

LSINC1252 & INFO1252

Systèmes Informatiques



Leçon 2 : Langage C – rappels et compléments

Pr. Etienne Rivière

etienne.riviere@uclouvain.be

Objectifs de ce cours

- Revoir les grands principes du langage C
 - Représentations des données et types
 - Structure d'un programme
 - Intégration avec le shell
 - Tableaux et structures
 - Fonctions
 - Pointeurs
- Détailler des fonctions plus avancées du langage
 - Pointeurs sur des pointeurs et sur des fonctions
 - Manipulation de bits

À la suite de I503

- Utilisation du C déjà couverte en partie dans le cours de projet LSINCI503-LEPLI503
- Comment valider / rattraper les notions non acquises ?
 - Ce cours : rappels mais survol forcément rapide
 - Le syllabus présente une description complète et détaillée du langage C avec de nombreux exemples
 - Les exercices du cours LEPLI503 sont disponibles sur Inginius (travail en autonomie, mais poser des questions aux assistants et tuteurs est autorisé et même encouragé)

Parties du syllabus couvertes

- Le langage C
- Types de données
 - Nombres entiers
 - Nombres réels
 - Les tableaux
 - Caractères et chaînes de caractères
 - Les pointeurs
 - Les structures
 - Les fonctions
 - Les expressions de manipulation de bits
- Déclarations
- Unions et énumérations
- Compléments de C
 - Pointeurs
 - De grands programmes en C
 - Traitement des erreurs

couvertes aujourd'hui

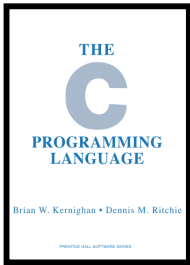
non couvertes aujourd'hui

Introduction



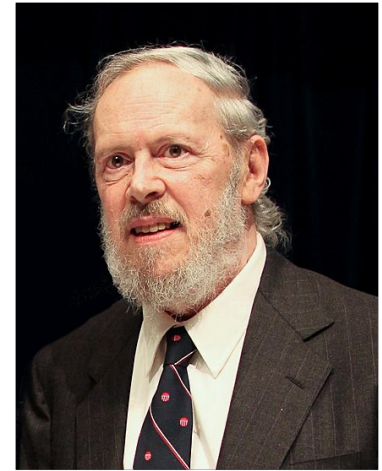
Ken Thompson et Dennis Ritchie,
inventeurs de UNIX

- 1970 : invention d'UNIX
 - Systèmes d'exploitation jusqu'alors écrits en langage d'assemblage
 - Utilisation directe du jeu d'instruction du processeur
 - Nécessaire pour certaines fonctions très proches du matériel : par exemple configuration des interruptions
 - 😞 Mais : développement d'algorithmes et de structures de données complexes très chronophage et difficile à maintenir
 - Le développement d'UNIX a montré la nécessité d'un langage haut niveau (structuré) mais restant proche du matériel
- De nos jours, le langage d'assemblage reste seulement utilisé dans des parties très spécifiques des SE (e.g., interactions avec le gestionnaire de périphérique, utilisation d'instruction spéciales dans des bibliothèques multimédia ou de cryptographie)



Introduction

- Language C et UNIX : une histoire commune
 - C inventé par Dennis Ritchie, co-inventeur d'UNIX
 - Pour pouvoir (ré) implémenter UNIX sur PDP11 en utilisant le moins de code d'assemblage possible
- Objectifs
 - Haut niveau (structuré)
 - Proche du matériel
 - Accès direct & gestion directe de la mémoire
 - Correspondance naturelle entre constructions du langage et traduction en instructions processeur
 - Efficacité
 - Pas de surcoût à l'exécution (contrairement à machine virtuelle avec Java ou interprétation de code avec Python)
 - Portabilité (dans une certaine mesure)



Dennis Ritchie, inventeur de C



Un mini-ordinateur PDP11

Nul n'échappe au hello world

The diagram shows a C program with several parts highlighted and annotated:

- commentaires**: Points to the multi-line comment at the top of the program.
- directives du préprocesseur**: Points to the `#include <stdio.h>` and `#include <stdlib.h>` lines.
- point d'entrée et arguments ligne cmd.**: Points to the `int main(int argc, char *argv[])` function signature.
- appel de fonction d'une librairie de `stdio.h`**: Points to the `printf("Hello, world!\n");` statement.
- termine avec un code de retour**: Points to the `return EXIT_SUCCESS;` statement.

```
/* *****  
 * Hello.c  
 *  
 * Programme affichant sur la sortie  
 * standard le message "Hello, world!"  
 *  
 * *****  
#include <stdio.h>  
#include <stdlib.h>  
  
int main(int argc, char *argv[])  
{  
    // affiche sur la sortie standard  
    printf("Hello, world!\n");  
  
    return EXIT_SUCCESS;  
}
```

```
gcc -o helloworld helloworld.c  
./helloworld  
Hello, world!
```

Préprocesseur

- Le code C est tout d'abord transformé par un préprocesseur avant d'être compilé en langage machine par le compilateur
 - Il s'agit d'une traduction texte-vers-texte
 - Transformation par remplacement de directives

```
#define ZERO 0  
#define STRING "LEPL1503"
```

```
#define EXIT_FAILURE 1  
#define EXIT_SUCCESS 0
```

dans `stdio.h`

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

```
#define DEBUG  
/* ... */  
#ifdef DEBUG  
printf("debug : ...");  
#endif /* DEBUG */
```

- `gcc -E` pour voir le résultat du préprocesseur

Sortie du préprocesseur (extrait)

```
gcc -E helloworld.c | tail -n 15
```

```

/*****
 * Hello.c
 *
 * Programme affichant sur la sortie
 * standard le message "Hello, world!"
 *
 *****/

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

int main(int argc, char *argv[])
{
    // affiche sur la sortie standard
    printf("Hello, world!\n");

    return EXIT_SUCCESS;
}

```

```
cat helloworld.c | wc -l
18
```

```

__attribute__((__nothrow__ ,
__leaf__)) __attribute__((__nonnull__
(1, 2, 3))) ;
# 948 "/usr/include/stdlib.h" 3 4
extern int getloadavg (double
__loadavg[], int __nelem)
__attribute__((__nothrow__ ,
__leaf__)) __attribute__((__nonnull__
(1)));
# 964 "/usr/include/stdlib.h" 3 4

# 11 "helloworld.c" 2

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

    printf("Hello, world!\n");

    return 0;

}

