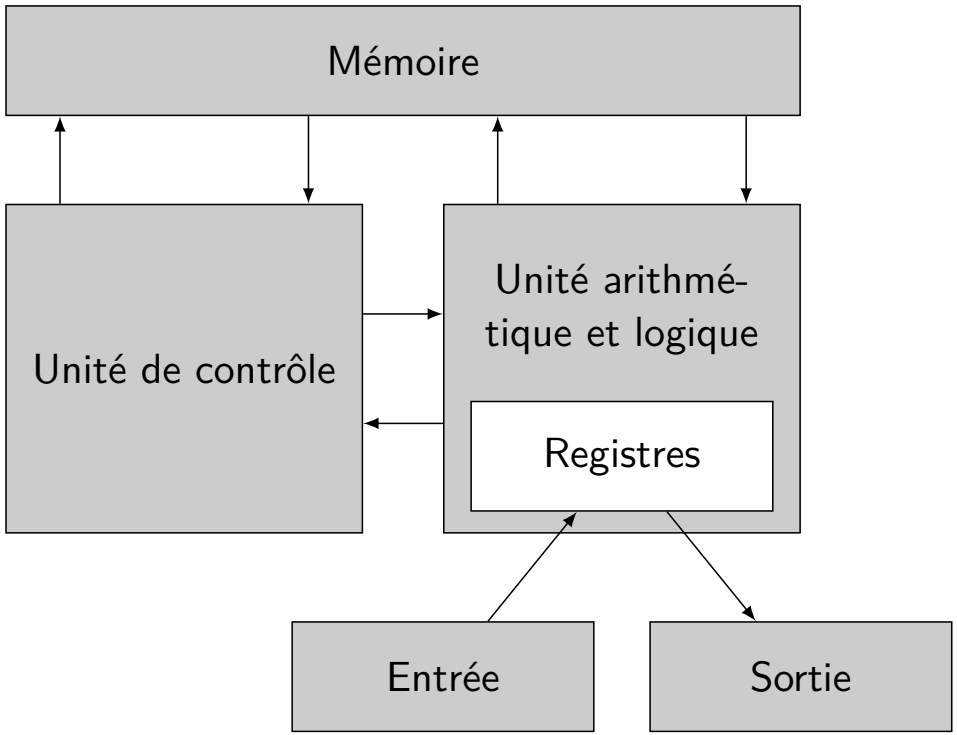
Algorithmique et Programmation 1 : les pointeurs

1re année Ensimag

https://algo-prog-1.pages.ensimag.fr/web/

Architecture de Von Neumann

- Un ordinateur est composé de façon simplifiée
  - d'un (ou plusieurs) processeur (UAL, PC, registres, ...)
  - de mémoire : de stockage (HD, SSD), vive (RAM), cache
  - de périphériques d'entrée (clavier, souris) et de sortie (écran)



La mémoire vive (*Random Access Memory*)

- sert à stocker des données et exécuter des programmes
- *Random Access* accès ~~aléatoire~~ direct
  - i.e. on peut accéder à n'importe-quelle case directement
  - par opposition à un accès séquentiel (ex. bande magnétique)

Représentation de la mémoire

- La mémoire peut être vue comme un tableau de mots
  - chaque case contient une valeur dans un certain intervalle
    - dit autrement : un mot d'une certaine taille
  - chaque case a un numéro appelé indice (ou index)
    - on parle d'*adresse mémoire*

Valeur	**45**	**154**	**58**	**78**	**31**	**5**	**74**
Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**

Un mot mémoire = une valeur « unitaire » d'une certaine taille

- On parle de taille d'un mot en bit (*Binary digIT* : 0 ou 1)
  - un mot d'un octet est composé de 8 bits
  - autres tailles courantes : 16, 32 ou 64 bits

Valeur codées dans des mots

- Un octet permet de représenter par exemple
  - un entier signé (relatif) dans $[-2^7..2^7 - 1]$ (`int8` : -128..127)
  - un entier non signé (naturel) dans $[0..2^8 - 1]$ (`uint8` : 0..255)
  - mais aussi un caractère, un morceau d'un flottant, ...

