Today's contents

- Function and pointer
  - Array and pointer
  - "Array of pointers" and "pointer to array"
  - Passing a pointer to a function
    - Pointer of pointer
  - Pointer of function
- Multi-dimensional array and pointer
  - Multi-dimensional array and dynamic memory allocation

今日の内容

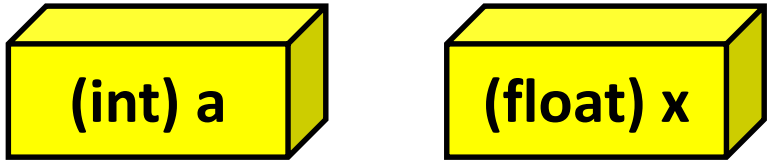
- 関数とポインタ
  - 配列とポインタ
  - 「ポインタの配列」と「配列へのポインタ」
  - 関数にポインタを渡す
    - ポインタのポインタ
  - 関数のポインタ
- 多次元配列とポインタ
  - 多次元配列と動的メモリ確保

POINTER ～REVIEW～

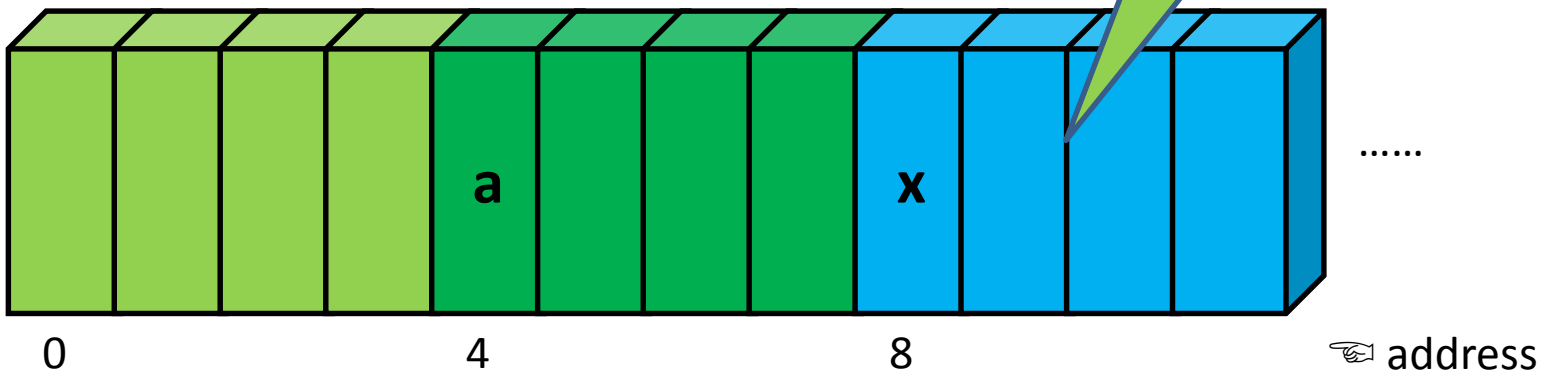
ポインタ　～前回のおさらい～

Memory model of computers

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

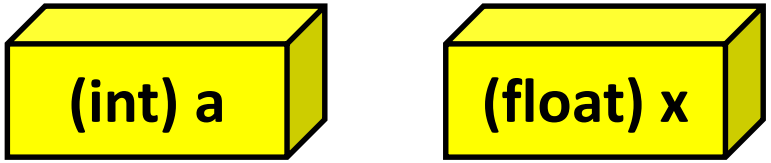


  - Actual computer memory
    - memorize by matching a variable name to an address

