MÉMOIRE, TYPES ET VARIABLES

Initiation à la programmation C

---

uf-info.ue.init-prog-c@diff.u-bordeaux.fr

université de BORDEAUX

BILAN DE L'EXERCICE À LA MAISON

ANALYSE GLOBALE

- Pourquoi des cases de même taille (unité indivisible) ?
  - Intérêts ?
  - Défauts ?
- Limitation forte des données enregistrées dans cet exercice ?
  - Comment surmonter cette limitation ?

ANALYSE GLOBALE

▷ Pourquoi des cases de même taille (unité indivisible) ?

▷ Intérêts ?

Plus simple à construire et à parcourir

▷ Défauts ?

▷ Limitation forte des données enregistrées dans cet exercice ?

▷ Comment surmonter cette limitation ?

ANALYSE GLOBALE

▷ Pourquoi des cases de même taille (unité indivisible) ?

▷ Intérêts ?

Plus simple à construire et à parcourir

▷ Défauts ?

Choix de l'unité et éventuelle perte de place si mal choisie

▷ Limitation forte des données enregistrées dans cet exercice ?

▷ Comment surmonter cette limitation ?

ANALYSE GLOBALE

▷ Pourquoi des cases de même taille (unité indivisible) ?

▷ Intérêts ?

Plus simple à construire et à parcourir

▷ Défauts ?

Choix de l'unité et éventuelle perte de place si mal choisie

▷ Limitation forte des données enregistrées dans cet exercice ?

▷ Comment surmonter cette limitation ?

Un seul type (ici des entiers positifs). Il faudrait être capable de coder d'autre types de données et d'indiquer qui est quoi ...

ANALYSE DE QUELQUES CAS

- Utilisation des couleurs
  - 1 couleur = 1 chiffre (e.g., 1 = bleu, 2 = rouge, 3 = noir, ...)
  - 1 couleur = 1 valeur (e.g., vert = 2015)
  - Codage binaire (uniquement 2 couleurs)
  - 1 couleur = 1 type d'information (e.g., rouge = Année de naissance)

ANALYSE DE QUELQUES CAS

▷ Utilisation des couleurs

▷ 1 couleur = 1 chiffre (e.g., 1 = bleu, 2 = rouge, 3 = noir, ...)

▷ 1 couleur = 1 valeur (e.g., vert = 2015) ⚠

▷ Codage binaire (uniquement 2 couleurs)

▷ 1 couleur = 1 type d'information (e.g., rouge = Année de naissance) ⚠

ANALYSE DE QUELQUES CAS

- Mémoire
  - Cases blanches = pas d'information
  - 1 information par ligne
  - Séparateur de données
  - Sens lecture (de G à D, de D à G, de H à B, ...)

ANALYSE DE QUELQUES CAS

- Informations de décodage
  - Associations couleurs
  - Ordre de stockage
  - Localisation
  - Longueur des données

ANALYSE DE QUELQUES CAS

- Différences
  - Sens de lecture
  - Choix codage
  - Organisation mémoire
  - Longueur de données

ANALYSE DE QUELQUES CAS

- Une couleur est associée à un chiffre
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - Sans le codage, impossible de décoder
- Les nombres sont représentés en représentation positionnelle (i.e., unité, dizaine, centaine, ...) comme à l'école
  -  = 2*1000 + 0*100 + 1*10 + 5*1
- Une information par ligne
  - Problème ?
- La taille a priori de certains nombres n'est pas connue
  - Solution ?

ANALYSE DE QUELQUES CAS

- Une couleur est associée à un chiffre
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - Sans le codage, impossible de décoder
- Les nombres sont représentés en représentation positionnelle (i.e., unité, dizaine, centaine, ...) comme à l'école
  -  = 2*1000 + 0*100 + 1*10 + 5*1
- Une information par ligne
  - Problème ? Perte de place
- La taille a priori de certains nombres n'est pas connue
  - Solution ?

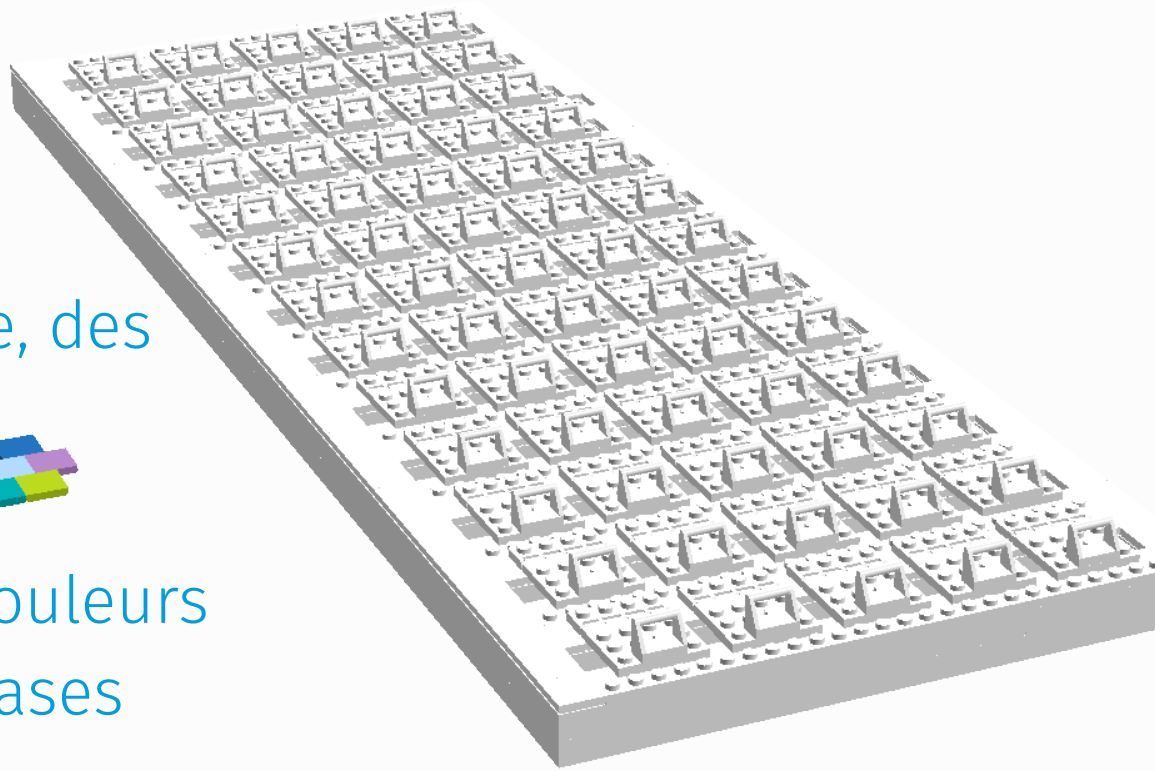
ANALYSE DE QUELQUES CAS

- Une couleur est associée à un chiffre
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - Sans le codage, impossible de décoder
- Les nombres sont représentés en représentation positionnelle (i.e., unité, dizaine, centaine, ...) comme à l'école
  -  = 2*1000 + 0*100 + 1*10 + 5*1
- Une information par ligne
  - Problème ? Perte de place
- La taille a priori de certains nombres n'est pas connue
  - Solution ? Fixer une taille maximale

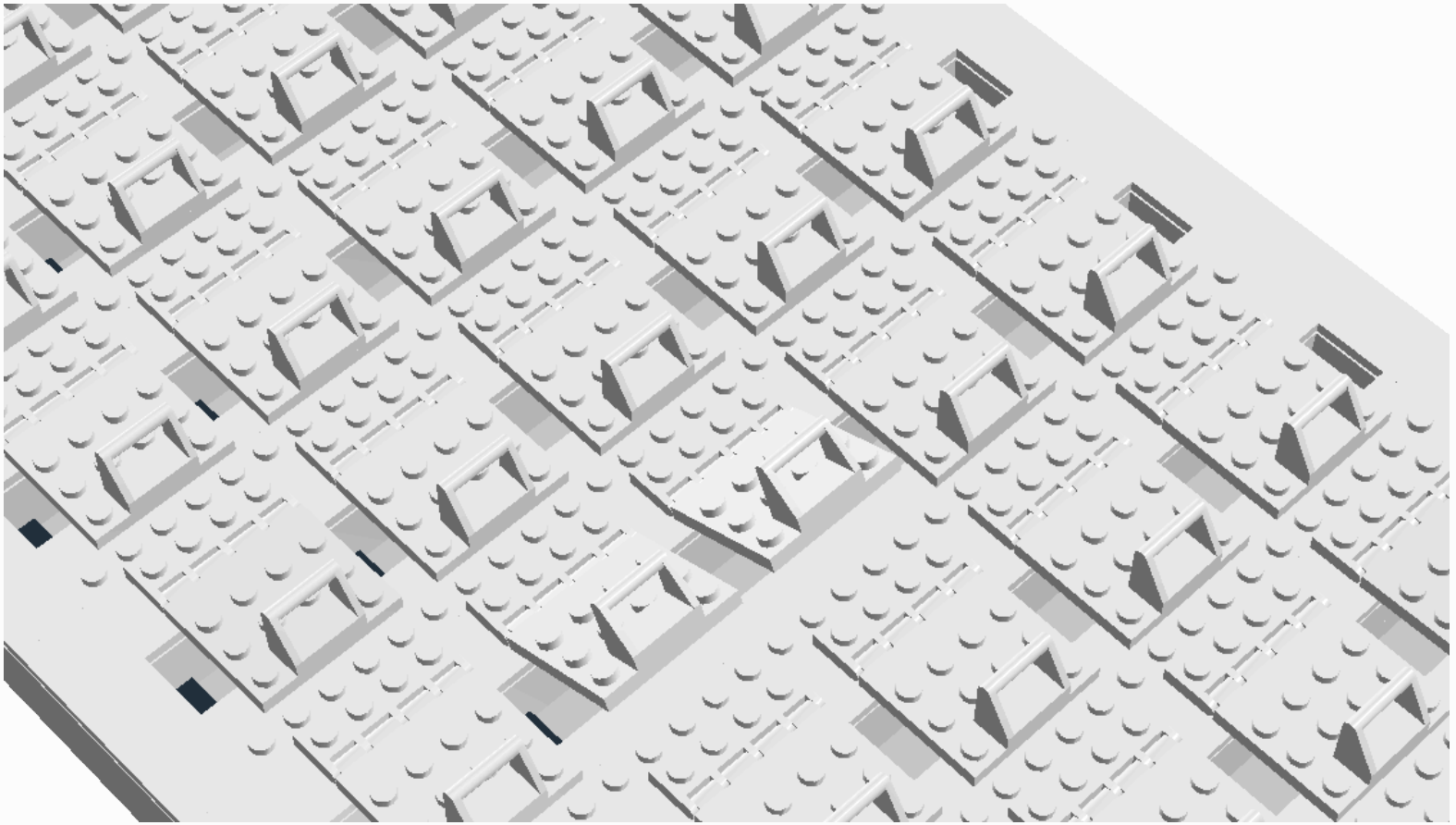
UN PEU DE LEGO

MODÈLE PLUS ÉVOLUÉ

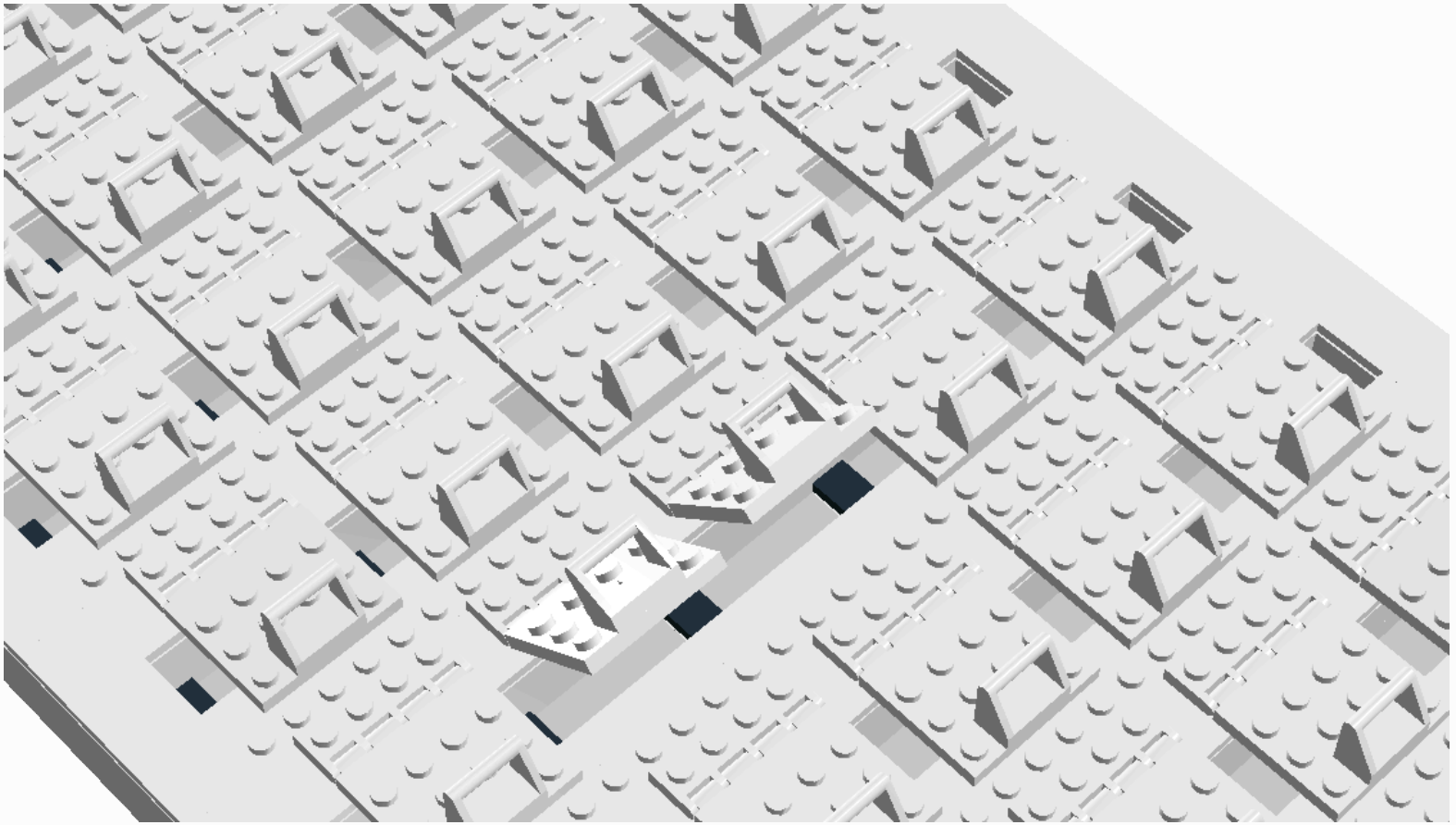
- Objectif :
  - Stocker du texte, des entiers et réels
- Moyens :
  - Seulement 10 couleurs
  - Un bloc de 65 cases



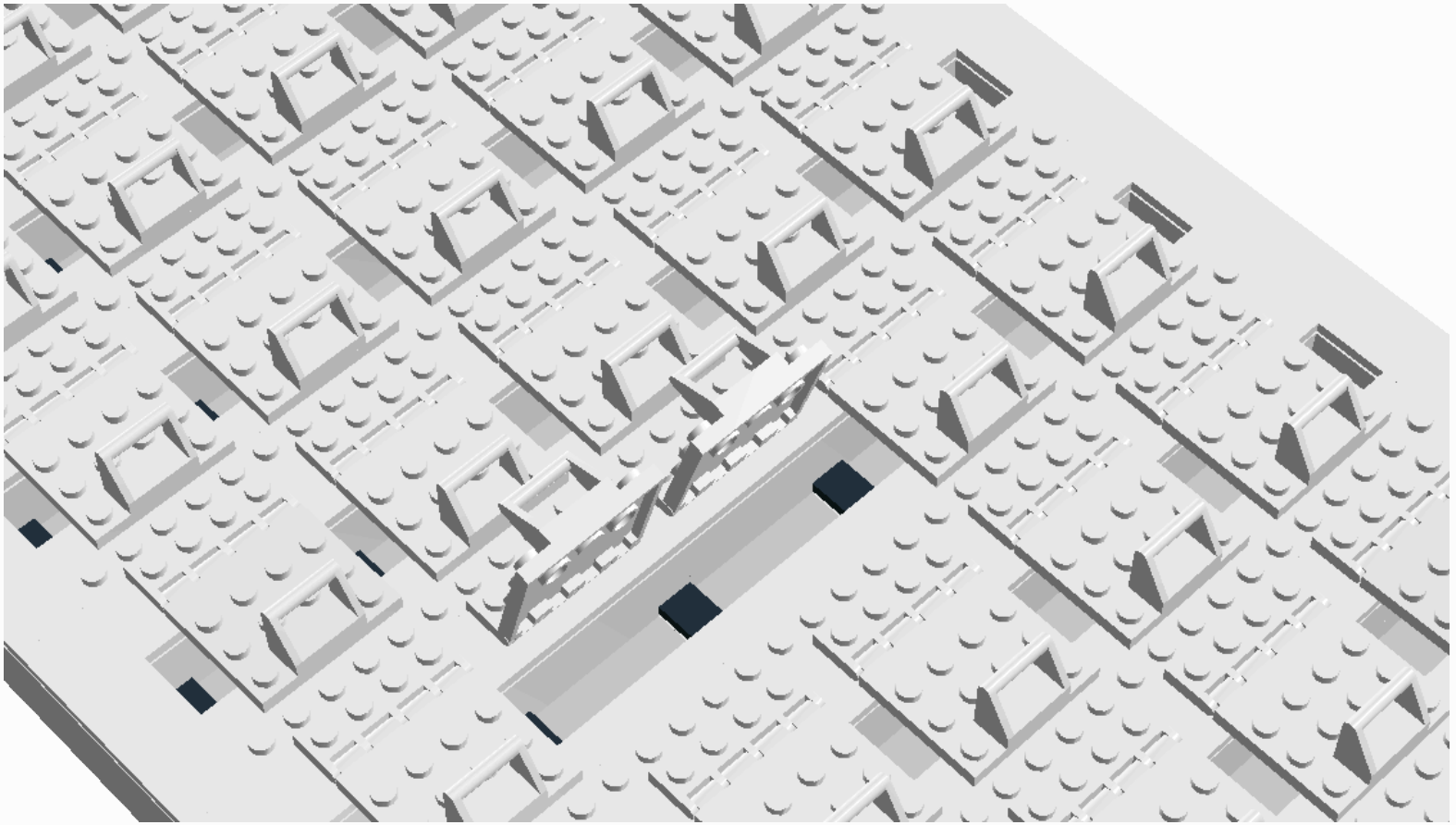
MODÈLE PLUS ÉVOLUÉ



MODÈLE PLUS ÉVOLUÉ



MODÈLE PLUS ÉVOLUÉ



Système d'adressage

▷ Proposer un système de coloriage pour que chaque case puisse être identifiée.

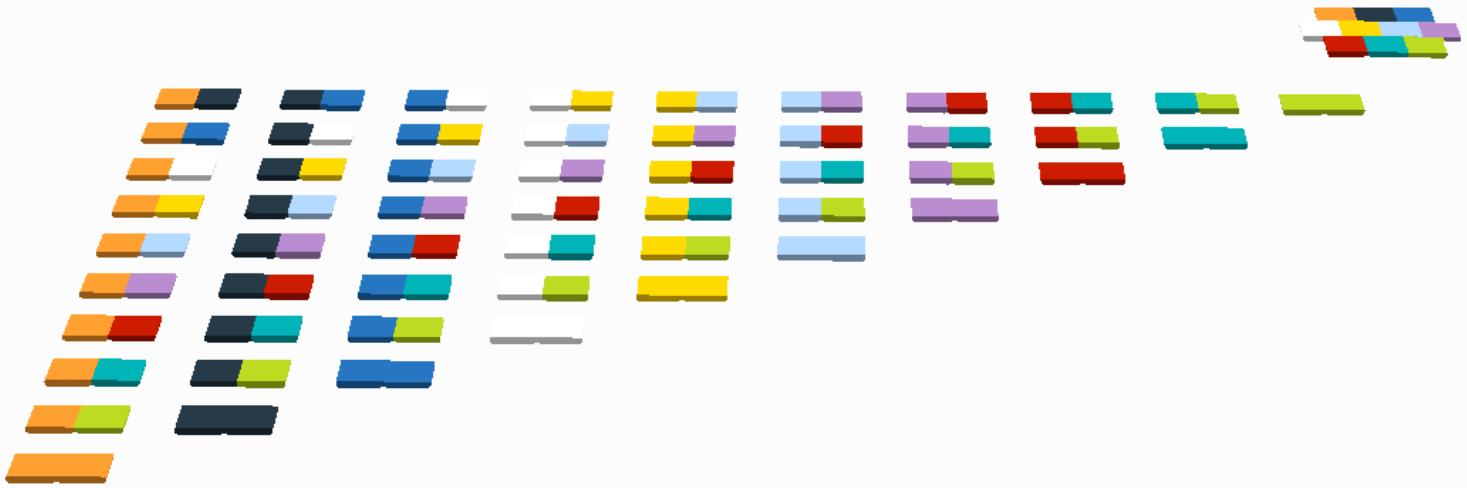
▷ Rappel : 65 cases et 10 couleurs...

Système d'adressage

▷ Proposer un système de coloriage pour que chaque case puisse être identifiée.

▷ Rappel : 65 cases et 10 couleurs...

▷ Il faut au moins des paires de couleurs par case



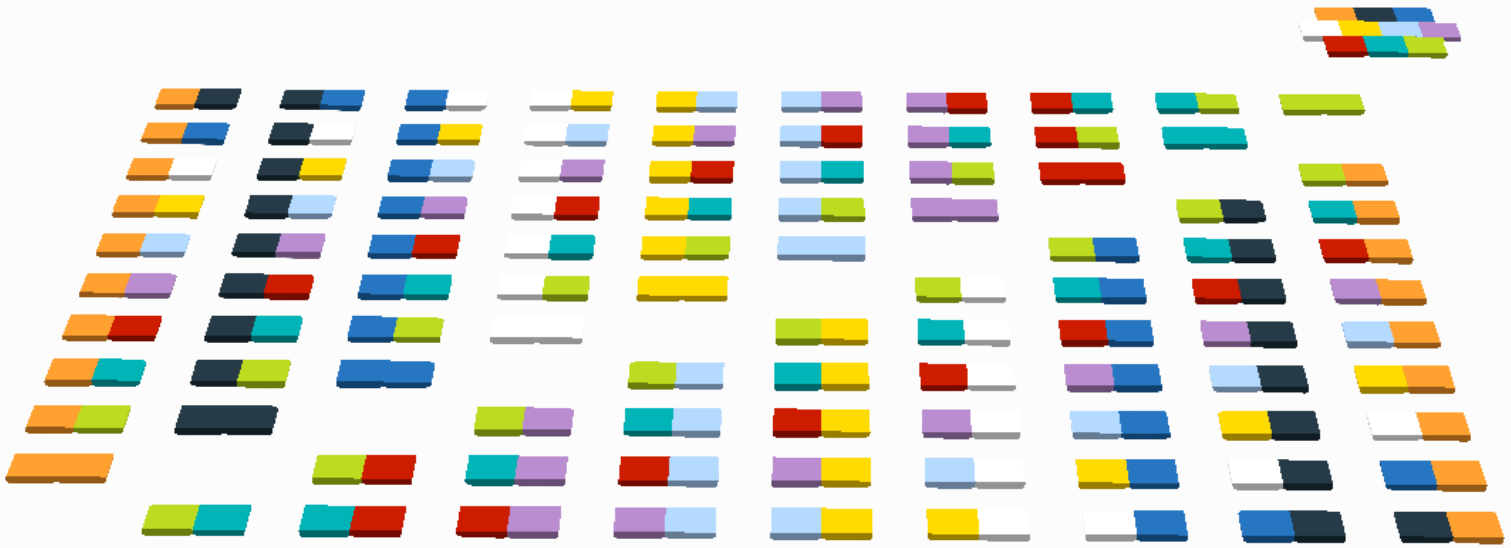
$$\frac{n(n+1)}{2} = 55$$

SYSTÈME D'ADRESSAGE

▷ Proposer un système de coloriage pour que chaque case puisse être identifiée.

