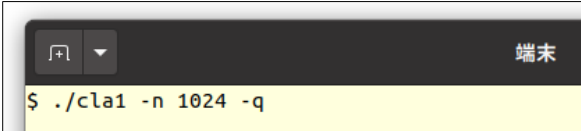
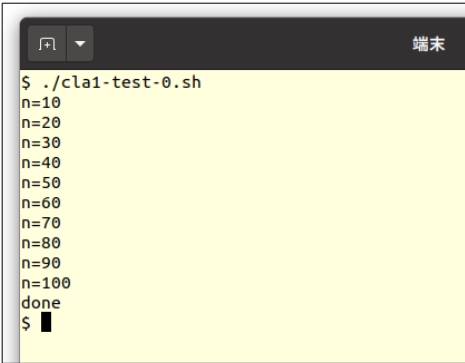


	もくじ	今回 (#15) の内容：シラバスでの該当部分
<div> <div>#15</div> <div>C 言語プログラムの実践的開発技法</div> <div>2022 年度 / プログラミング及び実習 III</div> </div> <div> <div>角川裕次</div> <div>龍谷大学 先端理工学部</div> </div> <div>1 / 63</div>	<div> <div>1 (独自) main 関数の活用</div> <ul style="list-style-type: none"> <li>■ コマンドライン引数 argc と argv</li> <li>■ コマンドライン引数の解析例 1 : cla1.c</li> <li>■ fcpy.c : ファイルのコピー (教科書 List 13-7 の改造)</li> <li>■ プログラムの終了コード</li> </ul> <div>2 (独自) 複数のファイルによるプログラム構成法</div> <ul style="list-style-type: none"> <li>■ ライブラリの利用</li> <li>■ 分割コンパイル</li> <li>■ 分割コンパイルをやってみる</li> </ul> <div>3 (独自) パフォーマンス測定と改善法</div> <ul style="list-style-type: none"> <li>■ 実行時間の測定</li> <li>■ プロファイラによるボトルネックの発見</li> </ul> <div>2 / 63</div> </div>	<div>小テーマ: C 言語プログラムの実践的開発技法</div> <div> 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 </div> <div>3 / 63</div>
重要概念リスト	今回の実習・課題 (manaba へ提出)	注意：今回は OS に強く依存する内容です
<ul style="list-style-type: none"> <li>■ コマンドライン引数 argc &amp; argv</li> <li>■ 終了コード : exit 関数の引数, main 関数の戻り値</li> <li>■ ライブラリのリンク</li> <li>■ 分割コンパイルと PIC ファイル</li> <li>■ time コマンドによる実行時間の測定</li> <li>■ gprof コマンドによる詳細な実行時間の分析</li> </ul> <div>4 / 63</div>	<div>実習内容と課題内容は講義途中に提示します</div> <div>(作成したファイル類は manaba に提出)</div> <div>5 / 63</div>	<div>Ubuntu / Linux / Unix に固有の内容が殆どです</div> <div>Windows ではたぶん使えません</div> <ul style="list-style-type: none"> <li>■ でも同様な別の方法でできるはず, きっと</li> <li>■ 各自で調べてみて下さい (すみません)</li> </ul> <div>Mac ではどうなのかよく分かりません</div> <ul style="list-style-type: none"> <li>■ 結構似ているらしい</li> <li>■ gprof は使えず別の方法を使うらしい</li> <li>■ 各自で調べてみて下さい (すみません)</li> </ul> <div>6 / 63</div>

