第三章　関数とポインタ

3.1　復習

関数を呼び出す際のメモリの使い方を復習します。

例題）

（１）簡単な関数を作る。この関数を呼び出すプログラムを作る。たとえば、引数は int、関数名は power、power の戻り値は引数の２乗。

（２）すべての変数、引数について、それらのアドレスを表示させる。

（３）関数を終了した後、引数の値は元に戻ることを確認する。

(1)

```
int	power(int n){
	return	n*n;
}
main(){
	int	m=3;
	printf("%d¥n",power(m));
}
```

(2)

```
int	power(int n){
	printf("&n = %p¥n",&n);
	return	n*n;
}
main(){
	int	m=3;
	printf("&m = %p¥n",&m);
	printf("%d¥n",power(m));
}
```

実行結果

```
&m = 0x7ffff9f4
&n = 0x7ffff9f0
9
```

(3)

```
int	power(int n){
	int	x;
```

```
        x=n*n;
        n=5;                        /* 引数を５に変更   */
        return   x;
}
main(){
        int        m=3;
        printf("%d¥n",power(m));
        printf("%d¥n",m);
}
```

実行結果

9

3

上の例題（２）で分かるように、引数として入力された変数の値は、「新たな変数」にコピーされています。関数の定義文の中で引数を変更するとこの「新たな変数」の値を変更したことになります。そのため、関数を呼び出した前後で、引数として用いられた変数の値は変わりません。

３．２　ポインタ変数を引数とする関数

次の例題は重要な示唆を含んでいます。

例題）変数の値を変更するプログラム。

```
int        fff(int *n){                /* 関数 fff の引数は、int へのポインタ型変数 */
        *n=4;
}
main(){
        int        m=3;
        fff(&m);
        printf("%d¥n",m);
}
```

実行結果

4

練習問題 3.2.1

上記例題の実行結果について、なぜこうなるかを説明せよ。分からない者は、納得行くまで議論、検討せよ。今までの知識を動員すれば、容易に説明可能である。

練習問題 3.2.2

二つの int 型変数の値を入れ替える関数 swap を作成せよ。文献等を参照せずに、自力で考えて解答せよ。

```
void    swap(int *a,int *b){
        考える
}
main(){
        int n=3,m=4;
        swap(&n,&m);
        printf("%d %d ¥n",n,m);
}
```

３．３　配列を引数として受け渡す方法

　C言語では、配列そのものを関数の引数として受け渡すことはできません。以下のように、配列名を関数に引き渡して、その代わりにしてきました。

例題）配列名を受け取り、受け取った配列の成分の和を返す関数

```
int add(int a[2])
{
        return a[0]+a[1];
}
main()
{
        int             b[2];
        b[0] = b[1] = 1;
        printf("%d¥n",add(b));
}
```

上のプログラムでは、成分数２の int 型配列の配列名（配列自身ではない）を関数 add が受け取る仕様になっています。さて、第２章で学んだ規則

```
        a[i]=*(a+i)
```

を思い出してください。

練習問題 3.3.1

上の規則をつかって、return a[0]+a[1]　の部分を書き換えよ。

補足）配列名を引数として使う際の書き換え規則

上の例題で、
int add(int a[2])の代わりに　int add(int *a)　と書くことができます。

この規則は第２章で学んだ
　　配列名＝配列の先頭成分へのアドレス
を思い出せば、自然な感じがします。

例題）

```
int add(int *a)
{
        return a[0]+a[1];
}
main()
{
        int                b[2];
        b[0] = b[1] = 1;
        printf("%d¥n",add(b));
}
```

実はこのプログラムは、以下のように書き換えても全く同じです。

```
int add(int a[])
{
        return a[0]+a[1];
}
main()
{
        int                b[2];
        b[0] = b[1] = 1;
        printf("%d¥n",add(b));
}
```

つまり、int add(int a[2])を int add(int *a)とか int add(int a[])に書き換えても良く、配列の成分数２は無視されます。

練習問題 3.3.2

（１）char*型配列（成分７）week を宣言せよ

（２）７つの文字列"Sunday","Monday","Tuesday","Wednesday","Thursday","Friday", "Saturday"をそれぞれ配列 a,b,c,d,e,f,g に格納する。２種類の方法があるが方法２でやること。

方法１：配列の宣言の際に初期化する。

例）char a[7]="Sunday", b[7]="Monday", ...