```

```
gcc -E helloworld.c | wc -l
1850
```

headers et `#include`

- Les directives `#include` du préprocesseur permettent d'importer des définitions de fonctions, de constantes, etc. d'une librairie
- Séparation entre fichiers headers (`.h`) et sources (`.c`)
 - Le header contient la définition des fonctions, suffisante pour savoir comment générer le code pour les appeler
 - Le code source contient la mise en œuvre de ces fonctions
- `#include "monheader.h"` pour inclure un header du dossier courant (du même projet)
- `#include <header.h>` pour inclure un header standard
 - Le préprocesseur cherche dans une liste de dossiers connus
 - `/etc/include` est le plus courant sous UNIX/Linux
- Information sur une librairie standard avec `man :man 3 stdio.h`

Types de variables (I)

- `int` et `long` : utilisés lors de la déclaration d'une variable de type entier
- `char` : utilisé lors de la déclaration d'une variable permettant de stocker un caractère
- `double` et `float` : utilisés lors de la déclaration d'une variable permettant de stocker un nombre représenté en virgule flottante.
- booléens :

```
#define false    0  
#define true     1
```

avant standard C99

```
#define false    (bool)0  
#define true     (bool)1
```

standard C99 :
définis dans `stdbool.h`

Types de variables (2)

- En Java, les chaînes de caractères sont représentées grâce à l'objet `String`. En C, une chaîne de caractères est représentée sous la forme d'un tableau de caractères dont le dernier élément contient la valeur `\0`.
- Alors que Java stocke les chaînes de caractères dans un objet avec une indication de leur longueur, en C il n'y a pas de longueur explicite pour les chaînes de caractères mais le caractère `\0` sert de marqueur de fin de chaîne de caractères.
- Lorsque le langage C a été développé, ce choix semblait pertinent, notamment pour des raisons de performance (pourquoi ?)
 - Pensez vous que ce soit toujours le cas ?

code de format

```
char string[10];  
string[0] = 'j';  
string[1] = 'a';  
string[2] = 'v';  
string[3] = 'a';  
string[4] = '\0';  
printf("String : %s\n", string);
```

`\n` = retour à la ligne
`\t` = tabulation

Structures de contrôle

- test →
`if (condition) { ... } else { ... }`
- boucle →
`while (condition) { ... }`
- boucle →
`do { ... } while (condition);`
- itération →
`for (init; condition; incr) { ... }`
- Les conditions ne rendent pas forcément un booléen ; tout ce qui n'est pas une suite de bits à zéro (e.g., 0x00000000 pour un mot de 32 bits) est vrai (0x00000001 tout comme 0xFD5A93EB)

Intégration avec le shell

- Point d'entrée : `main`

```

/*****
 * cmdline.c
 *
 * Programme affichant ses arguments
 * sur la sortie standard
 *
 *****/

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

int main(int argc, char *argv[])
{
    int i;
    printf("Ce programme a %d argument(s)\n", argc);
    for (i = 0; i < argc; i++)
        printf("argument[%d] : %s\n", i, argv[i]);
    return EXIT_SUCCESS;
}

```

valeur de retour (#? pour la lire en bash)
EXIT_FAILURE en cas d'échec

nombre d'arguments

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

arguments

`argv[0]` : nom du programme
 exécuté par l'utilisateur :

```

Ce programme a 5 argument(s)
argument[0] : ./cmdline
argument[1] : 1
argument[2] : -list
argument[3] : abcdef
argument[4] : linfo1252

```

Affichage formaté avec printf

- En Java
 - `System.out.println("Bonjour " + student.getName() + " !");`
- En C avec `printf` : utilisation de codes de formats, remplacés par une version formatée de la valeur de la variable (ou constante) utilisée :

The diagram illustrates the mapping between the format specifiers in the `printf` function and the values they format. Arrows point from each specifier in the input string to its corresponding formatted value in the output string.

Input: `printf("Color %s, Number %d, Float %4.2f", "red", 123456, 3.14);`

Output: Color red, Number 123456, Float 3.14

The mapping is as follows:

- `%s` maps to `"red"`
- `%d` maps to `123456`
- `%4.2f` maps to `3.14`

credit : I, Surachit, CC BY-SA 3.0, Wikimedia

Affichage formaté avec printf

```
char weekday[] = "Monday";
char month[] = "April";
int day = 1;
int hour = 12;
int min = 42;
char str[] = "INFO1252";
int i;

// affichage de la date et l'heure
printf("%s, %s %d, %d:%d\n", weekday, month, day, hour, min);

// affichage de la valeur de PI
printf("PI = %f\n", 4 * atan(1.0));

// affichage d'un caractère par ligne
for(i = 0; str[i] != '\0'; i++)
    printf("%c\n", str[i]);
```

```
Monday, April 1, 12:42
PI = 3.141593
I
N
F
O
1
2
5
2
```


Fonctions

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

// retourne vrai si c est un chiffre, faux sinon
// exemple simplifié, voir isdigit dans la librarire standard
// pour une solution complète
int digit(char c)
{
    return ((c >= '0') && (c <= '9'));
}

// affiche un message d'erreur
void usage()
{
    fprintf(stderr, "Ce programme ne prend pas d'argument\n");
    exit(EXIT_FAILURE);
}

int main(int argc, char *argv[])
{
    char c;

    if (argc > 1)
        usage();

