# 計算物理学II (第 7 回)

# 今回の内容

- 第５-６回の復習
- 関数副プログラム・サブルーチン副プログラム(Fortran)
- 関数(C)

  - 何度も使う一連の実行文をまとめて主プログラムの外にまとめる。
  - 数学の関数と考えてよいが、もっと複雑なものも関数にできる。

# プログラムの基本構造

Fortran

```
program プログラム名

宣言部 (implicit none宣言、変数宣言)


実行部


end program プログラム名
```

プログラム名は英字から始まる適当な名前

C

```
#include<ヘッダファイル名.h>

int main(){
    宣言部 (変数宣言など)

    実行部 (計算など)

}
```

プログラムはmain関数として実行される。
ヘッダファイルにはプログラム内で
ライブラリ関数などを用いる場合に読み込む。
数値計算で必要となるのは
stdio.h,math.h,stdlib.h,complex.hなど

# 変数

- 変数：数値・文字などのデータを格納する領域。名前をつけて区別。
- 型があり、FortranやCでは宣言部で全ての型を明示的に指定する

| 型 | Fortran | C | サイズ |
|---|---|---|---|
| 整数型 | **integer** | **int** | **4バイト（= 32ビット）** |
| 単精度実数型<br>(物理の数値計算ではあまり使わない) | real, real*4, real(4) | float | 4バイト |
| 倍精度実数型 | double precision, **real*8**, real(8) | **double** | **8バイト** |
| 複素数型(物理の数値計算ではあまり使わない) | complex, complex*8 | float complex<br>(complex.hが必要) | 8バイト |
| 倍精度複素数型 | complex(kind(0d0)),<br>**complex*16**, double complex | double complex<br>(complex.hが必要) | 16バイト |
| 論理型 | **logical** | bool (stdbool.hが必要) | trueかfalseか |
| 文字型 | **character** | **char** | **1文字1バイト** |

太字はよく使うもの。
変数名：英数字とアンダーラインが使える。英字から始まる

# 四則演算と代入文

- 数学の=(イコール)は左辺と右辺が等しいことを表すが
  プログラムの=は**右辺の計算結果を左辺の変数に代入する**、の意味
- 四則演算は +, -, *, /
- 累乗は Fortranでは** で表す。Cにはない (何度も掛けるか、数学関数のpowを使う)
- **同じ型同士の演算は同じ型になる。**
- 整数と実数の演算→実数、実数と複素数の演算→複素数

Fortran

```
a = 1.0d0
i = i + 5
a = 4**3
a = 5 / 2
```

C

```
a = 1.0;
i = i + 5; または i += 5;
a = 4*4*4; または a = pow(4.0,3.0);
a = 5 / 2;
```

Cは文の終わりに;が必要

倍精度実数型のaに1.0を代入
整数型のiの値を5増やす
aに4の3乗を代入
aは整数型でも実数型でも
2となる(整数と整数の演算は整数)
2.5にしたければ
5.0/2や5.0/2.0、5/2.0などを代入

# 実行文

- プログラムの実行文は上から順番に一行ずつ実行される(重要)

Fortran

```
integer :: a, b, c

a = 2
b = 5
c = a + b
```

C

```
int a, b, c;

a = 2;
b = 5;
c = a + b;
```

整数変数a,b,cを宣言

変数aに2を代入
変数bに5を代入
変数cにa + bを代入。この時点でaとbに値が入っているので
2+5が計算されて変数cに7が代入される

```
integer :: a, b, c

c = a + b
a = 2
b = 5
```

```
int a, b, c;

c = a + b;
a = 2;
b = 5;
```

順番を変えると結果が変わる。
cにa+bを代入。aとbに値がまだ入っていない。システム依存になるが
おそらくaにもbにも初期値の0が入っているのでcもおそらく0となる
aに2を代入
bに5を代入
cの値はその後変更していないので0のまま。

