プログラミング通論'19 ＃7 – 単連結リスト

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。

- 動的データ構造の考え方を知る。
- 単連結リストを様々に操作できるようになる。

1　動的データ構造と単連結リスト

1.1　動的データ構造と領域の管理

データ構造 (data structure) とは、プログラム (やアルゴリズム) が扱うデータの「かたち」を指す言葉です。これまでのコードで配列、構造体、構造体の配列などを扱い、またそれらを組み合わせてスタック、キューなどを作って来ましたが、これらはすべてデータ構造の例です。

動的データ構造 (dynamic data structure) とは、ポインタを使って構造どうしを結び合わせることにより作り出されているデータ構造のことを言います。動的データ構造の特徴は、データ構造の「かたち」が大きく変化し得ることです。[1]

動的データ構造を作り出したり操作するときには、`malloc` による動的なメモリ割り当てが必要になります。C 言語では本来、使わなくなった領域を `free` により解放する必要がありますが、どの領域を使わなくなったか正確に把握することは繁雑なので、ここでは省略しています。

ここで示す例題程度のプログラムでは、解放を省略してもメモリが不足する心配はありません。そして今日の C 以外のプログラミング言語の多くでは最初から、使わなくなった領域を自動回収する**ごみ集め** (garbage collection、GC) 機能が搭載されています。また C 言語でも、長時間動き続けるプログラムの場合、「保守的 GC(conservative GC)」と呼ばれるライブラリを使うことで、ごみ集め機能を利用できます。[2]

1.2　単連結リスト

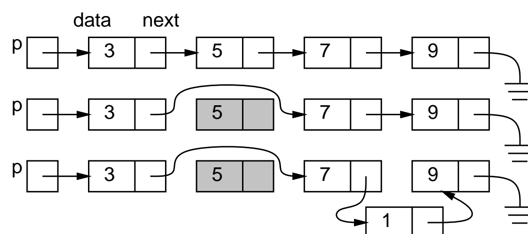


図 1: 単連結リストの例

動的データ構造の例として、**単連結リスト** (single linked list、単リストとも呼ぶ) を取り上げましょう。単リストは図 1 のように、複数の**節** (node) ないし**セル** (cell) がポインタによって直線的に結ばれた構造です (C 言語では通常、節には動的メモリ割り当てした構造体を使います)。

[1] これと対比して、これまで扱ったような、プログラムの冒頭で構造を作り出した後、その形が基本的に同じままのものを静的データ構造 (static data structure) と呼ぶこともあります。

[2] 必要になったら「conservative GC」で検索して探してください。

そして図の上から中のように、また中から下のように、ポインタを付け替えることでデータの並びを変化させられます (その過程で灰色の節のように使わなくなった節が生じることもあります)。配列などではデータの並びを変化させるときにはデータを移動させる必要がありましたが、単リストではポインタの付け替えだけで自由に順番が入れ換えられます。これが動的データ構造の利点です。なお、最後のアース印はポインタ値として NULL(終わりを表す特別な値で stdlib.h で定義、多くのシステムでは 0) が入っていることを表します。

それでは最初の例題として、コマンド引数で渡した文字列をそれぞれ実数値に変換してリストにし、続いて 2 番目を削除し、(元の)3 番目の次に「1.0」を挿入するというものを見てみましょう。まずノードの構造体を定義し、データと次の要素へのポインタ値 (または NULL) を渡して新しいセルを作りそのポインタを返す関数 node_new を定義します。

plist は各セルのデータを順に出力して最後に改行します。それには、まず p のデータを出力し、「p = p->next;」で p を次のセルに進め、ということを p が NULL でない限り繰り返せばよいわけです。

```
// slistdemo1.c --- single-linked list demonstration
#include <stdio.h>
#include <stdlib.h>
struct node { double data; struct node *next; };
typedef struct node *nodep;

nodep node_new(double d, nodep n) {
  nodep p = (nodep)malloc(sizeof(struct node));
  p->data = d; p->next = n; return p;
}
void plist(nodep p) {
  while(p != NULL) { printf(" %g", p->data); p = p->next; }
  printf("\n");
}
nodep mklist(int n, char *a[]) {
  nodep p = NULL;
  for(int i = n-1; i >= 0; --i) { p = node_new(atof(a[i]), p); }
  return p;
}
int main(int argc, char *argv[]) {
  nodep p = mklist(argc-1, argv+1); plist(p);
  p->next->next = p->next->next->next; plist(p);
  p->next->next = node_new(1.0, p->next->next); plist(p);
  return 0;
}
```

