資料

サークル有志

June 26, 2024

はじめに

これからC言語を学ぶ皆さんの健闘を祈る。誤字脱字などがあれば知らせてくれると助かります。わからないところがあってもとりあえず流し先に進むこと。いつかわかるときが来る。

目次 I

目次 II

目次 III

目次 IV

環境構築

wsl を用いた環境作りを行う

2 目次

3 環境構築

- ubuntu コマンド
- vim の使い方

wslのインストール

ここではwsl(windows subsystem for linux)を用いた，ubuntuのインストールの例を示す．（wslについて知っている人と一緒にやることを推奨する）

1. Windows PowerShellを起動する．
2. 以下を打ち込む．

Windows PowerShell

```
1    wsl --update
2    wsl --install
```

3. インストールが終了したら，画面下部のWindowsのタスクバーに"ubuntu"と打ち込み、ubuntuを起動する．
4. 初期設定などなど．誰かに聞いたらＯＫ（書きつくすにはスペースが少なすぎた）

コマンドなどなど

ubuntu は (Linux ディストリビューションの一つで,) OS(Operating System) なので，C 言語を動かすためだけに用いるにはあまりにも十分すぎる環境である． 関係のない機能が多すぎて逆に使いにくいと感じるかもしれない． しかし，今後 Python や GIT、ROS を扱う際にも使えるので、今のうちに慣れておくとよいだろう．閉じたいときは右上×ボタンから閉じられる。

表: 基本的なコマンド

コマンド	説明
ls	ディレクトリ及びファイルの表示
cd [DirectoryName]	ディレクトリの移動
mkdir [DirectoryName]	ディレクトリの作成
rmdir [DirectoryName]	ディレクトリの削除
touch [FileName]	ファイルの作成
rm [FileName]	ファイルの削除
explorer.exe .	エクスプローラーで開く

C 言語の環境構築

ubuntu で以下を打ち込む

ubuntu

```
1    sudo apt update
2    sudo apt install gcc
```

これで完了. 下に使い方を示す。

ubuntu

```
1    gcc hoge.c // g++ hoge.cpp
2    ./a.out
```

1 行目で hoge.c のコンパイルを行っており, コンパイルが完了すると a.out というファイルが生成される (math.h を使うときは末尾に-lm とつけてコンパイルする).2 行目で, 先ほど生成された a.out を実行している。ファイルを実行して, なかなか終了しないときは Ctrl+C でコードを強制終了できる.

テキストエディタの使い方

ubuntu にデフォルトでインストールされている vim を紹介する.

表: vim の使い方

ubuntu	
vi [FileName]	ファイルを作成し開く
Normal mode	
i	Insert mode へ (文字を入力できる)
:	Command-line mode へ
Command-line mode	
w	変更の保存
q	vim の終了
wq	上書き保存して終了する
q!	変更を破棄し終了する

他にもいろいろあるので, 興味があれば調べるのが良い. (vimtutor など)

C 言語とは

1972 年にデニス・リッチーによって開発された汎用プログラミング言語． C 言語は他の多くのプログラミング言語の基礎となっており，そのシンプルさと効率性が特徴である．

Hello, World!

とりあえず C 言語を動かしてみる

hello.c

```
1 #include<stdio.h>
2
3 int main(void)
4 {
5     printf("Hello, World!\n");
6     return 0;
7 }
```

出力

```
Hello world
```

解説

今回記述した内容は，今後のコード作成でも必ず記述することになるものなので，始めは理解するより丸暗記することをお勧めする．

- ’#include<stdio.h>’: 標準入出力ライブラリをインクルードする．
- ’int main(void)’: プログラムのエントリポイント． main 関数はプログラムの開始地点となる．
- ’printf("Hello,*\ n");’: 画面に"Hello, World!"を表示する\ n は改行文字．
- ’return 0;’: 関数の終了を示す．0 は正常終了を意味する。

コメントの挿入

- //: 以降から行末までがコメントになる．
- /* */:/*と*/で囲まれた部分がコメントになる． 複数行にまたげる．

変数と演算

2 目次

6 変数と演算
- 変数
- 演算
- 代入演算
- 更新処理
- 論理条件演算子
- 関係演算子
- ビット演算子

変数とデータ型

]

variable.c

```
1  #include<stdio.h>
2
3  int main(void)
4  {
5      int a = 10; //integer
6      float b = 3.14; //floating point
7      double c = 2.71828; //double precision floating point
8      char d = 'A'; //character
9
10     printf("a = %d\n", a);
11     printf("b = %f\n", b);
12     printf("c = %e\n", c);
13     printf("d = %c\n", d);
14
15     return 0;
16 }
```

解説

出力

```
10
3.14
2.71828
A
```

変数を使うには，変数の型を宣言する必要がある．
printf 関数を用いて変数の値を表示する時は，フォーマット指定子を用いる．

表：フォーマット指定子の例

%d	整数を 10 進数で出力
%f	実数を出力
%e	実数を指数表記で出力
%c	１文字を出力
%s	文字列を出力

演算 (その 1)

