プログラミング基礎
第11回

藤江 真也

2021年7月2日

準　備

- ファイル(0731.tgz)をダウンロード

```
$ wget http://sites.fujielab.org/ip/files/0702.tgz
```

- ダウンロードしたファイルを展開

```
$ tar zxvf 0702.tgz
```

- 展開されたディレクトリに移動

```
$ cd 0702
```

- sqr.c などのファイルがあることを確認

```
$ ls
```


関　数

プログラム例 (1)

sqr.c

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

int main(void) {
   double d = 20.0;
   double sqd = sqrt(d);

   printf("square root of %lf = %lf¥n", d, sqd);

   return 0;
}
```

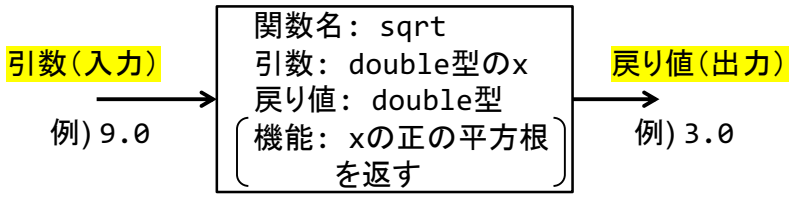
プログラム例 (1)

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

int main(void) {
   double d = 20.0;
   double sqd = sqrt(d);

   printf("square root of %lf = %lf¥n", d, sqd);

   return 0;
}
```

関数呼び出し式(function call expression)

- 関数を呼び出すとその関数の機能を実行できる
- 引数(argument)を与えることで振る舞いが変化する

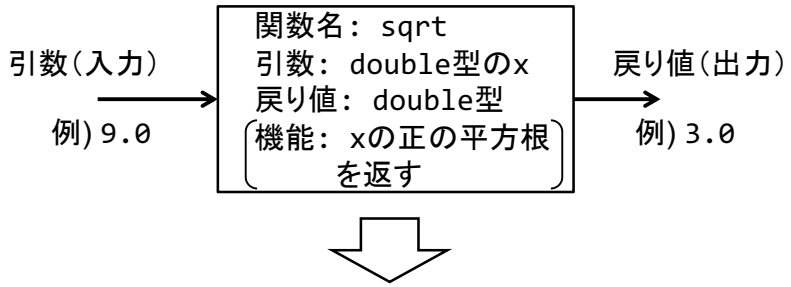
関数呼び出し



処理の詳細はわからない(ブラックボックス)
が，名前と入力と出力がわかれば呼べる(利用できる)

```
double d = 9.0;
double sqf = sqrt(d);
```


関数宣言




```
double sqrt(double x);
```

戻り値の型　関数名　引数の型，名前

カンマ区切りで複数与えることができる

プロブラム例(2)

add_i.c

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

```
int add_i(int a, int b);
```

```
int main(void) {
    int x = 10, y = 20, z;

    z = add_i(x, y);

    printf("%d + %d = %d¥n", x, y, z);

    return 0;
}
```

```
int add_i(int a, int b) {
    int c;

    c = a + b;

    return c;
}
```

プログラム例(2)

関数宣言

①**関数名**は add_i

②**引数**は int型のaと，int型のb

③**戻り値**はint型

```
#include <stdio.h>

int add_i(int a, int b);

int main(void) {
    int x = 10, y = 20, z;

    z = add_i(x, y);

    printf("%d + %d = %d¥n", x, y, z);

    return 0;
}

int add_i(int a, int b) {
    int c;

    c = a + b;

    return c;
}
```


プログラム例(2)

関数呼び出し

```
#include <stdio.h>

int add_i(int a, int b);

int main(void) {
    int x = 10, y = 20, z;

    z = add_i(x, y);

    printf("%d + %d = %d¥n", x, y, z);

    return 0;
}

int add_i(int a, int b) {
    int c;

    c = a + b;

    return c;
}
```