数学的には同じに見えるがプログラムでは結果は異なる。
プログラムを書くときは**右辺に使う変数に値が入っているかを確認**しながら書く

# ループ(doループ/forループ)

- 実行文は上から一行ずつ実行するのが原則だがよく使う例外は2つ(ループ、条件文)
- doループ(Fortran)：do~end doで囲まれた領域をカウンタを変えて繰り返し実行
- forループ(C) **for( 式1(実行文); 式2(条件文); 式3(実行文)){繰り返し文}**
  - 式1を実行(初期化)、式2の条件を判定、満たされていれば繰り返し文を実行
  - 式3を実行してから式2の条件判定に戻る

Fortran

```
sum = 0
do i = 1, n
    sum = sum + i
end do
```

C

```
sum = 0;
for(i = 1;i<n ; i++){
    sum = sum + i;
}
```

$$\sum_{i=1}^{n} i$$ を計算していることになっている

ループは和や積の計算、ベクトル、行列などindexを変えて行う計算、
パラメータを変えて同じ計算を繰り返すときなどに使われる

# 標準入出力

- 標準入力(キーボードからの入力)

Fortran

```
read (*,*) var
read (5,*) var1, var2
```

C

```
scanf(“%d”, &var):
scanf(“%lf %lf”, &var1, &var2);
```

C言語では入出力の型を明示する必要がある
%d：整数型、%lf倍精度実数型
scanfのときには変数名の前に&をつける
(変数のアドレス)

- 標準出力(画面への出力)
- 標準エラー出力(画面への出力)

```
write (*,*) var
write (6,*) var
write (0,*) var
print *, var
```

```
printf(“%d”, var);
printf(”var1 = %f,   var2 = %f¥n”, var1,var2);

fprintf(stderr, “var1 = %f¥n”, var1); //エラー出力
```

C言語の単精度・倍精度実数の出力は%fを使う

Fortran: read, writeの１つ目の引数で**ファイル装置番号**を指定する。*とすると標準入出力になるが、
Fortranでは5が標準入力、6が標準出力、0が標準エラー出力となっている
**シェル側でリダイレクトを使うことで標準入出力先をファイルなどに変更することができる**
**計算した結果はwriteかprintで必ず出力する**

# 組込関数

- 数学関数 (log, exp, sqrt, sin, cos, tanなど)は用意されている
- 角度の単位はラジアン

- C言語で数学関数を使うときの注意：
  - math.hをインクルードする
  - コンパイル時にオプション –lm をつける
  - **gcc ex4.c –lm**

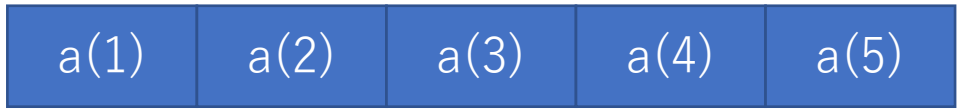
# 配列

- 配列とは：変数が複数並んだもの(メモリ上でも並んで配置される)。ベクトルや行列が表現できる。

**配列の宣言の基本**

Fortran

```
double precision :: a(5)   ! aは5つの要素を持つ倍精度実数変数の配列
```




←メモリ上にこのように5つの倍精度実数変数の領域が連続して確保される

C

```
double a[5];    // aは5つの要素を持つ倍精度実数変数の配列
```


←C言語では配列のラベルは0から始まる。
メモリ上にこのように5つの倍精度実数変数の領域が連続して確保される

# 配列の初期値設定

宣言部で設定

Fortran

```
double precision ::  a(1:5) = 0.0d0
integer :: k(1:5) = (/1, 2, 3, 4, 5/)
```

C

```
double a[5] = {0.,0.,0.,0.,0.};
int k[5] = {1, 2, 3, 4, 5};
```

実行文として設定してもよい(こちらのほうが自由度が高い)。**ループを使う**

```
do i = 1, 5
    a(i) = 0.0d0
end do
```

```
for (i=0;i<5;i++){
    a[i] = 0.0;
}
```

