プログラミング概論 & プログラミング基礎演習

(新潟大学　G コード (教養) 科目)

平成 28 年 9 月 9 日

元木 達也 (講義, 木曜 4 限演習)
motoki@ie.niigata-u.ac.jp

棚橋 重仁 (水曜 4 限演習)
tanahashi@eng.niigata-u.ac.jp

目 次

0　ガイダンス

- 科目の概要,
- 受講要件等,
- 受講に当っての留意事項,
- 科目のねらい, 学習の到達目標,
- 教科書、参考書,
- 授業予定,
- 成績評価の方法と基準

科目の概要:　データ処理の手順をコンピュータに指示することによって直接コンピュータを使ってみたい人、およびそれらの作業を通じてコンピュータというものを理解したいと考えている人のための講義です。「データ処理の手順をコンピュータに指示する」ための言語としては現在最も普及している C 言語を選び、例題を中心に説明する。

受講要件等:　内容を実際に体験するため「プログラミング基礎演習」(水曜４限または木曜４限) も並行して履修することが望ましい。 それゆえ、聴講受付の際は「プログラミング基礎演習」の受講者を優先する。

受講に当っての留意事項:

- 全ての受講者が C プログラミングの実習が出来る環境にあることを想定して授業を進める。
- 数学的な内容の箇所もありますので、高校時代に「数学 IA」しか履修していない人には厳しいかも知れません。　(前提としている数学的な内容についても、質問には応じるつもりでおりますが、.....。)
- 「プログラミング基礎演習」との同期をとるため、10 月に補講を行うことがあります。
- 理解を助けるために練習問題を時々出す予定です。
  $\Longrightarrow$ 必ず自分で考えてやって下さい。
- 授業はほぼ講義ノートに沿って進める予定なので、できるだけ予習をしておいて下さい。その方が結局は効率的です。

科目のねらい:

- プログラミングを通じてコンピュータがどの様に動くのかを理解する。

学習の到達目標:

- 小規模な処理手順なら自分で設計できる。

教科書、参考書:　講義ノート (購入してもらう予定) に従って話を進める。必要に応じて、参考書を読んで下さい。 例えば次の様な参考書がある。

- 蓑原隆「C プログラミングの基礎」(2001 年, サイエンス社, 1600 円＋税)
- 皆本晃弥「やさしく学べる C 言語入門 —基礎から数値計算入門まで—」(2004 年, サイエンス社,2400 円＋税)
- 鈴木正人「(情報学コアテキスト 23) 実践 C プログラミング—基礎から設計/実装/テストまで—」(2008 年, サイエンス社,1900 円＋税)

- 柴田望洋「新版明解C言語 入門編」(2004年, ソフトバンククリエイティブ,2200円＋税)
- 柴田望洋「新版明解C言語 実践編」(2004年, ソフトバンククリエイティブ,2200円＋税)
- H. シルト「独習C 第4版」(2007年, 翔泳社, 3200円＋税)
- 阿部圭一 (編)「(インターユニバーシティ) プログラミング」(1999年, オーム社,2300円＋税)
- 浦昭二&原田賢一 (編)「C入門」(1994年, 培風館, 2150円＋税)
- P.Prinz&U.Kirch-Prinz「Cデスクトップリファレンス」(2003年, オライリージャパン/オーム社,1200円＋税)
- B.W. カーニハン&D.M. リッチー「プログラミング言語C 第2版」(1989年, 共立出版, 2800円＋税)

授業予定:

［コンピュータ入門 (I)］ —プログラミングの話をする前の準備 —

1 序論
コンピュータの一般的な構成、動作原理

1〜2 コンピュータ内での情報の表現

［C プログラミング入門］

3〜4 プログラミング序論 —アルゴリズムの設計・記述—,
整数計算の簡単なプログラム例

4〜6 処理の選択と繰り返し

6〜8 実数データの扱い

8 数値計算 —方程式の解法, 数値積分—

9 配列 —「添字付きの名前」(例えば x_i) をCプログラムでどう表すか—

10〜11 関数の定義 —大規模な処理手順をC言語で表すための手法—

12〜13 基本的なデータ型と構造体, 共用体

14 ファイル入出力

［コンピュータ入門 (II)］ — プログラミングに関連した話 —

15 コンピュータのソフトウェア,
コンピュータ発展の歴史

試験

16 期末試験

成績評価の方法と基準: 上記目標の達成度を期末試験 (80%), ほぼ毎週出す練習問題の出来具合等 (20%) に基づいて評価し、60点以上を合格とする。(但し、授業の出席率が２／３未満の受講生については、原則として単位を認めない。)

1 序論

- コンピュータの利用されている身近な例,
- コンピュータの普及に伴って起きる社会問題

1.1 コンピュータの利用されている身近な例

コンピュータの利用例:

- 現象をシミュレーションによって解析,
  (特に現実の実験が困難な現象 (e.g. 構造物の解析, 分子軌道の解析) に対して、シミュレーションは有効。スーパーコンピュータがよく使われる。)
- 組合せ最適化問題の最適解を探索 ··· (巡回セールスマン問題, スケジューリング問題など),
- 実験データの処理 ··· (実験結果のデータを整理し、特徴を統計的にとらえるなど),
- 論文の執筆 ··· (ワードプロセッサ, LaTeX, お絵描きソフト),
- 画像処理 ··· (例えば、観測された画像からノイズを除去してより鮮明にするなど),
- 新聞紙面の作成/編集, 電子出版, DTP(Desk Top Publishing),
- 自動車の設計
  ··· (例えば、CAD(Compute Aided Design) を使って決まりきった部分はコンピュータに任せる。)
- 複数の産業用ロボットによる自動車の組立て,
  (各々の産業用ロボットにはその動作を制御する (マイクロ) コンピュータが置かれ、全てのロボットと自動車組立の流れを管理する計算機が中央に置かれる。)

ネットワーク経由での利用:

- 新潟大学学務情報システム,
- 新潟大学附属図書館所蔵の図書・雑誌の検索 (データベースの利用),
- 列車, 航空機の座席予約,
- 銀行の現金自動支払機,
- デパートや小売り店の POS(Point Of Sales；販売時点情報管理),
- 住民基本台帳ネットワークシステム,

ネットワークと組み合わせて利用:

- インターネット,
  (LAN(Local Area Network) 同士を繋いで出来上がった世界最大規模のコンピュータネットワーク。 電子メール, WWW, ネットニュースなどのサービスがある。)
- イントラネット ··· (インターネットと同様の仕組みを 1 つの会社内に閉じた形で構築したもの)

マイクロコンピュータチップ (5mm 角位の大きさ) が組み込まれた機器:

- 携帯電話,
- テレビゲーム機, 電卓, 時計, カメラ, ミシン, 炊飯器, 洗濯機, エレベータ,
  自動車 (排ガス規制の克服, 燃料消費効率の向上などのために)

1.2 コンピュータの普及に伴って起きる社会問題

- コンピュータ犯罪: 例えば、

◇ コンピュータ・ウイルス,

補足：
自己増殖/伝染する様に作られた犯罪プログラムのこと。例えば、Nimda は 2001 年に新潟大学内を始め各地で被害をもたらし新聞等の記事にもなった。他にも、Brain(1986), Cascade(1987), Jerusalem(1987), Morris Worm(1988), Japanese Christmas(1989), Michelangelo(1991), Concept(1995), ..., sister (2002), Sasser(2004), Cabir(2004), 山田ウイルス (2005), polipos(2006), 山田オルタナティブ (2006), Gumblar(2009) 等があったらしい。
(http://ja.wikipedia.org/wiki/コンピュータウイルス)

◇ コンピュータへの不正アクセス,

例えば、
1985 年、筑波の高エネルギー研究所の計算機に西ドイツからの不法侵入があり、大事なプログラムが壊された。

◇ コンピュータの不正使用,

例えば、
— 金融機関の内部職員が虚偽のデータを計算機に入力して、架空名義の預金口座に入金があったように見せかける。
— トロイの木馬：プログラムの中に内密の処理手順を挿入しておき、(プログラムの本来の目的を果たさせながら) 無許可の機能も実行させる。
— サラミ・テクニック：トロイの木馬的な方法により、多くの財産・資源から少しずつ (e.g. 利息計算の際の端数の処理に細工して) 目立たない様に盗みとる。

◇ 有料ソフトウェアを無断でコピーして販売,
◇ 顧客データをコピーして他社に売却,
◇ インターネット上で個人中傷,
◇ インターネット詐欺, 例えばオンラインショッピングによる代金, 商品の横領
◇ インターネットを利用したネズミ講, マルチ商法
◇ 非合法な物品 (薬物, 危険物など) や情報の販売
◇ キャッシュカードを不正に作出し、それを使って現金を引き出す。

- **新しい職業病**: 例えば、
  ◇ VDT(Video Display Terminal) 作業による視力障害,
  ◇ コンピュータ依存症 (対人恐怖，働き過ぎ),
  ◇ コンピュータ拒否症,
  ◇ IC 工場での極度のストレス
- **計算機システムの故障による混乱**: 例えば、JR の列車の座席予約を管理する計算機が地震などで停止すると、(一時的にではあるが) 日本中が混乱する。また、管理プログラムの小さな誤りのために大事なファイルが消されたりすると、しばらく混乱する。
- **雇用の問題**: 「計算機を導入して業務を合理化すると人が要らなくなるのではないか？」という不安。
- **プライバシーの侵害**: 個人のデータを（無断で）集めることによって起こる問題。例えば、ある犯罪の記録が事件に全く無関係な人のデータとして誤って入力され、この記録が信用調査録として広く行き渡ったとしたら、この人は知らぬ間に不利益を被ることになる。しかも、この人は誤ったデータを見ることも訂正することもできない。
- **ソフトウェアの著作権**: ソフトウェアやホームページ上の文書や画像は従来の著作物とは異なる性質を持つ。 これにどういう風に知的所有権を認めるか?

- インターネット中毒:

コンピュータについての正しい理解は現代では不可欠。

⇨ 正しい理解のための基礎を与えるのがこの講義の目的
(プログラミングを通じてコンピュータの動作を理解する。)

2 コンピュータの構成、動作原理

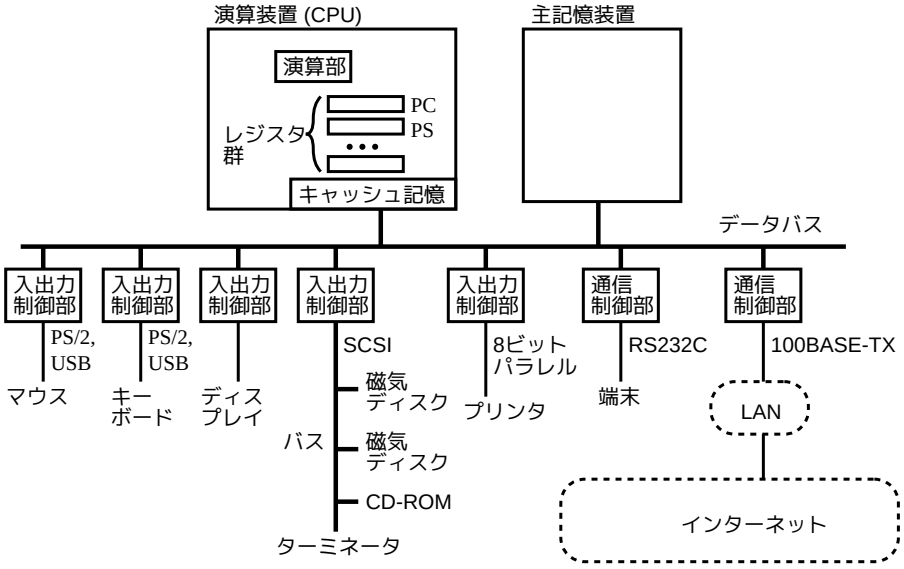
- コンピュータのハードウェア構成,
- コンピュータの動作原理

2.1 コンピュータのハードウェア構成

ハードウェアは、主に

- 演算装置 (CPU, プロセッサ),
- 主記憶装置 (メモリ),
- 入出力制御部 (または通信制御部),
- バス,
- 通信路,
- 周辺装置

から成る。(下図)



演算装置 (CPU, プロセッサ)： 計算機システムの処理を司る部分。

- 主記憶装置から**機械語命令を 1 つずつ順番に読み込み実行していく。**
  ($\Longrightarrow$ 2.2 節プログラム内蔵方式)
- 必要に応じて、データを主記憶装置から読み込んだり実行結果を主記憶装置に格納したりする。
- 主記憶装置との間のデータ転送はバスを介して行う。
- 演算・データ処理のために多くの**レジスタ** (i.e.CPU 内にあって演算などに使われる最高速記憶) を持っている。
  (1) **演算用レジスタ**：演算対象のデータ等を入れる。
  (2) **制御用レジスタ**：プロセッサの動作を制御するためのもの。

- **プログラムカウンタ** (program counter)
  - ··· 次の実行する命令のアドレスを記憶する。
- **プロセッサ状態レジスタ** (processor status register)
  - ··· 「走行モード」を始めとしたプロセッサの状態を保持する。

主記憶装置 (メモリ)： CPU から直接アクセスできる記憶装置。

- 使用中のデータだけでなく実行中のプログラム (の断片) も必ずこの中に入れておく。
- データの記憶場所を識別するために記憶領域には**番地** (address) が付けられ、CPU はこの番地を指定することにより主記憶内のデータの読み書きを行う。
- プロセッサやバスとの関係にも依存するが、連続する複数番地のデータを一度に読み書きできる。
- メモリの多くは揮発性である。

キャッシュ記憶： CPU から主記憶内のデータへのアクセス速度を見かけ上高速化するための高速記憶。

- アクセスされた主記憶の情報はキャッシュ記憶内に一時的に格納され、近い将来再びアクセスされた時はキャッシュ記憶から取り出される。
- メモリへのアクセス回数を抑えるため、キャッシュ記憶に無いデータをメモリから読み込む場合には、必要とするデータだけでなく近くのデータも一緒に読み込んで複製しておく。
- プログラムの実行は連続して進むことが多く、また、短い時間内では処理するデータも一部のものに集中することが多い。[こういう性質を**プログラムの局所参照性** (program locality) と言う。] プログラムの実行が局所参照性を持っているので、キャッシュ記憶を用いることによって主記憶内のデータへの実際上のアクセススピードを上げることが可能になる。

入出力制御部： 周辺装置、端末装置、他のコンピュータとの間の入出力を行う。

- 相手の装置が能動的である場合には**通信制御部**と呼ぶ。

バス： プロセッサ、メモリ、入出力制御部を結んで、それらの間のデータ授受を行うためのもの。

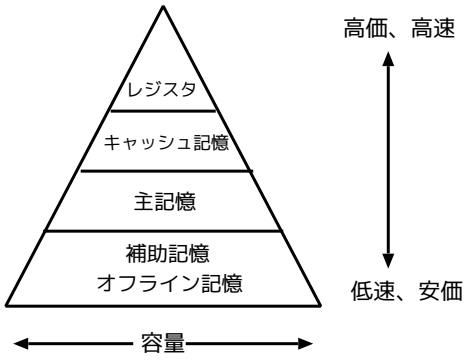
- 8 ビット以上のデータを並列に転送する。

通信路： 入出力装置との接続には様々な通信路を用いる。 次のような接続規格がある。[参考文献：小高知宏「計算機システム」(森北出版, 1999)]

名称	接続装置	特徴
RS-232C	モデム, 端末装置, プリンタ	低速のシリアルインターフェース
8 ビットパラレル (セントロニクス)	プリンタ	低速のパラレルインターフェース
SCSI	磁気ディスク, CD-ROM,MO, イメージスキャナ	高速 (5〜80Mbytes/s) のパラレルインターフェース
USB	キーボード, マウス, プリンタ, モデム, ディジタルカメラ	高速 (最大 12Mbits/s) の汎用インターフェース。キーボード, マウス,RS-232C などの低速な入出力装置が抱えている様々な問題を解決するために開発された。複数の機器をバス接続できる。
IEEE1394 (Fire Wire)	マルチメディア入出力装置	高速 (最大 400Mbits/s) の汎用インターフェース
ファイバーチャネル	高速ディスク装置	高速 (1Gbits/s) の汎用インターフェース
100BASE-TX	LAN	100Mbits/s, ツイストペア線
1000BASE-T	LAN	1Gbits/s, ツイストペア線

周辺装置： 計算機本体の周辺に置かれる装置を総称して言う。例えば、磁気ディスク、プリンタなど。

記憶装置の階層：



2.2 コンピュータの動作原理

現在の普通の計算機においては、

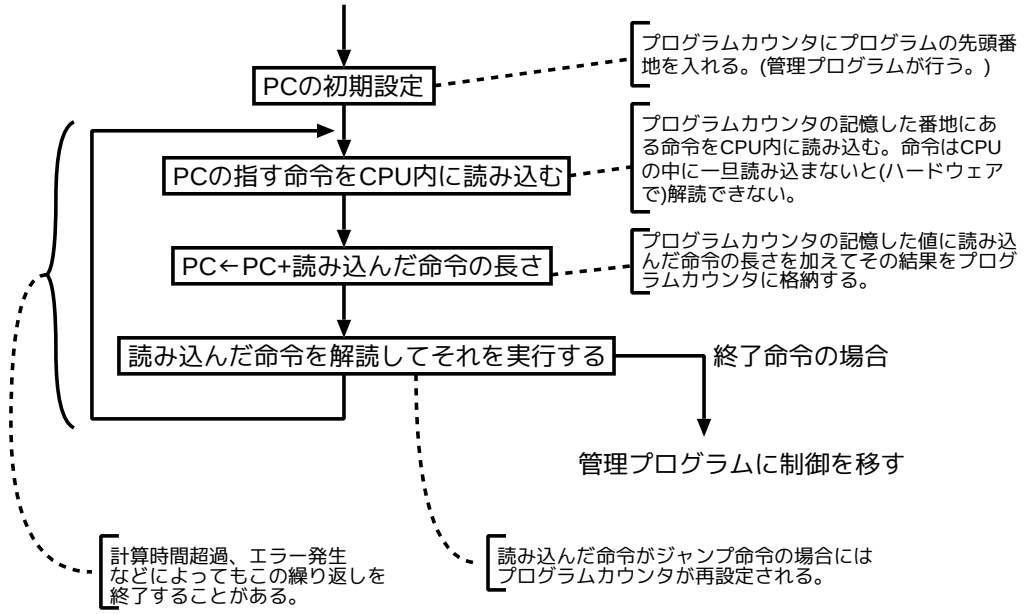
- プログラム (i.e. 処理手順) をデータと同様に記憶装置内に格納し、
- プログラムを構成する機械語命令を逐次的に読み出しては実行していくことにより自動的に動作させる、

いわゆる**プログラム内蔵方式** (stored program system, または**ノイマン型**) が採用されている。この方式ではプログラムをデータとして加工できるなど柔軟性のある計算機利用が可能なので、1947 年の EDSAC 以来ほとんど全ての計算機がこの方式を採用している。[厳密に言うと、1945 年にフォン・ノイマンが考案した基本構造を持つ計算機をノイマン型と言い、ノイマン型計算機の中で採用されたプログラムの記憶・実行の方式をプログラム内蔵方式と言う。]

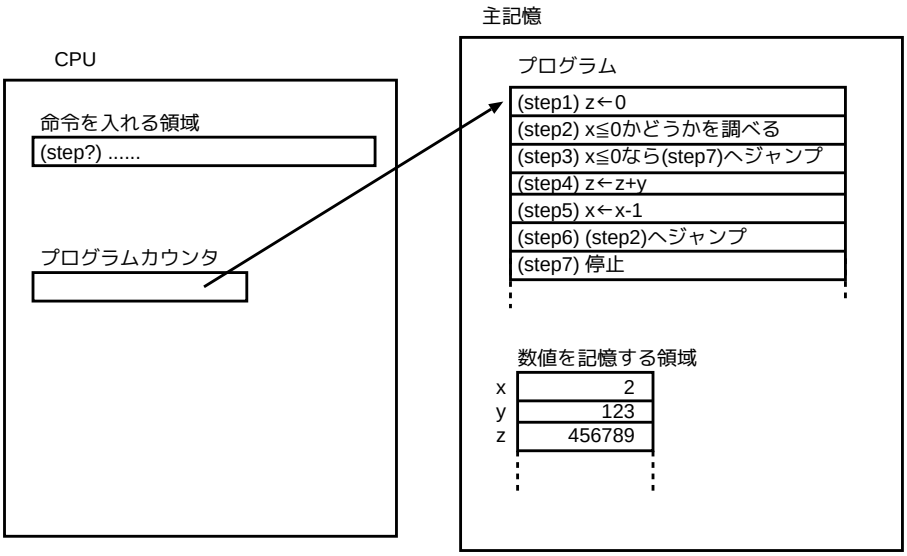
プログラム内蔵方式の計算機では、CPU は**プログラムカウンタ** (program counter) と呼ばれる**レジスタ記憶** (register; CPU 内にあって演算などに使われる最高速記憶) を用いてプログラム中の命令を逐次実行する。より具体的に言うと、プログラムカウンタは

次に実行する命令の入った (主記憶上の) 番地

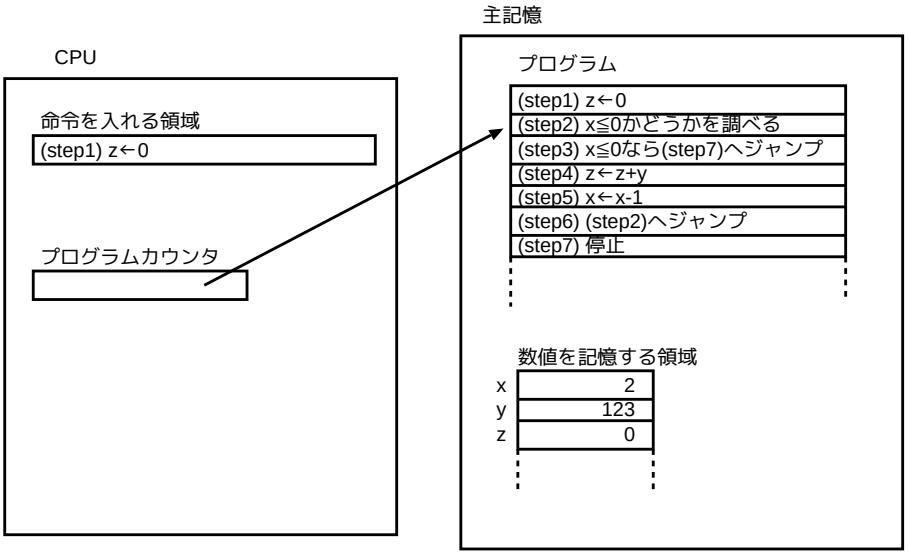
を常に記憶するものであり、CPU はこれを用いてプログラムを次の流れ図に従って動作させる。[流れ図で書いたが、この動作はハードウェアで自動的に行われる。 また、計算機システム全体を管理する管理プログラムもこれと同様の流れ図に従って動作する。]



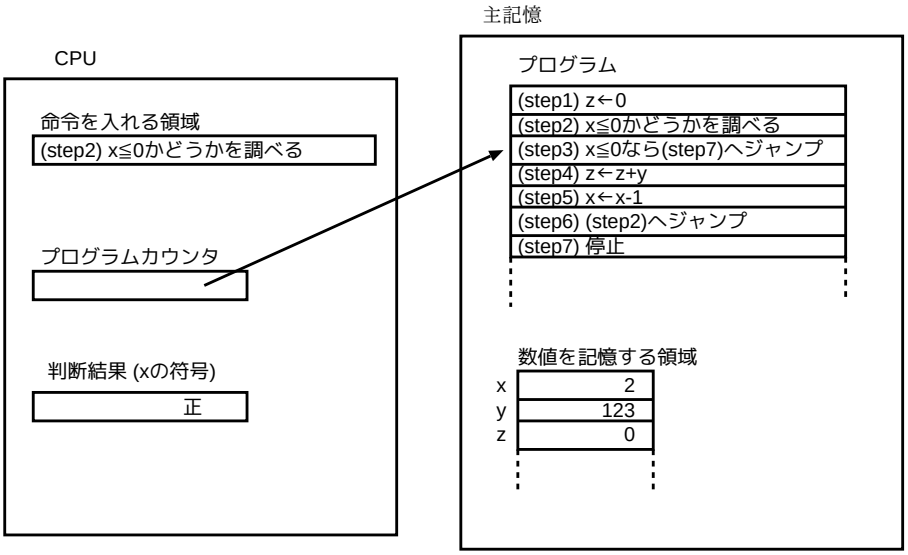
例 2.1 (プログラム内蔵方式計算機の動作)
(状況 1)



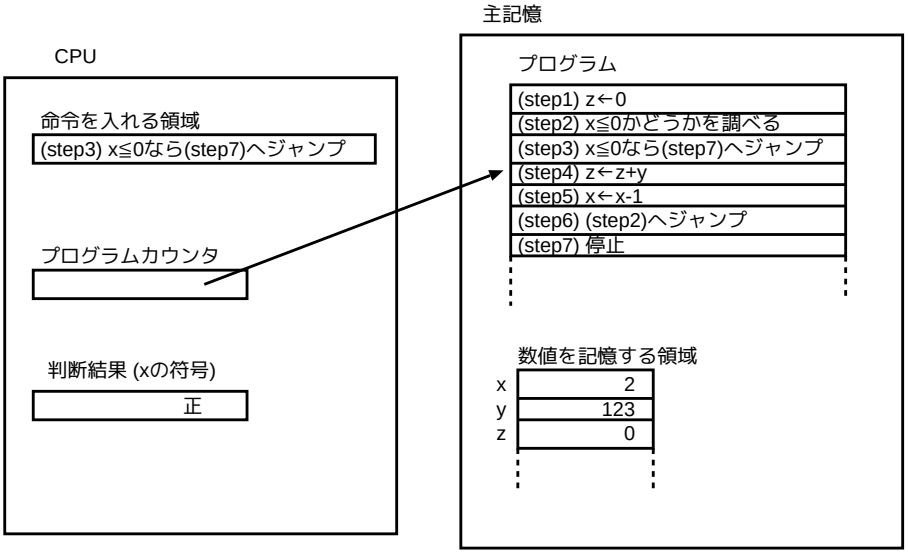
(状況 2)


主記憶
CPU
プログラム
命令を入れる領域
(step1) z←0
(step2) x≦0かどうかを調べる
(step3) x≦0なら(step7)へジャンプ
(step4) z←z+y
(step5) x←x-1
(step6) (step2)へジャンプ
(step7) 停止
プログラムカウンタ
数値を記憶する領域
x 2
y 123
z 0

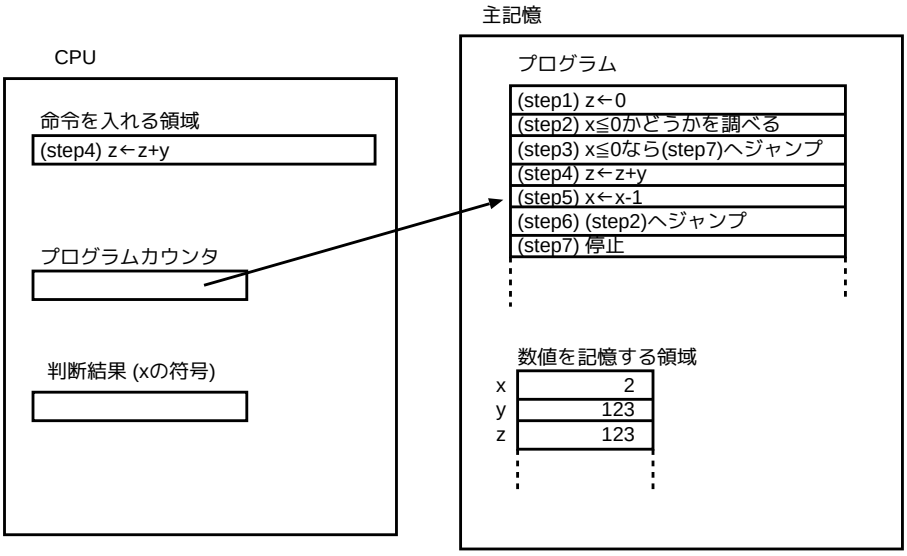
(状況 3)


主記憶
CPU
プログラム
命令を入れる領域
(step2) x≦0かどうかを調べる
(step1) z←0
(step2) x≦0かどうかを調べる
(step3) x≦0なら(step7)へジャンプ
(step4) z←z+y
(step5) x←x-1
(step6) (step2)へジャンプ
(step7) 停止
プログラムカウンタ
判断結果 (xの符号)
正
数値を記憶する領域
x 2
y 123
z 0

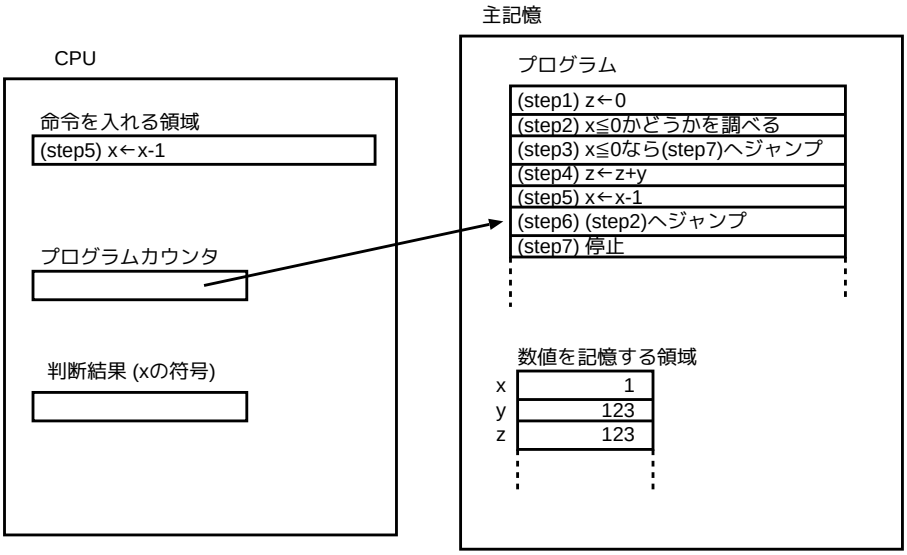
(状況 4)


主記憶
CPU
プログラム
命令を入れる領域
(step3) x≦0なら(step7)へジャンプ
(step1) z←0
(step2) x≦0かどうかを調べる
(step3) x≦0なら(step7)へジャンプ
(step4) z←z+y
(step5) x←x-1
(step6) (step2)へジャンプ
(step7) 停止
プログラムカウンタ
判断結果 (xの符号)
正
数値を記憶する領域
x 2
y 123
z 0

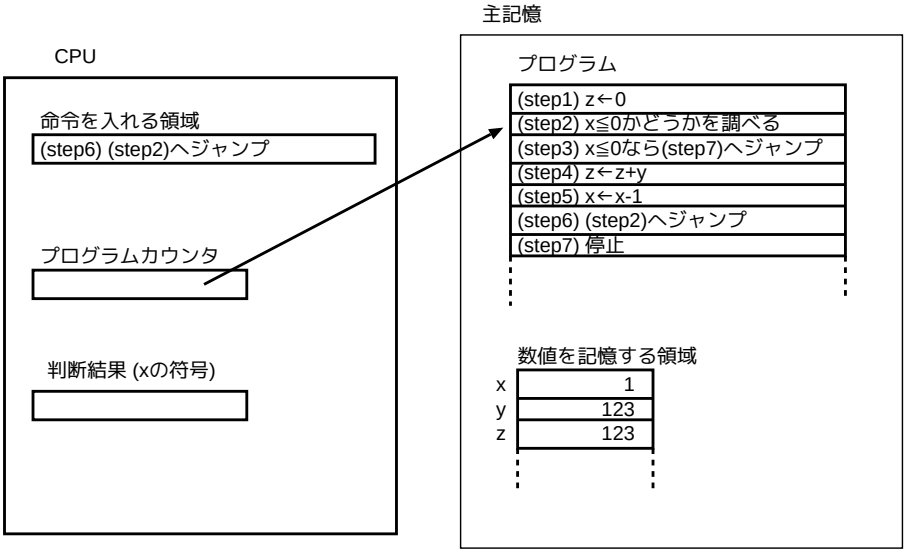
(状況 5)


CPU
命令を入れる領域
(step4) z←z+y
プログラムカウンタ
判断結果 (xの符号)
主記憶
プログラム
(step1) z←0
(step2) x≦0かどうかを調べる
(step3) x≦0なら(step7)へジャンプ
(step4) z←z+y
(step5) x←x-1
(step6) (step2)へジャンプ
(step7) 停止
数値を記憶する領域
x 2
y 123
z 123

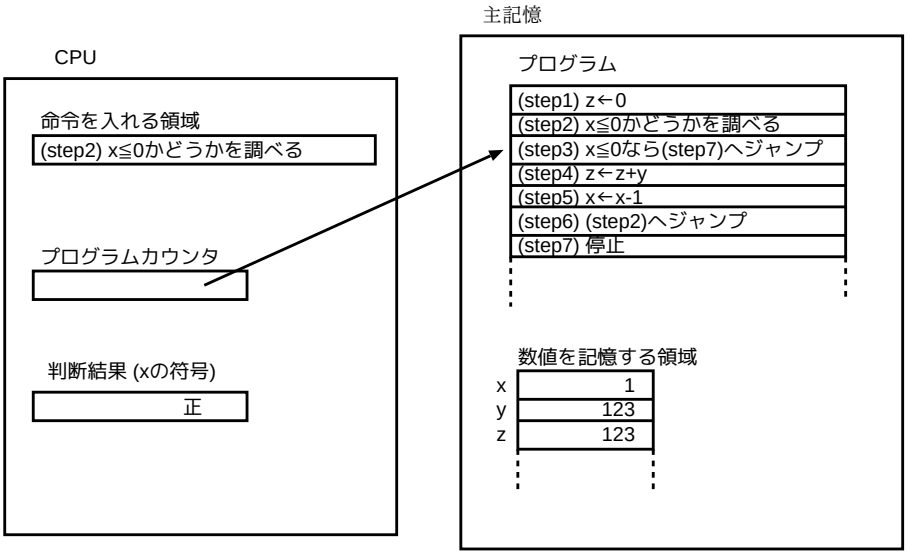
(状況 6)


CPU
命令を入れる領域
(step5) x←x-1
プログラムカウンタ
判断結果 (xの符号)
主記憶
プログラム
(step1) z←0
(step2) x≦0かどうかを調べる
(step3) x≦0なら(step7)へジャンプ
(step4) z←z+y
(step5) x←x-1
(step6) (step2)へジャンプ
(step7) 停止
数値を記憶する領域
x 1
y 123
z 123

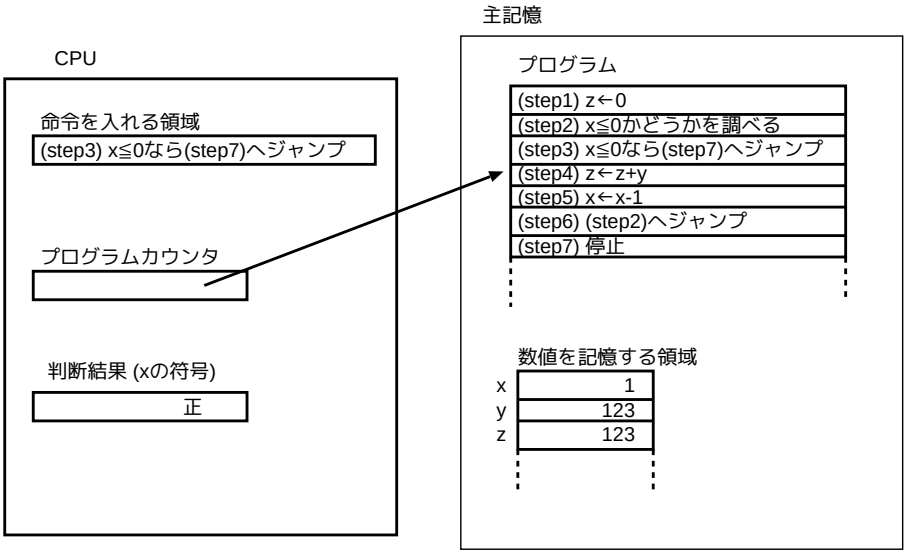
(状況 7)


CPU
命令を入れる領域
(step6) (step2)へジャンプ
プログラムカウンタ
判断結果 (xの符号)
主記憶
プログラム
(step1) z←0
(step2) x≦0かどうかを調べる
(step3) x≦0なら(step7)へジャンプ
(step4) z←z+y
(step5) x←x-1
(step6) (step2)へジャンプ
(step7) 停止
数値を記憶する領域
x 1
y 123
z 123

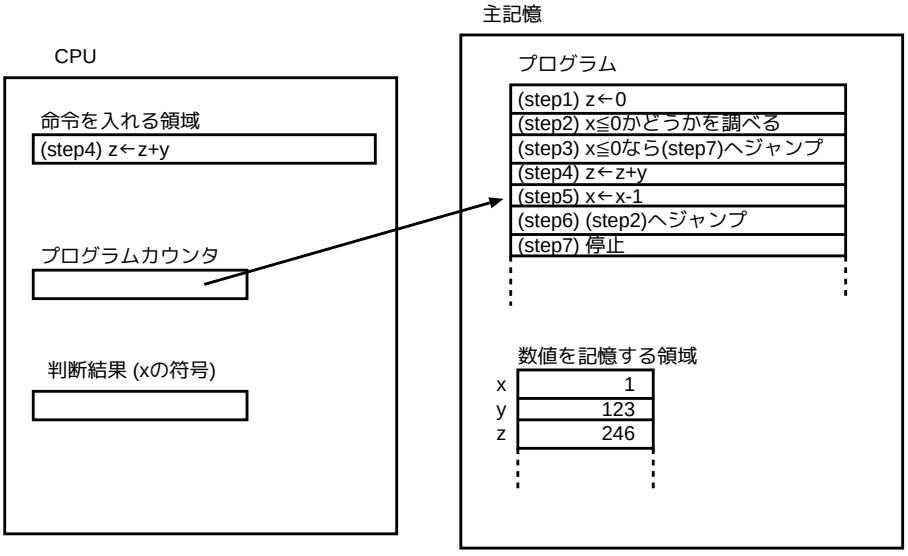
(状況 8)


主記憶
CPU
プログラム
(step1) z←0
(step2) x≦0かどうかを調べる
(step3) x≦0なら(step7)へジャンプ
(step4) z←z+y
(step5) x←x-1
(step6) (step2)へジャンプ
(step7) 停止
命令を入れる領域
(step2) x≦0かどうかを調べる
プログラムカウンタ
判断結果 (xの符号)
正
数値を記憶する領域
x 1
y 123
z 123

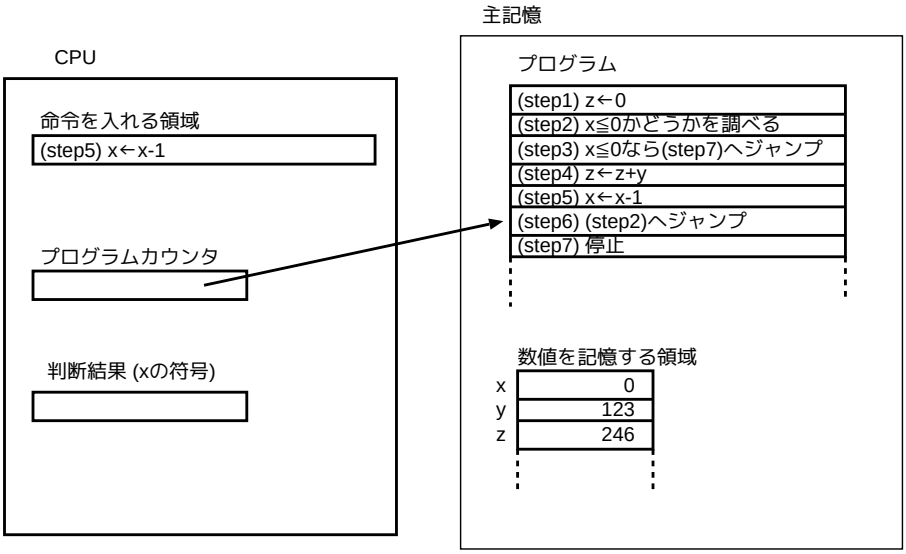
(状況 9)


主記憶
CPU
プログラム
(step1) z←0
(step2) x≦0かどうかを調べる
(step3) x≦0なら(step7)へジャンプ
(step4) z←z+y
(step5) x←x-1
(step6) (step2)へジャンプ
(step7) 停止
命令を入れる領域
(step3) x≦0なら(step7)へジャンプ
プログラムカウンタ
判断結果 (xの符号)
正
数値を記憶する領域
x 1
y 123
z 123

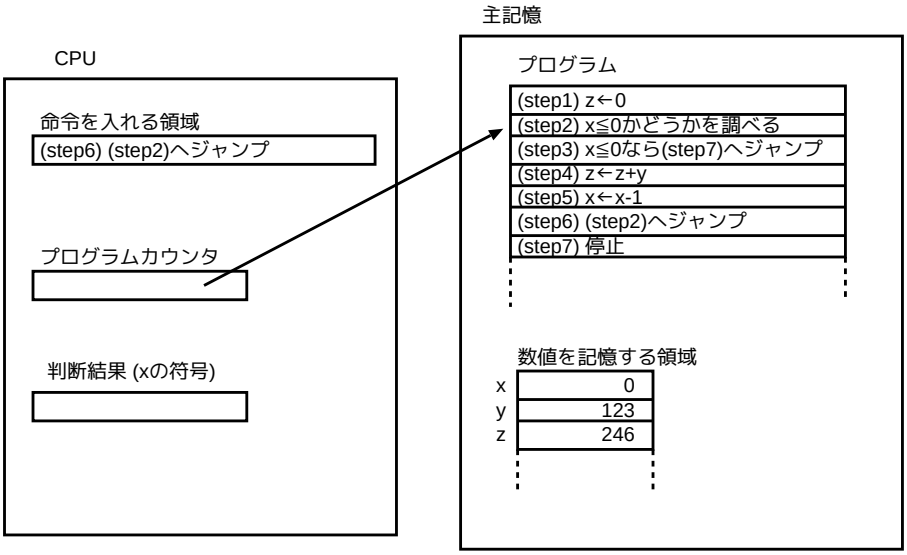
(状況 10)


主記憶
CPU
プログラム
(step1) z←0
(step2) x≦0かどうかを調べる
(step3) x≦0なら(step7)へジャンプ
(step4) z←z+y
(step5) x←x-1
(step6) (step2)へジャンプ
(step7) 停止
命令を入れる領域
(step4) z←z+y
プログラムカウンタ
判断結果 (xの符号)
数値を記憶する領域
x 1
y 123
z 246

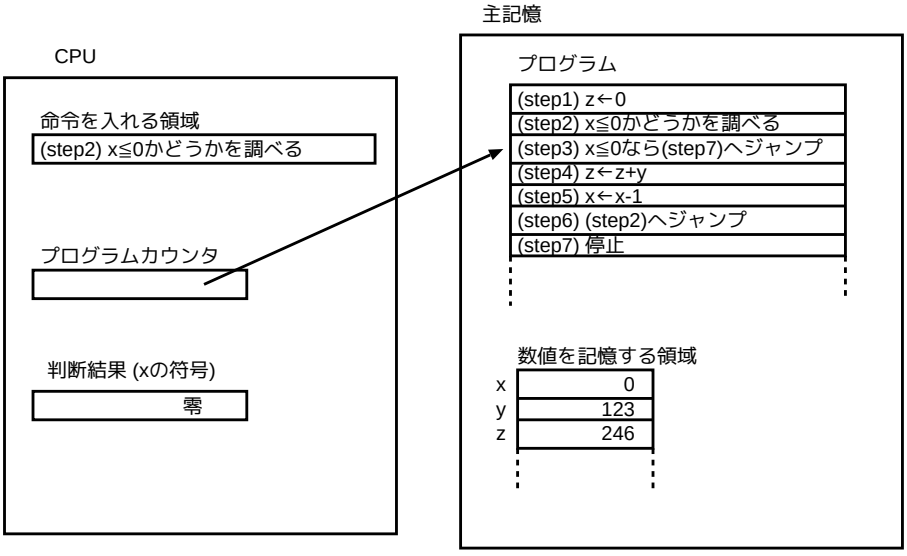
(状況 11)



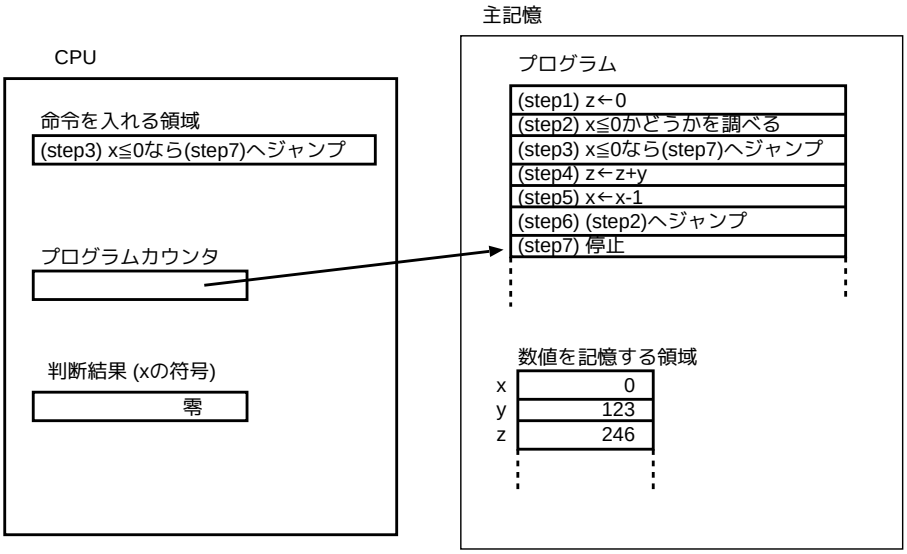
(状況 12)



(状況 13)



(状況 14)


主記憶
CPU
プログラム
(step1) z←0
(step2) x≦0かどうかを調べる
(step3) x≦0なら(step7)へジャンプ
(step4) z←z+y
(step5) x←x-1
(step6) (step2)へジャンプ
(step7) 停止
命令を入れる領域
(step3) x≦0なら(step7)へジャンプ
プログラムカウンタ
判断結果 (xの符号)
零
数値を記憶する領域
x 0
y 123
z 246

3　コンピュータ内での情報の表現

- 2 進法による情報の表し方,
- 文字データの内部表現,
- 2 進法による非負整数の表現,
- 整数データの内部表現,
- 実数表現の内部表現

3.1　2 進法による情報の表し方

情報の最小単位:　現在のほとんどの計算機は N.Wiener の提案に従って**フリップフロップ** (flip-flop)，すなわち

2 つの安定した状態 (e.g. 電圧の高低, 磁化の向き) を持つ素子

を基本要素にして構成されている。[これは元来この様な素子の構成の容易さや動作の安定性などの理由による。]

フリップフロップを用いた情報表現:　数値や文字, プログラムなどのデータはフリップフロップを複数個組み合わせて表される。 多くの計算機では、通常、数値は 32 **ビット** (bit;BInary digiT の略で、フリップフロップ何個分であるかを表す記憶容量の単位), 英数字は 8 ビット, 漢字は 16(または 24) ビットで表される。 例えば、0100 0001 という列で “A” という文字を表す。

2 進法による情報の表現:　フリップフロップの持つ 2 つの安定した状態が物理的にどんなものであるかは、記憶素子について議論する場合以外は重要でない。そこで、以下ではフリップフロップの 2 つの安定状態を 0 と 1 で表すことにする。

$\Longrightarrow$ 全てのデータは 0 と 1 の列によって表される。この表記法を **2 進法** (binary notation) という。

[以下では、10 進法と区別するために 0 と 1 の列は例えば B’10101101111’ と表す。]

8 進法と 16 進法:　2 進法ではデータを表すのに字数が長くなり、我々人間にとって不便である (i.e. 分かりにくく間違え易い)。　0 と 1 の列を 3〜4 ビット毎に区切ってそれぞれの区画を 1 文字で表す方がコンパクトで分かり易い。

$\Longrightarrow$ 次の左側の表に従って 3 ビットの列の各々を 0〜7 の 8 文字で代用する表記法を **8 進法** (octal notation)，右側の表に従って 4 ビットの列の各々を 0〜9,A〜F の 16 文字で代用する表記法を **16 進法** (hexadecimal notation) という。

例えば、
B’101011010111’ は 8 進法で 5327，16 進法で AD7 と表される。表示した数字列が 8 進法,16 進法のものであることを明示したい時は、以下では 2 進法の場合に倣ってそれぞれ例えば O’5327’, X’AD7’ と書くことにする。C 言語では、これらは各々 05327, 0xAD7 と表記される。

2 進法	8 進法
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

2 進法	16 進法
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

3.2 コンピュータ内での文字データの表現

我々人間が計算機を利用する際、**ビット列** (i.e. 0 と 1 の列) を入力してビット列の出力を得るというのでは不便である。文字列を入力して文字列の出力を得ることが出来ないといけない。そのため、6〜8 ビットを組み合わせて英数字や記号 1 文字を表す方式がいくつか定められている。

英数字何文字を記憶できるかを表す容量の単位としては**バイト** (byte) が用いられる。何ビットを組み合わせて 1 文字を表すか，すなわち 1 バイトが何ビットに相当するかは計算機の機種によって異なるが、通常 1 バイト=8 ビット である。[JIS でも、こう定められている。]

次に、一般的な文字符号体系を 3 つ示す。

- **EBCDIC 符号体系** (Extended Binary Coded Decimal Interchange Code)： IBM 社が 1964 年に定義したもので、IBM の汎用計算機 (とその互換機) で採用された。 IBM のメインフレームが圧倒的なシェアを占めていた 1960〜70 年代はこの符号体系 (または英小文字の代わりにカタカナを割り当てた EBCDIK 符号体系) がよく使われたが、PC や WS が普及し日本語処理が不可欠となった現在ではあまり用いられていない。 具体的なコード表は省略。

- **ASCII7ビット符号体系** (American Standard Code for Information Interchange)：

		上位3ビット							
		0	1	2	3	4	5	6	7
下位4ビット	0	NUL	DLE	空白	0	@	P	`	p
	1	SOH	DC1	!	1	A	Q	a	q
	2	STX	DC2	"	2	B	R	b	r
	3	ETX	DC3	#	3	C	S	c	s
	4	EOT	DC4	$	4	D	T	d	t
	5	ENQ	NAK	%	5	E	U	e	u
	6	ACK	SYN	&	6	F	V	f	v
	7	BEL	ETB	’	7	G	W	g	w
	8	BS	CAN	(	8	H	X	h	x
	9	HT	EM	)	9	I	Y	i	y
	A	LF	SUB	*	:	J	Z	j	z
	B	VT	ESC	+	;	K	[	k	{
	C	FF	FS	,	<	L	\	l	|
	D	CR	GS	-	=	M	]	m	}
	E	SO	RS	.	>	N	^	n	~
	F	SI	US	/	?	O	_	o	DEL

DEL . . . 機能文字

機能文字

補足：

機能文字 (functional character) は書式の変更や伝送データの開始・終了などの制御を行うための文字で、**制御文字** (control —) とも言う。それぞれの名称／意味は次の通り。

NUL···空白 (null)
SOH···ヘディング開始 (start of heading)
STX···テキスト開始 (start of text)
ETX···テキスト終了 (end of text)
EOT···伝送終了 (end of transmission)
ENQ···問合せ (enquiry)
ACK···肯定応答 (acknowledge)
BEL···ベル (bell)
BS ···後退 (backspace)
HT ···水平タブ (horizontal tabulation)
LF ···改行 (line feed)
VT ···垂直タブ (vertical tabulation)
FF ···書式送り (form feed)
CR ···復帰 (carriage return)
SO ···シフトアウト (shift out)
SI ···シフトイン (shift in)

DLE···伝送制御拡張 (data link escape)
DC1 ···装置制御 1(device control 1)
DC2 ···装置制御 2(device control 2)
DC3 ···装置制御 3(device control 3)
DC4 ···装置制御 4(device control 4)
NAK···否定応答 (negative acknowledge)
SYN ···同期信号 (synchronous idle)
ETB···伝送ブロック終結 (end of transmission block)
CAN···取消 (cancel)
EM ···媒体終端 (end of medium)
SUB ···置換文字 (substitute character)
ESC ···拡張 (escape)
FS ···ファイル分離文字 (file separator)
GS ···グループ分離文字 (group separator)
RS ···レコード分離文字 (record separator)
US ···ユニット分離文字 (unit separator)
DEL···抹消 (delete)

- **JIS8 ビット符号体系** (Japanese Industrial Standard)：

		上位4ビット															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
下位4ビット	0	NUL	DLE	空白	0	@	P	`	p	未定義			ー	タ	ミ	未定義	
	1	SOH	DC1	!	1	A	Q	a	q			。	ア	チ	ム		
	2	STX	DC2	"	2	B	R	b	r			「	イ	ツ	メ		
	3	ETX	DC3	#	3	C	S	c	s			」	ウ	テ	モ		
	4	EOT	DC4	$	4	D	T	d	t			、	エ	ト	ヤ		
	5	ENQ	NAK	%	5	E	U	e	u			・	オ	ナ	ユ		
	6	ACK	SYN	&	6	F	V	f	v			ヲ	カ	ニ	ヨ		
	7	BEL	ETB	’	7	G	W	g	w			ァ	キ	ヌ	ラ		
	8	BS	CAN	(	8	H	X	h	x			ィ	ク	ネ	リ		
	9	HT	EM	)	9	I	Y	i	y			ゥ	ケ	ノ	ル		
	A	LF	SUB	*	:	J	Z	j	z			ェ	コ	ハ	レ		
	B	VT	ESC	+	;	K	[	k	{			ォ	サ	ヒ	ロ		
	C	FF	FS	,	<	L	¥	l	|			ャ	シ	フ	ワ		
	D	CR	GS	-	=	M	]	m	}			ュ	ス	ヘ	ン		
	E	SO	RS	.	>	N	^	n	‾			ョ	セ	ホ	゛		
	F	SI	US	/	?	O	_	o	DEL			ッ	ソ	マ	゜		

X’5C’,X’7E’ 以外は ASCII と同じ

最後に、日本語符号体系としては次の 4 つがある。

- **JIS 漢字符号体系：** 日本語を扱うためには漢字が不可欠であるので、JIS では漢字符号 (Kanji code) が 1978 年に制定され 1983 年,1990 年に改訂されている 。JIS 漢字符号は 2 バイト/16 ビットのビット列で構成され、登録可能な漢字数 $2^{16} = 65536$ の内第 1 水準漢字 (平仮名, 片仮名, 英数字, 特殊文字も含む) として 2965 字, 第 2 水準漢字として 3390 字, 補助漢字として 5810 字, 計 12156 字が定められている。例えば「ＪＩＳ漢字コード」という文字列は X‘234A 2349 2353 3441 3B7A 2533 213C 2549’ という 16 バイトのビット列で表される。また、JIS 8 ビット符号と漢字符号を混ぜて使うために、漢字符号の開始を表すシフトコード (X‘1B2442’; i.e.‘Esc $B’ という文字列), 8 ビット符号の開始を表すシフトコード (X‘1B284A’; i.e.‘Esc (J’ という文字列) を用いる方法が定められている。JIS 符号体系を用いる利点は最上位ビットを使っていないということである。通信の際は最上位ビットが失われることが多いので、漢字を含む文書の送信は JIS コードで行うのがよい。
- **MS 漢字コード体系：** シフト JIS コード系とも呼ばれるが、JIS でなく JIS 漢字符号体系の変形である。Esc シーケンスを用いずに漢字符号と 8 ビット符号を混在させられる様に、1983 年に米国 Microsoft 社, アスキー, 日本 IBM, 三菱電機によって定義されたコード体系であり、長い間パソコン/MS-DOS /Windows の世界では事実上の標準になっていた。シフト JIS では、JIS 8 ビット符号で未定義の X‘81’〜X‘9F’, X‘E0’〜X‘FC’ のそれぞれにもう 1 バイト繋げることにより 2 バイトで 1 つの漢字を表す。この方法では、登録可能な漢字数は $(31 + 29)\times 2 = 15360$ である。漢字符号と 8 ビッ

ト符号を切り替える特別なコードは不要になったが、JIS コードとの対応が比較的複雑で、表せる文字数にも余裕がないという欠点もある。

- **EUC(Extended Unix Code) コード体系：** JIS 漢字符号体系の変形であり、UNIX-JIS コード, または UJIS とも呼ばれている。日本語 UNIX システム諮問委員会の検討 (AT&T 社からの要請による,1984〜5) の結果を基に 1986 年に AT&T 社が定めたコード体系であり、長い間 (2007 年頃まで？) UNIX/Linux における標準コードとして用いられていた。EUC コードでは、4 種類の文字集合 (ASCII, 漢字, 半角カタカナ, 補助漢字) の切り替えに各バイトの最上位ビットを使う。具体的には、B‘0... ....’ というビット列にはそのまま ASCII 文字を割り当てる。また、最上位ビットに 1 が立った部分については、最初のバイトが X‘8E’ なら次の 1 バイト (最上位ビットは 1) と合わせ合計 2 バイトで半角カタカナを, X‘8F’ なら次の 2 バイト (最上位ビットは 1) と合わせ合計 3 バイトで補助漢字 1 文字を, それら以外なら次の 1 バイト (最上位ビットは 1) と合わせ合計 2 バイトで漢字 1 文字を表すことになっている。ただ、EUC コードでは、キャラクタ端末上での文字幅と内部表現のバイト数が一致しない、などの理由で半角カタカナの扱いが敬遠されている。
- **Unicode 体系：** 世界各国の文字体系を共通化するために、IBM 社,Sun Microsystems 社, Microsoft 社, ノベル社などの米国企業が中心になって設立した Unicode コンソーシアムが提唱したもので、1993 年には ISO (International Organization for Standardization) の標準に、1995 年には JIS 規格となった。Unicode では、漢字, アルファベット, 中国語, ハングル語, などの文字をひとまとめに取り扱おうとする。日本語に関しては従来の第 1 水準, 第 2 水準, 補助漢字のコードが Unicode の中でもほぼそのまま使われているが、意味を無視して、外見がほぼ同じ中国の漢字と日本の漢字に同じコードを割り当てるといったことも行なわれているため、賛同しない人もいた。Unicode の中にも幾つかの符号化方式があるが、その内 UTF-8 は 1〜4 バイトにエンコードして ASCII に上位互換する方式で、最も一般的に利用されている。また、UTF-16 は最初の 65536 文字を 16 ビットで表し, それ以外を 32 ビットで表すもので、Windows では XP 以降の内部コードにこの方式が使われている。UTF-32 では各々の文字を 32 ビットで表す。

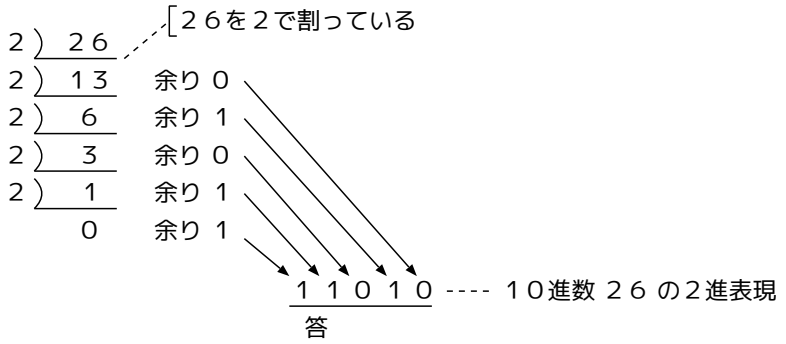
3.3 2 進法による非負整数の表現

10 進法では 0〜9 の数字列 $d_n d_{n-1}\cdots d_1 d_0$ によって $d_n\times 10^n + d_{n-1}\times 10^{n-1} + \cdots + d_1\times 10^1 + d_0\times 10^0$ という非負整数を表す。例えば $1234 = 1\times 10^3 + 2\times 10^2 + 3\times 10^1 + 4\times 10^0$ である。これに対して、2 進法では 0〜1 の数字列 $a_n a_{n-1}\cdots a_1 a_0$ によって $a_n\times 2^n + a_{n-1}\times 2^{n-1} + \cdots + a_1\times 2^1 + a_0\times 2^0$ という非負整数を表す。 例えば、2 進法の 11010 は (10 進) 非負整数 $1\times 2^4 + 1\times 2^3 + 0\times 2^2 + 1\times 2^1 + 0\times 2^0 = 26$ を表す。

10 進→ 2 進変換の実際的方法: 非負整数 x が 2 進法で B’$a_n a_{n-1}\cdots a_1 a_0$’ と表される場合、$x = a_n\times 2^n + a_{n-1}\times 2^{n-1} + \cdots + a_1\times 2^1 + a_0\times 2^0$ であるから各 a_i は x と i を用いて

$$a_i = \mathrm{mod}\left(\left\lfloor \frac{x}{2^i}\right\rfloor, 2\right)$$

但し $\lfloor\ \rfloor$ は小数点以下切り捨てを,
mod は第 1 引数を第 2 引数で割ったときの余りを表す
と表せる。この事実を用いると、(10 進) 非負整数が与えられた時その 2 進表現は例えば次の様にして求めることができる。



2 進法における加減乗除: 2 進法においても加減乗除は 10 進法の場合と全く同じ様に行える。例えば次の通り。

$$\begin{array}{r} 11011 \\ +\ 101111 \\ \hline 1001010 \end{array} \qquad \begin{array}{r} 101111 \\ -\ 11011 \\ \hline 10100 \end{array} \qquad \begin{array}{r} 11011 \\ \times\ 1011 \\ \hline 11011 \\ 11011 \\ 11011 \\ \hline 100101001 \end{array} \qquad \begin{array}{r} 101 \\ \hline 101\ )\ 11010 \\ 101 \\ \hline 110 \\ 101 \\ \hline 1 \end{array}$$

3.4 コンピュータ内での整数データの表現

10 進数 26 を 2 進法で表すと 11010 となる。それ故 10 進数 26 は 5 ビットで，そして一般に非負整数 m は $\lfloor \log_2 m \rfloor + 1$ ビットで表せるように思えるが、これでは計算機内のどこからどこまでが 1 つの整数を表すかの識別が大変で取扱いが不便である。
$\Longrightarrow$「ある決められたビット数で数値を表す」という様な標準化が必要。

（$\lfloor\ \rfloor$ は小数部を切り捨てる関数）

数値データを計算機内部で記憶する際の容量の標準単位としては、普通、**語** (word) が用いられる。語は何バイトかを集めて構成されるが、1 語が何バイトに相当するかは計算機の機種によって様々である。

補足：
語は単に数値データを記憶する時だけの基本単位ではなく、機械語プログラムを記憶する時の基本単位でもある。また、絶対値の小さな整数は**半語** (half word) あるいはバイトを基本単位として表されることもある。

非負整数データの内部表現: 2 進法による非負整数の表現がそのまま利用される。例えば 10 進数 26 は 2 進法で 11010 と表されるから、8 ビットで数値データを表す場合 10 進数 26 は計算機内部で 00011010 と表される。一般に n ビットで非負整数を表す場合は 0 〜$2^n - 1$ の間の整数が表現可能である。

整数データの内部表現: 表す整数データが非負の場合には「非負整数データの内部表現」と同じビット列で表す。例えば 10 進数 26 は 8 ビットではやはり 00011010 と表される。しかし、「整数データの内部表現」としては、負数の表し方によって次の 3 種類の方法が考えられた。以下、整数データを一般に n ビットで記憶する場合を考える。

実際には、
これら 3 種類の内 **2 の補数による方法がほとんど全ての計算機で採用されている。**

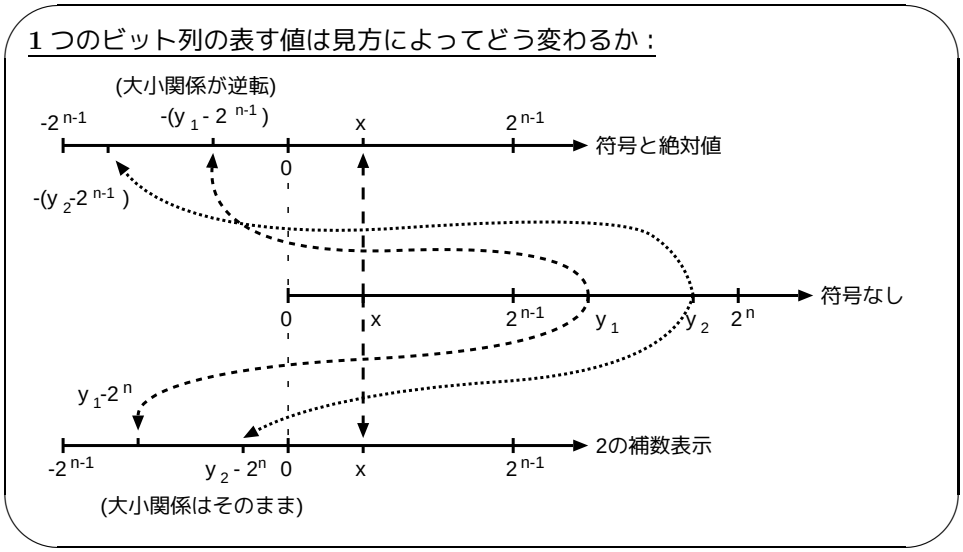
- **符号と絶対値による方法：** 最上位の 1 ビットで数値の符号を，残りの $n-1$ ビットで数値の絶対値を表す方法で、我々人間の表現法と本質的には同じものである。 表す整数データが正の場合に「非負整数データの内部表現」と同じにするために、普通、最上位ビットが 0 なら + 符号を, 1 なら − 符号を表す。例えば、10 進数 26 は 2 進法で 11010 と表されるから、$n=8$ の場合整数 $26, -26$ は計算機内部でそれぞれ 00011010, 10011010 と表される。 従って、この方法では $-(2^{n-1}-1)$〜$2^{n-1}-1$ の間の整数を表現可能であり、$a_{n-1}a_{n-2}\cdots a_1a_0$ なるビット列によって
$$(-1)^{a_{n-1}} \times (a_{n-2}\times 2^{n-2} + a_{n-3}\times 2^{n-3} + \cdots + a_1\times 2^1 + a_0\times 2^0)$$
すなわち
$$(-1)^{a_{n-1}} \times \textstyle\sum_{i=0}^{n-2} a_i 2^i$$
という整数を表す。

- **2 の補数による方法：** ほとんど全ての計算機で採用されている方法であり、表す整数 x が 0〜$2^{n-1}-1$ の間の非負整数なら x の 2 進表現 (の左側に必要なだけ 0 を埋めて n ビットに拡張したもの) を, そして x が -2^{n-1}〜-1 の間の負整数なら x の **2 の補数** (2's complement) 2^n+x の 2 進 (非負整数) 表現を内部表現とする。 例えば $n=8$ の場合、-26 の 2 の補数 2^8-26 は 2 進法で $2^8-26 = (11111111+1) - 00011010 = (11111111 - 00011010) + 1 = 11100101 + 1 = 11100110$ と計算できるから、整数 $26, -26$ は計算機内部でそれぞれ 00011010, 11100110 と表される。

  ここで、$0 \le x \le 2^{n-1}-1$ の時は $x < 2^{n-1}$ より最上位ビットは 0 になり、$-2^{n-1} \le x \le -1$ の時は $2^n+x \ge 2^{n-1}$ より最上位ビットは 1 になる。それゆえ、逆にビット列 $a_{n-1}a_{n-2}\cdots a_1a_0$ が与えられた時、このビット列の表す値は、
$$\begin{aligned}
a_{n-1}=0 \text{ の時}\quad & a_{n-1}\times 2^{n-1} + a_{n-2}\times 2^{n-2} + \cdots + a_1\times 2^1 + a_0\times 2^0 \\
& = a_{n-2}\times 2^{n-2} + \cdots + a_1\times 2^1 + a_0\times 2^0, \\
a_{n-1}=1 \text{ の時}\quad & -2^n + (a_{n-1}\times 2^{n-1} + a_{n-2}\times 2^{n-2} + \cdots + a_1\times 2^1 + a_0\times 2^0) \\
& = -2^n + (1\times 2^{n-1} + a_{n-2}\times 2^{n-2} + \cdots + a_1\times 2^1 + a_0\times 2^0) \\
& = -1\times 2^{n-1} + (a_{n-2}\times 2^{n-2} + \cdots + a_1\times 2^1 + a_0\times 2^0)
\end{aligned}$$
  すなわち
$$-a_{n-1}\times 2^{n-1} + \textstyle\sum_{i=0}^{n-2} a_i 2^i$$
  という整数を表す。 もちろん、この方法では -2^{n-1}〜$2^{n-1}-1$ の間の整数を表現可能である。

  また、
$$\begin{aligned}
& -\left(-a_{n-1}\times 2^{n-1} + \textstyle\sum_{i=0}^{n-2} a_i 2^i\right) \\
& = a_{n-1}\times 2^{n-1} - \textstyle\sum_{i=0}^{n-2} a_i 2^i \\
& = a_{n-1}\times 2^{n-1} + \textstyle\sum_{i=0}^{n-2} (1-a_i) 2^i - \sum_{i=0}^{n-2} 2^i \\
& = a_{n-1}\times 2^{n-1} + \textstyle\sum_{i=0}^{n-2} (1-a_i) 2^i - (2^{n-1}-1) \\
& = -(1-a_{n-1})\times 2^{n-1} + \textstyle\sum_{i=0}^{n-2} (1-a_i) 2^i + 1
\end{aligned}$$

であるから、この表現法の下で数値の符号を反転するには、単に各ビットを反転して 1 を加えるだけでよい。



- **1 の補数による方法：** 表す整数 x が 0〜$2^{n-1}-1$ の間の非負整数なら x の 2 進表現 (の左側に必要なだけ 0 を埋めて n ビットに拡張したもの) を, そして x が $-(2^{n-1}-1)$ 〜 -0 の間の負整数なら x の **1 の補数** (1's complement) 2^n-1+x の 2 進 (非負整数) 表現を内部表現とする。 例えば $n=8$ の場合、-26 の 1 の補数 2^8-1-26 は 2 進法で $(2^8-1)-26 = 11111111-00011010 = 11100101$ と計算できるから、整数 $26, -26$ は計算機内部でそれぞれ 00011010, 11100101 と表される。

  2 の補数で負数を表す方法と同様に考えると、この方法では $-(2^{n-1}-1)$〜$2^{n-1}-1$ の間の整数を表現可能であり、$a_{n-1}a_{n-2}\cdots a_1a_0$ なるビット列によって
  $$-a_{n-1}\times(2^{n-1}-1)+\sum_{i=0}^{n-2} a_i 2^i$$
  という整数を表すことが分る。 また、この表現法の下で数値の符号を反転するには、単に各ビットを反転するだけでよいことも分る。

□**演習 3.1** 1 の補数で負数を表す場合、次の 2 つの事柄を確認せよ。

(a) $a_{n-1}a_{n-2}\cdots a_1a_0$ なるビット列によって
$$-a_{n-1}\times(2^{n-1}-1)+\sum_{i=0}^{n-2} a_i 2^i$$
という整数を表す。

(b) 数値の符号を反転するには、単に各ビットを反転するだけでよい。

2 の補数表現が採用される理由: 他に比べて演算機構が簡単になる。例えば、

- n ビットの加算については
  ① 2 つのビット列を符号なし 2 進数として加算して、下位 n ビットだけを取る。その結果、
  ② もし、得られた結果の最上位ビットが元の 2 つの最上位ビットのいずれとも異なるなら **桁あふれ** (overflow) と判断され、さもなければ ①の結果が加算結果となる。
- 減算回路は、加算回路と符号反転回路を組み合わせて実現できる。

3.5 コンピュータ内での実数データの表現

実数データを計算機内部で記憶する際の容量の標準単位としては、整数データの場合と同じ様に語が用いられるが、特に高精度の計算を行う場合のために倍長語 (や 4 倍長語) も普通用意されている。

実数データの内部表現としては、整数データの場合と異なり、様々な方法が考えられ使用されてきた。次にその例を 3 つ与える。

- **IBM メインフレームでの表現法：** 1 語/32 ビットのビット列

  $$\underbrace{s}_{\text{符号部}}\ \underbrace{e_6e_5\cdots e_1e_0}_{\text{指数部}}\ \underbrace{d_1d_2\cdots d_{23}d_{24}}_{\text{仮数部}}$$

  によって

  $$(-1)^s\times M\times 16^E$$
  $$\text{但し}\quad M=\sum_{i=1}^{24} d_i2^{-i}$$
  $$E=\sum_{i=0}^{6} e_i2^i-64$$

  という実数を表す。

- **日本電気メインフレーム (ACOS) での標準的表現法：** 1 語/36 ビットのビット列

  $$\underbrace{e_7e_6e_5\cdots e_1e_0}_{\text{指数部}}\ \underbrace{d_0d_1d_2\cdots d_{26}d_{27}}_{\text{仮数部}}$$

  によって

  $$M\times 16^E$$
  $$\text{但し}\quad M=-d_02^0+\sum_{i=1}^{27} d_i2^{-i}$$
  $$E=-e_72^7+\sum_{i=0}^{6} e_i2^i$$

  という実数を表す。 (仮数部も指数部も 2 の補数によって負数を表している。)

- **IEEE 規格 754 での表現法：** 単精度、倍精度、4 倍精度における指数部、仮数部のビット数等は次の様に定められている。

	符号部	指数部	仮数部	全部で
単精度	1 ビット	8 ビット	23 ビット (10 進で 6〜7 桁)	32 ビット
倍精度	1 ビット	11 ビット	52 ビット (10 進で 15〜16 桁)	64 ビット
4 倍精度	1 ビット	15 ビット	112 ビット (10 進で 33〜34 桁)	128 ビット

  特に単精度の場合は、32 ビットの列

  $$\underbrace{s}_{\text{符号部}}\ \underbrace{e_7e_6\cdots e_1e_0}_{\text{指数部}}\ \underbrace{d_1d_2\cdots d_{22}d_{23}}_{\text{仮数部}}$$

  によって、

  $$\begin{cases} (-1)^s\times(1+M)\times 2^E & \text{if } -127<E<128 \\ (-1)^s\times M\times 2^{E+1} & \text{if } E=-127 \\ \texttt{Inf}(\text{無限大}) & \text{if } E=128, M=0 \\ \texttt{NaN}(\text{非数,Not a Number}) & \text{if } E=128, M\neq 0 \end{cases}$$

  という実数を表す。但し、

$M = \sum_{i=1}^{23} d_i \times 2^{-i}$
$E = \sum_{i=0}^{7} e_i \times 2^i - 127$

補足：
大抵の場合 $-127 < E < 128$ で、上記ビット列の表す実数は
$(-1)^s \times (1+M) \times 2^E$
$= (-1)^s \times \left(1 + \sum_{i=1}^{23} d_i \times 2^{-i}\right) \times 2^E$
ということになる。 それゆえ、この場合、仮数部から d_0=1 というビットが省かれていると暗に仮定し、
$1.d_1 d_2 \cdots d_{22} d_{23}$
という 2 進小数を仮数部が表すと考えられる。

実数計算における誤差の必然性 ($1 \div 10 \times 10 \neq 1$?): 実数データを扱う場合には、常に
記憶された数値が表そうとした数値の近似にすぎない
ことに注意しなければならない。例えば、10 進数 1 と 10 は $1 = 2^0 \times 1.0$(2 進小数), 10(10 進数) = 1010(2 進数) $= 2^3 \times 1.010$(2 進小数) と変形できるから、上記の IEEE 規格 754(単精度) の表現法ではそれぞれ X'3F800000' と X'41200000' と表される。これら 2 つの 2 進数の間で割り算 $(2^0 \times 1.0) \div (2^3 \times 1.010)$ を行うと、下の補足に示す様に $0.0\dot{0}01\dot{1}$(2 進小数) $= 2^{-4} \times 1.\dot{1}00\dot{1}$(2 進小数) という循環小数が得られるから、IEEE 規格 754(単精度) の表現法では 10 進実数の割算 $1 \div 10$ の結果は X'3DCCCCCC' と表される。ここで誤差が生じる。

更に、この誤差付きの除算結果
0.0001 1001 1001 1001 1001 1001 100(2 進小数) = 0.1999998(16 進小数)
に 10(10 進数) = 1010(2 進数) = A(16 進数) を掛けると、下の補足に示す様に
0.FFFFFF0(16 進小数)=$2^{-1} \times$1.FFFFFE(16 進小数)
が得られるから、IEEE 規格 754(単精度) の表現法では、10 進の式 $1 \div 10 \times 10$ を実数計算した結果は X'3F7FFFFF' と表される。 従って、実数表現の世界では、10 進の式 $1 \div 10 \times 10$ の計算結果は 10 進数 1 と等しくはならない。

補足 ($1 \div 10$)：

$$
\begin{array}{r}
0.0\dot{0}01\dot{1}0011 \\
1010 \overline{)\ 1.0000\ \vdots} \\
\underline{1010}\ \vdots \\
1100\ \vdots \\
\underline{1010}\ \vdots \\
1000
\end{array}
$$

補足 (16 進の掛け算 0.1999998×A)：

$$
\begin{array}{r}
0.1999998 \\
\times \qquad\quad A \\
\hline
0.FFFFFF0
\end{array}
\qquad
\begin{cases}
1 \times A = \ A & (16 \text{ 進の「九九」}) \\
9 \times A = 5A & (16 \text{ 進の「九九」}) \\
8 \times A = 50 & (16 \text{ 進の「九九」})
\end{cases}
$$

3.6 まとめ

データを表すビット列自体の中には、そのデータが文字列であるか、整数であるか、実数であるか、..... の情報は入っていない。

⇨ そのデータを処理する側では、
そのデータがある内部表現方式に従うものと見なして処理を進める。

□演習 3.2 (1999 年度定期試験問題)

(0) ビット列 0110 0100 0100 1110 を 16 進表記で表せ。
(1) (半角) 文字の並び 38 は JIS 8 ビット符号体系ではどんなビット列で表されるか？ 16 進表記で答えよ。
(2) 10 進整数の 38 は 8 ビットの整数データとしてどの様に表されるか？ 2 進表記で答えよ。
(3) 10 進整数の −38 は 2 の補数表示により 8 ビットの整数データとしてどの様に表されるか？ 2 進表記で答えよ。
(4) 10 進で表された実数値 38.0 は IEEE 規格 754 の単精度実数としてどの様に表されるか？ 2 進表記で答えよ。

□演習 3.3 (2000 年度定期試験問題)

(1) ビット列 0110 0100 0100 1010 を 16 進表記で表せ。
(2) ビット列 0110 0100 0100 1010 が JIS8 ビット符号体系の文字列を表しているとすると、何の文字列を表しているか？
(3) ビット列 0110 0100 0100 1010 が符号付き整数を表しているとすると、 何の整数を表しているか？
(4) ビット列 0110 0100 0100 1010 が符号付き整数を表していると見て、その正負の符号を反転するとどんなビット列になるか？ 但し、ここでは、負数は 2 の補数で表すとする。
(5) 10 進で表された実数値 3.0 は IEEE 規格 754 の単精度実数としてどの様に表されるか？ 2 進表記で答えよ。
(6) プログラム内蔵方式とは何か？

＜第 3~ 14 週＞ C プログラミング入門

4 プログラミング序論

- アルゴリズムとは何か?
- アルゴリズムの記述,
- アルゴリズム設計の重要性

4.1 アルゴリズムとは何か?

コンピュータは与えられた「アルゴリズム」に従って動く。 では、一体アルゴリズムとは何なのか? 手元の Oxford Paperback Dictionary(1979) は「アルゴリズム」(algorithm) を

a process or rules for calculating something, especially by machine

と説明し、また、岩波情報科学辞典 (1990) は「アルゴリズム」について

一般的な用語としては、問題を解くための計算法を意味する

と述べている。 こう書くと難しそうだが、要は「... を計算する方法」といった類のものがアルゴリズムに相当するわけで、小中学校で習った

- 三角形の面積を計算する方法,
- 二次方程式の解を求める方法

といったものは全てアルゴリズムになる。

例 4.1 (二次方程式の解を求めるアルゴリズム) 二次方程式 $ax^2 + bx + c = 0$ $(a \neq 0)$ が与えられた時、その解は次の式で求めることが出来る。

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

4.2 アルゴリズムの記述

どんな風にアルゴリズムを記述すべきか:

- コンピュータは、我々人間が与えた

  最初に何をして, 次に何をして,... という動作指示の列

  を逐次的に実行しているだけである。それゆえ、コンピュータに思い通りの計算を行わせたければ、その計算アルゴリズムを明らかにするだけでなく、最初に何をして, 次に何をして, ... という計算 (処理) の手順を明示的に記述する必要がある。

補足：
処理の手順を明示的に記述するというのは、プログラミングに限ったことではない。 普通に生活をしていても作業手順を明示したものは時々見かける。例えば、「ハウス バーモントカレー」の箱の裏にはカレーの作り方が次の様に示されている。

① 厚手の鍋にサラダ油を熱し、一口大に切った肉、野菜をよくいためます。
② 水を加え、沸騰したらあくを取り、材料が柔らかくなるまで弱火～中火で煮込みます。
③ いったん火を止め、ルウを割り入れて溶かし、再び弱火でとろみがつくまで煮込みます。

- コンピュータはその内部に格納された所在のはっきりしたデータだけを計算 (処理) に使うことが出来るから、**コンピュータの基本動作を指示する際は、その動作で扱うデータの所在を明らかにする必要がある。**

例 4.2 (二次方程式を解くアルゴリズム, コンピュータ向きの記述) 例 4.1 で二次方程式を解くアルゴリズムを示したが、これをそのままコンピュータで処理することは出来ない。その第 1 の理由として、日本語で書かれた記述をコンピュータが理解できないことを挙げることも出来るが、その他に、

例 4.1 に示されたアルゴリズムの記述では
コンピュータが何を行うかが明確になっていない

という理由もある。実際、例 4.1 のアルゴリズムに従った計算をコンピュータに行わせようとすると、次の様な問題が出て来る。

- $\frac{-b\pm\sqrt{b^2-4ac}}{2a}$ という計算をすることははっきりしているが、その中で使われる a, b, c のデータはどこから得られるのか?
- 計算しただけでは、その結果を我々が見えないのではないか?