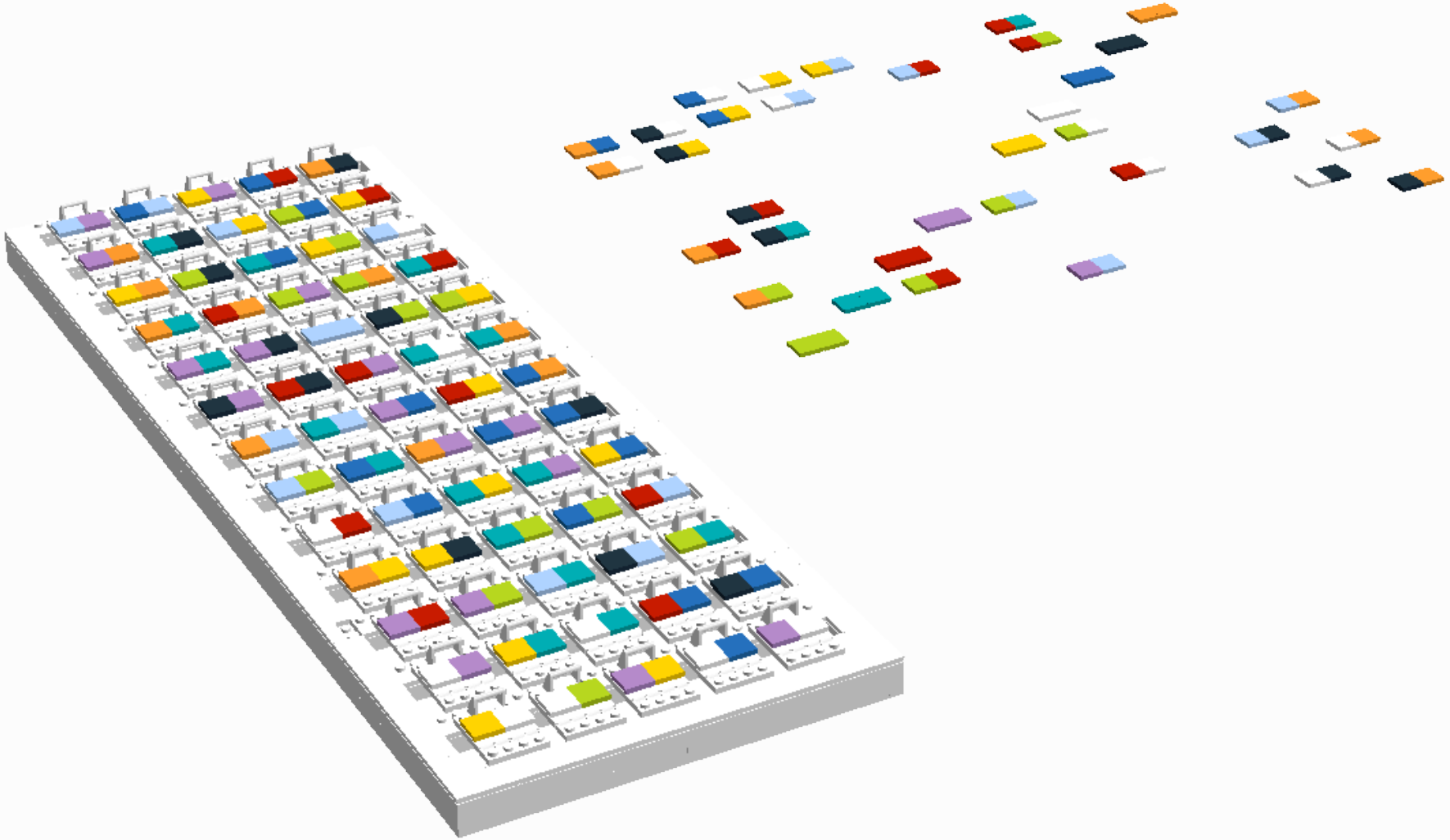
▷ Rappel : 65 cases et 10 couleurs...

▷ Il faut au moins des paires de couleurs par case et ordonnées



SYSTÈME D'ADRESSAGE

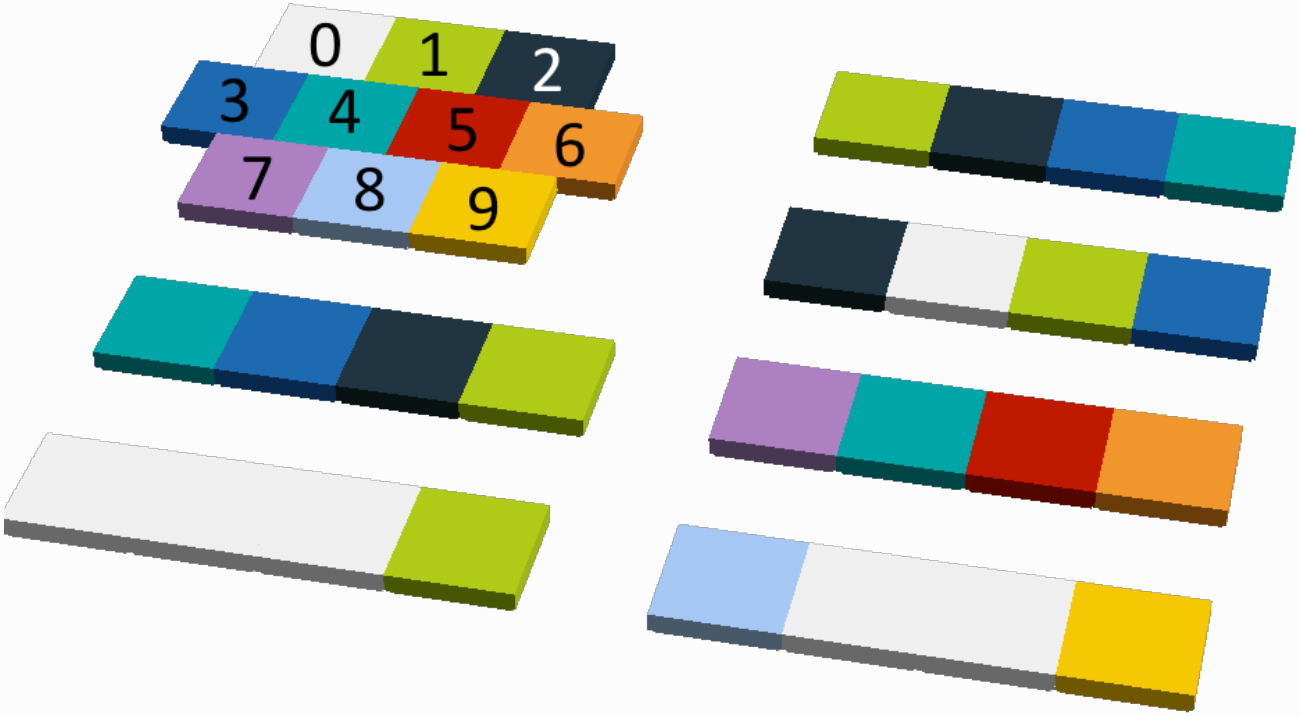
▷ Proposer un système de coloriage pour que chaque case puisse être identifiée.



Stockage d'entiers positifs de 0 à 9999

▷ Représentation positionelle

▷ $123 = 1 * 100 + 2 * 10 + 3 * 1$



▷ Notons qu'une limite en nombre de chiffres est nécessaire

Stockage d'entiers négatifs

- Peut-on stocker des entiers négatifs avec ce principe ?
  - Comment représenter le signe ?
  - Limitations induites ?

Stockage d'entiers négatifs

- ▷ Peut-on stocker des entiers négatifs avec ce principe ?
  - ▷ Comment représenter le signe ?

"Sacrifier" une des positions (e.g., la première)

  - ▷ Limitations induites ?

Stockage d'entiers négatifs

▷ Peut-on stocker des entiers négatifs avec ce principe ?

▷ Comment représenter le signe ?

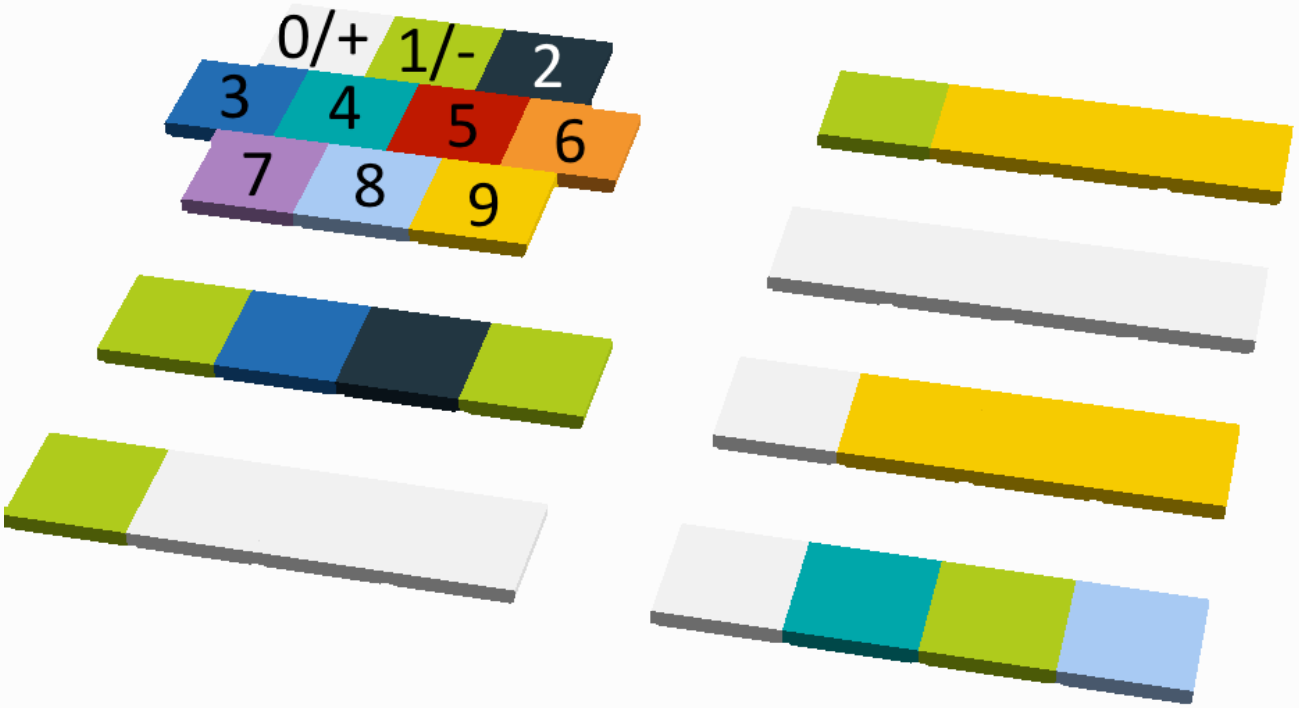
"Sacrifier" une des positions (e.g., la première)

▷ Limitations induites ?

Des nombres plus petits représentables

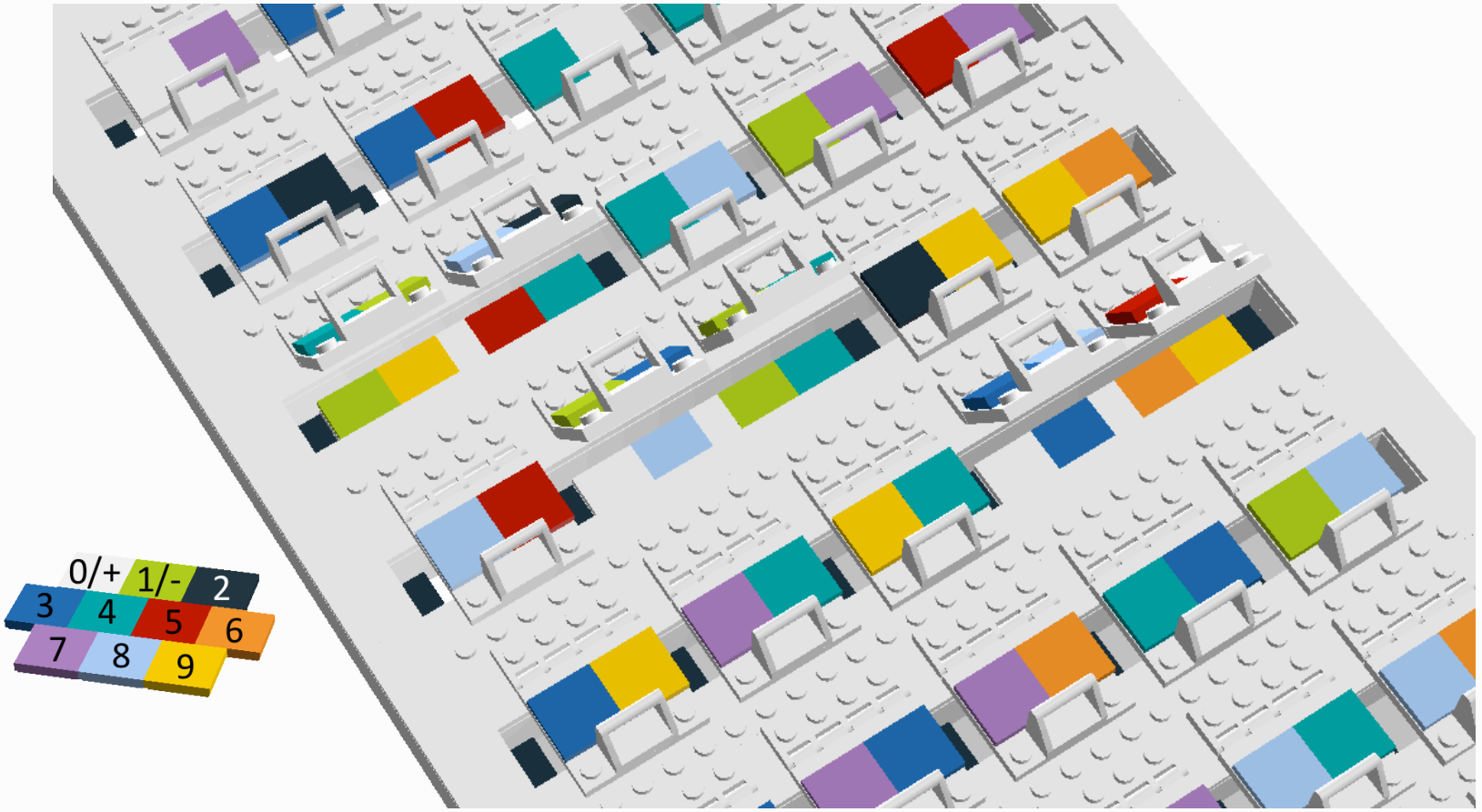
Stockage d'entiers négatifs

▷ Peut-on stocker des entiers négatifs avec ce principe ?



▷ Entiers entre -999 et +999 avec deux codages de 0

EXEMPLE DE STOCKAGE D'ENTIERS



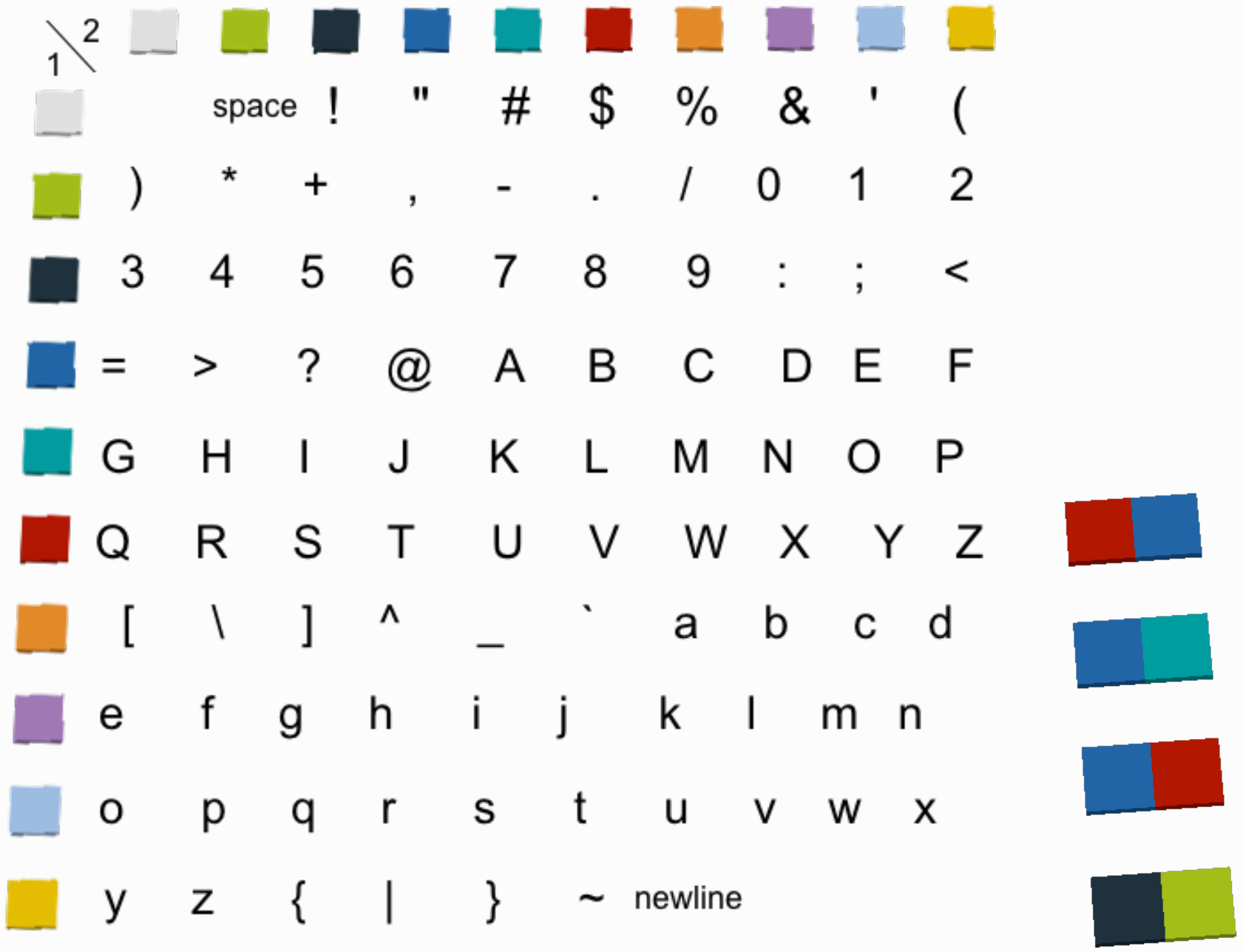
STOCKAGE DE TEXTE

- Comment stocker du texte en anglais ?
  - Base de 96 caractères

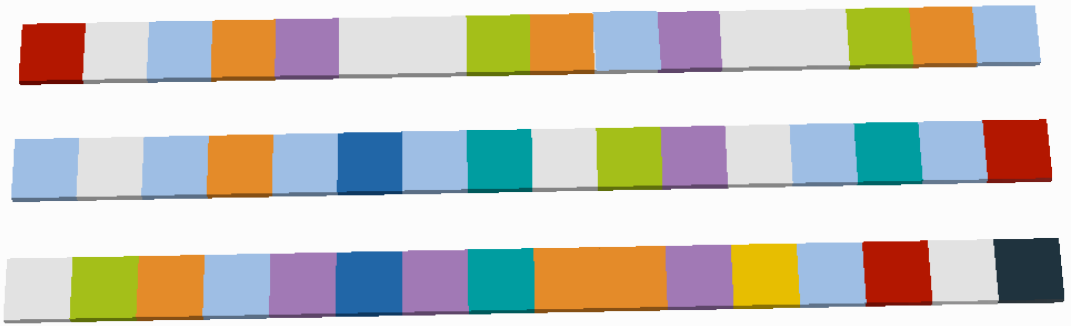
```
space ! ” # $ % & ’ ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A
B C D E F G H I J K L M N O P Q R S
T U V W X Y Z [ \ ] ^ _ ‘ a b c d e
f g h i j k l m n o p q r s t u v w
x y z { | } ~ newline
```

- Comment stocker un caractère ?

CODAGE D'UN CARACTÈRE

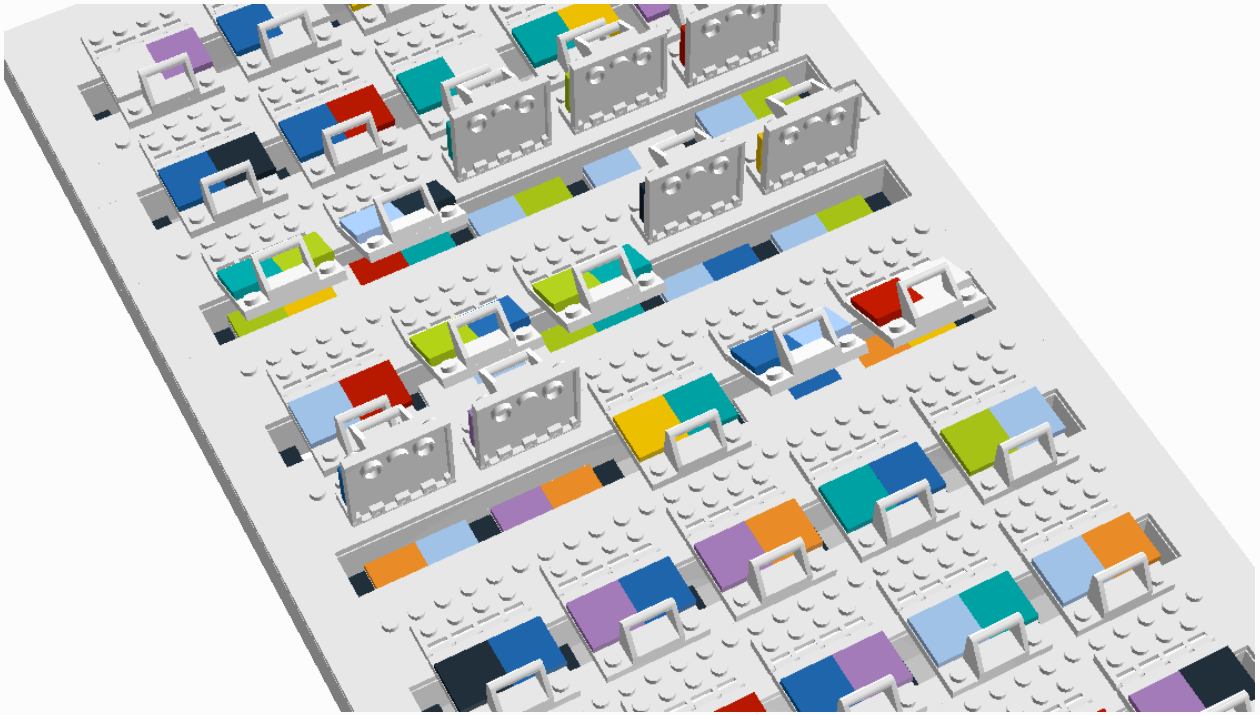


CODAGE D'UNE CHAÎNE DE CARACTÈRES

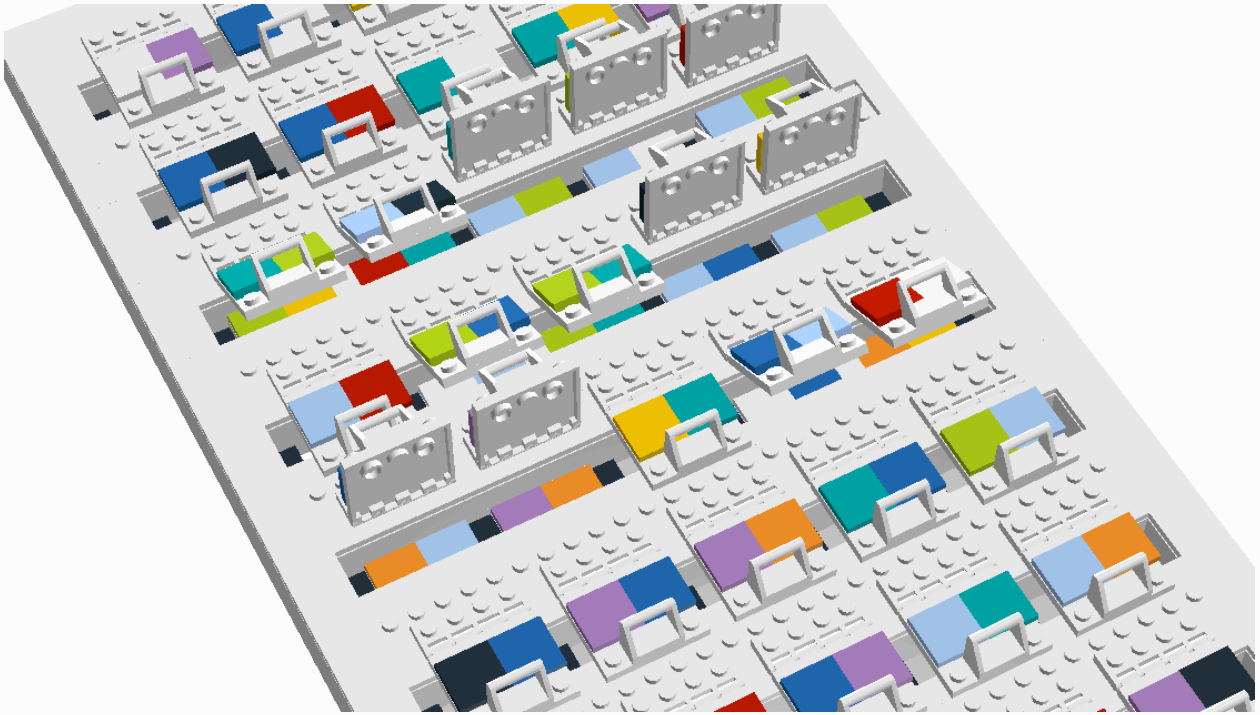


1 \ 2										
	space	!	"	#	$	%	&	'	(	
)	*	+	,	-	.	/	0	1	2	
3	4	5	6	7	8	9	:	;	<	
=	>	?	@	A	B	C	D	E	F	
G	H	I	J	K	L	M	N	O	P	
Q	R	S	T	U	V	W	X	Y	Z	
[	\	]	^	_	`	a	b	c	d	
e	f	g	h	i	j	k	l	m	n	
o	p	q	r	s	t	u	v	w	x	
y	z	{	|	}	~	newline				

