Informatique *Cours*
Programmation impérative avec C - Premiers pas Ipesup

Objectifs

- Comprendre les principes du langage C.
- Savoir écrire, compiler et exécuter un programme simple.
- Manipuler les types de base, les opérations et les structures de contrôle.
- Définir et appeler des fonctions, en comprenant l'usage de `return`.

Le langage C est un assembleur de haut niveau : il offre un contrôle précis sur la machine tout en restant portable entre architectures, systèmes et compilateurs. Son comportement est strictement défini par des standards successifs (C89, C99, C11, C18), et le code de cet ouvrage suit C99 et ses successeurs.

Le standard introduit les comportements non définis (undefined behavior), comme une division par zéro, un accès hors tableau ou un débordement arithmétique. Lorsqu'ils surviennent, l'exécution devient imprévisible : le programme peut s'arrêter ou produire des résultats erronés. Le compilateur, supposant leur absence, optimise alors le code très librement.

Cette leçon propose une première approche du langage C à travers des programmes simples manipulant entiers et booléens.

1 Généralités

Écrivons un programme C quiaffiche le texte `hello world!` sur la sortie standard, suivi d'un retour chariot.

```c
// mon premier programme C
#include <stdio.h>
int main() {
  printf("hello world!\n");
}
```

Sa structure est composée

- d'un commentaire : la ligne commençant par `//`,
- du chargement de la bibliothèque `stdio`,
- d'un programme principal `main` qui affiche le texte avec la fonction `printf`.

Pour exécuter ce programme, on le *compile* avec un compilateur C, par exemple `gcc` sous Linux.

```
$ gcc hello.c -o hello
```

Ici, on a demandé la construction d'un exécutable appelé `hello`. Si la compilation se déroule sans erreur, on peut alors lancer cet exécutable.

```
$ ./hello
hello world!
```

Il est intéressant de relever plusieurs différences significatives avec le langage Python :

— Le point d'entrée d'un programme C est toujours la fonction `main`, appelée en premier. D'autres fonctions peuvent bien sûr être définies et invoquées depuis `main`.
— Les retours à la ligne et l'indentation n'ont aucune signification pour le compilateur C. Le code pourrait tenir sur une seule ligne et rester valide.

```c
#include <stdio.h>
int main( ) { printf("hello world!\n"); }
```

L'indentation sert uniquement à la lisibilité, et la plupart des éditeurs C l'automatisent.
— Contrairement à Python, C est un langage compilé : le code source est traduit en langage machine avant exécution. Cela implique :
  — Une efficacité très élevée : un programme C est souvent cent fois plus rapide qu'un programme Python équivalent.
  — Une vérification stricte à la compilation : toute erreur de type, d'appel de fonction ou de variable inexistante provoque un échec avant exécution. C'est le *typage statique*.
  — En Python, ces erreurs ne sont détectées qu'à l'exécution, ce qui illustre le *typage dynamique*.

Attention

En général, « statique » désigne ce qui est connu avant l'exécution, et « dynamique » ce qui ne l'est qu'au moment de l'exécution.

2 Types et opérations élémentaires

Dans tout cet ouvrage, on suppose qu'on a chargé l'ensemble des bibliothèques suivantes au début de chaque fichier :

```c
#include <assert.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
```

Ceci introduit un certain nombre de types et de fonctions. Nous détaillons ici les principales.

Booléens.

Le type des booléens est `bool` et ses deux valeurs sont `true` et `false`.
On peut comparer des valeurs numériques avec les opérations `==`, `!=`, `<`, `>`, `<=` et `>=`.
Les opérations logiques sont `!` (négation), `&&` (conjonction) et `||` (disjonction). Ces deux dernières opérations sont paresseuses, c'est-à-dire qu'elles n'évaluent pas leur seconde opérande si la première suffit à déterminer le résultat.
Ainsi, on peut calculer le booléen :

```c
x != 0 && y/x < z
```

sans risquer une division par zéro, car l'expression `y/x < z` ne sera pas évaluée si `x != 0` vaut `false`.

