物理学情報処理演習

7-1. C言語の基礎 その4

ポインタと配列

文字列

参考文献

“明解C言語　入門編”

ソフトバンククリエイティブ　柴田望洋

“プログラミング言語C 第２版”

共立出版　B. W. カーニハン／D. M. リッチー

大久保晋

E-mail: buturi-johoshori@tiger.kobe-u.ac.jp

http://extreme.phys.sci.kobe-u.ac.jp/staffs/okubo/lectures/Programming/index.html

オブジェクト

- 変数：値を記憶するための箱
  - 例

```
char c;        /* char型の値 = 文字 */
int i;         /* int型の値 = 整数 */
```

- 値を記憶するための箱である変数は、メモリ上に存在する　→　オブジェクト
  - オブジェクトは、大きさとその存在位置を示すアドレスを持つ
  - 大きさは sizeof 演算子で知ることができる

```
sizeof(char): char型の大きさ = 1バイト
int i;
sizeof i; オブジェクト i の大きさ = 4バイト
```

ポインタ

- ポインタとは、メモリ上のアドレスを指し示している変数
- ポインタの定義
  - 例

```
char*    p_char;        /* char へのポインタ */
int*     p_int;         /* int へのポインタ */
```

- 単項演算子&は、オブジェクトのアドレスをl示す
  - 例

```
char  moji = 'X';       /* char型の変数 */
p_char = &moji;         /* p_charはmojiのアドレスを示す */
```

- 単項演算子 * は、ポインタの指し示しているオブジェクトを示す
  - 例

```
char  komoji;
komoji = tolower(*p_char);
```

関数の引数

- Cでは、関数は”値渡し”
  - 呼び出した側の引数に入っている値が、呼び出された側の引数に渡される
  - 例

```
double x = 1.0;
func(x);        /* 関数の呼び出し xの値を渡す*/

void func(double y)
{
  y = y + 5.0;
}
```

呼び出した側の引数 x に入っている値”1.0”が、呼び出された側の引数 y の値となる。

呼び出された側の変数 y の値を変更しても、呼び出した側の変数 x は変化しない

関数の引数

- アドレス渡し
  - 呼び出した側の引数としてポインタを指定すれば、ポインタの"値"つまりアドレスが、呼び出された側の引数に渡される
  - これを使うと、関数の処理によって変数の値を変えることができる。
  - 例

```
double x = 1.0;
func(&x); /* 関数の呼び出し x のアドレスを渡す */

void func(double *y)
{
  (*y) = (*y) + 5.0;
}
```

呼び出し側の引数 x のアドレスが、呼び出された側の引数であるポインタ変数 y の値となる。

呼び出された側で、y の指し示す変数の値を変更しているので、呼び出し側の変数 x も変化する。

演習7-1-1: A その１(次頁に続く)

- 以下の様に、memtest1.c を作成し、実行してみよう

```
#include <stdio.h>
int main(void)
{
    int         x = 10;
    int         y = 99;
    int*        ptr;

    /* print out x */
    printf (“x: address [%p] value = %d ¥n”, &x, x);
    /* print out y */
    printf (“y: address [%p] value = %d ¥n”, &y, y);

    /* set ptr to be address of x */
    printf (“ptr points address of x ¥n”);
    ptr = &x;            ポインタ ptr に x のアドレスを代入
    printf (“ptr = %p value = %d ¥n”, ptr, *ptr);
```

演習7-1-1: A その２(前頁から続く)

```
/* set pt to be address of y */
printf (“ptr points address of y ¥n”);
ptr = &y; ポインタ ptr に y のアドレスを代入
printf (“prt = %p value - %d ¥n”, ptr, *ptr);

/* change the value pointed by ptr */
printf (“an object which ptr points is changed to 111
¥n”);
*ptr = 111;        ポインタ ptr の示す先に値を代入する
printf (“prt = %p value = %d ¥n”, ptr, *ptr);
printf (“x: address [%p] value = %d ¥n”, &x, x);
printf (“y: address [%p] value = %d ¥n”, &y, y);

return (0);

}
```

