Today's contents

- File processing
  - File and data stream
  - Text and binary data
- Control flow
  - Selective control
  - Iterative control
  - Leap control
- Character string and pointer
  - Character array
  - Pointer
  - Array and pointer

今日の内容

- ファイル処理
  - ファイルとストリーム
  - テキストとバイナリ
- 制御構造
  - 選択制御
  - 反復制御
  - 跳躍制御
- 文字列とポインタ
  - 文字配列
  - ポインタ
  - 配列とポインタ

Open a file (1)

- Operations to use a file
- Firstly, define a pointer to “FILE” structure

```
/*
 * Open and close a file
 */
#include <stdio.h>

int main(void)
{
  FILE *fp;
```

← Pointer to “FILE” structure
“ fp” is a variable name (changeable).
Don’t forget attaching “*” !

ファイルを開く(1)

- ファイルを使えるようにするための操作
- まずはFILE構造体へのポインタを宣言

```
/*
 * ファイルのオープンとクローズ
 */
#include <stdio.h>

int main(void)
{
  FILE *fp;
```

← FILE構造体へのポインタ
fpは変数名なので変えられる
“＊”を忘れないように！

Open a file (2)

- Open a file by the function “fopen”
  - Arguments: file name, mode
    - “r”: read mode, “w”: write mode

```
   Open and close a file
*/
#include <stdio.h>

int main(void)
{
  FILE *fp;

  fp = fopen( “abc” ,  “r” ); /* Open a file */
```

↓Open a file named “abc” in read mode

ファイルを開く(2)

- fopen関数でファイルを開く
  - 引数はファイル名とモード
    - “r”で読み込み、“w”で書き込み

```
   ファイルのオープンとクローズ
*/
#include <stdio.h>

int main(void)
{
  FILE *fp;

  fp = fopen( “abc”,  “r” ); /* ファイルのオープン */
```

↓「abc」という名前のファイルを読み込みモードで開く

Open a file (3)

```
   Open and close a file
*/
#include <stdio.h>

int main(void)
{
  FILE *fp;

  fp = fopen( “abc”,  “r” ); /* Open a file */

  if (fp == NULL) {
    printf( “¥aCannot open the file.¥n” );
  } else {

    /* Read from the file, etc. */
```

← fopen returns NULL if opening the file was failed.

ファイルを開く(3)

```
   ファイルのオープンとクローズ
*/
#include <stdio.h>

int main(void)
{
  FILE *fp;

  fp = fopen( “abc”,  “r” ); /* ファイルのオープン */

  if (fp == NULL) {
    printf( “¥aファイルをオープンできません。¥n” );
  } else {

    /* ファイルから読込みなどを行う */
```

←オープンに失敗すると、fopenはNULLを返す

Close a file

- Close the file when you finish using the file.
  - Close the file by function “fclose”
  - Argument: pointer to the file
  - You might get error when opening many files without closing the files.

```
  if (fp == NULL) {
    printf( “¥aCannot open the file.¥n” );
  } else {

    /* Read from the file, etc. */

    fclose(fp); /* Close the file */
  }
  return (0);
}
```

ファイルを閉じる

- 作業が終わったらファイルを閉じる
  - fclose関数でファイルを閉じる
  - 引数はFILE構造体へのポインタ
  - 閉じ忘れたまま大量にファイルを開くとそのうちエラーに

```
  if (fp == NULL) {
    printf("¥aファイルをオープンできません。¥n");
  } else {

    /* ファイルから読込みなどを行う */

    fclose(fp); /* ファイルのクローズ */
  }

  return (0);
}
```

Input/output through FILE structure

- fscanf, fprintf
  - scanf / printf like functions for files
  - You need to specify input/output file
    - by using a pointer to FILE structure
  - Other arguments are the same as scanf, printf

```
scanf( “%d”, &n);
fscanf(fp, “%d”, &n);
```

```
printf( “n = %d¥n”, n);
fprintf(fp, “n = %d¥n”, n);
```

FILE構造体経由の入出力

- fscanf, fprintf
  - scanf / printf 的なもの
  - 入出力先を指定する
    - ファイル構造体へのポインタを使う！
  - それ以外の書式はscanf, printfと同じ

```
scanf( “%d”, &n);
fscanf(fp, “%d”, &n);
```

```
printf( “n = %d¥n”, n);
fprintf(fp, “n = %d¥n”, n);
```

Data stream

- Data stream = data current
  - File input/output = data current from/to a file
    - File stream
  - All input and output are data stream, including files.
  - Standard stream
    - Standard input, Standard output, Standard error output
    - stdin, stdout, stderr
    - Pointers to the special FILE structures prepared by the system.

```
scanf( “%d” , &n) = fscanf(stdin, “%d” , &n);
```

```
printf( “n = %d¥n” , n) = fprintf(stdout, “n = %d¥n” , n);
```

ストリーム

- ストリーム＝流れ
  - ファイルへの入出力はデータの流れ
    - ファイルストリーム
  - ファイルだけじゃなく、他の入出力も含めて、全てがデータの流れ
  - 標準ストリーム
    - 標準入力・標準出力・標準エラー出力
    - stdin, stdout, stderr
    - システムが用意している特殊なファイル構造体へのポインタ

```
scanf( “%d”, &n) = fscanf(stdin, “%d”, &n);
```

```
printf( “n = %d¥n”, n) = fprintf(stdout, “n = %d¥n”, n);
```

Read/write of binary data (1)

- Binary data
  - Data represented in binary (by bits)
  - Data representation inside of computers
- Text data
  - Data included only character code which human can read
  - ASCII code, etc.

- To store numerical data, using the data representation inside of computers is efficient.

バイナリデータの読み書き（1）

- バイナリデータ
  - ビットの並びで表現されたデータ
  - コンピュータの内部表現
- テキストデータ
  - 人間が読めるデータ
  - ASCIIコードなど

- 数値データを保存するのには、コンピュータの内部表現をそのまま用いる方が効率が良い

Read/write binary data (2)

- The file should be opened in “binary mode”.

```
fp = fopen( “abc” ,  “rb” );
```

```
fp = fopen( “def” ,  “wb” );
```

- The file closing is the same before.

```
fclose(fp);
```

バイナリデータの読み書き(2)

- ファイルのオープンはバイナリモードですること！

```
fp = fopen( “abc”, “rb” );
```

```
fp = fopen( “def”, “wb” );
```

- 閉じるのは同じ

```
fclose(fp);
```

Read/write binary data (3)

- Read binary data from a file: function “fread”

```
int x;
fread(&x, sizeof(int), 1, fp);
```

  - 1st argument: pointer to the memory which data should be stored.
  - 2nd argument: ”sizeof” of the variable type which data should be stored.
    - ”int” in the above example
  - 3rd argument: “1” is okay before you learn array
    - “fread” reads data which the size is (2nd arg. × 3rd arg.) bytes.

- Write binary data to a file: function “fwrite”

```
float x;
fwrite(&x, sizeof(float), 1, fp);
```

  - Arguments are the same as “fread”.
  - “fwrite” writes the Data in the 1st arg. to the file.

