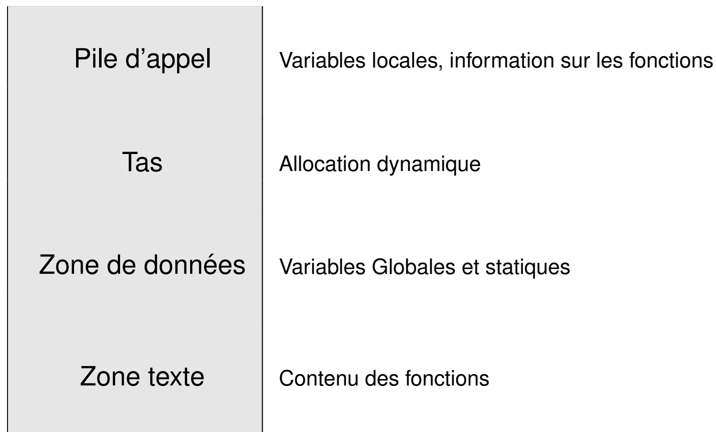
Cours de Programmation Impérative: Zones de mémoires et pointeurs

Julien David

A101 - david@lipn.univ-paris13.fr

Zone

Zones de mémoire



Zone texte

Rappel

Un fichier exécutable contient déjà :

- Une zone correspondant aux variables globales et statiques.
- Une zone correspondant à l'ensemble des fonctions décrit dans le programme.

À l'exécution du programme

- L'ensemble des fonctions est chargée en mémoire dans la **zone texte**.
- Chaque instruction de chaque fonction possède une adresse propre en mémoire.

Zone texte

- Cette zone est en lecture seule.
- Sa taille est donc déterminée avant l'exécution.

Zone texte

Rappel

Un fichier exécutable contient déjà :

- Une zone correspondant aux variables globales et statiques.
- Une zone correspondant à l'ensemble des fonctions décrit dans le programme.

À l'exécution du programme

- L'ensemble des fonctions est chargée en mémoire dans la **zone texte**.
- Chaque instruction de chaque fonction possède une adresse propre en mémoire.

Zone texte

- Cette zone est en lecture seule.
- Sa taille est donc déterminée avant l'exécution.

Zone texte

Rappel

Un fichier exécutable contient déjà :

- Une zone correspondant aux variables globales et statiques.
- Une zone correspondant à l'ensemble des fonctions décrit dans le programme.

À l'exécution du programme

- L'ensemble des fonctions est chargée en mémoire dans la **zone texte**.
- Chaque instruction de chaque fonction possède une adresse propre en mémoire.

Zone texte

- Cette zone est en lecture seule.
- Sa taille est donc déterminée avant l'exécution.

Zone de données

À l'exécution du programme

- La zone de données de l'exécutable est recopié dans la mémoire.
- La taille de cette zone est déterminée avant l'exécution.

Accès permanent

- Les variables dans la zone de donnée sont accessibles durant toute la durée du programme.
- L'utilisation de la mémoire n'est donc pas optimisée.

Pile d'appel

Caractérisation

La pile d'appel permet de :

- stocker les variables locales,
- stocker les paramètres des fonctions,
- sauver des informations sur les appels de fonctions.

Précision

Dans ce cours on représente les appels de fonctions avec une zone grisée dans la pile.
Cette zone contient des informations permettant, à la fin d'une fonction,

- de dépiler la pile,
- de revenir à la ligne de l'instruction qui a provoqué l'appel de fonction.

Les limites de la pile d'appel

Limites

- La taille de la pile
- Les limites d'accès des variables locales

Les limites de la pile d'appel

La taille de la pile

- La pile a une taille limitée.
- À chaque appel de fonction, on ajoute des informations dans la pile.
- Si un programme effectue trop d'appels imbriqués, la pile déborde.

