Programmation impérative

en langage C

année universitaire: 2021 -2022

Table des matières

I. Programmation impérative (langage C)

Table des matières

II. Tableaux, structures, pointeurs et listes chaînées

Table des matières

III. Compléments et autres types de données

Plan

Architecture de *von Neumann*

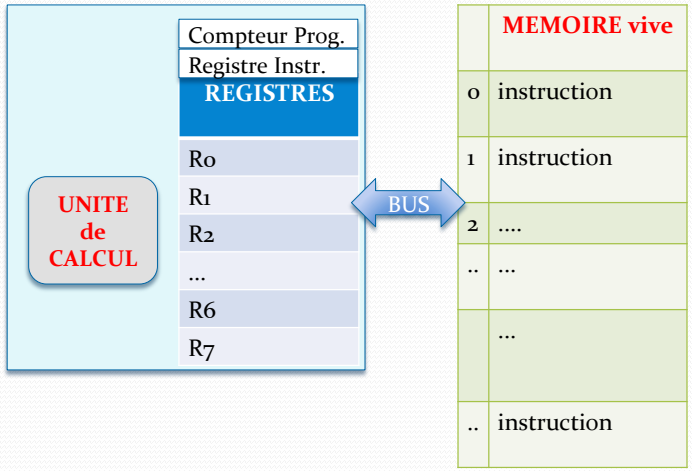
John William Mauchly et *John Eckert* y ont aussi beaucoup contribué. Cette machine, ancêtre de nos processeurs, permet d'exécuter un programme stocké en mémoire. Elle est faite :

- d'une Mémoire (de cases numérotées)
- d'une Unité de Calcul (UC)
- de registres numérotés + (CP) un compteur de programme + (RI)un registre d'instruction
- d'un bus (circuits pour les échanges entre les registres et la Mémoire)

Plan

Architecture des ordinateurs
Variable et affectation
Structure d'un programme C
Fonctions
Types primitifs
Expressions et Instructions
Compilation

Architecture de *von Neumann*



Plan

Architecture des ordinateurs
Variable et affectation
Structure d'un programme C
Fonctions
Types primitifs
Expressions et Instructions
Compilation

Architecture de *von Neumann*

Un programme (son code exécutable) est constitué d'un ensemble d'instructions codées en binaire (langage machine).

Pour être exécuté, il sera chargé en mémoire vive.

Il va s'exécuter en réitérant un cycle d'échanges entre la mémoire, l'unité de calcul et les registres.

Plan

Architecture des ordinateurs
Variable et affectation
Structure d'un programme C
Fonctions
Types primitifs
Expressions et Instructions
Compilation

Mémoire

La mémoire d'un ordinateur est constituée d'unités élémentaires, les bits.

Un *bit* (de **b**inary un**it**) vaut `0` ou `1`. Un ensemble de 8 bits consécutifs s'appelle un *octet*.

Le *byte* est la plus petite unité de mémoire adressable (en général 8 bits).

Plan

- Architecture des ordinateurs
- Variable et affectation
- Structure d'un programme C
- Fonctions
- Types primitifs
- Expressions et Instructions
- Compilation

Niveaux de langages

Les cases mémoires et les registres contiennent des mots mémoire : des suites de n bits, où n est fixé par l'architecture matérielle (souvent 16, 32, ou 64 bits).

Les instructions du langage machine sont écrites en binaire sur un mot.

Par exemple:

`111 000010` (ici, mot de 8 bits)

Les 3 premiers bits permettront de coder l'instruction (par exemple ajouter au registre R1, le contenu de la mémoire dont le numéro de la case est indiqué par les 5 derniers bits).

Plan

- Architecture des ordinateurs
- Variable et affectation
- Structure d'un programme C
- Fonctions
- Types primitifs
- Expressions et Instructions
- Compilation

Niveaux de langages

Les suites d'instructions binaires de la mémoire sont écrites en langage machine.

Un langage assembleur est un langage machine, traduit de manière symbolique pour être compris par des humains.

Ainsi, au lieu de noter l'instruction précédente

```
1 1 1  0 0 0 1 0
```

on l'écrira par exemple

```
add r1 2
```

signifiant ajouter à `r1` le contenu de la mémoire d'adresse `2`.

Plan

- Architecture des ordinateurs
- Variable et affectation
- Structure d'un programme C
- Fonctions
- Types primitifs
- Expressions et Instructions
- Compilation

Langage de programmation

Bien qu'écrit de manière symbolique, un langage assembleur ne permet pas de déclarer des variables. Le type d'instruction manipulée est un simple jeu d'opérations arithmétiques entre la mémoire, les registres, et l'unité de calcul.

Un langage de programmation comme le C est, par contraste, dit de haut niveau, ou évolué, car il permet de déclarer des variables, des fonctions, et il possède des instructions permettant des enchaînements d'instructions plus sophistiqués que la séquence ou les sauts.

Plan

- Architecture des ordinateurs
- Variable et affectation
- Structure d'un programme C
- Fonctions
- Types primitifs
- Expressions et Instructions
- Compilation

Langage impératif

Le langage C est un langage **impératif**.

Les langages impératifs sont basés sur la notion d'action exécutée, l'action la plus typique étant **l'affectation d'une valeur à une variable**.

Un programme impératif consiste en une suite d' instructions données à la machine, et exécutées les unes après les autres. D'où le nom de langage impératif car il consiste à donner des ordres.

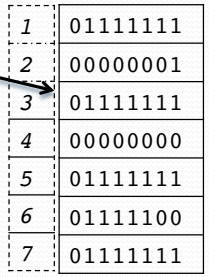
Notion de variable et d'affectation

année universitaire: 2020 -2021

Notion de variable

Une variable est une « case » ou zone mémoire dont on connait l'adresse de début, et la taille (longueur en nb de bytes) qui dépend du type de ses valeurs.

case d'adresse (notée ici n° 3)



Type et taille de variable

Le type d'une variable définit les valeurs abstraites que la variable peut prendre. Ces valeurs sont codées par une suite de bits, dont la taille (en bytes) dépend du type, et qui débute à l'adresse de la variable.