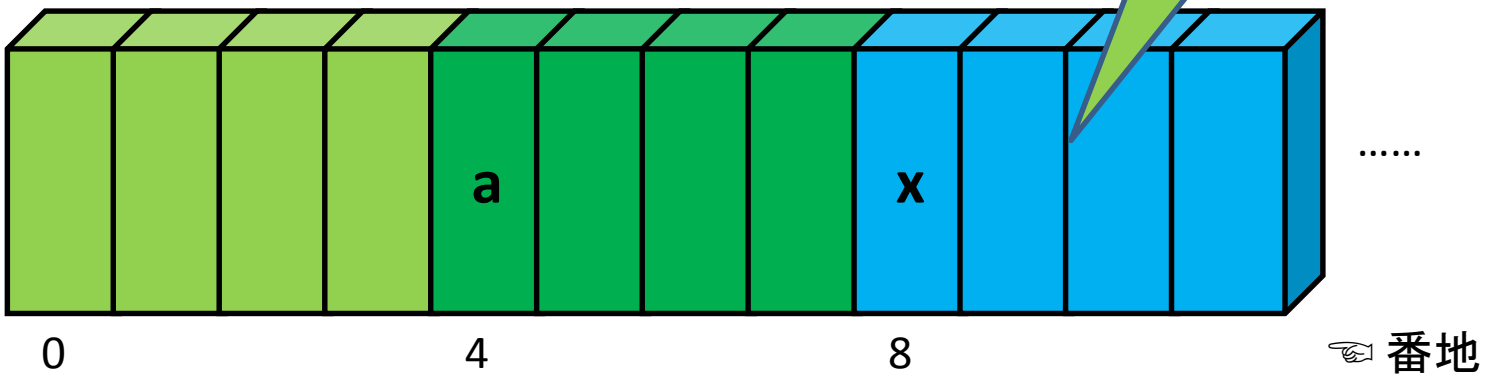


計算機のメモリモデル

- 変数の考え方
  - 箱モデル ⇒ 独立の箱に変数の名前

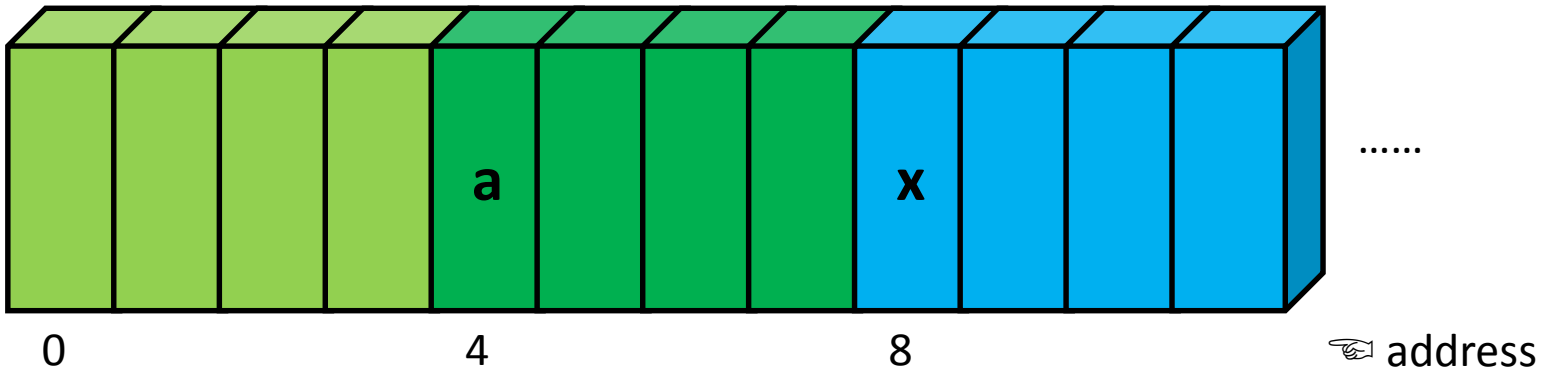


  - 実際の計算機のメモリ
    - 番地（アドレス）と変数名を対応させて記憶



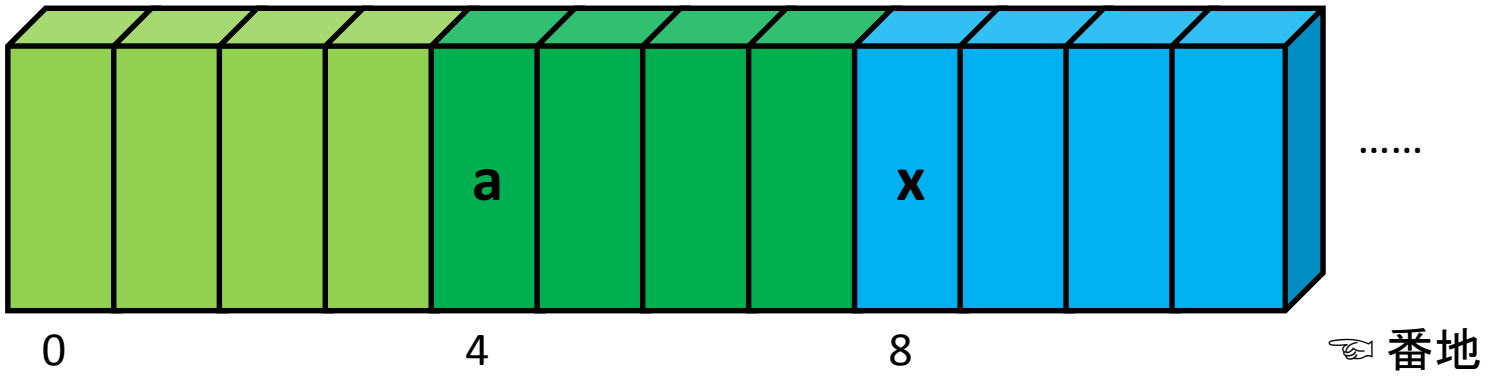
