C : Bases 1

Pierre Karpman

Lycée Champollion MP2I

https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/

À propos du C

Langage historique & récent

- Version initiale au début des 70's, dans le cadre du développement d'Unix
- Première norme en 1989 (« C89 »)
- Dernière version en date en 2023 (« C23 »), et ce n'est pas fini !
- Pour nous : « norme C99 ou plus récente »

Langage « bas-niveau »

- Peu d'abstractions ; interfaçage aisé avec l'assembleur
  - Contrôle précis : souvent bon pour la performance, mais moins agréable à l'écriture
- Gestion explicite de la mémoire
  - Source interminable de bugs :(

Langage subtil

https://wiki.sei.cmu.edu/confluence/display/c/CC.+Undefined+Behavior

Langage courant

Langage impératif

Programme : modifier un état par une suite d'instructions ; la sortie est l'état final

- Plutôt comme les automates, les machines de Turing
- Plutôt comme Python, plutôt pas comme OCaml

Langage compilé

Démo.

- Pour pouvoir s'exécuter un programme doit être compilé
- Ce n'est pas parce qu'un programme compile (ou s'exécute) qu'il n'est pas buggé !!

Table des matières

Structure d'un programme C

Sur un exemple :

```c
// ne fait rien, avec succès
#include <stdlib.h>

int main(void)
{
    return EXIT_SUCCESS;
}
```

Table des matières

Sorties 1

On souhaite généralement qu'un programme puisse interagir avec son environnement

- Comment un programme peut-il interagir avec son environnement ??

Sorties 1

On souhaite généralement qu'un programme puisse interagir avec son environnement

- Comment un programme peut-il interagir avec son environnement ??

https://youtu.be/gp_D8r-2hwk?si=QJ0Als1KmdvpLLaf&t=50

printf

`printf` en quelques points :

- *Fonction* d'affichage de texte formaté sur la *sortie standard*
- Déclarée dans `stdio.h`
- Souvent suffisante pour la plupart des usages
- Fonctions similaires : `fprintf`, `puts`, `fputs` (aussi dans `stdio.h`)

Exemple d'utilisation basique dans un programme simple : appel de la fonction `printf` avec la *chaîne de caractères* `"OHAI\n"` comme unique *argument* :

```c
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
    printf("OHAI\n");
    return EXIT_SUCCESS;
}
```

printf bis

`printf` en quelques points supplémentaires :

- Le premier argument de `printf` peut contenir des *spécifications de conversion*, remplacées à l'affichage par des arguments supplémentaires. Par exemple :
  - `%d` pour l'affichage décimal d'un entier de type `int`
  - `%s` pour l'affichage d'une chaîne de caractères

Exemple :

```c
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    printf("%s %d\n", "OHAI", 1);
    return EXIT_SUCCESS;
}
```

Table des matières

Fonctions : principe & utilité

(En C) une fonction peut se voir comme un fragment de code définissant une action précise à effectuer, qui le sera lors de son *appel*

- Une fonction prend zéro, un, deux… *paramètres* qui permettent de faire varier l'action effectuée lors de son appel
- À l'appel d'une fonction à n paramètres, il faut fournir autant d'*arguments*

L'utilisation de fonctions permet de structurer efficacement un programme

- Diminue les répétitions, favorise la modularité
- Améliore la lisibilité

Déclaration, définition en C

Pour beaucoup d'éléments du langage C (dont les fonctions), on distingue *déclaration* et *définition*

Déclaration

- Déclarer qu'une chose existe
- Permet de l'utiliser, à condition qu'elle soit bien définie *quelque part*
- Peut éventuellement être fait plusieurs fois

Définition

- Décrire ce en quoi elle consiste
- Doit être fait exactement une fois, si la chose est utilisée

Syntaxe de signature, déclaration & définition d'une fonction

Signature d'une fonction

```
nom_du_type_de_renvoi nom_de_la_fonction(paramètres de la fonction)
```

Avec nom_du_type_de_renvoi : `void` si l'on ne renvoie rien, et sinon `int`, etc.
Avec `paramètres de la fonction` :

- `void` s'il y en a zéro
- `t1 p1, t2 p2, ..., tn pn` avec `ti` les *types*, `pi` les noms de *variable*

Déclaration

```
sig;
```

Pour déclarer la fonction de signature `sig`
Parfois appelé un *prototype* de cette fonction

Définition

```
sig {
    instructions définissant ce que fait la fonction
}
```

Exemple

```c
#include <stdlib.h>
#include <stdio.h>

void print_greeting(int n);

int main(void)
{
    print_greeting(12);
    return EXIT_SUCCESS;
}

void print_greeting(int n)
{
    printf("OHAI %d\n", n);
}
```

Flot d'exécution & valeur de retour

Pour nous les fonctions ont :

- un unique point d'entrée (la première instruction de leur *corps*)
- possiblement plusieurs points de sortie : chaque instruction `return` & la dernière instruction du corps

Démo.