加算は+, 減算は-、乗算は*、除算は/である .int 型同士の商は少数切り捨ての整数となる . また, 剰余は%で求める。

```
9 + 2
```

```
11
```

```
9 - 2
```

```
7
```

```
9 * 2
```

```
18
```

```
9 / 2
```

```
4
```

```
9 % 2
```

```
1
```

演算 (その 2)

累乗は math.h を追加でインクルードし,pow を使って求める.

```
pow(9,2)
```

```
81
```

括弧を使うことで演算子の優先順位を変更できる.

```
(float(9) + float(2)) / 3
```

```
3.666667
```

$a \times 10^b$ のような指数表記は次のようにする.

```
double a = 1e-4
```

```
1.000000e-04
```

キャストト演算子を用いることで, 一時的に型を強制変換できる.

```
(double)a / b //a,b は int 型で宣言してある
```

```
double 型の計算結果を得る
```

代入演算

=を用いて代入を行う.C 言語で (左辺)=(右辺) という式は，(左辺) に (右辺) の値を代入する，という処理になる.

```
x=3
```

```
3
```

```
x = x + 1
```

```
4
```

数学で使われる＝とは意味が異なるものとなっている.
また,$x = x \pm 1$ という更新処理は次のように書ける.

```
x++
```

```
5      //インクリメントと言う
```

```
x--
```

```
4      //デクリメントと言う
```

更新処理

変数 x と変数 i の和差積商を x とする更新処理の場合，次のように書く．

```
x = x + i
```

```
x += i
```

```
x = x - i
```

```
x -= i
```

```
x = x * i
```

```
x *= i
```

```
x = x / i
```

```
x /= i
```

論理条件演算子 (その 1)

- 等値演算子:(左辺)==(右辺)
  両辺が等しい場合１, 等しくない場合０を返す.

```
int a = (2==2)
```

```
a = 1
```

- 非等値演算子:(左辺)!=(右辺)
  両辺が等しい場合０, 等しくない場合０を返す.

```
int a = (2!=2)
```

```
a = 0
```

論理条件演算子 (その 2)

- and 演算子:(左辺)&&(右辺)
  両辺が真の場合 1, それ以外の場合 0 を返す.

```
int a =((2 == 2)&&(3 == 3))
```

```
a = 1
```

- or 演算子:(左辺)||(右辺)
  両辺が偽の場合 0 , それ以外の場合 1 を返す.

```
int a =((2 == 2)||(3 == 3))
```

```
a = 1
```

- 否定演算子:!(条件式)
  条件式の真偽を反転させる.

```
int a = !(2==2)
```

```
a = 0
```

関係演算子

>, <, >=, <= のように記述する.
真の場合 1, 偽の場合 0 を返す.C 言語では 0 以外の数値は, 真偽値においては真と解釈される.

tips

- 条件演算子: (式 1)?(式 2):(式 3)
  (式 1) の真偽によって,(式 2) か (式 3) の値を返す.
  ((式 1) が真の場合 (式 2) の値を返す)

```
a=((2==3)?10:20)
```

```
a = 20
```

- コンマ演算子:(式 1),(式 2)
  まず (式 1) を評価し, 式としては (式 2) の値を返す.

```
x = (a = 3, a + 2)
```

```
x = 5
```

ビット演算子 (その 1)

- ビット毎の論理積:(式 1)&(式 2)
  各桁毎の論理積を返す.

```
a = 29&17
```

a = 17　　//$29 = 11101_{(2)}, 17 = 10001_{(2)}$

- ビット毎の排他的論理和:(式 1)^(式 2)
  各桁毎の排他的論理和を返す.

```
a = 29 ^17
```

a = 12　　//$29 = 11101_{(2)}, 17 = 10001_{(2)}$

- ビット毎の論理和:(式 1) | (式 2)
  各桁毎の論理和を返す.

```
a = 29 | 17
```

a = 29　　//$29 = 11101_{(2)}, 17 = 10001_{(2)}$

ビット演算子 (その 2)

- ビット反転演算子:~(式)
  (式) の全ビットを反転させる.

```
a = ~29
-30        //8bit 分計算されている
```

- シフト演算子:(式 1)<<(式 2)
  (式 1) の値を (式 2) のビット数だけシフトさせる.

```
a = 29<<2
a = 116        //左シフトは (式 1) を 2^n 倍した値となる
```

同様にして右シフトもできる.

```
a = 29>>2
a = 7        //右シフトは (式 1) を 2^-n 倍した整数値となる
```

演習問題

実行制御

2 目次

7 実行制御
- for 文
- while 文
- if 文
- if else if 文
- continue/break/goto
- switch 文

for 文 (反復処理)

繰り返しの回数が決まっている反復処理には for 文が使われる .

