

IN 101 - Cours 01

21 septembre 2011



présenté par
Matthieu Finiasz

Comment programmer en C

Organisation du cours

- × 13 séances de 1h de cours + 2h de TD sur machine ou sur papier
 - × des cours de programmation et des cours d'algorithmique,
 - × 1 ou 2 groupes forts pour les TD
 - questionnaire à rendre à la fin du TD 01.

- × Les buts de ce cours :
 - × vous apprendre un (premier) langage de programmation : le C,
 - il vous sera facile d'en apprendre d'autres après,
 - × vous apprendre les bases de l'algorithmique,
 - pour que vous sachiez que ça existe !

- × Tous les supports de cours/TD sont sur ma page web :
<http://www-roc.inria.fr/secret/Matthieu.Finiasz/teaching.shtml>

Pourquoi utiliser un langage de programmation ?

- × Un processeur ne comprend que des instructions très simples :
 - × écrire dans un registre, additionner le contenu de 2 registres, copier un registre dans une zone mémoire...
 - × chaque processeur est différent : x86, amd64, ARM, Cell...

- × Un langage de programmation permet d'écrire un calcul de façon **standard**, indépendante de l'architecture cible.

- × Il suffit ensuite de **traduire ce programme** pour l'exécuter :
 - × soit avec un **interpréteur** : Maple/Matlab, shell, javascript, Perl...
 - × soit avec un **compilateur** : C, OCaml, Fortran...
 - × soit un "peu des deux" : Java, Python...

Un premier exemple de programme en C

Hello World

hello.c

```
1 #include <stdio.h>
2
3 int main() {
4     printf("Hello World!\n");
5     return 0;
6 }
```

- ✘ Voici le programme en C le plus simple pour afficher le message :
Hello World!
- ✘ En shell ça serait simplement :
> echo "Hello World!"

Un premier exemple de programme en C

Hello World

hello.c

```
1 #include <stdio.h>
2
3 int main() {
4     printf("Hello World!\n");
5     return 0;
6 }
```

- ✘ Inclusion de la librairie `stdio.h` : Standard Input/Output
 - ✘ cette ligne est nécessaire pour pouvoir afficher/lire dans le terminal ou dans des fichiers.
- ✘ Les lignes qui commencent par # sont lues par le **préprocesseur** :
 - ✘ `#include <stdio.h>` est remplacé par le contenu du fichier `stdio.h` qui définit les fonctions d'entrée/sortie.

Un premier exemple de programme en C

Hello World

hello.c

```
1 #include <stdio.h>
2
3 int main() {
4     printf("Hello World!\n");
5     return 0;
6 }
```

- ✘ La fonction `main` contient l'ensemble des commandes à exécuter :
 - ✘ toutes les instructions entre les accolades seront exécutées **dans l'ordre** par le programme,
 - ✘ un programme en C doit toujours contenir une fonction `main`.
- ✘ Le type `int` (entier) au début correspond au type renvoyé par `main`
 - ✘ pour le `main` c'est toujours un entier.

Un premier exemple de programme en C

Hello World

hello.c

```
1 #include <stdio.h>
2
3 int main() {
4     printf("Hello World!\n");
5     return 0;
6 }
```

- ✘ La fonction `printf` permet d'afficher un message dans le terminal :
 - ✘ `"Hello World!\n"` est une chaîne de caractères,
 - ✘ `\n` est le retour à la ligne (il n'y en a pas sinon).
- ✘ Le point virgule ; en fin de ligne marque la fin de l'instruction
 - ✘ en C, chaque instruction simple se termine par un point virgule.

Un premier exemple de programme en C

Hello World

```

_____ hello.c _____
1 #include <stdio.h>
2
3 int main() {
4     printf("Hello World!\n");
5     return 0;
6 }

```

- ✘ La commande `return` permet de sortir d'une fonction en **retournant** ce qui est derrière :
 - ✘ ici, la valeur de retour est 0, c'est bien un entier (type `int`).
- ✘ Le `return` de la fonction `main` est le code de sortie du programme :
 - ✘ cela existe pour toute commande unix,
 - ✘ permet de tester si une commande a marché,
 - ✘ 0 signifie que le programme s'est exécuté normalement.