異なる配列の違う添字を持つ要素の代入・演算も可能

```
do i = 1, 5
     a(i) = b(i+3)
end do
```

```
for(i=0;i<5;i++){
     a[i] = b[i+3];
}
```

# 1次元配列の計算例

ベクトル和

ベクトルの内積

Fortran

```
real*8, dimension(1:5)  :: a, b, c
real*8 :: adotb

! ここでaとbには値を代入しておく(省略)

do i = 1, 5
    c(i) = a(i) + b(i)
end do
do i = 1, 5
    write(*,*) “c(“, i, “)=“,  c(i)
end do

adotb = 0.0d0
do i = 1, 5
    adotb = adotb + a(i)*b(i)
end do
print *, “a dot b = “, adotb
```

C

```
double a[5], b[5], c[5];
double adotb;

// ここでaとbには値を代入しておく(省略)

for( i = 0; i<5; i++){
    c[i] = a[i] + b[i];
}
for( i = 0;i<5;i++){
    printf(“c(%d)=%f¥n”, i,  c[i]);
}

adotb = 0.0;
for( i = 0; i<5; i++){
    adotb += a[i]*b[i];
}
printf(“a dot b = %f¥n“, adotb);
```

# 1次元配列の組み込み関数(Fortran)

和

```
c = sum(a(1:10))        ! a(1)からa(10)までの和をcに代入
```

最大値・最小値

```
d = maxval(a(1:n))   ! a(1)からa(n)の中の最大値をdに代入
e = minval(a(1:n))   ! a(1)からa(n)の中の最小値をeに代入
```

2つの1次元配列の内積

```
f = dot_product(a(1:n), b(1:n)) ! n要素の配列aとbの内積をfに代入
```

a, bが複素数型の場合は aの複素共役とbの積が計算される

# 行列

- 2次元配列で行列が表現できる
- 2次元配列の宣言と初期化

```
real*8, dimension(1:5,1:5) :: b
または real*8 :: b(5,5)
または real*8 :: b(1:5,1:5)など

do j = 1, 5
    do i = 1, 5
        b(i,j) = 0.0d0
    end do
end do
```

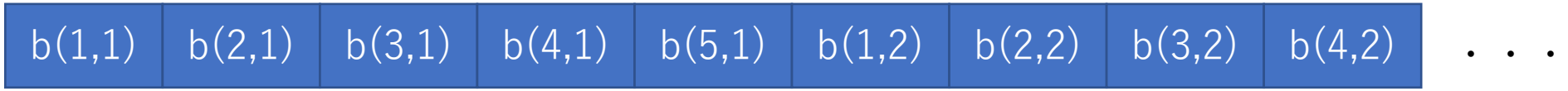
```
double b[5][5];

for(i=0;i<5;i++){
    for(j=0;j<5;j++){
        b[i][j] = 0.0;
    }
}
```

Fortran: b(1:5,1:5) = 0.0d0　!0を代入するだけなら1行でもかける。

# 2次元配列のメモリ上での配置

Fortranでの多次元配列 (b(5,5)の場合) は




C言語での多次元配列(b[5][5]の場合)は


の順でメモリ上に配置される。
doループ/forループで配列の値を使って計算する場合はメモリ上で連続的にアクセスするほうがよい(キャッシュミスが少なくなる)

例えばFortranではdoループは１つ目のインデックスを先に回すほうがよい

```
do j = 1, 5
    do i = 1, 5
        b(i,j) = 0.0d0
    end do
end do
```

# 行列の和・値の出力

Fortran

```
real*8, dimension(1:5,1:5) :: a, b, c

! a, bに値をここで代入(省略)

do j = 1, 5
    do i = 1, 5
        c(i,j) = a(i,j) + b(i,j)
    end do
end do

do i = 1, 5
    write(*,*) (c(i,j), j = 1, 5)
end do
```

