Variables et données

Vincent Archambault-B
IFT 2035 - Université de Montréal

Ce document est dédié au domaine public via CCO

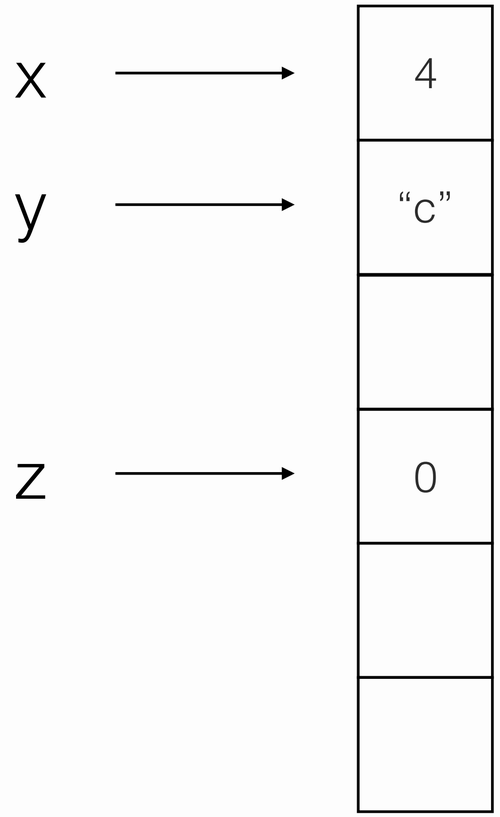
Pour obtenir le code source de ce document

- https://github.com/archambaultv/IFT2035-UdeM
- vincent.archambault-bouffard@umontreal.ca

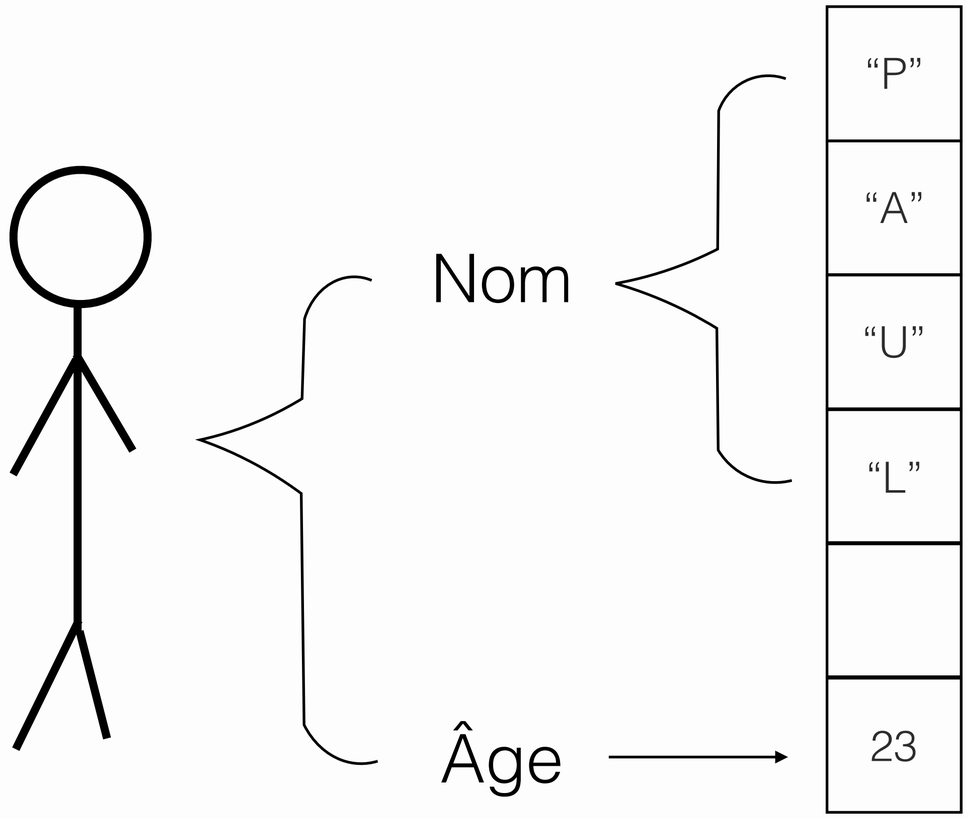
Rappel

Variable Espace de stockage associé avec un identificateur. L'espace de stockage peut être abstrait ou en lien avec un modèle de la mémoire.

