Roadmap

Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Optimizations
Memory & caches

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

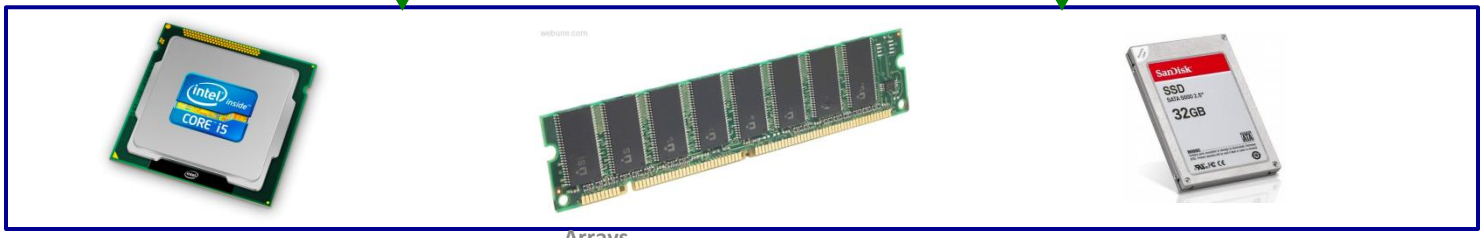
```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
11000001111110100011111
```

OS:

Computer system:



Cours 6: Tableaux & Structures de données

- **Tableaux**
  - Tableau multi-dimensions vs Tableau multi-niveaux
  - C vs Java
- **Structures de données**
  - Alignement
  - C vs Java

Tableau multi-dimensions vs Tableau multi-niveaux

```
typedef int zip_dig[5];

#define PCOUNT 3
zip_dig sea[PCOUNT] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 1, 5 }};
```

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig uw  = { 9, 8, 1, 9, 5 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

```
#define UCOUNT 3
int *univ[UCOUNT] = {uw, cmu, ucb};
```

Accès aux éléments

Tableau multi-dimensions

```
int get_sea_digit
   (int index, int dig)
{
  return sea[index][dig];
}
```

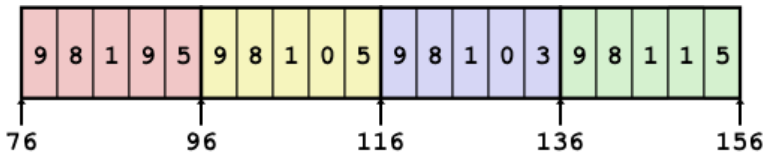
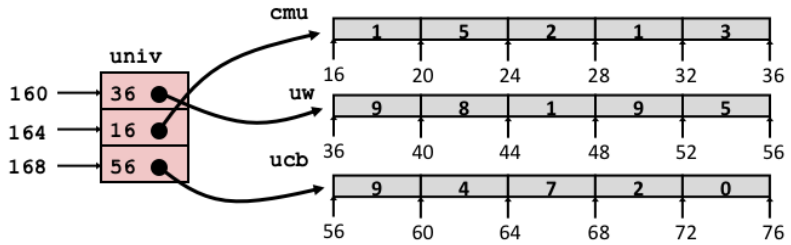


Tableau multi-niveaux

```
int get_univ_digit
   (int index, int dig)
{
  return univ[index][dig];
}
```



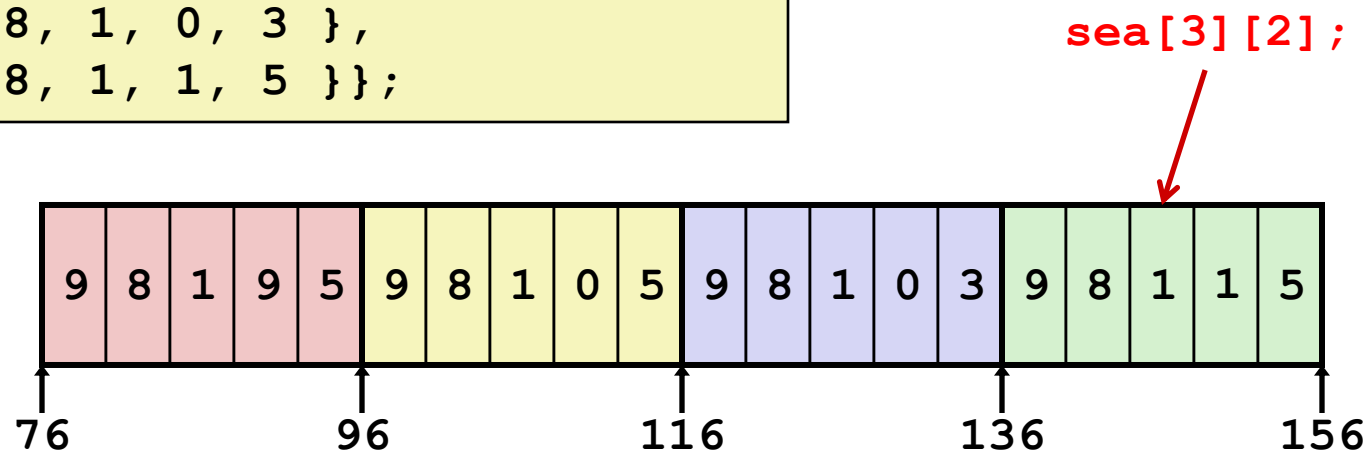
Accès différents !

