

# プログラミング演習Ⅱ レポート

課題名： 動的データ構造

学籍番号	212273B
氏名	高木 壱哲
提出日	令和 5 年 6 月 8 日
書式修正版提出日	令和 年 月 日

【チェック項目】

以下の項目が正しく記載されているか確認し，○をつけること．※がついているものは必須項目ではない．

項目		自己 チェック	担当記入欄	
			判定	コメント
(1) 表紙に必要な事項を記入したか？				
(2) 目的を記入したか？				
課題 1 の各項目を記入したか？	(3) 概要			
	(4) 文字列へのポインタ配列			
	(5) データ構造			
	(6) アルゴリズム			
	(7) ソースリスト			
	(8) 実行結果			
	(9) 考察			
課題 2 の各項目を記入したか？	(10) 概要			
	(11) 可変長文字列リスト			
	(12) データ構造			
	(13) アルゴリズム			
	(14) ソースリスト			
	(15) 実行結果			
	(16) 構造体のサイズ			
(17) 考察課題				
(18) 参考文献の書式は正しいか？				
(19) フローチャートの書き方は正しいか？				
(20) 図番号とタイトルをつけたか？				
(21) ページ番号をつけたか？				
※ オプション課題				
※ 感想				

【教員記入欄】

## 1. 目的

C 言語における静的メモリの管理について、動的メモリ管理を利用したリスト処理について、typedef 宣言についてそれぞれプログラムの作成を通して学ぶ。

## 2. レポート課題 1

### 2.1 概要

動的メモリ管理を用いてキーボードからの入力行単位で辞書順にソートし、出力するプログラムを作成する。

### 2.2 文字列へのポインタ配列

ソースコードを記述する段階ではメモリに割り当てる変数の大きさを決定できない場合、実行時に動的にメモリ空間を確保して使用する場合がある。これを動的メモリ管理と呼ぶ。逆に、ソースコード記述時に、通常通りメモリに割り当てる変数の大きさを決定し確保して使用することを静的メモリ管理という。

文字列の長さに合わせて動的にメモリ領域を確保するには `malloc` 関数か `calloc` 関数を用いる。`malloc` 関数を用いる場合について説明する。`malloc` 関数は引数の値バイトのメモリを割り当て、そのアドレスを返し、確保に失敗した場合は `NULL` を返す関数である。`malloc` 関数のプロトタイプ宣言は `void *` 型であるが、これはどんな型のポインタでも格納できる万能なポインタを表す。よってコーディングする際には適切なポインタ変数に代入し使用することとなる。

文字列の長さに合わせたメモリ空間を確保するには `malloc` 関数の引数に `strlen` 関数を用いて確保した文字数に 1 を加えた値をおく。これは文字列配列の最後に `NULL` 文字が入るためである。

### 2.3 データ構造とアルゴリズム

#### 2.3.1. データ構造

(1) 定数宣言

`MAXLINES` : 最大の行数

`LINELENGTH` : 1 行の最大文字数

(2) 型宣言

`static char *lines[MAXLINES]` : 文字列へのポインタの配列 (静的ローカル変数)

(3) グローバル変数

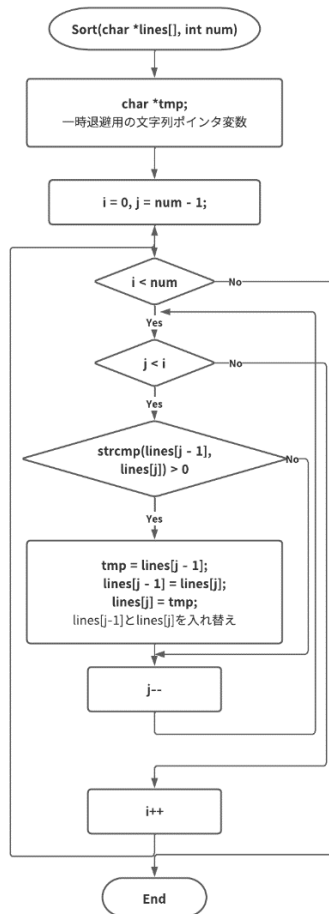
なし

`char *lines[MAXLINES]` について、`lines` はこの配列の先頭アドレスを示すポインタである。

#### 2.3.2. アルゴリズム

以下の図 1 に `Sort` 関数のフローチャートを示す。

図 1 Sort 関数のフローチャート



Sort 関数ではまず、入れ替えを行うため一時退避用の文字列ポインタ変数 `tmp` を用いる。`for` 文を 2 回回して配列の最後から比較する。`strcmp` を用いて、`j-1` 番目の文字列の文字コードが `j` 番目の文字列の文字コードより大きいとき（辞書順と逆になっているとき）、`tmp` を用いて入れ替えを行う。これを繰り返し、`for` 文が終了次第この関数も終了する。