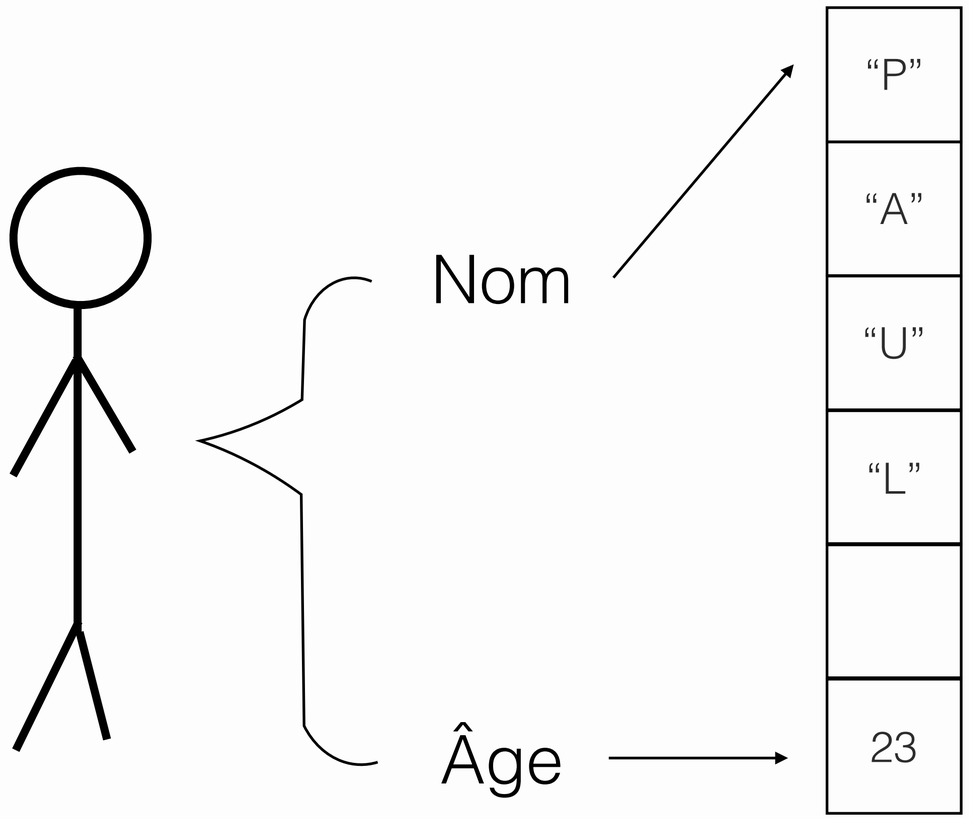
Variable ↔1 espace mémoire



Données ↔ plusieurs espaces mémoire



Variables et espace mémoire continu

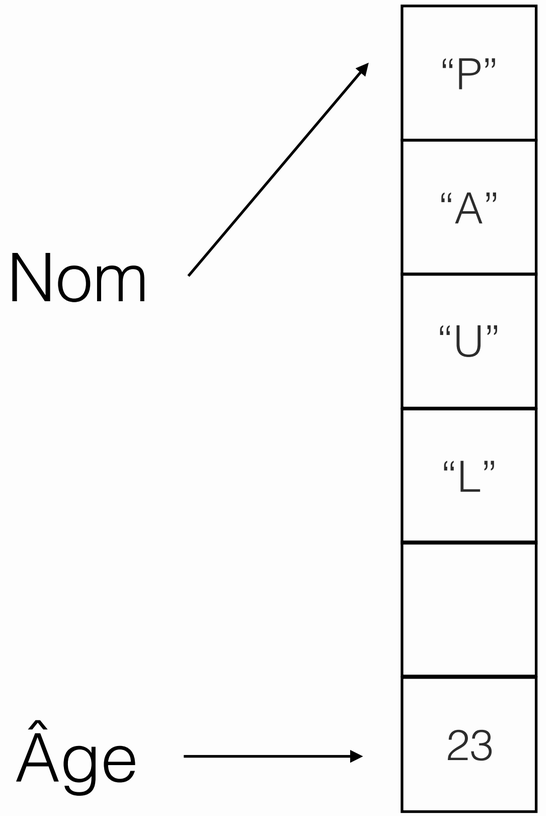


Variables et espace mémoire continu

```