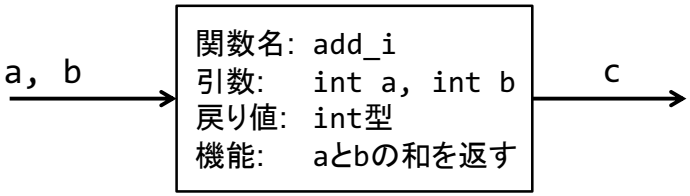

プログラム例(2)

関数定義

（実際の処理の内容）

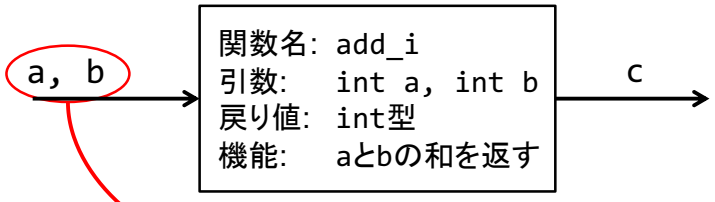
```
#include <stdio.h>

int add_i(int a, int b);

int main(void) {
    int x = 10, y = 20, z;

    z = add_i(x, y);

    printf("%d + %d = %d¥n", x, y, z);

    return 0;
}

int add_i(int a, int b) {
    int c;

    c = a + b;

    return c;
}
```


関数定義




```
int add_i(int a, int b) {
    int c;

    c = a + b;

    return c;
}
```

関数定義



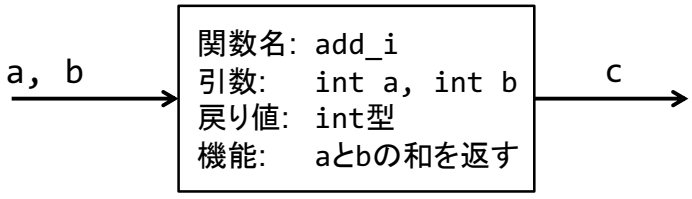

```
int add_i(int a, int b) {
   int c;

   c = a + b;

   return c;
}
```

引数として宣言されている変数は関数内でそのまま使える

初期化しなくていいの？

呼び出し時に与えられた値が代入されている状態なので必要ない！

関数定義

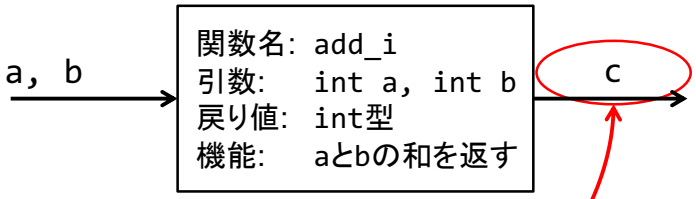



```
int add_i(int a, int b) {
   int c;

   c = a + b;

   return c;
}
```

関数内で使う変数は関数内で宣言する

新しい変数なので初期化／代入が必要

関数定義




```
int add_i(int a, int b) {
   int c;

   c = a + b;

   return c;
}
```

return文により関数の処理が終了し，指定された値が戻り値として出力される

確認

- （確認1） add_i.c の関数宣言（プロトタイプ宣言）を削除したらどうなるか？
- （確認2） add_i.cの関数宣言（プロトタイプ宣言）の(int a, int b) を (int, int) に変えたらどうなるか？
- （確認3） 同様に関数定義の (int a, int b) を (int, int) に変えたらどうなるか？
- （確認4） 変数cの名前をxに変えたらどうなるか？

（確認1）

関数は使う前に
宣言されてないといけない

```
#include <stdio.h>

int add_i(int a, int b);

int main(void) {
    int x = 10, y = 20, z;

    z = add_i(x, y);

    printf("%d + %d = %d¥n", x, y, z);

    return 0;
}

int add_i(int a, int b) {
    int c;

    c = a + b;

    return c;
}
```

こんな関数知らないよ，
ということでエラー
（実際には暗黙的宣言警告）

（確認2）

関数宣言時は
引数名を与えなくてもよい

```
#include <stdio.h>

int add_i(int a, int b);

int main(void) {
    int x = 10, y = 20, z;

    z = add_i(x, y);

    printf("%d + %d = %d¥n", x, y, z);

    return 0;
}

int add_i(int a, int b) {
    int c;

    c = a + b;

    return c;
}
```

