

Langage C

Apprendre à lire...

François Cuvelier

Laboratoire d'Analyse Géométrie et Applications
Institut Galilée
Université Paris XIII.

16 novembre 2012

Plan

- 1 Rappels sur les pointeurs
 - Déclaration et transcription
 - Pointeurs, tableaux et adresses
 - Affectation
 - Exemple
 - Exemple avec représentation mémoire
- 2 Exercice 1

Listing 1 – Exemple de déclarations

```
1 TRUC a; /* ESSAI */  
2 TRUC *pT;  
3 TRUC t [3];
```

Listing 1 – Exemple de déclarations

```
1   TRUC a; /* ESSAI */  
2   TRUC *pT;  
3   TRUC t [3];
```

- `a` est un `TRUC`,

Listing 1 – Exemple de déclarations

```
1   TRUC a; /* ESSAI */  
2   TRUC *pT;  
3   TRUC t [3];
```

- **a** est un **TRUC**,
- **pT** est un pointeur de **TRUC**.

Listing 1 – Exemple de déclarations

```
1   TRUC a; /* ESSAI */  
2   TRUC *pT;  
3   TRUC t [3];
```

- **a** est un **TRUC**,
- **pT** est un pointeur de **TRUC**.
- **t** est un tableau de 3 **TRUC**.

Lorsque le nom du tableau `t` est utilisé dans une expression il est **converti** en un pointeur valant l'adresse du premier élément du tableau (i.e. `&t[0]`). On note

$$t \rightsquigarrow \&t[0]$$

cette conversion.

Attention, la conversion n'a pas lieu dans les deux cas suivants :

- 1 utilisation de la fonction `sizeof`. `sizeof(t)` correspond à la taille globale du tableau.
- 2 utilisation de l'opérateur `&`. `&t[i]` est l'adresse du *i*-ème élément du tableau `t`.

Pour accéder aux éléments d'un tableau, on peut utiliser l'opérateur d'indilage []. En fait, `t[i]` est remplacé par le compilateur en `*(t+i)`. on note

$$t[i] \leftrightarrow *(t+i)$$

cette opération.

Pour effectuer la somme d'une adresse de `TRUC` avec un entier, le compilateur utilise la relation suivante :

$$t+i \rightsquigarrow \&t[0]+i*\text{sizeof}(TRUC)$$

et donc

$$*(t+i) \rightsquigarrow *(\&t[0]+i*\text{sizeof}(TRUC))$$

Pour affecter une valeur correcte à `pT`, il faut lui donner un pointeur (i.e. une adresse) de `TRUC` existant. Par exemple :

Listing 2 – Exemples simples d'affectation

```
1  pT=&a ;  
2  pT=&t [ 0 ] ;  
3  pT=t ;
```

Examinons les instructions suivantes

Listing 3 – Exemples complexes d'affectation

```
1  pT=t +1 ;  
2  *pT=a ;  
3  *++pT=a ;
```

Examinons les instructions suivantes

Listing 3 – Exemples complexes d'affectation

```
1  pT=t +1 ;  
2  *pT=a ;  
3  *++pT=a ;
```

- On a $t+1 \rightsquigarrow \&t[0]+1*\text{sizeof}(\text{TRUC})$. On affecte donc à pT la valeur de $\&t[1]$.

Examinons les instructions suivantes

Listing 3 – Exemples complexes d'affectation

```
1  pT=t +1 ;
2  *pT=a ;
3  *++pT=a ;
```

- On a $t+1 \rightsquigarrow \&t[0]+1*\text{sizeof}(\text{TRUC})$. On affecte donc à pT la valeur de $\&t[1]$.
- Pour la seconde instruction, $*pT$ correspond alors à $\&t[1]$ c'est à dire à $t[1]$. En d'autres termes, $*pT$ est l'objet pointé par pT . Donc, on affecte à $t[1]$ la valeur de a .

Examinons les instructions suivantes

Listing 3 – Exemples complexes d'affectation

```
1  pT=t +1;
2  *pT=a ;
3  *++pT=a ;
```

- On a $t+1 \rightsquigarrow \&t[0]+1*\text{sizeof}(\text{TRUC})$. On affecte donc à pT la valeur de $\&t[1]$.
- Pour la seconde instruction, $*pT$ correspond alors à $\&t[1]$ c'est à dire à $t[1]$. En d'autres termes, $*pT$ est l'objet pointé par pT . Donc, on affecte à $t[1]$ la valeur de a .
- La dernière instruction est équivalente à $++pT; *pT=a;$ ou encore $pT=pT+1; *pT=a;$, c'est à dire pT prend la valeur $\&t[2]$ et, ensuite, $t[2]$ la valeur de a .

