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 | 5 | 年 | 6 | 月 | 19 | 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】　------------------------------------------------------------------**

# 目的

　プログラム内容によって静的にメモリを確保する場合と動的にメモリを確保する場合が存在するため、メモリ管理が必要である。ソースコードを記述する際にメモリに割り当てる変数の大きさを決定できない場合に、プログラム実行時に動的にメモリ空間を確保することがある。これらを踏まえ、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。

# レポート課題１

## 概要

　動的メモリ管理を用いてキーボードからの入力を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。

## 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[1]。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。

## データ構造とアルゴリズム

### 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。これは最大の行数を宣言したMAXLINESという定数をサイズとした配列のポインタとなっており、その後のmain関数で利用され、free関数で解放される。

### 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  
A picture containing receipt, text, diagram

Description automatically generated

　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。

## 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。

リスト1　レポート課題1 Sort関数のソースリスト

|  |
| --- |
| 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。

## 実行結果

実行結果を以下の図2に示す。

A screenshot of a computer

Description automatically generated  
図2　レポート課題1の実行結果

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。

リスト2　レポート課題1 FreeLines関数のソースリスト

|  |
| --- |
| 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、メモリを解放している。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、メモリを解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。

A screenshot of a computer

Description automatically generated  
図3　レポート課題1 FreeLines導入後の実行結果

A screenshot of a computer

Description automatically generated  
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。

## 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。

# レポート課題２

## 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。

## 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。

可変長文字版リストはメモリの最大値まで自由な長さの文字列を入力でき、削除・解放できるため入力フォーム等に活用できると考えられる。

## データ構造とアルゴリズム

### データ構造

(1) 定数宣言  
なし

(2) 型宣言  
typedef struct cell{ char \*string, struct cell \*next; } CELL, \*PCELL;

：連結リストの1つのセルを宣言

(3) グローバル変数  
なし

　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。

### 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。

A picture containing text, diagram, receipt, pattern

Description automatically generated  
図5　レポート課題2のフローチャート

　ListInsert関数は指定された位置に新しいセルを挿入し、そのセルのアドレスを返す関数である。まず、新しいセルをメモリから確保し、確保が成功した場合新しいセルのstringメンバに文字列をコピーするために、適切なサイズでメモリを確保する。その後strcpy関数を使用して文字列を新しいセルのstringにコピーし、ポインタの付け替えを行う。最後に新しく確保したセルのアドレスを返す。

ListDelete関数は、posの次のセルを削除する関数である。一時変数pDelCellにposのnextを代入し、posの次の要素を次の次の要素に更新します。pDelCellの文字列変数のメモリを解放し、pDelCellのメモリを解放して終了する。

ListDestroy関数は、リスト全体を削除する関数である。whileループを使用してリストの先頭から順にセルを解放する。各セルの解放時には文字列変数のメモリも解放する。

## 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. if (pNewCell != NULL) 6. { 7. // 新しいセルへの文字列をコピーおよび 8. // ポインタの付け替えを行う 9. pNewCell->string = (char \*)calloc(strlen(string) + 1, sizeof(char)); 10. strcpy(pNewCell->string, string); 11. pNewCell->next = pos->next; 12. pos->next = pNewCell; 13. } 14. return pNewCell; 15. } 16. void ListDelete(PCELL pos) 17. { 18. // pos の次のセルを削除する 19. PCELL pDelCell; 20. pDelCell = pos->next; 21. pos->next = pos->next->next; 22. free(pDelCell->string); 23. free(pDelCell); 24. } 25. void ListDestroy(PCELL header) 26. { 27. PCELL pDesCell; 28. while (header) 29. { 30. pDesCell = header; 31. header = header->next; 32. free(pDesCell); 33. } 34. } |

3.3内でも示した通り、ListInsert関数は引数として受けたposの位置に新たな要素を挿入している。そして次の要素を指すポインタをそれぞれ書き換え、正しく表示されるようになっている。ListDelete関数は引数として受けたposの次の要素を削除・メモリの解放を行っている。ListDestory関数はリストの先頭から順に要素を削除・メモリの解放を行っている。

## 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。

A screenshot of a computer

Description automatically generated  
図6　レポート課題2の実行結果

　この実行結果より、これまで扱えていなかった可変長文字列が扱えており（「bananajuice」：11文字）、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。

A screenshot of a computer

Description automatically generated  
図7　レポート課題2 デバッグした際の実行結果

A screenshot of a computer

Description automatically generated  
図8　レポート課題2 デバッグ結果

　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。

## 考察

### 構造体のサイズ

### レポート課題2に関する考察

今回はListInsert関数・ListDelete関数・ListDestory関数の3つの関数を実装した。ListDelete・ListDestoryの2つについては確実にメモリの解放を行うことを忘れないように気をつけた。特に、文字型変数のメモリ解放を忘れないようにするのが重要であった。

# 参考文献

[1] C言語でmalloc関数の使い方についてわかりやすく詳しく解説,   
https://e-nine.com/programming-c-malloc/ , 2023/06/20閲覧