引数は，型と数だけわかれば
（名前が違っても）呼べる

（確認3）

```
#include <stdio.h>

int add_i(int a, int b);

int main(void) {
    int x = 10, y = 20, z;

    z = add_i(x, y);

    printf("%d + %d = %d¥n", x, y, z);

    return 0;
}

int add_i(int a, int b) {
    int c;

    c = a + b;

    return c;
}
```

関数定義時は
引数名を省略できない

こんな変数知らないよ，ということで
エラーになる

（確認4）

```
#include <stdio.h>

int add_i(int a, int b);

int main(void) {
    int x = 10, y = 20, z;

    z = add_i(x, y);

    printf("%d + %d = %d¥n", x, y, z);

    return 0;
}

int add_i(int a, int b) {
    int x;

    x = a + b;

    return x;
}
```

結果は変わらない

違う関数内で
宣言された変数は
互いに影響を与えない

スコープ（scope）

- スコープ（scope）：変数の有効範囲
- 変数は，**宣言されたブロック内とそのサブブロック内でのみ有効**
  - サブブロック … ブロックの中のブロック

```
int g;
int main(void) {
    int x;
    for(...) {
    ..xのスコープ
    }
}
int func(int y) {
    int z;
    ...
}
```

```
int g;
int main(void) {
    int x;
    for(...) {
    ...
    }
}
int func(int y) {
    int z;
    y, zのスコープ
}
```

```
int g;
int main(void) {
    int x;
    for(...) {
    ...
    } gのスコープ
} （グローバル変数）
int func(int y) {
    int z;
    ...
}
```


スコープの利点

```
#include <stdio.h>
int add_i(int a, int b);
int main(void) {
    int x = 10, y = 20, z;
    z = add_i(x, y);
    printf("%d + %d = %d¥n", x, y, z);
    return 0;
}
int add_i(int a, int b) {
    int x;
    x = a + b;
    return x;
}
```

関数を呼び出す側は，
関数内でどんな変数が
使われるかわからない

関数を定義する側も，
呼び出し側でどんな変数が
使われるかわからない

スコープがないと，
プログラム全体でどんな変数名が
使われるか知らなきゃいけない

実質的に不可能！！！

引数・戻り値無しの関数

- 引数や戻り値は無くてもよい

引数や戻り値が無い場合は，**void** という特殊な型を使う

例1）引数がなくて，int型の戻り値を持つ関数 func1 の宣言

```
int func1(void);
```

例2）引数がなくて，戻り値も無い関数 func2 の宣言

```
void func2(void);
```


続・ポインタ

文字列とポインタ

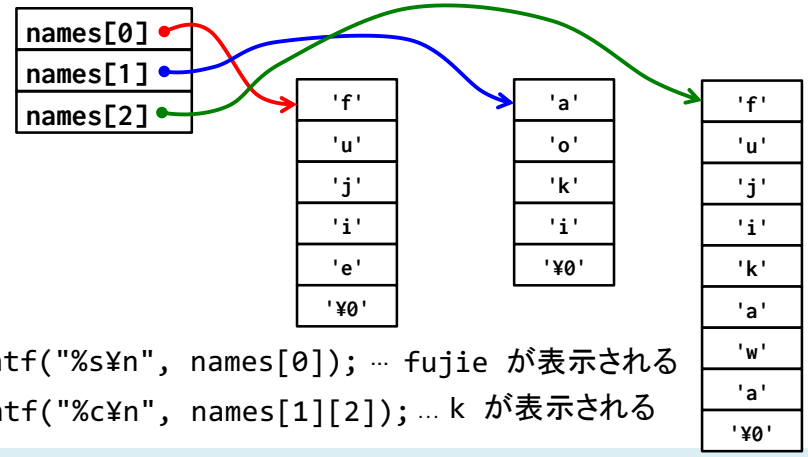
- 文字列はchar型の配列
- 配列を操作するときにはポインタをよく使う
- したがって文字列を操作するときはポインタをよく使う


```
#include <stdio.h>

int main (void) {
  char str[] = "Hello, World!";
  int i = 0;
  char *ptr = str;

  while (*(ptr + i) != '¥0') {
    printf ("%c¥n", *(ptr + i));
    i++;
  }

  return 0;
}
```