## 2.4 ソースリスト

文字列の Sort 関数のソースリストを以下のリスト 1 に示す。

```

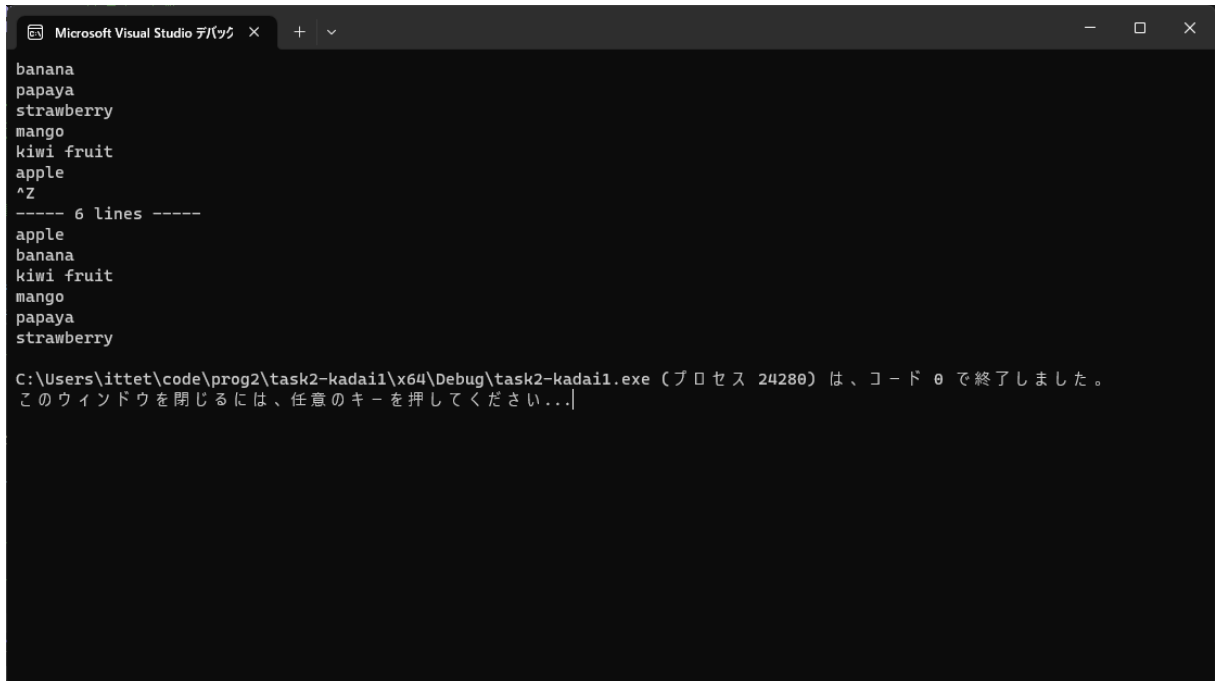
1: void Sort(char *lines[], int num)
2: {
3:     char *tmp;
4:     int i, j;
5:     for (i = 0; i < num - 1; i++)
6:     {
7:         for (j = num - 1; j > i; j--)
8:             if (strcmp(lines[j - 1], lines[j]) > 0)
9:             {
10:                 tmp = lines[j - 1];
11:                 lines[j - 1] = lines[j];
12:                 lines[j] = tmp;
13:             }
14:     }
15: }
```

2.3 で示したように、`for` 文を 2 回回し、`strcmp` を用いて文字列を比較し、辞書順にソートして行く関数である。

## 2.5 実行結果

実行結果を以下の図 2 に示す。

図 2 レポート課題 1 の実行結果



```
Microsoft Visual Studio デバッグ
banana
papaya
strawberry
mango
kiwi fruit
apple
^Z
----- 6 lines -----
apple
banana
kiwi fruit
mango
papaya
strawberry

C:\Users\ittet\code\prog2\task2-kadai1\x64\Debug\task2-kadai1.exe (プロセス 24280) は、コード 0 で終了しました。
このウィンドウを閉じるには、任意のキーを押してください...
```