PROBLÉMATIQUE DU STOCKAGE DES CHAÎNES DE CARACTÈRES



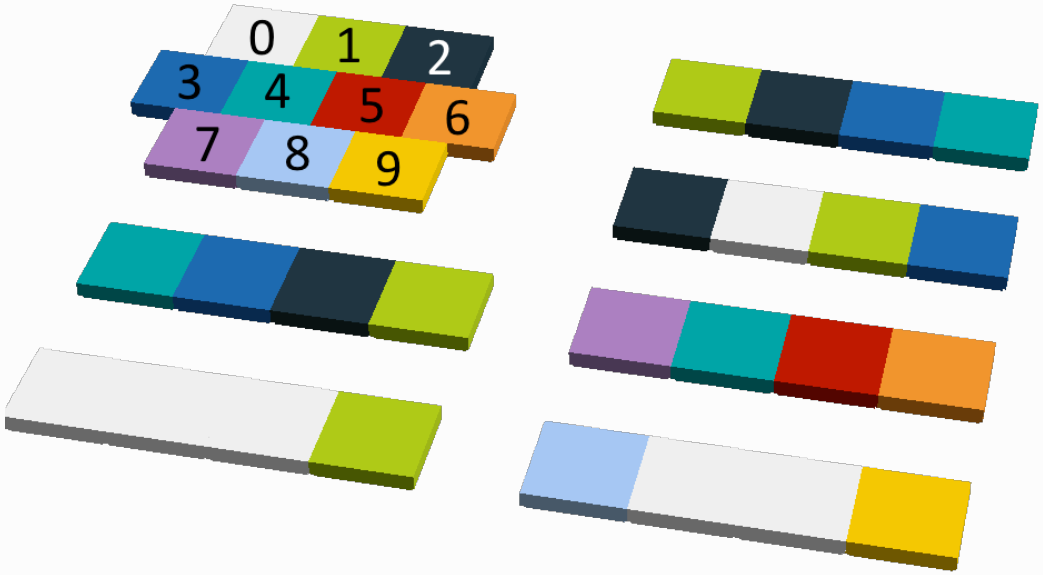
PROBLÉMATIQUE DU STOCKAGE DES CHAÎNES DE CARACTÈRES



- Convention de fin de chaîne de caractères
- Nécessité de consécutivité dans la mémoire

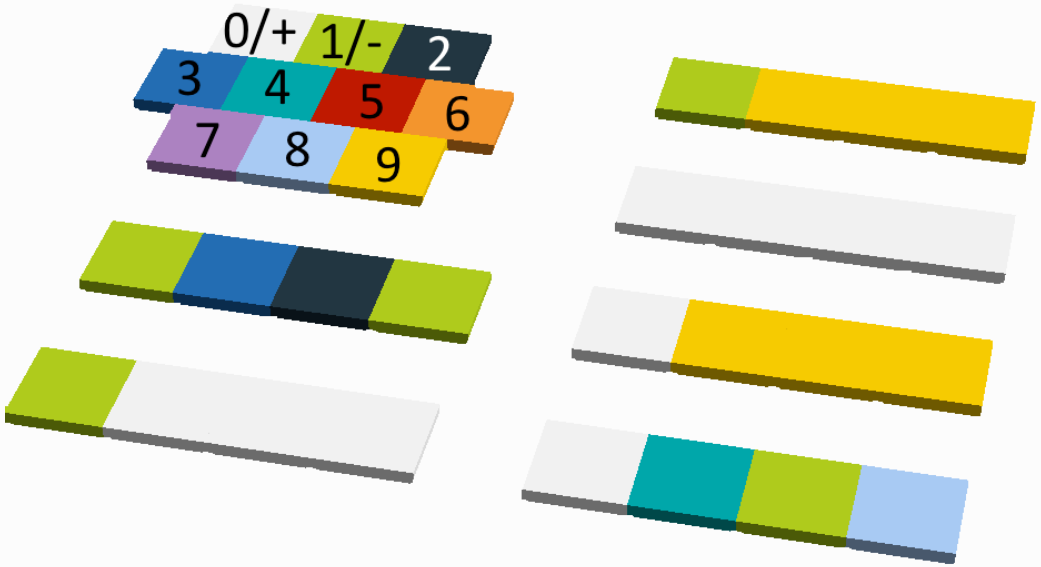
STOCKAGE DES RÉELS POSITIFS - REP. À VIRGULE FIXE

- Avec 4 couleurs
  - de 000,0 à 999,9 par pas de 0,1
  - de 00,00 à 99,99 par pas de 0,01
  - de 0,000 à 9,999 par pas de 0,001
  - de 0,000 à 0,9999 par pas de 0,0001



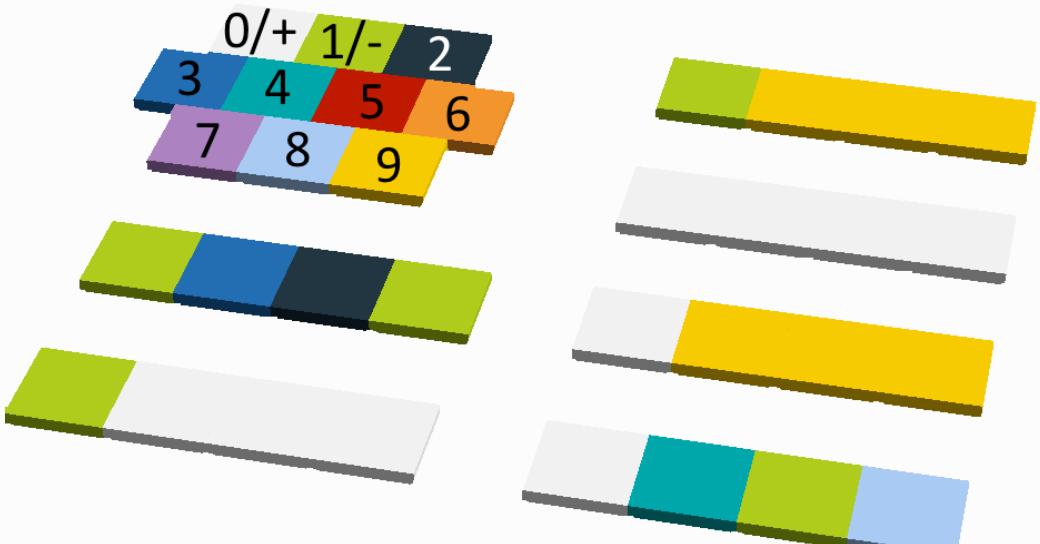
STOCKAGE DES RÉELS POSITIFS - REP. À VIRGULE FIXE

- Avec 4 couleurs
  - de ~~000,0~~ -99,9 à ~~999,9~~ 99,9 par pas de 0,1
  - de ~~00,00~~ -9,99 à ~~99,99~~ 9,99 par pas de 0,01
  - de ~~0,000~~ -0,999 à ~~9,999~~ 0,999 par pas de 0,001
  - de ~~0,000 à 0,9999~~ par pas de 0,0001



STOCKAGE DES RÉELS - REP. À VIRGULE FLOTTANTE

- 1 couleur = position de la virgule (e.g., à droite du signe)
- Avec 4 couleurs:
  - de -99 à 99 par pas de 1 [pos(,) = 0]
  - de -9,9 à +9,9 par pas de 0,1 [pos(,) = 1]
  - de -0,99 à +0,99 par pas de 0,01 [pos(,) = 2]
- Quel est l'intervalle avec pos(,) = 3 ?
- Plus de cases ?



STOCKAGE DES RÉELS - REP. À VIRGULE FLOTTANTE

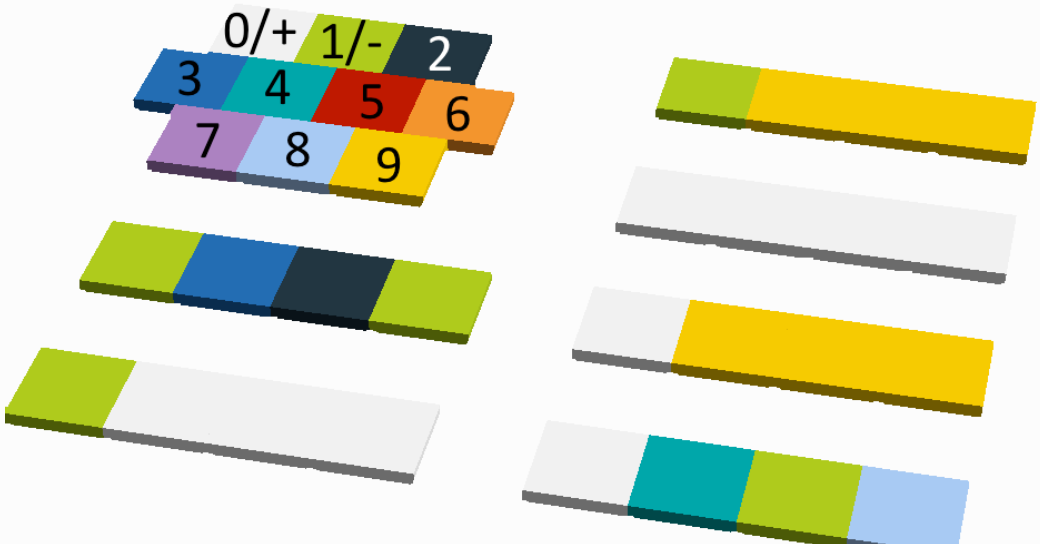
- ▷ 1 couleur = position de la virgule (e.g., à droite du signe)
- ▷ Avec 4 couleurs:
  - ▷ de -99 à 99 par pas de 1 [pos(,) = 0]
  - ▷ de -9,9 à +9,9 par pas de 0,1 [pos(,) = 1]
  - ▷ de -0,99 à +0,99 par pas de 0,01 [pos(,) = 2]
- ▷ Quel est l'intervalle avec pos(,) = 3 ?

-0,099 à -0,001

0

+0,001 à +0,099

- ▷ Plus de cases ?



STOCKAGE DES RÉELS - REP. À VIRGULE FLOTTANTE

- ▷ 1 couleur = position de la virgule (e.g., à droite du signe)
- ▷ Avec 4 couleurs:
  - ▷ de -99 à 99 par pas de 1 [pos(,) = 0]
  - ▷ de -9,9 à +9,9 par pas de 0,1 [pos(,) = 1]
  - ▷ de -0,99 à +0,99 par pas de 0,01 [pos(,) = 2]
- ▷ Quel est l'intervalle avec pos(,) = 3 ?

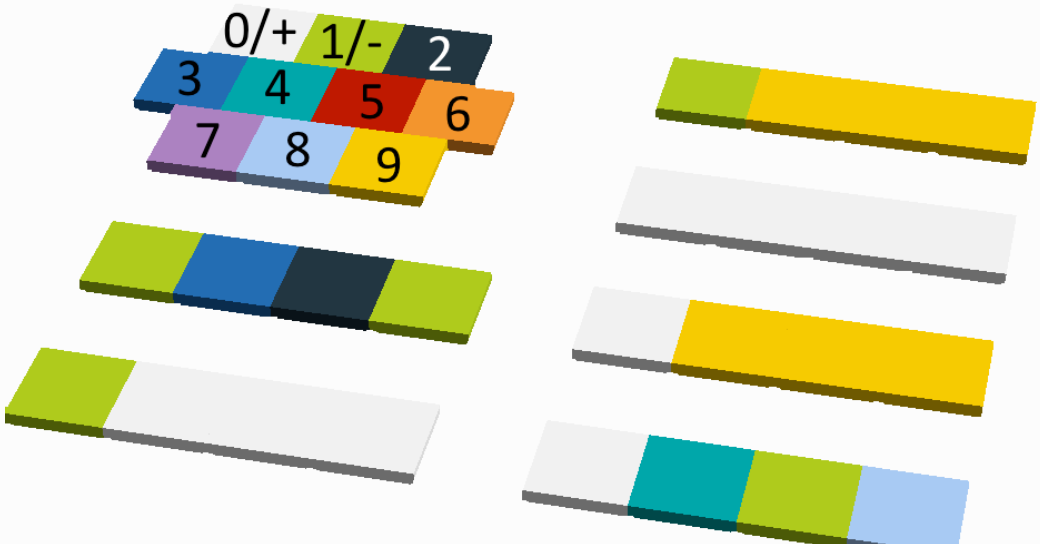
-0,099 à -0,001

0

+0,001 à +0,099

- ▷ Plus de cases ?

Plus de "précisions"



ET DANS UN VRAI ORDINATEUR ?

- Moins de couleurs ...
  - Binaire (0 ou 1) – bit
- C'est quoi le trip avec les Légos alors ?
  - C'est juste une question d'alphabet ...
  - Avec 4 bits on peut coder 16 informations

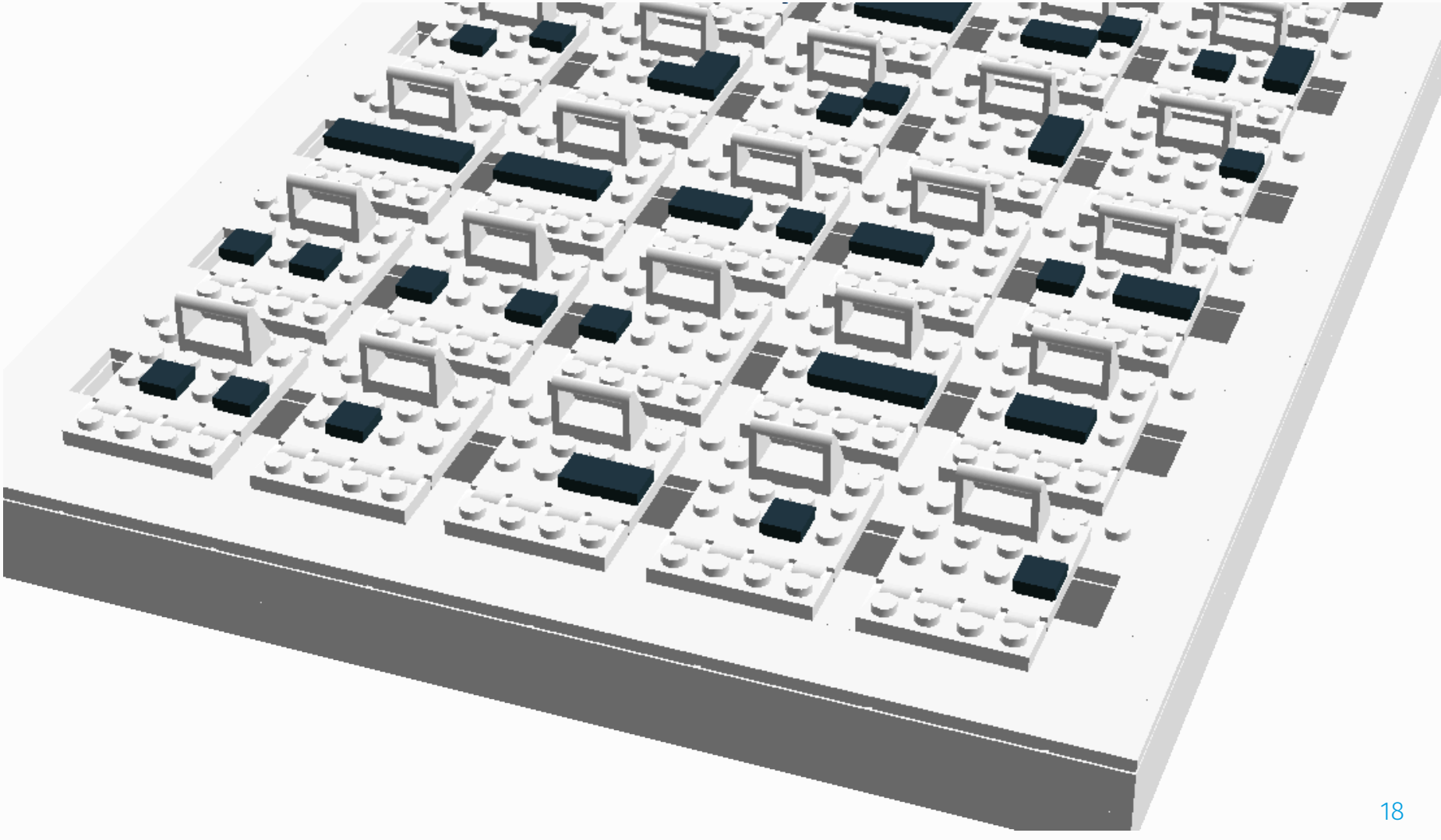
```
0000  0100  1000  1100
0001  0101  1001  1101
0010  0110  1010  1110
0011  0111  1011  1111
```

- C'est la représentation positionelle en base 2

$$1010_2 = 1*8+0*4+1*2+0*1 = 10_{10} = 1*2^3+0*2^2+1*2^1+0*2^0$$

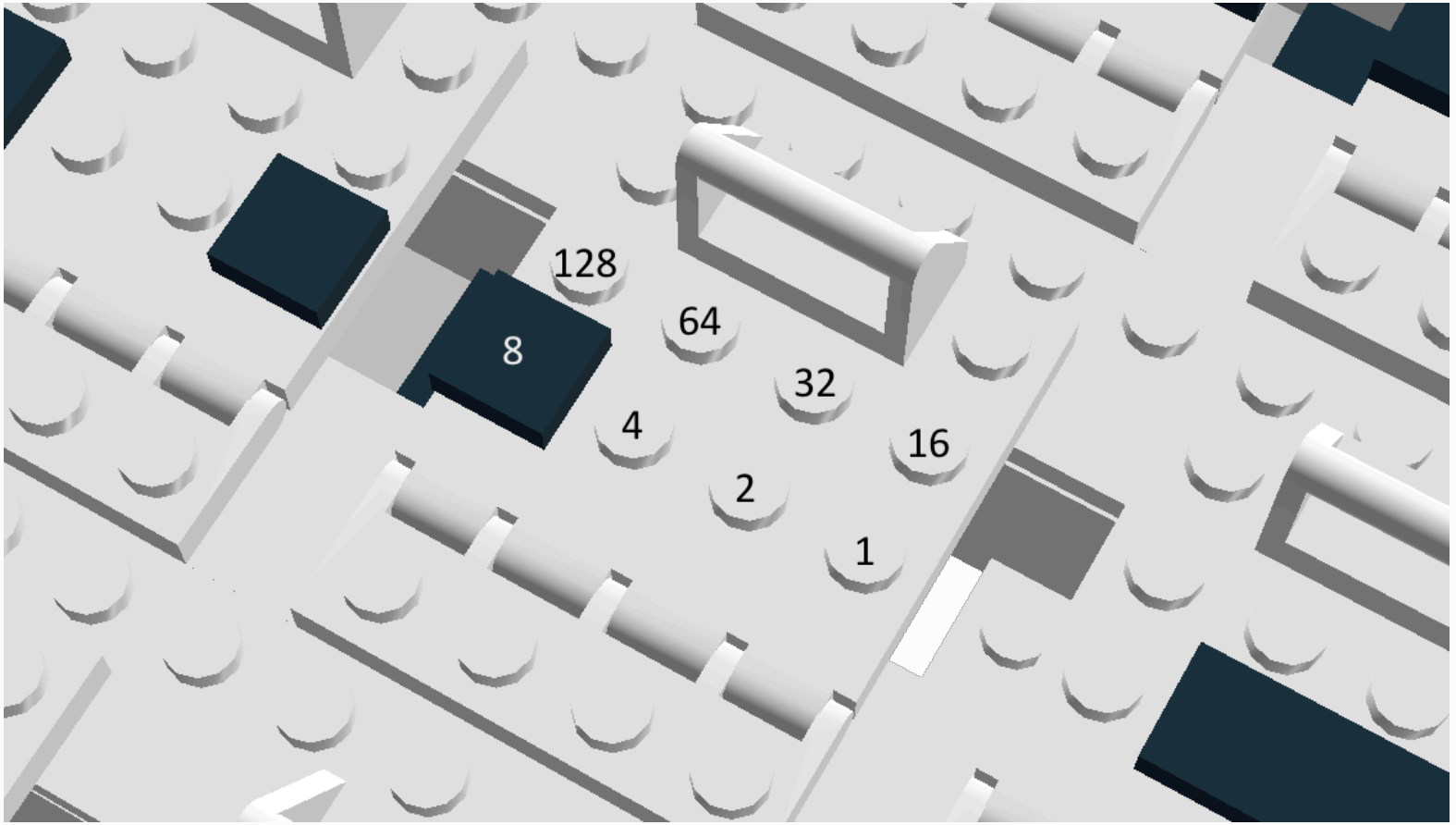
SYSTÈME D'ADRESSAGE

▷ Combinaisons de 8 tuiles (octet ou byte)



SYSTÈME D'ADRESSAGE

▷ Combinaisons de 8 tuiles (octet ou byte)

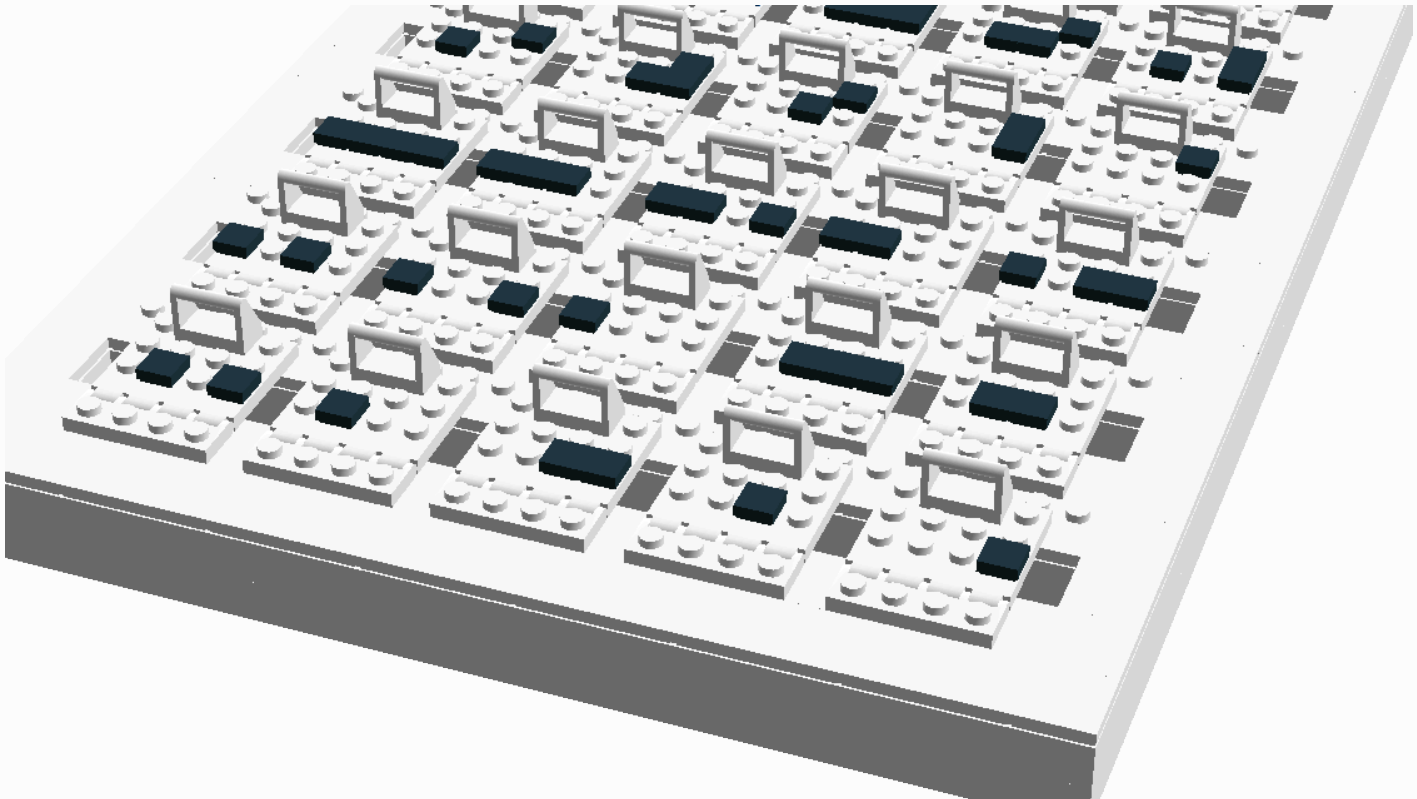


Système d'adressage

▷ Combinaisons de 8 tuiles (octet ou byte)

▷ Consécutivité des adresses (intérêt ?)