C

```
double a[5][5], b[5][5], c[5][5];

/* a, bに値をここで代入(省略) */

for(i = 0; i<5; i++){
    for(j = 0; j<5; j++){
        c[i][j] = a[i][j] + b[i][j];
    }
}

for (i = 0; i<5; i++){
    for(j = 0; j<5; j++){
        printf(" %f ", c[i][j]);
    }
    printf("¥n");
}
```

# 行列の積

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$ 一つの要素を計算するのにループが必要

Fortran

```
real*8, dimension(1:5,1:5) :: amat, bmat, cmat

! amatとbmatの値を代入

do j = 1, 5
   do i = 1, 5
      cmat(i,j) = 0.0d0
   end do
end do

do j = 1, 5
   do i = 1, 5
      do k = 1, 5
         cmat(i,j) = cmat(i,j) + amat(i,k) * bmat(k,j)
      end do
   end do
end do
```

C

```
double amat[5][5], bmat[5][5], cmat[5][5];

/* amatとbmatの値をここで代入 */

for(i = 0; i<5; i++){
   for(j = 0; j<5; j++){
      cmat[i][j] = 0.0;

      for (k = 0; k<5; k++){
         cmat[i][j] += amat[i][k] * bmat[k][j];
      }
   }
}
```

# 2次元配列の組み込み関数(Fortran)

転置行列

```
bmat(1:n,1:n) = transpose(amat(1:n,1:n))     ! bmatにamatの転置行列を代入
```

行列積

```
cmat(1:n,1:m) = matmul( amat(1:n,1:k), bmat(1:k,1:m))
! (n×k行列のamat とk×m行列のbmatの積をcmatに代入(n×m行列)
```

matmulは経験上計算速度が早くないので大行列の計算では他のライブラリを使うほうがよい

# 関数副プログラム(Fortran)

- 関数：与えられた入力変数に対して操作をして値を返す一連の手続き
- 例えば $4x^3-5x$を何度も様々なxに対して計算したいとする。
  プログラムのあちこちに$4x^3-5x$と繰り返して書くとミスのもとになる。
  また後から関数形に変更が必要になったときにすべて探し出して変更する必要がある。
- 関数副プログラムとして$f(x)=4x^3-5x$を計算する部分を独立させる。

- 関数副プログラムを置く場所
  1. 主プログラム(program ~end program) の中に置く(内部副プログラム)
  2. 主プログラムの外(end programの後)に置く(外部副プログラム)
  3. モジュールの中に置く(モジュール副プログラム)

# 関数副プログラム

**例１：主プログラムの中に関数副プログラムを置く場合(内部副プログラム)**

```
program example2
    implicit none
    double precision :: x
    x = 0.0d0
    do
        print *, x, func(x)     ! ここでfunc(x)が呼び出される。関数はcontainsより下で定義されている。
        x = x + 0.10d0
        if( x > 5.0d0) exit
    end do
contains                        ! contains以下に副プログラムを列挙する
    function func(x)            ! 関数見出し
        double precision, intent(in) :: x   ! xを倍精度実数として宣言。intent(in)がつくと関数内で値を変更できない
        double precision :: func            ! 関数の型を宣言する(倍精度実数型)
        func =  4.0d0*x**3 – 5.0d0*x
        return                              ! 関数を呼び出された元のプログラムに戻る
    end function func
end program example2
```

# 関数副プログラム

**例2：主プログラムの外に関数副プログラムを置く場合(外部副プログラム)**

```
program example2
    implicit none
    double precision :: x
    double precision :: func  ！外部副プログラムは主プログラムから見えないので型宣言が必要
    x = 0.0d0
    do
        print *, x, func(x)    ！ここでfunc(x)が呼び出される。
        x = x + 0.10d0
        if( x > 5.0d0) exit
    end do
end program example2
function func(x)          ！関数見出し
    double precision, intent(in) :: x   ！xを倍精度実数として宣言。intent(in)をつけると関数内で値を変更できない
    double precision :: func             ！関数の型を宣言する(倍精度実数型)
    func =  4.0d0*x**3 – 5.0d0*x
    return                    ！関数を呼び出された元のプログラムに戻る
end function func
```