Address operator

- 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! */
```

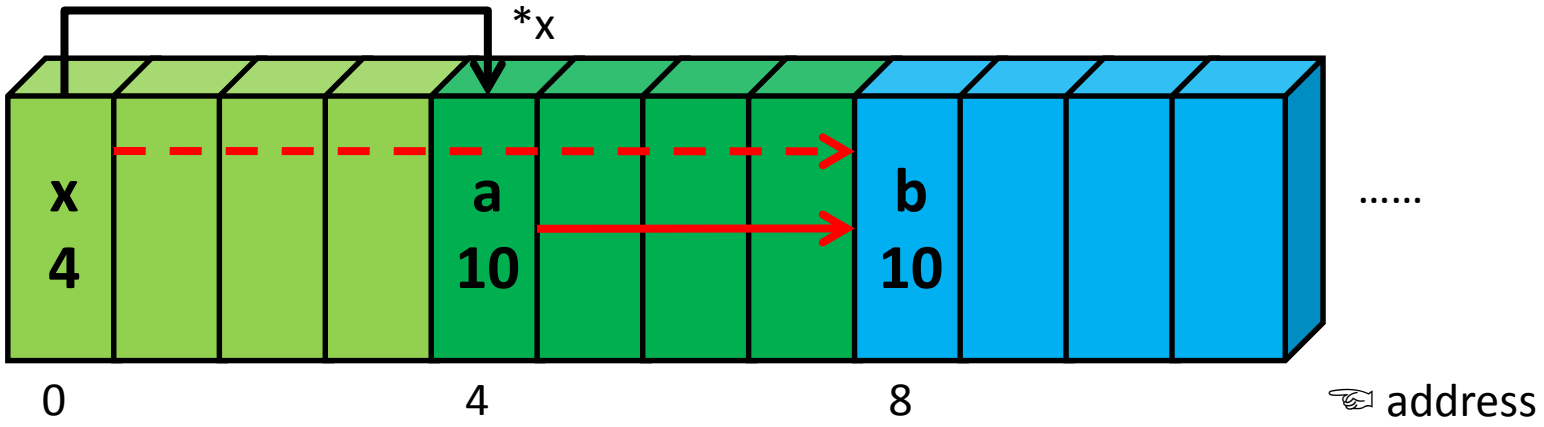
Indirection operator (2)

- 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” */
```