Entiers

Le langage C propose plusieurs types d'entiers prédéfinis, variant selon leur taille et leur signe.
Les types `int8_t`, `int32_t` et `int64_t` désignent des entiers signés sur 8, 32 et 64 bits, tandis que `uint8_t`, `uint32_t` et `uint64_t` correspondent aux versions non signées.
Lorsque la taille importe peu, on utilise simplement `int` (signé) ou `unsigned int` (non signé), généralement sur 32 bits dans les compilateurs modernes.
Les opérations arithmétiques sont les classiques `+`, `-`, `*`, `/` et `%`.
Pour un dividende positif, `/` et `%` suivent la division euclidienne ; pour un dividende négatif, le reste devient aussi négatif et la correspondance n'est plus exacte.
Un débordement sur un entier signé est un *undefined behavior*, tandis que pour un entier non signé, le calcul se fait modulo la taille du type.
Pour générer un entier aléatoire entre 0 inclus et n exclu, on écrit `rand() % n`. La fonction `rand`, issue de `stdlib`, renvoie un entier entre 0 et `RAND_MAX`, dont la valeur dépend de l'implémentation — sous Linux avec gcc, on a typiquement `RAND_MAX` = $2^{31} - 1$.

Flottants.

Le langage fournit plusieurs types de nombres flottants, dont un type `double` qui correspond, sur la plupart des systèmes, à des flottants 64 bits conformes à la norme IEEE 754.
Les opérations se notent `+`, `-`, `*`, `/`, comme pour l'arithmétique entière, mais ce sont des opérations différentes.

Caractères.

Le type des caractères est `char`. Il s'agit d'un type numérique, d'un seul octet. On ne peut donc représenter essentiellement que les caractères ASCII 7 bits.
Les caractères se notent entre guillemets simples : `'a'`, `'+'`, etc.
Le caractère `\n` est une séquence d'échappement qui permet d'écrire un retour chariot,
`\t` pour une tabulation,
`\'` pour un guillemet simple
`\\` pour le caractère `\`.
Le caractère `\0`, de code 0, est appelé *caractère nul*. Il est utilisé, en interne, pour terminer les chaînes de caractères.
Une chaîne de caractères est notée entre guillemets doubles, comme `"hello"`. Les mêmes séquences d'échappement que pour les caractères peuvent être utilisées dans les chaînes.
Une chaîne de caractères a le type `char*`.

2.1 Entrées et sorties simples

En C, les interactions avec l'utilisateur se font à l'aide des fonctions `printf` et `scanf`, définies dans la bibliothèque standard `<stdio.h>`. La fonction `printf` permet d'afficher du texte ou la valeur de variables à l'écran, tandis que `scanf` lit des valeurs saisies au clavier et les stocke dans des variables.

Les deux utilisent des *formats* pour préciser le type des données manipulées : `%d` pour un entier, `%f` pour un nombre réel, `%c` pour un caractère, et `%s` pour une chaîne de caractères.

Exemple

```
#include <stdio.h>

int main(void) {
    int age;
    printf("Entrez un entier pour votre age: ");
    scanf("%d", &age);  // & indique l'adresse où stocker la valeur
    printf("Vous avez %d ans.\n", age);
    return 0;
}
```

3 Structures de contrôle

3.1 Blocs et variables.

Une séquence d'instructions peut être regroupée dans un *bloc*, délimité par des accolades. Un bloc peut contenir la déclaration d'une ou plusieurs variables. La portée d'une variable s'étend jusqu'à la fin du bloc dans lequel elle est déclarée.

```
{
  int x = 41;
  x = x + 1;
}
// la variable x n'est plus visible ici
```

Un bloc est considéré comme une instruction. En particulier, un bloc peut contenir des sous-blocs.

3.2 Conditionnelle.

Une conditionnelle est introduite avec le mot-clé `if` et une expression booléenne entre parenthèses.

```
if (x > 0) { ... } else { ... }
```

La partie `else` peut être omise.