`Mem[sea+20*index+4*dig]`

`Mem[Mem[univ+4*index]+4*dig]`

Tableaux multi-dimensions

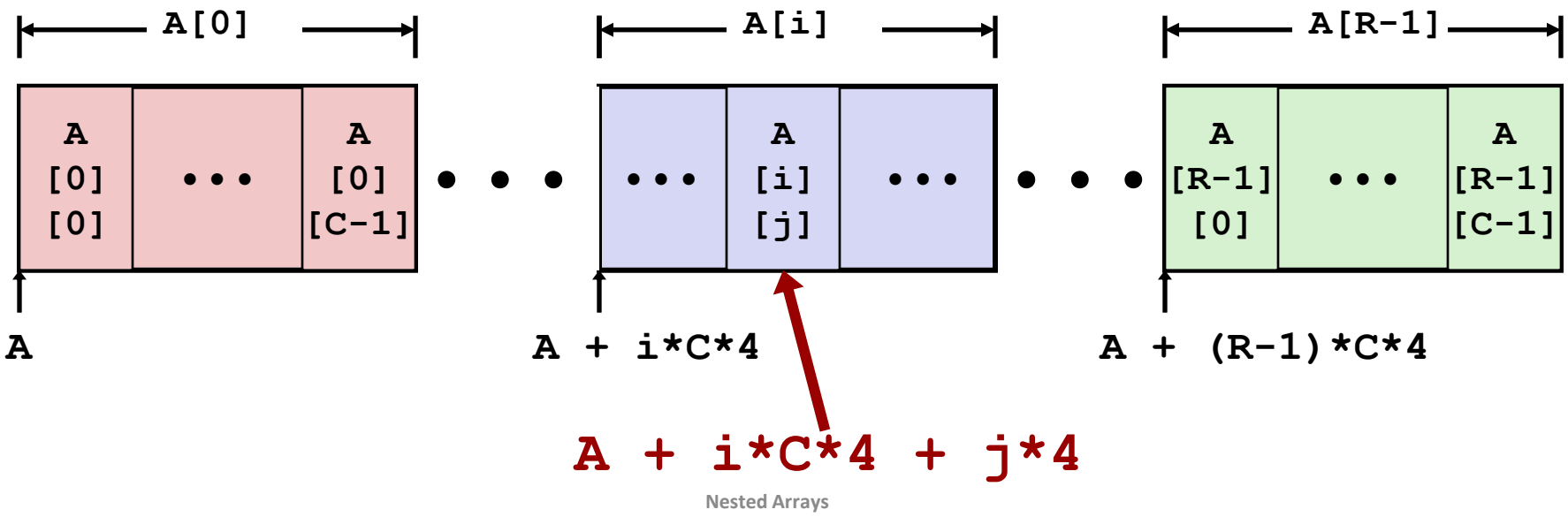
```
#define PCOUNT 4
zip_dig sea[PCOUNT] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```



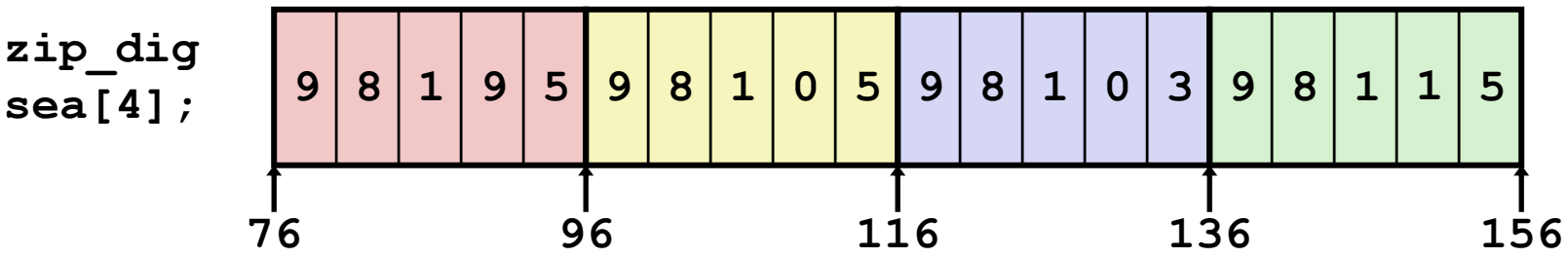
- Les éléments sont organisés en “row-major” (garanti)

Tableaux multi-dimensions

