SDD : TP 2

Mathieu Boutin - Jérémy Morceaux

March 26, 2018

1 Présentation générale

- Ce TP a pour but de créer les fonctions de base permettant de manipuler des piles et des files qui seront utiles pour le prochain TP. A l'aide de ses fonctions, on inverse une pile grâce à une file supplémentaire en vidant la pile dans la file, puis inversement, en vidant la file dans la pile.

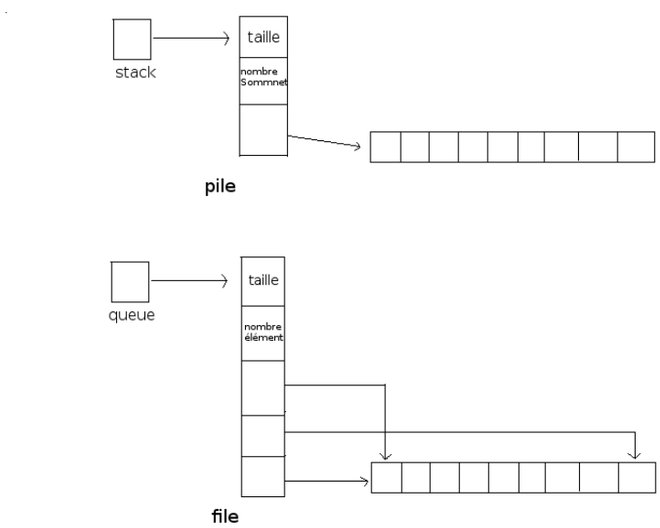

Schéma de base

Structure des fichiers de pile : prenons par exemple une pile de taille n

1ère ligne fichier : <taille de la pile>
2ème ligne du fichier: $valeur_0$ $valeur_1$ ... $valeur_{n-1}$

- Les fichiers sources se trouvent dans le dossier **src**. Les fonctions relatives aux piles sont dans le fichier **ZZ_stack.c** et celles des files sont dans le fichier **ZZ_queue.c**. Les entêtes sont dans le fichier **include**.

2 Détail de chaque fonction

2.1 initStack

Lexique :

- Paramètre(s) de la fonction
  - size est la taille de la pile.
  - errorCode est un pointeur sur un entier qui indique si la fonction s'est bien déroulée.
- Variable(s) locale(s)
  - p est un pointeur de la pile.

Programme commenté :

```c
/* ------------------------------------------------------------------- */
/* initStack        Init a stack properly                              */
/*                                                                     */
/* Inputs :    - size is the size of the queue                         */
/*             - errorCode is an integer that stores the result of the */
/*               function :  -1 - there is an issue while allocating   */
/*                               the queue                             */
/*                                                                     */
/*                           0 - there is an issue while allocation the */
/*                               the array in the queue                */
/*                                                                     */
/*                           1 - Everything went well !                */
/*                                                                     */
/* Output :  Return the adress of the pointer of the head of the stack */
/* ------------------------------------------------------------------- */
stack_t * initStack(int size, int * errorCode)
{
    stack_t * p;   /* we declare a pointer of the stack */

    *errorCode=-1;

    p = malloc(sizeof(stack_t));    /* we initialize the pointer */

    if (p != NULL)   /* If the pointer is not null we initialize the point stack */
    {
            p->begin = malloc(size * sizeof(typeStack)); /* we initalize the contiguous list which is pointed by the last block of the stack */
            *errorCode = 0;

            if ((p->begin)!= NULL)           /* if the previous allocation went wrong */
            {
                p->sizeMax = size;
                p->numSummit = -1;     /*there is 0 element in the stack but the stack is created so we put -1 */
                *errorCode = 1;
            }
            else
            {
                free(p);
                p=NULL;
            }
    }

    return p;
}
```

initStack

2.2 initStackFromFile

Lexique :

- Paramètre(s) de la fonction
  - fileName est le nom du fichier contenant la pile
  - errorCode est un pointeur sur un entier qui indique si la fonction s'est bien déroulée.
- Variables(s) locale(s)
  - stack est le pointeur de la pile.
  - size contient la taille de la pile.
  - codeE_init errorCode est un pointeur sur un entier qui indique si la fonction init s'est bien déroulée.
  - codeE_push errorCode est un pointeur sur un entier qui indique si la fonction push s'est bien déroulée.
  - issue indique s'il y a eu un problème pendant un push.
  - file est le flux du fichier.
  - value contient une valeur à mettre dans la pile.