方法２：配列に文字列を格納する関数 string_copy を作成し、この関数を用いて格納する。

例）

```
void string_copy(char *new, char *old){
各自
}
main(){
        char a[7];
        string_copy(a,"Sunday");
}
```

（３）week[0]=&a[0],week[1]=&b[0],...,week[6]=&g[0]とせよ。

（４）ユーザが入力した０から６までの数字に対応する曜日（例えば０は Sunday）を画面に表示する。ただし、上で定義した配列 week を利用すること。

練習問題 3.3.3

上の練習問題プログラムを、配列 a～g を用いずに作成せよ。ただし、配列 week を利用すること。

３．４　関数の戻り値として配列を返す方法

Ｃ言語では関数の戻り値として配列を返すことは残念ながらできません。しかし、ポインタ型変数を返すことで、その代わりにします。

例題）以下の関数を作成せよ。

関数名：swapped_array
引数：成分数２の int 型配列の配列名 a
戻り値：int へのポインタ型変数。
副作用：a[0]と a[1]が入れ替わる

```
int * swapped_array(int a[2]){
        int     tmp;
        tmp=a[0];
        a[0]=a[1];
        a[1]=tmp;
        return  a;                      /*      a は&a[0]の同義語       */
}
main(){
        int     c[2]={1,2};
        int     *b;
        printf("%d %d¥n",c[0],c[1]);
        b=swapped_array(c);
        printf("%d %d¥n",b[0],b[1]);   /*  実は b の値は c(=&c[0])
                                            であることに注意     */
}
```

実行結果
1 2
2 1

練習問題 3.4.1
上のプログラムで printf("%d %d¥n",b[0],b[1]);を＊演算子を用いた表現に書き換えよ。

練習問題 3.4.2
上のプログラムの main 文を次のように書き換えると実行結果は同じにならない。この理由を説明せよ。

```
main(){
        int     a[2]={1,2};
        printf("%d %d¥n",a[0],a[1]);
        printf("%d %d¥n",swapped_array(a)[0],swapped_array(a)[1]);
}
```

練習問題 3.4.3

上のプログラムの main 文を次のように書き換えるとコンパイルでエラーになる。この理由を説明せよ。

```
main(){
        int     a[2]={1,2};
        int     b[2];
        printf("%d %d¥n",a[0],a[1]);
        b=swapped_array(a);
        printf("%d %d¥n",b[0],b[1]);
}
```

上の練習問題から

配列名には値を代入できないが、ポインタ型変数には値を代入できる

と言うことが分かる。

教訓：配列名をポインタ型変数と混同するな（使い方は似ていても実体は違う）

練習問題 3.4.4

以下の関数を作成せよ。動作チェックもやること。

関数名　：　pickstr

引数　：　char s[],int n

戻り値　：　s[n]のアドレス

副作用　：　なし

関数名　：　dishead2

引数　：　char s[]

戻り値　：　s 中最初に現れる、”ブランク以外の文字”のアドレス

練習問題 3.4.5

プログラム中で文字型配列に適当な文字列を代入し、さらにその内容を表示するプログラムを作れ。但し、先頭にブランクが在る場合はそれを読み飛ばして表示する事。上で作成した dishead2 を利用する事。

練習問題 3.4.6

以下の関数を作成せよ

関数名　：　Dayname

引数　　：　int 型変数 n
戻り値　：　n が 0 から 6 までのとき、対応する曜日を表す文字列の先頭アドレス
（例えば n=0 のとき、char 型配列”Sunday"の先頭アドレス）を返す。

