プログラミング AI
(C プログラミング)

新潟大学工学部
知能情報システムプログラム２年第１ターム
協創経営プログラム　　３年第１ターム

新潟大学創生学部
知能情報システム領域学習科目パッケージ ２年第１ターム

平成 30 年 4 月 5 日

元木 達也
motoki@ie.niigata-u.ac.jp

目 次

0　ガイダンス

- 受講に当っての留意事項
- 授業の目標
- 教科書、参考書
- 授業予定

0.1　受講に当っての留意事項

旧カリキュラムにおける位置付け

- 平成 28 年度以前の情報工学科受講生においては、(合格したら)「プログラミング実習 I」に読み替えられる。

必要な予備知識

- この講義／実習は**プログラミングの入門コースではない**ので、1 年次第 3~4 タームの「プログラミング基礎 I, II」を既に履修して (ある程度の理解をして) いることを前提に話を進める。

授業の進め方

- 講義と演習／実習を交互に行う。$\Longrightarrow$ 基本的には、3 限は講義、4 限は演習／実習。
- C プログラムの書き方の詳細は講義ノートや参考書等に書かれているので、**授業では細かい話はしない。**

講義ノートや 参考書等を予め予習をして、授業を聞いても分からない部分は質問するようにして下さい。

授業に出席するだけではこの授業の単位を取れないことを自覚して下さい。

0.2　達成目標

- **C 言語**を自在に使いこなせるようになる。

例えば、
必要に応じてデータ構造を定義できる、動的データ構造も扱える、など。

C 言語(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る。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。

- UNIX プログラミング環境に慣れ、大規模なプログラムを効率的に開発するための考え方を理解する。

例えば、
make, ... など。

知能情報システムプログラムにおける到達目標との対応:

対応	プログラムの到達目標
	（１）知識・理解
	a)
	b)
○	c) コンピュータのソフトウェアに関する基礎的知識を修得する。
	d)
	e)
	（２）当該分野固有の能力
	a)
	b)
○	c) プログラム等の要求条件を理解し、プログラム設計等の作業スケジュールを立て、プログラム作成等を計画通りに実行できる。
	（３）汎用的能力
	a)
	b)
	c)
	d)
	e)
	（４）態度・姿勢
	a)
	b)
	c)

0.3 教科書、参考書

教科書：

講義ノート等を pdf の形で Web に配置しておくので、各自で download を行い必要な箇所の印刷を行なって下さい。

C 言語に関する参考書：

- A. ケリー&I. ポール「C の ABC (上)」(1993 年, アジソン・ウェスレイ/星雲社, 2718 円＋税)
- A. ケリー&I. ポール「C の ABC (下)」(1993 年, アジソン・ウェスレイ/星雲社, 1942 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)
- S.Oualline「C 実践プログラミング第 3 版」(1998 年, オライリー・ジャパン/オーム社, 4500 円＋税)
- H. シルト「独習 C 第 4 版」(2007 年, 翔泳社, 3200 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)
- P.S.Wang「ANSI C & UNIX 上」(1993 年, 共立出版, 3600 円＋税)
- P.S.Wang「ANSI C & UNIX 下」(1994 年, 共立出版, 4500 円＋税)
- H.M.&P.J. ダイテル「C 言語プログラミング」(1998 年, プレンティスホール出版, 3600 円＋税)

UNIX プログラミング環境に関する参考書：

- 遠藤俊徳「GCC GNU C Compiler Manual & Reference 増補改訂版」(1999 年, 秀和システム, 2000 円＋税)
- A.Robbins「GDB ハンドブック」(2005 年, オライリー・ジャパン/オーム社, 1200 円＋税)
- R.M.Stallman&R.H.Pesch「GDB 入門」(1999 年, アスキー出版局, 1900 円＋税)
- 山口和紀&古瀬一隆 (監)「新 The UNIX Super Text 下 改訂増補版」(2003 年, 技術評論社, 3670 円＋税)
- 工藤智行「UNIX プログラミングの道具箱」(2004 年, 技術評論社, 2480 円＋税)
- N.Matthew&R.Stones「Linux プログラミング 例題で学ぶ UNIX プログラミング環境のすべて」(1999 年, ソフトバンク, 4200 円＋税)
- 芦田幸治「LINUX デベロッパーズバイブル」(2001 年, ソフトバンク, 4800 円＋税)
- R.Mecklenburg「GNU Make 第 3 版」(2005 年, オライリー・ジャパン/オーム社, 2800 円＋税)
- A.Oram&S.Talbott「make 改訂版」(1997 年, オライリー・ジャパン/オーム社, 1800 円＋税)
- 伊藤和人「入門 Make & RCS」(1998 年, 秀和システム, 2200 円＋税)
- 古川芳孝「(UNIX Literacy Series Vol.10) make, lint, dbx の使い方」(1992 年,HBJ 出版局, 1942 円＋税)

0.4 授業予定

	3 限	4 限
1 回	●**処理の選択と繰り返し**... プログラム内部の変数の状態遷移を基に処理手順を構築する例題； ●**関数 (その 1)**... 関数定義, どの様にコンパイルが進むか, 再帰	**実習課題 1**: 変数の状態遷移を理解した上で、しっかりした構造の C プログラムを作る課題に取り組む。 最後は、LaTeX でレポートを完成させる。
2 回	●**関数 (その 2)**... 関数パラメータの受渡し, 関数呼出しの実装, 再帰計算 vs. 反復計算； ●**モジュール化について**... モジュール化, 静的外部変数等を利用した関数以外のモジュール	
3 回	●**構造体, 共用体,typedef**； ●**動的データ構造**... 自己参照的構造体	**実習課題 2**: モジュール化、配列の使い方に慣れるための課題に取り組む。 最後は、LaTeX でレポートを完成させる。
4 回	●**動的データ構造**... 線形連結リスト, 2 分木	

5 回	•**2 分木,push-down スタック,待ち行列** ... 配列を用いた 2 分木の実装, push-down スタックの実装	**実習課題 3**: 必要に応じてデータ構造を定義したり、動的データ構造を扱ったりするための能力を養うための課題に取り組む。 最後は、LaTeX でレポートを完成させる。
6 回	•**UNIX プログラミング環境**... プログラムの計算時間の測定	
7 回	•**UNIX プログラミング環境**...Makefile を用いた分割コンパイル, ライブラリファイルの作成・利用; • **プリプロセッサ**... ヘッダファイルの構成・利用, マクロ, 引数付きマクロ	**実習課題 4**: Makefile を用いたプログラムの保守・管理を経験するための課題に取り組む。 最後は、LaTeX でレポートを完成させる。
8 回	ターム末試験	

以下、第 1〜第 4 節では C 言語の復習も兼ねて、アルゴリズムを如何に構築していくかに重点をおいて説明する。

$\Longrightarrow$ プログラムを説明する際の参考にもなります。

1 復習,自習 C の基本構文

- 注釈,
- 変数の宣言, int 型, float 型, double 型, 定数,
- キャスト演算, 代入式,
- 混合演算, 自動型変換,
- `scanf`, `printf`,
- `#include`, `#define`,
- for 文,
- 増分演算子と減分演算子,
- 一次元配列

1.1 基本的なプログラム例

ここでは、①データの入力、②算術式の計算、③結果の出力、という単純な処理の流れが C 言語でどう表されるか例示する。

例題 1.1 (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。

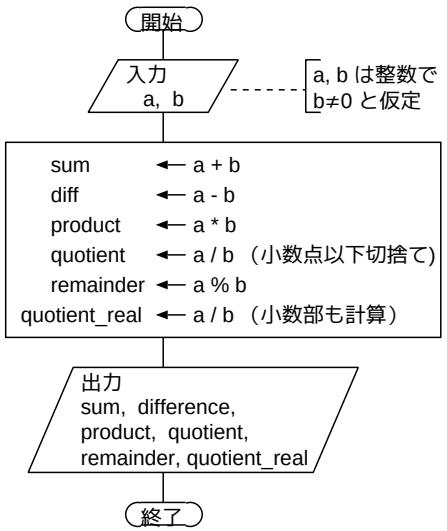
商としては小数点以下を切捨てたものと小数部も計算したものの 2 つを考える。指定された処理のためには、(プログラムの外から) 読み込むデータを格納する場所が必要である。 そこで、 これらの記憶領域を用意し各々 a, b という名前を付け、また、

和, 差, 積, 商 (小数点以下切捨て),
除算の際の余り, 商 (小数部も計算)

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, quotient_real`

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 fundamentals-arith.c  Enter
     1  /* 2 つの整数データを変数 a と b に読み込み、それらの和, */
```

```
 2  /* 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    double quotient_real;

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

 9    sum      = a+b;  /* 和 */
10    diff     = a-b;  /* 差 */
11    product  = a*b;  /* 積 */
12    quotient = a/b;  /* 商 */
13    remainder= a%b;  /* 除算の際の余り */
14    quotient_real = (double)a / (double)b;

15    printf("\nInput data: %d, %d\n\n"
16           "Sum:        %d\n"
17           "Difference: %d\n"
18           "Product:    %d\n"
19           "Quotient:   %d\n"
20           "Remainder:  %d\n"
21           "Quotient (over real): %g\n",
22           a, b, sum, diff, product, quotient, remainder,
23           quotient_real);

24    return 0;
25  }
```

```
[motoki@x205a]$ gcc fundamentals-arith.c Enter
[motoki@x205a]$ ./a.out Enter
11 3 Enter

Input data: 11, 3

Sum:        14
Difference: 8
Product:    33
Quotient:   3
Remainder:  2
Quotient (over real): 3.66667
[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です。

> 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。

```
/* 2つの整数データを変数 a と b に読み込み、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      */

#include <stdio.h>

int main(void)
{
  int    a, b, sum, diff, product, quotient, remainder;
                          ..............
```

- 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7~9行目の部分を1行にまとめて

```
double quotient_real;scanf("%d%d",&a,&b);sum=a+b;/*和*/
```

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 4~25 行目 はひとまとまりの処理を表す。このうち 4 行目は処理に名前を付けた部分、5~25 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 4~25 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 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。

> 補足：
> `/usr/include/stdio.h` の中に、8 行目, 15 行目で呼び出される入出力関数 scanf と printf の引数の型、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 6~7 行目 は、7 つの int 型データのための変数領域と、1 つの double 型データのための変数領域を確保し、それぞれ `a`, `b`, `sum`, `diff`, `product`, `quotient`, `remainder`, `quotient_real` と名付けることをコンパイラに知らせる宣言文。「`int`」と指定することにより、確保した領域が 整数を表すための最も標準的な内部表現形式 (5.3 節) に従ってデータを保持する様になり、「`double`」と指定することにより、確保した領域が 実数を表すための (最小の) 倍の精度の内部表現形式 (5.7 節) に従ってデータを保持する様になる。

> 補足：
> int や double 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。$\Longrightarrow$ 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。$\Longrightarrow$ 1.4.2 節を参照。
>
> 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 8 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。 この関数の第 1 引数は書式を表す文字列、2 番目以降の引数は入力データを格納する番地になっている。 この 8 行目の入力書式の中の `%d` は、読み込

んだデータを 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 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 9~14 行目 は代入文である。 具体的には、プログラムの 9 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る。 同様に、プログラムの 10~14 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,
  quotient_real⟵ a / b, ...(小数部も計算し、結果は実数になる)

  という動作を表している。
- C 言語においては、等号 = は代入演算子であり、変数 = 式 というもの自身が値をもつ式として扱われる。

> 補足：
> 変数に値が代入されるのは単なる副作用と考える。

- C 言語においては、セミコロン (;) は文と文を句切る記号ではなく、文の終わりを表す記号である。従って、sum=a+b は式であり、これにセミコロンが付いて文になる。
- 変数だけでなく、どの式にもデータ型が決まっている。例えば、int 型 / int 型 と書くと内部では int 型の除算が行われ結果は int 型になり、double 型 / double 型 と書くと内部では double 型の除算が行われ結果は double 型になる。
- プログラム 14 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。
- プログラムの 15~23 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。15~21 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 (22~23 行目) の式の値が出力される。 [15~21 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 7 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
Quotient␣(over␣real):␣第 9 引数で指定された式 quotient_real 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 %g の部分が第 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 %g は別途指定された式の値が浮動小数点数型 (実数型) 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%g の場所に出力することを指示する。
◇ 出力書式中の \n, %d, %g 以外の文字列はそのまま出力される。

- プログラムの 24 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ある。 -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る。
- 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することを意味する。

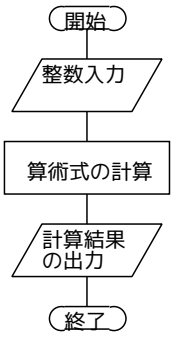
注目点：

- 式のそれぞれがデータ型をもつ。
- C 言語においては、"=" は「代入」という副作用を持った演算子として扱われる。
  $\Longrightarrow$ 変数 = 式 という形のものは**代入式**と呼ばれ、値を持つ。
- scanf( ) や printf( ) も C 言語の構文ではなく関数呼び出し。
  従って、関数値を持つ式として扱われる。
- 式の後にセミコロン (;) を付けたものが文。

例題 1.2 (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) 処理の流れは例題1.1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"
    12          "     = %f\n",
    13          r, h, PI*r*r*h/3.0);

    14   return 0;
    15 }
[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、

- cc, gcc といった C 言語処理系は C のソースコードを機械語に翻訳する前に**前処理**を行う。プログラム 5~6 行目 の#で始まる行はその前処理で何を行うかの指示をしている。
- 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 (最小の) 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 11~12 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 13 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 */
```

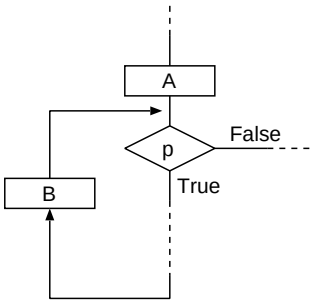
```
 7 int main(void)
 8 {
 9   float r, h;

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"
12          "     = %f\n",
13          r, h, PI*r*r*h/3.0f);

14   return 0;
15 }
[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、

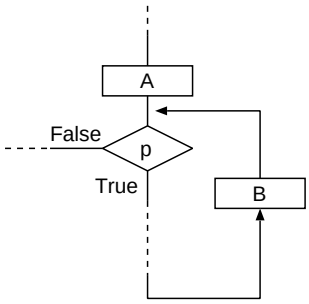
- プログラムの 6 行目 の 3.1415926f は float 型の実数値定数を表す文字列である。double 型定数と区別するために最後に f という文字が付いている。コンピュータの機種にも依存するが、float 型では 23 ビット,6~7 桁の有効桁を保持することが多いため、ここでは 8 桁分の精度で指定した。
- プログラム 9 行目 は、2 つの float 型データのための変数領域を確保し、それぞれ r, h と名付けることをコンパイラに知らせる宣言文である。 float 型は実数を表すための単精度の内部表現形式に対応したデータ型である。
- プログラム 10 行目 の入力書式中の %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 11~12 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 13 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。

1.2 処理の規則的な繰り返し

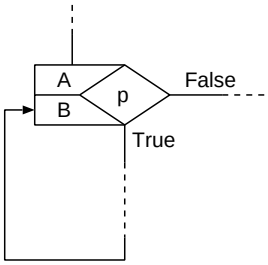
繰り返しの箱: 流れ図を用いて処理手順を表した場合、流れ図上では処理の繰り返しを表すために次の様なパターンが良く現れる。



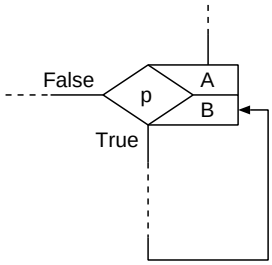
,



以下、これらの処理パターンを表すのにそれぞれ次の様な略記法を用いることにする。



,



そして、この  という形の箱をここでは**繰り返しの箱**と呼ぶ。

例題 1.3 (階乗;for 文の基本形) 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) 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。

$$
\begin{array}{lrcl}
\text{(step 1)} & 1! & = & 1 \\
\text{(step 2)} & 2! & = & 1!\times2 \quad = \quad 1\times2 \quad = \quad 2 \\
\text{(step 3)} & 3! & = & 2!\times3 \quad = \quad 2\times3 \quad = \quad 6 \\
\text{(step 4)} & 4! & = & 3!\times4 \quad = \quad 6\times4 \quad = \quad 24 \\
\text{(step 5)} & 5! & = & 4!\times5 \quad = \quad 24\times5 \quad = \quad 120 \\
 & & & \cdots\cdots\cdots\cdots\cdots\cdots \\
\text{(step } k\text{)} & k! & = & (k-1)!\times k \quad = \quad \cdots\cdots\cdots
\end{array}
$$

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

$$
\begin{array}{lrcl}
\text{(step 1)} & \boxed{1!\text{ の値を保持する変数}} & \longleftarrow & 1 \\
\text{(step 2)} & \boxed{2!\text{ の値を保持する変数}} & \longleftarrow & \boxed{1!\text{ の値を保持する変数}}\times2 \\
\text{(step 3)} & \boxed{3!\text{ の値を保持する変数}} & \longleftarrow & \boxed{2!\text{ の値を保持する変数}}\times3 \\
\text{(step 4)} & \boxed{4!\text{ の値を保持する変数}} & \longleftarrow & \boxed{3!\text{ の値を保持する変数}}\times4 \\
\text{(step 5)} & \boxed{5!\text{ の値を保持する変数}} & \longleftarrow & \boxed{4!\text{ の値を保持する変数}}\times5 \\
 & & \cdots\cdots\cdots\cdots\cdots\cdots & \\
\text{(step } k\text{)} & \boxed{k!\text{ の値を保持する変数}} & \longleftarrow & \boxed{(k-1)!\text{ の値を保持する変数}}\times k
\end{array}
$$

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)

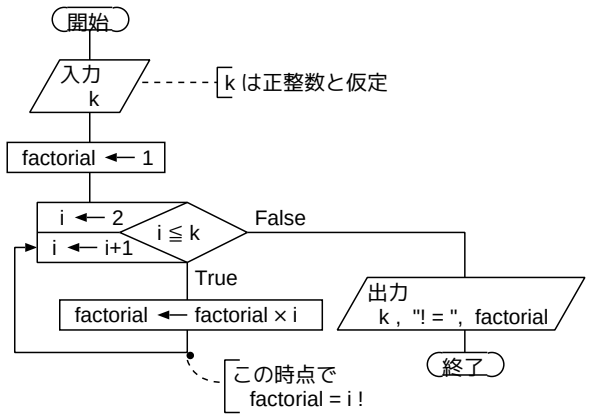
.....

(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 `double` 型変数を、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。

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-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 8 行目 の printf は、データ入力のためのプロンプトを出力している。
- プログラム 11~12 行目 は **for 文**と呼ばれる、繰り返しのための構文である。11 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 `i` 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 `i` 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、

  > `i=2` という設定を行った後で 次の文 (12 行目) を `1<=k` である間 繰り返す、
  > 但し、次の文 (12 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。この場合、変数 `i` 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 11 行目 に現われる `++i` は `i=i+1` と同等の式である。これと類似の `i++`, `--i`, `i--` 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**という。
- プログラム 12 行目 に現われる `factorial *= (double) i` は

  ```
  factorial = factorial * (double) i
  ```

  と同等の式である。これと同様の `+=` , `-=` , `/=` , `%=` , ... という演算子も C 言語ではよく使われる。これらも `=` と同様に**代入演算子**と呼ばれる。
- 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。

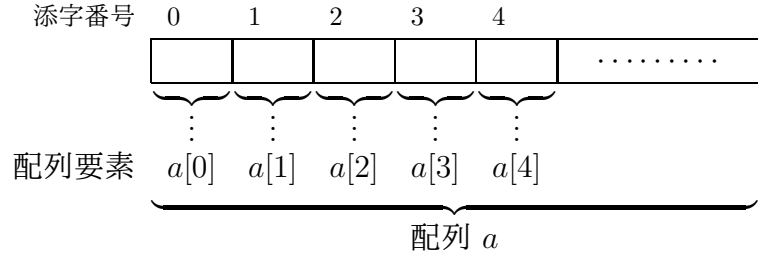
- プログラムの 14 行目 の出力書式中の `%21.16g` は、(半角)21 文字分の出力場所を確保し、そこに (最大) 有効桁 16 桁の精度で (出来れば指数部なしの形で) 出力することを表す。

  > **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 1.3 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。

1.3 一次元配列

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 1.4 (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、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 fundamentals-ave-var.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 fundamentals-ave-var.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 fundamentals-ave-var.c Enter
[motoki@x205a]$ ./a.out < fundamentals-ave-var.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 23 行目 の出力書式中の `%14.5e` は、float 型または double 型データを次の形式の指数部付き 10 進表示で出力することを表す。

$$\underbrace{\sqcup\sqcup s^{-}d^{+}.\overbrace{dddd}^{5\text{桁}}\mathtt{e}sdd}_{14\text{桁}}$$

  ここで、s^- は ␣ (空白) または `-` を、d^+ は `0` 以外の数字を、d は数字を、s は `+` または `-` を表す。
- 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。

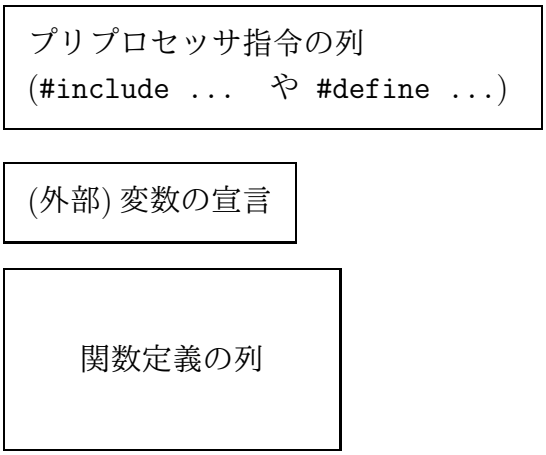
> 補足：
> 実際の出力を見ると `2.0825e-06` と有効桁が 5 桁になっているが、これは g 変換では最後に続く `0` および小数点は印字されない
> ためである。

- プログラム実行のために `./a.out < fundamentals-ave-var.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 `fundamentals-ave-var.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。

1.4 付録 基本文法のまとめ —C 文法のまとめ (1)—

1.4.1 C プログラムの構成

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。



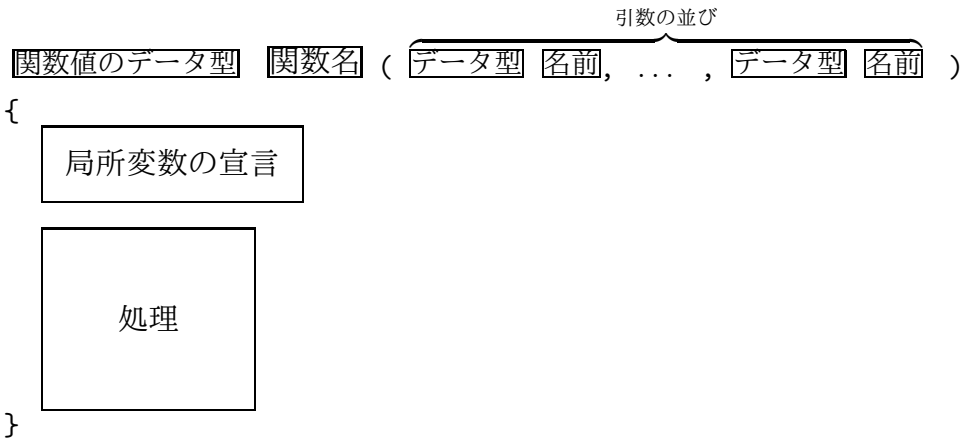
ここで、

- #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。

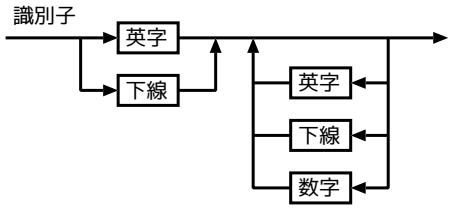
- 名前を表す文字列の途中を除いて、どこで改行してもよいし、どこに空白を挿入してもよい。
  $\Longrightarrow$ 普通は、1 行に 2 つ以上の文を書くのを避け、各行を**字下げ**する (i.e. 書き始めの位置を右にずらす) ことによって、プログラムが見易くなるように工夫する。

1.4.2 宣言、式、代入

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。

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 $\rightarrow$ 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。

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

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。

1.4.3 コンパイラの作業

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。

($\Longrightarrow$ 2.2 節を参照)

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。実際、cc や gcc 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

プリプロセッサ (前処理を行う部分)：

- C コンパイラの翻訳作業の前にヘッダファイルの読み込み等を行う。
- C プログラム中の # で始まる行がプリプロセッサへの指令。
[普通、1 カラム目に # を置く。]

例：
```
#include <stdio.h>                    ···{/usr/include/stdio.h}
#include "ファイル名"          ··· { ファイル名 は普通 .h で終わる。}
#define PI 3.14159           ···{ マクロ名には普通英大文字を使う。}
```

- 標準のヘッダファイル<stdio.h>, <stdlib.h> , ... の中には**関数プロトタイプ** (i.e. 関数がどういう型の引数をもらい、どういう型の値を返すか) の宣言等が入っている。

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。

標準ライブラリ：

- C 言語では、入出力は関数の呼出しによって行うので言語自体は軽くなっている。

(printf も scanf もライブラリ関数)

- ライブラリ関数は豊富に用意されている。(⟹ この講義ノート 4.7 節を参照)
- C 言語のコンパイラ本体は、各ライブラリ関数の**プロトタイプ**、すなわち関数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予め知っている訳ではない。
  ⟹ **必要なプロトタイプ宣言はプログラマが責任を持って行う。**

(#include 等を使う。)

1.4.4 字句要素、演算子

{ ケリー&ポール第 2 章 }

字句の認識： 実際のコンパイル作業はプログラムを構成する字句を認識することから始まる。例えば、プログラム

```
1  /* 2つの整数を読み込み、和を出力 */

2  #include <stdio.h>

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

6    printf("Input two integers:  ");
7    sacnf("%d%d", &a, &b);
8    sum = a + b;
9    printf("%d + %d = %d\n", a, b, sum);

10   return 0;
11 }
```

の場合、コンパイラは次の表に示される様な字句を認識する。[記号のまとめ方については、コンパイラに依存する可能性もある。]

	キーワード	識別子	定数	文字列定数	演算子	句切り記号
1 行目						注釈部
2 行目	プリプロセッサ指令					
3 行目		main			()	
4 行目						{
5 行目	int	a b sum				, ;
6 行目		printf		"Input two integers: "	()	;
7 行目		scanf a b		"%d%d"	() &	, ;
8 行目		sum a b			= +	;
9 行目		printf a b sum		"%d + %d = %d\n"	()	, ;
10 行目						}

注釈：

- /* と */ で囲まれた部分は注釈として扱われる。
- 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

キーワード：

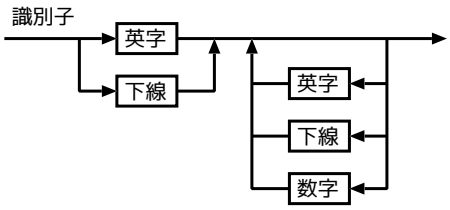
- 次のようなキーワードがある。

```
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
```

識別子：

- 変数, 配列, 関数, ... などに一意的な名前を付けるのに使われる。



- 意味のある名前を選ぶ。
- ANSI C 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。

定数：

整数定数	···	17 (10 進),017 (8 進), 0x17 (16 進), ...
		(注: -17 は定数式)
浮動小数点定数	···	123.4, 123e+5, ...
文字定数	···	'a', 'b', '\n' (改行文字), ' ', ...
列挙定数	···	$\Longrightarrow$ 講義ノート 5.5 節を参照。

- 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。
```

演算子：

- 次のような種類の演算子がある。

算術演算子	···	+ (単項, 恒等変換), - (単項, 符号反転),
		+ (加算), - (減算), * (乗算), / (除算),
		% (剰余; "a % b" は a を b で割った時の余り)
代入演算子	···	=, +=, -=, *=, /=
増分演算子	···	++変数等, 変数等++
減分演算子	···	--変数等, 変数等--
関係演算子	···	<, <=, >, >=, ==, !=
論理演算子	···	&&, ||, !
条件演算子	···	式 ? 式 : 式
間接演算子	···	* ポインタ
番地演算子	···	& 変数等

sizeof 演算子 ··· sizeof(オブジェクト)
キャスト演算子 ··· (データ型) 式
関数の引数をくくる丸括弧 ···
.....

- 例えば、次のプログラムでは下線部が演算子。

```
/* ３つの入力データの最大値 (その３) */
#include <stdio.h>

main()
{
  int  a, b, c, max;

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

  if (b<=a && c<=a)
    max = a;
  else if (c<=b)
    max = b;
  else
    max = c;

  printf("max = %d\n", max);
  return 0;
}
```

演算子の優先順位と結合性:

{⟹ 完全な表は講義ノート 11.4 節 }

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
= += -= *= /=	右から左

例 1.6 (優先順位) 1+2*3 は 1+(2*3) の意。
-a*b-c は ((-a)*b)-c の意。
((-(a*b))-c ではない。)

例 1.7 (結合性) a=b=c は a=(b=c) の意。

句切り記号:

- 丸括弧、波括弧、コンマ、セミコロン、など。
- 例えば、次のプログラムでは下線部が句切り記号。

```
/* ３つの入力データの最大値（その３） */
#include <stdio.h>

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

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

  if (b<=a && c<=a)
    max = a;
  else if (c<=b)
    max = b;
  else
    max = c;

  printf("max = %d\n", max);
  return 0;
}
```

1.4.5 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。
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 1.8 (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次に示す。

```
[motoki@x205a]$ nl fundamentals-printf-e-f-g-conversion.c
     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 fundamentals-printf-e-f-g-conversion.c
[motoki@x205a]$ ./a.out
-------------------------------------------------------
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]$
```

例 1.9 (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) 例題 1.2 のプログラム (double 型で処理する版) の 11~12 行目は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。

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。

1.4.6 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 $\Sigma = \begin{cases} \text{文字列に現れる文字の集合} & \text{if 文字列が '\^{}' 以外の文字で始まる} \\ \text{文字列に現れない文字の集合} & \text{if 文字列が '\^{}' という文字で始まる} \end{cases}$ 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 1.10 ([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]");
```

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。

1.4.7 配列

配列について：

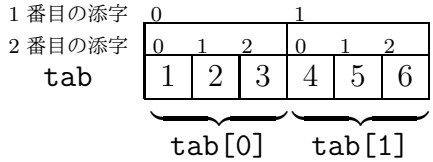
- 配列名が a、大きさが $k_1 \times k_2 \times \cdots \times k_n$ の配列の宣言／領域確保は次の様に行う。

  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。

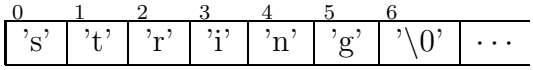
```
int num[12]={31,28,31,30,31,30,31,31,30,31,30,31};
```

```
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。)

演習問題

□演習 1.1 (プログラムの作成・実行) 例題1.1のプログラムを実際に作成･実行してみよ。

□演習 1.2 (四捨五入)

(A) 9 桁以下の 2 つの正整数 m, n をこの順に入力して、

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

を出力するプログラムを作成し、次の入力に対して各々どういう結果が得られるか調べよ。

入力① `8␣3`
入力② `987654321␣2` ←—(結果が正しい事を確認！)
入力③ `2345678901␣2` ←—(大きすぎる数を入力すると...)
入力④ `8␣0` ←—(0 で割ると...)
入力⑤ `8␣3.9` ←—(入力の仕方が間違っていると...)
入力⑥ `8e+1␣3` ←—(入力の仕方が間違っていると...)

(B) 2 つの実数 x, y をこの順に float 型の変数に入力して、

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

を出力するプログラムを作成し、次の入力に対して各々どういう結果が得られるか調べよ。

入力① `8.0␣3.0`
入力② `987654321.0␣2.0` ←—(結果は数学的に正しいか？)
入力③ `1e+100␣1e+100` ←—(大きすぎる数を入力すると...)

□**演習 1.3 (データ型)** 次の C コードを実行するとどういう出力が得られるか？

```
printf("(1) (int)(1+1/2*2.0): %d\n", (int)(1+1/2*2.0));
printf("(2) (int)(1+1.0/2*2): %d\n", (int)(1+1.0/2*2));
```

□**演習 1.4 (++演算子)** 次の C コードを実行するとどういう出力が得られるか？

```
int a;
a = 0;
printf("(3)%d\n", a++);
a = 0;
printf("(4)%d\n", ++a);
```

□**演習 1.5 (間違い探し)** 2 次元ベクトル (x, y) のノルム $\sqrt{x^2 + y^2}$ を計算して出力する C プログラムを作ろうとしたが、次の様に実行時エラーになってしまった。何が起こったのか説明し、プログラムの誤りを修正せよ。

```
xcsws02_49% nl test0107_2a.c
     1  #include <stdio.h>
     2  #include <math.h>

     3  int main(void)
     4  {
     5    double x, y;

     6    scanf("%lf %lf", x, y);
     7    printf("the norm of the vector (%g, %g) = %g\n",
     8           x, y, sqrt(x*x+y*y));
     9    return 0;
    10  }
xcsws02_50% cc test0107_2a.c -lm
xcsws02_51% a.out
3.0 4.0
Segmentation fault
xcsws02_52%
```

□**演習 1.6 (代入式)** 2 つの代入式

```
a=b+(b=c+(c=c+1))
```

と

```
a=(b=(c=c+1)+c)+b
```

は同じ計算結果をもたらすか？

□**演習 1.7 (ソースプログラムの文字コード)** 例題 1.2 のプログラムを JIS コード, EUC コードで作成して、それぞれでコンパイルエラーが出るかどうか確かめよ。 [コード変換するには、`nkf` コマンド, `jistoeuc` コマンド等があります。]

□**演習 1.8 (冪乗)** 実数 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 1.9 (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。

□演習 1.10 (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。

□演習 1.11 (シンプソンの公式) 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。$n = 1000$ としてこの公式を用いることによって $\displaystyle\int_0^1 \frac{4}{1+x^2}dx$ の近似値を計算する C プログラムを作成せよ。

□演習 1.12 (配列の有用性) 例題 1.4 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 1.13 (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

$$\begin{aligned} \mu &= \frac{1}{n}\sum_{i=0}^{n-1}x_i \\ V &= \frac{1}{n}\sum_{i=0}^{n-1}x_i^2 - \mu^2 \end{aligned}$$

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計算するプログラムを作ると例題1.4 で挙げたものより単純なものが得られる。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 1.14 (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 1.15 (整列化)** 50 個の整数データを入力して、それらを小さい順に出力する C プログラムを作成せよ。

□**演習 1.16 (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。

□**演習 1.17 (ヘッダファイルの中身)** 標準に用意されたヘッダファイルの中を覗いてみて下さい。 例えば、`/usr/include/stdio.h` の中で `printf` という関数がどの様に記述されているか調べよ。 [`cat /usr/include/stdio.h |grep printf` とすると簡単。]

□**演習 1.18 (printf 理解度チェック)** `printf` 関数で出力する際、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 1.19 (printf; 可変なフィールド幅, 精度)** `printf` 関数で出力する際、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);
```

□**演習 1.20 (scanf 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?

2 実習案内 C プログラミング環境

- 実習の進め方,
- 復習 Emacs を使って C ソースプログラムを作る際の注意, cc コマンド, a.out, indent コマンド, nl コマンド,
- プログラミング時の注意,
- レポート提出の形式,
- 復習 C コンパイラについて,

2.1 実習の進め方

- 基本的には、実習は授業日の 4 限に行う。
- 3 限講義に合わせた内容で、2 週間に 1 つの割合でプログラム作成のレポート課題が課される。

  レポート提出の形式 ⟶ 2.4 節
  レポートの提出先 ⟶ 直接担当者へ

- 授業日の 4 限だけでは要求されたプログラムとそれに関連したレポートが完成しないことが予想されるので、**他の時間にも自分で十分に実習**して下さい。
- Web 上 (`http://www.ce.ie.niigata-u.ac.jp/~motoki/ProgrammingAI.html`) にそれぞれの実習日に関連した連絡事項を配置していますので、実習開始直後に**その日の連絡事項**を読んで下さい。

  注意：この**連絡事項**の中には、
  ◇ 授業に関する連絡 (e.g. 補講, 教室の変更,...) や
  ◇ レポートプログラム作成上のヒント、注意事項、
  ◇ コンピュータ利用、LaTeX に関する注意事項、
  ◇ レポート再提出の指示、
  等が含まれるので、**必ず読んで**下さい。

2.2 復習 C プログラムの作成と実行

基礎知識

- C プログラムのソースファイルの拡張子は .c です。
- C コンパイラを起動するコマンドは cc または gcc です。(使い方は man コマンドで調べることが出来ます。)
- 何も指定しないと、コンパイル結果は a.out というファイルに生成されます。

C プログラム作成・実行の手順　例えば、次の様にします。

(1) C プログラミング用のディレクトリを用意し、そこに移る。

(2) 仮想端末 (例えば gnome-terminal) ウィンドウ上で emacs ファイル名 & とコマンド入力する。

(但し、C プログラムのファイル名は .c で終わる様にする。)

(3) Emacs ウィンドウと仮想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。 (できればウィンドウが重ならない様にする。)

(4) 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。
(Emacs エディタはまだ終了しない。)

(5) 仮想端末ウィンドウをアクティブ状態にする。

(6) gcc ソースファイル名 または cc ソースファイル名 とコマンド入力してプログラムをコンパイルする。
(a.out に実行ファイルが出来る。 cc コマンド／C コンパイラについては 2.5 節を参照。)

(7) 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。

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。

(8) コンパイルが無事 (i.e. エラー無しで) 終了したら、a.out とコマンド入力してプログラムを実行する。
(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。)

(9) 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。

① 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。[誤り箇所が分からない場合は、**デバッガ** (第 6 節) を用いてプログラムの実行追跡を行ったりする。]

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) 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。

例 2.1 (x/y の小数点以下切り上げ) プログラム表示／コンパイル／実行の様子を次に示す。(下線部はキーボードからの入力を表す。)

```
motoki@ap1:~/C-java$ cat lab-ex01-ceiling.c  Enter
#include <stdio.h>

int main(void)
{
  int x, y, sum, ceiling;

  scanf("%d %d", &x, &y);
  sum = x+y;
  ceiling = x/y + (x%y + 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);
```

```
  return 0;
}
motoki@ap1:~/C-java$ gcc lab-ex01-ceiling.c Enter
motoki@ap1:~/C-java$ ./a.out Enter
8 3 Enter
8+3=11
ceiling(8/3)=3
motoki@ap1:~/C-java$
```

プログラム作成の際に有用なキー／コマンドを次に紹介しておきます。

Emacs の下での C プログラム作成 Emacs の下で C プログラム (i.e. ＿＿.c という名前のファイル) を編集する時は、次のようなキーを使うと**字下げ**が楽に出来ます。

- Ctrl-j ··· 改行してから次の行の字下げを行う。(newline-and-indent)
- TAB ··· カーソルのある行の字下げ (調節) を行う。(c-indent-line)
- Esc Ctrl-\ ··· 指定されたリージョンの字下げ (調節) を行う。リージョン指定は予め行っておく。(indent-region)

C プログラムの整形 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 C プログラムを整形・表示してくれる次の UNIX コマンドが役立つことがありますが、**実習室の計算機にはインストールされていない**様です。

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

C プログラムに行番号を付ける gdb デバッガを使う際等に、プログラム内で途中結果を観察する場所の行番号が必要になることがあります。こんな時には、文書ファイルに行番号を付けて表示する次の UNIX コマンドが役に立ちます。(gdb デバッガの参考のために使う場合は -b a というオプションを付けて実行する。cat コマンドを -n オプション付きで実行してもよい。)

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を含む行だけに番号付け。
	その他	オンラインマニュアルを御覧下さい。(man nl)

2.3 プログラミング時の注意

- 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りにくくなります。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。

```
/* 科　目　名(○曜○限)      */
/* ○○学部○○○プログラム   */
/* 各自の学籍番号 各自の氏名 */
```

- 各自のホームディレクトリの下にディレクトリを作り、その中で C プログラム作成等の作業を行って下さい。 その際、そのディレクトリの保護モードを `rwx---r--` に設定して下さい。 例えば、

```
motoki@ap1:~$ mkdir c-lab Enter
motoki@ap1:~$ chmod g-rwx c-lab Enter
```

- 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。]

2.4 レポート提出の形式

レポートとして提出するもの： 各課題について次の 3 つのものを提出する。

(1) **プログラム，実行の様子を含む文書を印刷したもの：** まず、プログラムを (`cat` または `nl` コマンドで) 表示, コンパイル, 実行している会話の様子をテキストファイルとして記録したもの (**ログファイル**と言う) を作る。そして、**基本的には**、これを基に ①科目名と課題番号, ②提出者の在籍番号と氏名, ③ログファイルの内容, ④プログラムの説明等 (例えばアルゴリズム設計の考え方, 変数の使い方 など; C 文法の説明は不要), ⑤考察・その他 を順に並べた文書を **(できれば LaTeX で)** 構成して印刷したものを提出する。科目名, 課題番号, 在籍番号, 氏名だけを並べた表紙は不要です。

> 仮想端末上の会話の様子をファイルとして記録するには：
> **(方法 1)** gnome-terminal 上の会話の様子をコピー・アンド・ペーストで Emacs に取り込む。 簡単だが会話が長い場合は間違う可能性がある。
> **(方法 2)** emacs の中で shell を開き、その会話の様子をファイルに取り込む。
> (emacs の中で shell を開くには、Esc x shell とする。)
> **(方法 3)** `script` コマンドを用いる。 (⟶ 下で説明します。)

(2) **プログラムの (ソース) ファイル：** 学務情報システムのレポート機能を通じてソースファイル等を提出する。すなわち、

(2.1) レポートを受け取る側での整理のために、アップロードするファイルの名前に各々の在籍番号や課題番号を含ませておく。具体的に、
課題 1, 2, 3 の場合は、それぞれ
各自の在籍番号，小文字`rep1.c` , 各自の在籍番号，小文字`rep2.c` ,
各自の在籍番号，小文字`rep3.c`

という名前のソースファイルを作る。
課題 4 の場合は、関連するファイル群を 各自の在籍番号, 小文字_Rep4 というディレクトリに入れ、このディレクトリ以下を tar コマンドで 1 つにまとめ gzip コマンドで圧縮して 各自の在籍番号, 小文字_Rep4.tar.gz という名前のファイルを作る。

(2.2) 統合型学務情報システムにログインする。

(2.3) ログイン画面上部のメニューバー内の「レポート・小テスト・アンケート」ボタン、サブメニュー内の「レポート・小テスト・アンケート提出」ボタンを順に押す。

(2.4) 現れた一覧表の中から、科目名が「プログラミング AI」でタイトル欄が当該課題番号 (e.g.「課題 1」) の行を選び、右側の提出ボタンを押す。

(2.5) 現れたレポート提出画面上で、「ファイル添付」欄下の「参照」ボタンを押し (2.1) の該当ファイルを指定する。

(3) **印刷物の電子ファイル (pdf ファイル)**: 学務情報システムのレポート機能を通じて、(1) の印刷物の pdf ファイルを (2) のソースファイルと一緒に提出 (添付) する。但し、ここでもレポートを受け取る側での整理のために、課題 1~4 で添付する pdf ファイルの名前は、それぞれ 各自の在籍番号, 小文字_rep1.pdf ~ 各自の在籍番号, 小文字_rep4.pdf としておく。

script コマンドについて: プログラミング実習のレポートなどを作成する場合、単に出力結果を入出力リダイレクションでファイルに入れるだけでなく、画面に表示される会話の様子をそのまま**ログ** (log, 日誌) **ファイル**に記録できると便利です。 こういった場合のために、UNIX には script コマンドが用意されています。 script コマンドは、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 (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。 記録を終えた後でこのログファイルを編集して通常のテキストファイルにすれば、会話の記録をそのまま出力することもできます。 script コマンドの使い方は次の通りです。

(1) 仮想端末ウィンドウで script ログファイル名 と入力して記録を開始する。

(2) 必要な会話をする。

(3) exit と入力して記録を終了する。

(4) エディタを使ってログファイルを編集する。

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。

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。

同様に、コマンド行左端に含まれる "^[]0" ～ "^G" の部分のコードも全て削除します。さらに、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、これも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。**

例 2.2 (script コマンド；x/y の小数点以下切り上げ) 例 2.1 のプログラムがレポート課題であった場合は、実習室で例えば図 1 の様にします。 すると、図 2 の様なログファイルが出来ますから、これから **各コマンド行左端の "^[]0"～"^G" の部分と各行最後の "^M" を削除**することになります。

2.5 復習 C コンパイラについて

- 実習室の計算機では C 言語のコンパイラが 1 種類だけ用意されています。コンパイラを起動するためのコマンドとして gcc と cc の 2 つが用意されていますが、これらの実体は同じです。(他に c99-gcc 等もありますが、これらは特殊なオプション付きの gcc の様です。)

```
motoki@ap1:~$ which gcc
/usr/bin/gcc
motoki@ap1:~$ ls -l /usr/bin |grep cc
        (省略)
-rwxr-xr-x 1 root   root         428  6月 13  2013 c89-gcc
-rwxr-xr-x 1 root   root         454  6月 13  2013 c99-gcc
lrwxrwxrwx 1 root   root          20  2月 23  2015 cc -> /etc/alternatives/cc
        (省略)
```

```
motoki@ap1:~/C-java$ script report.log Enter
                          ......... ログファイル report.log への記録を開始
Script started, file is report.log
motoki@ap1:~/C-java$ cat lab-ex01-ceiling.c Enter
#include <stdio.h>

int main(void)
{
  int x, y, sum, ceiling;

  scanf("%d %d", &x, &y);
  sum = x + y;
  ceiling = x/y + (x%y + 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);
  reurn 0;
}
motoki@ap1:~/C-java$ gcc lab-ex01-ceiling.c Enter
motoki@ap1:~/C-java$ ./a.out Enter
8 3 Enter
8+3=11
ceiling(8/3)=3
motoki@ap1:~/C-java$ ./a.out Enter
999 3 Enter
999+3=1002
ceiling(999/3)=333
motoki@ap1:~/C-java$ exit Enter ..............ログファイルへの記録を終了
exit
Script done, file is report.log
motoki@ap1:~/C-java$
```

図 1: script コマンドを用いて会話の様子を report.log に記録

```
Script started on 2016年03月07日 17時11分33秒
^[]0;motoki@ap1: ~/C-java^Gmotoki@ap1:~/C-java$ cat lab-ex01-ceiling.c^M
#include <stdio.h>^M
^M
int main(void)^M
{^M
  int x, y, sum, ceiling;^M
^M
  scanf("%d %d", &x, &y);^M
  sum = x+y;^M
  ceiling = x/y + (x%y + y - 1)/y;     /* x/y の小数点以下切り上げ */^M
  printf("%d+%d=%d\n", x, y, sum);^M
  printf("ceiling(%d/%d)=%d\n", x, y, ceiling);^M
  return 0;^M
}^M
^[]0;motoki@ap1: ~/C-java^Gmotoki@ap1:~/C-java$ gcc lab-ex01-ceiling.c^M
^[]0;motoki@ap1: ~/C-java^Gmotoki@ap1:~/C-java$ ./a.out^M
8 3^M
8+3=11^M
ceiling(8/3)=3^M
^[]0;motoki@ap1: ~/C-java^Gmotoki@ap1:~/C-java$ ./a.out^M
999 3^M
999+3=1002^M
ceiling(999/3)=333^M
^[]0;motoki@ap1: ~/C-java^Gmotoki@ap1:~/C-java$ exit^M
exit^M

Script done on 2016年03月07日 17時13分03秒
```

図 2: ログファイルの中身 (生成直後,emacs で表示)

```
lrwxrwxrwx 1 root   root          7  2月 25  2015 gcc -> gcc-4.9
-rwxr-xr-x 1 root   root     353752 11月 28  2012 gcc-4.6
-rwxr-xr-x 1 root   root     832120 12月 26  2014 gcc-4.9
       (省略)
motoki@ap1:~$ ls -l /etc/alternatives/cc
lrwxrwxrwx 1 root root 12  2月 23  2015 /etc/alternatives/cc -> /usr/bin/gcc
motoki@ap1:~$
```

- 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。]

C のソースファイル □.c

⇓

①前処理プログラム /usr/local/lib/gcc-lib/···/cpp ..... [ヘッダファイルの取り込み, マクロの展開, 注釈の除去などを行う。

⇓

#で始まる行 (e.g. #include 行, #define 行) や
注釈を含まない C のソースプログラム□.i

⇓

②コンパイラの本体 /usr/local/lib/gcc-lib/···/ccl ..... [構文解析, 意味解析等はここで行う。

⇓

アセンブリプログラム □.s ..... [機械語プログラムの命令語、アドレス等を記号表記したもの。

⇓

③最適化プログラム (/usr/···??) ..... [-O オプションが指定された時は、実行速度, コードの大きさの改善を図る。

⇓

アセンブリプログラム □.s ..... [-S オプションの指定があればこのアセンブリプログラムも保存される。

⇓

④アセンブラ /usr/ccs/bin/as

⇓

オブジェクトプログラム
(のファイル) □.o ..... [2 進コードと再配置情報が入っている。-c オプションの指定があったりソースファイルが 2 個以上あったりするとファイルが残る。

⇓

⑤リンカ & ローダ /usr/ccs/bin/ld ..... [複数のオブジェクトファイルをまとめる。その際、必要なライブラリ関数のコードも取り込む。

⇓

実行形式プログラムのファイル ..... [-o オプションの指定がなければ a.out

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 gcc コマンドについては次の通り。

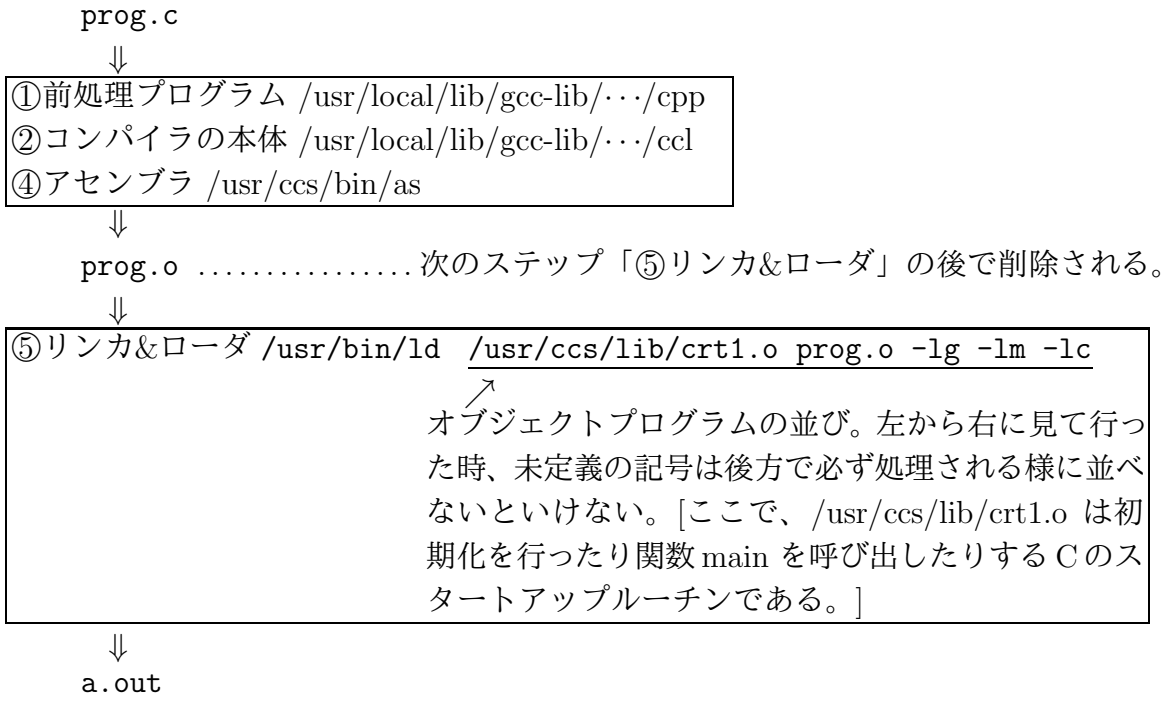
形式:	**gcc** [*options*] *files* [-l *library*]	
options:	-c	: オブジェクトファイルだけを生成し、リンカ&ローダを呼び出さない。
	-g	: デバッガのための特別なシンボル表もコンパイルの際に生成し、リンカ&ローダ ld に -lg オプションを引き渡す。
	-o ファイル名	: 実行形式プログラムを入れるファイルを a.out ではなくファイル名 とする。
	-O	: 最適化を行う。
	-D 名前 = 定義 -U 名前 -I パス名 -P -S -v -p -pg ⋮	(説明省略)
-l *library*:	-lx	: リンカ&ロ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 gcc 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&古瀬一隆 (監), 新 The UNIX SuperText 下 改訂増補版, 技術評論社, 2003.

- 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、% gcc -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。



演習問題

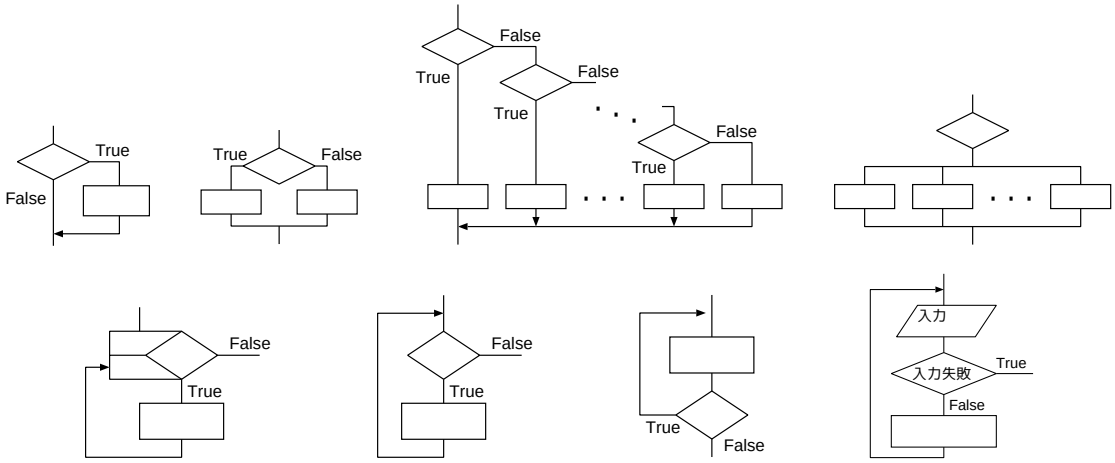
□**演習 2.1 (プログラム作成・修正・保存・実行)** この講義ノートの第1章, 参考書などから練習問題をいくつか選んで、C プログラムのコンパイル／実行／デバッグを行ってみよ。 その際、プログラムを故意に間違えて、どんなエラーメッセージが出るか試してみよ。

□**演習 2.2 (stty コマンド; 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。

3 復習 処理の選択と繰り返し

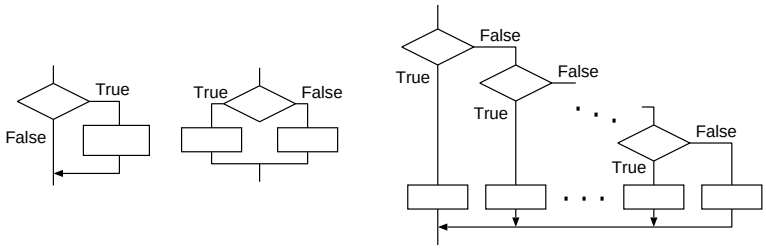
- 3 つの数の最大値 (if 文,if-else 構文, 論理演算, 条件演算子),
- 階乗 (for 文), どうやって繰り返し構造を見出すか?,
- 自習 素数 (break 文), 最大公約数 (ユークリッドの互除法),
- 自習 不定個の入力データの合計 (while 文),
- 自習 元号表記→西暦表記 (switch 文),
- 自習 プログラムを組み立てられない時は ...

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。



3.1 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。

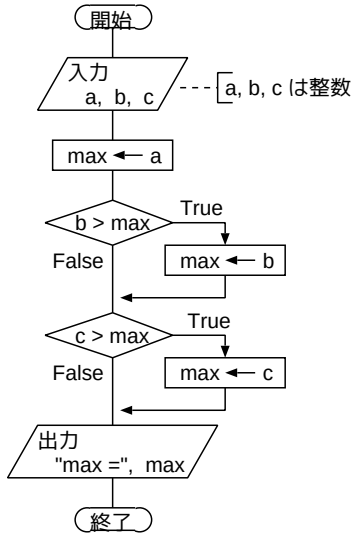


例題 3.1 (3 つの数の最大値; if 文,if-else 構文, 複合文) 整数を 3 つ読み込みその中の最大値を出力するC プログラムを作成せよ。

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 3.1 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]$
```

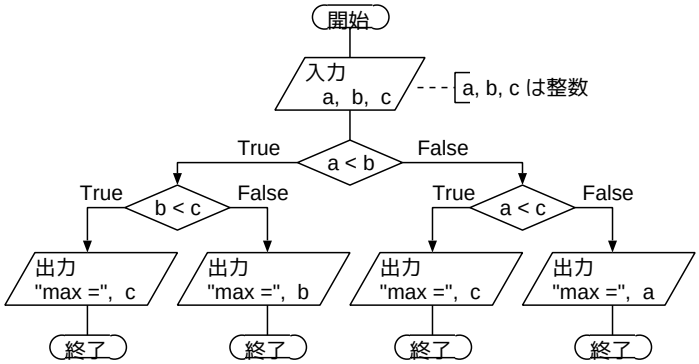
ここで、

- 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。

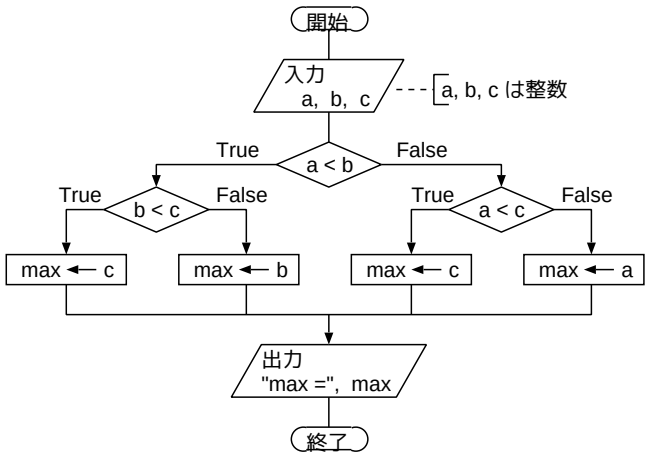
補足：
C 言語においては、max < b といった条件を表す式の評価結果は true, false などの論理値ではなく整数値である。従って、例えば 9~10 行目においては、実際には、条件 max < b が成立すればこの関係式の値は 1 と計算され、成立しなければ 0 と計算される。そして、この関係式の値が 1 の時だけ 10 行目の代入文が実行される。

例題 3.1 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]$
```

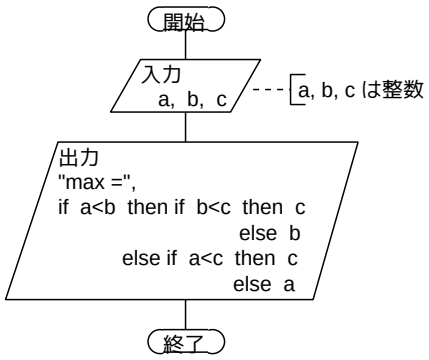
ここで、

- 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", (b<=a && c<=a) ? a : (c<=b ? b : c));
     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 (b<=a && c<=a) ? a : (c<=b ? b : c) 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 (b<=a && c<=a) が成り立つかどうかが調べられ、成り立てば a の値が、そして成り立たなければ式 (c<=b ? b : c) 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 3.1 に対するアルゴリズム (3) :

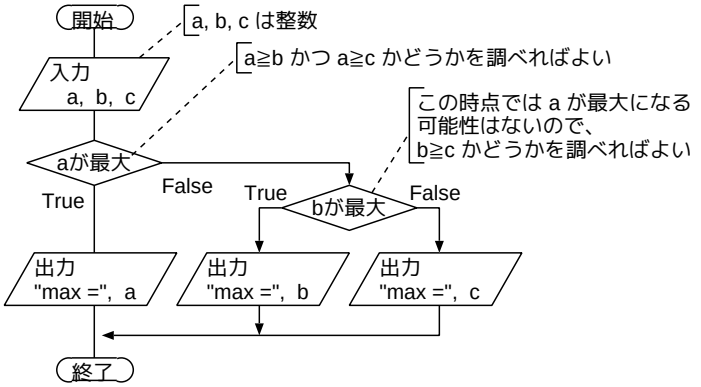
(考え方) 読み込んだ整数 a, b, c 間の大小関係を調べることにより、各々の要素が最大であるかどうかを判定することができる。例えば、

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

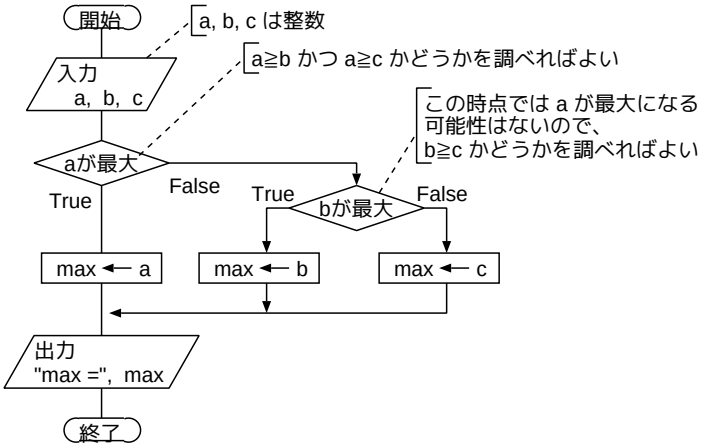
である。 それゆえ、整数 3 つを読み込んだあとで、まず a が最大かどうか で場合分けする。 その結果、もし a が最大 ならその値を出力し、もしそうでないなら 残った b, c の間

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。

C 言語における論理式の扱い (概略)： (⟹ 3.7.1 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。

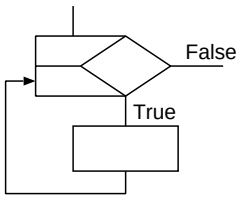
論理式の値が整数として処理されるために、
本来は文法エラーとなるべき記述もチェックされない。例えば、

- 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。

3.2 処理の規則的な繰り返し

この節では、1.2 節に引き続いて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。



右図の形の繰り返しを制御する変数は加法的に変化させる場合が多いが、それ以外の規則的な変化のさせ方も可能である。 次にその例を示す。

補足：
繰り返し制御の変数を加法的に変化させる場合のプログラム例については 1.2 節を御覧下さい。

例題 3.2 (べき乗；for 文, コンマ演算子) 指数部が非負整数のべき乗を計算する際は、等式

$$x^y = \begin{cases} 1 & \text{if} \quad y = 0 \\ (x^2)^{\lfloor y/2 \rfloor} & \text{if} \quad y \text{ が } 2 \text{ 以上の偶数} \\ (x^2)^{\lfloor y/2 \rfloor} \times x & \text{if} \quad y \text{ が奇数} \end{cases}$$

に注目して左辺を右辺のように変形して計算する作業を繰り返せば、乗算の回数が少くて済む。実数データ x と 非負整数データ y を読み込み、この方法でべき乗 x^y を計算して出力する C プログラムを作成せよ。

(考え方) 与えられた等式に基づけば、我々は 2^{27} の計算を次のように進めることができる。

$$
\begin{aligned}
2^{27} &= 2\times 4^{13} && (y\text{ が奇数なら } x^y=(x^2)^{\lfloor y/2\rfloor}\times x \text{ だから})\\
&= 2\times 4\times 16^6 && (y\text{ が奇数なら } x^y=(x^2)^{\lfloor y/2\rfloor}\times x \text{ だから})\\
&= 8\times 16^6\\
&= 8\times 256^3 && (y\text{ が偶数なら } x^y=(x^2)^{\lfloor y/2\rfloor} \text{ だから})\\
&= 8\times 256\times 65536^1 && (y\text{ が奇数なら } x^y=(x^2)^{\lfloor y/2\rfloor}\times x \text{ だから})\\
&= 2048\times 65536^1\\
&= 2048\times 65536\times 4294967296^0 && (y\text{ が奇数なら } x^y=(x^2)^{\lfloor y/2\rfloor}\times x \text{ だから})\\
&= 134217728\times 4294967296^0\\
&= 134217728\times 1 && (y=0\text{ なら } x^y=1 \text{ だから})\\
&= 134217728
\end{aligned}
$$

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

$$1\times 2^{27} \Longrightarrow 2\times 4^{13} \Longrightarrow 8\times 16^6 \Longrightarrow 8\times 256^3 \Longrightarrow 2048\times 65536^1 \Longrightarrow \cdots\cdots$$

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 $\boxed{\cdots}\times\boxed{\cdots}^{\boxed{\cdots}}$ を認識するために3つの$\boxed{\cdots}$の数値を記憶する変数を用意し、各々 $pow, base, exp$ 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。

変数 pow		変数 $base$		変数 exp	
1	$\times$	2		27	
2	$\times$	4		13	
8	$\times$	16		6	
8	$\times$	256		3	
2048	$\times$	65536		1	
134217728	$\times$	4294967296		0	

$\Longrightarrow$ 134217728

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。

変数 pow		変数 $base$		変数 exp		変数 pow		変数 $base$		変数 exp
a	$\times$	b		c	$\Longrightarrow$	if odd(c) then $a\times b$ else a	$\times$	b^2		$\lfloor c/2\rfloor$

