式は複素平面 x-y 内のある点へと収束するか, 有限範囲内での周期軌道へと収束していきます. しかし, 一方で初期条件の選び方がうまくなければこの連立漸化式は収束せず, $|\boldsymbol{x}_\infty| \equiv \sqrt{x_\infty^2 + y_\infty^2} \to \infty$ のように発散してしまいます. 従って, 全ての可能な初期条件はその初期条件か

ら出発した軌道が [発散してしまうもの] と [有限に留まるもの] との 2 種類に分類されることがわかります. つまり, 初期条件の作る集合 : $\{\boldsymbol{x}_0\}$ は

$$\begin{aligned} \{\boldsymbol{x}_0\}_\infty &\equiv \{\boldsymbol{x}_0|\ |\boldsymbol{x}_\infty| \to \infty\} \\ \{\boldsymbol{x}_0\}_{\rm finite} &\equiv \{\boldsymbol{x}_0|\ |\boldsymbol{x}_\infty| < \infty\} \end{aligned}$$

でそれぞれが定義される部分集合 $\{\boldsymbol{x}_0\}_\infty$ と $\{\boldsymbol{x}_0\}_{\rm finite}$ に分類されます.

連立漸化式 (3.26)(3.27) を数値的に解き, 複素平面 x-y 内にこの $\{\boldsymbol{x}_0\}_{\rm finite}$ をプロットしてできる図は, その作り方が上述のように極めて単純なのにも関わらず非常に複雑な形状を持ちます. それを図 3.14 に示すことにします. この集合 $\{\boldsymbol{x}_0\}_{\rm finite}$ のことを**ジュリア集合**と呼びます. 図 3.14 では $a = 3.3$ に選びました

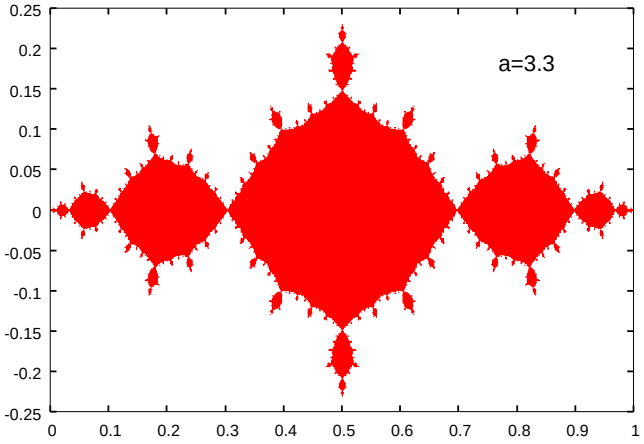

図 3.14: ジュリア集合. $a = 3.3$

が, この場合に得られるジュリア集合はフラクタルであり, 前述の自己相似性を有します. a の値をいろいろ変ええれば様々なジュリア集合が得られます. 図 3.15 にその中から数例を載せましょう.

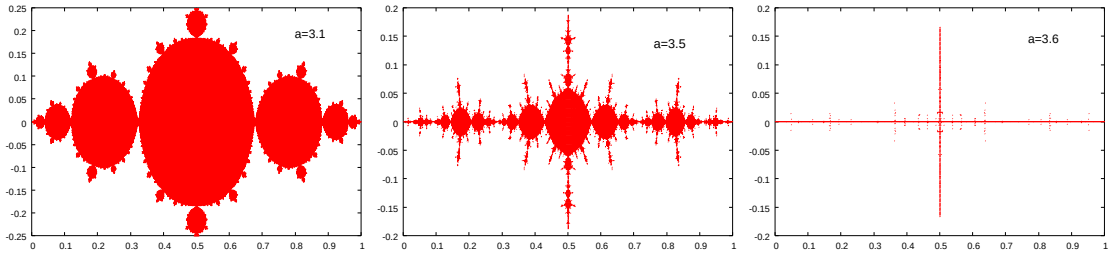

図 3.15: 左から $a = 3.1, 3.5$, 及び, $a = 3.6$ に対するジュリア集合.

複素力学系の数値解法 : ニュートン法の復習

具体的に $z_{n+1} = f(z_n)$ のタイプの漸化式の解は $z_{n+1} = z_n = z$ としたときの方程式 $f(z) - z \equiv g(z) = 0$ の解です. 従って, ジュリア集合を求める際にはこの方程式 $g(z) = 0$ の解 z が有限であるかどうかを判定すればよいことになります. このとき, $g(z) = 0$ の解法としては既に**ニュートン法**を学んでいますので, それがここで使えるわけです.

ニュートン法を簡単に復習しておきますと, 方程式 $g(z)=0$ の解の候補 $z=z_0$ で曲線 $y=g(z)$ に接する接線の方程式 :

$$y \quad = \quad g(z_0)+g^{'}(z_0)(z-z_0)$$

の z 軸との交点 :

$$z \equiv z_1 \quad = \quad z_0 - \frac{g(z_0)}{g^{'}(z_0)}$$

を解候補 z_0 の改良版とするもので, これをおし進めて n-番目の解候補と $n+1$ 番目の候補の間に成り立つ関係式 :

$$z_{n+1} \quad = \quad z_n - \frac{g(z_n)}{g^{'}(z_n)} \tag{3.28}$$

を z_n に関する漸化式とみて, この収束点をもって解きたい方程式 $g(z)=0$ の解とするものでした.

