第4章　C言語中級編

4.1　関数

関数とは ＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿ である。関数を使う利点には次のようなものが挙げられる。

- プログラムが階層化され、読みやすくなる。
- 詳細を隠すことが可能になる。
- 同じような処理を何度も繰り返し書かなくて済む。
- 設計の変更が容易になる。

4.1.1　関数の定義

関数の定義は次のような形式で行なう。

```
型 関数名 (型 変数名, ... , 型 変数名)
{
    ...
}
```

中かっこのあいだに、関数の定義本体を書く。

一般に、関数に呼び出し側からデータを渡して関数の振舞いを変えたり、逆に関数から呼び出し側に結果を戻したりすることができる。関数が呼び出し側から受取るデータを ＿＿＿ （＿＿＿＿＿）、関数が呼出し側に返すデータを ＿＿＿＿ または ＿＿＿＿ という。たとえば、sqrt(4) という関数呼出しでは引数は 4 で、戻り値は 2 になる。

上の関数定義の形式で、関数名の前にある型が戻り値の型である。これを省略すると戻り値の型はint と解釈される。戻り値がない関数の場合は特別に ＿＿＿ と書く。

「(」と「)」の中は引数の型と仮引数の宣言である。仮引数とは、引数が関数の呼出しの時に代入される変数である。

たとえば、sqrt という関数の定義は次の形になっている。

```
double sqrt(double x)
{
    ...
}
```

これは、sqrt が double 型の引数を 1 つ受取り、double 型の値を返す関数であることを示している。

一般に関数を呼出すには、

関数名 (式, ... , 式)

のように、括弧の中に式を引数の個数だけ並べる[1]。これらの式（の値）を _____という。これらの式を計算したあと、関数の対応する位置の仮引数にこれらの値を割り当てて、関数の定義を実行する。例えば、sqrt(4) という呼出しがあったときは、実引数の 4 が、仮引数の x に代入されて、関数の本体（ ... の部分）が実行される。

引数が無い関数を呼び出す時には、単に “関数名 ()” というように括弧の中は空になる。

関数が呼び出されるとプログラムは

- 関数の実行が終ったときに処理を再開できるように必要な情報（場所と変数の値）を保存する。
- 関数の先頭から実行を開始する。

関数から値を返すには return 文を用いる。

```
return 式;
```

式[2]を関数の戻り値として呼出し側に制御を返す。for 文や while 文などの繰り返しの中で使用しても、繰り返しから脱出して呼出し側に制御を戻す。そして、関数の戻り値がこの関数呼出しの式全体の値になる。

一方、戻り値のない（戻り値が void 型の）関数の場合、単に

```
return;
```

と書く。関数の最後ではこのreturn 文自体、省略可能である。

main は C 言語にとって特殊な関数名で _____。しかし定義の仕方は他の関数と全く同じである。また、main が 0 を戻り値として返すのは、プログラムの正常終了を表す。

関数の定義は入れ子にできない

関数を定義するときには、次のようなことに注意する。

- C 言語では関数（main を含む）の定義の中に、さらに関数の定義を _____。

  間違いの例:

```
main()
{
    void foo()
    {    /* 関数 main の中で別の関数 foo を定義しようとしている。*/
        printf("foo¥n");
    }
    foo();

    return 0;
}
```

[1] つまり関数を呼び出す時には型に関する情報は不要である。

[2] この式の周りに、括弧は必要ではない。return (式); という書き方は間違いではないが、冗長である。

- （原則として）関数は使用される場所よりも前に、定義・または _____（後述）されていなければならない。

例題 4.1.1 *int* 型を *1* つ受取り、その整数の *2* 乗を返す関数 *sq* を定義する。
ファイル sq.c

```
int sq(int n)
{
    return n * n;
}
```

n が仮引数である。

例題 4.1.2 *int* 型を *1* つ受取り、その整数の階乗を返す関数 *fact* を定義する。
ファイル fact.c

```
int fact(int n)
{
    int i;
    int ret = 1;

    for (i = 1; i <= n; i++) {
        ret *= i;
    }
    return ret;
}
```

このプログラムも *n* が仮引数である。

問 4.1.3 正の整数 n を受け取って 2^n（*2* の n 乗）を計算する関数 *pow2* を定義せよ。

問 4.1.4 正の整数 n, r を受け取って ${}_nC_r \left(= \dfrac{n \cdot (n-1) \cdot \cdots \cdot (n-r+1)}{r \cdot (r-1) \cdot \cdots \cdot 1}\right)$ を計算する関数 *combi* を定義せよ。