		端末のコマンドライン引数でのパラメータ指定法
<p>(独自) main 関数の活用</p>	<p>コマンドライン引数 argc と argv</p>	<div></div> <p>この上なく便利 (最初は不便そうに見えるけど)</p> <ul style="list-style-type: none"><li>■ シェルの履歴機能: 同じ引数で再実行</li><li>■ 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</li><li>■ シェルスクリプト: 一連の実行手順の完全自動化</li></ul> <p>その仕組み: コマンドライン引数 argc, argv</p> <ul style="list-style-type: none"><li>■ C 言語の main 関数の引数 (argc, argv) に与えられる</li><li>■ <code>int main(int argc, char *argv[])</code></li></ul>
7 / 63	8 / 63	9 / 63
コマンドライン引数はとても便利 (シェルスクリプト)	先程のシェルスクリプトの実行例	main(int argc, char *argv[]) の引数の値 <span>★重要★</span>
<p>まだ scanf 関数で消耗しているの?</p> <p>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</p> <ul style="list-style-type: none"><li>■ 長所: 一発 Enter キーを押すだけで後は全部やってくれる</li><li>■ 各 n での出力をファイル (out-10.txt, out-20.txt, ...) に自動記録</li><li>■ コンピュータにずっと張り付いた手作業をなくせる</li><li>■ 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</li></ul> <pre>#!/bin/sh n=10 while [ \${n} -le 100 ]; do    # n &lt;= 100 以下の間ループ     echo "n=\${n}"            # n の値を画面表示     ./cla1 -n \${n} &gt; 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</pre>	<div></div> <p>10 種類のパラメータで cla1 の実行を全自動で完了</p>	<p>argc: コマンドライン引数の数</p> <p>argv: 文字列の配列</p> <ul style="list-style-type: none"><li>■ argv[0]: プログラム名の文字列</li><li>■ argv[1]: 第 1 引数の文字列</li><li>■ argv[2]: 第 2 引数の文字列</li><li>...</li><li>■ argv[argc-1]: 第 (argc-1) 引数の文字列</li></ul> <p>ムダ知識</p> <ul style="list-style-type: none"><li>■ argc = “argument count” のこと</li><li>■ argv = “argument vector” のこと</li></ul>
10 / 63	11 / 63	12 / 63

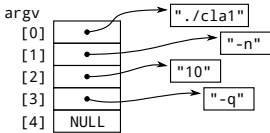
## コマンドライン引数と main 関数引数 argc & argv

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)

13 / 63

## 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?

14 / 63

コマンドライン引数の解析例 1 : cla1.c

15 / 63

## 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

16 / 63

## 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) ...
}
```

17 / 63

## 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

18 / 63

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)

19 / 63

cla1.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

20 / 63

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]);
        }
    }
}
```