Syntax of for loop

```
1  for (initialization; condition; updation) {
2      //body of the for loop
3  }
```

initialization	for 文に入るとき 1 度だけ実行される . 繰り返しをカウントする変数とその初期値を割り当て 例: int i = 0
condition	式の値が評価され, 真の場合 for 文内の処理を実行す 偽の場合 for 文処理を実行せず抜け出す . 例: i<10000
updation	for 文内の処理の後,initialization で割り当てた変 インクリメントやデクリメント , またはその他の処理 記述する . 例: i++

サンプルコード

pra_ for.c

```
1  #include<stdio.h>
2
3  int main(void)
4  {
5      for (int i = 0; i < 10; i++) {
6          printf("Hello World%d\n", i);
7      }
8
9      return 0;
10 }
```

出力

```
Hello World0
...
Hello World9 //10 個表示されている
```

演習問題

while 文 (条件付き反復)

繰り返しの条件が決まっている反復処理に使われる.

Syntax of while loop

```
1  initialization
2
3  while (condition) {
4      //body of the while loop
5  }
```

condition	式の値が評価され, 真の場合 while 文内の処理を実行する. 偽の場合 while 文内の処理を実行せず while 文を抜け出す

tips

- 繰り返しの処理内で,inisializetion に該当する変数の値が変わらないとき for 文を使い, 変わるとき while 文を使うとよい.
- condition を常に真としたい時,condition を 1 とすればよい.
- 繰り返しが無限に続かないように注意する.

サンプルコード

pra_while.c

```
1  #include<stdio.h>
2
3  int main(void)
4  {
5      double x = 199;
6
7      while (x > 1) {
8          x /= 9;
9      }
10
11     printf("%f", x);
12
13     return 0;
14 }
```

出力

0.272977

do while 文

始めに文内の処理を行い, その後 while 文と同じ挙動をする.
必ず一回は文内の処理を行いたいときに使う.

Syntax of do while loop

```
1   do {
2       // body of do-while loop
3   } while (condition);
```

「while 文ではループに入る条件を決め,do while 文ではループから出る条件を決める.」と区別できるが, どちらを使ってももコードがほとんど変わらないときが多く, そのようなときは while 文を使うのがよい.

演習問題

if 文 (条件分岐)

ある条件を満たしたときに, 行いたい処理があるときに使う .

Syntax of if

```
1   if (condition) {
2       //if body
3   }
```

condition	式の値が真の場合,if 文内の処理が行われる . 偽の場合,if 文は無視され続く文が実行される .

条件が満たされない場合に行いたい処理がある場合は else 節を使う .

Syntax of if-else

```
1   if (condition) {
2       //code executed when the condition is true
3   }
4   else {
5       //code executed when the condition is false
6   }
```

サンプルコード

pra_if.c

```
1  #include<stdio.h>
2  #include<math.h>
3
4  int main(void)
5  {
6      double x = pow(exp(1), M_PI);
7      double y = pow(M_PI, exp(1));
8
9      if (x > y) {
10         printf("e^pi is larger.\n");
11     }
12     else {
13         printf("pi^e is larger.\n");
14     }
15
16     return 0;
17 }
```

サンプルコード

出力

```
e^pi is larger.
```

上のコードは x = y の場合を考慮していないため，あまり良いコードではない. 例ということで許してほしい。

tips

math.h をインクルードするときは gcc hoge.c -lm を用いる.

if else if 文

if 文を連続して用いる際は次のように記述する.

Syntax of if-else-if

```
1  if (condition) {
2
3  } else if (codition) {
4
5  }
6  ... // once if-else ladder can have multiple else if
7  } else { // at the end we put else
8
9  }
```

continue/break/goto

ループの処理を行うときに、知っておくとと便利

`continue;`	繰り返しループの途中で強制的に次の繰り返しに進める.
`break;`	実行中のループを中断し, ループから抜け出す.
`goto 名前;`	名前と同じ名札 (ラベル) を付けた文に実行を移す.

Syntax of goto statement

```
1  goto label;
2  .
3  .
4  .
5  label:
6      //statements
```

goto 文は便利なように見えるが, 何重ものネストから一気に抜け出す場合にのみ用いるべきである.

switch 文

使ったことがないためよくわからない

Syntax of switch statement

```
switch (expression) {
case value1:
    statement_1;
    break;
case value2:
    statement_2;
    break;
.
.
.
case value_n:
    statement_n;
    break;
default:
    default statement;
    break;
}
```

サンプルコード

pra_switch.c

```
1  #include<stdio.h>
2
3  int main(void)
4  {
5      int a = 3;
6      int count_0, count_1, count_2 = 0;
7
8      for (int i = 1; i < 101; i++) {
9          switch (i%a) {
10         case 0:
11             count_0++;
12             break;
13         case 1:
14             count_1++;
15             break;
```

サンプルコード

pra_switch.c

```
16          case 2:
17              count_2++;
18              break;
19          }
20      }
21
22      printf("remainder=0\t%d\n", count_0);
23      printf("remainder=1\t%d\n", count_1);
24      printf("remainder=2\t%d\n", count_2);
25      return 0;
26  }
```

サンプルコード

出力

```
remainder=0     33
remainder=1     34
remainder=2     33
```

同様のコードを `if else if` 文を用いて書くことができるため，あまり使われない．

演習問題

関数

2 目次

8 関数
- 関数の基本
- 関数の作成

関数の基本

すでに出てきた関数として,main 関数や printf 関数などがある.
関数を使うとき, その返り値や引数を知る必要がある.