入力した 6 つの文字列が辞書順にソートされ出力されていることがわかる。しかし、この状態でデバッグを行うとメモリリークが起こっているとわかる。

これを解消するため、動的に確保したメモリを解放する関数 **FreeLines** を追加する。ソースを以下のリスト 2 に示す。

```
1: void FreeLines(char* lines[], int num)
2: {
3:     int i;
4:     for (i = 0; i < num ; i++)
5:     {
6:         free(lines[i]);
7:     }
8: }
```

この関数では **lines** の要素全てについて **free** 関数を実行し、メモリを開放している。この関数を、動的に確保したメモリを使い終えた **main** 関数の最後に挿入し、メモリを開放できるようにしてデバッグを実行したのが以下の図 3・4 である。

図3 レポート課題1 FreeLines 導入後の実行結果

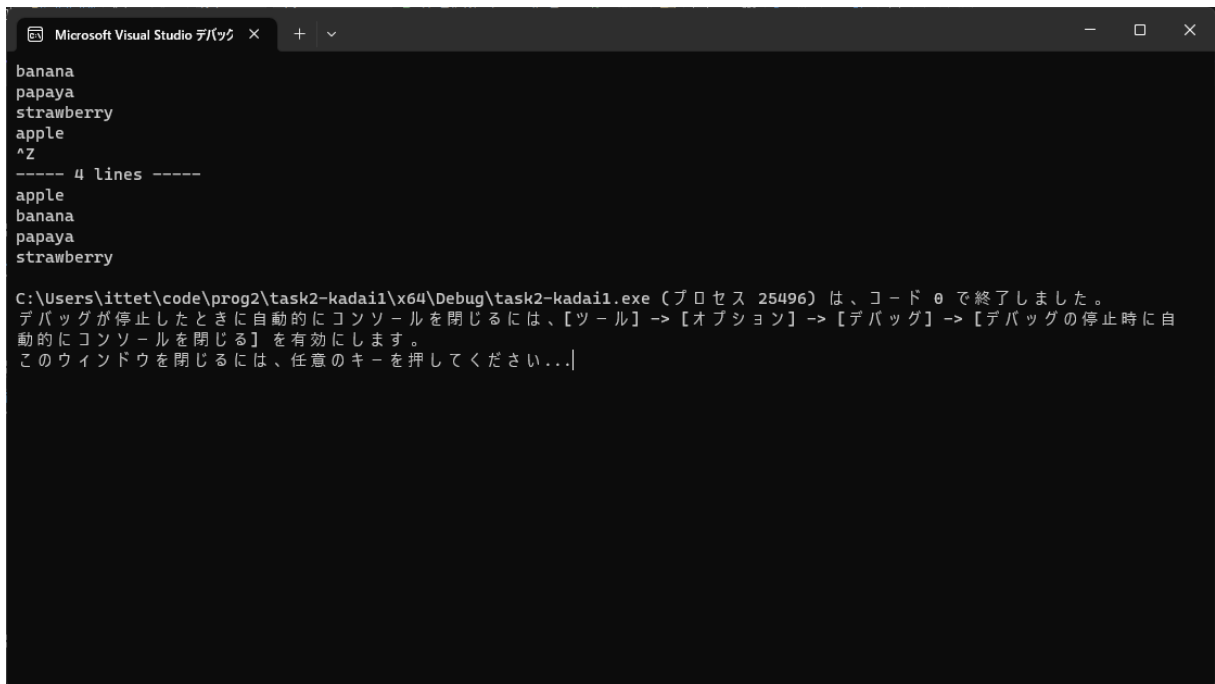


図 4 レポート課題 1 FreeLines 導入後のデバッグ実行結果

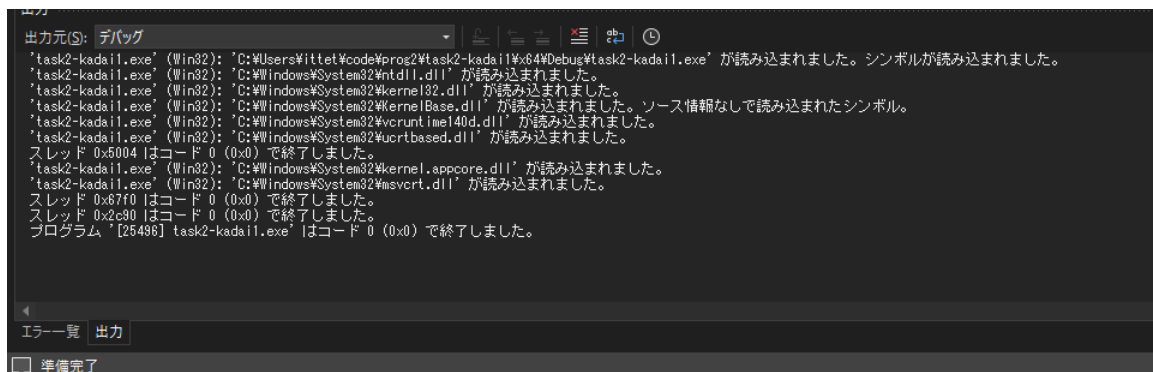


図 3 よりプログラムが正常に動いていることが、図 4 よりメモリリークが発生していないことがわかる。**CrtDumpMemoryLeaks** 関数をプログラム終了直前に記述することで、メモリリークが発生しているか調べることができ、発生していた場合デバッグ実行結果に「**detected memory leaks!**」と表示される。