今の複素数版ロジスティック写像の場合には $g(z)=az(1-z)-z$ ですから, 反復式 (3.28) は

$$z_{n+1} \quad = \quad z_n - \frac{az_n(1-z_n)-z_n}{a-2az_n-1}$$

となりますが, 例によって $z_{n+1}=x_{n+1}+iy_{n+1}, z_n=x_n+iy_n$ をこれに代入し, x, y に関する漸化式に直してみると, やや煩雑ですけれど

$$x_{n+1} \quad = \quad x_n - \frac{(a-2ax_n-1)[(a-1)x_n-a(x_n^2+y_n^2)]}{(a-2ax_n-1)^2+4a^2y_n^2} \tag{3.29}$$

$$y_{n+1} \quad = \quad y_n - \frac{y_n(a-2ax_n-1)^2+2ay_n\{a(x_n-x_n^2+y_n^2)-x_n\}}{(a-2ax_n-1)^2+4a^2y_n^2} \tag{3.30}$$

が得られます. これら (3.29)(3.30) 式を様々な初期条件からスタートし, その解が有限範囲に収まるか否かをチェックして行けばよいことになります.

練習問題 22.1

gnuplot を用いてジュリア集合 ($a=3.3$) を描くプログラムを作成せよ.

3.15.2　マンデルブロ集合

前節ではロジスティック写像をコントロールするパラメータ a を実数として扱いましたが, 先に述べたようにこれをも複素数に拡張することができます. $a=a_R+ia_I$ として (3.25) 式に代入し, 実部と虚部にわけた反復式を書き出してみると

$$x_{n+1} \quad = \quad a_R(x_n-x_n^2+y_n^2)-a_Iy_n(1-2x_n) \tag{3.31}$$

$$y_{n+1} \quad = \quad a_Ry_n(1-2x_n)+a_I(x_n-x_n^2+y_n^2) \tag{3.32}$$

が得られます. ジュリア集合を求めたときには, a の値を固定し, 反復式 (3.31)(3.32) が有限の値に収まるような初期条件の集合を複素平面内にプロットしました. ここでは逆に, 初期条件 $\{x_0, y_0\}$ が一つ与えられた場合, a_R, a_I の値を様々変えたときに, 反復式 (3.31)(3.32) の解が有限の値を持つような集合 $\{a_R, a_I\}$ を x-y 平面にプロットしたならばどのような図形が得られるのかを考えてみましょう. 図 3.16 に $x_0=0.5, y=0$ を初期条件に選んだ場合の結果を載せます. このような図形のことを**マンデルブロ集合**と呼びます.

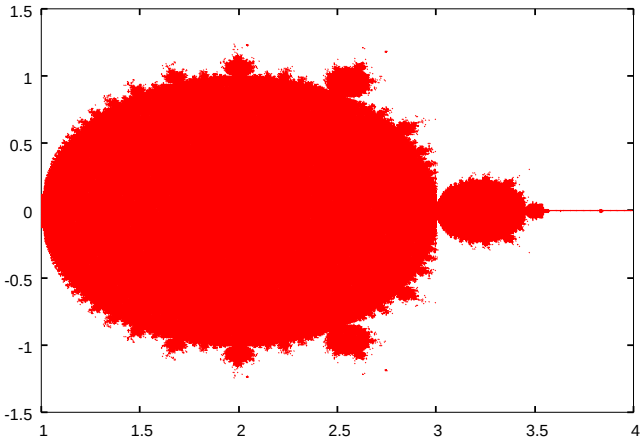

図 3.16: マンデルブロ集合. $x_0 = 0.5, y_0 = 0$ と初期条件を選んである.

練習問題 22.2

$x_0 = 0.5, y_0 = 0$ を初期条件に選んだ場合のマンデルブロ集合を gnuplot を用いて描くプログラムを作成せよ.

3.15.3　確率的フラクタル

ここからは確率を用いて描かれるフラクタルについて見ていきます. コンピュータ上で確率を扱うわけであるから, 擬似乱数を用いなければなりません. そこでまずはこの実験で用いる一様乱数を生成する関数を見ていきます.

