Pimp A - info121A

Programmation IMPérative Avancée

Frédéric Vernier

Programme

- *Tri*
- *Récursivité*
- *Dichotomie / Pointeurs*
- *Pointeurs / listes chainées*
- *Arbre*
- *Examen blanc*

Pointeurs : opportunité

- ***Toute variable possède un nom, une valeur (et un type).***
- *Toute variable est stockée en mémoire*
  - *Elle possède donc un numéro d'adresse*
  - *En interne le compilateur remplace les noms de variable par ce numéro*
  - *i=12; => adresse[285064207] =12;*

Pointeurs : définition

- *Un **pointeur** est en programmation une variable contenant une **adresse mémoire**.*

Pointeurs : super pouvoir

- *Si vous pouvez manipuler une variable par son adresse mémoire, vous pourrez*
  - *Vous affranchir de son type*
  - *Partager une variable (fonctions)*
  - *Pointeur de pointeur (jeu de piste)*
  - *Lire/Modifier le pointeur OU la variable au bout du pointeur*

Pointeur : 5 ième super pouvoir

- *Associer un pointeur à une variable« normale »*
- *Exemple*

	X=?	Y=000000000000
3495877563	i=12	Pt1=**3495877**567
3495877565	j=27	Pt2=1375878568
3495877567	h=11	Pt3=2945671567
	????	02579538247458

3495877564 *3495877566* *3495877568*

- *Le pointeur « à coté» de i pointe sur h*
- *Le pointeur « à coté» de j pointe sur ?*

Pointeur : propriété

- *2 variables déclarées « ensemble » dans une structure ou un tableau sont mises l'une après l'autre en mémoire*
- *Conséquence : si vous avez un pointeur p qui pointe vers la première variable l'instruction p++; modifie le pointeur qui pointe ensuite vers la 2ieme variable*
- *PS : si les pointeurs sont associés à une variable il faut plutôt faire p = p+2;*

Pointeur : Danger

- *Toutes les valeurs de pointeurs sont valides*

	X=?	Y=000000000000
3495877563	i=12	Pt1=**3495877567**
3495877565	j=27	Pt2=1375878568
3495877567	h=11	Pt3=2945671567
3495877569	????	02579538247458

3495877564 *3495877566* *3495877568*

- *Pt1++ pointe vers Pt3 (mauvais type)*
  - *Mais Pt1 est modifiée*
- *Pt1=Pt1+2 pointe vers ???? (zone mémoire jamais utilisée)*

The questions

- *Comment modifier la valeur au bout du pointeur ?*
  - *Toute modification de p modifie l'adresse*
- *Comment connaître le type de la variable*
  - *+entier ≠+string, division entière/flottante…*

Pointeur : notation C/C++

- *Déclaration avec une étoile = pointeur*

```
int *ptr; // ptr est de type int*
int i =12;
```

- *Utilisation du symbole & devant une variable = son adresse*

```
ptr = &i; // ca y est ptr pointe vers i !
```

- *Utilisation du nom de la variable avec étoile = la valeur au bout du pointeur*

```
*ptr   = 13; // on modifie i sans utiliser i !
ptr[0] = 13; // pareil. Permet de modifier derrière
```

Allocation dynamique

- *Si on peut modifier une variable sans utiliser son nom … pourquoi lui donner un nom ?*
- *malloc() alloue une zone mémoire, sans lui donner de nom ni de type.* *(Retourne un pointeur void*)*

```
int* ptr = (int*)malloc(4); // encore mieux malloc(sizeof(int));
*ptr = 27; ptr[2] = 13;
```

- *malloc(sizeof(X)) alloue une zone mémoire de taille parfaite pour une structure X*

Alloc. Dynamique 2/2

- *La mémoire n'est pas libérée* ***automatiquement*** *(contrairement aux variables locales)*
- *free(ptr) libère la mémoire (votre resp. !)*
- *Allocation en octets (8 bits !) un entier = 32 bits = 4 octets !*

Allocation de plusieurs variables (>tableaux)

```
float*ptr = malloc(n*sizeOf(float)); // n reels
ptr[1] = 13.7; // modifie la valeur du 2ieme nombre
```

Manipulation de structures

```
struct Pers{
  int age;
  string nom;
};
```

- *Si vous avez une variable a de type Pers*

```
Pers a;
a.age = 23;
```

ATTENTION: si a est déclarée dans un fonction. La mémoire est libérée à la sortie de la fonction

- *Si vous avez un pointeur ptr vers une variable de type Pers*

```
Pers *ptr = (Pers*)malloc(sizeof(Pers));
ptr->age = 23;
```

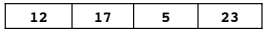
ATTENTION: si ptr est déclarée dans un fonction. La mémoire n'est pas libérée à la sortie de la fonction = oui return pers;