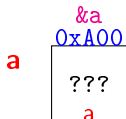
Listing 4 – Exemple.c

```
1 #include <stdio .h>
2
3 int main(){
4     int a;
5     int *pT;
6     int t[3]={1 ,2 ,3};
7
8     a=5;
9     pT=t +2;
10    *pT=a;
11    *--pT=a -1;
12    return 1;
13 }
```

```
int a;
```

Listing 4 – Exemple.c

```
1 #include <stdio .h>
2
3 int main(){
4     int a;
5     int *pT;
6     int t[3]={1,2,3};
7
8     a=5;
9     pT=t+2;
10    *pT=a;
11    *--pT=a-1;
12    return 1;
13 }
```

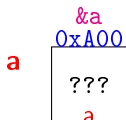


```
int a;
```

a est de type **int** (non initialisé)

Listing 4 – Exemple.c

```
1 #include <stdio .h>
2
3 int main(){
4     int a ;
5     ▶ int *pT ;
6     int t [3] = {1 ,2 ,3} ;
7
8     a =5 ;
9     pT =t +2 ;
10    *pT =a ;
11    *--pT =a -1 ;
12    return 1 ;
13 }
```



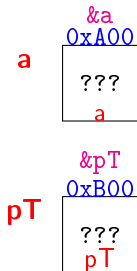
```
int *pT;
```

Listing 4 – Exemple.c

```

1 #include <stdio .h>
2
3 int main(){
4     int a;
5     int *pT;
6     int t[3]={1,2,3};
7
8     a=5;
9     pT=t+2;
10    *pT=a;
11    *--pT=a-1;
12    return 1;
13 }

```



```
int *pT;
```

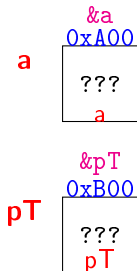
pT est un pointeur de **int** (non initialisé)

Listing 4 – Exemple.c

```

1 #include <stdio .h>
2
3 int main(){
4     int a;
5     int *pT;
6     int t[3]={1,2,3};
7
8     a=5;
9     pT=t+2;
10    *pT=a;
11    *--pT=a-1;
12    return 1;
13 }

```



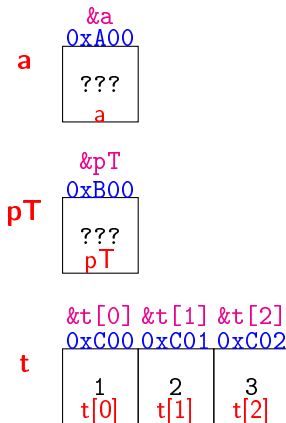
```
int t[3]={1,2,3};
```

Listing 4 – Exemple.c

```

1 #include <stdio .h>
2
3 int main(){
4     int a;
5     int *pT;
6     int t[3]={1,2,3};
7
8     a=5;
9     pT=t +2;
10    *pT=a;
11    *--pT=a-1;
12    return 1;
13 }

```



```
int t[3]={1, 2, 3};
```

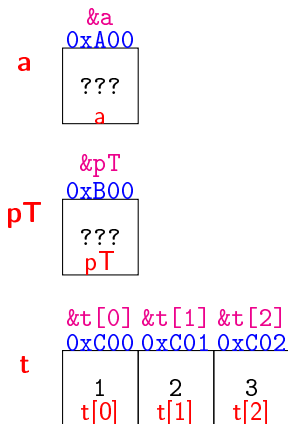
t est un tableau de 3 **int** initialisé par {1,2,3}

Listing 4 – Exemple.c

```

1 #include <stdio .h>
2
3 int main(){
4     int a;
5     int *pT;
6     int t[3]={1 ,2 ,3};
7
8     ▶ a=5;
9     pT=t +2;
10    *pT=a;
11    *--pT=a-1;
12    return 1;
13 }

```



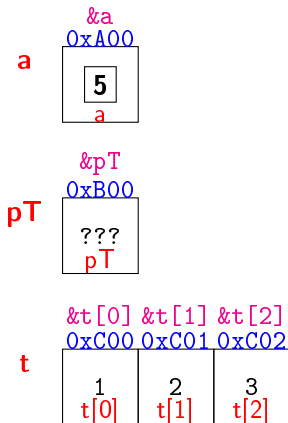
a=5;

Listing 4 – Exemple.c

```

1 #include <stdio .h>
2
3 int main(){
4     int a;
5     int *pT;
6     int t[3]={1,2,3};
7
8     ▶ a=5;
9     pT=t+2;
10    *pT=a;
11    *--pT=a-1;
12    return 1;
13 }

```



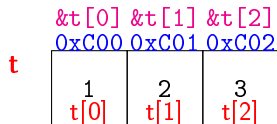
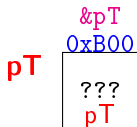
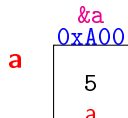
a=5;

On affecte 5 à a

Listing 4 – Exemple.c

```

1 #include <stdio .h>
2
3 int main(){
4     int a;
5     int *pT;
6     int t[3]={1 ,2 ,3};
7
8     a=5;
9     ► pT=t +2;
10    *pT=a;
11    *--pT=a -1;
12    return 1;
13 }
```



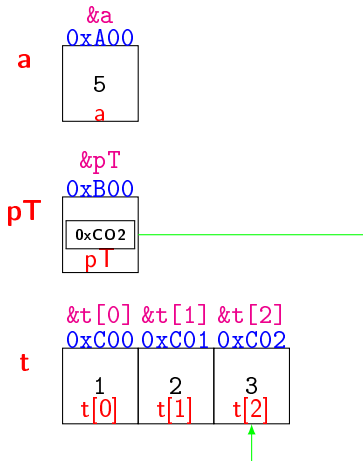
pT=t+2;