これらの点を明らかにした、コンピュータ向きのアルゴリズム記述を次に示す。

二次方程式を解くアルゴリズム (a)

(step1) 二次方程式の 3 つの係数の値を (キーボードから) 受け取り、それらのデータを各々 [a], [b], [c] という名前の付いた場所に格納する。
(以後、[a]に格納された値は 0 でないと仮定して計算を進める。)

(step2) $\dfrac{-(\boxed{b}\text{内の値})+\sqrt{(\boxed{b}\text{内の値})^2-4(\boxed{a}\text{内の値})(\boxed{c}\text{内の値})}}{2(\boxed{a}\text{内の値})}$ の計算をしてその結果を [x1] という名前の付いた場所に格納する。

(step3) $\dfrac{-(\boxed{b}\text{内の値})-\sqrt{(\boxed{b}\text{内の値})^2-4(\boxed{a}\text{内の値})(\boxed{c}\text{内の値})}}{2(\boxed{a}\text{内の値})}$ の計算をしてその結果を [x2] という名前の付いた場所に格納する。

(step4) ([x1]内の値), ([x2]内の値) をコンピュータの画面に表示する。

あるいは、略記的に

二次方程式を解くアルゴリズム (a')

(step1) 二次方程式の 3 つの係数の値を入力して、各々 a, b, c という場所に格納する。
(以後、$a\neq 0$ と仮定して計算を進める。)

(step2) $x1 \longleftarrow \dfrac{-b+\sqrt{b^2-4ac}}{2a}$

(step3) $x2 \longleftarrow \dfrac{-b-\sqrt{b^2-4ac}}{2a}$

(step4) $x1, x2$ の値を出力

コンピュータが虚数を基本データとして取り扱うことが出来ない場合は、更に手直しが必要である。 例えば、次の通り。

二次方程式を解くアルゴリズム (b)

(step1) 二次方程式の 3 つの係数の値を入力して、各々 a, b, c という場所に格納する。
(以後、$a\neq 0$ と仮定して計算を進める。)

(step2) $D \longleftarrow b^2-4ac$

(step3) $D \geq 0$ かどうかによって、次の (3.1), (3.2) のいずれかを実行する。

(3.1) $D \geq 0$ なら、

① $x1 \longleftarrow \dfrac{-b+\sqrt{D}}{2a}$

② $x2 \longleftarrow \dfrac{-b-\sqrt{D}}{2a}$

③ “実根を持つ” という表示を添えて、$x1, x2$ の値を出力

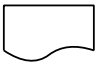
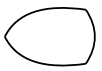
(3.2) $D < 0$ なら、

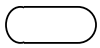
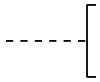
① $re \longleftarrow \dfrac{-b}{2a}$

② $im \longleftarrow \dfrac{\sqrt{-D}}{2a}$

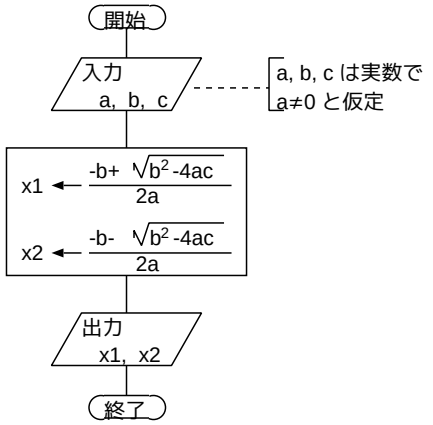
③ “虚根を持つ” という表示を添えて、実部 re と虚部 im の値を出力

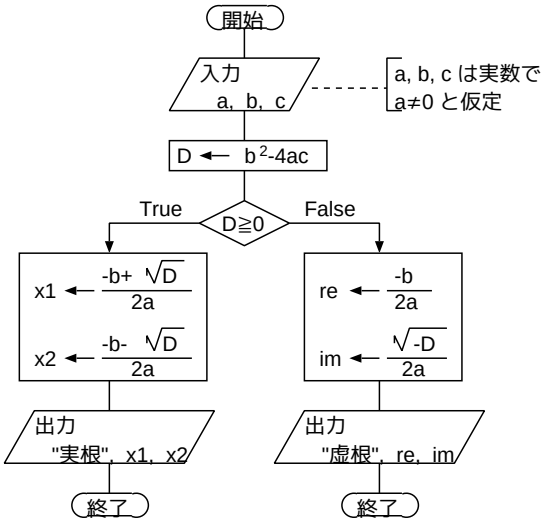
流れ図: アルゴリズムを視覚的に分かり易く表示するために、p.9 の中程 に例示された様な形の、**流れ図** (flowchart) と呼ばれる図が、コンピュータ出現直後の 1947 年頃から用いられている。 流れ図は処理内容の書き込まれた  ,  ,  , ... といった形の記号を繋げて構成される。各々の記号の意味は次の通りである。

記号の分類		記号	名称	説明
データ記号	基本データ記号		データ (data)	媒体を指定しないデータ (の入出力) を表す。
	個別データ記号		書類 (document)	人間の読める媒体上のデータ (の入出力) を表す。媒体の例としてはプリンタの出力。
			手操作入力 (manual input)	キーボードなどによる手操作入力、またはこれらの入力によるデータを表す。
			表示 (display)	人が利用する情報を表示する媒体 (e.g. ディスプレイ画面) 上のデータ、またはそれらのデータの出力を表す。
処理記号	基本処理記号		処理 (process)	任意の種類の処理機能を表す。
	個別処理記号		定義済み処理 (predefined process)	サブルーチンやモジュールなど、別の場所で定義された 1 つ以上の演算または命令群からなる処理を表す。
			判断 (decision)	1 つの入口と幾つかの択一的な出口を持ち、記号中に定義された条件の評価に従って唯一の出口を選ぶ、判断機能またはスイッチ形の機能を表す。
線記号	基本線記号		線 (line)	データまたは制御の流れを表す。標準的な流れの方向は左から右, 上から下であって、それ以外の時、または見易さを強調する時は矢先を付ける。
特殊記号			結合子 (connector)	同じ流れ図中の他の部分への出口または他の部分からの入口を表したり、線を中断し他の場所に続けたりするのに用いる。 対応する結合子は同一で一意の名前を含まなければならない。

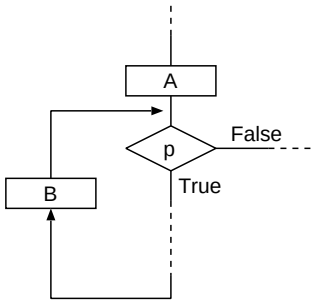
記号の分類	記号	名称	説明
特殊記号		端子 (terminator)	外部環境への出口、または外部環境からの入口を表す。
		注釈 (annotation)	明確にするために説明または注を付加するのに用いる。注釈記号の破線は関連する記号に付けるか、または記号群を囲んでも良い。説明または注は範囲を示す記号の近くに書く。

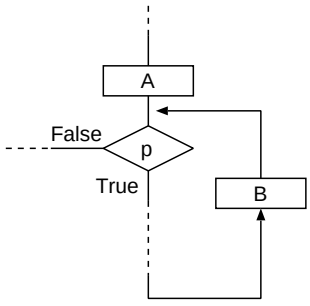
例 4.3 (二次方程式を解くアルゴリズムを流れ図で) 例 4.2 で示した二次方程式を解くアルゴリズム (a'), (b) を流れ図で表すとそれぞれ次の左図, 右図の様になる。





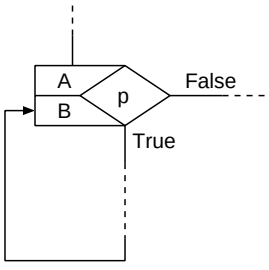
流れ図の拡張: 流れ図では、処理の繰り返しを表す手段が特別に用意されていない。この欠点を補うために拡張表記を個別に導入することもある。例えば、

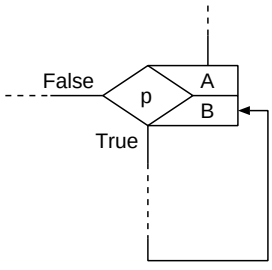




,

の処理を表すのにそれぞれ次の様な略記法を導入したりする。





,

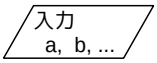
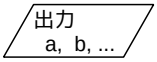
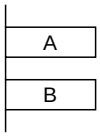
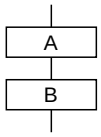
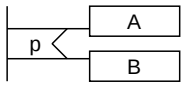
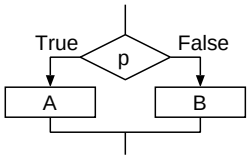
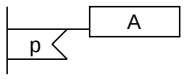
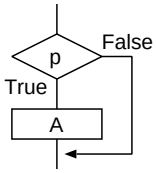
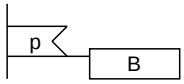
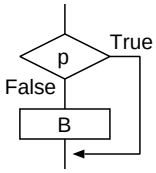
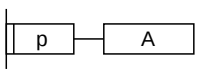
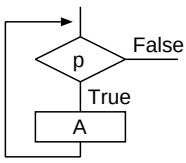
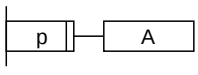
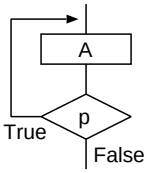
この  という形の箱を**繰り返しの箱**と呼ぶ。

流れ図に代わる表現形式: 流れ図は次の様な欠点を持つ。

- コンピュータへの動作指示 (プログラム) は文字列で表すことになるので、結局は一次元的な構造をしている。 これに対して流れ図では箱と箱を自由に線で結ぶことが出来るため、流れ図は二次元的な処理の流れを記述することができ、処理の流れが複雑になる可能性がある。それゆえ、流れ図で記述されたアルゴリズムをプログラムの形に直しても、見易いものができるとは限らない。
- 流れ図では処理の繰り返しを表す手段が特別に用意されていない。

⇨ 流れ図に代わる表現形式が色々と提案された。例えば、PAD(problem analysis diagram, 日立), NS チャート (Nassi-Schneiderman chart), HCP (hierarchical compact chart) など。

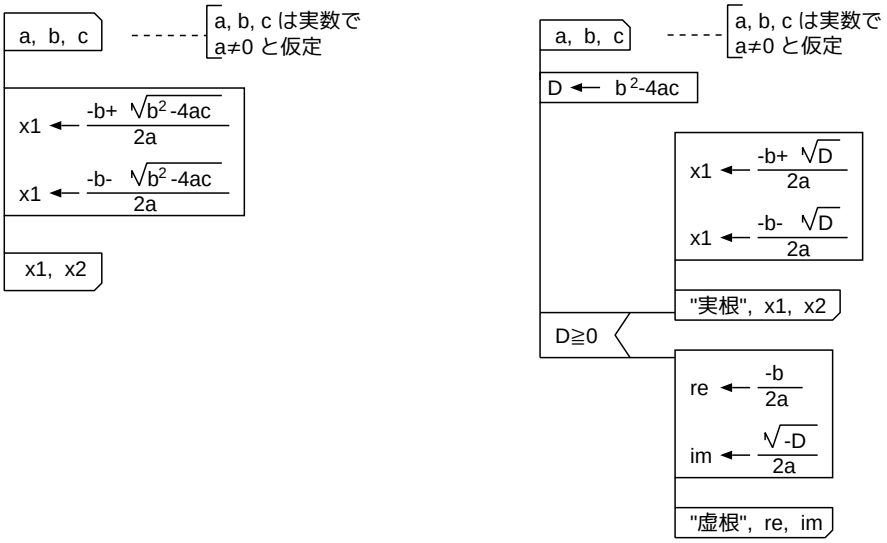
PAD(problem analysis diagram): PAD においては、処理の合成の仕方として 連接, 分岐, 繰り返し の 3 種類だけを用いて全ての処理の流れを表す。 連接, 分岐, 繰り返し、およびデータの入出力を次のように表す。

	PAD における表し方	同等の流れ図表現
入力		
出力		
連接		
分岐		
		
		
繰り返し		
		

補足 (処理手順の基本構造)：
PAD において 連接, 分岐, 繰り返し という 3 種類の処理合成の仕方しか用いないのは、次の理由による。

- 入り組んだ処理の流れは分かりにくい。
- 大抵のアルゴリズムは、これら 3 つの処理合成を組み合わせることによって表せる。
- 作り上げたアルゴリズムを実際にコンピュータに実行させる際には、アルゴリズムをコンピュータが理解できる「プログラム」の形に書き表さなければならない。その際 C 言語, Pascal, Fortaran 等の (高級) 手続き型プログラミング言語を使うことがほとんどであるが、これらの言語では 連接, 分岐, 繰り返し という 3 種の処理合成を組み合わせてアルゴリズムを記述するのが基本となっている。

例 4.4 (二次方程式を解くアルゴリズムを PAD で) 例 4.2 で示した二次方程式を解くアルゴリズム (a'), (b) を PAD で表すとそれぞれ次の左図, 右図の様になる。



4.3 アルゴリズム設計の重要性

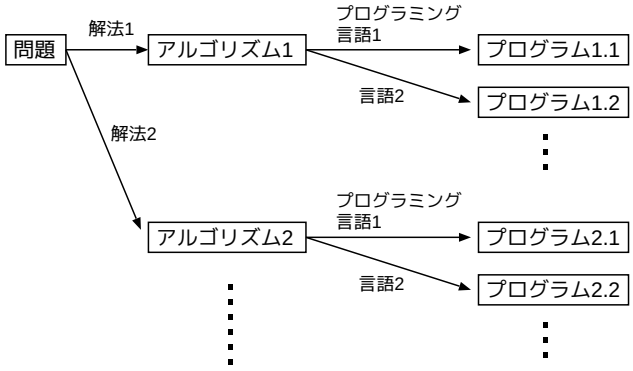
アルゴリズムの設計: コンピュータを用いて問題を解くとき、コンピュータが理解できる言語でアルゴリズムを細かく記述したものが**プログラム** (program) である。当然、悪い (e.g. 分かりにくい, 手間のかかる) アルゴリズムからは悪いプログラムしかできない。 それゆえ、プログラムを作成する上で最も大切なことは、

問題を分析して、
良い (e.g. 分かり易い, 速い) アルゴリズムを設計する

ことである。 1 つの問題に対して様々なアルゴリズムが考えられようが、その内で良いアルゴリズムを見つけることが大切である。

プログラミング言語の選択: また、アルゴリズムが決まったとしても、それをプログラムとして記述するために用いる言語 (**プログラミング言語**, programming language, と言う) も一意に決まる訳ではない。 これまで色々な利用目的に対して色々なプログラミング言語が設計され、そのうちの幾つかは現在も利用されている。これらの中から問題やアル

ゴリズムに合ったものを選ぶことも大切である。



補足：

- どのプログラミング言語を選ぶかで、プログラムが単純になったり複雑になったりする。 このことから、与えられた問題に対してまず適切なプログラミング言語を選び、次にこの言語に依存した形でアルゴリズムを記述することがある。 しかし、一般には、アルゴリズムはどのプログラミング言語にも依存しない形で記述することが望ましい。
- 実際には、使用するコンピュータ環境で全てのプログラミング言語が使える訳ではないので、言語の選択肢はそれほど多くはない。

5　整数計算の簡単なプログラム例

- 決められた文字列の出力,
- 四則演算,
- 付録 C プログラムの基本的な形
- 付録 プログラムのコンパイルと実行,
- 付録 書式付き入出力 —printf と scanf 関数—

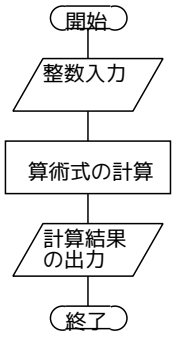
この節では、右の流れ図で表される様な、

① 整数データの入力,

② それを基に算術式計算,

③ 計算結果の出力

を順に行うタイプのアルゴリズムがC プログラムとしてどの様に記述されるのかを見てみよう。



5.1　決められた文字列の出力

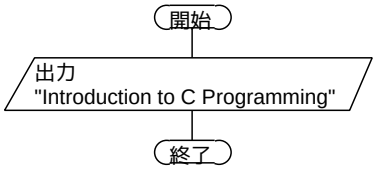
まず手始めに、データ入力も算術式計算もなく、決められた文字列を出力するだけのプログラムを例示しよう。

例題 5.1 (決められた文字列を出力)　単に

```
Introduction to C Programming
```

と出力するだけのC プログラムを作成せよ。

この処理のためのアルゴリズムは至って簡単で、右図の通りである。この処理を行うC プログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)




```
[motoki@x205a]$ nl output-constant-string.c Enter..... (ファイル表示)
     1 /* 決められた文字列 "Introduction to C Programming" */
     2 /* を出力するだけのCプログラム                        */

     3 #include <stdio.h>

     4 int main(void)
     5 {
     6   printf("Introduction to C Programming\n");
     7   return 0;
     8 }
[motoki@x205a]$ gcc output-constant-string.c Enter..... (コンパイル)
[motoki@x205a]$ ./a.out Enter..... (実行)
```

```
Introduction to C Programming
[motoki@x205a]$
```

ここで、

- "`[motoki@x205a]$ `" はコマンドラインに表示されるプロンプト (i.e. キーボードからの入力を促す文字列) である。

> 注意：
> この講義ノートで提示するプログラム実行はほとんどが実習室外の計算機によるものです。 そのため、プログラムによっては実行結果が実習室でのものと少し違うこともあり得ます。

- `nl` は文書ファイルを行番号付きで表示するためのコマンドです。行番号はプログラムの各行を説明するために付けただけで、実際のプログラムは次の部分になります。

```
/* 決められた文字列 "Introduction to C Programming" */
/* を出力するだけのCプログラム                       */

#include <stdio.h>

int main(void)
{
  printf("Introduction to C Programming\n");
  return 0;
}
```

- プログラムは 100 文字程度以内の行を縦に並べて表示されることが多いので、プログラムは 2 次元的な構造を持つように見える。 しかし、C プログラムは 1 次元的な文字の並びとして読み込まれ処理されるだけである。 実際、# で始まる行、2 重引用符 (") で囲まれた部分を除いて、空白 (半角), Tab文字, 改行文字 が多数連なっていてもそれらは両側を区切る働きしかしない。 それゆえ、例えばプログラムの 4〜8 行目の部分は、1 行にまとめて

  `int main(void){printf("Introduction to C Programming\n");return 0;}`

  と書いても実行結果は変わらない。

> 字下げ (indent)：
> では、なぜ余分な 半角空白, Tab文字, 改行文字 をプログラムの中に挿入するのかと言うと、それは我々人間のプログラムの見易さのためである。 コンピュータにとっては 1 次元的なものであっても、コンピュータが実行するアルゴリズム/処理手順は元々は構造を持ったものであるので、その構造を 改行 や空白を用いて **2** 次元的に表す。例えば、
> ◇ 2 つ以上の基本処理を 1 行の中に詰めることはしない。
> ◇ 処理手順の中に条件分岐があれば、その分岐の構造をすぐにプログラムから読み取れるように、分岐後の処理の部分を一律に何文字分か右にずらして表示する。
> ◇ 処理の繰り返しがあれば、その繰り返し構造をすぐにプログラムから読み取れるように、繰り返し部分を一律に何文字分か右にずらして表示する。
> この「何文字分か右にずらして表示する」ことを字下げ (indent) と呼ぶ。字下げの仕方には、プログラムを書いた人がプログラムの構造をどう見ているかが反映される。
> ⟹ 字下げのちゃんと出来ていないプログラムに関しては、それを書いた人がちゃんとアルゴリズムを理解しているかどうかが疑わしくなる。
>
> 半角空白, Tab文字, 改行文字 等を総称して空白類と呼ぶ。

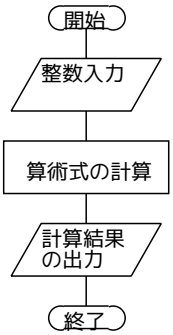
- プログラムの 1〜2 行目 は注釈。 (“/*” と “*/” で囲まれている。)
- プログラムの 4〜8 行目 はひとまとまりの処理を表す。このうち 4 行目は処理の名前等を明示した部分、5〜8 行目は処理の中身の部分である。 C 言語においては、この様な「ひとまとまりの処理」のことを**関数** (function) と呼び、それを記述した 4〜8 行目の様な部分を**関数定義**と呼ぶ。そして、この様な関数の定義を並べたもの (に何行か追加したもの) が**プログラム**になる。プログラム起動の際は `main` という名前の関数から実行を始めることになっている。
- プログラムの 4 行目 には、関数の名前が `main` であることと共に、関数の処理を実行する前に予め受け渡されるデータがないこと (”`void`” の部分)、関数の処理結果として整数データがプログラムを起動した側に報告されること (最初の ”`int`” の部分) が明示されている。
- プログラムの 3 行目 は、`/usr/include/stdio.h` というファイルの中身をこの場所に挿入してコンパイル作業を行うことを指示する。 `stdio.h` が標準に用意されている (ヘッダ) ファイルであることを示すために< と > でファイル名を囲っている。[`/usr/include/stdio.h` の中に、6 行目で呼び出される出力関数 printf の**引数**の型 (i.e. 括弧内で与えるデータの内部表現方式)、関数値の型等の情報が入っているので、それを読み込む。]
- プログラムの 6 行目 は、標準ライブラリの中に用意されている書式付き出力の関数 printf を呼び出している。2 重引用符で囲まれた部分が**出力書式**になっており、6 行目の様にこの中に%文字が現れない場合は、この部分の文字列がそのまま出力される。但し、出力書式中の `\n` は改行を意味する。
- プログラムの 7 行目 は、プログラムが正常終了したことを報告するために、`main()` 関数を起動した側に整数値 0 を戻している。(戻り値 0 が正常終了を、0 以外が異常終了を表す。)
- `gcc` は C 言語の**コンパイラ** (compiler;i.e. 人間に分かり易い言語で書かれたプログラムを機械語のプログラムに翻訳するソフトウェア) を起動するコマンドです。`-o` オプション で実行ファイルの名前を指定しなければ `a.out` という名前の実行ファイルがデフォルトで作られます。

  補足：
  実習室では `gcc` の他に `cc` コマンドも使えます。

- コマンドライン上の `./a.out` は出来上がった (`a.out` という名前の) 機械語プログラムを起動している所です。

□**演習 5.2 (文字列の出力)** ”Hello World!” と画面に出力するだけの C プログラムを作成せよ。

5.2 四則演算



ここでは、
① 整数データの入力,
② それを基に算術式計算,
③ 計算結果の出力
を順に行う C プログラムを例示する。

例題 5.3 (四則演算) 2 つの整数データを読み込み、それらの 和, 差, 積, 商, 除算の際の余り を出力する C プログラムを作成せよ。

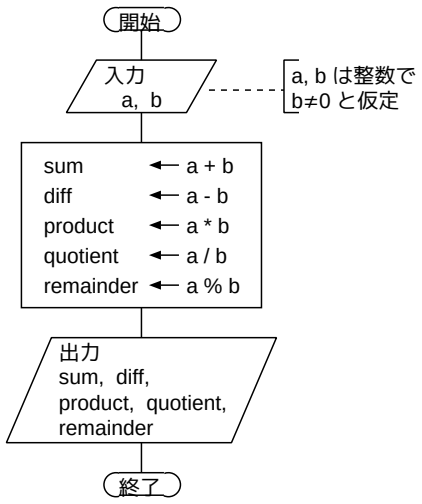
この処理のためには、(プログラムの外から) 読み込むデータを格納する場所が必要である。 そこで、 これらの記憶領域を用意し各々 a，b という名前を付け、また、

和，差，積，商，除算の際の余り

を計算した結果を格納する領域も用意しそれらに各々

sum，diff，product，quotient，
remainder

という名前を付けることにすれば、コンピュータが行うべき処理は右図の様に書き表すことができる。 ここでは、単に ①整数データの入力, ②算術式計算, ③計算結果の出力 を順に行うだけである。



この処理を行う C プログラムと、これをコンパイル/実行している様子は次に示す通りである。 (下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl arithmetic-operations-over-int.c  Enter
                                                  ..... (ファイル表示)
     1 /* 2つの整数データを変数 a と b に読み込み、それらの     */
     2 /* 和，差，積，商，除算の際の余り を出力するCプログラム */

     3 #include <stdio.h>

     4 int main(void)
     5 {
     6   int    a, b, sum, diff, product, quotient, remainder;

     7   scanf("%d%d", &a, &b);

     8   sum       = a+b;
     9   diff      = a-b;
```

```
10   product  = a*b;
11   quotient = a/b;
12   remainder= a%b;

13   printf("\nInput data: %d, %d\n\n"
14          "Sum:        %d\n"
15          "Difference: %d\n"
16          "Product:    %d\n"
17          "Quotient:   %d\n"
18          "Remainder:  %d\n",
19          a, b, sum, diff, product, quotient, remainder);
20   return 0;
21 }
[motoki@x205a]$ gcc -o arith-op arithmetic-operations-over-int.c [Enter]
                ..... (実行ファイルの名前を指定してコンパイル)
[motoki@x205a]$ ./arith-op [Enter]..... (実行)
11 3 [Enter]..... (データの入力)

Input data: 11, 3

Sum:        14
Difference: 8
Product:    33
Quotient:   3
Remainder:  2
[motoki@x205a]$
```

ここで、

- プログラムの 1〜2 行目 は注釈。
- プログラムの 3 行目 は、/usr/include/stdio.h というファイルの中身をこの場所に挿入してコンパイル作業を行うことを指示する。
- プログラムの 4〜21 行目 は main 関数の定義。
- プログラムの 6 行目 は、整数データを格納するための領域を 7 つ確保し、それぞれ a, b, sum, diff, product, quotient, remainder と名付けることをコンパイラに知らせる宣言文である。「int」と指定することにより、確保した領域が 整数データの内部表現形式 (3.4 節) に従ってデータを保持する様になる。

> **補足：**
> int の様に、内部表現形式に起因するデータの種類を一般に**データ型** (data type) と呼ぶ。 C 言語においては、int は整数を表すための最も標準的な内部表現形式に対応したデータ型である。int の他には実数データを保持するための float や double といった (基本) データ型も用意され、また、(基本) データ型を複数組み合わせて新しいデータ型を定義することもできる。⟹ 第 11 節を参照。

- **プログラミング** (programming;i.e. プログラム作成の作業) の際には、a, b, sum, ... の様に、プログラムの中でデータを記憶するために確保され使われる記憶領域のことを

一般に**変数** (variable) と呼ぶ。

> 変数の名前の付け方：
> C 言語では、変数に付ける名前として、
> 英字 (または下線) で始まり、それに英数字や下線が続く文字の並び
> を使うことが出来る。(大文字と小文字は区別する。) こういった制約の下で、**使い方/役割に応じた適切な名前を付ける**ことが大切である。⟹ p.44 を参照。
>
> 補足：
> 「変数」という用語は数学にも出て来る。プログラミングの場合も数学の場合も元々の英語の用語は “variable” で、ともに
> 使う時点によって 表す対象/内容が変わり得るもの
> という意味である。
>
> 確かに、数学で 「関数 $f(x) = x^2 + 1$」 と言った場合の変数 x は実数上で色々と変わる値を表すものだし、プログラムにおける変数 a はプログラムの実行状況に応じて色々な値を保持するもの (記憶領域) になっている。

- プログラムの7行目は標準ライブラリの中に用意されている書式付き入力の関数 scanf を呼び出している。 この呼出しの際に scanf 関数に引き渡されるデータ (**引数**, ひきすう, または**パラメータ**と言う) を指定しているのが、“scanf” に続く丸括弧 ( ) で囲まれた部分である。各々の引数はコンマで区切られているので、7 行目には全部で 3 つの引数が指定されていることになる。このうち、第 1 引数は入力の書式を表す文字列、2 番目以降の引数は入力データを格納する領域の**番地** (address) を表している。ここで指定された入力書式の中の %d は、読み込んだデータ (数字の列) を 整数 の内部表現形式に変換して、然るべき記憶領域に格納することを指示している。 また、例えば第 2 引数の &a は変数 a (が置かれる主記憶上) の番地を表す。

> 補足：
> 単に a と書いたのでは変数 a の保持する値が scanf 関数に送られてしまい、scanf 側では読み込んだデータの格納場所が分からない。

- プログラムの8行目は

  sum⟵ a + b

  すなわち

  (変数 a に保持されている値) + (変数 b に保持されている値)
  を計算して、その結果を変数 sum に格納する

  という動作を表しています。 同様に、プログラムの9〜12行目は各々

  diff⟵ a − b,
  product⟵ a × b,
  quotient⟵ a ÷ b, ...(小数部は捨てられ、結果は整数になる)
  remainder⟵ (a の値) を (b の値) で割った時の余り,

  という動作を表しています。
- プログラムの13〜19行目は、標準ライブラリの中に用意されている書式付き出力の関数 printf を呼び出している。13〜18 行目の、2 重引用符で囲まれた部分が出力書式になっていて、この指示に従った様式で関数 printf の第 2 引数以下 (19 行目) の部分で指定された式の値が順に出力される。 [13〜18 行目は途中にコンマが無いので、1 つの引数として扱われる。2 重引用符で囲まれた文字列が 6 つ並んでいるが、この部分はこれらの文字列を並べた 1 つの文字列と同等である。] これにより、結局次のような出力がなされる。但し、ここでは空白は ␣ と明示している。

空行
Input␣data:␣第 2 引数で指定された式 a の値,␣第 3 引数で指定された式 b の値
空行
Sum:␣␣␣␣␣␣␣␣第 4 引数で指定された式 sum の値
Difference:␣第 5 引数で指定された式 diff の値
Product:␣␣␣␣第 6 引数で指定された式 product の値
Quotient:␣␣␣第 7 引数で指定された式 quotient の値
Remainder:␣␣第 8 引数で指定された式 remainder の値

ここで注目すべき点は次の通りである。

◇ 出力書式中に現れる %d という部分が第 2 引数以降の式と順にペアにされ、式の値 (を我々が見える文字列で表したもの) に置き換えられる。
◇ 出力書式中の \n は改行を指示する。
◇ 出力書式中の %d は別途指定された式の値が固定小数点数型 (整数型) の内部表現形式に従っていると仮定して、これを 10 進の文字列に変換して、この%d の場所に出力することを指示する。
◇ 出力書式中の \n, %d 以外の文字列はそのまま出力される。

- gcc コマンドの次に　-o　ファイル名　という形のオプションを入れると実行ファイルの名前を指定することができる。 上のコンパイル例の場合は、これによって arith-op という名前の実行ファイルが作られている。

代入文：　“sum=a+b;” の様に、プログラムの中で

　　式の計算をして その結果を指定された変数に格納する動作を表す部分

を一般に**代入文** (assignment statement) と呼ぶ。特に C 言語においては、変数名や 123 といった数を表す文字列に 8〜12 行目の演算子や丸括弧を組み合わせて色々な 式を構成し、

変数名 = 式 ;
変数名 += 式 ; ..... (変数名 = 変数名 + 式 ; と同等)
変数名 -= 式 ; ..... (変数名 = 変数名 - 式 ; と同等)
変数名 *= 式 ; ..... (変数名 = 変数名 * 式 ; と同等)
変数名 /= 式 ; ..... (変数名 = 変数名 / 式 ; と同等)
変数名 %= 式 ; ..... (変数名 = 変数名 % 式 ; と同等)
.....

という形の代入文を書くことができる。

例 5.4 (代入文) int 型変数 sum，x に関して、次の 2 つの代入文は同じ計算を行う。

```
sum = sum + x;
sum += x;
```

算術式の計算：　(コンピュータによる) 代入文の実行においては、等号の右側の算術式の計算は通常の数学と同じ順序で 1 つずつ行われる。 すなわち、丸括弧によって計算の順序が指定されていない場合、

- 乗除算, 剰余演算 (*，/，%) を加減算 (+，-) より先に行う。
- 乗除算, 剰余演算 (*，/，%) の間では、左にあるものを優先する。
- 加減算 (+，-) の間では、左にあるものを優先する。

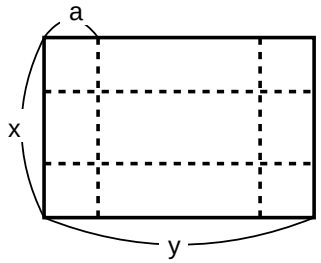
当然、途中で行われる各々の演算の結果はコンピュータの中に (一時的に) 保持されなければならず、演算結果を保持する内部表現形式も決まっているはずである。 そのために、**算術式を構成する各々の (部分) 算術式に対して、そのデータ型が定められている。** 特に int 型データ同士の演算の場合は、演算結果は int 型と定められている。

例 5.5 (算術式における演算順序の指定) int 型変数 h，m，s，time に関して、次の 4 つの代入文は同等の計算結果をもたらす。

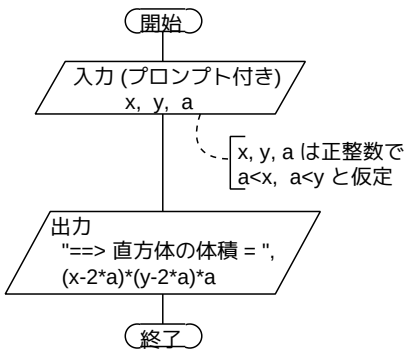
```
time = h*60*60 + m*60 + s;
time = (((h*60)*60) + (m*60)) + s;
time = (h*60 + m)*60 + s;
time = h*3600 + m*60 + s;
```

例 5.6 (式の計算に関する注意) int 型同士の除算においては商の小数部は捨てられる。そのため、数学的に等しい式であっても、C 言語の算術式としては異なる値を持つことがある。例えば、数学的には $1/2 * a = a * 1/2$ であるが、大抵の場合 2 つの代入文 x=1/2*a; と x=a*1/2; は異なる実行結果をもたらす。 実際、x=1/2*888; を実行すると変数 x には値 0 が格納され、x=888*1/2; を実行すると変数 x には 444 という値が格納される。

例題 5.7 (直方体の体積) 3 つの整数データ x, y, a を読み込み、それらを使って縦, 横が各々 $x$cm, $y$cm の厚紙の四隅を $a$cm 四方だけ切り取ってできる直方体の体積を計算して出力する C プログラムを作成せよ。



この処理のために、 厚紙の縦, 横, 切り取る正方形の一辺の長さを記憶する領域を用意し各々 x，y，a という名前を付けることにする。すると、縦が x-2*a cm, 横が y-2*a cm, 高さが a cm の直方体が出来るから、求める体積は (x-2*a)*(y-2*a)*a と計算できる。そこで、データ入力を促す文字列も出すことにし、また printf 関数を使って算術式の計算結果を変数に格納せずにそのまま出力することにすれば、コンピュータが行うべき処理は右図の様に書き表すことができる。



この処理を行うCプログラムと、これをコンパイル/実行している様子は次に示す通りである。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl rectangular-parallelepiped.c
     1 /* 3つの整数データ x, y, a を読み込み、それらを使って */
     2 /*     縦,横が各々 x,y の厚紙の四隅を a 四方だけ      */
     3 /*     切り取ってできる直方体                          */
     4 /* の体積を計算して出力するCプログラム                 */

     5 #include <stdio.h>

     6 int main(void)
     7 {
     8   int x, y, a;

     9   printf("厚紙の縦 = ? ");
    10   scanf("%d", &x);
    11   printf("厚紙の横 = ? ");
    12   scanf("%d", &y);
    13   printf("切り取る正方形の一辺 = ? ");
    14   scanf("%d", &a);

    15   printf("\n==> 直方体の体積 = %d 立方cm\n",
    16          (x-2*a)*(y-2*a)*a);
    17   return 0;
    18 }
[motoki@x205a]$ gcc rectangular-parallelepiped.c
[motoki@x205a]$ ./a.out
厚紙の縦 = ? 38
厚紙の横 = ? 57
切り取る正方形の一辺 = ? 8

==> 直方体の体積 = 7216 立方cm
[motoki@x205a]$
```

ここで、

- プログラムの1〜3行目は注釈。
- プログラムの8行目は、int型データを格納するための領域を3つ確保し、それぞれx, y, aと名付けることをコンパイラに知らせる宣言文である。
- プログラムの9行目,11行目,13行目はデータの入力を促す文字列(**プロンプト**と呼ぶ)を出力している部分である。続く10行目,12行目,14行目で、各々のプロンプトに対するデータ入力が行われる。
- プログラムの15〜16行目は、書式付き出力の関数printfを呼び出している。これにより、結局次のような出力がなされる。

空行
==> 直方体の体積 = 第 2 引数で指定された式 (x-2*a)*(y-2*a)*a の値 立方 cm

ここで注目すべき点は次の通りである。

◇ C 言語の printf 関数においては、出力データを算術式で指定することができる。

補足：
一般に、プログラムの中でコンピュータの基本動作に相当する部分を**文** (statement) と呼ぶ。特に C 言語においては、セミコロン (;) は文の終わりを表す記号である。

C 言語においては、等号 = は + や * と同じ様に**演算子** (operator) に分類され、**代入演算子**と呼ばれる。 そして、変数 = 式 というもの自身が値をもつ式として扱われる。 [変数に値が格納されるのは単なる副作用と考える。]

□**演習 5.8 (整数の商)** 9 桁以下の 2 つの正整数 m, n を入力して、

$$\frac{m}{n}\text{の小数部を切捨てて得られる整数値}$$

を出力するプログラムを作成せよ。

□**演習 5.9 (四捨五入)** 9 桁以下の 2 つの正整数 m, n を入力して、

$$\frac{m}{n}\text{の小数部を四捨五入して得られる整数値}$$

を出力するプログラムを作成せよ。

Hint：
このプログラムは、例えば $m = 7, n = 3$ なら 2 を、$m = 8, n = 3$ なら 3 を出力することになる。 そのために、実際にはプログラム内で

$$\left\lfloor \frac{m}{n} + \frac{1}{2} \right\rfloor = \left\lfloor \frac{2m+n}{2n} \right\rfloor = \left\lfloor \frac{m + \lfloor n/2 \rfloor}{n} \right\rfloor$$

の計算を int 型算術式で表せばよい。 ここで、$\lfloor x \rfloor$ は x を越えない最大整数を表す。

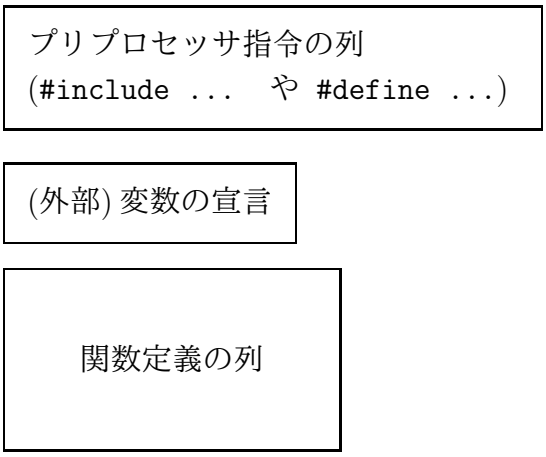
---

float 型だと、有効桁が少ないために、入力値が大きい時に正確な数値が出力されないことがある。

□**演習 5.10 (時間 → 秒)** 3 つの非負整数 h, m, s を入力して、h 時間 m 分 s 秒に相当する時間の長さを秒単位に変換して出力する C プログラムを作成せよ。

5.3 付録 C プログラムの基本的な形 —C 文法のまとめ (1)—

C プログラムの基本形式： C プログラムは次のような形をしている。



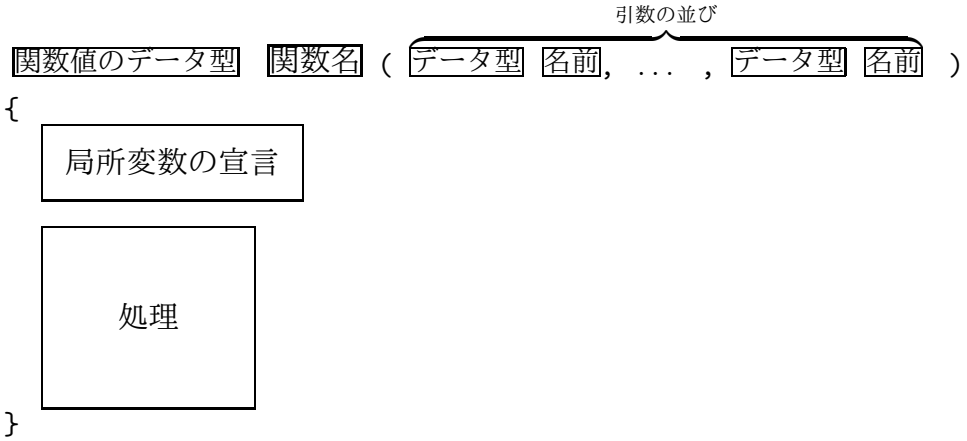
ここで、

- #で始まる行は、コンパイル (i.e. 機械語への翻訳) の前に行う作業を指示していて、**プリプロセッサ指令** (または**前処理指令**) と言う。
- 関数定義の外で変数を宣言することも出来る。

> **補足：**
> これらの変数が**外部変数**と呼ばれるのに対し、関数の中で宣言される変数は**自動変数**と呼ばれる。 外部変数が全ての関数の中から使用可能な**大域変数**として働くのに対して、自動変数は各々の宣言された関数本体の中だけで有効な**局所変数**として働く。

- プログラム内の `/*` と `*/` で囲まれた部分は注釈として扱われ、実行結果には何の影響も与えない。
- プログラム起動の際は `main` という名前の関数から実行が開始される。

関数定義の形式： C プログラムの関数定義は次のような形をしている。



ここで、

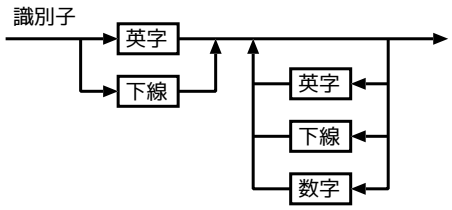
- 関数値のデータ型の部分は省略可能で、省略すると `int` と見なされる。
- 名前を表す文字列の途中を除いて、どこで改行してもよいし、どこに空白を挿入してもよい。

$\Longrightarrow$ 普通は、1 行に 2 つ以上の文を書くのを避け、各行を**字下げ**する (i.e. 書き始めの位置を右にずらす) ことによって、プログラムが見易くなるように工夫する。

変数や関数の名前の付け方： 変数や関数 (, 配列, ... など) の名前としては、

英字または下線で始まり、それに英数字または下線が続いた文字の並び

を使うことが出来る。



但し、

- 複数のものに同じ名前を付けることは出来ない。
- 英字の大文字と小文字は区別される。
- プログラムを読み易くするために、**変数や関数の役割に応じた名前を付ける**ことが大切である。
- C プログラムの中では、次の文字列 (**キーワード**と言う) は特別な役割を果たすので、変数や関数等の名前として使うことは出来ない。

```
auto      double  int       struct
break     else    long      switch
case      enum    register  typedef
char      extern  return    union
const     float   short     unsigned
continue  for     signed    void
default   goto    sizeof    volatile
do        if      static    while
```

- ANSI(American National Standards Institute) 規格の C 言語では、先頭の少なくとも 31 文字を識別することになっている。
- 下線で始まる識別子はシステムプログラムで使われたものと衝突することがある。

変数の宣言： 変数を使う時は、

データ型 変数名 , 変数名 , ... , 変数名 ;

という風に宣言する。

- 関数定義の最初に置く。(実行文の前。)
- メモリ領域の確保のため。
- 指定した演算を正しく行うため。

例えば、
整数型の加算と浮動小数点数型の加算では機械語命令コードが違うので、確保したメモリにどんな種類のデータを入れるかは処理系側が知っておかなければならない。

代入文： 変数に値をセットしたい場合は、次の様に書く。

変数等 = 式 ;

但し、

- 式 は定数、変数、関数呼出し等を演算子でつないだものである。
- **算術演算子**としては次のものがある。

算術演算子	機能
+	加算。但し、単項演算の場合は恒等変換を表す。
-	減算。但し、単項演算の場合は符号反転を表す。
*	乗算。
/	除算。
%	剰余。"a % b" は a を b で割った時の余りを表す。

- **整数定数**としては、例えば
  17 (10 進), 017 (8 進), 0x17 (16 進)
  といった表記のものを使うことが出来る。
- セミコロン (;) を付けると式が文になる。

代入演算子:

- C 言語では、代入を表す = は構文の一部ではなく演算子。
  $\Longrightarrow$ a=b+c　は式。
  セミコロンの付いた　a=b+c;　は文。
- 代入式は通常の算術式と同様に値を持っている。例えば、代入文

```
a = (b=2) + (c=3);
```

  は、次の代入文の列と同等。

```
b=2;
c=3;
a = b + c;
```

- 代入演算子には、=　だけでなく

  += 　 -= 　 *= 　 /= 　 %= 　 .....

  というのもある。一般に、
  変数等 op = 式
  は次の式と同等。
  変数等 = 変数等 op 式
  例えば、j *= k+3　は　j = j * (k+3)　と同等。

増分演算子と減分演算子:

++ 変数等　 ... 副作用として変数等の値を $+1$ する。そして、その結果を値とする。
-- 変数等　 ... 副作用として変数等の値を -1 する。そして、その結果を値とする。
変数等 ++　 ... 変数等の値を式の値とする。副作用として変数等の値を $+1$ する。
変数等 --　 ... 変数等の値を式の値とする。副作用として変数等の値を -1 する。

注釈：　/* と */ で囲まれた部分は注釈として扱われる。特に、

- 注釈を目立たせたい時には、例えば次の様な書き方をする。

```
/*
 *
 *
 */
```

注　　釈

```
/***********************************/
/*          注          釈         */
/*                                 */
/***********************************/
```

文字列定数 (文字リテラル)：

- 文字の列を 2 重引用符で囲むと**文字列定数**になる。
- 例えば、次のようなものがある。

```
"abc"
""
"a string with double quote \" within"
"a single backslash \\ is in this string"
"abc"   "def"                                  ⟵ "abcdef" と同じ。
```

5.4 付録 プログラムのコンパイルと実行

プログラムのコンパイルと実行： UNIX/Linux 上においては、Emacs 等のエディタを使って作られた C プログラムをコンパイルするには、一般に、cc や gcc といったコマンドが用いられる。例えば、prog1.c という名前の C プログラムが出来ている時、これをコンパイル・実行するには次の様にすればよい。

(例 1) `gcc prog1.c` .....(コンパイル)
`./a.out` .....(実行)

(例 2) `gcc -o prog1 prog1.c` .....(コンパイル)
`./prog1` .....(実行)

いずれの場合も、コンパイル直後にメッセージが出されたらそれはエラーメッセージで、よく読んでプログラムを修正した上で再度コンパイルする必要がある。

(⟹ 19.1 節を参照)

一般に、cc, gcc といった C 言語処理系は翻訳の前に前処理を行う。#で始まる行はその前処理で何を行うか指示をしている。

コンパイラの実際の作業手順について： 一般に、cc, gcc といった C 言語処理系は、実際には次のような手順でコンパイル作業を進める。

(1) 前処理 (**ヘッダファイル**、すなわち .h で終わるファイルの読み込み、等を行う。)

(2) プログラムを構成する文字の列を**字句**、すなわち
　　コンパイルの際に意味のある最小単位
の列に変換する。

補足：
字句には次の 6 種類がある。

キーワード	···	int, while, ...
識別子	···	変数名, 関数名, ...
定数	···	77, 12.3e+5, 'a', ...
文字列定数	···	"abc", ...
演算子	···	+, -, *, /, %, 関数名の次の括弧, ...
句切り記号	···	( ), {}, ;, ...

(3)
(4)
⋮
} 構文解析、翻訳コード生成、など

前処理作業の具体例： 前処理は C プログラム中の # で始まる行 (**前処理指令**) の指示に従って行われる。例えば、

- C プログラム中に

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

  という行があれば、プログラムのその場所に /usr/include/stdio.h というファイルの中身が挿入されたものとして、コンパイル作業が続けられる。

- C プログラム中に

  ```
  #include "mylib.h"
  ```

  という行があれば、自分で別に作成した ./mylib.h というファイルの中身がプログラムのその場所に挿入されたものとして、コンパイル作業が続けられる。

- C プログラム中に

  ```
  #define PI 3.1415926535897932
  ```

  という行があれば、それ以降は (空白等で区切られた) PI という文字列は自動的に 3.1415926535897932 という文字列に置き換えられる様になる。

- C プログラム中に

  ```
  #define square(x) ((x)*(x))
  ```

  という行があれば、それ以降は自動的に square(a) という文字列は ((a)*(a)) と置き換えられ、square(a+b) という文字列は ((a+b)*(a+b)) と置き換えられる様になる。

ヘッダファイルの中身は?：
C 言語においては、入出力を始めとした基本動作を行うために色々な関数が用意され、プログラムの中からそれらの関数を適宜呼び出す様になっている。例えば、`printf()` や `scanf()` もこういった関数で、プログラムの中で `printf( ... );` と書くことによって printf 関数の呼び出しを表している。 これらの関数は、予めコンパイルされ標準のライブラリの中に蓄えられていて、適切に呼び出されるのを待っている状態にある。ところが、コンパイラはこれらのライブラリ関数がどういう引数を取りどういう型の値を関数値とするのかについての情報を全く持っていないので、これらの情報をコンパイル時にコンパイラに知らせる必要がある。 これを行っているのが `#include <stdio.h>` 等の行である。

すなわち、標準のヘッダファイル`<stdio.h>, <stdlib.h>`, ... の中にはそれぞれの標準ライブラリ関数がどんな型のデータを引数に取りどんな型の値を返すかの情報をコンパイラに知らせるための文 (**関数プロトタイプ**と言う)、などが入っている。

#define で始まる行について：

- **マクロ定義**という。
- これを用いれば、プログラムのパラメータとなる定数、物理定数などに記号の名前 (**マクロ名** または **記号定数**という) を付け、以降のプログラム内で自由に使うことが出来る。
- 習慣的に、マクロ名には英大文字列を使う。
- マクロ定義は単に数値定数に名前を付けるためだけのものではない。 一般には、マクロ定義によってパラメータ付きの任意の文字列に名前を付けることが出来る。 例えば、「条件式」を用いて次の様なマクロ定義をすることが出来る。

```
#define  max(A, B)  ((A) > (B) ? (A) : (B))
```

  但し、この場合は注意が必要。 (`max(i++,j++)` とすると駄目。)
- マクロを定義する場合、マクロ名の右側の置換テキストは全体を丸括弧で囲むのが無難。 何故なら、例えば `#define square(x) (x)*(x)` とマクロ定義した場合は、`4/square(2)` は `4/(2)*(2)` と展開されてしまう。
- パラメータ付きマクロを定義する場合、マクロ名の右側の置換テキストにおいては各パラメータを丸括弧で囲むのが無難。何故なら、例えば `#define square(x) (x*x)` とマクロ定義した場合は、`square(z+1)` は `(z+1*z+1)` と展開されてしまう。

#include で始まる行について：

- `#include "  .h "` の形の指令は、自分で用意したヘッダファイル `./  .h` の中身を挿入することを指示している。
- `#include <  .h>` の形の指令は、標準に用意されたヘッダファイル `/usr/include/  .h` の中身を挿入することを指示している。
- ファイルの先頭に置くのが普通。
  ($\Longrightarrow$ 挿入指示のファイルを **ヘッダファイル** または **インクルードファイル** という。)
- ヘッダファイルの拡張子は習慣的に `.h` とする。
- ヘッダファイルの中に `#include` や `#define` で始まる行があってもよい。

- 標準に用意されたヘッダファイルの中には、それぞれの標準ライブラリ関数がどんな型のデータを引数に取りどんな型の値を返すかの情報をコンパイラに知らせるための文、などが入っている。

5.5 付録 書式付き出力 —printf—

{ 浦&原田付録 4, ケリー&ポール 11.2 節 }

関数 printf の構文：

- 関数 printf のデータ型は次の通り。

```
int printf( 書式 , 式 , 式 , ... );
```

- 書式 は「(データ) 変換指定」や出力したい文字 (改行コード '\n', tab コード '\t' 等も含む) を並べて、2 重引用符で囲むことによって指示する。(文字列定数になる。)
- 書式に続く 式 は、出力データを表す式である。
- **変換指定**は出力値の表示方法を指定したもので、その一般形は

  %[フラグ][最小フィールド幅][. 精度][型限定子] 変換指定子

  但し、[...] の部分はそれぞれオプションで、省略可。

  となっている。

関数 printf の実行の流れ：

- 書式 に書かれた順に出力が為される。
- 「変換指定」以外の部分はそのまま出力される。「変換指定」の部分は、第 2 引数以降から取り出された式の値を変換指定に従って文字列に置き換えて出力される。[書式と出力データの列を見比べながら処理が進む。]
- 出力が無事終了した場合は出力した文字の個数が関数値となり、エラーが発生した場合は負の値が関数値になる。

関数 printf における変換指定子： 次のような変換指定子が用意されている。

変換指定子	説 明
d	指定されたデータが int 型の内部表現形式に従っているものと見て、それを 10 進表記に変換して出力する。
i	
u	指定されたデータが unsigned int 型の内部表現形式に従っているものと見て、それを 10 進表記に変換して出力する。
o	指定されたデータが unsigned int 型の内部表現形式に従っているものと見て、それを 8 進表記に変換して出力する。

変換指定子	説 明
x	指定されたデータが unsigned int 型の内部表現形式に従っているものと見て、それを 16 進表記に変換して出力する。(x だと a〜f が、X だと A〜F が 16 進数字として使われる。)
X	
c	int 型 (または char 型) データの下位 8 ビットを文字コードに持つ文字を出力する。
f	指定されたデータが double 型の内部表現形式に従っているものと見て、それを次の形式の 10 進小数表記 (指数部無し) に変換して出力する。 [-] 数字列 . 数字列 出力指定された式が float 型の場合は、その値が double 型に変換された後に printf 関数が呼び出される。
e	指定されたデータが double 型の内部表現形式に従っているものと見て、それぞれ次の形式の指数部付きの浮動小数点表記に変換して出力する。 [-] 0 以外の数字 . 数字列 e ± 2 桁以上の数字列 [-] 0 以外の数字 . 数字列 E ± 2 桁以上の数字列 出力指定された式が float 型の場合は、その値が double 型に変換された後に printf 関数が呼び出される。
E	
g	f 変換と e(または E) 変換の変換結果のうち、短い方の文字列を出力する。
G	
s	(char *) 型の引数データの指す文字から初めて、ヌル文字'\0' が現れるまでの文字列をそのまま出力する。
p	ポインタ型引数データを番地データと見て、それを 16 進数表示で出力する。
n	文字は出力しない。引数で与えられた (int *) 型ポインタの指す領域に、 この printf 関数で出力されたそれまでの文字数 を格納する。
%	'%%' という変換指定により 1 つの%文字を出力する。

例 5.11 (s 変換指定子) s 変換を用いると (あまり好ましくありませんが) 次の様に書くことも出来る。

```
printf("%s %f, %s %f %s\n      = %f\n",
       "底面の半径が", r, "高さが", h, "の円錐の体積",
       PI*r*r*h/3.0);
```

関数 printf における型限定子： 出力データの入った領域の大きさについての認識を調節するために、次の指定が可能。

型限定子	説 明
(なし)	d または i 変換の時、引数のデータ型は int と見なされる。
	u, o, x または X 変換の時、引数のデータ型は unsigned int と見なされる。
	n 変換の時、引数のデータ型が (unsigned) int 型領域へのポインタと見なされる。
	e, E, f, g または G 変換の時、引数のデータ型は double と見なされる。
h	d または i 変換の時、引数のデータ型が short int と見なされる。
	u, o, x または X 変換の時、引数のデータ型が unsigned short int と見なされる。
	n 変換の時、引数のデータ型が (unsigned) short int 型領域へのポインタと見なされる。
l	影響の仕方は h 型限定子の場合に類似。(但し、この場合は short ではなく long int。)
L	e, E, f, g または G 変換の時、引数のデータ型が long double と見なされる。

関数 printf における最小フィールド幅の指定： 表の形に揃えて表示したい時のために、**出力フィールド** (i.e. 出力する場所) の大きさの最小値を正整数で、または星印 * で指定することが出来る。[省略も可。]

- 正整数が指定された場合は、作り出された出力文字列が指定された最小フィールド幅より大きければ、この指定は無視される。指定された最小フィールド幅より小さければ、
  - ◇ 左寄せ指定 (「フラグ」部) なら、右に空白列を補って (出力文字列の長さを指定に合わせて) 出力。
  - ◇ 右寄せ指定&最小フィールド幅が 0 以外で始まっているなら、左に空白列を補って出力。
  - ◇ 右寄せ指定&最小フィールド幅が 0 で始まっているなら、左に 0 の列を補って出力。
- 星印 * が指定された場合は、書式に続く引数の並びの中から * に対応するものが取り出され、その値が最小フィールド幅として使われる。

関数 printf における「.精度」の指定： 精度は非負整数または星印 * で指定することが出来る。[省略も可。]

- 精度が省略されピリオドだけ指定された場合は 精度=0 と見なされる。
- 非負整数が指定された場合は、
  - ◇ d, i, o, u, x または X 変換の時は、出力すべき最小の桁数を表す。(ピリオドも精度も省略された時は 精度=1 と見なされる。)

◇ e, E または f 変換の時は、小数点以下の桁数を表す。[ピリオドも精度も省略された時は精度=6 と見なされる。]

◇ g または G 変換の時は、最大有効桁数を表す。最後に続く 0 および小数点は印字されない。 [ピリオドも精度も省略された時は精度=6 と見なされる。]

◇ s 変換の時は、引数で指定された文字列の中から出力する最大文字数を表す。[ピリオドも精度も省略された時は 精度=(引数で指定された文字列の長さ) と見なされる。]

- 星印 * が指定された場合は、書式に続く引数の並びの中から * に対応するものが取り出され、その値が精度の指定値として使われる。

関数 printf におけるフラグ部の指定： 次の指定が可能。

フラグ	説 明
(なし)	右寄せ
-	左寄せ
+	d, i, e, E, f, g または G 変換の時、非負の値が + 符号付きで出力される。
空白	d, i, e, E, f, g または G 変換の時、非負の値が + 符号なしで出力される。[+も空白も指定された時は、+の方が優先される。]
#	o 変換の時、8 進数の出力の前に 0 が付く。
	x または X 変換の時、16 進数の出力の前に 0x または 0X が付く。
	e, E または f 変換の時、精度=0 という指定があっても必ず小数点が付く。
	g または G 変換の時、末尾の 0 も省略せずにちゃんと表示される。
0	出力フィールドを埋める文字として、空白ではなく 0 (ゼロ) を用いる。

□演習 5.12 (理解度チェック) フィールド幅を合わせるために、左に空白でなく 0 を埋めるにはどうするか? また、精度を 0 にして整数値 0 を出力するとどうなるか?

□演習 5.13 (可変なフィールド幅, 精度) 次のようにフィールド幅と精度を可変にしてプログラムを実行してみよ。

```
printf("x = %*.*f\n", m, n, x);
```

より詳しくは、浦&原田 (編)「C 入門」の付録 4、ケリー& ポール「C の ABC(下)」の第 11.1 節、等を参照して下さい。

5.6 付録 書式付き入力 —scanf—

{ 浦&原田付録 4, ケリー&ポール 11.2 節 }

関数 scanf の構文：

- 関数 scanf のデータ型は次の通り。

```
int scanf( 書式 , 式 , 式 , ... );
```

- 書式 は「(データ) 変換指定」や入力中に現れるはずの単語等を並べて、2 重引用符で囲むことによって指示する。
- 書式に続く 式 は、入力データを格納するための領域を指す (ポインタ型の) 式である。
- **変換指定**は文字列で表されている入力データをどのデータ型の内部表現形式に変換するかを指定したもので、その一般形は

  %[代入抑止文字] [最大フィールド幅] [型限定子] 変換指定子

  但し、[...] の部分はそれぞれオプションで、省略可。

  となっている。

関数 scanf の実行の流れ：

- 入力ストリームから取り出された個々の入力データは、順番に書式中の「変換指定」に従って内部表現形式に変換され、第 2 引数以下で指定された番地に 1 つずつ格納されてゆく。[入力ストリーム、書式、入力領域の列の 3 つを見比べながら処理が進む。]
- **関数値＝「入力に成功したデータの個数」**である。但し、途中で入力が無くなった場合は、EOF (マクロ; 普通 −1 が割り当てられている) を返す。
- 特殊な場合 (i.e. 書式の中の次の変換が%c または '[' 変換の場合) を除いて、入力ストリーム中の**空白類** (i.e. 空白、改行コード、tab コード) は入力データの区切りとして働き読み飛ばされる。
- 書式中 (「変換指定」の中を除く) に空白類が現れた場合には、入力中で次に非空白類の文字が現れるまで入力文字が読み飛ばされる。
- 書式中に「変換指定」の一部でも空白類でもない文字が現れた場合には、その文字が次の入力文字になっていなければならない。[一致しなければ、データ入力の実行は (途中であっても) 終了する。]

関数 scanf で可能な変換指定子： 次のような変換指定子が用意されている。

変換指定子	説 明
d	入力文字列を 10 進整数と見て int 型の内部表現形式に変換し、指定された記憶領域に格納する。
i	入力文字列が 0x または 0X で始まっていれば 16 進整数表記、それ以外で 0 で始まっていれば 8 進表記、それ以外なら 10 進表記の整数と見て int 型の内部表現形式に変換し、指定された記憶領域に格納する。
u	10 進整数表記の文字の並びを unsigned int 型の内部表現形式に変換し、指定された記憶領域に格納する。

変換指定子	説 明
o	8 進整数表記の文字の並びを unsigned int 型の内部表現形式に変換し、指定された記憶領域に格納する。[但し、符号付きの入力データも OK。数字部は 0 で始まっていても良い。]
x X	16 進整数表記の文字の並びを unsigned int 型の内部表現形式に変換し、指定された記憶領域に格納する。[但し、符号付きの入力データも OK。数字部は 0x や 0X で始まっていても良い。]
e E f g G	入力文字列を浮動小数点表記の実数と見て float 型 (型限定子によって double や long double に指定変更可) の内部表現形式に変換し、指定された 記憶領域に格納する。
c	「最大フィールド幅」部で指定された長さ (デフォルトは 1) の入力文字列を文字コードのまま (すなわち無変換で) 指定された記憶領域に格納する。但し、入力ストリームの途中に空白類が現れても読み飛ばさない。また、格納の際、ヌル文字 '\0' は (最後に) 付け加えられない。
s	空白類文字で区切られた入力文字列を次の入力と見て、その文字コードの列を指定された記憶領域に格納する。但し、格納の際、その文字コードの列の最後にヌル文字 '\0' を付け加える。
[文字列]	文字集合 $\Sigma = \begin{cases} \boxed{\text{文字列}}\text{に現れる文字の集合} \\ \qquad \text{if } \boxed{\text{文字列}}\text{が '^' 以外の文字で始まる} \\ \boxed{\text{文字列}}\text{に現れない文字の集合} \\ \qquad \text{if } \boxed{\text{文字列}}\text{が '^' という文字で始まる} \end{cases}$ 内の文字だけで構成される最長の文字列を入力データとして取り出し、その文字列の最後にヌル文字 '\0' を付けた文字コードの列を指定された記憶領域に格納する。
p	ポインタ型データの出力形式 (i.e.%p 変換による出力の形式; 処理系に依存) をポインタ型の内部表現に変換し、指定された記憶領域に格納する。
n	それまでに読み込まれた文字の数を指定された記憶領域に格納する。
%	次の文字が '%' であることを確認する。[単に、書式指定の中では '%%' で '%' という文字を表すということ。当然、入力ストリームで次の文字が % になっていないと、データ入力は中止 (エラー) となる。]

関数 scanf における型限定子： データを格納する記憶領域の大きさについての認識を調節するために、次の指定が可能。

型限定子	説 明
(なし)	d, i または n 変換の時、格納領域のデータ型が int と見なされる。
	u, o, x または X 変換の時、格納領域のデータ型が unsigned int と見なされる。
	e, E, f, g または G 変換の時、格納領域のデータ型が **float と見なされる。**
h	d, i または n 変換の時、格納領域のデータ型が short int と見なされる。
	u, o, x または X 変換の時、格納領域のデータ型が unsigned short int と見なされる。
l	d, i または n 変換の時、格納領域のデータ型が long int と見なされる。
	u, o, x または X 変換の時、格納領域のデータ型が unsigned long int と見なされる。
	e, E, f, g または G 変換の時、格納領域のデータ型が **double と見なされる。**
L	e, E, f, g または G 変換の時、格納領域のデータ型が long double と見なされる。

関数 scanf における最大フィールド幅の指定： 1 個の入力データを表すための最大文字数を正整数で指定できる。これが指定されていない場合は、文字数の上限は考慮されない。

関数 scanf における代入抑止文字の指定： 星印 * を指定すると、この変換指定子に対応する入力データは読み飛ばされる。

例 5.14 ([変換, 代入抑止文字) 行末までのデータを読み飛ばすには、例えば次のように書く。

```
scanf("%*[^\n]");
```

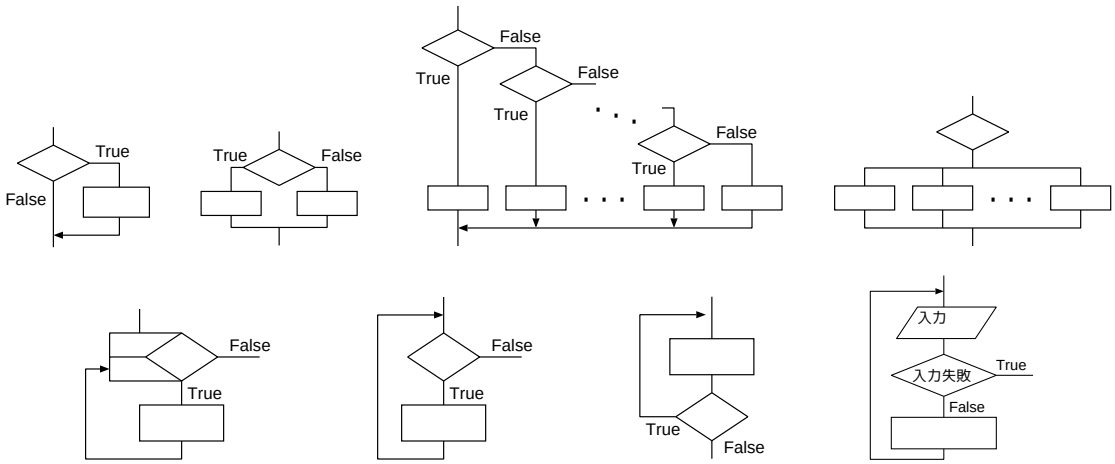
□演習 5.15 (理解度チェック) scanf の関数値は何か? また、途中の文字列を読み飛ばすにはどうするか? 指定した文字から構成される文字列を読み込むにはどうするか?

より詳しくは、浦&原田 (編)「C 入門」の付録 4、ケリー& ポール「C の ABC(下)」の第 11.2 節、等を参照して下さい。

6 処理の選択と繰り返し

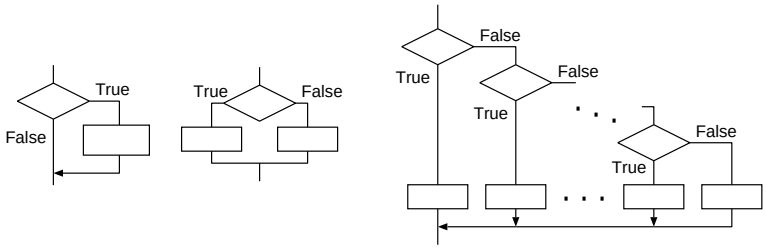
- 3 つの数の最大値 (if 文,if-else 構文, 論理演算, 条件演算子),
- 階乗 (for 文), どうやって繰り返し構造を見出すか?,
- 素数 (break 文), 最大公約数 (ユークリッドの互除法),
- 不定個の入力データの合計 (while 文),
- 元号表記→西暦表記 (switch 文),
- プログラムを組み立てられない時は ...,
- 付録制御構造のまとめ

この節では、下図の形の処理の流れがC 言語でどの様に記述されるのかを見る。



6.1 条件判断による処理の選択

まず手始めに、右図の形の処理の流れがC 言語でどの様に記述されるのかを見てみる。

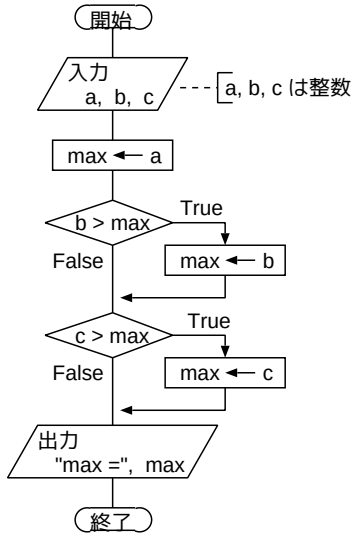


例題 6.1 (3 つの数の最大値; if 文,if-else 構文, 複合文) 整数を 3 つ読み込みその中の最大値を出力するC プログラムを作成せよ。

この、一見単純そうな問題に対しても、3 つのアルゴリズムが思い浮かぶ。以下、これらのアルゴリズムを順に説明していこう。

例題 6.1 に対するアルゴリズム (1) :

(考え方) 整数 3 つを読み込んだあとで、読み込んだ整数を順番に眺めていく。その際、常に「それまでに見た中での最大値」を保持する様にすれば、読み込んだ整数を全部眺め終った時点で、この保持データが求める最大値となっているはずである。



(プログラミング) そこで、読み込んだ整数データを格納するために a，b，c という名前の変数を、「それまでに見た中での最大値」を保持するために max という名前の変数を用意し、a，b，c の順に中のデータを眺めることにすれば、行うべき処理は右図の様に書き表すことができる。

この処理を行う C プログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl max-among-3-elem-no1.c Enter
     1  /* ３つの入力データの最大値（その１）*/

     2  #include <stdio.h>

     3  int main(void)
     4  {
     5    int  a, b, c, max;

     6    scanf("%d%d%d", &a, &b, &c);
     7
     8    max = a;
     9    if (max < b)
    10      max = b;
    11    if (max < c)
    12      max = c;

    13    printf("max = %d\n", max);
    14    return 0;
    15  }
[motoki@x205a]$ gcc max-among-3-elem-no1.c Enter
[motoki@x205a]$ ./a.out Enter
1 2 3 Enter
max = 3
[motoki@x205a]$
```

ここで、

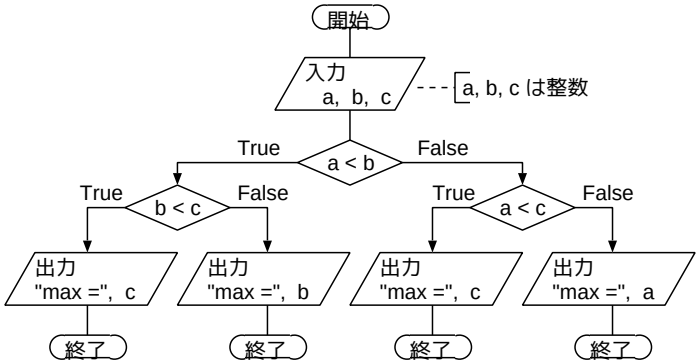
- プログラムの 9〜10 行目, および 11〜12 行目 は **if 文**と呼ばれる、条件分岐のための構文である。例えば 9〜10 行目は、括弧の中の条件「max < b」が成立すれば代入文「max = b」を実行し、成立しなければ何もしない、ということを表す。

補足：
C 言語においては、max < b といった条件を表す式の評価結果は true, false などの論理値ではなく整数値である。従って、例えば 9〜10 行目においては、実際には、条件 max < b が成立すればこの関係式の値は 1 と計算され、成立しなければ 0 と計算される。そして、この関係式の値が 1 の時だけ 10 行目の代入文が実行される。

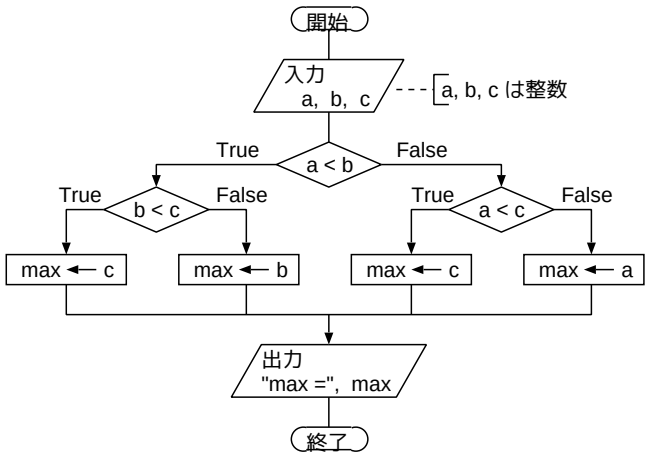
例題 6.1 に対するアルゴリズム (2)：

(考え方) 整数 3 つを読み込んだあとで、最大要素が特定できるまで、読み込んだ整数 a, b, c 間の大小関係についての場合分けを重ねる。 例えば、まず $a < b$ かどうかで場合分けする。そして、もし $a < b$ なら $b < c$ かどうかで場合分けし、もし $a \geq b$ なら $a < c$ かどうかで場合分けする。 これで、4 つの場合に分かれ、各々の場合に最大要素が特定できるので、どの場合でもあとはその値を出力するだけである。

(プログラミング) 読み込んだ整数データを格納するために a，b，c という名前の変数を用意することにすれば、行うべき処理は次の図の様に書き表すことができる。



あるいは、



この処理を行う C プログラムと、これをコンパイル/実行している様子を次に示す。 (下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl max-among-3-elem-no2.c Enter
     1  /* 3つの入力データの最大値(その2) */

     2  #include <stdio.h>
```

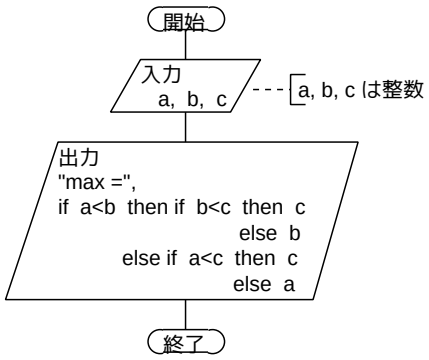
```
 3  int main(void)
 4  {
 5    int  a, b, c, max;

 6    scanf("%d%d%d", &a, &b, &c);
 7
 8    if (a < b){
 9      if (b < c)
10        max = c;
11      else
12        max = b;
13    }else{
14      if (a < c)
15        max = c;
16      else
17        max = a;
18    }

19    printf("max = %d\n", max);
20    return 0;
21  }
[motoki@x205a]$ gcc max-among-3-elem-no2.c Enter
[motoki@x205a]$ ./a.out Enter
1 2 3 Enter
max = 3
[motoki@x205a]$
```

ここで、

- プログラム 8 行目最後〜13 行目最初、13 行目最後〜18 行目 の部分は { と } で囲まれているので、それぞれ**複合文**と呼ばれ構文上は 1 つの文と同等に扱われる。
- プログラム 8 行目 の if は 13 行目 の else と組になり、**if-else 構文**と呼ばれる条件分岐の構文を構成している。このプログラムの場合は、8 行目の括弧の中の条件 a < b が成り立てば 8 行目最後〜13 行目最初の複合文が次に実行され、成り立たなければ 13 行目最後〜18 行目の複合文が次に実行されることになる。
- プログラム 9 行目,14 行目 の if はそれぞれ 11 行目, 16 行目 の else と組になり if-else 構文を構成し、条件分岐の働きをする。

(プログラミング,別の方向) C 言語においては、計算式の中で条件分岐を表すための構文が用意されている。 そこで、この構文の使用を前提にして

最大要素が特定できるまで場合分けを重ねるという考え方に従って処理手順を組み直すと、行うべき処理は右図の様に書き表すこともできる。



この処理を行う C プログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl max-among-3-elem-no4.c Enter
     1  /* ３つの入力データの最大値（その４） */

     2  #include <stdio.h>

     3  int main(void)
     4  {
     5    int  a, b, c;

     6    scanf("%d%d%d", &a, &b, &c);

     7    printf("max = %d\n", (a<b) ? (b<c ? c : b) : (a<c ? c : a));
     8    return 0;
     9  }
[motoki@x205a]$ gcc max-among-3-elem-no4.c Enter
[motoki@x205a]$ ./a.out Enter
1 2 3 Enter
max = 3
[motoki@x205a]$
```

ここで、

- プログラム 7 行目の (a<b) ? (b<c ? c : b) : (a<c ? c : a) は**条件演算子** (? と : の組) を入れ子に使って出来た式である。この式の値を求める時には、まず条件 (a<b) が成り立つかどうかが調べられ、成り立てば (b<c ? c : b) の値が、そして成り立たなければ式 (a<c ? c : a) の値が計算され、その結果が求める式の値として扱われる。

例題 6.1 に対するアルゴリズム (3) :

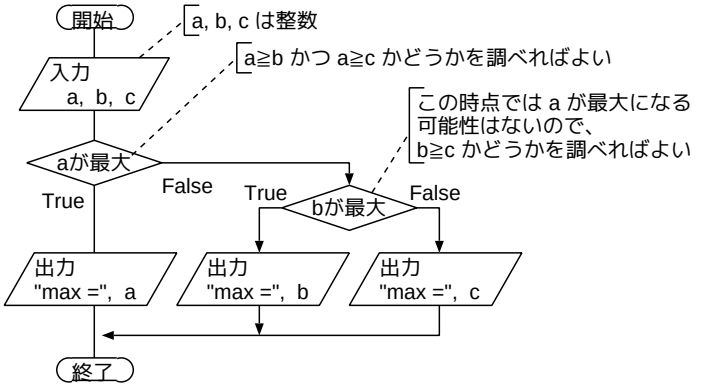
(考え方) 読み込んだ整数 a, b, c 間の大小関係を調べることにより、各々の要素が最大であるかどうかを判定することができる。例えば、

a が最大 $\Longleftrightarrow$ $a \geq b$ かつ $a \geq c$

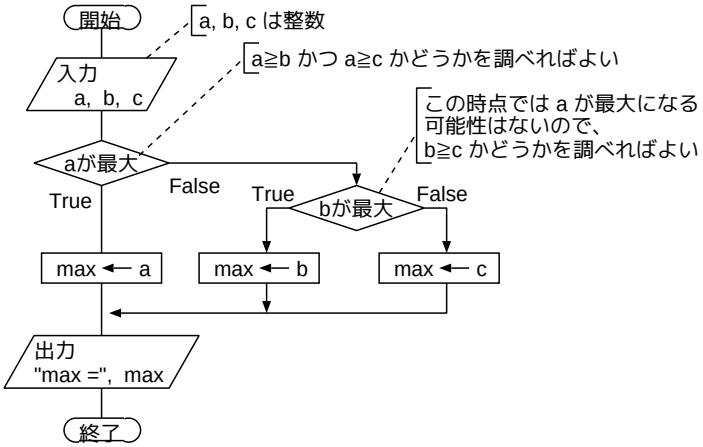
である。 それゆえ、整数 3 つを読み込んだあとで、まず a が最大かどうか で場合分けする。 その結果、もし a が最大 ならその値を出力し、もしそうでないなら 残った b, c の間

で b が最大かどうか で場合分けして、各々の場合について答えを出力すればよい。

(プログラミング) 読み込んだ整数データを格納するために a，b，c という名前の変数を用意することにすれば、行うべき処理は次の図の様に書き表すことができる。



あるいは、



この処理を行う C プログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl max-among-3-elem-no3.c Enter
     1  /* ３つの入力データの最大値（その３） */

     2  #include <stdio.h>

     3  int main(void)
     4  {
     5    int  a, b, c, max;

     6    scanf("%d%d%d", &a, &b, &c);
     7
     8    if (b<=a && c<=a)
     9      max = a;
    10    else if (c<=b)
    11      max = b;
    12    else
```

```
13      max = c;

14    printf("max = %d\n", max);
15    return 0;
16  }
[motoki@x205a]$ gcc max-among-3-elem-no3.c Enter
[motoki@x205a]$ ./a.out Enter
1 2 3 Enter
max = 3
[motoki@x205a]$
```

ここで、

- プログラム 8 行目 の式 `b<=a && c<=a` は「`b`≤`a` かつ `c`≤`a`」という意味の**論理式**である。
- プログラム 8〜13 行目 は if-else 構文が入れ子になった構造をしている。まず、論理式 `b<=a && c<=a` が成り立つかどうかが調べられ、もし成り立てば次に 9 行目の代入文が実行され、もし成り立たなければ次に 10 行目途中〜13 行目の if-else 構文が実行されることになる。

□**演習 6.2 (3 つの要素の最大値)** 先の例題で 3 つの要素の最大値を求める 3 種類のアルゴリズム、4 つの C プログラムが与えられているが、これらの内どれが良いか考えよ。

□**演習 6.3 (三角形が出来るかどうかの判定)** 3 つの整数 a, b, c を読み込み、a, b, c を 3 辺の長さとする三角形が存在するかどうかを判定する C プログラムを作成せよ。

C 言語における論理式の扱い (概略)： (⟹ 6.7.1 節を参照)

- C 言語には**真理値** (真と偽) を表すためのデータ型は用意されていない。
  ⟹ int 型で代用。
  $$\begin{cases} \text{真} & \cdots\ 0 \text{ 以外 (標準は 1)} \\ \text{偽} & \cdots\ 0 \end{cases}$$
- **関係演算子**として使えるのは `<`, `<=`, `>`, `>=`, `==`, `!=` の 6 つ。これにより、例えば `b*b-4*a*c>=0` や `x==0` といった**関係式**を条件判定に使うことができる。
- **論理演算子**として使えるのは `&&`, `||`, `!` の 3 つで、それぞれ AND, OR, NOT を表す。
  例えば、論理式

  ```
  a>0 && b>0 && c>0 && a+b>c && b+c>a && c+a>b
  ```

  は

  ```
  a>0 かつ b>0 かつ c>0 かつ a+b>c かつ b+c>a かつ c+a>b
  ```

  という意味であり、論理式

  ```
  !(a<=0 || b<=0 || c<=0)
  ```

  は

  ```
  (a<=0 または b<=0 または c<=0) でない
  ```

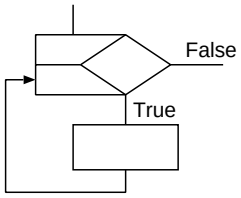
  という意味である。

- 式 $\boxed{p}$ `&&` $\boxed{q}$ は左の条件から順に評価され、$\boxed{p}$ の条件が不成立なら $\boxed{q}$ の評価を行うことなく、$\boxed{p}$ `&&` $\boxed{q}$ は不成立と判定される。同様に、$\boxed{p}$ `||` $\boxed{q}$ も左から順に評価され、$\boxed{p}$の条件が成立すれば$\boxed{q}$ の評価を行うことなく、$\boxed{p}$ `||` $\boxed{q}$ は成立と判定される。この様に式全体の評価値が確定した時点で評価を終える方式を**短絡評価**と言う。

> 論理式の値が整数として処理されるために、
> 本来は文法エラーとなるべき記述もチェックされない。例えば、
> - 誤った記述例： `if (a=1) ...` (正しくは `if (a==1) ...`)
>
> 条件部の「`a=1`」が代入式であるために、その値は (この場合) 常に 1 (真) となる。 それゆえ、変数 `a` の値が 1 以外の時も、`(a=1)` に続く (複合) 文が実行されてしまう。
> - 誤った記述例： `if (-127<E<128) ...`
>
> (正しくは `if (-127<E && E<128) ...`)
>
> 条件部の「`-127<E<128`」が
>
> `-127<E<128` $\Longrightarrow$ (式 `-127<E` の評価結果) < 128
> $\Longrightarrow$ (0 または 1) < 128
> $\Longrightarrow$ 1 (真を表す)
>
> という風に式変形/計算されるので、変数`E` の値に関わらずに条件 `-127<E<128` は常に成立すると判断され、`(-127<E<128)` に続く (複合) 文が必ず実行されてしまう。
>
> $\Longrightarrow$ 上記のような誤りは見つけにくいので特に気を付けること。

6.2 処理の規則的な繰り返し

この節では、右図の形の処理の流れがC言語でどの様に記述されるのかを見てみる。ここで扱うのは、特に繰り返しの見通し (e.g. 回数) がはっきりとしている場合である。



例題 6.4 (k^2, k^3, k^4 の表; for 文) 整数 $k = 1$〜10 について k^2, k^3, k^4 の値を計算し、それらの結果を表の形に見易く出力するCプログラムを作成せよ。

(考え方) ここでは半角空白を ␣ で表すことにして、出力する表の形を

```
␣k␣␣k^2␣␣␣k^3␣␣␣k^4
--␣␣---␣␣----␣␣-----
␣1␣␣␣␣1␣␣␣␣␣1␣␣␣␣␣␣1
␣2␣␣␣␣4␣␣␣␣␣8␣␣␣␣␣16
␣3␣␣␣␣9␣␣␣␣27␣␣␣␣81
      ............
10␣␣100␣␣1000␣␣10000
```

とする。 すると、これでコンピュータの行う出力の順序も決まるので、関連する計算を各々の出力の直前に行うことにすれば、処理手順を次のように書き下ろすことができる。

① 見出しの部分 (2 行分) を出力する。

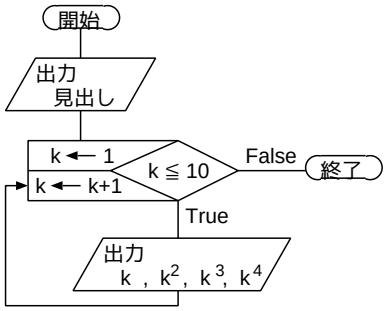
② $k = 1$ 対して k^2, k^3, k^4 の値を計算し、順に k, k^2, k^3, k^4 の値を出力する。

③ $k=2$ 対して k^2, k^3, k^4 の値を計算し、順に k, k^2, k^3, k^4 の値を出力する。
④ $k=3$ 対して k^2, k^3, k^4 の値を計算し、順に k, k^2, k^3, k^4 の値を出力する。
.....
⑪ $k=10$ 対して k^2, k^3, k^4 の値を計算し、順に k, k^2, k^3, k^4 の値を出力する。

(プログラミング) 上記の手順②～⑪は、

> k^2, k^3, k^4 の値を計算し
> 順に k, k^2, k^3, k^4 の値を出力する、

という処理を $k=1,2,3,...,10$ に対して順に行っているだけである。それゆえ、流れ図においてはこの②～⑪の部分を**繰り返しの箱** を用いて表すことができる。実際、$k=1$～10 の値を記憶するために k という名前の変数領域を用意することにすれば、行うべき処理は右図の様に書き表すことができる。



この処理を行う C プログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl table-of-k-k2-k3-k4.c Enter
     1 /* 整数 k=1～10 について k^2,k^3,k^4 の値を計算し、*/
     2 /* それらの結果を表の形に見易く出力する C プログラム */

     3 #include <stdio.h>

     4 int main(void)
     5 {
     6   int k;

     7   printf(" k  k^2   k^3   k^4\n"
     8          "--  ---  ----  -----\n");

     9   for (k=1; k<=10; ++k)
    10     printf("%2d  %3d  %4d  %5d\n", k, k*k, k*k*k, k*k*k*k);
    11   return 0;
    12 }
[motoki@x205a]$ gcc table-of-k-k2-k3-k4.c Enter
[motoki@x205a]$ ./a.out Enter
 k  k^2   k^3   k^4
--  ---  ----  -----
 1    1     1      1
 2    4     8     16
 3    9    27     81
 4   16    64    256
 5   25   125    625
```

```
 6   36   216   1296
 7   49   343   2401
 8   64   512   4096
 9   81   729   6561
10  100  1000  10000
[motoki@x205a]$
```

ここで、

- プログラム 9 行目 に現れる `++k` は `k=k+1` と同等の式である。 これと類似の `k++`, `--k`, `k--` という表現も C 言語ではよく使われる。 これらの表現の中の `++` と `--` をそれぞれ**増分演算子**, **減分演算子** と言う。 (⟹ p.45 を参照)
- プログラム 9〜10 行目 は **for 文**と呼ばれる、繰り返しのための構文である。9 行目は流れ図における繰り返しの箱  に相当するもので、`for` に続く括弧の中には、

  > 変数 `k` の最初の値をどう設定するか
  > 繰り返しを続けるための条件 (i.e. どんな `k` の値まで繰り返すか)
  > 1 回の繰り返し処理が終った後に変数 `k` をどう更新するか

  ということが書かれている。これらの記述によって、

  > `k=1` という設定を行った後で 次の文 (10 行目) を `k<=10` である間 繰り返す、
  > 但し、次の文 (10 行目) の実行が終るたびに、
  > `++k` を実行 して変数 `k` の保持する値を 1 だけ大きくする、

  ということを表す。この場合、変数 `k` は処理の繰り返しを制御する働きをするので、**繰り返し制御の変数**、**ループ制御の変数**、あるいは単に**制御変数**と言う。
- プログラムの 10 行目 の出力書式中の `%2d`, `%3d`, `%4d`, `%5d` は、それぞれ (半角) 2 文字分, 3 文字分, 4 文字分, 5 文字分 の出力場所を確保し、そこに右詰めで出力することを表す。

> **繰り返しの見通し (e.g. 回数) がはっきりしている場合は、**
> 上の例題 6.4 のプログラムで例示した様に、
> - 繰り返しを制御する変数を用意し
> - その**制御変数に関する操作を `for` に続く括弧の中に集めて**
>
> **for 文を構成する**のが良い。
>
> なぜなら、そういった for 文だと `for` に続く括弧の中だけを見て、逆にそこでどういう繰り返しが行われるかを容易に見通せるからです。

□**演習 6.5** (**k^2, k^3, k^4, k^5 の表**) 整数 $k = 0, 5, 10, 15, 20, 25, ..., 50$ について k^2, k^3, k^4, k^5 の値を計算し、それらの結果を表の形に見易く出力する C プログラムを作成せよ。

例題 6.6 (階乗;for 文) 正整数データ k を読み込み、その階乗値 $k! = 1{\times}2{\times}3{\times}\cdots{\times}k$ を整数として求めて出力する C プログラムを作成せよ。

(考え方) 我々が手で計算するとしたら、正整数 k が与えられたとき、次の様に計算を進める。

(step 1) $1! = 1$
(step 2) $2! = 1!\times 2 = 1\times 2 = 2$
(step 3) $3! = 2!\times 3 = 2\times 3 = 6$
(step 4) $4! = 3!\times 4 = 6\times 4 = 24$
(step 5) $5! = 4!\times 5 = 24\times 5 = 120$

.....

(step k) $k! = (k-1)!\times k = ...$

この計算をコンピュータに行わせれば良いわけであるが、この場合、どんな変数を用意すれば良いのだろうか? この計算の途中で出て来る $1!, 2!, 3!, ..., (k-1)!, k!$ の値は、計算のいずれかの時点でどこかの変数に記憶しておく必要がある。しかし、だからと言って、$1!, 2!, 3!, ...$ 各々毎に別の変数を用意して

(step 1) 1! の値を保持する変数 ⟵ 1
(step 2) 2! の値を保持する変数 ⟵ 1! の値を保持する変数×2
(step 3) 3! の値を保持する変数 ⟵ 2! の値を保持する変数×3
(step 4) 4! の値を保持する変数 ⟵ 3! の値を保持する変数×4
(step 5) 5! の値を保持する変数 ⟵ 4! の値を保持する変数×5

.....

(step k) $k!$ の値を保持する変数 ⟵ $(k-1)!$ の値を保持する変数×k

とするのでは、色々な k の値に対処するために際限のない個数の変数が必要になってしまう。

そこで、

> 次に $i!$ の値を計算する時点では、
> 計算に必要な値は $(i-1)!$ の値と i の値だけであり、
> $1!, 2!, ..., (i-2)!$ の値はそれ以降も必要ない

ことに注目する。 $i!$ の値が計算できてしまえばそれ以前に計算した 1!,2!,3!,...,$(i-1)!$ は保持しておく必要はないので、1!,2!,3!,... を保持するために 1 つだけ共通のデータ格納領域を用意し、

① 最初はそこに 1! の値を保持する,
② 2! が計算できれば保持されていた 1! の値は捨て代わりに 2! の値を保持する,
③ 3! が計算できれば保持されていた 2! の値は捨て代わりに 3! の値を保持する,

.....

ということにすれば良い。従って、1!,2!,3!,... の値を保持する変数 を用意して、次のアルゴリズムでコンピュータに計算させれば良い。

(step 0) 正整数 k を読み込む。
(step 1) 1!,2!,3!,... の値を保持する変数 ⟵ 1
(この時点で 1!,2!,3!,... の値を保持する変数 は 1! の値を保持しているはず)
(step 2) 1!,2!,3!,... の値を保持する変数 ⟵ 1!,2!,3!,... の値を保持する変数 ×2
(この時点で 1!,2!,3!,... の値を保持する変数 は 2! の値を保持しているはず)
(step 3) 1!,2!,3!,... の値を保持する変数 ⟵ 1!,2!,3!,... の値を保持する変数 ×3
(この時点で 1!,2!,3!,... の値を保持する変数 は 3! の値を保持しているはず)
(step 4) 1!,2!,3!,... の値を保持する変数 ⟵ 1!,2!,3!,... の値を保持する変数 ×4
(この時点で 1!,2!,3!,... の値を保持する変数 は 4! の値を保持しているはず)

(step 5) 1!,2!,3!,... の値を保持する変数 ⟵ 1!,2!,3!,... の値を保持する変数 ×5

(この時点で 1!,2!,3!,... の値を保持する変数 は 5! の値を保持しているはず)

.....

(この時点で 1!,2!,3!,... の値を保持する変数 は $(k-1)!$ の値を保持しているはず)

(step k) 1!,2!,3!,... の値を保持する変数 ⟵ 1!,2!,3!,... の値を保持する変数 ×k

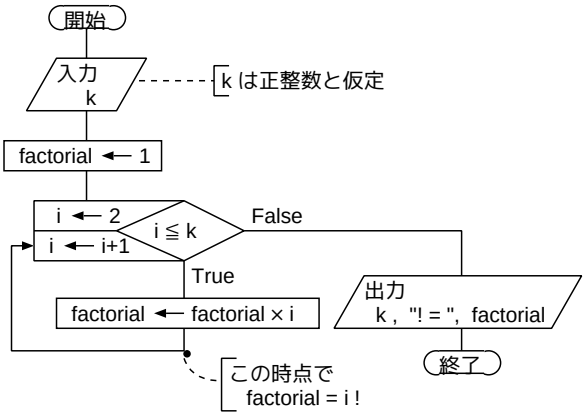
(この時点で 1!,2!,3!,... の値を保持する変数 は $k!$ の値を保持しているはず)

(step $k+1$) 1!,2!,3!,... の値を保持する変数の値を出力

(プログラミング) 上記の手順 (step 2)〜(step k) は、

1!,2!,3!,... の値を保持する変数 ⟵ 1!,2!,3!,... の値を保持する変数 × i

という処理を $i = 2, 3, 4, ..., k$ に対して順に行っているだけである。それゆえ、流れ図においてはこの (step 2)〜(step k) の部分を**繰り返しの箱** を用いて表すことができる。読み込んだ正整数を格納するために k という名前の変数を、1!,2!,3!,... の値を保持するために factorial という名前の変数を、そして $i = 2$〜k の値を記憶するために i という名前の変数を用意することにすれば、行うべき処理は次の流れ図の様に書き表すことができる。



補足：
"factorial" は階乗という意味の英単語である。
⟹ 階乗と何の関係のない計算に "factorial" という名前の変数を用いてはならない。

実際には、整数型 32 ビットでは 13! は記憶できないので、上記プログラムのように int 型で階乗値を正確に求めたい場合は、入力変数 k は 12 以下でなければならない。

この処理を行う C プログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl factorial.c  Enter
     1 /* 正整数データ (12 以下と仮定) を読み込み、              */
     2 /* その階乗値を整数として求めて出力する C プログラム */

     3 #include <stdio.h>

     4 int main(void)
     5 {
```

```
 6   int    k, i, factorial;

 7   printf("何の階乗を求めますか?: ");
 8   scanf("%d", &k);

 9   factorial = 1;
10   for (i=2; i<=k; ++i){
11     factorial *= i;   /* この時点で factorial = i! */
12   }

13   printf("%d! = %d\n", k, factorial);
14   return 0;
15 }
[motoki@x205a]$ gcc factorial.c Enter
[motoki@x205a]$ ./a.out Enter
何の階乗を求めますか?: 10 Enter
10! = 3628800
[motoki@x205a]$
```

□**演習 6.7 (総和 $\sum_{i=1}^{k} i$)** 正整数データ k を読み込み、1 から k までの総和 $\sum_{i=1}^{k} i = 1+2+3+\cdots+k$ を整数として求めて出力する C プログラムを作成せよ。

□**演習 6.8 (総和 $\sum_{i=1}^{k} i^3$)** 正整数データ k を読み込み、総和 $\sum_{i=1}^{k} i^3 = 1^3+2^3+3^3+\cdots+k^3$ を整数として求めて出力する C プログラムを作成せよ。

□**演習 6.9 (冪乗)** 整数 a と 非負整数 k を入力し、これらを基に a^k を計算して出力する C プログラムを作成せよ。但し、ここでは簡単のため $0^0 = 1$ と考えよ。

□**演習 6.10 (階乗の表)** 整数 $k = 1$〜12 について $k!$ の値を計算し、それらを表の形に見易く出力する C プログラムを作成せよ。

□**演習 6.11 (Fibonacci 数列)** 一般に、初期値 a_0, a_1 と漸化式

$$a_n = a_{n-1} + a_{n-2} \quad \text{for each } n \geq 2$$

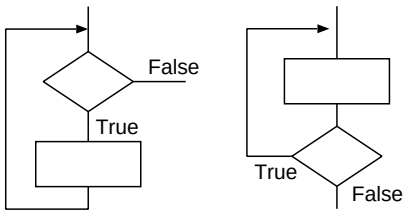
によって決まる数列 $\{a_n\}$ を **Fibonacci 数列**という。45 以下の非負整数 k を読み込んで