## 2.6 考察

Sort 関数においてバブルソートを実装した。この際、`strcmp` 関数を用いると昇順・降順に並び替えるのがともに便利である。`FreeLines` 関数についてはすべての要素に対して `free` 関数を実行する必要があるため、必要十分な実装ができた。

### 3. レポート課題2

### 3.1 概要

動的メモリ管理を用いて単方向リストプログラムを実装する。授業中の課題 2 で作成した固定長文字列でのプログラムを修正し、可変長文字列に対応できるようにする。

## 3.2 可変長文字列版リスト

可変長文字列版リストについて、これは構造体において文字列を格納する部分に文字列配列のアドレスを格納しておき、必要な長さの文字列配列を作成し、そのアドレスを通してその文字列を参照するというものである。固定長文字列版リストでは予め最大の文字数を決めておく必要があり、これは無駄が生じたり思い通りに出力されなかったりする場合がある。可変長版であれば文字列配列を作ることができる最大のサイズまでの文字数を格納でき、必要な分のみメモリを確保するためムダがない。

## 3.3 データ構造とアルゴリズム

### 3.3.1. データ構造

```
typedef struct cell{ char *string, struct cell *next; } CELL, *PCELL;
```

: 連結リストの 1 つのセルを宣言

Cell 型はリストのセルをあらわす構造体を宣言しており、Pcell 型は Cell を表すポインタ型として typedef を用いて表されている。typedef 宣言を用いることで、存在するデータ型に新たな名前をつけることができる。構造体の名前付け、ポインタの作成を用意にするためなどに用いられることがある。

