構造体とポインタ

構造体とポインタ

構造体

構造体は、単純変数、配列や構造体などを組み合わせて、まとまった構造として扱うための変数の集まりです。**構造体**の宣言・定義には **struct** キーワードを使います。

さっそく実例を見てみます。座標を表す構造体を作ってみましょう

```
struct point {
        int x;
        int y;
};
```

構造体の中に定義されている変数名をメンバーと呼びます。ここで注意して欲しいのは、この文では変数の定義をしていないということです。つまり、ここでは構造体の型の定義を行い、その型を point と名付けているだけなのです。決して、point という変数が使用可能になっているわけではありません。ここで point のように構造体の型を表す識別子を**構造体タグ**と呼びます。**構造体タグ**は省略可能です。構造体タグがあると、定義した構造体の型をその名前で使うことができます。

```
struct point    a, b;
```

上では、point 構造体の変数として、a と b とを定義しています。構造体変数 a のメンバーを使うときは、a.x や、a.y のように . を使って表記します。構造体タグが省略された時の使い方も示しておきましょう。

```
struct {
        int x;
        int y;
} a, b;
```

構造体タグが無い場合の変数の定義は、下を見てもらうと分かると思いますが、int などのような基本データ型の変数定義と同じ形をしています。

1. struct { ... } a, b;
2. int a, b;

つまり、struct { ... } の部分で1つの型を表しています。その型を名前で使えるようにしたものが、構造体タグというわけなのです。

構造体の変数を定義すると同時に初期化を行うこともできます。

```
struct point {
        int  x;
        int  y;
}  a = { 12, 34 },  b = { 56, 78 };
```

```
struct point z;
z = a;   /* 構造体の代入 */
```

最後の行は、代入の例です。

Typedef

ここで、型の別名を定義できる **typedef** 文を紹介します。

```
typedef 型 別名;
```

単純な型の言い換えだと、#define と、どこが違うのか分かりづらいかもしれません。ここで、typedef は、#define のように「別名 型」という並びではないことに着目して下さい。これは、変数定義の時の変数名の場所に別名を置く、というのがポイントです。配列の型の別名を定義してみると、よく分かります。

```
typedef int SquareMatrix[10][10];
SquareMatrix a;          /* 10×10 の int 型配列 */
```

構造体でもやってみましょう。

```
/* 座標構造体 struct point */
struct point {
        int x;
        int y;
};
```

```
/* 座標構造体の別名 Point */
typedef struct point Point;

/* どちらも同じ */
struct point a;
Point b;
```

struct の定義と同時に typedef を使うこともできます。

```
/* 座標構造体 struct point */
typedef struct point {
        int x;
        int y;
} Point;

/* 構造体タグを省略した形 */
typedef struct {
        int x;
        int y;
} Point;
```

typedef を使うことにより、コードをより見通しの良いものにすることができます。次は、10×10 の double 型配列を 2 つ受け取り、同じ型のポインタを返す関数のプロトタイプです。

```
double (*(add_matrix)(double mat1[10][10], double
```

```
typedef double SqMat[10][10];
SqMat *add_matrix(SqMat mat1, SqMat mat2);
```

どちらが見やすいかは、一目瞭然です。

関数と構造体

構造体では、構造体を単位とした代入の操作が可能です。これは、配列と違って、代入により全体がコピーされるということを意味しています。ただし、比較演算はできません。理由はいくつかあります。ひとつはポインタが含まれている場合、アドレスは異なっているがポインタが指す値は等しいといった場合、これを等しいと見なすか、あるいは異なっていると見なすか、実装者にとっては選択の余地があるということです。どちらも「等しい」あるいは「異なっている」とする相応の理由があり得るので、比較の定義をどちらか一方のみにすることが難しいのです。

別の理由として、構造体メンバーの配置には、実装に依存する部分があるため、変数間に余白が生じる可能性がある、ということが挙げられます。次の例を見てみましょう。

```
struct point {
        char name[10];
        int  x;
        int  y;
};
```

