プログラミング　第3回レポート

202111609　仲村和士

2022年6月22日

1　はじめに

今回はポインタを中心とした問題群である。個人的には、C言語は、多少のミスがSegmentation faultにつながるのでデバッグが難しい言語であることが実感できる回であったと思う。それでは早速はじめて行こう。

2　設問(1)

2.1　問題概要と方針

標準入力でファイルを受け取り、cat -n風のプログラムを作成する問題である。方針としては、fgetsの返り値を利用して、ファイルの中身をすべて入力し、strcpy関数でコピーしてフォーマット通りに出力すればよい。

2.2　実装

以下に実装を提示する。4, 5行目でファイルの1行の文字列の長さ、および、ファイルの行数をマクロ定義してる。main関数では、まず変数定義をする。変数を簡単に説明すると、sはバッファ、fileはバッファの内容をコピーして使う本番ファイルでそれぞれcharの2重配列、iはループ変数、line_counterは行数のカウンタである。ファイルの標準入力を行い、line_counterという変数によりそこでわかったファイルの長さ分だけstrcpyでコピーとprintfによる出力のループをそれぞれ回している。

Listing 1: s2111609-1.c

```
1 #include<stdio.h>
2 #include<string.h>
3
4 #define BSIZE 100
```

```
5  #define LSIZE 10000
6
7  int main(void){
8      char s[LSIZE][BSIZE];
9      char file[LSIZE][BSIZE];
10     int i;
11     int line_counter = 0;
12
13     while(fgets(s[i],BSIZE,stdin) != NULL){
14         i++;
15         line_counter++;
16     }
17
18     for(i=0; i<line_counter; i++){
19         strcpy(file[i], s[i]);
20     }
21     for(i=0; i<line_counter; i++){
22         printf("%5d %s", i+1, file[i]);
23
24     }
25 }
```

2.3 確認

設問条件について確認しよう。

1. 標準入力で複数行の文字列を受け取る。
2. 受け取った文字列はバッファ配列に一時的に格納し、本配列にコピーする。
3. 配列サイズはマクロ定義する。
4. 行番号を右詰め 5 桁で付けて出力する。ただし、行番号と本文の間には半角空白を一つ入れる。

上のプログラムはこれらを満たすように作られている。以下に実行例を提示する。tester は test 用のファイルであり、リダイレクトを用いて標準入力として与えられる。また、行番号を 5 桁右詰めにする部分はレポート上だと、整形される影響でわかりにくいが、実際に実行すると設問条件の通りになることに留意されたい。

実行例

```
$ gcc -std=c89 s2111609-1.c
$ ./a.out < tester
  1 This is the example file for the test.
  2 Use this by redirecting to stdin.
  3 Keep 5 digits for line numbers.
  4 A line has 100 characters(max).
  5 A file has 10000 lines(max).
  6 Use copy function.
  7 Line 7
  8 hogehoge
  9 piyopiyo
 10 fugafuga
 11 kinoko
 12 Line 12
 13 Line 13
 14 Line 14
 15 Line 15
```

2.4 難しかった点など

入力回数がファイル依存の標準入力は競技プログラミングなどでもほとんどみられないため、少々慣れない実装であったと思う。ファイルが最終行に到達したときに fgets が NULL 値を返すことは理解していたのでそれで判定できることをテストプログラムで検証してから本実装をした。

3 設問 (2)

3.1 課題内容と方針

取得した文字列を動的に確保したメモリに格納するように改造する課題である。前設問で配列を 2 次元配列を 2 つ宣言しているため、この設問内容だけだと解釈の仕方がたくさんあると思われる。私は (設問趣旨をはかりかねたので) 大は小を兼ねるということで 4 箇所すべて動的に確保することにした。[1]

[1] しかし、実際のところ、ファイルを受け取る前にサイズを判断するのは難しいため、あえてバッファの配列を動的に確保したメモリに格納する意味はないと思われる。

方針としては、malloc を用いて必要なだけメモリを確保する。特に、本番ファイルの方ではサイズがわかっているので最小限のメモリのみを確保する。

3.2 実装

以下に実装を提示する。変更点を見ていこう。まず、char の 2 重配列だった部分は char のダブルポインタに変更した。これは動的なメモリ確保のためである。また、変数 s はわかりやすいように buff という名前に変更した。次に、14 行目からは buff に対して動的にメモリを確保している。malloc を用いる際には、メモリ確保に失敗したときに NULL を返すのでそれを確認して強制終了させることを忘れないように注意する。[2]

34 行目からのコピーを行う部分では、入力サイズに合わせた動的なメモリ確保を行っている。file で確保するべき大きさは、入力行数であり、file[i] で確保すべき大きさは strlen 関数で求めた長さにヌル文字分として 1 足したものになる。