バイナリデータの読み書き(3)

- ファイルからデータを読み込む:fread関数

```
int x;
fread(&x, sizeof(int), 1, fp);
```

  - 第1引数は読み込んだデータを入れる領域へのポインタ
  - 第2引数の"sizeof"にデータを入れる変数の型を与える
    - 上の例では"int"
  - 第3引数は配列を覚えるまでは"1"で固定
    - 第2引数×第3引数[byte]のデータを読み込む

- ファイルにデータを書き出す:fwrite関数

```
float x;
fwrite(&x, sizeof(float), 1, fp);
```

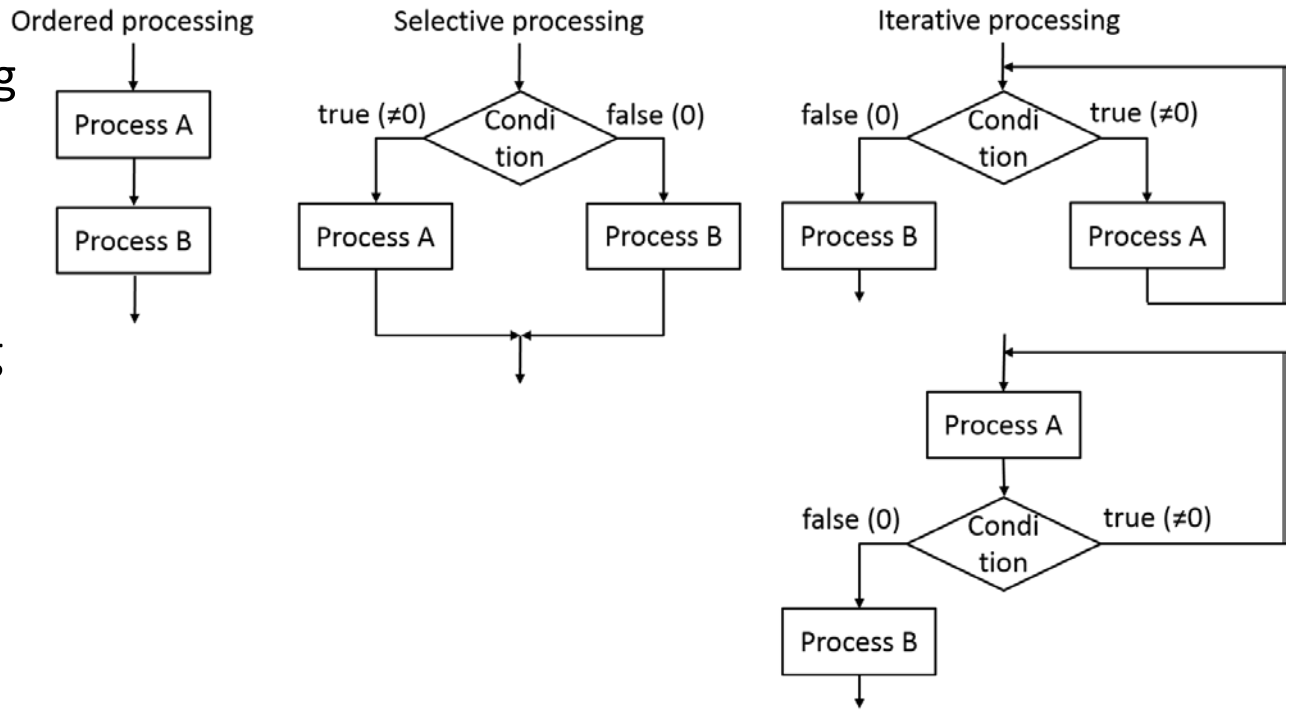
  - 引数の順序・書式はfreadと同じ
  - 第1引数に入っているデータをバイナリ形式でファイルに書き出す

CONTROL FLOW

制御構造

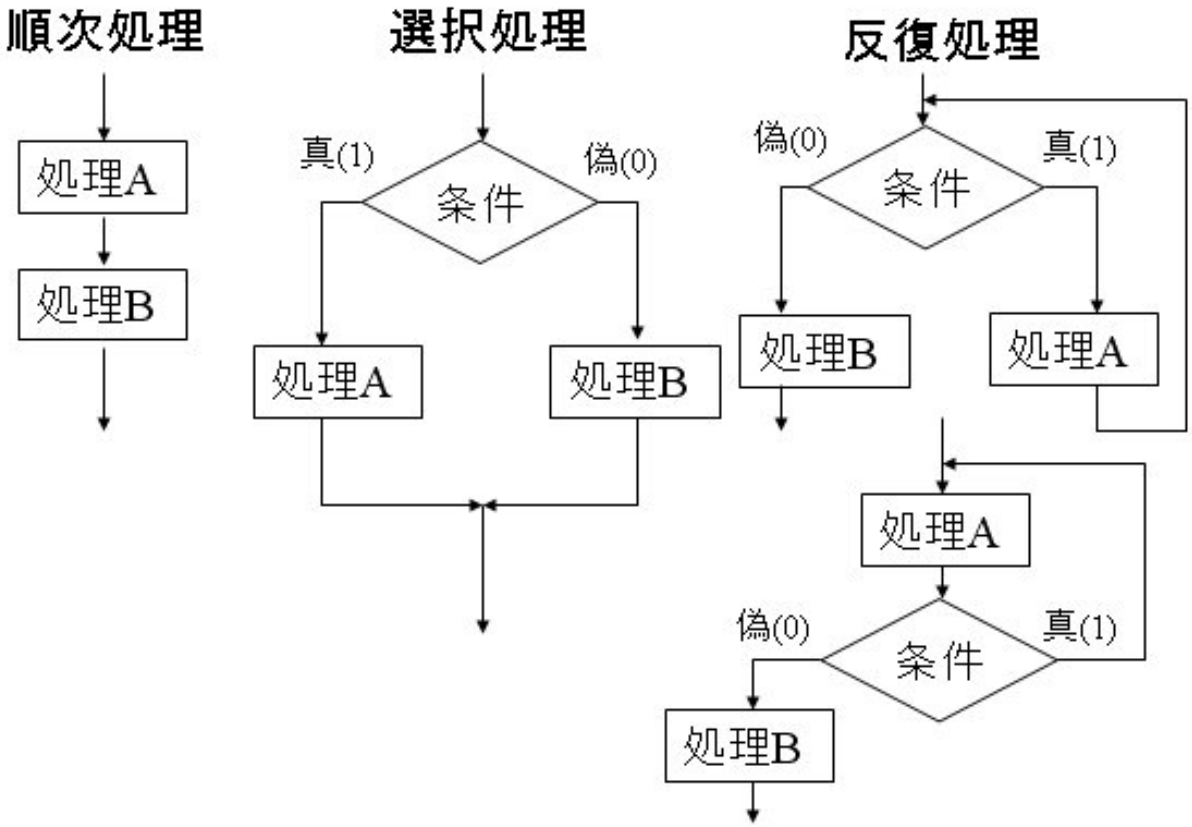
Control flow

- Ordered processing
- Selective processing
  - if – else
  - switch
- Iterative processing
  - while
  - do - while
  - for
- Leap processing
  - goto
  - continue / break
  - return



制御構造

- 順次処理
- 選択処理
  - if – else 文
  - switch 文
- 反復処理
  - while 文
  - do – while 文
  - for 文
- 跳躍処理
  - goto
  - continue / break
  - return



if – else

- if (expression) statement
  - Do “statement” if the value of “expression” is not zero
    - True / False in C language = “not zero” / “zero”
  - It’s okay that the statement will be a block “{…}”

- if (expression) statement_1 else statement_2
  - Do “statement_1” if the value of “expression” is not zero, or do “statement_2” if it’s zero
  - It’s okay that the sentences will be “if” block (Nest of if).
    - “else” will be corresponded to the nearest “if”.
    - Give correct indents in your code in order to see the nest correctly.

if – else 文

- if (式) 文
  - “式”の値が0でなければ、文が実行される
    - C言語における真・偽 ＝ ”0以外” ・ ”0”
  - 文はブロック “{ … }” でも良い

