

```
// 整数 1 からNまでの、奇数の 総和を求めるプログラム
```

```
// program2b.c
```

```
#include <HU.h>
```

```
void Main(void)
```

```
{
```

```
    char        yn;
```

```
    int         i, N, sum ;
```

```
    TextWindow(0,0,300,300); Title("Program2b");
```

```
START:
```

```
    printf("¥n N = "); scanf( "%d", &N );
```

```
    sum = 0 ;
```

```
    for( i = 1 ; i <= N ; i += 2 ){ sum += i ; } // i = i+2 でも良い
```

```
    printf("¥n Sum of Odd numbers = %d ", sum );
```

```
    printf("¥n¥n Retry ? (no;n) "); scanf("%c",&yn);if(yn!='n') goto START;
```

```
    exit(0);
```

```
}
```

```
// 0 からN までの、0.1刻みの実数 0.0, 0.1, 0.2, ... N の  
// 総和、平均、分散、標準偏差を求めるプログラム program2b.c
```

```
#include <HU.h>
```

```
void Main(void)
```

```
{
```

```
    char        yn;
```

```
    int         n;
```

```
    double      N, i, sum, mean, v, sd ;
```

```
    TextWindow(0,0,300,300);    Title("Program2b");
```

```
START:
```

```
    printf("¥n N = "); scanf("%lf", &N);
```

```
    sum = 0.0 ; v = 0.0 ; n = 0 ;
```

```
    for(i = 0.0 ; i <= N ; i += 0.1 ){ sum += i ; n++; }  
                                           // i の増分は 0.1
```

```
    mean = sum / ( double )n ;           // n は 数字の個数を示す
```

```

for( i = 0.0 ; i <= N ; i += 0.1 ){           // i の増分は 0.1
    v += ( mean - i ) * ( mean - i );
}

v /= ( double )n ;                          // v = v / ( double )n ;

sd = sqrt(v);

printf( "¥n Sum      = %lf ", sum );
printf( "¥n Mean    = %lf", mean);
printf( "¥n Variation = %lf", v );
printf( "¥n Deviation = %lf", sd );

printf( "¥n¥n Retry ? (no;n) " ); scanf( "%c", &yn);
if( yn != 'n' ) goto START;

exit(0);
}

```

```
for( i = 0.0 ; i <= N; i += 0.1 ){  
    v += ( mean - i ) * ( mean - i );  
}
```

```
for(初期条件; 実行条件; 更新条件){  
    プログラム文;  
}
```

まず初期条件の状態プログラム文が実行される。

次に更新条件の式が実行され、

2回目のプログラム文実行が行われる。

実行条件が満たされている間、実行が繰返される。

```
// Program3.c   Disp image
```

画像を表示する

```
#include <HU.h>
```

```
void Main(void)
```

```
{  
    char yn, f[100];
```

```
    unsigned char a, b;
```

```
    int i, j, count, img[260][260];
```

```
    FILE *fp;
```

```
    TextWindow(10,10,300,200);
```

```
    Title("Image Display");
```

```
START:
```

Get image data

```
strcpy( f, "c:\jouhou\program3\bone6" );
```

```
printf("\n Display image = \n\n %s", f );
```

```
fp = fopen( f , "rb" );
```

```
for( j=1; j<=256; j++ ){
```

```
    for( i=1; i<=256; i++ ){
```

```
        a = fgetc( fp );    b = fgetc( fp );
```

```
        img[ i ][ j ] = a * 256 + b;
```

```
    }
```

```
}
```

```
fclose(fp);
```

```
//----- Display image -----
```

```
GraphicWindow( 300, 300, BK_BLACK ) ;
```

```
for( j=1; j<=256; j++ ) { for( i=1; i<=256; i++ ) {
```

```
    count = img[ i ][ j ] ;
```

```
        if( count < 0 ) count = 0 ;
```

```
        if( count > 255 ) count = 255 ;
```

```
        SetColor( RGB( count, count, count ) ) ;
```

```
        SetPixel( i, j ) ;
```

```
    } }
```

```
//----- End -----
```

```
printf( "¥n¥n Retry? ( no: n ) " ) ;
```

```
scanf( "%c", &yn ) ; if( yn != 'n' ) goto START ;
```

```
exit(0) ;  
}
```

```
Image Display
Display image =
  c:%jouhou%program3%bone6
Retry? (no:n) |
```



program3 の実行結果

画像ファイル bone6 が表示される。

^{99m}Tc -MDP Bone scintigraphy

胸部背面spot像。 撮像6分間。 画素数 256 x 256


```
char f[100];
```

char は、文字型変数の宣言文。
f[100] は、文字を100個入れられる変数の配列。
配列名は f と宣言している。(どんな名前でもよい。)

C言語では、配列(行列)の大きさを [] ではさんで表現する。

char f[100]; と書くと、文字が100個まで入れられる文字配列が f という名前で用意される。

現在のパソコンは、メモリ量が十分あるので、大きめの配列を用意しておいたほうが賢明。

(配列に入りきらない文字列を配列に入れてしまうと、バグを見つけにくい解決困難なエラーが起きる。)