Programme commenté :

```c
/* ---------------------------------------------------------------------- */
/* initStackFromFile        Init a stack from a file                      */
/*                                                                        */
/* Inputs :   - fileName is the name of the file containing the stack.    */
/*            - errorCode is an integer that stores the result of the     */
/*              function :  -1 - there is an issue while trying to open   */
/*                              the file                                  */
/*                                                                        */
/*                           0 - there is an issue while allocation the   */
/*                              the array in the queue                    */
/*                                                                        */
/*                           1 - Everything went well !                   */
/*                                                                        */
/* Output :  Return the adress of the pointer of the head of the stack    */
/* ---------------------------------------------------------------------- */
stack_t * initStackFromFile(char * fileName, int *errorCode)
{

    stack_t*    stack;
    int         size;
    int         codeE_init; /* errorCode for init */
    int         codeE_push;
    int         issue;
    FILE*       file = fopen(fileName,"r"); /* creation of a flow */

    issue = 0;

    if(file != NULL) /* We make sure, we actually opened the file */
    {
        *errorCode = 0;

        /* We can get the length of the stack with the first line*/
        fscanf(file,"%d",&size);

        stack = initStack(size,&codeE_init); /* We initialize our stack */

        if(codeE_init == 1)
        {
            typeStack value;

            /* Everything went well */
            *errorCode = 1;

            /* we fill our stack with all the value we have */
            while(fscanf(file,"%d ",&value) == 1)
            {
                /* check our stack is not full, and there is no error related */
                if(issue == 0)
                {
                    /* We try to push this value */
                    push(stack,value,&codeE_push);

                    if(codeE_push != 1) /* if we had a problem during pushing, we stop the process */
                    {
                        *errorCode = 0;
                        issue = 1;
                    }
                }
            }
        }

        fclose(file); /* Don't forget to close the flow */
    }
    else
    {
        *errorCode = -1; /* We had a problem while opening the file*/
    }


    return(stack);
}
```

initStackFromFile

2.3 printStack

Lexique :

- Paramètre(s) de la fonction
  - p est la tête fictive de la pile.
- Variables(s) locale(s)
  - i permet de parcourir la pile.

Programme commenté :

```c
/* ------------------------------------------------------------------- */
/* printStack:        print a stack to a terminal                      */
/*                                                                     */
/* Input :    - p0 is a pointer of the stack                           */
/*                                                                     */
/* ------------------------------------------------------------------- */
void printStack(stack_t* p)
{
    if(!isStackEmpty(p))
    {
        int i;

        for(i = 0;i <= p->numSummit;i++)    /* We fill our stack */
        {
            printf("%d ",p->begin[i]);
        }
        printf("\n");
    }
    else
    {
        printf("The stack is empty\n");
    }
}
```

printStack

2.4 freeStack

Lexique :

- Paramètre(s) de la fonction
  - p est la tête fictive de la pile.

Programme commenté :

```c
/* ------------------------------------------------------------------- */
/* freeStack:        Free a stack properly                             */
/*                                                                     */
/* Inputs :    - p0 is a pointer of the stack                          */
/*                                                                     */
/* ------------------------------------------------------------------- */
void freeStack(stack_t * p)
{
    if (p->begin != NULL)  /* if there is a contiguous list, we free the pointer which link the stack to the contigous list */
    {
        free(p->begin);

    }
    if(p != NULL)         /* so the pointer of the contiguous list is maybe null but we have to free the stack */
    {
        free(p);
    }
    p = NULL;

}
```

freeStack

2.5 isStackEmpty

Lexique :

- Paramètre(s) de la fonction
  - p est le pointeur de tête fictive de la pile.

