Programmation Impérative - TP0

Rohan Fossé - Léo Mendiboure - Guillaume Mercier
{rohan.fosse,leo.mendiboure}@labri.fr, mercier@enseirb-matmeca.fr

2019-2020

1 Manipulation, observation, diagnostic

1.1 Initialisation de chaînes de caractères

1. Essayez de compiler le programme `string_init.c` qui se trouve ici :
   `/net/ens/mercier/PG109/TP0/string_init.c`
   Quelle est l'erreur produite par le compilateur ?
2. Commentez la ligne fautive, recompilez et exécutez le code. Que se passe-t-il ? Expliquez le phénomène.
3. Commentez les lignes fautives de façon à avoir un programme qui s'exécute normalement (ie sans erreur de segmentation) : que pouvez-vous en conclure concernant l'initialisation de chaîne ?

1.2 Déclaration de variables et localité

Rappel : les variables possèdent 6 propriétés/caractéristiques.

1. un identifiant (un nom) ;
2. un type ;
3. une valeur ;
4. une portée ;
5. une visibilité ;
6. une durée de vie.

Le but de cet exercice est de comprendre les règles et mécanismes qui permettent de «jouer» avec les trois dernières de ces propriétés.

1.2.1 Masquage de variables

Compilez et exécutez le fichier `/net/ens/mercier/PG109/TP0/local_mask.c`
L'affichage est-il conforme à vos attentes ? Expliquez.

1.2.2 Variables statiques locales

Compilez et exécutez le fichier `/net/ens/mercier/PG109/TP0/static_local.c`
L'affichage est-il conforme à vos attentes ? Expliquez.

1.2.3 Variables statiques globales

Récupérez *tous* les fichiers sources qui se trouvent dans le répertoire suivant :
`/net/ens/mercier/PG109/TP0/Static`.

- compilez les fichiers `static2.c` et `static3.c` de façon à obtenir un *fichier objet* et non pas un programme *exécutable* (avec la commande `gcc -c fichierX.c`)
- compilez le fichier `static.c` de façon à obtenir un programme exécutable (`gcc -o fichier fichier.c fichier2.o fichier3.o`) et lancez-le. Quel est le résultat obtenu ? Que pouvez-vous en déduire de l'utilisation des mots-clefs `extern` et `static` dans le cas de variables globales ?

— Enlevez le mot-clef `static` dans le fichier `static2.c` et mettez-le dans le fichier `static3.c` (au même endroit). Recompilez les trois fichiers comme précédemment et relancez le programme. Est-ce que l'affichage est conforme à vos attentes ?

2 Rappels sur les pointeurs

Les pointeurs sont des variables qui véhiculent deux informations :
— une informations d'*adresse*. Une adresse est un numéro de case mémoire et une case mémoire contient un *octet* (ang. *byte*) c'est-à-dire 8 *bits* ;
— une information de taille, c'est à dire le nombre de cases mémoire à récupérer en cas de *déréférencement* du pointeur, c'est-à-dire en cas d'utilisation de l'opérateur unaire `*`.

Comme les pointeurs sont des variables, toutes les règles concernant ces dernières s'appliquent, en terme de paramètre pour une fonction, de portée, visibilité, durée de vie, etc.

Il existe deux types d'opérations avec les pointeurs :
— l'opérateur unaire `*` qui permet de *déréférencer* un pointeur, c'est-à-dire de récupérer un ensemble de cases mémoire pointées par le pointeur. Ces cases mémoire contiennent en fait une *variable*, dont le type est déterminé par le type du pointeur. Ainsi, il est impossible de déréférencer un pointeur *générique* de type `void` ;
— l'opérateur unaire `&` qui permet de récupérer un *pointeur* à partir d'une variable. Par abus de langage, on dit que l'on récupère *l'adresse* de cette variable, mais n'oubliez pas que l'information de taille est également récupérée.

Il est possible de faire de *l'arithmétique pointeur*, c'est-à-dire de calculer une nouvelle adresse à partir d'une autre adresse, afin de lire ou d'écrire à la nouvelle adresse produite. De façon générale, il s'agit d'une opération comme :

$$ptr2 = ptr1 + val;$$

Où *ptr1* et *ptr2* sont des pointeur et *val* est une valeur entière. D'après cette formule, on déduit qu'*additionner une valeur à un pointeur permet d'obtenir un nouveau pointeur*. D'après cette première formule, on peut également déduire la nouvelle formule suivante :

$$val = ptr2 - ptr1;$$

Donc, la différence de deux pointeurs est une valeur entière (un nombre d'éléments en fait) et non pas un pointeur. La différence de deux pointeurs est même un type particulier : `ptrdiff_t` (mais que vous n'utiliserez que rarement *a priori*[1]).

2.1 Exercice d'observation et de déduction

Compilez le fichier `pointeur.c` disponible dans le répertoire `/net/ens/mercier/PG109/TP0/`.
— Exécutez ce programme. Son comportement est-il conforme à vos attentes ?
— Quelle règle pouvez-vous déduire concernant l'addition pour l'arithmétique pointeur ?
— Quelle conséquence cette règle a-t-elle pour les pointeurs génériques (de type `void`) ?

2.2 Pointeurs de fonctions

Les pointeurs sont des variables qui peuvent contenir non seulement l'adresse d'une variable mais également l'adresse d'une *fonction*. En `C`, le *nom d'une fonction est un pointeur sur cette fonction* et il n'est donc pas nécessaire de recourir aux opérateurs `*` et`&` (mais cela reste possible, bien entendu).