初期値が $a_0 = a_1 = 1$ の Fibonacci 数列の第 $(k+1)$ 項 a_k

を計算して出力する C プログラムを作成せよ。

6.3 条件判断による処理の繰り返し

この節では、右図の形の処理の流れがC言語でどの様に記述されるのかを見てみる。ここで扱うのは、特に繰り返しの見通し (e.g. 回数) が繰り返しを行う前にははっきりとせず、計算状況を見ながらその都度繰り返しを終了するかどうかの判断を行う場合である。



例題 6.12 (最大公約数, ユークリッドの互除法; while 文, do-while 文) 2 つの正整数を読み込み、それらの最大公約数を出力するC プログラムを作成せよ。

(考え方) 最大公約数を求めるアルゴリズムとしては、**ユークリッドの互除法** (あるいは**ユークリッドのアルゴリズム**) と呼ばれるものが有名である。(古代ギリシャ時代から伝わる「ユークリッド原論」という表題の本の中に書き記されている。) このアルゴリズムは次の事実に基づいて計算を進める。

命題 6.13 2つの正整数 a, b の最大公約数を $\gcd(a, b)$、整数 b を整数 a で割った時の余りを $\mathrm{mod}(b, a)$ と表すことにすれば、
(1) $a < b$ なら $\gcd(a, b) = \gcd(a, b - a)$
(2) $\gcd(a, b) = \gcd(\mathrm{mod}(b, a), a)$

この命題に基づけば、我々は 1596 と 308 の最大公約数 gcd(1596,308) の計算を次のように進めることができる。

$$
\begin{aligned}
\gcd(1596,\ 308) &= \gcd(\mathrm{mod}(308,1596),\ 1596) && \text{命題 6.13(2) より}\\
&= \gcd(308,\ 1596)\\
&= \gcd(\mathrm{mod}(1596,308),\ 308) && \text{命題 6.13(2) より}\\
&= \gcd(56,\ 308)\\
&= \gcd(\mathrm{mod}(308,56),\ 56) && \text{命題 6.13(2) より}\\
&= \gcd(28,\ 56)\\
&= \gcd(\mathrm{mod}(56,28),\ 28) && \text{命題 6.13(2) より}\\
&= \gcd(0,\ 28)\\
&= 28
\end{aligned}
$$

では、この場合どんな変数を用意すれば良いのか? この計算は、結局は

$$\gcd(1596,\ 308) \Longrightarrow \gcd(308,\ 1596) \Longrightarrow \gcd(56,\ 308) \Longrightarrow \ldots\ldots$$

という式変形を行っているだけである。 それゆえ、コンピュータ向きのアルゴリズムにおいては、この式変形のどこまで進んでいるのかを常に認識し、その認識に基づいて次の式変形を進めることになる。式変形の現在の状態 gcd(..., ...) を認識するために gcd( ) の第1引数, 第2引数を記憶する変数を用意し、各々 x, y という名前を付けることにすれば、先の計算例における変数の更新は次の様に進む。

$$
\begin{aligned}
&\gcd(\ \overset{\text{変数 } x}{\boxed{1596}}\ ,\ \overset{\text{変数 } y}{\boxed{308}}\ ) \Longrightarrow \gcd(\ \overset{\text{変数 } x}{\boxed{308}}\ ,\ \overset{\text{変数 } y}{\boxed{1596}}\ )\\
&\Longrightarrow \gcd(\ \overset{\text{変数 } x}{\boxed{56}}\ ,\ \overset{\text{変数 } y}{\boxed{308}}\ )
\end{aligned}
$$

$$\Longrightarrow \quad \gcd(\ \overset{\text{変数 } x}{\boxed{28}}\ ,\ \overset{\text{変数 } y}{\boxed{56}}\ )$$
$$\Longrightarrow \quad \gcd(\ \overset{\text{変数 } x}{\boxed{0}}\ ,\ \overset{\text{変数 } y}{\boxed{28}}\ )$$
$$\Longrightarrow \quad 28$$

それでは、これらの変数値更新のために実際にどういう処理を行えば良いのか? 1つの式変形の状態から次の状態への更新は、ほとんどの場合次の様に進む。

$$\gcd(\ \overset{\text{変数 } x}{\boxed{a}}\ ,\ \overset{\text{変数 } y}{\boxed{b}}\ ) \quad \Longrightarrow \quad \gcd(\ \overset{\text{変数 } x}{\boxed{\mathrm{mod}(b,a)}}\ ,\ \overset{\text{変数 } y}{\boxed{a}}\ )$$

この更新を行うには、例えば $next\_x$ という名前の変数を用意して

$$\begin{aligned} next\_x &\longleftarrow \mathrm{mod}(y,x) \\ y &\longleftarrow x \\ x &\longleftarrow next\_x \end{aligned}$$

とすれば良い。[変数への代入は一般には同時に行えないので、この様に3番目の変数が必要になる。] また、最後の

$$\gcd(\ \overset{\text{変数 } x}{\boxed{0}}\ ,\ \overset{\text{変数 } y}{\boxed{28}}\ ) \Longrightarrow 28$$

という式変形は、変数 x の値が 0 なので起こっていると考えられる。 結局、gcd(1596,308) の計算の場合に、どういう処理によってどういう風に状態が変わっていくかを具体的に明示すると次の様になる。

状態 $\gcd(\ \overset{\text{変数 } x}{\boxed{1596}}\ ,\ \overset{\text{変数 } y}{\boxed{308}}\ )$

$\Downarrow$ $next\_x \longleftarrow \mathrm{mod}(y,x)$
$\Downarrow$ $y \longleftarrow x$
$\Downarrow$ $x \longleftarrow next\_x$

状態 $\gcd(\ \overset{\text{変数 } x}{\boxed{308}}\ ,\ \overset{\text{変数 } y}{\boxed{1596}}\ )$

$\Downarrow$ $next\_x \longleftarrow \mathrm{mod}(y,x)$
$\Downarrow$ $y \longleftarrow x$
$\Downarrow$ $x \longleftarrow next\_x$

状態 $\gcd(\ \overset{\text{変数 } x}{\boxed{56}}\ ,\ \overset{\text{変数 } y}{\boxed{308}}\ )$

$\Downarrow$ $next\_x \longleftarrow \mathrm{mod}(y,x)$
$\Downarrow$ $y \longleftarrow x$
$\Downarrow$ $x \longleftarrow next\_x$

状態 $\gcd(\ \overset{\text{変数 } x}{\boxed{28}}\ ,\ \overset{\text{変数 } y}{\boxed{56}}\ )$

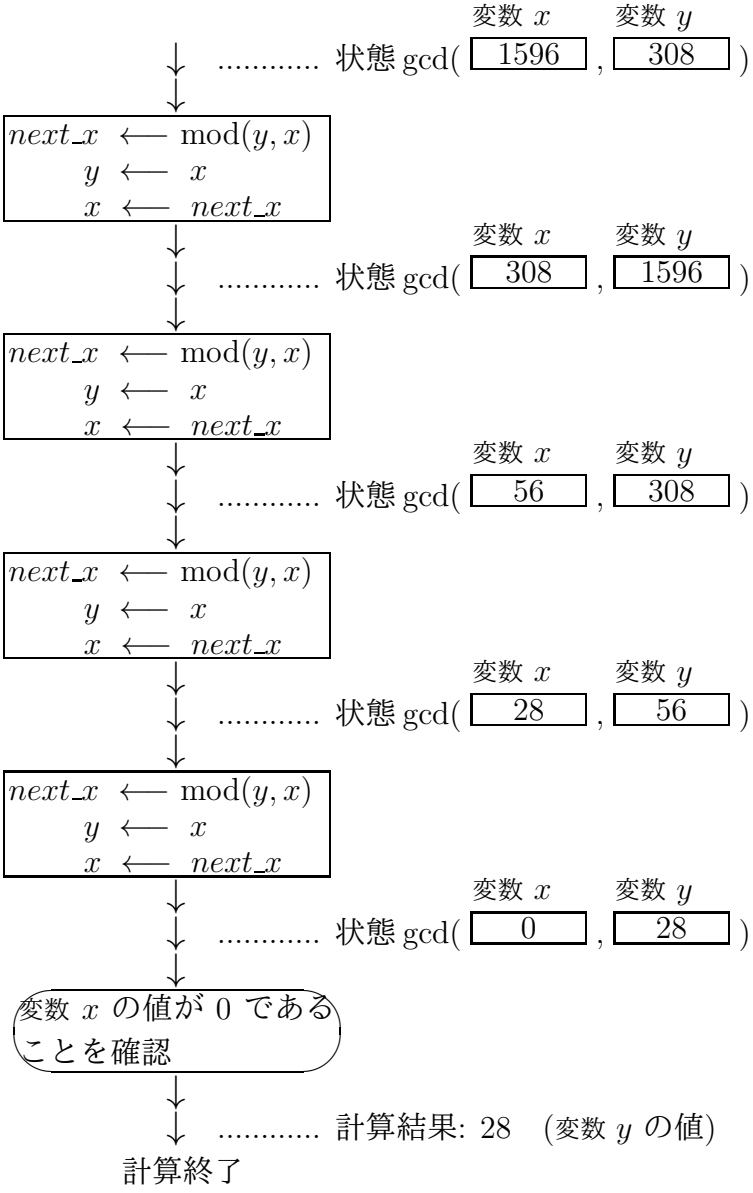
$\Downarrow$ $next\_x \longleftarrow \mathrm{mod}(y,x)$
$\Downarrow$ $y \longleftarrow x$
$\Downarrow$ $x \longleftarrow next\_x$

状態 $\gcd(\ \overset{\text{変数 } x}{\boxed{0}}\ ,\ \overset{\text{変数 } y}{\boxed{28}}\ )$

$\Downarrow$ 変数 x の値が 0 であることを確認

計算結果 28

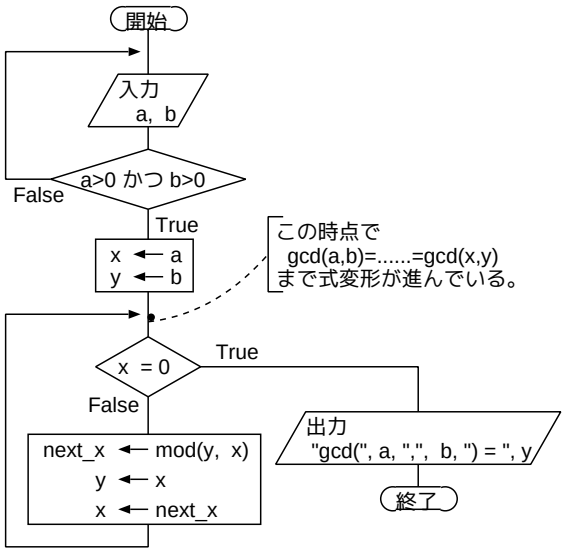
(プログラミング) 上述の、状態とそれらの間の遷移を引き起こす処理の関係図を、処理を中心に書き直すと次の様になる。



要するに、$x = 0$ となるまで

$$\left|\begin{array}{rcl} next\_x & \longleftarrow & \mathrm{mod}(y, x) \\ y & \longleftarrow & x \\ x & \longleftarrow & next\_x \end{array}\right.$$

という固定的な処理を繰り返すだけである。 変数 x の値が 0 になった時点では、計算結果は変数 y に保持されている。それゆえ、読み込んだ整数データを格納するために `a`, `b` という名前の変数を、式変形の際の gcd(..., ...) の第 1 引数, 第 2 引数を保持するために各々 `x`, `y` という名前の変数を、式変形を行う際に変数 `x` の次の値を一時的に保持するために `next_x` という名前の変数を用意することにすれば、行うべき処理は次の図の様に書き表すことができる。



この処理を行うCプログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl gcd-euclid-algorithm.c  Enter
     1 /* 2つの正整数を読み込み、それらの最大公約数を出力 */
     2 /* するCプログラム  (Euclidのアルゴリズム)          */

     3 #include <stdio.h>

     4 int main(void)
     5 {
     6   int  a, b, x, y, next_x;

     7   do {
     8     printf("最大公約数を計算します。正整数を2つ入力して下さい: ");
     9     scanf("%d%d", &a, &b);
    10   }while (!(a>0 && b>0));
    11   x = a;
    12   y = b;

    13   while (x != 0) {   /* この時点で gcd(a,b)=...=gcd(x,y) */
    14                      /* まで式変形が進んでいる。          */
    15     next_x = y%x;
    16     y      = x;
    17     x      = next_x;
    18   }

    19   printf("gcd(%d, %d) = %d\n", a, b, y);
    20   return 0;
    21 }
[motoki@x205a]$ gcc gcd-euclid-algorithm.c  Enter
```

```
[motoki@x205a]$ ./a.out Enter
最大公約数を計算します。正整数を２つ入力して下さい: 1596 308 Enter
gcd(1596, 308) = 28
[motoki@x205a]$
```

ここで、

- プログラムの 7〜10 行目 は **do-while 文**と呼ばれる、繰り返しのための構文である。これによって、

  > 条件 `!(a>0 && b>0)` を満たす間 は 8〜9 行目の実行を繰り返す
  > 但し、7〜10 行目の処理に入って 最初に行われるのは 8〜9 行目 で、
  > その後に 10 行目の条件チェックが続く

  ということを表す。
- プログラムの 13〜18 行目 は **while 文**と呼ばれる、繰り返しのための構文である。これによって、

  > 条件 `(x != 0)` を満たす間 は 14〜17 行目の実行を繰り返す
  > 但し、13〜18 行目の処理に入って 最初に行われるのは 13 行目の
  > 条件チェック で、その後に 14〜17 行目の実行が続く

  ということを表す。

> プログラムの注釈について：
> どういう変数を用意するか、変数を使って計算の現状態をどう表すか、といったことはアルゴリズム/プログラムを設計する上での重大な決定事項である。 それゆえ、上記プログラム 13〜14 行目 の様に
>
> > **繰り返しの各々の時点で計算状態がどうなっているか**が
> > 変数を使って明示されて
>
> いれば、プログラムは理解し易くなる。
> $\Longrightarrow$ **こういう注釈を入れるよう心掛ける**こと。

□**演習 6.14**　命題 6.13 を証明せよ。

□**演習 6.15 (素因数分解)** 正整数を読み込み、それを素因数分解して答える C プログラムを作成せよ。 但し、例えば 168 を読み込んだ場合、

```
168 = 2*2*2*3*7
```

という風に出力することにせよ。

例題 6.16 (素数; while 文,break 文)　2 以上の整数を読み込み、それが素数かどうかを判定して答える C プログラムを作成せよ。

(考え方)　2 以上の整数 k が与えられたとき、

k が素数 $\iff$ k は 2〜$k-1$ の整数で割り切れない　(素数の定義より)
$\iff$ k は 2〜$\sqrt{k}$ の整数で割り切れない

(i が k の約数なら k/i も k の約数で i と k/i のどちらかは $\sqrt{k}$ 以下となるから)

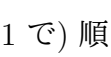
である。 従って、与えられた 2 以上の整数 k が素数であるかどうかを判定するためには、単に

2 が k を割り切るか,
3 が k を割り切るか,
4 が k を割り切るか,
.....
$\lfloor\sqrt{k}\rfloor$ が k を割り切るか,

ということを順に調べて、途中で「割り切る」という結果になったら即座に「素数でない」と判定を与え、途中で全然「割り切る」という結果にならなければ「素数だ」と判定を与えれば良い。

(プログラミング) 2 以上の整数 k が素数かどうかの判定は、基本的には

i が k を割り切るかどうかを調べ ...

という処理を $i = 2, 3, ..., \lfloor\sqrt{k}\rfloor$ に対して (すなわち $i = 2$ から始め条件 $i^2 \leq k$ を満たす間、刻み幅 $+1$ で) 順に行えば良いだけである。流れ図においては繰り返しの箱  を用いるだけである。 ただ、この繰り返しは次の 2 点において通常の繰り返しと違っている。

(1) 繰り返しは途中で中止する可能性もある。
(2) 繰り返し後は判定結果を出すだけの状態になっているので、この繰り返し処理は 2 つの出口を持つ。

実際、
◇ 繰り返しの途中で「割り切る」という結果になったら、即座に繰り返しを終了して 「素数でない」と判定を下し、また、
◇ 繰り返しの途中で全然「割り切る」という結果に結果にならなければ、繰り返し後に「素数だ」と判定を下したい。

一般に、予め処理手順を C 言語向きに構成しておかないと、実際に C プログラムを書く際に困ったことになる。 そこで、ここでは、上記 (1)〜(2) の特異点に対して次の様に対処する。

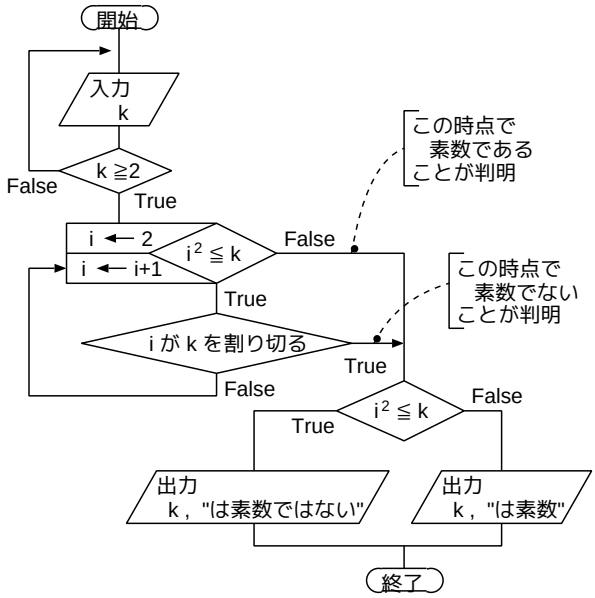
上記 (1) に対する方策: C 言語では、現在実行中の場所から見て最も内側の繰り返し (または switch 文) から脱出するために、**break 文**と呼ばれるものが用意されている。 プログラムを書く際は、それを使って繰り返しを途中で中止させる。

上記 (2) に対する方策: C 言語の繰り返しの構文はどれも出口が 1 箇所であるので、繰り返しを途中で中止する場合と最後まで行った場合を区別せずに、繰り返し終了直後の処理を共通に用意しなければならない。 [流れ図上では、繰り返しを途中で中止する場合の線と最後まで行った後の線が合流することになる。] しかし、一旦合流したとしても、合流直後の繰り返しの変数 i の値を調べて、

もし $i^2 \leq k$ なら 繰り返しを途中で中止した,
もし $i^2 > k$ なら 繰り返しを最後まで行った

と判断することができるので、すぐに元の 2 つの場合に分けて各々の場合に適切な処置を行う。

以上のことより、行うべき処理は次の流れ図の様に書き表すことができる。



この処理を行うCプログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl prime-number.c Enter
     1 /* 2以上の整数を読み込み、それが素数かどうかを判定して */
     2 /* 答えるCプログラム                                    */

     3 #include <stdio.h>

     4 int main(void)
     5 {
     6   int k, i;

     7   do {
     8     printf("素数かどうかの判定をします。"
     9            "2以上の整数を1つ入力して下さい: ");
    10     scanf("%d", &k);
    11   }while (!(k >= 2));

    12   for (i=2; i*i<=k; i++) {
    13     if (k%i == 0)       /* k%i==0 <==> i が k を割り切る */
    14       break;            /* この時点でkが素数でないことが判明 */
    15   }

    16   if (i*i<=k)
    17     printf("%dは素数ではない。\n", k);
    18   else
    19     printf("%dは素数です。\n", k);
    20   return 0;
```

```
    21 }
[motoki@x205a]$ gcc prime-number.c Enter
[motoki@x205a]$ ./a.out Enter
素数かどうかの判定をします。2 以上の整数を 1 つ入力して下さい: 24 Enter
24 は素数ではない。
[motoki@x205a]$ ./a.out Enter
素数かどうかの判定をします。2 以上の整数を 1 つ入力して下さい: 31 Enter
31 は素数です。
[motoki@x205a]$
```

ここで、

- プログラムの 7〜11 行目 の **do-while 文**によって、

  > 条件 `!(k>=2)` を満たす間 は 8〜10 行目の実行を繰り返す、
  > 但し、7〜11 行目の処理に入って 最初に行われるのは 8〜10 行目 で、
  > その後に 11 行目の条件チェックが続く

  ということを表す。
- プログラムの 12〜15 行目 の for 文 によって、

  > `i=2` という設定を行った後で 13〜14 行目を `i*i<=k` である間 繰り返す、
  > 但し、13〜14 行目の実行が終るたびに、
  > `i++` を実行 して変数 `i` の保持する値を 1 だけ大きくする、

  ということを表す。
- プログラム 13 行目 の `k%i==0` では、除算の際の余りを出す演算子 `%` を使って「`i` が `k` を割り切るかどうか」の条件を表している。
- プログラム 14 行目 の **break 文**が実行されると、 12〜15 行目の繰り返しは即座に終了し、次に 16 行目が実行される。

□**演習 6.17 (素数の表)** 1000 以下の素数を全て出力する C プログラムを作成せよ。

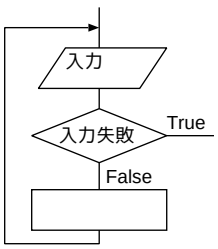
□**演習 6.18 (完全数)** 正整数 k が等式

$$k = (k \text{ の約数のうち、} k \text{ 以外のものの総和})$$

を満たすとき、k は**完全数**であると言う。例えば、6 の約数は 1,2,3,6 の 4 個であり、$6 = 1+2+3$ であるから、6 は完全数である。 1000 以下の完全数を全て出力する C プログラムを作成せよ。 (あまりないので、1 行に 1 個ずつ出力するというので良い。)

6.4 入力データが無くなるまで繰り返し

この節では、右図の形の処理の流れが C 言語でどの様に記述されるのかを見てみる。



例題 6.19 (不定個の入力データの合計) データが無くなるまで次々と整数データを読み込み、それらの数値の合計を求めて出力する C プログラムを作成せよ。

(考え方) 例えば入力データが 2, 5, 10, 33, 77, ... の時、我々が手で合計を出すとしたら、次の様に入力順に計算を進める。

(step 1) 1 番目のデータまでの合計 = 2
(step 2) 2 番目のデータまでの合計 = 1 番目のデータまでの合計 + 5 = 2 + 5 = 7
(step 3) 3 番目のデータまでの合計 = 2 番目のデータまでの合計 + 10 = 7 + 10 = 17
(step 4) 4 番目のデータまでの合計 = 3 番目のデータまでの合計 + 33 = 17 + 33 = 50
(step 5) 5 番目のデータまでの合計 = 4 番目のデータまでの合計 + 77 = 50 + 77 = 127

. . . . . .

これに相当する計算を一般的にコンピュータに行わせれば良い。

では、この場合、どんな変数を用意すれば良いのだろうか? データの個数が予め分かってないので、読み込むデータ毎に別々の記憶領域を用意する という訳にもいかない。

なぜなら、
プログラム上で十分な容量の領域を用意したつもりでも、データの個数がそれより多いということが起こり得るからである。

そこで、読み込んだデータを保持する変数を 1 個だけ用意し、

- そこへのデータ読み込みと、
- 読み込んだデータに対する処理

を交互に繰り返すことにする。 過去に読み込んだデータは保存されないので、次のデータを読む前に「読み込んだデータに対する処理」を十分に行わなければならない。この問題の場合、「読み込んだデータに対する処理」の直後には、それまでに読み込んだデータの合計が計算されどこかに保存されている必要がある。 それゆえ、次のように処理を進める。

(step 1.a) 整数データ 1 個を [入力データを保持する変数] に読み込む。
(step 1.b) [1 番目のデータまでの合計を保持する変数] ⟵ [入力データを保持する変数]
(step 2.a) 整数データ 1 個を [入力データを保持する変数] に読み込む。
(step 2.b) [2 番目のデータまでの合計を保持する変数]
⟵ [1 番目のデータまでの合計を保持する変数] + [入力データを保持する変数]
(step 3.a) 整数データ 1 個を [入力データを保持する変数] に読み込む。
(step 3.b) [3 番目のデータまでの合計を保持する変数]
⟵ [2 番目のデータまでの合計を保持する変数] + [入力データを保持する変数]

. . . . . .

(step k.a) 整数データ 1 個を [入力データを保持する変数] に読み込む。
(step k.b) [k 番目のデータまでの合計を保持する変数]
⟵ [$(k-1)$ 番目のデータまでの合計を保持する変数] + [入力データを保持する変数]
(final step) 読み込むデータが無くなったら、
[k 番目のデータまでの合計を保持する変数] の値を出力して終了。

これだと、それまでに読み込んだデータの合計を保持するために際限のない個数の変数が必要になる様に見えるが、実際には、

> i 番目のデータまでの合計を計算する時点では、
> 計算に必要な値は $(i-1)$ 番目までの合計と i 番目のデータ値だけであり、
> それ以外の合計の結果はそれ以降も必要ない

から、例題 6.6 の場合と同様に考えて、それまでに読み込んだデータの合計を保持するために共通のデータ格納領域を 1 つだけ用意すれば良いことが分かる。従って、共通のデータ格納領域として[それまでの合計を保持する変数]を用意して、次の手順でコンピュータに計算させれば良い。

(step 1.a) 整数データ 1 個を [入力データを保持する変数] に読み込む。
(step 1.b) [それまでの合計を保持する変数] ⟵ [入力データを保持する変数]
(step 2.a) 整数データ 1 個を [入力データを保持する変数] に読み込む。
(step 2.b) [それまでの合計を保持する変数]
⟵ [それまでの合計を保持する変数] + [入力データを保持する変数]
(step 3.a) 整数データ 1 個を [入力データを保持する変数] に読み込む。
(step 3.b) [それまでの合計を保持する変数]
⟵ [それまでの合計を保持する変数] + [入力データを保持する変数]
. . . . . .
(step k.a) 整数データ 1 個を [入力データを保持する変数] に読み込む。
(step k.b) [それまでの合計を保持する変数]
⟵ [それまでの合計を保持する変数] + [入力データを保持する変数]
(final step) 読み込むデータが無くなったら、
[それまでの合計を保持する変数] の値を出力して終了。

(プログラミング) 上記の手順 (step 2.a)〜(step k.b) は、

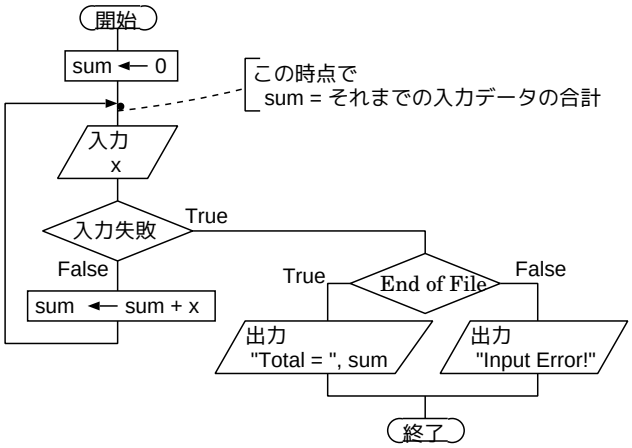
(a) 整数データ 1 個を [入力データを保持する変数] に読み込む。
(b) [それまでの合計を保持する変数]
⟵ [それまでの合計を保持する変数] + [入力データを保持する変数]

という処理を 入力データが無くなるまで繰り返しているだけである。手順 (step 1.a)〜(step 1.b) も、この共通の繰り返しパターンを使って

(step 0) [それまでの合計を保持する変数] ⟵ 0
(a) 整数データ 1 個を [入力データを保持する変数] に読み込む。
(b) [それまでの合計を保持する変数]
⟵ [それまでの合計を保持する変数] + [入力データを保持する変数]

} 共通の繰り返しパターン

と書き換えることができる。 また、C 言語では入力データ側に異常があった場合でも即座に実行時のエラーとはならないので、データ入力の失敗が起こった時その原因に応じた処置が必要である。 それゆえ、入力データを保持するために `x` という名前の変数を、そして、それまでの合計を保持するために `sum` という名前の変数を用意することにすれば、行うべき処理は次の流れ図の様に書き表すことができる。



この処理を行うCプログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl sum-input-until-eof.c Enter
     1 /* 不定個の整数入力データの合計を求めて出力するCプログラム */

     2 #include <stdio.h>

     3 int main(void)
     4 {
     5   int  x, sum, scanf_val;

     6   sum = 0;
     7   while ((scanf_val=scanf("%d", &x))==1){
     8     sum += x;       /* sum = それまでの入力の合計 */
     9   }

    10   if(scanf_val == EOF)
    11     printf("Total = %d\n", sum);
    12   else
    13     printf("Input Error!\n");
    14   return 0;
    15 }
[motoki@x205a]$ gcc sum-input-until-eof.c Enter
[motoki@x205a]$ ./a.out Enter
1 2 3 4 Enter
5 6 7 8 9 10 Enter
Ctrl-d
Total = 55
[motoki@x205a]$ ./a.out Enter
1 2 w 4 5 Enter
Input Error!
[motoki@x205a]$
```

ここで、

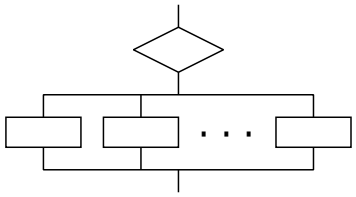
- プログラム 7 行目 の while 文の条件部 `(scanf_val=scanf("%d", &x))==1` に関して、一般に

  関数 scanf の値
  $$= \begin{cases} \text{データ入力に成功した回数} \\ \qquad\qquad \text{if 入力側に何らかのデータがあった} \\ \texttt{EOF} \quad (\text{普通 } -1 \text{ と} \texttt{<stdio.h>} \text{ で定義されるマクロ}) \\ \qquad\qquad \text{if 何も入力しないうちにファイルの終りに到達した} \end{cases}$$
  である。7〜9 行目の while 文においては、(後での使用のために) 一旦この scanf の値を変数 scanf_val に代すること、および scanf の値が 1 である (すなわちデータ入力に成功する) 間 8 行目を繰り返すことを指示している。
- プログラム 10 行目 の if 文の条件部においては、7 行目の代入式で変数 scanf_val に保存しておいた scanf の値を調べて、無事入力データの読み込みが終了したのか、それとも入力データ側にエラーがあったのかの判断を行っている。

□**演習 6.20 (不定個の入力データの平均)** データが無くなるまで次々と整数データを読み込み、それらの数値の平均を求めて出力する C プログラムを作成せよ。

6.5 式の値に基づいた処理の選択

この節では、右図の形の処理の流れが C 言語でどの様に記述されるのかを見てみる。

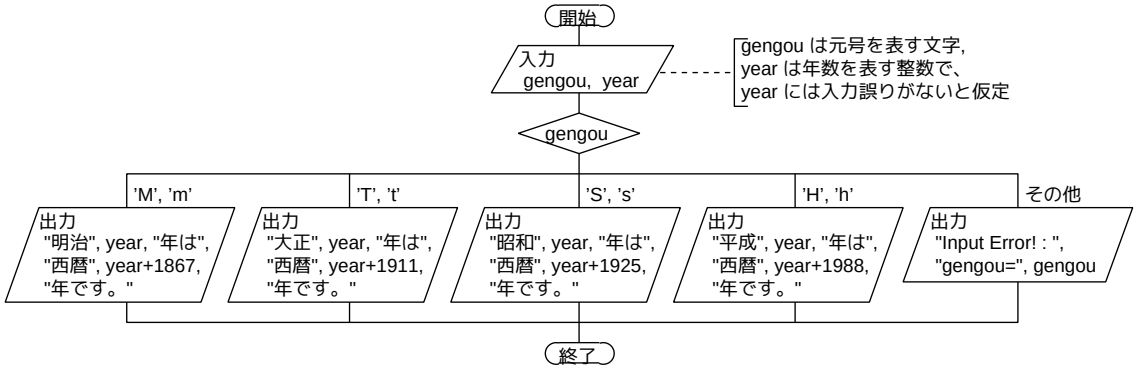


例題 6.21 (元号表記→西暦表記) 元号を表す文字 (M, m, T, t, S, s, H, または h) と年数を読み込み、その元号表記の年を西暦表記に変換して出力する C プログラムを作成せよ。

(考え方) 元号を表す文字と年数を読み込んだ後、元号を表す文字が何であるかによって場合分けするだけである。 読み込んだ年数に誤りがないとすると、具体的には、

元号文字が M または m の場合： 明治元年が西暦 1868 年だから、
　西暦の年数 読み込んだ年数 +1867 を出力すればよい。
元号文字が T または t の場合： 大正元年が西暦 1912 年だから、
　西暦の年数 読み込んだ年数 +1911 を出力すればよい。
元号文字が S または s の場合： 昭和元年が西暦 1926 年だから、
　西暦の年数 読み込んだ年数 +1925 を出力すればよい。
元号文字が H または h の場合： 平成元年が西暦 1989 年だから、
　西暦の年数 読み込んだ年数 +1988 を出力すればよい。
元号文字が M,m,T,t,S,s,H,h 以外の場合：
　入力データの誤りを指摘すればよい。

(プログラミング) 読み込んだ元号文字データ, 年数データを格納するためにそれぞれ gengou, year という名前の変数を、用意することにすれば、行うべき処理は次の図の様に書き表すことができる。



この処理を行う C プログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl trans-gengou-year-to-Gregorian-year.c Enter
```

```
 1 /* 元号を表す文字 (M,m,T,t,S,s,H, または h) と年数を読み込み、 */
 2 /* その元号表記の年を西暦表記に変換して出力する C プログラム */

 3 #include <stdio.h>

 4 int main(void)
 5 {
 6   char  gengou;
 7   int   year;

 8   scanf("%c%d", &gengou, &year);

 9   switch (gengou) {
10   case 'M': case 'm':
11     printf("明治%d 年は西暦%d 年です。\n",
12            year, year+1867);
13     break;
14   case 'T': case 't':
15     printf("大正%d 年は西暦%d 年です。\n",
16            year, year+1911);
17     break;
18   case 'S': case 's':
19     printf("昭和%d 年は西暦%d 年です。\n",
20            year, year+1925);
21     break;
22   case 'H': case 'h':
23     printf("平成%d 年は西暦%d 年です。\n",
24            year, year+1988);
```

```
25      break;
26    default:
27      printf("Input Error!: gengou='%c'\n", gengou);
28    }
29    return 0;
30  }
[motoki@x205a]$ gcc trans-gengou-year-to-Gregorian-year.c
[motoki@x205a]$ ./a.out
H15
平成15年は西暦2003年です。
[motoki@x205a]$ ./a.out
G15
Input Error!: gengou='G'
[motoki@x205a]$
```

ここで、

- プログラムの6行目では**char型**データのための変数領域を1つ確保している。 char型変数領域は本来1文字分の文字データを記憶するためのものであるが、C言語では文字が文字番号(文字コードを整数として見た時の整数値)で表されているので、char型領域は小さな整数値を保持するのに用いることも出来る。
- プログラム8行目の入力書式中の `%c` は、文字を1文字読み込みその文字コードをそのまま指定されたchar型領域に格納することを指示している。プログラム内では、char型変数 `gengou` は整数型の一種として扱われる。
- プログラム10行目の `'M'` と `'m'` はそれぞれ `M` と `m` という文字のコードを整数として見た時の値(int型)を表す。14行目, 18行目, 22行目の `'T'`, `'t'`, `'S'`, `'s'`, `'H'`, `'h'` も同様。
- プログラム9〜28行目は**switch文**と呼ばれる多肢選択の構文になっている。この構文の中で、9行目のswitchは10行目,14行目, 18行目,22行目のcaseラベル, および26行目のdefaultラベルと組になっており、変数 `gengou` の値が `'M'` または `'m'` の時には次に12行目に制御を移し、`'T'` または `'t'` 時には次に16行目に制御を移し、`'S'` または `'s'` の時には次に20行目に制御を移し、`'H'` または `'h'` の時には次に24行目に制御を移し、それ以外の時には次に28行目に制御を移す働きがある。
- プログラム13行目,17行目, 21行目,25行目のbreak文は、9〜28行目のswitch構文から抜け出る働きがある。これらのbreak文が無いと、次のcaseラベルを通り抜けてその後に続く文が次に実行されてしまう。

□**演習 6.22 (元号表記→西暦表記)** 明治は元年から45年まで、大正は元年から15年まで、昭和は元年から64年までである。これらのことを使って入力データのチェックも行うように、例題6.21のプログラムを手直ししてみよ。

□**演習 6.23 (元号表記→西暦表記; scanfの%c)** 例題6.21のプログラムで、変数 `gengou` をint型と宣言して実行すると

```
[motokix205a]$ ./a.out
H15
Input Error!:  gengou='H'
```

という結果になることがある。 この理由を考えよ。 [**Hint.** scanf の%c 変換では値をセットする変数として char 型が暗黙に想定されているため ... 。]

6.6 プログラムを組み立てられない時は ...

慣れて来ると与えられた問題を見ていきなりプログラムを書き下ろすということも出来る様になるであろうが、これは、過去の経験に基づき、

(1) 処理アルゴリズムを十分に理解した上で、
(2) 処理手順を計算機向きに構成し、更に
(3) 計算機向きに表された処理手順を C 言語で表す、

ということを頭の中で全て行っているからに他ならない。

⇨ プログラムを組み立てられない時は、
まず、プログラム作成のどの段階でつまづいているかを見定めた上で、つまづき段階に応じた適切な作業に入らなければならない。

⇨ 以下では、
つまづき段階を特定するための簡単な質問と、各々のつまづき段階に応じた対処例を示す。 質問 (1) から順番に試してみて下さい。

質問 (1)： 必要なデータが全て与えられた時、コンピュータに代わって、紙の上で計算/処理を行えますか?

Yes ⟹ ① 実際に行ってみて下さい。すなわち、
例題 6.6 の「考え方」で示した様に、**紙の上で計算/処理を行ってみる。**入力データに応じて計算の方向が代わって来る場合は、例題 6.12 や例題 6.19 の「考え方」で行った様に、具体的な入力データを幾つか考え、それらに対して紙の上で計算/処理を行ってみる。

② 質問 (2) へ jump。

No ⟹ 自分の手で出来ない計算を、コンピュータに行わせられるはずもありません。

⟹ ⓪ **与えられた問題を人間の手で解くために、関連した文献を調べる。**そして、

① 「Yes」の場合の①〜②を順に行う。

質問 (2)： どういう変数を用意すれば良いか分かりますか?

Yes ⟹ ① 変数を全て列挙し、保持するデータにふさわしい名前を各々に付けてみて下さい。

② 質問 (1) の所で行った紙の上での計算/処理の主要な時点において、それぞれ、①の変数の値がどうなっているかを明記した図/表を作り、計算/処理

の時間順に並べてみて下さい。　各々の変数の値がどうなっているかを表す図/表は、計算/処理の**状態**を表す。

③ 前ステップ②で注目した状態 (i.e. 各変数の値がどうなっているかを表す表) に関して、時間的に隣り合った状態を線で結び、それらの遷移を引き起こすために行った式計算/代入の列を線の脇に書き込むことによって、p.70 に示したものと同様の**状態遷移図** (i.e. 変数値の変遷の様子を表した図) を構成して下さい。

④ 前ステップ③で作った状態遷移図を、p.71 の様に**処理を中心に書き直して**下さい。

⑤ 前ステップ④で作った計算/代入の並んだ図の中に、**条件判断を必要に応じて挿入**することによって、一般的な (i.e. 個別の入力データによらない) アルゴリズムを適用した例として図を再構成して下さい。

⑥ 前ステップ⑤で計算/代入や条件判断結果の並んだ図が出来ているはずである。　この図を基に、一般的なアルゴリズムを**流れ図**として構成して下さい。

うまく流れ図が出来ない場合は、前ステップ⑤の作業に問題がある可能性が高いので、ステップ⑤に戻って下さい。

⑦ 質問 (3) へ jump。

No $\Longrightarrow$ ⓪ 質問 (1) の所で行った紙の上での計算/処理の途中に現れるデータ (e.g. 入力値, 式計算の結果) は全て、使う時点には何らかの変数の中 に記憶されている。これを明示するために、**紙の上の計算途中に現れるデータ全てを箱 ▭ で囲んで**下さい。

① 質問 (1) の所で行った紙の上での計算をプログラムとして表した場合、前ステップ⓪で描いた箱 ▭ は全てプログラム内の変数に相当する。また、紙の上の計算では全ての時点における計算結果が 1 枚の紙の上に現れているので、1 つの変数に相当する箱が何箇所にも現れる。

$\Longrightarrow$ **前ステップ⓪で描いた箱 ▭ を変数領域と見て、それらの脇に各々の保持するデータにふさわしい名前を付けて下さい。**但し、その際、

- プログラム内で同じ変数領域に出来そうな箱には、同じ名前を付ける。
- 箱に付ける名前の種類は出来るだけ少なくし、更には個別の入力値によらずに一定・有限にする。
- 箱に付ける名前は個別の入力値に依存させない。

$\Longrightarrow$ **以下では、箱 ▭ に付けた名前をプログラム内の変数名と考え、その名前の付いた箱で囲まれたデータを変数の値と見る。**

② 「Yes」の場合の②を行う。

すなわち、
質問 (1) の所で行った紙の上での計算/処理の主要な時点において、それぞれ、①の変数の値がどうなっているかを明記した図/表を作り、計算/処理の時間順に並べてみて下さい。　各々の変数の値がどうなっているかを表す図/表は、計算/処理の**状態**を表す。

③ 「Yes」の場合の③を行う。

すなわち、
前ステップ②で注目した状態 (i.e. 各変数の値がどうなっているかを表す表) に関して、時間的に隣り合った状態を線で結び、それらの遷移を引き起こすために行った式計算/代入の列を線の脇に書き込むことによって、p.70 に示したものと同様の**状態遷移図** (i.e. 変数値の変遷の様子を表した図) を構成して下さい。

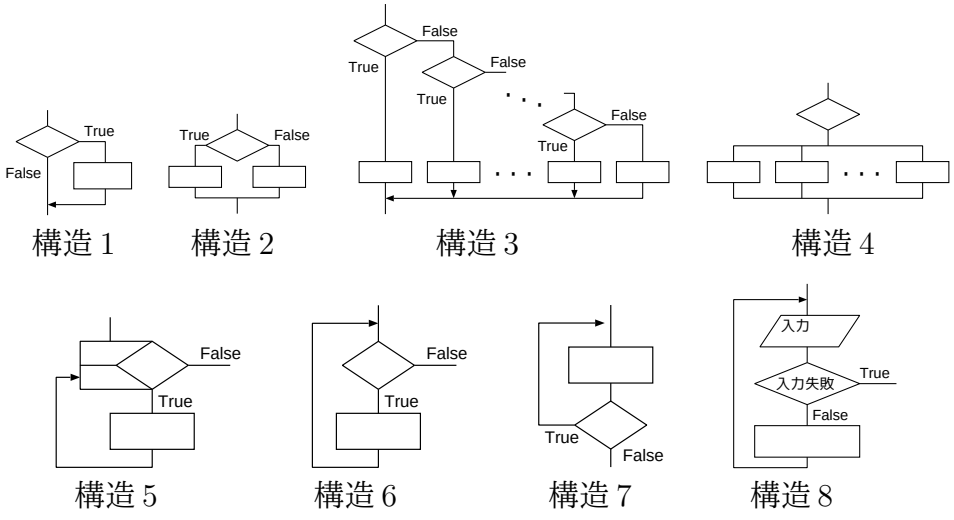
うまく式計算/代入の列が構成できない場合は、先のステップ①の作業に問題がある可能性が高いので、ステップ①に戻って下さい。

④ 「Yes」の場合の④〜⑦を順に行う。

質問 (3)： 前質問 (2) の所で構成した流れ図を基に C プログラムを構成できますか?

Yes $\Longrightarrow$ C プログラムを構成して下さい。

No $\Longrightarrow$ ① 流れ図が次の構造を組み合わせて構成されていることを確認する。



もしこれ以外の構造が含まれていたら、
流れ図を C 言語向きに (上の構造の合成になるように) 構成し直す。

補足：
break 文があるので、構造 5〜8 では繰り返しを途中で抜け出すことも可能である。しかし、この場合、例 6.16 の様に、繰り返しを出る線は一旦 1 つに合流させる必要がある。

② 次の点に留意して C プログラムを構成する。

- 上の図の中の構造 1 は if 文 で表す。 ($\Longrightarrow$ 例 6.1 のアルゴリズム (1))
- 上の図の中の構造 2 は if-else 構文 で表す。
  ($\Longrightarrow$ 例 6.1 のアルゴリズム (2))
- 上の図の中の構造 3 は if-else-if-···-if-else 構文 で表す。
  ($\Longrightarrow$ 例 6.1 のアルゴリズム (3))
- 上の図の中の構造 4 は switch-case 構文 で表す。 ($\Longrightarrow$ 例 6.21)
- 上の図の中の構造 5 は for 文 で表す。 ($\Longrightarrow$ 例 6.6)
- 上の図の中の構造 6 は while 文 で表す。 ($\Longrightarrow$ 例 6.12)
- 上の図の中の構造 7 は do-while 文 で表す。 ($\Longrightarrow$ 例 6.12)

- 上の図の中の構造 8 は 条件部に scanf を含む while 文 で表す。
  ($\Longrightarrow$ 例 6.19)

6.7 付録 制御構造のまとめ —C 文法のまとめ (2)—

6.7.1 関係演算子, 同等演算子、論理演算子

真理値の表し方： C 言語では真理値を次のように表す。

$$\begin{cases} \text{真} \quad \cdots \quad 0 \text{ 以外 (標準は 1)} \\ \text{偽} \quad \cdots \quad 0 \quad \left(\begin{array}{l}\text{浮動小数点数の } \texttt{0.0} \text{ でも、'\texttt{\textbackslash 0}' でも、ポインタ値} \\ \texttt{NULL} \text{ でも良い。}\end{array}\right) \end{cases}$$

演算子一覧：

種類	演算子	意味
関係演算子	<	より小さい
	>	より大きい
	<=	以下
	>=	以上
同等演算子	==	に等しい
	!=	に等しくない
論理演算子	!	論理否定 (単項)
	&&	論理積
	||	論理和

関係演算子：

- 演算結果は int 型の 0 または 1 。

	e1<e2 の値	e1>e2 の値	e1<=e2 の値	e1>=e2 の値
e1>e2 の場合	0	1	0	1
e1=e2 の場合	0	0	1	1
e1<e2 の場合	1	0	1	0

- 優先順位は算術演算子よりも低い。
  $\Longrightarrow$ 例えば、式 a-b<0 は (a-b)<0 と同等。
- 注意 式 -1<0<1 は文法的に誤りではなく、0(偽) という値になる。

> 何故なら、
> 関係演算子は左から右に結合するので、これは
> $\underbrace{\texttt{(-1<0)}}_{1}\texttt{<1} \Longrightarrow 1 < 1 \Longrightarrow 0\text{(偽)}$
> と計算されていくから。

同等演算子：

- 演算結果は int 型の 0 または 1 。

	e1==e2 の値	e1!=e2 の値
e1=e2 の場合	1	0
e1≠e2 の場合	0	1

- 優先順位は算術演算子や関係演算子よりも低い。
  ⟹ 例えば、式 `a<b==a+1<=b` は `(a<b) == ((a+1)<=b)` と同等。
  (見にくい部分は省略可能であってもカッコを付けた方が良い。)
- 注意 if 文を `if (a=1) ...` という風に書くと、変数 a の値が何であっても条件部は真と判定され (a=1) に続く (複合) 文が実行される。

> 何故なら、
> 条件部の「a=1」は代入式であり、その値は代入結果の値である 1 となるから。

論理否定演算子：

- 演算結果は int 型の 0 または 1 。

	!e の値
e=0 の場合	1
e≠0 の場合	0

- 否定演算 ! の優先順位は他の単項演算子 (e.g. 符号反転の-,++) と同じ。
- 注意 条件式 `!(!e)==e` は一般には不成立。

論理積と論理和：

- 演算結果は int 型の 0 または 1 。

	e1&&e2 の値	e1||e2 の値
e1=0, e2=0 の場合	0	0
e1=0, e2≠0 の場合	0	1
e1≠0, e2=0 の場合	0	1
e1≠0, e2≠0 の場合	1	1

演算子の優先順位：

優先順位高 ↑

演算子	結合性
関数の引数をくくる丸括弧	左から右
+ (単項)　- (単項)　++　--　sizeof( )　!　キャスト	右から左
*　/　%	左から右
+　-	左から右
<　<=　>　>=	左から右
==　!=	左から右
&&	左から右
||	左から右
=　+=　-=　*=　/=	右から左

短絡評価：

- e1&&e2 の評価の際、e1 の値が 0 となれば e2 の値の評価は省略され、式全体の値は即座に 0 と結論づけられる。
- e1||e2 の評価の際、e1 の値が 1 となれば e2 の値の評価は省略され、式全体の値は即座に 1 と結論づけられる。

例 6.24 (短絡評価であることの利用) 短絡評価であることを利用すれば、次のような書き方も出来る。

- ```
  do {
    printf("\n 正整数を 2 つ入力して下さい：  ");
  } while ((num_input=scanf("%d %d", &x, &y))==2 && (x<=0 || y<=0));
  if (num_input != 2) {
    printf("エラーメッセージ");
    exit(EXIT_FAILURE);
  }
  ```
- ```
  if (x!=0 && y/x>10) {
    ............
  }
  ```

6.7.2 複合文と空文

複合文：

```
{
    宣言
      ⋮
    宣言
    文
      ⋮
    文
}
```

ブロック：

複合文のうち、宣言が 1 個以上含まれるもの。

空文：

セミコロンだけの文。

6.7.3 条件分岐の制御構造

if 文：

- `if (` 式 `)`
  文
- `if (` 式 `)`
  複合文

すなわち

```
if ( 式 ) {
  文
  ⋮
  文
}
```

if-else 構文：

- 構文は

```
if ( 式 )
  複合文
else
  複合文
```

- 注意 `else` は最も近い `if` と結びつく。
  ⟹ 例えば、

```
if (a == 1)
  if (b == 2)
    printf("***\n");
else
  printf("###\n");
```

は次のものと同等。(間違った字下げはしない様に気を付ける。)

```
if (a == 1) {
  if (b == 2)
    printf("***\n");
  else
    printf("###\n");
}
```

switch 文：

- `if-else` 文を一般化した多分岐条件文。
- 構文は

```
switch ( 整数型の式 ) {
case 整数型の定数式 :
   ⋮
case 整数型の定数式 :
   文の列
   break;
```

```
case 整数型の定数式 :
   ⋮
case 整数型の定数式 :
   文の列
   break;
case 整数型の定数式 :
   ⋮
   ⋮
case 整数型の定数式 :
   ⋮
case 整数型の定数式 :
   文の列
   break;
default:
   文の列
   break;
}
```

- break 文がないと、実行は次の case ラベルを通り抜けてその後に続く文に移る。
- 例えば次のように使う。

```
switch (c) {
case 'a': case 'A':
  ++a_cnt;
  break;
case 'b': case 'B':
  ++b_cnt;
  break;
case 'c': case 'C':
  ++c_cnt;
  break;
default:
  other_cnt;
  break;
}
```

条件演算子：

- 構文は

  式 1 ? 式 2 : 式 3

- その意味は次の通り。

  if 式 1 **then** 式 2 **else** 式 3

- if - else 構文と違って、これを代入式の右側に持って来ることが出来る。

6.7.4 繰り返しの制御

while 文：

```
while ( 式 )
  文
```

for 文：

- 構文は

```
for ( 式1 ; 式2 ; 式3 )
  文
```

ここで、式1 〜式3 の中には、コンマ演算子を使って複数の式を並べることも可能。式2 が省略された場合、繰返しの本体は無条件に実行される。

- 利点　繰り返し制御の変数の操作を先頭にまとめることが出来る。

do 文：

```
do {
    文
    ⋮
    文
} while ( 式 )
```

6.7.5 その他

コンマ演算子：

- 構文は

```
式1 , 式2
```

- 注意　関数の実引数の場所で使いたい場合は、実引数全体を丸括弧で囲む。

break 文：

- 構文は

```
break;
```

- それを含む、最も内側のループ (i.e.for,while, または do-while による繰り返し) または switch 文から抜け出す。
- 例えば次のように使う。

```
while (1) {
  scanf("%lf", &x);
  if (x < 0.0)
    break;
  printf("%f\n", sqrt(x));
}
```

continue 文：

- 構文は

```
continue;
```

- それを含む最も内側のループ (for,while, または do-while) の、現在の繰り返し処理を終了し、次の繰り返し処理に移る。
- 例えば次のように使う。

```
for (i=0; i<TOTAL; ++i) {
  c =getchar();
  if ('0'<=c && c<='9')
    continue;
  ..............
}
```

ここで、getchar は標準入力のストリームから 1 文字だけ (空白も可) 読み込んで、その文字コードの値を返す関数である。[但し、ファイルの終りまたはエラーを検出した時は EOF (マクロ; 通常 -1 が割り当てられている) を返す。関数値の型は char ではなく int である。]

goto 文：

一般に goto 文は避けるべき。

$\Longrightarrow$ 説明省略

7　実数データの扱い

- 実数計算,
- 整数と実数の混合演算, キャスト演算,
- 実数データの入出力 —%f, %e, %g—,
- 数学関数の利用,
- コンパイルはどの様に進むか?,
- 誤差の発生とその対策

色々な形に構成された処理手順を C 言語でどの様に表すかということに焦点を当てるために、これまで整数データ (int 型) しか扱って来なかったが、この節と次の節では実数データの扱いを中心に述べる。

7.1　実数計算

まず手始めに、実数型の変数宣言, 定数の表記法, 実数データの基本的な入出力について、具体的なプログラムを通して説明しよう。

例題 7.1 (円錐の体積；float 型, double 型, マクロ名)　2 つの実数データ r, h を読み込み、

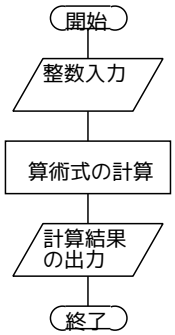
底面の半径が r、高さが h の円錐の体積

を出力する C プログラムを作成せよ。

(考え方)　処理の流れは第 5 節で考えた右図と同じである。 違いは、計算対象が整数データではなく実数データであるということと、計算式が

$$\text{体積} = \frac{\pi r^2 h}{3}$$

ということだけである。



(プログラミング)　実数データを表すためのデータ型として、C 言語では `float` 型，`double` 型，`long double` 型 (これらを総称して**浮動小数点数型**と呼ぶ) の 3 つが用意されている。このうち、良く使われるのは `float` 型 と `double` 型 の 2 つで、これら 2 つのデータ型で処理したプログラムをそれぞれ例示する。

`double` 型で処理するプログラム：
実数データ r, h をそれぞれ `r`，`h` という名前の `double` 型 変数に読み込み `double` 型で体積の計算を行う C プログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl volume-of-cone-double.c  Enter
     1 /* 2つの実数データ r と h を読み込み、          */
     2 /*       底面の半径が r、高さが h の円錐の体積  */
```

```
 3 /* を出力するCプログラム                          */
 4 /*   ---double型で計算する版---                  */

 5 #include <stdio.h>
 6 #define  PI    (3.1415926535897932)   /* 円周率 */

 7 int main(void)
 8 {
 9   double r, h;

10   scanf("%lf%lf", &r, &h);
11   printf("底面の半径が %f, 高さが %f の円錐の体積\n     = %f\n",
12          r, h, PI*r*r*h/3.0);
13   return 0;
14 }
[motoki@x205a]$ gcc volume-of-cone-double.c Enter
[motoki@x205a]$ ./a.out Enter
2.0 5.0 Enter
底面の半径が 2.000000, 高さが 5.000000 の円錐の体積
     = 20.943951
[motoki@x205a]$
```

ここで、

- プログラムの6行目は、Cのソースコードを機械語に翻訳する前に、以降に出て来るPIという名前を全て(3.1415926535897932)という文字列に置き換えることを指示する。 [こういった名前, 行を各々**マクロ (名)**, **マクロ定義**と言う。]
- プログラムの6行目の3.1415926535897932はdouble型の実数値定数を表す文字列である。コンピュータの機種にも依存するが、double型では52ビット,15〜16桁の有効桁を保持することが多いため、ここでは17桁分の精度で指定した。
- プログラム9行目は、2つのdouble型データのための変数領域を確保し、それぞれr, hと名付けることをコンパイラに知らせる宣言文である。 double型は実数を表すための(標準の)倍の精度の内部表現形式に対応したデータ型である。
- プログラム10行目の入力書式中の%lfは読み込んだデータをdouble型(倍精度実数型)の内部表現形式に変換して、別途指定された記憶領域に格納することを指示している。

  補足：
  %fだとfloat型(単精度実数型)の内部表現形式に変換される。
  %lfの中のlは「long」の意。

- プログラム11行目の出力書式中の%f は、別途指定された式の値がdoubleまたはfloat型の内部表現形式に従っていると仮定して、これを10進の文字列に変換して、この%fの場所に出力することを指示する。%lf ではなく %f となっているが、これは誤りではない。printfの場合はscanfの場合と違って、実数値を出力する際にlという拡張子は付けない。

補足：
float 型のデータが printf の引数として指定されていた場合、そのデータは double 型に変換されてから printf に渡される。このため、実数データの出力書式を指定する際には “l” という拡張子は全く意味を為さない。 付けても無視されるだけ？

- プログラムの 12 行目 の 3.0 は double 型の実数値定数を表す文字列である。

float 型で処理するプログラム：
実数データ r, h をそれぞれ r, h という名前の float 型 変数に読み込み float 型で体積の計算を行う C プログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl volume-of-cone-float.c [Enter]
     1 /* 2つの実数データ r と h を読み込み、          */
     2 /*      底面の半径が r、高さが h の円錐の体積   */
     3 /* を出力するCプログラム                        */
     4 /*   ---float型で計算する版---                 */

     5 #include <stdio.h>
     6 #define  PI    (3.1415926f)   /* 円周率 */

     7 int main(void)
     8 {
     9   float r, h;

    10   scanf("%f%f", &r, &h);
    11   printf("底面の半径が %f, 高さが %f の円錐の体積\n      = %f\n",
    12          r, h, PI*r*r*h/3.0f);
    13   return 0;
    14 }
[motoki@x205a]$ gcc volume-of-cone-float.c [Enter]
[motoki@x205a]$ ./a.out [Enter]
2.0 5.0 [Enter]
底面の半径が 2.000000, 高さが 5.000000 の円錐の体積
     = 20.943950
[motoki@x205a]$
```

ここで、

- プログラムの 6 行目 の 3.1415926f は float 型の実数値定数を表す文字列である。double 型定数と区別するために最後に f という文字が付いている。コンピュータの機種にも依存するが、float 型では 23 ビット,6〜7 桁の有効桁を保持することが多いため、ここでは 8 桁分の精度で指定した。
- プログラム 9 行目 は、2 つの float 型データのための変数領域を確保し、それぞれ r, h と名付けることをコンパイラに知らせる宣言文である。 float 型は実数を表すための単精度の内部表現形式に対応したデータ型である。

- プログラム 10 行目 の入力書式中の %f は読み込んだデータを float 型 (単精度実数型) の内部表現形式に変換して、別途指定された記憶領域に格納することを指示している。
- プログラム 11 行目 の出力書式中の%f は、別途指定された式の値が double または float 型の内部表現形式に従っていると仮定して、これを 10 進の文字列に変換して、この%f の場所に出力することを指示する。
- プログラムの 12 行目 の 3.0f は float 型の実数値定数を表す文字列である。double 型定数と区別するために最後に f という文字が付いている。

□**演習 7.2 (円の面積)** 1 つの実数データを変数 r に読み込み、
半径が r の円の面積
を出力する C プログラムを作成せよ。

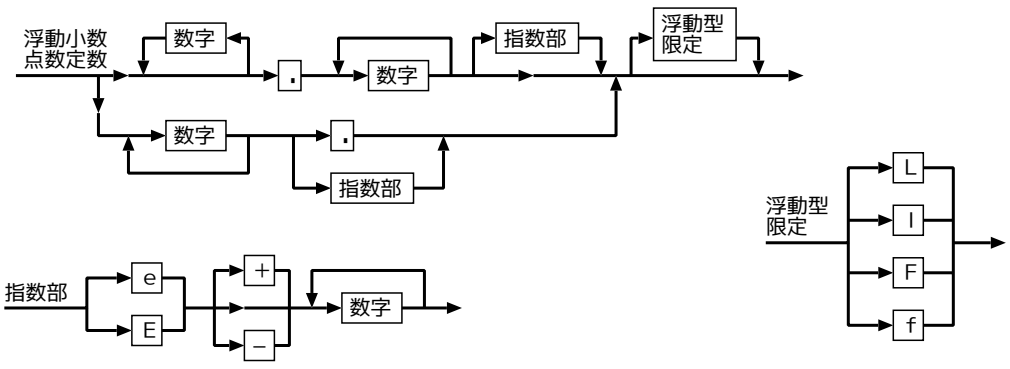
□**演習 7.3 (べき乗)** 実数データ a と非負整数データ k を読み込み、a^k を計算して出力する C プログラムを作成せよ。 但し、ここでは簡単のため $0^0 = 1$ と考えよ。

実数データ間の算術演算： 実数データ間では剰余演算子 % が使えないことを除いて、整数データの場合と同じである。 すなわち、次の算術演算子が使える。

算術演算子	機能
+	加算。但し、単項演算の場合は恒等変換を表す。
-	減算。但し、単項演算の場合は符号反転を表す。
*	乗算。
/	除算。

浮動小数点定数：

- 123.4, 123., .4, 123.4e5, .4E+5, 123e-5, ... といった書き方が出来る。 これらは double 型の定数で、それぞれ $123.4, 123.0, 0.4, 123.4\times10^5, 0.4\times10^5, 123\times10^{-5}, ...$ を表す。
- 定数を float 型にしたければ、最後に f または F という接尾語を付ける。 例えば、123.4f, .4E+5F, ... 。
- 定数を long double 型にしたければ、最後に l または L という接尾語を付ける。 例えば、123.4l, .4E+5L, ... 。



精度と指数部の表現可能な範囲：

- ごく普通の計算機の場合、float 型, double 型データは、各々4 バイト, 8 バイトの領域を占め、10 進で各々約 6 桁, 約 15 桁の精度を持つ。 また、指数部の表現可能な範囲は、およそ、各々 -38〜$+38, -308$〜$+308$ となる。 [指数部の表現可能な範囲は計算機のアーキテクチャに大きく依存する。]

7.2 整数と実数の混合演算, キャスト演算

コンピュータは同じデータ型同士の四則演算回路は持っているが、違ったデータ型の間の四則演算回路、例えば int 型データと float 型データの間の加算を行う回路は持っていない。しかし、算術式の書き方にこの様な制限を設けるとプログラムが書きにくくなるので、C 言語では算術式の中に違ったデータ型同士の四則演算が書けるようになっている。

補足：
コンピュータ内部では演算回路に流し入れるデータの型はやはり揃ってなければならないので、コンパイラが演算対象のデータ型を適宜変換して型を揃える操作を補う。

例題 7.4 (違ったデータ型同士の四則演算) 違ったデータ型同士の四則演算を算術式の中に書いた場合、プログラム実行においては演算の前に適当な型変換によってデータ型が揃えられる。 int 型, float 型 または double 型のデータの組に対して四則演算を行う際、演算の前に実際にどの様な型変換が行われるのかを調べよ。

(考え方) 除算の場合は、被除数 $a = 1$, 除数 $b = 3$ として a/b の演算結果の精度を見ることによってどういう型変換が行われるかを推察することが出来る。

(プログラミング) 被除数 $a = 1$ と除数 $b = 3$ のデータ型を int, float, double の範囲で色々と変えてみて、それらの計算結果を整数と仮定して出力したり実数と仮定して出力したりする C プログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

補足：
この課題では計算結果のデータ型を調べようとしているので、それらの結果を使ってプログラムを書く訳にもいかない。 そこで、出てきた結果を整数と見て出力することも実数と見て出力することも行った。

```
[motoki@x205a]$ nl division-between-different-data-type.c Enter
     1 /* int 型, float 型 または double 型のデータの組に対して */
     2 /* 四則演算を行う際、演算の前に実際にどの様な型変換が */
     3 /* 行われるのかを調べるための C プログラム             */

     4 #include <stdio.h>

     5 int main(void)
     6 {
```

```
 7   printf("a/b の結果を %%13d で表示:\n"
 8          "                   b=3(int)    b=3.0f(float)  b=3.0(double)\n"
 9          "                -------------  -------------  -------------\n");
10   printf("     a=1(int)  %13d", 1/3);
11   printf(                    "  %13d", 1/3.0f);
12   printf(                           "  %13d\n", 1/3.0);
13   printf("a=1.0f(float)  %13d", 1.0f/3);
14   printf(                    "  %13d", 1.0f/3.0f);
15   printf(                           "  %13d\n", 1.0f/3.0);
16   printf("a=1.0(double)  %13d", 1.0/3);
17   printf(                    "  %13d", 1.0/3.0f);
18   printf(                           "  %13d\n\n", 1.0/3.0);

19   printf("a/b の結果を %%13.11f で表示:\n"
20          "                   b=3(int)    b=3.0f(float)  b=3.0(double)\n"
21          "                -------------  -------------  -------------\n");
22   printf("     a=1(int)  %13.11f", 1/3);
23   printf(                       "  %13.11f", 1/3.0f);
24   printf(                                "  %13.11f\n", 1/3.0);
25   printf("a=1.0f(float)  %13.11f", 1.0f/3);
26   printf(                       "  %13.11f", 1.0f/3.0f);
27   printf(                                "  %13.11f\n", 1.0f/3.0);
28   printf("a=1.0(double)  %13.11f", 1.0/3);
29   printf(                       "  %13.11f", 1.0/3.0f);
30   printf(                                "  %13.11f\n", 1.0/3.0);
31   return 0;
32 }
[motoki@x205a]$ gcc division-between-different-data-type.c Enter
[motoki@x205a]$ ./a.out Enter
a/b の結果を %13d で表示:
                   b=3(int)    b=3.0f(float)  b=3.0(double)
                -------------  -------------  -------------
     a=1(int)              0     1610612736     1431655765
a=1.0f(float)     1610612736     1610612736     1431655765
a=1.0(double)     1431655765     1431655765     1431655765

a/b の結果を %13.11f で表示:
                   b=3(int)    b=3.0f(float)  b=3.0(double)
                -------------  -------------  -------------
     a=1(int)            nan  0.33333334327  0.33333333333
a=1.0f(float)  0.33333334327  0.33333334327  0.33333333333
a=1.0(double)  0.33333333333  0.33333333333  0.33333333333
[motoki@x205a]$
```

ここで、

- プログラムの7行目の出力書式中の %% は % という文字を1文字だけ出力することを表す。
- このプログラムでは計算結果のデータ型と printf の変換指定子が適合しないこともあるので、例えば10〜12行目を

```
printf("     a=1(int)  %13d  %13d  %13d\n", 1/3, 1/3.0f, 1/3.0);
```

と書くことは避けた。

補足：
1番目の出力項目と変換指定が適合しないと、それが原因で2番目の出力も乱れることもある。

- プログラムの22〜30行目の出力書式中の %13.11f は、(半角)13文字分の出力場所を確保し、そこに小数点以下11桁の精度で右詰めに出力することを表す。
- 実行結果の中で nan と表示されている箇所があるが、これは非数 (Not A Number) ということを表す。 IEEE754 規格の浮動小数点数の表し方だと、0.0/0.0 や inf(無限大)/inf(無限大) の計算結果が nan になる。

(実行結果の考察) a/b の演算結果は次の3種類に分類できる。

演算結果1 %13d では 0、%13.11f では nan と出力される場合：
これは (int)/(int) の場合だけである。この場合は、int 同士なので型変換は不要で、当然演算結果も int になる。

演算結果2 %13d では 1610612736、%13.11f では 0.33333334327 と出力される場合：
(int)/(float), (float)/(float), (float)/(int) の演算がこの場合に相当する。 このうち、(float)/(float) は float 同士で、演算結果も float になることが確実だから、これらの場合は必要なら int⟶float という型変換が行われ、演算結果は float になると推察される。 実際、演算結果の 0.33333334327 は7桁分の float に相当する精度しかない。

演算結果3 %13d では 1431655765、%13.11f では 0.33333333333 と出力される場合：
(int)/(double), (float)/(double), (double)/(double), (double)/(float), (double)/(int) の演算がこの場合に相当する。 このうち、(double)/(double) は double 同士で、演算結果も double になることが確実だから、これらの場合は必要なら int⟶double や float⟶double の型変換が行われ、演算結果は double になると推察される。実際、演算結果の 0.33333333333 は11桁以上の double に相当する精度があることを示している。

この結果から、演算 a/b は次の様に行われることが分かる。

	b (int)	b (float)	b (double)
a (int)	そのまま演算	float に揃えて演算	double に揃えて演算
a (float)	float に揃えて演算	そのまま演算	double に揃えて演算
a (double)	double に揃えて演算	double に揃えて演算	そのまま演算

補足：
他の演算 $a+b$, $a-b$, $a*b$ についても、これと全く同様の型変換が行われる。

□**演習 7.5 (平均)** 3つの整数値を入力し、それらの平均を出力する C プログラムを作成せよ。

データ型の変換は、**キャスト演算**を使ってプログラムの中で指定することも出来る。

例題 7.6 (階乗; キャスト演算) 正整数データ k を読み込み、その階乗値 $k! = 1\times2\times3\times\cdots\times k$ を double 型実数として 求めて出力する C プログラムを作成せよ。

(考え方) 計算の流れは例題6.6と同じである。例題6.6のプログラムとの違いは、1!,2!,3!,... の値を保持するための変数を int 型ではなくて double 型とするという点だけである。

(プログラミング) C プログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl factorial-double.c Enter
     1 /* 正整数データを読み込み、その階乗値を          */
     2 /* double 型実数として求めて出力する C プログラム */

     3 #include <stdio.h>

     4 int main(void)
     5 {
     6   int    k, i;
     7   double factorial;

     8   printf("何の階乗を求めますか?: ");
     9   scanf("%d", &k);

    10   factorial = 1.0;
    11   for (i=2; i<=k; ++i){
    12     factorial *= (double) i;   /* この時点で factorial = i! */
    13   }

    14   printf("%d! = %21.16g\n", k, factorial);
    15   return 0;
    16 }
[motoki@x205a]$ gcc factorial-double.c Enter
[motoki@x205a]$ ./a.out Enter
何の階乗を求めますか?: 53 Enter
53! = 4.274883284060025e+69
[motoki@x205a]$
```

ここで、

- プログラムの12行目の (double) はデータを double 型に変換する演算子である。C 言語では、どのデータ型に対しても （データ型の名前） という演算子がこういった型変換のために用意されており、**キャスト演算子**と呼ばれている。

補足：
この 12 行目のキャスト演算の場合は、変数 `factorial` に入っている `double` 型のデータと乗算を行うために、省略しても自動的に補われる。 しかし、不注意による間違いを出来るだけ避けるために、行う予定の型変換はこの 12 行目の様に明示するのが好ましい。

- プログラムの 14 行目 の出力書式中の `%21.16g` は、(半角)21 文字分の出力場所を確保し、そこに (最大) 有効桁 16 桁の精度で (出来れば指数部なしの形で) 出力することを表す。

□演習 7.7 (e の計算) ネピア数 (Napier number)

$$e = \lim_{x\to\infty}(1+\frac{1}{x})^x = \sum_{i=0}^{\infty}\frac{1}{i!} = 2.718281828\cdots\cdots$$

を小数点以下 14 桁まで計算するためには、$18! \approx 6.4\times10^{15}$ に注目して近似式

$$\begin{aligned} e &\approx \sum_{i=0}^{17}\frac{1}{i!} \\ &= (((\cdots((1\cdot\frac{1}{17}+1)\frac{1}{16}+1)\cdots)\frac{1}{3}+1)\frac{1}{2}+1)\frac{1}{1}+1 \end{aligned}$$

の計算を正確に行なえばよい。この計算を行う C プログラムを作成せよ。

算術計算の際の自動型変換：

- `int` 型,`float` 型,`double` 型の間では、四則演算 $a+b,\ a-b,\ a*b,\ a/b$ はどれも次の様に行われる。

	b (int)	b (float)	b (double)
a (int)	そのまま演算	float に揃えて演算	double に揃えて演算
a (float)	float に揃えて演算	そのまま演算	double に揃えて演算
a (double)	double に揃えて演算	double に揃えて演算	そのまま演算

- 実数 $\to$ 整数 間の型変換が実際にどう行われるかについては計算機に依存する。 [切捨て、切り上げ、四捨五入のいずれか。]

代入の際の自動型変換：

- 代入 変数等 = 式 において両辺の型が違えば、式 の値は 変数等 の型に強制的に変換される。

キャスト演算子：

- 明示的に型変換を行うことが出来る。
- 式の値をデータ型という型に変換したければ、次の様に書く。
  (データ型) 式
- キャストは単項演算子。
- 他の単項演算子 (e.g. 符号反転の`-`,`++`) と同じ優先順位、結合性 (右から左) を持つ。

例 7.8 (キャスト演算の優先順位) `(float) i+3` は `((float)i) + 3` と同等である。

7.3 実数データの入出力 — %f，%e，%g —

scanf や printf の下で実数データの入出力を行う際は、書式指定の中で %f，%e または %g 変換指定子を使う。 (⟹ printf の詳細については 5.5 節を、scanf の詳細については 5.6 節を参照して下さい。)

例題 7.9 (実数データの出力書式の比較) 指数関数 $f(x) = 3.14\times10^x$ の $x = -5, -4, -3, -2, ..., 7, 8$ に対する値が printf( ) に用意されている 3 つの変換指定子 e，f，g によって実際にどの様に出力されるのかを調べよ。

(考え方) それぞれの $f(x)$ の値が、例えば 4 つの変換指定 %12.5e，%12.5g，%#12.5g，%12.5f によって実際にどのように出力されるのかを比較すれば良い。

(プログラミング) $x = -5, -4, -3, -2, ..., 7, 8$ に対して $f(x) = 3.14\times10^x$ の値を順に出力するだけの単純な繰り返しである。ただ、数学関数のライブラリから冪乗関数 (pow(,)) をわざわざ呼び出すこともないので、ここでは $x = -5, -4, -3, -2, ..., 7, 8$ に対する $f(x)$ の値を次のように計算する。

$$\begin{aligned} f(-5) &= \text{C 言語上で double 型定数 3.14e-5 に相当} \\ f(-4) &= f(-5)\times10.0 \\ f(-3) &= f(-4)\times10.0 \\ f(-2) &= f(-3)\times10.0 \end{aligned}$$

$$\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$$

x の値を保持する int 型変数を x、$f(x)$ の値を保持する double 型変数を fx とすれば、$x = -5, -4, -3, -2, ..., 7, 8$ のそれぞれの値に対して $f(x)$ の値を %12.5e，%12.5g，%#12.5g，%12.5f という 4 つの変換指定で出力するプログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl printf-e-f-g-conversion.c Enter
     1 #include <stdio.h>

     2 int main(void)
     3 {
     4   int    x;
     5   double fx;

     6   printf("-------------------------------------------------------\n"
     7          "関数 f(x)=3.14*10^x の x=-5,-4,-3, ..., 7,8 に対する値が\n"
     8          "e,f,g 変換記述子によって実際にどの様に出力されるかを見る。\n"
     9          "-------------------------------------------------------\n"
    10          " x     %%12.5e        %%12.5g        %%#12.5g       %%12.5f\n"
    11          "--  ------------  ------------  ------------  ------------\n");

    12   fx=3.14e-5;
```

```
    13   for (x=-5; x<=8; x++) {
    14     printf("%2d  %12.5e  %12.5g  %#12.5g  %12.5f\n", x, fx, fx, fx, fx);
    15     fx *= 10.0;
    16   }
    17   return 0;
    18 }
[motoki@x205a]$ gcc printf-e-f-g-conversion.c Enter
[motoki@x205a]$ ./a.out Enter
------------------------------------------------------
関数 f(x)=3.14*10^x の x=-5,-4,-3, ..., 7,8 に対する値が
e,f,g 変換記述子によって実際にどの様に出力されるかを見る。
------------------------------------------------------
 x      %12.5e         %12.5g         %#12.5g        %12.5f
--   ------------   ------------   ------------   ------------
-5    3.14000e-05       3.14e-05    3.1400e-05        0.00003
-4    3.14000e-04       0.000314    0.00031400        0.00031
-3    3.14000e-03        0.00314     0.0031400        0.00314
-2    3.14000e-02          0.0314      0.031400        0.03140
-1    3.14000e-01           0.314       0.31400        0.31400
 0    3.14000e+00            3.14        3.1400        3.14000
 1    3.14000e+01            31.4        31.400       31.40000
 2    3.14000e+02             314        314.00      314.00000
 3    3.14000e+03            3140        3140.0     3140.00000
 4    3.14000e+04           31400        31400.    31400.00000
 5    3.14000e+05       3.14e+05    3.1400e+05   314000.00000
 6    3.14000e+06       3.14e+06    3.1400e+06  3140000.00000
 7    3.14000e+07       3.14e+07    3.1400e+07 31400000.00000
 8    3.14000e+08       3.14e+08    3.1400e+08 314000000.00000
[motoki@x205a]$
```

ここで、

- プログラムの 14 行目 の出力書式中の %12.5f は、(半角)12 文字分の出力場所を確保し、そこに小数点以下 5 桁の精度で右詰めに出力することを表す。 実行結果に見られるように、12 というのは 最小の 出力フィールド幅であって、整数部が大きすぎて全体で 12 文字以内に収まらない場合はその分だけ出力フィールド幅は伸びる。
- プログラムの 14 行目 の出力書式中の %12.5e は、(半角)12 文字分の出力場所を確保し、そこに指数部付きの次の形式で右詰めに出力することを表す。

  [-] 0 以外の数字 . $\underbrace{\text{数字列}}_{\text{3 桁}}$ e ± 4 桁以上の数字列

- プログラムの 14 行目 の出力書式中の %12.5g は、(半角)12 文字分の出力場所を確保し、そこに

  出力データに 5 桁 (以上) の有効桁があることを仮定して
  その桁まで e 変換の形, f 変換の形で出力を行った時の短い方

  を右詰めに出力することを表す。小数部末尾の 0 と小数点 は省略される。

- プログラムの 14 行目 の出力書式中の `%#12.5g` は、基本的には `%12.5g` と同じである。但し、ここでは `#` がフラグとして指定されているので、小数部末尾の `0` は省略されない。

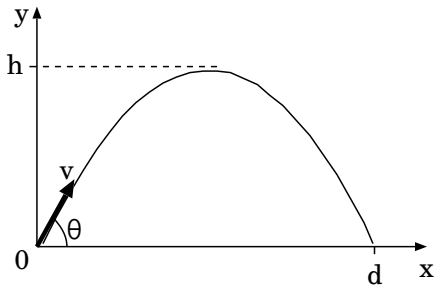
7.4 数学関数の利用

- C 言語では、三角関数, 対数関数, 指数関数, 冪乗関数, 平方根関数, ... 等の数学的関数は標準ライブラリの中で提供されている。
  $\Longrightarrow$ 数学的関数を使いたければ、
  - ◇ C プログラムの最初に `#include <math.h>` の宣言をする。
  - ◇ コンパイル時には `-lm` オプションを (最後に) 付ける。
- 数学的関数の引数、関数値はほとんどが `double` 型である。

例題 7.10 (ボール投げ; 三角関数) 初速度 vm/sec で地面に対して θ の角度でボールを投げた時、ボールの最高点の高さ h, 届く距離 d, 地面に落ちるまでの時間 τ は、重力加速度 $g = 9.8\text{m/sec}^2$ を使って

$$h = \frac{v^2 \sin^2\theta}{2g}, \quad d = \frac{v^2 \sin 2\theta}{g}, \quad \tau = \frac{2v\sin\theta}{g}$$

という風に求めることが出来る。 初速度 v を読み込み、その v に対して角度を $\theta = 5°, 10°, 15°, \ldots, 90°$ と変えた時に最高点の高さ h, 届く距離 d, 落ちるまでの時間 τ がどの様に変わるかを表の形に見易く表示する C プログラムを作成せよ。



補足 (ボール投げの物理学)：
時刻 t にボールがある座標を (x, y) とすると、微分方程式

$$\frac{d^2x}{dt^2} = 0, \quad \frac{d^2y}{dt^2} = -g$$

が得られる。これを初期条件

$$x|_{t=0} = y|_{t=0} = 0, \quad \left.\frac{dx}{dt}\right|_{t=0} = v\cos\theta, \quad \left.\frac{dy}{dt}\right|_{t=0} = v\sin\theta$$

を用いて解くと、

$$x = v\cos\theta\, t, \quad y = -\frac{g}{2}t^2 + v\sin\theta\, t$$

が得られる。 ここで、$y = -\frac{g}{2}(t - \frac{v\sin\theta}{g})^2 + \frac{v^2\sin^2\theta}{2g}$ と書き直せるから、ボールの最高点の高さ h は

$$h = \frac{v^2\sin^2\theta}{2g}$$

となる。また、$y = -\frac{g}{2}t(t - \frac{2v\sin\theta}{g})$ と書き直せるから、ボールが地面に落ちる時刻 τ は

$$\tau = \frac{2v\sin\theta}{g}$$

であり、ボールの届く距離 d (すなわち、その時の y) は

$$d = y|_{t=\tau} = \frac{2v^2\sin\theta\cos\theta}{g} = \frac{v^2\sin 2\theta}{g}$$

となる。

(考え方) 計算手順自体は $\theta = 5°, 10°, 15°, \ldots, 90°$ のそれぞれに対して $h = \frac{v^2\sin^2\theta}{2g}$, $d = \frac{v^2\sin 2\theta}{g}$, $\tau = \frac{2v\sin\theta}{g}$ を計算して出力するだけの単純な繰り返しであるので、処理の構造は例題 6.4 と同じである。

(プログラミング) ボールの初速度 v を保持するための変数を v, 地面に対する角度 θ を保持するための変数を degree, 角度 θ をラジアンに変換した値を保持するための変数を radian, $\sin\theta$ の値を一時的に保持するための変数を sin_radian としてプログラムを構成した。 この C プログラムと、これをコンパイル/実行している様子を次に示す。 (下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl throw-a-ball.c  Enter
     1 /* ボールの初速度 v を読み込み、地面に対する角度を */
     2 /* 5 度, 10 度, 15 度,..., 90 度 と変えていった時の     */
     3 /* ボールの最高点の高さ h, 届く距離 d, 落ちるまで  */
     4 /* の時間 t がどの様に変わるかを表の形に見易く表示 */
     5 /* する C プログラム                                  */

     6 #include <stdio.h>
     7 #include <math.h>

     8 #define  PI  (3.1415926535897932)  /* 円周率 */
     9 #define  G   (9.8)                 /* 重力加速度 */

    10 int main(void)
    11 {
    12   int    degree;
    13   double v, radian, sin_radian;
```

```
14   printf("Input the velocity: ");
15   scanf("%lf", &v);

16   printf("\nWhen velocity = %g m/sec, ...\n\n"
17          "degree    height     distance      time\n"
18          "------  ----------  ----------  ----------\n",
19          v);

20   for (degree=5; degree<=90; degree+=5) {
21     radian = (double)degree*PI/180.0;
22     sin_radian = sin(radian);
23     printf("%4d    %10.3f  %10.3f  %10.3f\n",
24            degree,
25            v*v*sin_radian*sin_radian/(2.0*G), /* 最高点の高さ */
26            v*v*sin(2.0*radian)/G,             /* 届く距離     */
27            2.0*v*sin_radian/G);           /* 落ちるまでの時間 */
28   }
29   return 0;
30 }
[motoki@x205a]$ gcc throw-a-ball.c -lm Enter
[motoki@x205a]$ ./a.out Enter
Input the velocity: 35.0 Enter

When velocity = 35 m/sec, ...

degree    height     distance      time
------  ----------  ----------  ----------
   5         0.475      21.706       0.623
  10         1.885      42.753       1.240
  15         4.187      62.500       1.849
  20         7.311      80.348       2.443
  25        11.163      95.756       3.019
  30        15.625     108.253       3.571
  35        20.562     117.462       4.097
  40        25.823     123.101       4.591
  45        31.250     125.000       5.051
  50        36.677     123.101       5.472
  55        41.938     117.462       5.851
  60        46.875     108.253       6.186
  65        51.337      95.756       6.474
  70        55.189      80.348       6.712
  75        58.313      62.500       6.899
  80        60.615      42.753       7.034
```

```
   85        62.025      21.706       7.116
   90        62.500       0.000       7.143
[motoki@x205a]$
```

ここで、

- プログラム 7 行目 は、/usr/include/math.h というファイルの中身をこの場所に挿入してコンパイル作業を行うことを指示する。プログラムの 22 行目,26 行目 で sin という数学的関数を使っているので、これらの関数の引数の型, 関数値の型をコンパイラに知らせるために、この # で始まる行 (すなわち**プリプロセッサ指令**) が必要となる。
- プログラム 21 行目 は角度の単位 (度) をラジアンに変換している。
- プログラム 22 行目 では、ライブラリ関数を呼び出すことによって sin radian の値を計算している。 高さ h, 距離 d, 時間 t の計算の前にこれだけ特別に計算しているのは、h, d, t の計算の中でライブラリ関数の呼び出し回数を出来るだけ抑えるためである。
- gcc コマンド の最後に付けた-lm オプションは、関数の翻訳コードを繋げて実行コードを作る際に、別途用意されている数学的関数の翻訳コードも取り込むことを指示している。 [数学的関数以外の標準ライブラリ関数、例えば printf や scanf などについては、オプション指定しなくても自動的に関数の翻訳コードが取り込まれるが、数学的関数については明示しないといけない。 この講義ノートの 7.5 節, 付録 A 節を参照。]

補足：
-lm オプションを付けないと次の様にコンパイルエラーになる。

```
[motoki@x205a]$ gcc throw-a-ball.c
/tmp/cc4TDjRQ.o: In function ‘main’:
/tmp/cc4TDjRQ.o(.text+0x83): undefined reference to ‘sin’
/tmp/cc4TDjRQ.o(.text+0xc9): undefined reference to ‘sin’
collect2: ld returned 1 exit status
[motoki@x205a]$
```

注目点：

- 数学的関数を使う場合、#include <math.h> という行は数学的関数を呼び出す部分を間違いなく翻訳するために必要となり、cc コマンドの-lm オプションは数学的関数の翻訳コードも取り込んで完全な実行コードを作るために必要となる。

□演習 7.11 (ヘロンの公式) ヘロンの公式によれば、3 辺の長さが a, b, c の三角形の面積 S は

$$S = \sqrt{s(s-a)(s-b)(s-c)}$$

但し、$s = (a+b+c)/2$

と求めることが出来る。この公式を用いることによって、三角形の 3 辺の長さを読み込みその面積を出力する C プログラムを作成せよ。

C 言語においては、次のような数学的関数が標準ライブラリに用意されている。

機能	関数名 (引数の並び)	引数の型	関数値の型	説明
切捨て	floor(a)	double	double	$\lfloor a \rfloor$
切上げ	ceil(a)	double	double	$\lceil a \rceil$
剰余	fmod(a, b)	double	double	$a \geq 0$ の時は $a - \lvert b \rvert \times \lfloor a/\lvert b \rvert \rfloor$
				$a < 0$ の時は $a - \lvert b \rvert \times \lceil a/\lvert b \rvert \rceil$
絶対値	fabs(a)	double	double	$\lvert a \rvert$
平方根	sqrt(a)	double	double	$\sqrt{a}$
べき乗	pow(a, b)	double	double	a^b
	ldexp(a, n)	double と int	double	$a \times 2^n$
指数	exp(a)	double	double	e^a
自然対数	log(a)	double	double	$\log_e a$
常用対数	log10(a)	double	double	$\log_{10} a$
正弦	sin(a)	double	double	$\sin a$, 但し a はラジアン
余弦	cos(a)	double	double	$\cos a$, 但し a はラジアン
正接	tan(a)	double	double	$\tan a$, 但し a はラジアン
逆正弦	asin(a)	double	double	$\sin^{-1} a \in [-\frac{\pi}{2}, \frac{\pi}{2}]$
逆余弦	acos(a)	double	double	$\cos^{-1} a \in [0, \pi]$
逆正接	atan(a)	double	double	$\tan^{-1} a \in [-\frac{\pi}{2}, \frac{\pi}{2}]$
	atan2(a, b)	double	double	$\tan^{-1} \frac{a}{b} \in [-\frac{\pi}{2}, \frac{\pi}{2}]$
双曲線正弦	sinh(a)	double	double	$\sinh a$
双曲線余弦	cosh(a)	double	double	$\cosh a$
双曲線正接	tanh(a)	double	double	$\tanh a$
整数部と小数部に分離	modf(a, ptr)	double と (double *)	double	a の小数部 (符号は a と同じ) を返し、a の整数部を ptr の指す領域に格納
仮数部と指数部に分離	frexp(a, ptr)	double と (int *)	double	関数呼び出し直後は $a =$ (関数値) $\times 2^{(ptr \text{ の指す int 型の値})}$

補足： C 言語では、数学的関数には分類されていないが次のような関数も標準ライブラリに用意されている。 [RAND_MAX は /usr/include/stdlib.h の中で定義されたマクロ名、div_t と ldiv_t は /usr/include /stdlib.h の中で定義された「構造体」の名前である。構造体についてはこの講義ノートの第 11.6 節を参照して下さい。]

機能	関数名 (引数の並び)	引数の型	関数値の型	説明
乱数	rand()	なし	int	区間 [0, RAND_MAX) の間の疑似乱数
	srand()	unsigned int	なし	疑似乱数発生器の状態を初期化
絶対値	abs(a)	int	int	$\lvert a \rvert$
	labs(a)	long	long	$\lvert a \rvert$
商と剰余	div(a,b)	int	div_t	a を b で割った時の商と剰余の組
	ldiv(a,b)	long	ldiv_t	a を b で割った時の商と剰余の組

7.5 コンパイルはどの様に進むか?

この講義ノートの付録 A 節でも触れられている様に、cc コマンド／gcc コマンドによる C プログラムの翻訳作業は実際には次の順に行われる。

① 前処理 (#include や#define で始まる行の処理等、すなわちヘッダファイルの取り込み, マクロの展開, 注釈の除去など。)

② コンパイル (各々の関数定義を機械語に翻訳, 場合によっては最適化も行う。)

③ リンク (各々の関数の翻訳コードを繋げて 1 つの実行コードを作る。)

これらの作業の様子を図示すると図 1 の様になる。

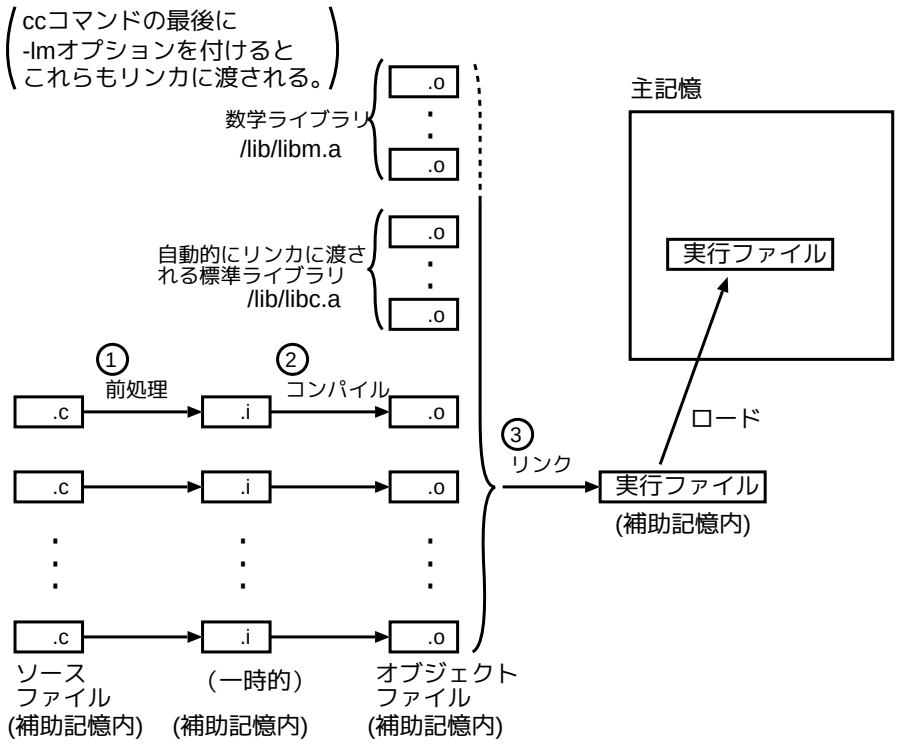


図 1: コンパイル作業の流れ

7.6 誤差の発生とその対策

この節では、実数計算のアルゴリズムを設計する際に注意すべきことを指摘しておく。そのために、まず (数学的に見ると) 奇妙な実行結果を幾つか示し、それらの起こる原因について考えてみる。 [これらの例は、数学的に正しいアルゴリズムでも、コンピュータで計算すると (数学的に) 誤った結果が得られる可能性があることを示している。]

例題 7.12 (メモリの有限性, 2 進-10 進変換による誤差) $x = 0.1$ とした時、3 つの計算式

(1) $(10^{16} + 1) - 10^{16}$,

(2) $x \times 10 - 1$,

(3) $x \times 100010 - 10000$

の値は、数学的には各々 1, 0, 1 になる。 これらの式をコンピュータで計算すると、実際にどういう計算結果が得られるか調べよ。

(プログラミング) 指定された計算を行うCプログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl error-by-finiteness-of-memory.c Enter
     1 /* メモリの有限性、2進-10進変換に起因する計算誤差 */

     2 #include <stdio.h>

     3 int main(void)
     4 {
     5   double x=0.1;

     6   printf("(1) %21.16g\n", (1e16+1)-1e16);
     7   printf("(2) %21.16g\n", x*10.0-1.0);
     8   printf("(3) %21.16g\n", x*100010-10000);
     9   return 0;
    10 }
[motoki@x205a]$ gcc error-by-finiteness-of-memory.c Enter
[motoki@x205a]$ ./a.out Enter
(1)                      0     ......正解 1
(2) 5.551115123125783e-17     ......正解 0
(3)     1.000000000000555     ......正解 1
[motoki@x205a]$
```

