# Les pointeurs : notions de base [1]

Salem BENFERHAT

Centre de Recherche en Informatique de Lens (CRIL-CNRS)
email : benferhat@cril.fr

[1] Version préliminaire du cours. Tout retour sur la forme comme sur le fond est le bienvenu.

# ANNONCE POUR 1 POSTE D'ETUDIANT ASSISTANT REMUNERE

- Nous avons besoin de volontaires pour un job étudiant afin d'aider un étudiant en situation de handicap et lui permettre de poursuivre ses études dans les meilleures conditions.

1 poste est à pourvoir dans votre groupe :

- Preneur de notes : il s'agit de prendre des notes de cours (CM, TD) les plus complètes et les plus rigoureuses possible et les déposer, en format Word, à chaque fin de journée de cours sur un espace partagé.
- Le poste est rémunéré au SMIC horaire pour trois heures de prise de notes.
- Si vous êtes intéressé(e), merci de postuler directement via OSE Étudiants ou de contacter le service handicap à celine..decodts@univ-artois.fr / 03 21 60 37 34 le plus rapidement possible.

# Les pointeurs

# Pointeurs

Indispensable pour comprendre :

- les fonctions,
- les tableaux, et surtout
- la gestion dynamique de la mémoire.

Cours d'aujourd'hui : introduire les notions de base des pointeurs.

# Informellement : accès direct et indirect

Supposons que vous avez effectué sur internet un achat, par exemple ”location d’une voiture”.

## Première réponse : accès direct

A la suite de votre achat, vous avez toutes les informations nécessaires :

- Numéro de place du parking,
- type de voiture,
- etc.

On parle alors d’un accès **direct** à l’information.

# Informellement : accès direct et indirect

## Deuxième réponse : accès indirect

Supposons maintenant qu'à la suite de votre de votre achat,

- vous avez juste un numéro d'un casier (au niveau de l'agence de location de voitures),
- et qu'à l'intérieur de ce casier vous trouverez les informations sur votre voiture (numéro de place du parking, type de voiture etc)

On parle alors d'un accès **indirect** à l'information.

# Informellement : accès direct et indirect

## Troisième réponse : plusieurs indirections

Supposons maintenant qu'à la suite de votre achat,

- vous avez juste un numéro d'un casier (au niveau de l'agence de location de voitures),
- à l'intérieur de ce casier, il y a encore un autre numéro de casier,
- dans ce dernier casier, vous trouverez enfin les informations sur votre voiture (numéro de place du parking, type de voiture etc)

# Informellement : accès direct et indirect

Ce qui rend la tâche un peu difficile est qu'en présence d'une information (un nombre par exemple), il faut toujours se poser la question :

- s'agit-il de l'information que l'on cherche (donc accès direct) ?

ou bien

- s'agit-il d'un numéro d'emplacement où on peut trouver l'information (donc accès indirect) ?

# Tentative d'accès à un casier qui n'est pas le votre

- Les casiers sont partagés par plusieurs utilisateurs
- Vous ne pouvez accéder qu'à votre casier.
- Sinon une alarme sera déclenchée!

# Par analogie avec l'accès à la mémoire

- La mémoire est généralement partagée par différents programmes.
- A chaque programme, un espace mémoire (pas nécessairement contigu) lui est réservé.

# Par analogie avec l'accès à la mémoire

- Quand vous déclarez une variable, par exemple "char c",
  - un espace mémoire (ici d'un octet) est réservé à votre programme.
  - Comme cet espace vous appartient, vous pouvez le modifiez, par exemple avec l'instruction c='A'.
  - Cependant, un autre programme ne peut pas modifier cet espace mémoire.
  - De manière réciproque, vous ne pouvez pas modifier le contenu d'un espace mémoire qui appartient à un autre programme.

# Erreur d'exécution

Attention : tenter d'accéder ou de modifier le contenu d'un espace mémoire qui n'appartient pas à votre programme peut générer une erreur d'exécution :

**Segmentation fault**

# Variables, types
# espaces mémoire et adresses

# Les variables ...

- Les langages de programmation permettent de décrire des données de natures et complexités différentes en utilisant la notion de *type*
- Nous avons déjà vu les types de base suivants :
  - Char
  - Short int
  - Int
  - Long int
  - version non-sgnée des entiers (unsigned)
  - float
  - double
- Le but de ce cours est de vous parler d'un nouveau type : les pointeurs.

# Les variables : rappels

- A chaque fois qu'une variable de type T (de base ou complexe) est déclarée un espace mémoire lui est réservé.

- En langage C, la commande **sizeof** permet de connaître l'espace mémoire associé à chaque type.

