C プログラミング

— スタック・キュー —

早稲田大学

本日の目標

- 動的なメモリの確保
- malloc 関数
- スタック
- キュー

動的なメモリの確保

これまでのプログラムでは

```
double x[3];
```

のように配列の大きさを宣言してから利用していた．しかしこれは

- 配列の要素数（ここでは 「3」）に依存してしまう
- 配列の要素数が変わるたびにプログラムの修正，コンパイル，実行を行わなければいけない

配列の要素数の内容に依存しないプログラムに変更してみよう．

- 配列の要素数の内容変更への対応
  動的なメモリ確保を行う関数 malloc( ) を利用する

  ⇒ プログラムの実行中に行列やベクトルの大きさに応じたメモリ領域を確保する．

動的なメモリ確保

関数 malloc(size)

- 引数 size で指定されたサイズのメモリ領域を確保して，その先頭へのポインタを返す関数
- メモリの確保に失敗した場合は，NULL を返す
- malloc( ) により確保したメモリ領域は，使い終わったら関数 free( ) により開放する必要がある
- malloc( ) を利用するにはヘッダファイル stdlib.h をインクルードする

関数 malloc() の基本的な使い方

```
double *x;
x = (double *)malloc(N*sizeof(double));
```

動的なメモリ確保

関数 malloc() の基本的な使い方

```
double *x;
x = (double *)malloc(N*sizeof(double));
```

- double 型のメモリサイズ N 個分の領域を確保し，その領域の先頭ポインタを x に代入

- sizeof 演算子…いろいろな型のメモリサイズ（バイト数）を求めることができる
  【例】sizeof(double)：8bytes，sizeof(int)：4bytes，sizeof(char)：1byte
- キャスト演算子…式で与えられているデータ型を型名へ変換する
  【例】`int a=1, b=2;` a/b ⇒ 0, (double)a/b ⇒ 0.5

動的なメモリ確保

関数 malloc() の基本的な使い方

```
double *x;
x = (double *)malloc(N*sizeof(double));
```

- メモリの確保に失敗した場合は，以降の処理を中止する
- 確保した領域は，使い終わった段階で開放する

Example (関数 malloc() の基本的な使い方)

```
x = (double *)malloc(N*sizeof(double));
if(x==NULL){
    printf("Can't allocate memory.\n");
    exit(1);
}

/* 必要な処理*/
free(x);
```

動的なメモリ確保

関数 malloc() の基本的な使い方

```
double *x;
x = (double *)malloc(N*sizeof(double));
```

⇒　ポインタ x が double 型の要素を N 個もつような配列と同じように扱うことができる.

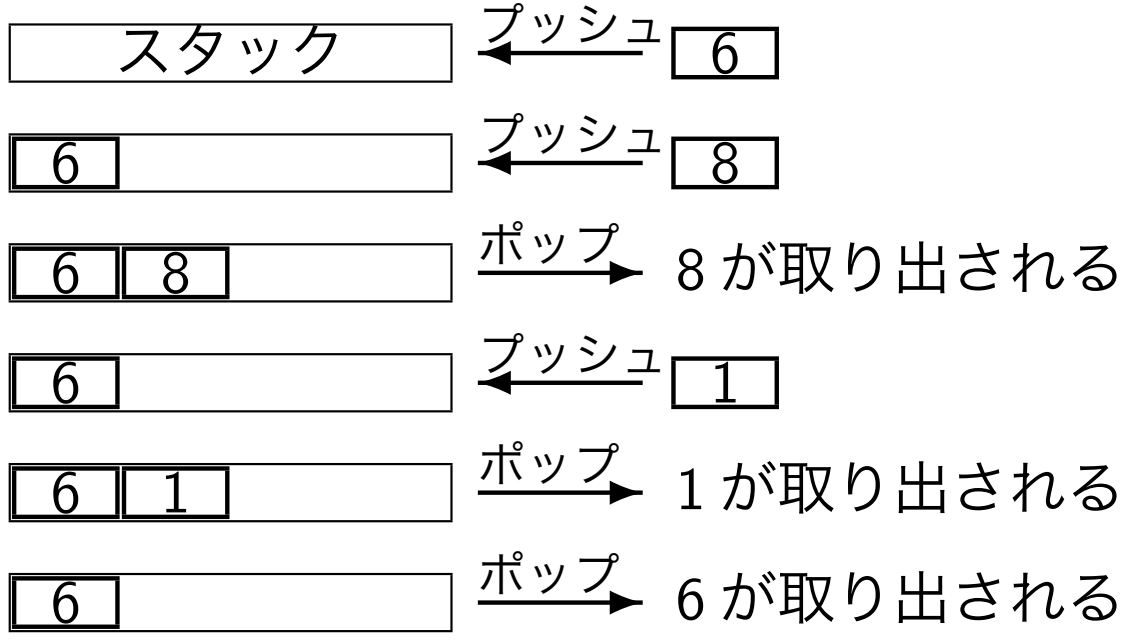
- x+i：i 番目の要素へのポインタ
- *(x+i)：i 番目の要素（$x[i]$ と同意）