21 / 63

cla1.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);
}
```

22 / 63

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): リダイレクトを使う

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

如何にしてこの機能を実現するのか?

24 / 63

23 / 63

ファイルのコピー (1) : コピーそのものを行う関数	ファイルのコピー (2) : コマンドライン解析部分	ファイルのコピー (3) : ファイルのオープンとコピー
<p>関数 void fcopy(FILE *fp1, FILE *fp2)</p> <ul style="list-style-type: none"><li>■ fp1 から1文字読んで fp2 へ書くことの繰り返し</li><li>■ fp1 のファイルの終端が来ると終わり</li></ul> <pre>void fcopy(FILE *fp1, FILE *fp2) {     int ch;     while ((ch = fgetc(fp1)) != EOF) {         fputc(ch, fp2);     } }</pre> <p>この関数の呼び出し方: fcopy(fp1, fp2);</p> <ul style="list-style-type: none"><li>■ fp1 と fp2 へのファイルオープンとかは後述</li></ul>	<pre>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 &gt;= 2) { /*FILE1指定あり*/     file1 = argv[1]; } if (argc &gt;= 3) { /*FILE2指定あり*/     file2 = argv[2]; } if (argc &gt;= 4) { /*余計なものの指定あり*/     printf("fcopy [FILE1 [FILE2]]\n");     exit(0); }</pre> <p>コマンドライン引数の個数を調べる</p> <ul style="list-style-type: none"><li>■ 2 (プログラム名+引数1つ): ファイル名1を得る</li><li>■ 3 (プログラム名+引数2つ): ファイル名1と2を得る</li></ul>	<pre>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);</pre>
25 / 63	26 / 63	27 / 63
ファイルのコピー (4) : コード理解のポイント		main 関数の返り値はプログラムの終了コード
<p>変数には予めデフォルト値を設定しておく</p> <pre>FILE *fp1 = stdin; /*デフォルトの読み出し元*/</pre> <p>コマンドライン引数で指定があれば値を置き換える</p> <pre>if (argc &gt;= 2) { /*FILE1指定あり*/     file1 = argv[1]; }</pre> <pre>if (file1 != NULL) { /*FILE1指定あり*/     if ((fp1 = fopen(file1, "r")) == NULL) {</pre> <p>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</p> <pre>fcopy(fp1, fp2);</pre>	プログラムの終了コード	<p>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</p> <pre>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 */ }</pre> <p>シェルスクリプト (foo 実行でエラー発生すると強制終了)</p> <pre>#!/bin/sh ... 略 ... ./foo if [ \$? -ne 0 ]; then # C言語プログラムを実行     echo "ERROR" # エラー発生を検出     exit 1 # エラーメッセージを表示 fi # 強制終了</pre> <p>シェル変数 \$? : 直前の実行プログラムの終了コードが設定</p>
28 / 63	29 / 63	30 / 63

シェルスクリプト例	終了コードを指定したプログラム終了方法：2通りあり★重要★	
<p>シェルスクリプト clal-test-1.sh (エラー発生には強制終了)</p> <pre>#!/bin/sh n=10 while [ \${n} -le 100 ] do     echo "n=\${n}"     ./foo -n \${n} &gt; 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"</pre> <p>if [ \$? -ne 0 ]; then</p> <ul style="list-style-type: none"><li>■ if 文の条件部分は「終了コードが 0 以外 (not equal) ならば」</li></ul>	<p>1 main 関数の return 値</p> <pre>int main(int argc, char *argv[]) {     ... 略 ...     if (エラー発生) {         return 1;     }     ... 略 ...     return 0; }</pre> <p>2 exit 関数 (システムコール) の呼出</p> <pre>... 略 ... if (エラー発生) {     exit(1); }</pre> <ul style="list-style-type: none"><li>■ exit 関数はプログラム中のどこで呼び出しても良い</li><li>■ そこで直ちにプログラムを強制終了</li></ul>	<p>(独自) 複数のファイルによるプログラム構成法</p>
	数学ライブラリは頻繁に使うかも	コンパイル (ライブラリ使用の場合)★重要★
ライブラリの利用	<p>数学関数 <math>\sin()</math>, <math>\cos()</math>, <math>\tan()</math>, <math>\exp()</math>, <math>\log()</math>, ...</p> <p>C 言語プログラムでの利用方法 2 ステップ</p> <p>1 ソースコード作成時：ヘッダのインクルード <code>#include &lt;math.h&gt;</code></p> <p>2 コンパイル時：数学ライブラリのリンク</p> <p>table-sin.c : <math>\sin</math> 関数の数表 (0 から <math>2\pi</math> まで 12 ステップ)</p> <pre>#include &lt;stdio.h&gt; #include &lt;math.h&gt; #define NSTEPS 12 int main(int argc, char *argv[]) {     for (int i = 0; i &lt;= NSTEPS; i++) {         double th = (double) 2.0 * M_PI * i / NSTEPS;         printf("sin(2PI*%d/%d)=%.4f\n", i, NSTEPS, sin(th));     }     return 0; }</pre>	<p>コンパイル (失敗例; -lm オプションを忘れた)</p> <pre>\$ 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</pre> <p>コンパイル (成功例)</p> <pre>\$ cc -o table-sin table-sin.c -lm</pre>

