

#15
C 言語プログラムの実践的開発技法
2022 年度 / プログラミング及び実習 III

角川裕次

龍谷大学 先端理工学部

もくじ

1 (独自) main 関数の活用

- コマンドライン引数 argc と argv
- コマンドライン引数の解析例 1 : cla1.c
- fcpy.c : ファイルのコピー (教科書 List 13-7 の改造)
- プログラムの終了コード

2 (独自) 複数のファイルによるプログラム構成法

- ライブラリの利用
- 分割コンパイル
- 分割コンパイルをやってみる

3 (独自) パフォーマンス測定と改善法

- 実行時間の測定
- プロファイラによるボトルネックの発見

今回 (#15) の内容：シラバスでの該当部分

小テーマ：C 言語プログラムの実践的開発技法

第 25 回：main の引数

第 26 回：ライブラリとリンク

第 27 回：分割コンパイル

第 28 回：プログラムの終了コード

第 29 回：実行時間の測定

第 30 回：まとめ

重要概念リスト

- コマンドライン引数 argc & argv
- 終了コード : exit 関数の引数, main 関数の戻り値
- ライブラリのリンク
- 分割コンパイルと PIC ファイル
- time コマンドによる実行時間の測定
- gprof コマンドによる詳細な実行時間の分析

今回の実習・課題 (manaba へ提出)

実習内容と課題内容は講義途中に提示します

(作成したファイル類は manaba に提出)

注意：今回は OS に強く依存する内容です

Ubuntu / Linux / Unix に固有の内容が殆どです

Windows ではたぶん使えません

- でも同様な別の方法でできるはず、きっと
- 各自で調べてみて下さい (すみません)

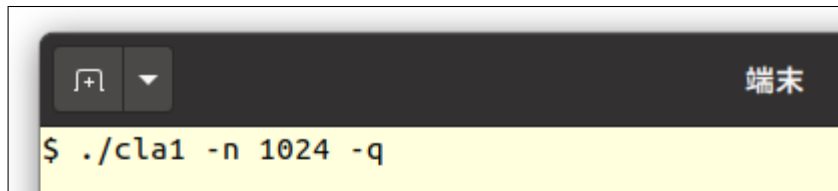
Mac ではどうなのかよく分かりません

- 結構似ているらしい
- gprof は使えず別の方法を使うらしい
- 各自で調べてみて下さい (すみません)

(独自) main 関数の活用

コマンドライン引数 argc と argv

端末のコマンドライン引数でのパラメータ指定法



この上なく便利（最初は不便そうに見えるけど）

- シェルのヒストリ機能：同じ引数で再実行
- シェルのコマンドライン編集機能：似た別の引数にして実行
- シェルスクリプト：一連の実行手順の完全自動化

その仕組み：コマンドライン引数 `argc`, `argv`

- C 言語の `main` 関数の引数 (`argc`, `argv`) に与えられる
- `int main(int argc, char *argv[])`

コマンドライン引数はとても便利 (シェルスクリプト)

まだ scanf 関数で消耗しているの？

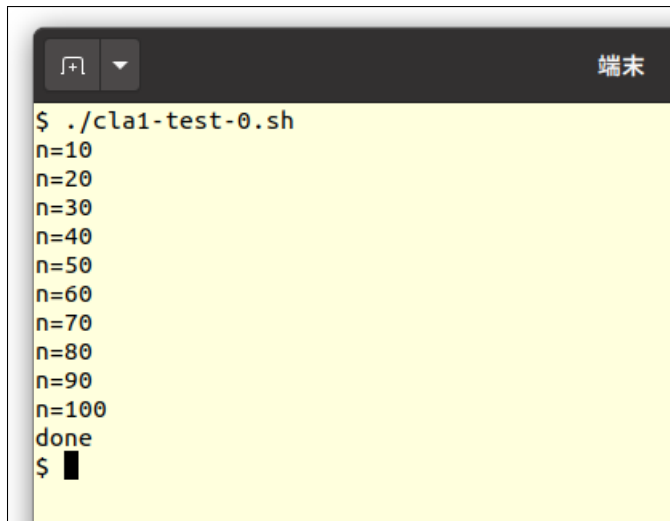
シェルスクリプトの例：引数 n を 10 から 100 まで 10 刻みで自動実行

