

# プログラミング演習2

## 期末レポート

氏名: 今田将也 (IMADA, Masaya)  
学生番号: 09430509

出題日: 2019 年 06 月 19 日  
提出日: 2019 年 07 月 27 日  
締切日: 2019 年 07 月 30 日

### 1 概要

プログラミング演習2においては、プログラミング演習1でC言語の実践的なプログラミングの演習を行った際に不足していた機能を追加した。ファイルからのcsv形式のデータの読み込み、並びに書き出し。指定語句でデータを検索する機能と、メモリ中のデータを並び替える機能である。また、完成したプログラムの結果を通して、さらなる不足機能の考察、既存コマンドの改良案と実装方法についての考察を行った。

なお、与えられたプログラムの基本仕様と要件、および、本レポートにおける実装の概要を以下に述べる。プログラムの使用方法についても記載した。

#### 1. 仕様

- (a) 標準入力からID, 学校名, 設立日, 住所, 備考からなるコンマ区切り形式 (CSV 形式) の名簿データを受け付けて、それらをメモリ中に登録する機能を持つ。CSV 形式の例を以下に示す。

```
5100046,The Bridge,1845-11-2,14 Seafield Road Longman Inverness,SEN Unit 2.0 Open  
5100224,Canisbay Primary School,1928-7-5,Canisbay Wick,01955 611337 Primary 56 3.5 Open  
:
```

- (b) ただし%で始まるコマンドを受け付けて、登録してあるデータを表示したり整列したりするなどの機能を持つ。実装するコマンドを表1に示す。

#### 2. 要件

- (a) 名簿データは配列などを用いて少なくとも10000件のデータを登録できるようにする。今回のプログラムでは、構造体 `struct profile` の配列 `profile_data_store[10000]` を宣言して、10000件のデータを格納できるようにする。
- (b) 名簿データは構造体 `struct profile` および構造体 `struct date` を利用して、構造を持ったデータとしてプログラム中に定義して利用する。実装すべきデータ構造は表2である。表中の  $n$  bytes とは、 $n$  バイトの `char` 型配列を意味する。

また、本レポートでは以下の考察課題について考察をおこなった。

表 1: 実装するコマンド

コマンド	解説	パラメータ範囲
%C	メモリ中のデータ件数を表示する	パラメータなし
%P	メモリ中データを, $n$ に応じて表示させる	$n$ : -10000~10000 (0:全件表示 $n > 0$ :前から指定件数正順表示 $n < 0$ :後ろから指定件数正順表示)
%Q	システムを終了する	パラメータなし
%R filename	filename ファイルから csv データを読みこむ	filename
%W filename	メモリ中のデータを filename ファイルに書き出す	filename
%F word	システムを終了する	word
%S n	システムを終了する	$n$ : 1 から 5 までの正整数

表 2: 名簿データ

ID	学校名	設立日	住所	備考
32bit 整数	70 bytes	struct date	70bytes	任意長

1. 不足機能に関する考察
2. エラー処理に関する考察
3. 新規コマンドの実装
4. 既存コマンドの改良

また、発展的な考察として、以下の内容についても考察を行った。

1. 構造体のサイズ
2. 本課題の要件に対する考察
3. コマンドの拡張
4. テキスト形式とバイナリ形式

## 2 プログラムの作成方針

プログラムをおおよそ以下の部分から構成することにした。それぞれについて作成方針を立てる。

1. 必要なデータ構造の宣言部 (2.1 節)
2. 標準入力から得た CSV データの解析部 (2.2 節)
3. 構文解析したデータの内部形式への変換部 (2.3 節)
4. 各種コマンド実現部 (2.4 節)

### 2.1 宣言部

“宣言部”は必要な構造体を宣言する部分である。このレポートでは概要で示した表 2 に基づいて、以下のように宣言する。

```

struct date {
    int y;
    int m;
    int d;
};

struct profile {
    int id;
    char name[70];
    struct date found;
    char add[70];
    char *others;
};

struct profile profile_data_store[10000];

int profile_data_nitems = 0;

```

ここでは、名簿管理に必要なデータを定義している。struct date においては、設立日の設定に必要な変数 y, m, d を定義した。順に、設立年、設立月、設立日を表している。struct profile では、一つ当たりデータの構造を作るために利用している。int id は ID, char name は学校名, struct date found は設立日, char add は住所, char others は備考を設定している。これにより仕様に必要なデータを格納することが可能になっている。

## 2.2 解析部

“解析部”は入力された文字列を判別し処理をおこなう箇所である。しかし、このままでは、仕様を実現するための方法が曖昧であるうえフローチャートも複雑になる懸念があるうえ、今回の仕様の実現には手間が多くかかりそうである。そこで、段階的詳細化の考え方に基づいてさらなる詳細化をおこなって、プロトタイプを作りながらボトムアップによる実装をすることにした。まず、下記の (a) から (e) のように分割することにする。

- (a) 標準入力から読むべき行が残っている間、文字の配列 char line[] に 1 行分を読み込む。
- (b) line の 1 文字目が, ' % ' ならば, 2 文字目をコマンド名, 3 文字目以降をその引数として, 決定されたコマンドを実行する。
- (c) さもなくば line を新規データとみなし, , ' ' を区切りとして 5 つの文字列に分割する。
- (d) 分割してできた 5 つの文字列を変換部に渡し構造体に代入する。
- (e) 次の行を読み込む

コマンドを入力させるか、新規データを入力させるか選択したのちに、以上の処理をさせるように一段階詳細化させることも考慮したが、名簿管理プログラムということが自明であるため、プログラム起動時にコマンドかそうでないかを判別して処理させることで実装した。ここで扱う文字列は最大数が 1024 に限定されているため入力文字数に注意する必要がある。

## 2.3 変換部

“変換部”は分割された CSV データもしくは新規入力データを項目毎に型変換し、対応する構造体メンバに代入する部分である。メンバとして様々な型を用いているため、適切な代入の使い分けが必要となる。

文字列は関数 `strcpy` を用いて代入する。数値の場合、関数 `strtol` を用いて文字列を数値に変換してから代入する。構造体 `struct date` であるメンバ `y`, `m`, `d` については `split` 関数を実行し、文字列を分割してから代入数値としてする。

なお、構造体への代入については、`strcpy` 関数を用いることで容易に実装することができる。例えば、"2014-10-25"のような文字列を `split` 関数により分割し、`strcpy` 関数によって入力されたデータを `struct profile` 内の `struct date` に年と月と日を格納するという処理は、入力された文字列を','により分割する処理と同じ処理である。従って、区切り文字がCSVの','とは異なり、区切り文字が'-'になること以外は同様に記述できるはずである。

また、解析部から与えられた文字列はメモリ内に保持されているものではないデータであることにも注意する必要がある。つまり、変換部で文字列を処理する際には、入力された文字列に対して変換を行い、結果を表示をするだけではなく、関数 `new_profile` を使って受け取ったデータをメモリ内に保持しておく作業を行わなければならないことに気をつける必要がある。

## 2.4 各種コマンド実現部

“各種コマンド実現部”は、表1にある実装コマンドの、実際の処理をおこなう部分である。このレポートでは、具体的には、登録されているデータ件数を表示する機能と、指定形式でデータ内容を表示する機能、外部ファイルからデータを読み込む機能、外部ファイルに書き出す機能、メモリ中のデータを並び替える機能、データを検索する機能、また、システムを終了させるための機能の7つを実装している。

