Méthodes informatiques et techniques de programmation

Cours

Millian Poquet
millian.poquet@inria.fr

23 décembre 2017

Sommaire

Programmation ?

Programmation : définition

La programmation est l'ensemble des activités qui permettent l'écriture des programmes informatiques. C'est une étape importante du développement de logiciels.

Programme ?

Programme : définition

C'est un ensemble d'opérations destinées à être exécutées par un ordinateur. Concrètement, en général, on peut dire qu'un programme est un fichier exécutable.

Programme ?

Programme : définition

C'est un ensemble d'opérations destinées à être exécutées par un ordinateur. Concrètement, en général, on peut dire qu'un programme est un fichier exécutable.

Un monde de programmes

Les programmes sont partout aujourd'hui : logiciels sur PC, applications sur smartphone, systèmes d'exploitation, véhicules, domotique...

Programme ?

Programme : définition

C'est un ensemble d'opérations destinées à être exécutées par un ordinateur. Concrètement, en général, on peut dire qu'un programme est un fichier exécutable.

Un monde de programmes

Les programmes sont partout aujourd'hui : logiciels sur PC, applications sur smartphone, systèmes d'exploitation, véhicules, domotique...

Types de programmes

On peut différencier deux grands types de programmes et par extension de langages de programmation : les langages interprétés et les langages compilés.

Langages interprétés

Dans certains langages de programmation dits **interprétés**, le code source d'un programme et le programme lui-même se confondent. C'est le cas de bash, python, PHP... Le code source est alors lu par un programme nommé **interpréteur**.

Interprétation et compilation

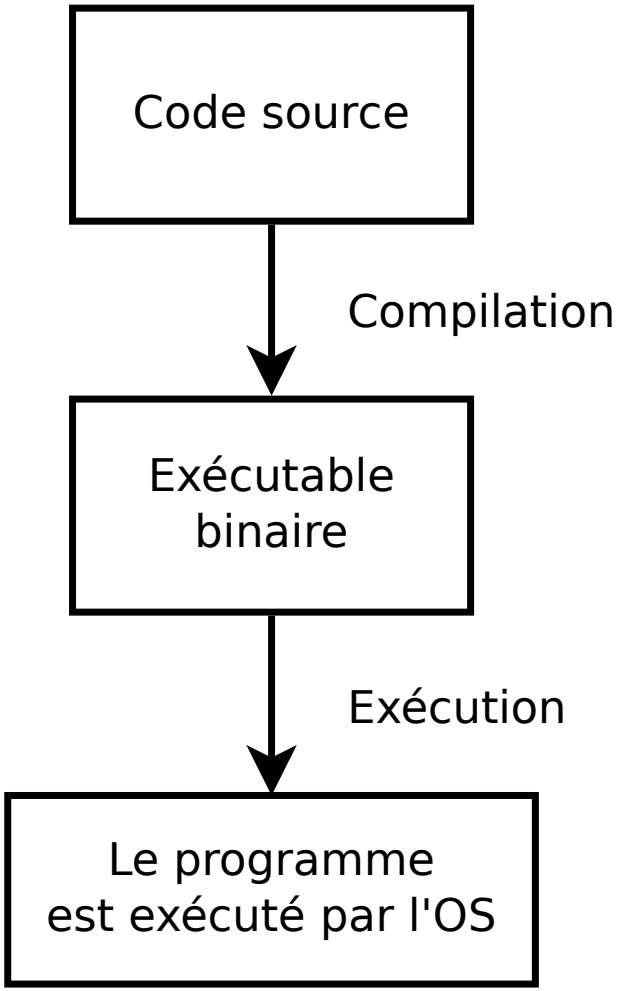
Langages interprétés

Dans certains langages de programmation dits **interprétés**, le code source d'un programme et le programme lui-même se confondent. C'est le cas de bash, python, PHP... Le code source est alors lu par un programme nommé **interpréteur**.

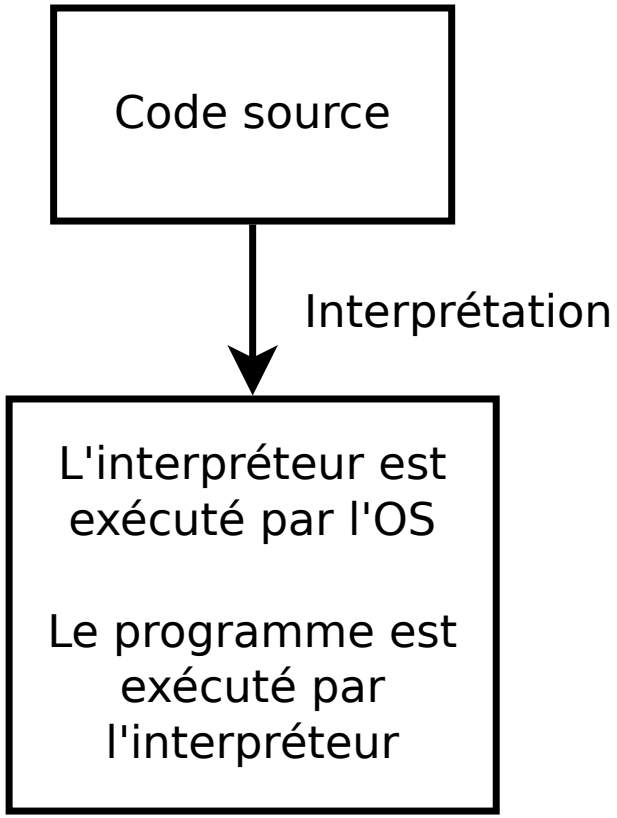
Langages compilés

D'autres langages ne sont pas directement compréhensibles par l'ordinateur : le code source doit être **compilé** vers un langage que la machine peut comprendre (et exécuter). C'est le cas du C, du C++...

Interprétation et compilation (suite)



Langage compilé



Langage interprété

Exemple d'interprétation (python)

Fichier source python *answer.py*

```
answer = 6 * 7
print('The answer is', answer)
```

Exemple d'interprétation (python)

Fichier source python *answer.py*

```
answer = 6 * 7
print('The answer is', answer)
```

Interprétation

```
# Appel de l'interpréteur
python answer.py
# Affichage du programme
The answer is 42
```

Exemple d'interprétation (js)

Fichier source JavaScript *batman.js*

```
a = Array(16).join("XKCD"-1) + " Batman"
print(a)
```

Exemple d'interprétation (js)

Fichier source JavaScript *batman.js*

```
a = Array(16).join("XKCD"-1) + " Batman"
print(a)
```

Interprétation

```
# Appel de l'interpréteur
js24 batman.js
# Affichage du programme
NaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaN Batman
```

Exemple de compilation

Fichier source C *hello.c*

```c
#include <stdio.h>
int main(void)
{
    printf("Hello world!\n");
    return 0;
}
```

Exemple de compilation

Fichier source C *hello.c*

```c
#include <stdio.h>
int main(void)
{
    printf("Hello world!\n");
    return 0;
}
```

Compilation

On crée le programme **hello** à partir du fichier source **hello.c**

```
clang hello.c -o hello
#gcc hello.c -o hello
```

Exemple de compilation

Fichier source C *hello.c*

```c
#include <stdio.h>
int main(void)
{
    printf("Hello world!\n");
    return 0;
}
```

Compilation

On crée le programme **hello** à partir du fichier source **hello.c**

```
clang hello.c -o hello
#gcc hello.c -o hello
```

Exécution

```
./hello
Hello world!
```

Quels outils pour développer ?

Plusieurs outils sont à votre disposition :

- Le compilateur : c'est un programme chargé de transformer votre *code source* en *programme exécutable*.
  Exemples : clang, gcc, icc...

Quels outils pour développer ?

Plusieurs outils sont à votre disposition :

- Le compilateur : c'est un programme chargé de transformer votre *code source* en *programme exécutable*.
  Exemples : clang, gcc, icc...
- L'éditeur de texte : c'est un programme qui vous permet de lire et d'éditer votre *code source*.
  Exemples : Sublime Text, Notepad++, Atom, emacs, vi, nano...

Quels outils pour développer ?

Plusieurs outils sont à votre disposition :

- Le compilateur : c'est un programme chargé de transformer votre *code source* en *programme exécutable*.
  Exemples : clang, gcc, icc...
- L'éditeur de texte : c'est un programme qui vous permet de lire et d'éditer votre *code source*.
  Exemples : Sublime Text, Notepad++, Atom, emacs, vi, nano...
- Le debugger : c'est un programme qui vous aide à détecter et à corriger des bugs.
  Exemple : gdb, valgrind, kdbg...

Quels outils pour développer ?

Plusieurs outils sont à votre disposition :

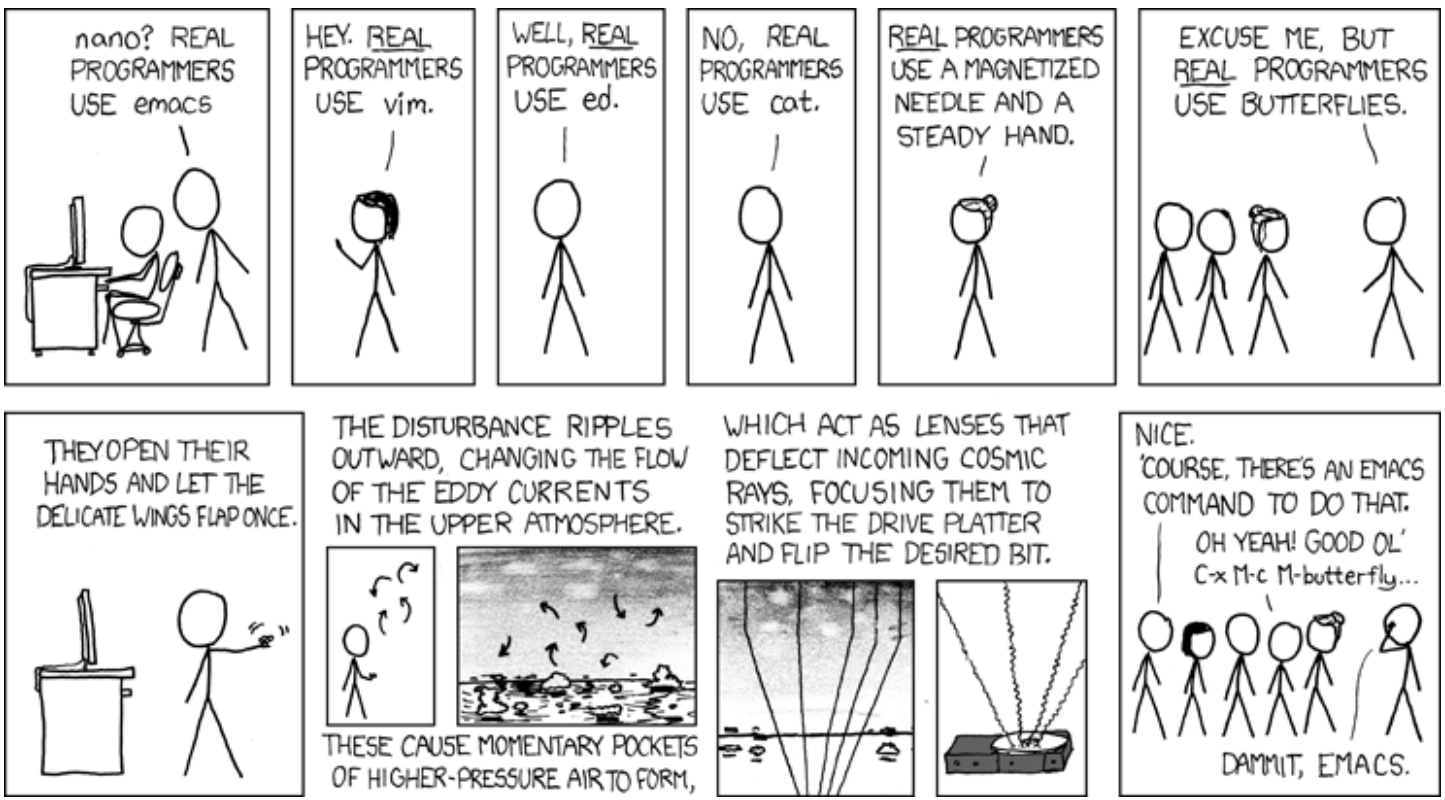
- Le compilateur : c'est un programme chargé de transformer votre *code source* en *programme exécutable*.
  Exemples : clang, gcc, icc...
- L'éditeur de texte : c'est un programme qui vous permet de lire et d'éditer votre *code source*.
  Exemples : Sublime Text, Notepad++, Atom, emacs, vi, nano...
- Le debugger : c'est un programme qui vous aide à détecter et à corriger des bugs.
  Exemple : gdb, valgrind, kdbg...
- Les environnements de développement intégrés (IDE) : ces programmes combinent les programmes précédents dans un seul logiciel.
  Exemples : Qt Creator, Code : :Blocks, Visual Studio, Xcode...

Quels outils pour développer ?

Essayez les différents outils, puis choisissez les vôtres !

Critères de choix importants

- Ergonomie
- Puissance
- Intérêt pédagogique



Quels outils choisir ? (1)

Compilateur + Éditeur de texte + Debugger

- Ce combo marche pour tous les langages de programmation.
- Cela vous permet de comprendre le rôle de chaque outil et de les maîtriser, ce qui est primordial si vous êtes de futurs informaticiens.
- C'est la combinaison la plus simple et la plus rapide à mettre en place sous Linux.
- Vous pouvez facilement changer un outil en conservant vos habitudes sur les autres.

Si vous faites ce choix, je vous conseille d'opter pour **clang**, **Sublime Text** et **kdbg** (une surcouche graphique à gdb) + **valgrind** (un outil simple à utiliser de vérification des manipulations mémoire).

Quels outils choisir ? (2)

IDE

- Il existe de nombreux IDE pour le C (même si la plupart gèrent le C++ en priorité puis le C).
- Le plus pratique sous Windows.
- Plus compliqué de comprendre les mécanismes derrière la création de programmes, attention en examen.
- Un IDE fonctionne par projet : il permet de modifier un projet, de le paramétrer, de le compiler et de le débugger. Vous devrez donc créer un projet pour chaque fichier source dans les TP, ce qui allonge le temps de mise en place.
- Meilleure autocomplétion et refactoring qu'un éditeur de texte.

Si vous faites ce choix, je vous conseille d'opter pour **Qt Creator** plutôt que `Code::Blocks` car Qt Creator est, à mon goût, plus stable, plus ergonomique et plus facilement configurable que `Code::Blocks`.

Le compilateur (1)

Conseil

Le **compilateur** est le programme qui permet de transformer vos fichiers sources en programmes exécutables.
C'est votre ami ! Les messages qu'il vous affiche sont là pour vous aider et sont riches en informations. Lisez-les bien, qu'il s'agisse d'erreurs ou d'avertissements.
Les avertissements sont des bugs potentiels, corrigez-les !

Options de compilation

Les compilateurs, comme beaucoup d'autres programmes, sont paramétrables via des *options*.
Nous allons utiliser la norme `ISO/IEC 9899:1999` du langage C, plus communément appelée C99.

Le compilateur (2)

Comment appeler le compilateur ?

```
# Appel simple du compilateur
clang hello.c -o hello
# gcc et clang s'utilisent de la même manière.

# Appel simple avec la norme C99
clang -std=c99 hello.c -o hello

# On ajoute la plupart des avertissements
clang -std=c99 -Wall -Wextra -Wfatal-errors hello.c -o
↪  hello

# Création d'un alias pour ne pas tout taper à chaque fois
alias clang99='clang -std=c99 -Wall -Wextra
↪  -Wfatal-errors'

# Utilisation de l'alias
clang99 hello.c -o hello
```

Le compilateur (3)

Comment appeler le compilateur ? (suite)

```
# On compile en mode 'debug'
clang99 -g hello.c -o hello

# On souhaite obtenir une version optimisée du code
clang99 -O2 hello.c -o hello

# On souhaite compiler plusieurs fichiers
clang99 hello.c good.c morning.c -o sentence

# On souhaite utiliser des bibliothèques (OpenGL)
clang99 -lgl hello.c
```

Quel environnement pour développer ? (1)

GNU/Linux.

Pourquoi ?

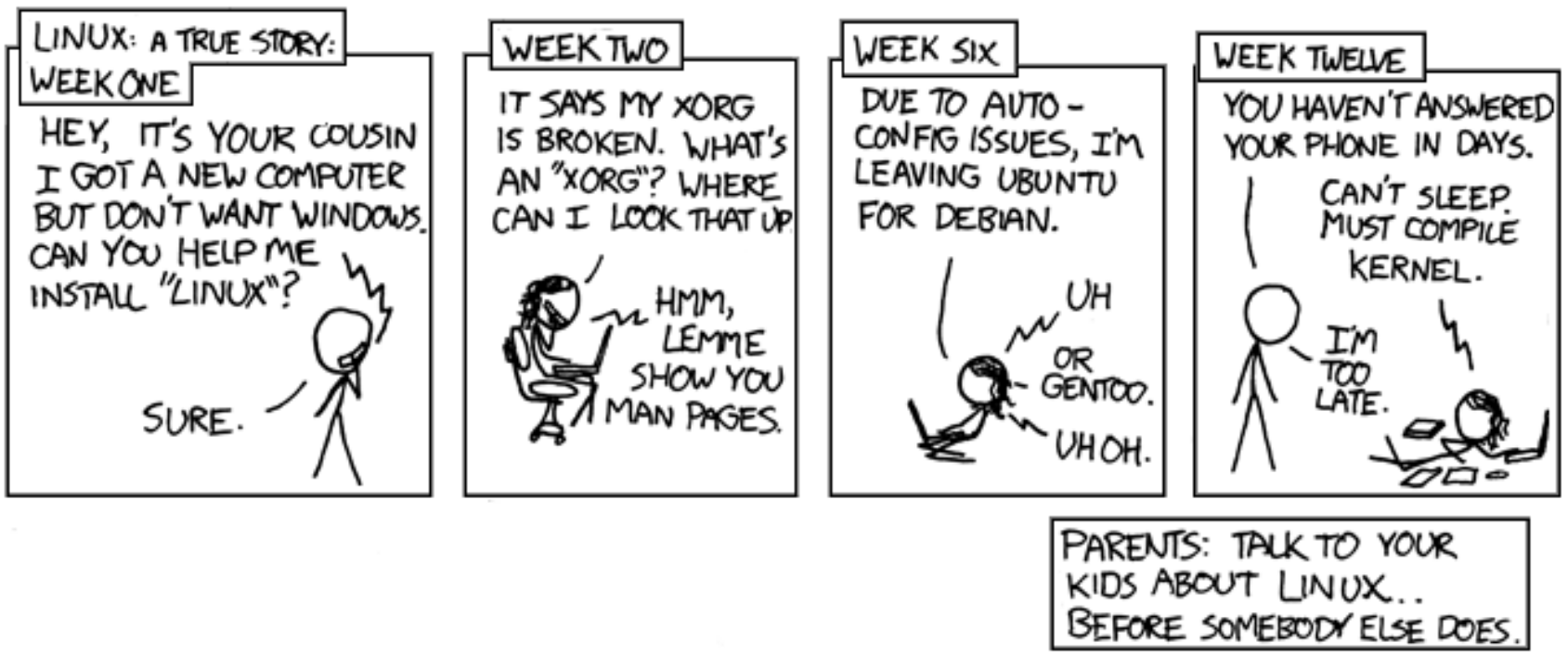
- Libre, la plupart des distro sont gratuites
- Pédagogique
  - On peut ouvrir le capot
  - Outils simples et spécifiques
- Très pratique pour le développement
  - Outils simples à combiner
  - Packaging de bibliothèques
  - Chaîne de compilation maîtrisée

Quel environnement pour développer ? (2)

Il existe beaucoup de **distributions** GNU/Linux différentes !

Quelle distribution choisir ?

- Les plus accessibles : Debian, Ubuntu...
- Plus avancé, orienté développeurs : Arch, Gentoo, NixOS...
- Trop simple ? LFS...



Analysons hello.c...

```c
#include <stdio.h>
int main(void)
{
    printf("Hello world!\n");
    return 0;
}
```

Explications de hello.c

```c
/* Inclut le fichier standard d'entrées-sorties,
   afin d'utiliser la fonction printf */
#include <stdio.h>

// Fonction principale.
// C'est ici que le programme commence !
int main(void)
{
    printf("Hello world!\n"); // On affiche du texte
    return 0; // On quitte tout le programme
}
```

Explications (plus détaillées) de hello.c

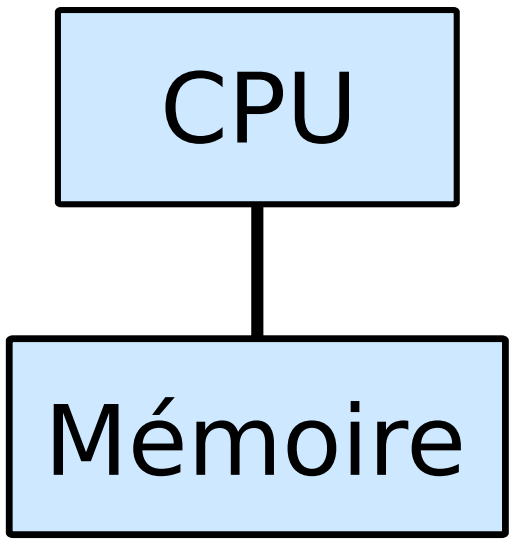
```c
/* Inclut le fichier stdio.h, situé dans
   le système (chevrons) */
#include <stdio.h>

// Fonction principale.
// int : Cette fonction renvoie un entier
// void : Cette fonction ne prend pas de paramètres
int main(void)
{
    /* On appelle la fonction printf
       Elle prend du texte en paramètre*/
    printf("Hello world!\n");

    return 0; // On quitte la fonction en renvoyant 0
}
// Lorsqu'on quitte la fonction main, le programme
// s'arrête. 0 veut dire que tout s'est bien passé.
```

Sommaire

Architecture d'un ordinateur (de très très haut)



Le CPU

Le processeur (ou *Central Processing Unit*) permet d'exécuter les instructions machines des programmes informatiques.

La mémoire

Elle permet de stocker des informations.

Comment la voir en tant que développeur ?

C'est un très gros tableau d'octets (1 octet = 8 bits).

Sur un octet, on peut stocker $2^8 = 256$ valeurs différentes, que l'on peut représenter par des entiers entre 0 et 255.

Chaque octet est situé à une **adresse** bien précise.

Adresse	Valeur
0	154
1	83
2	4
3	233

Une mémoire de 4 octets

La mémoire (suite)

Comment accéder à la mémoire ?

En C, on n'interagit généralement pas avec la mémoire directement (en spécifiant quelles adresses mémoire on souhaite utiliser).