Listing 4 – Exemple.c

```

1 #include <stdio .h>
2
3 int main(){
4     int a;
5     int *pT;
6     int t[3]={1,2,3};
7
8     a=5;
9     ▶ pT=t+2;
10    *pT=a;
11    *--pT=a-1;
12    return 1;
13 }

```



`pT=t+2;`

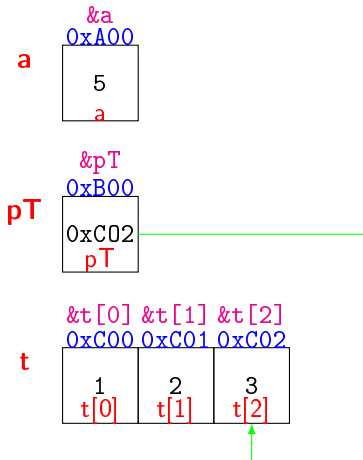
`t+2` \rightsquigarrow `&t[0]+2*sizeof(TRUC)` \Rightarrow `pT` \leftarrow `&t[2]`

Listing 4 – Exemple.c

```

1 #include <stdio .h>
2
3 int main(){
4     int a;
5     int *pT;
6     int t[3]={1 ,2 ,3};
7
8     a=5;
9     pT=t +2;
10    ► *pT=a;
11    *--pT=a -1;
12    return 1;
13 }

```

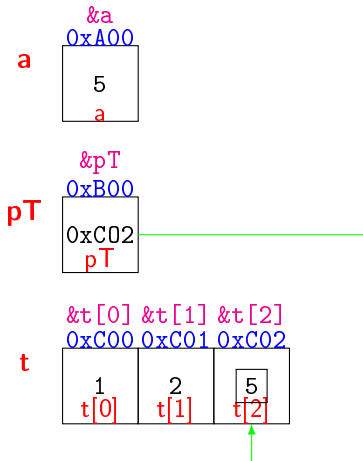


`*pT=a;`

Listing 4 – Exemple.c

```

1  #include <stdio .h>
2
3  int main(){
4      int a;
5      int *pT;
6      int t[3]={1,2,3};
7
8      a=5;
9      pT=t+2;
10     ▶ *pT=a;
11     *--pT=a-1;
12     return 1;
13 }
```



```
*pT=a;
```

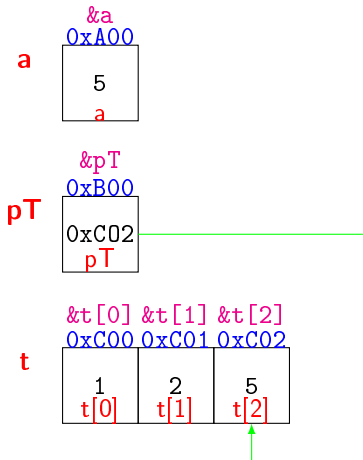
```
*pT ⇔ t[2] ⇒ t[2] ← a
```

Listing 4 – Exemple.c

```

1 #include <stdio .h>
2
3 int main(){
4     int a;
5     int *pT;
6     int t[3]={1 ,2 ,3};
7
8     a=5;
9     pT=t +2;
10    *pT=a;
11    ► *--pT=a-1;
12    return 1;
13 }

```



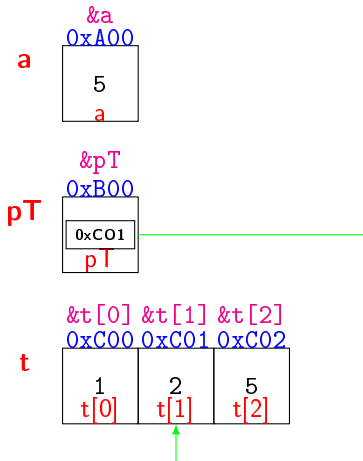
`*--pT=a-1;`

Listing 4 – Exemple.c

```

1 #include <stdio .h>
2
3 int main(){
4     int a;
5     int *pT;
6     int t[3]={1,2,3};
7
8     a=5;
9     pT=t+2;
10    *pT=a;
11    *--pT=a-1;
12    return 1;
13 }

```



```
*--pT=a-1;
```

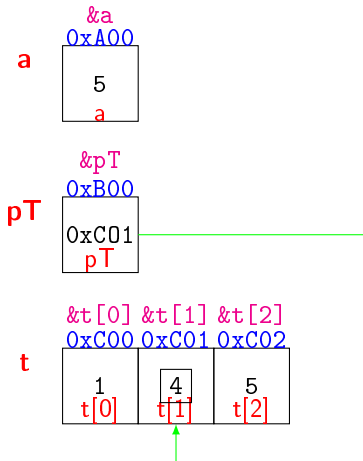
```
*--pT=a-1;  $\iff$  --pT; *pt=a-1;  $\implies$  pT  $\leftarrow$  &t[1]
```

Listing 4 – Exemple.c

```

1 #include <stdio .h>
2
3 int main(){
4     int a ;
5     int *pT ;
6     int t[3]={1 ,2 ,3};
7
8     a=5;
9     pT=t +2;
10    *pT=a ;
11    ► *--pT=a -1;
12    return 1;
13 }