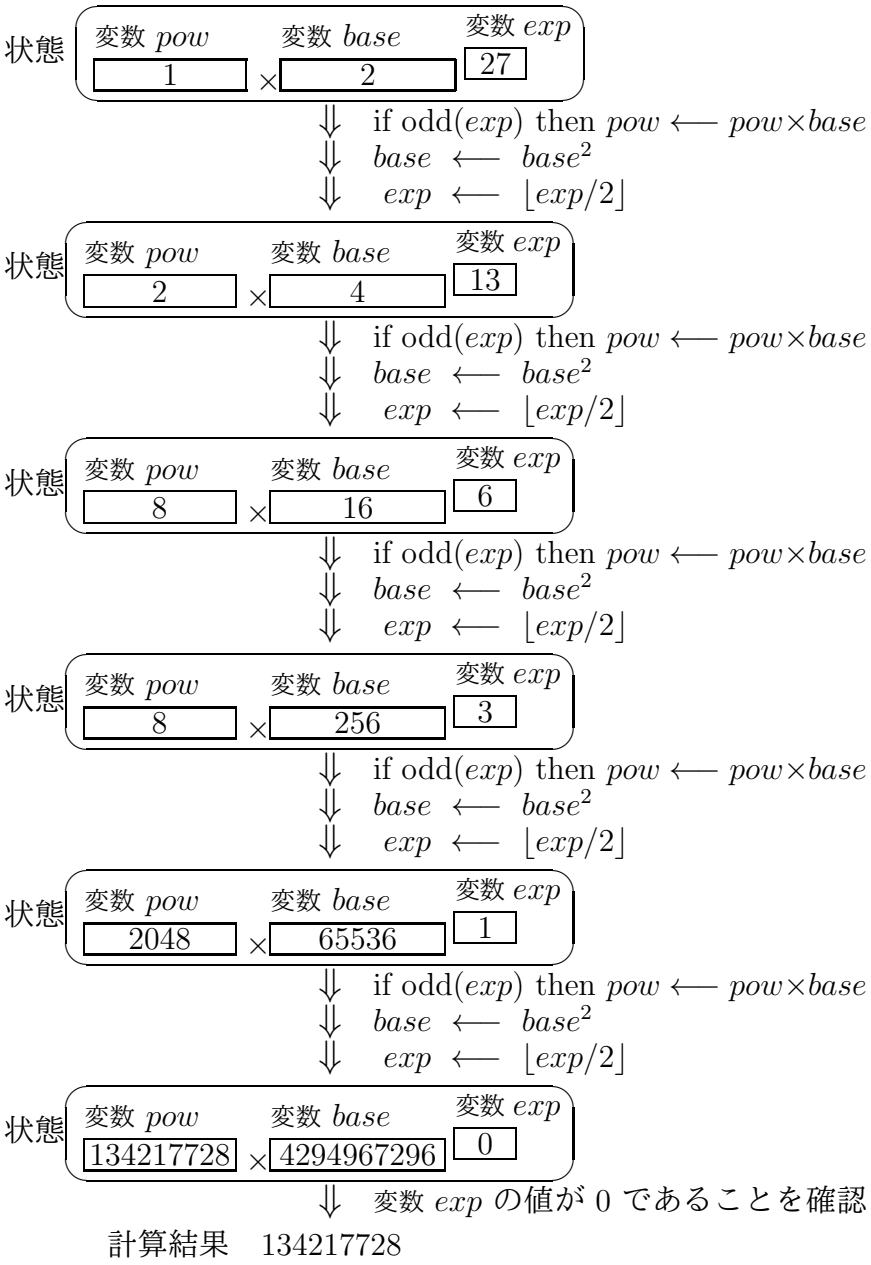
この更新を行うには、

$$
\begin{aligned}
&\text{if odd}(exp)\text{ then } pow \longleftarrow pow\times base\\
&base \longleftarrow base^2\\
&exp \longleftarrow \lfloor exp/2\rfloor
\end{aligned}
$$

とすれば良い。 また、最後の

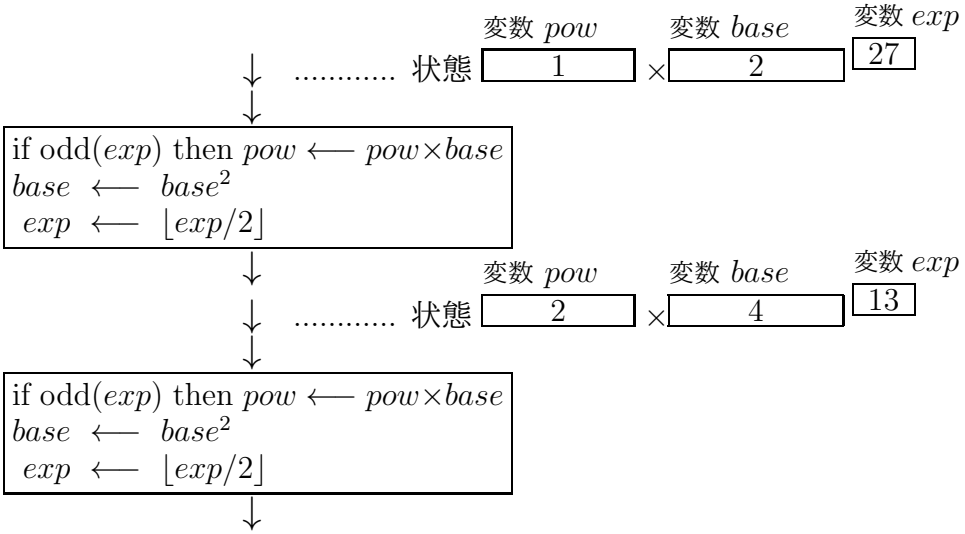
変数 pow		変数 $base$		変数 exp	
134217728	$\times$	4294967296		0	$\Longrightarrow$ 134217728

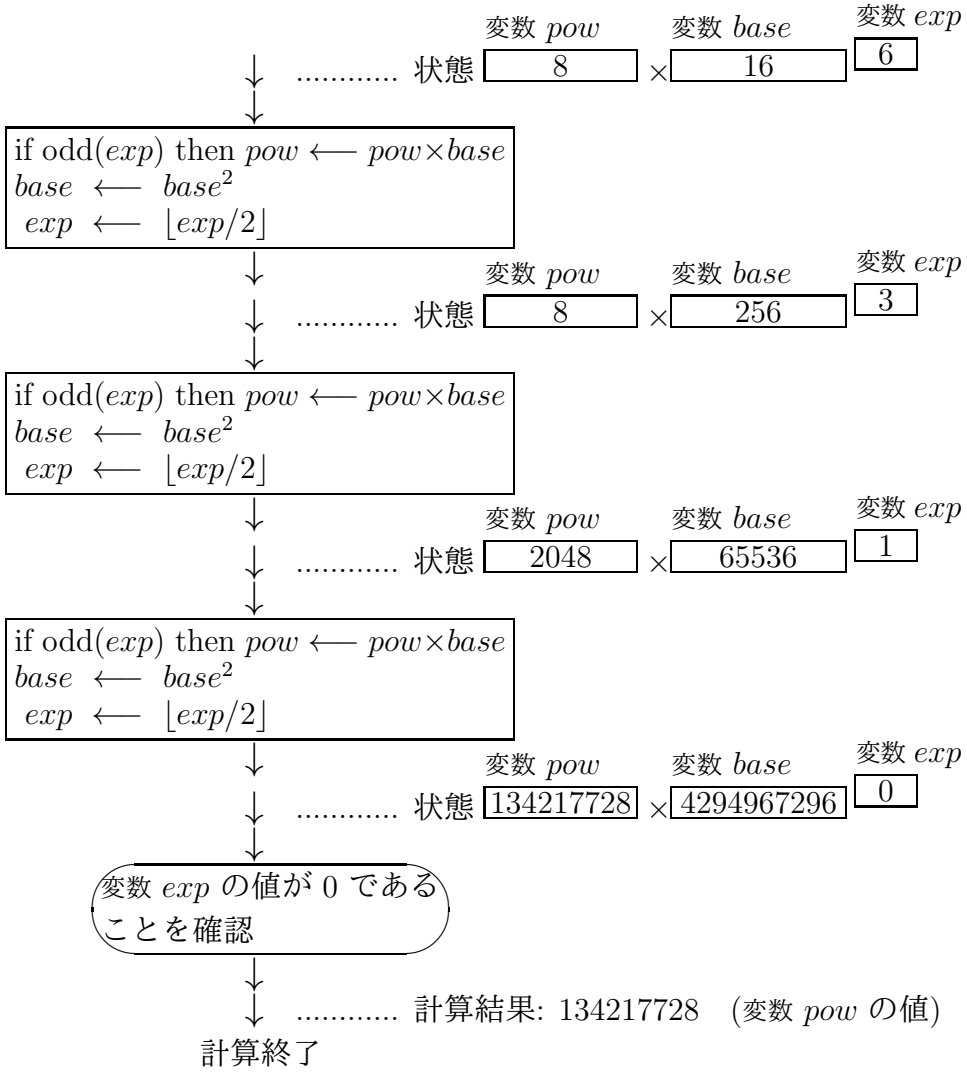
という式変形は、変数 exp の値が 0 なので起こっていると考えられる。 結局、2^{27} の計算の場合に、どういう処理によってどういう風に状態が変わっていくかを具体的に明示すると次の様になる。



計算結果　134217728

(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。





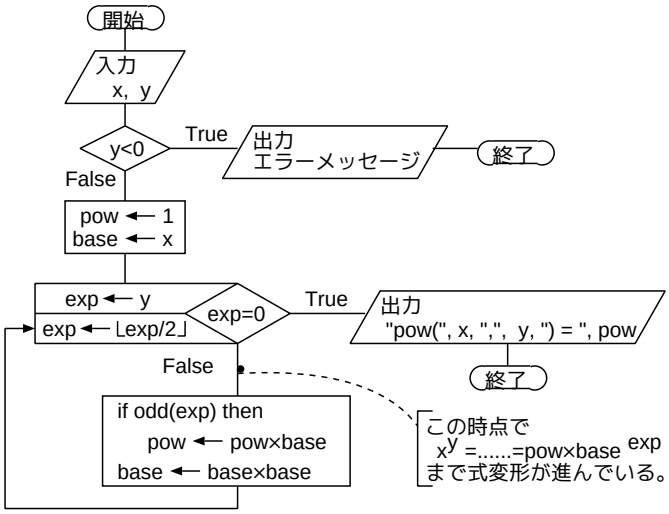
要するに、$exp = 0$ となるまで

$$\left|\begin{array}{l} \text{if odd}(exp) \text{ then } pow \longleftarrow pow \times base \\ base \longleftarrow base^2 \\ exp \longleftarrow \lfloor exp/2 \rfloor \end{array}\right.$$

という固定的な処理を繰り返すだけである。 変数 exp の値が 0 になった時点では、計算結果は変数 pow に保持されている。 一般に、べき乗 x^y を計算する場合、変数 exp の値は

$$y \longrightarrow \lfloor y/2 \rfloor \longrightarrow \lfloor y/2^2 \rfloor \longrightarrow \lfloor y/2^3 \rfloor \longrightarrow \cdots \longrightarrow 1 \longrightarrow 0$$

と確定的に変わり、この値が 0 になれば繰り返しを終了するので、この変数 exp は繰り返しを制御する変数として働く。それゆえ、読み込んだ実数データ, 非負整数データを格納するために各々 `x`，`y` という名前の変数を、式変形の際の $\boxed{\cdots} \times \boxed{\cdots}^{\boxed{\cdots}}$ の 1 番目,2 番目,3 番目の $\boxed{\cdots}$ の数値を保持するためにするために各々 `pow`，`base`，`exp` 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 power-function.c  Enter
     1 /* 実数 x と非負整数 y を読み込み、          */
     2 /* べき乗値 x^y を計算・出力する C プログラム */

     3 #include <stdio.h>
     4 #include <stdlib.h>

     5 int main(void)
     6 {
     7   double x, pow, base;
     8   int    y, exp;

     9   printf("x の y 乗を計算をします。\n"
    10          "実数 x と非負整数 y をこの順に入力して下さい：  ");
    11   scanf("%lf %d", &x, &y);
    12   if (y < 0) {
    13     printf("Input Error!\n");
    14     exit(EXIT_FAILURE);
    15   }

    16   pow  = 1.0;
    17   base = x;

    18   for (exp=y; exp>0; exp/=2) {
    19                          /* この時点で x^y=...=pow*base^exp */
    20     if (exp%2 == 1)      /* まで式変形が進んでいる。        */
    21       pow *= base;
    22     base *= base;
```

```
    23   }

    24   printf("pow(%g, %d) = %.16g\n", x, y, pow);
    25   return 0;
    26 }
[motoki@x205a]$ gcc power-function.c Enter
[motoki@x205a]$ ./a.out Enter
x の y 乗を計算をします。
実数 x と非負整数 y をこの順に入力して下さい： 2.0 27 Enter
pow(2, 27) = 134217728
[motoki@x205a]$ ./a.out Enter
x の y 乗を計算をします。
実数 x と非負整数 y をこの順に入力して下さい： 8.5 50 Enter
pow(8.5, 50) = 2.957646637126993e+46
[motoki@x205a]$ ./a.out Enter
x の y 乗を計算をします。
実数 x と非負整数 y をこの順に入力して下さい： 8.5 -5 Enter
Input Error!
[motoki@x205a]$
```

ここで、

- プログラム 4 行目 は、/usr/include/stdlib.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 14 行目 の exit(EXIT_FAILURE) は、終了状態を EXIT_FAILURE (通常は 1 と定義されたマクロ; 0 以外なので異常終了を表す) としてプログラムを強制終了させることを表す。exit() は scanf や printf と同様にコンピュータ内に予め用意された**標準ライブラリ関数**であり、その引数の型、関数値の型等の情報は /usr/include/stdlib.h に入っている。

> **補足：**
> 正常終了なら exit(EXIT_SUCCESS) と書く。EXIT_SUCCESS と EXIT_FAILURE は、通常 /usr/include/stdlib.h の中で
> `#define EXIT_SUCCESS 0`
> `#define EXIT_FAILURE 1`
> と定義されている。

- 変数 exp の値が $y \longrightarrow \lfloor y/2 \rfloor \longrightarrow \lfloor y/2^2 \rfloor \longrightarrow \lfloor y/2^3 \rfloor \longrightarrow \cdots \longrightarrow 1 \longrightarrow 0$ と変わるのに伴って変数 base の値は $x \longrightarrow x^2 \longrightarrow x^4 \longrightarrow x^8 \longrightarrow x^{16} \longrightarrow \cdots$ と確定的に変わって行く。繰り返し終了の判定に使われることはないが、base も繰り返しを制御するための重要な変数と考えることも出来る。そこで、exp と base を相補的な繰り返し制御の変数と見て、プログラム 17~23 行目 を次の様に書き換えることも出来る。

```
for (base=x, exp=y; exp>0; base*=base, exp/=2) {
                        /* この時点で x^y=...=pow*base^exp */
  if (exp%2 == 1)       /* まで式変形が進んでいる。        */
    pow *= base;
}
```

この中の `for` に続く丸括弧内に現われるコンマ (`,`) は演算子で、コンマで区切られた式を前から順に評価し最後の式の値を全体の式の値とする働きがある。

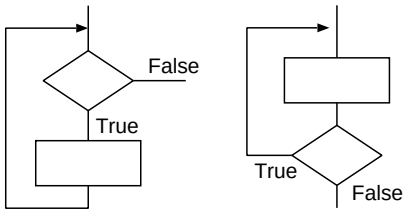
プログラムの注釈について：
どういう変数を用意するか、変数を使って計算の現状態をどう表すか、といったことはアルゴリズム/プログラムを設計する上での重大な決定事項である。 それゆえ、上記プログラム 19~20 行目 の様に

> 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。

3.3 自習 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 3.3 (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 3.4 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。

$$\begin{aligned}
\gcd(1596,\,308) &= \gcd(\mathrm{mod}(308,1596),\,1596) && \text{命題 3.4(2) より}\\
&= \gcd(308,\,1596)\\
&= \gcd(\mathrm{mod}(1596,308),\,308) && \text{命題 3.4(2) より}\\
&= \gcd(56,\,308)\\
&= \gcd(\mathrm{mod}(308,56),\,56) && \text{命題 3.4(2) より}\\
&= \gcd(28,\,56)\\
&= \gcd(\mathrm{mod}(56,28),\,28) && \text{命題 3.4(2) より}\\
&= \gcd(0,\,28)\\
&= 28
\end{aligned}$$

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

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

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。

$$\begin{aligned}
&\gcd(\boxed{\begin{matrix}\text{変数 } x\\ 1596\end{matrix}},\boxed{\begin{matrix}\text{変数 } y\\ 308\end{matrix}}) \Longrightarrow \gcd(\boxed{\begin{matrix}\text{変数 } x\\ 308\end{matrix}},\boxed{\begin{matrix}\text{変数 } y\\ 1596\end{matrix}})\\
&\Longrightarrow \gcd(\boxed{\begin{matrix}\text{変数 } x\\ 56\end{matrix}},\boxed{\begin{matrix}\text{変数 } y\\ 308\end{matrix}})\\
&\Longrightarrow \gcd(\boxed{\begin{matrix}\text{変数 } x\\ 28\end{matrix}},\boxed{\begin{matrix}\text{変数 } y\\ 56\end{matrix}})\\
&\Longrightarrow \gcd(\boxed{\begin{matrix}\text{変数 } x\\ 0\end{matrix}},\boxed{\begin{matrix}\text{変数 } y\\ 28\end{matrix}})\\
&\Longrightarrow 28
\end{aligned}$$

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。

$$\gcd(\boxed{\begin{matrix}\text{変数 } x\\ a\end{matrix}},\boxed{\begin{matrix}\text{変数 } y\\ b\end{matrix}}) \Longrightarrow \gcd(\boxed{\begin{matrix}\text{変数 } x\\ \mathrm{mod}(b,a)\end{matrix}},\boxed{\begin{matrix}\text{変数 } y\\ a\end{matrix}})$$

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

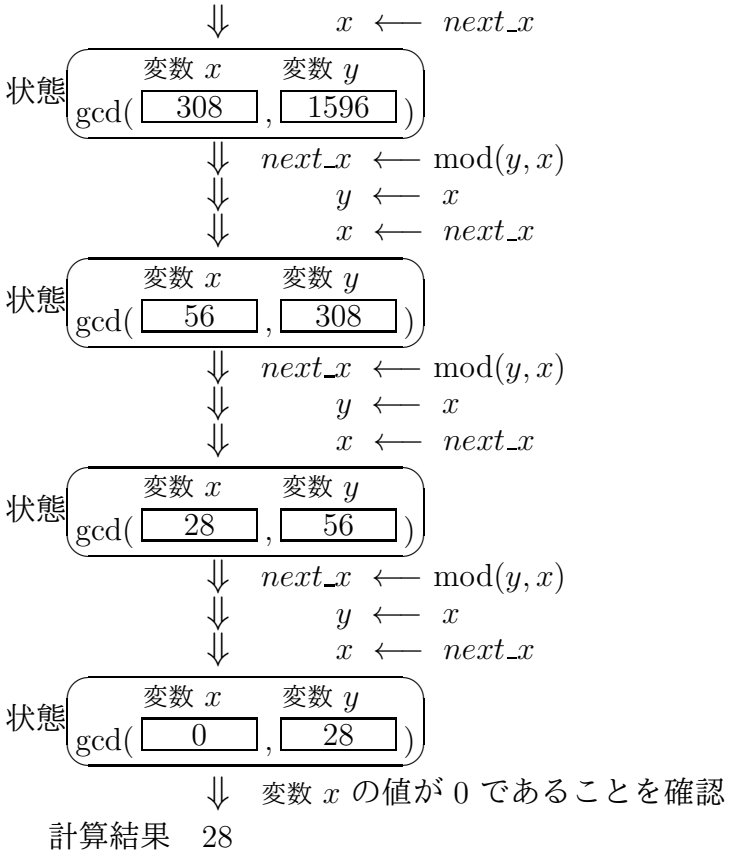
$$\begin{aligned}
next_x &\longleftarrow \mathrm{mod}(y,x)\\
y &\longleftarrow x\\
x &\longleftarrow next_x
\end{aligned}$$

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

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

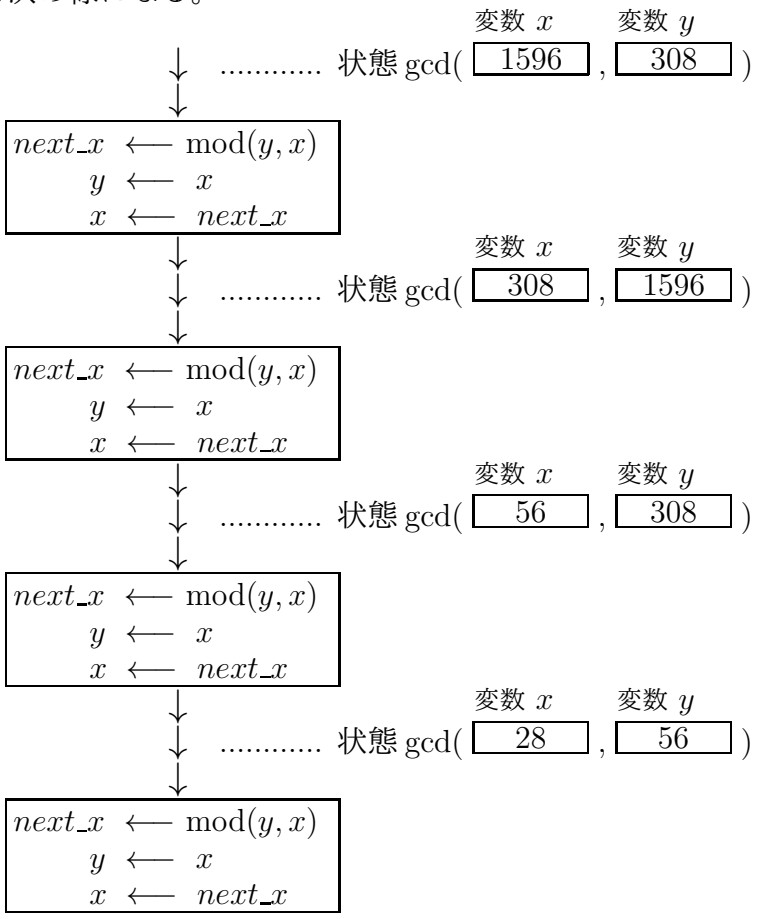
という式変形は、変数 x の値が 0 なので起こっていると考えられる。 結局、gcd(1596,308) の計算の場合に、どういう処理によってどういう風に状態が変わっていくかを具体的に明示すると次の様になる。

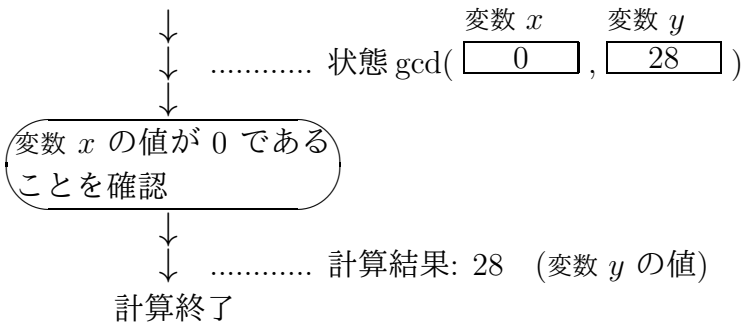
$$\text{状態}\ \underline{\gcd(\boxed{\begin{matrix}\text{変数 } x\\ 1596\end{matrix}},\boxed{\begin{matrix}\text{変数 } y\\ 308\end{matrix}})}$$
$$\Downarrow\quad next\_x \longleftarrow \mathrm{mod}(y,x)$$
$$\Downarrow\quad\quad y \longleftarrow x$$



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。

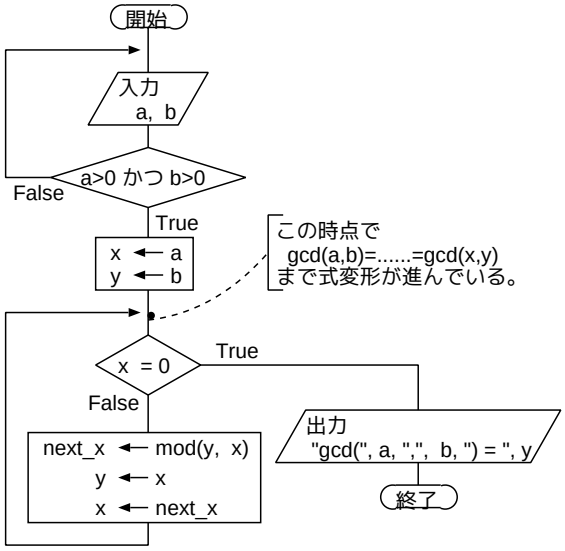




要するに、$x = 0$ となるまで

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

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。

例題 3.5 (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、

$$k\text{ が素数} \iff k\text{ は }2\sim k-1\text{ の整数で割り切れない}\quad(\text{定義より})$$
$$\iff k\text{ は }2\sim\sqrt{k}\text{ の整数で割り切れない}$$
$$\left(\begin{array}{l}i\text{ が }k\text{ の約数なら }k/i\text{ も }k\text{ の約数で}\\ i\text{ と }k/i\text{ のどちらかは }\sqrt{k}\text{ 以下となる}\\ \text{から}\end{array}\right)$$

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。

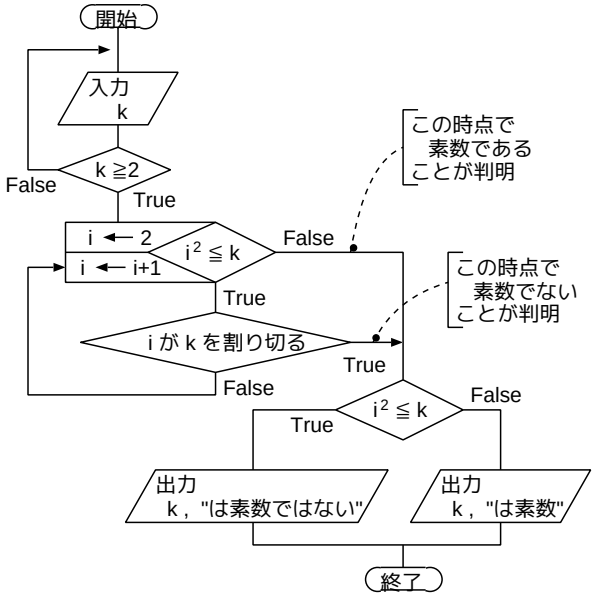
上記 (1) に対する方策: C 言語では、現在実行中の場所から見て最も内側の繰り返し (または switch 文) から脱出するために、**break 文**と呼ばれるものが用意されている。 プログラムを書く際は、それを使って繰り返しを途中で中止させる。

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 */
     2 /* 判定して答える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。2以上の整数を1つ入力して下さい: ");
     9     scanf("%d", &k);
    10   }while (!(k >= 2));

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

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

```
    19   return 0;
    20 }
[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。正整数を 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。正整数を 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 7〜10 行目 の **do-while 文**によって、

  > 条件 `!(k>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

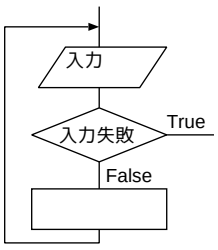
  ということを表す。
- プログラムの 11〜14 行目 の for 文 によって、

  > `i=2` という設定を行った後で 12〜13 行目を `i*i<=k` である間 繰り返す、
  > 但し、12〜13 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 12 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 13 行目 の **break 文**が実行されると、 11〜14 行目の繰り返しは即座に終了し、次に 15 行目が実行される。

3.4 自習 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 3.6 (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 1.3 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、

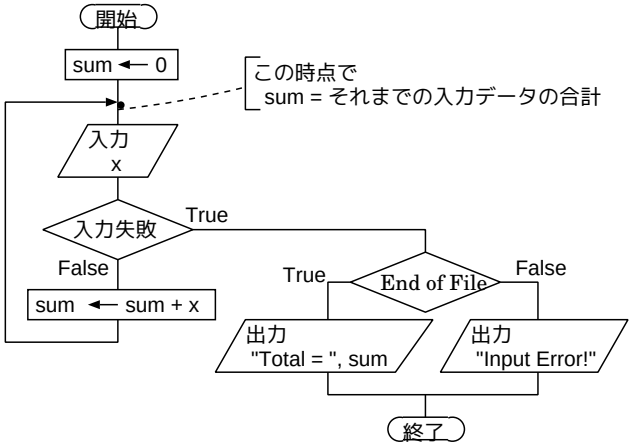
(a) 整数データ 1 個を 入力データを保持する変数 に読み込む。
(b) それまでの合計を保持する変数
⟵ それまでの合計を保持する変数 + 入力データを保持する変数

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

$$\text{関数 scanf の値} = \begin{cases} \text{データ入力に成功した回数} \\ \qquad\qquad \text{if 入力側に何らかのデータがあった} \\ \texttt{EOF} \quad (\text{普通 } -1 \text{ と} \texttt{<stdio.h>} \text{ で定義されるマクロ}) \\ \qquad\qquad \text{if 何も入力しないうちにファイルの終りに到達した} \end{cases}$$

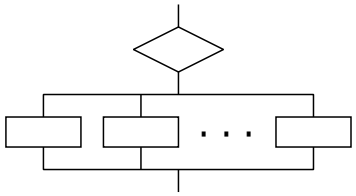
である。7~9 行目の while 文においては、(後での使用のために) 一旦この scanf の値を変数 scanf_val に代すること、および scanf の値が 1 である (すなわちデータ入力に成功する) 間 8 行目を繰り返すことを指示している。

- 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。

3.5 自習 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。

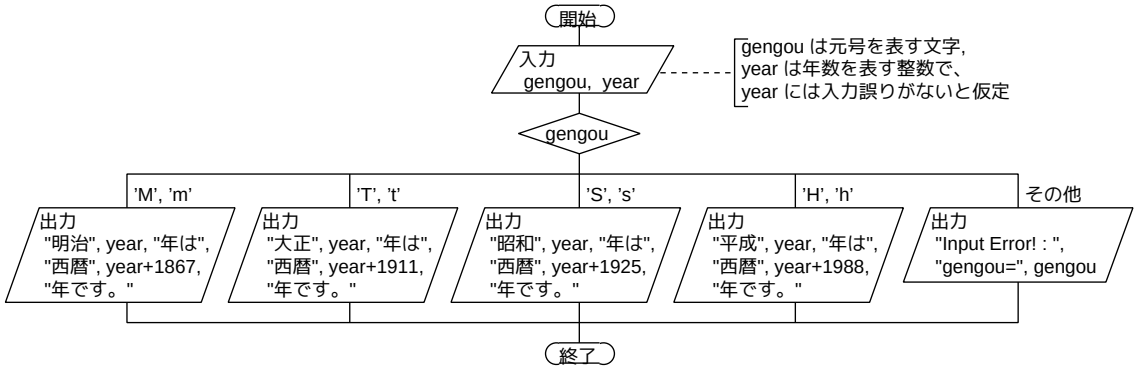


例題 3.7 (元号表記→西暦表記) 元号を表す文字 (M，m，T，t，S，s，H，または h) と年数を読み込み、その元号表記の年を西暦表記に変換して出力するCプログラムを作成せよ。

(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。

3.6 自習 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 1.3 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 3.3 や例題 3.6 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 $\Longrightarrow$ 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。

$\Longrightarrow$ 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 $\Longrightarrow$ 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.63 や p.68 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.63 や p.69 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.63 や p.68 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。

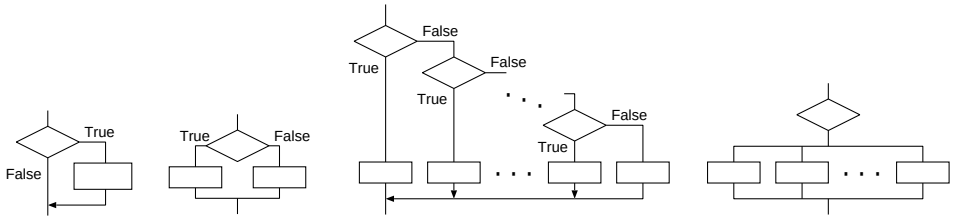
うまく式計算/代入の列が構成できない場合は、先のステップ①の作業に問題がある可能性が高いので、ステップ①に戻って下さい。

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。



構造 1　構造 2　構造 3　構造 4

構造 5　構造 6　構造 7　構造 8

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 3.5 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。($\Longrightarrow$ 例題 3.1 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。
  ($\Longrightarrow$ 例題 3.1 のアルゴリズム (2))
- 上の図の中の構造 3 は if-else-if-$\cdots$-if-else 構文 で表す。
  ($\Longrightarrow$ 例題 3.1 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。 ($\Longrightarrow$ 例題 3.7)
- 上の図の中の構造 5 は for 文 で表す。 ($\Longrightarrow$ 例題 1.3)
- 上の図の中の構造 6 は while 文 で表す。 ($\Longrightarrow$ 例題 3.3)
- 上の図の中の構造 7 は do-while 文 で表す。 ($\Longrightarrow$ 例題 3.3)
- 上の図の中の構造 8 は 条件部に scanf を含む while 文 で表す。
  ($\Longrightarrow$ 例題 3.6)

3.7 付録 制御構造のまとめ —C 文法のまとめ (2)—

3.7.1 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
> $\underbrace{\texttt{(-1<0)}}_{1}\texttt{<1} \Longrightarrow 1 < 1 \Longrightarrow 0$(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。
  $\Longrightarrow$ 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 3.8 (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) {
  ............
}
```

3.7.2 複合文と空文

複合文：

```
{
  宣言
  ⋮
  宣言
  文
  ⋮
  文
}
```

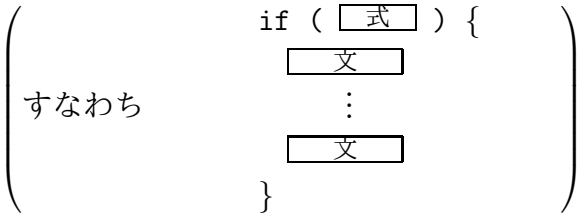
ブロック：
複合文のうち、宣言が 1 個以上含まれるもの。

空文：
セミコロンだけの文。

3.7.3 条件分岐の制御構造

if 文：

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



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。

3.7.4 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 )
```

3.7.5 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

演習問題

□**演習 3.1 (3 つの要素の最大値)** 例題 3.1 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 3.2 (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。

□**演習 3.3 (べき乗計算の効率)** 例題 3.2 で与えたべき乗計算プログラムでは一般の入力値 x と y に対して何回程度の乗算を行うことになるか調べよ。単純に x を掛ける作業を繰り返すアルゴリズムと計算効率を比較せよ。

□**演習 3.4**　命題 3.4 を証明せよ。

□**演習 3.5 (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。

□**演習 3.6 (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 3.7 (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。)

□**演習 3.8 (scanf の値)** 次の C プログラムを実行すると各々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]$ cat test0708_1a.c
#include <stdio.h>

int main(void)
{
  int i;
  printf("(1)scanf()=%d\n", scanf("%d", &i));
  return 0;
}
[motoki@x205a]$ gcc test0708_1a.c
[motoki@x205a]$ ./a.out
3
[                              ]
[motoki@x205a]$
```

□**演習 3.9 (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。

□**演習 3.10 (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 3.7 のプログラムを手直ししてみよ。

□**演習 3.11 (元号表記→西暦表記; scanf の%c)** 例題 3.7 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 ... 。]

4 復習 関数 (その1)

- 自習 数学的関数の利用, 標準ライブラリ,
- 自習 cc コマンドの-lm オプション,
- 関数定義,return 文, 関数プロトタイプ,
- コンパイル・リンクの処理の流れ,
- 自習 名前 (識別子) の有効範囲, 外部変数,
- 再帰,
- 付録 各種標準ライブラリ関数の使用案内

4.1 自習 数学的関数の利用

- C 言語では、三角関数, 対数関数, 指数関数, べき乗関数, 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。

例題 4.1 (三角関数の表) $x = 0°, 5°, 10°, \ldots, 90°$ に対して $\sin x, \cos x, \tan x$ の値を計算して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) 計算手順自体は $x = 5°, 10°, 15°, \ldots, 90°$ のそれぞれに対して順に $\sin x$, $\cos x$, $\tan x$ を計算して出力するだけの単純な規則的な繰り返しである。

(プログラミング) 角度 x を保持するための変数を `x`、角度 x をラジアンに変換した値を保持するための変数を `x_radian`、そして $\sin x, \cos x, \tan x$ の値を一時的に保持するための変数をそれぞれ `sin_x, cos_x, tan_x` 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 function-sin-cos-tan.c
     1  /* x=0 deg., 5 deg., 10 deg., ... , 90 deg.    */
     2  /* に対して sin x, cos x, tan x の値を計算して */
     3  /* 表の形に出力する C プログラム                */

     4  #include <stdio.h>
     5  #include <math.h>

     6  #define  PI  (3.1415926535897932)   /* 円周率 */

     7  int main(void)
     8  {
     9    int    x;
```

```
10    double x_radian, sin_x, cos_x, tan_x;

11    printf("x(degree)    sin(x)      cos(x)          tan(x)\n"
12           "-------------------------------------------------\n");
13    for (x=0; x<=90; x+=5) {
14      x_radian = (double)x * PI / 180.0;
15      sin_x = sin(x_radian);
16      cos_x = cos(x_radian);
17      tan_x = sin_x / cos_x;
18      printf("%6d     %10.8f  %10.8f  %15.8g\n",
19             x, sin_x, cos_x, tan_x);
20    }
21    return 0;
22  }
[motoki@x205a]$ gcc function-sin-cos-tan.c -lm
[motoki@x205a]$ ./a.out
x(degree)    sin(x)      cos(x)          tan(x)
-------------------------------------------------
     0     0.00000000  1.00000000               0
     5     0.08715574  0.99619470      0.087488664
    10     0.17364818  0.98480775       0.17632698
    15     0.25881905  0.96592583       0.26794919
    20     0.34202014  0.93969262       0.36397023
    25     0.42261826  0.90630779       0.46630766
    30     0.50000000  0.86602540       0.57735027
    35     0.57357644  0.81915204       0.70020754
    40     0.64278761  0.76604444       0.83909963
    45     0.70710678  0.70710678                1
    50     0.76604444  0.64278761        1.1917536
    55     0.81915204  0.57357644         1.428148
    60     0.86602540  0.50000000        1.7320508
    65     0.90630779  0.42261826        2.1445069
    70     0.93969262  0.34202014        2.7474774
    75     0.96592583  0.25881905        3.7320508
    80     0.98480775  0.17364818        5.6712818
    85     0.99619470  0.08715574        11.430052
    90     1.00000000  0.00000000    1.6331239e+16
[motoki@x205a]$
```

ここで、

- プログラム 5 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 15~16 行目 で sin, cos 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 14 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 18 行目 の出力書式中の %6d は、出力フィールドの大きさを 6 桁として右詰めに出力することを表す。 %10.8f は出力フィールドの大きさを 10 桁, 小数点以下の桁数を 8 桁として出力することを表す。[但し、いずれの場合も、データが大きい場合は必要なだけの桁数で出力される。] また、%15.8g は出力フィールドの大きさを 15 桁, 有効桁数を 8 桁として、出来るだけ指数部なしで出力することを表す。
- 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、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 2.5 節, 4.4 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 function-sin-cos-tan.c
/tmp/ccTg4Sgp.o: In function ‘main’:
/tmp/ccTg4Sgp.o(.text+0x60): undefined reference to ‘sin’
/tmp/ccTg4Sgp.o(.text+0x74): undefined reference to ‘cos’
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。

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

機能	関数名 (引数の並び)	引数の型	関数値の型	説明
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。[詳しくはケリー&ポール付録 A.13 節, 浦&原田付録 5 などを参照して下さい。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 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

4.2 関数定義

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を例示する。

例題 4.2 (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

$$\binom{n}{k} = \frac{\texttt{factorial}(n)}{\texttt{factorial}(k)\,\texttt{factorial}(n-k)}$$

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 1.3 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 function-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 function-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5~14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。main() については関数値の型が宣言されていないが、省略されているので 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、8~13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。

4.3 付録 関数の基本についてのまとめ —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。詳しくは、この講義ノートの 7.3 節を参照。)

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
  {
  　宣言
  　⋮
  　宣言
  　文
  　⋮
  　文
  } ... 本体 (複合文)

- 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。)

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。

4.4 どのようにコンパイル作業が進むのか？

{ 講義ノート 2.5 節 }

この講義ノートの 2.5 節で述べたように、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。)

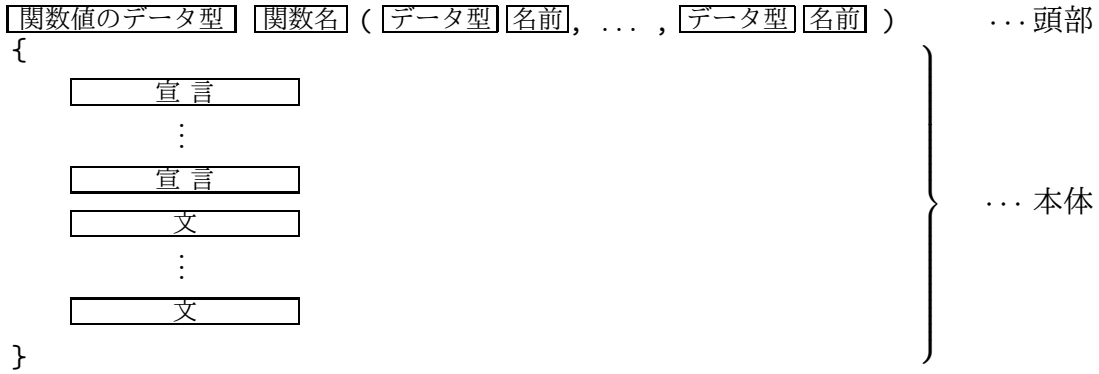
② コンパイル (各々の関数定義を機械語に翻訳, 場合によっては最適化も行う。)

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 3 の様になる。

4.5 自習 名前 (識別子) 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**

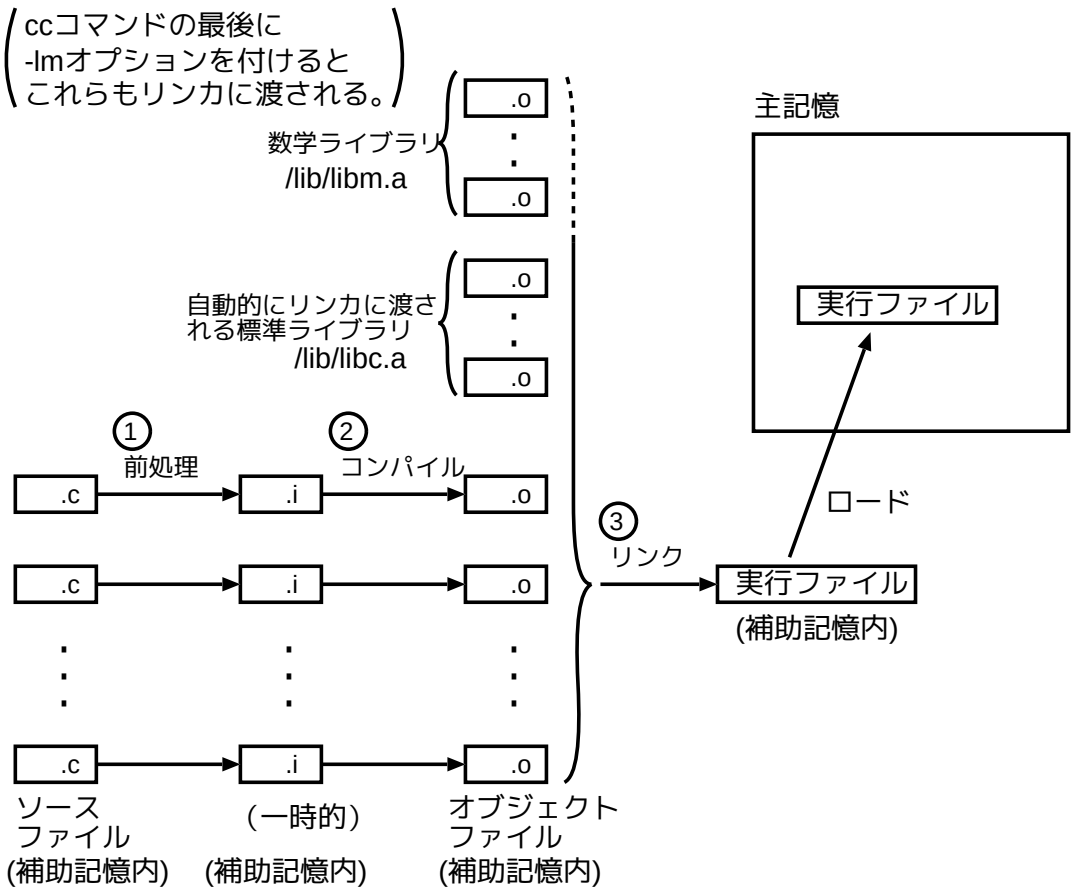


図 3: コンパイル作業の流れ

(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。

```
{
    宣 言
    ⋮
    宣 言
    文
    ⋮
    文
}
```

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 4.3 (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 function-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 function-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 6~16 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

$$\begin{cases} \text{8 行目の a は 7 行目で確保された a として、} \\ \text{11 行目の a は 10 行目で確保された a として、} \\ \text{13 行目の a は 7 行目で確保された a として、} \\ \text{19 行目の a は 4 行目で確保された a として、} \\ \text{20 行目の b は 18 行目で確保された b として} \end{cases}$$

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]
```

4.6 再帰

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 4.4 (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 4.2 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 4.2 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 4.2 の場合と変わらないので、main() 関数については例題 4.2 のものを何も変更せずにそのまま使える。従って、例題 4.2 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。

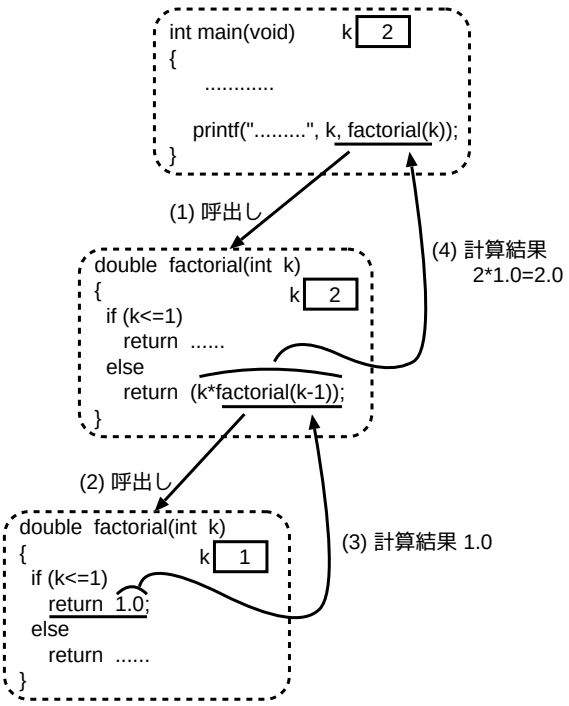
(プログラミング) 関数 factorial() を定義するにあたっては、例題 4.2 の場合と同じく仮引数として k という名前の int 型変数を用意した。作成した C プログラムと、これをコンパイル/実行している様子を次に示す。(下線部はキーボードからの入力を表す。)

```
[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。

例題 4.5 (クイック整列法) 大きさ 100 の int 型配列にランダムに整数を生成し、それらの配列要素を小さい順に並べ替えて出力する C プログラムを作成せよ。

(考え方) 行うべき処理は次の 3 つの独立した作業から成る。

- 大きさ 100 の int 型配列にランダムに整数を生成する作業。
- 与えられた配列内の要素を小さい順に並べ替える作業。
- 与えられた配列内の要素を順に出力する作業。

それゆえ、これらの作業を行う関数をそれぞれ別個に作り、main() 関数からこれらの関数を順に呼び出すことにする。

では、配列内にランダムに整数を生成する作業はどの様に行えば良いのか？ この例題の場合、良質の (疑似) 乱数を生成することが求められている訳ではないので、標準ライブラリ関数の `int rand(void)` を用いれば十分であろう。 (⟹ p.121 を参照。)

補足：
疑似乱数を用いて解を探索したりシミュレーションしたりする場合は、`int rand(void)` の様な安易な疑似乱数を用いたのでは実験結果そのものの信頼性も疑わしくなる。 MT(Mersenne Twister, http://www.math.keio.ac.jp/matumoto/mt.html) あたりを使うべきであろう。

「ランダム」というのは、単に我々の予測がつかないということではない。場合によっては、実験結果をさらに調べるために再実験する必要もある。

次に、配列内の要素を順に出力する作業はどの様に行えば良いのか？ 基本的には、1 番目の要素, 2 番目の要素, 3 番目の要素, ... と、順に出力するだけである。 ただその際、1 行の文字数が 50∼100 文字になる様に 1 行に出力するデータの個数を決め、その個数のデータ出力で 1 行が埋まり次第改行 (i.e. 改行コードを 1 個出力) した方が良い。そのためには、現在の行でデータ出力した個数を保持する変数 *count* を用意し、初期設定として *count*⟵0、データ出力の度に *count*⟵*count*+1、1 行が埋まる度に改行して *count*⟵0、とすれば良い。

また、配列内の要素を小さい順に並べ替える作業はどの様に行えば良いのか？ 整列化 (sorting) のアルゴリズムは色々なものがこれまでに考案されている。そのうち、ここでは **クイック整列法** (quicksort, **クイックソート**) を紹介しよう。

他の整列法としては、
選択整列法, 挿入整列法, バブル整列法, ヒープ整列法, シェル整列法, ... 等がある。クイック整列法は現在最も計算効率の良い整列化アルゴリズムとして有名である。

補足：
分割統治法 (divide-and-conquer method; 問題を規模の小さな問題に分割し、各々の小問題の解を統合して元の問題の解を構成する手法) と呼ばれるアルゴリズム構成法があるが、クイック整列法はこの分割統治手法を適用した代表例としても知られている。

具体的には、クイック整列法は

整列化の済んでいない部分列 $a[from], a[from+1], ..., a[to]$ が与えられた時、
それらの内容を並べ替えて

$$\underbrace{a[from], a[from+1], ..., a[p-1]}_{\text{全て } a[p] \text{ 以下}}, a[p], \underbrace{a[p+1], a[p+2], ..., a[to]}_{\text{全て } a[p] \text{ より大}}$$

(但し、p の値, $a[p]$ の値は任意。)
という風にする操作 (この操作を**分割操作**、$a[p]$ を**枢軸** (pivot) 要素という。)

を未整列の部分に繰り返し適用する方法で、次の様に整列処理を進める。

① 与えられたデータ $a[0], a[1], ..., a[n-1]$ に分割操作を適用する。(その結果、枢軸要素が $a[p]$ になったとする。)

② $a[0], a[1], ..., a[p-1]$ を整列化する小問題に対して、このアルゴリズムをさらに適用する。 (すなわち、$a[0], a[1], ..., a[p-1]$ に対して分割操作を適用し ..... 。)

③ $a[p+1], a[p+2], ..., a[n-1]$ を整列化する小問題に対して、このアルゴリズムをさらに適用する。 (すなわち、$a[p+1], a[p+2], ..., a[n-1]$ に対して分割操作を適用し ..... 。)

(プログラミング) 100 個の整数データを保持するために a という名前の int 型配列を用意し、

- 配列 a 内の各要素にランダムに整数を生成する作業,
- 配列 a 内の全要素を小さい順に並べ替える作業,
- 配列 a 内の全要素を順に出力する作業

を行うために各々 set_an_array_random(...), quicksort(...), pretty_print(...) という名前の関数を用意する。 配列 a[] の要素はこれら 3 つの関数から参照できる必要があるが、現時点では配列を関数の引数として受け渡す方法について説明していないので、ここでは配列 a[] は大域的なものとして宣言することにする。

では、関数 set_an_array_random(...) の引数と値はどう設定すれば良いのか？ 配列 a[] を大域的なものにしたので、main() 関数から引き渡すデータは何もないし、関数値として知らせてほしいものも何もない。従って、この関数のプロトタイプは次の様にすれば良い。

```
void set_an_array_random(void);
```

次に、関数 pretty_print(...) の引数と値はどう設定すれば良いのか？ この関数についても、set_an_array_random() 関数と同様に、main() 関数から引き渡すデータは何もないし、関数値として知らせてほしいものも何もない。従って、この関数のプロトタイプは次の様にすれば良い。

```
void pretty_print(void);
```

また、関数 quicksort(...) の引数と値はどう設定すれば良いのか？ クイック整列法を適用する部分は最初は配列 a[] 全体であるけれども、分割操作を何回か繰り返すことによって未整列の部分は配列内の色々な場所に小区間 (i.e. 添字の連続した配列要素の列) として残ることになる。そこで、関数 quicksort(...) の機能を単に配列 a[] 内の要素全体を小さい順に並べ替えるというものに限定するのではなく、配列 a[] 内の任意の小区間内の要素を小さい順に並べ替えれるものに一般化しておくと、分割操作後に出来る 2 つの小区間にクイック整列法を適用する処理は単に quicksort(...) を再帰的に呼び出す

だけで済む。実際、任意の小区間内の要素を小さい順に並べ替えれるものに拡張するために、並べ替える小区間内の最初と最後の要素番号 $from, to$ を関数 quicksort() の引数として使うことにすれば、quicksort($from, to$) の処理は次の様に書き表すことが出来る。

① 小区間内のデータ a[$from$], a[$from+1$], ..., a[to] に分割操作を適用する。(その結果、枢軸要素が a[p] になったとする。)

② quicksort($from, p-1$) を再帰的に呼び出す。

③ quicksort($p+1, to$) を再帰的に呼び出す。

関数 quicksort( ) の処理結果として main() 関数が受け取るものは、やはり何もない。従って、この関数のプロトタイプは次の様にすれば良い。

```
void quicksort(int from, int to);
```

以上の様に 3 つの関数 set_an_array_random(), quicksort(), pretty_print() を構成し、さらに quicksort() 関数の処理の見通しを良くするために小区間 a[from]～a[to] に対して分割操作を行い枢軸要素の添字番号を返す関数

```
int partition(int from, int to);
```

も構成することにする。 主関数 main() からこれらの関数を呼び出すことによって①ランダムに整数を生成して 100 個の配列要素 a[0]～a[99] を初期設定、②ランダムに生成されたデータの表示、③クイック整列法による配列内のデータの並べ替え、④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 function-quicksort.c
     1  /*****************************************************************/
     2  /* Quicksort  : 再帰計算の例                                     */
     3  /*---------------------------------------                        */
     4  /*    大きさ 100 の配列にランダムに整数を生成し、その配列要素を  */
     5  /*    Quicksort アルゴリズムを使って昇順に並べ替えて出力する。   */
     6  /*****************************************************************/

     7  #include <stdio.h>
     8  #include <stdlib.h>     /* 乱数発生のライブラリ関数を使うため */

     9  #define  SIZE   100
    10  #define  WIDTH  10
    11  #define  TRUE    1

    12  void set_an_array_random(void);
    13  void pretty_print(void);
    14  void quicksort(int from, int to);
    15  int  partition(int from, int to);

    16  int  a[SIZE];           /* 外部配列 */

    17  int main(void)
```

```
18  {
19    int  seed;

20    printf("Input a random seed (0 - %d):  ", RAND_MAX);
21    scanf("%d", &seed);
22    srand(seed);

23    set_an_array_random();
24    printf("\nbefore sorting:\n");
25    pretty_print();

26    quicksort(0, SIZE-1);
27    printf("\nafter sorting:\n");
28    pretty_print();
29    return 0;
30  }

31  /*-------------------------------------------------------------*/
32  /* 引数で与えられた配列の各要素をランダムに設定する          */
33  /*-----------------------------------------                    */
34  /* (仮引数) :  なし                                           */
35  /* (関数値) :  なし                                           */
36  /* (機能) :  配列要素 a[0]～a[SIZE-1] に 0～999 の間の乱数を  */
37  /*           設定する。                                       */
38  /*-------------------------------------------------------------*/
39  void set_an_array_random(void)
40  {
41    int i;

42    for (i=0; i<SIZE; ++i)
43      a[i] = rand() % 1000;
44  }

45  /*-------------------------------------------------------------*/
46  /* 引数で与えられた配列の要素を順番に全て出力する            */
47  /*-----------------------------------------                    */
48  /* (仮引数) :  なし                                           */
49  /* (関数値) :  なし                                           */
50  /* (機能) :  配列要素 a[0]～a[SIZE-1] の値を順番に全て出力   */
51  /*           する。但し、各々の値は横幅 7 カラムのフィールド  */
52  /*           に出力することにし、また、1 行に WIDTH 個の要素  */
53  /*           を出力する。                                     */
54  /*-------------------------------------------------------------*/
```

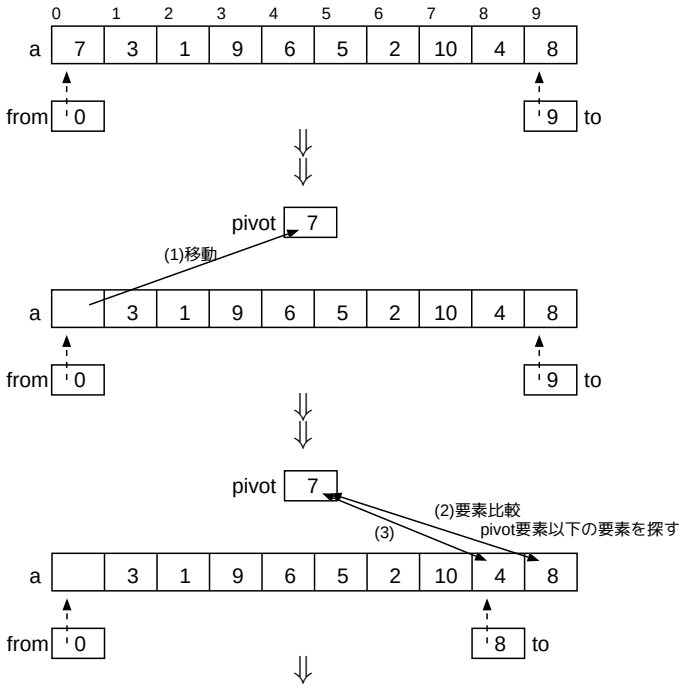
```
55  void pretty_print(void)
56  {
57    int i, count=1;

58    for (i=0; i<SIZE; ++i, ++count) {
59      printf("%7d", a[i]);
60      if (count >= WIDTH) {
61        printf("\n");
62        count = 0;
63      }
64    }
65    if (count > 1)
66      printf("\n");
67  }

68  /*-------------------------------------------------------------*/
69  /* 引数で与えられた配列要素を小さい順に並べ替える             */
70  /*-----------------------------------------                    */
71  /* (仮引数) from : int 型配列 a の添字                         */
72  /*          to   : int 型配列 a の添字                         */
73  /* (関数値) :  なし                                           */
74  /* (機能) :  quicksort アルゴリズムを使って、配列要素         */
75  /*                a[from],a[from+1],a[from+2], ..., a[to]     */
76  /*           を値の小さい順に並べ替える。                     */
77  /*-------------------------------------------------------------*/
78  void quicksort(int from, int to)
79  {
80    int  pivot_sub;              /* pivot subscript の意 */

81    if (from < to) {
82      pivot_sub = partition(from, to);       /* 分割操作 */
83      quicksort(from, pivot_sub - 1);
84      quicksort(pivot_sub + 1, to);
85    }
86  }

87  /*-------------------------------------------------------------*/
88  /* 引数で与えられた配列の部分列に quicksort の分割操作を施す  */
89  /*                                        (quicksort の関数)   */
90  /*-----------------------------------------                    */
91  /* (仮引数) from : int 型配列 a の添字                         */
92  /*          to   : int 型配列 a の添字                         */
93  /* (関数値) :  分割操作によって得られた枢軸要素の添字番号     */
```

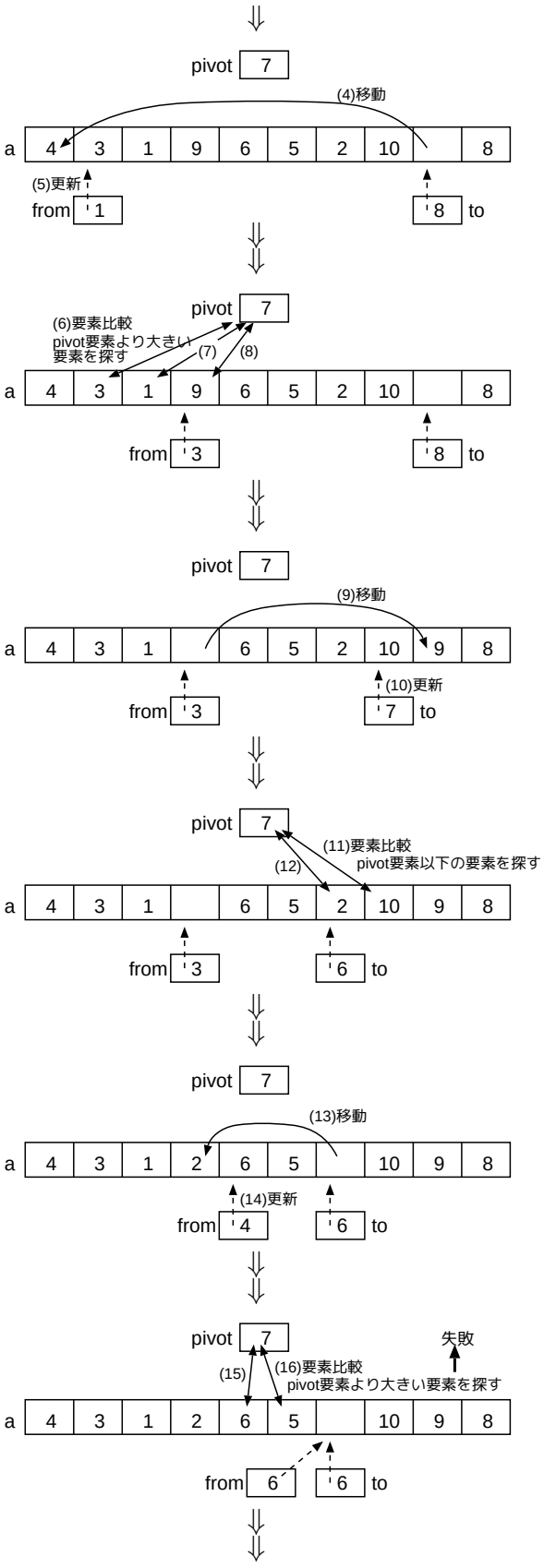
```
 94  /*              (以下の「(機能)」の項で出て来る pivot_sub)  */
 95  /* (機能) :  a[from]〜a[to] を並べ替えて                      */
 96  /*           max{a[from],...,a[pivot_sub-1]} <= a[pivot_sub]*/
 97  /*           a[pivot_sub] < min{a[pivot_sub+1],...,a[to]}   */
 98  /*           となるようにする。                               */
 99  /*-------------------------------------------------------------*/
100  int  partition(int from, int to)
101  {
102    int pivot;

103    pivot = a[from];      /* 最初の要素を枢軸要素に選ぶ。 */
104    while (TRUE) {        /* 工夫の余地あり。             */
105      for ( ; from<to && a[to]>pivot; --to)
106        ;
107      if (from == to) {
108        a[from] = pivot;
109        return from;
110      }
111      a[from++] = a[to];

112      for ( ; from<to && a[from]<=pivot; ++from)
113        ;
114      if (from == to) {
115        a[to] = pivot;
116        return to;
117      }
118      a[to--] = a[from];
119    }
120  }
[motoki@x205a]$ gcc function-quicksort.c
[motoki@x205a]$ ./a.out
Input a random seed (0 - 32767):  333

before sorting:
    556    289    435    368    666    319    214    273    132    585
     64    943    869    956     50    298    112    218      5    649
    603    936    515    385    671    776    137    886      4    563
    718    913    204    153    281    870    473    495    144    605
    432    208    548    653    517    950    951    629    520    957
    630    476    893    498    861    917    626    998    803    631
    913    521    544    470     27    825    340    500    672    836
    105    104    397      6    110    914    308     61    895    829
     18    878    305    264    376    518    181    354    517    336
```

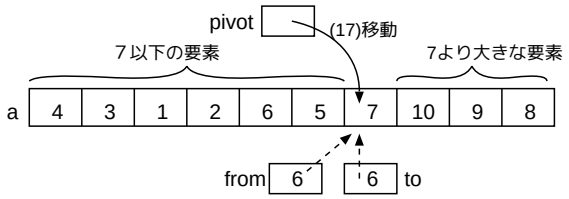
```
     985     782     857     881     252     236     706     945     736     730

after sorting:
       4       5       6      18      27      50      61      64     104     105
     110     112     132     137     144     153     181     204     208     214
     218     236     252     264     273     281     289     298     305     308
     319     336     340     354     368     376     385     397     432     435
     470     473     476     495     498     500     515     517     517     518
     520     521     544     548     556     563     585     603     605     626
     629     630     631     649     653     666     671     672     706     718
     730     736     776     782     803     825     829     836     857     861
     869     870     878     881     886     893     895     913     913     914
     917     936     943     945     950     951     956     957     985     998
[motoki@x205a]$
```

ここで、

- プログラムの 16 行目 は大きさ SIZE の int 型外部配列 a の確保を指示している。この配列は、後ろの 31~120 行目で定義された 4 つの関数 set_an_array_random, pretty_print, quicksort, partition からアクセスされることになる。 また、この講義ノートの 7.2 節で述べらている様に、この外部配列の領域は全てゼロに初期化される。
- このプログラムにおいては配列 a を外部配列として確保して大局的に使用したが、main 関数の中で局所的に確保して関数呼出し時に引数として配列を引き渡すことも勿論可能である。その方が実際には関数自体の独立性・汎用性が保てる。 配列を引数として受け渡す際の書き方については講義ノートの第 9 節等を参照。
- プログラムの 87~120 行目 に記述されている関数 partition(from, to) においては、データの移動は次の例の様に行われる。




pivot 7
(4)移動
a 4 3 1 9 6 5 2 10 8
(5)更新
from 1
8 to
pivot 7
(6)要素比較
pivot要素より大きい
要素を探す
(7)
(8)
a 4 3 1 9 6 5 2 10 8
from 3
8 to
pivot 7
(9)移動
a 4 3 1 6 5 2 10 9 8
(10)更新
from 3
7 to
pivot 7
(11)要素比較
pivot要素以下の要素を探す
(12)
a 4 3 1 6 5 2 10 9 8
from 3
6 to
pivot 7
(13)移動
a 4 3 1 2 6 5 10 9 8
(14)更新
from 4
6 to
pivot 7
失敗
(15)
(16)要素比較
pivot要素より大きい要素を探す
a 4 3 1 2 6 5 10 9 8
from 6
6 to



- プログラムの 78〜120 行目 に記述されている関数 quicksort() (と partition()) は次の様な形にコード化されることが多い。

```
[motoki@x205a]$ nl function-quicksort-std-code.c
         .........
    78 void quicksort(int from, int to)
    79 {
    80   int pivot, i, j, tmp;

    81   pivot = a[(from+to)/2];
    82   i = from;
    83   j = to;

    84   while (i <= j) {        /* 分割操作 */
    85     while (a[i] < pivot) i++;
    86     while (pivot < a[j]) j--;
    87     if (i <= j) {
    88       tmp  = a[i];  /* swap */
    89       a[i] = a[j];
    90       a[j] = tmp;
    91       i++;
    92       j--;
    93     }
    94   }