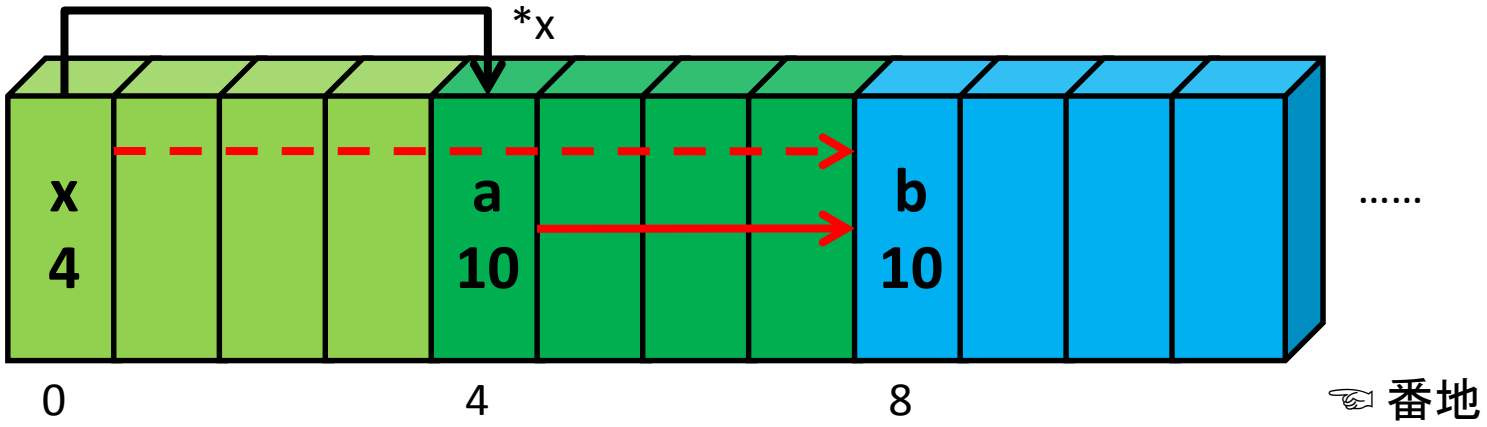
間接演算子(2)

- 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)

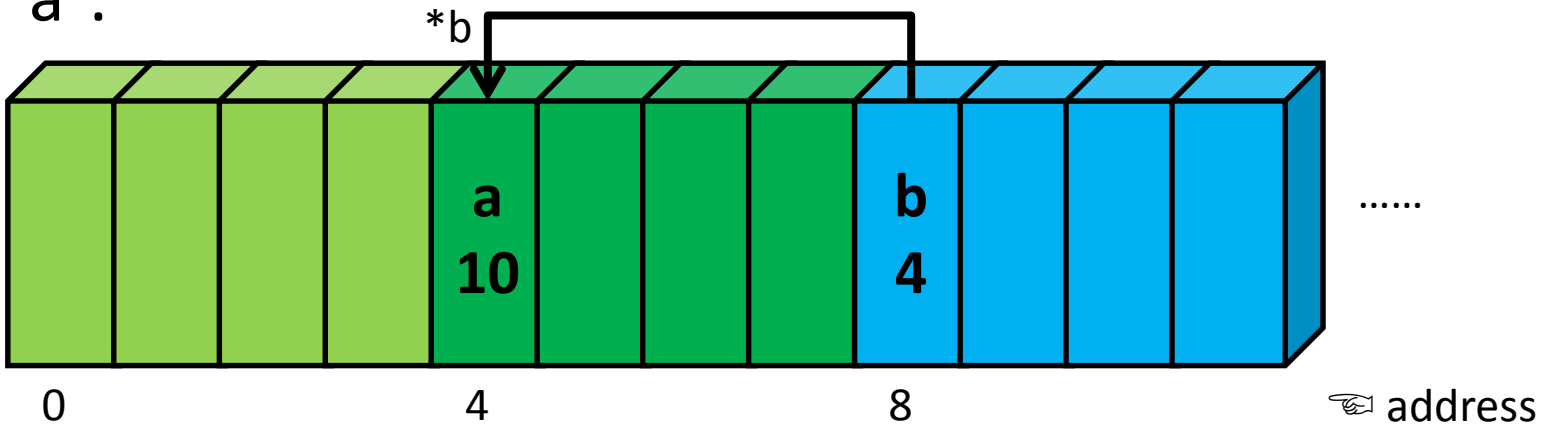
- ポインタでアドレスを渡す

```
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 */
  …
}
```