# Les variables : rappels

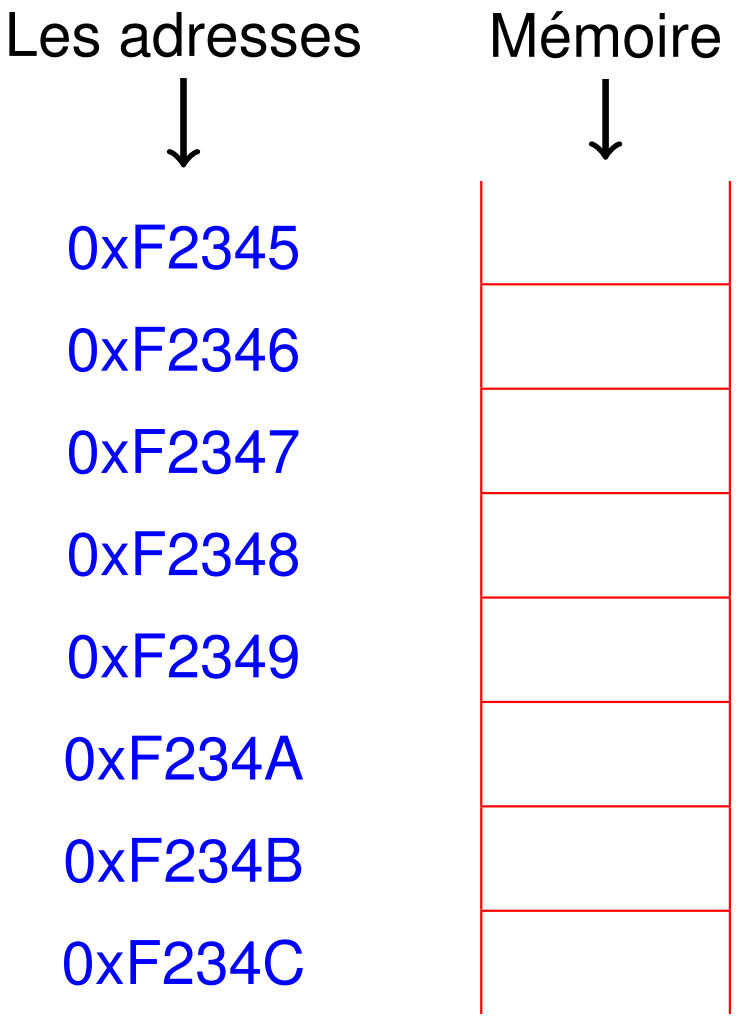
- A chaque fois qu'une variable de type T (de base ou complexe) est déclarée un espace mémoire lui est réservé.
- En langage C, la commande sizeof permet de connaître l'espace mémoire associé à chaque type.
- Par exemple, l'instruction :

```
1 printf ("char = %lu byte\n", sizeof(char));
```

permet d'afficher le nombre de mots (ou d'octets) associés à une variable de type char (ici 1 seul octet).

# Mémoire et adresses : emplacement des données

- Les programmes, les fonctions, les variables et les données sont stockés en mémoires.
- La mémoire peut-être vue comme un grand tableau où chaque case contient généralement un mot mémoire.
- Les indices de ce tableau sont appelés des adresses (en hexadécimal).
- Les indices du tableau sont des adresses mémoires.
- On accède aux informations grâce à ces adresses.



# Adresses mémoire et pointeurs

## Pointeurs

- Pour manipuler les adresses mémoires nous avons besoin de variables particulières appelées *pointeurs*.
- Un pointeur est donc une variable qui contient l'adresse d'une donnée contenue en mémoire.
- Un pointeur aura une valeur entière (qui représente une adresse mémoire).
- Grâce aux pointeurs, on sépare le contenu (une valeur) du contenant (une adresse).

# Déclaration de pointeurs

## Déclaration avec (*) en préfixe

Type *Nom_de_la_variable;

- Le type peut-être de type simple (entier, réel, ...) ou complexe.
- La présence de * indique que l'on ne travaille plus sur une variable normale, mais sur un pointeur.

# Déclaration de pointeurs

## Example

- La déclaration

$$int * p;$$

  - indique que p est une variable de type pointeur.
  - La donnée qui se trouve à l'adresse de p est de type entier.

- De même, la déclaration

$$float * p1;$$

  - indique que p1 est une variable de type pointeur.
  - La donnée qui se trouve à l'adresse de p1 est de type flottant.

# Taille d'une variable de type pointeur

- Au niveau de la compilation (allocation statique) :
  - seul l'espace dédié aux pointeurs est réservé.
  - La taille de cet espace est fixe : c'est la taille nécessaire pour stocker des adresses.
  - Cette taille est complètement indépendante du type de données "pointé" par la variable pointeur.

# Taille d'une variable de type pointeur : exemple

```
1  int main(void)
2  {
3   int *pointeur1;
4   double *pointeur2;
5   printf ("la taille d'un pointeur sur un type :\n");
6   printf ("\t entier est de : %lu bytes \n", sizeof(pointeur1));
7   printf ("\t double est de : %lu bytes \n", sizeof(pointeur2));
8   return (0);
9  }
```

# Taille d'une variable de type pointeur

## Exemple

- L'exécution du programme donne :
- la taille d'un pointeur :
  sur un type entier est de : 8 bytes
  sur un type double est de : 8 bytes
- La taille réservée à une variable de type pointeur est la même.