Les limites de la pile d’appel

Les limites d’accès des variables locales

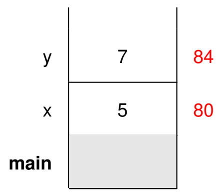
Avec nos connaissances actuelles :

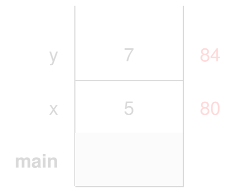
- Il n’est pas possible de modifier une variable locale d’une fonction *A* avec une fonction *B*.
- Exception faite du résultat de la fonction *B*.

Exemple

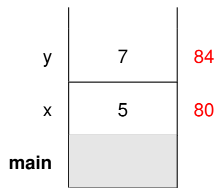
Essayons d’écrire une fonction qui échange la valeur de deux variables.

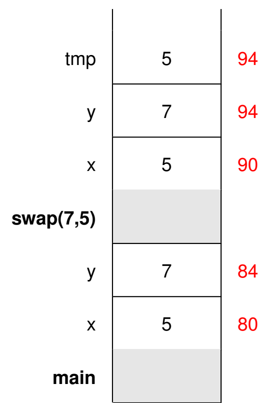
```
1 void swap(int x,int y){
2   int tmp=x;
3   x=y;
4   y=tmp;
5 }
6 int main(){
7   int x=5;
8   int y=7;
9   swap(x,y);
0   return EXIT_SUCCESS;
1 }
```

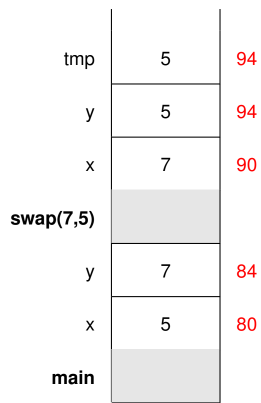


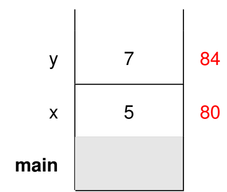


```
1  void swap(int x,int y){
2    int tmp=x;
3    x=y;
4    y=tmp;
5  }
6  int main(){
7    int x=5;
8    int y=7;
9    swap(x,y);
0    return EXIT_SUCCESS;
1  }
```









Constat

Notre accès à la mémoire est pour l'instant très limité.
Une fonction ne peut accéder qu'à ses propres variables locales ou à des variables de la zone de données.

Le

Pointeurs

Rappel

Une variable est caractérisé par :

- son nom
- son type
- sa valeur
- son adresse

Adresse d'une variable

Soit une variable `int x;`
Pour accéder à l'adresse de x, on écrit : &x

Pointeurs

Les pointeurs

- Un **pointeur** est une variable dont **la valeur est une adresse**.
- Si un pointeur contient l'adresse d'une variable, on dit *qu'il pointe sur elle*.
- Le type d'un pointeur est le type des variables sur lesquelles il peut pointer.

Pour faire quoi ?

En stockant l'adresse d'autres variables, on va pouvoir accéder n'importe qu'elle zone de la mémoire.

Pointeurs

Les pointeurs

- Un **pointeur** est une variable dont **la valeur est une adresse**.
- Si un pointeur contient l'adresse d'une variable, on dit *qu'il pointe sur elle*.
- Le type d'un pointeur est le type des variables sur lesquelles il peut pointer.

Pour faire quoi ?

En stockant l'adresse d'autres variables, on va pouvoir accéder n'importe qu'elle zone de la mémoire.

Pointeurs

Les pointeurs

- Un **pointeur** est une variable dont **la valeur est une adresse**.
- Si un pointeur contient l'adresse d'une variable, on dit *qu'il pointe sur elle*.
- Le type d'un pointeur est le type des variables sur lesquelles il peut pointer.

Pour faire quoi ?

En stockant l'adresse d'autres variables, on va pouvoir accéder n'importe qu'elle zone de la mémoire.

