La pile d'entiers et son application à la traduction d'une notation infixée et postfixée

Implémentation d'une (unique) pile d'entiers

Cette pile est de taille fixe, retrouvez en Figure 1 le code source de l'implémentation présentée en cours.

```c
/*!\file pile.h
 * \brief Bibliothèque de gestion de (une) pile de taille fixe
 * \author Farès Belhadj amsi@up8.edu
 * \date September 27, 2019
 */
#ifndef _PILE_H
#define _PILE_H
/*!\brief taille de la pile (statique) */
#define PILE_MAX 256

#ifdef __cplusplus
extern "C" {
#endif

  extern void push(int v);
  extern int  pop(void);
  extern int  empty(void);

#ifdef __cplusplus
}
#endif

#endif
```

```c
/*!\file pile.c
 * \brief Bibliothèque de gestion de (une) pile de taille fixe
 * \author Farès Belhadj amsi@up8.edu
 * \date September 27, 2019
 */
#include "pile.h"
#include <stdlib.h>
#include <assert.h>

/*! \brief indice indiquant le haut de la pile. */
static int _haut = -1;
/*! \brief tableau static utilisé pour le stockage de la pile. */
static int _pile[PILE_MAX];

/*!\brief Empiler la valeur \a v dans la pile.
 * \param v la valeur à empiler */
void push(int v) {
  _pile[++_haut] = v;
}

/*!\brief dépiler et renvoyer la valeur de l'élément en haut de la pile.
 * \return la valeur en haut de la pile. */
int pop(void) {
  return _pile[_haut--];
}

/*!\brief Indique si la pile est vide.
 * \return vrai si la pile est vide, faux sinon. */
int empty(void) {
  return _haut < 0;
}
```

FIGURE 1 – Bibliothèque de gestion d'une pile d'entiers (fichiers `pile.h` et `pile.c`).

Notations infixée, préfixée, polonaise et postfixée

≪ Les notations infixée (ou infixe), préfixée (ou préfixe) et postfixée (ou postfixe) sont des formes d'écritures d'expressions algébriques qui se distinguent par la position relative qu'y prennent les opérateurs et leurs opérandes. Un opérateur est écrit avant ses opérandes en notation préfixée, entre ses opérandes en notation infixée et après ses opérandes en notation postfixée.

La notation infixée n'a de sens que pour les opérateurs prenant exactement deux opérandes. C'est la notation la plus courante des opérateurs binaires en mathématiques. Les notations préfixée et postfixée permettent de se passer de parenthèses, conduisant à une notation plus compacte. Mais se passer de parenthèse suppose que l'on connaît la signature (autrement dit l'arité de tous les opérateurs) et que cette arité est un attribut des opérateurs qui ne peut pas être modifiable2. La signature sert à analyser les expressions lors de leur évaluation.

La notation préfixée fut proposée en 1924 par le mathématicien polonais Jan Łukasiewicz, c'est pourquoi elle est également appelée notation de Łukasiewicz, ou notation polonaise. Par analogie, la notation postfixée est appelée notation polonaise inverse. Ces deux notations (préfixée et postfixée) permettent de se passer de parenthèses dans le cas d'opérateurs d'arité fixée et connue et s'accordent à une évaluation naturelle de l'expression.≫

Ces définitions sont tirées du résumé de la page Wikipedia titrée "Notations infixée, préfixée, polonaise et postfixée".