- if (式) 文$_1$ else 文$_2$
  - “式”の値が0でなければ文$_1$が、0ならば文$_2$が実行される
  - 文が if 文でもよい（if文のネスト）
    - else は最も近い if に対応づけられる
    - どこに対応しているかすぐ分かるように、プログラムにインデントをきちんと入れるようにしましょう

switch

- switch (a) {
  case 0 :
  statement$_0$
  break;
  case 1 :
  statement$_1$
  break;
  ……
  default :
  statement$_{default}$
  }
  - Do "statement$_x$" corresponding x, the value of "expression" (integral type).
  - Do "statement$_{default}$" if the value of expression does not match to any labels.
  - Getting out from "switch" by "break"
    - If "break" is not there, "case" is treated as a label simply and the process will be forwarded (fall-down)

switch

- switch (a) {
  case 0 :
  文$_0$
  break;
  case 1 :
  文$_1$
  break;
  ……
  default :
  文$_{default}$
  }
  - “式”（整数型）の値によって、その値 x に対応するラベル（”case x :”）以下の文（文$_x$）が実行される
  - 式の値がどのラベルにも対応しなければ、default以下の文（文$_{default}$）が実行される
  - break文によりswitch文から抜け出す
    - break文がなければ、case文はラベルと見なされ、処理はそのまま続く（fall-down）

while / do ~ while

- while (expression) statement
  - Firstly, evaluate “expression”. If the result is not 0, do “statement”. If 0, forward next process.
  - After processing “statement”, evaluate “expression” again. If the result is not 0, do “statement” again. If 0, forward next process.
  - Repeat this.
- do statement while (expression) ;
  - Firstly, do “statement”, then evaluate “expression”. If the result is not 0, do “statement” again. If 0, forward next process.
  - After processing “statement”, evaluate “expression” again. If the result is not 0, do “statement” again. If 0, forward next process.
  - Repeat this.

- Infinite loop by “while (1) statement” or “do statement while (1);”

while / do ～ while 文

- while（式）文
  - まず”式”を評価し、0でなければ”文”を実行する。0ならば次の処理に進む
  - 文の実行後、再び式を評価し、0でなければ”文”を実行する。0ならば次の処理に進む
  - これを繰り返す
- do 文 while（式）;
  - まず”文”を実行し、その後”式”を評価する。式の値が0でなければ再び文を実行、0ならば次の処理に進む
  - 文の実行後、再び式を評価し、0でなければ”文”を実行する。0ならば次の処理に進む
  - これを繰り返す

- “while (1) 文” や “do 文 while (1);” とすることで無限ループに

for

- for ($expr_1$; $expr_2$; $expr_3$) statement
  1. Do "$expr_1$" ($expr_1$ is able to be omitted.)
  2. Evaluate "$expr_2$" (If no "$expr_2$", the value will be "not 0")
     1. If the value of "$expr_2$" is not 0, do "statement"
     2. If the value of "$expr_2$" is 0, finish this "for" loop.
  3. Do "$expr_3$" ($expr_3$ is able to omitted.)
  4. Back to 2.

- Infinite loop by "for (; ;) statement"

for 文

- for (式$_1$; 式$_2$; 式$_3$) 文
  1. “式$_1$”を実行（式$_1$は省略可）
  2. “式$_2$”を評価（式$_2$が無い場合は0でないと見なす）
     1. “式$_2$”が0でなければ、”文”を実行
     2. “式$_2$”が0なら、このfor文を終了。次の処理へ進む
  3. “式$_3$”を実行（式$_3$は省略可）
  4. 2. に戻る

- “for (; ;) 文” で無限ループに

Leap control

- goto label;
  - Forced jump to a label (“label:”) in the same scope (block) continue;
- continue;
  - Forced jump to the end of the current loop (not outside of the loop!)
- break;
  1. Getting away from switch
  2. Getting away from current (most inside) loop
- return expression ;
  - Get back to where a function is called from the function (In the case of “main” function, get back to shell)
  - If the function has a value (not “void” type), the function return a value of “expression”
    - Don’t write “expression” if the function type is “void”

跳躍制御

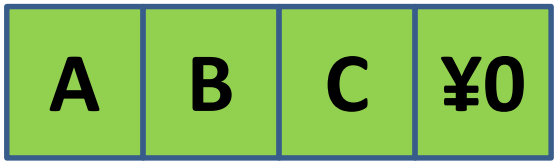
- goto ラベル;
  - 同じスコープ（ブロック）内のラベル（” ラベル : ”）へ強制ジャンプする
- continue;
  - 現在いるループの末尾（ループの外ではない！）に強制ジャンプする
- break;
  1. switch文から脱出する
  2. 現在いる（最も内側の）ループから脱出する
- return 式 ;
  - 関数から呼び出し元に戻る（mainの場合はshellに戻る）
  - 関数が値を持つ場合（void型以外の場合）は、”式”の値を返す
    - void型の場合は、”式”を書かない

CHARACTER STRING AND POINTER

文字列とポインタ

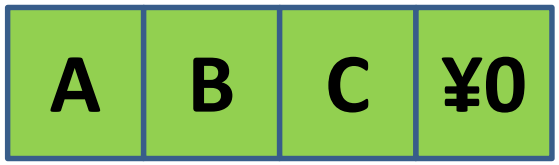
Character string (1)

- Character string constant
  - Double quoted character(s)
  - like ”ABC”
- Character string
  - No character string type in C language. Actual character string is “array of char type”
  - In the end of a character string, ‘¥0’ (numerical zero, NULL character) must be inserted.



文字列（1）

- 文字列リテラル（文字列定数）
  - ダブルクォートで文字を囲んだもの
  - 「”ABC”」
- 文字列
  - C言語には「文字列型」が存在しないので、文字列の実体は「文字型（char型）の配列」
  - 文字列の終端に ’¥0’（数字のゼロ。ヌル文字）が入っている



Character string (2)

- In “printf()”, character string is denoted by “%s”.
  - Display characters until the first NULL character in the array of char type.
  - If not found a NULL character, display characters over the pre-defined array range, which leads “Segmentation Fault”.
    - CAUTION: This is a frequent mistake in operating a character string.
- Initialization
  - The variable of character strings can be initialized by assignment at definition.

```
char ss[] = { ‘A’ ,  ‘B’ ,  ‘C’ ,  ‘¥0’ };
                              // the same as array initialization
char ss[] =  “ABC”            // usually this for simplicity
```

文字列(2)

- 文字列は printf() では ”%s” で表示できる
  - 最初のヌル文字までが表示される
  - ヌル文字が見つからなかったら、配列を定義した範囲を越えて延々と表示され続け、”セグメンテーションフォールト”になる。
    - 文字列操作でよくやるミスなので、注意すること
- 初期化
  - 宣言時に代入して初期化できる

```
char ss[] = { ‘A’ ,  ‘B’ ,  ‘C’ ,  ‘¥0’ };  // 配列の初期化と同じ
char ss[] =  “ABC”   // 面倒なので普通はこちら
```

Character string (3)

- Assignment
  - Character string constant can be used for initialization.
  - However, character string constant cannot be used for assignment.

```
str = { ‘A’ ,  ‘B’ ,  ‘C’ ,  ‘¥0’ };
str =  “ABC”                              // Cannot do both!
```

  - Assign character by character

```
str[0] =  ‘A’ ; str[1] =  ‘B’ ; str[2] =  ‘C’ ; str[3] =  ‘¥0’ ;
```

    - That’s the same as an array operation.