それではコマンド引数のような文字列の配列 (ただしデータは先頭から全部入っているものとします) をリストに変換する mklist を見てみましょう。リストは後ろから順番に作るのが作りやすいです。最初 p に NULL を入れ、そして「p = node_new(データ, p);」でこれまでのリストの先頭に新しいデータのセルをつけ加えることを繰り返します。

最後に main です。argv の 1 番目以降を渡して (データの個数は argc-1) リストを作り、次に 2 番目のセルをバイパスします。そして次に、(現在の)2 番目のセルの次として新たなセルを指させます。その新たなセルの後ろは (現在の)3 番目のセルにしておけばよいわけです。最初の状態、削除後、そして挿入後の状態を plist で表示します。実行例を見てみましょう。

```
% ./a.out 3 5 7 9
```

```
3 5 7 9
3 5 9
3 5 1 9
```

ときに、上の mklist と plist はループ版でしたが、再帰版も例示しておきます。

```
void plist(nodep p) {
  if(p == NULL) { printf("\n"); return; }
  printf(" %g", p->data); plist(p->next);
}
nodep mklist(int n, char *a[]) {
  if(n == 0) { return NULL; }
  return node_new(atof(*a), mklist(n-1, a+1));
}
```

plist は NULL なら改行するだけ。そうでなければ自分のデータを出力したあと「残りを打ち出して改行」するために自分自身を呼びます。mklist は「残っている個数が 0 なら NULL、そうでないなら自分を除いた残りのリストの前に自分のデータを持つセルをくっつけてそれを返す」わけです。

演習 1 上の例題をそのまま動かし、動作を確認しなさい。うまく動いたら、次のように変更してみよ。

a. 最後の要素を先頭につなぎ直す (その 1 つ前が最後になる)。
b. 先頭の要素を最後につなぎ直す (元の 2 番目が先頭になる)。
c. 2 番目と 3 番目のセルを入れ換える (データだけ入れ換えるのでなく、あくまでもつなぎ替えてセルの並びを入れ換えること)。
d. リストが何であれ、同じもの 2 つずつのリストに変更する (「1 2 3」→「1 1 2 2 3 3」のように)。
e. その他、好きなリストのつなぎ替えをおこなう。

1.3 例題: 並びエディタ

前節の例題ではデータは実行時に指定していましたが、操作はプログラムに書き込んであって固定でした。もっとその場で色々操作できた方が楽しそうなので、「並びエディタ」を作ってみます。これは、コマンドと (必要ならパラメタ) を 1 行打ち込むごとに指示された動作をします。当面次のコマンドがあります。

- e 値 値 ··· — (enter) 並びを設定する (これを「現在の並び」と呼ぶ)。
- a 値 値 ··· — (append) 指定した並びを現在の並びの後ろに追加。
- add 値 値 ··· — (add) 指定した並びの値を現在の並びに各々加算。
- d 番号 — (delete) 指定した番号の値を削除。
- p — (print) 現在の並びを表示する。
- q — (quit) プログラムを終了する。

実行のようすを示します。

```
% ./a.out
> e 1 2 3     ←現在の並び
> a 4 5 6     ←後ろに追加
> p           ←表示
```

```
1 2 3 4 5 6
> add 1 1 1 1  ←先頭の4つについて1を足す
> p
2 3 4 5 5 6
> d 2          ←2番を削除(先頭は0番)
> p
2 3 5 5 6
> q
%
```

ではコードを見てみましょう。だいぶ長くなったので、小分けにして見ていきます。まず先頭から plist まではこれまでと代わりません (include は増えていますが)。

```
// listeditor.c --- single-linked list editor
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
struct node { double data; struct node *next; };
typedef struct node *nodep;

nodep node_new(double d, nodep n) {
  nodep p = (nodep)malloc(sizeof(struct node));
  p->data = d; p->next = n; return p;
}
nodep mklist(int n, char *a[]) {
  if(n == 0) { return NULL; }
  return node_new(atof(*a), mklist(n-1, a+1));
}
void plist(nodep p) {
  if(p == NULL) { printf("\n"); return; }
  printf(" %g", p->data); plist(p->next);
}
```

