14 UNIXプログラミング環境

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。
- .....

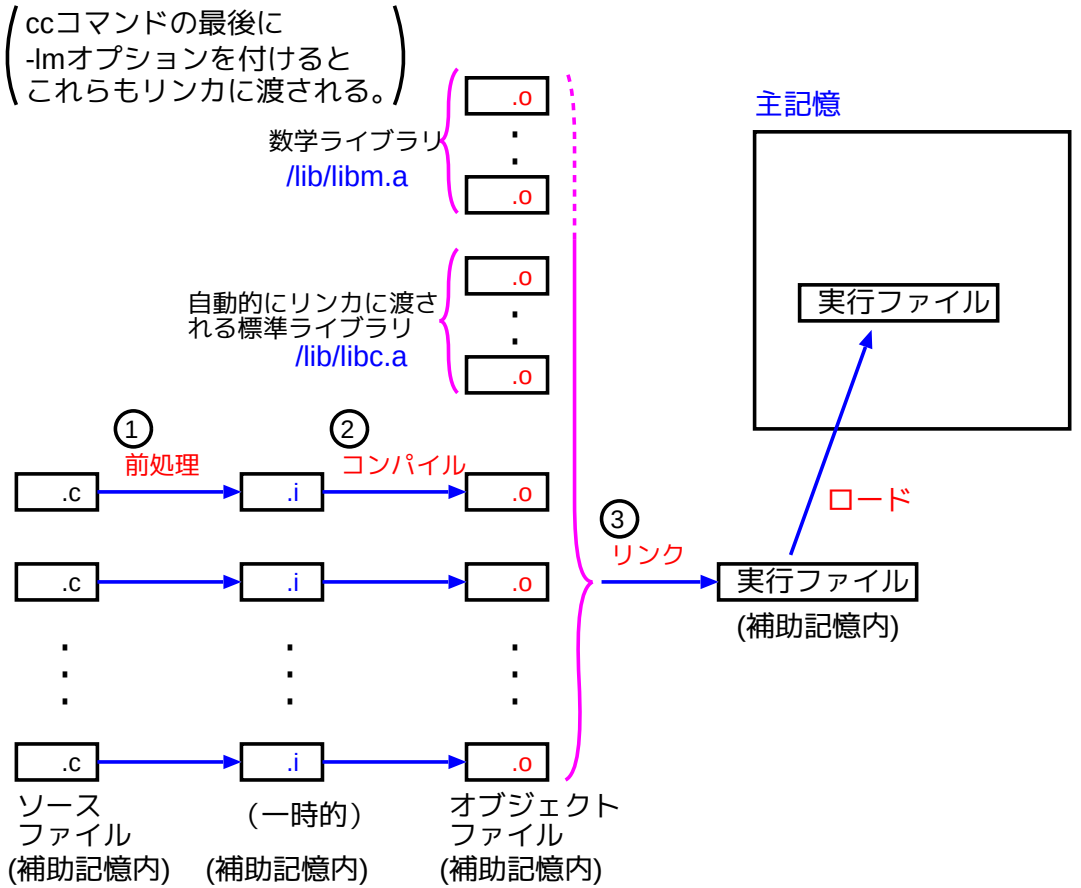
14–1 Cコンパイラ

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：
  各々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。
  ⇨ 大きなプログラムの場合に特に有効。

- `make`コマンドを使用しないと煩わしい。
  ⇨ 14.4節

cc コマンドの簡単な使用案内

- -o オプション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 オプションを使えばプリプロセッサとコンパイラ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、file 1.o, ..., file m.o という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
  ```

  ⇨ リンカだけを使いたければ .o ファイル群を cc コマンドに与える。

gccコマンドのオプション：

-c	コンパイルのみ。 .oファイルを生成する。
-g	デガッガ用のコードを生成する。
-o name	実行コードをファイルnameに入れる。
-p	プロファイラ prof 用の実行コードを生成する。
-pg	プロファイラ gprof 用の実行コードを生成する。
-D name=def	各.cファイルの先頭に次の行が挿入されていると仮定した
	#define name def
-E	プリプロセッサだけを起動する。
-I dir	ディレクトリdirの中にインクルードファイルを探す。
-M	make用の依存関係リストを出力する。コンパイルはしない
-O	最適化を行う。
-S	.sファイルにアセンブリコードを生成する。

14–2 自習 プロファイラを使う

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。

- 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 -xp
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 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13
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 今
度はgprofを使って表示してみよう。

補足：
この程度の問題だと、実行が速過ぎて計算時間が
0 秒になってしまう。

```
[motoki@applsv1_10]  gcc -pg check-sort-program.c btree-heapsc

[motoki@applsv1_11]  ./a.out
Input a random seed (0 - 2147483647):   333

before sorting:
    556     289     435     368     666     319     214     273      13
     64     943     869     956      50     298     112     218
    603     936     515     385     671     776     137     886
```

(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_p
  0.00      0.00     0.00        1     0.00     0.00  set_an_a
  0.00      0.00     0.00        1     0.00     0.00  sort
  0.00      0.00     0.00        1     0.00     0.00  sort_met


Call graph
```

```
granularity: each sample hit covers 2 byte(s) no time propagat

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_ranc
-----------------------------------------------
                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]
   [2] pretty_print            [4] sort
[motoki@applsv1_13]
```