何の変哲もない構造体の定義ですが、32 ビットマシンの上でコンパイルすると、メモリ配置は次のようになる可能性があります。

char [0,1,2,3]	
char [4,5,6,7]	
char[8,9]	
int	
int	

32 ビットは多くのマシンで 4 バイトとして表されており、メモリをアクセスする際には、CPU の処理ビット数単位で区切られた境界でアクセスするようにコンパイラで調整されます。これは、処理ビット境界をまたがないように配置されると高速にメモリアクセスできる場合が一般的だからです。

上の例では、char の後ろに配置される int は、char が 10 バイトだと、きちんと境界に配置されませんので、配置をそろえるよう余白部分が作られます。ここで、代入の場合は、この余白部分は使われることがないので、コピー元からメンバーと余白部分全体をコピーすることで問題は無いのですが、比較の時には、メンバ以外の余白部分を含めて比較すると問題が起こります。したがって、構造体同士を比較する場合は、構造体の全てのメンバーごとに比較を繰り返す必要があります。

さて、代入できるということは、関数の引数としてアドレス渡しではなく実体を渡せたり、逆に関数から戻り値として受け取ったりできるということになります。実例を見てみましょう。

```
struct point {
        int  x;
        int  y;
};

struct point getPoint(int x, int y)
{
```

```
        struct point temp = { x, y };
        return temp;
}
```

int 型の引数を 2 つとり、struct point 型の構造体を返す関数の例です。この関数を、

```
struct point pnt = getPoint(12345, 67890);
```

のように呼び出すと、pnt は struct point 構造体として初期化されます。このとき、関数内で生成される temp は自動変数で、関数の終了とともに消滅するのですが、pnt の実体は temp とは異なりますので、ポインタを使ったときのように、関数の終了と共に実体が失われるという事態とはなりません。これは、C 言語で構造体の代入がサポートされているからです。

配列を含む構造体に代入が行われたとき、配列はコピーされます。

```
struct str_wrapper {
        int len;
        char str[1024];
}  sw1, sw2;

strcpy(sw1.str, "abcdefg");      /* char[1024] に文
sw2 = sw1;       /* sw2 に代入（コピー） */

/* 文字列を受け取り struct str_wrapper 構造体を返す */
struct str_wrapper setstr(char *str)
{
        struct str_wrapper strstruct;
        strstruct.len = strlen(str);
        strcpy(strstruct.str, str);
        return strstruct;
}
```

ただし、関数の戻り値として構造体の実体を返すやり方は、よく考えて使うようにして下さい。非常に大きな領域の構造体の場合、関数呼び出しや戻り値をやりとりする時に用意されている領域を使い切って、実行時エラーになる場合があります。そのような場合は、ポインタによるアドレス渡しでやりとりするのが一般的です。

演習問題 4-1

2 つの複素数を受け取り、足しあわせた結果値を返す関数を書いて下さい。

- 関数プロトタイプは次の通りです。

  ```
  Complex enshu_4_1(Complex num1, Complex num2);
  ```

- ファイル名を enshu_4_1.c として下さい。
- 複素数を表す構造体の型を、以下のように定義して使って下さい。

  ```
  typedef struct {
          double real;
          double imaginary;
  }  Complex;
  ```

- メンバー real に実数部分が、imaginary に虚数部分が格納されます。

- 演算を行った結果を格納している構造体を返します。

演習問題 4-2

2 つの複素数を受け取り、かけあわせた結果値を返す関数を書いて下さい。

- 関数プロトタイプは次の通りです。

```
Complex enshu_4_2(Complex num1, Complex num2);
```

- ファイル名を enshu_4_2.c として下さい。
- 複素数を表す構造体の型を、以下のように定義して使って下さい。

```
typedef struct {
        double real;
        double imaginary;
}  Complex;
```

- メンバー real に実数部分が、imaginary に虚数部分が格納されます。
- 演算を行った結果を格納している構造体を返します。

演習問題 4-3

文字列の先頭からアルファベットを探索し、最初に見つけた単語を返す関数を書いて下さい。

- 関数プロトタイプは次の通りです。

```
Wordtab enshu_4_3(char *text);
```

- ファイル名を enshu_4_3.c として下さい。
- 単語を表す構造体の型を、以下のように定義して使って下さい。

```
typedef struct {
        char *wordptr;
        char *endptr;
        char word[16];
}  Wordtab;
```