- 各 n での出力結果をファイル (out-10.txt, out-20.txt, ...) に記録
- コンピュータにずっと張り付いた手作業をなくせる
- 実行時間の評価実験などを全自動で実行できてものすごく便利

```
#!/bin/sh
n=10
while [ ${n} -le 100 ] ; do      # n <= 100 以下の間ループ
    echo "n=${n}"                # n の値を画面表示
    ./clal -n ${n} > out-${n}.txt # 表示をリダイレクト
    n=`expr ${n} + 10`           # n の値を 10 増やす
done
echo "done"                      # 完了を画面表示
```

一発 Enter キーを押すだけで後は全部やってくれる

先程のシェルスクリプトの実行例

A terminal window with a dark header bar containing a window icon and a dropdown arrow on the left, and the text "端末" (Terminal) on the right. The main area has a light yellow background and displays the following text:

```
$ ./cla1-test-0.sh  
n=10  
n=20  
n=30  
n=40  
n=50  
n=60  
n=70  
n=80  
n=90  
n=100  
done  
$ █
```

10 種類のパラメータで cla1 の実行を全自動で完了

main(int argc, char *argv[]) の引数の値

argc : コマンドライン引数の数

argv : 文字列の配列

- argv[0] : プログラム名の文字列
- argv[1] : 第 1 引数の文字列
- argv[2] : 第 2 引数の文字列
- ...
- argv[argc-1] : 第 (argc-1) 引数の文字列

ムダ知識

- argc = “argument count” のこと
- argv = “argument vector” のこと

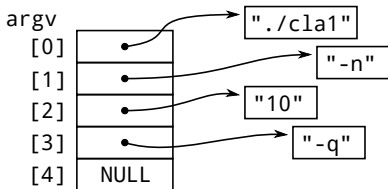
コマンドライン引数と main 関数引数 argc & argv

コマンドライン引数の例

```
$ ./cla1 -n 10 -q
```

このときの `main(int argc, char *argv[])` の引数の値：

- `argc = 4`
- `argv[0] = "./cla1"`
- `argv[1] = "-n"`
- `argv[2] = "10"`
- `argv[3] = "-q"`



データ型

- `argc` : 整数 (`int` 型)
- 各 `argv[i]` : 文字列 (`char*`型)

コマンドライン引数の解釈

例：コマンドラインで実行パラメータを指定

```
$ ./cla1 -n 10 -q
```

プログラムの変数値を以下のように設定したい

- `int arg_n = 10` — 整数値の設定
- `int arg_q = 1` — 指定の有無の設定

疑問：`argc` と `argv` の解析を実現するか？

コマンドライン引数の解析例 1 : cla1.c

cla1 実行例

コマンドライン引数を解析した結果を表示

```
$ ./cla1
-q : NO
-n : 100
$ ./cla1 -q
-q : YES
-n : 100
$ ./cla1 -n 123
-q : NO
-n : 123
$ ./cla1 -n 123 -q
-q : YES
-n : 123
$ ./cla1 -q -n 123
-q : YES
-n : 123
```

注意ポイント

- -q と -n *N* オプションは必須ではない
(無指定のときはデフォルト値が設定される)
- -q と -n *N* の順番は

解析ループの概略

概要 : argv[1], argv[2], ..., argv[argc-1] の順番で解析してゆく

```
for (int i = 1; i < argc; i++) {  
    ... argv[i] を 解 析 （ 次 の ス ラ イ ド で 説 明 ） ...  
}
```