問 4.1.5 正または *0* の整数 m, n を受け取って、ユークリッドの互除法により、m と n の最大公約数を計算する関数 *gcd* を定義せよ。

例題 4.1.6 整数を引数として受取り、その数だけ空白を出力する関数 *putSpaces* を定義する。
ファイル putSpaces.c

```
void putSpaces(int n)
{
    int i;

    for (i = 0; i < n; i++) {
        putchar(' ');
    }
    return;   /* この return は省略可能 */
}
```

この関数は戻り値がないので、戻り値の型は *void* と宣言されている。

putchar は文字を *1* 文字だけ出力する関数である。ちなみに、逆に文字を *1* 文字だけ入力する関数は *getchar* である。*char* 型の定数（文字定数）は *'!'* のように「'」（______________________________）で囲んで表す。

当然だが、仮引数の名前は適当に変えても構わない。

```
void putSpaces(int m)
{
    ...
    for (i = 0; i < m; i++) ...
}
```

のように書いても元の putSpaces と同じ関数である。

例題 4.1.7 *0* 以上の整数 n を受取り、$\sum_{i=1}^{n} \frac{1}{i}$ を計算する関数を定義する。

ファイル *sigma.c*

```
double bar(int n)
{
    double ret = 0;
    int i;

    for (i = 1; i <= n; i++) {
        ret += (double)1 / i;
    }
    return ret;
}
```

単に ret+=1/i; と書いてしまうと / が整数の割り算として解釈されてしまうので、n をどんな数にしても戻り値が 1 になってしまう。そこで (double)1 という式で 1 を double 型に型変換（____________）して、/ が実数の割り算として解釈されるようにしている。

キャスト演算子は、このようにかっこの中に型名を入れたものである。

(型) 式

問 4.1.8 いくつかの算術関数を定義し、次のプログラムを適当に改造して試せ。

ファイル fact.c

```
#include <stdio.h>

/* ここに fact など関数の定義を挿入する。*/

main()
{
    double num;

    printf("数を入力してください。");
    scanf("%d", &num);                    /* 必要に応じて %d は %lf に書き換える */
    printf("その数の階乗は %d です。¥n", fact(num)); /* 必要に応じて %d は %f に */

    return 0;
}
```

1. *2* つの *0* 以上の整数 m, n を受取り m^n を計算する関数 *mypow*。
2. *0* 以上の整数 n を受取り、$\sqrt{8\sum_{i=0}^{n}\frac{1}{(2i+1)^2}}$ を計算する関数 *mypi*。
3. *0* 以上の整数 n を受取り、$\sum_{i=0}^{n}\frac{1}{i!}$ を計算する関数 *mye*。

4.1.2 大域変数と局所変数

関数の仮引数や関数の中で宣言した変数などが、もしプログラム全体で通用するとすると、変数の名前の衝突が起こり、変数の名前を考えることが大変なことになってしまう。

幸いこれらの変数は、＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿有効である。このような有効範囲の限られた変数を＿＿＿＿＿＿＿＿と呼ぶ。

局所変数は、原則として:

- 他の関数からは直接利用できない。
- 他の関数に同じ名前の変数があっても、＿＿＿＿＿＿＿＿＿＿＿＿＿＿である。
- 関数の実行が終わったときに、その変数の有効期限は終わる。(つまり、次に関数を呼び出したときに、前回の呼出しの時の値は残っていない。)

また、関数の仮引数も局所変数である。仮引数に対する注意として:

- 関数の中で、仮引数に代入しても、呼出し側に影響を与えない。

例題 4.1.9 ファイル *foo.c*

```
#include <stdio.h>

void foo(int x)
{
    int y;

    x = 3;
    y = 4;
}

main()
{
    int x = 1, y = 2;
    printf("x = %d y = %d¥n", x, y);
    foo(x);
    printf("x = %d y = %d¥n", x, y);

    return 0;
}
```

関数 *foo* の中の変数 *x, y* は、*main* の中の *x, y* とは全く別の変数である。だから *foo* の実行の前後で *main* の *x, y* の値は _____。