On se sert de **variables**, qui permettent de stocker des informations dans la mémoire.

Adresse	Valeur	Variable
0	0	i
1	0	
2	0	
3	0	
4	'a'	c1
5	'b'	c2
6	0	s
7	0	

Une mémoire de 8 octets

Les variables

Elles permettent de stocker des informations dans la mémoire. En C, les variables ont un **type**. On dit que le C est un langage typé.

Les variables

Elles permettent de stocker des informations dans la mémoire. En C, les variables ont un **type**. On dit que le C est un langage typé.

Les principaux types utilisés en C

- char : stocke un caractère. Exemples : 'a', 'X' ou '8'
- int : stocke un nombre entier. Exemple : 0, -10 ou 116
- float : stocke un nombre réel. Exemple : 3.14, 0 ou 2e10
- double : stocke un nombre réel (de manière plus précise)
- bool (à partir de C99) : stocke un booléen (true ou false)

Que peut-on faire avec des variables ? I

La déclaration

La **déclaration** d'une ou plusieurs variables permet d'indiquer au compilateur l'existence de ces variables.

La définition

La **définition** d'une variable **instancie** la variable : elle la crée.
En C, la définition inclut la déclaration, ce qui permet d'omettre la déclaration très souvent, notamment pour les variables.

```
// Définition d'un entier i
int i;
// Définition d'un booléen ok
bool ok;
// Définition de deux flottants f et f2
float f, f2;
```

Que peut-on faire avec des variables ? II

L'affectation

L'**affectation** d'une variable permet d'attribuer une valeur à une variable.

```
// Affectation du flottant f à 4.2
f = 4.2;
// Affectation du booléen ok à vrai
ok = true;
```

Que peut-on faire avec des variables ? III

L'initialisation

L'**initialisation** d'une variable est l'affectation d'une variable à une valeur initiale.

```
// Définition de l'entier e
int e;
// Initialisation de e à 0 (affectation)
e = 0;
// Définition + initialisation
float z = 42.51;
```

En C, si vous n'initialisez pas vos variables, leur contenu est indéterminé !

Que peut-on faire avec des variables ? IV

La lecture

La **lecture** d'une variable consiste à utiliser la valeur liée à la variable.

```
k; // k est lu
f2 = f * f; // f est lu (deux fois)
printf("f2 vaut %f\n", f2); // f2 est lu
```

La suppression

La **suppression** d'une variable annule sa définition (libère la mémoire occupée par la variable) et sa déclaration (permet de déclarer plus tard une variable de même nom).
En C, la plupart des suppressions sont automatiques (détails en section 8 et en section 15).

Détails sur les types entiers

Les types entiers utilisés en C (x86-64)

Type	Taille (o)	Valeur min	Valeur max
(signed) char	1	-127	128
unsigned char	1	0	255
(signed) short (int)	2	-32767	32768
unsigned short (int)	2	0	65535
(signed) int	4	$-2^{31}+1$	$2^{31}-1 \approx 2 \cdot 10^9$
unsigned (int)	4	0	$2^{32}-1 \approx 4 \cdot 10^9$
(signed) long long (int)	8	$-2^{63}+1$	$2^{63}-1 \approx 9 \cdot 10^{18}$
unsigned long long (int)	8	0	$2^{64}-1 \approx 2 \cdot 10^{19}$

Et les variantes du *long* ?

En 32 et 64 bits, on peut considérer que int et long sont synonymes.

Détails sur les caractères (1)

Le type **char** est un entier de taille 1 octet, mais il est surtout utilisé pour représenter des caractères.

Pour cela, on a associé un nombre entier à chaque caractère. Différentes tables définissent comment cette association est faite. La plus importante est la table ASCII.

```c
#include <stdio.h>
int main(void)
{
    char c = 'a';
    char d = 97;
    char diff = c - d;
    printf("diff = %d\n",diff);
    printf("c = %c\n", c);
    printf("d = %c\n", d);
    return 0;
}
```

Résultat de l'exécution

```
diff = 0
c = a
d = a
```

Détails sur les caractères (2) : table ASCII

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Détails sur les types flottants (1)

La **virgule flottante** est une méthode d'écriture des nombres réels. Elle décompose un nombre r en $r = s \cdot m \cdot b^e$ où

- s est le signe : 1 ou -1
- m est la mantisse (ou *significande*) : un nombre entier qui représente la valeur *significative* du nombre
- b est une base (nombre entier)
- e est un exposant (nombre entier)

mantisse exposant

$$1.4251 = + 14251 \times 10^{-4}$$

signe base

Détails sur les types flottants (2)

La norme C en dit peu sur la manière dont la virgule flottante doit être mise en place. En pratique, la norme IEEE 754 est utilisée.

Précision	Taille	Représentation	Signe	Exposant	Mantisse
Simple	32	$(-1)^s \times m \times 2^{e-127}$	1	8	23
Double	64	$(-1)^s \times m \times 2^{e-1023}$	1	11	52

Les flottants définis par la norme IEEE 754. Les tailles sont en bits.

Détails sur les types flottants (2)

La norme C en dit peu sur la manière dont la virgule flottante doit être mise en place. En pratique, la norme IEEE 754 est utilisée.

Précision	Taille	Représentation	Signe	Exposant	Mantisse
Simple	32	$(-1)^s \times m \times 2^{e-127}$	1	8	23
Double	64	$(-1)^s \times m \times 2^{e-1023}$	1	11	52

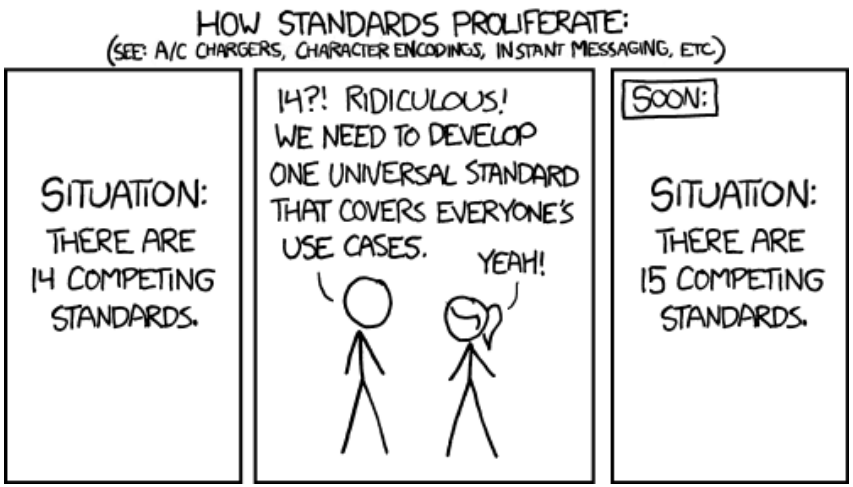
Les flottants définis par la norme IEEE 754. Les tailles sont en bits.

Remarque importante

Les nombres flottants ne sont pas équivalents aux réels !
Voyez-les plutôt comme des rationnels à précision limitée.

Détails sur les types flottants (3) : aperçu en x86

Type C	Implémentation	Taille (b)
float	IEEE 754 simple-precision	32
double	IEEE 754 double-precision	64
long double	IEEE 754 double-precision (Visual C++)	64
	x86 80-bit precision (GCC, clang... en x86)	80 ou 96 ou 128
	2 * IEEE 754 double-precision (PowerPC...)	128
	IEEE 754 quadruple-precision (SPARC...)	128



Sommaire

Les entrées-sorties

On appelle **entrées-sorties** (*input/output*, *I/O* ou *IO*) les échanges d'informations entre le programme et ce qui lui est extérieur.

Exemples d'entrées

- L'utilisateur a saisi une ligne de texte
- L'utilisateur déplace la souris
- Un paquet a été reçu par la carte réseau

Exemples de sorties

- Le programme écrit du texte dans un fichier
- Le programme affiche des pixels sur l'écran

Les entrées-sorties (suite)

Heureusement pour nous, nous n'avons pas à interagir directement avec les différents périphériques pour faire des IO.

Un des rôles du **système d'exploitation** est de gérer les périphériques et de simplifier la manière d'interagir avec eux.

Du point de vue d'un programmeur, cette interaction se fait généralement via des fonctions définies dans des **bibliothèques** (*libraries*). Nous allons voir quelques fonctions de la bibliothèque standard du C.

La fonction printf

Prototype

```c
#include <stdio.h>
int printf(const char *format, ...);
```

Description

- La fonction **printf** permet d'afficher du texte à l'écran en fonction d'un certain **format**, qui est son premier paramètre.
- Ce format est une chaîne de caractères (ce sera détaillé dans la suite du cours, vous pouvez vous dire que c'est un texte entouré de caractères " pour l'instant).
- Elle renvoie le nombre de caractères qui ont été affichés à l'écran.
- Cette fonction est spéciale, elle accepte (après le format) un nombre variable de paramètres qui peuvent être de différents types. C'est ce que l'on appelle une fonction **variadique**.

La fonction printf : exemple

```c
#include <stdio.h>
#include <stdbool.h>
int main()
{
    printf("Juste du texte\n");
    printf("Caractères : %c %c \n", 'a', 65);
    printf("Entiers (affichés en décimal) : %d %d\n", 1977, 3722);
    printf("Entiers (autres) : %d %x %o\n", 64, 64, 64);
    printf("Réels : %f %4.2f\n", 3.1416, 3.1416);
    printf("Booléens : %d %d\n", true, false);
    return 0;
}
```

La fonction printf : exemple

```c
#include <stdio.h>
#include <stdbool.h>
int main()
{
    printf("Juste du texte\n");
    printf("Caractères : %c %c \n", 'a', 65);
    printf("Entiers (affichés en décimal) : %d %d\n", 1977, 3722);
    printf("Entiers (autres) : %d %x %o\n", 64, 64, 64);
    printf("Réels : %f %4.2f\n", 3.1416, 3.1416);
    printf("Booléens : %d %d\n", true, false);
    return 0;
}
```

```
Juste du texte
Caractères : a A
Entiers (affichés en décimal) : 1977 3722
Entiers (autres) : 64 40 100
Réels : 3.141600 3.14
Booléens : 1 0
```

La fonction printf : formatage

Mais qu'est-ce donc ?

Le format est **toujours** le premier paramètre de la fonction **printf**. C'est le texte qui sera affiché par la fonction.

La fonction printf : formatage

Mais qu'est-ce donc ?

Le format est **toujours** le premier paramètre de la fonction **printf**.
C'est le texte qui sera affiché par la fonction.
Ce texte peut optionnellement contenir des **spécificateurs de format** qui sont remplacés par les valeurs des paramètres suivants de la fonction et affichés dans le format que l'on a indiqué.

La fonction printf : formatage

Mais qu'est-ce donc ?

Le format est **toujours** le premier paramètre de la fonction **printf**.
C'est le texte qui sera affiché par la fonction.
Ce texte peut optionnellement contenir des **spécificateurs de format** qui sont remplacés par les valeurs des paramètres suivants de la fonction et affichés dans le format que l'on a indiqué.

Syntaxe des spécificateurs de format

`%[flags][width][.precision][length]specifier`

La fonction printf : formatage

Mais qu'est-ce donc ?

Le format est **toujours** le premier paramètre de la fonction **printf**.
C'est le texte qui sera affiché par la fonction.
Ce texte peut optionnellement contenir des **spécificateurs de format** qui sont remplacés par les valeurs des paramètres suivants de la fonction et affichés dans le format que l'on a indiqué.

Syntaxe des spécificateurs de format

`%[flags][width][.precision][length]specifier`

Et en simple ?

`%specifier`

où specifier dépend du type d'affichage souhaité.

La fonction printf : formatage (suite)

Les spécificateurs les plus souvent rencontrés

Spécificateur	Signification	Affichage
d ou i	Entier décimal signé	-36
f	Flottant	1.420000
c	Caractère	h
u	Entier décimal positif	4427
e	Notation scientifique	3.42e+7
g	Représentation flottante la plus courte	1.5
s	Chaîne de caractères	Bonjour
p	Adresse	0x7ffc5f544588
%	Affiche le symbole %	%

La fonction printf : formatage (suite)

Les spécificateurs les plus souvent rencontrés

Spécificateur	Signification	Affichage
d ou i	Entier décimal signé	-36
f	Flottant	1.420000
c	Caractère	h
u	Entier décimal positif	4427
e	Notation scientifique	3.42e+7
g	Représentation flottante la plus courte	1.5
s	Chaîne de caractères	Bonjour
p	Adresse	0x7ffc5f544588
%	Affiche le symbole %	%

Je veux faire des choses plus avancées !

RTFM !

- man 3 printf
- documentation de la fonction (sur une page web, par exemple)

La fonction scanf

Prototype

```c
#include <stdio.h>
int scanf(const char *format, ...);
```

- La fonction **scanf** lit l'entrée standard (par défaut, ce que l'utilisateur saisit au clavier) en fonction du **format** qui lui est spécifié.
- Si des spécificateurs sont donnés dans le format, les résultats de ces conversions sont placés aux **adresses** spécifiées par les paramètres suivants de la fonction.

La fonction scanf

Prototype

```c
#include <stdio.h>
int scanf(const char *format, ...);
```

- La fonction **scanf** lit l'entrée standard (par défaut, ce que l'utilisateur saisit au clavier) en fonction du **format** qui lui est spécifié.
- Si des spécificateurs sont donnés dans le format, les résultats de ces conversions sont placés aux **adresses** spécifiées par les paramètres suivants de la fonction.

Attention !

Le type des paramètres après le format doit être en adéquation avec le format. Par exemple, si vous spécifiez le format "%d%f%c" Il faut que les trois paramètres suivants soient respectivement l'**adresse** d'un entier, l'**adresse** d'un flottant et l'**adresse** d'un caractère.

La fonction scanf : exemple

```c
#include <stdio.h>
#include <stdbool.h>
int main()
{
    int ent; char car; float flo;

    printf("Entrez un caractère : ");
    scanf("%c", &car);

    printf("Entrez un entier et un réel : ");
    scanf("%d %f", &ent, &flo);

    printf("Vous avez saisi ent=%d, car=%c, flo=%f\n",
        ent, car, flo);
    return 0;
}
```

Le buffer d'entrée

Lorsque vous utilisez scanf, les données sont lues depuis le **tampon d'entrée** (*input buffer*). Ce tampon peut être vu comme une succession de caractères.

Si vous lisez sur l'entrée standard (le texte rentré dans le terminal qui exécute votre programme), vous ajoutez des caractères dans ce *buffer* à chaque pression de la touche **Entrée**.

Le buffer d'entrée en fonction de vos actions

Imaginons qu'on vous demande de saisir deux entiers puis un caractère.

Tête de lecture	↓															
Buffer d'entrée	•															

- Au début, le buffer d'entrée est vide (• marque la fin du buffer).

Le buffer d'entrée en fonction de vos actions

Imaginons qu'on vous demande de saisir deux entiers puis un caractère.

Tête de lecture	↓															
Buffer d'entrée	4	2	\n	•												

- Au début, le buffer d'entrée est vide (• marque la fin du buffer).
- Vous saisissez le texte "42" et appuyez sur **Entrée**

Le buffer d'entrée en fonction de vos actions

Imaginons qu'on vous demande de saisir deux entiers puis un caractère.

Tête de lecture			↓													
Buffer d'entrée	4	2	\n	•												

- Au début, le buffer d'entrée est vide (• marque la fin du buffer).
- Vous saisissez le texte "42" et appuyez sur **Entrée**
- `scanf("%d", &i); // vous renvoie 42`

Le buffer d'entrée en fonction de vos actions

Imaginons qu'on vous demande de saisir deux entiers puis un caractère.

Tête de lecture			↓													
Buffer d'entrée	4	2	\n	3	7	\n	•									

- Au début, le buffer d'entrée est vide (• marque la fin du buffer).
- Vous saisissez le texte "42" et appuyez sur **Entrée**
- `scanf("%d", &i); // vous renvoie 42`
- Vous saisissez le texte "37" et appuyez sur **Entrée**

Le buffer d'entrée en fonction de vos actions

Imaginons qu'on vous demande de saisir deux entiers puis un caractère.

Tête de lecture						↓										
Buffer d'entrée	4	2	\n	3	7	\n	•									

- Au début, le buffer d'entrée est vide (• marque la fin du buffer).
- Vous saisissez le texte "42" et appuyez sur **Entrée**
- `scanf("%d", &i); // vous renvoie 42`
- Vous saisissez le texte "37" et appuyez sur **Entrée**
- `scanf("%d", &j); // vous renvoie 37`

Le buffer d'entrée en fonction de vos actions

Imaginons qu'on vous demande de saisir deux entiers puis un caractère.

Tête de lecture						↓										
Buffer d'entrée	4	2	\n	3	7	\n	a	\n	•							

- Au début, le buffer d'entrée est vide (• marque la fin du buffer).
- Vous saisissez le texte "42" et appuyez sur **Entrée**
- `scanf("%d", &i); // vous renvoie 42`
- Vous saisissez le texte "37" et appuyez sur **Entrée**
- `scanf("%d", &j); // vous renvoie 37`
- Vous saisissez le texte "a" et appuyez sur **Entrée**

Le buffer d'entrée en fonction de vos actions

Imaginons qu'on vous demande de saisir deux entiers puis un caractère.

Tête de lecture							↓									
Buffer d'entrée	4	2	\n	3	7	\n	a	\n	•							

- Au début, le buffer d'entrée est vide (• marque la fin du buffer).
- Vous saisissez le texte "42" et appuyez sur **Entrée**
- `scanf("%d", &i); // vous renvoie 42`
- Vous saisissez le texte "37" et appuyez sur **Entrée**
- `scanf("%d", &j); // vous renvoie 37`
- Vous saisissez le texte "a" et appuyez sur **Entrée**
- `scanf("%c", &c); // vous renvoie '\n'`

Le buffer d'entrée (suite et fin)

Les données d'entrée que vous lisez dans vos programmes passent par des buffers intermédiaires.

Il est nécessaire d'avoir cette notion en tête dès que l'on récupère des données depuis l'extérieur. Vous trouverez le comportement de vos programmes très étrange sinon ;).

Il existe aussi des buffers de sortie, mais leur usage est presque transparent.

Saisie plus sûre de données

Problème

Comment effectuer des saisies de l'utilisateur de manière plus sûre ?

Confiance envers l'utilisateur

- Dans un cadre réaliste, vous ne pouvez pas faire confiance à l'utilisateur. Il est important de vérifier toute donnée venant de lui.
- Si vous ne le faites pas, un utilisateur maladroit risque de aire planter votre programme...
- Ou pire, un utilisateur mal intentionné pourrait utiliser votre programme pour faire autre chose que ce qu'il devrait faire...
- Cependant, nous allons lui faire confiance dans les exercices afin de ne pas compliquer les codes. **Ce ne sera pas le cas en projet**.

Saisie plus sûre de données (bis)

Méthode 1 : sûre et générique

- Récupérer toute la ligne de texte que l'utilisateur a rentré, en l'enlevant du buffer d'entrée.
- Extraire depuis cette ligne les informations qui nous intéressent (un entier par exemple, ou deux flottants et un caractère...),
- Vérifier que les informations saisies sont valides,
- Si elles ne le sont pas, redemander la saisie ou lancer un mécanisme d'erreur quelconque (quitter le programme par exemple)...

Saisie plus sûre de données : fonctions pour la méthode 1

```c
// Pour récupérer la ligne
#include <stdio.h>
ssize_t getline(char **lineptr, size_t *n, FILE
↪  *stream);
ssize_t getdelim(char **lineptr, size_t *n, int delim,
↪  FILE *stream);
char *fgets(char *s, int size, FILE *stream);
// Pour parcourir des sous-parties de la ligne
#include <string.h>
char *strtok(char *str, const char *delim);
// Pour convertir le texte en nombres
#include <stdlib.h>
long int strtol(const char *nptr, char **endptr, int
↪  base);
double strtod(const char *nptr, char **endptr);
float strtof(const char *nptr, char **endptr);
```

Saisie plus sûre de données (ter)

Méthode 2 (moins sûre, moins générique)

- Faire une saisie avec scanf,
- Vider le buffer d'entrée,
- Vérifier que les informations saisies sont valides,
- Si elles ne le sont pas, redemander la saisie ou lancer un mécanisme d'erreur.

Saisie plus sûre de données (ter)

Méthode 2 (moins sûre, moins générique)

- Faire une saisie avec scanf,
- Vider le buffer d'entrée,
- Vérifier que les informations saisies sont valides,
- Si elles ne le sont pas, redemander la saisie ou lancer un mécanisme d'erreur.

Une fonction pour vider le buffer d'entrée

```
void clear_input_buffer()
{
    while (getchar() != '\n'); // Notez le ';'
}
```

Sommaire

Les expressions

Définition

Toute combinaison de symboles qui représente une valeur est une expression.

Les expressions

Définition

Toute combinaison de symboles qui représente une valeur est une expression.

Quelles sont les expressions dans cet extrait de code ?

```
i = j;
a = f(x);
```

Les expressions

Définition

Toute combinaison de symboles qui représente une valeur est une expression.

Quelles sont les expressions dans cet extrait de code ?

```
i = j;
a = f(x);
```

Solution

- j, i, i = j
- x, f(x), a, a = f(x)

Opérateurs

Définitions

Un opérateur est une fonction ayant un indentificateur spécial : il contient des caractères non autorisés pour les fonctions ordinaires. On appelle **opérande** un paramètre d'un opérateur.

Une liste des opérateurs du C (et du C++) se trouve ici.

Les instructions

Définition

C'est une étape d'un programme informatique. Un programme est constitué d'une suite d'instructions. Il existe différents types d'instructions en C...

Les instructions

Définition

C'est une étape d'un programme informatique. Un programme est constitué d'une suite d'instructions. Il existe différents types d'instructions en C...

L'instruction-expression

C'est le type d'instruction le plus simple. Il s'agit d'une expression suivie d'un point-virgule ' ;' (*semicolon* en anglais).

Exemples d'instructions-expressions

```
a + b + 7;
2 * (a + b);
a = a * b;
b = (b + 7) * (a - b);
```

Les blocs d'instructions

Définition

Un bloc d'instructions est une suite d'instructions entre {}.

Quels sont les blocs du code suivant ?

```
1  #include <stdio.h>
2  int main(void) {
3      int a;
4      scanf("%d", &a);
5      printf("%d ", a);
6      if (a > 0) {
7          while (a != 1) {
8              if (a % 2 == 0)
9                  a /= 2;
10             else
11                 a = a*3+1;
12             printf("%d ", a);
13         }
14     }
15     printf("\n");
16     return 0;
17 }
```

Les blocs d'instructions

Définition

Un bloc d'instructions est une suite d'instructions entre {}.