表: 関数を呼び出すときの構造

VariableNumber = FunctionName (Parameter_1, Parameter_2, ...)	
引数 Parameter	関数を呼び出す時, その関数に渡す値. main 関数のように引数を持たない関数もある. 例: double a = sin(x); // 引数は x
返り値 returning value	関数が返す値. 返り値がない関数もある. 例: double a = sin(x); // 返り値は a に代入される

tips

- printf 関数は出力した文字数を返り値としている.
- main 関数の返り値は, ubuntu の場合 echo $?で確認できる.

関数の作成

プロトタイプ宣言

自作の関数を作るには，まず関数を宣言する． その後，関数を定義する．

Syntax of function declaration

```
1 return_type name_of_the_function (para_1, para_2, ...);
```

return_type	返り値の型を宣言する．
name_of_the_function	関数の名前を宣言する．
para_n	パラメータの型を宣言する．

Example of function declaration

```
1 int sum(int a, int b); //Declaration with parameter names
2 int sum(int, int); //Declaration without parameter names
```

これらはプロトタイプ宣言と呼ばれ，関数を呼び出す前に記述しなければならない．(main 文の上に書く)

関数の作成

関数の定義

Syntax of function definition

```
1 return_type function_name (para1_type para1_name,
      para2_type para2_name, ...)
2 {
3     //body of the function
4 }
```

関数の定義は関数の宣言を含むため, 関数の呼び出しを行う前 (通常 main 関数の前) に関数の定義をすればプロトタイプ宣言を行わなくてよい. しかし,main 関数の上に関数の定義を置くと、コードの見やすさが損なわれるため, プロトタイプ宣言→ main 関数→関数の定義の順でコードを書くことをお勧めする.

関数の定義

補足

Example_1

```
1  #include<stdio.h>
2
3  void hoge(int *para)
4  {
5      //body of the hoge
           function
6  }
7
8  int main(void)
9  {
10     //body of the main
           funcion
11 }
```

Example_2

```
1  #include<stdio.h>
2
3  void hoge(int *);
4
5  int main(void)
6  {
7      //body of the main
           function
8  }
9
10 void hoge(int *para)
11 {
12     //body of the hoge
           function
13 }
```

右の書き方であれば, 自作関数をいくつ定義しても main 関数が必ず一番初めに定義されるので, 見やすくなる.

サンプルコード

ベクトルのノルムを計算する関数を作成する.

pra_func.c

```
1  #include<stdio.h>
2  #include<stdarg.h>
3  #include<math.h>
4
5  double Norm(int,...); //function declaration
6
7  int main(void)
8  {
9      double norm1, norm2 = 0;
10
11     norm1 = Norm(3, 2., 3., 4.); //call function
12     norm2 = Norm(3, 3., 4., 5.);
13
14     printf("norm1=%e\n", norm1);
15     printf("norm2=%e\n", norm2);
16     return 0;
17 }
```

サンプルコード

pra_func.c

```
18 double Norm(int data,...) //function definition
19 {
20     va_list list; //va_list type declaration
21     double value, norm_squared = 0;
22     int i;
23
24     //printf("demension=%d\n", data);
25
26     if (data < 1) {
27         perror("data error");
28         return 1;
29     }
30
31     va_start(list, data); //initialize list
```

サンプルコード

pra_func.c

```
32      for (i = 0; i < data; i++) {
33          value = va_arg(list, double);
34          //printf("data%d=%f\n", i, value);
35          norm_squared += pow(value, 2);
36      }
37
38      va_end(list); //release list
39      return sqrt(norm_squared);
40  }
```

出力

```
norm1=5.385165e+00
norm2=7.071068e+00
```

このような処理には配列を使うのが一般的である．例ということで許してほしい．

変数のあれこれ

2 目次

9 変数のあれこれ
- ローカル変数
- グローバル変数
- 定数の宣言

ローカル変数とグローバル変数

変数は下のように大きく二つに大別できる.

