Introduction au langage C

Système et environnement de programmation

Université Grenoble Alpes

Plan

Origine

Créé en 1972 par K. Thompson, D. Ritchie, B. Kernighan

- Aux débuts d'UNIX (1969) : développement du système
- Encore largement utilisé aujourd'hui
  - Noyau des systèmes d'exploitation
  - Programmes nécessitant une grande efficacité de calcul
  - Systèmes embarqués (peu de mémoire, runtime basique)
- Etendu en C++ par B. Stroustrup en 1983, notamment
  - Orienté objet
  - Exception
  - Polymorphisme paramétré
  - Références

Différences majeures avec Python

Mémoire gérée explicitement

- Variables typées
  - Il faut les déclarer avant toute utilisation
  - Taille fixe occupée en mémoire
  - Ensemble fini de valeurs représentables
- Allocation mémoire explicite
  - Il faut demander au système toute mémoire utilisée (allocation)
  - Aucune structure de données ne grossit toute seule en mémoire (tableaux, listes, ...)
  - Il faut libérer la mémoire qui ne sert plus

Forme et mécanismes un peu différents

- Indentation libre, blocs délimités par {}
- Il faut indenter (devoir pour les relecteurs)
- Non orienté objet

Pourquoi apprendre le C ?

Tout informaticien doit connaître le C

- Pour comprendre comment écrire les choses à plus bas niveau
- Pour savoir comment être efficace lorsque cela est necéssaire
- Parce que, parfois, on a besoin d'être à bas niveau
  - Noyau du système (gestion processus et mémoire, pilotes)
  - Interpréteur python, java, ... (gestion mémoire)

Souvent comparé à un langage d'assemblage portable

- Niveau immédiatement au dessus
- Compilable sur tout système/processeur

Mon premier programme en C

Tout programme doit contenir une fonction main

```
// Un commentaire débute par //
// #include va chercher le contenu d'un fichier
// stdio.h : affichage et lecture standards
#include <stdio.h>
int main() {
    // printf permet d'afficher à l'écran
    // \n est le caractère de fin de ligne
    printf("Hello world\n");
    // Toutes les instructions se terminent par un ;
    return 0;
}
```

Rappel : Dans un environnement UNIX, un programme renvoie :

- 0, si tout s'est bien passé
- Un code d'erreur non nul, en cas de problème

Execution de mon premier programme

Le texte d'un programme en C n'est pas exécutable

- Compilation : texte en C -> exécutable en langage machine
- Langage machine
  - Illisible par un être humain
  - Compris directement par le processeur (efficace)

Dans le cas de mon programme `hello_world.c`

```
clang hello_world.c
```

Produit un fichier exécutable `a.out` en langage machine
Pour l'exécuter :

```
./a.out
```

Plan

Structure conditionnelle

Syntaxe

```
if (<expression>) {
    <bloc instructions 1>
} else {
    <bloc instructions 2>
}
```

Facultatif

- Toute la partie else s'il n'y a rien a faire dans le sinon
- Les accolades si un bloc se réduit à une instruction

```
if (x>0)
    printf("Positif");
else
    printf("Négatif");
```

Boucle while

Syntaxe

```
while (<expression>) {
    <bloc instructions>
}
```

Facultatif

- Les accolades si le bloc se réduit à une instruction

```
printf("Divisons %d par %d : ", a, b);
q=0;
while (a>b) {
    q = q+1;
    a = a-b;
}
printf("on trouve %d reste %d\n", q, b);
```

Boucle for

Syntaxe

```
for (<expr 1>; <expr 2>; <expr 3>) {
    <bloc instructions>
}
```

équivaut à

```
<expr 1>
while (<expr 2>) {
    <bloc instructions>
    <expr 3>
}
```

Exemple

```
for (i=1 ; i<11 ; i++)
    s = s+i;
```

Plan

Le typage

En C, toutes les variables ont un type

- On doit déclarer une variable (et son type) avant de l'utiliser
- Le type explicite la nature et la taille du contenu de la variable
- Le compilateur vérifie le type des variables manipulées
  - Paramètres effectifs des appels de fonctions
  - Opérandes des opérateurs utilisés
- Lorsque les types de correspondent pas
  - Conversion automatique lorsque cela est possible
  - Erreur sinon