14–3 Cコードを計時するには

- 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。）

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 $0, 1, 2, 3, \ldots$ 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。
$\Longrightarrow$ 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 $10000/\mathrm{CLOCKS\_PER\_SEC} = 0.01$ 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time() は、実際には
> 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）　例題13.2で作成したヒープソートプログラムの計算時間は次の様に測ることが出来る。

［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。］

```
[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..
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 th
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
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...*/
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
```

```
    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 elem
    (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 elem
    (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、

- 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 el
    (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  heapif
```

```
  8.00        0.24      0.02     30000       0.67       0.67  set_an
  4.00        0.25      0.01     30000       0.33       7.67  sort
  0.00        0.25      0.00         1       0.00       0.00  sort_m


Call graph


granularity: each sample hit covers 4 byte(s) for 4.00% of 0

index % time    self  children    called     name
                                                 <spontaneou
[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_arra
                0.00    0.00       1/1           sort_method
-----------------------------------------------
```

```
                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_ra
-----------------------------------------------
                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。

まとめ：2つの時間量

- 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時間量を表す。

  これは、各々の計算機で独自に設定されており、この時間量の単位として clock_t というデータ型を考える。 実習室では、 <time.h> の中で

  ```
  typedef long    clock_t;
  ```

  と定義されている。

- ログインやファイル作成の時刻（暦上の日付、時刻）を記録するために、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ている。

  この時間量の単位として time_t というデータ型を考える。 実習室では、 <time.h> の中で

  ```
  typedef long    time_t;
  ```

  と定義されている。

例題14．3（時間計測のためのモジュール）　例題 **14.3** 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
     8  /*              は consumed_time が呼ばれた時点から現在ま..
```

```
 9  /*              消費したプロセッサ時間(秒)とカレンダー時間..
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 ==>...
18                                  /*       外からは見え..
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..
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
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 $\mathbf{5, 10, 25, 50, 100, 200}$ 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。

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？

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。

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。

実際の時間計測には例題14.4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.
 6 /*    ぞれ 800000回, 400000回, 160000回, 80000回, 40000.
 7 /*    20000回 次の作業を繰り返して所用時間を計り、後で1回..
 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 pr
40          "that sorts 5, 10, 25, 50, 100, or 200 element
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
51      }                                       /* を参考のために測る
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
```

```
60        sort(a, prob[k].size);                    /* を測る。
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 ini
70           "size    process_t  real_time    process_t  re
71           "         (m sec)    (m sec)      (m sec)    (
72           "----    ---------  ---------    ---------  --
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...*/
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
    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14.4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
```

size	** time for sort ** process_t (m sec)	real_time (m sec)	**time for initialize** process_t (m sec)	real_time (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。

```
[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...
15 /*                      を(文字列として)入れて返す。配...
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。

```
[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...
24 /*                      を(文字列として)入れて返す。配...
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...
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...*
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..
63 /*             *ptr_ptr がその線形リストの先頭を指し示す..
```

```
64 /*                  る。この仮定の下で、整数 num を新しい項目...
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]$
```

□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14.5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14.5に倣って計測してみよ。

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14.5(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。

```
check-sort-program.c
btree-heapsort.c          ... (例題13.2で定義)

bubblesort.c
llistsort.c               ... (例題14.5で定義)
clock-sort-5-10-etc.c

consumed_time.c           ... (例題14.4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.....

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.....

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**14.5(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                    clock-sort-5-10-etc.c
btree-heapsort.c            consumed_time.c
bubblesort.c                llistsort.c
check-sort-program.c
[motoki@x205a]$  nl Makefile
  1 # Makefile for clocking or checking 3 sort programs:
  2 #      (1) Heapsort                                    注釈
  3 #      (2) Bubblesort
  4 #      (3) Sort by insertion over linked-list

  5 CC     = gcc     マクロ定義

  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}
```

規則　依存関係行

コマンド行

```
 18 check_heap: ${SRCS_CHECK_HEAP}
 19 ^I${CC} -o check_heap ${SRCS_CHECK_HEAP}

 20 check_bubble: ${SRCS_CHECK_BUBBL}
 21 ^I${CC} -o check_bubble ${SRCS_CHECK_BUBBL}

 22 check_llist: ${SRCS_CHECK_LLIST}
 23 ^I${CC} -o check_llist ${SRCS_CHECK_LLIST}
[motoki@x205a]$
```

(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 bubb
```

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、...