malloc 関数使用例

```
#include<stdio.h>
#include<stdlib.h>

int main(void){
  int N;
  int *x;

  printf("Input N:");
  scanf("%d",&N);

  x = (int *) malloc(sizeof(int)*N);
  if(x==NULL){
        printf("Can't allocate memory.\n");
        exit(1);
  }

  for(i=0;i<N;i++) x[i]=i; // サイズ N の double 型配列 x として使用できる.

  for(i=0;i<N;i++) printf("x[%d]=%d\n",i,x[i]);

  free(x);
  return 0;
}
```

スタック

- データを記憶しておくためのデータ構造の一つ
  後入れ先出し方式（Last In First Out (LIFO)）をとる
- スタックに対しては次の二つの操作しかできない
  - プッシュ: スタックに新たなデータを追加する操作
  - ポップ: スタックから最も新しいデータを取り出す操作
- 最新のデータしか取り出すことはできないので不便である反面，どのデータを取り出すかを指定する必要がないので便利



スタック構造体

スタックの実現に必要なものは何だろうか？　ここでは配列を基にスタックを作ることにする.

- データを記憶しておくための配列と，配列のサイズ（= 最大格納数）
- 現在のデータ格納数

これらを構造体にまとめることにする．データは double 型とする.

- 構造体の型を表すには「struct 構造体タグ」のように，struct を必ず付ける必要があった．typedef を使うとこれを省略した書き方が可能になる

```
struct Stack{
  double *Data;   /*データ格納配列*/
  int Size;       /*最大格納数*/
  int Count;      /*現在の格納数*/
};
```

```
typedef struct{   /*構造体タグ省略*/
  double *Data;   /*データ格納配列*/
  int Size;       /*最大格納数*/
  int Count;      /*現在の格納数*/
} Stack;          /*構造体名を定義*/
```

- struct が省略されると構造体名であることが分かり難くなる半面，プログラムは簡潔になる．今後省略するスタイルで書く

スタックの作成

スタック自体を作成するには，次の操作が必要となる．

- スタック構造体を動的メモリ確保する
- データ格納配列を動的メモリ確保する
- 配列のサイズを設定する
- 現在の格納数を 0 にする

配列サイズは指定されるものとする．メモリ確保する順番を間違えてはならない．

- スタックを作成する関数は次のようになる．

```
Stack *CreateStack(int size) {
  Stack *s = (Stack*)malloc(sizeof(Stack));
  s->Data = (double*)malloc(sizeof(double) * size);
  s->Size = size;
  s->Count = 0;
  return s;
}
```

注: ここでは malloc のエラーチェックを省略しているが，次のスライドにある mallocx を使ってエラーチェックをすること．

エラーチェック付 malloc

```
void *mallocx(int size) {  /* エラーチェック付 malloc */
  void *p = malloc(size);
  if (p == NULL) {
    printf("cannot allocate memory\n");
    exit(1);
  }
  return p;
}
```

以後，malloc 関数の代わりに mallocx 関数を使用して，メモリの動的確保を行う.

スタックの廃棄

スタック自体を廃棄するには，次の操作が必要となる．

- データ格納配列を解放する
- スタック構造体を解放する

メモリを解放する順番を間違えてはならない．

- スタックを廃棄する関数は次のようになる

```
void DisposeStack(Stack *s) {
  free(s->Data);
  free(s);
}
```

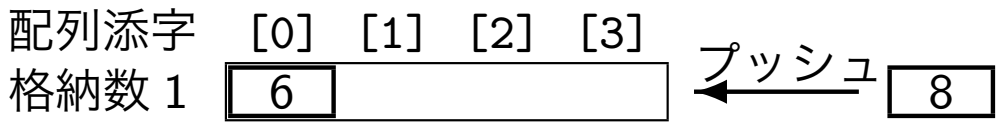
スタックへのデータ追加

プッシュ

指定されたデータをスタックに追加する操作のこと.