Table des matières

Variables

Les variables jouent un rôle central dans tout langage de programmation un tant soit peu avancé. Dans un langage impératif, elles forment une interface essentielle entre le code (tel qu'on l'écrit) et l'état d'un programme (lors de son exécution)

Définition

Variable :

- Identifiant pour un *objet*
- Possède un *type*, décidé *statiquement*
  - Détermine comment interpréter l'objet *référencé*

Utilisation

- Doit avoir été déclarée en amont
- Peut être *lue*, et (en général) *affectée* (« écrite »)

Syntaxe d'utilisation

- Déclaration seule (ne définit **pas** la variable):

```
nom_du_type nom_de_la_variable;
```

> La lecture d'une variable non définie est **interdite** : c'est un *undefined behaviour*, ou UB

- Déclaration avec initialisation :

```
nom_du_type nom_de_la_variable = initialiseur;
```

Largement préférable

- Lecture :

```
nom_de_la_variable
```

- Écriture :

```
nom_de_la_variable = valeur
```

Exemple

```c
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int tw = 12;

    printf("OHAI %d ", tw);
    printf("HAI %d\n", tw);

    return EXIT_SUCCESS;
}
```

Démo.

Table des matières

Expressions

Une expression est « quelque chose » qui peut s'évaluer pour produire une valeur

En C :

- Peu indépendant des instructions
- Toute expression a un type

Expressions les plus courantes :

- Constantes littérales : `12`, `"OHAI"`…
- Les noms de variable
- Les expressions arithmétiques, booléennes : `1 + 2`, `3 < 4`…
- Les appels de fonction : `printf("OHAI\n")`

Types entiers

Plus d'une douzaine (!) de types pour représenter les nombres entiers en C

Caractéristiques communes aux types entiers C

- Un type est *signé* ou *non signé*
- Précision finie
- *Overflow/underflow* pour un type signé : **interdit** (UB)
- *Overflow/undeflow* pour un type non signé : « réduction modulo » (*wraparound*)

Types de base « historiques »

- `int` : signé, représente *en pratique* l'intervalle $[\![-2\,147\,483\,648, 2\,147\,483\,647]\!]$
- `unsigned int` (ou tout simplement `unsigned`) : non signé, représente *en pratique* l'intervalle $[\![0, 4\,294\,967\,295]\!]$
  - Affichage en décimal avec `printf` avec `%u`, en hexadécimal (base 16) avec `%x` ou `%X`
- Et d'autres…

Expressions arithmétiques

- Opérateurs usuels : `+` ; `-` ; `*` ; `/` ; `%`
  - Mêmes règles de priorité qu'à l'école ; parenthésage possible
  - En l'absence d'UB, de wraparound, forment des expressions qui s'évaluent à « la même chose que dans $\mathbb{Z}$ »

Division

Le plus sage est de se limiter à des opérandes positives ; alors `q = x / y`, `r = x % y` sont t.q. :

- $r \in [\![0, y-1]\!]$
- $x = qy + r$

Attention aux divisions par zéro !

Type booléen

Type permettant de représenter deux valeurs : « vrai » et « faux »

`bool`

- Défini dans `stdbool.h`
- Deux littéraux : `true` et `false`
  - S'affichent comme des `int` (à `1` et `0` respectivement)

Expressions booléennes

Comparaison entre entiers

- *Via* `==` ; `!=` ; `<` ; `<=` ; `>` ; `>=`

Combinaison d'expressions booléennes

- *Via* `&&` ; `||` ; `!`
- `&&` et `||` sont *paresseux*

Exemple

```c
#include <stdlib.h>
#include <stdio.h>
bool is_multiple(unsigned x, unsigned y) {
    return (y != 0) && (x % y == 0);
}
bool is_multiple_unsafe(unsigned x, unsigned y) {
    return x % y == 0;
}
int main(void) {
    printf("%d\n", is_multiple(12, 3));
    printf("%d\n", is_multiple(13, 3));
    printf("%d\n", is_multiple(13, 0));
    printf("%d\n", is_multiple_unsafe(13, 0)); // boum

    return EXIT_SUCCESS;
}
```