Quels sont les blocs du code suivant ?

```
1  #include <stdio.h>
2  int main(void) {
3      int a;
4      scanf("%d", &a);
5      printf("%d ", a);
6      if (a > 0) {
7          while (a != 1) {
8              if (a % 2 == 0)
9                  a /= 2;
10             else
11                 a = a*3+1;
12             printf("%d ", a);
13         }
14     }
15     printf("\n");
16     return 0;
17 }
```

Solution (intervalles de lignes)

- 2 (fin) $\rightarrow$ 17
- 6 (fin) $\rightarrow$ 14
- 7 (fin) $\rightarrow$ 13

Les opérateurs arithmétiques

Les opérateurs binaires

Opérateur	Nom	Exemple
+	Addition	2 + 2
-	Soustraction	a - 2
*	Multiplication	a * b
/	Division	a / 7
%	Modulo	42 % 2

Les opérateurs unaires

Opérateur	Nom	Exemple
+	Plus	+a
-	Moins	-a

Division et modulo

Division entière

Si on divise deux entiers, une division entière (euclidienne) est effectuée :

- a/b renvoie le quotient
- a%b renvoie le reste

Le modulo n'est défini qu'entre deux entiers.

Division au sens habituel

Si on divise deux flottants, ou un entier et un flottant, une division au sens habituel est effectuée.

```
int a = 1 / 2;         // 0
int r = 1 % 2;         // 1
float b = 1.0 / 2;     // 0.5
float c = 1.0 / 2.0;   // 0.5
float d = 1 / 2.0;     // 0.5
```

Post/pré incr/décrementations

```
i++;
```

```
i = i + 1;
```

```
++j;
```

```
j = j + 1;
```

```
a = b++;
```

```
a = b;
b = b + 1;
```

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

```
y = y + 1;
x = x;
```

```
z = --z * z++; // warning
```

```
z = z - 1;
z = z * z;
// z = z + 1; n'est pas fait !
```

```
;
```

Les opérateurs d'affectation

Il existe des extensions de l'opérateur d'affectation :

Opérateur	Nom	Exemple
+=	Addition	a += 2 ;
-=	Soustraction	a -= b ;
*=	Multiplication	a *= a ;
/=	Division	a /= 2 ;
%=	Modulo	a %= 2 ;
...		

Équivalences

```
a += 2; // a = a + 2
a -= b; // a = a - b
a *= a; // a = a * a;
a /= 2; // a = a / 2;
a %= 2; // a = a % 2;
```

Les expressions booléennes

Définition

Les **expressions booléennes** sont des expressions qui représentent une valeur de vérité : **vrai** ou **faux**.

Les expressions booléennes

Définition

Les **expressions booléennes** sont des expressions qui représentent une valeur de vérité : **vrai** ou **faux**.

Vrai et faux en C

En C, l'entier zéro est considéré comme étant faux. Toutes les autres valeurs entières sont considérées comme étant vraies.

Les expressions booléennes

Définition

Les **expressions booléennes** sont des expressions qui représentent une valeur de vérité : **vrai** ou **faux**.

Vrai et faux en C

En C, l'entier zéro est considéré comme étant faux. Toutes les autres valeurs entières sont considérées comme étant vraies.
Depuis la norme C99, il est possible d'utiliser les constantes **true** et **false** et le type **bool** si on inclut le fichier **stdbool.h**.

Les expressions booléennes

Définition

Les **expressions booléennes** sont des expressions qui représentent une valeur de vérité : **vrai** ou **faux**.

Vrai et faux en C

En C, l'entier zéro est considéré comme étant faux. Toutes les autres valeurs entières sont considérées comme étant vraies.
Depuis la norme C99, il est possible d'utiliser les constantes **true** et **false** et le type **bool** si on inclut le fichier **stdbool.h**.

Opérateurs de comparaison et opérateurs logiques

Lorsqu'on s'interroge sur la véracité d'une propriété, l'emploi d'**opérateurs de comparaison** et d'**opérateurs logiques** est souvent approprié.

Les opérateurs de comparaison

Ces opérateurs permettent de comparer deux expressions et renvoient un booléen : soit vrai (**1**), soit faux (**0**).

Nom	Opérateur	Exemple
Égalité	==	a == 2
Différence	!=	a != b
Infériorité	<=	a <= 10
Supériorité	>=	a >= 0
Infériorité stricte	<	10 < a
Supériorité stricte	>	a < 20

Les opérateurs de comparaison : exemple

```c
#include <stdio.h>
#include <stdbool.h> // bool, true et false
int main(void)
{
    int i = 10, j = 4;
    bool egal = (i == j);     // false
    bool diff = (i != j);     // true
    bool inf = (i < j);       // false
    bool infEgal = (i <= j); // false
    bool sup = (i > j);       // true
    bool supEgal = (i >= j); // true
    return 0;
}
```

Les opérateurs logiques

Mmh ?

Ces opérateurs permettent de combiner deux valeurs booléennes selon des règles de négation, de conjonction ou de disjonction logique.

Nom	Opérateur	Alternative [1]	Exemple
Négation	!	not	!a
Conjonction	&&	and	a && b
Disjonction	||	or	a || b

1. Depuis la norme C90, des équivalents anglais aux symboles peuvent être utilisés à condition d'inclure le fichier **iso646.h**.

La négation logique (**non**)

Elle permet d'*inverser* la valeur de vérité de son paramètre. En langue naturelle, on la nomme **non**. En mathématiques et en logique, la négation de a est notée $\neg a$. En ingénierie, la négation de a est souvent notée $\overline{a}$. Elle correspond au complémentaire de la théorie des ensembles.

Entrée	Sortie
a	$\neg a$
false	true
true	false

Table de vérité

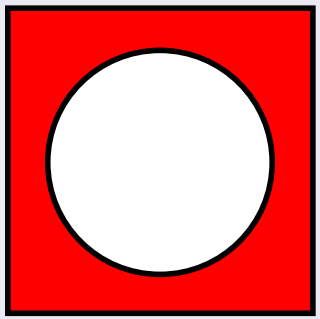


Diagramme de Venn

La conjonction logique (**et**)

Elle permet de répondre à la question : est-ce que mes paramètres sont tous vrais ? En langue naturelle, on la nomme **et**. En mathématiques et en logique, elle est notée $\wedge$. En ingénierie, on la note souvent $\cdot$. Elle correspond à l'intersection $\bigcap$ de la théorie des ensembles.

Entrée		Sortie
a	b	$a \wedge b$
false	false	false
false	true	false
true	false	false
true	true	true

Table de vérité

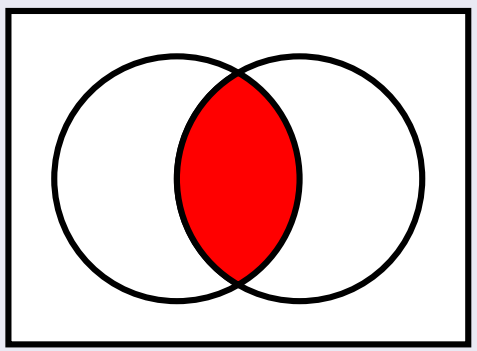


Diagramme de Venn

La disjonction logique (**ou**)

Elle permet de répondre à la question : est-ce qu'au moins un de mes paramètres est vrai ? En langue naturelle, on la nomme **ou (inclusif)**. En mathématiques et en logique, elle est notée $\vee$. En ingénierie, on la note souvent $+$. Elle correspond à l'union $\bigcup$ de la théorie des ensembles.

Entrée		Sortie
a	b	$a \vee b$
false	false	false
false	true	true
true	false	true
true	true	true

Table de vérité

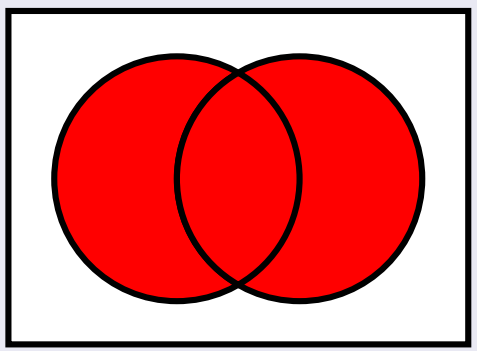


Diagramme de Venn

Les opérateurs logiques : exemple

```c
#include <stdio.h>
#include <stdbool.h> // bool, true et false
int main(void)
{
    bool a = true, b = false;
    bool not_a = !a;        // false
    bool not_b = !b;        // true
    bool a_and_b = a && b; // false
    bool a_or_b = a || b;  // true
    bool compound = !b && (a || b);
    return 0;
}
```

Remarques sur la conjonction (**et**)

- Lorsqu'on calcule la conjonction (**et**) d'un ensemble P de prédicats, dès que l'on rencontre un prédicat $p \in P$ faux, on sait que la conjonction est fausse.
- Il n'est donc pas nécessaire d'évaluer les autres prédicats.
- En C, les prédicats sont évalués de gauche à droite.
- **Si l'opérande gauche d'une conjonction est fausse, l'opérande droite n'est pas du tout évaluée.**

```
cond1 && cond2;
```

Opérateurs logiques et ordre d'exécution

Remarque sur la disjonction (**ou**)

- Lorsqu'on calcule la disjonction (**ou**) d'un ensemble P de prédicats, dès que l'on rencontre un prédicat $p \in P$ vrai, on sait que la disjonction est vraie.
- Il n'est donc pas nécessaire d'évaluer les autres prédicats.
- En C, les prédicats sont évalués de gauche à droite.
- **Si l'opérande gauche d'une disjonction est vraie, l'opérande droite n'est pas du tout évaluée.**

```
cond1 || cond2;
```

Les opérateurs bit à bit

Ces opérateurs effectuent les opérations bit par bit sur leurs opérandes.

Opérateur	Nom	Exemple
~	Bitwise NOT	~a
&	Bitwise AND	a & b
|	Bitwise OR	a | 3
^	Bitwise XOR	a ^2
«	Bitwise left shift	a «1
»	Bitwise right shift	a »2

Attention

Ces opérateurs ne sont pas équivalents aux opérateurs logiques ! Cependant, si on contraint leurs opérandes à être soit 0 soit 1, leur comportement est le même.

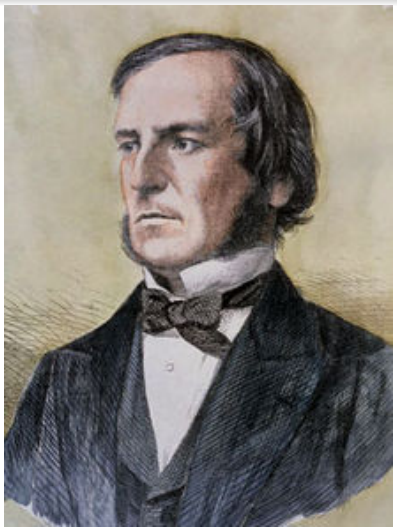
Les lois de De Morgan

¿Qué ?

Ces lois sont des identités logiques formulées par le mathématicien britannique Augustus De Morgan au xix^{e} siècle (un des fondateurs, avec George Boole, de la logique moderne). Elles peuvent permettre (entre autres) de développer la négation d'expressions logiques :

- $(\neg(A \wedge B)) \Leftrightarrow ((\neg A) \vee (\neg B))$,
- $(\neg(A \vee B)) \Leftrightarrow ((\neg A) \wedge (\neg B))$.

Augustus de Morgan



George Boole

Les lois de De Morgan : preuve par table de vérité

Proposition

Deux formules logiques sont équivalentes si et seulement si elles ont la même table de vérité.

Entrée		Calcul	Formule 1	Calcul		Formule 2
A	B	$A \wedge B$	$\neg(A \wedge B)$	$\neg A$	$\neg B$	$(\neg A) \vee (\neg B)$
false	false					
false	true					
true	false					
true	true					

Table de vérité de la première loi de De Morgan $(\neg(A \wedge B)) \Leftrightarrow ((\neg A) \vee (\neg B))$.

Les lois de De Morgan : preuve par table de vérité

Proposition

Deux formules logiques sont équivalentes si et seulement si elles ont la même table de vérité.

Entrée		Calcul	Formule 1	Calcul		Formule 2
A	B	$A \wedge B$	$\neg(A \wedge B)$	$\neg A$	$\neg B$	$(\neg A) \vee (\neg B)$
false	false	false				
false	true	false				
true	false	false				
true	true	true				

Table de vérité de la première loi de De Morgan $(\neg(A \wedge B)) \Leftrightarrow ((\neg A) \vee (\neg B))$.

Les lois de De Morgan : preuve par table de vérité

Proposition

Deux formules logiques sont équivalentes si et seulement si elles ont la même table de vérité.

Entrée		Calcul	Formule 1	Calcul		Formule 2
A	B	$A \wedge B$	$\neg(A \wedge B)$	$\neg A$	$\neg B$	$(\neg A) \vee (\neg B)$
false	false	false	**true**			
false	true	false	**true**			
true	false	false	**true**			
true	true	true	**false**			

Table de vérité de la première loi de De Morgan $(\neg(A \wedge B)) \Leftrightarrow ((\neg A) \vee (\neg B))$.

Les lois de De Morgan : preuve par table de vérité

Proposition

Deux formules logiques sont équivalentes si et seulement si elles ont la même table de vérité.

Entrée		Calcul	Formule 1	Calcul		Formule 2
A	B	$A \wedge B$	$\neg(A \wedge B)$	$\neg A$	$\neg B$	$(\neg A) \vee (\neg B)$
false	false	false	**true**	true		
false	true	false	**true**	true		
true	false	false	**true**	false		
true	true	true	**false**	false		

Table de vérité de la première loi de De Morgan $(\neg(A \wedge B)) \Leftrightarrow ((\neg A) \vee (\neg B))$.

Les lois de De Morgan : preuve par table de vérité

Proposition

Deux formules logiques sont équivalentes si et seulement si elles ont la même table de vérité.

Entrée		Calcul	Formule 1	Calcul		Formule 2
A	B	$A \wedge B$	$\neg(A \wedge B)$	$\neg A$	$\neg B$	$(\neg A) \vee (\neg B)$
false	false	false	**true**	true	true	
false	true	false	**true**	true	false	
true	false	false	**true**	false	true	
true	true	true	**false**	false	false	

Table de vérité de la première loi de De Morgan $(\neg(A \wedge B)) \Leftrightarrow ((\neg A) \vee (\neg B))$.

Les lois de De Morgan : preuve par table de vérité

Proposition

Deux formules logiques sont équivalentes si et seulement si elles ont la même table de vérité.

Entrée		Calcul	Formule 1	Calcul		Formule 2
A	B	$A \wedge B$	$\neg(A \wedge B)$	$\neg A$	$\neg B$	$(\neg A) \vee (\neg B)$
false	false	false	**true**	true	true	**true**
false	true	false	**true**	true	false	**true**
true	false	false	**true**	false	true	**true**
true	true	true	**false**	false	false	**false**

Table de vérité de la première loi de De Morgan $(\neg(A \wedge B)) \Leftrightarrow ((\neg A) \vee (\neg B))$.

Les lois de De Morgan : preuve par table de vérité

Proposition

Deux formules logiques sont équivalentes si et seulement si elles ont la même table de vérité.

Entrée		Calcul	Formule 1	Calcul		Formule 2
A	B	$A \vee B$	$\neg(A \vee B)$	$\neg A$	$\neg B$	$(\neg A) \wedge (\neg B)$
false	false					
false	true					
true	false					
true	true					

Table de vérité de la deuxième loi de De Morgan $(\neg(A \vee B)) \Leftrightarrow ((\neg A) \wedge (\neg B))$.

Les lois de De Morgan : preuve par table de vérité

Proposition

Deux formules logiques sont équivalentes si et seulement si elles ont la même table de vérité.

Entrée		Calcul	Formule 1	Calcul		Formule 2
A	B	$A \vee B$	$\neg(A \vee B)$	$\neg A$	$\neg B$	$(\neg A) \wedge (\neg B)$
false	false	false				
false	true	true				
true	false	true				
true	true	true				

Table de vérité de la deuxième loi de De Morgan $(\neg(A \vee B)) \Leftrightarrow ((\neg A) \wedge (\neg B))$.

Les lois de De Morgan : preuve par table de vérité

Proposition

Deux formules logiques sont équivalentes si et seulement si elles ont la même table de vérité.

Entrée		Calcul	Formule 1	Calcul		Formule 2
A	B	$A \vee B$	$\neg(A \vee B)$	$\neg A$	$\neg B$	$(\neg A) \wedge (\neg B)$
false	false	false	**true**			
false	true	true	**false**			
true	false	true	**false**			
true	true	true	**false**			

Table de vérité de la deuxième loi de De Morgan $(\neg(A \vee B)) \Leftrightarrow ((\neg A) \wedge (\neg B))$.

Les lois de De Morgan : preuve par table de vérité

Proposition

Deux formules logiques sont équivalentes si et seulement si elles ont la même table de vérité.

Entrée		Calcul	Formule 1	Calcul		Formule 2
A	B	$A \vee B$	$\neg(A \vee B)$	$\neg A$	$\neg B$	$(\neg A) \wedge (\neg B)$
false	false	false	**true**	true		
false	true	true	**false**	true		
true	false	true	**false**	false		
true	true	true	**false**	false		

Table de vérité de la deuxième loi de De Morgan $(\neg(A \vee B)) \Leftrightarrow ((\neg A) \wedge (\neg B))$.

Les lois de De Morgan : preuve par table de vérité

Proposition

Deux formules logiques sont équivalentes si et seulement si elles ont la même table de vérité.

Entrée		Calcul	Formule 1	Calcul		Formule 2
A	B	$A \vee B$	$\neg(A \vee B)$	$\neg A$	$\neg B$	$(\neg A) \wedge (\neg B)$
false	false	false	**true**	true	true	
false	true	true	**false**	true	false	
true	false	true	**false**	false	true	
true	true	true	**false**	false	false	

Table de vérité de la deuxième loi de De Morgan $(\neg(A \vee B)) \Leftrightarrow ((\neg A) \wedge (\neg B))$.

Les lois de De Morgan : preuve par table de vérité

Proposition

Deux formules logiques sont équivalentes si et seulement si elles ont la même table de vérité.

Entrée		Calcul	Formule 1	Calcul		Formule 2
A	B	$A \vee B$	$\neg(A \vee B)$	$\neg A$	$\neg B$	$(\neg A) \wedge (\neg B)$
false	false	false	**true**	true	true	**true**
false	true	true	**false**	true	false	**false**
true	false	true	**false**	false	true	**false**
true	true	true	**false**	false	false	**false**

Table de vérité de la deuxième loi de De Morgan $(\neg(A \vee B)) \Leftrightarrow ((\neg A) \wedge (\neg B))$.

La priorité des opérateurs

La priorité des opérateurs permet de savoir dans quel ordre les opérations doivent être effectuées.

Les opérations arithmétiques suivent l'ordre habituel de l'arithmétique.

La priorité des autres opérations est assez logique pour ne pas être remarquée dans la plupart des cas. **Dans le doute, utilisez des parenthèses !**

La priorité des opérateurs : exemple

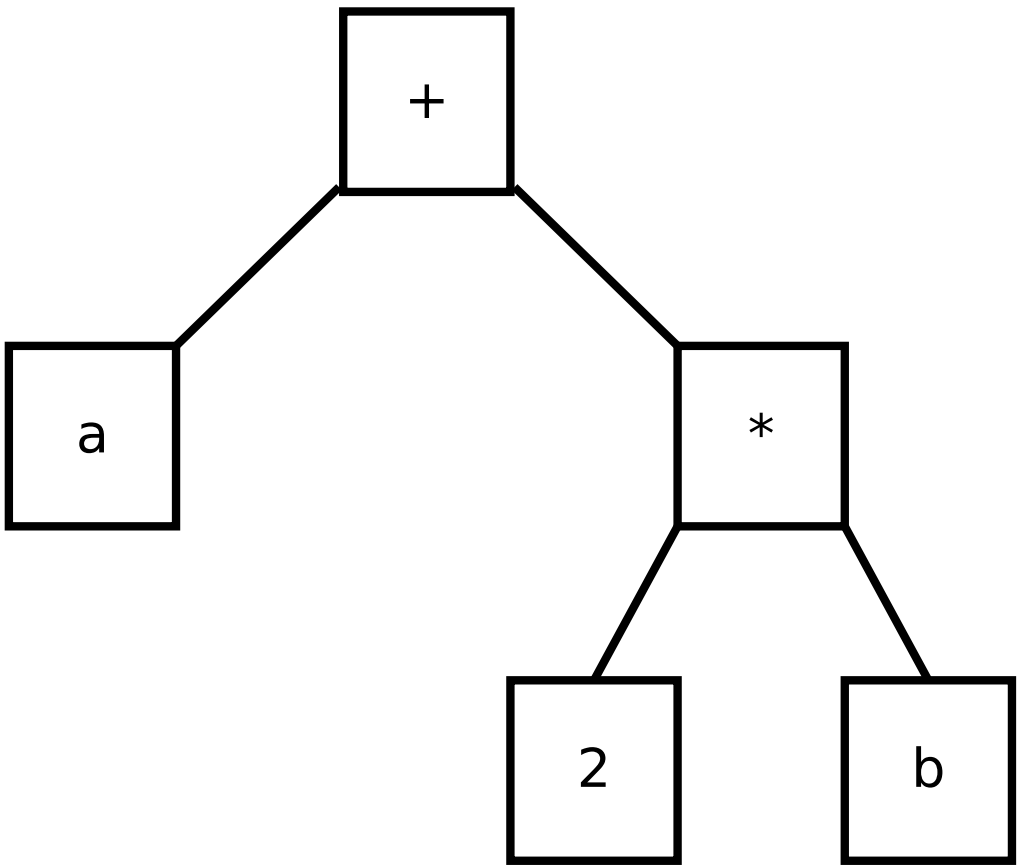
Comment évaluer les expressions suivantes ?

```
a + 2 * b
a = 42 * 57 + 3
1 / 2 * 3 / 6
```

La priorité des opérateurs : exemple

Comment évaluer les expressions suivantes ?