Le compilateur GCC

- ✘ GCC permet de traduire un programme en C en exécutable
 - ✘ on doit lui donner le nom du fichier source et du fichier de sortie (avec l'option `-o`) au minimum :
 - `gcc hello.c -o hello`
- ✘ il y a des dizaines d'options supplémentaires possibles :
 - ✘ `-Wall` : affiche tous les warnings,
 - ✘ `-g` : mode debug (permet de mieux suivre l'exécution),
 - ✘ `-O -O2 -O3 -Os` : optimisation de l'exécutable,
 - ✘ `-lm` : inclut la librairie mathématique dans l'exécutable,
 - ✘ `-S` : à la place de `-o hello`, crée `hello.s` en assembleur,
- ✘ Si le compilateur n'affiche rien, tout s'est bien passé,
 - ✘ s'il affiche une erreur, l'exécutable n'est pas créé,
 - ✘ s'il affiche un/des warnings, l'exécutable est créé, mais risque de ne pas fonctionner correctement.

Comment compiler/exécuter un programme ?

- ✘ Voici les commandes que vous devrez taper dans un terminal :

```

> emacs hello.c &
[1] 4470
> gcc -Wall hello.c -o hello
> ./hello
Hello World!

```

- ✘ `emacs` permet de taper le fichier `hello.c`,
 - ✘ vous pouvez aussi utiliser `vim`, ou n'importe quel éditeur,
- ✘ `gcc` (Gnu Compiler Collection) permet de transformer le fichier source `hello.c` en un programme exécutable `hello`,
 - ✘ il existe d'autres compilateurs : `icc`, `VisualStudio`...
- ✘ `./hello` exécute le programme `hello` dans le répertoire courant
 - ✘ on ajoute `./` devant car `.` n'est pas dans le `PATH`.

Les bases du C

Ce que peut contenir un programme en C

```

1 /* Commentaire long */
2 // Commentaire court
3
4 /* Instructions pour le préprocesseur */
5 #include <math.h>           // inclusion
6 #define TAILLE 3           // constante
7 #define SQ(x) ((x)*(x))    // macro
8
9 /* Nouveau type */
10 typedef unsigned long long int ull;
11
12 /* Variables globales */
13 ull globale1;              // déclaration
14 int globale2 = 0;          // decl. + initialisation

```

Ce que peut contenir un programme en C

```

15 /* Des fonctions */
16 int add(int a, int b) {    // prototype (ou en-tête)
17     int c;                 // déclaration de variable locale
18     c = a + b;             // instruction simple
19     return c;              // valeur de retour
20 }
21
22 int main() {
23     int a,b,c;             // déclaration de variables locales
24     a = SQ(5);            // utilisation de la macro
25     b = 17;
26     c = add(a,b);
27     printf("%d + %d = %d\n", a, b, c);
28     return 0;
29 }

```

✘ Après compilation/exécution, cela affichera : 25 + 17 = 42.

La mémoire

- ✘ Avant d'être exécuté, le programme est chargé en mémoire :
 - ✘ un pointeur indique où en est l'exécution
 - à chaque instruction il avance d'une case
 - ✘ il y a parfois des sauts :
 - appels de fonctions, tests, boucles...
- ✘ Pendant l'exécution, le programme peut aussi réserver de la mémoire :
 - ✘ chaque variable correspond à une case mémoire,
 - ✘ les variables globales sont réservées au début du programme,
 - ✘ les variables locales sont réservées au moment de l'entrée dans la fonction et libérées à la sortie
 - plusieurs appels à une fonction ne font pas augmenter la mémoire totale.

Le typage

- ✘ Un octet en mémoire peut avoir différentes significations. L'octet 01010111 peut représenter :
 - ✘ l'entier 87 (= 64 + 16 + 4 + 2 + 1),
 - ✘ le flottant $4.85 \cdot 10^{-270}$,
 - ✘ le caractère 'W',
 - ✘ une instruction en langage machine,
 - ✘ une adresse mémoire...
 - ✘ Il faut associer un type à chaque case mémoire, chaque variable
 - c'est son type qui va définir sa signification réelle
- ⚠ En C, le programmeur peut ne pas respecter le typage :
- ✘ en général le compilateur affiche un warning
 - ✘ l'exécution risque d'échouer...

Les types natifs en C

	taille (bits)	valeur		unsigned	
		min	max	min	max
<i>char</i>	8	-	-	0	255
<i>short</i>	16	-32768	32767	0	65535
<i>int</i>	32	-2^{31}	$2^{31} - 1$	0	$2^{32} - 1$
<i>long</i>	32 ou 64	dépend de l'architecture			
<i>long long</i>	64	-2^{63}	$2^{63} - 1$	0	$2^{64} - 1$

	taille (bits)	valeur		précision
		min	max	
<i>float</i>	24 + 8	$1.1 \cdot 10^{-38}$	$3.4 \cdot 10^{38}$	10^{-6}
<i>double</i>	53 + 11	$2.2 \cdot 10^{-308}$	$1.8 \cdot 10^{308}$	10^{-15}
<i>long double</i>	64 + 16	$3.4 \cdot 10^{-4932}$	$1.2 \cdot 10^{4932}$	10^{-18}

Déclaration et initialisation de variables

```

1 short a = 754;
2 float c;
3
4 int main() {
5     int i,j=3;
6     double b;
7     b = 0.00345;
8     c = 17.3;
9 }
```

- × On **déclare** une variable en donnant son type :
 - × cela réserve une case mémoire,
 - × elle contient une valeur aléatoire.
- × L'**initialisation** fixe sa valeur :
 - × cela peut se faire dès la déclaration,
 - × ou après, c'est équivalent.

