【C言語】入門

環境構築

■環境構築（Ubuntu）

- ▶ ☆ gccコンパイラをインストール

■環境構築（全般）

- ▶ ファをコンパイル＆実行

基礎文法編

■初歩的注意

- ▶ ※ 大文字と小文字を区別する言語である。
- ▶ ※ `{ }` で囲まれる範囲のことを「ブロック」という。

■データ型

- 整数　int　long　long long　※前に `unsigned` をつければ0以上の数だけ
- 実数　float　double
- 文字　char
- 値なし　void　※特別な型

■用語

- ▶ ブロック
- ▶ グローバル変数
- ▶ ローカル変数
- ▶ 静的変数
- ▶ スコープ

■基礎

- ▶ コメントのしかた
- ▶ ☆ テンプレ（メイン関数）
- ▶ 変数を定義
- ▶ 静的変数を定義

【C言語】入門

環境構築

■環境構築（Ubuntu）

- ▶ ☆ gccコンパイラをインストール

■環境構築（全般）

- ▶ ファをコンパイル＆実行　$ cd *dir*　$ gcc *./cFileName*　$ ./a.out

基礎文法編

■初歩的注意

- ▶ ※ 大文字と小文字を区別する言語である。
- ▶ ※ `{ }` で囲まれる範囲のことを「ブロック」という。

■データ型

- 整数　int　long　long long　※前に `unsigned` をつければ0以上の数だけ
- 実数　float　double
- 文字　char
- 値なし　void　※特別な型

■用語

- ▶ ブロック　{ } で囲まれる範囲
- ▶ グローバル変数　どのブロックにも属さず、全ブロックの外側で定義される共通の変数
- ▶ ローカル変数　ブロックの中で定義される変数。そのブロック内でしか参照できない
- ▶ 静的変数　ブロックを抜けても値を保持するようなローカル変数
- ▶ スコープ　変数の寿命や有効範囲などの総称

■基礎

- ▶ コメントのしかた　/* */ で囲めば改行可能
- ▶ ☆ テンプレ（メイン関数）
- ▶ 変数を定義　型 a = 値;　型 a;　型 a = 値, b = 値;　型 a, b;
- ▶ 静的変数を定義　ブロック内で　static 型 a = 値;　や　static 型 a;

- ※ グローバル変数、静的変数は自動的に初期化される（勝手に `0` といった初期値が代入される）一方で、静的変数を除く**ローカル変数は自動的に初期化されない**ので注意。
- ※ グローバル変数と同名のローカル変数を定義した場合、ローカル変数が優先される。（これにより関数の独立性が維持される）
- レジスタ変数を定義
- 関数内の定数を定義
- 整数値の定数を定義
- 実行中のCファの名前
- 実行中の関数の名前

■プリプロセッサ

- ※ C言語では、コンパイルの前に cpp というプリプロセッサによる前処理を行う。なお、文の最後に `;` は不要であることに注意。
- ヘッダファイル読込み
- 自作の〃読込み
- 定数を定義
- マクロを定義
- 定数やマクロを削除

■標準入出力

- １文字を入力
- 文字列を入力
- １文字を出力
- 文字列を出力
- 変数展開して出力

■条件分岐

- 条件分岐
- 単純なif文の略記
- ※ C言語では標準だとbool型が存在せず、 `0` 以外の数値を真、 `0` を偽と呼んでいる。
- 条件演算子
- 論理演算子

- ※ グローバル変数、静的変数は自動的に初期化される（勝手に `0` といった初期値が代入される）一方で、静的変数を除く**ローカル変数は自動的に初期化されない**ので注意。
- ※ グローバル変数と同名のローカル変数を定義した場合、ローカル変数が優先される。（これにより関数の独立性が維持される）