Intérêts du typage

- Aide le programmeur en soulignant certaines erreurs, ex :
  - On ne peut pas ajouter une chaine de caractères à un entier
  - Il faut convertir la chaine avant, si elle représente bien un entier
  - Diagnostiqué à la compilation (tôt)
- Représentation compacte et efficace en mémoire
  - Un `int` en C occupe en général 4 octets
  - Un entier en python occupe une place qui dépend de sa valeur
    - Occupe plus de place (taille à stocker)
    - Varie dynamiquement : surcoût à l'exécution
- Evite des conversions à la volée durant l'exécution

Un exemple avec des variables

```
// déclaration d'une variable x dans Z
int x;
// déclaration d'une variable y dans Q
float y;

x = 1000000000;    // Trop gros ou pas ?
y = 1.0/3.0;       // Quelle précision ?
```

Quelle valeurs sont-elles valides ?
Comment sont-elles représentées en mémoire ?
Quelle place occupent-elles ?

Les types entiers

- Les divers types entiers varient en taille (nombre de bits)
- Non standard en C :
  la taille d'un type donné peut varier selon l'architecture/OS
- Il existe des types standard pour préciser la taille exacte de la variable (hors de notre programme)
- Signés par défaut (entiers relatifs) il faut ajouter `unsigned` à la déclaration pour une variante non signée (entier naturel)

Exemple sur `Debian 8.10 / x86_64` (turing)

type	taille sur turing	minimum garanti par le standard
char	8	8
short	16	16
int	32	16
long	64	32
long long	64	64

Intervalles de valeurs

On choisit le type en fonction de la taille de l'entier à représenter

Octets	Bits	Valeurs	En signé	En non signé
1	8	256	[-128 ; 127]	[0 ;255]
2	16	65536	[-32768 ;32767]	[0 ;65535]
4	32	2^{32} (4 milliards)	[-2^{31} ;2^{31}-1]	[0 ;2^{32}-1]
8	64	2^{64} (16 milliards de milliards)	[-2^{63} ;2^{63}-1]	[0 ;2^{64}-1]

Exemple

```c
#include <stdio.h>

int main () {
    short n = 1000000; // Avertissement
    unsigned char p = 255;

    printf("n = %d\n", n);
    p=p+1;
    printf("p = %d\n", p);
    return 0;
}
```

```
n = 16960
p = 0
```

Que choisir ?

- A moins de progrès fulgurants en médecine, l'âge d'un humain pourra être représenté par un `unsigned char`
- Par contre pour compter le nombre d'étudiants dans une université on prendra plutôt un `unsigned short`
- Pour suivre le solde du compte en banque d'une personne, un `int` suffit généralement
- Pour compter le nombre d'être humains sur terre on devra utiliser un `unsigned long long`

Les booléens

En C les booléens **n'existent pas** !
Mais alors quel est le type de l'expression `a <= b` ?

- `int`
- "faux" est représenté par la valeur `0`
- "vrai" est représenté par toute valeur différente de `0`

Opérateurs booléens :

- et booléen `&&`
- ou booléen `||`
- négation `!`

Les types à virgule flottante

- Nombre rationnels caractérisés par une partie entière et une partie fractionnelle
- Représentés en machine sous la forme $mantisse * 2^{exposant}$ où $1 \leq mantisse < 2$
- Valeurs particulières incluses
  - NaN : Not a Number
  - +Inf/-Inf : Plus ou moins l'infini

Sur turing

Type	Octets	Mantisse	Exposant
float	4	23 bits	8 bits
double	8	52 bits	11 bits
long double	16	112 bits	15 bits

Particularités des types à virgule flottante

- Non standard en C
  - Beaucoup de formats coexistent
  - Standard (IEEE 754) parfois absent ou partiellement respecté
- Mantisse et exposant de taille finie
  - intervalle de valeurs fini
  - précision finie
- Beaucoup de rationnels dans l'intervalle de valeurs non représentables exactement
  - 1/3 qui a un infinité de chiffres après la virgule
  - Plus surprenant : 1/10 également car nous sommes en base 2
  - Seulement les nombres dont la mantisse est de la forme

$$1 + \sum_{i=1}^{n} \frac{c_i}{2^i}$$

où $\forall i, c_i \in \{0, 1\}$ et n est le nombre de bits de la mantisse

Calculer avec des nombres flottants

- La précision finie oblige à être conservatif
  - On ne compare pas à 0
  - On teste si la valeur absolue est inférieure à un ϵ fixé
- Les erreurs se propagent, les alternatives sont coûteuses
  - Arithmétique d'intervalles
  - Calcul en précision arbitraire
- En pratique
  - Double précision souvent acceptable pour le calcul scientifique
  - Simple précision pour la cao, les jeux vidéo
  - Demi précision pour certains domaines (deep learning)