コマンドライン引数からパラメータの値の取得

各 i に対する `argv[i]` の解析処理内容

```
if (strcmp(argv[i], "-q") == 0) {  
    arg_q = 1;  
} else if (strcmp(argv[i], "-n") == 0) {  
    if (i+1 >= argc) {  
        usage(argv[0]);  
    }  
    arg_n = atoi(argv[i+1]);  
    i += 1;  
} else {  
    usage(argv[0]);  
}
```

`argv[i] = "-q"` の場合

- `arg_q = 1` に設定

`argv[i] = "-n"` の場合

- `arg_n` に `argv[i+1]` を整数に変換した値を設定

cla1.c ソースコード (1/4)

このプログラムの動作

- -n オプションで整数値を指定
- -q オプションでフラグをセット (指定の有無)

```
/* cla1.c: コマンドライン引数の解析の例 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* コマンドライン引数の解析結果の記憶 */
/*   -n N (整数) */
#define ARG_N_DEFAULT 100 // -n 100
int arg_n = ARG_N_DEFAULT;
/*   -q (指定の有/無) */
#define ARG_Q_DEFAULT 0 // off
int arg_q = ARG_Q_DEFAULT;
```

パラメータのデフォルト値

- arg_n = 100
- arg_q = 0 (off)

clal.c ソースコード (2/4)

```
void parse_arg(int argc, char *argv[]);
void print_param(void);
void usage(char *prog);

int main(int argc, char *argv[])
{
    /* コマンドライン引数の解析 */
    parse_arg(argc, argv);
    /* パラメータの表示 */
    print_param();
    /* 終了 */
    return 0;
}
```

このプログラムの動作

- コマンドライン引数を解析：parse_arg 関数
- 解析結果を表示：print_param 関数

cla1.c ソースコード (3/4)

```
/* コマンドライン引数の解析 */
void parse_arg(int argc, char *argv[])
{
    for (int i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-q") == 0) {
            arg_q = 1;
        } else if (strcmp(argv[i], "-n") == 0) {
            if (i+1 >= argc) {
                usage(argv[0]);
            }
            arg_n = atoi(argv[i+1]);
            i += 1;
        } else {
            usage(argv[0]);
        }
    }
}
```

clal.c ソースコード (4/4)

```
/* パラメータの表示 */
void print_param(void)
{
    if (arg_q == 0) {
        printf("  -q : NO\n");
    } else {
        printf("  -q : YES\n");
    }
    printf("  -n : %d\n", arg_n);
}

/* コマンドラインの説明の表示 */
void usage(char *prog)
{
    printf("Example to parse command line arguments\n");
    printf("Usage: %s OPTION... \n", prog);
    printf("OPTION: \n");
    printf("  -n NUMBER\n");
    printf("  -q\n");
    exit(1);
}
```

fcpy.c : ファイルのコピー (教科書 List 13-7
の改造)

ファイルのコピー：使い方の例

注意 1: 存在しないファイルは読めません

注意 2: 重要なファイルへ上書きして消失しないよう注意

オプション 2 つ：FILE1.TXT を読んで FILE2.TXT へコピー

```
$ ./fcpy FILE1.TXT FILE2.TXT
```

オプション 1 つ：FILE1.TXT を読んで標準出力へ書き出す

```
$ ./fcpy FILE1.TXT
```

オプション無指定 (1)：標準入力を読んで標準出力へ書き出す

```
$ ./fcpy
```

オプション無指定 (2)：リダイレクトを使える

```
$ ./fcpy < FILE1.TXT > FILE2.TXT
```

如何にしてこの機能を実現するのか？

ファイルのコピー (1) : コピーそのものを行う関数

関数 void fcopy(FILE *fp1, FILE *fp2)

- fp1 から 1 文字読んで fp2 へ書くことの繰り返し
- fp1 のファイルの終端が来ると終わり

```
void fcopy(FILE *fp1, FILE *fp2) {  
    int ch;  
    while ((ch = fgetc(fp1)) != EOF) {  
        fputc(ch, fp2);  
    }  
}
```

