Programmation Avancée

TP n°7 : Les chaînes de caractères

Simon Forest

18 mars 2021

Exercice 1 : Caractères et C-chaînes

Dans cet exercice, on se familiarise avec l'encodage des caractères ainsi que les C-chaînes. Pour simplifier, on pourra supposer que les caractères manipulés sont dans la table ASCII.

1. Écrire une fonction `int est_chiffre(char c)` qui renvoie `1` si `c` est un chiffre et `0` sinon.

2. Écrire une fonction `int est_lettre(char c)` qui renvoie `1` si `c` est une lettre entre `'a'` et `'z'` ou entre `'A'` et `'Z'`, et `0` sinon.

3. Écrire une fonction `void inverser_cap(char *str)` qui inverse la capitalisation des lettres de `str`. Par exemple, si `str` pointe sur `"AabC3"`, un appel à `inverser_cap(str)` devra changer le contenu de `str` en `"aABc3"`.

4. Écrire une fonction `void my_strcpy(char *dest,const char *src)` qui recopie la C-chaînes contenue à `src` dans `dest`. Pour cela, il ne faudra pas utiliser `strcpy`.

5. Écrire une fonction `char* my_strdup(const char *str)` qui produit une copie de la C-chaîne `str`. Pour cela, il ne faudra pas utiliser `strdup`.

6. Écrire une fonction `int my_atoi(const char* str)` qui convertit une C-chaîne constituée de chiffres en l'entier associé. Par exemple, appeler `my_atoi` sur `"3421"` doit retourner l'entier 3421.

7. Tester les fonctions écrites sur quelques exemples à chaque fois.

Exercice 2 : Dictionnaires

Dans cet exercice, on implémente une structure de « dictionnaire », permettant de maintenir un nombre fini d'associations entre des chaînes de caractères et des `int`. Pour cela, on va utiliser les structures suivantes :

```
typedef struct {
  char *cle;
  int valeur;
} assoc;
typedef dico {
  int taille;
  assoc *tab;
};
```

1. Écrire une fonction `void dico_init(dico *ptr)` qui initialise un dictionnaire vide.

2. Écrire une fonction `void dico_ajouter(dico *ptr,char *cle,int valeur)` ajoute une association au dictionnaire. Si une association avec la même clé que `cle` existe déjà dans le dictionnaire, il faudra simplement mettre à jour la valeur. Sinon, il faudra ajouter l'association à `ptr->tab`.

3. Écrire une fonction `int dico_trouver(dico *ptr,char *cle,int *res)` qui renvoie 1 si `cle` est une clé présente dans le dictionnaire et 0 sinon. Dans le cas où 1 est renvoyé et que `res` n'est pas nul, on mettra dans `*res` la valeur associée à la clé dans le dictionnaire.
4. Écrire une fonction `void dico_detruire(dico *ptr)` qui libère le contenu du dictionnaire.
5. Tester vos fonctions sur plusieurs exemples.

Exercice 3 : Arbres arithmétiques

Dans cet exercice, on écrit des fonctions permettant de travailler avec des arbres arithmétiques comme sur la Figure 1. Pour cela, on va utiliser utiliser un type `noeud` pour représenter les nœuds de l'arbre, défini par

```
typedef struct noeud {
  int type;
  union {
    int constante;
    char *variable;
    struct operateur_bin op;
  } u;
} noeud;
```

où `operateur_bin` est défini par

```
typedef struct operateur_bin {
  int id;
  struct noeud *gauche, *droite;
} operateur_bin;
```

(en pratique, il faut mettre la définition de `operateur_bin` avant celle de `noeud` et précéder le tout de `struct noeud;`).

Pour les `noeud`, on prendra la convention que `type == 0` dans le cas d'une `constante`, `type == 1` dans le cas d'une variable et `type == 2` dans le cas d'un `op`. Pour les `operateur_bin`, on prendra la convention que `id == 0` correspond à l'addition, et `id == 1` correspond à la multiplication.

1. Écrire une fonction `noeud* noeud_constante(int v)` qui produit un `noeud` contenant la constante `v`.
2. Écrire une fonction `noeud* noeud_variable(char *nom)` qui produit un `noeud` contenant une variable nommée `nom`.
3. Écrire une fonction `noeud* noeud_op(int id,noeud *gauche,noeud *droite)` qui produit un arbre appliquant une opération de type `id` aux sous-arbres `gauche` et `droite`.

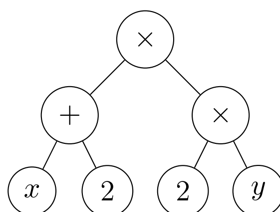

FIGURE 1 – Arbre arithmétique représentant l'expression $(x + 2) \times (2 \times y)$

4. Écrire une fonction `void noeud_detruire(noeud *arb)` qui désalloue complètement un arbre `arb`.
5. Écrire une fonction `int noeud_evaluer(noeud *arb,dico *d)` qui évalue un arbre arithmétique en utilisant pour les variables les valeurs stockées dans le dictionnaire `d`. Tester cette fonction sur des exemples.
6. Écrire une fonction `void noeud_infixe(noeud *arb)` qui affiche un arbre en notation infixe habituelle (il faudra ajouter des parenthèses pour que l'expression reste correcte).
7. Écrire une fonction `void noeud_postfixe(noeud *arb)` qui affiche un arbre en notation postfixe (notation polonaise).
8. Écrire une fonction `noeud* noeud_lire()` qui lit un arbre écrit sur l'entrée en notation polonaise.
9. Tester ces dernières fonctions sur des exemples. Vérifier avec `valgrind` que la mémoire est libérée correctement.