char nom[] = "Paul";

int age = 23;
```

Code C pour les variables nom et âge

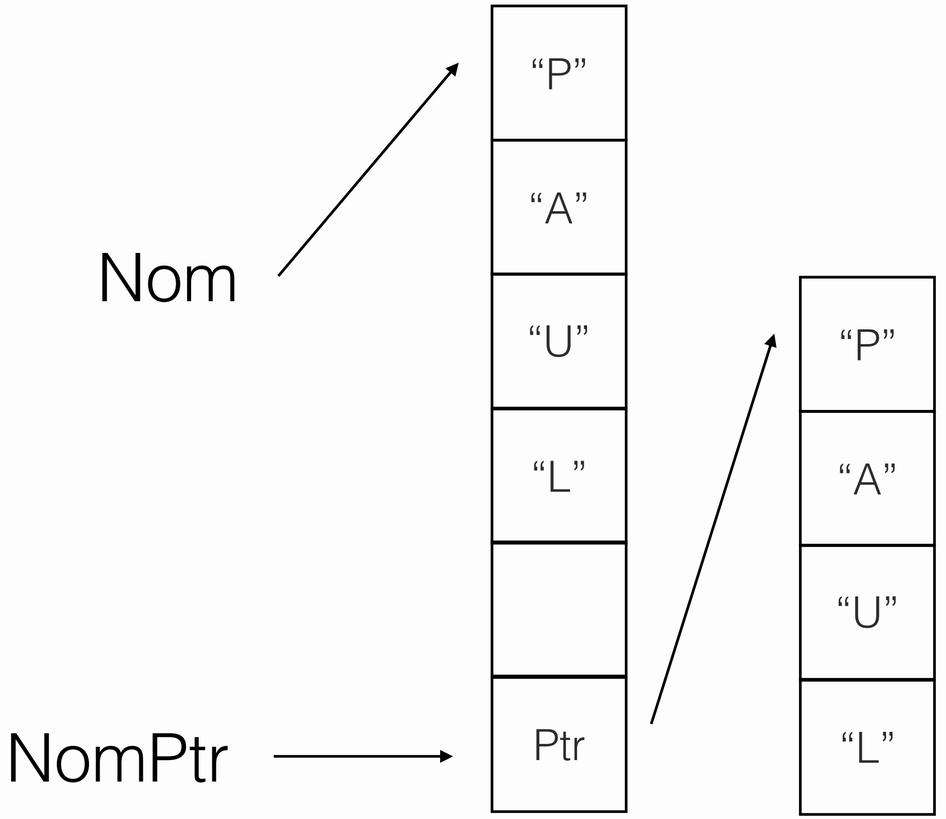


En C : char nom[] ≠ char* nom

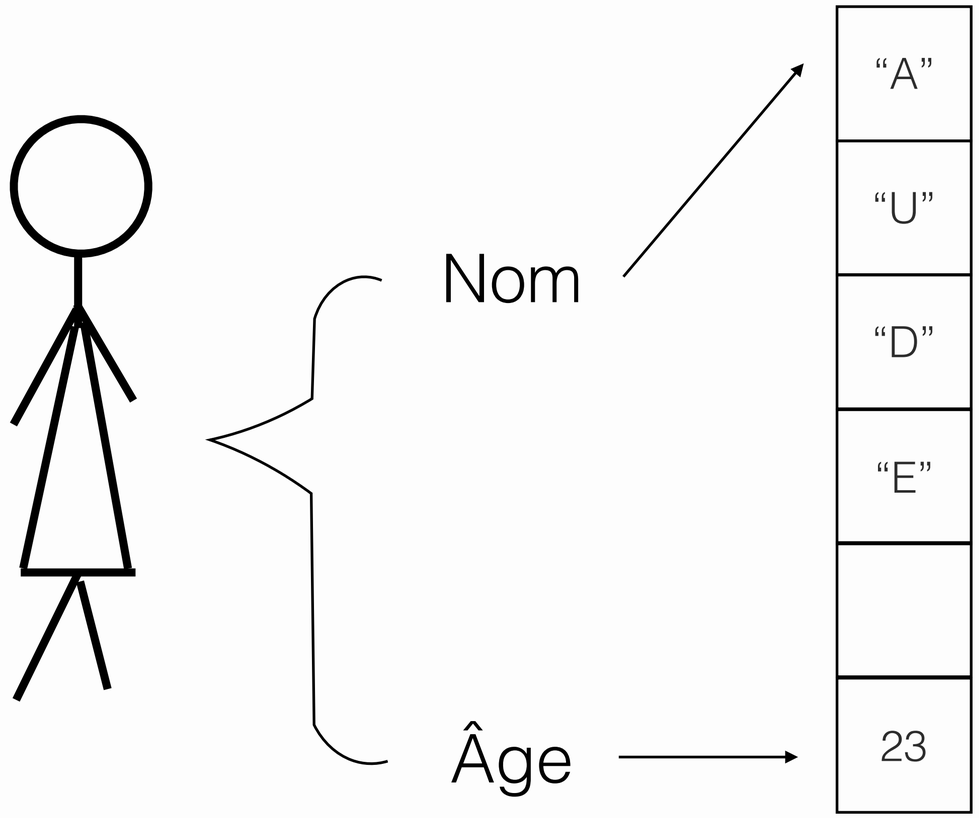
```
char nom[] = "Paul";

char* nomPtr = "Paul";
```

