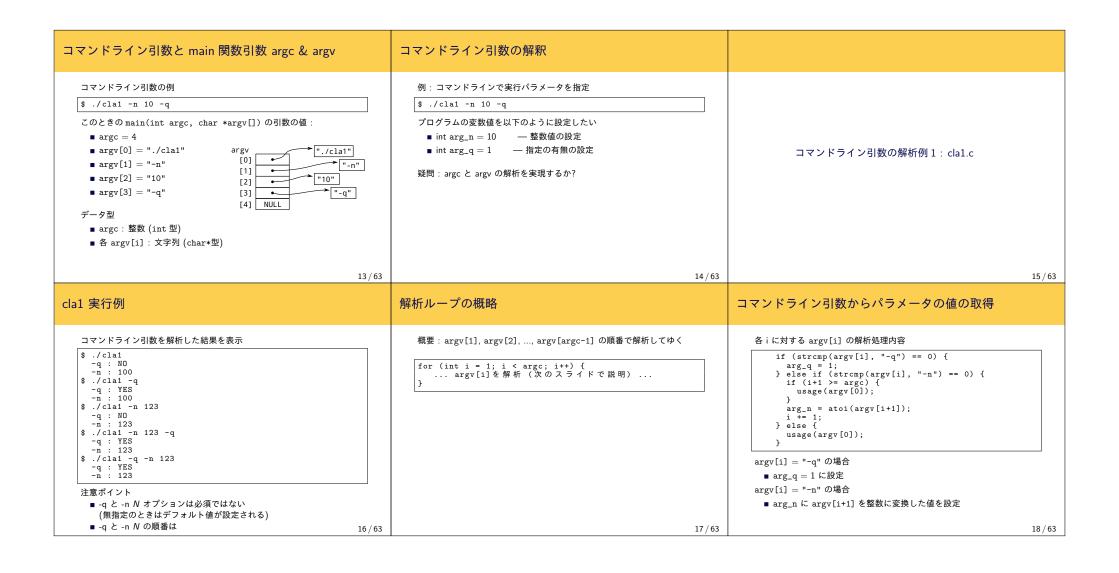
	もくじ	今回 (#15) の内容:シラバスでの該当部分
#15 C 言語プログラムの実践的開発技法 2022 年度 / プログラミング及び実習 III 角川裕次 ^{龍谷大学 先端理工学部}	■ (独自) main 関数の活用 ■ コマンドライン引数 argc と argv ■ コマンドライン引数の解析例 1: cla1.c ■ fcpy.c: ファイルのコピー (教科書 List 13-7 の改造) ■ プログラムの終了コード 2 (独自) 複数のファイルによるプログラム構成法 ■ ライブラリの利用 ■ 分割コンパイル ■ 分割コンパイル ■ 分割コンパイルをやってみる 3 (独自) パフォ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	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
1/63	2/63	3/63
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
■ 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	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)	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)
4/6	5/63	6/63

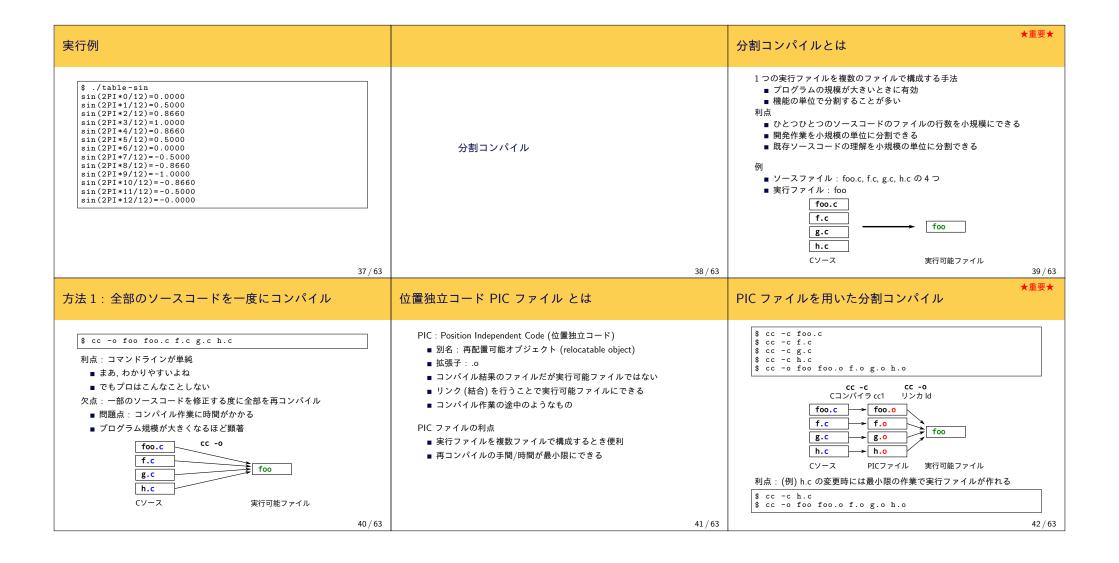




cla1.c ソースコード (1/4) cla1.c ソースコード (2/4) cla1.c ソースコード (3/4) このプログラムの動作 void parse_arg(int argc, char *argv[]); /* コマンドライン引数の解析 */ ■ -n オプションで整数値を指定 void print_param(void); void parse_arg(int argc, char *argv[]) ■ -q オプションでフラグをセット (指定の有無) void usage(char *prog); for (int i = 1; i < argc; i++) { /* cla1.c: コマンドライン引数の解析の例 */ int main(int argc, char *argv[]) if (strcmp(argv[i], "-q") == 0) { #include <stdio.h> arg_q = 1; } else if (strcmp(argv[i], "-n") == 0) { #include <stdlib.h> /* コマンドライン引数の解析 */ #include <string.h> parse_arg(argc, argv); /* パラメータの表示 */ if (i+1 >= argc) { usage(argv[0]); /* コマンドライン引数の解析結果の記憶 */ print_param(); /* -n N (整数) */ /* 終了 */ arg_n = atoi(argv[i+1]); #define ARG_N_DEFAULT 100 // -n 100 return 0; int arg_n = ARG_N_DEFAULT; } else { /* -q (指定の有/無) */ usage(argv[0]); #define ARG_Q_DEFAULT 0 // Off このプログラムの動作 int arg_q = ARG_Q_DEFAULT; } ■ コマンドライン引数を解析: parse_arg 関数 パラメータのデフォルト値 ■ 解析結果を表示: print_param 関数 ■ arg_n = 100 ■ arg_q = 0 (off) 19 / 63 20 / 63 21 / 63 ファイルのコピー: 使い方の例 cla1.c ソースコード (4/4) 注意 1: 存在しないファイルは読めません /* パラメータの表示 */ void print_param(void) 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 if (arg_q == 0) { printf(" -q : NO\n"); オプション 2 つ: FILE1.TXT を読んで FILE2.TXT ヘコピー } else { \$./fcpy FILE1.TXT FILE2.TXT printf(" -q : YES\n"); fcpy.c: ファイルのコピー (教科書 List 13-7 オプション1つ: FILE1.TXT を読んで標準出力へ書き出す の改造) printf(" -n : %d\n", arg_n); \$./fcpy FILE1.TXT /* コマンドラインの説明の表示 */ 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 void usage(char *prog) \$./fcpy printf("Example to parse command line arguments\n"); printf("Usage: %s OPTION... \n", prog); オプション無指定 (2): リダイレクトを使える printf("OPTION: \n"); printf(" -n NUMBER\n"); printf(" -q\n"); exit(1); \$./fcpy < FILE1.TXT > FILE2.TXT 如何にしてこの機能を実現するのか? 