Un type a donc une taille que l'on obtient par l'opérateur `sizeof`.

On peut appliquer l'opérateur `sizeof` à un type ou à une variable.

Déclaration de variable en C

On déclare une variable par son **type** et un **identificateur** (= son nom dans le texte source) suivi d'un point-virgule :

```
/* variables de types numériques */
short int angle_mort ;
unsigned long grandeDistance ;
float taille;

/* variable de type caractère */
char lettre ;
```

Un identificateur est un mot constitué de lettres de l'alphabet ou de chiffres ou encore du caractère souligné (*underscore* en anglais).

Cependant, un identificateur ne peut pas commencer par un chiffre, et certains mots ne sont pas des identificateurs valides car ce sont des mots réservés du langage. (Ceux qui permettent d'écrire un programme).

Mots réservés du langage C

En rouge ici, les mots utilisés pour les types simples.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Résumé: une variable possède

- **un identificateur** : le nom de la variable (e.g. x)
- **un type** : celui des valeurs qu'elle peut prendre.
- **une adresse** : sa place en mémoire vive.

  On l'obtient avec &x. Elle sera allouée soit *dynamiquement* durant l'exécution, soit *statiquement* à la génération du code.
- **une valeur**. La valeur est codée par une suite de bits débutant à l'adresse de la variable.

Pour une variable `x` de type `type`

`sizeof(type)` ou `sizeof(x)` renvoie le nombre de bytes utilisés pour coder la valeur de x.

L'affectation

L'affectation permet d'attribuer une valeur à une variable. Elle utilise le signe égal =, mais ce n'est pas une égalité. On lira ce signe « reçoit ».

$expr_{gauche}$ = expr

ex:
```
angle = 45        x = (y+7)/5
tab[2] = 128      *pt = 100
```

L'expression à gauche du signe d'affectation doit désigner une adresse mémoire. Ce sera généralement un identificateur de variable, ou la n^{ieme} case d'un tableau, ou la valeur d'un pointeur.

-> (4+ 3) = 2 n'est pas une affectation valide.

Plan

Initialisation

Après avoir déclaré une variable, on peut lui affecter une première valeur pour *l'initialiser*.

```
1  int x ;
2  x = 456;
```

On peut aussi l'initialiser dans sa déclaration

```
1  int x = 456;
2  for (int i=0; i<x ; i=i+1)
3        printf("i vaut %d\n", i);
```


Structure d'un programme C

année universitaire: 2020 -2021

Plan

Structure de programme C

Un programme C est une suite de déclarations de variables et de fonctions. Il peut aussi contenir des définitions de types et de constantes. En fait c'est une suite de définitions et de déclarations de noms (de variables, de types, de constantes, etc.).

La fonction principale, appelée `main` , fournit le bloc d'instructions à exécuter. Sa définition est nécessaire pour que le compilateur puisse produire un code exécutable.

Plan

Le langage C propose une librairie standard de fonctions `stdlib` qui implémente des opérations courantes et contient des déclarations de constantes et de types d'utilité générale.

Il existe d'autres bibliothèques de fonctions, comme la librairie standard `stdio` d'entrées/sorties, ou la librairie de fonctions mathématiques `math`.

Pour utiliser les fonctions d'une librairie, il faut inclure le fichier de leurs déclarations, ex: `#include <stdlib.h>`

```
1   #include <stdlib.h>
2   #include <stdio.h>
3
4   int main ()
5   {
6     printf("Bonjour !");
7     return EXIT_SUCCESS;
8   }
```

```
--------
Ce programme écrit  Bonjour ! sur la
console et retourne une valeur
constante (ici 0) indiquant que l'on
quitte le programme avec succès.
```


La **définition** d'une fonction est donnée par la déclaration de son prototype suivie d'un **bloc d'instructions**, entourées d'accolades et séparées par des points-virgules :

```
1  #include <stdlib.h>
2  #include <stdlib.h>

3  int main (int argc, char *argv[])
4  {
5     printf("Bonjour !") ;
6     return EXIT_SUCCESS ;
7  }
```


Bloc d'instructions

Les instruction sont soit des instructions simples (comme l'affectation) ou composées (comme un if-else ou un bloc), soit des expressions représentant une valeur, soit l'instruction `return`.

L'instruction `return` a pour effet d'interrompre la fonction en cours d'exécution et de rapporter la valeur de l'expression spécifiée après le `return` (si rien n'est spécifié c'est la valeur vide `void` qui est rapportée par la fonction).

Un bloc d'instructions peut aussi contenir des déclarations de variables qui n'existent que dans le bloc et dont l'adresse est allouée dynamiquement à chaque exécution du programme.

On dit que ces variables sont des variables locales.

Mais si des variables sont déclarées en tête de fichier, ce sont des variables globales. Elles auront alors une adresse statique (invariable) dans le code, et elles sont accessibles par toutes les fonctions définies dans le fichier.

```
1   #include <stdlib.h>
2   #include <stdio.h>
3   int x = 100; /* x est globale */
4   int main ()
5   {
6     int y = 0 ; /* y est locale */
7     printf("adresse de x: %p",&x);
8     printf("adresse de y: %p",&y);
9     return EXIT_SUCCESS;
10  }
--------
```

Contrairement à celle de x, l'adresse de y peut varier à chaque nouvelle exécution du programme.


```
1   #include <stdlib.h>
2   #include <stdio.h>
3   int x = 100; /* x est global */
4   int main ()
5   {
6     int x = 0 ; /x locale */
7     printf("valeur de x: %d", x);
8     return EXIT_SUCCESS;
9   }
--------
```

Ce programme imprime `valeur de x=0`

car une variable locale masque une variable de même nom déclarée à l'extérieur du bloc.

Exécution en mémoire vive

Variables locales, paramètres et infos sur les fonctions

Allocation dynamique

Variables globales et statiques

Contenu des fonctions

Exécution en mémoire vive

Ce qu'il faut retenir du schéma précédent est que les zones de mémoire dynamiques (le tas et la pile d'exécution) ont des dimensions finies.