この関数の呼び出し方 : fcopy(fp1, fp2);

- fp1 と fp2 へのファイルオープンとかは後述

ファイルのコピー (2) : コマンドライン解析部分

```
FILE *fp1 = stdin;    /*デフォルトの読み出し元*/
FILE *fp2 = stdout;   /*デフォルトの書き込み先*/
char *file1 = NULL;   /*読み出しファイル名*/
char *file2 = NULL;   /*書き込みファイル名*/
if (argc >= 2) { /*FILE1指定あり*/
    file1 = argv[1];
}
if (argc >= 3) { /*FILE2指定あり*/
    file2 = argv[2];
}
if (argc >= 4) { /*余計なものの指定あり*/
    printf("fcpy [FILE1 [FILE2]]\n");
    exit(0);
}
```

コマンドライン引数の個数を調べる

- 2 (プログラム名+引数 1 つ) :
ファイル名 1 を得る
- 3 (プログラム名+引数 2 つ) :
ファイル名 1 と 2 を得る

ファイルのコピー (3) : ファイルのオープンとコピー

```
if (file1 != NULL) { /*FILE1指定有り*/
    if ((fp1 = fopen(file1, "r")) == NULL) {
        printf("CANNOT OPEN %s\n", file1);
        exit(1);
    }
}
if (file2 != NULL) { /*FILE2指定有り*/
    if ((fp2 = fopen(file2, "w")) == NULL) {
        printf("CANNOT OPEN %s\n", file2);
        exit(1);
    }
}
/*コピー実行*/
fcopy(fp1, fp2);
```

ファイルのコピー (4) : コード理解のポイント

変数には予めデフォルト値を設定しておく

```
FILE *fp1 = stdin;    /*デフォルトの読み出し元*/
```

コマンドライン引数で指定があれば値を置き換える

```
if (argc >= 2) { /*FILE1指定あり*/  
    file1 = argv[1];  
}
```

```
if (file1 != NULL) { /*FILE1指定あり*/  
    if ((fp1 = fopen(file1, "r")) == NULL) {
```

コマンドライン引数に指定の有無に関わらず共通な呼出しの方法

```
fcopy(fp1, fp2);
```

プログラムの終了コード

main 関数の返り値はプログラムの終了コード

シェルスクリプトからの実行でとても有用

C 言語プログラム foo.c

```
int main(int argc, char *argv[]) {  
    ... 略 ...  
    if (エラー発生) {  
        return 1; /* プログラムの終了コード=1 */  
    }  
    ... 略 ...  
    return 0; /* プログラムの終了コード=0 */  
}
```

シェルスクリプト (foo 実行でエラー発生すると強制終了)

```
#!/bin/sh  
... 略 ...  
./foo  
if [ $? -ne 0 ]; then  
    echo "ERROR"  
    exit 1  
fi  
# C言語プログラムを実行  
# エラー発生を検出  
# エラーメッセージを表示  
# 強制終了
```

シェル変数 `$?` : 直前の実行プログラムの終了コードが設定

シェルスクリプト例

シェルスクリプト clal-test-1.sh (エラー発生には強制終了)

```
#!/bin/sh
n=10
while [ ${n} -le 100 ]
do
    echo "n=${n}"
    ./foo -n ${n} > out-${n}.txt
    if [ $? -ne 0 ]; then      # エラー発生を検出
        echo "ERROR"         # エラーメッセージを表示
        exit 1               # 強制終了
    fi
    n=`expr ${n} + 10`
done
echo "done"
```

if [\$? -ne 0]; then

- if 文の条件部分は「終了コードが 0 以外 (not equal) ならば」

終了コードを指定したプログラム終了方法：2通りあり

1 main 関数の return 値

```
int main(int argc, char *argv[]) {  
    ... 略 ...  
    if (エラー発生) {  
        return 1;  
    }  
    ... 略 ...  
    return 0;  
}
```

2 exit 関数 (システムコール) の呼出

```
... 略 ...  
if (エラー発生) {  
    exit(1);  
}
```

- exit 関数はプログラム中のどこで呼び出しても良い
- そこで直ちにプログラムを強制終了

(独自) 複数のファイルによるプログラム構成法

ライブラリの利用

数学ライブラリは頻繁に使うかも

数学関数 $\sin()$, $\cos()$, $\tan()$, $\exp()$, $\log()$, ...