    95   if (from < j)           /* 再帰処理 */
    96     quicksort(from, j);
    97   if (i < to)
    98     quicksort(i, to);
    99 }
```

このコードはデータ交換を繰り返しデータ移動の回数が多くなるのでその改善を図り、合わせて分割操作をはっきりとした形で関数化したのが上記 p.109〜 のプログラムである。

4.7 付録 標準ライブラリ関数についての案内

{ 浦&原田付録 5, ケリー&ポール付録 A}

- 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。講義ノート 16.7 節を参照。
<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 4.1 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

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

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, int 型
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

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 1.4.5 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 1.4.6 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。講義ノート 5.2 節を参照。
`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。講義ノート 5.2 節を参照。
`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。

エラーメッセージ出力の関数 <stdio.h>：

関数プロトタイプ ··· 説明

`void perror(const char *s)`
··· ライブラリ関数がエラーを検出すると、外部変数として定義され<errno.h>の中で extern 宣言されている int 型変数 errno にエラー番号 (正) が記録される。perror 関数は、この変数 errno の値に対応するエラーメッセージを、引数で指定された文字列 s とともに次の形式で stderr に出力する。
`fprintf(stderr, "%s: %s\n", s, "エラーメッセージ");`

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 $[0, \texttt{RAND\_MAX}]$ 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。商と余りを表すメンバ名はそれぞれ quote と rem である。
`ldiv_t` ··· 関数 ldiv( ) が返す構造体 (long int 型のペア) のデータ型名。商と余りを表すメンバ名はそれぞれ quote と rem である。
`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。講義ノート 14.3 節を参照。
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。講義ノート 14.3 節を参照。
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。講義ノート 14.3 節を参照。
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。

演習問題

□**演習 4.1 (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。

□**演習 4.2 (数学関数)** 実数値 x を入力して $\sin x$ の値を出力する C プログラムを作ろうとした所、次の様にエラーメッセージが出力されてしまった。

```
xcspc60_148% nl test9909_4.c
     1  #include <stdio.h>
     2  int main(void)
     3  {
     4    double x;
     5    scanf("%lf", &x);
     6    printf("%e\n", sin(x));
     7    return 0;
     8  }
xcspc60_149% cc test9909_4.c
test9909_4.c: In function ‘main’:
test9909_4.c:6: warning: type mismatch in implicit declaration
                                                    for built-in function ‘sin’
未定義の                              最初に参照している
シンボル                                  ファイル
sin                                    /var/tmp/ccVNs_V_1.o
ld: 重大なエラー: シンボル参照エラー。a.out に書き込まれる出力はありません
xcspc60_150%
```

(1) このプログラムあるいはコマンド実行の中の、誤り箇所 (2 箇所) を指摘し、それぞれ修正せよ。(上の会話記録に直接書き込んで修正して下さい。)

(2) 問 (1) の修正によって何がどう正常になるか説明せよ。

□**演習 4.3 (数学関数)** 2 つの実数 x, y を読み込み、それらを要素とするベクトルのノルム $\sqrt{x^2+y^2}$ を計算して出力する C プログラムを作ろうとしたが、コンパイル時に次の様にエラーメッセージが出てしまった。

```
sv01_43: cat -n test0308_2.c
     1  #include <stdio.h>
     2  #include <math.h>
     3
     4  typedef struct {
     5    double x;
     6    double y;
     7  }Vector;
     8
     9  int main(void)
    10  {
    11    Vector vec;
    12
    13    scanf("%lf%lf", &vec.x, &vec.y);
    14    printf("norm(%12.5g, %12.5g) = %12.5g\n",
    15           vec.x, vec.y, norm(vec));
    16    return 0;
    17  }
    18
```

```
    19  double norm(Vector v)
    20  {
    21    return sqrt(v.x*v.x + v.y*v.y);
    22  }
sv01_44: gcc test0308_2.c
test0308_2.c:19: warning: type mismatch with previous implicit declaration
test0308_2.c:15: warning: previous implicit declaration of ‘norm’
test0308_2.c:19: warning: ‘norm’ was previously implicitly declared
                                                          to return ‘int’
未定義の                          最初に参照している
シンボル                              ファイル
sqrt                               /var/tmp/ccIhRKpD.o
ld: 重大なエラー: シンボル参照エラー。a.out に書き込まれる出力はありません。
collect2: ld returned 1 exit status
sv01_45:
```

(1) このプログラムあるいはコマンド実行の中の、誤り箇所 (2 箇所) を指摘し、それぞれ修正せよ。(上の会話記録に直接書き込んで修正して下さい。)

(2) 問 (1) の修正によって何がどう正常になるか説明せよ。

□**演習 4.4 (ヘッダファイルの中身)** 大抵のプログラムでは、プログラムの最初に `#include <stdio.h>` という (プリプロセッサ) 指令が置かれる。これは、`/usr/include/stdio.h` というファイルの中身をこの場所に挿入することによって入出力関係の標準ライブラリ関数のプロトタイプを宣言したり、入出力に関するマクロ名を定義したりしていることに他ならない。 `/usr/include/stdio.h` の中身をしばらく眺めて、この中にどんな風に関数プロトタイプが並んでいるか確かめよ。また、例えば関数 `printf` のプロトタイプはどういう風に記述されているか?

> 補足:
> 実習室の計算機においては、`/usr/include/stdio.h` の最初の方に `#include <iso/stdio_iso.h>` という行が置かれていて、そこで標準的な関数プロトタイプ、標準的なマクロ定義が取り込まれている。従って、`printf` のプロトタイプを見るには `/usr/include/iso/stdio_iso.h` というファイルを調べることになる。 例えば、次の様にする。
>
> ```
> cat /usr/include/iso/stdio_iso.h |grep printf
> ```

□**演習 4.5 (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せよ。

□**演習 4.6 (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 4.7 (Fibonacci 数列)** 45 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 1.9) 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 4.8 (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。

□**演習 4.9 (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せよ。

□**演習 4.10 (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せよ。

□**演習 4.11 (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]$
```

□**演習 4.12 ((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 4.13 (Fibonacci 数列)** 45 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 1.9) 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 4.14 (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。]

$$f(x) = \begin{cases} x - 10 & \text{if } x > 100 \\ f(f(x + 11)) & \text{otherwise} \end{cases}$$

□**演習 4.15 (Ackermann 関数)** 次の漸化式によって定義される非負整数の組から非負整数への関数 $A(m, n)$ 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、この関数を再帰無しで定義してみよ。 [**注意：** この関数は $A(0, n) = n+1, A(1, n) = n+2, A(2, n) = 2n + 3, A(3, n) = 2^{n+3} - 3, ...$ という風に第 1 引数が少し大きくなるだけで巨大な関数値を持つので、実際に実行する時は $m \leq 3$ としておかなければならない。また、$m = 3$ の場合でも n を大きくしすぎると計算がなかなか終わらない恐れがあります。]

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } m \neq 0, n = 0 \\ A(m - 1, A(m, n - 1)) & \text{otherwise} \end{cases}$$

□**演習 4.16 (クイック整列法の通常コード)** 例題 4.5 の最後に補足説明として示した通常の形の `quicksort()` のコードに関して、85 行目で変数 `i` の値が `i>=SIZE` になったり、86 行目で変数 `j` の値が `j<0` になったりすることがないことを説明せよ。

□**演習 4.17 (<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。

□**演習 4.18 (<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。

□**演習 4.19 (<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。

計算機内部でどういう処理が為されているか理解する。

5 自習 基本的データ型

- 基本的データ型,
- C 言語における文字の扱い —整数型 char—,
- getchar() と putchar(),
- 整数型 int,short,long,unsigned,
- 列挙型, ビット演算,
- 浮動小数点数型 float,double,long double,
- 誤差の話,
- 型変換, キャスト, sizeof 演算子,
- 16 進定数と 8 進定数

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
  - 配列型 ···同じデータ型の記憶領域の並びを表す。
  - 関数型 ···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。
  - ポインタ型···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。
  - 構造体型 ···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。Pascal のレコード型に相当。
  - 共用体型 ···1 つのデータ領域を複数の型のデータで共用することを表す。

5.1 復習 C 言語における文字の扱い —— 整数型 char ——

{ケリー&ポール 3.3 節, 付録 C}

- 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次の ASCII コード表に基づいて決められている。

下位4ビット	上位3ビット 0	1	2	3	4	5	6	7
0	ヌル文字	伝送制御拡張	空白	0	@	P	`	p
1	ヘディング開始	装置制御 1	!	1	A	Q	a	q
2	テキスト開始	装置制御 2	"	2	B	R	b	r
3	テキスト終了	装置制御 3	#	3	C	S	c	s
4	伝送終了	装置制御 4	$	4	D	T	d	t
5	問合せ	否定応答	%	5	E	U	e	u
6	肯定応答	同期信号	&	6	F	V	f	v
7	ベル	伝送ブロック終結	’	7	G	W	g	w
8	後退	取消	(	8	H	X	h	x
9	水平タブ	媒体終端	)	9	I	Y	i	y
A	改行	置換文字	*	:	J	Z	j	z
B	垂直タブ	拡張	+	;	K	[	k	{
C	書式送り	ファイル分離文字	,	<	L	\	l	|
D	復帰	グループ分離文字	-	=	M	]	m	}
E	シフトアウト	レコード分離文字	.	>	N	^	n	~
F	シフトイン	ユニット分離文字	/	?	O	_	o	抹消

機能文字 ... 機能文字

主要な文字の番号は次の通りである。

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)

  { signed char
  { unsigned char

- **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 5.1 (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]$
```

5.2 復習 1 文字入出力 — getchar() と putchar() —

{ケリー&ポール 3.8 節}

- 文字をそのまま入出力するための関数として getchar と putchar が用意されている。
  - `int getchar(void)`
    … 標準入力 stdin から文字を読み込み、その文字番号を値とする。
  - `int putchar(int )`
    … 引数で指定された値を番号とする文字を標準出力 stdout に書き出す。

> 補足：
> ともに、char で表せる保証の無い EOF を値とすることもあるので、関数値の型は char でなく int に設定されている。
>
> 実際、通常は `#define EOF (-1)` と定義されることが多いが、一般には EOF の値はそれぞれの処理系の中でどの文字コードとも重ならないという制約を満たせば良いだけなので、char で表せる保証は無い。

- getchar の関数値の型は int なので、getchar で読み込んだデータは int 型の変数に格納するのが無難。
- 関数の引数結合は"値呼出し"で行われるので、putchar の引数としては int 型だけでなく、char 型や short 型, long 型の任意の式が許される。[文字番号として可能な値を持てば良い。]
- getchar も putchar も<stdio.h>の中で定義されている引数付きマクロ。

例題 5.2 (英小文字 → 大文字) 文字を読み込みそれが英小文字なら大文字に直して出力する、という作業を繰り返す C プログラムを作成せよ。

(考え方) 「文字を読み込む」ということは文字の表すビット列をそのまま指定した変数領域にセットするということで、その (int 型) 変数の値が読み込んだ文字の番号に他ならない。この値が EOF なら繰り返し作業を中止することになる。また、p.131 の文字番号の表を見るとアルファベット順に英小文字の番号は 97(='a')~122(='z'), 英大文字の番号は 65('A')~90('Z') が割り当てられているので、読み込んだ文字の番号 c が 'a' 以上'z' 以下なら $c - $'a'$ + $'A' が大文字に変換後の文字の番号となる。

(プログラミング) 読み込む文字データを格納するために c という名前の int 型変数を用意することにすれば、指示された処理は単純な繰り返しで表される。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。

```
[motoki@x205a]$ nl datatype-lower2upper-Kelley.c
     1 #include <stdio.h>

     2 int main(void)
     3 {
     4   int  c;
```

```
 5   while ((c = getchar()) != EOF)
 6     if ('a' <= c && c <= 'z')
 7       putchar(c - 'a' + 'A');
 8     else
 9       putchar(c);
10   return 0;
11 }
[motoki@x205a]$ gcc datatype-lower2upper-Kelley.c
[motoki@x205a]$ ./a.out
a
A
B
B
 Ctrl -d
[motoki@x205a]$
```

例題 5.3 (8bit での整数表現) 長さが 8 のビット列 (すなわち 0 と 1 の数字列)

$$b_7b_6b_5b_4b_3b_2b_1b_0$$

を入力して、

① このビット列を unsigned char 型データと見た時に表す (非負) 整数値 $\sum_{i=0}^{7} b_i \times 2^i$、および

② このビット列を signed char 型データと見た時に表す整数値 $-b_7 \times 2^7 + \sum_{i=0}^{6} b_i \times 2^i$

を出力する C プログラムを作成せよ。

(考え方) 例題 5.2 と同じ考え方で、読み込んだ文字 '0', '1' の番号 c からその数字の表す数値に変換するには $c - $ '0' を計算すれば良い。 また、$\sum_{i=0}^{7} b_i \times 2^i$ と $-b_7 \times 2^7 + \sum_{i=0}^{6} b_i \times 2^i$ は各々次の様に計算すれば効率的に計算できる。

$$\begin{aligned} \textstyle\sum_{i=0}^{7} b_i \times 2^i &= (((...((b_7 \times 2 + b_6) \times 2 + b_5)...) \times 2 + b_2) \times 2 + b_1) \times 2 + b_0 \\ -b_7 \times 2^7 + \textstyle\sum_{i=0}^{6} b_i \times 2^i &= (((...((-b_7 \times 2 + b_6) \times 2 + b_5)...) \times 2 + b_2) \times 2 + b_1) \times 2 + b_0 \end{aligned}$$

(プログラミング) 読み込む文字データを一時的に格納するために c という名前の int 型変数を、読み込んだ $b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0$ の数としての値を格納するために int 型配列 bit[7]∼bit[0] を用意し、また $\sum_{i=0}^{7} b_i \times 2^i$, $-b_7 \times 2^7 + \sum_{i=0}^{6} b_i \times 2^i$ という累算値を計算するためにそれぞれ unsigned_val, signed_val という名前の変数を用意する。指示された処理を行う 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。

```
[motoki@x205a]$ cat -n datatype-bit-string-as-char.c
     1 /* 長さが8の 0 と 1 の数字列を入力して                    */
     2 /* (1) このビット列を unsigned char 型データと見た時に */
     3 /*     表す (非負) 整数値、および                        */
     4 /* (2) このビット列を signed char 型データと見た時に   */
     5 /*     表す整数値                                        */
     6 /* を出力する C プログラム                               */
```

```
 7
 8 #include <stdio.h>
 9 #include <stdlib.h>
10
11 int main(void)
12 {
13   int  c, bit[8], i;
14   unsigned char  unsigned_val;
15   signed   char  signed_val;
16
17   printf("Input a bit string of length 8: ");
18   for (i=7; i >= 0; ){
19     switch (c = getchar()) {
20     case '0': case '1':
21       bit[i--] = c-'0';
22       break;
23     case ' ': case '\n': case '\t':
24       break;
25     default:
26       printf("Invalid character appears.\n");
27       exit(EXIT_FAILURE);
28     }
29   }
30
31   unsigned_val = bit[7];
32   signed_val   = -bit[7];
33   for (i=6; i>=0; i--) {
34     unsigned_val = unsigned_val*2 + bit[i];
35     signed_val   = signed_val*2 + bit[i];
36   }
37
38   printf("\n==>The input bit string can be interpreted to\n"
39          "   have a value %d as an unsigned char data, and\n"
40          "   have a value %d as a signed char data.\n",
41          unsigned_val, signed_val);
42   return 0;
43 }
[motoki@x205a]$ gcc datatype-bit-string-as-char.c
[motoki@x205a]$ ./a.out
Input a bit string of length 8: 1111 1010

==>The input bit string can be interpreted to
   have a value 250 as an unsigned char data, and
```

```
   have a value -6 as a signed char data.
[motoki@x205a]$ ./a.out
Input a bit string of length 8: 0 1 0 1
0 111

==>The input bit string can be interpreted to
   have a value 87 as an unsigned char data, and
   have a value 87 as a signed char data.
[motoki@x205a]$
```

5.3 復習 データ型 int

{ ケリー&ポール 3.4 節 }

- 整数値を表すための最も標準的なデータ型。
- 普通、int 型データは 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。

- 1 ワードが 32 ビットの計算機の場合、int 型データは長さが 32 のビット列で表される。
  $\Longrightarrow 2^{32}$ 個の整数を表すことが可能。
  $\Longrightarrow$ 普通の計算機だと、
  $$-2^{31} = -2147483648 \quad \text{以上、}$$
  $$2^{31}-1 = 2147483647 \quad \text{以下}$$
  の整数を表せる。
- ごく普通の計算機の場合、
  1 ワード=32 ビットであり、長さが 32 のビット列
  $$b_{31}b_{30}b_{29}\cdots\cdots b_2b_1b_0$$
  は、int 型データとしては
  $$-b_{31}\times 2^{31} + \sum_{i=0}^{30} b_i \times 2^i$$
  という整数値を持つ。
- 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。例えば次の通り。

```
[motoki@x205a]$ nl datatype-int-overflow-Kelley.c
     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
[motoki@x205a]$ ./a.out
a=-294967296  b=2000000000  c=2000000000
[motoki@x205a]$
```

$\Longrightarrow$ 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。

- **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)

5.4 復習 整数型： char, short, int, long, unsigned

{ケリー&ポール 3.5 節}

- 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。(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 $\geq$ int のビット数
  $\geq$ short のビット数
  $\geq$ 16 ビット
  long のビット数 $\geq$ 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
	.....	

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

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

整数の内部表現形式：

- 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

  $\sum_{i=0}^{n-1} b_i \times 2^i$

  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

  $-b_{n-1}\times 2^{n-1} + \sum_{i=0}^{n-2} b_i \times 2^i$

  である。

5.5 列挙型

- 有限集合の中の個々の要素に記号の名前 (**列挙定数**という) を付け、プログラム内でそれらの名前を (マクロ名と同様に) 自由に使える様にするための機構である。
- 計算機内部では、個々の要素には整数の識別番号が付けられる。
  $\Longrightarrow$ 実体は int の部分データ型。
- 列挙定数の宣言は次のように行なう。

宣言		説明
`enum boolean {NO, YES, FALSE=0, TRUE};`	···	列挙定数 NO, YES,FALSE, TRUE の値 (識別番号) がそれぞれ 0, 1, 0, 1 に設定される。
`enum month {JAN=1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};`	···	列挙定数 JAN, FEB, MAR, ..., DEC の値 (識別番号) がそれぞれ 1, 2, 3, ..., 12 に設定される。

- 列挙型の変数を確保するには、例えば次のように書く。

```
enum boolean sw;
```

5.6 ビット演算

{ケリー&ポール 7.1~7.3 節}

論理演算子：

~ 整数式 ... 整数式の各ビットを反転する。(すなわち、1 なら 0 に, 0 なら 1 にする。)

整数式 1 & 整数式 2 ... それぞれのビット位置毎に論理積をとる。(すなわち、両方が 1 なら 1, そうでなければ 0 にする。)

整数式 1 ^ 整数式 2 ... それぞれのビット位置毎に排他的論理和をとる。(すなわち、一方だけが 1 なら 1, そうでなければ 0 にする。)

整数式 1 | 整数式 2 ... それぞれのビット位置毎に論理和をとる。(すなわち、どちらか一方でも 1 なら 1, そうでなければ 0 にする。)

シフト演算子：

整数式 1 << 整数式 2 ... 左シフト。整数式 1 のビット表現が整数式 2 ビット分だけ左にシフトされる。右側の空いた場所には 0 が補充される。

整数式 1 >> 整数式 2 ... 右シフト。整数式 1 のビット表現が整数式 2 ビット分だけ右にシフトされる。整数式 1 が unsigned の場合は左側の空いた場所には 0 が補充されるが、signed の場合には左から何が補われるかは計算機に依存する。

例題 5.4 (1 の補数,2 の補数,2 のべき乗倍) 整数データを 1 個入力してその 1 の補数,2 の補数,2 の 5 乗倍,2 の -5 乗倍の値を出力する C プログラムを作成せよ。

(考え方) 整数データの 1 の補数を求めるには、その整数データを表す 2 進内部表現の各ビットを反転すれば良い。2 の補数を求めるには、2 進内部表現の各ビットを反転した後に 1 を加算すれば良い。また、整数データを 2 の 5 乗倍, 2 の -5 乗倍するには、その整数データを表す 2 進内部表現のビット列をそれぞれ 5 ビットだけ左シフト, 5 ビットだけ右シフトすれば良い。

(プログラミング) 指示された処理を行う 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。

```
[motoki@x205a]$ cat -n datatype-complement-2pow.c
     1 #include <stdio.h>
     2
     3 int main(void)
     4 {
     5   int a;
     6
     7   scanf("%d", &a);
     8   printf("1's complement of %d = %d\n"
     9          "2's complement of %d = %d\n"
    10          "%d * (2 raised to the 5 power)  = %d\n"
```

```
    11          "%d * (2 raised to the -5 power) = %d\n",
    12          a, ~a, a, ~a+1, a, a<<5, a, a>>5);
    13   return 0;
    14 }
[motoki@x205a]$ gcc datatype-complement-2pow.c
[motoki@x205a]$ ./a.out
1234567
1's complement of 1234567 = -1234568
2's complement of 1234567 = -1234567
1234567 * (2 raised to the 5 power)  = 39506144
1234567 * (2 raised to the -5 power) = 38580
[motoki@x205a]$
```

例題 5.5 (int 型データのビット表現) int 型データを 1 個入力してそのビット表現を 0 と 1 の文字列として出力する C プログラムを作成せよ。

(考え方) int 型が 32 ビットで構成されているのだとすると、式 1<<31 の値の 2 進内部表現は "10000000 00000000 00000000 00000000" である。それゆえ、整数データの 2 進内部表現と 1<<31 の 2 進内部表現の間でビット毎の論理積を取れば、その結果は整数データの 2 進内部表現の最上位ビットが 1 であるかどうかによって

"10000000 00000000 00000000 00000000"

または

"00000000 00000000 00000000 00000000" (整数としての値は 0)

になる。 従って、読み込んだ int 型データ a と 1<<31 の間で&(ビット毎の論理積) 演算を行った結果が 0 になれば a のビット表現の最上位ビットは '0'、そうでなければ a のビット表現の最上位ビットは '1' であることが分かる。 データ a を構成する残りのビットも上位から順に調べたければ、a のビット表現を 1 ビットずつ左にシフトしながらその最上位ビットを同じ要領で調べて行けば良い。

(プログラミング) 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。

```
[motoki@x205a]$ cat -n datatype-bit-rep-of-int.c
     1 #include <stdio.h>
     2
     3 int main(void)
     4 {
     5   int a, i, mask=1<<31;
     6
     7   scanf("%d", &a);
     8   printf("Integer %d is internally represented "
                                "by the bit sequence\n"
     9          "     ", a);
```

```
    10
    11   for (i=0; i<32; i++) {
    12     if (i%8 == 0)
    13       putchar(' ');
    14     putchar((a&mask)==0 ? '0' : '1'); /* a&mask==0 と書くとダメ */
    15     a = a<<1;
    16   }
    17   printf(".\n");
    18   return 0;
    19 }
[motoki@x205a]$ gcc datatype-bit-rep-of-int.c
[motoki@x205a]$ ./a.out
1024
Integer 1024 is internally represented by the bit sequence
      00000000 00000000 00000100 00000000.
[motoki@x205a]$ ./a.out
-1
Integer -1 is internally represented by the bit sequence
      11111111 11111111 11111111 11111111.
[motoki@x205a]$
```

5.7 復習 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。

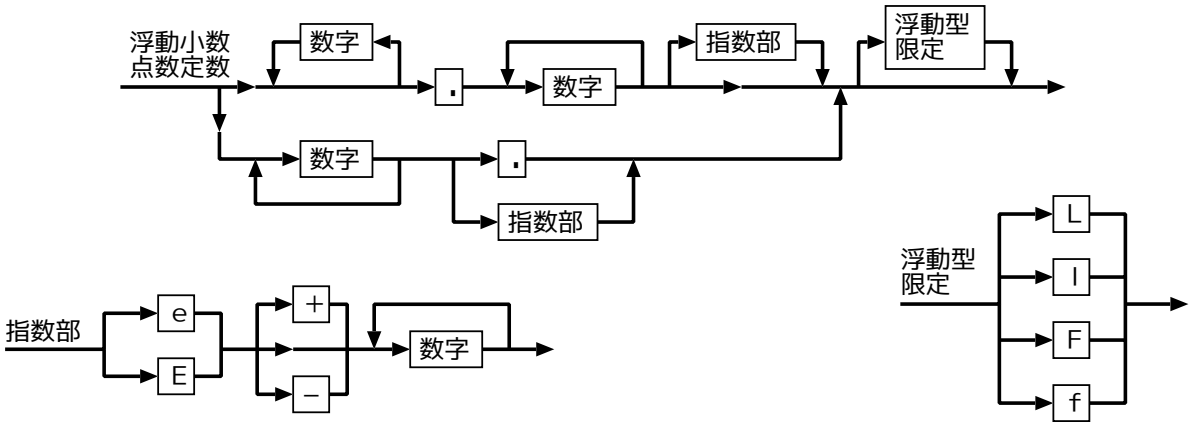
  `float` の精度 $\leq$ `double` の精度 $\leq$ `long double` の精度
- 標準ヘッダファイル `<float.h>` の中に、扱える最大の浮動小数点数などの情報が入っている。

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
.....		

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 $-38\sim+38, -308\sim+308$ 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、

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

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。

5.8 実数計算に伴って発生する誤差について

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 ↔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 →10 進変換) すると $9.5367431640625 \times 10^{-7}$ 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
$$1.23 \times 4.56 = 5.60\underline{88}$$
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。
$$\begin{array}{r} 1.234567 \\ +\ 0.04321098 \\ \hline 1.27777\underline{98} \end{array}$$
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。

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

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。]

5.9 復習 sizeof 演算子

{ ケリー&ポール 3.7 節 }

sizeof 演算子：

- 与えられたデータ型またはデータの大きさ (バイト数) を調べるための演算子。
- 次の様に書く。

```
sizeof( データ型 ),
sizeof( 式 ),        sizeof 式 ,
sizeof( 配列 ),      sizeof 配列 ,     ···配列全体の占めるバイト数
sizeof( 構造体 ),    sizeof 構造体 ,
                .............
```

- 関数呼出しと類似の表記法だが、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。
- 演算結果は普通 unsigned。

例 5.6 (基本データ型の大きさ,配列の大きさ) 基本データの大きさが実際にどうなっているかは <limits.h> を調べれば分かりますが、これは次の様にプログラムの中で調べることも出来ます。

```
[motoki@x205a]$ cat -n datatype-sizeof.c
     1 #include <stdio.h>
     2 int main(void)
     3 {
     4   int a[]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
     5
     6   printf("       char:%3d byte \n", sizeof(char));
     7   printf("      short:%3d byte \n", sizeof(short));
     8   printf("        int:%3d byte \n", sizeof(int));
     9   printf("       long:%3d byte \n", sizeof(long));
    10   printf("   unsigned:%3d byte \n", sizeof(unsigned));
    11   printf("      float:%3d byte \n", sizeof(float));
    12   printf("     double:%3d byte \n", sizeof(double));
    13   printf("long double:%3d byte \n\n", sizeof(long double));
    14
```

```
    15   printf("  array a[]:%3d byte \n", sizeof(a));
    16   return 0;
    17 }
[motoki@x205a]$ gcc datatype-sizeof.c
[motoki@x205a]$ ./a.out
       char:  1 byte
      short:  2 byte
        int:  4 byte
       long:  4 byte
   unsigned:  4 byte
      float:  4 byte
     double:  8 byte
long double: 12 byte

  array a[]: 40 byte
[motoki@x205a]$
```

5.10 復習 型変換とキャスト

{ ケリー&ポール 3.10 節 }

算術計算の際の自動型変換：

- 型の異なるデータ間で算術計算を行う際には、2 つのデータの型を揃えたりするために、内部では次の順に強制的に型変換が行われる。

  ① char 型や short 型のデータは int 型に変換される。

  ② データ型間の次の順序に従って、下位 (i.e. 左) の方の型が上位の型に変換される。(変換後の型が演算結果の型になる。)

  $$\texttt{int} < \texttt{unsigned} < \texttt{long} < \texttt{unsigned long} < \texttt{float} < \texttt{double} < \texttt{long double}$$

  $\Longrightarrow$ char 型同士の加算結果は char ではなく int 型。

- 実数 $\rightarrow$ 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 5.7 (キャスト演算の優先順位) `(float) i+3` は `((float)i) + 3` と同等。

5.11 復習 16進定数と8進定数

{ ケリー&ポール 3.11 節 }

16進数：

- 0~9, A~F(各々10~15の代わり) の16個の数字を用いて数を表したもの。
- 16進数字の列

  $$h_{n-1}h_{n-2}\cdots h_2h_1h_0$$

  によって、

  $$\sum_{i=0}^{n-1} h_i \times 16^i$$

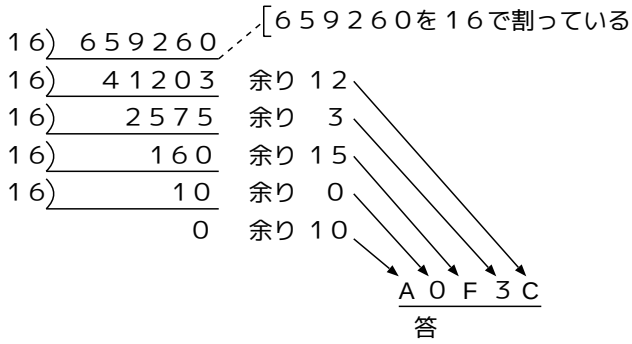
  という数を表す。例えば、16進数 A0F3C は次の10進数を表す。

  $$\underbrace{10}_{A}\times 16^4 + 0\times 16^3 + 0\times 16^3 + \underbrace{15}_{F}\times 16^2 + 3\times 16^1 + \underbrace{12}_{C}\times 16^0$$
  $$= 659260$$

- 非負整数 x を表す16進数を求めるには

  $$h_i = \mathrm{mod}(\lfloor x/16^i \rfloor, 16) = (\lfloor x/16^i \rfloor \text{ を } 16 \text{ で割った余り})$$

  を計算し、これらを並べればよい。例えば、$x = 659260$ の場合には次のように計算する。



8進数： (16進数の場合と同様)

16進定数と8進定数： C言語においては、

- 0 0~7の数字列 という形の字句は整数を8進表記したものと見なされる。
- 10~15を表す16進数字としては、英大文字 A~F と同様に英小文字 a~f も許され、0x 16進数字の列 または 0X 16進数字の列 という形の字句は整数を16進表記したものと見なされる。
- 変換指定を %x または %#x として書式付き出力 (e.g.printf) を行えば、整数を16進表記で出力できる。また、変換指定を %x として書式付き入力 (e.g.scanf) を行えば、16進表記の整数を入力できる。

- 変換指定を %o または %#o として書式付き出力 (e.g.printf) を行えば、整数を 8 進表記で出力できる。また、変換指定を %o として書式付き入力 (e.g.scanf) を行えば、8 進表記の整数を入力できる。
- unsigned や long の指定を行いたければ、10 進の場合と同様、それぞれ u(または U) や l(または L) を定数の最後に付ける。

例 5.8 (16 進定数,8 進定数の扱い) 16 進定数, 8 進定数, %x 変換, %#x 変換, %o 変換, %#o の使用例を次に示す。

```
[motoki@x205a]$ nl datatype-hexadecimal-const-Kelley.c
     1 #include <stdio.h>

     2 int main(void)
     3 {
     4   printf("%d %#x %#o\n", 19, 19, 19);

     5   printf("%d %x %o\n", 19, 19, 19);
     6   printf("%d %x %o\n", 0x1c, 0x1c, 0x1c);
     7   printf("%d %x %o\n", 017, 017, 017);

     8   printf("%d\n", 11 + 0x11 + 011);
     9   printf("%d\n", 2097151);
    10   printf("%d\n", 0x1FfFFf);
    11   return 0;
    12 }
[motoki@x205a]$ gcc datatype-hexadecimal-const-Kelley.c
[motoki@x205a]$ ./a.out
19 0x13 023
19 13 23
28 1c 34
15 f 17
37
2097151
2097151
[motoki@x205a]$
```

演習問題

□演習 5.1 (C 言語における文字の扱い) 次の C コードを実行するとどういう出力が得られるか？

```
printf("(5)%d\n", 49);
printf("(6)%c\n", 49);
```

□**演習 5.2 (C 言語における文字の扱い)** 次の C コードを実行するとどういう出力が得られるか？

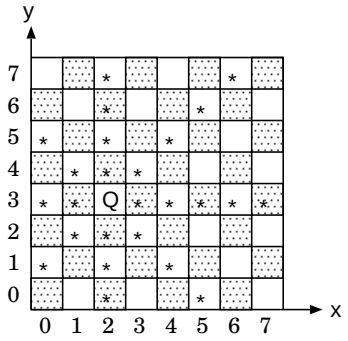
```
char a[]={'c', 'a', 't', 's', '\0'};
printf("(1)%c%c%c\n", a[0], a[1], a[2]);
printf("(2)%s\n", a+2);
printf("(3)%c%c%c\n", *a+1, *a+2, *a+3);
printf("(4)%3d%3d\n", a[0]-'a', a[1]-'a');
```

□**演習 5.3 (C 言語における文字の扱い)** 次の C コードを実行するとどういう出力が得られるか？

```
int a[4]={97,98,99,0};
printf("(2)%3d%3d%3d\n", a[0], a[1], a[2]);
printf("(3)%c%c%c\n", a[0], a[1], a[2]);
```

□**演習 5.4 (<stdio.h>の中身)** 2 つの関数 getchar と putchar が /usr/include/stdio.h の中でどのように定義されているか調べてみよ。

□**演習 5.5 (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*****
-***----
*-*-*---
--*--*--
```

□**演習 5.6 (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 5.7 (<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。

□**演習 5.8 (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。

□**演習 5.9 (右シフトの際の補充ビット)** 実習室の計算機において右シフトの際の補充ビットがどうなるか、調べてみて下さい。

□**演習 5.10 (シフト演算子の第 2 オペランドが負なら ...)** シフト演算子の第 2 オペランドが負の場合どういう演算結果になるか調べよ。

□**演習 5.11 (<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。

□**演習 5.12 (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：

$$\begin{aligned} & 2^{\sum_{i=0}^{7} e_i\times 2^i-127} \\ &= 2^{(e_7-1)\times 2^7+\sum_{i=0}^{6} e_i\times 2^i+1} \\ &= ((\cdots(((1/2^{(1-e_7)})^2\times 2^{e_6})^2\times 2^{e_5})^2\cdots)^2\times 2^{e_1})^2\times 2^{e_0}\times 2 \end{aligned}$$

□**演習 5.13 (データの内部表現)**

(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 5.14 (データの内部表現)

(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。

□演習 5.15 (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)))`

□演習 5.16 (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)

□演習 5.17 (10 進 →16 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 16 進数として表示する C プログラムを作成せよ。

6 自習 GDBデバッガ

- 実行時のエラーについて,
- core ファイルを用いたデバッグ,
- GDB を用いて実行追跡する例,
- GDB デバッガの使い方,
- 中断点を指定して実行追跡する例,
- GDB を使って変数の内部状態を調べる例,
- 実行中のプログラムの追跡,

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、実習室の計算機にはGDB というデバッガが備わっています。 GDB を用いれば、プログラムが異常終了した際に生成される (ことがある)`core` ファイルを使って異常の起こった場所を突き止めたり、C(や Fortran, Pascal などの) プログラムの実行を追跡して実行途中の変数値を調べたり、できます。

参考文献：

- R.M.Stallman&R.H.Pesch「GDB 入門」(1999 年, アスキー出版局, 1900 円＋税)
- 小山祐司&斉藤靖&佐々木浩&中込知之「UNIX 入門 フリーソフトウェアによる最新 UNIX 環境」(1996 年, トッパン)

6.1 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) ために、SIGSEGV(Segmentation Violation) という種類のシグナルが実行プロセスに送られて来た。 [デフォルトでは、このシグナルを送られたプロセスは `core` ファイルを生成して終了する。]
- `Bus error`
  ワード境界を無視してメモリアクセスを行ったために、SIGBUS という種類のシグナルが実行プロセスに送られて来た。 [デフォルトでは、このシグナルを送られたプロセスは `core` ファイルを生成して終了する。]
- `Illegal instruction`
  不正な (機械語) 命令を実行しようとしたために、SIGILL という種類のシグナルが実行プロセスに送られて来た。 [デフォルトでは、このシグナルを送られたプロセスは `core` ファイルを生成して終了する。]
- `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ために、SIGFPEという種類のシグナルが実行プロセスに送られて来た。 [デフォルトでは、このシグナルを送られたプロセスは `core` ファイルを生成して終了する。]

例 6.1 (バッファリングの影響) 入出力を効率的に行うために、通常、入出力装置との間のデータの受渡しはバッファ(buffer) と呼ばれるシステム内の記憶領域を介して行われる。すなわち、CPU による 主記憶 ↔ バッファ 間のデータ転送と、入出力装置による バッファ ↔ 入出力装置 間のデータ転送を並行して行い、また、入出力装置との間の実際のデータ転送の回数を少なく抑えることで、入出力の効率化が図られている。 この様なバッファリングの下では、実行時エラーのためにプログラムが強制終了させられた時に一部の出力がバッファ内に残ったままになることがあるので、注意が必要である。例えば、次のプログラム実行を考える。

```
[motoki@x205a]$ nl lab-divide-by-0.c
     1 /*------------------------------------------------------------*/
     2 /* 出力の効率を上げるためバッファリングが行われているので、 */
     3 /* 実行時エラーの直前の出力は画面に出力されないことがある。 */
     4 /*------------------------------------------------------------*/
     5 #include <stdio.h>

     6 int main(void)
     7 {
     8   int k;

     9   printf("Starting\n");
    10   printf("Before division... ");
    11   k = 1/0;                  /* 0 による除算 (エラー) */
    12   printf("After division\n");
    13   return 0;
    14 }
[motoki@x205a]$ gcc lab-divide-by-0.c
[motoki@x205a]$ ./a.out
Starting
Floating point exception (core dumped)
[motoki@x205a]$
```

プログラム 10 行目が実行されているにも関わらず `Before division` という文字列が画面に書き出されていないことに注目して下さい。このプログラム実行においては、プログラム 10 行目の `printf` の出力がバッファに入ったままの状態でプログラム 11 行目が実行され、それによってプログラムが停止しています。実行時エラーが発生すると、プログラムの実行は直ちに停止するのです。

$\Longrightarrow$ プログラムの出力列の正確な場所にエラーメッセージを入れたい場合は、次の様にこまめにバッファの内容を出力装置に吐き出します。

> 注意：
> これによって入出力の効率は確実に低下するので、fflush() 関数を普段使ってはいけません。

```
[motoki@x205a]$ nl lab-divide-by-0-fflush.c
     1 /*-------------------------------------------------------------*/
     2 /* 出力の効率を上げるためバッファリングが行われているので、 */
     3 /* 実行時エラーの直前の出力は画面に出力されないことがある。 */
     4 /* ==>これを避けるためにこまめにバッファの内容を出力装置に  */
     5 /*     吐き出すようにした。                                    */
     6 /*-------------------------------------------------------------*/
     7 #include <stdio.h>

     8 int main(void)
     9 {
    10   int k;

    11   printf("Starting\n");
    12   fflush(stdout);
    13   printf("Before division... ");
    14   fflush(stdout);
    15   k = 1/0;                  /* 0 による除算 (エラー) */
    16   printf("After division\n");
    17   fflush(stdout);
    18   return 0;
    19 }
[motoki@x205a]$ gcc lab-divide-by-0-fflush.c
[motoki@x205a]$ ./a.out
Starting
Before division... Floating point exception (core dumped)
[motoki@x205a]$
```

6.2 core ファイルを用いたデバッグ

UNIX では、プロセスが Segmentation fault などの原因で異常終了すると core という名前のファイルが作られ、そこに異常終了時のプログラムの状態 (メモリイメージ) が保存される様になっています。この core ファイルは巨大になりがちなので生成されない様に環境設定されていることもありますが、もし core ファイルが生成されているなら、これを使って GDB で異常発生の場所やその時の変数の値を調べることが出来ます。 [但し、GDB を効果的に使うためには gcc コマンドを -g オプション付きで実行して、得られた実行ファイルによって core ファイルが生成されている必要があります。]

例 6.2 (core ファイルを用いたデバッグ) core ファイルを用いて異常発生の場所を確かめている様子を次に示す。[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。]

> 注意：
> GDB との会話の中でプログラムの行番号を使うことがある。この時の行番号は空行もカウントしたものなので、GDB を使う際に行番号付きでプログラムを表示させる場合は、-a オプション付きの nl コマンド、または -n オプション付きの cat コマンドを使うべきである。

```
[motoki@x205a]$ cat -n lab-ex02-memory-overflow.c
                        ..................... 空行にも行番号を付けておく
     1 #include <stdio.h>
     2
     3 int main(void)
     4 {
     5   int  i, a[10];
     6
     7   a[1] = a[2] = 1;
     8   for (i=3; i<=10; ++i)             /* 間違っていても、エラー   */
     9     a[i] = a[i-1] + a[i-2];         /* にならないこともある。   */
    10
    11   printf("a[10]=%d\n", a[10]);      /* a[10] という領域は       */
    12                                     /* 本当は確保されていない。*/
    13
    14   printf("a[512]=%d\n", a[512]);    /* a[512] へのアクセスは出来る。*/
    15
    16   for (i=3; i<=1000; ++i)           /* i=513 の時になって       */
    17     a[i] = a[i-1] + a[i-2];         /* ようやくエラーになる。   */
    18   return 0;
    19 }
[motoki@x205a]$ ulimit -c 1000 ... core ファイルの大きさの上限を 1000kB に設定
```

> 補足：
> ulimit は bash(Bourne Again シェル) の下で使えるコマンドであって、実習室で標準になっている tcsh(Tenex C シェル) の下では同じ設定を　limit core 1000k　と書く。

```
[motoki@x205a]$ gcc -g lab-ex02-memory-overflow.c
                        ........ デバッグ情報付きの実行形式ファイルを生成
[motoki@x205a]$ ./a.out
a[10]=56
a[512]=0
Segmentation fault (core dumped)
[motoki@x205a]$ gdb a.out core ..... GDB デバッガを core ファイル付きで起動
GNU gdb 5.0
```

```
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or 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.
This GDB was configured as "i386-redhat-linux"...
Core was generated by `a.out'.
Program terminated with signal 11, セグメンテーション違反です.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x80484bc in main () at lab-ex02-memory-overflow.c:17
17      a[i] = a[i-1] + a[i-2];         /* ようやくエラーになる。   */
                                .... 異常があった行とその行番号が表示されている。
(gdb) list lab-ex02-memory-overflow.c:17
         ..... lab-ex02-memory-overflow.c というファイルの 17 行目付近を表示。
12                                      /* 本当は確保されていない。*/
13
14    printf("a[512]=%d\n", a[512]);    /* a[512] へのアクセスは出来る。*/
15
16    for (i=3; i<=1000; ++i)           /* i=513 の時になって        */
17      a[i] = a[i-1] + a[i-2];         /* ようやくエラーになる。   */
18    return 0;
19 }
(gdb) print i ................................ 異常発生時の変数 i の値を表示。
$1 = 513
(gdb) quit ..................................................... GDB デバッガを終了。
[motoki@x205a]$ ls -l core
-rw-------    1 motoki   motoki      61440 Mar  1 16:04 core
[motoki@x205a]$ rm core..... 大抵の場合 core ファイルは巨大になっているので、
                                削除しておく。
rm: `core' を削除しますか (yes/no)? y
[motoki@x205a]$
```

> **core** ファイルは普通は巨大なものになっていますから、
> 使った後は必ず削除しておくこと。

6.3 GDB を用いて実行追跡する例

GDB デバッガを用いれば、1 ステップずつ実行して時々変数値を観察することによって、プログラムの動作を細かく追跡することも出来ます。

例 6.3 (1 ステップずつ実行・追跡) 例 2.1 のプログラムの場合、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。]

```
[motoki@x205a]$ cat -n lab-ex01-ceiling.c ..... 空行にも行番号を付けておく
     1 #include <stdio.h>
     2 int main(void)
     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   return 0;
    12 }
[motoki@x205a]$ gcc -g lab-ex01-ceiling.c
[motoki@x205a]$ gdb a.out
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or 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.
This GDB was configured as "i386-redhat-linux"...
(gdb) break main ..... 中断点を関数 main の入口に設定
Breakpoint 1 at 0x80483ff: file lab-ex01-ceiling.c, line 6.
(gdb) run ..... 実行開始
Starting program: /home/motoki/C-Java2008/Programs-C/a.out

Breakpoint 1, main () at lab-ex01-ceiling.c:6
6   scanf("%d %d", &x, &y); ..... 次に実行する行が表示されている。
(gdb) next..... 関数 scanf の処理を 1 ステップと見なして 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 = 134513643 ..... { 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 normally.
(gdb) quit ..... GDB を終了
[motoki@x205a]$
```

6.4 GDB デバッガの使い方

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 [/*nfu*] *addr* ··· *addr* で指定されたアドレスから始まる、*n* 個のデータ (単位 *u*) の内容を書式 *f* で表示する。(このコマンド名は examine の意。) *n*,*f*,*u* の各々がオプション指定を表す。*n* 部は省略すると 1 と見なされる。*f* 部は print の *format* として許される指定に加え次の指定も可能で、省略すると (初期状態では) x(16 進) と見なされる。(デフォルト値は明示的な指定によって変わる。)

f(ormat)	意味
s	文字列表示
i	命令表示 (逆アセンブル)

  また、*u* 部は次の指定が可能で、省略すると (初期状態では) w(ワード) と見なされる。(デフォルト値は明示的な指定によって変わる。)

u(nit)	意味
b	byte
h	half word (2byte)
w	word (4byte)
g	giant word (8byte)

- 現在の中断点に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 *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。

6.5 中断点を指定して実行追跡する例

繰り返し構造のあるプログラムの場合は、実行ステップ数が大きく 1 ステップ実行だけでは実行追跡が煩わしくなります。こんな場合は、繰り返しループの中に数箇所の中断点を設け、その場所における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 6.4 (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。]

```
[motoki@x205a]$ cat -n lab-ex03-factorial.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 int main(void)
     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   return 0;
    16 }
[motoki@x205a]$ gcc -g lab-ex03-factorial.c
[motoki@x205a]$ gdb a.out
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or 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.
```

```
This GDB was configured as "i386-redhat-linux"...
(gdb) break 13
Breakpoint 1 at 0x8048430: file lab-ex03-factorial.c, line 13.
(gdb) run
Starting program: /home/motoki/C-Java2008/Programs-C/a.out
3..................................................... プログラムへの入力

Breakpoint 1, main () at lab-ex03-factorial.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-factorial.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 normally. ............................. 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   0x08048430 in main at lab-ex03-factorial.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   0x08048430 in main at lab-ex03-factorial.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   0x08048430 in main at lab-ex03-factorial.c:13
breakpoint already hit 2 times
(gdb) run.......................................... 2 度目のプログラム実行
Starting program: /home/motoki/C-Java2008/Programs-C/a.out
3..................................................... プログラムへの入力
```

```
Breakpoint 1, main () at lab-ex03-factorial.c:13
13      fact = fact * i;
2: fact = 1
1: i = 2
(gdb) cont
Continuing.

Breakpoint 1, main () at lab-ex03-factorial.c:13
13      fact = fact * i;
2: fact = 2
1: i = 3
(gdb) ..................................................... Enter のみ
Continuing.
3! = 6

Program exited normally.
(gdb) quit
[motoki@x205a]$
```

例 6.5 (自習 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。

```
[motoki@x205a]$ cat -n lab-ex04-factorial-func.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 int main(void)
     8 {
     9   int n;
    10
    11   scanf("%d", &n);
    12   printf("%d! = %ld\n", n, factorial(n));
    13   return 0;
    14 }
    15
    16 /************************************/
    17 /* 階乗計算の関数                   */
    18 /* -------------------------------- */
```

```
    19 /*   仮引数 n : 非負整数を想定       */
    20 /*   関数値   : n!                    */
    21 /************************************/
    22 long factorial(int n)
    23 {
    24   long fact;
    25
    26   if (n == 0)
    27     fact = 1;
    28   else
    29     fact = n * factorial(n-1);
    30
    31   return(fact);
    32 }
[motoki@x205a]$ gcc -g lab-ex04-factorial-func.c
[motoki@x205a]$ gdb a.out
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or 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.
This GDB was configured as "i386-redhat-linux"...
(gdb) break factorial
Breakpoint 1 at 0x8048442: file lab-ex04-factorial-func.c, line 26.
(gdb) break 30
Breakpoint 2 at 0x8048469: file lab-ex04-factorial-func.c, line 31.
(gdb) run
Starting program: /home/motoki/C-Java2008/Programs-C/a.out
3..................................................... プログラムへの入力

Breakpoint 1, factorial (n=3) at lab-ex04-factorial-func.c:26
26   if (n == 0)
(gdb) step
29     fact = n * factorial(n-1);
(gdb)

Breakpoint 1, factorial (n=2) at lab-ex04-factorial-func.c:26
26   if (n == 0)
(gdb) cont
Continuing.

Breakpoint 1, factorial (n=1) at lab-ex04-factorial-func.c:26
```

```
26   if (n == 0)
(gdb)
Continuing.

Breakpoint 1, factorial (n=0) at lab-ex04-factorial-func.c:26
26   if (n == 0)
(gdb)
Continuing.

Breakpoint 2, factorial (n=0) at lab-ex04-factorial-func.c:31
31   return(fact);
(gdb) display fact
1: fact = 1
(gdb) step
32 }
1: fact = 1
(gdb)

Breakpoint 2, factorial (n=1) at lab-ex04-factorial-func.c:31
31   return(fact);
1: fact = 1
(gdb) cont
Continuing.

Breakpoint 2, factorial (n=2) at lab-ex04-factorial-func.c:31
31   return(fact);
1: fact = 2
(gdb)
Continuing.

Breakpoint 2, factorial (n=3) at lab-ex04-factorial-func.c:31
31   return(fact);
1: fact = 6
(gdb)
Continuing.
3! = 6

Program exited normally.
(gdb) quit
[motoki@x205a]$
```

6.6 GDB を使って変数の内部状態を調べる

GDB の print コマンドはその時の式の値を表示するのに役立つが、2 進表示の指定がされた時でも上位の 0 が表示されない、といった風に変数の bit 毎の内容をそのまま正確に表示する訳ではない。指定したアドレスの内容を表示するには x コマンドを用いる。

例題 6.6 (整数型変数の内部の状態を調べる) 5.3~5.4 節ではプログラムの中で整数値がどの様に表されているか説明した。これらの事柄を GDB のコマンドを用いて実際に自分の目で観察してみよ。例えば、$-2, -1, 0, 1, 2, 3, 10, 57$ といった整数が実際に内部でどう表されるかを、GDB を用いて調べてみよ。

(考え方) 実際に $-2, -1, 0, 1, 2, 3, 10, 57$ といった整数値を保持した int 型変数を用意し、それらの変数を構成する内部のビット列の状態を GDB を用いてのぞき見すれば良い。GDB において指定したアドレスから始まる領域の内容を 2 進表示するには x コマンドに t オプションを付けて、x/t &変数名 という風な実行をすれば良い。

(プログラムと GDB 実行) 整数値 $-2, -1, 0, 1, 2, 3, 10, 57$ を int 型変数に保持したプログラムと GDB の x コマンドを用いてこれらの変数を構成する内部のビット列の状態を観察している様子を次に示す。

```
[motoki@x205a]$ cat -n peep-integer-variables.c
     1 #include <stdio.h>
     2
     3 int main(void)
     4 {
     5   int  minus2=-2, minus1=-1, zero =0,
     6        plus1 = 1, plus2 = 2, plus3=3,
     7        plus10=10, plus57=57;
     8
     9   printf(
    10     "minus2=%d minus1=%d zero =%d\n"
    11     "plus1 = %d plus2 = %d plus3=%d\n"
    12     "plus10=%d plus57=%d",
    13     minus2, minus1, zero,
    14     plus1,  plus2,  plus3,
    15     plus10, plus57);
    16   return 0;
    17 }
[motoki@x205a]$ gcc -g peep-integer-variables.c
[motoki@x205a]$ gdb a.out
       (GDB からのメッセージ)
(gdb) break 9
Breakpoint 1 at 0x8048406: file peep-integer-variables.c, line 9.
```

```
(gdb) run
Starting program: /home/motoki/C-Java2008/Programs-C/a.out

Breakpoint 1, main () at peep-integer-variables.c:9
9   printf(
(gdb) x/t &zero
0x7ffff7cc: 00000000000000000000000000000000
(gdb) x/t &plus1
0x7ffff7c8: 00000000000000000000000000000001
(gdb) x/t &plus2
0x7ffff7c4: 00000000000000000000000000000010
(gdb) x/t &plus3
0x7ffff7c0: 00000000000000000000000000000011
(gdb) x/t &plus10
0x7ffff7bc: 00000000000000000000000000001010
(gdb) x/t &plus57
0x7ffff7b8: 00000000000000000000000000111001
(gdb) x/t &minus1
0x7ffff7d0: 11111111111111111111111111111111
(gdb) x/t &minus2
0x7ffff7d4: 11111111111111111111111111111110
(gdb) cont
Continuing.
minus2=-2 minus1=-1 zero =0
plus1 = 1 plus2 = 2 plus3=3
plus10=10 plus57=57
Program exited normally.
(gdb) quit
[motoki@x205a]$
```

(観察結果について) 5.3〜5.4 節の説明の通りだとすると、例えば $-1\times 2^{31}+\sum_{i=0}^{30} 1\times 2^i = -1$ であるので、-1 という整数はコンピュータ内部では 111111$\cdots$111 というビット列によって表されるはずである。 上の観察結果は確かにこれと一致している。

例題 6.7 (8bit での整数表現) 例題 5.3 では、長さが 8 のビット列 $b_7b_6b_5b_4b_3b_2b_1b_0$ を入力して、

① このビット列を unsigned char 型データと見た時に表す (非負) 整数値 $\sum_{i=0}^{7} b_i\times 2^i$、および

② このビット列を signed char 型データと見た時に表す整数値 $-b_7\times 2^7+\sum_{i=0}^{6} b_i\times 2^i$

を出力する C プログラムを示した。このプログラムの場合、計算結果として得られた 2 つの整数値は、コンピュータ内部では入力したビット列で表されているはずであるが、これを GDB の x コマンドを用いて観察してみよ。

(考え方) 計算結果が得られた直後に中断点を設定した上で GDB 上でプログラムを実行し、計算結果の入っている 2 つの変数の内部のビット列の状態を例題 6.6 の場合と同じ様に x コマンドを用いて観察するだけである。

(GDB 実行) 観察している様子を次に示す。

```
[motoki@x205a]$ cat -n datatype-bit-string-as-char.c
         (例題 5.3 を参照)

[motoki@x205a]$ gcc -g datatype-bit-string-as-char.c
[motoki@x205a]$ gdb a.out
       (GDB からのメッセージ)
(gdb) break 38
Breakpoint 1 at 0x8048550: file datatype-bit-string-as-char.c, line 38.
(gdb) run
Starting program: /home/motoki/C-Java2008/Programs-C/a.out
Input a bit string of length 8: 0000 1011

Breakpoint 1, main () at datatype-bit-string-as-char.c:38
38   printf("\n==>The input bit string can be interpreted to\n"
(gdb) x /t &unsigned_val
0xbffff583: 11111111111111111111111100001011
         ... 最後の 8bit だけが unsigned_val の領域。残りは変数 i の領域の一部。
(gdb) x /tb &unsigned_val
0xbffff583: 00001011
(gdb) x /tb &signed_val
0xbffff582: 00001011
(gdb) x /2tb &signed_val
0xbffff582: 00001011 00001011
                     ..................... 2 つ目の 8bit は unsigned_val の内容
(gdb) cont
Continuing.

==>The input bit string can be interpreted to
   have a value 11 as a unsigned char data, and
   have a value 11 as a signed char data.

Program exited normally.
(gdb) run
Starting program: /home/motoki/C-Java2008/Programs-C/a.out
Input a bit string of length 8: 1111 1111

Breakpoint 1, main () at datatype-bit-string-as-char.c:38
38   printf("\n==>The input bit string can be interpreted to\n"
(gdb) x /tb &unsigned_val
```

```
0xbffff583: 11111111
(gdb) x /tb &signed_val
0xbffff582: 11111111
(gdb) cont
Continuing.

==>The input bit string can be interpreted to
   have a value 255 as a unsigned char data, and
   have a value -1 as a signed char data.

Program exited normally.
(gdb) quit
[motoki@x205a]$
```

6.7 DDD —GDB のグラフィカルなフロントエンド—

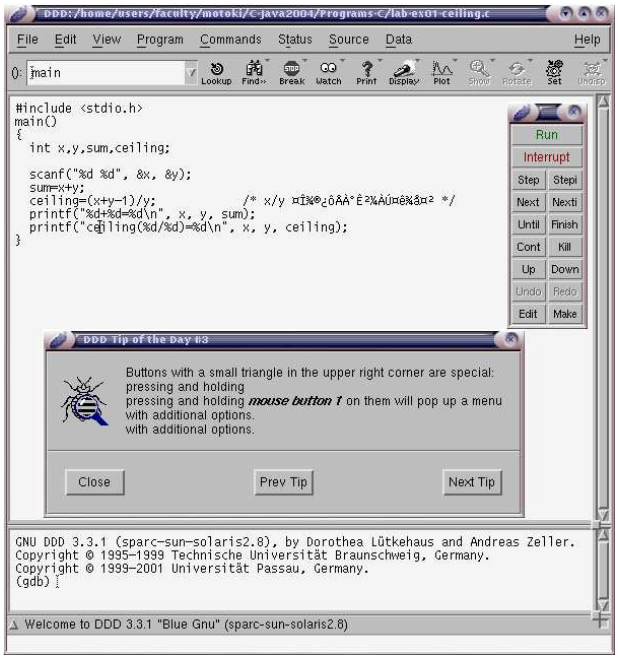
{ 工藤「UNIX プログラミングの道具箱」第 10 章 }

GDB のグラフィカルなフロントエンドとして動作する **DDD** と呼ばれるツールも存在します。実習室にはインストールされていませんが、DDD を使えば、GUI ベースに GDB を使えるのはもちろん、デバッグ対象のデータをグラフィカルに表示したり、配列内のデータ列をまとめてグラフ表示 (プロット表示) したり、といったことも出来ます。

DDD/GDB を用いて C プログラムの実行追跡をするには次の様にします。

(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。

(2) ddd 実行形式ファイル & とコマンド入力して DDD/GDB を起動する。



(3) Tips ウィンドウを閉じる。

(4) 様々な操作を行う。例えば、

- DDD というタイトルの縦長のウィンドウ (**コマンドツールウィンドウ**と言う) の中のボタンを押して step 実行等の指示を行う。
- ソースコードの行頭をマウスでクリックした後でツールバーの Break ボタンを押すと、その行がブレイクポイントとして設定される。
- ブレイクポイントとして登録された行頭をクリックすると Break ボタンが Clear ボタンに変わる。この状態で Clear ボタンを押すと、ブレイクポイントの設定が解除される。
- ツールバー左端の空欄に関数名を入れて Lookup ボタンを押すと、指定した関数のソースコードが表示される。
- 注目したい変数を選択しツールバーの Display ボタンを押すと、その変数の内容がグラフィカルに表示される様になる。

.....