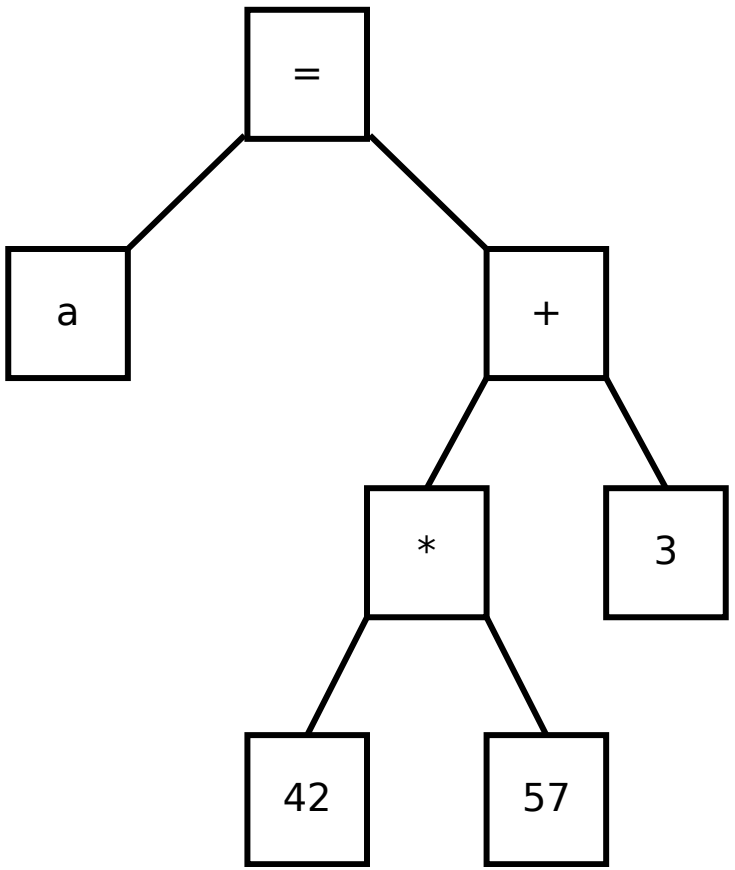
```
a + 2 * b
a = 42 * 57 + 3
1 / 2 * 3 / 6
```



La priorité des opérateurs : exemple

Comment évaluer les expressions suivantes ?

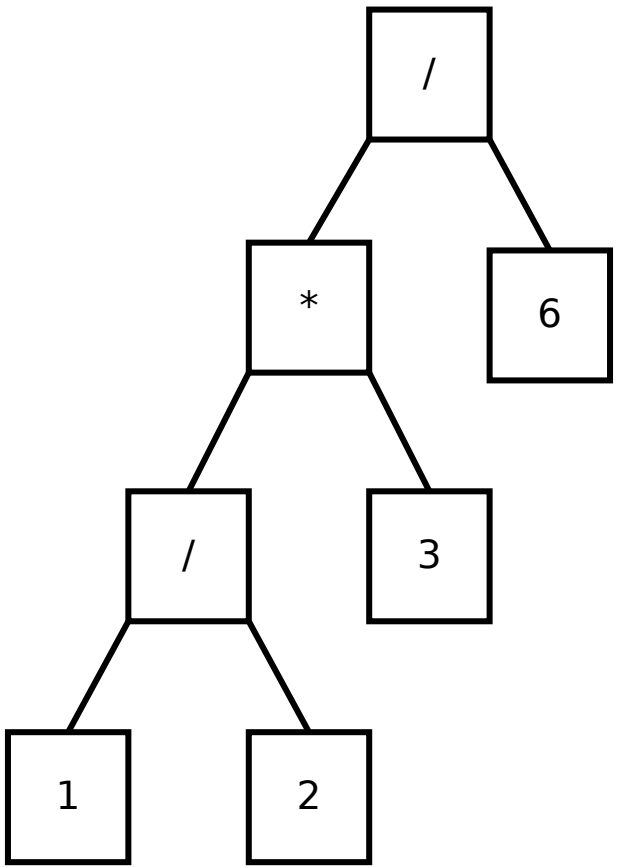
```
a + 2 * b
a = 42 * 57 + 3
1 / 2 * 3 / 6
```



La priorité des opérateurs : exemple

Comment évaluer les expressions suivantes ?

```
a + 2 * b
a = 42 * 57 + 3
1 / 2 * 3 / 6
```



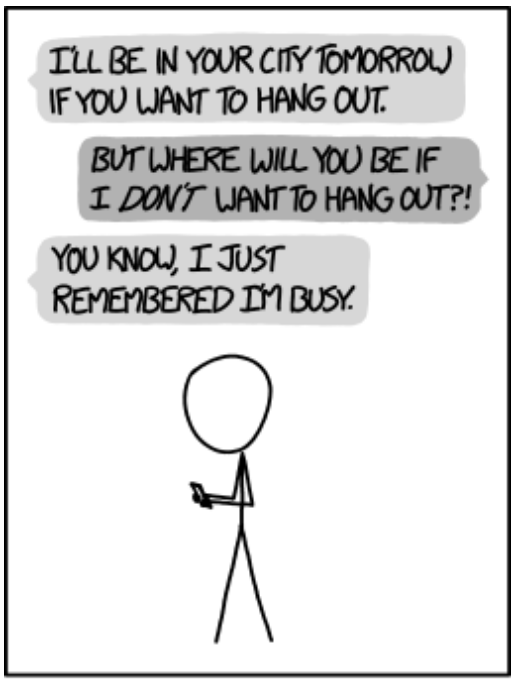
Sommaire

Les conditions

Les **structures conditionnelles** ou plus simplement conditions permettent à vos programmes de ne pas toujours faire la même chose.

En fonction du résultat d'un certain *test*, le programme va exécuter tel *bloc d'instructions* ou tel autre. Les structures conditionnelles sont des instructions. Plusieurs structures conditionnelles sont présentes en C :

- Le if, [[else if, [else if ...]], else].
- Le switch.



WHY I TRY NOT TO BE PEDANTIC ABOUT CONDITIONALS.

If, else if, else

```
if (<expr1>)
    <bloc1>
else if (<expr2>)
    <bloc2>
else if (<expr3>)
    <bloc3>
.
.
.
else if (<exprN>)
    <blocN>
else
    <blockElse>
```

- <bloc> est soit un bloc d'instructions, soit une seule instruction
- On doit mettre 1 *if*. On peut mettre entre 0 et plusieurs *else if*, 0 ou 1 *else*. Le *if* doit **toujours** être placé en premier et le *else* **toujours** en dernier.
- Une structure *if, else if, ..., else* est exécutée de haut en bas.
- Si une expression à l'intérieur des parenthèses d'un *if* ou d'un *else if* est évaluée comme étant vraie, le bloc suivant est exécuté puis on quitte la structure conditionnelle.
- Si l'expression n'est pas évaluée comme étant vraie, on passe à la suivante.

Conditions : exemple

```c
#include <stdio.h>
int main(void)
{
    char genre;
    printf("Êtes-vous un homme (H) ou une femme (F) ? ");
    scanf("%c", &genre);
    printf("Vous avez entré le caractère %c\n", genre);
    if (genre == 'H')
        printf("Vous êtes un homme\n");
    else if (genre == 'F')
        printf("Vous êtes une femme\n");
    else
        printf("Vous n'êtes ni un homme ni une femme\n");
    return 0;
}
```

Le switch

Utilisation

Le **switch** est une instruction qui permet de faire plusieurs tests d'égalité sur la valeur d'une seule expression.
Il permet de simplifier l'écriture de certains codes qui seraient très lourds avec une structure **if, else if... else**.

Limitation

Le switch est plus limité qu'une structure en **if, else if... else** :

- On ne peut faire que des tests d'**égalité**,
- Ces tests se font tous sur **la même expression**,
- Cette expression doit être **entière** (ou assimilable à un entier),
- On ne peut comparer l'expression qu'avec des expressions **constantes**.

Le switch : exemple

```c
#include <stdio.h>
int main() {
    char note = 'A';
    switch(note) {
    case 'A':
        printf("Excellente note\n");
        break;
    case 'B' :
        printf("Bonne note\n");
        break;
    case 'C' :
        printf("Note moyenne\n");
        break;
    case 'D' :
        printf("Mauvaise note\n");
        break;
    case 'E' :
        printf("Très mauvaise note\n");
        break;
    default:
        printf("Note inconnue\n");
        break;
    }
    return 0;
}
```

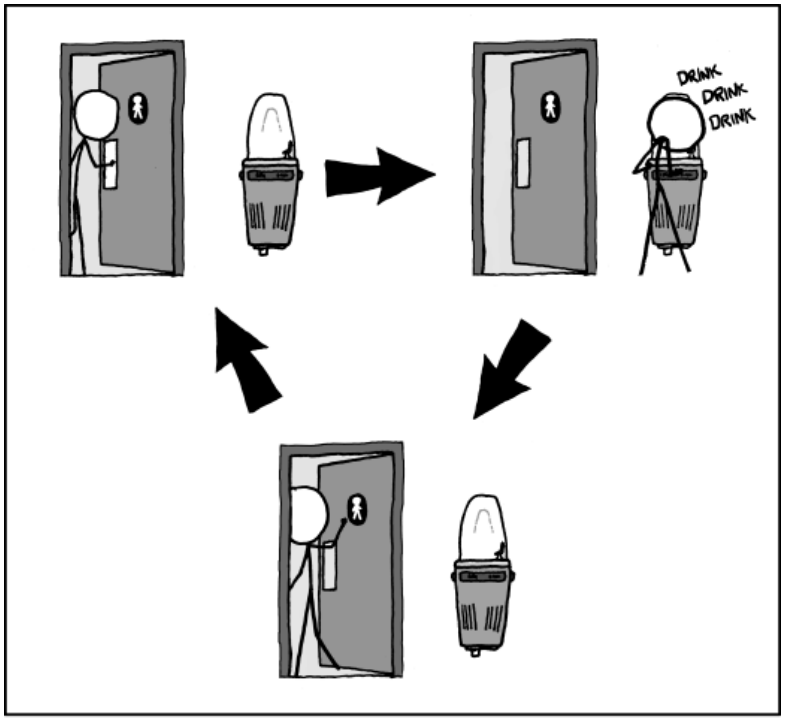
Sommaire

Les boucles

Les boucles sont des instructions qui permettent de répéter l'exécution d'un bloc d'instructions en fonction d'une certaine condition.

En C, plusieurs types de boucles sont à notre disposition :

- La boucle **while**,
- La boucle **do... while**,
- La boucle **for**.



I AVOID DRINKING FOUNTAINS OUTSIDE BATHROOMS BECAUSE I'M AFRAID OF GETTING TRAPPED IN A LOOP.

La boucle while

```
while (<expr>)
    <bloc>
```

- <bloc> est soit un bloc d'instructions, soit une seule instruction.
- Tant que <expr> est évaluée comme étant vraie, on exécute le bloc d'instructions <bloc>.
- On évalue <expr> **avant** d'exécuter <bloc>.

La boucle while : exemple

```c
#include <stdio.h>
int main(void)
{
    int i = 0;
    while (i < 3)
    {
        printf("%d\n", i);
        i++;
    }
    return 0;
}
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

La boucle while : exemple

```
1  #include <stdio.h>
2  int main(void)
3  {
4      int i = 0;
5      while (i < 3)
6      {
7          printf("%d\n", i);
8          i++;
9      }
10     return 0;
11 }
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

Simulation de l'exécution du code

Ligne	i	affichage	test

La boucle while : exemple

```
1  #include <stdio.h>
2  int main(void)
3  {
4      int i = 0;
5      while (i < 3)
6      {
7          printf("%d\n", i);
8          i++;
9      }
10     return 0;
11 }
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

Simulation de l'exécution du code

Ligne	i	affichage	test
4	0		

La boucle while : exemple

```
1   #include <stdio.h>
2   int main(void)
3   {
4       int i = 0;
5       while (i < 3)
6       {
7           printf("%d\n", i);
8           i++;
9       }
10      return 0;
11  }
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

Simulation de l'exécution du code

Ligne	i	affichage	test
4	0		
5	0		vrai

La boucle while : exemple

```c
#include <stdio.h>
int main(void)
{
    int i = 0;
    while (i < 3)
    {
        printf("%d\n", i);
        i++;
    }
    return 0;
}
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

Simulation de l'exécution du code

Ligne	i	affichage	test
4	0		
5 7	0 0	0	vrai

La boucle while : exemple

```
1  #include <stdio.h>
2  int main(void)
3  {
4      int i = 0;
5      while (i < 3)
6      {
7          printf("%d\n", i);
8          i++;
9      }
10     return 0;
11 }
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

Simulation de l'exécution du code

Ligne	i	affichage	test
4	0		
5 7 8	0 0 1	0	vrai

La boucle while : exemple

```
1   #include <stdio.h>
2   int main(void)
3   {
4       int i = 0;
5       while (i < 3)
6       {
7           printf("%d\n", i);
8           i++;
9       }
10      return 0;
11  }
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

Simulation de l'exécution du code

Ligne	i	affichage	test
4	0		
5	0		vrai
7	0	0	
8	1		
5	1		vrai

La boucle while : exemple

```
1   #include <stdio.h>
2   int main(void)
3   {
4       int i = 0;
5       while (i < 3)
6       {
7           printf("%d\n", i);
8           i++;
9       }
10      return 0;
11  }
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

Simulation de l'exécution du code

Ligne	i	affichage	test
4	0		
5 7 8	0 0 1	0	vrai
5 7	1 1	1	vrai

La boucle while : exemple

```
1   #include <stdio.h>
2   int main(void)
3   {
4       int i = 0;
5       while (i < 3)
6       {
7           printf("%d\n", i);
8           i++;
9       }
10      return 0;
11  }
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

Simulation de l'exécution du code

Ligne	i	affichage	test
4	0		
5 7 8	0 0 1	0	vrai
5 7 8	1 1 2	1	vrai

La boucle while : exemple

```
1   #include <stdio.h>
2   int main(void)
3   {
4       int i = 0;
5       while (i < 3)
6       {
7           printf("%d\n", i);
8           i++;
9       }
10      return 0;
11  }
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

Simulation de l'exécution du code

Ligne	i	affichage	test
4	0		
5 7 8	0 0 1	0	vrai
5 7 8	1 1 2	1	vrai
5	2		vrai

La boucle while : exemple

```
1   #include <stdio.h>
2   int main(void)
3   {
4       int i = 0;
5       while (i < 3)
6       {
7           printf("%d\n", i);
8           i++;
9       }
10      return 0;
11  }
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

Simulation de l'exécution du code

Ligne	i	affichage	test
4	0		
5 7 8	0 0 1	0	vrai
5 7 8	1 1 2	1	vrai
5 7	2 2	2	vrai

La boucle while : exemple

```
1   #include <stdio.h>
2   int main(void)
3   {
4       int i = 0;
5       while (i < 3)
6       {
7           printf("%d\n", i);
8           i++;
9       }
10      return 0;
11  }
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

Simulation de l'exécution du code

Ligne	i	affichage	test
4	0		
5	0		vrai
7	0	0	
8	1		
5	1		vrai
7	1	1	
8	2		
5	2		vrai
7	2	2	
8	3		

La boucle while : exemple

```
1   #include <stdio.h>
2   int main(void)
3   {
4       int i = 0;
5       while (i < 3)
6       {
7           printf("%d\n", i);
8           i++;
9       }
10      return 0;
11  }
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

Simulation de l'exécution du code

Ligne	i	affichage	test
4	0		
5 7 8	0 0 1	0	vrai
5 7 8	1 1 2	1	vrai
5 7 8	2 2 3	2	vrai
5	3		faux

La boucle while : exemple

```
1   #include <stdio.h>
2   int main(void)
3   {
4       int i = 0;
5       while (i < 3)
6       {
7           printf("%d\n", i);
8           i++;
9       }
10      return 0;
11  }
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

Simulation de l'exécution du code

Ligne	i	affichage	test
4	0		
5 7 8	0 0 1	0	vrai
5 7 8	1 1 2	1	vrai
5 7 8	2 2 3	2	vrai
5	3		faux
10	3		

La boucle do...while

```
do
    <bloc>
while (<expr>);
```

- <bloc> est soit un bloc d'instructions, soit une seule instruction.
- Tant que <expr> est évaluée comme étant vraie, on exécute le bloc d'instructions <bloc>.
- Identique au while ? Non !
- On évalue <expr> **après** avoir exécuté <bloc>.

Attention !

Cette instruction finit par un point-virgule !

Différence while do...while

```c
#include <stdio.h>
#include <stdbool.h>
int main(void)
{
    bool a = true, b = true;
    while (a != b)
    {
        printf("%d\n", a);
        a = !a;
    }
    return 0;
}
```

```c
#include <stdio.h>
#include <stdbool.h>
int main(void)
{
    bool a = true, b = true;
    do
    {
        printf("%d\n", a);
        a = !a;
    } while (a != b);
    return 0;
}
```

Exercice

Simulez l'exécution de ces deux codes. Qu'affichent-ils ?

Différence while do...while : exécution du code while

```c
#include <stdio.h>
#include <stdbool.h>
int main(void)
{
    bool a = true, b = true;
    while (a != b)
    {
        printf("%d\n", a);
        a = !a;
    }
    return 0;
}
```

Différence while do...while : exécution du code while

```c
#include <stdio.h>
#include <stdbool.h>
int main(void)
{
    bool a = true, b = true;
    while (a != b)
    {
        printf("%d\n", a);
        a = !a;
    }
    return 0;
}
```

Ligne	a	b	sortie	test

Différence while do...while : exécution du code while

```
1   #include <stdio.h>
2   #include <stdbool.h>
3   int main(void)
4   {
5       bool a = true, b = true;
6       while (a != b)
7       {
8           printf("%d\n", a);
9           a = !a;
10      }
11      return 0;
12  }
```

Ligne	a	b	sortie	test
5	true	true		

Différence while do...while : exécution du code while

```
1   #include <stdio.h>
2   #include <stdbool.h>
3   int main(void)
4   {
5       bool a = true, b = true;
6       while (a != b)
7       {
8           printf("%d\n", a);
9           a = !a;
10      }
11      return 0;
12  }
```

Ligne	a	b	sortie	test
5	true	true		
6	true	true		false

Différence while do...while : exécution du code while

```c
#include <stdio.h>
#include <stdbool.h>
int main(void)
{
    bool a = true, b = true;
    while (a != b)
    {
        printf("%d\n", a);
        a = !a;
    }
    return 0;
}
```

Ligne	a	b	sortie	test
5	true	true		
6	true	true		false
11	true	true		

Différence while do...while : exécution du code do...while

```c
#include <stdio.h>
#include <stdbool.h>
int main(void)
{
    bool a = true, b = true;
    do
    {
        printf("%d\n", a);
        a = !a;
    } while (a != b);
    return 0;
}
```

Différence while do...while : exécution du code do...while

```c
#include <stdio.h>
#include <stdbool.h>
int main(void)
{
    bool a = true, b = true;
    do
    {
        printf("%d\n", a);
        a = !a;
    } while (a != b);
    return 0;
}
```

Ligne	a	b	sortie	test

Différence while do...while : exécution du code do...while

```
1  #include <stdio.h>
2  #include <stdbool.h>
3  int main(void)
4  {
5      bool a = true, b = true;
6      do
7      {
8          printf("%d\n", a);
9          a = !a;
10     } while (a != b);
11     return 0;
12 }
```

Ligne	a	b	sortie	test
5	true	true		

Différence while do...while : exécution du code do...while

```
1  #include <stdio.h>
2  #include <stdbool.h>
3  int main(void)
4  {
5      bool a = true, b = true;
6      do
7      {
8          printf("%d\n", a);
9          a = !a;
10     } while (a != b);
11     return 0;
12 }
```

Ligne	a	b	sortie	test
5	true	true		
8	true	true	1	

Différence while do...while : exécution du code do...while

```
1  #include <stdio.h>
2  #include <stdbool.h>
3  int main(void)
4  {
5      bool a = true, b = true;
6      do
7      {
8          printf("%d\n", a);
9          a = !a;
10     } while (a != b);
11     return 0;
12 }
```

Ligne	a	b	sortie	test
5	true	true		
8	true	true	1	
9	false	true		

Différence while do...while : exécution du code do...while

```
1  #include <stdio.h>
2  #include <stdbool.h>
3  int main(void)
4  {
5      bool a = true, b = true;
6      do
7      {
8          printf("%d\n", a);
9          a = !a;
10     } while (a != b);
11     return 0;
12 }
```

Ligne	a	b	sortie	test
5	true	true		
8	true	true	1	
9	false	true		
10	false	true		true

Différence while do...while : exécution du code do...while

```
1   #include <stdio.h>
2   #include <stdbool.h>
3   int main(void)
4   {
5       bool a = true, b = true;
6       do
7       {
8           printf("%d\n", a);
9           a = !a;
10      } while (a != b);
11      return 0;
12  }
```

Ligne	a	b	sortie	test
5	true	true		
8	true	true	1	
9	false	true		
10	false	true		true
8	false	true	0	

Différence while do...while : exécution du code do...while

```
1   #include <stdio.h>
2   #include <stdbool.h>
3   int main(void)
4   {
5       bool a = true, b = true;
6       do
7       {
8           printf("%d\n", a);
9           a = !a;
10      } while (a != b);
11      return 0;
12  }
```

Ligne	a	b	sortie	test
5	true	true		
8	true	true	1	
9	false	true		
10	false	true		true
8	false	true	0	
9	true	true		

Différence while do...while : exécution du code do...while

```
1   #include <stdio.h>
2   #include <stdbool.h>
3   int main(void)
4   {
5       bool a = true, b = true;
6       do
7       {
8           printf("%d\n", a);
9           a = !a;
10      } while (a != b);
11      return 0;
12  }
```

Ligne	a	b	sortie	test
5	true	true		
8	true	true	1	
9	false	true		
10	false	true		true
8	false	true	0	
9	true	true		
10	false	true		false

Différence while do...while : exécution du code do...while

```c
#include <stdio.h>
#include <stdbool.h>
int main(void)
{
    bool a = true, b = true;
    do
    {
        printf("%d\n", a);
        a = !a;
    } while (a != b);
    return 0;
}
```

Ligne	a	b	sortie	test
5	true	true		
8	true	true	1	
9	false	true		
10	false	true		true
8	false	true	0	
9	true	true		
10	false	true		false
11	false	true		

La boucle for

```
for (<exprInit> ; <exprTest> ; <exprEnd>)
    <bloc>
```

- <bloc> est soit un bloc d'instructions, soit une seule instruction.
- <exprInit> est exécutée **uniquement** la première fois que l'on rencontre le *for*. C'est la première chose qui est faite lorsque le programme atteint le *for*.
- Tant que <exprTest> est évaluée comme étant vraie, <bloc> est exécuté. Dès que <exprTest> est évaluée comme étant fausse, on quitte directement la boucle.
- <exprEnd> est exécuté après la fin de <bloc> à chaque tour de boucle. Note : à la fin d'un tour de boucle, l'exécution de <exprEnd> est réalisée avant la nouvelle évaluation de <exprTest>.

