第 III 部

画像処理プログラミング

第 1 章

画像処理の基本

この章では、画像処理の**プログラミング**において、基本となる用語や操作について説明します。特に注意しない限り、ここでは、二次元配列で記述された画像を対象とします。

1.1 画像

画像を記述するにはいろいろなデータ表現が考えられます。ここで扱う画像というのは、連続的な二次元平面に対して**標本化** (sampling) と**量子化** (quantization) を行ない、そのデータを二次元配列にした**ディジタル画像**として扱います (図 1.1)。

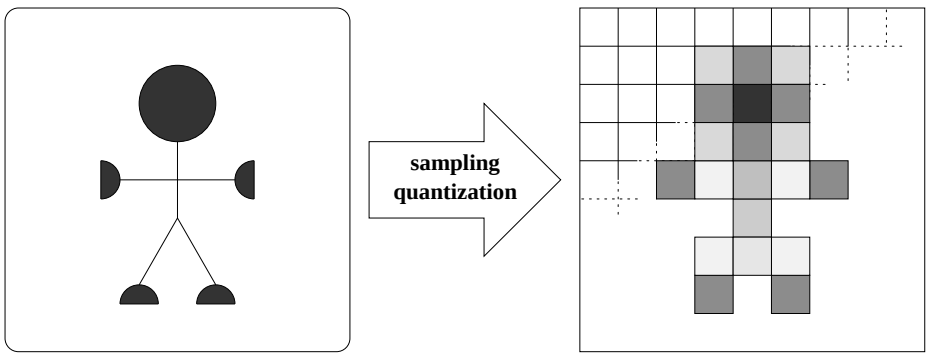


図 1.1: 画像の標本化と量子化

これは、コンピュータ上での基本的な画像の表現方法です。この二次元配列で表された画像を表現するのに必要な属性は、 x, y **画素数**、**画素型**があれば十分で

す。しかし、これだけでは、画像が (物理的な量として) 何を表しているのか分からないので**画像型**という属性を追加します。

以下に、画素型と画像型について説明します。画像処理においては、この二つが重要な画像の属性となります。また、この二つは非常に密接な関係を持っています。

1.1.1 画素型

画素型とは、その画素がどれだけのビット数を持っているかとか、その表現は整数であるか実数であるか、次元数はいくつかなどといった情報です (表 1.1)。

表 1.1: 画素型の情報

ビット数	どれだけの精度／範囲か
表現方法	整数／実数など
次元数	1 次元／ 3 次元／ 4 次元など、 n 次元

この画素型は論理的なデータの表現方法を規定するのです。この画素が何の意味 (物理的な意味) を持つかは画像型で表されます。通常、この画素型はプログラミング言語では一つの**型**として与えられます。 C 言語であるならば、 **unsigned char**, **float**, **unsigned char [3]** などといったものが考えられます。表 1.2に、いろいろな例を示しておきます。

表 1.2: 画素型のいろいろな例

型名	nbit[1]	ebit[2]	表現方法	次元	範囲[3]
unsigned char	8	1	整数	1	0, 1
unsigned char	8	8	整数	1	0 〜 255
unsigned char [3]	24	8	整数	3	0 〜 255
short [2]	32	11	整数	2	−1024 〜 1023
float	32	32	実数	1	0.0 〜 1.0
float [2]	64	32	複素数	2	−Inf 〜 +Inf

このように、画素型は論理的なことしか規定しないので、 (無限に) 自由に定義することができます。もちろん、画素型が構造を持って表現されるといった例を

簡単に作ることもできます。

1.1.2 画像型

画像型は、その画像の画素値が何を表しているのかといった属性です。つまり、画像型は、「二値画像」、「濃淡画像」、「カラー画像」といったものです。画像型は、画素の値の意味づけをします。

表 1.3に、画素型と画像型の関係を示しておきます。画像型は、一つの画素型によって拘束されるわけではありません。その画像が何であるかは解釈の問題です。

表 1.3: 画像型と画素型の関係

画像型	画素型
二値画像 (binary image)	**unsigned char** (ebit=1)
濃淡画像 (gray-scale image)	**unsigend char**, **short**, **float** など
特徴画像 (feature image)	**short**, **float** など
カラー画像 (RGB-color image)	**unsigned char [3]**, **float [3]** など
ラベル画像 (label image)	**unsigned short**, **long** など
ベクトル画像 (vector image)	**short [2]]**, **float [2]]** など
その他	**char**, **float**, **float [2]** など

1.1.3 画素型が多次元の画像の表現方法

画素型の次元が多次元のとき、プログラミングやファイルの保存などで画像の表現方法が問題となります。これらの表現方法には、**画素分離型**と**画素結合型**の二種類あります。

画素分離型とは、図 1.2の左の図のように画素型の各次元を一つの画像として扱い、それらの画像が集まった形で表現します。**画素結合型**とは、図 1.2の右の図のように画素の値を連続して一つの画像に格納します。

[1]nbit は、画素型の全体に必要なビット数のことです。

[2]ebit は、画素型の一次元あたりの有効ビット数のことです。

[3]範囲は一次元あたりの有効な範囲であり、これは使う人によって決めることができます。

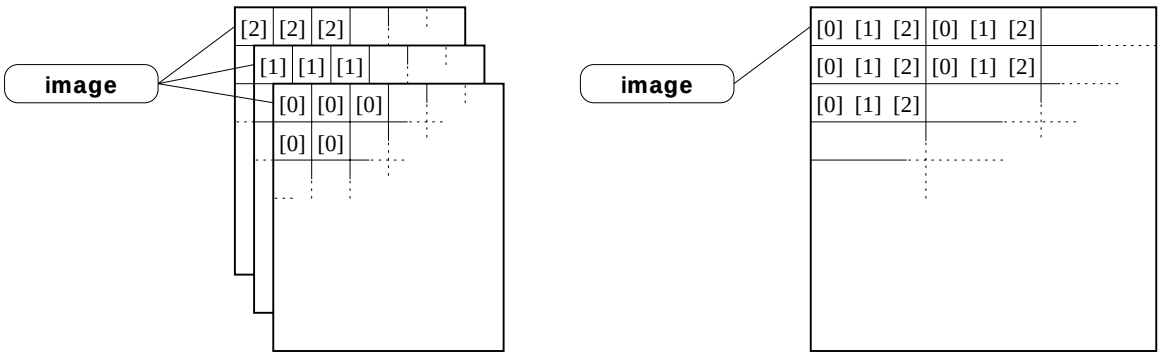


図 1.2: 画素分離型 (左) と画素結合型 (右)

1.2 画像ファイル

画像をファイルに格納したものを**画像ファイル**といいます。画像をファイルに格納する形式のことを**画像ファイル・フォーマット**といいます。現在、この形式には様々なものがあります。

これらの形式を完全には知っておく必要はありませんが、この形式は画素型と密接な関係にあり、この画像ファイル・フォーマットによって、画素型が制限されている場合があるので、このようなことは知っておく必要があります。

1.2.1 C2D フォーマット

C2D フォーマットは、CIL の二次元配列格納用の標準フォーマットです。このフォーマットは**任意の型**の二次元配列を格納することができます。ヘッダは以下のようになっています。

```
C2D␣<型名>␣<型バイト数>␣<x サイズ>␣<y サイズ>␣<コメント>\n
<バイナリデータ>
```

形式 1.1: C2D フォーマットのヘッダ

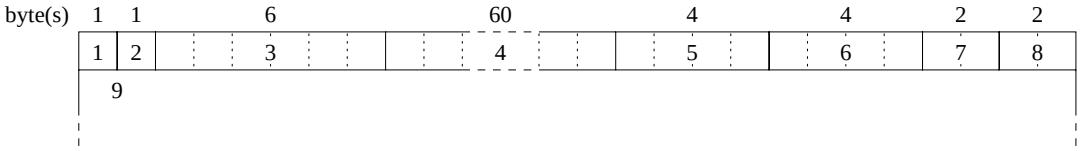
最初に “C2D” というマジックワードがきて、後はスペースで区切られたパラメータが並び、そして改行までコメントが入ります。このように、C2D のヘッダは一行で構成されています。

C2D において、画素型は型名と型バイト数で決まります。画像型は、コメント

やファイル名で利用者が表すことになります。また、型名は CIL に登録されていないものでも許されます。任意の型に対して読み込み書き込みが可能なのです。なぜならば、型バイト数があれば基本的にはファイルを読み込むことができるからです。もし、ある型のバイト数が整数倍でなかった場合は、特別な処理を行ないます。

1.2.2 J4 フォーマット

J4 フォーマットは、阿部研究室で作成されたフォーマットです。このフォーマットは、 SIDBA[1]のデータ形式を反映しています。図 1.3にヘッダの説明を示します。図に示されるように、画素型は物理ビットと有効ビットによって表されます。また、このフォーマットは基本的に画素分離型です。



番号	型／値	説明
1	H, V	スキャン方向、 H(水平方向), V(垂直方向)
2	R, G, B M, L	画像種類 (画像型)、 R (red), G (green), B (blue), M (mono tone), L (label)
3		拡張用スペース
4	文字列	コメント (左詰め)
5	整数文字列	サンプル数 (xsize)
6	整数文字列	スキャン数 (ysize)
7	整数文字列 (1, 8, 16)	nbit, 物理ビット数、 1 画素あたりのビット数
8	整数文字列 $\leq$ nbit	ebit, 有効ビット数
9	バイナリ	画像データ

図 1.3: J4 フォーマットの説明

[1]Standard Image Data Base の略、東京大学で作成された画像データベース。

1.2.3 その他のフォーマット

上記の 2 つは、阿部研究室で作成されたローカルなフォーマットです。現在、世の中に出回っているフォーマットが数多く存在しています。表 1.4にいくつか挙げておきます。 pbm, pgm ppm は、まとめて pnm フォーマットと呼ばれています。このフォーマットは、各種の画像フォーマットを変換するための中間ファイルのために存在しているものです。

表 1.4: 画像ファイル・フォーマットの例

pbm, pgm, ppm	pbmplus の 3 つのフォーマット 二値 (1bit)、濃淡 (8bit)、カラー (24bit)
tiff	Tag Image File Format.
gif	8 ビットカラーマップ方式
xwd	X Window dump
eps (part of image)	EPS の画像のデータのみ 二値 (1bit)、濃淡 (8bit)、カラー (24bit)

1.3 画像のデータの操作

画像が二次元配列で格納されていることは、節 1.1で述べました。この節では、画像をどのように扱うか説明します。

画像処理では、画像から特徴を計算する操作をよく行ないます。特徴を計算する操作にはいろいろありますが、画素の値を使って特徴を計算する方法を述べます。

1.3.1 一次元配列走査／ラスタスキャン

ラスタスキャンは、画像を一次元配列と見なして走査を行ないます。画像平面上では、図 1.4のように、左上から右下へ画素を順番に走査していきます。

一般に、この走査は、 1 画素で変換が完了する場合 (画素値の反転、単純線形変換、単純閾値処理など) や、画像全体の画素から何か計算する場合 (ヒストグラム、平均、最大値、最小値など) に使用されます。つまり、隣接した画素とは独立した操作を行なうときに有効な走査です。

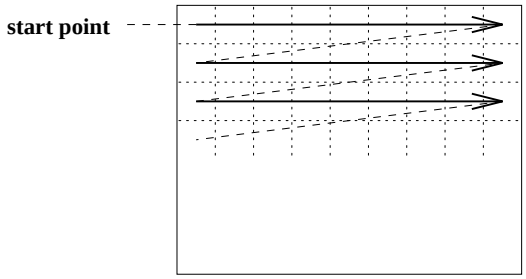


図 1.4: 画像のラスタスキャン

ラスタスキャンのスケルトン・プログラムを譜 1.2に示します。実際には、変数もいくつか増えるでしょうし、 7行目でいろいろな操作をすることになります。この操作の具体的な例は、節 1.4で述べます。

```
1: {
2:   TYPE *ptr = ( TYPE * )Image.raster( input );
3:   int n = Image.area( input );
4:   int i;
5:
6:   for ( i = 0; i < n; i++, ptr++ )
7:     do something ( *ptr );
8: }
```

譜 1.2: ラスタスキャン基本ルーチン

1.3.2 二次元配列走査

二次元配列走査は、局所演算に代表されるような、隣接する画素を使って操作を行なうときに有効な走査です。通常は、図 1.5の左図のようにラスタスキャンと走査の順番は同じように左上から右下へ走査します。しかし、右図のような走査も二次元配列走査としてあります。つまり、この走査は、 x, y を独立に走査することによって実現されるのです。

二次元配列走査の左側の図の順序で走査するスケルトン・プログラムを譜 1.3に示します。しかし、いろいろな状況に応じて細かいところが違ってくると思います。出力画像用の配列へのポインタが必要になってくるかも知れません。さらに、空間フィルタリング処理などでは、内側に局所範囲の x, y のループができると思

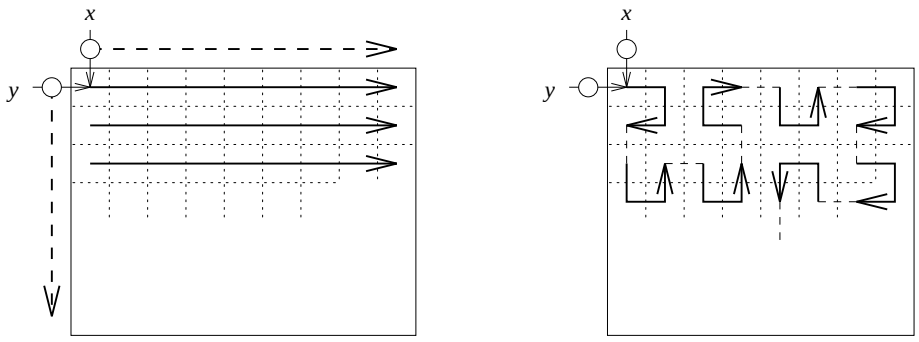


図 1.5: 画像の二次元配列走査

います。空間フィルタリング処理に関しては、次の節で述べます。

```
 1: {
 2:   int x, y;
 3:   TYPE **data = ( TYPE ** )Image.data( input );
 4:   int xsize = Image.xsize( input );
 5:   int ysize = Image.ysize( input );
 6:
 7:   for ( y = 0; y < ysize; y++ )
 8:     for ( x = 0; x < xsize; x++ )
 9:       do something ( data[ y ][ x ] );
10: }
```

譜 1.3: 二次元配列走査の基本ルーチン

二次元配列走査の処理の例は、ソーベルフィルタ、ラプラシアンフィルタなどの線形フィルタ、メディアンフィルタ、多数決フィルタなどの非線形フィルタ、ラベリング、境界線追跡など数多く挙げられます。具体的な例は、節 1.4で詳しく述べます。

1.3.3 空間フィルタリング処理

空間フィルタリング処理は局所演算の一種です。二次元配列で表されたフィルタと画像の局所範囲で、畳み込み積分をした値を出力とする操作です。図 1.6に、画像とフィルタの畳み込み積分の説明を示します。図のように、(x, y) を中心に、フィルタと画像の対応する各値を掛け合わせ、それらすべてを足し合わせた値を出力とします。このような操作をすべての画素について施す処理を、**空間フィルタリング処理**といいます。

フィルタの大きさの関係で、画像の外回りが正確に計算できない場合がありま

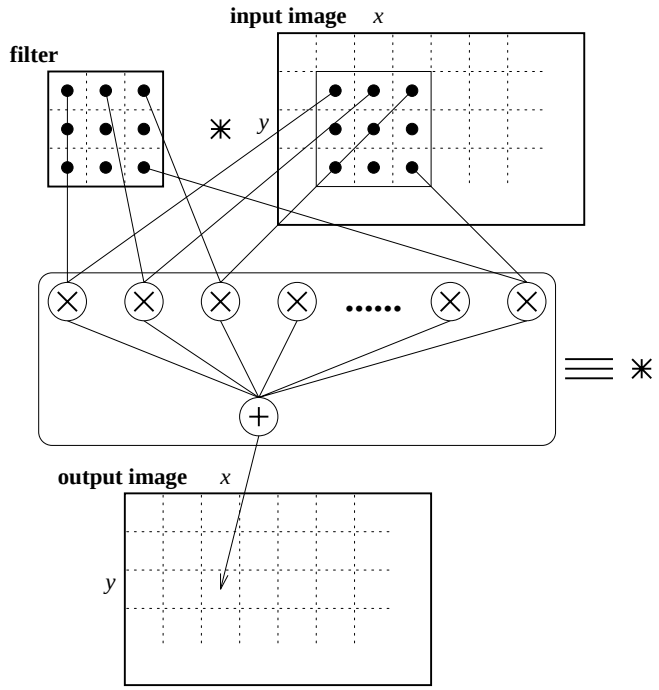


図 1.6: 画像とフィルタの畳み込み積分

す。この外回りの部分をどう処理するか問題になります。解決方法としては、以下のようなものが考えられます。

- フィルタが画像の外に、はみ出さないように内側だけにフィルタを施す
- サイズのチェックを入れ、はみ出した部分は計算しない
- フィルタのはみ出すところは、一番近い画素値を用いる
- フィルタのはみ出すところをあらかじめ補間する
- 外側だけ特別なフィルタを施す

これらは、「フィルタの性質」、「フィルタの後の処理」、「画像の外回りが重要かどうか」といったことに依存します。**どのような目的**で**どのようなフィルタ**を使うのかが重要になってきます。

1.4 操作例

この節では、ラスタスキャン走査と二次元配列走査の例を紹介します。以下に述べるプログラムは完全ではなく、説明する程度の記述にとどめておきます。エラーチェックもしていませんし、実行速度も洗練していません。また、基本的に CIL 準拠の記述になっているので、実際に動作するプログラムに変更するのは難しくないでしょう。

1.4.1 ヒストグラム

ヒストグラムは、画像の画素値の頻度を計算したものです。図 1.7に例を示します。左が原画像で右がヒストグラムです。処理は単純で、画像を一回走査して、各画素値の頻度に 1 を足すだけです。譜 1.4は、濃淡画像のヒストグラムを計算する関数です。

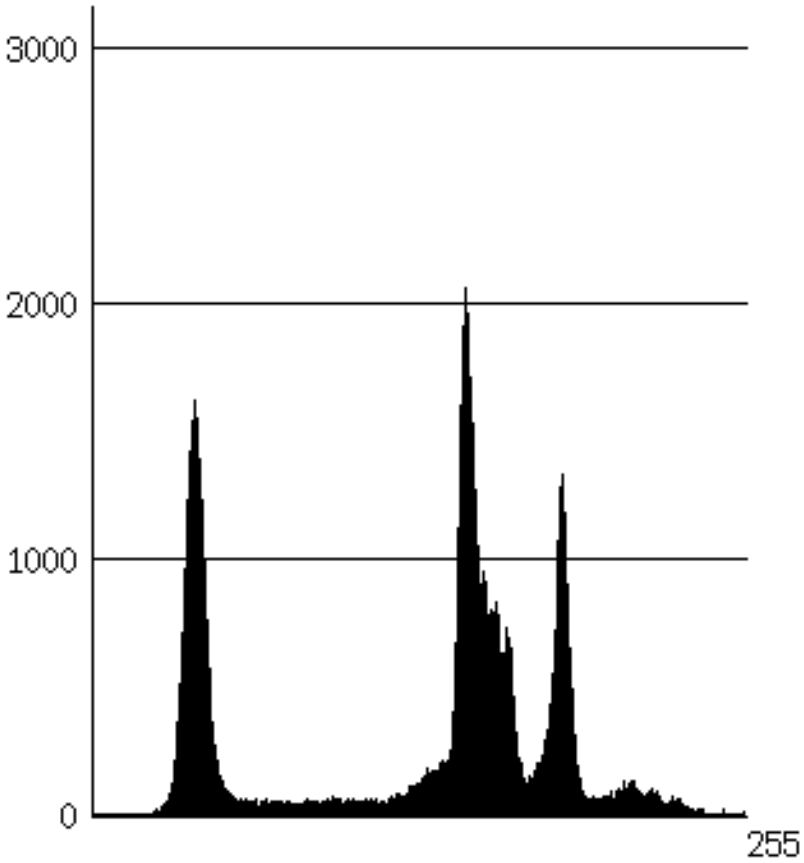
```
 1: long *get_histogram
 2:    _P1 (( image, input ))
 3: {
 4:    int i, n;
 5:    uchar *ptr;
 6:    long *freq;
 7:
 8:    ptr = ( uchar * )Image.raster( input );
 9:    n = Image.area( input );
10:    freq = typenew1( 256, long );
11:
12:    for ( i = 0; i < n; i++, ptr++ )
13:       freq[ *ptr ]++;
14:
15:    return freq;
16: }
```

譜 1.4: 濃淡画像のヒストグラム

10行目: 頻度を入れる一次元配列を typenew1 で新しく確保しています。

13行目: 画素値に対応する頻度を加算しています。

(a) 入力画像



(b) ヒストグラム (x 軸: 濃度値, y 軸: 頻度)

図 1.7: ヒストグラムの例

1.4.2 単純二値化

単純二値化は、濃淡画像をあるしきい値で 0 と 1 に分ける閾値処理です。図 1.8に例を示します。左が原画像で右が二値化した画像です。譜 1.5は、濃淡画像に対して二値化処理を行なう関数です。

```
 1: void simple_thresholding
 2:   _P3 (( image, output    ),
 3:        ( image, input     ),
 4:        ( uchar, threshold ))
 5: {
 6:   int i, n;
 7:   uchar *i_ptr;
 8:   bit1  *o_ptr;
 9:   long xsize, ysize;
10:
11:   xsize = Image.xsize( input );
12:   ysize = Image.ysize( input );
13:   Image.make( output, Bit1, xsize, ysize );
14:
15:   i_ptr = ( uchar * )Image.raster( input );
16:   o_ptr = ( bit1 * )Image.raster( output );
17:   n = Image.area( input );
18:
19:   for ( i = 0; i < n; i++, i_ptr++, o_ptr++ )
20:     *o_ptr = ( *i_ptr >= threshold ) ? 0 : 1;
21: }
```

譜 1.5: 濃淡画像に対する単純二値化処理

13行目: Bit1 型を画素型とし、入力画像の xsize, ysize と同じ大きさで出力画像のデータ領域を確保します。

20行目: 入力画素値が閾値 threshold 以上ならば 0 (背景) とし、小さければ 1 (図形) とし、出力画素値をセットします。

(a) 入力画像

(b) 二値画像 (閾値 = 166)

図 1.8: 単純二値化の例

1.4.3 濃度線形変換

濃度線形変換は、濃度を線形な式で変換する処理です。ここで例として取り上げる濃度線形変換は、濃度値の広がりを最大にする処理です。図 1.9に例を示します。左が原画像で右が線形変換を行なった画像です。譜 1.6は、濃淡画像に対して、図 1.9のような濃度線形変換の処理をする関数です。

```
 1: void simple_liner_scale_trans
 2:   _P2 (( image, output ),
 3:        ( image, input  ))
 4: {
 5:   int i, n, xsize, ysize;
 6:   uchar *i_ptr, *o_ptr;
 7:   uchar max, min;
 8:
 9:   xsize = Image.xsize( input );
10:   ysize = Image.ysize( input );
11:   Image.make( output, UChar, xsize, ysize );
12:
13:   i_ptr = ( uchar * )Image.raster( input );
14:   o_ptr = ( uchar * )Image.raster( output );
15:   n = Image.area( input );
16:
17:   max = 0; min = 255;
18:   for ( i = 0; i < n; i++ )
19:     if ( max < i_ptr[ i ] ) max = i_ptr[ i ]; else
20:     if ( min > i_ptr[ i ] ) min = i_ptr[ i ];
21:
22:   for ( i = 0; i < n; i++, i_ptr++, o_ptr++ )
23:     *o_ptr = 255 * ( *i_ptr - min ) / ( max - min
   );
24: }
```

譜 1.6: 濃淡画像に線形濃度変換の例

17行目: この 4 行で最大値と最小値を求めます。

23行目: 入力画素値の線形変換を行ないます。

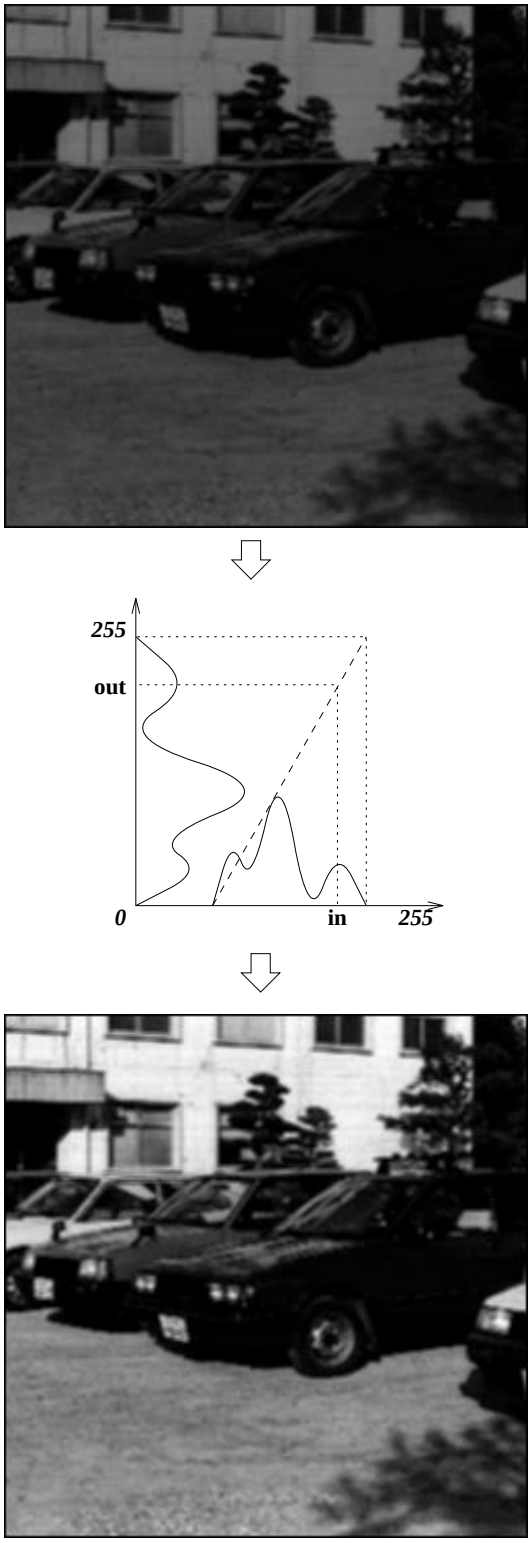


図 1.9: 濃度線形変換処理の例

1.4.4 線形フィルタリング

線形フィルタの例として、4 近傍ラプラシアンフィルタを紹介します。4 近傍ラプラシアンフィルタの例を図 1.10に示します。左が原画像、図の中央がフィルタの内容です。このフィルタと原画像を畳み込み積分したものが右の画像です。灰色の部分が 0 の値で、黒は負の値、白は正の値です。譜 1.7は、4 近傍ラプラシアンフィルタリング処理をする関数です。画像の外回りはフィルタを施していません。基本的に filter の内容を変更するだけで、3×3 のフィルタ処理ができます。さらにフィルタを関数のパラメータとして受けとるようにすれば、汎用的な関数が実現できます。

```
 1: void laplacian4_filtering
 2:   _P2 (( image, output ),
 3:        ( image, input  ))
 4: {
 5:   int x, y, i, j, xsize, ysize;
 6:   uchar **i_data;
 7:   short **o_data;
 8:   int filter[3][3] = {{  0, -1, 0  },
 9:                       { -1,  4, -1 },
10:                       {  0, -1, 0  }}
11:
12:   xsize = Image.xsize( input );
13:   ysize = Image.ysize( input );
14:   Image.make( output, Short, xsize, ysize );
15:   i_data = ( uchar ** )Image.data( input );
16:   o_data = ( short ** )Image.data( output );
17:
18:   for ( y = 1; y < ysize-1; y++ )
19:     for ( x = 1; x < xsize-1; x++ )
20:       for ( j = -1; j <= 1; j++ )
21:         for ( i = -1; i <= 1; i++ )
22:           o_data[ y ][ x ] +=
23:             filter[j+1][i+1] * i_data[y+j][x+i];
24: }
```

譜 1.7: 4 近傍ラプラシアンフィルタ処理

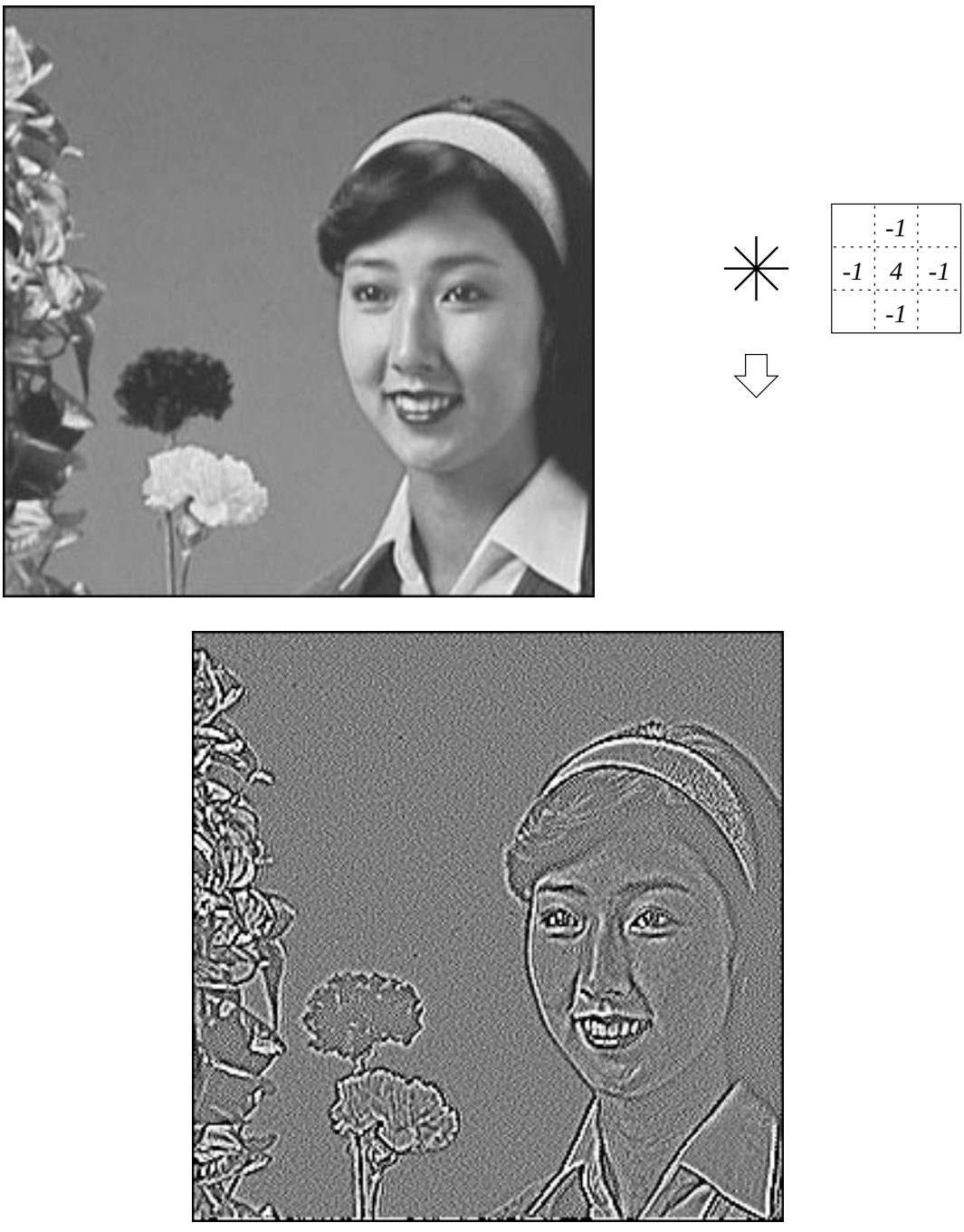


図 1.10: 4 近傍ラプラシアンフィルタ処理の例

1.4.5 非線形フィルタリング

非線形フィルタの例として、メディアンフィルタを紹介します。メディアンフィルタは、 8 近傍画素の値をソートした配列の中間値を出力とします。図 1.11に例を示します。左が原画像に胡麻塩雑音を加えた画像、右がメディアンフィルタを施した画像です。譜 1.8はメディアンフィルタを行なう関数です。 uchar_cmp は記述していませんが、 uchar 型を比較する関数です。

```
 1: void median_filtering
 2:   _P2 (( image, output ),
 3:        ( image, input  ))
 4: {
 5:   int x, y, i, j, n, xsize, ysize;
 6:   uchar **i_data, **o_data;
 7:   uchar pixels[9];
 8:
 9:   xsize = Image.xsize( input );
10:   ysize = Image.ysize( input );
11:   Image.make( output, UChar, xsize, ysize );
12:   i_data = ( uchar ** )Image.data( input );
13:   o_data = ( uchar ** )Image.data( output );
14:
15:   for ( y = 1; y < ysize-1; y++ )
16:     for ( x = 1; x < xsize-1; x++ )
17:       {
18:         for ( n = 0, j = -1; j <= 1; j++ )
19:           for ( i = -1; i <= 1; i++ )
20:             pixels[ n++ ] = i_data[ y + j ][ x + i
];
21:         qsort( pixels, 9, 1, uchar_cmp );/* ソート
*/
22:         o_data[ y ][ x ] = pixels[ 4 ];/* 中間値 */
23:       }
24: }
```

譜 1.8: メディアンフィルタ処理

(a) 入力画像

(b) 出力結果

図 1.11: メディアンフィルタの例

1.5 CIL プログラミング練習問題

以下に、練習問題を挙げておきます。番号が大きいほどレベルが高くなっています。細かい仕様は、各自で決めて使いやすくしましょう。

[1.1] 画像の情報の表示

画像の属性などを表示するコマンドを作りなさい。

[1.2] ソーベルフィルタ

ソーベルフィルタを施すコマンドを作りなさい。

[1.3] 二値画像演算ツール

二値画像の論理演算を行なうコマンドを作りなさい。

[1.4] ラベリング

ラベル画像を再ラベリングするコマンド作りなさい。

[1.5] 一般フィルタ

一般的なフィルタで、畳み込み積分を用いたフィルタリング処理をするコマンドを作りなさい。

第 2 章

C 画像ライブラリ CIL

CIL は、C 言語で画像を扱うための基本機能を提供するライブラリです。現在では、CIL は画像を扱うためのライブラリだけではなく、コマンド・ラインでのオプションの指定などの、共通のインターフェースを用意しています。さらに、CIL には、効率よくドキュメントを作成するための LaTeX のスタイル・ファイルがあります。CIL の構想を理解し、正しく使えば、かなり効率良く研究を進めることができるでしょう。

この章では、CIL についての説明をします。

- CIL の歴史
- CIL ライブラリの構想
- CIL ライブラリの構成

2.1 CIL の歴史

2.1.1 昔昔のその昔

僕が、4 年生になって阿部研究室に入った当時は、実際に研究のプログラムを書くときは、J4 フォーマットの画像ファイルを読み書きするルーチンがあっただけでした。実際これは、

まず自分のところにコピーしてきて、 それを自分用に変更して使う

というものでした (図 2.1)。ライブラリ化はされていませんでした。

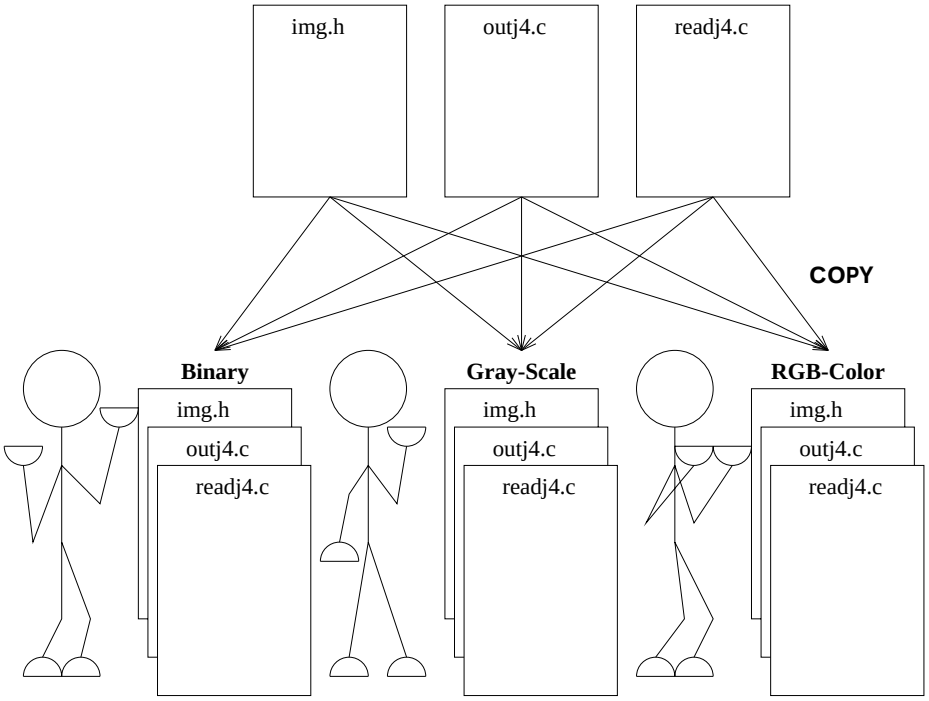


図 2.1: 過去の画像関係のルーチン

全体から見るとこれらのルーチンは多様化の方向に向かっていました。各個人にとっては、細かい指定ができてある意味で使いやすいでしょう。しかし、逆に自分で作らなければならないということがでてきてしまい、そういった作業で手間がとられたりしました。当然のことながら、バグが入る可能性も十分にあり、そのデバッグに研究時間をとられることもしばしば見られました。

実際、僕はこのルーチンを使いませんでした。そこで、自分なりに J4 ライブラリを作成していきました。

2.1.2 J4 ライブラリの誕生

J4 ライブラリは、ユーザから見て

「二値画像」も「濃淡画像」も同じように「画像」

として扱おうという考えで作成していきました (図 2.2)。

「カラー画像」に関しては、 Red, Green, Blue の「濃淡画像」が 3 つ集まったもの

として扱っていました。

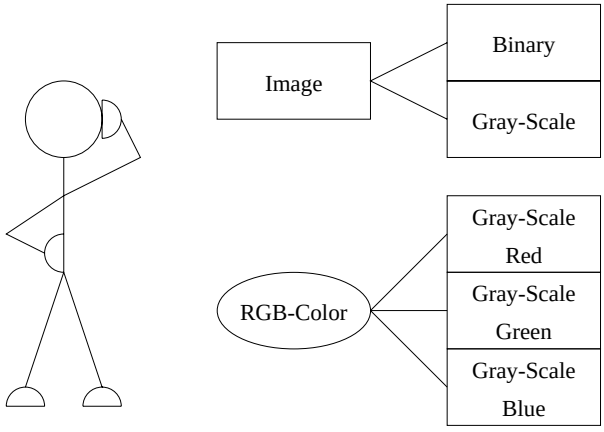


図 2.2: J4 ライブラリ構想

ファイル操作に関しても、J4 フォーマットに関しては、画像の型に依存せず一貫して行なうことができるようになりました。つまり、画像の型ごとに、関数を呼び出す必要はなくなりました (図 2.3)。

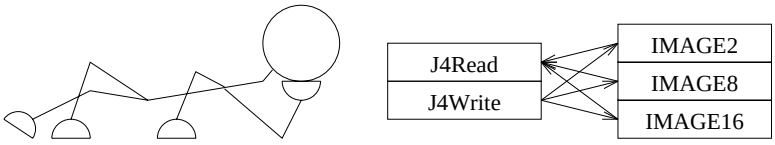


図 2.3: どんな型でも OK!

