第10章 スタックとキュー

スタック／キュー／環状バッファの応用

種々のアプリケーションを作る中で，特殊なデータ構造を利用することもあります．ここで解説するスタックとキューはその蓄積方法に特徴があります．

データの性格・内容が異なるデータ群に対しては，その格納場所を管理していないと，取り出すときに困ります．ちょうど多くの種類の商品を収納しておく倉庫では，柱や区画の番号，棚の番号を管理して，入庫・出庫するようなものです．区画の番号や棚の番号は，コンピュータのメモリでいえばアドレスということになります．しかし扱っている商品は1種類だけの場合はどうでしょうか．商品はどこの区画から取り出しても同じという場合には，収納する際にはあえて収納場所を特定する必要はなくなります．空いている場所に，あるいは入口から近い順に格納してもかまわないことになります．そして取り出しの際には出口に近い順から取り出してもよいのです．

そのような，収納場所を選ばない格納方式がスタック方式とキュー方式です．コンピュータのメモリでも，格納する場所のアドレスを具体的に管理することなく，必要に応じて放り込んでおく，必要に応じて手近なところから取り出すという方式です．これは格納される情報の内容や性格が一義的な場合に適用できる方式です．

コンピュータのハードウェア自身にはすでにこれらの機能をもっているものもあります．スタック・ポインタが管理するスタック領域などはその典型です．キューについてはパイプライン処理などがその典型でしょう．またファームウェアでその機能を持っているものはあっても，一般のソフトウェアでその機能を持っているものはほとんどありません．必要な場合には自作することになります．この章では，C言語によって，スタックの機能，キューの機能を実現してみます．

10.1 スタック

スタック方式による情報の格納と取り出しを解説します．

10.1.1 スタックとは

スタック（stack）とは本来は刈り草や藁を積み重ねた山のことであり，積み重ねること，積層状にすることの意味です．あまり例はよくないかもしれませんが，漬物桶に，大根や白菜を漬け込むことを想像してください．桶の底から塩や香料を混ぜながら，大根や白菜を置いていきます．重石をして時間をおきます．そして食べるときには，上から順に取り出していきます．つまり漬け込んだ状態は積み重ねの状態であり，取り出すときは，漬け込んだときと逆の順番で取り出していきます．

この様態はLIFO（Last In First Out）と呼ばれます．つまり後入れ先出しの方式です．コンピュータの世界でも，情

報を順に格納し，先に格納した情報を後で取り出すような方式をLIFO方式や，スタック方式などと呼びます．逆の言い方で，FILO（First In Last Out）も同じ意味です．

10.1.2　スタック領域，スタック・ポインタ

スタックによる情報格納の手法は，通常のプロセッサには備わっている機能です．サブルーチン・コールからの戻りや，割り込み発生時の状態の自動退避などに利用されています．そしてマルチタスク（multi task）機能実現の基礎技術になっています．

情報が格納される領域をスタック領域，どこのレベルまで格納されているかを示している指標をスタック・ポインタ（stack pointer）などと呼んでいます．スタック・ポインタはタンクの水位計（gauge）のような働きです．高級言語ではあまり頻出はしませんが，ハードウェアの動きと一体なアセンブラ言語では，つねに意識しながらプログラミングをする必要のある項目で，アセンブラ言語の経験者ならお馴染みの手法です．

スタック領域も，一般の情報記憶領域も同じメモリ，たとえばRAM（Random Access Memory）ですが，プログラム自身の制御で区画し，使い分けをすることになります．

C言語においても，関数の呼び出し，引数の引渡しに利用されています．この機能があるおかげで，構造化もしやすく，再帰（recursive）構造のプログラムも可能になっています．しかし実際には，プログラマとしては無意識に，スタックの機能を使っていてもプログラムはできてしまいます．

10.1.3　先入れ後出し

図10.1.1に先入れ後出しのイメージを示します．データを格納する際には，到達順に，容器の底から順に積み上げます．一方取り出すときには，最上位のものから順に取り出します．結果的にABCDEの順に到達・格納すれば，取り出すときには，EDCBAと逆順になります．

10.1.4　スタックを実現 ― その1

スタックに格納するデータは，前述のとおり定型的なものでなければなりません．C言語の引数の引き渡しにもスタックを利用していて，型はchar型，short型，long型など，さまざまですが，引数を設定する側と受ける側のインターフェースを合わせているので問題は起こりません．しかし関数のプロトタイプ宣言を忘れたときなどに正常に動作しないのは，この引数サイズの違いによるためです．

ここでは，ある領域にデータを格納する，そして取り出す場合に，LIFOになる形態をC言語で作成します．

例題10.1.4　記憶領域を気にせずに，push，popをする関数を作成する．

■アルゴリズム

スタックなどに投入することをpush，逆に取り出すことをpopするなどといいます．方法はいろいろ考えられますが，ここではスタックの領域を配列として確保してみます．**図10.1.1**と同じように，まずスタック領域を底の部分から利用する方法で考えてみます．