### 3.3.2. アルゴリズム

以下の図 5 に ListInsert 関数・ListDelete 関数・ListDestory 関数のフローチャートを示す。

図 5 レポート課題 2 のフローチャート

図

## 3.4 ソースリスト

ListInsert 関数・ListDelete 関数・ListDestory 関数のソースリストを以下のリスト 3 に示す。

リスト 3 レポート課題 2 のソースコード

```
1: PCELL ListInsert(PCELL pos, const char *string)
2: {
3:     PCELL pNewCell;
4:     pNewCell = calloc(1, sizeof(CELL));
5:
6:     if (pNewCell != NULL)
7:     {
8:         // ここを実装する
9:         // 新しいセルへの文字列をコピーおよび
10:        // ポインタの付け替えを行う
11:        pNewCell->string = (char *)calloc(strlen(string) + 1,
sizeof(char));
12:        strcpy(pNewCell->string, string);
13:        pNewCell->next = pos->next;
14:        pos->next = pNewCell;
15:    }
16:    return pNewCell;
17: }
18:
19: void ListDelete(PCELL pos)
```

```

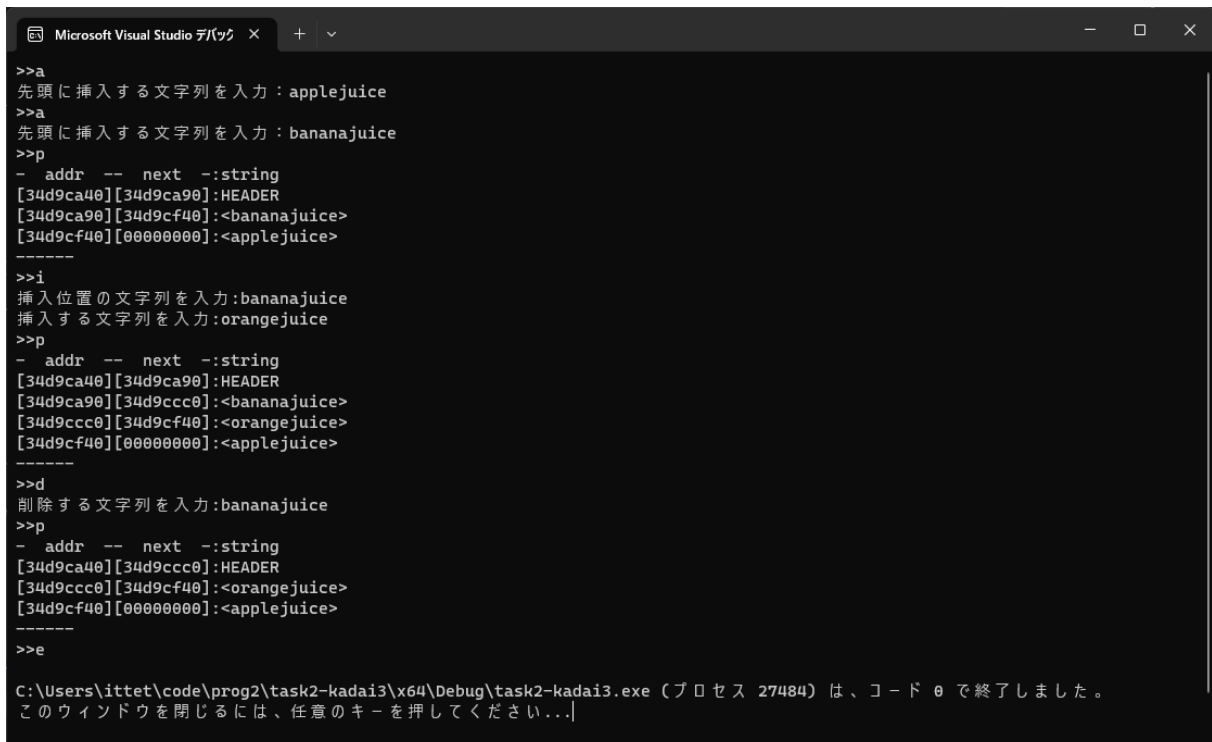
20:    {
21:        // ここを実装
22:        // pos の次のセルを削除する
23:        PCELL pDelCell;
24:        pDelCell = pos->next;
25:        pos->next = pos->next->next;
26:        free(pDelCell->string);
27:        free(pDelCell);
28:    }
29:
30: void ListDestroy(PCELL header)
31: {
32:     // ここを実装
33:     PCELL pDesCell;
34:     while (header)
35:     {
36:         pDesCell = header;
37:         header = header->next;
38:         free(pDesCell);
39:     }
40: }

```

### 3.5 実行結果

プログラムを実行し、今まで 7 文字までしか扱えなかったプログラムが可変長文字列を扱えるようになっているか確認したのが以下の図 6 である。

図 6 レポート課題 2 の実行結果



```

Microsoft Visual Studio デバッグ
>>a
先頭に挿入する文字列を入力:applejuice
>>a
先頭に挿入する文字列を入力:banana juice
>>p
- addr -- next -:string
[34d9ca40][34d9ca90]:HEADER
[34d9ca90][34d9cf40]:<banana juice>
[34d9cf40][00000000]:<applejuice>
-----
>>i
挿入位置の文字列を入力:banana juice
挿入する文字列を入力:orange juice
>>p
- addr -- next -:string
[34d9ca40][34d9ca90]:HEADER
[34d9ca90][34d9ccc0]:<banana juice>
[34d9ccc0][34d9cf40]:<orange juice>
[34d9cf40][00000000]:<applejuice>
-----
>>d
削除する文字列を入力:banana juice
>>p
- addr -- next -:string
[34d9ca40][34d9ccc0]:HEADER
[34d9ccc0][34d9cf40]:<orange juice>
[34d9cf40][00000000]:<applejuice>
-----
>>e

C:\Users\ittet\code\prog2\task2-kadai3\x64\Debug\task2-kadai3.exe (プロセス 27484) は、コード 0 で終了しました。
このウィンドウを閉じるには、任意のキーを押してください...

