プログラミング通論'19 ＃8 – 双連結リスト

久野 靖 (電気通信大学)

2019.4.20

今回は次のことが目標となります。

- 双連結リストの考え方と操作方法を理解する。
- 連結リストを使ったエディタアプリケーションについて知る。

双連結リスト

連結リストのバリエーション

連結リスト (linked list) はセルが直線的に並んだ動的データ構造です。前回までに扱った単連結リスト (単リスト) では、セルはデータに加えて次のセルを指すポインタ値のフィールド **next** を持ち、これによって片方向にセルを連結していました (図 1)。先頭や末尾へのセルの挿入・削除の操作をしやすくするために、頭 (head) や尻尾 (tail) と呼ばれるダミーセル (実際のデータは入れないセル) を常に置く方法があります。

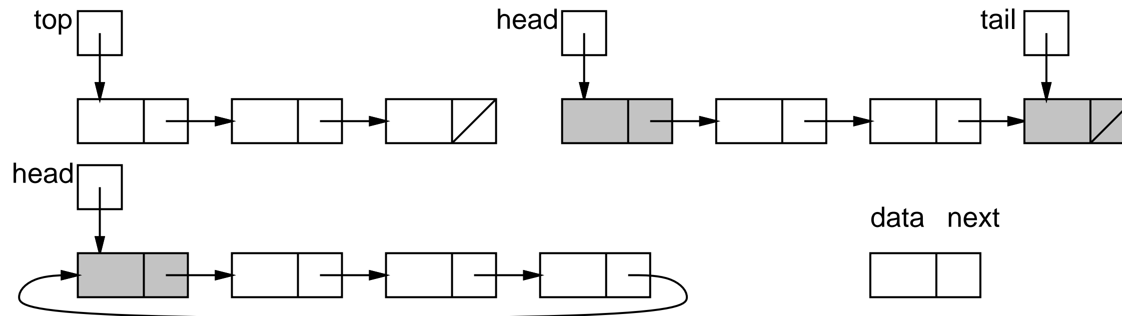

図 1: 単連結リストのバリエーション

連結リストのバリエーション (2)

リストの終わりのポインタには**NULL**または**nil**と呼ばれる特別な値を入れます(前回までアース記号で表していましたが、ここからは簡単のため斜線で表すようにしました)。最後のセルが先頭のセルを指すようにしたものを循環リストと呼び、この場合は**NULL**を使わなくて済みます(間違って**NULL**をたどるとプログラムが死ぬのでそれを避けられるという利点があります)。循環リストでも頭を持たせることができます。

単連結リストの弱点は、途中のセルをポインタで指している場合、次のセルをたどるのは容易だが、前のセルをたどることは手間が掛かる(先頭のポインタから繰り返し次をたどり、現在のセルの1つ手前まで来る必要がある)ことです。

このため、現在のセルの「次に」新しいセルを挿入したり、現在のセルの「次の」セルを削除することは簡単ですが、現在のセルの「前に」新しいセルを挿入したり、現在のセル「そのものを」削除するのは手間が掛かります。

これらの弱点を解消できる連結リストの方式に、双連結リスト(**double linked list**、双リスト、双方向リスト)があります(図**2**)。

連結リストのバリエーション (3)

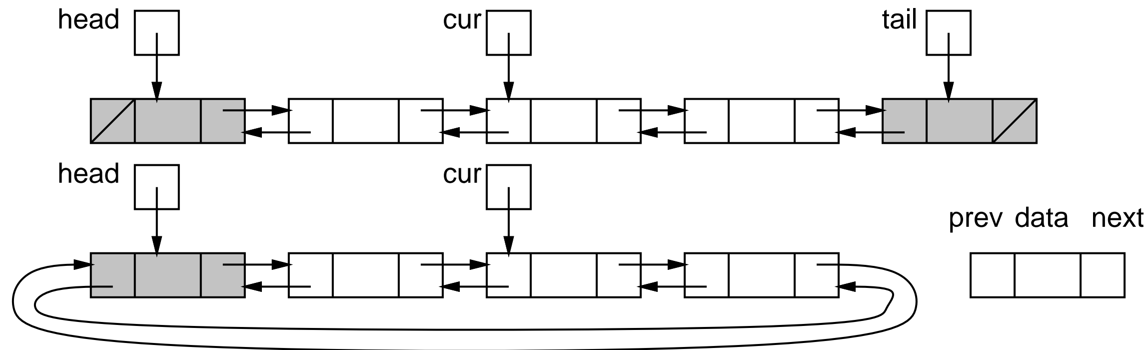

図 2: 双連結リスト (双方向リスト)