文字列(3)

- 代入
  - 初期化は文字列リテラルでできる
  - 文字列リテラルを代入することはできない

```
str = { 'A' ,  'B' ,  'C' ,  '¥0' };
str =  "ABC"                              // どちらもできない！
```

  - 代入は1文字ずつ

```
str[0] =  'A' ; str[1] =  'B' ; str[2] =  'C' ; str[3] =  '¥0' ;
```

    - すなわち「配列と同じ」です

Character string (4)

- Empty character string
  - Empty = only null character

- Reading character string by “scanf()”

```
char name[40];  // We don’t know how long string is entered.

scanf( “%s” , name);
```

“%s” for character string

“&” is not necessary before the variable name

文字列(4)

- 空文字列
  - 内容が空の文字列＝先頭がヌル文字

- scanf() による文字列の読み込み

```
char name[40];  // 何文字入れられるかは分からない

scanf( “%s” , name);
```

文字列の場合は %s

先頭に”&”はいらない

Danger of scanf

- You can input the arbitrary number of characters when scanf() uses ”%s”.
  ⇒ The character string could be over the pre-defined array range.
  ⇒ “Buffer overrun”
  - It could be a security hole if it happened in a system program.
  - Workaround: limiting the number of characters like “%10s”
    - But, it becomes difficult to use the program.
  - “gets()” have the same problem. There is no workaround for “gets()”.
    - Do not use “gets()” ! Use “fgets()” instead.

scanfの危険性

- scanf() で ”%s” 指定子を用いると、何文字でも入力できてしまう！
  ⇒ 用意した文字配列の要素を越えてしまう場合がある
  ⇒「バッファーオーバーラン」
  - システムプログラムで起きると、セキュリティホールになる場合がある
  - “%10s” のように入力できる文字数を制限することで回避できる
    - が、使い難くなる
  - gets() にも同じ問題。gets() の場合は回避策がない。
    - gets() は使わない！ 代わりに fgets()を使う。

Array of character strings

- Character string is an array of char type.
- 2-dimensional array of char type is “an array of character strings”

```
char cs[][6] = { “Turbo”,  “NA”,  “DOHC” };
```

More than max length of character string constants + 1

  - The same as 2-dimensional array initialization, but character string constants can be used.
  - cs[0][0] is ’T’, cs[1][0] is ’N’, cs[2][0] is ’D’、cs[0][1] is ’u’, cs[1][1] is ’A’, cs[2][1] is ’O’

文字列の配列

- 文字列は文字型の配列
- 2次元配列にすれば、文字列の配列になる

```
char cs[][6] = { “Turbo”, “NA”, “DOHC” };
```

要素文字列の最大長＋１以上必要

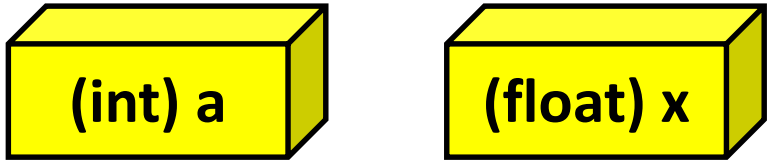
  - 文字列リテラルが使える点を除けば、2次元配列の初期化と同じ
  - cs[0][0]が’T’、cs[1][0]が’N’、cs[2][0]が’D’、cs[0][1]が’u’、cs[1][1]が’A’、cs[2][1]が’O’

POINTER

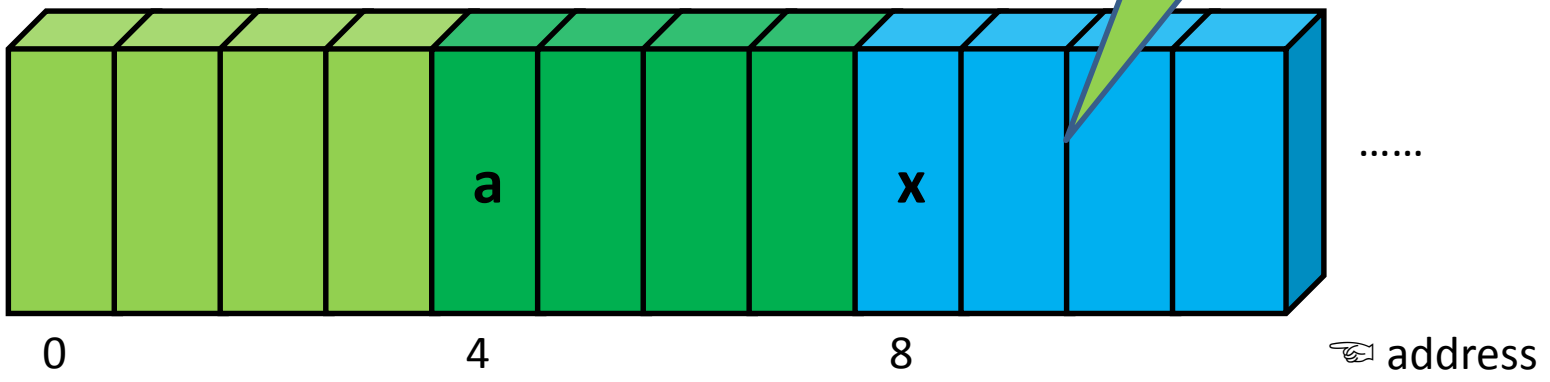
ポインタ

Memory model of computers

- Attitude
  - Box model ⇒ Individual box has a variable name.

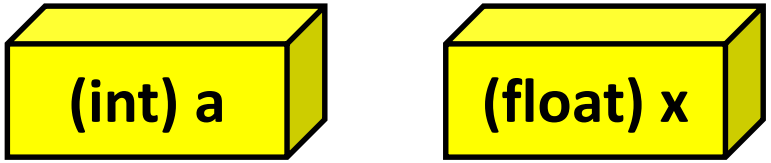


  - Actual computer memory
    - memorize by matching a variable name to an address