一様乱数生成関数

ここからは $[0, 1]$ 間の一様乱数を用いますがここでは予めそのような乱数を発生させる関数を配布し, それを使ってもらうことにします. 次のアドレスを netscape で開いてください.

```
http://chaosweb.complex.eng.hokudai.ac.jp/~j_inoue/PROG2007/PROG2007.html
```

次にここにある randomnumber.c というプログラムファイルをダウンロードします. うまくダウンロードができたなら, このプログラムファイルを Xemacs を用いて開き, この中の乱数の「種」SEED の値を幾つか変えて乱数の系列を確認してみてください. ここで, 生成された乱数は rand.dat というファイルに格納されるようになっています. つまり, コマンドラインから

```
% gcc randnumber.c -lm
% a.out
% more read.dat
```

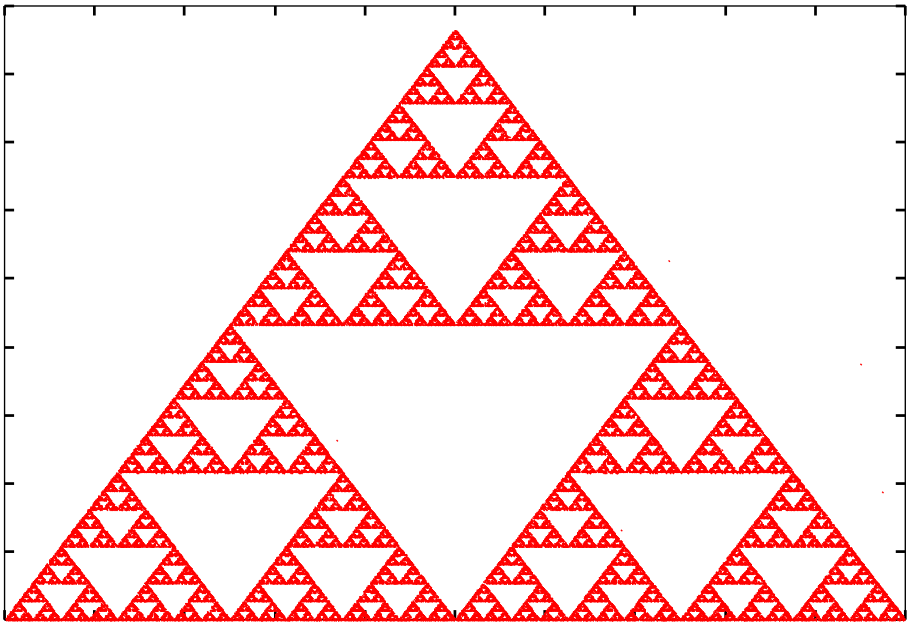

図 3.17: シェルピンスキー・ガスケット.

とすれば, 実際に $[0,1]$ の乱数が生成されたかどうかがわかるはずです. まずは各自がこれを確認してから次に進んでください.

シェルピンスキー・ガスケット

図 3.17 は**シェルピンスキー・ガスケット (Sierpinsky's gasket)** と呼ばれる典型的なフラクタル図形の一つです[10] . この図形は次に示すような非常に簡単なルールを用いて生成させることができます.

シェルピンスキー・ガスケット作成のアルゴリズム

(1) 図 3.18 のように頂点を $(X_1, Y_1) = (0,0)$ (点 O), $(X_2, Y_2) = (2l, 0)$ (点 A), $(X_3, Y_3) = (l, \sqrt{3}l)$ (点 B) とする正三角形を考える.
(2) この正三角形内部の任意の点 $\mathrm{P}^{(0)} = (x_0, y_0)$ を選ぶ.
(3) 数 0,1,2 を**ランダム**に選び
- $0 \Rightarrow$ O と $\mathrm{P}^{(0)}$ を結ぶ線分の中点に点 $\mathrm{P}^{(1)} = (x_1, y_1)$ を置く.
- $1 \Rightarrow$ A と $\mathrm{P}^{(0)}$ を結ぶ線分の中点に点 $\mathrm{P}^{(1)} = (x_1, y_1)$ を置く.
- $2 \Rightarrow$ B と $\mathrm{P}^{(0)}$ を結ぶ線分の中点に点 $\mathrm{P}^{(1)} = (x_1, y_1)$ を置く.

(4) プロセス (3) を十分多数回繰り返す.

上記のアルゴリズムにより, 点列

$$\mathrm{P}^{(0)}(x_0, y_0) \to \mathrm{P}^{(1)}(x_1, y_1) \to \cdots \mathrm{P}^{(n)}(x_n, y_n) \to \cdots$$

[10] ガスケット (gasket) とは「詰め物」の意味である.

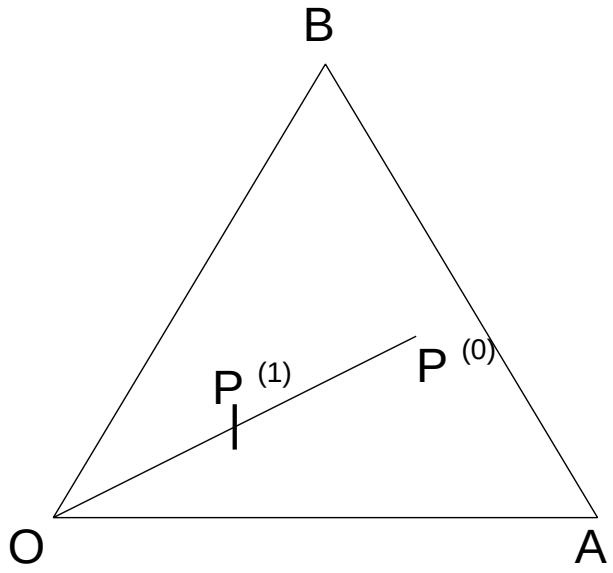

図 3.18: O が選ばれた場合には $P^{(0)}$ と O の中点に $P^{(1)}$ をおく.