双連結リストでは、**1** つ先のセルへのポインタ **next** に加えて、**1** つ前のセルへのポインタ **prev** も持つため、これを使って前方向へリストをたどれます。

双連結リストの弱点は、**1** つのセル当たりポインタのフィールドが **1** つ増えるので領域が余分に掛かることですが、今日ではメモリは潤沢にあるのであまり問題ありません。ポインタの付け変えの手間も単連結リストより多く (倍に) なりますが、それよりも操作しやすさによる利点の方が大きいと考えます。

双連結リストでも頭や尻尾を持たせることがあります。また、双連結リストの循環リストも多く使われます。

双連結リストを使ったエディタバッファ

それでは、双連結リストを使ったエディタバッファを作ってみましょう。エディタバッファというのは何かというと要するに、行単位で文字列を格納し、それぞれの行に移動したりその内容を書き換えたりできるような機能を指すものとします。例によって構造体で情報隠蔽します。

今回は簡単のため、それぞれのノードに100バイト(100文字)の文字配列を含め、そこに1行ぶんの文字列を格納します。なので、その100という長さもMAXSTRという名前でヘッダに定義します。ヘッダファイルは次の通り。

双連結リストを使ったエディタバッファ (2)

```
// ebuf.h --- editor buffer API.
#include <stdbool.h>
#define MAXSTR 100
struct ebuf;
typedef struct ebuf *ebufp;
ebufp ebuf_new();              // create ebuf
bool ebuf_iseof(ebufp e);      // see if current line is EOF
bool ebuf_forward(ebufp e);    // forward 1 line
bool ebuf_backward(ebufp e);   // backward 1 line
void ebuf_top(ebufp e);        // go to top
char *ebuf_str(ebufp e);      // obtain current line string
void ebuf_insert(ebufp e, char *s); // insert a line
```

双連結リストを使ったエディタバッファ (3)

バッファの最後には「**EOF(end of file のつもり)**」と書かれた特別な行 (**EOF** 行) があり、バッファが空のときは現在行は **EOF** 行です。`ebuf_iseof` で **EOF** 行にいるかどうか調べられます。また、`ebuf_top` で先頭に行き、`ebuf_forward` と `ebuf_backward` で **1** 行先/前へ動けますが、**EOF** 行より先や先頭より前は行けないのでそのような場合は `false` を返します。`ebuf_str` は現在行の文字列 (先頭文字へのポインタ) を返します。そして `ebuf_insert` は現在行の上に指定した文字列を持つ行を挿入します。

これを使った例をまず見ましょう。バッファを作り、**3** 行ぶん文字列を挿入し、先頭から順に表示して、そのあとまた前に戻りながら各行を表示します。

双連結リストを使ったエディタバッファ (4)

```
// ebufdemo.c --- demonstration of ebuf.
#include <stdio.h>
#include "ebuf.h"

int main(void) {
  ebufp e = ebuf_new();
  ebuf_insert(e, "abc");
  ebuf_insert(e, "def");
  ebuf_insert(e, "ghi");
  ebuf_top(e);
  while(!ebuf_iseof(e)) {
    printf("%s\n", ebuf_str(e)); ebuf_forward(e);
  }
  while(ebuf_backward(e)) { printf("%s\n", ebuf_str(e)); }
  return 0;
}
```

双連結リストを使ったエディタバッファ (5)

実行例は次の通り。

```
% gcc8 ebufdemo.c ebuf.c
% ./a.out
abc
def
ghi
ghi
def
abc
```

では実装を見てみましょう。内部では双連結循環リストを使いますから、構造体 ebuf は頭のセルへのポインタと現在セルへのポインタがあればよいです。セルそのものは構造体 line で表し、そのフィールドといては前後セルへのポインタと char の配列が含まれます。

双連結リストを使ったエディタバッファ (6)

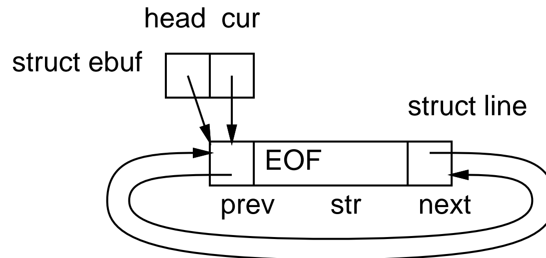

図 3: エディタバッファの初期状態

新しいバッファを作るときは(図3)、まず ebuf の構造体を割り当て、head フィールドに **EOF** 行のセルを指させます。そのあと、cur フィールドにも **EOF** 行の prev と next にもそのセルへのポインタを指させます。これで循環リストが初期化できました。最後に **EOF** 行には文字列「**EOF**」を格納し、構造体 ebuf のポインタを返します。