<div data-bbox="123 316 201 343">実行例</div> <div data-bbox="159 403 736 657"><pre>\$ ./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</pre></div> <div data-bbox="714 774 768 794">37 / 63</div>	<div data-bbox="916 523 1059 547">分割コンパイル</div> <div data-bbox="1386 774 1440 794">38 / 63</div>	<div data-bbox="1467 316 1691 343">分割コンパイルとは</div> <div data-bbox="2036 295 2105 319">★重要★</div> <div data-bbox="1505 387 2038 544"><p>1つの実行ファイルを複数のファイルで構成する手法</p><ul style="list-style-type: none"><li>■ プログラムの規模が大きいときに有効</li><li>■ 機能の単位で分割することが多い</li></ul><p>利点</p><ul style="list-style-type: none"><li>■ 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</li><li>■ 開発作業を小規模の単位に分割できる</li><li>■ 既存ソースコードの理解を小規模の単位に分割できる</li></ul><p>例</p><ul style="list-style-type: none"><li>■ ソースファイル：foo.c, f.c, g.c, h.c の4つ</li><li>■ 実行ファイル：foo</li></ul><div data-bbox="1610 635 1928 772"><pre>graph LR     subgraph CSources [Cソース]         foo_c[foo.c]         f_c[f.c]         g_c[g.c]         h_c[h.c]     end     foo_c --&gt; foo_exe[foo]     f_c --&gt; foo_exe     g_c --&gt; foo_exe     h_c --&gt; foo_exe     style foo_exe fill:#fff,stroke:#008000,stroke-width:2px</pre><p>Cソース</p><p>実行可能ファイル</p></div><div data-bbox="2058 774 2112 794">39 / 63</div></div>
<div data-bbox="123 823 665 850">方法1：全部のソースコードを一度にコンパイル</div> <div data-bbox="159 911 736 1260"><pre>\$ cc -o foo foo.c f.c g.c h.c</pre><p>利点：コマンドラインが単純</p><ul style="list-style-type: none"><li>■ まあ、わかりやすいよね</li><li>■ でもプロはこんなことしない</li></ul><p>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</p><ul style="list-style-type: none"><li>■ 問題点：コンパイル作業に時間がかかる</li><li>■ プログラム規模が大きくなるほど顕著</li></ul><div data-bbox="271 1121 624 1260"><pre>graph LR     subgraph CSources [Cソース]         foo_c[foo.c]         f_c[f.c]         g_c[g.c]         h_c[h.c]     end     foo_c -- "cc -o" --&gt; foo_exe[foo]     f_c -- "cc -o" --&gt; foo_exe     g_c -- "cc -o" --&gt; foo_exe     h_c -- "cc -o" --&gt; foo_exe     style foo_exe fill:#fff,stroke:#008000,stroke-width:2px</pre><p>Cソース</p><p>実行可能ファイル</p></div><div data-bbox="714 1281 768 1302">40 / 63</div></div>	<div data-bbox="795 823 1184 850">位置独立コード PIC ファイル とは</div> <div data-bbox="833 903 1317 1165"><p>PIC : Position Independent Code (位置独立コード)</p><ul style="list-style-type: none"><li>■ 別名：再配置可能オブジェクト (relocatable object)</li><li>■ 拡張子：.o</li><li>■ 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</li><li>■ リンク (結合) を行うことで実行可能ファイルにできる</li><li>■ コンパイル作業の途中のようなもの</li></ul><p>PIC ファイルの利点</p><ul style="list-style-type: none"><li>■ 実行ファイルを複数ファイルで構成するとき便利</li><li>■ 再コンパイルの手間/時間が最小限にできる</li></ul><div data-bbox="1386 1281 1440 1302">41 / 63</div></div>	<div data-bbox="1467 823 1886 850">PIC ファイルを用いた分割コンパイル</div> <div data-bbox="2036 802 2105 826">★重要★</div> <div data-bbox="1505 898 2020 1217"><pre>\$ 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</pre><div data-bbox="1610 1010 1928 1185"><pre>graph LR     subgraph CSources [Cソース]         foo_c[foo.c]         f_c[f.c]         g_c[g.c]         h_c[h.c]     end     subgraph PICFiles [PICファイル]         foo_o[foo.o]         f_o[f.o]         g_o[g.o]         h_o[h.o]     end     subgraph Executable [実行可能ファイル]         foo_exe[foo]     end     foo_c -- "cc -c" --&gt; foo_o     f_c -- "cc -c" --&gt; f_o     g_c -- "cc -c" --&gt; g_o     h_c -- "cc -c" --&gt; h_o     foo_o -- "cc -o" --&gt; foo_exe     f_o -- "cc -o" --&gt; foo_exe     g_o -- "cc -o" --&gt; foo_exe     h_o -- "cc -o" --&gt; foo_exe     style foo_exe fill:#fff,stroke:#008000,stroke-width:2px</pre><p>Cソース</p><p>PICファイル</p><p>実行可能ファイル</p></div><p>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</p><pre>\$ cc -c h.c \$ cc -o foo foo.o f.o g.o h.o</pre><div data-bbox="2058 1281 2112 1302">42 / 63</div></div>