を 2 次元平面内にプロットする. 具体的には

x_0 y_0

x_1 y_1

x_2 y_2

x_3 y_3

.....

という形式でデータファイル (例えば gasket.dat) に格納する.

練習問題 22.3

シェルピンスキー・ガスケットを gnuplot により描くプログラムを作成せよ. (※ 慣れないうちは, **シェルピンスキー・ガスケット作成のアルゴリズム**の手順を既に学んだ PAD 図に書き出してみると良い)

[**プログラム作成上のヒント**]

(3) の条件分岐の部分は r←$[0,1]$ 間の一様乱数 として

$$r \in [0, 1/3] \rightarrow \text{点 O を選ぶ.}$$
$$r \in [1/3, 2/3] \rightarrow \text{点 A を選ぶ.}$$
$$r \in [2/3, 1] \rightarrow \text{点 B を選ぶ.}$$

を if 文で表現すればよい. また, if 文を用いなくても.

```
rr = (int)(r*3.0);
```

とすると rr には $0, 1, 2$ がそれぞれ $1/3$ の確率で現れることから, switch 文を用いて

```
switch(rr){
case 0: (点 O を選ぶ);
break;
case 1: (点 A を選ぶ);
break;
case 2: (点 B を選ぶ);
break;
}
```

としてもよい.

(注) 今回のレポート提出に関して

今回の 3 つの練習問題は全てレポートにて提出して頂きます. 提出期限はこの講義の**最終回 (7/20) に回収します**ので, それまでに提出してください. なお, 提出すべきものは各問題に対し, プログラムソース・コードと描いたフラクタル図形をプリントアウトしたものの 2 点です.

次回 (7/20) 講義は最終回であり, この講義全体を通じての簡単な復習・確認をし, **期末試験に関する重要な伝達事項**がありますので, 必ず出席してください.

図 3.19: 7 月 13 日は井上イタリア出張のため休講.

中間試験 — 平成19年6月1日 —

注意: 問題は1問. 答案用紙裏面も用いてよいが, その場合には「裏面に続く」と明記する. 試験開始後30分経過するまでは退出できないので注意すること.

問題

計算機ディスプレイ上に

```
x=
```

と表示された後に実数 x の値をキーボード入力すると

```
--- メニュー: 番号を選んでください ---
    1: Sin(x) 2: Exp(x) 3: Root(x)
```

と表示され, この際に 1 ～ 3 の数字のいずれか 1 つをキーボードより入力すると該当する数学関数の値を

```
******
```

の形式でディスプレイ上に結果を表示するプログラムを作成せよ (※ ここで Root(x) は x の平方根).

ただし, プログラミングに際して次の 3 点に注意すること.

- math.h に定義された sin(),exp(), sqrt() を 使ってはならない.
- 関数 Sin(x),Exp(x) は**再帰的関数定義**により自作し, メイン関数内で用いること.
- 関数 Root(x) は**ニュートン法**を利用して自作し, メイン関数内で用いること.

作成するプログラムには, 必要に応じて随時コメント行を入れ, 説明を加えること.

※ 終了後, 解答例を 1 部ずつ持ち帰ること. 次回 (6/15) 答案用紙を返却し, 簡単な解説をします.

中間試験解答例 出題・採点 井上純一

注意: 下記プログラム以外でも問題の要求を満たし, 正しい結果を出力するものは全て正解とする.

```
/*************************************************************/
/*  計算機プログラミングＩ A クラス 中間試験　解答例         */
/*           01/06/2007       解答例作成　井上純一           */
/*************************************************************/
#include<stdio.h>
#include<math.h>
#define nmax 10   /* 展開項数の上限を単純マクロで定義 */
#define imax 100  /* ニュートン法の繰り返し上限を単純マクロで定義 */

double Sin(double x); /* 自作正弦関数のプロトタイプ宣言 */
double Exp(double x); /* 自作指数関数のプロトタイプ宣言 */
double Root(double x); /* 自作平方根を返す関数のプロトタイプ宣言 */

/*************************************************************/
/*                       メイン関数                          */
/*************************************************************/
main()
{
  int i;
  double x,y;

  printf("x=");
  scanf("%lf",&x);
  printf("--- メニュー: 番号を選んでください --- \n");
  printf("1: Sin(x) 2: Exp(x) 3: Root(x) \n");

  scanf("%d",&i);
/* swicth case 文で i=1,2,3 の値によって条件を分岐させて各関数の出力結果を表示 */
  switch(i){
  case 1:
    y = Sin(x);
    printf("%lf\n",y);
    break;
  case 2:
    y = Exp(x);
    printf("%lf\n",y);
```

```
    break;
  case 3:
    y = Root(x);
    printf("%lf\n",y);
    break;
  }
}
/*************************************************/
/*            正弦関数の再帰的関数定義           */
/*************************************************/
double sine(double x, int n)
{
  if(n==0){
    return (x);
  }else{
    return (-sine(x,n-1)*x*x/(2*n*(2*n+1)));
  }
}
/*************************************************/
/*            自作正弦関数の関数本体             */
/*************************************************/
double Sin(double x)
{
  int n;
  double y;
  for(n=0,y=0.0; n<=nmax; n++){
    y = y + sine(x,n);
  }
  return (y);
}
/*************************************************/
/*           指数関数の再帰的関数定義            */
/*************************************************/
double expo(double x, int n)
{
  if(n==0){
    return (1.0);
  }else{
    return (expo(x,n-1)*x/n);
  }
}
/*************************************************/
/*           自作指数関数の関数本体              */
/*************************************************/
double Exp(double x)
```

```
{
  int n;
  double y;

  for(n=0,y=0.0; n<=nmax; n++){
    y = y + expo(x,n);
  }
  return (y);
}

/*********************************************************/
/*    ニュートン法を用いた x のルートを出力する関数本体  */
/*********************************************************/
double Root(double x)
{
  int i;
  double y,z;

  for(i=1,y=1.1,z=0.0;i<=imax; i++){
    z = y - 0.5*(y*y-x)/y;
    y = z - 0.5*(z*z-x)/z;
    if(fabs(y-z)<1.0e-6) return (y);
  }
}
```