La boucle for : exemple

```c
#include <stdio.h>
int main(void)
{
    for (int i = 0; i < 3; i++)
        printf("%d\n", i);
    return 0;
}
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

La boucle for : exemple

```
1  #include <stdio.h>
2  int main(void)
3  {
4      for (int i = 0; i < 3; i++)
5          printf("%d\n", i);
6      return 0;
7  }
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

Ligne	i	affichage	test

La boucle for : exemple

```
1  #include <stdio.h>
2  int main(void)
3  {
4      for (int i = 0; i < 3; i++)
5          printf("%d\n", i);
6      return 0;
7  }
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

Ligne	i	affichage	test
4 (1)	0		

La boucle for : exemple

```
1  #include <stdio.h>
2  int main(void)
3  {
4      for (int i = 0; i < 3; i++)
5          printf("%d\n", i);
6      return 0;
7  }
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

Ligne	i	affichage	test
4 (1) 4 (2)	0 0		vrai

La boucle for : exemple

```
1  #include <stdio.h>
2  int main(void)
3  {
4      for (int i = 0; i < 3; i++)
5          printf("%d\n", i);
6      return 0;
7  }
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

Ligne	i	affichage	test
4 (1) 4 (2) 5	0 0 0	 0	 vrai

La boucle for : exemple

```
1  #include <stdio.h>
2  int main(void)
3  {
4      for (int i = 0; i < 3; i++)
5          printf("%d\n", i);
6      return 0;
7  }
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

Ligne	i	affichage	test
4 (1)	0		
4 (2)	0		vrai
5	0	0	
4 (3)	1		

La boucle for : exemple

```
1  #include <stdio.h>
2  int main(void)
3  {
4      for (int i = 0; i < 3; i++)
5          printf("%d\n", i);
6      return 0;
7  }
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

Ligne	i	affichage	test
4 (1) 4 (2) 5 4 (3)	0 0 0 1	0	vrai
4 (2)	1		vrai

La boucle for : exemple

```
1  #include <stdio.h>
2  int main(void)
3  {
4      for (int i = 0; i < 3; i++)
5          printf("%d\n", i);
6      return 0;
7  }
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

Ligne	i	affichage	test
4 (1) 4 (2) 5 4 (3)	0 0 0 1	0	vrai
4 (2) 5	1 1	1	vrai

La boucle for : exemple

```
1  #include <stdio.h>
2  int main(void)
3  {
4      for (int i = 0; i < 3; i++)
5          printf("%d\n", i);
6      return 0;
7  }
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

Ligne	i	affichage	test
4 (1)	0		
4 (2)	0		vrai
5	0	0	
4 (3)	1		
4 (2)	1		vrai
5	1	1	
4 (3)	2		

La boucle for : exemple

```
1  #include <stdio.h>
2  int main(void)
3  {
4      for (int i = 0; i < 3; i++)
5          printf("%d\n", i);
6      return 0;
7  }
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

Ligne	i	affichage	test
4 (1)	0		
4 (2)	0		vrai
5	0	0	
4 (3)	1		
4 (2)	1		vrai
5	1	1	
4 (3)	2		
4 (2)	2		vrai

La boucle for : exemple

```
1  #include <stdio.h>
2  int main(void)
3  {
4      for (int i = 0; i < 3; i++)
5          printf("%d\n", i);
6      return 0;
7  }
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

Ligne	i	affichage	test
4 (1) 4 (2) 5 4 (3)	0 0 0 1	0	vrai
4 (2) 5 4 (3)	1 1 2	1	vrai
4 (2) 5	2 2	2	vrai

La boucle for : exemple

```
1  #include <stdio.h>
2  int main(void)
3  {
4      for (int i = 0; i < 3; i++)
5          printf("%d\n", i);
6      return 0;
7  }
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

Ligne	i	affichage	test
4 (1)	0		
4 (2)	0		vrai
5	0	0	
4 (3)	1		
4 (2)	1		vrai
5	1	1	
4 (3)	2		
4 (2)	2		vrai
5	2	2	
4 (3)	3		

La boucle for : exemple

```
1  #include <stdio.h>
2  int main(void)
3  {
4      for (int i = 0; i < 3; i++)
5          printf("%d\n", i);
6      return 0;
7  }
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

Ligne	i	affichage	test
4 (1)	0		
4 (2)	0		vrai
5	0	0	
4 (3)	1		
4 (2)	1		vrai
5	1	1	
4 (3)	2		
4 (2)	2		vrai
5	2	2	
4 (3)	3		
4 (2)	3		faux

La boucle for : exemple

```
1  #include <stdio.h>
2  int main(void)
3  {
4      for (int i = 0; i < 3; i++)
5          printf("%d\n", i);
6      return 0;
7  }
```

Question

Simulez l'exécution de ce code.
Que fait-il ?

Ligne	i	affichage	test
4 (1)	0		
4 (2)	0		vrai
5	0	0	
4 (3)	1		
4 (2)	1		vrai
5	1	1	
4 (3)	2		
4 (2)	2		vrai
5	2	2	
4 (3)	3		
4 (2)	3		faux
6	3		

La boucle for : relation avec la boucle while

```c
#include <stdio.h>
int main(void)
{
    for (int i = 0; i < 3; i++)
        printf("%d\n", i);
    return 0;
}
```

La boucle for : relation avec la boucle while

```c
#include <stdio.h>
int main(void)
{
    for (int i = 0; i < 3; i++)
        printf("%d\n", i);
    return 0;
}
```

```c
#include <stdio.h>
int main(void)
{
    int i = 0;
    while (i < 3)
    {
        printf("%d\n", i);
        i++;
    }
    return 0;
}
```

La boucle for : relation avec la boucle while

```c
#include <stdio.h>
int main(void)
{
    for (int i = 0; i < 3; i++)
        printf("%d\n", i);
    return 0;
}
```

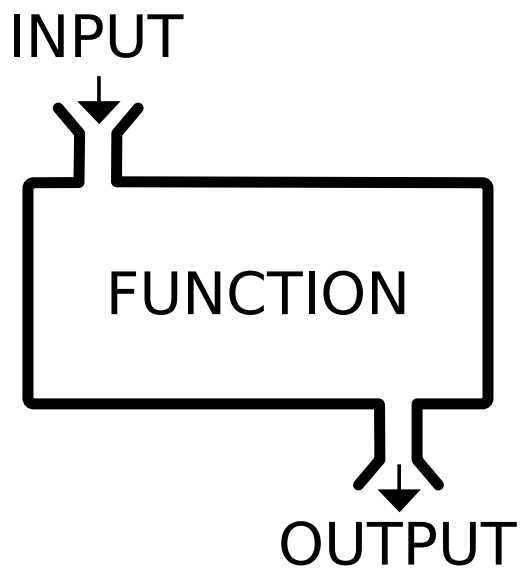
```c
#include <stdio.h>
int main(void)
{
    int i = 0;
    while (i < 3)
    {
        printf("%d\n", i);
        i++;
    }
    return 0;
}
```

Attention !

Il n'est pas possible de déclarer de variables dans la première partie du *for* avant la norme C99.

Sommaire

Les fonctions : introduction



Une fonction informatique peut être vue comme une **boîte noire**.

- Elle dispose d'une **entrée**. C'est un ensemble de variables que la fonction reçoit lorsqu'elle est **appelée**. Ces variables peuvent être utilisées à l'intérieur de la fonction.
- Elle dispose d'une **sortie**. C'est un ensemble de variables que la fonction émet lorsqu'elle termine. Ces variables peuvent être récupérées par l'entité qui a **appelé** la fonction (une autre fonction, nommée **fonction appellante**).
- La fonction réalise un ensemble d'opérations. En C, ces opérations sont décrites par une suite d'instructions.

Fonctions informatiques et mathématiques

Les fonctions informatiques sont plus que des fonctions mathématiques !

- Rien n'oblige à ce qu'une fonction informatique renvoie toujours la même sortie lorsqu'elle reçoit les mêmes entrées. Une fonction informatique est dite **déterministe** lorsque son comportement est entièrement défini par ses entrées.
- Les fonctions mathématiques sont abstraites et ne prennent pas en compte la **mémoire**. Les fonctions informatiques en tiennent compte (explicitement ou non).
- Une fonction informatique peut n'avoir aucune sortie.

Définition en C

- Une fonction est une suite d'instructions qui effectuent une tâche. Chaque programme C contient au moins une fonction (la fonction *main*).
- Les fonctions ont un **nom**, un **type de retour**, des **paramètres d'entrée** (ou *arguments*) et un **corps**.
- Une fonction qui ne renvoie rien (dont le type de retour est **void**) est appellée *procédure*.

Exemple de fonction

```c
#include <stdio.h> // Déclare les fonctions scanf, printf...

// Définition de la fonction f qui renvoie un entier
// et en reçoit un en paramètre (nommé i)
int f(int i)
{
    return i * 2; // Quitte la fonction f, en renvoyant i*2
}

// Définition de la fonction main qui renvoie un entier
// et ne reçoit aucun paramètre
int main(void)
{
    int a, b;
    scanf("%d", &a); // Appel de la fonction scanf
    b = f(a); // Appel de la fonction f que nous avons définie
    printf("f(%d) = %d\n",a,b); // Appel de la fonction printf
    return 0; // Quitte la fonction main, en renvoyant 0
}
```

Fonctions : déclaration et définition

Déclaration d'une fonction

Déclarer une fonction donne des informations sur la fonction au compilateur. Ces informations doivent comporter le **nom** de la fonction, son **type de retour** et le **type de ses paramètres**. Les noms des paramètres sont optionnels.

```
int somme_entiers(int, int, int);
```

La ligne ci-dessus forme le **prototype** de la fonction somme_entiers.

Fonctions : déclaration et définition

Déclaration d'une fonction

Déclarer une fonction donne des informations sur la fonction au compilateur. Ces informations doivent comporter le **nom** de la fonction, son **type de retour** et le **type de ses paramètres**. Les noms des paramètres sont optionnels.

```
int somme_entiers(int, int, int);
```

La ligne ci-dessus forme le **prototype** de la fonction somme_entiers.

Définition d'une fonction

La définition d'une fonction donne le **corps** de la fonction. Dans celle-ci, les paramètres doivent être nommés.

```
int somme_entiers(int a, int b, int c)
{
    return a + b + c;
}
```

Fonctions : déclaration et définition (II)

Déclaration + définition

Il est tout à fait possible de définir une fonction sans l'avoir déclarée précédemment.

Attention !

Lorsqu'on **appelle** une fonction, il faut qu'elle ait été **déclarée** avant (*au-dessus* de l'appel).

Fonctions : déclaration et définition (II)

Déclaration + définition

Il est tout à fait possible de définir une fonction sans l'avoir déclarée précédemment.

Attention !

Lorsqu'on **appelle** une fonction, il faut qu'elle ait été **déclarée** avant (*au-dessus* de l'appel).

Note

Lorsque vous incluez des fichiers de la bibliothèque standard, ils contiennent essentiellement des déclarations de fonctions. Si vous voulez vous servir de telles fonctions sans avoir inclus le fichier adéquat, vous obtiendrez une erreur ou un avertissement qui vous parlera très probablement de déclaration de ladite fonction.

Fonctions : paramètres formels et réels

On nomme différemment les paramètres selon s'ils sont utilisés dans la définition d'une fonction ou lors de son appel :

- Les paramètres **formels** sont ceux présents dans la définition d'une fonction.
- Les paramètres **réels** sont ceux présents lors de l'appel d'une fonction.

Lors d'un appel de fonction, la **valeur** des paramètres réels est **copiée** dans la **valeur** des paramètres **formels**.

Fonctions : paramètres formels et réels : exemple

```c
#include <stdio.h>
int f(int i)
{
    return i * 2;
}
int main(void)
{
    int a, b;
    scanf("%d", &a);
    b = f(a);
    printf("f(%d) = %d\n", a, b);
    return 0;
}
```

Question

Quels sont les paramètres formels et réels de la fonction f dans le code ci-dessus ?

Fonctions : paramètres formels et réels : exemple

```c
#include <stdio.h>
int f(int i)
{
    return i * 2;
}
int main(void)
{
    int a, b;
    scanf("%d", &a);
    b = f(a);
    printf("f(%d) = %d\n", a, b);
    return 0;
}
```

Question

Quels sont les paramètres formels et réels de la fonction **f** dans le code ci-dessus ?

i est le paramètre **formel** de la fonction **f**.

Fonctions : paramètres formels et réels : exemple

```
1  #include <stdio.h>
2  int f(int i)
3  {
4      return i * 2;
5  }
6  int main(void)
7  {
8      int a, b;
9      scanf("%d", &a);
10     b = f(a);
11     printf("f(%d) = %d\n", a, b);
12     return 0;
13 }
```

Question

Quels sont les paramètres formels et réels de la fonction **f** dans le code ci-dessus ?

i est le paramètre **formel** de la fonction **f**.

a est le paramètre **réel** de la fonction **f** lors de son appel ligne 10.

Retour d'une fonction

Le mot-clé return

- Le mot-clé **return** permet de quitter la fonction courante en renvoyant une valeur.
- Ce mot-clé n'est pas une fonction ! Il n'est pas nécessaire de placer des parenthèses autour de ce que l'on souhaite renvoyer.
- Si le type de retour de la fonction est **void**, on peut tout de même la quitter avec un **return**. Celui-ci n'a cependant pas de paramètre.

Retour d'une fonction : exemple

```c
#include <stdio.h>

// affiche i s'il est positif
void procedure(int i)
{
    if (i < 0)
        return;
    else
        printf("%d\n", i);
}

// valeur absolue d'un flottant
float val_abs(float f)
{
    if (f < 0)
        return -f;
    else
        return f;
}

int main()
{
    int entier;
    float flottant, absolue;

    printf("Entrez un entier : ");
    scanf("%d", &entier);
    printf("Entrez un flottant : ");
    scanf("%f", &flottant);

    procedure(entier);
    absolue = val_abs(flottant);
    printf("abs(%g) = %g\n", flottant,
           absolue);
    return 0;
}
```

Sommaire

La portée des variables

Définition

Ce que l'on appelle la portée d'une variable est une région du programme au sein de laquelle la variable existe et peut être utilisée. En dehors de cette portée, la variable n'existe pas et ne peut donc pas être utilisée.

La portée des variables

Définition

Ce que l'on appelle la portée d'une variable est une région du programme au sein de laquelle la variable existe et peut être utilisée. En dehors de cette portée, la variable n'existe pas et ne peut donc pas être utilisée.

Différentes portées

En C, on peut déclarer des variables à trois endroits :

- En dehors de toute fonction. De telles variables sont dites **globales**.
- Dans les paramètres de la définition d'une fonction. Ces variables sont des **paramètres formels**.
- À l'intérieur d'une fonction ou d'un bloc. De telles variables sont dites **locales**.

Variables globales

Les variables globales ont une portée globale : elles peuvent être utilisées depuis n'importe quelle fonction. Elles ont donc l'avantage de ne pas devoir être transmises en tant que paramètre aux fonctions.

Variables globales

Les variables globales ont une portée globale : elles peuvent être utilisées depuis n'importe quelle fonction. Elles ont donc l'avantage de ne pas devoir être transmises en tant que paramètre aux fonctions. Je vous déconseille **fortement** de vous servir de telles variables pour les raisons suivantes :

- Ces variables sont une source de bugs. Puisqu'elles peuvent être modifiées depuis n'importe quelle partie du code, il est difficile de comprendre quand elles sont modifiées et d'avoir un contrôle clair sur l'exécution du code.
- Elles empêchent souvent la réentrance (le fait d'être utilisable simultanément par plusieurs tâches utilisatrices). Il ne faut donc pas s'en servir lorsqu'on crée des bibliothèques.
- Elles empêchent le parallélisme.

Les paramètres formels

- Ces variables ont pour portée celle de la fonction dans laquelle elles sont déclarées.
- Ces variables sont donc utilisables à l'intérieur de la fonction uniquement.
- Elles sont initialisées avec les valeurs transmises lors de l'appel de la fonction (les paramètres réels).
- Elles sont détruites à la fin de la fonction.

Les variables locales

- Ce sont les variables que vous déclarez explicitement à l'intérieur d'une fonction.
- En C89, les variables locales ne peuvent être déclarées qu'au début d'une fonction et elles ont pour portée ladite fonction.
- À partir du C99, les variables locales peuvent être déclarées n'importe où dans un bloc et ont la portée dudit bloc. Attention à l'instruction *switch*.

Portée des variables : exercice

Exercice

Dans le code suivant, listez toutes les variables en indiquant si elles sont globales, locales ou des paramètres formels. Indiquez la portée de chaque variable (un intervalle de lignes).

```
1   int a;
2   void b(char c, int d)
3   {
4       int e = d%4;
5       if (e == 3)
6       {
7           int f = e-1;
8       }
9   }
10  char g;
11  char h()
12  {
13      return 'h';
14  }
```

```
15  int main()
16  {
17      int j = 42;
18      char k;
19      if (j == 42)
20      {
21          char l = 'l';
22          k = l;
23      }
24      else
25      {
26          char m = 'm';
27          k = m;
28      }
29
30      return j%2;
31  }
```

Portée des variables : exercice

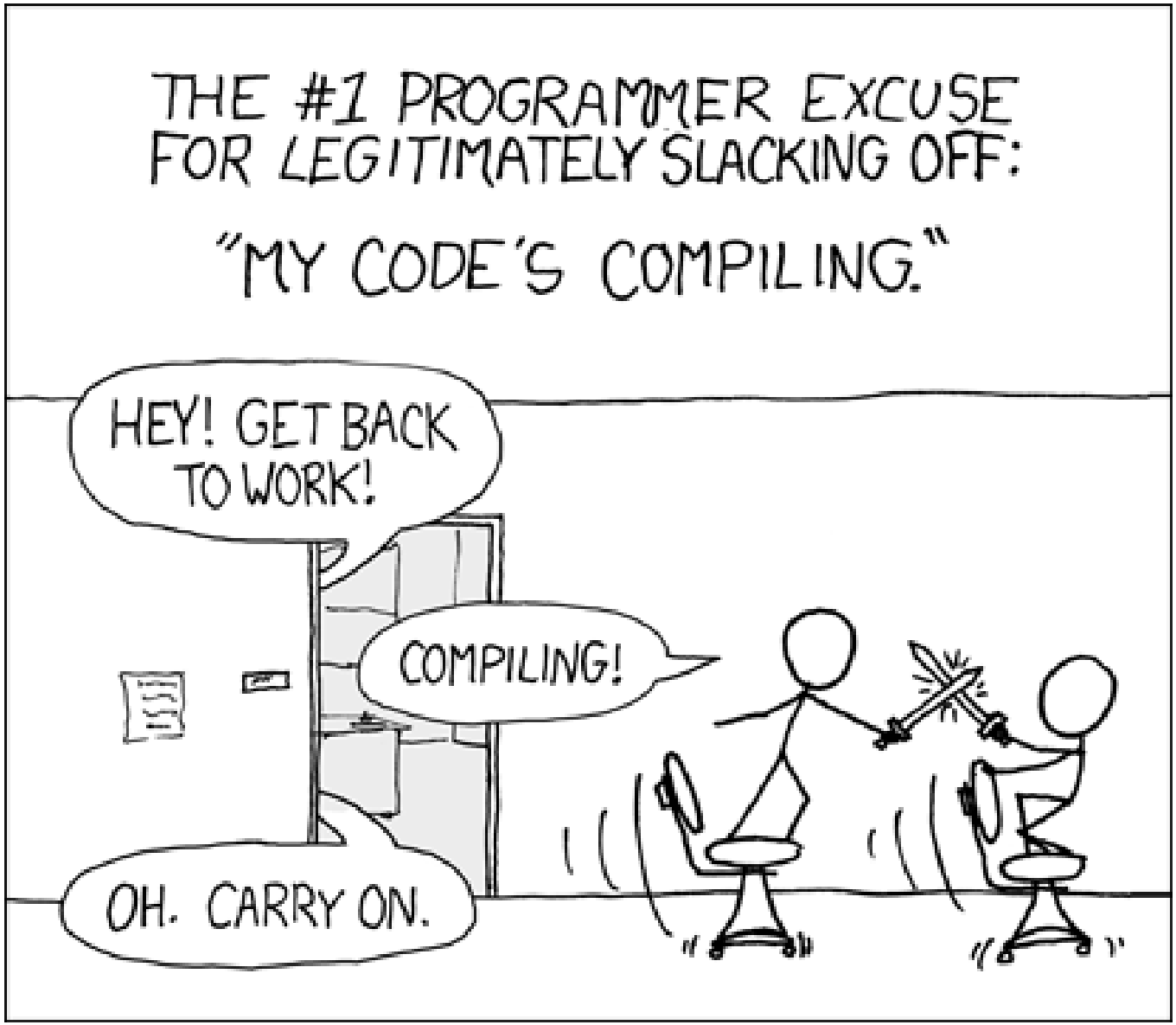
```
1   int a;
2   void b(char c, int d)
3   {
4       int e = d%4;
5       if (e == 3)
6       {
7           int f = e-1;
8       }
9   }
10  char g;
11  char h()
12  {
13      return 'h';
14  }
```