Il peut donc y avoir des erreurs d'allocation dynamique de mémoire si on fait trop d'appels de fonctions (débordement de la pile) ou si on stocke trop de données dans le tas.

Structure d'un fichier C avec main

```
/* inclusion de fichier.h  avec #include  */
/* définitions (constantes ou macros) avec #define  */
/* déclarations de type de données avec typedef */
   /* déclarations de fonctions utilisateur */
/* déclaration de variables globales */
int main(int argc, char* argv[]) {
   /* déclaration de variables locales */
   ...
}
   /* définitions de fonctions utilisateur */
```

```
   /* définitions de constantes */
#define  EXIT_SUCCESS    0
#define  BEEP     '\007'
#define  PI   3.1416

   /* définitions de macros */
#define  ABS(x)   ((x)>0 ? (x) : -(x))
```

Attention aux effets de bord ! Bien parenthéser les expressions (à cause des priorités des opérateurs). Mais cela ne suffit pas toujours

ABS(y++) fera ici deux incrémentations sur y.

Fonctions

Déclaration de fonction

Une **déclaration de fonction** indique l'identificateur (=le nom) de la fonction, le type de la valeur retournée par la fonction, et le type des arguments (ou paramètres) de la fonction.

C'est son prototype (ou signature) :

```
int somme (int x, int y)
void imprime()
```

L'absence de paramètres indique que la fonction ne prend pas d'arguments. Si le type retourné est `void` la fonction est une **procédure** (= bloc d'instructions qui ne retourne pas de valeur).

Définition de fonction

Une **définition de fonction** est constituée de la **déclaration de la fonction suivie du bloc (son corps)** définissant les instructions que la fonction doit exécuter lors d'un appel.

```
1  int somme (int x, int y)
2  {
3    return (x + y) ;
4  }
```


Appel de fonction

L'appel d'une fonction est constituée du nom de la fonction suivie de deux parenthèses, où peuvent figurer des expressions séparées par des virgules. Ce sont **les arguments d'appel** qui correspondent aux valeurs prises par les paramètres formels (s'ils existent). Exemples:

```
somme(12, 5*7);
imprime();
```

sont des appels à la fonction `somme` et à la fonction `imprime`.

Appel de fonction

Le calcul de **la valeur retournée par un appel de fonction** s'effectue de la façon suivante :

Si la définition de la fonction a des paramètres formels, alors des variables locales de même nom sont allouées *dynamiquement* (dans la pile d'exécution) et initialisées avec les valeurs des expressions figurant dans l'appel.

Le bloc de définition de la fonction est alors exécuté, et la valeur de l'appel est retournée par une instruction `return`.

Exemple

```
1  int x;
2  x = somme (12, 5*7);
```

`12` et `5*7` sont les arguments d'appel de la fonction `somme`. Des variables locales `x` et `y` sont allouées dans la pile d'exécution (pour instancier les paramètres formels), initialisés à `12` et à `35`, et le bloc de définition de `somme` est exécuté.

Quand `return(x + y)` est exécuté, l'environnement local est dépilé, et la valeur `47` est retournée comme valeur de l'appel (ligne `2`). `47` est alors affecté à la variable `x`.

Résumé

Seules les valeurs des arguments d'appel d'une fonction sont transmises aux paramètres formels d'une fonction pour son exécution.

- Le bloc de calcul de la fonction est exécuté dans cet environnement (qui disparait ensuite).
- Si une variable figure comme argument d'appel d'une fonction, ce n'est que sa valeur qui est transmise pour le calcul.

On dit qu'en C, **le passage des arguments** de l'appel d'une fonction (aux paramètres formels) **s'effectue par valeur**.

-> Une variable de l'environnement initial (celui de l'appel) ne sera jamais modifiée par le calcul quand elle est passée en argument d'appel.

Mais attention : si on transmet la valeur d'une adresse, la variable située à cette adresse est accessible, et sa valeur pourra être modifiée. (Ce sera le cas pour les variables de type tableau).

L'opérateur d'adresse &

Placé devant une variable, il désigne l'adresse en mémoire de la variable

On l'appelle *opérateur d'adresse* (ou opérateur de déréférence car une variable "référence" une valeur)

Il se prononce «et commercial», «esperluette» ou «adresse de»

L'opérateur étoile *

C'est l'opérateur inverse de l'opérateur d'adresse &

On a

valeur de x = *(&x)

Appliqué à une adresse de variable, il désigne la valeur de la variable stockée à cette adresse

[Appliqué à un pointeur il désignera la valeur pointée par ce pointeur]

Appels à f(x) ou f(&x)

- **f (x) : passage d'une variable x par valeur.**
La variable x elle-même n'est pas modifiable par le calcul de f.

- **f (&x) : passage d'une variable x par référence (ou par adresse)**
La valeur de la variable x peut être modifiée par le calcul de f.

Echange des valeurs de 2 variables

Pour échanger les valeurs de deux variables x et y, il faut utiliser une troisième variable. En effet, si l'on écrit directement

```
x = y;
y = x;
```

ça ne marche pas, car la valeur de x aura été perdue dès la première affectation (elle est « écrasée » par la valeur de y).

Echange des valeurs de 2 variables

```
int x, y, sauve;

sauve = x;
x = y;
y = sauve;
```

sont des lignes de C qui permettent de programmer l'échange des valeurs de x et de y.


```
void echange(int a, int b) {
   int sauve;
   sauve = a;
   a = b;
   b = sauve;
}
int main() {
   int x=0, y=4;
   echange (x,y);
   printf( "x=%d, y=%d », x,
y);
}
```

```
void echange(int *a, int *b) {
   int sauve;
   sauve = ...;
   ...
   ...    = sauve;
}
int main() {
   int x=0, y=4;
   echange (&x, &y);
   printf( "x=%d, y=%d », x,
y);
}
```



```
void echange(int *a, int *b) {
   int sauve;
   sauve = *a;
   *a   = *b;
   *b = sauve;
}
int main() {
   int x=0, y=4;
   echange (&x, &y);
   printf( "x=%d, y=%d », x,
y);
}
```


Passage de variable par référence (ou par adresse)

Avantages :

- **efficacité** en cas de variable structurée de valeur longue à recopier (cas des tableaux, des listes, etc.).
- **permet de modéliser des fonctions multiples**, c'est-à-dire des fonctions qui retournent non pas une seule, mais plusieurs valeurs.