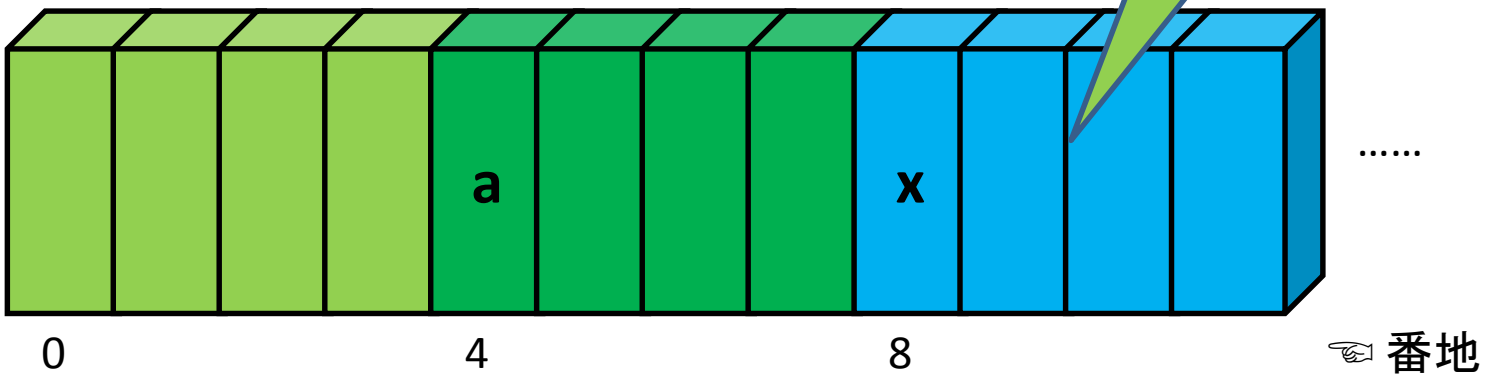


計算機のメモリモデル

- 変数の考え方
  - 箱モデル ⇒ 独立の箱に変数の名前

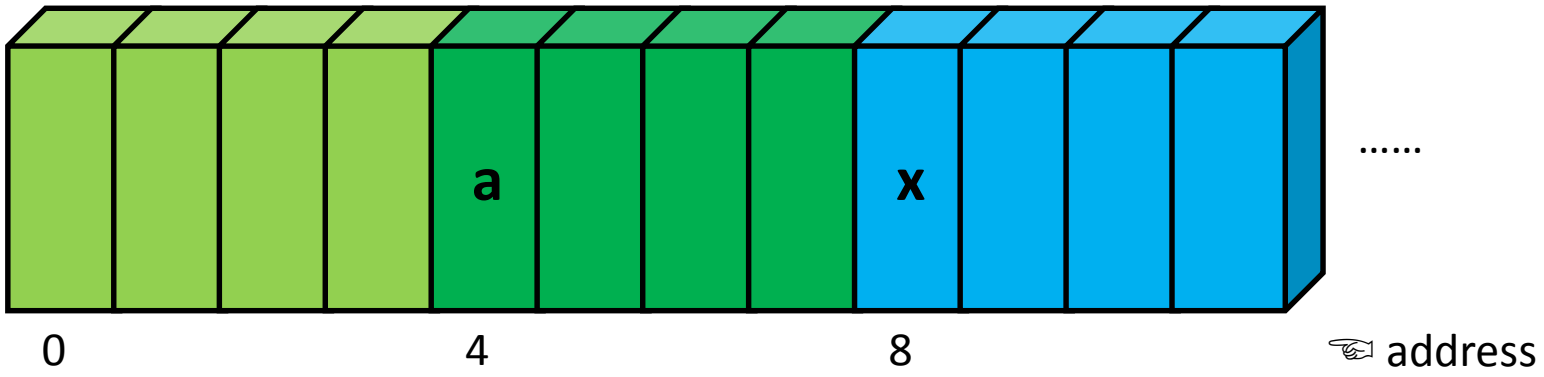


  - 実際の計算機のメモリ
    - 番地（アドレス）と変数名を対応させて記憶



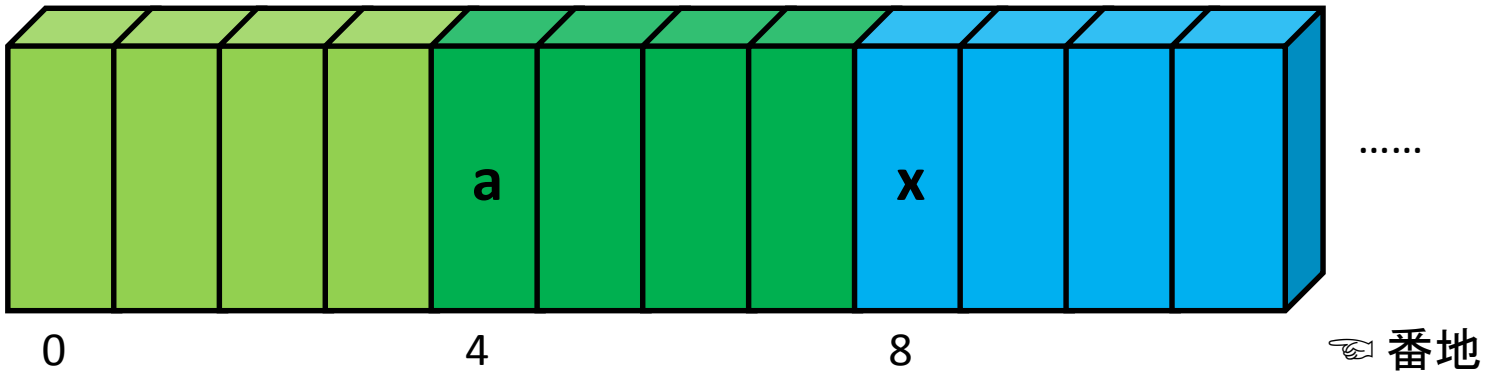
Address operator

- Want to know the address of variable
  - Address operator “&”
    - used in scanf(), etc.
    - The value is the “heading address” of variable.
      - “&a” is “4”, “&x” is “8”



アドレス演算子

- 変数のアドレスを知りたい！
  - アドレス演算子 “&”
    - scanf()なんかで使うアレ
    - 変数の”先頭番地”が値となる
      - “&a”は “4”、“&x”は“8”になる



Pointer (1)

- For what knowing address?
  - You can operate variables directly from address, not via variable names.
  - Very useful for operating arrays or character strings
  - Very useful for happening a side-effect in a function
    - Side-effect: Overwrite variables in a calling function
- Variable storing address ⇒ Pointer type
  - declare by using “*”

```
int *ptr;
```

ポインタ(1)

- アドレスを知ると何の役に立つのか？
  - 変数を介さずにアドレスから直接変数を操作できる！
  - 配列や文字列の操作にとても便利
  - 関数に副作用を起こさせるためにも便利
    - 副作用:呼出し元の変数の値を書きかえる効果
- アドレスを格納する変数 ⇒ ポインタ型
  - “*”を付けて宣言する

```
int *ptr;
```

Pointer (2)

- int type object
  - can store an integer type (int) value.
- pointer to int type object
  - can store the address of ≪the box storing an integer type value≫.

```
int a, *ptra;

ptra = &a;
/* assignment the address of variable  “a”
                                        to pointer  “ptra”  */
```

ポインタ(2)

- int型のオブジェクト
  - 整数(int型)の値を格納する
- intへのポインタ型のオブジェクト
  - ≪整数を格納する箱≫の“アドレス”を格納する

```
int a, *ptra;

ptra = &a;
/* 変数 a のアドレスをポインタ変数 ptra に代入 */
```

Indirection operator (1)

- Want to know the value in the address which stored in a pointer !
  - You can get the value by indirection operator “*”.
    - Same “*” as the pointer definition

```
int a, b, *ptra;

ptra = &a;
/* assignment the address of variable  “a”
                                    to pointer  “ptra”  */
a = 100;
b = *ptra;  /*  “b”  becomes 100
              because  “*ptra” is the same as  “a”  */
a = 200;
b = *ptra;  /*  “b”  becomes 200 */
```

間接演算子(1)

- ポインタ変数に格納されているアドレスにはどんな値が入っているか？
  - 間接演算子 “*” で値を取り出せる
    - 変数宣言と同じ “*”

```
int a, b, *ptra;

ptra = &a;
/* 変数 a のアドレスをポインタ変数 ptra に代入 */

a = 100;
b = *ptra;  /* “*ptra” は “a” と同じなので
                        b には 100 が入る! */
a = 200;
b = *ptra;  /* b には 200 が入る! */
```

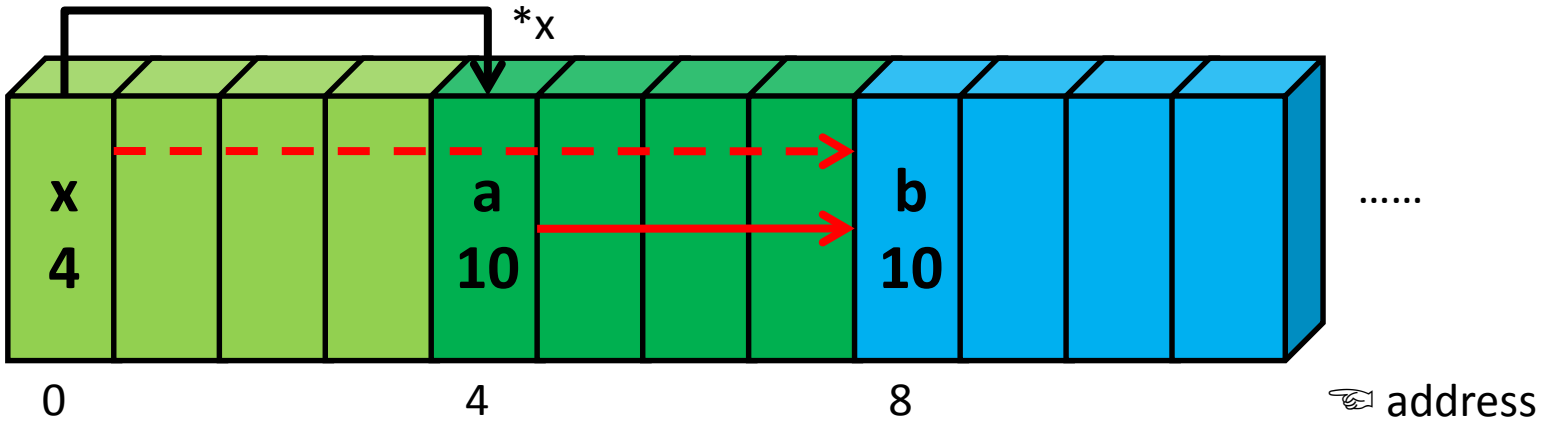
Indirection operator (2)