4 年生のころは、濃淡画像の輪郭線抽出に関する研究をしていたので、基本的には「濃淡画像」を主体においてライブラリを作成していきました。また、研究でよく使ったメモリ関係を操作するライブラリ、行列計算をするライブラリも作成しました。

もはや、このライブラリのソース・コードは残されていなくて、実際どういうものであったかは、はっきり思い出せません。自分の研究を進めていく上では、十分なライブラリであったと思います。

2.1.3 Image ライブラリの誕生

4 年生も終りになり、世の中には画像をファイルに格納するのにいろいろな形式があることが分かってきました。また、共通のライブラリの必要性が感じられました。そこで、 4 年の終りに共通ライブラリの基本構想を練りました。そこで作成したのが、 Image ライブラリです (図 2.4)。

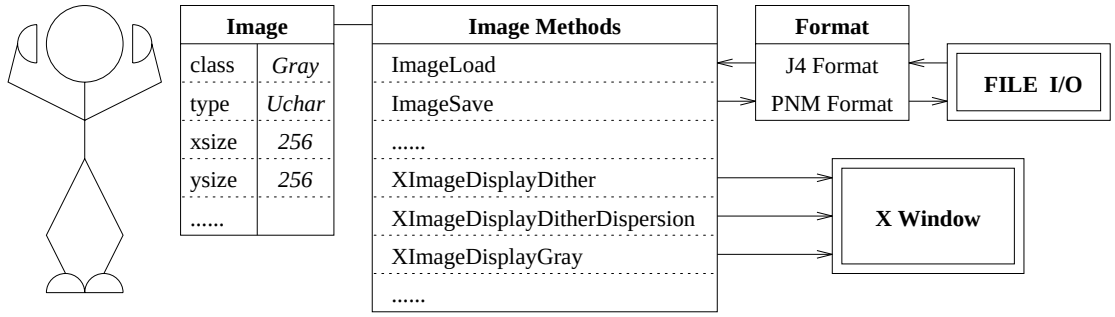


図 2.4: Image ライブラリの構想

Image ライブラリは、基本的には J4 ライブラリと同じですが、画像のタイプとクラスと分けて考えているところであるとか、新しく PNM フォーマット[1]をサポートしている点で違いがあります。

画像のタイプは画素の論理的な値を規定するもの

で、

画像のクラスは画素の値の意味を規定するもの

です。画像のタイプには、 Uchar, Short, Long, Float, Double がありました。画像のクラスには、 Red, Green, Blue, Gray, Binary, RGB, HSV がありました。

[1]PNM フォーマットは、 pbmplus の中の画像のフォーマット変換を行なうための中間フォーマットです。

また、 Image ライブラリはその他に、画像の演算、各種フィルタ、 RGB から HSV への変換、 X Window への二値ディザ表示などを行なう関数群も用意してありました。

このころから、ライブラリはオブジェクト指向を意識して作られています。といっても、完全にオブジェクト指向ではなく、今思うと、データ抽象の考えで留まっていたように思います。

2.1.4 CoreImage ライブラリの誕生

さらに、オブジェクト指向の勉強の深まった修士 1 年に、もう一度、 Image ライブラリを作り直そうと考えました。 Image ライブラリは、かなり使いやすくなりましたが、まだ不満が残りました。それは、カラー画像の扱いと、画像タイプと画像クラスが J4 フォーマットに大きく依存していたことです。そこで、もっと基本的なところからデータ構造を作り直そうということで、 CoreImage ライブラリを新しく作りました (図 2.5)。そして、画像のフォーマットは、 JPEG, GIF, PNM, J4 の読み込みと書き込みができるようになりました。

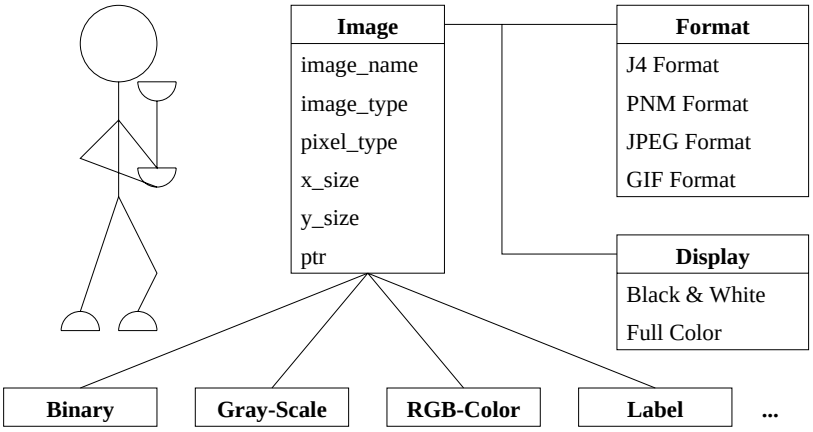


図 2.5: CoreImage ライブラリの構想

Image ライブラリでは、濃淡画像 3 枚でカラー画像を表現していました。つまり、プレーン独立型にして扱っていました。しかし、 Red, Green, Blue を独立して使うことはほとんどないことから、 CoreImage ライブラリでは、カラー画像の画素を UChar の 3 次元の配列の画素結合型として扱っています (図 2.6)。これによって、

「カラー画像」も「二値画像」も「濃淡画像」も同じ「画像」

として扱うことができるようになりました。

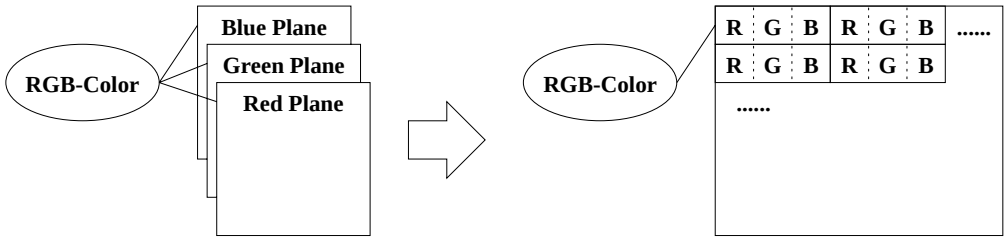


図 2.6: RGB 画素分離型と RGB 画素結合型

CoreImage ライブラリでは、画像タイプは画素タイプに、画像クラスが画像タイプに変更されました。これにより、画素と画像そのものの意味づけをはっきり区別することにしました。基本的に、濃淡画像は、 UChar, UShort, Float, Double など、どれでもいいのです。カラー画像も、 UChar3, Long3, Float3 など、どれでもいいわけです。また、特別な画像として、 UChar2, UShort2 など画素の次元が 2 次元でも扱うことができるようになりました。

CoreImage ライブラリはもう一つ特徴がありました。それは、画素値を参照したり変更したりする関数が準備されていたことです。 CoreImage ライブラリでは、画素型は次元と型で定義されていました。この情報を使って、どのような画素タイプに対しても一つの関数で読み書きができたのです (図 2.7)。

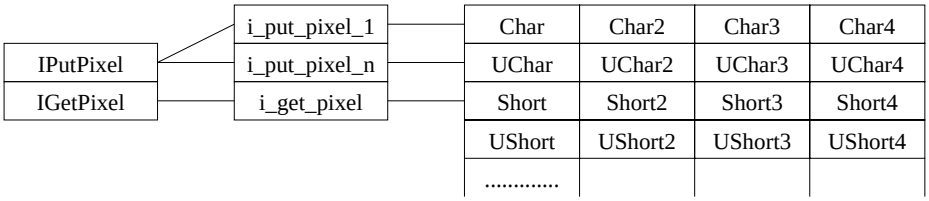


図 2.7: 画素値 get/put 関数

しかしながら、研究でプログラムを組むときは、使用する画素タイプというのは決まっているから、この関数はそんなに必要ではなかったと思う。

2.1.5 ImageTemplate ライブラリの誕生

修士 1 年の夏になって、 NTT の実習に行ってきました。ここでも、画像の研究をしてきました。計算機の環境は周りの人のを見ていると、非常に恵まれてい

ました。で、僕はというと、 PC98 を端末にしてホスト・コンピュータにアクセスして使うというもので、 X Window のない恵まれない環境で作業にとりかかりました。さらに、恵まれなかったことは、使用言語は FORTRAN だったこと。これにはまいって、担当の人と相談したら、「今あるプログラムを書き換えてくれるならいいよ。」ということなので、さっそく C 言語でやることに決めました。

担当の人は C 言語は良く分からないみたいだったので、他の社員の人に C 言語で画像を扱うためのライブラリを探し回ったけど、標準のライブラリはないと言われてしまった。他の画像関係のプロジェクトの人に聞いたけど、やっぱりなかった。とりあえず、何か作ろうと考えて、テレビ電話用のフォーマットだけを重点において作成した。しばらくすると、テレビ用のフォーマットも作って欲しいと別の社員の人に頼まれた。考えたすえ、画像クラスのテンプレートを作ってそれを書き直せば、どんなフォーマットでもできるようにした。

XToolkit の作法を真似て 拡張性が高くなるように作った。

それで、テンプレートを元にして作っていったので、 ImageTemplate ライブラリと呼ぶことしにした (図 2.8)。

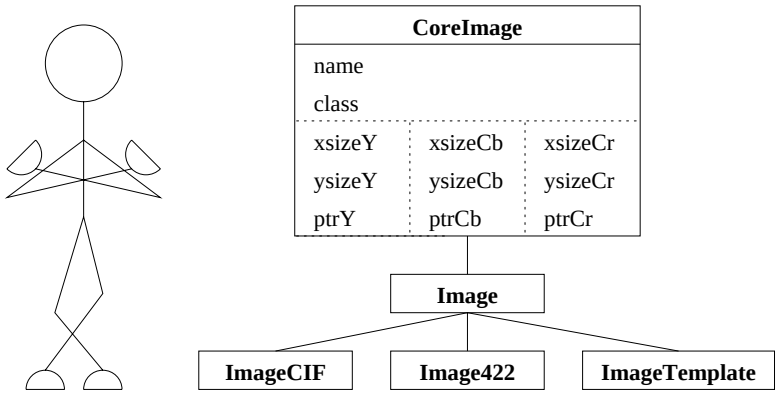


図 2.8: ImageTemplate ライブラリの構想

しかしながら、これを実現するのは非常に面倒な作業だった。拡張性に富んではいるが、拡張するにはそれなりの知識と作業がいるので、知らない人は拡張できない。実現してしまえば、使う側にとっては、使いやすいライブラリだったと思う。

今思うと、ちょっと、特殊なライブラリだった。しかし、要求は満たされたライブラリだったと自己満足のうちに実習は終った。

2.1.6 旧 CIL ライブラリの誕生

ここまでくると、 CIL の基本構想は頭の中にだいたいでき上がりました。すでに、修士 1 年も終りに近付いてきました。そろそろ、ライブラリをまとめようと考えて、今までの経験をもとに最初の CIL ライブラリを作成しました (図 2.9)。

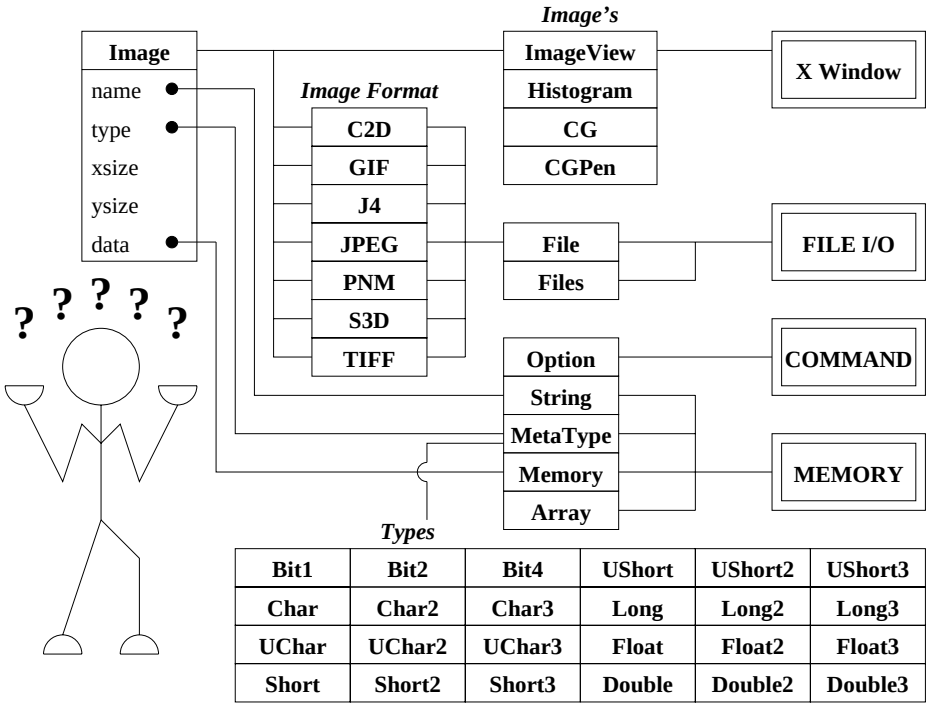


図 2.9: 旧 CIL ライブラリの構想

実際、 CoreImage ライブラリでも十分だったのですが、画像タイプに大きな問題点がありました。それは、画像タイプは、定義されていないものは扱うことができないというものです。距離画像、特徴画像など、画像のタイプはいくらでも作り出すことができるのです。ここでは、

> 画像タイプは必要ないと判断し、
> 画像の意味づけは、
> 実際にプログラムを作る人が自由に決める

ということにしました。

画像の意味づけを自由に行なうためには、このままだと、画素タイプに拘束されてしまいます。そこで、基本的な画素のタイプだけは用意して、「必要ならば自分の画素タイプを定義することができる」ようにと考えました。いわゆる、

meta_type を導入

しました。この導入によって、画像の操作は一貫性をもつことができ、自分で定義した画素タイプに対しても X Window への表示が、何の変更なくそのままできるようになりました (図 2.10)。

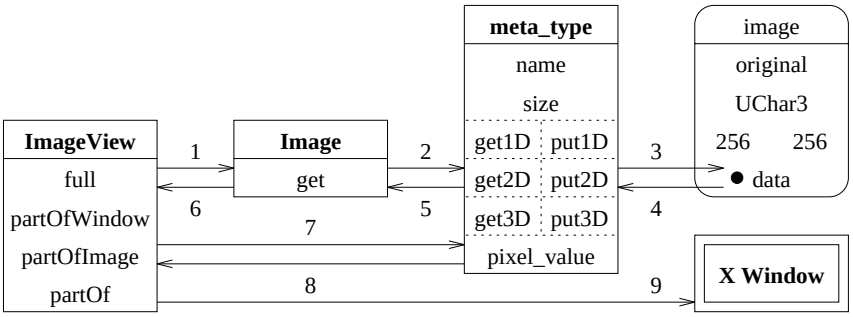


図 2.10: 表示までのデータの流れ

図 2.10は、 X Window へ表示するまでのデータの流れです。まず最初に、 ImageView が呼ばれ、 Image, meta_type を通して、画像の持つ画素値を得ます ($1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6$)。次に、その画素値を、 meta_type を通して、 RGB 24bit の画素値に変換します ($7 \rightarrow 8$)。最後にこれを X Window 上での画素値に変換して表示します。

つまり、各型は「データのアクセスの仕方」と「画素値の RGB 変換」を持たせ、表示する時は、それをもとに画素値をアクセスし、それを変換して X Window にデータを送るようにします。このようにして、画素の参照や変更も、統一的にできるようになりました。しかし、この方法では、実行速度の面からして非効率的といえるでしょう。いくら汎用にするとはいえ関数の呼び出しのオーバーラップが大き過ぎます。この場合、汎用性をとるか実行速度をとるか、非常に難しいところでしょう。

このころになると、頭はオブジェクト指向一本になってしまい、オブジェクト指向をどこまで実現できるか、 C 言語の限界に挑戦していました。実際、 NTT の実習の合間をぬって、 C 言語によるオブジェクト指向をどう実現するか、いろいろ考察を練っていました。そして、これらの (浅い) 経験をもとに、オブジェクト指向にこだわって、ライブラリを構成しました。結局、全体として、ごてごてしたライブラリになってしまいました。 C 言語で無理に、オブジェクト指向を実

行しようとした末路です。なんでもかんでも、オブジェクトとして捕らえてはいけないという、良い例です。

2.1.7 CIL ライブラリの誕生

しばらく旧 CIL を使っていて、なにかと面倒な点がいくつかでてきました。時はすでに修士 1 年が終った春休みになりました。そろそろ、もう一度、ライブラリを見直そうと思いました。

果たして、オブジェクト指向は必要だったろうか？
データ抽象の考えだけでいけないか？
研究に最低限必要なのはなにか？
汎用性はどの程度保つか？
プログラムを組むうえで手間のかかる部分は何か？

そんな思考と今までの失敗や経験から、 CIL の構想はできあがってきました。

2.2 CIL ライブラリ構想

CIL は、

C 言語で
Image(画像) を扱うための
Library(ライブラリ) を作ろう

という考えの中で生まれました。この節では、 CIL の構想について述べます。

2.2.1 CIL 基本構想

CIL は、基本的にデータ抽象化の考えを用いて、 image を中心に構成されています。ユーザは、アクセス関数を通して画像をアクセスします (図 2.11)。ユーザは、画像のデータ構造を直接アクセスする必要性はありません。細かい作業や面倒な作業は CIL が行ないます。

さらに、 CIL では画像だけでなく、入出力関係の半分がデータ抽象化されています (図 2.12)。

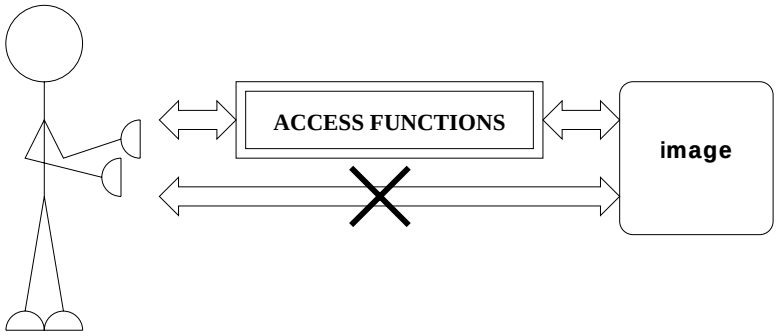


図 2.11: 画像へのアクセスはアクセス関数を通して

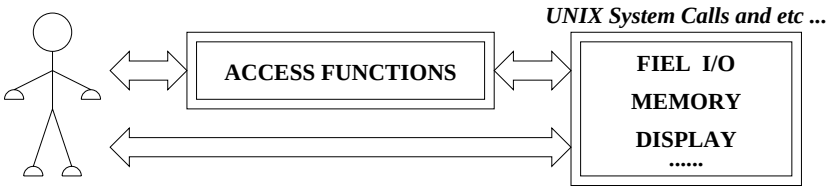


図 2.12: 入出力の抽象化

2.2.2 CIL の 3 大機能

C 言語で画像を扱うときに必要なルーチンのほとんどが、ユーザのボトルネックになっています。CIL はこれらの基本ルーチンをユーザに提供し、ユーザがプログラムを組むときに、できるだけ研究の部分に集中できるように考えてあります (図 2.13)。

CIL の 3 大機能であり、画像の研究には必要な処理で、ユーザが特にボトルネックになる部分は、

- 画像ファイルの読み書き処理
- X Window への表示処理
- コマンド・ラインのオプション処理

だと思います。どの処理も、特別な知識や (多少) 高度な技術がないと簡単には作れません。もしこれをユーザが一人で作るとなると、かなりの負担になり、それに関する勉強とバグ取りで何日か要すると思います。

全節でも説明したように、CIL はデータ抽象化されています。当然のことながら、これらの処理も抽象化されています。ユーザは必要なときに、これらのサー

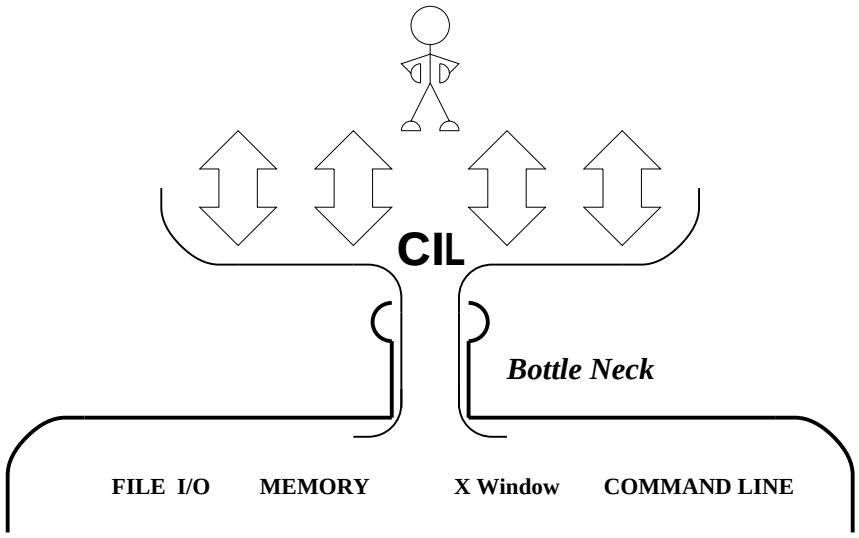


図 2.13: ボトルネック

ビスを受けることができます。画素型がなんであるかとか、ディスプレイのクラスがなんであるかとか、何も気にする必要はありません。

以下に、これらの部分に関して CIL の考えを述べます。

画像ファイルの読み書き処理

現在、画像をファイルに格納しておくフォーマットが数多くあります。基本的に、ユーザにとっては、それらのフォーマットを気にすることなく、

画像ファイルはどのようなフォーマットであっても、 同じ画像ファイルとして読み書きできる

というのが理想的です (図 2.14)。

さらに、どのような画素型に対しても読み込み書き込みができなければ、研究においては差しつかえが出てきてしまいます。通常、画素型は、 Bit1 (二値画像), UChar (濃淡画像), UChar3 (カラー画像) の 3 種類です。しかし、研究では、いろいろな画素型を用います。例えば、 x, y 方向でそれぞれ画像を微分したものは Float2 を用いるでしょうし、ラベル画像ならば Long, UShort を用いるでしょう。また、もっと複雑なデータ構造を用いるかもしれません。 CIL ではこれらの任意の画素型の画像の読み書きをサポートします (図 2.15)。

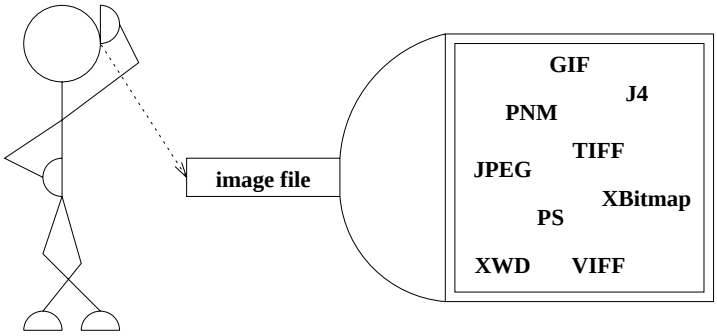


図 2.14: ユーザから見た画像ファイル

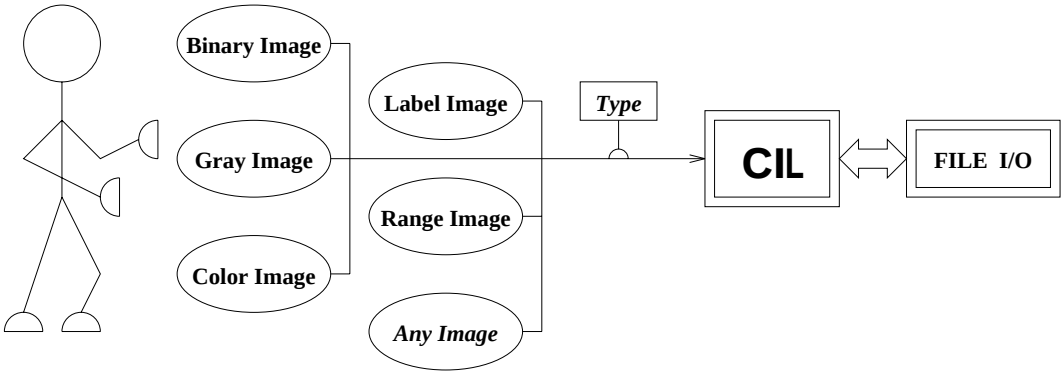


図 2.15: どんな画素型でも読み書きできます

X Window への表示処理

画像を X Window に表示するには、 X Window の知識が必要になります。また、いろいろなタイプのディスプレイがあります。モノクロ、グレイスケール、8bit カラー、フルカラーなど、どのようなディスプレイに対しても表示できるようにするのは至難の技です。

CIL では、ユーザが X Window に表示する関数を呼べば、あとは関数が、画像のタイプとディスプレイに応じて、最も適切であると思われる表示方法で画像を表示します (図 2.16)。ユーザはディスプレイだとか画像の型は気にしなくていいのです。

さらに、 CIL では、 X Window の知識がなくても、画像を表示することができます。そして、 X Window の一部のイベントを処理することもできます (図 2.17)。

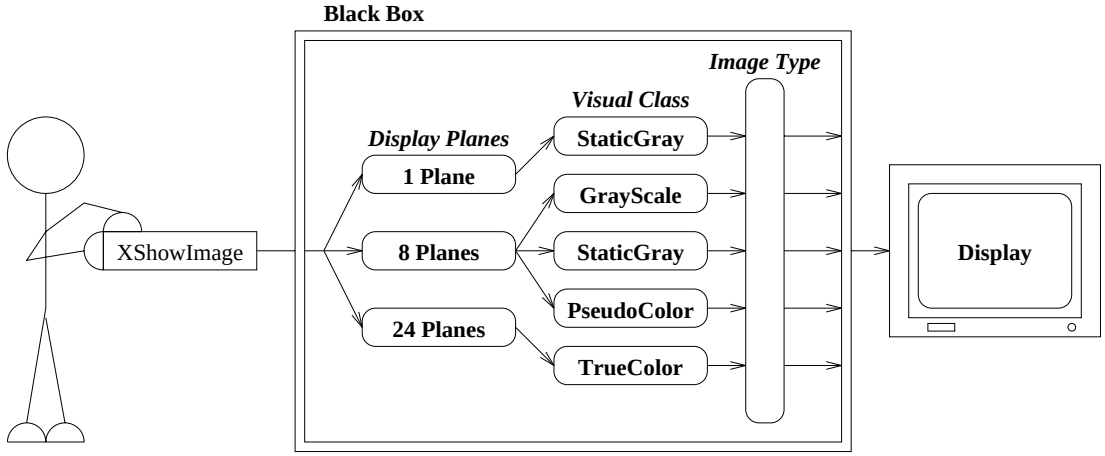


図 2.16: X Window への表示

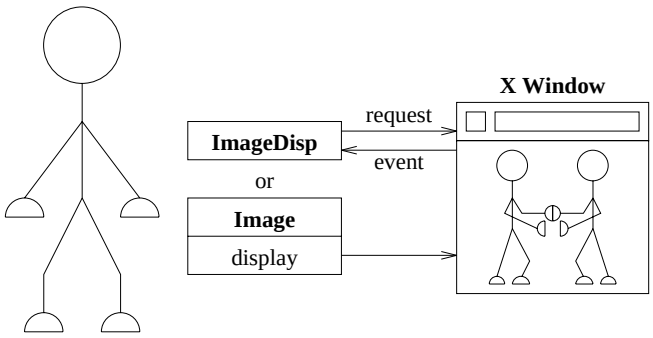


図 2.17: X Window を知らなくてもできます

コマンド・ラインのオプション処理

プログラムを作るときは必ずといって言いほど、コマンド・ラインのオプションの処理が必要です。これは、アプリケーションにパラメータを渡すための手段として必要な処理です。もちろん、対話的にパラメータを入力することも考えられます。いずれにせよ、パラメータの入力は必須の機能で、なんらかの処理でユーザからアプリケーションへパラメータを渡すことが必要となってきます (図 2.18)。

しかしながら、コマンド・ラインの処理は、作る人によって様々な方法があり、コマンド・ヘルプの表示も人によって様々です (ヘルプがないのは問題外)。実際に、このコマンド・ラインの処理を作るのは面倒です。対話的にパラメータを入力する処理は、エラー処理などを考えるともっと面倒です。また、ユーザから見

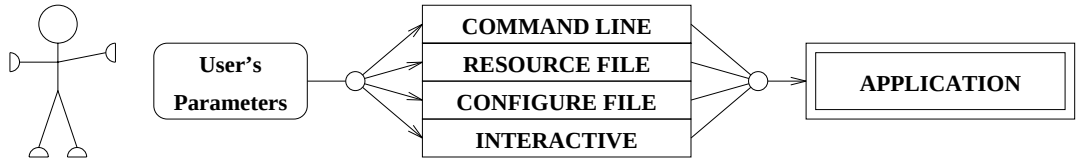


図 2.18: パラメータの渡し方いろいろ

れば、アプリケーションによってオプションの指定の様式が変わるのは嫌なものです。ユーザから見たら、同じインターフェースであることが望ましいのです (図 2.19)。

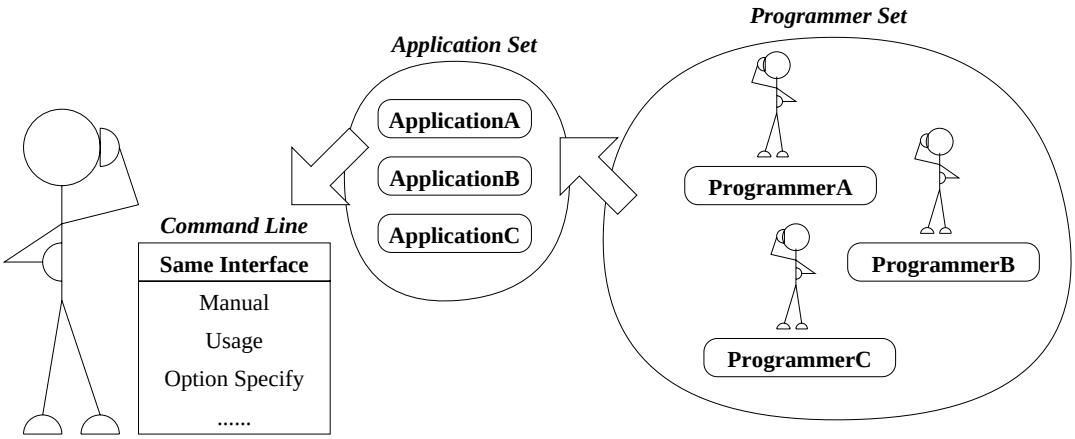


図 2.19: オプション処理はみんな同じにしよう

CIL では、このオプション処理をするためのサービス・ルーチンを準備し、コマンド・ラインでのオプションの指定方法やヘルプなどの共通のインターフェースを提供します。さらに、 CIL では、リソース・ファイル管理、コンフィグ・ファイル、対話的処理によるパラメータ入力なども準備しています。

2.2.3 共有メモリ画像システムの構想

共有メモリ画像システムとは、画像を共有メモリ上に置き、その画像をファイルシステム上の画像ファイルと同じよう扱うためのシステムです (図 2.20)。このシステムは、サーバによって共有メモリ画像ファイルの排他制御を行なっています。ユーザは、 ImageFile を通してみると、共有メモリ上にある画像も、あたか

もファイルシステム上にある画像ファイルと同じに見えるのです。

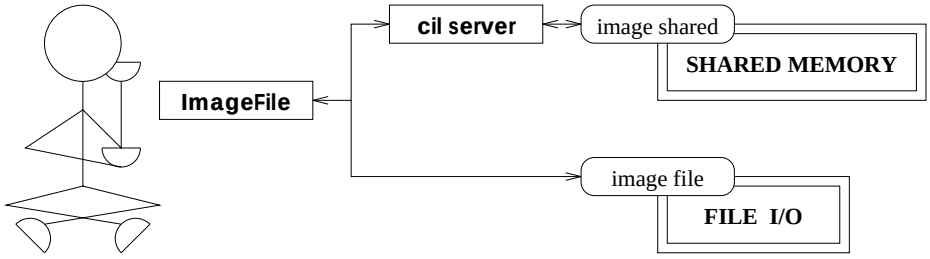


図 2.20: 共有メモリ画像システムにおけるファイル

共有メモリはプロセス間通信機構の一つです。この機構を利用すると、異なるプロセスから同じメモリ空間をアクセスすることができます。異なるプロセスで同じメモリ空間をアクセスできると、いろいろな利点があります。

- 高速で大量のデータの受渡しができる。
- データの一括管理ができる。
- プロセス間の同期制御することができる[1]。

共有メモリ画像ファイルシステムによって、ユーザがこれらの恩恵を簡単に受けることができるのです。利用者のアイデアで使い道もいろいろあると思います。例えば、「実験中にファイルに書き込むことなく途中結果の画像を表示したり」、「複数のプロセスが同じ画像を参照して処理を進める黒板画像処理を行なう」といったことも可能です。

また、このシステムはファイルシステムだけではなく、共有メモリとメモリ上にある画像でさえも、ユーザから見れば同じように扱うことができます。とりあえず、画像の処理の部分に関しては、共有メモリとかは気にしないで作ってもいいのです。画像を作成する時にのみ、共有メモリを指定すればいいのです。

2.3 CIL ライブラリ構成

CIL は、大きく分けて CIL 本体と Xcil と misc という 3 つのライブラリで構成されいます。それらの位置づけは、図 2.21のようになっています。 CIL 本体で用い

[1] もちろんプロセス同期のための専用の機能がプロセス間通信機構に準備されてます。

る入出力などの基本操作は、すべて misc ライブラリを使って書かれています。また、 Xcil は少し特別な位置づけにあります。 Xcil は単に Xlib の一部であって、アプリケーション、 CIL 本体、 X Window に対して透過なのです。

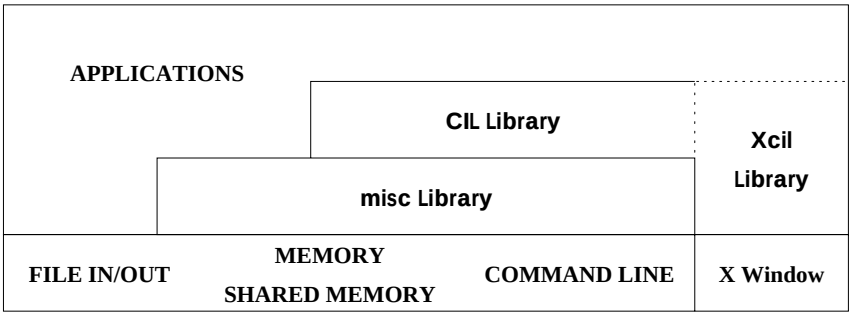


図 2.21: CIL ライブラリの階層

各ライブラリの機能は、表 2.1のようになっています。どのライブラリも基本的なサービス・ルーチンをユーザに提供します。

表 2.1: 各ライブラリの機能

名前	機能
CIL 本体	画像の操作、画像ファイルの入出力など
Xcil ライブラリ	X Window 上への画像の表示と基本図形の描写など
misc ライブラリ	入出力などの基本操作、オプション処理など

さらに、これらのライブラリを使って CIL サーバやクライアントが作られています。 CIL 本体の中には、 CIL サーバが動作していないと利用できないものもあります。これらのライブラリ全体の動作に関しては後節で説明します。

また、 CIL の中には拡張ライブラリ群があります。それらは cilext ライブラリと呼ばれ、画像処理手法の一部がライブラリ化されています。予定では、この cilext を二値画像、濃淡画像、カラー画像にクラス分けをし、代表的な画像処理手法をライブラリ化していきたいと思っています。

それでは、以下に各ライブラリの構成について説明します。

2.3.1 CIL 本体の構成

CIL 本体は、 image を中心に構成されています。基本的に image 構造体への操作は、 Image 関数群 (サービス・ルーチンの集まり) を通して行ないます。また、 Image は、 ImageFile, ImageDisp の 2 つの関数群、及び、 misc ライブラリによって構成されています (図 2.22)。