双連結リストを使ったエディタバッファ (7)

```
// ebuf.c --- editor buffer implementation
#include <stdlib.h>
#include <string.h>
#include "ebuf.h"
struct line {
  struct line *prev, *next; char str[MAXSTR];
};
struct ebuf { struct line *head, *cur; };
ebufp ebuf_new() {
  ebufp r = (ebufp)malloc(sizeof(struct ebuf));
  r->head = (struct line*)malloc(sizeof(struct line));
  r->cur = r->head->next = r->head->prev = r->head;
  strcpy(r->head->str, "EOF"); return r;
}
bool ebuf_iseof(ebufp e) { return e->cur == e->head; }
bool ebuf_forward(ebufp e) {
```

```
  if(e->cur == e->head) { return false; }
  e->cur = e->cur->next; return true;
}
bool ebuf_backward(ebufp e) {
  if(e->cur->prev == e->head) { return false; }
  e->cur = e->cur->prev; return true;
}
void ebuf_top(ebufp e) { e->cur = e->head->next; }
char *ebuf_str(ebufp e) { return e->cur->str; }
```

双連結リストを使ったエディタバッファ (8)

```
void ebuf_insert(ebufp e, char *s) {
  struct line *p = (struct line*)malloc(sizeof(struct line));
  strncpy(p->str, s, MAXSTR); p->str[MAXSTR-1] = '\0';
  p->prev = e->cur->prev; p->next = e->cur;
  e->cur->prev->next = p; e->cur->prev = p;
}
```

headは常に**EOF**行を指しているので、**EOF**行かどうかはcur がheadと等しいかどうか見れば分かります。**1**行進む場合はcurをcur->nextに変更すればいいのですが、その前に**EOF**行を超えて進まないようにチェックしています。**1**行戻る場合も同様です。先頭行は(循環リストなので)**EOF**行の次にcurを設定するだけです。文字列を返すのはcur->strを返すだけです。

双連結リストを使ったエディタバッファ (9)

最後の挿入が最もややこしいですが、まず新しい行のセルを割り当て、パラメタの文字列を strncpy でコピーし、万一最後のナル文字がないと (100 文字以上あったとき) トラブルになりがちなので最後にナル文字を格納します。その後が連結リストの操作で、まず新しいセルの次を現在行、前を現在行の 1 つ前とします。続いて、現在行の 1 つ前の次と現在行の 1 つ前をいずれも新しいセルとします (図 4)。cur->prev を先に書き換えてしまうとまずいので注意が必要です。

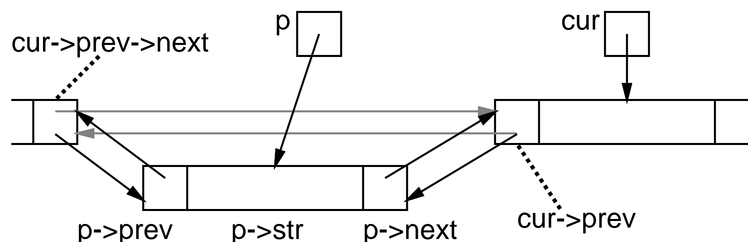

図 4: 双連結リストへの挿入

きちんと動くことを確認するため、単体テストを作りましょう。バッファのデータは文字列なので、#03 で作った expect_str をを使います。

双連結リストを使ったエディタバッファ (10)

```
// test_ebuf.c --- unit test for cbuf.
#include <stdio.h>
#include <string.h>
#include "ebuf.h"
void expect_str(char *s1, char *s2, char *msg) {
  printf("%s '%s':'%s' %s\n", strcmp(s1, s2)?"NG":"OK", s1, s2, msg);
}
int main(void) {
  ebufp e = ebuf_new(); ebuf_insert(e, "abc"); ebuf_insert(e, "def");
  ebuf_top(e); expect_str(ebuf_str(e), "abc", "line 1: abc");
  ebuf_forward(e); expect_str(ebuf_str(e), "def", "line 2: def");
  ebuf_insert(e, "ghi"); ebuf_top(e);
  ebuf_forward(e); expect_str(ebuf_str(e), "ghi", "new line 2: ghi");
  ebuf_forward(e); expect_str(ebuf_str(e), "def", "new line 3: def");
  return 0;
}
```

双連結リストを使ったエディタバッファ (11)

動かしたようすは次の通り。

```
% ./a.out
OK 'abc':'abc' line 1: abc
OK 'def':'def' line 2: def
OK 'ghi':'ghi' new line 2: ghi
OK 'def':'def' new line 3: def
```

