グラフとアルゴリズムとプログラムのやさしいおはなし

渡邉敏正

2021 年 11 月 27 日

第 8 回

1　C 言語によるプログラム記述の基本

第 7 回は、第 6 回の ”グラフにおいて基点 s から連結な点をすべて求めるアルゴリズム $\mathbf{compo}(G, s)$” のプログラム作成に向けた準備として、以下の 2 点に焦点を当てました。

- コンピュータの大まかな仕組み、特にメモリと演算装置の関係を簡潔に説明すること
- アルゴリズム、プログラムおよびコンピュータの関係をわかりやすく説明すること

今回は、プログラム作成に直接的に関係する以下の事柄を中心に、実例を使いながら説明します。

- ベクトルや行列をコンピュータのメモリに配列として格納すること
- これらに関連して繰り返し処理を C 言語の for 文や while 文で記述すること
- 条件によって操作を選択して実行する処理を C 言語の条件判断文 (if〜else 文) で記述すること

例題として、$\mathbf{compo}(G, s)$ のプログラム作成時に利用できるいくつかの部分問題のプログラムを作成します。プログラミングに慣れると同時に、本題のプログラムを考えることにつながると思います。

1.1　C 言語プログラムの概形

C 言語プログラムの基本形式 (標準形式とよんでおきましょう) を示します。

```
1  /* コメント文：プログラムタイトルやファイル名など */
2  #include<stdio.h>
3  int main(void)
4  {
5      プログラムの記述;   // 種々の処理（複数可）を記述
6
7      return 0;
8  }
9  //左端の行番号は説明用に書いています。実際のプログラムに書くとエラーになります。
```

この通りに書かなくてもコンパイルを通る場合もありますが、この形式で書いておけば大丈夫ということです。

- 左端の行番号は説明用に書いています。実際のプログラムに書くとエラーになります。
- プログラムはいくつかの文で構成されます。各文は;(セミコロン) で終わります。文については宣言、代入、繰り返し、判定などいろいろと機能に応じて記述の仕方があります。
- 3 行目 `main(void)` のすぐ後の`{`から最後の 8 行目の`}`で囲んだ部分がプログラムの本体で、ここには一つ以上の文によって様々な処理を書きます。本体の最後には `return 0;` を書きます。
- 2 行目の `#include<stdio.h>` は今のところ、常にこのように書いておく、と覚えておいてください。みなさんがもう少しプログラミングに慣れてから説明します。
- 1 行目の `/* ... */` と 5 行目の `// ...` はコメント文とよばれます。 `...` の部分に様々なコメントを書きます。コンピュータには無視され、プログラムの実行には影響を与えません。`// ...` はその行だけ有効です。 `...` 部分が複数行にまたがる場合は `/*` と `*/` で囲む必要があります。

以下では、プログラム本体のみ (`main` の中のみ) を記述します。

1.2 宣言と代入

プログラムに出てくる宣言や変数に着目して、これらとコンピュータの仕組みとの関係を説明します。プログラムで何か処理をする場合、必要な値を格納する**変数** (variable: データを入れる容器のイメージ) を使います。また、処理の方法や内容を具体的に記述しなければなりませんが、その代表的な書き方に**代入文** (assignment statement) があります。以下ではこれらとコンピュータとの関係を例を使いながら説明します。

1.2.1 変数の宣言

プログラムで使用する変数については、変数名とともに各変数にどのような型のデータを格納するかをプログラムの最初に明示しなければなりません。

- **データ型** (data type) としては、整数、小数、文字、などをはじめとして種々のデータを扱うことができます。
- データ型は呼び方が「何々型」と決まっており、その型をプログラムの最初に記述します[*1]。これを (変数の) **宣言** (declaration) といいます。例えば、整数型変数 `a` を使うのであればプログラムの最初の部分に

  ```
  int a;
  ```

  と書きます。「整数を扱う変数 `a` を使いますよ」ということをコンピュータに知らせて準備をさせる、という意味を持っています。
- プログラム中で使う変数はこのようにすべて宣言をしなくてはなりません。宣言しない変数を使うとコンパイルがエラーメッセージを出してきます。なぜ宣言が必要なのかについてはもう少し先の回で説明します。

1.2.2 代入

次に代入を説明します。例えば、

[*1] 自分で新しくデータ型を作って〜型と名前を付けて使うこともできます。

```
int a,b,c;
a=45;
b=27;
c=a+b;
```

と記述したとします。

- ここでは 1 行目が宣言文で、下側の 3 行の各々が代入文です。なお、宣言は同じ型の変数であれば 1 行目の記述のようにカンマで続けて同じ行に書くことができます。
- すでに何回か注意をしていますが、C 言語プログラム中の = についてです。上記の = は等号ではありません。= の右側の式などの値を左側の変数に代入することを意味します。代入の意味を明示するため、しばしば a ← 45; や c ← a+b; と矢印で書くことも言いました。
- 上記の記述の意味は、整数型変数 a,b,c を宣言した後で、a に 45 を、b に 27 をそれぞれ代入し、さらに a と b の和を計算して、結果を c に代入する、という操作です。(ここでは結果を表示する操作は入っていません。)

もう一つの例として、プログラムの最初に整数型変数 k を

```
int k;
```

と宣言し、プログラムの中で代入文を

```
k=k+1;
```

と記述したとしましょう。

- この代入文の意味は

  現在の変数 k に 1 を加えて、これを新しく k の値とする

  ということです。数学の等号と全く異なる意味です。数学では有限値でこのような式は成立しません。
- 対応するコンピュータ内部での動作を図 60 に示します[*2]。同図は k の値が 3 の場合を示しています。
- k+1; の部分で変数 k の値 3 がレジスタ EAX に読み出され、それに 1 が加えられます。EAX の値が 4 となり、k= の部分でその値が変数 k に書き込まれます。これが k=k+1; という代入文の実際の姿です。プログラム中の =という記号と代入という動作を結びつけてください。