- 探索する単語は、アルファベットが1文字以上連続している部分で、アルファベットが連続する範囲が 1 単語となります。
- wordptr に単語の先頭アドレスをセットします。
- endptr に単語の末尾の次のアドレスをセットします。つまり、単語の先頭から見て、最初にアルファベット以外の文字が現れた位置を表します。
- word には、単語をコピーします。末尾にはヌル文字 '\0' をセットしなければなりません。
- 従って、word にセットされる単語は、最大 15 文字長です。endptr は、最大長と関係なく、単語の末尾の次のアドレスをセットします。

- アルファベットかどうか調べる関数は `int isalpha(int)` を使います。アルファベットであれば、真となります。
- `isalpha()` は `ctype.h` で宣言されています。
- 単語が見つからなかった場合は、`wordptr`、`endptr` とも `NULL` をセットします。`word` には、長さ `0` の文字列をセットします。
- 引数の `text` ポインタが `NULL` である場合も、文字が見つからなかった場合の動作となります。
- 演算を行った結果を格納している構造体を返します。

構造体へのポインタ

ポインタは、単純データ型より、構造体と組み合わせて使われることが多いです。構造体へのポインタを整理しておきましょう。

```
struct point {
    int x;
    int y;
};
```

として構造体を宣言します。このとき、次の宣言

```
struct point *pp;
```

は、`pp` が `struct point` という型の構造体へのポインタということを表しています。ポインタは、実体を指し示す付箋のようなもので実体が必要となりますので、実際に構造体の中身を定義して実体を作り、ポインタで指し示すことにしましょう。

```
struct point origin = { 123, 456 };
struct point *pp = origin;
```

`origin.x` は `123` で、`origin.y` は `456` なので、`(*pp).x` が `123`、`(*pp).y` が `456` となります。構造体へのポインタはよく使われるため、簡便にコードを書けるよう `->` 演算子が用意されています。

上記の例で言うと、`(*pp).x` というコードは `pp->x` と書き直すことができます。ポインタが指す変数値を間接指定する単項演算子 `*` と、構造体のメンバにアクセスする演算子 `.` では、`.` 演算子の方が優先順位が高いため、`*` 演算子をかっこでくくる必要がありますので、記述が複雑になる場合もあります。構造体へのポインタはよく使われるため、コードが見やすくなるように、`->` 演算子が導入されたというわけです。以下の例をご覧下さい。

```
1.  (*(*(*pAccessControl).pCredentials).pPAM).p
2.  pAccessControl->pCredentials->pPAM->pUserInfo
```

`1.` と `2.` では、同じ内容を表すコードですが、`1.` では、かっこと `*` 演算子がきちんと対応しているかどうか、チェックするのも大変だということが分かると思います。

`->` 演算子は左から右へと結合します。結合順序をかっこでくくると、以下のようになります。ここでは、結合順序を強調するために、かっこを使った記法を示しましたが、通常の左から右の結合で使うときは、かっこでくくる必要はありません。

```
(((pAccessControl->pCredentials)->pPAM)->pUserInf
```

-> 演算子はよく使われるので、使い方に慣れておきましょう。

演習問題 4-4

文字列ポインタを含む構造体へのポインタ配列を引数にとり、ポインタ配列を構造体の文字列順でソートする再帰関数を書いて下さい。

- 関数プロトタイプは次の通りです。

```
void enshu_4_4(WordHolder wh
[], int lpos, int rpos);
```

- ファイル名を enshu_4_4.c として下さい。
- 文字列ポインタを含む構造体は、以下の定義を使います。

```
typedef struct {
        int count;
        char *word;
}  WordHolder;
```

- 配列のソート範囲は、lpos から rpos まで(それぞれの添字要素を含む)となります。
- 文字列は辞書順に整列するものとします。
- 文字列の比較には strcmp() を使ってもよいです。ctypes.h で宣言されています。
- strcmp() では、文字列が同じ時に、0 が、小さいときには負の数が、そして、大きいときには 0 より大きい数が返されます。int strcmp (const char*, const char*) として宣言されています。

