Université de Lille 1 SCIENCES ET TECHNOLOGIES

Licence d'informatique
2e année
Module de BPC

Bases de la Programmation en C. Thème 3.

Équipe pédagogique de BPC [1]

février 2021

Ce sujet est disponible en version html à l'adresse suivante :
`https://www.fil.univ-lille1.fr/~ballabriga/bpc/tdtp/tp_theme3.html`.

Table des matières

1. d'après un document CC-BY-SA de Gilles GRIMAUD et Philippe MARQUET, 2018-2020.

Ce support sert à la fois pour les TDs et les TPs. Sauf mention contraire, chaque exercice est à préparer en TD, et ensuite à implémenter, compiler et tester en séance de TP.

Semaine 5 : Les types composés (1/2)

Pour ce thème 3 du module BPC, vous allez travailler sur un autre dépot git :

`https://gitlab-etu.fil.univ-lille1.fr/ls4-pdc/theme3-g<i>-y<yy>`

où *<i>* est votre numéro de groupe, et *<yy>* est l'année.

1 Exercices sur les tableaux

Dans cette partie du sujet, vous réaliserez quelques exercices préparatoires sur les tableaux et chaines de caractères.

Exercice 1 (Copie de tableau)
Soient les tableaux définis par :

```
#define SIZE            12
int tsrc[SIZE], tdest[SIZE];
```

- Donnez le code C permettant d'affecter à `tdst` les éléments de `tsrc`.
- Donnez le code C permettant d'affecter à `tdst` seulement les éléments strictement positifs de `tsrc` (on complètera la fin du tableau destination par des valeurs nulles).

Exercice 2 (Fonction de copie d'entiers)
- Donnez le prototype d'une fonction pour copier les valeurs d'un tableau d'entiers source dans un second tableau destination.
- Donnez la définition de cette fonction de recopie de tableaux.

Exercice 3 (Fonction de copie d'entiers)
- Donnez le prototype d'une fonction pour copier les valeurs d'une chaine de caractères source dans une chaine de caractères destination.
- Donnez la définition de cette fonction de recopie de chaines de caractères.

2 Mes Commandes Unix

Dans cette partie du sujet, vous allez implémenter une version simplifiée de quelques commandes Unix existantes. Pour cela, vous allez d'abord réaliser quelques fonctions communes, et ensuite vous travaillerez sur l'implémentation des commandes proprement dites.

Tout au long de ce travail, vous allez devoir utiliser les Makefile et la compilation modulaire.

Nous travaillerons dans le répertoire `mcu/`, *mes commandes Unix*, de notre dépôt.

Un filtre est une commande qui lit un texte à traiter sur son entre standard (`stdin`), produit son résultat sur la sortie standard (`stdout`), et produit éventuellement des messages d'erreur — dans l'idéal sur la sortie d'erreur (`stderr`). De plus, un filtre renvoie 0 s'il se termine correctement et un code d'erreur dans le cas contraire.

Pour faire simple, nous n'allons implanter que des filtres ne prenant aucune option et qui traitent l'entrée standard ligne par ligne. On convient que nos versions basiques de ces commandes considèrent que les lignes font au plus 80 caractères. Si l'entrée standard fournie contient une ligne de plus de 80 caractères, le filtre se termine sur un échec et le code d'erreur retourné est 1.

Nous allons implanter des versions simplifiées suivantes de commandes UNIX standard :

- `mcu_wc` qui affiche sur la sortie standard le nombre de caractères — octets — composant l'entrée standard, notre propre version de `wc -c`, *word count* ;
- `mcu_wl` qui affiche le nombre de lignes lues sur l'entrée standard (une ligne se termine par le caractère `'\n'`), notre propre version de `wc -l`;

— `mcu_rev` qui renverse l'ordre des caractères pour chaque ligne ; notre propre version de la commande `rev`. Par exemple :

```
bash$ echo "la vie est belle" | mcu_rev
elleb tse eiv al
bash$
```

— `mcu_uniq` qui reproduit sur la sortie standard chaque ligne lues depuis l'entrée standard sauf si elle fait doublon avec la ligne précédente ; une version simplifiée de la commande `uniq`. Par exemple, on a :

```
bash$ echo "Hello" > essai
bash$ for((i=0;i<5;i++));do echo "Hello World" >> essai ;done;
bash$ mcu_uniq < essai
Hello
Hello World
bash$
```

La compilation et la mise à jour des exécutables en fonction des évolutions du code se fera en utilisant la commande `make`. Vous devez donc construire le `Makefile` correspondant au fur et à mesure des exercices.

