コンピュータとプログラミング C言語編

5. メモリ管理

阿部洋丈 / ABE Hirotake
habe@cs.tsukuba.ac.jp

静的メモリと動的メモリ

- 静的メモリ
  - プログラム中の宣言によって確保されるメモリ（注： static 修飾子とは別の概念）
  - プログラム**実行前**に必要なメモリサイズが分かっていれば、静的メモリだけでも特に問題ない
- 動的メモリ
  - 必要なメモリ量を事前に見積もることが難しい場合に、必要に応じてプログラム**実行後**に確保されるメモリ
  - 外部（ファイル、ネットワーク）から大量の入力を受け付けるような場合が典型的

Python における動的メモリ管理はどうなってる？

- …と聞かれても皆さんの大半は答えられないはず
- なぜなら意識しなくて良かったから

```
fa = open('a.txt')
lines = fa.readlines()
fb = open('b.txt')
lines = fb.readlines()
```

- a.txt や b.txt の中身はどこのメモリに格納された？
- lines に b.txt の中身を格納した後、a.txt の中身はどうなった？
- etc…

Garbage Collection (GC)

- 動的に（＝プログラム実行開始後に）確保されたメモリ領域について、その必要性を判断し、不要になった領域を回収して再利用可能にする機能。
- Python, Ruby, Java, Swift 等には言語設計レベルで備わっているが、C や C++ には無い。
- なのでCプログラマは自分でやるしかない
  - 高級アセンブラと揶揄される最大の理由の一つ
  - ただし、後付けの GC なら存在する。興味のある人は Boehm GC について調べると良い

GC が無いことのメリット

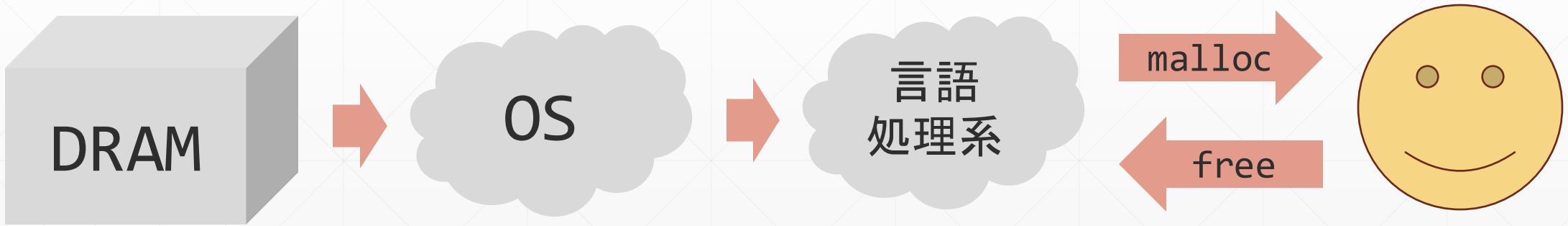
- GC 処理自体のためにCPUやメモリが消費されない
- GC の処理が動くことによるプログラム本体の実行中断が起きない
  - エンジンやモーターなどの制御プログラムでは重要
- メモリ割り当てを常に意識する必要があるため、メモリを無駄に使わないプログラミングを実現しやすい
  - 組み込み系ではメモリ容量が非常に限られていることがある

なぜメモリ管理が必要か

- **メモリリーク** (memory leak) を防ぐため
  - 不必要なメモリ領域が時間と共に増えていく現象
  - 最終的に、メモリが足りなくなり、プログラムが停止
  - そこまで行かなくても、プログラムの性能低下が生じる恐れ
  - ずっと動き続けるプログラム（OS, デバイスドライバ、サーバ、etc.）では特に問題
    - 再起動すれば一時的に改善されるが、再起動中はプログラムが応答できなくなる（ソフトウェア若化; rejuvenation）