ヒント クイックソートのアルゴリズムを使ってみましょう。

メモリのアロケーション

ここまでの説明で、ポインタによるオブジェクトの操作の準備ができました。それでは、メモリを扱う関数を説明しましょう。今までポインタの説明をしてきたのは、このメモリのアロケートを行うための準備だと言ってもよいくらいです。

```
void *malloc(size_t size);

void free(void *ptr);
```

size_t というのは、sizeof 演算子が返す符号なし整数型のことです。**void *** 型は、任意のポインタ型にキャストできる総称的なポインタ型で、ライブラリ関数などで実際に使われる型を用いずにポインタを表現する場合に使われます。stdlib.h でプロトタイプ宣言されています。

自動変数は、関数に局所的な変数で、関数から抜けるとその実体も消滅するのでした。**malloc()** と **free()** はこの自動変数のためのメモリ割り当てと同じような働きをします。**malloc()** は **free()** と対で使用されるもので、関数の自動変数の領域とは違うメモリ領域を確保して、free() で領域を解放するまで定常的にメモリを使

用できるようにするためのものです。このように動的に割り当てて確保できるメモリ領域は**ヒープ領域**と呼ばれます。

自動変数で起きる、関数から抜け実体が消えてしまったものを使おうとして起こるバグとは逆に、ヒープメモリの利用では、割り当てられたものを解放し忘れ、もう使われないにも関わらず居残っているという状態があります。使われないメモリ領域が残っているだけでは、ポインタアクセス時のエラーが必ずしも起きるわけではありませんが、何度も繰り返しメモリが割り当てられることで、使えるメモリリソースが減少して、OS の実行にも影響を及ぼすような状況に陥ることもあります。こういった意図しないメモリ領域の管理ミスで、解放し忘れのメモリ領域が増えることを**メモリリーク**といいます。

それでは、コードで実例を見てみます。

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

int main()
{
        char *text = "The quick brown fox jumps o

        /* コピー元の長さを得る */
        int textlen = strlen(text);

        /* メモリアロケート */
        char *buff = malloc(textlen + 1);
        if (buff != NULL) {
                /* アロケートしたメモリにコピー */
                strcpy(buff, text);

                /* 新しい領域を使って処理 */

                /* 領域を開放 */
                free(buff);
        }
        else {
                /* エラー処理 */
                }
        }
```

malloc() でヒープ領域が確保されたアドレスを、buff ポインタに保持します。何らかの理由で malloc() が失敗すると、NULL が返りますので、その場合は領域確保に失敗した場合の処理を書きます。作業が終わったら、free() を呼んでヒープ領域を開放します。malloc() は void * 型を返すため、ポインタへの代入に際しては暗黙のキャストが行われ、 明示的にキャストする必要はありません。

文字列の領域を確保するときは、文字列の長さより 1 以上大きい領域をとらなければいけません。終端を示すヌル文字 \0 が必要だからです。

演習問題 4-5

文字列ポインタの配列を受け取り、バッファ領域で文字列を連結する関数を書いて下さい。

これは、演習問題 3-8 の動的メモリアロケート版です。

- 関数プロトタイプは次の通りです。