演習7-1-1: A 実行結果(PPC G5 OSX)

```
x: address [0xbffffc5c] value = 10
y: address [0xbffffc60] value = 99
```

ポインタ ptr に x のアドレスを代入

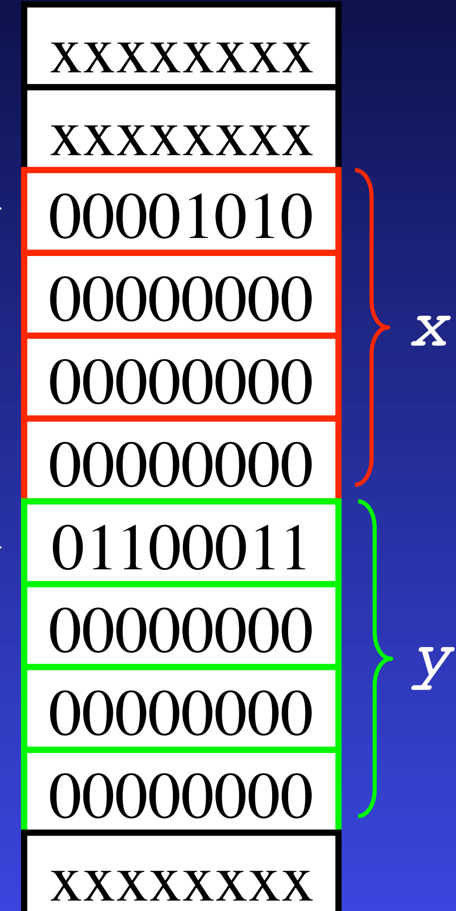
```
ptr points address of x
ptr = 0xbffffc5c value = 10
```

ポインタ ptr に y のアドレスを代入

```
ptr points address of y
prt = 0xbffffc60 value = 99
```

ポインタ ptr の示す先に値111を代入する

```
an object which ptr points is changed to 111
prt = 0xbffffc60 value = 111
x: address [0xbffffc5c] value = 10
y: address [0xbffffc60] value = 111
```



演習7-1-1: B

2つの関数 func1 と func2

```
double func1(double y)
{
  y = y + 5.0;
  return y;
}
```

```
double func2(double* y)
{
  (*y) = (*y) + 5.0;
  return *y;
}
```

の違いを見てみよう。

```
int main(void)
{
  double x = 10.1;
  /* print out x */
  printf (“x: address [%p] value = %f ¥n”, &x, x);

  /* use func1 */
  printf (“func1(x) = %f ¥n”, func1(x));
  printf (“x: address [%p] value = %f ¥n”, &x, x);

  /* use func2 */
  printf (“func2(x) = %f ¥n”, func2(&x));
  printf (“x: address [%p] value = %f ¥n”, &x, x);

  return (0);
}
```

型と数値表現

- 計算機上では全ての情報は２進数で表される
  - Bit: 0 or 1
  - Byte: 8 bit 0 ~ 255
  - Word: 16 bit = 2 byte 0 ~ 65535
  - Long Word: 32 bit = 4 byte 0 ~ 4294967295

- 変数：「情報」を示すもの
  - 名前：情報を区別するためのもの
  - 型：情報である２進数の長さとその解釈の仕方を規定
  - アドレス：情報の格納位置

scanf

- 整数変数 i に標準入力から代入するときに

```
int i;
scanf("%d", &i);
```

とした。これは scanf に渡されたアドレスに

"%d" で指定された型、すなわち整数型の変数があり、標準入力から読み取った値を"整数値"として解釈して２進数に変換して &i で指定されたアドレスにその値をいれよ

