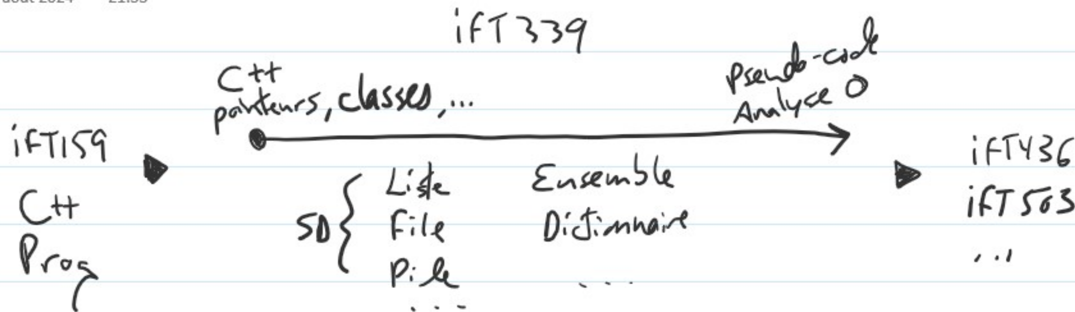
Intro

25 août 2024 21:55



1) Comprendre l'implémentation des structures de données
   avantages + inconvénients
2) Choisir la bonne structure pour une tâche

__Structure de données__ : organisation de données pour réaliser certaines tâches

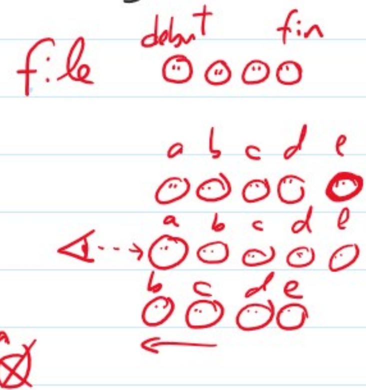
On distingue les types de données abstraits (TDA) de leur implémentation (représentation)

TDA = tâches à réaliser   Impl = façon de les réaliser

ex: TDA __file__

élément

Tâches Ajouter elt à la fin
        Voir elt au début
        Retirer elt au début



Impl. 1: tableau C++

Impl. 2: liste chainée

...

TDA ensemble

Tâches ajouter elt
supprimer elt
voir si elt présent ou non

Impl: tableau, arbre de recherche, table de hachage, ...

TDA : entier (représente un nombre entier)

Tâches: + - * ÷ % etc.

Impl.: en C++, plusieurs impl.
int, long, unsigned int
uint32_t, uint64_t, ...

Adresse mém	Contenu	Var
0x1234	00000101	x

`int x=5;` Réserve un espace mémoire et y stocke 5 en binaire (voir ift209).

`x=10;` Copie "10" dans cette zone.

	00001010	x

`int y;` Mém. non-initialisée.

	Contenu	
	⋮	
0x1234	00001010	x
	⋮	
0x2111	01101110	y
	⋮	

`y=x;` Copie le contenu de x dans la mém

00001010	x
:	

y = x;   Copie le contenu de x dans la mém de y

0000 1010 x
⋮
0000 1010 y

x = 100;   Ne modifie pas y.

0110 100 x

0000 1010 y

Important: en C++, l'assignation = fait toujours une copie.

```
a = b;
```

fait une copie de b et a deviens cette copie, même si a et b sont des objets complexes.

```
vector<int> a, b;
...
a = b;   // copie tout le vecteur b.
```

Passage par valeur/référence

Si on appelle une fct C++:

```
void f(int x){      //passage par valeur
    x = 5;
}
```

```
int a=10;
f(a);
std::cout<<a;
```

l'argument x est <u>copié</u>. Le a n'est <u>pas</u> modifié car x est une copie de a.

Pour éviter la copie, on passe par référence avec &

```
void f(int& x){
    x = 5;
}
int a=10;
f(a);
cout<<a;    // 5
```

Le & indique qu'on passe la zone mémoire de l'argument directement, sans copie.

++ rapide, surtout avec objets complexes

-- crée des bugs (effet de bord)

- Le & sert aussi à obtenir l'adresse d'une var x (si x utilise plsrs mots mémoire, donne le 1er).

```
int x=10;
```

```
cout << &x;    // 0x12AB
```

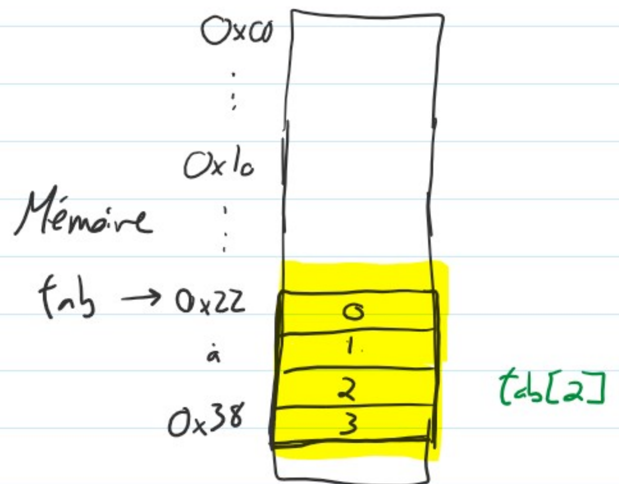
- Voir sizeof(x), nb d'octets

TDA : nombre réel
Impl : float, double

TDA : chaîne de caractères
Impl. : #include <string>
string x = "allo";

TDA : tableau de taille fixe

En C++ : int tab[4];



- Réserve un espace contigu pour 4 int (4 × sizeof(int)) à une certaine adresse x
- tab est un pointeur : il contient l'adresse où l'espace contigu débute.
- Ensuite, l'instruction
  tab[i]
  1) va à l'adresse début où tab "pointe"
  2) avance de i élts (i × sizeof(int))
  3) retourne une réf. sur l'élt à cette pos

2) avance de i elts (i * sizeof(int))
3) retourne une réf. sur l'elt à cette pos

```
tab[2]=10;              //modifie à la pos x+2

std::cout<< tab[2];     //10

int v = tab[2];
v=5;
std::cout << tab[2];    //10

cout<<tab;              //0x1234
```

Passage de tableau

```
void f(int t[4]){
    t[2]=0;
}

int tab[4];
tab[2]=10;

f(tab);

cout<< tab[2];
```

Affiche 0 car le type de tab (et t) est "adresse vers une zone mémoire",

et non " tableau de taille 4 ".

Donc f(tab) copie l'<u>adresse</u> de tab.

Donc t pointe à la même place que tab,
et donc

t[2]

accède à la même place que

tab[2]