C 言語プログラムでの利用方法 2 ステップ

- 1 ソースコード作成時：ヘッダのインクルード `#include <math.h>`
- 2 コンパイル時：数学ライブラリのリンク

table-sin.c : \sin 関数の数表 (0 から 2π まで 12 ステップ)

```
#include <stdio.h>
#include <math.h>
#define NSTEPS 12
int main(int argc, char *argv[]) {
    for (int i = 0; i <= NSTEPS; i++) {
        double th = (double) 2.0 * M_PI * i / NSTEPS;
        printf("sin(2PI*%d/%d)=%.4f\n", i, NSTEPS, sin(th));
    }
    return 0;
}
```

コンパイル (ライブラリ使用の場合)

コンパイル (失敗例; -lm オプションを忘れた)

```
$ cc -o table-sin table-sin.c
/tmp/ccXN1Jlo.o: 関数 'main' 内:
table-sin.c:(.text+0x48): 'sin' に対する定義されていない参照です
collect2: error: ld returned 1 exit status
```

コンパイル (成功例)

```
$ cc -o table-sin table-sin.c -lm
```

実行例

```
$ ./table-sin
sin(2PI*0/12)=0.0000
sin(2PI*1/12)=0.5000
sin(2PI*2/12)=0.8660
sin(2PI*3/12)=1.0000
sin(2PI*4/12)=0.8660
sin(2PI*5/12)=0.5000
sin(2PI*6/12)=0.0000
sin(2PI*7/12)=-0.5000
sin(2PI*8/12)=-0.8660
sin(2PI*9/12)=-1.0000
sin(2PI*10/12)=-0.8660
sin(2PI*11/12)=-0.5000
sin(2PI*12/12)=-0.0000
```

分割コンパイル

分割コンパイルとは

1つの実行ファイルを複数のファイルで構成する手法

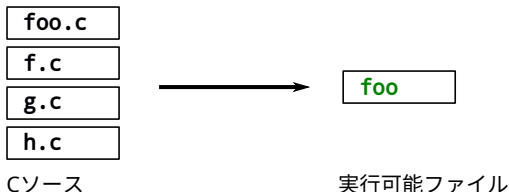
- プログラムの規模が大きいときに有効
- 機能の単位で分割することが多い

利点

- ひとつひとつのソースコードのファイルの行数を小規模にできる
- 開発作業を小規模の単位に分割できる
- 既存ソースコードの理解を小規模の単位に分割できる

例

- ソースファイル：foo.c, f.c, g.c, h.c の4つ
- 実行ファイル：foo



方法 1 : 全部のソースコードを一度にコンパイル

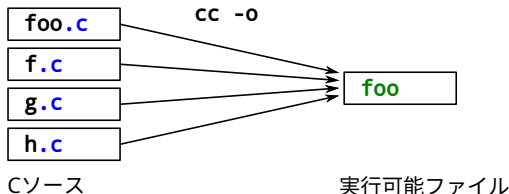
```
$ cc -o foo foo.c f.c g.c h.c
```

利点 : コマンドラインが単純

- まあ, わかりやすいよね
- でもプロはこんなことしない

欠点 : 一部のソースコードを修正する度に全部を再コンパイル

- 問題点 : コンパイル作業に時間がかかる
- プログラム規模が大きくなるほど顕著



位置独立コード PIC ファイル とは

PIC : Position Independent Code (位置独立コード)