文字列は，最後の要素が
ヌル文字（'¥0'）で終えるという
きまりがあるため，それが見つかる
まで繰り返せば1文字ずつ取り出せる

ポインタ配列（文字列配列）

```
char *names[] = {"fujie", "aoki", "fujikawa"};
```



printf("%s¥n", names[0]); … fujie が表示される

printf("%c¥n", names[1][2]); … k が表示される

例2

ap.c

```
#include <stdio.h>

int main(void) {
  char *str[] = {"fujie", "aoki"};
  char *p1, *p2;
  int r = 0;

  p1 = str[0];
  p2 = str[1];

  while (1) {
    if (*p1 == '¥0' && *p2 == '¥0') {
      r = 0;
      break;
    } else if (*p1 == '¥0') {
      r = -1;
      break;
    } else if (*p2 == '¥0') {
      r = 1;
      break;
    }
    p1++;
    p2++;
  }

  printf("RESULT=%d¥n", r);

  return 0;
}
```

RESULTに表示される数字の意味は?

関数呼び出しとポインタ

関数宣言，定義，呼び出し（復習）

関数宣言

```
#include <stdio.h>

int add_i(int a, int b);

int main(void) {
    int x = 10, y = 20, z;

    z = add_i(x, y);

    printf("%d + %d = %d¥n", x, y, z);

    return 0;
}

int add_i(int a, int b) {
    int c;

    c = a + b;

    return c;
}
```

①**関数名**は add_i

②**引数**は int型のaと，int型のb

③**返り値**はint型

関数呼び出し

関数定義

（実際の処理の内容）

スコープ（scope）（復習）

- スコープ（scope）: 変数の有効範囲
- 変数は，宣言されたブロック内とそのサブブロック内でのみ有効
  - サブブロック … ブロックの中のブロック

```
int g;
int main(void) {
    int x;
    for(...) {
    ..xのスコープ
    }
}
int func(int y) {
    int z;
    ...
}
```

```
int g;
int main(void) {
    int x;
    for(...) {
    ...
    }
}
int func(int y) {
    int z;
    y, zのスコープ
}
```

```
int g;
int main(void) {
    int x;
    for(...) {
    ...
    } gのスコープ
} （グローバル変数）
int func(int y) {
    int z;
    ...
}
```


関数実行のメモリ変化のイメージ

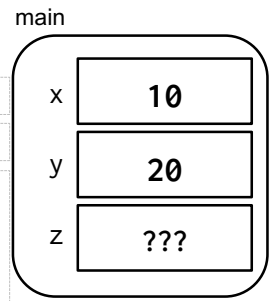
```
#include <stdio.h>
```

```
int add_i(int a, int b);
```

```
int main(void) {
    int x = 10, y = 20, z;

    z = add_i(x, y);

    printf("%d + %d = %d¥n", x, y, z);

    return 0;
}
```

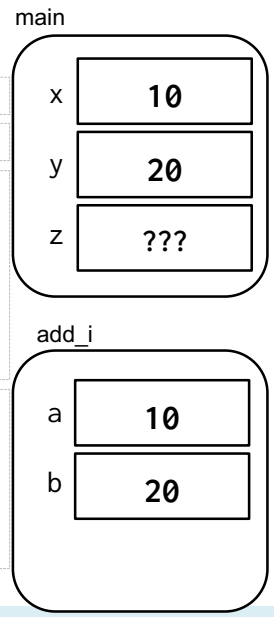
```
int add_i(int a, int b) {
    int c;

    c = a + b;

    return c;
}
```

```c
#include <stdio.h>

int add_i(int a, int b);

int main(void) {
    int x = 10, y = 20, z;

    z = add_i(x, y);

    printf("%d + %d = %d¥n", x, y, z);

    return 0;
}

int add_i(int a, int b) {
    int c;

    c = a + b;

    return c;
}
```