2.1 Fonctions partagées

Nous allons commencer par coder des fonctions qui vont être utilisées dans plusieurs exécutables que nous devons implanter. Pour décrire ces fonctions, nous indiquons ci-dessous une partie des fichiers d'entêtes contenant leurs prototypes et des macros utiles :

— `src/mcu_macros.h`
— `src/mcu_fatal.h`
— `src/mcu_readl.h`
— `src/mcu_putint.h`

Exercice 4 (Implantation des fonctions partagées)
Travail à réaliser en TD :

Vous préparerez en TD les fonctions `fatal` et `readl` de prototypes suivants :

```
void fatal(int assert, const char message[], int status);
int readl(char line[]);
```

La fonction `fatal` quitte le programme avec un code de retour non-nul après avoir affiché un message, si jamais l'argument `assert` correspond à une valeur vraie (c-a-d différente de 0). Pour quitter le programme vous utiliserez la fonction `exit` de la librairie standard, et vous utiliserez une boucle avec `putchar` pour afficher le message.

La fonction `readl` lit une ligne (terminée par un retour à la ligne) sur l'entrée standard, et remplit le tableau `line` avec son contenu (y compris le retour à la ligne ainsi que le 0 terminateur). La fonction renvoie le nombre de caractères lus, ou `EOF` si la fin du fichier est atteinte avant la fin de la ligne. On considère que le la taille du tableau `line` est de `MAXLINE`. La fonction terminera le programme (à l'aide de la fonction `fatal`) si jamais le nombre de caractères lus (en comptant le 0 terminateur) est supérieur à `MAXLINE`.

Pour lire les caractères sur l'entrée standard, vous utiliserez la fonction `getchar`, qui renvoie le code ASCII du prochain caractère lu, ou `EOF` en cas de fin du fichier.

Travail à réaliser en TP :

Implanter ces fonctions et assurez vous de leurs bon fonctionnement. Prenez soin à ce que chaque fonction soit dans un fichier source différent et que votre code fonctionne avec le fichier de test `src/mcu_test.c` pour produire le résultat suivant :

```
bash$ ./build/mcu_test < src/mcu_test.c ; echo "le code de retour est $?"
#include <stdio.h>
18
1==0 is not true
le code de retour est 2
bash$
```

après la compilation :

```
bash$ make mcu_test
gcc -Wall -Werror -ansi -pedantic -c src/mcu_putint.c
gcc -Wall -Werror -ansi -pedantic -c src/mcu_fatal.c
gcc -Wall -Werror -ansi -pedantic -c src/mcu_readl.c
gcc -Wall -Werror -ansi -pedantic -c src/mcu_test.c
gcc -Wall -Werror -ansi -pedantic -o build/mcu_test src/mcu_putint.o src/mcu_fata
bash$
```

□

Vous noterez que, dans les lignes de commandes de compilation, à chaque fois qu'il est attendu un nom de fichier, vous avez le droit de le préfixer par un `nomderepertoire/` pour indiquer que votre fichier est dans un répertoire donné. Vous pourrez utiliser ceci pour gérer des projets pour lesquels les divers fichiers doivent être dans des répertoires différents (ceci fonctionne aussi à l'interieur des Makefile)

2.2 Les commandes UNIX proprement dites

Exercice 5 (Commandes de comptage)
Travail à réaliser en TD :

Réalisez une fonction qui lit, avec `getchar`, des caractères jusqu'à rencontrer la fin du fichier (vous n'avez pas besoin de `readl`).

Faites ensuite une version de cette fonction qui compte le nombre de caractères lus, puis une autre version qui compte le nombre de retours à la ligne lus.

Travail à réaliser en TP :

En utilisant les fonctions préparées en TD, réalisez les filtres `mcu_wc` (comptage des caractères) et `mcu_wl` (comptage des retours à la ligne), et testez les :

Les filtres `mcu_wc` et `mcu_wl` doivent fournir les résultats suivants sur ces exemples :

```
bash$ echo 'Hello Unix World!' | ./build/mcu_wc
18
bash$ echo 'Hello\nunix\nWorld!' | ./build/mcu_wl
3
bash$
```

Exercice 6 (commande `mcu_rev`)
Nous allons maintenant utiliser la fonction `readl()` de notre librairie pour implanter le filtre `mcu_rev`. Vous devriez obtenir ce genre de résultat.

```
bash$ echo "Hello World," > essai
bash$ echo "Hello Unix World!" >> essai
bash$ ./build/mcu_rev < essai
,dlrow olleH
!dlroW xinU olleH
bash$
```

Travail à réaliser en TD :

Réalisez une fonction `void rev(char tab[])` acceptant un tableau (de taille `MAXLINE`) représentant une chaîne de caractères, et qui renverse l'ordre des caractères dans cette chaîne. Attention : prenez en compte le fait que la chaîne de caractères est peut-être plus petite que le tableau...