<div>C コンパイラ cc のコマンドラインオプション</div> <div>形式: cc OPTION ファイル...</div> <div>OPTION (代表的なもの)</div> <div><div>-cPIC ファイル形式でコンパイル結果を出力</div><div>-o FILE出力のファイル名を FILE とする</div><div>-llibリンクするライブラリの指定</div></div> <div>例</div> <div><div>■ cc -o foo foo.c</div><div>■ cc -o bar bar.c -lm</div><div>■ cc -c g.c</div><div>■ cc -o baz f1.o f2.o f3.o</div></div> <div>43 / 63</div>	<div>エラーメッセージを読み解く: 起きていること (1/2)<div>★重要★</div></div> <div>(再掲) コンパイルの失敗例: -lm オプションを忘れた</div> <div><div>\$ cc -o table-sin table-sin.c</div><div>/tmp/ccXN1Jlo.o: 関数 'main' 内:</div><div>table-sin.c:(.text+0x48): 'sin' に対する定義されていない参照です</div><div>collect2: error: ld returned 1 exit status</div></div> <div><div>cc -c</div><div>Cコンパイラ cc1</div><div>table-sin.c</div><div>sin()を呼出</div><div>/tmp/ccXN1Jlo.o</div><div>cc -o</div><div>リンカ ld</div><div>table-sin</div><div>sin()を呼出</div><div>sin()の定義が無いからできません</div></div> <div><div>Cソース</div><div>PICファイル</div><div>作業用の一時ファイル</div><div>実行可能ファイル</div></div> <div>44 / 63</div>	<div>エラーメッセージを読み解く: 起きていること (2/2)<div>★重要★</div></div> <div>(再掲) コンパイルの成功例: -lm オプションを付けた</div> <div><div>\$ cc -o table-sin table-sin.c -lm</div></div> <div><div>cc -c</div><div>Cコンパイラ cc1</div><div>table-sin.c</div><div>sin()を呼出</div><div>/tmp/ccXN1Jlo.o</div><div>cc -o -lm</div><div>リンカ ld</div><div>table-sin</div><div>sin()を呼出</div><div>libm.so</div><div>ライブラリ</div><div>sin()の定義</div><div>sin()の定義があるからできますよ</div></div> <div><div>Cソース</div><div>PICファイル</div><div>作業用の一時ファイル</div><div>実行可能ファイル</div></div> <div>45 / 63</div>
<div>cc コマンド: ツールチェーン</div> <div>cc コマンド: コンパイルのためのプログラム群を駆動</div> <div><div>■ ツールチェーン (tool chain) の一種</div><div>■ 関連ツールを次々と呼出</div></div> <div>ツールチェーンの利点</div> <div><div>■ 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</div></div> <div>cc での関連ツールと呼出順 (Unix 系での典型例)</div> <div><div>1 cpp (C プリプロセッサ): C ソースのマクロ処理</div><div>2 cc1 (C コンパイラ本体): C をアセンブリ言語へ翻訳</div><div>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</div><div>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</div></div> <div>46 / 63</div>	<div>分割コンパイルをやってみる</div> <div>47 / 63</div>	<div>例題 sincos: sin と cos の値の表</div> <div>実行プログラム名: sincos</div> <div>ソースファイル 2 つで構成</div> <div><div>■ sincos-main.c</div><div>■ sincos-table.c</div></div> <div>コンパイル手順</div> <div><div>% cc -c sincos-main.c</div><div>% cc -c sincos-table.c</div><div>% cc -o sincos sincos-main.o sincos-table.o -lm</div></div> <div><div>■ 3 行目: .c (C ソース) でなく .o (PIC ファイル) に注意</div></div> <div>48 / 63</div>



<div>sincos 実行例：0 度から 360 度まで, 30 度ごと</div> <div>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)</div> <div><pre>\$ ./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</pre></div> <div>左から, 角度, sin, cos の値</div> <div>49 / 63</div>	<div>ファイル 1：sincos-main.c</div> <div><pre>#include &lt;stdio.h&gt; #include &lt;stdlib.h&gt; void sincos_table(int d1, int d2, int dstep);  int main(int argc, char **argv) {     int d1 = 0, d2 = 0, ds = 0;     if (argc &lt; 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; }</pre></div> <div>50 / 63</div>	<div>ファイル 2：sincos-table.c</div> <div><pre>#include &lt;stdio.h&gt; #include &lt;math.h&gt;  void sincos_table(int d1, int d2, int dstep) {     for (int d = d1; d &lt;= d2; d += dstep) {         double th = M_PI * d / 180.0;         printf("%3d %9.6f %9.6f\n", d, sin(th), cos(th));     } }</pre></div> <div>■ 角度 (0 から 360) をラジアン (0 から <math>2\pi</math>) に変換 ■ M_PI (マクロによる記号定数) は円周率 <math>\pi</math> の値に定義</div> <div>51 / 63</div>