```
int A[R][C];
```



Exemples de référence



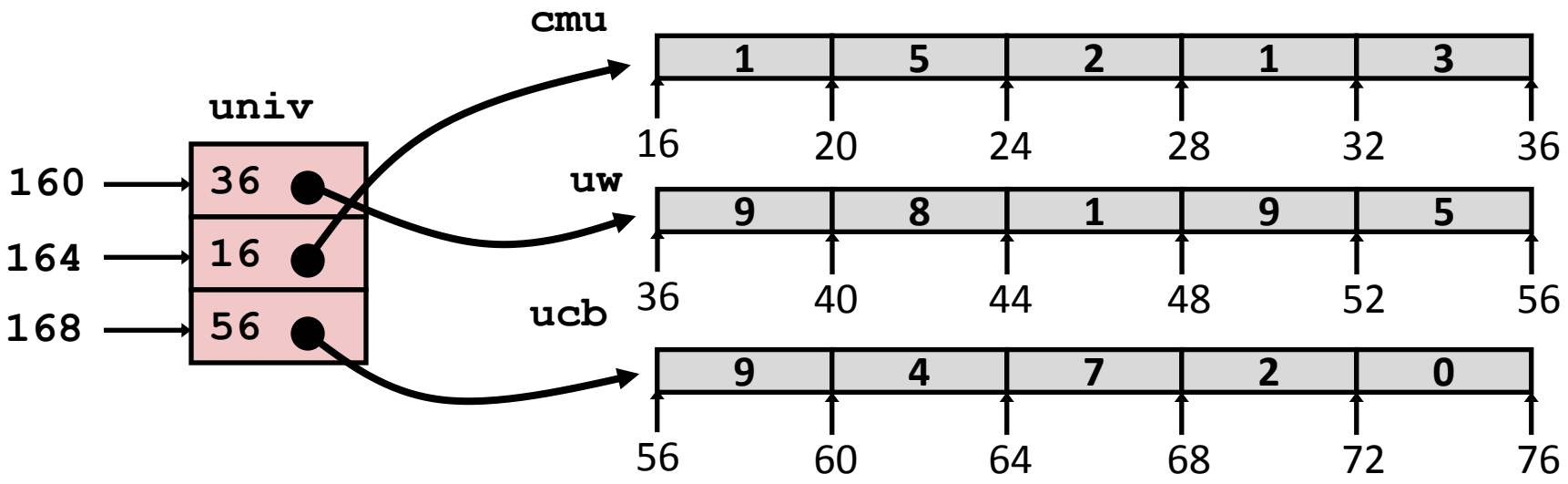
Référence	Adresse	Valeur	Garantie ?
`sea[3][3]`	`76+20*3+4*3 = 148`	`1`	Yes
`sea[2][5]`	`76+20*2+4*5 = 136`	`9`	Yes
`sea[2][-1]`	`76+20*2+4*-1 = 112`	`5`	Yes
`sea[4][-1]`	`76+20*4+4*-1 = 152`	`5`	Yes
`sea[0][19]`	`76+20*0+4*19 = 152`	`5`	Yes
`sea[0][-1]`	`76+20*0+4*-1 = 72`	`??`	No

- Pas de bounds checking !
- Ordre des éléments dans le tableau est garanti.

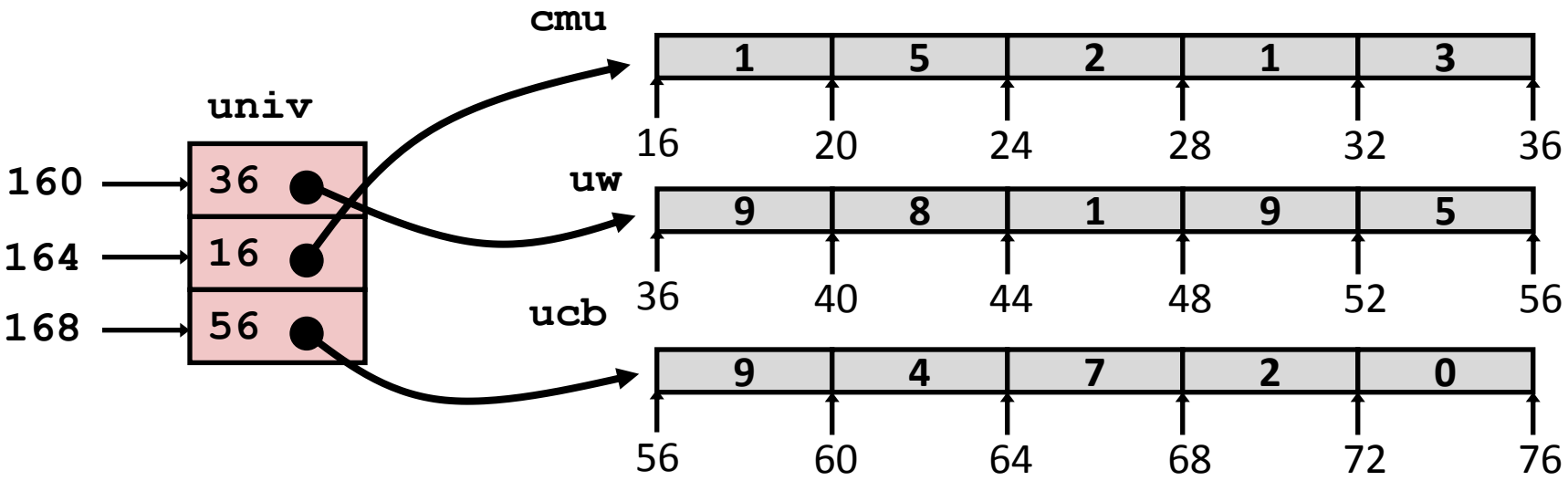
Tableaux multi-niveaux

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig uw  = { 9, 8, 1, 9, 5 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