`unsigned char a, b ;`

`unsigned` とは、マイナスの値をもたないことを意味する。
`signed` とは、マイナスの値をもつことを意味する。

`unsigned char a ;` と宣言された変数 `a` は、
0 ~ 255 までの整数を持つことができる。

`signed char a ;` と宣言された変数 `a` は、
-128 ~ 127 までの整数を持つことができる。

`char a ;` と宣言された変数 `a` は、Visual C++ では
`unsigned char` と同じ扱いになる。(コンパイラに依存性)

従って、この場合は `unsigned` を省略してもよいが、
マイナスの値も取りうるプログラムを作成する場合や
異なるコンパイラ、異なるOS を用いる場合には、
`signed`、`unsigned` を明記するほうが賢明。

char 型整数変数は、256通りの整数を持つことができるので、1バイト変数と呼ばれる。

(1バイト=8ビット)

(8ビットの情報量は、 $2^8=256$)

半角英数文字は全て 0~255 までの数字に対応している (アスキーコード ASCII)。

文字列や画像ファイルデータは、1バイトデータの集合体なので、

文字や画像ファイルを扱う変数として char 型変数がよく用いられる。

```
int img[260][260];
```

int は、4バイトまで入る整数変数の宣言文。
img[260][260] は、画素数が 260 x 260 の画像を
入れられる2次元配列(2次元行列)。
配列名は img と宣言している。(どんな名前でもよい。)

C言語は、多次元配列(行列)の宣言が可能。
3次元なら volume[100][100][100] などと記述する。

現在のパソコンは、メモリ量が十分あるので、
実際に扱う画像サイズより大きめの配列を
用意しておいたほうが賢明。

(配列に入りきらない画像を配列に入れてしまうと、
バグを見つけにくい解決困難なエラーが起きる。)

FILE *fp ; とは、

FILE型変数（データファイルを扱う関数に、使うファイルを示すための変数）*fp を宣言している。

FILE型として宣言された変数を、ファイルを開いたり閉じたりする関数が受取ると、それが普通の変数ではなく、ファイルだとわかるようになっていてる。

ファイルを扱う関数は、FILE型変数自体ではなく、そのポインタ(pointer、FILE型変数が記録されているメモリ内の先頭アドレス)を必要としている。(scanf 関数と同様。)

C言語では、変数自体よりも、その**変数のメモリ内の先頭アドレス(ポインタ)**を使って**演算を行う関数**がたくさんある。

ファイルを開く関数 `fopen()`、ファイルの内容を讀出す関数 `fgetc()`、ファイルを閉じる関数 `fclose()` など**ファイル型変数の先頭アドレス(ポインタ)**を記入する。

普通に宣言した変数でも**&記号**を付けてポインタ変数に変換できるが、いちいちファイル型変数に**&記号**を付けて、ファイル操作関数の中でポインタに変換するのは面倒なので(プログラム内が **&**だらけになる)

はじめからポインタとして宣言しておいた変数のほうが、プログラムを書きやすい。

例えば、`FILE fp;` と宣言すると、

ファイル型変数は `fp` で、その先頭アドレス(ポインタ)は `&fp` となる。

`FILE *fp;` と宣言すると、

ファイル型変数は `*fp` で、その先頭アドレス(ポインタ)は、`fp` となる。

*** (アスタリスク)と & は、逆の働きをする記号(演算子)。**

`fp` という名前でもかまわないが、C言語では、ファイル型変数のポインタは、file pointer であることを明示するために、`fp` とか `fp2` などと宣言するのが普通。

```
strcpy( f, "c:\jouhou\program3\bone6" );
```

**strcpy関数は、文字列を別の配列にコピーする関数。
(string copy)**

任意の文字列をC言語で文字列(文字列リテラル)として扱うには“ ”で挟む。

**c:\jouhou\program3\bone6 の意味は、
Cドライブにあるフォルダjouhouの中にサブフォルダ
program3 があって、その中の画像ファイル bone6
を示している。**

**各自のコンピュータ内のフォルダの場所に対応した
記述に変更して下さい (ただし空白文字や日本語を
含むフォルダ名は避けた方が無難です)。**


```
strcpy( f, "c:¥¥jouhou¥¥program3¥¥bone6" );
```

