Informatique *Cours*
Programmation impérative avec C - Entrées/Sorties et Fichiers Ipesup

Objectifs

- Afficher des valeurs sur la sortie standard avec `printf`.
- Lire des valeurs depuis l'entrée standard avec `scanf`.
- Manipuler les arguments de la ligne de commande (`argc`/`argv`).
- Ouvrir, lire/écrire et fermer des fichiers (`fopen`, `fscanf`, `fprintf`, `fclose`).

1 Entrées-sorties

1.1 Ligne de commande

Les arguments de la ligne de commande sont transmis à `main` via deux paramètres : un entier `argc` (nombre d'éléments) et un tableau de chaînes `argv`.

```
1 int main(int argc, char **argv) {
2 printf("la somme vaut %d\n", atoi(argv[1]) + atoi(argv[2]));
3 return 0;
4 }
```

`argv[0]` contient le nom de l'exécutable. Exemple de compilation et d'exécution :

```
1 $ gcc somme.c -o somme
2 $ ./somme 34 55
3 la somme vaut 89
```

Gestion de fichiers

Pour manipuler des fichiers, on utilise `fopen`, qui prend deux chaînes : le nom du fichier et son *mode* d'ouverture.

```
1 FILE *f = fopen("data/sudoku.txt", "r");
```

Le mode `"r"` ouvre le fichier en lecture (*read*), `"w"` en écriture (*write*), et `"a"` en ajout (*append*). La variable `f` de type `FILE*` est ensuite utilisée avec `fscanf`, `fprintf` ou `getc`, variantes de `scanf`, `printf` et `getchar` prenant un argument supplémentaire `FILE*`.
Lecture d'un entier :

```
1 int e;
2 fscanf(f, "%d\n", &e);
```

Écriture d'un entier :

```
1 fprintf(f, "%d\n", e);
```

Pour garantir la sauvegarde effective des données, il faut toujours fermer le fichier avec `fclose`, qui libère les ressources et vide les tampons :

```
fclose(f);
```

Sous UNIX, trois fichiers sont ouverts par défaut : `stdin`, `stdout` et `stderr`. On peut écrire sur la sortie d'erreur avec :

```
fprintf(stderr, ...);
```

Ces fichiers sont automatiquement fermés à la fin du programme.

2 Modularité

Lorsqu'un programme C commence à devenir gros, il est intéressant de découper son code en plusieurs fichiers. Par ailleurs, certains de ces fichiers pourront être réutilisés dans d'autres programmes ; on les appelle des bibliothèques.

Supposons ainsi qu'on écrive une partie de notre programme dans un premier fichier, `arith.c`, contenant la définition d'une fonction :

```
int power(int x, int n) { ... }
```

et le reste de notre programme dans un second fichier, `main.c`, qui analyse la ligne de commande, fait des calculs en utilisant la fonction `power` et affiche des résultats. On peut alors compiler notre programme en passant ces deux fichiers au compilateur C :

```
$ gcc arith.c main.c -o main
```

Le programme est effectivement compilé et son exécution n'est pas différente de celle d'un programme qui aurait été écrit dans un seul fichier. Cependant, le compilateur s'est plaint, avec un avertissement, de l'utilisation dans `main.c` d'une fonction `power` qu'il ne connaît pas :

```
main.c:8:18: warning: implicit declaration of function 'power'
```

En effet, tout se passe ici comme si on compilait successivement, et indépendamment, les deux fichiers `arith.c` et `main.c`, avant de réaliser une édition de liens avec les deux codes compilés `arith.o` et `main.o` :

```
$ gcc -c arith.c
$ gcc -c main.c
$ gcc arith.o main.o -o main
```

On appelle cela de la *compilation séparée*. C'est dans la deuxième commande — c'est-à-dire la compilation de `main.c` — que le compilateur se plaint de ne pas connaître la fonction `power`.