注意 8.1 整数型変数 k に対して、整数値を加えたり引いたりする場合によく使われる記述をまとめておきます。現状では、左右の並びはほぼ同じ意味の異なる表現と考えておいてください[*3]。

```
k=k+1     k++
k=k+2     k+=2
  ⋮         ⋮
k=k-1     k--
k=k-2     k-=2
  ⋮         ⋮
```

[*2] 同様のコンピュータ内部の変化の様子は第 7 回でも示しています。

[*3] k++ は**後置増分演算子** (postfixed increment operator) ++ により k に 1 を加える式です。k=k+2 は k に 2 を加えた値を k に代入する代入文ですが、これを**複合代入演算子** (compound assignment operator) +=を使って書いたのが k+=2 です。

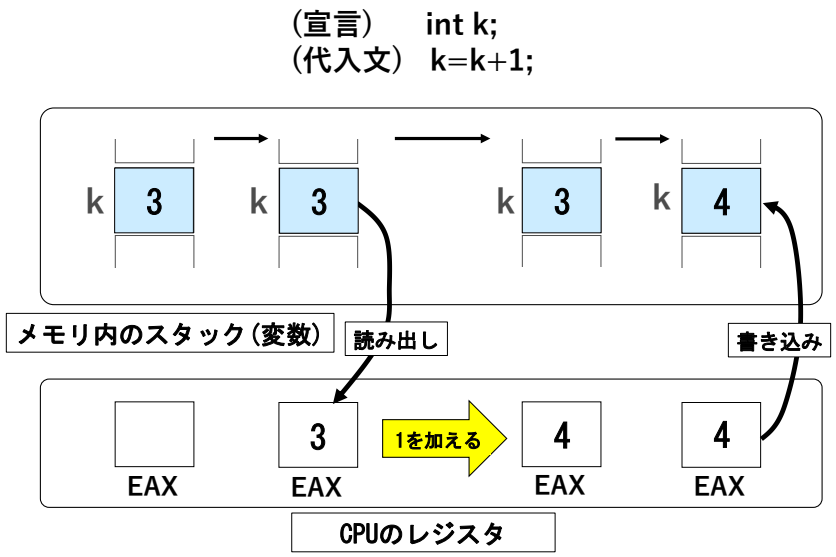

図 60　代入文 **k=k+1;** の実行状況 (変数 k の値が 3 の場合)

1.3　繰り返し処理

ある一定の処理を繰り返す場合の記述の仕方は種々ありますが、ここでは `for` 文と `while` 文を説明します。

1.3.1　for 文

まず、例えば

```
int i;
```

と整数型変数 `i` を宣言しておきます。そしてプログラム中で例えば以下のような形式の記述をします。

```
for (i=0; i<=6; i++) {
    操作内容の記述;   /* 複数の処理の記述が可能 */
}
```

- この記述は、i の値を 0 から 6 まで 1 ずつ増やしながら、各 `i` について操作内容の記述にある操作を繰り返すことの指示です。
- 例として 1 から 10 までの和を計算するプログラムを考えてみましょう。直接的に表現したプログラムとしては (和の計算部分だけ記述します)

  ```
  int s;
  s=1+2+3+4+5+6+7+8+9+10;
  ```

  が考えられます。`s` の値をディスプレイに表示させていませんので、データはメモリの中にあって外部には見えません。(ディスプレイへの表示方法については後述します。)
- 上記の書き方はわかりやすいですが、和が 100 まであるいは 1000 までの場合には同じ書き方では大変なことになることは容易に想像がつくと思います。もちろんこのような等差数列の和は数学的に求める

ことができます。では途中でいくつか抜けて規則的な数列でない場合にはどうしましょうか。このような場合には後述の配列が有用なのですが、それについては配列の項で説明します。

- 話を元に戻して、for 文の利用を考えます。例えば以下のような記述をしてみましょう。

```
int i, s;  // 次行の s=0; を合わせて、宣言で int s=0; と書くこともできます
s=0;
for (i=1; i<=10; i++) {
    s=s+i;
}
```

- プログラムはまず $s \leftarrow 0$ と設定して、次に $i=1$ から $i=10$ まで (i の値を 1 ずつ増やしながら) 代入文 `s=s+i;` を反復実行します。その計算の様子を次に示します。以下の記述での $=$ は等号です。$\leftarrow$ の右側で計算した値を $\leftarrow$ の左側の変数 `s` に代入します。i が 1 から 10 へと増えていくことに伴って
  - 変数 s の値が刻々と変化していくこと
  - 変化した s の値が次の右辺 $s+i$ の計算に使われること

  に注意してください。カッコ内は $i=k$ のときの $s+i$ の計算です。この値が左辺の s の値となって、次の $i=k+1$ のときの右辺 $s+i$ の計算で使われます。
  - (開始前) $s \leftarrow 0$;
  - $i=1:\ s \leftarrow s+1\,(=0+1=1)$;
  - $i=2:\ s \leftarrow s+2\,(=1+2=3)$;
  - $i=3:\ s \leftarrow s+3\,(=3+3=6)$;
  - $i=4:\ s \leftarrow s+4\,(=6+4=10)$;
  - $i=5:\ s \leftarrow s+5\,(=10+5=15)$;
  - $i=6:\ s \leftarrow s+6\,(=15+6=21)$;
  - $i=7:\ s \leftarrow s+7\,(=21+7=28)$;
  - $i=8:\ s \leftarrow s+8\,(=28+8=36)$;
  - $i=9:\ s \leftarrow s+9\,(=36+9=45)$;
  - $i=10:\ s \leftarrow s+10\,(=45+10=55)$;

  となって、確かに最終的に $s=55$ と正しく計算されています。この書き方であれば 100 や 1000 までも同じ書き方です。上記の書き方は加算の場合によく使われますので、慣れておくと便利です。
- 上記では「i の値を 1 から最大 10 まで 1 ずつ増やしながら」としていますが、i の値を 1 から最大 10 まで 2 ずつ増やしながら (実質は 9 まで)、i の値を 10 から最小 1 まで 1 ずつ減少させながら、i の値を 10 から最小 1 まで 2 ずつ減少しながら (実質は 2 まで)、など様々な反復形式が記述可能です。
  - /* i の値を 1 から最大 10 まで 2 ずつ増やしながら (実質は 9 まで) */

```
for (i=1; i<=10; i+=2) {
    操作内容の記述;  /* 複数の処理の記述が可能 */
}
```