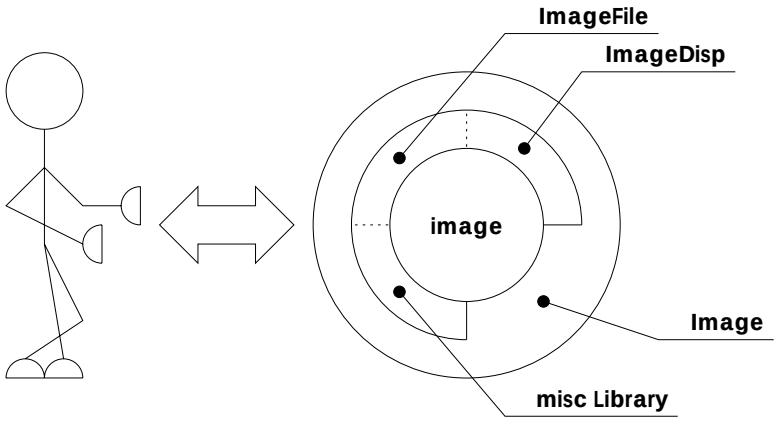


図 2.22: CIL 本体の構成

3 つの関数群 Image, ImageFile, ImageDisp は、それぞれ、 image, **File**, **Display** を抽象化します (図 2.23)。最終的に、 Image が画像に関係しているものを抽象化します。ユーザは、画像ファイルのフォーマットであるとかディスプレイのタイプであるとかは気にしなくていいのです。画像ファイルは J4, PS, PNM などの、どのようなフォーマット形式であっても、ユーザから見れば単なる画像ファイルなのです。ディスプレイも、モノクロ、グレイスケール、カラーなどなんであっても、ユーザから見れば単なるディスプレイです。

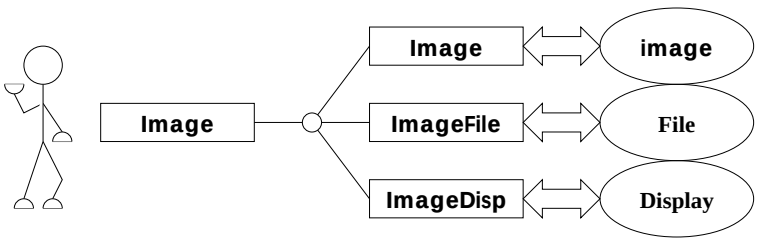


図 2.23: 抽象化の対象と関係

以下に詳しく、 image 構造体、 Image 関数群、 ImageFile 関数群について説明します。 ImageDisp 関数群については後節で説明します。

image 構造体

image は、 2 次元配列で表現された 2 次元画像を表現するための構造体です。画像を表現するための基本となる属性だけで構成しています。 image の持っている属性は、図 2.24のようになっています。

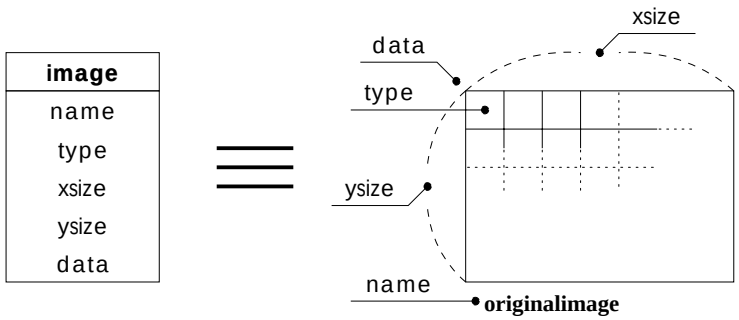


図 2.24: image 構造体の定義

以下に、各属性について説明します。

- **name**

  画像の名前を name に与えることによって、ユーザがその画像に意味づけをします。これは、デバッグのときや、サーバとの通信、共有メモリ管理などに用いられます。 C 言語からすれば名前は単なる文字列ですが、 CIL では名前の持つ意味は重要です。

- **type**

  type は画素型で、画素型は画素の構造を規定するものです。実際には、 type は型識別子を値として持ちます。型識別子は、 misc/typelib.h に宣言されており、 PackedBit1, Bit1, Bit4, Char, Char2, Char3, UChar, UChar2, UChar3 などが定義されています。もちろん、ユーザが新しく定義することもできます。 CIL は、この型識別子から画素に関する情報を得ます。

- **xsize, ysize**

  xsize, ysize は画像のサイズです。実際には、画素数を与えます。

· **data**

画像データへのポインタです。画素のデータが入っている二次元配列へのポインタが入りますが、具体的な型は持っていません。実際、ユーザがこのポインタを使うときは、なんらかの型の二次元配列のポインタに代入してから使うことになります。

Image 関数群

Image 関数群は、 image 構造体の操作を行なうためのサービス・ルーチンの集まりです。実際に持っている大まかな基本操作は図 2.25のようです。 Image は image を抽象化しているので、ユーザは Image の持つ基本的な操作に関して画素型を気にしなくていいです。

Image	
create/destroy	- - -➢ create, destroy, createFromImage, createFromFilename, createMake
make/free	- - -➢ make, free
load/save	- - -➢ load, save
reference	- - -➢ name, type, xsize, ysize, data, area, byte, raster, print, display, undisplay
change	- - -➢ copy, clear, resize, sub, swap

図 2.25: Image 関数群の大まかな内容

Image 関数群は、大きく分けて 5 つのグループに分けることができます。それぞれ、生成破壊、確保解放、ファイル入出力、参照、変更です。以下に、基本操作ごとに機能の説明します。

· **生成破壊： create / destroy**

create / destroy は、 CIL において必須の機能です。 create は、 CIL に対して画像を使用することを宣言し、**属性をいれる領域**を確保します。 destroy は、 CIL に対して画像を破棄し、**属性をいれる領域**を解放します (図 2.26)。

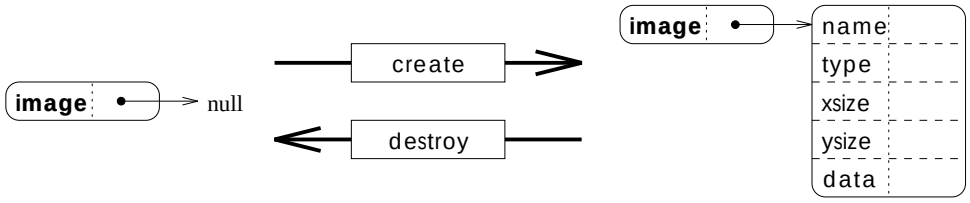


図 2.26: image の生成と破壊

create と destroy は、一つのプログラムの中で必ず “対” になります。基本的に、一つの関数の中でも対にした方が理解しやすいでしょう。 CIL のサービスを受けるためには、まず最初に create を行ないます。一度生成された image は、 destroy が行なわれるまで有効です。 create には、他の関数と組み合わせた、 createFromImage, createFromFilename, createMake があります。実際に、これらは内部で、 copy, load, make が呼ばれます。

- **確保解放： make / free**

make / free も CIL において必須の機能です。 make は、生成されている画像に対して、**画像のデータの領域**を確保します。 free は、生成されている画像に対して、**画像のデータの領域**を解放します (図 2.27)。

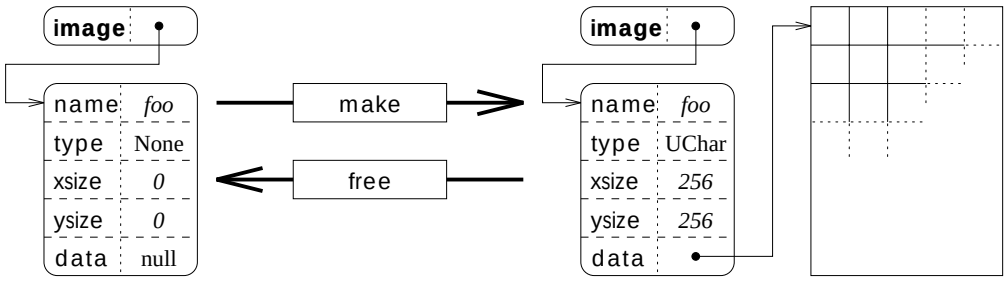


図 2.27: image のデータ領域の確保と解放

領域の確保は、通常メモリ上に行なわれますが、共有メモリ上に作成することもできます。

· **ファイル入出力：load / save**

画像ファイルの読み込みと書き込みを行ないます。load はファイルシステムや共有メモリから画像を読み込み、save はファイルシステムや共有メモリに画像を書き込みます。ユーザは、画像ファイルのフォーマットや画素値の型を気にすることなく、読み込みと書き込みを行なうことができます。これらに関しては、ImageFile で詳しく述べます。

· **参照：reference**

image の持つ属性にアクセスするための関数群です。ユーザは、必ずこの関数を通して image にアクセスします。参照できる属性は、name, type, xsize, ysize, data, area, byte, raster です。また、画像を参照する、print, display, undisplay もあります。X Window への表示もこの関数群に属しています。X Window の表示などに関しては、ImageDisp で詳しく述べます。

· **変更：change**

image の属性の変更を行ないます。ユーザは必ずこの関数を通して変更を行ないます。画像の属性の変更には、copy, clear, resize, sub, swap があります。もし、ユーザが image の属性の変更を直接行なったときは、ライブラリの動作は保証できません。

ImageFile 関数群

ImageFile は、画像ファイルの読み込みと、メモリ上の画像の書き込みを行なうための関数群です。ImageFile の関数群は図 2.28のようになっています。また、右側の表は ImageFile がサポートしている画像ファイル・フォーマットです。

ImageFile は画像ファイルをファイル名によってデータ抽象化しています。ユーザは、画像をファイルから読み出したり、ファイルに書き込んだりする場合には、画像ファイルがどのようなフォーマット形式であるかとか、どのような画素型をもっているかなどは気にする必要はありません。

load と save は、それぞれ、画像ファイルの読み込みと画像の書き込みを行ないます。getLoadFormat は、最後に読んだ画像ファイルのフォーマットを返します。getComment は、最後に読んだ画像ファイルのコメントを返します。set-

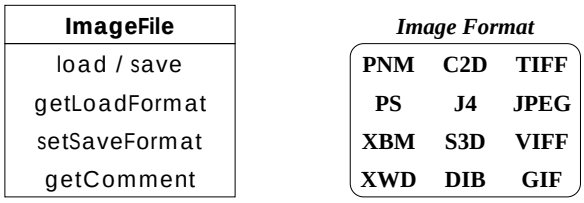


図 2.28: ImageFile とサポートしているフォーマット

SaveFormat は、書き込む画像ファイルのフォーマットを指定します。ここでのフォーマットは、すべてフォーマット名 (文字列) で扱います。

また、画像ファイルが別のコマンド (compress, gzip など) によって圧縮されている場合でも読み込むことができます (図 2.29)。最初に ImageFile は、圧縮されたファイルかどうかを識別します。そして、圧縮されていたならば、それを展開のコマンドでファイルを展開してから、画像ファイルとして読み込むという作業を行ないます。ラベル画像などは、かなりの圧縮率が期待できるので圧縮しておくと良いと思います。

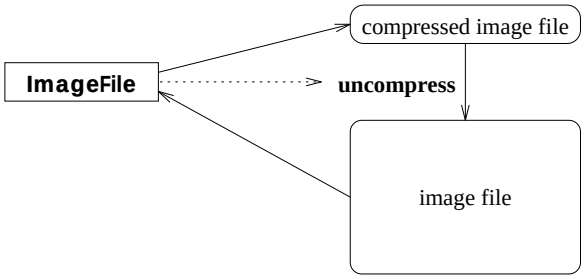


図 2.29: 圧縮されたファイルも読めます

ImageDisp 関数群

ImageDisp は、画像の X Window への表示と、 X Window からのイベントの獲得を行なうための関数群です。 ImageDisp の関数群は図 2.30のようになっています。右側の表は ImageDisp が処理できるイベントです。

ImageDisp は X Window 上のウインドウを image によってデータ抽象化しています。ユーザは X Window のウインドウ ID やディスプレイを気にすることなく画像に対して画像の表示やマウスやキーボードに対するイベント処理ができます。

Exec と Quit は、それぞれ、画像の表示コマンドの起動と終了です。 OK は、

図 2.30: ImageDisp とイベント

X Window 上の画像が通信可能であるかチェックします。 SelectEvent は、処理するイベントを選択します。 NextEvent は発生したイベントを一つ返します。 CheckEvent はイベントか発生したかどうか調べて、発生していればそのイベントを返し、発生していなければイベントが発生していないことを返します。

この関数群を使ってプログラミングする場合は、イベント駆動型のプログラムになります。また、 ImageDisp で処理できるイベントは、マウスがウインドウに入った時 (MouseEnter)、出た時 (MouseLeave)、動いた時 (MouseMove)、マウスボタンが押された時 (MousePress)、離された時 (MouseRelease)、キーボードが押された時 (KeyboardPress)、離された時 (KeyboardRelease) です。イベント駆動型のプログラミングに関しては X Window のプログラミングに関する書籍に説明を委ねます。

2.3.2 Xcil ライブラリの構成

Xcil ライブラリは、画像を X Window 上に表示するための Xlib レベルでのライブラリです。つまり、プログラマは X Window の知識がないと Xcil は使えません。さらに、 Xcil の拡張部分では、 XToolkit 相当のことができる関数群もあります。これには XToolkit の知識がないと使えません。

Xcil のライブラリの構成は図 2.31のようになっています。ここでは、画像を表示する関数群を説明し、拡張部分については別の節で説明します。

画像処理に関する関数群は、「CIL で定義されている image」を「Xlib で定義されている **XImage**」に変換する関数で成り立っています (図 2.32)。 ImageShowing 関数群は、「画像の画素型に意味を持たせて」 XCreateImage 関数群で表示できる型に変換します。 XCreateImage 関数群は、「ディスプレイのプレーン数」、「ビジュアル・クラス」、「画像の画素型」から最適と思われる表示方法を決定し、表示用の画像を生成します。 XShowImage 関数群は、その生成され

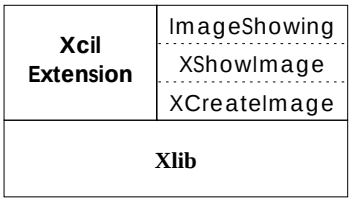


図 2.31: Xcil の構成

た画像を X Window に表示します。

2.3.3 misc ライブラリの構成

misc ライブラリは、 CIL 本体を作成していく上で必要になった基本操作を集めたライブラリです。「ファイル関係」、「メモリ関係」、「共有メモリ関係」、「コマンド・ライン」へのアクセスは、すべて、これらの misc ライブラリよって行なわれます。 misc ライブラリは、表 2.2のように、 8 個の関数群によって構成されています。

表 2.2: misc ライブラリの構成

名前	機能
optlib	オプション、リソースなどの管理と操作
typelib	型の管理
fillib	ファイル操作とディレクトリ操作
fileslib	複数ファイル入出力
memlib	メモリの操作
shmemlib	共有メモリの管理と操作
strlib	文字列の操作
socketlib	ソケットの操作

- optlib **関数群**

  コマンド・ラインのオプションに関する処理を行ないます。これに関しては後で詳しく述べます。

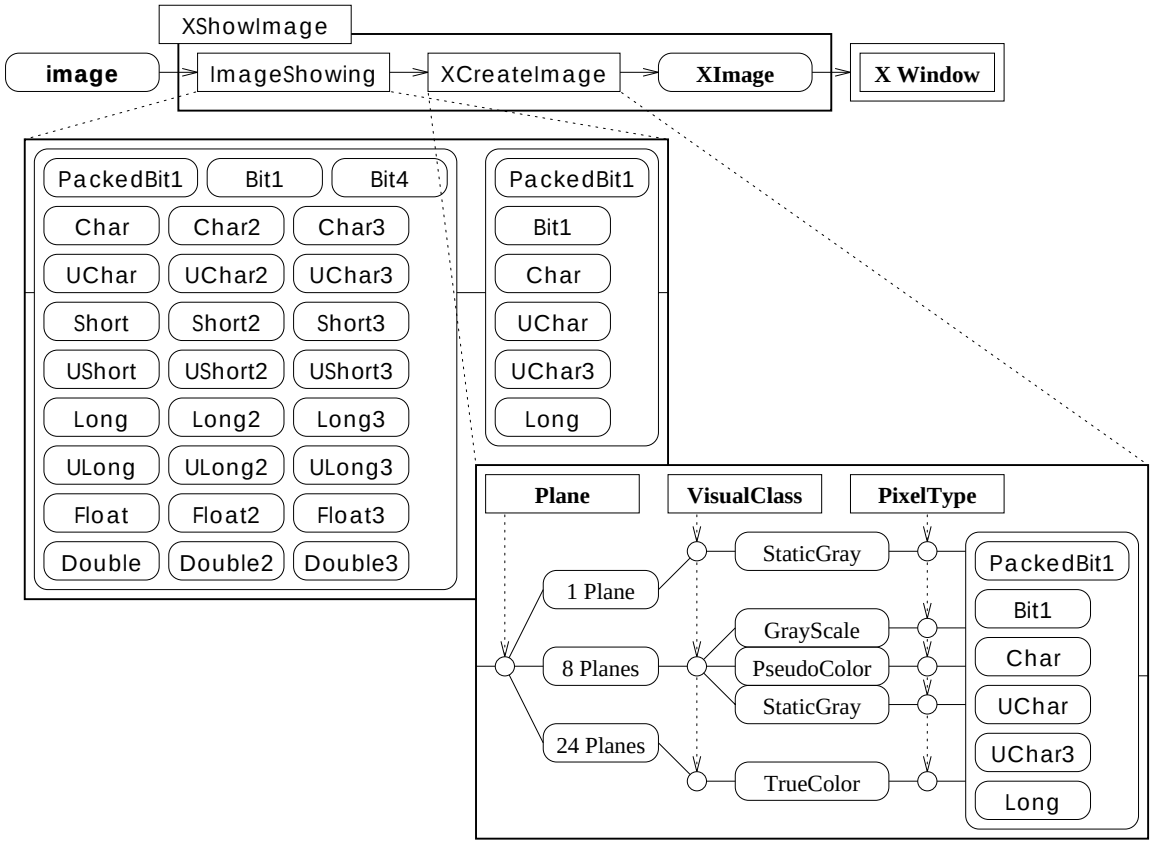


図 2.32: ImageShowing と XShowImage と XCreateImage の仕組み

· typelib **関数群**

typelib は、型の管理を行ないます。型情報は名前と識別子とサイズで構成されています。 CIL は、この型識別子を使ってサイズを得て、任意の型の画像の操作を実現しています。

· filelib, fileslib **関数群**

ファイル操作に関する関数群で、ファイルの入出力を行ないます。画像などの大量のデータをパイプを使って受渡しすることも考慮して、バッファリング処理に対応しています。

· memlib, shmemlib **関数群**

メモリ、共有メモリに関する操作を行ないます。1 次元、2 次元、3 次元配列の確保と解放ができます。

· strlib **関数群**

文字列に関する操作を行ないます。文字列から他の値への変換、文字列の分離、JIS から EUC への変換などが使えます。

· socketlib **関数群**

ソケットを使った通信に関する処理を行ないます。CIL サーバは、このソケットを用いて通信を行ないます。

optlib **の説明**

optlib は、ユーザとアプリケーションのパラメータの引渡しを行なうための関数群です。プログラマが optlib に与える情報は、「パラメータ名」、「オプション指示」、「パラメータ数」、「初期値」、「説明」です (図 2.33)。このリストとコマンド・ラインの入力を与えることにより、optlib は、これらを解析しパラメータを得ます。さらに、この情報からマニュアルや使用法を表示することができます。プログラマは、これらの体裁を気にする必要はありません。optlib が体裁良く表示してくれます。

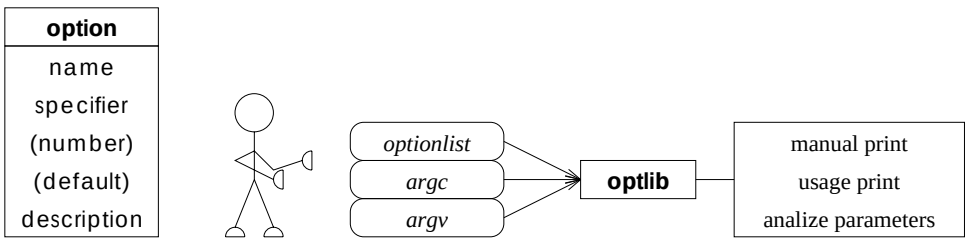


図 2.33: optlib に与える情報

プログラマは、パラメータの値が必要な場所でパラメータ名を指定して optlib から値をもらいます (図 2.34)。optlib は、その名前を使って、(1) コマンド・ライン、(2) 初期値、(3) コンフィグ・ファイル、(4) リソース・ファイルの順にパラメータを探しに行き、すべてなければユーザに値の入力を要求します。

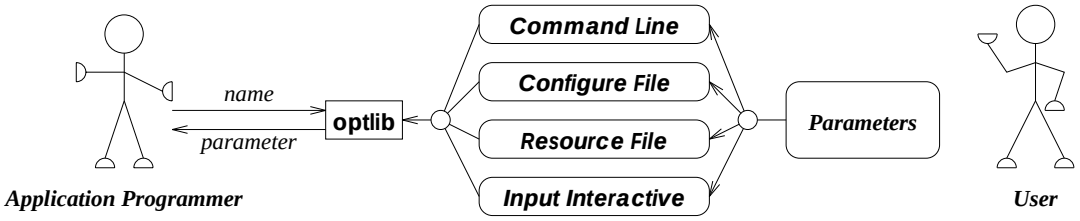


図 2.34: ユーザと optlib のやりとり

optlib は、プログラマの負担を減らし、ユーザにアプリケーションに値を渡す共通のインターフェースを提供します。

2.3.4 全体のまとめ

CIL ライブラリの全体像を、もう一度まとめて図 2.35に示しておきます。

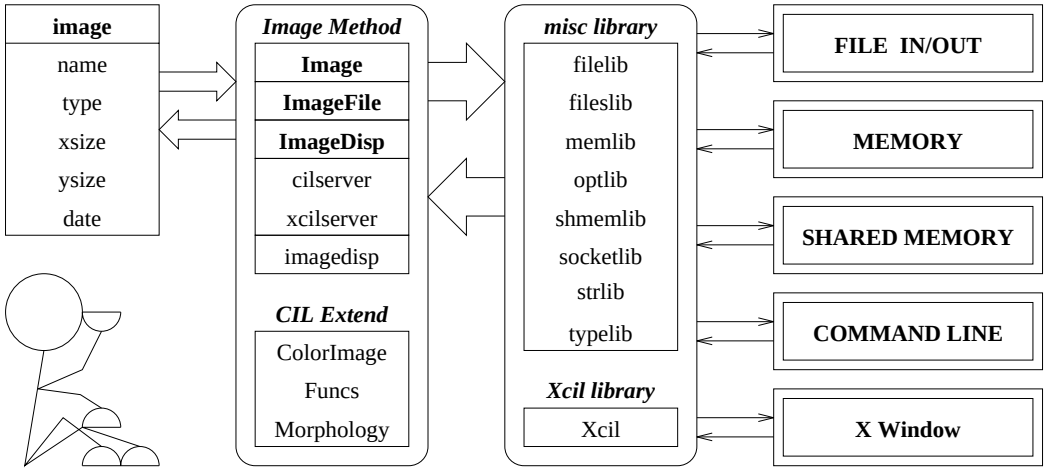


図 2.35: CIL ライブラリの構想

図ように、 image へのアクセスは基本的に Image を通して行なわれ、さらに misc ライブラリを通して「ファイル入出力」、「メモリ操作」、「共有メモリ操作」、「コマンド・ライン処理」、「X Window 関係」を行ないます。

第 3 章

CIL 使い始め

CIL の実際の構成を知ることによって、 image の構築のようすや、システム資源 (ファイル、メモリ、共有メモリなど) をどう扱っているか理解できます。この章では、具体的に CIL の説明をします。

ここでは、 CIL について以下のことを説明します。

- どのようなファイル、ディレクトリ構成か
- どうコンパイルするのか
- 環境変数はどういったものがあるのか

3.1 ファイルとディレクトリ構成

現在、 CIL のホームディレクトリは以下のディレクトリにあります。このディレクトリを CILHOME で示すことにします。

```
/home/abe/common/src/CIL
```

CILHOME は、以下のディレクトリのように、大きく 3 つのグループに分けることができます。

(1) **システムディレクトリ**には、コンパイルやライブラリのテストのときに必要なファイルとドキュメントが入っています。

```
./
include/
lib/
bin/
shared/
test/
manuals/
```

(2) **基本ディレクトリ**には、 CIL の基本関数群や宣言ファイル、サーバとクライアント群、基本コマンド群が入っています。

```
./
misc/
Xcil/
ImageFile/
server/
clients/
```

(3) **拡張ディレクトリ**には、画像処理の関数群が入っています。

```
Image/
BinaryImage/
GrayImage/
ColorImage/
```

以下に、詳しく各ディレクトリについて説明します。

3.1.1 システムディレクトリ

システムディレクトリは、CIL ライブラリを作成しテストするのに必要なファイルが入っています。CIL のテストは完全に閉じた環境、つまり、CILHOME の中で行なうことができます。実際には、システムディレクトリは一般ユーザにはまったく関係ありません。以下に各ディレクトリについて説明します。

CILHOME(.) ディレクトリ

このディレクトリで、システムと関係あるファイルは Makefile です。実際には、SunOS4.* 用の Makefile.sun4 と SunOS5.* 用の Makefile.sun5 があります。この中の Makefile は Makefile.sun4 がリンクされています。

include ディレクトリ

CIL のヘッダファイルがリンクされています。このディレクトリはコンパイルとテストのときにしか参照されません。この中には、ユーザには公開されていないヘッダファイルもあります。例えば、cilserver.h は、サーバと直接通信をするクライアントしか必要ないファイルなので、一般ユーザには公開していません。

lib ディレクトリ

ここには、テスト用のライブラリが入るディレクトリです。一般ユーザはこのディレクトリを参照することはありません。

bin ディレクトリ

このディレクトリには、実際に CIL のクライアント群のテストに使用されます。これらの実行形は、各ディレクトリにある実行形が直接がリンクされています。

shared ディレクトリ

このディレクトリは、共有ライブラリを作成するときのオブジェクトファイルが一時的に格納されるディレクトリです。

test ディレクトリ

このディレクトリは、CIL のいろりろな機能を実現する上で、思考錯誤されテストされたファイルが入っています。興味があれば見てみるのもいいかもしれません。

manuals ディレクトリ

このディレクトリは、オンライン・マニュアルが入っています。また、CIL に

ついてメールで報告したドキュメントなども入っています。まだ、完全なマニュアルはありませんが、現在執筆中です。

3.1.2 基本ディレクトリ

このディレクトリは、 CIL 本体の全ソースファイルが入っています。さらに、 CIL 内部の動きが知りたい人は、このディレクトリを探索してファイルの中身を見ると CIL のすべてが分かります。

CILHOME(.) ディレクトリ

このディレクトリには、 image の定義、関数群など image についてすべての基本ルーチンがあります。また、 3 次元画像である voxel の定義や基本ルーチンもこの中にあります。 ImageFile.[ch] と ImageShowing.c と ImageDisp.[ch] は、単なる呼びだしルーチンで、本体は、別のファイルに存在しています。

Image.h Image.c	image 構造体と関数群
ImageFile.h ImageFile.c	画像ファイルの入出力
ImageShowing.c	画像を表示可能な型に変換する
ImageDisp.h ImageDisp.c	画像を X Window に表示する
Voxel.h Voxel.c	voxel 構造体と関数群
ColorImage.h	カラーの空間の変換
ImageCG.h	画像に単純な図形を描く

misc ディレクトリ

このディレクトリは、いろいろな資源にアクセスするためのライブラリのソースファイルがあります。 UNIX のシステムコールなどをもう少し使いやすくしたライブラリといってもいいでしょう。

comacros.h	共通マクロ集、プロトタイプ宣言用マクロ
debug.h debug.c	CIL のデバッグ用
filelib.h filelib.c	ファイル入出力
fileslib.h fileslib.c	複数のファイルを一つのファイルとして扱う
memlib.h memlib.c	1, 2, 3 次元配列などメモリ操作
optlib.h optlib.c	コマンドラインのオプション処理
shmemlib.h shmemlib.c	共有メモリ操作
socketlib.h socketlib.c	ソケットを用いた通信
strlib.h strlib.c	文字列の操作
timelib.h timelib.c	タイマの操作
typelib.h typelib.c	型識別子などデータベース化

Xcil ディレクトリ

このディレクトリは、 X Window に関係のあるライブラリのソースファイルが入っています。 Xcil ライブラリ、 **XImage** 構造体を使って X Window に画像を表示する関数群、 image、 **XImage** 構造体に簡単な図形を描く関数群などです。

Xcil ライブラリは、 XToolkit の機能を少なくして、もう少しプログラミングをしやすくしたライブラリです。

画像を表示する関数は、プレーン数と画素型ごとに分けて表示を行ないます。そのため、下のように画素型ごとのファイルに分かれています。そして、画素型ごとに物理的な意味が解釈されて表示されます。物理的な意味とは、カラー画像であるとか、濃淡画像であるとか、二値画像であるとか、ラベル画像であるといったことです。

簡単な図形を描く関数群は、 X Window のグラフィック機能を使わないで実現しています。しかし、文字列の表示は、 X Window のフォントを使用しているので、 Xcil の中に属しています。

Xcil.h Xcil_lib.c Xcil_misc.c	Xcil ライブラリ本体
Xcil_event.c Xcil_hdr.c Xcil_dummy.c	イベント関係
Xcil_confirm.c Xcil_panel.c	ウインドウ部品
Xcil_popup.c Xcil_scroll.c	
Xcil_string.c	文字列操作
Xcil_text.c Xcil_text_loc.c	テキスト操作
XImage.c XImage.h	画像を X Window に表示する
XImage1p.c XImage8p.c	プレーン数ごとの関数
XImage16p.c XImage24p.c	
bit1.c uchar.c uchar3.c	画素型ごとの関数
char.c long.c	
XImageCG.c XImageCG.h xfont.c	**XImage** 構造体に図形を描く
ImageCG.c ImageCG.h cglib.c cglib.h	image に図形を描く
imgfont.c fontlib.c fontlib.h font.c	

ImageFile ディレクトリ

このディレクトリは、画像ファイルを読み込むためのソースファイルが入っています。FImage.[ch] は、画像ファイルは読み込みのときに、フォーマットを自動的に識別して各種フォーマットの読み込みルーチンを呼び出します。書く込みのときは、指定されたフォーマットを呼び出します。

現在、サポートされているフォーマットは表に示されるファイルだけです。追加するときは、このディレクトの FImage.[ch] に宣言して、ソースを追加して **make** をすれば追加されます。

FImage.c FImage.h	画像ファイルの読み込み書き込み
c2d.c j4.c s3d.c xbm.c	各種フォーマットの
xwbitmap.c dib.c gif.c	読み込みと書き込み
pnm.c ps.c xwd.c	
tiff.c viff.c jpeg.c	

server ディレクトリ

このディレクトリは、共有メモリ画像と imagedisp の制御などを行なうサーバ及びクライアントのソースが入っています。また、 CIL で用いているプロトコルについてのドキュメントも入っています。

サーバと直接通信をして、メッセージを送るためのコマンド群もこのディレクトリにあります。もしサーバと通信をするプログラムを書くときには参考になるでしょう。また、このコマンド群は、サーバの動作がおかしきなったときに、正常に戻すためにも使用されます。

また、サーバは共有メモリの排他制御するのに使用しており、各個人で一つのサーバを起動することになります。そのため、一つのマシンに対して一つのサーバということになります。ネットワークを通しての共有メモリの管理はしていません。

Makefile Makefile.sun4 Makefile.sun5	メイクファイル
server.doc	プロトコルに関するドキュメント
cilserver.h cilserver.c cilclient.c	サーバ・クライアント制御用 ライブラリ関数
cilcp.c cills.c cilrm.c ciladd.c cilview.c	サーバ・クライアント制御用 コマンド
shmls.c shmrm.c	共有メモリの操作

clients ディレクトリ

このディレクトリは、 CIL を使って書かれたコマンド群のソースファイルが入っています。各コマンドのソースファイルは、ディレクトリの中に入っています。ほとんどのコマンドが CIL では必要なものとなっています。

imagedisp/	画像を表示する (server と通信する)
simpledisp/	画像を表示する
imageedit/	画像を編集する
imagergb/	RGB 色空間を表示する
voxeldisp/	voxel を表示する
dataplot/	テキスト数値データをプロットする
pipe/	画像処理ルーチン
util/	画像変換など便利なツール
disk/	モフォロジで使用する画像
test/	テストで使用した

3.1.3 拡張ディレクトリ

このディレクトリには、各種画像処理関数がまとめられています。画像型に固有な処理 (カラーモデル変換など) と複数の画像型に有効な処理 (メディアン・フィルタ、領域コピー、濃度値反転など) とに分けられています。もし、画像処理関数を作成したら追加して下さい。

BinaryImage ディレクトリ

このディレクトリには、二値画像専用の処理関数のソースファイルが入っています。二値画像処理のほとんどは、 Image に入っています。しかし、二値画像処理のルーチンはまだほとんどないので協力をお願いします。

GrayImage ディレクトリ

このディレクトリには、濃淡画像専用の処理関数のソースファイルが入っています。しかし、濃淡画像処理のほとんどは Image に入っています。まだ、このあたりのディレクトリは整理されていませんので、近いうちに整理をしたいと思います。

ColorImage ディレクトリ

このディレクトリには、カラー画像専用の処理関数のソースファイルが入っています。主に、カラーモデルの変換関数群が入っています。領域分割などもっと増やしたいと思っていますので協力をお願いします。

Image ディレクトリ

このディレクトリには、各種画像処理関数のソースファイルが入っています。このうちのほとんどが、二値画像と濃淡画像の処理となっています。フィルタ類はカラー画像もサポートしています。

Funcs.h	関数群宣言
type-check.c nosupport.c caller.c binddata.c copyarea.c separate2.c longtype.c fillin.c fillmoat.c	型ルックアップテーブル 特別な処理関数
thresholding.c	単純閾値処理
reverse.c liner.c histflat.c	濃度変換
median.c sobel.c average.c laplacian4.c laplacian8.c gaussian.c dgaussian.c dgaussabs.c	各種フィルタ
labeling.c	ラベリング
Morphology.h	モフォロジ関数群宣言
erosion.c dilation.c opening.c closing.c	モフォロジ演算
Filter.h	一般フィルタ宣言
filter.c filtering.c roundpix.c gauss_loc.c	各種フィルタ生成

3.2 コンパイルの仕方

CIL のプログラムを組んで、コンパイルする場合に、ヘッダファイルとライブラリ本体を参照します。 CILHOME/{include, lib} とは別に、利用者参照用のディレクトリがあります。利用者は、以下のディレクトリを実際に参照します。

```
/home/abe/common/include     ヘッダファイル
/home/abe/common/lib/sun4    SunOS4.* 用ライブラリ本体
/home/abe/common/lib/sun5    SunOS5.* ライブラリ本体
```

これらのディレクトリを、それぞれ CILINC, CILLIB で示すことにします。以下に、各ディレクトリ内のファイルの説明をして、実際にコンパイルを行なってみます。

3.2.1 include ディレクトリ

CILINC の中には、利用者が C プログラムの中から参照するヘッダファイルがあります。各ファイルは以下の通りです。

基本ヘッダファイル	
Image.h ImageFile.h ImageDisp.h ImageCG.h ColorImage.h Voxel.h	image 操作用関数群
基本ヘッダファイル misclib	
misc/	ディレクトリ
comacros.h	共通マクロ
optlib.h typelib.h	資源インターフェース
filelib.h fileslib.h	
memlib.h shmemlib.h	
strlib.h socketlib.h	
Xcil ライブラリ・ヘッダファイル	
Xcil/	ディレクトリ
XImage.h Xcil.h	X Window インターフェース
XImageCG.h ImageCG.h	
拡張ライブラリ・ヘッダファイル	
Image/	ディレクトリ
Funcs.h Filter.h	各種画像処理関数
Morphology.h	

3.2.2 lib ディレクトリ

CILLIB の中には、利用者がコンパイルのときに参照するライブラリ本体が入っています。各種ライブラリは以下のようになっています。

libcil.a	CIL 静的ライブラリ
libcil.sa.1.0	CIL 初期化済み共有ライブラリ
libcil.so.1.0	CIL 共有ライブラリ
libXcil.a	Xcil 静的ライブラリ
libXcil.sa.1.0	Xcil 初期化済み共有ライブラリ
libXcil.so.1.0	Xcil 共有ライブラリ
libcilext.a	cilext 静的ライブラリ
libcilext.so.1.0	cilext 共有ライブラリ

3.2.3 コンパイル方法

cc によるコンパイル方法を説明する前に、以下の環境変数を設定しておくこととします。これは、 SunOS4.* の設定です。

```
earth % setenv CILINC /home/abe/common/include
earth % setenv CILLIB /home/abe/common/lib/sun4
```

SunOS5.* では、ライブラリが異なるので、 CILLIB の設定は以下のようにします。

```
blue % setenv CILLIB /home/abe/common/lib/sun5
```

これで準備は完了しました。

CIL 基本ライブラリのみ使用プログラム

では、 sample.c をコンパイルしてみましょう。 -lcil で CIL 基本ライブラリをリンクします。

```
earth % cc sample.c -I$CILINC -L$CILLIB -lcil
```

これで、実行形式の a.out ができます。

CIL 基本ライブラリと数学ライブラリ使用プログラム

数学ライブラリ関数を使用している samplem.c の場合は以下のようにします。オプション -lm で数学ライブラリをリンクします。

```
earth % cc samplem.c -I$CILINC -L$CILLIB -lcil -lm
```

CIL 基本ライブラリ、 Xcil ライブラリ使用プログラム

Xcil ライブラリを使用している xsample.c の場合は以下のようにします。 Xcil は、 Xlib を使用するので、 -lX11 オプションでリンクします。

```
earth % cc xsample.c -I$CILINC -L$CILLIB -lcil -lXcil
-lX11
```

CIL 基本ライブラリ、 cilext ライブラリ使用プログラム

cilext ライブラリを使用している esample.c の場合は以下のようにします。 cilext は、数学ライブラリを使用するので、 -lm オプションでリンクします。

```
earth % cc esample.c -I$CILINC -L$CILLIB -lcil -lcilext
-lm
```