    while ((c = getchar()) != EOF) {
        if (digit(c))
            putchar(c);
    }
    return EXIT_SUCCESS;
}
```

digit retourne un int
caractères codés sur un octet
(char);

ici on fait l'hypothèse que les
chiffres sont encodés avec des
valeurs consécutives

usage ne retourne rien

fprintf permet de spécifier
le flux à utiliser pour
l'impression du message, ici le
flux d'erreur (STDERR) plutôt
que le flux standard
(STDOUT)

Types de données

- Valeurs stockées sous forme de séquences de bits
- Nombre entiers
 - signés (`int` notamment en C)
 - non-signés (`unsigned int` notamment en C)
- Exemple pour un nibble (4 bits) :
- Représentation hexadécimale pratique pour représenter un nibble avec un seul symbole !

binaire	octal	hexadécimal	décimal
0	0	0	0
1	1	1	1
10	2	2	2
11	3	3	3
100	4	4	4
101	5	5	5
110	6	6	6
111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	A	10
1011	13	B	11
1100	14	C	12
1101	15	D	13
1110	16	E	14
1111	17	F	15

Types de données

- L'entier décimal 123 s'écrit 0x7b en notation hexadécimale et 0b000000000000000000000000000000001111011 en notation binaire
- L'entier décimal 7654321 s'écrit 0x74cbb1 en notation hexadécimale et 0b000000000011101001100101110110001 en notation binaire

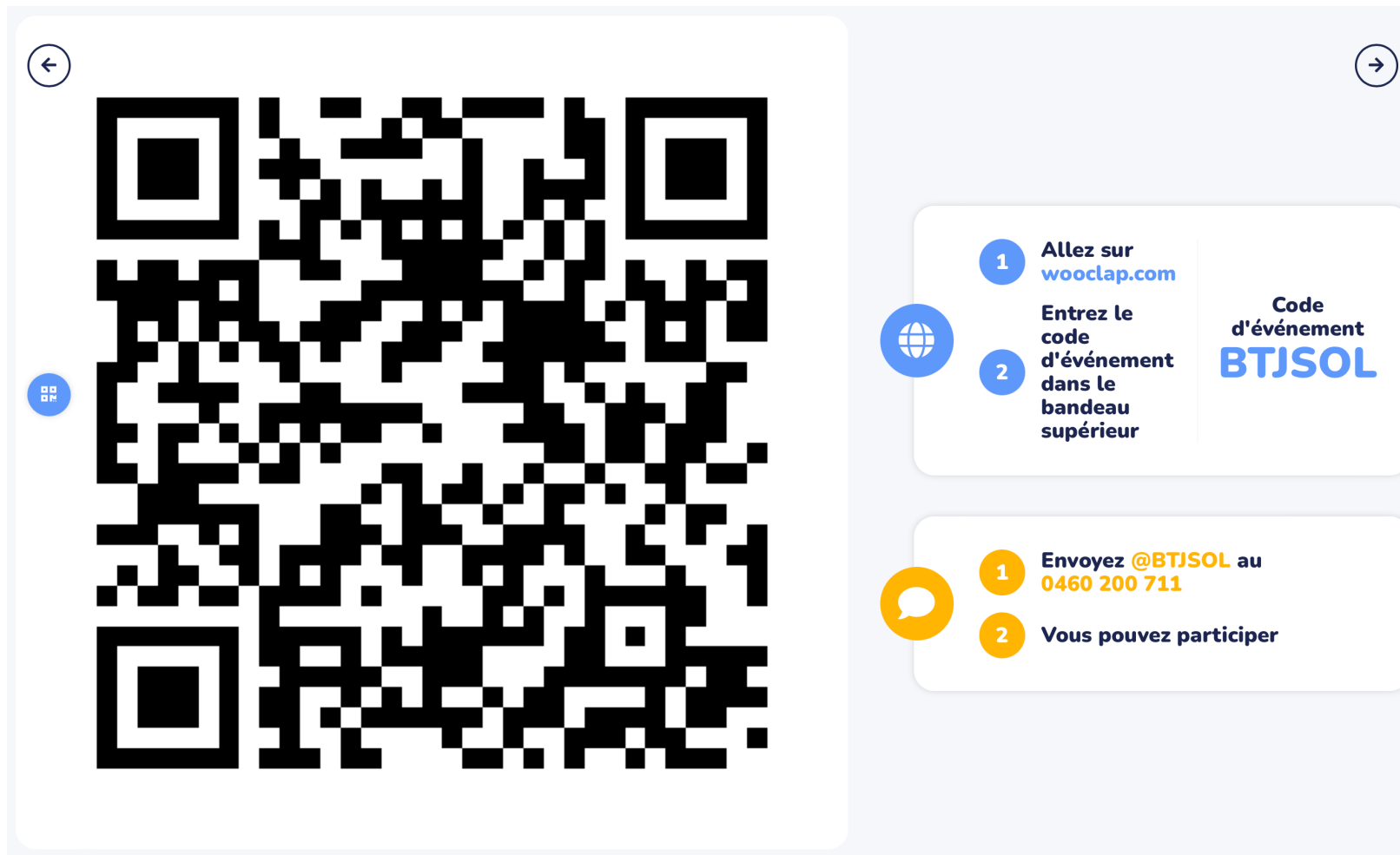
```
int i;
i = 123;    // décimal
i = 0x7b;   // hexadécimal
i = 0173;   // octal !!
// i = 0b1111011; // binaire,
// seulement certains compilateurs
```



notation octale :

```
int i, j;
i = 65;    // décimal
j = 065;   // octal !!!
if (i == j)
    printf("%d et %d sont égaux\n", i, j);
else
    printf("%d et %d sont différents\n", i, j);
```

Calculons en hexadécimal



←

QR

→

1 Allez sur wooclap.com

2 Entrez le code d'événement dans le bandeau supérieur

Code d'événement **BTJSOL**

1 Envoyez @BTJSOL au 0460 200 711

2 Vous pouvez participer

<https://app.wooclap.com/BTJSOL?from=instruction-slide>

Entiers non signés

- Types :

Type	Explication
<code>unsigned short</code>	Nombre entier non signé représenté sur au moins 16 bits
<code>unsigned int</code>	Nombre entier non signé représenté sur au moins 16 bits
<code>unsigned long</code>	Nombre entier non signé représenté sur au moins 32 bits
<code>unsigned long long</code>	Nombre entier non signé représenté sur au moins 64 bits

- on obtient le nombre d'octets utilisés par un type donné avec `sizeof (type)` ; e.g. `sizeof (int)` est souvent = à 4 (32 bits)
- Définitions dans `stdint.h`

Entiers signés

- Représentation en *complément à 2*
- Exemple : +3 sur 4 bits est 0011
 - Inversion des bits : 0011 \Rightarrow 1100
 - Ajout de 1 : 1100 \Rightarrow 1101
 - -3 est donc 1101
- Il y a toujours une valeur négative sans son inverse (e.g. -128 mais pas +128)

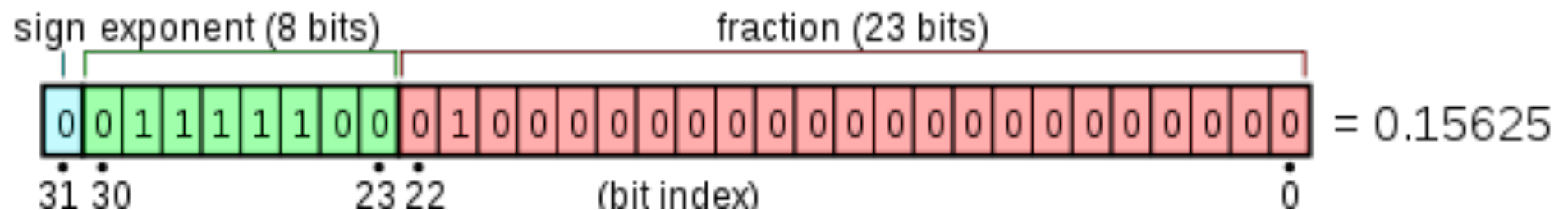
Type	Explication
short	Nombre entier signé représenté sur au moins 16 bits
int	Nombre entier signé représenté sur au moins 16 bits
long	Nombre entier signé représenté sur au moins 32 bits
long long	Nombre entier signé représenté sur au moins 64 bits

Eight-bit signed integers

Decimal value ↕	Two's-complement Representation ↕
0	0000 0000
1	0000 0001
2	0000 0010
126	0111 1110
127	0111 1111
-128	1000 0000
-127	1000 0001
-126	1000 0010
-2	1111 1110
-1	1111 1111

Nombres flottants

- Réel impossibles à représenter de façon complète ($1/3$, π , e , ...)
- Approximation avec une représentation flottante, selon un format standardisé (IEEE-754) :



- simple précision (`float`) : séquence de 32 bits
- double précision (`double`): séquence de 64 bits.