```
15  int main()
16  {
17      int j = 42;
18      char k;
19      if (j == 42)
20      {
21          char l = 'l';
22          k = l;
23      }
24      else
25      {
26          char m = 'm';
27          k = m;
28      }
29
30      return j%2;
31  }
```

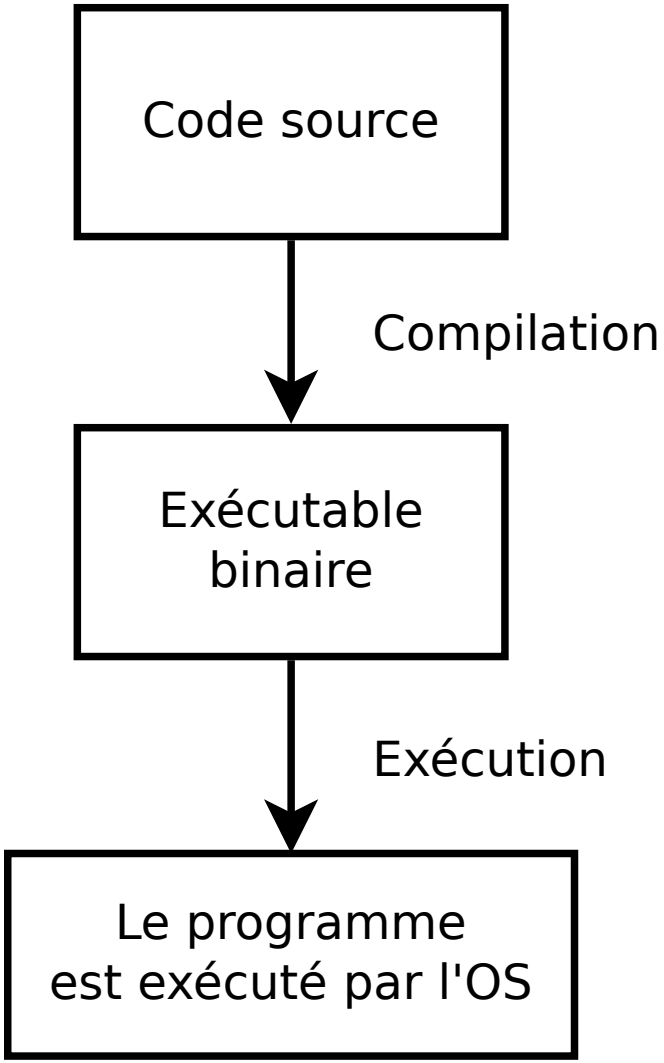
- **Variables globales** : a (1 → 31), g (10 → 31),
- **Paramètres formels** : c, d (3 → 9),
- **Variables locales** : e (4 → 9), f (7 → 8), j (17 → 31), k (18 → 31), l (21 → 23), m (26 → 28).

Sommaire

Compilation de projets



Compilation de projets : rappel de compilation



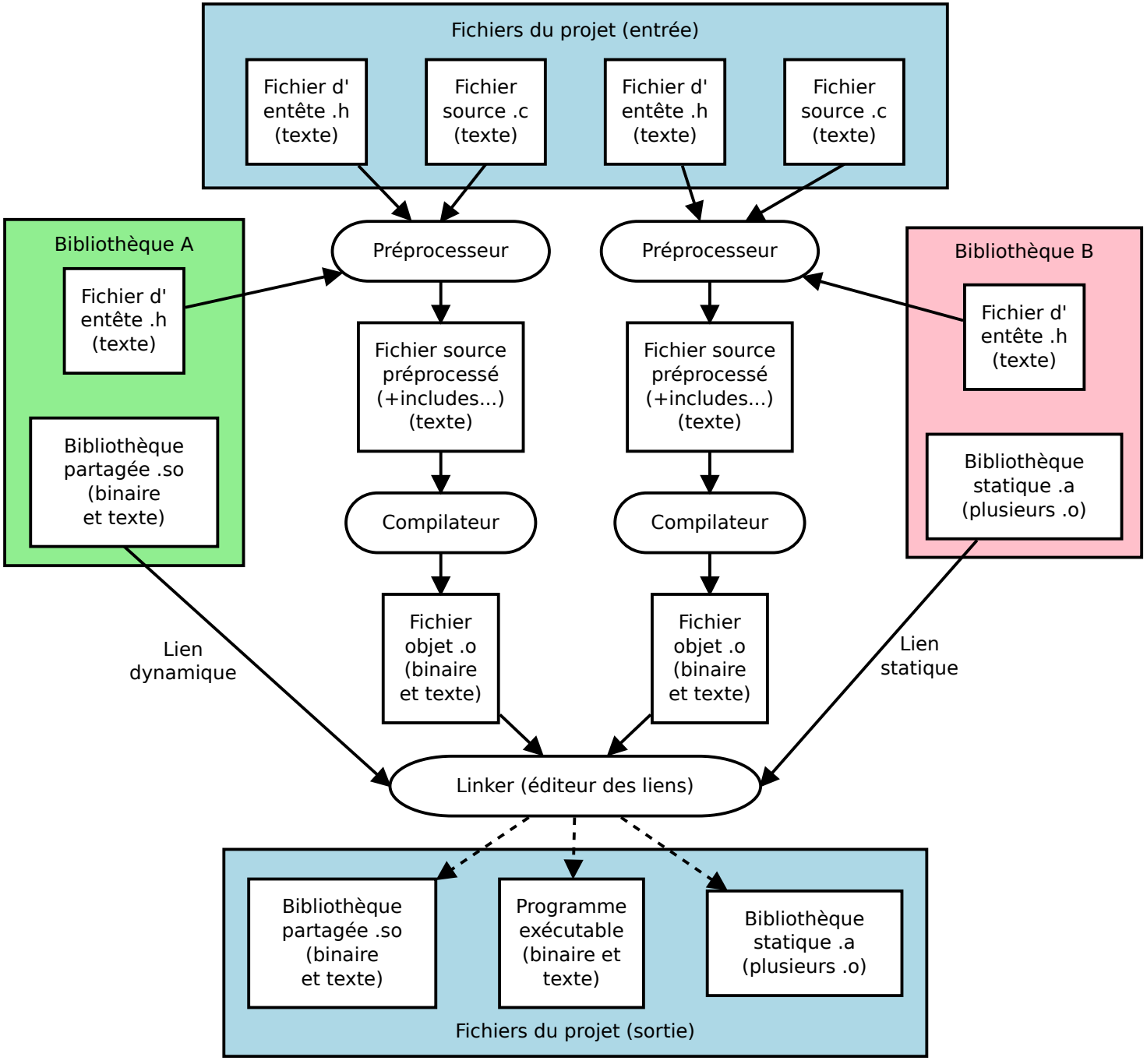
Attention

Ce schéma est simple.

Trop simple pour comprendre comment compiler un projet !

Langage compilé

Compilation de projets : schéma



Compilation de projets : notes

- Le **préprocesseur** est chargé de résoudre les différentes macros de vos codes : #include, #define, #ifdef... On peut l'appeler via la commande **cpp**.
- Le **compilateur** est chargé de transformer du code d'un langage source vers un langage de destination. Dans notre cas, il transforme du C en code machine (en passant par des *langages intermédiaires* puis des *langages d'assemblage*).
- L'éditeur des liens permet de regrouper différents fichiers objets **.o**, de relocaliser leurs données et de résoudre les références des différents fichiers **.o**. On peut l'appeler via la commande **ld**

Note

Il est assez rare de devoir appeler **cpp** ou **ld** explicitement. En général, on se sert du compilateur en lui donnant les options adéquates.

Fichiers d'en-tête et fichiers source

En général, en C, le code source est divisé en deux types de fichiers.

Les fichiers d'en-tête **.h**

- Déclarations de fonctions
- Déclarations et définitions de types

Les fichiers source **.c**

- Définition de fonctions

Une paire composée d'un fichier *foo.h* et d'un fichier *foo.c* forme un **module** *foo*.

Fichiers d'en-tête et fichiers source : exemple (1)

my_bool.h

```
1  // Évite les inclusions multiples.
2  // À préférer aux #ifndef et co.
3  #pragma once
4
5  // Déclaration + définition de types
6  enum my_bool
7  {
8      my_true = 1,
9      my_false = 0
10 };
11
12 // Déclaration de fonctions
13 void display_my_bool(enum my_bool b);
```

Fichiers d'en-tête et fichiers source : exemple (2)

my_bool.c

```
1  // Inclusion du .h correspondant
2  #include "my_bool.h" // "", pas <>
3
4  // Déclare la fonction printf
5  #include <stdio.h>
6
7  // Définition de fonctions
8  void display_my_bool(enum my_bool b)
9  {
10     if (b == my_false)
11         /* Appel de la fonction printf, déclarée mais
12            non définie. OK, on verra plus tard où est
13            le code de la fonction (édition des liens). */
14         printf("false\n");
15     else
16         printf("true\n");
17 }
```

Fichiers d'en-tête et fichiers source : exemple (3)

my_bool_main.c

```c
/* L'inclusion du fichier déclare et définit le type my_bool,
   et déclare la fonction display_my_bool */
#include "my_bool.h"

int main()
{
    // Instancie un my_bool. OK, le type est défini.
    enum my_bool b = 0;
    /* Appel de la fonction my_bool, déclarée mais non définie.
       OK, on verra plus tard où est le code de la fonction
       correspondante (lors de la résolution des liens).*/
    display_my_bool(b);

    b = 1;
    // Idem
    display_my_bool(b);

    return 0;
}
```

Fichiers d'en-tête et fichiers source : exemple (5)

Génération des fichiers objet **.o**

```
clang -c -o my_bool.o my_bool.c
clang -c -o my_bool_main.o my_bool_main.c
```

Génération d'un fichier exécutable **prog**

```
# Nos deux .o et la bibliothèque standard (pour printf)
clang -o prog my_bool.o my_bool_main.o -lc
```

Tout faire d'un coup

```
clang -o prog my_bool.c my_bool_main.c -lc
```

Je n'ai pas besoin de -lc d'habitude !

Certaines bibliothèques peuvent être liées par défaut avec vos programmes en fonction de votre environnement.

Compilation modulaire

Compiler tout un projet en une commande est pratique mais demande de tout recompiler même si on ne modifie qu'un seul fichier...

Faire toutes les étapes de construction à la main est laborieux...

Système de construction

Un **système de construction** (*build system*) est un outil qui permet de compiler un projet.

L'un des plus simples d'entre eux est le **Makefile**. Certains *build systems* permettent de générer des Makefile (CMake, qmake...).

Les IDE intègrent un ou des *build system(s)*.

Makefile qui compile le projet my_bool

```
1  compiler=clang
2  cflags=-std=c99 -Wall -Wextra -Werror -Wfatal-errors -g
3  ldflags=-lc
4
5  prog: my_bool.o my_bool_main.o
6      $(compiler) -o $@ $^ $(ldflags)
7
8  my_bool.o: my_bool.c my_bool.h
9      $(compiler) -o $@ -c $< $(cflags)
10
11 my_bool_main.o: my_bool_main.c
12     $(compiler) -o $@ -c $< $(cflags)
13
14 clean:
15     rm -f my_bool*.o
16
17 mrproper: clean
18     rm -f prog
```

Utilisation

```
make # Compile prog (et + si nécessaire)
make -B # force la reconstruction
make clean # supprime les .o
```

Comment créer un Makefile ?

Tutoriel ici.

Sommaire

Les tableaux

Définition

Un tableau est une structure de données qui permet de stocker un nombre fixe $n \in \mathbb{N}^*$ d'éléments du même type T.
Les différents éléments d'un tableau sont **contigus en mémoire**.

Exemple de tableau

```c
#include <stdio.h>
int main(void)
{
    // Déclaration d'un tableau de 4 caractères
    char a[4];
    // Initialisation du tableau
    a[0] = 'o'; // La première case contient 'o'
    a[1] = 'k'; // La seconde 'k'
    a[2] = 'a'; // La troisième 'a'
    a[3] = 'y'; // La dernière 'y'
    return 0;
}
```

Les tableaux (suite)

Quelques notes

- Un tableau contient différentes *cases*. Chaque *case* a une position dans le tableau. Cette position est appelée **indice**.
- Le premier indice d'un tableau est 0, pas 1.
- Vous pouvez initialiser les premières cases d'un tableau lors de sa définition. Si vous ne spécifiez pas toutes les cases, des 0 sont placés dans celles suivantes. Par défaut, le tableau n'est pas initialisé !
- Si vous initialisez un tableau directement, sa taille peut être omise.

Initialisation directe d'un tableau

```
4  // Déclaration + initialisation
5  char a[4] = {'o', 'k', 'a', 'y'};
6  // Taille omise
7  char b[] = {'\\', '[', 'T', ']', '/'};
```

Variables, tableaux et mémoire

```
1  int i = 0;
2  char a[2] = {'a', 'b'};
3  short int s = 0;
```

Adresse	Valeur	Variable
0	0	i
1	0	
2	0	
3	0	
4	'a'	a[0]
5	'b'	a[1]
6	0	s
7	0	

Une mémoire de 8 octets,
après la ligne 3 du programme

Variables, tableaux et mémoire

```
1  int i = 0;
2  char a[2] = {'a', 'b'};
3  short int s = 0;
```

Adresse	Valeur	Variable
0	0	i
1	0	
2	0	
3	0	
4	'a'	a[0]
5	'b'	a[1]
6	0	s
7	0	

Une mémoire de 8 octets,
après la ligne 3 du programme

Question

Que se passe-t-il si l'on exécute le code suivant ?

```
a[3] = 'X';
```

Variables, tableaux et mémoire

```
1  int i = 0;
2  char a[2] = {'a', 'b'};
3  short int s = 0;
```

Adresse	Valeur	Variable
0	0	i
1	0	
2	0	
3	0	
4	'a'	a[0]
5	'b'	a[1]
6	0	s
7	'X'	

Une mémoire de 8 octets, après avoir exécuté a[3] = 'X'

Question

Que se passe-t-il si l'on exécute le code suivant ?

```
a[3] = 'X';
```

Attention !

En C, vous êtes responsables des adresses auxquelles vous accédez. Vous pouvez facilement corrompre vos variables si vous ne faites pas attention aux indices des cases des tableaux dont vous vous servez.

L'opérateur sizeof

Définition

L'opérateur unaire **sizeof** permet de calculer la taille, en nombre d'octets[a], de n'importe quel type.
Cet opérateur peut également être appelé sur une variable et permet de connaître la taille qu'elle occupe en mémoire.

a. Ou plutôt en nombre de bytes, mais supposons que 1 octet = 1 byte.

Exemple

```
sizeof(int); // 4 en x86_64
int i;
sizeof(i); // égal à sizeof(int)
```

Tableaux et sizeof

Vous pouvez trouver des exemples de code où **sizeof** est utilisé pour connaître la taille d'un tableau.

```
int array[4];
sizeof(array); // 4 * sizeof(int)
int size = sizeof(array) / sizeof(int); // 4
```

Attention

Je vous déconseille **fortement** d'utiliser cette méthode :

- Elle ne marche que sur des tableaux déclarés dans la fonction courante.
- Elle ne marche que sur des tableaux alloués statiquement.

Préférez stocker la taille du tableau dans une variable !

Tableaux multidimensionnels

Un **tableau multidimensionnel** de dimension $n \in \mathbb{N}, n > 1$ est un tableau dont les éléments sont des tableaux de dimension $n - 1$.

Exemple

```
5   // Déclaration seule
6   int matrix[4][4];
7   // Déclaration + initialisation
8   int i[2][2] = {{1,0}, {0,1}};
9   int i2[2][2] = {1,0,0,1}; // Identique à i (warning) !
10  // Toutes les dimensions sauf la plus grande doivent être définies
11  char words[][10] = {{'p', 'r', 'a', 'i', 's', 'e'},   // words[0]
12                      {'t', 'h', 'e'},                  // words[1]
13                      {'s', 'u', 'n'}                 }; // words[2]
14  bool b[][2][3] = {{{true, true, true},  {true, true, false}}, // b[0]
15                    {{true, false, true}, {true, false, true}},  // b[1]
16                    {{false, true, true}, {false, true, true}}}; // b[2]
17  // Accès
18  matrix[0][0] = 42;
19  printf("i[0][0] = %d\n", i[0][0]); // 1
20  printf("i[0][1] = %d\n", i[0][1]); // 0
```

Les tableaux et le C99

La norme C99 a bien amélioré la manière d'utiliser les tableaux. Elle permet notamment les tableaux à taille variable, que ce soit dans la définition de variables locales ou de paramètres formels.

Attention

Le code ci-dessous est invalide avant le C99 !

```
int size = 4;
int arr[size];
void func(int size, int arr[size]);
void func(int n, int m, int arr[n][m]);
```

Passer un tableau en paramètre

Voici un exemple qui permet de passer un tableau à une fonction.

```c
#include <stdio.h>
void f(int taille, int tab[taille])
{
    printf("Taille : %d\n", taille);
    printf("Contenu : [");
    if (taille > 0)
    {
        printf("%d", tab[0]);
        for (int i = 1; i < taille; ++i)
            printf(", %d", tab[i]);
    }
    printf("]\n");
}
```

```c
int main()
{
    int t[10];
    f(10, t);
    return 0;
}
```

Passer un tableau en paramètre (2)

Syntaxe

Il existe plusieurs syntaxes **strictement équivalentes** pour transmettre un tableau à 1 dimension en paramètre à une fonction. Ne soyez pas surpris de les croiser !

```
int func(int size, int * array);
int func(int size, int array[]);
int func(int size, int array[size]); //à partir de C99

int main()
{
    int arr[6] = {1,2,3,4,5,6};
    func(6, arr);
}
```

Passer un tableau en paramètre (3)

Il existe également différentes syntaxes pour passer un tableau multidimensionnel en paramètre.

```c
// C89 et supérieur
const int N = 2; // global ou pire
const int M = 3; // global ou pire
int func(int (*array)[M]);
// C99 et supérieur
int func(int n, int m, int (*array)[m]);
int func(int n, int m, int array[n][m]);
// La 1ere dimension peut être omise
int func(int n, int m, int array[][m]);
```

```c
int main() {
    int arr[2][3] = {{1,2,3}, {4,5,6}};
    func(2, 3, arr);
    return 0;}
```

Attention

La syntaxe suivante n'est pas équivalente et ne marche pas, même si elle génère "juste" un warning sur la plupart des compilateurs !

```c
int func(int n, int m, int ** array);
```

Explication : la fonction reçoit un pointeur vers un tableau et non un pointeur vers un pointeur, ce qui est différent. Plus de détails ici.

Passer un tableau en paramètre (4)

Rappel sur l'appel d'une fonction

Lors d'un appel de fonction, la **valeur** des paramètres réels est **copiée** dans la **valeur** des paramètres **formels**.

Attention

Lorsque vous passez un tableau en paramètre à une fonction, le contenu du tableau n'est pas copié !

En fait, en C, un tableau est identifié par une adresse, qui dit où le tableau commence dans la mémoire. C'est cette adresse qui est copiée depuis la fonction *appellante* vers la fonction *appelée*.

Ainsi, lorsqu'une fonction modifie un tableau qu'elle a reçu en paramètre, cette modification est conservée après la fin de l'appel de la fonction !

Sommaire

Les chaînes de caractères

Définition

En C, une chaîne de caractères (ou *string*) est un tableau de caractères dont la fin est marquée par un caractère nul. Le caractère nul a pour valeur 0 et s'écrit '\0'.

Exemple

```c
#include <stdio.h>
int main()
{
    const char hello1[] = {'H', 'e', 'l', 'l', 'o', '!', '\n', '\0'};
    const char * hello2 = "Hello!\n";
    printf("1 : %s", hello1);
    printf("2 : %s", hello2);
    return 0;
}
```

Exécution

```
1 : Hello!
2 : Hello!
```

Les chaînes de caractères (suite)

Exemple

```c
#include <stdio.h>
int main()
{
    const char hello1[] = {'H', 'e', 'l', 'l', 'o', '!', '\n', '\0'};
    const char * hello2 = "Hello!\n";
    printf("1 : %s", hello1);
    printf("2 : %s", hello2);
    return 0;
}
```

Explications

- Les deux syntaxes proposées sont équivalentes.
- Lorsqu'on écrit du texte entre double quotes, un '\0' est mis automatiquement à la fin du texte.

'H'	'e'	'l'	'l'	'o'	'!'	'\n'	'\0'

Le contenu des deux chaînes

La taille d'une chaîne de caractères

Attention

La taille d'une chaîne de caractères est différente de la taille du tableau qui permet de la stocker.

Exemple

```
5  char chaine[10];
6  chaine[0] = 'H';
7  chaine[1] = 'i';
8  chaine[2] = '!';
9  chaine[3] = '\0';
```

'H'	'i'	'!'	'\0'	?	?	?	?	?	?

Le contenu du tableau

- Le tableau est de taille 10.
- La chaîne est de taille 3.

Mmh ?

La bibliothèque standard C fournit un ensemble conséquent de fonctions pour manipuler les chaînes de caractères. En voici un extrait :

```c
#include <string.h>
size_t strlen(const char *str);
int strcmp(const char *str1, const char *str2);
char *strcpy(char *dest, const char *src);
char *strtok(char *str, const char *delim);
char *strcat(char *dest, const char *src);
#include <stdio.h>
int sprintf(char *str, const char *format, ...);
```

Nous allons malheureusement, pour raisons pédagogiques, ne pas nous servir tout de suite de ces fonctions. Lorsque vous développerez de vrais programmes, utilisez les fonctions de la bibliothèque standard plutôt que les vôtres !

Sommaire

Un problème épineux

On aimerait créer une fonction qui renvoie plusieurs valeurs. Dans cet exemple, un personnage abat un monstre, ce qui fait baisser la barre d'endurance du personnage et augmente son nombre d'âmes.

Code qu'on aimerait écrire

```
int,int abattre_monstre(int endurance, int ames)
{
    endurance -= 10;
    ames += 100;
    return endurance, ames;
}
```

Problème

Ce code est invalide en C !
Une fonction ne peut renvoyer qu'au plus une valeur en C...
Note : la modification des variables *endurance* et *ames* ne concerne qu'une copie locale à la fonction, elle n'a aucune incidence sur la fonction appelante.

Rappel : variable et mémoire

Les variables dont nous nous servons ne sont rien d'autre que des **suites de bits** situés dans la **mémoire** de notre ordinateur. Une variable se situe donc à une **adresse** en mémoire.

Adresse	**Valeur**	**Variable**
0	0	i
1	0	
2	0	
3	0	
4	'a'	a[0]
5	'b'	a[1]
6	0	s
7	0	

Une mémoire de 8 octets

L'opérateur unaire &

Définition

L'opérateur unaire **&**, ou opérateur *address_of*, permet de renvoyer l'adresse de son paramètre.

Exemple

```
int i=0;
char c='c';
int f=0;
printf("Adresse de i : %p\n", &i);
printf("Adresse de c : %p\n", &c);
printf("Adresse de f : %p\n", &f);
```

Exécution

```
Adresse de i : 0x7ffef7aa6e38
Adresse de c : 0x7ffef7aa6e37
Adresse de f : 0x7ffef7aa6e30
```

L'opérateur unaire & (suite)

```
int i=0;
char c='c';
int f=0;
```

```
Adresse de i : 0x7ffef7aa6e38
Adresse de c : 0x7ffef7aa6e37
Adresse de f : 0x7ffef7aa6e30
```

Adresse	Valeur	Variable
...	...	...
0x7ffef7aa6e30	0	f
0x7ffef7aa6e31	0	
0x7ffef7aa6e32	0	
0x7ffef7aa6e33	0	
...	...	...
0x7ffef7aa6e37	'c'	c
0x7ffef7aa6e38	0	i
0x7ffef7aa6e39	0	
0x7ffef7aa6e3a	0	
0x7ffef7aa6e3b	0	
...	...	...

La mémoire correspondante

Les pointeurs

Définition

Un pointeur est une variable qui permet de stocker une **adresse mémoire**.

Exemple

```
3  int i = 42; // On crée un entier i
4  int * p_i = &i; // On crée un pointeur vers i
5  printf("Valeur de i : %d\n", i);
6  printf("Adresse de i : %p\n", &i);
7  printf("Valeur de p_i : %p\n", p_i);
8  printf("Adresse de p_i : %p\n", &p_i);
```

Exécution

```
Valeur de i : 42
Adresse de i : 0x7ffed19c3f78
Valeur de p_i : 0x7ffed19c3f78
Adresse de p_i : 0x7ffed19c3f70
```

Les pointeurs (suite)

```
3   int i = 42; // On crée un entier i
4   int * p_i = &i; // On crée un pointeur vers i
```

```
Valeur de i : 42
Adresse de i : 0x7ffed19c3f78
Valeur de p_i : 0x7ffed19c3f78
Adresse de p_i : 0x7ffed19c3f70
```

Adresse	Valeur	Variable
...	...	...
0x7ffed19c3f70 ↓ 0x7ffed19c3f77	0x7ffed19c3f78	p_i
0x7ffed19c3f78	42	i
0x7ffed19c3f79		
0x7ffed19c3f7a		
0x7ffed19c3f7b		
...	...	...

La mémoire correspondante

Affectation d'un pointeur

Définition

On peut modifier la valeur d'un pointeur, c'est-à-dire la zone mémoire vers laquelle le pointeur pointe.