次の nconc はリスト p の後ろに q をくっつけます。それには、p の最後のセルまで行き、その next フィールドを q にすればよいのです。

```
void nconc(nodep p, nodep q) {
  while(p != NULL && p->next != NULL) { p = p->next; }
  if(p != NULL) { p->next = q; }
}
```

削除は、数えながら削除するセルの1つ手前まで行きます。そして1つ手前のセルの次を削除する次のセルにすればよいわけです。

```
void delnode(nodep p, int n) {
  while(--n > 0 && p != NULL) { p = p->next; }
  if(p != NULL && p->next != NULL) { p->next = p->next->next; }
}
```

足し算ですが、2 つのリストの長い方に長さが揃うことにして、再帰を使っています。単純なケースですが、片方が NULL ならそれ以上足し算は必要でないので、他方が結果でよいわけです。残っているのは両方とも null でない場合なので、両方のリストの残りを再帰呼び出しにより足し算して、その前に自分の担当する 2 つのデータを足した値を持つセルを作ります。

```
nodep addlist(nodep x, nodep y) {
  if(x == NULL) { return y; }
  if(y == NULL) { return x; }
  return node_new(x->data+y->data, addlist(x->next, y->next));
}
```

ここから先は単リストと関係ありません。まず getl は指定された文字配列に 1 行ぶん読み込むもので、前にも使いました。getchar で 1 文字ずつ読みながら、読んだ文字が改行でも終わりの印でもなければそれをバッファに追加し、バッファの書き込み位置は 1 つ進めます (後置の++の働き)。最後に文字列の末尾にナル文字を入れますが、できた文字列の長さが 0 でファイル終わりなら false、それ以外は true を返します。

```
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;
}
```

次に parse は、1 行入力したものを空白で分けてそれぞれの文字列の先頭を別の配列 (後の並びの配列) に入れます。空白はナル文字に書き換えることで、それぞれの文字列の終わりがあるようにします。文字を順番に調べながら、空白をナル文字に書き換えます。次に現在位置がナル文字であり、かつ「それが先頭文字か 1 つ手前の文字がナル文字なら」単語の先頭だということでその現在位置のポインタを語の並びの配列に入れ、入れる位置は 1 つ進めます。最後に語の数を返します。

```
int parse(char *a[], char *s) {
  int i, k = 0, len = strlen(s);
  for(i = 0; i < len; ++i) {
    if(s[i] == ' ') { s[i] = '\0'; }
    if(s[i] != '\0' && (i == 0 || s[i-1] == '\0')) { a[k++] = s+i; }
  }
  return k;
}
```

それでは main です。まず 1 行入力のバッファと語のリストの領域があり、並びはノードのポインタです。無限ループに入り、先頭で 1 行入力しますが入力が読めなければループを抜け出ます。読めた場合は語に分割し、先頭の語が何であるかによって各コマンドの動作に分岐します。「q」はループを抜けるだけ。「e」はコマンドより後部分を mklist でリストにしてそれを list に入れます。「a」は同様ですが、list に入れるかわりにその末尾に nconc でくっつけます。「add」も同様ですが、元のリストと指定したリストを addlist で足し算し、結果を list に入れ直します。「d」は dellist を読んで指定した番号を削除します。そして「p」は…上記以外の何であっても plist で表示するようにしています。

```
int main(int argc, char *argv[]) {
```

```
  char buf[200], *cmd[20];
  nodep list = NULL;
  while(true) {
    printf("> "); if(!getl(buf, 200)) { break; }
    int k = parse(cmd, buf);
    if(k > 0 && strcmp(cmd[0], "q") == 0) {
      break;
    } else if(k > 0 && strcmp(cmd[0], "e") == 0) { // enter list
      list = mklist(k-1, cmd+1);
    } else if(k > 0 && strcmp(cmd[0], "a") == 0) { // append list
      nconc(list, mklist(k-1, cmd+1));
    } else if(k > 1 && strcmp(cmd[0], "add") == 0) { // add list
      list = addlist(list, mklist(k-1, cmd+1));
    } else if(k > 1 && strcmp(cmd[0], "d") == 0) { // delete item
      delnode(list, atoi(cmd[1]));
    } else {
      plist(list);
    }
  }
  return 0;
}
```