```
#define FLT_MIN 1.17549435e-38F
#define FLT_MAX 3.40282347e+38F

#define DBL_MIN 2.2250738585072014e-308
#define DBL_MAX 1.7976931348623157e+308
```

défini dans `float.h`

credit :Wikipedia

Tableaux

```
#define N 10
int vecteur[N];
float matriceC[N][N];
float matriceR[N][2*N];
```

```
int i;
int sum = 0;
for (i = 0; i < N; i++)
{
    sum += v[i];
}
```

```
#define L 2
#define C 3
float matriceR[L][C] = { {1.0,2.0,3.0},
                          {4.0,5.0,6.0} };

int i, j;
float min = FLT_MAX;
for (i = 0; i < L; i++)
    for (j = 0; j < C; j++)
        if (matriceR[i][j] < min)
            min=matriceR[i][j];
```


Caractères

- Représentation ASCII : standard (accord international) sur la correspondance symbole (e.g. "A", "@" ou "6") \Leftrightarrow la valeur le codant
 - Attention, le code du caractère 1 ne vaut pas la valeur 1 !
 - Les caractères en majuscule ("M") ont un code différent de ceux en minuscule ("m")
- En C, char = un octet = un caractère
 - 0x0??????? permet de représenter les caractères courants en anglais (sans accents, norme originelle)
 - 0x1??????? permet de représenter les caractères accentués ç é è ù (norme ISO-8859) pour les langues européennes comme le français

Code ASCII (7 bits)

HEX	DEC	CHR	CTRL
00	0	NUL	^@
01	1	SOH	^A
02	2	STX	^B
03	3	ETX	^C
04	4	EOT	^D
05	5	ENQ	^E
06	6	ACK	^F
07	7	BEL	^G
08	8	BS	^H
09	9	HT	^I
0A	10	LF	^J
0B	11	VT	^K
0C	12	FF	^L
0D	13	CR	^M
0E	14	SO	^N
0F	15	SI	^O
10	16	DLE	^P
11	17	DC1	^Q
12	18	DC2	^R
13	19	DC3	^S
14	20	DC4	^T
15	21	NAK	^U
16	22	SYN	^V
17	23	ETB	^W
18	24	CAN	^X
19	25	EM	^Y
1A	26	SUB	^Z
1B	27	ESC	
1C	28	FS	
1D	29	GS	
1E	30	RS	
1F	31	US	

HEX	DEC	CHR
20	32	SP
21	33	!
22	34	"
23	35	#
24	36	\$
25	37	%
26	38	&
27	39	'
28	40	(
29	41)
2A	42	*
2B	43	+
2C	44	,
2D	45	-
2E	46	.
2F	47	/
30	48	0
31	49	1
32	50	2
33	51	3
34	52	4
35	53	5
36	54	6
37	55	7
38	56	8
39	57	9
3A	58	:
3B	59	;
3C	60	<
3D	61	=
3E	62	>
3F	63	?

HEX	DEC	CHR
40	64	@
41	65	A
42	66	B
43	67	C
44	68	D
45	69	E
46	70	F
47	71	G
48	72	H
49	73	I
4A	74	J
4B	75	K
4C	76	L
4D	77	M
4E	78	N
4F	79	O
50	80	P
51	81	Q
52	82	R
53	83	S
54	84	T
55	85	U
56	86	V
57	87	W
58	88	X
59	89	Y
5A	90	Z
5B	91	[
5C	92	\
5D	93]
5E	94	^
5F	95	_

HEX	DEC	CHR
60	96	`
61	97	a
62	98	b
63	99	c
64	100	d
65	101	e
66	102	f
67	103	g
68	104	h
69	105	i
6A	106	j
6B	107	k
6C	108	l
6D	109	m
6E	110	n
6F	111	o
70	112	p
71	113	q
72	114	r
73	115	s
74	116	t
75	117	u
76	118	v
77	119	w
78	120	x
79	121	y
7A	122	z
7B	123	{
7C	124	
7D	125	}
7E	126	~
7F	127	DEL

Limitations et remplacements du code ASCII

- Le code ASCII sur 7 ou 8 bits permet de représenter la majorité des caractères utilisés pour les langues européennes/latines
- Quid des langues avec de nombreux symboles?
- Codage Unicode
 - Le nombre d'octets utilisés pour un caractère varie
 - Un bit de poids fort (le plus à gauche) à 1 indique que l'octet suivant fait partie du code du caractère
 - Compatibilité avec ASCII 7 bits : les codes à un seul octet commençant par un 0 encodent les mêmes caractères qu'ASCII 7 bits
 - Représentation des alphabets khmer, arabe, thaï, etc.

Exemple de caractères Unicode (UTF-8)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000																
0010																
0020		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
0030	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0040	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0050	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
0060	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0070	p	q	r	s	t	u	v	w	x	y	z	{		}	~	

10E0	ᲀ	ᲁ	ᲂ	ᲃ	ᲄ	ᲅ	ᲆ	ᲇ	ᲈ	Ᲊ	ᲊ	᲋	᲌	᲍	᲎	᲏
10F0	Ა	Ბ	Გ	Დ	Ე	Ვ	Ზ	Თ	Ი	Კ	Ლ	Მ	Ნ	Ო	Პ	Ჟ
1100	Რ	Ს	Ტ	Უ	Ფ	Ქ	Ღ	Ყ	Შ	Ჩ	Ც	Ძ	Წ	Ჭ	Ხ	Ჯ
1110	Ჰ	Ჱ	Ჲ	Ჳ	Ჴ	Ჵ	Ჶ	Ჷ	Ჸ	Ჹ	Ჺ	᲻	᲼	Ჽ	Ჾ	Ჿ
1120	Რ	Ს	Ტ	Უ	Ფ	Ქ	Ღ	Ყ	Შ	Ჩ	Ც	Ძ	Წ	Ჭ	Ხ	Ჯ
1130	Ჰ	Ჱ	Ჲ	Ჳ	Ჴ	Ჵ	Ჶ	Ჷ	Ჸ	Ჹ	Ჺ	᲻	᲼	Ჽ	Ჾ	Ჿ

Chaînes de caractères

- Une chaîne de caractères (`char`) est stockée sous la forme d'un tableau :