Exemple

```
int main() {
    double a = 0.1;
    double b = 0.8;
    double somme = 0;

    for (int i=0; i<8; i++)
        somme = somme + a;
    if (somme == b)
        printf("somme == b\n");
    else
        printf("somme != b\n");
}
```

```
somme != b
```

Plan

La table ASCII

- En C
  - les caractères n'existent pas
  - on ne manipule que des codes entiers qui les représentent
  - traduction à la lecture, l'ecriture ou dans les constantes
- Le standard ASCII associe un caractère à chaque valeur numérique de [0 ;127]
- 128 caractères définis $\Rightarrow$ 7 bits de stockage
- On peut donc stocker cette valeur numérique sur un octet
- 8ème bit utilisé dans des certaines variantes pour représenter des caractères spécifiques à une langue
  - Windows latin-1
  - Mac OS roman

Partie imprimable du code ACSII

	30	40	50	60	70	80	90	100	110	120
0:		(	2	<	F	P	Z	d	n	x
1:		)	3	=	G	Q	[	e	o	y
2:		*	4	>	H	R	\	f	p	z
3:	!	+	5	?	I	S	]	g	q	{
4:	"	,	6	@	J	T	^	h	r	|
5:	#	-	7	A	K	U	_	i	s	}
6:	$	.	8	B	L	V	‘	j	t	~
7:	%	/	9	C	M	W	a	k	u	SUP
8:	&	0	:	D	N	X	b	l	v	
9:	´	1	;	E	O	Y	c	m	w	

Particularités du type `char`

- Type permettant de stocker une valeur entière signée sur un octet ($\in [-128; 127]$)
- On peut aussi y mettre un code ASCII (entre 0 et 127)
- En C, pas de distinction entre un caractère et la valeur entière de son code ASCII
- On lève l'ambiguité lors de la saisie ou l'affichage

Entier ou caractère

Le f de printf signifie "format", indique comment afficher

- %c pour un caractère
- %d pour un entier

Qu'affiche ce code ?

```
char     c = 65;
int      i = 65;
printf("c1=%c, c2=%d\n", c, c);
printf("i1=%c, i2=%d\n", i, i);
```

```
c1=A, c2=65
i1=A, i2=65
```

- Deux fois la même chose !
- Un `int` stocke aussi bien qu'un `char` la valeur numérique d'un caractère (il prend juste plus de place)

Exercice

Ecrire le corps de la fonction C suivante :

```
char capitalize(char c);
```

- Si c est une lettre minuscule la fonction renvoie la lettre majuscule associée
- Si c n'est pas une lettre minuscule, renvoie c

Plan

Déclaration d'un tableau

Un tableau est un ensemble d'éléments consécutifs de même type
La taille d'un tableau :

- est le nombre d'éléments qu'il contient
- est donnée à la déclaration
- ne varie pas au cours de l'exécution
- n'est pas stockée par le C (à nous de le faire)

Déclaration

```
int tableau[10];
```

On accède à un élément à l'aide de son indice dans le tableau

```
tableau[3] = 42;
```

Les indices sont dans $[0; \mathit{taille} - 1]$

Exemple

```
int t[10];

for (int i=0; i<10; i++) {
    t[i] = 1;
    for (int j=i-1; j>0; j--)
        t[j] = t[j] + t[j-1];
    for (int j=0; j<=i; j++)
        printf("%d ", t[j]);
    printf("\n");
}
```

Débordements

Attention, le code suivant marche !

```
int tableau[10];
tableau[99] = 42;
```

- Le C ne vérifie pas que l'indice donné est dans le tableau
- La commande ci-dessus marche
  - Le programme va écrire dans la mémoire à la case demandée
  - C'est au delà de la fin du tableau
- Deux cas de figure pour l'erreur
  - La case est en dehors de la mémoire octroyée à l'utilisateur
    $\Rightarrow$ `Segmentation fault`
  - Sinon, elle contient potentiellement autre chose, son contenu sera écrasé, bug très insidieux
- Pour éviter cela, il faut être attentif, toujours vérifier qu'on accède bien aux cases de 0 à $taille - 1$