Pour y remédier, il faut déclarer la fonction `power` dans le fichier `main.c` avant de l'utiliser. Pour cela, on écrit la ligne :

```
int power(int x, int n);
```

qui déclare l'existence d'une fonction `power` et donne son type. On note que la ligne se termine par un point-virgule, sans corps pour la fonction. Avec cette déclaration, le compilateur C

dispose de toute l'information nécessaire (nombre et types des arguments, type du résultat) pour compiler le fichier `main.c`.

Lors de l'édition de liens (troisième commande), le compilateur C va vérifier que la fonction `power` promise dans `main.c` est bien présente, en l'occurrence dans `arith.o`. Si elle venait à manquer, la compilation échouerait avec un message du type suivant :

```
main.c:(.text+0x46): undefined reference to ‘power’
```

Le compilateur C accepte, modulo un avertissement, de compiler un fichier qui fait référence à une fonction non déclarée — même s'il n'est pas conseillé de le faire — mais il refusera en revanche de construire un exécutable dans lequel il manquerait une fonction.

Fichiers d'en-têtes

On comprend que si on commence à ajouter d'autres fonctions dans `arith.c`, il va falloir les déclarer partout où elles seront utilisées. Si, en particulier, on utilise `arith.c` dans plusieurs programmes, il faudra dupliquer d'autant les déclarations des fonctions fournies par `arith.c`, à minima celles qui sont effectivement utilisées. Et si le nom ou le type d'une de ces fonctions vient à changer, il faudra mettre à jour toutes les déclarations. Ce n'est pas satisfaisant.
Une solution consiste à écrire les déclarations des fonctions fournies par `arith.c` dans un unique fichier, une fois pour toutes. On appelle cela un *fichier d'en-tête* (suffixe `.h`, de l'anglais *header*). Dans notre cas, on écrit donc un fichier `arith.h` contenant une seule ligne :

```
int power(int x, int n);
```

On peut alors inclure ce fichier d'en-tête dans notre fichier `main.c` en utilisant la directive :

```
#include "arith.h"
```

C'est identique à l'inclusion de fichiers d'en-têtes de bibliothèques (comme `stdlib.h`), la seule différence étant l'usage de guillemets autour de `arith.h`, qui indique au compilateur de le chercher localement (dans le répertoire courant) plutôt que dans la bibliothèque du système.
Supposons maintenant que nous augmentions `arith.c` avec d'autres déclarations, par exemple un type pour la division euclidienne et une fonction correspondante :

```
typedef struct Euclidean { int quotient, remainder; } euclidean;

euclidean division(int a, int b);
```

Pour éviter de répéter la définition du type dans chaque fichier, il suffit d'inclure `arith.h` dans `arith.c` :

```
#include "arith.h"

euclidean division(int a, int b) { ... }
int power(int x, int n) { ... }
```

Ainsi, le compilateur peut vérifier que les définitions dans `arith.c` sont conformes aux déclarations dans `arith.h`.

Si notre programme contient de nombreux fichiers et fichiers d'en-têtes inclus en cascade, il arrive qu'un même en-tête soit inclus plusieurs fois. Cela peut provoquer une erreur de redéfinition, typiquement :

```
error: redefinition of 'struct ...'
```

Une solution à ce problème consiste à rendre l'inclusion d'un en-tête idempotente, c'est-à-dire sans effet la seconde fois, en se servant des directives `#ifndef` et `#define` de la manière suivante (ici sur l'exemple de notre fichier `arith.h`) :

```
#ifndef ARITH_H
#define ARITH_H

int power(int x, int n);

#endif
```

La première fois que le fichier est inclus, la macro `ARITH` n'est pas définie et tout le contenu du fichier est donc considéré. En particulier, `#define` définit la macro `ARITH` — avec un contenu vide, en l'occurrence. Si le fichier est inclus de nouveau par la suite, la macro étant maintenant définie, tout le bloc entre `#ifndef` et `#endif` est ignoré, ce qui est l'effet attendu.

Espace de noms.