- In the case of the figure below...

```
int *x, a, b;

x = &a;
/* assignment the address (4) of variable “a” to pointer “x” */

a = 10;
b = *x;  /* b becomes 10 because “*x” is the same as “a” */
```



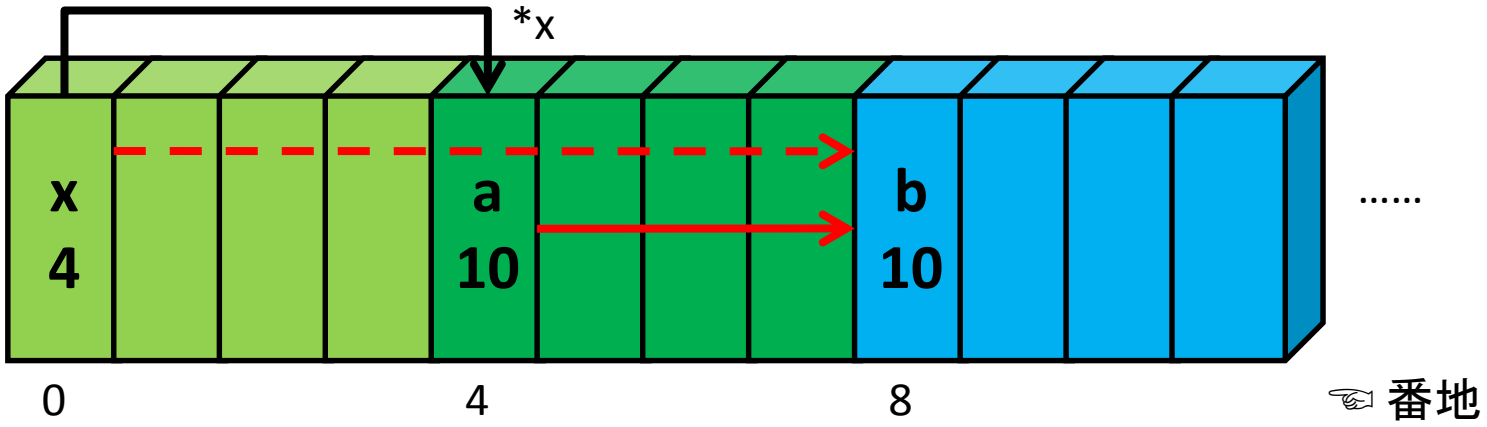
間接演算子(2)

- 下の図のような場合

```
int *x, a, b;

x = &a;
/* 変数 a のアドレス(4)をポインタ変数 x に代入 */

a = 10;
b = *x;  /* “*x” は “a” と同じなので
                                b には 10 が入る! */
```



Pointer and function (1)

- The default variable passing rule to function is “Call-by-Value” in C language.
  - The function’s arguments and variables are independent from caller’s variables.
    ⇒ Caller’s variables are unchanged.

- It’s good for “independence of functions”, but sometimes you want to change the value of caller’s variable in the function.
  - Change the value indirectly by passing a pointer to the function !
  - The technique can be used for returning multiple values from a function.

ポインタと関数（１）

- C言語での関数の引数は“値渡し”
  - 関数の引数・変数は呼出し側の変数とは独立
    ⇒ 呼出し側の変数の値は変更されない

- “関数の独立性”という観点からは良いが、値を変更したいときもある
  - ポインタを関数に渡して間接的に変更する！
  - １つの関数から複数の値を返したい場合にも使える

Pointer and function (2)

- Passing an address by a pointer

```
void function(int *b)  /* attach "*" to the argument */
{
  if (*b < 10)
      *b = 10;  /* assign 10 to the address pointed by b */
}

int main(void)
{
  int a;
  …
  function(&a);  /* passing the address of a to function */
  /* The value of a has changed when getting back from function */
  …
}
```

ポインタと関数(2)

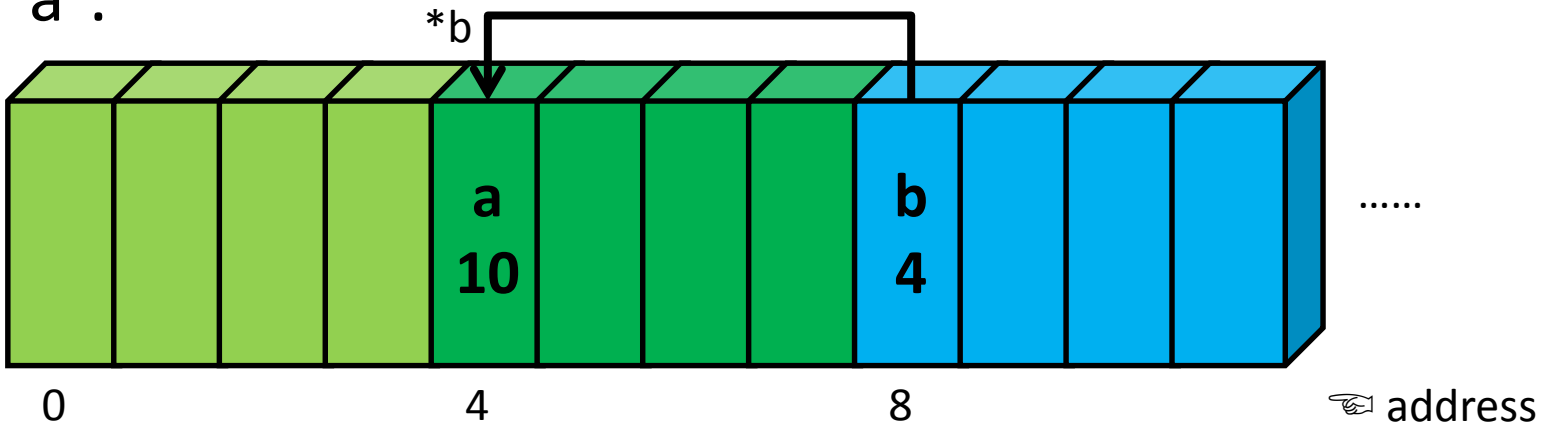
- ポインタでアドレスを渡す

```
void function(int *b)  /* 引数に “*” を付ける */
{
  if (*b < 10)
      *b = 10;  /* bが示すアドレスに 10 を格納 */
}

int main(void)
{
  int a;
  …
  function(&a);  /* 関数に a のアドレスを渡す */
  /* 関数から戻ってくると、aの値が変わっている */
  …
}
```

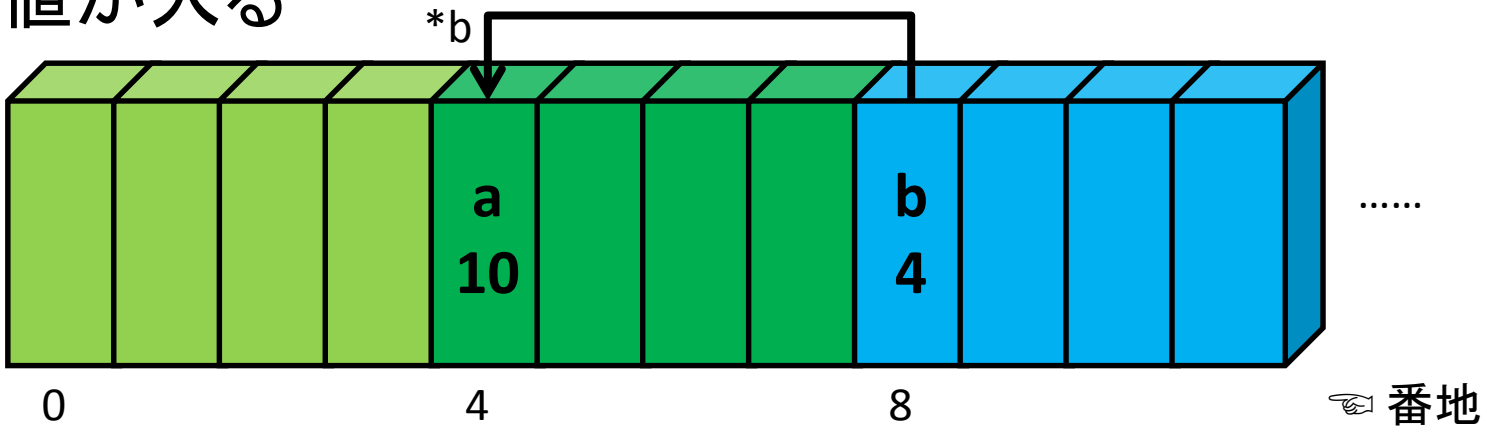
Pointer and function (3)