- ここでは，配列の空きの先頭に置くことにする．格納数は 1 増加する

```
void PushStack(Stack *s, double x) {
  s->Data[s->Count] = x;
  s->Count++;
}
```



- スタックが満杯の場合はデータを追加することができない．これをスタックオーバーフローと呼ぶ．このような場合はエラーを表示して強制終了する

```
  if (s->Count == s->Size) {
    printf("stack overflow\n");
    exit(1);
  }
```

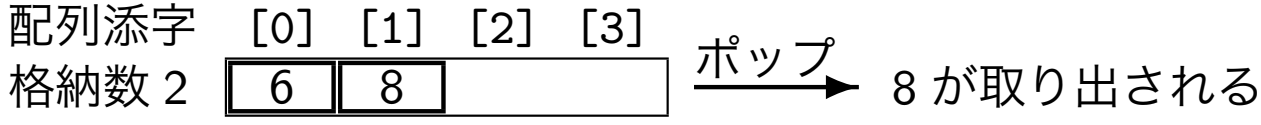
スタックからのデータ取出

ポップ

スタックからデータを取り出す操作のこと.

- 格納されているデータの中で，最も新しいデータを取り出す．格納数は 1 減少する

```
double PopStack(Stack *s) {
  s->Count--;
  return s->Data[s->Count];
}
```



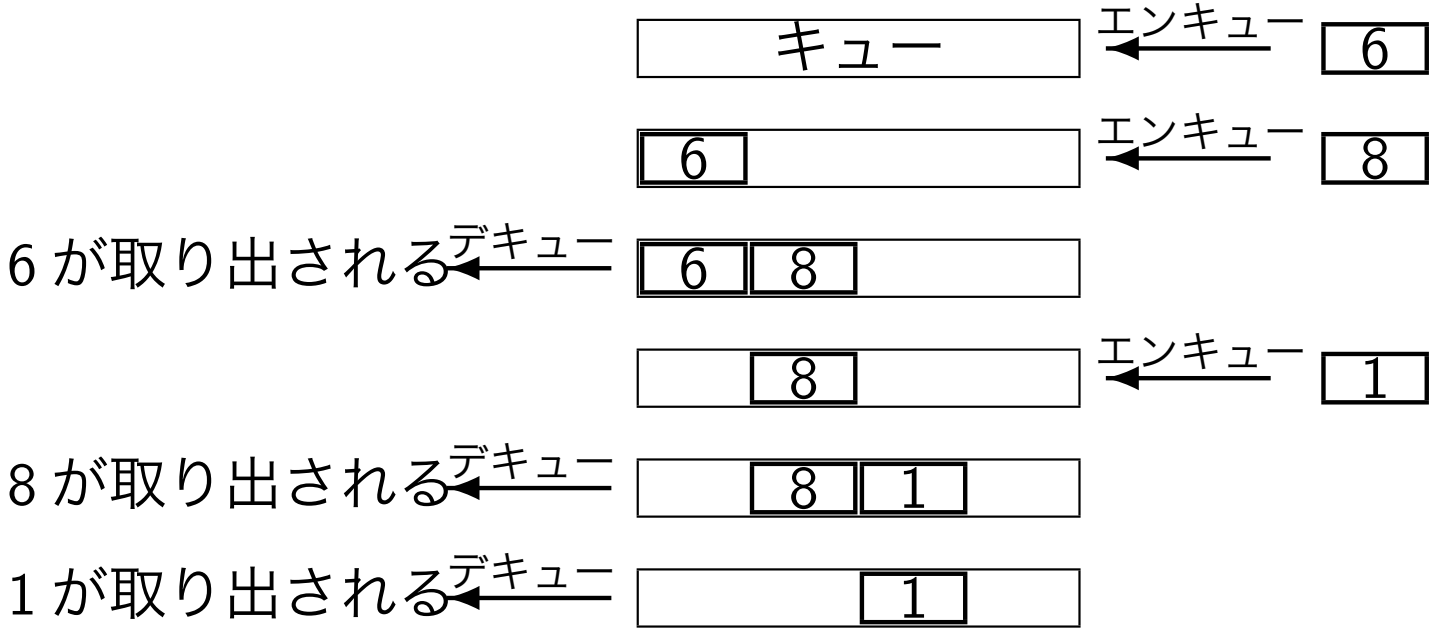
- スタックが空の場合はデータを取り出すことができない．これをスタックアンダーフローと呼ぶ．このような場合はエラーを表示して強制終了する

```
if (s->Count == 0) {
  printf("stack underflow\n");
  exit(1);
}
```

- 何故ポップした時にデータを消去しなくても良いのだろうか？

キュー

- データを記憶しておくためのデータ構造の一つ.
  先入れ先出し方式（First In First Out (FIFO)）をとる
- キューに対しては次の二つの操作しかできない
  - エンキュー: キューに新たなデータを追加する操作.
  - デキュー: キューから最も古いデータを取り出す操作.
- 最古のデータしか取り出すことはできないので不便である反面, どのデータを取り出すかを指定する必要がないので便利



キュー構造体

キューの実現に必要なものは何だろうか？　ここでは配列を基にキューを作ることにする.