  - /* i の値を 10 から最小 1 まで 1 ずつ減少させながら */

```
for (i=10; i>=1; i--) {
    操作内容の記述;  /* 複数の処理の記述が可能 */
}
```

  - /* i の値を 10 から最小 1 まで 2 ずつ減少しながら (実質は 2 まで) */

```
for (i=10; i>=1; i-=2) {
```

```
        操作内容の記述;  /* 複数の処理の記述が可能 */
    }
```

- なお、このような反復部分をループとよび、特にこの場合は**一重ループ** (single loop) と言います。i の変化分 (繰り返しの間隔) を**刻み幅** (step size) とよぶことがあります。刻み幅が 1 のときには特に指定する必要はありません。

1.3.2 while 文

プログラム中で例えば以下のような形式の記述をします。

```
while (実行条件) {
    操作内容の記述;  /* 複数の処理の記述が可能 */
    実行条件の変更;
}
```

- この記述では、まず実行条件が成り立つか否かをチェックします。成り立てば操作内容の記述にある操作を実行し、操作終了時に実行条件を修正して最初の実行条件の判定に戻ります。このように実行条件が成り立つ限り操作を繰り返すことの指示です。実行条件が成り立たない場合にはこの while 文を終了して次の操作 (`}` の次にある記述) に進みます。
- 先程 for 文の例題で取り上げた 1 から 10 までの和を計算するプログラムを while 文で書いてみると、例えば以下のようになります。

```
int i=1, s=0;
while (i<=10) {
    s=s+i;
    i++;
}
```

- この記述ではまず、$i \leftarrow 1$, $s \leftarrow 0$ と設定しておき、while 文で i の値が 10 以下である限り `s=s+i;` と `i++;` をペアで繰り返します。計算は以下のように進行します。以下の記述での $=$ はやはり等号です。
  - (開始前) $i \leftarrow 1;\ s \leftarrow 0;$
  - $i = 1 \leq 10:\ s \leftarrow s + 1\,(= 0 + 1 = 1);\ i \leftarrow 2;$
  - $i = 2 \leq 10:\ s \leftarrow s + 2\,(= 1 + 2 = 3);\ i \leftarrow 3;$
  - $i = 3 \leq 10:\ s \leftarrow s + 3\,(= 3 + 3 = 6);\ i \leftarrow 4;$
  - $i = 4 \leq 10:\ s \leftarrow s + 4\,(= 6 + 4 = 10);\ i \leftarrow 5;$
  - $i = 5 \leq 10:\ s \leftarrow s + 5\,(= 10 + 5 = 15);\ i \leftarrow 6;$
  - $i = 6 \leq 10:\ s \leftarrow s + 6\,(= 15 + 6 = 21);\ i \leftarrow 7;$
  - $i = 7 \leq 10:\ s \leftarrow s + 7\,(= 21 + 7 = 28);\ i \leftarrow 8;$
  - $i = 8 \leq 10:\ s \leftarrow s + 8\,(= 28 + 8 = 36);\ i \leftarrow 9;$
  - $i = 9 \leq 10:\ s \leftarrow s + 9\,(= 36 + 9 = 45);\ i \leftarrow 10;$
  - $i = 10 \leq 10:\ s \leftarrow s + 10\,(= 45 + 10 = 55);\ i \leftarrow 11;$
  - $i = 11 > 10:$ while 文を終了
- 繰り返しの始めと終わり、および刻み幅が明確な場合には for 文が便利です。一方、これらの条件に当

てはまらない繰り返しの場合には while 文が便利です。それぞれに使い道があります。

1.4 配列の宣言と代入

繰り返し処理に便利なデータ格納の仕組みとして配列があります。1 つの配列に格納できるデータ型はどれか 1 種類で混在はできませんが繰り返し処理に適しています。ここでは **1 次元配列** (one-dimensional array) と **2 次元配列** (two-dimensional array) を説明します。

1.4.1 1 次元配列

宣言と代入文は例えば以下のように記述します。

```
int d[7];
d[4]=2;
```

- この宣言で、7 個の整数型変数 `d[0],d[1],d[2],d[3],d[4],d[5],d[6]` が主記憶に用意されます (図 61 参照)。
- この配列の**長さ** (length) は 7 です。
- 1 次元配列の要素は例えば `d[4]` というように

  配列名 `[i]`

  と記載します。`i` を**配列添字** (array index) といいます。
- `d[4]=2;` という代入文のコンピュータ内部での動作は変数の場合と同じです (図 62 参照)。

