TD11 : Pile et file

Exercice 1 : Validité du parenthésage d'une expression

Un problème fréquent pour les compilateurs et les traitements de texte est de déterminer si les parenthèses d'une chaîne de caractères sont équilibrées et proprement incluses les unes dans les autres. On désire donc écrire une fonction qui teste la validité du parenthésage d'une expression :

– on considère que les expressions suivantes sont valides : "()", "[(5bonjour+]essai)7plus-1("

– alors que les suivantes ne le sont pas : "(", ")(", "4(essai]".

Notre but est donc d'évaluer la validité d'une expression en ne considérant que ses parenthèses et ses crochets. On suppose que l'expression à tester est dans une chaîne de caractères, dont on peut ignorer tous les caractères autres que '(', '[', ']' et ')'. Écrire en langage algorithmique la fonction valide(ch : chaîne de caractères) : booléen qui retourne vrai si l'expression passée en paramètre est valide, faux sinon.

Exercice 2 : Notation polonaise

La notation polonaise, ou encore notation post-fixée, consiste à faire précéder les opérandes des opérateurs. Par exemple, au lieu d'écrire 42 + 13, en notation polonaise on écrit 42 13 +. Un des avantages de cette notation est qu'elle rend inutile l'usage des parenthèses : pour $3 \times (42 + 13) - 5$, on note 3 42 13 + × 5 − et il n'y a aucune ambiguïté. Dans cet exercice on va définir une fonction `polonaise` qui prendra en paramètres une expression post-fixée sous la forme d'un tableau de chaînes de caractères ({"3","42","13","+","*","5","-"} pour l'exemple précédent) et sa taille, et renverra le résultat (numérique) de l'évaluation de cette expression.

L'algorithme est le suivant. On lit les éléments du tableau un par un et on les empile sur une pile initialement vide. A chaque fois qu'on rencontre un opérateur, plutôt que de l'empiler, on l'applique aux deux derniers éléments de la pile et le résultat remplace ces deux derniers éléments.

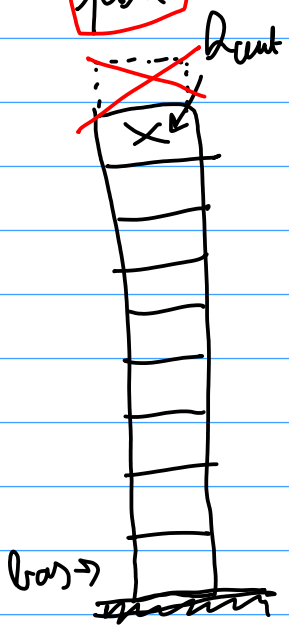
a. Créer une fonction `estOperateur` en C++ qui prend une chaîne de caractères en paramètre (classe string) et retourne vrai si cette chaîne représente un opérateur valide (ici seulement "+", "-", "/" ou "*"), et faux sinon.

b. Créer une fonction `calcul` qui prend en paramètre des chaînes de caractères représentant un opérateur `op` et deux opérandes `a` et `b`, et qui retourne la valeur numérique `a op b`. Vous pourrez utiliser la fonction de conversion suivante : `double atof (const char * c);`

c. En utilisant les deux fonctions précédentes, écrire la fonction `polonaise`. Vous pourrez convertir un réel en une chaîne de caractères en utilisant les instructions suivantes :

```
ostringstream ss; ss << monReel; string maChaine (ss.str());
```

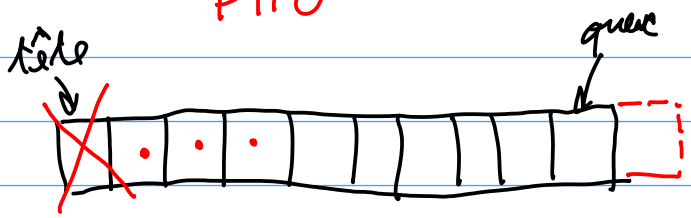
pile LIFO



1 opération de lecture : consulter le haut
2 opérations de modif : — empiler
— dépiler

File

FIFO



1 opération de lecture : consulter la tête
2 opérations de modif : - enfiler
- défiler