中間試験総評 文責 井上純一

受験者数: 71, 欠席者数: 4.

6 月 1 日に実施した中間試験の採点をしてみました. 第 7 回講義まで学習した内容: 変数の宣言, 繰り返し, 条件分岐, 関数等 (今回「配列」は含まず) の全てを使って一つのまとまったプログラムを作ってもらう問題でしたが, 出題側の指示を全て満たした完全な正解が全体の約 1 割, 問題の要求を満たしていなくても, ともかく, 3 つの関数全てに正しい出力を与えるプログラムが全体の約 3 割, 3 つの自作数学関数のうちどれかが間違っている, あるいは完成していないものがやはり 3 割. 残りの 3 割の答案がコンパイルの段階で何らかのエラーが出るもので, 極少数ですが, 持ち込み可にもかかわらず, ほとんどプログラムが書けていない答案もありました. そんなわけで, 全体的にみて決して良い出来とは言えない結果でした. 今までは各項目ごとに練習問題を解いてもらっているので, 各項目は理解しているというものの, それらを組み合わせて一つのプログラムを書くという訓練はしていなかったので, そのあたりが現段階では難しいと言えば難しかったのかもしれません.

いずれにしても, ここでの誤りをいかに今後に活かすか, がとても重要ですので, 以下に全てではないですが, 皆さんの答案に散見された間違いを列挙しておきます. 各自が再度自分の答案 (※ 青ペンで添削しておきました. 余談ですが私は採点業務に普段「赤色」ではなく, 「青色」か「緑色」を使います. 「『青』や『緑』なら同じ×でも『目に優しい』」) と照らし合わせて確認し, 今後の学習上の参考にしてください.

皆さんの答案に見られたいくつかの問題点

- scanf() 関数で, 変数 x にキーボード入力する際の

  ```
  scanf("%lf",&x);
  ```

  の & を忘れたもの.

- C 言語で等式は “=” ではなく “==” ですが, これを書き間違えたもの多数. これは紙に書く, キーボード入力するにかかわらず, 頻繁におこす間違いで, 実行結果にも影響を与える場合がありますので, 要注意です.

- 指数関数 Exp(x) は問題の指示に従わず, かつ, math.h に定義された指数関数 exp(x) を使わなくても, 自然対数の底を例えば $e = 2.71$ などと置き, 変数 x の a 乗を与える数学的関数: pow(x,a) を用いて

  ```
  return (pow(2.71,x));
  ```

  のように値を返す関数を作ることもできます (もちろん, わざわざ関数にしなくても, メイン関数内で pow(2.71,x) を直接的に表示させることも可能). しかし, この math.h に定義された関数 pow(x,a) を使うためにはプログラム先頭で

  ```
  #include<math.h>
  ```

と, このヘッダファイルをインクルードする必要があります. これを忘れると, pow(x,a) が未定義としてコンパイル・エラーが出ます.

- 指数関数 Exp(x) の作り方としてはこの他に

```
double Exp(double x)
{
   int i;
   double y;
   for(i=1,y=2.71; i<=x; i++){
    y = y*2.71;
    }
   return (y);
}
```

としたものが複数ありました. 気持ちはわかりますが, まず, 上の関数は Exp(x) の引数 x が整数の場合しか使えません. 期待通りの出力結果を与えないという意味で, コンパイルは通っても実行時にエラーが出ます.

- Sin(x) の作り方とその使い方に関して

```
double sin(double x, int j)
{
     if(j==1){
     return (x);
     }else{
     return (pow(-1,j-1)*pow(x,2j-1)/fact(2j-1) + sin(x,j-1));
     }
}
```

のように指数関数の展開と sin(x, j) を再帰的に用いて作成しても OK です. ただし, j = 1 の場合の関数値を予め与えておかないとメイン関数の中で使う際に実行時エラー (セグメンテーション違反) が出ますから注意が必要です.