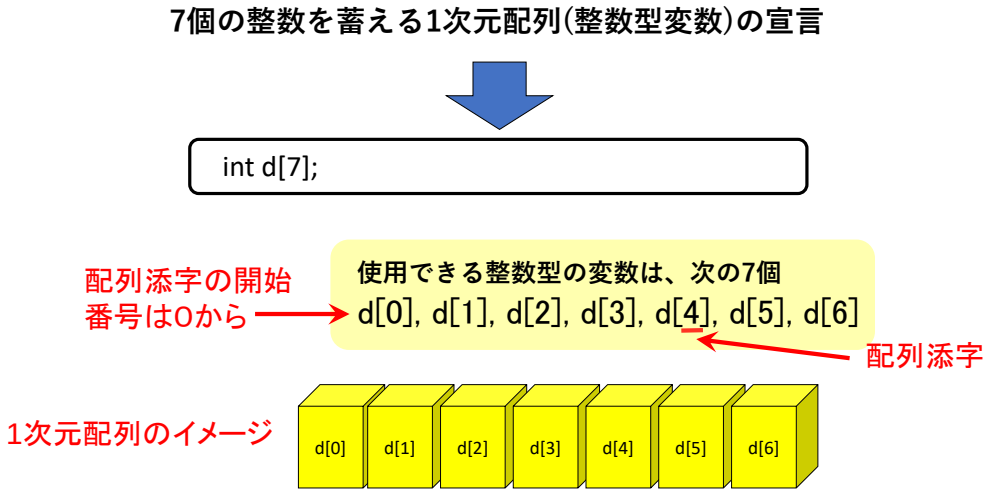

図 61　1 次元配列の宣言とメモリ内のイメージ

注意 8.2 ここで注意すべきことは、配列添字が 0 から始まるということで、この点はいろいろな処理で少し注意しておく必要があります。例えば図 63 では、`d[1]=3, d[4]=2` でそれぞれ 2 番目の要素、5 番目の要素です。配列添字が 0〜6 で要素が 1〜7 番目という通常の番号付けと 1 だけずれていることに注意してください。

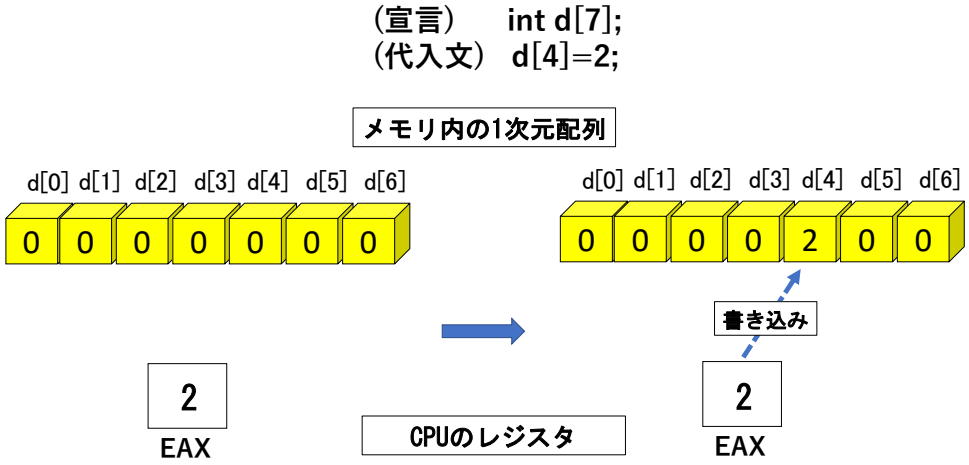

図 62　1 次元配列 **d** への代入操作

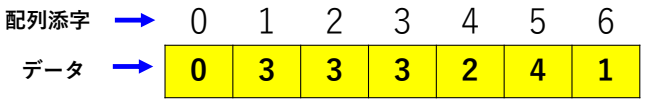

図 63　整数型 1 次元配列の例 (黄色の部分)

- 格納するデータの型は 1 種類ですが `for` 文との相性がよく、例えばこの配列のすべての要素に 5 を加える操作は `for` 文を使って (宣言文は省略します)

```
for (i=0;i<=6;i++){
    d[i]=d[i]+5;
}
```

  と記述できます。
- あるいは `d[0]` から `d[6]` までの総和を求めることはやはり for 文を使って

```
s=0;
for (i=0;i<=6;i++){
    s=s+d[i];
}
```

  と記述できます。図 63 について和計算の進行を見てみると
  - (開始) $s \leftarrow 0$;
  - $i = 0$: $s \leftarrow s + d[0]\ (= 0 + 0 = 0)$;
  - $i = 1$: $s \leftarrow s + d[1]\ (= 0 + 3 = 3)$;
  - $i = 2$: $s \leftarrow s + d[2]\ (= 3 + 3 = 6)$;
  - $i = 3$: $s \leftarrow s + d[3]\ (= 6 + 3 = 9)$;
  - $i = 4$: $s \leftarrow s + d[4]\ (= 9 + 2 = 11)$;
  - $i = 5$: $s \leftarrow s + d[5]\ (= 11 + 4 = 15)$;
  - $i = 6$: $s \leftarrow s + d[6]\ (= 15 + 1 = 16)$;

となって、確かに $s = 16$ と正しく計算されています。

1.4.2　2 次元配列

1 次元配列を何本か束ねたイメージです。宣言は例えば以下のように記述します。代入の説明は省略します。

```
int a[5][6];
```

- この宣言で、5 ×6 (= 30) 個の整数型変数 a[0][0], ··· , a[4][5] が用意されます。別に言い方をすると、長さ 6 の 1 次元配列

  a[i][0],a[i][1],a[i][2],a[i][3],a[i][4],a[i][5]

  が $i = 0, \cdots, 4$ について 1 本ずつ計 5 本用意されるということです (図 64 参照)。
- このように 2 次元配列の要素は例えば a[2][3] というように

  配列名 [i][j]

  の形で記載します。i, j が配列添字ですが、i が行の配列添字、j が列の配列添字です[*4]。

注意 8.3 再び添字に関する注意です。例えば図 65 の整数型 2 次元配列 a[3][4] では、

a[0][1]=3, a[1][2]=11, a[2][3]=9

ですが、それぞれ 1 行 2 列の要素、2 行 3 列の要素、3 行 4 列の要素ということになります。行添字が 0〜2、列添字が 0〜3 ですが、行として 1〜3 行目、列として 1〜4 列目で、番号付けが 1 だけずれるということです。配列添字がいずれも 0 から始まるということはいろいろな処理で少し注意しておくべきことです。