ということを意味している。

よって宣言された変数の型と、scanf 文中の format ("" でくくられたもの)が一致していなければならない

整数型	int	%d
倍精度実数型	double	%lf (lが倍精度、fが実数)
文字型	char	%c
文字列	char*	%s

例

キーボードから"4"を入力したとき

scanf("%d", &i):整数値4→00000100

scanf("%c", &c):文字４→00110100

と変換される

整数表現

- 厳密な表現
- 表現範囲（32bitの場合）
  - 符号付き -2,147,483,658 ~ 2,147,483,657
  - 符号なし 0 ~ 4,294,967,295

- 例：（符号付き）

10進数	16進数	2進数
256(= 2^8)	0000 0000	0000 0000 0000 0000 0000 0001 0000 0000 0000
$2^{31}-1$	7FFF FFFF	0111 1111 1111 1111 1111 1111 1111 1111 1111
-1	FFFF FFFF	1111 1111 1111 1111 1111 1111 1111 1111 1111
-2^{31}	1000 0000	1000 0000 0000 0000 0000 0000 0000 0000 0000

整数表現：演算

- 符号つき整数表現： 2の補数
  - 例：8 bit 整数 -256 ～ +255
    - 正の数：MSB = 0, (下 7 bit)
    - 負の数：MSB = 1, (下 7 bit) -256
  - 例：

    255(=2^8 -1) 0111 1111

    -1 1111 1111
  - 例：2+(–3) = -1 12+(-4) = 8

2	0000 0010	12	0000 1100
+)-3	1111 1101	+)-4	1111 1100
-1	1111 1111	8	0000 1000

  - 例：(–3)x 2 = -6 (-2)x(-1) = 2

-3	1111 1101	-2	1111 1110
x) 2	0000 0010	x) -1	1111 1111
-6	1111 1010	2	0000 0010

実数表現

- 浮動小数点表示

  s x M x B^{e-E}

  s:符号ビット（x or -）

  M:仮数（正の整数）

  B:基底（=2）

  e:指数（符号なし整数）

  E:指数のゲタ

  近似値としての表現

  例：

  (E=151) s e M(23bits)

  1.0 0 1000 0001 100 0000 0000 0000 0000 0000

  -0.5 1 1000 0000 100 0000 0000 0000 0000 0000

  10-7 0 0011 1001 110 1011 0101 1111 1100 1010

文字表現

- 文字も２進数表現される
  - ASCIIコード

    １バイトを１文字に当てはめ、アルファベット、数字、記号（制御文字）を表現する

    例：　文字　値

    A　0x41

    1　0x31

    CR(改行)　0x0a

  - Shift-JISコード
  - EUCコード
  - UNICODE

    ２バイトを１文字に当てはめ

    漢字（全角かな、記号を含む）を表現する

例：文字　値	SJIS	EUC	UNICODE
神	0x905f	0xbfc0	0x795e
戸	0x8ccb	0xb8cd	0x6238
１（全角）	0x8250	0xa3b1	0xff11

配列

- 配列の宣言

  例

```
char array[128]; /* 128個の文字からなる配列 */
```

- 配列を示す変数には、配列先頭アドレスが入っている
  - 配列を示す変数は、ポインタとしても使用できる
    - 例： moji1, moji2 は同じ文字となる

```
char moji1, moji2;
moji1 = array[0]; /* 0番目の文字 */
moji2 = *array;   /* 配列の先頭の文字 */
```

    - 例： moji1, moji2 は同じ文字となる

```
char moji1, moji2;
int i;
moji1 = array[i]; /* i番目の文字 */
moji2 = *(array+1); /* 配列の先頭からi番目の文字
*/
```

配列：関数の引数として配列を使う方法

- 関数の実装部でも、引数が配列となることを”[ ]”で表します
- 大きさは宣言していませんが、添字をつけて配列中の要素を参照できます。
  - 配列の先頭アドレスを”値”で渡したのであり、そのアドレスから配列の各要素にアクセスできる
- 関数内で配列の要素が変更された場合、関数から戻ってもその変更は有効です。
  - 例：以下のようなプログラムを実行すると”a[0]=0”と表示されます。

```
void function(int a[]);

int main(void)
{
  int a[10];
  a[0]=5;
  function(a);
  printf(“a[0]=%d ¥n”,a[0]);
  return (0);
}

void function(int a[])
{
  a[0]=0;
}
```