<div>分割コンパイルの利点</div> <div>デバッグ時の再コンパイル作業の時間短縮</div> <div>本当に必要な作業だけすれば良い</div> <div>例</div> <div>■ 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)</div> <div><pre>\$ cc -c sincos-main.c \$ cc -o sincos sincos-main.o sincos-table.o -lm</pre></div> <div>ソースファイル数が多いと時間の節約効果大きい</div> <div>■ 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</div> <div>52 / 63</div>	<div>(独自) パフォーマンス測定と改善法</div> <div>53 / 63</div>	<div>★重要★</div> <div>プログラムを高速化する前に忘れてはならないこと</div> <div>頑張っても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</div> <div>まずは遅くていいから正しく動くプログラムを書こう</div> <div>■ 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</div> <div>高速化を考えるのはそれから</div> <div>パフォーマンス測定を行ってみる</div> <div>■ 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?</div> <div>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</div> <div>■ 的外れな部分の改善は労力の浪費</div> <div>54 / 63</div>

	time コマンドによる実行時間の測定 <span>★重要★</span>	time コマンドの応用例																																																																																																																														
実行時間の測定	<div>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)</div> <div><pre>\$ time ./sort-bubble -n 100000 -q -nt real    0m50.311s user    0m49.612s sys     0m0.068s</pre></div> <div>見方</div> <div><ul style="list-style-type: none"><li>■ real 0m50.311s Enter キー押してから実行が終わるまでの時間 [分秒]</li><li>■ 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]</li><li>■ sys 0m0.068s システム時間 (OS 内で動作していた時間) [分秒]</li></ul></div> <div>※若干の誤差あり</div>	<div>プログラムの実行速度の測定</div> <div><ul style="list-style-type: none"><li>■ アルゴリズムの良し悪しの比較</li></ul></div> <div>プログラム性能の改善</div> <div><ul style="list-style-type: none"><li>■ プログラム修正による速度向上の測定</li></ul></div>																																																																																																																														
55 / 63	56 / 63	57 / 63																																																																																																																														
	プロファイラとは	プロファイラ (gprof) 実際の出力例																																																																																																																														
プロファイラによるボトルネックの発見	<div>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</div> <div>(イメージ) プロファイラの出力</div> <div><ul style="list-style-type: none"><li>■ 全体の実行時間：100 秒</li><li>■ 関数 foo：70 秒 (70%)</li><li>■ 関数 bar：20 秒 (30%)</li><li>■ 関数 xyz：10 秒 (10%)</li></ul></div> <div>利点：どの関数の実行が遅いか一目瞭然</div> <div><ul style="list-style-type: none"><li>■ プログラムの実行を遅くする主原因の特定</li><li>■ 上の例：関数 foo</li></ul></div> <div>遅い部分の改善が全体の速度向上の効果が高い</div> <div><ul style="list-style-type: none"><li>■ 使用しているアルゴリズム/データ構造を再検討, など</li><li>■ 関数 xyz を倍速にしても全体では 5%の高速化だけ</li><li>■ 関数 foo を 3 割改善するだけで全体が 21%の高速化</li></ul></div>	<div><table><tr><th>% time</th><th>cumulative seconds</th><th>self seconds</th><th>calls</th><th>self s/call</th><th>total