TD11 : Pile et file

Exercice 1 : Validité du parenthésage d'une expression

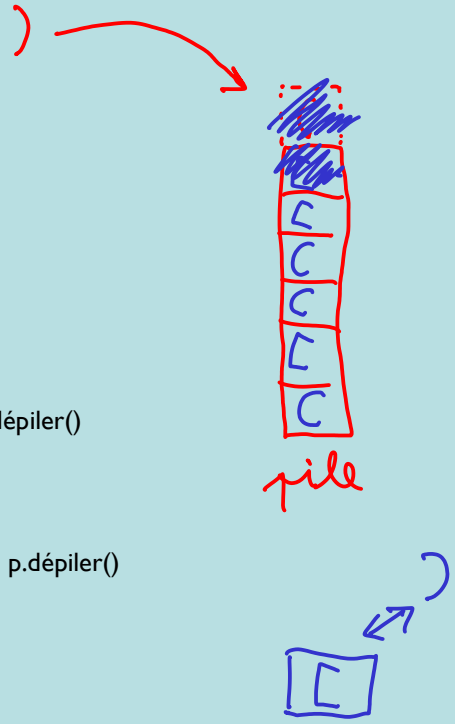
Un problème fréquent pour les compilateurs et les traitements de texte est de déterminer si les parenthèses d'une chaîne de caractères sont équilibrées et proprement incluses les unes dans les autres. On désire donc écrire une fonction qui teste la validité du parenthésage d'une expression :

– on considère que les expressions suivantes sont valides : "()", "[([bonjour+]essai)7plus-];"

– alors que les suivantes ne le sont pas : "(", ")(", "4(essai]".

Notre but est donc d'évaluer la validité d'une expression en ne considérant que ses parenthèses et ses crochets. On suppose que l'expression à tester est dans une chaîne de caractères, dont on peut ignorer tous les caractères autres que '(', '[', ']' et ')'. Écrire en langage algorithmique la fonction valide(ch : chaîne de caractères) : booléen qui retourne vrai si l'expression passée en paramètre est valide, faux sinon.

Fonction valide (ch : chaîne de caractères) : booléen
Précondition : ch se termine par un '\0'
Postcondition : aucune
Résultat : vrai si les parenthèses et crochets de ch sont bien équilibrés et correctement inclus les uns dans les autres, faux sinon
Paramètre en mode donnée : ch
Variables locales :
    p : Pile de caractères
    i : entier
Début
    i ← 1 {*rappel : en algo les indices commencent à 1*}
    Tant que (ch[i] ≠ '\0') Faire
        Si ch[i] = '(' ou ch[i] = '[' Alors
            p.empiler(ch[i])
        Sinon
            Si ch[i] = ')' Alors
                Si non(p.estVide()) et p.consulterSommet() = '(' Alors p.dépiler()
                Sinon retourner faux ; FinSi
            Sinon
                Si ch[i] = ']' Alors
                    Si non(p.estVide()) et p.consulterSommet() = '[' Alors p.dépiler()
                    Sinon retourner faux ; FinSi
                FinSi
            FinSi
        FinSi
        i ← i +1
    FinTantQue
    retourner p.estVide()
Fin valide



Exercice 2 : Notation polonaise

La notation polonaise, ou encore notation post-fixée, consiste à faire précéder les opérandes des opérateurs. Par exemple, au lieu d'écrire 42 + 13, en notation polonaise on écrit 42 13 +. Un des avantages de cette notation est qu'elle rend inutile l'usage des parenthèses : pour $3 \times (42 + 13) - 5$, on note 3 42 13 + × 5 − et il n'y a aucune ambiguïté. Dans cet exercice on va définir une fonction `polonaise` qui prendra en paramètres une expression post-fixée sous la forme d'un tableau de

chaînes de caractères ({"3","42","13","+","*","5","-"} pour l'exemple précédent) et sa taille, et renverra le résultat (numérique) de l'évaluation de cette expression.