Modèles mathématiques de fonction

- Fonction mathématique usuelle
  y = f(x) ;
- Fonction de plusieurs arguments
  y = f (x1, ..., xN) ;
- Fonction qui renvoie plusieurs valeurs y1, ..., yN

  (void) f (x1, ... , xN, &y1, ..., &yN) ;

Passage de variable par référence (ou par adresse)

Inconvénients :

- **complexité syntaxique**: opérateur &, opérateur *, etc.
- **effets de bords possibles** et non visibles, car les affectations sur la variable ne sont pas visibles depuis le texte du programme principal.

Pour rendre visible le fait qu'une variable rapporte une valeur et sera modifiée en retour de fonction :

- Ajouter _return (ou _out) au nom du paramètre formel :

```
int  LookupDefColor (char *name,
                     Color
*def_return);
```


Types primitifs

(ou types scalaires)

Types entiers

On dispose en C de plusieurs types d'entiers :

`int`	entiers signés
`unsigned int`	entiers >= 0
`long int (ou long)`	entiers longs
`unsigned long`	entiers longs >= 0
`short int (ou short)`	entiers petits
`unsigned short int`	entiers >=0 petits

La taille d'un `int` varie selon les ordinateurs, mais on aura souvent 8 bits pour les `short`, 16 (ou 32) bits pour les `int`, et 32 (ou 64) bits pour les `long`.

Si les short sont stockés sur 8 bits, ils permettent de coder les entiers relatifs appartenant à [-128, 127] ; et les unsigned short int permettent de coder les entiers positifs de 0 à 255.

Pour connaître la taille d'une variable ou d'un type, on peut utiliser l'opérateur sizeof qui retourne la taille en byte (1 byte = 8 bits).

ATTENTION aux débordements: les bits supplémentaires seront perdus et les calculs peuvent être faux.

Taille des entiers

Type	Taille mémoire	Domaine des valeurs
char	1 byte	-128 à 127 ou 0 à 255
unsigned char	1 byte	0 à 255
signed char	1 byte	-128 à 127
int	2 ou 4 bytes	-32 768 à 32 767 ou -2 147 483 648 à 2 147 483 647
unsigned int	2 ou 4 bytes	0 à 65 535 or 0 à 4 294 967 295
short	2 bytes	-32 768 à 32 767
unsigned short	2 bytes	0 à 65 535
long	8 bytes	-9223372036854775808 à 9223372036854775807
unsigned long	8 bytes	0 à 18446744073709551615

Types réels

La représentation utilisée est *en virgule flottante* et est constituée d'un signe, d'une mantisse et d'un exposant :

$$1.3254 = +13254 \times 10^{-4}$$

En machine, on utilise la base 2 et on encode l'exposant + un décalage sur un certain nombre de bits, donnant lieu à une certaine précision:

signe x *mantisse* × $2^{(exposant - décalage)}$

Types de variables réelles

On peut ainsi coder en C des nombres réels en fait rationnels, appelés *flottants*.

Il existe trois types de flottants, de la plus faible à la plus grande précision: `float`, `double`, et `long double`.
On peut aussi les préfixer de `unsigned`.

La précision utilisée pour chacun de ces types dépend de l'implémentation.

float représente des valeurs flottantes en simple précision sa taille est généralement de 32 bits.

le type double représente des valeurs à virgule flottante en double précision. Sa taille est de 2*sizeof(float).

Pour résumer, en général

Type	mot machine de 32 bits	mot machine de 64 bits	
char	8 bits (1 byte)	8 bits	
short	16 bits (2 bytes)	16 bits	
int	32 bits (4 bytes)	32 bits	
long	32 bits (4 bytes)	64 bits	
float	32 bits (4 bytes)	32 bits	
double	64 bits (8 bytes)	64 bits	

Des constantes sont définies dans les fichiers limits.h et float.h de la librairie standard pour indiquer les tailles et précisions des variables numériques.

En général

Type	Taille	Valeurs	Précision
float	4 bytes	1.2E-38 à 3.4E+38	6 décimales
double	8 bytes	2.3E-308 à 1.7E+308	15 décimales
long double	10 bytes	3.4E-4932 à 1.1E+4932	19 décimales

Précision des flottants

Le codage en virgule flottante permet de coder de plus grands réels, mais cela au détriment de de la précision.

On utilisera préférentiellement le type `double` pour avoir une meilleure précision.

En effet, la simple précision des floats est très insatisfaisante et peut conduire par exemple à trouver `10` fois la somme de `0.1` égale à `0.999999` au lieu de `1.0`.

Type char

Le type `char` désigne les caractères. Ils étaient encodés par le code américain standard `ASCII` (sur 8 bits) mais le sont souvent aujourd'hui avec `unicode` (sur 16 bits).

Les constantes de type `char` sont notées entre apostrophes (*quote*):

'a' désigne le caractère a.

'\012' désigne le caractère de code octal 12, c'est-à-dire 10.

'\n' désigne le caractère *return*

Code ASCII

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~

Formats d'impression de `printf`

Format	Sortie	Exemple
%d ou %i	Entier décimal signe	392
%u	Entier décimal non signe	7235
%o	Octal non signe	610
%x	Entier hexadécimal non signe	7fa
%X	Entier hexadécimal non signe	7FA
%f	Décimal virgule flottante, minuscules	392.65
%F	Décimal virgule flottante, majuscules	392.65
%e	Notation mantisse/exposant minuscule	3.9265e+2
%E	Notation mantisse/exposant majuscule	3.9265E+22
%g	Utilise le format le + court: %e ou %f	392.65
%G	Variante avec majuscules: %E ou %F	392.65
%c	Caractère (char)	a
%s	Chaine de caractères	bonjour
%p	Adresse (pointeur)	b8000000

Formats d'impression

A noter que pour les flottants, on peut indiquer le nombre de caractères figurant avant et après la virgule, comme `%3.1f` pour obtenir l'affichage `392.6` au lieu de `392.65`.