- ローカル変数　　関数内で宣言された変数
- グローバル変数　　関数外で宣言された変数

これらはにはさらに静的変数かそうでないかという違いがある.
(変数宣言時に `static` をつけると静的変数となる.
例: `static int a = 3;`)

tips

- 寿命: 変数の存続時間
- スコープ: 変数の有効範囲

ローカル変数

関数内で宣言された変数はローカル変数となる．

	寿命	スコープ	暗黙の初期値
ローカル変数	宣言時から 関数の終了まで	宣言された関数内	不定
静的ローカル変数	宣言時からプログラム の終了まで	宣言された関数内	0

tips

ローカル変数はメモリのスタック領域に配置されるが，静的ローカル変数，グローバル変数、静的グローバル変数はメモリの静的領域に配置される．

グローバル変数

関数外で宣言された変数はグローバル変数となる．
main 関数の前に宣言されることが多い．

	寿命	スコープ	暗黙の初期値
グローバル変数	宣言時から プログラムの終了まで	プログラム内及び その他のファイル	0
静的グローバル変数	宣言時からプログラム の終了まで	プログラム内のみ (他ファイルからの 参照を許さない)	0

グローバル変数と同じ名前のローカル変数が宣言されたとき，その関数からはそのグローバル変数の参照ができなくなる．

tips

- 関数も static をつけて宣言できる．(他ファイルからの参照を許さくなる)
- 他ファイルで宣言されたグローバル変数や，関数を使うには extern をつけて宣言する．

定数の宣言

みかん

- const を使う

Syntax

```
1 const data_type cariable_name = initial_value;
```

- マクロを使う

Syntax

```
1 #define name value
```

- enum を使う

配列

2 目次

10 配列
- 配列の宣言
- 配列の要素
- 配列の初期化
- 実行例
- サンプルコード

配列の宣言

C 言語の配列とは同じ型の、複数の値を一つの名前で扱うことができるデータ構造の一つである。より踏み込んだ説明をするならば、メモリ上に同じ型の配列要素が連続して並んだ領域のことである。

Syntax of array declaration

```
1 data_type array_name[size];
2
3 // N-dimensional array
4 data_type array_name[size1] [size2] ... [sizeN];
```

data_type	配列の要素の型を宣言する.
array_name	配列には任意の名前を付けられる。
size	配列の要素数。定数で指定する。

配列の要素

配列の個々の要素へはの次のようにしてアクセスできる。配列の要素は変数と同じように扱える。

```
1 array_name[index];
```

インデックス index	0〜(配列のサイズ-1) までの整数型をとり、整数型であれば変数でもよい。

注意

C 言語ではコード内で使われるインデックスの値が、インデックスのとりうる値を超えているかの検査はコンパイル時も実行時も行われない。インデックスの値が 0〜(size-1) にあるかどうかには十分気を付けること。(配列外の値にアクセスできるが、エラーは出ない)

配列の初期化

配列の初期値は不定であるため、初期化を行う必要がある。大きさが10000程度の配列を宣言し、一度確認してみるとよいだろう。

Example of Array Initialization

```
1  // Initialization with Declaration
2  int a[3] = {1, 2, 3};
3  int b[10000] = {1, 0, 2, 0, 3}; // **Other Elements
4                                  // are Initialized to Zero
5
6  // Initialization with Declaration without Size
7  double c[] = {1, 2 ,3, 4, 5, 6};
8
9  // Initialization after Declaration (Using Loops)
10 int d[10000];
11 for (int i = 0; i < 10000; i++) {
12     d[i] = i;
13 }
```

** この方法で初期化したとき、環境によっては他の要素が不定となる場合がある。

実行例

int a[3]; //初期化なし

a[0]=不定, a[1]=不定, a[2]=不定

int b[3] = {1};

b[0]=1, b[1]=0, b[2]=0 //b[1],b[2] の値は環境により異なる. 私の環境では 0 となった。

int c[] = {1, 2, 3}; //サイズは宣言した要素の数となる

c[0]=1, c[1]=2, c[2]=3

printf("%d", c[3]); //配列外の要素を参照している。

1307256064 //配列外の要素であるが、問題なく出力される。バグの要因となる.

サンプルコード

配列は列ベクトルとみなすことができるため、今回はベクトルの合成を考える。

pra_array.c

```
1  #include<stdio.h>
2
3  int main(void)
4  {
5      double a[3] = {3, 5, 3};
6      double b[3] = {1, 5, 7};
7
8      double c[3];
9      for (int i = 0; i < 3; i++) {
10         c[i] = a[i] + b[i];
11         printf("c[%d]=%f\n", i, c[i]);
12     }
13
14     return 0;
15 }
```

サンプルコード

出力

```
c[0]=4.000000
c[1]=10.000000
c[2]=10.000000
```

tips

- sizeof 演算子
  変数に対して作用させることで、変数の占める領域のサイズがバイト単位で得られる。配列名に対して作用させることで、その配列が占める領域のサイズが得られる。
  例: `a = sizeof(int);`
  a には int 型のメモリサイズである 4 が代入される。

演習問題

メモリ

今まで何度か出てきたメモリの解説

- メモリ

  メモリとはデータを記憶するパーツのことで、読み書きのできるRAM(Random Access Memory)と読み取り専用のROM (Read Only Memory)の２種類がある。一般的にメモリと言った際にはRAMを指すことが多く、このスライドで何度か出るメモリもRAMを指している.(より正確に言うと仮想メモリを指している。)