6.8 実行中のプログラムの追跡

GDB は実行中のプロセスも追跡対象にすることが出来ます。具体的には、次の様にします。

```
xcspc70_43% gdb 実行形式ファイル
.....
(gdb) attach process-ID ..... ID 番号が process-ID のプロセス (実行中)
                               を GDB に接続
.....
(gdb) detach process-ID ..... 以前に attach したプロセス (実行中) を
                               GDB から切り離す
.....
(gdb) quit
```

演習問題

□**演習 6.1 (四則演算)** GDB を用いて例題 1.1 のプログラムの実行を追跡してみよ。

□**演習 6.2 (円錐の体積)** GDB を用いて例題 1.2 のプログラムの実行を追跡してみよ。

□**演習 6.3 (3 要素の最大値, その 2)** GDB を用いて例題 3.1 アルゴリズム (2) のプログラムの実行を追跡してみよ。

□**演習 6.4 (二項係数の計算)** GDB を用いて例題 4.2 のプログラムの実行を追跡してみよ。

□**演習 6.5 (print コマンドの引数)** GDB の print コマンドの引数として与えられる式の中に、数学的関数を使うことが出来るかどうか調べよ。 使えないとすると、それは何故か？

□**演習 6.6 (GDB デバッガ)** これまでに作成した C プログラムの実行を GDB デバッガで追跡してみよ。

□**演習 6.7 (GDB デバッガ)** 2 つの実数 x, y を入力しその和を計算して出力する C プログラムを作ろうとしたが、次の様に明らかに間違った結果が出力されてしまった。何が起こったのかを簡単に説明し、プログラムの誤りを修正せよ。 (ここで、下線部はキーボードからの入力を表す。参考のため、GDB で変数の値を追跡・観察した様子も示す。)

```
[motoki@x205a]$ cat -n test0208_2.c
     1 #include <stdio.h>
     2
     3 int main(void)
     4 {
     5   float x,y,sum;
     6
     7   scanf("%d%d", &x, &y);
     8   sum=x+y;
     9   printf("%d + %d = %d\n", x, y, sum);
    10   return 0;
    11 }
[motoki@x205a]$ gcc -g test0208_2.c
[motoki@x205a]$ ./a.out
1 -1
0 + 916455424 = -536870912
[motoki@x205a]$
```

参考： GDB による追跡記録

```
[motoki@x205a]$ gdb a.out
GNU gdb 5.0
Copyright 2000 ...(以下省略)...
(gdb) break main
Breakpoint 1 at 0x80483fe:
             file test0208_2.c, line 7.
(gdb) break 9
Breakpoint 2 at 0x804841c:
             file test0208_2.c, line 9.
(gdb) run
Starting program:
 /home/motoki/C-Java2002/Exam/year02/a.out

Breakpoint 1, main () at test0208_2.c:7
7   scanf("%d%d", &x, &y);
(gdb) cont
Continuing.
1 -1

Breakpoint 2, main () at test0208_2.c:9
9   printf("%d + %d = %d\n", x, y, sum);
(gdb) print x
$1 = 1.40129846e-45
(gdb) x/t &x
0x7ffff7a4: 00000000000000000000000000000001
(gdb) print y
$2 = -NaN(0x7fffff)
(gdb) print sum
$3 = -NaN(0x7fffff)
(gdb) print (double)y
$4 = -NaN(0xfffffe0000000)
(gdb) print (double)sum
$5 = -NaN(0xfffffe0000000)
(gdb) cont
Continuing.
0 + 916455424 = -536870912

Program exited with code 033.
(gdb) quit
[motoki@x205a]$
```

□**演習 6.8 (実数型変数の内部の状態を調べる)** 次のプログラムの中の float 型変数 zero, plus1, plus2, plus3, minus1, one10th, inf, nan がコンピュータ内部でどういうビット

列で表されているかGDBを用いて観察してみよ。これらの観察結果はIEEE規格754による実数の表現方法と一致しているか？

```
 1  #include <stdio.h>
 2
 3  int main(void)
 4  {
 5    float zero  = 0.0,   plus1  = 1.0,
 6          plus2 = 2.0,   plus3  = 3.0,
 7          minus1=-1.0,   one10th= 0.1,
 8          inf   = 1e100, nan    = 0.0/0.0;
 9
10    printf("zero  =%f  plus1 =%f\n"
11           "plus2 =%f  plus3 =%f\n"
12           "minus1=%f  one10th=%f\n"
13           "inf =%f     nan =%f\n",
14           zero, plus1, plus2, plus3,
15           minus1, one10th, inf, nan);
16    return 0;
17  }
```

□演習 6.9 (32bitでの整数表現) 演習5.8では、長さが32のビット列 $b_{31}b_{30}b_{29}\cdots b_4b_3b_2b_1b_0$ 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プログラムを演習課題とした。このプログラムの場合、計算結果として得られた2つの整数値は、コンピュータ内部では入力したビット列で表されているはずであるが、これをGDBの `x` コマンドを用いて観察してみよ。

□演習 6.10 (IEEE規格754による実数表現) 演習5.12では、長さが32のビット列 $se_7e_6\cdots e_1e_0d_1d_2d_3\cdots d_{23}$ 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プログラムを作成することを演習課題とした。このプログラムの場合、使用した計算機の実数表現方式がIEEE規格754に従っているなら、計算結果として得られた実数値はコンピュータ内部では入力したビット列で表されているはずであるが、これをGDBの `x` コマンドを用いて調べてみよ。

7 関数 (その2)

- 復習, 自習 4 つの記憶領域のクラス auto, extern, register, static,
- 復習, 自習 暗黙の初期化,
- 復習 関数パラメータの受渡し方法,
- 自習 一次元配列を関数パラメータとして受渡しする方法,
- 関数呼出しの実装,
- 再帰計算 vs. 反復計算

7.1 復習, 自習 4 つの記憶領域のクラス auto, extern, register, static

{ ケリー&ポール 5.8~5.11 節 }

C 言語においては、変数 (, 配列) や関数はデータ型の他に、割り当てられる記憶領域が局所的に使われるかどうか、永続的に使われるかどうか、高速性が要求されるかどうか、といった利用区分 (**記憶域クラス**という) を属性として持つ。 次の 4 種類の記憶域クラスが用意されている。

```
auto
extern
register
static
```

記憶域クラス auto : ブロック (関数本体も含む) に入ると**自動的に**新たに割り付けられ、ブロックから出ると割り付けが解除される記憶領域を指す。従って、この記憶域クラスの変数は**自動変数**と呼ばれ、ブロックの中で局所的なものとなる。 例えば、関数本体の最初に宣言されている変数や配列は暗黙に auto として処理される。

記憶域クラス extern : 全てのブロックの**外で**確保された記憶領域を指す。従って、この記憶域クラスの変数は**外部変数**と呼ばれ、大域的なものとなる。 例えば、関数や、関数の外で宣言された変数 (, 配列) はこのクラスに属する。 [**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る。] また、変数宣言の際にデータ型の前に extern という修飾子を付けると、「ブロックの外、あるいは別ファイルの中でこの変数が確保されている」ことをコンパイラに知らせたことになる。

記憶域クラス register : 変数宣言の際にデータ型の前に register という修飾子を付けると、「この変数領域に高速レジスタを割り付けてほしい」ことをコンパイラに知らせたことになる。レジスタの割り付けが不可能な場合は auto クラスとして扱われる。

記憶域クラス static : ブロックの中で変数宣言する際にデータ型の前に static という修飾子を付けると、この変数領域はブロックに付随した固有の変数として永続的に生き

続ける。 すなわち、ブロックから出てもその記憶領域は保存され、その値は次にブロックに入った時にそのまま引き継がれる。

例 7.1 (extern 宣言) 次のような 2 つのソースファイル func-extern-pt1-Kelley.c と func-extern-pt2-Kelley.c を考える。

```
[motoki@x205a]$ cat -n func-extern-pt1-Kelley.c
     1 #include <stdio.h>
     2
     3 int  f(void);           /* 関数プロトタイプ */
     4
     5 int  a=1, b=2, c=3;     /* 外部変数 */
     6
     7 int main(void)
     8 {
     9   printf("%3d\n", f());
    10   printf("%3d%3d%3d\n", a, b, c);
    11   return 0;
    12 }
[motoki@x205a]$ cat -n func-extern-pt2-Kelley.c
     1 int f(void)
     2 {
     3   extern int  a;
     4   int  b, c;            /* 自動変数 */
     5
     6   a = b = c = 4;
     7   return  a+b+c;
     8 }
```

これらのソースファイルに関して、

コンパイル例 (1)　func-extern-pt2-Kelley.c の 3 行目で宣言されている変数 a は extern 宣言されているので、こちらの処理単位の中で記憶域は確保されない。従って、このブロックもしくはファイルの外で a という外部変数が宣言されていれば func-extern-pt2-Kelley.c のコンパイルは成功する。

```
[motoki@x205a]$ gcc func-extern-pt1-Kelley.c func-extern-pt2-Kelley.c
[motoki@x205a]$ ./a.out
 12
  4  2  3
[motoki@x205a]$
```

コンパイル例 (2)　コンパイルのみ (-c オプション) の gcc コマンドが成功したとしても、リンクの時点で a という外部変数が外で宣言されていなければ、その時点でやはりエラーになる。

```
[motoki@x205a]$ gcc -c func-extern-pt2-Kelley.c
[motoki@x205a]$ gcc func-extern-pt2-Kelley.c
/usr/lib/crt1.o: In function `_start':
```

```
/usr/lib/crt1.o(.text+0x18): undefined reference to ‘main’
/tmp/ccnaNiDG.o: In function ‘f’:
/tmp/ccnaNiDG.o(.text+0x16): undefined reference to ‘a’
/tmp/ccnaNiDG.o(.text+0x1f): undefined reference to ‘a’
collect2: ld returned 1 exit status
[motoki@x205a]$
```

補足：

- エラーメッセージの 1~2 行目 に現れる/usr/lib/crt1.o は初期化を行ったり関数 main を呼び出したりする C のスタートアップルーチンである。/usr/lib/crt1.o の中の関数_start から呼び出すはずの main がどこにもないので、それを警告している。
- エラーメッセージの 3~5 行目 に現れる/tmp/ccnaNiDG.oは与えられたソース func-extern-pt2-Kelley.c をコンパイルして出来たオブジェクトプログラムを一時的に格納したファイルである。このモジュール内の関数 f の中で使われている変数 a の領域が どこにも確保されていないので、それを警告している。
- エラーメッセージの 6 行目 では、リンカ&ローダ ld のプロセスの終了状態値 (exit status) が異常終了を表す 1 であることを言っている。
- オプション -c を指定した場合は、オブジェクトプログラム func-extern-pt2-Kelley.o が生成されるだけでリンクが行われないので、エラーにはならない。

コンパイル例 (3)　func-extern-pt2-Kelley.c で、もし 3 行目の extern int の宣言が無いと (次の func-extern-pt2a-Kelley.c のようになるが、これだと) 外に a という外部変数が宣言されていたとしても 3 行目の a はその外部変数として認識されない。

```
[motoki@x205a]$ cat -n func-extern-pt2a-Kelley.c
     1 int f(void)
     2 {
     3   /* extern int  a; */
     4   int  b, c;            /* 自動変数 */
     5
     6   a = b = c = 4;
     7   return  a+b+c;
     8 }
[motoki@x205a]$ gcc func-extern-pt1-Kelley.c func-extern-pt2a-Kelley.c
func-extern-pt2a-Kelley.c: In function ‘f’:
func-extern-pt2a-Kelley.c:6: ‘a’ undeclared (first use in this function)
func-extern-pt2a-Kelley.c:6: (Each undeclared identifier is reported only once
func-extern-pt2a-Kelley.c:6: for each function it appears in.)
[motoki@x205a]$
```

7.2 復習,自習 暗黙の初期化

{ ケリー&ポール 5.13 節 }

auto 変数
register 変数
} ...暗黙に初期化されることは期待できない。

`extern` 変数
`static` 変数 } ··· C 言語処理系によってゼロに初期化される。

7.3 復習 関数パラメータの受渡し方法 —値呼出し vs. 参照呼出し—

{ ケリー&ポール 1.7 節,5.5 節 }

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のために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、例 7.2 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。

補足：
主記憶内での番地を値とする変数のことを「ポインタ」と呼ぶ教科書もある。

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 7.2 (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。

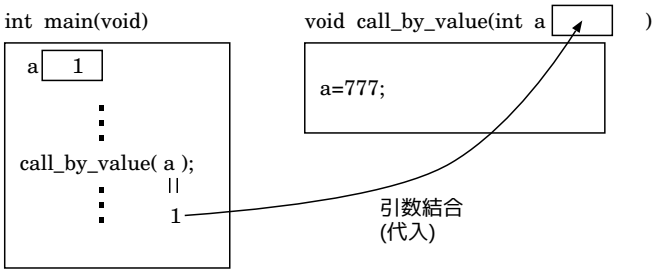
(考え方) プログラムの 6 行目, 14 行目, 18 行目で同じ a という名前の変数が宣言されているが、これらの変数はそれぞれ 5~13 行目, 14~17 行目, 18~21 行目 が有効範囲の別々の変数として扱われる。 C 言語では、

関数引数の結合が値呼出しによって行われる

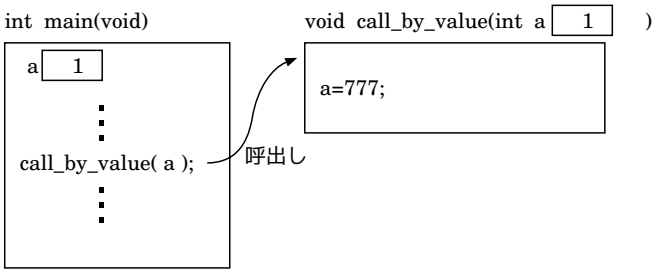
から、

- 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。

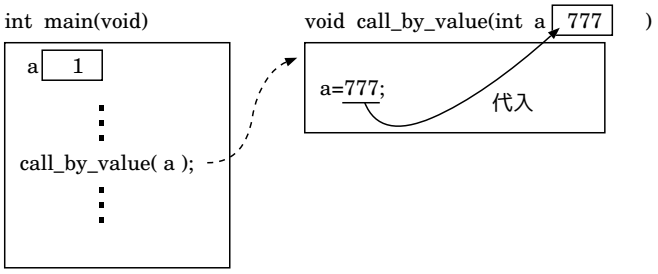
(8 行目, 引数結合)



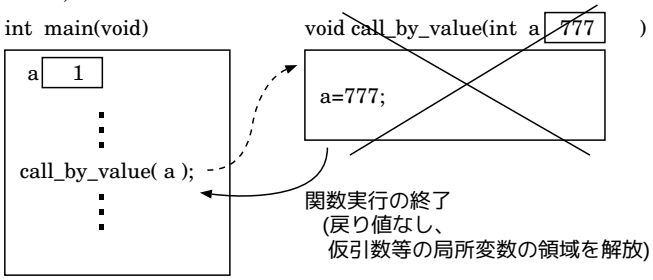
(8 行目, 関数呼び出し)



(16 行目, 実行後)



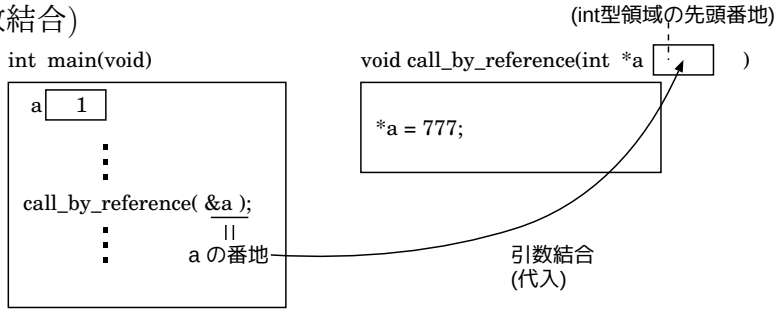
(17 行目, 関数実行終了)



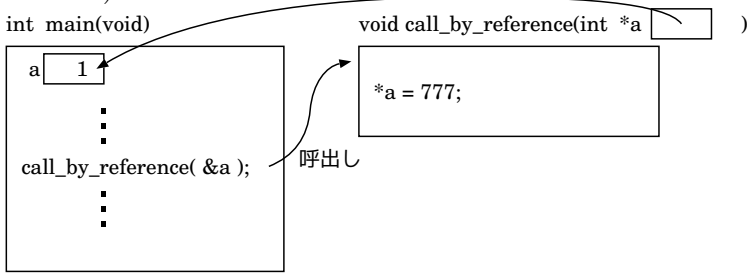
$\Longrightarrow$ 8 行目の関数実行終了後も 6 行目の a の値は 1 のまま変わらない。

- 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。

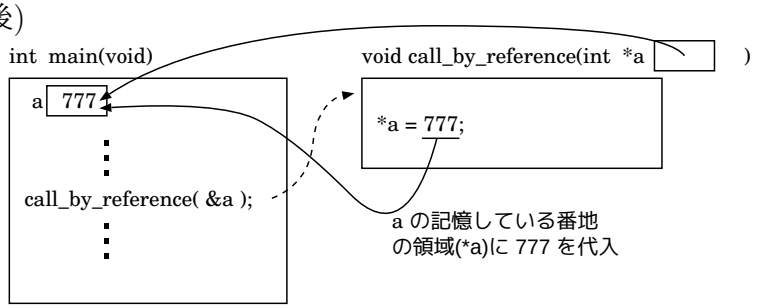
(10 行目, 引数結合)



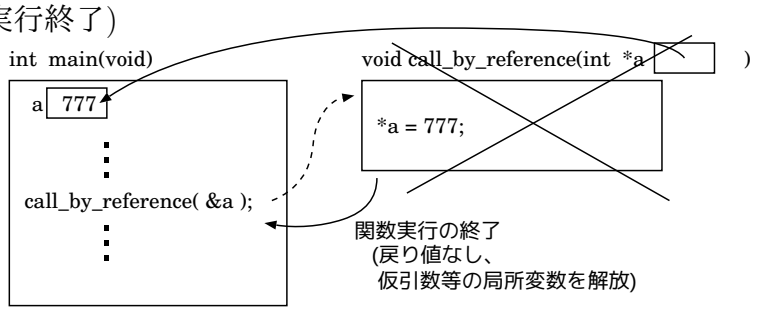
(10 行目, 関数呼び出し)



(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。

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。

7.4 自習 一次元配列を関数パラメータとして受渡しする方法

{ ケリー&ポール 6.6~6.8 節 }

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。

例題 7.3 (Quicksort; 外部配列を使わない版) 例題 4.5 で Quicksort のプログラムを提示した時は、並び替える要素の入った配列 a[SIZE] を外部配列としてこの配列を 5 つの関数 main, set_an_array_random, pretty_print, quicksort, partition に共有させていたが、これではこれら 5 つの関数はこの配列についてのサービスを提供する特殊な関数としての役割しか持たない。 そこで、各々の関数が独立なモジュール (i.e. プログラム部品) として働く様に、例題 4.5 のプログラムを修正せよ。

(考え方) 5 つの関数が共有して使うことになる配列 a[SIZE] は main 関数の中で確保し、残りの関数を呼び出す際にはその配列領域の先頭番地とその配列の大きさを関数パラメータとして受け渡す様にすれば良い。

(プログラミング) 修正例を次に示す。 [下線部は例題 4.5 のプログラムと違う箇所を表す。]

```
[motoki@x205a]$ nl function-quicksort-2.c
     1  /*************************************************************/
     2  /* Quicksort  : 一次元配列を関数パラメータとして受渡しする例  */
     3  /*-------------------------------------                      */
     4  /*     大きさ 100 の配列にランダムに整数を生成し、その配列要素を */
     5  /*     Quicksort アルゴリズムを使って昇順に並べ替えて出力する。 */
     6  /*************************************************************/

     7  #include <stdio.h>
     8  #include <stdlib.h>      /* 乱数発生のライブラリ関数を使うため */
```

```
 9  #define  SIZE   100
10  #define  WIDTH  10
11  #define  TRUE    1

12  void set_an_array_random(int x[], int size);
13  void pretty_print(int x[], int size);
14  void quicksort(int x[], int from, int to);
15  int  partition(int x[], int from, int to);

16  int main(void)
17  {
18    int  a[SIZE], seed;    /* a[SIZE] を外部配列にはしない */

19    printf("Input a random seed (0 - %d):  ", RAND_MAX);
20    scanf("%d", &seed);
21    srand(seed);

22    set_an_array_random(a, SIZE);
23    printf("\nbefore sorting:\n");
24    pretty_print(a, SIZE);

25    quicksort(a, 0, SIZE-1);
26    printf("\nafter sorting:\n");
27    pretty_print(a, SIZE);
28    return 0;
29  }

30  /*-------------------------------------------------------------*/
31  /* 引数で与えられた配列の各要素をランダムに設定する          */
32  /*-----------------------------------------                    */
33  /* (仮引数) x    : int 型配列                                  */
34  /*          size : int 型配列 x の大きさ                       */
35  /* (機能) :  配列要素 x[0]～x[size-1] に 0～999 の間の乱数を  */
36  /*           設定する。                                        */
37  /*-------------------------------------------------------------*/
38  void set_an_array_random(int x[], int size)
39  {
40    int i;

41    for (i=0; i<size; ++i)
42      x[i] = rand() % 1000;
43  }
```

```
44  /*-------------------------------------------------------------*/
45  /* 引数で与えられた配列の要素を順番に全て出力する              */
46  /*-----------------------------------------                    */
47  /* (仮引数) x    : int 型配列                                   */
48  /*          size : int 型配列 x の大きさ                        */
49  /* (機能) :  配列要素 x[0]～x[size-1] の値を順番に全て出力   */
50  /*           する。但し、各々の値は横幅 7 カラムのフィールド  */
51  /*           に出力することにし、また、1 行に WIDTH 個の要素   */
52  /*           を出力する。                                      */
53  /*-------------------------------------------------------------*/
54  void pretty_print(int x[], int size)
55  {
56    int i, count=1;

57    for (i=0; i<size; ++i, ++count) {
58      printf("%7d", x[i]);
59      if (count >= WIDTH) {
60        printf("\n");
61        count = 0;
62      }
63    }
64    if (count > 1)
65      printf("\n");
66  }

67  /*-------------------------------------------------------------*/
68  /* 引数で与えられた配列要素を小さい順に並べ替える              */
69  /*-----------------------------------------                    */
70  /* (仮引数) x    : int 型配列                                   */
71  /*          from : int 型配列 x の添字                          */
72  /*          to   : int 型配列 x の添字                          */
73  /* (関数値) :  なし                                            */
74  /* (機能) :  quicksort アルゴリズムを使って、配列要素         */
75  /*               x[from],x[from+1],x[from+2], ..., x[to]   */
76  /*           を値の小さい順に並べ替える。                      */
77  /*-------------------------------------------------------------*/
78  void quicksort(int x[], int from, int to)
79  {
80    int  pivot_sub;              /* pivot subscript の意 */

81    if (from < to) {
82      pivot_sub = partition(x, from, to);       /* 分割操作 */
83      quicksort(x, from, pivot_sub - 1);
```

```
 84      quicksort(x, pivot_sub + 1, to);
 85    }
 86  }

 87  /*-------------------------------------------------------------*/
 88  /* 引数で与えられた配列の部分列に quicksort の分割操作を施す   */
 89  /*                                          (quicksort の関数) */
 90  /*-----------------------------------------                    */
 91  /* (仮引数) x    : int 型配列                                  */
 92  /*          from : int 型配列 x の添字                         */
 93  /*          to   : int 型配列 x の添字                         */
 94  /* (関数値)  :  分割操作によって得られた枢軸要素の添字番号   */
 95  /*              (以下の「(機能)」の項で出て来る pivot_sub)    */
 96  /* (機能) :  x[from]～x[to] を並べ替えて                       */
 97  /*           max{x[from],...,x[pivot_sub-1]} <= x[pivot_sub]*/
 98  /*           x[pivot_sub] < min{x[pivot_sub+1],...,x[to]}   */
 99  /*           となるようにする。                               */
100  /*-------------------------------------------------------------*/
101  int  partition(int x[], int from, int to)
102  {
103    int pivot;

104    pivot = x[from];      /* 最初の要素を枢軸要素に選ぶ。 */
105    while (TRUE) {        /* 工夫の余地あり。             */
106      for ( ; from<to && x[to]>pivot; --to)
107        ;
108      if (from == to) {
109        x[from] = pivot;
110        return from;
111      }
112      x[from++] = x[to];

113      for ( ; from<to && x[from]<=pivot; ++from)
114        ;
115      if (from == to) {
116        x[to] = pivot;
117        return to;
118      }
119      x[to--] = x[from];
120    }
121  }
```

ここで、

- プログラムの 22 行目,24～25 行目,27 行目,82～84 行目が、配列を引数にして関数を呼

んでいる部分である。この様に、配列を関数に引き渡したい時には、単に配列名を実引数として与えればよい。[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るから、これで、配列の先頭番地が関数に引き渡されることになります。]

- プログラムの 12~15 行目,38 行目,54 行目,78 行目,101 行目に現われる `int x[]` の部分は、仮引数 `x` が int 型一次元配列の先頭番地と結合することを宣言している。[但し、プログラマ側が「int 型一次元配列の先頭番地」という意図で宣言したとしても、コンパイラ側はこれを「int 型領域へのポインタ」と理解するだけである。] この部分は、

  ```
  int *x
  ```

  あるいは

  ```
  int x[ 配列の大きさ ]
  ```

  と書くことも出来る。
- プログラム 39~43 行目,55~66 行目,79~86 行目,102~121 行目の関数本体の中では、仮引数の配列はこれまでと全く同じ様に使うことが出来ます。

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 7.4 (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`]`$= 2^{49-k}$ $(k = 0\sim49)$

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]+`$\cdots$`+v[49]`,
`v[40]+v[41]+v[42]+`$\cdots$`+v[49]`,
`v[20]+v[21]+v[22]+`$\cdots$`+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] $\longleftarrow$ 1,
v[48] $\longleftarrow$ v[49]$\times$2,
v[47] $\longleftarrow$ v[48]$\times$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-Kelley.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-Kelley.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、

- 4.5 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。

7.5 関数呼出しの実装

計算機内部でどのように関数呼出しが実現されているか：

- それぞれの関数の計算に必要な作業用領域を確保するために、push-down スタックを一つ用意する。
- 関数が呼び出されるたびに、
  ① 呼び出し元の戻り番地を入れておく領域、仮引数の変数、新しい自動変数のための領域を必要なだけスタック上に確保し、そこに、呼び出し元の戻り番地、実引数の値等を初期設定する。
  ② 呼び出し先の関数に制御を渡す。(図 4)

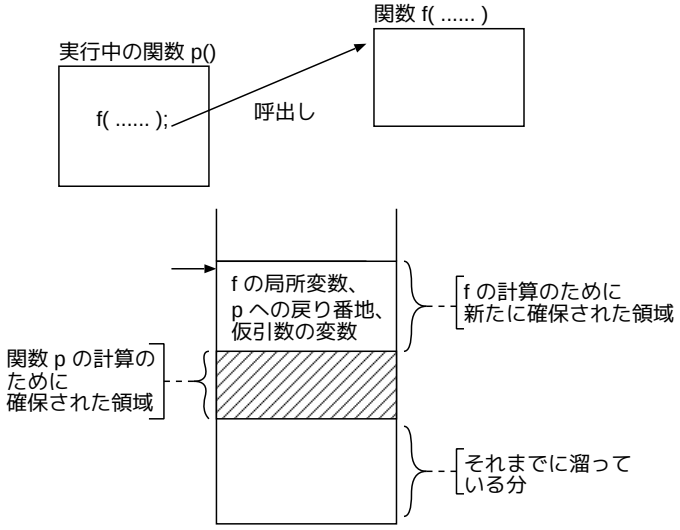


図 4: 関数呼び出しの際の様子

- 関数実行が終了するたびに、
  ① 関数値と呼び出し元への戻り番地をどこかに保持した上で、呼び出し先の関数の計算のために確保した領域を解放する。
  ② 呼び出し元の関数の計算を再開する。

例 7.5 (バッファ・オーバーフロー) 次のプログラムとその実行結果を考える。

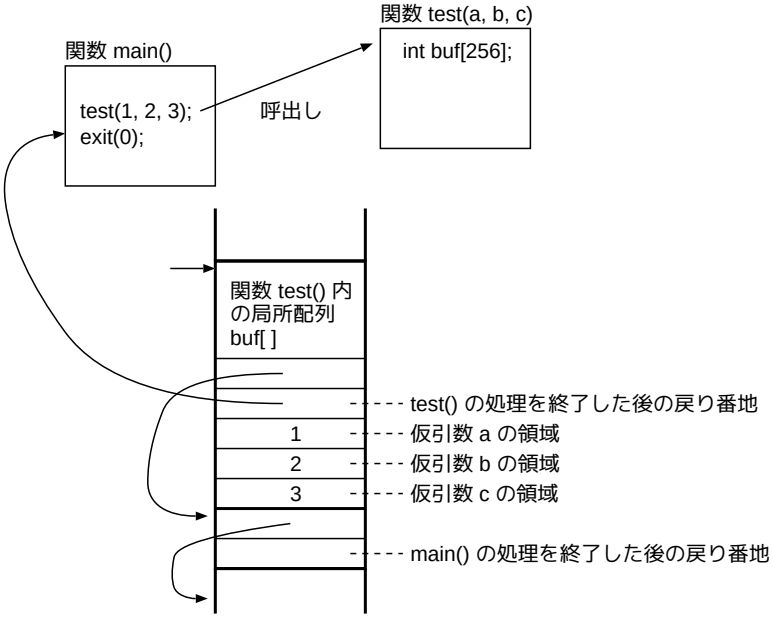
```
[motoki@x205a]$ cat -n buffer-overflow.c
     1 /*-------------------------------------------------------*/
     2 /* 関数呼出しの際に引数の値や局所変数等が                 */
     3 /*       共通のスタックに積まれることを理解するための例   */
     4 /*-------------------------------------------------------*/
     5 /* これらのことが悪用されてコンピュータが不正侵入される */
     6 /* こともある。(バッファオーバーフロー攻撃)               */
     7 /*-------------------------------------------------------*/
     8 #include <stdio.h>
     9 #include <stdlib.h>
    10
    11 void test(int a, int b, int c);
    12
    13 int main(void)
    14 {
    15   test(1, 2, 3);
    16   exit(EXIT_SUCCESS);
    17
    18   printf("\n ここには来ないはずだが、、、。\n");
    19   return 0;
    20 }
    21
    22 void test(int a, int b, int c)
    23 {
    24   int buf[256];
    25
    26   printf("(関数 test 内) a= %d\n", a);
    27   buf[258] = 999;
    28   printf("(関数 test 内) a= %d\n", a);
    29
    30   buf[257] += 16;
    31 }
[motoki@x205a]$ gcc buffer-overflow.c
[motoki@x205a]$ ./a.out
(関数 test 内) a= 1
(関数 test 内) a= 999

ここには来ないはずだが、、、。
[motoki@x205a]$
```

このプログラムの場合、27 行目と 30 行目では確保したメモリ外への書き込みを行っている。この部分を見当外れの代入文と見て無視すると、プログラムは次の様な実行結果を出してくれるはずである。

```
(関数 test 内) a= 1
```

（関数 test 内）a= 1

しかし、上で示した実行結果では、この期待に反して 28 行目の printf() で出力される値が 999 になっているし、実行されるはずのない 18 行目が実行されている。では何故こういう実行結果になったのかというと、それは関数呼び出しの実装方法に由来する。実は、上のプログラムで関数 test() が呼び出された直後には、関数呼び出しを実装するために用意されている push-down スタックの状態は次の様になっている。



この図を見ると、プログラム 27 行目で不当に参照している buf[258] という領域は関数 test() の第 1 仮引数 a のために用意された領域と重なり、30 行目で不当に参照している buf[257] という領域は関数 main() への戻り番地を保持する領域と重なっていることが分かる。それゆえ、

- 27 行目の代入文で 仮引数 a への 999 という値の代入が行われ、それが 28 行目の printf() で出力される。 また、
- 30 行目の代入文で main() 関数への戻り番地が 16 バイトだけ後にずらされ、その結果、test() 関数を終了すると次に 16 行目の exit() ではなく 18 行目の printf() が実行されてしまう。

バッファ・オーバーフロー攻撃：

もし、外部へのサービスを行っているサーバプログラムにユーザ名入力の場面があり、そこからサーバプログラムの局所変数領域内に実行コード (e.g. シェル実行) が組み込まれ、「関数処理終了後の戻り番地」を保持する領域にこの実行コードの番地がセットされてしまうと、関数処理終了と同時に組み込まれたコードが (**root 権限で**) 実行されてしまう。また、setuid ビットが設定され所有者が root のコマンド (e.g.passwd コマンド) の引数に、同様のバッファオーバーフローを引き起こすバイト列が与えられると、悪意のあるコードが root 権限で実行されてしまう。そういった理由で、関数呼び出しのための上記の機構はインターネット等を通じたコンピュータ/サーバへの攻撃の足掛かりにされることもある。一旦不正侵入されると、root 権限で任意のプログラムが実行され得るので甚大な被害を被る危険性も高い。

Buffer Overflow のセキュリティホールに繋がる関数としては C 言語では次の様なものを挙げることが出来る。

```
gets(), strcpy(), strcat(), sprintf(), vsprintf(),
scanf(), vscanf(), .....
```

7.6 再帰計算 vs. 反復計算

{ ケリー&ポール 5.15 節 }

- 再帰計算と同等のことは、無理なく反復計算で記述できることが多い。
- 再帰計算の方が、変数も少なくて済み、分かり易いプログラムになることが多い。
- 再帰計算を行うと、関数呼出しが多くなるので、その分計算時間も記憶領域も多く必要になる。
- 再帰計算によって、異常に非効率な計算が起こることもある。

例 7.6 (Fibonacci 数列の再帰計算) 次のプログラムの場合、再帰計算によって異常に非効率な計算になる。

```
[motoki@x205a]$ nl func-bad-recursion-fibo-Kelley.c
     1 #include <stdio.h>

     2 int  fibonacci(int n);

     3 int main(void)
     4 {
     5   int  i;

     6   scanf("%d", &i);
     7   printf("fibonacci(%d) = %d\n", i, fibonacci(i));
     8   return 0;
     9 }

    10 int fibonacci(int n)
    11 {
    12   printf("called fibonacci(%d)\n", n);
    13          /* どういう計算が行われるかを観察するために */
    14   if (n <= 1)
    15     return n;
    16   else
    17     return fibonacci(n-1)+fibonacci(n-2);
    18 }
[motoki@x205a]$ gcc func-bad-recursion-fibo-Kelley.c
[motoki@x205a]$ ./a.out
6
called fibonacci(6)
called fibonacci(5)
called fibonacci(4)
called fibonacci(3)
called fibonacci(2)
called fibonacci(1)
```

```
called fibonacci(0)
called fibonacci(1)
called fibonacci(2)
called fibonacci(1)
called fibonacci(0)
called fibonacci(3)
called fibonacci(2)
called fibonacci(1)
called fibonacci(0)
called fibonacci(1)
called fibonacci(4)
called fibonacci(3)
called fibonacci(2)
called fibonacci(1)
called fibonacci(0)
called fibonacci(1)
called fibonacci(2)
called fibonacci(1)
called fibonacci(0)
fibonacci(6) = 8
[motoki@x205a]$
```

演習問題

□演習 7.1 (実習室における初期化の実際) 実習室の C 言語処理系において、外部変数がゼロに初期化されていることを確かめよ。 また、自動変数が実際に初期化がされていないかどうか調べよ。

□演習 7.2 (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);
```

(右上へ続く ↗)

(↗ 左下からの続き。)

```
  a = 0;
  a = sub2(a, a);
  printf("(sub2) %d\n", a);

  a = 0;
  sub3(&a, &a);
  printf("(sub3) %d\n", a);
  return 0;
}
```

(次ページへ続く ↗)

(↗ 前ページからの続き。)

```
void sub1(int x, int y)
{
  x++;
  y++;
}

int sub2(int x, int y)
{
  x++;
  y++;
  return y;
}
```

(右上へ続く ↗)

(↗ 左下からの続き。)

```
void sub3(int *x, int *y)
{
  (*x)++;
  (*y)++;
}
```

□演習 7.3 (名前の有効範囲, 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]$ cat test0508_1b.c
#include <stdio.h>

int sub1(int, int);
int sub2(int *, int *);
int c=333;

int main(void)
{
  int a;

  a = 0;
  printf("sub1(a,a)=%d\n",
         sub1(a, a));
  printf("a=%d\n", a);

  a = 0;
  printf("sub2(&a,&a)=%d\n",
         sub2(&a, &a));
  printf("a=%d\n", a);
  return 0;
}
```

(右上へ続く ↗)

(↗ 左下からの続き。)

```
int sub1(int x, int y)
{
  x++;
  y++;
  return c;
}
[motoki@x205a]$ cat test0508_1c.c
static int c=666;

int sub2(int *x, int *y)
{
  (*x)++;
  (*y)++;
  return c;
}
[motoki@x205a]$ gcc test0508_1b.c
                          test0508_1c.c
[motoki@x205a]$ ./a.out
[                        ]
[motoki@x205a]$
```

□演習 7.4 (フィボナッチ数列の再帰計算) 例7.6のプログラムで、fibonacci(4) の実行によって関数 fibonacci が全部で何回呼び出されることになるか？ 一般に fibonacci(x) の実行の場合はどうか？

8 モジュール化について

- モジュール化, 段階的詳細化,
- 静的外部変数, 静的関数

8.1 モジュール化

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。

例題 8.1 (Napier 数 e の 1000 桁計算) 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 1000 桁まで計算して出力する C プログラムを作成せよ。

(考え方) スターリング (Stirling) の公式

$$m! \approx \sqrt{2\pi}\, m^{m+\frac{1}{2}} e^{-m} \quad (m \longrightarrow \infty)$$

より

$$451! \approx \sqrt{2\pi}\, 451^{451.5} e^{-451} \approx 7.8 \times 10^{1002}$$

が得られることに注目しよう。これより、e の 1001 桁計算のためには近似式

$$\begin{aligned} e &\approx \sum_{i=0}^{450} \frac{1}{i!} \\ &= (((\cdots((1 \cdot \frac{1}{450} + 1)\frac{1}{449} + 1)\cdots)\frac{1}{3} + 1)\frac{1}{2} + 1)\frac{1}{1} + 1 \end{aligned}$$

による計算を正確に行なえばよいことが分かる。この計算は、実数を正確に記憶できる変数 v_{ideal} があれば、次のように行うことが出来る。

```
v_ideal ← 1 ;
for k ← 450 step −1 until 1 do
    v_ideal ← v_ideal/k + 1 ;
v_ideal の値を出力 ;
```

しかし、実数を正確に記憶できる変数などあり得ないので、その代わりに小数点以下 1000 桁の 10 進小数

$c_3c_2c_1c_0.d_1d_2d_3d_4d_5d_6d_7d_8 \cdots\cdots d_{997}d_{998}d_{999}d_{1000}$
(各 c_i, d_iは 0∼9 の整数を表す。)

を

$$\underbrace{c_3c_2c_1c_0}_{e[0]\text{ に記憶}} . \underbrace{d_1d_2d_3d_4}_{e[1]\text{ に記憶}} \underbrace{d_5d_6d_7d_8}_{e[2]\text{ に記憶}} \cdots\cdots \underbrace{d_{997}d_{998}d_{999}d_{1000}}_{e[250]\text{ に記憶}}$$

という風に記憶する整数型 (4 バイト以上) 配列 e を用意する。そして、上で示した計算手順に現われる処理の基本単位

① 配列 e ← 1　　(すなわち、配列 e の表す 10 進小数を 1 に設定する処理),

② 配列 e ← 配列 e の保持する値 / k ,

③ 配列 e ← 配列 e の保持する値 + 1 ,

④ 配列 e の保持する値 の出力

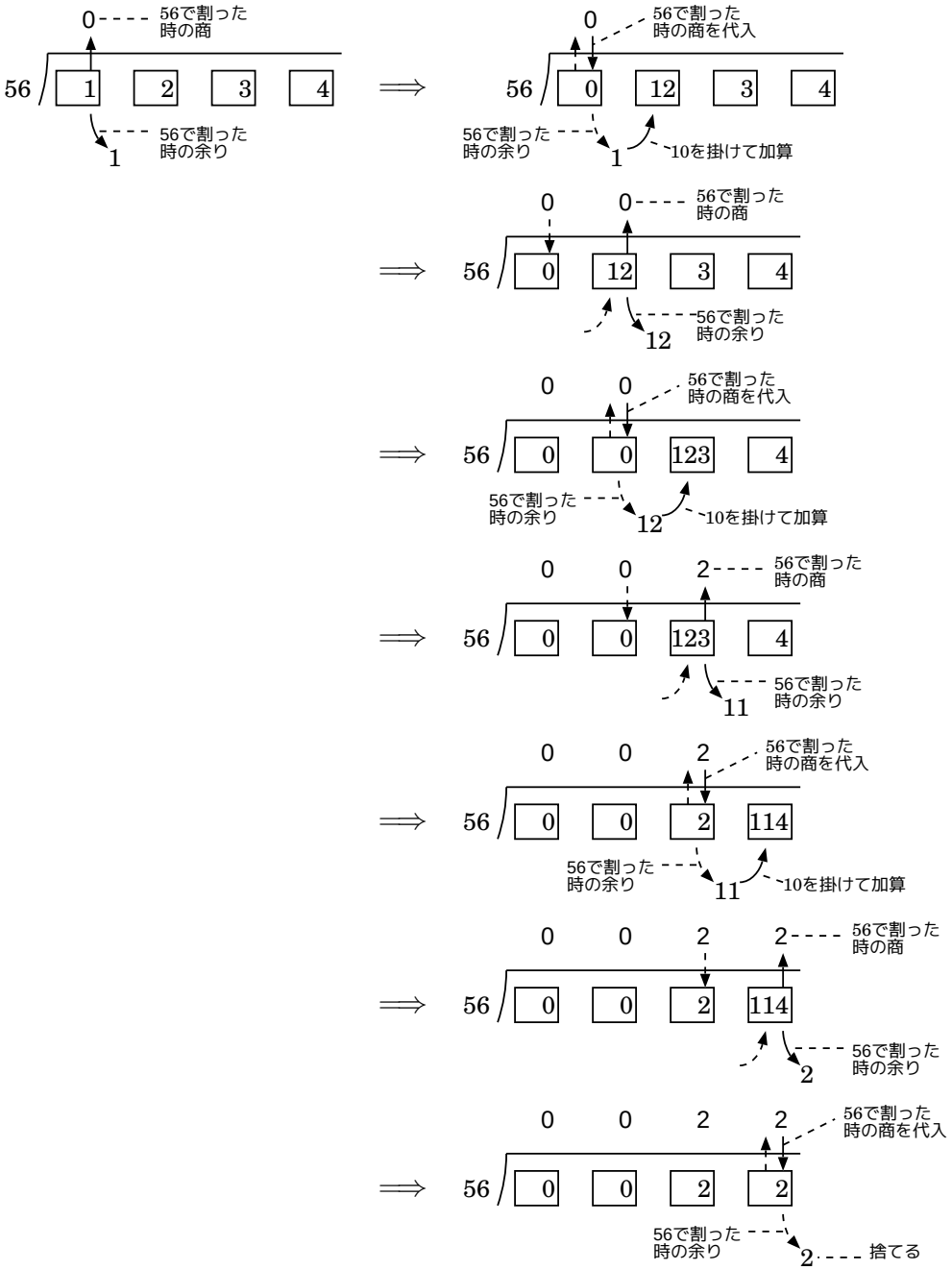
を一般化したものを関数モジュールとして用意することにすれば、これらの関数モジュールを用いて上で示した**計算手順を見通し良く書き表す**ことが出来る。

> 補足：
> このアルゴリズムは、結局は、10000 進法のそれぞれの桁を記憶するために配列要素 e[0]∼e[250] を使い、10000 進法で Napier 数を計算するものである。

基本処理「配列 e ← 配列 e の保持する値 / k」の実装：小学校時に行った 10 進の割り算の考えを拡張すればよい。例えば、配列要素に 10 進の 1 桁ずつを保持することにして

```
      22
56 ) 1234
     112
     ---
      114
      112
      ---
        2
```

という割り算をコンピュータ内で行う場合は、次の様に処理を進めることができる。



この様な処理を拡張して配列要素に 10 進の 4 桁ずつ保持する様にすれば良い。

(プログラミング)　例えば次のような C プログラム／実行の様子が得られる。

```
[motoki@x205a]$ nl modules-napier.c
     1 /*************************************************************/
     2 /* Napier数 e の1000桁計算                                   */
     3 /*-------------------------------------                      */
     4 /*     大きさ251の配列 e に小数点以下1000桁の10進小数を      */
     5 /*         e[0]=整数部の数,                                  */
     6 /*         e[1]=小数点以下1〜4桁目の数,                      */
     7 /*         e[2]=小数点以下5〜8桁目の数,                      */
     8 /*             ......                                        */
```

```
 9 /*        e[250]=小数点以下 996〜1000 桁目の数                    */
10 /*     という風に保存して高精度計算を行う                          */
11 /*************************************************************/

12 #include  <stdio.h>
13 #define   LIMIT   7

14 void set_array_to_keep_integer(int a[], int size, int init_value);
15 void divide_array_value_by_integer(int a[], int size, int divisor);
16 void add_integer_to_array(int a[], int size, int value);
17 void print_array_value(int a[], int size, char *padding);

18 int main(void)
19 {
20   int  e[251], k;

21   set_array_to_keep_integer(e, 251, 1);        /* e <-- 1 */

22   for (k=450; k>0; --k) {
23     divide_array_value_by_integer(e, 251, k);  /* e <-- e/k */
24     add_integer_to_array(e, 251, 1);           /* e <-- e+1 */
25   }

26   printf("e =");
27   print_array_value(e, 251, "   ");           /* 結果の出力 */
28   return 0;
29 }

30 /*-----------------------------------------------------------*/
31 /* 配列全体で表す値 (10 進小数) を初期設定する                 */
32 /*-----------------------------------------                  */
33 /* (仮引数)   a    : int 型配列                               */
34 /*            size : int 型配列 a の大きさ                    */
35 /*      init_value : 初期設定したい値 (int 型)                */
36 /* (関数値)   : なし                                         */
37 /* (機能) :  配列 a 全体の表す値が init_value になる様にする*/
38 /*-----------------------------------------------------------*/
39 void set_array_to_keep_integer(int a[], int size, int init_value)
40 {
41   int i;
42
43   a[0] = init_value;
44   for (i=1; i<size; ++i)
```

```
45     a[i] = 0;
46 }

47 /*-------------------------------------------------------------*/
48 /* 配列全体で表す値 (10 進小数) を整数で割る                   */
49 /*-------------------------------------------                  */
50 /* (仮引数)   a    : int 型配列                                */
51 /*            size : int 型配列 a の大きさ                     */
52 /*         divisor : 除数                                      */
53 /* (関数値)   : なし                                           */
54 /* (機能) :  配列 a 全体の表す値を整数 divisor で割る          */
55 /*-------------------------------------------------------------*/
56 void divide_array_value_by_integer(int a[], int size, int divisor)
57 {
58   int i;
59
60   for (i=0; i<size-1; ++i) {
61     a[i+1] += (a[i] % divisor) * 10000;
62     a[i]   /= divisor;
63   }
64   a[size-1] /= divisor;
65 }

66 /*-------------------------------------------------------------*/
67 /* 配列全体で表す値 (10 進小数) に整数を加算する               */
68 /*-------------------------------------------                  */
69 /* (仮引数)   a    : int 型配列                                */
70 /*            size : int 型配列 a の大きさ                     */
71 /*          addend : 加数                                      */
72 /* (関数値)   : なし                                           */
73 /* (機能) :  配列 a 全体の表す値に整数 addend を加算する       */
74 /*-------------------------------------------------------------*/
75 void add_integer_to_array(int a[], int size, int addend)
76 {
77   a[0] += addend;
78 }

79 /*-------------------------------------------------------------*/
80 /* 配列全体で表す値 (10 進小数) を出力する                     */
81 /*-------------------------------------------                  */
82 /* (仮引数)   a    : int 型配列                                */
83 /*            size : int 型配列 a の大きさ                     */
84 /*         padding : 2 行目以降の左端に必ず出力する文字列      */
```

```
 85 /*（関数値）   : なし                                              */
 86 /*（機能）:  配列 a 全体の表す 10 進小数値を出力する。その際、*/
 87 /*             ＊小数点以下は 8*LIMIT 桁毎に改行する，          */
 88 /*             ＊２行目以降の出力においては左端に必ず文字列  */
 89 /*               padding を埋める。                              */
 90 /*--------------------------------------------------------------*/
 91 void print_array_value(int a[], int size, char *padding)
 92 {
 93   int i, count;
 94
 95   printf("%2d.", a[0]);
 96   count = 1;
 97   for (i=1; i<size; i+=2, ++count) {
 98     printf("%04d%04d ", a[i], a[i+1]);
 99     if (count >= LIMIT) {
100       printf("\n%s   ", padding);
101       count = 0;
102     }
103   }
104   printf("\n");
105 }
[motoki@x205a]$ gcc modules-napier.c
[motoki@x205a]$ ./a.out
e = 2.71828182 84590452 35360287 47135266 24977572 47093699 95957496
      69676277 24076630 35354759 45713821 78525166 42742746 63919320
      03059921 81741359 66290435 72900334 29526059 56307381 32328627
      94349076 32338298 80753195 25101901 15738341 87930702 15408914
      99348841 67509244 76146066 80822648 00168477 41185374 23454424
      37107539 07774499 20695517 02761838 60626133 13845830 00752044
      93382656 02976067 37113200 70932870 91274437 47047230 69697720
      93101416 92836819 02551510 86574637 72111252 38978442 50569536
      96770785 44996996 79468644 54905987 93163688 92300987 93127736
      17821542 49992295 76351482 20826989 51936680 33182528 86939849
      64651058 20939239 82948879 33203625 09443117 30123819 70684161
      40397019 83767932 06832823 76464804 29531180 23287825 09819455
      81530175 67173613 32069811 25099618 18815930 41690351 59888851
      93458072 73866738 58942287 92284998 92086805 82574927 96104841
      98444363 46324496 84875602 33624827 04197862 32090021 60990235
      30436994 18491463 14093431 73814364 05462531 52096183 69088870
      70167683 96424378 14059271 45635490 61303107 20851038 37505101
      15747704 17189861 06873969 65521267 15468895 70350354
[motoki@x205a]$
```

ここで、

- プログラム 98 行目 に現われる %04d は、**詰め込み文字**を (空白ではなく)0 にして大きさ 4 のフィールドに整数を右詰めで出力することを指示する。"%04d"の中の文字 0 がフラグになっている。

8.2 モジュール化の方法 ——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。

例題 8.2 (自習 与えられた月のカレンダーを出力する) 2 つの正整数 y と $m(\leq 12)$ を入力して西暦 y 年 m 月のカレンダーを次の様な形で出力する C プログラムを作成せよ。

```
      2004 May
Sun Mon Tue Wed Thu Fri Sat
                        1
  2   3   4   5   6   7   8
  9  10  11  12  13  14  15
 16  17  18  19  20  21  22
 23  24  25  26  27  28  29
 30  31
```

(考え方) この問題に対して、次のように段階的詳細化の考え方を適用できる。

(第 1 段階) 全体の処理の流れの記述

3 つの関数

$$\mathrm{name}(m) = \begin{cases} \text{"January"} & \text{if } m = 1 \\ \text{"February"} & \text{if } m = 2 \\ \quad\vdots & \\ \text{"December"} & \text{if } m = 12 \end{cases}$$

$\mathrm{number\_of\_days}(y, m) =$ 西暦 y 年 m 月の日数

$$\text{what\_day}(y,m) = \text{西暦 } y \text{ 年 } m \text{ 月 1 日の曜日を表す指標}$$
$$= \begin{cases} 0 & \text{if 西暦 } y \text{ 年 } m \text{ 月 1 日が日曜日} \\ 1 & \text{if 西暦 } y \text{ 年 } m \text{ 月 1 日が月曜日} \\ 2 & \text{if 西暦 } y \text{ 年 } m \text{ 月 1 日が火曜日} \\ \vdots & \\ 6 & \text{if 西暦y年m月 1 日が土曜日} \end{cases}$$

の存在を仮定すれば、与えられた正整数 y, m に対して 西暦 y 年 m 月のカレンダーを求められた形に出力するアルゴリズムとしては次のようなものが考えられる。

```
年数、月数を表す正整数を入力して各々 year, month と名付ける ;
                { プロンプトの出力、入力データのチェックなどは適宜行なう。 }
1 行目の表題のために整数 year と月名 name(month) を出力 ;
曜日の見出し ("␣Sun␣Mon␣...") を出力 ;
what_day(year,month) 個の "␣␣␣␣"を出力 ;
column ← what_day(year,month)+1 ;
for day ← 1 until number_of_days(year,month) do
    begin
        日付 day を出力 ;
        if column=7 then 改行
                    else column ← 0 ;
        day ← day+1 ;
        column ← column+1
    end ;
if column>1 then 改行
```

(第 2 段階) 関数 name の詳細化

関数 name(m) は次の 2 次元 `char` 型配列 name$[i][j]$ で表せる。

$\longrightarrow j$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
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'						

(第 3 段階) 関数 number_of_days の詳細化

m 月の標準日数を表す関数

$$\text{n\_days}(m) = \begin{cases} 31 & \text{if } m = 1 \\ 28 & \text{if } m = 2 \\ 31 & \text{if } m = 3 \\ \vdots & \\ 31 & \text{if } m = 12 \end{cases}$$

と「西暦 y 年 m 月がうるう月かどうか」を表す関数

$$\mathrm{leap}(y,m) = \begin{cases} 1 & \text{if 暦 } y \text{ 年がうるう年で } m=2 \\ 0 & \text{otherwise} \end{cases}$$

を用いて

$$\mathrm{number\_of\_days}(y,m) = \mathrm{n\_days}(m) + \mathrm{leap}(y,m)$$

と表せる。従って、第1段階で示した全体の処理の流れの中で、$\mathrm{number\_of\_days}(y,m)$ の代わりに $\mathrm{n\_days}(m) + \mathrm{leap}(y,m)$ という式を用いれば良い。

(第4段階) 関数 n_days の詳細化

関数 $\mathrm{n\_days}(m)$ は `int` 型配列で表せる。

(第5段階) 関数 leap の詳細化

グレゴリオ歴によると「西暦年数が4で割り切れ100で割り切れない年、または400で割り切れる年を閏年とする」ことになっている。それゆえ、$\mathrm{leap}(y,m)$ の計算は次のように行なうことが出来る。[ここで、$\mathrm{mod}(y,i)$ は y を i で割った余りを表す。]

```
if m = 2∧ ((mod(y,4)=0 ∧ mod(y,4) ≠ 0) ∨ mod(y,400)=0)
    then return 1
    else return 0
```

(第6段階) 関数 what_day の詳細化

1月1日から同年 $(m-1)$ 月末日までの標準日数を表す関数

$$\mathrm{total\_days}(m) = \textstyle\sum_{i=1}^{m-1} \mathrm{n\_days}(i)$$

を用いると、西暦1年1月1日から西暦 y 年 m 月1日までの日数は

(西暦1年1月1日から西暦 $(y-1)$ 年12月31日までの日数)
+ (西暦 y 年1月1日から西暦 y 年 m 月1日までの日数)

$$= 365(y-1)+\lfloor\tfrac{y-1}{4}\rfloor-\lfloor\tfrac{y-1}{100}\rfloor+\lfloor\tfrac{y-1}{400}\rfloor+\mathrm{total\_days}(m)+\mathrm{leap}(y,\min\{m-1,2\})+1$$

となる。 それゆえ、定数 c をうまく選べば、

$$\begin{aligned}
&\mathrm{what_day}(y,m)\\
&= \mathrm{mod}(\text{西暦1年1月1日から西暦 } y \text{ 年 } m \text{ 月1日までの日数} + c, 7)\\
&= \mathrm{mod}((7\times 53+1)(y-1) + \lfloor\tfrac{y-1}{4}\rfloor - \lfloor\tfrac{y-1}{100}\rfloor + \lfloor\tfrac{y-1}{400}\rfloor\\
&\qquad +\mathrm{total_days}(m) + \mathrm{leap}(y,\min\{m-1,2\}) + 1 + c, 7)\\
&= \mathrm{mod}((y-1) + \lfloor\tfrac{y-1}{4}\rfloor - \lfloor\tfrac{y-1}{100}\rfloor + \lfloor\tfrac{y-1}{400}\rfloor\\
&\qquad +\mathrm{total_days}(m) + \mathrm{leap}(y,\min\{m-1,2\}) + 1 + c, 7)
\end{aligned}$$

となるはずである。 ここで、what_day(1999,4)=4 であるから、

$$\begin{aligned}
\mathrm{what_day}(1999,4) &= \mathrm{mod}(1998+499-19+90+0+1+c, 7)\\
&= \mathrm{mod}(4+\mathrm{c},7)
\end{aligned}$$

ということと合わせて、例えば $c=0$ と選べばよい。以上のことから、関数 $\mathrm{what\_day}(y,m)$ の計算は次のように詳細化できる。

```
return mod( (y − 1) + ⌊(y−1)/4⌋ − ⌊(y−1)/100⌋ + ⌊(y−1)/400⌋
                    +total_days(m) + leap(y, min{m − 1, 2}) + 1,   7 )
```

(第7段階) 関数 total_days の詳細化

関数 $\mathrm{total\_days}(m)$ は `int` 型配列で表せる。

(プログラミング) 以上の段階的詳細化によって明らかになったアルゴリズムは、次のよ

うにC言語で記述してコンパイル／実行を行なうことが出来る。

```
[motoki@x205a]$ cat modules-calendar.c
#include <stdio.h>
#include <stdlib.h>

#define  DUMMY     0
#define  min(A,B)  ((A) < (B) ? (A) : (B))

int what_day(int year, int month);   /* ○年○月１日の曜日を計算          */
int leap(int year, int month);       /* ○年○月がうるう月かどうかを判定 */

/*****************************************/
/* ○月○日のカレンダーを表示する         */
/*---------------------------------------*/
/*    (入力) 正整数 y と m (<=12)        */
/*    (出力) 西暦 y 年 m 月のカレンダー  */
/*****************************************/
int main(void)
{
  int year, month, day, column, k, limit;
  int n_days[13]={
          DUMMY,                     /* n_days[k]           */
          31, 28, 31, 30, 31, 30,    /* = k月の標準的な日数 */
          31, 31, 30, 31, 30, 31
      };
  char name[][15]={                          /* name[k]           */
          "Illegal month",                   /* = k番目の月の名前 */
          "January", "February", "March",
          "April", "May", "June",
          "July", "August", "September",
          "October", "November", "December"
        };

  printf("Please type in year and month numbers.\n");
  if ( scanf("%d %d", &year, &month) != 2
                    || year<1 || month <1 || month>12 ){
    printf("\nIllegal input!\n");
    printf("   ==> Input a positive integer and an integer in [1,12].\n");
    exit(EXIT_FAILURE);
  }

  printf("         %d %s\n", year, name[month]);
  printf(" Sun Mon Tue Wed Thu Fri Sat\n");
```

```
  limit = what_day(year,month);
  for ( column=1 ; column <= limit ; column++ )
    printf("    ");

  limit = n_days[month]+leap(year,month);
  for ( day=1 ; day <= limit ; day++, column++ ){
    printf("%4d", day);
    if (column == 7 ){
      printf("\n");
      column=0;
    }
  }

  if (column>1)
    printf("\n");
  return 0;
}

/************************************************/
/* ○年○月 1 日の曜日を計算して返す             */
/*----------------------------------------    */
/*   (仮引数)  y : 西暦年 (正整数)              */
/*             m : 月の番号 (1 以上 12 以下の整数) */
/*   (関数値)  0  if 西暦 y 年 m 月 1 日が日曜日 */
/*             1  if          〃          月曜日 */
/*                  ……                    */
/*             7  if          〃          土曜日 */
/************************************************/
int what_day(int y, int m)
{
  int total_days[13]={
          DUMMY,                            /* total_days[k]         */
            0,  31,  59,  90, 120, 151,     /* = n_days[1]+n_days[2] */
          181, 212, 243, 273, 304, 334      /*      + ... +n_days[k] */
      };

  return (y+(y-1)/4-(y-1)/100+(y-1)/400
            +total_days[m]+leap(y,min(m-1,2))) % 7;
}

/************************************************/
/* ○年○月がうるう月かどうかを判定してして返す     */
/*----------------------------------------      */
```

```
/*   (仮引数)  y : 西暦年 (正整数)                    */
/*             m : 月の番号 (1 以上 12 以下の整数)     */
/*   (関数値)  0  if 西暦 y 年 m 月がうるう月でない */
/*             1  if          〃       うるう月       */
/*************************************************/
int leap(int y, int m)
{
  if ( m==2 && ( (y%4==0 && y%100!=0) || y%400==0 ) )
    return 1;              /* うるう月 */
  else
    return 0;
}
[motoki@x205a]$ gcc modules-calendar.c
[motoki@x205a]$ ./a.out
Please type in year and month numbers.
2004 5
         2004 May
 Sun Mon Tue Wed Thu Fri Sat
                           1
   2   3   4   5   6   7   8
   9  10  11  12  13  14  15
  16  17  18  19  20  21  22
  23  24  25  26  27  28  29
  30  31
[motoki@x205a]$
```

8.3 静的外部変数, 静的関数 ——関数以外のモジュール——

{ ケリー&ポール 5.12 節 }

静的外部変数, 静的関数： 外部変数を宣言する際にデータ型の前に `static` という修飾子を付けると、その変数の有効範囲が同一ファイル内のその宣言以降に制限される。また、関数定義の前に `static` という修飾子を付けると、その関数の有効範囲が同一ファイル内に制限される。こういった変数, 関数を各々**静的外部変数**, **静的関数**という。

関数以外のモジュール： 静的外部変数はそのファイル内の関数が共有する局所的なデータを保有することになるので、そのファイル内の関数は独立なものではなく、この共通のデータを協調して管理する関数群と見なすことが出来る。従って、このソースファイル内の静的外部変数群、関数群を合わせたものも 1 つのモジュールと見ることが出来る。[この考え方は「オブジェクト指向」へと発展する。実際、こういったモジュールが「オブジェクト指向」における「オブジェクト」に相当する。]

情報隠蔽： (大きな) プログラムを作る上での設計指針の 1 つに、

各モジュールの外部インターフェースとして必要なものを用意し、(ソフトウェアの信頼性のために) モジュール内部の実装に関わる細部は外部からは見えない様にする

というものがあります。C 言語においては、関数以外のモジュールに対してこの情報隠蔽を進めるために静的外部変数や静的関数を導入することが出来ます。

例 8.3 (静的外部変数, 静的関数) 次の C ソースファイルを考える。

```
static int  i;  /* 静的外部変数 */
                /* 別のファイルからは参照できない */

static void f(...);  /* 静的関数のプロトタイプ */

void g1(...)
{
  .........     /* ここでは配列 a を参照できない */
}

void g2(...)
{
  .........     /* ここでは配列 a を参照できない */
}

static int  a[100];  /* 静的外部配列 */
                     /* 別のファイルからは参照できない */
                     /* 上の関数 g1, g2 からも参照できない */

static void f(...)   /* 静的関数 */
{                    /* 別のファイルからは呼び出せない */
  .........
}
```

このファイル／モジュールの場合、上から順に、

- 変数 i は、static 宣言されているので外部から参照されることはなく、それゆえこのモジュール内の内部データを記録したものと見なすことが出来る。
- 関数 g1 と g2 は外部へのサービスを提供するための関数と見ることが出来る。
- 配列 a も、変数 i と同様に static 宣言されているので外部から参照されることはなく、それゆえこのモジュール内の内部データを記録したものと見なすことが出来る。ただ、a の場合は、上で定義された関数 g1, g2 からも参照できない。
- 関数 f は外部とのインターフェースである g1 や g2 からの依頼を受けて内部の細かな作業を行う関数と見ることが出来る。

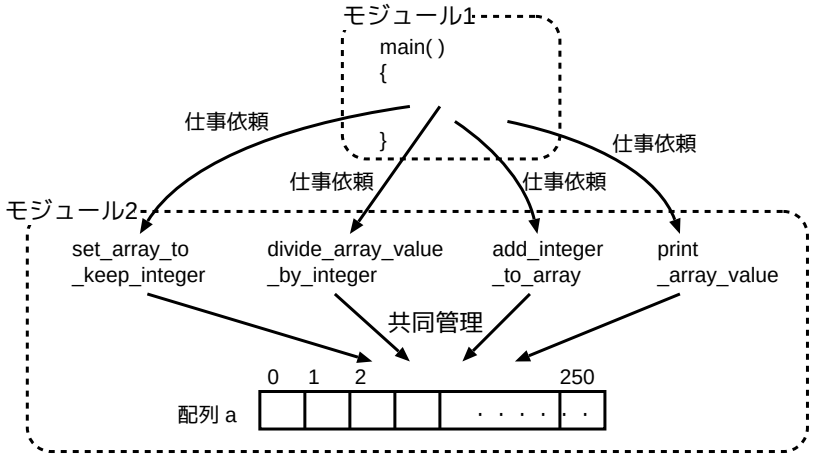
例題 8.4 (e の 1000 桁計算) 個々の関数をモジュールと見るのではなく、関連するデータとそれらを管理する関数群の集まりをモジュールと見る観点に立てば、例題 8.1 で挙げたプログラムは、

① 小数点以下 1000 桁の 10 進小数を記憶する int 型配列 `e[]` とこのデータ領域を管理／操作する関数 `set_array_to_keep_integer`, `divide_array_value_by_integer`, `add_integer_to_array`, `print_array_value` の集まり、

② 外部インターフェース用に提供されている 4 つの関数 (`set_array_to_keep_integer`, ...) を然るべき順序に呼び出すことによって①のモジュール内のデータ領域 `e` に Napier 数が記録されるようにしている部分、

の 2 つのモジュールに分けることが出来る。この考えの下で例題 8.1 のプログラムを構成し直してみよ。

(考え方) ここで言う「モジュール」の実体は 1 つのソースファイルに他ならない。例題 8.1 では int 型配列 `e[]` は main 関数内で確保し、main 内で他の関数を呼び出す際にその配列のアクセス権 (i.e. 配列の先頭番地と大きさ) を呼び出し先の関数にパラメータとして引き渡していた。 これに対して、指示された観点に立てば、配列 `e[]` はこれを直接使う関数群の共有するものなので、`e[]` は静的外部配列として確保され、この配列を共有する関数群と合わせて 1 つのソースファイルを作り、これを 1 つのモジュールと見る。 そして、残った main 関数 (と main から呼び出す関数のプロトタイプ) を基に別のソースファイルを作り、これを 2 つ目のモジュールと見る。



(再構成例) 例題 8.1 のプログラムは、例えば次のように 2 つのモジュール (ソースファイル) に分割できる。

注目点：
モジュール `modules-napier-sub1.c` の中では小数点以下 1000 桁の 10 進小数がどういう風に表されているかについては全く関知しない、従って小数点以下 1000 桁の 10 進小数の表し方はモジュール `modules-napier-sub2.c` の内部に隠蔽されている。

```
[motoki@x205a]$ nl modules-napier-sub1.c
     1  /*************************************************************/
     2  /* Napier数 e の1000桁計算                                   */
```

```
 3  /*-----------------------------------------                    */
 4  /*    別途用意された、                                          */
 5  /*      (1) 小数点以下 1000 桁の 10 進小数を記憶するための領域と  */
 6  /*      (2) そのデータ領域を操作するための関数群                  */
 7  /*    から成るモジュールに然るべき順序に然るべき指令を出す      */
 8  /*    ことによって Napier 数 e の値を小数点以下 1000 桁まで出力  */
 9  /*    するモジュール                                            */
10  /*    (このモジュールの中では、小数点以下 1000 桁の 10 進小数が  */
11  /*     どういう風に表されているかについては全く関知しない。)    */
12  /*************************************************************/

13  void set_array_to_keep_integer(int init_value);
14  void divide_array_value_by_integer(int divisor);
15  void add_integer_to_array(int value);
16  void print_array_value(char *padding);

17  int main(void)
18  {
19    int  k;

20    set_array_to_keep_integer(1);        /* e <-- 1 */

21    for (k=450; k>0; --k) {
22      divide_array_value_by_integer(k);  /* e <-- e/k */
23      add_integer_to_array(1);           /* e <-- e+1 */
24    }

25    printf("e =");
26    print_array_value("   ");           /* 結果の出力 */
27    return 0;
28  }
```

```
[motoki@x205a]$ nl modules-napier-sub2.c
```

```
 1  /*************************************************************/
 2  /* 小数点以下 1000 桁の 10 進小数を                           */
 3  /*       e[0]=整数部の数,                                    */
 4  /*       e[1]=小数点以下 1〜4 桁目の数,                      */
 5  /*       e[2]=小数点以下 5〜8 桁目の数,                      */
 6  /*           ......                                          */
 7  /*       e[250]=小数点以下 996〜1000 桁目の数                */
 8  /* という風に保存する配列 e と、                              */
 9  /* このデータ領域を管理する関数群のモジュール                 */
10  /*************************************************************/
```

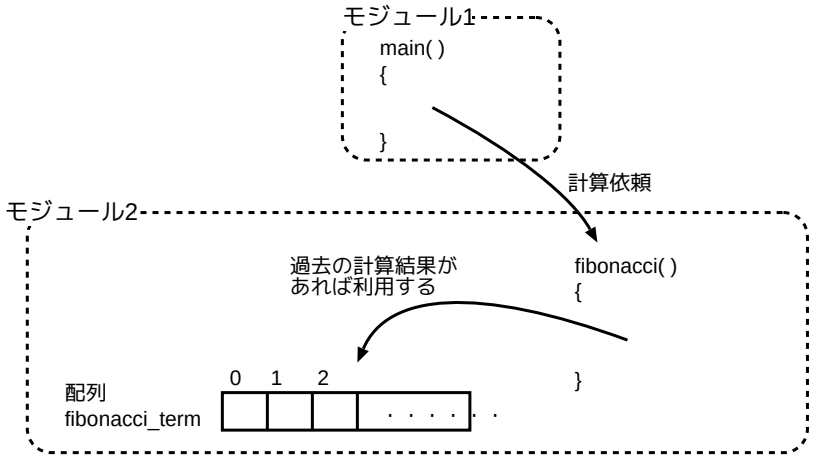
```
11  #include  <stdio.h>
12  #define   LIMIT    7
13  #define   SIZE   251

14  static  int  e[SIZE];

15  /*-----------------------------------------------------------*/
16  /* 10 進小数を表す静的外部配列 e の値を初期設定する         */
17  /*-----------------------------------------                  */
18  /* (仮引数) init_value : 初期設定したい値 (int 型)          */
19  /* (関数値) : なし                                          */
20  /* (機能) :  静的外部配列 e の表す値が init_value に       */
21  /*           なる様にする                                   */
22  /*-----------------------------------------------------------*/
23  void set_array_to_keep_integer(int init_value)
24  {
25    int i;
26
27    e[0] = init_value;
28    for (i=1; i<SIZE; ++i)
29      e[i] = 0;
30  }

31  /*-----------------------------------------------------------*/
32  /* 10 進小数を表す静的外部配列 e の値を整数で割る           */
33  /*-----------------------------------------                  */
34  /* (仮引数) divisor : 除数                                  */
35  /* (関数値) : なし                                          */
36  /* (機能) :  静的外部配列 e の表す値を整数 divisor で割る */
37  /*-----------------------------------------------------------*/
38  void divide_array_value_by_integer(int divisor)
39  {
40    int i;
41
42    for (i=0; i<SIZE-1; ++i) {
43      e[i+1] += (e[i] % divisor) * 10000;
44      e[i]   /= divisor;
45    }
46    e[SIZE-1] /= divisor;
47  }

48  /*-----------------------------------------------------------*/
49  /* 10 進小数を表す静的外部配列 e の値に整数を加算する       */
```

```
50  /*-----------------------------------------                    */
51  /* (仮引数) addend : 加数                                       */
52  /* (関数値) : なし                                              */
53  /* (機能) :  静的外部配列 e の表す値に整数 addend を加算する*/
54  /*-------------------------------------------------------------*/
55  void add_integer_to_array(int addend)
56  {
57    e[0] += addend;
58  }

59  /*-------------------------------------------------------------*/
60  /* 10 進小数を表す静的外部配列 e の値を出力する                 */
61  /*-----------------------------------------                    */
62  /* (仮引数) padding : 2 行目以降の左端に必ず出力する文字列    */
63  /* (関数値) : なし                                              */
64  /* (機能) :  静的外部配列 e の表す 10 進小数値を出力する。     */
65  /*           その際、                                           */
66  /*             *小数点以下は 8*LIMIT 桁毎に改行する,           */
67  /*             *2 行目以降の出力においては左端に必ず文字列    */
68  /*               padding を埋める。                             */
69  /*-------------------------------------------------------------*/
70  void print_array_value(char *padding)
71  {
72    int i, count;
73
74    printf("%2d.", e[0]);
75    count = 1;
76    for (i=1; i<SIZE; i+=2, ++count) {
77      printf("%04d%04d ", e[i], e[i+1]);
78      if (count >= LIMIT) {
79        printf("\n%s   ", padding);
80        count = 0;
81      }
82    }
83    printf("\n");
84  }
```

```
[motoki@x205a]$ gcc modules-napier-sub1.c modules-napier-sub2.c
[motoki@x205a]$ ./a.out
e = 2.71828182 84590452 35360287 47135266 24977572 47093699 95957496
      69676277 24076630 35354759 45713821 78525166 42742746 63919320
                (途中省略)
      15747704 17189861 06873969 65521267 15468895 70350354
[motoki@x205a]$
```