100文字まで入れられる文字列配列 f[100] に、文字列 “c:¥¥jouhou¥¥program3¥¥bone6” をコピーしている。

strcpy関数が欲しい情報は、文字列自体ではなく文字列が書き込まれているメモリ内の先頭アドレスと新たに書き込む配列 f[100] のメモリ内の先頭アドレス。

配列を宣言した場合、その配列名だけを記述するとその配列の中身が記録されているメモリ内の先頭アドレス(ポインタ)を示す。

任意の文字列を “ ” で挟むと、C言語では、その文字列を記憶できる大きさの文字列配列を自動的に宣言してメモリに記録する（**文字列リテラル**）。

そして、その**配列の名前**が、この場合では“**c:¥¥jouhou¥¥program3¥¥bone6**”となる。

つまり“**c:¥¥jouhou¥¥program3¥¥bone6**”は配列名で、C言語プログラム上では**ポインタ**である。

strcpy関数の文法を簡単に書くと、

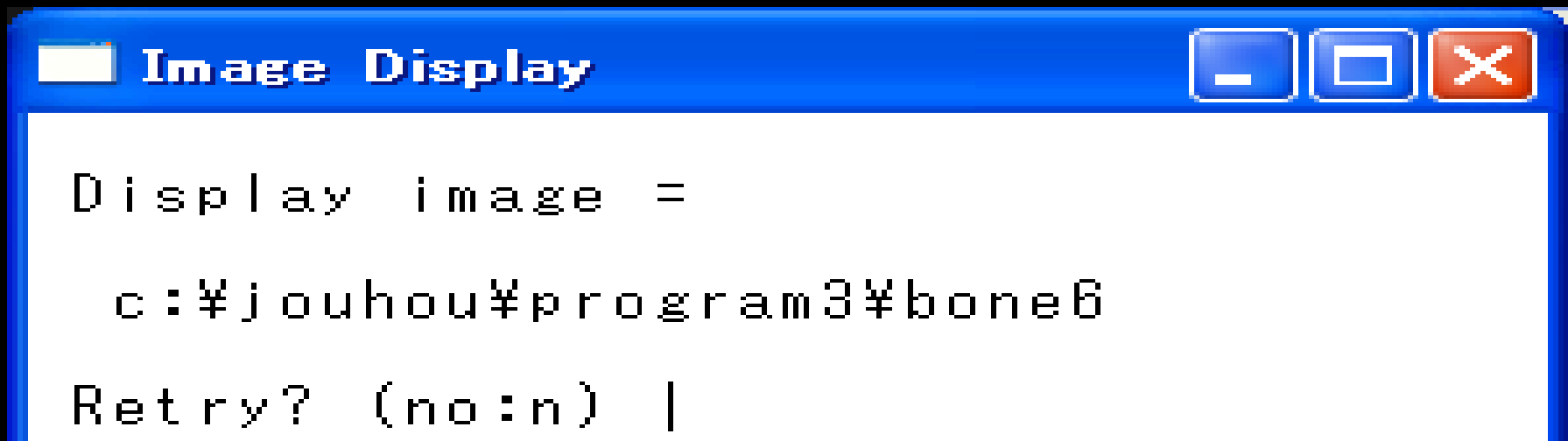
strcpy(コピー配列のポインタ, オリジナル原稿配列のポインタ)

```
printf( “ %n Display image = %n%n %s ”, f );
```

printf関数内で **文字列は %s** と表現する。(string)
(1文字(半角英数文字)の場合は **%c**) (character)

文字列を表示したい場合はカンマ, の次に
表示したい文字列のポインタを記述する。

つまり、この場合は **表示したい文字列をコピーしてある
文字列配列の名前 f** を記述する。



```
Image Display  
Display image =  
c:\jouhou\program3\bone6  
Retry? (no:n) |
```

プログラム実行結果の テキストウィンドウ内の記述を見ると、

c:¥¥jouhou¥¥program3¥¥bone6 ではなく、

c:¥jouhou¥program3¥bone6 と記述されている。

C言語では、フォルダ名を指す ¥ を ¥¥ と記述する。

改行記号の ¥n など、¥ で始まる特別な文字列を **エスケープ文字列**という。

¥¥ も その一つで、¥ を記述してから改めてもう一つ ¥ を記述して、フォルダを指す記号 ¥ と認識される。

```
fp = fopen( f , "rb" );
```

fopen関数 (file open) ファイルを開く関数

fopen関数の文法は、

```
fopen( 開くファイル名を書いた文字列のポインタ , " モード " )
```

モードとは、開いたファイルを読むのか、書き込むのか、その内容はテキスト形式か、バイナリ形式かを明記すること。

テキスト形式：文字列のファイル。

バイナリ形式：文字列以外のファイル。画像など。

fopen関数の主なモード

rb	read binary	バイナリ形式ファイルを読む。
rt	read text	テキスト形式ファイルを読む。
wb	write binary	バイナリ形式ファイルを書く。
wt	write text	テキスト形式ファイルを書く。

テキストファイルの文字化けや、画像データの乱れなどが生じたら、fopen関数のモード設定が正しいか確認する。

fopen関数は、出力値を持つ。

出力するものは、開いたファイルの情報が入ったFILE型変数を書き込んだメモリ内の先頭アドレス(ポインタ)。

fp = fopen(f , “rb”); と記述することによってファイル型変数 *fp (そのポインタが fp) に、開いたファイルの情報(どのファイルを開いたのか、ファイルを読むのか、書くのか、どんな形式なのか)を渡す(=代入する)ことができる。

fclose(fp); (file close) ファイルを閉じる関数
開いたファイルは使い終わったら必ず閉じること。
fclose を忘れると、ファイル破損の危険がある。

```
for( j=1; j<=256; j++ ){  
    for( i=1; i<=256; i++ ){  
        a = fgetc( fp );    b = fgetc( fp );  
        img[ i ][ j ] = a * 256 + b ;  
    }  
}
```

開いた画像ファイル bone6 は、
たて256画素 x よこ256画素、
1画素2バイト(0~256x256 カウント)である。

この画像を、2次元配列 img に読み込む作業を行っている。


```
a = fgetc( fp );
```

fgetc関数 (file get char)