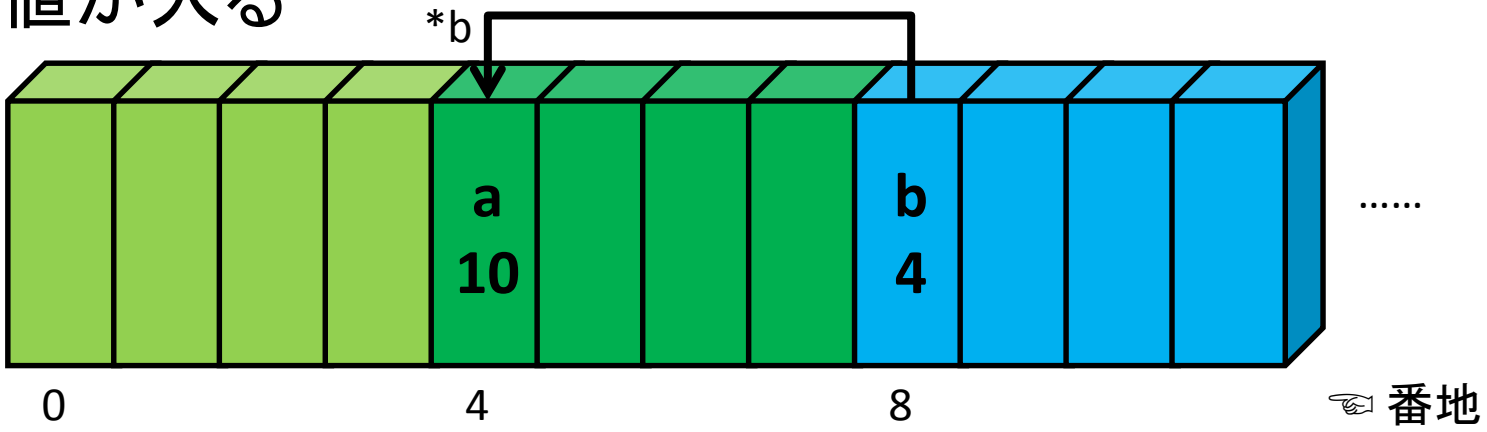
Pointer and function (3)

- 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



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
  - 例外２：“&vc” の場合
- 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

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

REVIEW END

おさらい、ここまで

Dynamic memory allocation and pointer

- Array definition
  - For example, “ int vec[20];”
  - “20 int type component array named vec”

- If automatic variable (auto), allocated in stack area automatically
  - Large array could not able to be allocated because stack area is usually not so big.
- “static” variables and file scope variables are allocated in data segment.
  - The area which invariable things while running the code is stored

- How to get large variable memory area while running the code ?

動的メモリ確保とポインタ

- 配列定義
  - 例えば、“ int vec[20];”
  - “int型の要素数20の配列vec”

- 自動変数（auto）ならstack領域に動的に配置される
  - そんなに大きくない領域なので、大きな配列を確保できなかったりする
- staticやファイルスコープ変数なら、データセグメントに配置される
  - 実行中に変化しないものが置かれる領域

- プログラム実行中に大きさの変化する大きな配列領域を確保するためには？

Function malloc()

- memory allocation function
  - Allocates memory to the program dynamically from heap area
  - void *malloc(size_t size)
    - allocates “size” byte memory in heap area, then returns a heading address of the memory area
    - Return value is a pointer of “void” type (no type). You have to give a type by casting.
      - If not, you cannot use subscript operation and pointer operation.
      - If failed to allocate, returns NULL.
    - Including of <stdlib.h> is necessary.

malloc()関数

- memory allocation（メモリ割り付け）関数
  - プログラムに対して、ヒープ領域から動的に使用メモリを割り当てる
  - void *malloc(size_t size)
    - “size” bytesの領域をヒープ領域に確保して、その先頭のアドレスを返す
    - 返り値のアドレスはvoid型（型無し）のポインタのため、キャストしてポインタの型を与える
      - 与えないと、添え字演算子やポインタ演算が動かない
      - メモリの確保に失敗すると、NULLを返す
    - <stdlib.h>のインクルードが必要

Function calloc()

- “contiguous” memory allocation function
  - Allocates memory to the program dynamically from heap area as “contiguous” area
  - void *calloc(size_t count, size_t size)
    - allocates contiguous “count” memory block of “size” bytes in heap area, then returns a heading address of the memory area.
    - Return value is a pointer of “void” type (no type). You have to give a type by casting.
      - If not, you cannot use subscript operation and pointer operation.
      - If failed to allocate, returns NULL.
    - Unlike malloc(), allocate area is initialized by zero.
    - Including of <stdlib.h> is necessary.