# 関数副プログラム

**例3: モジュールの中に置く（モジュール副プログラム）**

```
module mod_func          ! program よりも手前にmoduleを配置。mod_funcは適当につけたモジュール名
  implicit none          ! module内でもimplicit none宣言
contains                 ! containsより下にモジュール副プログラムを配置
  function func(x)       ! 関数見出し
    double precision, intent(in) :: x   ! xを倍精度実数として宣言。intent(in)をつけると関数内で値を変更できない
    double precision :: func            ! 関数の型を宣言する(倍精度実数型)
    func =  4.0d0*x**3 - 5.0d0*x
    return               ! 関数を呼び出された元のプログラムに戻る
  end function func
end module mod_func
program example2
  use mod_func           ! モジュールを使う場合はimplicit noneより手前でuse モジュール名 としてモジュールを参照
  implicit none
  double precision :: x = 0.0d0
  do
     print *, x, func(x)    ! ここでfunc(x)が呼び出される。
     x = x + 0.10d0
     if( x > 5.0d0) exit
  end do
end program example2
```

# 関数副プログラムの例：外積を計算

```
program test
    implicit none
    double precision :: a(1:3), b(1:3), c(1:3)
    a(1) = 6.0d0; a(2) = 3.0d0; a(3) = 4.0d0
    b(1) = 3.0d0; b(2) = -2.0d0; b(3) = -4.0d0
    c(1:3) = vectorproduct( a(1:3), b(1:3) )    ! 3要素の1次元配列型の関数
    write(*,*) c(1:3)
contains
    function vectorproduct(a, b)
        double precision, intent(in) :: a(1:3),  b(1:3)
        double precision :: vectorproduct(1:3)   ! 関数の型を要素数3の1次元配列として宣言
        vectorproduct(1) = a(2)*b(3)-a(3)*b(2)
        vectorproduct(2) = a(3)*b(1)-a(1)*b(3)
        vectorproduct(3) = a(1)*b(2)-a(2)*b(1)
        return
    end function vectorproduct
end program test
```

# 関数副プログラムの例(再帰関数)

- 再帰(recurrsion)とは：関数副プログラム(やサブルーチン)の中で自分自身(関数副プログラムやサブルーチン)を引用すること。
- 再帰的に関数副プログラム(やサブルーチン)を呼び出す場合はrecursiveをつける。
- 関数の実行結果は関数名ではなくresult句で指定された変数に代入される。

**フィボナッチ数列の例**

```
recursive fuction fibonacci(n) result(fibo)
    integer, intent(in) :: n
    integer :: fibo                ！結果を代入する変数
    if (n<0) then
        fibo = -1
        return
    end if
    if (n == 0) then
        fibo = 0
    else if (n == 1) then
        fibo = 1
    else
        fibo = fibonacci(n-1) + fibonacci(n-2)  ！ 自分自身を呼び出す
    end if
    return
end function fibonacci
```

# サブルーチン副プログラム

- 関数副プログラムとほぼ同じ。違う点は
  - 値を返さない(関数には型があるがサブルーチンにはない)
  - ただし引数として(複数の)値を返すことができる。
  - call文で呼び出す
  - サブルーチン副プログラムは1つの実行文となるが、関数副プログラムは実行文ではない
- 長いプログラムを書く時は主プログラムに長々と書かずサブルーチンに分割する