ポインタ fp が示すファイルから
1バイトずつデータを切り出して読み込む。

fgetc関数は出力値をもつ。
出力する値は、読み込んだ1バイトのchar型の値。
char型として宣言した変数 a に出力値を
代入している。

1バイト読み終わったら読み込みを終了するが、
次に同じ関数が起動したときは、次の1バイトを
読むように指令を出している (seek set)。

```
a = fgetc( fp );    b = fgetc( fp );
```

```
img[ i ][ j ] = a * 256 + b ;
```

開いた画像ファイル bone6 は、
1画素2バイト(0~256x256 カウント)である。

まず最初の1バイト目の値が a に入る。
2バイト目の値が b に入る。

この画像の最初(1番左上)の画素値 `img[1][1]` は、
 $a \times 256 + b$ になる。

この作業が、よこ座標 `i`、たて座標 `j` が、1から256
まで for ループしているので、画像が `img` に入る。

画素値 $img[i][j]$ が $a \times 256 + b$ になる理由

ある部位の画素値(カウント)が 0 から 255 までの値であれば、 a は 0 で、 b に 255 以下の値が入っている。
その場合は、画素値 $img[i][j] = 0 \times 256 + b$

ある部位の画素値(カウント)が 256 であれば、 a は 1 で、 b には 0 の値が入っている。
その場合は、画素値 $img[i][j] = 1 \times 256 + 0$

ある部位の画素値(カウント)が 515 であれば、 a は 2 で、 b には 3 の値が入っている。
その場合は、画素値 $img[i][j] = 2 \times 256 + 3$

```
GraphicWindow( 300, 300, BK_BLACK );
```

グラフィックウィンドウを作成する関数。
この1行だけで画像を描画するウィンドウが作れる。

HU.h が用意した関数の中で 最もありがたい関数。

カッコ内のはじめの 300 は、グラフィック画面の横幅、
次の 300 は、縦幅 を示す(単位はピクセル)。
次の BK_BLACK はグラフィック画面の背景色を設定。

BK_BLACK	黒背景
BK_WHITE	白背景
BK_GRAY	灰背景

```
SetColor( RGB( count, count, count ) );
```

```
SetPixel( i, j );
```

グラフィック画面に画像を表示させる作業は、
1画素ずつ画素の色を変えて点描する作業になる。

パソコン画面の各画素の色は、
赤(R)が0～255、緑(G)が0～255、青(B)が0～255
の、各1バイトずつの数値で決まる。
(数値が小さいほど暗い色。)

モノクロ(無彩色)の画像を表示する場合は、
R, G, B の値を同じにする(彩度が0になる)。

SetColor() 関数

描画する点(画素)の色を決める関数。
カッコの中には COLORREF型変数という、
Microsoftが定義した妙な変数を入れるが、
実際には RGB関数を使って簡単に扱う。

RGB(Red , Green , Blue)関数

描画する画素色の、赤(R)、緑(G)、青(B)の値を
カッコ内に順番にカンマで区切って入れる。
出力は、COLORREF型変数になるので
SetColor() 関数と組み合わせて使うと便利。
モノクロ(無彩色)の画像を表示する場合は、
R, G, B の値を同じにする。

SetPixel(i, j);

SetColor(RGB()) 関数で色が決定した画素を描く関数。

カッコ内に描画する座標(よこ、たての順)を入れる。

グラフィック画面の座標の、**原点(0, 0)は左上。**

横の値は左縁が 0 で、大きくなるほど右に移る。

縦の値は上縁が 0 で、大きくなるほど下に移る。

```
for( j=1; j<=256; j++ ){ for( i=1; i<=256; i++ ){  
    count = img[ i ][ j ] ;  
    if( count < 0 ) count = 0 ;  
    if( count > 255 ) count = 255 ;  
    SetColor( RGB( count, count, count ) ) ;  
    SetPixel( i, j ) ;  
} }
```

img[i][j]に入っている画像を1画素ずつ変数 count に代入して、その値が負ならば 0 に、255 より大きければ 255 に制限している。(RGB関数に 0から255の範囲外の数を入れるとエラーになるため。)

変数 i(画像の横座標), j(縦座標)が各々 1 から 256 までループするので、2次元の点分布像 (= 画像)が表示される。


```
// Program3b.c    Disp image2
```

```
#include <HU.h>
```

```
void Main(void)  
{
```

```
    char a, b, yn, f[100] ;
```

```
    int i, j, count, maxcount, img[260][260] ;
```

```
    FILE *fp ;
```

```
    TextWindow(10,10,300,200) ;
```

```
    Title("Image Display") ;
```

```
START:    printf( "¥n Select Display Image ¥n" ) ;
```

program3.c の

改良版

```
//----- Get image data -----
```

```
GetFileName( f , 0 ) ;
```

```
printf( “\n Display image = \n\n %s ”, f ) ;
```

```
fp=fopen(f,“rb”) ;
```

```
for( j = 1; j <= 256; j++ ) { for( i = 1; i <=256; i++ ) {
```

```
    a = fgetc( fp );  b = fgetc( fp ) ;
```

```
    img[ i ][ j ] = a * 256 + b ;
```

```
}}
```

```
fclose( fp ) ;
```