- 同じ正弦関数を作るにも for ループで数え上げをしていく直前での変数の初期化失敗が致命的な間違いを起こしているような答案も複数ありました. 例えば

```
double sin(double x)
{
     int i, n=100;
     double ans=0,term=0,mx2=-x*x;
     for(i=1; i<n; i++){
        term *= mx2/(2*i*(2*i-1));
        ans += term;
        }
     return (ans);
}
```

としてしまうと, term はゼロで初期化されてますから, これ以降, for ループ中の term *= ... で何を何回掛け合わせようが値はゼロです. for ループを用いた繰り返し演算を行う際には**面倒でも具体的にはじめの数項を紙に書き出してチェックしてみる**ことがこの手の実行時エラーを少なくする方法の一つです.

- 関数の名前の付け方に関し, sin(x) など既に math.h に定義されたものを使う場合, コンパイル時に名前の重複があったとしてエラーが出ます. プログラム中に他の数学関数を一切使わないのであれば, math.h のインクルードを外すことで, この種のエラーは無くなりますが, それは現実的選択ではないので, 解答例に示したように先頭の文字を Sin(x) のように大文字にするなど, 名前の付け方を工夫する必要があります. また, 関数名の途中にスペースを入れて Sin func(x) などとすることもできません. コンパイル時にエラーがでます.

- ニュートン法の収束性チェックの条件文で

```
if(fabs(x-y) < EPS){
return (y);
}
```

などとする際に変数 EPS をマクロ定義しないで用いた答案も見られました. プログラムの先頭で

```
#define EPS 1.0e-10
```

などと定義すべきです. あるいは, 今後頻繁に使うのであれば, ヘッダファイル (constant.h など名づける) を作成し, そこにまとめてこの手の定数を定義して書き込んでおくと良いと思います. その際には既に学んだように

```
#include "constant.h"
```

のように"..."で囲ってインクルードすることを忘れないように.

- こちらの全く想定していなかったもので, よくあったものとして, メイン関数の中で別の関数を定義して用いているような答案がありました. 例えば整数の階乗を表示させるプログラムで書けば

```
#include<stdio.h>
#include<math.h>
double fact1(int n);
main()
{
        int a;
        printf("a=");
        scanf("%d",&a);
/* メイン関数の中で別の関数本体を定義 */
 double fact1(int n)
 {
       int i;
       double ans=1.0;
        for(i=1; i<=n; i++){
        ans *= i;
```

```
        }
        return (ans);
}
        printf("%d!=%f (def.1)\n", a,fact1(a));
}
```

のようにメイン関数の中で関数 fact1 を定義し, この値を出力させて使っている答案です. 意外なことにこのプログラムはコンパイルも通り, 正しい答えを出力しますが, プログラム構造上の観点からみると, fact1 を関数にした意味がほとんど無く, 非常に読みにくい (つまり, きれいでない) プログラムになってしまいます.

- 関数を全く使わないでメイン関数の中に全てを書き込んでしまっているような答案もかなりの数ありました (これは今まで私が教えた経験上, 高学年へと進んだ際の「学生実験」等でプログラムを書いてもらう際にも見受けられるので注意してください. おそらく, 気をつけないと, 知らず知らずのうちに全てをメイン関数の中に書き込んでプログラムを作るというのが, ある種, 自分のプログラミングにおける「スタイル」になってしまうのだと思う). 上の例で書けば

```
#include<stdio.h>
#include<math.h>
main()
{
        int a;
        printf("a=");
        scanf("%d",&a);

       double ans=1.0;
        for(i=1; i<=a; i++){
        ans *= i;
        }
        printf("%d!=%f (def.1)\n", a,ans);
}
```

とするようなものです. この例程度のプログラムならば, これでも何も問題ではないのですが, 今後, 様々な関数を用いて複数の処理を行わせるような場合, この手の書き方でメイン関数を大きくしていくと, 見通しが悪く, バグを見つけることが非常に困難なプログラムになってしまいます. おそらく, このようなプログラムを書いた方の多くは関数の作り方と使い方をまだ理解していないものと思われますので, 今のうちに再度確認しておいてください.

- 関数プロトタイプ宣言の場所をメイン関数の直後

```
main(){
double Sin(double x); /* 自作正弦関数のプロトタイプ宣言 */
double Exp(double x); /* 自作指数関数のプロトタイプ宣言 */
double Root(double x); /* 自作平方根を返す関数のプロトタイプ宣言 */
.........
.......
......
```

にしているものあり. コンパイルエラーが出ます.

- 一番多かったプログラム上の問題を含む答案としては, 間違いではないですが, 指示通りに「再帰的関数定義」を使っていないものです. 再帰的関数定義とは自分自身を return 文の中に含むもので, 次のような構造を持っています.

```
double sine(double x,int n)
{
  ...........
  ........
  .....
  return ((何かの式)*sine(x,n-1));
 }
```

こういう再帰構造を使って関数を定義し, それを用いて正弦関数を自作し, その値を出力させることがここでの課題でした. 従って, 厳密な採点基準をとれば次のプログラムは再帰的な関数定義を使っていないので「不正解」です.

```
double Sin(double a)
{
      int i;
      double t,s=0;
      for(i=1;i<=10;i++){
       t = pow(-1,i-1)*pow(a,2*i-1)/fact(2*i-1);
       s += t;
       }
      return (s);
}
```

(ここで, fact() は関数マクロ, あるいは外部関数として定義されていると思ってください). プログラムの実行結果だけを見ると間違いではありませんが, 問題の指示とは明らかに違います. 「再帰的関数」の「再帰的」という意味を上の例でもう一度確認しておいてください.