calloc()関数

- contiguous memory allocation（連続メモリ割り付け）関数
  - プログラムに対して、ヒープ領域から動的に使用メモリを連続領域として割り当てる
  - void *calloc(size_t count, size_t size)
    - “size” bytes の領域を、“count”個連続してヒープ領域に確保し、その先頭のアドレスを返す
    - 返り値のアドレスはvoid型（型無し）のポインタのため、キャストしてポインタの型を与える
      - 与えないと、添え字演算子やポインタ演算が動かない
      - メモリの確保に失敗すると、NULLを返す
    - malloc() と異なり、確保した領域は0で初期化される
    - <stdlib.h>のインクルードが必要

Function free()

- Free the allocated memory

- The memory address allocated by malloc() or calloc() will be used by assigning to a pointer variable.
- If you overwrite the pointer, you will lose the allocated memory address.
- C language has no garbage collection. If you don't free the memory, it will come short of memory (heap area) soon.

- Free the area properly if you finish to use it !
- void free(void *ptr)

free() 関数

- 確保したメモリを解放する関数

- malloc()やcalloc()で確保したメモリのアドレスは、ポインタ変数に入れて使う
- ポインタ変数を上書きしてしまうと、確保したメモリのアドレスが分からなくなってしまう
- C言語にはガベージコレクションがないので、そのうちメモリ（ヒープ領域）が足りなくなる

- 使い終わったら、きちんと解放しましょう！
- void free(void *ptr)

How to use

```
int *ptri;
int size = 100;

ptri = (int*)malloc(size * sizeof(int)); // area for 100 int type
if (ptri == NULL) {
  perror( “malloc failed” ); // If failed to allocate
}

free(ptri);
```

```
double *ptrd;
int size = 100;

ptrd = (double*)calloc(size , sizeof(double)); // 100 double type
if (ptrd == NULL) {
  perror( “calloc failed” ); // If failed to allocate
}

free(ptrd);
```

こんな感じで使う

```
int *ptri;
int size = 100;

ptri = (int*)malloc(size * sizeof(int)); // int型100個分の領域
if (ptri == NULL) {
  perror(“メモリの確保に失敗”); // メモリの確保に失敗した場合
}

free(ptri);
```

```
double *ptrd;
int size = 100;

ptrd = (double*)calloc(size , sizeof(double)); // double型100個分
if (ptrd == NULL) {
  perror(“メモリの確保に失敗”); // メモリの確保に失敗した場合
}

free(ptrd);
```

Dynamic memory allocation of multi-dimensional array (1)

- Common array : 1 dimension

- Use 1-dimensional array like 2-dimensional
  - Example
  - int mat[12]; intension: 4 x 3 array
  - mat[n * 3 + m] access to (n+1)th column, (m+1)th row

- Memory allocation is the same as the before examples.

- However ……

多次元配列の動的確保（１）

- 普通の配列：１次元
- １次元配列を２次元配列のように使う
  - 例
  - int mat[12];　　4x3の行列のつもり
  - mat[n * 3 + m]　(n+1)行(m+1)列
- メモリの確保は先ほどの例と同じ
- しかし……

Dynamic memory allocation of multi-dimensional array (2)

- We want to write like 2-dimensional array
  - like mat[n][m] for N-column and M-row matrix
  - Easy to understand visually

- "mat[n][m]" means "accessing to a matrix component of (n+1)th column and (m+1)th row"
- What "mat[n]" means ?
  - Array of "pointers to 1-dimensional array having M components" having N components
  - Allocate an array of pointers dynamically for this
- What "mat" means ?
  - Pointer to "array of pointers having N components"
  - Pointer of pointers !