Un bloc `else if` est une autre construction `if`, pour enchaîner les conditions.

```
if (x > 0) { ... }
else if (x < 0) { ... }
else { ... }
```

Il existe également une construction d'*expression conditionnelle*, qui se note avec une expression de test suivie d'un point d'interrogation et deux expressions séparées par deux points.

Exemple

On peut écrire :

```
int z = x > y ? x : y;
```

pour déclarer une variable `z` qui reçoit le maximum de `x` et `y`.

3.3 Boucles.

Une boucle « tant que » est introduite avec le mot-clé `while` et son test s'écrit entre parenthèses, comme pour une conditionnelle.

```
while (n > 0) { ... }
```

Exemple

Si on souhaite parcourir tous les entiers de 0 à n–1 avec une variable i, on peut le faire de façon élémentaire avec une boucle `while`, comme ceci :

```
int i = 0; while (i < n) { ...; i = i+1; }
```

Il existe cependant une construction plus idiomatique pour cela, en utilisant le mot-clé `for`.

```
for (int i = 0; i < n; i = i+1) { ... }
```

La portée de la variable `i` est alors limitée au corps de la boucle.

Que ce soit dans une boucle `while` ou une boucle `for`, l'instruction `break` permet de sortir de la boucle et l'instruction `continue` permet de sauter immédiatement à l'itération suivante de la boucle.

Exemple

```
int main()
{
    for (int i = 0; i < 10; i++)
    {
        if (i == 3)
            continue; // saute le reste du corps quand i == 3
        if (i == 7)
            break; // quitte completement la boucle quand i == 7
        printf("%d ", i);
    }
    printf("\nFin de la boucle.\n");
    return 0;
}
```

Le programme affiche :

```
0 1 2 4 5 6
Fin de la boucle.
```

Pour une boucle `for`, cela inclut l'instruction d'incrémentation spécifiée en troisième position, `i = i+1` dans notre exemple.

Comme il est fréquent de devoir incrémenter une variable entière, notamment pour écrire une boucle `for`, il existe des constructions plus compactes. Ainsi, on peut écrire `i += 1` pour ajouter 1 à la variable `i`, et même tout simplement `i++` pour incrémenter la variable `i`, ce qui conduit à cette écriture traditionnelle d'une boucle for sur les n premiers entiers :

```
for (int i = 0; i < n; i++) { ... }
```

De la même façon, on peut décrémenter une variable avec `i-`.

Forme generale

La boucle `for` du C a une forme plus générale, où les trois composantes entre parenthèses ne sont pas limitées à la déclaration d'une variable, son test et sa mise à jour. Ainsi, on peut écrire des choses comme

```
for (; x != 1; x = f(x)) { ... }
```

pour une variable `x` définie en dehors de la boucle ou encore

```
for (start(); test(); step()) { ... }
```

pour trois fonctions `start`, `test` et `step` définies par ailleurs.

Opérateurs ++ et −

En C, l'expression `n++` renvoie la valeur de `n` avant de l'incrémenter. À l'inverse, `++n` incrémente d'abord la variable puis renvoie sa nouvelle valeur.
Les opérateurs `n-` et `-n` fonctionnent de la même façon pour la décrémentation.
Ainsi, l'instruction

```
while (n-- > 0) { ... }
```

exécute le bloc exactement `n` fois.
Il ne faut cependant pas abuser de ces opérations et notamment garder à l'esprit que l'ordre d'évaluation des opérandes, ainsi que des arguments d'un appel, est non spécifié en C.

Au-delà des booléens

La construction `if` et les constructions de boucles autorisent l'utilisation d'une expression numérique comme test.
Le test est vrai si la valeur de cette expression est non nulle.

Exemple

Il est possible d'écrire :

```
if (x) ...
```

pour tester si l'entier `x` est non nul.