- せっかく関数自体は書けているのに, return 文を忘れてしまったばっかりに値を返さない関数を書いてしまっているものも見受けられました. メイン関数の中で処理だけを実行するような値を返さない関数も作れますが, その場合には void 型関数として宣言したことを確認しておいてください.

- 再帰的な関数定義は使ってませんが (その意味で完全な正解ではないですが), 次のようにして指数関数を作ることもできます. この手の答案を書いてくださった方も居ました.

```
double Exp(double x,int n)
{
      int i;
      double term, term2,sum;
      for(i=1,sum=1,term=1,term2=1,i<=n; i++){
      term = i*term;
      term2 = x*term2;
```

```
        sum = sum + term2/term;
        }
        return (sum);
    }
```

この他にも細かくみると printf 関数を問題の指示通りに使っていないもの, $\sqrt{x}$ の値を計算する関数を作る際に指示に従ったニュートン法ではなく, 2 分法で作成したものなどありました. 実際に紙に鉛筆で書いたプログラムでは細かな部分が間違っている場合があり, そうした自分のミスに気づかないこともあると思います. そこで, 各自が返却された自分の答案を Xemacs でファイルに書き込み, コンパイル・実行することで自分の答案の問題点を確認しておいてください.

期末試験 — 平成19年8月6日 —

注意事項 〜はじめる前にお読みください 〜

- 解答用紙には氏名・学籍番号を楷書で丁寧に書くこと.
- **問題 1** 〜 **問題 4** の計 4 問の全てに答えること. 制限時間 90 分.
- 試験開始 30 分間は退出できない. 試験開始から 30 分経過した後の入室を認めない.
- 不明な点がある場合には黙って挙手する. ※ 出題内容によっては答えられないものもある.
- 解答用紙の裏面を使っても良い. その場合には「裏につづく」と記すこと.

問題1

下記のようなディレクトリ構造を持つ Linux システムに関して問い (1)〜(4) に答えよ.

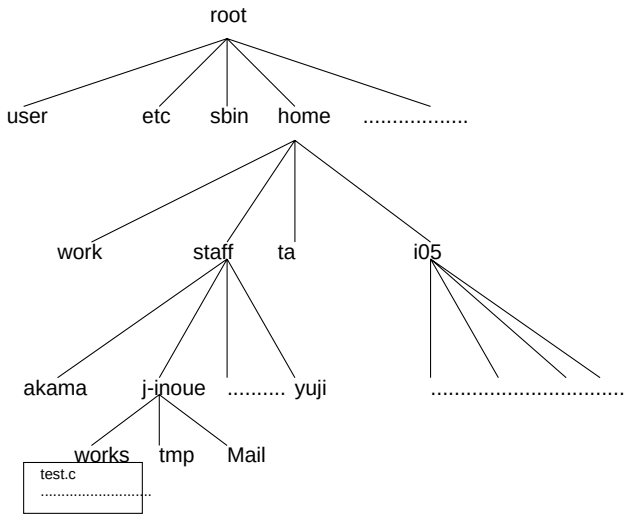

図 3.20: ここで問題とする Linux システムのディレクトリ構造.

(1) j-inoue の直下にあるディレクトリ works に置かれているファイル test.c を絶対パス指定で書け.
(2) 自分が j-inoue にいる場合, (1) のファイルを相対パス指定で書け.
(3) 自分のホームディレクトリが j-inoue であり, カレントディレクトリはホームディレクトリの直下の works としょう. このとき, ディレクトリ akama にある sample.dat というテキストファイルを自分の

カレントディレクトリにコピーするためのコマンドを書け.

(4) (3) での操作で, このファイル sample.dat へのアクセスがシステムによって拒否された場合, どのような理由が考えられるか述べよ. また, この問題を回避するために sample.dat の所有者が行うべき操作コマンドを書け.

問題2

下記の不完全なプログラムに関して

(1) コメント行に記された (A) の操作を行わせる関数を書け.

(2) プロトタイプ宣言に記された形式の引数と出力型を持つ**「2 つの整数のうち大きいほうの数を返す」**という内容の関数本体を書け. ただし, 両者が等しい場合にはその数を返すようにすること.

```
#include<stdio.h>
#include<math.h>
int max(int a, int b);  /* プロトタイプ宣言 */
main()
{
         int x,y,c;
        /* (A) x,y の値をキーボード入力する */
         c = max(x,y);
         printf("max(%d, %d) = %d\n", x,y,c);
 }
```

(3) (1)(2) で扱ったプログラムの max(x,y) を関数ではなく, **引数付きマクロ**として書き直せ.