```
#define UCOUNT 3
int *univ[UCOUNT] = {uw, cmu, ucb};
```



Exemples de référence



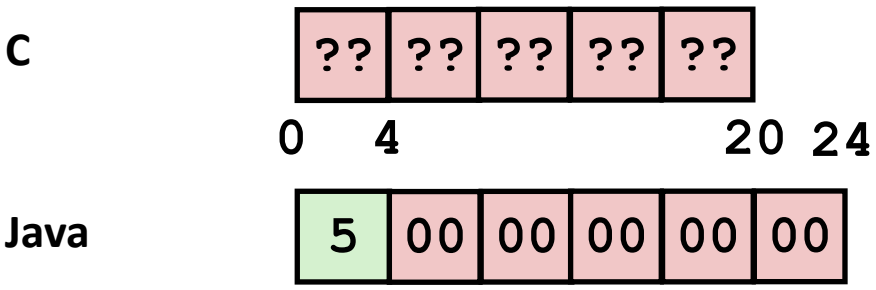
Référence	Adresse	Valeur	Garantie ?
`univ[2][3]`	`56+4*3 = 68`	`2`	Yes
`univ[1][5]`	`16+4*5 = 36`	`9`	No
`univ[2][-1]`	`56+4*-1 = 52`	`5`	No
`univ[3][-1]`	`??`	`??`	No
`univ[1][12]`	`16+4*12 = 64`	`7`	No

- Pas de bounds checking !

Tableaux en Java

- **Chaque élément est initialisé à 0**
- **La longueur est spécifiée dans un champs caché au début de tableau (4 octets)**
  - *array.length*
  - *Hmm, qu'est-ce qu'il fait avec cette information ?*

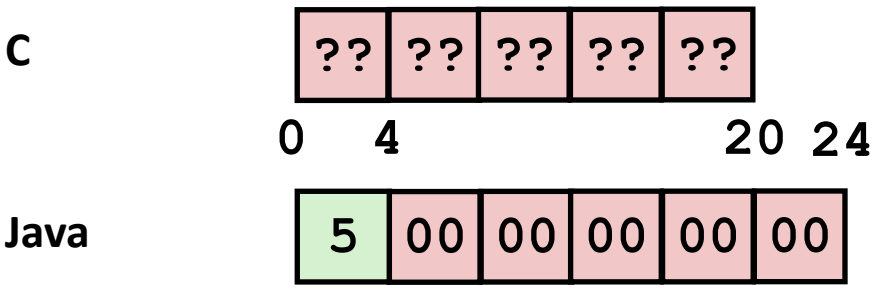
int array[5]:



Tableaux en Java

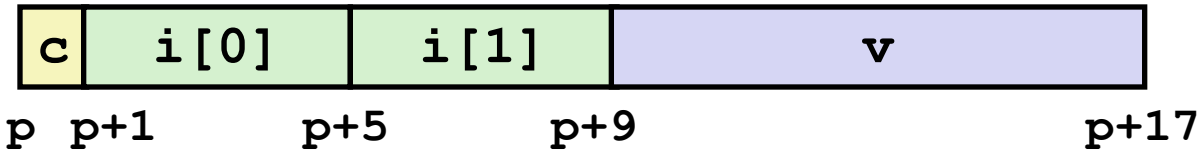
- **Chaque élément est initialisé à 0**
- **La longueur est spécifiée dans un champs caché au début de tableau (4 octets)**
  - Chaque accès au tableau émet un bounds-check
    - Code ajouté
    - Exception si out-of-bounds

int array[5]:



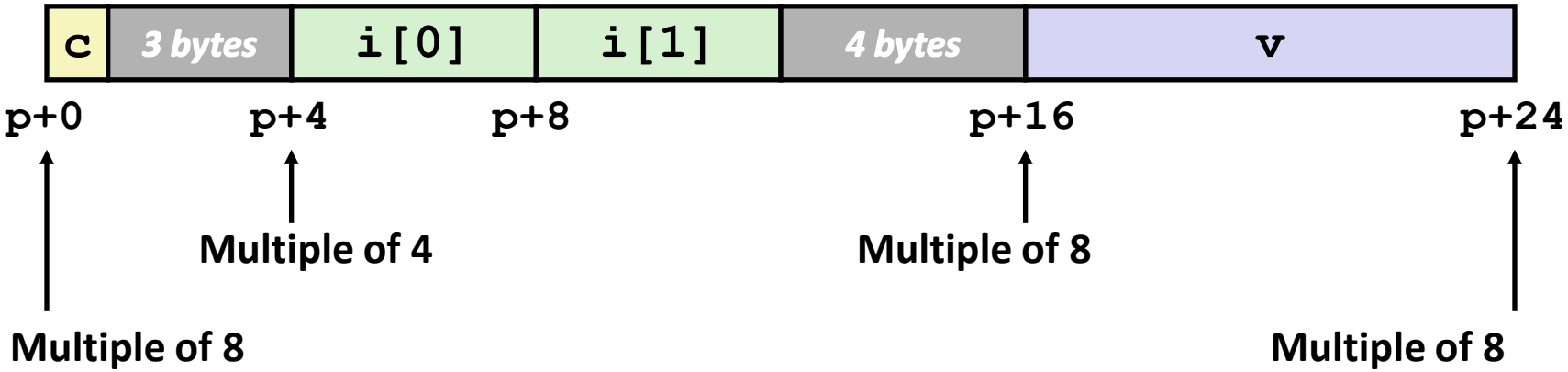
Structures & Alignement

- Données non alignées




```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

- Données alignées



Principes d'alignement

- **Données alignées**
  - Si le type primitif demande K octets, l'adresse doit être un multiple de K
- **Alignement de données est obligatoire sur certaines machines et « recommandé » sur IA32**
  - Traité différemment avec IA32 Linux, x86-64 Linux, et Windows!
- **Motivation**
  - Mémoire physique est accédée par « chunks » de 4 ou 8 octets, load et store plus efficace
- **Compilateur**
  - Insérer « padding » en structure pour assurer l'alignement correct des champs
  - => Utiliser **`sizeof()`** pour avoir la vraie taille de structure !