---

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。

```
[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
[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14.6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]$
```

bashスクリプト

シェル変数

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     13
     64     943     869     956      50     298     112     218
                (途中省略)
    985     782     857     881     252     236     706     945     73
```

```
after sorting (Sorting by Insertion in linked list):
      4       5       6      18      27      50      61      64     10
    110     112     132     137     144     153     181     204     20
                (途中省略)
    917     936     943     945     950     951     956     957     98
[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
assert.o
assert-perr.o
ctype.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14.4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。

```
[motoki@x205a]$  gcc -c consumed_time.c
[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                    clock-sort-5-10-etc.c
Makefile.simple             consumed_time.c
Makefile.with-.o-file       consumed_time.o
btree-heapsort.c            llistsort.c
bubblesort.c                mylib.a
check-sort-program.c
[motoki@x205a]$  ar t mylib.a  ...(確認)
consumed_time.o
[motoki@x205a]$
---
```

補足：　ar コマンドのキーの意味は

- r ··· replace(置き換え)
- u ··· update(更新)
- v ··· verbose(詳細報告)

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14.9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

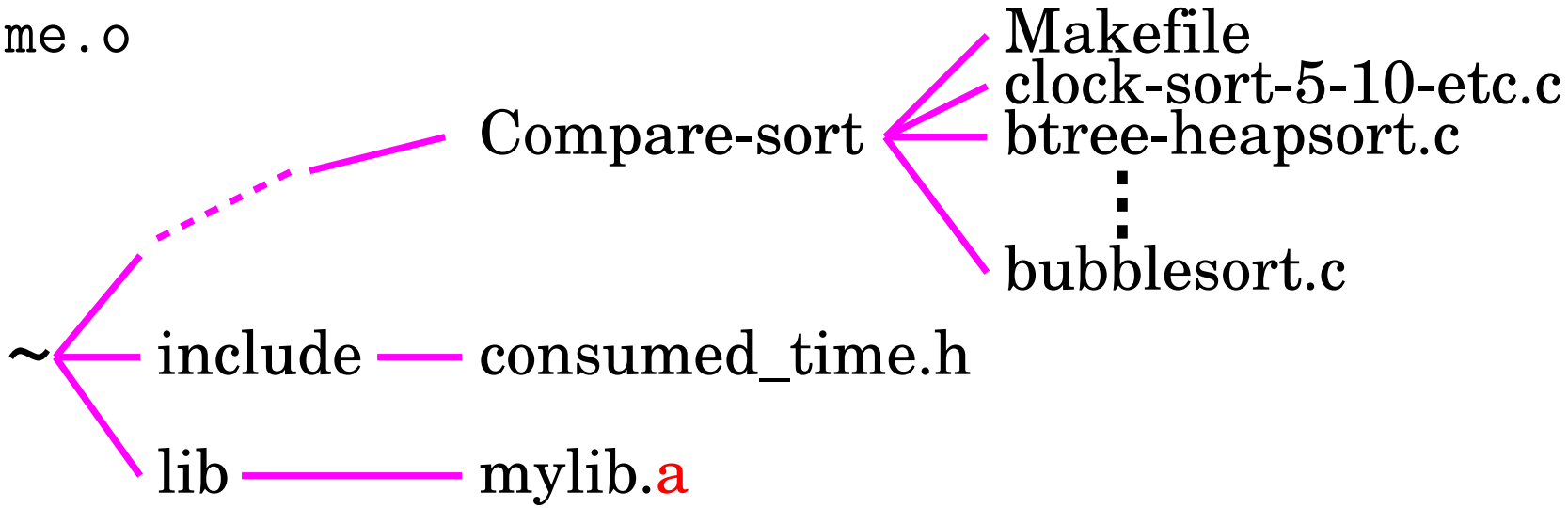
```
#include " .h "
```

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14.9において、モジュール consumed_time.o (例題14.4) 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、例えば次の様なディレクトリ構成にする。

```
[motoki@x205a]$  ls ~/include
consumed_time.h
[motoki@x205a]$  ls ~/lib
mylib.a
[motoki@x205a]$  ar t ~/lib/mylib.a
consumed_time.o
```



```
[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

例14.11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14.7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14.5)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14.7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-
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}
17 clock_bubble: ${OBJS_CLOCK_BUBBL}
18 [tab] ${CC} -o clock_bubble ${INCLS}
                              ${OBJS_CLOCK_BUBBL} ${LIBS}

19 clock_llist: ${OBJS_CLOCK_LLIST}
```

${INCLS} と ${LIBS} の項が追加された。

```
20 [tab] ${CC} -o clock_llist ${INCLS}
                              ${OBJS_CLOCK_LLIST} ${LIBS}
```

以下省略 $\longrightarrow$ 例14.7の21~43行目と同じ

```
[motoki@x205a]$
```

このメイクファイルを使えば、例えば次のようになる。

```
[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 llistsort.o /home/motoki/lib/mylib.a
[motoki@x205a]$
```

14–6 自習 その他の有用なツール

touchコマンド：

makeコマンドには色々な使い方が可能である。

ただ、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