Programme commenté :

```c
/* ------------------------------------------------------------------- */
/* isStackEmpty:       Check if the stack is empty                     */
/*                                                                     */
/* Input :      - p0 is a pointer of the stack                         */
/* Output:      Return  boolean :                                      */
/*                      * 0 if the stack is not empty                  */
/*                      * 1 if it is                                   */
/* ------------------------------------------------------------------- */
int isStackEmpty(stack_t * p)
{
    return (p->numSummit == -1);  /* if the stack is empty so there is 0 element , so numSummit = -1 */
}
```

isStackEmpty

2.6 push

Lexique :

- Paramètre(s) de la fonction
  - p est le pointeur de tête fictive de la pile.
  - errorCode est un pointeur sur un entier qui indique si la fonction s'est bien déroulée.
  - v est la valeur que l'on veut mettre dans la pile.

Programme commenté :

```c
/* ---------------------------------------------------------------------- */
/* push:            Push a value in the stack                             */
/*                                                                        */
/* Inputs :     - p0 is a pointer of the stack                            */
/*              - v is a value we want to stack                           */
/*              - errorCode is an integer that stores the result of the   */
/*                function :  0 - the stack is full !                     */
/*                                                                        */
/*                            1 - Everything went well !                  */
/* ---------------------------------------------------------------------- */
void push(stack_t * p, typeStack v ,int * errorCode)
{
        *errorCode = 0;

        if (p->numSummit+1 <= (p->sizeMax) ) /* if the stack is not full */
        {
                p->begin[p->numSummit+1] = v; /* we assign to the first free block of the contiguous list the value v */
                p->numSummit += 1;  /* there is a further element in the stack */
                *errorCode = 1;
        }

}
```

push

2.7 pop

Lexique :

- Paramètre(s) de la fonction
  - p est le pointeur de tête fictive de la pile.
  - errorCode est un pointeur sur un entier qui indique si la fonction s'est bien déroulée.
  - v est la variable dans laquelle on va mettre l'élément que l'on dépile.

Programme commenté :

```c
/* ---------------------------------------------------------------------*/
/* pop:        Unstack a value from the stack                           */
/*                                                                      */
/* Inputs :     - p0 is a pointer of the stack                          */
/*              - *v is the adress of the value                         */
/*                 we want to unstack                                   */
/*              - errorCode is an integer that stores the result of the */
/*                function :  0 - the stack is empty, nothing to        */
/*                            retrieve                                  */
/*                                                                      */
/*                          1 - Everything went well !                  */
/* ---------------------------------------------------------------------- */
void pop( stack_t * p, typeStack * v, int * errorCode)
{
        *errorCode = 0;

        if (!isStackEmpty(p))      /* if the stack is not empty so we can unstack */
        {
                *v = p->begin[p->numSummit];  /*  we pick up the value of the last element added and we assign it in a variable */
                p->numSummit -= 1; /* we unstack the last element added so there is one element less stored in the stack */

                *errorCode = 1;
        }
}
```

pop

2.8 initQueue

Lexique :

- Paramètre(s) de la fonction
  - size est la taille de la file.
  - errorCode est un pointeur sur un entier qui indique si la fonction s'est bien déroulée.
- Variables(s) locale(s)
  - p0 est le pointeur vers la file qui sera retournée.

Programme commenté :

```c
/* ------------------------------------------------------------------- */
/* initQueue         Init a queue properly                             */
/*                                                                     */
/* Input :     - p0 is a pointer to the queue                          */
/*             - size is the size of the queue                         */
/*             - errorCode is an integer that stores the result of the */
/*               function :  -1 - there is an issue while allocating  */
/*                               the queue                             */
/*                                                                     */
/*                           0 - there is an issue while allocation the */
/*                               the array in the queue                */
/*                                                                     */
/*                           1 - Everything went well !                */
/* ------------------------------------------------------------------- */
queue_t* initQueue(int size,int* errorCode)
{
        queue_t * p0;

        *errorCode = -1;

        /* We create the queue */
        p0 = (queue_t*)malloc(sizeof(queue_t));

        if(p0 != NULL)
        {
                /* Init the queue data */
                p0->size = size;
                p0->number = 0;

                /* allocate an array to store the data */
                p0->base = (queueType*)malloc(size * sizeof(queueType));
                *errorCode = 0;

                /* If the allocation went well */
                if(p0->base != NULL)
                {
                        *errorCode = 1;
                        p0->start = 0;
                        p0->end = size-1;
                }
        }

        return(p0); /* return the pointer of the queue */
}
```

initQueue

2.9 enterQueue

Lexique :