- レジスタ変数を定義　register 型 a = 値;　register 型名 a;
- 関数内の定数を定義　const HOGE = 値;　※一応ソースファ冒頭に書いても使える
- 整数値の定数を定義　enum {A, B, C,}　や　enum {A, B = 4, C,}　※0,1,2 や 0,4,5となる
- 実行中のCファの名前　__FILE__　※ `__LINE__` でその行数を知れる
- 実行中の関数の名前　__func__

■プリプロセッサ

- ※ C言語では、コンパイルの前に cpp というプリプロセッサによる前処理を行う。なお、文の最後に `;` は不要であることに注意。

- ヘッダファイル読込み　#include <*header*.h>
- 自作の〃読込み　#include "*header*.h"　※真っ先にカレディを探すようになる
- 定数を定義　#define HOGE 値
- マクロを定義　#define HOGE(p1, p2) コード様の文字の羅列
- 定数やマクロを削除　#undef HOGE

■標準入出力

- １文字を入力　**int** c;　c = **getchar**();　※: 文字の数値; `EOF` (= `-1`)
- 文字列を入力　char str[十分な字数];　**gets**(str);　※: なし
- １文字を出力　putchar(c※[1]);　※[1] `'a'` 等もアリ　※: 文字の数値; `EOF` (= `-1`)
- 文字列を出力
  - ・自動で改行：　puts(s※[1]);　※[1] `"Hello"` 等もアリ　※: なし
  - ・改行しない：　printf(s※[2]);　※[2]〃だが `%` を含めるな
- 変数展開して出力　printf("書式文字列", 変数1, 変数2, ...);　※: 文字数; `EOF` (= `-1`)

■条件分岐

- 条件分岐　if (条件$_1$) { 処理$_1$; } **else if** (条件$_2$) { 処理$_2$; } else { 処理$_3$; }
- 単純なif文の略記　if (条件) 1行の処理;

- ※ C言語では標準だとbool型が存在せず、 `0` 以外の数値を真、 `0` を偽と呼んでいる。

- 条件演算子　<　<=　>　>=　==　!=
- 論理演算子　&&　||　!　!( )

▶ ※ 条件演算子や論理演算子によって作られた条件式は最終的に `1` か `0` という数値として返る。

▶ ２股分岐の略記

▶ switch文

■繰り返し処理

▶ *n*回処理を繰り返す

▶ while文

▶ do-while文

▶ 中断し、次へ・脱出

■その他の制御

▶ ラベルへジャンプ

■関数

▶ 関数を定義

▶ 仮引数がない場合

▶ 返り値がない場合

▶ 関数の呼び出し

▶ ※ 関数もブロックをつくるので、関数内で定義した変数はその関数内でしか使えない。

▶ ☆ 引数を参照渡しする

■例外処理

▶ 強制終了

■配列

▶ ※ 配列は、**同じ型**の変数の集まり（順番あり）である。

▶ 配列を宣言

▶ 宣言と同時に初期化

▶ 要素の値を参照

▶ 要素数

▶ 配列をコピー

▶ 多次元配列

▶ ※ 条件演算子や論理演算子によって作られた条件式は最終的に `1` か `0` という数値として返る。

▶ ２股分岐の略記　条件 ? 真での値 : 偽での値

▶ switch文　switch (変数) { case **整数値**: 処理$_1$; **break;**　…　default: 処理$_o$; }

■繰り返し処理

▶ *n*回処理を繰り返す　int i;　for (i = 1; i <= *n*; i++) { 処理; }

▶ while文　while (条件) { 処理;　**条件に関する処理**; }

▶ do-while文　do { 処理;　**条件処理**; } while (条件)**;**　※一度は必ず実行

▶ 中断し、次へ・脱出　continue;　・　break;

■その他の制御

▶ ラベルへジャンプ　goto *label*;　※ `label:` でどこかにラベルを設けておく

■関数

▶ 関数を定義　返り値の型 hogehoge(型1 p1※, …);　‥　※省略可
返り値の型 hogehoge(型1 p1, …)　{ 処理;　return 返り値; }