3.2.4 コンパイルのまとめ

コンパイルに関するディレクトリをまとめておきます。

表 3.1: コンパイルに関するディレクトリ

ヘッダファイル	/home/abe/common/include
SunOS4.* 用ライブラリ本体	/home/abe/common/lib/sun4
SunOS5.* 用ライブラリ本体	/home/abe/common/lib/sun5

オプションの組合せはいろいろ組合せは考えられますが、整理しておくと以下のようになります。

表 3.2: ライブラリとオプション

ライブラリ名	ファイル名	コンパイル・オプション
CIL 基本ライブラリ	libcil.*	-lcil
Xcil ライブラリ	libXcil.*	-lXcil -lX11
cilext ライブラリ	libcilext.*	-lcilext -lm

3.3 CIL の持つ環境変数

CIL には、ライブラリの動作を決定するための環境変数と、 CIL が参照するデータベースのディレクトリの環境変数が 5 つあります。これらの環境変数はすべて指定する必要はありませんが、これらを理解し指定することによって少し高度な CIL の機能が使えたり、作業効率が上がったりします。以下に詳しく説明します。

CIL_IMAGE_DIR	画像データベースのディレクトリ
CIL_RESOURCE_DIR	optlib が参照するリソースデータベース
CIL_VERBOSE	CIL のデバッグ用に使用
CIL_UNCOMPRESS	圧縮画像を読み込む時の展開コマンド
CIL_OPTION_pnmtops	**pnmtops** コマンドのオプション

3.3.1 環境変数 CIL_IMAGE_DIR

これは、画像を読み込むディレクトリを指定します。複数指定することもできます。 CIL は指定された画像ファイルを読み込むとき、まず最初に、そのファイル名のまま (カレントディレクトリの中を) 見にいきます。もし、ファイルがなければ、環境変数 CIL_IMAGE_DIR に指定されるディレクトリの順に探していきます。

これを設定しておけば、ファイル名の入力の手間を省くことができます。設定の例を以下に示しておきます。

```
setenv CIL_IMAGE_DIR "/home/abe/image:/home/abe/image/SIDBA"
```

3.3.2 環境変数 CIL_RESOURCE_DIR

これは、 optlib が参照するリソースデータベースを格納してあるディレクトリを指定します。 optlib を使って作成されたアプリケーションは、オプションの指定によるパラメータの受けとり以外に、リソースファイルによるパラメータの受けとりも行なうことができます。このリソースファイルを格納しておくディレクトリを指定します。

設定例を以下に示しておきます。このように、設定しておくと、 optlib を使って作成されたアプリケーションが起動すると、このディレクトリからリソースファイルを読み込み、このリソースファイルに設定されているパラメータを初期値として設定します。通常、よく指定するオプションを設定しておきます。

```
setenv CIL_RESOURCE_DIR "/home/abe/sugiyama/.CIL"
```

3.3.3 環境変数 CIL_VERBOSE

これは、 CIL の画像ファイル読み込みルーチンで参照される環境変数です。画像ファイルを読み込む機構は複雑で、この機構を知らない利用者は、自分の思っているファイルとは違うファイルを読み込む可能性があります。この環境変数が設定されていると、 CIL_IMAGE_DIR に記述されているディレクトリ・リストの、どのディレクトリからファイルが読まれたかを表示します。

設定方法は、以下の通りです。

```
setenv CIL_VERBOSE ""
```

3.3.4 環境変数 CIL_UNCOMPRESS

これは、何か別のコマンドで圧縮された画像ファイルを読み込む時の、展開用のコマンドを指定します。設定例は以下の通りです。しかし、設定しなくても、読み込みのときには拡張子で **uncompress** か **gunzip** を選択します。

```
setenv CIL_UNCOMPRESS "gunzip"
```

3.3.5 環境変数 CIL_OPTION_pnmtops

これは、 PNM フォーマットから PS フォーマットへ変換する **pnmtops** コマンドのオプションを指定します。実際に、 **pnmtops** は CIL 内部で、画像を PostScript ファイルに変換するときに呼び出される pbmplus のコマンドです。 **pnmtops** はバージョンによってオプションが違いますし、自分の好きなオプションを設定できるように、この環境変数を作りました。

設定例は以下の通りです。スケールは 1 倍にし、画像の大きさによって 90 回転する機能を止めるオプションです。

```
setenv CIL_OPTION_pnmtops "-scale 1 -noturn"
```

第 4 章

CIL プログラミング入門

この章では、 CIL の基本的な動作を理解し基本的なプログラムが組めるようになるまでを説明します。ここで対象とするものは、 image 構造体と Image 関数群です。特に、 image のライフ・サイクルについて理解し、 CIL でよく使う関数を覚えましょう。

- image ライフ・サイクルと Image 関数群
- 画像ファイルの読み込みと書き込み
- 画像の簡単な加工

また、入力を簡単にするためと OS の違いの便宜をはかるために、ヘッダ・ファイルとライブラリのディレクトリの環境変数を設定しておきましょう。

```
earth % setenv CILINC /home/abe/common/include
earth % setenv CILLIB /home/abe/common/lib/sun4
```

Solaris で行なう場合は、以下のように設定しておきます。

```
blue % setenv CILLIB /home/abe/common/lib/sun5
```

4.1 image ライフ・サイクル

CIL で最も重要な image は、決まったライフ・サイクルを持っています。ライフ・サイクルというのは、「人生の移り変わり」といった意味です。ここでは、image の持つライフ・サイクルを説明します。

4.1.1 ライフ・サイクル表示プログラム

まず最初に、基本的な image のライフ・サイクルを表示するプログラムを譜 4.1に紹介します。ファイル名は cycle.c としておきましょう。実際に何が行なわれているかは、次の節で詳細に説明します。とりあえず動かしてみましょう。

```
 1: #include "Image.h"
 2: #include <stdio.h>
 3:
 4: void main( argc, argv )
 5:      int argc;
 6:      char *argv[];
 7: {
 8:   image foo;
 9:   Image.print( foo );
10:
11:   foo = Image.create( "foo" );
12:   Image.print( foo );
13:
14:   Image.make( foo, UChar, 256, 256 );
15:   Image.print( foo );
16:
17:   Image.free( foo );
18:   Image.print( foo );
19:
20:   Image.destroy( foo );
21:   Image.print( foo );
22: }
```

譜 4.1: 基本的ライフ・サイクルの表示

コンパイルは以下のようにすればできます。

```
earth % cc -I$CILINC -L$CILLIB cycle.c -lcil
```

これで、 a.out ができます。実行してみると、次のような結果を出力します。

```
earth % a.out

 1: ================================
 2:    THIS IMAGE IS NOT CREATED.
 3: ================================
 4: ================================
 5:              Image
 6: ================================
 7:     name  : foo
 8:     type  : None
 9:     xsize : 0
10:     ysize : 0
11:     data  : 0x00000000
12: ================================
13: ================================
14:              Image
15: ================================
16:     name  : foo
17:     type  : UChar
18:     xsize : 256
19:     ysize : 256
20:     data  : 0x00006200
21: ================================
22: ================================
23:              Image
24: ================================
25:     name  : foo
26:     type  : None
27:     xsize : 0
28:     ysize : 0
29:     data  : 0x00000000
30: ================================
31: ================================
32:    THIS IMAGE WAS DESTROIED.
33: ================================
```

· 1 **行目〜** 3 **行目**

譜 4.1の
```
 8:   image foo;
 9:   Image.print( foo );
```

最初の Image.print の出力結果です。この状態では、 image の foo は何も操作されていません。つまり、単に C 言語の変数として宣言されただけです。「この画像は生成されていません」というメッセージを出します。

· 4 行目〜 12 行目

譜 4.1の
```
11:   foo = Image.create( "foo" );
12:   Image.print( foo );
```

foo は、 Image.create によって生成された状態にあります。 xsize, ysize, 画像データ data には 0 が入っています。つまり、画像のデータ領域は確保されていない状態です。 type は、 None になっていますが、これは「画素型はない」ということです。つまり、 Image.create は、属性を入れる箱を生成するだけなのです。

· 13 行目〜 21 行目

譜 4.1の
```
14:   Image.make( foo, UChar, 256, 256 );
15:   Image.print( foo );
```

foo は、 Image.make によって画像のデータ領域が確保されている状態です。画素型 type は UChar, xsize は 256, ysize は 256, 画像データ data には 0x00006200 が入っています。これは、メモリ空間が割り当てられている状態です。 Image.make は、画像のデータ領域を確保する命令なのです。

また、 Image.make は、間接的に別の関数から呼ばれることもあります。明示的に Image.make を呼ぶことだけが、画像のデータ領域を確保する方法ではありません。

ここにきてようやく画像は操作する対象になります。本来ならば、この後にいろいろな処理が行なわれることになります。

· 22 行目〜 30 行目

譜 4.1の
```
17:   Image.free( foo );
18:   Image.print( foo );
```

foo は、 Image.free によって、画像のデータ領域が解放された状態になりました。この状態では、 Image.create したすぐ後と全く同じ状態です。また、 Image.free は、メモリ空間を効率的に使う場合を除いては、通常明示的に呼び出されるときはありません。それは、次の Image.destory の中で Image.free が呼び出されるからです。

· 31 行目〜 33 行目

譜 4.1の
```
20:    Image.destroy( foo );
21:    Image.print( foo );
```

foo は、 Image.destroy によって、すべてメモリ空間から解放されてしまいました。つまり、属性を入れておく領域も解放されています。この状態は、非常に危険な状態です。必ずしも、 foo に null が入っているわけではないからです。 Image.destroy の後に表示できたのは、偶然、「破壊された画像」であると判断できたからです。

4.1.2 ライフ・サイクルの詳細

これまで、プログラムの属性の表示と文章で説明してきただけでしたが、もう少し詳しく見てみましょう。一つのプログラムの中で、 image のライフ・サイクルを考えると、基本的に図 4.1のような決まりきったライフ・サイクルが見えてきます。

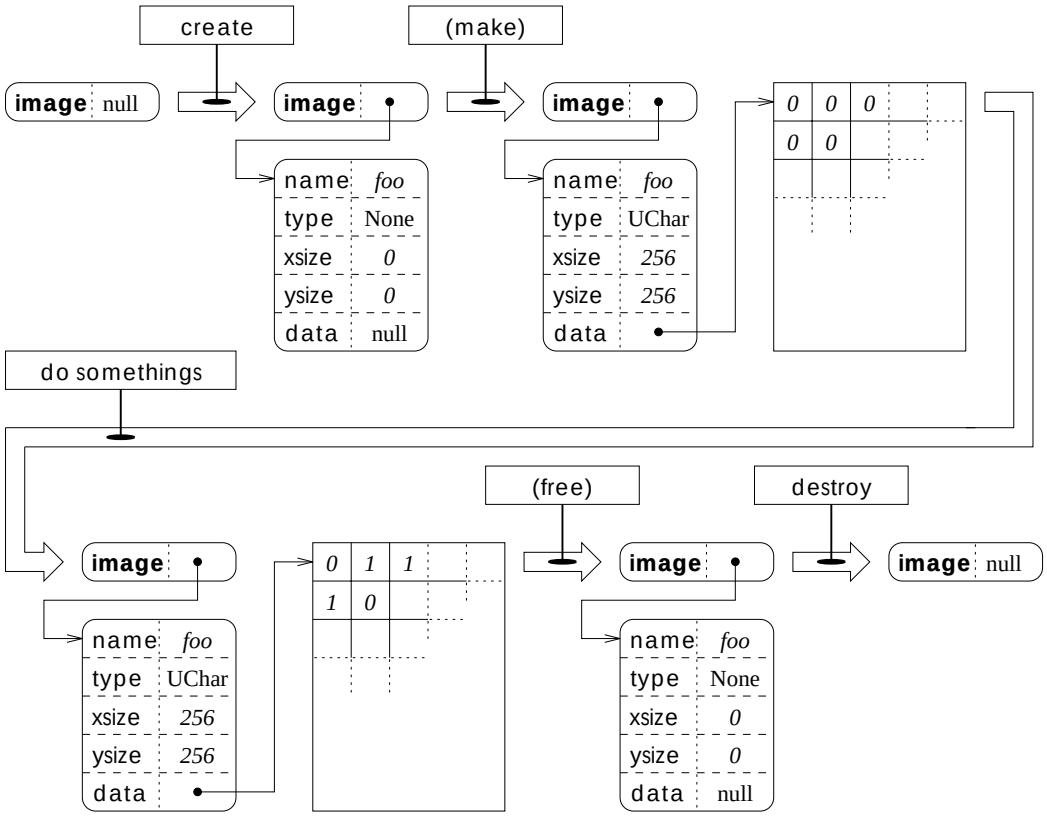


図 4.1: image のライフ・サイクル

まず最初に、 Image.create によって画像が生成され、**画像の属性を入れる領域が確保**されます。次に、間接的もしくは直接的に Image.make が呼ばれ、**画像のデータ領域が確保**されます。一端、この状態になると画像に対していろいろな作業ができます。作業などがすべて終ってから Image.free を呼び、画像のデータ領域を解放します。最後に、 Image.destroy が呼ばれ、画像の属性を入れる領域が解放されます。

Image.make が間接的に呼び出されるというのは、明示的に呼び出してデータ領

域の確保をしなくても、必要なら各関数内部で Image.make を呼び出しデータ領域の確保を行なうからです。例えば、 Image.load では、ファイルから属性を読みとった時点で Image.make を呼び出します。

実際のところ、 Image.free は呼ぶ必要はありません。それは、 Image.destroy の中で Image.free を呼んでいるからです。これは、メモリを効率的に使うときに使用します。

4.2 属性参照

image 構造体はカプセル化されており、データへのアクセスはすべて、 Image 関数群を用いなければなりません。しかし、実行効率を考えると関数を呼び出すのはあまりよくありません。属性参照に関しては、 Image 関数群以外に属性参照マクロを用意しています。以下にこれらを説明します。

4.2.1 Image 関数群による属性参照

Image 関数群には、表 4.1の関数が用意されています。引数はすべて image 型の一つです。上の 5 つは、画像の属性そのものを返し、下の 3 つは、画像の属性から計算をして返します。この関数を使ったプログラム例を譜 4.2に示します。

表 4.1: Image の属性参照関数

関数名	説明
Image.name	画像の名前
Image.type	画素型 (型識別子)
Image.xsize	x 画素数
Image.ysize	y 画素数
Image.data	二次元配列へのポインタ
Image.area	全画素数
Image.byte	画像データの全バイト数
Image.raster	ラスタスキャンへのポインタ

```
 1: void my_image_print
 2:   _P1 (( image, info ))
 3: {
 4:   printf( "name = \"%s\"\n", Image.name( info ) );
 5:   printf( "type = %d\n", Image.type( info ) );
 6:   printf( "xsize = %d\n", Image.xsize( info ) );
 7:   printf( "ysize = %d\n", Image.ysize( info ) );
 8:   printf( "data = 0x%08x\n", Image.data( info ) );
 9:   printf( "area = %d\n", Image.area( info ) );
10:   printf( "byte = %d\n", Image.byte( info ) );
11:   printf( "raster = 0x%08x\n", Image.raster( info
) );
12: }
```

譜 4.2: 属性表示関数

4.2.2 マクロによる属性参照

表 4.2は、先ほどの節で説明した関数のマクロ版です。返す値はすべて Image 関数群と同じです。 type は実際の画素型で、 C 言語の型名を与えます。これは、マクロ内部でキャストして型変換を行なうためです。

表 4.2: 属性参照マクロ

マクロ名	説明
__NAME (image)	画像の名前
__TYPE (image)	画素型 (型識別子)
__XSIZE (image)	x 画素数
__YSIZE (image)	y 画素数
__DATA (image, type)	二次元配列へのポインタ
__AREA (image)	全画素数
__BYTE (image)	画像データの全バイト数
__RASTER (image, type)	ラスタスキャンへのポインタ

4.3 Image によるファイルの読み書き

通常、画像処理は、画像ファイルから画像を読み込み、何か処理をして結果を画像ファイルに書き込みます。この節では、簡単な画像処理をするプログラムを紹介して、画像ファイルの入出力の仕方を説明します。

4.3.1 画像処理プログラムの典型

図 4.2に示すように、大抵の画像処理プログラムは、はじめに原画像を画像ファイルから入力します。そして、原画像に何か処理をして、その結果を画像ファイルとして出力します。入力画像や出力画像は処理によって複数になったりします。

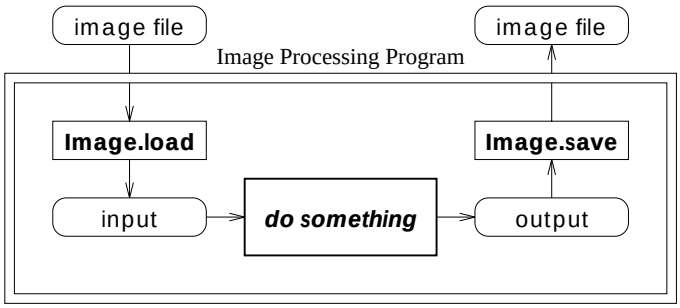


図 4.2: 典型的な画像処理の全体のデータの流れ

Image には、画像ファイルを読み込む関数の Image.load と、画像ファイルを書き込む関数の Image.save があります。これらの二つを使って画像ファイルの入出力を行ないます。

また、画像を生成して画像ファイルを読み込む作業は頻繁にあるので、画像の生成と画像ファイルの読み込みを同時に行なう関数 Image.createFromFilename があります (図 4.3)。

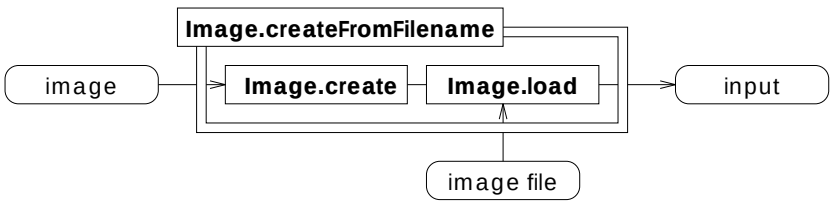


図 4.3: Image.createFromFilename の説明

4.3.2 実例：二値画像反転処理

ここでは、実際の実例を見ながら画像の入出力を説明します。画像処理の例は、二値画像反転処理です。譜 4.3にプログラムを示します。ファイル名 breverse.c のコンパイルを以下のように行なうと、 a.out という実行形式ができます。そして、実行すると toola1.j4 の反転した画像が rtoola1.j4 に得られます。

```
earth % cc -I$CILINC -L$CILLIB breverse.c -lcil
earth % setenv CIL_IMAGE_DIR /home/abe/imgae/tools
earth % a.out toola1.j4 rtoola1.j4
```

```
 1: #include "Image.h"
 2:
 3: void binary__reverse
 4:   _P2 (( image, output ),
 5:        ( image, input  ))
 6: {
 7:   int i, n;
 8:   bit1 *i_ptr, *o_ptr;
 9:
10:   Image.make(output, Bit1, __XSIZE(input), __YSIZE(input));
11:   n = __AREA( input );
12:   i_ptr = __RASTER( input , bit1 );
13:   o_ptr = __RASTER( output, bit1 );
14:   for ( i = 0; i < n; i++, i_ptr++, o_ptr++ )
15:     *o_ptr = ( *i_ptr ) ? 0 : 1;
16: }
17:
18: void main( argc, argv )
19:      int argc;
20:      char *argv[];
21: {
22:   image input, output;
23:
24:   input = Image.createFromFilename("input", argv[1]);
25:   output = Image.create( "output" );
26:   binary__reverse( output, input );
27:   Image.save( output, argv[ 2 ], "Reverse Image" );
28:   Image.destroy( input );
29:   Image.destroy( output );
30: }
```

譜 4.3: 二値画像反転処理

24行目: Image.createFromFilename で、画像を生成して画像ファイルを読み込みます。画像ファイル名は、コマンドラインの第 1 引数で指定されます (`argv[1]`)。
27行目: Image.save で、演算の結果を画像ファイルに書き込みます。出力の画像ファイル名は、コマンドラインの第 2 引数で指定されます (`argv[2]`)。

エラーチェックについて

ここで示したプログラムは、エラーチェックは一切行なっていません。みなさんが、実際に何かコマンドを作成する場合は、次のようなエラーチェックを行なって下さい。

(1) ファイル名が入力されたかどうかのチェック
(2) 画像ファイルが読み込めたかどうかのチェック
(3) 画像の画素型が正しいかどうかのチェック

また、オプションが必要ない場合でも、なんらかの引数をとる場合には optlib を使用することを勧めます。それは、エラーチェックを軽減させることができるからです。 optlib を使えば (1) のエラーチェックは必要なくなります。 optlib は、引数が入力されていない場合ユーザに入力を要求するからです。 optlib を使用したメインプログラムの一部を、譜 4.4に紹介しておきます。 optlib については、次の章で説明します。

```
 1: static char *options[] = {
 2:   "Reverse Binary Image.",
 3:   "input:*:1:(image) name of the input binary image",
 4:   "output:*:1:(image) name of the output binary image",
 5: };
 6:   ......
 7:   image input, output;
 8:   char *input_name, *output_name;
 9:
10:   OPTION_SETUP( options );
11:
12:   input_name  = optvalue( "input" );
13:   output_name = optvalue( "output" );
14:   input  = Image.createFromFilename( "input", input_name
);
15:   output = Image.create( "output" );
16:
17:   binary__reverse( output, input );
18:   Image.save( output, output_name, "Reverse Image" );
19:   ......
```

譜 4.4: optlib を使ったメイン・プログラム

4.4 今まで出てきた関数の説明

4.4.1 Image.create

```
image Image.create
  P1 (( char *, name )) /* 画像名 */
```

引数は $name$で、返り値は image です。 Image.create は、 $name$を持つ画像の**属性の領域**を確保します。この関数は、通常、 Image.destroy と**対**に使われます。

4.4.2 Image.destroy

```
void Image.destroy
  P1 (( image, self )) /* 画像 */
```

引数は、 image です。 Image.destroy は、画像 $self$ の持つ領域をすべて解放します。内部では、 Image.free が呼ばれ、画像のデータ領域が解放された後、画像の属性の領域を解放します。

4.4.3 Image.make

```
void Image.make
  P4 (( image, self  ), /* 画像 */
      ( long , type  ), /* 型識別子 */
      ( long , xsize ), /* x画素数 */
      ( long , ysize )) /* y画素数 */
```

引数は、 image、型識別子 $type$、 x, y 画素数 $xsize, ysize$の 4 つです。 Image.make は、画像の**データ領域**を確保し初期化します。表 4.3に、登録されている型識別子を示します。実際の C 言語での型は、すべて小文字にしたものです。例えば、型識別子 UChar の型は、 uchar です。

PackedBit1 は実際には UChar ですが、 1byte の各 8bit を 8 画素分として扱います。 Bit1 は、実際には UChar で、有効ビットが 1bit のものです。 Bit4 は有効ビットが 4bit のものです。他の型の後ろの数字は画素の次元数です。例えば、型識別子の UChar3 の型は uchar3 で、次のように型定義されています。

表 4.3: 登録されている型識別子

PackedBit1		Bit1	Bit4		
Char	Char2	Char3	UChar	UChar2	UChar3
Short	Short2	Short3	UShort	UShort2	UShort3
Int	Int2	Int3	UInt	UInt2	UInt3
Long	Long2	Long3	ULong	ULong2	ULong3
Float	Float2	Float3	Double	Double2	Double3

```
typedef struct { uchar at[3]; } uchar3;
```

4.4.4 Image.free

```
void Image.free
  P1 (( image, self )) /* 画像 */
```

引数は、 image です。 Image.free は、画像 *self* のもつ画像データ領域だけを解放します。メモリを効率良く利用する目的以外では使用しません。画像を、 Image.destroy する場合は、呼ぶ必要はありません。内部で Image.free が呼び出されます。

4.4.5 Image.createFromFilename

```
image Image.createFromFilename
  P2 (( char *, name     ), /* 画像名 */
      ( char *, filename )) /* 画像ファイル名 */
```

引数は、 *name* と入力画像ファイル名 *filename*、返り値は image、もしくは、0 です。 Image.createFromFilename は、 image を生成した後、指定されたファイル名 *filename* から画像ファイルを読み込みます。もし、画像ファイルの読み込みに失敗した場合は、画像を破壊して 0 を返します。

4.4.6 Image.load

```
long Image.load
  P2 (( image , self     ), /* 画像 */
      ( char *, filename )) /* 画像ファイル名 */
```

引数は、 image と入力画像ファイル名 *filename*、返り値は読み込みに成功したかどうかです。 Image.load は、指定された画像ファイル名 *filename* を読み込みます。画像の読み込みに成功した場合は 1 を返し、失敗した場合は 0 を返します。

4.4.7 Image.save

```
long Image.save
  P3 (( image , self     ), /* 画像 */
      ( char *, filename ), /* 画像ファイル名 */
      ( char *, comment  )) /* コメント */
```

引数は、 image と出力画像ファイル名 *filename* とコメント *comment*で、返り値は書き込みに成功したかどうかです。 Image.save は、指定された画像ファイル名 *filename* に画像を書き込みます。画像の書き込みに成功した場合は 1 を返し、失敗した場合は 0 を返します。コメント *comment*は、もし画像フォーマットにコメントのフィールドがある場合は、コメントを書き込みますが、もしなければコメントは無視されます。画像フォーマットの変更については、次の章で説明します。

4.4.8 Image.print

```
void Image.print
  P1 (( image, self )) /* 画像 */
```

引数は、 image です。 Image.print は、画像の持つ属性を直接表示します。

第 5 章

CIL プログラミング基本

この章では、 CIL のプログラミングの基本的事項を説明します。プログラミング・スタイルは厳しくいいませんが、 CIL には**推奨スタイル**があります。この推奨スタイルを使用することによって、いろいろな恩恵を受けることができます。

ここでは、以下のことについて説明します。

- 関数記述のスタイル
- 関数作成のスタイル
- コマンド作成のスタイル
- オプションの処理

5.1 関数宣言の方法

関数の宣言方法を工夫することよって、K&R C と ANSI C の違いを吸収することができます。

5.1.1 K&R C と ANSI C の違い

K&R C と ANSI C によって、関数の宣言の仕方が異なります (譜 5.1)。K&R C は古い文法で、ANSI C は、現在の C 言語の標準的な規格です。ANSI C の強力な機能に、関数の引数の型チェック機能があります。この引数の型チェックは、プログラム作成時におけるバグを少なくすることができます。

```
/* K&R C 関数の前宣言 */
void image_thresholding();

/* K&R C 関数本体の宣言 */
void image_thresholding(output,input,threshold)
     image output;
     image input;
     double threshold;
{
  codes
}

/* ANSI C 関数の前宣言 */
void image_thresholding(image output,image input,double threshold);

/* ANSI C 関数本体の宣言 */
void image_thresholding(image output,image input,double threshold)
{
  codes
}
```

譜 5.1: K&R C と ANSI C の関数宣言の違い

しかし、まだ、K&R C と ANSI C が混在しています。K&R C の文法で書いた場合は、ANSI C と K&R C の使えるマシンでコンパイルできますが、ANSI C の型チェック機能は使えません。ANSI C の文法で書いた場合は、ANSI C の使えるマシンでしかコンパイルできません。しかし、両方のマシンでコンパイルできるようにするには、すべて K&R C で書くか、ANSI C と K&R C のときで場合分けをして二つの宣言を書くかのどちらかです。

5.1.2 CIL による関数宣言の方法

CIL では、以上のような問題を解決するために、関数宣言用のマクロを用意しました。例えば、先ほどの宣言は、譜 5.2のようになります。

```
/* CIL による 関数の前宣言 */
void image_thresholding
  P3 (( image , output    ),
      ( image , input     ),
      ( double, threshold ))

/* CIL による 関数本体の宣言 */
void image_thresholding
  _P3 (( image , output    ),
       ( image , input     ),
       ( double, threshold ))
{
  codes
}
```

譜 5.2: CIL による関数宣言

マクロは、 P#n, _P#n の二つで、 #n は、引数の数が入ります。前宣言と本体宣言とでは、マクロの前に ‘_’ (アンダーバー) が入るかどうかの違いです。通常、前宣言を書いてからコピーしてアンダーバーをつけます。このマクロは、次のような規則で使用します。

関数前宣言マクロ P#n

```
<returen-type> <function-name>
   P#n (( <type>1, <arg-name>1 ),
        ( <type>2, <arg-name>2 ),
         ......
        ( <type>n, <arg-name>n ))
```

関数本体宣言マクロ _P#n

```
<returen-type> <function-name>
   _P#n (( <type>1, <arg-name>1 ),
         ( <type>2, <arg-name>2 ),
          ......
         ( <type>n, <arg-name>n ))
{
  codes
}
```

5.2 画像処理の関数

画像処理の関数を書くときに、引数の順番などが統一されていると、他の人が見るときに理解しやすくなります。この節では、画像処理の関数を書くにあたっての、 CIL の推奨スタイルを説明します。

5.2.1 一般的な画像処理関数

一般的な画像処理関数を考えたとき、図 5.1のようになります[1]。入力画像 input に対して、何か処理した結果を出力画像 output に出力します。実際に、入力や出力は画像だけでなく、何か値かも知れませんし、一次元配列で表されたテーブルかも知れません。基本的に入力か出力のどちらかは画像です。

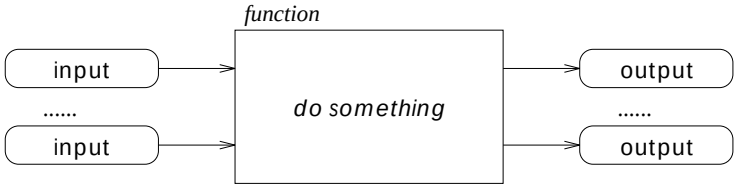


図 5.1: 一般的な画像処理関数

5.2.2 CIL による画像処理関数のスタイル

CIL では、画像処理に関する関数の定義のスタイルを統一する意味で、推奨スタイルを規定しています。これは、主に関数の**引数の順番**などを決めています。また、入力画像と出力画像に対して、**同じ画像を指定できるかどうか**なども重要なポイントです。さらに、入力画像や出力画像の**画素型**なども処理の流れを考えると重要です。以下に順に説明します。

引数の順序

CIL の画像処理関数の引数は、基本的に、以下のような順番になっています。感覚的に、入力が先にして出力を後にした方がいいという意見もありますが、変化する対象を先に指定するようにして、変化しないものは後に並べるようにして

[1]画像処理関数に関わらず関数というものは、入力に対して出力を返します。

います。このようにすると、**作用を受ける方**と**作用をする方**とでうまく分けることができます。

```
void <procedure-name>（ 出力，入力，パラメータ ）
```

入力画像と出力画像

出力画像は、関数内部で画像を make すべきです。このようにして、利用者はあらかじめ make しておく必要はなくなります。また、入力画像と出力画像の指定に同じ画像を指定した場合、それらの処理をしていないと副作用が生じてしまいます。CIL では、基本的に、入力画像と出力画像に同じ画像を指定しても動作を保証することにしています。

これらの特別な処理は、画像の処理する内容によって違ってきますが、簡単な実現方法を譜 5.3に示します。

```
 1: void <proc-name>
 2:   _P2 (( image, output ),
 3:        ( image, input  ))
 4: {
 5:   image tmp_output;
 6:
 7:   tmp_output = Image.createMake( "<proc-name>:output",
 8:                                   <type>, <xsize>, <ysize>
);
 9:
10:   do something for tmp_output
11:
12:   Image.copy( output, tmp_output );
13:   Image.destroy( tmp_output );
14: }
```

譜 5.3: 入力と出力が同じでも OK

画素型

入力画像の画素型をどれだけ受け入れるかは重要な問題です。また、出力画像の画素型は入力画像の画素型によって変化するかも知れませんし、常に一定かも知れません。これは、すべて関数を作る人の判断に委ねられます。よって、関数を作る人は、これらのことを明確にしなければなりません。関数のコメント、場合によっては関数名の中に、必ずそれらの情報を入れるようにしましょう。

例を譜 5.4に示しておきます。関数の名前とコメントの中に、画素型 uchar が入っています。個人で使用する場合には、関数の中には入れる必要はありません。

```
void image_threshold_uchar
  P3 (( image , output    ), /* 出力画像 (image.bit1) */
      ( image , input     ), /* 入力画像 (image.uchar)
*/

      ( double, threshold )) /* 閾値 */
```

譜 5.4: 画素型に関するコメントの例

5.3 コマンドの作成

何かコマンドを作成する場合は、自分一人だけしか使わないコマンドはないと思って作って下さい。誰かが**同じような問題**につき当たって、**同じようなコマンド**を作ることになる場合が多々にあります。このようなことは、毎年、何度でも起きます。これは、ぜひとも減らしたいところです。

また、作る人によってオプションの指定の仕方や振舞いが違ったりします。これも使う側から見れば面倒なことです。作る側から見れば、使う人のことを考えてオプションの処理を作るとなるとかなり面倒です。

このようなことから、大量に、埋もれた (他の人には使えない) 個人専用のコマンド群が発生します。はっきりいって、これは大きな財産に**成り得る**卵です。一歩間違えればゴミ同然になってしまいますし、使いようによっては宝になります。

CIL では、これらのことを考えて簡単な仕組みでオプション処理を実現できるライブラリ optlib を提供しています。 optlib は、ユーザとのインターフェースを統一するともに、プログラマの負担を軽減するライブラリです。また、拡張性に富んだコマンドが作成できます。以下に簡単に特徴をまとめておきます。

- オプションの追加／削除が簡単
- ユーザの統一的なインターフェースを提供
- プログラマの労力を軽減
  - Help と Usage の自動整形機能
  - 入力問い合わせ機能

5.3.1 optlib の基本的な考え方

optlib は、ユーザとアプリケーション (コマンド) との**パラメータの受渡し**に関するインターフェースを提供するライブラリです (図 5.2)。 optlib は、ユーザから見た場合とプログラマから見た場合とでは、かなり位置づけが違います。

optlib は、ユーザにパラメータを設定するためのインターフェースを提供します。 optlib がユーザに提供するパラメータの受渡しの方法は、次のようなものがあります。

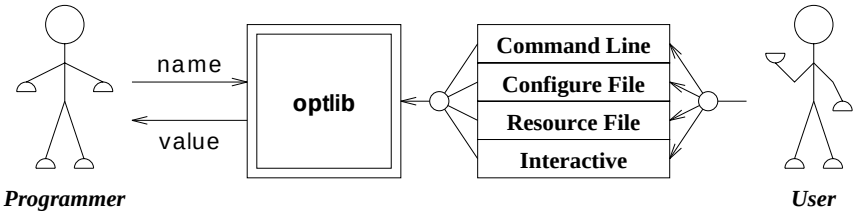


図 5.2: optlib の位置付け

- **コマンドラインによる受渡し**

  もっとも頻繁に使用される方法です。コマンドを実行するときに、同一行にパラメータを指定します。

- **コンフィグ・ファイルによる受渡し**

  コマンド実行時にコンフィグ・ファイルを指定して、パラメータを指定します。コンフィグ・ファイルの中には、パラメータが記述してあります。

- **リソース・ファイルによる受渡し**

  コマンドを実行すると無条件でこのリソース・ファイル読み込みます。このリソース・ファイルによってパラメータを指定します。

- **対話的な受渡し**

  実行すると、一つ一つパラメータをユーザに聞いてきて値を要求します。この対話的な入力によってパラメータを指定します。

optlib は、プログラマに対してはパラメータの受渡しをする関数を提供するだけです。 optlib を使えば、受渡し方法についての機能は自動的に提供されるので、プログラマはこのようなことは考える必要はありません。プログラマからは、自分で定義したパラメータ名を使ってパラメータを得ます。

また、 optlib は、パラメータの値すべてを文字列として扱っていますので、適当な変換を行なう必要があります。 optlib におけるコマンドラインの処理の方法は一括変換型です。

5.3.2 optlib の使い方

ここでは、 optlib の**プログラマ側**の使い方を説明します。 optlib では、プログラムの最初にオプション定義と初期化を行なえば、あとは、定義したオプション名を使ってパラメータ値にアクセスできます。

optlib を使用した例を譜 5.5に示しておきます。プログラムとしては完全ですがエラー処理は行なっていないので、注意して下さい。

· 5 **行目〜** 11 **行目**

オプションの定義を行なっています。オプションは文字列の配列で定義されます。各行の文字列がオプション定義で、各オプションの属性値は、‘:’(コロン) によって区切られています。

各属性の文法と意味は、以下の通りになっています。

名前: 指示:[引数の数:][初期値:]*[(型名)] 説明

· **名前**は、プログラム内部で参照されるときの文字列です。
· **指示**は、コマンドラインでユーザが実際にオプションを指定するときの文字列です。**指示**を必要としない場合は‘*’(アスタリスク) を指定します。これは、オプションでない引数を定義したオプションの順番に入れていきます。
· **引数の数**は、**指示**のあとに続く引数の数です。引数がない場合は指定する必要はありません。
· **初期値**は、オプションが省略されたときの初期値です。通常、引数の数より多く指定することはできません。初期値を決めることができない場合や初期値を持たない場合は、指定する必要はありません。
· **型名**は引数がある場合に( )で括って指定します。
· また、 5 行目のように、コロンで区切られていない行は、ヘルプ表示のときに表示します。

· 21 **行目**

オプションの初期設定を行なっています。 argv, argc はマクロ OPTION_SETUP によって自動的に指定されます。これを実行した後は、 optlib の関数群が使えるようになります。

· 23 **行目**，34 **行目**

コマンドラインに引数が一つもない場合は、optusage によって Usage を表示します。次に、optspecified によって "help"オプションが指定されているかどうか調べて、指定されていれば optmanual によって Help を表示します。