▷ Une adresse = un entier positif = un indice dans un grand tableau d'octets



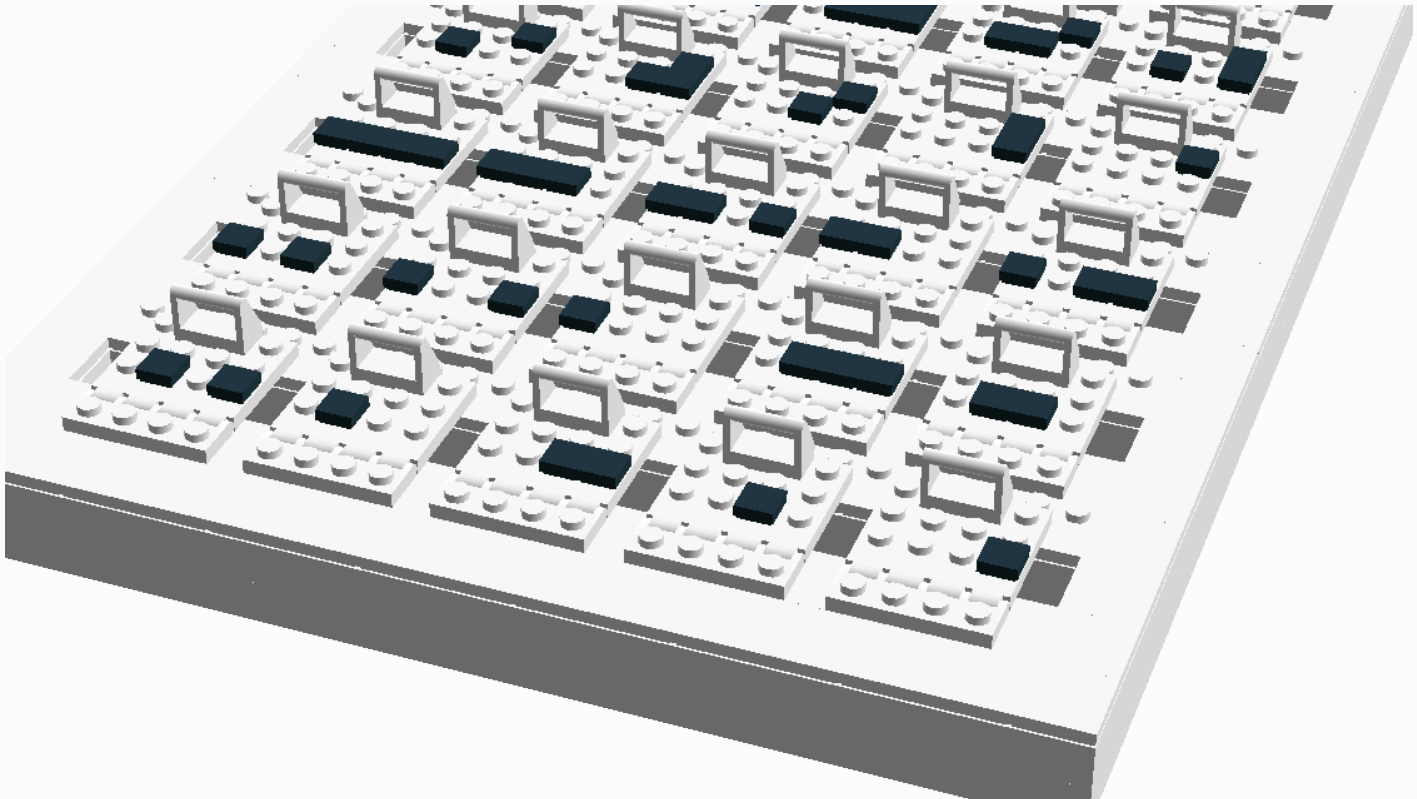
Système d'adressage

▷ Combinaisons de 8 tuiles (octet ou byte)

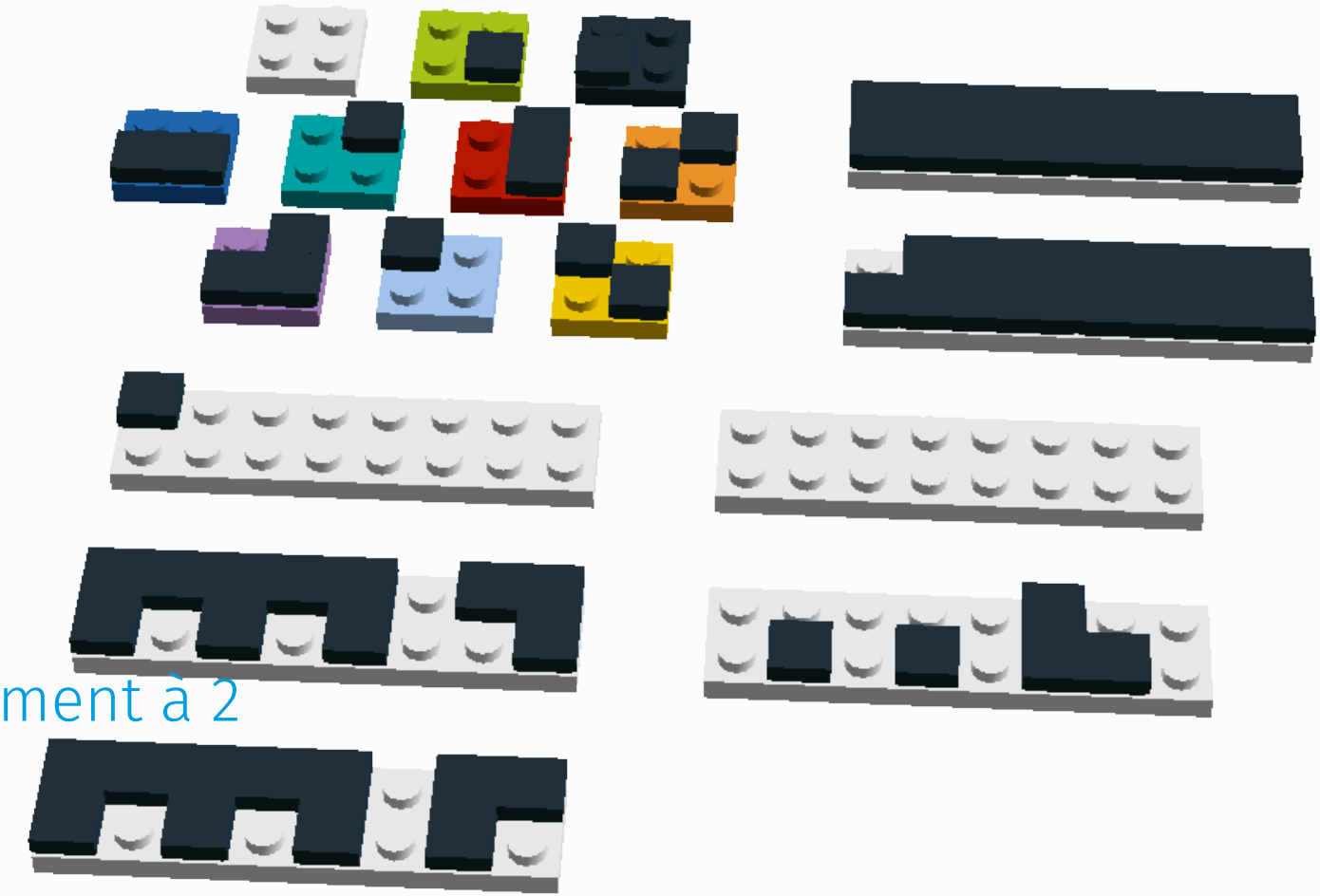
▷ Consécutivité des adresses (intérêt ?)

Cela permet de l'arithmétique sur les adresses

▷ Une adresse = un entier positif = un indice dans un grand tableau d'octets



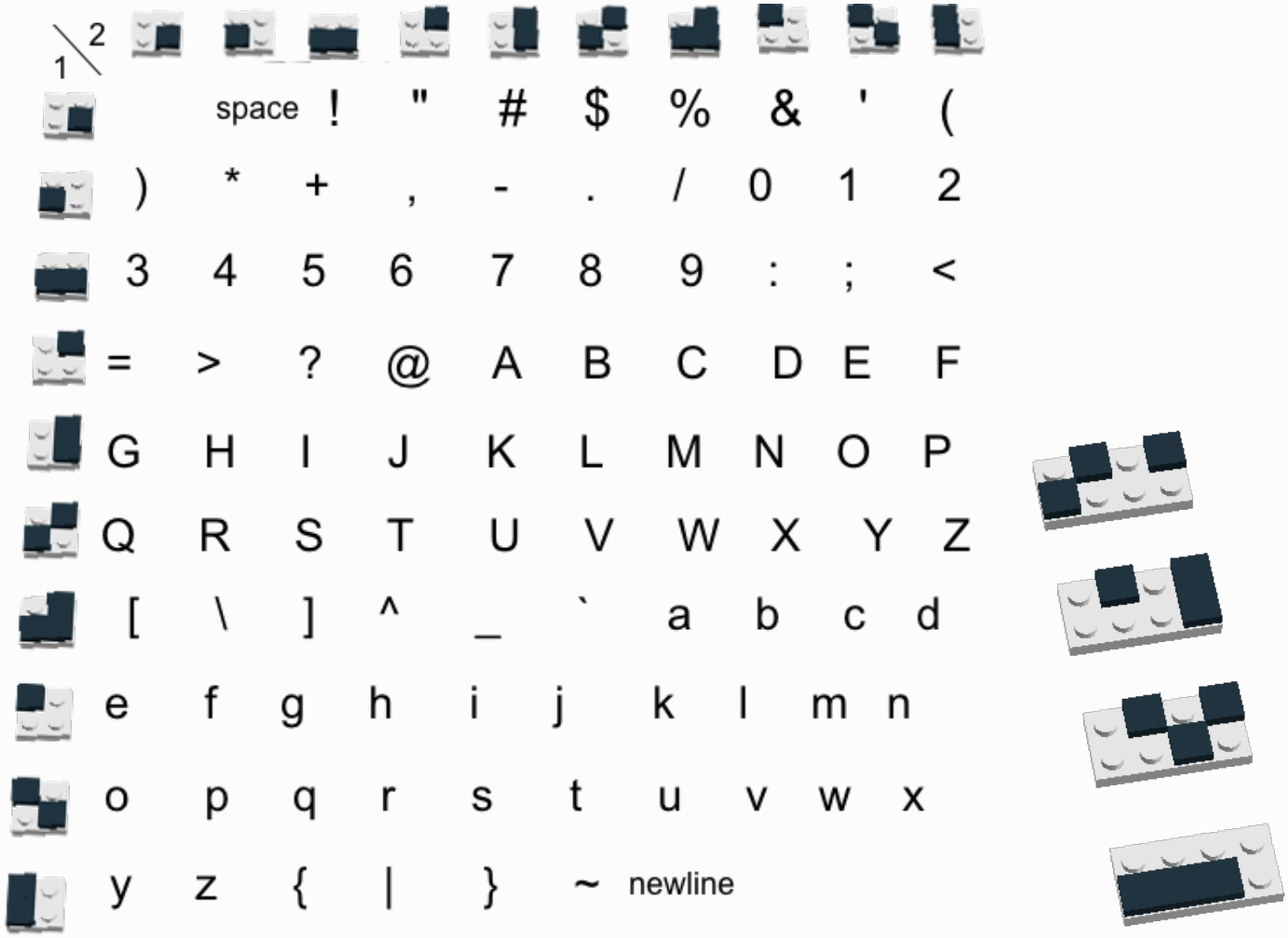
Représentation des entiers sur k bits



- Complément à 2

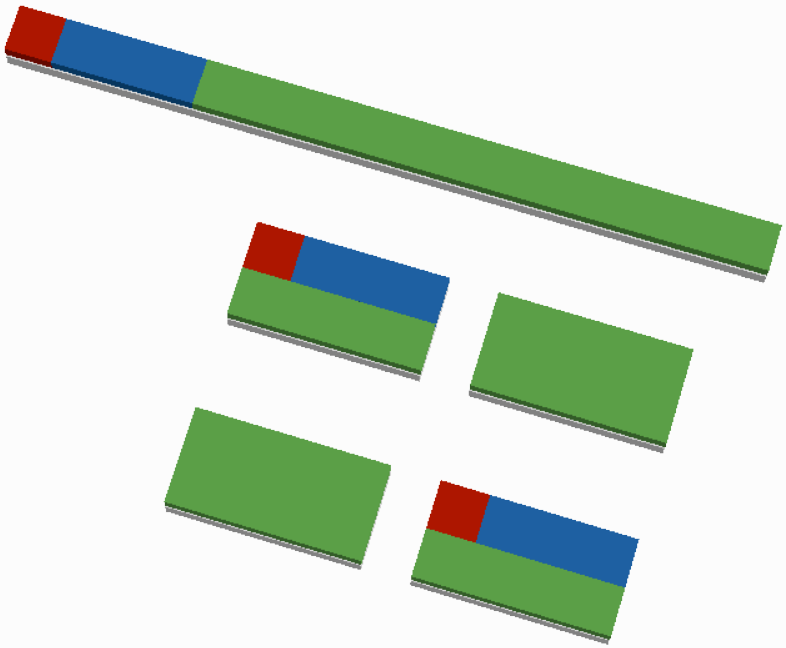
CODAGE D'UN CARACTÈRE

▷ Caractère = entier positif = indice



Représentation à virgule flottante (simplifiée)

- 1 bit pour le signe
- Quelques bits pour la position de la virgule
- Représentation virgule fixe + décalage
- Dans quel ordre les stocker ?
  - Little et Big Endian



RETOUR AU C

LA MÉMOIRE

▷ On manipule toujours de l'espace mémoire à l'aide d'adresse

▷ Chaque adresse correspond à une case mémoire d'un octet (8 bits) = entier positif

LES TYPES EN C

▷ Type = taille de zone mémoire + interprétation (signé ou non, nombre entier/flottant, caractère ...)

▷ Possiblement dépendant de la machine

▷ Seules contraintes imposées en C : ordre des tailles en mémoire

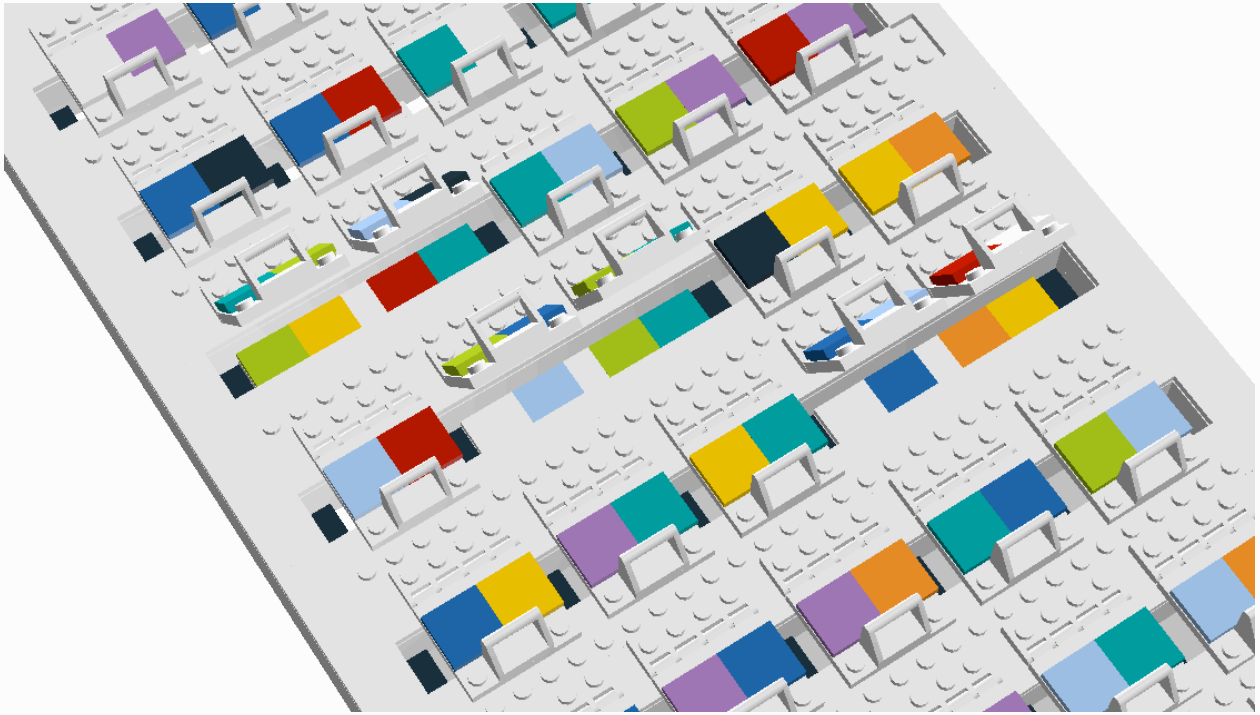
  ▷ caractere $\leq$ petit entier $\leq$ entier $\leq$ grand entier

LES VARIABLES EN C

▷ Variable = adresse + type + valeur

▷ Pour le programmeur, on lui associe un libellé compréhensible (utile que pour lui car remplacé par l'adresse lors de la compilation)

LES VARIABLES EN C



Adresse	Type	Valeur
	`entier`	`-954`
	`caractere`	`'`
	`entier positif`	`369`

DOGGY BAG

TO TAKE AWAY ...

- Adresse = entier positif = case mémoire d'un octet
- Type = taille de zone mémoire + interprétation
- Variable = adresse + type + valeur
- Tout est binaire et est une question d'interprétation

Questions?

PASSER DU PYTHON AU C

Initiation à la programmation C

---

uf-info.ue.init-prog-c@diff.u-bordeaux.fr

université de BORDEAUX

GRANDES DIFFÉRENCES

SYNTHÈSE DES DIFFÉRENCES

Langage C	Langage Python
Langage impératif	Langage multi-paradigmes (dont impératif)
Le code source est compilé une fois, exécuté n fois	Le code source est lu, analysé et exécuté n fois
Basé sur une syntaxe de blocs délimités par des accolades	Basé sur le respect d'une indentation
Typage statique	Typage dynamique et automatique
Gestion de la mémoire est manuelle	Gestion de la mémoire est automatique

MONO VS MULTI-PARADIGMES

▷ Un paradigme correspond plus ou moins à un style de programmation: e.g. les programmations impérative, fonctionnelle ou encore orienté-objets

▷ Il est lié au moyen d'abstraction principal utilisé

MONO VS MULTI-PARADIGMES

▷ Un paradigme correspond plus ou moins à un style de programmation: e.g. les programmations impérative, fonctionnelle ou encore orienté-objets

▷ Il est lié au moyen d'abstraction principal utilisé

  ▷ La procédure pour la plupart des langages impératifs

"Le programme décrit les opérations en séquences d'instructions exécutées par l'ordinateur pour modifier l'état du programme (modèle des machines à états)"

MONO VS MULTI-PARADIGMES

▷ Un paradigme correspond plus ou moins à un style de programmation: e.g. les programmations impérative, fonctionnelle ou encore orienté-objets

▷ Il est lié au moyen d'abstraction principal utilisé

  ▷ La procédure pour la plupart des langages impératifs

  ▷ La fonction (i.e. une procédure ne pouvant pas modifier l'environnement à partir duquel elle est appelée – sans effet de bord) pour les langages fonctionnels

"Le programme est une application, au sens mathématique, qui ne donne qu'un seul résultat pour chaque ensemble de valeurs en entrée à l'aide d'un emboîtement de fonctions qui agissent comme des "boîtes noires" que l'on peut imbriquer les unes dans les autres" - à voir en Semestre 3

MONO VS MULTI-PARADIGMES

▷ Un paradigme correspond plus ou moins à un style de programmation: e.g. les programmations impérative, fonctionnelle ou encore orienté-objets

▷ Il est lié au moyen d'abstraction principal utilisé

  ▷ La procédure pour la plupart des langages impératifs

  ▷ La fonction pour les langages fonctionnels

  ▷ L'objet (i.e. un concept, une idée ou toute entité du monde physique) pour les langages orienté-objets.

"Le programme consiste en la définition et l'assemblage de briques logicielles appelées objets possédant une structure interne et un comportement, et pouvant interagir avec d'autres objets" - à voir en Semestre 5

INTERPRÉTATION VS COMPILATION

▷ Python est un langage interprété / interactif

▷ Le programmeur dialogue en permanence avec le système via une boucle d'interaction qui est chargée de lire, analyser et exécuter les lignes de texte tapées par le programmeur

▷ C'est un programme auxiliaire - l'interpréteur - qui traduit au fur et à mesure les instructions du programme

INTERPRÉTATION VS COMPILATION

▷ Python est un langage interprété / interactif

▷ Pour démarrer le système Python, il suffit de taper python dans un shell

```
$ python
Python 2.7.12rc1 (default, Jun 13 2016, 09:20:59)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> x=4
>>> if x > 3:
...     y=2
... else:
...     y=5
...
>>> y
2
>>> x
4
>>>
```

INTERPRÉTATION VS COMPILATION

▷ C est un langage compilé

▷ Le programme va être traduit une fois pour toutes par un programme annexe, appelé compilateur, afin de générer un nouveau fichier qui sera autonome (i.e. exécutable)

▷ Toute modification du fichier source nécessite de re-compiler le programme pour que les modifications prennent effet

INDENTATION VS BLOC

▷ En python, la lecture d'un programme est basée sur l'utilisation de l'indentation

▷ Python utilise le passage à la ligne pour séparer les instructions, deux points et l'indentation pour séparer les blocs de code

▷ En C, les instructions sont séparées par des points-virgules et les blocs sont encadrés par des accolades

TYPAGE DYNAMIQUE VS STATIQUE

▷ En python, une variable a un type implicite (déterminé par l'interpréteur) qui dépend de sa valeur actuelle et peut donc en changer (dynamique)

▷ En C, le type de chaque variable doit être connu du compilateur (donc avant l'exécution du programme). On parle de typage statique. C'est au programmeur d'indiquer le type, on parle de typage explicite

TO DECLARE OR NOT DECLARE

▷ En C, tout ce qui est utilisé doit être au préalable déclaré (e.g. les variables, les fonctions) car connu à la compilation

▷ En python, si on utilise quelque chose de non défini, l'erreur se produit à l'exécution

```
$ python
Python 2.7.12rc1 (default, Jun 13 2016, 09:20:59)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> z
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'z' is not defined
>>> f(8)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'f' is not defined
>>>
```

GESTION DE LA MÉMOIRE

▷ En python la gestion de la mémoire est automatique: le programmeur ne s'en soucie pas

▷ En C, le programmeur doit gérer manuellement la mémoire en demandant explicitement de l'espace pour stocker une variable et, dans certains cas, en demandant explicitement de libérer cette dernière lorsqu'elle n'est plus utile - vu au Semestre 3

PROGRAMME PRINCIPAL OBLIGATOIRE

▷ En python, toutes les instructions ayant l'indentation la plus basse sont exécutées dans l'ordre d'apparition par défaut

▷ En C, un programme doit comporter un point d'entrée unique sous la forme d'une fonction nommée main

```
...

int main(void){
... CODE EXECUTE
}
...
```

ETUDES DE CAS

EXEMPLE 1

▷ $f : x \to x^2 + x + 1$

```
def f(x):
    return x*x+x+1
```

- ▷ Que désigne `x` ?
- ▷ Où commence/finit une instruction ?
- ▷ Où commence/finit une fonction ?