- × Le compilateur affiche un warning si on utilise une variable non initialisée :
 - × sa valeur est "ce qu'il y avait dans la mémoire"
 - ça n'a pas de sens de l'utiliser...
 - × en général c'est un oubli...

Déclaration et initialisation de variables

```

1 short a = 754;
2 float c;
3
4 int main() {
5     int i,j=3;
6     double b;
7     b = 0.00345;
8     c = 17.3;
9 }
```

- × On **déclare** une variable en donnant son type :
 - × cela réserve une case mémoire,
 - × elle contient une valeur aléatoire.
- × L'**initialisation** fixe sa valeur :
 - × cela peut se faire dès la déclaration,
 - × ou après, c'est équivalent.

- × Voici à quoi ressemble la mémoire avec ce code :

short a	float c	int i	int j	double b	
754	17.30000000	? ? ? ?	0 0 0 3	0.00345000000000000000	
variables globales		variables locales			

Les opérations de base en C

L'affectation avec =

- × L'opération '=' permet de stocker une valeur dans une variable,
 - × `a = 3;` stocke 3 dans la variable a,
 - × `3 = a;` n'a aucun sens...
 - cela n'a rien à voir avec un égal mathématique !
- × Ce qui est **à gauche du égal** est une 'case mémoire' :
 - × un nom de variable,
 - × une case d'un tableau,
 - × un élément d'une structure...
- × Ce qui est **à droite du égal** est évalué. Cela peut être :
 - × une constante,
 - × une autre variable
 - l'évaluation va lire la valeur qu'elle contient,
 - × un appel de fonction
 - on récupère la valeur retournée par la fonction...

Les opérations de base en C

L'affectation avec =

- ✖ Si les types de la case mémoire à gauche et de l'évaluation à droite ne correspondent pas, il peut y avoir un problème :
 - ✖ soit les types n'ont rien à voir et on a un warning
 - par exemple, un entier à la place d'une adresse mémoire,
 - ✖ soit les types sont compatibles, et une conversion a lieu (avec ou sans warning du compilateur)
 - ⚠ une conversion peut faire perdre en précision.


```
short a; int c = 70000; a = c;
```
- ✖ On peut forcer une conversion de type à l'aide d'un `cast` :
 - ✖ 3.5 est un flottant,
 - ✖ `(int) 3.5` est un entier et vaut 3,
 - ✖ `(char) 517` convertit 517 en `char` sur 8 bits
 - cela vaut 5, car on perd les bits de poids fort.

Les opérations de base en C

Opérations arithmétiques

```

1 int a,b,c;
2 b = 5 + 3;           // addition
3 c = 13 - 5;         // soustraction
4 a = b * c;          // multiplication
5 float d = 15 / 6;   // division euclidienne -> d vaut 2
6 d = (float) 15/6;   // vraie division -> d vaut 2.5
7 d = 15.0/6;         // d vaut 2.5
8 a = c % 3;          // modulo 3 (reste de la division)
9
10 /* opérations logiques */
11 a = 8 & 5;          // ET bit à bit
12 a = 6 | 7;          // OU bit à bit
13 a = 6 ^ 7;          // OU exclusif bit à bit
14 a = ~5;             // NON bit à bit

```

Les opérations de base en C

Opérations arithmétiques abrégés

- ✖ Il existe des versions courtes de certaines des opérations courantes

```

1 int i=0;
2 i++;                // i = i + 1;
3 i--;                // i = i - 1;
4 i += 3;             // i = i + 3;
5 i -= 3;             // i = i - 3;
6 i *= 3;             // i = i * 3;
7 i /= 3;             // i = i / 3;
8 i %= 3;             // i = i % 3;

```

- ✖ Cela change le code, pas le programme final :
 - ✖ ce n'est pas plus rapide à exécuter que la forme longue.