Les booléens eux-mêmes ne sont d'ailleurs que des valeurs numériques, valant 1 (pour `true`) ou 0 (pour `false`). On décourage cependant l'utilisation des entiers comme boo-

léens et inversement, car cela nuit à la clarté du code.
Pour autant, cela reste un usage répandu en C que de tester le caractère non nul d'un entier directement avec `if` ou `while`.

Exemple

Si `read_data` est une fonction qui lit des entrées, les stocke dans une structure globale et renvoie le nombre d'entrées qui ont été lues, il est courant de voir du code comme :

```
1 while (read_data()) { ... }
```

pour lire et traiter les entrées tant qu'il y en a.

3.4 Fonctions.

Une fonction est définie en donnant son type de retour, son nom et la liste de ses paramètres avec leurs types. Le corps de la fonction est un bloc. La fonction renvoie un résultat avec l'instruction `return`.

Exemple

Exemple d'une fonction `quotient` avec deux paramètres de type `int` et un résultat de type `int` :

```
1 int quotient(int a, int b) {
2   int q = 0;
3   while (a >= b) { a -= b; q++; }
4   return q;
5 }
```

Ici, on renvoie la valeur de la variable `q`.

L'instruction `return` peut apparaître à plusieurs endroits dans la fonction, y compris à l'intérieur d'une branche de conditionnelle ou d'une boucle.

Attention

S'il manque un `return` sur l'une des branches du flot de contrôle, comme par exemple dans cette fonction :

```
1 int f(int x) { if (x < 0) return 42; }
```

alors le compilateur le signale avec un avertissement :

```
1 warning: control reaches end of non-void function
```

En l'absence d'instruction `return`, la fonction termine lorsqu'elle atteint la fin du corps de la fonction. Il est cependant possible de terminer l'appel avant cela, en utilisant une instruction `return` sans argument.

Si une fonction ne renvoie pas de valeur, elle est déclarée avec le mot-clé `void` à la place du type de retour.

```
void print_true(bool b) { if (b) { printf("true"); } }
```

Il ne s'agit pas d'un type. En particulier, on ne peut pas déclarer une variable avec un type qui serait `void`.

On aura remarqué que la fonction `main` est déclarée avec le type de retour `int`. Ceci permet au programme de renvoyer au système un code de retour, interprété par l'environnement. Sur la plupart des systèmes, on signale une bonne exécution avec le code de retour 0 ou au contraire un problème avec un code de retour différent de 0, dont la valeur peut alors porter une signification.

Le compilateur C ajoute une instruction `return 0;` à la fin de `main` si l'utilisateur n'a pas renvoyé de résultat explicitement.

Fonctions récursives Une fonction peut être récursive. Pour définir deux fonctions mutuellement récursives, ou plus, il faut commencer par *déclarer* l'existence de la seconde fonction, définir la première puis enfin définir la seconde.

Exemple

```
bool even(unsigned int x);
bool odd(unsigned int x) { return x != 0 && even(x-1); }
bool even(unsigned int x) { return x == 0 || odd(x-1); }
```

Ici la déclaration de `even` permet au compilateur de compiler la fonction `odd`, et notamment de la typer.

3.5 Constantes.

On peut ajouter le qualificatif `const` devant la déclaration d'une variable pour en faire une *constante*, c'est-à-dire une variable qu'il n'est pas possible de modifier après son initialisation.

```
const int len = 10;
```

Si on tente de la modifier, on obtient une erreur à la compilation :

```
error: assignment of read-only variable 'len'
```

Le compilateur C propage parfaitement les valeurs des constantes.

Devant un paramètre d'une fonction, le qualificatif `const` signifie que la fonction ne peut modifier cette variable. Là encore, le compilateur fait la vérification et s'interrompt avec une erreur si on cherche à faire une affectation de la variable.

4 Modèle d'exécution

Pour bien utiliser un langage, il faut comprendre son modèle d'exécution. C'est nécessaire en premier lieu pour la correction de nos programmes -cette donnée est-elle copiée ou bien partagée ?- mais également pour leur efficacité -quel est le coût de cette instruction ?-.