C におけるメモリ管理の基本

- 必要になった時点で **malloc** 関数を使ってメモリ領域を取得
- 不要になった時点で **free** 関数を使って解放



malloc 関数 (memory allocation)

```
void *malloc(size_t size);
```

- size で指定された大きさのメモリ領域を確保する。成功したらその領域の先頭へのポインタを返す。失敗したら NULL を返す。
- size_t は、変数のサイズを表すための型。（その実体は unsigned long int）

sizeof 演算子

注： 結果は実行環境によって違うので注意

- 型名や変数名を与えると、それを格納するのに必要なサイズを size_t 型で返す演算子。

```
printf("char: %d\n",        sizeof (char));          // 1
printf("short: %d\n",       sizeof (short));         // 2
printf("int: %d\n",         sizeof (int));           // 4
printf("long: %d\n",        sizeof (long));          // 8
printf("float: %d\n",       sizeof (float));         // 4
printf("double: %d\n",      sizeof (double));        // 8
printf("long double: %d\n", sizeof (long double));   // 16

printf("char *: %d\n",      sizeof (char *));        // 8
printf("char[10]: %d\n",    sizeof (char[10]));      // 10
printf("10 * (char): %d\n", 10 * sizeof (char));     // 10
```

malloc によるメモリ確保

```
#define MAX 10
char *s;
s = malloc(sizeof (char) * MAX);   // キャストしなくても良い
if (s == NULL) {
  fprintf(stderr, "malloc failed\n");   // エラー内容を表示
  exit(EXIT_FAILURE);
}
```

- NULL が返されるということは、既にメモリ不足に陥っているということなので、大抵は諦めて exit するしかない
  - このチェックは常に必要。assert だと #define NODEBUG で無効化されるので適さない

malloc のバリエーション

- calloc: 確保したメモリを初期化
  - 初期化のために追加の処理時間を要する
- realloc: 既に確保してある領域を拡張もしくは縮小
  - その場での拡張・縮小ができない場合は、指定のサイズの新しい領域が別途確保され、そこに古い領域の内容がコピーされ、古いメモリは開放される。
  - メモリの確保に失敗した場合は古い領域はそのまま残るので注意が必要 (ptr = realloc(ptr, size) とすると古いアドレスが消える)
- 追加のコストを理解した上で、必要に応じて使うべき

free 関数

```
void free(void *ptr);
```

- malloc で確保された領域を開放し、再び malloc で割り当てられるように準備する
- 開放が成功したかどうかは分からないが、気にしても仕方ないので気にしない
- 大事なこと：

free した後はそのメモリを絶対に使わない！
free を2回呼ぶのは更に絶対ダメ！！

なぜダメ？

- free した後のメモリは再利用される
  - そこを書き換えたり勝手に free することは、配列の境界を超えて書き換えるよりも更に厄介なバグを生む
  - 同じ理由で、まだ使っているメモリを free するのもダメ
- 二重に free を呼ぶとセキュリティ上の脆弱性を生み出す結果になることがある
  - double-free 脆弱性

free の使い方

```
#define MAX 10

char *s;
s = malloc(sizeof (char) * MAX);
if (s == NULL) { fprintf(stderr, "malloc failed\n"); exit(-1); }

(... s を使った処理 ...)

free(s);     // s を開放
s = NULL;    // s の中身を消去

free(s);     // 何も起きない
```

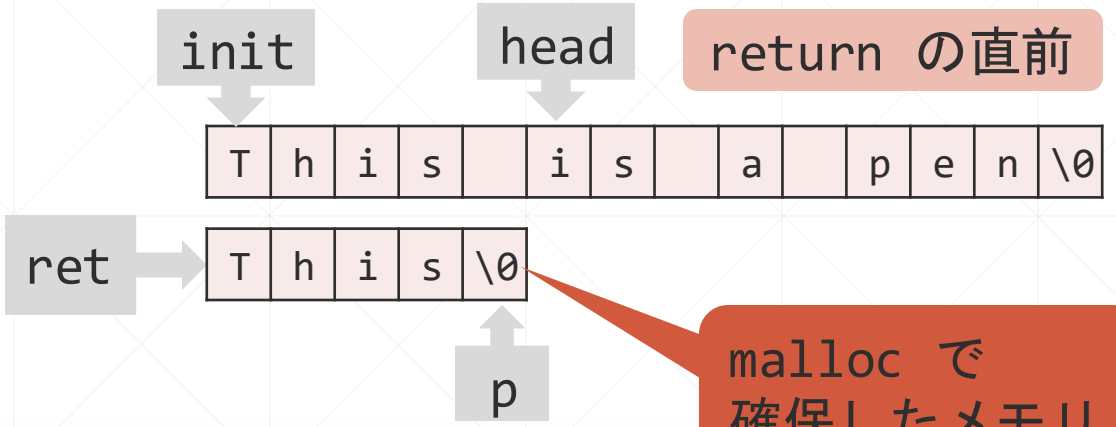
free は、そのポインタの先の割り当てサイズを知っているので、サイズを伝える必要は無い