- 仮想メモリ

  仮想メモリとはＯＳが物理的なメモリを、ソフトウェアなどのために専用の連続したメモリのように扱えるようにしたもの。

アドレス

- アドレス

  アドレスとはメモリ上の位置を表す値のことである。

  私たちが main 関数内で int 型の変数 x を宣言したとき、CPU は仮想メモリ内のスタック領域に、変数 x の値を格納するための 4[byte] 分の領域を確保する。

  この変数 x の値を参照するとき私たちは単に変数名の x を呼ぶだけでよい。では CPU はこの値をどのようにして参照しているのか。

  CPU は変数の値を格納しているメモリ領域に名前を付け、その名前に格納されている値を参照している。このときのメモリ領域の名前をアドレスと呼ぶ。

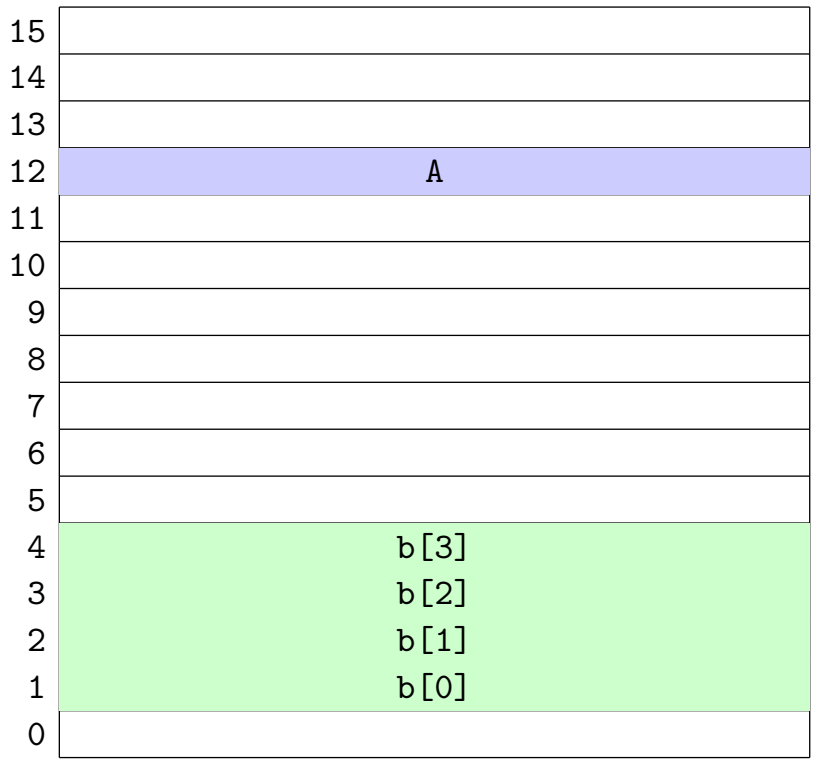
  領域の確保をしてアドレスを作るというような説明になったが、実際はアドレスが先に決まっており、そこに必要に応じて領域を確保している。

  アドレスは 1 バイト毎に割り振られている。

図にしてみた

例として、char a = 'A'; の宣言により、図 (メモリ) の青色の領域に値が格納されたとする。私たちはここに格納された値に a を用いてアクセスし、CPU は 12 を用いてアクセスする。

また、char b[4]; により、緑色の領域が確保されたとする。このとき b[0] を指すアドレスは 1,b[3] を指すアドレスは 4 となる。



メモリアドレスとデータ

(a) 左側の数値はアドレスを表す

(b) セル 1 つが 1[byte] を表す

ポインタ

C言語学習の山場（文字の説明ばかりとなってしまうため、わかりにくいと感じたら、図解や実行例を見るとよいだろう。）

ポインタの基本

- ポインタ
  変数のアドレスを値として格納できる変数をポインタという。
- ポインタの型
  ポインタには型があり、特定の型の変数を表すアドレスしか格納できない。
- ポインタの宣言
  変数名の前に*(ポインタ宣言子) をつければよい。

Example of Pointer Declaration

```
1 int *pint;
2 double *px, *py;
3 int i, *pi;
```

ポインタの初期化

ポインタとして宣言した変数にはアドレスを代入する。変数のアドレスは&(アドレス演算子) を変数に作用させることで得る。

Example of Pointer Initialization

```
1 int n, *np;
2 double d, *dp;
3
4 np = &n;
5 dp = &d;
```

アドレスの値が気になるときはフォーマット指定子を%p として表示させてみるとよいだろう。(16 進数表記で出力される。)

tips

ポインタには変数のアドレスが格納される。

ポインタによる間接アクセス

間接演算子*をポインタに作用させた式は、ポインタに格納されたアドレスの指す値となる。(同じ記号を使うがポインタ宣言子とは名前が異なる)

Pointer Example

```
1 int Var = 10;
2 int *Ptr;
3
4 Ptr = &Var;
5 printf("Var=%d, *Ptr=%d", Var, *Ptr);
6
7 *Ptr = 5;
8 printf("Var=%d, *Ptr=%d", Var, *Ptr);
```