- Paramètre(s) de la fonction
  - p0 est le pointeur de tête fictive de la file.
  - errorCode est un pointeur sur un entier qui indique si la fonction s'est bien déroulée.
  - element est l'élément que l'on veut insérer dans la file.

Programme commenté :

```c
/* ------------------------------------------------------------------- */
/* enterQueue         Put a value in the queue                         */
/*                                                                     */
/* Input :     - p0 is a pointer to the queue.                         */
/*             - element is the value that is entering the queue.      */
/*             - errorCode is an integer that stores the result of the */
/*               function :   0 - the queue is full, no more data can  */
/*                               enter the queue                       */
/*                                                                     */
/*                           1 - everything went well !!               */
/* ------------------------------------------------------------------- */
void enterQueue(queue_t* p0,queueType element,int* errorCode)
{
        *errorCode = 0;
        /* if the queue is not full */
        if(p0->number <= p0->size)
        {
                *errorCode = 1;
                /* we actualize the index end, and the number of value in the queue*/
                p0->end = (p0->end + 1) % p0->size;
                p0->base[p0->end] = element;
                p0->number += 1;

        }
        else /* the queue is full */
        {
                fprintf(stderr,"[WARNING] This queue is full \n");
        }
}
```

enterQueue

2.10 leaveQueue

Lexique :

- Paramètre(s) de la fonction
  - p0 est le pointeur de tête fictive de la file.
  - errorCode est un pointeur sur un entier qui indique si la fonction s'est bien déroulée.
- Variables(s) locale(s)

– res est la variable qui contient l'élément qui sera défilé.

Programme commenté :

```c
/* ------------------------------------------------------------------- */
/* leaveQueue         Remove a value in the queue                      */
/*                                                                     */
/* Input :    - p0 is a pointer to the queue.                          */
/*            - errorCode is an integer that stores the result of the  */
/*              function :   0 - the queue is empty, there is no data  */
/*                               to retrieve                           */
/*                                                                     */
/*                           1 - everything went well                  */
/* ------------------------------------------------------------------- */
queueType leaveQueue(queue_t* p0,int* errorCode)
{
        queueType res;
        *errorCode = 0;

        /* Make sure the queue is not empty */
        if(!isQueueEmpty(p0))
        {
                *errorCode = 1;
                /* retrieve the value which will be returned */
                res = p0->base[p0->start];

                /* actualiaze the index start, and the number of value in the queue */
                p0->start = (p0->start + 1 ) % p0->size;
                p0->number -= 1;
        }

        return(res); /* return the value */
}
```

leaveQueue

2.11 isQueueEmpty

Lexique :

- Paramètre(s) de la fonction
  - p0 est le pointeur de tête fictive de la file.

Programme commenté :

```c
/* ------------------------------------------------------------------- */
/* isQueueEmpty         Check if the queue is empty or not             */
/*                                                                     */
/* Input :    - p0 is a pointer to the queue.                          */
/*                                                                     */
/* Ouput :  - 1 : the queue is empty                                   */
/*          - 0 : it isn't                                             */
/* ------------------------------------------------------------------- */
int isQueueEmpty(queue_t* p0)
{
        return(p0->number == 0); /* simply return either the queue is full or not */
}
```

isQueueEmpty

2.12 freeQueue

Lexique :

- Paramètre(s) de la fonction
  - p0 est la tête fictive de la file.