```

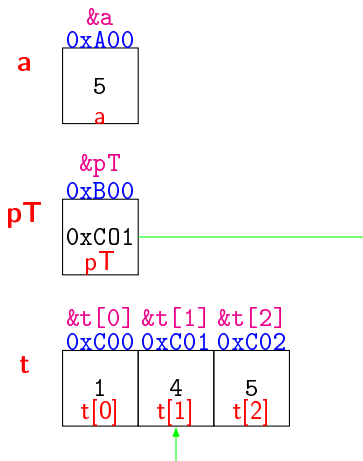


```
*--pT=a-1;
```

```
*--pT=a-1; ⇔ --pT; *pt=a-1; ⇒ pT ← &t[1] puis t[1] ← a-1
```

Listing 4 – Exemple.c

```
1 #include <stdio .h>
2
3 int main(){
4     int a;
5     int *pT;
6     int t[3]={1 ,2 ,3};
7
8     a=5;
9     pT=t +2;
10    *pT=a;
11    *--pT=a-1;
12    return 1;
13 }
```



Plan

- 1 Rappels sur les pointeurs
 - Déclaration et transcription
 - Pointeurs, tableaux et adresses
 - Affectation
 - Exemple
 - Exemple avec représentation mémoire
- 2 Exercice 1

Question

Expliquer (à l'aide de schémas de représentation mémoire,...) le déroulement du programme suivant. Qu'affiche-t-il ? Décrire le contenu des variables `c`, `cp` et `cpp` en fin d'exécution.

Listing 5 – Exo1.c

```
1  #include <stdio .h>
2
3  int main(){
4      char *c [] = { "ENTER" , "NEW" , "POINT" , "FIRST" };
5      char **cp [] = { c+3, c+2, c+1, c };
6      char ***cpp = cp;
7
8      printf ("%s", **++cpp);
9      printf ("%s", *--*++cpp+3);
10     printf ("%s", *cpp[-2]+3);
11     printf ("%s\n", cpp[-1][-1]+1);
12 }
```

0xA00 0xA01 0xA02 0xA03 0xA04 0xA05

'E'	'N'	'T'	'E'	'R'	'\0'
-----	-----	-----	-----	-----	------

0xA10 0xA11 0xA12 0xA13

'N'	'E'	'W'	'\0'
-----	-----	-----	------

0xA20 0xA21 0xA22 0xA23 0xA24 0xA25

'P'	'O'	'I'	'N'	'T'	'\0'
-----	-----	-----	-----	-----	------

0xA30 0xA31 0xA32 0xA33 0xA34 0xA35

'F'	'I'	'R'	'S'	'T'	'\0'
-----	-----	-----	-----	-----	------

```
char *c [] = {"ENTER", "NEW", "POINT", "FIRST"};
```

```
{"ENTER", "NEW", "POINT", "FIRST"} ?
```

0xA00 0xA01 0xA02 0xA03 0xA04 0xA05

'E'	'N'	'T'	'E'	'R'	'\0'
-----	-----	-----	-----	-----	------

0xA10 0xA11 0xA12 0xA13

'N'	'E'	'W'	'\0'
-----	-----	-----	------

0xA20 0xA21 0xA22 0xA23 0xA24 0xA25

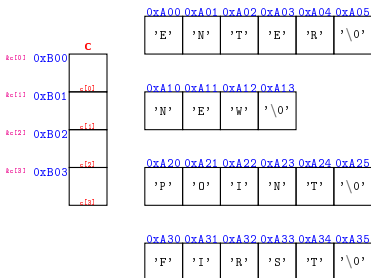
'P'	'O'	'I'	'N'	'T'	'\0'
-----	-----	-----	-----	-----	------

0xA30 0xA31 0xA32 0xA33 0xA34 0xA35

'F'	'I'	'R'	'S'	'T'	'\0'
-----	-----	-----	-----	-----	------

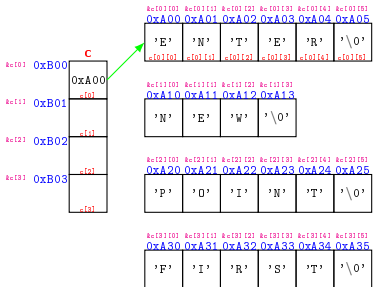
```
char *c[] = {"ENTER", "NEW", "POINT", "FIRST"};
```

- c est un ?



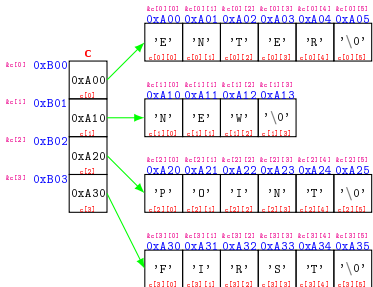
```
char *c [] = {"ENTER", "NEW", "POINT", "FIRST"};
```

- `c` est un tableau de 4 (nombre d'éléments de la liste) pointeurs de `char`.
- `c[0]` contient ?



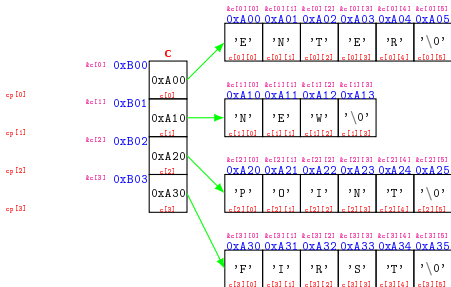
```
char *c[] = {"ENTER", "NEW", "POINT", "FIRST"};
```

- c est un tableau de 4 (nombre d'éléments de la liste) pointeurs de char.
- c[0] contient l'adresse du premier élément de la chaîne "ENTER",
- c[1] contient ?