La source de l'article est Notations infixée, préfixée, polonaise et postfixée du Wikipédia en français (auteurs)

Le code source présenté en Figure 2 permet de traduire automatiquement une expression infixée, saisie par l'utilisateur, en expression postfixée. Le `Makefile` donné en Figure 3 permet de fabriquer le programme en utilisant la bibliothèque "pile d'entiers" ; il contient d'autres fonctionnalités, à vous de lire et de les découvrir.

```
1  /*!\file infix2postfix.c
2   *
3   * \brief Utilisation d'une pile pour transformer des calculs infixés
4   * (bien parenthésés) en postfixés (écriture polonaise inverse).
5   *
6   * \author Farès Belhadj amsi@up8.edu
7   * \date September 27, 2019
8   */
9  #include "pile.h"
10 #include <stdio.h>
11
12 /*!\brief écrire dans \a d le contenu de la chaîne \a s en transformant
13  * les calculs infixés en opérations postfixées.
14  * \param s la chaîne source
15  * \param d la chaîne destination
16  */
17 static void infix2postfix(char * s, char * d) {
18   while(*s) {
19     if(*s >= '0' && *s <= '9') {
20       do {
21         *d++ = *s++;
22       } while( *s >= '0' && *s <= '9');
23       *d++ = ' ';
24       if(!*s) break;
25     }
26     if((*s == ')')  && !empty()) { *d++ = (char)pop(); *d++ = ' '; }
27     else if(*s == '+') push((int) *s);
28     else if(*s == '*') push((int) *s); /* et les autres ? - et / */
29     s++;
30   }
31   while(!empty()) {*d++ = (char)pop(); *d++ = ' '; }
32   *d = '\0';
33 }
34
35 int main(int argc, char ** argv) {
36   char source[BUFSIZ], destination[BUFSIZ << 1];
37   do {
38     if(!fgets(source, BUFSIZ, stdin)) break;
39     infix2postfix(source, destination);
40     printf("l'expression infixée : %s", source);
41     printf("s'écrit : %s en postfixé\n", destination);
42   } while(1);
43   return 0;
44 }
```

FIGURE 2 – Transformer une expression infixée (bien parenthésée) en expression postfixée (ou polonaise inverse) en utilisant la structure de données pile (fichier `infix2postfix.c`).

```
1  # Auteur Farès Belhadj amsi@ai.univ-paris8.fr
2  # Date le 02/10/2013
3  # Makefile générique
4
5  # définition des commandes utilisées
6  CC = gcc
7  ECHO = echo
8  RM = rm -f
9  TAR = tar
10 MKDIR = mkdir
11 CHMOD = chmod
12 CP = rsync -R
13 # déclaration des options pour gcc
14 PG_FLAGS =
15 CFLAGS = -Wall -O3 $(PG_FLAGS)
16 CPPFLAGS = -I.
17 LDFLAGS = $(PG_FLAGS)
18 # définition des fichiers et dossiers
19 PROG = infix2postfix
20 PACKAGE = infix2postfix
21 VERSION = 1.0
22 distdir = $(PACKAGE)-$(VERSION)
23 HEADERS = pile.h
24 SOURCES = infix2postfix.c pile.c
25 OBJS = $(SOURCES:.c=.o)
26 DOXYFILE = documentation/Doxyfile
27 DISTFILES = $(SOURCES) Makefile $(HEADERS) $(DOXYFILE)
28
29 all: $(PROG)
30
31 $(PROG): $(OBJS)
32         $(CC) $^ $(LDFLAGS) -o $(PROG)
33
34 %.o: %.c
35         $(CC) $(CPPFLAGS) $(CFLAGS) -c $<
36
37 doc: $(DOXYFILE)
38         cd documentation && doxygen && cd ..
39
40 dist: distdir
41         $(CHMOD) -R a+r $(distdir)
42         $(TAR) czvf $(distdir).tgz $(distdir)
43         $(RM) -r $(distdir)
44 distdir: $(DISTFILES)
45         $(RM) -r $(distdir)
46         $(MKDIR) $(distdir)
47         $(CHMOD) 777 $(distdir)
48         $(CP) $(DISTFILES) $(distdir)
49 clean:
50         @$(RM) $(PROG) $(OBJS) *~ $(distdir).tgz gmon.out documentation/*~ -r documentation/html
```

FIGURE 3 – Un exemple de `Makefile` permettant de fabriquer le traducteur `infix2postfix` à partir du fichier `infix2postfix.c` et de la bibliothèque de gestion de d'une pile d'entiers.