· 27 **行目〜** 29 **行目**

各パラメータの値を変数に代入しています。optvalue は、オプションの値を文字列のポインタで返します。もし、指定されていない場合は初期値を返します。初期値も指定されていない場合はユーザに値の入力を要求します。optvalueint は、内部で optvalue を呼んでから整数に変換してその値を返します。

```
 1: #include "Image.h"
 2: #include "misc/optlib.h"
 3: #include "Image/Funcs.h"
 4:
 5: char *option[] = {/* オプション定義 */
 6:   "The thresholding of an uchar image.",
 7:   "input:*:1:(image) name of the input uchar image",
 8:   "output:*:1:(image) name of the output bit1 image",
 9:   "threshold:-t:1:128:(uchar) value of the thresholding",
10:   "help:-h:print help messages",
11: };
12:
13: void main(argc,argv)
14:      int argc;
15:      char **argv;
16: {
17:   image input, output;
18:   char *input_name, *output_name;
19:   long threshold;
20:
21:   OPTION_SETUP( option ); /* 初期設定 */
22:
23:   if ( argc == 1 ) optusage( 1 );
24:   if ( optspecified( "help" ) ) optmanual( 1 );
25:
26:   /* パラメータ値の設定 */
27:   input_name  = optvalue( "input" );
28:   output_name = optvalue( "output");
29:   threshold   = optvalueint( "threshold" );
30:
31:   input = Image.createFromFilename("input", input_name);
32:   output = Image.create( "output");
33:   image__thresholding( output, input, threshold );
34:   Image.save( output, output_name, "thresholding" );
35:   Image.destroy( input );
36:   Image.destroy( output );
37: }
```

譜 5.5: optlib の例

5.4 まとめ

CIL には推奨スタイルがいくつかあります。これを守ることによって、いろいろな恩恵を受けることができます。また、 optlib を使うには、**オプション定義**と**初期設定**をまず最初に行ないます。そして、オプション定義で定義したオプション名を使って、パラメータの値を参照していきます。以下に、これらをまとめておきます。

5.4.1 関数定義

```
/* 関数前宣言 */
<returen-type> <function-name>
   P#n (( <type>1, <arg-name>1 ),
        ( <type>2, <arg-name>2 ),
         ......
        ( <type>n, <arg-name>n ))

/* 関数本体宣言 */
<returen-type> <function-name>
   _P#n (( <type>1, <arg-name>1 ),
         ( <type>2, <arg-name>2 ),
          ......
         ( <type>n, <arg-name>n ))
{
  codes
}
```

関数の定義を行なうためのマクロ P#n と _P#n は、 K&R C と ANSI C に応じて引数を展開するためのマクロです。 #n は、引数の数を示しており、マクロの引数と実際の引数は一致していなければなりません。また、画像処理関数の引数は、**出力画像、入力画像、パラメータ**の順番で統一します。

5.4.2 オプション定義

```
static char *option[] = {
  "コメント",
  "名前: 指示:[引数の数:][初期値:]*[(型名)] 説明",
  .....,
};
```

オプションの定義は、文字列の一次元配列で構成されます。各行は一つのオプションもしくはコメントに対応しています。一つのオプション定義は、**名前**と**指示**と**説明**を必ず属性として持ちます。また、オプションで引数を持つものは、**引数の数**と**初期値**と**型名**を指定することができます。各属性は、基本的に‘:’(コロン) で区切られています。また、マニュアルを表示する場合の制御用の文字がありますが、ここでは説明しません。

名前

名前は、プログラム内部でオプションを指定するときに使用するものです。各オプションは、ユニークでなければなりません。

指示

指示は、ユーザがコマンドラインでオプションを指定するときに使用します。これもユニークでなければなりません。もし、**指示**を持たないオプションであるならば、つまり、値をコマンドラインで直接指定するパラメータは、‘*’(アスタリスク) を指定します。この指定によるパラメータの順番は、定義した順番になります。

引数の数

引数の数には、コマンドライン上で**指示**のあとに、引数を指定するオプションの引数の数を指定します。引数を持たない場合は、何も指定する必要はありません。また、‘*’(アスタリスク) を指定することによって、可変引数も設定できます。コマンドラインでの指定における、可変引数の明示的な終了コード (EOL, end of list) は‘--’(マイナスが二つ) です。

初期値

初期値は、コマンドラインで指定されなかったオプションに対するパラメータの初期値を指定します。もし、初期値を持たない場合は指定する必要はありません。この初期値の指定の数は、引数の数を越えてはいけません。

型名、説明

型名は、ユーザに何を入力させるのかを明確にすることができます。**説明**は、このパラメータの説明を指定します。できるだけ人が読んで分かるように書きましょう。場合によっては、値の取り得る範囲も明記した方がいいでしょう。

5.4.3 OPTION_SETUP

```
OPTION_ SETUP( option );
```

OPTION_SETUP は、オプションを初期化します。これはマクロで宣言されており、マクロ展開するときに、 argc，argv を指定しています。これによって、以下に説明する optlib 関数群が使用できます。

5.4.4 optmanual

```
void optmanual
  P1 (( long, ext_code )) /* 終了コード */
```

引数は終了コードです。 optmanual は、オプション定義からマニュアル表示を行ないます。オプション定義のオプション名以外のすべての情報を整形して出力します。表示の後は、終了コードで終了します。

5.4.5 optusage

```
void optusage
  P1 (( long, ext_code )) /* 終了コード */
```

引数は終了コードです。 optusage は、オプション定義から使用方法を簡易表示します。オプション名とオプション指示とその引数を表示します。表示の後は、

終了コードで終了します。

5.4.6 optspecified

```
long optspecified
  P1 (( char *, name )) /* オプション名 */
```

引数はオプション定義で指定したオプション名です。 optspecified は、オプション $name$ が指定されたかどうか真理値で返します。もし、指定されていれば 1 を返し、そうでなければ 0 を返します。

5.4.7 optvalue, optvalueint, optvaluefloat

```
char *optvalue
  P1 (( char *, name )) /* オプション名 */

long optvalueint
  P1 (( char *, name )) /* オプション名 */

double optvaluefloat
  P1 (( char *, name )) /* オプション名 */
```

引数はオプション定義で指定したオプション名です。 optvalue は、オプション $name$ の指定されたパラメータ値を文字列を返します。もし、指定されていない場合は初期値を返します。初期値を持たないオプションは、対話的にユーザに値を要求します。 optvalueint は、 optvalue で得た値を整数に変換して返します。 optvaluefloat は、 optvalue で得た値を実数に変換して返します。

5.5 コマンドの例：単純閾値処理

この節では、単純閾値処理を例にとって一つの完全なコマンドを作成します。何かコマンドを作るときには、少なくとも頭の中には仕様を考えておくことが必要です。そして、コマンドを作成したら、みんなのためにマニュアルを書いてください。

5.5.1 単純二値化処理コマンドの仕様

仕様は、プログラムを作る上で問題を整理するとともに、このコマンドを利用する人に正確な情報を与えるためのものです。プログラムを作る人は、コマンドを作成するにあたって必要な情報をすべて明記しなければなりません。

何をするコマンドか

このコマンドは単純二値化処理を行なうコマンドです。単純二値化は、ある値を用いて閾値処理を行ない二値化する処理です。このコマンドは、入力画像の画素値が、ある閾値 `threshold` 以上であれば 0 (背景) とし、そうでなければ 1 (図形) とします。

入出力は何か

このコマンドは、画素型が UChar の濃淡画像を入力とし、画素値が Bit1 の二値画像を出力します。

コマンドの指定方法

パラメータは、入力画像 `input`、出力画像 `output`、閾値 `threshold` の 3 つあります。入力画像と出力画像はオプションなしで指定し、閾値はオプション指定 `-t` をしてから値を指定します。

5.5.2 単純二値化処理コマンドのプログラム

単純閾値処理の完全なプログラムを以下に示します。このプログラムでは、一応エラーチェックも行なっています。閾値のエラーチェックは行なっていませんが動作には問題ありません。

```
 1: /*
 2:  * 濃淡画像に単純二値化処理をするコマンド
 3:  *    filename : thresholding.c
 4:  *    author   : Takahiro Sugiyama
 5:  *    date     : Tuesday, May 10, 1994
 6:  */
 7:
 8: #include "Image.h"
 9: #include "misc/optlib.h"
10: #include <stdio.h>
11:
12: /* 濃淡画像の単純二値化処理関数の前宣言 */
13: void image_thresholding
14:   P3 (( image, output    ), /* 出力画像(image.uchar) */
15:       ( image, input     ), /* 入力画像(image.bit1) */
16:       ( long , threshold )) /* 閾値 */
17:
18: /* オプション定義 */
19: char *option[] = {
20:   "Thresholding of an uchar image.",
21:   "input:*:1:(image) name of the input uchar image",
22:   "output:*:1:(image) name of the output bit1 image",
23:   "threshold:-t:1:128:(uchar) value of the thresholding",
24:   "help:-h:print help messages",
25: };
26:
27: void main(argc,argv)
28:      int argc;
29:      char **argv;
30: {
31:   image input, output;
32:   char *input_name, *output_name;
33:   long threshold;
34:
35:   OPTION_SETUP( option ); /* 初期設定 */
36:
37:   if ( argc == 1 ) optusage( 1 );
38:   if ( optspecified( "help" ) ) optmanual( 1 );
39:
40:   /* パラメータ値の設定 */
41:   input_name  = optvalue( "input" );
42:   output_name = optvalue( "output ");
43:   threshold   = optvalueint( "threshold" );
```

```
44:   input  = Image.createFromFilename( "input", input_name );
45:   if ( input == 0 )
46:     {
47:       fprintf( stderr, "can't open file %s\n", input_name );
48:       exit( -1 );
49:     }
50:   if ( __TYPE( input ) != UChar )
51:     {
52:       fprintf( stderr, "type %s is wrong\n", input_name );
53:       exit( -1 );
54:     }
55:   output = Image.create( "output ");
56:
57:   image_thresholding( output, input, threshold );
58:   Image.save( output, output_name, "thresholding" );
59:
60:   Image.destroy( input );
61:   Image.destroy( output );
62: }
63:
64: /*
65:  * 濃淡画像の単純二値化処理関数の本体宣言
66:  */
67: void image_thresholding
68:   _P3 (( image, output    ), /* 出力画像(image.uchar) */
69:        ( image, input     ), /* 入力画像(image.bit1) */
70:        ( long , threshold )) /* 閾値 */
71: {
72:   image tmp_output;
73:   long  xsize, ysize;
74:   register long i, n;
75:   register uchar *i_ptr;
76:   register bit1  *o_ptr;
77:
78:   if ( input == output )
79:     tmp_output = Image.create( "thre:output" );
80:   else
81:     tmp_output = output;
82:
83:   xsize = __XSIZE( input );
84:   ysize = __YSIZE( input );
85:   Image.make( tmp_output, Bit1, xsize, ysize );
86:
87:   n = __AREA( input );
88:   i_ptr = __RASTER( input , uchar );
89:   o_ptr = __RASTER( tmp_output, bit1  );
90:   for ( i = 0; i < n; i++ )
91:     *o_ptr++ = ( (long)*i_ptr++ >= threshold ) ? 0 : 1;
```

```
92:   if ( input == output )
93:     {
94:       Image.copy( output, tmp_output );
95:       Image.destroy( output );
96:     }
97: }
```

第 6 章

CIL プログラミング応用

第 7 章

CIL ライブラリ・マニュアル

この章は、 CIL 全体のライブラリ構成と、 CIL 本体の関数群のリファレンス・マニュアルです。この章では、以下のことについて説明します。

- 全体のライブラリ構成
- image 構造体
- Image 関数群
- Image マクロ集
- ImageFile 関数群
- ImageDisp 関数群
- voxel 構造体
- Voxel 関数群

7.1 ライブラリ本体とヘッダファイルの関係

7.1.1 ヘッダファイル

ライブラリを使用するときは、ヘッダファイルを読み込む必要があります。表 7.1は、ライブラリと読み込むヘッダファイルの関係です。

表 7.1: ライブラリとヘッダファイル

ライブラリ	ヘッダファイル
Image 関数群	`Image.h`
ImageFile 関数群	`ImageFile.h`
ImageDisp 関数群	`ImageDisp.h`
Voxel 関数群	`Voxel.h`
typelib 関数群	`misc/typelib.h`
optlib 関数群	`misc/optlib.h`
memlib 関数群	`misc/memlib.h`
strlib 関数群	`misc/strlib.h`
timelib 関数群	`misc/timelib.h`
Image/Morphology 関数群	`Image/Morphology.h`
Image/Funcs 関数群	`Image/Funcs.h`
Image/Filter 関数群	`Image/Filter.h`
ColorImage 関数群	`ColorImage.h`
ImageCG 関数群	`ImageCG.h`
XImage 関数群	`Xcil/XImage.h`

7.1.2 ライブラリ本体

ライブラリを用いたプログラムをコンパイルするときは、ライブラリをリンクする必要があります。表 7.2は、ライブラリとライブラリ本体とリンクするときのコンパイル・オプションの関係です。

Solaris でコンパイルする場合は、 `-lsocket` をコンパイル・オプションに付け加えて下さい。

表 7.2: ライブラリ本体とオプション

ライブラリ	ライブラリ本体	コンパイル・オプション
Image 関数群	`libcil.*`	`-lcil`
ImageFile 関数群	`libcil.*`	`-lcil`
ImageDisp 関数群	`libcil.*`	`-lcil`
Voxel 関数群	`libcil.*`	`-lcil`
typelib 関数群	`libcil.*`	`-lcil`
optlib 関数群	`libcil.*`	`-lcil`
memlib 関数群	`libcil.*`	`-lcil`
strlib 関数群	`libcil.*`	`-lcil`
timelib 関数群	`libcil.*`	`-lcil`
Image/Morphology 関数群	`libcilext.*`	`-lcilext -lcil`
Image/Funcs 関数群	`libcilext.*`	`-lcilext -lcil -lm`
Image/Filter 関数群	`libcilext.*`	`-lcilext -lcil -lm`
ColorImage 関数群	`libcilext.*`	`-lcilext -lcil -lm`
ImageCG 関数群	`libXcil.*`	`-lXcil -lcil -lX11 -lm`
XImage 関数群	`libXcil.*`	`-lXcil -lcil -lX11 -lm`

7.2 image 構造体

image 構造体は、譜 7.1のように宣言してあります。これらの属性へのアクセスは直接行なうことは許されておりません。必ず、関数やマクロを通してアクセスして下さい。以下に、各属性について説明します。

```
typedef struct imageRec {
  char  *name;
  long   type;
  long   xsize;
  long   ysize;
  char **data;
} *image;
```

譜 7.1: image 構造体の定義

7.2.1 画像名 name

画像名は、プログラマ自身が画像に意味づけをするためにあるだけでなく、画像のデータ領域が、**通常のメモリ上**か**共有メモリ上**のどちらかに確保されているかという情報を持っています。共有メモリ上にデータ領域を確保したい場合は、名前の先頭を‘@’(アットマーク) にします。

画像に意味づけをするということで、命名は十分注意して行なって下さい。プログラム中に数多くある画像を見分ける識別子だと思って下さい。思わぬバグを発見する手助けになります。

7.2.2 x, y 画素数 xsize, ysize

x, y 画素数は、そのまま二次元配列の行列に対応しています。

7.2.3 画像データポインタ data

画像データポインタには、通常、二次元配列のポインタが入ります。通常のメモリ上のポインタも共有メモリ上のポインタも同様にして扱うことができます。また、この二次元配列はラスタスキャンができる構造で確保されています。

7.2.4 画素型 type

画素型には、型識別子が入ります。表 7.3に、登録されている型識別子を示します。 C 言語での型は型識別子の文字列をすべて小文字にしたものです。例えば、型識別子 UChar の型は、 uchar です。また、 typelib を使うことによって、自分の型を登録して新しく型を作ることができます。これに関しては、 typelib を参照して下さい。

PackedBit1 は実際には UChar ですが、 1byte の各 8bit を 8 画素分として扱います。 Bit1 は、実際には UChar で、有効ビットが 1bit のものです。 Bit4 は有効ビットが 4bit のものです。他の後ろの数字は画素の次元数を表しています。

表 7.3: 登録されている型識別子

PackedBit1		Bit1	Bit4		
Char	Char2	Char3	UChar	UChar2	UChar3
Short	Short2	Short3	UShort	UShort2	UShort3
Int	Int2	Int3	UInt	UInt2	UInt3
Long	Long2	Long3	ULong	ULong2	ULong3
Float	Float2	Float3	Double	Double2	Double3

7.2.5 画素型 type と画像型

CIL で定義された image には、**画素型**を属性として持っていますが、**画像型**は持っていません。しかし、実際に一部の関数の内部では、画素型に応じて画像型を規定しています。表 7.4に、画素型と画像型の対応表を示しておきます。また、画素型を持たない場合は、 type には、_None が入っています。

表 7.4: 画素型と画像型の対応表

画像型	画素型
二値画像	PackedBit1, Bit1
濃淡画像	Bit4, UChar, Short, ULong, Float, Double
カラー画像	UChar3, UShort3, ULong3
ラベル画像	UShort, Long
キャラクタ画像	Char
ベクトル画像	Short2, UShort2, Long2, ULong2, Float2, Double2
3 次元ベクトル画像	Short3, Long3, Float3, Double3
特徴画像	Char2, UChar2, Char3

7.3 Image 関数群

Image 関数群は、表 7.5のように分類できます。

表 7.5: Image 関数群の系統

生成破壊系	Image.create, Image.destroy Image.createMake, Image.createFromFilename Image.createFromImage
確保解放系	Image.make, Image.free
入出力系	Image.load, Image.save
参照系	Image.name, Image.type, Image.xsize, Image.ysize Image.data, Image.area, Image.byte, Image.raster
変更系	Image.copy, Image.clear Image.resize, Image.sub, Image.swap
表示系	Image.print, Image.display, Image.undisplay

7.3.1 生成破壊系

Image.create

```
image Image.create
  P1 (( char *, name )) /* 画像名 */
```

引数は $name$で、返り値は image です。 Image.create は、 $name$を持つ画像の**属性の領域**を確保します。この関数は、通常、 Image.destroy と**対**に使われます。

Image.destroy

```
void Image.destroy
  P1 (( image, self )) /* 画像 */
```

引数は、 image です。 Image.destroy は、画像 $self$ の持つ領域をすべて解放します。内部では、 Image.free が呼ばれ、画像のデータ領域が解放された後、画像の属性の領域を解放します。

Image.createFromFilename

```
image Image.createFromFilename
  P2 (( char *, name     ), /* 画像名 */
      ( char *, filename )) /* 画像ファイル名 */
```

引数は、 $name$と入力画像ファイル名 $filename$、返り値は image もしくは 0 です。 Image.createFromFilename は、 image を生成した後、指定されたファイル名 $filename$ から画像ファイルを読み込みます。もし、画像ファイルの読み込みに失敗した場合は、画像を破壊して 0 を返します。

Image.createMake

```
image Image.createMake
  P4 (( char *, name  ), /* 画像名 */
      ( long  , type  ), /* 型識別子 */
      ( long  , xsize ), /* x 画素数 */
      ( long  , ysize )) /* y 画素数 */
```

引数は、画像名 $name$、型識別子 $type$、 x, y 画素数 $xsize, ysize$の 4 つです。返り値は、領域の確保された image です。 Image.createMake は、 Image.create によって画像を生成し、 Image.make によってデータ領域を確保します。

Image.createFromImage

```
image Image.createFromImage
  P2 (( char *, name   ), /* 画像名 */
      ( image , source )) /* 原画像 */
```

引数は、画像名 $name$、複製する画像 $source$ です。返り値は、同じ内容の画像データが確保された image です。 Image.createFromImage は、 Image.create によって画像を生成し、 Image.copy によって $source$ をコピーします。

7.3.2 確保解放系

Image.make

```
void Image.make
  P4 (( image, self  ), /* 画像 */
      ( long , type  ), /* 型識別子 */
      ( long , xsize ), /* x 画素数 */
      ( long , ysize )) /* y 画素数 */
```

引数は、 image、型識別子 $type$、 x, y 画素数 $xsize, ysize$の 4 つです。 Image.make は、画像の**データ領域**を確保し初期化します。

Image.free

```
void Image.free
  P1 (( image, self )) /* 画像 */
```

引数は、 image です。 Image.free は、画像 $self$ のもつ画像データ領域だけを解放します。メモリを効率良く利用する目的以外では使用しません。画像を、 Image.destroy する場合は、呼ぶ必要はありません。内部で Image.free が呼び出されます。

7.3.3 入出力系

Image.load

```
long Image.load
  P2 (( image , self     ), /* 画像 */
      ( char *, filename )) /* 画像ファイル名 */
```

引数は、 image と入力画像ファイル名 $filename$、返り値は読み込みに成功したかどうかです。 Image.load は、指定された画像ファイル名 $filename$ を読み込みます。画像の読み込みに成功した場合は 1 を返し、失敗した場合は 0 を返します。

Image.save

```
long Image.save
  P3 (( image , self     ), /* 画像 */
      ( char *, filename ), /* 画像ファイル名 */
      ( char *, comment  )) /* コメント */
```

引数は、 image と出力画像ファイル名 $filename$ とコメント $comment$で、返り値は書き込みに成功したかどうかです。 Image.save は、指定された画像ファイル名 $filename$ に画像を書き込みます。画像の書き込みに成功した場合は 1 を返し、失敗した場合は 0 を返します。コメント $comment$は、もし画像フォーマットにコメントのフィールドがある場合はコメントを書き込みますが、もしなければコメントは無視されます。

7.3.4 参照系

Image.name, Image.type, Image.xsize, Image.ysize, Image.data

```
char *Image.name
  P1 (( image, self )) /* 画像 */

long Image.type
  P1 (( image, self )) /* 画像 */

long Image.xsize
  P1 (( image, self )) /* 画像 */

long Image.ysize
  P1 (( image, self )) /* 画像 */

char **Image.data
  P1 (( image, self )) /* 画像 */
```

各関数は、それぞれの属性を**そのまま**返します。 Image.data で返される型は char ** 型なので、実際に使用するときは、キャストをして型変換を行なって二次元配列のポインタに代入します。

Image.area, Image.byte, Image.raster

```
long Image.area
  P1 (( image, self )) /* 画像 */

long Image.byte
  P1 (( image, self )) /* 画像 */

char *Image.raster
  P1 (( image, self )) /* 画像 */
```

各関数は、それぞれの属性から計算して値を返します。 Image.area は、全画素数を返します。 Image.byte は、バイト数を返します。 Image.raster は、ラスタスキャンの先頭アドレスを返します。 Image.raster で返される型は char * 型なので、実際に使用するときは、キャストをして型変換を行なって一次元配列のポインタに代入します。

7.3.5 変更系

Image.copy

```
void Image.copy
  P2 (( image, self   ), /* コピー先画像 */
      ( image, source )) /* コピー元画像 */
```

コピー先画像 $source$ からコピー元画像 $self$ へ内容をコピーします。実際には、$self$ に $source$ と同じサイズで同じ画素型の領域が新たに確保され、データ領域をコピーします。

Image.clear

```
void Image.clear
  P1 (( image, self )) /* 画像 */
```

画像のデータ領域をゼロで埋めます。この関数は画素型に関わらず、データ領域をビット 0 で埋めるので、 Float, Double などの型の場合にはゼロにはなりません。

Image.resize

```
void Image.resize
  P4 (( image, self   ), /* 出力画像 */
      ( image, source ), /* 入力画像 */
      ( long , xsize  ), /* x画素数 */
      ( long , ysize  )) /* y画素数 */
```

入力画像 $source$ の画像の大きさを変更し、出力画像 $self$ に出力します。単純補間、単純間引き、によって拡大縮小を行ないます。

Image.sub

```
void Image.sub
   P6 (( image, self   ), /* 出力画像 */
       ( image, source ), /* 入力画像 */
       ( long , xtop   ), /* 左上 x 座標 */
       ( long , ytop   ), /* 左上 y 座標 */
       ( long , xsize  ), /* x 画素数 */
       ( long , ysize  )) /* y 画素数 */
```

入力画像 *source* の (*xtop*, *ytop*) 座標から、 (*xsize*, *ysize*) だけの大きさの画像を出力画像 *output* に出力します。

Image.swap

```
void Image.swap
  P2 (( image, self1 ), /* 交換画像 */
      ( image, self2 )) /* 交換画像 */
```

画像 *self1* と *self2* の名前以外の属性を**そのまま**交換します。ただし、名前 *name* は変更しません。

7.3.6 表示系

Image.print

```
void Image.print
  P1 (( image, self )) /* 画像 */
```

引数は、 image です。 Image.print は、画像の持つ属性を直接表示します。

Image.display

```
void Image.display
  P2 (( image , self   ), /* 画像 */
      ( char *, option )) /* オプション文字列 */
```

Image.display は、 cilserver を通して imagedisp コマンドを使って、画像を表示します。この関数を使用するには、あらかじめ、 cilserver コマンドを起動してお

く必要があります。オプション文字列 *option* は、 imagedisp コマンドが持つオプションをコマンドラインでの指定と同じように指定します。

Image.undisplay

```
void Image.unsdisplay
  P1 (( image, self )) /* 画像 */
```

Image.undisplay は、 cilserver を通して、 Image.display で起動した imagedisp コマンドを終了します。この関数を使用するには、あらかじめ cilserver コマンドを起動しておく必要があります。

7.4 Image マクロ集

CIL には高速に属性にアクセスするために、マクロが定義されています。マクロは、表 7.6のように 2 つの系統に分けることができます。

表 7.6: Image マクロ集の系統

参照系	__NAME, __XSIZE, __YSIZE, __TYPE, __DATA __AREA, __BYTE, __RASTER
画素系	__PIXEL_PBIT1, __PIXEL_PBIT1_SET, __PIXEL_PBIT1_RESET __PIXEL, __PIXEL0, __PIXEL1, __PIXEL2

7.4.1 参照系

__NAME, __XSIZE, __YSIZE, __TYPE, __DATA

```
char *__NAME( image )

long __TYPE( image )

long __XSIZE( image )

long __YSIZE( image )

type **__DATA( image, type )
```

各属性の値を直接返します。 __DATA に関しては、マクロでキャストを行なうので、 *type* には、 C 言語の型名を直接指定して下さい。

__AREA, __BYTE, __RASTER

```
long __AREA( image )

long __BYTE( image )

type *__RASTER( image, type )
```

各属性値から計算して値を返します。 __AREA は、全画素数を返します。 __BYTE は、全バイト数を返します。 __RASTER は、ラスタスキャンの先頭番地を返しま

す。 __RASTER はマクロでキャストを行なうので、 $type$ には C 言語の型名を直接指定して下さい。

7.4.2 画素系

__PIXEL_PBIT1, __PIXEL_PBIT1_SET, __PIXEL_PBIT1_RESET

```
long __PIXEL_PBIT1( image, x, y )

__PIXEL_PBIT1_SET( image, x, y )

__PIXEL_PBIT1_RESET( image, x, y )
```

__PIXEL_PBIT1 は、 PackedBit1 型を持つ画像の (x, y) 座標の画素値を参照します。 __PIXEL_PBIT1_SET は、 (x, y) 座標の画素値を 1 にします。 __PIXEL_PBIT1_RESET は、 (x, y) 座標の画素値を 0 にします。マクロなので、画像の PackedBit1 型や x, y の範囲には十分注意して下さい。

__PIXEL, __PIXEL0, __PIXEL1, __PIXEL2

```
type __PIXEL( image, x, y, type )

type.at[0] __PIXEL0( image, x, y, type )

type.at[1] __PIXEL1( image, x, y, type )

type.at[2] __PIXEL2( image, x, y, type )
```

__PIXEL は、型 $type$ を持つ画像の (x, y) 座標の画素値を参照／変更します。変更するときはそのまま代入します。 __PIXEL0 は、多次元の型 $type$ を持つ画像の (x, y) 座標の、画素の第 1 次元目の値を参照／変更します。同様に、 __PIXEL1 は画素の第 2 次元目の値を、 __PIXEL2 は画素の第 3 次元目の値を参照／変更します。マクロなので、画像型や x, y の範囲には十分注意して下さい。

7.5 ImageFile 関数群

ImageFile 関数群は、画像ファイル・フォーマットの情報をさらに詳しく知るための関数群です。また、書き込み画像のファイル・フォーマットを指定するときにも、この関数群を使います。また、 Image 関数群の load と save は ImageFile を使って実現されています。

7.5.1 画像ファイル・フォーマット

現在、 CIL でサポートしているフォーマットは、表 7.7のようになっています。フォーマットはすべてフォーマット名で参照します。大文字でも小文字でも構いません。

表 7.7: CIL でサポートしているフォーマット

フォーマット名	読み	書き	可能画素型
c2d	○	○	任意
pnm	○	○	Bit1, UChar, UChar3
j4	○	○	Bit1, Bit4, UChar, UChar3, UShort
tiff	○	○	Bit1, UChar, UChar3
ps	○	○	Bit1, UChar, UChar3
viff	○	○	Bit1, UChar, UChar3
gif	○	○	Bit1, UChar, UChar3 (256 色パレット)
jpeg	○	○	Bit1, UChar, UChar3
xbm	○	○	Bit1
xwd	○	○	Bit1, UChar, UChar3
s3d	○	×	Bit1, Bit4
dib	○	×	UChar3 (1bit,4bit,8bit)

7.5.2 画像ファイル読み込み規則

画像ファイルを読み込むときにファイル名を指定する場合、拡張子を省略することができたり、 **gzip, compress** などで圧縮してあるファイルを読み込んだり

することができます。また、画像データベース・ディレクトリを環境変数に指定しておくことによって、画像ファイルのディレクトリを省略することもできます。CIL には画像ファイルを読み込むときの**画像ファイル読み込み規則**があります。

画像ファイルの探索は、次のような順番で行なわれます。

(1) 指定されたファイル名
(2) 拡張子を付加したファイル名
(3) 画像データベース・ディレクトリを付加したファイル名
(4) 画像データベース・ディレクトリと拡張子を付加したファイル名

ファイルが見つかると次に圧縮してあるかどうか調べて、圧縮がしてあれば展開をします。そして、画像ファイルのヘッダからフォーマットを識別して、各フォーマットの読み込みルーチンで画像ファイルを読み込みます。

以下に、拡張子の付加規則と画像データベース・ディレクトリの指定方法について述べます。

拡張子の付加規則

画像ファイルの指定のときは拡張子を省略することができます。指定した画像ファイルが見つからない場合、拡張子を付加して画像ファイルを探しにいきます。この付加する拡張子と順番は、表 7.8のようになっています。

表 7.8: 拡張子の付加規則

フォーマット名	拡張子リスト
j4	j4, r（rgb カラー画像）
pnm	pbm, pgm, ppm
c2d	c2d
tiff	tiff, tif
viff	viff, vif
gif	gif
j4	m（濃淡画像）

画像データベース・ディレクトリ CIL_IMAGE_DIR

画像データベース・ディレクトリを指定することによって、ファイル名を入力するときにディレクトリを省略することができます。画像データベース・ディレクトリの指定は、環境変数 CIL_IMAGE_DIR に設定することで行ないます。ディレクトリの指定は複数可能で、各ディレクトリは ‘:’(コロン) で区切ります。ディレクトリは指定された順番に付加されます。

7.5.3 フォーマット関係

ImageFile.getLoadFormat

```
char *ImageFile.getLoadFormat();
```

最後にファイルから読み込んだファイルのフォーマットを、フォーマット名で返します。

ImageFile.getSaveFormat

```
char *ImageFile.getSaveFormat();
```

現在の書き込みファイル・フォーマットをフォーマット名で返します。

ImageFile.setSaveFormat

```
void ImageFile.setSaveFormat
  P1 (( char *, format_name )) /* フォーマット名 */
```

書き込むファイル・フォーマットをフォーマット名で設定します。 Image.save は、 ImageFile を使用しているので、 Image.save での書き込むフォーマットもこの指定によります。

7.5.4 ヘッダ関係

ImageFile.getComment

```
char *ImageFile.getComment();
```

最後にファイルから読み込んだファイルのコメントを返します。この関数は、ファイル・フォーマットがコメントを持っているものに限って有効です。

ImageFile.getHeader

```
char *ImageFile.getHeader();
```

最後にファイルから読み込んだファイルのヘッダ部分を返します。この関数は、ファイル・フォーマットが文字列のヘッダを持っているものに限って有効です。

ImageFile.getQuality

```
long ImageFile.getQuality();
```

現在設定されている書き込みクオリティを返します。この関数は、ファイル・フォーマットがクオリティを持っているものに限って有効です (jpeg, gif)。

ImageFile.setQuality

```
void ImageFile.getQuality
  P1 (( long, quality )) /* クオリティ */
```

画像ファイルを書き込むときに指定するクオリティを設定します。 0 〜 100 の範囲で指定します。この関数は、ファイル・フォーマットがクオリティを持っているものに限って有効です (jpeg, gif)。

7.5.5 入出力関係

ImageFile.load

```
long ImageFile.load
  P2 (( image , self     ), /* 画像 */
      ( char *, filename )) /* ファイル名 */
```

画像ファイルをファイル名 *filename* から画像 *self* に読み込みます。通常は、この関数は使用しないで、 Image.load の方を使用して下さい。

ImageFile.save

```
void ImageFile.save
  P3 (( image , self     ), /* 画像 */
      ( char *, filename ), /* ファイル名 */
      ( char *, comment  )) /* コメント */
```

画像 *self* をファイル名 *filename* で画像ファイルに書き込みます。通常は、この関数は使用しないで、 Image.save の方を使用して下さい。

7.6 ImageDisp 関数群

ImageDisp 関数群は、アプリケーションが cilserver を通して imagedisp と通信するための関数群です。 imagedisp から X Window のイベントを獲得したり再描写要求を出したりします。これらの関数は Image.display から呼び出されています。これらの関数群を使ってプログラミングする場合は、基本的にはイベントドリブンの形になります。これに関しては、 X Window のプログラミングに関する本を参照して下さい。

7.6.1 イベントの説明

imagedisp が処理できるイベントは、表 7.9のようになっています。これらのイベントとイベントマスク、イベント構造体の関係が示されています。

表 7.9: imagedisp で処理されるイベント

イベント名	マスク	構造体	説明
NoEvent			何もない
ImageExpose	ImageExposeMask	CILAnyEvent	再描写要求
MouseMove	MouseMoveMask	CILMouseEvent	マウスが動いた
MousePress	MousePressMask	CILMouseEvent	マウスが押された
MouseRelease	MouseReleaseMask	CILMouseEvent	マウスが離された
MouseEnter	MouseEnterMask	CILMouseEvent	マウスが入った
MouseLeave	MouseLeaveMask	CILMouseEvent	マウスが入った
KeyboardPress	KeyboardPressMask	CILKeyboardEvent	キーが押された
KeyboardRelease	KeyboardReleaseMask	CILKeyboardEvent	キーが押された

7.6.2 構造体の説明

イベント構造体

```
typedef union {
  long type; /* イベントタイプ */
  CILAnyEvent any; /* 任意のイベント構造体 */
  CILMouseEvent mouse;        /* マウスイベント構造体 */
  CILKeyboardEvent keyboard; /* キーボードイベント構造
体 */
} CILEvent;
```

基本的にこの構造体を通してイベントは受け渡されます。

任意のイベント構造体

```
typedef struct {
  long type; /* イベントタイプ */
  long detail; /* 詳細情報 */
  long x, y; /* マウスの絶対座標 */
  long window_x, window_y; /* マウスのウインドウ座標 */
} CILAnyEvent;
```

任意のイベントの構造体です。

マウスのイベント構造体

```
typedef struct {
  long type; /* イベントタイプ */
  long button; /* ボタン情報 */
  long x, y; /* マウスの絶対座標 */
  long window_x, window_y; /* マウスのウインドウ座標 */
} CILMouseEvent;
```

マウスに関するイベントの構造体です。

キーボードのイベント構造体

```
typedef struct {
  long type; /* イベントタイプ */
  long ascii; /* アスキーコード */
  long x, y; /* マウスの絶対座標 */
  long window_x, window_y; /* マウスのウインドウ座標 */
  long keysym; /* キーボードシンボル */
} CILKeyboardEvent;
```

キーボードに関するイベントの構造体です。

7.6.3 関数の説明

ImageDispExec

```
long ImageDispExec
  P2 (( image , self ), /* 画像 */
      ( char *, opt  )) /* imagedisp のオプション */
```

ImageDispExec は、画像 $self$ をオプション opt を指定して imagedisp で表示します。

ImageDispOK

```
long ImageDispOK
  P1 (( image, self )) /* 画像 */
```

ImageDispOK は、 imagedisp が通信可能状態かどうか調べます。通信可能であるならば 0 以外の値が返されます。

ImageDispQuit

```
void ImageDispQuit
  P1 (( image, self )) /* 画像 */
```

ImageDispQuit は、 imagedisp を終了します。

ImageDispDraw

```
void ImageDispDraw
  P1 (( image, self )) /* 画像 */
```

ImageDispDraw は、再描写要求を発行します。

ImageDispDrawArea

```
void ImageDispDrawArea
  P5 (( image, self  ), /* 画像 */
      ( long , x     ), /* 左上 x 座標 */
      ( long , y     ), /* 左上 y 座標 */
      ( long , xsize ), /* 幅 */
      ( long , ysize )) /* 高さ */
```

ImageDispDrawArea は、指定された領域 (x, y)-$(xsize, ysize)$ の再描写要求を発行します。

ImageDispSelectEvent

```
void ImageDispSelectEvent
  P2 (( image, self ), /* 画像 */
      ( long , mask )) /* イベントマスク */
```

ImageDispSelectEvent は、イベントマスクを設定します。

ImageDispNextEvent

```
long ImageDispNextEvent
  P2 (( image     , self  ), /* 画像 */
      ( CILEvent *, event )) /* イベント構造体 */
```

ImageDispNextEvent は、 imagedisp に設定されているイベントが発生したら、その情報をイベント構造体 $event$ に格納します。正しくイベントが発生していれば 1 が返ってきます。

ImageDispCheckEvent

```
long ImageDispCheckEvent
  P2 (( image     , self  ), /* 画像 */
      ( CILEvent *, event )) /* イベント構造体 */
```

ImageDispCheckEvent は、 imagedisp に設定されているイベントが発生しているかどうか調べて、発生していればその情報をイベント構造体 $event$ に格納し 0 を返し、そうでなければ 0 を返します。