双連結リストを使ったエディタバッファ (12)

演習 1 エディタバッファの例題をそのまま動かせ。単体テストも動かしてみること。**OK** なら、次のことをやってみよ。追加した機能が正しく動くことを確認できるように単体テストを作成すること。

a. `ebuf` に現在行の内容を指定した文字列に取り換える命令 `ebuf_replace` を追加する。

b. 削除命令 `ebuf_delete` を追加する。**EOF** 行は消さないようにすること。

双連結リストを使ったエディタバッファ (13)

c. 上と同様だが、ただし「別の」リストを用意しておき、図削除したセルがそちらにつながるようにする。そして、その別のリストからセルを外して現在位置の上に挿入する ebuf_yank を作る (図 5 のように、消してから移動して別の場所で戻すことで行を移動できるようになる)。

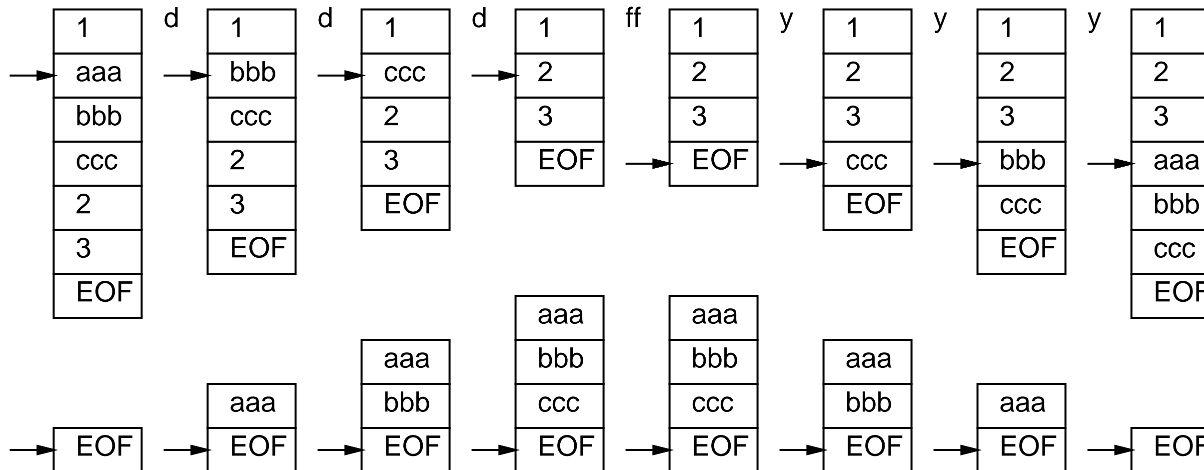

図 5: カットバッファの実装

d. 上記に加えて、「削除はせずに現在行のコピーを別の場所に挿入する」命令 ebuf_copy を作る (現在行は 1 つ下に移る)。これにより、複数行をコピーして別の場所に挿入できるようになる。

双連結リストを使ったエディタバッファ (14)

e. 図 **5** を見ると、「別の場所」も本体のバッファと同じ構造でよさそうである。そこで実際にそのようにして、「本体の場所と別の場所を交換する」命令 `ebuf_swap` を作る (交換したあとの現在位置は **EOF** 行でよい)。これにより、数行だけ必要なときは必要な行だけ消してから交換すればよくなる。

f. バッファの内容を上下ひっくり返す (先頭行が最後になる) 機能を追加する。

g. そのほか、エディタバッファにあると便利と思う機能を追加する。

行エディタを作る
基本的な行エディタ

行エディタ (**line editor**) とは、行が編集できるテキストエディタ···ではなく (普通のテキストエディタはどれも行は編集できます)、行単位で現在位置を移動したり内容を編集できるような、画面エディタ (**screen editor**) ではないエディタを言います。

画面エディタとは画面上に編集バッファの内容が常に表示されていて、それを見ながら挿入や削除ができるもので、今日の普通のエディタはすべてそうです。しかし、現在のような画面上の表示ができるようになる前に、タイプライタ端末 (表示はタイプライタのように紙に印字することで行う) しか無かった時代があり、そのときは画面エディタが使えず、行エディタが多く使われていました (今でも **Unix** には **ed** など行エディタが標準で備わっています)。

基本的な行エディタ (2)

ここでは簡単な行エディタを作ってみせます。コマンドは次のものです。

- q — エディタを終わる
- p — 現在行を表示
- t — 先頭行に行く
- f — 1 行下に行く
- b — 1 行上に行く
- i 文字列 — 文字列を行として現在行の直前に挿入
- 空行 (その他任意の行) — f と p の動作を実行