EXEMPLE 1

▷ $f : x \to x^2 + x + 1$

```
int f (int x)
{
    return x*x+x+1;
}
```

- ▷ Que désigne `x` ?
- ▷ Où commence/finit une instruction ?
- ▷ Où commence/finit une fonction ?

EXEMPLE 2

▷ pair : x → est_pair(x)?

```
def pair(n):
    p=0
    i=1
    if n%2==0:
        return p
    else :
        return i
```

▷ Que désignent `n`, `i` et `p` ?

▷ Où commence/finit une instruction ?

▷ Où commencent/finissent le `if` et le `else` ?

▷ Où commence/finit une fonction ?

EXEMPLE 2

▷ pair : x → est_pair(x)?

```
int pair(int n){
    int p=0;
    int i=1;
    if(n%2==0){
        return p;
    }else{
        return i;
    }
}
```

▷ Que désignent `n`, `i` et `p` ?

▷ Où commence/finit une instruction ?

▷ Où commencent/finissent le `if` et le `else` ?

▷ Où commence/finit une fonction ?

EXERCICES DE ”TRADUCTION”

EXERCICE

```
age = 12
if age < 18:
    prix = 10
else :
    prix = 5
```

CORRIGÉ

```
age = 12
if age < 18:
    prix = 10
else :
    prix = 5
```

```
int main(void){
    int prix = 0;
    int age = 12;
    if(age < 18){
        prix = 10;
    }else{
        prix = 5;
    }
    return EXIT_SUCCESS;
}
```

EXERCICE

```
age = 12
if age <= 5:
    prix = 0
else :
    if age < 18:
        prix = 10
    else :
        prix = 5
```

CORRIGÉ

```
age = 12
if age <= 5:
    prix = 0
else :
    if age < 18:
        prix = 10
    else :
        prix = 5
```

```
int main(void){
    int age = 12;
    int prix = 0;
    if(age <= 5){
        prix = 0;
    }else{
        if(age < 18){
            prix = 10;
        }else{
            prix = 5;
        }
    }
    return EXIT_SUCCESS;
}
```

EXERCICE

```
x = 3
if x % 2 == 0:
    y = x - 1
else :
    y = x + 1
x = x + 1
```

CORRIGÉ

```
x = 3
if x % 2 == 0:
    y = x - 1
else :
    y = x + 1
x = x + 1
```

```
int main(void){
    int x = 3;
    int y = 0;
    if(x % 2 == 0){
        y = x - 1;
    }else{
        y = x + 1;
    }
    x = x + 1;
    return EXIT_SUCCESS;
}
```

EXERCICE

```
x = 3
if x % 2 == 0:
    y = x - 1
else :
    y = x + 1
    x = x + 1
```

CORRIGÉ

```
x = 3
if x % 2 == 0:
    y = x - 1
else :
    y = x + 1
    x = x + 1
```

```
int main(void){
    int x = 3;
    int y = 0;
    if(x % 2 == 0){
        y = x - 1;
    }else{
        y = x + 1;
        x = x + 1;
    }
    return EXIT_SUCCESS;
}
```

EXERCICE

```
def f(x):
    if x % 2 == 0:
        y = x - 1
    else :
        y = x + 1
    return y

z = f(2)
```

CORRIGÉ

```python
def f(x):
    if x % 2 == 0:
        y = x - 1
    else :
        y = x + 1
    return y

z = f(2)
```

```c
int f(int x){
    int y = 0;
    if(x % 2 == 0){
        y = x - 1;
    }else{
        y = x + 1;
    }
    return y;
}
int main(void){
    int z = 0;
    z = f(2);
    return EXIT_SUCCESS;
}
```

DOGGY BAG

TO TAKE AWAY ...

- ▷ Le programmeur doit être explicite en C
- ▷ Le code source est compilé une fois, exécuté n fois
- ▷ Le code en C est basé sur une syntaxe de blocs délimités par des accolades
- ▷ Toute instruction est terminée par un ';'

Questions?

MANIPULATION DES TYPES ET OPÉRATEURS

Initiation à la programmation C

---

uf-info.ue.init-prog-c@diff.u-bordeaux.fr

université de BORDEAUX

LES TYPES EN C

RAPPELS

- Type = taille de zone mémoire + interprétation (signé ou non, nombre entier/flottant, caractère ...)
- Possiblement dépendant de la machine
- Seules contraintes imposées en C : ordre des tailles en mémoire
  - caractere $\leq$ petit entier $\leq$ entier $\leq$ grand entier

MOTS CLÉS

Mot clé	Signification
char	Caractère
unsigned char	Caractère non signé
short int	Entier court
unsigned short int	Entier court non signé
int	Entier
unsigned int	Entier non signé
long int	Entier long
unsigned long int	Entier long non signé
float	Flottant (réel)
double	Flottant double
bool*	Booléen

*C99

TAILLE D'UN TYPE

▷ L'opérateur **sizeof** permet d'avoir le nombre d'octets occupés par un type donné à la compilation

```
int main(void){
  printf("Size of short: %d \n", sizeof(short));
  printf("Size of int: %d \n", sizeof(int));
  printf("Size of long: %d \n", sizeof(long));
  printf("Size of float: %d \n", sizeof(float));
  printf("Size of double: %d \n", sizeof(double));
  printf("Size of char: %d \n", sizeof(char));
  return EXIT_SUCCESS ;
}
```

LIMITS.H SUR UNE MACHINE 32 BITS

Macro	Valeur exemple	Description
CHAR_BIT	8	Number of bits in a byte
SCHAR_MIN	-128	Min value for a signed char
SCHAR_MAX	127	Max value for a signed char
UCHAR_MAX	255	Max value for an unsigned char
CHAR_MIN	0	Min value for type char
CHAR_MAX	127	Max value for type char
SHRT_MIN	-32768	Min value for a short int
SHRT_MAX	+32767	Max value for a short int
USHRT_MAX	65535	Max value for an unsigned short int
INT_MIN	-2147483648	Min value for an int
INT_MAX	+2147483647	Max value for an int
UINT_MAX	4294967295	Max value for an unsigned int
LONG_MIN	-2147483648	Min value for a long int
LONG_MAX	+2147483647	Max value for a long int
ULONG_MAX	4294967295	Max value for an unsigned long int

Représentations des entiers

- En décimal (base 10)
  - `1234`
- En hexadécimal (base 16)
  - `0x4D2` ou `0X4d2`
- En octale (base 8)
  - `02322`
  - en math, `00012=12`, pas en C
- Pas de représentation binaire

OPÉRATEURS SUR LES ENTIERS

- Addition: `9+4` (13)
- Soustraction: `9-4` (5)
- Multiplication: `9*4` (36)
- Quotient de la division entière: `9/4` (2)
  - Contrairement à Python où il existe deux symboles `/` et `//`, c'est le type des opérandes qui conditionnent le type de division (entière ou réelle) en C
- Reste de la division entière: `9%4` (1)

Problème de débordement

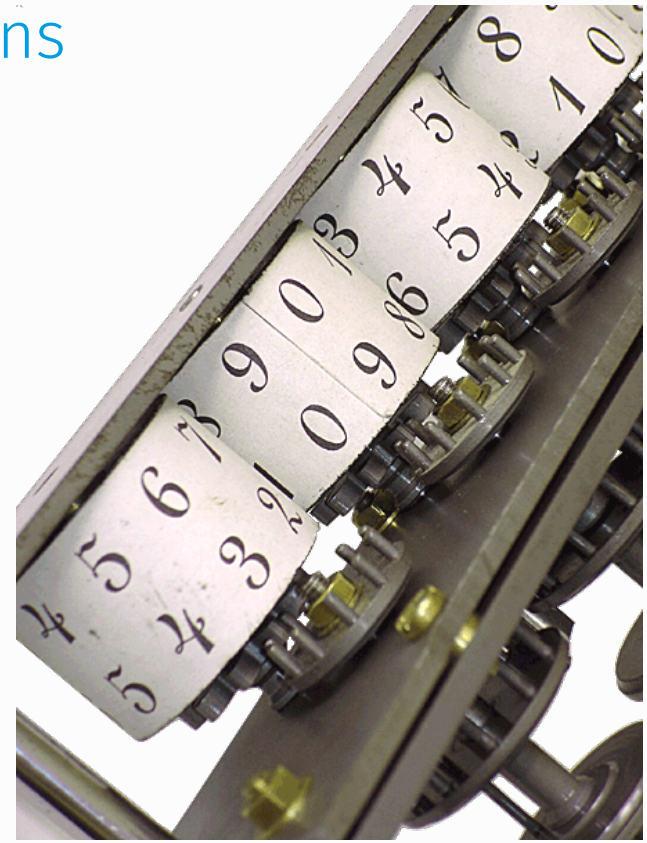
▷ Aucune vérification

▷ Après la valeur représentée par le plus grand code, on trouve la valeur représentée par le plus petit

▷ On peut imaginer les informations représentables sur un rouleau

▷ Peut générer des boucles infinies inattendues

▷ Attendre qu'un `unsigned int` soit strictement inférieur à 0 peut être long ...



LES RÉELS

- Calcul en virgule flottante
    - Approximations (ne pas utiliser pour des calculs exacts, e.g., finances)
- `float`: réel simple précision
- `double`: réel double précision

Représentations des réels

- `12.34` ou `1234.` ou `.1234`
- Notation mantisse $*$ 10^{exposant}
  - `1723.68 = 1.72368e3 = 17.2368E2`
  - `0.015 = 1.5e-2`
- Par défaut, les constantes sont `double`
  - `245.45f = 245.45F` $\rightarrow$ `float`
  - `245.45 = 245.45l = 245.45L` $\rightarrow$ `double`
- Environ 7/15 chiffres significatifs pour les `float/double`

OPÉRATEURS SUR LES RÉELS

- `+ - *` comme pour les entiers
- `/` donne une division réelle
- Si un opérateur a des opérandes de différents types, les valeurs des opérandes sont converties dans un type commun
  - Des types plus "petits" vers les types plus "larges"
  - `2+/-/*3.5` ⇒ `2.0+/-/*3.5`
  - `6/1.5` ⇒ `6.0/1.5`
  - `8.4/2` ⇒ `8.4/2.0`

Problème de précision

▷ Se méfier beaucoup des `float`

  ▷ Une simple somme de `1000` `float` à `0.1f` correspond à `99.999046`

▷ Se méfier quand même des `double`

  ▷ Une simple somme de `10 000 000` `double` à `0.1L` correspond à `999999.999839`

LES CARACTÈRES

- ▷ On interprète les `char` comme des codes de caractères
  - ▷ Pas de problème entre `0` et `127` (commun à tous les encodages)
  - ▷ Au-delà, dépend de l'encodage du système (à éviter)
- ▷ On désigne le code du caractère `x` par `'x'`
- ▷ On peut utiliser les opérateurs entiers sur ces codes
  - ▷ Par exemple, `'a'+1` vaut `'b'`

CARACTÈRES SPÉCIAUX

▷ Attention à ne pas confondre un caractère et la valeur d'un chiffre

  ▷ Le caractère `'9'` n'a pas du tout comme code `9` (mais `57`).

▷ Certains caractères sont spéciaux

  ▷ `'\n'` : saut de ligne

  ▷ `'\r'` : retour au début de ligne

  ▷ `'\t'` : tabulation

  ▷ `'\\'` : le caractère `\`

  ▷ `'\"'` : le caractère `"`

CARACTÈRES ACCENTUÉS

▷ Codes supérieurs à 127

▷ Posent des problèmes de portabilité et d'encodage

▷ Ne marche plus si on change de système et/ou de codage

▷ Nécessite une gestion fine de la part du programmeur et des éventuelles adaptations des bibliothèques standards de manipulation de chaînes de caractères

LES VARIABLES EN C

RAPPELS SUR LES VARIABLES EN C

▷ Variable = adresse + type + valeur

▷ Pour le programmeur, on lui associe un libellé compréhensible (utile que pour lui car remplacé par l'adresse lors de la compilation)

LES VARIABLES EN C

- Identificateur: `[_a-zA-Z][_a-zA-Z0-9]*`
  - `hello_89` `__tmp` `foo` `bar`
- Eviter ceux ressemblant à des mots-clés (e.g., `new, class`)
- Doivent être déclarées et initialisées[†] avant d'être utilisées
  - Déclaration: `int a=3;`
  - Déclarations multiples: `float f1=0.3,f2=0.0;`

[†] C'est une recommandation

AFFECTATION D'UNE VARIABLE EN C

▷ L'accès à la valeur d'une variable se fait en utilisant son nom directement

▷ La valeur d'une variable peut être modifiée via l'opération d'affectation en utilisant l'opérateur =

```
int a=0, b=0;
a=2+3;
b=a+2;
```

▷ Lors d'une affectation, la donnée à droite du signe d'égalité est convertie dans le type à gauche du signe d'égalité avec possible perte de précision

- ▷ `double a=2;` ⇒ `double a=2.0;`
- ▷ `int a=2.45;` ⇒ arrondi à `int a=2;`

CONVERSION DE TYPES

▷ Possibilité d'interpréter explicitement différemment la valeur d'une variable

▷ Notion de conversion / cast

  ▷ variable = (nouveau_type) ...

```
char a = 'A';
int b = 0;
b = (int) a+1; // b vaut 66
```

▷ ⚠ Il faut être sûr de ce que l'on fait

PRINTF

PRINTF

▷ La fonction `printf` permet l'écriture formatée sur le flux standard de sortie, grâce à une chaîne de caractères qui contient des spécifications de format

▷ Exemple :

```
printf("%d", nombre)
```

▷ L'utilisation de `%d` comme spécification de format permet d'afficher la valeur d'une variable entière, ici la variable `nombre`

Questions?

LES STRUCTURES RÉPÉTITIVES

Initiation à la programmation C

---

uf-info.ue.init-prog-c@diff.u-bordeaux.fr

université de BORDEAUX

DÉFINITION

▷ Trois structures permettant la définition de boucles conditionnelles

- ▷ `while`
- ▷ `do ... while`
- ▷ `for`

▷ Théoriquement interchangeables mais il est recommandé de choisir toujours la structure la mieux adaptée au cas actuel (explicité plus tard)

WHILE

TANT QUE

▷ En langage algorithmique

```
tant que (<expression logique>) faire
        <bloc d'instructions>
fin tant que
```

▷ Tant que l'expression logique est évaluée à vraie, le bloc d'instructions est exécuté

▷ Lorsque l'expression logique est évaluée à faux lors du test, l'exécution se poursuit avec l'instruction suivant `fin tant que`

▷ Le bloc d'instructions peut être exécuté zéro ou plusieurs fois

STRUCTURE WHILE EN C

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

▷ Tant que l'expression est évaluée à une valeur différente de zéro ou `false`, le bloc d'instructions est exécuté

▷ Lorsque l'expression est évaluée à zéro ou `false` lors du test, l'exécution se poursuit avec l'instruction suivant l'accolade fermante du **`while`**

▷ Le bloc d'instructions peut être exécuté zéro ou plusieurs fois

EXEMPLES

▷ Exemple 1

```
int i = 0;
while(i<10){
    printf("%d\n", i);
    i=i+1;
}
```

▷ Exemple 2

```
int i = 10;
while(i){
    printf("%d\n", i);
    i=i-1;
}
```

AUTRE EXEMPLE

```
int i = 0;
int j = 10;
while(i < j){
   printf("i is still lower than j\n");
   i=i+1;
   j=j-1;
}
```

BOUCLE ”SANS” INSTRUCTION

▷ Seule l’expression est obligatoire, le bloc d’instructions peut être vide

```
int i = 10;
while(i=i-1){
}
printf(”i is finally null\n”);
```

▷ Une affectation est une expression retournant le résultat de son application ; ici `i=i-1` est évaluée comme la valeur actuelle de `i` après l’affectation (i.e., `i-1`)

DO … WHILE

STRUCTURE DO ... WHILE EN C

```
do{
    <bloc d'instructions>
}while(<expression>);
```

▷ La structure `while` évalue l'expression avant d'exécuter le bloc d'instructions

▷ La structure `do ... while` évalue l'expression après avoir exécuté le bloc d'instructions

▷ Le bloc d'instructions est exécuté au moins une fois et aussi longtemps que l'expression est évaluée à une valeur différente de zéro

▷ ⚠ le `';'` après l'expression est obligatoire

EXEMPLE

```
unsigned int n = 5;
unsigned int res = 1;
do{
  res=res*n;
  n=n-1;
}while(n > 0);
printf("factorial of %d is %d\n", n, res);
```

▷ Quelle est la valeur de `n` à la sortie de la boucle ?

AUTRE EXEMPLE

▷ En pratique, la structure **do ... while** n'est pas si fréquente que **while** mais dans certains cas, elle fournit une solution plus élégante

```
int n = 13;
do{
    n=n+1;
}while(n%3);
```

```
int n = 13;
n=n+1;
while(n%3){
    n=n+1;
}
```

▷ Que fait ce programme ?

FOR

STRUCTURE FOR EN C

```
for(<expr_init>; <expr_cond>; <expr_evo>){
    <bloc d'instructions>
}
```

▷ `<expr_init>` est évaluée une fois avant le passage dans la boucle

  ▷ Utilisée pour initialiser les données de la boucle

▷ `<expr_cond>` est évaluée avant chaque passage dans la boucle

  ▷ Utilisée pour décider si la boucle est répétée ou non

▷ `<expr_evo>` est évaluée à la fin de chaque passage dans la boucle

  ▷ Utilisée pour faire évoluer les données de `<expr_cond>`

STRUCTURE FOR EN C

▷ La structure `for` en C

```
for(<expr_init>; <expr_cond>; <expr_evo>){
    <bloc d'instructions>
}
```

est équivalente à

```
<expr_init>;
while(<expr_cond>){
    <bloc d'instructions>
    <expr_evo>;
}
```

WHILE VS FOR

- Question de lisibilité

```
int i;
for(i=0; i<10; i=i+1){
    printf("i=%d\n",i);
}
```

```
int i=0;
while(i<10){
    printf("i=%d\n",i);
    i=i+1;
}
```

STRUCTURE FOR EN C

▷ `for` est souvent utilisé comme une boucle de comptage

```
int i;
for (i=0; i<=20 ; i=i+1){
     printf("square of %d is %d \n", i, i*i);
}
```

▷ En pratique, `<expr_init>` et `<expr_evo>` contiennent souvent plusieurs instructions séparées par des virgules

```
int n, total;
for (total=0, n=1 ; n<11 ; n=n+1){
    total=total+n;
}
printf("Sum of integers in [1,10] is %d\n", total);
```

CHOIX DE LA STRUCTURE RÉPÉTITIVE

CONSEILS

▷ Si le bloc d'instructions ne doit pas être exécuté si la condition est fausse, alors utilisez `while` ou `for`

▷ Si le bloc d'instructions doit être exécuté au moins une fois, alors utilisez `do - while`

▷ Si le nombre d'exécutions du bloc d'instructions dépend d'une ou de plusieurs variables qui sont modifiées à la fin de chaque répétition, alors utilisez `for`

▷ Si le bloc d'instructions doit être exécuté aussi longtemps qu'une condition extérieure est vraie, alors utilisez `while`

EXERCICES

EXERCICE 1

▷ Ecrivez un programme qui calcule et affiche la somme, le produit et la moyenne des n premiers entiers où n est une variable définie en début du programme

▷ Choisissez un type approprié pour les valeurs à afficher

▷ Trois versions du programme seront données en utilisant respectivement `while`, `do ... while` et `for`

▷ Laquelle des trois variantes est la plus naturelle pour ce problème?

CORRECTION

```
int n=10, s=0, p=1, i=10;
while(i>0){
    s = s + i;
    p = p * i;
    i = i - 1;
}
printf("sum of [1,%d] : %d",n,s);
printf("product of [1,%d] : %d",n,p);
printf("average of [1,%d] : %f",n,s/(n*1.0));
```

CORRECTION

```
int n=10, s=0, p=1, i=n;
do{
    s = s + i;
    p = p * i;
    i = i - 1;
}while(i>0);
printf("sum of [1,%d] : %d",n,s);
printf("product of [1,%d] : %d",n,p);
printf("average of [1,%d] : %f",n,s/(n*1.0));
```

CORRECTION

```
int n=10, s=0, p=1, i;
for(i=n; i>0; i = i-1){
    s = s + i;
    p = p * i;
}
printf("sum of [1,%d] : %d",n,s);
printf("product of [1,%d] : %d",n,p);
printf("average of [1,%d] : %f",n,s/(n*1.0));
```

▷ A priori la solution la plus naturelle

EXERCICE 2

▷ Appliquez le programme précédent au plus petit multiple de 3 supérieur à n après l'avoir trouvé (le programme doit trouver ce dernier automatiquement)

- ▷ Quelle structure répétitive utilisez-vous?
- ▷ Pourquoi?

CORRECTION

```
int n=10, s=0, p=1, i;

do{
    n = n + 1;
}while(n%3);
...
printf("sum of [1,%d] : %d",n,s);
printf("product of [1,%d] : %d",n,p);
printf("average of [1,%d] : %f",n,s/(n*1.0));
```

▷ On utilise le `do ... while` car on sait que le nombre à trouver est au moins `n+1`

EXERCICE 3

▷ Calculez par des soustractions successives le quotient entier et le reste de la division entière de deux entiers définis par des variables n et m en début de programme

CORRECTION

```
int n = 5;   /* numerator */
int m = 2;   /* denominator */
int q;   /* quotient */
int r;   /* remainder */

r=n;
q=0;
while(r >= m){
    r = r - m;
    q = q + 1;
}

/*
for (r=n, q=0 ; r >= m ; q = q + 1){
      r = r - m;
}
*/
```

PORTÉE DES VARIABLES ET DÉFINITION DE FONCTIONS

Initiation à la programmation C

---

uf-info.ue.init-prog-c@diff.u-bordeaux.fr

université de BORDEAUX

PORTÉE D'UNE VARIABLE

NAISSANCE ET VIE D'UNE VARIABLE

▷ Un bloc est délimité par des accolades

▷ Toute variable a une durée de vie bornée au bloc où elle est déclarée

▷ Ce bloc définit la portée de la variable

▷ L'espace mémoire pour stocker la variable est alloué lors de sa déclaration et libéré à la fin du bloc où elle a été allouée

VARIABLE GLOBALE

▷ Les variables déclarées hors de tout bloc sont appelées globales et utilisables dans tout le fichier (à éviter)

▷ Une variable locale masque obligatoirement une variable de même nom située dans une portée plus englobante

```
int a=5, b=12;
int main(void)
{
    int a=3, i=0;
    printf("%d",a);//display 3
    for(i=0; i<10; i=i+1){
        int a=4;//A EVITER POUR MEILLEUR LISIBILITE
        printf("%d",a);//display 4
    }
    printf("%d",b);//display 12
}
```

EXERCICE 1

```c
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    int x = 3;
    {
        int x = 17;
        printf("%d\n", x);
    }
    printf("%d\n", x);
    return EXIT_SUCCESS;
}
```

▷ L'exécution du programme donne le résultat ci-dessous. Expliquez.

```
xxxx@init-prog:~/src/portee$ ./masquage1
17
3
xxxx@init-prog:~/src/portee$
```

EXERCICE 2

```c
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    {
        int a = 3;
        printf("%d\n", a);
    }
    int b = a;
    printf("%d\n", b);
    return EXIT_SUCCESS;
}
```

▷ La compilation du programme donne le résultat ci-dessous.