La plupart des formats d'impression de `printf` dans le tableau précédent restent valables comme format de lecture pour `scanf`.

Formats de lecture avec `scanf`

Ainsi `%e`, %f, et `%g` permettent de lire des `float`, et `%lf` des `double`.

A noter que si l'on préfixe l'indicateur de format d'un nombre, cela précise le nombre de caractères lus par `scanf`.

Mais si l'utilisateur tape plus de caractères, les caractères supplémentaires restent dans le tampon qui lit les entrées et seront présentés à la lecture suivante.

Conversions implicites

Le compilateur fait des conversions de types dans les opérations sur les variables (dans le but de produire un code exécutable).

Exemples:

```
1.    double x;
      int y = 3.4; /* double -> int */
      x = y + 4.5; /* y == 3 */
```

Puis l'addition entre un objet de type `int` et un `double` conduit à calculer une valeur `double` (ici `7.5`) qui sera affectée à `x`.

Conversions implicites

```
2.    y = y + 4.5;
```

Cette fois, la valeur `double` (`7.5`) sera convertie en le type de `y`, ici `int`.

Il y aura suppression de bits et `y` se verra attribuer la valeur entière `7`.

Conversions implicites

Les règles du C concernant les opérateurs (+,-,*,/) entre valeurs de différents types sont de convertir l'opérande du type le plus « étroit » en celui du type le plus « large », et de retourner une valeur du type le plus large.

Les inclusions de types, en allant du plus étroit vers le plus large, vont dans cet ordre :

char short int long float double long double

Le C autorise aussi des inclusions de types `signed` ou `unsigned`, mais les résultats peuvent être parfois étranges et sont dépendants de la plateforme.

Conversions forcées: cast

On peut aussi forcer la conversion d'un type en faisant une conversion explicite (ou conversion forcée), appelée *cast* ou *casting* en anglais. On place alors le type souhaité entre parenthèses devant l'expression à convertir :

```
int  x =  (int) 3.8;
```

Ici, le programmeur force la conversion de `3.8` en `int`.

Sans cette conversion forcée, le compilateur aurait émis un *warning*.

Plan

Conversions forcées: cast

Quelque soit le type de conversion effectuée dans le code (explicite ou implicite), la conversion peut se faire au détriment des données.

Ainsi, un `double` converti en `short` ou en `char` sera tronqué sur les bits de poids forts – ce qui ne correspondra pas toujours à prendre la partie entière du nombre `double` correspondant. Même problème dans une conversion de `long` en `char`.

Expressions et instructions

année universitaire: 2020 -2021

Plan

Les expressions

Une **expression** est une **instruction qui représente** ou retourne **une valeur**.

Il y a toutes sortes d'expressions en C: les constantes, les variables, les expressions arithmétiques, les expressions logiques, mais aussi l'affectation, les appels de fonction, etc.

Les autres instructions modifient l'ordre d'exécution des instructions mais ne rapportent pas elle-même de valeur. Ce sont des **instructions de contrôle**.

Plan

Expressions constantes

Nous avons déjà introduit des constantes.

Constantes entières

- décimale: `372`
- hexadécimale: `0x5a2b`, `0X5a2b` ou `0X5A2B`
- octale: `0477` `010` (= 8 et non pas 10)

On peut les suffixer avec u ou U, et l ou L, pour indiquer `unsigned` et/ou `long`.

Constantes de type `char`

- `'a'`, `'0'`, ou `'\007'`, `'\0'`, `'\13'`, etc.
- `'\n'`, `'\t'` , etc.
- `'\''`, `'\\'`, `'\"'` ou `'"'`, `'\?'` ou `'?'`

etc.

Notation BNF

La syntaxe d'un identificateur *ident* est définie ici avec le symbole ::= dans la notation BNF (pour *Backus, Naur Form*) par :

ident ::= lettre (lettre | chiffre)*
lettre ::= **a** | **b** |… |**z** | **A**| … |**Z** |**_**
chiffre ::= **0** | **1** | **2** | … | **9**

exemples: x, _BiDoN, jour_ferie

Notation BNF

(a) : les parenthèses permettent de délimiter une construction pour lui appliquer un opérateur.

a* : est la répétition de a un certain nombre de fois, éventuellement zéro.

a+ : est la répétition de a au moins une fois.

a | b : est soit a, soit b.

[a] : a entre crochets signifie que a est optionnel et apparaît 1 fois ou pas du tout.

Les symboles terminaux sont écrit dans une typographie différente des autres. On a utilisé le rouge et le gras dans ce qui précède.

Expressions arithmétiques

Les expressions arithmétiques suivent la syntaxe suivante:

expr-arith ::= expr op-binaire expr
| op-unaire expr
op-binaire ::= + | - | * | / | %
op-unaire :: = - | +

ex: `3 + angle*7`
`8 % 3`
`-(3*16)`

Expressions logiques

Les expressions booléennes peuvent se combiner avec les opérateurs logiques "et" et "ou". La table suivante indique les valeurs de vérité de leurs combinaisons :

P	Q	P et Q	P ou Q
F	F	F	F
F	V	F	V
V	F	F	V
V	V	V	V

On peut aussi leur appliquer l'opérateur de négation qui inverse leur valeur.

Expressions logiques

Le C n'a pas de type booléen explicite.

A la place, une valeur entière est interprétée comme une valeur booléenne de vérité :

- si une expression a pour valeur 0, elle aura comme valeur booléenne la valeur *false* (=FAUX).
- si une expression a une valeur différente de 0, elle aura pour valeur booléenne la valeur *true* (VRAI).