```

この実行結果より、可変長文字列が扱えており、挿入・削除ともに問題なく実行できていることがわかる。

このプログラムの終了直前に CrtDumpMemoryLeaks 関数を挿入し、デバッグした様子が以下の図 7・8 である。

図 7 レポート課題 2 デバッグした際の実行結果

```

Microsoft Visual Studio デバッグ
+
>>a
先頭に挿入する文字列を入力: bananajuice
>>a
先頭に挿入する文字列を入力: applejuice
>>p
- addr -- next -:string
[3dc3f010][3dc3f100]:HEADER
[3dc3f100][3dc3f330]:<applejuice>
[3dc3f330][00000000]:<bananajuice>
-----
>>i
挿入位置の文字列を入力:applejuice
挿入する文字列を入力:orangejuice
>>p
- addr -- next -:string
[3dc3f010][3dc3f100]:HEADER
[3dc3f100][3dc3f4c0]:<applejuice>
[3dc3f4c0][3dc3f330]:<orangejuice>
[3dc3f330][00000000]:<bananajuice>
-----
>>d
削除する文字列を入力:applejuice
>>p
- addr -- next -:string
[3dc3f010][3dc3f4c0]:HEADER
[3dc3f4c0][3dc3f330]:<orangejuice>
[3dc3f330][00000000]:<bananajuice>
-----
>>e

C:\Users\ittet\code\prog2\task2-kadai3\x64\Debug\task2-kadai3.exe (プロセス 23712) は、コード 0 で終了しました。
デバッグが停止したときに自動的にコンソールを閉じるには、【ツール】->【オプション】->【デバッグ】->【デバッグの停止時に自
動的にコンソールを閉じる】を有効にします。
このウィンドウを閉じるには、任意のキーを押してください...|

```

図 8 レポート課題 2 デバッグ結果

```
出力元(S): デバッグ
'task2-kada13.exe' (Win32): 'C:\Users\mittet\code\wprog2\task2-kada13\64\Debug\task2-kada13.exe' が読み込まれました。シンボルが読み込まれました。
'task2-kada13.exe' (Win32): 'C:\Windows\System32\ntdll.dll' が読み込まれました。
'task2-kada13.exe' (Win32): 'C:\Windows\System32\kernel32.dll' が読み込まれました。
'task2-kada13.exe' (Win32): 'C:\Windows\System32\kernelBase.dll' が読み込まれました。
'task2-kada13.exe' (Win32): 'C:\Windows\System32\vcrtime140d.dll' が読み込まれました。
'task2-kada13.exe' (Win32): 'C:\Windows\System32\ucrtbased.dll' が読み込まれました。
スレッド 0x3a9c13 はコード 0 (0x0) で終了しました。
'task2-kada13.exe' (Win32): 'C:\Windows\System32\kernel.appcore.dll' が読み込まれました。
'task2-kada13.exe' (Win32): 'C:\Windows\System32\msvcrt.dll' が読み込まれました。
スレッド 0x63a413 はコード 0 (0x0) で終了しました。
スレッド 0x66a013 はコード 0 (0x0) で終了しました。
プログラム '23712' task2-kada13.exe' はコード 0 (0x0) で終了しました。
|
```

図 6 より正しく実行できていることが、図 7 よりメモリリークが生じていないことがわかる。調べ方に関してはレポート課題 1 の時と同様であり、`CrtDumpMemoryLeaks` 関数を導入した後デバッグ結果に「`detected memory leaks!`」と表示されなければメモリリークが生じていないことがわかる。

### 3.6 考察

### 3.6.1. 構造体のサイズ

### 3.6.2. 考察課題

#### 4. 参考文献

なし