# Données et adresses

# L'opérateur &

- Pour rappels, les variables sont stockées en mémoire.
- L'opérateur &, appliqué à une variable, permet de récupérer l'adresse de la variable.
  - Par exemple, supposons que nous avons la déclaration suivante :

$$int \quad i;$$

  - alors l'opération :

$$\&i$$

    donne l'adresse mémoire (en hexadécimal) où est stockée la variable *i*.

# Le format %p dans printf

- Il est possible d'afficher le contenu d'une variable de type pointeur.
- Il faut pour cela utiliser le format d'affichage "%p" dans la fonction printf.

**Exemple**

Ecrire un programme C qui déclare une variable *i* de type entier et affiche son adresse.

# Le format %p dans printf : un exemple

```
#include <stdio.h>
int main (void)
{
    int i;
    printf ("L'addresse de la variable i est : %p \n", &i);
    return(0);
}
```

# Le format %p dans printf : un exemple

```
#include <stdio.h>
int main (void)
{
    int i;
    printf ("L'addresse de la variable i est : %p \n", &i);
    return(0);
}
```

Le résultat affiché est :
L'addresse de la variable i est : 0x7fff58b0bb98

# Adresse et valeur

- Nous avons déjà vu que si A est une variable de type T, alors

$$\&A$$

  donne l'adresse de l'emplacement de A dans la mémoire.

- Si p est une variable de type pointeur vers une donnée de type T, alors

$$*p$$

  donne le contenu de l'emplacement mémoire dont l'adresse est p.

# Pointeurs

Ce que nous avons vu :

- Déclarer un pointeur :

  Type *Nom_de_la_variable;

- Deux opérateurs (unaires)
  - L'opération "&" pour récupérer l'adresse d'une variable
  - L'opération "*" pour récupérer le contenu d'une adresse

# Quelques Remarques

# Pointeurs : Pas seulement une adresse

- Un pointeur contient un adresse
- Mais également le type de données qui se trouve à cette adresse.
- Par exemple, lorsque l'on déclare :

$$int * p;$$

  - La présence de "*" indique que p est un pointeur (contient une adresse)
  - La présence de "int" indique que l'information pointée par *p* est de type "int".

# Exemple

Le programme C suivant :

```
...
 int i;
 int *p;
 double *p1;
...
```

# Exemple

Le programme C suivant :

```
...
 int i;
 int *p;
 double *p1;
...
```

contient les déclarations :

# Exemple

Le programme C suivant :

```
...
 int i;
 int *p;
 double *p1;
...
```

contient les déclarations :

- d'une variable *i* de type int,

# Exemple

Le programme C suivant :

```
...
 int i;
 int *p;
 double *p1;
...
```

contient les déclarations :

- d'une variable *i* de type int,
- d'une variable *p* de type adresse vers int, et

# Exemple

Le programme C suivant :

```
...
 int i;
 int *p;
 double *p1;
...
```

contient les déclarations :

- d'une variable *i* de type int,
- d'une variable *p* de type adresse vers int, et
- d'une variable *p*1 de type adresse vers double.

# Exemple

A partir de ces déclarations :

```
...
 int i;
 int *p;
 double *p1;
...
```

# Exemple

A partir de ces déclarations :

```
...
 int i;
 int *p;
 double *p1;
...
```

l'affectation :

# Exemple

A partir de ces déclarations :

```
...
 int i;
 int *p;
 double *p1;
...
```

l'affectation :

```
...
 p=&i;
...
```

est permise.

# Exemple

Par contre, à partir de ces déclarations :

```
...
 int i;
 int *p;
 double *p1;
...
```

# Exemple

Par contre, à partir de ces déclarations :

```
...
 int i;
 int *p;
 double *p1;
...
```

l'affectation :

```
...
 p1=&i;
...
```

# Exemple

Par contre, à partir de ces déclarations :

```
...
 int i;
 int *p;
 double *p1;
...
```

l'affectation :

```
...
 p1=&i;
...
```

n'est pas autorisée.

# Exemple

A partir de :

```
...
 int i=1;
 int *p;
 double *p1;
...
```

l'affectation avec cast :

```
...
 p1=(double *) &i;
 *p1++;
...
```

peut-être dangereuse à l'exécution dès que la taille de double est plus grande que celle de int.

# Le caractère ”*”
# double usage

# Le symbole ”*” : double usage

**Attention :** Le symbole ”*” a un double usage :

- Dans la déclaration des pointeurs, par exemple :

$$int * p;$$

- Lors de l'accès à l'information pointée par $p$, par exemple :

$$printf(''\%d'', *p);$$

- Ce double usage peut créer de la confusion.
- C'est vrai que ca aurait été mieux si on avait un mot clef pour déclarer les pointeurs (exemple *point*).