Plan

Les fonctions sont typées

Une fonction est caractérisée par

- son nom
- le type de sa valeur de retour (`void` si aucune)
- le type et le nom de chacun de ses paramètres formels

```
int max(int a, int b) {
    if (a < b)
        return b;
    else
        return a;
}
```

Passage de paramètres

Les paramètres d'une fonction sont passés par valeur

- Un emplacement mémoire est créé pour les paramètres formels
- La valeur des paramètres effectifs y est copiée

=> les variables passées à une fonction restent inchangées

```
#include <stdio.h>
void echange(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}
int main() {
    int qd = 42;        int ue = 203;
    echange(qd, ue);
    printf("En %d on parle de %d\n", ue, qd);
    return 0;
}
```

Le cas des tableaux

Les tableaux font exception à la règle et sont passés par adresse

```c
#include <stdio.h>
void echange(int Tab[], int i, int j) {
    int tmp;
    tmp = Tab[i];
    Tab[i] = Tab[j];
    Tab[j] = tmp;
}
int main() {
    // Initialisation de taille et valeurs
    int Tab[] = {42, 203};
    echange(Tab, 0, 1);
    printf("En %d on parle de %d\n", Tab[0], Tab[1]);
    return 0;
}
```

Plan

La base dix

Une valeur entière est représentée dans une base

- Les humains ont dix doigts et comptent donc en base dix
  - on regroupe les unités par paquets de dix, ce qui forme des dizaines
  - on fait de même pour les dizaines, centaines, milliers, etc.
  - les restes de tous ces regroupements nous donnent les chiffres d'un nombre
  - ce choix de regrouper en paquets de 10 n'est qu'une convention pour représenter la valeur d'un nombre
- Plus formellement, un nombre n sera représenté en base 10 par des chiffres $\{c_0, \ldots, c_m\}$ tels que

$$n = \sum_{i=0}^{m} c_i * 10^i$$

Changement de base

Changement dans la manière de représenter un nombre n

- en base 10 c'est la séquence de chiffres $\{c_0, \ldots, c_m\}$ chacun dans $\{0, \ldots, 9\}$ tels que

$$n = \sum_{i=0}^{m} c_i * 10^i$$

- en base b c'est la séquence de chiffres $\{c_0, \ldots, c_k\}$ chacun dans $\{0, \ldots, b-1\}$ tels que

$$n = \sum_{i=0}^{k} c_i * b^i$$

En général, pour faciliter l'écriture, on fait correspondre à chaque valeur dans $\{0, \ldots, b-1\}$ un unique caractère.

Bornes et intervalle de valeurs

En base 10, sur 4 chiffres, on peut écrire les entiers naturels

- de 0 à 9999 ($10^4 - 1$)
- cela fait 10^4 valeurs différentes

En base b, sur n chiffres, on peut écrire les entiers naturels

- de 0 à $b^n - 1$
- cela fait b^n valeurs différentes

Le binaire : la base 2

- Information élémentaire stockée sous la forme de bits pouvant prendre deux valeurs (signal ou pas, 0 ou 1)
- On assemble une séquence de plusieurs de ces bits pour représenter une information plus complexe
- La base 2 est la base dans laquelle le processeur travaille

chiffres utilisés :

`0 1`

Exemples

Décimal	0	1	2	3	5	7	11
Binaire	0	1	10	11	101	111	1011

L'octal : la base 8

- Un chiffre octal correspond à 3 chiffres binaires
- On l'utilise pour les droits Unix (`chmod 764`)

chiffres utilisés :

`0 1 2 3 4 5 6 7`

Exemples

Décimal	0	1	2	3	5	7	11	13	17	19
Octal	0	1	2	3	5	7	13	15	21	23

L'hexadécimal : la base 16

- Un chiffre hexadécimal correspond à 4 chiffres binaires
- Permet une représentation compacte de données en mémoire : 4 octets valant `11011110101011011011111011101111` se représentent mieux en `DEADBEEF`

chiffres utilisés :

`0 1 2 3 4 5 6 7 8 9 A B C D E F`

Exemples

Déc.	0	1	2	3	5	7	11	13	17	19	23	29	31
Hexa.	0	1	2	3	5	7	B	D	11	13	17	1D	1F

Exemple

Parmi les nombres suivants

```
10        14        ADA        78        0
```

- 2 sont valides en binaire
- 3 sont valides en octal
- 4 sont valides en décimal
- 5 sont valides en hexadecimal

Exercice

Combien de valeurs peut-on représenter avec les 5 doigts d'une main ?