DM01

— En utilisant la fonction `infix2postfix` (la modifier pour prendre en compte les flottant - nombres décimaux) et une structure de Pile (support $n^o 2$), réaliser le programme *miniCalculatrice* (votre archive miniCalculatrice.tgz contient uniquement vos sources (.c et .h) et un Makefile ; ce dernier commence avec 3 lignes de commentaires indiquant respectivement **NOM : <nom et prénom>, NUMERO : <numéro d'étudiant>, EMAIL : <email>**) capable de calculer la valeur d'une expression infixée et parenthésée dans laquelle nous utilisons des opérateurs binaires (deux arguments) tels que "+, -, *, /" et des entiers naturels ou des flottant dont nous garderons que les trois premières décimales.

— Utiliser la bibliothèque `ratio` afin d'obtenir au préalable un résultat sous la forme d'une fraction rationnelle. Ainsi, voici comment doît se dérouler une exécution réussie du DM attendu :

— CAS 1 :

```
bash$ ./miniCalculatrice
(((47.500+22.500)/3.000) + (28.000 / 6.500))
47.500 22.500 + 3.000 / 28.000 6.500 / +
1078 / 39 = 27.641
bash$
```

— CAS 2 :

```
bash$ echo "(((47.5+22.5)/3) + (28 / 6.50))" | ./miniCalculatrice
47.500 22.500 + 3.000 / 28.000 6.500 / +
1078 / 39 = 27.641
bash$
```

La procédure de remise du **DM01** sera disponible en ligne (sur la page du cours et non sur le moodle) à partir du 5 octobre. La date limite de rendu diffère en fonction du groupe :

— Pour le groupe A-1, le 19 octobre à 10h30 ;
— Pour le groupe A-2, le 26 octobre à 23h59 ;
— Pour tout le groupe B, le 27 octobre à 23h59.

Tout rendu hors délais ne sera pas comptabilisé.

Perfectionnement

1. **(Pour DM01)** Écrire une fonction `void parse_numbers(const char * s);` parcourant la chaîne `s` et permettant d'extraire les nombres (entiers ou décimaux) qui y sont stockés. Ces nombres seront imprimés à la fois sous la forme de décimaux au millième près et aussi sous la forme de fraction au millième près à l'aide de la bibliothèque ratio (multipliez le flottant par 1000 et stockez-le dans le numérateur entier du rationnel ; mettez 1000 au dénominateur). Voici un exemple d'exécution appliqué à la chaîne "`Hello 0.5 abcd1.4xyz 6`" :
   Hello
   **** flottant 0.500, sous la forme de fraction 1/2**
   abcd
   **** flottant 1.400, sous la forme de fraction 7/5**
   xyz
   **** flottant 6.000, sous la forme de fraction 6/1**

   (voir la fonction `strtof`, et pour plus tard, la fonction `snprintf`).