基本的な行エディタ (3)

実際に動かす様子を見てみましょう。

```
% gcc8 editor.c ebuf.c
% ./a.out
> iThis is a pen.   ←挿入
> iThat is a dog.   ←挿入
> t                 ←先頭へ
> p                 ←表示
  This is a pen.
> f                 ←1行下へ
> iHow are you?     ←挿入
> t                 ←先頭へ
> p                 ←表示
  This is a pen.
>                   ←次の行へ行き表示
  How are you?
>                   ←次の行へ行き表示
```

```
    That is a dog.
>                        ←次の行へ行き表示
    EOF
> q                      ←終了
```

基本的な行エディタ (4)

まあ、使いやすくはないかもですが、一応編集はできそうです (削除は後で追加するとして)。ではコードを見てみましょう。getl は前から使っているものです。

```
// editor.c --- a simple line editor.
#include <stdio.h>
#include "ebuf.h"

bool getl(char s[], int lim) {
  int c, i = 0;
  for(c = getchar(); c != EOF && c != '\n'; c = getchar()) {
    s[i++] = c; if(i+1 >= lim) { break; }
  }
  s[i] = '\0'; return c != EOF;
}
int main(void) {
  char buf[200];
```

```
ebufp e = ebuf_new();
printf("> ");
while(getl(buf, 200)) {
  if(buf[0] == 'q') { // quit
    break;
  } else if(buf[0] == 'p') { // print
    printf("   %s\n", ebuf_str(e));
  } else if(buf[0] == 't') { // top
    ebuf_top(e);
  } else if(buf[0] == 'f') { // fwd
    ebuf_forward(e);
  } else if(buf[0] == 'b') { // back
    ebuf_backward(e);
  } else if(buf[0] == 'i') { // insert
    ebuf_insert(e, buf+1);
  } else {  // other --- fwd and print
    ebuf_forward(e); printf("   %s\n", ebuf_str(e));
```

```
    }
    printf("> ");
  }
  return 0;
}
```

要するに、**1**行読み込み、先頭の文字に応じてそれぞれのコマンドの動作を実行すればよいわけです。

ファイルの読み書き

ファイルが読み書きできないと実際の編集に使えないので、とりあえず最低限を説明します。読むときも書くときも、次の手順によります。

- 「`FILE *f = foepn(ファイル名, モード);`」でストリームを開き、結果を`FILE*`型として受け取ります。ストリームとは、読み書きする対象の文字をやりとりする「通路」だと思ってください。何か問題があれば (例: ファイルが無い、ディレクトリ間違い等)、`NULL`が返されます。
- 書くときは`printf`の類似版「`fprintf(ストリーム, 書式文字列, 値)`」を使うことで任意のものが出力できます。
- 読むときは1行単位で読むのが簡単ですが、それには「`fgets(文字配列, 長さ上限, ストリーム)`」を使い、文字配列に1 行ぶん読み込みます。ただしこの場合、改行文字が最後についています。`fgets`はこれ以上読めない場合 (ファイルの終わり等) は`NULL`を返します。
- いずれにせよ、最後は「`fclose(ストリーム)`」で閉じる (後始末する) 必要があります。

ファイルの読み書き (2)

では、読む方から見ましょう。fopenに失敗したらすぐ戻ります。そうでない場合はfgetsを繰り返し呼び(NULLが返されたら終わる)、読み込んだ文字列の最後の文字(改行文字のはず)をナル文字に書き換えてからebufに挿入します。

```
bool readfile(ebufp e, char *fname) {
  char str[200];
  FILE *f = fopen(fname, "r");
  if(f == NULL) { return false; }
  while(fgets(str, 200, f) != NULL) {
    int len = strlen(str);
    if(len > 0) { str[len-1] = '\0'; }
    ebuf_insert(e, str);
  }
  fclose(f); return true;
}
```

ファイルの読み書き (3)

書く方がどちらかといえば簡単です。fopenに失敗したらすぐ戻ります。そうでない場合は、まず先頭に行き、**EOF**行でない間、現在行をfprintfで出力してから次の行に進みます。

```
bool writefile(ebufp e, char *fname) {
  FILE *f = fopen(fname, "w");
  if(f == NULL) { return false; }
  ebuf_top(e);
  while(!ebuf_iseof(e)) {
    fprintf(f, "%s\n", ebuf_str(e)); ebuf_forward(e);
  }
  fclose(f); return true;
}
```