# サブルーチン副プログラム

```
program example2
    implicit none
    double precision :: x, y
    x = 0.0d0
    do
        call func(x,y)          ! call文でfuncサブルーチンを呼び出し
        print *, x, y
        x = x + 0.10d0
        if( x > 5.0d0) exit
    end do
contains                        ! contains以下に副プログラムを列挙する
    subroutine func(x, y)           ! サブルーチン見出し。入力も出力も引数に入れる。
        implicit none                ! サブルーチンではimplicit noneを使うことを推奨
        double precision, intent(in) :: x      ! intent(in)がつくとサブルーチン内で値を変更できない
        double precision, intent(out) :: y     ! intent(out)がつくとサブルーチン内で値をセットしなければならない
        y =  4.0d0*x**3 – 5.0d0*x
        return                       ! 関数を呼び出された元のプログラムに戻る
    end subroutine func
end program example2
```

# サブルーチン副プログラム

- サブルーチンの引数にはintent属性をつけることを推奨
  - intent(in) : 入力引数、サブルーチンに値が渡され、サブルーチン内で値の変更ができない
  - intent(out) : 出力変数、サブルーチンで値がセットされる変数
  - intent(inout) : 入出力変数、サブルーチンに値が渡され、サブルーチンで値を変更できる変数
  - サブルーチン内だけで使う変数には何もつけない。

# サブルーチン例：外積を計算

```
program test
    implicit none
    double precision :: a(1:3), b(1:3), c(1:3)
    a(1) = 6.0d0; a(2) = 3.0d0; a(3) = 4.0d0
    b(1) = 3.0d0; b(2) = -2.0d0; b(3) = -4.0d0
    call vectorproduct( a(1:3), b(1:3), c(1:3) )    ！入力も出力も引数に入れる
    write(*,*) c(1:3)
contains
    subroutine vectorproduct(a, b, c)
        implicit none
        double precision, intent(in) :: a(1:3),  b(1:3)
        double precision, intent(out) :: c(1:3)
        c(1) = a(2)*b(3)-a(3)*b(2)
        c(2) = a(3)*b(1)-a(1)*b(3)
        c(3) = a(1)*b(2)-a(2)*b(1)
        return
    end subroutine vectorproduct
end program test
```

# 変数の共有(内部副プログラムの場合)

- 変数を主プログラムと副プログラム(関数やサブルーチン)で共有する
- 基本は引数として渡すのが間違いが少ない

```
program test
  implicit none
  real*8 :: x0, a=1.0d0, b=2.0d0, c=3.0d0
  x0 = 1.0d0
  print *, x0, f(x0)            ! 副プログラムのxとしてx0を渡す
  a = 2.0d0
  print *, x0, f(x0)            ! aの値が変わったのでf(x0)の値が変わる
contains
  function f(x)                 ! 2次関数を計算する関数副プログラム
    real*8, intent(in) :: x     !  xは引数として主プログラムから副プログラムに渡される
    real*8 :: f
    f = a*x**2 + b*x + c        !  a, b, cは主プログラムの値を参照する
    return
  end function f
end program test
```

短いプログラムの場合は内部副プログラムが簡単に書けるが、**内部副プログラムは独立性が低い**
(この副プログラムだけ他のコードにコピーしても使えない(a,b,cが中で未定義なので))

# 変数の共有(外部副プログラムの場合)

- 基本は引数にすべて書く。引数に書かれていないものは参照できない

```
program test
  implicit none
  real*8 :: x0, a=1.0d0, b=2.0d0, c=3.0d0, f   ! a,b,cの値は外部プログラムからは直接参照できない
  x0 = 1.0d0
  print *, x0, f(x0,a,b,c)          ! x0以外にa,b,cも引数として関数に渡す
  a = 2.0d0
  print *, x0, f(x0,a,b,c)
end program test
function f(x,a,b,c)                 ! 様々なa,b,cの2次関数を計算をしたい場合はa,b,cも引数に入れる
  real*8, intent(in) :: x,a,b,c
  real*8 :: f
  f = a*x**2 + b*x + c
  return
end function f
```