Travail à réaliser en TP :

En utilisant la fonction préparée en TD et `readl`, implantez le filtre `mcu_rev`. □

Exercice 7 (commande `uniq`)
Travail à réaliser en TD :

Préparez une fonction `copier(char cible[], char source[])` qui copie la chaîne de caractères représentée par `source` vers `cible`.

Que se passe-t'il si l'appelant utilise cette fonction alors que le tableau `cible` est trop petit pour contenir la chaîne stockée dans `source` ? Lorsque vous appellerez cette fonction (en TP), vous vous assurerez donc que ceci ne se produit pas.

Préparez une fonction `comparer(char chaine1[], char chaine2[])` qui compare les deux chaînes passées en paramètre, et renvoie 0 si elles sont identiques, ou 1 dans le cas contraire. Quand vous appelerez cette fonction en TP, vous prendrez les précautions nécessaires, comme avec la fonction `copier`.

Travail à réaliser en TP :

Implanter le filtre `mcu_uniq` décrit précédemment. Vous allez utiliser `readl`, ainsi que les deux fonctions préparées en TD. □

Semaine 6 : Les types composés (2/2)

3 Lecture et écriture de données binaires sur l'entrée sortie standard

Dans cette partie, vous allez vous familiariser avec la représentation mémoire de divers types (entiers, floats, structures...) En effet, chaque variable en C peut être représentée comme une suite d'octets. Vous allez écrire un programme permettant d'exprimer une structure sous la forme d'une suite d'octets, et d'écrire cette suite d'octets vers la sortie standard.

Ensuite, vous écrirez un programme réalisant l'opération inverse, c'est-à-dire lisant une série d'octets sur l'entrée standard, et reconstituant le contenu d'une ou plusieurs structures.

Vous travaillerez dans le sous-répertoire `representation/` qui se trouve dans votre dépot. Certains fichiers à compléter sont fournis (sources, Makefile...)

3.1 Lecture : les unions

Les unions se déclarent de manière similaire à une structure (excepté le fait qu'on utilise le mot clef `union` au lieu de `struct` mais ont différence fondamentale : tous les membres d'une même union occupent le même espace dans la mémoire : tous les membres sont en quelque sorte "superposés". Le code et la figure ci-dessous explique ce qu'il se passe :

```
struct personne_s {
        int numero;
        char nom[8];
        char age;
};
union monunion_s {
struct personne_s pers;
char data[13];
};
union monunion_s monUnion;
monUnion.pers.numero = 0x11223344;
strcpy(monUnion.pers.nom, "DUPONT");
maPersonne.pers.age = 43;
```

Ici, dans la variable `monUnion`, le champ `monUnion.pers` et `monUnion.data` occupent exactement le même espace dans la mémoire. La figure ci-dessous schématise cette superposition :

Adresses	0	1	2	3	4	5	6	7	8	9	10	11	12
Champ dans pers	numero				nom								age
monUnion.pers	0x11223344				'D'	'U'	'P'	'O'	'N'	'T'	0	...	43
Index dans data	0	1	2	3	4	5	6	7	8	9	10	11	12
monUnion.data	0x44	0x33	0x22	0x11	'D'	'U'	'P'	'O'	'N'	'T'	0	...	43

Ainsi, écrire dans `monUnion.pers.numero` revient à écrire dans `monUnion.data[0]` à `monUnion.data[3]` (sur une machine little-endian, `monUnion.data[0]` contiendra l'octet de poids faible du numéro, et ainsi de suite), écrire dans `monUnion.pers.nom[0]` revient à écrire dans `monUnion.data[4]`, écrire dans `monUnion.pers.age` revient à écrire dans `monUnion.data[12]`, et ainsi de suite.

Le champ `monUnion.data` permet donc d'accéder (en lecture ou écriture) à la représentation de notre structure `struct personne_s` sous la forme d'une suite d'octets.

3.2 Travail à réaliser

Vous trouverez les fichiers nécessaires dans votre dépot gitlab. La structure sur laquelle vous allez travailler représente un étudiant, qui possède un numéro d'étudiant (int), et une moyenne générale (float). Cette structure est définie dans `commun.h`.

Vous avez aussi un squelette pour les deux programmes, `output.c` et `input.c`, permettant respectivement d'écrire et de lire la suite d'octets mentionnée ci-dessus, ainsi qu'un Makefile.

Exercice 8
Exercice sur les unions

Soit le code suivant :

```
union test_u {
  unsigned int val;
  unsigned char data[sizeof(int)];
};
int main() {
  union test_u test;
  int i;
  test.val = 0x41424344;
  for (i = 0; i < sizeof(int); i++) {
    putchar(test.data[i]);
  }
}
```

Travail à réaliser en TD :

Si on considère que le type `unsigned int` est codé sur 32 bits, et qu'on utilise une machine "Little endian" (rappel : cela veut dire qu'on stocke en premier l'octet de poids faible, et en dernier l'octet de poids fort), quel est la représentation de la valeur 0x41424344 ?