Le modèle d'exécution du langage C est assez subtil. En particulier, il est plus complexe que celui d'OCaml ou de Python. Dans cette section, nous en donnons les grandes lignes. Dans la section suivante, nous rentrerons dans les détails en décrivant les pointeurs, les tableaux et les structures.

Comme dans la majorité des langages de programmation, les appels de fonctions en C sont *imbriqués* : si une fonction f appelle une fonction g, alors cet appel à g termine avant que l'appel à f ne termine. Cette propriété permet une compilation très efficace des appels de fonctions, en utilisant une *pile*, qu'on nomme *pile d'appels*.

Lors d'un appel à une fonction f, les données relatives à cet appel (paramètres, adresse de retour, variables locales, etc.) sont placées au sommet de la pile.

L'ensemble de ces données forment ce qu'on appelle un *tableau d'activation* (en anglais *stack frame*). Si le code de la fonction f vient à appeler une fonction g, alors le tableau d'activation de g viendra se placer au sommet de la pile, par dessus celui de f. Lorsque l'appel à g termine, le tableau d'activation de g est dépilé et on retrouve celui de f au sommet de la pile. En particulier, la pile est d'autant plus grande qu'il y a d'appels de fonctions imbriqués en cours d'exécution.

Le compilateur C va organiser les différents espaces mémoire nécessaires à l'exécution du programme selon le schéma suivant : le code du programme est placé dans les adresses basses de la mémoire. Au-dessus, on trouve les données statiques, c'est-à-dire les données connues au moment de la compilation, comme par exemple une chaîne de caractères présente dans le code source. Le reste de la mémoire va être utilisé dynamiquement, c'est-à-dire pendant l'exécution du programme. Il se partage entre la pile d'appels, placée tout en haut de la mémoire, et le *tas*, qui est constitué de blocs de mémoire alloués avec la fonction de bibliothèque `malloc`. Avec cette organisation, la pile n'interfère pas avec le tas. La pile croît vers les adresses basses, c'est-à-dire que le fond de la pile est tout en haut de la mémoire et le sommet de la pile est plus bas.

Mode de passage. Le langage C est muni de ce qu'on appelle un *passage par valeur*. Cela signifie que, lors d'un appel de fonction, les paramètres effectifs de l'appel sont d'abord évalués, puis leurs valeurs sont *copiées* dans autant de nouvelles variables qu'il y a de paramètres, et enfin le code de la fonction est exécuté. Illustrons-le avec une fonction `incr` qui reçoit un entier en paramètre dans une variable `x` :

```
1 void incr(int x) {
2   x = x + 1;
3 }
4 void f() {
5   int v = 41;
6   incr(v);
7   // v vaut toujours 41
8 }
```

Ici, la fonction `incr` reçoit, dans une *nouvelle* variable `x`, la *valeur* de la variable `v`. C'est la variable `x` qui est incrémentée. La variable `x` est locale à la fonction `incr` et elle cesse d'exister avec le retour de cette fonction. La valeur de la variable `v` n'a donc pas été modifiée.

Il y a bien deux variables en jeu, allouées à des endroits différents. Il est important de bien comprendre que les noms « v » et « x » ne sont nulle part représentés en mémoire. Le compilateur a déterminé des emplacements mémoire pour ces variables (dans un registre, sur

la pile, dans le segment de données, etc.) et il produit du code qui y fait référence. Ce sont uniquement nos schémas qui matérialisent ces noms de variables pour notre compréhension.

Débordement de la pile d'appels. Sur la plupart des systèmes, la pile d'appels a une taille maximale, relativement petite. Sous Linux, par exemple, elle est de 8 Mo. Si on vient à épuiser cette ressource par un trop grand nombre d'appels de fonctions imbriqués, on va provoquer un *débordement de pile* (en anglais *stack overflow*), ce qui se manifeste par une *erreur de segmentation*, c'est-à-dire un accès à une zone mémoire non autorisée, et une interruption du programme. C'est souvent le symptôme d'une récursion qui ne termine pas, par exemple parce qu'on a oublié un cas de base.