```
5       int i = 42, j = 51;
6       int * p = &i;
7       p = &j;
```

Adresse	Valeur	Variable
...	...	...
0x100 ↓ 0x107	0x108	p
0x108 ↓ 0x10b	42	i
0x10c ↓ 0x10f	51	j
...	...	...

La mémoire (après la ligne 6)

Affectation d'un pointeur

Définition

On peut modifier la valeur d'un pointeur, c'est-à-dire la zone mémoire vers laquelle le pointeur pointe.

```
5       int i = 42, j = 51;
6       int * p = &i;
7       p = &j;
```

Adresse	Valeur	Variable
...	...	...
0x100 ↓ 0x107	0x10c	p
0x108 ↓ 0x10b	42	i
0x10c ↓ 0x10f	51	j
...	...	...

La mémoire (après la ligne 7)

L'opérateur d'indirection *

Définition

L'opérateur unaire d'indirection (ou de déréférençage) * renvoie l'objet vers lequel un pointeur pointe.

```
4   int i = 42;
5   int * p = &i;
6   printf("Indirection de p : %d\n",
7      *p);

    Indirection de p : 42
```

Adresse	Valeur	Variable
...	...	...
0x100 ↓ 0x107	0x108	p
0x108 ↓ 0x10b	42	i
...	...	...

L'opérateur d'indirection * (suite)

Définition (suite)

L'opérateur unaire d'indirection * renvoie l'objet vers lequel un pointeur pointe, ce qui permet d'affecter directement ce qui est pointé par un pointeur.

```
4   int i = 42;
5   int * p = &i;
6   *p = 37;
7   printf("Valeur de i : %d\n", i);
```

Exécution

```
Valeur de i : 37
```

Adresse	Valeur	Variable
...	...	...
0x100 ↓ 0x107	0x108	p
0x108 ↓ 0x10b	42	i
...	...	...

La mémoire (après la ligne 5)

L'opérateur d'indirection * (suite)

Définition (suite)

L'opérateur unaire d'indirection * renvoie l'objet vers lequel un pointeur pointe, ce qui permet d'affecter directement ce qui est pointé par un pointeur.

```
4   int i = 42;
5   int * p = &i;
6   *p = 37;
7   printf("Valeur de i : %d\n", i);
```

Exécution

```
Valeur de i : 37
```

Adresse	Valeur	Variable
...	...	...
0x100 ↓ 0x107	0x108	p
0x108 ↓ 0x10b	37	i
...	...	...

La mémoire (après la ligne 6)

Comparaison de pointeurs

Des adresses peuvent être comparées via les opérateurs de comparaison usuels.

Exemple

```
5     int tab[10];
6     int * case0 = &tab[0];
7     int * case9 = &tab[9];
8     if (tab == case0)
9         printf("tab == &tab[0]\n");
10    if (case0 < case9)
11        printf("Les adresses de tab sont croissantes\n");
12    else
13        printf("Les adresses de tab sont décroissantes\n");
14
```

Exécution

```
tab == &tab[0]
Les adresses de tab sont croissantes
```

Arithmétique des pointeurs

Définition

Certaines opérations arithmétiques sont définies sur les pointeurs :

- $p + 10$ est l'adresse du dixième élément en mémoire situé après p.
- $p1 - p2$ renvoie le nombre d'élements entre les deux adresses.

Exemple

```
5     int tabI[10];
6     char tabC[10];
7     printf("Il y a %ld cases entre les cases"
8            " 2 et 9\n", &tabI[9] - &tabI[2]);
9     if (&tabC[2] + 4 == &tabC[6])
10        printf("Ça marche !\n");
```

Exécution

```
Il y a 7 cases entre les cases 2 et 9
Ça marche !
```

Arithmétique des pointeurs (suite)

Attention

Il est déconseillé d'utiliser l'arithmétique de pointeurs dans vos codes puisque cela rend les codes dépendants de l'architecture sous-jacente.

Attention

Les pointeurs sont typés en C. L'arithmétique des pointeurs dépend du type de vos pointeurs !

- Si p est un caractère, $p + 1$ pointe une adresse après p.
- Si p est un entier, $p + 1$ pointe X adresses après p, où X désigne la taille du type entier (en nombre d'octets).

L'opérateur unaire **sizeof**

L'opérateur unaire **sizeof** permet de calculer la taille, en nombre d'octets[2], de n'importe quel type.

Exemple

```
printf("sizeof(char) = %ld\n", sizeof(char));
printf("sizeof(unsigned char) = %ld\n", sizeof(unsigned char));
printf("sizeof(int) = %ld\n", sizeof(int));
printf("sizeof(long long) = %ld\n", sizeof(long long));
printf("sizeof(bool) = %ld\n", sizeof(bool));
printf("sizeof(float) = %ld\n", sizeof(float));
printf("sizeof(double) = %ld\n", sizeof(double));
printf("sizeof(long double) = %ld\n", sizeof(long double));
printf("sizeof(char*) = %ld\n", sizeof(char*));
printf("sizeof(int*) = %ld\n", sizeof(int*));
```

Exécution (x86-64, Arch Linux du 2016-08-18)

```
sizeof(char) = 1
sizeof(unsigned char) = 1
sizeof(int) = 4
sizeof(long long) = 8
sizeof(bool) = 1
sizeof(float) = 4
sizeof(double) = 8
sizeof(long double) = 16
sizeof(char*) = 8
sizeof(int*) = 8
```

2. Ou plutôt en nombre de bytes, mais supposons que 1 octet = 1 byte.

Un problème épineux (suite)

Rappel de la situation

On aimerait créer une fonction qui renvoie plusieurs valeurs. Dans cet exemple, un personnage abat un monstre, ce qui fait baisser la barre d'endurance du personnage et augmente son nombre d'âmes.

```c
#include <stdio.h>
void tuer_monstre(int * endurance,
                  int * ames)
{
    *endurance -= 10;
    *ames += 100;
}
```

```c
int main()
{
    int pts_endu = 100;
    int nb_ames = 0;
    tuer_monstre(&pts_endu,
                 &nb_ames);
    printf("Endu = %d\nAmes = %d\n",
           pts_endu, nb_ames);
    return 0;
}
```

Un problème épineux (suite)

```
1   #include <stdio.h>
2   void tuer_monstre(int * endurance,
3                     int * ames)
4   {
5       *endurance -= 10;
6       *ames += 100;
7   }
8   int main()
9   {
10      int pts_endu = 100;
11      int nb_ames = 0;
12      tuer_monstre(&pts_endu,
13                   &nb_ames);
14      printf("Endu = %d\nAmes = %d\n",
15             pts_endu, nb_ames);
16      return 0;
17  }
```

Adresse	Valeur	Variable
...	...	...
...	...	...
0x200	0	nb_ames
↓		
0x203		
0x204	100	pts_endu
↓		
0x207		
...	...	...

La mémoire (après la ligne 11)

Un problème épineux (suite)

```
1   #include <stdio.h>
2   void tuer_monstre(int * endurance,
3                     int * ames)
4   {
5       *endurance -= 10;
6       *ames += 100;
7   }
8   int main()
9   {
10      int pts_endu = 100;
11      int nb_ames = 0;
12      tuer_monstre(&pts_endu,
13                   &nb_ames);
14      printf("Endu = %d\nAmes = %d\n",
15             pts_endu, nb_ames);
16      return 0;
17  }
```

Adresse	Valeur	Variable
...	...	...
0x100 ↓ 0x107	0x200	ames
0x108 ↓ 0x10f	0x204	endurance
...	...	...
0x200 ↓ 0x203	0	nb_ames
0x204 ↓ 0x207	100	pts_endu
...	...	...

La mémoire (après la ligne 4)

Un problème épineux (suite)

```
1   #include <stdio.h>
2   void tuer_monstre(int * endurance,
3                     int * ames)
4   {
5       *endurance -= 10;
6       *ames += 100;
7   }
8   int main()
9   {
10      int pts_endu = 100;
11      int nb_ames = 0;
12      tuer_monstre(&pts_endu,
13                   &nb_ames);
14      printf("Endu = %d\nAmes = %d\n",
15             pts_endu, nb_ames);
16      return 0;
17  }
```

Adresse	Valeur	Variable
...	...	...
0x100 ↓ 0x107	0x200	ames
0x108 ↓ 0x10f	0x204	endurance
...	...	...
0x200 ↓ 0x203	0	nb_ames
0x204 ↓ 0x207	90	pts_endu
...	...	...

La mémoire (après la ligne 5)

Un problème épineux (suite)

```
1   #include <stdio.h>
2   void tuer_monstre(int * endurance,
3                     int * ames)
4   {
5       *endurance -= 10;
6       *ames += 100;
7   }
8   int main()
9   {
10      int pts_endu = 100;
11      int nb_ames = 0;
12      tuer_monstre(&pts_endu,
13                   &nb_ames);
14      printf("Endu = %d\nAmes = %d\n",
15             pts_endu, nb_ames);
16      return 0;
17  }
```

Adresse	Valeur	Variable
...	...	...
0x100 ↓ 0x107	0x200	ames
0x108 ↓ 0x10f	0x204	endurance
...	...	...
0x200 ↓ 0x203	100	nb_ames
0x204 ↓ 0x207	90	pts_endu
...	...	...

La mémoire (après la ligne 6)

Un problème épineux (suite)

```
1   #include <stdio.h>
2   void tuer_monstre(int * endurance,
3                     int * ames)
4   {
5       *endurance -= 10;
6       *ames += 100;
7   }
8   int main()
9   {
10      int pts_endu = 100;
11      int nb_ames = 0;
12      tuer_monstre(&pts_endu,
13                   &nb_ames);
14      printf("Endu = %d\nAmes = %d\n",
15             pts_endu, nb_ames);
16      return 0;
17  }
```

Adresse	Valeur	Variable
...	...	...
...	...	...
0x200	100	nb_ames
↓		
0x203		
0x204	90	pts_endu
↓		
0x207		
...	...	...

La mémoire (après la ligne 7)

Un problème épineux (suite)

```
1   #include <stdio.h>
2   void tuer_monstre(int * endurance,
3                     int * ames)
4   {
5       *endurance -= 10;
6       *ames += 100;
7   }
8   int main()
9   {
10      int pts_endu = 100;
11      int nb_ames = 0;
12      tuer_monstre(&pts_endu,
13                   &nb_ames);
14      printf("Endu = %d\nAmes = %d\n",
15             pts_endu, nb_ames);
16      return 0;
17  }
```

Adresse	Valeur	Variable
...	...	...
...	...	...
0x200	100	nb_ames
↓		
0x203		
0x204	90	pts_endu
↓		
0x207		
...	...	...

La mémoire (après la ligne 13)

Le passage de paramètre par adresse

Définition

La technique que nous venons de voir s'appelle le passage de paramètres par adresse. Elle permet à une fonction de modifier *réellement* ses paramètres (par opposition à une modification des paramètres formels uniquement).

Cette technique permet également d'éviter la copie d'objets volumineux (que nous créerons plus tard avec des structures). Dans ce cas, on utilise un pointeur constant pour dire que l'on ne veut pas (et que l'on ne doit pas !) modifier le paramètre.

Récapitulatif

```
void f(int * entier_modifiable, char * caractere_modifiable);
void f2(const char * chaine_non_modifiable);
```

Les tableaux ne sont pas beaucoup plus que des pointeurs !

```
int array[3] = {1,2,3};
printf("array = %p\n", array);
for (int i = 0; i < 3; ++i)
    printf("&array[%d] = %p\n",
           i, &array[i]);
```

Exécution

```
array = 0x7ffd5d56b134
&array[0] = 0x7ffd5d56b134
&array[1] = 0x7ffd5d56b138
&array[2] = 0x7ffd5d56b13c
```

Adresse	Valeur	Variable
0x7ffd5d56b134 ↓ 0x7ffd5d56b137	1	*array array[0]
0x7ffd5d56b138 ↓ 0x7ffd5d56b13b	2	*(array+1) array[1]
0x7ffd5d56b13c ↓ 0x7ffd5d56b140	3	*(array+2) array[2]

Pointeurs et tableaux (2D)

Les tableaux ne sont pas beaucoup plus que des pointeurs !

```
int array[2][2] = {{1,2},{3,4}};
printf("array = %p\n", array);
for (int i = 0; i < 2; ++i)
{
    printf("array[%d] = %p\n",
           i, array[i]);
    for (int j = 0; j < 2; ++j)
        printf("&array[%d][%d] = %p\n",
               i, j, &array[i][j]);
}
```

Adresse	Valeur	Variable
0x7ffd5d56b130 ↓ 0x7ffd5d56b133	1	*(*array) *(array[0]+0) array[0][0]
0x7ffd5d56b134 ↓ 0x7ffd5d56b137	2	*(*array+1) *(array[0]+1) array[0][1]
0x7ffd5d56b138 ↓ 0x7ffd5d56b13b	3	*(*array+2) *(array[1]+0) array[1][0]
0x7ffd5d56b13c ↓ 0x7ffd5d56b13f	4	*(*array+3) *(array[1]+1) array[1][1]

Exécution

```
array = 0x7ffd5d56b130
array[0] = 0x7ffd5d56b130
&array[0][0] = 0x7ffd5d56b130
&array[0][1] = 0x7ffd5d56b134
array[1] = 0x7ffd5d56b138
&array[1][0] = 0x7ffd5d56b138
&array[1][1] = 0x7ffd5d56b13c
```

Passage de paramètres

Un pointeur peut désigner une seule variable tout comme un tableau...

Par exemple, le type int* peut désigner un pointeur vers un entier comme un tableau d'entiers.

Comment éviter de telles ambiguïtés ?

- Préférez les syntaxes spécifiques aux tableaux
- Nommez clairement vos variables
- Documentez votre code !

Doxygen

Je vous conseille Doxygen pour la documentation de vos codes C. Vous pouvez en trouver un tutoriel ici ou là. Voici un exemple de fonction documentée :

```
/**
 * @brief Trie le tableau passé en paramètre de taille donnée.
 * @details L'algorithme utilisé est le tri par sélection.
 *
 * @param[in,out] tableau Le tableau qui est trié par la fonction.
 * @param[in] taille Le nombre d'éléments du tableau.
 */
void tri_selection(int * tableau, int taille);
```

Documentation

Doxygen

Je vous conseille Doxygen pour la documentation de vos codes C. Vous pouvez en trouver un tutoriel ici ou là. Voici un exemple de fonction documentée :

```
/**
 * @brief Trie le tableau passé en paramètre de taille donnée.
 * @details L'algorithme utilisé est le tri par sélection.
 *
 * @param[in,out] tableau Le tableau qui est trié par la fonction.
 * @param[in] taille Le nombre d'éléments du tableau.
 */
void tri_selection(int * tableau, int taille);
```

Générer la documentation

La plupart des IDE et des éditeurs de texte simplifient la création de documentation. Par exemple, sous *Qt Creator* ou *Sublime Text*, il suffit de commencer un commentaire de documentation au dessus d'une fonction pour que tous les champs soient générés automatiquement.

Sommaire

Les structures

En C, une structure est un type de données qui vous permet de regrouper un ensemble de variables. Contrairement aux tableaux, ces variables peuvent être de type différent.
On appelle **variable membre**, **attribut** ou encore **champ** une variable à l'intérieur d'une structure.

```c
#include <stdio.h>

struct color_t
{
    unsigned char r;
    unsigned char g;
    unsigned char b;
};
```

```c
int main()
{
    struct color_t rouge;
    rouge.r = 255;
    rouge.g = 0;
    rouge.b = 0;

    printf("Rouge = (%d,%d,%d)\n",
           rouge.r, rouge.g, rouge.b);
    printf("Rouge = #%02x%02x%02x\n",
           rouge.r, rouge.g, rouge.b);

    return 0;
}
```

Le mot-clé struct

En C, le mot-clé **struct** permet de déclarer ou de définir une structure.
Il est également nécessaire lorsqu'on veut **instancier** un objet d'un type structuré. Afin d'éviter ce mot-clé lors de l'instanciation, on peut se servir d'un **typedef**.

```c
#include <stdio.h>

struct color_t
{
    unsigned char r;
    unsigned char g;
    unsigned char b;
};
typedef struct color_t Color;

int main()
{
    Color rouge;
    rouge.r = 255;
    rouge.g = 0;
    rouge.b = 0;

    return 0;
}
```

Le mot-clé struct

En C, le mot-clé **struct** permet de déclarer ou de définir une structure.
Il est également nécessaire lorsqu'on veut **instancier** un objet d'un type structuré. Afin d'éviter ce mot-clé lors de l'instanciation, on peut se servir d'un **typedef**.

```c
#include <stdio.h>

typedef struct color_t
{
    unsigned char r;
    unsigned char g;
    unsigned char b;
} Color;
```

```c
int main()
{
    Color rouge;
    rouge.r = 255;
    rouge.g = 0;
    rouge.b = 0;

    return 0;
}
```

Le mot-clé struct

En C, le mot-clé **struct** permet de déclarer ou de définir une structure.
Il est également nécessaire lorsqu'on veut **instancier** un objet d'un type structuré. Afin d'éviter ce mot-clé lors de l'instanciation, on peut se servir d'un **typedef**.

```c
#include <stdio.h>

typedef struct
{
    unsigned char r;
    unsigned char g;
    unsigned char b;
} Color;

int main()
{
    Color rouge;
    rouge.r = 255;
    rouge.g = 0;
    rouge.b = 0;

    return 0;
}
```

Structures et structures

Il est tout à fait possible d'utiliser des structures dans d'autres structures.

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

struct color_t
{
    unsigned char r;
    unsigned char g;
    unsigned char b;
};

struct circle_t
{
    double x_center;
    double y_center;
    double radius;
    struct color_t border_color;
    struct color_t fill_color;
    bool is_filled;
    bool borderless;
};
```

```c
int main()
{
    struct color_t rouge = {255, 0, 0};
    struct circle_t cercle =
    {0,0,1,rouge,rouge,false,false};

    return 0;
}
```

L'opérateur .

Définition

L'opérateur . permet d'accéder aux champs d'une structure.

```c
#include <stdio.h>

struct color_t
{
    unsigned char r;
    unsigned char g;
    unsigned char b;
};

int main()
{
    struct color_t rouge;
    rouge.r = 255;
    rouge.g = 0;
    rouge.b = 0;

    printf("Rouge = (%d,%d,%d)\n",
           rouge.r, rouge.g, rouge.b);
    printf("Rouge = #%02x%02x%02x\n",
           rouge.r, rouge.g, rouge.b);

    return 0;
}
```

Exécution

```
(255,0,0)
#ff0000
```

Mmh ?

On peut très bien faire un pointeur vers une structure. On peut alors accéder aux champs en utilisant l'opérateur d'indirection * puis en utilisant l'opérateur . pour accéder au champ qui nous intéresse.

```c
#include <stdio.h>

struct color_t
{
    unsigned char r;
    unsigned char g;
    unsigned char b;
};

int main()
{
    struct color_t rouge = {255, 0, 0};
    struct color_t * r = &rouge;

    printf("Rouge = #%02x%02x%02x\n",
           (*r).r, (*r).g, (*r).b);

    return 0;
}
```

Attention !

Sans les parenthèses, vous obtiendrez une erreur car l'opérateur . a une plus haute priorité que l'opérateur *.

L'opérateur ->

L'opérateur -> permet de faire à la fois une indirection et d'accéder à un champ.

```c
#include <stdio.h>

struct color_t
{
    unsigned char r;
    unsigned char g;
    unsigned char b;
};

int main()
{
    struct color_t rouge = {255, 0, 0};
    struct color_t * r = &rouge;

    printf("Rouge = #%02x%02x%02x\n",
           r->r, r->g, r->b);

    return 0;
}
```

Structures et pointeurs (suite et fin)

Question

Et si on a un pointeur vers un pointeur ?

Structures et pointeurs (suite et fin)

Question

Et si on a un pointeur vers un pointeur ?

Réponse

Il n'existe pas d'opérateur pour faire une double indirection puis accéder à un champ. Rien ne nous empêche de faire une indirection *via* l'opérateur * pour obtenir un pointeur vers une structure. Nous n'avons ensuite plus qu'à utiliser l'opérateur -> pour accéder au champ qui nous intéresse.

```c
#include <stdio.h>

struct color_t
{
    unsigned char r;
    unsigned char g;
    unsigned char b;
};

int main()
{
    struct color_t rouge = {255, 0, 0};
    struct color_t * r = &rouge;
    struct color_t ** p = &r;

    printf("Rouge = #%02x%02x%02x\n",
           (*p)->r, (*p)->g, (*p)->b);

    return 0;
}
```

Sommaire

Les types énumérés

Définition

Un type énuméré est un type de données qui consiste en un ensemble de constantes, appelées **énumérateurs**.

```c
#include <stdio.h>

enum jour_semaine_t
{
    LUNDI,
    MARDI,
    MERCREDI,
    JEUDI,
    VENDREDI,
    SAMEDI,
    DIMANCHE
};

int main()
{
    enum jour_semaine_t jour = LUNDI;

    if (jour == SAMEDI || jour == DIMANCHE)
        printf("Weekend.\n");

    return 0;
}
```

Types énumérés et typedef

Il est tout à fait possible d'utiliser le mot-clé **typedef** sur un type énuméré.

```
#include <stdio.h>

enum jour_semaine_t
{
    LUNDI,
    MARDI,
    MERCREDI,
    JEUDI,
    VENDREDI,
    SAMEDI,
    DIMANCHE
};

typedef enum jour_semaine_t Jour;
```

Valeurs par défaut des énumérateurs

Premier énumérateur

Par défaut, le premier énumérateur a pour valeur 0.

Les autres énumérateurs

Par défaut, la valeur des autres énumérateurs est égale celle de l'énumérateur les précédant incrémentée de 1.

Comment changer ce comportement ?

```
enum day_t
{
    MONDAY = -10,
    TUESDAY, // -9
    WEDNESDAY = 42
    THURSDAY, // 43
    // ...
};
```

Sommaire

La mémoire : plus de détails

Jusqu'à présent, toutes les variables que nous avons créées ont **allouées** dans un espace mémoire. Nous n'avons cependant pas eu à gérer explicitement cette mémoire :

- L'espace mémoire d'une variable était automatiquement alloué lors de la **définition** de la variable.
- L'espace mémoire d'une variable était automatiquement supprimé lorsque le programme quittait la **portée** de la variable.