Code C pour les variables nom et âge



Personne = Nom + Âge

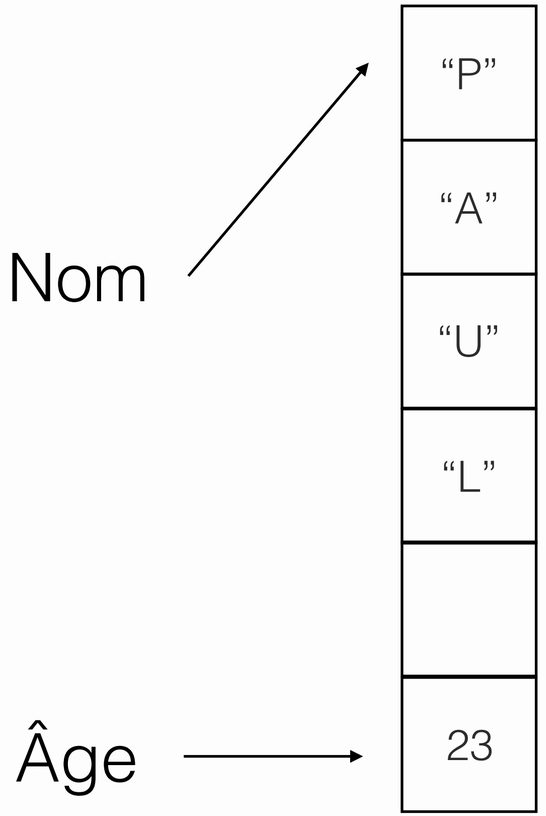


Comment regrouper Nom et Âge ?

```
char nom[] = "Paul";

int age = 23;
```

Code C pour les variables nom et âge

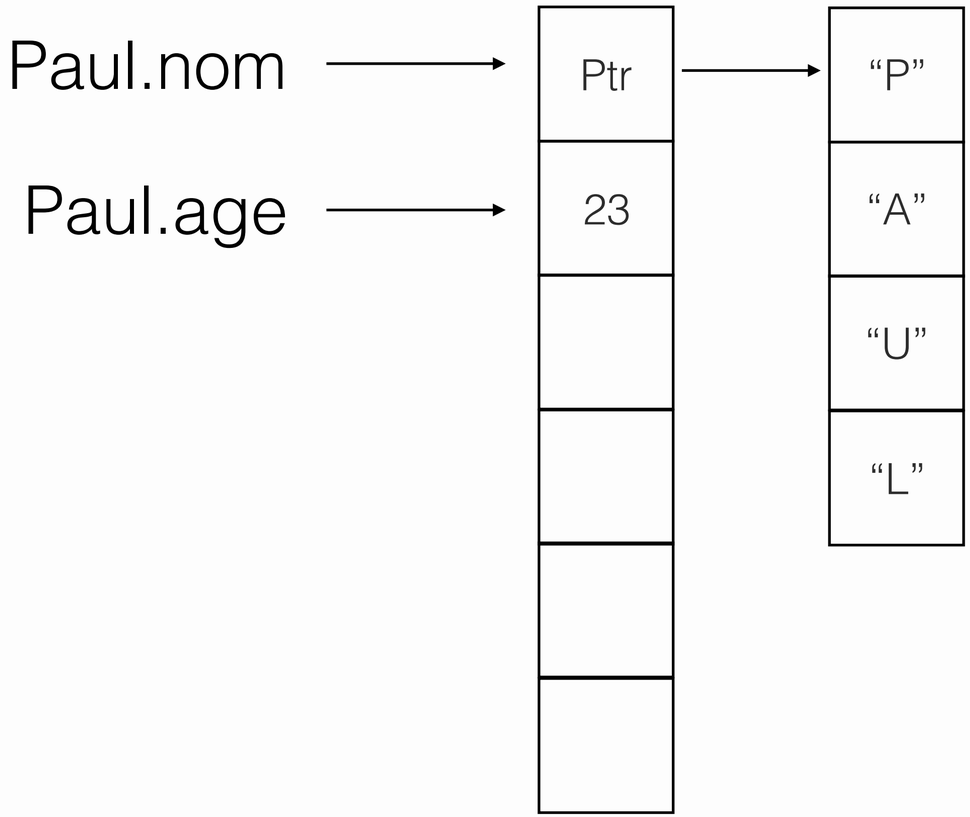


Struct en C

```
struct person {
   char* nom;
   int age;
};

struct person paul = {"Paul", 23};
```

Code C pour une struct person

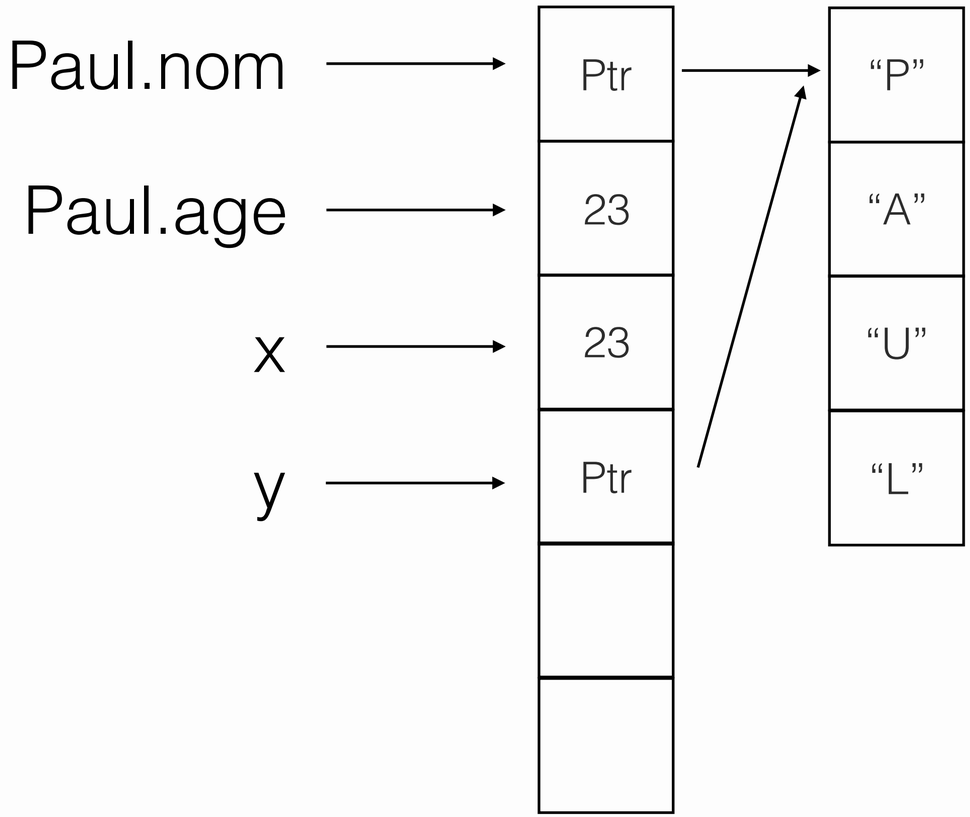


Struct en C

```
struct person {
   char* nom;
   int age;
};

struct person paul = {"Paul", 23};

int x = paul.age;
char* y = paul.nom;
```



Code C pour les variables nom et âge

Struct en C : typedef

```
struct person {
   char* nom;
   int age;
};

struct person paul = {"Paul", 23};

struct person foo(struct person p){
  struct person noName =
     {“NoName”, p.age};
  return noName;
}
```

Code C pour avec struct person

```
struct person {
   char* nom;
   int age;
};

typedef struct person person;

person paul = {"Paul", 23};

person foo(person p){
  person noName = {“NoName”, p.age};
  return noName;
}
```

Code équivalent utilisant typedef

Struct en C : typedef

```
struct person {
   char* nom;
   int age;
};

struct person paul = {"Paul", 23};

struct person foo(struct person p){
  struct person noName =
     {“NoName”, p.age};
  return noName;
}
```