# L'opération "*" : n'accéder qu'aux espaces mémoires qui vous appartiennent

- Garder toujours en tête que la mémoire est partagée par plusieurs programmes.
- De ce fait, vous ne pouvez utiliser l'opérateur "*" que sur des espaces mémoires qui appartiennent à vos programmes.
- Par exemple, l'instruction:

$$A = *p;$$

n'a de sense que si :

  - l'espace mémoire qui se trouve dans l'adresse donnée par $p$ vous appartient.

# C’est valable aussi pour le scanf

Pour information (ou rappel),

$$scanf(\text{"}\%d\text{"}, \&a);$$

Signifie :

- lire un entier
- puis le placer au niveau de l’adresse où se trouve la variable *a*.
- Au fait, scanf est une fonction.
- Et nous verrons plus tard pourquoi dans certaines fonctions il est important de passer l’adresse des variables (comme içi dans scanf).

# C’est valable aussi pour le scanf

- Avec les pointeurs, on peut ré-écrire autrement les scanf.
- Par exemple, en déclarant :

$$int * p;$$

- on peut utiliser p dans le scanf:

$$scanf(\text{"}\%d\text{"}, p);$$

- Mais ce scanf ne peut se faire que si l’espace pointé par *p* vous appartient (exemple avant le scanf mettre p=&a).

# Opérations
# sur
# les pointeurs.

# Affectations

# Affectations d'une constante à un pointeur

- Un pointeur contient une adresse (ou un indice si on voit la mémoire comme un tableau d'octets),
- Un pointeur peut donc contenir un entier qui représente une adresse.
- De ce fait, on peut (en théorie) affecter une constante entière à un pointeur.

# Affectations d'une constante à un pointeur

Par exemple, on peut peut tenter d'affecter une constante entière (ici 13) au pointeur *p* (un pointeur vers un entier).

```
#include <stdio.h>
int main (void)
{
    int *p;
    p = 13;
    return 0;
}
```

# Affectations d'une constante à un pointeur

**Need your help**

# Affectations d'une constante à un pointeur

**Need your help**

**Un ''warning'' à la compilation**

# Affectations d'une constante à un pointeur

```
#include <stdio.h>
int main (void)
{
    int *p;
    p = 13;
    return 0;
}
```

En effet, à la compilation, un warning est généré :

warning: incompatible integer to pointer conversion assigning to 'int *' from 'int' [-Wint-conversion]

# Question?

**Comment**
**corriger ce programme?**

# Affectations d'une constante à un pointeur

```
1 #include <stdio.h>
2 int main (void)
3 {
4     int *p;
5     p = 13;
6     return 0;
7 }
```

Dans ce programme :

- La valeur 13 est une constante entière.
- La variable p est un pointeur qui contient un entier qui représente une adresse.
- Si on veut dire que 13 est un entier qui représente une adresse, il faut utiliser l'opérateur *cast* (conversion explicite).

# Affectations d'une constante à un pointeur

Voici donc le programme corrigé (avec l'impression de la valeur du pointeur) :

```
#include <stdio.h>
int main (void)
{
    int *p;
    p = (int *) 13;
    printf("La valeur de p est :%p\n", p);
    return 0;
}
```

# Affectations d'une constante à un pointeur

- Affecter une constante (de type adresse) à un pointeur n'est pas du tout utile,
- Car il existe très très peu de chance que l'adresse affectée au pointeur vous appartienne!
- Et gardez toujours en mémoire que seuls les espaces mémoires qui appartiennent à vos programmes peuvent-être modifiés.
- Son intérêt ici reste pédagogique
  - introduire le cast par exemple,
  - ou faire la différence entre un entier et une adresse.

# Autres affectations

- Soit $p_1$ et $p_2$ deux pointeurs du même type T.
- Soit A une variable de type T.
- Alors, les opérations suivantes d'affectation d'adresse à un pointeur sont permises :
  - $p_1 = p_2$
  - $p_1 = \&A$
  - $p_1 = \text{NULL}$

# Autres affectations

Nous verrons plus tard que nous pouvons affecter à une variable de type pointeur un nouvel espace mémoire grâce aux opérations d'allocations dynamiques de la mémoire comme malloc.

# Affectations entre les valeurs ”pointées” par les pointeurs

# Affectations

- Soit $p_1$ et $p_2$ deux pointeurs du même type T.
- Soit A une variable de type T.
- Alors, les opérations suivantes d'affectation de valeur sont permises :
  - $*p_1 = *p_2$
  - $*p_1 = A$
  - $A = *p_1$