配列

- 配列とポインタの違い
  - ポインタを宣言したときは、
    - アドレスの値を入れる　メモリ領域　が確保される
  - 配列を示す変数は、ポインタとしても使用できる
    - 配列全体を格納する　メモリ領域　が確保される　と共に
    - アドレスの値を入れる　メモリ領域　が確保され
    - 配列の　先頭アドレス　がセットされる

文字列

- N個の文字からなる文字列は、N+1個の大きさの文字配列
- 末尾の文字は、NULL文字（'¥0'）
  - 例

```
const char* hello="HELLO!";
```

address	value						
&hello	'H'	'E'	'L'	'L'	'O'	'!'	'¥0'

演習7-1-2:

A) 文字列の構造を確認しよう

```
void dumpstring(const char str1[], int n_size)
{
    int i, length;

    length = strlen(str1);    文字列の長さ
    printf(“ size = %d  length = %d ¥n”, n_size, length);

    for (i=0; (i<n_size)||(i<length); ++i){
      printf(“%d: address [%p]  char=%c  value = %d
         %x ¥n”, i, &(str1[i]), str1[i],
         (int)(str1[i]), (int)(str1[i]) );
    }  i番目の文字のアドレス、文字、値
}
```

演習7-1-2:

２つの文字列で試してみよう

```
#include <stdio.h>
#include <string.h>        文字列を扱う関数ライブラリ

void dumpstring(const char str[], int size);

#define SIZE_S 8

int main(void)
{
    const char* cs = “Hello !”;        文字列 cs
    char      s1[SIZE_S] = “Ciao !”;        文字列 s1

    /* dump string of ‘cs’ */
    printf(“ cs = %s ¥n”, cs);
    dumpstring( cs, SIZE_S);
    /* dump string of ‘s1’ */
    printf(“ s1 = %s ¥n”, s1);
    dumpstring( s1, SIZE_S);

    return (0);
}
```

演習7-1-2: 実行結果 例(PPC G4 on OSX)

cs = Hello ! 　　文字列 cs
size = 8 length = 7
0: address [0x2f90] char=H value = 72 48
1: address [0x2f91] char=e value = 101 65
2: address [0x2f92] char=l value = 108 6c
3: address [0x2f93] char=l value = 108 6c
4: address [0x2f94] char=o value = 111 6f
5: address [0x2f95] char= value = 32 20
6: address [0x2f96] char=! value = 33 21
7: address [0x2f97] char= value = 0 0
s1 = Ciao ! 　　文字列 s1
size = 8 length = 6
0: address [0xbffffd48] char=C value = 67 43
1: address [0xbffffd49] char=i value = 105 69
2: address [0xbffffd4a] char=a value = 97 61
3: address [0xbffffd4b] char=o value = 111 6f
4: address [0xbffffd4c] char= value = 32 20
5: address [0xbffffd4d] char=! value = 33 21
6: address [0xbffffd4e] char= value = 0 0
7: address [0xbffffd4f] char= value = 0 0

0x2f90→

0x48	→ H
0x65	→ e
0x6c	→ l
0x6c	→ l
0x6f	→ o
0x20	→ （空白）
0x21	→ !
0x00	→(NULL)