free した後に誤って使ってしまう可能性を減らす
（バグよりは SEGV の方がマシ）

例: malloc 版の tokenizer

```
char *tokenizer(char *init) {
  char *p, *ret;
  static char *head = NULL;
  size_t len = 0;

  if (head == NULL && init == NULL) return NULL;
  if (init != NULL) head = init;
  if (*head == '\0') return NULL;

  p = head;
  while (*p != '\0' && *p != ' ') { p++; len++;}

  ret = malloc((len + 1) * sizeof (char));
  if (ret == NULL) exit(EXIT_FAILURE);

  p = ret;
  do {
    *(p++) = *(head++);
  } while (p - ret < len);
  *p = '\0';
  head++;

  return ret;
}
```

元の配列が書換不可でも動く。
その代わり、呼び出し側でメモリを free する必要が生じる。



単語の長さを覚えておく
(int 型でも動く)

以下を簡略化した書き方：
*p = *head; p++; head++;
(実行順序に注意が必要なので、
慣れていない場合は勧めない)

いつ free すべき？ そもそも free って必要？

- メモリが不要になるタイミングが明らかなら簡単だが、必ずしもそういうケースばかりではない
- ポインタへの代入や、ループの終わりや関数の終わりで、不要になったメモリがないかどうかよく考えるようにすることが大事
  - この問題に興味のある人は Rust 言語の region を調べると良い
- プログラムの終了直前（≒ main 関数の終わり）なら、無理に開放しなくても良い
  - プログラムが終了すればそのプログラム全体がOSに回収される
  - ※ 課題等で「不要なメモリは開放すること」と明示された場合は除く

malloc/free に関してさらに注意

- malloc 等で動的に確保されたアドレス以外を free してはいけない。
- malloc で確保された領域の前後を書き換えてはいけない。

いずれも、意図しないメモリ書換が起こり、
発見困難なバグを引き起こす原因になる。

今日の演習

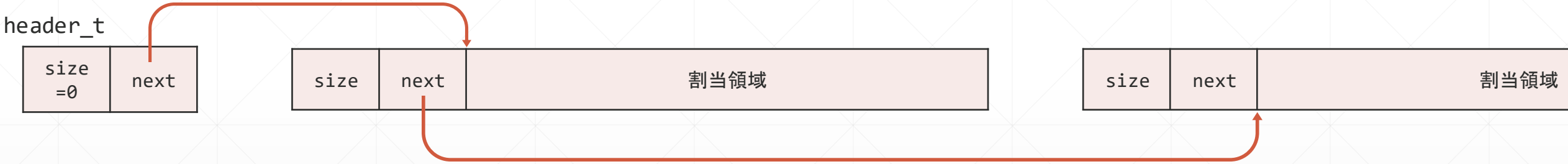
- 以下のいずれか１つを選択して行うこと
  - 課題A： どんなに長い文字列が来ても（メモリが足りる限りは）文字列をメモリ上に読み込むことができる関数 readline を完成させよ
  - 課題B（難易度：高）： 原始的なバージョンの malloc (minimalloc, minifree) を完成させよ

課題Aの解説

- 以下の方針で実装すると良い
  1. mallocで適当なサイズのメモリ領域を確保
  2. fgets 関数を使って、確保した領域に文字を読み込む
  3. すべての文字を読み込めたかどうかを検査する。行の末尾まで読み込めたならば、文字列に '\n' が含まれているはず（ファイルの終わりに到達した場合はその限りではないので注意）
  4. すべて読み込めていたら領域へのポインタを返す。すべて読み込めていなかったら、realloc で領域を拡張し、2へ戻る。
- 動的に確保するメモリ領域のサイズは、読み込むべき行のサイズとぴったり一緒でなくても良い。

課題Bの説明

- minimalloc は、init_minimalloc 関数に渡された char 配列を使ってメモリの割当要求に応える
- minimalloc は以下のように割当済み領域を管理する



- mimimalloc は、未割当の領域を前方から順に検査し、最初に見つかった割当可能領域を使って要求に応える
  - この方法は、賢いやり方であるとは全く言えない。本物の malloc はもっと賢いやり方をしている