  ```
  char *enshu_4_5(char **argv);
  ```

- ファイル名を enshu_4_5.c として下さい。
- argv は文字列ポインタの配列です
- 文字列ポインタが NULL になると、それ以上の文字列がないということを示します。
- 連結するときに、スペースや改行を追加する必要はありません。

- 文字列は、ヌル文字 `\0` で集結する必要があります。
- 演算を行った結果を格納する構造体を動的に割り当てて、関数はそのポインタを返します。
- `malloc()` が `NULL` を返した場合は、そのまま関数値として `NULL` を返して下さい。

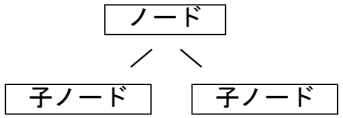
ヒント 最初から割り当てるべきメモリ量はわからないので、ループを 2 回行うとよいでしょう。

自己参照のポインタを持つ構造体

これまで、文字列を処理するコードをいくつも見てきましたが、どれも共通して「文字列を配列に格納し、文字列をポインタでアクセスする」という方式だったと言えそうです。これは、ポインタが便利だから、というよりはむしろ、配列ではなく動的にメモリを確保する場合、どうしてもポインタによるオブジェクトの使用ということが不可避だからである、ともいえるでしょう。ところが、実行時にファイルを読み込むなど、外部からのデータを扱う場合には、配列の長さをどれだけにすればよいのかがコンパイル時に決定できないことから、配列による表現には限界があります。

ヒープメモリを使えば、実行時に必要なだけメモリを確保できるようになります。演習問題 4-5 では、まさにこの動的にメモリを確保するという方法で、演習問題 3-8 にあった問題を解決しています。しかし、長さを求めるためにループを 2 回繰り返す方法では、似たような処理を繰り返さなければならない、という問題が発生しています。これは、データが長くなってきたり、データを処理するのに外部記憶とのやりとりが行われるなど、ループにかかるコストが無視できない場合には、重要な問題となるでしょう。

これから見ていくのは、データの処理中にその構造が動的に拡張していく、という基本的なデータ構造です。配列にも、伸張可能な配列はありますが、配列は直列に配置しなければならないため、これ以上伸張できない、という制限が起こりやすいのです。



まずは図のような、**ツリー構造**と呼ばれる、複数の**ノード（頂点）**を互いに**リンク（線）**で結んだデータ構造を見ていきます。ツリー構造は、根本にルートノードと呼ばれる 1 つのノードがあり、それ以外のノードは他のノードやリンクをたどっていくことで到達できます。図のツリー構造は、根本が上で、下に向かって成長していきます。また、任意のノードにたどり着くリンクの経路は、1 通りで、これはつまり循環する経路が無いと言うことを示しています。

親ノード、子ノードというのは、単に派生する関係を示したもので、どちらかのノードが機能制限されていたり、持っている属性が異なったり、ということはありません。子ノードが子ノードを持つことも可能で、その場合は、子ノードへのリンクを持つノードが親ノードということになります。つまり、相対的な関係性を言っているわけです。

なお、ここでは子ノードの数が2個までのものを扱います。そのようなツリー構造を二分木と呼びます。

さて、二分木のノードを構造体で表すとして、リンクはどのように表せばいいのでしょう？ リンクは構造体の一部でありながら、他の（同じ型の）構造体を指すものとして表現すればよいわけです。ノードは限りなくつながっていく可能性がありますが、全てのノードを統一的に構造体で表現し、それに、リンクとしてのポインタを2つ備え付ければ、図のようなノード間の関係を表現することができます。

あとは、データの要素をメンバーとして持たせると、データを格納できるはずです。約束として、リンクの先がない状態では、ポインタ値を NULL とします。

```
struct treenode {
        char *word;
        int count;
        struct treenode *left;
        struct treenode *right;
};
```

struct treenode の中で、ポインタ型 struct treenode * が出てきますが、これは自分自身を指すものではなくて、ノードを表現する型（型は同じです）の他のオブジェクトに対するポインタということになります。

少し脇道にそれますが、typedef の場合は、 typedef で定義した型名を自己参照のポインタを表すものとして使うことはできません。typedef 文が終わりまで到達していないので、typedef で定義される型名がまだ有効になっていないからです。struct タグ名 * は、構造体のポインタ型だと分かる（ポインタ型のサイズだけが分かれば、この時点でのコンパイルには支障がない）ため、使うことが許されます。

それでは、コードサンプルです。このコードは、単語リストを処理しながらツリー構造に単語と頻度を格納します。

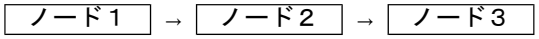
```
#include <stdio.h>
#include <stdlib.h>

typedef struct treenode {
        char *word;
        int count;
        struct treenode *left;
        struct treenode *right;
} TreeNode;


/* ノードを挿入する場所を再帰的に探索してノード追加 */
TreeNode *addTree(TreeNode *p, char *word)
{
        if (p == NULL) {
                /* 新たなノードを作成 */
                p = malloc(sizeof (TreeNode));
                if (p != NULL) {
                        /* 単語、カウンタをセット */
                        p->word = word;
                        p->count = 1;