2. **(Pour DM01)** Partant de la pile d'entiers présentée/donnée en semaine 2, écrire une bibliothèque permettant de gérer une (unique) pile de rationnels : utiliser comme base-projet l'exemple `ratio` fourni sur la page du cours ; nommer les fichiers `rpile.c`, `rpile.h` et nommer les fonctions de pile de rationnels `rpush`, `rpop` et `rempty` ; faire en sorte de tester l'ensemble.
3. Écrire une structure (la bibliothèque) de pile stockant des `int` et dont la taille est dynamique ;
4. Donner la possibilité de gérer plusieurs piles en parallèle ;
5. **(Nécessite qu'on ait déjà abordé la généricité)** Réfléchir à la généricité (quelque soit le type de donnée empilé) de la bibliothèque et écrire les prototypes (faire la conception).
6. **(Nécessite qu'on ait déjà abordé la généricité)** Réfléchir à un exemple d'usage d'une pile générique ;
7. **(Nécessite qu'on ait déjà abordé la généricité)** Écriture de la pile générique ;
8. Lire et tester le code, fourni sur la page du cours (`test_strcpy.c`), comparant différentes implémentations de `strcpy`. Écrire une nouvelle implémentation utilisant `strlen`, copiant `int` par `int` la majorité de la chaîne puis octet par octet le reste (ce reste est nécessairement inférieur à la taille d'un `int`). Comparer, en utilisant le même système de mesure de temps, votre implémentation à celles déjà présentes dans le code fourni.
9. Tester ce code :

```
1  /* CODE a compiler et executer (Linux) de deux manieres :
   gcc -O0 -fno-stack-protector curiosite.c && ./a.out
   gcc -O0 curiosite.c && ./a.out
   */
   #include <stdio.h>
   #include <stdlib.h>
   #include <string.h>
   static void stack(const char * str);
   int main(void) {
     stack("Hello");
     stack("Hello world!");
     return 0;
   }
   static void stack(const char * str) {
     static char test[1];
     char tmp[10], * ptr = malloc(10 * sizeof *ptr);
     strcpy(tmp, str);
     printf("Avec \"%s\"\n", tmp);
     printf("Curiosite : main %p, stack %p, tmp %p, test %p, ptr %p\n", main, stack, tmp, test, ptr);
     free(ptr);
   }
```

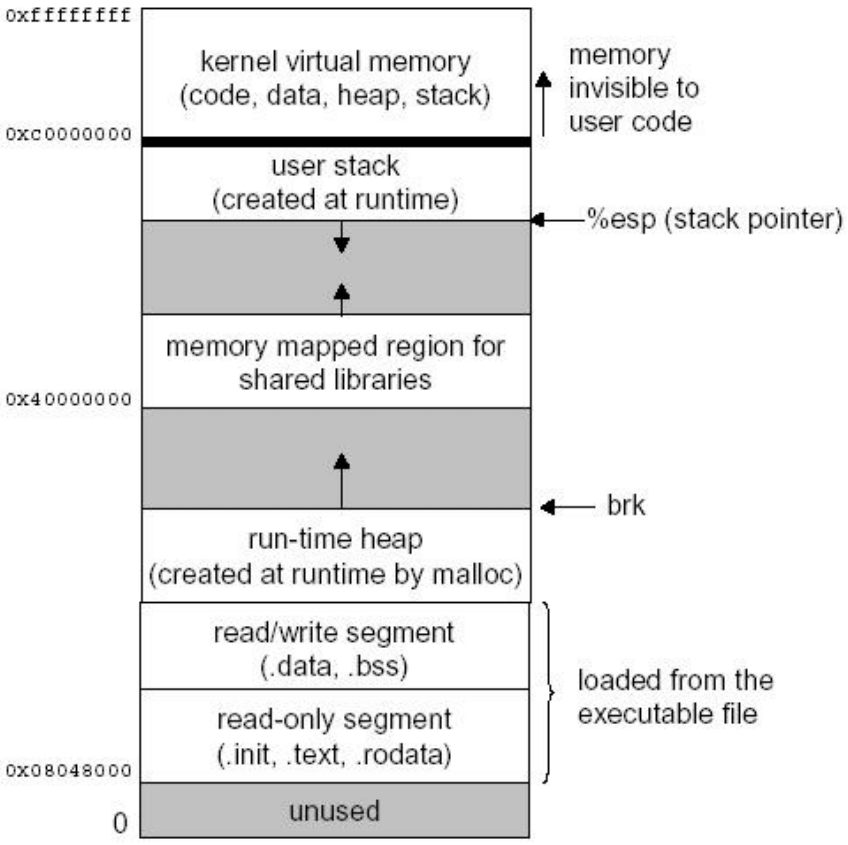

FIGURE 4 – Système Unix : Utilisation de la mémoire par un processus (crédits Monjurul Karim – *Source Code based Buffer Overflow Detection Technology* – 2015).