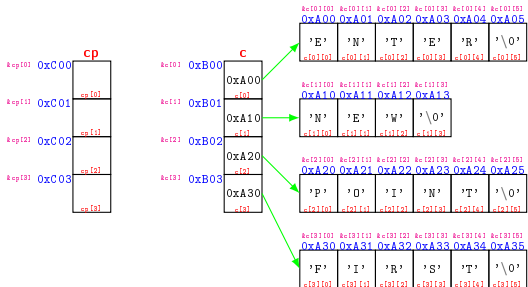
```
char *c[] = {"ENTER", "NEW", "POINT", "FIRST"};
```

- `c` est un tableau de 4 (nombre d'éléments de la liste) pointeurs de `char`.
- `c[0]` contient l'adresse du premier élément de la chaîne "ENTER",
- `c[1]` contient l'adresse du premier élément de la chaîne "NEW",
- `c[2]` contient l'adresse du premier élément de la chaîne "POINT",
- `c[3]` contient l'adresse du premier élément de la chaîne "FIRST",



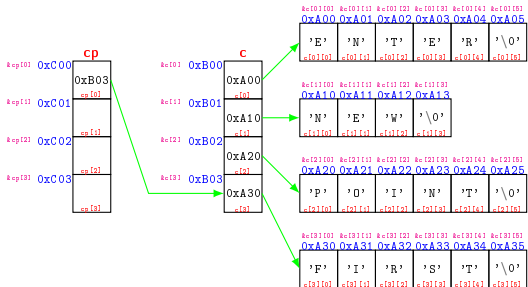
```
char **cp[] = {c+3,c+2,c+1,c};
```

- cp est un ?



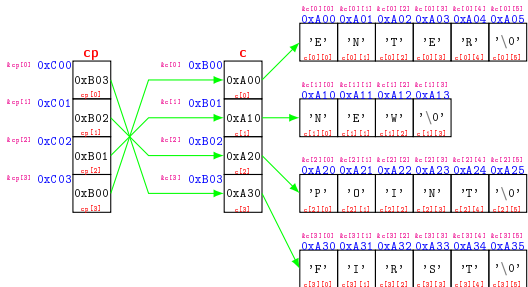
```
char **cp[] = {c+3,c+2,c+1,c};
```

- cp est un tableau de 4 pointeurs de pointeurs de char.
- cp[0] contient ?



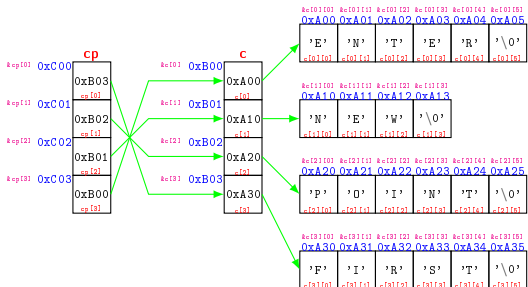
```
char **cp[] = {c+3,c+2,c+1,c};
```

- `cp` est un tableau de 4 pointeurs de pointeurs de `char`.
- `cp[0]` contient `c+3` \iff `&c[0]+3*sizeof(char *)` \iff `&c[3]`.
- `cp[1]` contient ?



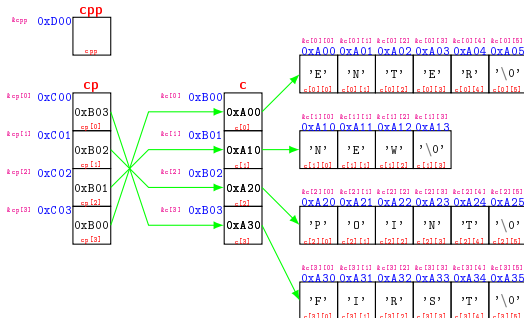
```
char **cp[] = {c+3,c+2,c+1,c};
```

- `cp` est un tableau de 4 pointeurs de pointeurs de `char`.
- `cp[0]` contient `c+3` \iff `&c[0]+3*sizeof(char *)` \iff `&c[3]`.
- `cp[1]` contient `c+2` \iff `&c[0]+2*sizeof(char *)` \iff `&c[2]`.
- `cp[2]` contient `c+1` \iff `&c[0]+sizeof(char *)` \iff `&c[1]`.
- `cp[3]` contient `c` \iff `&c[0]`.



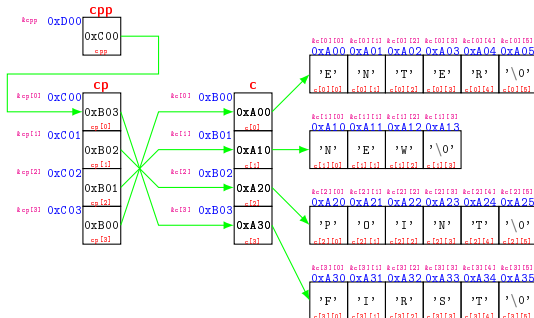
```
char ***cpp = cp;
```

- `cpp` est un ?