要素数nの配列を用意します．格納位置を管理する変数Sposを用意し，スタックの底，n－1に初期化しておきます．これはアセンブラのときのスタック・ポインタの働きをします．

格納するデータが発生したら，配列[Spos]に格納します．そして格納後にSpos--で格納位置をマイナス方向に更新しておきます．この値が0に達したら，スタッ

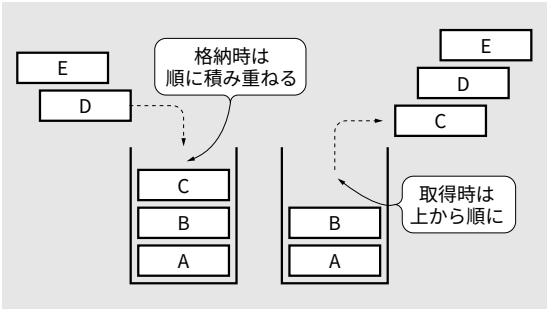

図10.1.1　スタックのイメージ

クの領域は満杯になったことになります．

一方，取り出しのときには，SPos+1の位置からデータを取り出します．C言語でプログラミングする場合には，++Sposなどと，増減演算子を前に付加すればよいわけです．この値が，配列の最大要素番号になればスタック領域にはデータは無くなったということになります．

このアルゴリズムでプログラムしてみます．

■プログラム･リスト

リスト10.1.4にプログラミングの例を示します．初期化の関数STKinit( )，スタックに格納する関数STKput( )，スタックから取り出す関数STKget( )に機能分割してあります．それぞれの関数仕様は，

□ STKinit( )関数

機能　スタック操作に利用する変数を初期化する．スタック利用の前に1回だけ呼び出す．

引数　無し

戻り値　無し

□ STKput( )関数

機能　スタック領域にデータを格納する．

引数　格納するデータ

戻り値　EOF　正常に格納できなかった

>=0　正常に格納した．格納位置．

□ STKget( )関数

機能　スタックからデータを取り出す．

引数　取り出したデータの格納先へのポインタ

戻り値　EOF　データがなかった

>=0　正常に取り出した．取り出し位置．

リスト10.1.4　配列利用のスタック操作(1)

```
/*  スタック(stack)の操作      */
#include <stdio.h>

#define  STKMax  5

void STKInit ( void );                  /* 関数の原型宣言       */
int  STKput ( int  );
int  STKget ( int *);

int Sbuf[STKMax];                       /* 共通変数             */
int Spos;

int main ( void )                       /* テスト用 main関数    */
{
  int job, end=0, dat, stat;

  STKInit ();                           /* スタック初期化関数   */
  do {
    printf ("格納:1  取り出し:2  終了:3  -->>");
    scanf ("%d", &job );
    switch ( job ){
    case 1:
      printf ("  格納するデータ --->>");
      scanf ("%d", &dat );
      stat = STKput ( dat );            /* push 関数            */
      if ( stat == EOF ){
        printf ("  スタックが満杯で格納できませんでした¥n");
      } else {
        printf ("  データを格納しました");
        printf ("  このデータは %d 番目です¥n", STKMax-stat );
      }
      break;
    case 2:
      stat = STKget ( &dat );           /* pop 関数             */
      if ( stat == EOF ){
        printf (" スタックにデータはありませんでした¥n");
      } else {
        printf ("  スタックから取り出したデータは %d です", dat );
        printf ("  このデータは %d 番目でした¥n", STKMax-stat );
      }
      break;
    case 3:
      while ( EOF !=  STKget(&dat) ){ /* 残りをpop            */
        printf ("スタックに残っていたデータ %d ¥n", dat );
      }
      end = 1;
      break;
    default:
      printf ("指定が違います¥n");
    }
  } while ( end == 0 );
  return 0;
}

/*  スタック処理の初期化関数  */
void STKInit (void)
{
  Spos = STKMax-1;                      /* スタックポインタ設定 */
}

/*  スタックにデータを書き込む関数    */
int STKput ( int dat )                  /* 引数は格納データ     */
{
  if ( Spos < 0 ){                      /* 先頭に達しているとき */
    return EOF;                         /* エラー・リターン     */
  }
  Sbuf[Spos--] = dat;                   /* スタックに書き込む   */
  return Spos+1;                        /* 格納位置を返す       */
}

/*  スタックからデータを取り出す関数    */
int STKget ( int *pi )                  /* 引数は格納先ポインタ */
{
  if ( Spos >= STKMax-1 ){              /* スタックポインタ初期 */
    return EOF;                         /* データなし           */
  }
  *pi = Sbuf[++Spos];                   /* データ取出し         */
  return Spos;                          /* 取出し位置を返す     */
}
```