Exercice

Combien de valeurs peut-on représenter avec les 5 doigts d'une main ?

Une première idée serait 6 (de 0 à 5 doigts levés), cela correspond à un codage dit "batonnets"

Exercice

Combien de valeurs peut-on représenter avec les 5 doigts d'une main ?

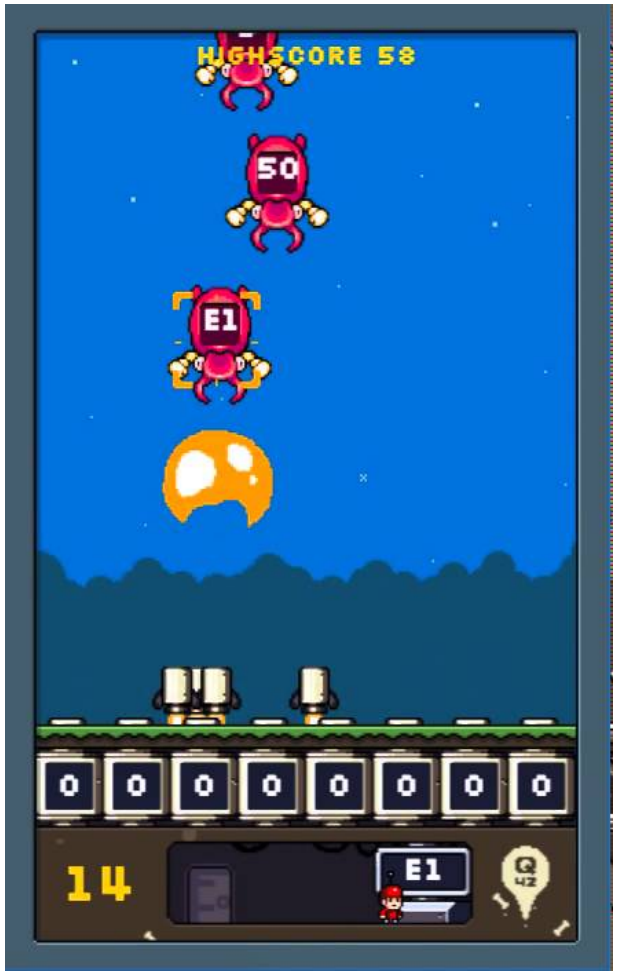
Une première idée serait 6 (de 0 à 5 doigts levés), cela correspond à un codage dit "batonnets"

Mais en étant bien souple, avec la base 2, on arrive à 2^5

- Premier doigt plié ou non
- Pour chacune de ces deux possibilités on a le nombre de valeurs représentables sur 4 doigts

Pour s'exercer un peu plus

- Les conversions de base c'est utile et super rigolo
- Mais pour ceux qui trouveraient ça ennuyeux, il y a Flippy bit
  - Application Android
  - Il faut repousser une invasion d'aliens hexadécimaux
  - On tire en traduisant en binaire



Notations en C pour les constantes

```
int n1 = 0b10;      // binaire      (vaut 2)
int n2 = 010;       // octal        (vaut 8)
int n3 = 10;        // decimal      (vaut 10)
int n4 = 0x10;      // hexadécimal  (vaut 16)
```

- Attention ! La notation C pour l'octal est très dangereuse, il est facile de la confondre avec du décimal
- Ces notations se retrouvent dans de nombreux autres langages

Affichage en C selon la base

A l'aide de la fonction printf en utilisant le bon format

- %o : octal
- %d : décimal
- %x : héxadécimal
- pas d'affichage en base 2

```
int n = 42;

printf("octal : %o, décimal : %d, "
       "héxadécimal : %x\n", n, n, n);
```

```
octal : 52, décimal : 42, héxadécimal : 2a
```