Ce type d'allocation est très pratique et transparent ! Mais il est cependant assez limitant :(.

Questions

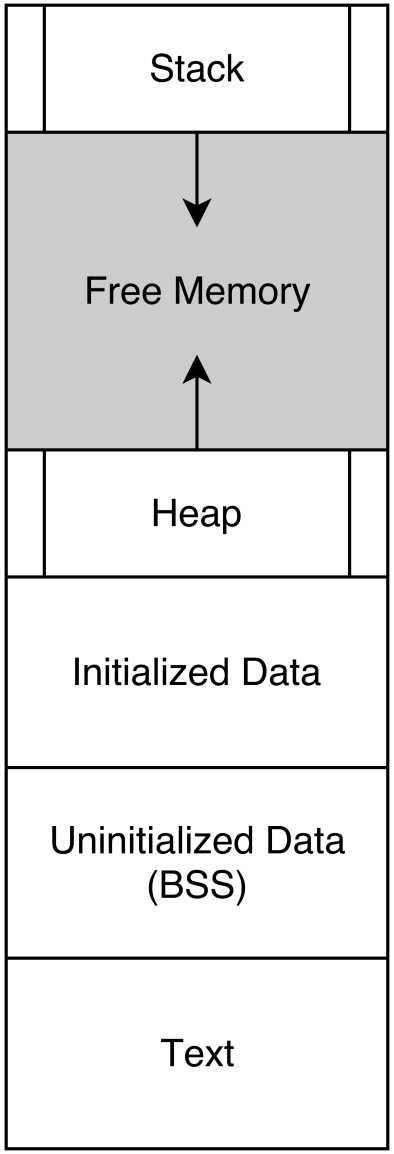
Comment gérer autrement la mémoire ?
Quels sont les différents espaces mémoire ?

Programme et processus

Lorsque le système d'exploitation exécute un programme, il ne l'exécute pas directement. Il **instancie** d'abord le programme dans la mémoire de l'ordinateur avant de l'exécuter.

À chaque instance de programme en cours d'exécution sont associés des espaces mémoire spécifiques. Un **processus** regroupe une instance de programme et les espaces mémoire qui y sont liés.

Les différents espaces mémoires d'un processus



(crédit à Majenko)

- Le segment de **code** (ou de **texte**) contient les instructions du processus.
- Le segment de **données** (ou **data**) contient les variables globales initialisées du programme.
- Le segment **bss** contient les variables globales non initialisées du programme.
- Le **tas** (ou **heap**) contient les variables qui sont allouées dynamiquement (ce que nous allons détailler ensuite).
- Enfin, la **pile** (ou **stack**) contient les variables locales et les paramètres formels du programme (dans une structure de pile, qui permet de savoir où on en est dans le programme, de savoir où aller en quittant une fonction...).

Un problème épineux

On souhaiterait créer une fonction **create_array** qui prend en paramètre un nombre **n**. Cette fonction doit créer un tableau de taille **n**, mettre toutes les cases du tableau à 1 et renvoyer le tableau nouvellement créé.

Une tentative de solution...

```
1  int * create_array(int n)
2  {
3      int array[n];
4      for (int i = 0; i < n; ++i)
5          array[i] = 1;
6      return array;
7  }
```

Un problème épineux

On souhaiterait créer une fonction **create_array** qui prend en paramètre un nombre **n**. Cette fonction doit créer un tableau de taille **n**, mettre toutes les cases du tableau à 1 et renvoyer le tableau nouvellement créé.

Une tentative de solution...

```
1  int * create_array(int n)
2  {
3      int array[n];
4      for (int i = 0; i < n; ++i)
5          array[i] = 1;
6      return array;
7  }
```

Qui ne marche pas !

La variable **array** peut être détruite dès la fin de la fonction ! Le pointeur renvoyé par la fonction peut donc être invalide. Un bon compilateur génère un warning sur ce code.

L'allocation dynamique

- Le **tas** (ou **heap**) est la zone mémoire du processus qui est laissée entièrement sous votre contrôle.
- En général, cette zone peut devenir très grande alors que la pile a une taille limitée (et assez petite).
- Les allocations dans cette zone sont faites lors de l'exécution de vos programmes. C'est pour cela qu'on les appelle **dynamiques**.

L'allocation dynamique : les fonctions à utiliser

Prototypes

```c
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

La fonction malloc

Prototype

```
#include <stdlib.h>
void *malloc(size_t size);
```

- La fonction **malloc** alloue **size** octets et renvoie un pointeur vers la zone mémoire allouée.
- La mémoire créée n'est pas initialisée.
- Si l'allocation n'a pas pu être réalisée, NULL est renvoyé.

La fonction calloc

Prototype

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

- La fonction **calloc** alloue de la mémoire pour un tableau de **nmemb** éléments de **size** octets chacun et renvoie un pointeur vers la mémoire allouée.
- La mémoire créée est initialisée à zéro.
- Si l'allocation n'a pas pu être réalisée, NULL est renvoyé.

La fonction free

Prototype

```
#include <stdlib.h>
void free(void *ptr);
```

- La fonction **free** libère l'espace mémoire sur lequel **ptr** pointe, qui doit avoir été renvoyé par un appel précédent à **malloc**, **calloc** ou **realloc**.
- Si **ptr** ne pointe pas vers un tel type d'espace mémoire ou si cet espace mémoire a déjà été libéré, un comportement indéfini se produit (en général, le programme plante).
- Si **ptr** vaut NULL, rien ne se produit.

La fonction realloc

Prototype

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

- La fonction **realloc** essaye de changer la taille du bloc mémoire sur lequel **ptr** pointe (préalablement alloué par **malloc**, **calloc** ou **realloc**).
- Cette fonction renvoie un pointeur vers une zone mémoire capable d'accueillir **size** octets.
- Attention, il est possible que la zone mémoire renvoyée commence ailleurs que **ptr**. C'est souvent le cas lorsqu'on souhaite agrandir une zone mémoire. Dans ce cas, la zone mémoire vers laquelle **ptr** pointait est copiée vers le début de la zone mémoire nouvellement allouée et l'ancienne zone mémoire est libérée par **free**.
- Si l'allocation n'a pas pu être réalisée, NULL est renvoyé.

Un problème épineux : suite

On souhaiterait créer une fonction **create_array** qui prend en paramètre un nombre **n**. Cette fonction doit créer un tableau de taille **n**, mettre toutes les cases du tableau à 1 et renvoyer le tableau nouvellement créé.

Un problème épineux : suite

On souhaiterait créer une fonction **create_array** qui prend en paramètre un nombre **n**. Cette fonction doit créer un tableau de taille **n**, mettre toutes les cases du tableau à 1 et renvoyer le tableau nouvellement créé.

```c
#include <stdlib.h>
int * create_array(int n)
{
    int * array = malloc(sizeof(int)*n);
    if (array)
    {
        for (int i = 0; i < n; ++i)
            array[i] = 1;
    }
    return array;
}
```

```c
int main()
{
    int * tab;
    tab=create_array(10);
    free(tab);
    return 0;
}
```

Les fuites mémoire

Définition

Une fuite (de) mémoire (ou *memory leak*) est un problème qui apparaît lorsqu'un programme ne gère pas correctement sa mémoire, de telle sorte que de la mémoire qui n'est plus utilisée n'est pas libérée.

Les fuites mémoire

Définition

Une fuite (de) mémoire (ou *memory leak*) est un problème qui apparaît lorsqu'un programme ne gère pas correctement sa mémoire, de telle sorte que de la mémoire qui n'est plus utilisée n'est pas libérée.

Comment les éviter ?

- Toujours faire attention à libérer au bon endroit/moment la mémoire qu'on a allouée **dynamiquement**.
- Vérifier régulièrement que le programme n'a pas de fuite grâce à **valgrind**.

Les fuites mémoire

Définition

Une fuite (de) mémoire (ou *memory leak*) est un problème qui apparaît lorsqu'un programme ne gère pas correctement sa mémoire, de telle sorte que de la mémoire qui n'est plus utilisée n'est pas libérée.

Comment les éviter ?

- Toujours faire attention à libérer au bon endroit/moment la mémoire qu'on a allouée **dynamiquement**.
- Vérifier régulièrement que le programme n'a pas de fuite grâce à **valgrind**.

Attention !

Pour que valgrind vous fournisse des informations utiles, il faut que le compilateur génère des informations de debug. L'option -g permet de les générer avec **clang** ou **gcc**.

Exemple d'utilisation de valgrind

Compilation de votre programme

```
clang -g hello.c -o hello
```

Exécution du programme via valgrind

```
valgrind --leak-check=full ./hello
```

Sortie de valgrind

```
==13732== HEAP SUMMARY:
==13732==     in use at exit: 4 bytes in 1 blocks
==13732==   total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==13732==
==13732== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==13732==    at 0x4C28BED: malloc (vg_replace_malloc.c:263)
==13732==    by 0x400559: main (hello.c:6)
==13732==
==13732== LEAK SUMMARY:
==13732==    definitely lost: 4 bytes in 1 blocks
==13732==    indirectly lost: 0 bytes in 0 blocks
==13732==      possibly lost: 0 bytes in 0 blocks
==13732==    still reachable: 0 bytes in 0 blocks
==13732==         suppressed: 0 bytes in 0 blocks
==13732==
==13732== For counts of detected and suppressed errors, rerun with: -v
==13732== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 4 from 4)
```

L'allocation dynamique : mot final

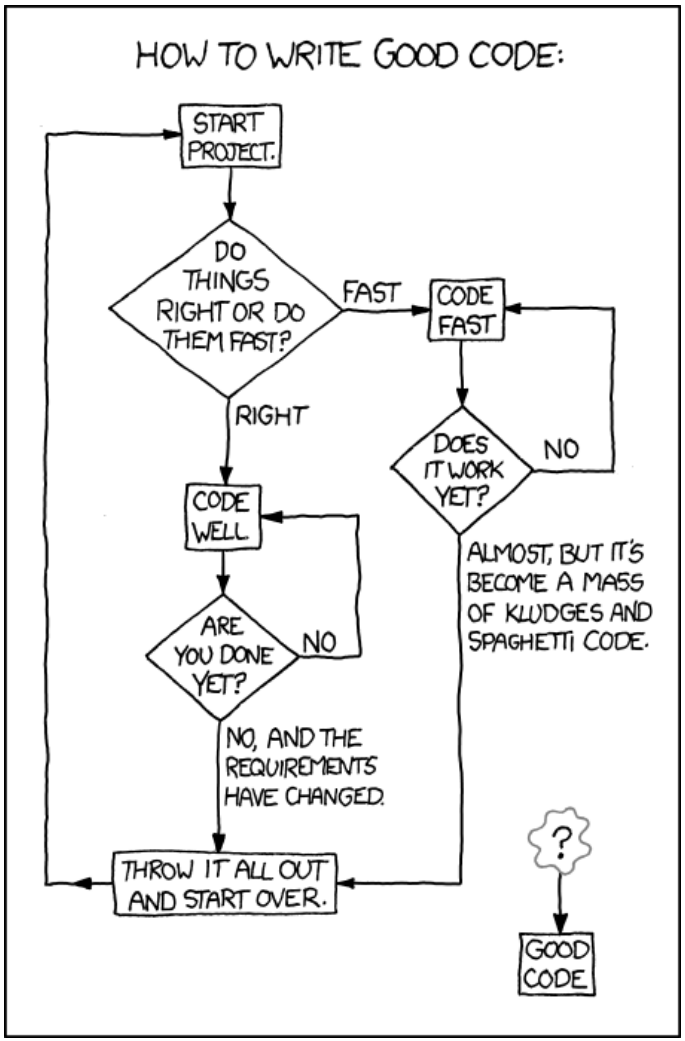
Quand utiliser l'allocation dynamique ?

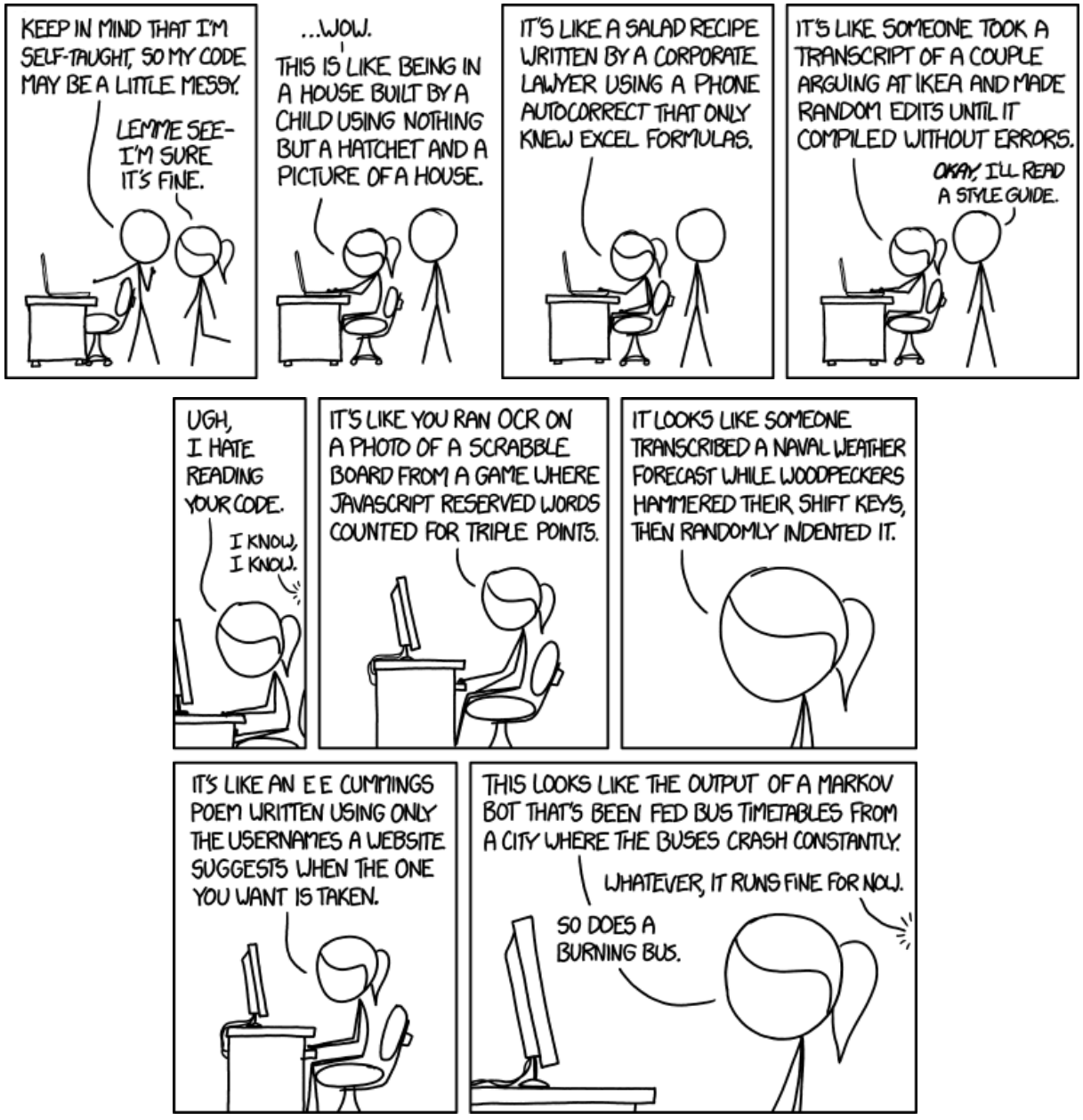
Très souvent dans un vrai programme !

- Si vous voulez allouer un tableau de grande taille (la pile est très petite !),
- Si vous voulez créer un tableau dont la taille ne peut être connue qu'à l'exécution (sauf depuis C99),
- Si vous voulez allouer une structure de données dynamique (tableau à taille variable, liste, arbre, graphe...).
- Si la durée de vie de variables ne peut être déterminée qu'à l'exécution. Typiquement, dès que l'utilisateur interagit avec le programme, ses actions peuvent créer des structures en mémoire ou les supprimer.

Sommaire

Bien programmer...





Citations

Citation

Programs should be written for people to read, and only incidentally for machines to execute.
Harold Abelson and Gerald Jay Sussman

Citation

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.
Martin Fowler

Citation

If the computer doesn't run it, it's broken. If people can't read it, it will be broken. Soon.
Charlie Martin

Critères de qualité d'un programme

- Correction : il doit respecter certaines spécifications si elles ont été faites.
- Robustesse : le programme ne doit jamais planter. Les entrées non désirées doivent être détectées et des messages clairs d'erreur doivent être émis.
- Fiabilité : le programme doit pouvoir marcher de longues périodes de temps, marcher quelles que soient les données utilisées...
- Efficacité : le programme ne doit pas être trop lent ou prendre trop de mémoire.
- Modifiabilité : le programme doit être simple à lire, à comprendre et à modifier.
- Portabilité : le programme doit pouvoir être exécuté sur diverses machines et environnements.
- Respect des standards, être bien documenté...

Critères conflictuels

Ces critères sont conflictuels...
La fiabilité et l'efficacité. Il est possible de réaliser des opérations flottantes très rapides si on ignore certains nombres spéciaux comme NaN !
La robustesse et la modifiabilité : un code très robuste sera probablement plus difficile à lire qu'un code plus simple.
L'efficacité et la modifiabilité : un code très optimisé n'est que très difficilement modifiable !
etc.

Où doit-on placer la barre ?

Ça dépend des cas !
De manière générale, certaines pratiques sont bonnes à prendre.

KISS

Keep **I**t **S**imple, **S**tupid.

- Pourquoi faire simple quand on peut faire compliqué ?
- Un programme simple est plus facile à maintenir et à comprendre
- Toute complexité non indispensable devrait être évitée dans toute la mesure du possible.
- Ne pas optimiser son code avant de maîtriser totalement une version simple de ce que l'on crée.
- Ai-je vraiment besoin de cette fonctionnalité ?
- GNU en est un bon exemple : un ensemble d'utilitaires simples et spécifiques, simples à combiner.

Programmation par contrat

La quasi-totalité des codes que vous créez ou utilisez se basent sur des règles implicites :

- **préconditions** : que doivent respecter les entrées pour que ce bout de code marche ?
- **postconditions** : que dois-je attendre de la sortie de ce bout de code ?
- **invariants** : qu'est-ce qui devrait être toujours vrai dans ce bout de code ?

La programmation par contrat propose d'expliciter ces règles via des **assertions**.
Ces **assertions** peuvent alors être vérifiées lors de l'exécution, ce qui permet de détecter tout de suite un cas anormal et d'arrêter le programme. De plus, il suffit de changer une option de compilation pour ne plus exécuter les assertions, ce qui permet de ne pas ralentir le programme en production.

Programmation par contrat : exemple

```
// Complexité : O(size)
bool is_sorted(int size,
               const int array[size]) {
    for (int i = 1; i< size; ++i)
        if (array[i-1] > array[i])
            return false;
    return true;
}
// Complexité : O(size)
bool all_different2(int size,
                    const int array[size]) {
    assert(is_sorted(size, array));
    for (int i = 1; i < size; ++i)
        if (array[i-1] == array[i])
            return false;
    return true;
}
```

```
// Complexité : O(size^2)
bool all_different(int size,
                   const int array[size]) {
    for (int i = 0; i < size; ++i) {
        for (int j = i+1; j < size; ++j) {
            if (array[i] == array[j])
                return false;
        }
    }
    return true;
}
```

Programmation par contrat : exemple

```c
// Complexité : O(size)
bool is_sorted(int size,
               const int array[size]) {
    for (int i = 1; i< size; ++i)
        if (array[i-1] > array[i])
            return false;
    return true;
}
// Complexité : O(size)
bool all_different2(int size,
                    const int array[size]) {
    assert(is_sorted(size, array));
    for (int i = 1; i < size; ++i)
        if (array[i-1] == array[i])
            return false;
    return true;
}
```

```c
// Complexité : O(size^2)
bool all_different(int size,
                   const int array[size]) {
    for (int i = 0; i < size; ++i) {
        for (int j = i+1; j < size; ++j) {
            if (array[i] == array[j])
                return false;
        }
    }
    return true;
}
```

Compilation & exécution (release)

```
$ clang99 -DNDEBUG=1 contract_programming.c && ./a.out
[1, 2, 3, 4] : all_different ? 1 1
[1, 5, 5, 7] : all_different ? 0 0
[8, 7, 6, 8] : all_different ? 0 1
```

Erreur invisible :(

Programmation par contrat : exemple

```c
// Complexité : O(size)
bool is_sorted(int size,
               const int array[size]) {
    for (int i = 1; i< size; ++i)
        if (array[i-1] > array[i])
            return false;
    return true;
}
// Complexité : O(size)
bool all_different2(int size,
                    const int array[size]) {
    assert(is_sorted(size, array));
    for (int i = 1; i < size; ++i)
        if (array[i-1] == array[i])
            return false;
    return true;
}
```

```c
// Complexité : O(size^2)
bool all_different(int size,
                   const int array[size]) {
    for (int i = 0; i < size; ++i) {
        for (int j = i+1; j < size; ++j) {
            if (array[i] == array[j])
                return false;
        }
    }
    return true;
}
```

Compilation & exécution (debug)

```
$ clang99 -g contract_programming.c && ./a.out
[1, 2, 3, 4] : all_different ? 1 1
[1, 5, 5, 7] : all_different ? 0 0
a.out: contract_programming.c:27: _Bool all_different2(int, const int *):
       Assertion `is_sorted(size, array)' failed.
Abandon (core dumped)
```

Erreur détectée ! Plus qu'à lancer gdb et on pourra en savoir plus.

Programmation par contrat : exemple

```
$ gdb ./a.out
{...}
(gdb) run
Starting program: /home/carni/teaching/inf111/cours/src/a.out
[1, 2, 3, 4] : all_different ? 1 1
[1, 5, 5, 7] : all_different ? 0 0
a.out: contract_programming.c:27: _Bool all_different2(int, const int *):
       Assertion `is_sorted(size, array)' failed.

Program received signal SIGABRT, Aborted.
0x00007ffff7a6b295 in raise () from /usr/lib/libc.so.6
(gdb) bt
#0  0x00007ffff7a6b295 in raise () from /usr/lib/libc.so.6
#1  0x00007ffff7a6c6da in abort () from /usr/lib/libc.so.6
#2  0x00007ffff7a64297 in __assert_fail_base () from /usr/lib/libc.so.6
#3  0x00007ffff7a64342 in __assert_fail () from /usr/lib/libc.so.6
#4  0x0000000000400765 in all_different2 (size=4, array=0x7fffffffde60) at
                       contract_programming.c:27
#5  0x0000000000400a67 in main () at contract_programming.c:65
```

→ L'appel de all_different2, depuis le main, ligne 65 de contract_programming.c, est invalide : il ne respecte pas la précondition → go fix your code.