呼び出し方ですが、先のmainでiコマンドと同様、readfile(e, buf+1)、writefile(e, buf+1)のように呼び出せばよいですが、読み書きが失敗だったとき(falseが返されたとき) は何かその旨を出力した方がよいでしょう。

ファイルの読み書き (4)

演習 2 前節にある簡単な行エディタを動かし、動作を確認しなさい。**OK** なら、以下のことをやってみなさい。実際に簡単なプログラムを改良したエディタで作成/編集してみて体験を報告すること。

a. ファイル読み書きコマンド「r ファイル名」「w ファイル名」を追加する。

b. 移動コマンドを「f 行数」「b 行数」のように何行移動するかを指定もできるようにする (他のコマンドや以下で追加するコマンドも必要に応じて同様に)。

b. 行削除コマンド「d」を追加する。

c. 行の内容を取り換えるコマンド「s 文字列」を追加する。

d. 演習 **1c** のように削除した結果を戻せるコマンド「y」を追加する。

e. 演習 **1d** のようにカットバッファと現在のバッファを交換するコマンド「x」を追加する。バッファをもっと多数持てるようにしてもよい。

f. その他あったらよいと思う機能を追加する。

画面エディタ option
ncursesとその機能

エディタが作れるようになりましたが、やはり行エディタよりは画面エディタの方がいいですね。画面エディタを作るには、画面の任意の位置で文字を表示したり消したりできる必要があります。その方法は色々ありますが、ここでは**Unix**に備わっている、端末(ターミナル)上で画面制御を行うライブラリ**ncurses**を使います。いきなり例題を見てみましょう。

ncursesとその機能 (2)

```
// ncursesdemo.c --- show usage of ncurses.
#include <ncurses.h>
#include <stdlib.h>

int main(void) {
  initscr(); noecho(); cbreak(); system("stty raw"); clear();
  move(10, 10); addstr("press any key"); refresh();
  int ch = getch(); addch('a'); addch('b'); refresh();
  ch = getch(); move(10, 15); insch('a'); insch('b'); refresh();
  ch = getch(); delch(); move(10, 20); clrtoeol(); refresh();
  ch = getch(); endwin(); return 0;
}
```

ncursesとその機能 (3)

ヘッダファイルはncurses.hとstdlib.hが必要です。冒頭部分はだいたい固定で、initscr()でncursesの初期化をおこない、noechoでキー入力を画面にそのまま表示しないモードに変更し(エディタなので表示は自前で制御したい)、cbreakで1文字入力したらすぐそれが読めるモードにし(通常は改行文字が来るまでプログラムには渡さない)、さらにsystemコマンドはUnixコマンドを実行する関数ですが、それを使って「^Cなど制御文字の扱いを止める」ようにします(^Cを打ったらエディタが強制終了してしまうのでは悲しいですから)。これらと対になる後始末はendwinで、これでncursesが各種状態を復元してくれます。

プログラム中で繰り返し使う個別の機能は箇条書で説明しましょう。refreshが分かりづらいと思いますが、エディタでは「getchで1文字読む直前にrefreshを読んで画面を更新する」と思っていればよいです。

ncursesとその機能 (4)

- `move(`Y, X`)` — 上から Y 行目の X 文字目にカーソルを移動する。
- `addch(`C`)`、`addstr(`S`)` — カーソル位置に1文字または文字列を表示し、カーソルは表示したぶんだけ進める。
- `insch(`C`)` — カーソル位置に文字を挿入し、以後の文字を1文字右にずらす。カーソル位置は変化しない。
- `delch()` — カーソル位置の文字をし、右側の文字を文字左に詰める。
- `clear()`、`cleartoeol()` — 画面全体をクリア、カーソル位置から右側をクリア。
- `refresh()` — ここまでに呼んだ機能を実際に適用して画面を更新する(これを呼ぶまでは内部で保留している)。
- `getch()` — キーボードから1文字入力して文字を返す。

コンパイル時のオプションとして「`-lncurses`」が必要です。

```
% gcc8 ncursesdemo.c -lncurses
% ./a.out
```

ncursesとその機能 (5)

画面制御を行うプログラムは例示がしにくいですが、起動するととりあえず画面の10行目の10文字目以降に次のようにメッセージが表示されます。

press any key_

ここで何かキーを打つと追加の文字列が表示されます。

press any keyab_

再度キーを打つと今度は行の途中に挿入がなされます。

pressba any keyab

再度キーを打つとさっきのカーソル位置の文字と20文字目以降が消えます。

pressa any_

演習3 ncursesを使って何か「画面上でお絵描きする」プログラムを作ってみなさい。

1行ウィンドウの画面エディタ

ではなるべく簡単な例として、「ウィンドウサイズが1行だけの(カーソルのある行だけが見えている)」画面エディタを作ります。ファイルの冒頭部分を示します(あとreadfile、writefileも必要。

```
// sedit.c --- very primitive screen editor w/ ncurses.
#include <stdio.h>
#include <stdbool.h>
#include <ncurses.h>
#include <stdlib.h>
#include <string.h>
#include "ebuf.h"
// readfile, writefile here
```