Table des matières

Instructions

Les instructions sont des constructions syntaxiques servant à contrôler l'état et le flot d'exécution d'un programme

En C :

- « Presque tout » est une instruction
- Exécution d'un programme ≈ exécution d'une suite d'instructions

Instructions « élémentaires »

- `return`
- affectation de variable
- appel de fonction suivi de « ; »
- liste d'instructions `i1; i2; i3; ...`
- bloc d'instructions `{ i1; i2; ... }` (et de déclarations)

Instructions de sélection

Une seule au programme : `if` :

```
if (expression) instruction_si_vrai
if (expression) instruction_si_vrai else instruction_si_faux
```

Pour nous :

```
if (expression_booléenne)
{
    instructions_si_vrai
}
else
{
    instructions_si_faux
}
```

Sémantique de `if` ; `if`s multiples

L'exécution d'un `if` est en deux étapes :

1. On évalue `expression_booléenne` (attention aux effets de bord !)
2. Si elle s'est évaluée à `true` on exécute `instructions_si_vrai`, sinon on exécute `instructions_si_faux`

On peut enchaîner & imbriquer des `if`s :

```
if (cond1) {
    // ...
    if (subcond1) {
        // ...
    }
}
else if (cond2) {
    // ...
}
else {
    // ...
}
```

Exemple

```
int max(int x, int y) {
    if (x > y) {
        return x;
    }
    else {
        return y;
    }
}

int max_var(int x, int y) {
    if (x > y) {
        return x;
    }
    return y;
}
```

Instructions d'itération

Deux au programme : `while` et `for` :

```
while (expression) instruction
for (clause ; expression1 ; expression2) instruction
```

Pour nous :

```
while (expression_booléenne)
{
    instructions
}
for (clause ; expression_booléenne ; expression_d_affectation)
{
    instructions
}
```

Sémantique de while

On commence par évaluer la condition expression_booléenne ; si elle s'évalue à false l'exécution de l'instruction while est terminée, sinon on exécute le corps de boucle, puis (à supposer que la boucle while n'a pas terminé pour d'autres raisons) on évalue à nouveau expression_booléenne ; si elle s'évalue à false l'exécution de l'instruction while est terminée, sinon on exécute le corps de boucle, puis (à supposer que la boucle while n'a pas terminé pour d'autres raisons) on évalue à nouveau expression_booléenne ; si elle s'évalue à false l'exécution de l'instruction while est terminée, sinon on exécute le corps de boucle, puis (à supposer que la boucle while n'a pas terminé pour d'autres raisons) on évalue à nouveau expression_booléenne ; si elle s'évalue à false l'exécution de l'instruction while est terminée, sinon on exécute le corps de boucle, puis (à supposer que la boucle while n'a pas terminé pour d'autres raisons) on évalue à nouveau expression_booléenne ; si elle s'évalue à false l'exécution de l'instruction while est terminée, sinon on exécute le corps de boucle, puis (à supposer que la boucle while n'a pas terminé pour d'autres raisons) on évalue à nouveau expression_booléenne ; si …

Boucles infinies ; `break`

- On veut généralement éviter les boucles *infinies*
  - Autant que possible, prouver que les `while`s terminent
- Mais des boucles *a priori* infinies sont parfois utiles :

```
while (true)
{
    // corps de boucle
}
```

  Pas forcément *réellement* infini si le corps contient un `return` ou un `break`

- `break` : interrompt l'exécution de la boucle (on passe à l'éventuelle instruction suivante)

Sémantique de `for`

```
for (clause ; expression_booléenne ; expression_d_affectation)
{
    instructions
}
```

expression d'affectation : instruction d'affectation sans le « ; »

- clause : déclaration de variable (avec init.) ou expression d'affectation

Soit `for (init ; test ; incr)` :

- `init` : exécutée une fois au début de l'exécution
- `test` : même rôle que pour `while`
- incr : exécutée à chaque fin d'itération
  - formes classiques : `i = i + 1`, `i = i - 1`, `i = i / 2`, ...
  - formes classiques hors programme  : `i += 1`, `i++`, `i--`, ...