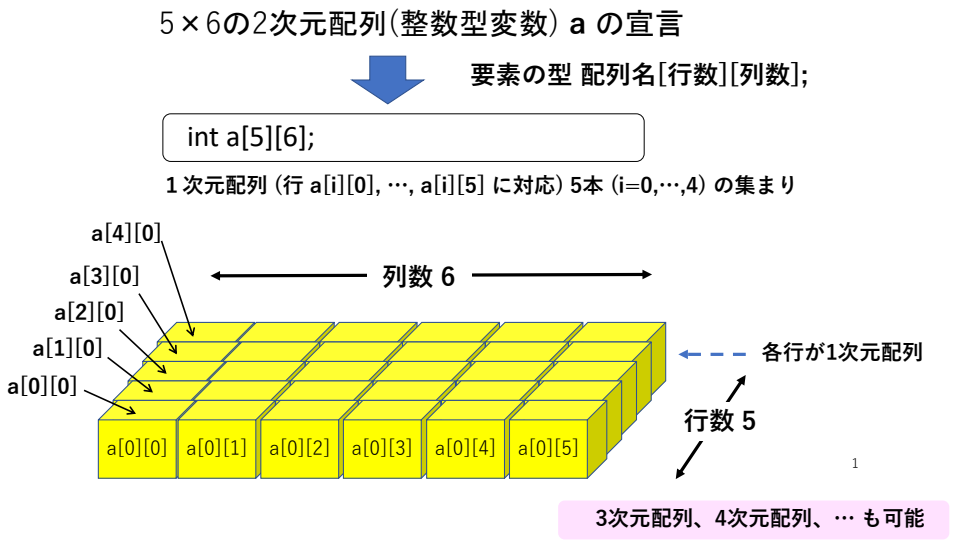

図 64　2 次元配列の宣言とメモリのイメージ (行数=1 次元配列の本数; 列数=1 次元配列の長さ)

[*4] 行は横方向の並び 1 本ずつ、列は縦方向の並び 1 本ずつのことです。横方向の並びが 5 本、縦方向の並びが 6 本という碁盤の目の構造というイメージです。

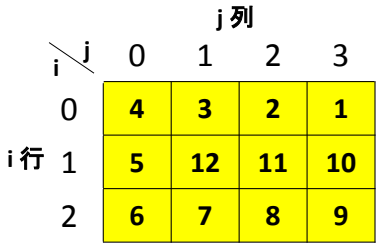

図 65　整数型 2 次元配列の例 (黄色の部分)

- 2 次元配列と for 文の関係を例題でみておきましょう。for 文を使って図 65 の要素の総和を求めてみましょう。例えば以下のような記述になります。

```
s=0;
for (i=0;i<=2;i++){
    for (j=0; j<=3;j++){
        s=s+a[i][j];
    }
}
```

和計算の進行を見てみると

(開始) $s \leftarrow 0$;

$i = 0$ のとき

- $j = 0$: $s \leftarrow s + a[0][0]$ $(= 0 + 4 = 4)$;
- $j = 1$: $s \leftarrow s + a[0][1]$ $(= 4 + 3 = 7)$;
- $j = 2$: $s \leftarrow s + a[0][2]$ $(= 7 + 2 = 9)$;
- $j = 3$: $s \leftarrow s + a[0][3]$ $(= 9 + 1 = 10)$;

$i = 1$ のとき

- $j = 0$: $s \leftarrow s + a[1][0]$ $(= 10 + 5 = 15)$;
- $j = 1$: $s \leftarrow s + a[1][1]$ $(= 15 + 12 = 27)$;
- $j = 2$: $s \leftarrow s + a[1][2]$ $(= 27 + 11 = 38)$;
- $j = 3$: $s \leftarrow s + a[1][3]$ $(= 38 + 10 = 48)$;

$i = 2$ のとき

- $j = 0$: $s \leftarrow s + a[2][0]$ $(= 48 + 6 = 54)$;
- $j = 1$: $s \leftarrow s + a[2][1]$ $(= 54 + 7 = 61)$;
- $j = 2$: $s \leftarrow s + a[2][2]$ $(= 61 + 8 = 69)$;
- $j = 3$: $s \leftarrow s + a[2][3]$ $(= 69 + 9 = 78)$;

となって、計算結果は確かに出現している数値 1〜12 の総和 78 に等しくなっています。

- i の 1 重ループの中に j の 1 重ループが含まれる形になっています。このような構造を **2 重ループ** (double loop) といいます。2 重ループは 2 次元配列の処理をはじめとして様々な操作に現れますので、慣れておくと役に立つ記述の形です。

1.5 情報の画面表示

プログラム実行中に何らかの情報をディスプレイに出力する (表示する) printf 文を説明します。

- 整数型変数 k の値をディスプレイに表示するには、printf 文を使って例えば次のように記述します。

```
printf("k=%d\n",k);
```

ここで、printf が表示命令でカッコ内が表示する文字列やデータおよびそれらの表示形式が合体した記号列です。

- 具体的に言いますと、"..."の部分が表示する文字やデータおよびそれらの表示形式が合体した記号列です。カンマの右側の k がこの記号列の中の %d の位置に表示する整数データ (変数 k が格納しているデータ) であることを示しています。k の値をこの記号列で指示した位置に置いた形式でディスプレイに表示しなさい、という命令文です。

注意 8.4 上記の説明で出てきた %d ですが、これは C 言語では整数値を意味する記号で、変数名や配列名とは無関係です。変数が a でも配列名が b でもプログラム中で整数値は常に %d で表示位置を指定します。

- ここで

```
k=%d\n
```

の意味は、まず画面に k= と表示し、その次に続けて整数を表示して改行せよ、ということです。\n は改行を表す記号で、これを記号列の中に含ませると表示がその位置まできたところで改行の動作をします (ディスプレイ上で次のデータの表示位置が次行の左端に移動します)。

- カッコ内には、%d の位置に表示する整数の値はカンマの右側 (この場合はカッコ内の最後部分) にある変数 k の値 (つまり 3) とする、ということを指示しています。したがって、例えば k の値が 3 であれば

```
k=3
```

と表示し、表示後に改行します。

- 別の例を示しましょう。この例は、**1.2.1 変数の宣言**で取り上げた、2 つの変数 a と b の値の加算を実行して結果を変数 c に代入するプログラムの部分です。この最後部分に a,b,c の値を表示する操作を追加してみましょう。例えば以下のような記述になります (プログラムの中の関係する部分のみを記載しています)。

```
int a=45;
int b=27;
int c;
c=a+b;
printf("a=%d, b=%d, c=%d\n",a,b,c);
```