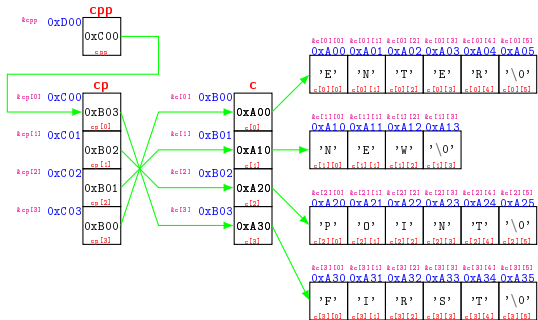
```
char ***cpp = cp;
```

- `cp` est un pointeur de pointeur de pointeur de `char`.
- `cpp` contient ?



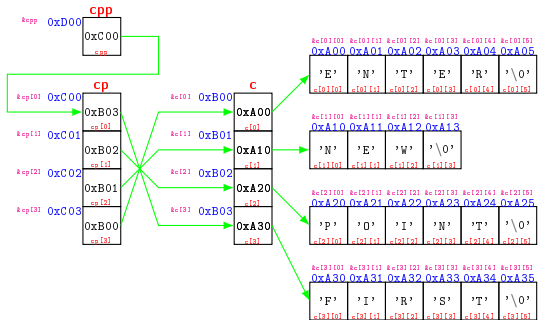
```
char ***cpp = cp;
```

- `cpp` est un pointeur de pointeur de pointeur de `char`.
- `cpp` contient `cp` \iff `&cp[0]`.



```
printf ("%s",**++cpp);
```

↑ *mémoire avant instruction* ↑

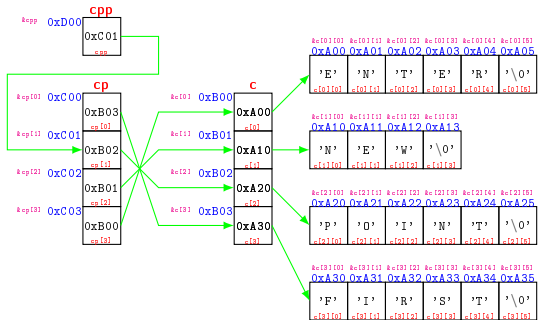


`printf ("%s",**++cpp);` ↑ *mémoire avant instruction* ↑

On évalue tout d'abord l'expression `**++cpp` qui est équivalente à `*(++cpp)`.
`++cpp` est évalué avant le calcul de l'expression totale et correspond à `cpp=cpp+1`, c'est à dire `cpp=&cp[0]+1=&cp[1]`.

Ensuite, `*++cpp` correspond à `*(&cp[1])=cp[1]=&c[2]`. Puis `**++cpp` correspond à `*(&c[2])=c[2]` qui est l'adresse de la chaîne pointée par `c[2]` c'est à dire "POINT".

En résumé, `cpp` prend la valeur `&cp[1]` et on affiche la chaîne "POINT".



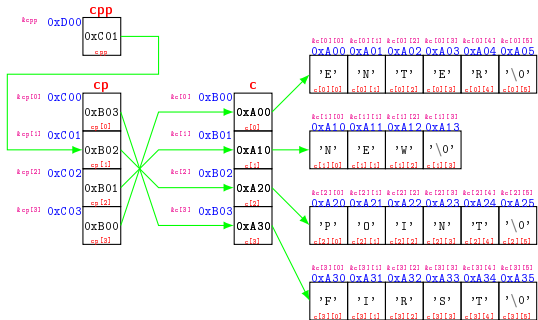
```
printf ("%s",**++cpp);
```

↑ *mémoire après instruction* ↑

On évalue tout d'abord l'expression `**++cpp` qui est équivalente à `*(++cpp)`.
`++cpp` est évalué avant le calcul de l'expression totale et correspond à `cpp=cpp+1`, c'est à dire `cpp=&cp[0]+1=&cp[1]`.

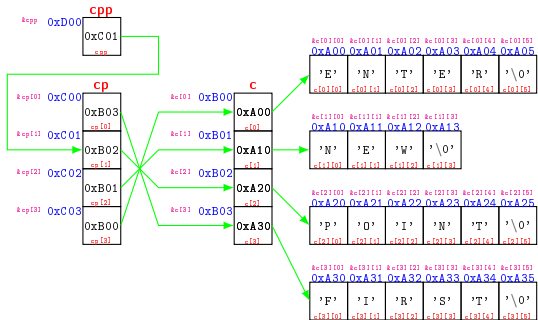
Ensuite, `*++cpp` correspond à `*(&cp[1])=cp[1]=&c[2]`. Puis `**++cpp` correspond à `*(&c[2])=c[2]` qui est l'adresse de la chaîne pointée par `c[2]` c'est à dire "POINT".