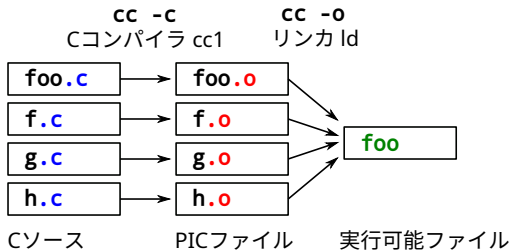
- 別名 : 再配置可能オブジェクト (relocatable object)
- 拡張子 : .o
- コンパイル結果のファイルだが実行可能ファイルではない
- リンク (結合) を行うことで実行可能ファイルにできる
- コンパイル作業の途中のようなもの

PIC ファイルの利点

- 実行ファイルを複数ファイルで構成するとき便利
- 再コンパイルの手間/時間が最小限にできる

PIC ファイルを用いた分割コンパイル

```
$ cc -c foo.c
$ cc -c f.c
$ cc -c g.c
$ cc -c h.c
$ cc -o foo foo.o f.o g.o h.o
```



利点：(例) h.c の変更時には最小限の作業で実行ファイルが作れる

```
$ cc -c h.c
$ cc -o foo foo.o f.o g.o h.o
```

C コンパイラ cc のコマンドラインオプション

形式 : `cc` *OPTION* ファイル...

OPTION (代表的なもの)

- c PIC ファイル形式でコンパイル結果を出力
- o *FILE* 出力のファイル名を *FILE* とする
- l*LIB* リンクするライブラリの指定

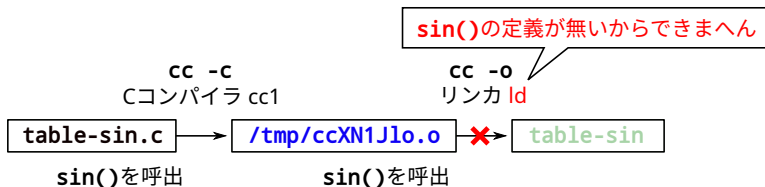
例

- `cc -o foo foo.c`
- `cc -o bar bar.c -lm`
- `cc -c g.c`
- `cc -o baz f1.o f2.o f3.o`

エラーメッセージを読み解く：起きていること (1/2)

(再掲) コンパイルの失敗例：-lm オプションを忘れた

```
$ cc -o table-sin table-sin.c
/tmp/ccXN1Jlo.o: 関数 'main' 内:
table-sin.c:(.text+0x48): 'sin' に対する定義されていない参照です
collect2: error: ld returned 1 exit status
```



Cソース

PICファイル

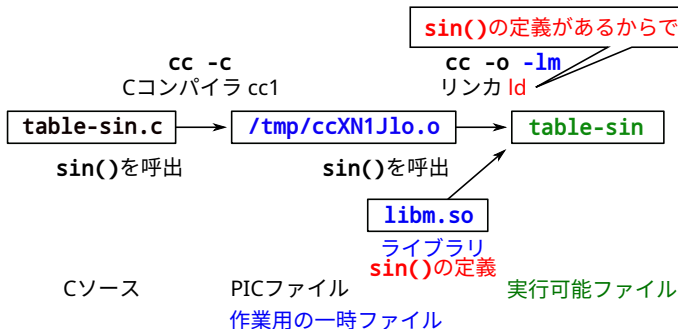
実行可能ファイル

作業用の一時ファイル

エラーメッセージを読み解く：起きていること (2/2)

(再掲) コンパイルの成功例：-lm オプションを付けた

```
$ cc -o table-sin table-sin.c -lm
```



cc コマンド : ツールチェーン

cc コマンド : コンパイルのためのプログラム群を駆動

- ツールチェーン (tool chain) の一種
- 関連ツールを次々と呼出

ツールチェーンの利点

- 利用者は複雑な関連ツールの間の関係を気にせずに簡単に使える

cc での関連ツールと呼出順 (Unix 系での典型例)