L'algorithme est le suivant. On lit les éléments du tableau un par un et on les empile sur une pile initialement vide. A chaque fois qu'on rencontre un opérateur, plutôt que de l'empiler, on l'applique aux deux derniers éléments de la pile et le résultat remplace ces deux derniers éléments.

a. Créer une fonction `estOperateur` en C++ qui prend une chaîne de caractères en paramètre (classe string) et retourne vrai si cette chaîne représente un opérateur valide (ici seulement "+", "-", "/" ou "*"), et faux sinon.

```
bool estOperateur (const string & chaine) {
   return chaine.compare("+") == 0 || chaine.compare("-") == 0 || chaine.compare("/")
== 0 || chaine.compare("*") == 0;
}
```

b. Créer une fonction `calcul` qui prend en paramètre des chaînes de caractères représentant un opérateur `op` et deux opérandes `a` et `b`, et qui retourne la valeur numérique `a op b`. Vous pourrez utiliser la fonction de conversion suivante : **double** `atof` (**const** **char** * c);

```
double calcul (const string & op, const string & a, const string & b) {
    if (op.compare("+")==0)
        return atof(a.c_str()) + atof(b.c_str());
    if (op.compare("-")==0)
        return atof(a.c_str()) - atof(b.c_str());
    if (op.compare("/")==0 && atof(b.c_str())!=0.0)
        return atof(a.c_str()) / atof(b.c_str());
    if (op.compare("*")==0)
        return atof(a.c_str()) * atof(b.c_str());
    return 0.0; // division par zéro ou opérateur non reconnu
}
```

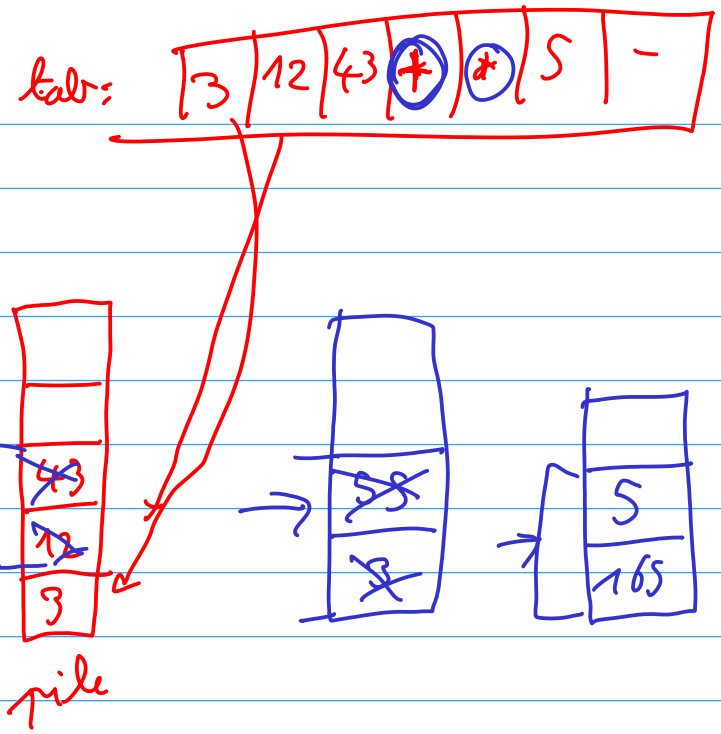
c. En utilisant les deux fonctions précédentes, écrire la fonction `polonaise`. Vous pourrez convertir un réel en une chaîne de caractères en utilisant les instructions suivantes :

```
ostringstream ss; ss << monReel; string maChaine (ss.str());
```

On crée une pile vide au départ, puis on traite les caractères trouvés dans l'expression post-fixée rentrée en paramètre de la façon suivante. Quand on arrive sur un opérateur `op`, on dépile les deux éléments `a` et `b` du sommet de la pile, on calcule `a op b` et on empile le résultat. Quand on arrive sur un nombre (tout ce qui n'est pas un opérateur ici), on l'empile. Lorsqu'on a fini de parcourir l'expression, la pile contient un seul élément qui est la valeur de l'expression que l'on retourne.

```
double polonaise (const string * tab, unsigned int taille) {
    Pile p;
    for (unsigned int i = 0; i < taille; i++) {
        if (estOperateur(tab[i])) {
            string b = p.consulterSommet();
            p.depiler();
            string a = p.consulterSommet();
            p.depiler();

            ostringstream ss;
            ss << calcul(tab[i],a,b);
            string res (ss.str());
            p.empiler(res);
        }
        else {
            string nombre (tab[i]);
            p.empiler(nombre);
        }
    }
    return atof(p.consulterSommet().c_str());
}
```


tab:
3
12
43
+
*
5
-
43
12
3
pile
55
3
5
165