1行ウィンドウの画面エディタ (2)

次に、画面エディタでは行中の任意の位置pで文字を挿入したり消したりするので、その処理を行う下請け関数を用意します。挿入のときは行の最後から前に向かって1文字ずつずらして場所をあけます。いずれも文字列の長さは渡すこととします。

```
void inschar(char *s, char ch, int p, int len) {
  int i;
  for(i = len + 1; i > p; --i) { s[i] = s[i-1]; }
  s[p] = ch;
}
void delchar(char *s, int p, int len) {
  int i;
  for(i = p; i < len; ++i) { s[i] = s[i+1]; }
}
```

1行ウィンドウの画面エディタ (3)

これらを呼び出しながら、また**ncurses**の関数を呼び出しながら、文字の挿入や削除およびカーソル移動を行う関数`handleline`を示します。自分が取り扱わないコマンド(1文字ですが)が来たときは`false`を返し、取り扱えた場合は`true`を返します。また、この関数の中でカーソル位置や文字列の長さを変化させるので、この2つについては整数へのポインタを渡してもらい、間接参照を使ってもとの(呼び出し側の)変数を変更します。

1行ウィンドウの画面エディタ (4)

```
bool handleline(int c, char *s, int *pos, int *len) {
  if(c >= ' ' && c <= '~') {
    if(*len >= MAXSTR-1) { return false; }
    insch(c); inschar(s, c, *pos, (*len)++); move(10, ++(*pos));
  } else if(c == 'B'-'@') {
    if(*pos > 0) { move(10, --(*pos)); }
  } else if(c == 'F'-'@') {
    if(*pos < *len) { move(10, ++(*pos)); }
  } else if(c == 'H'-'@') {
    if(*pos <= 0) { return false; }
    move(10, *pos - 1); delch(); delchar(s, --(*pos), (*len)--);
  } else {
    return false;
  }
  return true;
}
```

1行ウィンドウの画面エディタ (5)

まず文字がスペースからチルダまでの範囲であれば普通の文字なので、その文字を挿入しますが、その前にもし文字列の長さが`MAXSTR-1` 以上ならこれ以上増やせないのでだめですといって帰ります。**OK** の場合はカーソル位置に文字を挿入し、また文字列の方もその位置に文字を挿入し、長さを**1**増やします。カーソル位置は**1**進めます (`insch`はカーソル位置を変更しないので`move`で変更する必要があります)。

その先、コマンドが`^F`や`^B`の場合は (コントロール文字の文字コードは「その文字の大文字のコードから@のコードを引いた値」になっています)、カーソル位置を増減しますが、**0**より手前や行長より先には行けないようにしています。

BS文字 (文字コードでは`^H`と同じ) の場合、行頭でなければカーソル位置の**1**つ手前の文字を消し、長さも**1**減らします。これら以外の文字は`false`を返します。

1行ウィンドウの画面エディタ (6)

ではmainを見てみましょう。まずebufを作り、readfileでファイルを読み込みますが、そのファイル名はコマンド引数で渡したファイルということにしました。そして先頭行に行き、各種変数を用意しますが、変数showは現在の1行を表示し直すかどうかを制御するフラグです(最初および行が別の行に変化した時に「true」にします)。次に**ncurses**の初期設定を行い、無限ループに入ります。ループの冒頭でshowが「はい」なら文字列、長さ、カーソル位置を変数にいれ、画面に表示し、カーソル位置は行の先頭にし、showはfalseに変更します。その後、refreshを呼んでからキーボード入力を待ちます。

1行ウィンドウの画面エディタ (7)

```
int main(int argc, char *argv[]) {
  ebufp e = ebuf_new();
  if(!readfile(e, argv[1])) { printf("?\n"); return 1; }
  ebuf_top(e);
  int len, pos, ch; char *str;
  bool show = true;
  initscr(); noecho(); cbreak(); system("stty raw"); clear();
  while(true) {
    if(show) {
      str = ebuf_str(e); len = strlen(str); pos = 0;
      move(10, 0); addstr(str); clrtoeol(); move(10, 0); show = false;
    }
    refresh(); ch = getch();
    if(handleline(ch, str, &pos, &len)) {
      // do nothing
    } else if(ch == 'J'-'@') {
```

```
      ebuf_forward(e); ebuf_insert(e, "");
      ebuf_backward(e); show = true;
    } else if(ch == 'N'-'@') {
      ebuf_forward(e); show = true;
    } else if(ch == 'P'-'@') {
      ebuf_backward(e); show = true;
    } else if(ch == 'Z'-'@') {
      break;
    }
  }
  endwin(); writefile(e, argv[1]); return 0;
}
```