7.7 voxel 構造体

voxel 構造体は、譜 7.46のように宣言してあります。これらの属性へのアクセスは直接行なうことは許されておりません。必ず、関数を通してアクセスして下さい。以下に、各属性について説明します。

```
typedef struct voxelRec {
  char   *name;
  long    type;
  long    xsize;
  long    ysize;
  long    zsize;
  char ***data;
} *voxel;
```

譜 7.46: voxel 構造体の定義

7.7.1 ボクセル名 name

ボクセル名は、プログラマ自身がボクセルに意味づけをするためのものです。ボクセルに意味づけをするということで、命名は十分注意して行なって下さい。プログラム中に数多くあるボクセルを見分ける識別子だと思って下さい。思わぬバグを発見する手助けになります。

7.7.2 x, y, z 画素数 xsize, ysize, zsize

x, y, z 画素数は、そのまま三次元配列の行列の大きさに対応しています。

7.7.3 ボクセルデータポインタ data

ボクセルデータポインタには、三次元配列のポインタが入ります。この三次元配列は二次元配列のリストとして確保されています。

7.7.4 画素型 type

画素型には、型識別子が入ります。表 7.3に、登録されている型識別子を示します。 C 言語での型は型識別子の文字列をすべて小文字にしたものです。例えば、型識別子 UChar の型は、 uchar です。また、 typelib を使うことによって、自分の

型を登録して新しく型を作ることができます。これに関しては、 typelib を参照して下さい。

7.8 Voxel 関数群

Voxel 関数群は、表 7.10のように分類できます。

表 7.10: Voxel 関数群の系統

生成破壊系	Voxel.create, Voxel.destroy
	Voxel.createMake, Voxel.createFromFilename
確保解放系	Voxel.make, Voxel.free
入出力系	Voxel.load, Voxel.save
参照系	Voxel.name, Voxel.type
	Voxel.xsize, Voxel.ysize, Voxel.xsize
	Voxel.volume, Voxel.byte
	Voxel.data, Voxel.data2D, Voxel.zraster
変更系	Voxel.copy, Voxel.clear
	Voxel.resize, Voxel.sub, Voxel.swap
表示系	Voxel.print

7.8.1 生成破壊系

Voxel.create

```
voxel Voxel.create
  P1 (( char *, name )) /* ボクセル名 */
```

引数は $name$で、返り値は voxel です。 Voxel.create は、 $name$を持つボクセルの**属性の領域**を確保します。この関数は、通常、 Voxel.destroy と**対**に使われます。

Voxel.destroy

```
void Voxel.destroy
  P1 (( voxel, self )) /* ボクセル */
```

引数は、 voxel です。 Voxel.destroy は、ボクセル $self$ の持つ領域をすべて解放します。内部では、 Voxel.free が呼ばれ、ボクセルのデータ領域が解放された後、ボクセルの属性の領域を解放します。

Voxel.createFromFilename

```
voxel Voxel.createFromFilename
  P2 (( char *, name     ), /* ボクセル名 */
      ( char *, filename )) /* ボクセルファイル名 */
```

引数は、 $name$と入力ボクセルファイル名 $filename$、返り値は voxel もしくは 0 です。 Voxel.createFromFilename は、 voxel を生成した後、指定されたファイル名 $filename$ からボクセルファイルを読み込みます。もし、ボクセルファイルの読み込みに失敗した場合は、ボクセルを破壊して 0 を返します。

Voxel.createMake

```
voxel Voxel.createMake
  P5 (( char *, name  ), /* ボクセル名 */
      ( long  , type  ), /* 型識別子 */
      ( long  , xsize ), /* x 画素数 */
      ( long  , ysize ), /* y 画素数 */
      ( long  , zsize )) /* z 画素数 */
```

引数は、ボクセル名 $name$、型識別子 $type$、 x, y, z 画素数 $xsize, ysize, zsize$の 5 つです。返り値は、領域の確保された voxel です。 Voxel.createMake は、 Voxel.create によってボクセルを生成し、 Voxel.make によってデータ領域を確保します。

7.8.2 確保解放系

Voxel.make

```
void Voxel.make
  P5 (( voxel, self  ), /* ボクセル */
      ( long , type  ), /* 型識別子 */
      ( long , xsize ), /* x画素数 */
      ( long , ysize )) /* y画素数 */
      ( long , zsize )) /* z画素数 */
```

引数は、 voxel、型識別子 $type$、 x, y, z 画素数 $xsize, ysize, zsize$の 5 つです。 Voxel.make は、ボクセルの**データ領域**を確保し初期化します。

Voxel.free

```
void Voxel.free
  P1 (( voxel, self )) /* ボクセル */
```

引数は、 voxel です。 Voxel.free は、ボクセル $self$のもつボクセルデータ領域だけを解放します。メモリを効率良く利用する目的以外では使用しません。ボクセルを、 Voxel.destroy する場合は、呼ぶ必要はありません。内部で Voxel.free が呼び出されます。

7.8.3 入出力系

Voxel.load

```
long Voxel.load
  P2 (( voxel , self     ), /* ボクセル */
      ( char *, filename )) /* ボクセルファイル名 */
```

引数は、 voxel と入力ボクセルファイル名 $filename$、返り値は読み込みに成功したかどうかです。 Voxel.load は、指定されたボクセルファイル名 $filename$ を読み込みます。ボクセルの読み込みに成功した場合は 1 を返し、失敗した場合は 0 を返します。

Voxel.save

```
long Voxel.save
  P3 (( voxel , self     ), /* ボクセル */
      ( char *, filename ), /* ボクセルファイル名 */
      ( char *, comment  )) /* コメント */
```

引数は、 voxel と出力ボクセルファイル名 $filename$ とコメント $comment$で、返り値は書き込みに成功したかどうかです。 Voxel.save は、指定されたボクセルファイル名 $filename$ にボクセルを書き込みます。ボクセルの書き込みに成功した場合は 1 を返し、失敗した場合は 0 を返します。コメント $comment$は、もしボクセルフォーマットにコメントのフィールドがある場合はコメントを書き込みますが、もしなければコメントは無視されます。

7.8.4 参照系

Voxel.name, Voxel.type, Voxel.xsize, Voxel.ysize, Voxel.zsize, Voxel.data

```
char *Voxel.name
  P1 (( voxel, self )) /* ボクセル */

long Voxel.type
  P1 (( voxel, self )) /* ボクセル */

long Voxel.xsize
  P1 (( voxel, self )) /* ボクセル */

long Voxel.ysize
  P1 (( voxel, self )) /* ボクセル */

long Voxel.zsize
  P1 (( voxel, self )) /* ボクセル */

char ***Voxel.data
  P1 (( voxel, self )) /* ボクセル */
```

各関数は、それぞれの属性を**そのまま**返します。 Voxel.data で返される型は char *** 型なので、実際に使用するときは、キャストをして型変換を行なって三次元元配列のポインタに代入します。

Voxel.area, Voxel.byte, Voxel.data2D, Voxel.zraster

```
long Voxel.volume
  P1 (( voxel, self )) /* ボクセル */

long Voxel.byte
  P1 (( voxel, self )) /* ボクセル */

char **Voxel.data2D
  P2 (( voxel, self ), /* ボクセル */
      ( long , z    )) /* z座標 */

char *Voxel.zraster
  P2 (( voxel, self ), /* ボクセル */
      ( long , z    )) /* z座標 */
```

各関数は、それぞれの属性から計算して値を返します。 Voxel.volume は、全画素数を返します。 Voxel.byte は、バイト数を返します。 Voxel.data2D は、 z 軸の x, y 平面の先頭アドレスを返します。 Voxel.data2D で返される型は `char **` 型なので、実際に使用するときは、キャストをして型変換を行なって二次元配列のポインタに代入します。 Voxel.zraster は、 z 軸の x, y 平面のラスタスキャンの先頭アドレスを返します。 Voxel.zraster で返される型は `char *` 型なので、実際に使用するときは、キャストをして型変換を行なって一次元配列のポインタに代入します。

7.8.5 変更系

Voxel.copy

```
void Voxel.copy
  P2 (( voxel, self   ), /* コピー先ボクセル */
      ( voxel, source )) /* コピー元ボクセル */
```

コピー先ボクセル $source$ からコピー元ボクセル $self$ へ内容をコピーします。実際には、 $self$ に $source$ と同じサイズで同じ画素型の領域が新たに確保され、データ領域をコピーします。

Voxel.clear

```
void Voxel.clear
  P1 (( voxel, self )) /* ボクセル */
```

ボクセルのデータ領域をゼロで埋めます。この関数は画素型に関わらず、データ領域をビット 0 で埋めるので、 Float, Double などの型の場合にはゼロにはなりません。

Voxel.resize

```
void Voxel.resize
  P5 (( voxel, self   ), /* 出力ボクセル */
      ( voxel, source ), /* 入力ボクセル */
      ( long , xsize  ), /* x 画素数 */
      ( long , ysize  ), /* y 画素数 */
      ( long , zsize  )) /* z 画素数 */
```

入力ボクセル *source* のボクセルの大きさを変更し、出力ボクセル *self* に出力します。単純補間、単純間引き、によって拡大縮小を行ないます。

Voxel.sub

```
void Voxel.sub
   P8 (( voxel, self   ), /* 出力ボクセル */
       ( voxel, source ), /* 入力ボクセル */
       ( long , xtop   ), /* 左上 x 座標 */
       ( long , ytop   ), /* 左上 y 座標 */
       ( long , ztop   ), /* 左上 z 座標 */
       ( long , xsize  ), /* x 画素数 */
       ( long , ysize  ), /* y 画素数 */
       ( long , zsize  )) /* z 画素数 */
```

入力ボクセル *source* の $(xtop,\ ytop,\ ztop)$ 座標から、 $(xsize,\ ysize,\ zsize)$ だけの大きさのボクセルを出力ボクセル *output* に出力します。

Voxel.swap

```
void Voxel.swap
  P2 (( voxel, self1 ), /* 交換ボクセル */
      ( voxel, self2 )) /* 交換ボクセル */
```

ボクセル *self1* と *self2* の名前以外の属性を**そのまま**交換します。ただし、名前 *name* は変更しません。

7.8.6 表示系

Voxel.print

```
void Voxel.print
  P1 (( voxel, self )) /* ボクセル */
```

引数は、 voxel です。 Voxel.print は、ボクセルの持つ属性を直接表示します。

第 8 章

misc ライブラリ・マニュアル

この章は、 misc ライブラリのリファレンス・マニュアルです。 misc ライブラリは、 CIL 本体とは完全に独立していますが、 CIL では重要な位置付けにあります。 misc は主に低水準入出力ルーチンとよく使う関数が集められています。この章では、特にアプリケーションを書くときに必要になると思われるライブラリについて説明します。

- typelib ライブラリ
- optlib ライブラリ
- memlib ライブラリ
- strlib ライブラリ
- timelib ライブラリ

8.1 typelib ライブラリ

typelib ライブラリは、 misc ライブラリのひとつで、主に型管理を行なっています。 misc/typelib.h には、 CIL でよく用いる画素型が定義されています。また、型セレクション機能 (型による動的 switch 文) も提供しています。

8.1.1 新しく定義されている型

以下に、 misc/typelib.h で定義されている型をそのまま示します。 packedbit1 は、 1 バイトを 8 ビットで 8 画素分として使用しています。 bit1 は、 1 バイトを 1 ビットだけ有効として使用しています。 bit4 は、 1 バイトを 4 ビットだけ有効として使用しています。

```
typedef unsigned char packedbit1;
typedef unsigned char bit1;
typedef unsigned char bit4;

typedef unsigned char  uchar;
typedef unsigned short ushort;
typedef unsigned int   uint;
typedef unsigned long  ulong;

typedef struct { char   at[ 2 ]; } char2;
typedef struct { short  at[ 2 ]; } short2;
typedef struct { int    at[ 2 ]; } int2;
typedef struct { long   at[ 2 ]; } long2;
typedef struct { uchar  at[ 2 ]; } uchar2;
typedef struct { ushort at[ 2 ]; } ushort2;
typedef struct { uint   at[ 2 ]; } uint2;
typedef struct { ulong  at[ 2 ]; } ulong2;
typedef struct { float  at[ 2 ]; } float2;
typedef struct { double at[ 2 ]; } double2;

typedef struct { char   at[ 3 ]; } char3;
typedef struct { short  at[ 3 ]; } short3;
typedef struct { int    at[ 3 ]; } int3;
typedef struct { long   at[ 3 ]; } long3;
typedef struct { uchar  at[ 3 ]; } uchar3;
typedef struct { ushort at[ 3 ]; } ushort3;
typedef struct { uint   at[ 3 ]; } uint3;
typedef struct { ulong  at[ 3 ]; } ulong3;
typedef struct { float  at[ 3 ]; } float3;
typedef struct { double at[ 3 ]; } double3;
```

8.1.2 型管理と型識別子

typelib では型を**型名**と**バイト数**と**型識別子**で管理しています。型の情報であるバイト数や型名を得る場合は、型識別子を使って行ないます。以下に、登録されている型識別子を示します。

_None
PackedBit1 Bit1 Bit4
Char Char2 Char3 UChar UChar2 UChar3
Short Short2 Short3 UShort UShort2 UShort3
Int Int2 Int3 UInt UInt2 UInt3
Long Long2 Long3 ULong ULong2 ULong3
Float Float2 Float3 Double Double2 Double3

8.1.3 関数の説明

typeinit

```
long typeinit
  P0 ( void )
```

型情報の初期化を行ないます。通常は使用しません。

typeprint

```
void typeprint
  P1 (( long, typeid )) /* 型識別子 */
```

型識別子 $typeid$ の持つ型情報である、型名と型サイズを表示します。

typeenter

```
long typeenter /* 型識別子 */
  P2 (( char *, name ), /* 型名     */
      ( long  , size )) /* 型サイズ */
```

型名 $name$ と型サイズ $size$ を指定して、新しい型を登録します。返り値として、

登録された型識別子を返します。

typeget

```
long typeget /* 型識別子 */
  P1 (( char *, name )) /* 型名 */
```

型名 $name$ から型識別子を得ます。

typename

```
char *typename /* 型名 */
  P1 (( long, typeid )) /* 型識別子 */
```

型識別子 $typeid$ の持つ名前を返します。

typesize

```
long typesize /* 型サイズ */
  P1 (( long, typeid )) /* 型識別子 */
```

型識別子 $typeid$ の持つ型サイズを返します。

typeselect

```
char *typeselect /* 要素ポインタ */
  P4 (( long  ,typeid ), /* 型識別子   */
      ( char *,table  ), /* テーブル   */
      ( long  ,num    ), /* 要素数     */
      ( long  ,size   )) /* 要素サイズ */
```

typeselect は、型セレクション機能を提供する関数です。型セレクション機能とは、型識別子とあるデータを 1 対 1 の対応づけをしているテーブルから、指定した型識別子に対応するデータを選択する機能です。

typeselect の引数は、型識別子 $typeid$、一次元配列のテーブルのポインタ $table$、その要素数 num と要素サイズ $size$ です。 typeselect は、指定した型識別子に対応している、テーブルに存在しているデータのポインタを返します。もし、対応す

るデータがない場合は 0 を返します。返す型は char のポインタなのでキャストする必要があります。

typeselectconst

```
type *typeselectconst( typeid, type, table )
```

typeselectconst は、 typeselect 関数をマクロ定義したもので、内部で要素数と要素サイズを、指定した型から計算しています。指定するテーブルは静的に宣言されたものでなければなりません。

typeselectconst の引数は、型識別子 $typeid$、テーブルの要素の型 $type$、テーブルのポインタ $table$ です。 typeselectconst の返す値は、指定した型 $type$ のポインタにキャストしてあります。

8.2 optlib ライブラリ

optlib ライブラリは、ユーザとアプリケーションのパラメータの受渡しのインタフェースを提供するライブラリです。 optlib プログラミングは、大きく分けて**オプション定義**、**オプション初期設定**、**パラメータ参照**から成っています。また、ユーザから見た optlib は、 4 つの方法でパラメータの入力を行なうことができます。

8.2.1 パラメータの受渡し方法

optlib には、あらかじめ設定されているオプションがあり、それを使用することによって多くの機能が利用できます。これらのオプションのことを、**システム・オプション**と呼びます。このシステム・オプションを用いることによって、ユーザからアプリケーションにパラメータを渡す方法を全部で 4 つ提供しています。以下に 4 つの方法を説明します。

コマンドライン

これは、最も良く利用する方法で、コマンドラインからオプション指定によって行なわれるパラメータの渡し方です。

コンフィグ・ファイル

コンフィグ・ファイルを使ってパラメータを指定する方法です。この方法は、一度にたくさんのオプションを指定した場合や、実験などで一部だけ違うオプションを指定したいときなどに有効です。 optlib では、システム・オプションにコンフィグ・ファイルを作成する機能があります。この機能で作成されるコンフィグ・ファイルは**シェルスクリプト**で書かれており、それ自身で実行できる形になっています。また、このファイルをシステム・オプションを使って読み込むこともできます。

コンフィグ・ファイルの利用方法はいろいろ考えられます。研究のパラメータの記録にも使えると思いますし、実験するためシェルスクリプトのスケルトン・プログラムにも使えると思います。

リソース・ファイル

リソース・ファイルは、良く指定するオプションをあらかじめ設定しておくファイルです。システム・オプションにリソース・ファイルを作成する機能があります。通常、リソース・ファイルは環境変数 CIL_RESOURCE_DIR に指定されるディレクトリに格納され、コマンドが実行されるとそのコマンドのリソース・ファイルをそのディレクトリから読み込みます。

また、システム・オプションには、リソース・ファイルに**タイプ**を割り当て、それに対応したリソース・ファイルを読み込む機能もあります。

対話的

このパラメータの受渡し方法は、いろいろな場面で重要な役割をします。だいたい、以下の 4 つの場面で利用されることになると思います。

(1) プログラム中で値が必ず必要なオプションに対して、入力がなされていなかったとき、対話的に入力を要求します。プログラマはオプションが入力されたかされなかったかは考えなくていいです。

(2) コンフィグ・ファイル、リソース・ファイルを作成するときに、**オプション・シェル**に入って対話的にパラメータを設定します。

(3) プログラム中で、動的にパラメータの入力が必要になったとき、ユーザに対話的に入力を要求することができます。プログラマは明示的にパラメータの入力要求を指定することになります。

(4) 考えながら、ゆっくりパラメータを入力するときに、**オプション・シェル**に入って対話的にパラメータを設定することができます。

オプション・シェルは、パラメータの入力を行なうためのシェルモードです。これについては、後で説明します。

8.2.2 システム・オプション

システム・オプションは、あらかじめ optlib に定義されているオプションです。システム・オプションには、表 8.1のようなオプションがあります。

表 8.1: システム・オプション

指定方法	簡略形	パラメータ	説明
--usage--	--U	なし	Usage を表示する。
--help--	--H	なし	Help を表示する。
--check--	--C	なし	解析結果を表示する。デバッグ用。
--parameter--	--P	なし	パラメータを順番に入力していく。
--resource--	--R	<type>	タイプで指定したリソースを読み込む。
--noresource--	--NR	なし	リソースを読み込まない。
--prresource--	--PR	なし	リソースを表示する。
--shell--	--S	なし	オプション・シェルモードに入る。
--output--	--O	<filename>	コンフィグ・ファイルを書き出す。
--input--	--I	<filename>	コンフィグ・ファイルを読み込む。

8.2.3 オプション・シェル

オプション・シェルは、コンフィグ・ファイルやリソース・ファイルを作成するときや、考えながらパラメータの設定を行なうときなどいろいろな用途で使用されます。

基本的にオプション・シェルは、C シェルのようなものですが、C シェルの内部コマンドはありません。独自の内部コマンドと通常の実行形式なら実行できます。以下に、内部コマンドを説明します。

· ?

オプション・シェルモードの簡単なヘルプメッセージを表示します。また、‘?’ は、対話的にオプションの値を入力するモードでも有効なコマンドです。

· quit, ^D

オプション・シェルを終了します。すべての設定したオプションは無効になります。‘^D’(コントローラキーを押しながら ‘D’) は、キー入力のときならいつでも有効です。

· usage

コマンドの簡単な説明を表示します。内部でシステム・オプション --usage-- を使って表示しています。

· help

コマンドのヘルプを表示します。内部でシステム・オプション --help-- を使って表示しています。

· show {<options>|*}

オプションに何が設定されているか表示します。引数に '*' を指定すると、すべてのオプションの状態を表示します。

· set {<options>|<options> = <values>|*}

オプションに値を設定します。引数には、オプション名かオプション指示を指定します。すべてのオプションの値を設定したい場合は、'*'(アスタリスク) を指定します。何も引数を指定しないと、すべてのオプションの状態を表示します。また、引数にオプション名のあとに '='(イコール)、値を指定すると、値を直接設定できます。

· unset {<options>|*}

引数にオプション名を指定して、オプションを指定していないことにします。すべてのオプションを指定していない状態にしたい場合は、'*'(アスタリスク) を指定します。

· load <filename>

コンフィグ・ファイルをロードします。基本的には save コマンドでセーブされたコンフィグ・ファイルだけが有効です。

· save <filename>

コンフィグ・ファイルをセーブします。コンフィグ・ファイルはシェルスクリプトで書かれており、それ自身でも実行できるファイルになっています。

· source [<type>]

リソース・ファイルを読み込みます。環境変数 CIL_RESOURCE_DIR が設定されている場合は、そのディレクトリから読み込みます。

· resource [<type>]

リソース・ファイルを書き込みます。環境変数 CIL_RESOURCE_DIR が設定されている場合は、そのディレクトリに書き込みます。ファイル名は**コマンド名**.rc になります。

· run

コマンドを実行します。シェルモードで設定したオプションの状態でコマンドを呼び出します。

8.2.4 オプション定義の方法

プログラムの中でオプション定義の文法について説明します。オプションの定義は、以下のように文字列の一次元配列で構成されます。各行は一つのオプションもしくはコメントに対応しています。

```
static char *option[] = {
  "コメント文",
  "オプション定義文",
  .....,
};
```

オプション定義文

```
名前: 指示:[引数の数:][初期値:]*[(型名)] 説明
```

一つのオプション定義は、**名前**と**指示**と**説明**を必ず属性として持ちます。また、オプションで引数を持つものは、**引数の数**と**初期値**と**型名**を指定することができます。各属性は、基本的に ‘:’(コロン) で区切られています。

· **名前**

名前は、プログラム内部でオプションを参照するときの名前です。一つのオプション定義リスト内部での名前はユニークでなければなりません。

· **指示**

指示は、ユーザがコマンドラインでオプションを指定するときに使用します。これもユニークでなければなりません。もし、**指示**を持たないオプションであるならば、つまり、値をコマンドラインで直接指定するパラメータは、‘*’(アスタリスク) を指定します。この指定によるパラメータの順番は、定義した順番になります。

· **引数の数、初期値**

引数の数には、コマンドライン上で**指示**のあとに、引数を指定するオプションの引数の数を指定します。引数を持たない場合は、何も指定する必要はありません。また、‘*’(アスタリスク) を指定することによって、可変引数も設定できます。コマンドラインでの指定における、可変引数の明示的な終了コード (EOL, end of list) は‘--’(マイナスが二つ) です。**初期値**は、コマンドラインで指定されなかったオプションに対するパラメータの初期値を指定します。もし、初期値を持たない場合は指定する必要はありません。この初期値の指定の数は、引数の数を越えてはいけません。

· **型名、説明**

型名は、ユーザに何を入力させるのかを明確にすることができます。**説明**は、このパラメータの説明を指定します。できるだけ人が読んで分かるように書きましょう。場合によっては、値の取り得る範囲も明記した方がいいでしょう。

コメント文

コメントは、マニュアルの表示のときに有効になります。コメント文には、コメントをどこに表示するかを制御する制御文字があります。その制御文字は、‘<’

と ‘>’ です。コメント文は、以下の 3 つのパターンがあります。

```
1:   コメント
2:   <: コメント
3:   >: コメント
```

2 行目のように、先頭が ‘<:’ となっているコメントは、 Usage の前に表示されます。 3 行目のような、先頭が ‘>:’ となっているコメントはタブづけされて表示されます。タブは、一つ前のオプションの説明の文の始まりに設定されます。

8.2.5 関数説明

optinit

```
void optinit
  P4 (( int    , optionnum  ), /* オプションの数      */
      ( char **, optionlist ), /* オプションのリスト */
      ( int    , argc       ), /* 引数の数            */
      ( char **, argv       )) /* 引数のリスト        */
```

optinit は、オプションを初期化しコマンドラインを解析します。 *optionnum* にはオプション定義の数を指定し、 *optionlist* にはそのポインタを指定します。 *argc, argv* には main 関数の引数をそのまま指定します。通常この関数は使用しないで、次のマクロを使用します。

OPTION_SETUP

```
OPTION_SETUP( option );
```

OPTION_SETUP は、オプションを初期化します。これはマクロで宣言されており、マクロ展開するときに *argc, argv* を指定しています。

optmanual

```
void optmanual
  P1 (( long, ext_code )) /* 終了コード */
```

引数は終了コードです。 optmanual は、オプション定義からマニュアル表示を行ないます。オプション定義のオプション名以外のすべての情報を整形して出力します。表示の後は、終了コードで終了します。

optusage

```
void optusage
  P1 (( long, ext_code )) /* 終了コード */
```

引数は終了コードです。 optusage は、オプション定義から使用方法を簡易表示します。オプション名とオプション指示とその引数を表示します。表示の後は、終了コードで終了します。

optspecified

```
long optspecified
  P1 (( char *, name )) /* オプション名 */
```

引数はオプション定義で指定したオプション名です。 optspecified は、オプション $name$ が指定されたかどうか真理値で返します。もし、指定されていれば 1 を返し、そうでなければ 0 を返します。

optvalue, optvalueint, optvaluefloat

```
char *optvalue
  P1 (( char *, name )) /* オプション名 */

long optvalueint
  P1 (( char *, name )) /* オプション名 */

double optvaluefloat
  P1 (( char *, name )) /* オプション名 */
```

引数はオプション定義で指定したオプション名 $name$ です。 optvalue は、オプション $name$ の指定されたパラメータ値を文字列を返します。もし、指定されていない場合は初期値を返します。初期値を持たないオプションは、対話的にユーザに値を要求します。 optvalueint は、 optvalue で得た値を整数に変換して返します。 optvaluefloat は、 optvalue で得た値を実数に変換して返します。

optnvalue, optnvalueint, optnvaluefloat

```
char *optnvalue
  P2 (( char *, name  ), /* オプション名 */
      ( long  , index )) /* i番目の引数 */

long optnvalueint
  P2 (( char *, name  ), /* オプション名 */
      ( long  , index )) /* i番目の引数 */

double optnvaluefloat
  P2 (( char *, name  ), /* オプション名 */
      ( long  , index )) /* i番目の引数 */
```

引数はオプション定義で指定したオプション名 $name$ と、そのオプションのパラメータ値の引数の添字 $index$ です。この関数は、指定したオプションが複数の引数を持っている場合に有効です。 optnvalue は、オプション $name$ の指定されたパラメータ値の $index$ 番目の値を返します。もし、指定されていない場合は初期値を返します。初期値を持たないオプションは、対話的にユーザに値を要求します。 optnvalueint は、 optnvalue で得た値を整数に変換して返します。 optnvaluefloat は、 optnvalue で得た値を実数に変換して返します。

optvaluenum, optvaluelist

```
long optvaluenum /* オプションの値の数 */
  P1 (( char *, name )) /* オプション名 */

char **optvaluelist /* オプションの値のリスト */
  P1 (( char *, name )) /* オプション名 */
```

これらは、オプションの引数の数が可変引数のときに有効です。 optvaluenum は、指定された引数の数を返します。 optvaluelist は、指定された引数の文字列のリストを返します。

optinput

```
void optinput
  P1 (( char *, name )) /* オプション名 */
```

指定したオプションの値をユーザに対話的に入力要求します。もし、オプション名が 0 のときはすべてのオプションを入力要求します。 "ALL"ならば、コマンドラインなどで指定されなかったオプションすべてに対して入力を要求します。 "NEED"ならば、コマンドラインなどで指定されておらず、初期値を持たないオプションすべてに対して入力を要求します。

optinputconfigure

```
void optinputconfigure
  P1 (( char *, filename )) /* コンフィグ・ファイル名 */
```

コンフィグ・ファイルを読み込みます。すでにオプションが設定されている場合は、設定されているオプションの方を優先します。

optoutputconfigure

```
void optoutputconfigure
  P1 (( char *, filename )) /* コンフィグ・ファイル名 */
```

設定されいるオプションの状態をコンフィグ・ファイルに書き込みます。

optshellmode

```
void optshellmode();
```

オプション・シェルモードに入ります。

8.3 memlib ライブラリ

memlib ライブラリは、通常のメモリの確保と解放などの操作を行ないます。一次元配列、二次元配列、三次元配列の確保解放ができます。このライブラリは、画像のデータ領域を確保するときに使用されるライブラリです。このライブラリには、**バイト数**で指定する関数と、**型**で指定するマクロがあります。

8.3.1 メモリ確保チェック

すべての関数でメモリが確保されたかどうかチェックしています。この確保できなかったときのライブラリの動作は二種類あります。**強制終了**か 0 **を返す**のどちらかです。この動作をコントロールするための大域変数 MEMLIB_ERROR があります。通常は、この大域変数には 1 が設定されており、メモリが確保できない場合は**強制終了**します。もし、強制終了せずに 0 を返して欲しい場合は、この大域変数の値を 0 に設定しておきます。

8.3.2 単純メモリ操作

単純メモリ操作は、バイト単位でメモリを確保したりする操作です。ここで扱うメモリは、配列のような構造は持ちません。

memnew

```
char *memnew /* ポインタ */
  P1 (( long, size )) /* バイト数 */
```

memnew は、メモリをバイト数 $size$ だけ確保します。確保できない場合は強制終了するか 0 を返します。

memrenew

```
char *memrenew /* ポインタ */
  P2 (( char *, ptr  ), /* ポインタ */
      ( long  , size )) /* バイト数 */
```

memrenew は、ポインタ ptr の指し示すメモリをバイト数 $size$ に再確保します。

返り値は、新しく確保したメモリのポインタです。 ptr が 0 のときは、 memnew を呼び出しポインタを返します。

memfree

```
void memfree
  P1 (( char *, ptr )) /* ポインタ */
```

memfree は、ポインタ ptr の指し示すメモリを解放します。 ptr が 0 のときは何もしません。

memcopy

```
void memcopy
  P3 (( char *, dest ), /* ポインタ */
      ( char *, src  ), /* ポインタ */
      ( long  , size )) /* バイト数 */
```

memcopy は、ポインタ src からポインタ $dest$ へ、メモリの内容をバイト数 $size$ だけコピーします。

memfill

```
void memfill
  P3 (( char *, dest ), /* ポインタ */
      ( long  , c    ), /* 埋める文字 */
      ( long  , size )) /* バイト数 */
```

memfill は、ポインタ $dest$ のメモリをバイト数 $size$ だけキャラクタ c で埋めます。

typenew **マクロ**

```
type *typenew( type );
```

typenew は、型 $type$ のサイズだけメモリ確保します。返す値は、型 $type$ のポインタ ($type$ *) にキャストしています。

typerenew **マクロ**

```
type *typerenew( ptr, type );
```

typerenew は、ポインタ ptr の指し示すメモリを型 $type$ のサイズに再確保します。返す値は、型 $type$ のポインタ ($type$ *) にキャストしています。

typefree **マクロ**

```
void typefree( ptr );
```

typefree は、ポインタ ptr のメモリを解放します。 ptr が 0 のときは何もしません。

8.3.3 一次元配列操作

memnew1

```
char *memnew1 /* 一次元配列のポインタ */
  P2 (( long, num  ), /* 要素数 */
      ( long, size )) /* 要素サイズ */
```

memnew1 は、要素数 num、要素サイズ $size$ の一次元配列のメモリ確保を行ないます。確保できない場合は強制終了するか 0 を返します。

memrenew1

```
char *memrenew1 /* 一次元配列のポインタ */
  P3 (( char *, ptr ), /* 一次元配列のポインタ */
      ( long  , num ), /* 要素数 */
      ( long  , size)) /* 要素サイズ */
```

memrenew1 は、ポインタ ptr の指し示すメモリを、要素数 num、要素サイズ $size$ の一次元配列に再確保します。 ptr が 0 のときは、 memnew1 を呼びポインタを返します。

memfree1

```
void memfree1
  P1 (( char *, ptr )) /* 一次元配列のポインタ */
```

memfree1 は、ポインタ ptr のメモリを解放します。ptr が 0 のときは何もしません。

memcopy1

```
void memcopy1
  P4 (( char *, dest ), /* 一次元配列のポインタ */
      ( char *, src  ), /* 一次元配列のポインタ */
      ( long  , num  ), /* 要素数 */
      ( long  , size )) /* 要素サイズ */
```

memcopy1 は、要素サイズが $size$ の一次元配列 src から $dest$ へ、要素数 num だけメモリの内容をコピーします。

memfill1

```
void memfill1
  p4 (( char *, dest ), /* 一次元配列のポインタ */
      ( long  , c    ), /* 埋める文字 */
      ( long  , num  ), /* 要素数 */
      ( long  , size )) /* 要素サイズ */
```

memfill1 は、ポインタ $dest$ のメモリをバイト数 $size \times num$ だけキャラクタ c で埋めます。

typenew1 **マクロ**

```
type *typenew1( num, type );
```

typenew1 は、型 $type$、要素数 num の一次元配列を確保します。返す値は、型 $type$ のポインタ ($type$ *) にキャストされています。

typerenew1 **マクロ**

```
type *typerenew1( ptr, num, type );
```

typerenew1 は、一次元配列ポインタ ptr を、型 $type$、要素数 num の一次元配列に再確保します。返す値は、型 $type$ のポインタ ($type$ *) にキャストしています。

typefree1 **マクロ**

```
void typefree1( ptr );
```

typefree1 は、ポインタ ptr のメモリを解放します。 ptr が 0 のときは何もしません。

8.3.4 二次元配列操作

memenw2

```
char **memnew2 /* 二次元配列のポインタ */
  P3 (( long, xnum ), /* Xの要素数 */
      ( long, ynum ), /* Yの要素数 */
      ( long, size )) /* 要素サイズ */
```

memnew2 は、行数 $ynum$、列数 $xnum$、要素サイズ $size$ の二次元配列のメモリ確保を行ないます。確保できない場合は強制終了するか 0 を返します。

memfree2

```
void memfree2
  P1 (( char **, ptr )) /* 二次元配列のポインタ */
```

memfree2 は、二次元配列 ptr を解放します。 ptr が 0 のときは何もしません。

memcopy2

```
void memcopy2
  P5 (( char **, dest ), /* 二次元配列のポインタ */
      ( char **, src  ), /* 二次元配列のポインタ */
      ( long   , xnum ), /* Xの要素数 */
      ( long   , ynum ), /* Yの要素数 */
      ( long   , size )) /* 要素サイズ */
```

memcopy2 は、行数 $ynum$、列数 $xnum$、要素サイズ $size$ の二次元配列 src から $dest$ へ、メモリの内容をコピーします。

memfill2

```
void memfill2
  P5 (( char **, dest ), /* 二次元配列のポインタ */
      ( long   , c    ), /* 埋める文字 */
      ( long   , xnum ), /* Xの要素数 */
      ( long   , ynum ), /* Yの要素数 */
      ( long   , size )) /* 要素サイズ */
```

memfill2 は、行数 $ynum$、列数 $xnum$、要素サイズ $size$ の二次元配列 $dest$ のメモリをキャラクタ c で埋めます。

typenew2 **マクロ**

```
type **typenew2( xnum, ynum, type );
```

typenew2 は、型 $type$、行数 $ynum$、列数 $xnum$ の二次元配列を確保します。返す値は、型 $type$ の二次元配列ポインタ ($type$ **) にキャストされています。

typefree2 **マクロ**

```
void typefree2( ptr );
```

typefree2 は、二次元配列 ptr を解放します。 ptr が 0 のときは何もしません。

8.3.5 三次元配列操作

memnew3

```
char ***memnew3 /* 三次元配列のポインタ */
  P4 (( long, xnum ), /* Xの要素数 */
      ( long, ynum ), /* Yの要素数 */
      ( long, znum ), /* Zの要素数 */
      ( long, size )) /* 要素サイズ */
```

memnew3 は、z 要素数 $znum$、y 要素数 $ynum$、x 要素数 $xnum$、要素サイズ $size$ の三次元配列のメモリ確保を行ないます。確保できない場合は強制終了するか 0 を返します。

memfree3

```
void memfree3
  P1 (( char ***, ptr )) /* 三次元配列のポインタ */
```

memfree3 は、三次元配列 ptr を解放します。ptr が 0 のときは何もしません。

memcopy3

```
void memcopy3
  P6 (( char ***, dest ), /* 三次元配列のポインタ */
      ( char ***, src  ), /* 三次元配列のポインタ */
      ( long    , xnum ), /* Xの要素数 */
      ( long    , ynum ), /* Yの要素数 */
      ( long    , znum ), /* Zの要素数 */
      ( long    , size )) /* 要素サイズ */
```

memcopy3 は、z 要素数 $znum$、y 要素数 $ynum$、x 要素数 $xnum$、要素サイズ $size$ の三次元配列 src から $dest$ へ、メモリの内容をコピーします。

memfill3

```
void memfill3
  P6 (( char ***, dest ), /* 二次元配列のポインタ */
      ( long    , c    ), /* 埋める文字 */
      ( long    , xnum ), /* Xの要素数 */
      ( long    , ynum ), /* Yの要素数 */
      ( long    , znum ), /* Zの要素数 */
      ( long    , size )) /* 要素サイズ */
```

memfill3 は、z 要素数 $znum$、y 要素数 $ynum$、x 要素数 $xnum$、要素サイズ $size$ の三次元配列 $dest$ のメモリをキャラクタ c で埋めます。

typenew3 **マクロ**

```
type ***typenew3( xnum, ynum, znum, type );
```

typenew3 は、型 $type$、z 要素数 $znum$、y 要素数 $ynum$、x 要素数 $xnum$ の三次元配列を確保します。返す値は、型 $type$ の三次元配列ポインタ ($type$ ***) にキャストされています。

typefree3 **マクロ**

```
void typefree3( ptr );
```

typefree3 は、三次元配列 ptr を解放します。ptr が 0 のときは何もしません。

8.4 strlib ライブラリ

strlib ライブラリは、よく使う文字列の操作を行ないます。文字列の確保や、文字列の分離、漢字コードの変換などを行ないます。その他、このライブラリには string.h にあるのと同様な関数群がありますが、エラーチェックをするという以外、内容はまったく変わりません。