Code C pour avec struct person

```
struct person {
   char* nom;
   int age;
};

typedef struct person toto;

toto paul = {"Paul", 23};

toto foo(toto p){
  toto noName = {“NoName”, p.age};
  return noName;
}
```

Code équivalent utilisant typedef

Struct en C : Padding

- Les données sont alignées sur une case mémoire dont l'adresse est un multiple de leur taille
- Ceci pour des raisons d'efficacité
- Le padding peut être désactivé

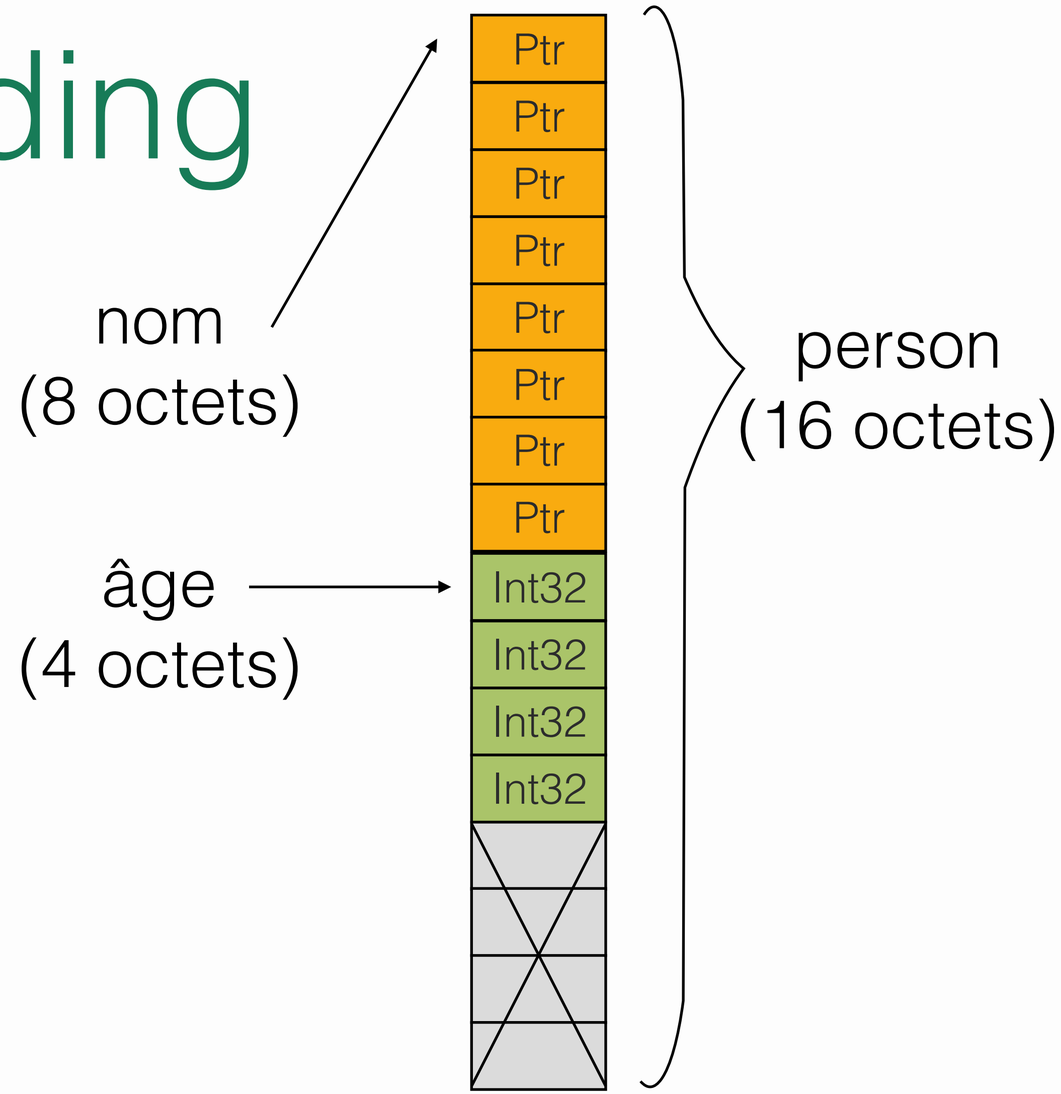
- int : aligné sur un multiple de 4
- char : aligné sur un multiple de 1
- char * : aligné sur un multiple de 8