変数 x, y, z が作られる（スコープはmain関数内）

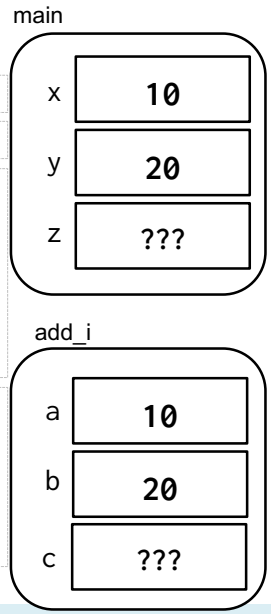

```c
#include <stdio.h>

int add_i(int a, int b);

int main(void) {
    int x = 10, y = 20, z;

    z = add_i(x, y);

    printf("%d + %d = %d¥n", x, y, z);

    return 0;
}

int add_i(int a, int b) {
    int c;

    c = a + b;

    return c;
}
```



変数xとyの値が引数として渡される

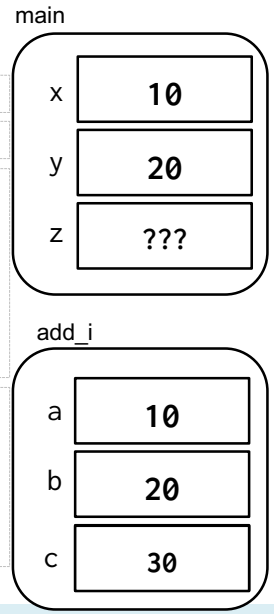
変数 a, b が作られる（スコープはadd_i関数内）変数は呼び出し時の引数で初期化される


```c
#include <stdio.h>

int add_i(int a, int b);

int main(void) {
    int x = 10, y = 20, z;

    z = add_i(x, y);

    printf("%d + %d = %d¥n", x, y, z);

    return 0;
}

int add_i(int a, int b) {
    int c;

    c = a + b;

    return c;
}
```



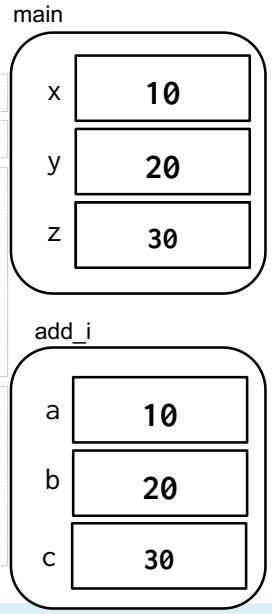
変数 c も作られる（スコープはadd_i関数内）引数とは無関係


```c
#include <stdio.h>

int add_i(int a, int b);

int main(void) {
    int x = 10, y = 20, z;

    z = add_i(x, y);

    printf("%d + %d = %d¥n", x, y, z);

    return 0;
}

int add_i(int a, int b) {
    int c;

    c = a + b;

    return c;
}
```

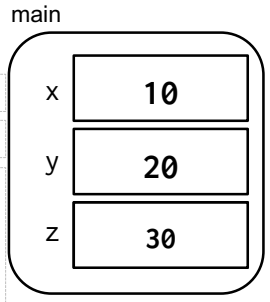


```c
#include <stdio.h>

int add_i(int a, int b);

int main(void) {
    int x = 10, y = 20, z;

    z = add_i(x, y);

    printf("%d + %d = %d¥n", x, y, z);

    return 0;
}

int add_i(int a, int b) {
    int c;

    c = a + b;

    return c;
}
```

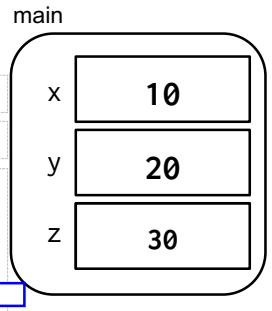
変数cの値が戻される




```c
#include <stdio.h>

int add_i(int a, int b);

int main(void) {
    int x = 10, y = 20, z;

    z = add_i(x, y);

    printf("%d + %d = %d¥n", x, y, z);

    return 0;
}

int add_i(int a, int b) {
    int c;

    c = a + b;

    return c;
}
```