本章のまとめ
（１）ポインタ型変数を関数の引数にする方法について
（２）ポインタ型変数を返す関数について
（３）配列名とポインタ型変数は、扱い方は似ているが、異なるモノである。

１章から３章までで
　　ポインタ型変数の概念
　　ポインタ型変数の使い方
　　配列とポインタ型変数
　　関数とポインタ型変数
を学んだ。これらの知識を総動員してプログラムを作ってみよう。

練習問題 3.4.7
キーボードから英文を入力すると、この英文を単語に分割し、線形リストに格納する関数 save_sentence を作成せよ。引数は char * 型変数、戻り値は、先頭の構造体へのポインタ(struct ?*型変数)とする。
練習問題 3.4.8（チャレンジ・２分探索木）
練習問題 3.4.7 の関数 save_sentence を用いて２分探索木のプログラムを作成する。まず次の構造体 node を作る。

```
struct   node{
        char        word[20];
        int   count;
        struct node *left;
        struct node *right;
};
```

キーボードから入力された英文（大文字）を単語に分割し、struct node 型構造体に格納したい。まず、以下の説明を読み、次に次頁の図を見て、プログラムを作成せよ。

(1) word 成分には単語を格納する。count には出現頻度を記録する

(2) 最初に入力された単語（次頁の例では TO）を struct node 型変数に格納する。この node（仮に TO ノードと呼ぶ）には以下のような値が記録されている。

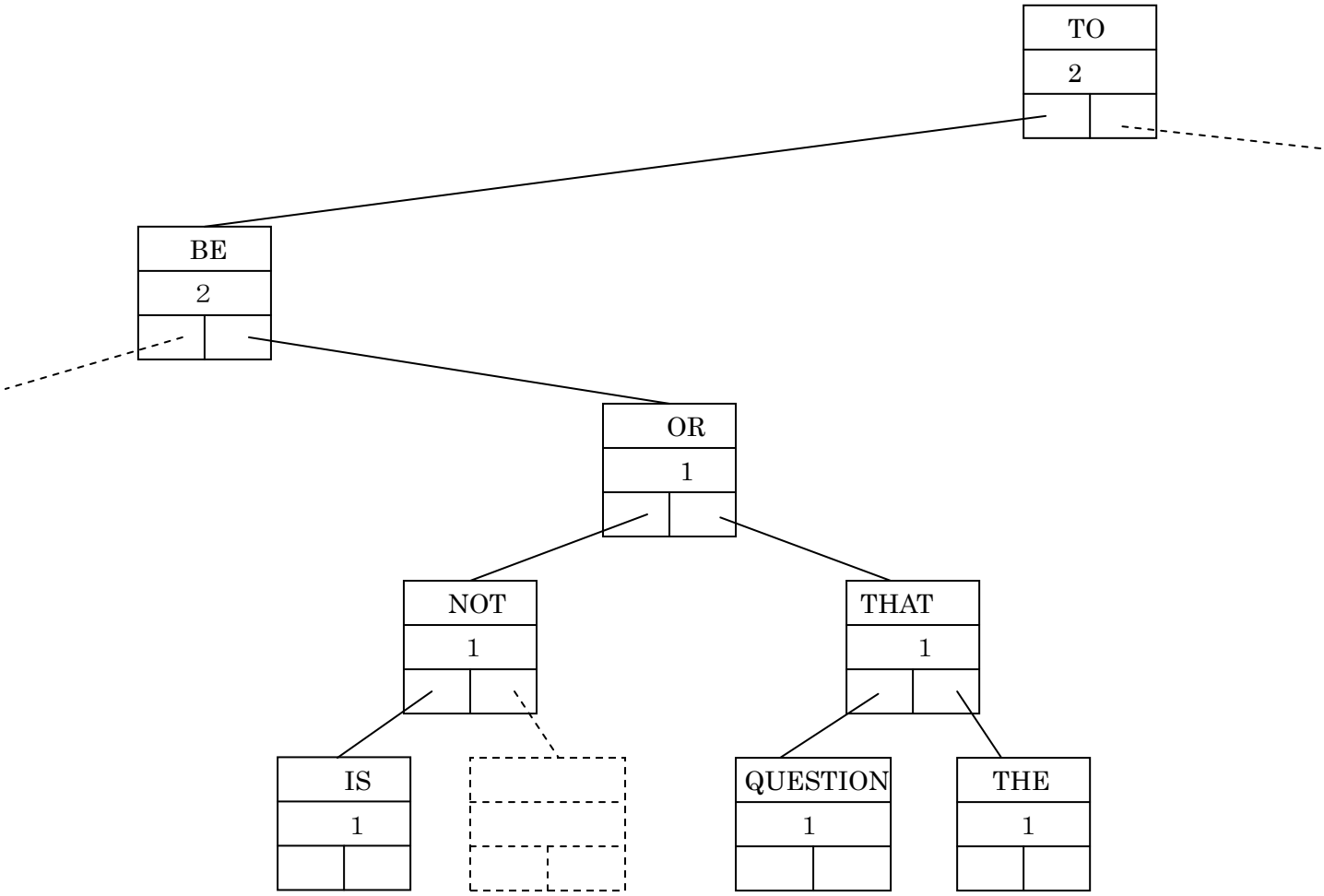
word　"TO"
count　1
left　NULL すなわち(struct node *) 0
right　NULL すなわち(struct node *) 0