# Exemple

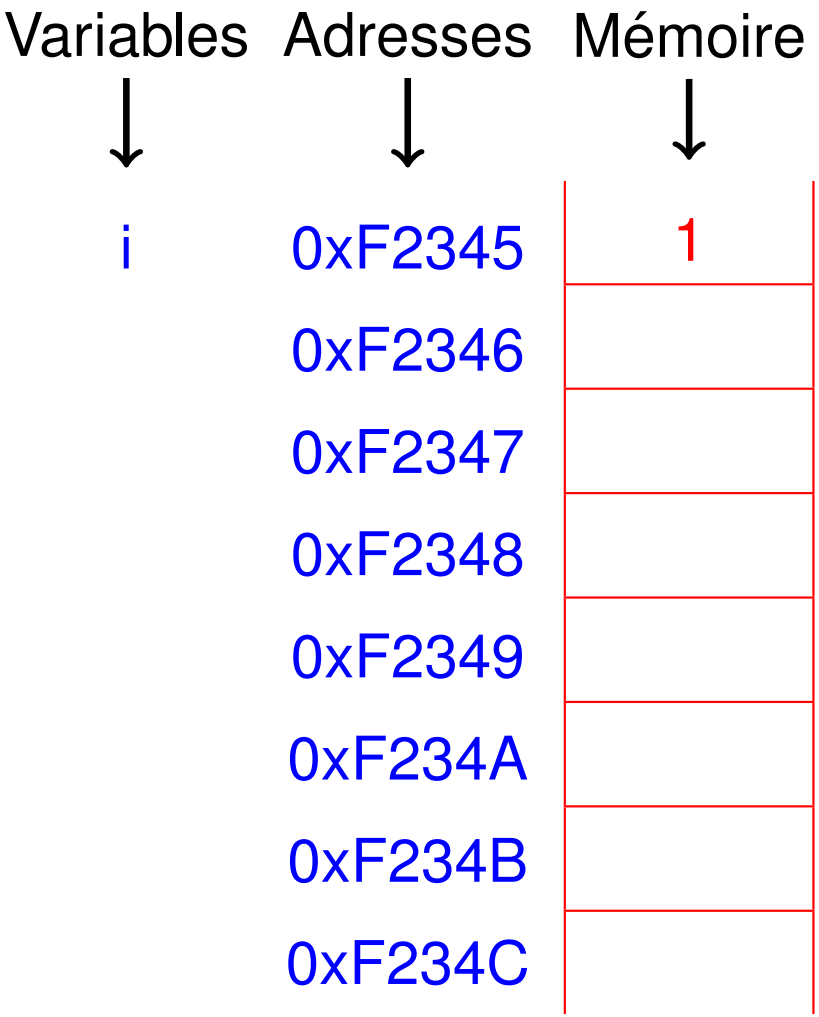
# Exemple détaillé

Exécutons le programme ci-dessous, et regardons l'état de la mémoire, après l'exécution de chaque instruction.

```
#include <stdio.h>
int main (void)
{
 int i=1;
 int *p1, *p2;
 p1=&i;
 p2=p1;
 *p2+=3;
 printf ("La valeur de i : %d  \n", i);
 return(0);
}
```

# Exemple détaillé

```
#include <stdio.h>
int main (void)
{
 int i=1;
 int *p1, *p2;
 p1=&i;
 p2=p1;
 *p2+=3;
 printf ("La valeur de i : %d  \n", i);
 return(0);
}
```