Programme commenté :

```c
/* ------------------------------------------------------------------- */
/* freeQueue         Free correctly a queue                            */
/*                                                                     */
/* Input :    - p0 is a pointer to the queue.                          */
/*                                                                     */
/* ------------------------------------------------------------------- */
void freeQueue(queue_t* p0)
{
        if(p0->base != NULL) /* we first free the array */
        {
                free(p0->base);
        }
        if(p0 != NULL) /* and then we can free the queue */
        {
                free(p0);
        }
}
```

freeQueue

2.13 printQueue

Lexique :

- Paramètre(s) de la fonction
  - p0 est la tête fictive de la file.
- Variables(s) locale(s)
  - i permet de parcourir la file.

Programme commenté :

```c
/* ------------------------------------------------------------------- */
/* printQueue         print a Queue to a terminal                      */
/*                                                                     */
/* Input :    - p0 is a pointer to the queue.                          */
/*                                                                     */
/* ------------------------------------------------------------------- */
void printQueue(queue_t *p0)
{
        int i;

        /* simply display each value of the queue */
        for(i=0;i < p0->number;i++)
        {
                printf("%d ",p0->base[(p0->start +i ) % p0->size]);
        }
        printf("\n");
}
```

printQueue

2.14 main

Lexique :

- Variables(s) locale(s)
  - queue pointe vers la pile.
  - stack pointe vers la file.
  - errorCode est un pointeur sur un entier qui indique si une fonction s'est bien déroulée.
  - fileName contient le nom du fichier contenant la pile.
  - i permet de vider la pile pour remplir la file et inversement.
  - numberSummit contient le nombre de valeur dans la pile.
  - empty et full sont des variables permettant de stopper les boucles for au cas où il y est une erreur (une pile vide ou pleine, de même pour la file).

Programme commenté :

```c
int main(int argc, char * argv[])
{
    if(argc == 2)
    {
        /* Declare our two structures */
        queue_t     * queue;
        stack_t     * stack;
        int           errorCode;
        char        * fileName;

        /* we registered some stack in the Stack directory */
        fileName = argv[1];

        printf("%sI/ Loading the stack from the file %s.%s\n",VERT,fileName,NORMAL);

        /* Init our stack */
        stack = initStackFromFile(fileName,&errorCode);

        if(errorCode == 1)
        {
            int i;
            int numberSummit; /* save the number of summit */
            int empty;
            int full;


            empty = 0; /* once a stack is empty, we have to stop the iteration */
            full = 0; /* same thing here*/

            numberSummit = stack->numSummit + 1;

            /* Init our queue */
            queue = initQueue(numberSummit,&errorCode);

            /* Display it to the screen */
            printf("Current stack : ");
            printStack(stack);

            printf("%sII/ Emptying the stack to fill the queue.%s\n",VERT,NORMAL);
            /* We empty our stack and fill in a queue */
            for(i=0;i < numberSummit; i++)
            {
                if(empty == 0 && full == 0)
                {
                    typeStack valueStack;

                    /* We retrieve the value from the queue */
                    valueQueue = leaveQueue(queue,&errorCode);

                    /* If the queue is not empty, we fill the value to the stack */
                    if(errorCode == 1)
                    {
                        push(stack,valueQueue,&errorCode);

                        /* We can't push the value because the stack is full */
                        if(errorCode == 0)
                        {
                            fprintf(stderr,"[WARNING] You're trying to push data to a full stack \n");
                            full = 1;
                        }
                    }
                    else
                    {
                        fprintf(stderr,"[WARNING] You're trying to remove data from an empty queue \n");
                        empty = 1;
                    }
                }
            }

            printf("%sIV/ Result : %s\n",VERT,NORMAL);

            /* Once it's done, we can print the reversed stack */
            printf("Reversed stacked : ");
            printStack(stack);

            /* And of course, don't forget to free the stack and the queue */
            freeStack(stack);
            freeQueue(queue);
        }
        else
        {
            if(errorCode == -1)
            {
                fprintf(stdout,"%s[ERROR] Can't open the file %s !%s\n",ROUGE,fileName,NORMAL);
            }
            else
            {
                fprintf(stdout,"%s[ERROR] Don't have enough space to allocate this stack%s\n",ROUGE,NORMAL);
            }
        }
    }
    else
    {
        fprintf(stdout,"%s[ERROR] Too few arguments ! %s\n",ROUGE,NORMAL);
    }
    return(EXIT_SUCCESS);
}
```