▷ Expliquez et corrigez l'erreur.

```
gcc -std=c99 -Wall -Werror  portee1.c -o portee1
portee1.c: In function 'main':
portee1.c:11: erreur: 'a' undeclared (first use in this function)
portee1.c:11: erreur: (Chaque identificateur non declaré est rapporté
portee1.c:11: erreur: une seule fois pour chaque fonction dans
portee1.c:11: erreur: laquelle il apparaît.)

Compilation exited abnormally with code 1 at Tue Feb 21 18:36:52
```

LES FONCTIONS

DÉFINITION

▷ La fonction est l'unité modulaire fondamentale en langage C

▷ Une fonction est généralement conçue pour effectuer une tâche spécifique et son nom reflète souvent cette tâche

▷ Une fonction permet de factoriser du code

▷ Une fonction contient des déclarations et des instructions

Déclaration de fonction

- Une fonction est caractérisée par son prototype
  - Un nom
    - Un liste (possiblement vide) de paramètres (correspondant à des variables locales de la fonction)
  - Un type de retour

```
type_retour nom(arguments);
float max(float a, float b);
```

VOID

▷ `void` est un type spécial

▷ En type de retour, il indique l'absence de valeur de retour

```
void print_sum(int a, int b){
    ...
}
```

▷ En paramètre, il indique l'absence de paramètre

```
int get_day(void){
...
}
```

Définition vs déclaration

▷ Définition = code de la fonction

▷ Déclaration = juste son prototype suivi de ';' (indispensable avant toute utilisation)

▷ Une définition vaut pour déclaration

```
void foo(void);

void bar(void){
    ...
    foo();
    ...
}

void foo(void){
    ...
    bar();
    ...
}
```

RETOURNER UNE VALEUR

▷ L'instruction `return` permet de quitter la fonction - quelque soit l'endroit où le return est appelé

▷ Renvoie la valeur produite par une expression si le type de retour est différent de `void`

▷ Indispensable si le type de retour est différent de `void`

```
void print_sum(float a, float b){
    printf("%f+%f=%f\n",a,b,a+b);
}
...
float print_and_return_sum(float a, float b){
    printf("%f+%f=%f\n",a,b,a+b);
    return a+b;
}
```

VALEUR DE RETOUR

▷ On peut ignorer la valeur de retour d'une fonction

▷ On ne peut pas utiliser une fonction ne renvoyant rien dans une expression

```
float print_and_return_sum(float a, float b){
    printf("%f+%f=%f\n",a,b,a+b);
    return a+b;
}
int main(void){
    float sum=0;
    print_and_return_sum(3.5, 1.1);
    sum=print_and_return_sum(3.5, 1.1);
    printf("%f",sum);
    return EXIT_SUCCESS;
}
```

Appel d'une fonction

▷ Un appel de fonction est une expression qui passe le contrôle et des arguments (le cas échéant) à une fonction et qui se présente sous la forme

```
nom_fonction (liste_d_expressions)
```

où `nom_fonction` est un nom de fonction et où `liste_d_expressions` est une liste d'expressions (séparées par des virgules)

```
max(5,6)
```

▷ Les valeurs des expressions de `liste_d_expressions` sont les arguments passés à la fonction

PASSAGE DE PARAMÈTRES : UNIQUEMENT PAR VALEUR

▷ Tous les arguments sont passés par valeur

▷ Cela signifie que seule la valeur de l'expression correspondant à l'argument est assignée au paramètre

▷ Les valeurs des arguments sont donc possiblement converties pour correspondre aux types des paramètres

▷ La fonction ne connaît pas l'origine (e.g. emplacement mémoire) de la valeur de l'argument passé

▷ La fonction utilise cette valeur sans affecter les éventuelles variables de l'expression de laquelle elle était initialement dérivée

APPEL D'UNE FONCTION

```
int max(int a, int b){
//1er appel a=3 et b=4
//2nd appel a=1 et b=4
    if(a<b){
        return b;
    }
    return a;
}
int main(void){
    int a=1, x=3, y=4, m=0;
    m=max(x, y); // equivalent a m=max(3,4)
    printf("max(%d,%d)=%d", x, y, m);
    printf("max(%d,%d)=%d", a, y, max(a,y));
}
```

DÉFINIR UNE FONCTION

ECRIRE UNE FONCTION

- Il faut réfléchir à l'utilité de la fonction
- Une fonction doit correspondre à une tâche
  - Par exemple, on ne mélange pas calcul et affichage

```
int minimum(int a, int b){
    int min=b;
    if(a<b){
        min=a;
    }
    printf("minimum=%d\n",min); //A EVITER !
    return min;
}
```

  - Pourquoi ?

Définir le prototype

- De quoi la fonction a-t-elle besoin ?
  - Paramètres
- Retourne-t-elle quelque chose ?
- Y a-t-il des cas d'erreurs ?
- Si oui, il y a 3 solutions
  - Mettre un commentaire
  - Renvoyer un code d'erreur
  - Afficher un message et quitter le programme

LE COMMENTAIRE

▷ Le commentaire est utile quand la fonction n'est pas censée être appelée pour certains cas

```
/**
* Copies the array 'src' into the 'dst' one.
* 'dst' is supposed to be large enough.
*/
void copy(int src[], int dst[]);
```

▷ Mais ne permet pas de prévenir les erreurs du programmeur liées au non respect de ce dernier

Le code d'erreur

▷ Il est possible de renvoyer un code d'erreur si la fonction ne devait rien renvoyer

▷ ⚠Sinon, on doit toujours pouvoir distinguer un cas d'erreur d'un cas normal (attention à la valeur choisie)

```
#define ERROR_CODE -1

int unsigned_short_divide(unsigned short num, unsigned short den){
    if(den==0){
        return ERROR_CODE;
    }
    return num/den;
}
```

EXERCICE

▷ Proposer la signature d'une fonction `short_divide` qui serait capable de gérer des petits entiers signés

LES ERREURS GRAVES

▷ En cas d'erreur insurmontable, il est possible d'interrompre le programme en appelant la fonction `exit` déclarée dans `stdlib.h` qui prend en paramètre `EXIT_FAILURE` ou `EXIT_SUCCESS`

```
int foo(){
    ...
    if(erreur_fatale){
        fprintf(stderr,"A fatal error occurred");
        exit(EXIT_FAILURE);
    }
    ...
}
```

▷ Il est important de quitter dès que possible une fonction en traitant les cas d'erreur le plus tôt possible

DOGGY BAG

TO TAKE AWAY ...

▷ Toute variable a une durée de vie bornée au bloc où elle est déclarée

▷ Eviter les variables globales

▷ Toute fonction doit être déclarée ou définie avant utilisation

▷ Tous les arguments et la valeur de retour d'une fonction sont le résultat d'évaluation d'expressions

▷ On doit toujours pouvoir distinguer un cas d'erreur d'un cas normal

Questions?

LES TABLEAUX À UNE DIMENSION

Initiation à la programmation C

---

uf-info.ue.init-prog-c@diff.u-bordeaux.fr

université de BORDEAUX

DÉFINITION

▷ Un tableau uni-dimensionnel en C correspond à un nombre entier et fini, disons `taille`, de variables simples du même type

▷ Il n'est pas utilisable pour stocker des informations de types différents

▷ Il est stocké de façon contiguë en mémoire (permet un accès rapide)

EXEMPLE

```
int fibo[12]={0,1,1,2,3,5,8,13,21,34,55,89};
```

définit un tableau de 12 variables entières initialisées par les 12 premières valeurs de la suite de Fibonacci

▷ L'accès à un élément du tableau se fait via un indice compris entre 0 et taille-1 (i.e., 11 ici)

▷ L'accès à l'élément d'indice j du tableau `tab` se fait à l'aide de `tab[j]`

▷ L'accès au i^{eme} élément ($1 \leq i \leq$ taille) du tableau `tab` se fait à l'aide de `tab[i-1]` (i.e. l'indice $i - 1$)

Déclaration

- Un tableau peut être déclaré pour tout type d'éléments
- `type nom[nb_elem];`

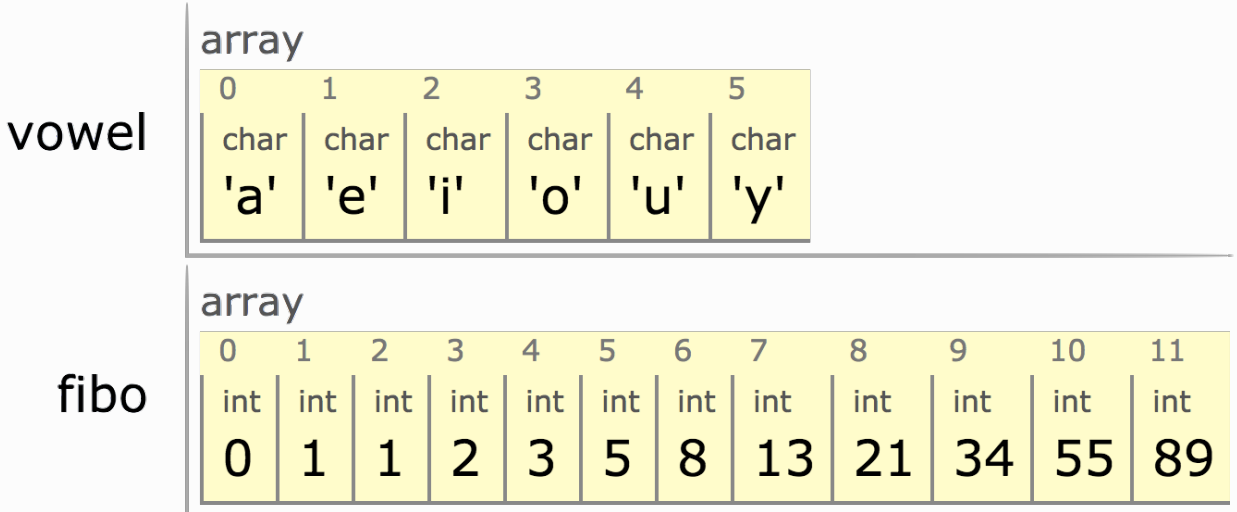
```
int t[10];
double cosinus[360];
char vowel[6];
...
```

MÉMORISATION

▷ En C, le nom du tableau représente l'adresse de son premier élément et les adresses des autres éléments sont calculées (automatiquement) relativement à l'adresse de son premier élément

▷ Exemple:

```
char vowel[6]={'a','e','i','o','u','y'};
int fibo[12]={0,1,1,2,3,5,8,13,21,34,55,89};
```



MÉMORISATION

▷ Exemple:

```
char vowel[6]={'a','e','i','o','u','y'};
int fibo[12]={0,1,1,2,3,5,8,13,21,34,55,89};
```

▷ Si un tableau est formé de `n` éléments chacun stocké sur `m` octets en mémoire, alors le tableau occupera `n*m` octets

▷ En supposant qu'une variable de type `int` occupe `4` octets, les tailles respectives de `vowel` et `fibo` sont de `6` et `48` octets

DIFFÉRENCES ENTRE VOWEL ET VOWEL[I]

▷ Etant donné

```
char vowel[6]={'a','e','i','o','u','y'};
```

la variable `vowel` correspond à l'adresse du tableau alors que `vowel[3]` correspond à un caractère

INITIALISATION

▷ Lors de la déclaration d'un tableau, on peut initialiser les éléments du tableau en indiquant la liste des valeurs successives entre accolades

▷ Exemples

```
int a[5]   = {10, 20, 30, 40, 50};
float b[4] = {-1.05, 3.33, 87e-5, -12.3E4};
int c[10]  = {1, 0, 0, 1, 1, 1, 0, 1, 0, 1};
```

▷ Il faut veiller à ce que le nombre de valeurs dans la liste corresponde à la taille du tableau

RÉSERVATION AUTOMATIQUE

▷ Si la taille n'est pas indiquée explicitement lors de l'initialisati
alors le compilateur réserve automatiquement le nombre d'octets nécessaires

▷ Exemples

```
int a[]   = {10, 20, 30, 40, 50};
//5 ints
float b[] = {-1.05, 3.33, 87e-5, -12.3E4};
// 4 floats
int c[]  = {1, 0, 0, 1, 1, 1, 0, 1, 0, 1};
// 10 ints
```

RÉSERVATION AUTOMATIQUE

▷ Attention, sans initialisation, un tableau doit avoir une taille fixée et connue

▷ Exemples

```
int a[]; // interdit
int b[5]; // autorisé
int i = 5;
int c[i]; //depuis c99
```

PROPRIÉTÉS

▷ ⚠Un tableau ne connaît pas sa taille

▷ L'opérateur `sizeof` ne fonctionne que pour les tableaux dont la taille est connue à la compilation (pas magique…)

▷ C'est la responsabilité du programmeur de connaître la taille d'un tableau

TAILLE

▷ Trois solutions

▷ Stocker la taille grâce à une constante

▷ Transmettre à l'aide de variables, la taille aux fonctions du programme en ayant besoin (i.e., ceux qui vont le parcourir)

▷ Utiliser une convention de fin de tableau en insérant un élément particulier dans sa dernière case

EXERCICE 1

▷ Ecrivez un programme affichant les éléments d'un tableau `t` d'entiers positifs séparés par un espace

▷ Proposez 3 versions différentes pour stocker la taille

CORRECTIONS

```c
//V1
#define N 5
...
unsigned int t[]={1,2,3,4,5};
unsigned int i;
for(i=0; i< N; i=i+1){
    printf("%u ", t[i]);
}
```

```c
//V2
unsigned int t[]={1,2,3,4,5};
unsigned int n=5
unsigned int i;
for(i=0; i< n; i=i+1){
    printf("%u ", t[i]);
}
```

```c
//V3
int t[]={1,2,3,4,5,-1};
unsigned int i=0;
while(t[i]>=0){
    printf("%d ", t[i]);
    i=i+1;
}
```

```c
//V4
#include <limits.h>
...
unsigned int t[]={1,2,3,UINT_MAX};
unsigned int i=0;
while(t[i]<UINT_MAX){
    printf("%u ", t[i]);
    i=i+1;
}
```

DÉBORDEMENT

▷ Il n'y a aucun contrôle de débordement (i.e., accès en dehors du tableau) ; ni à la compilation, ni à l'exécution

▷ Un accès à une zone mémoire non utilisée par le programme peut générer une erreur fatale engendrant l'arrêt de ce dernier

▷ Attention aux surprises, un débordement crée au mieux une erreur grave

```
int t[10];
int j=12;
printf("%d",t[10]);
```

```
$> ./a.out
12
```

EXERCICE 2

▷ Complétez le programme suivant afin d'afficher les éléments du tableau `t` dans l'ordre puis dans l'ordre inverse

```
#include <stdio.h>
#include <stdlib.h>
#define N 10
int main(void){
  int t[]={0,3,4,2,8,5,6,1,7,9};
  /* ... */
  return EXIT_SUCCESS;
}
```

CORRECTIONS

```c
#include <stdio.h>
#include <stdlib.h>
#define N 10
int main(void){
//N is supposed to be not null
  int t[]={0,3,4,2,8,5,6,1,7,9};
  unsigned int i;
  for(i=0; i<N; i=i+1){
    printf("%d ",t[i]);
  }
  printf("\n");
  for(i=N-1; i>0; i=i-1){
    printf("%d ",t[i]);
  }
  printf("%d ",t[0]);
  printf("\n");
  return EXIT_SUCCESS;
}
```

EXERCICE 2 (SUITE)

▷ Modifiez le programme afin que entre les deux affichages les valeurs du tableau soient renversées

▷ L'affichage devrait donc être

```
0 3 4 2 8 5 6 1 7 9
0 3 4 2 8 5 6 1 7 9
```

CORRECTIONS

```c
#include <stdio.h>
#include <stdlib.h>
#define N 10
int main(void){
  int t[]={0,3,4,2,8,5,6,1,7,9};
  unsigned int i;
  for(i=0; i<N; i=i+1){
    printf("%d ",t[i]);
  }
  printf("\n");
  for(i=0; i<N/2; i=i+1){
    int tmp=t[i];
    t[i]=t[N-1-i];
    t[N-1-i]=tmp;
  }
  for(i=N-1; i>0; i=i-1){
    printf("%d ",t[i]);
  }
  printf("%d\n",t[0]);
  return EXIT_SUCCESS;
}
```

array.mp4

EXERCICE 3 - FONCTIONS ET TABLEAUX

▷ En vous inspirant du programme précédent, proposez une fonction display permettant d'afficher dans l'ordre les éléments d'un tableau d'entiers

```
#define N 10
int main(void){
  int t[]={0,3,4,2,8,5,6,1,7,9};
  int fibo[]={0,1,1,2,3,5,8,13,21,34};
  display(t); //A noter que c'est t qui est
  //utilisé; pas t[]
  display(fibo);
  return EXIT_SUCCESS;
}
```

CORRECTIONS

```
void display(int tab[]){
  unsigned int i;
  for(i=0; i<N; i=i+1){
    printf("%d ",tab[i]);
  }
  printf("\n");
}
```

▷ Ici la taille n'est pas spécifiée entre les crochets car elle peut varier d'un appel à un autre

▷ C'est au programmeur de la connaître

EXERCICE 3 (SUITE)

▷ Modifiez votre programme pour que l'on puisse afficher des tableaux de tailles différentes

```
int main(void){
  int t[]={0,3,4,2,8};
  int fibo[]={0,1,1,2,3,5,8,13,21,34};
  //call to the new display function
  return EXIT_SUCCESS;
}
```

CORRECTIONS

```
void display(int tab[], unsigned int n){
  unsigned int i;
  for(i=0; i<n; i=i+1){
    printf("%d ",tab[i]);
  }
  printf("\n");
}

int main(void){
  int t[]={0,3,4,2,8};
  int fibo[]={0,1,1,2,3,5,8,13,21,34};
  display(t,5);
  display(fibo,10);
  return EXIT_SUCCESS;
}
```

EXERCICE 4

▷ Ecrivez une fonction scalar qui calcule le produit scalaire de deux vecteurs d'entiers **u** et **v** (de même taille définie par une variable **n**)

▷ Exemple:

```
[3,2,-4]*[2,-3,5] = 3*2+2*(-3)+(-4)*5 = -20
```

```
int main(void){
   int u[]={3,2,-4};
   int v[]={2,-3,5};
   unsigned int n=3;
   //To be completed
   return EXIT_SUCCESS;
}
```

CORRECTIONS

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

long scalar(int u[], int v[], unsigned int n){
   long res=0;
   unsigned int i;
   for (i=0; i<n; i=i+1){
      res = res + u[i]*v[i];
   }
   return res;
}

int main(void){
   int u[]={3,2,-4};
   int v[]={2,-3,5};
   unsigned int n=3;
   printf("Scalar product : %ld\n", scalar(u,v,n));
   return EXIT_SUCCESS;
}
```

LES CHAINES DE CARACTÈRES

- Pas un type, mais une convention de stockage
  - Tableau de caractères dont le dernier est `'\0'`

```
char msg[]={'W','e','l','c','o','m','e',' ',
'b','a','c','k','!','\0'};
```

CARACTÉRISTIQUES

▷ Une chaîne vide correspond à un tableau dont la première case contient '\0'

▷ La longueur d'une chaîne de caractères correspond au nombre de caractères avant '\0'

▷ Accès au n^{eme} caractère

```
char s[]={'w', 'x', 'y', 'z', '\0'};
char c=s[2]; //c vaut 'y'
```

EXERCICE 5

▷ Ecrivez une fonction `display_string` qui affiche tous les caractères d'une chaîne de caractères.

▷ Avez-vous besoin de connaitre le nombre de caractères à l'avance ?

▷ Que se passe-t-il si on appelle votre fonction avec le tableau suivant ?

```
char s[]={'t','o','o',' ','b','a','d'};
displayString(s);
```

▷ Pouvez-vous y faire quelque chose ?

CORRECTIONS

```
void display_string(char s[]){
    unsigned int i=0;
    while(s[i]!='\0'){
        printf("%c",s[i]);
        i=i+1;
    }
}
```

▷ Bravo, vous faites ce que `printf("%s",s)` fait déjà pour vous (mais pas mieux).

EXERCICE 6

▷ Ecrivez une fonction `string_len` qui détermine la taille d'une chaîne de caractères

CORRECTIONS

```
void string_len(char s[]){
    unsigned int i=0;
    while(s[i]!='\0'){
        i=i+1;
    }
    return i;
}
```

▷ Bravo, vous faites ce que la fonction **`strlen`** de la bibliothèque standard (fonction déclarée dans string.h) fait déjà pour vous (mais pas mieux).

ADRESSES ET POINTEURS

Initiation à la programmation C

---

uf-info.ue.init-prog-c@diff.u-bordeaux.fr

université de BORDEAUX

RAPPELS

- Adresse = entier positif = localisation d'un octet
- Type = taille de zone mémoire + interprétation
- Variable = adresse + type + valeur
- Tout est binaire et est une question d'interprétation

EXERCICE

▷ Comment écrire une fonction nommée swap permettant d'intervertir les valeurs de deux variables entières?

▷ Avec des variables locales, c'est impossible

```c
#include <stdlib.h>
void swap(int a, int b){
    int tmp = a;
    a = b;
    b = tmp;
}
```

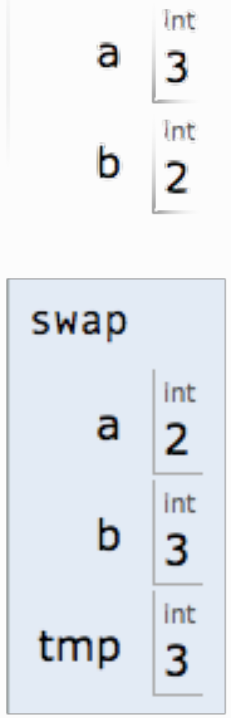
```c
int main(void){
    int a = 3;
    int b = 2;
    swap(a,b);
    return EXIT_SUCCESS;
}
```

EXERCICE

C (gcc 4.8, C11) EXPERIMENTAL!
see known bugs and report to philip@pgbovine.net

```
1  #include <stdlib.h>
2  void swap(int a, int b){
3      int tmp = a;
4      a = b;
5      b = tmp;
6  }
7  int main(void){
8      int a = 3;
9      int b = 2;
10     swap(a,b);
11     return EXIT_SUCCESS;
12 }
```

Frames



▷ Aucun lien entre les variables des fonctions main et swap

EXERCICE

▷ Utiliser des variables globales

```c
#include <stdlib.h>
int a, b;
void swap(){
    int tmp = a;
    a = b;
    b = tmp;
}
```

```c
int main(void){
    a = 3;
    b = 2;
    swap();
    return EXIT_SUCCESS;
}
```

▷ Quelle utilité d'une fonction alors ?

▷ C'est impossible d'appeler swap sur d'autres entiers

STOCKER DES ADRESSES

▷ Il est intéressant de pouvoir stocker des adresses de variables

▷ Au lieu de transmettre la valeur d'une variable, on pourra alors transmettre son adresse

▷ Il faut connaitre l'adresse mais aussi le type de la variable

▷ Un pointeur est un type particulier "dénoté par *" dont la valeur est une adresse et dont le type "cible" est spécifié

▷ e.g., `int *`, `char *`

MANIPULER DES ADRESSES

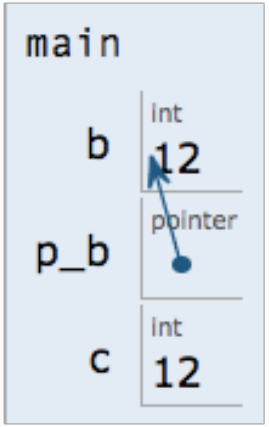
▷ L'opérateur & permet de récupérer l'adresse d'une variable sous la forme d'un pointeur

▷ L'opérateur * permet de récupérer le contenu de la mémoire situé à une adresse donnée par un pointeur

C (gcc 4.8, C11) EXPERIMENTAL!
see known bugs and report to philip@pgbovine.net