                        /* リンクポインタを初期化 */
                        p->left = NULL;
                        p->right = NULL;
                }
                else {
                        /* エラー処理 */
                }
        } else {
                /* word を比較 */
                int cond = strcmp(word, p->word);
                if (cond == 0)
                        /* 既に登録されている場所を発見
                        p->count++;
                else if (cond < 0)
                        /* word のほうが小さいので、左
                        p->left = addTree(p->left
                else
                        /* word のほうが大きいので、右
                        p->right = addTree(p->rig
        }

        return p;
}

/* ツリー構造を順序に従って出力 */
void printTree(TreeNode *p)
{
        if (p != NULL) {
                /* 小さい方（左側）を出力 */
                printTree(p->left);
```

```
                printf("%3d:  %s\n", p->count, p-
                /* 大きい方（右側）を出力 */
                printTree(p->right);
        }
}

int main()
{
        /* ツリー構造のルートポインタ */
        TreeNode *root = NULL;

        /* 単語リスト */
        char *txtlist[] = {
                "that", "a", "the", "a", "the", "
        };

        int i;
        for (i = 0; i < sizeof txtlist / sizeof t
                /* 単語をツリーに追加 */
                char *word = txtlist[i];
                root = addTree(root, word);
        }

        /* 木全体を出力 */
        printTree(root);
}
```

このプログラムでは、データの処理とツリーの伸張を同時に進めることができるため、あらかじめデータを前処理する必要がありません。また、データ構造の中にキーを分散させて格納する仕組みを使い、ツリーがソートされている状態で構築されるのも特長です。すなわち、あるノードについて、その `left` のリンク配下にあるデータは、ノードよりも小さいデータというような構造になっています。

リスト

ツリーを構成するノードはリンクを複数個持っていましたが、次のようにリンクを 1 つしか持っていないようなデータ構造を**（単方向）リスト**と呼びます。



単語文字列によるリストのノードの構造体は、次のようになるでしょう。

```
struct listnode {
        char *word;
        struct listnode *next;
};
```

リストは、シンプルで柔軟な構造なのでよく使われます。ポインタ付け替えは比較的コストがかからない処理なので、リスト同士の連結や、リストの途中からノードを削除する、あるいは挿入するといった処理がやりやすいのです。ただし、配列の場合は要素が定間隔で並んでいるので、どの要素に対してもほぼ一瞬でそのアドレスを得ることができますが、リストは要素にたどり着くまで、リンクを最初からたどる必要があり、要素の後ろの方へのアクセスが遅くなるという特性があります。

したがって、リストは配列のように後ろに要素を追加していくよりも、先頭に要素を挿入していく方が効率が良くなります。配列の場合は、先頭に追加するときは、配列の全要素を後ろにずらす作業が必要になるため遅くなります。

演習問題 4-6

文字列ポインタの配列からリストを生成する関数を書いて下さい。

- 関数プロトタイプは次の通りです。

```
WordList *enshu_4_6(char **argv);
```

- ファイル名を enshu_4_6.c として下さい。
- argv は文字列ポインタの配列です。
- 文字列ポインタが NULL になると、それ以上の文字列がないということを示します。
- 関数から返される構造体は、以下定義になります。

```
typedef struct wordlist {
        char *word;
        struct wordlist *next;
}  WordList;
```

- 終端リストでは next を NULL にします。
- 演算を行った結果を格納する構造体を動的に割り当てて、関数はそのポインタを返します。
- malloc() が NULL を返した場合は、そのまま関数値として NULL を返して下さい。

演習問題 4-7

引数で渡されるリストから逆順にしたリストを構築し、そのリスト返す関数を書いて下さい。

- 関数プロトタイプは次の通りです。

```
WordList *enshu_4_7(WordList *wl);
```

- ファイル名を enshu_4_7.c として下さい。
- WordList は演習問題 4-6 のものと同じです。
- 文字列ポインタが NULL になると、それ以上の文字列がないということを示します。
- 関数から返される構造体は、以下定義になります。

```
typedef struct wordlist {
        char *word;
        struct wordlist *next;
}  WordList;
```

- 終端リストでは next を NULL にします。
- 演算を行った結果を格納する構造体を動的に割り当てて、関数はそのポインタを返します。
- malloc() が NULL を返した場合は、そのまま関数値として NULL を返して下さい。
- 引数のリストを書き換えてはいけません。

演習問題索引

演習問題 4-1　演習問題 4-2　演習問題 4-3　演習問題 4-4
演習問題 4-5　演習問題 4-6　演習問題 4-7