%Q はシステムを終了させる。

登録されているデータ件数を表示するためには(%C)、グローバル変数にて宣言している `profile_data_nitems` の値を表示すればよい。グローバル変数で宣言したのにも理由があり、`main` 関数内でこの変数を宣言してしまうと、別関数で利用する際に値の受け渡しが発生し、手間が増えるためグローバル変数として宣言した。

登録されているデータを表示するには(%P n)は `printf` 関数でメモリ内のデータを各項目毎に表示すればよい。ただし、与えられた引数が負の場合は、逆順ではなくデータを後ろから正順で表示するため、ポインタの位置に注意する必要がある。また、データ件数が0件の場合でも上記コマンドは実行されるが、データがないという表示を行わせている。

外部ファイルからのデータの入力(%R)はファイル構造体のポインタを作成し、`fopen()` 関数を用いた。データの書き出し(%W)も同様に実装したが、オープンモードが異なることに注意しなければならない。なお、書き出しの際は読み込んだファイルのCSVデータと同様の形式で書き出す仕様である。

データ検索(%F)は、引数として入力された語句がそれぞれのデータの要素に完全一致しているかどうかで実装した。設立日については、年月日を分割して保存したため再度文字列に変換してから文字列比較を行った。しかし後に部分一致による実装も行った。

データ整列(%S)は整列の条件式でデータの大小を比較してバブルソートにて実装していたがクイックソートへ変更した。変更した理由については後ほど述べる。比較の際、文字列データについては2つの文字列の大小を比較できる `strcmp()` 関数を用いた。設立日は年月日ごとに比較を行わせた。数値データは単純に減算し大小比較をしている。

### 3 プログラムおよびその説明

プログラムリストは9節に添付している。プログラムは全部で661行からなる。以下では、前節の作成方針における分類に基づいて、プログラムの主な構造について説明する。

#### 3.1 汎用的な関数の宣言（107行目から161行目）

まず、汎用的な文字列操作関数として、`subst()` 関数を107から117行目で宣言し、`split()` 関数を119から145行目で宣言、さらに `get_line()` に関係する関数を147から161行目で宣言している。

`subst` は、引数の `str` が指す文字列中の `c1` 文字を `c2` に置き換える。プログラム中では、入力文字列中の末尾に付く改行文字をヌル文字で置き換えるために使用している。

`split` は 引数の `str` が指す文字列を区切文字 `c` で分割し、分割した各々の文字列を指す複数のポインタからなる配列を返す関数である。プログラム中では、CSV を',,'で分割し、分割後の各文字列を返すのに使用されている。また、"2004-05-10" のような日付を表す文字列を '-' で分割して、`struct date` を生成する際にも使用している。

`get_line()` は、標準入力からの入力を受け付ける処理を当初実装していたが、ファイルポインタからの入力に対応ができていなかった。そのため、それに対応した `get_line_fp()` 関数をファサードした。 `get_line_fp()` 関数はファイルポインタからの読み込みを行う関数で、`get_line()` 関数ではその引数として `stdin` を渡して標準入力からも受け取れるようにした。

構造体のデータを一件出力するための関数として `printdata()` を、構造体を入れ替える関数として `swap_struct()` 関数を宣言した。

#### 3.2 変換部（32行目から44行目、611行目から647行目）

32から36行目は `struct date` 型の宣言部である。メンバについては、変数 `y` は設立年、変数 `m` は設立月、変数 `d` は設立日にそれぞれ対応させている。

38から44行目は `struct profile` 型の宣言部である。なお611~647行目はそれを扱う関数 `new_profile` である。メンバについては、設立日を入れ子構造にしている。こうすることで、要素を管理しやすくできる。なお、備考に対応する文字列 `*others` は任意長を許すようにしているため、`malloc` 関数と `strlen` 関数を用いて文字列を動的に格納できるようにした。文字列から各データ型への変換を担う関数は、`struct new\_profile` とすることで、変換部であることを明確にした。具体的な処理内容としては、受け取った文字列 `str` を分割し、分割した文字列を `ret1[]` に格納し、その後要素ごとに対応する構造体メンバにエラー検出のある `strncpy` 関数を用いて格納している。設立日については、`ret2[]` を用意し、各メンバに対応するよう格納させている。

#### 3.3 各種コマンド実現部（244行目から604行目）

244行目からの各種コマンド実現に必要な関数群は、`cmd_処理名` という名前に統一することで、関数であることを明確にした。コマンド `%P` は `cmd_print()`、コマンド `%C` は `cmd_check()`、コマンド `%Q` は `cmd_quit()`、コマンド `%R` は `cmd_read()`、コマンド `%W` は `cmd_write()`、コマンド `%F` は `cmd_find()`、コマンド `%S` は `cmd_sort()` にそれぞれ対応している。

192から242行目は、`%P,%C,%Q,%R,%W,%F,%S` のコマンドを解釈して適切な関数を呼び出す部分である。

%P は、279 から 321 行目に記載した。%C は、274 から 277 行目に記載した。%Q は、263 から 272 行目に記載した。%R は、353 から 366 行目に記載した。%W は、368 から 386 行目に記載した。%F は、411 から 436 行目に記載した。%S は、526 から 549 行目に記載した。

これらの関数の内容は、表 1 に記載した。

### 3.4 解析部

97 から 105 行目は `main()` 関数で、179 から 190 行目は、`parse_line()` 関数であり、作成方針で説明した解析部の動作におおよそ相当する。ただし (c) の 5 つの文字列に分割する部分は、解析部の `main()` 関数では実現せず、処理内容を明確にするために変換部である `new_profile()` 関数中で `split` を呼出し、各要素ごとに分割を行うことにしている。

## 4 プログラムの使用方法

本プログラムは名簿データを管理するためのプログラムである。CSV 形式のコンマ区切りのデータと % で始まるコマンドを標準入力から受け付け、処理結果を標準出力に出力する。入力形式の詳細については、第 1 節を参照されたい。

プログラムは、CentOS で動作を確認しているが、一般的な UNIX で動作することを意図している。gcc でコンパイルした後、標準入力から入力ファイルおよびデータを与える。

```
% gcc -Wall -o program1 program1.c
% ./program1 < test.txt
```

プログラムの出力結果としては CSV データの各項目を読みやすい形式で出力する。例えば、下記の `test.txt` に対して、

```
111,The Bridge,1845-11-2,Okayama,SEN Unit 2.0 Open
222,Bower School,1908-1-19,Kagawa,01955 641225 Primary 25 2.6 Open
333,Canisbay School,1928-7-5,Tokyo,01955 611337 Primary 56 3.5 Open
%C
%P 0
%Q
```

以下のような出力を得る。

```
param is 0.
*****print record data*****
data :      1 -----
Id    : 111
Name  : The Bridge
Birth : 1845-11-02
Addr  : Okayama
Com.  : SEN Unit 2.0 Open
-----
data :      2 -----
Id    : 222
Name  : Bower School
Birth : 1908-01-19
Addr  : Kagawa
Com.  : 01955 641225 Primary 25 2.6 Open
-----
```

```

data :      3 -----
Id   : 333
Name : Canisbay School
Birth : 1928-07-05
Addr  : Tokyo
Com.  : 01955 611337 Primary 56 3.5 Open
-----