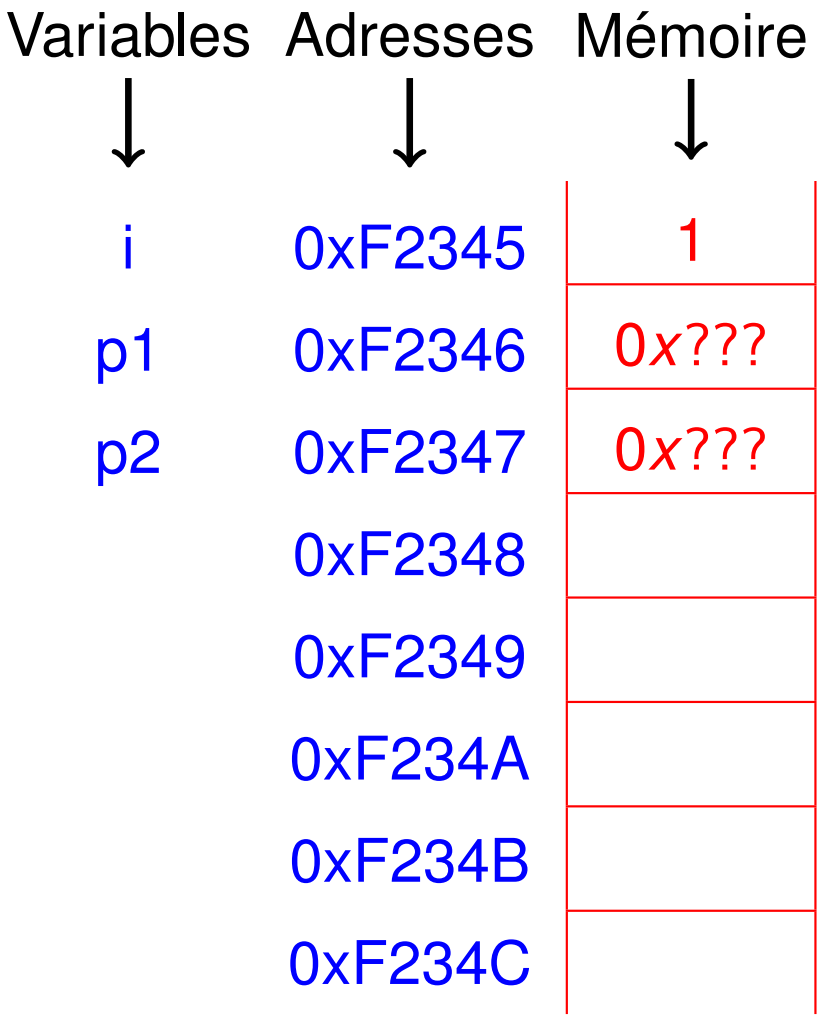


# Exemple détaillé

```
#include <stdio.h>
int main (void)
{
 int i=1;

 int *p1, *p2;

 p1=&i;
 p2=p1;
 *p2+=3;
 printf ("La valeur de i : %d  \n", i);
 return(0);
}
```

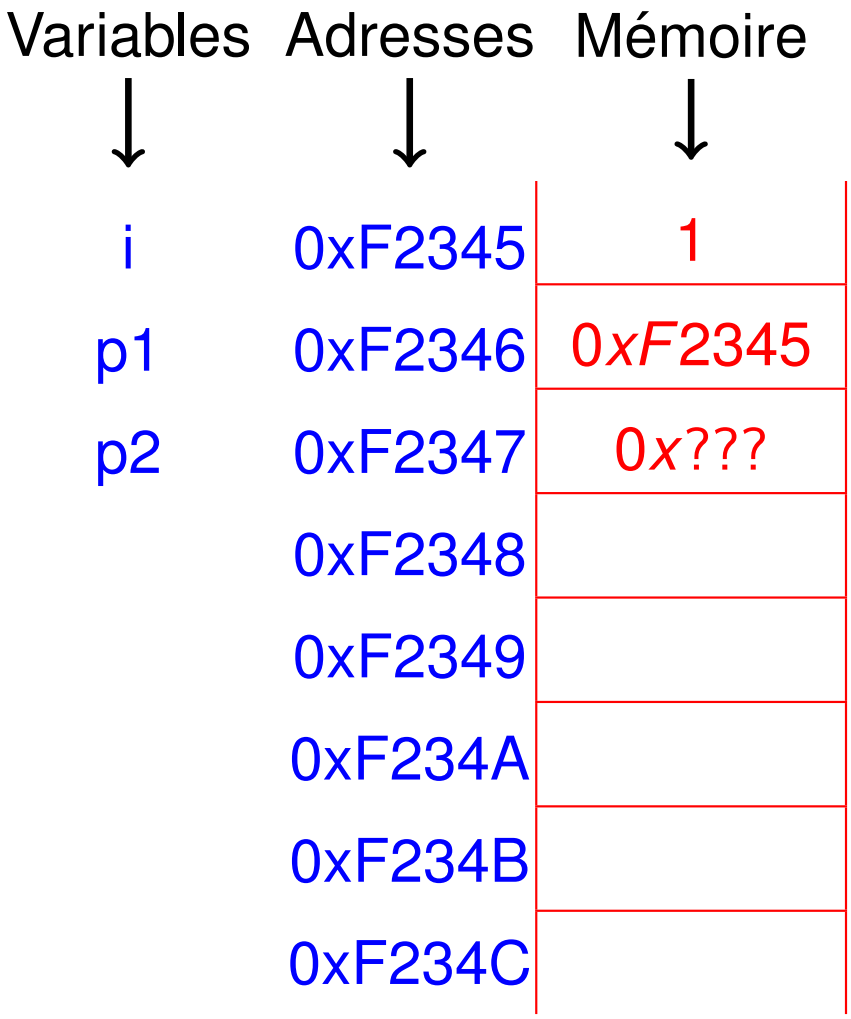


# Exemple détaillé

```
#include <stdio.h>
int main (void)
{
 int i=1;
 int *p1, *p2;

p1=&i;

 p2=p1;
 *p2+=3;
 printf ("La valeur de i : %d  \n", i);
 return(0);
}
```