```
char name1[] = { 'U', 'n', 'i', 'x', '\0' };  
char name2[] = { "Unix" };  
char name3[] = "Unix";
```

- Le dernier élément du tableau contient le caractère `'\0'` (valeur ASCII == 0)
- Exemple, calcul de la longueur :

```
int length(char str[])  
{  
    int i = 0;  
    while (str[i] != 0) {// '\0' et 0 sont égaux  
        i++;  
    }  
    return i;  
}
```

Pointeurs

- C est un langage proche de la machine (dit de *bas niveau*) permettant d'interagir directement avec la mémoire
 - Et en particulier de savoir où les éléments (variables, fonctions, etc.) sont stockés
 - Différence avec les langages de plus haut niveau comme Java ou Python !
- La mémoire peut être vue comme une suite de 'cases' (octets) avec des adresses consécutives (index de chaque octet)
 - Les adresses sont des entiers non signés
 - sur 32 ou 64 bits selon les processeurs / jeu d'instruction
- Un **pointeur** est une variable qui contient l'*adresse* à laquelle est stockée un autre élément (variable, fonction, etc.)

Adresses

- Exemple avec adresses de 3 bits
 - 2^3 octets = 8 octets
- `&var` permet d'obtenir l'adresse à laquelle une variable est stockée
 - `&c` donne 0b101
 - `&(name[0])` donne 0b000

```
char name[ ] = "Unix";
char c = 'z';
```

Adresse	Conten
111	u
110	0
101	Z
100	0
011	x
010	i
001	n
000	U

Exemple

```
int i = 1252;
char str[] = "info1252";
char c = 'c';

printf("i vaut %d, occupe %ld bytes et est stocké à l'adresse : %p\n",
      i, sizeof(i), &i);
printf("c vaut %c, occupe %ld bytes et est stocké à l'adresse : %p\n",
      c, sizeof(c), &c);
printf("str contient \"%s\" et est stocké à partir de l'adresse : %p\n",
      str, &str);
```

```
i vaut 1252, occupe 4 bytes et est stocké à l'adresse : 0x7fff89f99cbc
c vaut c, occupe 1 bytes et est stocké à l'adresse : 0x7fff89f99caf
str contient "info1252" et est stocké à partir de l'adresse : 0x7fff89f99cb0
```


Utilisation des pointeurs

- Un pointeur est défini avec :
`type *nom_du_pointeur;`
- Exemples :

```
int i = 1;           // entier
int *ptr_i;          // pointeur vers un entier
char c = 'z';        // caractère
char *ptr_c;         // pointeur vers un char
```
- On obtient ou assigne la valeur de la variable pointée par le pointeur en le dérérérençant avec l'opérateur *

Exemple d'utilisation des pointeurs

```

int i = 1;           // entier
int *ptr_i;          // pointeur vers un entier
char str[] = "Unix";
char *s;             // pointeur vers un char

ptr_i = &i;
printf("valeur de i : %d, valeur pointée par ptr_i : %d\n", i, *ptr_i);

*ptr_i = *ptr_i + 1252;
printf("valeur de i : %d, valeur pointée par ptr_i : %d\n", i, *ptr_i);

s = str;

for (i = 0; i < strlen(str); i++){
    printf("valeur de str[%d] : %c, valeur pointée par *(s+%d) : %c\n",
           i, str[i], i, *(s+i));
}

```

```

valeur de i : 1, valeur pointée par ptr_i : 1
valeur de i : 1253, valeur pointée par ptr_i : 1253
valeur de str[0] : U, valeur pointée par *(s+0) : U
valeur de str[1] : n, valeur pointée par *(s+1) : n
valeur de str[2] : i, valeur pointée par *(s+2) : i
valeur de str[3] : x, valeur pointée par *(s+3) : x

```

Équivalence pointeur/tableau

- En pratique en C, les notations `char*` et `char []` sont équivalentes et l'une peut s'utiliser à la place de l'autre.
- En utilisant les pointeurs, la fonction de calcul de la longueur d'une chaîne de caractères peut se réécrire comme suit :

```
int length(char *s)
{
    int i = 0;
    while (*(s+i) != '\0')
        i++;
    return i;
}
```

Arithmétique des pointeurs (I)

- On peut effectuer des opérations arithmétiques sur les pointeurs (+, -, ...) et donc sur les adresses qu'ils contiennent
- Exemple avec un tableau de `int` (32 bits, 4 octets)

```
#define SIZE 3
unsigned int tab[3];
tab[0] = 0x01020304;
tab[1] = 0x05060708;
tab[2] = 0x090A0B0C;
```

```
int i;
for (i = 0; i < SIZE; i++) {
    printf("%X est à l'adresse %p\n",
        tab[i], &(tab[i]));
}
```

est équivalent à

```
unsigned int* ptr = tab;
for (i = 0; i < SIZE; i++) {
    printf("%X est à l'adresse %p\n",
        *ptr, ptr);
    ptr++;
}
```

```
1020304 est à l'adresse 0x7fff5fbff750
5060708 est à l'adresse 0x7fff5fbff754
90A0B0C est à l'adresse 0x7fff5fbff758
```

Arithmétique des pointeurs (2)

Assignation `ptr=tab`.

-> Lorsque `tab` est déclaré par la ligne `unsigned int tab[3]`, le compilateur considère que `tab` est une constante qui contiendra toujours l'adresse du premier élément du tableau.

-> Il faut noter que puisque `tab` est considéré comme une constante, il est interdit d'en modifier la valeur en utilisant une assignation comme `tab=tab+1`.

Le pointeur `ptr`, par contre, correspond à une zone mémoire qui contient une adresse. Il est tout à fait possible d'en modifier la valeur. Ainsi, l'assignation `ptr=tab` (ou `ptr=&(tab[0])`) place dans `ptr` l'adresse du premier élément du tableau. Les pointeurs peuvent aussi être modifiés en utilisant des expressions arithmétiques :