Qu'affiche ce programme ? Pourquoi ?

Travail à réaliser en TP :

La commande `hd` vous permet d'avoir une vue en hexadécimal d'un fichier. En TP, compilez ce programme et testez le, en redirigeant sa sortie standard vers un fichier (ex : `./prog > data.bin`) puis ensuite examinez le fichier produit grace à la commande `hd data.bin`. Qu'est ce que vous obtenez ? Pourquoi ?

Exercice 9
Ecriture vers la sortie standard d'une suite d'octets correspondant à une struct etudiant

Travail à réaliser en TD :

Soit les déclarations suivantes (que vous retrouverez dans le fichier `commun.h` de votre dépôt en TP) :

```
struct etudiant_s {
    unsigned int numero;
    float moyenne;
};

union bloc_u {
    struct etudiant_s etu;
    char data[sizeof(struct etudiant_s)];
};

main() {
        struct etudiant_s monEtudiant;
        monEtudiant.numero = 12345;
        monEtudiant.moyenne = 12.58;
```

}

Comment accéder à la suite d'octets qui représente la variable `monEtudiant` ? (indice : inspirez-vous de ce qui a été fait à l'exercice précédent)

Travail à réaliser en TP :

Dans la fonction `main` du fichier `output.c`, déclarez une variable de type `struct etudiant_s` et initialisez la avec les valeurs de votre choix.

Ensuite, en utilisant la méthode décrite en TD, et en regardant l'`union` qui est définie dans `commun.h`, faites une boucle permettant d'écrire, sur la sortie standard, la suite d'octets qui représente votre structure étudiant.

Testez votre programme en redirigeant sa sortie standard vers un fichier, et examinez le fichier avec la commande `hd`. Qu'observez-vous ? Pouvez-vous repérer, dans l'affichage produit par `hd`, les octets qui correspondent au numéro d'étudiant ?

Exercice 10

Lecture d'une sequence d'octets représentant une structure étudiant depuis l'entrée standard Dans la fonction `main` du fichier `input.c`, réalisez l'opération inverse de l'exercice précédent : lisez `sizeof(struct etudiant_s)` octets depuis l'entrée standard avec `getchar\`, et affichez le numéro d'étudiant et la moyenne de l'étudiant correspondant.

Travail à réaliser en TD :

Préparez votre travail en TD, réfléchissez à votre code et préparez le au brouillon...

Travail à réaliser en TP :

Réalisez l'implémentation de ce que vous avez préparé en TD.

Testez votre programme en réalisant une redirection d'entrée standard, pour que votre programme lise le fichier provenant de la sortie standard du programme `output.c`. Vérifiez que les valeurs affichées (numéro d'étudiant et moyenne) sont les mêmes que celles que vous avez spécifiées lors de l'exercice précédent.

Exercice 11

Nom et prénom de l'étudiant Proposez une modification de la structure `struct etudiant_s`, qui permettra de stocker également le nom et prénom de l'étudiant. Modifiez `common.h`, `output.c` et `input.c` en conséquence.

Travail à réaliser en TD :

Préparez votre travail en TD, réfléchissez à votre code et préparez le au brouillon...

Travail à réaliser en TP :

Réalisez l'implémentation de ce que vous avez préparé en TD, et testez son bon fonctionnement.

Exercice 12

Gérer une classe On considère maintenant qu'on gère une classe d'exactement 30 étudiants. Vous pouvez définir une macro `NB_ETU` dans `common.h` pour fixer le nombre d'étudiants gérés.

Réalisez une modification des types définis dans `common.h` pour permettre la gestion d'exactement 30 étudiants au lieu d'un seul. Modifiez également `output.c` et `input.c` en conséquence (vous pouvez utiliser la fonction `rand` pour génerer aléatoirement des notes pour vos 30 étudiants, et utiliser un compteur pour génerer les numéros d'étudiants automatiquement...)

Travail à réaliser en TD :

Préparez votre travail en TD, réfléchissez à votre code et préparez le au brouillon...

Travail à réaliser en TP :

Réalisez l'implémentation de ce que vous avez préparé en TD, et testez son bon fonctionnement.

Exercice 13

Gérer un nombre variable d'étudiants Dans l'exercice précédent, le nombre d'étudiants à écrire/lire est prédéterminé, et fixé à 30. On voudrait maintenant pouvoir gérer un nombre d'étudiants variable, entre 1 et 30 (inclus).

Proposez une méthode permettant de réaliser ceci, puis implémentez et testez la.

Travail à réaliser en TD :

Préparez votre travail en TD, réfléchissez à votre code et préparez le au brouillon...

Travail à réaliser en TP :

Réalisez l'implémentation de ce que vous avez préparé en TD, et testez son bon fonctionnement.