(3) 2 番目に入力された単語（BE）が、辞書式順序で言って、一番目の単語（すなわち TO）より前なので、left エントリに次の node (仮に BE ノードと呼ぶ) のアドレスを記入する。BE ノードは当然、calloc 等で確保する。確保した BE ノードには count に 1 、left および right に NULL を入れておく。

(4) 3 番目に入力された単語（OR）について OR ノードを以下の手順で作成する。
① OR は TO より前なので、まず、TO ノードの left を参照して、BE ノードに移動。
② OR は BE の後で、しかも、BE ノードの後に来るノードは無いので、BE ノードの次に OR ノードを作成する。

(5) 4 番目以降に入力された単語についても(4)と同じ要領で作成する。既に入力された単語が再び入力された場合には count に 1 を足すだけでよい。

例文として TO BE OR NOT TO BE THAT IS THE QUESTION を考えると、以下のような概念図となる。このように各節の左のデータは必ずその節のデータよりも小さく、右のデータは必ずその節のデータよりも大きい 2 分木を 2 分探索木という（下の例の場合、データの大小は辞書式順序での大小を意味している)。



付録：知識の整理と若干の補足事項

１．　データ構造に関する知識

計算機内部やプログラムにおいて、データを効率的に記憶したり操作するためのデータの表現をデータ構造と言います。以下、今まで学んだデータ構造について簡単におさらいしておきます。

１．０　基本データ型

C 言語における int 型や char 型等のデータ型も、広い意味ではデータ構造ですが、これら基本データ型については２年までに学んでいるので説明を省きます。各データ型の変数のサイズがどの程度かを調べる方法をすでに学んでいます。

１．１　配列

複数の同じ型の変数をまとめて扱うデータ構造。添え字を利用できるので便利です。

１．２　構造体

複数の変数をまとめて扱うデータ構造ですが、配列と違い、扱う変数の型は異なっても構いません。構造体に含まれる変数をメンバと呼びます。下の例で構造体のメンバは gakuseki, gakka, kokugo,suugaku です。また、構造体を識別する名称 seiseki を「構造体タグ」と言います。

例　struct seiseki{

```
        int gakuseki;
        char gakka;
        int kokugo;
        int suugaku;
    }
```