```
ptr = ptr + 1; // ligne 1
ptr++;        // ligne 2
ptr = ptr - 2; // ligne 3
```

Arithmétique des pointeurs (3)

```
ptr = ptr + 1; // ligne 1
ptr++;       // ligne 2
ptr = ptr - 2; // ligne 3
```

Après l'exécution de la première ligne :

-> `ptr` va contenir l'adresse de l'élément `1` du tableau `tab` (c'est-à-dire `&(tab[1])`).

Ce résultat peut surprendre car si l'élément `tab[0]` se trouve à l'adresse `0x7fff5fbff750` c'est cette adresse qui est stockée dans la zone mémoire correspondant au pointeur `ptr`. On pourrait donc s'attendre à ce que l'expression `ptr+1` retourne plutôt la valeur `0x7fff5fbff751`. Il n'en est rien.

-> En C, lorsque l'on utilise des calculs qui font intervenir des pointeurs, **le compilateur prend en compte le type du pointeur qui est utilisé.**

-> Comme `ptr` est de type `unsigned int*`, il pointe toujours vers une zone mémoire permettant de stocker un entier non signé sur 32 bits. L'expression `ptr+1` revient en fait à calculer la valeur `ptr+sizeof(unsigned int)` et donc `ptr+1` correspondra à l'adresse `0x7fff5fbff754`. Pour la même raison, l'exécution de la deuxième ligne placera l'adresse `0x7fff5fbff758` dans `ptr`. Enfin, la dernière ligne calculera `0x7fff5fbff758-2*sizeof(unsigned int)`, ce qui correspond à `0x7fff5fbff750`.

Pointeur vers un pointeur

- Un pointeur est une “case” mémoire contenant l’adresse d’une variable ; On peut tout à fait déclarer un nouveau pointeur vers ce premier pointeur : Ce *pointeur vers un pointeur* (ou *pointeur de pointeur*) contiendra l’adresse du premier pointeur
- Un pointeur vers un `int` est déclaré `int *`
 - `int *ptr_vers_int;`
 - `int a;`
 - `ptr_vers_int = &a;`
- Un pointeur vers un pointeur vers un `int` est déclaré `int **`
 - `int **ptr_vers_ptr;`
 - `ptr_vers_ptr = &ptr_vers_int;`
- On peut suivre la double indirection avec l’opérateur `**`
 - `a = 3;`
 - `printf("La valeur de a est %d.\n", **ptr_vers_ptr);`
 - `printf("L'adresse de la variable a est %d.\n", *ptr_vers_ptr);`

Exemple de pointeur vers un pointeur

La signature de la fonction `main` :

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

est équivalente à :

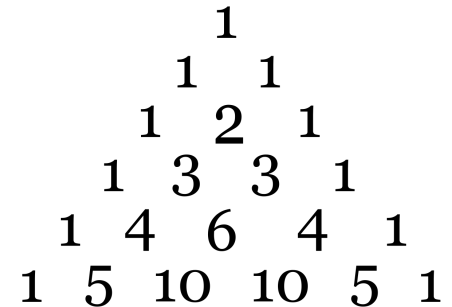
```
int main(int argc, char **argv)
```

- La liste des arguments est un *tableau* de pointeurs dont chaque élément pointe vers une chaîne de caractères (i.e. le premier `char` de cette chaîne)
 - `argv[0]` est un pointeur vers une chaîne de caractères (le nom du programme appelé depuis la ligne de commande)
 - ce tableau de pointeurs est terminé par le pointeur `NULL` (qui contient l'adresse 0)

```
int main(int argc, char **argv) {  
  
    char **p;  
    p=argv;  
    printf("Arguments :");  
    while(*p!=NULL) {  
        printf(" %s", *p);  
        p++;  
    }  
    printf("\n");  
    return(EXIT_SUCCESS);  
}
```


Utilisation des pointeurs vers des pointeurs

- On retrouve les pointeurs vers des pointeurs dans principalement deux situations :
 1. Lorsque l'on doit manipuler des structures multidimensionnelles
 - Par rapport à un tableau à deux dimensions pré-alloué (où chaque “ligne” contient le même nombre d’éléments — de “colonnes”), on peut envisager un nombre d’éléments différents pour chacune des listes pointées (ex : triangle de Pascal)
 2. Lorsque l'on souhaite qu'une fonction puisse manipuler une adresse qu'elle a reçue en argument
 - Si on passe la valeur du pointeur, la fonction en aura une copie dont la modification ne sera pas visible par l'appelant
 - En passant un pointeur vers le pointeur à modifier, la fonction pourra aller modifier l'adresse pointée par ce dernier



Exemple avec strtol

The **strtol()** function converts the initial part of the string in *nptr* to a long integer value according to the given *base*, which must be between 2 and 36 inclusive

```
#include <stdlib.h>
long
strtol(const char *restrict str, char **restrict endptr, int base);
```

pointeur vers la chaîne à
convertir

adresse d'un
pointeur vers un
chaîne de caractères

base de conversion (e.g. 8)

The remainder of the string is converted to a *long int* value in the obvious manner, stopping at the first character which is not a valid digit in the given base. (In bases above 10, the letter 'A' in either upper or lower case represents 10, 'B' represents 11, and so forth, with 'Z' representing 35.) If *endptr* is not NULL, **strtol()** stores the address of the first invalid character in **endptr*. If there were no digits at all, **strtol()** stores the original value of *nptr* in **endptr* (and returns 0). In particular, if **nptr* is not '\0' but ***endptr* is '\0' on return, the entire string is valid.

Exemple avec strtol

```
#include <stdlib.h>
long
strtol(const char *restrict str, char **restrict endptr, int base);
```

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
    char *p, *s;
    long li;
    s = "1252";
    li = strtol(s,&p,10);
    if(*p != '\0') {
        printf("Caractère erroné : %c\n",*p);
        // p pointe vers le caractère en erreur
    }
    printf("Valeur convertie : %s -> %ld\n",s,li);

    s = "12m52";
    li = strtol(s,&p,10);
    if(*p != '\0') {
        printf("Caractère erroné : %c\n",*p);
    }
    printf("Valeur convertie : %s -> %ld\n",s,li);

    return(EXIT_SUCCESS);
}
```

```
Valeur convertie : 1252 -> 1252
Caractère erroné : m
Valeur convertie : 12m52 -> 12
```