- 1 cpp (C プリプロセッサ) : C ソースのマクロ処理
- 2 cc1 (C コンパイラ本体) : C をアセンブリ言語へ翻訳
- 3 as (アセンブラ) : アセンブリ言語を PIC コードへアセンブル
- 4 ld (リンカ/ローダ) : PIC コードとライブラリから実行ファイルへ

分割コンパイルをやってみる

例題 sincos : sin と cos の値の表

実行プログラム名 : sincos

ソースファイル 2 つで構成

- sincos-main.c
- sincos-table.c

コンパイル手順

```
% cc -c sincos-main.c
% cc -c sincos-table.c
% cc -o sincos sincos-main.o sincos-table.o -lm
```

- 3 行目 : .c (C ソース) でなく .o (PIC ファイル) に注意

sincos 実行例 : 0 度から 360 度まで, 30 度ごと

1 行で開始角度, 終了角度, 角度ステップ の 3 つを入力 (整数値)

```
$ ./sincos 0 360 30
  0   0.000000   1.000000
 30   0.500000   0.866025
 60   0.866025   0.500000
 90   1.000000   0.000000
120   0.866025  -0.500000
150   0.500000  -0.866025
180   0.000000  -1.000000
210  -0.500000  -0.866025
240  -0.866025  -0.500000
270  -1.000000  -0.000000
300  -0.866025   0.500000
330  -0.500000   0.866025
360  -0.000000   1.000000
```

左から, 角度, sin, cos の値

ファイル 1 : sincos-main.c

```
#include <stdio.h>
#include <stdlib.h>
void sincos_table(int d1, int d2, int dstep);

int main(int argc, char **argv) {
    int d1 = 0, d2 = 0, ds = 0;
    if (argc < 4) {
        fprintf(stderr,
            "ERR: 開始角度    終了角度    角度ステップ\n");
        exit(1);
    }
    d1 = atoi(argv[1]);
    d2 = atoi(argv[2]);
    ds = atoi(argv[3]);
    sincos_table(d1, d2, ds);
    return 0;
}
```

ファイル 2 : sincos-table.c

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

void sincos_table(int d1, int d2, int dstep) {
    for (int d = d1; d <= d2; d += dstep) {
        double th = M_PI * d / 180.0;
        printf("%3d %9.6f %9.6f\n", d, sin(th), cos(th));
    }
}
```

- 角度 (0 から 360) をラジアン (0 から 2π) に変換
- M_PI (マクロによる記号定数) は円周率 π の値に定義

分割コンパイルの利点

デバッグ時の再コンパイル作業の時間短縮

本当に必要な作業だけすれば良い

例

- sincos-table.c を OK
- sincos-main.c を修正
- 以下の作業で OK
(sincos-table.c の再コンパイルは不要)

```
$ cc -c sincos-main.c  
$ cc -o sincos sincos-main.o sincos-table.o -lm
```

ソースファイル数が多いと時間の節約効果大きい

- コンパイル手順自動化ツールの make と組み合わせると便利

(独自) パフォーマンス測定と改善法

プログラムを高速化する前に忘れてはならないこと

頑張ってどんなに高速化してもプログラムが正しくなければ意味がない

まずは遅くていいから正しく動くプログラムを書こう

- 単純明快で素直な構造のプログラム
- 拡張・改造・理解が容易なプログラム

高速化を考えるのはそれから

パフォーマンス測定を行ってみる

- そもそも実行時間は本当に遅いのか？
- 実行時間の大部分を占めているのはプログラム中のどの部分か？

本当に遅いならば遅い部分を集中的へ改善の努力を投入

- 的外れな部分の改善は労力の浪費

実行時間の測定

time コマンドによる実行時間の測定

例：あるプログラムの実行時間測定
(Ubuntu 上 bash にて; かなりシステムに依存)