22 / 63 23 / 63 24 / 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 関数 void fcopy(FILE *fp1, FILE *fp2) FILE *fp1 = stdin; /*デフォルトの読み出し元*/ if (file1 != NULL) { /*FILE1指定有り*/ FILE *fp2 = stdout; /*デフォルトの書き込み先*/ ■ fp1 から 1 文字読んで fp2 へ書くことの繰り返し if ((fp1 = fopen(file1, "r")) == NULL) { char *file1 = NULL; /*読み出しファイル名*/ printf("CANNOT OPEN %s\n", file1); ■ fp1 のファイルの終端が来ると終わり char *file2 = NULL; /*書き込みファイル名*/ exit(1); if (argc >= 2) { /*FILE1指定あり*/ void fcopy(FILE *fp1, FILE *fp2) { file1 = argv[1]; if (file2 != NULL) { /*FILE2指定有り*/ while ((ch = fgetc(fp1)) != EOF) { if (argc >= 3) { /*FILE2指定あり*/ if ((fp2 = fopen(file2, "w")) == NULL) { fputc(ch, fp2); file2 = argv[2]; printf("CANNOT OPEN %s\n", file2); exit(1); if (argc >= 4) { /*余計なもの指定あり*/ printf("fcpy [FILE1 [FILE2]]\n"); /*コピー実行*/ } この関数の呼び出し方: fcopy(fp1, fp2); fcopy(fp1, fp2); ■ fp1 と fp2 へのファイルオープンとかは後述 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 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 変数には予めデフォルト値を設定しておく 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 FILE *fp1 = stdin; /*デフォルトの読み出し元*/ int main(int argc, char *argv[]) { コマンドライン引数で指定があれば値を置き換える ... 略 ... if (エラー発生) { return 1; /* プログラムの終了コード=1 */ if (argc >= 2) { /*FILE1指定あり*/ file1 = argv[1]; プログラムの終了コード return 0: /* プログラムの終了コード=0 */ if (file1 != NULL) { /*FILE1指定あり*/ if ((fp1 = fopen(file1, "r")) == NULL) { 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 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 ... 略 ... fcopy(fp1, fp2); # C言語プログラムを実行 if [\$? -ne 0]; then # エラー発生を検出 # エラーメッセージを表示 echo "ERROR" exit 1 # 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 28 / 63 29 / 63 30 / 63

```
★重要★
                                                        終了コードを指定したプログラム終了方法:2通りあり
シェルスクリプト例
                                                            1 main 関数の return 値
   シェルスクリプト cla1-test-1.sh (エラー発生には強制終了)
                                                               int main(int argc, char *argv[]) {
  #!/bin/sh
  n=10
                                                                if (エラー発生) {
   while [ ${n} -le 100 ]
                                                                 return 1;
    echo "n=${n}"
                                                                                                                          (独自)複数のファイルによるプログ
                                                                .... 略 ....
    ./foo -n ${n} > out-${n}.txt
                                                                return 0:
    if [$?-ne 0]; then #エラー発生を検出
                                                                                                                           ラム構成法
      echo "ERROR"
                         # エラーメッセージを表示
                         #強制終了
      exit 1
                                                            2 exit 関数 (システムコール) の呼出
    n='expr ${n} + 10'
   done
                                                                ... 略 ...
if (エラー発生) {
   echo "done"
                                                                 exit(1);
                                                                }
  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」
                                                                ■ 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
                                                 31 / 63
                                                                                                          32 / 63
                                                                                                                                                                   33 / 63
                                                                                                                                                                 ★重要★
                                                                                                                 コンパイル (ライブラリ使用の場合)
                                                        数学ライブラリは頻繁に使うかも
                                                                                                                    コンパイル (失敗例; -lm オプションを忘れた)
                                                           数学関数 sin(), cos(), tan(), exp(), log(), ...