文字列処理

文字列を処理するには、標準ライブラリの<string.h>に定義してある文字列処理ルーチンを使う

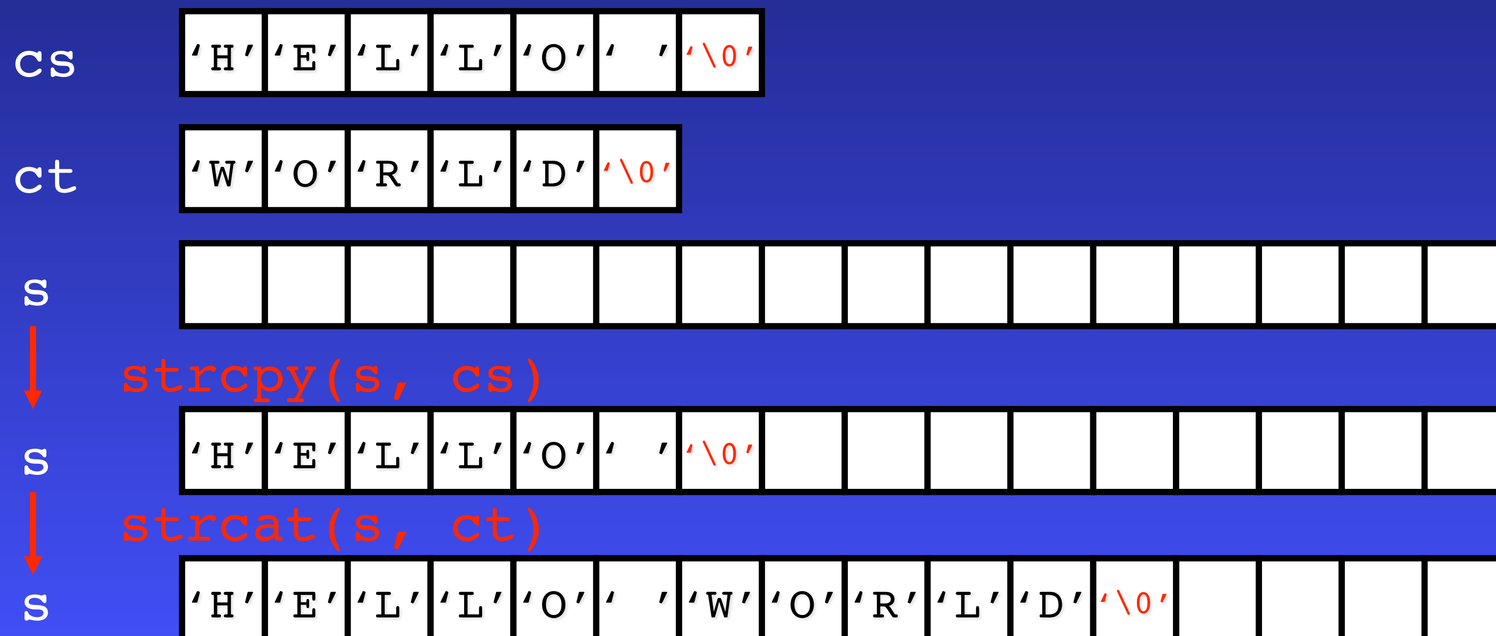
char* strcpy(char* s, const char* ct)	NULL文字を含めて文字列ctをsにコピーし、sを返す
char* strncpy(char* s, const char* ct, int n)	文字列ct内n文字をsにコピーし、sを返す。ctがn文字より少ないときはNULL文字をつめる
char* strcat(char* s, const char* ct)	文字列ct をsの終わりに連結し、sを返す
char* strncat(char* s, const char* ct, int n)	文字列ct内の最大n文字をsの終わりに連結し、sを返す
int strcmp(const char* cs, const char* ct)	文字列csと文字列ctを比較する
char* strchr(const char* cs, char c)	文字列csの中de文字cが最初に洗われる位置へのポインタ。なければNULLを返す
char* strstr(const char* cs, const char* ct)	文字列csの中de文字列ctが最初に現れる位置へのポインタ。なければNULLを返す
size_t strlen(const char* cs)	文字列csの長さを返す
char* strtok(char* s, const char* ct)	文字列sの中で文字列ctによって区切られるトークンを返す