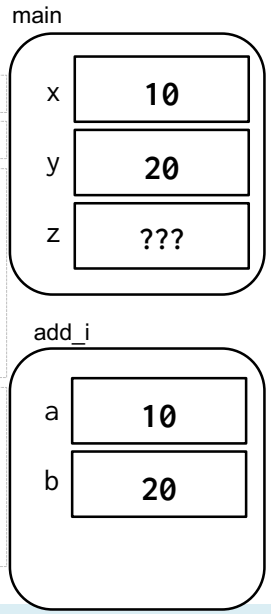

```c
#include <stdio.h>

int add_i(int a, int b);

int main(void) {
    int x = 10, y = 20, z;

    z = add_i(x, y);

    printf("%d + %d = %d¥n", x, y, z);

    return 0;
}

int add_i(int a, int b) {
    int c;

    c = a + b;

    return c;
}
```



注目ポイント

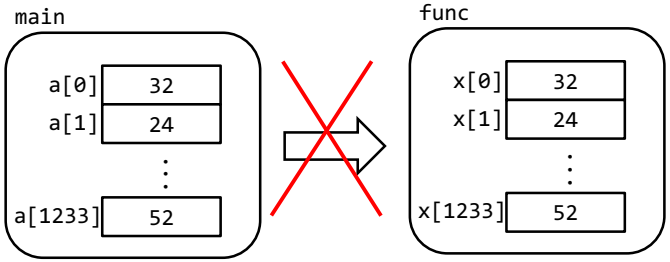
```c
#include <stdio.h>

int add_i(int a, int b);

int main(void) {
    int x = 10, y = 20, z;

    z = add_i(x, y);

    printf("%d + %d = %d¥n", x, y, z);

    return 0;
}

int add_i(int a, int b) {
    int c;

    c = a + b;

    return c;
}
```

引数で渡された値



配列データの受け渡し

- 配列はたくさんの要素を持つこともできる
  - 時には数千～　　int a[1234];
- たくさんの値を関数の引数として渡すとき，スコープを気にして全てをコピーすると非効率



配列データの受け渡し

- おさらい
  - **配列名**は配列の**先頭要素のアドレス**
  - アドレスなので**ポインタに代入**できる
  - **ポインタを介して配列にアクセス**できる

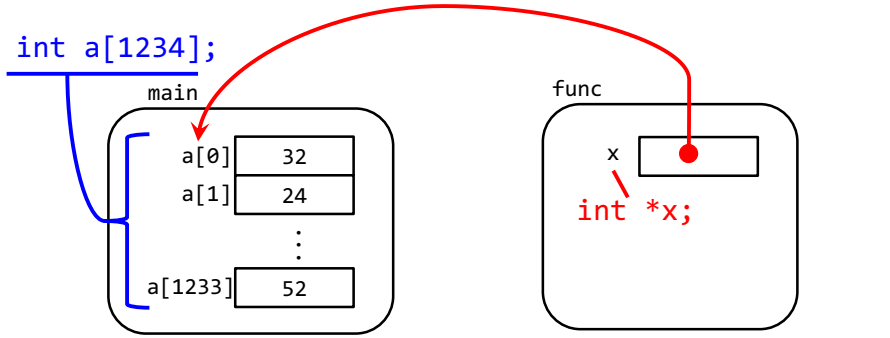
```
int a[1234];
int *ptr = a;
```

a[i]　*(ptr + i)　ptr[i]