                                                                                                                    $ cc -o table-sin table-sin.c
                                                           C 言語プログラムでの利用方法 2 ステップ
                                                                                                                    /tmp/ccXN1Jlo.o: 関数 'main' 内:
                                                                                                                    table-sin.c:(.text+0x48): 'sin' に対する定義されていな
                                                            ■ ソースコード作成時: ヘッダのインクルード #include <math.h>
                                                                                                                    collect2: error: ld returned 1 exit status
                                                            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 2\pi まで 12 ステップ)
         ライブラリの利用
                                                                                                                    コンパイル (成功例)
                                                           #include <stdio.h>
                                                           #include <math.h>
                                                                                                                    $ cc -o table-sin table-sin.c -lm
                                                           #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:
                                                 34 / 63
                                                                                                          35 / 63
                                                                                                                                                                   36 / 63
```



C コンパイラ cc のコマンドラインオプション	★重要★ エラーメッセージを読み解く: 起きていること (1/2)	★重要★ エラーメッセージを読み解く:起きていること (2/2)
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 -1LIB 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	(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 Sin()の定義が無いからできまへん cc -c cc -o リンカ ld Table-sin.c /tmp/ccXN1Jlo.o ** table-sin Sin()を呼出 sin()を呼出 まin()を呼出	(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 -c
cc コマンド:ツールチェイン		例題 sincos : sin と cos の値の表
cc コマンド: コンパイルのためのプログラム群を駆動 ■ ツールチェイン (tool chain) の一種 ■ 関連ツールを次々と呼出 ツールチェインの利点 ■ 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) ■ 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 コードヘアセンブル ③ 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	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 c c - c sincos-table.c c c - c sincos sincos-main.o sincos-table.o - 1m 3行目: .c (C ソース) でなく.o (PIC ファイル) に注意
46 / 63	47 / 63	48/63

sincos 実行例: 0 度から 360 度まで. 30 度ごと ファイル1: sincos-main.c ファイル2: sincos-table.c. 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) #include <stdio.h> #include <stdio.h> #include <stdlib.h> #include <math.h> \$./sincos 0 360 30 void sincos_table(int d1, int d2, int dstep); 0 0.000000 1.000000 void sincos_table(int d1, int d2, int dstep) { 30 0.500000 0.866025 int main(int argc, char **argv) { int d1 = 0, d2 = 0, ds = 0; for (int d = d1; d <= d2; d += dstep) { 0.866025 0.500000 double th = M_PI * d / 180.0; 90 1.000000 0.000000 if (argc < 4) { printf("%3d %9.6f %9.6f\n", d, sin(th), cos(th)); 120 0.866025 -0.500000 fprintf(stderr, 150 0.500000 -0.866025 "ERR: 開始角度 終了角度 角度ステップ\n"); 180 0.000000 -1.000000 exit(1): 210 -0.500000 -0.866025 240 -0.866025 -0.500000 ■ 角度 (0 から 360) をラジアン (0 から 2π) に変換 d1 = atoi(argv[1]); 270 -1.000000 -0.000000 d2 = atoi(argv[2]); ■ M_PI (マクロによる記号定数) は円周率 π の値に定義 300 -0.866025 0.500000 ds = atoi(argv[3]); 330 -0 500000 0 866025 sincos_table(d1, d2, ds); 360 -0.000000 1.000000 return 0; 左から, 角度, sin, cos の値 49 / 63 50 / 63 51/63 ★重要★ 分割コンパイルの利点 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 デバッグ時の再コンパイル作業の時間短縮 まずは遅くていいから正しく動くプログラムを書こう 本当に必要な作業だけすれば良い ■ 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 ■ sincos-table.c を OK (独自)パフォーマンス測定と改善法 高速化を考えるのはそれから ■ sincos-main.c を修正 ■ 以下の作業で OK パフォーマンス測定を行ってみる (sincos-table.c の再コンパイルは不要) ■ そもそも実行時間は本当に遅いのか? \$ cc -c sincos-main.c ■ 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? \$ 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 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 ■ 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 52 / 63 53 / 63 54 / 63

	time コマンドによる実行時間の測定	time コマンドの応用例
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	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
55 / 63	56 / 63	57/63
	プロファイラとは	プロファイラ (gporf) 実際の出力例
プロファイラによるボトルネックの発見	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	Seconds Seconds Self Self
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	
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	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 司数が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 司数が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 司数が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
61/63	62 / 63	63 / 63