```

入力中の%C はこれまでの入力データの件数を表示することを示し，%P 0 は入力したデータのうち，全件のデータを表示することを示している．なお，%Q はシステムを終了することを示す．

## 5 作成過程における考察

第2節で述べた実装方針に基づいて，第3節ではその実装をおこなった．しかし，実装にあたっては実装方針の再検討が必要になる場合があった．本節では，名簿管理プログラムの作成過程において検討した内容，および，考察した内容について述べる．

### 5.1 関数 split についての考察

関数 split については方針通りに実装することができたが，容易に実装することはできなかった．当初はコンマまでの文字列を別の配列に保存することを繰り返して実装しようとしていたが，これではコンマの数で文字列を判断することになるため失敗した．そこで文字列を破壊的に分割し別途規定数用意した文字配列にアドレスを格納することで実装できた．文字列を丸ごとコピーすることも考えられたが，その方法は，入力した倍のメモリ量が必要な上に使わなくなったメモリを開放する手間が増えるため用いなかった．

### 5.2 関数 get\_line についての考察

標準入力からの入力について当初は，main 関数の中で while 文繰り返し入力を行わせて，入力の度に入力内容が NULL でないか調べ関数 subst を適用する方法をとっていたが，while 文を脱する処理も記述しなければならないため手間が増えた．そこで，今回は入力内容に問題がなければ 1 を，あれば 0 を返す方針で実装を行った．これで，もし別の関数内で標準入力からの入力を行う際でも使いまわすことができ汎用性を持たせることができる．

### 5.3 関数 new\_profile についての考察

関数 new\_profile() の実装では，単に文字列を受け取り，その文字列を操作した後に，用意している配列 profile\_data\_store にコピーする方法も考えられたが，値を渡すことになり使用するメモリの量が増えると考えた．そのため，ポインタによるアドレス渡しによって実装を行った．また，配列を構造体配列として宣言しているので，ここでは構造体を返り値として設定した．そして，文字列を数値に変換する際にはエラー検出のある strtol 関数を用いた．

## 5.4 関数 `exec_command` についての考察

仕様を満たす実装はできた。標準入力からのデータは文字列であるため、各種コマンドへの引数を数値に変換する作業を行っている。この際、`atoi` 関数だとうまく変換されないことがあったため `strtol` 関数を用いている。また、定義されていないコマンドが入力された際は該当するコマンドがないという表示を出すようにした。

## 5.5 関数 `cmd_print` についての考察

まず、受け取った `param` の値が正か負か 0 を判断させなくてはならない。その後、正ならば指定件数分前から順に表示させ、0 ならば全件表示を行わせて、負ならば `param` の値を一度正に戻し指定件数分ポインタを移動させた後に正のとき同様に表示処理を行わせる手順で実装を行った。表示させる部分については、`find` 関数にて利用するため `printdata` 関数を別途作成し、当該関数内で表示させた。

## 5.6 関数 `cmd_check` についての考察

この関数の実装はあらかじめ、登録件数を保存するための変数をグローバル変数にて宣言することで容易に実装することができた。誤ったデータが入力された際も当初は件数が増えてしまう実装であったため、増やさないように改変を行った。

## 5.7 関数 `cmd_quit` についての考察

この関数の実装は、`stdio.h` にある `exit` 関数を利用することにより実装を行った。

## 5.8 関数 `cmd_read` についての考察

外部ファイルからのデータ入力は、`stdio.h` にある `fopen` 関数並びに `fclose` 関数を利用することで実装した。ファイルポインタの内部の仕組みを理解できてはいるが、作成中にファイル名が異なると開けずにシステムが強制終了するため回避する処理が必要だった。

## 5.9 関数 `cmd_write` についての考察

上記の `cmd_read` 同様に実装を行った。ただし、データを書き込むためオープンモードを書き込み状態にした。`fprintf` を用いてファイルポインタとして開いたファイルにコンマ区切りで書き込むため、データ形式には気を付けた。1つのデータの終わりに改行を置くことを当初忘れていたため望む実装ができず苦戦した。

## 5.10 関数 `cmd_find` についての考察

本関数は、各要素が `%F` の引数として与えられたデータを完全に一致しているかどうかで実装をしようとしたが、入力される引数が文字データであることを考慮しておらず苦悩した。構造体 `profile` の要素の `name`, `add`, `others` は文字であるから `strcmp` 関数にて比較可能で、`id` についても `strtol` 関数を用いた比較が可能である。しかし、`found` は年月日ごとに別の構造体に数値とし



てバラバラにあるため、一度 csv データと同じ‘-’で繋がった形式に別途関数を介し変換させた後に別のデータ同様に文字列として比較を行った。

### 5.11 関数 `cmd_sort` についての考察

ソートについては苦労が大きく実装に手間取った。文字列の比較がわからなかったが `strcmp` 関数が数値を返すと知り、それを用いることで比較を行い要素の大小比較の実装ができた。要素の大小の結果をソートの比較条件に利用した。データ数が多くなるとソート完了までに時間がかかっていたため、交換の回数をカウントしバブルソートとクイックソートの比較を行った（第 6.4 節）結果、クイックソートも実装した。

## 6 結果に関する考察

演習課題のプログラムについて仕様と要件をいずれも満たしていることをプログラムの説明および使用法における実行結果例によって示した。ここでは、概要で挙げた以下の項目について考察を述べる。

1. 不足機能についての考察
2. エラー処理についての考察
3. 新規コマンドの実装
4. 既存コマンドの改良

### 6.1 不足機能についての考察

不足機能については、以下の内容が考えられる。

1. 入力後のデータ修正機能
2. 指定要素のみ表示させる機能

#### 6.1.1 入力後のデータ修正機能

現在の機能では、データの形式さえあっていれば名簿データとして追加されるため、データの内容を間違えて入力しても追加される。これを回避するために、6.3.1 節にて指定データを削除する機能を実装している。しかし、削除に再度長いデータを入力する必要があり不便である。

名前を間違えたら名前のみを、住所を再度編集したい場合は住所のみを書き換えるという機能があればより現実的に利用ができる名簿管理プログラムになるのではないだろうか。

#### 6.1.2 指定要素のみ表示させる機能

これは `cmd_print` を改良すれば実装ができそうである。`cmd_print` はすべての要素を表示させている。そこに一つ引数を増やすかもしくは別のコマンドを作成し、表示させたい要素と数値を紐付けする。例えば 1 ならば ID のみを %P のように表示させる実装が可能そうだが、%P の引数が表示件数と要素の 2 つが必要になりそうであるから実装は困難になるかもしれない。

## 6.2 エラー処理についての考察

1. CSV データ処理中のエラー処理
2. 登録件数超過おけるエラー処理
3. split 関数のエラー処理
4. get\_line 関数のエラー処理
5. データ表示時のエラー処理
6. ファイル入出力のエラー処理
7. find 関数のエラー処理

### 6.2.1 CSV データ処理中のエラー処理

CSV データ中に、不正なデータが含まれていた場合の処理について考察する。

エラーのあった行を指摘せず、終了または無視するという方法も考えられるが、正常終了との区別が付かない上に、どの状態でエラーが発生しているのか確認をとることができないため実用的でないと考えた。今回は、エラーのあった行を指摘して、無視する方法で実装を行っている箇所が多々ある。