- この結果、ディスプレイの表示は以下のようになり、表示後に改行があります。

```
a=45, b=27, c=72
```

前の printf 文の例から類推できると思いますが、この printf 文の意味を簡単に説明しておきます。

- ディスプレイ上で a= に続けて整数値を表示して半角スペースを入れ、次に b= に続けて整数値を表示して半角スペースを入れ、さらに c= に続けて整数値を表示して最後に改行します。(なお、半角スペー

スを増減すれば間隔を変えることができます。半角を全角 (半角 2 個分) にすることも可能です。) %d の位置に表示する整数値は、"...", の右側にある変数 a,b,c の値を**この順に**使う、という指示です。「この順に」という変数の順序は重要です。

1.6 コンピュータへのデータ入力

プログラム実行中にデータをコンピュータに入力する (読み込ませる) 方法はいくつかありますが、ここではキーボードから値を読み込ませる方法とプログラム内に値を記述する方法を説明します。

1.6.1 キーボードから値を読み込ませる方法

- 整数型の宣言をした変数 k にキーボードから整数値を入力する (格納する) には、例えば以下のように記述します。

```
printf("\n キーボードから数値、enter の順に打ち込んでください。\n");
scanf("k=%d",&k);
```

- 上側の行で、人間に入力を促す案内メッセージをディスプレイに表示させ、その後で改行させています。下側の行が実際にデータを入力する操作で、キーボードから入力された整数を変数 k に代入します。
- データを格納する変数については、scanf の場合には

  &k と変数の前に&を付けること

  が注意点ですが、ここではその理由は省略します。
- scanf を使って図 63 のデータを設定してみましょう。プログラム例としては図 66 のようになり、その実行結果は図 67 になります。
- 図 67 の 1〜2 行目

```
gcc -o data-input data-input.c
```

  はファイル data-input.c 内のソースプログラムを実行可能プログラムに変換して、それをファイル data-input に格納するという、コンパイル操作です。続いて 3 行目

```
./data-input
```

  で data-input にある実行可能プログラムを実行させています。このように、ソースプログラム、実行可能プログラムいずれもファイル名で扱われます。
- プログラムを実行すると、10 行目〜14 行目でディスプレイに d[0]= と表示させ = のあとに促進マーク (次のデータを表示する位置を示すマーク) が点滅します。
- キーボードで 0 と打って enter キーを押すと (コンピュータは d[0] に値 0 を格納し)、改行して d[1]= と表示が変わり、= のあとに促進マークが点滅します。ここで今度は 3 をキーボードから打って、$\cdots$ と同様の動作が $i = 1, \cdots, n\,(= 6)$ と続きます。
- $i = n\,(n = 6)$ まで入力が終了すると、入力した全データを表示して終了します。図 67 は終了時のディスプレイの様子 (入力時のデータと全データの表示) を示しています。

注意 8.5 図 66 の 4 行目にある

```
# define n 6
```

という記述は**オブジェクト形式マクロ** (object-like macro) とよばれます。単に**マクロ**という場合もあります。「これ以降に出現する n を 6 に置換せよ」という指令です。これにより、このプログラム中に

```
1  //data input from keyboard
2
3  #include <stdio.h>
4  #define n 6
5
6  int main(void)
7  {
8      int d[n+1],i;
9
10     printf("\n");
11     for (i=0; i<=n; i++){
12         printf("d[%d]=",i);
13         scanf("%d",&d[i]);
14     }
15     printf("\ndata in d[0]...d[%d]:\n",n);
16     for (i=0; i<=n; i++){
17         printf("d[%d]=%d ",i,d[i]);
18     }
19     printf("\n\n");
20
21     return 0;
22 }
```

図 66　scanf を使った 1 次元配列へのデータの入力プログラム例 (プログラム名: data-input.c)。左端の行番号は説明用に書いています。実際のプログラムに書くとエラーになります。

```
Toshimasa-no-MacBook-Pro:repos watanabe$ gcc -o data-input data-input.c
Toshimasa-no-MacBook-Pro:repos watanabe$ ./data-input

d[0]=0
d[1]=3
d[2]=3
d[3]=3
d[4]=2
d[5]=4
d[6]=1

data in d[0]...d[6]:
d[0]=0 d[1]=3 d[2]=3 d[3]=3 d[4]=2 d[5]=4 d[6]=1
```

図 67　図 66 のプログラムの実行結果

出てくる n はすべて整数 6 に置き換えてからコンパイルされます。

プログラム中でよく使う数値などはマクロを利用して文字などで記述しておくこと

が賢明です。その理由はすぐにわかると思います。もし 6 を使ってプログラムを記述していたとしましょう。仮に 6 を 10 に変更することになったとするとプログラム中のすべての 6 を 10 に書き換えなければなりませんが、`# define n 6` なるマクロを使って n で記述していれば `# define n 10` と変更するだけで変更完了です。

- `scanf` はデータ入力が必要になった場合にはよく使われます。この方法の良いところはいつでも自由にデータ入力ができるところです。一方で少し困るのはプログラムを実行するたびにデータの入力操作

をしなければならないことです。データの個数が多いときには大変です。通常は一度入力したデータはファイルに保存して再利用しますが、この方法については少し準備が必要ですので別の機会に説明します。

1.6.2 プログラム内に値を記述する方法

- プログラムの宣言で初期値の設定をすることができます。[*5]これを利用してデータの入力をすることができます。これに関してはすでに和の計算 a+b の動作説明のときに以下のような記述をしています。

```
int a=45;
int b=27;
int c;
```

上記の 2 行目と 3 行目で変数の宣言と同時に初期値設定をしています。

- 配列でも同様な初期値設定ができます。例えば 1 次元ベクトル[*6] $\boldsymbol{d} = (3,3,3,2,4,1)$ について、1 次元配列を

```
int d[7]={0,3,3,3,2,4,1};
```