1行ウィンドウの画面エディタ (8)

キー入力があったら、まず`handleline`で行内編集コマンドとしての処理を試み、**OK**ならそれで終わりで次の周回へ行きます。**OK**でないなら、他のコマンドですが、ここでは改行 (コードは`^J` — 1行下へ行き、直前に空行を挿入し、そこへ行く)、上下の行への移動 (`^N`と`^P`) があります。これらでは行が変わるので`show`を「はい」にする必要があります。そして終了は`^Z`で、このときは**curses**を後始末して**ebuf**の内容を元のファイルに書き戻して終わります。実際に動かしてみると、1行しか表示されないわりには、それなりに編集できる感じです。試してみてください。

1行ウィンドウの画面エディタ (9)

演習4 1行ウィンドウの画面エディタを動かしてみなさい。適当なプログラムのファイルを編集してみること。動いたら、次のような改良をしてみなさい。

a. 1行ウィンドウはつらいので、上の方や下の方も一定行数(たとえば5行とか)表示されるようにする。現在位置は10行目で固定でよいです。

b. emacsのようにカーソル位置が画面内で自由に動かせ、画面から外に出そうになったらスクロールするようにする。

c. `^J`のとき空行を挿入するのでなく、カーソル位置で現在の行を2つに分けるようにする。

d. 行の先頭で`^H`のときに無視するのでなく、前の行と今の行がくっつくようにする。カーソル位置もくっついた位置になっているとなおよい。

e. `^F`や`^B`で行末や行頭を超えようとしたときに無視するのでなく次の行や前の行に移るようにする。(ヒント: 現在は`handleline`でこれらの文字は常に「はい」が返るようになっていので、これ以上行けないときは「いいえ」を返すように修正する必要があります。)

1行ウィンドウの画面エディタ (10)

f. 演習1や演習2で扱ったような機能を使えるようにする。

g. ファイルの読み書きをコマンド引数で指定したファイル固定でなく、画面からファイル名を入力して行えるようにする。

h. その他、あったらいいと思う機能を追加する。

なお、課題のいくつかのためには画面の幅や高さを知りたいですね? その場合「`initscr()`」の後で「`getmaxyx(stdscr, height, width);`」を呼びます。ここで`height`や`width`は別の名前でもよく、とにかく整数変数です。[1]

[1]なぜ「`&`」をつけなくても値が変化させられるか疑問に思うでしょうけれど、これはマクロ機能を使っているためです…が、マクロについては後の回で余裕があったら取り上げますのでそれまで保留で。

本日の課題 8A

「演習1」〜「演習6」で動かしたプログラム1つを含むレポートを本日中(授業日の23:59まで)に提出してください。

1. solまたはCED環境で「`/home3/staff/ka002689/prog19upload 8a` ファイル名」で以下の内容を提出。
2. 学籍番号、氏名、ペアの学籍番号(または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
3. プログラムどれか1つのソースと「簡単な」説明。
4. レビュー課題。提出プログラムに対する他人(ペア以外)からの簡単な(ただしプログラムの内容に関する)コメント。
5. 以下のアンケートの回答。

Q1. 双連結リストの概念を理解しましたか。

Q2. 行エディタのプログラムを理解し直せるようになりましたか。

Q3. リフレクション(今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 8B

「演習 1」〜「演習 6」(ただし 8A で提出したものは除く) の (小) 課題から選択して 2 つ以上プログラムを作り、レポートを提出しなさい。できるだけ複数の演習から選ぶこと。レポートは次回授業前日 **23:69** を期限として提出すること。

1. sol または CED 環境で「`/home3/staff/ka002689/prog19upload 8b ファイル名`」で以下の内容を提出。
2. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
3. 1 つ目の課題の再掲 (どの課題をやったか分かればよい)、プログラムのソースと「丁寧な」説明、および考察 (課題をやってみて分かったこと、分析、疑問点など)。
4. 2 つ目の課題についても同様。
5. 以下のアンケートの回答。

Q1. 双連結リストの操作が自由にできるようになりましたか。

Q2. 画面エディタについてどのように理解しましたか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。