例題 8.5 (連想計算;Fibonacci 数列の再帰計算を効率的にする) 例 7.6 において、フィボナッチ数列を定義した漸化式に基づいて数列の項を再帰計算すると異常に非効率な計算になることを示した。これに対して、漸化式に (ほぼ) 忠実に従った計算方法を保ったまま計算の非効率性を避けることが出来るかどうか考えよ。

(考え方) Fibonacci 数列を計算するモジュール内に静的外部配列を用意して、この配列に過去に計算したことのある計算結果を保存する。そして、漸化式に従った計算を続ける前に過去に計算結果が保存されているかどうかをチェックする様にすれば、同じ計算を何度も繰り返す無駄を省くことが出来る。すなわち、過去に計算結果が保存されていない場合だけ、漸化式に従って再帰計算することになる。



(プログラミング) 例えば次の様にプログラムを構成すれば、漸化式に (ほぼ) 忠実に従った計算方法を保ったまま計算の非効率性を避けることが出来る。

```
[motoki@x205a]$ nl modules-fibonacci-sub1.c
     1  #include <stdio.h>

     2  int  fibonacci(int n);

     3  int main(void)
     4  {
     5    int  i;

     6    scanf("%d", &i);
     7    printf("fibonacci(%d) = %d\n", i, fibonacci(i));
     8    return 0;
     9  }
```

```
[motoki@x205a]$ nl modules-fibonacci-sub2.c
     1  #define  SIZE  100
     2  #define  TRUE  1
```

```
 3  typedef  int  Boolean;

 4  static Boolean Already_computed[SIZE]; /* 過去に計算したことがあるか */
 5                                         /* どうかを記憶する静的外部変数; */
 6                                         /* ゼロ (false) に初期化される。   */
 7  static int     fibonacci_term[SIZE];   /* 過去の計算結果をここに保存 */

 8  static int record(int num, int value);

 9  int fibonacci(int n)
10  {
11    printf("called fibonacci(%d)\n", n);
12          /* どういう計算が行われるかを観察するために */
13    if (Already_computed[n])
14      return fibonacci_term[n];
15
16    if (n <= 1)
17      return record(n, n);
18    else
19      return record(n, fibonacci(n-1)+fibonacci(n-2));
20  }

21  static int record(int n, int value)
22  {
23    Already_computed[n] = TRUE;
24    fibonacci_term[n]   = value;

25    return value;
26  }
[motoki@x205a]$ gcc modules-fibonacci-sub1.c modules-fibonacci-sub2.c
[motoki@x205a]$ ./a.out
6
called fibonacci(6)
called fibonacci(5)
called fibonacci(4)
called fibonacci(3)
called fibonacci(2)
called fibonacci(1)
called fibonacci(0)
called fibonacci(1)
called fibonacci(2)
called fibonacci(3)
called fibonacci(4)
```

```
fibonacci(6) = 8
[motoki@x205a]$
```

例題 8.6 (疑似乱数発生) 標準ライブラリ関数 `int rand( )` を用いずに、疑似乱数発生のための汎用のモジュールを作成せよ。

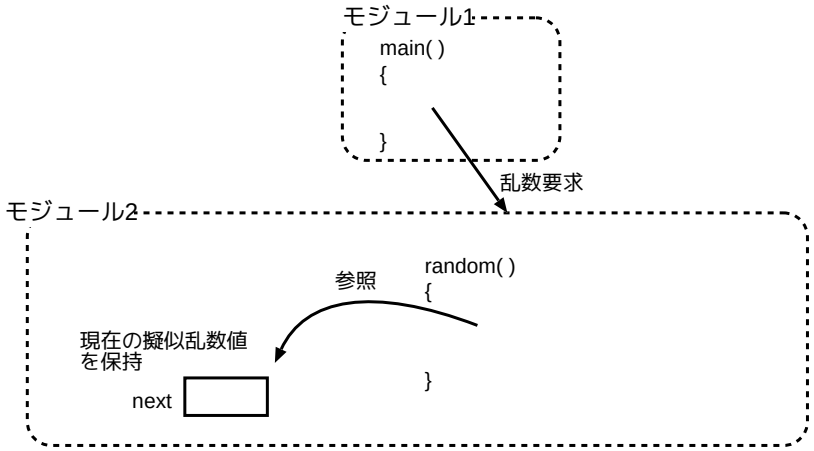
(考え方) 疑似乱数を発生させる最も単純な方法は**線形合同法**と呼ばれる方法で、この方法では定数 a, b, N を定めて、

$$r_{k+1} \longleftarrow \mathrm{mod}(a \times r_k + b, N)$$

という式に基づいて現在の疑似乱数値 r_k から次の疑似乱数値 r_{k+1} を求める。 結果的に、初期値 r_0 を与えてやれば $0 \sim N-1$ の間の整数列 $r_0, r_1, r_2, r_3, ...$ が決まるので、これを乱数の列と見做そうというのである。 この線形合同法で疑似乱数列を発生させる際は、

- 現在の疑似乱数値を保持する変数,
- 現在の疑似乱数値を更新してその結果を返す関数

の 2 つから成るモジュールを考えれば良い。



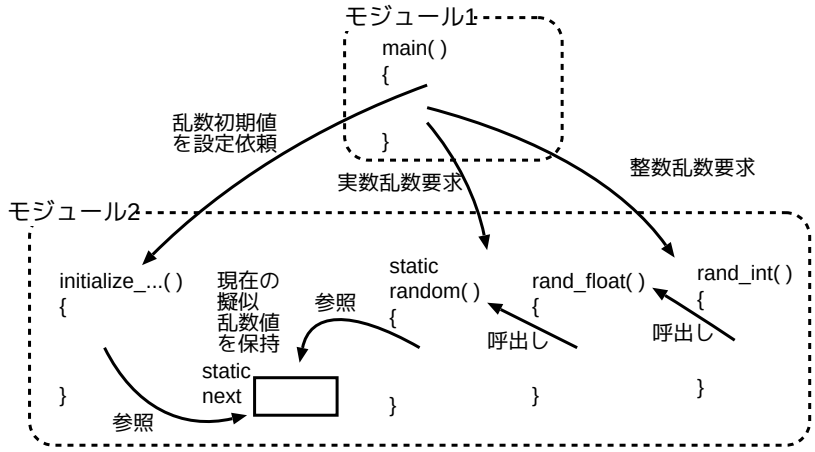
(プログラミング) 定数 a, b, N を $a = 1103515245, b = 12345, N = 2^{32}$ としてプログラムを構成した。 また、生成する疑似乱数列の明白な周期性を避けるために現在の疑似乱数値を構成する 32 ビットの下位から 31~17 ビット目を取り出すことにし、使い易さのために、その取り出した値を基に

- 指定された数未満の非負整数を生成する関数,
- 0 以上 1 未満の実数値を生成する関数

の 2 つと、

- 保持している疑似乱数値を初期設定する関数,

もモジュールに追加した。



出来たモジュールを次に示す。

```
[motoki@x205a]$ nl modules-random.c
```

```
 1  /*************************************************************/
 2  /* (1) 指定された数以下の非負整数をランダムに割り当てる関数、 */
 3  /* および (2) [0,1) 内の実数をランダムに割り当てる関数、      */
 4  /* を提供するモジュール                                       */
 5  /*-----------------------------------------------------------*/
 6  /*      ここでは、標準ライブラリで提供された疑似乱数生成系   */
 7  /*      とほぼ同等のものを示す。(あまり良くない。)            */
 8  /*************************************************************/

 9  static unsigned long next=1;

10  double     rand_float(void);
11  static int random(void);

12  /*-----------------------------------------------------------*/
13  /* random seed を初期設定する関数                            */
14  /*-------------------------------------                     */
15  /* (仮引数) seed : random seed の初期値を記憶した変数        */
16  /* (関数値) :  なし                                          */
17  /*-----------------------------------------------------------*/
18  void initialize_randomizer(unsigned long seed)
19  {
20    next = seed;
21  }

22  /*-----------------------------------------------------------*/
23  /* 指定された数未満の非負整数をランダムに割り当てる関数      */
24  /*-------------------------------------                     */
25  /* (仮引数) to : 生成すべき乱数値の上限 (+1) を記憶した変数  */
```

```
26  /* (関数値) :  区間 [0, to) 内の整数疑似乱数値                */
27  /*-------------------------------------------------------------*/
28  int rand_int(int to)
29  {
30    int  k;

31    k = (int) (rand_float() * to);
32    return (k==to ? to-1 : k);
33  }

34  /*-------------------------------------------------------------*/
35  /* [0,1) 内の実数をランダムに割り当てる関数                    */
36  /*-----------------------------------------                    */
37  /* (仮引数)  : なし                                            */
38  /* (関数値)  :  区間 [0, 1) 内の実数疑似乱数値                 */
39  /*-------------------------------------------------------------*/
40  double rand_float(void)
41  {
42    return (double)random() / 32768;
43  }

44  /*-------------------------------------------------------------*/
45  /* 次の疑似乱数に移るために用意された局所的な関数              */
46  /*-----------------------------------------                    */
47  /* (仮引数)  : なし                                            */
48  /* (関数値)  :  0 以上 32767 以下の疑似整数乱数値              */
49  /* (機能) :  疑似乱数生成のためのに用意した静的外部変数        */
50  /*           next の値を線形合同法で更新し、結果の next の値 */
51  /*           を基に 0 以上 32767 以下の疑似整数乱数値を生成   */
52  /*-------------------------------------------------------------*/
53  static int random(void)
54  {
55    next = next*1103515245 + 12345;
56    return (unsigned int) ((next/65536) % 32768);
57  }
```

ここで、

- プログラム 56 行目 に現われる 65536 は 2^{16} の値を表し、32768 は 2^{15} の値を表す。従って、((next/65536) % 32768) は変数 next の下から 17〜31 ビット目の部分を取り出す操作を表している。

このモジュールは色々なモジュールと組み合わせて使うことが出来る。例えば次の通り。

```
[motoki@x205a]$ nl modules-random-main.c
     1  #include <stdio.h>
```

```
 2  void initialize_randomizer(unsigned long seed);
 3  int rand_int(int to);
 4  double rand_float(void);

 5  int main(void)           /* こちら側からは            */
 6  {                        /* modules-random.c 内の静的 */
 7    int i;                 /* 外部変数 next は見えない  */
 8    unsigned long seed;

 9    printf("Input a random seed:  ");
10    scanf("%d", &seed);
11    initialize_randomizer(seed);
12    for (i=0; i<7; ++i)
13      printf("%6d", rand_int(100));
14    printf("\n");
15    for (i=0; i<5; ++i)
16      printf("%f ", rand_float());
17    printf("\n");
18    return 0;
19  }
[motoki@x205a]$ gcc modules-random-main.c modules-random.c
[motoki@x205a]$ ./a.out
Input a random seed:  333
    11     1    58    91    11    68    27
0.380646 0.934570 0.318390 0.709808 0.658264
[motoki@x205a]$
```

演習問題

□**演習 8.1 (e の 1000 桁計算; モジュール化しない版との比較)**　例題 8.1 のプログラムを複数の関数に機能分割せずに書き直してみよ。書き直して得られたプログラムと例題 8.1 のものを見比べて、どちらがどういう風に分かり易くなっているか検討せよ。

□**演習 8.2 (階乗の表)**　1!~53! を計算して、それらを桁を揃えて出力せよ。
[$53! \approx 4.27\times10^{69}$ となります。]

□**演習 8.3 (静的外部変数)** 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。

```
xcspc60_147% cat test0107_1b.c
#include <stdio.h>

int a();
int b();

int main(void)
{
  printf("a(): %d\n", a());
  printf("b(): %d\n", b());
  return 0;
}

static int c=1;

int a()
{
  return c;
}
```

(右上へ続く ↗)

(↗ 左下からの続き。)

```
xcspc60_148% cat test0107_1c.c
static int c=2;

int b()
{
  return c;
}
xcspc60_149% cc test0107_1b.c test0107_1c.c
xcspc60_150% a.out
```

```
xcspc60_151%
```

□演習 8.4 (階乗の表) 演習 8.2 で作ったプログラムを例題 8.4 に倣って 2 つのモジュールに分けてみよ。

□演習 8.5 (住所録管理のモジュール) 住所録を管理するモジュールを設計してみて下さい。

プログラムを自在に作るための道具立て

9 自習 配列, ポインタ, 文字列

- 復習 一次元配列, ポインタ,
- 配列とポインタの関係, ポインタ算術,
- 文字列の扱い, 不揃い配列
- 多次元配列

9.1 復習 一次元配列

{ 講義ノート 1.3 節, 1.4.7 節 }

配列宣言の例：

宣言		説明
`int a[100];`	···	大きさが 100 の `int` 型一次元配列。`a[0]`, `a[1]`, ..., `a[99]` が確保される。
`int f[5] = {0, 1, 2, 3, 4};`	···	`f[0]`=0, `f[1]`=1, ... と初期設定される。
`int a[100] = {0};`	···	初期値指定が足りない分は 0 と見なされ、結局全て 0 に初期設定される。
`double a[] = {2.0, 3e-2, -5.};`	···	初期値指定が完全な場合は、配列の大きさ指定は省略できる。
`char s[] = "abc";`	···	`char` 型配列で文字列を表す際の略記法。`char s[4] = {'a', 'b', 'c', '\0'};` と宣言するのと同等。

注意：

- 配列の添字は 0 から始まる。
- 配列の添字が有効範囲にあることをチェックするのはプログラマ。
- 添字式を囲む四角括弧 [ ] は実は演算子。結合の優先順位は、関数引数を囲む ( ) と共に最も高い。
- 配列の大きさは定数式で指定するのが無難である。実際、例えば次の右下の書き方は誤りで、左下の書き方も ANSI C99 より前の規格では誤りである。

```
int f(int size)
{
  int a[size];
  ......
```

```
......
scanf("%d", &size);
double x[size];
......
```

この制約はコンパイル時に確保する領域の大きさが確定している必要があるためである。必要な配列の大きさがコンパイル時に決まらない場合は、通常は多少の無駄があっても十分な大きさの配列を用意する。どうしても実行時に動的に配列の大きさを決めたい場合は、`malloc()` 関数等を用いて各々次の様な書き方をすればよい。

```
int f(int size)
{
  int *a;
  a=(int*)malloc(sizeof(int)*size);
  ......
```

```
int *x;
......
scanf("%d", &size);
x=(int*)malloc(sizeof(int)*size);
......
```

9.2 復習 ポインタ

{ 講義ノート 7.3 節 }

番地演算子 & と間接演算子 * (7.3 節)：

`&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。

9.3 配列とポインタの関係, ポインタ算術

{ ケリー&ポール 6.4~6.5 節,8.14 節,11.16 節 }

配列名：

- **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。**(添字式を囲む四角括弧 [ ] は演算子である。)

ポインタ算術： ポインタに関する算術は特殊である。例えば、

- ポインタ p に関して ++p という演算を施すと、このポインタはメモリ空間上の次の番地ではなく配列の次の要素を指すようになる。
- ポインタ p と int 型変数 i の間の加算 p+i は実際には

  p + i× sizeof(p の指すデータ型) 番地

  を表す。

  ⟹ 配列要素 a[i] の代わりに*($a+i$) という式を用いることが出来る。また、逆に、配列要素を指すポインタ p があったとすると、p という名前の配列が宣言されてなくても、*(p+i) の意味で p[i] と書くことができる。
- p と q が同じ配列を指しているポインタである時、p-q は実際には p の指している所から q の指している所までの要素の個数、すなわち

  $$\frac{\text{q の指す番地から p の指す番地までのバイト数 (符号あり)}}{\texttt{sizeof(p の指すデータ型)}}$$

  という整数を表す。

例 9.1 (ポインタ算術)

```
[motoki@x205a]$ nl pointer-arithmetic-Kelley.c
     1 #include <stdio.h>

     2 int main(void)
     3 {
     4   double  a[2], *p, *q;
```

```
 5   p = a;
 6   q = p+1;                          /* ポインタ算術 */
 7   printf("%d\n", q-p);              /* ポインタ算術 */
 8   printf("%d\n", (int)q - (int)p);  /* 通常の算術   */
 9   return 0;
10 }
[motoki@x205a]$ gcc pointer-arithmetic-Kelley.c
[motoki@x205a]$ ./a.out
1
8
[motoki@x205a]$
```

配列とポインタの関係について：

- 配列名は、その配列の先頭要素を指す定数ポインタとして振舞う。 従って、
  ① s[i] と*(s+i) は等価である。(s が char 型以外の時も、これは成立する。)
  ② 配列要素を指すポインタ p があったとすると、*(p+i) の意味で p[i] と書くことも出来る。

例 9.2 (配列とポインタの関係) 例題 7.4 の中で挙げた関数 sum の定義

```
14  double sum(double a[], int size)
15  {
16    int    i;
17    double sum=0.0;

18    for (i=0; i < size; ++i)
19      sum += a[i];
20    return sum;
21  }
```

は、次のように書くことも出来る。

```
14  double sum(double a[], int size)
15  {
16    int    i;
17    double sum=0.0;

18    for (i=0; i < size; ++i)
19      sum += *(a+i);
20    return sum;
21  }
```

これは、さらに次のように書くことも出来る。

```
14  double sum(double a[], int size)
15  {
16    int    i, *p;
```

```
17    double sum=0.0;

18    for (p=a; p < &a[size]; ++p)
19      sum += *p;
20    return sum;
21  }
```

例題 9.3 (Quicksort; 未整列区間を配列要素へのポインタで表して引数結合を行う版) 例題 7.3 で挙げた Quicksort のプログラムにおいては、未整列の区間の両端の位置 (関数の引数 from と to) を表すのに配列の添字を用いていた。配列の添字の代わりに配列要素を指すポインタを関数引数として用いるように、例題 7.3 のプログラムを書き換えてみよ。

(考え方) 処理すべき配列要素の両端へのポインタを quicksort( ) 関数に引き渡すので、配列の先頭要素へのポインタを値とする配列名を quicksort( ) 関数に引き渡す必要は無くなる。従って、main( ) 内からの quicksort( ) 関数の呼び出しは quicksort(a, &a[0], &a[SIZE-1]); ではなく quicksort(&a[0], &a[SIZE-1]); という風に書けば良い。これに対応して、quicksort( ) の関数プロトタイプは次のようにすれば良い。

```
void quicksort(int *from, int *to);
```

呼び出された quicksort( ) 関数の中では、引数として受け取ったポインタを使って処理内容を書き表すことになるので、quicksort( ) の中から partition( ) を呼び出す際に引数として引き渡すデータも配列添字ではなく配列要素へのポインタとし、partition( ) の関数値もポインタとするのが妥当である。従って、partition( ) の関数プロトタイプは次のようにすれば良い。

```
int *partition(int *from, int *to);
```

(プログラミング) 修正例を次に示す。

```
[motoki@x205a]$ nl pointer-quicksort-3.c
     1 #include <stdio.h>
     2 #include <stdlib.h>      /* 乱数発生のライブラリ関数を使うため */

     3 #define  SIZE   100
     4 #define  WIDTH  10
     5 #define  TRUE    1

     6 void set_an_array_random(int a[], int size);
     7 void pretty_print(int a[], int size);
     8 void quicksort(int *from, int *to);
     9 int  *partition(int *from, int *to);

    10 int main(void)
    11 {
```

```
12   int      a[SIZE], seed;

13   printf("Input a random seed (0 - %d):  ", RAND_MAX);
14   scanf("%d", &seed);
15   srand(seed);

16   set_an_array_random(a, SIZE);
17   printf("\nbefore sorting:\n");
18   pretty_print(a, SIZE);

19   quicksort(&a[0], &a[SIZE-1]);
20   printf("\nafter sorting:\n");
21   pretty_print(a, SIZE);
22   return 0;
23 }

24 /*****************************************************************/
25 /* 引数で与えられた配列の各要素をランダムに設定する             */
26 /*---------------------------------------                        */
27 /* (仮引数) x    : int 型配列                                    */
28 /*          size : int 型配列 x の大きさ                         */
29 /* (関数値)  :  なし                                             */
30 /* (機能) :  配列要素 x[0]～x[size-1] に 0～999 の間の乱数を   */
31 /*           設定する。                                          */
32 /*****************************************************************/
33 void set_an_array_random(int x[], int size)
34 {
35   int i;

36   for (i=0; i<size; ++i)
37     x[i] = rand() % 1000;
38 }

39 /*****************************************************************/
40 /* 引数で与えられた配列の要素を順番に全て出力する               */
41 /*---------------------------------------                        */
42 /* (仮引数) x    : int 型配列                                    */
43 /*          size : int 型配列 x の大きさ                         */
44 /* (関数値)  :  なし                                             */
45 /* (機能) :  配列要素 x[0]～x[size-1] の値を順番に全て出力   */
46 /*           する。但し、各々の値は横幅 7 カラムのフィールド   */
47 /*           に出力することにし、また、1 行に WIDTH 個の要素    */
48 /*           を出力する。                                        */
```

```
49 /*************************************************************/
50 void pretty_print(int x[], int size)
51 {
52   int i, count=1;

53   for (i=0; i<size; ++i, ++count) {
54     printf("%7d", x[i]);
55     if (count >= WIDTH) {
56       printf("\n");
57       count = 0;
58     }
59   }
60   if (count > 1)
61     printf("\n");
62 }

63 /*************************************************************/
64 /* 引数で与えられた配列要素を小さい順に並べ替える            */
65 /*-----------------------------------------                  */
66 /* (仮引数) from : int 型配列要素を指すポインタ              */
67 /*          to   : int 型配列要素を指すポインタ              */
68 /* (関数値)  :  なし                                        */
69 /* (機能) :  quicksort アルゴリズムを使って、配列要素       */
70 /*                *from,*(from+1),*(from+2), ..., *to       */
71 /*           を値の小さい順に並べ替える。                   */
72 /*************************************************************/
73 void quicksort(int *from, int *to)
74 {
75   int *pivot_pointer;

76   if (from < to) {
77     pivot_pointer = partition(from, to);    /* 分割操作 */
78     quicksort(from, pivot_pointer - 1);
79     quicksort(pivot_pointer + 1, to);
80   }
81 }

82 /*************************************************************/
83 /* 引数で与えられた配列の部分列に quicksort の分割操作を施す  */
84 /*                                        (quicksort の関数) */
85 /*-----------------------------------------                  */
86 /* (仮引数) from : int 型配列要素を指すポインタ              */
87 /*          to   : int 型配列要素を指すポインタ (> from)     */
```

```
 88 /* (関数値) : 分割操作によって得られた枢軸要素を指すポインタ*/
 89 /*            (以下の「(機能)」の項で出て来る pivot_pointer)*/
 90 /* (機能) : *from,*(from+1), ..., *to を並べ替えて          */
 91 /*          max{*from,...,*(pivot_point-1)} <= *pivot_point */
 92 /*          *pivot_point < min{*(pivot_point+1),...,*to}    */
 93 /*          となるようにする。                               */
 94 /*****************************************************************/
 95 int *partition(int *from, int *to)
 96 {
 97   int pivot;

 98   pivot = *from;       /* 最初の要素を枢軸要素に選ぶ。 */
 99   while (TRUE) {       /* 工夫の余地あり。             */
100     for ( ; from<to && *to>pivot; --to)
101       ;
102     if (from == to) {
103       *from = pivot;
104       return from;
105     }
106     *from = *to;

107     for ( ; from<to && *from<=pivot; ++from)
108       ;
109     if (from == to) {
110       *to = pivot;
111       return to;
112     }
113     *to = *from;
114   }
115 }
```

注意：
この例は、配列要素へのポインタを引数結合することによって配列の受渡しをすることも出来ることを示しているだけで、そうすべきだと言っている訳ではない。 実際、そうすることによって、引数を受け取る関数のソースコード上では配列を取り扱っているということが明示的に現れなくなるので、プログラムが分かりにくくなることが多い。

9.4 文字列の扱い, 不揃い配列

{ 講義ノート 1.4.7 節, ケリー&ポール 6.9~6.10 節,6.14 節 }

文字列についてのまとめ (**1.4.7** 節)：

- 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'	$\cdots$

- 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 4.7 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";` $\cdots$ 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";` $\cdots$ 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。]

例題 9.4 (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

① 英文章の次の 1 行の読み込み,
② 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,
③ 変換後の 1 行の出力

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。($\Longrightarrow$ 4.7 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。($\Longrightarrow$ 4.7 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。($\Longrightarrow$ 4.7 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。($\Longrightarrow$ 4.7 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：

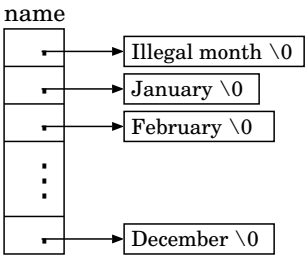
- 長さの様々な文字列を 2 次元の char 型配列に入れておくのはメモリに無駄が出来る。例えば、例 8.2 で作ったプログラムの中 (main) に

```
char name[][15]={                          /* name[k]           */
        "Illegal month",                   /* = k 番目の月の名前 */
        "January", "February", "March",
        "April", "May", "June",
```

```
            "July", "August", "September",
            "October", "November", "December"
        };
```

という宣言があるが、この部分は次のように書き直すとメモリが節約できる。(**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"
        };
```



9.5 多次元配列

{ 講義ノート 1.4.7 節, ケリー&ポール 6.11 節 }

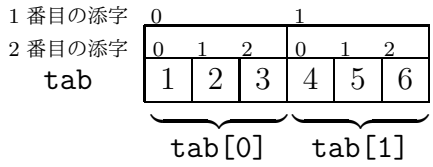
多次元配列の宣言 (3.8 節):

- 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}};
```



多次元配列の表し方： C 言語においては、1 次元配列を 1 つの要素とする配列を 2 次元配列、2 次元配列を 1 つの要素とする配列を 3 次元配列、..... と考える。
$\Longrightarrow$ 全ての要素はメモリ上に連続的に配置されることになる。

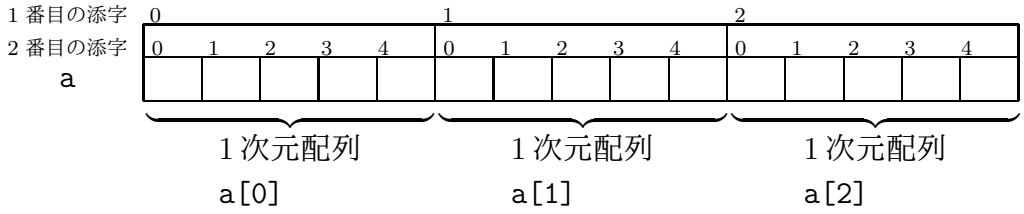
例 9.5 (2 次元配列の様子) 例えば、

```
int a[3][5];
```

という宣言は、演算子 [ ] の結合性 (左から右) から

```
int (a[3])[5];
```

と同等である。 この宣言は、配列 a は a[0]～a[2] から成り、配列要素 a[0]～a[2] は各々 int 型の 1 次元配列であると言っている訳だから、次のようにメモリが確保される。



配列名の値：

- 1 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る。(9.3 節)
- C 言語においては、配列の機構としては 1 次元のものしか用意されていない。[多次元配列は 1 次元配列の組合せにすぎない。]

$\Longrightarrow$ 配列 A の名前は計算機内部では A[0] への定数ポインタ &A[0] と同等に扱われる。

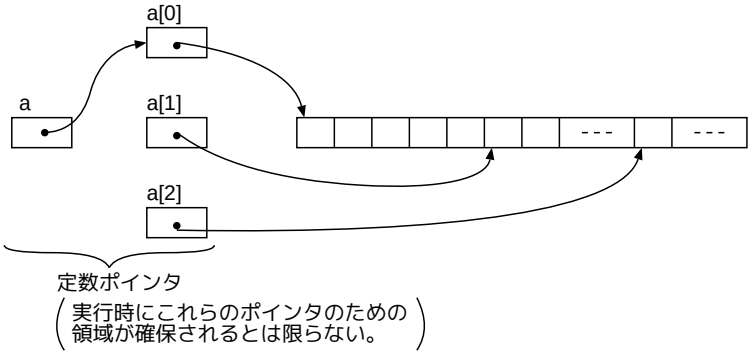
例 9.6 (配列名の値) 宣言

```
int a[3][5];
```

によって確保される 2 次元配列の場合、

配列名 a は 大きさ 5 の int 型 1 次元配列 a[0] への定数ポインタ &a[0] と同等、
配列名 a[0] は int 型データ領域 a[0][0] への定数ポインタ &a[0][0] と同等、
配列名 a[1] は int 型データ領域 a[1][0] への定数ポインタ &a[1][0] と同等、
配列名 a[2] は int 型データ領域 a[2][0] への定数ポインタ &a[2][0] と同等、

ということになる。



例 9.7 (配列要素へのアクセスの仕方) 多次元配列の場合、配列要素へのアクセスの仕方は沢山ある。例えば、宣言

```
int a[3][5];
```

によって確保される 2 次元配列の場合、次の式は全て同等である。

```
a[i][j]
(*(a+i))[j]
*(a[i]+j)
*(*(a+i)+j)
```

……(1 次元配列の場合に、一般に A[i] と *(A+i) が同等であることを用いているだけ。)

```
*(&a[0][0]+5*i+j)
```

……(配列の内部構造を考えている。)

例 9.8 (多次元配列の初期設定) 多次元配列の初期設定の仕方は、1 次元配列の場合の考え方を自然に拡張しただけである。 例えば、1 次元配列の場合は、

```
int a[3]={ a[0] の初期値 , a[1] の初期値 , a[2] の初期値 );
```

という風に初期設定する。 いま、a[0]～a[2] を int 型データ領域ではなく大きさ 5 の int 型配列と見ることにして、同じ書き方に乗っ取って書けば、

```
int a[3][5]={ a[0] の初期値 , a[1] の初期値 , a[2] の初期値 );
```

すなわち、

```
int a[3][5]={ { a[0][0] の初期値 , ... , a[0][4] の初期値 },
              { a[1][0] の初期値 , ... , a[1][4] の初期値 },
              { a[2][0] の初期値 , ... , a[2][4] の初期値 }};
```

ということになる。 指定が欠けている場合に 0 が補充されるという規則も、1 次元の場合と同様に生きている。

演習問題

□演習 9.1 (C 言語における文字の扱い) 次の C コードを実行するとどういう出力が得られるか？

```
char a[]={'d', 'o', 'g', 's', '\0'};
printf("(1)%s\n", a);
printf("(2)%c%c%c\n", a[0], a[1], a[2]);
printf("(3)%c%c%c\n", *a, *(a+1), *(a+2));
printf("(4)%c%c%c\n", *a+1, *a+2, *a+3);
```

□演習 9.2 (C 言語における文字の扱い) 次の C コードを実行するとどういう出力が得られるか？

```
int a[4]={97,98,99,0}, d[4]={49,49,49,0}, *x=a, *y=d, *tmp;
printf("(4)%s\n", a);
printf("(5)%d\n", *d+1);
printf("(6)%d\n", *(d+1));
printf("(7)%d\n", x[0]);
tmp = x;
x   = y;
y   = tmp;
printf("(8)%d\n", x[0]);
```

□演習 9.3 (文字列定数) 次の C コードを実行するとどういう出力が得られるか？

```
printf("(1)%c\n", "cats"[1]);
printf("(2)%s\n", &("cats"[2]));
printf("(3)%c\n", (*"cats")+1);
printf("(4)%c\n", *("cats"+1));
```

10 自習 関数 (その 3)

- 復習 参照呼出し, 一次元配列を関数の引数として受渡しする方法,
- 多次元配列を関数の引数として受渡しする方法,
- 関数 main の引数—コマンドラインでパラメータを指定する方法—,
- 関数を関数の引数として受渡しする方法

10.1 復習 参照呼出し

{ 講義ノート 7.3 節 }

番地演算子 & と間接演算子 * (7.3 節) :

`&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。

参照呼出しと同等のことを行なう方法 (7.3 節) :

- 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のポインタ (i.e. 番地) を与える。

例 10.1 (参照呼出し)

```
[motoki@x205a]$ nl func-call-by-ref-Kelley.c
     1 #include <stdio.h>

     2 void swap(int *p, int *q);

     3 int main(void)
     4 {
     5   int i=3, j=5;

     6   swap(&i, &j);
     7   printf("i=%d  j=%d\n", i, j);
     8   return 0;
     9 }

    10 void swap(int *p, int *q)
    11 {
    12   int temp;

    13   temp = *p;
    14   *p   = *q;
    15   *q   = temp;
    16 }
```

```
[motoki@x205a]$ gcc func-call-by-ref-Kelley.c
[motoki@x205a]$ ./a.out
i=5  j=3
[motoki@x205a]$
```

ここで、

- プログラム 2 行目の `*p` は、**ポインタ `p` の指す記憶領域** (すなわち、`p` という変数に番地が入っていると見て、その `p` 番地の記憶領域) を表す。この `*p` が `int` 型であると宣言している訳だから、この 2 行目のプロトタイプ宣言により、関数 `swap` の第 1 引数として `int` 型記憶領域へのポインタを受け取ることになる。
- プログラム 6 行目の `&i` は、**変数 `i` の番地** (すなわち、変数 `i` へのポインタ) を表す。
- プログラム 12 行目に現われる `*p` は、やはりポインタ `p` の指す記憶領域を表す。普通の変数もそうだが、こういうものが代入式において等号の右側に来ると、その領域に入った値が式の評価に使われる。
- プログラム 13 行目に現われる `*p` も、やはりポインタ `p` の指す記憶領域を表す。ここでは、代入式において等号の左側に来ているので、等号の右の式の評価結果がこの記憶領域に代入されることになる。

10.2 復習 一次元配列を関数の引数として受渡しする方法

{ 講義ノート 7.4 節 }

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。[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。]

10.3 多次元配列を関数の引数として受渡しする方法

{ ケリー&ポール 6.11 節 }

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のは、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。]

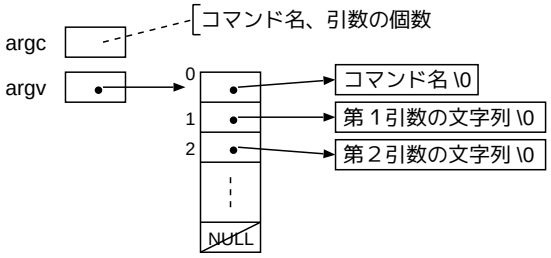
例 10.2 (多次元配列を関数の仮引数とする場合) 3 次元配列 `int a[7][9][2]` を引数として受渡したい時には、仮引数部は次のように書く。

```
int a[][9][2]
int a[7][9][2]                          } いずれか
int (*a)[9][2]        ←― 明示的な書き方 }
```

10.4 関数 main の引数 ―コマンドラインでパラメータを指定する方法―

コマンドラインでパラメータ指定する方法について：

- 主ルーチンの引数部を `main(int argc, char *argv[])` と書けば、コマンドラインでパラメータを指定できる様になる。これによって、関数 `main` の起動直後には、`argc` と `argv` は次の図の様に設定される。



例題 10.3 (コマンドラインでのパラメータ指定) 整数乱数を次々と生成・出力する C プログラムを作成せよ。但し、このプログラムの実行に当たってはコマンドラインから次の形式のオプションを指定できるようにせよ。

- `-seed` 非負整数 ··· random seed を非負整数に初期設定する、という指示。(デフォルトは 1。)
- `-max` 正整数 ··· [0, 正整数] の間の乱数を生成する、という指示。(デフォルトは `RAND_MAX`。また、指定した正整数が `RAND_MAX` よりも大きい場合も、`RAND_MAX` を用いる。)
- `-num` 正整数 ··· 正整数個の乱数を生成・出力する、という指示。(デフォルトは 10。)

(考え方) この例題はコマンドラインでオプション指定する方法を例示するためのものなので、疑似乱数の生成にはライブラリ関数として用意されている

```
int srand(unsigned seed); と
int rand(void);
```

を用いることにする。実行直後にコマンドラインオプションを解読することになるが、そのためには main( ) の引数 argc と argv を用いてコマンドに続く空白で区切られた文字列 argv[1], argv[2], ..., argv[argc-1] を順に調べて行けば良い。そして、もしその文字列が "-seed", "-max" または "-num" というものであったなら、標準ライブラリ関数 sscanf( ) を用いてそれに続く文字列を int 型のデータに変換して対応する変数に格納する。文字列が例えば"-seed" と等しいかどうかの判定には、標準ライブラリ関数strcmp( ) を用いれば良い。

(プログラミング) 指示された処理を行う 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。

```
[motoki@x205a]$ nl func-commandline-param-random.c
     1 #include <stdio.h>
     2 #include <stdlib.h>
     3 #include <string.h>

     4 #define  WIDTH  6

     5 int main(int argc, char *argv[])
     6 {
     7   int  i, count=1;
     8   int  seed=1, max=RAND_MAX, num=10;

     9   for (i=1; i<argc; i+=2) {     /* コマンドライン指定の解読 */
    10     if (*argv[i] != '-') {
    11       printf("Invalid description: %s\n", argv[i]);
    12       exit(EXIT_FAILURE);
    13     }
    14     switch (*(argv[i]+1)) {
    15     case 's':
    16       if (strcmp(argv[i],"-seed")!=0
    17           || sscanf(argv[i+1],"%d",&seed)!=1 || seed<0) {
    18         printf("Invalid description: %s %s\n", argv[i], argv[i+1]);
    19         exit(EXIT_FAILURE);
    20       }
    21       break;
    22     case 'm':
    23       if (strcmp(argv[i],"-max")!=0
    24           || sscanf(argv[i+1],"%d",&max)!=1 || max<=0) {
    25         printf("Invalid description: %s %s\n", argv[i], argv[i+1]);
    26         exit(EXIT_FAILURE);
    27       }
```

```
    28       break;
    29     case 'n':
    30       if (strcmp(argv[i],"-num")!=0
    31           || sscanf(argv[i+1],"%d",&num)!=1 || num<=0) {
    32         printf("Invalid description: %s %s\n", argv[i], argv[i+1]);
    33         exit(EXIT_FAILURE);
    34       }
    35       break;
    36     default:
    37       printf("Invalid description: %s\n", argv[i]);
    38       exit(EXIT_FAILURE);
    39     }
    40   }

    41   srand(seed);
    42   for (i=0; i<num; ++i, ++count) {     /* 乱数の生成・出力 */
    43     if (max < RAND_MAX)
    44       printf("%12d", rand() % (max+1));
    45     else
    46       printf("%12d", rand());
    47     if (count >= WIDTH) {
    48       printf("\n");
    49       count = 0;
    50     }
    51   }
    52   if (count > 1)
    53     printf("\n");
    54   return 0;
    55 }
[motoki@x205a]$ gcc -o random func-commandline-param-random.c
[motoki@x205a]$ ./random
  1804289383   846930886  1681692777  1714636915  1957747793   424238335
   719885386  1649760492   596516649  1189641421
[motoki@x205a]$ ./random -seed 123 -max 99 -num 20
          93          13          73          30          79          31
          95          22          26           1          24          68
          21          91          16          81           9           7
          46          93
[motoki@x205a]$ ./random -aa 123
Invalid description: -aa
[motoki@x205a]$ ./random -seed aa
Invalid description: -seed aa
[motoki@x205a]$
```

ここで、

- プログラムの5行目が、コマンドラインからのパラメータ指定を可能にした部分である。主ルーチン main の引数をこの様に指定することによって、コマンドライン上の字句の個数が引数 argc に初期設定され、コマンドライン上の字句 (文字列) へのポインタが順に argv[0], argv[1], ..., argv[argc-1] に初期設定される。
- プログラム 12行目,19行目,26行目,36行目 の exit(EXIT_FAILURE) は、終了状態を EXIT_FAILURE (通常は 1 と定義されたマクロ; 0 以外なので異常終了を表す) としてプログラムを強制終了させることを表す。
- プログラム 16行目,23行目,32行目 に現われる strcmp は引数で与えられた 2 つの文字列を比較するライブラリ関数である。第 1 引数の文字列の方が辞書順の下で第 2 引数より小さいか、等しいか、大きいかに応じて、それぞれ、負、ゼロ、正の値を返す。
- プログラム 17行目,24行目,33行目 に現われる sscanf は、第 1 引数で指定した文字列から scanf と同様の書式付き入力を行なうためのライブラリ関数である。

10.5 関数を関数の引数として受渡しする方法

{ ケリー&ポール 6.15~6.16 節 }

関数を引数として受渡しする方法： 例えば、$\overset{(引数)}{\texttt{double}} \longrightarrow \overset{(値)}{\texttt{double}}$ の、すなわち、double 型の引数を受け取って double 型の値を返す関数を引数とする場合は、次のようにします。

- 仮引数部 ··· `double f(double)` または `double (*f)(double)` と書く。
  (コンパイラがポインタと解釈してくれる。) (明示的な書き方。)
- 引数の参照 ··· 仮引数が f だと、 `f(`囡`)` または `(*f)(`囡`)` と書く。
- 実引数 ··· 関数名だけを書く。(&は付けない。)

演習問題

□演習 10.1 (シンプソンの公式による数値積分)

(1) シンプソンの公式に基づいて定積分 $\int_a^b f(x)dx$ の近似値

$$\frac{h}{3}\Big[\ f(a_0)+f(a_{2n})+4(\ f(a_1)+f(a_3)+\cdots+f(a_{2n-3})+f(a_{2n-1})\ ) \\ +2(\ f(a_2)+f(a_4)+\cdots+f(a_{2n-2})\ )\ \Big]$$

但し、$h=\dfrac{b-a}{2n}$, $a_i=a+ih$

を計算する C の関数

```
double simpson(double f(double), double a, double b, int n)
```

を作成せよ。

(2) 解析学の演習書によれば、

$$\int_0^1 \frac{4}{1+x^2}dx=\int_0^2 \sqrt{4-x^2}dx=\int_0^{0.5}(12\sqrt{1-x^2}-\sqrt{27})dx=\pi$$

であることが示されているが、これらの定積分の近似値を問 (1) で作成した関数を用

いて色々と求めてみよ。例えば、$n = 10, n = 1000$ に対して、

simpson($\dfrac{4}{1+x^2}$, 0, 1, n),

simpson($\sqrt{4-x^2}$, 0, 2, n),

simpson($(12\sqrt{1-x^2} - \sqrt{27})$, 0, 0.5, n),

の値を比較してみよ。

11 構造体、共用体、typedef

- typedef —新しいデータ型を定義する機構—,
- 復習 構造体の定義, 構造体メンバーへのアクセス,
- 演算子の優先順位と結合性：まとめ,
- 例題：複素多項式の計算,
- 関数引数としての構造体,
- 自習 構造体の初期化,
- 共用体,
- 自習 ビットフィールド

11.1 typedef —新しいデータ型を定義する機構—

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 11.1 (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 11.2 (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。

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。

例題 11.3 (N 次元ベクトル空間の世界) 一般に、行列 A と (縦) ベクトル $\vec{x}, \vec{y}$ に対して $A(\vec{x}+\vec{y}) = A\vec{x} + A\vec{y}$ であるが、これを

$$A = \begin{pmatrix} 1.1 & -2.2 & 3.3 \\ -4.4 & 5.5 & -6.6 \\ 7.7 & -8.8 & 9.9 \end{pmatrix}, \qquad \vec{x} = \begin{pmatrix} 1.0 \\ 2.0 \\ 3.0 \end{pmatrix}, \qquad \vec{y} = \begin{pmatrix} -0.1 \\ 0.2 \\ -0.3 \end{pmatrix}$$

に対して確認するプログラムを作成せよ。 但し、新しいデータ型を定義したり、機能分割を適切に行ったりすることによって、プログラムは出来るだけ読み易く構成せよ。

(考え方) ここで出てくる基本的なデータは3次元ベクトルと3×3行列であるので、これらのためのデータ型にデータの意味を表す名前を付けると、プログラムが読み易くなる。そこで、例えば次の様にすれば良い。

```
#define N 3
typedef double Scalar;
typedef Scalar Vector[N];
typedef Scalar Matrix[N][N];
```

機能分割に関しては、$A(\vec{x}+\vec{y})$ と $A\vec{x}+A\vec{y}$ の計算をして各々の計算結果を出力する必要があることを考慮に入れなければならない。ベクトルの加算や線形変換の計算, 計算結果 (ベクトル) の表示を `main()` 関数の中で見通し良く記述するためには、これらの基本的な処理を例えば関数として定義したり、引数付きマクロとして定義すればよい。

(プログラミング) 問題の中で使われた名前に出来るだけ合わせるために、ベクトル $\vec{x}, \vec{y}, \vec{x}+\vec{y}, A(\vec{x}+\vec{y}), A\vec{x}, A\vec{y}, A\vec{x}+A\vec{y}$ を表す変数として各々 `x`, `y`, `x_y`, `Ax_y`, `Ax`, `Ay`, `Ax_Ay` 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。

```
[motoki@x205a]$ nl typedef-vector-space.c
     1 #include <stdio.h>

     2 #define  N  3

     3 #define  Print(title, vector)                \
     4            printf("%s\n", title);            \
     5            printf("   (%7.3f ", vector[0]);  \
     6            for (i=1; i<N; ++i)               \
     7              printf("%7.3f ", vector[i]);    \
     8            printf(")\n")

     9 typedef  double  Scalar;
    10 typedef  Scalar  Vector[N];
    11 typedef  Scalar  Matrix[N][N];

    12 void add(Vector x, Vector y, Vector z);          /* x=y+z */
    13 void linear_trans(Vector x, Matrix A, Vector y); /* x=A*y */
```

```
14 int main(void)
15 {
16   int     i;
17   Vector  x = {1.0, 2.0, 3.0}, y = {-0.1, 0.2, -0.3},
18           x_y, Ax_y, Ax, Ay, Ax_Ay;
19   Matrix  A = {{1.1, -2.2, 3.3},
20                {-4.4, 5.5, -6.6},
21                {7.7, -8.8, 9.9}};    /* これらのA,x,yに関して */
22                                      /* A*(x+y)=A*x+A*yを確認 */
23   add(x_y, x, y);
24   linear_trans(Ax_y, A, x_y);
25   Print("A*(x+y) =", Ax_y);

26   linear_trans(Ax, A, x);
27   linear_trans(Ay, A, y);
28   add(Ax_Ay, Ax, Ay);
29   Print("A*x+A*y =", Ax_Ay);
30   return 0;
31 }

32 void add(Vector x, Vector y, Vector z)
33 {
34   int  i;

35   for (i=0; i<N; ++i)
36     x[i] = y[i]+z[i];
37 }

38 void linear_trans(Vector x, Matrix A, Vector y)
39 {
40   int  i, k;

41   for (i=0; i<N; ++i) {
42     x[i] = 0.0;
43     for (k=0; k<N; ++k)
44       x[i] += A[i][k] * y[k];
45   }
46 }
[motoki@x205a]$ gcc typedef-vector-space.c
[motoki@x205a]$ ./a.out
A*(x+y) =
   (  5.060  -9.680  14.300 )
A*x+A*y =
```

```
    (  5.060  -9.680  14.300 )
[motoki@x205a]$
```

ここで、

- プログラム 3~8 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 (25 行目,29 行目) に現われる `Print(`*title*`,` *vector*`)` という形の文字列は全てコンパイル前に 4~8 行目の文字パターンで置き換えられることになる。3~7 行目右端のバックスラッシュ(\) は指令が次の行に続くことをプリプロセッサに知らせている。
- プログラム 9~11 行目 でデータ型 `Scalar`, `Vector`, `Matrix` が新しく定義されているので、例えばプログラム 12 行目に現われる

  ```
  Vector x
  ```

  は、10 行目の 「`Scalar Vector[N]`」 内の文字列 `Vector` を `x` で置き換えたもの

  ```
  Scalar x[N]
  ```

  と同等、さらにこれは 9 行目の 「`double Scalar`」内の文字列 `Scalar` を `x[N]` で置き換えたもの

  ```
  double x[N]
  ```

  と同等、さらにこれは 2 行目の `#define` 行より

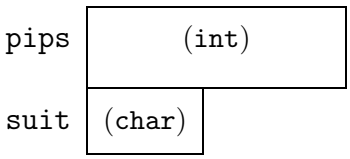
  ```
  double x[3]
  ```

  と同等ということになる。

11.2 復習 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。(Pascal の場合は**レコード**と呼んでいた。) 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 11.4 (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



`pips` … 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。)

`suit` … `'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。(例えば、下の例 11.7)

11.3 復習 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 11.5 (構造体要素へのアクセス; トランプのカード) 先の例 11.4 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 11.6 (配列を構成要素とする構造体) 構造体

```
struct person {
   int id;
   char name[40];
   long phone;
};
```

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。）

11.4 演算子の優先順位と結合性：まとめ

{ケリー&ポール 9.4 節, 付録 D}

優先順位高 ↑

演算子	結合性
関数の引数をくくる丸括弧 ( ) 配列添字をくくる四角括弧 [] メンバアクセス演算子 -> .	左から右
+ (単項) - (単項) ++ -- sizeof( ) ! キャスト ビット反転~ 間接演算子* 番地演算子&	右から左
* / %	左から右
+ -	左から右
左シフト<< 右シフト>>	左から右
< <= > >=	左から右
== !=	左から右
ビット積&	左から右
ビット排他的和^	左から右
ビット和|	左から右
&&	左から右
||	左から右
条件演算子 条件 ? 式 : 式	右から左
= += -= *= /=	右から左
コンマ演算子,	左から右

11.5 例題：複素多項式の計算

複素数についての復習： 複素数については知っていると思いますが、次の例 11.7 を読むための予備知識として念のためおさらいしておきます。

- **純虚数** $\sqrt{-1}$ を i で表す。
- 複素数は次の様な形で表せる。この時の x を実部、 y を虚部という。

$$x + iy \qquad \text{(但し、} x \text{ と } y \text{ は実数、} i \text{ は純虚数。)}$$

- ２つの複素数 $z_1 = x_1 + iy_1$ と $z_2 = x_2 + iy_2$ の和は
$$\begin{aligned} z_1 + z_2 &= (x_1 + iy_1) + (x_2 + iy_2) \\ &= (x_1 + x_2) + i(y_1 + y_2) \end{aligned}$$
- ２つの複素数 $z_1 = x_1 + iy_1$ と $z_2 = x_2 + iy_2$ の積は
$$\begin{aligned} z_1 z_2 &= (x_1 + iy_1)(x_2 + iy_2) \\ &= (x_1 x_2 - y_1 y_2) + i(x_1 y_2 + y_1 x_2) \end{aligned}$$

例題 11.7 (複素多項式の計算)　非負整数データ n と複素数データ $C_n, C_{n-1}, ..., C_0$ を読み込み、これらの定数値の下で複素多項式 $C_n z^n + C_{n-1} z^{n-1} + C_{n-2} z^{n-2} + \cdots + C_1 z + C_0$ の値が
$$z = e^{i\pi k/5} = \cos\tfrac{\pi k}{5} + i \sin\tfrac{\pi k}{5} \qquad (k = 0, 1, 2, 3, ..., 9)$$
のそれぞれの値に対してどの様に変化するかを、表の形に見易く出力するC プログラムを作成せよ。

(設計方針)

- この問題の場合、複素数を基本的なデータとして扱うことが出来れば計算処理を実数型の多項式の計算と全く同じ様に進めることが出来る。そこで、複素数を構造体で表し、そのデータ型に `Complex` という名前を付ける。
- 多項式の係数 $C_n, C_{n-1}, ..., C_0$ を保持するために (`Complex` 型) 配列を用意するのが妥当である。この配列領域を宣言によって確保する場合その大きさはコンパイル時に確定してなければならないので、多項式の最大次数をマクロで設定して、実際の処理では確保した係数用の配列領域の一部だけを使う。
- 多項式の計算を効率的に行うために、(例えば「川合,(岩波講座ソフトウェア科学 2) プログラミングの方法, 岩波書店」p.145 ~146 で説明されている)**Horner の方法**に従って
$$(((C_n z + C_{n-1})z + C_{n-2})z + \cdots + C_1)z + C_0$$
という順序で計算する。
- 複素多項式の計算をプログラム上で実多項式の場合と全く同じ様に見通し良く記述したい。 そのために、
　　複素数同士の加算をして、その結果を値として返す関数 `sum_of()` と
　　複素数同士の乗算をして、その結果を値として返す関数 `product_of()`
を用意する。

(プログラミング)　上で説明した設計方針に沿って構成した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。

```
[motoki@x205a]$ nl struct-complex-polynomial.c
     1 /**********************************/
     2 /*                                */
     3 /* Horner 法による複素多項式の計算 */
     4 /*                                */
     5 /**********************************/

     6 #include <stdio.h>
```

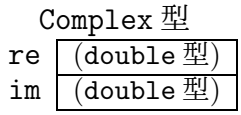
```
 7 #include <math.h>

 8 #define  MAX_DEGREE  100              /* 複素多項式の次数の上限 */
 9 #define  PI  3.1415926535897932       /* 円周率                 */

10 typedef struct{             /*--- 複素数の構造体 ---*/
11     double re;              /* 実部, real part      */
12     double im;              /* 虚部, imaginary part */
13 } Complex;

14 Complex sum_of(Complex z1, Complex z2);    /*引数の和を計算して返す*/
15 Complex product_of(Complex z1, Complex z2);/*引数の積を計算して返す*/

16 /*-------------------------------------------------------*/
17 /* 主プログラム                                          */
18 /*-------------------------------------------------------*/
19 /*   多項式の次数と複素係数を読み込み、                  */
20 /*   色々な変数値 z に対して多項式の値を表の形に表示する。 */
21 /*-------------------------------------------------------*/
22 int main(void)
23 {
24   int degree, i, k;
25   Complex c[MAX_DEGREE+1], /* 多項式の係数を入れる配列     */
26           z,               /* 多項式の変数 z を入れる領域 */
27           result;          /* 計算結果を溜めていく領域     */

28   printf("Input the degree(<=100) of the polynomial: ");
29   scanf("%d", &degree);
30   for (i=degree; i>=0; --i) {
31     printf("  Input the real and the imaginary part"
32            " of the %d-th coefficient: ", i);
33     scanf("%lf %lf", &c[i].re, &c[i].im);
34   }

35   printf("\ndegree = %d\n", degree);
36   for (i=degree; i>=0; --i)
37     printf("  c[%d] = (%e)+(%e)i\n", i, c[i].re, c[i].im);
38   printf("\n k  %16sz%15s  c[d]*z^d+c[d-1]*z^(d-1)+ ... +c[1]*z+c[0]\n"
39          "--  ------------------------------"
40          "  ----------------------------------------\n", "", "");

41   for (k=0; k<10; ++k) {
42     z.re = cos(PI*k/5);
```

```
    z.im = sin(PI*k/5);
    result = c[degree];
    for (i=degree-1; i>=0; --i)
      result = sum_of(product_of(result, z), c[i]);
    printf("%2d  (%13.6e)+(%13.6e)i  (%13.6e)+(%13.6e)i\n",
           k, z.re, z.im, result.re, result.im);
  }
  return 0;
}

/*-------------------------------------------------------------*/
/* 複素数の和を求めて返す関数 sum_of                           */
/*-------------------------------------------------------------*/
/*   (仮引数) z1 : 複素数                                      */
/*            z2 : 複素数                                      */
/*   (関数値) : z1+z2 の値                                     */
/*-------------------------------------------------------------*/
Complex sum_of(Complex z1, Complex z2)
{
    Complex result;

    result.re = z1.re+z2.re;
    result.im = z1.im+z2.im;
    return result;
}

/*-------------------------------------------------------------*/
/* 複素数の積を求めて返す関数 product_of                       */
/*-------------------------------------------------------------*/
/*   (仮引数) z1 : 複素数                                      */
/*            z2 : 複素数                                      */
/*   (関数値) : z1*z2 の値                                     */
/*-------------------------------------------------------------*/
Complex product_of(Complex z1, Complex z2)
{
    Complex result;

    result.re = z1.re*z2.re - z1.im*z2.im;
    result.im = z1.re*z2.im + z1.im*z2.re;
    return result;
}
[motoki@x205a]$ gcc struct-complex-polynomial.c -lm
[motoki@x205a]$ ./a.out
```

```
Input the degree(<=100) of the polynomial: 3
  Input the real and the imaginary part of the 3-th coefficient: 1.0 -2.0
  Input the real and the imaginary part of the 2-th coefficient: -3.0 4.0
  Input the real and the imaginary part of the 1-th coefficient: 5.0 -6.0
  Input the real and the imaginary part of the 0-th coefficient: -7.0 8.0

degree = 3
  c[3] = (1.000000e+00)+(-2.000000e+00)i
  c[2] = (-3.000000e+00)+(4.000000e+00)i
  c[1] = (5.000000e+00)+(-6.000000e+00)i
  c[0] = (-7.000000e+00)+(8.000000e+00)i

 k                  z                   c[d]*z^d+c[d-1]*z^(d-1)+ ... +c[1]*z+c[0]
--  ---------------------------------  ------------------------------------------
 0  ( 1.000000e+00)+( 0.000000e+00)i  (-4.000000e+00)+( 4.000000e+00)i
 1  ( 8.090170e-01)+( 5.877853e-01)i  (-2.566385e+00)+( 6.036813e+00)i
 2  ( 3.090170e-01)+( 9.510565e-01)i  (-1.657253e+00)+( 6.932006e+00)i
 3  (-3.090170e-01)+( 9.510565e-01)i  ( 1.572893e+00)+( 1.093085e+01)i
 4  (-8.090170e-01)+( 5.877853e-01)i  (-2.430068e+00)+( 2.021529e+01)i
 5  (-1.000000e+00)+( 1.224606e-16)i  (-1.600000e+01)+( 2.000000e+01)i
 6  (-8.090170e-01)+(-5.877853e-01)i  (-2.089617e+01)+( 6.728984e+00)i
 7  (-3.090170e-01)+(-9.510565e-01)i  (-1.219093e+01)+(-9.308531e-01)i
 8  ( 3.090170e-01)+(-9.510565e-01)i  (-6.016509e+00)+( 2.123722e+00)i
 9  ( 8.090170e-01)+(-5.877853e-01)i  (-5.815581e+00)+( 3.963187e+00)i
[motoki@x205a]$
```

ここで、

- プログラムの 10~13 行目 が、複素数を構造体で表しそのデータ型に Complex という名前を付けている部分である。 この定義によれば、この構造体は re という名前の double 型メンバと im という名前の double 型メンバにより構成される。



- プログラム 25~27 行目 は Complex 型の配列 c と変数 z, result の宣言を行なっている。
- プログラム 33 行目 の c[i].re, c[i].im は Complex 型の配列要素 c[i] の中の各々 re, im という名前の付いた小区画 (すなわちメンバ) を表している。 (37 行目, 48 行目, 61~62 行目, 75~76 行目 についても同様。)
- プログラム 14~15 行目 の関数プロトタイプから分かるように、**構造体を関数の引数として用いることも関数値として用いることも出来る。**

11.6 関数引数としての構造体

構造体を関数の引数として渡すことも、関数値として返すことも可能である。しかし、

- 構造体を関数引数として引数結合する際、および
- 構造体を関数値として変数に代入する際

には構造体が丸ごとコピーされることになる。

$\Longrightarrow$ 大きな構造体データを受渡したい場合は、計算効率の低下を防ぐために、参照呼び出し (i.e. 構造体へのポインタの受渡し) を使うべき。
[例題 11.3 で例示した引数付きマクロを関数の代わりに使うという手もある。]

11.7 自習 構造体の初期化

{ ケリー&ポール 9.6 節 }

配列の場合と同様の書き方で構造体の初期化を行なうことが出来る。

例 11.8 (構造体の初期化) 宣言

```
typedef struct {
  char suit;
  int  pips;
}Card;
Card  c={'h', 13};
```

によって、 `c.suit='h'`, `c.pips=13` と初期設定され、宣言

```
typedef struct {
  char *name;
  int  calories;
}Fruit;
Fruit apple={"apple", 150};
```

によって、 `apple.name="apple"`, `apple.calories=150` と初期設定される。また、宣言

```
typedef struct {
  double re;
  double im;
}Complex;
Complex a[3][2] = {
  {{1.0, 2.0}, {3.0}},
  {{5.0, 6.0}, {7.0, 8.0}}
};
```

によって、次のように初期設定される。[指定されなかった `a[2]` と `a[0][1].im` については暗黙にゼロクリアされる。]

```
a[0][0].re=1.0,  a[0][0].im=2.0,  a[0][1].re=3.0,  a[0][1].im=0.0,
a[1][0].re=5.0,  a[1][0].im=6.0,  a[1][1].re=7.0,  a[1][1].im=8.0,
a[2][0].re=0.0,  a[2][0].im=0.0,  a[2][1].re=0.0,  a[2][1].im=0.0
```

11.8 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 11.9 (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 つの変数に記憶されたデータを `float` 型と見たり `int` 型と見たりする訳だから、共用体を使えば良い。

`i` / `f` ⟩ (int 型 / float 型) ⋯ { ある時は `int` 型のデータが入っていると見る / ある時は `float` 型のデータが入っていると見る

また、`int` 型の値を 16 進表示するためには、単に `printf()` の書式指定の中で "`%x`" という変換指定をしてやれば良い。

(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。

```
[motoki@x205a]$ nl struct-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 struct-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。

11.9 自習 ビットフィールド

- 構造体や共用体の中の int 型,unsigned 型のメンバは、ビット長を指定することが出来る。(こういうメンバをビットフィールドという。)
  $\Longrightarrow$ コンパイラは、それらを最小限のワードに詰め込む。

例題 11.10 (ビットフィールド) 実数型データを float 型で読み込み、そのビット列を 2 進表示する C プログラムを作成せよ。

(考え方) 読み込んだデータを構成するビット列をビット毎 (あるいは数ビット毎) に調べることが出来れば、上の桁から順に調べてその結果に応じて対応する 2 進文字列を出力することが出来る。 そこで、読み込んだ float 型データを 4 ビット毎に調べられる様に、読み込んだデータを格納する変数を float 型と int 型の共用体として用意し、その共用体の中の int 型メンバは更に細かく 4 ビットのビットフィールドの集まりとして構成する。実際、この様にしておくと、4 ビット毎にその (整数としての) 値を調べることが出来るから、あとはその 16 種類の値の各々に対して 16 種類の文字列 0000, 0001, ..., 1111 の中の対応する文字列を出力するだけである。

(プログラミング) 指示された処理を行う 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。

```
[motoki@x205a]$ nl struct-union-bit-field.c
     1 /*-------------------------------------------------------*/
     2 /* 実数型データを float 型で読み込み、                   */
     3 /* そのビット列を 4bit の束の列と見て 2 進表示する C プログラム */
     4 /*-------------------------------------------------------*/

     5 #include <stdio.h>

     6 typedef struct {
     7   unsigned  b0:4, b1:4, b2:4, b3:4, b4:4, b5:4, b6:4, b7:4;
     8 }Nibble_seq;

     9 typedef union {
    10   float      f;          /* 4 byte を仮定 */
    11   int        i;          /* 4 byte を仮定 */
    12   Nibble_seq nibble;
    13 }Number;

    14 typedef  char  *String;

    15 int main(void)
    16 {
    17   Number  num;
    18   String  bit_seq[16] = {
    19     "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
    20     "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"
    21   };

    22   scanf("%f", &num.f);
    23   printf("Float number %g and int number %d are\n"
    24          "commonly represented by a bit sequence\n"
    25          "    %s%s %s%s %s%s %s%s.\n",
```

```
    26          num.f, num.i,
    27          bit_seq[num.nibble.b7], bit_seq[num.nibble.b6],
    28          bit_seq[num.nibble.b5], bit_seq[num.nibble.b4],
    29          bit_seq[num.nibble.b3], bit_seq[num.nibble.b2],
    30          bit_seq[num.nibble.b1], bit_seq[num.nibble.b0]);
    31   return 0;
    32 }
[motoki@x205a]$ gcc struct-union-bit-field.c
[motoki@x205a]$ ./a.out
1.0
Float number 1 and int number 1065353216 are
commonly represented by a bit sequence
    00111111 10000000 00000000 00000000.
[motoki@x205a]$
```

ここで、

- プログラム 6~8 行目 で、4 ビットの `unsigned int` フィールド 8 個で構成される構造体が定義され、それに `Nibble_seq` というデータ型名が付けられている。

> 注意：
> 計算機の種類によっては、プログラム 27~30 行目の出力の順序を逆に、すなわち `bit_seq[num.nibble.b0]`, `bit_seq[num.nibble.b1]`, `bit_seq[num.nibble.b2]`, ..., `bit_seq[num.nibble.b7]` という順序にしなければならない。

演習問題

□演習 11.1 (構造体の定義) レストラン (の情報) が レストラン名, 所在地, 代表メニュー 1 点, 平均価格 で表せるとして、個々のレストランを表す C の構造体を定義してみよ。

□演習 11.2 (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
```

□**演習 11.3 (関数引数としての構造体; 複素多項式の計算)** 先の例題 11.7(複素多項式の計算) において、多項式の次数と係数を 1 つの構造体 Polynomial としてまとめ上げ、入力部を関数にすると、下のようなプログラムになる。このプログラムは大きな構造体をそのまま関数値としているため、計算結果の受渡しの際にコピーの手間が必要になる。そこで、計算結果を入れる番地を関数の引数として受け渡すようにこのプログラムを修正してみよ。

```
      (冒頭の見出し,#include 宣言は省略)
 8  #define  MAX_DEGREE  100              /* 複素多項式の次数の上限 */
 9  #define  PI  3.1415926535897932       /* 円周率                 */

10  typedef struct {            /*--- 複素数の構造体 ---*/
11      double re;              /* 実部, real part      */
12      double im;              /* 虚部, imaginary part */
13  } Complex;

14  typedef struct {                /*--- 多項式の構造体 ---*/
15      int     degree;             /* 次数, degree         */
16      Complex coef[MAX_DEGREE+1]; /* 係数, coefficient    */
17  } Polynomial;

18  Polynomial input(void);             /*多項式の次数と係数を入力*/
19  Complex    sum_of(Complex z1, Complex z2);      /*引数の和を返す*/
20  Complex    product_of(Complex z1, Complex z2);  /*引数の積を返す*/

21  /*-------------------------------------------------------------*/
22  /* 主プログラム                                                */
23  /*-------------------------------------------------------------*/
24  /*   多項式の次数と複素係数を読み込み、                        */
25  /*   色々な変数値 z に対して多項式の値を表の形に表示する。  */
26  /*-------------------------------------------------------------*/
27  int main(void)
28  {
29    Polynomial  poly;        /* 多項式の次数と係数を入れる構造体 */
30    Complex     z,           /* 多項式の変数 z を入れる領域 */
31                result;      /* 計算結果を溜めていく領域    */
32    int i, k;

33    poly = input();
34    printf("\ndegree = %d\n", poly.degree);
35    for (i=poly.degree; i>=0; --i)
36      printf("  c[%d] = (%le, %le)\n",
                i, poly.coef[i].re, poly.coef[i].im);
37    printf("\n k%18sz%16s c[d]*z^d+c[d-1]*z^(d-1)+ ... +c[1]*z+c[0]\n"
38           "--  ------------------------------"
39           "  ----------------------------------------\n", "", "");
```

```
40    for (k=0; k<10; ++k) {
41      z.re = cos(PI*k/5);
42      z.im = sin(PI*k/5);
43      result = poly.coef[poly.degree];
44      for (i=poly.degree-1; i>=0; --i)
45        result = sum_of(product_of(result, z), poly.coef[i]);
46      printf("%2d  (%14.7le, %14.7le)  (%14.7le, %14.7le)\n",
47             k, z.re, z.im, result.re, result.im);
48    }
49    return 0;
50  }

51  /*-------------------------------------------------------------*/
52  /* 複素多項式の次数と係数を入力する関数 input                  */
53  /*-------------------------------------------------------------*/
54  /*   (仮引数) : なし                                           */
55  /*   (関数値)   : 複素多項式の次数と係数から成る構造体        */
56  /*-------------------------------------------------------------*/
57  Polynomial input(void)
58  {
59    Polynomial  p;
60    int         i;

61    printf("Input the degree(<=100) of the polynomial: ");
62    scanf("%d", &p.degree);
63    for (i=p.degree; i>=0; --i) {
64      printf("  Input the real and the imaginary part"
65             " of the %d-th coefficient: ", i);
66      scanf("%lf %lf", &p.coef[i].re, &p.coef[i].im);
67    }
68    return p;
69  }

70  /*-------------------------------------------------------------*/
71  /* 複素数の和を求めて返す関数 sum_of                           */
72  /*-------------------------------------------------------------*/
73  /*   (仮引数) z1 : 複素数                                      */
74  /*              z2 : 複素数                                    */
75  /*   (関数値)      : z1+z2 の値                               */
76  /*-------------------------------------------------------------*/
77  Complex sum_of(Complex z1, Complex z2)
```