Structures

- On a souvent besoin de manipuler des données composées
 - Un nombre complexe
 - Une fiche étudiant avec nom, prénom, NOMA, etc.
- Il n'y a pas de notion d'objet en C comme il y a en Java
- Mais on peut déclarer des types de données composés sous forme de structures

```
// structure pour stocker une coordonnée 3D
struct coord {
    int x;
    int y;
    int z;
};

struct coord point = {1, 2, 3};
struct coord p;

// structure pour stocker une fraction
struct fraction {
    int numerator;
    int denominator;
};

struct fraction demi = {1, 2};
struct fraction f;

// structure pour représenter un étudiant
struct student {
    int matricule;
    char prenom[20];
    char nom[30];
};

struct student s = {1, "Linus", "Torvalds"};
```

Structures

- On définit une structure avec la construction
`struct coord { [type nom_element;]+ };`
- Définition d'une variable de type struct coord :
 - `struct coord ma_var = { [valeur,]+ };`
- Accès aux éléments en utilisant l'opérateur `.` :
e.g., si on a déclaré `struct coord point ;` :

```
point.x = 1;  
point.y = 2;  
point.z = 3;
```

Utilisation d'alias

- On peut donner un nom plus court que “struct fraction” pour un type de structure que nous avons déclaré en utilisant l'opérateur typedef
 - Il permet de renommer des types de données existants
 - On peut utiliser indifféremment l'ancien et le nouveau nom ensuite
 - Détails dans syllabus sur les bonnes/mauvaises pratiques

```
// structure pour stocker une fraction
typedef struct fraction {
    double numerator;
    double denominator;
} Fraction ;

typedef int Entier;

int main(int argc, char *argv[])
{
    Fraction demi = {1, 2};
    Entier i = 2;
    // ...
    return EXIT_SUCCESS;
}
```

Fonctions

- On peut déclarer une fonction sous le format :
`type_retour nom_fonction`
`([type_arg nom_arg]+) { ... }`
 - `void` : pas de résultat
- Passage des arguments par valeur
 - La fonction reçoit une *copie* de la valeur dans une zone mémoire propre à cette fonction
 - Les allocations sont locales à la fonction
- Si la fonction doit modifier une valeur visible par l'appelant, il faut utiliser un pointeur !

```
int times_two(int *n)
{
    return (*n) + (*n);
}

int timestwo(int *n)
{
    *n = (*n) + (*n);
    return *n;
}

void f()
{
    int i = 1252;
    printf("i:%d\n", i);
    printf("times_two(&i)=%d\n", times_two(&i));
    printf("après times_two, i:%d\n", i);
    printf("timestwo(&i)=%d\n", timestwo(&i));
    printf("après timestwo, i:%d\n", i);
}
```

```
i:1252
times_two(&i)=2504
après times_two, i:1252
timestwo(&i)=2504
après timestwo, i:2504
```

Exemples

```
int length(char *s)
{
    int i = 0;
    while (*(s+i) != '\0')
        i++;
    return i;
}
```

```
void plusun(int size, int *v)
{
    int i;
    for (i = 0; i < size; i++)
        v[i]++;
}

void print_vecteur(int size, int*v) {
    int i;
    printf("v={");
    for (i = 0; i < size - 1; i++)
        printf("%d,", v[i]);

    if (size > 0)
        printf("%d}", v[size - 1]);
    else
        printf("}");
}
```

```
int vecteur[N] = {1, 2, 3, 4, 5};
plusun(N, vecteur);
print_vecteur(N, vecteur);
```


Fonctions et pointeurs

```

struct fraction init(int num, int den)
{
    struct fraction f;
    f.numerator = num;
    f.denominator = den;
    return f;
}

int equal(struct fraction f1, struct fraction f2)
{
    return ((f1.numerator == f2.numerator)
        && (f1.denominator == f2.denominator));
}

int equalptr(struct fraction *f1, struct fraction *f2)
{
    return ((f1->numerator==f2->numerator)
        && (f1->denominator==f2->denominator));
}

void initptr(struct fraction *f, int num, int den)
{
    f->numerator = num;
    f->denominator = den;
}

```

opérateur flèche : sucre
syntaxique bien pratique pour
accéder aux éléments d'une
structure passée par pointeur :
f->numerator est identique
à (*f).numerator

```

struct fraction quart;
struct fraction tiers;
quart = init(1, 4);
initptr(&tiers, 1, 3);
printf("equal(tiers,quart)=%d\n", equal(tiers, quart));
printf("equalptr(&tiers,&quart)=%d\n", equalptr(&tiers, &quart));

```

Fonctions et déclarations

- Le compilateur C analyse le code de façon linéaire, en une seule passe
- Une fonction doit avoir été définie avant d'être utilisée
- Petits programmes : on peut jouer sur l'ordre des fonctions dans le fichier source
- Mais plus généralement, on déclare les signatures des fonctions au début, avant de donner leur implémentation
 - Ces signatures sont souvent regroupées dans un fichier de header dont le nom se termine par `.h`
 - Comme nous l'avons vu pour les headers des librairies standards !

```
int times_two(int *);  
int timestwo(int *);
```

Pointeur vers une fonction

- Nous avons vu des pointeurs vers des variables (`int`, `char`, ...) et vers des pointeurs
- Une fonction est une séquence d'instructions
 - Cette séquence commence à l'adresse mémoire où est stockée sa première instruction
- On peut stocker l'adresse d'une fonction dans un *pointeur de fonction*
 - Déclaration : `type_retour (*ptr_vers_func) ([type_arg]+);`
 - `void (*f1)(int, int, char*);`
 - `int (*f2)(int);`
 - `f2 = &ma_function;`
 ou `f2 = ma_function;` (conversion implicite, déconseillé car ambigu)
- L'appel se fait en *déréférencant* avec `*` le pointeur sur fonction ; ceci doit être fait entre parenthèses :
 - `int ret = (*f2)(3);`
 ou `f2(3);` (conversion implicite, déconseillé car ambigu)

Exemple d'utilisation d'un pointeur vers une fonction

```
void qsort(void *base, size_t nel, size_t width,
          int (*compar)(const void *, const void *));
```