一般に、中かっこで囲まれた _____の最初で宣言された変数は、原則として _____ _____有効である。

逆に関数の外で宣言された変数は、_____有効である。このような変数を _____と呼ぶ。

4.1.3 関数の宣言

関数を使用するところより後や、あるいは別のファイルで定義したい場合は、関数を使用する前にとりあえず宣言しておく必要がある。

型 関数名 (型 変数名, ... , 型 変数名);

関数の宣言は関数の定義の中かっこより前の部分を抜き出して、最後に「_」（セミコロン）をつけたものである。ただし仮引数の変数名は省略可能である。

```
int fact(int n);
```

または

```
int fact(int);
```

引数が一つも無い場合は、特別に、括弧の中にvoid と書く。

```
int getchar(void);
```

関数の宣言は関数の中と外のどちらでも可能であるが、通常は関数の外で宣言しておく。

たとえば、fact という関数の定義が、使用されるよりも前にあるプログラム:

```
int fact(int n)
{
    /* fact の定義 */
    ...
}

main()
{
    ...  fact(10) ...  /* fact の呼出し */
}
```

では、宣言は不要である。しかし、次の例のようにこの順番が逆になったり、

```
int fact(int n);          /* 使用する前に宣言が必要 */

main()
{
    ...  fact(10) ...  /* fact の呼出し */
}

int fact(int n)
{
    /* fact の定義は使用されるよりも後ろ */
    ...
}
```

あるいは、fact が別のファイルで定義されているときは関数を使用する前に宣言が必要である。

4.1.4 インクルードファイル

実はこれまで呪文のように使ってきた #include <stdio.h>や #include <math.h>の stdio.h や math.h はこのような関数の宣言などが集められたファイル（＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿）である。#include は、＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿という特殊な命令で、他のコンパイルの作業に先立って実行される。このため#include 命令は、他の C の文と異なり、

- 行の最初から書き始めなくてはいけない。
- 最後にセミコロンをつけない。

という特殊性がある。

putchar や getchar などの入出力に関する関数、sin, cos などの数学関数は C に標準で用意されている関数（＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿）である。これらの関数はstdio.h や math.h に宣言されている。なおこのようにシステムが標準的に用意しているインクルードファイルを読み込むときには #include < ... .h>という形式を、プログラマが用意したインクルードファイルを読み込むときは、#include " ... .h"という形式を使う。（コンパイラがインクルードファイルを検索するフォルダが異なる。）

問 4.1.10 今まで紹介したインクルードファイルとそれを必要とする関数についてまとめよ。

.....
.....

.....
.....
.....

4.1.5　マクロ

#include 文と同じような特殊な命令に、#define 文がある。#define 文は定数に名前をつけるための命令である。

```
#define 名前 文字列
```

この命令も他のコンパイルの作業に先立って実行される。#define 文以降の「名前」を「文字列」に置換する。例えば左のようなプログラムは、右のように展開される。

```
#include <stdio.h>
#define NUM 100

main()
{
    int i;
    for (i = 0; i < NUM; i++) {
        printf("hello!¥n");
    }

    return 0;
}
```

```
#include <stdio.h>


main()
{
    int i;
    for (i = 0; i < 100; i++) {
        printf("hello!¥n");
    }

    return 0;
}
```

このように #define された名前（この例の場合 NUM）を _____という。マクロは大文字から始める慣習がある。

#define 文を用いる意義としては、

- 何度も用いる定数をマクロにすれば、後で変更が容易である。
- プログラムが読みやすくなる。

などがある。例えば、math.h の中では次のようなマクロが定義されている。

```
#define M_PI 3.1415926
```

4.1.6　再帰関数

関数はその定義の中で自分自身を呼出すことができる。このような関数を _____（recursive function）と呼ぶ。再帰関数は自分自身を呼出すということ以外は定義の仕方も呼出しの仕方も全く普通の関数である。

例題 4.1.11 階乗を再帰関数として定義する。

ファイル factrec.c

```
int fact(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fact(n-1);
    }
}
```

これは

$$n! = \begin{cases} 1 & (n = 0) \\ n \cdot (n-1)! & (n > 0) \end{cases}$$

という定義をストレートに C 言語に書き直したものである。

仮引数 n は関数の呼出し毎に、別の領域が割当てられる。一般に、関数の中で宣言された局所変数は

- 同じ関数の中の同じ名前の変数であっても、呼出しが異なれば、別の変数である。

.....
.....
.....
.....