出力

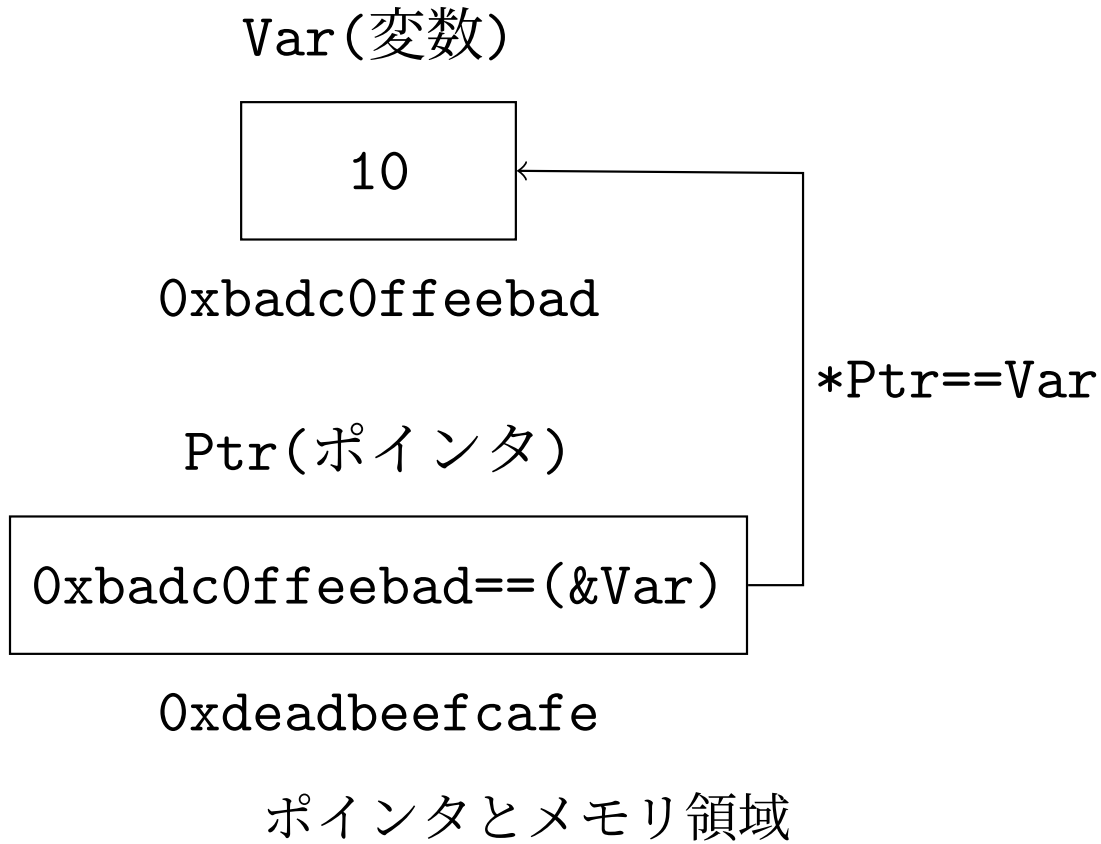
```
Var=10, *Ptr=10
Var=5, *Ptr=5
```

図解

左側にコード、右側にその時のメモリ領域の図を示す。

```
1 int Var = 10;
2 int *Ptr;
3
4 Ptr = &Var;
```

(右図でわかるように*Ptrと Var は等価であるため、*Ptr に対して値を代入することは Var に値を代入することと等しい。)



ポインタとメモリ領域

ポインタのポインタ

みかん

ポインタと配列

配列名は配列の最初の要素を表すアドレスとなっている。つまり,int a[10]; により配列を宣言した時、a と&a[0] は等価となる。このとき、a で初期化したポインタに整数値 n を足したポインタ ptr+n の値は、a[n] のアドレスと等価となる。

Exercise

```
1  int v[1000];
2  int *ptr;
3
4  ptr = v; // == ptr = &v[0];
5
6  for (int i = 0; i < 1000; i++) {
7      printf("----------------------\n");
8      printf("&v[%d]=\t%p\n", i, &v[i]);
9      printf("ptr=\t\t%p\n", ptr);
10     printf("v[%d]=\t\t%d\n", i, v[i]);
11     printf("*ptr=\t\t%d\n", *ptr);
12     ptr++;
13 }
```

出力の仕方

今回のように数千行を出力するときは、./a.out > output などとして実行し、ファイルに実行結果を出力するのが良い。

ポインタと関数

　ポインタを引数に取る関数を定義することができる。これにより、関数内で配列の値を参照できるようになる。(C 言語では配列をそのまま引数とすることはできない)