```
1  #include <stdlib.h>
2
3  int main(void){
4      int b = 12;
5      int * p_b = &(b);
6      int c = *(p_b); // c=12
7      return EXIT_SUCCESS;
8  }
```

Frames



EXERCICE 1

▷ Complétez les fonctions suivantes

```c
#include <stdlib.h>
void set(int *p_a, int val){

}

void reset(int *p_a){

}
```

```c
int main(void){
    int a = 3;
    int b = 2;
    int c = 1;
    reset(&(a));
    set(&(b),4);
    set(&(c),b);
    return EXIT_SUCCESS;
}
```

CORRECTION

```
void set(int *p_a, int val){
    *(p_a)=val;
}

void reset(int *p_a){
    set(p_a,0);
}
```

set_reset.mp4

UTILITÉ DES ADRESSES

```c
#include <stdlib.h>
void swap(int *p_a, int *p_b){
    int tmp = *(p_a);
    *(p_a) = *(p_b);
    *(p_b) = tmp;
}
int main(void){
    int a = 3;
    int b = 2;
    int c = 1;
    swap(&(a),&(b));
    swap(&(b),&(c));
    return EXIT_SUCCESS;
}
```

pointer.mp4

▷ Pas besoin de faire des copies des données elles-mêmes, il suffit de connaître leur adresse

EXERCICE 2

▷ Ecrivez une fonction `min_max` qui calcule le minimum et le maximum de deux nombres entiers positifs

▷ Ecrivez une fonction `main` permettant d'appeler cette fonction et d'afficher le minimum et le maximum

CORRECTION

```
void min_max(unsigned int a, unsigned int b,
        unsigned int *min, unsigned int *max){
    *(max) = a;
    *(min) = a;
    if (a < b){
        *(max) = b;
    }else{
        *(min) = b;
    }
}
```

CORRECTION - SUITE

```
void min_max(unsigned int a, unsigned int b,
        unsigned int *min, unsigned int *max);

int main (void) {
    unsigned int a =5, b=12;
    unsigned int min = 0, max = 0;
    min_max(a, b, &(min), &(max));
    printf("min of %u and %u = %u\n", a, b, min);
    printf("max of %u and %u =  %u\n", a, b, max);
    return EXIT_SUCCESS;
}
```

RETOUR SUR LES TABLEAUX

▷ Un tableau ne connaît pas sa taille

▷ Tableau = adresse de son premier élément et un type

▷ `int tab[5]` et `int *p_a` sont tous deux l'adresse d'un élément de type `int`

RETOUR SUR LES TABLEAUX

▷ ⚠L'espace alloué n'est pas le même: `tab` correspond à une adresse où 5 entiers peuvent être stockés; `p_a` correspond à une variable pouvant stocker une adresse

C (gcc 4.8, C11) EXPERIMENTAL!
see known bugs and report to philip@pgbovine.net

```
1  #include <stdlib.h>
2
3  int main(void){
4      int a = 1;
5      int tab[5]={0,0,0,0,0};
6      tab[0]=a;
7      int * p_a = &(a);
8      return EXIT_SUCCESS;
9  }
```

Frames

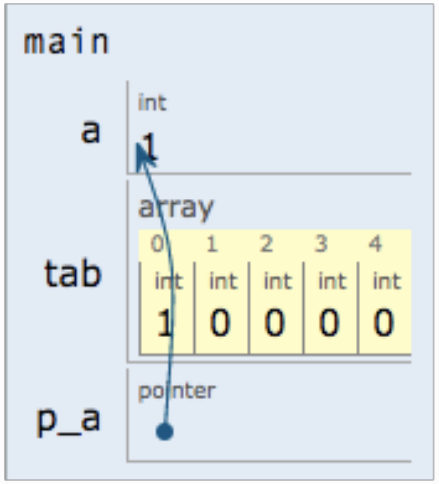


Tableau en paramètre d'une fonction

▷ A votre avis, que se passe-t-il lorsqu'un tableau est mis en paramètre d'une fonction ?

▷ Quelle valeur est transmise à la fonction ?

array_pointer.mp4

Tableau en retour d'une fonction

▷ A votre avis, peut-on renvoyer un tableau depuis une fonction ?

▷ Un tableau passé en paramètre ?

▷ Un tableau correspondant à une variable locale de la fonction ?

Tableau en paramètre d'une fonction

▷ A votre avis, peut-on renvoyer un tableau depuis une fonction ?

▷ Un tableau passé en paramètre ?

▷ Peu d'intérêt puisque l'on peut le modifier directement dès lors qu'il est en paramètre

▷ Un tableau correspondant à une variable locale de la fonction ?

▷ Impossible car la variable correspondante meurt à la fin de la fonction

▷ Pour éventuellement copier les éléments du tableau, il faudrait connaitre sa taille

EXERCICE 3

▷ Ecrivez une fonction swap_tab permettant d'inverser l'ordre des éléments d'un tableau de double

CORRECTIONS

```
void swap_tab(double t[], unsigned int n){
    for(unsigned int i=0; i<n/2; i=i+1){
        double tmp=t[i];
        t[i]=t[n-1-i];
        t[n-1-i]=tmp;
    }
}
```

ADRESSES INVALIDES

▷ Par convention, une adresse invalide est représentée par la constante symbolique notée `NULL` qui est définie dans `stdio.h`

▷ L'accès à l'adresse `NULL` provoque l'interruption du programme avec un `Segmentation fault`

ARITHMÉTIQUE DES POINTEURS

Initiation à la programmation C

---

uf-info.ue.init-prog-c@diff.u-bordeaux.fr

université de BORDEAUX

RAPPEL : LES POINTEURS

- Une adresse = un entier positif
- Un pointeur est un type particulier "dénoté par *" dont la valeur est une adresse et dont le type "cible" est spécifié
  - e.g., `int *`, `char *`
- L'opérateur `&` permet de récupérer l'adresse d'une variable sous la forme d'un pointeur
- L'opérateur `*` permet de récupérer le contenu de la mémoire situé à une adresse donnée par un pointeur

RAPPEL SUR LES TABLEAUX

▷ Utile pour stocker plusieurs éléments de même type

▷ N'est pas utilisable pour stocker des informations de types différents

▷ Stocker de façon contiguë en mémoire

▷ Déclaration: `type nom[nb_elem];`

▷ L'accès au i^{me} élément du tableau `tab` se fait à l'aide de `tab[i-1]`

RAPPEL SUR LES TABLEAUX

▷ Un tableau ne connaît pas sa taille

▷ Tableau = adresse de son premier élément et un type (comme les pointeurs)

▷ `int tab[2]` et `int *p_t` sont "PRESQUE" équivalents

ARITHMÉTIQUE DES POINTEURS

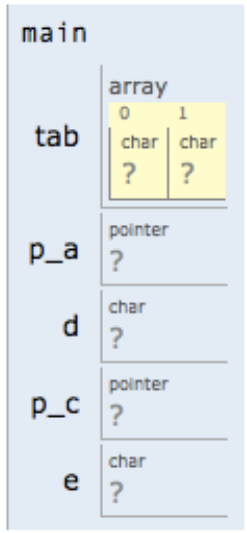
EXEMPLE 1

C (gcc 4.8, C11)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void) {
5    char tab[] = {'A','C'};
6    char *p_a = &(tab[0]);
7    char d = *(p_a+1);
8    char *p_c = p_a+1;
9    char e = *(p_a)+1;
10   printf("%p %p\n",p_a,p_c);
11   return EXIT_SUCCESS;
12 }
```

Print output (drag lower right corner to resize)

Variables Objets



arithmetique-1.mp4

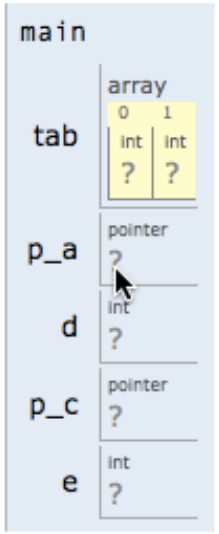
EXEMPLE 2

C (gcc 4.8, C11)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void) {
5    int tab[] = {257,3};
6    int *p_a = &(tab[0]);
7    int d = *(p_a+1);
8    int *p_c = p_a+1;
9    int e = *(p_a)+1;
10   printf("%p %p\n",p_a,p_c);
11   return EXIT_SUCCESS;
12 }
```

Print output (drag lower right corner to resize)

Variables Objets



arithmetique-2.mp4

ARITHMÉTIQUE DES POINTEURS

▷ Une opération arithmétique au sein de l'opérateur `*` est liée à la taille du type cible

▷ Etant donné un pointeur `p` sur un objet de type `type`, `p + j` pointe `j*sizeof(type)` octets plus loin que `p`

RETOUR SUR LES TABLEAUX

ARITHMÉTIQUE DES POINTEURS ET TABLEAUX

▷ `tab[1]` et `*(tab+1)` sont égaux – à savoir la valeur du 2ème élément du tableau `tab`

▷ A quoi est équivalent `tab[j]` ?

ARITHMÉTIQUE DES POINTEURS ET TABLEAUX

- `tab[1]` et `*(tab+1)` sont égaux – à savoir la valeur du 2ème élément du tableau `tab`
- A quoi est équivalent `tab[j]` ?
  - `*(tab+j)`

EXERCICE 1

▷ Soit `p` un pointeur qui 'pointe' sur la première case d'un tableau `t` :

```
int t[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};
int *p = &(t[0]); // equivalent à p = t
```

Quelles valeurs ou adresses fournissent ces expressions ?

▷ `*(p)+2`

▷ `*(p+2)`

▷ `*p+2`

EXERCICE 1

▷ Soit `p` un pointeur qui 'pointe' sur la première case d'un tableau `t` :

```
int t[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};
int *p = &(t[0]); // equivalent à p = t
```

Quelles valeurs ou adresses fournissent ces expressions ?

▷ `*(p)+2`

  ▷ `t[0]+2 = 14`

▷ `*(p+2)`

  ▷ `t[2] = 34`

▷ `*p+2`

  ▷ `t[0]+2 = 14`

EXERCICE 1

▷ Soit `p` un pointeur qui 'pointe' sur la première case d'un tableau `t` :

```
int t[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};
int *p = &(t[0]); // equivalent à p = t
```

Quelles valeurs ou adresses fournissent ces expressions ?

▷ `&(t[4])-3`

▷ `t+3`

▷ `p+(*(p)-10)`

EXERCICE 1

▷ Soit `p` un pointeur qui 'pointe' sur la première case d'un tableau `t` :

```
int t[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};
int *p = &(t[0]); // equivalent à p = t
```

Quelles valeurs ou adresses fournissent ces expressions ?

▷ `&(t[4])-3`

  ▷ `&(t[1]) == p+1`

▷ `t+3`

  ▷ `&(t[3]) == p+3`

▷ `p+(*(p)-10)`

  ▷ `&(t[0]) + (12 - 10) == &(t[2]) == p+2`

EXERCICE 2

▷ Que fait la fonction suivante ?

```
bool f1(int *t, int n){
  for(int i=1 ; i<n ; i=i+1){
    if ( *(t+i) > *(t+i-1) ){
      return false;
    }
  }
  return true;
}
```

▷ Réécrire la fonction en utilisant l'indexation (opérateur [ ]) et non pas l'arithmétique des pointeurs.

SOLUTION

▷ La fonction teste si le tableau est trié dans l'ordre croissant

```
bool f1(int t[], int n){
  for(int i=1 ; i<n ; i=i+1){
    if ( t[i] > t[i-1] ){
      return false;
    }
  }
  return true;
}
```

EXERCICE 3

▷ Que fait la fonction suivante ?

```
bool f2(int *t, unsigned int n){
  int *d, *g;
  g = t;
  d = t+n-1;
  while (g<d){
    if ( *g != *d){
      return false;
    }
    g = g+1;
    d = d-1;
  }
  return true;
}
```

SOLUTION

▷ La fonction teste si le tableau est symétrique

```
bool f2(int t[], unsigned int n){
  unsigned int g = 0;
  unsigned int d = n-1;
  while (g<d){
    if (t[g] != t[d]){
      return false;
    }
    g = g+1;
    d = d-1;
  }
  return true;
}
```

INTERACTION AVEC L'UTILISATEUR

Initiation à la programmation C

uf-info.ue.init-prog-c@diff.u-bordeaux.fr

université de BORDEAUX

ARGUMENTS DE MAIN

▷ La fonction `main` peut récupérer les informations fournies sur la ligne de commande lors de l'appel du programme

▷ Par exemple, avec `$>./a.out AA "10 2" 1.0 b` on peut récupérer sous forme de chaînes de caractères

- ▷ `{'.','/','a','.','o','u','t','\0'}`
- ▷ `{'A','A','\0'}`
- ▷ `{'1','0',' ','2','\0'}`
- ▷ `{'1','.','0','\0'}`
- ▷ `{'b','\0'}`

▷ L'objectif est de fournir un moyen de passer des informations à un programme

ARGUMENTS DE MAIN

▷ Pour ce faire, la fonction `main` prend en paramètre un nombre et un tableau de chaînes de caractères

```
int main(int arg_count, char* arg_values[]);
```

▷ `arg_count` correspond au nombre de paramètres accessibles (le premier étant toujours le nom de l'exécutable)

▷ `arg_values` est un tableau de **chaînes de caractères** contenant les paramètres du programme

ARGUMENTS DE MAIN

```
int main(int arg_count, char* arg_values[]){
  int i;
  for(i=0; i<arg_count; i=i+1){
    printf("arg #%d=%s\n",i,arg_values[i]);
  }
  return EXIT_SUCCESS;
}
```

```
$>./a.out AA "10 2" e
  arg #0=./a.out
  arg #1=AA
  arg #2=10 2
  arg #3=e
```

POURQUOI INT ARG_COUNT (PLUTÔT QUE UNSIGNED INT) ?

▷ Le language C n'avait tout simplement pas de type non signé au début

▷ Par soucis de compatibilité ascendante, le type `int` a été conservé

▷ Notons que la norme garantit qu'au minimum le nom de l'exécutable est toujours fourni et donc `arg_count` est toujours supérieur à 0

EXERCICE 1

▷ Pour parcourir la liste des paramètres, on peut utiliser

▷ le nombre de paramètres accessibles `arg_count`

▷ ou le fait que le tableau de paramètres se termine par un pointeur `NULL` (la sentinelle)

▷ Modifiez le programme précédent en exploitant la seconde possibilité avec un `for` puis un `while`

```
int main(int arg_count, char* arg_values[]){
  int i;
  for(i=0; i<arg_count; i=i+1){
    printf("arg #%d=%s\n",i,arg_values[i]);
  }
  return EXIT_SUCCESS;
}
```


FONCTION USAGE

▷ Afin de traiter les cas de non respect du nombre de paramètres passés en argument, il est courant d'écrire une fonction `usage`

```
void usage(char *cmd){
  fprintf (stderr, "Usage: %s uint1 uint2 code_op\n", cmd);
  fprintf (stderr, "where uint1 and uint2 are unsigned values\n");
  fprintf (stderr, "and code_op is either \"min\" or \"max\"\n");
  fprintf (stderr, "Example: %s 14 5678 max\n", cmd);
  exit(EXIT_FAILURE);
}

int main(int arg_count, char* arg_values[]){
  if(arg_count != 4){ //4 arguments are expected
    usage(arg_values[0]);
  }
  ...
  return EXIT_SUCCESS;
}
```


UTILISATION DES ARGUMENTS DE MAIN

▷ Les paramètres sont transmis sous la forme de chaînes de caractères

▷ Si un paramètre est utilisé dans le programme comme un type numérique, il devra être converti

▷ La bibliothèque standard fournit des fonctions de conversion (déclarées dans `stdlib.h`)

```
long int strtol(const char *str, char **endptr, int base)
unsigned long int strtoul(const char *str, char **endptr, int base)
double strtod(const char *str, char **endptr)
```

LA FONCTION STRTOL

▷ La fonction ”string to long” de la bibliothèque standard renvoie un entier long à partir de 3 paramètres

```
long int strtol(const char *str, char **endptr, int base)
```

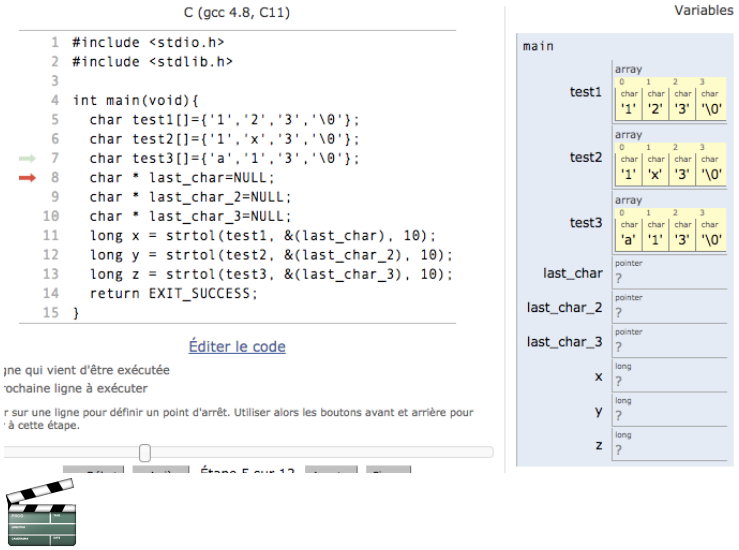
▷ `str` représente la chaîne de caractères à décoder

▷ Le modificateur `const` signifie que la fonction garantit que la chaîne ne sera pas modifiée

▷ `endptr` est l’adresse d’une variable qui contiendra, au retour de la fonction, l’adresse du premier caractère de `str` qui n’a pas pu être décodé

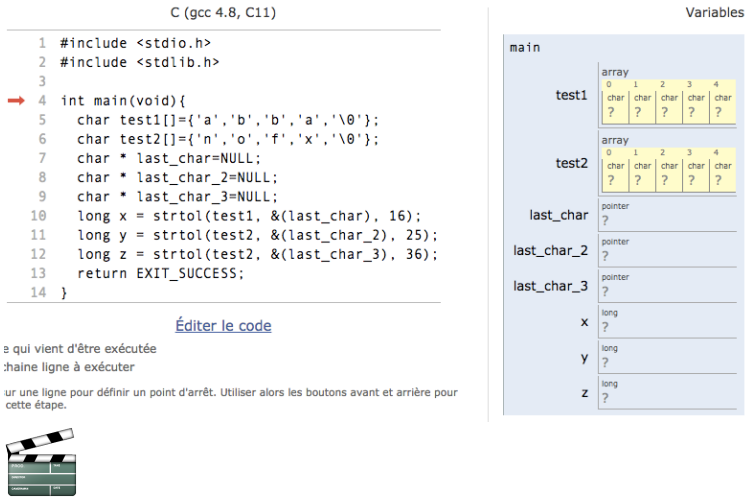
▷ `base` est la base dans laquelle est exprimé le nombre en entrée ($2 \leq$ `base` ≤ 36)

LA FONCTION STRTOL



strtol-1.mp4

LA FONCTION STRTOL



strtol-2.mp4

LES FONCTIONS STRTOUL ET STRTOD

▷ La fonction ”string to unsigned long” fonctionne de manière similaire à `strtol` mais renvoie un entier non signé

```
unsigned long int strtoul(const char *str, char **endptr, int base)
```

▷ La fonction ”string to double” ne prend pas de base en paramètre et renvoie un double

```
double strtod(const char *str, char **endptr)
```

EXERCICE 2

▷ Proposer un programme pourvu d'une fonction **usage** permettant de récupérer deux entiers **a** et **b** et un double **c** et qui vérifie et affiche si la division de **a** par **b** donne **c**

▷ Par exemple

```
$> ./a.out 5 2 2.5
True
$> ./a.out 3 2 2.5
False
$> ./a.out 3 0 3
Division by 0 impossible
$> ./a.out 3 2
Usage: ./a.out int1 int2 res_div
where int1 and int2 are long values
and res_div is the expected result of the division of int1 by int2
Example: ./a.out 5 2 2.5
```

TYPES STRUCTURÉS ET TYPEDEF

Initiation à la programmation C

---

uf-info.ue.init-prog-c@diff.u-bordeaux.fr

université de BORDEAUX

TYPES STRUCTURÉS

LES STRUCTURES

▷ Une structure est une construction permettant de regrouper séquentiellement des données C de types différents

▷ Chaque donnée de la structure est appelée un **champ**

▷ Déclaration d'un type structuré

```
struct nom_type {
    type_champ1 nom_champ1;
    type_champ2 nom_champ2;
    ...
    type_champn nom_champn;
};
```

▷ ⚠ Le `;` est nécessaire

Déclaration d'un type structuré

▷ Exemples :

```
struct pixel {
  unsigned short x;
  unsigned short y;
  unsigned long  color;
};
```

```
struct foo {
  char a;
  int  b;
  char c;
};
```

▷ `struct pixel` et `struct foo` sont des nouveaux types

▷ Une fois ces types déclarés, on peut déclarer des variables de ces types.

DÉCLARATION ET INITIALISATION

▷ La déclaration d'une variable de type structuré s'effectue ainsi

```
struct nom_type nom_var;
```

▷ Exemple : `struct pixel p;`

  ▷ La variable `p` est de type `struct pixel`

▷ L'initialisation d'une variable de type structuré lors de sa déclaration peut s'effectuer ainsi

```
struct nom_type nom_var={val1,val2,...,valn};
```

▷ Exemple : `struct foo t={'x',145,'y'};`

  ▷ Le champ `a` de la variable `t` sera égal à `'x'`, le champ `b` de la variable `t` sera égal à `145`, le champ `c` sera égal à `'y'`

Accès à un champ d'une structure : opérateur .

▷ L'accès à un champ d'une structure s'effectue comme suit: `nom_var.nom_champ`

```
struct complex {
    double real; //real part
    double img;  //imaginary part
};

int main(void) {
    struct complex c = {1.2, 6.3};
    printf("%.1lf +%.1lf i\n", c.real, c.img);
    c.real = c.real+5.0;
    printf("%.1lf +%.1lf i\n", c.real, c.img);
    return EXIT_SUCCESS;
}

$>./a.out
1.2 +6.3 i
6.2 +6.3 i
```

OPÉRATIONS SUR UNE STRUCTURE

▷ L'affectation d'une structure fonctionne et produit la recopie des valeurs des champs.

```
int main(void) {
    struct complex c1 = {1.2, 6.3};
    printf("c1 = %.1lf +%.1lf i\n", c1.real, c1.img);
    struct complex c2 = c1;
    c2.real = c2.real+5.0;
    printf("c1 = %.1lf +%.1lf i\n", c1.real, c1.img);
    printf("c2 = %.1lf +%.1lf i\n", c2.real, c2.img);
    return EXIT_SUCCESS;
}
```

les bits de la structure `c1` sont copiés dans la structure `c2`. L'instruction `c2 = c1;` est équivalente à `c2.real = c1.real;` et `c2.img = c1.img;`.

▷ **Exercice :** Qu'affiche ce programme ?

CORRECTION

```
c1 = 1.2 +6.3 i
c1 = 1.2 +6.3 i
c2 = 6.2 +6.3 i
```

OPÉRATIONS SUR UNE STRUCTURE

```c
struct foo {
  char a;
  int  b;
  char c;
};
int main(void) {
  struct foo f, g, h;
  f = (struct foo) {'x', 10, 'y'}; //cast nécessaire pour une affectatio
  g = (struct foo) {.a = 'y'}; // autres champs initalisés à leur
                               // valeur par défaut
  h = (struct foo) {.c = 'z'};
  printf("<%c, %d, %c>\n", f.a, f.b, f.c);
  printf("<%c, %d, %c>\n", g.a, g.b, g.c);
  printf("<%c, %d, %c>\n", h.a, h.b, h.c);
  return EXIT_SUCCESS;
}
/*
<x, 10, y>
<y, 0, >
<, 0, z>
*/
```

Opérations sur une structure

▷ Comparaison de structures

▷ Il n'y a pas d'opérateur de comparaison de deux structures

▷ Par exemple, l'opération `c1==c2` n'est pas correcte syntaxiquement (erreur de compilation)

▷ **Exercice :** Ecrire une expression permettant de tester si deux complexes `c1` et `c2` sont égaux champ à champ.

CORRECTION

Il faut faire une comparaison champ par champ et écrire le code suivant :

```
(c1.real == c2.real ) && (c1.img == c2.img)
```

Structures en paramètres d'une fonction

▷ Ce sont des copies des structures qui sont passées en paramètre

▷ Dans les deux cas suivants, il faut transmettre l'adresse de la structure (sous forme de pointeur) et non la structure :

  ▷ Si la structure occupe beaucoup de place en mémoire, car la copie de la structure peut être coûteuse en temps.

  ▷ Si la fonction doit modifier la structure.

EXERCICE

Qu'affiche ce programme ?