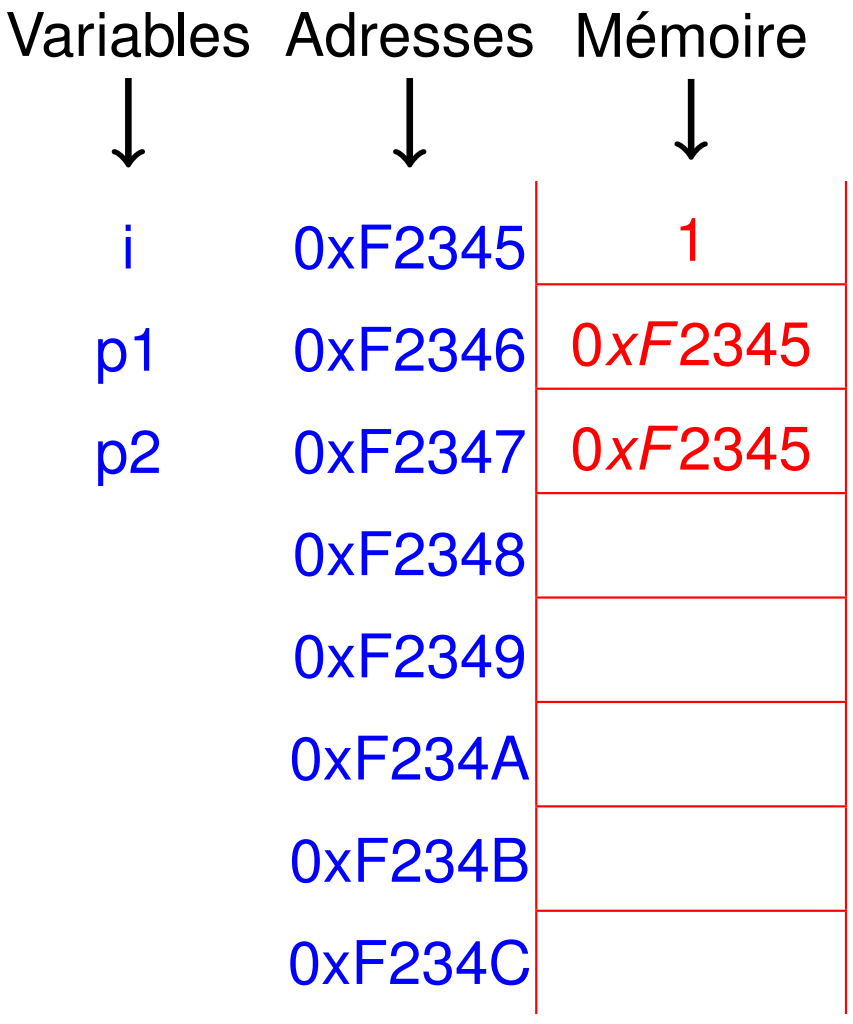


# Exemple détaillé

```
#include <stdio.h>
int main (void)
{
 int i=1;
 int *p1, *p2;
 p1=&i;

 p2=p1;

 *p2+=3;
 printf ("La valeur de i : %d  \n", i);
 return(0);
}
```

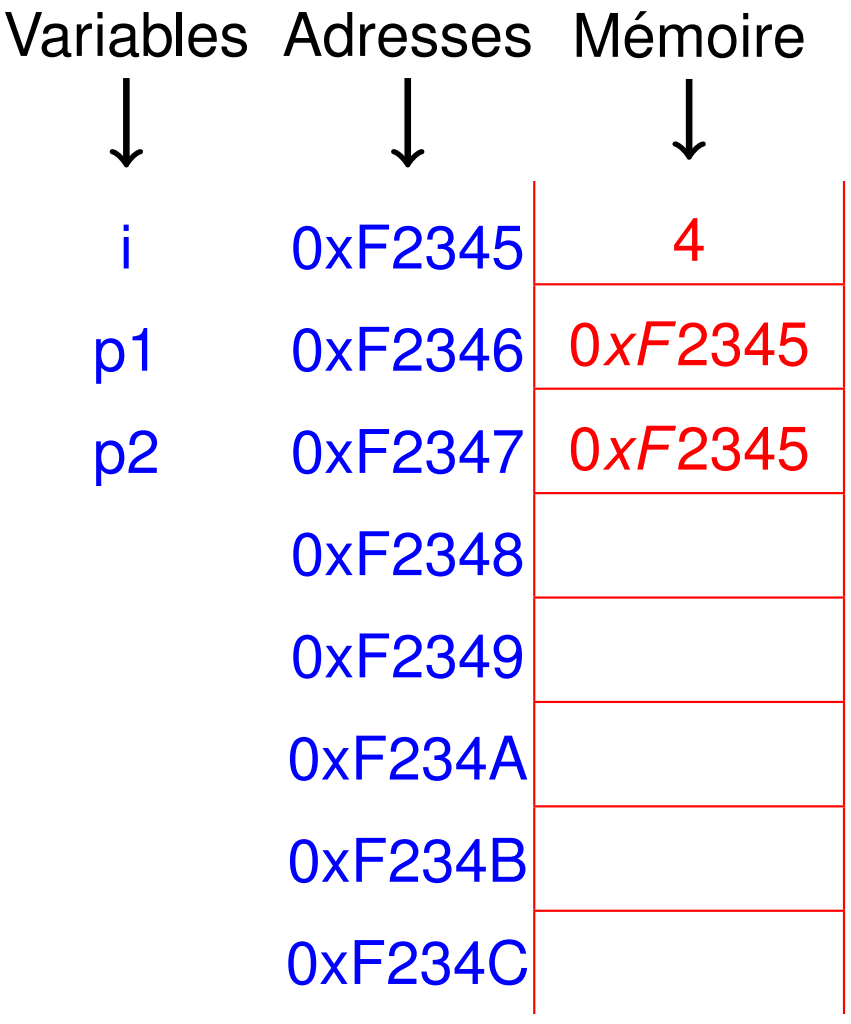


# Exemple détaillé

```
#include <stdio.h>
int main (void)
{
 int i=1;
 int *p1, *p2;
 p1=&i;
 p2=p1;

 *p2+=3;

 printf ("La valeur de i : %d  \n", i);
 return(0);
}
```