Cas spécifiques d'alignement (IA32)

- **1 byte: char, …**
  - Pas de restriction en adresse
- **2 bytes: short, …**
  - Le bit le plus bas de l'adresse doit être 0_2
- **4 bytes: int, float, char *, …**
  - Les 2 bits les plus bas de l'adresse doivent être 00_2
- **8 bytes: double, …**
  - Windows (et autres OSs & jeux d'instruction): 3 bits plus bas 000_2
  - Linux: 2 bits plus bas 00_2
- **12 bytes: long double**
  - Windows, Linux : 2 bits plus bas 00_2

Alignement avec structures

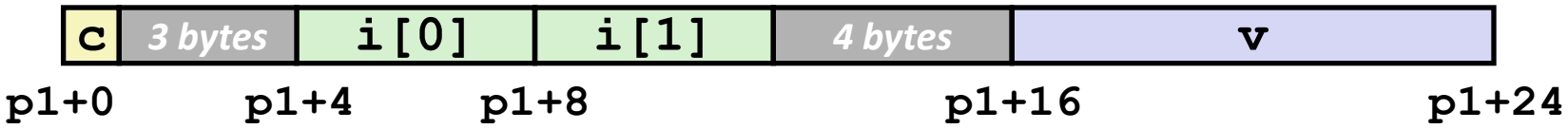
- **Dans la structure:**
  - Respecter l'exigence d'alignement de chaque membre
- **Placement de la structure**
  - Chaque structure a une exigence d'alignement K
    - K = alignement le plus grand
  - Adresse initiale et longueur de la structure doivent être un multiple de K

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p1;
```

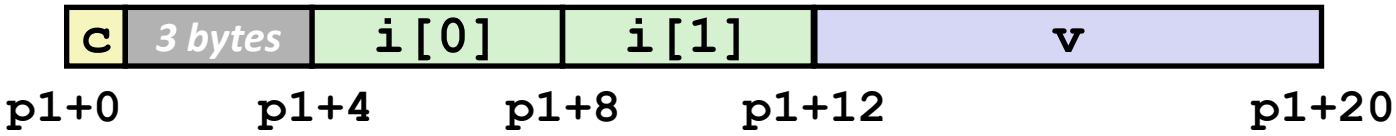
Différentes conventions d'alignement

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p1;
```

- **IA32 Windows or x86-64:**
  - K = 8



- **IA32 Linux:**
  - K = 4



Sauver de la place !

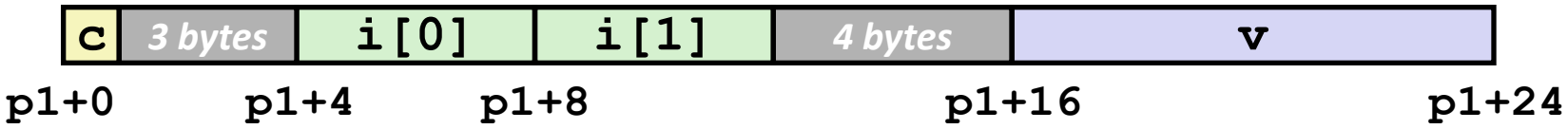
- **Mettre les données de type plus grand d'abord :**

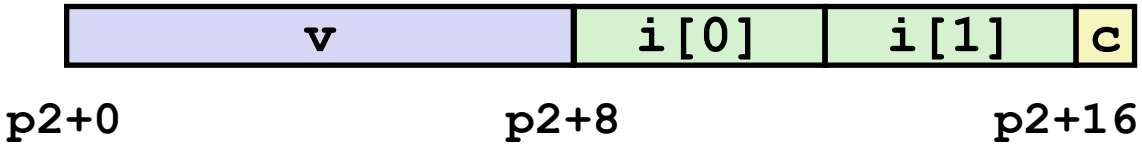
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p1;
```