と宣言で記述すると図 68 のように初期値設定 (データの格納) ができます。これは図 63 と同じものですが、図 61 と同様のイメージで示したものです。(d[0] の値 0 についてはあとで説明します。)

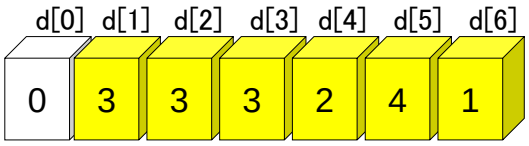

図 68　ベクトル $\boldsymbol{d} = (3,3,3,2,4,1)$ の 1 次元配列 $\mathbf{d}$ への格納状況 (黄色部分に格納)

- 一方、

```
int a[3][4]={ {4,3,2,1},{5,12,11,10},{6,7,8,9} };
```

と記述すると図 65 のようなデータ設定ができます。

- あるいは次の 6 行 6 列の行列

$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

に対して、2 次元配列を

```
int A[7][7]={ {0,0,0,0,0,0,0},{0,0,1,0,1,1,0},{0,1,0,1,0,1,0},
              {0,0,1,0,1,1,0},{0,1,0,1,0,0,0},{0,1,1,1,0,0,1},
              {0,0,0,0,0,1,0} };
```

と記述すると図 69 のように初期値設定ができます。(1 行 1 列の値 0 についてはあとで説明します。)

[*5] この記述方法は宣言の初期値設定のみ有効です。

[*6] スペースの関係でベクトルを行ベクトル (要素が横方向に並ぶ形) として記載します。

j 列

i 行 \ j	0	1	2	3	4	5	6
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	**0**	**0**	**1**	**0**	**1**	**1**	**0**
2	**0**	**1**	**0**	**1**	**0**	**1**	**0**
3	**0**	**0**	**1**	**0**	**1**	**1**	**0**
4	**0**	**1**	**0**	**1**	**0**	**0**	**0**
5	**0**	**1**	**1**	**1**	**0**	**0**	**1**
6	**0**	**0**	**0**	**0**	**0**	**1**	**0**

図 69　行列 A の 2 次元配列 **A**[7][7] への格納 (黄色部分に格納)

- 行のデータを列数だけ並べた書き方ですが、記述の仕方は他にもあります。一度記述しておけばプログラム実行時に自動的に値の設定ができるのでプログラム開発の途中ではデータ入力が省略できて少し楽になります。一方で、別のデータに対して実行する場合にはこの部分を書き換える必要があります。キーボード入力と比較してどちらを採用するかは状況によって判断することになります。

注 8.6 図 63 (あるいは図 68) および図 69 に関して補足説明をしておきます。

- 例えば、整数値の 1 次元ベクトル $\boldsymbol{d} = (3, 3, 3, 2, 4, 1)$ を 1 次元配列に格納する場合に、1 次元配列として通常 `int d[7];` と宣言します。7 個の要素用に `d[0]`, $\cdots$, `d[6]` が用意されます。このとき `d[0]`, $\cdots$, `d[5]` にベクトルの 1 番目, $\cdots$, 6 番目の要素を格納すると、配列添字 0〜5 とベクトル要素の 1 番目〜6 番目という番号が 1 だけずれてわかりにくくなります。それで `d[0]` は使わないこととし (ここでは `d[0]` の値を 0 と設定)、`d[1]`, $\cdots$, `d[6]` にベクトルの 1 番目の要素、$\cdots$、6 番目の要素を格納することにします。
- 同様に、例えば上記の 6 行 6 列の行列 A を格納するため `int A[7][7];` と宣言した 2 次元配列の場合も 1 行、1 列は使わないこととし、2 行〜7 行、2 列〜7 列を使用します。2 次元配列の 1 行、1 列の配列添字は共に 0 ですので、要素は `A[1][1]`〜`A[6][6]` に格納します。これで、$i, j = 1, \cdots, 6$ について、A の i 行 j 列の要素 $A(i, j)$ と配列の要素 `A[i][j]` が対応します。
- このような格納の仕方をするとベクトルや行列の要素と配列の要素の対応がスッキリしてわかりやすくなります。上記の初期値設定の例で、図 68 の 1 次元配列では最初に 0 があり、図 69 の 2 次元配列では 1 行と 1 列に 0 があるのはこのためです。このようなデータの格納形式は C 言語の配列を扱う場合にしばしば採用されます。
- 繰り返しになりますが、皆さんの中には「不便なのにどうして配列添字と要素番号を合わせないのだろう」と考える方もいると思います。実は別の観点からこの C 言語の配列添字が 0 から始まることには大きなメリットがあるのですが、その理解にはメモリ操作などコンピュータの仕組に関する知識が必要になりますので、残念ながらこれ以上の説明はここでは省略します。

1.6.3 コンピュータからのデータの出力方法

プログラム実行中に何らかの情報をコンピュータから出力する (書き出す、表示する) ことが必要になります。

- ここでは、1.5 **情報の画面表示**で説明しました `printf` 文を使ってディスプレイに文字列やデータを表示することを取り上げます。
- 図 65 の要素をディスプレイに表示させてみましょう。プログラム例は図 70 で、その実行結果は図 71 になります。
- `a[3][4]` のデータを、i, j の 2 重ループにより printf 文を使って `a[i][j]=要素`、という形でディスプレイに表示しています。
- 図 71 の 1〜2 行目はコンパイルと実行の指示です。1 行置いてデータを表示しています。
- 皆さんで 2 重ループによる表示動作を確認してみてください。

```
1  //data-output to display
2
3  #include <stdio.h>
4  #define n 2
5  #define m 3
6
7  int main(void)
8  {
9      int a[n+1][m+1]={ {4,3,2,1}, {5,12,11,10}, {6,7,8,9} };
10     int i, j;
11
12     printf("\n****** Data in 2d-array a[%d][%d] *****\n",n+1,m+1);
13     for (i=0; i<=n; i++){
14         for (j=0; j<=m; j++){
15             printf("a[%d][%d]=%d ", i, j, a[i][j]);
16         }
17         printf("\n");
18     }
19     printf("\n");
20
21     return 0;
22 }
```