演習 2 上の例題をそのまま動かし、さまざまなリスト操作をやってみよ。納得したら、次のような機能追加をおこなえ (コマンド名や指定方法は好きに決めてよい)。

a. 足し算のかわりに掛け算をおこなう機能。

b. 並びを逆向きに変更する機能。

c. 並びを右巡回 (末尾の要素を先頭に移し残りを繰り下げる)。

d. 並びを左順解 (先頭の要素を末尾に移し残りを繰り上げる)。

e. 指定位置に要素を挿入する機能 (1 つだけでも並びでもよい)。

f. その他面白いと思う機能。

演習 3 上の並びエディタは「先頭の値は消せない」「先頭に要素を挿入できない」という欠陥がある。下の説明を読んで修正してみよ。

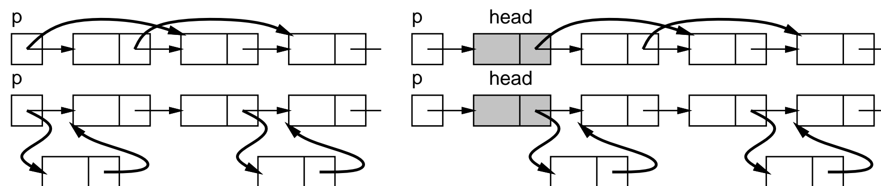


図 2: 普通の単リスト (左) と頭を持つ単リスト (右)

演習 3 のような問題が起きるのは、リストの先頭はセルではなく単独の変数で「形が違う」ところにあります。これを修正する 1 つの方法は、リストの先頭としてダミーのセルを常に置くことにして、その次からをデータとして扱うことです。このダミーのセルを「頭 (head)」と呼びます。演習 3 は、リストが常に頭を持つように変更することで解決可能です (別の方法もありますが)。図 2 の右側のように、頭を置くことで先頭への追加/削除が途中への追加/削除と同じ形の操作にできるわけです。

2　単連結リストを使ったスタックとキューの実装

2.1　単連結リスト版と配列版の比較

単連結リストはさまざまな用途に使えますが、その代表例としてスタックやキューの実装に使うというものがあります。以前にスタックやキューを扱った時は実装に配列を使っていましたが、配列版と対比して単連結リストを使う版には次のような得失があります。

- ○ データ量の上限を設けなくて済む。
- △ データ 1 個あたりの領域オーバヘッド (データ本体以外に必要とする余分な記憶領域) は多くなる。

前者ですが、配列は最初にサイズを決めて割り当てるので、そのサイズまで使ったら「満杯」になります。一方、単連結リストでは上限はなく、動的メモリ割り当てができる限りは要素を増やして行けます。ただし、配列を使う実装でも「満杯になったらより大きい配列を取り直して内容をコピーする」方法を使えば上限の問題を回避できます。

後者については、配列では格納するデータの値が配列要素として並んでいるので、余分に必要な場所はレコード領域など少量の、しかも決まった大きさの領域だけです。しかし単連結リストでは、データを 1 個格納するたびに「次の要素のポインタ」が必要なので、データ量に比例して余分に必要な場所が増えます。

2.2　単連結リストを使ったスタックの実装

スタックは片方の端だけでデータを出し入れするデータ構造であるので、単連結リストとの相性はとても良いです。図 3 のように、セル以外に必要な領域は top のポインタだけになります。

push するときは新しいセルを作ってこれまでの先頭を next で指した上で、top には新しいセルを指させます (top が NULL だった場合もコードは同じままで大丈夫です)。pop するときはその逆で、top が指しているセルのデータを返し、top はそのセルの next に書き換えます。

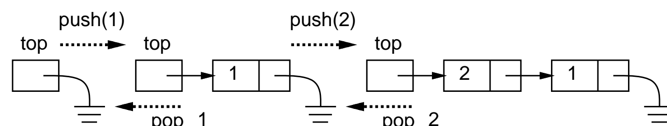


図 3: 単リストを使ったスタックの実装

以下に `istack.h` を再掲します。インタフェースはこれと同じままで、内部では図 3 の構造を用いるようにすればよいわけです。

```
// istack.h --- int type stack interface
#include <stdbool.h>
struct istack;
typedef struct istack *istackp;
istackp istack_new(int size); // allocate new stack
bool istack_isempty(istackp p); // test if the stack is empty
void istack_push(istackp p, int v); // push a value
int istack_pop(istackp p);      // pop a value and return it
int istack_top(istackp p);      // peek the topmost value
```

1 つだけ、作るときに size(上限サイズ) を渡していましたが、それはどうしましょうか? いくつか可能性があります。どれも一定の合理性があります。

- 単連結リスト版では不要なので、引数 size を削除する。
- インタフェースは変えたくないので、size は削除しないが単に使わないことにする。
- 現在いくつデータが入っているかも数えるようにして、size も保持して満杯かどうかの管理をする。