```
$ time ./sort-bubble -n 100000 -q -nt
real    0m50.311s
user    0m49.612s
sys     0m0.068s
```

見方

- real 0m50.311s
Enter キー押してから実行が終わるまでの時間 [分秒]
- user 0m49.612s
ユーザー時間 (ユーザープログラムで動作していた時間) [分秒]
- sys 0m0.068s
システム時間 (OS 内で動作していた時間) [分秒]

※若干の誤差あり

time コマンドの応用例

プログラムの実行速度の測定

- アルゴリズムの良し悪しの比較

プログラム性能の改善

- プログラム修正による速度向上の測定

プロファイラによるボトルネックの発見

プロファイラとは

プログラム内の関数それぞれでの実行時間を測定するツール

(イメージ) プロファイラの出力

- 全体の実行時間：100 秒
- 関数 foo：70 秒 (70%)
- 関数 bar：20 秒 (30%)
- 関数 xyz：10 秒 (10%)

利点：どの関数の実行が遅いか一目瞭然

- プログラムの実行を遅くする主原因の特定
- 上の例：関数 foo

遅い部分の改善が全体の速度向上の効果が高い

- 使用しているアルゴリズム/データ構造を再検討, など
- 関数 xyz を倍速にしても全体では 5%の高速化だけ
- 関数 foo を 3 割改善するだけで全体が 21%の高速化

プロファイラ (gporf) 実際の出力例

%	cumulative	self		self	total	
time	seconds	seconds	calls	s/call	s/call	name
35.79	0.59	0.59	3333051	0.00	0.00	partition
30.03	1.09	0.50	76836245	0.00	0.00	array_swap
24.26	1.49	0.40	3333051	0.00	0.00	verify
3.64	1.55	0.06	1	0.06	0.06	array_alloc
2.43	1.59	0.04	1	0.04	1.49	quick_sort
1.82	1.62	0.03	1	0.03	0.03	array_init_ascend
0.61	1.63	0.01	5000000	0.00	0.00	random_get
0.61	1.64	0.01	1	0.01	0.08	array_init_random
0.61	1.65	0.01	1	0.01	0.01	array_is_ascend_permu
0.30	1.65	0.01	2	0.00	0.00	debug_array_print
0.00	1.65	0.00	4	0.00	0.00	debug_print
0.00	1.65	0.00	1	0.00	0.00	array_free
0.00	1.65	0.00	1	0.00	0.08	init_array
0.00	1.65	0.00	1	0.00	0.00	parse_arg
0.00	1.65	0.00	1	0.00	1.50	sort
0.00	1.65	0.00	1	0.00	0.00	timer_get
0.00	1.65	0.00	1	0.00	0.00	timer_start

- name 欄 (右端) : 対象の関数名
- time 欄 : 実行時間の総和 (実行時間全体での比率%)
- self seconds 欄 : 実行時間の総和 (秒)
- calls 欄 : その関数を呼出した回数

プロファイラの利用方法 (Ubuntu/Unix 系)

1. コンパイル時：-p オプションを付加

```
$ cc -p -o foo foo.c
```

2. 普通に実行：プロファイリング情報ファイル gmon.out が自動生成

```
$ ./foo
```

- 注意：実行時間はある程度の長さがないと正確性に欠ける
- ≥ 10 秒程度?
- 時間測定の精度が 0.01 秒程度のため

3. プロファイル結果の表示：gprof コマンドを使用

```
$ gprof foo
```

- gprof はプログラムコードと gmon.out を参照して表示

番外編の課題 1 : sincos-main.c の改造

コマンドラインでの開始角度, 終了角度, 角度ステップを省略可能にする

- 引数が 3 つのとき: 開始角度, 終了角度, 角度ステップ, の指定

```
$ ./sincos 0 180 45
```

- 引数が 2 つのとき: 開始角度, 終了角度, の指定
(角度ステップ=30 とする)

```
$ ./sincos 30 180
```

- 引数が 1 つのとき: 開始角度, の指定
(終了角度=360, 角度ステップ=30 とする)

```
$ ./sincos 60
```

- 引数が 0 のとき:
(開始角度=0, 終了角度=360, 角度ステップ=30 とする)

```
$ ./sincos
```

おわり