(誤差発生の原因について) いずれの計算結果にも計算誤差が生じている。これらの誤差は、基本的にはどれも

> 数値を記憶する領域が有限であり、そのため
> **記憶された実数データが本来の値の近似似すぎない**

ことに起因する。 特に (1) の計算結果からは、実数データ $10^{16}+1$ を最も良く表す (i.e. 近似する) 内部表現が 10^{16} の内部表現と同一になっていることが分かる。 また、(2),(3) の計算結果は、

> コンピュータの数値の記憶方式が10進ではなく2進であり、
> **10進で桁数の小さな小数も2進では循環 (従って無限) 小数になり得る**

ことにも起因する。実際、10進小数の 0.1 を2進に**基数変換** (radix conversion) すると、$0.0\dot{0}01\dot{1}$ という循環小数になる。これを有限ビットで表そうとすると、必然的にこの無限小数のある桁以降は切捨て (または切り上げ) られてしまう。

> 補足：
> (2),(3) の場合は、基数変換による誤差が発生した後、(程度の差はあるがいずれも) 次の例題 7.13 で説明される「桁落ち」によって誤差がクローズアップされてしまった。

例題 7.13 (桁落ち) 数学的に等価な 2 つの式

$$(1)\ \frac{1}{x}-\frac{1}{x+1},$$

$$(2)\ \frac{1}{x(x+1)}$$

の値を $x = 10^0,\ 10^3,\ 10^6,\ \ldots,\ 10^{21}$ に対してコンピュータで計算して比較してみよ。

(考え方) 計算手順自体は $x = 10^0,\ 10^3,\ 10^6,\ \ldots,\ 10^{21}$ のそれぞれに対して $\frac{1}{x}-\frac{1}{x+1}$ と $\frac{1}{x(x+1)}$ を計算して出力するだけの単純な繰り返しであるので、処理の構造は例題 6.4 と同じである。

(プログラミング) 実数データ x を保持するために x という double 型変数を用意し、これを for 文の制御変数としてプログラムを構成した。 この C プログラムと、これをコンパイル/ 実行している様子を次に示す。 (下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl error-cancel-of-sig-digits.c Enter
     1 /* x=10^0,10^3,10^6, ..., 10^21 に対して 2 つの式    */
     2 /* 1/x-1/(x+1) と 1/(x*(x+1)) の計算値を比較して、*/
     3 /* 桁落ちによる相対誤差の拡大の様子を調べる          */

     4 #include <stdio.h>

     5 int main(void)
     6 {
     7   double x;

     8   printf("   x        (1) 1/x-1/(x+1)      (2) 1/(x*(x+1))\n"
     9          "-------  --------------------  --------------------\n");

    10   for (x=1.0; x<1e22; x*=1000.0)
    11     printf("%7.2g  %21.16g  %21.16g\n",
    12            x,
    13            1.0/x-1.0/(x+1.0),
    14            1.0/(x*(x+1.0)));
    15   return 0;
    16 }
[motoki@x205a]$ gcc error-cancel-of-sig-digits.c Enter
[motoki@x205a]$ ./a.out Enter
   x        (1) 1/x-1/(x+1)      (2) 1/(x*(x+1))
-------  --------------------  --------------------
      1                   0.5                   0.5
  1e+03  9.990009990009991e-07  9.990009990009991e-07
  1e+06  9.999990000009847e-13  9.999990000010001e-13
```

```
  1e+09  9.999999989616781e-19          9.99999999e-19
  1e+12  1.000000019541481e-24       9.99999999999e-25
  1e+15  1.000039123623072e-30    9.99999999999999e-31
  1e+18    9.4039548065783e-37  9.999999999999999e-37
  1e+21                      0                  1e-42
[motoki@x205a]$
```

(実験結果の考察) 絶対値の十分小さな ϵ に対して $\frac{1}{1+\epsilon} \approx 1-\epsilon$ と近似でき、$|\frac{1}{1+\epsilon}-(1-\epsilon)| \approx \epsilon^2$ であるから、十分大きな x に対して $\frac{1}{x(x+1)} = x^{-2}\frac{1}{1+x^{-1}} \approx x^{-2}(1-x^{-1})$ である。 それゆえ、2 つの計算結果のうち (2) はほぼ正しい計算値になっている。従って、(1) の方は x の値が大きくなるにつれて徐々に有効桁が失われていることになる。この原因としては、(1) では、ほぼ同じ大きさの数の間の減算になり、上位の有効桁が打ち消し合ったことが挙げられる。この様に、上位の有効桁が打ち消し合って計算結果の有効桁が一挙に失われる現象を**桁落ち**と呼んでいる。桁落ちは、次の様に手計算の際にも起きる。

$$\begin{array}{rl} 1.234567 & \cdots \ 7\text{桁の有効数字} \\ -\ 1.234566 & \cdots \ 7\text{桁の有効数字} \\ \hline 0.000001 & \cdots \ 1\text{桁の有効数字} \end{array}$$

□演習 7.14 (どちらの計算式が良いか) `double` 型変数 `x` に対して式

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

の値を計算するのに、次のどちらの算術式が精度の点で好ましいか？ その理由も述べよ。

(a) `sqrt(fabs(x)+1)-sqrt(fabs(x))`

(b) `1/(sqrt(fabs(x)+1)+sqrt(fabs(x)))`

例題 7.15 (情報落ち) $\log_e 2 = \displaystyle\sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k} = 0.69314718\cdots$ の近似式

$$a = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots + \frac{1}{99999} - \frac{1}{100000}$$

の値を次の 2 通りの順序で計算して、それらの結果を真値 $a = 0.693142180\cdots$ と比較してみよ。

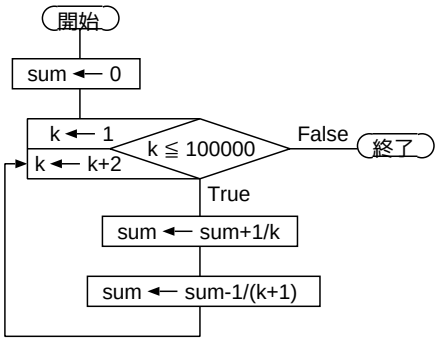
(1) $\dfrac{1}{1} - \dfrac{1}{2} + \dfrac{1}{3} - \dfrac{1}{4} + \cdots + \dfrac{1}{99999} - \dfrac{1}{100000}$ (定義通りに累算)

(2) $-\dfrac{1}{100000} + \dfrac{1}{99999} - \cdots - \dfrac{1}{4} + \dfrac{1}{3} - \dfrac{1}{2} + \dfrac{1}{1}$ (定義の逆順に累算)

(考え方) 誤差が打ち消しあって偶然真値に近い値が出るということもあるので、`double` 型だけではなく `float` 型でも計算することにする。また、真値を知り計算値と比較するために (2) の計算を `long double` 型でも行うことにする。 処理の組み立てに関しては、何を繰り返すかに注意する必要がある。例えば (1) の計算で、各々の項の加減算を繰り返しの単位にすると、加算と減算を毎回切替える必要があるので面倒になる。次の様に計算すると、繰り返し処理を簡単に書き表すことが出来る。

$$0+\underbrace{\frac{1}{1}-\frac{1}{2}}_{\text{繰り返し}}+\underbrace{\frac{1}{3}-\frac{1}{4}}_{\text{繰り返し}}+\cdots+\underbrace{\frac{1}{99999}-\frac{1}{100000}}_{\text{繰り返し}}$$

(プログラミング) 例えば (1) の順序の累算は右図の様に行えば良い。逆の順序の累算も同程度の規則的な繰り返しで書ける。そこで、float 型,double 型,long double 型の累算値を保持するために各々 sum_float, sum_double, sum_long_double という名前の変数を用意してプログラムを構成した。この C プログラムと、これをコンパイル/ 実行している様子を次に示す。(下線部はキーボードからの入力を表す。)




```
[motoki@x205a]$ nl error-loss-of-trailing-digits.c Enter
     1 /* 累算の順序によって計算結果が変わる例 */

     2 #include <stdio.h>

     3 int main(void)
     4 {
     5   int         k;
     6   float       sum_float;
     7   double      sum_double;
     8   long double sum_long_double;

     9   printf("    float で計算         double で計算\n"
    10          "    -----------    --------------------\n");

    11   /* (1) の計算 */
    12   sum_float=0.0f;
    13   sum_double=0.0;
    14   for (k=1; k<100000; k+=2) {
    15     sum_float  += 1.0f/(float)k;
    16     sum_float  -= 1.0f/(float)(k+1);
    17     sum_double += 1.0/(double)k;
    18     sum_double -= 1.0/(double)(k+1);
    19   }
    20   printf("(1) %11.9f    %20.18f\n", sum_float, sum_double);

    21   /* (2) の計算 */
    22   sum_float =0.0f;
    23   sum_double=0.0;
    24   sum_long_double=0.0L;
    25   for (k=100000; k>1; k-=2) {
```

```
    26      sum_float -= 1.0f/(float)k;
    27      sum_float += 1.0f/(float)(k-1);
    28      sum_double -= 1.0/(double)k;
    29      sum_double += 1.0/(double)(k-1);
    30      sum_long_double -= 1.0L/(long double)k;
    31      sum_long_double += 1.0L/(long double)(k-1);
    32    }
    33    printf("(2) %11.9f    %20.18f\n", sum_float, sum_double);
    34    printf("    %13.11Lf  %30.28Lf  (long_double で計算:真値)\n",
    35           sum_long_double, sum_long_double);
    36    return 0;
    37 }
[motoki@x205a]$ gcc error-loss-of-trailing-digits.c Enter
[motoki@x205a]$ ./a.out Enter
    float で計算         double で計算
    -----------     --------------------
(1) 0.693133771     0.693142180584982004      ..........  下線部は真値と違う箇所
(2) 0.693142176     0.693142180584945367
    0.69314218058   0.693142180584945309424101l120  (long_double で計算:真値)
[motoki@x205a]$
```

ここで、

- プログラム 8 行目 は `long double` 型変数の宣言である。
- プログラム 24 行目 の `0.0L`, 30〜31 行目 の `1.0L` は `long double` 型の定数を表す。
- プログラム 34 行目 の書式指定において、2 つの f 型変換 `%13.11Lf`, `%30.28Lf` の中で使われている `L` は対応するデータが `long double` 型であることを表す。

(実験結果の考察) `long double` 型で計算した結果と比較することにより、(1) の順序では `float` 型で 4 桁, `double` 型で 13 桁の精度の計算結果が得られ、(2) の順序では `float` 型で 7 桁, `double` 型で 16 桁の精度の計算結果が得られていることが分かる。(2) の順序で累算すると各々のデータ型で最高に近い精度が得られている一方で、(1) では (2) より 3 桁程精度が落ちている。この違いは、同じ有効桁の数でも絶対値の大きさが桁違いに違うと、それらの 2 数を加減算すると小さい方の下位の有効桁が失われてしまうことによる。

$$\begin{array}{r} 1.234567 \\ +\ 0.04321098 \\ \hline 1.27777798 \end{array}$$

この様な誤差発生の現象を**情報落ち** (loss of trailing digits) または**情報埋没** (swamp) と呼んでいる。 加算 $a+b$ (または減算 $a-b$) で情報落ちが起こる場合、発生する誤差の上限はほぼ $\max\{|a|,|b|\}$ に比例する。それゆえ、加減算を繰り返す場合、絶対値の小さな数同士の加減算の割合が増える様に加減算の順序を工夫した方が、誤差の累積は小さく抑えられる。 例えば、IEEE 規格 754 の単精度で実数データを表す場合を考える。この場合、仮数部は実質 24 ビットで表されるので、$a+b$ という加算の際に発生する誤差の上限は

$$|\text{加算の際に発生する誤差}| < \max\{|a|,|b|\}\times 2^{-23}$$

と見積もることが出来る。 ここで、

- (1) の順序で累算する場合は、どの加減算においても $0.5 \le \max\{|\text{被演算数}|, |\text{演算数}|\} \le 1$ であるから、(1) の累算で発生する誤差の上限は
$$|\text{(1) の累算で発生する誤差}| < 2^{-23}\times(100000-1) \approx 1.2\times10^{-2}$$
と見積もることが出来る。 プログラムの実行結果ではこの上限の $\frac{1}{100}$ 程度の誤差しか発生していないが、これは加算による誤差と減算による誤差が打ち消し合ったためと考えられる。
- (2) の順序で累算する場合は、各々の加減算において $\max\{|\text{被演算数}|, |\text{演算数}|\} \le$ 演算数 であるから、(2) の累算で発生する誤差の上限は次の様に見積もることが出来る。

$$\begin{aligned}
|\text{(2) の累算で発生する誤差}| &< \left(\frac{1}{99999}+\frac{1}{99998}+\cdots+\frac{1}{2}+\frac{1}{1}\right)\times2^{-23}\\
&< \left(\int_1^{99999}\frac{1}{x}dx+\frac{1}{1}\right)\times2^{-23}\\
&= (\log 99999+1)\times2^{-23}\\
&\approx 1.56\times10^{-6}
\end{aligned}$$

$\max\{|\text{被演算数}|, |\text{演算数}|\} \le$ 演算数 について：

◇ 減算の場合、非負の数に関しては 相乗平均≤相加平均 であるから、
$$\begin{aligned}
&\left|-\tfrac{1}{100000}+\tfrac{1}{99999}-\cdots-\tfrac{1}{k+2}+\tfrac{1}{k+1}\right|\\
&= \tfrac{1}{100000\times99999}+\cdots+\tfrac{1}{(k+2)\times(k+1)}\\
&\le \tfrac{1}{2}\left(\tfrac{1}{100000^2}+\tfrac{1}{99999^2}+\cdots+\tfrac{1}{(k+2)^2}+\tfrac{1}{(k+1)^2}\right)\\
&\le \tfrac{1}{2}\int_k^{100000}\tfrac{1}{x^2}dx\\
&\le \tfrac{1}{k}
\end{aligned}$$

◇ 加算の場合、
$$\begin{aligned}
&\left|-\tfrac{1}{100000}+\tfrac{1}{99999}-\cdots-\tfrac{1}{k+3}+\tfrac{1}{k+2}-\tfrac{1}{k+1}\right|\\
&= \left|\tfrac{1}{100000\times99999}+\cdots+\tfrac{1}{(k+3)\times(k+2)}-\tfrac{1}{k+1}\right|\\
&< \tfrac{1}{k+1}\quad\text{(減算の場合の結果より}\\
&\qquad \tfrac{1}{100000\times99999}+\cdots+\tfrac{1}{(k+3)\times(k+2)} < \tfrac{1}{k+1}\text{だから)}\\
&< \tfrac{1}{k}
\end{aligned}$$

□演習 7.16 (累算の順序) $\log_e 2 = \displaystyle\sum_{k=1}^{\infty}\frac{(-1)^{k+1}}{k} = 0.69314718\cdots$ の近似式

$$a = 1-\frac{1}{2}+\frac{1}{3}-\frac{1}{4}+\cdots+\frac{1}{99999}-\frac{1}{100000}$$

の値を次の 4 通りの順序で計算して、それらの結果を真値 $a = 0.693142180\cdots$ と比較してみよ。 また、それぞれの計算においてどの様な誤差／現象が発生するかを考えてみることによって、これらの計算順序のどれが良いか検討せよ。

(1) $1-\frac{1}{2}+\frac{1}{3}-\frac{1}{4}+\cdots+\frac{1}{99999}-\frac{1}{100000}$ (定義通りに累算)

(2) $\left(1-\frac{1}{2}\right)+\left(\frac{1}{3}-\frac{1}{4}\right)+\cdots+\left(\frac{1}{99999}-\frac{1}{100000}\right)$ (2 項ずつ組にして 大きい順に累算)

(3) $-\frac{1}{100000}+\frac{1}{99999}-\cdots-\frac{1}{4}+\frac{1}{3}-\frac{1}{2}+1$ (定義の逆順に累算)

(4) $\frac{1}{100000\times99999}+\cdots+\frac{1}{4\times3}+\frac{1}{2\times1}$ (式を変形して 小さい順に累算)

誤差の発生と対策 (まとめ)： 計算機を用いて数値計算を行う際、次の様な誤差／現象が起こります。[この内、計算機特有のものは①基数変換に伴う誤差だけであり、残りの 3 種は手計算の際も起こる。]

① **基数変換** (i.e.2 進 $\leftrightarrow$10 進変換) **に伴う丸め：**
例えば、10 進小数 0.1 は 2 進法では $0.0\dot{0}01\dot{1}$ という循環小数になる。それゆえ、各数値を 2 進有限固定長 (普通 32 ビット) で記憶する計算機としては、表し切れない下位の桁を**丸め** (四捨五入, 切り捨て, または切り上げ) ることになり、10 進小数 0.1 を正確に記憶することはできない。従って、計算機内部で 0.1×10.0 の計算をしても結果は 1 にはならない。

一方、10 進数 2^{-20} は計算機内部では実数データとして正確に記憶されるが、この値を 10 進表記 (i.e.2 進 $\rightarrow$10 進変換) すると $9.5367431640625\times10^{-7}$ ということになる。この数値は有限小数には違いないが、これを 10 進 7 桁の精度で出力すると 8 桁目以降は捨てられ誤差が発生する。

② **演算に伴う丸め：**
例えば、有効桁 3 桁同士の乗算 1.23×4.56 を行うと

$$1.23\times4.56 = 5.60\underline{88}$$

となり、10^{-3} の位以下が丸め (四捨五入, 切り捨て, または切り上げ) られる。この種の誤差に対処するには、**式の簡素化などにより演算回数をできるだけ少なくする**しかない。

③ **情報落ち (情報埋没)：**
これは②の誤差の一種である。絶対値の大きさが桁違いに違う 2 数を加減算すると小さい方の下位の桁が失われてしまう。例えば、次の加算では下線部が失われる。

$$\begin{array}{r} 1.234567 \\ +\ 0.04321098 \\ \hline 1.2777779\underline{8} \end{array}$$

数回の加減算では大した誤差は累積しないが、大量の実数値データを累算する場合は誤差が大きく累積することもある。この情報落ち誤差が大きく累積しない様にするためには、**多数の実数データの累算は絶対値の小さいものから順に行う**様に心掛ける。[実数データを累算する毎に誤差が少しずつ溜まってゆくので、次の④桁落ちほど気を付ける必要はない。]

④ **桁落ち：**
大きさのほぼ等しい 2 数を減算する時，あるいはそれと同等の加算をする時、有効桁が大きく失われてしまう。 例えば、次の通り。

$$\begin{array}{rl} 1.234567 & \cdots\ 7\text{桁の有効数字} \\ -\ 1.234566 & \cdots\ 7\text{桁の有効数字} \\ \hline 0.000001 & \cdots\ 1\text{桁の有効数字} \end{array}$$

実数計算においては精度が重要であるから、**最終結果に影響を及ぼす様な桁落ちは絶対に避けなければならない。** [厳密に言うと、桁落ちにおいては新たな (絶対) 誤差が発生する訳ではない。上位の桁が失われてしまうために、それまで累積していた誤差部分の (以後の計算における) 影響度／注目度が大きくなるのである。]

8　数値計算

- 方程式の解法 (Newton 法),
- 数値積分 (Simpson の公式)

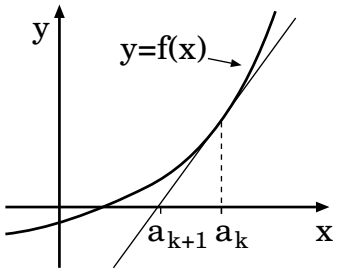
この節では、実数計算の代表例として数値計算問題を 2 つ取り上げる。まず最初に方程式の数値解を求める問題を考え、次に数値積分を行う問題を考える。

8.1　方程式の解法 — Newton-Raphson 法 —

例題 8.1 (方程式の数値解法；Newton-Raphson 法)　方程式 $f(x) = x - \cos x = 0$ の実根を数値的に求めて出力する C プログラムを作成せよ。

(考え方)　一般に、方程式 $f(x) = 0$ の実根を数値的に求めるための方法としては、**Newton-Raphson 法** (または **Newton 法**) , **二分法** (bisection method), **擬点法** (regula falsi, method of false position) 等が有名である。このうち、Newton-Raphson 法は、適当な近似解 $x = a_1$ から出発して、漸近式

$$a_{k+1} \quad = \quad a_k - \frac{f(a_k)}{f'(a_k)} \qquad (k = 1, 2, 3, ...)$$



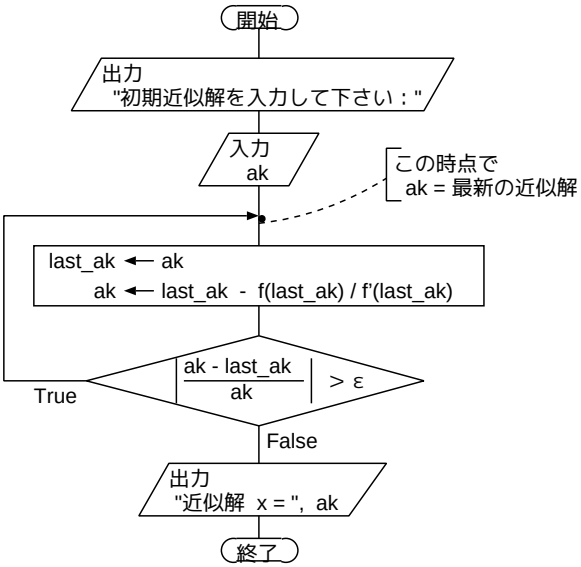
補足：
$y = f(x)$ のグラフを
点 $(a_k, f(a_k))$ における接線
$y - f(a_k) = f'(a_k)(x - a_k)$
で近似して、x 軸との交点を求めることに相当する。

により、次々とより良い近似解 $x = a_k (k = 1, 2, 3, ...)$ を求めていこうというものである。もちろん、$a_1, a_2, a_3, ...$ が十分に収束したと判断できる時点で、このアルゴリズムは終了させる。 具体的な終了条件としては、正数 ϵ を十分小さく選んで、

$$|a_{k+1} - a_k| \le \epsilon \quad \text{または} \quad \left| \frac{a_{k+1} - a_k}{a_{k+1}} \right| \le \epsilon$$

という形のものを考えれば良い。　[近似根の列 $a_1, a_2, a_3, ...$ は収束するとは限らないが、ほとんどの場合速く収束することが知られている。]

(プログラミング)　次々と近似根 a_k を更新してゆくだけなら、a_k が出来た時点でそれまでの $a_1, a_2, a_3, ..., a_{k-1}$ は不要であるので、$a_1, a_2, a_3, ...$ を保持するために常に最新の a_k を記憶する変数だけがあれば良い。しかし、$a_1, a_2, a_3, ..., a_k$ が収束したかどうかの判定を行うために、最も最近の 2 つの近似根 a_{k-1}, a_k を (終了判定の時に) 保持しておく必要がある。そこで、これらの連続した 2 つの近似根 a_{k-1}, a_k を保持するために、それぞれ `last_ak, ak` という名前の変数を用意すれば、行うべき処理は次の流れ図の様に書き表すことが出来る。



この処理を行うCプログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl newton-raphson.c Enter
     1 /* 方程式 f(x)=x-cos(x)=0 の実根を Newton-Raphson 法により */
     2 /* 数値的に求めて出力するCプログラム                       */

     3 #include <stdio.h>
     4 #include <math.h>

     5 #define  f(x)             ((x) - cos(x))
     6 #define  derivative_f(x)  (1.0 + sin(x))
     7 #define  EPSILON          (0.5e-15)

     8 int main(void)
     9 {
    10   double  ak, last_ak;

    11   printf("方程式 x-cos(x)=0 の初期近似解を入力して下さい:  x= ");
    12   scanf("%lf", &ak);

    13   do {
    14     last_ak = ak;
    15     ak = last_ak - f(last_ak)/derivative_f(last_ak);
    16   }while (ak==0.0 || fabs((ak-last_ak)/ak) > EPSILON);

    17   printf("最終の近似解は     x=%#21.15g.\n", ak);
    18   return 0;
    19 }
[motoki@x205a]$ gcc newton-raphson.c -lm Enter
```

```
[motoki@x205a]$ ./a.out Enter
方程式 x-cos(x)=0 の初期近似解を入力して下さい:  x= 1.0 Enter
最終の近似解は      x=     0.739085133215161.
[motoki@x205a]$
```

ここで、

- プログラムの5行目は**引数付きマクロ**を定義したプリプロセッサ指令である。これによって、以降 (15行目) に現れる `f(`x`)` という形の文字列は全てコンパイル前に (`$x$ - cos(`x`)`) という文字パターンに置き換えられることになる。
- プログラムの6行目は、$f(x)$ の導関数 $f'(x)$ を引数付きマクロとして定義したプリプロセッサ指令である。これによって、以降 (15行目) に現れる `derivative_f(`x`)` という形の文字列はコンパイル前に (`1 + sin(`x`)`) という文字パターンに置き換えられることになる。
- プログラムの16行目は収束判定を行なっている箇所である。流れ図を忠実にCのコードに直すと

  ```
  }while (fabs((ak-last_ak)/ak) > EPSILON);
  ```

  ということになるが、これだと偶然 `ak` の値が0になった時に `fabs((ak-last_ak)/ak)` の値が `Inf` または `Nan` になってしまい収束判定の条件が多少曖昧になってしまう。そこで、繰返しを続ける条件の前に「`ak==0.0 ||`」(「$a_k = 0$ または」) という文字列を挿入し、$a_k = 0$ の時は強制的に繰り返しを続ける様にした。($x = 0$ が求める根の時には困るが、その様な根は与えられた式からすぐに出てくるのでプログラムを使って割り出せなくても大きな問題にならない。)
- プログラムの7行目と16行目によって、

  $$a_k \neq 0 \ \text{ かつ } \ a_k - |a_k| \times 0.5 \times 10^{-15} \leq a_{k-1} \leq a_k + |a_k| \times 0.5 \times 10^{-15}$$

  の時に収束の判断をして繰返しを抜けることになる。これによって、得られている最後の2つの近似解 a_k と a_{k-1} が互いに、上位16桁目 (頭の0は数えない) 以下を丸めた時の丸め誤差の範囲内にあることが保証される。
- プログラム17行目の出力書式中の `%#21.15g` は、計算結果を有効桁15桁で表示するための指定である。`g` 変換では通常 (数値の大きさに影響を与えない) 末尾の `0` が省略されるが、ここでは `#` がフラグとして指定されているので末尾の `0` は省略されない。
- 繰り返しを続ける条件として16行目以外の記述を行った、「難あり」のプログラム例を幾つか次に例示する。

  16行目の代わりに `}while (fabs(f(ak)) > EPSILON);` とした場合:

  $\Longrightarrow$ 例えば $f(x) = 10^{-20}(x - \cos x)$ の時に十分な精度の解が得られない。

```
[motoki@x205a]$ nl newton-raphson_bad1.c
     1 /* 方程式 f(x)=1e-20*(x-cos(x))=0 の実根を Newton-  */
     2 /* Raphson 法により数値的に求めて出力する C プログラム */
     3 /* (繰り返しを続ける「難あり」の条件を用いた例 1)     */

     4 #include <stdio.h>
     5 #include <math.h>

     6 #define  f(x)             (1e-20*((x) - cos(x)))
     7 #define  derivative_f(x)  (1e-20*(1.0 + sin(x)))
     8 #define  EPSILON          (0.5e-15)
```

```
 9 int main(void)
10 {
11   double  ak, last_ak;

12   printf("方程式 x-cos(x)=0 の初期近似解を入力して下さい:  x= ");
13   scanf("%lf", &ak);

14   do {
15     last_ak = ak;
16     ak = last_ak - f(last_ak)/derivative_f(last_ak);
17   }while (fabs(f(ak)) > EPSILON);

18   printf("最終の近似解は     x=%#21.15g.\n", ak);
19   return 0;
20 }
[motoki@x205a]$ gcc newton-raphson_bad1.c -lm
[motoki@x205a]$ ./a.out
方程式 x-cos(x)=0 の初期近似解を入力して下さい:  x= 1.0
最終の近似解は     x=     0.750363867840244.
[motoki@x205a]$
```

16 行目の代わりに }while (((last_ak-ak)/ak) > EPSILON); とした場合：

$\Longrightarrow$ 例えば $(a_0 - a_1)/a_1 < 0$ の時には収束してなくても繰り返しを終了してしまう。結果として、十分な精度の解が得られないことが多い。

```
[motoki@x205a]$ nl newton-raphson_bad2.c
 1 /* 方程式 f(x)=x-cos(x)=0 の実根を Newton-Raphson 法により */
 2 /* 数値的に求めて出力する C プログラム                      */
 3 /* (繰り返しを続ける「難あり」の条件を用いた例 2)          */

 4 #include <stdio.h>
 5 #include <math.h>

 6 #define  f(x)             ((x) - cos(x))
 7 #define  derivative_f(x)  (1.0 + sin(x))
 8 #define  EPSILON          (0.5e-15)

 9 int main(void)
10 {
11   double  ak, last_ak;

12   printf("方程式 x-cos(x)=0 の初期近似解を入力して下さい:  x= ");
13   scanf("%lf", &ak);

14   do {
15     last_ak = ak;
16     ak = last_ak - f(last_ak)/derivative_f(last_ak);
17   }while (((last_ak-ak)/ak) > EPSILON);

18   printf("最終の近似解は     x=%#21.15g.\n", ak);
19   return 0;
20 }
[motoki@x205a]$ gcc newton-raphson_bad2.c -lm
```

```
[motoki@x205a]$ ./a.out
方程式 x-cos(x)=0 の初期近似解を入力して下さい:  x= -1.0
最終の近似解は     x=      8.71621695877957.
[motoki@x205a]$
```

16 行目の代わりに }while (fabs(ak-last_ak) > EPSILON); とした場合：

$\Longrightarrow$ 例えば $f(x) = (10^{20}x) - \cos(10^{20}x)$ の時に十分な精度の解が得られない。また、$f(x) = (10^{-20}x) - \cos(10^{-20}x)$ の時に、$a_{k+1} \neq a_k, a_{k+2} = a_k, a_{k+3} = a_{k+1}, \ldots$ という風に振動し繰り返しを終了しない可能性もある。

```
[motoki@x205a]$ nl newton-raphson_bad3.c
     1 /* 方程式 f(x)=(1e20*x)-cos(1e20*x)=0 の実根を Newton- */
     2 /* Raphson 法により数値的に求めて出力する C プログラム     */
     3 /* (繰り返しを続ける「難あり」の条件を用いた例 3)        */

     4 #include <stdio.h>
     5 #include <math.h>

     6 #define  f(x)             ((1e20*x) - cos(1e20*x))
     7 #define  derivative_f(x)  (1e20 + 1e20*sin(1e20*x))
     8 #define  EPSILON          (0.5e-15)

     9 int main(void)
    10 {
    11   double  ak, last_ak;

    12   printf("方程式 x-cos(x)=0 の初期近似解を入力して下さい:  x= ");
    13   scanf("%lf", &ak);

    14   do {
    15     last_ak = ak;
    16     ak = last_ak - f(last_ak)/derivative_f(last_ak);
    17   }while (fabs(ak-last_ak) > EPSILON);

    18   printf("最終の近似解は     x=%#21.15g.\n", ak);
    19   return 0;
    20 }
[motoki@x205a]$ gcc newton-raphson_bad3.c -lm
[motoki@x205a]$ ./a.out
方程式 x-cos(x)=0 の初期近似解を入力して下さい:  x= 1.0
最終の近似解は     x=-1.52870240130842e-16.
[motoki@x205a]$
```

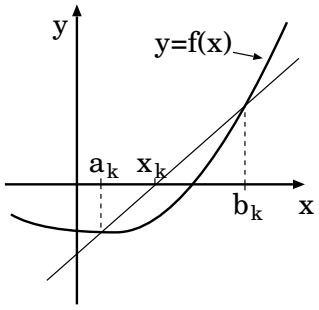
□演習 8.2 (擬点法) 擬点法は

$$f(a_1)f(b_1) < 0$$

となるような近似解の組 $x = a_1, x = b_1$ から出発して、漸化式

$$x_k = \frac{b_k f(a_k) - a_k f(b_k)}{f(a_k) - f(b_k)} \qquad (k = 1, 2, 3, ...)$$

$$(a_{k+1}, b_{k+1}) = \begin{cases} (a_k, x_k) & \text{if } f(a_k)f(x_k) < 0 \\ (x_k, b_k) & \text{otherwise} \end{cases}$$



により、次々とより良い近似解 $x = x_k (k = 1, 2, 3, ...)$ を求めていこうというものである。$f(x_k) = 0$ となった時点、あるいは $x_1, x_2, x_3, ...$ が十分に収束したと判断できる時点で、この計算は終了する。　擬点法を用いて方程式 $f(x) = x - \cos x = 0$ の近似解を小数点以下 15 桁まで求める C プログラムを作成せよ。但し、ここでは $a_1 = 0, b_1 = 1$ とせよ。

8.2 数値積分 —Simpson の公式—

例題 8.3 (数値積分；Simpson の公式)　定積分 $\displaystyle\int_0^1 \frac{4}{1+x^2} dx$ の値を数値的に求めて出力する C プログラムを作成せよ。

(考え方)　一般に、定積分 $\displaystyle\int_a^b f(x)dx$ の値を数値的に求めるための方法としては、ニュートン・コーツ (Newton-Cotes) の積分公式、エルミット (Hermite) の内挿公式、ガウス (Gauss) 形の積分公式 等がある。 このうち、**シンプソンの公式**は Newton-Cotes の積分公式の一種で、解析学の教科書にも載るほど有名なものである。 シンプソンの公式によれば、定積分値は

$$\begin{aligned} \int_a^b f(x)dx \approx \frac{h}{3}\Big[& f(a_0) + f(a_{2n}) \\ & +4(\ f(a_1) + f(a_3) + \cdots + f(a_{2n-3}) + f(a_{2n-1})\ ) \\ & +2(\ f(a_2) + f(a_4) + \cdots + f(a_{2n-2})\ ) \Big] \end{aligned}$$

但し、$h = \frac{b-a}{2n}$

$a_i = a + ih$

と近似でき、この近似の際の誤差は

$$|\text{誤差}| \le \frac{(b-a)^5}{2880n^4} \max\left\{ |f^{(4)}(\xi)| \;\middle|\; a \le \xi \le b \right\}$$

と見積もられる。

補足：

- n は **1** 以上の整数で、十分大きく選ぶ。

  この n で指定された個数に積分区間 $[a,b]$ は等分割され、各々の小区間において $f(x)$ が 2 次多項式で近似され定積分の近似値が求められることになる。

- 特に $f(x)=\dfrac{4}{1+x^2}$ の場合は、

$$\begin{aligned} f'(x) &= \frac{-8x}{(1+x^2)^2} \\ f''(x) &= \frac{8(3x^2-1)}{(1+x^2)^3} \\ f'''(x) &= \frac{96x(1-x^2)}{(1+x^2)^4} \\ f^{(4)}(x) &= \frac{96(5x^4-10x^2+1)}{(1+x^2)^5} \\ f^{(5)}(x) &= \frac{-960(x^2-3)(3x^2-1)}{(1+x^2)^6} \end{aligned}$$

  であるから、

$$\begin{aligned} &\max\left\{|f^{(4)}(\xi)| \;\middle|\; 0\leq\xi\leq 1\right\} \\ &= \max\left\{|f^{(4)}(\xi)| \;\middle|\; \xi=0,\ \xi=1 \text{ または } (f^{(5)}(\xi)=0 \text{ かつ } 0\leq\xi\leq 1)\right\} \\ &= \max\left\{|f^{(4)}(0)|,\ |f^{(4)}(1)|,\ \left|f^{(4)}(\frac{1}{\sqrt{3}})\right|\right\} \\ &= \max\left\{|96|,\ |-12|,\ \left|-\frac{81}{2}\right|\right\} \\ &= 96 \end{aligned}$$

  となる。 それゆえ、$\displaystyle\int_0^1 \frac{4}{1+x^2}dx$ の近似値を求めるのにシンプソンの公式を使った場合、小区間の個数が $n=10$ では

$$\begin{aligned} |\text{誤差}| &\leq \frac{(1-0)^5}{2880\times 10^4}\times 96 + |\text{計算に伴う誤差}| \\ &\approx 3.3\times 10^{-6} + |\text{計算に伴う誤差}| \end{aligned}$$

  となり、また $n=1000$ では

$$\begin{aligned} |\text{誤差}| &\leq \frac{(1-0)^5}{2880\times 1000^4}\times 96 + |\text{計算に伴う誤差}| \\ &\approx 3.3\times 10^{-14} + |\text{計算に伴う誤差}| \end{aligned}$$

  となる。

- 真値は $\displaystyle\int_0^1 \frac{4}{1+x^2}dx = \pi = 3.14159265358979323846264338327 9\cdots$

(プログラミング) 与えられた式に従って計算するだけである。累算する箇所が 2 つあるが、そのうち $4(\ f(a_1)+f(a_3)+\cdots+f(a_{2n-3})+f(a_{2n-1})\ )$ の累算値を保持するために `sum4` という名前の `double` 型変数を、$2(\ f(a_2)+f(a_4)+\cdots+f(a_{2n-2})\ )$ の累算値を保持するために `sum2` という名前の `double` 型変数を用意してプログラムを構成した。$n=1000$ としてこの計算を行う C プログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl numerical-integral-by-simpson.c Enter
     1 /* 定積分 \int_0^1 \frac{4}{1+x^2} dx の値を Simpson の公式 */
     2 /* により数値的に求めて出力する C プログラム                  */

     3 #include <stdio.h>

     4 #define  N     (1000)
     5 #define  A     (0.0)
     6 #define  B     (1.0)
     7 #define  f(x)  (4.0/(1.0+(x)*(x)))

     8 int main(void)
     9 {
    10   double ans, sum4, sum2;
    11   int    i;

    12   sum4 = 0.0;
    13   for (i=2*N-1; i>=1; i-=2)
    14     sum4 += f(A+(B-A)*((double)i)/(2.0*(double)N));

    15   sum2 = 0.0;
    16   for (i=2*N-2; i>=2; i-=2)
    17     sum2 += f(A+(B-A)*((double)i)/(2.0*(double)N));

    18   ans = (f(A)+f(B)+4.0*sum4+2.0*sum2)*(B-A)/(6.0*(double)N);
    19   printf("定積分値 (近似) は %22.16g\n", ans);
    20   return 0;
    21 }
[motoki@x205a]$ gcc numerical-integral-by-simpson.c Enter
[motoki@x205a]$ ./a.out Enter
定積分値 (近似) は      3.141592653589791
[motoki@x205a]$                          ↖ 下線部は真値と一致する箇所
```

ここで、

- プログラムの 7 行目 は引数付きマクロを定義したプリプロセッサ指令である。これによって、以降 (14 行目,17〜18 行目) に現れる `f(`x`)` という形の文字列は全てコンパイル前に `(4.0/(1.0+(`x`)*(`x`)))` という文字パターンに置き換えられることになる。
- プログラムの 13 行目, 16 行目 は、それぞれ

  `for (i=1; i<=2*N-1; i+=2)`, `for (i=2; i<=2*N-2; i+=2)`

  と書いた方が分かり易いかも知れない。しかし、こう書くと次の繰り返しを進めるかどうかの判定の度に `2*N-1` もしくは `2*N-2` の計算を行わなければならないので、与えられた式とは逆の順番に累算することにした。

□**演習 8.4 (Simpson の公式)** Simpson の公式を使えば、$n = 10, 100$ の時に実際にどの

程度の精度で $\displaystyle\int_0^1 \frac{4}{1+x^2}dx$ の値が求まるか調べよ。

□**演習 8.5** 次の定積分の値を数値的に求めて出力する C プログラムを作成せよ。 [いずれの定積分値も π になることが、解析学の演習書の中に載っている。]

(1) $\displaystyle\int_0^2 \sqrt{4-x^2}dx$

(2) $\displaystyle\int_0^{0.5} (12\sqrt{1-x^2}-\sqrt{27})dx$

□**演習 8.6 (台形公式)** Newton-Cotes の積分公式の中で、最も単純な (, そして最も近似精度の悪い) ものは**台形公式**と呼ばれている。 台形公式によれば、一般に、定積分値は

$$\begin{aligned}\int_a^b f(x)dx \quad \approx \quad & \frac{h}{2}\Big[\ f(a_0)+f(a_n) \\ & \qquad +2(\ f(a_1)+f(a_2)+\cdots+f(a_{n-2})+f(a_{n-1})\ )\ \Big]\end{aligned}$$

但し、$h=\frac{b-a}{n}$

$a_i = a+ih$

と近似でき、この近似の際の誤差は

$$|\text{誤差}| \le \frac{(b-a)^5}{12n^3}\max\left\{|f^{(2)}(\xi)| \;\middle|\; a\le\xi\le b\right\}$$

と見積もられる。 この台形公式を用いて $\displaystyle\int_0^1 \frac{4}{1+x^2}dx$ の値を数値的に求めて出力する C プログラムを作成せよ。

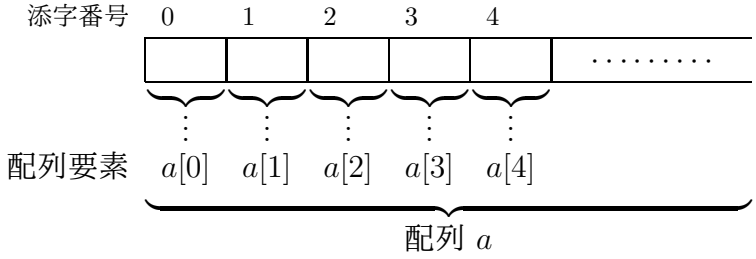
9 配列

- 1 次元配列を用いた計算,
- 整列化,
- 2 次元配列,
- 付録 配列のまとめ

複数のものに添字番号付きの名前を付けると、それらを系統的に表せることがある。例えば数学で、一般的な数列を $a_1, a_2, a_3, \ldots$ という風に表したりする。この節では、この様な「添字番号付きの名前」を使って大量のデータ領域のそれぞれに系統的な名前を付け、それらの領域を規則的に扱う方法を紹介する。

9.1 一次元配列を用いた計算

同じデータ型の領域を連続的に並べたものを一般に**配列** (array) と呼び、その中の個々のデータ領域を**配列要素** (array element) と呼ぶ。配列を使えば大量の同種のデータを規則的に並べて格納し、その中の各々のデータに対して同じ処理を繰り返すことが簡単にできるので、大抵のプログラミング言語で配列が使えるようになっている。特に C 言語においては、配列の先頭に位置する要素から順に $0, 1, 2, 3, ...$ という添字番号が各々の配列要素に割り振られ、配列 a の中の添字番号が k の配列要素は $a[k]$ と表される。



例題 9.1 (平均と分散) 50 個の実数データ $x_0, x_1, x_2, \ldots, x_{49}$ を読み込み、それらの平均 μ と分散 V を定義式

$$\mu = \frac{1}{50}(x_0 + x_1 + x_2 + \cdots + x_{49})$$
$$V = \frac{1}{50}\sum_{i=0}^{49}(x_i - \mu)^2$$

に従って求めて出力する C プログラムを作成せよ。

(考え方) 平均 μ を計算するだけなら、例題 6.19 で行ったように、読み込んだデータを保持する変数を 1 個だけ用意し、

- そこへのデータ読み込みと、
- 読み込んだデータを別の累算値を保持する変数に加える作業

を交互に繰り返せばよい。しかし、指定された式に従って分散 V を計算するなら、V の計算には平均 μ の計算結果が必要になるので、分散 V の計算で指定された累算をデータの読み込みと並行して行うわけにはいかない。従って、読み込んだ 50 個の実数データ $x_0, x_1, x_2, \ldots, x_{49}$ は全て保持しておく必要がある。これらのデータ保持に配列を用いる。

(プログラミング) 50 個の実数データ $x_0, x_1, x_2, \ldots, x_{49}$ を保持するために x という名前の double 型配列を用意し、$x_0 + x_1 + x_2 + \cdots + x_{49}$, $\sum_{i=0}^{49}(x_i - \mu)^2$ の累算をそれぞれ ave, var という名前の double 型変数上に行うことにしてプログラムを構成した。この計算を行う C プログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl average-variance.c Enter
     1  /* 50 個の実数データ x0, x1, x2, ... , x49 の平均 mu と     */
     2  /* 分散 V を定義式                                          */
     3  /*   mu = (x0+x1+x2+ ...  + x49)/50                        */
     4  /*    V = {(x0-mu)^2 + (x1-mu)^2 + ... + (x49-mu)^2} / 50  */
     5  /* に従って求め、それらの値を出力する C プログラム          */

     6  #include <stdio.h>

     7  int main(void)
     8  {
     9    int     i;
    10    double  x[50], ave, var;

    11    ave = 0.0;
    12    for (i=0; i<50; ++i) {
    13      scanf("%lf", &x[i]);
    14      ave += x[i];
    15    }
    16    ave /= 50.0;

    17    var = 0.0;
    18    for (i=0; i<50; ++i)
    19      var += (x[i]-ave)*(x[i]-ave);
    20    var /= 50.0;

    21    printf("\nInput data:\n");
    22    for (i=0; i<50; i+=5)
    23      printf("%14.5e%14.5e%14.5e%14.5e%14.5e\n",
    24             x[i], x[i+1], x[i+2], x[i+3], x[i+4]);
    25    printf("\nAverage  = %14.6g\n"
    26           "Variance = %14.6g\n", ave, var);
    27    return 0;
```

```
    28  }
[motoki@x205a]$ cat average-variance.data Enter
1.0000  1.0001  1.0002  1.0003  1.0004
1.0005  1.0006  1.0007  1.0008  1.0009
1.0010  1.0011  1.0012  1.0013  1.0014
1.0015  1.0016  1.0017  1.0018  1.0019
1.0020  1.0021  1.0022  1.0023  1.0024
1.0025  1.0026  1.0027  1.0028  1.0029
1.0030  1.0031  1.0032  1.0033  1.0034
1.0035  1.0036  1.0037  1.0038  1.0039
1.0040  1.0041  1.0042  1.0043  1.0044
1.0045  1.0046  1.0047  1.0048  1.0049
[motoki@x205a]$ gcc average-variance.c Enter
[motoki@x205a]$ ./a.out < average-variance.data Enter

Input data:
   1.00000e+00   1.00010e+00   1.00020e+00   1.00030e+00   1.00040e+00
   1.00050e+00   1.00060e+00   1.00070e+00   1.00080e+00   1.00090e+00
   1.00100e+00   1.00110e+00   1.00120e+00   1.00130e+00   1.00140e+00
   1.00150e+00   1.00160e+00   1.00170e+00   1.00180e+00   1.00190e+00
   1.00200e+00   1.00210e+00   1.00220e+00   1.00230e+00   1.00240e+00
   1.00250e+00   1.00260e+00   1.00270e+00   1.00280e+00   1.00290e+00
   1.00300e+00   1.00310e+00   1.00320e+00   1.00330e+00   1.00340e+00
   1.00350e+00   1.00360e+00   1.00370e+00   1.00380e+00   1.00390e+00
   1.00400e+00   1.00410e+00   1.00420e+00   1.00430e+00   1.00440e+00
   1.00450e+00   1.00460e+00   1.00470e+00   1.00480e+00   1.00490e+00

Average  =        1.00245
Variance =     2.0825e-06
[motoki@x205a]$
```

ここで、

- プログラムの 10 行目 で大きさ 50 の double 型配列 x (と double 型変数 ave, var) が確保されている。 C 言語では配列の添字は必ず 0 から始まるので、これで配列要素

  x[0], x[1], x[2] , . . . , x[49]

  が double 型変数と同じように使えることになる。 double x[50]; と宣言していますが、x[50] という配列要素が使えるわけではないことに注意して下さい。
- プログラム 25 行目 の出力書式中の %14.6g は、出力フィールドの大きさを 14 桁、(最大) 有効桁数を 6 桁として、f 変換と e 変換の短くなる方で出力することを表す。

> 補足：
> 実際の出力を見ると 2.0825e-06 と有効桁が 5 桁になっているが、これは g 変換では最後に続く 0 および小数点は印字されないためである。

- プログラム実行のために `./a.out < average-variance.data` とコマンド入力しているが、この中の “<” は UNIX/Linux に備わっている **標準入力のリダイレクション** の機能を使っている。 この “<” 以降の指示によって、キーボードからの入力に代わって、ファイル `average-variance.data` 内のデータが順に標準入力のデータ列として扱われるようになる。

□**演習 9.2 (配列の有用性)** 例題 9.1 において、もし読み込んだ 50 個のデータを保持するのに配列ではなく 50 個の変数 `x0`，`x1`，`x2`，`x3`，...，`x49` を用意するとしたら、どの様なプログラムができるか考えよ。

相当大きくて退屈なプログラムができるでしょう。

□**演習 9.3 (平均と分散)** n 個のデータ $x_0, x_1, x_2, \ldots, x_{n-1}$ の平均 μ と分散 V は数学的には

$$\mu = \frac{1}{n}\sum_{i=0}^{n-1} x_i$$

$$V = \frac{1}{n}\sum_{i=0}^{n-1} x_i^2 - \mu^2$$

と計算できる。

(1) これらの式に従って平均と分散を計算する C プログラムを作成せよ。

(2) これらの式に従って平均と分散を計算すると例題 9.1 で挙げたものより単純なプログラムが得られる。しかし、分散 V の値に大きな誤差を引き起こす可能性があるので、この式に基づいてプログラムを作るのは一般には好ましくない。 どんな時に V の値に大きな誤差が生じるか考えよ。

□**演習 9.4 (多項式の計算)** 多項式 $f(x) = c_n x^n + c_{n-1}x^{n-1} + \cdots + c_2 x^2 + c_1 x^1 + c_0 x^0$ の次数 n と多項式の係数 $c_n, c_{n-1}, ..., c_2, c_1, c_0$ を順に読み込み、変数 $x = 0.0, 0.1, 0.2, 0.3, ..., 0.9, 1.0$ に対する多項式 $f(x)$ の値を計算して見易い形に出力する C プログラムを作成せよ。

Hint：
$f(x) = ((\cdots((c_n x + c_{n-1})x + c_{n-2})\cdots)x + c_1)x + c_0$
と計算すると単純で計算効率の良いプログラムができる。

□**演習 9.5 (1000 以下の素数)** 1000 以下の素数を全て出力する C プログラムを作成せよ。但し、出力は 1 行に 10 個ずつとする。

9.2 整列化

コンピュータ内に蓄えられた (大量の) データをある指定された順序に並べ直す作業は **整列化** (sorting) と呼ばれ、そのアルゴリズムは色々と考え出されている。

例えば、
選択整列法, 挿入整列法, バブル整列法, ヒープ整列法, クイック整列法, シェル整列法, ... 等がある。 整列化の問題はアルゴリズムを学ぶ際の基本的な問題として扱われることが多い。

特に整列化が配列と密接な関係がある訳ではないが、配列に蓄えられたデータの処理をする代表的な問題として、次に整列化を考えよう。

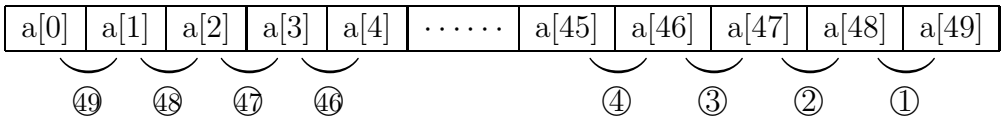
例題 9.6 (バブル整列法) 50 個の整数データを読み込み、それらを小さい順に出力する C プログラムを作成せよ。

(考え方) 50 個の整数データは最初に全て配列に読み込む。その後の整列化については色々な考えで処理を進めることができるが、ここでは、実用的ではないが最も簡単な部類に属する**バブル整列法** (bubble sort) を紹介しよう。読み込んだデータは a[0]〜a[49] に格納されているものとする。

バブル整列法では、大小順が逆になっている隣り同士の要素を交換する、という操作を繰り返す。 まず第 1 段階 として、

a[49] と a[48], a[48] と a[47], a[47] と a[46], a[46] と a[45], ...
....., a[4] と a[3], a[3] と a[2], a[2] と a[1], a[1] と a[0]

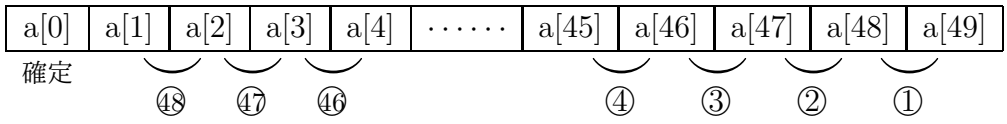
という組に対して、順に「逆順になっていれば交換」という操作を行う。



これで、あたかも (軽い) 泡が水面に出て来るように、全体の中で最小の要素が一つずつ配列の先頭方向に移動し、最後に a[0] に最小要素が入る。あとは a[1]〜a[49] を小さい順に並べ直せば良いので、次に 第 2 段階 として、

a[49] と a[48], a[48] と a[47], a[47] と a[46], a[46] と a[45], ...
....., a[4] と a[3], a[3] と a[2], a[2] と a[1]

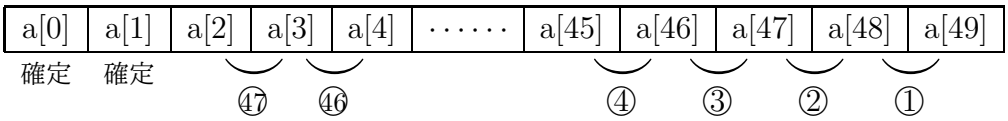
という組に対して、順に「逆順になっていれば交換」という操作を行う。



これで a[1] に 2 番目に小さな要素が入るので、第 3 段階 としては、

a[49] と a[48], a[48] と a[47], a[47] と a[46], a[46] と a[45], ...
....., a[4] と a[3], a[3] と a[2]

という組に対して、順に「逆順になっていれば交換」という操作を行う。



これで a[2] に 3 番目に小さな要素が入り、a[0]〜a[2] の部分は最終的に確定した要素が入ったことになる。以下、a[0]〜a[49] の部分に最終的に確定した要素が入るまで、この

手順を続けるだけである。

第 4 段階は分かりますよね。
第 4 段階では a[49] と a[48], a[48] と a[47], a[47] と a[46], a[46] と a[45], ..., a[4] と a[3] という順に「逆順になっていれば交換」という操作を行う。

(プログラミング) 50 個の整数データを読み込んで保持するために a という名前の int 型配列を用意する。データ入力後、①入力データの表示、②バブル整列法による配列内のデータの並べ替え、③整列後のデータの表示、を順に行う C プログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl bubblesort.c  Enter
     1 /* 50 個の整数データを読み込み、それらを小さい順に出力する */
     2 /* C プログラム (バブル整列法)                              */

     3 #include <stdio.h>
     4 #define  SIZE  50

     5 int main(void)
     6 {
     7   int a[SIZE], temp, k, i;

     8   for (k=0; k<SIZE; k++)
     9     scanf("%d", &a[k]);

    10   printf("整列前 : ");
    11   for (k=0; k<SIZE; k++) {
    12     if (k%5 == 0)
    13       printf("\n");
    14     printf("  %11d", a[k]);
    15   }
    16   printf("\n");

    17   for (k=0; k<SIZE-1; k++) {
    18     for (i=SIZE-1; i > k; i--)
    19       if (a[i-1] > a[i]) {      /* a[i-1] と a[i]    */
    20         temp   = a[i-1];        /* の大小を調べて、  */
    21         a[i-1] = a[i];          /* 逆順なら交換する。*/
    22         a[i]   = temp;
    23       }
    24   }

    25   printf("整列後 : ");
    26   for (k=0; k<SIZE; k++) {
    27     if (k%5 == 0)
    28       printf("\n");
```

```
    29     printf("  %11d", a[k]);
    30   }
    31   printf("\n");
    32   return 0;
    33 }
[motoki@x205a]$ cat bubblesort.data Enter
-1234   1000   3456  -7890    11   0  3333  6666  9876   4860
    1      2      4      9     8   7     6     5     0      3
  -45    -95    333  11111  7890  34  5555   234   784  -7420
 5893  -3099  77777  88888   452  99   470  -999  -672    -13
-2222  12345  -6789  -9876    -5  -7  -400  5505  1980  12800
[motoki@x205a]$ gcc bubblesort.c Enter
[motoki@x205a]$ ./a.out < bubblesort.data Enter
整列前：
        -1234         1000         3456        -7890           11
            0         3333         6666         9876         4860
            1            2            4            9            8
            7            6            5            0            3
          -45          -95          333        11111         7890
           34         5555          234          784        -7420
         5893        -3099        77777        88888          452
           99          470         -999         -672          -13
        -2222        12345        -6789        -9876           -5
           -7         -400         5505         1980        12800
整列後：
        -9876        -7890        -7420        -6789        -3099
        -2222        -1234         -999         -672         -400
          -95          -45          -13           -7           -5
            0            0            1            2            3
            4            5            6            7            8
            9           11           34           99          234
          333          452          470          784         1000
         1980         3333         3456         4860         5505
         5555         5893         6666         7890         9876
        11111        12345        12800        77777        88888
[motoki@x205a]$
```

ここで、

- プログラムの20〜22行目で逆順になっているデータを交換している。配列要素 a[i-1] と a[i] を交換するのに、

```
a[i-1] = a[i];
a[i]   = a[i-1];
```

としたのでは、元々 a[i-1] に保持されていたデータが失われてしまうので、この20〜22行目の様に a[i-1], a[i] とは別の変数 (この例では temp) が必要になる。

□**演習 9.7 (整列化)** 50 個の整数データを読み込み、それらを小さい順に出力する C プログラムを作成せよ。 但し、ここでは、上の例題 9.6 で紹介したバブル整列法を使うのではなく、自分の頭だけで整列化を行うアルゴリズムを考え出してみて下さい。

9.3 2 次元配列

C 言語においては添字番号の個数が複数の配列も用いることができる。同じデータ型の領域を 2 次元格子状に並べたものを**2 次元配列**、3 次元格子状に並べたものを**3 次元配列**と呼び、次元が 2 以上の配列を総称して**多次元配列**と呼ぶ。どの次元も $0, 1, 2, 3, ...$ という添字番号が付けられ、一般に n 次元配列 a の中の添字番号が $i_1, i_2, i_3, ..., i_n$ の配列要素は $a[i_1][i_2][i_3]\cdots[i_n]$ と表される。

添字番号	0	1	2		k_2-1
0	a[0][0]	a[0][1]	a[0][2]	·····	a[0][k_2-1]
1	a[1][0]	a[1][1]	a[1][2]	·····	a[1][k_2-1]
2	a[2][0]	a[2][1]	a[2][2]	·····	a[2][k_2-1]
	⋮	⋮	⋮		⋮
k_1-1	a[k_1-1][0]	a[k_1-1][1]	a[k_1-1][2]	·····	a[k_1-1][k_2-1]

補足：
C 言語においては、1 次元配列を 1 列に並べたものが 2 次元配列、2 次元配列を 1 列に並べたものが 3 次元配列、..... となっており、内部では多次元配列は 1 次元配列を入れ子にしたものとして処理される。 例えば、上に図示された 2 次元配列においては、

- a[0][0], a[0][1], ..., a[0][k_2-1] の部分が a[0] という 1 次元配列、
- a[1][0], a[1][1], ..., a[1][k_2-1] の部分が a[1] という 1 次元配列、

.....

であり、同じ大きさの 1 次元配列を並べた a[0], a[1], a[2], ..., a[k_1-1] の部分 (全体) が a という 2 次元配列になる。

例題 9.8 (行列の積) 3×3 実数値行列 $A = [a_{ij}]$, $B = [b_{ij}]$ の要素を読み込み、行列の積 AB を計算してその要素を 2 次元状に見易く出力する C プログラムを作成せよ。

(考え方) 2 つの行列 A, B とそれらの積 AB の結果は、各々3×3 の 2 次元配列に保持すれば良い。 積 AB も 3×3 行列で、その i 行,j 列 要素は

$$AB \text{ の } (i,j) \text{ 要素} = \sum_{k=1}^{3} a_{ik}b_{kj} = a_{i1}b_{1j} + a_{i2}b_{2j} + a_{i3}b_{3j}$$

と定められているので、積 AB の各要素をこの式の通りに計算して、結果を見易い形に出力するだけである。

(プログラミング) 3×3 実数値行列 A, B の各要素 a_{ij}, b_{ij} を保持するためにそれぞれ `a`, `b` という名前の 2 次元 `double` 型配列を用意し、積 AB の各要素を計算して `productAB` という名前の 2 次元 `double` 型配列に記録することにしてプログラムを構成した。この C

プログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl matrix-multiplication.c Enter
     1 /* 3×3実数値行列 A=[a_ij], B=[b_ij] の要素を読み込み、 */
     2 /* 行列の積 AB を計算してその要素を2次元状に見易く出力する  */
     3 /* Cプログラム                                              */

     4 #include <stdio.h>

     5 #define  N  (3)

     6 int main(void)
     7 {
     8   double a[N][N], b[N][N], productAB[N][N];
     9   int  i, j, k;

    10   /* 行列 A,B の各要素を入力する */
    11   for (i=0; i<N; i++) {
    12     printf("行列 A の %d 行目の要素を順に入力して下さい:  ", i);
    13     for (j=0; j<N; j++)
    14       scanf("%lf", &a[i][j]);
    15   }
    16   printf("\n");
    17   for (i=0; i<N; i++) {
    18     printf("行列 B の %d 行目の要素を順に入力して下さい:  ", i);
    19     for (j=0; j<N; j++)
    20       scanf("%lf", &b[i][j]);
    21   }

    22   /* 行列の積 AB の各要素を計算して配列 productAB に記録する */
    23   for (i=0; i<N; i++) {
    24     for (j=0; j<N; j++) {
    25       productAB[i][j] = 0.0;
    26       for (k=0; k<N; k++)
    27         productAB[i][j] += a[i][k]*b[k][j];
    28     }
    29   }

    30   /* 行列の積 AB の結果を表示する */
    31   printf("\n行列の積 AB の結果:\n");
    32   for (i=0; i<N; i++) {
    33     printf("   ");
    34     for (j=0; j<N; j++)
```

```
35        printf("  %12.5g", productAB[i][j]);
36      printf("\n");
37    }
38    return 0;
39 }
[motoki@x205a]$ gcc matrix-multiplication.c Enter
[motoki@x205a]$ ./a.out Enter
行列 A の 0 行目の要素を順に入力して下さい:  1  2  3 Enter
行列 A の 1 行目の要素を順に入力して下さい:  4  5  6 Enter
行列 A の 2 行目の要素を順に入力して下さい:  7  8  9 Enter

行列 B の 0 行目の要素を順に入力して下さい:  -1  1  1  Enter
行列 B の 1 行目の要素を順に入力して下さい:   1 -1  1 Enter
行列 B の 2 行目の要素を順に入力して下さい:   1  1 -1 Enter

行列の積 AB の結果:
                4             2             0
                7             5             3
               10             8             6
[motoki@x205a]$
```

ここで、

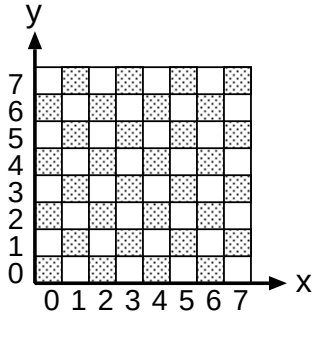
- プログラムの 5 行目 では、このプログラムのパラメータである 3 をマクロとして定義している。
  $\Longrightarrow$ 6 行目以降では一般に N×N 行列の積を計算するコードが示される。
- プログラムの 8 行目 で大きさ N×N の 2 次元 double 型配列 a, b, productAB が確保されている。
- 例えばプログラムの 14 行目 の a[i][j] は配列要素で、ここには行列 A の i 行,j 列要素を保持させる。

□演習 9.9 (行列の和) 3×3 実数値行列 $A = [a_{ij}]$, $B = [b_{ij}]$ の要素を読み込み、行列の和 $A + B$ を計算してその要素を 2 次元状に見易く出力する C プログラムを作成せよ。

□演習 9.10 (行列のスカラー倍) 3×3 実数値行列 $A = [a_{ij}]$ の要素と実数 r を読み込み、行列のスカラー倍 rA を計算してその要素を 2 次元状に見易く出力する C プログラムを作成せよ。

□演習 9.11 (Queen の勢力範囲) チェス盤の状況は 8×8 の文字パターンで表すことができる。 Queen の駒の位置を表す 2 つの非負整数 x, y (右図参照) を読み込んで、

- Queen の居る場所は文字 Q を、



- Queen の勢力範囲、すなわち
  Queen と同じ行の場所 $(0,y),(1,y),...,(7,y)$,
  Queen と同じ列の場所 $(x,0),(x,1),...,(x,7)$,
  右上がりの対角線上の場所
  $.....,(x-2,y-2),(x-1,y-1),\quad (x+1,y+1),(x+2,y+2),.....,$
  右下がりの対角線上の場所
  $.....,(x-2,y+2),(x-1,y+1),\quad (x+1,y-1),(x+2,y-2),.....,$
  には星印 * を、
- その他の場所にはハイフン - を

入れた文字パターンを出力するプログラムを作成せよ。 このプログラムは、例えば $x=2, y=3$ に対しては次の様な文字パターンを出力することになる。

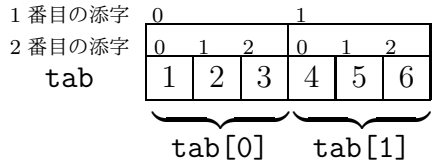
```
--*---*-
--*--*--
*-*-*---
-***----
**Q*****
-***----
*-*-*---
--*--*--
```

9.4 付録 配列のまとめ

配列について：

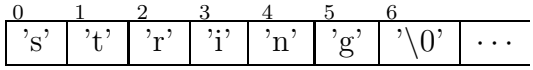
- 配列名が a、大きさが $k_1 \times k_2 \times \cdots \times k_n$ の配列の宣言／領域確保は次の様に行う。
  データ型 a[k_1][k_2] $\cdots$ [k_n]
- 宣言時の配列の初期化は例えば次の様に行う。

```
int num[12]={31,28,31,30,31,30,31,31,30,31,30,31};
char tab[2][3]={{1,2,3}, {4,5,6}};
```



文字列について：

- char 型の配列を使う。
- 文字列の終わりの印として文字列の最後にヌル文字'\0' を置く。
  $\Longrightarrow$ (配列の大きさ) $\geq$ (文字列の長さ) + 1 でなければならない。



- 文字列を 2 重引用符で囲めば文字列定数になる。
- char 型配列で文字列を表す場合は、初期設定を次の様に行うことが出来る。

```
char s[]="string";
```
（`char s[]={'s','t','r','i','n','g','\0'};` と同等。）

10 関数の定義 —処理の分割—

- 関数定義の例,
- 名前の有効範囲, 局所変数, 大域変数,
- 再帰計算,
- パラメータの受渡し方法,
- 配列を関数パラメータとして受け渡す,
- 段階的詳細化,
- 付録 関数についてのまとめ —C 文法のまとめ (3)—,
- 付録 標準ライブラリ関数のまとめ

これまでは、コンピュータに処理させたい手順全てを main() 関数の処理部に書き込んできた。 しかし、(幾つかのパラメータを除いて) 全く同一の処理を 1 つのプログラムの中で複数回行いたいこともある。この様な場合、それら各々の細かな処理を別々に手順の中に書き込むと、プログラムが長く読みにくくなってしまう。

補足：
本質的に同じ処理がプログラムのあちこちで繰り返されていない場合でも、長い処理手順は広い範囲で共有される変数を含むので概して分かりにくくなる。

そこで、C 言語ではプログラムの複数箇所に現れる同一の処理を main() とは別の関数として記述し、その関数を printf(), scanf() や数学関数の様に呼び出すことができる様になっている。 この節では、この様にプログラムを複数の**機能単位** (ここでは関数) に分けて構成する方法について説明する。

10.1 関数定義の例

まず手始めに、簡単な例題から始めよう。

例題 10.1 (二項係数の計算) n 個のものから k 個を選ぶ組合せの数 $\binom{n}{k}$ は

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

と計算できる。 2 つの正整数データ n と k を読み込みこの計算式に基づいて組合せの数 $\binom{n}{k}$ を計算して出力する C プログラムを作成せよ。

(考え方) 計算式に階乗計算が 3 箇所もあるので、main() 関数とは別に、階乗計算を行う関数 factorial() を定義するのが自然であろう。 引数として整数値を受け取りその階乗値を返す関数 factorial() が記述されていれば、組合せの数 $\binom{n}{k}$ の計算は、数学関数と同じ様に factorial() を呼び出して

$$\binom{n}{k} = \frac{\texttt{factorial(}n\texttt{)}}{\texttt{factorial(}k\texttt{)}\,\texttt{factorial(}n-k\texttt{)}}$$

という風に行うことができる。関数 factorial() に与える引数データは、我々が入力する正整数、およびそれらの差であるので、そのデータ型は int とするのが妥当である。

また、factorial() の関数値は本来整数であるが、int 型で表せる範囲を越えてしまう危険性もあるので、そのデータ型を実数型にして階乗値も組合せの数も近似計算する方が無難である。　階乗計算については、例題 7.6 と同じ風に行えば良い。

(プログラミング)　階乗計算を行う関数 factorial() は、引数として int 型のデータを受け取り、関数値として double 型のデータを返すものとする。関数 factorial() の中では、引数として受け取るデータを格納するために k という名前の int 型変数を、1!,2!,3!,... の値を保持するために fact という名前の double 変数を、そして for 文による繰り返しを制御するために i という名前の int 型変数を用意する。　また、main() 関数の中では、読み込んだ正整数 n と k を格納するために各々 n と k という名前の int 型変数を用意してプログラムを構成した。この C プログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl binomial-coeff.c Enter
     1 /* 2つの正整数データ n と k を読み込み              */
     2 /* 二項係数 n!/(k!*(n-k)!) を出力する C プログラム */

     3 #include  <stdio.h>

     4 double factorial(int k);

     5 int main(void)
     6 {
     7   int  n, k;

     8   printf("It will compute a binomial coefficient.\n"
     9          "Input two positive integers n and k(<=n):  ");
    10   scanf("%d%d", &n, &k);

    11   printf("\nThe number of the combinations of\n"
    12          "      n objects taken k at a time = %20.14g\n",
    13          factorial(n)/(factorial(k)*factorial(n-k)));
    14   return 0;
    15 }

    16 /*-------------------------------------------------------*/
    17 /* 階乗値を計算してその結果を返す関数                    */
    18 /*-------------------------------------------------------*/
    19 /*   (仮引数) k : 何の階乗を計算するかを表す整数        */
    20 /*   (関数値)   : k! の値を double で                    */
    21 /*-------------------------------------------------------*/
    22 double factorial(int k)
    23 {
    24   int    i;
```

```
    25   double fact;

    26   fact = 1.0;
    27   for (i=2; i<=k; ++i)
    28     fact *= (double)i;
    29   return fact;
    30 }
[motoki@x205a]$ gcc binomial-coeff.c Enter
[motoki@x205a]$ ./a.out Enter
It will compute a binomial coefficient.
Input two positive integers n and k(<=n):  50 25 Enter

The number of the combinations of
      n objects taken k at a time =  1.2641060643775e+14
[motoki@x205a]$
```

ここで、

- プログラムの4行目は、22〜30行目で定義した関数 factorial() がどういう型の引数を受け取りどういう型の値を返すのかを宣言した文で、**関数プロトタイプ** (または **関数原型**) と呼ばれる。 4行目には k という名前の変数名が書かれているが、これは省略可能で、これによって変数領域の確保を指示している訳ではない。一般に関数プロトタイプは次のような構造をしている。

  関数値のデータ型 関数名 (データ型 名前, ... , データ型 名前);

  または

  関数値のデータ型 関数名 (データ型 , ... , データ型);

  コンパイラはプログラムを前から順に見て行くので、この宣言が無いとコンパイラは13行目を見る時点で、呼び出された関数 factorial() の引数として計算したものをどういうデータ型に変換して factorial() に引き渡せば良いかも分からないし、factorial() の計算結果として返って来たデータをどういう風に解釈すれば良いかも分からない。 #include の指令は、大抵の場合、printf() や scanf() といった標準ライブラリ関数についての、関数プロトタイプを読み込むのが目的となっている。
- プログラムの5〜15行目は関数 main() を定義した部分、22〜30行目は関数 factorial() を定義した部分になっている。関数値の型の宣言が省略された場合は、int 型が暗黙に仮定される。
- プログラムの7行目にも22行目にも k という名前の変数が確保されているが、これらは別の変数として扱われる。 実際、8〜14行目で変数 k を使うと7行目で確保した変数 k として扱われ、23〜29行目で変数 k を使うと22行目で確保した変数 k として扱われる。
- プログラムの13行目では関数 factorial() が3回呼び出されている。関数が呼び出されると、関数に引き渡されたデータの値を記憶する変数 k、および関数本体の最初に宣言されている変数 i のための領域が新たに動的に確保される。
- 一般に、呼び出しの際に関数名の後の丸括弧の中で指定し、関数に実際に引き渡すデータのことを**実引数**と呼ぶ。これに対し、実引数の値を記憶するために関数の中に

用意される変数のことを**仮引数**と呼ぶ。

- プログラム 29 行目 の `return` 文は、関数の実行を終了し `factorial()` の計算結果 (関数値) として式 `fact` の値を呼び出し元に返すことを表す。

> 補足：
> 関数の呼び出し元に戻される値は関数の計算結果を表すので、これまでこの値のことを**関数値**と呼んできたが、C 言語では通常この値のことを**戻り値** (あるいは**返却値**) と呼ぶ。

- プログラムの 16〜21 行目 は関数 `factorial()` の仕様、すなわちこの関数を呼び出す側に対してどういう機能を提供するかを明確に記述した注釈である。

> 関数の仕様を記述することの利点：
> 各々の関数に仕様が書かれていると、作り上げた関数の処理内容を理解する際、その関数から呼び出す別の関数については処理内容を詳しく見る代わりに仕様を見るだけで良いので、一度に把握するプログラムの範囲が小さくて済む。
>
> $\Longrightarrow$ ◇ プログラムを理解し易くなる。
> ◇ 多数の関数が複雑に絡み合った大きなプログラムを作る場合もしっかりとしたプログラムを作ることが可能となる。
>
> ---
>
> 関数仕様の書き方について：
> 仕様としては、関数を使う側に対してどういう機能を提供するのかを書く。
> それゆえ、
> ◇ 与えられた引数に対して、どういう関数値が返されるかを書く。
> ◇ 関数の外側で確保された変数等の値を変える作用、いわゆる副作用がある場合は、どういう副作用があるかも書く。
> ◇ 関数の内部の細かな変数や、処理手順についての記述は控えるべきである。

□**演習 10.2 (円錐の体積)** 2 つの実数データ r と h をパラメータとして受け取り、底面の半径が `r` で高さが `h` の円錐の体積を計算結果として返す関数を定義することによって、例題 7.1 のプログラムを作り直してみよ。

□**演習 10.3 (最大公約数)** 2 つの正整数をパラメータとして受け取りその最大公約数を計算結果として返す関数を定義することによって、例題 6.12 のプログラムを作り直してみよ。

□**演習 10.4 (素数かどうかの判定)** 正整数を 1 個パラメータとして受け取り、それが素数かどうかを判定してその結果を返す関数を定義することによって、例題 6.16 のプログラムを作り直してみよ。

□**演習 10.5 (冪乗)** 実数 a と 非負整数 k をパラメータとして受け取り a^k を計算結果として返す関数を定義せよ。

□**演習 10.6 (Fibonacci 数列)** 一般に、初期値 a_0, a_1 と漸化式

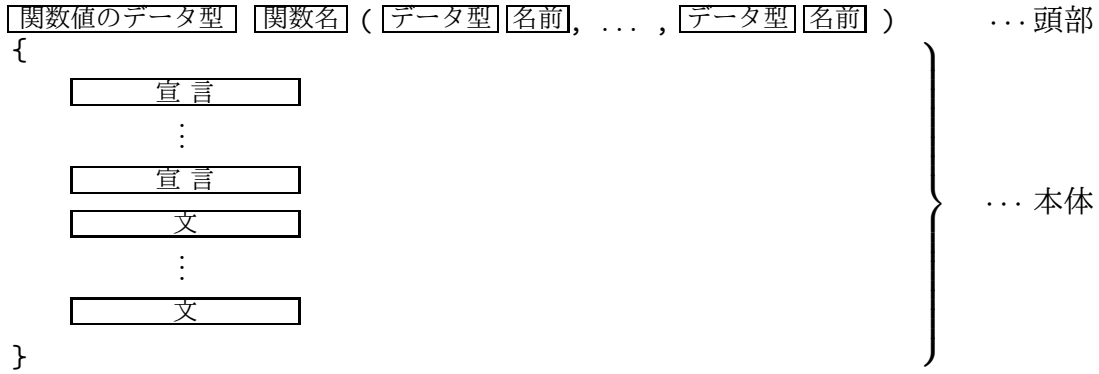
$$a_n = a_{n-1} + a_{n-2} \qquad \text{for each } n \geq 2$$

によって決まる数列 $\{a_n\}$ を **Fibonacci 数列**という。46 以下の非負整数 k をパラメータとして受け取り、初期値が $a_0 = a_1 = 1$ の Fibonacci 数列の第 $(k+1)$ 項 a_k を計算結果として返す関数を定義せよ。

□**演習 10.7 (三角形が出来るかどうかの判定)** 3 つの実数 a, b, c をパラメータとして受け取り、a, b, c を 3 辺の長さとする三角形が存在するかどうかの判定結果を返す関数を定義せよ。

10.2 名前の有効範囲, 局所変数, 大域変数

もし仮に変数や配列が**大域的** (global) でプログラム内のどの場所からでもアクセスできるとしたら、そのプログラムを 同時に全て見渡して 各部分の役割を見定めない限り、そのプログラムの動作を見極めることはできない。大域的な変数や配列はプログラムを分かりにくくする原因にもなる。 それゆえ、C 言語においては、関数定義



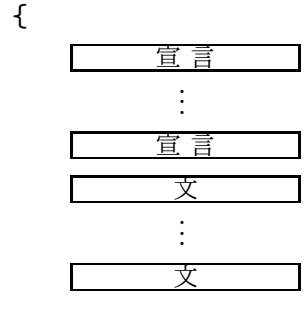
の中で宣言され確保される変数や配列は、その関数定義の本体の中だけで使える**局所的** (local) なものとして扱われ、この関数の外部からはアクセスできない様になっている。

補足：

- 通常の場合は、関数定義の中で宣言されている変数や配列は、その関数が呼び出された直後に領域が確保され関数実行が終了するとともに領域が解放されるので、関数の外部からアクセスしようにもできない。
- `printf()` や `sin()` も関数である。もし仮にこれらの関数の内部で使われている変数に我々の作った関数からアクセスできるとしたら、プログラムの動作が不安定になってしまう。
  $\Longrightarrow$ 変数や配列の有効範囲の局所性はプログラムの信頼性の上でも大切。

実際には、関数定義の**本体部** (i.e. 仮引数列に続く `{` と `}` で囲まれた部分) は、ブロックの一種と考えられる。

ブロックと複合文： C 言語においては次の構造のものを**複合文**と呼び、そのうち実際に宣言が 1 個以上含まれているものを**ブロック**と呼ぶ。



複合文／ブロックは、1 つの文が書ける所であればどこにでも置くことが出来るので、ブロックの中により小さなブロックが入り、その内側のブロックの中にまた別のブロックが入り、..... という**入れ子構造** も可能である。プログラムの中に出来るこの「ブロックの入れ子構造」に基づいて、変数等に付けた**名前の有効範囲**が次の様に決まる。

(規則 1) どの名前 (の領域) も、それが宣言されたブロックの中だけでアクセスできる。

(規則 2) 外側のブロックで宣言された名前を内側のブロックで再定義すると、外側の名前の領域 (i.e. その名前を持った外側の変数) は内側のブロックからはアクセスできなくなる。

(規則 1) の理由：
ブロック内で宣言された変数や配列の領域は、ブロック内の宣言の場所に制御が移ると自動的に確保され、ブロックの出口に制御が移ると解放される。

ブロックの中で宣言され局所的に使われるこれらの変数を**自動変数**という。

大域的な名前：

- 関数名はそのファイルのどの場所からでもアクセスできる。
- 関数の外で宣言された変数や配列はそのファイルのどの場所からでもアクセスできる。 (**外部変数, 外部配列**という。)

$\Longrightarrow$ ファイル全体をブロックの一種と見なすことも出来る。

注意：
外部変数を使い過ぎると、副作用のために関数同士の独立性がなくなり、分かりにくいプログラムになる恐れがあります。

次の例題は、C 言語における名前の有効範囲の規則を例示するものである。

例題 10.8 (名前の有効範囲, 外部変数) 次の C プログラムを実行するとどういう出力が得られるか？　下の [　　] の部分に予想される出力文字列を入れよ。但し、ここでは空白は 」と明示せよ。

```
[motoki@x205a]$ nl scope-of-name.c Enter
     1  /* 名前の有効範囲、外部変数の理解のためのプログラム例 */

     2  #include <stdio.h>

     3  void sub(void);

     4  int  a = 1;       /* 外部変数 */

     5  int main(void)
     6  {
     7    int  a = 22;    /* 自動変数 */
     8    printf("(1) %d\n", a);

     9    {                       /* ブロックの始まり */
    10      int  a = 333;
    11      printf("(2) %d\n", a);
    12    }                       /* ブロックの終わり */
```

```
    13    printf("(3) %d\n", a);
    14    sub();
    15    return 0;
    16  }

    17  void sub(void)
    18  {
    19    int  b = 4444;

    20    printf("(4) %d\n", a);
    21    printf("(5) %d\n", b);
    22  }
[motoki@x205a]$ gcc scope-of-name.c Enter
[motoki@x205a]$ ./a.out Enter
[motoki@x205a]$
```

(文法上の注意)

- プログラム 3 行目 の 1 つ目の void は関数 sub() の戻り値がないことを明示し、2 つ目の void は関数 sub() の引数がないことを明示している。
- プログラム 4 行目 は、int 型外部変数 a を確保しその初期値を 1 とすることを指示している。

(考え方) プログラム 9〜12 行目, および 6〜16 行目, 18〜22 行目 がブロックとなっているから、このプログラムの中に出来る「ブロックの入れ子構造」 を明示すると次の様になる。

```
 1  /* 名前の有効範囲、外部変数の理解のためのプログラム例 */
 2  #include <stdio.h>
 3  void sub(void);
 4  int  a = 1;       /* 外部変数 */
 5  int main(void)
 6  {
 7    int  a = 22;    /* 自動変数 */
 8    printf("(1) %d\n", a);
 9    {                       /* ブロックの始まり */
10      int  a = 333;
11      printf("(2) %d\n", a);
12    }                       /* ブロックの終わり */
13    printf("(3) %d\n", a);
14    sub();
15    return 0;
16  }
17  void sub(void)
18  {
19    int  b = 4444;
20    printf("(4) %d\n", a);
21    printf("(5) %d\n", b);
22  }
```

それゆえ、

- プログラムの 4 行目 で宣言された外部変数 a は、同じ名前の変数が宣言された内側の 6〜15 行目のブロックの外側で有効である。
- プログラムの 7 行目 で宣言された自動変数 a は 6〜16 行目のブロックの入口で宣言されており、また同じ名前の変数が更に内側の 9〜12 行目のブロックでも宣言されている。従って、7 行目の a は 6〜8 行目, 13〜16 行目で有効となる。
- プログラムの 10 行目 で宣言された自動変数 a は、9〜12 行目のブロックの入口で宣言されており、またこのブロックは別のブロックを含まない。従って、10 行目の a は 9〜12 行目のブロック内で有効となる。
- プログラムの 19 行目 で宣言された自動変数 b は、18〜22 行目のブロックの入口で宣言されており、またこのブロックは別のブロックを含まない。従って、19 行目の b は 18〜22 行目のブロック内で有効となる。

(実行結果) 結局、プログラムの

$$\begin{cases} 8\text{行目の a は} \quad 7\text{行目で確保された a として、} \\ 11\text{行目の a は} \quad 10\text{行目で確保された a として、} \\ 13\text{行目の a は} \quad 7\text{行目で確保された a として、} \\ 20\text{行目の a は} \quad 4\text{行目で確保された a として、} \\ 21\text{行目の b は} \quad 19\text{行目で確保された b として} \end{cases}$$

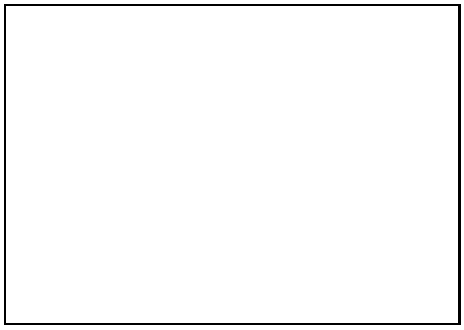
解釈されることになるから、実行結果は次の様になる。

```
[motoki@x205a]$ ./a.out Enter
(1)␣22
(2)␣333
(3)␣22
(4)␣1
(5)␣4444
[motoki@x205a]
```

□**演習 10.9 (名前の有効範囲)** 次の C プログラムを実行するとどういう出力が得られるか？ 下の [　　] の部分に予想される出力文字列を入れよ。但し、ここでは空白は ␣ と明示せよ。

```
[motoki@x205a]$ nl scope-of-name-lab.c Enter
     1 #include <stdio.h>

     2 int  a=-1;

     3 int main(void)
     4 {
     5   int  b=2;
     6   printf("(1)a=%3d  b=%3d\n", a, b);
     7   {
     8     int    a=33;
     9     float  b=50;
```

```
10      printf("(2)a=%3d  b=%3.0f\n", a, b);
11      {
12        int  b=777;
13        printf("(3)a=%3d  b=%3d\n", a, b);
14      }
15      printf("(4)a=%3d  b=%3.0f\n", a, b);
16      {
17        int  b=999;
18        printf("(5)a=%3d  b=%3d\n", a, b);
19      }
20      printf("(6)a=%3d  b=%3.0f\n", a, b);
21    }
22    printf("(7)a=%3d  b=%3d\n", a, b);
23    return 0;
24  }
[motoki@x205a]$ gcc scope-of-name-lab.c Enter
[motoki@x205a]$ ./a.out Enter
```