とします．

ここでは，動きの確認だけなので，配列の大きさは5としています．テスト用のmain()関数は，スタック操作の動きがわかるように，メニュー方式にしてあります．これで随時出し入れができ，そのときの状態も監視できます．

スタック操作においては，利用する側，すなわちここではmain側では，どこの位置に格納するか，あるいはどこの位置から取り出すかということは考慮しません．格納すべきデータが発生したら送り込む，取り出したいときは要求するという操作だけです．これらの要請を受けて，スタック操作の関数群がブラックボックスで処理をします．

スタック領域として確保した容量以上は格納できませんので，つねに監視は必要です．現在どの程度の利用状況かを監視する関数もあればよいでしょう．次の節で追加してみます．

10.1.5　スタックを実現 ― その2

前例では容器の底から格納する説明をしています．しかしスタックはブラックボックスで，利用側は内部動作の詳細を関知しなくてもよいわけですから，スタック領域，つまりここでは配列を先頭から下方へ利用することでもよいわけです．ではその方法も考えてみます．むしろ，この方法のほうがわかりやすいかもしれません．機能は前述と同様です．

例題10.1.5　配列を先頭から利用するスタック機能を実現する．

スタック領域利用のイメージを**図10.1.2**に示します．Sposはここで

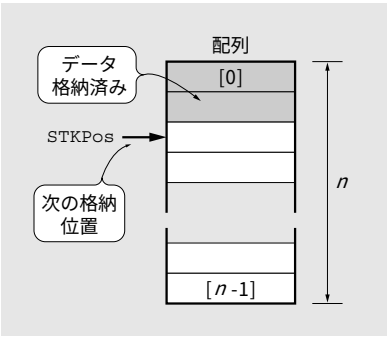

図10.1.2　配列利用のスタック領域

リスト10.1.5　配列利用のスタック操作(2)

```
/*  スタック(stack)の操作 -2    */
#include <stdio.h>

#define STKMax  5

void STKinit2 ( void );                 /* 関数の原型宣言       */
int  STKput2 ( int  );
int  STKget2 ( int *);
int  STKcheck ( void );

int Sbuf[STKMax];                       /* 共通変数             */
int Spos;

/*  テストのためのmain関数      */
int main ( void )
{
  int job, end=0, dat, stat;

  STKinit2 ();                          /* スタック処理初期化   */
  do {
    printf ("格納:1  取り出し:2  終了:3  -->>");
    scanf ("%d", &job );
    switch ( job ){
    case 1:
      printf ("  格納するデータ --->>");
      scanf ("%d", &dat );
      stat = STKput2 ( dat );           /* push                 */
      if ( stat == EOF ){
        printf ("  スタックが満杯で格納できませんでした¥n");
      } else {
        printf ("  データを格納しました");
        printf ("  スタックは %d 個空いています¥n", STKcheck() );
      }
      break;
    case 2:
      stat = STKget2 ( &dat );          /* pop                  */
      if ( stat == EOF ){
        printf (" スタックにデータはありませんでした¥n");
      } else {
        printf ("  スタックから取り出したデータは %d です", dat );
        printf ("  スタックは %d 個空いています¥n", STKcheck() );
      }
      break;
    case 3:
      while ( EOF != STKget2 (&dat)){ /* 残りをpop             */
        printf ("スタックに残っていたデータ %d ¥n", dat );
      }
      end = 1;
      break;
    default:
      printf ("指定が違います¥n");
    }
  } while ( end == 0 );
  return 0;
}

/*  スタック処理の初期化関数    */
void STKinit2 (void)
{
  Spos = 0;                             /* スタックポインタ設定 */
}

/*  スタックにデータを書き込む関数          */
int STKput2 ( int dat )                 /* 引数は格納データ     */
{
  if ( Spos >= STKMax ){                /* 最大値に達している   */
    return EOF;                         /* エラー・リターン     */
  }
  Sbuf[Spos++] = dat;                   /* スタックに書き込む   */
  return Spos-1;                        /* 格納位置を返す       */
}

/*  スタックからデータを取り出す関数        */
int STKget2 ( int *pi )                 /* 引数は格納先ポインタ */
{
  if ( Spos <= 0 ){                     /* スタック初期位置     */
    return EOF;                         /* 格納データなし       */
  }
  *pi = Sbuf[--Spos];                   /* データ取出し         */
  return Spos+1;                        /* 格納位置を返す       */
}

/*  スタックの空きをチェックする関数        */
int STKcheck ( void )
{
  return STKMax - Spos;                 /* 戻り値は空き容量     */
}
```