Question

- Une case mémoire contient la valeur binaire 11111111.
  S'agit-il de la valeur -1, 255 ou du caractère ÿ en Latin-1 ?

Rappels sur les variables

- On a déjà utilisé des variables
  - scalaires (`bool`, `int`, `float64`)
  - composées (`struct`, tableaux)
- Nom de variable = symbole représentant l'adresse de la donnée, c'est le compilateur qui fait la traduction
- Le type de la variable détermine sa taille en octets
  - `int8`, `byte` ≡ `uint8`, `bool` : 1 octet
  - `int32` ≡ `rune`, `uint32` : 4 octets
  - `int64`, `uint64`, `float64` : 8 octets
  - Cas particulier : `int`, `uint` : 4 ou 8 octets (selon l'archi)
  - Un tableau statique : `len(tab)` × taille d'un élément
  - Une structure : parfois plus que la somme des tailles des champs (*padding* : cf. ILM)
  - ...

Variables pointeurs

- Un pointeur est une variable dont la valeur est l'adresse d'une autre variable (plus généralement, d'une case mémoire)
- En pratique : un entier dans [0..*TAILLE_MEM*[
  - mais attention, on ne peut pas s'en servir comme un entier en Go (pas d'arithmétique des pointeurs comme en C)

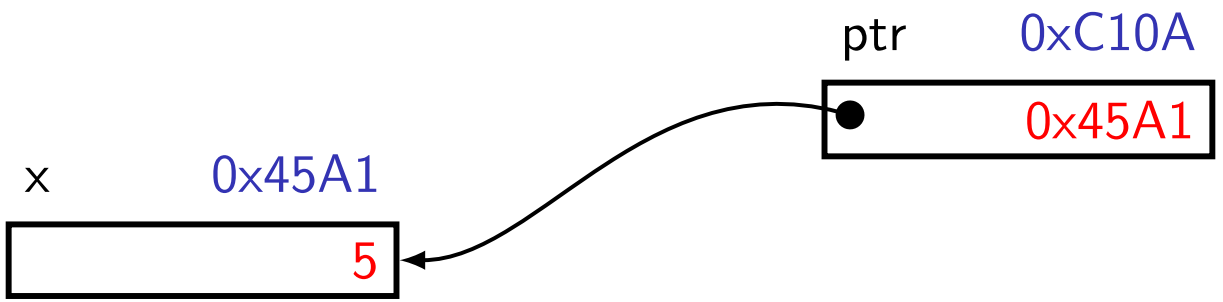
```
var ptrOctet *uint8
var ptrEntier *int32
var ptrReel *float64
var ptrComplexe *nombreComplexe
```

- Un pointeur est toujours défini par rapport au type de la variable pointée

Initialisation d'un pointeur

- On peut récupérer l'adresse d'une variable avec l'opérateur &

```
var x int = 5
var ptr *int
ptr = &x
```



Déréférencement d'un pointeur

- On récupère la valeur de la variable pointée avec l'opérateur *
- **Ne pas confondre avec** `var ptr *int`

```
fmt.Println(x, "==", *ptr) // affiche "5 == 5"
```

Utilisation des pointeurs

- On a déjà utilisé souvent des pointeurs sans le savoir

```
// func Open(name string) (*File, error)
fichier, err := os.Open("mon_fichier.txt")
```

  - `fichier` est un pointeur vers une `type File struct`
  - cette `struct` est peut-être de très grande taille (non)
- Un pointeur a une taille fixe dépendant de l'architecture
  - exemple : processeur x86_64 $\Rightarrow$ pointeur sur 64 bits
- Important si le langage pratique le passage de paramètre par copie pour les fonctions

Passage de paramètres aux fonctions en Go

- Passage par copie ≡ paramètres recopiés à chaque appel

```
func afficher(tab [100_000]float64) {
    fmt.Println(tab)
}

func main() {
    var monTab [100_000]float64 // init à 0.0
    for idx := 0; idx < 1_000_000; idx++ {
        afficher(monTab)
    }
}
```