main

3 Compte rendu d'exécution

Makefile :

```
CC = gcc
LD_FLAGS = -I./include -I./src
CX_FLAGS = -Wall -Wextra -ansi -pedantic -I./include -I./src

all: main

main: main.o stack.o queue.o
    $(CC) ./obj/main.o ./obj/queue.o ./obj/stack.o -o ./bin/main_program $(LD_FLAGS)

main.o: ./src/main.c ./include/ZZ_queue.h ./include/ZZ_stack.h
    $(CC) -c ./src/main.c -o ./obj/main.o $(CX_FLAGS)

stack.o: ./src/ZZ_stack.c ./include/ZZ_stack.h
    $(CC) -c ./src/ZZ_stack.c -o ./obj/stack.o $(CX_FLAGS)

queue.o: ./src/ZZ_queue.c ./include/ZZ_queue.h
    $(CC) -c ./src/ZZ_queue.c -o ./obj/queue.o $(CX_FLAGS)
```

Makefile

Jeux de test complets:

```
Trace 0: Pas assez d'argument

[FORMAT] : ./bin/main_program fileNameStack

./bin/main_program

[ERROR] Too few arguments !

Press any key to continue ...
```

Terminal

```
Trace 1: Nom de fichier invalide

[FORMAT] : ./bin/main_program fileNameStack

./bin/main_program Stack/hello_world.txt

I/ Loading the stack from the file Stack/hello_world.txt.
[ERROR] Can't open the file Stack/hello_world.txt !

Press any key to continue ...
```

Terminal

```
Trace 2 : Stack vide

[FORMAT] : ./bin/main_program fileNameStack

./bin/main_program Stack/stack0.txt

I/ Loading the stack from the file Stack/stack0.txt.
Current stack : The stack is empty
II/ Emptying the stack to fill the queue.
III/ Emptying the queue to fill the stack.
IV/ Result :
Reversed stacked : The stack is empty

Press any key to continue ...
```

```
stack0.txt ×   stack1.txt ×   stack_
0
```

Terminal | fichier pile

```
Trace 3 : Stack 1 élément

[FORMAT] : ./bin/main_program fileNameStack

./bin/main_program Stack/stack1.txt

I/ Loading the stack from the file Stack/stack1.txt.
Current stack : 15
II/ Emptying the stack to fill the queue.
III/ Emptying the queue to fill the stack.
IV/ Result :
Reversed stacked : 15

Press any key to continue ...
```

stack0.txt × | stack1.txt ×

```
1
15
```

Terminal | fichier pile

```
Trace 4 : Stack pleine

[FORMAT] : ./bin/main_program fileNameStack

./bin/main_program Stack/stack_full.txt

I/ Loading the stack from the file Stack/stack_full.txt.
Current stack : 1 5 7 8 9 6 5
II/ Emptying the stack to fill the queue.
III/ Emptying the queue to fill the stack.
IV/ Result :
Reversed stacked : 5 6 9 8 7 5 1

Press any key to continue ...
```

stack_full.txt × | stack_exceed.txt

```
7
1 5 7 8 9 6 5
```

Terminal | fichier pile

```
Trace 5 : Stack à moitié vide/pleine

[FORMAT] : ./bin/main_program fileNameStack

./bin/main_program Stack/stack_semi_full.txt

I/ Loading the stack from the file Stack/stack_semi_full.txt.
Current stack : 1 2 3 4
II/ Emptying the stack to fill the queue.
III/ Emptying the queue to fill the stack.
IV/ Result :
Reversed stacked : 4 3 2 1

Press any key to continue ...
```

stack_semi_full.txt × | stack_full.txt ×

```
10
1 2 3 4
```

Terminal | fichier pile