Pointeurs

Déclaration

`int * p;`

- Nom : `p`
- Type : pointeur sur un entier (`int *`)

`char * c;`

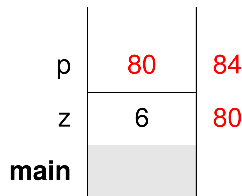
- Nom : `c`
- Type : pointeur sur un caractère (`char c`)

Utilisation

Soit un pointeur `int * z`
Pour accéder à la case pointée par `z`, on écrit `*z`

Pointeurs

```
1 int main(){
2   int z=6;
3   int * p=&z;
4   printf("%d\n",*p);
5   return EXIT_SUCCESS;
6 }
```



Le programme affiche 6

Initialisation d'un pointeur

Initialisation

Comme toute variable, un pointeur doit être initialisé.
Si un pointeur n'est pas initialisé, il peut contenir une adresse quelconque et tenter d'accéder à la variable correspondante peut avoir de grave conséquence.

Ne pointer vers rien

Soit un pointeur `p`. Pour exprimer qu'un pointeur ne pointe vers aucune variable, on écrit :

```
p=NULL;
```

Cette instruction est valable, quelque soit le type du pointeur.

Incrémenter un pointeur

Incrémenter un pointeur

Soit un pointeur `p`. Lorsque l'on écrit `p+1`, cela signifie
"l'adresse contenu dans `p`" + "la taille d'une case du type de `p`"

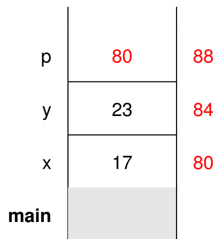
Exemple

Supposons que `p` contient l'adresse 80

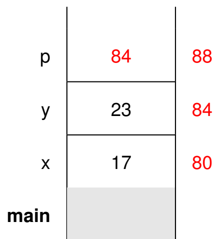
- Si `p` pointe vers un type `int`, alors `p+1`=84
- Si `p` pointe vers un `char`, alors `p+1`=81
- Si `p` pointe vers un `double`, alors `p+1`=88

Incrémenter un pointeur

```
1 int main(){
2   int x=17;
3   int y=23;
4   int *p=&x;
5   printf("%d\n",*p);
6   p=p+1;
7   printf("%d\n",*p);
8   return EXIT_SUCCESS;
9 }
```



Ligne 5 : Affiche 17

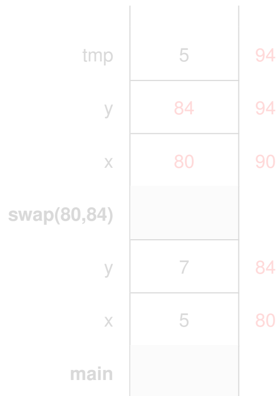


Ligne 7 : Affiche 23

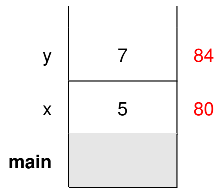
Échange

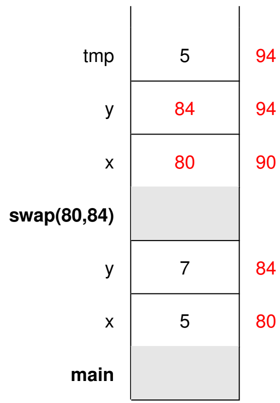
On peut à présent réécrire la fonction `swap`

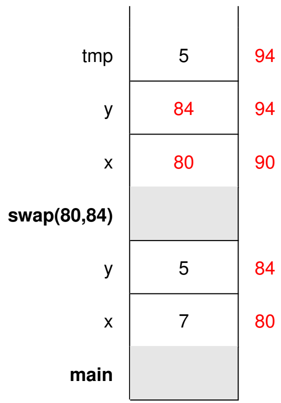
```
1 void swap(int * x,int * y){
2   int tmp=*x;
3   *x=*y;
4   *y=tmp;
5 }
6 int main(){
7   int x=5;
8   int y=7;
9   swap(&x,&y);
0   return EXIT_SUCCESS;
1 }
```

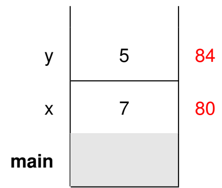


```
1  void swap(int * x,int * y){
2    int tmp=*x;
3    *x=*y;
4    *y=tmp;
5  }
6  int main(){
7    int x=5;
8    int y=7;
9    swap(&x,&y);
0    return EXIT_SUCCESS;
1  }
```









Question

Question

Ecrire la fonction `void ajouter(int * p,int val)` qui ajoute la valeur de `val` à la variable pointée par `p`.