- tableau recopié à chaque appel à `afficher` ... et à `Println`

⇒ pas efficace et gaspille de la mémoire

Passage de paramètres aux fonctions en Go

- Passage par pointeur

```
func afficher(tab *[100_000]float64) {
    fmt.Println(*tab) // * nécessaire sinon
                      //   ça affiche &[0 0...0 0]
}

func main() {
    var monTab [100_000]float64 // init à 0.0
    for idx := 0; idx < 1_000_000; idx++ {
        afficher(&monTab)
    }
}
```

- on recopie un pointeur sur 64 bits = 8 octets

Conséquence du passage par copie

```
func echanger(x int, y int) {
    x, y = y, x
}

func main() {
    a := 2
    b := 5
    echanger(a, b)
    fmt.Println("a =", a, "b =", b)
}
```

Question

- Qu'affiche ce code ?

Conséquence du passage par copie

```
func echanger(x *int, y *int) {
    *x, *y = *y, *x
}

func main() {
    a := 2
    b := 5
    echanger(&a, &b)
    fmt.Println("a =", a, "b =", b)
}
```

Question

- Qu'affiche ce code ?

Organisation mémoire

- La mémoire est partagée entre les différents programmes (processus)
  - le système d'exploitation réserve de la mémoire lorsqu'on lance un processus
  - il la libère quand l'exécution du processus se termine
- La mémoire allouée à un processus est elle-même souvent divisée en 4 grandes zones
  - la zone `text` contenant le code du programme
  - la zone `data` contenant les données statiques (en Go : des constantes)
  - la pile d'exécution (*stack*) qui sera détaillée en cours d'ILM
  - le tas (*heap*) qui contient des données dynamiques (la plupart des variables en Go)
- C'est la gestion de cette zone du tas qui nous intéresse ici

Gestion de la mémoire

Deux concepts fondamentaux

- « Allouer » signifie réserver une zone pour y stocker des données
  - `var x, y int32` réserve deux zones de 32 bits chacune
- « Désallouer » signifie libérer cette zone quand on n'en a plus besoin
  - sinon on pourrait saturer la mémoire allouée au processus

Implantation basique en C

- `ptr = malloc(16)` alloue 16 octets et renvoie un pointeur sur le début de la zone allouée
- `free(ptr)` libère cette même zone mémoire
- Cette gestion basique et « à la main » entraîne souvent des erreurs. . .

Ramasse-miettes – *Garbage Collector*

- Un GC est un programme qui désalloue des zones mémoires
- Le GC détermine quelles zones ne sont plus utilisées
- En Go, on allouera donc de la mémoire, mais on ne la désallouera jamais

```
func fct() *int {
    x := 5
    y := x + 1
    return &x
}
```

- `y` n'est plus utilisée en dehors de `fct` $\Rightarrow$ désallouable
- mais pas `x` dont on renvoie l'adresse à la fonction appelante
  - on dit que `x` s'échappe de la fonction `fct`

Allocation mémoire en Go

- On sait récupérer un pointeur vers une variable

```
var x int32
ptr := &x
```

- On peut aussi allouer directement une zone mémoire et récupérer son pointeur avec la fonction `func new(Type) *Type`

```
ptr := new(int32)
```

- Dans les deux cas, la zone mémoire de 32 bits est initialisée à 0 (valeur nulle du type `int32`)
- On comprendra l'utilité quand on utilisera des listes chaînées
- Il existe aussi une fonction `make` qui sert notamment à allouer **et initialiser** des *slices*

Définition

- Un *slice* est une structure de données contenant
  - un pointeur vers le tableau contenant les données
  - une taille **`len`**, le nombre d'éléments dans le tableau
  - une capacité **`cap`**, l'espace total allouée pour le tableau
  - on a donc forcément $len \leq cap$
- Passage efficace par copie : 2 entiers + 1 pointeur
- Les *slices* sont des tableaux dynamiques
  - **`len`** et **`cap`** peuvent évoluer pendant l'exécution, alors que :

```
var tab [10]int // tableau STATIQUE (tab [0]int aussi !)
fmt.Println(len(tab), cap(tab)) // affiche "10 10"
// len et cap sont constants et toujours égaux
```