```
struct S2 {
  double v;
  int i[2];
  char c;
} *p2;
```

- **Effet (exemple x86-64, K=8)**

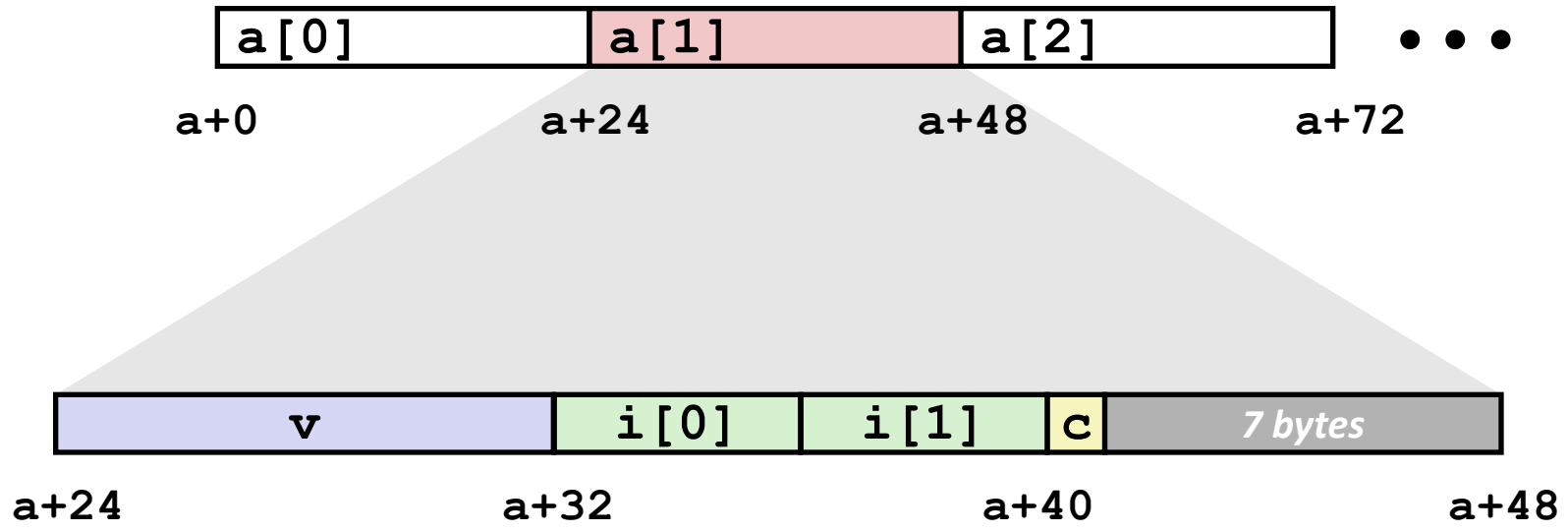




Malheureusement, la taille totale de la structure n'est pas un multiple de K

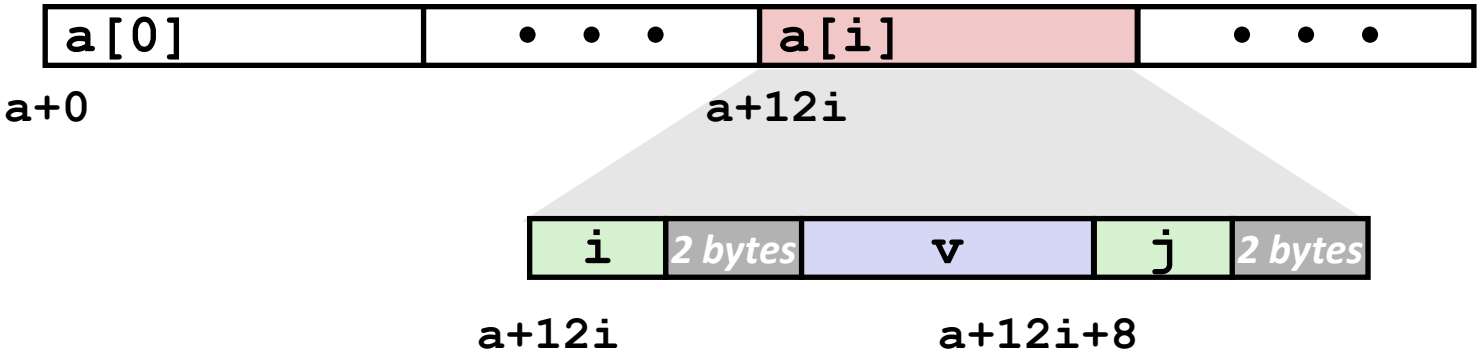
Tableau de structures

```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Un autre exemple

```
struct S3 {
  short i;
  float v;
  short j;
} a[10];
```



Structures de données (objets) en Java

- Les objets (structs) peuvent inclure des types primitifs seulement, ou il faut utiliser les références pour les types complexes (tableaux, d'autres objets, etc.)

C

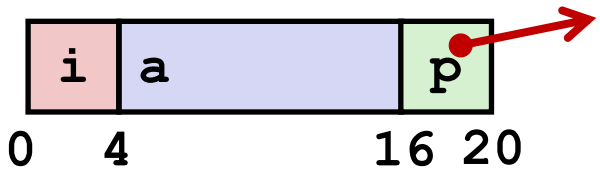
```
struct rec {
  int i;
  int a[3];
  struct rec *p;
};
```

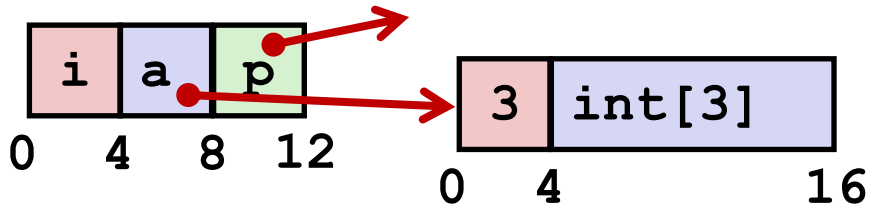
Java

```
class Rec {
  int i;
  int[] a = new int[3];
  Rec p;
…
};
```

```
struct rec *r = malloc(...);
struct rec r2;
r->i = val;
r->a[2] = val;
r->p = &r2;
```

```
r = new Rec;
r2 = new Rec;
r.i = val;
r.a[2] = val;
r.p = r2;
```



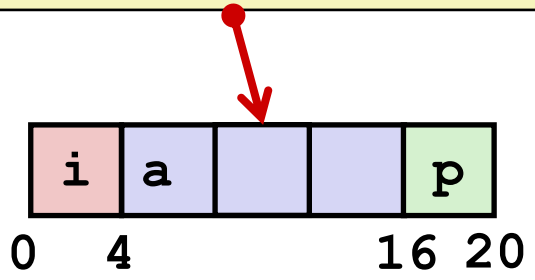


Pointers/Références

- **Pointer en C peut pointer à n'importe quelle adresse mémoire**
- **Référence en Java peut seulement pointer à un objet**
  - Et seulement à son 1er élément !