(途中省略; 例 11.7 と同じ)

```
84  /*-------------------------------------------------------------*/
85  /* 複素数の積を求めて返す関数 product_of                       */
86  /*-------------------------------------------------------------*/
87  /*   (仮引数) z1 : 複素数                                      */
88  /*              z2 : 複素数                                    */
89  /*   (関数値)      : z1*z2 の値                               */
90  /*-------------------------------------------------------------*/
91  Complex product_of(Complex z1, Complex z2)
```

(以下省略; 例 11.7 と同じ)

□**演習 11.4 (関数引数としての構造体)** 上の演習 11.3 の関数 input() を例題 11.3 に倣って引数付きマクロで表してみよ。

□**演習 11.5 (ビットフィールド)** 例題 11.10 のプログラムを実習室の計算機上で実行するとどの様な結果になるか？ (下の様な結果になるのではないか？) 例 11.10 の実行結果と違うとすると、それは何故か？

```
sv01_41: gcc struct-union-bit-field.c
sv01_42: a.out
1.0
Float number 1 and int number 1065353216 are
commonly represented by a bit sequence
    00000000 00000000 00001000 11110011.
sv01_43:
```

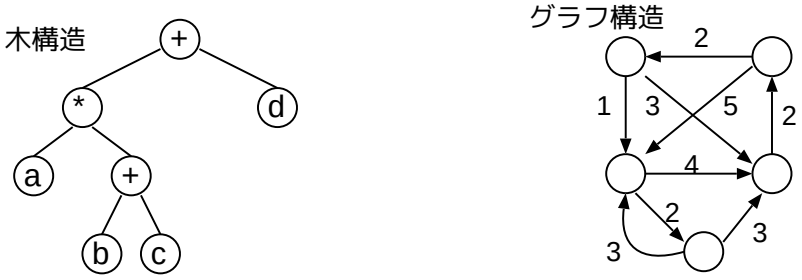
12　動的データ構造

- 動的データ構造, 自己参照的構造体,
- 線形リスト,
- 2 分木,2 分木の走査

12.1　動的データ構造

変数宣言によって確保された配列や構造体等のデータ記憶領域はそれぞれ塊になっており、そのデータ構造 (i.e. データを記憶し操作するための表現形式) は実行の途中に形や大きさを変更することは出来ない。このように物理的な構成が固定されたデータ構造を**静的データ構造**という。これに対して、小さめの記憶領域を**動的に** (i.e. 実行中に) 確保出来れば、それらをポインタでつなげて色々な形／大きさのデータ構造を表すことができ、また実行中にも (i.e. 動的に) 形や大きさを変えることが出来る。このように物理的な構成が可変であるデータ構造を**動的データ構造**という。

例 12.1 動的データ構造を用いれば、次のような形にデータを配置し結ばれた線に沿ってデータをたどることも出来る。

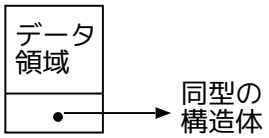


スタック領域とヒープ領域：　講義ノート 7.5 節で述べた様に、関数の本体部 (あるいはブロック) の最初に宣言された変数／配列 (のためのメモリ) は、関数が呼ばれたり新しいブロックに入ったりする度に**スタック領域**上に確保される。一方、計算機内部では、動的な記憶領域確保の要求に応えるための領域 (**ヒープ領域**という) が用意されており、`malloc` や `calloc` といったライブラリ関数の呼び出しに対してこの中の小領域を割り当てたり、`free` というライブラリ関数の呼び出しに対して不要になった小領域を再利用可能なメモリとして登録し直したり、といった作業がなされる。

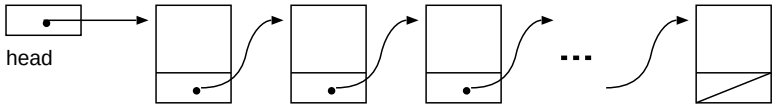
12.2　自己参照的構造体

自己参照的構造体：　自分と同型の構造体へのポインタをメンバに持つ構造体を**自己参照的構造体**という。

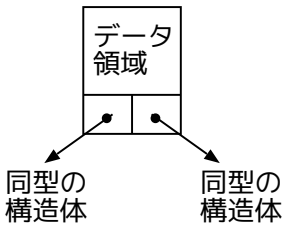
- 「自分と同型の構造体へのポインタ」を 1 個だけ持つ構造体は次のような構成をしている。



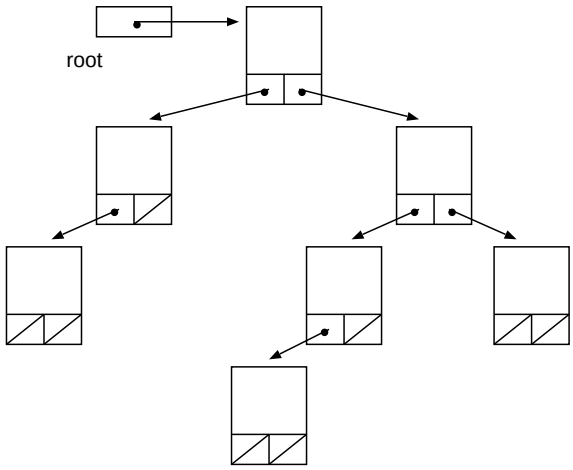
従って、この種の構造体からは次のような形の (動的) データ構造を構成することが出来る。このような構造を**線形リスト** (linear list) または**連結リスト** (linked list) という。



- 「自分と同型の構造体へのポインタ」を 2 個だけ持つ構造体は次のような構成をしている。



従って、この種の構造体からは次のような形の (動的) データ構造を構成することが出来る。このような構造を **2 分木**という。



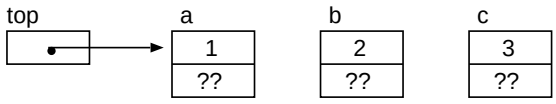
例 12.2 (自己参照的構造体) 小さな線形リストを構成する C プログラムを次に示す。

```
[motoki@x205a]$ nl dynamic-build-small-llist.c
     1 #include <stdio.h>

     2 struct node {
     3   int          data;
     4   struct node  *next;
     5 };

     6 int main(void)
     7 {
     8   struct node  *top, a, b, c;
```

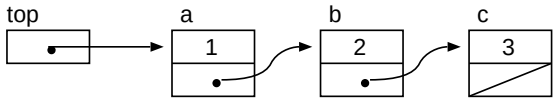
```
 9   a.data = 1;
10   b.data = 2;
11   c.data = 3;

12   top    = &a;
13   a.next = &b;
14   b.next = &c;
15   c.next = NULL;

16   printf("%3d%3d%3d\n", top->data, top->next->data,
17          top->next->next->data);
18            /* a,b,c という変数名を使わずにアクセスできる */
19   return 0;
20 }
[motoki@x205a]$ gcc dynamic-build-small-llist.c
[motoki@x205a]$ ./a.out
  1  2  3
[motoki@x205a]$
```

ここで、

- プログラム 13 行目 を実行する直前の時点では、各変数の状況は次の様になっている。

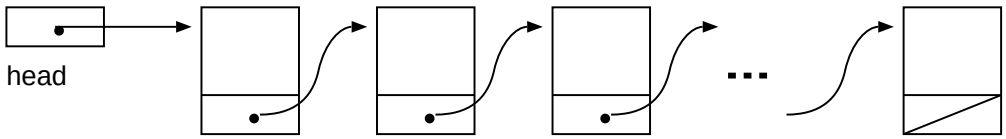


- プログラム 16 行目 の時点では、各変数の状況は次の様になっている。



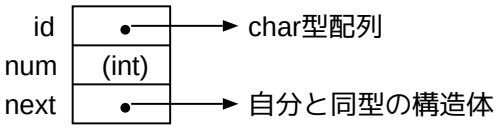
12.3 線形リスト

この節では線形リスト、すなわち次のような形のデータ構造の扱い方を例示する。



例題 12.3 (線形リスト)
① 入力ストリームに現れる 識別子 (1個以上の空白) 整数 という形のデータを読み込んでは、
② それを 識別子 に関して辞書順になる様に線形リストに登録する、
という作業を繰り返し、最後に線形リストに登録されたものを順に出力するプログラムを作成せよ。

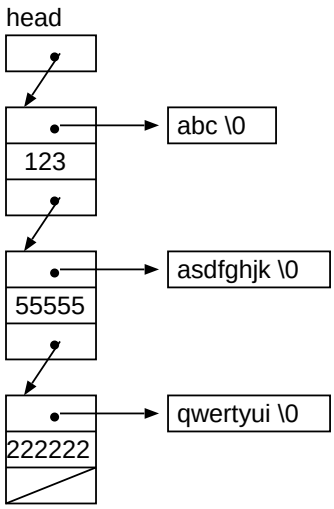
(考え方) 識別子 は英字で始まり英数字の続く文字列のことで、その長さに上限はない。従って、読み込んだ識別子全体を構造体の中に保持することは出来ない。そこで、線形リストを構成する自己参照的構造体として次の様なものを考える。



すると、例えば

```
asdfghjk 55555
qwertyui 222222
abc      123
```

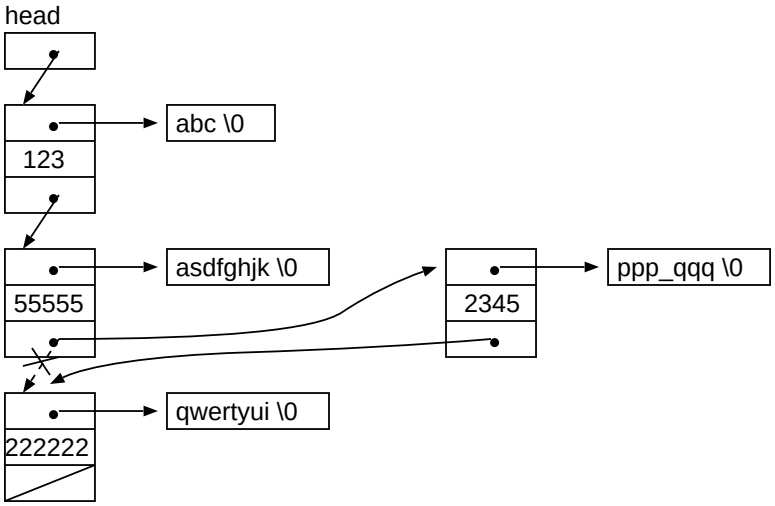
という入力に対して、次の様な線形リストが構成されることになる。



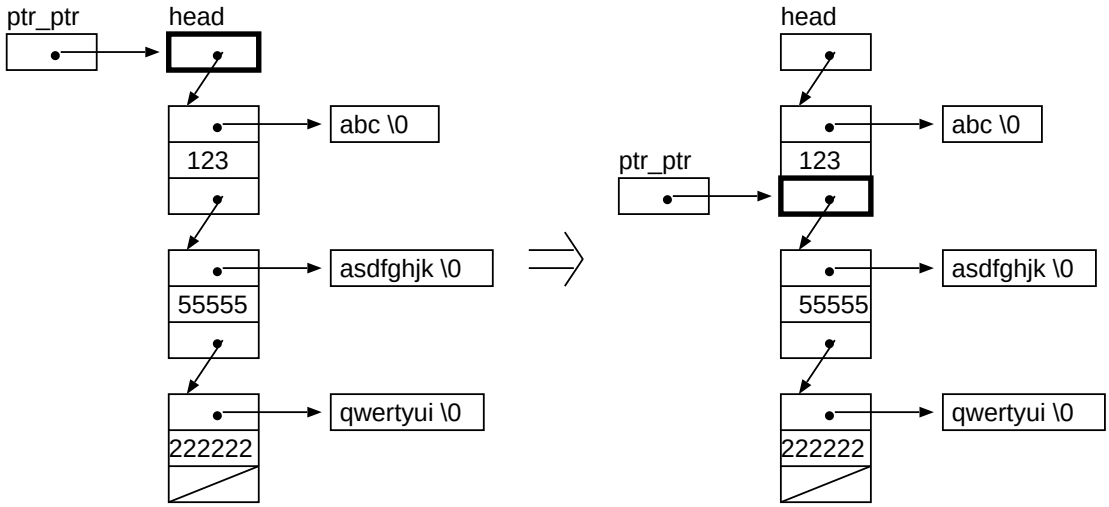
ここでさらに、

```
ppp_qqq  2345
```

という入力があれば、挿入位置の探査、必要なメモリの確保等を行なった上で次のようなポインタの付け替えを行えば良い。



「挿入位置の探査」を行うには、次の様に線形リストを先頭から順にたどり、各々の時点で、この次が新しい要素の挿入場所かどうかを標準ライブラリ関数 `strcmp()` を用いて判定すれば良い。



「必要なメモリの確保」を実行時に動的に行うには、標準ライブラリ関数 `malloc()` を用いれば良い。

(プログラミング) プログラムの見通しを良くするために、入力した識別子と整数の組を(辞書順を保つ様に)線形リストに挿入する機能を果たす関数 `add_item()` と、線形リストとして繋げられたデータを先頭から順に出力する機能を果たす関数`listprint()` を用意した。 そして、簡単のため、入力したものが本当に識別子になっているかどうかのチェックは省略してプログラムを構成した。このプログラムと、これをコンパイル/実行している様子を次に示す。

```
[motoki@x205a]$ nl dynamic-llist.c
     1 /*********************************************************/
     2 /*                                                       */
     3 /* 線形リストを用いて識別子 (と整数) を辞書順に登録・出力 */
     4 /*                                                       */
     5 /*********************************************************/
     6
```

```
 7 #include <stdio.h>
 8 #include <string.h>
 9 #include <stdlib.h>
10
11 #define  MAXLENGTH  100
12
13 #define Is_empty(list)  ((list) == NULL)
14
15 typedef char *String;
16
17 typedef  struct list_item  *List;
18 typedef struct list_item {
19     String id;
20     int    num;
21     List   next;
22 } List_item;
23
24 void add_item(List *ptr_ptr, String id, int num);
25 void listprint(List);
26
27 int main(void)
28 {
29   List head;
30   char identifier[MAXLENGTH+1];
31   int  count, num;
32
33   head = NULL;
34   while ((count=scanf("%100s %d", identifier, &num)) == 2)
35     add_item(&head, identifier, num);
36
37   if (count != EOF) {
38     printf("Input Error!\n");
39     exit(EXIT_FAILURE);
40   }
41
42   listprint(head);
43   return 0;
44 }
45
46 /*-------------------------------------------------------------*/
47 /* 線形リストへ識別子と整数の組を追加                          */
48 /*-------------------------------------------------------------*/
49 /* (仮引数) ptr_ptr : "線形リストへのポインタ領域"へのポインタ  */
```

```
50 /*          id      : 文字列データへのポインタ                    */
51 /*          num     : 整数                                        */
52 /* (関数値) : なし                                                */
53 /* (機能)   : 識別子と整数の組を 1 つの項目として、それらが識別子 */
54 /*            の辞書順に線形リストの形に繋げられており、          */
55 /*            *ptr_ptr がその線形リストの先頭を指し示すと仮定す */
56 /*            る。この仮定の下で、id の指し示す識別子と整数 num */
57 /*            の組を新しい項目として（辞書順を保つ様に）線形リ   */
58 /*            ストに挿入する。                                    */
59 /*----------------------------------------------------------------*/
60 void add_item(List *ptr_ptr, String id, int num)
61 {
62   List_item *new_item;
63
64   while (! Is_empty(*ptr_ptr) && strcmp(id, (*ptr_ptr)->id) > 0)
65     ptr_ptr = &((*ptr_ptr)->next);            /* 挿入場所を探す */
66
67   new_item = (List_item *) malloc(sizeof(List_item));
68                                             /* 新しい item 枠を作る */
69   if (new_item == NULL) {
70     printf("*** fail in memory allocation ***");
71     exit(EXIT_FAILURE);
72   }
73
74   new_item->id = (String) malloc(strlen(id)+1);   /*id を入れる領*/
75   if (new_item->id == NULL) {                     /*域を確保し、*/
76     printf("*** fail in memory allocation ***"); /*そこへのポイ*/
77     exit(EXIT_FAILURE);                           /*ンタを代入  */
78   }
79   strcpy(new_item->id, id);                  /* 新領域に id を複写 */
80
81   new_item->num = num;
82
83   new_item->next  = *ptr_ptr;              /* ポインタの付け替え */
84   *ptr_ptr = new_item;
85 }
86
87 /*----------------------------------------------------------------*/
88 /* 線形リストの中味を出力                                         */
89 /*----------------------------------------------------------------*/
90 /* (仮引数) ptr : 線形リストへのポインタ                          */
91 /* (関数値) : なし                                                */
92 /* (機能)   : 識別子と整数の組を 1 つの項目として、それらが識別子 */
```

```
 93 /*            の辞書順に線形リストの形に繋げられており、ptr が  */
 94 /*            その線形リストの先頭を指し示すと仮定する。        */
 95 /*            この仮定の下で、線形リストに記憶されたデータを    */
 96 /*                  番号  整数  識別子                          */
 97 /*            という書式で出力する。                            */
 98 /*--------------------------------------------------------------*/
 99 void listprint(List ptr)
100 {
101   int n;
102
103   printf("番号        整数  識別子\n");
104   for (n=1; ! Is_empty(ptr); ptr = ptr->next, ++n)
105     printf("%4d%11d  %s\n", n, ptr->num, ptr->id);
106 }
[motoki@x205a]$ cat dynamic-llist.data
asdfghjk 55555
qwertyui 222222
abc      123
ppp_qqq  2345
zxcv     987
kkk      456
efghi    77
[motoki@x205a]$ gcc dynamic-llist.c
[motoki@x205a]$ ./a.out <dynamic-llist.data
番号        整数  識別子
   1         123  abc
   2       55555  asdfghjk
   3          77  efghi
   4         456  kkk
   5        2345  ppp_qqq
   6      222222  qwertyui
   7         987  zxcv
[motoki@x205a]$
```

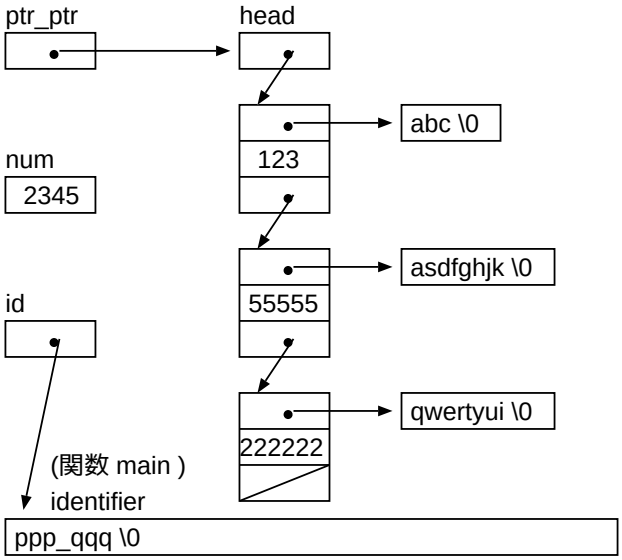
ここで、

- プログラム 13 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、以降に現われる `Is_empty(` *list* `)` という形の文字列は全てコンパイル前に `((` *list* `)␣==␣NULL)` という文字パターンで置き換えられることになる。
- プログラム 17~22 行目 では、構造体を自己参照的に定義し、それにタグ名 `list_item` とデータ型名 `List_item` を付け、その構造体を指すポインタのデータ型の総称として `List` という名前を付けている。この部分を、例えば次のように略記することは出来ない。

```
typedef struct {
```

```
    String id;
    int    num;
    List   next;  /* この時点では List という名前は未定義 */
} List_item, *List;
```

- プログラム 37~40 行目 は入力ストリームに間違いがあるかどうかをチェックしている。[34 行目の scanf はファイル終了を検知すると EOF を値として返すので、34~35 行目のループを抜けた時点で count≠EOF なら入力ストリーム中に不正な文字があったことになる。 最後の行に 2 番目のデータが欠けていた場合も、その最後の部分を読み込む際に 34 行目の scanf の値は 1 になるので入力エラーとして検出される。]
- プログラム 46~85 行目 で定義された関数 add_item が呼び出された直後は、例えば次の様になっている。



- プログラム 64 行目 の strcmp は 2 つの引数で指定された文字列を辞書式順序に従って比較するライブラリ関数であり、そのプロトタイプは<string.h>に入っている。第 1 引数の文字列が第 2 引数の方より辞書順で前に来るか, 同じ場所に来るか, 後ろに来るかに応じて、それぞれ 負, ゼロ, 正 の値を返す。
- プログラム 67 行目,74 行目 の malloc は引数で指定されたバイト数の記憶領域をヒープ領域から切り出して来るライブラリ関数であり、そのプロトタイプは<stdlib.h>に入っている。メモリ確保に成功するとそこへのポインタを返し、失敗すると NULL を返す。

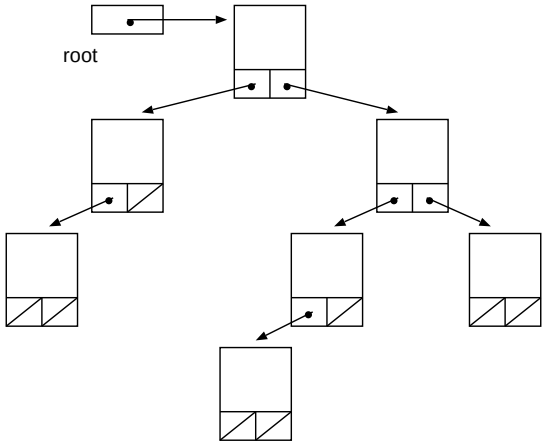
配列 vs. 線形リスト：

- どちらも、データを一列に並べたものを表せる。
- 各々の特徴は次の通り。

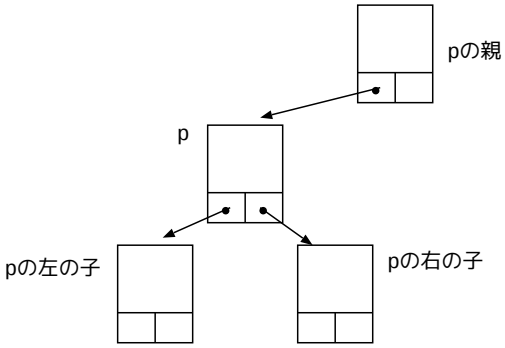
配列	···	要素の挿入・削除がない場合は、アクセス時間、記憶容量の点で最適な方法。
線形リスト	···	要素の挿入・削除が簡単に行なえる。しかし、アクセスに時間がかかる。

12.4 2分木データ構造

この節では2分木構造、すなわち次のような形のデータ構造の扱い方を例示する。



一般に、このような構造の構成要素である構造体  を**節点** (node) と言い、節点と節点を結ぶ線を**枝** (branch) と言う。2分木では、どの節点 p についても、それと1本の節点で結ばれた節点は p の上方, 左下, 右下にそれぞれ高々1個しか存在しない。p の上方, 左下, 右下にある節点をそれぞれ p の**親** (parent), **左の子** (left child), **右の子** (right child) と言い、p の左右の子をまとめて p の**子** (child) と言う。



そして、親を持たない節点を**根** (root)、子を持たない節点を**葉** (leaf) と言う。また、根と節点を結ぶのに必要な枝の本数をその節点の**レベル** (level) といい、レベルの最大値を2分木の**高さ** (height) という。

例題 12.4 (2分木の中間順走査) 例題12.3では、①入力ストリームからデータを読み込んでは ②そのデータ内の 識別子 項目に関して辞書順になる様に線形リストに挿入してゆき、最後に線形リストに保存されたものを順に出力するプログラムを示した。配列ではなく線形リストを用いれば扱えるデータ数に上限がなくなるが、線形リストだと読み込んだデータを挿入する場所を探すのに相当の手間がかかってしまう。 そこで、常に

(左の子とその子孫の識別子) $\leq$ (親の識別子)

(親の識別子) $<$ (右の子とその子孫の識別子)

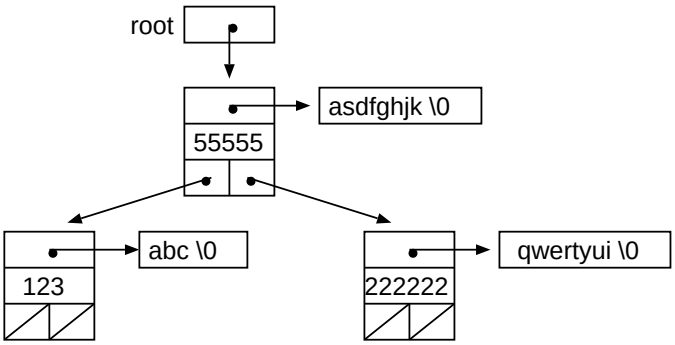
を満たすようにデータを記録すれば、データを辞書順に2分木上に保存できることに着目する。線形リストではなくこの様な2分木状に識別子と整数の組を保持することにして、例題12.3のプログラムを作り変えてみよ。

(考え方) 常に

(左の子とその子孫の識別子) ≤ (親の識別子)

(親の識別子) < (右の子とその子孫の識別子)

を満たすようにデータを記録する訳だから、出来上がった 2 分木は例えば次の様なものになる。



また、求められているプログラムは、

① 入力ストリームに現れる 識別子 (1 個以上の空白) 整数 という形のデータを読み込んでは、

② それを、常に

(左の子とその子孫の識別子) ≤ (親の識別子)

(親の識別子) < (右の子とその子孫の識別子)

を満たすように 2 分木に登録する、

という作業を繰り返し、最後に 2 分木に登録されたものを識別子の辞書順に出力することになる。 このプログラムを作成するに当たって考慮すべき、最も重要なことは次の 2 点である。

- 2 分木内に蓄えられたデータをどの様な手順で辞書順に取り出すか。
- 入力した識別子と整数の組を (辞書順を保つ様に) 2 分木のどこに、またどの様な手順で挿入するか。

以下、これら 2 点について考える。

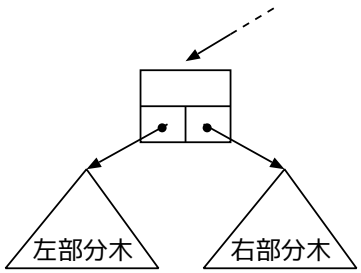
2 分木内に蓄えられたデータを辞書順に全て取り出したい時、2 分木の節点を再帰的にたどり、各々の節点で

① 左部分木内に蓄えられたデータを辞書順に全て取り出す (再帰)、

② 立ち寄った節点に蓄えられたデータを取り出す、

③ 右部分木内に蓄えられたデータを辞書順に全て取り出す (再帰)、

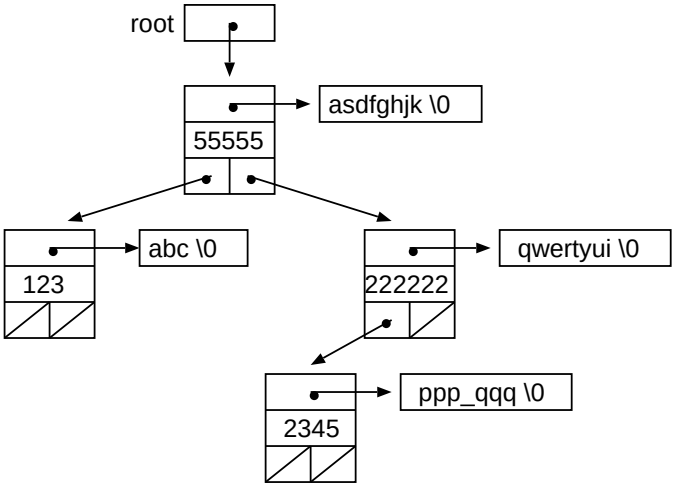
ということを行えば良い。



2 分木に新しいデータを追加したい時、2 分木の葉の先の何処か然るべき場所を見つけてそこに新データを挿入することにより、辞書順を維持できる。例えば、上の図のような 2 分木が出来ていた時、新しいデータとして

```
ppp_qqq  2345
```

というものがあれば、これを次の様に追加すれば辞書順を保ったままに出来る。



そして、この追加作業は 2 分木の再帰的な構造に沿って行えば良い。実際、ある与えられた節点 v 以下の 2 分木に新データを挿入したい場合は、新データを v の左部分木に挿入すべきか、右部分木に挿入すべきかの判断をして、

もし左部分木に挿入すべきと判断されたら

①新データを v の左部分木に挿入 (再帰) して

②出来上がった新しい左部分木へのポインタを v の構造体の中に格納する。

そして、もし右部分木に挿入すべきと判断されたら

①新データを v の右部分木に挿入 (再帰) して

②出来上がった新しい右部分木へのポインタを v の構造体の中に格納する、

ということを行えば良い。

注目：
新しいデータを登録する際、例 12.3 では挿入場所を探す作業を `while` 文による繰り返しで行ったが、ここでは 2 分木の構造に沿った再帰で行っている。

自己参照的構造体をつなげて出来るデータ構造は再帰的な構造をしているため、一般にこれらのデータ構造の処理は繰り返しでうまく書き表せるとは限らない。この例のようにデータ構造の持っている再帰的な構造に沿ってプログラムを再帰的に構成する方が無難である。

(プログラミング) プログラムの見通しを良くするために、入力した識別子と整数の組を (辞書順を保つ様に)2 分木に挿入する関数 `add_item()` と、2 分木内に蓄えられたデータを辞書順に出力し同時にメモリ解放も行う関数 `inorder_traverse_and_print_free()` を用意してプログラムを構成した。このプログラムと、これをコンパイル/実行している様子を次に示す。

```
[motoki@x205a]$ nl dynamic-btree.c
     1 /*************************************************************/
     2 /*                                                           */
     3 /*  2 分木上に識別子 (と整数) を辞書順に登録し、最後に 2 分木を    */
     4 /* 中間順に走査することによって蓄えられたデータを辞書順に出力 */
     5 /*                                                           */
     6 /*************************************************************/

     7 #include <stdio.h>
     8 #include <string.h>
```

```
 9 #include <stdlib.h>

10 #define  MAXLENGTH  100     /* 識別子の最大の長さ */

11 #define Is_empty(btree)  ((btree) == NULL)

12 typedef char *String;

13 typedef  struct btree_item  *Btree;
14 typedef struct btree_item {
15     String id;
16     int    num;
17     Btree  left_child, right_child;
18 } Btree_item;

19 Btree add_item(Btree tree, String id, int num);
20 void inorder_traverse_and_print_free(Btree);

21 int main(void)
22 {
23   Btree root;
24   char  identifier[MAXLENGTH+1];
25   int   count, num;

26   root = NULL;
27   while ((count=scanf("%100s %d", identifier, &num)) == 2)
28     root = add_item(root, identifier, num);

29   if (count != EOF) {
30     printf("Input Error!\n");
31     exit(EXIT_FAILURE);
32   }

33   printf("番号        整数   識別子\n");
34   inorder_traverse_and_print_free(root);
35   return 0;
36 }

37 /*-------------------------------------------------------------*/
38 /* 2分木上に識別子と整数の組を追加                             */
39 /*-------------------------------------------------------------*/
40 /* (仮引数) tree : データの登録先となる2分木へのポインタ       */
41 /*          id   : 文字列データへのポインタ                    */
```

```
42 /*          num   : 整数                                          */
43 /* (関数値) : データ登録後の２分木へのポインタ                    */
44 /* (機能)   : 識別子と整数の組を１つの項目として、それらが        */
45 /*                左の子とその子孫の識別子 <= 親の識別子          */
46 /*                親の識別子       <   右の子とその子孫の識別子   */
47 /*            を常に満たすように２分木状に保存されており、        */
48 /*            tree がその２分木の根を指し示すと仮定する。         */
49 /*            この仮定の下で、id の指し示す識別子と整数 num の    */
50 /*            組を新しい項目として（辞書順を保つ様に）２分木に    */
51 /*            挿入して、出来た２分木へのポインタを返す。          */
52 /*----------------------------------------------------------------*/
53 Btree add_item(Btree tree, String id, int num)
54 {
55   Btree_item *new_item;

56   if (Is_empty(tree)) { /*新しい item を作りそこへのポインタを返す*/
57     new_item = (Btree_item *) malloc(sizeof(Btree_item));
58     if (new_item == NULL) {
59       printf("*** fail in memory allocation ***");
60       exit(EXIT_FAILURE);
61     }
62     new_item->id = (String) malloc(strlen(id)+1); /*id を入れる領*/
63     if (new_item->id == NULL) {                   /*域を確保し、*/
64       printf("*** fail in memory allocation ***");/*そこへのポイ*/
65       exit(EXIT_FAILURE);                         /*ンタを代入  */
66     }
67     strcpy(new_item->id, id);               /* 新領域に id を複写 */
68     new_item->num         = num;
69     new_item->left_child  = NULL;
70     new_item->right_child = NULL;
71     return new_item;
72   }else if (strcmp(id, tree->id) <= 0) { /* 左部分木へ挿入 */
73     tree->left_child = add_item(tree->left_child, id, num);
74     return tree;
75   }else {                                 /* 右部分木へ挿入 */
76     tree->right_child = add_item(tree->right_child, id, num);
77     return tree;
78   }
79 }

80 /*----------------------------------------------------------------*/
81 /* ２分木の中味を辞書順に出力，同時に２分木のメモリを解放        */
82 /*----------------------------------------------------------------*/
```

```
 83 /*（仮引数）ptr : 2分木へのポインタ                                    */
 84 /*（関数値）: なし                                                    */
 85 /*（機能）   : 識別子と整数の組を1つの項目として、それらが            */
 86 /*                左の子とその子孫の識別子 <= 親の識別子              */
 87 /*                親の識別子       <   右の子とその子孫の識別子       */
 88 /*            を常に満たすように2分木状に保存されており、ptr          */
 89 /*            がその2分木の根を指し示すと仮定する。この仮定の        */
 90 /*            下で、2分木を中間順に走査することによって、2分        */
 91 /*            木に蓄えられたデータを                                  */
 92 /*                  番号  整数  識別子                                */
 93 /*            という書式で出力する。                                  */
 94 /*            また、同時に2分木のメモリを解放する。                  */
 95 /*--------------------------------------------------------------------*/
 96 void inorder_traverse_and_print_free(Btree ptr)
 97 {
 98   static int n=1;

 99   if (ptr != NULL) {
100     inorder_traverse_and_print_free(ptr->left_child);
101     printf("%4d%11d  %s\n", n++, ptr->num, ptr->id);
102     inorder_traverse_and_print_free(ptr->right_child);
103     free(ptr->id);
104     free(ptr);
105   }
106 }
[motoki@x205a]$ cat dynamic-llist.data
asdfghjk 55555
qwertyui 222222
abc      123
ppp_qqq  2345
zxcv     987
kkk      456
efghi    77
[motoki@x205a]$ gcc dynamic-btree.c
[motoki@x205a]$ ./a.out <dynamic-llist.data
番号        整数  識別子
   1         123  abc
   2       55555  asdfghjk
   3          77  efghi
   4         456  kkk
   5        2345  ppp_qqq
   6      222222  qwertyui
   7         987  zxcv
```

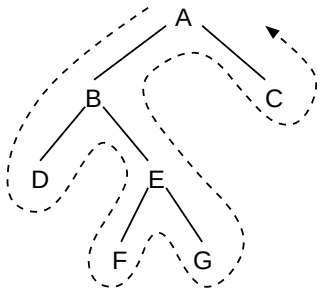
```
[motoki@x205a]$
```

ここで、

- プログラム 57 行目,62 行目 の malloc は引数で指定されたバイト数の記憶領域をヒープ領域から切り出して来るライブラリ関数であり、そのプロトタイプは<stdlib.h>に入っている。メモリ確保に成功するとそこへのポインタを返し、失敗すると NULL を返す。
- プログラム 72 行目 の strcmp は 2 つの引数で指定された文字列を辞書式順序に従って比較するライブラリ関数であり、そのプロトタイプは<string.h>に入っている。第 1 引数の文字列が第 2 引数の方より辞書順で前に来るか, 同じ場所に来るか, 後ろに来るかに応じて、それぞれ 負, ゼロ, 正 の値を返す。
- プログラム 98 行目 で確保された変数 n は static 宣言されているので、関数 inorder_traverse_and_print_free に固有の変数として見なされ、呼び出された全ての inorder_traverse_and_print_free 関数によって共有される。
- プログラム 103~104 行目 の free は引数で指定した記憶領域を解放してヒープ領域に戻すライブラリ関数であり、そのプロトタイプは<stdlib.h>に入っている。

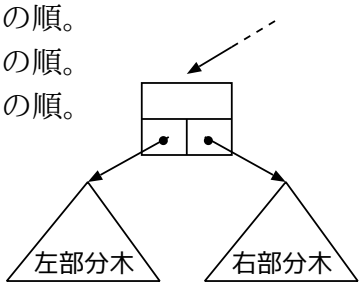
2 分木の全ての節点をたどる方法：

- 親から子へのポインタ (だけ) で節点を繋げて 2 分木を表す場合は、「深さ優先」でたどるしかない。



- 立ち寄った節点で何らかの処理を行いたい場合、木の構造に基づいて再帰的にアルゴリズムを書くことが出来る。左部分木の方を右部分木より先にたどることにすると、各節点の処理の順番としては次の 3 種類が可能。

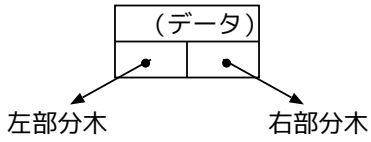
  先行順 ··· 立ち寄った節点, 左部分木, 右部分木の順。
  中間順 ··· 左部分木, 立ち寄った節点, 右部分木の順。
  後行順 ··· 左部分木, 右部分木, 立ち寄った節点の順。



演習問題

□**演習 12.1 (2 分木の大きさ, 高さ,...)** 整数データを節点に持つ 2 分木は、

```
typedef  struct node  *Tree;
typedef  struct node  {
   int  data;
   Tree left_subtree;
   Tree right_subtree;
} Node;
```



という風に定義された構造体を木の節点とし、これらをポインタで繋げることによって表すことができる。 この様に表された 2 分木の根へのポインタが引数として与えられた時、

(1) その 2 分木の**大きさ** (i.e. 節点の個数) を調べて返す C の関数

```
int size(Tree t)
```

を定義せよ。

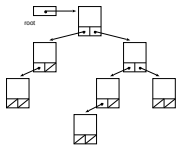
Hint. 2 分木の大きさに関する
2 分木の大きさ
= (2 分木の左部分木の大きさ) + (2 分木の右部分木の大きさ) + 1
という漸化式に着目すれば簡単。

(2) その 2 分木の**高さ** (すなわち、根節点から葉節点までの枝の本数の最大値) を調べて返す C の関数

```
int height(Tree t);
```

を定義せよ。

補足：次の 2 分木の高さは 3 である。

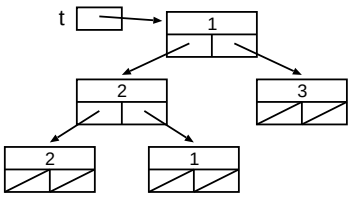


(3) その 2 分木内の節点に保持された整数データの総和を計算して返す C の関数

```
int sum_data(Tree t);
```

を定義せよ。

計算例：次の 2 分木の場合、`sum_data(t)`=1+2+2+1+3=9 である。

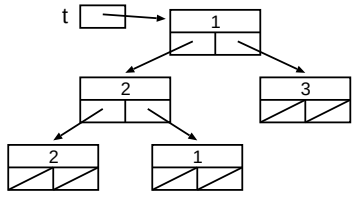


(4) その 2 分木内の**節点に保持された整数データの 2 乗の総和**を計算して返す C の関数

```
int sum2_data(Tree t);
```

を定義せよ。

計算例：次の 2 分木の場合、`sum2_data(t)`= $1^2+(2^2+2^2+1^2)+(3^2) = 19$ である。



□**演習 12.2 (2 分木の高さ)** 2 分木をランダムに 100 個生成 (高さ制限 6) し、それらの高さの分布を調べる C プログラムを作成せよ。

□**演習 12.3 (2 分木の表示)** 2 分木をランダムに 10 個生成 (高さ制限 6) し、節点レベルを揃えてそれらを表示 (出力) する C プログラムを作成せよ。

13 2 分木, push-down スタック, 待ち行列

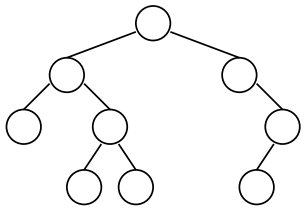
- 2 分木,
- push-down スタック,
- 自習 待ち行列

動的データ構造を用いると自由に色々なデータ構造を構成できるが、データへのアクセスに手間がかかることも多く、またヒープ領域のメモリを動的に使うことになるので、デバッグ等も難しくなる。 [例えば、使わなくなった領域を解放し忘れると知らず知らずの内に使えるメモリが少くなってゆく。(**メモリ洩れ**という。) また逆に、まだ使っている領域を間違って解放してしまうと、同じ領域を 2 重に使うために訳の分からない結果になる。]

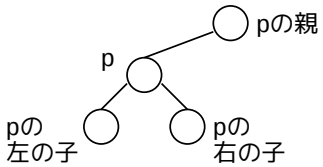
そこで、この節では静的に (i.e. 宣言によって) 確保された領域であっても色々な用途に使えることを例示する。データ量の上限が予め分かっているなら、静的な領域を用いて 2 分木構造等を動的に表すことが出来る。

13.1 2 分木

2 分木： 一般に、(データ構造に限らず) 次のような形の階層構造を **2 分木**と呼ぶ。



この様な構造の中の○をやはり**節点**と言い、節点と節点を結ぶ線を**枝**と言う。親, 子, 根, ... といった用語の定義も 2 分木データ構造の場合と同じである。すなわち、どの節点 p についても、p の上方, 左下, 右下にある節点をそれぞれ p の**親**, **左の子**, **右の子**と言い、p の左右の子をまとめて p の**子**と言う。



そして、親を持たない節点を**根**、子を持たない節点を**葉**と言う。また、根と節点を結ぶのに必要な枝の本数をその節点の**レベル**といい、レベルの最大値を 2 分木の**高さ**という。

2 分木の表現： 2 分木をプログラムの中で表す方法としては、次の 2 つがよく使われる。

- **ポインタを用いる方法**
  既に 12.2 節, 12.4 節で見てきた方法である。すなわち、節点のデータを入れる小さな記憶領域を節点毎に動的に確保し、それらをポインタで繋ぐ。(2 分木の各枝をポインタで表す。)
- **配列で表す方法**
  この方法は、2 分木の大きさの上限が決まっている場合にのみ可能である。 この方法

では、次の様に節点に番号付けすれば、2 分木の各節点と非負整数を 1 対 1 に対応づけれることに着目する。

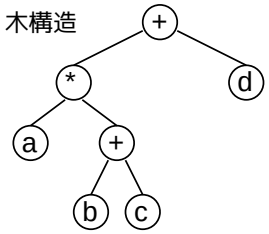
$$\begin{cases} \text{根の番号} = 0, \\ \text{節点 } p \text{ の番号が } i \text{ なら} \\ \qquad p \text{ の左の子の番号} \;=\; 2i+1, \\ \qquad p \text{ の右の子の番号} \;=\; 2i+2, \\ \qquad p \text{ の親の番号} \qquad =\; \lfloor (i-1)/2 \rfloor \end{cases}$$

$\Longrightarrow$ 節点の番号と配列の添字を対応させて

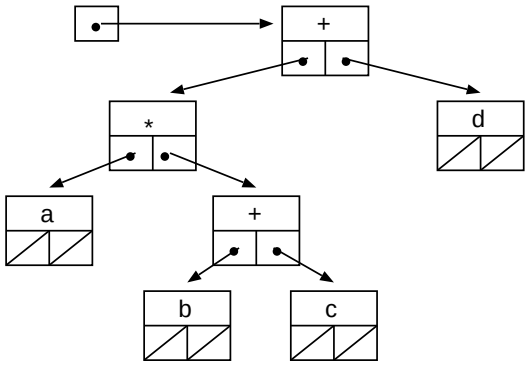
a[i] = 番号 i の節点のデータ

とすればよい。

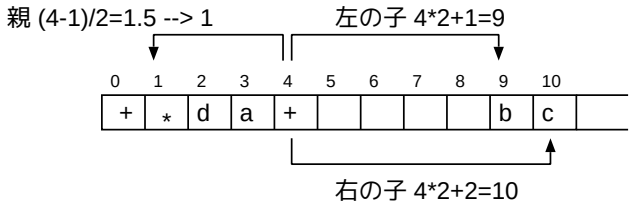
例 13.1 (2 分木の表現) 算術式 a*(b+c)+d の構造は次の様な 2 分木で表せる。

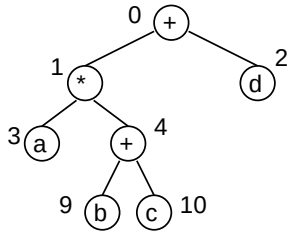


この 2 分木は、ポインタを用いて表すと次の様になる。

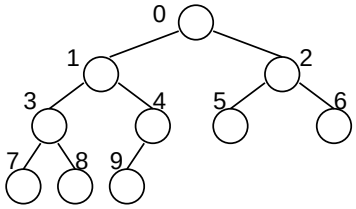


また、配列で表すと次の様になる。





完全 2 分木： 高さ h の 2 分木において、レベル $i(< h)$ の節点が 2^i 個 (満杯) 存在しレベル h の節点が左詰めに並んでいる時、この 2 分木を特に**完全 2 分木** (complete binary tree) という。例えば、10 個の節点からなる完全 2 分木は次の様な構造を持つ。

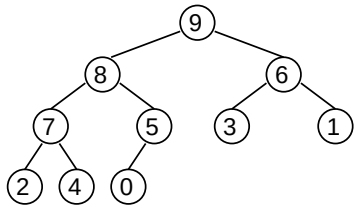


$\Longrightarrow$ 節点数 n の完全 2 分木を配列 a で表す場合、各節点の情報は a[0] $\sim$ a[$n-1$] に格納される。 [途中に穴は出来ない。]

ヒープ： 完全 2 分木において、各節点に数値データがラベル付けされていて、どの枝についても

(親の数値データ) $\geq$ (子の数値データ)

が成り立つ時、この完全 2 分木、あるいは、この完全 2 分木を表す配列を特に**ヒープ** (heap, **整列 2 分木**) という。例えば、各節点に非負整数をラベル付けした完全 2 分木



あるいは、この完全 2 分木を表す配列

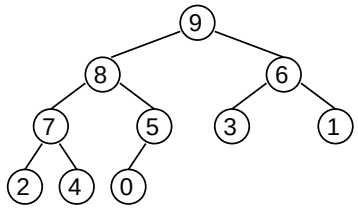
0	1	2	3	4	5	6	7	8	9
9	8	6	7	5	3	1	2	4	0

はヒープである。

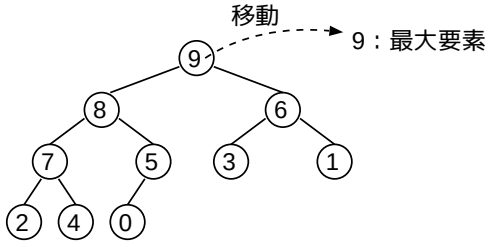
例題 13.2 (heapsort) int 型配列に入ったデータをヒープソート手法で小さい順に並べ替えるための汎用のモジュールを作成せよ。

(考え方) ヒープソート (heapsort, **整列 2 分木法**) は、最悪の場合でもそれなりに効率良く整列化を行うアルゴリズムとして有名なものである。この整列化法はヒープ (の性質) をうまく利用してデータを整列化する方法で、そのアルゴリズムは次の様に書き表すことが出来る。

① 整列すべきデータが節点のラベルとして振り分けられたヒープを構築する。

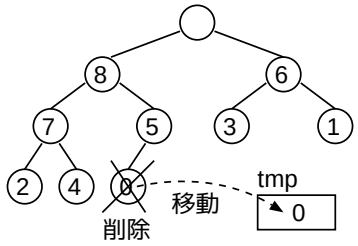


② 根にラベル付けされたデータは最大であることが分かるので、このデータを最大要素として出力用配列に移す。

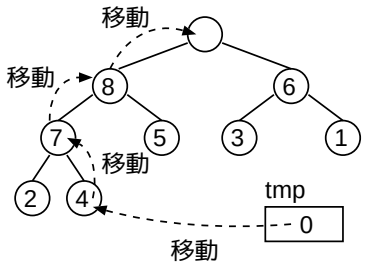


③ レベルが最大の葉節点の内、最も右側の節点をヒープから除去し、その中のデータを一時記憶領域 tmp に移す。[$\Longrightarrow$ tmp の中のデータを根節点に入れてもヒープになら

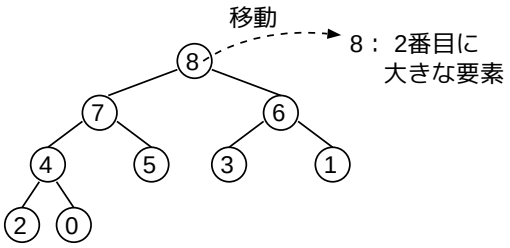
ない。]



④ ラベルの無い節点の中に大きい方の子節点データを移動する操作を繰り返し、適当なところで tmp の中のデータを「ラベルの無い節点」に移すことによって、再びヒープにする。

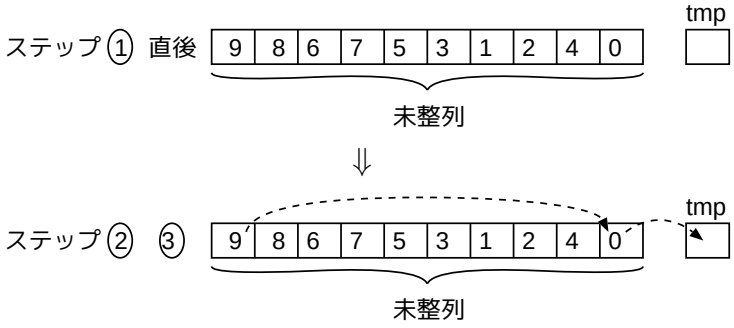


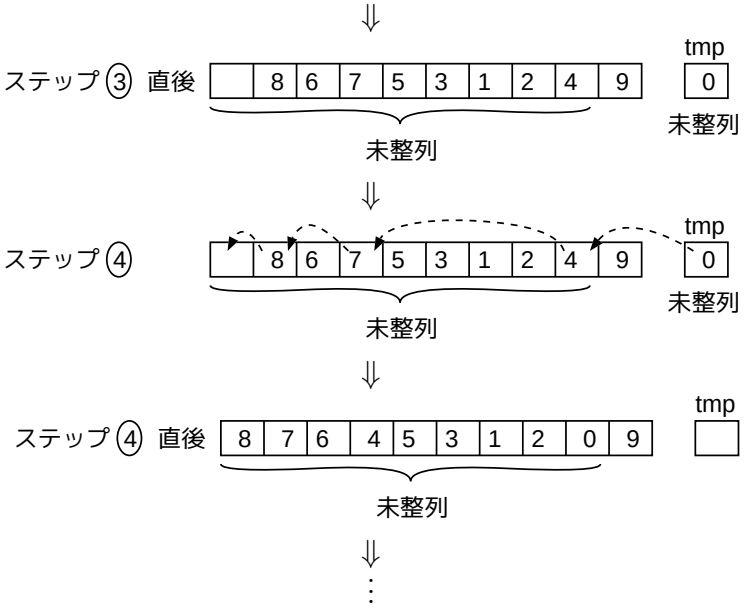
⑤ 根にラベル付けされたデータはヒープに残ったデータの中で最大であることが分かるので、このデータを出力用配列の然るべき場所に移す。



⑥ 節点数が 1 になるまで③~⑤を繰り返す。

では、上記ヒープをプログラム内でどう表現すれば良いのか？ ポインタを用いて上記ヒープを表そうとすると、上記ステップ③でレベルが最大の節点のうちで最も右側の節点 (およびこの節点の親節点) へのアクセスが必要になる。しかし、ポインタを用いてヒープを表す場合は、(可能ではあるが) 未整列データの個数からこれらの節点を割り出す簡単な方法はない。 そこで、配列を用いて上記ヒープを表すことにする。この場合、上記ステップ②で 2 分木の根に配置されたデータの整列後の配置場所は、残った未整列データの構成するヒープの中でレベルが最大の節点のうちで最も右側の節点になるので、上記ステップ②と③(および⑤と③) は入れ換えた方が処理が効率良く進む。上記の例の場合、ヒープを表す配列の様子は次の様に変わって行く。





モジュールの構成はどうすれば良いのか？ int 型配列に入ったデータを小さい順に並べ替える作業を外部からこのモジュールに依頼できる様にしなければならない。そのためには、int 型配列の名前 a[] とその大きさ size を引数として受け取り、指定された配列内のデータをヒープソート手法で小さい順に並べ替える関数

```
void sort(int a[], int size)
```

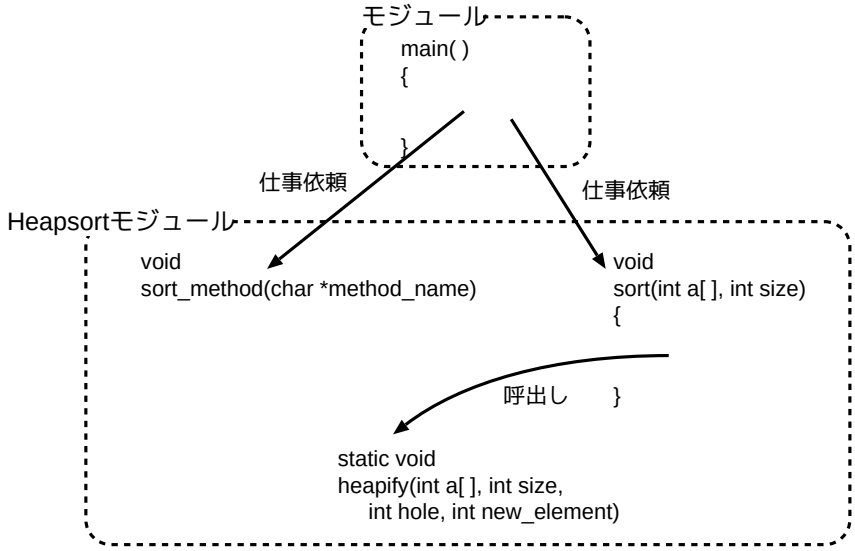
を用意すれば良い。また、「汎用のモジュール」ということなので、ソーティング手法を外部に説明する必要があるかも知れない。そのためには、例えば引数として指定された char 型配列に ”Heapsort” という文字列を入れる関数

```
void sort_method(char *method_name)
```

を用意すれば良い。 最後に、ヒープソートの手順の見通しを良くするために、上記手順のステップ④を行う関数

```
heapify()
```

を用意できれば良い。この関数は実際には 2 分木の再帰的な構造に沿って再帰的に構成することが出来る。 この関数 heapify() は外部から直接使うことはないので、もちろん static な外部関数にして外部からは見えなくすべきである。



(プログラミング) 出来たモジュールを次に示す。

```
[motoki@x205a]$ nl btree-heapsort.c
```

```
 1 /*************************************************************/
 2 /* Heapsort モジュール : 2分木の利用例                       */
 3 /*-------------------------------------                      */
 4 /*    外部へのサービスを行うために、次の2つの関数がこの     */
 5 /*    モジュールの中に用意されている。                       */
 6 /*    (1) 整列化アルゴリズムの名前を答える関数 sort_method,  */
 7 /*    (2) 配列要素を Heapsort アルゴリズムで                 */
 8 /*                  で小さい順に並べ替える関数 sort          */
 9 /*************************************************************/

10 #include <stdio.h>

11 static void heapify(int a[], int size, int hole, int new_element);

12 /*-----------------------------------------------------------*/
13 /* 整列化アルゴリズムの名前を答える                          */
14 /*-------------------------------------                      */
15 /* (引数) method_name : 出力用。この char 型配列に方式の名前  */
16 /*                      を (文字列として) 入れて返す。配列の  */
17 /*                      大きさは 9 文字分必要。               */
18 /*-----------------------------------------------------------*/
19 void sort_method(char *method_name)
20 {
21   sprintf(method_name, "Heapsort");
22 }

23 /*-----------------------------------------------------------*/
24 /* 配列要素を小さい順に並べ替える (heapsort)                 */
25 /*-------------------------------------                      */
26 /* (仮引数) a    : int 型配列                                */
27 /*          size : int 型配列 a の大きさ                     */
28 /* (関数値)  :  なし                                         */
29 /* (機能) :  heapsort アルゴリズムを使って、配列要素         */
30 /*               a[0],a[1],a[2], ..., a[size-1]              */
31 /*           を値の小さい順に並べ替える。                    */
32 /*-----------------------------------------------------------*/
33 void sort(int a[], int size)
34 {
35   int  k, tmp;

36   /* 下から heap を構築してゆく */
```

```
37   for (k=size/2-1; k>=0; --k)       /* 葉以外の個数 = size/2 */
38     heapify(a, size, k, a[k]);

39   /* 大きい順に heap から取り出してゆく */
40   for (k=size-1; k>=1; --k) {
41     tmp  = a[k];
42     a[k] = a[0];
43     heapify(a, k, 0, tmp);
44   }
45 }

46 /*-------------------------------------------------------------*/
47 /* 番号 hole の節点より下の部分が heap の条件を満たす時、       */
48 /* 新要素を加えて hole 以下の部分が heap の条件を満たす様にする。*/
49 /*-------------------------------------------                  */
50 /* (仮引数) a     : int 型配列                                  */
51 /*      tree_size : 2分木と見做す部分配列 a[0],a[1],... の大きさ*/
52 /*           hole : a[0]〜a[tree_size] の表す2分木内の節点番号*/
53 /*   new_element : 2分木の節点に振り分けていない値             */
54 /* (関数値)  :  なし                                           */
55 /* (機能) :  番号 hole の節点のデータ記憶域は空で、その分    */
56 /*           new_element という値がどの節点にも記録されてい */
57 /*           ない、また、hole より下の部分が heap の条件を満  */
58 /*           たしている、という状況を想定する。この様な状況 */
59 /*           の時に、hole より下にあるデータを上に shift up */
60 /*           する操作を繰り返し行い、適当な時点で空の節点に */
61 /*           新しい要素 new_element を割り当てることにより、*/
62 /*           hole 以下の部分が全面的に heap の条件を満たす様にする。*/
63 /*-------------------------------------------------------------*/
64 static void heapify(int a[], int tree_size, int hole, int new_element)
65 {
66   int  siftup_cand;      /* siftup candidate */

67   while ((siftup_cand = hole*2+1) < tree_size) {
68     if (siftup_cand+1<tree_size                /*右の子も居て*/
69         && a[siftup_cand]<a[siftup_cand+1])    /*右の子の方が*/
70       ++siftup_cand;                           /*大きい場合は*/
71                                          /*右の子が siftup の候補*/
72     if ( new_element >= a[siftup_cand])
73       break;       /* new_element を hole の場所に入れれば良い */

74     a[hole] = a[siftup_cand];     /* sift up */
75     hole    = siftup_cand;
```

```
76   }
77   a[hole] = new_element;
78 }
```

ここで、

- プログラムの 21 行目 に現われる sprintf 関数は、printf と同様の出力を引数のポインタ method_name で指定された char 型配列上に行うものである。ただ、printf と違って最後に空文字 \0 も書き出す。

(実行) この heapsort モジュールの動作確認は例題 4.5(quicksort) と同じように行うことが出来る。次の通り。

```
[motoki@x205a]$ nl check-sort-program.c
```

```
 1  /*************************************************************/
 2  /* Sort プログラムの動作確認                                 */
 3  /*-------------------------------------                      */
 4  /*    大きさ 100 の配列にランダムに整数を生成し、            */
 5  /*    その配列要素を別途用意された整列化プログラムを使って  */
 6  /*    昇順に並べ替えて出力する。                             */
 7  /*************************************************************/

 8  #include <stdio.h>
 9  #include <stdlib.h>   /* 乱数発生のライブラリ関数を使うため */

10  #define  SIZE   100
11  #define  WIDTH  10

12  void set_an_array_random(int a[], int size);
13  void pretty_print(int a[], int size);
14  void sort(int a[], int size);
15  void sort_method(char *method_name);

16  int main(void)
17  {
18    int   a[SIZE], seed;
19    char  sort_name[40];

20    printf("Input a random seed (0 - %d):  ", RAND_MAX);
21    scanf("%d", &seed);
22    srand(seed);

23    set_an_array_random(a, SIZE);
24    printf("\nbefore sorting:\n");
25    pretty_print(a, SIZE);
```

```
26    sort(a, SIZE);
27    sort_method(sort_name);        /* Sort モジュールに          */
28                                   /* 何のアルゴリズムか尋ねる */
29    printf("\nafter sorting (%s):\n", sort_name);
30    pretty_print(a, SIZE);
31    return 0;
32  }

33  /*-------------------------------------------------------------*/
34  /* 引数で与えられた配列の各要素をランダムに設定する          */
35  /*-----------------------------------------                    */
36  /* (仮引数) a     : int 型配列                                 */
37  /*           size : int 型配列 a の大きさ                      */
38  /* (関数値)  :  なし                                           */
39  /* (機能) :  配列要素 a[0]～a[size-1] に 0～999 の間の乱数  */
40  /*            を設定する。                                     */
41  /*-------------------------------------------------------------*/
42  void set_an_array_random(int a[], int size)
43  {
44    int i;

45    for (i=0; i<size; ++i)
46      a[i] = rand() % 1000;
47  }

48  /*-------------------------------------------------------------*/
49  /* 引数で与えられた配列の要素を順番に全て出力する            */
50  /*-----------------------------------------                    */
51  /* (仮引数) a     : int 型配列                                 */
52  /*           size : int 型配列 a の大きさ                      */
53  /* (関数値)  :  なし                                           */
54  /* (機能) :  配列要素 a[0]～a[size-1] の値を順番に全て出力  */
55  /*            する。但し、各々の値は横幅 7 カラムのフィールド  */
56  /*            に出力することにし、また、1 行に WIDTH 個の要素   */
57  /*            を出力する。                                     */
58  /*-------------------------------------------------------------*/
59  void pretty_print(int a[], int size)
60  {
61    int i, count=1;

62    for (i=0; i<size; ++i, ++count) {
63      printf("%7d", a[i]);
64      if (count >= WIDTH) {
```

```
    65        printf("\n");
    66        count = 0;
    67      }
    68    }
    69    if (count > 1)
    70      printf("\n");
    71  }
[motoki@x205a]$ gcc check-sort-program.c btree-heapsort.c
[motoki@x205a]$ ./a.out
Input a random seed (0 - 2147483647):  333

before sorting:
    556    289    435    368    666    319    214    273    132    585
     64    943    869    956     50    298    112    218      5    649
    603    936    515    385    671    776    137    886      4    563
    718    913    204    153    281    870    473    495    144    605
    432    208    548    653    517    950    951    629    520    957
    630    476    893    498    861    917    626    998    803    631
    913    521    544    470     27    825    340    500    672    836
    105    104    397      6    110    914    308     61    895    829
     18    878    305    264    376    518    181    354    517    336
    985    782    857    881    252    236    706    945    736    730

after sorting (Heapsort):
      4      5      6     18     27     50     61     64    104    105
    110    112    132    137    144    153    181    204    208    214
    218    236    252    264    273    281    289    298    305    308
    319    336    340    354    368    376    385    397    432    435
    470    473    476    495    498    500    515    517    517    518
    520    521    544    548    556    563    585    603    605    626
    629    630    631    649    653    666    671    672    706    718
    730    736    776    782    803    825    829    836    857    861
    869    870    878    881    886    893    895    913    913    914
    917    936    943    945    950    951    956    957    985    998
[motoki@x205a]$
```

13.2 push-down スタック

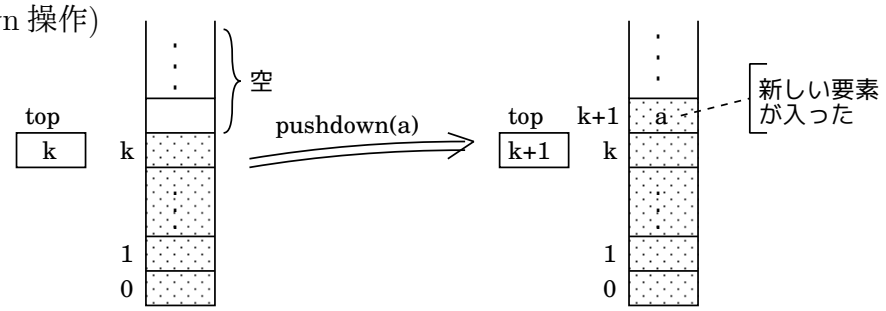
push-down スタック： 「pushdown」と「popup」の操作を備え、データの出し入れが後入れ先出し (last-in-first-out,LIFO) になるデータ記憶領域を一般に **push-down スタック** (push-down stack) または単にスタックという。