Sur x86-64

Struct en C : Padding

```
struct person {
    char* nom;
    int age;
};
```

Code C : struct person

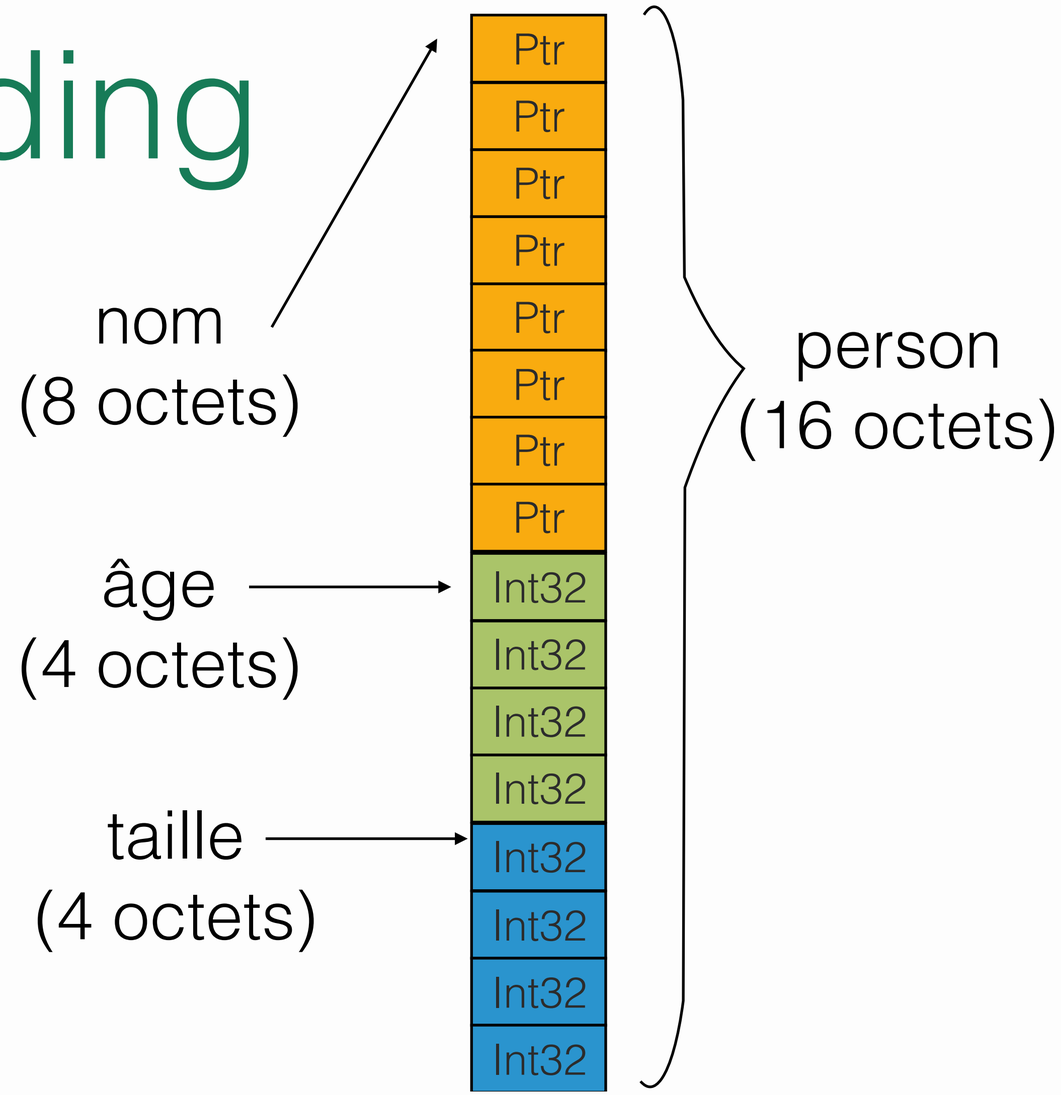


person sur x86-64. Chaque case correspond à un octet.

Struct en C : Padding

```
struct person {
    char* nom;
    int age;
    int taille;
};
```

Code C : struct person

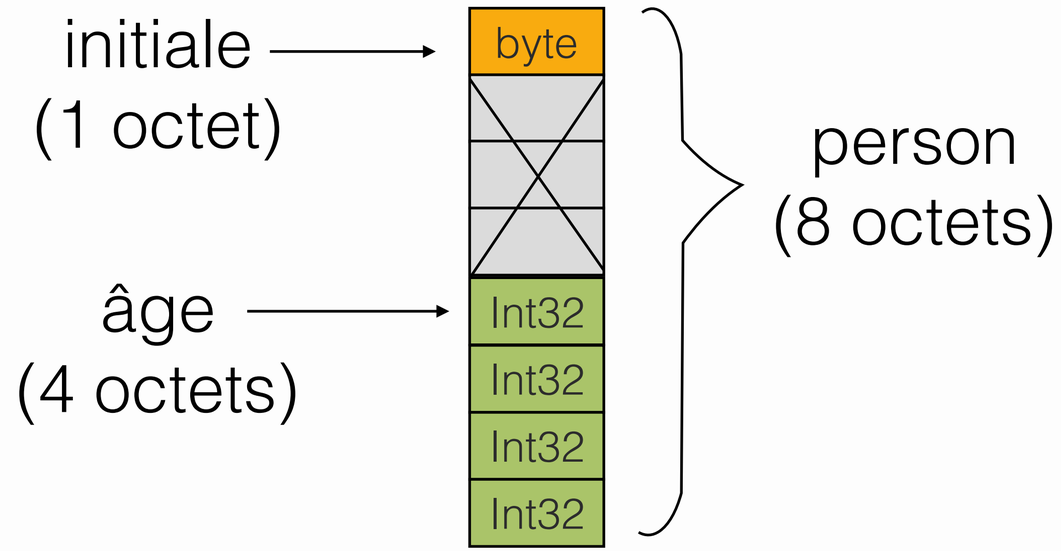


person sur x86-64. Chaque case correspond à un octet.