$\Rightarrow$ **`len`** n'a pas vraiment de sens pour un tableau statique : impossible de « calculer » le nombre d'éléments significatifs

Allocation d'un *slice*

- On crée un *slice* avec la fonction `make`

```
var slc []int = make([]int, 3)
fmt.Println(slc, len(slc), cap(slc)) // [0 0 0] 3 3

var slc1 []int // fait juste un new, pas un make !
fmt.Println(slc1, len(slc1), cap(slc1),
            slc1 == nil) // [] 0 0 true
var slc2 []int = make([]int, 0)
fmt.Println(slc2, len(slc2), cap(slc2),
            slc2 == nil) // [] 0 0 false
slc3 := []int{} // syntaxe compacte
fmt.Println(slc3, len(slc3), cap(slc3),
            slc3 == nil) // [] 0 0 false
slc4 := []int{1, 2, 3}
fmt.Println(slc4, len(slc4), cap(slc4)) // [1 2 3] 3 3
```

Utilisation transparente des *slices*

- On a dit qu'un *slice* est une `struct`
- Mais on écrit bien `slc[2] = 5` et pas `slc.tab[2] = 5`

  $\Rightarrow$ c'est le langage qui cache l'accès aux champs de la `struct`

Dynamicité des *slices*

- La fonction `append` ajoute des éléments à la fin du *slice*

```
slc := make([]int, 1)       // [0]
slc2 := []int{4, 5, 6} // alloue ET init [4 5 6]
slc = append(slc, 1)        // [0 1]
slc = append(slc, 2, 3)     // [0 1 2 3]
slc = append(slc, slc2...) // [0 1 2 3 4 5 6]
```

- `slc2...` « déplie » le *slice* en suite de paramètres entiers
- sans les `...`, c'est faux (erreur de typage)

Augmentation de la capacité

- La capacité d'un *slice* peut augmenter (mais pas diminuer)

```
slc := make([]int, 1)
fmt.Println("cap =", cap(slc), "len =", len(slc),
            "contenu =", slc)
for i := 1; i <= 3; i++ {
    slc = append(slc, i)
    fmt.Println("cap =", cap(slc), "len =", len(slc),
                "contenu =", slc)
}
// cap = 1 len = 1 contenu = [0]
// cap = 2 len = 2 contenu = [0 1]
// cap = 4 len = 3 contenu = [0 1 2]
// cap = 4 len = 4 contenu = [0 1 2 3]
```

Mais combien ça coûte ?

- `append` a un coût *amorti* constant
- Tant qu'il reste de la place, coût constant
- Quand on doit augmenter la capacité d'un *slice*
  - on alloue un nouveau tableau interne de la nouvelle `cap`
    - $\not\equiv$ `realloc` du C qui peut (parfois) juste agrandir la zone
  - on copie le contenu de l'ancien tableau dans le nouveau : $O(N)$
  - MAIS ça arrivera de moins en moins souvent !

Questions

- Il existe en Go des fonctions équivalentes à
  - `inserer(slc, val, idx)` insère `val` à l'indice `idx` de `slc`
  - `val = retirer(slc, idx)` supprime `slc[idx]` et la renvoie
- Quel est le coût (à la louche) de ces fonctions ?
- Qu'est-ce qui va se passer si on les utilise dans des boucles ?