- データを記憶しておくための配列と，配列のサイズ（= 最大格納数）
- 現在のデータ格納数
- 最古データの格納位置

これらを構造体にまとめることにする．データは double 型とする.

- 構造体は次のようになる

```
typedef struct{
  double *Data;  /* データ格納配列 */
  int Size;      /* 最大格納数 */
  int Count;     /* 現在の格納数 */
  int Index;     /* 最古データの格納位置 */
} Queue;
```

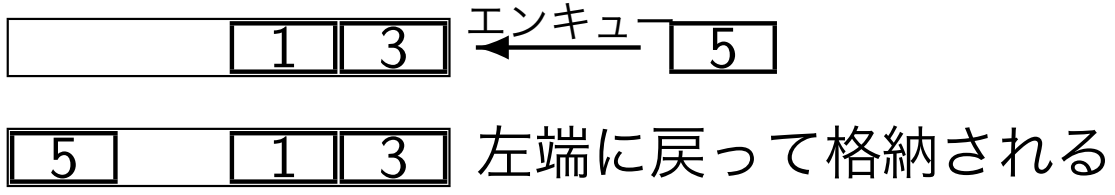
リングバッファ

問題点

配列を用いてデータを出し入れしていくと，右端まで格納したところで新たに格納するスペースが右側にはなくなる.

リングバッファ

配列の末尾（右端）と先頭（左端）が繋がっているものとして扱うバッファのこと.



キューの作成

キュー自体を作成するには，次の操作が必要となる．

- キュー構造体を動的メモリ確保する
- データ格納配列を動的メモリ確保する
- 配列のサイズを設定する
- 現在の格納数を 0 にする
- 最古データの格納位置を 0 にする（配列内のどこかを指しておく）

配列サイズは指定するとし，メモリ確保する順番を間違えてはならない．

- キューを作成する関数は次のようになる

```
Queue *CreateQueue(int size) {
  Queue *q = (Queue*)mallocx(sizeof(Queue));
  q->Data = (double*)mallocx(sizeof(double) * size);
  q->Size = size;
  q->Count = 0;
  q->Index = 0;
  return q;
}
```

キューの廃棄

キュー自体を廃棄するには，次の操作が必要となる．

- データ格納配列を解放する
- キュー構造体を解放する

メモリを解放する順番を間違えてはならない．

- キューを廃棄する関数は次のようになる

```
void DisposeQueue(Queue *q) {
  free(q->Data);
  free(q);
}
```

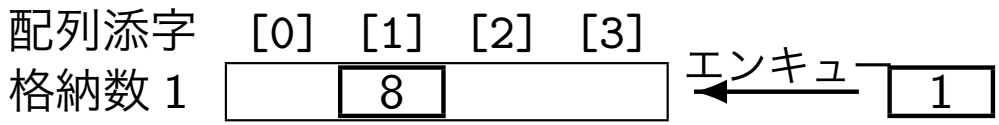
キューへのデータ追加

エンキュー

指定されたデータをキューに追加する操作のこと.

- 配列の空きの先頭に置くことにする. 格納数は 1 増加する

```
void Enqueue(Queue *q, double x) {
  q->Data[(q->Index + q->Count) % q->Size] = x;
  q->Count++;
}
```



- キューが満杯の場合はデータを追加することができない. これをキューオーバーフローと呼ぶ. このような場合はエラーを表示し, 強制終了する.

```
  if (q->Count == q->Size) {
    printf("queue overflow\n");
    exit(1);
  }
```

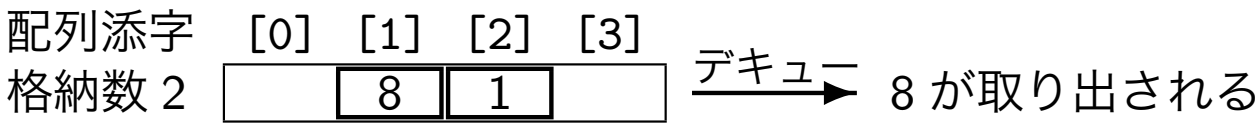
キューからのデータ取出

デキュー

キューからデータを取り出す操作のこと.

- 最も古いデータを取り出す. 格納数は 1 減少する

```
double Dequeue(Queue *q) {
  int i = q->Index;
  q->Count--;
  q->Index = (q->Index + 1) % q->Size;
  return q->Data[i];
}
```



- キューが空の場合はデータを取り出すことができない. これをキューアンダーフローと呼ぶ. このような場合はエラーを表示し, 強制終了する.

```
if (q->Count == 0) {
  printf("queue underflow\n");
  exit(1);
}
```

本日のまとめ

- 動的なメモリの確保
- malloc 関数
- スタック
- キュー