//----- Calculate Max count -----

```
maxcount = img[ 1 ][ 1 ] ;
```

```
// とりあえず初期値を適当に決めておく。
```

```
// ( 最初の画素値 img[ 1 ][ 1 ] を代入しておく。 )
```

```
for( j = 1; j <= 256; j++ ) { for( i = 1; i <= 256; i++ ) {
```

```
    if ( img[ i ][ j ] > maxcount ) maxcount = img[ i ][ j ] ;
```

```
}}
```

```
printf( “ ¥n¥n max count = %d ”, maxcount ) ;
```

//----- Display image -----

```
GraphicWindow( 260, 260, BK_BLACK );
```

```
for( j = 1; j <= 256; j++ ) { for( i = 1; i <= 256; i++ ) {
```

```
    count = img[ i ][ j ] ;
```

```
    count *= 255 / maxcount ;
```

```
    SetColor( RGB( count, count, count ) ) ;
```

```
    SetPixel( i, j ) ;
```

```
}}}
```

ファイル選択ダイアログが表示される。
program3フォルダ内のbone1 ~ bone6の
どれかを選択。

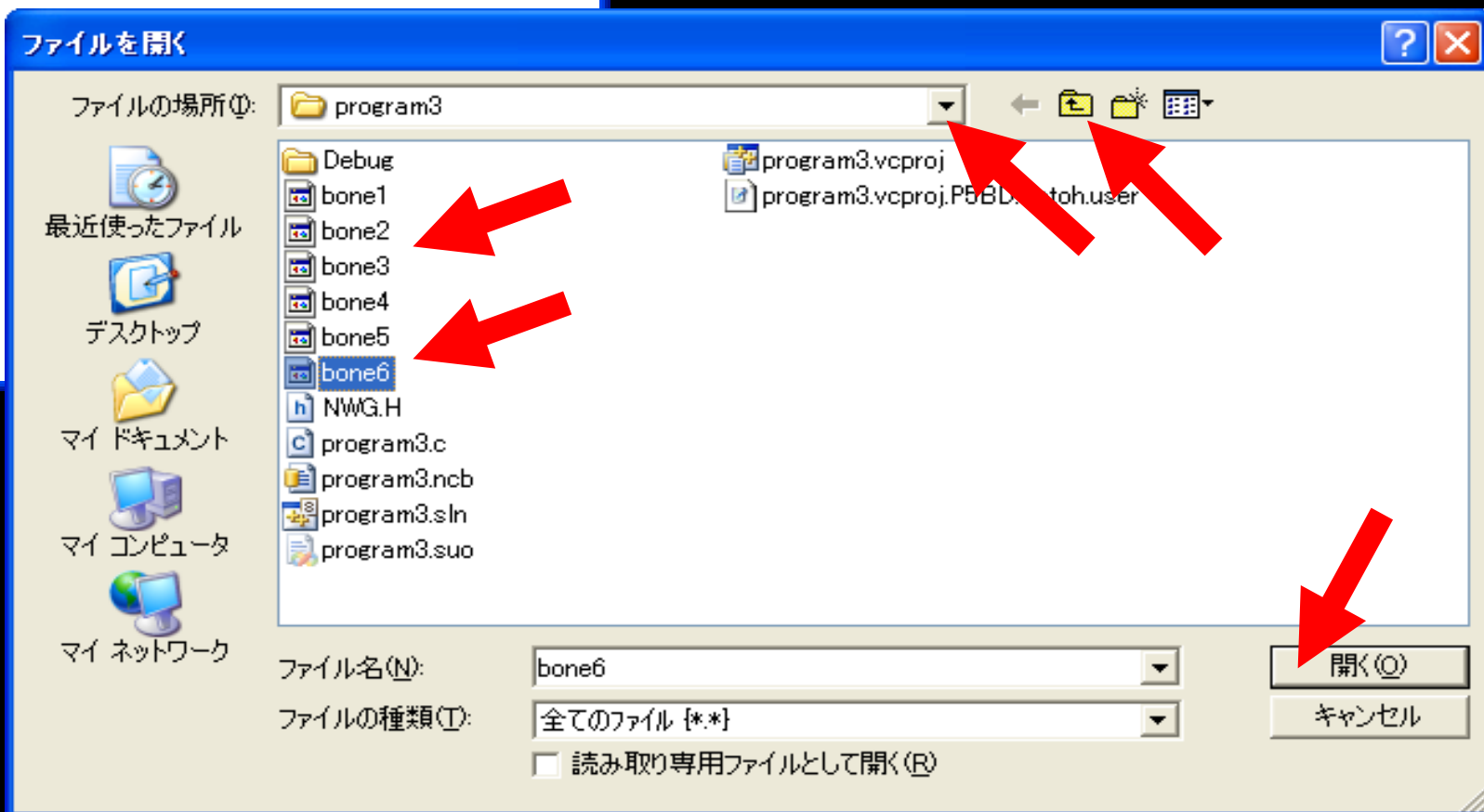


Image Display

Select Display Image

Display image =

C:\jouhou\program3\bone6

max count = 109

Retry? (no:n)

Quantified PET Image



画像内カウントの最大値が表示される。
最大値カウントが白く表示されるように
表示ウィンドウ幅が自動調整されて
画像がきれい描画される。

```
char a, b ;
```

Visual C++ では、char と unsigned char が同じであることを確認できる。

```
int maxcount ;
```

画像内の最大画素値を計算するための整数変数宣言を追加。

```
START: printf( "¥n Select Display Image ¥n" );
```