**外部プログラムは独立性が高い**。計算に必要な変数がすべて引数に入っているので外部プログラムだけを他のコードで使うことができる。ただし引数が煩雑になる。

# 変数の共有(モジュール副プログラム)

- モジュール内に変数を記述し、モジュールを参照する
- 引数には色々な値で計算することを想定されるものだけを入れる

```
module func_mod !モジュールはprogramより手前に配置
  implicit none
  real*8, save :: a, b, c
!副プログラムで使う変数をsave属性つきで定義
!save属性をつけておくと前に代入された値を保持し続ける
contains
  function f(x) !  モジュール副プログラム
    real*8, intent(in) :: x
    real*8 :: f
    f = a*x**2 + b*x + c
!モジュール副プログラムはモジュール内変数を参照できる
    return
  end function f
end module func_mod
```

```
program test
  use func_mod !implicit noneの手間でモジュールを参照
  implicit none
  real*8 :: x0
  a=1.0d0; b=2.0d0; c=3.0d0 !モジュール変数に値をセットできる
  x0 = 1.0d0
  print *, x0, f(x0) !モジュール関数も呼び出せる
  a = 2.0d0
  print *, x0, f(x0)
end program test
```

**モジュールは独立性が高い。**モジュールの中で他のモジュールをuse文で参照することも可能
module, programの順番に書く(別のファイルに分ける場合はこの順番にコンパイル)
複数のmoduleがある場合は順番(依存関係)に注意

# 関数(C言語)

- 関数：与えられた入力変数に対して操作をして値を返す一連の手続き
- mainも関数。int型であれば整数値をreturnで返す。
- 例えば $4x^3-5x$を何度も様々なxに対して計算したいとする。
  プログラムのあちこちに$4x^3-5x$と繰り返して書くとミスのもとになる。
  また後から関数形に変更が必要になったときにすべて探し出して変更する必要がある。
- 関数として$f(x)=4x^3-5x$を計算する部分を独立させる。

- 関数を置く場所
  1. main関数の手前
  2. main関数の後ろ

- return文は関数の処理を中断してその段階の値を返す。

# 関数

**例１：関数funcがmain関数より前に読み込まれる場合**

```
#include<stdio.h>

double func(double x){
    double y;
    y = 4.0*x*x*x – 5.0*x;
    return(y);                    // 倍精度実数型変数yに入っている値を返す
}

int main(){
    double x;

    for(x=0.0;;){
    printf(“%f %f¥n”, x, func(x));  // ここで関数にxの値を渡して呼び出す
    x += 0.1;
    if( x > 5.0) break;
    }
}
```

# 関数

**例2：関数funcがmain関数の後に読み込まれる場合**

```
#include<stdio.h>

int main(){
    double x;
    double func(double);  // 関数プロトタイプ宣言。mainより手前でもよい。
    for(x=0.0;;){
    printf("%f %f¥n", x, func(x));  // ここで関数にxの値を渡して呼び出す
    x += 0.1;
    if( x > 5.0) break;
    }
}

double func(double x){
    double y;
    y = 4.0*x*x*x – 5.0*x;
    return(y);                    // 倍精度実数型変数yに入っている値を返す
}
```

# 再帰関数

- 再帰(recursion)とは：関数で自分自身を引用すること

**フィボナッチ数列**

```
int fibonacci(int n){
    int fibo;
    if( n<0 ) {
        fibo = -1;
        return(fibo);
    }
    if( n==0 ){
        fibo = 0;
    }else if(n == 1) {
        fibo = 1;
    } else {
        fibo = fibonacci(n-1) + fibonacci(n-2);  // 自分自身を呼び出す
    }
    return(fibo);
}
```