En résumé, `cpp` prend la valeur `&cp[1]` et on affiche la chaîne "POINT".



```
printf ("%s",*--*++cpp+3);
```

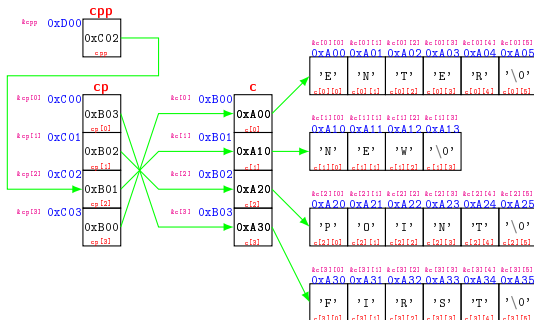
↑↑ *mémoire avant instruction* ↑↑



`printf ("%s",*--*++cpp+3);` ↑↑ *mémoire avant instruction* ↑↑

$$*--*++cpp+3 \iff (*(--(*(++cpp))))+3$$

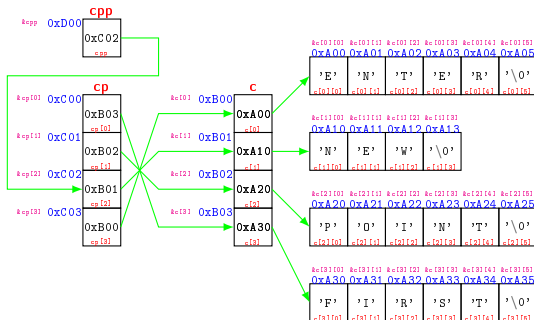
`++cpp` : on incrémente `cpp`, puis on l'évalue. `cpp=&cp[1]+1=&cp[2]`.



`printf ("%s",*--*++cpp+3);` ↑↑ *Après évaluation de* `++cpp` ↑↑

$$*--*++cpp+3 \iff (*(--(*(++cpp))))+3$$

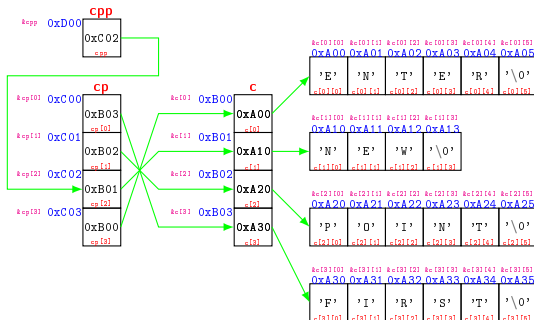
`++cpp` : on incrémente `cpp`, puis on l'évalue. `cpp=&cp[1]+1=&cp[2]`.



`printf ("%s",*--*++cpp+3);` ↑ Après évaluation de `++cpp` ↑

$$*--*++cpp+3 \iff (*(--(*(++cpp))))+3$$

`*(++cpp)` correspond à la valeur pointée par `++cpp`. $*(++cpp) \iff cp[2]$.



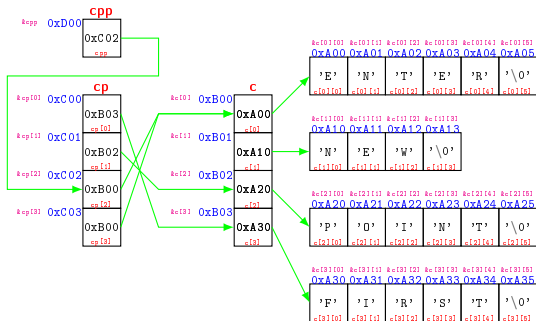
```
printf ("%s",*--*++cpp+3);
```

↑↑ *Après évaluation de* `++cpp` ↑↑

$$*--*++cpp+3 \iff (*(--(*(++cpp))))+3$$

`--(*(++cpp))` \iff `--(cp[2])` : on décrémente `cp[2]`, puis on l'évalue.

`--(cp[2])` = `&c[1]-1` = `&c[0]`.

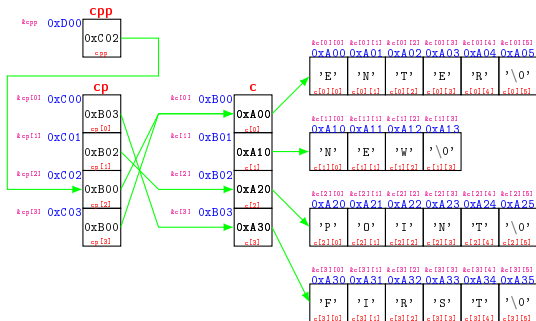


`printf ("%s",*--*++cpp+3);` \uparrow *Après évaluation de* `--*++cpp` \uparrow

$$*--*++cpp+3 \iff (*(--(*(++cpp))))+3$$

`--(*(++cpp))` \iff `--(cp[2])` : on décrémente `cp[2]`, puis on l'évalue.