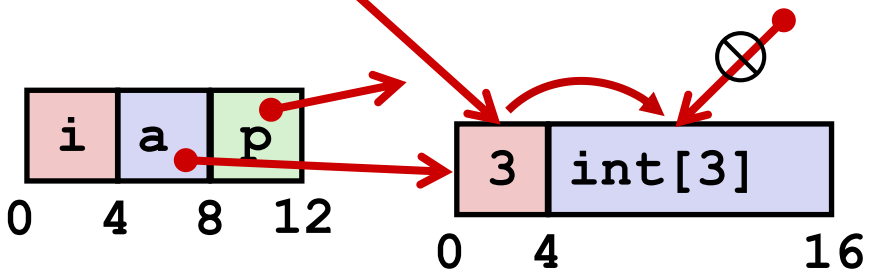
C

```
struct rec {
  int i;
  int a[3];
  struct rec *p;
};
some_fn(&(r.a[1]))   //ptr
```



Java

```
class Rec {
  int i;
  int[] a = new int[3];
  Rec p;
…
};
some_fn(r.a, 1)   //ref & index
```



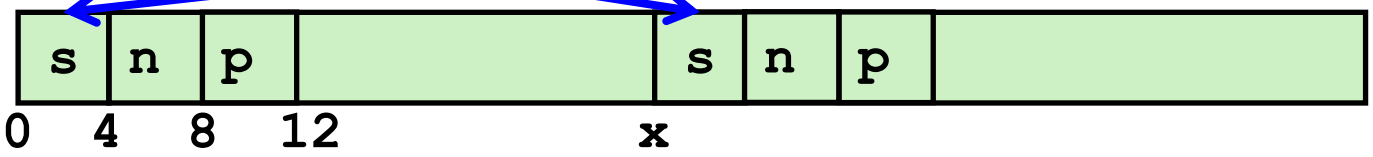
Casting en C

- On peut caster un pointer en n'importe quel autre pointer

```
struct BlockInfo {
        int sizeAndTags;
        struct BlockInfo* next;
        struct BlockInfo* prev;
};
typedef struct BlockInfo BlockInfo;
…
int x;
BlockInfo *b;
BlockInfo *newBlock;
…
newBlock = (BlockInfo *) ( (char *) b + x );
…
```

Cast b into char pointer so that you can add byte offset without scaling

Cast back into BlockInfo pointer so you can use it as BlockInfo struct



Casting en C : petit test

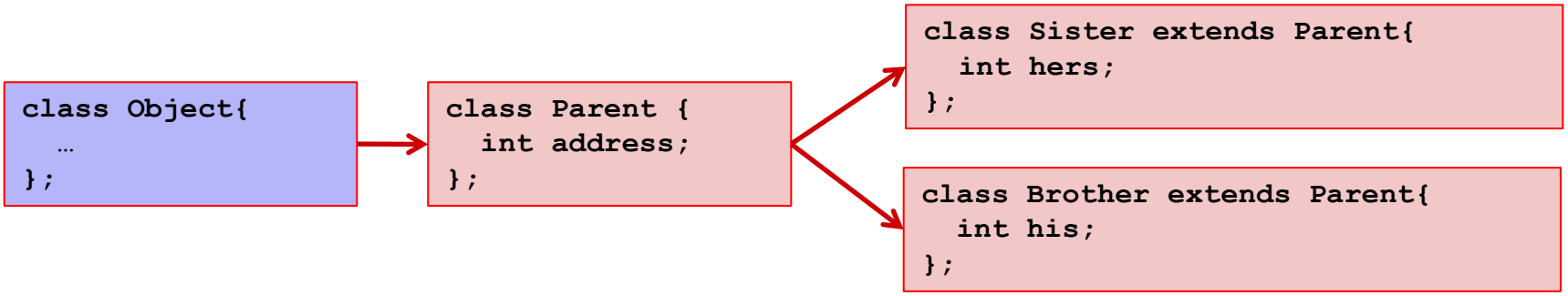
- **char * p**

- **(int *) p +7**

- **(int *)(p+7)**

Casting en Java

- **Peut seulement caster des références d'objets compatibles**




```
// Parent is a super class of Brother and Sister, which are siblings
Parent    a = new Parent();
Sister   xx = new Sister();
Brother  xy = new Brother();
Parent   p1 = new Sister();          // ok, everything needed for Parent
                                     // is also in Sister
Parent   p2 = p1;                    // ok, p1 is already a Parent
Sister  xx2 = new Brother();         // incompatible type – Brother and
                                     // Sisters are siblings
Sister  xx3 = new Parent();          // wrong direction; elements in Sister
                                     // not in Parent (hers)
Brother xy2 = (Brother) a;           // run-time error; Parent does not contain
                                     // all elements in Brother (his)
Sister  xx4 = (Sister) p2;  // ok, p2 started out as Sister
Sister  xx5 = (Sister) xy;  // inconvertible types, xy is Brother
```