演習 4 連結リスト版のスタックの実装を作れ。これまでに作ったスタックを利用するプログラムと組み合わせて動かし動作を確認すること。size の扱いは好きに決めてよい。

2.3 単連結リストを使ったキューの実装

キューの場合は、データを入れる場所と取り出す場所が違っているので、top と last という 2 つのポインタが必要で、スタックより実装は複雑です (配列版でもそうでした)。

コードを簡潔にするには、先に説明した頭 (head) を持つリストを使うのがおすすめです。図 4 にそのような実装を示しました。空っぽのときは頭のセル (灰色にしてあります) だけがあり、top も last もそこを指しています。

データを追加するときは、last の指しているセルの next に新しいセルをくっつけてデータを入れ、last はその新しいセルに変更します。データを取り出すときは、top の指しているセル…は頭ですから、その next が指しているセルからデータを取り、next はそのまた next で置き換えます。

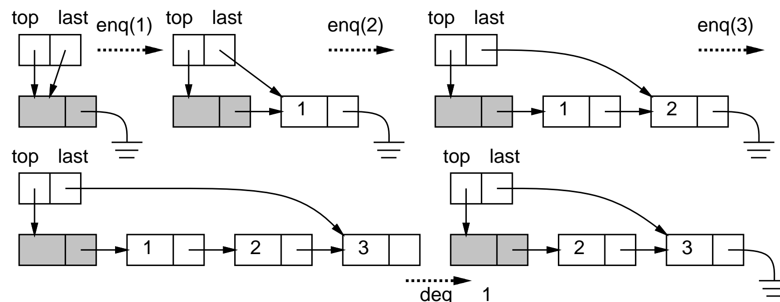


図 4: 単リストを使ったキューの実装 (頭つき)

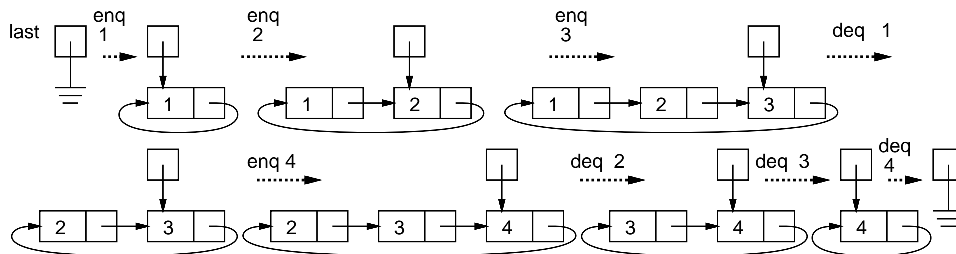
以下に iqueue.h を再掲します。これもサイズの扱いは先に述べたような選択肢があります。さらに、isfull があるのですが、これはデータ数を管理しない場合は「決して満杯にならない」ことにしてもよいですし、動的メモリ割り当てが失敗したときが満杯ということにしてもよいです。

```
// iqueue.h --- int type queue interface
#include <stdbool.h>
struct iqueue;
typedef struct iqueue *iqueuep;
iqueuep iqueue_new(int size);
bool iqueue_isempty(iqueuep p);
bool iqueue_isfull(iqueuep p);
void iqueue_enq(iqueuep p, int v);
int iqueue_deq(iqueuep p);
```

演習 5 連結リスト版のキューの実装 (頭つき版) を作れ。size の扱い、isfull の扱いは好きに決めてよい。単体テストを実施すること (#3 の単体テストが参考になる)。

演習 6 頭なしでもキューの実装は可能である (空のときを特別扱いする必要が生じるだけ)。そのような版の実装を作れ。size の扱い、isfull の扱いは好きに決めてよい。単体テストを実施すること (#4 の単体テストが参考になるが、満杯の扱いが違うと思われるのでそこは違うはず)。

演習 7 循環 (環状) リストとは下図のように、リストの最後の次が先頭を指すようにして循環した形のリストをいう。循環リストを使うことで、last ポインタだけでキューを実現できる (循環リストでは last の次は先頭なので先頭を別に覚えなくてよい)。ただし要素が 1 個もないときは別扱いとなる。循環リストを用いたキューの実装を作成せよ。size の扱い、isfull の扱いは好きに決めてよい。これも単体テストを実施すること。