```
struct test{
  int i;
};

void set_to_zero(struct test a) {
  a.i = 0;
}

int main(void) {
  struct test x = {18};
  set_to_zero(x);
  printf("x : %d\n", x.i);
  return EXIT_SUCCESS;
}
```

RETOURNER UNE STRUCTURE DANS UNE FONCTION

▷ Retourner une structure est possible

▷ Dans le cas où la structure occupe beaucoup de place en mémoire, retourner une structure pose le même problème (coût de la copie) que le passage en argument d'une structure :

▷ → il ne faut alors l'utiliser que si on ne peut pas faire autrement

EXERCICE

▷ L'objectif est d'écrire une fonction `min_max` qui calcule le minimum et le maximum de deux nombres entiers positifs.

▷ Définir le type `struct res_min_max` contenant deux champs de type `unsigned int`.

▷ Ecrire la fonction `min_max` qui ne retourne rien mais modifie la structure de type `struct res_min_max` dont l'adresse est passée en paramètre.

CORRECTION

```c
struct res_min_max {
  unsigned int min, max;
};
void min_max(unsigned int a, unsigned int b,
             struct res_min_max *p_res) {
  (*p_res).min = a;
  (*p_res).max = a;
  if (a < b){
    (*p_res).max = b;
  }else{
    (*p_res).min = b;
  }
}
int main(void){
  unsigned int a = 125, b = 45;
  struct res_min_max res;
  min_max(a, b, &(res));
  printf("%u %u\n", res.min, res.max);
  return EXIT_SUCCESS;
}
```

TYPEDEF

TYPEDEF

▷ Le constructeur `typedef` permet de donner un nom à un type simple ou composé via la déclaration suivante

`typedef type synonyme;`

▷ L'intérêt de `typedef` est d'améliorer la lisibilité des programmes

```
typedef unsigned int uint;

int main(void) {
    uint i = 34;
    unsigned int j = i;
    return EXIT_SUCCESS;
}
```

`uint` est alors un type synonyme de `unsigned int`

TYPEDEF

```c
struct complex{
  double real, img;
};

typedef struct complex complex_t;

int main(void) {
    complex_t c1 = {2.7, 6.3};
    struct complex c2 = c1;
    return EXIT_SUCCESS;
}
```

`complex_t` est alors un type synonyme de `struct complex`

POUR ALLER PLUS LOIN

EXEMPLE DE STRUCTURE CONTENANT UN TABLEAU

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

#define N 100000
struct array {
  int t[N];
  int size;
};

void display(struct array* p_a) {
  for (int i=0; i< (*p_a).size; i=i+1) {
    printf("%d\n",(*p_a).t[i]);
  }
}
int main(void) {
  struct array a = {{12, 56, -4, 25}, 4};
  display(&(a));
  return EXIT_SUCCESS;
}
```

EXEMPLE DE TABLEAU CONTENANT DES STRUCTURES

```c
struct complex {
  double real, img;
};
typedef struct complex complex_t;
void display_complex(complex_t *p_c){
  printf("%.2lf %+.2lf i\n", (*p_c).real, (*p_c).img);
}
void display_tab_complex(complex_t *p, int n) {
  for (int i=0; i< n; i=i+1){
    display_complex(p+i);
  }
}
int main(void) {
  complex_t tab_complex[] = {{2.0, 3.5}, {-3.0, 3.5}, {34.0, 2.0}};
  display_tab_complex(tab_complex, 3);
  return EXIT_SUCCESS;
}
/*
2.00 +3.50 i
-3.00 +3.50 i
34.00 +2.00 i
*/
```

POINTEURS SUR STRUCTURES, OPÉRATEUR ->

▷ Si `p` est un pointeur sur une structure, l'accès aux champs peut se faire avec l'expression `(*p).champ` ⚠ Les parenthèses sont indispensables.

▷ Il est également possible d'utiliser l'opérateur `->`

  ▷ `(*p).champ` et `p->champ` sont équivalents

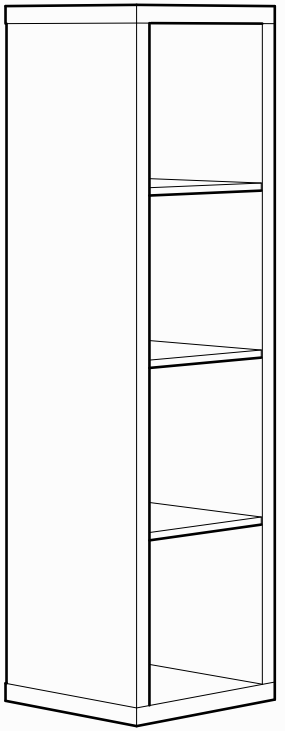
▷ Exemple

```
struct complex c1 = {1.2, 6.3};
struct complex *pc = &(c);
/* pc est de type pointeur sur struct complex */
/* pc est inialisé avec l'adresse de c */
(*pc).real = 14.5;
pc->real = 14.5;
/* (*pc).real et pc->real sont des notations */
/* équivalentes */
```

DES SOURCES À L'EXÉCUTABLE

LES FICHIERS SOURCES

KALLAX



IKEA
Design and Quality
IKEA of Sweden

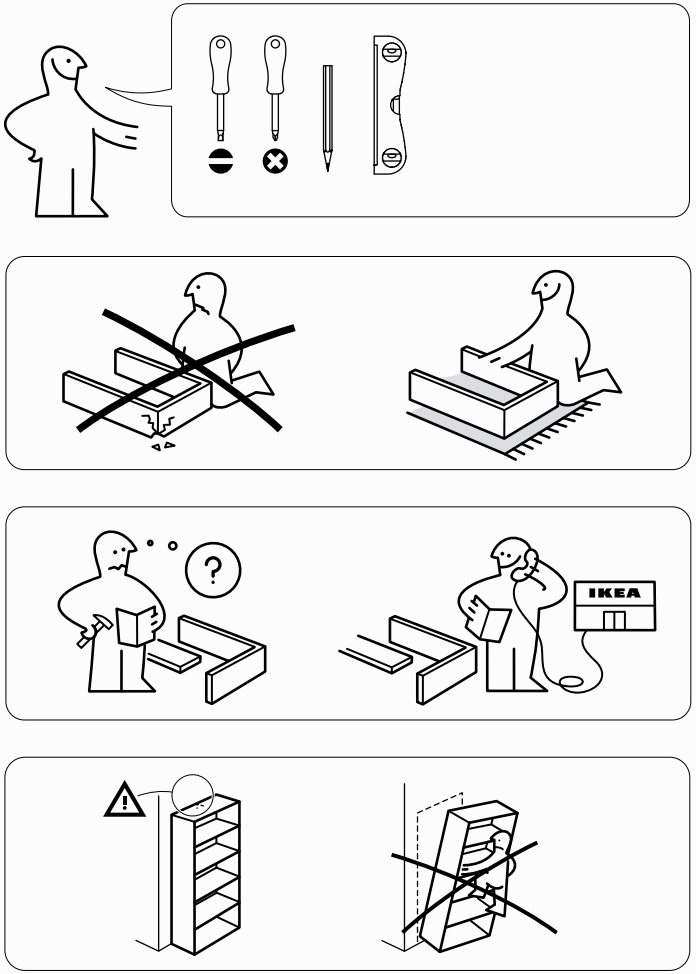
▷ Un fichier source est un texte exprimé dans un langage de programmation décrivant des instructions permettant d'obtenir un objet fonctionnel précis

LES FICHIERS SOURCES

▷ Pouvant suivre des conventions données

▷ Ce sont bien des règles de bons sens

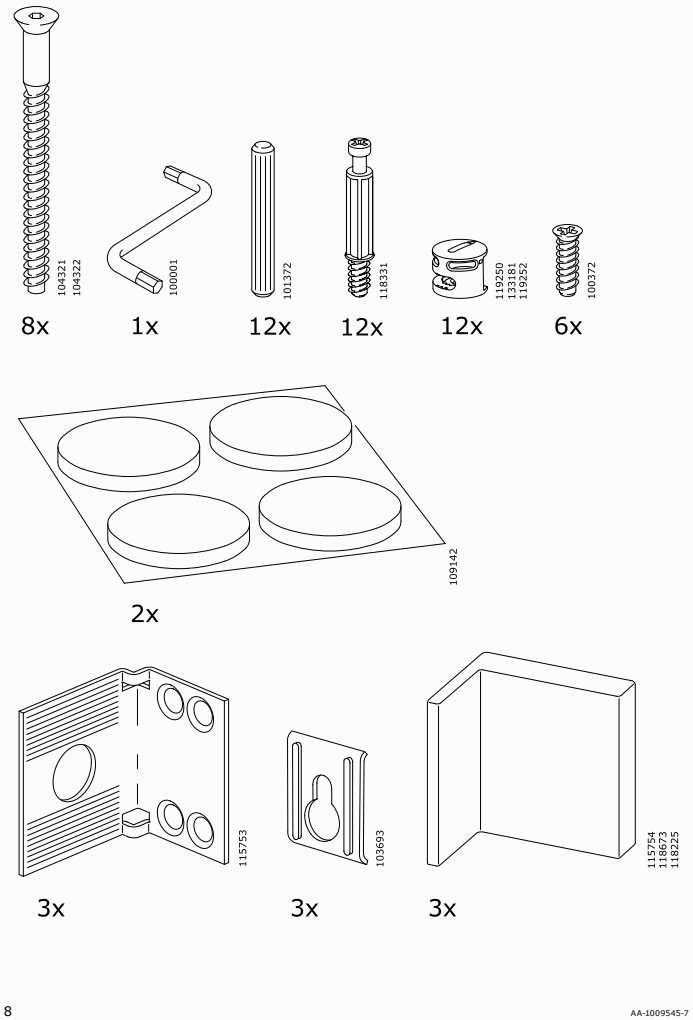
▷ Que l'on est pas obligé de suivre mais ⚠



7

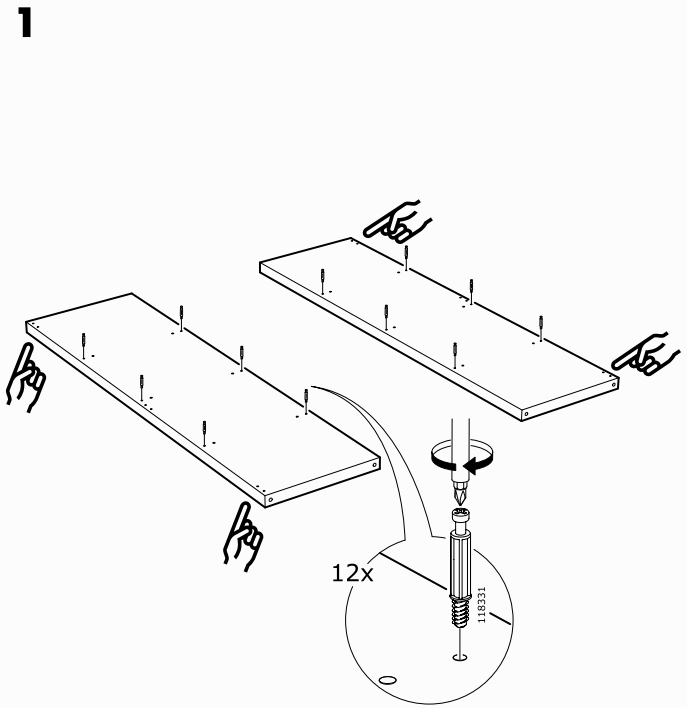
LES FICHIERS SOURCES

▷ Utilisant des outils pré-existants et disponibles (librairies)



LES FICHIERS SOURCES

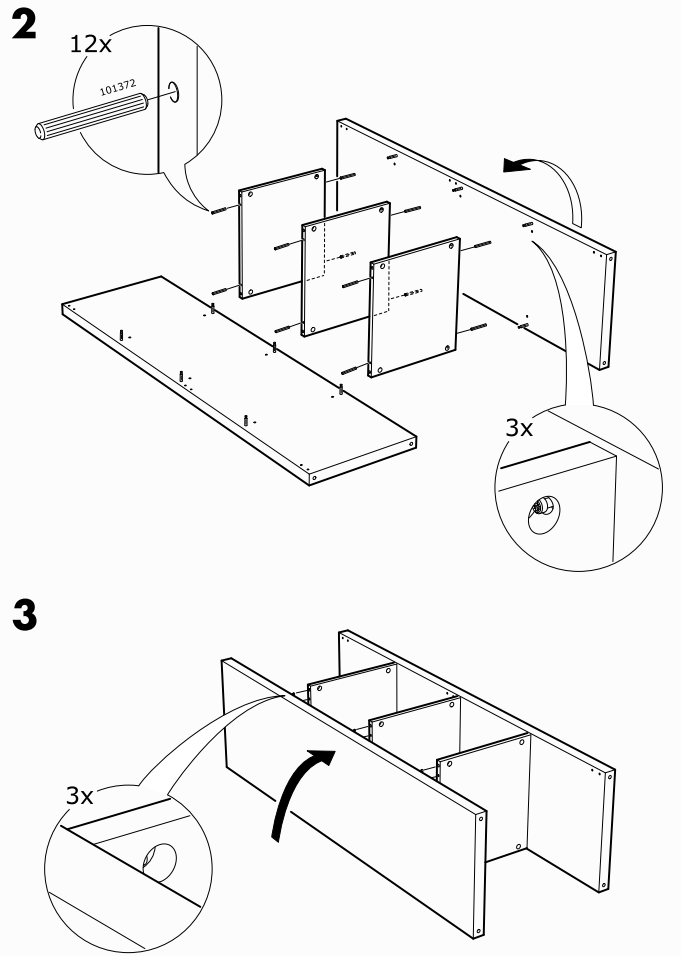
▷ Correspondant à une suite d'instructions (souvent répétitives)



9

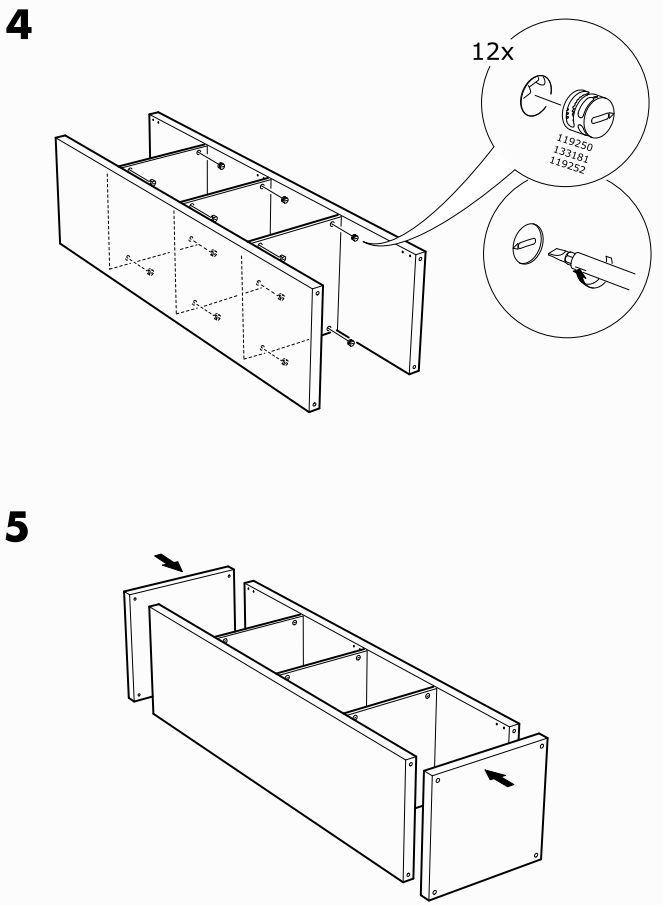
LES FICHIERS SOURCES

▷ Correspondant à une suite d'instructions (souvent répétitives)



LES FICHIERS SOURCES

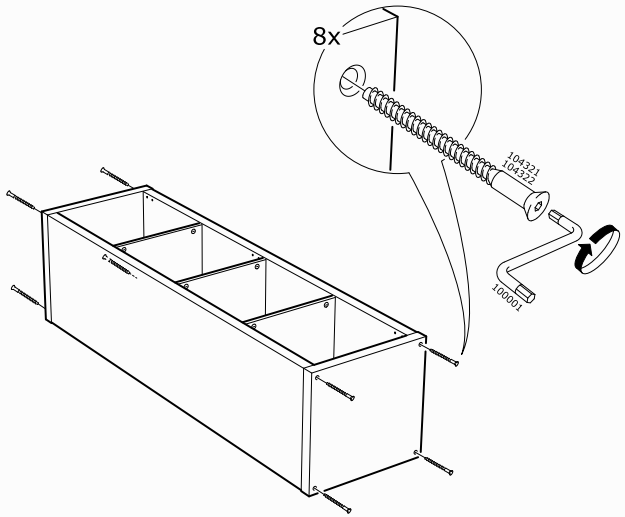
▷ Correspondant à une suite d'instructions (souvent répétitives)



LES FICHIERS SOURCES

▷ Correspondant à une suite d'instructions (souvent répétitives)

6



12

AA-1009545-7

LES FICHIERS SOURCES

▷ Fichiers textes contenant les instructions

▷ Extension spéciale .c

▷ Syntaxe très précise à respecter

▷ Séparateurs = espace + tabulation + retour à la ligne

▷ Le C est sensible à la casse

▷ Les séparateurs peuvent être librement utilisés pour la mise en page, car ils sont équivalents à un seul espace dans la plupart des cas

LES FICHIERS BINAIRES

▷ Un binaire est un fichier qui contient des instructions machines correspondant à une entité que l'on ne peut plus modifier mais que l'on peut utiliser dans un exécutable

LES FICHIERS EXÉCUTABLES

▷ Un exécutable est un binaire ayant un point de début d'exécution et pouvant correspondre à plusieurs fichiers binaires.

Des sources à l'exécutable

▷ La génération d'un exécutable à partir des fichiers sources se fait en plusieurs étapes, qui sont souvent automatisées à l'aide d'outils (e.g., make)

▷ Le compilateur est un programme qui lit des sources et produit un binaire possiblement exécutable

DES SOURCES À L'EXÉCUTABLE

- La compilation comporte 4 étapes :
  - 1 - pré-compilation (.c $\rightarrow$ .c)
  - 2 - compilation (.c $\rightarrow$ .s)
  - 3 - assemblage [‡] (.s $\rightarrow$ .o)
  - 4 - édition de liens (.o $\rightarrow$ exe)

[‡] Parfois regroupée avec la précédente par établissement d'un flux de données interne

1 - La pré-compilation

▷ La pré-compilation prend un ou plusieurs fichiers sources en entrée et produit un unique fichier source exempt de directives de pré-compilation (définies par des lignes commençant par ♯)

▷ Les fichiers sources sont analysés par le préprocesseur qui effectue des transformations purement textuelles

▷ Les macros `#define` sont interprétées et remplacées par leurs évaluations → chercher-remplacer

▷ Tous les `#include` sont remplacés par leurs contenus → copier-coller

La pré-compilation - exemple 1 - #define

```c
#define MAX 10
int main(){
  int i=0, j=0;
  for(i=0;i<MAX;++i){
    j+=i;
  }
  return j;
}
```

```c
int main(){
  int i=0, j=0;
  for(i=0;i<10;++i){
    j+=i;
  }
  return j;
}
```

- Directive de substitution : `#define NAME VALUE`
- Par convention, les noms de macro sont en majuscules

LA PRÉ-COMPILATION - EXEMPLE 2 - #INCLUDE

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

#define MESSAGE "hello, world!\n"

int main(void)
{
    printf(MESSAGE);
    return EXIT_SUCCESS;
}
```

```c
typedef signed char __int8_t;
typedef unsigned char __uint8_t;
typedef short __int16_t;
...
void perror(const char *);
int printf(const char * rest...
int putc(int, FILE *);
...
int main(){
  printf("hello, world!\n");
  return 0;
}
```

▷ Directive d'inclusion : `#include NOM_FICHIER`

▷ Les fichiers .h sont disponibles dans `/usr/include`

2 - COMPILATION

▷ Traduction du fichier généré par le préprocesseur en assembleur

▷ Instructions du micro-processeur dans un langage lisible (si, c'est vrai)

```
int main(void){
  int i=0, j=0;
  for(i=0;i<10;++i){
    j+=i;
  }
  return j;
}
```

```
 0: push   %ebp
 1: mov    %esp,%ebp
 3: sub    $0xc,%esp
 6: movl   $0x0,-0x4(%ebp)
 d: movl   $0x0,-0x8(%ebp)
14: movl   $0x0,-0xc(%ebp)
1b: movl   $0x0,-0x8(%ebp)
22: cmpl   $0xa,-0x8(%ebp)
29: jge    4a <_main+0x4a>
2f: mov    -0x8(%ebp),%eax
32: mov    -0xc(%ebp),%ecx
35: add    %eax,%ecx
37: mov    %ecx,-0xc(%ebp)
3a: mov    -0x8(%ebp),%eax
3d: add    $0x1,%eax
42: mov    %eax,-0x8(%ebp)
45: jmp    22 <_main+0x22>
4a: mov    -0xc(%ebp),%eax
4d: add    $0xc,%esp
50: pop    %ebp
51: ret
```

COMPILATION - OPTIMISATION

▷ Même code généré avec de l'optimisation.

0x2d = 45 = somme des nombres de 0 à 9

```
int main(void){
  int i=0, j=0;
  for(i=0;i<10;++i){
    j+=i;
  }
  return j;
}
```

```
0: push   %ebp
1: mov    %esp,%ebp
3: mov    $0x2d,%eax
8: pop    %ebp
9: ret
```

3 - ASSEMBLAGE - FICHIER OBJET

▷ Transformation du code assembleur en un fichier binaire (i.e., directement compréhensible par le processeur)

```
int main(void){
  int i=0, j=0;
  for(i=0;i<10;++i){
    j+=i;
  }
  return j;
}
```

```
<v1.o>
55 89 e5 83 ec 0c c7 45 fc
00 00 00 00 c7 45 f8 00 00
00 00 c7 45 f4 00 00 00 00
c7 45 f8 00 00 00 00 81 7d
f8 0a 00 00 00 0f 8d 1b 00
00 00 8b 45 f8 8b 4d f4 01
c1 89 4d f4 8b 45 f8 05 01
00 00 00 89 45 f8 e9 d8 ff
ff ff 8b 45 f4 83 c4 0c 5d
c3

<v2.o>
55 89 e5 b8 2d 00 00 00 5d
c3
```

▷ Généralement, la compilation et l'assemblage se font dans la foulée

4 - EDITION DES LIENS

▷ Un programme est souvent séparé en plusieurs fichiers source, pour des raisons de clarté mais aussi parce qu'il fait généralement appel à des librairies de fonctions standard déjà écrites

▷ Une fois chaque code source assemblé, il faut donc lier entre eux les différents fichiers objets en utilisant les symboles fournis (e.g., `printf`) et manquants de chaque fichier objet

▷ Lorsqu'un symbole est manquant dans un objet mais fournit dans un autre, alors les deux sont liés et le symbole est résolu

▷ L'édition de liens produit alors un fichier dit exécutable

LA COMPILATION EN PRATIQUE - GCC

- ▷ GNU Compiler Collection - GCC
- ▷ Compilation et assemblage

```
$>gcc HelloWorld.c -c
```

- ▷ Edition des liens

```
$>gcc HelloWorld.o -o HelloWorld
```

- ▷ Execution

```
$>./HelloWorld
hello, world!
```

OPTIONS DE COMPILATION

- `-std=c99` : définition de la norme à suivre
- `-Wall` : affiche tous les avertissement

```
int main(void){
  int i, j;
  for(i=0;i<10;++i){
    j+=i;
  }
  return j;
}
```

```
toto.c:4:5: warning: variable 'j' is
uninitialized when used here
    j+=i;
    ^
toto.c:2:11: note: initialize the
variable 'j' to silence this warning
  int i, j;
          ^
           = 0
1 warning generated.
```