ラベル(START:)と同じ行にプログラム文を書いてもよい。

GetFileName(f , 0) ;

GetFileName 関数。

ファイル選択ダイアログを表示する関数。

HU.h が用意した大変便利な関数。

カッコ内の最初に、選択したファイル名を入れる文字列配列の名前(ポインタ)を記述し、カンマ,をはさんで、次に、モードを記述する。

2種類のモードがある。

0 : ファイルを読む

1 : ファイルを書く(保存する) 書込み用にも使える。


```
maxcount = img[ 1 ][ 1 ];  
for( j = 1; j <= 256; j++ ) { for( i = 1; i <= 256; i++ ) {  
    if ( img[ i ][ j ] > maxcount ) maxcount = img[ i ][ j ];  
}}
```

最大値を求める常套手段。

forループが終了すると変数 maxcount に最大値が入っている。

始めに maxcount を最初の画素値に設定しておく。

次々と画像内の画素カウントと maxcount を比較し、画素カウントのほうが大きい場合は、その画素カウントを 変数 maxcount に代入する。

```
count = img[ i ][ j ] ;
```

```
count *= 255 / maxcount ;
```

```
SetColor( RGB( count, count, count ) ) ;
```

```
SetPixel( i, j ) ;
```

count *= 255 / maxcount ; は、
count = count * 255 / maxcount ; と同じ文。

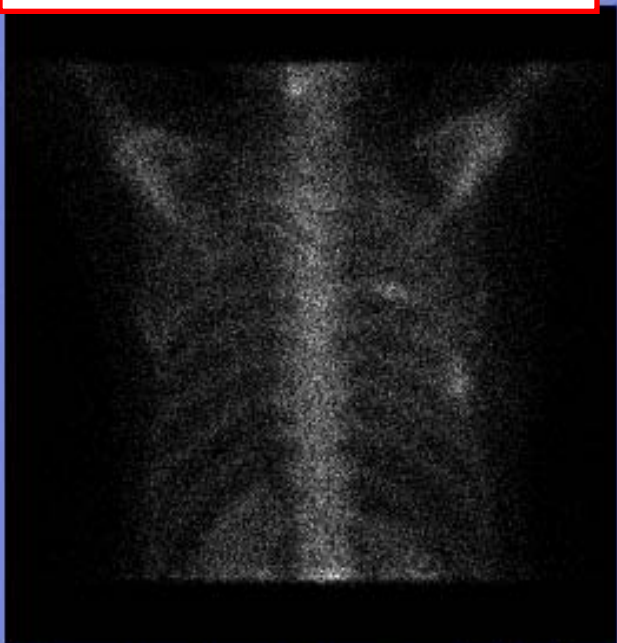
255 / maxcount は、画素最大値を 255 に変換する
比例定数。count の値を **255 / maxcount** 倍している。

この操作によってパソコン画面の真っ黒から真っ白
の色調(明度)を全部使って画像を表示できるので、
コントラストの良い画像が描画できる。

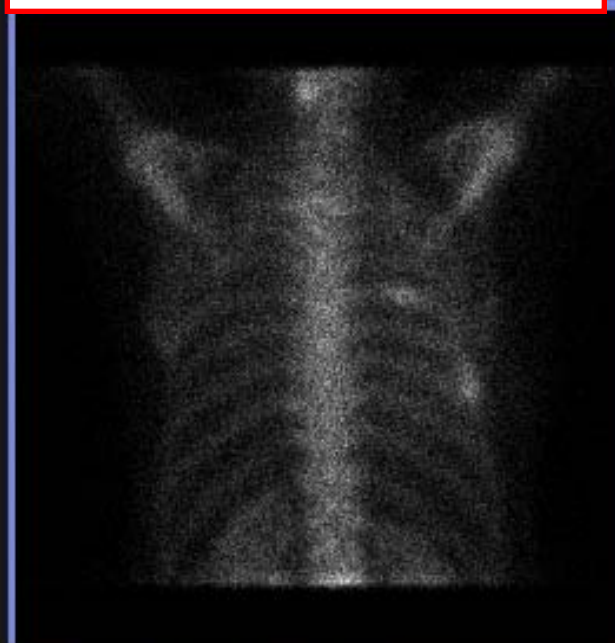
フォルダ program3b内の フォルダDebug内にある program3b.exe をダブルクリックする方法で起動すると program3b.exe は 多重起動できるので、 bone1 から bone6 までの画像を同時に見ることができる。