文字列処理

strcat, strcpyは、結果の文字列が入るメモリ領域を確保しておくことを忘れずに！

例

```
const char*     cs = "HELLO "
const char*     ct = "WORLD!"
char s1[16];
strcpy(s1, cs);
strcat(s1, ct);
```



演習7-1-3　文字列操作

文字列の操作をしてみよう

```
#define SIZE_S 16
int main(void)
{
        const char* cs = "Hello ";
        const char* ct = "World!";
        char        s1[SIZE_S];

        /* copy cs to s1 */
        strcpy(s1, cs);      文字列のコピー

        /* dump string of 's1' */
        printf(" s1 = %s \n ", s1);
        dumpstring( s1, SIZE_S);

        /* concatnate ct to s1 */
        strcat(s1, ct);      文字列の連結

        /* dump string of 's1' */
        printf(" s1 = %s \n ", s1);
        dumpstring( s1, SIZE_S );

        return (0);
}
```

ポインタの配列・多次元配列

複数の文字列をつくるには、

- （文字の）多次元配列

```
char array[A_SIZE][STR_SIZE]
```

- （文字への）ポインタからなる配列

```
char array[A_SIZE]
```

- （文字への）ポインタへのポインタ

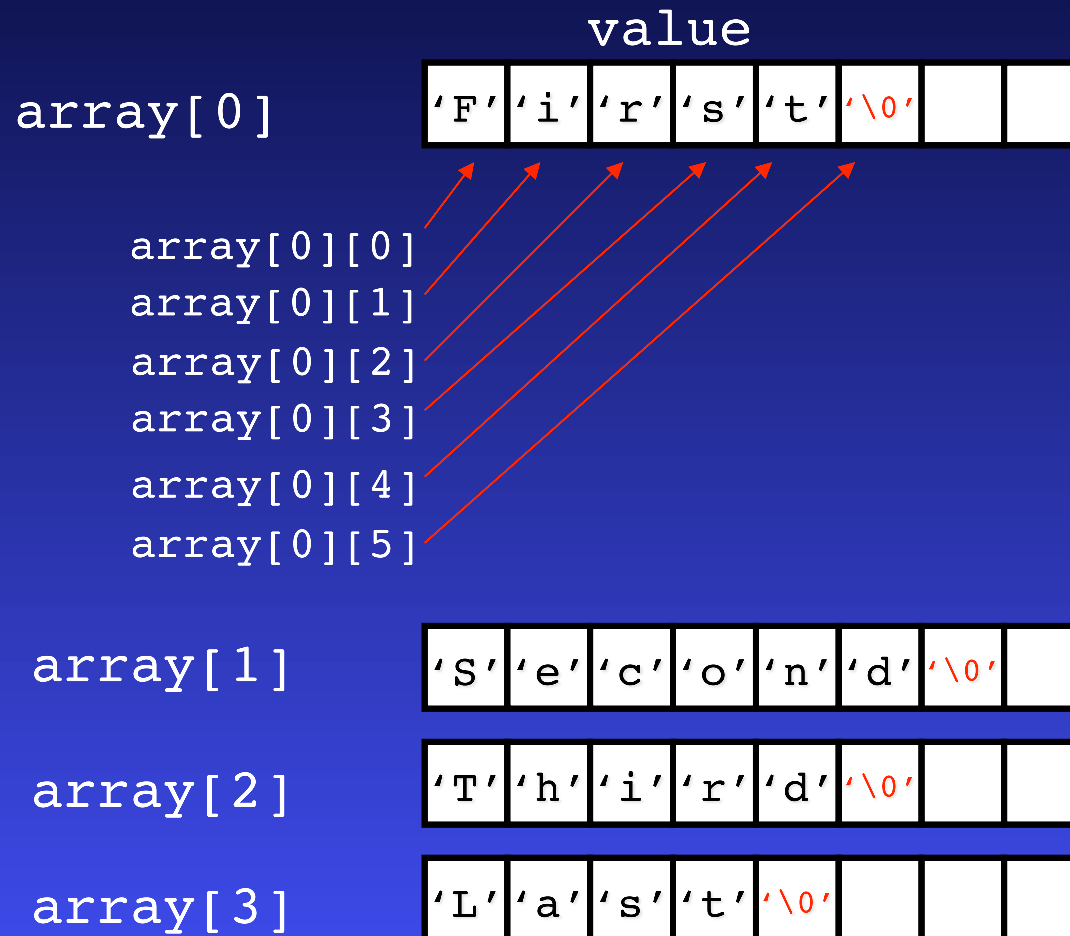
```
char** array
```

が使用できる

多次元配列

例

```
char array[4][8] = { "First", "Second", "Third", "Last" };
```



ポインタの配列

例

```
char* array[] = { "First", "Second", "Third", "Last" };
```

	address	value
array	&array	n0
array[0]	n0	m1
array[1]	n0+off	m2
array[2]	n0+2*off	m3
array[3]	n0+3*off	m4

アドレスが参照するのは
配列の先頭

m1	'F'	'i'	'r'	's'	't'	'\0'		
m2	'S'	'e'	'c'	'o'	'n'	'd'	'\0'	
m3	'T'	'h'	'i'	'r'	'd'	'\0'		
m4	'L'	'a'	's'	't'	'\0'			

演習7-1-4：ポインタの配列・多次元配列 その１

A) 文字の二次元配列を作ってみよう
（関数dumpstringは演習7-2と同じ）

```
#define SIZE_S 16
#define SIZE_A 4

int main(void)
{