▶ 仮引数がない場合　返り値の型 hoge(void);　返り値の型 hoge(void){ 〃 }

▶ 返り値がない場合　void hoge(‥);　void hoge(‥){ ‥ };

▶ 関数の呼び出し　関数(*arg1*, …)　※ 末尾に `;` が必要なことも当然ある

▶ ※ 関数もブロックをつくるので、関数内で定義した変数はその関数内でしか使えない。

▶ ☆ 引数を参照渡しする

■例外処理

▶ 強制終了　exit(0);

■配列

▶ ※ 配列は、**同じ型**の変数の集まり（順番あり）である。

▶ 配列を宣言　型名 a[*num*];　※ `num` は必ず整数値で**変数は使用不可**

▶ 宣言と同時に初期化　型名 a[*num*※[1]] = {値1, 値2, 値3, …};　※[1]省略可

▶ 要素の値を参照　a[*n*]

▶ 要素数　sizeof(a) / sizeof(a[0])

▶ 配列をコピー　#include <string.h>　memcpy(a2, a1, sizeof(a1));

▶ 多次元配列　型名 a[行数][列数];　などで定義
a[行番号][列番号]…　で要素の値にアクセス

■数値

- ▶ 算術演算子
- ▶ 複合代入演算子
- ▶ ｲﾝｸﾘﾒﾝﾄ,ﾃﾞｸﾘﾒﾝﾄ演算子
- ▶ m×10^n で表現
- ▶ 整数様文字列を整数に
- ▶ 実数様文字列を実数に

■文字

- ▶ ※ 文字は `''` でくくる。（いっぽう文字列は `""` でくくる）
- ▶ 特殊な文字を表現
- ▶ 文字を変数に格納

#include <ctype.h>

- ▶ 英数字かどうか
- ▶ 英字・数字かどうか
- ▶ 英大・小文字かどうか
- ▶ 記号かどうか

■文字列

- ▶ ※ 文字列は `""` でくくる。`""` で囲んだ文字列は特に**文字列リテラル**と呼ばれる。
- ▶ 文字列用の変数を宣言
- ▶ 宣言と同時に初期化
- ▶ 変数展開
- ▶ 文字列リテラルの連結

#include <string.h>

- ▶ 後方に連結
- ▶ 文字列のコピー
- ▶ 〃の前方だけコピー
- ▶ 文字数
- ▶ ２つの文字列が同じか

■数値

- ▶ 算術演算子　+　-　*　/　%
- ▶ 複合代入演算子　+=　-=　*=　/=　%=
- ▶ ｲﾝｸﾘﾒﾝﾄ,ﾃﾞｸﾘﾒﾝﾄ演算子　n++　++n　n--　--n
- ▶ m×10^n で表現　1.23e-4
- ▶ 整数様文字列を整数に　#include <stdlib.h>　atoi(s)　※ `s` には `+` `-` も使える
- ▶ 実数様文字列を実数に　〃　atof(s)　※〃

■文字

- ▶ ※ 文字は `''` でくくる。（いっぽう文字列は `""` でくくる）
- ▶ 特殊な文字を表現　\n　\t
- ▶ 文字を変数に格納　char c = 'A';　※ 全角文字は入れられない。

#include <ctype.h>

- ▶ 英数字かどうか　isalnum(c)
- ▶ 英字・数字かどうか　isalpha(c)　・　isdigit(c)　※ `isxdigit(c)` だと16進数
- ▶ 英大・小文字かどうか　isupper(c)　・　islower(c)
- ▶ 記号かどうか　ispunct(c)　※記号とは `!"#$%&'()*+,-/:;<=>?@^_`{|}~`