図 70　2 次元配列のデータ表示例 (プログラム名: data-output.c)。左端の行番号は説明用に書いています。実際のプログラムに書くとエラーになります。

1.7 条件判断文

最後に、**条件判断文** (conditional statement) の説明をします。これはプログラム作成では頻繁に出現し、非常に重要です。

- 条件が成り立つか否かで次に実行する操作を選択、実行させるための書き方です。一般形は

```
Toshimasa-no-MacBook-Pro:repos watanabe$ gcc -o data-output data-output.c
Toshimasa-no-MacBook-Pro:repos watanabe$ ./data-output

****** Data in 2d-array a[3][4] *****
a[0][0]=4 a[0][1]=3 a[0][2]=2 a[0][3]=1
a[1][0]=5 a[1][1]=12 a[1][2]=11 a[1][3]=10
a[2][0]=6 a[2][1]=7 a[2][2]=8 a[2][3]=9
```

図 71　図 70 のプログラムの実行結果

```
if（条件 C1 ）{ 操作 1 } else { 操作 2 }
```

です。条件 $C1$ が成り立つならば操作 1 を実行し、条件 $C1$ が成り立たないならば操作 2 を実行します。プログラムでは { 操作 1 } の前、else の前、あるいは { 操作 2 } の前で改行されている場合が普通です。

- もう一つの重要な条件判断文が以下の形式です。

```
if（条件 C2 ）{ 操作 3 }
```

else 以下の部分がない形です。これは条件 $C2$ が成り立つときだけ操作 3 を実行します。条件 $C2$ が成り立たないときは何もせず、次の処理に移ります。この形もプログラム作成ではしばしば出現します。

- ベクトル $\boldsymbol{d} = (3, 3, 3, 2, 4, 1)$ が図 63 あるいは図 68 で示したように 1 次元配列 d の d[1]..d[6] に格納さているとして、偶数データと奇数データそれぞれの個数をカウントするプログラムを考えてみましょう。if〜else 文を使う一つの例です。プログラム例は図 72 となります。その実行結果は図 73 です。
- 図 72 の 11〜15 行は配列 d のデータの表示です。17〜26 行が偶数、奇数のカウントです。要素 d[i] に対する 2 による整数除算[*7]のあまりは d[i]%2 と表します。これが 0 か否かで d[i] の値の偶奇を判定しています。奇数の個数をカウントしなくても、if 文 (else なし) で偶数の個数を得てから全体の個数 6 から引けば求められますが、ここでは if〜else 文を使ってみました。皆さんで動作を追ってみてください。図 73 の説明は省略します。
- なお、19 行目の == は C 言語での等号です。また、22 行目のコメント文中の != は $\neq$ (等しくない) を表す C 言語の書き方です。

1.8 まとめ

第 6 回において作成した、グラフの基点 s から連結な点をすべて求めるアルゴリズム $\mathbf{compo}(G, s)$ をプログラムとして実現することを目指しています。そのためにはプログラミングについての準備が必要です。前回はプログラミングの基礎のそのまた基礎として、コンピュータの仕組、コンピュータ内部でのプログラムの処理過程などについての大まかな説明をしました。今回はプログラミングの直接的な基礎となる

- ベクトルや行列をコンピュータのメモリに配列として格納すること
- これらに関連して繰り返し処理を C 言語の for 文で記述すること

[*7] 例えば、整数除算 $10 \div 3$ の商は 3 であまりは 1 ですので、$10\%3$ は 1 です。

```
1  //checking parity of data in 1d-array
2
3  #include <stdio.h>
4  #define n 6
5
6  int main(void)
7  {
8      int d[n+1]={0,3,3,3,2,4,1};
9      int i, even=0, odd=0;  //even:偶数の個数  odd:奇数の個数
10
11     printf("\n****** Data in 1d-array d[1]..d[%d] *****\n",n);
12     for (i=1; i<=n; i++){
13         printf("d[%d]=%d ",i,d[i]);
14     }
15     printf("\n\n");
16
17     printf("***** Counting the numbers of even/odd elements *****\n");
18     for (i=1; i<=n; i++){
19         if (d[i]%2==0){   // d[i] is even
20             even=even+1;
21         }
22         else{             // d[i]%2!=0 (or d[i] is odd)
23             odd=odd+1;
24         }
25     }      //odd=n-even; として計算してもよい
26     printf("  偶数の個数=%d; 奇数の個数=%d\n\n", even, odd);
27
28     return 0;
29 }
```

図 72　1 次元配列における偶数データ、奇数データの個数計算 (プログラム名: checking-parity.c)。左端の行番号は説明用に書いています。実際のプログラムに書くとエラーになります。

```
Toshimasa-no-MacBook-Pro:repos watanabe$ gcc -o checking-parity checking-parity.c
Toshimasa-no-MacBook-Pro:repos watanabe$ ./checking-parity

****** Data in 1d-array d[1]..d[6] *****
d[1]=3 d[2]=3 d[3]=3 d[4]=2 d[5]=4 d[6]=1

***** Counting the numbers of even/odd elements *****
  偶数の個数=2; 奇数の個数=4
```

図 73　図 72 のプログラムの実行結果

- 条件によって操作を選択して実行する処理を C 言語の条件判断文 (if〜else 文) で記述すること

を中心に説明しました。ここまではプログラミング全般についてのイントロダクションになっています。プログラミングに不慣れな方々も含めて、会員の皆さん、さらには中高生や大学生などの若い方達にもプログラミングの雰囲気が伝わることを期待しています。

さて次に考えるべきことは、グラフをコンピュータでどのようにして扱うかです。ここでは、グラフを 2 次

元配列で扱うこととします。次回はこのような考えに基づいて、2 点が隣接しているかどうかの判定、基点 s から連結な点を求めること、などから始めて、少しずつ我々の目標のプログラム作成に入っていきたいと考えています。