8.4.1 文字列の分離

文字列を特定の文字で分離する処理は、簡単な字句解析を行なうときによく使用します。 strsplit 関数はその操作を行なう関数です。

strsplit

```
long strsplit /* 文字列の数 */
  P3 (( char  *, words      ), /* 文字列 */
      ( char **, head       ), /* 分離文字列リスト */
      ( char  *, separaters )) /* 区切り文字リスト */
```

strsplit は、文字列 $words$ を区切り文字 $separaters$ で分離して、各先頭を $head$ に入れます。返り値として区切れた文字列の数を返します。

もし、区切り文字リスト $separaters$ によって、文字列 $words$ がすべてなくなってしまった場合は 0 を返します。また、‘"’(ダブルクオート) か ‘’’(シングルクオート) で区切られている文字列は分離されません。

文字列 $words$ の内容は、破壊されるので注意が必要です。もとの文字列 $words$ を後で使用する場合は、あらかじめコピーを取っておいて下さい。また、文字列リスト head は、文字列の一次元配列ですが、あらかじめ多めに宣言しておいて下さい。例えば、次のようにします。

```
char *head[64];
```

· strsplit **の例**

一つ例を示しておきます。次のような文字列

```
"Hello, how are you?"
```

に対して、区切り文字 "␣,?"で strsplit 関数を施すと分離結果は次のようになります。

```
head[0] -> "Hello"
head[1] -> "how"
head[2] -> "are"
head[3] -> "you"
```

この場合、 strsplit の返り値は、 4 となります。

8.4.2 文字列の基本関数

strnew

```
char *strnew /* 文字列 */
  P1 (( char *, str )) /* 文字列 */
```

strnew は、文字列 str と同じ文字列を確保してそのポインタを返します。実際には、

```
strcpy( malloc( strlen(str) + 1 ), str )
```

が行なわれます。

strfree

```
void strfree
  P1 (( char *, str )) /* 文字列 */
```

strfree は、文字列 str を解放します。

strlength

```
long strlength
  P1 (( char *, str )) /* 文字列 */
```

strlength は、文字列 str の文字数を返します。文字列 str が NULL ポインタの時は、 0 を返します。

strprintlength

```
long strlength
  P1 (( char *, str )) /* 文字列 */
```

strprintlength は、文字列 str の表示可能な文字数を返します。文字列の中に JIS-Code が含まれている場合などに正確にバイト単位の文字数を計算します。文字列 str が NULL ポインタの時は、 0 を返します。

strprintf

```
long strprintf
  Pn (/* char *buf, char *format, ... )
```

strprintf は、文字列 buf にフォーマット $format$ にしたがった文字列をプリントします。 sprintf と全く同じ働きをします。

8.4.3 文字列比較関数

strcompare

```
int strcompare /* 文字列の差 */
  P2 (( char *, str1 ), /* 文字列1 */
      ( char *, str2 )) /* 文字列2 */
```

strcompare は、文字列 $str1$ と $str2$ を比較してその差を返します。全く同じ文字列の時は 0 を返します。

strsubcompare

```
int strsubcompare /* 文字列の差 */
  P3 (( char *, str1 ), /* 文字列1 */
      ( char *, str2 ), /* 文字列2 */
      ( int   , n    )) /* 文字数 */
```

strsubcompare は、文字列 $str1$ と $str2$ の先頭 n 文字を比較してその差を返します。全く同じ文字列の時は 0 を返します。

strxcompare

```
int strxcompare /* 文字列の差 */
  P2 (( char *, str1 ), /* 文字列 1 */
      ( char *, str2 )) /* 文字列 2 */
```

strxcompare は、文字列 $str1$ と $str2$ を大文字小文字を無視して比較してその差を返します。全く同じ文字列の時は 0 を返します。

strxsubcompare

```
int strxsubcompare /* 文字列の差 */
  P3 (( char *, str1 ), /* 文字列 1 */
      ( char *, str2 ), /* 文字列 2 */
      ( int   , n    )) /* 文字数 */
```

strxsubcompare は、文字列 $str1$ と $str2$ の先頭 n 文字を大文字小文字を無視して比較してその差を返します。全く同じ文字列の時は 0 を返します。

strcopy

```
char *strcopy /* コピー先文字列ポインタ */
  P2 (( char *, dest ), /* コピー先文字列 */
      ( char *, src  )) /* コピー元文字列 */
```

strcopy は、文字列 src から $dest$ へ文字をコピーします。返り値はコピー先文字列のポインタです。

strsubcopy

```
char *strsubcopy /* コピー先文字列ポインタ */
  P3 (( char *, dest ), /* コピー先文字列 */
      ( char *, src  ), /* コピー元文字列 */
      ( int   , n    )) /* コピー文字数 */
```

strcopy は、文字列 src から $dest$ へ文字を n 文字コピーします。返り値はコピー先文字列のポインタです。

strequal

```
int strequal /* 文字列の比較 */
  P2 (( char *, str1 ), /* 文字列1 */
      ( char *, str2 )) /* 文字列2 */
```

strequal は、文字列 $str1$ と $str2$ を比較して全く同じならば 1 をそうでなければ 0 を返します。

strsubequal

```
int strsubequal /* 文字列の比較 */
  P3 (( char *, str1 ), /* 文字列1 */
      ( char *, str2 ), /* 文字列2 */
      ( int   , n    )) /* 文字数 */
```

strsubequal は、文字列 $str1$ と $str2$ の先頭 n 文字を比較して全く同じならば 1 をそうでなければ 0 を返します。

strxequal

```
int strxequal /* 文字列の比較 */
  P2 (( char *, str1 ), /* 文字列1 */
      ( char *, str2 )) /* 文字列2 */
```

strxequal は、文字列 $str1$ と $str2$ を大文字小文字を無視して比較し、全く同じならば 1 をそうでなければ 0 を返します。

strxsubequal

```
int strxsubequal /* 文字列の比較 */
  P3 (( char *, str1 ), /* 文字列1 */
      ( char *, str2 ), /* 文字列2 */
      ( int   , n    )) /* 文字数 */
```

strxsubequal は、文字列 $str1$ と $str2$ の先頭 n 文字を大文字小文字を無視して比較し、全く同じならば 1 をそうでなければ 0 を返します。

8.4.4 変換関数

strtolong

```
long strtolong
  P1 (( char *, str )) /* 文字列 */
```

strtolong は、文字列 str を long 型に変換します。

strtoulong

```
unsigned long strtoulong
  P1 (( char *, str )) /* 文字列 */
```

strtoulong は、文字列 str を unsigned long 型に変換します。

strtodouble

```
double strtodouble
  P1 (( char *, str )) /* 文字列 */
```

strtodouble は、文字列 str を double 型に変換します。

strtoboolean

```
long strtoboolean
  P1 (( char *, str )) /* 文字列 */
```

strtoboolean は、文字列 str を boolean 型に変換します。文字列が true ならば 1 を返し、それ以外ならば 0 を返します。

strfromlong

```
char *strfromlong
  P2 (( long  , src  ), /* 整数値 */
      ( char *, dest )) /* 文字列 */
```

strfromlong は、整数値 src を文字列 $dest$ に変換します。

strfromlong

```
char *strfromlong
  P2 (( long   , src  ), /* 整数値 */
      ( char *, dest )) /* 文字列 */
```

strfromlong は、整数値 src を文字列 $dest$ に変換します。文字列 $dest$ が NULL ポインタならば変換された文字列を一時的な文字列に格納してそのポインタを返します。

strfromulong

```
char *strfromulong
  P2 (( unsigned long, src  ), /* 整数値 */
      ( char *       , dest )) /* 文字列 */
```

strfromulong は、整数値 src を文字列 $dest$ に変換します。文字列 $dest$ が NULL ポインタならば変換された文字列を一時的な文字列に格納してそのポインタを返します。

strfromdouble

```
char *strfromdouble
  P2 (( double, src  ), /* 実数値 */
      ( char *, dest )) /* 文字列 */
```

strfromdouble は、実数値 src を文字列 $dest$ に変換します。文字列 $dest$ が NULL ポインタならば変換された文字列を一時的な文字列に格納してそのポインタを返します。

strfromboolean

```
char *strfromboolean
  P2 (( long   , src  ), /* 実数値 */
      ( char *, dest )) /* 文字列 */
```

strfromboolean は、真理値 src を文字列 $dest$ に変換します。文字列 $dest$ が NULL

ポインタならば変換された文字列を一時的な文字列に格納してそのポインタを返します。真理値 src が、 0 でなければ true を、 0 ならば false に変換されます。

8.4.5 その他の関数

strjistoeuc

```
void strjistoeuc
  P2 (( char *, dst ), /* EUC 漢字コード文字列 */
      ( char *, str )) /* JIS 漢字コード文字列 */
```

strjistoeuc は、 JIS 漢字コードをもつ文字列 str を EUC 漢字コードの文字列 dst に変換します。文字列 str が、 EUC 漢字コードであっても副作用はありません。 JIS 漢字コードのタイプは、スタートコード 0x1b2440、エンドコード 0x1b**** です。

8.5 timelib ライブラリ

timelib ライブラリは、プログラムの実行時間を測定するためのライブラリです。このライブラリでは、8 つのタイマーを持っておりストップウオッチのように操作できる関数が用意されています。なお、基本的にこのライブラリは、SunOS 用のみ提供されています。また、現在の年月日および時刻を得るための関数も用意されています。

8.5.1 基本的説明

計算機では、ユーザモードとシステムモードの 2 種類でプログラムは実行されています。これらの時間を測定するのがこのライブラリです。ユーザ時間とは主にユーザが作成したプログラムコードにおいてユーザモードで実行時間を指します。システム時間とは主にユーザから出されたシステムコールによってシステムモードで実行された時間を指します。実時間というのは、ユーザ時間とシステム時間およびその切替えに要した時間の合計を指します。通常、プログラムの実行時間というのはユーザ時間のことを指します。

timelib ライブラリではこれらの時間を測定するため 8(0,. . .,7) つのタイマーを持っています。このタイマーを操作することによって実行時間を測定します。

8.5.2 タイマーの操作に関する関数

timestart

```
void timestart
  P1 (( long, no )) /* タイマー番号 */
```

timestart は、タイマー番号 no で示されるタイマーの時間を開始します。

timestop

```
void timestop
  P1 (( long, no )) /* タイマー番号 */
```

timestop は、タイマー番号 no で示されるタイマーの時間を停止します。

timepause

```
void timepause
  P1 (( long, no )) /* タイマー番号 */
```

timepause は、タイマー番号 no で示されるタイマーの時間を一時停止します。

timerestart

```
void timerestart
  P1 (( long, no )) /* タイマー番号 */
```

timerestart は、タイマー番号 no で示されるタイマーの時間を再開します。

timelapstart

```
void timelapstart
  P1 (( long, no )) /* タイマー番号 */
```

timelapstart は、タイマー番号 no で示されるタイマーの時間のラップを開始します。

8.5.3 タイマーの値参照に関する関数

timeuser

```
double timeuser
  P1 (( long, no )) /* タイマー番号 */
```

timeuser は、タイマー番号 no で示されるタイマーのユーザ時間を秒単位で返します。

timesystem

```
double timesystem
  P1 (( long, no )) /* タイマー番号 */
```

timesystem は、タイマー番号 no で示されるタイマーのシステム時間を秒単位

で返します。

timelapuser

```
double timelapuser
  P1 (( long, no )) /* タイマー番号 */
```

timelapuser は、タイマー番号 no で示されるタイマーのラップのユーザ時間を秒単位で返します。

timelapsystem

```
double timelapsystem
  P1 (( long, no )) /* タイマー番号 */
```

timelapsystem は、タイマー番号 no で示されるタイマーのラップのシステム時間を秒単位で返します。

timeusrmin

```
long timeusrmin /* 計測時間 (分) */
  P2 (( long    , no     ), /* タイマー番号 */
      ( double *, second )) /* 計測時間 (秒) */
```

timeusrmin は、タイマー番号 no で示されるタイマーのユーザ時間を分単位で返します。ポインタ $second$ にはあまりの時間が秒単位で格納されます。

timesystemmin

```
long timesystemmin /* 計測時間 (分) */
  P2 (( long    , no     ), /* タイマー番号 */
      ( double *, second )) /* 計測時間 (秒) */
```

timesystemmin は、タイマー番号 no で示されるタイマーのシステム時間を分単位で返します。ポインタ $second$ にはあまりの時間が秒単位で格納されます。

timelapusermin

```
long timelapusrmin /* 計測時間 (分) */
  P2 (( long    , no     ), /* タイマー番号 */
      ( double *, second )) /* 計測時間 (秒) */
```

timelapusrmin は、タイマー番号 no で示されるタイマーのラップのユーザ時間を分単位で返します。ポインタ $second$ にはあまりの時間が秒単位で格納されます。

timelapsystemmin

```
long timelapsystemmin /* 計測時間 (分) */
  P2 (( long    , no     ), /* タイマー番号 */
      ( double *, second )) /* 計測時間 (秒) */
```

timelapsystemmin は、タイマー番号 no で示されるタイマーのラップのシステム時間を分単位で返します。ポインタ $second$ にはあまりの時間が秒単位で格納されます。

8.5.4 日付に関する説明

dateRec **構造体**

```
struct dateRec {
  long year, month, day;      /* 年 (下二桁) 月日 */
  long week;                  /* 週 (0= 日、 1= 月...) */
  long hour, minute, second; /* 時分秒 */
  char *s_month;              /* 月の英語文字列 */
  char *s_week;               /* 週の英語文字列 */
  char str[ 64 ];             /* 全体の文字列 */
};
```

dateRec 構造体は、関数 timedate へポインタとして渡され各値が設定されます。

timedate

```
long timedate /* 計測時間（分） */
  P1 (( struct dateRec *, date ))
                /* 日時の型ポインタ */
```

timedate は、現在の時間と日にちを文字列で返します。構造体 $date$ には各細かい設定が返されます。

第 9 章

cilext ライブラリ・マニュアル

この章は、 cilext ライブラリのリファレンス・マニュアルです。画像処理でよく使う基本的な処理が集めてあります。 Funcs ライブラリは、簡単な画像の演算やラベリングなどが集めてあります。 Morphology ではモフォロジ演算が集めてあります。 ColorImage ではカラー画像の変換関数が集めてあります。

- Funcs ライブラリ
- Morphology ライブラリ
- ColorImage ライブラリ

9.1 Image/Funcs ライブラリ

Image/Funcs ライブラリは、画像を加工するためのいくつかの関数を集めたものです。

9.1.1 演算関数

image__reverse

```
void image__reverse
  P2 (( image, dest ), /* 出力画像 */
      ( image, src  )) /* 入力画像 */
```

image__reverse は、入力画像 src の画素値を反転して画像 $dest$ に出力します。

image__and

```
void image__and
  P3 (( image, dest ), /* 出力画像 */
      ( image, src1 ), /* 入力画像 1 */
      ( image, src2 )) /* 入力画像 2 */
```

image__and は、入力画像 $src1$ と $src2$ の積 (AND) をとり出力画像 $dest$ に出力します。

image__or

```
void image__or
  P3 (( image, dest ), /* 出力画像 */
      ( image, src1 ), /* 入力画像 1 */
      ( image, src2 )) /* 入力画像 2 */
```

image__or は、入力画像 $src1$ と $src2$ の和 (OR) をとり出力画像 $dest$ に出力します。

image__eor

```
void image__eor
  P3 (( image, dest ), /* 出力画像 */
      ( image, src1 ), /* 入力画像 1 */
      ( image, src2 )) /* 入力画像 2 */
```

image__eor は、入力画像 $src1$ と $src2$ の排他的論理和 (EOR) をとり出力画像 $dest$ に出力します。

9.1.2 フィルタリング関数

image__gaussian

```
void image__gaussian
  P3 (( image , dest  ), /* 出力画像 */
      ( image , src   ), /* 入力画像 */
      ( double, sigma )) /* ガウス関数の標準偏差 */
```

image__gaussian は、画像 src を標準偏差 $sigma$ のガウス関数で畳み込みをして、結果を画像 $dest$ に Float 型で出力します。有効な画素型は UChar, Short, Float, Double, Char3 型です。

image__gaussian_gradient_abs

```
void image__gaussian_guradient_abs
  P3 (( image , dest  ), /* 出力画像 */
      ( image , src   ), /* 入力画像 */
      ( double, sigma )) /* ガウス関数の標準偏差 */
```

image__gaussian_gradient_abs は、画像 src を標準偏差 $sigma$ のガウス関数を x, y 方向に一階微分したフィルタで畳み込みをして、その各画素値の大きさを結果として画像 $dest$ に Float 型で出力します。有効な画素型は UChar, Short, Float, Double 型です。

image__gaussian_gradient

```
void image__gaussian_guradient
  P3 (( image , dest   ), /* 出力画像 */
      ( image , grad_x ), /* x 方向微分画像 */
      ( image , grad_y ), /* y 方向微分画像 */
      ( image , src    ), /* 入力画像 */
      ( double, sigma  )) /* ガウス関数の標準偏差 */
```

image__gaussian_gradient は、画像 src を標準偏差 $sigma$ のガウス関数を x, y 方向に一階微分したフィルタで畳み込みをして、結果を画像 $grad\_x$ と $grad\_y$ に Float 型で出力します。また、その各画素値の大きさを結果として画像 $dest$ に Float 型で出力します。有効な画素型は UChar, Short, Float, Double 型です。

9.1.3 その他の画像処理関数

image__copyarea

```
void image__copyarea
  P8 (( image, dest   ), /* 出力画像 */
      ( image, src    ), /* 入力画像 */
      ( long , dest_x ), /* コピー先 x 座標 */
      ( long , dest_y ), /* コピー先 y 座標 */
      ( long , src_x  ), /* コピー元 x 座標 */
      ( long , src_y  ), /* コピー元 y 座標 */
      ( long , width  ), /* コピー元幅 */
      ( long , height )) /* コピー元幅 */
```

image__copyarea は、画像 src の左上座標 (src_x, src_y) から範囲 $(width, height)$ の領域を画像 $dest$ の左上座標 $(dest\_x, dest\_y)$ にコピーします。画像 $dest$ はあらかじめ領域が確保されていなければなりません。

image__liner

```
void image__liner
  P2 (( image, dest ), /* 出力画像 */
      ( image, src  )) /* 入力画像 */
```

image__liner は、入力画像 src の画素値を 0 〜 255 に線形変換して画像 $dest$ に

出力します。

image__thresholding

```
void image__thresholding
  P3 (( image , dest      ), /* 出力画像 */
      ( image , src       ), /* 入力画像 */
      ( double, threshold )) /* 閾値 */
```

image__thresholding は、画像 src を閾値 $threshold$ で閾値処理をして、結果を画像 $dest$ に二値画像として出力します。有効な画素型は Char, UChar, Short, UShort, Long, ULong, Float, Double 型です。

image__long_type

```
void image__long_type
  P2 (( image, dest ), /* 出力画像 */
      ( image, src  )) /* 入力画像 */
```

image__long_type は、画像 src の画素型を Long 型に変換して画像 $dest$ に格納します。

image__N4_labelling

```
void image__N4_labelling
  P2 (( image, dest ), /* 出力画像 */
      ( image, src  ), /* 入力ラベル画像 */
      ( long , type )) /* 出力ラベル画像の型:
                          UChar,Short,UShort,Long */
```

image__N4_labelling は、画像 src の画素型を 4 連結でラベリングして指定した type 型に変換して画像 $dest$ に格納します。

image__N8_labelling

```
void image__N8_labelling
  P2 (( image, dest ), /* 出力画像 */
      ( image, src  ), /* 入力ラベル画像 */
      ( long , type )) /* 出力ラベル画像の型:
                          UChar,Short,UShort,Long */
```

image__N8_labelling は、画像 src の画素型を 8 連結でラベリングして指定した type 型に変換して画像 $dest$ に格納します。

image__N4_labelling_foreground

```
void image__N4_labelling_foreground
  P2 (( image, dest ), /* 出力画像 */
      ( image, src  ), /* 入力ラベル画像 */
      ( long , type )) /* 出力ラベル画像の型:
                          UChar,Short,UShort,Long */
```

image__N4_labelling_foreground は、画像 src の画素型を 4 連結でラベリングして指定した type 型に変換して画像 $dest$ に格納します。ただし、 0 のラベルは無視します。

image__N8_labelling_foreground

```
void image__N8_labelling_foreground
  P2 (( image, dest ), /* 出力画像 */
      ( image, src  ), /* 入力ラベル画像 */
      ( long , type )) /* 出力ラベル画像の型:
                          UChar,Short,UShort,Long */
```

image__N8_labelling_foreground は、画像 src の画素型を 8 連結でラベリングして指定した type 型に変換して画像 $dest$ に格納します。ただし、 0 のラベルは無視します。

image__labeling

```
void image__labeling
  P2 (( image, dest ), /* 出力画像 */
      ( image, src  )) /* 入力画像 */
```

image__labeling は、画像 src を再ラベリングして画像 $dest$ に格納します。ラベルは 1 番から割り当てられます。有効な入力画像の画素型は Long, UChar, Bit1, Char, UShort, Short 型です。過去の互換のために残されています。

9.2 Image/Morphology ライブラリ

Image/Morphology ライブラリは、モフォロジ演算を行なうための基本ライブラリです。入力および出力は二値画像 (Bit1) もしくは濃淡画像 (UChar) です。

9.2.1 ストラクチャ・エレメントの説明

円形のストラクチャ・エレメントは、 /home/abe/image/disk の中に二値画像として用意してあります。サイズは 1 から 1000 まであり、ファイル名は各 4 桁の直径で表されます。

9.2.2 関数の説明

image__dilation

```
void image__dilation
  P5 (( image, dest     ), /* 出力画像 */
      ( image, src      ), /* 入力画像 */
      ( image, str_elem ), /* 構造要素 */
      ( long , spot_x   ), /* 中心 x 座標 */
      ( long , spot_y   )) /* 中心 y 座標 */
```

image__dilation は、入力画像 src にモフォロジ演算のディレーションを施し出力画像 $dest$ に出力します。ストラクチャ・エレメントは str_elem で与えられ、座標 $(sopt\_x, spot\_y)$ はストラクチャ・エレメントの中心座標です。

image__erosion

```
void image__erosion
  P5 (( image, dest     ), /* 出力画像 */
      ( image, src      ), /* 入力画像 */
      ( image, str_elem ), /* 構造要素 */
      ( long , spot_x   ), /* 中心 x 座標 */
      ( long , spot_y   )) /* 中心 y 座標 */
```

image__erosion は、入力画像 src にモフォロジ演算のエロージョンを施し出力画像 $dest$ に出力します。ストラクチャ・エレメントは str_elem で与えられ、座標 $(sopt\_x, spot\_y)$ はストラクチャ・エレメントの中心座標です。

image__opening

```
void image__opening
  P5 (( image, dest     ), /* 出力画像 */
      ( image, src      ), /* 入力画像 */
      ( image, str_elem ), /* 構造要素 */
      ( long , spot_x   ), /* 中心 x 座標 */
      ( long , spot_y   )) /* 中心 y 座標 */
```

image__opening は、入力画像 src にモフォロジ演算のオープニングを施し出力画像 $dest$ に出力します。ストラクチャ・エレメントは str_elem で与えられ、座標 $(sopt\_x, spot\_y)$ はストラクチャ・エレメントの中心座標です。

image__closing

```
void image__closing
  P5 (( image, dest     ), /* 出力画像 */
      ( image, src      ), /* 入力画像 */
      ( image, str_elem ), /* 構造要素 */
      ( long , spot_x   ), /* 中心 x 座標 */
      ( long , spot_y   )) /* 中心 y 座標 */
```

image__closing は、入力画像 src にモフォロジ演算のクロージングを施し出力画像 $dest$ に出力します。ストラクチャ・エレメントは str_elem で与えられ、座標 $(sopt\_x, spot\_y)$ はストラクチャ・エレメントの中心座標です。

9.3 ColorImage 関数群

Image/ColorImage 関数群は、カラー画像の色空間を変換するための関数や距離を計算するための関数を集めたものです。しかしながら、カラー画像の RGB から他の色空間への変換し関してはまだ不確定の部分があるので、ここでは紹介しません。

第 10 章

Xcil ライブラリ・マニュアル

この章は、 Xcil ライブラリのリファレンス・マニュアルです。主に X Window の入出力を使って構成されている関数群が集められています。 Xcil ライブラリ群は以下のライブラリで構成されています。画像に基本的な図形や文字を書くための ImageCG ライブラリ、画像を表示可能な画素型に変換する ImageShowing ライブラリ、画像を X Window 上に表示する XImage ライブラリ、そして、 GUI(グラフィカル・ユーザ・インターフェース) を構築するための Xcil ライブラリです。この章では、特にアプリケーションを書くときに必要になると思われるライブラリについて説明します。

- ImageCG ライブラリ
- ImageShowing ライブラリ
- XImage ライブラリ

10.1 ImageCG ライブラリ

ImageCG ライブラリは、画像に、基本図形である**点**、**線分**、**四角**、**円** (**円弧**、**楕円**)、**多角形**と**文字**を書き込むためのライブラリです。なお、ここで扱う座標系は左上が原点の座標系です。

10.1.1 画素値の説明

画素値の指定の仕方は、少し特殊ですので説明しておきます。基本的にここで説明する描写関数はすべて、 ImageDrawPoint マクロを使って画像に書き込んでいます。画像の任意の画素型に対応するため ImageDrawPoint では、画像の画素型にしたがって LookupTable を使って関数を呼び出します。そのため同じ関数の引数で異なった型の画素値を受け渡すために次のようにしました。 PackedBit1, Bit1, Bit4, Char, Short, Long, UChar, UShort, ULong の画素型は、直接その値で指定して、それ以外は画素型のポインタで指定します。

10.1.2 構造体とその他の値について

ImgPoint

```
typedef struct {
  short x; /* x 座標 */
  short y; /* y 座標 */
  char *pixel; /* 画素値 */
} ImgPoint;
```

ImgSegment, ImgBox

```
typedef struct {
  int x1, y1;  /* 第 1 端点 */
  int x2, y2;  /* 第 2 端点 */
  char *pixel; /* 画素値 */
} ImgSegment, ImgBox;
```

ImgRectangle

```
typedef struct {
  int x, y;                    /* 左上の座標 */
  unsigned int width, height; /* 幅と高さ */
  char *pixel;                 /* 画素値 */
} ImgRectangle;
```

ImgArg

```
typedef struct {
  short x, y;                      /* 左上の座標 */
  unsigned short width, height; /* 幅と高さ */
  short angle1;              /* 第 1 角度 (360 ° *64) */
  short angle2;              /* 第 2 角度 (360 ° *64) */
  char *pixel;                     /* 画素値 */
} ImgArc;
```

座標モード

- CoordModeOrigin ··· リスト中の座標はすべて絶対座標で示されます。
- CoordModePrevious ··· リスト中の座標は最初は絶対座標で、次からは前の座標から相対的な座標で示さます。

10.1.3 基本図形描写関数

ImageDrawPoint **マクロ**

```
void ImageDrawPoint
  P4 (( image , img   ), /* 画像 */
      ( long  , x     ), /* x 座標 */
      ( long  , y     ), /* y 座標 */
      ( char *, pixel )) /* 画素値 */
```

ImageDrawPoint は、画像 img の座標 (x, y) に画素値 $pixel$ の点を描写します。

ImageDrawPoints

```
void ImageDrawPoints
  P4 (( image     , img  ), /* 画像 */
      ( ImgPoint *, list ), /* 点リスト */
      ( long      , num  ), /* 点の数 */
      ( long      , mode )) /* 座標モード */
```

ImageDrawPoints は、座標リスト $list$ で示される座標および画素値の点を num 個だけ画像 img に描写します。

ImageDrawLine

```
void ImageDrawLine
  P6 (( image , img   ), /* 画像 */
      ( long  , x1    ), /* 端点 1 の x 座標 */
      ( long  , y1    ), /* 端点 1 の y 座標 */
      ( long  , x2    ), /* 端点 2 の x 座標 */
      ( long  , y2    ), /* 端点 2 の y 座標 */
      ( char *, pixel )) /* 画素値 */
```

ImageDrawLine は、端点 1$(x1, y1)$ から端点 2$(x2, y2)$ へ画素値 $pixel$ で線分を描写します。

ImageDrawLines

```
void ImageDrawLines
  P4 (( image     , img  ), /* 画像 */
      ( ImgPoint *, list ), /* 点リスト */
      ( long      , num  ), /* 点の数 */
      ( int       , mode )) /* 座標モード */
```

ImageDrawLines は、点リスト $list$ で示される num 個の各点を結び線分を描写します。

ImageDrawSegments

```
void ImageDrawSegments
  P3 (( image       , img  ), /* 画像 */
      ( ImgSegment *, list ), /* 線分リスト */
      ( long        , num  )) /* 線分の数 */
```

ImageDrawSegments は線分リスト $list$ 示される線分を num 個だけ画像 img に描写します。

ImageDrawBox

```
void ImageDrawBox
  P6 (( image , img   ), /* 画像 */
      ( long  , x1    ), /* 第 1 点の x 座標 */
      ( long  , y1    ), /* 第 1 点の y 座標 */
      ( long  , x2    ), /* 第 2 点の x 座標 */
      ( long  , y2    ), /* 第 2 点の y 座標 */
      ( char *, pixel )) /* 画素値 */
```

ImageDrawBox は、第 1 点 $(x1, y1)$ と第 2 点 $(x1, y1)$ を対角に持つ四角形を画像 img に画素値 $pixel$ で描写します。

ImageDrawBoxes

```
void ImageDrawBoxes
  P3 (( image    , img  ), /* 画像 */
      ( ImgBox  *, list ), /* 四角形リスト */
      ( long     , num  )) /* 四角形の数 */
```

ImageDrawBoxs は、四角形リスト $list$ で示される四角形を num 個だけ画像 img に描写します。

ImageDrawRectangle

```
void ImageDrawRectangle
  P6 (( image       , img    ), /* 画像 */
      ( int         , x      ), /* 左上の x 座標 */
      ( int         , y      ), /* 左上の y 座標 */
      ( unsigned int, width  ), /* 幅 */
      ( unsigned int, height ), /* 高さ */
      ( char *      , pixel  )) /* 画素値 */
```

ImageDrawRectangle は、左上の座標 (x, y) と幅 $width$ と高さ $height$ で示される四角形を画素値 $pixel$ で画像 img に描写します。

ImageDrawRectangles

```
void ImageDrawRectangles
  P3 (( image         , img  ),
      ( ImgRectangle *, list ),
      ( long          , num  ))
```

ImageDrawRectangles は、四角形リスト $list$ で示される四角形を num 個だけ画像 img に描写します。

ImageDrawArc

```
void ImageDrawArc
  P8 (( image       , img    ), /* 画像 */
      ( int         , x      ), /* 左上の x 座標 */
      ( int         , y      ), /* 左上の y 座標 */
      ( unsigned int, width  ), /* 幅 */
      ( unsigned int, height ), /* 高さ */
      ( int         , angle1 ), /* 第 1 角度 */
      ( int         , angle2 ), /* 第 2 角度 */
      ( char *      , pixel  )) /* 画素値 */
```

ImageDrawArc は、左上の点 (x, y) と幅 $width$ と高さ $height$ の四角形に接する楕円を角度 $angle1/64°$ から角度 $angle2/64°$ まで画素値 $pixel$ で画像 img に描写します。

ImageDrawArcs

```
void ImageDrawArcs
  P3 (( image   , img  ), /* 画像 */
      ( ImgArc *, list ), /* 円弧リスト */
      ( long    , num  )) /* 円弧の数 */
```

ImageDrawArcs は、円弧リスト $list$ で示される円弧を num 個だけ画像 img に描写します。

ImageFillRectangle

```
void ImageFillRectangle
  P6 (( image        , img    ), /* 画像 */
      ( int          , x      ), /* 左上の x 座標 */
      ( int          , y      ), /* 左上の y 座標 */
      ( unsigned int, width  ), /* 幅 */
      ( unsigned int, height ), /* 高さ */
      ( char *       , pixel  )) /* 画素値 */
```

ImageDrawRectangle は、左上の座標 (x, y) と幅 $width$ と高さ $height$ で示される四角形の範囲を画素値 $pixel$ で画像 img に塗りつぶします。

ImageFillRectangles

```
void ImageFillRectangles
  P3 (( image          , img  ), /* 画像 */
      ( ImgRectangle *, list ), /* 四角形リスト */
      ( long           , num  )) /* 四角形の数 */
```

ImageFillRectangles は、四角形リスト $list$ で示される num 個の四角形の範囲を画像 img に塗りつぶします。

ImageFillArc

```
void ImageFillArc
  P8 (( image       , img    ), /* 画像 */
      ( int         , x      ), /* 左上の x 座標 */
      ( int         , y      ), /* 左上の y 座標 */
      ( unsigned int, width  ), /* 幅 */
      ( unsigned int, height ), /* 高さ */
      ( int         , angle1 ), /* 第 1 角度 */
      ( int         , angle2 ), /* 第 2 角度 */
      ( char *      , pixel  )) /* 画素値 */
```

ImageFillArc は、左上の点 (x, y) と幅 $width$ と高さ $height$ の四角形に接する楕円を角度 $angle1/64°$ から角度 $angle2/64°$ まで画素値 $pixel$ で画像 img に塗りつぶします。

ImageFillArcs

```
void ImageFillArcs
  P3 (( image   , img  ), /* 画像 */
      ( ImgArc *, list ), /* 円弧リスト */
      ( long    , num  )) /* 円弧の数 */
```

ImageFillArcs は、円弧リスト $list$ で示される円弧を num 個だけ画像 img に塗りつぶします。

ImageFillPolygon

```
void ImageFillPolygon
  P6 (( image     , img   ), /* 画像 */
      ( ImgPoint *, list  ), /* 点リスト */
      ( long      , num   ), /* 点の数 */
      ( long      , shape ), /* 未使用 */
      ( long      , mode  ), /* 座標モード */
      ( char *    , pixel )) /* 画素値 */
```

num 個の点リスト $list$ で表される多角形を画素値 $pixel$ で画像 img を塗りつぶす。

10.1.4 文字書き込み関数

内部的で X Window の関数を呼んでいますので、 X Window で使用できるフォントを指定することができます。

ImageSetStringFont

```
void ImageSetStringFont
  P1 (( char *, name )) /* フォント名 */
```

ImageSetStringFont は、以下の関数 (ImageDrawString, ImageDrawImageString, ImageCreateString) で使用するフォントを $name$ で指定されるフォントに設定します。

ImageSetStringFont16

```
void ImageSetStringFont16
  P1 (( char *, name )) /* フォント名 */
```

ImageSetStringFont16 は、以下の関数 (ImageDrawString16, ImageDrawImageString16) で使用するフォントを $name$ で指定されるフォントに設定します。

ImageDrawString

```
void ImageDrawString
  P6 (( image , img    ), /* 画像 */
      ( long  , x      ), /* x 座標 */
      ( long  , y      ), /* y 座標 */
      ( char *, str    ), /* 文字列 */
      ( long  , length ), /* 文字数 */
      ( char *, pixel  )) /* 画素値 */
```

ImageDrawString は、文字列 str を文字数分 $length$ だけ、画像 img の座標 (x, y) 上に画素値 $pixel$ で描写します。

ImageDrawString16

```
void ImageDrawString16
  P6 (( image    , img    ), /* 画像 */
      ( long     , x      ), /* x 座標 */
      ( long     , y      ), /* y 座標 */
      ( XChar2b *, str    ), /* 2Byte 系文字列 */
      ( long     , length ), /* 文字数 */
      ( char *   , pixel  )) /* 画素値 */
```

ImageDrawString16 は、 2Byte 系文字列 str を文字数分 $length$ だけ、画像 img の座標 (x, y) 上に画素値 $pixel$ で描写します。

ImageDrawImageString

```
void ImageDrawImageString
  P7 (( image , img                ), /* 画像 */
      ( long  , x                  ), /* x 座標 */
      ( long  , y                  ), /* y 座標 */
      ( char *, str                ), /* 文字列 */
      ( long  , length             ), /* 文字数 */
      ( char *, foreground_pixel   )) /* 文字の画素値 */
      ( char *, background_pixel   )) /* 背景の画素値 */
```

ImageDrawImageString は、文字列 str を文字数分 $length$ だけ、画像 img の座標 (x, y) 上に画素値 $foregorund\_pixel$ で描写します。文字の背景は画素値 $backgorund\_pixel$ で塗りつぶされます。

ImageDrawImageString16

```
void ImageDrawImageString16
  P7 (( image    , img    ), /* 画像 */
      ( long     , x      ), /* x 座標 */
      ( long     , y      ), /* y 座標 */
      ( XChar2b *, str    ), /* 2Byte 系文字列 */
      ( long     , length ), /* 文字数 */
      ( char *   , fg_pixel  )) /* 文字の画素値 */
      ( char *   , bg_pixel  )) /* 背景の画素値 */
```

ImageDrawImageString16 は、 2Byte 系文字列 str を文字数分 $length$ だけ、画像 img の座標 (x, y) 上に画素値 fg_pixel で描写します。文字の背景は画素値 bg_pixel で塗りつぶされます。

ImagePutString

```
void ImagePutString
  P6 (( image , img      ), /* 画像 */
      ( long  , offset_x ), /* x 座標 */
      ( long  , offset_y ), /* y 座標 */
      ( char *, position ), /* 位置 */
      ( char *, str      ), /* 文字列 */
      ( char *, pixel    )) /* 画素値 */
```

ImagePutString は、文字列 str を、画像 img の座標 (x, y) 上の位置 $position$ に、画素値 $pixel$ で描写します。位置の指定は、't'(top), 'b'(bottom), 'c'(center), 'l'(left), 'r'(right) が指定できます。