なお、struct seiseki　型の変数の宣言は

```
        struct seiseki wada;
```

という形で、他の変数型と同様の書式で行い、宣言した変数のメンバを参照する際は

```
        wada.kokugo
```

という書式を使います。

１．３　リスト（線形リスト）

複数個のデータに対して、これらを列としてつなげて記録したデータ構造です。具体的には、データ A(1), A(2), A(3), ...に対して、各データ A(k)にデータ A(k+1)の在処（アドレス）を含めて記録します。

C 言語では､構造体をポインタによってつなぎ､これをリストと見なす事がよくあります。新しくデータを追加する際は calloc 関数や malloc 関数などで記憶領域を新しく確保します。

リストにはいくつかの種類があります。或るデータ A(k)と次のデータ A(k+1)が互いに参照可能なように､「お互いに」そのアドレスを所有しているリストもあります。この種のリストを双方向（両方向）リストといいます。双方向リストに対して､「次のデータ」の在処のみを所有している単純なリストは単方向（一方向）リストといいます。そして、最後のデータが最初のデータを知っているリスト構造を環状リストと言います。

一般にリストは、最初もしくは最後にデータを追加する場合は一定時間ですみますが、リストの途中にデータを挿入しようとすると時間がかかります。と言っても、配列では物理的にデータをずらす必要があるのに対して、線形リストはポインタのつなぎ換えだけで良いので、配列よりも高速です。

また、データの参照もリストの最初もしくは最後は一定時間ですむのですが、リストの途中のデータにアクセスしようとすると時間がかかります。

リストの練習問題を既にいくつか解きました。save_sentence で作ったリストは記憶に新しいと思います。

１．４　木

木とは、木のように分岐した形をしたデータ構造です。根(ルート)というデータが１つあり、その下に複数の枝が伸びています。それぞれの枝に節(ノード)というデータが付いていますが、木の末端には節が付いていない枝があり、これを葉と呼びます。

C 言語では、節を構造体で表現し、枝は他の構造体へのポインタ、葉は NULL ポインタで表現されることがよくあります。

以下に、木に関する重要な用語について簡単に説明します。

・二分木（バイナリツリー）

根や節の下に節や葉が「２つ以下」しかつかない木。完全二分木のことを指す場合もあります。

・完全二分木

最下層の葉（ルートから最も遠い葉）以外は、すべて葉や枝を２つずつ持っています。最下層の葉が偶数枚の場合は、各節から２枚ずつ葉を出しており、奇数枚の場合は、一つの節を除いてすべて２枚ずつ葉を持ちます。最下層の葉は、左側に集中しています。２月の小テストに出題した木は、完全二分木です。

・二分探索木

各節の左のデータは必ずその節のデータよりも小さく、右のデータは必ずその節のデータよりも大きい二分木です。データを二分探索木に格納することで、探索の時間を飛躍的に減らすことがあります（どの程度減らせるかは重要)。また、ポインタのプリントの練習問題 3.4.8 も二分探索木です。二分探索木を用いてデータを探索する関数の例を以下に記します。

例：二分探索木を用いてデータを探索する関数　search_data

```
typedef struct data{
        int       key;
        struct data       *left;
        struct data       *right;
} node;

void print_data(node *attension){
        if (attension==NULL) return ;
        else {
                print_data(attension->left);
                printf("%d¥n",attension->key);
                print_data(attension->right);
        }
}
```

```
NODE *search_data(NODE *attension,int data){
        if (attension==NULL || attension==data) return attension;
        else {
                if ((attension->key) < data)
                        search_data(attension->right,data);
                else    search_data(attension->left,data);
        }
}
```

１．５　スタック

データを出し入れする方式のうち「最後に記憶されたデータを最初に取り出す」という方式を　LIFO (Last In First Out) といいます。LIFO という方式を実現するデータ構造をスタックと言います。