多次元配列の動的確保（2）

- ２次元配列らしく書きたい
  - N行M列の行列で、mat[n][m] というように
  - こっちの方が視覚的に分かり易い

- mat[n][m]は(n+1)行(m+1)列の行列要素へのアクセス
- mat[n]は？
  - 「要素M個の1次元配列へのポインタ」の配列（要素N個）
  - ポインタ配列を動的確保すれば良い
- matは？
  - 「要素N個のポインタ配列」へのポインタ
  - ポインタのポインタ！

Dynamic memory allocation of multi-dimensional array (3)

```
int **ptri;
int col = 200, row = 100, i;

ptri = (int**)malloc(col * sizeof(int*));
                                    // 200 pointers of int type
if (ptri == NULL) {
  perror( “failed to allocate” ); // If failed to allocate
}
for (i = 0; i < col; i++) {
  ptri[i] = (int*)malloc(row * sizeof(int));
                                          // 100 int type
  if (ptri[i] == NULL) {
    perror( “failed to allocate” );
  }
}
```

多次元配列の動的確保(3)

```
int **ptri;
int col = 200, row = 100, i;

ptri = (int**)malloc(col * sizeof(int*));
                                    // int型のポインタ200個分の領域
if (ptri == NULL) {
  perror("メモリの確保に失敗"); // メモリの確保に失敗した場合
}
for (i = 0; i < col; i++) {
  ptri[i] = (int*)malloc(row * sizeof(int));
                                          // int型100個分の領域
  if (ptri[i] == NULL) {
    perror("メモリの確保に失敗");
  }
}
```

Dynamic memory allocation of multi-dimensional array (4)

- Free memory in reverse order

```
for (i = 0; i < col; i++) {
  free( ptri[i] );
}
free( ptri );
```

- If you would free “ptri” before “ptri[i]”, you cannot access “ptri[i]” just after freeing “ptri” and cannot free all the allocated memory.

- The same idea for more than 3-dimensional array

多次元配列の動的確保(4)

- メモリの解放は、確保と逆の順序で

```
for (i = 0; i < col; i++) {
  free( ptri[i] );
}
free( ptri );
```

- “ptri”を先に解放すると、解放した時点で“ptri[i]”にアクセスできなくなるので、全てのメモリを解放することができなくなる

- 3次元以上の配列でも、考え方は同じ

Recommendation of functionize and macronize

- You must not omit error processes when using malloc() / calloc().
  - Using memory which cannot be allocated causes “Segmentation fault” or “Bus error”
- However, writing “if (ptr == NULL) ...” for all the allocation bothers you and looks bad.

- Good to functionize or micronize the memory allocation process including error process.

関数化・マクロ化の勧め

- malloc() / calloc() を使うときに、エラー処理を省略してはいけない
  - 確保できてないメモリを使おうとすると、Segmentation faultかBus errorになる
- しかし、いちいち”if (ptr == NULL) …”と全部に書いていくのも面倒だし、見栄えも悪い

- メモリがきちんと確保できたかどうかの判定まで含めて、関数化・マクロ化しておくと楽

Extension in C99 standard (1)

- “ISO C90 standard”
  - the same as “ANSI C89”, so-called “ANSI C”
- New standard “ISO C99”

```
int func(int size, int n, int m, double mat[n][m])
{
  double newmat[size][size];
  // Comment
  ...
}
```

  - We can do like such above things.
    - Notice: we cannot use this by C90 compilers.

C99での拡張（1）

- 「ISO C90」という規格
  - ANSI C89。俗に言う「ANSI C」
- 新しい規格「ISO C99」

```
int func(int size, int n, int m, double mat[n][m])
{
  double newmat[size][size];
  // コメント
  ...
}
```

  - のようなことができるようになった
    - C90ではこれはできないので注意

Extension in C99 standard (2)

- You can write variable definitions among executable statements after C99.
  - The following code causes a compile error in C90.

```
int n; // variable definition

n = 0; // executable statement

int m; // variable definition
```

- Newer standard “C11”
  - The newest GCC follows most of this standard.

C99での拡張(2)

- ブロックの先頭部分以外(実行文の途中)に変数宣言があってもエラーにならないのは、C99以降
  - 下のような書き方は、C90ではエラー

```
int n; // 変数定義

n = 0; // 実行文

int m; // 変数定義
```

- C11というさらに新しい規格がある
  - 最新版のGCCなら、この企画にかなり追随している

“Array of pointers”
“Pointer to array”

- Array of pointers
  - Making array with “pointer” objects
  - For example, “int *p[10];”
    - 10-component array of pointers to int type

- Pointer to array
  - Pointer to “whole” array
  - For example, “int (*p)[10];”
    - A pointer to an 10-component array of int type

「ポインタの配列」「配列へのポインタ」

- ポインタの配列
  - ポインタ変数をオブジェクトとして、配列を作る
  - 例えば、”int *p[10];”
    - int型のポインタ変数の配列（要素数10）

- 配列へのポインタ
  - 配列全体へのポインタ
  - 例えば、”int (*p)[10];”
    - int型の配列（要素数10）へのポインタ

What's different ?