プログラム中に `ERROR:` で始まる表示を書いてエラー目印としている。また、エラーのあった内容を指摘するためには、`enum` という機能を利用してどのエラーなのかユーザが一度見て理解できるように、標準エラー出力を利用してエラーの内容を表示させている。

しかし、これはデータの型や仕様に指定されたデータ件数時にしかエラー処理を行っていないため他にも実装の余地はあると考える。例えば設立年月日について見ると、設立年はマイナス値、一桁や二桁は不自然だろうし設立月についても 12 より大きい数字は現在の暦では利用されていないはずである。利用する上でのエラーというものを考慮する必要がある。

### 6.2.2 登録件数超過におけるエラー処理

本プログラムは、データがすでに 10000 件ある状態で新規入力が行われた場合でも一度 `new_profile` 関数を実行する。といっても、関数のはじめにデータ件数を確認しており 10000 件を超えるデータは保存されないようになっている。

その状態だと条件を調べるために `new_profile` 関数を介しメモリを確保するなど無駄な作業が発生しているため、内部的な処理目的のためのエラー処理を導入してもよいのではと考えた。

実装案としては、`parse_line` を実行時に条件分岐を行う際に `profile_data_nitems` の中身を確認し、10000 件を超えるようであれば処理を行わなくすれば良い。以下実装案のプログラムである。

```
void parse_line(char *line){
    if(line[0]=='%'){
        exec_command(line[1], &line[3]);
    }
    else if(profile_data_nitems==10000){
        //件数エラーと表示するもしくは、本関数から抜ける。
    }
    else{
        new_profile(&profile_data_store[profile_data_nitems],line);
    }
}
```

```
}  
}
```

### 6.2.3 split 関数のエラー処理

split 関数は、あらかじめ分割数が定められるように実装されている。そのため、その分割数を満たしていないもしくは、分割数を超過している場合は意図したデータが入力されていないということを判断できる材料になる。

だから、それを利用し分割数を満足していない場合と超過している場合はエラー処理を行わせている。そして、本プログラムでは、split 関数は最後まで処理を行わせている。その後の処理を継続するかどうかについては利用先の関数内で判断させて実装をした。

### 6.2.4 get\_line 関数のエラー処理

標準入力からのデータの受付を行う際は、NULL というデータを受け取ることはなかった。しかし、csv データを読み込むときに本関数を利用したところ、EOP に反応しシステムが異常終了した。そこで、NULL ならエラーとし、return させるように実装した。

### 6.2.5 データ表示時のエラー処理

このエラー処理は%pに関連する、メモリ内に保存されているデータを表示する時のエラー処理である。以下の状態の時を考慮した。

1. データ件数が0件のとき
2. 引数が登録件数より多いとき

他にも、パラメータが負のとき、引数が0のときなどを考慮しようとしたが仕様上、役割を持っているため考慮していない。仮に、何も役割がないとするならば、負のときは正に、引数が0は不正な値として処理させるだろう。

### 6.2.6 ファイル入出力のエラー処理

%R,%Wともに、ファイル名が入力されない場合、もしくは同名のファイルが既に存在してなおかつアクセス権限が与えられていない場合、システムが落ちる。これを回避するためにファイルポインタがNULLの場合つまりファイルを開けないときにエラー処理を行わせた。

### 6.2.7 find 関数のエラー処理

このエラー処理は作成したプログラムに実装しているものだが、データが見つからなかった場合の表示を行っている。引数のkeywordに一致したデータが見つかった場合1を、見つからなかった場合を0として予め判定用の変数を用意して処理を行わす。

具体的には、checkという変数を用意し、検索処理が完了しデータを画面上に表示したあとに変数に1を代入し見つかったということにしている。見つからなかった場合はエラー出力に見つからなかった旨を表示している。

## 6.3 新規コマンドの実装

新規コマンドについては第 6.1 節を基に以下の実装を行った。

1. 指定データの削除コマンド
2. どのようなコマンドあるか表示するコマンド
3. 指定番号のデータのみ表示させるコマンド

### 6.3.1 指定データの削除コマンド

本関数は、`exec_command` に `%D` として新たに定義して利用できるようにした。処理内容としては簡単なものであり、削除したいデータをその一つあとのデータですべてのデータを上書きするものである。しかし、データ件数の値である `profiel_data_nitems` の値は減らないため、その値を一つ減らすことで対応している。

プログラムは第 9 節の 597 行目から 609 行目にある関数 `cmd_delete` である。

### 6.3.2 どのようなコマンドあるか表示するコマンド

本関数は、増えてきた本プログラムの機能をわかりやすくユーザに伝えるために作成した。どのような機能があるかを設定し `fprintf` 関数を用いて標準エラー出力に出力させている。出力データを記載した外部データを用意することも考えたが読み込み分手間だからそれは用いなかった。

実装方法は、28 行目から 30 行目の `enum` と `char` 型二次元配列を用いて、列挙させた。これを利用した理由としては今後実装するコマンドが増えたとしても変更が容易だからである。

プログラムは第 9 節の 244 行目から 261 行目にある関数 `cmd_help` である。`%H` コマンドにて確認できる。

### 6.3.3 指定番号のデータのみ表示させるコマンド

本関数はソートを行う際のデータの整合性を確認しているときに実装した関数である。例えば 3000 件あるデータの中で、2900 件目を見たいとき、今の段階では少なくとも 100 件分のデータも一緒に表示されてしまうため処理的にも確認作業を行う上でも無駄が多くなってしまう。

実装としては `%P` の機能を用いず新たに関数を作り、引数が 0 のときまたは登録件数を超えたときはエラー処理を行い、引数が負ならば正に変えているがこれは、データ番号に負はないからである。

プログラムは第 9 節の 333 行目から 351 行目にある関数 `cmd_pex` である。`%E` コマンドにて確認できる。

## 6.4 既存コマンドの改良

既存コマンドの改良は以下の内容で行った。

1. 語句の検索の部分一致
2. `%S` コマンドの高速化
3. システム終了時の終了確認

#### 6.4.1 語句の検索の部分一致

現在の `cmd_find` は文字列の完全一致による実装であるため、システムとして利用するには不便である。完全一致ではなく部分一致を行うことができれば検索機能が柔軟になると考えた。

実装方法としては、検索される文字列と探したい文字列を比較を行う。検索される文字列の中に探したい文字列の先頭文字があるかどうか探す。そこから各文字列の文字を一つずつ見ていき、探したい文字列がすべて見終わったら部分文字列が一致していることになるため検索が可能になると考える。以下考察したプログラムである。部分文字列が一致すると 0 を返し、それ以外は 1 を返す。

```
int find_kai(char *s, char * cp)
{
    char *s1, *s2;
    if( *cp == '\0') return s; /*cp の文字列長が 0 なら s を返す*/

    while( *s != '\0'){
        while(*s != '\0' && *s != *cp) { /*先頭文字が合うまで探す*/
            s++;
        }
        if(*s == '\0') return 1; /*見つからない*/
        s1 = s;
        s2 = cp;
        while ( *s1 == *s2 && *s1 != '\0'){ /*cp の先頭以降の文字列が一致するか*/
            s1++;
            s2++;
        }
        if( *s2 == '\0'){ /* cp の文字列は、全て一致した*/
            return 0;
        }
        s++; /*次の位置から、調べ直す*/
    }
    return 1; /*見つからない*/
}
```