Pointeurs et structure

Pointeurs et structures

Pointeurs et structures

Un pointeur peut contenir l'adresse d'un type scalaire comme d'un type structuré.

Pointeurs sur `point_t`

Déclaration du pointeur : `point_t * p;`

Pointeurs et structures

```
1 /* point.h */
2 struct point_s{
3   int x;
4   int y;
5 };
6 typedef struct point_s point_t;
```

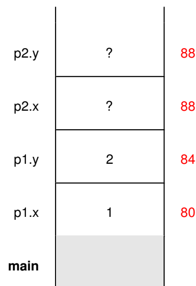
Accès aux champs de la strucure

Soit le pointeur : `point_t * p;`
Accès au champ `x`.

- soit `(*p).x`
- soit `p->x`

Pointeurs et structures

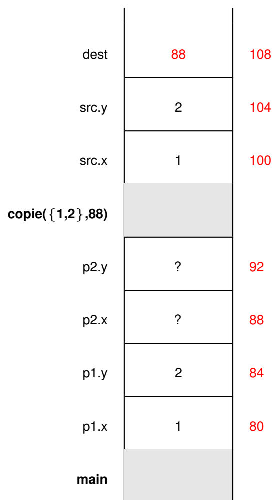
```
1  void copie(point_t src, point_t * dest){
2    dest->x=src.x;
3    dest->y=src.y;
4  }
5
6  int main(){
7    point_t p1={1,2};
8    point_t p2;
9    copie(p1,&p2);
0    return EXIT_SUCCESS;
1  }
```



Initialisation

Pointeurs et structures

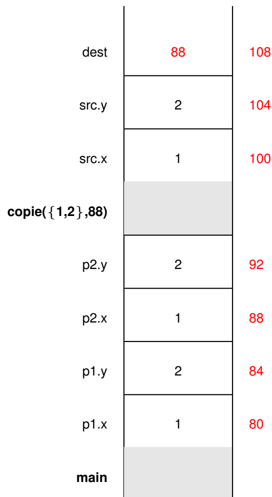
```
1  void copie(point_t src, point_t * dest){
2    dest->x=src.x;
3    dest->y=src.y;
4  }
5
6  int main(){
7    point_t p1={1,2};
8    point_t p2;
9    copie(p1,&p2);
0    return EXIT_SUCCESS;
1  }
```



Appel à copie

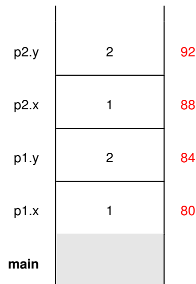
Pointeurs et structures

```
1  void copie(point_t src, point_t * dest){
2    dest->x=src.x;
3    dest->y=src.y;
4  }
5
6  int main(){
7    point_t p1={1,2};
8    point_t p2;
9    copie(p1,&p2);
0    return EXIT_SUCCESS;
1  }
```



Pointeurs et structures

```
1  void copie(point_t src, point_t * dest){
2    dest->x=src.x;
3    dest->y=src.y;
4  }
5
6  int main(){
7    point_t p1={1,2};
8    point_t p2;
9    copie(p1,&p2);
0    return EXIT_SUCCESS;
1  }
```