`for` : remarques

- On peut utiliser `break` dans un `for` comme dans un `while`
- Toute ou partie de `init`, `test`, `incr` peuvent être laissés vide
  - Par ex. : `for ( ; test ; )` est équivalent à `while (test)` (peu utile)

Exemple

```
unsigned sum_n_while(unsigned n) {
    unsigned res = 0;
    unsigned i = 0;
    while (i <= n) {
        res = res + i;
        i = i + 1;
    }
    return res;
}

unsigned sum_n_for(unsigned n) {
    unsigned res = 0;
    for (unsigned i = 0; i <= n; i++) {
        res = res + i;
    }
    return res;
}

unsigned sum_n_math(unsigned n) {
    return (n * (n + 1)) / 2;
}
```

Table des matières

Portée

Les identifiants de variable ont une portée (*scope*) qui détermine les points du programme où ils sont visibles, et donc utilisables dans des instructions, des expressions…

Masquage

- Il se *peut* que plusieurs variables de même nom pourraient être visibles en un même point :

```
int i = -1;
if (i < 0)
{
  int i;
  i = 0 // pas la même variable qu'à la première ligne !
}
```

- La plus *interne* (« plus récente ») *masque* alors toutes les autres
- C'est facilement source de bugs… à éviter

Différents types de portée

Portée fichier

- Pour les variables *globales* déclarées en dehors de toute fonction
  - À utiliser le moins possible (idéalement : pas du tout)
- Visibles dans tout le fichier

Portée bloc

- Pour les paramètres de fonction, les variables déclarées dans le `init` d'un `for`, dans un bloc…
- Visibles dans tout le bloc (corps de fonction, corps du `for`, bloc…)

Table des matières

Politique de passage

En C, le passage des arguments d'une fonction est *par valeur* (*call-by-value*)

Soit un appel de fonction `fun(e)` :

- On évalue `e`
- On affecte sa valeur *v* à la variable dénotant le paramètre de `fun`

Une conséquence

Modifier une variable-paramètre dans une fonction ne modifie pas la valeur référencée par une éventuelle variable passée en argument

Exemple

```c
#include <stdlib.h>
#include <stdio.h>

void no_incr(unsigned x) {
    printf("%s@%d: x = %u\n", __func__, __LINE__, x);
    x = x + 1;
    printf("%s@%d: x = %u\n", __func__, __LINE__, x);
}

int main(void) {
    unsigned x = 0;
    printf("%s@%d: x = %u\n", __func__, __LINE__, x);
    no_incr(x);
    printf("%s@%d: x = %u\n", __func__, __LINE__, x);
    return EXIT_SUCCESS;
}
```

Table des matières

scanf

`scanf` en quelques points :

- *Fonction* d'entrée de texte formaté sur l'*entrée standard*
- Déclarée dans `stdio.h`
- Assez peu pratique, et complexe à utiliser correctement
- Pour nous : utilisations très basiques, typiquement pour des entrées numériques écrites en base 10, à stocker dans des `int` (`%d`) ou `unsigned` (`%u`)

Exemple d'utilisation :

```
unsigned g = 0;
int rd = scanf("%u", &g);
if (rd != 1)
{
    // gestion d'erreur
}
// g contient la valeur lue
```

Table des matières

Bases du développement (C)

Options de compilation

- Activez les avertissements, et **prenez les en compte** :
  `cc -Wall -Wextra -std=c11 -pedantic -o ohai ohai.c`
- (Éventuellement) utilisez un outil de compilation comme `make`
- (Plus tard : activez les *sanitizers* ; utilisez un niveau d'optimisation adapté)

Éditeur de texte

- Trouvez en un qui vous plaît bien, avec coloration syntaxique et indentation automatique

Convention de style

- Tirez profit de l'indentation automatique de votre éditeur de texte
- Adoptez un style homogène

Bases du développement (C) *bis*

Documentation

- Commentez votre code… à bon escient

Tests

- Testez votre code… tout le temps

`assert`

- Utilisez `assert` (défini dans `assert.h`) pour valider les préconditions, postconditions, invariants… : documentation et tests à la fois !

  `assert(be)`
  - fait échouer l'exécution avec un message explicite si l'expression booléenne `be` s'évalue à `false`
  - se désactive quand compilé avec `-DNDEBUG`
- Démo.