問 4.1.12

$$m^n = \begin{cases} 1 & (n = 0) \\ m \cdot m^{n-1} & (n > 0) \end{cases}$$

という関係を利用して、m^n を求める再帰関数 *mypow2* を定義せよ。

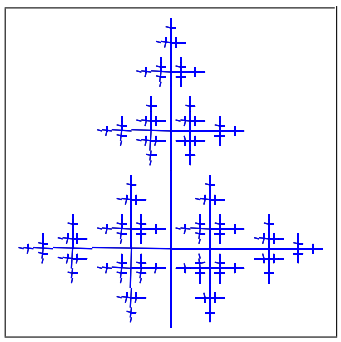
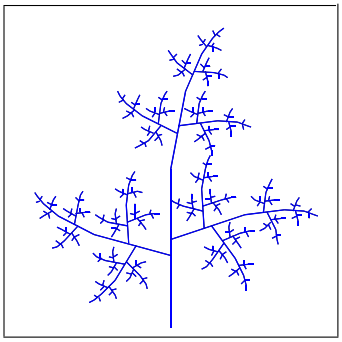
問 4.1.13

$$gcd(m, n) = \begin{cases} m & (n = 0) \\ gcd(n, m\%n) & (n > 0) \end{cases}$$

という関係を利用して、m と n の最大公約数を求める再帰関数 *gcd2* を定義せよ。

一般的には、少しの書換えで繰り返し（for 文や while 文など）にできる再帰関数は（少なくとも C 言語では）繰り返しに書き直した方が効率が良い。しかし、繰り返しに書き直すことが難しい再帰関数もある。次に紹介する drawTree などもその例である。

例題 4.1.14 再帰関数をグラフィックスに利用すると、単純な定義で意外に複雑な図形を描くことができる。左側の図形は下の関数（*drawTree*）が描いたものである。

少しパラメータを工夫すると右側の図形のように本物っぽい樹状図形を描くこともできる。

ファイル *simpleTree.c*

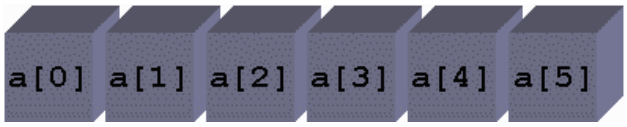
```
#include <stdio.h>
#include <math.h>

void drawTree(int d, double x, double y, double r, double t)
{
    /* d        --- 再帰呼出しの深さ */
    /* (x, y)   --- 枝の根元の座標 */
    /* r        --- 枝の長さ */
    /* t        --- 枝の角度（ラジアン） */
    double r1;
    if (d == 0) return;
    printf("<line x1='%d' y1='%d' x2='%d' y2='%d' stroke='blue' /n>¥n",
             (int)x, (int)y, (int)(x + r * cos(t)), (int)(y + r * sin(t)));
    r1 = r;
    drawTree(d - 1, x + r1 * cos(t), y + r1 * sin(t), 0.5 * r, t + 0);
    r1 = 0.5 * r;
    drawTree(d - 1, x + r1 * cos(t), y + r1 * sin(t), 0.5 * r, t + M_PI/2);
    r1 = 0.5 * r;
    drawTree(d - 1, x + r1 * cos(t), y + r1 * sin(t), 0.5 * r, t - M_PI/2);
}

main()
{
    printf("<?xml version='1.0' ?>¥n");
    printf("<!DOCTYPE svg PUBLIC '-//W3C/DTD SVG 1.0//EN'¥n");
    printf("                     'http://www.w3.org/TR/SVG/DTD/svg10.dtd'>¥n");
    printf("<svg width='%d' height='%d'>¥n", 256, 256);

    drawTree(6, 128, 255, 128, -M_PI / 2);

    printf("</svg>¥n");

    return 0;
}
```

このプログラムは、*double* から *int* のキャスト（型変換）を使っている。*double* から *int* のキャストは切り捨てになる。

問 4.1.15 *drawTree* 中のパラメータ（0.5 や M_PI/2 など）をいろいろ変えて試せ。

4.2 配列

配列とは同じ型ののデータの（フラットな）並びである。



4.2.1 配列宣言

型 変数名 [要素数];
型 変数名 [] = {初期値$_0$, ... , 初期値$_{n-1}$};

1 番目の書き方をすると「型」のサイズの “箱” が「要素数」だけ用意される。
2 番目は初期値を同時に指定する書き方である。この場合「要素数」は省略できる。
例:

```
double a[6];
int b[] = {4, 6, 2, 7, 1, 9};
```