利用する際は `%FB` にて利用する。引数は `%F` と同様である。

#### 6.4.2 %S コマンドの高速化

当初の `%S` コマンドでのソートはバブルソートを用いて行っていた。sample.csv データの 2886 件程度のデータのソートならば時間は気にならなかったが、10000 件でソートを行うと待たされる時間があり、もっと早くできないかと思った。

そこで、クイックソートによるソートの実装を行った。 `%QS` というコマンドを利用することでクイックソートによるソートを実行する。 `%S` はバブルソートである。

バブルソートとクイックソートによる交換回数の比較を行った。すべて、ID によるソート後の Name のソート回数である。

表 3: 比較回数

	データ件数	交換回数
bubble sort	2886	193900
quick sort	2886	17715
bubble sort	10000	23290889
quick sort	10000	76963

データ件数が 2886 件のときでもクイックソートのほうが約 10 倍速いことがわかる。10000 件になるとさらに明確でになり，クイックソートの方が約 300 倍速い。そこで，ソートはクイックソートを実装した。なお，バブルソートでもソートできるよう両方の機能を利用できるようにしている。

### 6.4.3 システム終了時の終了確認

作成中に誤って %q コマンドを実行してしまいデータを保存せずに終了する事故が発生した。それを回避するために，コマンド実行後さらに終了してよいかの確認をユーザから受け付けるようにできればよいと考える。

実装はシンプルで `getc` 関数を用いて，入力された文字が `y` ならば終了させている。それ以外の文字や文字列ならば `return` させて `main` 関数に再度戻る。ただし，文字列の場合も先頭の一文字のみで判断するため `yyyyyy` でも反応する。

## 7 発展課題

### 7.1 構造体のサイズ

作った `profile` 構造体のサイズを調べるために新たにコマンドを作り確認した。第 9 節の 649 行目から 661 行目である。以下その結果を示す。

```
>>>>>%SIZE
struct profile = 168
id = 4
name = 70
add = 70
found = 12
found.y = 4
found.m = 4
found.d = 4
Com. = 8
```

各メンバごとのサイズを合計すると，164 であるが，`profile` 構造体のサイズは 168 と差がある。つまり，4 バイト分だけ余分にメモリを確保していることになる。このことについて調べてみると，多くの CPU がメモリにアクセスする際，4 バイトごとにアクセスすることが分かった。このため，4 バイト境界をまたぐデータを扱うと，メモリにアクセスする回数が増える。だから，メンバの `name` と `add` が 4 の倍数でないため，それを次の 4 の倍数である 72 に揃えるために，それぞれ 2 バイトずつの計 4 バイトが詰められていると推測できた。

## 7.2 本課題の要件に対する考察

本課題の要件に対する考察として以下のことが考えられる。

1. 入力データの厳密性
2. コマンド引数の区切り文字
3. ソート条件の引数

### 7.2.1 入力データの機密性

本課題の要件には、住所と備考の文字数についての制限が記載されているがその他のデータの制限はとくに決まっていない。そのため、設立年月日の各メンバに例えば、月に 8 桁の数字などの本来存在しない値が保存されてしまう。

また、文字をデータとして与える際に英語の半角文字限定での制限なのか日本語も含めた制限なのかについての記載もない。日本語だとエンコード設定によっては文字化けする可能性もあるため入力データについては厳密性が必要になると考える。

### 7.2.2 コマンド引数の区切り文字

本課題の要件にある各コマンドの区切り文字は空白であるが、これは%F で文字列を検索する際に不便である。登録されるデータの住所や備考には空欄が入っておりその空欄との判別をする手間がかかりシステムが複雑になることが懸念されるからである。例えば、次のように入力したとする。

```
%F The Bridge
```

このようにすると空白が 2 つ含まれた文字列になり、文字列を空白で `split` 関数を用いて分割していると、%F と The と Bridge と 3 つにわけることができ、本来ないはずの 3 つ目の引数として解釈される可能性もある。

これを避けるためにも、入力データ内で利用されていない区切り文字や csv データの区切り文字であるコンマなどは避けるべきであると考える。

### 7.2.3 ソート条件の引数

本課題におけるソート関数である%S の仕様として、引数に数字でソートする要素が指定されているが、ユーザがどの数字がどのデータに対応しているか覚えている必要があるためユーザ体験としては悪いものであると感じた。

そのため、引数を数字ではなく文字で指定するべきであると考えた。具体的には、Name や Add などのように%P で表示した際の要素の名前で入力させる。これもユーザがメンバの名前を覚える必要があると思うが、数字がどの要素に対応しているかよりも感覚的にわかりやすく、要素の名前も%P で確認できるため忘れても問題ないと考える。

## 7.3 コマンドの拡張

1 文字入力のコマンドを 2 文字以上の入力に対応させるため、`exec_command` 関数で当初利用していた `switch` 文は利用をやめた。理由として `switch` 文は文字列の分岐に対応していないからである。そこで、`parse_line` で % の次の文字を `exec_command` 関数に送っていた実装も、`split` 関数を介して文字列を `exec_command` 関数に渡すように変更した。空白以降のコマンドの引数も問題なく送れる。

`exec_command` 関数での判定は、`strcmp` 関数を用いて受け取った文字列を判定している。これにより、2 文字よりも多いコマンドでも受け付けることができるようになる。例として、構造体などのサイズを調べるコマンドとして `%SIZE` が利用できる。

なお、拡張後の関数が 9 節の 192 行目から 242 行目と 179 行目から 190 行目に記載してある。

## 7.4 テキスト形式とバイナリ形式

最後の発展課題としてバイナリ形式での書き出しの実装を行おうとした。まず、書き出しを実装しようとファイルモードをバイナリに対応した `"wb"` モードにし、1 個のデータを書き出そうとしたが 392 行目から 404 行目の実装では実現できなかったため断念した。バイナリ形式のデータが得られなかったため読み込みの実装もできなかった。

## 8 感想

1 学期の内容よりも遥かに高度で、ポインタやデータ構造における正しい理解が求められる内容に感じた。特に文字列を扱う関数の実装は難しく大変だった。数値の比較と文字列の比較は完全に別物と考える必要があったし、もともと数値であるものを文字列に戻し比較を行う処理などもあり苦労したが、その分、実装時に利用した関数の特徴や癖なども理解できたため大きな経験になった。

そして、自分が欲しいと思った機能（例えば、部分一致やクイックソート）はすでにヘッダファイル内で準備されており、自作をしなくとも使えることを知ったが、ブラックボックスの状態を利用するよりも自分で内部構造を理解した上で使えることが重要だと感じたため、容易には用いずに、理解した上で利用したい。