```
[motoki@x205a]$
```

10.3 再帰計算

例えば、漸化式

$$f_i = \begin{cases} 1 & \text{if } i = 1 \\ i \times f_{i-1} & \text{if } i \geq 2 \end{cases}$$

が与えられていれば、f_i の値は

$$f_i = i \times f_{i-1} = i \times (i-1) \times f_{i-2} = i \times (i-1) \times (i-2) \times f_{i-3} = \cdots = i!$$

と計算できる。 従って、この漸化式で f_i の計算式の中に f_{i-1} が出て来るのと同じ様に、関数定義の中に自分自身 (i.e. 定義しようとしている関数) の呼び出しを書くことができれば、漸化式に相当する関数定義を行い、漸化式による計算と同等の計算をその関数定義に基づいて行うことが、原理的にできるはずである。

実際、C 言語においては、関数定義の中で自分自身を呼び出すことが許されており、またその様な関数を実行する機構も備わっているので、漸化式による計算と同等の計算をプログラム上で行うことができる。一般に、関数定義の中で自分自身を呼び出すことを**再帰呼び出し** (recursive call) と言い、再帰呼び出しを伴う関数の実行を**再帰計算**と言う。 2 つ

の関数定義の中で互いに相手の関数を呼び出し合っている場合も、間接的に自分自身を呼び出しているので、再帰呼び出しの一種と考える。

例題 10.10 (二項係数; 階乗の再帰計算) 漸化式

$$f_i = \begin{cases} 1 & \text{if } i = 1 \\ i \times f_{i-1} & \text{if } i \geq 2 \end{cases}$$

によって $f_i = i!$ と定まる。これを考慮に入れて、整数を 1 個引数として受け取りその階乗値を double 型で計算して返す関数 factorial() を再帰的に定義せよ。 そして、この関数を用いて例題 10.1 と同じことを行う C プログラムを作成せよ。すなわち、2 つの正整数データ n と k を読み込み、n 個のものから k 個を選ぶ組合せの数を $\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$ と計算して出力する C プログラムを作成せよ。

(考え方) 例題 10.1 で構成した main() 関数は、定義変更が求められている関数 factorial() の呼び出しを含んでいる。しかし、ここで定義する関数 factorial() の仕様 (呼び出す側に対して提供する機能) 自体は例題 10.1 の場合と変わらないので、main() 関数については例題 10.1 のものを何も変更せずにそのまま使える。従って、例題 10.1 で示したプログラムの中で、関数 factorial() を指示に従って再帰的に定義し直すだけである。

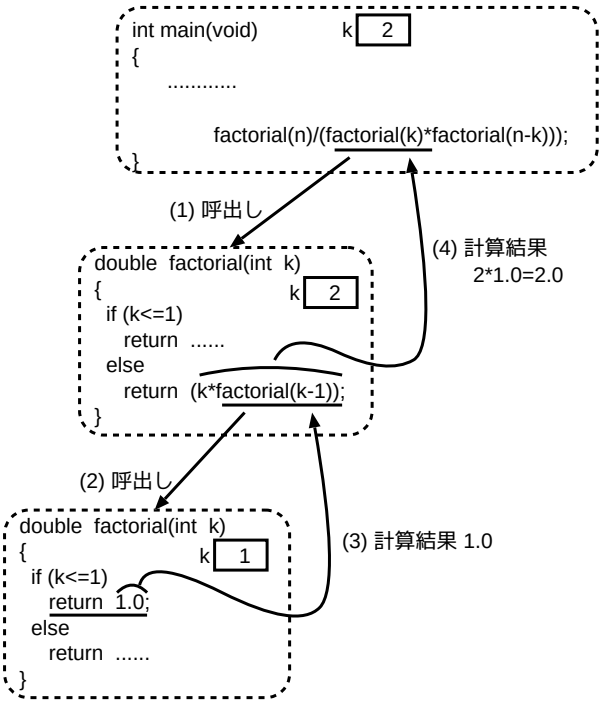
(プログラミング) 関数 factorial() を定義するにあたっては、例題 10.1 の場合と同じく仮引数として k という名前の int 型変数を用意した。作成した C プログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl binomial-coeff-using-rec-fatorial.c Enter
     1 /* 2つの正整数データ n と k を読み込み             */
     2 /* 二項係数 n!/(k!*(n-k)!) を出力する C プログラム */
     3 /*  (階乗値を再帰的に計算する関数を用意する。)     */

     4 #include  <stdio.h>

     5 double factorial(int k);

     6 int main(void)
     7 {
     8   int  n, k;

     9   printf("It will compute a binomial coefficient.\n"
    10          "Input two positive integers n and k(<=n):  ");
    11   scanf("%d%d", &n, &k);

    12   printf("\nThe number of the combinations of\n"
    13          "      n objects taken k at a time = %20.14g\n",
    14          factorial(n)/(factorial(k)*factorial(n-k)));
    15   return 0;
    16 }
```

```
17 /*-------------------------------------------------------*/
18 /* 階乗値を計算してその結果を返す関数（再帰版）          */
19 /*-------------------------------------------------------*/
20 /*   （入力引数）k : 何の階乗を計算するかを表す整数      */
21 /*   （関数値）    : k! の値を double で                 */
22 /*-------------------------------------------------------*/
23 double factorial(int k)
24 {
25   if (k <= 1)
26     return 1.0;
27   else
28     return (k * factorial(k-1));
29 }
[motoki@x205a]$ gcc binomial-coeff-using-rec-fatorial.c Enter
[motoki@x205a]$ ./a.out Enter
It will compute a binomial coefficient.
Input two positive integers n and k(<=n):  50 25 Enter

The number of the combinations of
      n objects taken k at a time =  1.2641060643775e+14
[motoki@x205a]$
```

ここで、

- プログラムの 23〜29 行目 で階乗計算の関数 `factorial()` を定義しているが、この中の 28 行目 で今計算手順を書こうとしている `factorial()` 自身を再帰的に呼んでいる。
- 11 行目で `k` の値として 2 が入力された場合 の `factorial(k)` の処理の様子を次に示す。



- 漸化式の計算を行いたい場合、再帰を用いれば漸化式の形をそのまま反映した形で容易に関数定義を行うことが出来ることに注目せよ。

□**演習 10.11 ((3n+1) 数列)** 初期値 a_1 と漸化式

$$a_{n+1} = \begin{cases} 1 & \text{if } a_n = 1 \\ a_n/2 & \text{if } a_n\text{が偶数} \\ 3a_n + 1 & \text{otherwise} \end{cases}$$

によって定まる数列 $\{a_n\}$ を考える。 2 つの正整数 a と k をパラメータとして受け取り、初期値が $a_1 = a$ の時のこの数列の第 k 項の値を計算して返す関数を再帰的に定義してみよ。

□**演習 10.12 (Fibonacci 数列)** 46 以下の非負整数 k をパラメータとして受け取り、初期値が $a_0 = a_1 = 1$ の Fibonacci 数列 (演習 10.6) の第 $(k+1)$ 項 a_k を計算結果として返す関数を再帰的に定義してみよ。

□**演習 10.13 (McCarthy の 91 関数)** 次の漸化式によって定義される整数から整数への関数 $f(x)$ を計算する関数 (プログラム) を再帰的に定義してみよ。また、この関数 (プログラム) を再帰無しで定義してみよ。 [**補足：** 実は $x > 100$ の時には $f(x) = x - 10, x \leq 100$ の時には $f(x) = 91$ となる。]

$$f(x) = \begin{cases} x - 10 & \text{if } x > 100 \\ f(f(x+11)) & \text{otherwise} \end{cases}$$

10.4 パラメータの受渡し方法 —値呼出し vs. 参照呼出し—

実引数と仮引数の対応付け： 関数呼び出しの際には、呼ぶ側と呼ばれる側の情報交換、すなわち関数呼び出し側の引数 (**実引数**または**実パラメータ**という) と関数定義側の引数 (**仮引数**または**仮パラメータ**という) の結合／対応付けが行われる。 **引数結合**の方式としては次の 2 つが一般によく用いられている。

- **値呼出し** (call by value) ··· 実引数として与えられた式が評価／計算され、その値が仮引数の変数の初期値として使われる。
- **参照呼出し** (call by reference) ··· 実引数として与えられた変数の記憶領域と仮引数の変数領域を同一視する。従って、呼び出された関数が直接呼出し側の変数を操作することになる。

これらの内 C 言語で行えるのは値呼出しのみであるが、下の例 10.14 で例示されているように、変数の主記憶内での番地 (**ポインタ**という) を関数に引き渡すことにより参照呼出しと同等のことも行える。

関数実行のプロセス： 関数呼出しがあると、その処理は次のような順序で進む。

(1) 各々の実引数を評価。
(2) (1) の結果を対応する**仮引数のデータ型**に変換。
(3) (2) の結果を対応する仮引数 (変数) に代入。
} (値呼出し)

(4) 関数の本体を実行する。実行の途中に、

(場合 1) `return;` という文に出会うと、制御を呼出し元に戻す。(関数値なし)

(場合 2) 本体の実行が終了すると、制御を呼出し元に戻す。(関数値なし)

(場合 3) `return` 式 ; という文に出会うと、式 の値を評価し、その値をその**関数が本来返すべきデータ型**に変換する。 そして、その結果を関数値として制御を呼出し元に戻す。

次の例題は、

①C 言語においては引数結合が値呼出しによって行われていること、そして

②値呼出しを用いて参照呼出しと同等のことも行えること

を説明している。

例題 10.14 (値呼出し, 参照呼出し) 次の C プログラムを実行するとどういう出力が得られるか？ 下の [　　] の部分に予想される出力文字列を入れよ。但し、ここでは空白は ␣ と明示せよ。

```
[motoki@x205a]$ nl func-binding-parameters.c Enter
     1 #include <stdio.h>
     2 void call_by_value(int);
     3 void call_by_reference(int *);

     4 int main(void)
     5 {
     6   int  a=1;

     7   printf("%d\n", a);
     8   call_by_value(a);       /* 値呼出し*/
     9   printf("%d\n", a);      /* a の値は不変！*/

    10   call_by_reference(&a); /* 参照呼出し*/
    11   printf("%d\n", a);      /* a の値は変わる！*/
    12   return 0;
    13 }

    14 void call_by_value(int a)
    15 {
    16   a = 777;
    17 }
```

```
    18 void call_by_reference(int *a)
    19 {
    20   *a = 777;
    21 }
[motoki@x205a]$ gcc func-binding-parameters.c Enter
[motoki@x205a]$ ./a.out Enter
[motoki@x205a]$
```

(文法上の注意)

- プログラム 2〜3 行目, 14 行目, 18 行目 で関数値の型が `void` と宣言されているが、これは関数値を返さないことを表す。
- プログラム 10 行目 の `&a` は変数 `a` にアクセスするためのデータ (**ポインタ**という) を表す。ポインタの実体は主記憶内の番地である。
- プログラム 18 行目, 20 行目 の `*a` はポインタ `a` の指す (すなわち `a` 番地の) 記憶領域を表す。

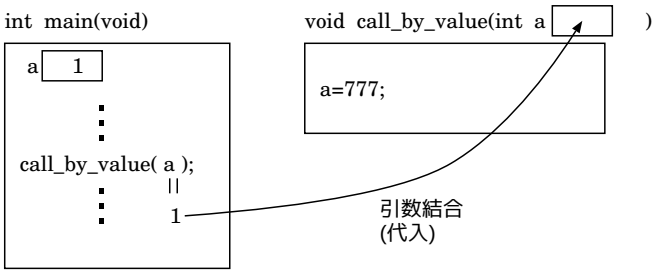
(**考え方**) プログラムの 6 行目, 14 行目, 18 行目で同じ `a` という名前の変数が宣言されているが、これらの変数はそれぞれ 5〜13 行目, 14〜17 行目, 18〜21 行目 が有効範囲の別々の変数として扱われる。 C 言語では、

関数引数の結合が値呼出しによって行われる

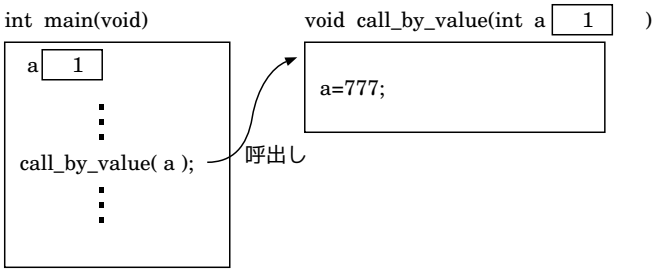
から、

- もし実行が 8 行目に移り `call_by_value()` が次に実行されることになれば、
  この関数呼び出しにより、6 行目で宣言された `a` の値 (この時点では 1 のはず) が 14 行目で宣言された変数 (仮引数) `a` の初期値として引き渡され、15 行目以降の関数の処理が進む。すなわち、次の様に実行が進む。

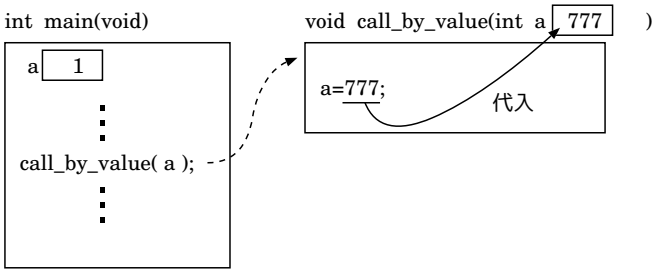
(8 行目, 引数結合)



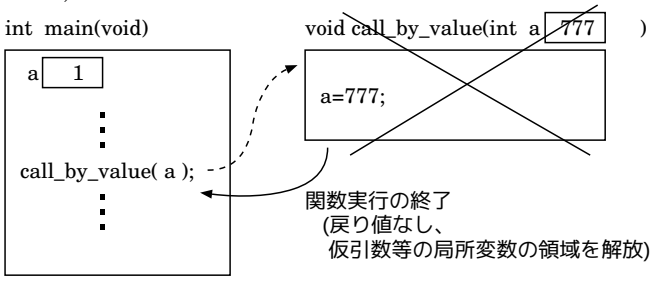
(8 行目, 関数呼び出し)



(16 行目, 実行後)



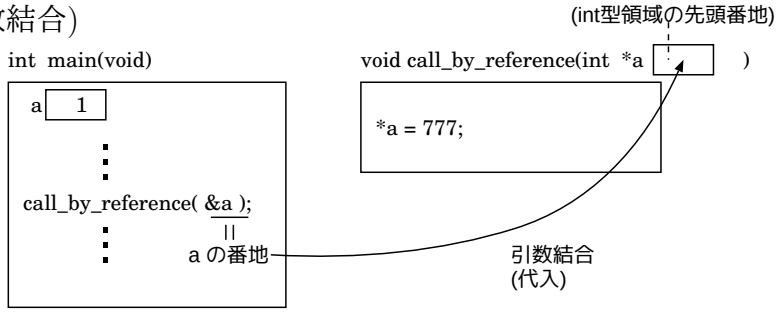
(17 行目, 関数実行終了)



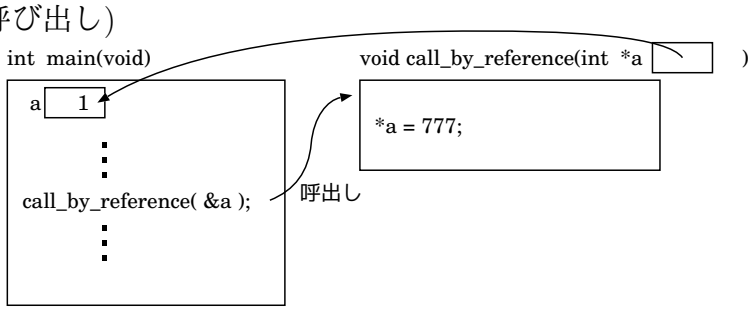
$\Longrightarrow$ 8 行目の関数実行終了後も 6 行目の a の値は 1 のまま変わらない。

- もし実行が 10 行目に移り call_by_reference() が次に実行されることになれば、この関数呼び出しにより、&a の値, すなわち 6 行目で宣言された変数 a の番地が 18 行目で宣言された変数 (仮引数) a の初期値として引き渡され、19 行目以降の関数の処理が進む。すなわち、次の様に実行が進む。

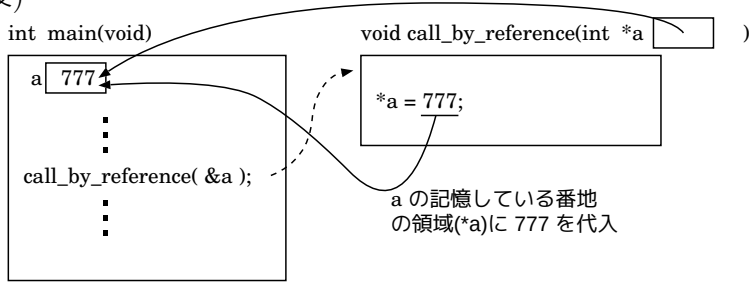
(10 行目, 引数結合)



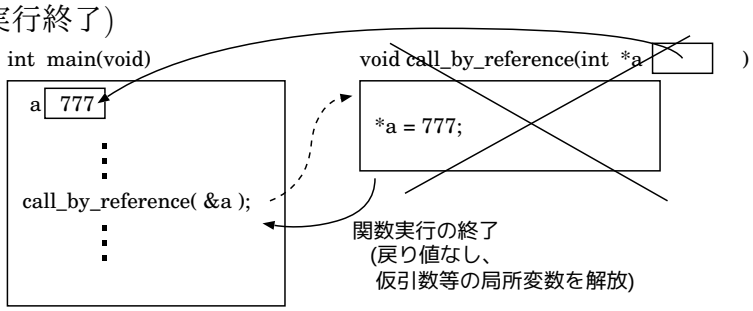
(10 行目, 関数呼び出し)



(20 行目, 実行後)



(21 行目, 関数実行終了)



$\Longrightarrow$ 10 行目の関数実行によって 6 行目の a の値は 777 に変わる。

(実行結果) 結局、プログラムの

$$\begin{cases} \text{7 行目では a の値は} \quad 1, \\ \text{9 行目では a の値は} \quad 1, \\ \text{11 行目では a の値は} \quad 777 \end{cases}$$

になるから、実行結果は次の様になる。

```
[motoki@x205a]$ ./a.out Enter
1
1
777
[motoki@x205a]
```

番地演算子 & と間接演算子 * :

&v ··· 変数 v へのポインタ (≈ 番地)。

*p ··· ポインタ p の指す記憶領域、すなわち p 番地の記憶領域。
$\Longrightarrow$ 変数 v があった時、*(&v) と v は同等。

参照呼出しと同等のことを行なう方法 :

- 参照呼出しの仮引数は、ポインタとして宣言する。
- 関数の本体部では、参照呼出しの仮引数は間接演算子 * を付けて使う。
- 関数を呼ぶ時、参照呼出しの実引数として変数等へのポインタ (i.e. 番地) を与える。

□**演習 10.15 (値呼出し, 参照呼出し)** 次の C プログラムを実行するとどういう出力が得られるか?

```
#include <stdio.h>

void sub1(int, int);
int  sub2(int, int);
void sub3(int *, int *);

int main(void)
{
  int a;

  a = 0;
```

```
  sub1(a, a);
  printf("(sub1) %d\n", a);

  a = 0;
  a = sub2(a, a);
  printf("(sub2) %d\n", a);

  a = 0;
  sub3(&a, &a);
  printf("(sub3) %d\n", a);
  return 0;
}

void sub1(int x, int y)
{
  x++;
  y++;
}

int sub2(int x, int y)
{
  x++;
  y++;
  return y;
}

void sub3(int *x, int *y)
{
  (*x)++;
  (*y)++;
}
```

10.5 配列を関数パラメータとして受け渡す

C 言語における関数パラメータ受渡しの方法は値呼出しであると言っても、配列データの受渡しを行いたい場合、配列要素毎に値呼出しによる引数結合を行っていたのでは引数結合に相当の時間がかかってしまう。 そこで、C 言語では、配列データの受渡しを行いたい場合には、その配列 (の先頭要素) へのポインタを呼び出し先の関数に引き渡す様にする。

一次元配列 a を関数の引数として受渡しする方法：

- 仮引数側では、

  データ型 配列名 [] または データ型 *配列名 または データ型 配列名 [大きさ]

  という書き方をする。

  補足：
  配列の大きさを明示する必要はない。 明示したとしても捨てられる。

- 関数本体の中では、仮引数の配列要素の参照はこれまでと全く同じ様な書き方をする。
- 実引数側では、

  a または &a[0]

  という書き方をする。

配列名は、
計算機内部では**先頭要素を指す定数ポインタ**として扱われている。

例題 10.16 (一次元配列の一部を関数パラメータとして受け渡す) double 型一次元配列の部分要素列についての情報を引数として受け取り、その部分配列内の要素の総和を計算して返す関数 sum() を定義せよ。 そして、

v[k]$= 2^{49-k}$ $(k = 0$〜$49)$

という風に値の設定された大きさ 50 の double 型一次元配列について、この関数を用いて、

v[0]+v[1]+v[2]+···+v[49],
v[40]+v[41]+v[42]+···+v[49],
v[20]+v[21]+v[22]+···+v[39]

の値を計算して出力する C プログラムを作成せよ。

(考え方) 素直に考えるなら、部分配列内の要素の総和を計算する関数 sum() には

① 配列の名前 (i.e. 先頭要素の番地),
② 総和の始めとなる配列要素の添字番号,
③ 総和を締めくくる配列要素の添字番号

の 3 つを引数として引き渡すことが頭に浮かぶ。もちろん、これは妥当な考えで、関数 sum() も使い易くなる。($\Longrightarrow$ 具体的なプログラミングは演習問題として残しておく。)

しかし、配列を関数パラメータとして受け渡しする場合、実際に受け渡されるのは配列要素へのポインタ (i.e. 番地) に他ならない。 呼ばれる関数側としては、そのポインタが実際に関数を呼んだ側で確保された配列の先頭番地を指しているかどうかは重要ではなく、そのポインタが指す領域以降に然るべき型のデータ領域が十分に長く (i.e. 関数実行の間に配列アクセス違反を起こさない程度に) 確保されていれば良いだけである。 従って、逆に、これさえ守れば良い訳で、もし

> double 型配列の名前 a と大きさ size を引数として受け取り、
> a[0]+a[1]+···+a[size-1] を計算して返す関数
> double sum(double a[], int size)

が定義できているなら、この関数を sum(&v[from], size) という風に使うことも許されるはずで、この呼び出しによって部分配列の総和 v[from]+v[from+1]+···+v[from+size-1] が計算されることになる。

確認：
配列要素 v[from]〜v[from+size-1] が sum() を呼び出す側で確保されていれば、確かに、
◇ &v[from] は配列要素を指すポインタで、
◇ &v[from] 番地以降にも同じ型のデータが十分長く続いている。

呼び出された側の関数が実際にメモリ確保された領域だけを使うようにするのはプログラマの責任である。

(プログラミング) (部分) 配列の要素の総和を計算する関数 sum() は、引数として配列の名前 (先頭要素の番地) a と大きさ size を受け取り、a[0]+···+a[size-1] を計算して返すものとする。この関数 sum() の中では、途中までの総和 a[0]+···+a[i] ($0 \leq i <$size) を保持するために関数名と同じ sum という名前の double 型局所変数を用意する。 また、main() 関数の中では、配列要素 v[0]〜v[49] に対する値の設定は数学関数 pow() を使うのではなく

v[49] ⟵ 1,
v[48] ⟵ v[49]×2,
v[47] ⟵ v[48]×2,
.....

という風に行うことにして、プログラムを構成した。この C プログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl func-bind-part-of-array.c  Enter
     1 #include <stdio.h>

     2 double sum(double a[], int size);

     3 int main(void)
     4 {
     5   int    i;
     6   double v[50];

     7   v[49] = 1.0;
     8   for (i=48; i>=0; --i)
     9     v[i] = v[i+1] * 2.0;

    10   printf("v[0] +v[1] + ... +v[49] = %16.0f\n", sum(v, 50));
    11   printf("v[40]+v[41]+ ... +v[49] = %16.0f\n", sum(&v[40], 10));
    12   printf("v[40]+v[41]+ ... +v[49] = %16.0f\n", sum(v+40, 10));
    13   printf("v[20]+v[21]+ ... +v[39] = %16.0f\n", sum(v+20, 20));
    14   return 0;
    15 }

    16 /*-------------------------------------------------------------*/
    17 /* double 型配列 (もしくは配列の断片) の要素の総和を計算して返す */
    18 /*-------------------------------------------------------------*/
    19 /*   (仮引数)    a : double型配列                              */
    20 /*             size : double型配列 a の大きさ                 */
    21 /*   (関数値) : a[0]+a[1]+a[2]+...+a[size-1]                   */
    22 /*-------------------------------------------------------------*/
    23 double sum(double a[], int size)
    24 {
    25   int    i;
```

```
26   double sum=0.0;

27   for (i=0; i < size; ++i)
28     sum += a[i];
29   return sum;
30 }
[motoki@x205a]$ gcc func-bind-part-of-array.c Enter
[motoki@x205a]$ ./a.out Enter
v[0] +v[1] + ... +v[49] = 1125899906842623
v[40]+v[41]+ ... +v[49] =               1023
v[40]+v[41]+ ... +v[49] =               1023
v[20]+v[21]+ ... +v[39] =         1073740800
[motoki@x205a]$
```

ここで、

- 10.2節で説明した名前の有効範囲の規則により、sum という名前は、プログラムの24〜30行目のブロックの中では26行目で宣言された局所変数として解釈され、そのブロックの外では2行目と23行目で宣言された関数名として解釈される。
- プログラムの11行目は、一次元配列の一部 v[40]〜v[49] の先頭番地とその大きさを関数 sum() に引き渡している。

> 補足：
> 関数側では受け渡されるものが元々(呼出し側で)配列として確保されたものかどうかのチェックはしない。仮引数として与えられるものを配列の先頭番地と見なして処理を進めるだけである。

- プログラムの12行目も11行目と同じ処理をしている。一次元配列の場合、配列名は計算機内部では先頭要素を指す定数ポインタとして扱われているので、v+40 は先頭から40個後の要素を指すポインタ値である。

> 補足：
> ポインタに関する算術は普通の算術演算とは異る。v+40 の番地は実際には &v[0]+40×sizeof(double) 番地、すなわち &v[40] 番地 である。

□演習 10.17 (一次元配列の一部を関数パラメータとして受け渡す) double型配列の名前 a と2つの配列添字 from, to を引数として受け取り、a[from]+a[from+1]+···+a[to] を計算して返す関数 double sum(double a[], int from, int to) を定義することによって、例題10.16のプログラムを再構成してみよ。

□演習 10.18 (整列化の関数) 例題9.6のプログラム (バブル整列法) の中の整列化を行っている部分 (17〜24行目) を関数化して、プログラムを再構成してみよ。

> **Hint**：
> 一般的な整列化の関数を構成するのであれば、データの入った配列 (の先頭番地) とその大きさは引数として受け渡す必要があります。

多次元配列を関数の引数として受渡しする方法：

- 仮引数側では、1 次元目を除く全ての次元の大きさを指定して、
  データ型 配列名 [] [大きさ] ··· [大きさ]
  または　データ型 配列名 [大きさ] [大きさ] ··· [大きさ]
  または　データ型 (*配列名) [大きさ] ··· [大きさ]
  という書き方をする。

2 次元目以降の次元の大きさを指定する理由：
そうしないと、コンパイラが配列の添字の値からその配列要素の番地を割り出せないからである。 また、1 次元目の配列の大きさについては明示する必要はない。 明示したとしても捨てられる。

- 関数本体の中では、仮引数の配列要素の参照はこれまでと全く同じ様な書き方をする。
- 実引数側では、
  a　または　`&a[0]`
  という書き方をする。

例えば：
a が 3 次元配列の場合、配列名 a は計算機内部では 2 次元配列 `a[0]` を指す定数ポインタとして扱われている。

例 10.19 (多次元配列を関数の仮引数とする場合) 3 次元配列 `int a[7][9][2]` を引数として受渡したい時には、仮引数部は次のように書く。

```
int a[][9][2]
int a[7][9][2]                    } いずれか
int (*a)[9][2]      ←― 明示的な書き方
```

例題 10.20 (行列の積を計算する関数)　例題 9.8 では、3×3 実数値行列 $A = [a_{ij}]$, $B = [b_{ij}]$ の要素を読み込み、これらの行列の積 AB を計算してその結果を表示するプログラムを提示した。このプログラムの中の行列の積を計算する部分を関数化し、プログラム全体を再構成してみよ。

(考え方)　計算結果は単一の数値ではないので、これを関数の戻り値とするのは無理がある。("構造体"を使えばできないこともない。) そこで、乗算対象の行列データを保持する 2 つの 2 次元配列 `a[][]`, `b[][]` だけでなく、計算結果のデータを格納する 2 次元配列 `productAB[][]` (の先頭番地) も関数引数として受け渡す、

```
void matrix_multiplication(double x[][3], double y[][3],
                                          double productXY[][3]);
```

という関数を考え、この中で

(配列 `productXY` の表す行列) ←― (配列 `x` の表す行列) × (配列 `y` の表す行列)

という処理を行なうようにすれば良い。

(プログラミング)　次の通り。

```
[motoki@x205a]$ nl matrix-multiplication-function.c  Enter
     1 /* 3 × 3 実数値行列 A=[a_ij], B=[b_ij] の要素を読み込み、 */
     2 /* 行列の積 AB を計算してその要素を 2 次元状に見易く出力する  */
     3 /* C プログラム                                              */
```

```
 4 #include <stdio.h>

 5 #define  N  (3)

 6 void matrix_multiplication(double x[][N], double y[][N],
 7                             double productXY[][N]);

 8 int main(void)
 9 {
10   double a[N][N], b[N][N], productAB[N][N];
11   int  i, j;

12   /* 行列 A,B の各要素を入力する */
13   for (i=0; i<N; i++) {
14     printf("行列 A の %d 行目の要素を順に入力して下さい:  ", i);
15     for (j=0; j<N; j++)
16       scanf("%lf", &a[i][j]);
17   }
18   printf("\n");
19   for (i=0; i<N; i++) {
20     printf("行列 B の %d 行目の要素を順に入力して下さい:  ", i);
21     for (j=0; j<N; j++)
22       scanf("%lf", &b[i][j]);
23   }

24   /* 行列の積 AB の各要素を計算して配列 productAB に記録する */
25   matrix_multiplication(a, b, productAB);

26   /* 行列の積 AB の結果を表示する */
27   printf("\n行列の積 AB の結果:\n");
28   for (i=0; i<N; i++) {
29     printf("   ");
30     for (j=0; j<N; j++)
31       printf("  %12.5g", productAB[i][j]);
32     printf("\n");
33   }
34   return 0;
35 }

36 /*-------------------------------------------------------------*/
37 /* 行列の積を計算                                              */
38 /*-------------------------------------------------------------*/
39 /*   (仮引数)    x : double型2次元配列                         */
```

```
 40 /*                y : double型2次元配列                          */
 41 /*        productXY : double型2次元配列, 計算結果の格納場所     */
 42 /*   (関数値) : なし                                             */
 43 /*   (機能) : (x[][]の表す行列)×(y[][]の表す行列) の計算を      */
 44 /*             行ないその結果を productXY[][] に格納する。      */
 45 /*-------------------------------------------------------------*/
 46 void matrix_multiplication(double x[][N], double y[][N],
 47                            double productXY[][N])
 48 {
 49   int i, j, k;

 50   for (i=0; i<N; i++) {
 51     for (j=0; j<N; j++) {
 52       productXY[i][j] = 0.0;
 53       for (k=0; k<N; k++)
 54         productXY[i][j] += x[i][k]*y[k][j];
 55     }
 56   }
 57 }
[motoki@x205a]$
```

10.6 段階的詳細化

複雑な仕事内容を計算機で処理する際は、プログラムのモジュール化、すなわち、

> 小さなプログラムを部品 (module) として構成し、それらの部品を使うことによってより大きなプログラムを構築する、

というソフトウェア構築法が有効である。[プログラムの大きさが2倍になった場合、分かり難さは2倍では済まない。]

モジュール化の利点：

- プログラムの構造化、系統化 $\Longrightarrow$ プログラムが分かり易く修正し易くなる。

  (特に、プログラム内の各モジュールの独立性が高く、各モジュールについての仕様 (i.e. 何をパラメータに持ち、外部に対してどんな働きをするかを説明した文書) を知るだけでプログラム全体の処理の流れを理解できる様になっていれば、プログラム全体の処理が見通せて分かり易いものとなる。)

- 大きなプログラムを何人かで分担して作るのに都合が良い。
- 同一の処理単位を何回も重複してプログラムに組む込む必要がなくなる。
  $\Longrightarrow$ プログラムの簡素化

何をモジュールと考えるか： 通常のプログラミング言語では、機能的にまとまりのあるプログラム断片を 1 つの関数または手続きとして定義／登録でき、また、これらの関数や手続きを必要に応じて呼び出せる様になっているので、一般にはプログラムを設計する際に**関数や手続きをモジュールと考えるのが最も素朴で自然**である。

段階的詳細化： モジュール化を実際に進めるための手法としては**段階的詳細化** (stepwise refinement)、すなわち、

> 処理手順を少しずつ詳細化していくことによってプログラム／アルゴリズムを作り上げてゆく、

という考え方がよく用いられている。 このプログラム設計方針に従えば、処理手順は大雑把なものから (十分に) 細かなものへと少しずつ詳細化されていくことになるから、**詳細化の途中に現われる処理単位のうち機能的にまとまりのあるものをモジュールにすれば**、元の大きな処理は比較的きれいに分割され、プログラムのモジュール化が自然に進むことになる。

10.7 付録 関数についてのまとめ —C 文法のまとめ (3)—

C 言語における関数の扱い：

- C 言語においては、値を返さない関数を手続きと考えて、手続きを定義する手段は用意されていない。
- 関数の定義を並べたものがプログラムになる。
- 全ての関数は同一水準にある。 すなわち、関数定義の中で別の関数を局所的に定義することは許されていない。
- プログラムの起動は `main` という名前の関数の実行で始まる。(`main` が主プログラム。)
- (標準ライブラリ関数も含めた) 全ての関数は、使用する前にその引数の型、関数値の型、すなわち**関数プロトタイプ**を宣言しておかなければならない。(これが分かってないとコンパイラは翻訳できない。) 関数プロトタイプの宣言は例えば次の様に行う。

  ```
  double pow(double x, double y);
  ```
  または `double pow(double, double);`
- 標準ライブラリ関数については、関数プロトタイプの宣言は `<stdio.h>` 等のヘッダファイルの中に置かれている。
- 関数呼び出しの際の引数結合は常に値呼出しで行われる。(但し、`&` 演算子を用いれば、参照呼出しと同等のことも行える。)

関数定義：

- 一般形は次の通り。

  関数値のデータ型 関数名 (データ型 名前, ... , データ型 名前) ・・・頭部

- 値を返さない関数を定義する場合は[関数値のデータ型]の部分は `void` とする。
- [関数値のデータ型]の部分を省略すると `int` が暗黙に仮定される。
  (しかし、これを当てにして省略するのは良くない。)
- `static` という修飾子を付けて宣言しない通常の場合は、関数定義の中で宣言されている変数や配列は、その関数が呼び出された直後に領域が確保され関数実行が終了するとともに領域が解放される。

return 文：

- 構文は次のいずれか。

```
return;
return 式 ;
```

- `return` 文に出くわすと、その関数の実行は終了する。(呼出し元に戻る。)
- [式]が指定されていると、その値 (を指定されたデータ型に変換したもの) が関数値になる。
- `return` 文に出会わないまま関数の本体部の処理が終わった場合も、その関数の実行は終了する。 (当然、関数値はない。)

関数プロトタイプ：

- 構文は次のいずれか。

```
関数値のデータ型 関数名 ( データ型 名前, ... , データ型 名前 );
関数値のデータ型 関数名 ( データ型 , ... , データ型 );
```

- 関数の引数の個数と型、および関数値の型をコンパイラに知らせるための宣言。
- 関数を呼び出す前に、その関数を定義するかプロトタイプを宣言しないといけない。
  [この情報が分からないと、コンパイラは例えば戻って来た計算結果 (ビット列) をどう解釈してよいか分からない。]
- 標準ライブラリ関数のプロトタイプは `<stdio.h>`, `<stdlib.h>`, ... に入っている。

10.8 付録 標準ライブラリ関数のまとめ

- 標準ライブラリ関数については、関数プロトタイプの宣言は次のいずれかの標準ヘッダファイルの中に置かれている。

```
<assert.h>  <limits.h>  <signal.h>  <stdlib.h>
<ctype.h>   <locale.h>  <stdarg.h>  <string.h>
<errno.h>   <math.h>    <stddef.h>  <time.h>
<float.h>   <setjmp.h>  <stdio.h>
```

$\Longrightarrow$ 標準ライブラリ関数を使いたければ、その関数のプロトタイプが入っている標準ヘッダファイルをインクルードしなければならない。

- 標準ヘッダファイルの中には、用途別に関数プロトタイプだけでなくマクロ定義なども入っている。各々の内容は次の通り。

標準ヘッダファイル	内容
<assert.h>	プログラムが思惑通に働いているかをチェックするための、引数付きマクロの定義が入っている。
<ctype.h>	文字の種類 (e.g. 制御文字, 印字可能文字, 数字, 小文字,...) をテストしたり変換したりするための関数のプロトタイプが入っている。
<errno.h>	ライブラリ関数がエラーを検出したとき、その報告をするのに使うマクロ等が定義されている。
<float.h>	浮動小数点数型 (e.g. float, double) の各々の特性と限界を定めるマクロ、例えば表せる正の最小値を定めたマクロ等が入っている。
<limits.h>	整数型 (e.g. char, short, int, long) の各々の特性と限界を定めるマクロ、例えば表せる最大値を定めたマクロ等が入っている。
<locale.h>	地域化処理のためのデータ型、マクロ、関数プロトタイプが入っている。
<math.h>	数学関数に関するマクロ、関数プロトタイプが入っている。講義ノート 7.4 節を参照。
<setjmp.h>	非局所的分岐： 関数の実行環境を保存したり復元したりするためのデータ型、マクロ、関数プロトタイプが入っている。
<signal.h>	実行時のエラー, 外部からの割り込みといった、実行時に起こる例外状態を処理するためのマクロ、関数プロトタイプが入っている。
<stdarg.h>	可変引数リストを持つ関数 (e.g.printf) の引数を処理するためのデータ型、マクロが定義されている。
<stddef.h>	共通に使われるデータ型、マクロの定義が入っており、その中にはコンパイラに固有のものもある。

標準ヘッダファイル	内容
<stdio.h>	入出力に関するデータ型、マクロ、関数プロトタイプが入っている。
<stdlib.h>	記憶域確保, 疑似乱数発生, 強制終了, 文字列を数値に変換, など、いわゆるユーティリティ関数のプロトタイプが入っている。
<string.h>	文字列を操作するための関数のプロトタイプが入っている。
<time.h>	日付と時刻を扱うためのデータ型、マクロ、関数プロトタイプが入っている。

以下、標準ヘッダファイルの中で定義されているデータ型, マクロ, 関数プロトタイプの内で、有用そうなものを簡単に紹介する。

文字種類テストの関数/引数付きマクロ <ctype.h>：

関数プロトタイプ	説明
int isalnum(int c)	c が英数字か?
int isalpha(int c)	c が英字か?
int iscntrl(int c)	c が制御文字か?
int isdigit(int c)	c が数字か?
int isgraph(int c)	c が空白以外の印字可能文字か?
int islower(int c)	c が小文字か?
int isprint(int c)	c が印字可能文字 (空白も含む) か?
int ispunct(int c)	c が区切り文字か?
int isspace(int c)	c が空白類か?
int isupper(int c)	c が大文字か?
int isxdigit(int c)	c が 16 進数字か?

文字種類変換の関数 <ctype.h>：

関数プロトタイプ	説明
int tolower(int c)	c を小文字に変換
int toupper(int c)	c を英大文字に変換

□**演習 10.21 (<ctype.h>の中のマクロ)** <ctype.h>の中を覗いて、isalpha(c), isupper(c), isdigit(c), isalnum(c), isspace(c), isprint(c), iscntrl(c), isascii(c), ... がどのように定義されているか見てみよ。

共通に使うデータ型, マクロ <stddef.h>：

名前	説明
ptrdiff_t	「2 つのポインタの差」を表すデータ型
size_t	sizeof 演算の結果を表すデータ型で、 typedef unsigned int size_t; と定義されている。
NULL	ヌルポインタを表すマクロ
wchar_t	「多バイト文字の番号」を表すデータ型を

□**演習 10.22** (<stddef.h>の中のマクロ) <stddef.h>の中を覗いて、ptrdiff_t, size_t, NULL, wchar_t, ... がどのように定義されているか見てみよ。

入出力に関するデータ型, マクロ <stdio.h>：

名前	説明
FILE	ファイルのアクセス状況を記録した構造体のデータ型。 入力用か出力用か、次の読み込み文字の位置、ファイル終端が起きたかどうか、などの情報から成る。
EOF	「ファイルの終わり」の値を表すマクロ
NULL	空ポインタを表すマクロ
stdin	標準入力を表すマクロ
stdout	標準出力を表すマクロ
stderr	標準エラー出力を表すマクロ

□**演習 10.23** (<stdio.h>の中のマクロ) <stdio.h>の中を覗いて、FILE, EOF, NULL, stdin, stdout, stderr, getc(p), putc(x, p) , ... がどのように定義されているか見てみよ。

ファイルをオープン・クローズする関数 <stdio.h>：

関数プロトタイプ ··· 説明
FILE *fopen(const char *filename, const char *mode) ··· ファイルをオープンし、そのファイルポインタを返す。オープンに失敗すると NULL を返す。ここで、filename はオープンするファイルの名前（文字列）へのポインタである。mode が"r" の時は読み込みを、"w" の時は書き出しを、"a" の時は追加書き出しを、"rb" の時はバイナリファイルの読み込みを、"r+" の時はテキストファイルを読み書き両用にオープンすることを表す。
int fclose(FILE *fp) ··· ファイルをクローズする。ここで、fp はファイルポインタ。
FILE *freopen(const char *filename, const char *mode, FILE *fp) ··· ファイルポインタ fp に付随するファイルをクローズし、代わりに新しくファイルをオープンし fp に結びつける。

書式付き入出力の関数 <stdio.h>：

関数プロトタイプ ··· 説明
int printf(const char *cntrl_string, ...) ··· 標準出力への書式付き出力。講義ノート 5.5 節を参照。
int fprintf(FILE *fp, const char *cntrl_string, ...) ··· 指定した出力ストリームへの書式付き出力。
int sprintf(char *s, const char *cntrl_string, ...) ··· 指定した char 型配列への書式付き出力。最後に空文字\0 も出力して、出力結果を文字列とする。
int scanf(const char *cntrl_string, ...) ··· 標準入力からの書式付き入力。講義ノート 5.6 節を参照。
int fscanf(FILE *fp, const char *cntrl_string, ...) ··· 指定した入力ストリームからの書式付き入力。
int sscanf(char *s, const char *cntrl_string, ...) ··· 指定した文字列 (char 型配列) からの書式付き入力。 **注意**： 実行する度に、指定した配列の先頭から入力作業を開始する。

1文字入出力の関数 <stdio.h>：

関数プロトタイプ ··· 説明
`int getchar(void)` ··· 標準入力のストリームから 1 文字だけ (空白も可) 読み込んで、その文字コードの値を返す。但し、ファイルの終りまたはエラーを検出した時は EOF を返す。
`int fgetc(FILE *fp)` ··· 指定した入力ストリームから 1 文字だけ (空白も可) 読み込んで、その文字コードの値を返す。ファイルの終りまたはエラーを検出した時は EOF を返す。
`int ungetc(int c, FILE *fp)` ··· 指定した入力ストリームに c という文字コードを戻す。
`int putchar(int c)` ··· 標準出力ストリームに文字コード c の文字を書き出す。成功すると (int)(unsigned char)c を返し、失敗すると EOF を返す。
`int fputc(int c, FILE *fp)` ··· 指定した出力ストリームに文字コード c の文字を書き出す。

1行入出力の関数 <stdio.h>：

関数プロトタイプ ··· 説明
`char *gets(char *s)` ··· 標準入力ストリームから改行コード又はファイルの終りまでの文字の並びを読み込み、char 型配列 s に格納する。その際、改行コードは空文字\0 に置き換えられる。通常は s が返されるが、ファイル終了又はエラー発生時には NULL が返される。セキュリティ上の問題 (バッファオーバーラン) があり使うべきでない関数とされ、2011 年の言語仕様改定で C11 の標準 C ライブラリから廃止された。gcc では使うと警告が出るらしい。
`char *fgets(char *line, int n, FILE *fp)` ··· 指定した入力ストリームから、改行コード又はファイルの終りまでの文字の並び (但し長くなっても n-1 文字で打ち切り) を読み込み、最後に空文字 \0 を付けて char 型配列 line に格納する。通常は line の値が関数値として返されるが、ファイル終了又はエラー発生時には NULL が返される。
`int puts(const char *s)` ··· 標準出力ストリームに文字列 s を書き出す。但し、文字列の最後の空文字 \0 の代わりに改行コードを書き出す。成功すると非負の値を返し、失敗すると EOF を返す。
`int fputs(const char *s, FILE *fp)` ··· 指定した出力ストリームに文字列 s を書き出す。但し、文字列の最後の空文字 \0 は出力しない。[puts と違って、代わりに改行コードを書き出すこともしない。]

バイナリファイルの入出力を行う関数 <stdio.h>：

関数プロトタイプ ··· 説明
`size_t fread(void *a_ptr, size_t el_size, size_t n, FILE *fp)` ··· 指定した入力ストリームから、1 要素 el_size バイトのデータを n 個 (但しファイル終了になるとそこまで)、a_ptr が指す配列に格納する。関数値は読み込んだ要素数である。
`size_t fwrite(const void *a_ptr, size_t el_size, size_t n, FILE *fp)` ··· a_ptr が指す配列から、1 要素当たり el_size バイトのデータを n 個取り出し、指定した出力ストリームに書き出す。関数値は書き出しに成功した要素数である。

ファイルの読み込み位置/書き込み位置を設定する関数 <stdio.h>：

- オープンしたファイルは、通常、**前から順に**処理しますが、ファイルの先頭や末尾からの距離を指定して、(原理的には) 任意の場所にアクセスすることが出来る。また、現在見ている場所 (先頭からのバイト数) を知ることも出来る。
- 内部的には、ファイル中の現在処理している場所は、**ファイル位置指示子**と呼ばれる記憶領域の中に記録される。　これはファイルポインタの指す FILE 型構造体のメンバで、通常は、この値がファイルの先頭場所から始まって少しずつ大きくなる。

関数プロトタイプ　　･･･　　説明
`int fseek(FILE *fp, long offset, int place)` ･･･ ファイル位置指示子の値を place から offset バイト離れた所に設定する。ここで、place としては SEEK_SET (ファイルの先頭を表す; 通常 0), SEEK_CUR (現在位置を表す; 通常 1), SEEK_END (ファイルの末尾を表す; 通常 2) のいずれかを指定する。成功すると 0 を返し、失敗すると 0 以外の値 を返す。
`void rewind(FILE *fp)` ･･･ ファイル位置指示子をファイルの先頭に設定する。
`long ftell(FILE *fp)` ･･･ ファイル位置指示子の現在の値 (先頭からのバイト数) を返す。但し、エラーを検出した時は -1 を返す。

一時ファイルをオープンする関数 <stdio.h>：

関数プロトタイプ　　･･･　　説明
`FILE *tmpfile(void)` ･･･ 一時的な使用目的のための (バイナリ) ファイルを "wb+" という利用モードでオープンし、そのファイルポインタを返す。オープンに失敗すると NULL を返す。この一時ファイルは、クローズまたはプログラム終了時に削除される。

入出力に関するその他の関数 <stdio.h>：

関数プロトタイプ　　･･･　　説明
`int fflush(FILE *fp)` ･･･ fp で指定したストリームが出力用の時、そのストリーム向けに溜ったバッファデータを実際にストリームに吐き出す。
`int feof(FILE *fp)` ･･･ 指定したストリームにファイル終了の標識が立っているかどうかを調べ、立っていれば 0 以外、立っていなければ 0 を返す。
`int remove(const char *filename)` ･･･ 指定したファイルを削除する。
`int rename(const char *from, const char *to)` ･･･ ファイルの名前を変更する。

記憶域を動的に確保する関数 <stdlib.h>：

関数プロトタイプ ··· 説明
void *malloc(size_t size) ··· size バイトの記憶域を (ヒープ領域から) 確保し、その先頭へのポインタを返す。記憶域確保に失敗すれば空ポインタ NULL を返す。
void *realloc(void *ptr, size_t size) ··· ptr の指す記憶域の内容を保存したまま、その大きさを size に変更する。成功すれば、変更後の記憶域の先頭へのポインタを返し、失敗すれば空ポインタ NULL を返す。
void *calloc(size_t n, size_t el_size) ··· 1 要素が el_size バイトで要素数が n 個の配列のための連続領域を (ヒープ領域から) 確保し、全てのビットを 0 にクリアした後、その先頭へのポインタを返す。失敗すれば空ポインタ NULL を返す。
void free(void *ptr) ··· ptr が指す記憶域を解放する。ptr が NULL の時は何も起きない。

疑似乱数発生のためのマクロ,関数 <stdlib.h>：

関数プロトタイプ/マクロ名 ··· 説明
RAND_MAX ··· 関数 rand() が返す int 型疑似乱数の最大値を表すマクロ
int rand(void) ··· 区間 [0, RAND_MAX] の間の疑似整数乱数を返す。
int srand(unsigned seed) ··· 関数 rand() の生成する疑似乱数の種を seed に設定する。デフォルトでは seed=1 である。

プログラムを強制終了するためのマクロ,関数 <stdlib.h>：

関数プロトタイプ/マクロ名 ··· 説明
void exit(int status) ··· プログラムを正常終了させ、status を主ルーチンの関数値として呼び出し元 (OS) に返す。呼び出し元は、status=0 の時にプログラムが正常終了したと判断する。
EXIT_SUCCESS ··· 関数 exit() の引数として使うマクロで、通常は 0 と定義されている。成功終了を表す。
EXIT_FAILURE ··· 関数 exit() の引数として使うマクロで、通常は 1 と定義されている。異常終了を表す。

環境変数へのアクセス,OS コマンド実行ための関数 <stdlib.h>：

関数プロトタイプ ··· 説明
char *getenv(const char *name) ··· 指定した環境変数の値 (文字列) へのポインタを返す。
int system(const char *s) ··· 指定したコマンドを OS が提供するコマンドインタープリタに実行してもらう。

文字列を数値に変換するための関数 <stdlib.h>：

関数プロトタイプ ··· 説明
`double atof(const char *s)` ··· s の指す文字列を実数と見て、それを double 型の内部表現形式に変換して返す。
`int atoi(const char *s)` ··· s の指す文字列を整数と見て、それを int 型の内部表現形式に変換して返す。
`int atol(const char *s)` ··· s の指す文字列を整数と見て、それを long int 型の内部表現形式に変換して返す。

検索, 整列のための関数 <stdlib.h>：

関数プロトタイプ ··· 説明

`void *bsearch(const void *key_ptr, const void *a_ptr, size_t n,`
`size_t el_size, int (*compar)(const void *, const void *) )`
··· 昇順に並んだ 1 次元配列の中から key_ptr が指すものと等しい要素を探し出し、そこへのポインタを返す。見つからなければ NULL を返す。ここで、a_ptr は昇順に並んだ 1 次元配列 (の先頭要素) を指すポインタ、n は配列の大きさ、el_size は配列要素 1 個の占めるバイト数、compar は 2 つの要素の大小を判定する関数 (**比較関数**という) へのポインタである。比較関数の 2 つの引数は大小を比較する要素へのポインタであり、これらの引数を基に比較関数は
(第 1 引数の指す要素) < (第 2 引数の指す要素) なら 負、
(第 1 引数の指す要素) = (第 2 引数の指す要素) なら 零、
(第 1 引数の指す要素) > (第 2 引数の指す要素) なら 正
の値を返す。データ型の所で指定された const は引数の値が変えられないことを宣言している。また、引数の型として指定されている (void *) は総称的なポインタ型で、この型のポインタはどんなポインタ変数にも代入可能である。

`void *qsort(const void *a_ptr, size_t n, size_t el_size,`
`int (*compar)(const void *, const void *) )`
··· 1 次元配列の要素を比較関数 compar の基準に従って昇順に並べ換える。 ここで、a_ptr は昇順に並んだ 1 次元配列 (の先頭要素) を指すポインタ、n は配列の大きさ、el_size は配列要素 1 個の占めるバイト数である。

整数の絶対値, 商と剰余のペアを求める関数 <stdlib.h>：

関数プロトタイプ/データ型 ··· 説明
`div_t` ··· 関数 div() が返す構造体 (int 型のペア) のデータ型名
`ldiv_t` ··· 関数 ldiv() が返す構造体 (long int 型のペア) のデータ型名
`int abs(int i)` ··· i の絶対値 (int 型) を返す。
`long labs(long i)` ··· i の絶対値 (long int 型) を返す。
`div_t div(int number, int denom)` ··· number を denom で割った時の商と剰余の組を返す。
`ldiv_t ldiv(long number, long denom)` ··· number を denom で割った時の商と剰余の組を返す。

文字列の長さを測るための関数 <string.h>：

関数プロトタイプ ··· 説明
size_t strlen(const char *s) ··· 文字列 s の長さを返す。

文字列の連接, コピーをするための関数 <string.h>：

関数プロトタイプ ··· 説明
char *strcat(char *s1, const char *s2) ··· 文字列 s2 を文字列 s1 の末尾にコピーし、s1 の値を返す。
char *strncat(char *s1, const char *s2, size_t n) ··· 文字列 s2 を文字列 s1 の末尾にコピーし、s1 の値を返す。但し、s2 の長さが n を越える場合は最初の n 文字だけをコピーし、その後に空文字\0 を追加する。
char *strcpy(char *s1, const char *s2) ··· 文字列 s2 を末尾の空文字 \0 も含めて s1 の指す領域にコピーし、s1 の値を返す。
char *strncpy(char *s1, const char *s2, size_t n) ··· 文字列 s2 を s1 の指す領域にコピーし、s1 の値を返す。但し、s2 の長さが n 以上の時は最初の n 文字だけをコピーする。(空文字 \0 は追加しない。) s2 の長さが n 未満の時は n-(s2 の長さ) 個の空文字をコピーの末尾に埋めておく。

2 つの文字列を比較するための関数 <string.h>：

関数プロトタイプ ··· 説明
int strcmp(const char *s1, const char *s2) ··· 2 つの文字列 s1 と s2 を辞書式順序で比較する。その結果、s1 が s2 より小さければ負、等しければゼロ、大きければ正の値を返す。
int strncmp(const char *s1, const char *s2, size_t n) ··· 2 つの文字列 s1 と s2 の最初の n 文字の部分を辞書式順序で比較する。その結果、s1 が s2 より小さければ負、等しければゼロ、大きければ正の値を返す。

文字列の中で文字を探索する関数 <string.h>：

関数プロトタイプ ··· 説明
char *strchr(const char *s, int c) ··· 文字 (コード) c を文字列 s の最初から探す。見つかればその最初の位置へのポインタを返し、見つからなければ空ポインタ NULL を返す。
char *strrchr(const char *s, int c) ··· 文字 (コード) c を文字列 s の最後から逆向きに探す。見つかればその最初の位置へのポインタを返し、見つからなければ空ポインタ NULL を返す。
char *strpbrk(char *s1, const char *s2) ··· 文字列 s2 内に含まれる文字を文字列 s1 の最初から探す。 見つかればその最初の位置へのポインタを返し、見つからなければ空ポインタ NULL を返す。

文字列を探索する関数 <string.h>：

関数プロトタイプ ··· 説明
`char *strstr(char *s1, const char *s2)` ··· 文字列パターン s2 を文字列 s1 の最初から探す。見つかればその最初の部分文字列の先頭位置へのポインタを返し、見つからなければ空ポインタ NULL を返す。
`size_t strspn(char *s1, const char *s2)` ··· 文字列 s1 の先頭からの部分文字列で、文字列 s2 内に含まれる文字だけで構成される部分の長さを返す。
`size_t strcspn(char *s1, const char *s2)` ··· 文字列 s1 の先頭からの部分文字列で、文字列 s2 内に含まれない文字だけで構成される部分の長さを返す。
`char *strtok(char *s1, const char *s2)` ··· 文字列 s2 内の各文字を区切り記号と見て文字列 s1 を走査し、s1 の中に現れる字句 (i.e. 区切り記号以外から成る文字の並び) を探す。字句が見つかればその直後の文字が空文字に書き換えられた上でその字句の先頭位置へのポインタを返し、見つからなければ空ポインタ NULL を返す。引き続き、s2 を空ポインタにしてこの関数が呼び出されると、前回の走査の続きの位置から走査が始まる。

バイト列をコピーするための関数 <string.h>：

関数プロトタイプ ··· 説明
`void *memcpy(void *to, const void *from, size_t n)` ··· from の指す長さ n のバイト列 (i.e. 文字の並び; 空文字\0 が最後に来る保証はない) を to の指す領域にコピーし、to の値を返す。領域が重なっている場合の動作は定義されない。
`void *memmove(void *to, const void *from, size_t n)` ··· from の指す長さ n のバイト列 (i.e. 文字の並び; 空文字\0 が最後に来る保証はない) を to の指す領域にコピーし、to の値を返す。領域が重なっていても正しくコピーされる。
`void *memset(void *p, int c, size_t n)` ··· p の指す領域に 1 バイトデータ (unsigned char)c を続けて n 個格納し、p の値を返す。

2 つのバイト列を比較するための関数 <string.h>：

関数プロトタイプ ··· 説明
`int memcmp(const void *p1, const void *p2, size_t n)` ··· p1 と p2 の指す 2 つのバイト列の最初の n バイトの部分を辞書式順序で比較する。その結果、p1 の方が p2 のバイト列より小さければ負、等しければゼロ、大きければ正の値を返す。

バイト列の中で文字を探索する関数 <string.h>：

関数プロトタイプ ··· 説明
`void *memchr(const void *p, int c, size_t n)` ··· バイトデータ (unsigned char)c を p の指すバイト列の最初から高々 n バイト探す。見つかればその最初の位置へのポインタを返し、見つからなければ空ポインタ NULL を返す。

日付と時間に関するデータ型, マクロ <time.h>：

名前	説明
CLOCKS_PER_SEC	関数 clock() の想定している 1 秒当たりのクロック数を表すマクロ。
clock_t	各々の計算機で独自に設定されている時間量 (クロック数) 表すためのデータ型で、clock() の返す関数値もこのデータ型を持つ。
time_t	暦上の日付, 時刻を表すためのデータ型で、time() の返す関数値もこのデータ型を持つ。
struct tm	日付と時間の情報をまとめた構造体

時間計測の関数 <time.h>：

関数プロトタイプ ··· 説明
clock_t clock(void) ··· プログラム実行のためにそれまでにプロセッサを使用した時間 (クロック数) を返す。
double difftime(time_t t2, time_t t1) ··· 2 つのカレンダ時刻 t2 と t1 の差 t2-t1 を計算し、それに相当する秒単位の時間を double 型で返す。

現在の時刻を知るための関数 <time.h>：

関数プロトタイプ ··· 説明
time_t time(time_t *tp) ··· 現在のカレンダー時間として、1970 年 1 月 1 日 0 時 0 分 0 秒からの経過秒数を返す。
struct tm *localtime(const time_t *t_ptr) ··· t_ptr が指すカレンダ時間をローカル時間に変換し、その時間に相当する struct tm 型データへのポインタを返す。
char *asctime(const struct tm *tp) ··· tp が指す struct tm 型データを例えば Sun Feb 24 17:30:27 2002 といった文字列に変換し、その文字列へのポインタを返す。
char *ctime(const time_t *t_ptr) ··· asctime(localtime(t_ptr)) と同等。すなわち、t_ptr が指すカレンダ時間をローカルな時刻を表す Sun Feb 24 17:30:27 2002 といった文字列に変換し、その文字列へのポインタを返す。
size_t strftime(char *s, size_t n, const char *format, const struct tm *tp) ··· tp が指す struct tm 型時刻データを format が指す書式に従って変換し、得られた文字列を s が指す領域に格納する。但し、n 文字を越えた文字列が得られた場合は最初の n 文字だけを格納する。関数値は、格納された文字の長さである。

11　基本的なデータ型と構造体, 共用体

- 整数型, 浮動小数点数型,
- C 言語における文字, 文字列の扱い,
- typedef による新しいデータ型の定義,
- 構造体,
- 共用体

第 3 節で見たように、コンピュータ内ではいくつかの標準的な内部表現方式に従って色々なデータがビット列 (0 と 1 の列) で表されている。ここで、各々の記憶領域に記録されたビット列がどういう内部表現方式に従っているかは、ビット列自身の中に明示的な情報として含まれている訳ではなく、その記憶領域を使う側 (プログラム側) が決めることであった。

従って、一般に、プログラムはその中で使用される各々の変数や配列の中のデータがどういう内部表現方式に従うかの情報を含む。C 言語でこれらの情報を明示しているのは変数や配列の宣言で、

`int a;`　と宣言すると 例えば 32 ビット, 負数は 2 の補数で整数を表す方式が、
`float a;`　と宣言すると 例えば 32 ビット,IEEE 規格 754 で実数を表す方式が、
`double a;` と宣言すると 例えば 64 ビット,IEEE 規格 754 で実数を表す方式が、

変数 `a` の計算機内部の表現方式として想定される。これらの、計算機内部のデータ表現方式に対応する `int, float, double, ...` といったものを、プログラム上では**データ型** (data type) と呼ぶ。プログラミング言語によっては複数のデータをまとめて 1 つの変数として扱う仕組みも用意されているが、この様な変数の内部構成もやはりプログラム内に宣言されていて、計算機内部のデータ表現の方式に対応するので、データ型と考える。

C 言語におけるデータ型は次のように分類できる。ここで、[.....] は省略可能である部分を表す。

- 基本型
  - 算術型
    - 整数型
      - `char` 型…8 ビットの整数を表せる。
        `char`, `signed char`, `unsigned char`
      - `int` 型 …[`signed`] `int`, `unsigned` [`int`],
        [`signed`] `short` [`int`], `unsigned short` [`int`],
        [`signed`] `long` [`int`], `unsigned long` [`int`]
      - 列挙型 …実体は `int` の部分データ型。
    - 浮動小数点数型
      - `float`
      - `double`
      - `long double`
  - `void` 型 … 関数値や仮引数が無いことを明示したい時などに使う。
- 派生型
  - 配列　…同じデータ型の記憶領域の並び。
  - 関数　…例えば (`char (*)(int,double)`) は、`int` 型と `double` 型のデータを引数として受け取り `char` 型の値を返す関数のデータ型。
  - ポインタ…例えば (`int *`) は `int` 型データ領域を参照するためのデータ (≈ 番地) の型。
  - 構造体　…様々の型のデータを組み合わせて 1 つのデータ型として扱う。
  - 共用体　…1 つのデータ領域を複数のデータ型で共用する。

以下、この節では基本的なデータ型である整数型, 浮動小数点数型についてまとめ、複数の (同じ型とは限らない) データを組み合わせて 1 つのデータとして扱う仕組み (構造体, 共用体)、新しく構成したデータ型に名前を付ける方法を紹介する。また、上の分類の示す様に C 言語では文字を表すためのデータ型が特別に設けられている訳ではないので、C 言語における文字や文字列の扱いについて簡単に触れておく。

11.1 整数型： char, short, int, long, unsigned

- 整数値を表すためのデータ型としては `char`, `short`, `int`, `long` (および、各々を `unsigned` にしたもの) が用意されているが、ANSI 規格で定められているのは次のことだけである。 (あとはコンピュータ／処理系に依存。)

  `char` のビット数 = 8 ビット
  `long` のビット数 ≥ `int` のビット数
  ≥ `short` のビット数
  ≥ 16 ビット
  `long` のビット数 ≥ 32 ビット

- 標準ヘッダファイル `<limits.h>` の中に、扱える最大整数値などの情報が入っている。

マクロ名		意味
CHAR_BIT	···	`char` のビット数
INT_MIN	···	`int` の最小値
INT_MAX	···	`int` の最大値
LONG_MIN	···	`long` の最小値
LONG_MAX	···	`long` の最大値
.....		

- 整数値を表すための最も標準的なデータ型は `int` で、普通 1 ワードに格納される。

> ワード：
> コンピュータ／CPU が一度に処理するデータの単位のこと。昔のパソコンでは 1 ワード=16 ビットだったが、今はどれも 1 ワード=32 ビット以上。最近では 1 ワード=64 ビット もある。

□演習 11.1 (`<limits.h>`の中身) `/usr/include/limits.h` の中身を覗いて実習室の計算機で扱える最大整数値がどうなっているか調べてみて下さい。

表せる範囲：

- 8 ビット、16 ビット、32 ビットで表せる最大整数、最小整数は各々次の通り。

ビット数		表せる最小整数	表せる最大整数
signed	8	−128	127
	16	−32768	32767
	32	−2147483648	2147483647
unsigned	8	0	255
	16	0	65535
	32	0	4294967295

- 普通の C 言語処理系だと**オーバーフローのチェックはしてくれない。**
  $\Longrightarrow$ C 言語では、オーバーフローしない様にするのはプログラマの責任。

例えば：

```
[motoki@x205a]$ nl datatype-int-overflow-Kelley.c Enter
     1 #include <stdio.h>
     2 #define  BIG  2000000000

     3 int main(void)
     4 {
     5   int  a, b=BIG, c=BIG;

     6   a = b + c;
     7   printf("a=%d  b=%d  c=%d\n", a, b, c);
     8   return 0;
     9 }
[motoki@x205a]$ gcc datatype-int-overflow-Kelley.c Enter
[motoki@x205a]$ ./a.out Enter
a=-294967296  b=2000000000  c=2000000000
[motoki@x205a]$
```

整数定数：

- 普通の整数定数は `int`, `long`, または `unsigned long` 型 のデータとして扱われる。
  (表せる最小の型が選ばれる。)
- 整数定数の型を `long`, `unsigned`, ... などに指定することができる。例えば、

`37u`,	`37U`	は	`unsigned` 型
`37l`,	`37L`	は	`long` 型
`37ul`,	`37UL`	は	`unsigned long` 型

- **8 進整数を 10 進整数と混同しないこと。**例えば、

456	は	10 進定数、
0456	は	8 進定数 (10 進で $4\times8^2+5\times8^1+6\times8^0=302$ に相当)
0x456	は	16 進定数 (10 進で $4\times16^2+5\times16^1+6\times16^0=1110$ に相当)
0xaBc	は	16 進定数 (10 進で $10\times16^2+11\times16^1+12\times16^0=2748$ に相当)

整数の内部表現方式：

- `unsigned` の場合の整数データの記憶方式は全て同じ。すなわち、長さが n のビット列

  $$b_{n-1}b_{n-2}b_{n-3}\cdots\cdots b_2b_1b_0$$

  が符号なし整数を表すと見た場合、その値は

  $$\sum_{i=0}^{n-1} b_i\times2^i$$

  である。
- `signed` の場合の整数データの記憶方式は、ほぼ全て同じ。すなわち、普通の計算機の場合、長さが n のビット列

  $$b_{n-1}b_{n-2}b_{n-3}\cdots\cdots b_2b_1b_0$$

  が符号付き整数を表すと見た場合、その値は

  $$-b_{n-1}\times2^{n-1}+\sum_{i=0}^{n-2} b_i\times2^i$$

  である。

□**演習 11.2 (10 進 →2 進変換)** 2147483647 以下の非負整数を入力として受け取り、その値を 2 進数として表示する C プログラムを作成せよ。

□**演習 11.3 (32bit での整数表現)** 長さが 32 の 0 と 1 の数字列

$$b_{31}b_{30}b_{29}\cdots b_4b_3b_2b_1b_0$$

を入力して、

① このビット列を unsigned 型 (すなわち unsigned int 型) データと見た時に表す (非負) 整数値 $\sum_{i=0}^{31} b_i \times 2^i$、 および

② このビット列を int 型 (すなわち signed int 型) データと見た時に表す整数値 $-b_{31}\times 2^{31} + \sum_{i=0}^{30} b_i \times 2^i$

を出力する C プログラムを作成せよ。

11.2 浮動小数点数型

浮動小数点数型：

- 精度の保証された広範囲の実数値を表すためのもの。 $\Longrightarrow$ 誤差に注意。
- `float` (単精度),`double` (倍精度;**C 言語では標準**), `long double` (4 倍精度) の 3 つが用意されている。
- データ領域の大きさはコンピュータに依存している。

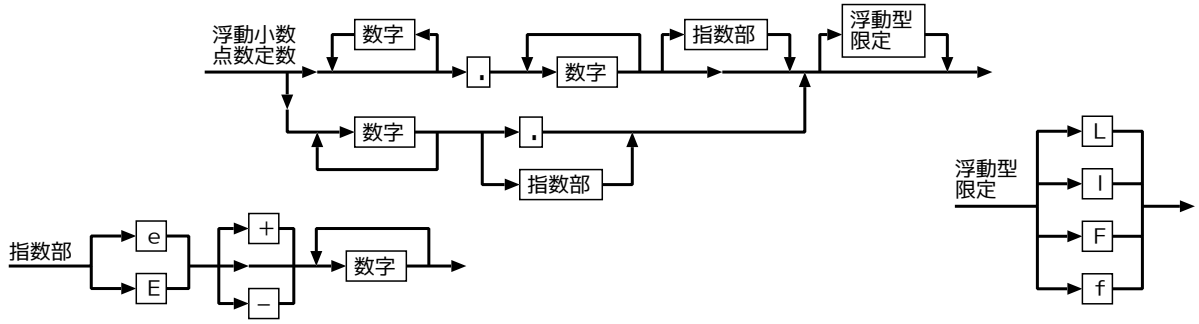
  `float` の精度 $\leq$ `double` の精度 $\leq$ `long double` の精度
- 標準ヘッダファイル `<float.h>` の中に、扱える最大の浮動小数点数などの情報が入っている。

マクロ名		意味
FLT_MAX	···	最大の `float` 型浮動小数点数
DBL_MAX	···	最大の `double` 型浮動小数点数
.....		

□**演習 11.4 (<float.h>の中身)** `/usr/include/float.h` の中身を覗いて実習室の計算機で扱える最大の浮動小数点数がどうなっているか調べてみて下さい。

浮動小数点定数：

- `123.4`, `123.`, `.4`, `123.4e5`, `.4E+5`, `123e-5`, ... といった書き方が出来る。 これらは `double` 型の定数になる。
- 定数を `float` 型にしたければ、最後に `f` または `F` という接尾語を付ける。 例えば、`123.4f`, `.4E+5F`, ... 。
- 定数を `long double` 型にしたければ、最後に `l` または `L` という接尾語を付ける。 例えば、`123.4l`, `.4E+5L`, ... 。



精度と指数部の表現可能な範囲：

- ごく普通の計算機の場合、`float` 型, `double` 型データは、各々4 バイト, 8 バイトの領域を占め、10 進で各々約 6 桁, 約 15 桁の精度を持つ。 また、指数部の表現可能な範囲は、およそ、各々 -38〜$+38, -308$〜$+308$ となる。 [指数部の表現可能な範囲は計算機のアーキテクチャに大きく依存する。]

IEEE 規格 754：

- 単精度、倍精度、4 倍精度における指数部、仮数部のビット数等は次の様に定められている。

	符号部	指数部	仮数部	全部で
単精度	1 ビット	8 ビット	23 ビット (10 進で 6〜7 桁)	32 ビット
倍精度	1 ビット	11 ビット	52 ビット (10 進で 15〜16 桁)	64 ビット
4 倍精度	1 ビット	15 ビット	112 ビット (10 進で 33〜34 桁)	128 ビット

- 単精度の場合、32 ビットの列

$$\underbrace{s}_{\text{符号部}}\underbrace{e_7e_6\cdots e_1e_0}_{\text{指数部}}\underbrace{d_1d_2\cdots d_{22}d_{23}}_{\text{仮数部}}$$

によって、

$$\begin{cases} (-1)^s\times(1+M)\times 2^E & \text{if } -127 < E < 128 \\ (-1)^s\times M\times 2^{E+1} & \text{if } E = -127 \\ \texttt{Inf}(\text{無限大}) & \text{if } E = 128, M = 0 \\ \texttt{NaN}(\text{非数,Not a Number}) & \text{if } E = 128, M\neq 0 \end{cases}$$

という実数を表す。但し、

$$M = \sum_{i=1}^{23} d_i\times 2^{-i}$$
$$E = \sum_{i=0}^{7} e_i\times 2^i - 127$$

□演習 11.5 (IEEE 規格 754 による実数表現) 長さが 32 の `0` と `1` の列

$$se_7e_6\cdots e_1e_0d_1d_2d_3\cdots d_{23}$$

を入力して、このビット列の表す実数値 (実数表現の仕方は IEEE 規格 754 に従うものと

仮定する)

$$\begin{cases} (-1)^s \times (1+M) \times 2^E & \text{if } -127 < E < 128 \\ (-1)^s \times M \times 2^{E+1} & \text{if } E = -127 \\ \texttt{Inf}(\text{無限大}) & \text{if } E = 128, M = 0 \\ \texttt{NaN}(\text{非数, Not a Number}) & \text{if } E = 128, M \neq 0 \end{cases}$$

ここで、$M = \sum_{i=1}^{23} d_i \times 2^{-i}, E = \sum_{i=0}^{7} e_i \times 2^i - 127$

を出力する C プログラムを作成せよ。[**Hint.** $2^{\sum_{i=0}^{7} e_i \times 2^i - 127} = 2^{e_7 \times 2^7 + \sum_{i=0}^{6} (e_i - 1) \times 2^i} = ((\cdots(((2^{e_7})^2/2^{(1-e_6)})^2/2^{(1-e_5)})^2 \cdots)^2/2^{(1-e_1)})^2/2^{(1-e_0)}$]

11.3 C 言語における文字の扱い — 整数型 char —

- C 言語では、**文字は小さな整数として扱われる**。例えば a という文字を表したい時にはプログラムの中では**文字定数**'a' を用いるが、これは内部では 97 という整数として扱われる。各々の文字の番号は ASCII コードに基づいて次の表のように決められている。

印字可能文字の場合					
文字定数	'a'	'b'	'c'	.....	'z'
(8 進表記)	'\141'	'\142'	'\143'	.....	'\172'
(16 進表記)	'\x61'	'\x62'	'\x63'	.....	'\x7A'
文字の番号	97	98	99	.....	122
文字定数	'A'	'B'	'C'	.....	'Z'
文字の番号	65	66	67	.....	90
文字定数	'0'	'1'	'2'	.....	'9'
文字の番号	48	49	50	.....	57
文字定数	'&'	'*'	'+'	.....	
文字の番号	38	42	43	.....	

機能文字の場合			
文字定数	'\0' (ヌル文字)	'\a' (警告)	'\b' (後退)
(8 進表記)	'\000'	'\007'	'\010'
(16 進表記)	'\x00'	'\x07'	'\x08'
文字の番号	0	7	8
文字定数	'\t' (水平タブ)	'\n' (改行)	'\v' (垂直タブ)
文字の番号	9	10	11
文字定数	'\f' (紙送り)	'\r' (復帰)	'\"' (2 重引用符)
文字の番号	12	13	34
文字定数	'\'' (引用符)	'\\' (バックスラッシュ)	
文字の番号	39	92	

- 文字の番号 (小さな整数) を記憶するためのデータ型として char 型が用意されている。
- char 型は次のいずれかと同等。(コンパイラ次第)

$$\begin{cases}\texttt{signed char}\\ \texttt{unsigned char}\end{cases}$$

- **文字定数'a', 'b', ... は char 型ではなく実は int 型。**
- **char 型変数は 1 バイトの領域を占める。**例えば文字定数 'a' と同じ値をもつ char 型の領域は計算機内部で次の様に表される。

7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	1

$\Longrightarrow$ 文字 'a' の番号$= 2^6 + 2^5 + 2^0$
$= 97$

一般に、ビット列
$b_7b_6b_5b_4b_3b_2b_1b_0$
は、unsigned char 型データとしては
$\sum_{i=0}^{7} b_i \times 2^i$
という値を持ち、signed char 型データとしては
$-b_7 \times 2^7 + \sum_{i=0}^{6} b_i \times 2^i$
という値を持つ。 (負数は 2 の補数で表す。)

$\Longrightarrow$ 表せる整数の範囲は、
unsigned char 型なら 0〜255、
signed char 型なら -128〜127。

- C 言語における char 型は整数型の一種に他ならないので、記憶した整数値を数字列で出力することは勿論出来る。ただ、char 型変数に文字を記憶させたい場合のために、入力した文字からその番号を割り出したり、記憶した整数番号の文字を出力したり出来るようになっている。 [実は、こちらの方がデータ変換が無くて単純。]

$\Longrightarrow$ ◇ int 型 (または short 型, long 型) でも文字を表せる。
◇ char 型変数は小さな整数値を保持するためにも使える。

例 11.6 (C 言語における文字の扱い) 次のプログラムの様に、printf の出力書式において %d を指定して整数値を 10 進表記で出力することも、%c を指定して与えられた番号の文字を出力することも出来る。

```
[motoki@x205a]$ nl datatype-char-Kelley.c  Enter
     1 #include <stdio.h>

     2 int main(void)
     3 {
     4   char c='a';

     5   printf("%c %c %c\n", c, c+1, c+2);
     6   printf("%d %d %d\n", c, c+1, c+2);
     7   return 0;
     8 }
```

```
[motoki@x205a]$ gcc datatype-char-Kelley.c Enter
[motoki@x205a]$ ./a.out Enter
a b c
97 98 99
[motoki@x205a]$
```

11.4 C 言語における文字列の扱い — char 型配列 —

C 言語における文字列の表し方について：

- char 型の配列を使う。
- 文字列の終わりの印として文字列の最後に**ヌル文字**'\0' を置く。
  $\Longrightarrow$ (配列の大きさ) $\geq$ (文字列の長さ) + 1 でなければならない。

0	1	2	3	4	5	6	
's'	't'	'r'	'i'	'n'	'g'	'\0'	···

- 文字列を 2 重引用符で囲めば**文字列定数**になる。
- char 型配列で文字列を表す場合は、初期設定を次の様に行うことが出来る。

```
char s[]="string";
```

(`char s[]={'s','t','r','i','n','g','\0'};` と同等。)

- 文字列操作のライブラリ関数が多数用意されている。
  $\Longrightarrow$ この講義ノート 10.8 節等を御覧下さい。

補足：

◇ 文字列定数の値はその文字列が確保されている領域へのポインタになっている。

$\Longrightarrow$ `char *p="abc";` という宣言も出来る。
`"abc"[1]` や `*("abc"+2)` は文法的に正しい式になる。

◇ 2 つの宣言の違い:

`char *p="abc";` ··· p という名前のポインタのために記憶領域が確保される。
$\Longrightarrow$ 計算している内に p が"abc"という文字列を指さなくなることもある。

`char a[]="abc";` ··· a は定数ポインタ。

注意：

◇ ヌル文字 '\0' を出力しないよう気を付けること。 [印字可能文字ではなく機能文字であるため。]

□**演習 11.7 (文字列定数)** `"abc"[1]` や `*("abc"+2)` がどういう値を持つか考えよ。

例題 11.8 (文字列操作のライブラリ関数) 長さが 10 以下の英単語 w と 1 行が 80 文字以下の英文章を読み込み、英文章中に現れる単語 w を全て大文字に変換して得られる文章を出力する C プログラムを作成せよ。

(考え方) 最初に英単語 w を読み込むことにすれば、あとは

$$\left\{\begin{array}{l}\text{① 英文章の次の 1 行の読み込み,}\\ \text{② 読み込んだ 1 行の中に現れる単語 } w \text{ を全て大文字に変換,}\\ \text{③ 変換後の 1 行の出力}\end{array}\right.$$

という作業を繰り返すだけである。ここで、

- 英文章の次の 1 行を読み込むためには、`fgets()` というライブラリ関数を利用できる。 ($\Longrightarrow$ 10.8 節を参照; `scanf("%s", ...)` としたのでは、空白や改行コード等で区切られた小さな文字列しか取り出せない。)
- 文字列の中から小さな文字列パターンを探索するためには、`strstr()` という文字列操作のライブラリ関数を利用できる。 ($\Longrightarrow$ 10.8 節を参照)
- 英単語 w の長さを測るためには、`strlen()` という文字列操作のライブラリ関数を利用できる。 ($\Longrightarrow$ 10.8 節を参照)
- 英字を大文字に変換するためには、`toupper()` という文字種類変換のライブラリ関数を利用できる。 ($\Longrightarrow$ 10.8 節を参照)

(プログラミング) 英単語 w を保持するために `word[]` という名前の char 型配列を、1 行分の文字列を保持するために `line[]` という名前の char 型配列を、そして読み込んだ 1 行分の文字列を前から順に走査するために `remaining_seq` という名前のポインタ変数を用意した。そして、入力する英文章と出力する英文章をきれいに分離するために、入力する英単語と英文章を入れたデータファイルを別に作り、そのファイルからのデータストリームが入力リダイレクションにより標準入力に流れることを想定して、プログラムを構成した。この C プログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl toupper-some-words-in-sentences.c Enter
     1 /* 長さが10以下の英単語 w と1行が80文字以下の英文章を */
     2 /* 読み込み、英文章中に現れる単語wを全て大文字に変換  */
     3 /* して得られる文章を出力するCプログラム              */
     4 /*              (入力リダイレクションにより          */
     5 /*               ファイルから入力することを想定する。) */

     6 #include <stdio.h>
     7 #include <string.h>
     8 #include <ctype.h>

     9 int main(void)
    10 {
    11   char word[11], WORD[11], line[82], *remaining_seq;
    12   int  word_length, i;       /* 英文章の1行は最長で    */
    13                              /* 80文字+改行コード+'\0' */

    14   scanf("%10s", word);     /* 大文字にする英単語を入力 */

    15   word_length=strlen(word);
```

```
 16   for (i=0; i<word_length; i++)
 17     WORD[i]=toupper(word[i]);
 18   WORD[word_length]='\0';

 19   printf("単語 %s を大文字に換えて得られる文章：\n", word);
 20   while (fgets(line, 82, stdin)!=NULL) {  /*次の 1 行を読む*/
 21     remaining_seq = line;
 22     while ((remaining_seq=strstr(remaining_seq, word))!=NULL) {
 23       for (i=0; i<word_length; i++)
 24         remaining_seq[i]=WORD[i];
 25       remaining_seq += word_length;
 26     }
 27     printf("   %s", line);
 28   }
 29   return 0;
 30 }
[motoki@x205a]$ gcc toupper-some-words-in-sentences.c Enter
[motoki@x205a]$ cat toupper-some-words-in-sentences.data Enter
language
------
Why C?
------
C is a small language.  And small is beautiful in programming.
C has fewer keywords than Pascal, where they are known as
reserved words, yet it is arguably the more powerful language.
C gets its power by carefully including the right control
structures and data types and allowing their uses to be nearly
unrestricted where meaningfully used.  The language is readily
learned as a consequence of its functional minimality.  C is
the native language of UNIX, and UNIX is a major interactive
operating system on workstation, servers, and mainframes.
Also, C is the standard development language for personal
computers.  Much of MS-DOS and OS/2 is written in C.  Many
windowing packages, database programs, graphics libraries, and
other large-application packages are written in C.
(A.Kelly&I.Pohl, "A Book on C" 4th ed., Addison-Wesley, 1998.)
[motoki@x205a]$ ./a.out <toupper-some-words-in-sentences.data Enter
単語 language を大文字に換えて得られる文章：

   ------
   Why C?
   ------
   C is a small LANGUAGE.  And small is beautiful in programming.
```

```
    C has fewer keywords than Pascal, where they are known as
    reserved words, yet it is arguably the more powerful LANGUAGE.
    C gets its power by carefully including the right control
    structures and data types and allowing their uses to be nearly
    unrestricted where meaningfully used.  The LANGUAGE is readily
    learned as a consequence of its functional minimality.  C is
    the native LANGUAGE of UNIX, and UNIX is a major interactive
    operating system on workstation, servers, and mainframes.
    Also, C is the standard development LANGUAGE for personal
    computers.  Much of MS-DOS and OS/2 is written in C.  Many
    windowing packages, database programs, graphics libraries, and
    other large-application packages are written in C.
    (A.Kelly&I.Pohl, "A Book on C" 4th ed., Addison-Wesley, 1998.)
[motoki@x205a]$
```

ここで、

- プログラム 7 行目 の include 行は、15 行目, 22 行目で文字列操作のライブラリ関数 strlen(), strstr() のコンパイルを間違いなく行うために入れてある。
- プログラム 8 行目 の include 行は、17 行目文字種類変換のライブラリ関数 toupper() のコンパイルを間違いなく行うために入れてある。
- プログラムの 11 行目 では 1 行分の文字列を保持するために 長さが 80 ではなく 82 の char 型配列 line[] が確保されているが、これは 1 行入力の関数 fgets() を使うと行の最後の改行コードと、文字列の最後に来るべきヌル文字 '\0' が char 型配列の中に格納されるからである。
- プログラム 14 行目 の入力書式中の%10s は、空白 (類) を含まない文字列を標準入力から読み込む、但し空白 (類) なしで 11 文字以上の文字が続いている場合は最初の 10 文字だけを読み込む、ということを表す。読み込まれた文字列の次にはヌル文字 '\0' が付加される。
- プログラム 15 行目 の strlen() は、引数で指定された文字列の長さを返すライブラリ関数である。
- プログラムの 16〜18 行目 では、読み込んだ英単語 word[] を大文字に変換した文字列を WORD[] という名前の char 型配列上に構成している。
- プログラム 17 行目 の toupper() は、引数で指定された文字の番号を大文字の番号に変換して返すライブラリ関数である。
- プログラム 20 行目 の fgets(line, 82, stdin) は、標準入力 stdin から、改行コード又はファイルの終りまでの文字の並び (但し長くなっても $82-1=81$ 文字で打ち切り) を読み込み、最後に空文字 \0 を付けて char 型配列 line に格納するライブラリ関数である。通常は line の値が関数値として返されるが、ファイル終了又はエラー発生時には NULL が返される。
- プログラムの 21〜26 行目 は、読み込んだ 1 行の中に現れる単語 w を全て大文字に変換している部分である。
- プログラム 22 行目 の strstr(remaining_seq, word) は、word[] の中に入っている

文字列パターンを remaining_seq[] の中の文字列の最初から探すライブラリ関数である。見つかればその最初の部分文字列の先頭位置へのポインタを返し、見つからなければ空ポインタ NULL を返す。

- プログラム 22 行目 の remaining_seq=strstr(...) は代入 式 で、代入される値がこの式の値となる。
- プログラムの 27 行目 の出力書式の中には改行コード \n が含まれていないが、これは 1 行分の文字列を保持している char 型配列 line[] の最後に (多分) 改行コードが含まれるためである。

不揃い配列： 文字列自体が (1 次元) 配列で表されるので、複数の文字列を配列に保持しようとすると 2 次元の char 型配列が必要になる。 例えば、13 個の "Illegal month", "January", "February", "March", ..., "December" という文字列を系統的に保持したい場合は、通常、2 次元 char 型配列を用いて次のように宣言する。

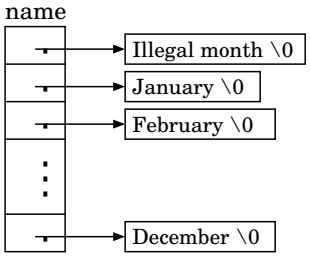
```
char name[][14]={                          /* name[k]           */
        "Illegal month",                   /* = k 番目の月の名前 */
        "January", "February", "March",
        "April", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
      };
```

$\longrightarrow j$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	'I'	'l'	'l'	'e'	'g'	'a'	'l'	'␣'	'm'	'o'	'n'	't'	'h'	'\0'
1	'J'	'a'	'n'	'u'	'a'	'r'	'y'	'\0'						
2	'F'	'e'	'b'	'r'	'u'	'a'	'r'	'y'	'\0'					
3														
	.....													
10														
↓ 11	'N'	'o'	'v'	'e'	'm'	'b'	'e'	'r'	'\0'					
i 12	'D'	'e'	'c'	'e'	'm'	'b'	'e'	'r'	'\0'					

しかし、この宣言を次のように書き直すとメモリが節約できる。こういった配列を**不揃い配列**という。

```
char *name[]={                             /* name[k]           */
        "Illegal month",                   /* = k 番目の月の名前 */
        "January", "February", "March",
        "April", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
      };
```



11.5 typedefによる新しいデータ型の定義

C 言語では、データの使い方に合わせてデータ型に分かり易い名前を付けることが出来る。

例 11.9 (新しいデータ型の定義) 宣言

```
typedef  int  CentiMeter, Meter, KiroMeter;
```

が行なわれていれば、以降ではデータ型 int の別名として CentiMeter, Meter, KiroMeter という名前を用いて、プログラム中で

```
CentiMeter height;
KiroMeter  distance;
```

という風な書き方も出来る。

例 11.10 (新しいデータ型の定義) 宣言

```
typedef  char  *String;
```

が行なわれていれば、以降では

```
String  s = "abc";
```

という宣言と

```
char  *s = "abc";
```

注釈：
「char *String;」の中の「String」を変数名 s で置き換え、初期設定部をつなげた。

という宣言は同等になる。 従って、上の typedef 宣言は「char 型へのポインタ」の総称として String というデータ型名を使うことを宣言している。

例 11.11 宣言

```
typedef  float  Vector[10];
```

が行なわれていれば、以降では

```
Vector  v1, v2;
```

という宣言と

```
float  v1[10], v2[10];
```

という宣言は同等になる。 従って、上の typedef 宣言は「大きさ 10 の float 型の配列」の総称として Vector というデータ型名を使うことを宣言している。

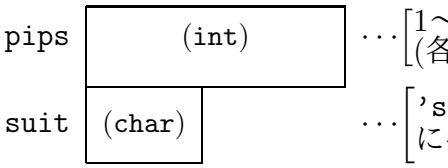
typedef 宣言の利点：

- 変数宣言をコンパクトに行なうことが出来る。
- 使い方に合わせてうまくデータ型に名前を付ければ、プログラムが読み易くなる。
- プログラムの移植性向上に役立つ。