Exercice 1 : Map/Reduce Vous allez implémenter deux types d'opérations :
— Une opération `Map`, qui prend un vecteur d'éléments et une fonction et qui «applique» cette fonction à chaque élément du vecteur. `Map` renvoie le nouveau vecteur ainsi produit. Le prototype de `Map` sera donc du style :

1. Faire un `#include <stddef.h>` pour utiliser ce type.

```
int *Map(int tab[], int size, int (*map_func)(int));
```

Question subsidiaire : à quoi sert le paramètre `size` ?

— Une opération `Reduce` qui prend un vecteur d'éléments et une fonction et qui effectue une opération de «réduction» qui combine tous les éléments du tableau en une seule valeur et qui renvoie le résultat produit. Le prototype de `Reduce` sera donc du style :

```
int Reduce(int tab[], int size, int (*reduce_func)(int,int));
```

Par exemple si l'opération de réduction est l'addition, le résultat sera la somme de tous les éléments du tableau.

— Récupérez les fichiers `mapreduce.c` `mystere.o` qui se trouvent ici : `/net/ens/mercier/PG109/TP0/` ;

— Modifiez le fichier `mapreduce.c` pour implémenter les fonctions `Map`, `Reduce` et testez-les dans le `main` du programme ;

— Compilez votre programme de façon à utiliser les fonctions `map_func_mystere` et `reduce_func_mystere` avec la commande :

```
gcc -o mapreduce mapreduce.c mystere.o
```

et exécutez-le. Que se passe-t-il ? Que font les fonctions `map_func_mystere` et `reduce_func_mystere` ?

— Implémentez vos propres fonctions `map_func_mystere` et `reduce_func_mystere` afin de vérifier que votre implémentation de `Map` et `Reduce` est correcte.

Exercice 2 : Map/Reduce (part deux) Effectuez le même travail mais en utilisant les fichiers `mapreduce2.c` et `mystere2.o`. Dans ce cas, vous allez travailler avec des *chaînes de caractères* et non plus des entiers.

3 Cryptage avec des unions

Soient les définitions suivantes :

```
#define MSG_MAX_SIZE 35

union MessageCrypte{
  int  a[MSG_MAX_SIZE];
  char b[MSG_MAX_SIZE*4];
};
```

3.1 Premier message

Je me suis servi de cette union afin de crypter un message (d'au plus 140 caractères) dont vous allez devoir retrouver le sens. Le message est contenu dans le fichier `msg_a_decoder.txt` (dans le répertoire `/net/ens/mercier/PG109/TP0/`).

En fait, le décryptage est facile : il suffit de récupérer les entiers qui sont dans le fichier `msg_a_decoder.txt` [2] , et les stocker dans une variable de type `union MessageCrypte` (en utilisant le champ `a`) avant d'afficher le tout sous forme de caractères (en utilisant le champ `b` cette fois-ci). Vous pouvez lire le contenu du fichier en lisant l'entrée standard et en faisant une redirection de cette dernière comme ceci :

```
./decrypt < msg_a_decoder.txt
```

Questions :

— Que dit le message ?

— Quelle est la taille de la variable de type `union MessageCrypte` ?

— Que pouvez-vous en déduire sur la façon dont elle est stockée en mémoire ?

3.2 Cryptage/décryptage

Maintenant que vous avez compris le fonctionnement, vous allez mettre en place le programme qui permet d'effectuer le cryptage d'un message. Ce message sera stocké dans un fichier qui sera passé en argument au programme `crypt` qui produira un nouveau fichier contenant le résultat du cryptage :

2. Mais qui y sont stockés sous forme de *chaînes de caractères*

```
./crypt < fichier_a_crypter.txt > fichier_resultat.txt
```

Le cryptage est l'inverse du décryptage : vous allez récupérer les caractères du message d'origine et les stocker dans une variable de type `union MessageCrypte` (en utilisant le champ `b`). Ensuite, vous allez afficher le contenu de cette union mais sous forme d'*entiers* (en utilisant le champ `a`) ;

3.3 Megamix

Maintenant que vous avez les opérations de cryptage/décryptage de base qui sont opérationnelles, vous pouvez renforcer le cryptage/décryptage en utilisant :

1. D'abord une opération de type `Map` sur les entiers résultant de l'opération de cryptage initiale (cf partie précédente de l'exercice). Par exemple, la fonction à appliquer sur le vecteur d'entiers pourra prendre une clef (par exemple le premier entier du vecteur) et consistera à calculer la différence entre cette clef et chaque élément du vecteur).
2. Ensuite, ces nouveaux entiers seront manipulés sous forme de chaînes de caractères (rour rappel, quand vous affichez un entier à l'écran, il l'est sous forme de chaînes de caractères) et une opération de réduction `Reduce` pour leur être à son tour appliquée. Par exemple, une opération de concaténation avec un séparateur non numérique.

4 Exercice (pour se reposer un peu)

Vous allez maintenant écrire un programme permettant de jouer au *mastermind*. Le programme génère une chaîne de caractère représentant la séquence des six couleurs que le joueur doit deviner.

Exemple : R B V J 0 N
où R = rouge, B = bleu, V = vert , J = jaune , O = orange , N = Noir.

Cette séquence peut être générée aléatoirement avec la fonction `srand`.

Ensuite, le programme demande en combien de coups la partie doit se jouer : la partie sera donc finie si le joueur trouve la bonne combinaison en le nombre de coups impartis ou bien quand ce nombre est atteint. Le programme va consister ensuite à demander au joueur de rentrer une chaîne de caractères, la comparer à la solution et à renvoyer une information concernant la chaîne rentrée par le joueur : le programme répond alors B pour une couleur bien placée et N si la couleur est présente mais mal placée (il peut donc répondre avec une chaîne plus petite que la taille totale de la chaîne à deviner, soit 6).

Exemple : si la solution est B R R B J V et que le joueur répond B V B O N V, le programme donnera comme information : B B N (il y a deux éléments bien placés et un mal placé)