## 9 作成したプログラム

作成したプログラムを以下に添付する。

---

```
1  /*
2   * File: meibo.c
3   * Author: 09430509
4   *
5   * Created on 2019/04/10
6   * update on 2019/07/26
7   */
8
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include <string.h>
12
13 #define LIMIT 1024
14 #define maxsplit 5//最大分割数
15 #define luck -1
16 #define over -2
17 #define endp NULL//strtol 用ポインタ
```



```

18  #define base1 10//10進数
19
20  typedef enum{
21      null,LUCK,OVER,NOTDEFINED,
22      NORECORD,OVERNUMBERRECORD,
23      FORMATINPUT,FORMATID,FORMATDATE,
24      NUMITEM,ERRORNUM,NOFILEOPEN,
25      OVERNITEMS,PARAMERROR,
26  } ERROR;
27
28  typedef enum{
29      Q,C,P,E,R,W,BR,BW,F,FB,S,QS,D,SIZE,LIST
30  }HELP;
31
32  struct date {
33      int y;//year
34      int m;//month
35      int d;//day
36  };
37
38  struct profile{
39      int id;//id
40      char name[70];//schoolname
41      struct date found;
42      char add[70];//address
43      char *others;//備考
44  };
45
46  /*subst*/
47  int subst(char *str,char c1,char c2);
48
49  /*split*/
50  int split(char *str,char *ret[],char sep,int max);
51  void error_split(int check);
52
53  /*get_line*/
54  int get_line(char *input);
55  int get_line_fp(FILE *fp,char *input);
56
57  /*parse_line*/
58  void parse_line(char *line);
59
60  void printdata(struct profile *pro, int i);
61  /*cmd*/
62  void exec_command(char *cmd, char *param);
63  void cmd_quit();
64  void cmd_check();
65  void cmd_print(struct profile *pro,int param);
66  void cmd_pex(int param);
67  void cmd_read(char *filename);
68  void cmd_write(char *filename);
69  void cmd_binread(char *filename);
70  void cmd_binwrite(char *filename);
71  void cmd_find(char *keyword);
72  void cmd_findb(char *keyword);
73  void swap_struct(struct profile *i, struct profile *j);
74  int compare_profile(struct profile *p1, struct profile *p2, int column);
75  int compare_date(struct date *d1, struct date *d2);
76  void cmd_sort(int youso);
77  void cmd_qsort(int youso);
78  int partition (int left, int right,int youso);
79  void quick_sort(int left, int right,int youso);
80  void cmd_delete(int param);
81  void cmd_help();
82  void cmd_size();
83  int find_kai(char *s, char * cp);
84
85
86
87  /*profile*/
88  struct profile *new_profile(struct profile *pro,char *str);
89  char *date_to_string(char buf[],struct date *date);
90
91  /*GLOBAL*/
92  struct profile profile_data_store[10000];
93  int profile_data_nitems = 0;

```

```

94     int quick_count = 0;
95
96     /*MAIN*/
97     int main(void){
98
99         char line[LIMIT + 1];
100         while (get_line(line)) {
101             parse_line(line);
102         }
103         return 0;
104     }
105
106     int subst(char *str,char c1,char c2){
107         int count = 0;
108         while(*str != '\0'){
109             if(*str == c1){
110                 *str = c2;
111                 count++;
112             }
113             str++;
114         }
115         return count;
116     }
117 }
118
119 int split (char *str,char *ret[],char sep,int max){
120     int count = 0;//分割数
121
122     while (1) {
123         if(*str == '\0') {
124             break;//からもじなら抜ける
125         }
126
127         if(count>max)break;
128         ret[count++] = str;//
129         //strをいじればretも変わるように分割後の文字列にはポインタを入れる
130         while( (*str != '\0') && (*str != sep) ){//区切り文字が見つかるまでポインタすすめる
131             str++;
132         }
133
134         if(*str == '\0') {
135             break;//区切り文字がなかったら抜ける=文字列はそのまま
136         }
137
138         *str = '\0';//必ず区切り文字のはずだからくぎる
139         str++;//インクリメントさせる
140     }
141
142     if(count<max)count = luck;
143     else if(count>max)count = over;
144     return count;
145 }
146
147 int get_line(char *input){
148     return get_line_fp(stdin,input);
149 }
150
151 int get_line_fp(FILE *fp,char*input){
152     fprintf(stderr,"\n>>>>");
153
154     if (fgets(input, LIMIT + 1, fp) == NULL){
155         fprintf(stderr,"ERROR %d:NULL--getline()\n",null);
156         return 0; /* 失敗EOF */
157     }
158     subst(input, '\n', '\0');
159
160     return 1; /*成功*/
161 }
162
163 void error_split(int check){
164     switch(check){
165         case luck:
166             fprintf(stderr,"ERROR %d:luck--split()\n",LUCK);
167             break;

```

```

168
169         case over:
170             fprintf(stderr, "ERROR %d: over--split()\n", OVER);
171             break;
172
173         default:
174             break;
175     }
176     return;
177 }
178
179 void parse_line(char *line){
180     char *ret[2];
181     int com=0;
182
183     if(line[0]=='%'){
184         com=split(line,ret,' ',2);
185         exec_command(ret[0], ret[1]);
186     }
187     else{
188         new_profile(&profile_data_store[profile_data_nitems],line);
189     }
190 }
191
192 void exec_command(char *cmd, char *param){
193     if(strcmp(cmd,"%Q")==0||strcmp(cmd,"%q")==0){
194         cmd_quit();
195     }
196     else if(strcmp(cmd,"%C")==0||strcmp(cmd,"%c")==0){
197         cmd_check();
198     }
199     else if(strcmp(cmd,"%E")==0||strcmp(cmd,"%e")==0){
200         cmd_pex(strtol(param,endl,base1));
201     }
202     else if(strcmp(cmd,"%P")==0||strcmp(cmd,"%p")==0){
203         cmd_print(&profile_data_store[0],strtol(param,endl,base1));
204     }
205     else if(strcmp(cmd,"%R")==0||strcmp(cmd,"%r")==0){
206         cmd_read(param);
207     }
208     else if(strcmp(cmd,"%W")==0||strcmp(cmd,"%w")==0){
209         cmd_write(param);
210     }
211     else if(strcmp(cmd,"%F")==0||strcmp(cmd,"%f")==0){
212         cmd_find(param);
213     }
214     else if(strcmp(cmd,"%FB")==0||strcmp(cmd,"%fb")==0){
215         cmd_findb(param);
216     }
217     else if(strcmp(cmd,"%D")==0||strcmp(cmd,"%d")==0){
218         cmd_delete(strtol(param,endl,base1));
219     }
220     else if(strcmp(cmd,"%S")==0||strcmp(cmd,"%s")==0){
221         cmd_sort(strtol(param,endl,base1));
222     }
223     else if(strcmp(cmd,"%QS")==0||strcmp(cmd,"%qs")==0){
224         cmd_qsort(strtol(param,endl,base1));
225     }
226     else if(strcmp(cmd,"%H")==0||strcmp(cmd,"%h")==0){
227         cmd_help();
228     }
229     else if(strcmp(cmd,"%BW")==0||strcmp(cmd,"%bw")==0){
230         cmd_binwrite(param);
231     }
232     else if(strcmp(cmd,"%BR")==0||strcmp(cmd,"%br")==0){
233         cmd_binread(param);
234     }
235     else if(strcmp(cmd,"%SIZE")==0||strcmp(cmd,"%size")==0){
236         cmd_size();
237     }
238     else {
239         fprintf(stderr, "ERROR %d:%s command is not defined.--exec_command()\n",
240             NOTDEFINED,cmd);
241         fprintf(stderr,"command list : %%H\n");
242     }
243 }