Création d'objet en Java

```
class Point {
        double x;
        double y;

Point() {
        x = 0;
        y = 0;
    }

boolean samePlace(Point p) {
        return (x == p.x) && (y == p.y);
    }

}
…
Point newPoint = new Point();
…
```

champs

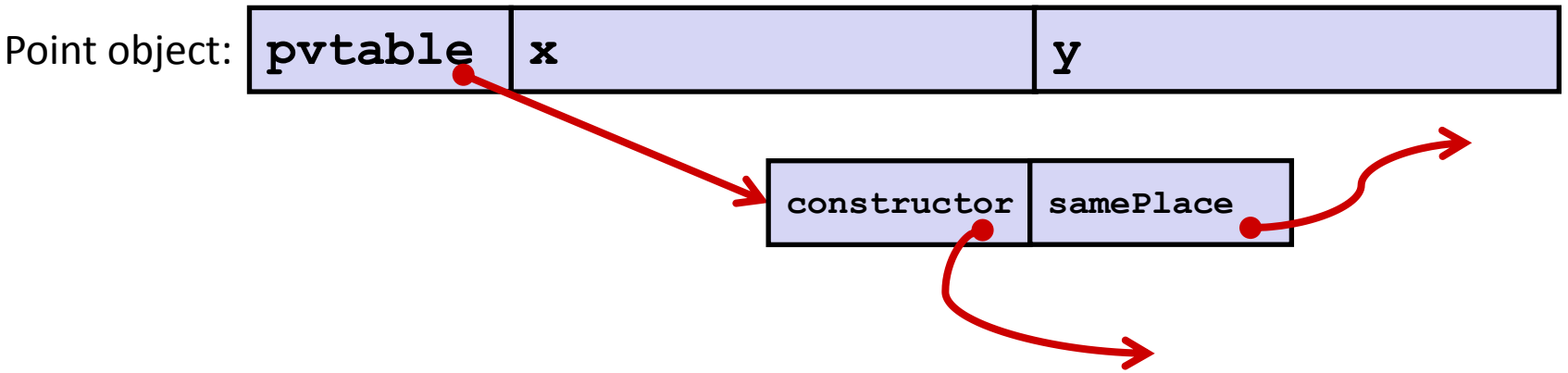
constructeur

méthode

création

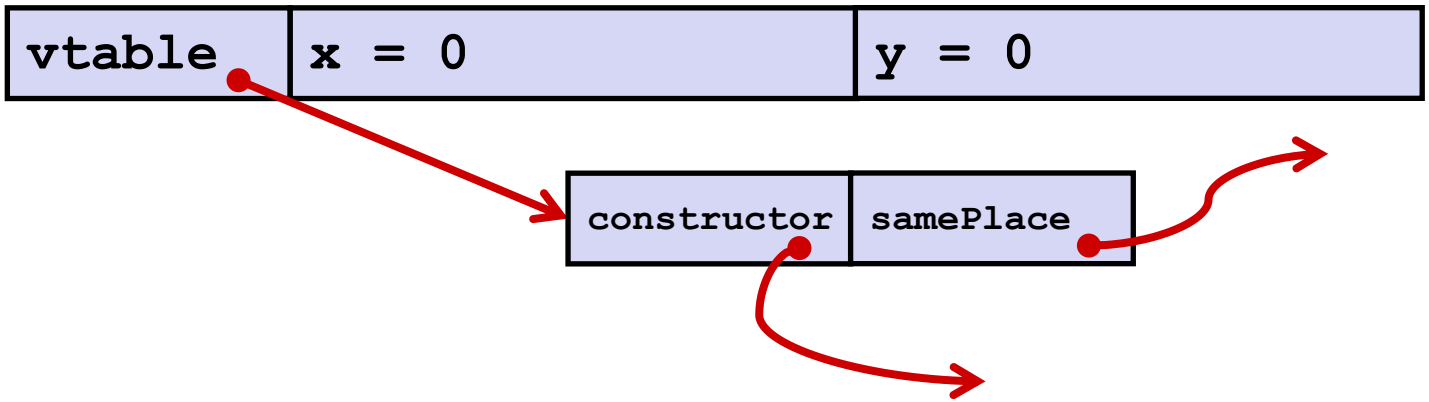
Création d'objet en Java

- "new"
  - Allouer de l'espace pour les champs de données
    - Ajouter un pointer à "vtable" (table virtuelle, partagée par tous les objets de la classe, contenant des pointers aux codes des méthodes et les champs « static »)
  - Retourner la référence au nouvel objet en mémoire
  - Exécuter la méthode "constructeur"



Création d'objet en Java

- **"new"**
  - Allouer de l'espace pour les champs de données
    - Ajouter un pointer à "vtable" (table virtuelle, partagée par tous les objets de la classe, contenant des pointers aux codes des méthodes et les champs « static »)
  - Retourner la référence au nouvel objet en mémoire
  - Exécuter la méthode "constructeur"
- **Initialisation :**



Subclassing

```
class PtSubClass extends Point{
   int aNewField;
   boolean samePlace(Point p2) {
       return false;
   }
   void sayHi() {
      System.out.println("hello");
   }
 }
```

- “aNewField” va où ?
- Le pointer vers le code de 2 nouvelles méthodes va où ?

Subclassing

```
class PtSubClass extends Point{
   int aNewField;
   boolean samePlace(Point p2) {
       return false;
   }
   void sayHi() {
      System.out.println("hello");
   }
 }
```