配列の要素は

配列変数名 [式]

という形でアクセスする。例えば a[2] という式は a という変数に入っている配列の 2 番目（?）の箱を表す。この “[” と “]” に囲まれた部分（この例だと2）を配列の _____（index）という。C 言語の場合、配列の添字は 0 から始まる。a は要素数が 6 であるから、

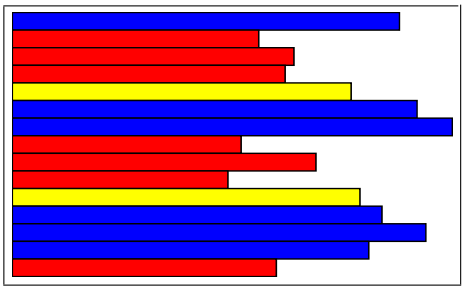
a[0] — _____の箱
a[5] — _____の箱

となる。

例題 4.2.1 整数の配列を棒グラフにする。

ファイル graph.c

```
#include <stdio.h>

int xs[] = {88, 56, 64, 62, 77, 92, 100, 52, 69, 49, 79, 84, 94, 81, 60};
int n = 15; /* xs の要素数 */

main()
{
    int i, color;

    printf("<?xml version='1.0' ?>¥n");
    printf("<!DOCTYPE svg PUBLIC '-//W3C/DTD SVG 1.0//EN'¥n");
    printf("                     'http://www.w3.org/TR/SVG/DTD/svg10.dtd'>¥n");
    printf("<svg width='%d' height='%d'>¥n", 300, 180);

    for (i = 0; i < n; i++) {
        if (xs[i] >= 80)        { /* 優 */
            color = 0x0000ff;  /* 青 */
        } else if (xs[i] >= 70) { /* 良 */
            color = 0xffff00;  /* 黄 */
        } else {                  /* 可以下 */
            color = 0xff0000;  /* 赤 */
        }
        printf("<rect x='0' y='%d' width='%d' height='12' ¥n", i * 12, xs[i] * 3);
        printf("      stroke='black' fill='#%06X' />¥n", color);
    }

    printf("</svg>¥n");

    return 0;
}
```



この例では、点数によってグラフの色を変えている（右図）。（先頭に0x のついているのは、16 進数で表された定数である。）

以前に紹介したcontinue 文を使って、60 点以下の場合はグラフを描かないようにするには次のようにする。

```
    for (i = 0; i < n; i++) {
        if (xs[i] >= 80)        { /* 優 */
            color = 0x0000ff;  /* 青 */
        } else if (xs[i] >= 70) { /* 良 */
            color = 0xffff00;  /* 黄 */
        } else if (xs[i] >= 60) { /* 可 */
            color = 0xff0000;  /* 赤 */
        } else { /* 60 点未満  ---  不可 */
            continue; /* グラフを書かない。*/
        }
        printf("<rect x='0' y='%d' width='%d' height='12'¥n", i * 12, xs[i] * 3);
        printf("      stroke='black' fill='#%06X' />¥n", color);
    }
```

この例の場合、continue; を実行すると、次のprintf を実行せずに、i++に実行が移る。

問 4.2.2 整数の配列を折れ線グラフにするプログラムを作成せよ。

例題 4.2.3 *int* の配列を受取って、要素の最大値を求める関数 *maximum* を定義する。ただし、要素はすべて正または *0* の数とし、データの終わりを示すために負の数を用いるものとする。

ファイル *maximum.c*

```
int maximum(int a[])
{
    int i;
    int max = a[0];

    for (i = 1; a[i] >= 0; i++) {
       if (a[i] > max) {
          max = a[i];
       }
    }
    return max;
}
```

配列を関数の引数として指定するときは、この例題のように配列の要素数を指定する必要はない。ただし、配列そのものから要素数を知る方法はないので、要素数を別の引数として渡すか、上の例題のようにデータの最後に終わりを示す特別な要素を入れておくなどの方法をとる必要がある。

この関数を呼び出すときには、配列を用意して、その配列名を引数として渡せばよい。

```
main ()
{
    int data[] = {52, 63, 12, 98, 34, 48, 22, -1};
    ...   maximum(data) ...
}
```

問 4.2.4 *maximum* と反対に配列の最小値を求める関数 *minimum* を定義せよ。

問 4.2.5 *double* の配列を受取って、要素の和を求める関数 *sumArr* を定義せよ。ただし、要素はすべて正または *0* の数とし、負の数はデータの終わりを示すために用いるものとする。