本日の課題 7A

「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 7a 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. 単連結リストの概念を理解しましたか。

   Q2. 並びエディタのようなコマンドに応答するプログラムの作り方を納得しましたか。

   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。

次回までの課題 7B

「演習 1」～「演習 6」(ただし 7A で提出したものは除外、以後も同様) 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 を期限とします。

1. sol または CED 環境で「`/home3/staff/ka002689/prog19upload 7b 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. 単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. スタックやキューの単連結リスト実装について理解しましたか。

   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。

3 付録: デバッガ gdb

皆様はプログラムが動かないとき、どのようにデバッグしますか? 「コードをよく見る」「要所で値を printf 出力」が普通だと思いますが、別の方法として「デバッガ (debugger)」というツールがあります。デバッガでは、コマンドを使って「ここで止める」「値を表示する」「少しずつ動かす」ができるので、効率よくデバッグできる…「こともあり」ます。

ただし、少しずつ実行することは、とても時間を取る (遅いという意味ではなく人間がそれを見るので手間取る) ので、結果として効率が悪く、結局プログラムをよく見る方が良かった (それなら修正すべき箇所が直接わかる)、ということも起こります。なので万能ではありませんよ、という警告はした上で、**gdb** というデバッガを紹介します。

gdb は gcc と一緒のプロジェクトで開発されてきたデバッガであり、gcc が生成するデバッグ情報を使ってさまざまな表示をしてくれます。デバッグ情報を生成させるには、gcc に「`-g`」オプションを指定する必要があります。そのため、使うときは次のようにます。

```
% gcc8 -g その他の指定はこれまで同様
% gdb8 a.out     ← gcc が生成した実行形式ファイルを指定
... メッセージ多数 ...
(gdb)            ← gdb のプロンプト
```

gdb で最低限覚えて欲しいコマンドは次のものがあります (良く使うものなので大抵は 1 文字の省略形が使えます。「bt」のみ 2 文字ですが)。

- break (b) — 中断点を設定する。中断点を設定しないで動かすと、一気に全部動いてしまい途中の様子が調べられないので、通常はまず中断点を指定する。次の 3 つの書き方を覚えるとよい。
  - 「`b 関数名`」 — 指定した関数の入口で停止。
  - 「`b 行番号`」 — 現在のファイルの指定した行で停止。
  - 「`b ファイル名:行番号`」 — 指定ファイルの指定行で停止 (複数ファイルから成るプログラムで使用)。
- run (r) — プログラムを実行開始する。コマンド引数をここで指定する。たとえば「`./a.out 1 2`」のように動かすプログラムであれば、gdb の中では「`r 1 2`」で実行開始させる。とくにコマンド引数を指定しないプログラムなら「`r`」でよい。
- quit (q) — gdb を終了する。

r コマンドで実行開始したあと、中断点に到達すると実行が中断され、gdb プロンプトが表示されます (また、bus error などのエラー終了が起きた場合は中断点を設定していなくてもそこで中断が起きます)。中断した箇所で実行するコマンドに次のものがあります。

- backtreace (bt) — 「どの関数が何行目でどの関数を呼び…」という呼び出の連鎖を表示。
- list (l) — 中断点付近のソースコードを表示する。
- print (p) — 変数や式の値を表示する。「`p i`」のように単独変数の値を表示するほか、「`p a[i]`」「`p q->data`」のように配列、レコード、ポインタを扱うこともできる。
- continue (c) — 実行を再開する。次にまた中断点に到達するまで一気に実行される。
- next (n) — 1 行ぶん実行する。次の行に来たときにそこで止まるので、中断点を設定していなくても 1 行ずつ実行の流れを調べ、そこでの値を (p コマンドで) 調べることができる。なお、関数呼び出しがあった場合はその中は一気に実行して次の行に進む。
- step (s) — next と同様だが、関数呼び出しがあった時にはその関数内の次の行へ行く。

では実際に使ってみるとして、サンプルコードに「再帰版とループ版の階乗」を用意しました。

```
// factdemo.c --- two fact function for gdb exercise.
#include <stdio.h>
#include <stdlib.h>
int fact1(int n) {
  int i, r = 1;
  for(i = 1; i <= n; ++i) {
    r *= i;
  }
  return r;
}
int fact2(int n) {
  if(n <= 0) { return 1; }
  int r = fact2(n-1);
  return n * r;
}
int main(int argc, char *argv[]) {
  int n = atoi(argv[1]);
  printf("fact1(%d) == %d\n", n, fact1(n));
  printf("fact2(%d) == %d\n", n, fact2(n));
  return 0;
}
```