11.6 構造体

構造体の定義： Pascal や Fortran90 等の他の (命令型) プログラミング言語と同様に、C 言語においても、関連するデータを 1 つにまとめて扱うことが出来る。 C 言語の場合は、「関連するデータを 1 つにまとめたもの」を**構造体**と呼ぶ。また、構造体の構成要素を**メンバ**、メンバを区別するための名前を**メンバ名**という。

例 11.12 (構造体の宣言; トランプのカード) トランプのカードを識別するためのデータ構造



…[1〜13 の間の整数をこの記憶域に保持する。(各々 A,2,3, ..., 9,10,J,Q,K を表す。)

…['s', 'h', 'd', 'c' のいずれかをこの記憶域に保持する。(各々 ♠, ♡, ♢, ♣ を表す。)

を持った変数 c1, c2 は次のように宣言することが出来る。

(方法 1) 直接定義する。

```
struct {
   int pips;
   char suit;
}c1, c2;
```

} 毎回長いのを書かないといけない。

(方法 2) まず構造体の形に名前 (**タグ**という) を付けてから、...。

```
struct card {
   int pips;
   char suit;
};
struct card c1, c2;
```

} 「struct card」をデータ型の名前として使うことが出来る。

(方法 3) まず構造体の形に名前付け、さらに、それに新しいデータ型としての名前を付けてから、...。

```
struct card {
   int pips;
   char suit;
};
typedef struct card Card;
Card c1, c2;
```

} 「struct card」と「Card」をデータ型の名前として使うことが出来る。

(方法 4) 構造体の形に新しいデータ型としての名前を付けてから、...。

```
typedef struct {
   int pips;
   char suit;
}Card;
Card c1, c2;
```

「Card」をデータ型の名前として使うことが出来る。

構造体はいくらでも複雑に出来る。 例えば、

- 配列や構造体をメンバに出来る。
- 構造体の配列も許される。

構造体メンバへのアクセス：

- 構造体メンバへのアクセスの仕方は次の 2 つ。

  構造体変数 . メンバ名

  構造体へのポインタ -> メンバ名 ... 次のものと同等。 (* 構造体へのポインタ) . メンバ名

- 計算機内部では . も -> も演算子 (**メンバアクセス演算子**という) として扱われる。

例 11.13 (構造体要素へのアクセス; トランプのカード) 先の例 11.12 の様に変数 c1, c2 が宣言されていた場合、例えば

```
c1.pips = 3;
c1.suit = 's';
c2 = c1;
```

により、スペードの 3 を表すコードが 2 つの変数 c1, c2 にセットされる。

例 11.14 (配列を構成要素とする構造体) 構造体

```
struct person {
   char id[9];
   char name[40];
   int  age;
};
```

に関して、次の様なアクセスが可能である。

構造体変数 .name[5]

↑ こちらの方が強い。 (. も [] も演算子で、共に最高の優先順位を持つが、この中では左側のものが優先される。)

例題 11.15 (学生データの整理) 100 人以下の学生について
学籍番号 (5 桁の英数字列), 数学の得点 (100 点満点)
を次々に読み込んで、各人のデータを学籍番号の順 (辞書順) に出力する C プログラムを作成せよ。

(考え方) 5 桁の英数字列 (学籍番号) を文字列として保持するには '\0' も含めて長さが 6 の char 型配列があれば良く、また 0〜100 の整数 (数学の得点) を保持するには 8 ビッ

ト以上の整数領域があれば良い。 従って、学生一人分のデータは例えば次の様な構造体で表すことが出来る。

id	(長さ 6 の char 型配列)	···学籍番号 (5 桁の英数字列) を文字列として保持
math	(int 型領域)	···数学の得点 (0〜100 の整数) を保持

学生の人数が 100 人以下とあるので、この構造体領域が 100 個連なった配列を用意すれば学生全員のデータを保持することが出来る。

また、並べ換えに関しては、例えば例題 9.6 で示したバブルソートアルゴリズムを用いることが出来る。 2 つの文字列の大小関係 (辞書順かどうか) を調べるためには、文字列比較のライブラリ関数 `strcmp()` を用いれば良い。($\Longrightarrow$ 10.8 節を参照)

(プログラミング) 学生一人分のデータの型として `Student` という名前を用い、学生全員のデータを保持するために `student` という名前の配列を用意した。(大文字で始まるのがデータ型名、小文字で始まるのが配列名。) そして、データ入力後はバブルソートを使って学籍番号が辞書順になるように学生データを並べ換える、というプログラムを構成した。この C プログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl sort-student-struct-data.c Enter
```

```
 1 /* 100 人以下の学生について                                    */
 2 /*     学籍番号 (5 桁の英数字列), 数学の得点 (100 点満点)      */
 3 /* を次々に読み込んで、各人のデータを学籍番号の順 (辞書順) */
 4 /* に出力する C プログラム                                     */

 5 #include <stdio.h>
 6 #include <string.h>

 7 typedef struct {
 8     char id[6];
 9     int  math;
10 } Student;

11 int main(void)
12 {
13   Student student[101], temp;
14   int     num, scanf_val, k, i;

15   /* データ入力 */
16   for (num=0; num<101; ) {
17     scanf_val = scanf("%5s %d", student[num].id, &student[num].math);
18     if (scanf_val == 2)
19       num++;
20     else if (scanf_val == EOF)
21       break;
```

```
    22     else {
    23       printf("Warning: Illegal data appears. (%d-th data)\n", num);
    24       break;
    25     }
    26   }

    27   if (num == 101) {
    28     printf("Warning: There are 101 or more students.\n"
    29            "         ==> We process first 101 students.\n");
    30   }

    31   /* 辞書順に整列 */
    32   for (k=0; k<num-1; k++) {
    33     for (i=num-1; i>k; i--)
    34       if (strcmp(student[i-1].id, student[i].id) > 0) {
    35         temp         = student[i-1]; /*student[i-1] と student[i]*/
    36         student[i-1] = student[i];   /*が辞書順でないなら交換  */
    37         student[i]   = temp;
    38       }
    39   }

    40   /* 整列後のデータを出力 */
    41   printf(" id.   math.\n"
    42          "-----  -----\n");
    43   for (k=0; k<num; k++)
    44     printf("%5s   %3d\n", student[k].id, student[k].math);
    45   return 0;
    46 }
[motoki@x205a]$ gcc sort-student-struct-data.c Enter
[motoki@x205a]$ cat sort-student-struct-data.data Enter
T8100  100
T8050   78
T8022   80
T8011   50
T8064   35
T8037   90
T8001   72
T8005    0
T8046   68
T8055   46
[motoki@x205a]$ ./a.out < sort-student-struct-data.data Enter
 id.   math.
-----  -----
```

```
T8001     72
T8005      0
T8011     50
T8022     80
T8037     90
T8046     68
T8050     78
T8055     46
T8064     35
T8100    100
[motoki@x205a]$
```

ここで、

- プログラム 6 行目 の include 行は、35 行目で文字列操作のライブラリ関数 strcmp() のコンパイルを間違いなく行うために入れてある。
- プログラム 7〜10 行目 では、学生一人分のデータを表すための構造体を定義し、それに Student というデータ型名を付けている。
- プログラム 13 行目 では学生全員のデータを保持するために 大きさが 100 ではなく 101 の配列 student[] が宣言されているが、これは学生数が 100 人を越えた場合をうまく処理するためである。学生データが 101 人分以上あった場合は、一旦配列の 101 番目の要素 student[100] に読み込み、その後の 27〜30 行目 でエラー処理がなされる。
- プログラム 16〜26 行目 は、データが無くなる (20 行目 で検出) か、101 人分のデータを読み込む (16 行目 で検出) か、データ読み込みに失敗する (23 行目 で検出) まで、学生のデータを次々と配列 student[] に格納している部分である。
- プログラム 32〜39 行目 は、バブルソートを使って学籍番号が辞書順になるように学生データを並べ換えている部分である。
- プログラム 34 行目 の strcmp() は、引数で指定された 2 つの文字列 (student[i-1].id と student[i].id) を辞書式順序で比較する。その結果、第 1 引数の文字列が第 2 引数の文字列より小さければ (i.e. 辞書順で前に来れば) 負、等しければゼロ、大きければ正の値を返す。

□演習 11.16 (平面上の点の座標を構造体で表す) 平面上の 2 点の座標を読み込み、それら 2 点の距離を出力する C プログラムを作成せよ。 但し、プログラム作成においては、**平面上の点の座標を構造体で表し、それに Point というデータ型名を付ける** ものとする。

□演習 11.17 (試験の成績処理) 大勢の学生について

学籍番号 (5 桁の数字列), 英語, 数学, 国語の得点 (各 100 点満点)

を次々に読み込んで、各人の総得点を計算の上、①総得点の高い順に各人のデータを、さらに②各々の平均と標準偏差を次の形に出力する C プログラムを作成せよ。但し、ここでは学生の人数は 100 人以下で不定とする。

```
Id-No␣␣␣Eng␣␣Math␣␣␣Jap␣␣␣Total
-----␣␣----␣␣----␣␣----␣␣␣-----
```

```
90805␣␣␣␣83␣␣␣100␣␣␣␣84␣␣␣␣␣267
90808␣␣␣␣85␣␣␣␣80␣␣␣␣90␣␣␣␣␣255
90809␣␣␣␣74␣␣␣100␣␣␣␣65␣␣␣␣␣239
            ..............
90803␣␣␣␣44␣␣␣␣65␣␣␣␣51␣␣␣␣␣160
90806␣␣␣␣58␣␣␣␣30␣␣␣␣57␣␣␣␣␣145
-------------------------------
Ave␣␣␣␣72.5␣␣75.8␣␣68.1␣␣␣216.4
Dev␣␣␣␣15.3␣␣19.6␣␣12.4␣␣␣␣33.3
```

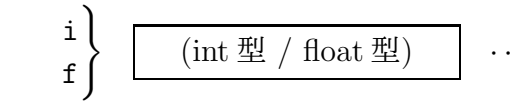
11.7 共用体

共用体：

- 色々な種類のデータを**選択的に** 1 つのデータ領域で表せる様にしたもの。
- 共用体定義や参照の構文は構造体の場合とほぼ同じ。(キーワードが `struct` から `union` になっただけ。)
- 共用体の中のデータを正しく解釈して使うのはプログラマの責任。

例題 11.18 (共用体) 実数型データを `float` 型で読み込み、そのビット列を `int` 型と見て 16 進表示する C プログラムを作成せよ。

(考え方) 1 つのデータ領域を `float` 型と見たり `int` 型と見たりする訳だから、このデータ領域を共用体として確保すれば良い。



(プログラミング) int 型データと float 型データを切替えて保持する領域に対して `Number` という名前のデータ型を導入した。そして、このデータ型の `num` という名前の変数領域を用意して、プログラムを構成した。この C プログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl union-int-or-float.c  Enter
     1 /*-------------------------------------------------*/
     2 /* 実数型データを float 型で読み込み、             */
     3 /* そのビット列を int 型と見て 16 進表示する C プログラム */
     4 /*-------------------------------------------------*/

     5 #include <stdio.h>

     6 typedef union {
     7   int   i;
```

```
 8   float f;
 9 }Number;

10 int main(void)
11 {
12   Number  num;

13   scanf("%f", &num.f);
14   printf("Float number %g and int number %d are\n"
15          " commonly represented by a bit sequence %#.8x.\n",
16          num.f, num.i, num.i);
17   return 0;
18 }
[motoki@x205a]$ gcc union-int-or-float.c Enter
[motoki@x205a]$ ./a.out Enter
1.0 Enter
Float number 1 and int number 1065353216 are
 commonly represented by a bit sequence 0x3f800000.
[motoki@x205a]$ ./a.out Enter
1e-38 Enter
Float number 1e-38 and int number 7136238 are
 commonly represented by a bit sequence 0x006ce3ee.
[motoki@x205a]$ ./a.out Enter
1e-45 Enter
Float number 1.4013e-45 and int number 1 are
 commonly represented by a bit sequence 0x00000001.
[motoki@x205a]$
```

この実行結果により、
同じビット列であっても、それがどういう内部表現方式に従っているかによって表されるデータが全く違うものになることが例示されている。 実際、最初の実行結果は、16 進で X'3f800000' というビット列を float 型データと見るとその値は 1.0 となり、int 型データと見るとその値は 1065353216 となることを言っている。

ここで、

- プログラム 6〜9 行目 が、共用体を定義してそれに Number というデータ型名を付けた部分である。
- プログラム 13 行目,16 行目 の num.f, num.i は各々共用体変数 num 内の f というメンバ, i というメンバ を表す。
- プログラム 15 行目 の %#.8x は整数データを 16 進表記で出力することを表す。 精度 =8 と指定されているので「出力すべき最小桁数」が 8 と解釈され、値部の数字列が 8 桁以上になる様に上位桁にゼロが埋められる。また、フラグ # が指定されているので出力の頭に 0x という文字列が付く。

12　ファイル入出力

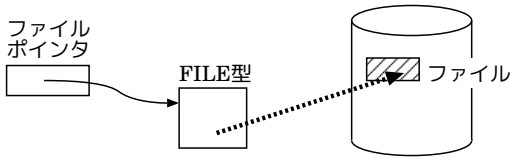
- ファイル入出力, fopen(), fclose()

12.1　ファイル入出力 — fopen() と fclose() —

C プログラムで操作するファイルは通常はテキストファイルで、前から順に処理される。例えばファイルからデータを入力する場合は、ファイルの先頭から順に 1 文字ずつ読んで行き、その上にある文字列を整数データや実数データ, 文字データ, 文字列データとして解読する。また、ファイルへデータ出力する場合は、出力文字列を次々と付け足して行くことによって、出力ファイルを前から順に構成する。従って、ファイル操作の間、ファイルのどの場所を現在処理しているかの情報を常にどこかに保持しておく必要があり、この情報も含めてファイル操作を快調に行うための情報を維持しておく必要がある。しかし、ディスク上のファイルに直接アクセスするという操作は、システム管理の問題と関わって来るので、一般ユーザには許されていない。

そこで、C 言語においては、

- プログラムからの要求に応じて、言語処理系/OS がこれらのファイル操作に必要/有用な情報を FILE というデータ型名の付いた構造体の中に詰め込み、
- プログラムの中でこの FILE 型構造体へのポインタ (i.e. 所在番地) を保持する、



そして

- プログラムの中では、FILE 型構造体へのポインタを明示することによって、間接的に操作目的のファイルを指定する、

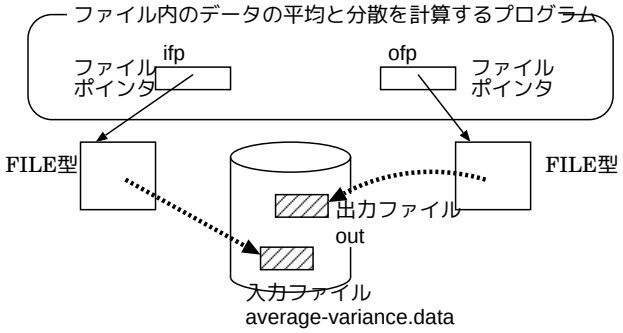
という仕組みが取られている。 この FILE 型構造体へのポインタのことを**ファイルポインタ**と言う。また、操作を始めたいファイルに関する FILE 型構造体を言語処理系/OS に作ってもらってファイル操作の準備を行うことを、**ファイルをオープンする**と言い、逆に操作の終ったファイルの FILE 型構造体の領域を解放してファイル操作の後始末を行うことを**ファイルをクローズする**と言う。

例題 12.1 (ファイル内のデータの平均と分散) 例題 9.1 と同じ様に、50 個の実数データ $x_0, x_1, x_2, \ldots, x_{49}$ を読み込み、それらの平均 μ と分散 V を定義式

$$\mu = \frac{1}{50}(x_0 + x_1 + x_2 + \cdots + x_{49})$$
$$V = \frac{1}{50}\sum_{i=0}^{49}(x_i - \mu)^2$$

に従って求めて出力する C プログラムを作成せよ。但し、ここでは入力データは average-variance.data という名前のファイルから読み込み、出力データは out というファイルに書き出すことにする。

(考え方) 入力ファイルに繋がるファイルポインタ, 出力ファイルに繋がるファイルポインタの領域が必要である。



これらのものが用意されていれば、基本的なデータ処理の流れは例題 9.1 と同じで良いので、次の 4 点に注意して例題 9.1 のプログラムに少し手を加えるだけである。(⟹10.8 節を参照)

- データ処理の前にライブラリ関数 `fopen()` を用いて 2 つのファイルをオープンする必要がある。
- データ処理の後にライブラリ関数 `fclose()` を用いて 2 つのファイルをクローズする必要がある。
- 指定されたファイルからデータを入力するので、`scanf()` じゃなく `fscanf()` を用いる。
- 指定されたファイルへデータを出力するので、`printf()` じゃなく `fprintf()` を用いる。

(プログラミング) 例題 9.1 で考えた配列や変数の他に、入力用, 出力用のファイルポインタを保持するために各々 `ifp`, `ofp` という名前の変数を用意して、プログラムを構成した。この C プログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl average-variance-io-through-files.c
     1 /* 50個の実数データ x0, x1, x2, ... , x49 の平均 mu と      */
     2 /* 分散 $V$ を定義式                                        */
     3 /*   mu = (x0+x1+x2+ ...  + x49)/50                        */
```

```
 4 /*    V = (x0-mu)^2 + (x1-mu)^2 + ... + (x49-mu)^2 / 50  */
 5 /* に従って求め、それらの値を出力するプログラム             */
 6 /* 但し、ここでは入力データは average-variance.data という */
 7 /* 名前のファイルから読み込み、出力データは out というファ */
 8 /* イルに書き出すことにする。                               */

 9 #include <stdio.h>
10 #include <stdlib.h>

11 int main(void)
12 {
13   int     i;
14   double  x[50], ave, var;
15   FILE    *ifp, *ofp;

16   if ((ifp=fopen("average-variance.data", "r")) == NULL) {
17     printf("ファイルをオープン出来ません: average-variance.data\n");
18     exit(1);
19   }
20   if ((ofp=fopen("out", "w")) == NULL) {
21     printf("ファイルをオープン出来ません: out\n");
22     exit(1);
23   }
24
25   ave = 0.0;
26   for (i=0; i<50; ++i) {
27     fscanf(ifp, "%lf", &x[i]);
28     ave += x[i];
29   }
30   ave /= 50.0;

31   var = 0.0;
32   for (i=0; i<50; ++i)
33     var += (x[i]-ave)*(x[i]-ave);
34   var /= 50.0;

35   fprintf(ofp, "\nInput data:\n");
36   for (i=0; i<50; i+=5)
37     fprintf(ofp, "%14.5e%14.5e%14.5e%14.5e%14.5e\n",
38             x[i], x[i+1], x[i+2], x[i+3], x[i+4]);
39   fprintf(ofp, "\nAverage  = %14.6g\n"
40           "Variance = %14.6g\n", ave, var);
```

```
    41   fclose(ifp);
    42   fclose(ofp);
    43   return 0;
    44 }
[motoki@x205a]$ gcc average-variance-io-through-files.c
[motoki@x205a]$ ./a.out
[motoki@x205a]$ cat out

Input data:
   1.00000e+00   1.00010e+00   1.00020e+00   1.00030e+00   1.00040e+00
   1.00050e+00   1.00060e+00   1.00070e+00   1.00080e+00   1.00090e+00
   1.00100e+00   1.00110e+00   1.00120e+00   1.00130e+00   1.00140e+00
   1.00150e+00   1.00160e+00   1.00170e+00   1.00180e+00   1.00190e+00
   1.00200e+00   1.00210e+00   1.00220e+00   1.00230e+00   1.00240e+00
   1.00250e+00   1.00260e+00   1.00270e+00   1.00280e+00   1.00290e+00
   1.00300e+00   1.00310e+00   1.00320e+00   1.00330e+00   1.00340e+00
   1.00350e+00   1.00360e+00   1.00370e+00   1.00380e+00   1.00390e+00
   1.00400e+00   1.00410e+00   1.00420e+00   1.00430e+00   1.00440e+00
   1.00450e+00   1.00460e+00   1.00470e+00   1.00480e+00   1.00490e+00

Average  =         1.00245
Variance =     2.0825e-06
[motoki@x205a]$
```

ここで、

- プログラム 15 行目 は、ifp と ofp がファイルポインタであることを宣言している。

  補足：
  FILE というデータ型は <stdio.h> の中で定義されている。

- プログラム 16 行目 の fopen("average-variance.data", "r") では、カレントディレクトリ上の average-variance.data という名前のファイルを読み込み用に**オープン** (i.e. 操作可能な状態に) し、これに関するファイルポインタを変数 ifp にセットしている。 指定ファイルをオープンできない時は fopen の関数値は NULL(空のポインタ) になるので、この時は 17〜18 行目のエラー処理を行っている。
- プログラム 20 行目 の fopen("out", "w") では、カレントディレクトリ上の out という名前のファイルを書き出し用にオープンし、これに関するファイルポインタを変数 ofp にセットしている。指定ファイルをオープンできない時は fopen の関数値は NULL(空のポインタ) になるので、21〜22 行目のエラー処理を行っている。
- プログラムの 41〜42 行目 の fclose 関数は、ファイルポインタ ifp, ofp に関わるファイル操作の後始末をし、ファイルポインタの指定していた FILE 型構造体も解放している。 (ファイルを**クローズ**すると言う。)
- プログラムの 27 行目 の fscanf 関数 は、ファイルポインタ ifp の指すファイルへ書式付き入力を行っている。

- プログラムの 35 行目,37〜40 行目 の `fprintf` 関数 は、ファイルポインタ `ofp` の指すファイルへ書式付き出力を行っている。

□演習 12.2 (不定個の入力データの合計)　例題 6.19 のプログラムを入力ファイル ”inputdata” からデータを読み込む様に変形せよ。

□演習 12.3 (バブル整列法)　例題 9.6 のプログラムを入力ファイル ”inputdata” からデータを読み込み出力ファイル ”output” に処理結果を書き出す様に変形せよ。

□演習 12.4 (ファイルからのデータ入力、ファイルへの出力)　“in_file” という名前のファイルの中に多数の整数データが空白や改行コードで区切られて並んでいると仮定した上で、この入力ファイルの中のデータを 1 行に 5 個ずつきれいに並べて “out_file” という名前のファイルに出力する C プログラムを作成せよ。 入力ファイル “in_file” の内容が

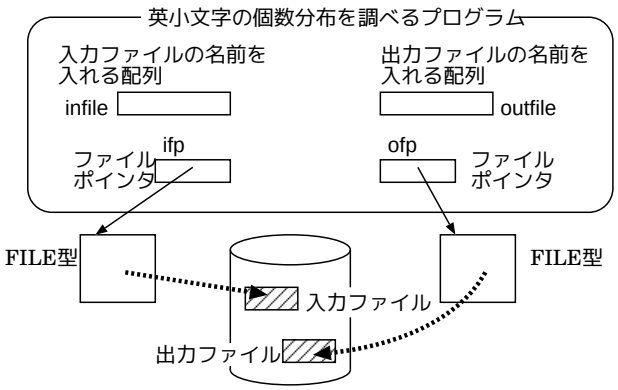
```
-1111111111 2222222 333333 4444 55 6 7 8 9 0 1 2 3 4
5           6                7890123
-5666
```

の時には、このプログラムは例えば次のような内容を出力ファイル “out_file” に書き出す。

```
␣-1111111111␣␣␣␣␣2222222␣␣␣␣␣333333␣␣␣␣␣4444␣␣␣␣␣␣␣␣␣55
␣␣␣␣␣␣␣␣␣␣6␣␣␣␣␣␣␣␣␣␣7␣␣␣␣␣␣␣␣␣␣8␣␣␣␣␣␣␣␣␣␣9␣␣␣␣␣␣␣␣␣␣0
␣␣␣␣␣␣␣␣␣␣1␣␣␣␣␣␣␣␣␣␣2␣␣␣␣␣␣␣␣␣␣3␣␣␣␣␣␣␣␣␣␣4␣␣␣␣␣␣␣␣␣␣5
␣␣␣␣␣␣␣␣␣␣6␣␣␣␣␣7890123␣␣␣␣␣␣␣-5666
```

例題 12.5 (指定したファイル中の英小文字の個数分布)　①入力ファイル、出力ファイルの名前を標準入力から受け取り、②入力ファイル中の英小文字の個数分布を出力ファイルに書き出す C プログラムを作成せよ。

(考え方)　入力ファイルと出力ファイルの名前を文字列として保持する十分長い char 型配列、それから入力ファイルに繋がるファイルポインタ, 出力ファイルに繋がるファイルポインタの領域が必要である。



これらのものが用意されていれば、

- ライブラリ関数 `fopen()` を利用して、char 型配列で指定したファイルをオープンできる。(⟹ 10.8 節を参照)
- ライブラリ関数 `fclose()` を利用して、ファイルポインタで指定したファイルをクローズできる。(⟹ 10.8 節を参照)
- ライブラリ関数 `fgetc()` を用いて、ファイルポインタで指定した先から次のデータを 1 文字分だけ読み込むことができる。(⟹ 10.8 節を参照)
- ライブラリ関数 `fprintf()` や `fputc()` を用いて、ファイルポインタで指定した先に出力データを流し込むことができる。(⟹ 10.8 節を参照)

度数分布を調べるためには、単に、英小文字 26 文字それぞれの出現回数を数えるカウンタ (初期値 0) を用意して、入力ファイルから英小文字を検出する度にその文字のカウンタを 1 だけ増やす操作を続ければ良い。 その際、カウンタを配列 `count_of_letter[0]`〜`count_of_letter[25]` として確保して

文字 'a' の出現回数を `count_of_letter[0]` で,
文字 'b' の出現回数を `count_of_letter[1]` で,
文字 'c' の出現回数を `count_of_letter[2]` で,
.....,
文字 'z' の出現回数を `count_of_letter[25]` で

数える場合は、

'a'-'a'= 97 − 97 = 0,
'b'-'a'= 98 − 97 = 1,
'c'-'a'= 99 − 97 = 2,
.....
'z'-'a'= 112 − 97 = 25

⟹ 11.3 節を参照

と計算できるので、一般には

変数 c に記憶されている文字 (番号) の出現回数は
`count_of_letter[c-'a']` で数える

ことになる。

(プログラミング) 入力ファイル, 出力ファイルの名前を文字列として保持するために各々 `infile[]`, `outfile[]` という名前の char 型配列を、入力用, 出力用のファイルポインタを保持するために各々 `ifp`, `ofp` という名前の変数を、そして英小文字 26 文字の各々の出現回数を数えるために `count_of_letter[]` という名前の int 型配列を用意して、プログラムを構成した。この C プログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[motoki@x205a]$ nl counting-lowercase-letters.c Enter
     1 /* (1) 入力ファイル，出力ファイルの名前を標準入力から受け取り、  */
     2 /* (2) 入力ファイル中の英小文字の個数分布を出力ファイルに書き出す*/
     3 /* C プログラム                                                  */

     4 #include <stdio.h>
     5 #include <stdlib.h>
```

```
 6 int main(void)
 7 {
 8   int   c, i, count_of_letter[26];
 9   char  infile[100], outfile[100];
10   FILE  *ifp, *ofp;

11   printf("Type in a name of an input file:  ");
12   scanf("%99s", infile);
13   printf("Type in a name of an output file: ");
14   scanf("%99s", outfile);
15   if ((ifp=fopen(infile, "r")) == NULL) {
16     printf("ファイルをオープン出来ません: %s\n", infile);
17     exit(1);
18   }
19   if ((ofp=fopen(outfile, "w")) == NULL) {
20     printf("ファイルをオープン出来ません: %s\n", outfile);
21     exit(1);
22   }

23   for (i=0; i<26 ; ++i)      /* カウンタを全て 0 に初期化 */
24     count_of_letter[i] = 0;

25   while ((c=fgetc(ifp)) != EOF)
26     if (c>='a' && c<='z')
27       ++count_of_letter[c-'a'];

28   for (i=0; i<26; ++i) {
29     fprintf(ofp, "%c:%4d      ", 'a'+i, count_of_letter[i]);
30     if (i%5==4)
31       fputc('\n', ofp);
32   }
33   fputc('\n', ofp);
34   fclose(ifp);
35   fclose(ofp);
36   return 0;
37 }
[motoki@x205a]$ gcc counting-lowercase-letters.c Enter
[motoki@x205a]$ ./a.out Enter
Type in a name of an input file:  counting-lowercase-letters.c Enter
Type in a name of an output file: out Enter
[motoki@x205a]$ cat out
a:  13      b:   1      c:  20      d:   6      e:  33
f:  47      g:   1      h:   4      i:  52      j:   0
```

```
k:   0      l:  20      m:   3      n:  34      o:  29
p:  23      q:   0      r:  15      s:  10      t:  34
u:  16      v:   1      w:   2      x:   2      y:   2
z:   1
[motoki@x205a]$
```

ここで、

- プログラム 9 行目 は、ifp と ofp がファイルポインタであることを宣言している。
- プログラム 14 行目 の fopen(infile, "r") では、配列 infile[] の中の文字列を名前として持つファイルを読み込み用に**オープン** (i.e. 操作可能な状態に) し、これに関するファイルポインタを変数 ifp にセットしている。 指定ファイルにアクセスできない時は fopen の関数値は NULL(空のポインタ) になので、この時は 15〜16 行目のエラー処理を行っている。

> 補足：
> 入力ファイル名が例えば "abc" と固定されているなら、
> `ifp = fopen("abc", "r");`
> という書き方でも良い。

- プログラム 18 行目 の fopen(outfile, "w") では、配列 outfile[] の中の文字列を名前として持つファイルを書き出し用にオープンし、これに関するファイルポインタを変数 ofp にセットしている。
- プログラム 24 行目 の fgetc(ifp) は、ファイルポインタ ifp で指定される入力ストリームから文字列を 1 個読み込み、その文字コードを値とするライブラリ関数である。 fgetc は読み込む文字が無くなると EOF を返す。

> 補足：
> EOF は<stdio.h>の中で定義されているマクロ定数で、入力した文字の番号と区別しなければならないため 8 ビットで表せる整数となる保証はない。
> $\Longrightarrow$ fgetc() の関数値は char ではなく int 型と決められている。
> $\Longrightarrow$ 関数 fgetc() の値を保持する変数 c を char 型にしてはいけません。

- プログラム 26 行目 の c-'a' は、読み込んだ英小文字の番号が a の番号からどれだけ離れているかを表す。char 型も整数型の一種なのでこういう計算が出来る。
- プログラムの 28 行目 の fprintf 関数 は、ファイルポインタ ofp の指すファイルへ書式付き出力を行っている。
- プログラム 29〜30 行目 では、1 行に 5 文字分のカウント結果を出力したら改行している。 式 i%5 は i を 5 で割った余りを表す。
- プログラムの 30 行目,32 行目 の fputc('\n', ofp) は、改行コードをファイルポインタ ofp の指すファイルへ出力している。
- プログラムの 33〜34 行目 の fclose 関数は、ファイルポインタ ifp, ofp に関わるファイル操作の後始末をし、ファイルポインタの指定していた FILE 型構造体も解放している。 (ファイルを**クローズ**すると言う。)

□**演習 12.6 (学生データの整理)** 例題 11.15 のプログラムを標準入力で指定した入力用ファイルからデータを読み込み標準入力で指定した出力用ファイルに処理結果を書き出す様に変形せよ。

□**演習 12.7 (英文章中の指定した単語を大文字にする)**　例題 11.8 のプログラムを標準入力で指定した入力用ファイルからデータを読み込み標準入力で指定した出力用ファイルに処理結果を書き出す様に変形せよ。

ファイルについてのまとめ：

- C 言語においては (標準入出力も含めた) ファイルへのアクセスは、通常、ファイルポインタを介して行う。
- 内部的には、**ファイルポインタ**は
  ⎧ 入力用か出力用か、
  ⎨ 次の読み込み文字の位置 (または次の書き込み場所)、
  ⎩ ファイル終端が起きたかどうか、
  などの情報から成る (FILE 型) 構造体へのポインタであり、特に標準入力, 標準出力, 標準エラー出力にアクセスするためのファイルポインタとしては、<stdio.h>の中でそれぞれ stdin , stdout , stderr という名前のもの (マクロ) が用意されている。

  補足：
  FILE 型の定義はシステムによって異なっている様である。例えば、平林雅英「ANSI C/C++辞典」(共立出版,1996) には FILE 型の定義として次の様なものが例示されている。

```
typedef struct {
  unsigned char *fpi;   /* ファイル位置指示子 */
  unsigned char *bptr;  /* バッファへのポインタ */
  unsigned int  flags;  /* ファイル状態フラグ */
  signed   int  blevel; /* バッファの充満レベル */
  signed   int  bsize;  /* バッファの大きさ */
  signed   char fd;     /* ファイル記述子 */
  unsigned char hols;   /* ungetc の確保 */
  unsigned int  istemp; /* 一時ファイル */
  signed   int  token;  /* 有効印 */
}FILE;
```

  他には、B.W. カーニハン&D.M. リッチー「プログラミング言語 C 第 2 版」(共立出版,1989)8.5 節にも簡単なのが載っている。 実際のシステムだと、VineLinux2.1 の場合 /usr/include/stdio.h と /usr/include/libio.h の中に定義されている。

  ファイルの現在操作中の位置を指すもの (e.g.FILE 型構造体の中のファイル位置指示子) を**ファイルポインタ**と呼ぶこともある。

- ファイル処理は一般に次の様に行う。
  ① `#include <stdio.h>`
  ② `FILE *` ファイルポインタ型変数 ;
  ③ ファイルポインタ型変数 `= fopen(`ファイル名を表す文字列 (へのポインタ)`,` 使用モード`);`
  但し、使用モード としては次の 3 つが可能。
  ⎧ `"w"` ··· 新規書き込み
  ⎨ `"a"` ··· 追加書き込み
  ⎩ `"r"` ··· 読み込み
  ④ (ファイル操作)
  ⑤ `fclose(` ファイルポインタ型変数 `);`

< 第 15 週 > コンピュータ入門 (II)

13 コンピュータのソフトウェア

- コンピュータの処理形態 (利用形態),
- プログラミング言語, オブジェクト指向プログラミング, どの言語を使えば良いのか?,
- プログラムの実行過程,
- オブジェクト指向言語プログラミング,
- オペレーティングシステムとその目的,
- オペレーティングシステムの構成

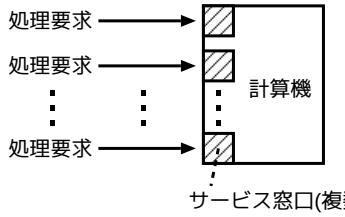
計算機の装置そのものを**ハードウェア**と呼ぶのに対して、計算機の利用技術 (特にプログラム) を**ソフトウェア**と呼ぶ。

13.1 コンピュータの処理形態 (利用形態)

通常、我々は手元にあるコンピュータを占有して使ったり、(ネットワークを通して) 共用のコンピュータを利用したりする。これら様々な利用形態に対して、サービスを提供するコンピュータ側にも色々な処理形態が用意されている。 次に、代表的な処理形態を列挙する。

- **会話型処理**: 我々は、コンピュータにコマンド (動作指示; マウスのクリックも含む) を与え、その応答を見て次のコマンドを与え、 ... という会話を通してコンピュータを利用することがほとんどである。例えば、1 台のパソコンを占有してその上でワープロや表計算等の応用ソフトを利用したり、遠隔地の UNIX 系コンピュータにログインしてファイル操作等を行ったりするのが、この処理形態の利用に当たる。

  この種の利用を可能にするために、コンピュータ側は、利用者との会話の窓口を開いて、そこからの指示を次々と実行する仕組みを用意している。 特に UNIX 系のコンピュータの場合には、会話の窓口を複数用意し、会話相手を切替えながら複数の利用者からの動作要求に並行して対処できる様になっている。



補足：
実際には、複数の相手との会話は次のように行われる。
◇ 計算機内に動作要求の待ち行列を保持し、基本的にはこの順番に処理を行う。
◇ 1 つの要求に対する処理がある時間を越えると、この処理は途中で打ち切り待ち行列の最後に移す。

コンピュータが高価であった時代には、中央の 1 台の計算機を複数の利用者が同時に, そしてオンラインで会話的に利用する方式がよく用いられていた。 この処理方式を特に**タイムシェアリング処理** (time sharing processing, **時分割処理**,**TSS 処理**) と呼ぶ。

- **リアルタイム処理** (real time processing, **実時間処理**): 発生したデータ (または処理要求) に対して、計算機への入力, 処理, 出力が要求された時間 (普通数秒から数十秒) 内に行われる処理方式をいう。通常、計算機への入力はオンライン (i.e. 処理要求発生の地点から通信回線を通して直接計算機へ入力する方式) で行われるので、**オンラインリアルタイム処理** (online real-time —) となる。

リアルタイム処理の対象分野としては、
例えば次のものがある。

① **自動機械, 工程などの制御** ··· センサからの入力に応じて計算機が動作し、計算結果を基に計算機が直接全体を制御する。

② **交通の制御/管制** ··· センサや人間, 他の計算機などからの入力に応じて計算機が動作し、計算結果と人間の意志を基に交通を制御/管制する。

③ **問い合わせ応答** (e.g. 座席予約, バンキング, 情報検索サービス) ··· 数多くの端末からランダムに入力される問い合わせに対して、計算機はその答を返す。一般に、中央に保持するデータベースへのアクセス結果によって問い合わせの答が決まる。

1970 年代頃までは、
バッチ処理 (batch processing, **一括処理**) と呼ばれる処理形態もあった。これは、**クローズドショップ**式の利用形態 (i.e. 利用者が直接計算機を操作しない利用方式) の下で計算機の稼働率を上げるために考え出された処理方式であり、ある期間内に集められた**ジョブ** (job; 利用者から見た、計算機に処理させる仕事の単位) を幾つかずつまとめて処理していくというものである。 ここで、1 回の処理のためにまとめられたジョブの束を**バッチ** (batch) と呼んだ。また、計算機から離れた場所から**オンライン** (online;i.e. 通信回線を介して) でジョブの入力/結果の出力を行う場合を、特に**リモートバッチ処理** (remote batch —) と呼んだ。

現在では、個々のコンピュータの処理能力が向上し、ほとんど全ての処理が手元のパソコンで賄える様になってきたし、また、手元のパソコンの手に負えないジョブを共用のスーパーコンピュータに投入する場合も、(過重負荷にならないなら) ジョブ投入後即座にジョブの実行が開始されるので、昔の、ジョブを溜めた後で複数を組み合わせて並行に実行するタイプのバッチ処理は今では全く行われていないだろう。

13.2 プログラミング言語

計算機に何らかの処理を行わせたい時には、その処理手順を計算機の理解できる形で明確に記述して計算機に教え込ま (i.e. 入力し) なければならない。一般に、処理手順の記述形式のうち計算機の理解できる様に人工的に設計されたものは、計算機と利用者の間のコミュニケーションの手段を与えるので**プログラミング言語** (programming language) と呼ばれる。そして、1 つのプログラミング言語の下で処理手順を記述したものを**プログラム** (program) という。特に裸の計算機の理解できる言語は、その計算機の**アーキテクチャ** (architecture; ハードウェアの論理的構造) に直接関わるので**機械語** (machine language) と呼ばれる。機械語のプログラムは計算機の基本動作を表すビット列をそのまま並べたものであるため抽象度が低く分かりにくい。

初期の計算機ではプログラミング言語としては機械語しかなく、プログラム作りは非常

に困難であった。そこで、**アセンブリ言語** (assembly language) と呼ばれる言語が次に出現した。この言語では、プログラムの基本動作はビット列ではなく「基本動作の論理的意味を反映した記号」を基に構成される。そのためプログラムの書き易さ, 理解し易さはある程度改善されたが、アセンブリ言語のプログラムの各ステップは計算機の基本動作とほぼ1対1に対応しているため、我々人間が簡単と感じる処理でも多くのステップが必要になる。その上、機械語同様計算機のアーキテクチャを反映した言語であるため、1つの計算機についてアセンブリ言語を修得したとしても、それで別の計算機のプログラムを作れるとは限らない。

もちろん、アセンブリ言語のプログラムはそのままの形では計算機が理解できないので、一旦機械語に翻訳されなければならない。この翻訳を行うプログラムを**アセンブラ** (assembler) という。

プログラミング言語としてはアセンブリ言語よりもっと人間に近いものが望ましい。できることなら日本語, 英語などの自然言語をプログラミング言語として、どんな人でもすぐにプログラミングができる様になればよい。この考えの下に、数式や日常言語に近い形でのプログラミングを目指して最初に設計された言語がFORTRAN(1957年完成,IBM社) である。FORTRAN は計算機使用者に対してプログラムの作成時間を大幅に短縮させ、その結果計算機の使用者をどんどんと増やしていった。そして、FORTRAN の成功に伴い、日常言語に近い形でのプログラミングを目指した言語 (**高水準言語**または**高級言語**,high-level language, という) が次々と開発されていった。高水準言語は個別の計算機アーキテクチャに依存せず, 人間にとって比較的分かり易く, また処理手順の記述力も強いため、プログラムの生産性や保守性の点で優れている。

もちろん、高水準言語のプログラムもそのままでは実行できない。アセンブリ言語の場合と同様一旦機械語に翻訳してから実行したり、プログラムを1ステップ毎に理解/解釈しながら実行させたり、しなければならない。高水準言語のプログラムを機械語に翻訳するソフトウェアを**コンパイラ** (compiler), 高水準言語のプログラムを解釈しながら実行するソフトウェアを**インタープリタ** (interpreter) という。

次に、高水準言語の例をいくつか列挙する。

- **FORTRAN**(FORmula TRANslator,1957〜): 最も古く、昔は (汎用機の下で)COBOL に次いで世界中でよく使われていた。元々科学技術計算用に IBM 社により開発されたものであるが、FORTRAN I(1958),FORTRAN IV(1962),FORTRAN66(1966 年に国際的に標準化),FORTRAN77,Fortran90 と改良が進み、それに伴って汎用性も高くなっていった。FORTRAN は記憶容量の面でも実行時間の面でもハードウェアの性能を最大限に引き出せる様に設計されており、数値計算や統計計算などに有用な組込み関数, ライブラリが豊富に用意されている。
- **ALGOL60**(ALGOrithmic Language,1958〜): 構造化された (i.e. アルゴリズムを綺麗に記述する能力を持つ) 科学技術計算用言語で、ヨーロッパで生まれた。大手計算機メーカーの積極的支持が得られなかったため、1981 年に国際規格から,1987 年に JIS から除外された。
- **ALGOL68**(1968〜): ALGOL60 の後継言語として設計されたが、実行時の効率を余りにも重視したため ALGOL60 とは異なる基本的概念を持つ。様々の機能 (e.g. データ型の変換, 並列処理) を備えており、Pascal や Ada を始めとする最近のプログラミング言語に大きな影響を与えた。
- **Pascal**(1968〜): プログラミングの組織的教育, 計算効率の良さ, 処理系の作り易さを目指して N.Wirth によって設計された言語であり、研究・教育用として現在よく用いられている。全般的に ALGOL60 の骨格を採用し、系統的プログラミングに適している (i.e. アルゴリズムを自然に分かり易く記述する能力を持つ)。
- **Ada**(1980〜): ソフトウェア費用 (特に、古いソフトウェアの保守) の増大に対処するために米国国防総省 (DOD) の主導で開発された。元々、組込み型計算機 (i.e. 武器, 航空機, 船舶, ミサイル, 高速交通機関などの電子機械システムに組み込まれた計算機) に入れる制御用ソフトウェア開発のための統一言語である。Ada は Pascal を基礎に設計されたが、その目標である高信頼性プログラミング，保守容易性，効率，プログラムの読み易さを実現するために様々な機能 (e.g. モジュール化, 並行処理, 例外

処理) を備えた汎用高水準プログラミング言語となった。また、その名前は C.Babbage の解析機関のプログラムについて考察して史上最初のプログラマとなった Ada Augusta(Lovelance 伯爵夫人で詩人 Byron の娘) に因んで付けられた。

- **C**(1972〜): AT&T ベル研究所のミニコンピュータ PDP-11 上に開発されていた UNIX オペレーティングシステム (アセンブリ言語で記述されていた) を高水準言語で書き換えるために D.Ritchie によって設計された言語であり、その系譜を言語 C ←言語 B(1970〜) ←言語 BCPL(1967〜; データ型のない言語で、これを用いて英国ケンブリッジ大学では OS-6 というオペレーティングシステムを開発した) とさかのぼることができる。改良/標準化の際には Pascal の影響も受けた。データの取扱いに関して「低水準言語」の性格を持っているためシステム記述言語という色彩が強いが、構造的プログラミングのための道具立ても揃っているため科学技術計算のための汎用言語として使うこともできる。
- **C++**(1983〜): 代表的なオブジェクト指向言語の 1 つ。 元々は、離散事象のシミュレーションを効率的に行うために、C 言語と Simula67 を土台にして AT&T ベル研究所の B.Stroustrup によって 1980 年頃に設計が始まった言語で、最初は「クラス付き C (C with classes)」という名前で発表された。C++と命名された 1983 年以降も C 言語との互換性を保ちながら拡張/改良が加えられたため、従来の手続き型プログラミングにもオブジェクト指向プログラミングにも対応できる様になっている。
- **Java**(1995〜): 現在最も注目を集めているオブジェクト指向言語。元々は、情報家電製品にソフトウェアを組み込んでその配付やバージョンアップを円滑に行うために、C 言語と C++を土台にして Sun Microsystems 社の J.Gosling によって 1991 年に設計が始まった言語で、最初は Oak と名付けられていた。(開発社の部屋の窓から Oak の大木が見えたから。) しかし、Oak という名前の言語が既にあることが判明し、Java という名前になった。(開発者達が喫茶室に集まった時、コーヒーの銘柄にちなんで提案された。) 本来の Sun 社のねらった情報家電製品の市場は伸びなかったけれども、その後、1993 年以降の WWW の爆発的な人気により、Java はダイナミックな Web ページ作成ツールとして急速に発展・普及している。
- **Modula-2**(MODUlar LAnguage,1979〜): モジュール構造を持ったプログラムが容易に作れる様にすることによって構造的プログラミングを強力に支援し、更に多重プログラミング／並行プロセス環境をサポートする汎用システム記述言語を目指して、N.Wirth が Pascal を基に設計した。
- **SIMULA**(1964〜): 元々、離散的事象を扱うシミュレーションとシステム記述のために ALGOL60 を拡張して開発された言語 (SUMULA I) であるが、その後の改良/改訂により汎用システム記述言語 (SIMULA67) となった。SIMULA67 は抽象データ型を扱える最初の言語であり、Ada,CLU,Smalltalk,Modula などの言語に多くの影響を与えた。
- **Smalltalk**(1972〜): 元々「子供の教育に計算機を使うにはどうすればよいか?」ということからゼロックス・パロアルト研究所の Alan Kay のプロジェクトで開発された言語であるが、現在では高性能ワークステーション (または高機能パソコン) のためのオブジェクト指向プログラミング言語として有名である。その設計思想としては“優れたマン・マシン・インターフェース”を目指し、最初のシステム Smalltalk-72 以来 2 年毎に新しいシステムが開発されていった。そして、ウィンドゥシステム, アクティブドキュメントシステムなどが新しく組み込まれた Smalltalk-80 になって初めて世の中に公開された。優れたマン・マシン・インターフェースを持つためにベストセラーになったアップル社の Macintosh は Smalltalk を使って開発 (ユーザ環境が Smalltalk で記述) され、そのハードウェア自身もゼロックス社内の研究用に開発された高性能ワークステーション ALTO を原型としてもつ。
- **COBOL**(COmmon Business Oriented Language,1959〜): 事務データ処理を行うための共通言語として CODASYL(the COnference on DAta SYstems Languages; 計算機メーカーや利用者が協力して設立した組織) が設計したもので、英語に似た言語仕様で様式が統一されている。COBOL は事務処理用言語としては最も古く、昔は汎用機の下で世界中で最も広く普及 (全使用量の 6 割?) していた。言語の改良も COBOL60 → COBOL61 → COBOL65 → COBOL74 → COBOL85 と進んでいる。
- **RPG**(Report Program Generator,1960〜): 報告書作成, データ検索, 給与計算, 受注業務, 生産計画など広汎な経営事務用のプログラムを作成するために設計されたプログラミング言語である。RPG は FORTRAN や COBOL などの通常の手続き型言語と異なり表記入方式であり、入出力ファイルや出力すべき報告の内容や書式に関する情報をパラメータとして受け取りこれを基に所要のプログラムを生成する。従って構文上の柔軟性に欠けるためプログラミング言語でないと見ることもできるが、プログラム開発・保守上の生産性が高く利用者も多かった。言語の改良も RPG(IBM1401 用) → RPG(IBM System 360 用,1964) → RPG II(1969) → RPG III(1978) と進んでいた。
- **PL/I**(Programming Language/one,1965〜): FORTRAN,ALGOL60,JOVIAL,COBOL を基に科学計算と経営事務計算を含む汎用のプログラミング言語として IBM 社と SHARE(ユーザ団体) によって開発された。記憶領域の動的割付, 並行処理など様々な機能を備えた非常に大きな言語仕様になっている。
- **BASIC**(Beginner's All-purpose Symbolic Instruction Code,1965〜): 1964 年にダートマス大学で開発された TSS(i.e. タイムシェアリング処理システム) 用に、初心者でも楽に使える様に設計された会

話型プログラミング言語。元々はコンパイラ言語として開発されたが、初期のパソコンではインタープリタが無料で付いているためインタープリタ言語として広く普及していた。

- **APL**(A Programming Language,1962〜): MIT の K.E.Iverson によって提唱されたため初期の頃はアイバーソン言語とも呼ばれた。多次元配列を含む数式を簡潔に記述できる様に数多くの演算子が用意され、更に新しい演算子を定義することもできる。数式を記述するのに括弧を使わず前置記法を使うのが特徴で、対話型プログラミング言語として科学技術計算にも商業事務計算にも広く用いられている。
- **Lisp**(LISt Processor,1958〜): 非数値情報の処理のために MIT の J.McCarthy らによって開発された関数型プログラミング言語であり、リスト (list; データの並び) の処理が得意。その出現以来、定理の証明, 数式処理, パターン認識, 計算機によるゲーム, ロボットのソフトウェアなど、人工知能研究用の言語として主に発展し、Lisp 専用の計算機 (Lisp マシンという) もいくつか開発されていた。 方言も多い (e.g.InterLisp,MacLisp,FranzLisp,Scheme,UTILisp) が、標準化を求める声の高まりによって Common Lisp が案として設計された。
- **Prolog**(PROgramming in LOGic,1972〜): 元々自然言語の構文解析のために A.Colmerauer らによって開発されたプログラミング言語であるが、R.A.Kowalski の論文 (1974) によって記号論理との関連が明らかにされてからは論理プログラミング言語として知られる様になった。そして、D.H.D.Warren らによって実用的な処理系 (DEC-10 Prolog と呼ばれるものでインタープリタとコンパイラを含む,1977) が作られたこと，さらには日本の第五世代コンピュータプロジェクト (1982〜) の核言語として採用されたことによって、Lisp と並ぶ人工知能用言語として広く普及した。ICOT(Institute for new generation COmputer Technology, 新世代コンピュータ技術開発機構; 第五世代コンピュータプロジェクトのために作られた研究組織,1982〜) によって開発された ESP,GHC を始め Prolog の改良/拡張も盛んに行われた。標準化のための国際的な活動は、事実上の標準であった DEC-10 Prolog を基礎言語として 1988 年に開始され、1992 年の時点で最終段階に入った。
- **OPS**(Official Production System,1977〜): 元々人工知能の認知心理学的研究のために C.L.Forgy らによって開発された言語であるが、その汎用性から実用的エキスパートシステム構築ツールとして発展し成功を納めた。OPS(1977) → · · · → OPS4(1979) → OPS5(1981) → OPS83(1983) と改良が繰り返されてきたが、その内 OPS5 が計算機システムの構成を行うエキスパートシステム XCON(DEC 社),OS 管理エキスパートシステム YES/MVS(IBM 社), 電話ケーブル保守のエキスパートシステム ACE(AT&T 社) などの構築ツールとして使用されたことは有名である。
- **LOGO**(ギリシャ語の $\lambda o \gamma o \varsigma$ に由来,1967〜): 子供が概念を習得する過程をとらえた心理学者 J.Piaget のユニークな発生的認識論 (「概念は直観的モデルによってとらえられ、直観的モデルは身体運動感覚をベースにした経験を構造化して作り上げられてゆく」というもの?) に基づいて、心のモデルを計算機上に実現するために MIT の S.Papert によって考案されたプログラミング言語である。「子供が身体経験を基に直観に訴えながら幾何学図形や物理学の運動法則を擬人的にプログラム化できる言語」, あるいは「子供が教師なしで自立的に学習するための、従って新しい創造教育のためのツール」として普及し、実際の教育現場では LOGO を使った教育開拓も行われていた。プログラミング言語としては Lisp の影響が強く、その後の改良/拡張により人工知能向きの言語としても利用価値が高い。
- **SNOBOL**(StriNg-Oriented symBOlic Language,1964〜): AT&T ベル研究所で開発された文字列処理用プログラミング言語で、文字列の検査, 置換, 連結などの操作が簡単にできる。言語翻訳, 言語解析などの分野で用いられている。
- **GPSS**(General Purpose System Simulator,1961〜): IBM 社の G.Gordon らによって開発されたシステムシミュレーション専用のプログラミング言語 (システムシミュレータという) であり、待ち行列などの社会現象を始め複雑なシステムの (離散的) シミュレーションに用いられる。例えば、計算機システム (オンライン・リアルタイムシステム, タイムシェアリングシステムも含む), 工程管理システム, 在庫管理システム, 交通システム, 事務管理システムの設計/解析を始め、政治, 経済, 社会システムの様にその実態の把握が困難なシステムの分析にも応用される。
- **QBE**(Query By Example,1975〜): 関係データベースについて表の表示や条件の例示により問い合わせを行うためのエンドユーザ (i.e. 専任のプログラマやオペレータ以外の利用者) 用簡易言語である。IBM 社の汎用機に実装されていた。
- **SQL**(Structured Query Language,1974〜): 関係データベースにおいてデータの定義, 操作, 制御を行うための言語であり、FORTRAN,COBOL,PL/I,Pascal で書かれた応用プログラムからでも使用可能である。この最初の版は D.D.Chamberlin と IBM 社サンノゼ研究所で開発され SEQUEL と名付けられていたが、法律 (登録商標) 上の問題からその後の改良の際に SQL と改名された。発展の経緯をたどると SEQUEL(1974) → SEQUEL/2(1977,IBM 社のデータベースシステム R 上に実装された) → SQL(197?) → · · · → SQL2(1988) となり、1986 年にアメリカ国家規格 ANSI に, 1987 年に国際規格 ISO に,1988 年に日本工業規格 JIS に制定されるなど標準化も進んだ。現在の主要な関係データベースシステムでは SQL 言語を使用できる。

オブジェクト指向プログラミング:

- 大規模なプログラミングを効率的に行うためには、個々のソフトウェアモジュールの独立性を高め、それらを再利用可能なソフトウェア部品としなければならない。

  ⇨ モジュール化、「抽象データ型」といった考え方を更に 1 歩進めたものが**オブジェクト指向** (object-oriented) の考え方。

- オブジェクト指向プログラミングにおいては、データとそのデータに関わる操作をカプセル化した**オブジェクト** (object) と呼ばれる、自律した動作主を考え、それらのオブジェクトが互いに動作依頼のメッセージを送り合いながらプログラムの実行を進めてゆく。

  ⇨ プログラムの動作を、オブジェクト同士の相互作用のシミュレーションと見ることが出来る。

どの言語を使えば良いのか?: 使用目的に合った言語を選べば、プログラムも作り易い。例えばアセンブリ言語だと、

- ハードウェア動作を理解するのが目的なら、これが良い。
- うまく書くと高速でコンパクトなプログラムが可能。
- 入出力機器の制御といった細かなプログラミングも可能。

しかし、

- コンピュータの機種に依存したプログラムしか出来ない。(i.e. 移植性がない。)
- 大規模なプログラミングには向かない。

⇨ 応用プログラム作成にどうしても必要という場合でも、アセンブリ言語は本当に必要な箇所だけに限定すべき。

高級言語に関しては、

- 科学計算なら、C, C++, Fortran, ... あたり。
- C 言語は UNIX/Linux に必ず付いているので、色々なコンピュータ上で実行できる。
- Java は実行速度が遅いので、科学計算には向かない。
- 全ての可能性について試行錯誤を繰り返したければ、Prolog が便利。
- 人工知能関連の処理をしたければ、Lisp, Prolog, ... 。
- ほとんどの言語はシステムプログラムを書くのに向かないが、C 言語のようにシステムプログラムを書くのに適した言語もある。

  例えば、
  UNIX はほとんどの部分が C 言語で書かれている。
  プリンタの動作を制御するプログラムも C 言語で書かれている。

13.3 プログラムの実行過程

前節 13.2 で見た様に、高水準言語, アセンブリ言語のプログラムはそのままの形では計算機で実行できず、実行のためにはインタープリタまたはコンパイラが必要である。以下、これらのソフトウェアを用いた 2 つの実行方式について簡単に説明する。

インタープリタによるプログラムの実行: プログラムの実行を我々人間の手で追跡する場合はプログラムを 1 ステップずつ理解/解釈して、そのステップでどういう影響/変化がもたらされるかを記録しながら実行手順を進めてゆく。この様な追跡はもちろん計算機で行うことも可能で、それを行うプログラムを**インタープリタ** (interpreter) と言う。元のプログラムを (コンパイラやプリプロセッサによって) 仮想機械語や構文解析木などの中間言語に変換して、それを基にインタープリタで実行させることもある。しかし、インタープリタはあくまでプログラムの各ステップ (の意味) を解釈しながら実行を進めてゆくだけで、各々のステップを機械語に翻訳することはない。 実際には、Java や APL, BASIC, Lisp, Prolog, Smalltalk などの会話型言語に対してインタープリタがよく用いられている。

補足：
Java, Smalltalk の場合は、通常、プログラムを一旦仮想機械語に翻訳しその結果をインタープリタで実行する。

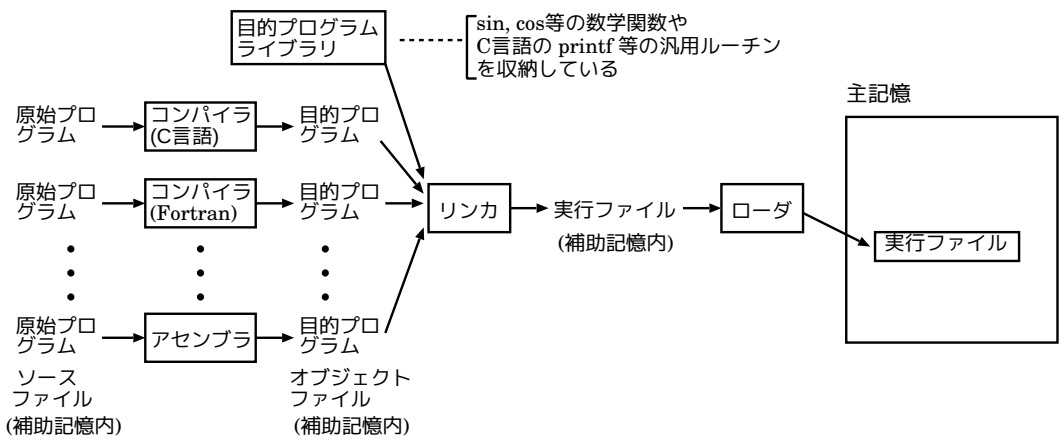
⇨ コンパイラ方式と比較した場合のインタープリタ方式の長所と短所は次の通りである。

長所 (1) 動的な言語機能 (e.g. 動的なデータ型, 変数や関数の動的宣言, 名前の動的有効範囲) に対処し易い,
(2) 処理系の作成が容易,
(3) プログラムの部分的実行が容易,
(4) デバッグ (i.e. プログラムの修正) が容易,
(5) 翻訳などの時間が不要になる。

短所 (1) 実行速度が遅い (翻訳されたプログラムの場合の数倍〜数十倍の時間がかかる),
(2) コードの最適化に相当することができにくい。

コンパイラによるプログラムの実行: 高水準言語のプログラム (**原始プログラム**,source program, と言う) は、通常、**コンパイラ**と呼ばれるソフトウェアによって機械語または中間言語 (e.g. 仮想機械語, 構文解析木) に翻訳 (compile) されてから実行される。しかし、コンパイラによって機械語に翻訳されたからと言って、それがそのままの形で計算機で実行されるとは限らない。一般には、高水準言語のプログラムは次の図の様なプロセスを経て実行される。

従って、コンパイラはプログラムの全ての部分を翻訳する訳ではない。



ここで、コンパイラ, アセンブラ, リンカ (linker; リリンケージエディタ, linkage editor, ともいう), ローダ (loader) は各々次の様な働きをするソフトウェアである。

コンパイラ, アセンブラ ··· 高水準またはアセンブリ言語の原始プログラムを**目的プログラム** (object program) と呼ばれるソフトウェアモジュール (module, 部品) に変換する。目的プログラムは原始プログラムを翻訳して得られた機械語コードの他に、外部記号 (external symbol;e.g. sin, cos などの外部関数名) の表なども含む。外部記号は原始プログラムを見ただけでは翻訳できない部分である。

リンカ ··· コンパイラ, アセンブラ等によって生成された目的プログラム群, ライブラリ内の目的プログラム群を結合して、ロードモジュール (load module) のプログラムに変換する。ロードモジュールのプログラムもやはり外部記号の表等を含むこともあり、それ故再びリンカへの入力となれる。

補足：
複雑な処理手順はプログラムの作成／保守が容易になる様に綺麗に分割してプログラミングされる。その分割の際にできた小さなモジュールはそれぞれ異なる言語を用いてプログラミングが可能である。また、sin や cos などの汎用の関数計算ルーチン, 統計計算用ルーチンはライブラリの中に収納されている。コンパイラやアセンブラは個々のモジュールについて翻訳はするけれどもこれらを統合することはないので、ある程度以上の大きなプログラムの実行にはこのソフトウェアが必要になる。

ローダ ··· それまでで未定の部分 (e.g. 場合によっては主記憶上の番地) があればそれらを決定しながらロードモジュールのプログラムを主記憶に移す (i.e.load する)。もちろん、外部記号の表などはもはや主記憶上にはない。

13.4 オペレーティングシステムとその目的

裸の (i.e. ハードウェアだけの) 計算機では 13.2 節でも述べた様に使い勝手が悪く、しかも「利用希望者は予約を取りその時間は一人で計算機を独占する」という利用形態しか考えられないため計算機の有効利用は望めない。

そこで、計算機を利用する際には一般に高水準言語を用いるわけであるが、高水準言語のプログラムを実行させる場合には 13.2 節で見た様にインタープリタ, またはコンパイラ, リンカ, ローダというソフトウェアが必要になる。また、作成したプログラムを次回の使用のために磁気ディスク等に保存しておくためのソフトウェア, 入出力装置を使う際にはそれらの装置に固有の制御を行うソフトウェア, さらに通信回線を介して計算機を使う際には端末と計算機の間のデータのやり取りを行うソフトウェアも必要になる。 この様に、高水準言語のプログラムとそれを実行する計算機ハードウェアの間には大きなギャップがあり、これを埋める必要がある。もちろん、そのために用意するソフトウェアの機能はプログラミングの際に用いた言語に依らず、汎用性の高いものでなければならない。

⇨ 利用者と計算機ハードウェアの間に**オペレーティングシステム** (operating system,**OS**, **基本ソフト**) と呼ばれるソフトウェアを設け、計算機システムの運用に関して次の様な目標の達成 (または性能の向上) を図らせる。

(1) 計算機システムの有効利用: 初期の頃は計算機が高価であったためこの目標を達成することが最も重要な課題であり、これがOS誕生の動機となった。この目標の達成度の具体的目安としては**スループット** (throughput; 単位時間内に処理されるジョブの量, 件数) が一般に使われる。スループットの向上のためには、OSはシステムの各資源 (e.g.CPU, 主記憶, 補助記憶, 入出力装置, システムソフトウェア) が最大限にバランス良く利用される様にしなければならない。これらの処理のためにOS自身がCPUを使うこともあり、このための時間を**オーバーヘッド** (overhead) という。

(2) 快適な使用環境 (ユーザインターフェース) の実現:

①汎用性の向上

- 会話型処理, リアルタイム処理など様々な処理形態を (同時に) 受け入れる。

②応答時間の短縮

③使い易さの向上

- 一般ユーザが快適にコンピュータを使える環境を提供する。

例えば、
マウス, アイコン, ウィンドウシステム, ... による GUI。

- 実際のハードウェア環境への依存性をできるだけ少なくする。すなわち、計算機内部について詳細に知らなくてもよい様にする。

例えば、
プログラムの中で入出力を容易に実行できる。

補足：
入出力機器は機器毎に制御の仕方が違う。
$\Longrightarrow$ 入出力機器の細かな制御も個々のシステムプログラマに任せてしまうというのでは、プログラマの負担が大きくなる。
$\Longrightarrow$ 入出力などの共通機能を容易に実行できる環境をハードウェアに付随して提供すれば、プログラマの生産性向上につながる。 これが初期のOSの目的であった。

- マニュアル (i.e. 取扱い説明書) なしでもシステムからの指示によって容易に使いこなせる様にする。
- 豊富な種類のプログラミング言語が用意されている。
- 希望者には、ジョブの実行を自分で管理する環境が設定される。
- 利用者の身近な場所からシステムを自由に利用できる。
- 運用の自動化・省力化が図られている。オペレーションが容易な様にする。
- プログラムの開発を支援する機能がある。
- 共同作業が容易にできる様になっている。

④拡張性の向上

- 機能の拡張がシステム全体に影響することなく局所的に容易に行える様にする。

⑤**RASIS**の向上

- Reliability(信頼性) $\cdots$ 故障しにくい。
- Availability(可用性) $\cdots$ 正常に稼働している時間の割合が高い。[故障しにくくても修理に時間がかかれば問題。]
- Serviceability(保守性) $\cdots$ 故障の際、修理/復旧が容易。
- Integrity(保全性) $\cdots$ 誤動作やプログラムの誤り等によってシステムが簡単に破壊される様なことがない。

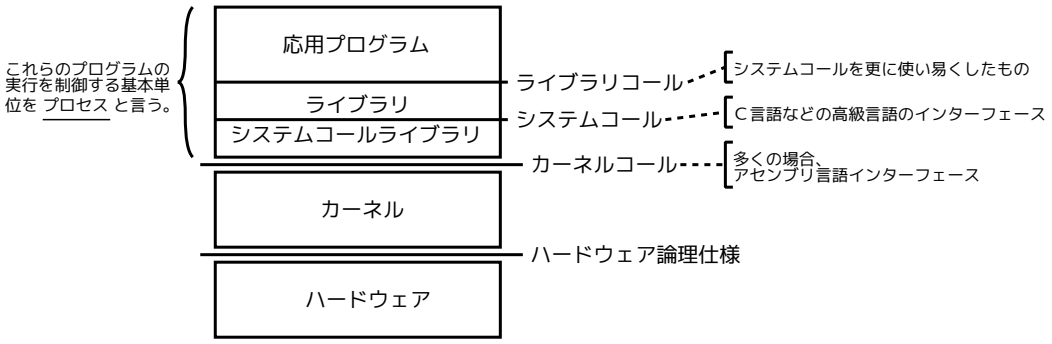
- Security(機密性) ··· ソフトウェアやデータを本来の利用者以外から保護する。すなわち、盗用されたり破壊されたりしない様にする。

これらの目標の中には相反するものもあるが、これらは利用者の期待を全て満たす様にバランス良く達成されなければならない。

例えば、
使い易さや信頼性を十分に確保しようとすれば、
その分だけシステムが重くなることがある。

13.5 オペレーティングシステムの構成

ソフトウェアの階層：



カーネルの機能： カーネルは次の 4 つの機能ブロックから成る。

(1) システム制御
- 開始処理,
- 終了処理,
- 装置管理,
- 障害の管理

(2) 実行管理
- プロセス管理,
- プロセス間通信管理,
- メモリ管理,
- 割込み制御,
- プログラム管理,
- 共通処理

(3) 入出力制御
- 周辺入出力制御,
- 通信制御

(4) ファイル管理
- 入出力効率化のためのアクセス制御,
- 外部記憶装置の領域管理,
- ディレクトリ処理,
- ファイル操作の機能を提供

14 コンピュータ発展の歴史

- 計算機が生まれるまで,
- 歯車式計算機,
- カード式計算機,
- 電子計算機,
- 記憶素子の進歩,
- OS 発展の流れ

14.1 計算機が生まれるまで

B.C.600 頃 (バビロニア, 中国) 数を表す文字
12 世紀 アラビア数字の普及 (インドで発明されアラビア人によってヨーロッパに伝わった。)
13 世紀 (中国) そろばん
16 世紀 計算尺
} 計算器 (calculator)

14.2 歯車式計算機

1642 年 B.Pascal(仏) 加算機
1694 年 G.W.Leibniz(独) 四則演算機 (平方根の計算もできる。)
1820 年 トーマス (仏) 四則演算機の商品化
1823 年 C.Babbage(英) **階差機関** (Difference Engine) を設計

(対数計算用の計算機械。計算手順を機械的に覚え、それに従って計算を自動的に進める機構を備えていた。1854 年に完成。)

1833 年 C.Babbage(英) **解析機関** (Analytical Engine) を構想

(ジャカールの発明した模様織り織機のパンチカードからヒントを得て考案した (歯車式) 汎用ディジタル計算機で、未完成。 貯蔵部 (store) と演算部 (mill) から成り、計算手順とデータをパンチカードで入力して自動的に計算する機構を持っており、現在の計算機の基本概念をほとんど備えていると言われている。)

14.3 カード式計算機

1799 年 J.-M.Jacquard(仏) ジャカール機械の発明

(パンチカードで機械の動作を制御することによって複雑な模様織りの工程も簡単に操作できる自動織機。)

1889 年 H.Hollerith(米) パンチカード式統計作表機の開発

(1890 年の国勢調査に採用され威力を発揮した。ここで使われたパンチカードは当時の 1 ドル紙幣の大きさで、現在使われているカードはこれを横に 3/4 インチ伸ばしたものである。)

1896 年 H.Hollerith(米) Tabulating Machine 社を設立 (後に **IBM 社**に発展。)

14.4 電子計算機

ここでは、使われ方を基にコンピュータの発展過程を幾つかの世代に分けて考える。

第 0 世代 (黎明期):

1936 年 A.Turing(英) **チューリング機械**の概念を発表 (現在の計算機の数学的モデル。)

1937 年 H.H.Aiken(米 Harvard 大)&IBM 社
大型リレー式計算機 **Harvard Mark I** の開発に着手

(世界最初の高速計算機で 1944 年に完成。リレーは制御電流により電気回路の開閉を機械的に行う動接触部品であるので、これは電子計算機ではない。)

1939 年 アタナソフ&ベリー (米 Iowa 大) 電子的に加減算を行う計算機

(真空管を用いた演算機構, **2 進法**の計算, コンデンサを用いた動的な記憶回路。)

1940 年 N.Wiener(米) 電子計算機の**基本構想**を提唱 (例えば 2 進法の採用など。)

第 1 世代 (商用化が始まった時期; 計算処理): 一部の専門家が機械語，アセンブラ語等を用いて主に科学技術計算を行った時代で、文字に対する配慮が不十分であった。

論理素子 (演算部)···真空管 (1904 年に発明されている。動作時間は 10m 秒〜1μ 秒 (i.e.10^{-2}〜10^{-6} 秒))
記憶素子 (主記憶)···超音波水銀遅延線, 静電管 (, 磁気ドラム)

1943 年 J.P.Eckert&J.W.Mauchly(米 Pennsylvania 大)
大型電子計算機 **ENIAC** (Electronic Numerical Integrator And Calculator) の開発に着手

(長い間、これが世界最初の電子計算機として認識されていた。1946 年に完成。18000 本の真空管と 1500 個のリレーが使われ、重さ 30t, 消費電力 150kW であった。 10 桁の 10 進固定小数点方式で数値を記憶し、5000 回／秒の加減算, 357 回／秒の乗算, 166 回／秒の除算が出来た。当時としては驚異的な計算速度 (Harvard Mark I の約 1000 倍) を持ち、約 10 年間米国陸軍で弾道計算に使われた。しかし、配電盤上に配線を行ってプログラム (i.e. 計算手順) を与える方式であった。この方式と真空管の寿命の短さのために、2 週間の内で実際に計算のために ENIAC が稼働したのは 2 時間位であり、修理と配線にほとんどの時間が費やされたと言われている。)

1945 年 J.von Neumann(米) **プログラム内蔵方式** (Stored Program System) を提唱

(プログラム (i.e. 処理手順) を記憶装置中に格納する方式で、ノイマン型ともいう。この方式では、データと全く同様にプログラムを加工できる。)

1946 年 米 Pennsylvania 大 (J.P.Eckert と J.W.Mauchly は去っている)
プログラム内蔵方式,2 進法を採用した電子計算機 EDVAC (Electronic Delay Strage Automatic Calculator) の開発に着手

(1952 年に完成。データとプログラムを記憶するために超音波水銀遅延線 (1024 語) が採用され，3000 本の真空管が使われた。EDSAC が 1949 年に完成したため、世界最初のプログラム内蔵方式の計算機にはならなかった。)

1947 年 M.V.Wilkes(英 Cambridge 大;ENIAC プロジェクトの講習会に参加)
プログラム内蔵方式, 2 進法を採用した電子計算機 **EDSAC** (Electronic Delay Strage Automatic Calculator) の開発に着手

世界で最初に動作したプログラム内蔵方式の計算機で、1949 年に完成。ここでは、負数を表すために 2 の補数表示が採用され、浮動小数点演算はインタープリタで実現された。主記憶に超音波水銀遅延線を採用して真空管の個数を減らすなど、アーキテクチャは EDVAC に類似しているが、このプロジェクトを通してソフトウェアに関する重要な概念が数多く見い出され、プログラム内蔵方式の採用によってもたらされるプログラミングの柔軟性・可能性が具体的に示された。例えば、サブルーチンとサブルーチンリンクの標準化, 一般利用者に有用なプログラムをライブラリプログラムとして提供, システムプログラムの原典となった初期入力ルーチン (initial order; プログラムを記憶装置の中に入れるための長さ 40 語のプログラム), など。

1947 年 H.H.Goldstein&J.von Neumann(米) **流れ図** (flowchart) の使用を提案

1950 年 J.W.Forrester(米 MIT),J.A.Rajchman(米 RCA,1〜2 年後に独立に)
磁気コア記憶を発明

外形 0.5〜2mm 程度のドーナツ状に整形されたフェライト素子であり、磁化方向により 2 つの安定した状態のいずれかを記憶する。 1970 年代始めまで主記憶の代表素子として使用された。

1951 年 Computer and Control Co.(J.P.Eckert と J.W.Mauchly の作った会社)
商用大型計算機 UNIVAC I を発売

世界最初の商用計算機で、米国の国勢調査のために 1950 年に開発が始まった。事務処理用に設計されたため文字処理機能も備え、数を内部表現するために 10 進法が採用された。内部記憶としては容量 1000 語の超音波水銀遅延線を備え、外部記憶として磁気テープ装置の接続が可能になっている。Computer and Control Co. は後に Remington Rand 社に買収されて Reminton Rand Univac となり、更にこれが UNIVAC, SPERRY, UNISYS へと変わっていった。

1952 年 IBM 社 (米) 商用計算機 IBM701 を発表

世界最初の蓄積プログラム型商用計算機。

1953 年 H.Grosch(米) **グロッシュの法則**を提唱

計算機の性能は価格の 2 乗に比例するという経験則。同じ計算機複数台よりもそれら全ての性能を合わせ持った 1 台の計算機の方が価格が安くなることから、処理の集中化が進んだ。しかし、この法則が有効なのは 1970 年代頃までであり、最近は製造技術の進歩によりこの法則は成立しなくなった。

1953 年 Remington Rand 社 (米) 磁気コア記憶を使用した UNIVAC1103 を発表

1953 年 IBM 社 (米) IBM650 シリーズを発表

世界最初の量産型計算機で、1948 年に開発が始まった。IBM 社としては最初のプログラム内蔵方式の計算機で、数の内部表現に 10 進法が採用された。主記憶として磁気ドラムが標準装備され、後に磁気テープ, 磁気ディスク, 端末の付加が可能となった。

1953 年 MIT(米) 最初の**高速実時間計算機** Whirlwind が完成

1954 年 J.Backus 他 (米 IBM 社)
プログラミング言語 FORTRAN I の設計・開発に着手

世界で最初に実用化された高水準言語であり、1957 年に IBM704 上に処理系 (**コンパイラ**) が完成した。科学技術計算用で、名称は FORmula TRANslation に由来する。

1955 年 J.McCarchy(米 MIT)
“**Artificial Intelligence(人工知能)**” という言葉を用いる

1956 年 岡崎文次 (富士フィルム工業) 日本で最初の電子計算機 Fujic を完成

レンズの光軸計算を高速化するために 1949 年に研究が始まった。2 進法で計算され、記憶部に超音波水銀遅延線が使われた。

第 2 世代 (普及期; データ処理): 計算機を制御／監視するソフトウェア (モニタと呼ばれた) が開発され、運転の自動化が進んだ。色々なプログラミング言語の普及・発展によって一部の計算機専門家以外にも計算機が使える様になり、科学技術計算だけでなく事務処理も行われる様になった。周辺装置としては磁気テープ装置が主流。また、入出力を含めたシステム全体の効率的使用も考えられた。例えば、演算処理と入出力処理を並行に動作させるために入出力チャネルが使われる様になった。

論理素子 (演算部) ···トランジスタ (1948 年に発明されている。動作時間は 1μ 秒〜10n 秒 (i.e.10^{-6}〜10^{-8} 秒))	
記憶部	主記憶 ···磁気コア 補助記憶···磁気ドラム (, 磁気ディスク) 外部記憶···磁気テープ

1957 年 Sperry Rand 社 (米) USSC(UNIVAC Solid-State Computer) を発表

1958 年 J.McCarchy(米) プログラミング言語 LISP を開発

処理系は 1960 年に試作された。名称は LISt Processor に由来する。

1958 年 米軍 航空管制システム SAGE の稼働開始

1950 年に開発が始まったシステムで、オンライン実時間システムの基本例となった。名称は Semi Automatic Ground Environment に由来する。

1959 年 IBM 社 (米) IBM7090 (科学技術計算用)

1959 年 C.Strachey&J.McCarchy(米 MIT) **時分割処理システムの基本概念**を提唱

多くの利用者が通信回線で接続された端末から同時に計算機を対話形式で利用する形態で、TSS(Time Sharing System) と略記される。

1960 年 DEC 社 (米) 初のミニコンピュータ PDP-1 を開発

1960 年 UNIVAC 社 (米) UNIVAC LARC

1960 年 IBM 社 (米) STRETCH

UNIVAC との技術開発競争の結果誕生した計算機で、その開発は 1956 年に始まった。開発プロジェクトの設計目標は、 当時の科学計算用大型計算機IBM704 の 100 倍の性能， 汎用化の追求 (当時は科学計算用と事務計算用の計算機を別々に開発・出荷しており、この 2 種類を統合することは利用者側にも開発者側にも利点が多いと考えた)，の 2 点であった。超高性能化のために、計算機アーキテクチャの革新，多重プログラミングの実現が図られた。 多重プログラミング (Multiprogramming) は 2 つ以上のプログラムを同時に主記憶に入れそれらのプログラムを交互に走らせることによって計算機内の資源の有効利用を図る技法であり、1950 年代の終わりに Honeywell 社の H800 システム用の ARGUS オペレーティングシステムに始まる。

1960 年 IFIP　プログラミング言語 ALGOL60 を制定

手続き的なアルゴリズムの記述に適した高水準言語で、構文記述にバッカス記法 (BNF) を用いて言語仕様を明確に規定した最初の言語。IFIP は International Federation for Information Processing の略称。

1960 年 CODASYL(米)　プログラミング言語 COBOL60 を開発

英文に近い形で事務処理プログラムが書けることを目指した言語で、名称は COmmon Business Oriented Language に由来する。CODASYL は COnference on DAta SYstems Languages の略称。

1961 年 G.Gordon(米 IBM 社)　プログラミング言語 GPSS を開発

離散システム，特に待ち行列型システムをシミュレーションするための言語で、名称は General Purpose Simulation System に由来する。

1961 年 MIT の MAC プロジェクト (米)　時分割処理システム CTSS の開発に着手

30 端末をサポートし、これによって TSS の概念が確立された。ただ、ここでは多重プログラミングを行わずに、サービスを一定時間受けたが入出力待ちになったプログラムを主記憶から追い出し次にサービスするプログラムを主記憶にロードする、という方式をとった。MAC プロジェクトの名称は Machine Aided Cognition と Multi Access Computer System の 2 つの略。

1962 年 IBM 社 (米)　BCDIC 符号を定義

Binary Coded Decimal Interchange Code の略。

1962 年 T.M.Kilburn(英)

仮想記憶 (Virtual Memory) の概念を発表し、英 Manchester 大の ATLAS 計算機 (1959) にページングによる仮想記憶機構を導入

計算機アーキテクチャの提供する論理的 (仮想的) アドレス空間を実際の主記憶のアドレス空間から分離して独立に考えることによって、主記憶領域を大きく見せかける記憶管理方式。

1963 年 ASA(American Standard Association)　**ASCII 符号**を制定

American national Standard Code for Information Interchange の略称。

1963 年 IBM 社 (米)　EBCDIC 符号を定義

Extended Binary Coded Decimal Interchange Code の略。BCDIC 符号 (6 ビット) を 8 ビットに拡張したもの。

第 3 世代 (普及・発展期; 汎用処理): バッチ処理 (i.e. 一括処理) だけでなく実時間処理や時分割処理などにも対応できる汎用オペレーティングシステムが提供される様になり、座席予約, 銀行の預金管理, 在庫管理, 工場の FA 化などの本格的情報処理が可能になったため計算機の利用範囲が一段と広がった。主記憶としては低価格・大容量の磁気コア記憶が使用可能になり、補助記憶としては磁気ディスクや磁気ドラムが標準的に用いられる様になった。

論理素子 (演算部) ···IC	(集積回路 (Integrated Circuit) の略称。トランジスタや抵抗などの回路素子を数十〜数百個集めて 5mm 平方位の (シリコン) チップ内に納めたもの。動作時間は 50n 秒〜1n 秒 (i.e.5×10^{-8}〜10^{-9} 秒))
記憶部	主記憶 ···磁気コア 補助記憶 ···磁気ドラム, 磁気ディスク 外部記憶 ···磁気テープ

1964 年 IBM 社 (米) 汎用計算機 **IBM System 360 シリーズ**を発表

(事務計算, 科学計算, シミュレーションなど, あらゆる問題に対処できることを目指して開発された。小型から大型まで 10 種以上のモデルが 1 つのアーキテクチャ／命令セットで統一された最初のファミリーマシンであり、利用者の要望に合致し一世を風靡した。)

1964 年 アメリカン航空 **(オンライン実時間) 座席予約システム** SABRE によるサービスを開始

(約 1100 の端末から 4 万人分の座席予約を行うシステムで、その名称は Semi Automatic Business environment REsearch に由来する。)

1964 年 日本国有鉄道 (オンライン実時間) 座席予約システム MARS101 によるサービスを開始

1964 年 D.J.Farber(米 AT&T ベル研)
文字列処理用のプログラミング言語 SNOBOL を開発

1964 年 Dartmouth 大 (米) 時分割処理システムが実用化される

1964 年 SRI(米) **マウス**を開発

1965 年 J.G.Kemeny&T.E.Kurtz(米 Dartmouth 大)
対話型プログラミング言語 BASIC を発表

(TSS 環境下で文科系学生対象の計算機教育を目的として設計。名称は Beginner's All-purpose Symbolic Instruction Code の略。)

1965 年 DEC 社 (米) 初の量産ミニコンピュータ PDP-8 を発表

(4K 語の主記憶と入出力装置としてテレタイプがついて 1 万ドルという (当時としては) 破格の値段。特殊な装置をつないで自動的にデータを収集させたり、装置を直接制御させて化学プラントのプロセスオートメーションを図ったりという、(汎用機ではできない) 小回りの利く使い方を求める利用者の支持を受け、工場や研究機関で爆発的に使われた。)

1965 年 IBM 社 (米) プログラミング言語 PL/I を発表

1965 年 MIT の MAC プロジェクト (米) 仮想記憶を本格的に採用した時分割処理システム **MULTICS の構想** を発表

利用効率の向上が第一の時代に利用者へのサービスを中心に考え、セグメンテーション方式とページング方式を組み合わせることによって仮想記憶を実用化したシステムであり、その名称は MULTiplexed Information and Computing Service に由来する。この TSS システムは多重プログラミング, 仮想記憶, ファイルシステムの設計思想, プロセスの概念, 共同利用の際の記憶保護の徹底など、重要な考え方を多く含み、以後の計算機システムに多大な影響を与えた。例えば、このプロジェクトに参加した AR&T ベル研究所は MULTICS が余りにも大規模な汎用ＴＳＳになった反省として 1969 年に UNIX 初版を生み出した。—結局、汎用大型 TSS にアプローチしたその設計思想は優れたものであったが、当時のハードウェア技術からすると先進的すぎたのである。

1965 年 IBM 社 (米)　汎用オペレーティングシステム **OS/360** を発表

System 360 シリーズ用に磁気ディスクをベースに設計・開発され、1966 年に最初の版が利用可能になった。OS/360 は当初バッチ処理と実時間処理用の OS であったが、その後時分割処理用の機能がオプションとして追加されるなど、それまでと比べて機能も規模も格段に大きなものとなった。

1965 年 A.C.Hearn(米 Utah 大)　**汎用数式処理システム** REDUCE を開発

1965 年 J.Lederberg&E.A.Feigenbaum(米 Stanford 大)

エキスパートシステム DENDRAL の開発に着手

有機化合物の構造を決定するためのシステムで、1970 年代の始めに完成。この研究をきっかけとしてエキスパートシステムの開発, 知識工学 (Knowledge Engineering) という研究分野が開けていった。

1965 年 三井銀行　日本初のオンラインバンキングシステムを実用化

1966 年 Automatic Language Processing Advisory Committee(米)

機械翻訳に関する報告書 (ALPAC 報告書) を公表

米国科学アカデミーが 1964 年に組織した委員会において、アメリカの機械翻訳に関する現状と今後の研究援助の方向を様々な角度から分析したもの。この報告書によって機械翻訳は計算言語学の研究に転換すべきであることが示唆され、それまでの 10 年間に渡って 2000 万ドル以上の研究開発費を投じて行われていたアメリカの機械翻訳の研究・開発は中止された。

1967 年 S.Papert(米 MIT)　プログラミング言語 LOGO を開発

(子供が概念を習得する過程をとらえた) 発生的認識論に基づいて心のモデルを計算機上に実現するために考案された言語であり、教育現場では LOGO を使った教育方法の開拓も行われていた。

1968 年 E.W.Dijkstra(蘭)　GoTo 文有害論　**構造化プログラミング**

1968 年 N.Wirth(スイス Federal Institute of Technology,ETH)

プログラミング言語 **Pascal** を設計・開発

手続き的なアルゴリズム記述に適した高水準言語で、ALGOL60 の実質的な後継言語。系統的プログラミング (教育) の容易さ, 計算効率の良さ, 処理系の作り易さなどに重点をおいて設計されたため、データ構造が豊富で構造化プログラミングに適している。(i.e. アルゴリズムを自然に分かり易く記述する能力を持つ。)

1968 年 日本　**郵便番号制度**を実施

1969 年 C.A.R.Hoare(英)　プログラム証明法, 公理的意味論に関する**ホーア論理**を提案

1969 年 K.L.Thompson&D.M.Ritchie(米 AT&T ベル研)

TSS 用のオペレーティングシステム **UNIX** の開発に着手

MIT,GE, ベル研が共同で開発を進めた MULTICS が余りにも大規模な汎用 TSS になった反省から生まれた、柔軟性が高くて使い勝手の良い TSS 用の (小型)OS。 MULTICS の影響が強く出ている。その開発経過は次の通り。1969 年、初版をミニコン PDP-7 上に開発。1971 年、ミニコン PDP-11 に移植 (第 2 版)。1972 年、C 言語とそのコンパイラを開発。1973 年、C 言語を用いて UNIX を全面的に書き換え (第 5 版)。1975 年頃から (第 6 版) 原始コードが外部の大学や研究所にも配布される様になった。1980 年、California 大でスーパーミニコン VAX 用にバークレー版 4.1 が開発される。1983 年、AT&T が UNIX SystemV を発表。

1969 年 米国　世界初の**研究機関間ネットワーク ARPAnet** の運用を開始

国防総省高等研究計画局 (Advanced Research Projects Agency) の支援の下に構築が進められたのでこの名がある。

1969 年 IBM 社 (米)　IBM System 360 モデル 85

初めて**キャッシュ記憶**が採用された計算機。

1970 年 E.F.Codd(米 IBM 社)　**関係データベース**の概念を提唱

1970 年 DEC 社　ミニコンピュータ PDP-11/20 を発表

第 3.5 世代 (高度利用期; 総合的情報処理):　仮想記憶の実用化, データベース技術の発展・普及，計算機同士の結合技術の進展により、計算機の利用は高度化／多様化してきた。また、相互利用を目的とした計算機ネットワークが構築され始めた。

論理素子 (演算部) ･･･LSI	(大規模集積回路 (Large Scale Integration) の略称。回路素子を千～数万個集めて 5mm 平方位の (シリコン) チップ内に納めたもの。動作時間は 10n 秒～数百 p 秒 (i.e.10^{-8}～?$\times 10^{-9}$ 秒))
記憶部	主記憶 ･･･IC 補助記憶･･･磁気ドラム, 磁気ディスク 外部記憶･･･磁気テープ

1970 年 IBM 社 (米)　汎用計算機 IBM System 370 シリーズを発表

System 360 シリーズの後継機種であり、汎用機市場での IBM 社の地位を不動のものにした。 System 370 用の汎用 OS としては、1970 年に SVS が, 1972 年に VM/370 が, 1974 年に MVS が, 1983 年に MVS/XA が用意された。