Expressions logiques en C

Les expressions logiques suivent la syntaxe suivante (notation BNF):

```
expr-logic  ::= expr op-log expr
              | op-neg expr
              | expr op-comp expr
op-log      ::= && | ||
op-neg      ::= !
op-comp     ::=  > | < | >= | <=
              | == | !=
```

```
ex: x > 0     (x >=0) && (x < 100)
   !(x>100)   (x > 0) || (x < -2)
   !( (x >=0) && (x < 100) )      !x
```


L'instruction `if`

L'instruction `if` est une instruction de contrôle. Elle modifie l'ordre d'exécution des instructions selon que l'expression placée entre parenthèse après le `if` est vraie ou non.

if-cond ::= **if** (expr) instr [**else** instr]

Ex: Le if (sans le else, en réalité optionnel) :

```
1    int x =68;
2    if (x%2 == 0)
3         return x/2;
4    .../* suite du programme */
```


L'instruction `if`

Avec un else :

```
1    int x =68;
2    if (x%2 == 0)
3         x = x/2;
4    else
5         x = (x+1)/2;
6    return x;
```

L'instruction `if`

En fait un bloc d'instructions est une instruction particulière et on peut avoir :

```
1    x = 0xffabcd;
2    if (x%2 == 0)
3          x = x/2;
4    else
5    {
6          x = x + 1;
7          return x/2;
8    }
9    return x;
```


Indentation

L'important est d'aligner les accolades fermantes des blocs avec le if et le else

```
1    x = 0xffabcd;
2    if (x%2 == 0)
3          x = x/2;
4    else {
5          x = x + 1;
6          return x/2;
7    }
8    return x;
```


Indentation

```
x = 0xffabcd;
if (x%2 == 0) {
        x = x/2;
        ...;
}
else {
        x = x + 1;
        return x/2;
}
return x;
```


indentation pour "`else if`"

```
if (x%2 == 0)
      x = x/2;
else if (x%3 == 0) {
      x = x + 1;
      return x/2;
}
else if (x%7 == 0) {
      ...
}
```

Instructions de contrôle

Parmi les autres instructions de contrôle, on trouve les boucles (`while`, `do`, et `for`). Tout fragment de code contenant une boucle peut être réécrit avec une autre boucle. Certains programmeurs n'utilisent que le `for` d'autres que le `while`.

Le `swich` est une instruction de branchement dans un long bloc. Les instructions `break` et `continue` sont des instructions contrôlant l'ordre des instructions dans un bloc ou une boucle.

Boucles

Le principe est toujours le même:

L'instruction ou le bloc des instructions de la boucle est exécuté une première fois si une certaine condition est remplie, et on repart ensuite en tête de boucle : on teste à nouveau la condition, et si elle est vraie, on répète l'instruction(ou le bloc) de la boucle, etc. et on recommence jusqu'à ce que la condition ne soit plus vraie.

Si la condition est toujours vraie, la boucle ne s'arrête jamais (boucle infinie).

Boucle while

La boucle `while` ne peut être exécutée que si une certaine condition est vraie. Cette condition est une *expression logique* (ou expression booléenne). On l'appelle parfois "condition d'entrer".

Sa valeur doit être VRAI (=non nulle) pour que l'on puisse entrer dans la boucle.

Quand la dernière instruction de la boucle est exécutée, on repart au début du `while` pour tester à nouveau la condition et entrer à nouveau ou non dans la boucle.

Boucle `while`

```
while   (expression)
     instruction ;
/*   ici (expression == 0 */
```

ou, avec un bloc

```
while   (expression)   {
     instruction1 ;
     ...
}
```

Boucle for

Le `for` permet d'exécuter facilement "Pour i variant de 1 à N, répéter ..." d'où son nom (car *for* signifie *pour*) :

```
for (int i=1 ; i <= N; i++)
    instruction;
```

De même, "Pour i décroissant de N à 1, répéter l'instruction"

```
for (int i = N; i > 0 ; i--)
    instruction;
```

Mais la syntaxe du `for` permet d'autres types de traitements.

Boucle for

Sa définition syntaxique, en BNF:

```
for ([for-init]; [expr-log] ; [expr])
        instr
```

`for-init` peut être une initialisation ou déclaration/initialisation de variable locale comme `int i=1`, ou une instruction quelconque. Elle ne sera exécutée qu'une seule fois, au début.

`expr-log` est la condition d'entrer dans la boucle. Elle doit être non nulle (= VRAI) pour autoriser l'entrée dans la boucle.

Enfin, la troisième expression dans les parenthèses du `for` sera exécutée après chaque tour de boucle, avant qu'on teste à nouveau l'entrée pour un nouveau tour.

```
          0          1        3
for ([for-init]; [expr-log] ;[expr])
          2
        instr ;
```

Remarque: l'instruction `instr` de la boucle peut être remplacée par un bloc d'instructions entre accolades.

Exemples de for

```
for (int i=0; (i < 100) ; i++)
    printf( "compteur = %d", i);
```

```
for ( ; ((c < 'a') || (c > 'z')); ){
  printf("entrer une lettre " );
  scanf("%c", &c);
}
```

On peut aussi écrire des boucles infinies :

```
for (... ; ;... )
  instruction;
```

(L'instruction vide est évaluée comme vraie)

Instruction break

L'instruction break est une instruction qui permet de sortir d'une boucle en cours d'exécution.

```
for (i = 1; i < N; i = i + 1)
{
   printf("i = %d", i);
   if (i%5 == 0)
        break;
}
```


Instruction continue

L'instruction continue crée aussi une interruption: le tour de boucle est interrompu, mais au lieu de quitter l'instruction de boucle comme avec un break, on continue la boucle, mais en enchaînant sur le tour suivant (test de la condition d'entrer dans la boucle).

```
for (i = 0  ; i < 5;  i = i + 1)  {
   if ( i%2 == 0 )
        continue;
   printf( "i vaut %d\n", i);
}
```


Branchements

Les branchements sont des instructions qui modifient la suite des instructions. On a déjà introduit le branchement conditionnel `if`.

Mentionnons aussi au passage une expression réalisant réalisant un branchement dans son calcul: l'expression conditionnelle

```
expr-cond ::=   expr0 ?   expr1 :  expr2
```

Elle retourne la valeur de `expr1` si `expr0` est évaluée à VRAI (= non nulle), et celle de `expr2` sinon.