■文字列

- ▶ ※ 文字列は `""` でくくる。`""` で囲んだ文字列は特に**文字列リテラル**と呼ばれる。
- ▶ 文字列用の変数を宣言　char s[*num*];　※EOS (`\0`) 含めて `num` 文字だけ代入可能
- ▶ 宣言と同時に初期化　char s[*num*※[1]] = {'H', 'e', 'l', 'l', 'o', **'\0'**※[1]};　か
  char s[*num*※[1]] = "Hello";　※[1]省略可
- ▶ 変数展開　sprintf(s, "書式文字列", s1, s2, ...);
- ▶ 文字列リテラルの連結　"He" "llo"

#include <string.h>

- ▶ 後方に連結　strcat(s, s2※[1]);　※[1]文字列リテラルも可
- ▶ 文字列のコピー　strcpy(s, s2※[1]);　※[1] 〃　⇒**文字列リテラルでの代入**が可能に
- ▶ 〃の前方だけコピー　strcpy(s, s2※[1], *n*);　**s[*n*] = '\0';**　※[1] 〃
- ▶ 文字数　strlen(s※[1]);　※[1] 〃　※EOF分を除いた文字数が返る
- ▶ ２つの文字列が同じか　strcmp(s1※[1], s2※[1]) == 0　※[1] 〃

■構造体

- ▶ ※ 構造体は、**型が異なるものも含めて**、複数の変数を１パッケージ（順番あり）として扱うものである。
- ▶ 構造体を宣言
- ▶ 構造体変数を宣言
- ▶

■ファイル操作

- ▶ 開く
- ▶ １文字読み取る
- ▶ １行読み取る
- ▶ １文字上書きか追記
- ▶ 文字列を上書きか追記
- ▶ 閉じる

■ポインタ

- ▶ ポインタ変数とは
- ▶ ※ ポインタがもつデータは、先頭アドレスと、記憶領域の大きさ。
- ▶ ポインタ変数を宣言
- ▶ 変数のアド
- ▶ アドが指す変数の値

■構造体

- ▶ ※ 構造体は、**型が異なるものも含めて**、複数の変数を１パッケージ（順番あり）として扱うものである。
- ▶ 構造体を宣言　struct 構造体タグ名 {　ﾒﾝﾊﾞの型名1 ﾒﾝﾊﾞ名1;
  ﾒﾝﾊﾞの型名2 ﾒﾝﾊﾞ名2[要素数];　…　};
- ▶ 構造体変数を宣言　　struct 構造体タグ hoge;
- ▶

  変数.メンバ1 = 値1;　変数.ﾒﾝﾊﾞ2 = { 値2_1, 値2_2, … };　…

■ファイル操作

- ▶ 開く　　FILE *fp;　fp = **fopen**("～.txt", "*mode*");　　※: ストリーム; `NULL`
- ▶ １文字読み取る　　**int** c;　　c = **getc**(fp);　　※: 文字の数値; `EOF` (= `-1`)
- ▶ １行読み取る　　**fgets**(代入先の配列, 最大字数, fp);　　※: 配列; `NULL`
- ▶ １文字上書きか追記　　**int** c;　　c = 'a';　　**putc**(c, fp);　　※: 非負数; `EOF` (= `-1`)
- ▶ 文字列を上書きか追記　　char str[] = "abc";　　**fputs**(str, fp);　　※: 非負数; `EOF` (= `-1`)
- ▶ 閉じる　　**fclose**(fp);　　※: `0`; `EOF` (= `-1`)

■ポインタ

- ▶ ポインタ変数とは　　値としてアドレスをもつ変数
- ▶ ※ ポインタがもつデータは、先頭アドレスと、記憶領域の大きさ。
- ▶ ポインタ変数を宣言　　型※1 *変数名;　※1どのような型のデータへのポインタであるか
- ▶ 変数のアド　　普通の変数・配列の要素：　&hoge　・　&a[*n*]
  配列変数：　a　※ `&` 不要！そして `&a[0]` と同じ値。
  ※つまり、**単なる配列名**は最初の要素の**アドレス**を表している
- ▶ アドが指す変数の値　　*アド