Sauf que …

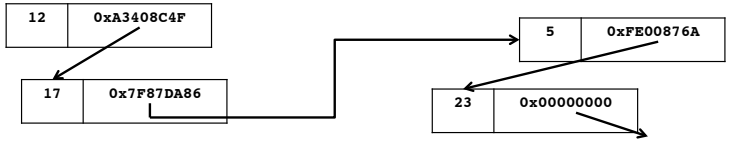
- *Les structures doivent avoir une taille fixe pour que le malloc fonctionne*
  - *Quelle est la taille d'une string ?*
  - *Solution = replacer string par string **

Listes chainées

Listes chainées : motivation

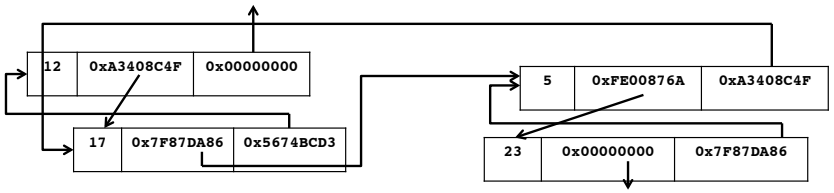


- *Tableaux = taille fixe à la compilation*
- *Alloc dynamique = taille fixe à l'allocation*
- *Liste chainée = taille vraiment dynamique*



Listes chainées

- *Ensemble d'éléments contenant chacun des informations ET un/des pointeur(s) vers les éléments suivants/précédents*
- *Exemple : 1 valeur entière, 2 pointeurs*



Propriétés

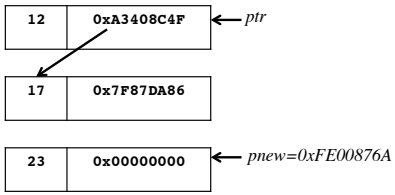
- *Une seule structure, allouée n fois*
- *Pointeur 0x00000000 pour arrêter la liste*
- *Aucune garantie à la compilation de cohérente (suivant du précédent = lui-même)*

Avantages / Inconvénients

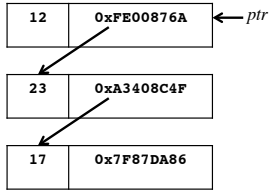
- *Avantages*
  - *L'insertion et la suppression en O(1)*
  - *Une taille vraiment dynamique*
- *Inconvénients*
  - *+1 pointeur par élément*
    - **5 pour des caractères sur 1 octet par exemple !*
  - *Cohérence à vérifier*

Insertion

- *Avant*



- *Apres*



pnew->next = ptr->next->next;

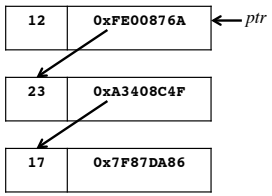
ptr->next = pnew;

Insertion 2/2

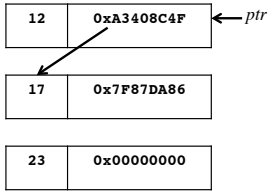
- *Insertion en tête de liste ?*
  - *Et si une autre partie du code avait une copie de ptr ?*
- *Insertion en queue de liste ?*
  - *On doit tout parcourir ?*
- *Insertion dans une liste double chainée ?*
  - *En tête, en queue …*
- *Insertion ordonnée (garantie l'ordre)*

Suppression

- *Avant*



- *Apres*




```
void *tmp = ptr->next;
ptr->next = ptr->next.next;
free(tmp);
```

Suppression 2/2

- *Tester avant de supprimer !*
- *Suppression en tête*
- *Suppression en queue*

Structure supplémentaire

- *Structure de liste chainée*
  - *Pointeur vers le premier*
  - *(pointeur vers le dernier)*
  - *Taille de la liste*

Commentons ce code internet 1/3

```
#include <stdlib.h>

typedef struct element element;
struct element
{
    int val;
    struct element *nxt;
};

typedef element* llist;
```

http://sdz.tdct.org/sdz/les-listes-chainees-2.html

Commentons ce code internet 2/3

```
int main(int argc, char **argv) {
    // Déclarons 3 listes chaînées de
    //façons différentes mais équivalentes
    llist ma_liste1 = NULL;
    element *ma_liste2 = NULL;
    struct element *ma_liste3 = NULL;

    return 0;
}
```

Les typedef rendent les choses plus difficiles à lire au début

Commentons ce code internet 3/3

```
llist ajouterEnTete(llist liste, int valeur) {
    /* On crée un nouvel élément */
    element* nouvelElement = malloc(sizeof(element));

    /* On assigne la valeur au nouvel élément */
    nouvelElement->val = valeur;

    /* On assigne l'adresse de l'élément suivant au nouvel
élément */
    nouvelElement->nxt = liste;

    /* On retourne la nouvelle liste, i.e. le pointeur sur
le premier élément */
    return nouvelElement;
}
```

Encapsulation … comment utiliser la fonction ?

Conclusion

- *Avantages*
  - *Insertion sans effet de bord O(n)*
  - *Suppression sans effet de bord O(n)*
- *Inconvénients*
  - *Taille mémoire*
  - *Accès nieme/ permutation très couteux*