53 行目からの部分では free 関数でメモリを解放し、メモリリーク対策をしている。[3]

Listing 2: s2111609-2.c

```
1  #include<stdio.h>
2  #include<string.h>
3  #include<stdlib.h>
4
5  #define BSIZE 100
6  #define LSIZE 10000
7
8  int main(void){
9      char **buff;
10     char **file;
11     int i;
12     int line_counter = 0;
13
14     /* Make a 2-dimension dinamic array */
15     buff= (char**) malloc(sizeof(char*)*LSIZE);
16     if(buff == NULL){
17         puts("memory allocation failure");
18         exit(EXIT_FAILURE);
19     }
20     for(i=0; i<LSIZE; i++){
21         buff[i]= (char*) malloc(BSIZE);
```

[2] ほかの言語に慣れていると、こういうときは例外を投げてほしいという気分になるのは私だけではないはず...

[3] プログラムが終了したら解放されるようにできていることも多いらしいが、組み込み系では深刻な問題になりうる。

```
22          if(buff[i] == NULL){
23              puts("memory allocation failure");
24              exit(EXIT_FAILURE);
25          }
26      }
27      /*Get lines*/
28      i=0;
29      while(fgets(buff[i],BSIZE,stdin) != NULL){
30          i++;
31          line_counter++;
32      }
33      /*Make a perfectly fited 2-dimension dinamic array */
34      file = (char**) malloc(sizeof(char*)*line_counter);
35      if(file == NULL){
36          puts("memory allocation failure");
37          exit(EXIT_FAILURE);
38      }
39
40      for(i=0; i<line_counter; i++){
41          file[i]= (char*) malloc(strlen(buff[i])+1 );
42          if(file[i] == NULL){
43              puts("memory allocation failure");
44              exit(EXIT_FAILURE);
45          }
46          strcpy(file[i], buff[i]);
47      }
48      /*output*/
49      for(i=0; i<line_counter; i++){
50          printf("%5d %s", i+1, file[i]);
51      }
52
53      /*free*/
54      for(i=0; i<LSIZE; i++){
55          free(buff[i]);
56      }
57      for(i=0; i<line_counter; i++){
58          free(file[i]);
59      }
60      free(buff);
61      free(file);
62  }
```

3.3 確認

この設問では動的なメモリ確保をすればよい。どの程度変更するべきかは解釈によるだろうが、すべて変更したので設問条件は確実に満たしている。注意事項

はおそらくメモリ確保失敗時の例外処理だろうと思われるが、それも対応している。以下は実行例である。内部的な実装を変更しただけなので出力は特に変わらない。

実行例

```
$ gcc -std=c89 s2111609-2.c
$ ./a.out < tester
  1 This is the example file for the test.
  2 Use this by redirecting to stdin.
  3 Keep 5 digits for line numbers.
  4 A line has 100 characters(max).
  5 A file has 10000 lines(max).
  6 Use copy function.
  7 Line 7
  8 hogehoge
  9 piyopiyo
 10 fugafuga
 11 kinoko
 12 Line 12
 13 Line 13
 14 Line 14
 15 Line 15
```

3.4 難しかった点など

単純なミスで Segmentation fault が発生したが、手がかりが何もないのでデバッグが大変であった。

4 設問 (3)

4.1 課題内容と方針

コマンドライン引数を取っていたら第 1 引数を用いて検索し、それを含む行をすべて表示、コマンドライン引数がなければ前問までと同様の出力をするという処理を関数に分けて作るという課題である。コマンドライン引数がある場合の行番号の振り方は解釈が複数あるが、grep -n 風の出力が得られるものと解釈して話を進める。

文字列中の検索には strstr 関数を用いれば良い。あとは、argc で適切に場合分けすればよい。

4.2 実装

実装を提示する。作成したのは output_plain 関数と output_grep 関数である。前者は引数を取ってはいるものの、中身は前問までの処理を仮引数に直してそのまま移植しただけなので特に説明することはない。

後者を説明する。strstr 関数は文字列中に与えられた文字列を含むかを調べる関数であり、含まれないときに NULL を返す。今回は NULL かどうかだけを判定すれば十分である。そのため、NULL でないときのみ出力するような if 文を組んでいる。

main 関数では、62 行目以降、argc の値から、コマンドライン引数の有無を確認してどちらの output 関数を走らせるか判定し、コマンドライン引数があるときには argv[1] を探索する文字列として渡すようにしている。