s/call</th><th>name</th></tr><tr><td>35.79</td><td>0.59</td><td>0.59</td><td>3333051</td><td>0.00</td><td>0.00</td><td>partition</td></tr><tr><td>30.03</td><td>1.09</td><td>0.50</td><td>76836245</td><td>0.00</td><td>0.00</td><td>array_swap</td></tr><tr><td>24.26</td><td>1.49</td><td>0.40</td><td>3333051</td><td>0.00</td><td>0.00</td><td>verify</td></tr><tr><td>3.64</td><td>1.55</td><td>0.06</td><td>1</td><td>0.06</td><td>0.06</td><td>array_alloc</td></tr><tr><td>2.43</td><td>1.59</td><td>0.04</td><td>1</td><td>0.04</td><td>1.49</td><td>quick_sort</td></tr><tr><td>1.82</td><td>1.62</td><td>0.03</td><td>1</td><td>0.03</td><td>0.03</td><td>array_init_ascend</td></tr><tr><td>0.61</td><td>1.63</td><td>0.01</td><td>5000000</td><td>0.00</td><td>0.00</td><td>random_get</td></tr><tr><td>0.61</td><td>1.64</td><td>0.01</td><td>1</td><td>0.01</td><td>0.08</td><td>array_init_random</td></tr><tr><td>0.61</td><td>1.65</td><td>0.01</td><td>1</td><td>0.01</td><td>0.01</td><td>array_is_ascend_permu</td></tr><tr><td>0.30</td><td>1.65</td><td>0.01</td><td>2</td><td>0.00</td><td>0.00</td><td>debug_array_print</td></tr><tr><td>0.00</td><td>1.65</td><td>0.00</td><td>4</td><td>0.00</td><td>0.00</td><td>debug_print</td></tr><tr><td>0.00</td><td>1.65</td><td>0.00</td><td>1</td><td>0.00</td><td>0.00</td><td>array_free</td></tr><tr><td>0.00</td><td>1.65</td><td>0.00</td><td>1</td><td>0.00</td><td>0.08</td><td>init_array</td></tr><tr><td>0.00</td><td>1.65</td><td>0.00</td><td>1</td><td>0.00</td><td>0.00</td><td>parse_arg</td></tr><tr><td>0.00</td><td>1.65</td><td>0.00</td><td>1</td><td>0.00</td><td>1.50</td><td>sort</td></tr><tr><td>0.00</td><td>1.65</td><td>0.00</td><td>1</td><td>0.00</td><td>0.00</td><td>timer_get</td></tr><tr><td>0.00</td><td>1.65</td><td>0.00</td><td>1</td><td>0.00</td><td>0.00</td><td>timer_start</td></tr></table></div> <div><ul style="list-style-type: none"><li>■ name 欄 (右端)：対象の関数名</li><li>■ time 欄：実行時間の総和 (実行時間全体での比率%)</li><li>■ self seconds 欄：実行時間の総和 (秒)</li><li>■ calls 欄：その関数を呼出した回数</li></ul></div>	% time	cumulative seconds	self seconds	calls	self s/call	total 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
% time	cumulative seconds	self seconds	calls	self s/call	total 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																																																																																																																										
58 / 63	59 / 63	60 / 63																																																																																																																														

プロファイラの利用方法 (Ubuntu/Unix 系)★重要★	番外編の課題 1 : sincos-main.c の改造	
<div>1. コンパイル時 : -p オプションを付加</div> <div>\$ cc -p -o foo foo.c</div> <div>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</div> <div>\$ ./foo</div> <div>■ 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</div> <div>■ ≥ 10 秒程度?</div> <div>■ 時間測定の精度が 0.01 秒程度のため</div> <div>3. プロファイル結果の表示 : gprof コマンドを使用</div> <div>\$ gprof foo</div> <div>■ gprof はプログラムコードと gmon.out を参照して表示</div> <div>61 / 63</div>	<div>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</div> <div>■ 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</div> <div>\$ ./sincos 0 180 45</div> <div>■ 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)</div> <div>\$ ./sincos 30 180</div> <div>■ 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)</div> <div>\$ ./sincos 60</div> <div>■ 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)</div> <div>\$ ./sincos</div> <div>62 / 63</div>	<div>おわり</div> <div>63 / 63</div>