全て同じ意味を持つ

配列データの受け渡し

- つまり… 配列の**先頭アドレス**だけを渡せば **自由に配列を参照**することができる



x[i] や *(x + i) で，aの要素にアクセスできる

例4

```
#include <stdio.h>

void bubble_sort(int *, int);

int main(void) {
  int array[10] = {7, 10, 1, 9,
                   3, 5, 6, 8, 4, 2};
  int i, n = 10;

  printf("BEFORE: ");
  for(i = 0; i < n; i++) {
    printf("%d ", array[i]);
  }
  printf("¥n");

  bubble_sort(array, n);

  printf("AFTER: ");
  for(i = 0; i < n; i++) {
    printf("%d ", array[i]);
  }
  printf("¥n");

  return 0;
}
```

sort.c

```
void bubble_sort(int *x, int n) {
  int i, j, s;

  for(i = 0; i < n - 1; i++) {
    for(j = 0; j < n - i - 1; j++){
      if (x[j] > x[j + 1]) {
        s = x[j];
        x[j] = x[j + 1];
        x[j + 1] = s;
      }
    }
  }

  return;
}
```

例4

sort.c

```
#include <stdio.h>

void bubble_sort(int *, int);

int main(void) {
  int array[10] = {7, 10, 1, 9,
                   3, 5, 6, 8, 4, 2};
  int i, n = 10;

  printf("BEFORE: ");
  for(i = 0; i < n; i++) {
    printf("%d ", array[i]);
  }
  printf("¥n");

  bubble_sort(array, n);

  printf("¥n");

  return 0;
}
```

配列 array の，先頭アドレス（array）と要素数 n を引数にした，関数 bubble_sort の呼び出し

```
void bubble_sort(int *x, int n) {
  int i, j, s;

  for(i = 1; i < n; i++){
    for(j = 0; j < n - 1; j++){
      if(x[j] > x[j + 1]){
        s = x[j];
        x[j] = x[j + 1];
        x[j + 1] = s;
      }
    }
  }

  return;
}
```

関数 bubble_sort 内では，受け取った配列の先頭アドレスがポインタ x に代入されていることを利用して，x[i]，*(x + i) などとして，配列の要素にアクセスすることができる

文字列の受け渡し

- おさらい:
  - 文字列はchar型の配列
  - 最後はヌル文字（'¥0'）で終わる

int型の配列と同様にポインタで受け渡しできる

あらかじめ配列の要素数（文字数）が分からなくても，最後がヌル文字で終わることを利用することに適切に処理できる

例5

count.c

```
#include <stdio.h>

int strcount(char *);

int main (void) {
  char str[] = "Hello, World!";
  int c;

  c = strcount(str);
  printf("count of '%s' = %d¥n", str, c);

  return 0;
}

int strcount(char *x) {
  int y = 0;
  char *ptr;

  for(ptr = x; *ptr != '¥0'; ptr++)
    y++;

  return y;
}
```

ptrの指す要素を文字列xの先頭からヌル文字の手前まで1文字ずつ変化させていくfor文

値渡し／参照渡し

- 値渡し（call by value）
  - 値をコピーすることで渡す方法
    例）void func(int a, int b);
  - 呼び出し元と関数内の変数には関係が無い
  - 呼び出し元の変数が変更されることは無い
- 参照渡し（call by reference）
  - アドレスを渡す方法
    例）void func(int *a);
  - 関数内からポインタ変数を介して呼び出し元の変数にアクセスできる
  - 呼び出し元の変数の内容を関数内から変更できる

main関数

- **mainも関数**

```
int main(void) {
   int x, y;
   ...
   return 0;
}
```

- 本当は引数も戻り値も持つ
  - 引数 … **コマンドライン引数**
    → プログラム自体に与えるパラメータ
  - 戻り値（int型） … **プログラムの終了ステータス**
    → **0は正常終了**，それ以外は異常終了
    （値によって意味が異なる）

どんな関数があるか

- 標準Cライブラリ（Wikipedia）など
  を見るとたくさん掲載されている
  - 標準なので，準拠している環境であればどれでも使える
- ライブラリとは … 関数の集合体
- ライブラリを使うには … 対応したヘッダを読み込む
  必要がある

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

  により，関数 `printf` が使えるようになる
  - 通常ヘッダの読み込みは最初に行う
- JM Project（https://linuxjm.osdn.jp/index.html）で
  関数の説明などが読める