Struct en C : Padding

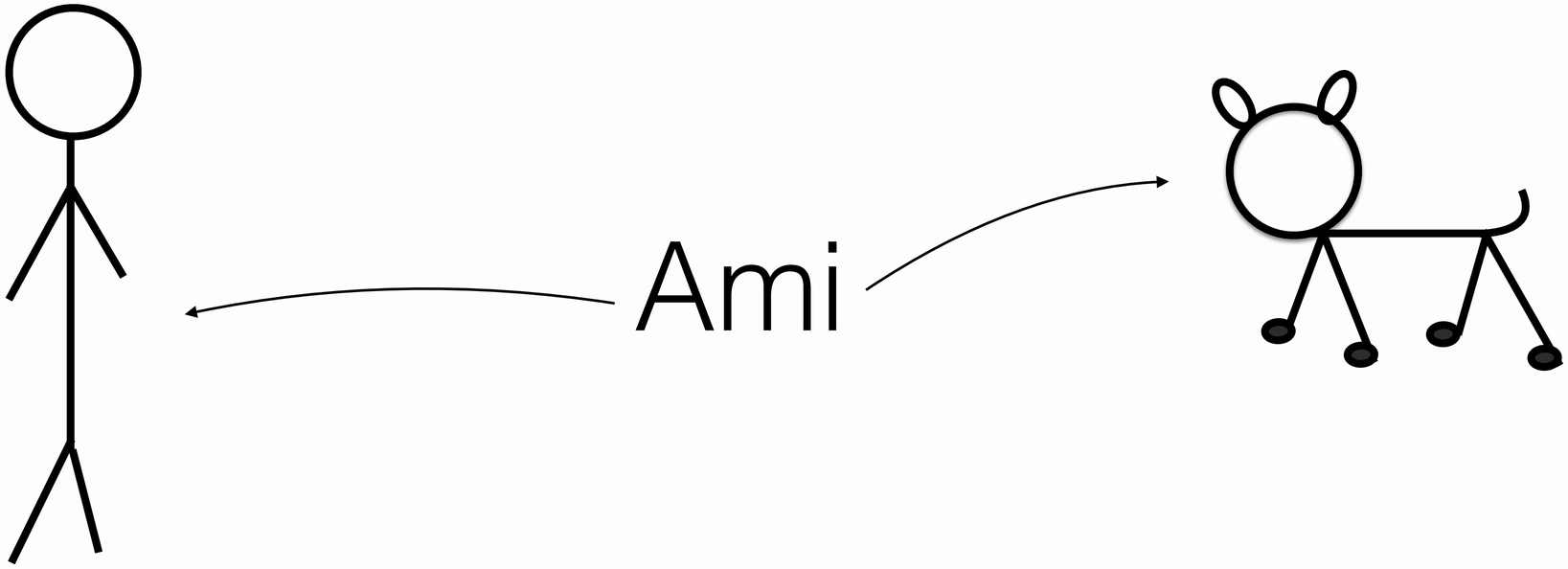
```
struct person {
   char initiale;
   int age;
};
```

Code C : struct person



person sur x86-64. Chaque case correspond à un octet.

Ami = Chien ou Personne



Union en C

```c
typedef struct person person;
struct person {
   char* nom;
   int age;
};

typedef struct chien chien;
struct chien {
  int age;
  int nbPuces;
};
```

Code C pour définir person et chien

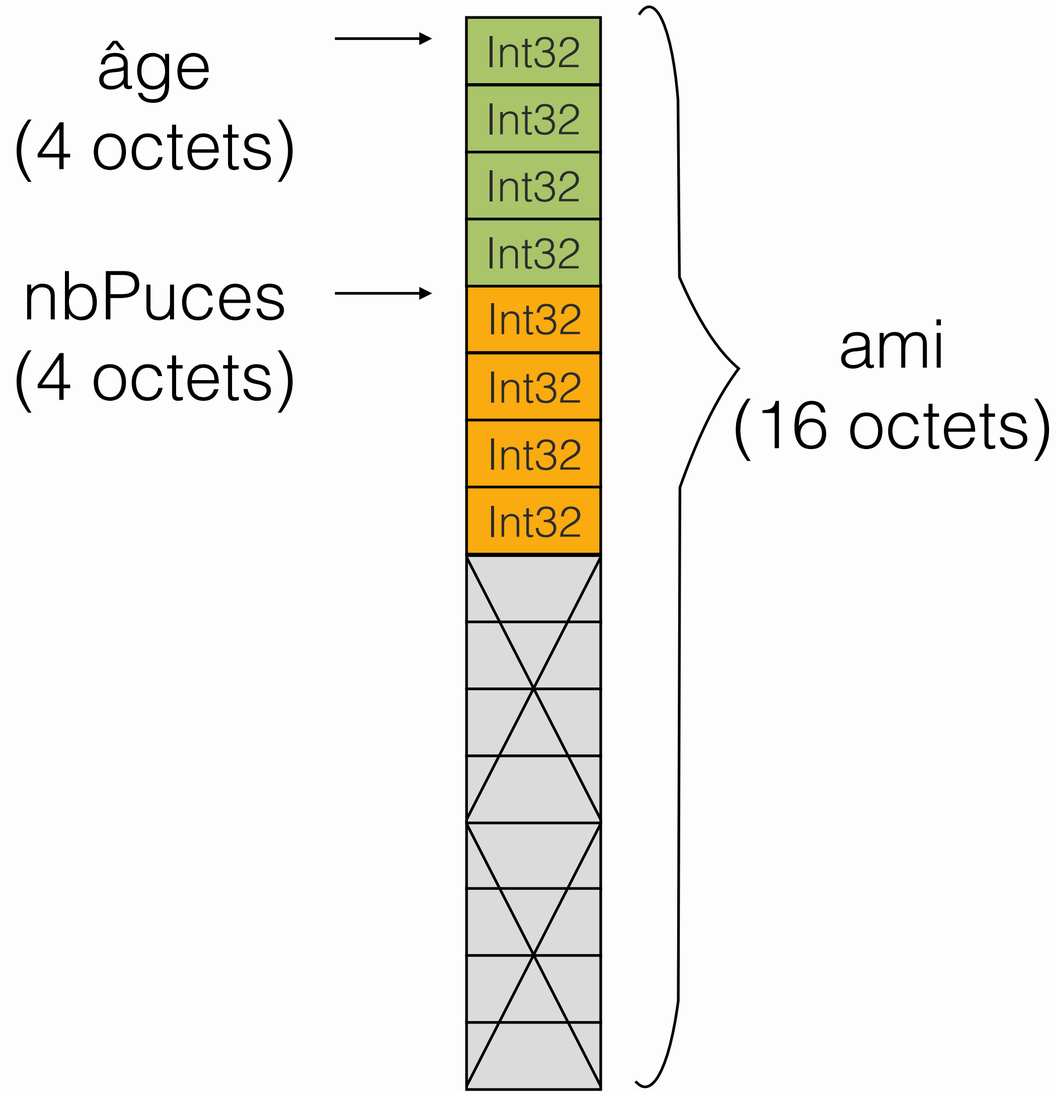
```c
typedef union ami ami;
union ami{
  person humain;
  chien  canin;
};
```