問題3

(1) 以下で $n!$ の値を再帰的関数定義で求め, その値を返す関数 fact2 を作成し, プログラムを完成させよ.

```
#include<stdio.h>
#include<math.h>
int fact2(int n);

main()
{
        int a;
        printf("a=");
        scanf("%d",&a);
        printf("%d\n", fact2(a));
}

/* 再帰的関数定義で n! を返す関数をここに作る */
```

(2) 下記の不完全なプログラムが**「キーボード入力された 5 つの異なる実数のうちの最小のものを表示する」**ようにプログラム中の (A) ???????????? の部分を埋め, このプログラムを完成させよ.

```
#include<stdio.h>
#include<math.h>

main()
{
        double a[5],min;
        int i;
        for(i=0; i<5; i++)
        {
              scanf("%lf",&a[i]);
        }
        min=a[0];

        (A) ??????????????????????????

        printf("min=%lf\n",min);
   }
```

(3) (2) で作成したプログラムを実行し (実行形式は a.out), 結果を result.dat にリダイレクションするためのコマンドラインを書け.

問題 4

以下のアルゴリズムを C 言語でコーディングせよ.

シェルピンスキー・ガスケット作成のアルゴリズム

(1) 頂点を $(X_1, Y_1) = (0, 0)$ (点 O), $(X_2, Y_2) = (2l, 0)$ (点 A), $(X_3, Y_3) = (l, \sqrt{3}l)$ (点 B) とする正三角形を考える.

(2) この正三角形内部の任意の点 $\mathrm{P}^{(0)} = (x_0, y_0)$ を選ぶ.

(3) 数 0,1,2 を**ランダム**に選び

- $0 \Rightarrow$ O と $\mathrm{P}^{(0)}$ を結ぶ線分の中点に点 $\mathrm{P}^{(1)} = (x_1, y_1)$ を置く.
- $1 \Rightarrow$ A と $\mathrm{P}^{(0)}$ を結ぶ線分の中点に点 $\mathrm{P}^{(1)} = (x_1, y_1)$ を置く.
- $2 \Rightarrow$ B と $\mathrm{P}^{(0)}$ を結ぶ線分の中点に点 $\mathrm{P}^{(1)} = (x_1, y_1)$ を置く.

(4) プロセス (3) を十分多数回繰り返す.

ただし, プログラムで用いる $[0, 1]$ 擬似一様乱数は関数 RAND() を用いること (自分で作らなくて良い). この関数はメイン関数の中で呼ばれる毎に異なる乱数を生成し, その値を返すものとする. また, x_0, y_0, l, 及び, 繰り返し回数などは各自が適当に決めてよい. その際, 置かれる点を result.dat という名前のファイルに書き込むようにすること.

期末試験解答例

出題・採点 井上純一

注意

- 解答例以外にも正解とする, あるいは部分点を与える場合がある.
- 自分の成績を知りたい者は 8 月 24 日以降に情報科学研究科棟 8-13 まで来ること.

問題 1

(1) `/home/staff/j-inoue/works/test.c`

(2) `./works/test.c`

(3) `cp /home/staff/akama/sample.dat ~/works/`

(3) ファイルの所有権が他人に対して読み込み不可になっている. これを解消するには

```
chmod o+r sample.dat
```

とすればよい.

問題 2

(1)
```
scanf("%d", &x);
scanf("%d", &y);
```

(2)
```
int max(int a, int b)
{
      if(a >= b){
        return (a);
        }else{
        return (b);
      }
}
```

(3)
```
#define max(a,b) ((a)>(b)?(a):(b))
```

問題 3

(1)
```
int fact2(int n)
{
         if(n==0){
         return (1);
         }else{
         return (n*fact2(n-1));
         }
}
```

(2)
```
for(i=1; i < 5;  i++){
        if(a[i] < min){
            min=a[i];
            }else{
            min=min;
            }
    }
```
(3)
```
./a.out > result.dat
```

問題 4

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#define l 10.0  /* 正三角形の辺の半分 */
#define N 10000 /* 繰り返し回数 */
double RAND(); /* この問題に限ってはあっても無くても良い */
main()
{
        FILE *fpr;
        double x0,y0,x1,y1,x2,y2,x3,y3,x,y,rr;
        int i;
    /* 三角形の定義 */
        x1=0;
        y1=0;
        x2=2.0*l;
        y2=0;
        x3=l;
        y3=sqrt(3.0)*l;
   /* 初期条件の設定（三角形の内部に選ぶ）*/
        x0=y0=l;
        x=x0;
        y=y0;
 if((fpr = fopen("result.dat", "wt")) !=NULL){ /* ファイルオープン */
    for(i=1; i <= N; i++){
        rr = RAND();  /* [0,1] 一様乱数を rr に代入 */
       if(rr < 1.0/3){
                x=0.5*(x+x1);
                y=0.5*(y+y1);
          }else if((rr >= 1.0/3) && (rr < 2.0/3)){
                x=0.5*(x+x2);
                y=0.5*(y+y2);
          }else{
                x=0.5*(x+x3);
                y=0.5*(y+y3);
```

```
}
        fprintf(fpr,"%lf %lf\n", x,y); /* ファイルクローズ */
 }
}
 fclose(fpr);
}
```