- スタックに入れる要素の個数がいくらでも大きくなり得る場合は、線形リストを用い

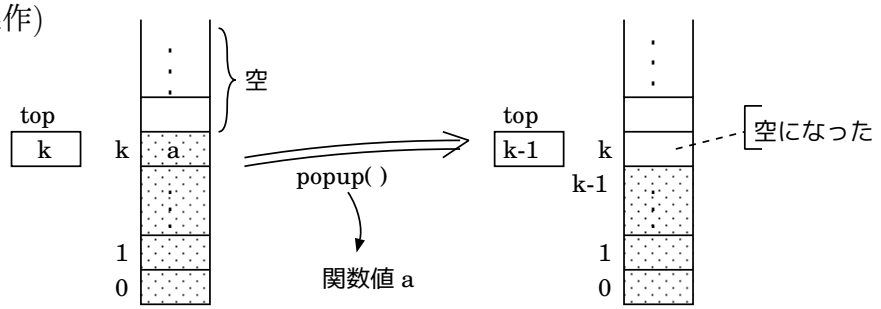
てスタックを実装するしかない。

- スタックの場合は蓄えられたデータの途中に要素を挿入することも途中から要素を抜き出すこともないので、スタックに入れる要素の個数の上限が分かっているなら、配列を用いてスタックを実装するのが良い。(その方が、メモリの量も少くて済むし処理速度も速い。)

(pushdown 操作)



(popup 操作)



例題 13.3 (スタックを使って文字列を反転する) int 型データが最大 100 個入るスタックを表す汎用のモジュールを作成せよ。そして、このスタックモジュールを利用して、①文字列を 1 個読み込み ②それを反転した後 ③出力する プログラムを作成せよ。

補足：
char 型データを int のスタックに入れることになるが、char 型は整数型の一種であるので問題は起きない。char 型データ用のスタックを用いてもよいが、int 型用スタックにしておくと別の用途にも使えるので、ここでは int 型データ用のスタックモジュールを作成することにした。

(考え方) スタックモジュールを実装するに当たって最初に考えなければならないことは、スタック領域をどこに確保するかということである。pushdown 操作と popup 操作はこのスタックモジュールが提供するのであろうが、スタック領域をスタックモジュールを呼び出す側で確保して pushdown や popup の操作を行う関数に引数としてスタック領域のアドレスを引き渡すというのでは、元々スタックモジュールを呼び出す側で任意の時点でスタック領域に直接手を加えられることになり、「後入れ先出し」の原則が保証されなくなる。 従って、スタック領域 (とスタックの top 要素を示す変数) はスタックモジュールの中に確保し、(static 宣言して) これらの領域へは外部から直接アクセスできなくするべきである。 pushdown や popup の操作を行う関数は、これらの領域を外部から間接的に操作するためのインターフェースとして働くことになる。結局、スタックモジュールの構成要素としては、少なくとも

- スタック領域 `static int stack[100]`,
- スタックの top 要素を示す変数 `static int top`,
- pushdown 操作を行う関数 `void pushdown(int)`,
- popup 操作を行う関数 `int popup( )`

の 4 つは必要である。その他にも、スタックを正しく使うために

- スタックの初期化を行う関数 `initialize_stack(void)`,
- スタックが空かどうかを判定する関数 `int is_empty(void)`

という関数を用意すれば良い。

一方、スタックモジュールを利用する側に関しては、①文字列を 1 個読み込み ②それを反転した後 ③出力するという作業の中で、スタックが有用となるのは文字列反転の部分である。文字列を反転するには文字列の先頭から 1 文字ずつスタックに pushdown していき、それが終われば popup を繰り返すだけである。popup は元々の文字列の最後尾から逆順に行われることになるので、これを入力文字列の入っていた配列に先頭から順に格納していけば良い。

(プログラミング) 指示されたプログラムと、これらをコンパイル/実行している様子を次に示す。

```
[motoki@x205a]$ nl stack-int100.c
```

```
 1  /*************************************************************/
 2  /* int 型データが最大 100 個入るスタックを実装したモジュール    */
 3  /*-------------------------------------------                  */
 4  /*     外部へのサービスを行うために、次の 4 つの関数がこの     */
 5  /*     モジュールの中に用意されている。                        */
 6  /*     (1) スタックを空に初期化する関数 initialize_stack,      */
 7  /*     (2) スタックが空かどうかを調べる関数 is_empty,          */
 8  /*     (3) スタックに要素を 1 つ push-down する関数 pushdown,  */
 9  /*     (4) スタックから要素を 1 つ pop-up する関数 popup       */
10  /*************************************************************/

11  #include <stdlib.h>
12  #define  TRUE   1
13  #define  FALSE  0

14  typedef  int  Boolean;

15  static int  stack[100];      /* モジュール外からは見えない */
16  static int  top;             /* モジュール外からは見えない */

17  void initialize_stack(void)
18  {
19    top = -1;
20  }
```

```
21  Boolean is_empty(void)
22  {
23    if (top < 0)
24      return TRUE;
25    else
26      return FALSE;
27  }

28  void pushdown(int k)
29  {
30    if (++top >= 100) {
31      printf("stack overflow\n");
32      exit(EXIT_FAILURE);
33    }
34    stack[top] = k;
35  }

36  int popup(void)
37  {
38    if (top < 0) {
39      printf("popup from empty stack\n");
40      exit(EXIT_FAILURE);
41    }
42    return stack[top--];
43  }
```

```
[motoki@x205a]$ nl stack-reverse-word.c
```

```
 1  /*************************************************************/
 2  /* スタックを利用する例                                      */
 3  /*-------------------------------------                      */
 4  /*     文字列を 1 個読み込み、                               */
 5  /*     それをスタックモジュールを用いて反転した後出力する    */
 6  /*************************************************************/

 7  #include <stdio.h>

 8  typedef  int  Boolean;

 9  void initialize_stack(void);
10  Boolean is_empty(void);
11  void pushdown(char c);
12  char popup(void);
```

```
13  int main(void)
14  {
15    char s[100];
16    int  i;

17    printf("Input a string:  ");
18    scanf("%s", s);
19    initialize_stack();
20    for (i=0; s[i]!='\0'; ++i)
21      pushdown(s[i]);
22    for (i=0; !is_empty(); ++i)
23      s[i] = popup();
24    printf("Reversed string: %s\n", s);
25    return 0;
26  }
[motoki@x205a]$ gcc stack-reverse-word.c stack-int100.c
[motoki@x205a]$ ./a.out
Input a string:  abc123
Reversed string: 321cba
[motoki@x205a]$
```

逆ポーランド記法： 算術式は普通、(2 項) 演算子を被演算数の間に置く中置記法を用いて書き表すが、他に、**前置記法** (**ポーランド記法**ともいう)、**後置記法** (**逆ポーランド記法**ともいう) という書き方も可能である。例えば、

```
(17 * 5) + 2
```

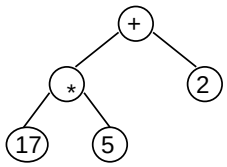
という算術式は、前置記法では

```
+  *  17  5  2
```

後置記法では

```
17  5  *  2  +
```

と書ける。 前置記法と後置記法では計算の順序を明示するのに括弧は不要であるが、被演算数同士を区切るもの (例えば空白) が必要になる。



逆ポーランド記法で書かれた算術式の評価： スタックを用いれば、逆ポーランド記法で書かれた算術式を容易に評価することが出来る。 実際、算術式を左から読みながら、次のことを行えば良い。

- 被演算数が読まれると、それをスタックに pushdown する。
- 符号反転の単項演算子 ~ が読まれると、スタックから要素を 1 つ popup し、その符号を反転したものをスタックに pushdown する。
- 2 項演算子 +, -, * または / が読まれると、スタックから要素を 2 つ popup し、
  (後で popup された値) 読まれた演算 (先に popup された値)
  という計算を行った後、その結果をスタックに pushdown する。
- 読み込むものが無くなっていれば、スタック内に算術式の評価値が 1 つだけ残ってい

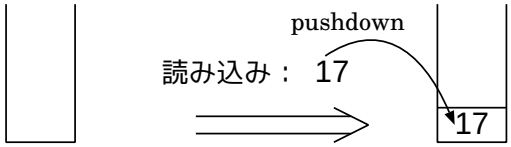
るはずである。
この方法では、例えば

17 5 * ~ 2 +

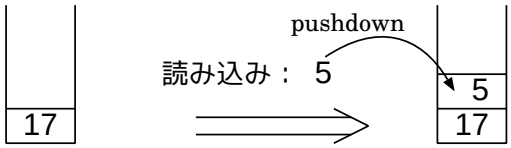
という算術式に対しては、次のように計算が進むことになる。

⓪ (初期状態)

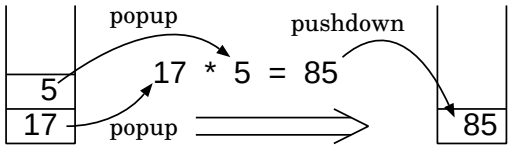
① 被演算数 17 が読まれる。



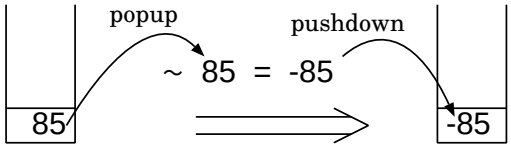
② 被演算数 5 が読まれる。



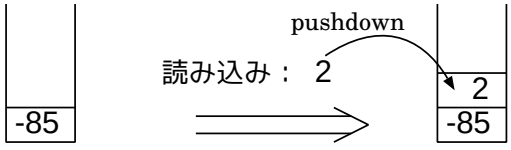
③ 演算子 * が読まれる。



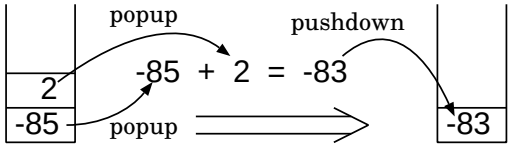
④ 演算子 ~ が読まれる。



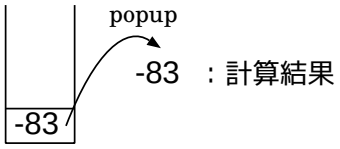
⑤ 被演算数 2 が読まれる。



⑥ 演算子 + が読まれる。



⑦ 読むものが無くなった。



例題 13.4 (逆ポーランド記法で書かれた算術式をスタックを用いて評価する) 10 進非負整数と四則演算 (2 項の加減乗除 +, -, *, / と符号反転 ~) だけから成る、逆ポーランド記法で書かれた算術式を文字列として読み込んで、その値を求めるプログラムを作成せよ。

(考え方) 例題 13.3 で作成したスタックモジュールを用いて、p.285 で説明された手順をそのままプログラムで表すだけである。

(プログラミング) ここでは簡単のため、読み込む算術式内の演算子や整数は空白で区切られ、数式の終わりは Ctrl-d で表されているものと仮定し、また、算術式の誤りのチェックは省略してプログラムを構成した。このプログラムと、これをコンパイル/実行している様子を次に示す。但し、スタックモジュール stack-int100.c は例題 13.3 で使ったものと同じであるので、ここでは再提示しない。

```
[motoki@x205a]$ nl stack-eval-expression.c
```

```
 1  /*************************************************************/
 2  /* スタックを利用する例                                      */
 3  /*-------------------------------------------                */
 4  /*    10 進非負整数と四則演算 (2 項の加減乗除 +, -, *, / と符  */
 5  /*    号反転 ~) だけから成る、逆ポーランド記法で書かれた算術 */
 6  /*    式を文字列として読み込んで、その値を求める             */
 7  /*************************************************************/

 8  #include <stdio.h>

 9  typedef  int  Boolean;

10  void    initialize_stack(void);
11  Boolean is_empty(void);
12  void    pushdown(int k);
13  int     popup(void);

14  int main(void)
15  {
16    int  a, b, num, i;
17    char buf[13];
```

```
    initialize_stack();
    while (scanf("%12s", buf) > 0) {
      switch (buf[0]) {
      case '~':
        a = popup();       /* buf[1]=='\0' のチェックは省略 */
        pushdown(-a);
        break;
      case '+':
        b = popup();       /* buf[1]=='\0' のチェックは省略 */
        a = popup();
        pushdown(a+b);
        break;
      case '-':
        b = popup();       /* buf[1]=='\0' のチェックは省略 */
        a = popup();
        pushdown(a-b);
        break;
      case '*':
        b = popup();       /* buf[1]=='\0' のチェックは省略 */
        a = popup();
        pushdown(a*b);
        break;
      case '/':
        b = popup();       /* buf[1]=='\0' のチェックは省略 */
        a = popup();
        pushdown(a/b);
        break;
      default:             /* 数字の列のはず */
        num = 0;
        for (i=0; buf[i]!='\0'; ++i)
          num = num*10 + (buf[i]-'0');  /* buf[i] が数字であること */
        pushdown(num);                  /* のチェックは省略       */
        break;
      }
    }
    printf("\nresult of evaluation: %d\n", popup());
                        /* スタックが空になっていることのチェックは省略 */
    return 0;
  }
[motoki@x205a]$ gcc stack-eval-expression.c stack-int100.c
[motoki@x205a]$ ./a.out
17 5 * ~ 2 +
 Ctrl -d
result of evaluation: -83
```

```
[motoki@x205a]$
```

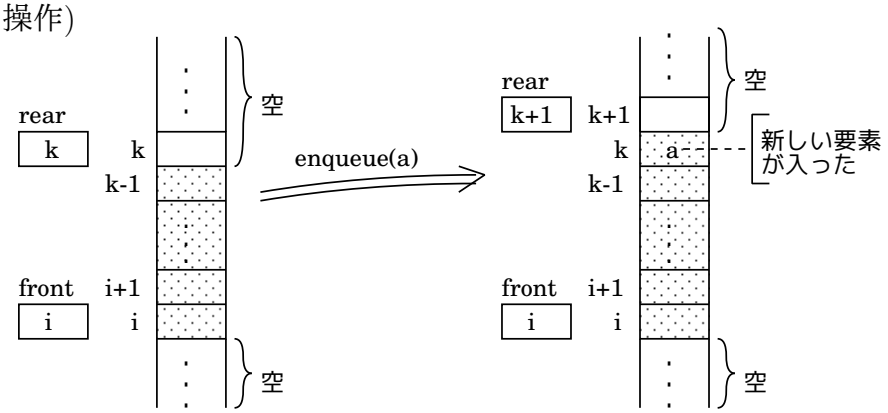
13.3 自習 待ち行列

{ ケリー&ポール 10.7 節 }

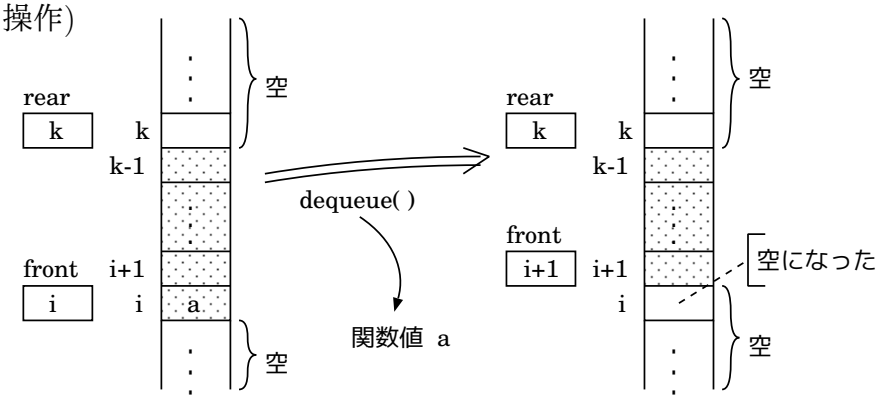
待ち行列： 「enqueue」と「dequeue」の操作を備え、データの出し入れが先入れ先出し (first-in-first-out) に従うデータ記憶領域を、一般に**待ち行列** (queue) という。

- 待ち行列に入る要素の個数に上限がないなら、線形リストを用いて待ち行列を実装するしかない。[スタックの場合と同様。]
- 待ち行列に入る要素の個数の上限が分かっているなら、配列を用いて待ち行列を実装するのが良い。[スタックの場合と同様。]

(enqueue 操作)



(dequeue 操作)



但し、この場合、何も工夫しないと時間と共に `rear` の値も `front` の値も大きくなる一方で、確保した配列の領域を超えてしまう。

$\Longrightarrow$ 大きさ N の配列 `queue` で待ち行列を表す場合は、`queue[` $N-1$ `]` の次に `queue[0]` が繋がっていると見なす。

授業では詳細は省略します。 必要になったら、「ケリー＆ポール、C の ABC(下)、星雲社」 の 10.7 節等を読んで下さい。

演習問題

□**演習 13.1 (heapsort)** ポインタを用いて 2 分木を表すことにして、ヒープソートを実装してみよ。

□**演習 13.2 (逆ポーランド記法への変換)** 中置記法で書かれた算術式 (文字列) を逆ポーランド記法に変換して出力するプログラムを作成せよ。 但し、ここで入力する算術式は簡単のため変数名 `x` と演算子 `+`, `-`, `*`, `/`, 丸括弧 `(`, `)` だけから成るものとする。また、逆ポーランド記法に変換する際、単項の `-`(符号反転) は `~` という文字に変換せよ。

14　UNIX プログラミング環境

- C コンパイラ,
- 自習 プロファイラを使う,
- C コードを計時するには,
- make コマンドによる自動分割コンパイル,
- ライブラリ,
- 自習 touch コマンド, その他の有用なツール

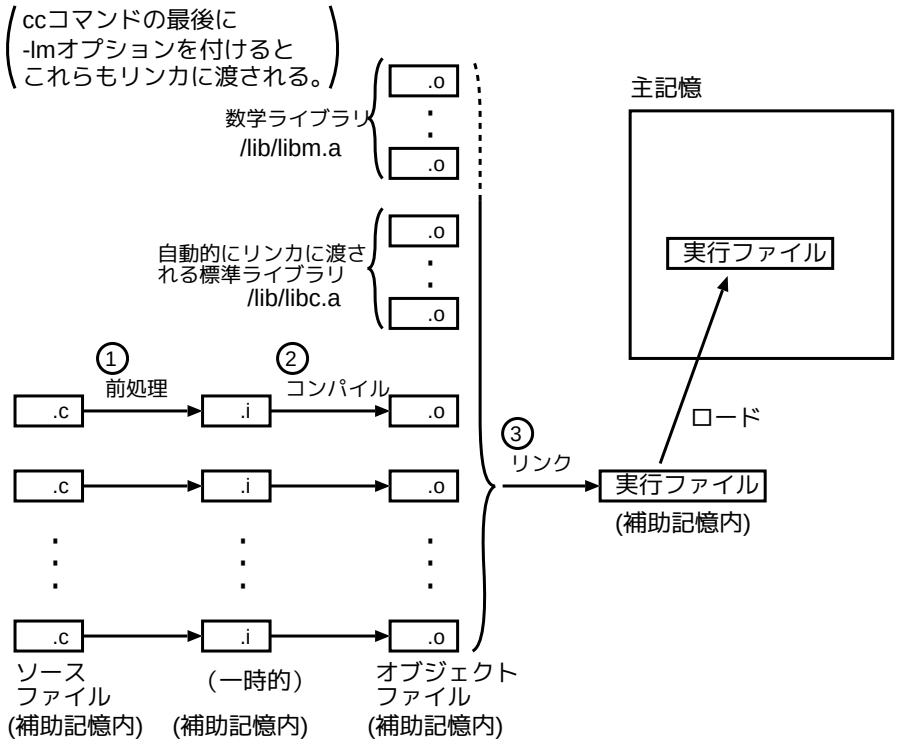
オペレーティングシステムの提供するソフトウェア開発環境：

コンパイラ
make　　･･･　分割コンパイルのサポート。
プロファイラ　･･･　プログラムのどの部分にどれ位の時間がかかっているのか、調べる。
･･･････

14.1　C コンパイラ

復習 **cc コマンドの処理：**　次の 3 つに分かれている。

① 前処理　　･･･　#include,#define 等の処理 (除去)。
② コンパイル　･･･　各々の関数定義の翻訳。
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。



分割コンパイル：　プログラムを複数のファイルに分けて書き、各々のファイル毎にコンパイルして、それらの翻訳結果をリンカにかけて実行コードを生成することも出来る。(分割コンパイルという。)

分割コンパイルの利点／特徴：

- 効率的にデバッグを行うことが出来る。

  その理由：
  文法上のチェックは独立に行うことが出来る。また、実行時のデバッグの際も、修正しなかった部分については再度デバッグする必要がないので、コンパイル時間の短縮になる。
  $\Longrightarrow$ 大きなプログラムの場合に特に有効。

- `make` コマンド (14.4 節) を使用しないと煩わしい。

cc コマンドの簡単な使用案内：

- `-o` オプションをを使えば実行ファイルの名前を任意に与えることが出来る。例えば、

  `cc -o` [実行ファイルの名前] [C のソースファイルの名前]

- `-c` オプションをを使えば前処理とコンパイルの処理だけを行うことが出来る。例えば、

  `cc -c` [*file 1*`.c`] ... [*file m*`.c`]

  これにより、文法上の誤りが無かったものについては、[*file k*`.o`] という名前のオブジェクトファイルが生成される。

- 実行ファイルを生成するのに、`.c` ファイルと `.o` ファイルの混ざったものを `cc` コマンドに与えることが出来る。例えば、

  ```
  cc main.c file1.o file2.o
  ```

  この場合は、`file1.o` と `file2.o` に対しては前処理とコンパイルの処理は行われない。(行おうとしても行えない。)
  $\Longrightarrow$ リンカだけを使いたければ `.o` ファイル群を `cc` コマンドに与えればよい。

cc コマンドのオプション：

講義ノート 2.5 節の表, 等を参照。

14.2 自習 プロファイラを使う

{ ケリー&ポール 11.14 節 }

プロファイラ： プログラムの実行時の動作特性 (**実行プロファイル**という; e.g. 各関数の呼出し回数、実行時間) を測るためのツールで、**プロファイラ**と呼ばれるものがある。

参考：
profile ... *n.* 横顔、輪郭、[新聞] 人物紹介

- 実習室で使えるプロファイラは `gprof` である。
- プロファイラ `gprof` の使い方は次の通り。
  ① `gcc` コマンドを `-pg` オプション付きで、または `cc` コマンドを `-xpg` オプション付きで実行。

  補足：
  このオプション指定によって、途中の実行状況を `gmon.out` という名前のファイルに記録するためのコードも実行ファイルの中に埋め込まれる。 記録する内容は `-p` オプションの場合より詳しい。

② 前ステップ①で出来たコードを実行。

③ gprof 実行ファイルの名前

補足：
長い説明文を省略したければ -b オプションを付ける。

例 14.1 (gprofによる heapsort プログラムの実行プロファイル) 例題 13.2 で作成したヒープソートプログラムの実行プロファイルを実習室でgprof を使って表示してみよう。

```
[motoki@applsv1_10] gcc -pg check-sort-program.c btree-heapsort.c
[motoki@applsv1_11] ./a.out
Input a random seed (0 - 2147483647):  333

before sorting:
    556    289    435    368    666    319    214    273    132    585
     64    943    869    956     50    298    112    218      5    649
    603    936    515    385    671    776    137    886      4    563

          (途中省略)

[motoki@applsv1_12] gprof -b a.out
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00      149     0.00     0.00  heapify
  0.00      0.00     0.00        2     0.00     0.00  pretty_print
  0.00      0.00     0.00        1     0.00     0.00  set_an_array_random
  0.00      0.00     0.00        1     0.00     0.00  sort
  0.00      0.00     0.00        1     0.00     0.00  sort_method


Call graph


granularity: each sample hit covers 2 byte(s) no time propagated

index % time    self  children    called     name
                0.00    0.00     149/149         sort [4]
[1]      0.0    0.00    0.00     149         heapify [1]
-----------------------------------------------
                0.00    0.00       2/2           main [10]
```

```
[2]      0.0    0.00    0.00       2         pretty_print [2]
-----------------------------------------------
                0.00    0.00       1/1           main [10]
[3]      0.0    0.00    0.00       1         set_an_array_random [3]
-----------------------------------------------
                0.00    0.00       1/1           main [10]
[4]      0.0    0.00    0.00       1         sort [4]
                0.00    0.00     149/149         heapify [1]
-----------------------------------------------
                0.00    0.00       1/1           main [10]
[5]      0.0    0.00    0.00       1         sort_method [5]
-----------------------------------------------


Index by function name

   [1] heapify                 [3] set_an_array_random     [5] sort_method
   [2] pretty_print            [4] sort
[motoki@applsv1_13]
```

14.3 C プログラムの計算時間を測るには

- 計算機の内部計時にアクセスするために標準ライブラリ関数が用意されている。
  [これらの関数プロトタイプは <time.h> の中にある。$\Longrightarrow$ 講義ノート 4.7 節も参照。]

例題 14.2 (ヒープソートの計算時間を測る) 例題 13.2 で作成したソースプログラム check-sort-program.c に手を加えて、ヒープソートモジュール内の sort() 関数の計算時間を測って出力する様にせよ。

(考え方) 標準ライブラリ関数 clock(), time(), difftime() やマクロ CLOCKS_PER_SEC を使えば、プロセスの消費した時間 (**プロセッサ時間**,**CPU 時間**と言う; 測れる上限あり) やプロセス実行中の経過時間 (**カレンダー時間**, **実時間**と言う; 秒単位) をプログラムの中から知ることが出来る。

では、プロセッサ時間はどうやって測るのか？ 関数 clock_t clock(void) は、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を返す。但し、この時間は、一定周期で計算機の中でパルス (**クロック**という) が発生していると見て、それらのパルスを数えたもので表される。このクロック数を表すためのデータ型として clock_t というものが<time.h>の中で定義されている。

補足：
clock() の値は **0, 1, 2, 3, ...** と細かく変わる訳ではなく、平成 **13** 年度実習室(の PC) では 0 の次は 10000 になっていた。 これは、計算機内部で複数のプロセスを並行して走らせるために、CPU の使用権が 10000 クロック数程度にスライスされて各プロセスに配分されているためであろう。多分、完全に消費した時間スライス分だけがそのプロセスの消費したクロック数としてカウントされているものと思われる。
$\Longrightarrow$ あまり簡単な処理だと、clock() の値は 0 のまま終る。

<time.h>の中で定義されているマクロ定数 CLOCKS_PER_SEC が 1 秒当たりのクロック数 (i.e.1 秒が何クロック数に相当するか) を表しているので、clock( ) で得られた時間 (クロック数) を CLOCKS_PER_SEC で割れば秒単位の時間になる。

補足：
プロセスに割り当てられる時間スライスは 平成 **13** 年度実習室(の PC) では、最初は 10000 クロック数の様だから clock() 関数による時間計測の精度は 10000/CLOCKS_PER_SEC = 0.01 秒ということになる。

カレンダー時間はどうやって測るのか？ 関数 time_t time(time_t *tp) は、現在のカレンダー時刻 (i.e. 日付と時刻) を返すライブラリ関数である。このカレンダー時刻を表すためのデータ型として time_t というものが<time.h>の中で定義されている。

補足：
UNIX においては、グリニッジ標準時 1970 年 1 月 1 日 0 時 0 分 0 秒からの経過秒数が内部で保持され、ログインやファイル作成等の際に時間を記録するのに利用されている。 time( ) は、実際にはこの 1970 年 1 月 1 日 0 時 0 分 0 秒からの経過秒数を返す。
$\Longrightarrow$ time( ) による時間計測の精度は **1** 秒。

プログラムの開始直後と終了直前のカレンダー時刻が分かれば、その間の経過時間は difftime( ) 関数を使って difftime(終了直前のカレンダー時刻, 開始直後のカレンダー時刻) と表すことが出来る。

(プログラミング) 計算時間 (プロセッサ時間とカレンダー時間) も出力されるように例題 13.2 のヒープソートプログラムを書き換えて得られたプログラムと、これをコンパイル/実行している様子を次に示す。但し、コンパイルに使った Heapsort のモジュール btree-heapsort.c は例題 13.2 で使ったものと同じであるので、ここでは再掲示しない。

```
[motoki@x205a]$ nl clock-sort-100.c
     1 /*************************************************************/
     2 /* 要素数 100 の場合の整列化プログラムの実行時間を計る          */
     3 /*-----------------------------------------                  */
     4 /*     次の作業を 30000 回繰り返して所用時間を計り、          */
     5 /*     後で 1 回当りの計算時間を割り出す。                    */
     6 /*         | 大きさ 100 の配列にランダムに整数を生成し、      */
     7 /*         | その配列要素を別途用意された整列化プログラムを   */
     8 /*         | 使って昇順に並べ替える。                        */
     9 /*************************************************************/
```

```
10 #include <stdio.h>
11 #include <stdlib.h>   /* 乱数発生のライブラリ関数を使うため */
12 #include <time.h>

13 #define  SIZE   100

14 void set_an_array_random(int a[], int size);
15 void sort_method(char *method_name);
16 void sort(int a[], int size);

17 int main(void)
18 {
19   int      a[SIZE], seed, i;
20   char     sort_name[40];
21   clock_t  begin_clock, end_clock;
22   time_t   begin_time,  end_time;
23   double   process_time, real_time;

24   sort_method(sort_name);
25   printf("Clocking the execution time of the program that "
26          "sorts 100 elements.\n    (%s)\n"
27          "Input a random seed (0 - %d):  ",
28          sort_name, RAND_MAX);
29   scanf("%d", &seed);

30   begin_clock = clock();
31   begin_time  = time(NULL);

32   for (i=0; i<30000; ++i) {        /* 同じ処理を 30000 回繰 */
33     srand(seed);                   /* 返して時間を測り、   */
34     set_an_array_random(a, SIZE);  /* 後で 1 回当たりの計算 */
35     sort(a, SIZE);                 /* 時間を割り出す。     */
36   }

37   end_clock = clock();
38   end_time  = time(NULL);
39   process_time = (end_clock - begin_clock)
40                    / (CLOCKS_PER_SEC * 30000.0);
41   real_time    = difftime(end_time, begin_time) / 30000.0;
42   printf("Process time = %6.3f m sec\n"
43          "Real    time = %6.3f m sec\n",
44          process_time*1000.0, real_time*1000.0);
45   return 0;
```

```
46 }

47 /*-------------------------------------------------------*/
48 /* 引数で与えられた配列の各要素をランダムに設定する          */
49 /*---------------------------------------                 */
50 /* (仮引数) a    : int 型配列                              */
51 /*          size : int 型配列 a の大きさ                   */
52 /* (関数値)  :  なし                                      */
53 /* (機能) :  配列要素 a[0]〜a[size-1] に 0〜999 の間の乱数  */
54 /*           を設定する。                                 */
55 /*-------------------------------------------------------*/
56 void set_an_array_random(int a[], int size)
57 {
58   int i;

59   for (i=0; i<size; ++i)
60     a[i] = rand() % 1000;
61 }
[motoki@x205a]$ gcc clock-sort-100.c btree-heapsort.c
[motoki@x205a]$ ./a.out
Clocking the execution time of the program that sorts 100 elements.
    (Heapsort)
Input a random seed (0 - 2147483647):  333
Process time =  0.012 m sec
Real    time =  0.033 m sec
[motoki@x205a]$ ./a.out
Clocking the execution time of the program that sorts 100 elements.
    (Heapsort)
Input a random seed (0 - 2147483647):  333
Process time =  0.013 m sec
Real    time =  0.000 m sec
[motoki@x205a]$
```

ここで、

- プログラム 12 行目 の include 行は、時間に関する標準的なデータ型定義、マクロ定義、関数プロトタイプを取り込むためにある。
- プログラム 21 行目 で用いられているデータ型 clock_t は、各々の計算機で独自に設定されている時間量を表すためのものである。実習室では、<time.h> の中で

  ```
  typedef long    clock_t;
  ```

  と定義されている。
- プログラム 22 行目 で用いられているデータ型 time_t は、暦上の日付、時刻を表すためのものである。実習室では、<time.h> の中で

  ```
  typedef long    time_t;
  ```

  と定義されている。

- プログラム 30 行目, 37 行目 に現われる clock() は、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を返すライブラリ関数である。
- プログラム 31 行目, 38 行目 に現われる time() は、現在のカレンダー時間 (i.e. 日付と時刻) を返すライブラリ関数である。 引数が NULL 以外になっていれば、そのポインタが指す変数にもカレンダー時間がセットされる。
- プログラム 40 行目 に現われる CLOCKS_PER_SEC は、1 秒当たりのクロック数 (i.e.1 秒が何クロック数に相当するか) を明示したマクロ定数であり、**平成 13 年度実習室**(の PC) では <time.h> の中で

```
#define CLOCKS_PER_SEC          1000000
```

と定義されていた。
- プログラム 41 行目 に現われる difftime() は、第 2 引数のカレンダー時刻から第 1 引数のカレンダー時刻までの時間を秒単位で返すライブラリ関数である。
- 上の実行結果を見ると、同じ実行を行っても Real time の結果の方は全然違ったものになっている。 このことからも分かるように、time() と difftime() を用いて個々のプログラムの計算時間を正確に測ることは出来ない。
- このプログラムについて実行プロファイルを作成すると次の様になる。

```
[motoki@x205a]$ gcc -pg clock-sort-100.c btree-heapsort.c
[motoki@x205a]$ ./a.out
Clocking the execution time of the program that sorts 100 elements.
    (Heapsort)
Input a random seed (0 - 2147483647):  333
Process time =  0.015 m sec
Real    time =  0.000 m sec
[motoki@x205a]$ gprof -b a.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 88.00      0.22     0.22  4470000     0.05     0.05  heapify
  8.00      0.24     0.02    30000     0.67     0.67  set_an_array_random
  4.00      0.25     0.01    30000     0.33     7.67  sort
  0.00      0.25     0.00        1     0.00     0.00  sort_method


Call graph


granularity: each sample hit covers 4 byte(s) for 4.00% of 0.25 seconds

index % time    self  children    called     name
                                                 <spontaneous>
```

```
[1]    100.0    0.00    0.25                 main [1]
                0.01    0.22   30000/30000       sort [2]
                0.02    0.00   30000/30000       set_an_array_random [4]
                0.00    0.00       1/1           sort_method [5]
-----------------------------------------------
                0.01    0.22   30000/30000       main [1]
[2]     92.0    0.01    0.22   30000         sort [2]
                0.22    0.00 4470000/4470000     heapify [3]
-----------------------------------------------
                0.22    0.00 4470000/4470000     sort [2]
[3]     88.0    0.22    0.00 4470000         heapify [3]
-----------------------------------------------
                0.02    0.00   30000/30000       main [1]
[4]      8.0    0.02    0.00   30000         set_an_array_random [4]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[5]      0.0    0.00    0.00       1         sort_method [5]
-----------------------------------------------


Index by function name

   [3] heapify                 [2] sort
   [4] set_an_array_random     [5] sort_method
[motoki@x205a]$
```

補足：
これは VineLinux6 上での実行です。同じ gprof でも、Version によって出力量が違うようです。

例題 14.3 (時間計測のためのモジュール) 例題 14.2 からも分かるように、ヘッダファイル <time.h> の中に用意されたものだけを使ってプログラムの実行時間を計ろうとすると、データ型 clock_t, time_t やマクロ CLOCKS_PER_SEC を始めとして内部のことを十分に理解した上で、クロック数を時間に変換したりしなければならない。そこで、ストップウォッチを使う感覚で容易に実行時間を計るための汎用のモジュールを構成せよ。

(考え方) モジュール内の関数を 2 回呼び出して、2 回目の呼び出し時に 1 回目と 2 回目の呼び出しの間の時間が秒単位で返って来る様にできれば良い。そのために、1 回目に呼び出された関数は呼び出された時刻をモジュール内に記録し、2 回目に呼び出された関数は 1 回目の呼び出し時に記録された時刻からの経過時間を割り出して返す様にする。もちろん、1 回目の呼び出し時に内部に記録する時刻は外部からは見えなくすべきである。

また、扱える時間の種類は、プロセッサ時間とカレンダー時間の両方が良い。これら各々の種類の時間に対して上記のような関数等を別々に用意しても良いが、関数呼び出しの回数を少なくするために、プロセッサ時間とカレンダー時間の組 (構造体) が時間計測結果として返される様にする。

(プログラミング) 次の 4 つから成るモジュールを構成した。

- 前回の関数呼び出し時の時刻 (の組) を記録する変数 previous,
- 今回の関数呼び出し時の時刻 (の組) を記録する変数 current,
- 時間計測の最初に呼び出す関数 void start_timekeeper(void),
- 前回の関数呼び出しからの経過時間 (の組) を返す関数 Second consumed_time(void)

出来たモジュールを次に示す。

```
[motoki@x205a]$ nl consumed_time.c
   1  /*************************************************************/
   2  /* 計算時間を計るための汎用モジュール                        */
   3  /*-------------------------------------                      */
   4  /*    外部へのサービスを行うために、次の２つの関数がこの     */
   5  /*    モジュールの中に用意されている。                       */
   6  /*    (1)start_timekeeper ... 時間測定を開始させる関数       */
   7  /*    (2)consumed_time ... 以前に関数 start_timekeeper また */
   8  /*           は consumed_time が呼ばれた時点から現在までに   */
   9  /*           消費したプロセッサ時間 (秒) とカレンダー時間 (秒) */
  10  /*           の組み (構造体) を返す関数                      */
  11  /*************************************************************/

  12  #include <time.h>

  13  typedef struct {
  14     clock_t  clock;
  15     time_t   time;
  16  } Time;

  17  static Time  previous, current; /* 静的外部変数 ==>ファイル */
  18                                  /*          外からは見えない */
  19  typedef struct {
  20     double  process_time;
  21     double  real_time;
  22  } Second;

  23  /*-----------------------------------------------------------*/
  24  /* 計算時間を測定する際の開始時点を (静的外部変数に) 記録する */
  25  /*-----------------------------------------------------------*/
  26  void start_timekeeper(void)
  27  {
```

```
28    previous.clock = clock();
29    previous.time  = time(NULL);
30  }

31  /*-------------------------------------------------------------*/
32  /* 以前に start_timekeeper() または consumed_time() が       */
33  /* 呼ばれた時点から現在までに消費したプロセッサ時間 (秒)、     */
34  /* とカレンダー時間 (秒)、の組み (構造体) を返す。              */
35  /*-------------------------------------------------------------*/
36  Second consumed_time(void)
37  {
38    Second  consumed;

39    current.clock = clock();
40    current.time  = time(NULL);
41    consumed.process_time = (current.clock - previous.clock)
42                                      / (double) CLOCKS_PER_SEC;
43    consumed.real_time = difftime(current.time, previous.time);
44    previous = current;
45    return consumed;
46  }
```

例題 14.4 (ヒープソート vs. バブルソート vs. 線形リストを用いた挿入ソート) 例題 13.2 のヒープソートと、素朴な整列化アルゴリズムとして有名なバブルソート、それから例題 12.3 の様に線形リスト上に要素を昇順に配置しその結果を配列に戻すソート方法、の実際の計算時間を要素数が 5, 10, 25, 50, 100, 200 の時に各々比較せよ。

(考え方) 3 つの整列化手法の各々に対して全く同じ外部仕様の汎用モジュールを作っておけば、その仕様の整列化プログラムの計算時間を測るモジュールは 3 つの整列化手法間で共通で済む。 すなわち、ヒープソートの汎用モジュールと実行時間計測のモジュールを組み合わせてコンパイル・実行すればヒープソートの処理効率を調べることが出来るし、バブルソートの汎用モジュールと先程と同じ実行時間計測のモジュールを組み合わせてコンパイル・実行すればバブルソートの処理効率を調べることが出来るようになる。そこで、バブルソートと線形リスト上に要素を昇順に配置しその結果を配列に戻すソート手法の各々に対しても、例題 13.2 で作成したヒープソートモジュールと同じ外部仕様、すなわち次の 2 つの外部関数を持つ汎用の整列化モジュールを作ることにする。

- `void sort_method(char *method)` ··· 引数として指定された char 型配列に 整列化手法の名前を表す文字列を入れる関数,
- `void sort(int a[], int size)` ··· int 型配列の名前 `a[]` とその大きさ `size` を引数として受け取り、指定された配列内のデータを各々の整列化手法で小さい順に並べ替える関数

バブルソートについては、例えば「川合、(岩波講座ソフトウェア科学2) プログラミングの方法、岩波書店」p.176~177 を参照して下さい。

では、整列化プログラムの計算時間を測るモジュールとしてはどの様なものを作るか?
要素数が 5, 10, 25, 50, 100, 200 の時の計算時間を調べることが要求されているが、計算時間は与えられた問題例に依存する。 そこで、ここでは各々の要素数に対して、

> 具体的な問題例 (i.e. 配列の初期データ配置) をランダムに設定して、
> その問題例に対して整列化プログラムを実行する

という作業を繰り返してそれらの時間を計ることにより、平均的な計算時間を求めることにする。繰り返しの総時間が短くなり過ぎると計算時間が正確に計れないし、また繰り返しの総時間が大くなり過ぎると計算資源の無駄な利用にもなるので、これらのことを考慮に入れて、要素数が 5, 10, 25, 50, 100, 200 の場合に対して　問題例のランダムな設定, 整列化プログラムの実行　を各々 800000 回, 400000 回, 160000 回, 80000 回, 40000 回, 20000 回 繰り返すことにした。 (この回数の選び方はコンピュータに依存する。)　実際の時間計測には例題 14.3 で挙げた時間計測のためのモジュール `consumed_time.c` を利用すれば良い。

(整列化プログラムの計算時間を測るモジュールのプログラミング)
要素数が 5, 10, 25, 50, 100, 200 の各々の場合に対して平均計算時間を割り出す。 これらの平均計算時間の結果を表の形に見易く表すためには全ての実行を 1 つのプログラムで表すことが必要である。その際、整列化する要素数と整列化の繰り返し回数を組 (構造体) にして処理の順に配列に入れておけば、作業全体を `for` 文による繰り返しでコンパクトに表すことが出来る。 この理由で次の構造体を導入する。

```
typedef struct {
   int  size;     /* 整列化する要素数      */
   int  ite_num;  /* 整列化の繰り返し回数 */
} Problem;
```

そして、出来たモジュールは次の通りである。

```
[motoki@x205a]$ nl clock-sort-5-10-etc.c
   1 /*************************************************************/
   2 /* 要素数が 5，10，25，50，100，200 の場合について、          */
   3 /* 整列化プログラムの平均実行時間を計る                       */
   4 /*-------------------------------------------                */
   5 /*   要素数が 5，10，25，50，100，200 の場合について、それ  */
   6 /*   ぞれ 800000 回，400000 回，160000 回，80000 回，40000 回，  */
   7 /*   20000 回 次の作業を繰り返して所用時間を計り、後で 1 回当り */
   8 /*   の計算時間を割り出す。                                   */
   9 /*       |配列上に要素数分だけランダムに整数を生成し、       */
  10 /*       |その配列要素を別途用意された整列化プログラムを     */
  11 /*       |使って昇順に並べ替える。                           */
  12 /*************************************************************/

  13 #include <stdio.h>
  14 #include <stdlib.h>
```

```
15 #define  MAX_SIZE  200
16 #define  PROB_NUM  6

17 typedef struct {
18    int  size;
19    int  ite_num;
20 } Problem;

21 typedef struct {
22    double  process_time;
23    double  real_time;
24 } Second;

25 void   start_timekeeper(void);
26 Second consumed_time(void);
27 void set_an_array_random(int a[], int size);
28 void sort_method(char *method_name);
29 void sort(int a[], int size);

30 int main(void)
31 {
32   int     a[MAX_SIZE], seed, i, k;
33   char    sort_name[100];
34   Problem prob[PROB_NUM] = {{5,800000}, {10,400000}, {25,160000},
35                             {50,80000}, {100,40000}, {200,20000}};
36   Second  time_for_sort[PROB_NUM],
37           time_for_init[PROB_NUM];

38   sort_method(sort_name);
39   printf("Clocking the average execution time of the program\n"
40          "that sorts 5, 10, 25, 50, 100, or 200 elements.\n"
41          "  (*** %s ***)\n"
42          "Input a random seed (0 - %d):  ",
43          sort_name, RAND_MAX);
44   scanf("%d", &seed);

45   /* ソート以外の部分の計算時間を測定 */
46   srand(seed);
47   for (k=0; k<PROB_NUM; ++k) {
48     start_timekeeper();
49     for (i=0; i<prob[k].ite_num; ++i) {     /* この部分の */
50       set_an_array_random(a, prob[k].size); /* 計算時間を */
51     }                                       /* 参考のために測る。*/
```

```
52     time_for_init[k] = consumed_time();
53   }

54   /* ソートプログラムの平均計算時間を測定 */
55   srand(seed);
56   for (k=0; k<PROB_NUM; ++k) {
57     start_timekeeper();
58     for (i=0; i<prob[k].ite_num; ++i) {     /* この部分の */
59       set_an_array_random(a, prob[k].size); /* 計算時間を */
60       sort(a, prob[k].size);                /* 測る。      */
61     }                                       /*            */
62     time_for_sort[k] = consumed_time();
63                 /* 次に sort 部分だけの計算時間を割り出す。 */
64     time_for_sort[k].process_time -= time_for_init[k].process_time;
65     time_for_sort[k].real_time    -= time_for_init[k].real_time;
66   }

67   /* 測定結果の出力 */
68   printf("\n"
69          "        ** time for sort **    **time for initialize**\n"
70          "size    process_t  real_time    process_t  real_time\n"
71          "         (m sec)    (m sec)      (m sec)    (m sec)\n"
72          "----    ---------  ---------    ---------  ---------\n");
73   for (k=0; k<PROB_NUM; ++k)
74     printf("%4d  %11.5f%11.5f  %11.5f%11.5f\n",
75            prob[k].size,
76            time_for_sort[k].process_time*1000.0/prob[k].ite_num,
77            time_for_sort[k].real_time*1000.0/prob[k].ite_num,
78            time_for_init[k].process_time*1000.0/prob[k].ite_num,
79            time_for_init[k].real_time*1000.0/prob[k].ite_num);
80   return 0;
81 }

82 /*-------------------------------------------------------------*/
83 /* 引数で与えられた配列の各要素をランダムに設定する          */
84 /*-----------------------------------------                    */
85 /* (仮引数) a    : int 型配列                                  */
86 /*          size : int 型配列 a の大きさ                       */
87 /* (関数値)  :  なし                                          */
88 /* (機能) :  配列要素 a[0]～a[size-1] に 0～999 の間の乱数   */
89 /*           を設定する。                                      */
90 /*-------------------------------------------------------------*/
91 void set_an_array_random(int a[], int size)
```

```
    92 {
    93   int i;

    94   for (i=0; i<size; ++i)
    95     a[i] = rand() % 1000;
    96 }
[motoki@x205a]$
```

(各々の整列化手法における計算時間の計測)

(1) ヒープソートの平均計算時間について： 例題 14.3 で挙げた時間計測のための汎用モジュール consumed_time.c, 上で挙げた 整列化プログラムの計算時間を測るモジュール, および例題 13.2 で作成した Heapsort のモジュール btree-heapsort.c を組み合わせてコンパイルすることによって、次の様に計測することが出来ます。

```
[motoki@x205a]$ gcc clock-sort-5-10-etc.c consumed_time.c btree-heapsort.c
[motoki@x205a]$ ./a.out
Clocking the average execution time of the program
that sorts 5, 10, 25, 50, 100, or 200 elements.
  (*** Heapsort ***)
Input a random seed (0 - 2147483647):  333

        ** time for sort **    **time for initialize**
size    process_t  real_time    process_t  real_time
         (m sec)    (m sec)      (m sec)    (m sec)
----    ---------  ---------    ---------  ---------
   5      0.00014    0.00000      0.00009    0.00000
  10      0.00040    0.00250      0.00020    0.00000
  25      0.00156    0.00000      0.00044    0.00000
  50      0.00362    0.00000      0.00100    0.00000
 100      0.00900    0.02500      0.00175    0.00000
 200      0.02050    0.00000      0.00350    0.00000
[motoki@x205a]$
```

(2) バブルソートの平均計算時間について： 2 つの関数 sort_method (整列化アルゴリズムの名前を答える) と sort (引数で指定された配列要素を bubblesort アルゴリズムで昇順に並べ替える) から成るモジュール bubblesort.c を作れば、あとは、これと先程使った 2 つのモジュール clock-sort-5-10-etc.c と consumed_time.c を組み合わせてコンパイル・実行するだけである。 バブルソートの汎用モジュールと、コンパイル・実行の様子を次に示す。

```
[motoki@x205a]$ nl bubblesort.c
     1 /*************************************************************/
     2 /* Bubblesort モジュール                                     */
     3 /*-------------------------------------                     */
     4 /*     外部へのサービスを行うために、次の 2 つの関数がこの     */
```

```
 5 /*    モジュールの中に用意されている。                              */
 6 /*    (1) 整列化アルゴリズムの名前を答える関数 sort_method,  */
 7 /*    (2) 配列要素を Bubblesort アルゴリズムで                    */
 8 /*                    小さい順に並べ替える関数 sort              */
 9 /*************************************************************/

10 #include <stdio.h>

11 /*-----------------------------------------------------------*/
12 /* 整列化アルゴリズムの名前を答える                            */
13 /*-------------------------------------------                 */
14 /* (引数) method_name : 出力用。この char 型配列に方式の名前  */
15 /*                      を (文字列として) 入れて返す。配列の  */
16 /*                      大きさは 11 文字分必要。              */
17 /*-----------------------------------------------------------*/
18 void sort_method(char *method_name)
19 {
20   sprintf(method_name, "Bubblesort");
21 }

22 /*-----------------------------------------------------------*/
23 /* 配列要素を小さい順に並べ替える ( bubblesort)               */
24 /*-------------------------------------------                 */
25 /* (仮引数) a    : int 型配列                                 */
26 /*          size : int 型配列 a の大きさ                      */
27 /* (関数値)  :  なし                                         */
28 /* (機能) :  bubblesort アルゴリズムを使って、配列要素       */
29 /*               a[0],a[1],a[2], ..., a[size-1]              */
30 /*           を値の小さい順に並べ替える。                    */
31 /*-----------------------------------------------------------*/
32 void sort(int a[], int size)
33 {
34   int i, j, temp;

35   for (i=0; i<size-1; ++i)
36     for (j=size-1; j > i; --j)
37       if (a[j-1] > a[j]) {      /* a[j-1] と a[j]    */
38         temp   = a[j-1];        /* の大小を調べて、  */
39         a[j-1] = a[j];          /* 逆順なら交換する。*/
40         a[j]   = temp;
41       }
42 }
[motoki@x205a]$ gcc clock-sort-5-10-etc.c consumed_time.c bubblesort.c
```

```
[motoki@x205a]$ ./a.out
Clocking the average execution time of the program
that sorts 5, 10, 25, 50, 100, or 200 elements.
  (*** Bubblesort ***)
Input a random seed (0 - 2147483647):  333

        ** time for sort **    **time for initialize**
size    process_t  real_time    process_t  real_time
         (m sec)    (m sec)      (m sec)    (m sec)
----    ---------  ---------    ---------  ---------
   5      0.00010    0.00000      0.00010    0.00000
  10      0.00050    0.00000      0.00018    0.00000
  25      0.00262    0.00000      0.00050    0.00000
  50      0.00975    0.01250      0.00100    0.00000
 100      0.03925    0.05000      0.00175    0.00000
 200      0.15800    0.10000      0.00400    0.05000
[motoki@x205a]$
```

(3) 線形リストを用いた挿入ソートの平均計算時間について： 2 つの関数

- sort_method ··· 整列化アルゴリズムの名前を答える関数、
- sort ··· 引数で指定された配列要素を例 12.3 の様に線形リスト上に昇順に配置しその結果を配列に戻すことによって小さい順に並べ替える関数、

から成るモジュール llistsort.c を作れば、あとは、これと先程使った 2 つのモジュール clock-sort-5-10-etc.c と consumed_time.c を組み合わせてコンパイル・実行するだけである。 このソート法の汎用モジュール llistsort.c と、コンパイル・実行の様子を次に示す。

```
[motoki@x205a]$ nl llistsort.c
     1 /*************************************************************/
     2 /* Llistsort モジュール                                      */
     3 /*-------------------------------------                     */
     4 /*    外部へのサービスを行うために、次の 2 つの関数がこの    */
     5 /*    モジュールの中に用意されている。                       */
     6 /*    (1) 整列化アルゴリズムの名前を答える関数 sort_method,  */
     7 /*    (2) 引数で指定された配列要素を線形リスト上             */
     8 /*        に昇順に挿入していき、その結果を配列に             */
     9 /*        戻すことによって小さい順に並べ替える関数  sort     */
    10 /*************************************************************/

    11 #include <stdio.h>
    12 #include <stdlib.h>

    13 #define Is_empty(list)  ((list) == NULL)
```

```
14 typedef  struct list_item  *List;
15 typedef struct list_item {
16     int    num;
17     List   next;
18 } List_item;

19 static void add_item(List *ptr_ptr, int num);

20 /*-------------------------------------------------------------*/
21 /* 整列化アルゴリズムの名前を答える                            */
22 /*-----------------------------------------                    */
23 /* (引数) method_name : 出力用。この char 型配列に方式の名前  */
24 /*                      を (文字列として) 入れて返す。配列の  */
25 /*                      大きさは 38 文字分必要。              */
26 /*-------------------------------------------------------------*/
27 void sort_method(char *method_name)
28 {
29   sprintf(method_name, "Sorting by Insertion in linked list");
30 }

31 /*-------------------------------------------------------------*/
32 /* 配列要素を小さい順に並べ替える (線形リスト上での挿入繰返し) */
33 /*-----------------------------------------                    */
34 /* (仮引数) a    : int 型配列                                  */
35 /*          size : int 型配列 a の大きさ                       */
36 /* (関数値)  :  なし                                          */
37 /* (機能) :  配列要素                                         */
38 /*               a[0],a[1],a[2], ..., a[size-1]               */
39 /*           線形リスト上に昇順に挿入していき、その結果を     */
40 /*           配列に戻すことによって小さい順に並べ替える       */
41 /*-------------------------------------------------------------*/
42 void sort(int a[], int size)
43 {
44   int  i;
45   List list, head_item;

46   list = NULL;
47   for (i=0; i<size; i++)
48     add_item(&list, a[i]);

49   for (i=0; i<size; i++) {
50     head_item = list;             /* 先頭の構造体を          */
51     list      = list->next;       /* 線形リストから切り離す */
```

```
    52     a[i] = head_item->num;
    53     free(head_item);            /* 不要になったメモリを解放 */
    54   }
    55 }

    56 /*---------------------------------------------------------------*/
    57 /* 線形リストへ整数を追加                                        */
    58 /*---------------------------------------------------------------*/
    59 /* (仮引数) ptr_ptr : "線形リストへのポインタ領域"へのポインタ  */
    60 /*          num     : 整数                                       */
    61 /* (関数値) : なし                                               */
    62 /* (機能)   : 整数が小さい順に線形リストの形に繋げられており、  */
    63 /*            *ptr_ptr がその線形リストの先頭を指し示すと仮定す */
    64 /*            る。この仮定の下で、整数 num を新しい項目として   */
    65 /*            (小さい順を保つ様に)線形リストに挿入する。        */
    66 /*---------------------------------------------------------------*/
    67 static void add_item(List *ptr_ptr, int num)
    68 {
    69   List_item *new_item;

    70   while (! Is_empty(*ptr_ptr) && (*ptr_ptr)->num < num)
    71     ptr_ptr = &((*ptr_ptr)->next);          /* 挿入場所を探す */

    72   new_item = (List_item *) malloc(sizeof(List_item));
    73                                           /* 新しい item 枠を作る */
    74   if (new_item == NULL) {
    75     printf("*** fail in memory allocation ***");
    76     exit(EXIT_FAILURE);
    77   }
    78   new_item->num = num;

    79   new_item->next  = *ptr_ptr;             /* ポインタの付け替え */
    80   *ptr_ptr = new_item;
    81 }
[motoki@x205a]$ gcc clock-sort-5-10-etc.c consumed_time.c llistsort.c
[motoki@x205a]$ ./a.out
Clocking the average execution time of the program
that sorts 5, 10, 25, 50, 100, or 200 elements.
  (*** Sorting by Insertion in linked list ***)
Input a random seed (0 - 2147483647):  333

        ** time for sort **    **time for initialize**
size    process_t  real_time    process_t  real_time
```

```
          (m sec)    (m sec)      (m sec)    (m sec)
----    ---------  ---------    ---------  ---------
   5      0.00029    0.00000      0.00010    0.00000
  10      0.00068    0.00000      0.00018    0.00000
  25      0.00206    0.00625      0.00050    0.00000
  50      0.00550    0.00000      0.00088    0.00000
 100      0.01600    0.00000      0.00175    0.02500
 200      0.05050    0.05000      0.00400    0.00000
[motoki@x205a]$
```

14.4 make コマンドによる自動分割コンパイル

例題 13.2(heapsort) や例題 14.4(heapsort,bubblesort 等の計算効率比較) で行ったプログラミング作業においては、プログラムは幾つかのモジュールに分けて作られていた。これらの例題の中で使われたモジュールは次の 6 つである。

```
check-sort-program.c  }
btree-heapsort.c      } ···(例題 13.2 で定義)
bubblesort.c          }
llistsort.c           } ···(例題 14.4 で定義)
clock-sort-5-10-etc.c }
consumed_time.c          ···(例題 14.3 で定義)
```

これらのモジュールを組み合わせて、例えば

```
cc check-sort-program.c btree-heapsort.c
```

とコマンド入力すればヒープソートの実行を要素数 100 で行うためのコンパイルになるし、また、同じ btree-heapsort.c モジュールを使って

```
cc clock-sort-5-10-etc.c consumed_time.c btree-heasort.c
```

とコマンド入力すればヒープソートの平均実行時間を要素数 5, 10, 25, 50, 100, 200 に対して計測するためのコンパイルになった。main 関数を含むモジュールの選び方が 2 通り、sort モジュールの選び方が 3 通りであるため、上の 6 つのモジュールを組み合わせることによって全部で $2\times3 = 6$ 通りのコンパイルが可能である。

ところが、もっと大規模なプログラムを多数のソースファイルに分割した場合、どれとどれを組み合わせてどういうオプション指定をしてコンパイルするかの記述が相当長く (場合によっては複雑にも) なり、コンパイルの度に cc コマンドの引数やオプションを打ち込むのが煩わしくなる。 こんな時のために、UNIX ではコンパイルの仕方の詳細をテキストファイル (**メイクファイル**という) に書いておき、make コマンドにその指定に従ったコンパイルを行わせることが出来る様になっている。

例 14.5 (make コマンドを用いた分割コンパイル; 最も素朴な版) 例題 13.2(heapsort) と例題 14.4(sort の計算効率比較) で行ったプログラミング作業は make コマンドを用いて例えば次の様に行うことが出来る。

(1) ディレクトリを 1 つ作り、その中に関連するソースファイルを構成する。例えば、

```
[motoki@x205a]$ mkdir Compare-sort
[motoki@x205a]$ cd Compare-sort
/home/motoki/C-Java2003/Programs-C/Compare-sort
[motoki@x205a]$   ......
```

(ソースファイルを作る)

```
                ......
[motoki@x205a]$ ls
btree-heapsort.c        check-sort-program.c    consumed_time.c
bubblesort.c            clock-sort-5-10-etc.c   llistsort.c
[motoki@x205a]$
```

(2) (1) で作ったディレクトリの中に Makefile または makefile という名前のメイクファイルを作る。例えば、

```
[motoki@x205a]$ ls
Makefile                check-sort-program.c    llistsort.c
btree-heapsort.c        clock-sort-5-10-etc.c
bubblesort.c            consumed_time.c
[motoki@x205a]$ nl Makefile
     1 # Makefile for clocking or checking 3 sort programs:
     2 #     (1) Heapsort
     3 #     (2) Bubblesort
     4 #     (3) Sort by insertion over linked-list

     5 CC    = gcc

     6 SRCS_CLOCK_HEAP  = clock-sort-5-10-etc.c consumed_time.c \
                          btree-heapsort.c
     7 SRCS_CLOCK_BUBBL = clock-sort-5-10-etc.c consumed_time.c \
                          bubblesort.c
     8 SRCS_CLOCK_LLIST = clock-sort-5-10-etc.c consumed_time.c \
                          llistsort.c
     9 SRCS_CHECK_HEAP  = check-sort-program.c btree-heapsort.c
    10 SRCS_CHECK_BUBBL = check-sort-program.c bubblesort.c
    11 SRCS_CHECK_LLIST = check-sort-program.c llistsort.c

    12 clock_heap: ${SRCS_CLOCK_HEAP}
    13 [ tab ] ${CC} -o clock_heap ${SRCS_CLOCK_HEAP}

    14 clock_bubble: ${SRCS_CLOCK_BUBBL}
    15 [ tab ] ${CC} -o clock_bubble ${SRCS_CLOCK_BUBBL}

    16 clock_llist: ${SRCS_CLOCK_LLIST}
    17 [ tab ] ${CC} -o clock_llist ${SRCS_CLOCK_LLIST}
```

```
18 check_heap: ${SRCS_CHECK_HEAP}
19 [ tab ] ${CC} -o check_heap ${SRCS_CHECK_HEAP}

20 check_bubble: ${SRCS_CHECK_BUBBL}
21 [ tab ] ${CC} -o check_bubble ${SRCS_CHECK_BUBBL}

22 check_llist: ${SRCS_CHECK_LLIST}
23 [ tab ] ${CC} -o check_llist ${SRCS_CHECK_LLIST}
[motoki@x205a]$
```

ここで、

- Makefile の 1~4 行目は、#で始まっているので**注釈** である。
- Makefile の 5~11 行目は **マクロ定義** である。例えば 6 行目は SRCS_CLOCK_HEAP というマクロ名を定義しており、これ以後 ${SRCS_CLOCK_HEAP} または $(SRCS_CLOCK_HEAP) という文字列が現われたら、その部分は

  ```
  clock-sort-5-10-etc.c consumed_time.c btree-heapsort.c
  ```

  という文字列に展開される。
- Makefile の 12~13 行目 は、clock_heap というファイルの最新版を確保するために何をしなければならないか、の規則を記述している。この規則により、
  - ◇ もし clock_heap というファイルが無かったり、あっても、それが作成された日付がコロンの右側のファイルの日付よりも古い様なことがあれば、最新版を確保するために 13 行目のコマンドを実行する、

  という (make への) 指示を行っている。 14~15 行目, 16~17 行目, 18~19 行目, 20~21 行目, 22~23 行目 の 5 つの規則についても同様である。これらの規則は、ターゲットファイルを生成するための依存情報を記述した部分 (**依存関係行**という) と、ターゲットファイルを生成するために実行するコマンドを記述した部分 (**コマンド行**という)、の 2 つから構成されている。12~23 行目の中では、12 行目,14 行目,16 行目,18 行目,20 行目,22 行目が依存関係行、残りがコマンド行である。
- Makefile の 13 行目, 15 行目, 17 行目, 19 行目, 21 行目, 23 行目 は**コマンド行であるのでtabコードで始まっている。**

(3) (1) で作ったディレクトリの中で、例えば

```
make clock_heap
```

とコマンド入力すれば、Makefile の 12~13 行目の規則が適用される。すなわち、clock_heap というファイルが無い場合、またはあってもそれが作成された日付がコロンの右側の (どれかの) ファイルの日付よりも古い場合にのみ、13 行目のコマンドが実行され (エラーが無ければ) clock_heap という名前の実行ファイルが生成される。一般に、

```
make target
```

とコマンド入力すると、Makefile の中から *target* で始まる規則が見つけ出され、それが適用される。 また、*target* を省略して

```
make
```

とコマンド入力すると、Makefile の中で最初に現われる 12~13 行目の規則が適用さ

れる。

make コマンドの使用例：

```
[motoki@x205a]$  make clock_bubble
gcc -o clock_bubble clock-sort-5-10-etc.c consumed_time.c bubblesort.c
```

実際に実行されるコマンドが表示される。

```
[motoki@x205a]$ ./clock_bubble
Clocking the average execution time of the program
that sorts 5, 10, 25, 50, 100, or 200 elements.
  (*** Bubblesort ***)
Input a random seed (0 - 2147483647):  333

        ** time for sort **    **time for initialize**
size    process_t  real_time    process_t  real_time
         (m sec)    (m sec)      (m sec)    (m sec)
----    ---------  ---------    ---------  ---------
   5      0.00011    0.00000      0.00009    0.00000
  10      0.00047   -0.00250      0.00020    0.00250
  25      0.00262    0.00625      0.00044    0.00000
  50      0.00988    0.01250      0.00100    0.00000
 100      0.03900    0.02500      0.00175    0.00000
 200      0.15850    0.15000      0.00350    0.00000
[motoki@x205a]$ make
gcc -o clock_heap clock-sort-5-10-etc.c consumed_time.c btree-heapsort.c
[motoki@x205a]$ ls
Makefile                check-sort-program.c    clock_heap*
btree-heapsort.c        clock-sort-5-10-etc.c   consumed_time.c
bubblesort.c            clock_bubble*           llistsort.c
[motoki@x205a]$ make clock_bubble "CC=gcc -g"
make: ‘clock_bubble’ は更新済です
```

補足：
生成目標である clock_bubble は既に出来ており、出来て以降原材料である 3 つのソースファイルは更新されていないので、コンパイルしても同じものを上書きするだけと判断され、コンパイルは新たに行われなかった。

make コマンドの最後に付いている "CC=gcc -g" はメイクファイルの中で定義された CC というマクロを再定義するものである。 こういう使い方が出来るので、Makefile の記述の中にわざわざ CC = gcc というマクロ定義を置いているのである。

```
[motoki@x205a]$ rm clock_bubble
rm: ‘clock_bubble’ を削除しますか (yes/no)? y
[motoki@x205a]$ make clock_bubble "CC=gcc -g"
gcc -g -o clock_bubble clock-sort-5-10-etc.c consumed_time.c bubblesort.c
```

マクロ再定義の指示 "CC=gcc -g" に従って確かに -g オプション付きでコンパイルされている。

```
[motoki@x205a]$
```

メイクファイルを作る際、コンパイルとリンクの作業を別にして各ソースファイルに対して .o ファイルも作るように指定すれば、同一のソースファイルを何度も繰り返しコンパイルする無駄を避けることによって、コンパイルの計算効率を上げることが出来る。