`--(cp[2]) = &c[1] - 1 = &c[0]`.



```
printf ("%s",*--*++cpp+3);
```

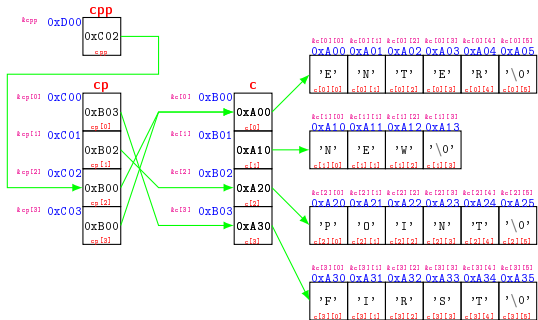
↑ *Après évaluation de* `--*++cpp` ↑

`*--*++cpp+3` \iff `(*(--(*(++cpp))))+3`

`(*(--(*(++cpp))))` \iff `(**cpp)+3` \iff `c[0]+3` \iff `&c[0][0]+3`

\iff `&c[0][3]`.

on affiche donc la chaîne "ER".



printf ("%s",*cpp[-2]+3); ↑ *mémoire avant instruction* ↑

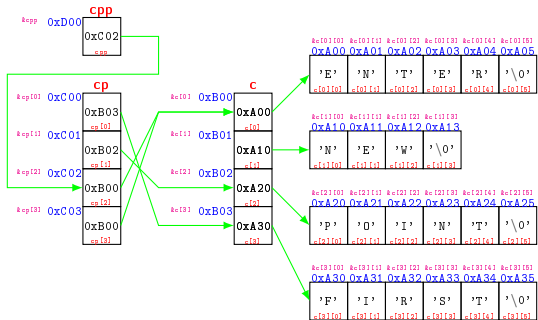
$$*cpp[-2]+3 \iff (*(cpp[-2]))+3$$

$$cpp[-2] \iff *(cpp-2) \iff *(&cp[2]-2) \iff *(&cp[0]) \iff cp[0] \iff &c[3]$$

$$*(cpp[-2]) \iff *(&c[3]) \iff c[3]$$

$$*(cpp[-2])+3 \iff c[3]+3 \iff &c[3][0]+3 \iff &c[3][3]$$

on affiche donc la chaine "ST".



printf ("%s",*cpp[-2]+3); ↑ *mémoire après instruction* ↑

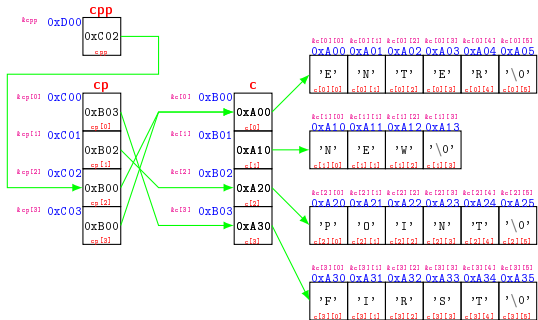
$$*cpp[-2]+3 \iff (*(cpp[-2]))+3$$

$$cpp[-2] \iff *(cpp-2) \iff *(&cp[2]-2) \iff *(&cp[0]) \iff cp[0] \iff &c[3]$$

$$*(cpp[-2]) \iff *(&c[3]) \iff c[3]$$

$$*(cpp[-2])+3 \iff c[3]+3 \iff &c[3][0]+3 \iff &c[3][3]$$

on affiche donc la chaine "ST".



`printf ("%s\n",cpp[-1][-1]+1);` ↑ *mémoire avant instruction* ↑

$$\text{cpp}[-1][-1]+1 \iff ((\text{cpp}[-1])[-1])+1$$

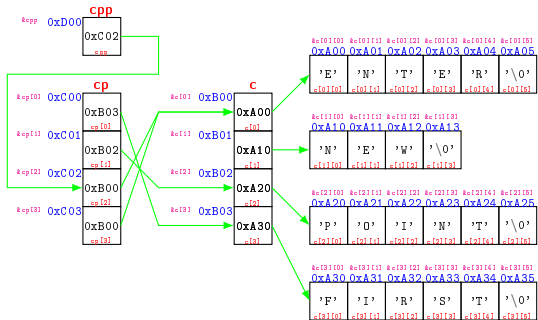
$$\text{cpp}[-1] \iff *(\text{cpp}-1) \iff *(&\text{cp}[2]-1) \iff *(&\text{cp}[1]) \iff \text{cp}[1] \iff &\text{c}[2]$$

$$(\text{cpp}[-1])[-1] \iff (\text{cp}[1])[-1] \iff *(\text{cp}[1]-1) \iff *(&\text{c}[2]-1)$$

$$\iff *(&\text{c}[1]) \iff \text{c}[1]$$

$$((\text{cpp}[-1])[-1])+1 \iff \text{c}[1]+1 \iff &\text{c}[1][0]+1 \iff &\text{c}[1][1]$$

on affiche donc la chaine "EW" et on passe à la ligne.



```
printf ("%s\n",cpp[-1][-1]+1);
```

↑ *mémoire après instruction* ↑

$$\text{cpp}[-1][-1]+1 \iff ((\text{cpp}[-1])[-1])+1$$

$\text{cpp}[-1] \iff *(\text{cpp}-1) \iff *(&\text{cp}[2]-1) \iff *(&\text{cp}[1]) \iff \text{cp}[1] \iff \&\text{c}[2]$
 $(\text{cpp}[-1])[-1] \iff (\text{cp}[1])[-1] \iff *(\text{cp}[1]-1) \iff *(&\text{c}[2]-1)$
 $\iff *(&\text{c}[1]) \iff \text{c}[1]$
 $((\text{cpp}[-1])[-1])+1 \iff \text{c}[1]+1 \iff \&\text{c}[1][0]+1 \iff \&\text{c}[1][1]$

on affiche donc la chaine "EW" et on passe à la ligne.