問 4.2.6 *double* の配列を受取って、要素の平均値を求める関数 *average* を定義せよ。ただし、要素はすべて正または *0* の数とし、負の数はデータの終わりを示すために用いるものとする。

問 4.2.7 （難）長さ *22* の *int* 型の配列を、ボーリングで倒したピンの数とみなして、スコアを計算する関数 *int score(int data[])* を定義せよ。

4.2.2 argc と argv

プログラムをコマンドライン（MS-DOS プロンプト）や「ファイル名を指定して実行」から起動するときは、プログラムに引数（＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿）を渡すことができる。例えば、「bcc32 foo.c」の「foo.c」、「cd ¥windows」の「¥windows」はコマンドラインパラメータである。

コンソールアプリケーションでコマンドラインパラメータを利用するには、main を次のように書く。

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

（char *argv[] の *はポインタ型というものを表すが、ここでは詳しく紹介しない。）
　argc がコマンドラインパラメータの数、argv がコマンドラインパラメータの入った配列である。argv の各要素は文字列である。

例題 4.2.8 コマンドラインパラメータをそのまま出力する。
ファイル argvexample.c

```
#include <stdio.h>

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

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

    return 0;
}
```

実際にこのプログラムを実行してみるとわかるが、*argv[0]* には、実行されるプログラム自身のフルパスが渡される。本来のコマンドラインパラメータは *argv[1]* 以降である。

例題 4.2.9 コマンドラインとして渡された配列をグラフにする（キャラクター版）。
ファイル chargraph.c

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

void chargraph(int n)
{
    int j;
    for (j = 0; j < n; j++) {
        putchar('*');
    }
    putchar('¥n');
}

int main (int argc, char *argv[])
{
    int i;
    for (i = 1; i < argc; i++) {
        chargraph(atoi(argv[i]));
    }

    return 0;
}
```

実行例*:*

```
... >chargraph 1 5 7 8 2
*
*****
*******
********
**
```

この例題では文字列から整数（*int*）への変換に *C* の標準ライブラリ関数 *atoi* を使っている。ちなみに文字列から *double* への変換には *atof* という関数を用いる。（*atoi*, *atof* を使うためには#include <stdlib.h> が必要である。）

問 4.2.10 コマンドラインからいくつかの数値を受取り、最大値、平均値、和などを計算するプログラムを作成せよ。

4.2.3　乱数

（疑似）乱数（でたらめな数）を発生させるには rand という関数を用いる。プログラム中でrand を使うときは、プログラムの最初でrandomize という関数を実行しておく。rand は 0 から、ある非常に大きな数（RAND_MAX というマクロの値、bcc32 の場合、約 20 億）の間の整数を返す。このため

0 から 5 の乱数が必要なときはrand()%6 のような式を用いる。なお、rand(), randomize() を用いるときは、#include <stdlib.h>と#include <time.h>が必要である。

例題 4.2.11 *0〜5* の目が出るサイコロを *5* 個ふるときの合計の数はどういう数になりやすいか、*100* 回試行を繰り返して実験する。

ファイル rand.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define NUM 26

/* 関数 chargraph の定義は省略 */

int main(int argc, char *argv[])
{
    int i, k;
    int result[NUM];  /* 大きさ 26（添字 0 〜 25）の配列 */

    randomize();  /* 問: これがないとどうなるか? */

    for (i = 0; i < NUM; i++) { /* 0 に初期化しておく */
        result[i] = 0;
    }

    for (k = 0; k < 100; k++) {
        int sum = rand() % 6 + rand() % 6 + rand() % 6
                  + rand() % 6 + rand() % 6;
        result[sum]++;
    }
    /* 残りの部分は chargraph.c とほぼ同じ */
   for (i = 1; i < NUM; i++) {
       printf("%2d ", i);
       chargraph(result[i]);
   }

   return 0;
}
```

4.2.4 文字列

C 言語の文字列は、文字（char）型 の配列である。

文字列の終りには**ヌル文字**（'¥0' と書く）という特別な文字が入っている。 これは、'0' という数字（ASCII コードで 48）とは別のものである。

```
char str[] = "Hello!";
```

は、

```
char str[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

と書くのと同じである。（つまり n 文字の文字列は $n+1$ 個の要素を持つchar 型の配列として表現されている。） さらに、これは ASCII コードでは、