ImageCreateString

```
image ImageCreateString
  P2 (( char *, str    ),  /* 文字列 */
      ( long  , length ))  /* 文字数 */
```

ImageCreateString は、文字列 str を文字数分 $length$ だけ描写した二値画像を返します。この画像を使用して必要なくなった場合は Image.destroy 関数で解放しま

す。

10.1.5 画像描写関数

ImageDrawPointImage

```
void ImageDrawPointImage
  P6 (( image , img      ), /* 画像 */
      ( long  , offset_x ), /* x 座標 */
      ( long  , offset_y ), /* y 座標 */
      ( image , points   ), /* 二値画像 */
      ( char *, position ), /* 位置 */
      ( char *, pixel    )) /* 画素値 */
```

ImageDrawPointImage は、二値画像 $points$ を、画像 img の座標 (x, y) 上の位置 $position$ に、画素値 $pixel$ で描写します。位置の指定は、't'(top), 'b'(bottom), 'c'(center), 'l'(left), 'r'(right) が指定できます。

ImageRotateLeft

```
void ImageDrawPointImage
  P1 (( image , img )) /* 画像 */
```

ImageRotateLeft は、画像 img を、 90 度左に回転します。

10.2 ImageShowing ライブラリ

ImageShowing ライブラリは、画像の画素型に意味を与えて X Window 上に表示できる型に変換します。画素型には、 feature, gray, font, label, color-label の意味をつけることができます。また、各型の意味は環境変数を用いて設定することができます。このライブラリは、 XImage ライブラリから呼び出されます。

10.2.1 XImage における表示可能の型への変換

ImageShowing は、 XImage ライブラリにおいて表示可能な型、 Bit1(二値画像), Char(テキスト画像), UChar(濃淡画像), UChar3(カラー画像), Long(ラベル画像) に画像を変換します。図 10.1に示すように、基本的に 1 次元の画素型 (bit1, bit4, char, short, long, uchar, ushort, ulong, float, double) は各画素型に解釈されてそれぞれの画素型へ、 2 次元の画素型は vector 画像として解釈され UChar3 型へ、3 次元の画素型は rgb-color 画像と解釈され UChar3 型へ、それぞれ変換され XImage に渡されます。

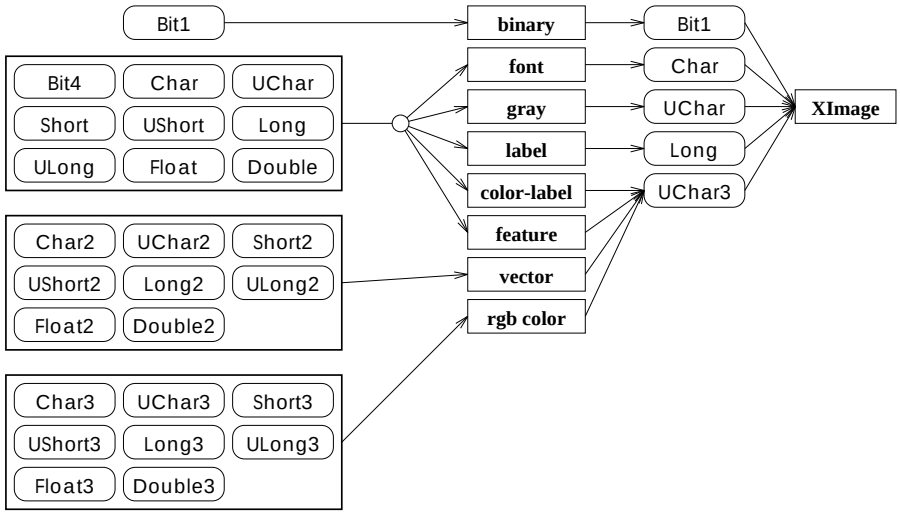


図 10.1: 変換の流れ

10.2.2 画素型の意味づけ

各画素値には、その値が何を表しているかという意味づけがなされています。ここでは、 CIL で用いられている画素値の意味について説明します。

binary 画像

binary 画像、すなわち、二値画像は白と黒の二つの値で表現された画像です。CIL では、0 を白、1 を黒として扱います。

font 画像

font 画像は、文字コードで表現さえた画像です。 CIL では、各フォントに応じて各コードに割り当てられた文字で表示されます。特徴を記号で表したいときに利用でされます。

gray 画像

gray 画像は白黒濃淡画像です。 CIL では、値が小さい方が暗く、値が大きい方が明るく表示されます。すべての 1 次元画素型は gray 画像に意味づけすることができます。

label 画像

label 画像は、各画素にラベルが付けられた画像です。基本的に領域分割をした結果を表示するために利用します。 CIL は、同じラベルは同じ領域として、境界線を表示します。ラベルの値が 0 の場合は、特別なラベルとして灰色で表示されます。ラベルの値が負の場合は、ディザで灰色表示されます。すべての 1 次元画素型は label 画像に意味づけすることができます。

color-label 画像

color-label 画像は、基本的に label 画像と同じですが、各領域が色づけされて表示されます。すべての 1 次元画素型は color-label 画像に意味づけすることができます。

feature 画像

feature 画像は、各画素の値をなんらかの特徴値としてカラー表示されます。値の小さいほうから大きい方へ、青－緑－赤と変化します。すべての 1 次元画素型は feature 画像に意味づけすることができます。

vector 画像

vector 画像は、 2 次元の画素値をベクトルとして扱います。 CIL における vector 画像の表示の方法は、図 10.2に示すような空間において対応する RGB に変換してカラー画像として表示します。すべての 2 次元画素型は vector 画像に意味づけされます。

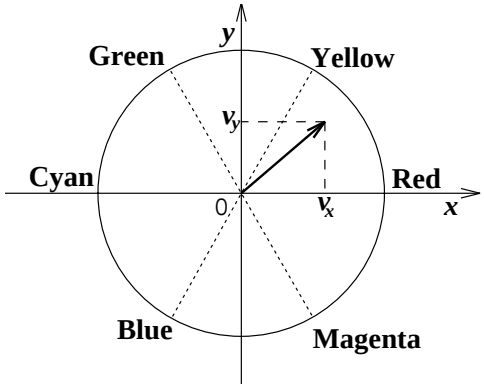


図 10.2: vector と RGB の対応

rgb color 画像

rgb color 画像は、 RGB で表されるカラー画像です。すべての 3 次元画素型は rgb color 画像に意味づけされます。

10.2.3 1 次元画素型の意味づけの方法

1 次元の画素型は、いくつかの意味づけが可能です。各画素型の意味の初期値は表 10.1のようになっています。各画素型の意味づけは環境変数を使って行ないます。

環境変数の値は、⟨ 意味 ⟩(gray, font, label, color-label, feature) を直接指定します。意味 gray, color-label, feature に関しては各値の最大値を指定することができます。最大値は、⟨ 意味 ⟩#⟨ 最大値 ⟩ と指定します。

表 10.1: 画素型と意味の初期値

画素型	初期値	環境変数
Bit4	gray#15	CIL_DP_BIT4
Char	font	CIL_DP_CHAR
UChar	gray#255	CIL_DP_UCHAR
Short	gray	CIL_DP_SHORT
UShort	label	CIL_DP_USHORT
Long	label	CIL_DP_LONG
ULong	gray	CIL_DP_ULONG
Float	gray	CIL_DP_FLOAT
Double	gray	CIL_DP_DOUBLE

10.2.4 関数の説明

ImageShowing

```
void ImageShowing
  P6 (( image, output ), /* 出力画像 */
      ( image, input  ), /* 原画像 */
      ( long , x      ), /* 左上の x 座標 */
      ( long , y      ), /* 左上の y 座標 */
      ( long , xsize  ), /* x サイズ */
      ( long , ysize  )) /* y サイズ */
```

ImageShowing は、原画像 $input$ を変換テーブルにしたがって、(x, y) から範囲 $(xsize, ysize)$ を変換します。結果の画像は $output$ に作成されます。

10.3 XImage ライブラリ

XImage ライブラリは画像を X Window 上へ表示するためのライブラリです。任意の画素型を各ディスプレイ (1Plane, 8Plane, 16Plane, 24Plane) に応じて最適な方式で表示します。また、カラー画像および濃淡画像においては、ディスプレイ特性を読み込んで変換したり、ガンマ補正をかけることができます。

10.3.1 ディスプレイ特性とガンマ補正

画像は、各画素値に応じた電圧に変換さえれブラウン管を通してディスプレイに映し出されます。このとき画素値 (電圧) とディスプレイ上の明るさが比例しないで、指数的なカーブを描きます。このときの指数をガンマ γ といいます。

画素値 $I(x,y)$ とディスプレイ上の明るさ $G(x,y)$ はガンマ γ をもちいて一般に次のような式で表されます。

$$G(x,y) = I(x,y)^{\gamma} \tag{10.1}$$

ただし、$0 \leq I(x,y) \leq 1$ です。

つまり、$\gamma > 1$ のときは入力画像より全体的に暗くなり、$\gamma < 1$ のときには入力画像より全体的に明るくなります。通常ディスプレイには、$\gamma_d > 1$ のガンマ補正が、プリンタ、スキャナには $\gamma_i < 1$ のガンマ補正がかけられています。そして、これらがうまく打ち消しあって全体として、

$$\gamma_d \gamma_i = 1 \tag{10.2}$$

となりスキャナで入力した画像がディスプレイ上で正しい色で見ることができ、プリンタで正しく印刷されるのです。しかし、各ディスプレイ、各スキャナ、各プリンタで正確な γ の値がわからなければ正確な色の再現は不可能です。出版業界では昔から重要な問題となっています。最近、DTP がはやりだし、マッキントッシュなどでも色合わせの技術が導入されてきました。

実際のところ γ に従うようなディスプレイは少ないようです。正確に色を再現しようと思ったら各ディスプレイに応じて、各画素値とディスプレイ上での明るさを測定しなければなりません。そこで、CIL では、ルックアップテーブルによる補正と γ によるガンマ補正の両方を用いることにしました。

ディスプレイごとの特性のルックアップテーブルについては、環境変数 CIL_DP_DIR で示されるディレクトリに、ディスプレイのホスト名で格納しておきます。そうすれば自動的に XImage ライブラリがディスプレイ特性を読み込んで変換して表示します。

しかしながらこれだけでは、なんらかの装置によって入力された画像に対しては正しく表示されません。その入力装置自体の γ 補正があるからです。残念ながら現在のところそれらの入力装置の正確な γ 補正はわかりません。よってこのような入力画像に対してはディスプレイ上では正しく見ることができません。とりあえず、 XImage ライブラリでは、 γ を指定することができます。環境変数 CIL_GAMMA に γ の値を指定します。

10.3.2 関数の説明

XShowImage

```
void XShowImage
  P12 (( Display *, display ), /* Display */
       ( Window   , w       ), /* Window */
       ( GC       , gc      ), /* Graphic Context */
       ( image    , img     ), /* 画像 */
       ( int, win_x ), /* Window の左上の x 座標 */
       ( int, win_y ), /* Window の左上の y 座標 */
       ( unsigned int, win_xsize ), /* Window の表示幅
*/
       ( unsigned int, win_ysize ), /* Window の表示高
さ */
       ( int, img_x ), /* 画像の左上の x 座標 */
       ( int, img_y ), /* 画像の左上の y 座標 */
       ( unsigned int, img_xsize ), /* 画像の表示幅 */
       ( unsigned int, img_ysize )) /* 画像の表示高さ
*/
```

XShowImage は、画像 img の領域 (img_x, img_y)-(img_xsize, img_ysize) をウィンドウ w の領域 (win_x, win_y)-(win_xsize, win_ysize) 上に表示します。

XShowImagePartOfImage

```
void XShowImagePartOfImage
  P8 (( Display *, display ), /* Display */
      ( Window   , w       ), /* Window */
      ( GC       , gc      ), /* Graphic Context */
      ( image    , img     ), /* 画像 */
      ( int, img_x ), /* 画像の左上の x 座標 */
      ( int, img_y ), /* 画像の左上の y 座標 */
      ( unsigned int, img_xsize ), /* 画像の表示幅 */
      ( unsigned int, img_ysize )) /* 画像の表示高さ */
```

XShowImagePartOfImage は、画像 img の領域 (img_x, img_y)-(img_xsize, img_ysize) をウィンドウ w 上に表示します。

XShowImagePartOfWindow

```
void XShowImagePartOfWindow
  P8 (( Display *, display ), /* Display */
      ( Window   , w       ), /* Window */
      ( GC       , gc      ), /* Graphic Context */
      ( image    , img     ), /* 画像 */
      ( int, win_x ), /* Window の左上の x 座標 */
      ( int, win_y ), /* Window の左上の y 座標 */
      ( unsigned int, win_xsize ), /* Window の表示幅 */
      ( unsigned int, win_ysize )) /* Window の表示高さ
*/
```

XShowImagePartOfWindow は、画像 img をウィンドウ w の領域 (win_x, win_y)-

(win_xsize, win_ysize) 上に表示します。

XShowImageFull

```
void XShowImageFull
  P4 (( Display *, display ), /* Display */
      ( Window   , w       ), /* Window */
      ( GC       , gc      ), /* Graphic Context */
      ( image    , img     ), /* 画像 */
```

XShowImageFull は、画像 img をウィンドウ w 上に表示します。

XDrawImage

```
void XDrawImage
  P13 (( Display *, display ), /* Display */
       ( Window   , w       ), /* Window */
       ( GC       , gc      ), /* Graphic Context */
       ( int, win_x ), /* Window の左上の x 座標 */
       ( int, win_y ), /* Window の左上の y 座標 */
       ( unsigned int, win_xsize ), /* Window の表示幅
*/
       ( unsigned int, win_ysize ), /* Window の表示高
さ */
       ( long   , type    ), /* 画像の画素型 */
       ( char **, data    ), /* 画像のデータポインタ */
       ( unsigned int, img_xsize ), /* 画像の表示幅 */
       ( unsigned int, img_ysize )) /* 画像の表示高さ
*/
```

XDrawImage は、画素型 $type$ とデータポインタ $data$ と大きさ (img_xsize, img_ysize) で示される画像をウィンドウ w の領域 (win_x, win_y)-(win_xsize, win_ysize) 上に表示します。

XCreateImageFromData

```
XImage *XCreateImageFromData
  P7 (( Display *, display ), /* Display */
      ( unsigned int, win_xsize ), /* Window の表示幅 */
      ( unsigned int, win_ysize ), /* Window の表示高さ
*/
      ( long     , type    ), /* 画像の画素型 */
      ( char **  , data    ), /* 画像のデータポインタ
*/
      ( unsigned int, img_xsize ), /* 画像の表示幅 */
      ( unsigned int, img_ysize )) /* 画像の表示高さ */
```

XCreateImageFromData は、画素型 $type$ とデータポインタ $data$ と大きさ (img_xsize, img_ysize) で示される画像を、大きさ (win_xsize, win_ysize) の XImage 型に変化してそのポインタを返します。

表 10.2: CIL で定義されているフォント名

フォントの種類	フォント名 (後ろの数字はサイズ)
ゴシック	g2, g3, g4, g5, g6, g7, g8
特徴フォント	f7, f8, f9, 10
4 方向	dir4-3, dir4-5, dir4-7
8 方向	dir8-5, dir8-7, dir8-9
16 方向	dir16-9
32 方向	dir32-17

XImageFontSet

```
char *XImageFontSet
  P1 (( char *, name ))
```

XImageFontSet は、 char 型の表示に用いられるフォントをフォント名 $name$ に設定します。フォント名は表 10.2に示された CIL で定義された値のみ有効です。

XImageFontSetSpace

```
void XImageFontSetSpace
  P2 (( long, xspace ),
      ( long, yspace ))
```

XImageFontSetSpace は、文字と文字のすき間を $xspace$, $yspace$ にそれぞれ設定します。

第 11 章

CIL クライアント紹介

この章では、 CIL で書かれたアプリケーションについて説明します。これらのアプリケーションのうち、 imagedisp は CIL の中でもっとも重要な位置にあります。

- 画像表示コマンド imagedisp
- 簡易画像表示コマンド simpledisp
- 画像エディタ imageedit
- 画像情報表示コマンド imageinfo
- 複数画像観察コマンド imageobserve
- 画像ファイル・フォーマット変換 ifconv
- ラベル画像変換 patchimage
- ラベル画像ラベル描写 drawlabel
- PS フォーマット変換 mkepsf
- CIL サーバ cilserver 関係
- ボクセル表示コマンド voxeldisp

11.1 画像表示コマンド imagedisp

imagedisp は、画像表示するためのコマンドです。また、 imagedisp は、 cilserver コマンドのクライアントとして起動することもできます。このコマンドには、キーボードなどによっていくつかの操作ができます。また、いくつかの濃度変換、拡大縮小、近傍値表示などができ、画像を**観る**ためのツールです。

11.1.1 オプションの説明

引数

- `<input>...`

  表示したい画像ファイル名を指定します。複数指定することが可能です。

一般オプション

- `-h`

  ヘルプメッセージを表示します。

- `-d`

  共有メモリのデバッグモードです。通常は使用しません。

- `-v`

  処理の過程を表示します。

- `-D <display-name>`

  表示する X Window のディスプレイ名を指定します。

- `-k <name>`

  cilserver で、共有メモリや X Window の管理に使用されるキーを指定します。複数指定する場合は、‘"’(ダブルクオート) で囲いスペースで区切って指定して下さい。

- `-r`

  白黒、濃淡値などを反転してから表示します。

```
Usage:  imagedisp  [options] [input]... [options]
    input <image>... : name of the input image.
    -h               : print help message.
    -d               : debug mode for shared memory.
    -v               : verbose mode.
    -D <display>     : name of the X window's display.
    -k <name>        : key of the imagedisp.
    -r               : reverse image.
    -ws <int> <int>  : x,y size of the window.
    -p <float>       : size of the pixel (1.0).
    -px <float>      : X size of the pixel (1.0).
    -py <float>      : Y size of the pixel (1.0).
    -o <int> <int>   : x,y offset of the top of image (0,0).
    -s <int> <int>   : x,y size of the view region.
*) display gamma correction setting
    -g <float>  : gamma of display for correction [#r,#g,#b] (1.0).
    -lut <file> : name of lookup table file.
*) display property setting
    -gp <float> : maximum of gray level [all,#].
    -fp <float> : maximum of feature level [all,#].
    -lp         : label image.
    -cp <float> : maximum of color label level [all,#].
*) For char image.
    -f <font>        : name of the font (g5).
            Gothic ................. g2, g3, g4, g5, g6, g7, g8
            Feature ................ f7, f8, f9, 10
            Four Directions ......... dir4-3, dir4-5, dir4-7
            Eight Directions ........ dir8-5, dir8-7, dir8-9
            Sixteen Directions ...... dir16-9
            Thirty-Two Directions ... dir32-17
    -spc <int>       : space of the font (1).
    -u               : change Char to UChar.
    -c               : change UChar or Bit1 to Char.
    -xys <int> <int> : x,y space of char font (1,1).
*) other options.
    -T : set tile window position.
    -C : set center window position.
```

ウインドウ・サイズ、ピクセル・サイズ

· `-ws <xsize> <ysize>`

表示するウインドウのサイズを指定します。

· `-p <size>, -px <xsize>, -py <ysize>`

1 画素あたりのディスプレイ上でのピクセルの大きさを画素単位で指定します。 0 より大きい実数値を指定することができます。初期値は 1.0 です。 `-px`, `-py` オプションは、 x と y を別々に指定します。

表示領域

· `-o <xoffset> <yoffset>`

· `-s <xsize> <ysize>`

画像の表示範囲を指定します。

ディスプレイ特性とガンマ変換

· `-g <float>`

ガンマ変換の γ の値を指定します。カンマ, で区切って指定すると r, g, b それぞれの γ の値を指定することができます。

· `-lut <filename>`

ディスプレイ特性のルックアップテーブルのファイルを指定します。

画素型の意味づけ

· `-gp <float>`

指定した画像を濃淡画像として表示します。引数には濃淡画像のレベルの最大値を指定します。

· `-fp <float>`

指定した画像を特徴画像として表示します。引数には特徴画像の最大値を指定します。

- -lp
  指定した画像をラベル画像として表示します。

- -cp <float>
  指定した画像を色付けされたラベル画像として表示します。引数にはラベル画像の最大色数を指定します。

char 画像について

- -f <font-name>
  char 画像の表示のときの、フォント名を指定します。フォント名はフレームフォント f7，f8，f9，f10、アスキーキャラクタのフォント g2，g3，g4，g5，g6，g7，g8、4 方向フォント dir4-3，dir4-5，dir4-7、8 方向フォント dir8-5，dir8-7，dir8-9、16 方向フォント dir16-9、32 方向フォント dir32-17 です。初期値は g5 です。
- -u
  キャラクタ画像を濃淡画像として表示します。

- -c
  濃淡画像をキャラクタ画像として表示します。

- -spc <space>
  文字間のスペースを指定します。

- -xys
  文字間のスペースを x, y 別々に指定します。

ウィンドウの表示位置

- -T
  複数のウィンドウをタイル状に並べます。これは初期値です。

- -C
  ウィンドウを中心に重ねて表示します。

11.1.2 使い方の説明

```
key operations.
        q/Q ... quit (all)
        r ... redraw
        R ... reverse
        t ... liner scale translate (toggle)
        e ... histogram translate (toggle)
        o ... original image
        I ... infomation of image
        ? ... print this message
        c ... load image
        S ... silence mode
        V ... verbose mode
        v ... small region's value display
        w ... scaling window mode
        p ... scaling pixel mode
        i ... scaling image mode
        d ... scaling defined window size mode
        1-9 ... scale image/window
        SHIFT + 1-9 ... 0.? scale pixel/window
        h/H ... move left
        j/J ... move down
        k/K ... move up
        l/L ... move right
        f ... resize window to fit image
        s ... scaling image to fit window
        F ... display property is feature
        G ... display property is gray
        N ... display property is label
        C ... display property is color label
        Button1 ... print pixel infomation
        Button2 ... move to point (rel)
        Button3 ... move to point (abs)
```

11.2 簡易画像表示コマンド simpledisp

imagedisp は、画像表示するためのコマンドです。このコマンドは、 X Window 上に自動的に整列して表示します。

11.2.1 オプションの説明

```
Usage:  simpledisp  [options] [input]... [options]
    input <image>... : name of the input image.
    -h               : print this message.
    -t <title>...    : name of the window.
    -g <geometry>    : geometry of the top window
                       <XSIZE>x<YSIZE>[+-]<X>[+-]<Y>.
    -c <int>         : number of the window column.
    -s <int*2>       : x,y space of the window
                       <XSPACE>x<YSPACE> (5x5).
    -rev             : reverse image.
    -r               : display on the root window.
    -R               : display on the whole root window.
```

引数

· <input>...

表示したい画像ファイル名を指定します。複数指定することが可能です。

オプション

· -h

ヘルプメッセージを表示します。

· -rev

白黒、濃淡値などを反転してから表示します。

· -t <title>...

ウィンドウのタイトルを指定します。

- -g <xsize>x<ysize>[+-]<x>[+-]<y>

  最初のウィンドウの座標を指定します。サイズはすべてのウィンドウに対して有効です。また、 ¡xsize¿ のみ指定した場合は、 ¡ysize¿ は原画像のサイズの比で計算されます。

- -c <column>

  ウィンドウを並べる横の数を指定します。

- -s <xspace>x<yspace>

  ウィンドウとウィンドウのスペースを指定します。ウィンドウマネージャによっては正しくスペースが反映されないようです。

- -r

  表示をルートウィンドウにします。

- -R

  表示をルートウィンドウ一杯にします。画像ファイルを複数指定した場合は最初の画像のみ有効となります。

11.2.2 操作の説明

- q,Q

  小文字の q は、そのウィンドウだけ閉じます。大文字の Q は、すべてのウィンドウを閉じて終了します。

11.3 画像エディタ imageedit

imageedit は、画像を編集するためのコマンドです。 Bit1, UChar, UChar3 型の画像に基本図形を描いたりすることができます。非常にバグが多いためこまめにセーブして編集することを勧めます。特に難しいところはないので、とりあえず使ってみることです。

11.4 画像情報表示コマンド imageinfo

imageinfo は、画像の情報を表示するためのコマンドです。画像フォーマット、型、サイズ、画素値の最大値、最小値、平均、そして、ヒストグラムを見ることができます。これらの情報がテキストで見れることがこの特徴です。

11.5 複数画像観察コマンド imageobserve

imageobserve は、複数の画像を同時に観察するためのコマンドです。このコマンドには、キーボードなどによっていくつかの操作ができます。また、複数の画像に対して、同じ位置の拡大縮小、近傍値表示などができ、画像を**観察**するためのツールです。特に、複数の画像に対して、同じ位置の画素の値を見るときに威力を発揮します。

11.5.1 オプションの説明

```
Usage:  imageobserve  [options] [入力]... [options]
    入力 <画像>...   : 入力画像のファイル名.
    -h               : ヘルプメッセージを表示する.
    -s <整数>        : 観察ウィンドウに表示する画素の大きさ (8).
    -f <フォント名> : 値表示におけるフォントの指定 (a14).
*) ウィンドウの表示位置に関するオプション.
    -T : ウィンドウをタイル状に並べる（初期）.
    -O : ウィンドウを重ねて表示する.
    -C : ウィンドウを中心に並べる.
    -R : 観察窓を表示ウィンドウの右に並べる（初期）.
    -B : 観察窓を表示ウィンドウの下に並べる.
```

引数

· <input>...

表示したい画像ファイル名を指定します。複数指定することが可能です。

一般オプション

· -h

ヘルプメッセージを表示します。

- -s

  観察ウィンドウの１画素の大きさを指定します。初期値は 8 倍です。

- -f

  画素値表示におけるフォント名の指定します。X Window で用いることのできるフォント名が使用できます。初期値は a14 です。

ウインドウの表示位置

- -T

  複数のウィンドウをタイル状に並べます。これは初期値です。

- -O

  ウィンドウを重ねて、左上から右下へ少しずつ移動しながら表示します。

- -C

  ウィンドウを中心に重ねて表示します。

- -R

  観察ウィンドウを表示ウィンドウの右に並べて表示します。これは初期値です。

- -R

  観察ウィンドウを表示ウィンドウの下に並べて表示します。

11.5.2 使い方の説明

キー操作一覧.
●拡大領域関連
s ... すべての画像に対して移動の操作を同期します.
v ... 拡大表示において数値をテキストで表示します.
1-9,0,-,=, ... 画像を拡大します.
●移動関連
h/H ... 拡大領域を左に移動（大文字は大きく移動）
j/J ... 拡大領域を下に移動（大文字は大きく移動）
k/K ... 拡大領域を上に移動（大文字は大きく移動）
l/L ... 拡大領域を右に移動（大文字は大きく移動）
Button1 ... 領域の移動
（領域をドラッグして移動することも可能）
Button3 ... 領域の中心座標を表示する（toggle）
●描写関連
r ... 再描写します
R ... 濃度値を反転します
t ... 濃度値を線形変換します（toggle）
o ... 原画像に戻します
F ... 特徴画像としてスペクトル表示します
G ... 濃淡画像として濃淡表示します
N ... ラベル画像として境界線表示します
C ... カラーラベル画像としてカラー表示します
●その他
? ... ヘルプメッセージを表示します
Q ... すべて終了します
q ... ウィンドウを閉じます
I ... 画像の情報を表示します

11.6 画像ファイル・フォーマット変換 ifconv

ifconv は画像ファイル・フォーマットを変換するためのコマンドです。

11.6.1 使用法

```
ifconv <format> [input] [output]
```

画像フォーマットは、 C2D, PNM, J4, TIFF, XWD, XBM, JPEG, GIF, PS, DIB が指定できます。入力画像および出力画像が省略されたときは、標準入力および標準出力に入出力されます。

11.7 ラベル画像変換 patchimage

patchimage は、原画像とラベル画像と用いて画像を加工します。各領域を原画像の領域内の濃度値の平均値で塗りつぶしたり、境界線を描いたりします。

11.7.1 オプションの説明

```
Usage:  patchimage  [options] <original> <label> <output> [options]
    original <image> : name of the input color or gray image.
    label <image>    : name of the input label image.
    output <image>   : name of the output patched image.
    -h               : print this message.
    -s <float>       : scale of the image size (1.0).
    -a               : use the mean of intensity in a region.
    -nb              : no bound.
    -nm              : use minus regions as normal regions.
    -nu              : use uncertain regions as normal regions.
    -bm <uchar>      : intensity of the boundaries (255).
    -bc <r> <g> <b>  : color of the boundaries (255,255,255).
    -mm <uchar>      : intensity of the minus regions (-1).
    -mc <r> <g> <b>  : color of the minus region (-1,-1,-1).
    -um <uchar>      : intensity of the uncertain regions (-1).
    -uc <r> <g> <b>  : color of the uncertain regions (-1,-1,-1).
```

引数

· <original>

加工したい原画像ファイル名を指定します。

· <label>

加工したいラベル画像ファイル名を指定します。

· <output>

出力画像のファイル名を指定します。

オプション

· -h

ヘルプメッセージを表示します。

- `-s <scale>`

  出力画像を $scale$ に応じて拡大縮小します。

- `-a`

  領域を濃度の平均値で塗りつぶします。このオプションが指定されていない時は、領域内は原画像のままになります。

- `-nb`

  このオプションが指定されると領域の境界線を描写しません。

- `-nm`

  領域のラベルがマイナスの場合においても通常の領域として扱います。

- `-nu`

  領域のラベルがゼロの場合においても通常の領域として扱います。

- `-bm <uchar>, -bc <red> <green> <blue>`

  境界線の色を指定します。

- `-mm <uchar>, -mc <red> <green> <blue>`

  ラベルがマイナスの領域の色を指定します。

- `-um <uchar>, -uc <red> <green> <blue>`

  ラベルがゼロの領域の色を指定します。

11.8 ラベル画像ラベル描写 drawlabel

drawlabel は、ラベル画像に文字のラベルを自動で描写します。このコマンドは、ルックアップテーブルを用いて、ラベル番号に対応するテキストをその領域に描写することもできます。基本的に各領域の距離画像を作成して境界から最も遠い位置にラベルを書きます。

11.8.1 オプションの説明

```
Usage:  drawlabel  [options] <input> [options]
    input <image> : name of input label image.
    -o <image>    : name of output labeled image (labelno.out).
    -lut <text>   : name of lookup table file.
    -s <float>    : scale for output image (1).
    -fn <font>    : name of label font (a14).
    -a <int>      : threshold for region area (200).
    -h            : print this messages.
```

引数

· <input>

ラベルを描写したいラベル画像ファイル名を指定します。

オプション

· -h

ヘルプメッセージを表示します。

· -o <output>

出力画像の名前を指定します。省略した場合の出力ファイル名は labelno.out になります。

· -lut <lookuptable>

ラベルのルックアップテーブルファイルを指定します。ルックアップテーブルは、

<ラベル番号> <ラベル>

のように記述され、先頭が # の場合はコメントとしてスキップされます。また、ラベルにスペースを含むテキストを指定したい場合はダブルクオーテーションでくくります。

· -s <scale>

出力画像を拡大縮小します。フォントはこの影響を受けません。

· -fn <fontname>

X Window で指定されるフォント名を指定します。初期値は a14 です。

· -a <area>

ラベルを描写する領域の面積の最小値を指定します。領域の面積がこの値より小さい場合はラベルを描写しません。初期値は 200 です。

11.9 PS フォーマット変換 mkepsf

mkepsf は、画像を eps ファイルに変換します。

11.9.1 オプションの説明

```
Usage:  mkepsf  [options] <input> <output> [options]
    input <image>   : name of the input image.
    output <epsf>   : name of the output epsf file.
    -h              : print this message.
    -s <size/scale> : image size [<XSIZE>x<YSIZE>] / [<SCALE>].
    -t <uchar>      : threshold and output a binary image (128).
    -R              : reverse image.
    -f              : write frame.
    -F <uchar>      : intensity of the frames (0).
Followings are used by the long label image.
    -b          : borders superior to uncertain regions.
    -B <uchar> : intensity of the borders between regions (0).
    -C <uchar> : intensity of the certain regions [L>0] (255).
    -U <uchar> : intensity of the uncertain regions [L=0] (150).
    -P <uchar> : intensity of the previous regions [L<0] (230).
Followings are used by the ushort or short image.
    -ub : ushort image is binary image.
    -ul : ushort image is long-label image.
```

引数

· <input>

変換したい入力画像ファイル名を指定します。

· <output>

出力画像の名前を指定します。

オプション

· -h

ヘルプメッセージを表示します。

- -s <xsize>x<ysize>, -s <scale>

  出力画像のサイズ、もしくは、スケールを指定します。

- -t <threshold>

  閾値処理をして出力します。

- -R

  入力画像の濃度値を反転します。

- -f

  出力画像の枠を描写します。

- -F <frame>

  出力画像の枠の値を指定します。初期値は 0 です。二値画像の時はこの値を 1 として指定して下さい。

ラベル画像に関するオプション

- -b

  ラベルの値が 0 の領域に対しては、境界線を優先して描写します。

- -B <uchar>

  境界線の濃度値を指定します。初期値は 0 です。

- -C <uchar>

  ラベルが正の値の領域内部の濃度値を指定します。初期値は 255 です。

- -U <uchar>

  ラベルがゼロの領域内部の濃度値を指定します。初期値は 150 です。

- -P <uchar>

  ラベルが負の領域内部の濃度値を指定します。初期値は 230 です。

ushort 画像に関するオプション

- `-ub`

  ushort 画像を二値画像として扱います。

- `-ul`

  ushort 画像をラベル画像として扱います。

11.10 CIL サーバ cilserver 関係

cilserver は、 imagedisp コマンドと共有メモリや ImageDisp 関数群を用いたアプリケーション間で通信するためのサーバです。共有メモリを使った一種のファイルサーバだと思って下さい。ここではサーバ以外に、サーバにコマンドを送る cilcp, cills, cilrm, cilview などについても説明します。

11.10.1 サーバ cilserver

cilserver は、共有メモリに格納された画像を管理するサーバです。共有メモリを使った一種のファイルサーバだと思って下さい。 CIL **では、ファイル名の先頭に @ マークがある場合、共有メモリ上のファイルとしてアクセスします。**

```
Usage : cilserver [-uid <uid>] [-r] [-c] [-h] [-f]
        -h   : print this messaged
        -uid : user ID
        -r   : reset server
        -c   : clear cil table
        -f   : free cil table
```

- `-h`

  ヘルプメッセージを表示します。

- `-uid <uid>`

  ユーザ ID を指定して起動します。ルートのみ有効です。

- `-r`

  サーバを起動し直します。

- `-c`

  共有メモリ画像の管理テーブルをクリアします。

- `-f`

  共有メモリ画像の管理テーブルを解放してサーバを終了します。

11.10.2 リストコマンド cills

cills は共有メモリにある画像をリストします。

```
Usage : cills [-a] [-l] [-h]
        -a : all user information.
        -l : all system information.
        -h : print this messages.
```

- -h

  ヘルプメッセージを表示します。

- -a

  ユーザに必要な情報を詳細に表示します。

- -l

  共有メモリ上にある画像についてシステムの持っている情報をすべて表示します。

11.10.3 コピーコマンド cilcp

cilcp は共有メモリにある画像をコピーします。

```
Usage : cilcp [input] [output]
```

引数は、コピー元、コピー先の順で指定します。これらは、共有メモリ上からファイルシステム上、またはその逆が可能です。

11.10.4 削除コマンド cilrm

cilrm は共有メモリにある画像を削除します。

```
Usage : cilrm <name> ...
        cilrm <id> ...
        cilrm all
```

引数には、画像の名前、 ID、 all が指定できます。

11.10.5 表示コマンド cilview

cilview は共有メモリにある画像を表示します。

```
Usage : cilview [input]
```

引数は、画像の名前を指定します。

11.11 ボクセル表示コマンド voxeldisp

voxeldisp は、ボクセル表示するためのコマンドです。このコマンドには、キーボードとパネルによっていくつかの操作ができます。また、いくつかの濃度変換、拡大縮小、近傍値表示などができ、ボクセルを**観る**ためのツールです。 voxeldisp の環境は図 11.1に示してあります。オプションの指定の時など参考にして下さい。

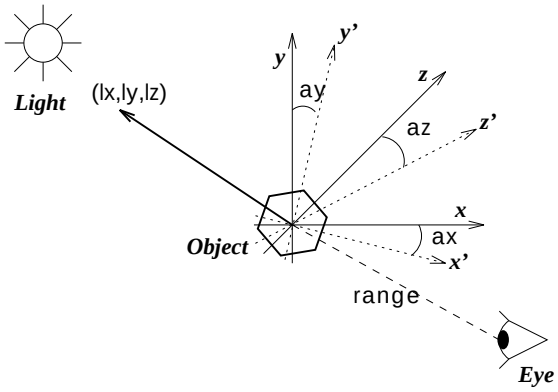


図 11.1: voxeldisp の環境

11.11.1 オプションの説明

```
Usage:  voxeldisp  [options] <input> [options]
    input <voxel>     : name of input voxel image.
    -a <ax> <ay> <az> : angles of rotation[degree] (0,0,0).
    -l <lx> <ly> <lz> : vector of the light (-0.2,-1,1.5).
    -n                : no use the light effection.
    -s <float>        : scale of a voxel size (5).
    -r <float>        : range between object and view point (10).
    -h                : print this messages.
```

引数

· <input>

表示したいボクセルファイル名を指定します。

一般オプション

- `-h`

  ヘルプメッセージを表示します。

- `-a <ax> <ay> <az>`

  ボクセルの回転角度を指定します。初期値は $(0, 0, 0)$ です。

- `-l <lx> <ly> <lz>`

  光の方向ベクトルを指定します。指定したベクトル (lx, ly, lz) 方向に光源が設定されます。初期値は $(-0.2, -1, 1.5)$ です。

- `-n`

  光を使いません。

- `-s <scale>`

  1 画素のボクセルの一辺の大きさを指定します。

- `-r <range>`

  対象物と視点の距離を指定します。初期値は 10 です。