Exemple: (x < o ? - x : x). Cette expression retourne la valeur absolue de x.

Le switch

Le switch est un branchement dans un bloc à une certaine hauteur, appelée case :

```
switch (nb) {
     case 1:   printf("un");
                    break;
     case 2:   printf("deux");
                    break;
     case 3:   printf("trois");
                    break;
     default:  printf("erreur");
}
```

Le switch

L'expression figurant entre parenthèses après le mot `switch` est évaluée, et sa valeur est comparée successivement aux différentes constantes introduites par les différents `case`. L'entrée dans le bloc s'effectue au niveau du premier cas d'égalité trouvé.

S'il n'y a aucun cas d'égalité, on peut forcer l'entrée dans le cas par défaut introduit par `default` et placé à la fin.

Le switch

Le `switch` n'est pas un branchement sur des cas séparés – bien qu'on l'utilise très souvent de cette manière (on termine alors chaque liste d'instructions correspondant aux différents `case` par un `break`).

Le `switch` est un branchement dans un bloc sur une instruction particulière, introduite par `case`, mais cette instruction fait partie d'un bloc plus large, comportant les instructions de tous les cas successifs mis bout-à-bout.

Pour interrompre l'exécution du bloc, il faut rencontrer l'instruction `break` qui permet de sortir du `switch`.


```
switch ( nb ) {
   case 1:
   case 5:
   case 3:  instruction;
            instruction;
            break;
   case 4:
   case 6:  instruction;
   case 8:  instruction;
            break;
   default: instruction;
            /* ici break inutile */
}
```


Un `switch` pour traiter des cas de manière identique :

```
switch (nb) {
   case 1:
   case 5:
   case 3:   printf("impair");
             break;
   case 4:
   case 6:
   case 8:   printf("pair");
             break;
   default:  printf("erreur");
}
```

```
switch (c) {
        case 'a':
        case 'b':
          ...
        case 'z': printf("lettre");
                  break;
        case '0':
        case '1':
          ...
        case '9': printf("chiffre");
                  break;
        default:  printf("erreur");
}
```


Une source fréquente d'erreur, souvent difficile à déceler, est d'oublier d'écrire l'instruction `break` pour quitter le `switch` après le traitement d'un cas.

Retenez qu'un point essentiel avec swich est de ne pas oublier les instructions `break` *quand elles sont nécessaires pour séparer les différents cas.*

Compilation

année universitaire: 2021 -2022

Compilation

La compilation est la traduction d'un texte de programme écrit dans un langage de programmation évolué, le code source, en un code binaire exécutable par la machine: le code machine.

Le langage C est un langage *compilé*. La traduction du code source en code machine s'effectue d'un seul tenant - contrairement aux langages *interprétés* (comme le noyau des commandes: le *shell*) où elle a lieu pas à pas c'est-à-dire, ligne à ligne.

Compilation

La compilation s'effectue en analysant et transformant plusieurs fois le fichier du texte source. Les étapes de la compilation sont :

1. Préprocesseur (transformation du texte)
2. Analyse lexicale
3. Analyse syntaxique
4. Analyse sémantique
5. Génération du code
6. Édition de liens

Préprocesseur

Le préprocesseur du compilateur exécute les instructions commençant par un dièse, e.g. les #define ou les #include.

Il supprime aussi les commentaires.

Le texte initial est ainsi transformé en un autre, par substitution textuelle.

ex: `#define MAX 100`

-> remplace partout MAX par `100`

`#include "fichier.h"`

-> substitue à cette ligne tout le texte du fichier `fichier.h`

Analyse lexicale

Elle identifie les unités lexicales (les *lexèmes* ou mots du texte). Les caractères blancs servent à les séparer, comme dans `int x` , mais sinon ils sont ignorés (comme dans `3 * x + 1`).

- Erreur lexicale : `int x = @;`

```
>> error: stray '@' in program
```

- Erreur détectée uniquement au moment de l'analyse sémantique : `intx = 0;`

```
>> error: 'intx' undeclared
(first use in this function)
```


Analyse syntaxique

Elle vérifie que la grammaire est conforme à celle du langage et construit un arbre syntaxique qui sera utilisé ensuite pour générer du code.

- exemple, le mot `if` aurait du ici précéder le mot `else`:

```
x = 10;
else x = 12;
```

```
>> error: expected expression
before `else'
```

Analyse sémantique

Elle vérifie que le programme a bien un sens en listant les objets manipulés par le programme et en vérifiant que les actions du programme sur ces objets sont bien autorisées.

ex: variable `x` utilisée mais non déclarée

```
>> error: 'x' undeclared (first use in this function)
```

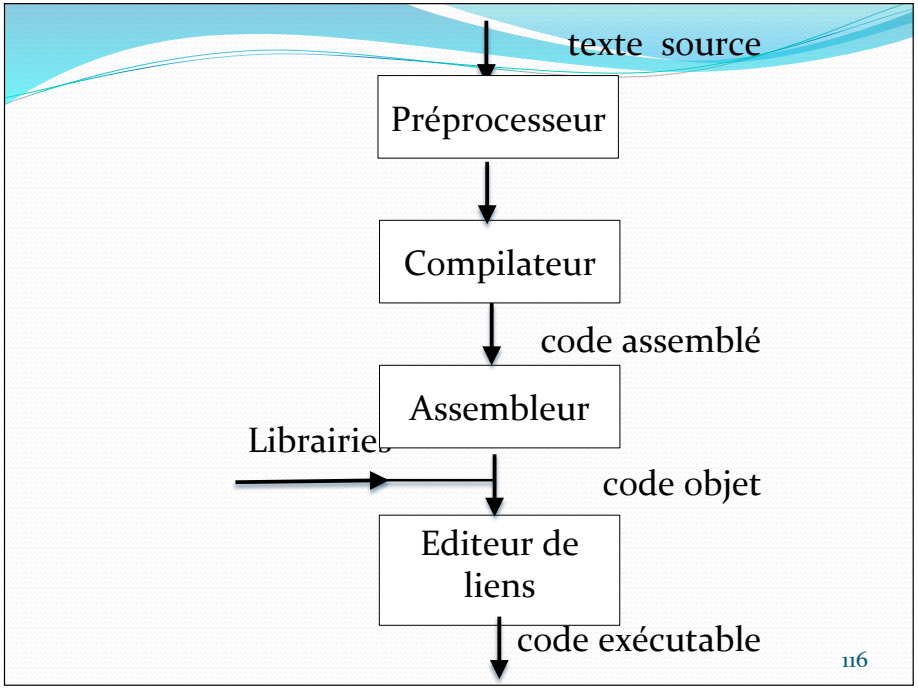
Remarque: l'inverse (variable déclarée mais non utilisée), génère un *Warning* si on compile avec l'option `-Wall` de `gcc`.