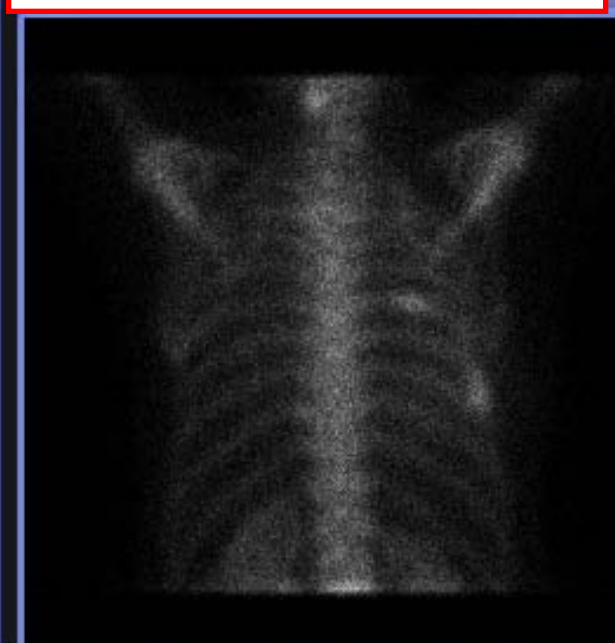
Bone1 撮像1分 max 28



Bone2 撮像2分 max 46



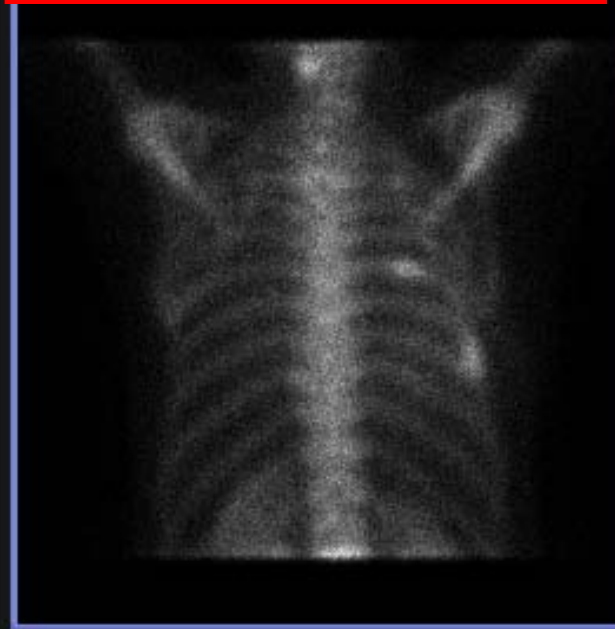
Bone3 撮像3分 max 66



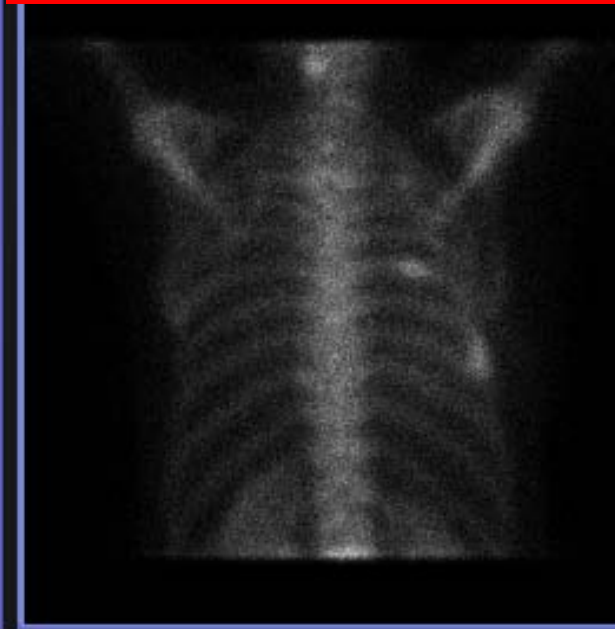
Bone4 撮像4分 max 84



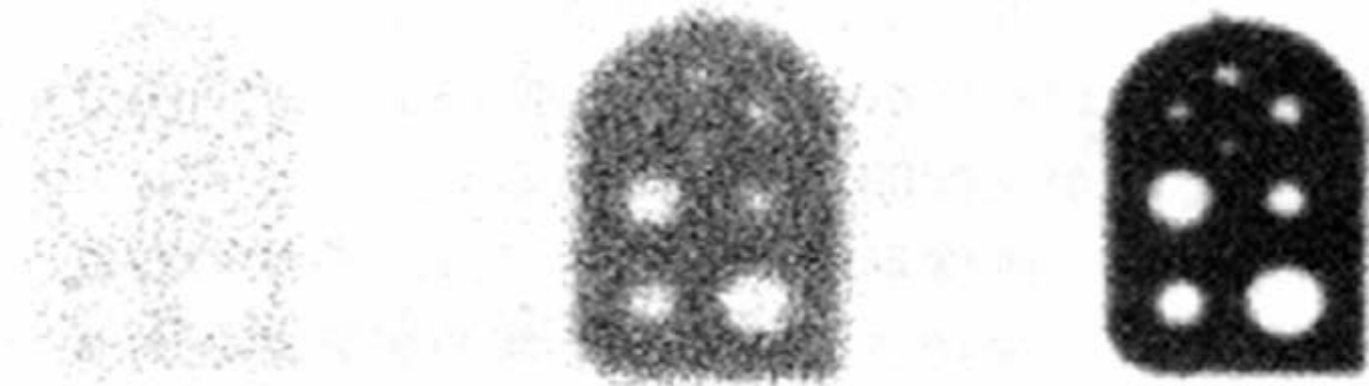
Bone5 撮像5分 max 95



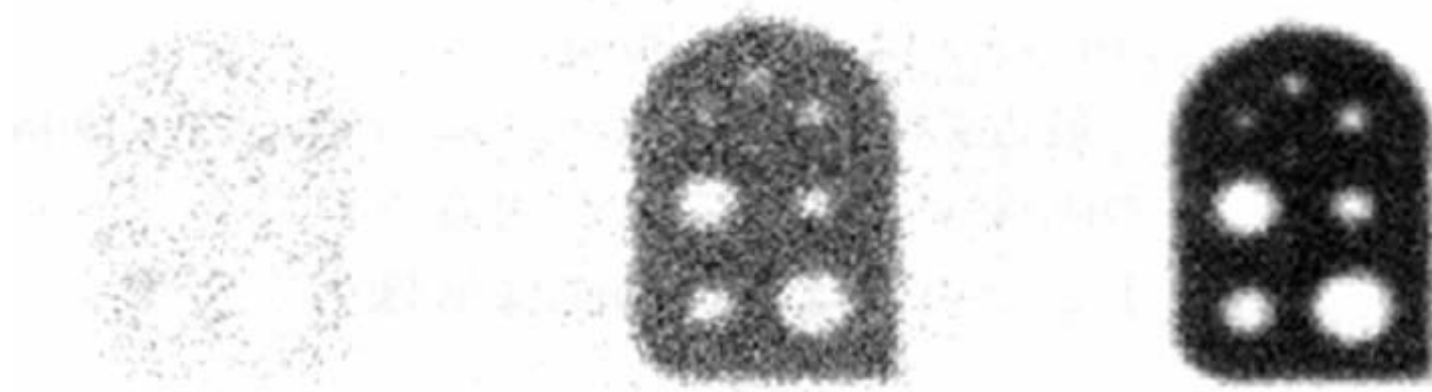
Bone6 撮像6分 max 109



LEGP
コリメータ



LEHR
コリメータ



計数密度
[counts/cm²]

20

200

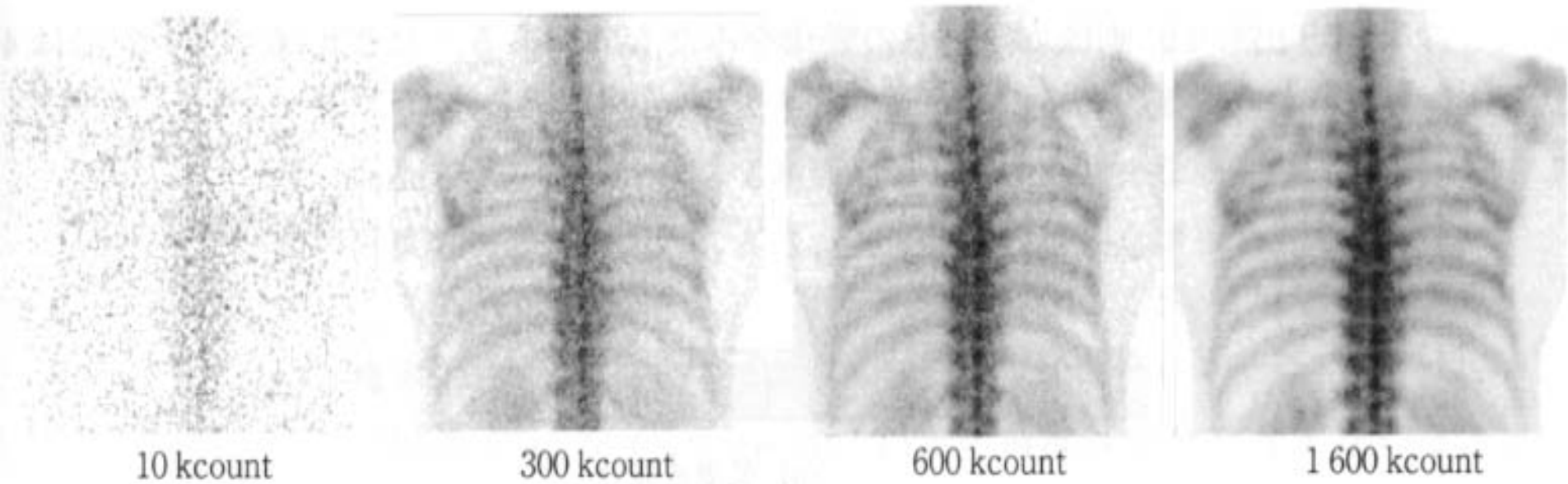
2 000

図 4・5 シンチカメライメージングにおける画像を形成するガンマ光子数（計数密度）と画質

コリメータ：低エネルギー汎用（LEGP）および低エネルギー高分解能（LEHR）、肝スライスファントムに^{99m}Tcを封入
欠損の直径：8, 10, 12, 16, 20, 25, 32, 40 mm

計数密度

1 cm² あたりの収集カウント数



UFOV 32x32cm (約1000 cm²) の骨シンチグラフィ。

計数密度 (counts/cm²) は 左図から 10, 300, 600, 1600。

1000 counts / cm² 以下のシンチグラフィは、

量子ノイズが目立つ。

課題3

bone1 ~ bone6 の bone scintigraphy は、
43.4 cm x 43.4 cm UFOV (有効撮像視野)の像。

program3b.c を改良して

bone1 ~ bone6 の各画像における 計数密度
(counts/cm²)を求めるプログラムを作り、

プログラムコードと計算結果をメールに添付して

ホームページの課題提出ボタンから今週中に送って下さい。