Argument	Rôle
<code>void *base</code>	Début d'une zone mémoire à trier
<code>size_t nel</code>	Nombre d'éléments à trier
<code>size_t width</code>	Taille de chaque élément
<code>int (*compar)(const void *, const void *)</code>	<p>Fonction de comparaison qui prend deux arguments : retourne un <code>int < 0</code> si premier élément précède (e.g. plus petit) le 2ème ; ou un nombre positif ou nul sinon</p> <p>Les pointeurs sont définis comme <code>void *</code> (pointeur sans type) pour rendre la fonction générique — c'est pourquoi la taille des éléments doit être fournie !</p> <p>Le mot clé <code>const</code> indique au compilateur que la fonction n'a pas l'autorisation de modifier la donnée référencée par ce pointeur (c'est une mesure de prudence)</p>

Exemple d'utilisation d'un pointeur vers une fonction

```
void qsort(void *base, size_t nel, size_t width,  
          int (*compar)(const void *, const void *));
```

```
#define SIZE 5  
double array[SIZE]= { 1.0, 7.32, -3.43, 8.7, 9.99 };  
  
void print_array() {  
    for(int i=0;i<SIZE;i++)  
        printf("array[i]:%f\n",array[i]);  
}  
  
int cmp(const void *ptr1, const void *ptr2) {  
    const double *a=ptr1;  
    const double *b=ptr2;  
    if(*a==*b)  
        return 0;  
    else  
        if(*a<*b) return -1;  
        else return +1;  
}  
  
int main(int argc, char *argv[]) {  
    printf("Avant qsort\n\n");  
    print_array();  
    qsort(array,SIZE,sizeof(double),cmp);  
    printf("Après qsort\n\n");  
    print_array();  
    return(EXIT_SUCCESS);  
}
```

Expressions de manipulation de bits

- Il est parfois nécessaire de réaliser des opérations binaires sur des représentations de valeurs en mémoire
- Opérateur unaire : un seul argument
 - \sim : inversion des bits
- Opérateur binaire : deux arguments
 - $\&$ — ET : chaque bit du retour est 1 seulement si **les deux** bits correspondants des deux arguments sont 1, 0 sinon
 - $|$ — OU : chaque bit du retour est 1 si **un ou les deux** bits correspondants des deux arguments est 1, 0 sinon
 - \wedge — OU exclusif (XOR) : chaque bit du retour est 1 seulement si **un seul des deux** bits correspondants des deux arguments est 1, 0 sinon

A	NOT(A)
0	1
1	0

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

```

~ 00000000 = 11111111
11111010 & 01011111 = 01011010
11111010 | 01011111 = 11111111
11111010 ^ 01011111 = 10100101
  
```

```

r = ~a;    // négation bit à bit
r = a & b; // conjonction bit à bit
r = a | b; // disjonction bit à bit
r = a ^ b; // xor bit à bit
  
```

Utilisation des manipulations de bits

- Forcer un bit à zéro avec un ET logique ou forcer un bit à un avec un OU logique
- Les différents bits peuvent, par exemple, représenter des paramètres ou options sans avoir besoin d'utiliser un char pour chacun d'entre eux :

```
r = c & 0x7E; // 0b01111110 force les bits de poids faible et fort à 0
```

```
r = d | 0x18; // 0b00011000 force les bits 4 et 3 à 1
```

- On peut isoler un de ces bits en utilisant un masque :

```
v = r & 0x08; // 0b00001000 récupère seulement la valeur du bit 3
```

Expressions logiques et opérateurs binaires

- Les symboles utilisés pour les expressions logiques, `||` et `&&` (e.g. dans la condition de terminaison d'une boucle `for`) sont très ressemblants aux opérateurs binaires `|` et `&`
- Différence importante : l'évaluation de `|` et `&` considère les deux arguments, alors que `&&` ne va pas considérer le deuxième si le premier est vrai (`!=0`), resp. faux (`==0`)
 - `((ptr != NULL) && (ptr->den > 0))` est OK même si `ptr` est `NULL`
 - `((ptr != NULL) & (ptr->den > 0))` va essayer de déréférencer `ptr` qui est `NULL`, ce qui va déclencher une interruption par le processeur pour instruction illégale (et le SE terminera le processus avec une *segmentation fault*)

Déclarations

Portée d'une variable

- Deux types de portées
- Portée globale
 - Une variable définie en dehors de toute fonction
 - Accessible dans toutes les fonctions présentes dans le fichier
 - Pas 2 variables globales de même nom !
- Portée locale
 - Déclarée à l'intérieur d'une fonction ou d'un bloc de code { } où elle est utilisée
 - Même nom : ok (e.g. indice de boucle i)

⚠ Une variable de portée locale du même nom qu'une globale sera seule visible !

```
float g;    // variable globale

int f(int i) {

    int n;    // variable locale
    // ...
    for(int j=0; j<n; j++) { // variable locale
        // ...
    }

    //...

    for(int j=0; j<n; j++) { // variable locale
        // ...
        int n = 3;
    }
}
```

↑
définition la plus proche utilisée !

Constantes

- Avant C99: seulement avec directive define du pré-processeur
 - Remplacement de texte dans le code du programme
 - Seulement nombres et chaînes
- C99 : most-clé const
 - La valeur est stockée en mémoire à une adresse spécifique
 - N'importe quel type de données, incluant structures
 - Facilite le débogage

```
// extrait de <math.h>
#define M_PI
3.14159265358979323846264338327950288;

const double
pi=3.14159265358979323846264338327950288;

const struct fraction {
    int num;
    int denom;
} demi={1,2};
```

Variables statiques

- Une variable globale est définie pour tous les modules d'un programme
 - Différents fichiers sources du programme
 - Problème si le même nom de variable est utilisé comme globale dans différents fichiers !
- Variable définie comme `static`
 - En dehors d'un bloc : accessible à toutes les fonctions de ce module, mais pas aux autres modules
 - Dans un bloc (une fonction) : une variable définie comme `static` garde sa valeur d'une invocation à l'autre de la fonction
 - Et placée dans le segment des données initialisées

Conclusion

Conclusion

- Nous avons fait un survol assez rapide de C
- Si vous n'avez pas compris une notion
 - Relire le syllabus (plus de détails !)
 - Réaliser les exercices LEPL/SINCI503 si nécessaire
- Cette semaine :
 - Exercices Inginiuous sur les notions les plus avancées (non vues en LEPL/SINCI503 : obligatoires pour tout le monde)
- La semaine prochaine : Organisation de la mémoire et gestion de la mémoire dynamique