　また、C 言語の関数は返り値を 1 つしか持てないが、ポインタを使えば複数の値を返せる。

実行例

```
int v, *pv; //int 型変数、int 型ポインタの宣言
```

```
v = 3; //変数の初期化
pv = &v; //ポインタの初期化 (ポインタにはアドレスを代入する)
```

```
printf("v=%d, &v=%p\n", v, &v);
printf("pv=%p, &pv=%p\n", pv, &pv);
```

```
v=3, &v=0xcafecafecafe
pv=0xcafecafecafe, &pv=0xcafebeefcafe
```

変数名 v, アドレス 0xcafecafecafe には 3 が格納されている。
変数名 pv, アドレス 0xcafebeefcafe には 0xcafecafecafe が格納されている。

実行例

～ポインタによる間接アクセス～

*pv は,pv に格納されている 0xcafecafecafe の指すメモリ領域に格納されている 3 にアクセスする.

```
printf("*pv=%d", *pv); //ポインタによる間接アクセス
```

```
*pv=3 //v の値
```

```
v = 9
```

```
v=9, *pv=9
```

```
*pv = 2
```

```
v=2, *pv=2
```

実行例

～ポインタと配列～

```
int a[3] = {1, 2, 3};
```

pv = a //a[0] のアドレスを代入することと同じ

```
*pv, *(pv+1), *(pv+2) を出力
```

*pv=1 //a[0] の値
*(pv+2)=2 //a[1] の値
*(pv+3)=3 //a[2] の値

～ポインタと関数（配列を関数に渡す）～

```
//プロトタイプ宣言
void func(int size, int *array_name);
```

func(3, a) //関数呼び出し

このようにすることで、自作関数内で配列の値を参照できるようになる。

サンプルコード

2 つの変数に格納されている値を入れ替えるコード

pra_pointer1.c

```
1  #include<stdio.h>
2
3  void exchange(int *, int *);
4
5  int main(void)
6  {
7      int x = 3;
8      int y = 8;
9      printf("(x,y)=(%d,%d)\n", x, y);
10
11     exchange(&x, &y);
12
13     printf("(x,y)=(%d,%d)\n", x, y);
14     return 0;
15 }
```

サンプルコード

pra_pointer.c

```
16 void exchange(int *x, int *y)
17 {
18     int ver;
19
20     ver = *x;
21     *x = *y;
22     *y = ver;
23 }
```

出力

```
(x,y)=(3,8)
(x,y)=(8,3)
```

サンプルコード２

n 次元列ベクトルの和・スカラー倍・内積・ノルムを求める

pra_pointer.c

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

void sum(int, double*, double*, double*);
void scalar_multiple(double, int, double*);
double inner_product(int, double*, double*);
double norm(int, double*);

int main(void)
{
    double a[100];
    double b[100];

    for (int i = 0; i < 100; i++) {
        a[i] = i;
        b[i] = i;
    }
```

サンプルコード２

n 次元列ベクトルの和・スカラー倍・内積・ノルムを求める

pra_pointer.c

```
19      double c[100];
20      sum(100, c, a, b);
21
22      for (int i = 0; i < 100; i++) {
23          printf("c[%d]=%f\n", i, c[i]);
24      }
25
26      printf("------------------\n");
27      scalar_multiple(2, 100, c);
28
29      for (int i = 0; i < 100; i++) {
30          printf("c[%d]=%f\n", i, c[i]);
31      }
32
33      printf("------------------\n");
34      double ans;
35      ans = inner_product(100, a, b);
```

サンプルコード２

n 次元列ベクトルの和・スカラー倍・内積・ノルムを求める

pra_pointer.c

```
36      printf("ans=%f\n", ans);
37
38      printf("------------------\n");
39      ans = norm(100, a);
40      printf("ans=%f\n", ans);
41
42      return 0;
43  }
44
45  void sum(int size, double *ans, double *array1, double *
        array2)
46  {
47      for (int i = 0; i < size; i++) {
48          *(ans++) = *(array1++) + *(array2++);
49      }
50  }
```

サンプルコード２

n 次元列ベクトルの和・スカラー倍・内積・ノルムを求める

pra_pointer.c

```
52 void scalar_multiple(double scalar, int size, double *
       array)
53 {
54     for (int i = 0; i < size; i++) {
55         *(array++)*=scalar;
56     }
57 }
58
59 double inner_product(int size, double *array1, double *
       array2)
60 {
61     double answer = 0;
62
63     for (int i = 0; i < size; i++) {
64         answer += *(array1++) * *(array2++);
65     }
```

サンプルコード 2

n 次元列ベクトルの和・スカラー倍・内積・ノルムを求める

pra_pointer.c

```
67      return answer;
68  }
69
70  double norm(int size, double *array1)
71  {
72      double answer = 0;
73
74      for (int i = 0; i < size; i++) {
75          answer += *array1 * *(array1++);
76      }
77
78      return sqrt(answer);
79  }
```

出力 (一部)

ans=328350.000000 // inner_product の計算結果

文字列

文字列の基本

char 型配列を使えば文字列として扱える。文字コードは ASCII コードが使われている。