```
char str[] = {72, 101, 108, 108, 111, 0};
```

と同じである。

例題 4.2.12 文字列中の文字の個数を求める関数 *mystrlen* を定義する。

ファイル mystrlen.c

```
int mystrlen(char str[])
{
    int i;
    for (i = 0; str[i] != 0; i++) {
        /* 何もしない */
    }
    return i;
}
```

str[i] が *0*、つまり文字列の終の印になるまで、*i* を増やしながら、*str[i]* を調べていく。str[i] が *0* になったときの *i* が文字の個数である。

例題 4.2.13 文字列と文字を受け取り、文字の文字列中の最後の出現位置を返す関数 *int mystrrchr(char str[], char c)* を定義せよ。ただし、全く出現しなければ *-1* を返すものとする。例えば、*mystrrchr("ehime", 'e')* は *4*（最初の文字は *0* 番目）、*mystrrchr("ehime", 'x')* は *-1* である。

ファイル mystrrchr.c

```
int mystrrchr(char str[], char c)
{
    int i, ret = -1;
    for (i = 0; str[i] != '¥0'; i++) {
        /* ↑文字列を操作するときの典型的な for 文の形 */
        if (str[i] == c) {
            ret = i;
        }
    }
    return ret;
}
```

問 4.2.14

1. 文字列と文字を受け取り、文字が文字列中現れる回数を返す関数 int count(char str[], char c) を定義せよ。例えば、count("ehime", 'e') は *2*、*count("ehime", 'x')* は *0* である。
2. 文字列と文字を受け取り、文字の文字列中の最初の出現位置を返す関数 *int mystrchr(char str[], char c)* を定義せよ。ただし、全く出現しなければ *-1* を返すものとする。例えば、mystrchr("ehime", 'e') は *0*（最初の文字は *0* 番目）、mystrchr("ehime", 'x') は *-1* である。

4.2.5 多次元配列

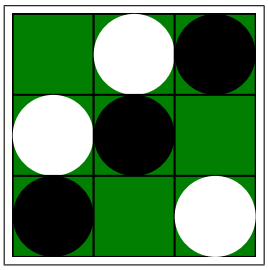
当然、配列の要素がまた配列であっても良い。そのような配列は _____ と呼ばれる。

例:

```
double a[3][4];

int b[2][12] = {{31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
                {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}};
```

多次元配列の宣言の場合、初期値を与えても、最初の次元の要素数しか省略できない。上の b の場合、 b[][12] とはできるが、b[][] や b[2][] とはできない。

例題 4.2.15



int 型の *3*× *3* の大きさの配列の配列を調べて、 *1* なら白丸、*2* ならば黒丸を画面上の対応する位置に描画する（右図）。

ファイル *othello.c*

```
#include <stdio.h>

#define SIZE 3
#define NONE  0
#define WHITE 1
#define BLACK 2
int xs[SIZE][SIZE] = {{NONE, WHITE, BLACK},
                      {WHITE, BLACK, NONE},
                      {BLACK, NONE, WHITE}};

main ()
{
    int i, j;

    printf("<?xml version='1.0' ?>¥n");
    printf("<!DOCTYPE svg PUBLIC '-//W3C/DTD SVG 1.0//EN'¥n");
    printf("                     'http://www.w3.org/TR/SVG/DTD/svg10.dtd'>¥n");
    printf("<svg width='%d' height='%d'>¥n", 120, 120);

    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++) {
            printf("<rect x='%d' y='%d' width='40' height='40'¥n",
                   i * 40, j * 40);
            printf("      stroke='black' fill='green' />¥n");
            if (xs[i][j] == WHITE) {
                printf("<circle cx='%d' cy='%d' r='20' ¥n",
                       i * 40 + 20, j * 40 + 20);
                printf("        stroke='none' fill='white' />¥n");
            } else if (xs[i][j] == BLACK) {
                printf("<circle cx='%d' cy='%d' r='20' ¥n",
                       i * 40 + 20, j * 40 + 20);
                printf("        stroke='none' fill='black' />¥n");
            }
        }
    }

    printf("</svg>¥n");

    return 0;
}
```

キーワード:

関数、引数、戻り値、void、仮引数、return 文、実引数、キャスト、インクルードファイル、マクロ、再帰、局所変数、大域変数、配列、添字、コマンドラインパラメータ、文字列、文字列定数、