```

```

243
244 void cmd_help(){
245     int i;
246     char help_list[LIST][40]=
247     {
248         "Q : quit system","C : check data num","P [value] : print data",
249         "E : print specified data","R [filename] : read csv data","W [filename] : write
            csv data",
250         "BR : read binary data","BW : write binary data",
251         "F [word] : Exact match search","FB [word] : Partial match search",
252         "S [value] : sort (bubble)","QS [value] : quick sort",
253         "D [value] : delete data","SIZE : size check",
254     };
255
256     for(i=0;i<LIST;i++){
257         fprintf(stderr,"%s\n",help_list[i]);
258     }
259
260     return ;
261 }
262
263 void cmd_quit(){
264     fprintf(stderr,"Are you sure you want to quit?(y or n)>>>");
265     if(fgetc(stdin)=='y'){
266         fprintf(stderr, "END SYSTEM.\n");
267         exit(0);
268     }
269     else{
270         return;
271     }
272 }
273
274 void cmd_check(){
275     fprintf(stdout,"%d profile(s)\n",profile_data_nitems);
276     return;
277 }
278
279 void cmd_print(struct profile *pro,int param){
280     if(profile_data_nitems == 0){
281         fprintf(stderr,"ERROR %d:No record. No print.--cmd_print()\n",NORECORD);
282         return ;
283     }
284     int i;
285
286     if(param == 0){//0 のとき
287         fprintf(stderr, "*****print record data*****\n");
288         for(i=0;i<profile_data_nitems;i++){
289             printdata(pro+i,i);
290         }
291     }
292
293     else if(param > 0){//正のとき
294
295         if( param > profile_data_nitems ){
296             fprintf(stderr,"ERROR %d:over number of record.--cmd_print()\n",
                OVERNUMBERRECORD);
297             fprintf(stderr,"ERROR %d:number of item is %d\n",NUMITEM,profile_data_nitems
                );
298             return;
299         }
300         fprintf(stderr, "*****print record data*****\n");
301         for(i = 0;i<param;i++){
302             printdata(pro+i,i);
303         }
304     }
305
306     else if(param < 0){//負の時
307
308         param *= -1;
309         if( param > profile_data_nitems ){
310             fprintf(stderr,"ERROR %d:over number of record.--cmd_print()\n",
                OVERNUMBERRECORD);
311             fprintf(stderr,"ERROR %d:number of item is %d\n",NUMITEM,profile_data_nitems
                );
312             return;
313         }

```

```

314         pro += profile_data_nitems-param;
315         fprintf(stderr, "*****print record data*****\n");
316         for(i=0 ;i<param;i++){
317             printdata(pro+i,profile_data_nitems-param+i);
318         }
319     }
320     return;
321 }
322
323 void printdata(struct profile *pro, int i){
324     fprintf(stderr,"data : %5d -----\\n",i+1);
325     fprintf(stdout,"Id : %d\\n",pro->id);
326     fprintf(stdout,"Name : %s\\n",pro->name);
327     fprintf(stdout,"Birth : %04d-%02d-%02d\\n",pro->found.y,pro->found.m,pro->found.d);
328     fprintf(stdout,"Addr : %s\\n",pro->add);
329     fprintf(stdout,"Com. : %s\\n\\n",pro->others);
330     fprintf(stderr,"-----\\n");
331 }
332
333 void cmd_pex(int param){
334     if(profile_data_nitems == 0 || param == 0){
335         fprintf(stderr,"ERROR %d:No record. No print.--cmd_print()\\n",NORECORD);
336         return ;
337     }
338
339     if(param<0){
340         param*=-1;
341     }
342
343     if( param > profile_data_nitems){
344         fprintf(stderr,"ERROR %d:over number of record.--cmd_print()\\n",
345             OVERNUMBERRECORD);
346         fprintf(stderr,"ERROR %d:number of item is %d\\n",NUMITEM,profile_data_nitems);
347         return;
348     }
349     param-=1;
350     printdata(&profile_data_store[param],param);
351     return;
352 }
353
354 void cmd_read(char *filename){
355     char line[LIMIT+1];
356     FILE *fp;
357
358     if((fp = fopen(filename,"r"))==NULL){
359         fprintf(stderr,"ERROR %d:openfile error!!!---cmd_read()\\n",NOFILEOPEN);
360         return;
361     }
362     while(get_line_fp(fp,line)){
363         parse_line(line);
364     }
365     fclose(fp);
366     return;
367 }
368
369 void cmd_write(char *filename){
370     FILE *fp;
371     int i;
372     if((fp = fopen(filename,"w"))==NULL){
373         fprintf(stderr,"ERROR %d:openfile error!!!---cmd_write()\\n",NOFILEOPEN);
374         return;
375     }
376     for(i=0;i < profile_data_nitems;i++){
377         fprintf(fp,"%d,",profile_data_store[i].id);
378         fprintf(fp,"%s,",profile_data_store[i].name);
379         fprintf(fp,"%04d-%02d-%02d,",profile_data_store[i].found.y,profile_data_store[i].found.m,profile_data_store[i].found.d);
380         fprintf(fp,"%s,",profile_data_store[i].add);
381         fprintf(fp,"%s",profile_data_store[i].others);
382         fprintf(fp,"\\n");
383     }
384     fclose(fp);
385     fprintf(stderr,"wrote %s\\n",filename);
386     return;
387 }

```

```

387
388 void cmd_binread(char *filename){
389     return;
390 }
391
392 void cmd_binwrite(char *filename){
393     FILE *fp;
394     int i=0;
395     if (fopen(filename, "wb") == NULL)
396     {
397         fprintf(stderr,"ERROR %d:openfile error!!!---cmd_write()\n",NOFILEOPEN);
398         return;
399     }
400     fwrite(&profile_data_store[0],sizeof(struct profile),1,fp);
401     fclose(fp);
402     fprintf(stderr,"wrote %s\n",filename);
403     return;
404 }
405
406 char *date_to_string(char buf[],struct date *date){
407     sprintf(buf,"%04d-%02d-%02d",date->y,date->m,date->d);
408     return buf;
409 }
410
411 void cmd_find(char *keyword){
412     int i,check=0;
413     struct profile *p;
414     char found_str[11];
415
416     for(i=0;i < profile_data_nitems;i++){
417         p=&profile_data_store[i];
418         date_to_string(found_str,&p->found);
419         if(
420             (p->id) == strtol(keyword,endl,base1)||
421             strcmp(p->name,keyword)==0||
422             strcmp(p->add,keyword)==0||
423             strcmp(p->others,keyword)==0||
424             strcmp(found_str,keyword)==0
425         ){
426             printdata(p,i);
427             check=1;
428         }
429     }
430
431     if(check==0){
432         fprintf(stderr,"No match data.\n");
433     }
434
435     return;
436 }
437
438 void cmd_findb(char *keyword){
439     int i,check=0;
440     struct profile *p;
441     char found_str[11];
442
443     for(i=0;i < profile_data_nitems;i++){
444         p=&profile_data_store[i];
445         date_to_string(found_str,&p->found);
446         if(
447             (p->id) == strtol(keyword,endl,base1)||
448             find_kai(p->name,keyword)==0||
449             find_kai(p->add,keyword)==0||
450             find_kai(p->others,keyword)==0||
451             find_kai(found_str,keyword)==0
452         ){
453             printdata(p,i);
454             check=1;
455         }
456     }
457     if(check==0){
458         fprintf(stderr,"No match data.\n");
459     }
460     return;
461 }
462