# 関数での複数の値・配列の受け渡し

関数の戻り値は一変数なので複数変数、あるいは配列の値を戻り値としてほしいときは
引数に**変数や配列のアドレス**を渡し、関数の中でその変数の値を更新する。
アドレス：メモリ上での番地。変数aの場合は&aがaのアドレス。配列a[n]の場合はa、または&a[0]が先頭のアドレス

**例：外積を計算**

```
#include<stdio.h>
int main(){
    double a[3], b[3], c[3];
    void vectorproduct(double [3], double [3], double [3]); // voidは戻り値がないタイプの関数
    a[0] = 6.0; a[1] = 3.0; a[2] = 4.0;
    b[0] = 3.0; b[1] = -2.0; b[2] = -4.0;
    vectorproduct( a, b, c );                    // 配列のアドレスを渡す。(配列名が配列のアドレス)
    printf("%f %f %f ¥n", c[0], c[1], c[2]);
    return 0;
}
void vectorproduct(double a[3], double b[3], double c[3]){  // double *a, double *b, double *cでもよい
        c[0] = a[1]*b[2]-a[2]*b[1];
        c[1] = a[2]*b[0]-a[0]*b[2];
        c[2] = a[0]*b[1]-a[1]*b[0];
        return;                                  // 関数に戻り値はない
}
```

# 変数の共有

- 基本は引数にすべて書く。引数に書かれていないものは参照できない

```
#include<stdio.h>
double f(double x, double a, double b, double c){    // 2次関数を計算する関数
    return(a*x*x + b*x + c);
}

int main(){
  double  x0;
  x0 = 1.0;
  a = 1.0; b = 2.0; c = 3.0;
  printf("%f %f¥n", x0, f(x0,a,b,c));  // 関数にx0,a,b,cを渡す
  a = 2.0;
  printf("%f %f¥n", x0, f(x0,a,b,c));  // aの値が変わったのでf(x0,a,b,c)の値が変わる
}
```

**関数fの独立性が高い。**計算に必要な変数がすべて引数に入っているので
この関数だけを他のコードで使うことができる。ただし引数が煩雑になる。

# 変数の共有(グローバル変数)

- グローバル変数をmain関数とその他の関数で共有する。
- 関数はグローバル変数に依存するため独立性が低くなる
- 基本は引数として渡すのが間違いが少ない

```
#include<stdio.h>
double a, b, c;                    // グローバル変数。mainの外側で宣言する

double f(double x){                // 2次関数を計算する関数副プログラム
    return(a*x*x + b*x + c);       // a, b, cはグローバル変数の値を参照する
}

int main(){
  double  x0;
  x0 = 1.0;
  a = 1.0; b = 2.0; c = 3.0;
  printf("%f %f¥n", x0, f(x0));  // 関数のxとしてx0を渡す
  a = 2.0;
  printf("%f %f¥n", x0, f(x0));  // aの値が変わったのでf(x0)の値が変わる
}
```

# main関数について

main関数は整数型とすることが多い
正常終了は0を返し、異常終了の場合は0以外を返す
返り値はシェルから参照することができる。省略してもよい。

```
int main(){
        return (1);   // エラーの場合などは1を返す
}
```

```
hinohara.nobuo.ga@icho:~$ cat test.c
int main()
{
  return(1);
}
hinohara.nobuo.ga@icho:~$ gcc test.c
hinohara.nobuo.ga@icho:~$ ./a.out
hinohara.nobuo.ga@icho:~$ echo $?
1
hinohara.nobuo.ga@icho:~$
```

変数$?に一つ手前のコマンドの返り値が格納される

# 第2回レポート

- 締め切り 12月15日(金)
- 講義資料のページに掲載(https://wwwnucl.ph.tsukuba.ac.jp/~hinohara/compphys2-23/)
- 作ったプログラムと回答をpdfファイルにしてmanabaで提出。
- 講義第7回までの内容です。
- FortranまたはCで作成(C++やPythonでもOK)
- 授業中・チャット・メール等での質問も歓迎します。