最初にループ版、次に再帰版の階乗計算の関数が呼ばれるので、それぞれを中断点に指定して動かすことにします。中断したあと、ループ版ではステップ実行しつつ、時々変数の値を表示して検討します。再帰版では再帰呼び出しのたびに中断点に来るので、「c」を使って継続していけばよいですが、最後の再帰呼び出しのあと戻ってきたときの計算を調べるにはステップ実行を使っています (または行番号指定で中断点を設定しておく方がよかったかも知れません)。

```
% gcc8 -g factdemo.c
% gdb a.out
Copyright (C) 2018 ...
Reading symbols from a.out...done.
(gdb) break fact1    ←fact1 の入口で中断
Breakpoint 1 at 0x40056d: file factdemo.c, line 5.
(gdb) break fact2    ←fact2 の入口で中断
Breakpoint 2 at 0x4005a3: file factdemo.c, line 12.
(gdb) r 5
Starting program: /home3/staff/ka002689/a.out 5
Breakpoint 1, fact1 (n=5) at factdemo.c:5 ←中断設定
5         int i, r = 1;
(gdb) s                                  ←ステップ実行
6         for(i = 1; i <= n; ++i) {
(gdb) s
7           r *= i;
(gdb) s
6         for(i = 1; i <= n; ++i) {
```

```
(gdb) s
7           r *= i;
(gdb) s
6         for(i = 1; i <= n; ++i) {
(gdb) s
7           r *= i;
(gdb) p r                                ←r を表示
$1 = 2
(gdb) p i                                ←i を表示
$2 = 3
(gdb) c                                  ←一気に実行
Continuing.
fact1(5) == 120                ←プログラムによる表示
Breakpoint 2, fact2 (n=5) at factdemo.c:12 ←中断設定
12        if(n <= 0) { return 1; }
(gdb) c                                  ←再開
Continuing.
Breakpoint 2, fact2 (n=4) at factdemo.c:12 ←中断
12        if(n <= 0) { return 1; }
(gdb) c
Continuing.
Breakpoint 2, fact2 (n=3) at factdemo.c:12
12        if(n <= 0) { return 1; }
(gdb) c
Continuing.
Breakpoint 2, fact2 (n=2) at factdemo.c:12
12        if(n <= 0) { return 1; }
(gdb) c
Continuing.
Breakpoint 2, fact2 (n=1) at factdemo.c:12
12        if(n <= 0) { return 1; }
(gdb) c
Continuing.
Breakpoint 2, fact2 (n=0) at factdemo.c:12
12        if(n <= 0) { return 1; }
(gdb) bt                    ←呼び出し系列を表示
#0  fact2 (n=0) at factdemo.c:12
#1  0x00000000004005bd in fact2 (n=1) at factdemo.c:13
#2  0x00000000004005bd in fact2 (n=2) at factdemo.c:13
#3  0x00000000004005bd in fact2 (n=3) at factdemo.c:13
#4  0x00000000004005bd in fact2 (n=4) at factdemo.c:13
#5  0x00000000004005bd in fact2 (n=5) at factdemo.c:13
#6  0x0000000000400618 in main (argc=3, argv=0x7fffffffd7f8) at factdemo.c:19
(gdb) s                          ←ステップ実行
15      }
(gdb) s
14        return n * r;
```

```
(gdb) p r                                    ←r を表示
$3 = 1
(gdb) p n                                    ←n を表示
$4 = 1
(gdb) s
15      }
(gdb) s
14        return n * r;
(gdb) s
15      }
(gdb) s
14        return n * r;
(gdb) p r                                    ←r を表示
$5 = 2
(gdb) p n                                    ←n を表示
$6 = 3
(gdb) c                                      ←再開
Continuing.
fact2(5) == 120          ←プログラムからの出力
[Inferior 1 (process 44205) exited normally]
(gdb) q                                      ←終了
```

このように、デバッガを使うと「途中でどんな値になってるのか不明」「どの経路を通っているのか不明」なときに、そのことが調べられるという点では便利です。ただ、ダメなプログラムを長時間掛けて調べるよりは、分かりにくいと思ったらきれいに書き直す方が早道でお得ということもいえるので、うまく考えて使ってください。