Le découpage d'un programme en plusieurs fichiers, et plus généralement la construction de bibliothèques, pose un autre problème : celui de l'*espace de noms*. Dans le langage C, l'espace des noms est complètement à plat. Deux fonctions d'un même programme, même si elles résident dans des fichiers différents, ne peuvent pas porter le même nom. Ainsi, on ne pourrait pas avoir, à côté de `arith.c`, un autre fichier `modular.c` introduisant :

```
int power(int x, int n, int m) { ... }
```

pour calculer x à la puissance n modulo m, car la fonction `arith.c` contient déjà une fonction `power`. (Il n'y a pas de surcharge en C.)
La même contrainte d'unicité vaut pour les structures (`struct`) et les types (`typedef`). Tant qu'on maîtrise l'ensemble des fichiers, on peut s'en sortir en donnant des noms explicites et uniques à toutes les fonctions, structures et types. Mais lorsqu'on développe une bibliothèque destinée à être utilisée ailleurs, on risque des conflits de noms.
Une bonne pratique consiste à **préfixer les noms** avec leur origine : par exemple `arith_power` et `arith_division`.

3 Type incomplet

Il est possible de déclarer l'existence d'une structure sans en donner la définition complète (ses champs). On appelle cela un *type incomplet*. Ainsi, on peut écrire :

```
struct MyStruct;
```

ou encore :

```
typedef struct MyStruct mytype;
```

et définir la structure plus loin, dans un autre fichier par exemple. Cette technique est utile pour les bibliothèques : on peut révéler uniquement l'existence d'un type dans un fichier d'en-tête, sans exposer sa représentation.

Imaginons une bibliothèque d'ensembles d'entiers, dans un fichier `set.c`. Son fichier d'en-tête `set.h` pourrait contenir :

```
typedef struct Set set;

set *set_create(void);
void set_add(set *s, int x);
bool set_mem(set *s, int x);
void set_remove(set *s, int x);
int set_card(set *s);
void set_delete(set *s);
```

Ici, on révèle qu'un ensemble est un pointeur vers une structure `Set`, mais on ne montre pas sa définition. Cela suffit pour que le compilateur sache manipuler des variables de type `set*` et vérifier les appels de fonctions.

La définition précise de la structure `Set` n'est donnée que dans le fichier `set.c`, où ses champs sont effectivement utilisés.

La création d'un ensemble (et donc l'allocation de sa structure sous-jacente) se fait via la fonction `set_create`. Une telle allocation ne pourrait pas être réalisée ailleurs sans connaître la taille exacte de la structure ni ses champs. De même, la désallocation est assurée par `set_delete`, afin d'éviter tout oubli de libération mémoire. Libérer directement un `set*` avec `free(s)` pourrait conduire à des fuites mémoire.

Exemple

Programme organisé en plusieurs fichiers

Le fichier `arith.h` contient les déclarations exportées, protégées contre les inclusions multiples :

```
#ifndef ARITH
#define ARITH
int arith_power(int x, int n);
typedef struct Euclidean { int quotient, remainder; } euclidean;
euclidean arith_division(int a, int b);
#endif
```

Le fichier `arith.c` contient les définitions :

```
#include <assert.h>
#include "arith.h"

int arith_power(int x, int n) { ... }
euclidean arith_division(int a, int b) { ... }
```

Le fichier `main.c` utilise la bibliothèque :

```
#include <stdlib.h>
#include <stdio.h>
#include "arith.h"
```

```c

int main(int argc, char **argv) {
    int x = atoi(argv[1]), n = atoi(argv[2]);
    printf("%d\n", arith_power(x, n));
}
```

Compilation :

```
$ gcc arith.c main.c -o main
```

ou compilation séparée :

```
$ gcc -c arith.c
$ gcc -c main.c
$ gcc arith.o main.o -o main
```

On profite de cet exemple pour remarquer que la fonction `set_create` a été déclarée avec le profil `set_create(void)`. La présence de `void` en argument indique explicitement que la fonction ne reçoit aucun paramètre. À l'inverse, une déclaration `set_create()` signifierait que la fonction accepte un nombre d'arguments non spécifié.