```

```

463 int find_kai(char *s, char * cp){
464     char *s1, *s2;
465     if( *cp == '\0') return 1; /*cp の文字列長が 0 なら s を返す*/
466
467     while( *s != '\0'){
468         while(*s != '\0' && *s != *cp) { /*先頭文字が合うまで探す*/
469             s++;
470         }
471         if(*s == '\0') return 1; /*見つからない*/
472         s1 = s;
473         s2 = cp;
474         while ( *s1 == *s2 && *s1 != '\0'){ /*cp の先頭以降の文字列が一致するか*/
475             s1++;
476             s2++;
477         }
478         if( *s2 == '\0'){ /* cp の文字列は、全て一致した*/
479             return 0;
480         }
481         s++; /*次の位置から、調べ直す*/
482     }
483     return 1; /*見つからない*/
484 }
485
486 void swap_struct(struct profile *i, struct profile *j){
487     struct profile temp;
488
489     temp = *j;
490     *j = *i;
491     *i = temp;
492
493     return;
494 }
495
496 int compare_profile(struct profile *p1, struct profile *p2, int youso){
497     if(youso < 0)youso*=-1;
498     switch (youso) {
499         case 1:
500             return (p1->id) - (p2->id);break;
501
502         case 2:
503             return strcmp(p1->name,p2->name);break;
504
505         case 3:
506             return compare_date(&p1->found,&p2->found);break;
507
508         case 4:
509             return strcmp(p1->add, p2->add);break;
510
511         case 5:
512             return strcmp(p1->others, p2->others);break;
513
514         default:
515             return 0;break;
516     }
517 }
518
519 int compare_date(struct date *d1, struct date *d2){
520     if (d1->y != d2->y) return d1->y - d2->y;
521     if (d1->m != d2->m) return d1->m - d2->m;
522     return (d1->d) - (d2->d);
523 }
524
525 void cmd_sort(int youso){
526     int i,j;
527     int check=0;
528
529     if(youso>5||youso<1){
530         fprintf(stderr,"ERROR %d:sort param is 1 to 5.---cmd_sort()\n",PARAMERROR);
531         return;
532     }
533
534     if(profile_data_nitems<=0){
535         return;
536     }
537

```

```

538
539     for(i=0;i<profile_data_nitems;i++){
540         for(j=0;j<profile_data_nitems-1;j++){
541             if(compare_profile(&profile_data_store[j],&profile_data_store[j+1],youso) >
542                 0){
543                 swap_struct(&profile_data_store[j],&profile_data_store[j+1]);
544                 check++;
545             }
546         }
547     }
548     fprintf(stderr,"%d swap.\n",check);
549     return;
550 }
551
552 void cmd_qsort(int youso){
553     if(youso>5||youso<1){
554         fprintf(stderr,"ERROR %d:sort param is 1 to 5.---cmd_sort()\n",PARAMERROR);
555         return;
556     }
557     if(profile_data_nitems<=0){
558         return;
559     }
560     quick_sort(0,profile_data_nitems-1,youso);
561     fprintf(stderr,"quicksort end.===count:%d\n",quick_count);
562     quick_count=0;
563     return;
564 }
565
566 void quick_sort(int left, int right,int youso){
567     int i,j,pivot;
568
569     i=left;
570     j=right;
571     pivot=right;
572
573     while(1){
574         while (compare_profile(&profile_data_store[i],&profile_data_store[pivot],youso)
575             < 0){
576             i++;
577         }
578         while (compare_profile(&profile_data_store[pivot],&profile_data_store[j],youso)
579             < 0){
580             j++;
581         }
582         if(i>=j)break;
583         swap_struct(&profile_data_store[i],&profile_data_store[j]);
584         quick_count++;
585         i++;
586         j--;
587     }
588
589     if (left < i - 1){ /* 基準値の左に 2 以上要素があれば */
590         quick_sort(left, i-1,youso);/* 左の配列をソートする */
591     }
592     if (j + 1 < right){ /* 基準値の右に 2 以上要素があれば */
593         quick_sort(j+1, right,youso);/* 右の配列をソートする */
594     }
595     return;
596 }
597
598 void cmd_delete(int param){
599     int i;
600     if( param > profile_data_nitems||param <=0){
601         fprintf(stderr,"ERROR %d:error param.--cmd_delete()\n",OVERNUMBERRECORD);
602         fprintf(stderr,"ERROR %d:number of item is %d\n",NUMITEM,profile_data_nitems);
603         return;
604     }
605     for(i=param-1;i<profile_data_nitems-1;i++){
606         profile_data_store[i]=profile_data_store[i+1];
607     }
608     profile_data_nitems--;
609     return;
610 }

```



```

610
611 struct profile *new_profile(struct profile *pro,char *str){
612     char *ret1[maxsplit],*ret2[maxsplit-2];
613     int count=0;
614     if(profile_data_nitems>=10000){
615         fprintf(stderr,"ERROR %d:Can't add record--new_profile()\n",OVERNITEMS);
616         return NULL;
617     }
618     count=split(str,ret1,',',maxsplit);
619     if(count!=maxsplit){
620         error_split(count);
621         fprintf(stderr,"ERROR %d:wrong format of input(ex.001,name,1999-01-01,address,
622             other)--new_profile()\n",FORMATINPUT);
623         return NULL;
624     }//文字列用
625     pro->id = strtol(ret1[0],endp,base1);
626     if( pro->id == 0){
627         fprintf(stderr,"ERROR %d:ID is NUMBER.--new_profile()\n",FORMATID);
628         return NULL;
629     }
630
631     strncpy(pro->name, ret1[1],70);//名前のコピー
632     strncpy(pro->add, ret1[3],70);//住所
633     pro->others = (char *)malloc(sizeof(char)*(strlen(ret1[4])+1));
634     strcpy(pro->others, ret1[4]);//備考,MAX 1024bytes
635
636     if(split(ret1[2],ret2,'-',maxsplit-2)!=maxsplit-2){
637         fprintf(stderr,"ERROR %d:wrong format of date.(ex.1999-01-01)--new_profile()\n",
638             FORMATDATE);
639         return NULL;
640     }//設立日
641     pro->found.y = strtol(ret2[0],endp,base1);
642     pro->found.m = strtol(ret2[1],endp,base1);
643     pro->found.d = strtol(ret2[2],endp,base1);
644
645     fprintf(stderr,"Add profile.\n");
646     profile_data_nitems++;
647     return pro;
648 }
649
650 void cmd_size(){
651     fprintf(stderr,"struct profile = %d\n",sizeof(struct profile));
652     fprintf(stderr,"id = %d\n",sizeof(profile_data_store[0].id));
653     fprintf(stderr,"name = %d\n",sizeof(profile_data_store[0].name));
654     fprintf(stderr,"add = %d\n",sizeof(profile_data_store[0].add));
655     fprintf(stderr,"found.y = %d\n",sizeof(profile_data_store[0].found.y));
656     fprintf(stderr,"found.m = %d\n",sizeof(profile_data_store[0].found.m));
657     fprintf(stderr,"found.d = %d\n",sizeof(profile_data_store[0].found.d));
658     fprintf(stderr,"Com. = %d\n",sizeof(profile_data_store[0].others));
659     return;
660     //araiment seiyaku
661 }

```

---