- In the case of the figure below...
  - The value of “&a” is “4”
  - “4” is passed to function() as the argument “b”
    - The address is passed on the “Call-by-Value” rule.
  - When assigning “*b”, the value is assigned caller’s “a”.



ポインタと関数（3）

- 下の図のような場合
  - “&a”の値は“4”
  - 関数function()の引数“b” には“4”が渡される
    - アドレスが“値渡し”される
  - “*b”に代入すると、呼び出し元の“a”に値が入る



Call-by-Reference

- Passing the “reference” (property of the variable, such as place) of variable, not the “value” of variable
  - not in C language. C++ language has.
  - “Passing a pointer” is “Call-by-Value of pointer”.

- Actually,
  - “Passing a pointer” is so-called “Call-by-Reference” in some books.
  - You have to take notice that it is true “Call-by-reference”, or just so-called “Call-by-Reference” in the book.

参照渡し

- 変数の値ではなく変数の参照（変数の場所等、諸々の属性）を渡すやり方
  - C言語にはない。C++言語にはある。
  - ポインタ渡しは、“ポインタの値渡し”

- ただし
  - C言語におけるポインタ渡しを、
    俗に“参照渡し”と呼んでいる場合がある
  - 本当の意味での“参照渡し”を指しているのか、
    俗な言い方での“参照渡し”を指しているのか、
    本を読んだりするときに気を付けること

Pointer and array (1)

```
int main()
{
  int i;
  int vc[5] = { 10, 20, 30, 40, 50 };
  int *ptr = &vc[0];

  for (i = 0; i < 5; i++)
    printf( "vc[%d] = %d  ptr[%d] = %d  *(ptr + %d) = %d¥n" ,
              i, vc[i], i, ptr[i], i, *(ptr + i));
```

- Initialization: ptr = &vc[0]
- "ptr + i" means the pointer of "i" objects behind from the object pointed by "ptr" (= &vc[0])
  - That is, *(ptr + i) ⇒ the same as "vc[i]" in the above case

ポインタと配列（１）

```
int main()
{
  int i;
  int vc[5] = { 10, 20, 30, 40, 50 };
  int *ptr = &vc[0];

  for (i = 0; i < 5; i++)
    printf( “vc[%d] = %d  ptr[%d] = %d  *(ptr + %d) = %d¥n”,
              i, vc[i], i, ptr[i], i, *(ptr + i));
```

- ptr = &vc[0]と初期化
- “ptr + i”は、ptrが指すオブジェクト（ここでは配列vc[]の先頭）の i 個後ろの要素を指すポインタとなる
  - すなわち、*(ptr + i) ⇒ vc[i] と同じ

Pointer and array (2)

- Why pointer has “types” in spite of address?

- “ptr + i” is NOT “i bytes behind from the address pointed by ptr”.
  It means “i objects behind from the object pointed by ptr”
  - The size of the object should be known to decide an actual address.
  - “Type of pointer” decides it.

ポインタと配列（２）

- ポインタはアドレスを指しているのに、何故「型」があるのか？

- “ptr + i” は、
  “メモリ上のアドレス ptr の i 番地後ろ”
  ではなくて、
  “ptrが指すオブジェクトの i 個後ろの要素”
  である
  - アドレスを決定するためには、１個のオブジェクトの大きさが分からないといけない
  - それを決めているのが「ポインタの型」

Pointer and array (3)

- In the case of “float *ptr”, sizeof(float) is 4 bytes, so the actual address of “ptr + 1” becomes the address of “ptr” + “4”.
- In the case of “double *ptr”, sizeof(double) is 8 bytes, so the actual address of “ptr + 1” becomes the address of “ptr” + “8”.

- The address calculation will be done automatically, if you give a correct type to the pointer.

ポインタと配列（３）

- “float *ptr”だったら、sizeof(float) は 4 Byte なので、“ptr + 1” の実際のアドレスは、ptrに“4”を加えたものになる
- “double *ptr”だったら、sizeof(double) は 8 Byte なので、“ptr + 1” の実際のアドレスは、ptrに“8”を加えたものになる

- ポインタの型を正しく与えておけば、このような計算は自動で正しく行われる

Pointer and array (4)

- Writing like “&vc[0]” bothers me.

- Only “vc” has the same mean of “&vc[0]”.
  - Exception 1: “sizeof(vc)” (means whole array size)
  - Exception 2: “&vc” (means a pointer to whole array)

- How large memory pointed by a pointer can be used?
  - Pointer points just an address, and does not manage the memory space. You have to manage the memory space by yourself.
    - You have to think how large memory you can use while programming.
    - This is a difference from “reference”.

ポインタと配列（４）

- いちいち “&vc[0]” って書くの面倒だよね
- “vc” とだけ書くと “&vc[0]” と同じ意味になる！
  - 例外１：“sizeof(vc)” の場合（配列領域全体のサイズ）
  - 例外２：“&vc” の場合（配列領域全体を指す型へのポインタ）
- ポインタが指す領域はどこまで使えるか？
  - ポインタは単なるアドレスなので、配列の大きさ（領域）に対する配慮はプログラマが自分でしなければならない
    - どこまで使って良いか、自分で考えながらプログラミングする
    - それが “参照” との違い

Passing array to function (1)

- Passing array to function
  - In the function definition, only "[]" is attached.

```
int function(int x[], int no) /* int *x is okay,
                                 but it has different meaning precisely */
{
  return x[no];
}
```

    - Writing the number of components is okay.
  - In the caller function, "[]" is unnecessary.

```
  int array[100], num, value;

  value = function(array, num);
```

配列の受渡し(1)

- 配列の関数への受渡し
  - 関数定義側では、"[]"だけを付ける

```
int function(int x[], int no) /* int *x でも良いが、
                                 厳密には意味が異なる*/
{
  return x[no];
}
```

    - 要素数を書いても間違いではない
  - 呼び出し側では"[]"を付けない

```
  int array[100], num, value;

  value = function(array, num);
```

Passing array to function (2)

- Caller

```
int array[100], num, value;

value = function(array, num);
```

  - Just “array” as the argument. “array” means “&array[0]”, that is, the head address of the array.
    ⇒ Passing it to the function.

配列の受渡し(2)

- 呼出し側

```
int array[100], num, value;

value = function(array, num);
```

  - 配列名だけ書いているので、“array” は “&array[0]” と同じ意味になり、配列の先頭アドレスを指す ⇒ それを関数に渡している

Passing array to function (3)

- Function side

```
int function(int x[], int no)  /* int *x has the same meaning
                                   only as arguments of a function */
{
  return x[no];
}
```

  - Usually, “x[]” is different from “*x”.
    - An array variable is “const”.
  - They are the same thing as arguments of a function (just passing an address).