例 14.6 (make コマンドを用いた分割コンパイル;.o ファイルも作る版) 例 14.5 において Makefile を例えば次の様に書いておけば .o ファイルも生成される様になり、これによって同一のソースファイルのコンパイルが高々1 回に抑えられる様になる。

```
[motoki@x205a]$ ls
Makefile                bubblesort.c            consumed_time.c
Makefile.simple         check-sort-program.c    llistsort.c
btree-heapsort.c        clock-sort-5-10-etc.c
[motoki@x205a]$ nl Makefile
     1 # Makefile for clocking or checking 3 sort programs:
     2 #     (1) Heapsort
     3 #     (2) Bubblesort
     4 #     (3) Sort by insertion over linked-list

     5 CC    = gcc

     6 OBJS_CLOCK_HEAP  = clock-sort-5-10-etc.o consumed_time.o \
     7                    btree-heapsort.o
     8 OBJS_CLOCK_BUBBL = clock-sort-5-10-etc.o consumed_time.o \
     9                    bubblesort.o
    10 OBJS_CLOCK_LLIST = clock-sort-5-10-etc.o consumed_time.o \
    11                    llistsort.o
    12 OBJS_CHECK_HEAP  = check-sort-program.o btree-heapsort.o
    13 OBJS_CHECK_BUBBL = check-sort-program.o bubblesort.o
    14 OBJS_CHECK_LLIST = check-sort-program.o llistsort.o

    15 clock_heap: ${OBJS_CLOCK_HEAP}
    16 [ tab ] ${CC} -o clock_heap ${OBJS_CLOCK_HEAP}

    17 clock_bubble: ${OBJS_CLOCK_BUBBL}
    18 [ tab ] ${CC} -o clock_bubble ${OBJS_CLOCK_BUBBL}

    19 clock_llist: ${OBJS_CLOCK_LLIST}
    20 [ tab ] ${CC} -o clock_llist ${OBJS_CLOCK_LLIST}

    21 check_heap: ${OBJS_CHECK_HEAP}
    22 [ tab ] ${CC} -o check_heap ${OBJS_CHECK_HEAP}

    23 check_bubble: ${OBJS_CHECK_BUBBL}
    24 [ tab ] ${CC} -o check_bubble ${OBJS_CHECK_BUBBL}
```

```
25 check_llist: ${OBJS_CHECK_LLIST}
26 [tab] ${CC} -o check_llist ${OBJS_CHECK_LLIST}

27 clock-sort-5-10-etc.o: clock-sort-5-10-etc.c
28 [tab] ${CC} -c clock-sort-5-10-etc.c

29 consumed_time.o: consumed_time.c
30 [tab] ${CC} -c consumed_time.c

31 check-sort-program.o: check-sort-program.c
32 [tab] ${CC} -c check-sort-program.c

33 btree-heapsort.o: btree-heapsort.c
34 [tab] ${CC} -c btree-heapsort.c

35 bubblesort.o: bubblesort.c
36 [tab] ${CC} -c bubblesort.c

37 llistsort.o: llistsort.c
38 [tab] ${CC} -c llistsort.c

39 clean:
40 [tab] for i in *.o clock_heap clock_bubble clock_llist \
41 [tab]              check_heap check_bubble check_llist ; do \
42 [tab]   if [ -f $$i ] ; then rm $$i ; fi \
43 [tab] done
[motoki@x205a]$
```

ここで、

- 例 14.5 の場合と同様に、Makefile の 1~4 行目は、#で始まっているので注釈である。
- 例 14.5 の場合と同様に、Makefile の 5~14 行目は マクロ定義である。
- Makefile の例えば 15~16 行目 は、clock_heap というファイルの最新版を確保するために何をしなければならないか、の規則を記述している。この規則により、まず
  - ① 別の規則 (この場合は 27~28 行目, 29~30 行目, 33~34 行目 の 3 つ) を用いてコロン (:) の右側で指定されている clock-sort-5-10-etc.o, consumed_time.o, btree-heapsort.o の最新版を確保し、その結果、
  - ② もし clock_heap というファイルが無かったり、あっても、それが作成された日付がコロンの右側のファイルの日付よりも古い様なことがあれば、最新版を確保するために 16 行目のコマンドを実行する、

  という (make への) 指示を行っている。 17~26 行目 の 5 つの規則についても同様である。
- Makefile の 27~28 行目 は、clock-sort-5-10-etc.o というファイルの最新版を確保するために何をしなければならないか、の規則を記述している。この規則により、

◇ もし clock-sort-5-10-etc.o というファイルが無かったり、あっても、それが作成された日付がコロンの右側のファイルの日付よりも古い様なことがあれば、最新版を確保するために 28 行目のコマンドを実行する、

という (make への) 指示を行っている。 29~38 行目 の 5 つの規則についても同様である。

- 実際には 27~38 行目 の規則を一般化した

```
.SUFFIXES : .o .c
.c.o :
    tab  $(CC) $(CFLAGS) -c $<
```

という風な規則 (この種のものを**サフィックス規則**という) が暗黙に仮定されているので、27~38 行目の 6 つの規則は省略可能である。

補足：

◇ .SUFFIXES で始まる行 は、どういうサフィックスを持ったファイルをサフィックス規則の処理の対象とするかを指定しています。

◇ .c.o で始まる行 は、依存関係行に相当するもので、*name*.c という名前のファイルから *name*.o という名前のファイルを作り出す規則の始まりを表しています。

◇ tab で始まる行 は、コマンド行に相当するもので、実際にターゲットファイルを作り出す方法が書かれています。 その中の CFLAGS は暗黙のマクロで、コマンドのオプションを指定する時のために用意されています。また、最後の $< も暗黙のマクロで、ターゲット (この場合は *name*.o) よりも後で変更された**コンポーネント** (i.e. ターゲットを作り出すために必要なファイルのリスト; この場合は*name*.c, または 空リスト) を表す。

補足：どういう規則やマクロが暗黙に仮定されているかを知るには、例えば次の様にします。

```
[motoki@x205a]$ make -p -f /dev/null |more
      ......
SUFFIXES := .out .a .ln .o .c .cc .C .cpp .p .f .F .r .y .l .s .S
.mod .sym .def .h .info .dvi .tex .texinfo .texi .txinfo .w .ch
.web .sh .elc .el
      ......
CC = cc
      ......
OUTPUT_OPTION = -o $@
      ......
COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
      ......
.c.o:
$(COMPILE.c) $(OUTPUT_OPTION) $<
      ......
```

- 39~43 行目 の規則についても、他の 15~38 行目の規則と同様に使うことが出来る。ただ、

```
make clean
```

とコマンド入力した場合は clean というファイルは (多分) 無いので、常に 40~43 行目のコマンドが実行されることになる。メイクファイルの中でシェル変数を参照する場合は、マクロ参照と区別するために 42 行目の $$i のように $ 記号を 2 つ重ねて指定する。

補足：
40〜43 行目は、make で生成される .o ファイル, および実行ファイルを全て消去する作業を記述した Bourne シェルスクリプトです。
オプション変更してコンパイルし直したい場合 (e.g.make "CC=gcc -g") のように、過去に生成した実行コード群を全て消去したいこともあります。こういう時のために、この規則を用意しました。

- 例 14.5 の場合と同様に、

  ```
  make
  ```

  とだけコマンド入力すると Makefile の中で最初に現われる 15〜16 行目の規則が適用される。

このメイクファイルを使えば、例えば check_llist という実行ファイルの最新版を得るための作業は、(このメイクファイルのあるディレクトリの中で)

```
make check_llist
```

とコマンド入力するだけである。プログラムの一部に修正を加えた場合も、make *target* とコマンド入力するだけで必要最小限の箇所だけ再コンパイルを行ってくれる。[どの再コンパイルが必要なのかは、メイクファイルの中に書かれているプログラムの構成要素間の依存関係と、各ファイルの生成年月日を見て make が判断してくれる。]

```
[motoki@x205a]$ make check_llist
gcc -c check-sort-program.c
gcc -c llistsort.c
gcc -o check_llist check-sort-program.o llistsort.o
[motoki@x205a]$ ls
Makefile                check-sort-program.c    consumed_time.c
Makefile.simple         check-sort-program.o    llistsort.c
btree-heapsort.c        check_llist*            llistsort.o
bubblesort.c            clock-sort-5-10-etc.c
[motoki@x205a]$ ./check_llist
Input a random seed (0 - 2147483647):  333

before sorting:
    556    289    435    368    666    319    214    273    132    585
     64    943    869    956     50    298    112    218      5    649
              (途中省略)
    985    782    857    881    252    236    706    945    736    730

after sorting (Sorting by Insertion in linked list):
      4      5      6     18     27     50     61     64    104    105
    110    112    132    137    144    153    181    204    208    214
              (途中省略)
    917    936    943    945    950    951    956    957    985    998
[motoki@x205a]$ make clean
for i in *.o clock_heap clock_bubble clock_llist \
            check_heap check_bubble check_llist ; do \
```

```
  if [ -f $i ] ; then rm $i ; fi \
done
[motoki@x205a]$ ls
Makefile                bubblesort.c            consumed_time.c
Makefile.simple         check-sort-program.c    llistsort.c
btree-heapsort.c        clock-sort-5-10-etc.c
[motoki@x205a]$
```

分割コンパイルの方法 (まとめ):

- 1 つのプログラムのためにディレクトリを 1 つ作り、そこに
  ① そのプログラムを構成するファイル群と、
  ② 目的とする実行ファイルを作るためのコンパイル手順等を記述したファイル (**メイクファイル**という)、
  だけを入れる。
- 汎用のライブラリファイルや、それに付随するヘッダファイルは、個別のプログラム用ディレクトリとは別の共通の場所に置く。(次の 14.5 節を参照。)
- メイクファイルの暗黙の名前は `Makefile` または `makefile` である。 これらの名前のファイルがカレントディレクトリにあれば、単に

  `make` *target*

  とコマンド入力するだけで、目的ファイル *target* の最新版を確保するために `Makefile`(または `makefile`) の指示に従って必要最小限の箇所だけ (再) コンパイルを行ってくれる。
- メイクファイルには、次のような事柄が記述されている。
  ◇ プログラムの**構成要素間の依存関係**、すなわち、目標とする実行ファイルやオブジェクトファイルの各々がどのファイルの更新に影響をうけるか。
  ◇ 目標とする実行ファイルやオブジェクトファイルを生成するために、どんなコンパイル作業を行えばよいか。

14.5 ライブラリ

- ユーザ独自の関数ライブラリを作成して、標準ライブラリと同じ様に使うことが出来る。
- UNIX においては、ライブラリを生成し管理するユーティリティは**アーカイバ**と呼ばれている。コマンド名は `ar` である。
- UNIX においては、ライブラリファイルの拡張子は `.a` である。

例 14.7 (ライブラリファイルの中身を見る) ライブラリファイルの中身を見るためには、`t` というキーを付けて `ar` コマンドを実行する。例えば、標準ライブラリ関数のオブジェクトファイルが詰まった `/usr/lib/libc.a` の中身を見るには次の様にする。

```
[motoki@x205a]$ ar t /usr/lib/libc.a |more
```

```
init-first.o
libc-start.o
set-init.o
sysdep.o
version.o
errno-loc.o
   ......
```

(以下省略、全部で 1082 行)

例 14.8 (自分専用のライブラリファイルを作成する) 例題 14.3 で挙げた計算時間計測のためのモジュール consumed_time.o は汎用性が高いので、これを自分専用のライブラリファイルに入れておきたい。この場合は次の様にする。

```
[motoki@x205a]$ gcc -c consumed_time.c ..................（コンパイル）
[motoki@x205a]$ ar ruv mylib.a consumed_time.o ..........（追加登録）
a - consumed_time.o
[motoki@x205a]$ ranlib mylib.a ..... (generate index to archive)
[motoki@x205a]$ ls ..........................................（確認）
Makefile                bubblesort.c            consumed_time.o
Makefile.simple         check-sort-program.c    llistsort.c
Makefile.with-.o-file   clock-sort-5-10-etc.c   mylib.a
btree-heapsort.c        consumed_time.c
[motoki@x205a]$ ar t mylib.a ................................（確認）
consumed_time.o
[motoki@x205a]$
```

ここで、

- 1 行目は前処理とコンパイルだけ (リンクなし) を行ってオブジェクトファイル consumed_time.o を生成することを指示している。
- 2 行目の ar コマンドは、オブジェクトファイル consumed_time.o をライブラリファイル mylib.a に追加登録している。ライブラリファイル mylib.a がない場合は新たに生成される。

> **補足：**
> ar コマンドのキーの意味は次の通りです。
> - r … replace(置き換え)。ファイルをライブラリ内に挿入する。但し、同じ名前のものが既にあれば置き換える。
> - u … update(更新)。'r' キーと共に使うことが出来る。これが指定されていれば、同じ名前のものが既にライブラリ内にあった時、新しいものを古いもので置き換えるのは避けられる。
> - v … verbose(詳細報告)。
> - t … table(内容物の表; 索引)。ライブラリ内に蓄えられている内容物の表を表示。
> - s … 索引を作る。$\Longrightarrow$ 'ar s' は 'ranlib' と同じ。

- 6 行目の ranlib コマンドは、ライブラリの中に詰まっている .o ファイルの索引 (index) を作り、それもライブラリにに格納することを指示している。

> 補足：
> ライブラリの中に.o ファイルの索引が用意されていればリンク作業が速くなり、またライブラリ内のルーチンが互いに別のルーチンを呼び合うことが出来るようになります。
>
> 上の例の場合は、ライブラリに 1 つの.o ファイルしか入っていないので、実際には ranlib コマンドを使う必要はないかも知れない。

例 14.9 (自分専用のライブラリファイルを使う) ライブラリファイルは cc コマンドの右側に並べることが出来る。例えば、上の例 14.8 の様にライブラリファイルが構成され (カレントディレクトリ上に置かれ) ていた場合には、

```
gcc clock-sort-5-10-etc.c consumed_time.c btree-heapsort.c
```

というコンパイルは

```
gcc clock-sort-5-10-etc.c btree-heapsort.c mylib.a
```

という風に行うことも出来る。ライブラリファイルからは必要な .o ファイルだけが取り出され、最終的な実行ファイルにロードされる。

自分専用のライブラリをより汎用なものにするためには：

① 自分専用のライブラリファイルは、そのライブラリを参照するそれぞれのディレクトリに置くのではなく、ホームディレクトリ付近の然るべき場所 (例えば~/lib) に置く。

② メイクファイルの中では絶対パスを指定して自分専用のライブラリファイルを参照する。

③ 自分専用のライブラリ関数のプロトタイプ宣言を並べてインクルードファイルを構成し、ホームディレクトリ付近の然るべき場所 (例えば~/include) に置く。

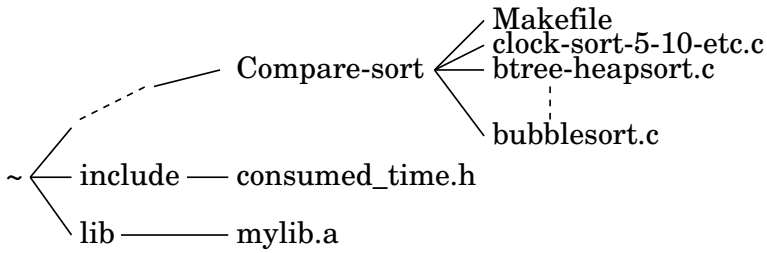
④ 自分専用のライブラリ関数を呼び出すプログラムにおいては、プログラムの始めの方で③のインクルードファイルを

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

という形で取り込む。

⑤ メイクファイルの中では、③で用意したディレクトリをインクルードファイルの探索場所に追加指定してコンパイルするようにする。

例 14.10 (ライブラリファイル等の配置) 例 14.8 において、モジュール consumed_time.o (例題 14.3) を自分専用のライブラリファイル mylib.a に入れる例を示した。このライブラリを汎用的なものとして使いたい場合は、例えば次の図の様にホームディレクトリの下に ~/include と ~/lib というディレクトリを用意して、~/lib の中にライブラリ mylib.a を、~/include の中にライブラリ関連のヘッダファイルを配置する。



ここで、ライブラリ ~/lib/mylib.a とヘッダファイル ~/include/consumed_time.h の中身は次の通りである。

```
[motoki@x205a]$ pwd
/home/motoki/C-Java2003/Programs-C/Compare-sort
[motoki@x205a]$ mkdir ~/lib
[motoki@x205a]$ cp mylib.a ~/lib
[motoki@x205a]$ ar t ~/lib/mylib.a
consumed_time.o
[motoki@x205a]$ mkdir ~/include
                 (ヘッダファイルを作成)
[motoki@x205a]$ nl ~/include/consumed_time.h
     1 typedef struct {
     2    double  process_time;
     3    double  real_time;
     4 } Second;

     5 void start_timekeeper(void);
     6         /*
     7          * 計算時間を測定する際の開始時点を (静的外部変数に)
     8          * 記録する。
     9          */

    10 Second consumed_time(void);
    11         /*
    12          * 以前に start_timekeeper() または consumed_time()
    13          * が呼ばれた時点から現在までに消費したプロセッサ
    14          * 時間 (秒) とカレンダー時間 (秒) の組み (構造体) を返す。
    15          */
[motoki@x205a]$
```

例 14.11 (make コマンドを用いた分割コンパイル; 自分専用のライブラリも利用する版) 例14.10のようにライブラリファイル ~/lib/mylib.a とインクルードファイル ~/include/consumed_time.h が用意されている場合には、例 14.6 で使用したプログラム等は次のように簡略化できる。

(1) Compare-sort 内に consumed_time.c (や mylib.a) を置く必要が無くなる。

(2) ソースファイル clock-sort-5-10-etc.c (例題 14.4) の中の consumed_time.c に関わる typedef と関数プロトタイプの部分

```
21 typedef struct {
22    double  process_time;
23    double  real_time;
24 } Second;

25 void   start_timekeeper(void);
26 Second consumed_time(void);
```

は

```
   #include "consumed_time.h"
```

という行で置き換えることが出来る。

(3) 例 14.6 の Makefile は例えば次の様に書き換えることが出来る。

```
[motoki@x205a]$ ls
Makefile                btree-heapsort.c        clock-sort-5-10-etc.c
Makefile.simple         bubblesort.c            consumed_time.c
Makefile.with-.o-file   check-sort-program.c    llistsort.c
[motoki@x205a]$ nl Makefile
 1 # Makefile for clocking or checking 3 sort programs:
 2 #     (1) Heapsort
 3 #     (2) Bubblesort
 4 #     (3) Sort by insertion over linked-list

 5 CC    = gcc
 6 BASE  = /home/motoki ............................. (追加された行)
 7 INCLS = -I${BASE}/include ........................ (追加された行)
 8 LIBS  = ${BASE}/lib/mylib.a ...................... (追加された行)

 9 OBJS_CLOCK_HEAP  = clock-sort-5-10-etc.o btree-heapsort.o
10 OBJS_CLOCK_BUBBL = clock-sort-5-10-etc.o bubblesort.o
11 OBJS_CLOCK_LLIST = clock-sort-5-10-etc.o llistsort.o
12 OBJS_CHECK_HEAP  = check-sort-program.o btree-heapsort.o
13 OBJS_CHECK_BUBBL = check-sort-program.o bubblesort.o
14 OBJS_CHECK_LLIST = check-sort-program.o llistsort.o

15 clock_heap: ${OBJS_CLOCK_HEAP}
16 [ tab ] ${CC} -o clock_heap ${INCLS} ${OBJS_CLOCK_HEAP} ${LIBS}
                                  (${INCLS} と ${LIBS} の項が追加された。)
17 clock_bubble: ${OBJS_CLOCK_BUBBL}
18 [ tab ] ${CC} -o clock_bubble ${INCLS} ${OBJS_CLOCK_BUBBL} ${LIBS}

19 clock_llist: ${OBJS_CLOCK_LLIST}
```

```
20 [tab] ${CC} -o clock_llist ${INCLS} ${OBJS_CLOCK_LLIST} ${LIBS}
```

以下省略 ⟶ 例 14.6 の 21~43 行目と同じ

```
[motoki@x205a]$
```

ここで、

- 6 行目 で定義したマクロ BASE はホームディレクトリの絶対パスを示すためのもので、この部分はユーザ毎に異なる。
- 7 行目 で定義したマクロ INCLS は 16 行目,18 行目,20 行目の gcc コマンドのオプションとして使われている。これによって、コンパイル時の暗黙のインクルードファイル探索場所として /home/motoki/include というディレクトリが追加される。
- 8 行目 で定義したマクロ LIBS は使用するライブラリファイルを絶対パスで指定したものである。これを 16 行目,18 行目,20 行目の cc コマンドの引数とすることにより、ライブラリ ~/lib/mylib.a から consumed_time.o を取り込める様になる。

このメイクファイルを使えば、例えば次のようにコンパイルが進む。

```
[motoki@x205a]$ make clock_llist
gcc -c clock-sort-5-10-etc.c
gcc -c llistsort.c
gcc -o clock_llist -I/home/motoki/include clock-sort-5-10-etc.o ll\
istsort.o /home/motoki/lib/mylib.a
[motoki@x205a]$
```

14.6 自習 その他の有用なツール

{ ケリー&ポール 11.18 節 }

touch コマンド： make コマンドには色々な使い方が可能である。例えば、オプションでメイクファイル内のマクロ定義を外部から修正指定して make を行うことも出来る。 ただ、このような場合、make には、ファイルの生成時間の新旧情報を基に再コンパイルを省略することなく、コンパイルを全てやり直してもらいたいことも起こる。

こういった場合のために、指定したファイルの作成日時を現在の時刻に設定する touch というコマンドが用意されている。

形式： touch [ファイル名]

演習問題

□演習 14.1 (実行プロファイル) これまで作成したプログラムの実行プロファイルを表示してみよ。

□演習 14.2 (クイックソートの平均計算時間) 例題 7.3 のプログラムを基に 2 つの外向けの関数

- sort_method ··· 整列化アルゴリズムの名前を答える関数、
- sort ··· 引数で指定された配列要素を quicksort アルゴリズムで昇順に並べ替える関数、

を含むモジュール quicksort.c を作れ。そして、このソートプログラムの平均計算時間を例題 14.4 に倣って計測してみよ。

□**演習 14.3 (2 分木の中間順走査によるソートの平均計算時間)** 2 つの外向けの関数

- sort_method ··· 整列化アルゴリズムの名前を答える関数、
- sort ··· 引数で指定された配列要素を例題 12.4 の様に常に

  左の子とその子孫のデータ値 $\leq$ 親のデータ値

  親のデータ値 $<$ 右の子とその子孫のデータ値

  となる様に 2 分木状に記録していき、出来た 2 分木を中間順に走査することにより、昇順に並べ替える関数、

を含むモジュール btree-traversesort.c を作れ。そして、このソートプログラムの平均計算時間を例題 14.4 に倣って計測してみよ。

□**演習 14.4 (makefile)** #include " □ " という形の include 行を含まない 3 つの C ソースファイル main.c, abc.c, pqr.c を合わせてコンパイル (&リンク) して、run という名前の実行ファイルを作りたい。この作業を分割コンパイルで行うための makefile を書け。

□**演習 14.5 (makefile)** 4 つの C ソースファイル aaa.c, bbb.c, ppp.c, xxx.c が用意されたディレクトリ内で、2 種類のコンパイル

```
gcc -o apx aaa.c ppp.c xxx.c -lm
gcc -o bpx bbb.c ppp.c xxx.c
```

を効率良く行うための Makefile を書け。[「効率良く...」ということだから、もちろん .o ファイルも生成する。]

□**演習 14.6 (makefile)** 4 つの C ソースファイル main1.c, main2.c, aaa.c, xxx.c が用意されたディレクトリ内で、2 種類のコンパイル

```
gcc -o exe1 main1.c aaa.c
gcc -o exe2 main2.c aaa.c xxx.c -lm
```

を効率良く行うための Makefile を書け。[「効率良く...」ということだから、もちろん .o ファイルも生成する。]

□**演習 14.7 (最も素朴な版のメイクファイル)** 演習 14.2 と演習 14.3 で作成したモジュール quicksort.c と btree-traversesort.c に対して、例 14.5 と同様のことを行ってみよ。

□**演習 14.8 (.o ファイルも作る版のメイクファイル)** 演習 14.2 と演習 14.3 で作成したモジュール quicksort.c と btree-traversesort.c に対して、例 14.6 と同様のことを行ってみよ。

□**演習 14.9 (ライブラリファイルの中身を見る)** 数学関数のオブジェクトファイルが詰まった /usr/lib/libm.a の中身を調べてみよ。

□**演習 14.10 (自分専用のライブラリファイルを作成)** 例 14.8 で説明されていることを実際に自分で行ってみて下さい。

□**演習 14.11 (自分専用のライブラリファイルを使う)** 例 14.9 で説明されていることを実際に自分で行ってみて下さい。

□**演習 14.12 (自分専用のライブラリも利用する版のメイクファイル)** 演習 14.2 と演習 14.3 で作成したモジュール `quicksort.c` と `btree-traversesort.c` に対して、例 14.11 と同様のことを行ってみよ。

15 自習 ファイル入出力と OS とのインタフェース

- 復習 ファイル入出力,fopen(),fclose(),
- ファイル記述子による入出力,
- C プログラム中からコマンド実行, system(),
- C プログラム中からパイプを使う,
- 環境変数へのアクセス, getenv(),
  main 関数の第 3 の引数 `env[]`

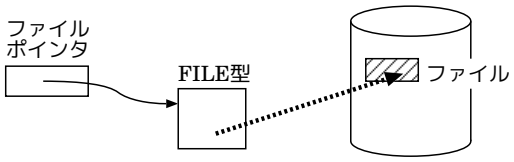
15.1 復習 ファイル入出力 — fopen() と fclose() —

{ ケリー&ポール 11.4~11.5 節 }

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、

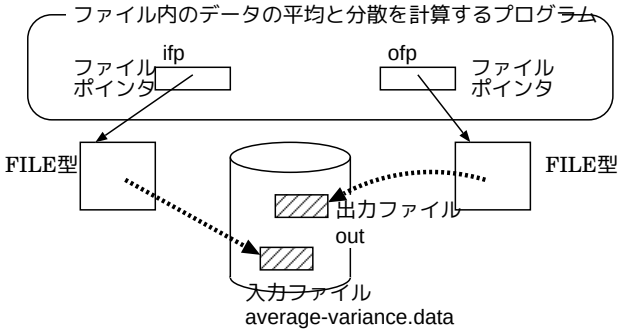
という仕組みが取られている。 この FILE 型構造体へのポインタのことを**ファイルポインタ**と言う。また、操作を始めたいファイルに関する FILE 型構造体を言語処理系/OS に作ってもらってファイル操作の準備を行うことを、**ファイルをオープンする**と言い、逆に操作の終ったファイルの FILE 型構造体の領域を解放してファイル操作の後始末を行うことを**ファイルをクローズする**と言う。

例題 15.1 (ファイル内のデータの平均と分散) 例題 1.4 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

$$\begin{aligned} \mu &= \frac{1}{50}(x_0 + x_1 + x_2 + \cdots + x_{49}) \\ V &= \frac{1}{50}\sum_{i=0}^{49}(x_i - \mu)^2 \end{aligned}$$

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 1.4 と同じで良いので、次の 4 点に注意して例題 1.4 のプログラムに少し手を加えるだけである。(⟹4.7 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()` ではなく `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()` ではなく `fprintf()` を用いる。

(プログラミング) 例題 1.4 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、

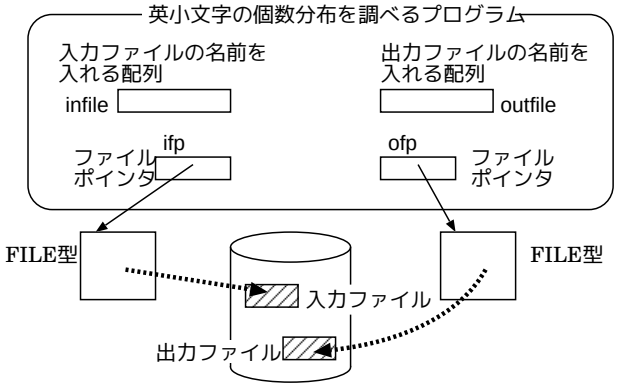
- プログラム 15 行目 は、ifp と ofp がファイルポインタであることを宣言している。

> **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。

例題 15.2 (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。(⟹ 4.7 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。(⟹ 4.7 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。(⟹ 4.7 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。(⟹ 4.7 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

(⟹ 5.1 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("%s", infile);
    13   printf("Type in a name of an output file: ");
    14   scanf("%s", 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:   5      e:  32
f:  47      g:   1      h:   4      i:  50      j:   0
k:   0      l:  20      m:   3      n:  32      o:  28
p:  23      q:   0      r:  13      s:  10      t:  32
u:  15      v:   0      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。
⟹ fgetc() の関数値は char ではなく int 型と決められている。
⟹ 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。)

ファイルについてのまとめ：

- C プログラムの中では (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。

補足：
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 `);`

15.2 ファイル記述子による入出力

A ケリー&ポール 11.8 節, 大倉&谷田部「UNIX システムコールハンドブック」(BNN,1995), カート・オール「例題で学ぶ Linux プログラミング」(ピアソン,2001)7 章, 山口 (監)「新 The UNIX Super Text 下」(技術評論社,2003)58.4.2 節

標準ライブラリ関数以外に、直接 OS に処理を依頼するための関数 (システムコールという) も用意されている。

例えば、
ファイル入出力のシステムコール, ファイル情報獲得のシステムコール, プロセス生成のシステムコール, プロセス間で通信するためのシステムコール, ... などがある。

特にファイル入出力関連では、オープン, クローズ, 読み込み, 書き込み という基本操作のためのシステムコールが用意されている。但し、

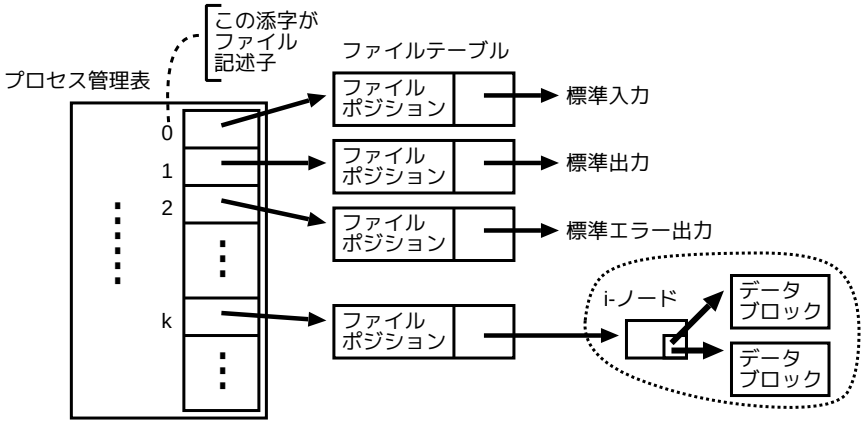
- システムコールの場合は、処理するファイルの指定はファイルポインタではなく、プロセスが開いたファイルに付けられるファイル記述子と呼ばれる識別番号によって行う。

ファイル記述子：
カーネルは、プロセスが開いたファイルの各々に対して
①次に読み書きする位置を示すファイルポジションと
②ファイルの実体を束ねる i-ノードへのポインタ
から成る、ファイルテーブルと呼ばれる構造体を構成する。 そして、プロセスが関与するファイルテーブルへのポインタの配列をプロセス毎に用意して管理する。これらの配列の添字はファイル記述子 (file descriptor) と呼ばれ、ファイル入出力に利用することが出来る。

◇ プロセス起動時には、各プロセスに対して標準入力, 標準出力, 標準エラー出力の 3 つのファイルがオープンして与えられる。各々のファイル記述子は次のように決められている。

ファイル	ファイル記述子	<unistd.h>で定義されたマクロ
標準入力	0	STDIN_FILENO
標準出力	1	STDOUT_FILENO
標準エラー出力	2	STDERR_FILENO

◇ 初期の UNIX(Version7) では、各プロセスは 0∼19 までのファイル記述子が使用できた。



- システムコールの場合は、入出力用のバッファ(i.e. まとめて入出力を行うための char 型配列) もプログラマが確保する必要がある。

補足：
これに対して、fopen() の返り値として得られるファイルポインタを通じてファイルを扱う場合には、大抵の場合、暗黙のうちにバッファリングが行われる。 [実際、FILE 型構造体はファイルの実体へと繋がっているファイル記述子の情報の他にバッファリングの情報も持つことが多い。]

補足の補足：
通常、ファイルを記録した物理媒体との入出力は、入出力時間の短縮を狙ってある大きさのブロック単位で行う。これをブロッキングと言う。

また、低速な入出力装置と高速な CPU との協調を図るため、バッファと呼ばれる記憶場所を用意して、プロセス ↔ バッファ 間のデータ転送は CPU が、バッファ↔ 入出力装置 間のデータ転送は入出力装置が行うようにする。これをバッファリングと言う。

ファイル記述子を用いた入出力の概要：

- ファイル入出力のシステムコールを使うためには、

```
#include <fcntl.h>
#include <unistd.h>
```

というヘッダファイル宣言が必要である。 また、入出力用のバッファ(i.e. まとめて入出力を行うための char 型配列) もプログラマが確保する必要もある。

- ファイルをオープンしてそのファイル記述子を受け取るには、次の様に書く。

int 型変数 = open(ファイル指定 , 使用モードの指定);

または

int 型変数 = open(ファイル指定 , 使用モードの指定 , パーミッション指定);

補足：

◇ open() の関数値はオープンしたファイルのファイル記述子である。但し、オープンに失敗すると −1 である。

◇ 引数 ファイル指定 は、オープンするファイルをフルパスで指定した文字列を表す。

◇ 指定したファイルが無く第 3 引数が指定されている場合は、ファイルは新たに生成される。

◇ 引数 使用モードの指定 はファイルの使い方を表したもので、次のマクロを OR 演算子 (|) で繋げて指定する。

O_RDONLY ··· 読み出し専用にオープンする。
O_WRONLY ··· 書き込み専用にオープンする。
O_RDWR ··· 読み書き両用にオープンする。
O_APPEND ··· 追加書き込み用にオープンする。
O_CREAT ··· ファイルが存在しない場合に作成する。これを指定した場合は第 3 引数で保護モードを指定する。
O_TRUNC ··· 書き込みでオープンし、長さを 0 に切り詰める。
O_EXCL ··· 作成しようとしているファイルが既に存在する場合、エラーを返す。(関数値は −1。)

◇ 引数 パーミッション指定 は、ファイルの保護モードを表す。例えば自分には読み書きを許し、同じ一グループのユーザと他人には読み出しだけ許す場合は、8 進表示で 0644 と指定すればよい。

- ファイル記述子を指定してファイルをクローズするには、次の様に書く。

close(ファイル記述子);

- ファイル記述子を指定してファイルからデータをバッファに読み込むためには、次の様に書く。

read(ファイル記述子 , バッファ名 , バッファの大きさ);

補足：

◇ read() が呼ばれると、引数の指定に従って既にオープンされたファイルの現在操作中の位置 (この位置を指すポインタも ファイルポインタ と呼ぶ) からデータを読み込む。但し、指定したバイト数のデータが無くなれば、そこで読み込みはおしまいである。読み込みに成功すれば読み込んだデータのバイト数が関数値として返され、ファイルポインタは読み込んだバイト数だけ移動する。 また、失敗すれば −1 が返される。[⟹ 関数値が 0 だとファイルを EOF まで読んだことになる。]

◇ 引数 ファイル記述子 は読み込み元のファイル記述子である。

◇ 引数 バッファ名 は読み込んだデータを保存するバッファのアドレスを表す。

◇ 引数 バッファの大きさ は読み込むデータのバイト数を表す。

- ファイル記述子を指定してバッファ内のデータをファイルに書き込むためには、次の様に書く。

```
write( ファイル記述子 , バッファ名 , データの大きさ );
```

補足：

◇ write() が呼ばれると、引数の指定に従ってファイルへの書き込みが為される。成功すると書き出されたバイト数が返され、ファイルポインタは書き出されたバイト数だけ移動する。失敗すると -1 が返される。

◇ 引数 ファイル記述子 は書き出し先のファイル記述子である。

◇ 引数 バッファ名 は書き出すデータを保存するバッファのアドレスを表す。

◇ 引数 データの大きさ は書き出すデータのバイト数を表す。

注意：

◇ read(),write() のために システムバッファが用意される場合 は、read(),write() はシステムバッファ上の仮想ファイルに対してだけ行われ磁気ディスクへの書き込みはしばらく後になる。
$\Longrightarrow$ ファイルデータ上の矛盾発生を避ける等のために即座に磁気ディスクに書き込みたい場合は sync() システムコールを使う。

◇ システムバッファが用意されない場合は、read(),write() の度に磁気ディスクにアクセスすることになるので、プログラムの実行速度がバッファサイズ バッファの大きさ によって大きく左右されることになる。

例題 15.3 (大文字 $\leftrightarrow$ 小文字の反転) コマンドラインの 1 番目の引数で指定したファイルの大文字と小文字を反転して、その結果を 2 番目の引数で指定したファイルに書き出すプログラムをファイル記述子を使って構成してみよ。

(考え方) ファイル記述子を使って入出力を行う訳だから、引数で指定された 2 つのファイルを open() システムコールを使ってオープンし、

① 入力用ファイルの先頭からある大きさのブロック単位で順に文字列を read() システムコールで読み出し、

② 読み込んだ各々の文字の大文字・小文字を islower(), toupper(), tolower() ライブラリ関数を使って反転し、

③ その結果を write() システムコールを使って出力用ファイルに書き出す、

ということを繰り返せば良い。入力用のファイルが無かったり、出力用のファイルが既に存在している場合はエラーとして強制終了させれば良い。その際、ライブラリ関数 perror() を用いれば、open() の際にどういうエラーがコンピュータ内部で起こったのかを表示できる。

(プログラミング) 入力ファイルから一度に読み込むブロックの最大サイズを 1024 (BUFSIZE) 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 file-descriptor-flip-lower-upper.c
     1 /* 大文字-小文字の反転 */
     2 #include <stdio.h>
     3 #include <ctype.h>
```

```
 4 #include <fcntl.h>
 5 #include <unistd.h>
 6 #include <stdlib.h>

 7 #define BUFSIZE  1024

 8 int main(int argc, char **argv)
 9 {
10   char buffer[BUFSIZE], message[100];
11   int  in_fd, out_fd, in_size, k;

12   if ((in_fd=open(argv[1], O_RDONLY))==-1) {
13     sprintf(message, "open(%s, O_RDONLY)", argv[1]);
14     perror(message);
15     printf("  ===> Execution was aborted.\n");
16     exit(EXIT_FAILURE);
17   }
18   if ((out_fd=open(argv[2], O_WRONLY|O_EXCL|O_CREAT, 0644))==-1) {
19     sprintf(message, "open(%s, O_WRONLY|O_EXCL|O_CREAT, 0644)", argv[2]);
20     perror(message);
21     printf("  ===> Execution was aborted.\n");
22     exit(EXIT_FAILURE);
23   }
24   while ((in_size=read(in_fd, buffer, BUFSIZE)) > 0) {
25     for (k=0; k<in_size; k++) {
26       if (islower(buffer[k]))
27         buffer[k]=toupper(buffer[k]);
28       else
29         buffer[k]=tolower(buffer[k]);
30     }
31     write(out_fd, buffer, in_size);
32   }
33   close(in_fd);
34   close(out_fd);
35   return 0;
36 }
```

```
[motoki@x205a]$ gcc file-descriptor-flip-lower-upper.c
[motoki@x205a]$ ./a.out abc out
open(abc, O_RDONLY): No such file or directory
  ===> Execution was aborted.
[motoki@x205a]$ ./a.out file-descriptor-flip-lower-upper.c out
open(out, O_WRONLY|O_EXCL|O_CREAT, 0644): File exists
  ===> Execution was aborted.
[motoki@x205a]$ rm out
```

```
rm: ‘out’ を削除しますか (yes/no)? y
[motoki@x205a]$ ./a.out file-descriptor-flip-lower-upper.c out
[motoki@x205a]$ ls -l {file-descriptor-flip-lower-upper.c,out}
-rw-rw-r--  1 motoki  motoki  966 Mar 21 16:01 file-descriptor-flip-lower-upper.c
-rw-r--r--  1 motoki  motoki  966 Mar 21 16:05 out
[motoki@x205a]$ cat out
/* 大文字-小文字の反転 */
#INCLUDE <STDIO.H>
#INCLUDE <CTYPE.H>
#INCLUDE <FCNTL.H>
#INCLUDE <UNISTD.H>
#INCLUDE <STDLIB.H>

#DEFINE bufsize  1024

MAIN(INT ARGC, CHAR **ARGV)
{
  CHAR BUFFER[bufsize], MESSAGE[100];
  INT  IN_FD, OUT_FD, IN_SIZE, K;

     ......(以下省略)......

[motoki@x205a]$
```

ここで、

- プログラム 12 行目,18 行目 の open( ) システムコールで、コマンド引数に指定されたファイルを開いて各々のファイル記述子を in_fd, out_fd に格納している。
- プログラム 14 行目,20 行目 の perror( ) ライブラリ関数は、それぞれ 12 行目,18 行目 で起こったエラーの原因を 13 行目,19 行目 で構成した文字列とともに stderr に出力する。
- プログラム 24 行目 の read( ) システムコールで、入力ファイルからデータを buffer 配列領域に読み込んでいる。
- プログラム 26 行目 の islower( ) は英小文字かどうかを判定する引数付きマクロである。 (ヘッダファイル<ctype.h> の中で定義されている。)
- プログラム 27 行目, 29 行目 の toupper( ), tolower( ) はそれぞれ引数の文字コードを英大文字, 英小文字のコードに変換する標準ライブラリ関数である。
- プログラム 31 行目 の write( ) システムコールで、加工済のデータを buffer 配列領域から出力ファイルに書き出している。
- ls -l {file-descriptor-flip-lower-upper.c,out} の実行結果を見てみると、確かにプログラム 18 行目 の open( ) で指定した保護モードの出力ファイルが出来ている。

15.3 C プログラムの中からのコマンド実行

{ ケリー&ポール 11.10 節 }

- C プログラムの中から OS のコマンドを実行するには、次の様に書く。

```
system(" コマンド ");
```

(#include <stdlib.h> の宣言も必要。)

例 15.4 (C プログラムの中から emacs を起動する)

```
char  command[MAXLENGTH], file_name[MAXLENGTH];
    ......
sprintf(command, "emacs %s &", file_name);
printf("メッセージ");
system(command);
    ......
```

15.4 C プログラムの中からのパイプの利用

{ケリー&ポール 11.11 節, 山口 (監)「The UNIX Super Text 下」(技術評論社,1992)57.3.3 節カート・オール「例題で学ぶ Linux プログラミング」(ピアソン,2001)15.1.3 節,}

- UNIX の場合は、C プログラムの中からパイプを使うためのライブラリ関数が用意されている。

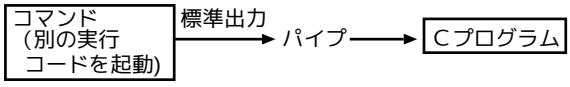
```
[motoki@x205a]$ cat /usr/include/stdio.h |grep popen
extern FILE *popen __P ((__const char *__command, __const char *__modes));
/* Close a stream opened by popen and return the status of its child.  */
[motoki@x205a]$ cat /usr/include/stdio.h |grep pclose
extern int pclose __P ((FILE *__stream));
[motoki@x205a]$
```

$\Longrightarrow$ ヘッダファイル <stdio.h> の宣言が必要。

- 実行コードを起動してその出力文字列を C プログラムの入力ストリームとして取り込むには、次の様に書く。

```
ファイルポインタ型変数 = popen(" コマンド ", "r");
```

これによって、指定されたコマンドが別プロセス上で起動されその標準出力からの出力文字列を読み出すためのファイルポインタが関数値として返される。 あとはこのファイルポインタを指定して通常の入力を行うだけである。

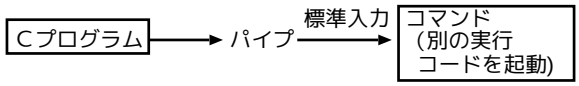


- 別の実行コードを起動して、C プログラムの出力文字列をそこへの入力ストリームとして送り込むには、次の様に書く。

```
(FILE*) 型変数 = popen(" コマンド ", "w");
```

これによって、指定されたコマンドが別プロセス上で起動されその標準入力へ文字列を書き込むためのファイルポインタが関数値として返される。 あとはこのファイル

ポインタを指定して通常の出力を行うだけである。



- popen() 関数によって出来たパイプを解除するには、次の様に書く。
  pclose(ファイルポインタ);

例 15.5 (ls コマンドの出力を全て大文字に変換して表示)

```
[motoki@x205a]$ nl file-popen-ls-to-toupper-Kelley.c
     1 /* popen("ls","r")でオープンしたストリームを大文字に変換して出力 */
     2 #include <stdio.h>
     3 #include <ctype.h>

     4 int main(void)
     5 {
     6   int  c;
     7   FILE *ifp;

     8   ifp = popen("ls", "r");
     9   while ((c=getc(ifp)) != EOF)
    10     putchar(toupper(c));
    11   pclose(ifp);
    12   return 0;
    13 }
[motoki@x205a]$ gcc file-popen-ls-to-toupper-Kelley.c
[motoki@x205a]$ ./a.out
COMPARE-SORT
REMARK-ON-PRINTF.C
REPORT-3.C
REPORT-3.LOG
REPORT-6
REPORT-6.TAR
A.C
A.OUT
BTREE-HEAPSORT.C
     (途中省略)
STRUCT-UNION-INT-OR-FLOAT.C
TYPEDEF-VECTOR-SPACE.C
[motoki@x205a]$
```

ここで、

- プログラム 10 行目 に現われる toupper(c) は、c が英小文字の場合に対応する大文字を返し、それ以外の場合には c をそのまま返す標準ライブラリ関数である。 (プロトタイプは <ctype.h> に入っている。)

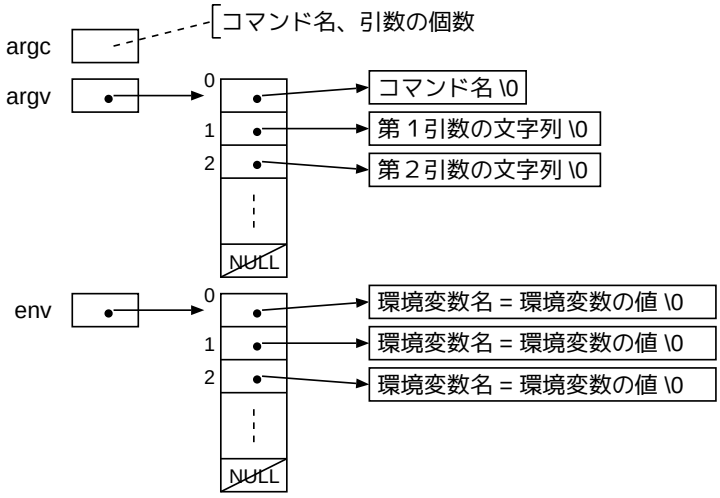
15.5 環境変数へのアクセス

{ ケリー&ポール 11.12 節 }

- main 関数の第 3 引数を指定すると、環境変数がどう設定されているかの情報を得ることが出来る。すなわち、

  ```
  main(int argc, char *argv[], char *env[])
  ```

  とすると、関数 main の起動直後には argc, argv, env は次の様に設定される。



- C プログラムの中から特定の環境変数の値を得たい時のために、次の標準ライブラリ関数が用意されている。

  ```
  getenv(" 環境変数名 ")
  ```

  $\Longrightarrow$ ヘッダファイルの宣言 `#include <stdlib.h>` が必要。

例 **15.6** (設定されている環境変数を全て表示)

```
[motoki@x205a]$ nl file-print-env-variables1-Kelley.c
     1 #include <stdio.h>

     2 int main(int argc, char *argv[], char *env[])
     3 {
     4   int i;

     5   for (i=0; env[i] != NULL; ++i)
     6     printf("%s\n", env[i]);
     7   return 0;
     8 }
[motoki@x205a]$ gcc file-print-env-variables1-Kelley.c
[motoki@x205a]$ ./a.out
LESSOPEN=|lesspipe.sh %s
USERNAME=
CANNA_SERVER=localhost
COLORTERM=gnome-terminal
HTTP_HOME=file:/usr/doc/HTML/index.html
```

```
     (途中省略)
LS_COLORS=no=00:fi=00:...(省略)...
[motoki@x205a]$
```

例 **15.7** (指定した環境変数の値を表示)

```
[motoki@x205a]$ nl file-print-env-variables2-Kelley.c
     1 #include <stdio.h>
     2 #include <stdlib.h>

     3 int main(void)
     4 {
     5   printf("   Host: %s\n"
     6          "   User: %s\n"
     7          "  Shell: %s\n",
     8          getenv("HOST"), getenv("USER"),getenv("SHELL"));
     9   return 0;
    10 }
[motoki@x205a]$ gcc file-print-env-variables2-Kelley.c
[motoki@x205a]$ ./a.out
   Host: ...(省略)...
   User: motoki
  Shell: /bin/bash
[motoki@x205a]$
```

演習問題

□**演習 15.1** (不定個の入力データの合計)　例題 3.6 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 15.2** (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
```

□**演習 15.3** (英文章中の指定した単語を大文字にする)　例題 9.4 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。

16 プリプロセッサ

- プリプロセッサ,
- #include の使い方,
- #define の使い方,
- 引数付きマクロの使い方,
- 自習 演算子#と##,
- 条件付きコンパイル,
- 既定義のマクロ,
- 自習 assert() マクロ, #error と#pragram, #line, 引数付きマクロと同じ名前の標準ライブラリ関数

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。

コンパイラの作業：

(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

プリプロセッサ (前処理を行う部分)：

- C プログラム中の # で始まる行は、どういう前処理を行うかの指示をしている。
[**プリプロセッサ指令**という。普通、1カラム目に # を置く。]

例：
```
#include <stdio.h>                    ···{/usr/include/stdio.h}
#include "ファイル名"          ··· { ファイル名 は普通 .h で終わる。}
#define PI 3.14159           ···{ マクロ名には普通英大文字を使う。}
```

- プリプロセッサ指令の効力は、その指令の場所からファイルの終わりまで有効 (但し、効力打ち消しの指令があればそこまで)。

16.1 #include の使い方

{ ケリー&ポール 8.1 節 }

#include で始まる行についてのまとめ (1.3.1 節)：

- `#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。

補足事項：

- `#include "  .h "` の形の指令の場合、カレントディレクトリに `  .h` という名前のファイルが無ければ、C 言語処理系毎の規則に従ってファイルの探索が行なわれる。(UNIX の場合は`/usr/include` 辺り。 `cc` コマンドの `-I` オプションをを使って指定することも出来る。)
- 複数のファイルにまたがる大きなプログラムを構築する場合、

```
#include <stdio.h>
        ⋮
```

マクロ定義の列
構造体定義の列
新しいデータ型定義の列
関数プロトタイプの列

  というヘッダファイルを作り、各ファイルの先頭にこのヘッダファイルを `include` する指令を入れておけば、同じ定義／宣言をあちこちのファイルの先頭で繰り返す手間が省ける。

16.2 #define の使い方

{ ケリー&ポール 8.2 節 }

例 16.1 (#define 行の使用例) マクロ定義を使った例としては、例題 4.1(三角関数の表), 例題 4.5(quicksort), 例題 8.1(Napier 数 e の 1000 桁計算) を挙げることが出来る。

```
 6  #define  PI  (3.1415926535897932)    /* 円周率 */
                                    ···(例題 4.1, 三角関数の表)
 9  #define  SIZE   100                  ···(例題 4.5,quicksort)
10  #define  WIDTH  10
11  #define  TRUE    1
13  #define   LIMIT   7          ···(例題 8.1,e の 1000 桁計算)
```

#define で始まる行についてのまとめ (1.3.1 節)：

- **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。

#define 行の 2 つの形式：

- `#define` [マクロ名] [文字列]
- `#define` [マクロ名]`(` [仮引数名]`, ..., ` [仮引数名]`)` [文字列]

  ↑ 間に空白なし

補足事項：

- 行の終りにバックスラッシュ(\) を置けば次の行に続けることが出来る。
- マクロ定義をうまく使えば、プログラムの明解さ、可搬性が向上する。

  ◇ 特殊な定数,
  ◇ プログラムのパラメータ
  } ⟶ 引数なしのマクロ定義

マクロ定義の解消： 次の様に書く。

```
#undef  マクロ名
```

16.3 引数付きマクロの使い方

例 16.2 (引数付きマクロの使用例) 引数付きマクロ定義を使った例としては、例題 8.2(○月○日のカレンダー), 例題 11.3(N 次元ベクトル空間の世界) を挙げることが出来る。

```
4  #define  DUMMY     0                    ···(例題 8.2, ○年○月のカレンダー)
5  #define  min(A,B)  ((A) < (B) ? (A) : (B))
```

```
1  #define  N  3                    ···(例題 11.3,N 次元ベクトル空間の世界)
2  #define  Print(title, vector)                \
3             printf("%s\n", title);            \
4             printf("   (%7.3f ", vector[0]);  \
5             for (i=1; i<N; ++i)               \
6               printf("%7.3f ", vector[i]);    \
7             printf(")\n")
```

引数付きマクロは、簡単な式だけでなく、繰り返しや条件分岐等も含む作業手順を 1 つにまとめ上げるのに有用である。

(マクロ呼び出しの部分は前処理時に定義に従って展開されるので、関数の場合と比べてオブジェクトコードが幾分大きくなるが、関数パラメータの引渡しは不要である。)

形式：

#define マクロ名 (仮引数名, ..., 仮引数名) 文字列

機能：

- 以降に現われる マクロ名 (仮引数名, ..., 仮引数名) というパターンを全て 文字列 に置き換える。その際、 文字列 の中に現われる引数名は, マクロ呼出し時の対応する実引数でそれぞれ置き換える。
  $\Longrightarrow$ 関数の代わりに使えば、計算効率が良い。
- 文字列置き換え後の構文が正しいかどうかのチェックはしない。
- 文字列 中に現われる仮引数名 の前に # が (1 個だけ) 付いていると、対応する実引数を 2 重引用符で囲んだものに置き換わる。
- 文字列 中に ## があると、仮引数が対応する実引数に置き換えられた後で、## とその両側の空白が除去される。

例 16.3 (良い例)

- `#define  SQ(x)  ((x)*(x))`
- `#define  Min(x,y)  (((x) < (y)) ? (x) : (y))`

例 16.4 (失敗例)

- `#define  SQ(x)  (x*x)`
  (理由: `SQ(a+b)` が `(a+b*a+b)` に展開されてしまう。)
- `#define  SQ(x)  (x)*(x)`
  (理由: `4/SQ(2)` が `4/(2)*(2)` に展開されてしまう。)
- `#define␣␣SQ␣(x)␣␣((x)*(x))`
  (理由: `SQ(7)` が `(x)␣␣((x)*(x))(7)` に展開されてしまう。)
- `#define  SQ(x)  ((x)*(x));`
  (理由: `SQ(2)+1` が `((2)*(2));+1` に展開されてしまう。)

16.4 自習 演算子#と##

例 16.5 (#演算子) デバッグ用に

```
#define  Dprint(expr)  printf(#expr " = %g\n", expr)
```

と定義すれば、例えば、

```
Dprint(x/y);
```

は次の様に置き換わる。

```
printf("x/y" " = %g\n", x/y)
```

すなわち

```
printf("x/y = %g\n", x/y)
```

例 16.6 (##演算子)

```
#define  X(i)  x ## i
```

と定義すれば、例えば、

```
X(1)=X(2)=X(3);
```

は次の様に置き換わる。

```
x1=x2=x3;
```

16.5 条件付きコンパイル

{ ケリー&ポール 8.7 節 }

目的：

- プログラム開発を容易にする (例えば、デバッグ用のコードを使うかどうかの切り替えを容易に行なう) ため。
- 容易に移植できるコードを書くため。

使い方：

- プログラムの中の

  ```
  #if 定数式
      ⋮
  #endif
  ```

  の部分は 定数式 の値がゼロ以外 (真) の時だけコンパイルされる。
- プログラムの中の

  ```
  #ifdef マクロ名
      ⋮
  #endif
  ```

  または

  ```
  #if defined( マクロ名 )
      ⋮
  #endif
  ```

  の部分は #ifdef の場所で マクロ名 の値が定義されている時だけコンパイルされる。
- プログラムの中の

  ```
  #ifndef マクロ名
      ⋮
  #endif
  ```

  または

  ```
  #if !defined( マクロ名 )
      ⋮
  #endif
  ```

の部分は #ifndef の場所で マクロ名 の値が定義されていない時だけコンパイルされる。

例 16.7 (デバッグのためのコードをコンパイルするかどうか、の切り替えを簡単に行なう)
変数 a の途中の値を観察することがプログラムの動作をチェックする上で有用な場合、a の値を覗いてみたい場所に

```
#if DEBUG
  printf("debug: a = %d\n", a);
#endif
```

というコードを埋め込んでおけば、プログラムの先頭で

```
#define DEBUG 1
```

とするか

```
#define DEBUG 0
```

とするかの切り替えだけで、変数 a の途中の値を観察するかどうかの切り替えを行なうことが出来る。

その他：

- if-else 文に似た制御構造も用意されている。

```
#if 定数式          /* #ifdef や #ifndef も可 */
  .........
#elif 定数式
  .........
#elif 定数式
  .........
  .........
#else
  .........
#endif
```

16.6 既定義のマクロ

{ケリー&ポール 8.8 節}

既定義マクロ名	値
__FILE__	ソースファイルのファイル名から成る文字列。
__LINE__	現在処理中の行番号を表す整数。
__DATE__	前処理時の日付を表す文字列。
__TIME__	前処理時の時間を表す文字列。
__STDC__	ANSI C コンパイラであれば、ゼロでない整数。

16.7 自習 assert() マクロ

{ケリー&ポール 8.10 節}

assert() マクロ：

- 標準ヘッダファイル <assert.h> の中で定義されている引数付きマクロ。
- プログラムの中の assert(式); というマクロ呼出しは
  式 が偽 (0) ならメッセージを出して強制終了する
  というコードに置き換えられる。

参考のため、
assert() マクロの定義の様子を次に示す。

```
[motoki@x205a]$ cat /usr/include/assert.h

        (中略)

#ifdef  NDEBUG

# define assert(expr)           ((void) 0)

        (中略)

#else /* Not NDEBUG.  */

__BEGIN_DECLS

/* This prints an "Assertion failed" message and aborts.  */
extern void __assert_fail __P ((__const char *__assertion,
                                __const char *__file,
                                unsigned int __line,
                                __const char *__function))
     __attribute__ ((__noreturn__));

/* Likewise, but prints the error text for ERRNUM.  */
extern void __assert_perror_fail __P ((int __errnum,
                                       __const char *__file,
                                       unsigned int __line,
                                       __const char *__function))
     __attribute__ ((__noreturn__));

__END_DECLS

# define assert(expr)                                                  \
  ((void) ((expr) ? 0 :                                               \
           (__assert_fail (__STRING(expr),                            \
                           __FILE__, __LINE__, __ASSERT_FUNCTION), 0)))

        (中略)

#endif /* NDEBUG.  */
```

16.8 自習 #error と #pragma, #line

{ ケリー&ポール 8.11~8.12 節 }

#error 指令：

- 形式は

  #error エラーメッセージ

- (前処理時に) #error に出会うとコンパイル時にエラーと判断し、#error に続く文字列を画面に出力させるための指令。
- この指令が実行されるとコンパイル処理は中断され、実行コードは生成されない。

例 16.8 (#error 指令)

```
[motoki@x205a]$ nl preprocessor-error-directive.c
     1 #include <stdio.h>

     2 int main(void)
     3 {
     4   printf("ファイル \"%s\" のコンパイル開始 (date: %s, time: %s)\n",
     5          __FILE__, __DATE__, __TIME__);

     6 #error ---------Check---------

     7   printf("%d\n", 777);    /* この行はコンパイルされない */
     8
     9   return 0;
    10 }
[motoki@x205a]$ gcc preprocessor-error-directive.c
preprocessor-error-directive.c:8: #error ---------Check---------
[motoki@x205a]$ ./a.out
bash: ./a.out: そのようなファイルやディレクトリはありません
[motoki@x205a]$
```

#line 指令：

- 形式は

  #line 行番号 " ファイル名 "

- コンパイル中に C コンパイラが保持する

  { 行番号 と ...(マクロ __LINE__ で表される)
  { ソースファイル名 ...(マクロ __FILE__ で表される)

  の情報を強制的に変更するための指令。
- 行番号 は次の行に設定する行番号を表す。

例 16.9 (#line 指令)

```
[motoki@x205a]$ nl preprocessor-line-directive.c
```

```
 1 #include <stdio.h>

 2 int main(void)
 3 {
 4   printf("ファイル \"%s\" の %d 行目をコンパイル中\n"
 5  "    (date: %s, time: %s)\n",
 6  __FILE__, __LINE__, __DATE__, __TIME__);

 7 #line 100 "myprog.c"

 8   printf("ファイル \"%s\" の %d 行目をコンパイル中\n"
 9  "    (date: %s, time: %s)\n",
10  __FILE__, __LINE__, __DATE__, __TIME__);
11
12   return 0;
13 }
[motoki@x205a]$ gcc preprocessor-line-directive.c
[motoki@x205a]$ ./a.out
ファイル "preprocessor-line-directive.c" の 7 行目をコンパイル中
    (date: Mar 14 2003, time: 17:06:06)
ファイル "myprog.c" の 103 行目をコンパイル中
    (date: Mar 14 2003, time: 17:06:06)
[motoki@x205a]$
```

#pragma 指令：

- 形式は

  #pragma 命令

- 個々のコンパイラに応じた処理をするための指令。
- コンパイラが認識できない#pragma 指令は無視される。

例 16.10 (#pragma 指令) OpenMP はコンパイルオプションを付けるだけで簡単に並列処理を実現するツールで、gcc だとバージョン 4.2 以降で利用可能である。この OpenMP では、プログラムの然るべき場所に#pragma 指令を埋めこむことによって並列処理の指示を行う。

```
[motoki@x205a]$ nl preprocessor-openMP.c
 1 // #pragma 指令の使用例 (OpenMP)
 2 #include <omp.h>
 3 #include <stdio.h>

 4 int main(void)
 5 {
 6   int i, a[10]=0;
```

```
 7   printf(" a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]\n");
 8   #pragma omp parallel
 9   {
10     #pragma omp for
11     for (i=0; i<10; i++)
12     {
13       a[i] = 80 + i;
14       printf("%4d %4d %4d %4d %4d %4d %4d %4d %4d %4d"
15              " ...(i=%2d, thread_num=%d)\n",
16              a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7], a[8], a[9],
17              i, omp_get_thread_num());
18       fflush(stdout);   //起こった順に正確に表示するため
19     }
20   }
21   return 0;
22 }
[motoki@x205a]$ gcc -fopenmp preprocessor-openMP.c
[motoki@x205a]$ ./a.out
 a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]
  80    0    0    0    0   85    0    0    0    0 ...(i= 5, thread_num=1)
  80    0    0    0    0    0    0    0    0    0 ...(i= 0, thread_num=0)
  80   81    0    0    0   85    0    0    0    0 ...(i= 1, thread_num=0)
  80   81   82    0    0   85    0    0    0    0 ...(i= 2, thread_num=0)
  80   81   82   83    0   85    0    0    0    0 ...(i= 3, thread_num=0)
  80   81   82   83   84   85    0    0    0    0 ...(i= 4, thread_num=0)
  80   81   82   83   84   85   86    0    0    0 ...(i= 6, thread_num=1)
  80   81   82   83   84   85   86   87    0    0 ...(i= 7, thread_num=1)
  80   81   82   83   84   85   86   87   88    0 ...(i= 8, thread_num=1)
  80   81   82   83   84   85   86   87   88   89 ...(i= 9, thread_num=1)
[motoki@x205a]$
```

16.9 自習 引数付きマクロと同じ名前の標準ライブラリ関数

{ ケリー&ポール 8.13 節 }

- 標準ライブラリ関数と同じ名前の引数付きマクロが用意されていることもある。
- 標準ライブラリ関数の方を呼び出したければ、関数名を丸括弧で囲んで書く。例えば、

```
(isalpha)(c)
```

演習問題

□演習 16.1 (引数付きマクロ, 関数) 平面上の 2 つの点の座標 $(a_x, a_y), (b_x, b_y)$ を読み込み、それらの点の間の距離 $\sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$ を計算して出力する C プログラムを作ろうとしたが、コンパイル時に次の様にエラーメッセージが出てしまった。

```
sv01_44: cat -n test0608_2.c
     1  #include <stdio.h>
     2  #include <math.h>
     3
     4  #define sq(x) ((x)*(x));
     5
     6  typedef struct {
     7      double x;
     8      double y;
     9  } Point;
    10
    11  int main(void)
    12  {
    13    Point a, b;
    14
    15    scanf("%lf%lf", &a.x, &a.y);
    16    scanf("%lf%lf", &b.x, &b.y);
    17    printf("distance between (%12.5g, %12.5g) and (%12.5g, %12.5g)\n"
    18           "  = %12.5g\n",
    19           a.x, a.y, b.x, b.y, distance(a,b));
    20    return 0;
    21  }
    22
    23  double distance(Point a, Point b)
    24  {
    25    return sqrt(sq(a.x-b.x)+sq(a.y-b.y));
    26  }
sv01_45: gcc test0608_2.c
test0608_2.c:23: warning: type mismatch with previous implicit declaration
test0608_2.c:19: warning: previous implicit declaration of ‘distance’
test0608_2.c:23: warning: ‘distance’ was previously implicitly declared
                                                              to return ‘int’
test0608_2.c: In function ‘distance’:
test0608_2.c:24: parse error before ‘;’
test0608_2.c:24: parse error before ‘)’
sv01_46:
```

このプログラムあるいはコマンド実行の中の、誤り箇所 (3 箇所) を指摘し、それぞれ修正せよ。(上の会話記録に直接書き込んで修正して下さい。)

> **Hint：** 誤り箇所 3 箇所の内 2 箇所はコンパイル時のエラーメッセージに反映されていますが、残りの 1 箇所は反映されていません。

□演習 16.2 (引数付きマクロ) $x = 1, 2, 3, 4, 5$ に対して $1/x^3$ を計算しその結果を表の形に見易く出力する C プログラムを作ろうとしたが、次の様に明らかに間違った結果が出

力されてしまった。 何が起こったのか説明し、プログラムの誤りを修正せよ。(プログラム等に直接書き込んで修正して下さい。)

```
sv01_49: cat test0508_2b.c
#include <stdio.h>

#define  cube(x)  x*x*x

int main(void)
{
  int    i;
  double x;

  printf("  x     1/cube(x)\n"
         "-----  -----------\n");
  for (i=1; i<=5; ++i) {
    x = (double)i;
    printf("%4.1f    %g\n",
           x, 1/cube(x));
  }
  return 0;
}
```

(右上へ続く ↗)

(↗ 左下からの続き。)

```
sv01_50: gcc test0508_2b.c
sv01_51: ./a.out
  x     1/cube(x)
-----  -----------
 1.0    1
 2.0    2
 3.0    3
 4.0    4
 5.0    5
sv01_52:
```

索 引

記　号

A

さ 行

た 行

ま 行

や 行

ら 行

わ 行