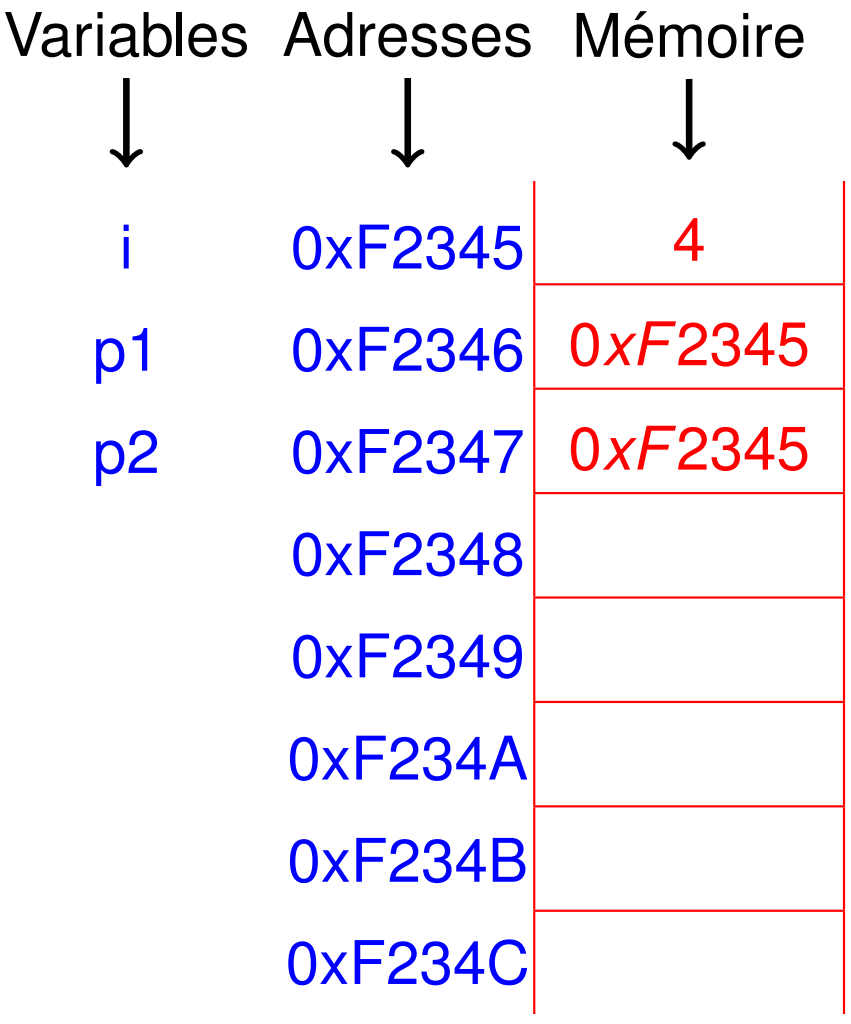
# Exemple détaillé

```
#include <stdio.h>
int main (void)
{
 int i=1;
 int *p1, *p2;
 p1=&i;
 p2=p1;
 *p2+=3;

 printf ("La valeur de i : %d  \n", i);

 return(0);
}
```

Sans surprise le programme affichera la valeur 4.



# Remarques sur les opérations sur les pointeurs

- Si p est un pointeur vers une donnée de type T, alors :
  - *p se comporte comme une variable ordinaire de type T, et
  - peut-être utilisée, par exemple, dans une expression arithmétique et booléenne.
- Il est permis d'utiliser certaines opérations arithmétiques sur les pointeurs (++, −, + un entier, - un entier),
- comme il est possible d'avoir des instructions comparatives entre pointeurs .

# Remarques sur les opérations sur les pointeurs

- Par exemple, les instructions :
  - $p = p + 1;$
  - $p = p + 10;$
  - $p - -;$
  - $p_1 > p_2$

  sont permises.

# Remarques sur les opérations sur les pointeurs

- Si p est un pointeur vers une donnée de type T, alors après l'instruction :
  - $p = p + 1;$

  p aura comme valeur l'ancienne adresse plus la taille de T (en nombre d'octets).
- Attention, après une telle affectation, (*p) n'a de sens que si l'espace mémoire est bien réservé par le programme.