配列の受渡し(3)

- 受け取り側

```
int function(int x[], int no)  /* 仮引数の場合のみ、
                                  int *x でも同じ意味に */
{
  return x[no];
}
```

  - 通常、“x[]” と “*x” は別物である
    - 配列変数は const である
  - 関数の仮引数では、同じもの(単なるアドレスの受渡し)になる

Character string constant and pointer

- A character string constant is stored in somewhere on the memory.
- When evaluating a character string constant, the pointer to heading character of the character string will be returned.

```
char str[] =  “ABC” ; /* &str[0] = address of ‘A’  */
char *ptr  =  “123” ; /* ptr = address of ‘1’  */
```

  - The first component of the array “str” is initialized by ’A’, the second component by ’B’, the third component by ’C’, and the fourth component by zero.
  - “ptr” is assigned the heading address of a character string constant stored in somewhere on the memory.
    - The address of “ptr” itself on the memory is not related to the character string constant.

文字列リテラルとポインタ

- 文字列リテラルは、メモリ上のどこかに確保されている
- 文字列リテラルを評価すると、文字列の先頭文字へのポインタが返される

```
char str[] = “ABC” ; /* &str[0] = ‘A’ のアドレス となる */
char *ptr  = “123” ; /* ptr =  ‘1’ のアドレス となる */
```

  - 配列strの最初の要素が'A'、2番目の要素が'B'、3番目の要素が'C'、4番目の要素が0で初期化される
  - ptrには、メモリ上のどこかに確保された文字列の先頭アドレスが代入される
    - ptrが確保される場所と文字列リテラルの場所は無関係

Common point between array and pointer

- You can use subscript operator “[ ]” to a pointer.
  - It has the same effect as for an array.
  - You can do the same thing as an array by a pointer.

```
char str[] =  “ABC” ; /* &str[0] = address of ‘A’  */
char *ptr  =  “123” ; /* ptr = address of ‘1’  */
```

  - In this case, ptr[0] is ‘1’, ptr[1] is ‘2’, and ptr[2] is ‘3’.
    - This is the same as an array because a pointer has only the heading address of a memory region (the number of components is ignored).

配列とポインタの共通点

- 配列で使う添字演算子“[” “]”をポインタに対しても使うことができる
  - 効果も同じ
  - ポインタを使って、配列と同じことができる

```
char str[] = “ABC” ; /* &str[0] = ‘A’ のアドレス となる */
char *ptr  = “123” ; /* ptr =  ‘1’ のアドレス となる */
```

  - この場合、ptr[0]は‘1’ 、ptr[1]は‘2’、 ptr[2]は‘3’である
    - C言語では、配列の先頭アドレスしか見ていない（要素数は無視）ので全く同じになる

Difference between array and pointer (1)

- An array variable is “const” so that an array variable is not assigned.
- Pointer can be assigned.

```
char str[] = “ABC” ;
char *ptr  = “123” ;

str = “DEF” ;  /* Error ! */
ptr = “456” ;  /* Correct ! */
```

  - Assignment of character string constant to an array variable is okay only at initialization.
  - Assignment of character string constant to a pointer is okay anytime, because the pointer has the heading address to a character string constant stored in somewhere on the memory.
    - The character string constant itself is not assigned to the pointer.

配列とポインタの違い（1）

- 配列変数はconst扱いなので、代入できない
- ポインタ変数は代入できる

```
char str[] = “ABC”;
char *ptr  = “123”;

str = “DEF”;  /* エラー！ */
ptr = “456”;  /* 正しい！ */
```

  - 配列変数への文字列リテラルの代入は、初期化の場合のみ可能
  - ポインタ変数の場合は、メモリ上のどこかに確保される文字列リテラルの先頭文字へのポインタが代入されるだけなのでプログラム中でも可能
    - ポインタ変数に文字列リテラルそのものが代入される訳ではない

Difference between array and pointer (2)

- An array can allocate extra memory which is unused when initializing because the number of components can be specified.
  - char str[256] = “string”; You can use the 256 byte space. When initializing, only 6 + 1 bytes are used.

- Pointer is an isolated variable to have an address.
  - A character string constant uses minimum memory space to keep the string.
  - A character string constant cannot allocate memory like an array.

配列とポインタの違い（2）

- 配列は、初期化時に要素数を指定できるので、使用しない領域を確保しておくことが可能
  - char str[256] = “string”; とすることで、256 bytesの領域を確保することが可能。初期化の段階で使っているのはその先頭の 6 + 1 bytes分
- ポインタはあくまでもアドレスを保持するだけの単独の変数
  - 文字列リテラルは、文字列に必要な領域しか確保されない
  - 配列のような領域確保はできない

Array of character strings (1)

- An array of character strings implemented by array-of-arrays (2-dimensional array) is considerably different from an array of character strings implemented by array-of-pointers in terms of memory entity.

- Array of character strings implemented by array-of-arrays
  - Array-of-arrays (2-dimensional array) in C language is a long array on memory entity which is treated as 2-dimensional.
    - char st[3][6] is the same as st[18] substantially.
    - st[2][5] is accessed as st[2 * 6 + 5].
  - That is, it's allocated on memory continuously.

文字列の配列（１）

- 文字列の配列は、配列の配列で実現する場合と、ポインタの配列で実現する場合で（メモリ上の実体が）かなり異なる

- 配列による文字列の配列の場合
  - C言語での配列の配列（2次元配列）は、1次元配列を2次元的に扱っているだけ
    - char st[3][6] と定義されたものは実質 st[18] と同じ
    - st[2][5] は st[2 * 6 + 5] としてアクセスされる
  - すなわち、メモリ上には連続的に配置される

Array of character strings (2)

- Array of character strings implemented by array-of-pointers
  - Each component of an array of pointers points only the heading address of each character string constant.
  - Each character string is somewhere on memory.
    - “Somewhere” depends on the computer.
- Array of character strings implemented by array-of-arrays
  - The empty spaces are filled by zero as the same as 2-dimensional array initialization.
  - In the case of pointer, not zero filled (because of no empty space), not allocated in continuous memory space.
    - An array of pointer itself is the same as other arrays (allocated in continuous memory space).

文字列の配列（2）

- ポインタによる文字列の配列の場合
  - ポインタ配列の要素は、それぞれの文字列リテラルの先頭アドレスを指すだけ
  - 文字列はそれぞれがメモリ上のどこかに存在する
    - 実際にどこに配置されるかは処理系依存
- 2次元配列による文字列の配列の場合
  - 配列で実現する場合は、2次元配列の初期化と同様に、間に0が埋め込まれている
  - ポインタの場合は0は入らないし、連続した領域に確保されてもいない
    - ポインタの配列も、他の配列と同じように扱われるので、連続した領域に確保される

Common and different point

- Common point
  - Both of array and pointer can use “subscript operator”.
    - It’s the same in writing and using rule for both.
    - No problem without distinction between them.
- Different point
  - Character string array by Array-of-arrays (2-dim. array)
    - Character string entities are allocated in continuous memory space. The entity is 2-dimensional array, so that the empty spaces are filled by zero.
    - Assignment of character string can be done only when initializing. Each character can be replaced (because it’s an array!).
  - Character string array by Array-of-pointers
    - Pointers are allocated in continuous memory space, but character string entities are not always allocated in continuous memory space.
    - A component can be replaced by another character string constant.

共通点・相違点まとめ

- 共通点
  - 配列にもポインタにも添字演算子が使える
    - 書き方も使い方も結果も同じ
    - 区別なく使って特に問題ない
- 相違点
  - 配列の配列（2次元配列）による文字列の配列
    - その実体が連続した領域に確保される（2次元配列なので、文字のないところには0が入る）
    - 配列なので、文字列の代入は初期化時しかできない。文字単位の置き換えはできる。
  - ポインタの配列による文字列の配列
    - ポインタは連続した領域に確保されるが、実体は必ずしも連続した領域に確保されるわけではない
    - 要素を別の文字列リテラルで置き換えることができる。