1970 年 Hawaii 大 (米)　ALOHA ネットワークが稼働

ハワイ諸島を無線で結ぶ計算機ネットワークで、**Ethernet** 発想の原点。

1971 年 N.Wirth(スイス ETH)　**段階的詳細化**によるプログラム作成法を提唱

構造化プログラミングに関する先駆的論文を発表。

1971 年 Intel 社 (米)　4 ビット**マイクロプロセッサ** i4004 を開発

日本企業の要請を受け、計算機の中核である中央処理装置の部分 (i.e. 演算部) を電卓用に LSI 化したもの。

1972 年 A.C.Kay(米 Xerox 社 Palo Alto 研)

オブジェクト指向言語 Smalltalk の設計に着手

優れたマンマシンインターフェースの実現を目指して設計されたプログラミング言語。最初のシステム Smalltalk-72 以来 2 年毎に新しいシステムが開発され、Smalltalk-80 になって初めて公開された。

1972 年 D.M.Ritchie(米 AT&T ベル研) **プログラミング言語 C** を開発

ミニコン PDP-11 上に開発されていた UNIX オペレーティングシステム (アセンブリ言語で記述されていた) を高水準言語で書き換えるために設計された言語。構造的プログラミングのための道具立ても揃っているため科学技術計算のための汎用言語として使うこともできるが、Pascal の様にプログラミング教育を目的に設計された訳でなく、またデータの取扱いに関して低水準言語の性格を持っているため、専門家向けのシステム記述言語という色彩が強い。

1972 年 A.Colmerauer 他 (仏 Marseilles 大)
論理型プログラミング言語 Prolog の処理系を初めて開発

FORTRAN で記述されたインタープリタが開発された。Prolog は元々自然言語処理のために作られた言語であり、その名前は PROgramming in LOGic に由来する。R.A.Kowalski の論文 (1974) で記号論理との関係が明らかにされて以来、論理型プログラミング言語として人工知能分野の研究・開発に広く使用される様になった。

1973 年 Xerox 社 Palo Alto 研 (米) 革新的な計算機 Alto を開発

ビットマップディスプレイ, マウス, ネットワーク機能を有し、その後のワークステーションの原型となった。

1973 年 I.Nassi&B.Shneiderman
流れ図に代わるプログラム図式として NS チャートを提案

1973 年 Intel 社 (米) 8 ビットマイクロプロセッサ i8080 を発表

1974 年 G.Kildall(米 Digital Research 社) パソコン用 OS の CP/M を開発

i8080 を CPU に持つパーソナルコンピュータ用の OS で、その名前は Control Program for Microprocessor に由来する。

1974 年 K.L.Thompson&D.M.Ritchie(米 AT&T ベル研)
TSS 用のオペレーティングシステム **UNIX** を発表

1974 年 日本 **大学間ネットワーク** N-1 ネットワークの開発が始まる

異機種間の研究用広域計算機ネットワークであり、1987 年の時点約 150 台の大学の計算機がつながっていた。

1974 年 R.A.Kowalski(英 Edindurgh 大) **論理型プログラミング**を提唱

1974 年 E.H.Shortliffe 他 (米 Stanford 大)
医療診断用エキスパートシステム MYCIN を完成

代表的なエキスパートシステムの 1 つで、血液感染症, 髄膜炎の診断, 抗生物質の投与を助言する。1970 年に開発が始まった。

1975 年 W.Gates(米) パソコン用 Basic インタープリタを開発し **Micro Soft 社**を設立

Harvard 大に入学中の 19 歳。

1975 年 R.M.Metcalfe&D.R.Boggs(米 Xerox 社)
ローカルエリアネットワーク **Ethernet** を開発

同軸ケーブルをデータ伝送媒体に用いた直列バス型 LAN の製品名。1980 年に Xerox 社,DEC 社,Intel 社が共同で改良型を開発。

1975 年 Cray Research 社 商用**スーパーコンピュータ** Cray-1 を発表

抜きん出て超高速の処理能力を持つ計算機を一般にスーパーコンピュータと呼ぶ。普通は超高速を実現するために (パイプライン方式や多重プロセッサ方式の) 並列処理を行う。Cray-1 の場合は、ベクトル型データをパイプライン方式で並列処理する様に設計され、ピーク性能は 160MFLOPS(mega floating-point operations per second)。1991 年末にはピーク性能が 1TFLOPS(tera —) のスーパーコンピュータ (CM-5, 米 Thinking Macines 社) も出現した。

1976 年 日本科学技術情報センター (JICST)
オンライン文献検索システム JOIS-I によるサービスを開始

科学技術文献情報に関するオンライン検索システムで、その名称は JICST Online Information System の略。

1977 年 Apple 社 (米) **パソコン** Apple II を発表

現在のパソコンの原型。

1977 年 J.Backus(米 IBM 社) 関数型言語 FP を提唱

関数の定義と関数適用の組み合わせによってプログラムを記述するタイプの言語を一般に**関数型言語**という。FP の他に純 LISP や Miranda, ML 等が有名。

1977 年 E.A.Feigenbaum(米 Stanford 大) **知識工学**の名称を初めて用いる

1977 年 N.Wirth(スイス ETH) 汎用システム記述言語 Modula-2 の設計に着手

1979 年に完成。

1977 年 W.N.Joy(米 California 大 Berkley 校) **バークレー版 UNIX** を開発

1982 年 Sun Microsystems 社の設立に加わる。

1977 年 DEC 社 (米) 初のスーパーミニコンピュータ VAX-11/780 を発表

1978 年 東芝 日本初の**日本語ワープロ**を製品化

1978 年 Intel 社 (米) 16 ビットマイクロプロセッサ **i8086** を発表

1978 年 日本規格協会 **漢字 JIS 符号**を制定

1978 年 D.E.Knuth(米 Stanford 大) 文書整形システム **TEX** を発表

第 4 世代 (小型化, 通信との融合が始まった時期; 知識処理, 分散処理): いつから第 4 世代とするかについては共通の認識が得られていない。集積回路技術の進歩により計算機ハードウェアの小型化, 低価格化, 大容量化が進み、パーソナルコンピュータやワークステーションが個人レベルにまで普及し始めた。そして、事務の OA 化，人工知能の応用システムの普及を始めとして計算機が社会のあらゆる面に大きな影響を及ぼす様になった。処理形態としては、様々な計算機を通信回線などで有機的に結合して情報処理を行う、いわゆる**分散処理** (distributed processing) が浸透してきた。

論理素子 (演算部) ···VLSI	超大規模集積回路 (Very Large Scale Integration) の略称。回路素子を数万〜数十万個集めて 5mm 平方位の (シリコン) チップ内に納めたもの。動作時間は数百 p 秒〜数十 p 秒 (i.e.?$\times 10^{-9}$〜?$\times 10^{-10}$ 秒?)
記憶部	主記憶 ···LSI, VLSI 補助記憶 ···磁気ディスク 外部記憶 ···磁気テープ

1979 年 IBM 社 (米) IBM4300 シリーズを発表

1979 年 二村良彦 他 (日立製作所) **PAD** の使用を提案

流れ図に代わってアルゴリズムを系統的に記述するための道具であり、その名前は Problem Analysis Diagram の略。

1979 年 日本電気 8 ビットパソコン PC-8001 を発表

1979 年 W.van Melle 他 (米 Stanford 大)
エキスパートシステム構築ツール EMYCIN を開発

医療診断用エキスパートシステム MYCIN から専門知識を除外して得られたソフトウェア機構であり、MYCIN と同等の知識表現機構, 推論機構を持つ。E は empty の意。

1979 年 ブリックリン (米) 最初の表計算ソフトウェア VISICALC を発表

Apple II 用。

1980 年 IBM 社 (米) IBM308X シリーズを発表

1980 年 J.Ichbiah 他 (米) 汎用高水準言語 Ada を設計・開発

ソフトウェア費用 (特に、古いソフトウェアの保守) の増大に対処するために米国国防総省 (DOD) の主導で開発された。元々、組込み型計算機 (i.e. 武器, 航空機, 船舶, ミサイルなどの電子機械システムに組み込まれた計算機) に入れる制御用ソフトウェア開発のための統一言語である。Ada は Pascal を基礎に設計されたが、その目標である高信頼性プログラミング, 保守容易性, 効率, プログラムの読み易さを実現するために様々な機能 (e.g. モジュール化, 並行処理, 例外処理) を備えた汎用プログラミング言語となった。また、その名前は C.Babbage の解析機関のプログラムについて考察して史上最初のプログラマとなった Ada Augusta (Lovelance 伯爵夫人で詩人 Byron の娘) に因んで付けられた。

1981 年 W.Gates(米 Micro Soft 社) パソコン用 OS の **MS-DOS** を開発

Intel 社 16 ビットマイクロプロセッサ i8088 を CPU に持つ IBM PC 用に開発した。IBM PC 用は PC-DOS と呼ばれた。

1981 年 W.D.Hillis(米 MIT) コネクションマシン構想を発表

1981 年 Sun Microsystems 社 (米) ワークステーション Sun を発表

1981 年 IBM 社 (米) パーソナルコンピュータ **IBM PC** を発表

1982 年 日本電気 16 ビットパソコン **PC-9800** を発表

1982 年 通産省 (日本) 第五世代計算機プロジェクトを発足

10 年計画で、ノイマン型 (i.e. プログラム内蔵方式) に代わる新しい方式の計算機の開発を目指す。このために作られた研究組織が新世代コンピュータ技術開発機構 (Intsitute for new generation COmputer Technology, 通称 ICOT)。

1982 年 J.J.Hopfield(米 California 工科大)
相互結合型ニューラルネットワークのモデルを発表

古典的連想モデルにエネルギー関数を導入してニューロコンピュータの研究が盛んになるきっかけを作った。(久間&中山,1992)

1983 年 S.E.Fahlman,G.E.Hinton&T.J.Sejnowski ボルツマン機械を提案

確率的な動作をする相互結合型ニューラルネットワークのモデル。

1983 年 B.Stroustrup(米 AT&T ベル研)
汎用のオブジェクト指向プログラミング言語 **C++** を開発

オブジェクト指向プログラミングができる様に、C 言語を拡張したもの。

1983 年 新世代コンピュータ技術開発機構 (ICOT, 日本)
世界最初の**逐次型推論マシン** PSI を発表

20MB の主記憶,1200×912 の 17 インチディスプレイ,38MB×2 台の固定ディスク装置を持ち、サイクルタイム 200n 秒 (i.e.2×10^{-7} 秒), 30KLIPS(Logical Instruction Per Second) で処理を行う。DEC-10 Prolog を拡張した言語 KL-0 が機械語に相当する。

1983 年 カポー&ザックス (米) 表計算ソフトウェア Lotus1-2-3 を発表

1984 年 Apple 社 (米) パソコン **Macintosh** を発売

1984 年 日本 **研究者用計算機ネットワーク** JUNET の運用開始

Japan Unix NETwork の略称。

1984 年 坂村健 (東大) **TRON プロジェクト**を発足

The Realtime Operating-system Nucleus の略称。

1985 年 日本 **プログラムの著作権**を保護するため著作権法を改正

1985 年 J.J.Hopfield(米 California 工科大)&D.W.Tank(米 AT&T ベル研)
最適化問題を **Hopfield ネットワーク**を使って解く方法を提案

Hopfield ネットワークを簡単なアナログ電子回路で実現し、この電子回路モデルを用いて巡回セールスマン問題を具体的に解いた。

1985 年 ジャストシステム社 **日本語ワープロソフト一太郎**を発表

1985 年 Sun Microsystems 社 (米) NFS (**Network File System**)

1986 年 日本 **データベースの著作権**を保護するため著作権法を改正

1986 年 D.E.Rumelhart(米 California 大),G.E.Hinton(米 CMU)&R.J.Williams(??)
多層神経回路網モデルに対して**誤差逆伝搬学習法**を提案

多層神経回路網モデルはパーセプトロン (1958,3 層) を一般化したもの。この学習法はパーセプトロンの限界を突破するために考案され、音声認識, 文字認識, ロボット制御を始め様々な応用分野において盛んに用いられている。D.E.Rumelhart は心理学者。

1986 年 MIT(米) **X-Window システム**を開発

1987 年 Microsoft 社&IBM 社 (米) パソコン用 OS の OS/2 を発売

多重タスキングが可能。

1988 年 NeXT 社 (米) ワークステーション NeXT を発表

Apple 社を創設し、Apple I, Apple II, Macintosh の設計・開発に深く関わった S.P.Jobs が設立した会社。

第 5 世代 (インターネット時代, 大衆化):

1988 年 日本 **WIDE プロジェクト**発足

日本でも TCP/IP プロトコルに基づいたネットワークの運用が始まった。

1989 年 ARPAnet が NSFNET に吸収される

NSFNET は元々全米 5 箇所のスーパーコンピュータを接続するために、政府の全米科学財団 (National Science Foundation, NSF) が 1986 年に発足させたネットワーク。その後、NSFNET は NREN(National Research and Education Network) に再編され、全米情報スーパーハイウェイの一翼を担うものとなった。

1991 年 Linus Torvalds(フィンランド) 無償提供の MINIX クローン
1992 年 Linus Torvalds(フィンランド) **Linux0.12**

これ以降様々な人達が Linux の開発に参加

1992 年 MicroSoft 社 (米) Windows3.1
1992 年 **Internet Society** 発足

インターネット技術の進歩や利用に関する国際的な協力・標準化を推し進めるために作られた学会

1992 年 Tim Berners-Lee(欧州共同原子核研究機関 CERN, スイス)
WWW(World Wide Web) を発表
1993 年 日本 WIDE とパソコン通信との電子メール接続、WIDE と商用プロバイダの相互接続
1993 年 NCSA(National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign)
X Window System 版 **Web ブラウザ** Mosaic 1.0

WWW の爆発的な成長をもたらした記念碑的な Web ブラウザ。後の Netscape Navigator へと繋がる。

1995 年 MicroSoft 社 (米) Windows95
1998 年 MicroSoft 社 (米) Windows98

14.5 記憶素子の進歩

IC
LSI
VLSI
} ··· トランジスタ等の半導体素子を多数集めて 5mm 平方位の (シリコン) チップに納めたもの

集積度:
IC ··· 数十〜数百個の素子
LSI ··· 千〜数万個
VLSI ··· 数万〜数十万個

集積することの利点:
①計算機を置く場所が小さくてすむ ⟹ **記憶容量の増大**
②素子間の距離が短くなる ⟹ **計算速度が速くなる**
③消費電力が小さくなる ⟹ 発熱量も小さくなる
④計算機の信頼度が向上

14.6 OS 発展の流れ

オペレーティングシステムはハードウェア技術, ハードウェア価格, 時代の要請に大きな影響を受けながら発展してきた。FORTRAN の開発による利用者の増加は OS の必要性を決定的にした。1960 年代半ば頃までの IC のない時代にはハードウェア (その中でも特に CPU) が高価だったので CPU の有効利用が OS の最重要課題に，それ以後やはりハードウェアが高価だった 1970 年代半ば頃までは計算能力を集中化しシステム全体を有効利用することが最重要課題であったが、それ以降ハードウェア価格の低下に伴い計算能力を分散化し利用者に快適な使用環境を提供することが OS のより大きな課題となってきている。また、CPU の有効利用のための最も有効な手段となっている多重プログラミング (multiprogramming; 同時に複数のプログラムを主記憶に置き並行して CPU に処理させる技術) も、そのアイデアの出された 1950 年代半ばにはハードウェア技術不足のために成功には至らず、磁気ディスクの様な大容量ランダムアクセス (補助) 記憶装置が出現する 1960 年頃まで実現されなかった。 [多重プログラミングの実現は 1950 年代の終わりにハネウェル社の H800 システム用に開発された ARGUS オペレーティングシステムが最初とされている。]

OS 発展のこれまでの歴史は 5 つの時期

$$\left\{\begin{array}{ll} \text{第 1 期} & \cdots \quad \text{OS なしの時代,} \\ \text{第 2 期} & \cdots \quad \text{専用機と専用 OS の時代,} \\ \text{第 3 期} & \cdots \quad \text{汎用機と汎用 OS による集中処理の時代,} \\ \text{第 4 期} & \cdots \quad \text{分散型システムと分散型 OS の時代} \\ \text{第 5 期} & \cdots \quad \text{通信・マルチメディアの時代} \end{array}\right.$$

に分けることができる。それぞれ次の様な時代であった。

(第 1 期:~ 1959)OS なしの時代: OS 発展の最初の芽は 1949 年に稼働した史上初のプログラム内蔵型計算機 EDSAC(英国 Cambridge 大学,M.V.Wilkes) に見ることができる。EDSAC は記号を用いて記述されたプログラムを読み込みそれを機械語に変換しながら主記憶内に蓄えるための初期入力ルーチン (initial program loader) を備えていた。これは、たった 40 語のプログラムであったが、利用者の作ったプログラムと裸の計算機ハードウェアとのギャップを埋めるために開発された歴としたシステムプログラムである。
[EDSAC はプログラム内蔵方式によってもたらされるプログラミングの柔軟性や可能性を世界に広めただけでなく、プログラミングにおける重要な概念をいくつも実現した。例えば、サブルーチンとサブルーチンリンクの標準化, プログラムの共有方法としてのサブルーチンライブラリ, 各種ユーティリティプログラム, 初期入力ルーチン (上記) である。その他、EDSAC は数の表現体系として 2 進補数表示を採用しインタープリタで浮動小数点演算を実現していた。]

その後、商用の計算機として UNIVAC I(1951),IBM701(1952) が現れたが、ここでは「利用者はそれぞれ計算機の利用可能な時間帯が割り当てられ、その時間になると自分自身で自由に計算機を操作する」というオープンショップ (open shop) の利用形態がとられていた。これでは操作卓からプログラムの 1 ステップずつを計算機で実行させながらデバッグを行ったり不慣れな人が計算機を操作することになるため、高価なシステムを非常に効率悪く使っていたことになる。 そこで、使用効率向上のために「利用者はプログラムと

データ (のカード) に操作手順書を添えて専任のオペレータ (operator, 操作員) に渡し処理結果をオペレータから受け取る」というクローズドショップ (closed shop) に利用形態が変更された。クローズドショップの利用形態ではデバッグは処理結果に基づき机上で行われる。

(第 2 期:1955~ 1965) 専用機と専用 OS の時代: この時代まで計算機は科学用と事務用の 2 つに分かれており、同一の会社の製品でもそれぞれ異なる系列のアーキテクチャを持っていた。 例えば IBM702(195?), IBM704(1956), IBM7090(1959), IBM7094(19??), IBM7040(19??), IBM7044(19??) は科学計算用, 1 語=36 ビットのワードマシンであり、また IBM705(1956 頃), IBM7070(1959; 初めて割り込み機能が付いた) は事務計算用のキャラクタマシンであった。 [IBM は 1956 年当時は 704 と 705 を, 1960 年当時は 7090 と 7070 を主力計算機としていた。]

OS も当然各々の計算機アーキテクチャを反映したものになったが、それ以上に OS の性格を決めたのは計算機の (固定された) 利用形態であった。実際、

$$\left\{\begin{array}{l}\text{クローズドショップ処理 (特にバッチによる処理形態),}\\ \text{オンラインリアルタイム処理,}\\ \text{タイムシェアリング処理}\end{array}\right.$$

という 3 つの処理形態に専用の OS がそれぞれ開発され発展していった。従って、この時代には 3 つのほぼ独立した OS 発展の流れができた。この内クローズド処理用の OS 発展の流れの中からはジョブやバッチ処理といった概念が, オンラインリアルタイム処理用の OS 発展の流れの中からは多重プログラミングやタスク, データ通信といった概念が, そしてタイムシェアリング処理用の OS 発展の流れの中からは (タスクの) **スケジューリングアルゴリズム** (scheduling algorithm; 実行待ちのタスクの中から次に実行するタスクを選ぶ方法) や**仮想記憶** (virtual memory), **記憶保護** (memory protection) といった概念が生まれた。

具体的には、クローズドショップ処理用 OS は「操作の自動化による計算機システム利用の効率化」を目指した。その結果、ジョブの概念が生まれバッチ処理という運用形態になったのであった。この時代に開発された OS の主なものを列挙すると次の様になる。

- **IBM701 モニタ**

  世界最初の OS。1955 年にゼネラルモータース (general Mortors) がノースアメリカン航空会社 (North American Aviation Inc.) と共同で IBM701 用に開発した。

- **FORTRAN II モニタ**

  1959 年にノースアメリカン航空会社が IBM704 上の FORTRAN ジョブ専用に開発した。

- **SOS**(SHARE Operating System)

  1960 年頃に SHARE というユーザ団体が IBM7090 用に開発した。使用できる言語は SCAT(SHARE Compiler and Assembler Translator) と呼ばれるマクロアセンフリ言語のみであった。 SHARE(Society for Handling Avoid Repetitional Effort) は IBM 科学計算用大型計算機のユーザ団体で、Boeing Aircraft 社, Lockeed Space and Missile 社, Rand 社 (第二次世界大戦時に OR の分野で米空軍に協力していたグループ), McDonell Douglas 社などが属していた。

- **IBSYS/IBJOB**

1962 年 (稼働は 1964 年?) に IBM 社が IBM7090/7094,7040/7044 用に開発した OS であり、主記憶にその一部が常駐される基本モニタ IBSYS とジョブ管理を行うジョブモニタ IBJOB から成る。IBJOB は IBSYS のサブシステムであり、FORTRAN,COBOL, アセンブリ言語といった複数言語によるプログラミングを可能にするために、その下にコンパイラやアセンブラを持っていた。

また、2 番目のオンラインリアルタイム処理用の OS は多数の通信回線からの入力を同時に (実際には、時分割の様に) 処理しなければならないので、この OS の発展の流れの中からは多重プログラミングやタスク, データ通信といった概念が生まれた。この時期に開発されたオンラインシステムには次の様なものがある。

- **SAGE**(Semi-Automatic Ground Environment)

  1950 年に開発が始まり 1958 年に稼働開始した米軍用作戦指令システムであり、多くの点でオンラインリアルタイムシステムの基本となった。中央の計算機 FSQ/7 と精巧なレーダ, 更にこれらを結ぶリアルタイム通信システムから構成される。レーダは常時北アメリカ大陸全土の上空の状態を調べ、これらのデータを基に中央の計算機は空中戦闘を指揮するために必要なデータを 2.5 秒おきに要員に与える様になっている。 SAGE ではサイクル処理が基本で、あるサイクル内に入力されたデータは次のサイクルで処理される様になっている。当然 1 サイクルの大きさは十分余裕をもってとられ、CPU の処理能力も常に処理量を上回る様にしている。

- **MERCURY 計画用システム**

  MERCURY 計画は米国 (NASA) 最初の人工衛星計画で、人間を宇宙へ送り出そうとするものである。世界中 17 地点とケープカナベラル (現ケープケネディ) 発射場で観測を行い、これらのデータを基に中央の 2 重 (duplex) の IBM7090 が「Go か否か」,「逆噴射ロケットの点火時点」を管制官に示す様になっていた。このシステムでは処理能力を越える入力が一時的にあっても、それに対処できる様に多重プログラミングが行われた。 [しかし、システム内の (タスクでなく) プログラムを並行処理するタイプの多重プログラミングであったので、このシステムでは仮に外部からの 2 つのメッセージがシステム内の同一のプログラムの実行を要求したとしても、これら 2 つのメッセージは並行して処理できなかった。]

- **SABRE**(Semi-Automatic Business environment REsearch)

  約 10 年の歳月を費やしてアメリカンエアライン社が開発した世界最初の大がかりな座席予約システムであり、1964 年にサービスを開始した。約 1100 台の端末機が中央の計算機 IBM7090 につながり、これで約 4 万人分の座席予約ができる様になっていた。このシステムでは MERCURY 計画用システムで同一プログラムに対する要求が並行処理されない点を改善している。すなわち、各端末からの個々の要求を基にタスクを生成しこれらのタスクを並行処理するタイプの多重プログラミング (いわゆる多重タスク処理) を行っている。

3 番目のタイムシェアリング処理は元々「人間と計算機とで協力して問題解決を行うためには?」ということから考え出された利用形態であり、このための OS 発展の流れの中からはタスクのスケジューリングアルゴリズム, **仮想計算機** (virtual machine) といった概念が生まれた。この時期に開発された**タイムシェアリングシステム** (time-sharing system,TSS)/TSS 用プログラミング言語には次の様なものがある。

- **CTSS**(Compatible Time Shared System)

バッチ処理はマンマシンコミュニケーションの欠如のために計算機を用いた教育, 人工知能の研究には向かない。バッチ処理に対するこの様な反省, あるいは「計算機と人間がもっと緊密に協力して問題を解く形式で計算機を使用したい」という願望から、MIT(Massachusetts Institute of Technology) は 1961 年に **MAC**(Machine Aided Cognition/Multiple Access Computer) プロジェクトを発足させた。「人工知能の父」といわれる John McCarthy も参加したこの MAC プロジェクトは SAGE の成果を多く取り入れ、タイムシェアリングシステムの研究・開発を行った。その第 1 段階として作られたのが CTSS である。CTSS は IBM7094(主記憶を通常の 7090 の 2 倍の 64K 語 (36 ビット) に特別使用で拡張したもの) をベースに開発され、最大 30 人の同時利用者を対象に 1964 年から会話型処理のサービスを開始した。そして、このシステムは拡張性の高さにより使用言語が次々と付け加えられていった。 [ただ、ここでは多重プログラミングを行わずに、サービスを一定時間受けたが入出力待ちになったプログラムを主記憶から追い出し次にサービスするプログラムを主記憶にロードする、という方式をとっていた。]

- **JOSS**(JOHNIAC Open Shop System)

  1963 年 Rand 社は JOHNIAC(1953 頃,John von Neumann に因んで名付けられた) という計算機上に小規模の TSS を開発した。このシステムは 8 台のタイプライタ端末に対して卓上計算機と汎用計算機の中間程度のサービスを提供した。プログラムは JOSS という簡単な言語で書かれ、インタープリタによって処理された。

- **BASIC**(Beginner's All-purpose Symbolic Instruction Code)

  1964 年にダートマス大学では GE-235 と DARANET-30 上に 30 台の端末のつながった TSS を開発した。ここでは、初心者でも楽に計算機が使える様に会話型の言語 BASIC が用意され、プログラムはコンパイラで処理された。

- **MULTICS**(MULTiplexed Information and Computing Service)

  1964 年に MIT の MAC プロジェクトの下で研究・開発が開始された TSS システムである。このシステムは多重プログラミング, 仮想記憶, ファイルシステムの設計思想, プロセスの概念, 共同利用の際の記憶保護の徹底など、重要な考え方を多く含み、以後の計算機システムに多大な影響を与えた。例えば、このプロジェクトに参加した AT&T ベル研究所は MULTICS が余りにも大規模な汎用 TSS になった反省として 1969 年に UNIX 初版を生み出した。 [結局、汎用大型 TSS にアプローチしたその設計思想は優れたものであったが、当時のハードウェア技術からすると先進的すぎたのである。]

(第 3 期:1964~ 1980) 汎用機と汎用 OS による集中処理の時代: 第 2 期 (専用機と専用 OS の時代) も終わり頃になると、ハードウェア技術の進歩により個々の計算機の寿命もある程度長くなり、また計算機の利用の仕方 (e.g. 処理形態,FORTRAN や COBOL の利用) についてもかなり定まったものになり、本格的な OS が登場する環境が整ってきた。この様な状況の下で、IBM 社は開発・製造コストの減少,1 台の計算機の多目的利用, プログラムの互換性などの目的のために 1964 年に**汎用計算機** (general-purpose computer; 科学計算にも事務計算にも向く計算機) System 360 を, そして 1965 年に**汎用 OS** (general-purpose OS; 大型から小型まで全てのハードウェアをカバーし、しかもバッチ処理もオンライン処理もタイムシェアリング処理も同時に行える OS) OS/360 の構想を発表した。

それ以後 OS/360 は現在の汎用 OS の基本となり、計算機も OS も汎用化へと進んでいった。そして、この様な汎用化の流れの中からは**仮想計算機の概念** (virtual machine concept; 複数の OS を 1 つの計算機上で並行に動かそうという考え,1972 年に IBM の発表した VM/370 が最初), 更には**多重仮想空間** (multiple virtual storage; 利用者／ジョブ毎に仮想空間を設ける仮想記憶方式, 現在の大型システムで採用している), **多重機密保護** (multiple security; ソフトウェアやデータを本来の所有者以外から保護するための手段が多重になっていること), **データ保全** (data integrity; 共有されるデータの信頼性が高いこ

と) といった概念が生まれた。

補足：
結局、当時のハードウェア規模が小さすぎたため「全ての処理形態をサポートする」という目標は OS/360 では達成できず、タイムシェアリング処理に対しては別に TSS/360 が開発された。

IBM System360 は 1972 年に仮想記憶方式を取り入れた System370 に, 更に 1981 年にアドレス空間を 31 ビット分に拡張した 370/XA になり、これに伴って OS も MVS,MVS/XA にバージョンアップされた。

(第 4 期:1975~ 1990 頃) 分散型 OS の時代: 高度並列計算機 ILLIAC IV の様な特殊機器や各種研究情報を各地の計算機利用者が共同利用することを目的に 1970 年に稼働開始した ARPA ネットワーク (米国国防総省高等計画研究局,Advanced Research Project Ag-ency, の支援の下で構築された) の成功以来、各種の計算機ネットワークが構築される様になった。そして半導体の集積化技術の急激な進歩に伴い、1975 年以降高性能のスーパーミニコンが出現, 更にはワークステーションやパソコンが出現・高性能化・低価格化して、これらのネットワーク化によって計算能力／資源の分散化が急速に進んだ。

この様なネットワーク化／分散化に伴い、当然ネットワーク管理機能の付いた OS が必要になる。この内、ARPA ネットの様にネットワーク上の計算機が各々独立な OS を持ちこれらの OS が他の計算機と通信を行う場合、これらの OS を**ネットワーク OS** (network OS) と言う。また、利用者や応用プログラムにデータや機器の物理的分散を意識させないものを**分散型 OS** (distributed OS) と言う。分散型 OS により、利用者はネットワーク内の計算機全てが 1 つの大局的 (global) な OS に管理されている様に感じる。

補足：
ARPA ネットは最初 4 つのホスト計算機を結んだものであったが、その後ホスト数は増大し (例えば 1977 年で 100 以上) 全米規模のネットワークになった。

分散型 OS の研究は実際のところワークステーションの普及と LAN(local are-anetwork; 数 km の狭い範囲のネットワーク) 技術の発達によって始まった。分散システムの狙いはディスクや高級プリンタ, あるいはデータベースやファイルの共有による資源の有効利用にある。共有可能な資源は LAN 内のどれか 1 つのワークステーションにつなげてそれを共有することになる。しかし、利用者は各々の共有資源の所在場所まで意識せずにワークステーションを使いたいものであり、利用者のこの様な願いをかなえる OS が分散型 OS なのである。

(第 5 期:1990 頃~) 通信・マルチメディアの時代: コンピュータの高性能化・低価格化・使い易さ・利便性が進み、また、インターネットの有用性が一般に認められる様になったので、オフィス, 一般家庭を始め至る所で色々な人がコンピュータを使う様になった。扱えるデータも 音声, 静止画, 動画, ... など、様々である。この様な状況の中では、OS も大規模なネットワークを想定する必要があり、**セキュリティ対策**も必要になる。

15 「プログラミング基礎演習」ガイダンス

15.1 授業／演習の進め方

(1) ガイダンス (1 回)：

- 授業／演習の予定
- 教科書
- 実習室、コンピュータの利用案内
- ユーザ ID について

(2) **UNIX 演習 (3 回)**：
この講義ノートに基づいて、Linux ／ X ウィンドウ／ Emacs エディタの操作法に慣れてもらう。最後にレポート課題が 1 つ用意されている。(このプリントの p.271。)

UNIX/Linux の参考書：
2008 年度までは、次の図書を教科書指定していた。
「九州工業大学情報科学センター (編),
Linux で学ぶコンピュータ・リテラシー, 朝倉書店, 3000 円＋税」
しかし、この演習では UNIX/Linux の一部の機能しか使わないので、この演習で使う事柄だけを簡潔に講義ノートにまとめることにし、上記図書は参考書扱いとします。

UNIX と Linux：
平成 24 年 3 月より教育用コンピュータが新しくなり，情報基盤センター内の実習室では、コンピュータ起動時に OS(Windows か Linux) を選択する DualBoot システムから、基本的には Windows7 を動作させ、Unix 系 OS を使いたい場合は別途運用されている UNIX サーバに接続して使う様にしたシステムに変更されました。**この演習では Linux の一種の RedHat Enterprise Linux 5 を使います。** [Linux は UNIX 系オペレーティングシステムで、中核部分 (カーネル) については UNIX と等しいと考えて構いません。情報基盤センターでは、このカーネルに種々なプログラムを加えたパッケージ (ディストリビューション) として、RedHat Enterprise Linux 5 と呼ばれる製品を導入しています。]

到達度確認:
各演習問題の前にチェックボックス □ を付けました。演習問題が出来たら、チェックマークを入れ次のステップに進んで下さい。

(3) **C プログラミング演習 (11 回)**：
C プログラム／アルゴリズムの設計、実行、デバッグ、等に慣れてもらう。レポート課題も 5 つを予定。(この演習とペアになっている「プログラミング概論」の授業の中で、レポート課題を指定します。)

15.2 教科書について

教科書： 次の 1 冊を教科書として用います。各自購入して下さい。

- **「プログラミング概論」,「プログラミング基礎演習」の講義ノートを生協で印刷して冊子にしたもの (この冊子,1 冊), ????円＋税.**

参考書： この演習のコンピュータ環境として仮定している UNIX/Linux については、プログラミングに関連する事柄を簡潔に講義ノート (この冊子) にまとめるので、特に教科書指定はしません。必要に応じて図書館やインターネットを通じて各自で調べて下さい。ただ、2008 年度まで教科書していた次の 1 冊を参考書として紹介します。

- **九州工業大学情報科学センター (編), Linux で学ぶコンピュータ・リテラシー, 朝倉書店, 3000 円＋税. (UNIX/Linux の参考書)**

この図書は、KNOPPIX と呼ばれる Linux ディストリビューションを仮定しており、情報基盤センターのコンピュータの設定／状況と一致しない所もあり、また、この中に書いてあることが全て必要という訳でもありません。しかし、2008 年度まで教科書として指定していた経緯もあり、以下の講義ノートの各々の箇所で、関連するこの参考書のページ番号等が記載されています。

UNIX/Linux の参考書の中でこの講義に関連するのは次の章です。

1	コンピュータを使う前に
2	はじめて UNIX を使う方へ
3	ファイルとディレクトリ
4	文書の作成
10	UNIX コマンドを使う
11	UNIX におけるプログラミング (特に 11.1〜2 節)

15.3 どこで授業／実習を行うか？

「プログラミング基礎演習」の授業は情報基盤センター (2 食・厚生センターの裏で、生協書籍部と農学部の間の道、あるいは教育学部と 2 食の間の道から入る) の PC 実習室 A で行う。PC 実習室 A に行くには、図 2 の様に

① センター東側脇 (農学部側) の道を通って奥の B 棟まで行き、

② B 棟南側に用意されている学生玄関から入る。そして、

③ スリッパに履き替えて西に向かう。 [階段の向こうに PC 実習室 A。]

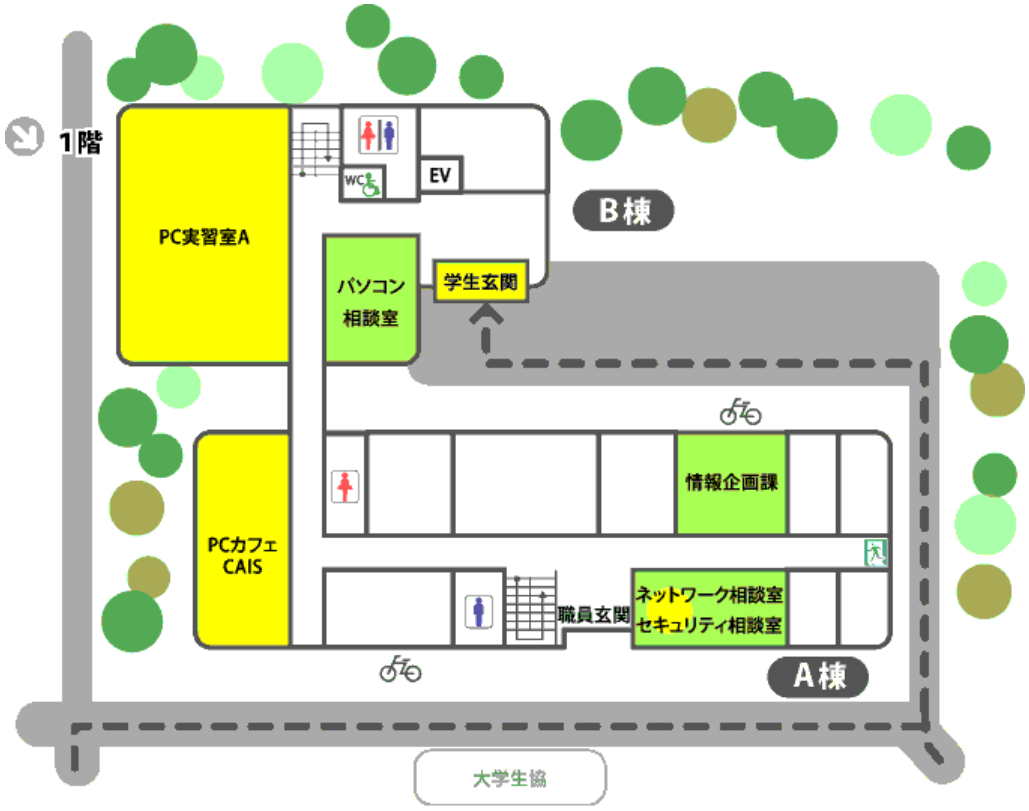


図 2: 情報基盤センター PC 実習室 A

15.4 実習室の利用心得

(1) 実習者用玄関から入る。
(2) 土足厳禁。
(3) 飲食・喫煙禁止。
(4) 他の実習者の迷惑にならない様に。

15.5 いつ実習を行えるか？

授業時間中はもちろん実習ができます。しかし、端末数が十分でない可能性もあり、授業時間内だけでレポート課題で要求されている作業が完了するとは限りません。こんな時は**時間を見つけて各自で実習を行って下さい。** PC 実習室 A だけでなく PC 実習室 B でも Linux 上の実習が可能です。 情報基盤センター PC 実習室 A,B は次の時間帯を除き **月 ~ 金の 8:30~ 19:00** の間は自由に利用できます。

- (履修科目以外の) 授業で使っている時は実習室を利用できません。
  (実習室入口の白板や情報基盤センターのホームページに実習室の利用予定が掲示されます。これらを参考にして下さい。)

15.6 UNIX コンピュータの利用心得

{UNIX/Linux の参考書 1.5 節 }

(1) 実習室では**どのコンピュータを使っても同じ環境**で使えるようになっています。ユーザが作成したファイルは**ファイルサーバ**と呼ばれる大型のコンピュータで管理されており、各コンピュータから操作することができます。毎回同じマシンを使う必要はありません。

(2) 実習室のコンピュータはパーソナルコンピュータなので、基本的に個人で利用しています。作業終了後、正常にシステムを終了し、電源を切断してください。ウィンドウを不必要に多く開いたり、ゲームプログラムを走らせたりすると、中央演算処理装置 (CPU) に大きな負荷がかかることになります。

(3) 個人認証のためにアカウントが必要です。**アカウント**とは、ユーザ名とパスワードとの組で、情報基盤センターの教育用 PC (図書館や総合教育研究棟等にも配備) を使うためのアカウントは、入学者全員に最初から用意されています。アカウントは Windows を使う場合、Linux を使う場合に共通で、パスワードは学務情報システムとも連動しています。ユーザ名は各々の学籍番号 (但し英字は小文字)、パスワードは学生証の左下に書かれている文字列に初期設定されています。

入学してからパスワードをまだ変更していない人は一旦 Windows 上でパスワード変更を行った方が無難かもしれません。変更後のパスワードは忘れないようにして下さい。また、**パスワードを他人に教えたり、管理を疎かにしてはいけません**。定期的にパスワードを変更しましょう。

UNIX 演習

16　とにかく使ってみよう —UNIX 演習 (その 1)—

{UNIX/Linux の参考書第 2 章 }

16.1　キーボードについて

{UNIX/Linux の参考書 2.1 節 }

まとめ：

- Windows の場合と同じ使い方です。

16.2　マウスの基本操作

{UNIX/Linux の参考書 2.3.1 節 }

まとめ：

- クリック、ダブルクリック、ドラッグ など、マウスの基本操作は Windows の場合と同じです。

16.3　システムの起動とログイン

{UNIX/Linux の参考書 2.2 節 }

情報基盤センターでは、Linux オペレーティングシステムの下でコンピュータを使いたい時は、次の (1)〜(6) のようにします。

(1) コンピュータの起動：
ディスプレイの電源 (下部中央付近) を入れてから、コンピュータ本体の電源を入れます。(周辺装置から電源を入れましょう。)

> しばらくすると、画面中央に
> ログオンするには **Ctrl+Alt+Del** を押して下さい
> という指示が現れる。

(2) **Window7 システムへのログインを申し出：**
Ctrl キー, Alt キーを押しながら Delete キー を押します。

> しばらくすると、ユーザ認証のために次の様な表示が画面中央に現れます。
> ユーザ名
> パスワード
> ログオン先：NIIGATAU
> 別のドメインにログオンするには

(3) **Windows7 システムへのログイン手続き** (ユーザ名とパスワードの入力)：
ユーザ認証の表示の中で、「ユーザ名」という文字列の入った箱 [　　] の中にユーザ名 (学籍番号) を小文字で入力し、また「パスワード」という文字列の入った箱 [　　] の中にパスワードを入力して最後に Enter キーを打ちます。
パスワードとしては、

- 過去にセンターの PC/学務情報システムを使ったことがある場合
  ⟶ そこで使っていたパスワードを、
- 過去にセンターの PC/学務情報システムを使ったことがない場合
  ⟶ 学生証の左下に書かれている文字列を

使います。

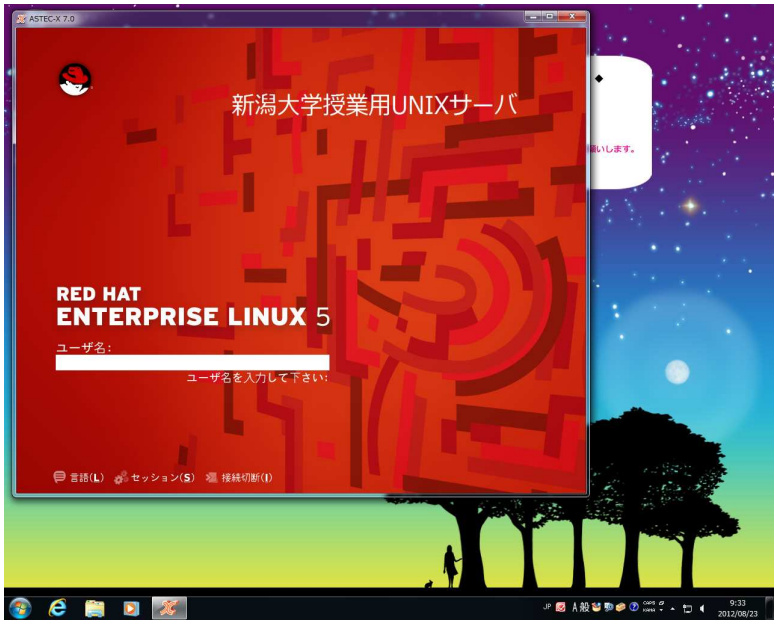
これで、Windows7 の画面が表示されるはずです。

(4) **Linux サーバへの接続：**
Windows 画面左下のスタートボタンを押して現れるメニューから ASTEC-X→ 授業用 UNIX への接続.axp を選ぶ。

補足： ASTEC-X は、Windows 環境上で Unix/Linux 向けの X-Window 環境を提供するための X 端末エミュレータソフトである。ASTEC-X を起動して生成されるウィンドウ内が Linux の環境になる。

操作の様子は次の通り。

これで、次の様な画面になるはずです。



(5) **Linux サーバへのログイン手続き：**

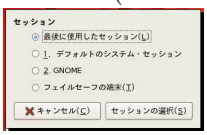
(5.1) 初めての手続き時のみ, 必須 **言語の選択：**
ASTEC-X ウィンドウ左下の「言語 (L))」という文字列をクリックする。すると、次の様なダイアログ (問い合わせ用の小ウィンドウ) が現れる。

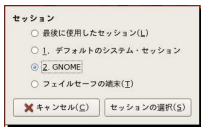
これに対しては、「日本語」という項目を選び、ダイアログ右下の「言語の変更 (L)」ボタンを押す。

> **注意**: この設定をしておかないと、ログイン後にキーボードからの文字入力が思う様にならない。
> ⟹ 最初のログイン時に必ず行う

(5.2) 初めての手続き時のみ セッションの選択：

ASTEC-X ウィンドウ左下の「セッション (S))」という文字列をクリックする。すると、次の様なダイアログ (問い合わせ用の小ウィンドウ) が現れる。





これに対しては、「GNOME」という項目を選び、ダイアログ右下の「セッションの選択 (S)」ボタンを押す。

(5.3) ユーザ名の入力：

Windows7 システムへログオンする際に入力したのと同じユーザ名を、ASTEC-X ウィンドウ左に用意された入力フィールドに入れ、最後に Enter キーを打ち込む。

(5.4) パスワードの入力：

Windows7 システムへログオンする際に入力したのと同じパスワードを、ASTEC-X ウィンドウ左に用意された入力フィールドに入れ、最後に Enter キーを打ち込む。

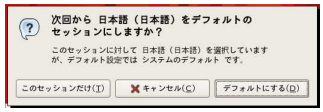
(5.5) 初めての手続き時のみ セッションのデフォルト設定：

上のステップ (5.2) を行なった場合は、次の様なダイアログが現れることがある。

これに対しては、ダイアログ右下の「デフォルトにする (D)」ボタンを押す。

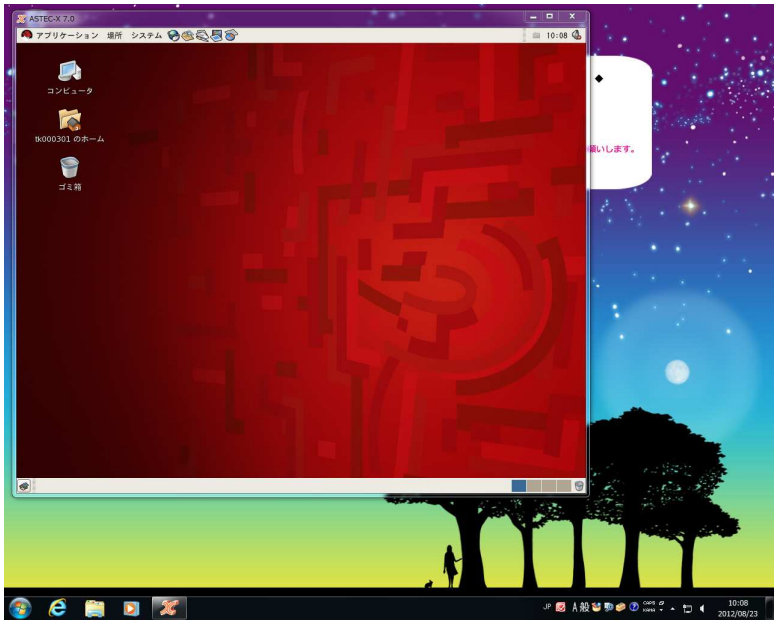
(5.6) 初めての手続き時のみ 言語のデフォルト設定：

上のステップ (5.1) を行なった場合は、次の様なダイアログが現れることがある。



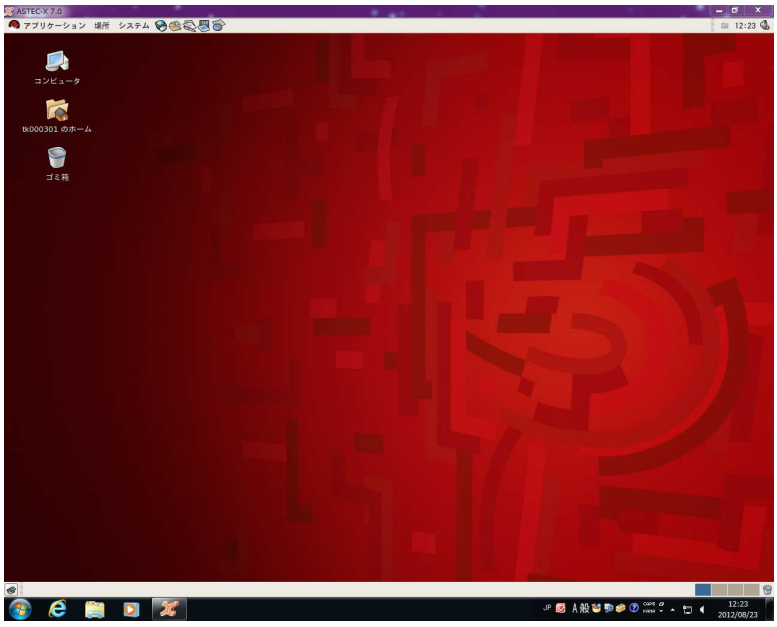
これに対しても、ダイアログ右下の「デフォルトにする (D)」ボタンを押す。

しばらくすると、次の様な画面になるはずです。



(6) **ASTEC-X ウィンドウの最大化：**
ASTEC-X ウィンドウの中で Linux の様々な作業を行うことになるので、右上の最大化ボタンを押すなどして、ウィンドウサイズを調節する。

最大化した場合、次の様な画面になるはずです。



□**演習 16.1** システムを起動し、ログインしてみて下さい。

16.4 ログイン後のマウス操作

{UNIX/Linux の参考書 2.3 節 }

3 ボタンマウスのエミュレート：

- 実習室では Windows 向きのホィール (wheel) 付き 2 つボタンマウスが設置されていますが、UNIX/Linux/X-Window システムでは 3 ボタンマウスを想定しています。 そこで、2 つボタンマウスの下で UNIX/Linux を使う際は、通常、2 つのボタンを同時に押すことで 3 ボタンマウスの真中ボタンを押す操作の代用とします。
- ホィールマウスの左右のボタンの間にある小さなホィールは本来は回して指先一つで画面スクロールするためのものですが、センター実習室の **Linux** 上ではこれを **3** ボタンの真中ボタンとして使えるようです。

Linux メインメニュー: Linux デスクトップ画面の左上にある アプリケーション ボタン, 場所 ボタン, システム ボタンをマウスで左クリックすると、Linux におけるメインメニューが表示されます。例えば、アプリケーション ボタンを押して現れるメインメニューからは、文書作成ツール (Emacs,gedit) やオフィスツール (OpenOffice.org)、Pdf ビューア (AdobeReader9)、Web ブラウザ (Firefox)、ターミナルウィンドウ (GNOME 端末) 等を起動することができます。

例えば、ターミナルウィンドウ (GNOME 端末) の場合は
アプリケーション ⟶ アクセサリ ⟶GNOME 端末
と選択します。 さらに、Emacs テキストエディタの場合は
アプリケーション ⟶ アクセサリ ⟶Emacs テキストエディタ、
Web ブラウザ Firefox の場合は
アプリケーション ⟶ インターネット ⟶Firefox ウェブブラウザ、
メールリーダ Thunderbird の場合は
アプリケーション ⟶ インターネット ⟶Thunderbird Email
と選択します。

□**演習 16.2** メインメニューの項目を確認してみてください。

□**演習 16.3** メインメニューからターミナルウィンドウ (GNOME 端末) を複数開いてみて下さい。

GNOME 端末ウィンドウの選択：

- スクリーン上に開いた GNOME 端末ウィンドウをアクティブな状態にするには、そのウィンドウを左クリックする。
- 選択されてアクティブな状態になった GNOME 端末ウィンドウには、キーボードからの文字入力が可能になる。

□**演習 16.4** スクリーン上に GNOME 端末ウィンドウが複数開いている状態で、これらの GNOME 端末ウィンドウの選択操作を行なってみて下さい。

パネルからのアプリケーションの起動： デスクトップ画面左上には、メインメニューを出すための アプリケーション ボタン, 場所 ボタン, システム ボタンの他にも幾つかのアイコン

が並んでいます。これらをクリックすると、各々対応するアプリケーションが起動されます。

これらのアイコン上にマウスカーソルを移動させると、対応したアプリケーションの簡単な説明が表示されます。

□**演習 16.5** デスクトップ画面左上のアイコンの上にマウスカーソルを移動させ、各々のアイコンがどんなアプリケーションに対応しているか確かめよ。

ルートウィンドウのメニュー： UNIX/Linux の種類によっては、ルートウィンドウでマウスを押し続けることによってアプリケーションの起動や各種設定のためのメニューが現れることがあります。実習室の Linux では、例えば、ルートウィンドウ上で右ボタンを押して現れるメニューから、GNOME 端末を開いたり、背景変更したり出来る様になっています。

□**演習 16.6** センター実習室で、マウスの左ボタン, 中ボタン, 右ボタンによってそれぞれどのようなルートウィンドウメニューが現れるか確認せよ。

16.5 ターミナルウィンドウの使い方

{UNIX/Linux の参考書 2.4 節 }

ターミナルウィンドウ上でのコマンド入力：

- UNIX では直接**コマンド**と呼ばれる文字列を入力することにより、種々の細かな作業を行なうことが可能です。これらのコマンドを入力するためのウィンドウを**ターミナルウィンドウ** (**仮想端末**) と呼びます。
- 実習室の RedHat Enterprise Linux5 では、GNOME 端末ウィンドウがターミナルウィンドウに当たります。
- 既に 16.4 節で見たように、実習室でターミナルウインドウを表示するには、
  ◇ メインメニューから「GNOME 端末」を選んだり、
  ◇ デスクトップ上部のメニューバー上の GNOME 端末のアイコンをクリックしたりします。 (デスクトップ上部のメニューバーに GNOME 端末のアイコンを出すには、メインメニューから「GNOME 端末」を選びドラッグ操作でメニューバー上に落とせばよい。)

□**演習 16.7** ターミナルウィンドウが画面上にない場合は、開いて下さい。そして、ターミナルウィンドウ上で `cal` Enter とコマンド実行してみて下さい。 (1 ケ月分のカレンダーが表示されたはずです。)

UNIX コマンドの形式： コマンドの一般的な入力形式は次のようになります。

$ コマンド名 -オプション ... -オプション 引数... 引数

ここで、最初の "$" はコマンド入力のプロンプトを表します。オプションはコマンドの動作を変更するもので、必要に応じて設定します。引数はコマンドの実行に必要な情報を

指定するものです。

コマンドの強制終了：
コマンドの実行途中で障害が発生した場合、コマンドを強制終了することができます。この場合は Ctrl-c、すなわち Ctrl キーを押しながら c キーを打って下さい。また、コマンドがユーザの入力を待っている場合があります。この場合は Ctrl-d、すなわち Ctrl キーを押しながら d キーを打って下さい。

□演習 16.8 ターミナルウィンドウ上で次のコマンド入力を行なってみて下さい。

(1) `cal 12 2015` Enter (2015 年 12 月のカレンダーを表示)
(2) `cal 2015` Enter (2015 年のカレンダーを表示)
(3) `cal -3` Enter (先月～来月のカレンダーを表示)
(4) `cal -m` Enter (週の最初の曜日を月曜として今月のカレンダーを表示)
(5) `cal -y` Enter (今年 1 年のカレンダーを表示)

□演習 16.9 新しいターミナルウィンドウは、メインメニュー (やデスクトップ画面下のボタン) からだけでなく、ターミナルウィンドウ内で `gnome-terminal&` とコマンド入力することによっても開くことが出来ます。お試し下さい。(コマンド名は `gnome-terminal` の部分だけ。コマンドの最後に `&` を付けると、一般にコマンドを実行したターミナルウィンドウと並行してコマンドが実行されることになります。その結果、コマンド実行の終わりを待つことなく、元のターミナルウィンドウ上に次のコマンドを打ち込むことができます。)

□演習 16.10 (利用状況を表示するコマンド) 次の `whoami` コマンド, `hostname` コマンド, `date` コマンドをそれぞれ実行してみて下さい。[それぞれのコマンド名を入力するだけ。]

- `whoami` ... 利用者のユーザ id を表示
- `hostname` ... 使っている計算機の名前を表示
- `date` ... 日付と時間を表示

過去に入力したコマンドの再利用： Ctrl-p や ↑ キーを押すことによって、過去に入力したコマンドをコマンドライン上に表示することができます。

キー操作	動作
Ctrl-p または ↑	1 つ前のコマンドを表示
Ctrl-n または ↓	1 つ後のコマンドを表示
! *str*	*str* を先頭文字列にもつ最新コマンドを再実行
! !	直前のコマンドを再実行

□演習 16.11 (コマンドの再利用) コマンドライン上で Ctrl-p や ↑, Ctrl-n, ↓ を押して、過去に入力したコマンドを再実行してみて下さい。

オンラインマニュアル： UNIX/Linux にはオンラインマニュアルが備わっていて、うろ覚えのコマンドがあった場合でも、その使い方を画面に表示することができます。

形式:	**man** [options] *string*	
options:	-k	: 全てのコマンドの要約説明から *string* を含むものを表示
	-f	: *string* に関する要約説明を表示
	その他	オンラインマニュアルを御覧下さい。(man man)

説明:	指定したコマンド等のオンラインマニュアルを表示。 最初のページが表示された後、次のキーで表示内容を切り替えることができる。(more コマンドの場合と同じキー。) Space 次の画面を表示。 b 1 つ前の画面を表示。 q 表示を終了。
使用例:	man コマンド名 man -k キーワード

□**演習 16.12 (man コマンド)** man コマンドを使って、cal コマンド, whoami コマンド, hostname コマンド, date コマンド, man コマンド が何をするコマンドであるか調べてみて下さい。

16.6 ファイルを作ってみる

{UNIX/Linux の参考書 2.5 節 }

ファイル操作のための基本コマンド：

- **コマンド実行の結果をファイルに保存するには、**
  コマンド > ファイル名 とコマンド入力します。
  (**標準出力のリダイレクション**と呼びます。)
- (カレントディレクトリ内の) **ファイルを一覧表示するには、**
  ls とコマンド入力します。
- **ファイル内容を表示するには、**
  cat ファイル名 とコマンド入力します。

□**演習 16.13 (ファイル操作の例)** 上の記述を参考にして、次の (a)〜(c) のファイル操作を順に行なってみて下さい。
(a) 今年のカレンダーを calendar という名前のファイルに保存する。
(b) calendar という名前のファイルが本当に出来ているかどうか確認する。
(c) calendar という名前のファイルの中身を確認する。

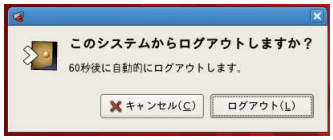
16.7 ログアウトとシステムの終了

{UNIX/Linux の参考書 2.6 節 }

情報基盤センターでは、Linux の利用終了時は次の (1)〜(6) のようにします。

(1) **Linux サーバからのログアウトを申し出：**
Linux デスクトップ画面左上の[システム]ボタンを押して現れるメニューから「[ユーザ名]のログアウト...」という項目を選びます。

これによって、次の様なダイアログ (確認のための小窓) が画面中央に現れます。

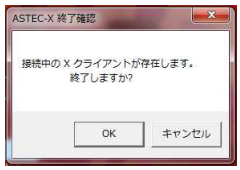


(2) **Linux サーバからのログアウト：**
表示されたダイアログ右下の「ログアウト (L)」というボタンを押す。

これによって、Windows 画面左下のスタートボタンで ASTEC-X→ 授業用 UNIX への接続.axp と選んで現れたのと同じ、Linux のログイン受付のウィンドウに戻る。

(3) **Linux サーバとの接続を解除：**
Linux へのログインを受け付けている (ASTEC-X) ウィンドウ右上の☒ボタンを押して、ウィンドウを消す。

これによって、次の様なダイアログ (確認のための小窓) が画面中央に現れます。



これに対しては、ダイアログ中央下の「OK」ボタンを押す。

補足：ウィンドウ下部中央付近に「接続切断」ボタンも用意されているが、押して切断してもすぐ元に戻る。

(4) **他のシステム利用者が待っている場合は、**
Windows 画面左下のスタートボタンを押して現れるメニューから「ユーザーの切り替え」または「ログオフ」を選びます。そして、しばらくして画面中央に

「ログオンするには Ctrl+Alt+Del を押して下さい」

という表示が現れたことを確認した上で、待っている人に席を譲ります。

席を譲られた人は、16.3 節の起動・ログイン手続きのステップ (2) から始めます。

(5) **コンピュータの停止：**
他にシステム利用者がいない場合は、Windows 画面左下のスタートボタンを押して現れるメニューから「シャットダウン」を選びます。

これで、Windows7 も終了し、**自動的にコンピュータ本体の電源が切れます。**

(6) コンピュータ本体の電源が切れたら、**ディスプレイの電源 (ボタンは下部中央付近に**

ある) を切って下さい。

□**演習 16.14** ログアウト (とコンピュータの停止) を行ってみて下さい。

16.8 パスワードの変更について

{UNIX/Linux の参考書 1.5 節 }

UNIX 等のマルチユーザオペレーティングシステムでは、ユーザ名とパスワードとによってユーザ認証します。他人に不正利用されないように、頻繁にパスワードを変更しましょう。パスワードを変更しないと自分だけではなく、他人にも迷惑をかける恐れがあります。情報基盤センターの PC に関しては、パスワード変更は情報基盤センターの Web ページ

`http://www.cc.niigata-u.ac.jp/security/password/`

の中の「NIIGATA-U アカウントのパスワード変更はここをクリック」という箇所を左クリックし指示に従うことによって行います。

注意：
パスワードを紙等にメモした場合には、その紙を不注意に放置しないで下さい。

UNIX/Linux の参考書 14 ページ下段の脚注の中で紹介されている `passwd` コマンドや `yppasswd` コマンドは利用できません。

16.9 X ウィンドウの基本操作

{UNIX/Linux の参考書 2.7 節 }

X Window System：
UNIX が開発された当初 (1969 年〜1980 年代中頃) は、UNIX のユーザーインターフェースはコマンドラインのみしかありませんでした。その UNIX に **GUI**(Graphical User Interface) を導入するために開発されたのが **X Window System** (以下 **X** と表記) です。SunView、GWM などといった X 以外の Window System も存在しましたが、現在生き残っているのは X のみといって良いでしょう。 [但し、X は GUI を作成するためのプロトコルのみの規格であり、実際のインターフェース (ウィンドウのどこにどのようなボタンを配置するか、マウスのどのボタンをクリックしたら何が起るか等) はベンダーやユーザが自由に作ることができます。]

□**演習 16.15 (ウィンドウに付属の操作ボタン)** それぞれのウィンドウの右上部には 3 つのボタン (最小化ボタン、最大化ボタン、強制終了ボタン) が付いています。 このボタンを押すことによって、どんな処理が実行されるかを確認してみて下さい。

□**演習 16.16 (ウィンドウサイズの変更)** マウスカーソルはウィンドウの外枠上に置くと形が両矢印に変わります。この状態で左ドラッグすることにより、ウィンドウの大きさを変更できます。GNOME 端末 ウィンドウで試してみて下さい。

□**演習 16.17 (ウィンドウの移動)** ウィンドウ上部のタイトルバーをマウス左ボタンでドラッグすることによって、ウィンドウを移動してみて下さい。

16.10 コピー・アンド・ペースト

{UNIX/Linux の参考書 2.8 節 }

マウス操作によるコピー・アンド・ペースト： 次の様にすることで、簡単に (ターミナルウィンドウ等の中に表示されている) 文字列を別の場所にコピーすることができます。

(1) **文字列を** (一時的な作業領域に) **コピー：**
  (1.1) コピーしたい文字列の入っているウィンドウを選択。
  (1.2) マウスカーソルを文字列の最初の位置に合わせて、左ボタンを押す。
  (1.3) マウスの左ボタンを押したままマウスカーソルを文字列の最後の位置まで動かし、目的の文字列が反転表示された状態で、マウスボタンを離す。

(2) (一時的な作業領域内の) **文字列を目的の場所に張り付け：**
  (2.1) 張り付け先のウィンドウを選択。
  (2.2) マウスカーソルを張り付け先の位置に移動。
  (2.3) マウスの真中ボタンを押す。

□**演習 16.18** 上の記述に従って、コピー・アンド・ペーストの機能を使ってみて下さい。例えば、

(a) `cal 10 2012` とコマンド入力して、2012 年 10 月のカレンダーを画面に表示する。
(b) コピー・アンド・ペーストの機能を使ってもう一度コマンドラインに `cal 10 2012` と表示させ、実行する。

16.11 スクロールバー操作

{UNIX/Linux の参考書 2.9 節 }

ターミナルウィンドウ内の過去の実行履歴： GNOME 端末ウィンドウの右側にはスクロールバーが付いていて、

- マウスドラッグでスライダー部を上下させたり、
- スクロールバー上部の△を左クリックしたり

することにより、ターミナルウィンドウ内の過去の実行履歴を表示することができます。

□**演習 16.19** GNOME 端末ウィンドウに付属のスクロールバーの操作を行ってみて下さい。

17 UNIX 入門 —UNIX 演習 (その 2)—

{UNIX/Linux の参考書第 3 章 }

17.1 ファイルとディレクトリ

{UNIX/Linux の参考書 1.2.3 節, 第 3 章 }

現在使われている一般的なオペレーティングシステムでは、**ファイル**という単位で情報を保存します。作成した文書も、プログラムも、すべてオペレーティングシステムから見ればファイルとして扱われます。ですので、ファイルの操作（ファイルの新規作成、削除、内容の変更、名前の変更等）はコンピュータ操作の基本です。

ファイル名：

- ファイル名の構成要素としては主に英数字を使う。
- 大文字と小文字は区別する。
- 次の文字は特殊な用途に使うので、ファイル名には使わない。

  `/ * ? ' " ; & ( ) | < > \ [ ] ! { }`
- ファイル名がピリオドで区切られ最後のピリオド以降がファイルの種類を表すことがある。この場合、この最後のピリオド以降の部分を**拡張子**という。例えば次の通り。

拡張子	ファイルの種類
`.c`	C 言語のソースファイル
`.h`	C 言語のヘッダファイル
`.o`	オブジェクトファイル
`.tex`	LaTeX のソースファイル
`.ps`	Postscript 形式の文書ファイル
`.pdf`	pdf 形式の文書ファイル
`.png`	PNG(Portable Network Graphics) 形式の画像ファイル
`.gif`	GIF 形式の画像ファイル
`.docx`	Microsoft Word の文書ファイル
`.xlsx`	Microsoft Excel のファイル
`.pptx`	Microsoft Powerpoint のファイル
`.txt`	Text 形式の文書ファイル
`.tar`	tar 形式のアーカイブファイル
`.gz`	Lempe–Ziv(LZ77) アルゴリズムに基づく圧縮ファイル

- 名前がピリオドで始まるファイルは**ドットファイル**と呼ばれ、利用者の環境設定情報を保管するなどの用途に使われる。こうしたファイルは単に `ls` とコマンド入力しただけでは表示されません。

ファイルの階層的な保管：

- Windows の場合と同じで、ファイルは階層的に保管される。
- Windows で「フォルダ」と呼ばれているものは、UNIX/Linux では**ディレクトリ** と呼ばれています。

- 階層構造の最上位に位置するディレクトリを**ルートディレクトリ** と呼び、/ と表します。
- 例えば、ルートディレクトリの中に home00 というディレクトリがあり、その中に t00i000a というディレクトリがあり、その中に c-lab というディレクトリがあり、その中に abc.c というファイルがあった場合、このファイルを識別するのに　/home00/t00i000a/c-lab/abc.c　という書き方ができます。この種の書き方で表されたファイル名を **絶対パス名** と言います。

ファイル位置の相対的な指定：

- ターミナルウィンドウ上でコマンド指示等を行なっている際は、ファイル操作はファイルシステム内の特定のディレクトリ内に集中していることが多い。そこで、利用者が現在作業しているディレクトリは **カレントディレクトリ** と呼ばれ、ターミナルウィンドウ上で (暗黙に) 仮定/認識される様になっている。
- ログイン直後等に暗黙に仮定されるディレクトリは **ホームディレクトリ** と呼ばれ、~　と表されます。
- カレントディレクトリは明示的に　.　と表されます。
- カレントディレクトリより 1 つ上位のディレクトリ (i.e. カレントディレクトリを要素として含むディレクトリ) は明示的に　..　と表されます。
- 例えば、カレントディレクトリが /home00/t00i000a/c-lab であった場合、ファイル /home00/t00i000a/c-lab/abc.c を指定するのに　abc.c や ./abc.c や ../c-lab/abc.c　といった書き方ができます。この様に、カレントディレクトリを始点とした書き方で表されたファイル名を **相対パス名** と言います。

ディレクトリ名	意味
/	ルートディレクトリ
.	カレントディレクトリ
..	カレントディレクトリの親ディレクトリ
~	ホームディレクトリ

17.2 小さなテキストファイルの新規作成

{UNIX/Linux の参考書 10.3.1 節, 10.2.2 節, 10.1.1 節 }

テキストファイルの新規作成には普通エディタを用いますが、小さなテキストファイルを作る場合は、ファイルの内容を表示するためのコマンドである cat と UNIX に備わった「入出力リダイレクション」機能 (UNIX/Linux の参考書 10.1.1 節, 10.2.2 節) を使って次のように行なうことができます。

```
$ cat > 新規ファイルの名前 Enter
┌──────────────────────────────┐
│                              │
│   新規ファイルの中身をそのまま打ち込む   │
│                              │
└──────────────────────────────┘
Ctrl-d  ..... (Ctrlキーを押しながらdキーを押す)
```

□**演習 17.1** コマンド `cat > text1` を実行してアルファベット 26 文字からなるファイル `text1` を作ってみて下さい。

17.3 ファイルとディレクトリの基本操作

{UNIX/Linux の参考書 10.3.1〜2 節, 第 3 章 }

ファイルやディレクトリを操作するために、次の様なコマンドが用意されています。(以下のコマンド形式の記述の中の `[...]` は、この部分が省略可であることを表します。)

ディレクトリやファイルの属性情報を表示

形式:	**ls** `[options]` `[` *files/dirs* `]`	
`options:`	`-a`	: ドットファイルも含めて表示。
	`-l`	: ファイルやディレクトリの名前だけでなく、それらに付属の詳細な属性情報も表示する。
	`-R`	: サブディレクトリ内のファイル群も再帰的に全て表示する。
	その他	オンラインマニュアルを御覧下さい。(`man ls`)

説明:
- 引数部 (*files/dirs*) が省略された場合は、カレントディレクトリ内のファイルを一覧表示する。
- 引数部 (*files/dirs*) にディレクトリ名が指定された場合は、そのディレクトリ内のファイルを一覧表示する。
- 引数部 (*files/dirs*) にファイル名が指定 (ワイルドカードを含む指定も可) された場合は、指定に該当するファイルを一覧表示する。

テキストファイルの内容を表示

形式:	**cat** `[options]` `[` *files* `]`	
`options:`	`-n`	: 全ての行を行番号付きで表示。
	その他	オンラインマニュアルを御覧下さい。(`man cat`)

説明:
- 引数部 (*files*) が指定された場合は、指定されたファイルの内容を表示する。
- 引数部 (*files*) に複数のファイル名が空白で区切られて指定された場合は、ファイル内容を指定順に続けて表示する。
- 引数部 (*files*) が省略された場合は、標準入力ストリームを指定ファイルとみて処理する。

テキストファイルの内容を行番号付きで表示

形式:	**nl** `[options]` `[` *file-name* `]`	
`options:`	`-b` *type*	どの行に番号を付けるかを指定します。 指定できる *type* とその意味は次の通り 。 `a` ··· 全行に番号付け。 `t` ··· 印書可能なテキストのある行だけに番号付け。(デフォルト) `n` ··· 番号づけをしません。 `p` *exp* ··· *exp* という正規表現パターンに合致する行だけに番号付け。
	その他	オンラインマニュアルを御覧下さい。(`man nl`)

テキストファイルの内容を画面単位に表示

形式:	**more** [options] [*files*]
options:	オンラインマニュアルを御覧下さい。(man more)

説明: 内容を 1 画面分表示後に一時停止する。
これ以降は例えば次のコマンドで表示内容を上下させることができる。

Space	次の画面を表示。
b	1 つ前の画面を表示。
Enter	1 行分先に進める。
q	表示を終了。
h	可能なコマンドの説明。

ファイルのコピー

形式:	**cp** [options] *sourcefile destinationfile*	
	cp [options] *sourcefiles destinationdir*	
	cp [options] *sourcedir destinationdir*	
options:	-i	: コピー先 (*destination...*) にファイルが既に存在する場合、その度に上書きするかどうかの確認が為される。
	-R	: コピー元とコピー先の両方にディレクトリが指定された場合、コピー元のディレクトリ (*sourcedir*) 内のファイル群を全て再帰的にコピー先のディレクトリ (*destinationdir*) 内にコピー。
	その他	オンラインマニュアルを御覧下さい。(man cp)

説明:
- *source...* で指定されたファイル (群) を *destination...* という先にコピーする。
- コピー先 (最後の引数) にディレクトリが指定された場合は、複数のファイル名を空白で区切って並べたり、* や ? といったワイルドカードを使ったりすることで、複数のファイルをコピー元 (*source...*) として指定することもできる。

ファイルの移動

形式:	**mv** [options] *sourcefile destinationfile*	
	mv [options] *sourcefiles destinationdir*	
	mv [options] *sourcedir destinationdir*	
options:	-i	: 移動先 (*destination...*) にファイルが既に存在する場合、その度に上書きするかどうかの確認が為される。
	その他	オンラインマニュアルを御覧下さい。(man mv)

説明:
- *source...* で指定されたファイル (群) を *destination...* という先に移動する。
- 同一ディレクトリ内での移動の場合、単にファイルやディレクトリの名前変更として機能します。
- 移動先 (最後の引数) にディレクトリが指定された場合は、複数のファイル名を空白で区切って並べたり、* や ? といったワイルドカードを使ったりすることで、複数のファイルを移動元 (*source...*) として指定することもできる。

ファイルの削除

形式:	**rm** [options] *files* **rm** [options] *dirs*
options:	-i : 削除するファイル毎に、本当に削除して良いかどうかの確認が為される。 -r : ディレクトリを (中のファイル群と共に) 削除したい場合に指定する。 その他 オンラインマニュアルを御覧下さい。(man rm)

説明:
- 引数で指定されたファイル (群) を全て削除する。
- 引数部には、複数のファイル名を空白で区切って並べたり、* や ? といったワイルドカードを使ったりすることで、複数のファイルを指定することもできる。

カレントディレクトリの表示

形式:	**pwd**

説明: カレントディレクトリを表示する。

ディレクトリの作成

形式:	**mkdir** [options] *dirs*
options:	-m *mode* : 作成するディレクトリの保護モードを *mode* に設定する。 その他 オンラインマニュアルを御覧下さい。(man mkdir)

説明:
- 引数で指定されたディレクトリを作成する。
- 引数部には、ディレクトリ名を空白で区切って並べることによって、複数のディレクトリを指定することもできる。

ディレクトリの削除

形式:	**rmdir** [options] *dirs*
options:	-p : 指定パスに含まれるディレクトリの内、子ディレクトリを削除して空になるディレクトリは全て削除する。 その他 オンラインマニュアルを御覧下さい。(man rmdir)

説明:
- 引数で指定されたディレクトリについて、空なら削除する。
- 引数部には、ディレクトリ名を空白で区切って並べることによって、複数のディレクトリを指定することもできる。

カレントディレクトリの移動

形式:	**cd** [options] [*dir*]
options:	オンラインマニュアルを御覧下さい。(man cd)

説明:
- カレントディレクトリを引数で指定されたディレクトリに移動させる。
- 引数が省略された場合は、ホームディレクトリに移動する。

□**演習 17.2** 次の (a)〜(e) のファイル／ディレクトリ操作を順に行なってみて下さい。

(a) 自分のホームディレクトリにファイル text1 が存在することを確認する。[ファイル text1 が見当たらない場合は演習 17.1 に戻る。ヒント：ls]

(b) ファイル text1 の内容を確認する。[cat, more, less]
(c) ファイル text1 をファイル text2 にコピーする。
ファイル text2 のファイル名を text3 に変更する。
ファイル text3 を削除する。[ヒント：cp, mv, rm]
(d) cat コマンドを用いて次の 3 行からなるテキストファイル text2 を作成する。

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890,.;:+-=_ (){}[]<>/\?|"'‘~!@#$%^&*
```

(e) ディレクトリ box を作成、text2 を box に移動させ、確認する。そして，ディレクトリ box を削除する。[ヒント： mkdir, rm -r, rmdir, pwd, cd]

17.4 ファイル管理

{UNIX/Linux の参考書 10.1.2 節, 10.3.1 節, 10.3.3 節 }

ファイルの保護モード：

- UNIX/Linux では、各々のファイルやディレクトリについて
  - ◇ 所有者が内容を読めるかどうか,
  - ◇ 所有者が内容を変更できるかどうか,
  - ◇ 所有者が実行 (一般ファイルの場合)/ 検索 (ディレクトリの場合) できるかどうか,
  - ◇ 同じグループのユーザが内容を読めるかどうか,
  - ◇ 同じグループのユーザが内容を変更できるかどうか,
  - ◇ 同じグループのユーザが実行 (一般ファイルの場合)/ 検索 (ディレクトリの場合) できるかどうか,
  - ◇ 無関係の他ユーザが内容を読めるかどうか,
  - ◇ 無関係の他ユーザが内容を変更できるかどうか,
  - ◇ 無関係の他ユーザが実行 (一般ファイルの場合)/ 検索 (ディレクトリの場合) できるかどうか

  が設定されています。これらのアクセス権の情報のことを**保護モード**と呼んでいます。
- ファイルやディレクトリの保護モードは、-l というオプション付きで ls コマンドを実行することにより容易に確認できます。すなわち、ターミナルウィンドウ上で

  ```
  ls -l [ ファイル名/ディレクトリ名 ]
  ```

  というコマンドを実行した時に、各行の
  - ◇ 1 文字目はファイルの種類 (d ならディレクトリ, - なら一般ファイル) を表し,
  - ◇ 2 文字目は所有者の可読権の有無 (r なら有, - なら無) を表し,
  - ◇ 3 文字目は所有者の書込み権の有無 (w なら有, - なら無) を表し,
  - ◇ 4 文字目は所有者の実行/検索権の有無 (x なら有, - なら無) を表し,
  - ◇ 5 文字目は同じグループユーザの可読権の有無 (r なら有, - なら無) を表し,
  - ◇ 6 文字目は同じグループユーザの書込み権の有無 (w なら有, - なら無) を表し,
  - ◇ 7 文字目は同じグループユーザの実行/検索権の有無 (x なら有, - なら無) を表し,
  - ◇ 8 文字目は無関係の他ユーザの可読権の有無 (r なら有, - なら無) を表し,
  - ◇ 9 文字目は無関係の他ユーザの書込み権の有無 (w なら有, - なら無) を表し,
  - ◇ 10 文字目は無関係の他ユーザの実行/検索権の有無 (x なら有, - なら無) を表し

ます。

- **注意：** ファイルへのアクセス権が設定されていたとしても、そのファイルの属する上位のディレクトリの探索権が設定されていないと、実際にはそのファイルを使うことができません。

□演習 17.3 (保護モード等の確認) ホームディレクトリにあるファイルやディレクトリの保護モード, 所有者, 大きさ, 作成日などがどうなっているか、`ls` コマンドで調べてみて下さい。

ファイルやディレクトリの保護モードを変更するためのコマンドとしては、次の様なものがあります。

保護モードの変更

形式:	**chmod** `[options]` *mode files/dirs* **chmod** `[options] [ugoa][+-=][rwxst]` *files/dirs*
`options`:	`-R` : 指定したディレクトリ下の全てのファイル/ディレクトリ群を対象とする。 その他 オンラインマニュアルを御覧下さい。(`man chmod`)
mode:	所有者についての権限, 同じグループのユーザについての権限, 無関係の他ユーザについての権限のそれぞれを 0〜7 の数値で表し、それらを並べた形で、変更後の保護モードを明示したもの。ここで、個々のユーザについての権限は、ls コマンドの表示の `---` を整数 0 に対応させ、同様に `--x` を 1 に, `-w-` を 2 に, `-wx` を 3 に, `r--` を 4 に, `r-x` を 5 に, `rw-` を 6 に, `rwx` を 7 に対応させることによって、0〜7 の数値で表す。
`[ugoa]`:	保護モードを変更したいユーザを表す文字 (所有者なら `u`, 同じグループのユーザなら `g`, 無関係の他ユーザなら `o`, 全員なら `a`) を並べたものを書く。省略すると全員 (`a`) と解釈される。
`[+-=]`:	権限を追加したい場合は `+` と書き、削除したい場合は `+`, 新たに設定したい場合は `=` と書く。
`[rwxst]`:	権限の種類を表す文字 (可読権なら `r`, 書込み権なら `w`, 実行/検索権なら `x`, ...) を並べたものを書く。

説明:
- 引数部 (*files/dirs*) で指定されたファイルやディレクトリについて、モード指定部の指示に従って保護モードの変更を行なう。
- 2 番目のコマンド形式においては、`[ugoa][+-=][rwxst]` の部分はコンマで区切って複数並べることもできる。

□演習 17.4 (保護モードの変更) 次の (a)〜(g) の操作を順に行なってみて下さい。

(a) ホームディレクトリの下に `tmp` というディレクトリがない場合は、作る。
(b) 演習 17.1 で作成したテキストファイルを `~/tmp/test` にコピーする。
(c) ファイル `~/tmp/test` の保護モードを `-w-------` にする。
(d) `~/tmp/test` の内容が見えない、すなわち `cat ~/tmp/test` がうまく実行できないことを確認する。

(e) ~/tmp/test の保護モードを rw-r--r-- にする。

(f) ディレクトリ ~/tmp の保護モードが rwx------の場合, rw-------の場合, -wx------の場合, -w-------の場合のそれぞれについて、

- ディレクトリ ~/tmp の内容を見ることができるかどうか、
- ~/tmp にカレントディレクトリを移動できる かどうか、
- ~/tmp/test の内容を見ることができるかどうか、

を調べて次の表を完成させよ。(実行が可能なら◯、禁止されていれば × を入れる。)

実行内容	~/tmp の保護モード			
	rwx------	rw-------	-wx------	-w-------
ls ~/tmp				
cd ~/tmp				
cat ~/tmp/test				

(g) ~/tmp/test は削除し、~/tmp の保護モードは rwxr-xr-x にしておく。

17.5 タッチタイピングの練習

最初に計算機を使う時はキーボード上に散らばったアルファベットや記号を探しながら使うことになると思いますが、このやり方では時間がかかるし、慣れたとしても大きなテキストファイルを作成する時には非常に疲れることになります。そこで、キーボード面を見ずにキーを打つ、**タッチタイピング** (または**ブラインドタッチ**) の練習をしておいて下さい。キーボードは計算機を使う際に最も長時間使う部分なので、キーボードが楽に速く打てればそれだけ計算機が快適に使えることになります。

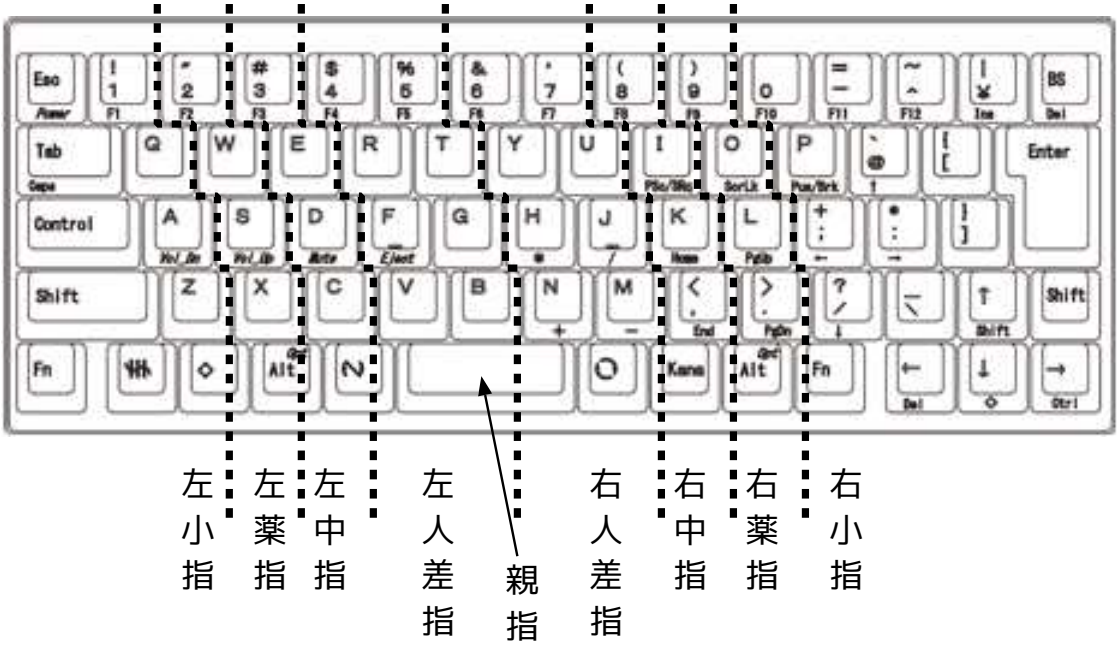


図 3: それぞれの指のタイプ範囲

タッチタイピング　両手の 10 本の指を全部使います。まず、左手と右手の人差し指をそれぞれ f キーと j キーの上に軽く置き、親指以外の残りの指は左右に水平にそれぞれのキーの上に軽く置きます。これを**ホームポジション**といいます。このポジションの下で、g, f, d, s, a のキーはそれぞれ左手人差し指, 人差し指, 中指, 薬指, 小指で打ちます。(人差し指は他の指より守備範囲が広い。) 同様に、h, j, k, l, ; のキーはそれぞれ右手人差し指, 人差し指, 中指, 薬指, 小指で打ちます。また、親指ではスペースキーを打ちます。その他のキーは、それぞれの指の守備範囲 (図 3) に従って指をそのまま上下に移動して打つだけです。

□**演習 17.5 (読み飛ばし可)** キーボードを見ずに

```
jfk kdd fd dj kkk
```

と打てるように練習して下さい。更に

```
de jure I die kid referee riff
rude jerk kid err Eur jeer JR
fried free Fiji fur rider ruff
KKK fire feud fifer kerf juju
```

と打てるように練習して下さい。

17.6 プリンタ操作

{UNIX/Linux の参考書 10.3.6 節 }

プリンタ操作のために実習室の UNIX/Linux で用意されているコマンドは次の 3 つです。

プリンタへの出力要求

形式:	**lpr** [options] *files*
options:	-P*prn* : 出力先プリンタ名を *prn* に指定します。
	-#*n* : 複写部数を *n* に指定します。
	その他 オンラインマニュアルを御覧下さい。(man lpr)

説明:	指定したファイルの内容をプリンタに印刷する。

印刷状況の表示

形式:	**lpq** [options] [*job-numbers*] [*user-names*]
options:	-l : 詳細情報を表示します。
	-P*prn* : 印刷状況を表示したいプリンタ名を *prn* に指定します。
	その他 オンラインマニュアルを御覧下さい。(man lpq)

印刷の中止

形式:	**lprm** [options] [*job-numbers*]
options:	-P*prn* : プリンタ *prn* に対する印刷要求を全て中止します。
	- : 既に印刷要求した全ての印刷を全て中止します。
	その他 オンラインマニュアルを御覧下さい。(man lprm)

実習室における特殊事情：

- 実習室では、日本語を含むテキストファイルの場合、単に `lpr` ファイル名 としただけでは日本語の部分が正しく印字されない様です。

  ⇨ テキストファイルを印刷したい場合は、

  `a2psj -p -f11` ファイル名 `| lpr`

  として下さい。

  > "`a2psj`" は指定されたテキストファイルを postscript 形式のファイルに変換するソフトウェアで、"`-p`"は portrait 出力を、"`-f11`"はフォントの大きさを指定しています。"`-f10`"や"`-f12`"といった指定も可能です。 また、"|" は次の 17.7 節で出てくるパイプです。⟹ UNIX/Linux の参考書 10.1.1 節, 10.2.2 節のパイプに関する記述を参考にして下さい。

- 実習室の PC 端末には各々デフォルトのプリンタが設定されていて、`lpr, lpq, lprm` コマンドにおけるプリンタ指定 (`-P` オプション) は省略可能です。

- 実習室で使えるプリンタは全て PostScript ページプリンタ (A4 用紙) で、それらの名前は次の通りです。

  PC 実習室 A： ca1pr01
  PC 実習室 B： ca2pr01

17.7 標準入出力とリダイレクション、パイプ

{UNIX/Linux の参考書 10.1.1 節, 10.2.2 節 }

標準入出力： UNIX/Linux では、入出力の先は固定させるべきではなく、各々の時点で標準的な入出力の先が決まっていればよい、と考える。具体的には、次の 3 つを標準的な入出力の先として想定する。

名称	意味
標準入力	標準的な入力の先, デフォルトではキーボード
標準出力	標準的な出力の先, デフォルトではディスプレイ
標準エラー出力	エラーメッセージ等を出す標準的出力の先, デフォルトではディスプレイ

入出力のリダイレクション： コマンドライン上で標準入出力の先をファイル等に繋ぎ替えることを**リダイレクション**と呼びます。具体的には、コマンドライン上で次の様な書き方をすると、リダイレクションが行なわれます。

- 標準入力先をファイルに切替：

  コマンド `<` *file*

- 標準出力先をファイルに切替：

  コマンド `>` *file*

- 標準出力先をファイルに切替 (ファイルの最後に追加出力)：

  コマンド `>>` *file*

- 標準出力と標準エラー出力の先を同一のファイルに切替：
  コマンド **&>** *file*
- 標準入力先と標準出力の先を別々のファイルに切替：
  コマンド **<** *infile* **>** *outfile*
- 複数のコマンドを順次実行しそれらの標準出力先をファイルに切替：
  (コマンド ; コマンド ; ... ; コマンド) **>** *file*

□**演習 17.6 (標準出力のリダイレクション；読み飛ばし可)** 上の記述を参考にして、次の (a)〜(f) の操作を順に行なってみて下さい。

(a) `date` コマンドの実行結果をリダイレクションにより `tmpf` というファイルに出力し、その結果 `tmpf` の内容がどうなったか確認する。
(b) `date` コマンドの実行結果をリダイレクションによりより `tmpf` というファイルに**追加**出力し、その結果 `tmpf` の内容がどうなったか確認する。
(c) `cat` コマンドとリダイレクションを用いて `tmpf` を `tmpf2` にコピーする。
(d) `date` コマンドを再実行しその実行結果を `tmpf` に上書き出力する。その結果 `tmpf` の内容がどうなったか確認する。
(e) `cat` コマンドとリダイレクションを用いて `tmpf` と `tmpf2` の内容の連結結果をファイル `tmpf3` に記録し、その結果 `tmpf3` の内容がどうなったか確認する。
(f) `tmpf`, `tmpf2`, `tmpf3` を削除する。

□**演習 17.7 (標準入出力のリダイレクション；読み飛ばし可)** 上の記述を参考にして、次の (a)〜(d) の操作を順に行なってみて下さい。

(a) 次の内容のファイル `expression` を作る。(大きなファイルでないので、プリント 3.2 節で行なったように `cat` コマンドを使うとよい。)