  char  array[SIZE_A][SIZE_S] = { “First”, ”Second”, “Third”, ”Last”};
  int i;

  /* dump string of ‘s ’ */
  for (i=0; i<SIZE_A; ++i){
    printf(“ array[%d] = %s \n”, i, array[i]);
    dumpstring( &(array[i][0]), SIZE_S);
  }

  return (0);
}
```

演習7-1-4：ポインタの配列・多次元配列 その2

B) 文字へのポインタ配列を作ってみよう
　　(関数dumpstringは演習7-2と同じ)

```
#define SIZE_S 16
#define SIZE_A 4

int main(void)
{
  char*  array[SIZE_A] = { “First”, ”Second”, “Third”, ”Last”};
  int i;

  /* dump string of ‘s ’ */
  for (i=0; i<SIZE_A; ++i){
    printf(“ array[%d] = %s \n”, i, array[i]);
    dumpstring( array[i], SIZE_S);
  }

  return (0);
}
```

mainの関数の引数

main 関数には引数がある

```
int main ( int argc, char* argv[])
```

ここで、

argc: プログラム実行時の引数の個数

argv[]: プログラム実行時の引数（文字列）

である。

例

program という名のexecutable（実行）ファイルに、arg_1, arg_2 という引数を与えて実行したとする

```
program arg_1 arg_2
```

とすると

argc = 2

argv[0] = “program”（実行ファイル名）

argv[1] = “arg_1”（第１引数）

argv[2] = ”arg_2”（第二引数）

となる。これを使えばプログラム実行時にプログラムにパラメーターを渡すことができる。

演習7-1-5：main関数の引数

main 関数に引数を与えて、プログラム実行時に与えた文字を返すプログラムを作ってみよう。

7-1-5.c ソース

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    int i;

    printf("argc= %d¥n",argc);
    for (i=0; i<argc; i++)
    {
        printf("argv[%d]= %s ¥n", i, argv[i]);
    }

}
```

演習7-1-5：main関数の引数

7-1-5.c の動作結果

```
gcc -o 7-5 7-5.c

$./7-5 iroha ni hoheto chiri nuruo

argc= 6
argv[0]= ./7-5
argv[1]= iroha
argv[2]= ni
argv[3]= hoheto
argv[4]= chiri
argv[5]= nuruo
```

引数として入力した文字列が 返ってきた！