Entrées/sorties élémentaires

La fonction printf

- ✖ La fonction `printf` permet d'afficher sur la **sortie standard** :
 - ✖ un message `printf("Hello World!\n");`
 - ✖ la valeur de variables `printf("%d = %d + %d\n",a+b,a,b);`
 - ✖ Chaque % dans la chaîne à afficher correspond à une variable,
 - ✖ la lettre qui suit le % définit le type/la façon d'afficher.
 - %d un *int*
 - %lld un *long long int*
 - %u un *unsigned int*
 - %x un *int* en hexadécimal
 - %f un *double* (ou un *float*)
 - %e un *double* en notation scientifique
 - %.7f un *double* avec 7 chiffres après la virgule
 - %c un *char* (comme caractère ASCII, pas comme entier)
- Dans un terminal, `man 3 printf` donne la liste complète.

Exécution conditionnelle

IN101 - 2011-2012

- × Les opérations vues jusqu'à présent étaient toujours exécutées dans l'ordre de leur apparition dans le main.
 - × les tests permettent de changer l'exécution en fonction des valeurs entrées.

```
1 if (a > 0) {
2   printf("a est positif.\n");
3 }
4
5 if (a == 0) {
6   printf("a est nul.\n");
7 } else {
8   printf("a est non nul.\n");
9 }
```

- × Il existe plusieurs comparaisons possibles :
 - × (a==b) ou (a!=b) → test d'égalité ou de différence
 - × (a<b) ou (a>b) → comparaison stricte
 - × (a<=b) ou (a>=b) → inférieur/supérieur ou égal
- × Le type booléen n'existe pas en C, on utilise des entiers :
 - × un entier nul signifie faux,
 - × un entier non nul signifie vrai.
- × Les comparaisons renvoient donc un entier nul ou non :
 - × en général, 0 et 1, mais pas forcément,
 - × le `if` est effectué si l'argument est non nul
 - vous pouvez utiliser autre chose qu'une comparaison

```
1 if (7-5) {
2   printf("Ce message sera affiché.\n");
3 }
```

- × Il est possible de combiner plusieurs comparaisons avec des opérateurs logiques :
 - × (a<b)&&(b<c) : ET logique → vrai si les 2 tests sont vrais
 - × (a<b)|| (b<c) : OU logique → vrai si l'un des 2 tests est vrai
 - × !(a<b) : NON logique → vrai si le test est faux
- × Quelques remarques :
 - × il ne faut pas hésiter à ajouter des parenthèses à `a<9 && a>3`
 - `(a<9)&&(a>3)` est différent de `(a<(9&&a))>3`
 - × l'évaluation de `&&` et `||` est paresseuse
 - le deuxième terme n'est pas évalué si le premier suffit à déterminer le résultat. Par exemple :

```
1 if ((a!=0) && (b/a != 0)) { // pas de division par zéro
2   printf("b est supérieur ou égal à a.\n");
3 }
```

- ✘ Pour répéter un segment de code on utilise une **boucle** :
 - ✘ pour un nombre fixé d'itérations
 - boucle **for** (voir cours suivant),
 - ✘ pour itérer tant qu'une condition est vérifiée
 - boucle **while** ou **do-while**.

collatz.c

```

1 int main() {
2   int a = 25;
3   while (a!=1) {      // répéter tant que a est non nul
4     if (a%2 == 0) {   // si a est pair
5       a = a / 2;     // on le divise par 2
6     } else {         // sinon
7       a = 3*a + 1;   // on le rend pair
8     }
9   }
10  return 0;
11 }
```

- ✘ Il existe deux syntaxes pour les boucles while :
 - ✘ while (test) {...}
 - n'exécute le code {...} que si test est vrai.
 - ✘ do {...} while (test);
 - exécute le code {...} toujours au moins une fois.
- ⚠ Attention aux **boucles infinies** si le test est toujours vrai !
 - ✘ le programme ne s'arrêtera pas tout seul,
 - ✘ dans le terminal Contrôle + C permet de "tuer" un programme.
- ✘ Pensez à bien **indenter** votre code :
 - ✘ Tab dans emacs fait ça tout seul,
 - ✘ permet de bien voir ce qui est dans la boucle
 - simplifie la tâche des chargés de TD...
 - ✘ il y a plusieurs conventions d'indentation
 - si vous n'avez pas d'avis, faites comme moi !

- ✘ oubli d'un include (stdio.h pour les printf),
- ✘ utilisation d'une variable non déclarée,
- ✘ faute de frappe dans un nom de variable/fonction,
- ✘ oubli d'un ; en fin d'instruction,
- ✘ test avec = au lieu de == (pas d'erreur à la compilation),
- ✘ mauvais nombre d'accolades fermées,
 - important d'indenter le code correctement,
- ✘ oubli de la fonction main
 - c'est le point de départ de l'exécution, donc indispensable.
- ⚠ Les messages d'erreur de gcc ne sont pas toujours très clairs, mais il est important de bien les lire quand même,
 - ✘ toujours commencer par le premier message d'erreur
 - corriger la faute, puis recompiler.