Passage de *slices* en paramètre d'une fonction

- En Go, **tous les paramètres sont passés par copie**

```
func fctOK(tab []int) {
    tab[0], tab[len(tab)-1] = tab[len(tab)-1], tab[0]
}
func fctKO(tab []int) {
    tab = append(tab, 6)
}
func main() {
    tab := []int{1, 2, 3, 4, 5}
    fmt.Println(tab) // [1 2 3 4 5]
    fctOK(tab)
    fmt.Println(tab) // [5 2 3 4 1]
    fctKO(tab)
    fmt.Println(tab) // [5 2 3 4 1] : koa ?
    fmt.Println(len(tab), cap(tab)) // 5 5 : gni ?
}
```

Passage de *slices* en paramètre d'une fonction

- Rappel : un *slice* d'entiers en Go est

```
struct {
    len int
    cap int
    data *[]int // pointeur vers le tableau statique
        contenant les données, ce tableau est alloué
        dans le tas (zone mémoire globale)
}
```

⇒ si on change `len` ou `cap` d'un *slice* passé en paramètre d'une fonction, on change **leur copie**

⇒ on peut changer les données : `tab[idx] = tab[idx+1]`

- modification en place comme dans les tris ou le pivot

Les tranches de *list* Python

- En Python, on a l'habitude d'écrire

```
tab1 = [1, 2, 3, 4, 5, 6, 7]
print(tab1) // [1, 2, 3, 4, 5, 6, 7]
tab2 = tab1[2:5] // intervalle ouvert à droite
print(tab2) // [3, 4, 5]
tab2.append(-1)
print(tab2) // [3, 4, 5, -1]
print(tab1) // [1, 2, 3, 4, 5, 6, 7]
```

- Mais qu'est-ce que ça implique ?

En Python, quand on utilise une tranche d'une *list*

- Python alloue une nouvelle *list*
- Puis il recopie les éléments de l'intervalle (*shallow copy*)

$\Rightarrow$ les deux *list* sont totalement indépendantes

Les tranches de *slices* (pléonasme)

- En Go, ça ne va pas marcher comme prévu...

```
tab1 := []int{1, 2, 3, 4, 5, 6, 7}
fmt.Println(tab1) // [1 2 3 4 5 6 7]
tab2 := tab1[2:5]
fmt.Println(tab2) // [3 4 5]
tab2 = append(tab2, -1)
fmt.Println(tab2) // [3 4 5 -1] : tout va bien...
fmt.Println(tab1) // [1 2 3 4 5 -1 7] : aïe !!
```

En Go, quand on utilise une tranche d'une *slice*

- Il alloue une nouvelle **`struct`** avec dedans
  - le pointeur sur le 1er élément de la tranche (ici 3)
  - **`len`** : le nombre d'éléments de la tranche (ici 3)
  - **`cap`** : la capacité jusqu'à la fin de la zone initiale (ici 5)

$\Rightarrow$ une tranche de *slice* est une fenêtre sur les données initiales

Tu bluffes Martoni, c'est encore plus compliqué que ça !

```
tab1 := []int{1, 2, 3, 4, 5, 6, 7}
fmt.Println(tab1) // [1 2 3 4 5 6 7] (cap = 7)
tab2 := tab1[2:5]
fmt.Println(tab2) // [3 4 5] (cap = 5)
tab2 = append(tab2, -1, -2, -3)
fmt.Println(tab2) // [3 4 5 -1 -2 -3] (cap = 10)
fmt.Println(tab1) // [1 2 3 4 5 6 7] (cap = 7)
```

- `cap(tab2)` ne permettait pas d'ajouter trois valeurs
- On doit de toute façon réallouer/recopier, autant dissocier

Comportement de `append` en Go

- Ça dépend de ce qu'on ajoute et de la `cap` initiale !
- **Ça peut même changer d'une exécution à l'autre !**

Conseil pour les tranches de *slices*

- C'est pratique pour sélectionner une partie d'un `slice`
  - ou d'une `string`, qui est en fait un *slice* non-modifiable
- Ça ne pose pas de problème tant qu'on ne fait que lire les valeurs dans la tranche
  - ex. : afficher une sous-partie d'un tableau
- Ou à la limite, si on ne modifie que les valeurs à l'intérieur de la tranche
  - ex. : échanger des éléments dans une sous-partie d'un tableau
- Ne faites **jamais** un `append` sur une tranche
- Il existe une fonction Go qui marche comme en Python

```
res := slices.Concat([]int{0, 1, 2, 3}, []int{4})
fmt.Println(res) // [0 1 2 3 4]
```