Analyse sémantique

L'essentiel de l'analyse sémantique en C est la vérification de la cohérence des types de variables et de données, dans les affectations et dans les appels ou définitions de fonctions.

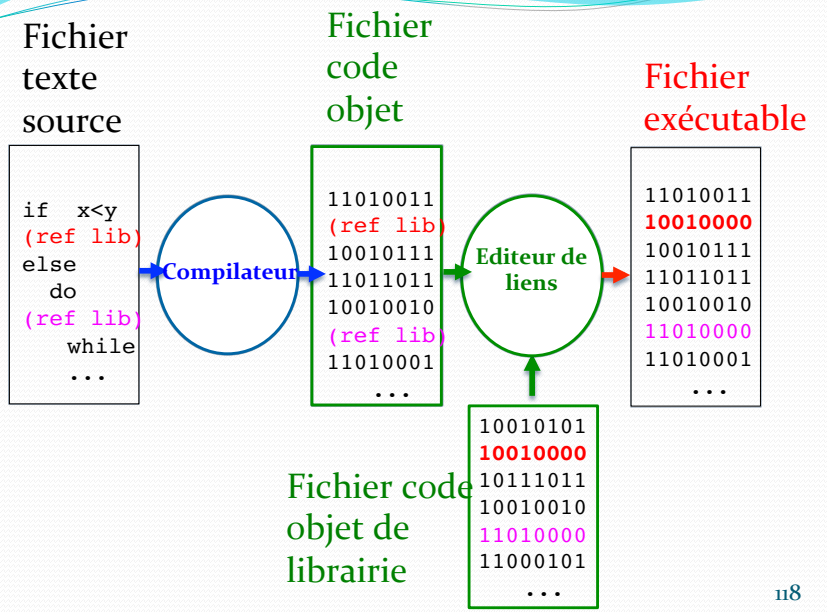
Génération de code

- La génération de code s'effectue en passant par un code en assembleur, avec optimisation et allocation des registres, suivi d'une traduction en code objet.
- Le code objet est un code binaire qui n'est pas exécutable par la machine car il lui manque certaines parties - comme le code objet de fonctions de bibliothèques, ou celui de la fonction `main`.



Editions de liens

- La génération d'un code exécutable s'effectue en créant des liens entre les différents fichiers de code objet utilisés, comme ceux des fonctions de librairies ou ceux des fonctions écrites par le programmeur et compilés dans des fichiers séparés.
- Un code exécutable n'est créé que s'il existe le code de la fonction `main`.



On compilera un programme C sous Linux avec la commande gcc (de GNU):

$ gcc main.c
→crée un fichier exécutable `a.out` si main.c contient une fonction main()

$ gcc main.c –o progexe
→crée un fichier exécutable `progexe`

$ gcc –c monfichier.c
→crée un code objet intitulé `monfichier.o`

$ gcc fichier.o main.c
→crée un fichier exécutable `a.out`

$ gcc fichier.o main.c -o progexe
→crée un fichier exécutable `progexe`

Les options –l et –L indiqueront à l'éditeur de liens où trouver les codes des fonctions de bibliothèques si ils ne sont pas trouvés dans les répertoires standards.

L'option –Wall permet l'affichage de tous les messages d'avertissement (*Warning*) pouvant alerter le programmeur sur d'éventuelles erreurs possibles.

L'option –g permet de créer un code qui sera ensuite interprétable par un *debugger*.

$ gcc –g –c fichier.c

$ gcc –g fichier.o main.c –o exec

$ gdb exec

permet de lancer le débugger GNU gdb avec l'executable exec - qui devra avoir été compilé avec l'option –g de gcc.

On peut aussi lancer gdb, puis charger un executable avec la commande file de gdb.

gdb une fois lancé, on peut lancer les commandes suivantes (entre autres!):

help file run list where quit

Dans la pratique…

La compilation avec la commande

`gcc –Wall monfichier.c`

affiche les warning en sus des erreurs. Un code exécutable est quand même produit s'il n'y a que des warning et pas d'erreur.

ex: int x = 4.5; provoque l'affichage

```
warning: implicit conversion from
'double' to 'int' changes value from
4.5 to 4
```


On retiendra que pour supprimer les *warning* sur des types, on peut corriger le programme en rajoutant les opérations de cast souhaitées et indiquées par le compilateur. Mais souvent cela révèlera une erreur de conception.

L'intérêt des *warning* est d'attirer l'attention du programmeur sur les incohérences dans l'usage de ses types de données.

Commande exécutable

- Le code exécutable s'appelle `a.out` ou porte un nom spécifique comme progexe.
- la valeur de retour de la commande exécutable est celle de retour du `main`. Elle pourra être interprétée par le shell.

Par convention, la fonction `main` doit retourner

- EXIT_SUCCESS (qui vaut en général zéro), si tout s'est bien passé

- un nombre différent en cas d'erreur.

Commande exécutable

Si le `main` est prototypé par

```
int main(int argc,
         char*argv[])
```

`argv` est un tableau de chaines de caractères : celles tapées sur la ligne de lancement du programme.

`argv[0]` est le nom de la commande

`argv[1]` est le mot suivant, etc.

`argc` (*argument count*) est le nombre d'éléments de `argv` et est toujours `>=1`.