上で紹介した線形リストや木は、物理的なメモリー構造でした。これに対し、スタックは物理的なメモリー構造ではなく、用途で分けたデータ構造です。つまり、LIFO を実現しているデータ構造は皆スタックと呼んでも良く、線形リストを利用したスタックや配列を利用したスタックというものがあります。

スタックの要素数が不確定の場合はリストを使って実現することが多いようです。

スタックに情報を積むことを「push する」と言い、情報を取り出すことを「pop する」と言います。スタックは練習問題 2.4.7 で学びました。

１．６　キュー

スタックでは追加した順とは逆の順で取り出すデータ構造でした。 キューはこのように最初に入れた物は最後に取り出すという LIFO（例えばスタック）とは逆順のデータ構造です。すなわち、最初に入れたデータは最初に取り出す FIFO (First In First Out)の構造です。入れた順番と同じ順番で取り出すことができます。

スタックと同様、キューも物理的なメモリー構造ではなく、用途で分けたデータ構造です。キューは練習問題 2.4.8 で学びました。

1.　7　静的データ構造 static data structure)と動的データ構造 (dynamic data structure)

処理過程を通して不変なデータ構造を静的データ構造と言います。静的データ構造は変数名によって参照します.

一方、処理の途中で構造が変化するデータ構造を動的データ構造と言います。随時 calloc 関数などで領域が割り当てられます。

２．　C 言語プログラムに関する知識の整理と補足

２．１　キャスト

式の型を変換する演算子。(データ型)式　という書式で使います。

２．２　ポインタ

メモリ領域のアドレスを格納する変数

２．３　再帰呼び出し

関数定義文で、自分自身を呼び出している関数を「再帰的に定義された関数」と言い、自分自身を呼び出している動作を再帰呼び出しと言います。

２．４　局所（ローカル）変数と大域（グローバル）変数

プログラムの中のもっとも外側のブロックで宣言された変数をグローバル変数と言います。大域変数とも呼ばれます。プログラム全体に対して定義される変数で、プログラムのどこからでも参照・更新することができます。

これに対して、宣言されたブロック内だけで有効な変数をローカル変数、または局所変数と言います。C 言語ではグローバル変数と同名のローカル変数が存在した場合は、ローカル変数が優先されます。

２．５　typedef

既存のデータ型に新しい名前をつけるときに使います。

２．６　#define

ある文字列を別の文字列で置き換えたいときに使います。

（書き方）

#define　文字列１　文字列２　　（文字列１ を文字列２で置き換える）

２．７　#include

#include は指定したファイルを読み込みたいときに使います。

（書き方）

#include　<標準ヘッダファイル>

#include　"自作のヘッダファイル"

標準ヘッダファイルに納められている関数を標準ライブラリ関数と言います。

２．８　文字列定数

引用符“で囲んだ文字列を文字列リテラルまたは文字列定数といいます。文字列定数は書き換え不可ですので注意してください。以下のプログラムは char c1[5]="abcd"　という記述で配列 c1 に文字列を入力しています。さらに char *c2="abcd"　では、文字列定数の先頭アドレスを c2 に入れています。前者は文字列定数を扱う命令でないことに注意してください。このプログラムの実行結果はどうなるでしょうか。考えてみてください。

```
void string_copy(char a[],char b[]){
int i=0;
        while(b[i]!=0){
        a[i]=b[i];
        i++;
```

```
            }
a[i]='¥0';
}

main(){
char c1[5]="abcd";
char *c2="abcd";
char c3[5];

c1[1]='x';
printf("%s¥n",c1);

c2[1]='x';
printf("%s¥n",c2);

string_copy(c3,"abcd");
c3[1]='x';
printf("%s",c3);
}
```

２．９　演算子の優先順位

演算子（*　や　->　など）には優先順位があります。10+3*5 と(10+3)*5 は意味が異なります。詳しくは優先順位表(../manual/yusen.pdf)を見てください。