```
2+3*4
1+10+100+1000
```

(b) `bc <expression` とコマンド入力してみる。
(c) `bc <expression >result` とコマンド入力した後、`result` の内容が (b) の実行結果と一致することを確かめる。
(d) `expression`, `result` を削除する。

パイプ： コマンドライン上で、1 つのコマンドの標準出力を別のコマンドの標準入力に繋いでこれら複数のコマンドを並行に実行させることができます。この場合の、標準出力と別のコマンドの標準入力との連結を**パイプ**と呼びます。(正確に言うと、連結するために用意されるメモリバッファがパイプと呼ばれるものの実体です。) 具体的には、コマンドライン上で次の様な書き方をすると、複数のコマンドをパイプで繋げて並行に実行させることになります。

- **2 つのコマンドをパイプで繋げて並行に実行：**
  コマンド 1 **|** コマンド 2
  (コマンド 1 の標準出力をコマンド 2 の標準入力に繋ぐ。)
- **3 つ以上のコマンドをパイプで繋げて並行に実行：**
  コマンド 1 **|** コマンド 2 **|** ... **|** コマンド n

(コマンド 1 の標準出力をコマンド 2 の標準入力に、コマンド 2 の標準出力をコマンド 3 の標準入力に、...、コマンド $n-1$ の標準出力をコマンド n の標準入力に繋ぐ。)

□**演習 17.8 (パイプ；読み飛ばし可)** 上の記述を参考にして、次の (a)〜(e) の操作を順に行なってみて下さい。

(a) `ls -alF /usr/bin` というコマンド入力の実行結果をリダイレクションを用いて一旦 `tmpf` というファイルに出力し、その内容を `more` コマンドを用いてページ単位で表示させる。

(b) (a) の `tmpf` のような中間ファイルを作らずに、パイプを使って `ls -alF /usr/bin` の実行結果をページ単位で表示させる。

(c) `ls -alF /usr/bin` というコマンドとパイプ, `grep` コマンド, リダイレクションを用いて、emacs に関する情報だけを取り出し `tmpf` に出力 (上書き) する。その結果 `tmpf` がどうなったか確認する。

(d) `ls -alF ~` というコマンドとパイプ, `sort` コマンドを用いて、ホームディレクトリ内のファイルに関する情報をファイル容量の大きな順に画面に表示する。なお、`sort` コマンドのオプションは教科書と異なるため、`man` コマンドによって `sort` コマンドの利用方法を調べること。

(e) `tmpf` を削除する。

17.8 プロセスとジョブ

{UNIX/Linux の参考書 10.1.3 節, 10.3.4〜5 節 }

プロセス：

- UNIX/Linux 等のマルチタスク OS の下では、複数のプログラムが並行に動作している。
- 複数のプログラム実行を並行して進めるための仕掛けの中で扱われる、プログラム実行の単位 (i.e. プログラム実行の途中の状況を表したもの) を**プロセス**または**タスク**と呼びます。
- 複数のコマンドがパイプで繋がれた場合、それぞれのコマンドは別々のプロセスとして OS 内で管理され、コマンドに対応する複数のプロセスが独立に並行して動作することになります。
- OS 内でプロセスを識別するために、各々のプロセスには**プロセス ID**(PID) と呼ばれる番号が付けられています。
- 各々のプロセスの状態を表示したり、プロセスを一時停止したり、強制終了させたりする操作のことを、**プロセス制御**と言います。

プロセス制御のためのコマンドとしては、次の様なものがあります。

プロセス情報の表示

形式:	**ps** [options]	
options:	a	: 他ユーザのプロセスも含めて、全てのプロセスを表示する。
	u	: 利用者名,CPU やメモリの使用状況, 起動時間等の情報を表示する。
	l	: 詳細な情報を表示する。
	x	: 制御端末に関連のないプロセスの情報も表示する。
	その他	オンラインマニュアルを御覧下さい。(man ps)

説明:
- オプション部の指定に従って、プロセス情報を表示する。
- オプション部がない場合は、
  そのターミナルウィンドウ (制御端末) で起動したプロセスについて、プロセス ID(PID), 端末番号 (TTY), プロセスが消費した CPU 時間 (TIME), コマンド名 (CMD) が表示されるだけである。

プロセスに終了等のシグナルを送る

形式:	**kill** [options] [*pids*]	
options:	-TERM	: TERM シグナルを送る。
	-KILL	: KILL シグナルを送る。
	-l	: シグナルの一覧を表示する。
	その他	オンラインマニュアルを御覧下さい。(man kill)

説明:
- 引数で指定されたプロセスにオプション部で指定されたシグナルを送る。
- オプション部が省略された場合は TERM シグナルが送られる。
- TERM シグナルが送られたプロセスは、多くの場合終了するが終了しないこともある。
- KILL シグナルが送られたプロセスは、必ず終了する。
- 引数部には、プロセス ID を空白で区切って並べることによって、複数のプロセス ID を指定することもできる。

ターミナルウィンドウを閉じる (終了)

形式: **exit**

説明: ターミナルウィンドウ内でコマンド処理を担当していたシェル (プロセスの 1 種) を終了する。

□**演習 17.9 (プロセス制御；読み飛ばし可)** 上の記述を参考にして、次の (a)〜(e) の操作を順に行なってみて下さい。

(a) ps コマンドによって (バックグラウンドのものも含めて) 全てのプロセスの状態を表示する。
(b) xeyes& とコマンド入力する。
(c) ps コマンドによって (バックグラウンドのものも含めて) 全てのプロセスの状態を表示する。
(d) (b) で起動した xeyes のプロセスを kill コマンドによって強制終了させる。
(e) ps コマンドによって (バックグラウンドのものも含めて) 全てのプロセスの状態を表示する。

ジョブ：

- OS 側から見たプログラム実行の単位をプロセスと呼ぶのに対して、ユーザ側から見たひとまとまりのプログラム実行の単位を**ジョブ**と呼びます。
- 複数のコマンドがパイプで繋がれた場合、これら一連のコマンド実行が 1 つのジョブになります。
- OS 内でジョブを識別するために、各々のジョブには番号 (ジョブ番号という) が付けられています。
- 各々のターミナルウィンドウでは、コマンド/ジョブが順次実行され、通常は実行中のジョブが終了するまで次のコマンド/ジョブの指示を出すことができません。こういう種類の (i.e. 終了するまでターミナルウィンドウを支配し次のコマンドの受付けをできなくする) ジョブのことを**フォアグラウンドジョブ**と呼びます。
  各々のターミナルウィンドウにおいて、各時点でフォアグラウンドジョブは高々1個しかありません。
- 各々のターミナルウィンドウにおいて、フォアグラウンドジョブとは別のジョブを裏で並行して実行させることができます。この様に裏で並行して実行させるジョブのことを**バックグラウンドジョブ**と呼びます。
  各々のターミナルウィンドウにおいて、バックグラウンドジョブは同時に多数存在する可能性があります。
- 各々のターミナルウィンドウにおいて、0〜1 個のフォアグラウンドジョブと 0 個以上のバックグラウンドジョブが並行して走ることになります。

ジョブ制御：

- 各々のターミナルウィンドウにおいて、ジョブをバックグラウンドジョブとして起動させたり、フォアグラウンドジョブを強制終了・一時停止させたり、一時停止中のジョブをバックグラウンドジョブとして実行再開させたり、バックグラウンドジョブをフォアグラウンドに切り替えたり、といった操作のことを、**ジョブ制御**と言います。
- ジョブ制御の方法をまとめると図 4 の様になります。
- バックグラウンドジョブの起動、フォアグラウンドジョブの終了・一時停止は、コマンドライン上で次の様に行ないます。
  - ◇ バックグラウンドジョブの起動：
    コマンド **&**
  - ◇ フォアグラウンドジョブの終了 (**INT** シグナルを送る)：
    Ctrl-**c**　(すなわち、Ctrl キーを押しながら c キーを打つ)
  - ◇ フォアグラウンドジョブの一時停止 (**TSTP** シグナルを送る)：
    Ctrl-**z**　(すなわち、Ctrl キーを押しながら z キーを打つ)
- 処理時間が長いプログラムを実行する場合は、例えば
  `$` 実行プログラム名 `> result &`
  とすれば、ログアウト後も実行を続けて、実行結果は `result` というファイルに入れてもらえます。

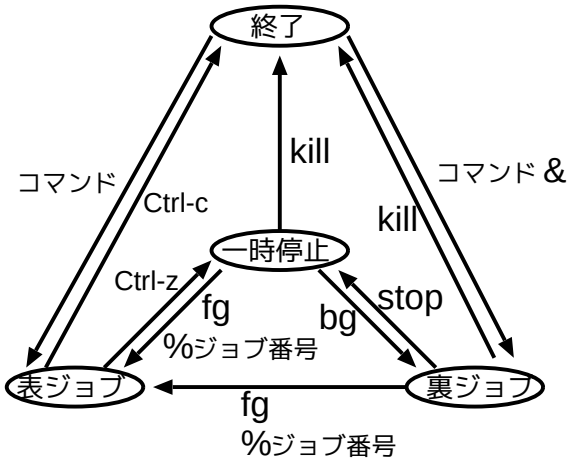


図 4: ジョブの状態制御

ジョブ制御のためのコマンドとしては、次の様なものがあります。

ジョブの稼働状況を一覧表示

形式:	**jobs** [options] [*jobs*]
options:	-l : プロセス ID も表示する。 その他 オンラインマニュアルを御覧下さい。(man jobs)
jobs:	引数部においては、ジョブは例えば %ジョブ番号 という風に指定する。

説明:
- 引数で指定されたジョブの稼働状況を一覧表示する。
- 引数 (ジョブ番号) が省略された場合は、全ての (バックグラウンド) ジョブの稼働状況を一覧表示する。
- 引数部には、空白で区切って並べることによって、複数のジョブを指定することもできる。

バックグラウンドジョブをフォアグラウンドに切り替える

形式:	**fg** [*job*]
job:	引数部においては、ジョブは例えば %ジョブ番号 という風に指定する。

説明:
- 引数で指定されたジョブをフォアグラウンドに切り替える。
- 引数が省略された場合は、最後に操作したジョブ (i.e.jobs による一覧表示で +という印が付いたジョブ) をフォアグラウンドに切り替える。

一時停止中のジョブをバックグラウンドグラウンドで再開

形式:	**bg** [*job*]
job:	引数部においては、ジョブは例えば %ジョブ番号 という風に指定する。

説明:
- 引数で指定されたジョブをバックグラウンドグラウンドで再開する。
- 引数が省略された場合は、最後に操作したジョブ (i.e.jobs による一覧表示で +という印が付いたジョブ) をバックグラウンドグラウンドで再開する。

ジョブを構成するプロセスに終了等のシグナルを送る

形式:	**kill** [options] [*jobs*]
options:	-TERM : TERM シグナルを送る。 -KILL : KILL シグナルを送る。 -l : シグナルの一覧を表示する。 その他 オンラインマニュアルを御覧下さい。(man kill)
job:	引数部においては、ジョブは例えば %ジョブ番号 や %% という風に指定する。ここで、%% は最後に操作したジョブ (i.e.jobs による一覧表示で +という印が付いたジョブ) を表す。

説明:
- 引数で指定されたジョブに対応するプロセスにオプション部で指定されたシグナルを送る。
- 他の点に関しては、プロセスに対して kill コマンドを適用する場合と同じである。

□**演習 17.10 (ジョブ制御；読み飛ばし可)** 上の記述を参考に、次の (a)〜(f) の操作を順に行なってみて下さい。

(a) xclock -digital -update 1 とコマンド入力する。
(b) (a) のジョブを一時停止させる。
(c) jobs コマンドで (a) のジョブ番号を確認する。
(d) (a) のジョブをバックグラウンドジョブで再実行する。
(e) (a) のジョブをフォアグラウンドに切替える。
(f) (a) のジョブを強制終了させる。

(Coffee Break)

GUIって使いやすいの？

ある人に言わせると、

> GUI は初めてコンピュータに触れる人にとってはわかりやすいかもしれません。しかし、実際にコンピュータを使って仕事をする人にとってはどうでしょうか?コンピュータを使って絵を書くような場合はともかく、文書を作ったり、プログラミングをしたりする人にとっては、わざわざマウスに手を伸ばさないと何もできないようなインターフェイスは邪魔でしかありません。キーボードに慣れれば慣れるほど、極力キーボードのホームポジションから手を離したくないのです。実際、私が普段使用している環境は、絵を書くとき以外は、まったくマウスにさわる必要が無いようにカスタマイズしてあります。

だそうです。

18 Emacs エディタ —UNIX 演習 (その 3)—

{UNIX/Linux の参考書第 4 章 }

エディタはプログラムなどのテキストファイルを新規作成したり編集したりするためのソフトウェアであり、UNIX の下では、標準装備されシステム管理者に必須の vi とフリーソフトウェアである GNU Emacs 系統のものがよく使われています。Emacs の系統は R.Stallman が作成した元々の Emacs に始まり、Emacs を日本語が扱えるように拡張した NEmacs、日本語だけでなく他の言語も同時に扱うことも考慮に入れて拡張した Mule、Mule を統合した新版 Emacs(参考書)、グラフィックスにも対応するように新版 Emacs を更に拡張した XEmacs へとバージョンアップが繰り返されてきていますが、基本操作についてはあまり変わっていません。

18.1 Emacs エディタの起動、終了

{UNIX/Linux の参考書 4.1〜2 節節 }

情報基盤センターの RedHat Enterprise Linux5 には Emacs がインストールされています。**この演習では標準的なエディタとして Emacs を使います。**

UNIX/Linux の参考書を読む場合、メタキー Meta は Esc に置き換えて読んで下さい。

Emacs エディタの起動: ターミナルウインドウ上で次の様にコマンド入力すれば Emacs が起動し、編集テキストの初期設定も行われます。

$ emacs 各種オプション (省略可) ファイル名 (省略可) & Enter

編集ファイル名を省略した場合は、編集テキストは空に初期設定されます。

Emacs 系のエディタは指定するファイルの拡張子からファイルの種別を認識して、その種別に合った動作をします。
⟹ 例えば **C プログラムを作成する場合は、".c" という拡張子のファイル名を指定して Emacs を起動する**と、ファイルの編集をスムーズに行うことが出来ます。

起動の際に ファイル名 (省略可) を指定しないと、最初に図 5 の様に案内文が表示され、ウィンドウ内部を左クリックすると文頭に 3 行のメッセージが出ます。これらのメッセージは BackSpace キーで消して下さい。

Emacs エディタの終了: Emacs エディタの終了は、エディタのメニューバーで File⟶ExitEmacs と選択することによってもできますが、Emacs ウィンドウが選択されている状態で次のキーボード操作を行なうことによっても可能です。

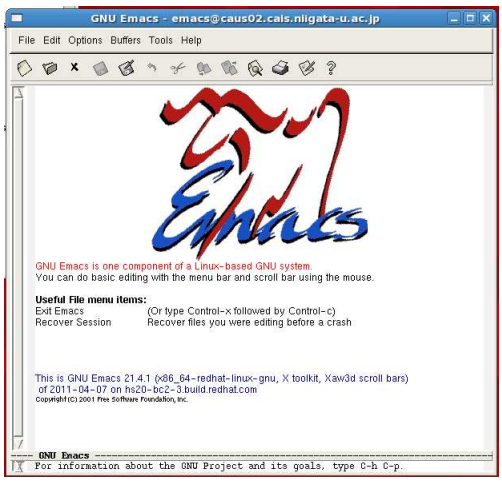


図 5: Emacs の初期画面 (ファイル名の指定なしで起動した場合)

キー操作	効果
Ctrl-x Ctrl-c ···	Emacs の終了。編集中の文書の最新版が未保存の場合は、文書を保存するかどうかの質問がミニバッファ上に現れ、保存する場合はミニバッファ上でのやりとりを通じてファイルとして保存されることになる。

□**演習 18.1** 次の (a)〜(d) の操作を順に行なってみて下さい。

(a) 上の記述に従って Emacs を起動する。(ここではファイル名を指定しなくても結構です。)

(b) Emacs 画面の構成について、次のことを確認して下さい。

- 上部に**メニューバー** (上から 2 段目)、**ツールバー** (上から 3 段目)、左側に**スクロールバー**がある。
- 中央に編集テキストを表示する部分がある。起動直後に図 5 の様になっている場合は、ウィンドウ内部を左クリックすると文頭に 3 行のメッセージが出ます。
- 下から 2 段目の行は**モード行**と呼ばれ、 文字コードを表す記号 (u は unicode, E は EUC,...), 編集状態を表す記号 (--は未編集,**は編集中, %%は編集不可), ファイル名 (指定しなかった場合は*scratch*) が左から順に並んでいる。右側の --L 数字列-- という部分には、編集テキスト中の上から何行目にカーソルがあるかを表示している。
- 最下段の行は**ミニバッファ**と呼ばれる行で、 Emacs のメッセージが表示される。場合によっては、この場所に Emacs への応答を入力することもあります。
- 上部のメニューバーにどんな操作が用意されているか。(メニューバーの「File」や「Edit」,... を左クリックすると、選択可能な操作の一覧表が表示されます。またツールバー上のボタンの所にマウスカーソルを移すと、そのボタンの簡単な説明が表示されるはずです。)

(c) 編集テキストを表示するウィンドウ内で、文字の入力や削除、カーソル移動等を行なってみて下さい。

(d) Emacs を終了して下さい。

18.2 Emacs エディタの下でのテキストファイルの作成／編集

{UNIX/Linux の参考書 4.2.2〜5 節, 4.3.1〜2 節 }

文字の入力と削除： 次の様なキーボード操作が可能です。

キー操作		効果
文字キー	···	カーソルのある場所に打ち込んだ**文字が挿入**され、カーソルは右に 1 文字進む。(場合によっては、字下げのために空白の挿入/削除を伴う。)
BackSpace	···	**カーソルの直前の文字が削除**される。カーソルが行の左端にあった場合は前の行の改行コードが削除され、カーソルのあった行が前の行の最後に繋がる。
Ctrl-d	···	**カーソル上の文字が削除**される。(delete) カーソルが行の右端 (改行コード上) にあった場合は改行コードが削除され、行の後ろに次の行が繋がる。
Enter	···	カーソルのある場所に**改行コードが挿入**され、カーソルは次の行の先頭に移る。
		注意：C ソースプログラムを作成する場合は、各行の終わりには Enter ではなく次の Ctrl-j を使うべきである。
Ctrl-j	···	**改行と次の行の字下げ**が行なわれ、カーソルはその字下げの先まで移る。

基本的なカーソル移動操作： カーソルを左クリックした場所に Emacs エディタのカーソル ▉ が移動するようになっています。また、次の様なキーボード操作が可能です。

キー操作		効果
→	···	カーソルが右に 1 文字進む。
Ctrl-f	···	同上。(**f**orward)
←	···	カーソルが左に 1 文字進む。
Ctrl-b	···	同上。(**b**ackward)
↑	···	カーソルが上に 1 行進む。
Ctrl-p	···	同上。(**p**revious)
↓	···	カーソルが下に 1 行進む。
Ctrl-n	···	同上。(**n**ext)

但し、画面上は空白でも、何の文字も入っていない所にはカーソルは移動できません。[空白文字の入っている所と何の文字も入っていない所は、見た目には同じでも内部では違っています。] そのために、例えば、カーソルが行の右端にある状態で→キーを打ち込んだ場合は、次の行があればカーソルは次の行の先頭に移り、次行がなければカーソルは動かない。

編集中のテキストをファイルに保存： ファイルへの保存は、メニューバーで機能選択を行なったり、ツールバーのボタンを押したりすることによってもできますが、次のキーボード操作によっても可能です。

キー操作		効果
Ctrl-x Ctrl-s	···	編集ファイル名を既に指定している場合 は、 その名前のファイルとして**編集中の文書が保存** (save) され、 その結果が最下段のミニバッファ上に報告される。 また、ファイル名を未指定の場合 は、最下段のミニバッファ 部にファイル名を要求するメッセージが表示され、それに 続くミニバッファ部にファイル名を入れると編集中の文書が 保存され、その結果が最下段のミニバッファ上に報告される。
Ctrl-x Ctrl-w	···	**ファイル名を新たに指定して、そこに保存**する時に用いる。 (write)　左のコマンドを入力すると、最下段のミニバッファ 部にファイル名を要求するメッセージが表示され、それに 続くミニバッファ部にファイル名を入れると編集中の文書が 保存され、その結果が最下段のミニバッファ上に報告される。

注意 (保存操作をしないと...)：Emacs ウィンドウ右上の ⊠ ボタン を押したただけでは、ファイルはちゃんとした形で保存されない。(両側が “#” で囲まれた名前のファイルが残ることもあるが、これは Emacs 内部の作業用ファイルで、Emacs で表示しても文字化け等がある。)

□**演習 18.2 (新規ファイル作成と保存)** 次の (a)〜(d) の操作を順に行なってみて下さい。

(a) ファイル名を指定して、Emacs を起動して下さい。(C ソースプログラムを構成することになるので、それに見合った拡張子を付けて下さい。)

(b) 例題 5.3 で示した C 言語のプログラム (p.36) を Emacs ウィンドウ上に構成してみよ。[但し、示したプログラムの左側の 数字:␣ の部分はプログラムの一部ではありません。 注意して下さい。 また、ここではプログラムの意味を理解しなくても結構です。C プログラムの場合は、**行の最後に Enter ではなく Ctrl-j を打ち込む**と、字下げは Emacs が自動的に行ってくれます。]

(c) 作成した文書を保存する。

(d) (b) で作成／更新されたファイルの中身を (`more` コマンドで) 覗いて、ちゃんとファイルが構成されていることを確認する。

□**演習 18.3 (拡張子 `.c` を指定しないと...)** 拡張子なしのファイル名を指定して Emacs を起動した上で、例題 5.3 で示した C 言語のプログラム (p.36) を Emacs ウィンドウ上に構成してみて下さい。ここでは、Ctrl-j というキー操作はどう働くでしょうか？

ファイルの読み込み/挿入：　ファイルの読み込み/挿入は、メニューバーで機能選択を行なったり、ツールバーのボタンを押したりすることによってもできますが、次のキーボード操作によっても可能です。

キー操作		効果
Ctrl-x Ctrl-f	···	**指定ファイルの内容を新たな「編集中のテキスト」として設定**する。具体的な手続きとしては、左のコマンドを入れると、最下段のミニバッファ部にファイル名を要求するメッセージが表示され、それに続くミニバッファ部にファイル名を入れると、指定ファイルの内容が新たに「編集中のテキスト」として設定されることになる。
Ctrl-x i	···	カーソル位置に**指定ファイルの内容を挿入する。**具体的な手続きとしては、左のコマンドを入れると、最下段のミニバッファ部にファイル名を要求するメッセージが表示され、それに続くミニバッファ部にファイル名を入れると、指定ファイルの内容がカーソル位置に挿入されることになる。

マウス操作によるコピー・アンド・ペースト： 第 16.10 節で説明したコピー・アンド・ペーストの操作は、編集中の文書を表示しているウィンドウ内でも有効です。これによって、次の様な操作を楽に行なうことができます。

- 編集中の文書内の文字列 (複数行に渡っていても良い) を Emacs ウィンドウの別の場所にもコピー (挿入) する。
- ターミナルウィンドウ上に表示された文字列を Emacs ウィンドウ内にコピー (挿入) する。

カーソル移動操作 (追加)： 次の様なキーボード操作も可能です。

キー操作		効果
Ctrl-a	···	カーソルが行の先頭に移動。
Ctrl-e	···	カーソルが行末に移動。
Esc <	···	カーソルが文書の先頭に移動。
Esc >	···	カーソルが文書の末尾に移動。
Esc x goto-line	···	指定した行番号の行に移動。(Esc x と打ち込むと、それがミニバッファ内に表示されるので、続けて goto-line Enter と打ち込む。すると、今度はミニバッファ内に Goto line: と表示されるので、これに続けて移動先の行番号を入れ Enter を押す。)

画面スクロール： 次の様なキーボード操作も可能です。

キー操作		効果
Ctrl-v	···	1 画面分スクロールアップ (次の 1 画面分を表示)
Esc v	···	1 画面分スクロールダウン (前の 1 画面分を表示)

行末までの文字列を削除： 次の様なキーボード操作も可能です。

キー操作		効果
Ctrl-k	···	カーソルの場所から行末までの文字列を削除

入力中のコマンドのキャンセル： 次の様なキーボード操作が可能です。編集作業中に障害が発生した場合にも試してみて下さい。

キー操作		効果
Ctrl-g	···	例えば、Esc x goto-line とした後で、「指定した行番号の行へのカーソル移動」を中止したい場合に、ミニバッファ内で Ctrl-g と打ち込むと、この実行途中のコマンドがキャンセルされる。

操作の取消： 次の様なキーボード操作も可能です。

キー操作		効果
Ctrl-u	···	過去の操作までさかのぼって 1 操作ずつ取り消す。
Esc x undo	···	直前の 1 つの操作を取り消す。(2 度続けて実行すると、2 回目には直前の取り消し操作自体が取り消されることになる。)

□**演習 18.4 (Emacs/ファイル操作)** 次の (a)〜(f) の操作を順に行なってみて下さい。

(a) 編集ファイル名を指定せずに Emacs を起動する。[&を忘れずに。]
(b) ファイル読み込みのコマンドを使ってみる。例えば、演習 18.2 で作った文書ファイルを読み込んでみる。(UNIX/Linux の参考書 4.3.2 節,p.50)
(c) カーソル移動を行ってみる。 (UNIX/Linux の参考書 4.3.1 節)
(d) 画面スクロールを行ってみる。 (UNIX/Linux の参考書 4.3.1 節)
(e) 行末までの文字列削除を行ってみる。 (UNIX/Linux の参考書 4.3.1 節)
(f) 操作の取消を行ってみる。 (UNIX/Linux の参考書 4.3.8 節)
(g) ファイルを保存せずに Emacs を終了する。

文書表示ウィンドウの分割： Emacs では、編集中の文書を表示するウィンドウを上下や左右に分割して、各々のウィンドウに文書の別々の箇所を表示させることができます。また、ウィンドウ毎に別の文書を表示させることもできます。次の様なキーボード操作が可能です。

キー操作		効果
Ctrl-x 2	···	ウィンドウが上下に分割される。(同じ文書を表示)
Ctrl-x 3	···	ウィンドウが左右に分割される。(同じ文書を表示)
Ctrl-x 1	···	選択中の (i.e. カーソルのある) ウィンドウ以外が閉じられて、ウィンドウが 1 個の状態に戻る。
Ctrl-x o	···	複数のウィンドウ間をカーソルが循環的に移動する。
Ctrl-x b	···	バッファ(i.e. 編集中の文書を入れている領域) の切り替え。
Ctrl-x Ctrl-b	···	Emacs が現在使っているバッファの一覧表示。
Ctrl-x k	···	編集中のバッファの削除。

□**演習 18.5 (複数のウィンドウを使った編集；読み飛ばし可)** 次の (a)〜(f) の操作を順に行ってみて下さい。

(a) 編集ファイル名を指定せずに Emacs を起動する。[&を忘れずに。]
(b) ウィンドウを上下に分割する。(UNIX/Linux の参考書 4.3.3 節, p.54)

(c) 1 つのウィンドウに文書ファイルを読み込む。 (例えば、演習 18.2 で作った文書ファイル。)
(d) もう 1 つのウィンドウには、コピーアンドペースト操作を使って (c) と同じ文書を入れる。(UNIX/Linux の参考書 4.3.7 節, p.62)
(e) ウィンドウ間移動を行ってみる。 (UNIX/Linux の参考書 4.3.3 節, p.54)
(f) ウィンドウ削除を行ってみる。 (UNIX/Linux の参考書 4.3.3 節, p.55)
(g) ファイルを保存せずに Emacs を終了する。

文字列の検索と置換： 次の様なキーボード操作も可能です。

キー操作		効果
Ctrl-s	···	現カーソル位置から始めて**順方向に (文末に向けて) 別途指定の文字列の場所を検索**する。具体的には、左のキー操作でミニバッファ内に検索文字列の入力を促すメッセージが出る。ミニバッファ内に検索文字列を入れることになるが、1 文字入れる度にそれまでに入れた文字列を用いた検索が為され画面表示は順方向に進む。再度 Ctrl-s と打てば、次の検索文字列まで検索が更に進む。
Ctrl-r	···	現カーソル位置から始めて**逆方向に (文頭に向けて) 別途指定の文字列の場所を検索**する。
Esc %	···	**確認付きの文字列置換**を現カーソル位置から順方向に行なう。具体的には、左のキー操作でミニバッァ内に検索文字列の入力を促すメッセージが出、これに答えると今度は置換後の文字列入力を促すメッセージが出る。これに答えると、現カーソル位置から順方向に (文末に向けて) 検索が始まり、検索文字列が見つかる度に本当に置換を行なって良いかどうかの質問がミニバッァ内に表示される。これに答えると、次の検索文字列へと進む。
Esc x replace-string	···	**確認なしの文字列置換**を現カーソル位置から順方向に行なう。

□演習 18.6 (文字列検索, 文字列置換；読み飛ばし可) 次の (a)〜(e) の操作を順に行なってみて下さい。
(a) 編集ファイル名を指定せずに Emacs を起動する。[&を忘れずに。]
(b) 文書ファイルを読み込む。 (例えば、演習 18.2 で作った文書ファイル。)
(c) 文字列検索を行ってみる。 (UNIX/Linux の参考書 4.3.5 節)
(d) 文字列置換を行ってみる。 (UNIX/Linux の参考書 4.3.5 節)
(e) ファイルを保存せずに Emacs を終了する。

18.3 文章作成以外の機能

{UNIX/Linux の参考書 4.4 節 }

ディレクトリの編集 (Dired モード)： Emacs では、テキストファイルだけでなくディレクトリ (UNIX/Linux ではファイルの一種として扱われる) の中身も編集 (e.g. 所属するファイルの削除, 名前変更) することができる。次の様なキーボード操作が可能です。(動

作未確認)

キー操作		効果
Esc x dired	···	Dired モードに入る
		(以下、Dired モード内で)
d	···	カーソルの示すファイルに削除マークを付ける
u	···	カーソルの示すファイルから削除マークを外す
x	···	削除マークの付いたファイルを実際に削除
f	···	カーソルがテキストファイルの場合はそのファイルを読み込み, ディレクトリ場合には表示をそのディレクトリに変更する
R	···	カーソルの示すファイル名/ディレクトリ名の変更
v	···	ファイルの表示
C	···	コピー
q	···	Dired モードを終了

□演習 18.7 (Dired モード；読み飛ばし可) Emacs を Dired モードで使ってみる。(UNIX/Linux の教科書 4.4.2 節)

オンラインドキュメントの表示 (Info モード)： 次の様なキーボード操作が可能です。(動作未確認)

キー操作		効果
Esc x info	···	Info モード内のメニュー表示の画面に入る
		(以下、Info モードのメニュー表示画面で)
Tab	···	次のタグに移動
Esc Tab	···	前のタグに移動
Space	···	参考文書をみる
q	···	Info モードを終了
		(以下、Info モード内の文書表示画面で)
d	···	最初のメニューに戻る
n	···	次のページへ
p	···	前のページへ
u	···	前の大項目へ

18.4 日本語入力

{UNIX/Linux の参考書 4.6 節 }

- UNIX ワークステーションの時代には、普通 Wnn(うんぬ), または Canna(かんな) といったクライアント/サーバ形式のかな漢字変換エンジンが動作していました。
- マルチユーザのクライアント/サーバ形式を用いた場合、セキュリティの問題がクライアント側にもサーバ側にも発生し、またサーバ開発にもより多くの手間がかかるので、最近では、シングルユーザで動作するかな漢字変換エンジンとして Anthy(参考書 4.6 節) や PRIME といったソフトウェアが開発され、多言語入力環境を提供する SCIM(スキム) というソフトと組み合わせて使われている様です。

- センター実習室の Linux システムでは、Anthy と SCIM がインストールされています。
- センター実習室の Linux システムでは **日本語入力モード ⟷ アルファベット入力モード の切替えは、次のキーで行います。**

  Emacs 内なら ⟶ Ctrl-\
  それ以外なら ⟶ Ctrl-Space

> 補足 :Emacs 外で Ctrl-Space キーを押すと、画面右下に次の様な表示が現れる。
>
> 
>
>
> この中のボタンを使って入力方法の切り替えを行うこともできる。
>
> ---
>
> 日本語を使う場合、ASTEC-X ウィンドウ上部バー右端 (右から 3 つ目) のキーボードアイコンを
> English/keyboard (英語/キーボード) ⟶ 日本語
> と変更しておくと、Emacs 上で
> 文節を伸ばすキー Shift-→ や
> 文節を縮めるキー Shift-←
> が使える様になる。

- 文字コードを尋ねられたら、暗黙に仮定されている utf-16-le ではなく utf-8 を指定する。

> 補足 :Emacs ウィンドウ左下の文字コード表示の箇所は、テキスト保持に採用している文字コードが utf-16-le の場合は大文字の `U` となり、また文字コードが utf-8 の場合は小文字の `u` となる。 文字コードを指定するには
> Options ⟶ Mule (Multilingual Environment)
> ⟶ Set Coding Systems ⟶ For Saving This Buffer
> とメニュー選択すると、ミニバッファ部に「Coding system for saving file (default nil):」と出てカーソルがその右に行くので、そこに utf-8 と指定して Enter。

□**演習 18.8** 次の 3 行からなる日本語テキストファイルを作成せよ。 (UNIX/Linux の参考書 4.6.1 節)

```
/* プログラミング基礎演習 (○曜○限) */
/* 新潟大学○学部○○○学科          */
/* 各自の学籍番号 各自の氏名        */
```

レポート課題 1 演習 18.2 で作成した文書ファイル (C プログラム) の先頭に演習 18.8 で作成した文書ファイルの内容を挿入せよ。そして、その結果を `a2psj` コマンド (17.6 節) を用いてプリンタに出力して、レポートとして提出せよ。

- 提出期限 : 10 月 28 日 (金) 辺り?
  $\Longrightarrow$ 実習担当者の指示に従う。
  ...多少遅れても必ず出して下さい。
- 提出先 : 教養校舎教務係のレポート提出ボックス、
  または直接担当者に。

□**演習 18.9** この講義ノート 17.6 節の `lpq` コマンドを用いて幾つかのプリンタの印刷状況を調べよ。

□**演習 18.10** 演習 18.9 において自分が出した無用な印刷要求が見つかった場合、17.6 節の `lprm` コマンドを用いてそれらの印刷要求をすべて取り消して下さい。

(Coffee Break)

キーボードへのこだわり

ある人に言わせると、

> プログラマーが最も使うことになるインターフェースはキーボードです。そのため、キーボードに異常なまでにこだわる人がいます。(私もその一人ですが。) Control キーは A の隣!とか、スペースはやっぱりでかくないと、とか、キータッチは軽く (重く) ないと!等々... いくところまでいってしまうと例えばこうなります。
>
> http://www.meisiya.net/maltron/

だそうです。

Cプログラミング演習

19 Cプログラムの作成と実行

{UNIX/Linuxの参考書11.1〜2節, 11.4節}

- Cプログラムの入ったソースファイルは .c という拡張子で終らせる。
- Cプログラムのコンパイル (機械語への翻訳) には `cc` または `gcc` コマンドを使う。[このコマンドにより、デフォルトでは `a.out` という名前の実行可能ファイルが生成される。]

19.1 プログラム作成・修正・保存・実行の典型的な手順

{UNIX/Linuxの参考書11.1〜2節, 11.4節}

普通、次の様な手順を経てプログラムを作成・実行する。

(1) ターミナル (GNOME端末) ウィンドウが無い場合は、1個開く。

(2) Cプログラミング用のディレクトリが無い場合は新規に作る。 例えば、GNOME端末ウィンドウ上で

`mkdir ~/c-lab` Enter

とする。

{UNIX/Linuxの参考書10.3.2節, 第3章}

(3) GNOME端末ウィンドウ上で、Cプログラミング用のディレクトリに移る。例えば、

`cd ~/c-lab` Enter

とする。

{UNIX/Linuxの参考書10.3.2節}

(4) GNOME端末ウィンドウ上で `emacs` ファイル名 `&` とコマンド入力してEmacsを起動する。

{UNIX/Linuxの参考書11.2節 (p.224), 10.3.5節}

> このコマンド入力によってEmacsウィンドウが生成される。
> ここで、新規にプログラムを作る場合はそのプログラムを入れるファイルの名前をファイル名 の場所に指定する。また、過去に作ったプログラムを修正する場合はそのプログラムの入ったファイルの名前をファイル名の場所に指定する。但し、Cプログラムのファイル名は .c で終わる様にする。(UNIX/Linuxの参考書3.1節参照。)

(5) EmacsウィンドウとGNOME端末ウィンドウの大きさ／位置を調整して画面を見易くする。

{UNIX/Linuxの参考書2.7節}

できればウィンドウが重ならない様に (例えば図 6 または UNIX/Linux の参考書 p.224 の図 11.3 の様に) する。 Emacs ウィンドウ 1 個と GNOME 端末ウィンドウ 1 個以外に余計なウィンドウがあれば、ウィンドウ右上の強制終了ボタンによって消したり、最小化ボタンによってアイコン化して画面下のパネル内に置いたりすると良い。

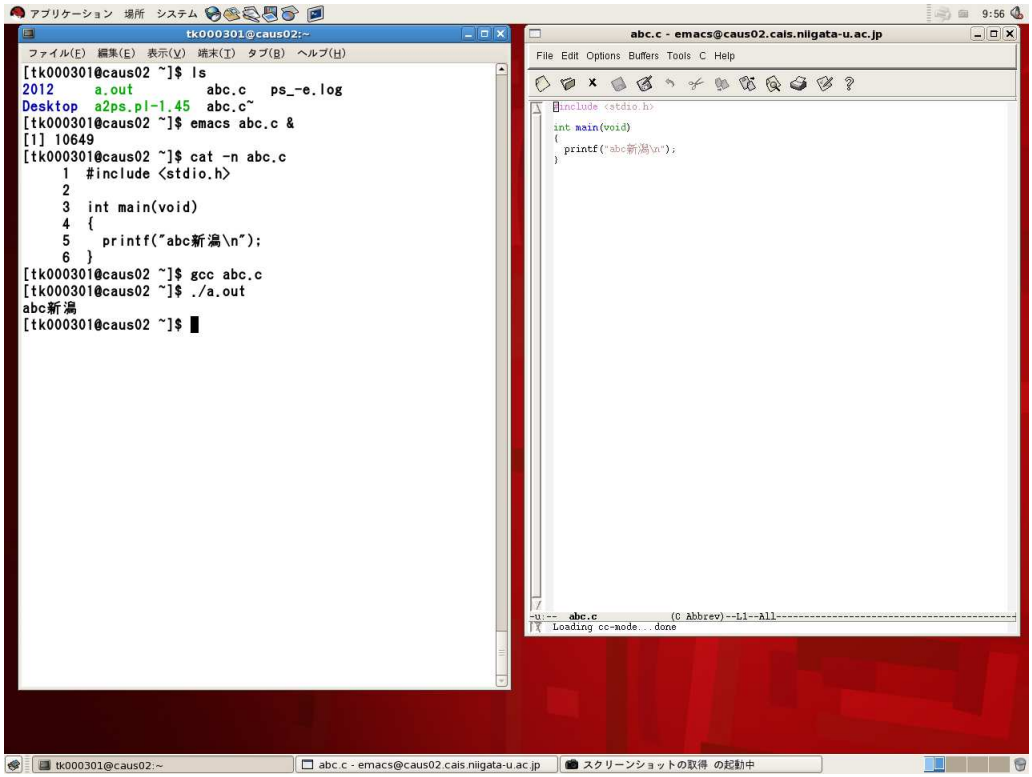


図 6: ASTEC-X ウィンドウ上で C プログラミングを行う時の画面

(6) Emacs エディタの下でプログラムを作成 (または修正)・保存する。
{UNIX/Linux の参考書 11.2 節 (p.224〜226), 11.4.1〜2 節 (p.229〜231), 第 4 章 }

Emacs エディタはまだ終了しない。センター実習室 Linux に入っている日本語入力システムは Anthy-SCIM である。

(7) GNOME 端末ウィンドウを選択してアクティブ状態にする。
{UNIX/Linux の参考書 2.3.2 節, 2.7 節 }

(8) cc コマンドによってプログラムをコンパイル (i.e. 機械語に翻訳) する。
{UNIX/Linux の参考書 11.2 節 (p.225)}

cc ソースファイル名
とコマンド入力する。 cc コマンド／C コンパイラについてはこのプリントの付録 A を参照。

「bash: cc: command not found」というメッセージが返って来て cc コマンドを実行できない場合は、
gcc ソースファイル名
とコマンド入力してみて下さい。

(9) コンパイラからのエラーメッセージがある間は次の①〜③を繰り返し行う。
{UNIX/Linux の参考書 11.2 節 (p.225〜226)}

① エラーメッセージを手掛かりにソースプログラム内の文法的な誤りを見つけ、Emacs ウィンドウ上で修正する。

② Ctrl-x Ctrl-s とキーを押すことによってソースファイルを更新する。

③ 再度プログラムをコンパイルする。

(10) コンパイルが無事 (i.e. エラー無しで) 終了したら、a.out と入力してプログラムを実行する。 {UNIX/Linux の参考書 11.2 節 (p.226)}

「bash: a.out: command not found」というメッセージが返って来て a.out を実行できない場合は、
./a.out Enter
とコマンド入力する。

プログラムが終了しない場合は Ctrl-c キーを押して強制終了させる。 (UNIX/Linux の参考書 10.3.5 節を参照。)

例えば、次の通り。(下線部がキーボードからの入力。)

```
$ cat ex01.c Enter
#include <stdio.h>
main()
{
   int x,y,sum,ceiling;

   scanf("%d %d", &x, &y);
   sum=x+y;
   ceiling=(x+y-1)/y;       /* x/y の小数点以下切り上げ */
   printf("%d+%d=%d\n", x, y, sum);
   printf("ceiling(%d/%d)=%d\n", x, y, ceiling);
}
$ cc ex01.c Enter
$ ./a.out Enter
8 3 Enter
```

```
8+3=11
ceiling(8/3)=3
$
```

(11) 実行結果に満足できない場合は、次の①～③を順に行った後にステップ (7) に戻る。
{UNIX/Linux の参考書 11.2 節 (p.225～226)}

① エラーメッセージを解読したり、**GDB デバッガ** (プログラムの実行追跡を行って途中の状況を観察するためのツール, このプリントの付録 B を参照) を用いる等してプログラムの誤り箇所を突き止め、Emacs ウィンドウ上で修正する。

② Ctrl-x Ctrl-s とキーを押すことによってソースファイルを更新する。

③ 再度プログラムをコンパイルする。

実行時のエラーについて：プログラム実行時に起こるエラーとしては、次の様なものがあります。

- `Segmentation fault`： プロセスに割り当てられていない領域へのアクセス、または書き込み禁止領域への書き込みを行おうとした (e.g. 配列の添字が確保した範囲を超えた場合)。
- `Bus error`： ワード境界を無視してメモリアクセスを行った。
- `Illegal instruction`： 不正な (機械語) 命令を実行しようとした。
- `Stack Overflow`： プログラムが使用する変数領域が大きくなり過ぎた時に発生する。プログラム自体が多き過ぎることで発生する可能性もあるが、大抵の場合、無限再帰が原因である。
- `Floating point exception`： 0 による除算, オーバーフローまたはアンダーフローが起こった。

(12) プログラムが正しく動作することが確認されたら、おしまい。

□演習 19.1 (プログラム作成・修正・保存・実行) レポート課題 1 で作成した C プログラムのコンパイル、実行、デバッグを上の手順に従って行ってみよ。

□演習 19.2 (C プログラムに行番号を付ける; 読み飛ばし可) レポートを書く際等に、プログラムに行番号が付いていると便利なことがあります。こんな時には、文書ファイルに行番号をつけて表示する UNIX コマンドも役に立ちます。お試し下さい。

□演習 19.3 (gdb デバッガ; 読み飛ばし可) レポート課題 1 で作成した C プログラムの実行を gdb デバッガで追跡してみよ。(付録 B を参照。)

□演習 19.4 (C プログラムの整形; 読み飛ばし可) デバッグしている内に字下げの仕方がめちゃくちゃになり C プログラムが見にくくなった場合には、C プログラムを整形・表示してくれる次の UNIX コマンドも役に立ちます。お試し下さい。

形式:	**indent** `[options]` [*file-name*]	
`options`:	`-kr`	Kernighan と Ritchie のスタイルで整形
	`-gnu`	GNU プロジェクトのスタイルで整形
	`-origs`	BSD の indent 互換スタイルで整形
	`-i`*num*	字下げ幅を *num* として整形

19.2 プログラミング時の注意

(1) 一度に大勢で同じ課題に取り組むためプリンタの出力が誰のだか分からなくなる恐れがあります。そこで、作成する各プログラムの **1**~ **3** 行目は次の様な注釈行にして下さい。

```
/* プログラミング基礎演習 (○曜○限) */
/* 新潟大学○学部○○○学科          */
/* 各自の学籍番号  各自の氏名        */
```

(2) キーボードからのデータ入力を終了させる (i.e. 入力ファイルを閉じる) ためには、Ctrl-d とキーを押します。 [**補足 :** UNIX ではプログラムの停止, 入力の終り, 表示の一時中断, ... といった特殊機能が幾つかのキーに割り当てられており、それをユーザが設定することも出来ます。どの特殊機能がどのキーに割り当てられているかを調べるには `stty -a` とコマンド入力します。]

□**演習 19.5 (読み飛ばし可)** `stty -a` とコマンド入力して、どの特殊機能がどのキーに割り当てられているかを調べてみて下さい。

19.3 レポート課題 **2**~ **6**

最初のガイダンス (15.1 節) の時に言った様に、**課題 2 以降のレポート課題は、この演習とペアになっている「プログラミング概論」の授業の中で出すことにします。**

19.4 レポートの形式

{UNIX/Linux の参考書 11.4.2 節 (p.229)}

GNOME 端末ウィンドウ上の会話の様子をそのまま文書ファイル (**ログファイル**という) として記録するための方法としては、次の様なものがあります。

(方法 1) GNOME 端末ウィンドウ上の会話の様子を、**マウスを用いたコピー・アンド・ペースト**で Emacs 内に取り込む。(16.10 節)
会話が長い場合は間違う可能性もあるが、**簡単**。

(方法 2) `script` コマンドを用いる。(後で補足説明。)

これらの内、**この演習で C プログラミングのレポートを作成する場合は、(方法 1) で**

プログラムを表示, コンパイル, 実行している様子

をファイルとして記録し、この出力に氏名、説明等を書き加えたものをレポートとして提出して下さい。

補足 (`script` コマンド) : GNOME 端末ウィンドウに表示された文字列をそのまま文書ファイル (**ログファイル**という) として記録するための方法としては `script` コマンドを用いる方法もあります。`script` コマンドの使い方は次の通りです。

(1) 図 7 に例示されるように、仮想端末 (GNOME 端末) ウィンドウで

`script` ログファイルの名前

と入力して記録を開始する。

(2) 必要な会話をする。

(**注意**：計算機の処理結果だけでなくプロンプトやキーボードからの入力も (さらには、最終的に画面に表示されるものだけでなく、ミスタイプやそれを削除するコードも含めて)、画面に表示された順に、指定したログファイルに記録していきます。)

(3) `exit` と入力して記録を終了する。

(これによって、例えば図 7 の会話の場合には図 8 に示される内容の `report.log` という名前のファイルが出来ます。)

(4) エディタを使ってログファイルを編集する。

(→詳細については、すぐ後の補足説明をご覧下さい。)

短い会話だと会話の様子をコピー・アンド・ペーストで Emacs に取り込む方が簡単ですが、長い会話の場合には `script` コマンドは有用です。

---

(4) ログファイル編集の作業：

ログファイルには CR(carriage return) コード `^M` が至る所に入っていますから、まずこれを削除します。Emacs の場合は、Esc x replace-string というコマンドを用いて次の様にすれば `^M` を一度にまとめて削除できます。

① (必要なら Esc < とコマンド入力して、) カーソルをファイルの先頭にもって来る。

② Esc `x replace-string` Enter とコマンド入力して文字列の検索置換処理を開始する。(UNIX/Linux の参考書 4.3.5 節, p.60)

③ 画面下に `Replace string:` と表示されるが、これに対しては Ctrl-q Ctrl-m Enter と返答する。

④ 画面下に `Replace string ^M with:` と表示されるが、これに対しては Enter とだけ返答する。

また、記録したコマンド会話列の中でミスタイプをしていれば、そのミスタイプ部も記録されていますから、これを削除する必要があります。

注意：

`script` コマンドで記録されるログファイルには、仮想端末上に表示された全ての文字が記録されます。ですから、Ctrl-p や ↑ キーを押して入力コマンドを再利用した場合は途中に現れるコマンドも全て記録され、そのまま印刷するとそれらのコマンドが重なって印刷されます。

⇨ `script` コマンドで会話の記録をとっている場合は、**Ctrl-p** や ↑ キーで入力コマンドの再利用を行うのは避けた方が無難です。

```
$ script report.log Enter
                              ....ログファイル report.log への記録を開始
script コマンド開始。ファイルは report.log です。
$ cat ex01.c Enter
#include <stdio.h>
main()
{
  int x,y,sum,ceiling;

  scanf("%d %d", &x, &y);
  sum=x+y;
  ceiling=(x+y-1)/y;
  printf("%d+%d=%d\n", x, y, sum);
  printf("ceiling(%d/%d)=%d\n", x, y, ceiling);
}
$ cc ex01.c Enter
$ ./a.out Enter
8 3 Enter
8+3=11
ceiling(8/3)=3
$ ./a.out Enter
999 3 Enter
999+3=1002
ceiling(999/3)=333
$ exit Enter ..............................ログファイルへの記録を終了
```

図 7: script コマンドを用いて会話の様子を report.log に記録

```
script コマンドが 1999年04月05日（月）17時15分06秒 で起動されました。
$ cat ex01.c^M^M
#include <stdio.h>^M
main()^M
{^M
  int x,y,sum,ceiling;^M
^M
  scanf("%d %d", &x, &y);^M
  sum=x+y;^M
  ceiling=(x+y-1)/y;^M
  printf("%d+%d=%d\n", x, y, sum);^M
  printf("ceiling(%d/%d)=%d\n", x, y, ceiling);^M
}^M
$ cc ex01.c^M^M
$ ./a.out^M^M
8 3^M
8+3=11^M
ceiling(8/3)=3^M
$ ./a.out^M
999 3^M
999+3=1002^M
ceiling(999/3)=333^M
$ exit^M^M
exit^M

script コマンドが 1999年04月05日（月）17時16分04秒 で終了しました。
```

図 8: ログファイルの中身 (生成直後)

以上より、レポート作成の手順は次の様になります。

レポート作成の手順:

(1) プログラムの作成、デバッグ。

(2) **copy-and-paste(教科書 16.10 節) を用いてログファイルを作る場合は** 次の会話を行う。

こちらの方が簡単

① プログラムを cat コマンドで表示,
② (ファイルからデータを入力する場合、その) データファイルの中身を cat コマンドで表示,
③ cc コマンドでコンパイル,
④ a.out で何回か実行。(正しく動作することをチェックする。)

(2') script コマンドを用いてログファイルを作る場合は script コマンドを用いて、次の会話の様子を (ログ) ファイルに記録する。(例えば図 7 の様にする。その際、Ctrl-p や ↑ キーで入力コマンドの再利用を行うのは避ける。)

① プログラムを cat コマンドで表示,
② (ファイルからデータを入力する場合、その) データファイルの中身を cat コマンドで表示,
③ cc コマンドでコンパイル,
④ a.out で何回か実行。(正しく動作することをチェックする。)

(3) **copy-and-paste を用いてログファイルを作る場合は**、前ステップ (2) の会話をそのままログファイルとして記録する。(**16.10 節**)

(3') script コマンドを用いてログファイルを作った場合は 前ステップ **(2')** で作ったログファイル (例えば図 **8**) から、改行コード (画面上では^M と表示) を次の様にして取り除く。

① (必要なら **Esc** < とコマンド入力して、) カーソルをファイルの先頭にもって来る。
② **Esc** x replace-string **Enter** とコマンド入力して文字列の検索置換処理を開始する。(**UNIX/Linux の参考書 4.3.5 節, p.60**)
③ 画面下に Replace string: と表示されるが、これに対しては **Ctrl-q Ctrl-m Enter** と返答する。
④ 画面下に Replace string ^M with: と表示されるが、これに対しては **Enter** とだけ返答する。

更に、ステップ **(2)** でミスタイプをしていれば、そのミスタイプ部も削除する。

補足:
2011 年度のシステムでは、**Back space** のコードがログファイルに含まれていると、その部分が **XEmacs** で **unicode** の文字として認識されませんでした。
$\Longrightarrow$ こんな場合は **gedit** を開いて編集して下さい。

(4) 前ステップ **(3)** または **(3')** で得られたログファイルの内容をプリンタに出力し、この出力に次の事柄を書き加える。 [プリンタ出力が **2** 枚以上になる場合は、左上をホッチキスで留めて下さい。]

① 受講している演習の曜限 (右上に),
② 学部, 学科, 学籍番号, 氏名 (右上に; 念のため),

③ プログラムの説明, 実行結果についてのコメント／考察など,
④ 実習, レポート課題などについての意見／感想／不満など。

□**演習 19.6** レポート課題 1 で作成した C プログラムについて、上の手順 (1)〜(3) を実行してみよ。

19.5 レポートの提出先

水曜 4 限 (担当：棚橋→総合研究棟 (情報理工系)8 階 806 室)

⟹ 直接担当者に手渡す。
授業時間外は 総合教育研究棟 (教養棟)A 棟 1 階の教務課の前にレポートボックスを用意しているので、そちらに出しても良いです。

木曜 4 限 (担当：元木→工学部 A2 棟 2 階 205 室)

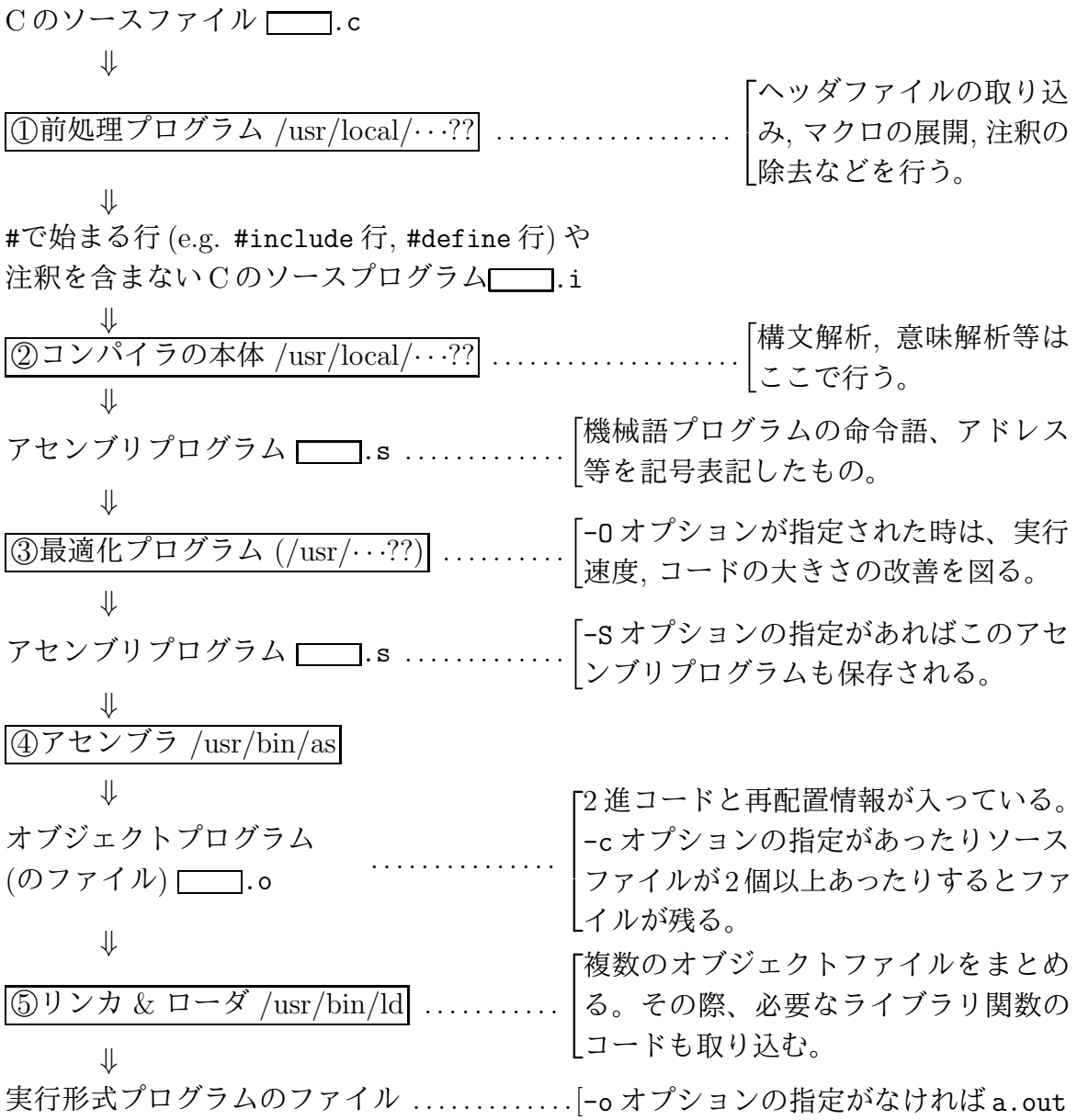
⟹ 直接担当者に手渡す。
授業時間外は 総合教育研究棟 (教養棟)A 棟 1 階の教務課の前にレポートボックスを用意しているので、そちらに出しても良いです。

付 録

A Cコンパイラについて

以下は、Cコンパイラについての一般的なお話です。Cプログラミングの際の参考にして下さい。[cc と gcc に共通した話です。]

cc コマンド／gcc コマンドによるCプログラムの翻訳作業は、次に示される様に①前処理，②コンパイル，③最適化 ，④アセンブリ，⑤リンク&ロード，という5段階の処理を経て行われる。[実際の処理プログラムの名前, 配置ディレクトリはシステムによって異なるかも知れませんが、処理の大きな流れは同じです。]



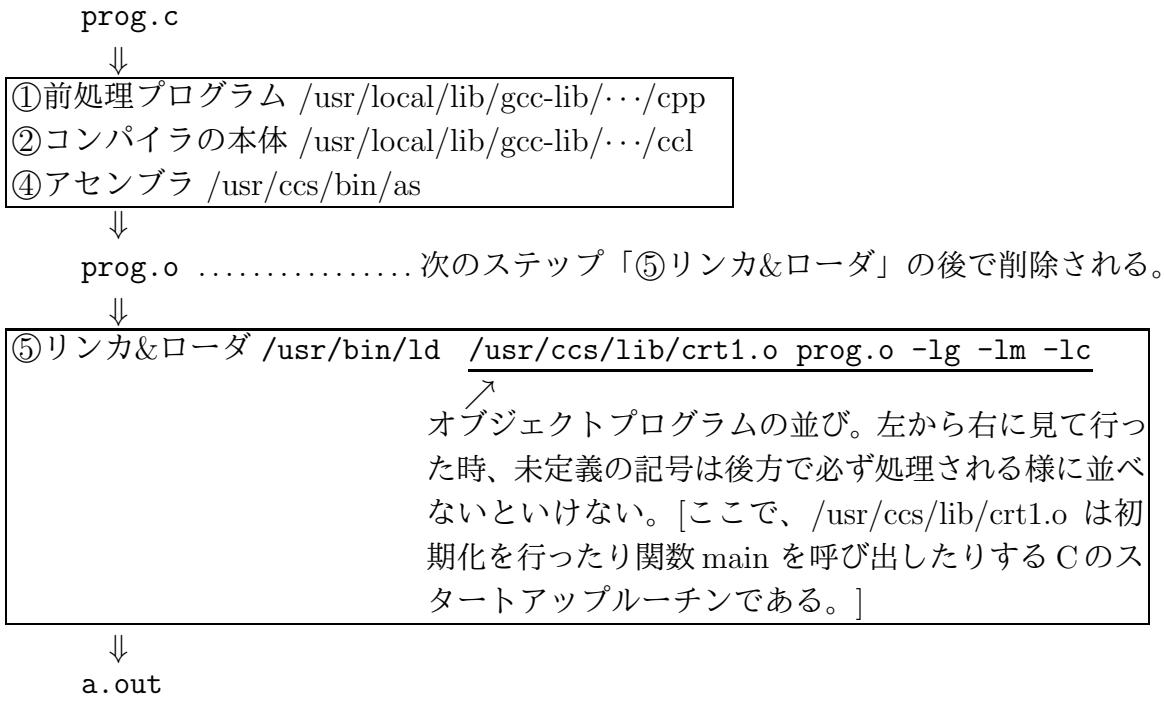
これら5段階の動作を制御するために、cc コマンド／gcc コマンドには様々なオプションが用意されています。 例えば cc コマンドについては次の通り。

形式:	**cc** [*options*] *files* [-l *library*]	
options:	-c	: オブジェクトファイルだけを生成し、リンカ&ローダを呼び出さない。
	-g	: デバッガのための特別なシンボル表もコンパイルの際に生成し、リンカ&ローダ ld に -lg オプションを引き渡す。
	-o ファイル名	: 実行形式プログラムを入れるファイルを a.out ではなくファイル名 とする。
	-O	: 最適化を行う。
	-D 名前 = 定義	(説明省略)
	-U 名前	
	-I パス名	
	-P	
	-S	
	-v	
	-p	
	-pg	
	⋮	
-l *library*:	-lx	: リンカ&ローダにこのオプションをそのまま引き渡し、オブジェクトプログラムをまとめ上げる際、必要に応じてライブラリ (アーカイブファイル) libx.a 内の関数 (オブジェクトコード) を用いることを指示する。 12 個のヘッダファイル <assert.h>, <ctype.h>, <float.h>, <limits.h>, <setjmp.h>, <signal.h>, <stdarg.h>, <stdio.h>, <stdlib.h>, <string.h>, <time.h> 内で宣言された関数は ANSI 標準で、/lib/libc.a というライブラリ (アーカイブファイル) にまとめられている。この標準ライブラリに関する -lc というオプションは cc コマンドに付加しなくても自動的にリンカ&ローダに引き渡されるが、数学ライブラリ関数は ANSI 標準でないため、数学関数を用いた場合は数学ライブラリ/lib/libm.a の使用をリンカ&ローダに申告するための-lm オプションを cc コマンドの最後に付加しなければならない。付加しないと「undefined symbol (未定義の記号がある)」というエラーになる。

詳しくは、man cc でオンラインマニュアルを調べるなり、次の図書を見るなりして下さい。

- 山口和紀 (監), The UNIX SuperText 下, 技術評論社, 1992.
- P.S.Wang, ANSI C & UNIX 下, 共立出版, 1994.

例えば prog.c というCプログラムがある時、% cc -g prog.c -lm とコマンド入力すると計算機内部では次の様な処理が行われます。



B デバッガ GDB について

文法的に正しいプログラムの**虫取り** (debug; i.e. **虫**,bug, すなわち 誤りを取り除くこと) のために、UNIX では gdb とか dbx といったデバッガがよく使われています。 gdb を用いればC(や Fortran, Pascal などの) プログラムの実行を追跡して、対話的にプログラム実行を指定位置で一時停止／中断させたり実行途中の変数値を調べたりできます。

GDB を用いてCプログラムの実行追跡をするには普通次の様にします。

(1) -g オプション を指定して gcc (または cc) コマンドを実行する。 [-g オプションを指定してコンパイルすると、宣言した変数や関数のデータ型、実行形式コードのアドレスとソースコードの行番号の対応、等の**デバッグ情報**がオブジェクトファイルの中に格納されます。]

(2) gdb 実行形式ファイル とコマンド入力して GDB を起動する。 [これによって、(gdb) というプロンプトが現われるはずです。**この状態で** help **と入力すると GDB コマンド群の簡単な説明一覧が表示され、**help コマンド群の名前 **と入力するとそのコマンド群の中のコマンドの簡単な説明一覧が表示され、また、**help コマンド名 **と入力するとそのコマンドの簡単な説明が表示されます。**]

(3) プログラム実行の途中で止まって変数値が意図した通りになっているかどうかをチェックする場所 (**中断点**,breakpoint, という) を指定する。 [実行時のエラーでプログラムが中断される場合は、これを行わずにプログラムを実行させ、エラーで実行中断されてからその時の変数値等を調べてもよい。]

例えば、次の様な指定ができます。

`break` [*filename*:] *linenum* ··· (ソースファイル *filename* の) *linenum* 行目で実行を中断。

`break` [*filename*:] *function* ··· (ソースファイル *filename* の) 関数 *function* の呼び出し直後に実行を中断。

`watch` *exp* ··· 式 *exp* の値が前回と違っていたら実行を中断。(実行速度が著しく低下するので、なるべく使用は避ける。)

(4) (gdb) というプロンプトに対して `run` [*args*] [< *file1*] [> *file2*] とコマンド入力して、GDB の下でプログラムを実行する。[この後、GDB は最初の中断点でプログラムを一時停止させ、コマンド待ちの状態になる。]

(5) プログラムの実行が終了するまで次の操作を繰り返し行う。[行う順序は任意。]

- それまでの実行追跡で表示された事柄を吟味する。
- 現在の中断点における変数値等を表示させる。

  例えば次の様なコマンド入力ができます。

  `print` [`/` *format*] *expr* ··· 式 *expr* の値を表示する。*format* 部 (オプション) には次の指定が可能です。

format	意味
t	2 進表示
o	8 進表示
x	16 進表示
d	符号付き 10 進表示
u	符号なし 10 進表示
f	浮動小数点表示
c	文字表示
a	アドレス表示

  `x` [`/` *size format*] *addr* ··· *addr* で指定されたアドレスから始まり、大きさが *size* ワードの領域の内容を表示する。(このコマンド名は examine の意。) *size* 部は省略すると 1 と見なされ、*format* 部は `print` で許される指定に加え次の指定も可能です。

format	意味
s	文字列表示
i	命令表示 (逆アセンブル)

- 現在の中断点止まる度に変数値等を表示する様に指示する。

  例えば次の様なコマンド入力ができます。

  `display` [`/`*format*] *expr* ··· 式 *expr* の値を表示する。*format* 部 (オプション) には `print` で許される指定が可能です。

- 中断点からの実行を再開する。

例えば次の様なコマンド入力ができます。

`cont`	···	次の中断点まで実行。
`next`	···	次の 1 行だけ実行して中断。 [次が関数呼び出しの時は、関数呼び出しを含む行全体を「次の 1 行」と考える。]
`nexti`	···	次の 1 機械語命令だけ実行して中断。 [次が関数呼び出しの時は、関数呼び出しを「次の 1 機械語命令」と考える。]
`step`	···	次の 1 行だけ実行して中断。 [次が関数呼び出しの時は、関数本体中の最初の 1 行を「次の 1 行」と考える。]
`stepi`	···	次の 1 機械語命令だけ実行して中断。 [次が関数呼び出しの時は、関数本体中の最初の 1 命令を「次の 1 機械語命令」と考える。]

- 前回と同じコマンドを実行する。
  (Enter だけを押す。)

- 中断点を (追加) 指定する。
  (上記 (3) の `break` コマンドと `watch` コマンド)

- 現在の追跡状況等を表示する。

  例えば次の様なコマンド入力ができます。

`where`	···	現在の止まっている中断点での関数の呼出し状況 (どの関数の何行目で関数が呼ばれ、その関数の何行目でまた別の関数が呼ばれ、... といった情報) を表示する。
`info breakpoint`	···	その時点で考慮されている中断点の情報を表示。
`whatis` *name*	···	識別子 *name* の型を表示。
`ptype` *type*	···	データ型 *type* の定義を表示。

- 中断点の指定を解除／復活する。

  例えば次の様なコマンド入力ができます。

`disable` *bp-num*	···	(`info breakpoint` コマンドで表示される) **中断点番号** *bp-num* の中断点を (一時的に) 無効にする。
`enable` *bp-num*	···	中断点番号 *bp-num* の中断点を有効にする。
`delete` *bp-num*	···	中断点番号 *bp-num* の中断点の登録を抹消する。
`clear` *position*	···	プログラム上の位置 *position* に設定されている中断点の登録を抹消する。

- ソースプログラムの一部を表示する。

  例えば次の様なコマンド入力ができます。

`list`	···	現在の止まっている中断点付近のソースコードを表示。
`list` [*filename*:] *line-num*	···	*line-num* 行目付近のソースコードを表示。

- プログラムを無視して、変数の値を強制的に変えてみる。

  例えば次の様なコマンド入力ができます。

`set variable` *var* = *expr*	···	式 *expr* の値を変数数 *var* に代入する。

- 現在の実行を強制終了させる。
  (Ctrl-c を押す。)

(6) まだ実行追跡を行いたければ (3) または (4) に戻る。

(7) quit と入力して GDB を終了。

例 B.1 (1 ステップずつ実行・追跡)　図 7 で示したプログラムの場合、GDB を用いて次の様にプログラムの実行追跡を行うことが出来ます。[ここで、下線部はキーボードからの入力を表す。また、補足説明の必要な箇所にはその右側に注釈／説明を加えてあります。]

```
xcspc70_41% cat -n lab-ex01.c ..... 空行にも行番号を付けておく
     1  #include <stdio.h>
     2  main()
     3  {
     4    int x,y,sum,ceiling;
     5
     6    scanf("%d %d", &x, &y);
     7    sum=x+y;
     8    ceiling=(x+y-1)/y;            /* x/y の小数点以下切り上げ */
     9    printf("%d+%d=%d\n", x, y, sum);
    10    printf("ceiling(%d/%d)=%d\n", x, y, ceiling);
    11  }
xcspc70_42% gcc -g lab-ex01.c
xcspc70_43% gdb a.out
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.16 (i486-sun-solaris2.6),
Copyright 1996 Free Software Foundation, Inc...
(gdb) break main ..... 中断点を関数 main の入口に設定
Breakpoint 1 at 0x80487e7: file lab-ex01.c, line 6.
(gdb) run ..... 実行開始
Starting program: /home/home204/faculty/motoki/C-Java/Programming-C/a.out
warning: Unable to find dynamic linker breakpoint function.
warning: GDB will be unable to debug shared library initializers
warning: and track explicitly loaded dynamic code.

Breakpoint 1, main () at lab-ex01.c:6
6         scanf("%d %d", &x, &y); ..... 次に実行する行が表示されている。
(gdb) step ..... 1 ステップ実行
8 3 ..... プログラムへの入力
7         sum=x+y;
```

```
(gdb) print x .....この時点での x の値を表示
$1 = 8
(gdb) print y .....この時点での y の値を表示
$2 = 3
(gdb) print sum
$3 = 3 .....{ まだ sum には値がセットされていない。}
(gdb) step .....1 ステップ実行
8         ceiling=(x+y-1)/y;              /* x/y の小数点以下切り上げ */
(gdb) print sum
$4 = 11
(gdb) cont .....次の中断点 (無い) まで実行
Continuing.
8+3=11
ceiling(8/3)=3

Program exited with code 017.
(gdb) quit .....GDB を終了
xcspc70_44%
```

例 B.2 (ループの中に中断点を設けて実行追跡) 繰り返し処理のあるプログラムの場合、GDB を用いて次の様にプログラムの実行追跡を行うことが出来ます。[ここで、下線部はキーボードからの入力を表す。また、補足説明の必要な箇所にはその右側に注釈／説明を加えてあります。]

```
xcspc70_41% cat -n lab-ex03.c .....空行にも行番号を付けておく
     1  /* 階乗の計算 */
     2
     3  #include <stdio.h>
     4
     5  main()
     6  {
     7    int   i,n;
     8    float fact;
     9
    10    scanf("%d", &n);
    11    fact = 1;
    12    for (i=2; i<=n; ++i)
    13      fact = fact * i;
    14    printf("%d! = %.0f\n", n, fact);
    15  }
xcspc70_42% gcc -g lab-ex03.c
xcspc70_43% gdb a.out
GDB is free software and you are welcome to distribute copies of it
```

```
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.16 (i486-sun-solaris2.6),
Copyright 1996 Free Software Foundation, Inc...
(gdb) break 13
Breakpoint 1 at 0x8048810: file lab-ex03.c, line 13.
(gdb) run
Starting program: /home/home204/faculty/motoki/C-Java/Programming-C/a.out
warning: Unable to find dynamic linker breakpoint function.
warning: GDB will be unable to debug shared library initializers
warning: and track explicitly loaded dynamic code.
3 ..........................................................プログラムへの入力

Breakpoint 1, main () at lab-ex03.c:13
13          fact = fact * i;
(gdb) display i ..............この中断点に止まる度に変数 i の値を表示する様に指示
1: i = 2
(gdb) display fact
2: fact = 1
(gdb) cont
Continuing.

Breakpoint 1, main () at lab-ex03.c:13
13          fact = fact * i;
2: fact = 2
1: i = 3
(gdb)  ..........................Enter のみ打って、前回と同じコマンド実行を指示
Continuing.
3! = 6

Program exited with code 07. ..........................プログラムの実行が終了
(gdb) info breakpoint .....................現在設定されている中断点の情報を表示
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x08048810 in main at lab-ex03.c:13
        breakpoint already hit 2 times
(gdb) disable 1 ..................... 中断点番号 1 の中断点を一時的に無効にする
(gdb) info breakpoint
Num Type           Disp Enb Address    What
1   breakpoint     keep n   0x08048810 in main at lab-ex03.c:13
        breakpoint already hit 2 times
(gdb) enable 1 ..........................中断点番号 1 の中断点を有効なものに戻す
(gdb) info breakpoint
```

```
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x08048810 in main at lab-ex03.c:13
        breakpoint already hit 2 times
(gdb) run .....2度目のプログラム実行
Starting program: /home/home204/faculty/motoki/C-Java/Programming-C/a.out
warning: Unable to find dynamic linker breakpoint function.
warning: GDB will be unable to debug shared library initializers
warning: and track explicitly loaded dynamic code.
3 .....プログラムへの入力

Breakpoint 1, main () at lab-ex03.c:13
13          fact = fact * i;
2: fact = 1
1: i = 2
(gdb) cont
Continuing.

Breakpoint 1, main () at lab-ex03.c:13
13          fact = fact * i;
2: fact = 2
1: i = 3
(gdb)  .....Enter のみ
Continuing.
3! = 6

Program exited with code 07.
(gdb) quit
xcspc70_44%
```

例 B.3 (幾つかの関数から成るプログラムの実行追跡) プログラムが幾つかのモジュールに階層的に分割されている場合は、関数呼び出し時のパラメータと関数終了 (i.e. return) 時の主要変数値を観察することによって、プログラムの動作を追跡することが出来ます。例えば、再帰的関数を含むプログラムの場合、GDB を用いて次の様にプログラムの実行追跡を行うことが出来ます。

```
xcspc70_41% cat -n lab-ex04.c
     1  /* 関数呼出しによる階乗計算 */
     2
     3  #include <stdio.h>
     4
     5  long factorial(int);
     6
     7  main()
```

```
 8  {
 9    int n;
10
11    scanf("%d", &n);
12    printf("%d! = %ld\n", n, factorial(n));
13  }
14
15  /*************************************/
16  /* 階乗計算の関数                    */
17  /* ---------------------------------- */
18  /*   入力パラメータ n : 非負整数を想定 */
19  /*   関数値           : n!             */
20  /*************************************/
21  long factorial(int n)
22  {
23    long fact;
24
25    if (n == 0)
26      fact = 1;
27    else
28      fact = n * factorial(n-1);
29
30    return(fact);
31  }
xcspc70_42% gcc -g lab-ex04.c
xcspc70_43% gdb a.out
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.16 (i486-sun-solaris2.6),
Copyright 1996 Free Software Foundation, Inc...
(gdb) break factorial
Breakpoint 1 at 0x8048846: file lab-ex04.c, line 25.
(gdb) break 30
Breakpoint 2 at 0x8048878: file lab-ex04.c, line 30.
(gdb) run
Starting program: /home/home204/faculty/motoki/C-Java/Programming-C/a.out
warning: Unable to find dynamic linker breakpoint function.
warning: GDB will be unable to debug shared library initializers
warning: and track explicitly loaded dynamic code.
3 .....プログラムへの入力

Breakpoint 1, factorial (n=3) at lab-ex04.c:25
25        if (n == 0)
```

```
(gdb) step
28          fact = n * factorial(n-1);
(gdb)

Breakpoint 1, factorial (n=2) at lab-ex04.c:25
25        if (n == 0)
(gdb) cont
Continuing.

Breakpoint 1, factorial (n=1) at lab-ex04.c:25
25        if (n == 0)
(gdb)
Continuing.

Breakpoint 1, factorial (n=0) at lab-ex04.c:25
25        if (n == 0)
(gdb)
Continuing.

Breakpoint 2, factorial (n=0) at lab-ex04.c:30
30        return(fact);
(gdb) display fact
1: fact = 1
(gdb) step
31      }
1: fact = 1
(gdb)
factorial (n=1) at lab-ex04.c:30
30        return(fact);
1: fact = 1
(gdb) cont
Continuing.

Breakpoint 2, factorial (n=2) at lab-ex04.c:30
30        return(fact);
1: fact = 2
(gdb)
Continuing.

Breakpoint 2, factorial (n=3) at lab-ex04.c:30
30        return(fact);
1: fact = 6
(gdb)
```

```
Continuing.
3! = 6

Program exited with code 07.
(gdb) quit
xcspc70_44%
```

C　実習室 Linux で WWW にアクセスするためには...

{UNIX/Linux の参考書第 6 章 }

- 実習室 Linux で利用できる WWW ブラウザは Firefox です。
- 実習室 Linux で Firefox を起動するには、GNOME 端末ウィンドウ上で

  `firefox &`

  とコマンド入力するか、メインメニューから

  アプリケーション ⟶ インターネット ⟶Firefox ウェブブラウザ

  と選択します。

  補足：実習室では Windows 環境の中に Linux の X-Window 環境を構築しているので、Linux 使用時でも外側の Windows 環境の中で Internet Explorer を使うことができる。
  ⟹ 使い慣れている方を使えば良い。

- 新潟大学, 新潟大学工学部のホームページのアドレスは、それぞれ

  `http://www.niigata-u.ac.jp/`　,

  `http://www.eng.niigata-u.ac.jp/`

  です。

D　実習室 Linux で電子メールを扱うためには...

{UNIX/Linux の参考書第 5 章 }

- 各自のメールアドレスは

  各自のユーザ id(学籍番号)`@mail.cc.niigata-u.ac.jp`

  です。
- 実習室では、Linux 上で電子メールを出したり読んだりするためのソフト (メールリーダ) として Thunderbird が用意されています。
- 実習室 Linux で Thunderbird を起動するには、GNOME 端末ウィンドウ上で

  `thunderbird &`

  とコマンド入力するか、メインメニューから

  アプリケーション ⟶ インターネット ⟶Thunderbird Email

  と選択します。

  補足：実習室では Windows 環境の中に Linux の X-Window 環境を構築しているので、Linux 使用時でも外側の Windows 環境の中で Thunderbird を使うことができる。
  ⟹ 使い慣れている方を使えば良い。

E USB メモリの利用, 文字コードの変換, パッケージ化

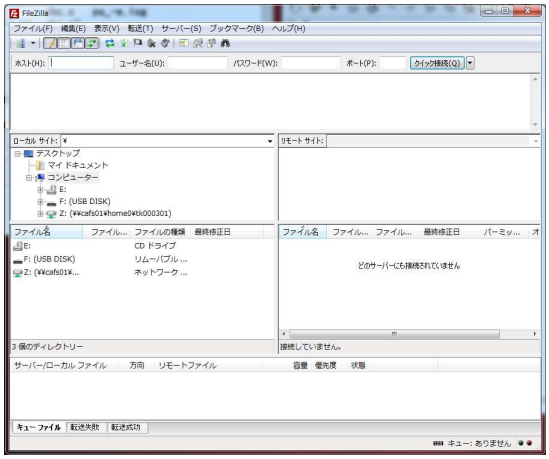
USB メモリの利用： Windows のファイルシステムと Linux のファイルシステムが繋がってなく、また実習室 PC に USB メモリを接続すると Windows のファイルシステムに組み込まれる (マウントという) ため、Linux 上で作成したファイルを USB メモリに保存したり、逆に USB メモリ内のファイルを Linux のファイルシステム内にコピーするのは、多少面倒な作業になる。これは、例えば、Windows7 内にインストールされている FileZilla と呼ばれる FTP クライアントソフトを利用して、次の様に行う事ができる。

(1) **USB メモリを PC に接続**：USB メモリを PC に差し込み、Windows のファイルシステムに組み込まれたことを確認する。

(2) **FileZilla の起動**：Windows 画面左下のスタートボタンを押して現れるメニューから FileZilla FTP Client→FileZilla を選ぶ。

操作の様子は次の通り。

これで、次の様なウィンドウが生成されるはずです。



また、最初に FileZilla を起動した時などは、このウィンドウの上に次の様なダイアログが表示される。

(3) **FileZilla から Linux サーバへの接続手続き：**

(3.1) **サイトマネージャの呼出し：**

FileZilla ウィンドウ左上のメニューで ファイル (F)→ サイトマネージャー (S)...

と選択する。

すると、次の様なウィンドウが現れるはずです。

(3.2) **新しい接続先 (サイト) 設定の申し出：**

サイトマネージャーのウィンドウ左側中程にある「新しいサイト (N)」ボタンをクリックする。

すると、サイトマネージャーウィンドウは次の様になり、右側のフィールドに文字列を入力できる様になる。

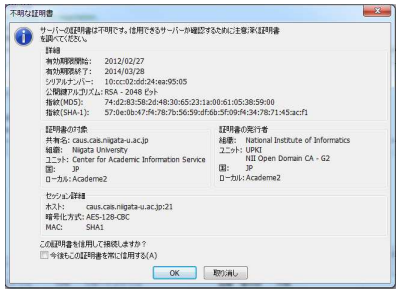
(3.3) **新しい接続先 (サイト) の情報を設定：**

サイトマネージャーウィンドウ右側の「一般」タブ内を次の様に設定する。

- ホスト (H): ··· `caus.cais.niigata-u.ac.jp`
- ポート (P): ··· (空白のまま)
- Protocol: ··· FTP -File Transfer Protocol
- Encryption: ··· Require explicit FTP over TLS
- ログオンの種類 (L): ··· 通常
- ユーザー: ··· (学籍番号, 小文字)
- パスワード: ··· (Windows7 にログインする際に使ったのと同じもの)

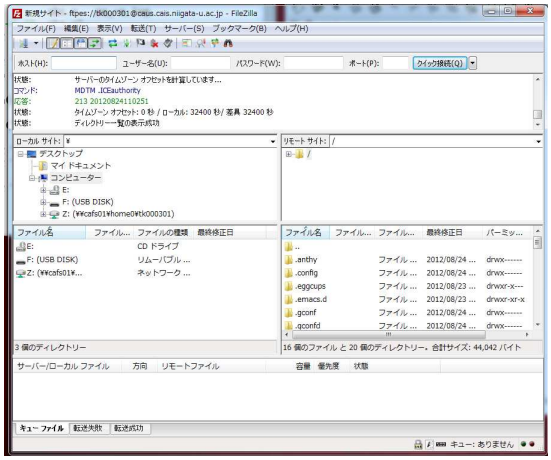
(3.4) **Linux サーバに接続：**

サイトマネージャーウィンドウ下部の「接続 (C)」ボタンを押す。すると、サイトマネージャーウィンドウは消え、代わりに次の様なダイアログが現れる。



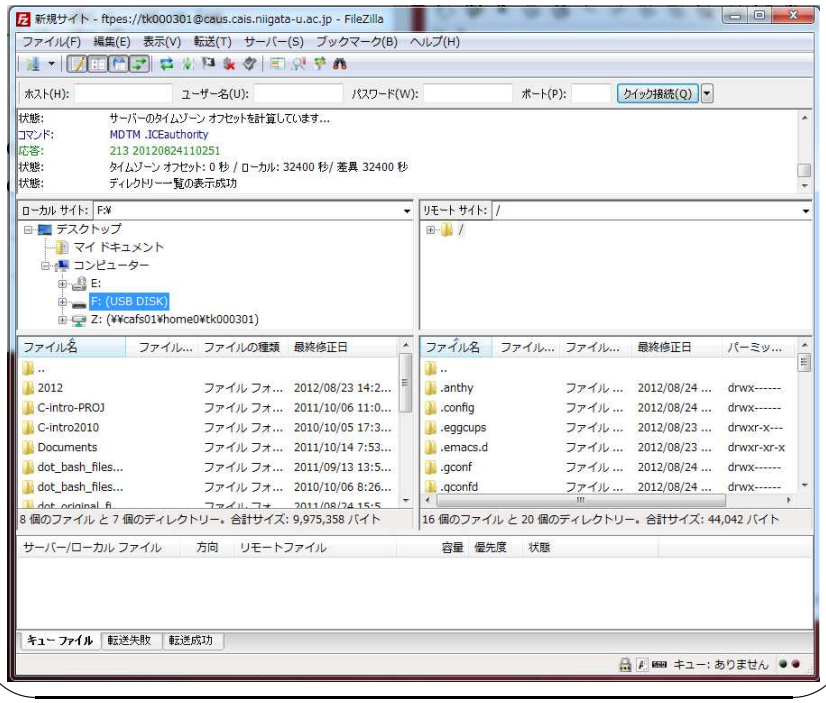
これに対しては、下から 2 行目「今後もこの証明書を常に信用する (A)」という文言の左側にチェックマークを入れ、最下段の「OK」ボタンを押す。

これで、Linux サーバとの接続が確保され、FileZilla ウィンドウは次の様になる。



(4) ファイル転送：
FileZilla ウィンドウ左側中央付近の「F:(USB DISK)」という表示を (ダブル) クリックする。これで、FileZilla ウィンドウの左側中央付近に USB ディスク内のファイルの様子が表示され、右側中央付近に Linux ファイルシステムのホームディレクトリの様子が表示される。

FileZilla ウィンドウの様子は次の通り。



後は、ディレクトリ名をダブルクリックしてディレクトリを移ったり、マウスのドラッグ&ドロップ操作で　USB メモリ ↔Linux ファイルシステム　間のファイル転送を行なったりするだけである。　(**注意**：「..」は親ディレクトリ (i.e.1 つ上のディレクトリ) を表す。)

(5) **FileZilla の終了**：
FileZilla ウィンドウ左上のメニューで　ファイル (F)→ 終了 (X)　と選択する。

文字コードの変換： 次の要領で行うことができます。

文字コードの変換 (任意→ EUC)	···	`nkf -e <`	テキストファイル名	`>` EUC ファイル名
文字コードの変換 (任意→ JIS)	···	`nkf -j <`	テキストファイル名	`>` JIS ファイル名
文字コードの変換 (任意→シフト JIS)	···	`nkf -s <`	テキストファイル名	`>` シフト JIS ファイル名
文字コードの変換 (任意→ UTF-8)	···	`nkf -w <`	テキストファイル名	`>` UTF-8 ファイル名

ファイル群のパッケージ化： 次の要領で行うことができます。

パッケージ化	···	`tar cf`	`.tar`	ディレクトリ名
		(ディレクトリ以下のファイル群を 1 つにまとめる)		
パッケージを解く	···	`tar xf`	`.tar`	ディレクトリ名
パッケージの内容を確認	···	`tar tf`	`.tar`	

F トラブル対策

キーボードから文字が打ち込めない。

⟹ **間違って Ctrl-s と打ち込んだ場合は:** Ctrl-q と打ち込む。[Ctrl-s と打ち込むと入力表示が一時中断される。]

エディタが異常終了した場合等は: Ctrl-j と打ち込んで反応があるかどうか見てみる。もし反応があるなら、Ctrl-j `tset` Ctrl-j とコマンド入力する。[文字は打ち込めるが、画面に表示されなかったり Enter キーが無効になっていた場合に有効な処置。Ctrl-j は Enter の代わりになる。]

cc コマンドがプログラムファイルを受け付けてくれない。

⟹ プログラムファイル名の最後が `.c` で終わっているかどうかを調べる。 もしそうなっていなければ、ファイル名の最後を `.c` という風に変更する。

プログラムが終了しない。

⟹ **入力データを終了させたい場合は:** Ctrl-d とキーを打って end-of-file のコードを入力する。

無限ループを終了させたい場合は: Ctrl-c キーを押す。[あるいは Ctrl-z で実行を一時中断させてから、終了させたいプロセスの番号を確認 (`ps -e`) の上、該当プロセスを `kill` する。]

コンパイラを走らせると

```
ld: a.out: fatal error: cannot create output file.
Text file busy(errrno = 26).
```

というエラーメッセージが出る。

$\Longrightarrow$ Ctrl-z によって実行中断中のプロセスがあると思われます。ps コマンドで調べてこのプロセスを kill して下さい。

索 引

記　号

G

H

I

J

K

か 行

さ 行

た 行

な 行

は 行

ま 行

わ　行