Listing 3: s2111609-3.c

```
1  #include<stdio.h>
2  #include<string.h>
3  #include<stdlib.h>
4
5  #define BSIZE 100
6  #define LSIZE 10000
7
8  void output_plain(char** file, int length){
9      int i;
10     for(i=0; i<length; i++){
11         printf("%5d %s", i+1, file[i]);
12     }
13 }
14 void output_grep(char* word, char** file, int length){
15     int i;
16     for(i=0; i<length; i++){
17         if(strstr(file[i], word) != NULL){
18             printf("%5d %s", i+1, file[i]);
19         }
20     }
21 }
22 int main(int argc, char** argv){
23     char **buff;
24     char **file;
25     int i;
26     int line_counter = 0;
27
28     /* Make a 2-dimension dinamic array */
29     buff= (char**) malloc(sizeof(char*)*LSIZE);
30     if(buff == NULL){
31         puts("memory allocation failure");
```

```
32          exit(EXIT_FAILURE);
33      }
34      for(i=0; i<LSIZE; i++){
35          buff[i]= (char*) malloc(BSIZE);
36          if(buff[i] == NULL){
37              puts("memory allocation failure");
38              exit(EXIT_FAILURE);
39          }
40      }
41      /*Get lines*/
42      i=0;
43      while(fgets(buff[i],BSIZE,stdin) != NULL){
44          i++;
45          line_counter++;
46      }
47      /*Make a perfectly fited 2-dimension dinamic array */
48      file = (char**) malloc(sizeof(char*)*line_counter);
49      if(file == NULL){
50          puts("memory allocation failure");
51          exit(EXIT_FAILURE);
52      }
53
54      for(i=0; i<line_counter; i++){
55          file[i]= (char*) malloc(strlen(buff[i])+1 );
56          if(file[i] == NULL){
57              puts("memory allocation failure");
58              exit(EXIT_FAILURE);
59          }
60          strcpy(file[i], buff[i]);
61      }
62      /*output*/
63      if(argc >= 2){
64          output_grep(argv[1], file, line_counter);
65      }else{
66          output_plain(file, line_counter);
67      }
68
69      /*free*/
70      for(i=0; i<LSIZE; i++){
71          free(buff[i]);
72      }
73      for(i=0; i<line_counter; i++){
74          free(file[i]);
75      }
76      free(buff);
77      free(file);
78  }
```

4.3 確認

設問条件を確認しよう。

コマンドライン変数がないときには、読み込んだ文字列を格納する配列、その要素数を引数にとり、前問までと同様の処理を行う関数を走らせる。

コマンドライン引数があるときには、それに加えて検索する文字列の計 3 引数をとり、grep 風の結果を出力する関数を走らせる。

これらをどちらも満たしていることは実行例から確認できる。

実行例

```
$ gcc -std=c89 s2111609-3.c
$ ./a.out < tester
  1 This is the example file for the test.
  2 Use this by redirecting to stdin.
  3 Keep 5 digits for line numbers.
  4 A line has 100 characters(max).
  5 A file has 10000 lines(max).
  6 Use copy function.
  7 Line 7
  8 hogehoge
  9 piyopiyo
 10 fugafuga
 11 kinoko
 12 Line 12
 13 Line 13
 14 Line 14
 15 Line 15
$ ./a.out < tester Line
  7 Line 7
 12 Line 12
 13 Line 13
 14 Line 14
 15 Line 15
$ ./a.out < tester kinoko
 11 kinoko
```

4.4　難しかった点など

今回も軽微なミスにより、Segmentation fault が出たが、変更範囲は少ないため、すぐ特定できた。デバッグしにくいので今後は要注意である。

5　感想

私が過去にポインタについて細かく学んだのは Python の特殊な仕様を理解しようとしていたときである。[4]当時は少し難しく感じられたが一度理解すれば混乱することはたまにあるにせよ、難しく感じることはほとんどなくなった。しかし、2年生になって新たに理解が深まった点もあり、それはアセンブリとの対比である。「コンピュータシステムと OS」の授業ではアセンブリとして CASL2 を学習したが、それとの対比は有益であった。たとえば、malloc は DS 命令とそっくりであったり、*(ptr+i) というのは指標レジスタの考え方をより便利にしたものである、などといったことが考えられるようになったのは理解を深めるのに有益であった。今回も特に新しく調べることはなかったので参考文献は設けていない。(どちらかというと、Segmentation fault に悩まされた。)

今回のレポートは内容よりも解釈が一意に定まらない点で割と悩んだ。内部の実装を変更する問題では、題意が一意に定まるように記述し、出力が変更される問題では競技プログラミングの問題のように出力例があれば解答する上でよりミスマッチが発生しなくなると思われる。

[4]Python は一見値渡しに見えるが、実はすべて参照渡しである。値渡しに見えることが多いのはイミュータブルオブジェクトの仕様がやや特殊なせいである。