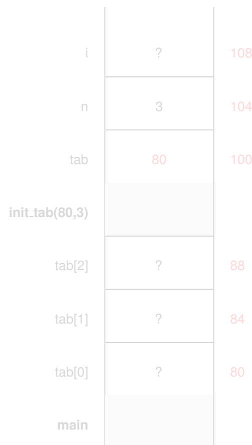


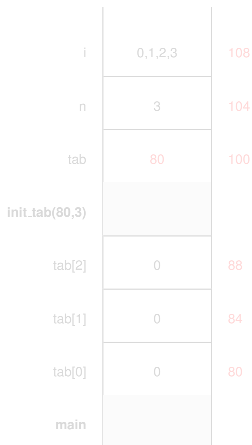
On dépile

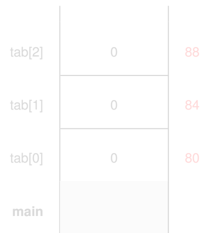
Pointeurs, tableaux et fonctions

Fonctions avec tableaux en arguments

```
1  int init_tab(int tab[], int n){          int main(){
2    int i;                                   int tab[3];
3    for(i=0;i<n;i++)                         init_tab(tab,3);
4      tab[i]=0;                              return EXIT_SUCCESS;
5    }                                      }
```



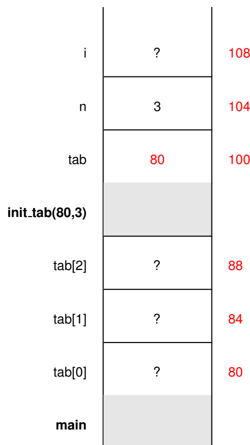


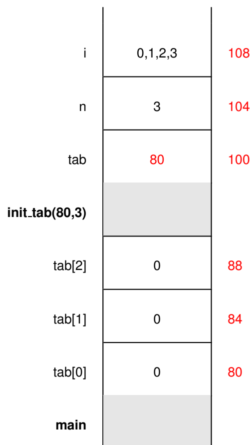


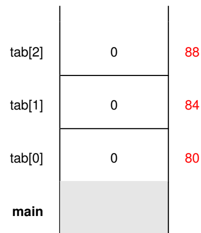
On dépile

Fonctions avec tableaux en arguments

```
1  int init_tab(int tab[], int n){          int main(){
2    int i;                                    int tab[3];
3    for(i=0;i<n;i++)                          init_tab(tab,3);
4      tab[i]=0;                               return EXIT_SUCCESS;
5  }                                         }
```







On dépile

Tableaux et pointeurs

Twist hollywoodien

Un **tableau de** `type` est en réalité un **pointeur vers** `type`.

Déclaration de fonctions

Il est identique d'écrire :

```
void init_tab(int tab[],int n)
```

et

```
void init_tab(int * tab,int n)
```

Les petites différences

- Déclaration d'un tableaux.
- La fonction `sizeof`
- L'opérateur &

Question

Qu'affiche ce programme ?

```
1 int main(){
2   int table[3]={4,2,3};
3   int * p=table;
4   int i;
5   for(i=0;i<3;i++){
6     printf("%d\n", *(p+i));
7   }
8   return EXIT_SUCCESS;
9 }
```

Opérations sur la tableaux

Opération

Lorsque l'on accède à la case *i* d'un tableau `tab`

```
tab[i]
```

On effectue en réalité la commande suivante : `*(tab+i)`

Au prochain TD

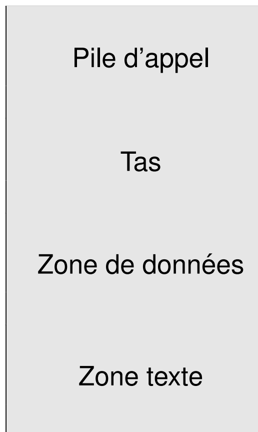
On s'entrainera à

Écrire des fonctions qui manipulent :

- des structures,
- des pointeurs sur des structures,
- des tableaux.

Le tout avec une petite couche de pile d'appel.

Zones de mémoire



À quoi sert cette zone ?