- In the case of "int *p[10]"
  - 10 pointers for int type will be prepared as an array.
  - Each component of the array will be a pointer for int type
  - "(p + 1)" points the address of "p[1]".

- In the case of "int (*p)[10]"
  - Variable is only one (p).
  - "(p + 1)" points the address of "p + 10 * sizeof(int)".
  - Pointer size is different from the above case.

何が違うねん！

- “int *p[10]” の場合
  - int型のポインタ変数が10個分配列として用意される
  - 配列の各要素がint型のポインタ変数
  - “(p + 1)” は “p[1]” のアドレスを指す

- “int (*p)[10]” の場合
  - 用意されるポインタ変数は1個（p）だけ
  - “(p + 1)” は “p + 10 * sizeof(int)” のアドレスを指す
  - ポインタ1つ分のサイズが違う

For what ?

- Scanning “column” for 2-dimensional array

```
int n = 4, m = 3;
int a[4][3] = {{1,2,3}, {4,5,6}, {7,8,9}, {10,11,12}};
                                                  // 4x3 matrix
int (*p)[3] = a;   // pointer to “row”

printf( “(*p)[2] = %d¥n”, (*p)[2]); // will be “3”

p += 1; // p becomes &a[1] (p forward “1 row”)
printf( “(*p)[2] = %d¥n”, (*p)[2]); // will be “6”

p += 1; // p becomes &a[2] (p forward “1 row”)
printf( “(*p)[2] = %d¥n”, (*p)[2]); // will be “9”
```

どう使えるのか

- ２次元配列で行の要素だけ走査する

```
int n = 4, m = 3;
int a[4][3] = {{1,2,3}, {4,5,6}, {7,8,9}, {10,11,12}};
                                                    // 4x3の行列
int (*p)[3] = a;    // 列に対するポインタ

printf( “(*p)[2] = %d¥n”, (*p)[2]); //  “3” になる

p += 1; // p は &a[1] になる（列１個分アドレスが進む）
printf( “(*p)[2] = %d¥n”, (*p)[2]); //  “6” になる

p += 1; // p は &a[2] になる（列１個分アドレスが進む）
printf( “(*p)[2] = %d¥n”, (*p)[2]); //  “9” になる
```

Pointer to function

- Treat a function as a pointer
- When running a “functional” function with different criterions, preparing functions according to the criterions is inefficient.
  - The criterion is not always fixed and unknown sometimes.
- Work the functional by preparing a function of the “criterion” and passing the pointer of function
- Understand ? ⇒ Example

関数ポインタ

- 関数をポインタとして扱う
- 同じ機能の関数を異なる基準で動かしたい場合に、基準毎に同じ機能の関数を複数用意するのは非効率
  - 基準が固定されていれば良いが、不明な場合もある
- その「基準」を関数として用意し、そのポインタを渡すことで、その機能を動作させる
- 意味が分からん？ ⇒ 具体例

Example of pointer to function

- UNIX system standard sort function
  - Example: quick sort
  - void qsort(void *base, size_t nel, size_t width,
    int (*compar)(const void*, const void*));

  - “compar” is a pointer to comparing function
  - Comparing the data is necessary when sorting. However, a system cannot know a kind of data and decide the comparing method previously.
  - User prepares data and a function to compare the data, and passes “a pointer to the comparing function” to the sorting function.

- Useful when making a library function changing the comparing criterion by a kind of data

関数ポインタの実例

- システム標準のソート関数
  - 例：クイックソート
  - void qsort(void *base, size_t nel, size_t width,
    int (*compar)(const void*, const void*));
  - “compar”が関数ポインタ
  - ソートするときに、どちらが大きいか比較しないといけないが、ソートするデータが何か分からないので、比較方法を予めシステムで決めておくことができない
  - ソートするデータと一緒に、大きさの決め方をユーザが「関数」として用意して、関数ポインタとしてソート関数に渡す
- データによって評価基準が変化するようなアルゴリズムをライブラリ化するときに便利