Code C pour définir ami

Union en C

```
person paul = {"Paul", 23};
chien wouf = {2, 245};
ami a;
a.canin = wouf;
```

Code C : utilisation de l'union ami

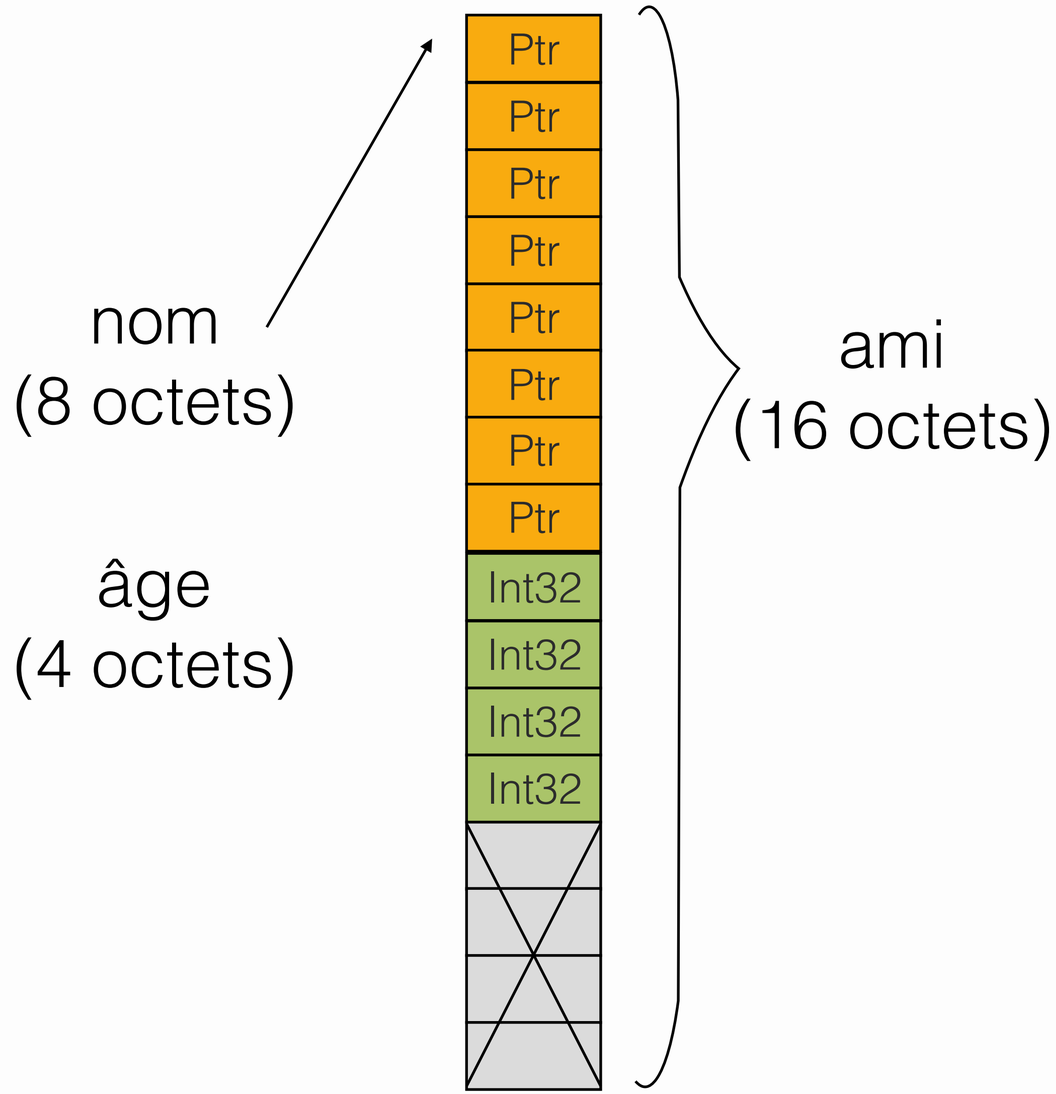


ami sur x86-64. Chaque case correspond à un octet.

Union en C

```
person paul = {"Paul", 23};
chien wouf = {2, 245};
ami a;
a.humain = paul;
```

Code C : utilisation de l'union ami



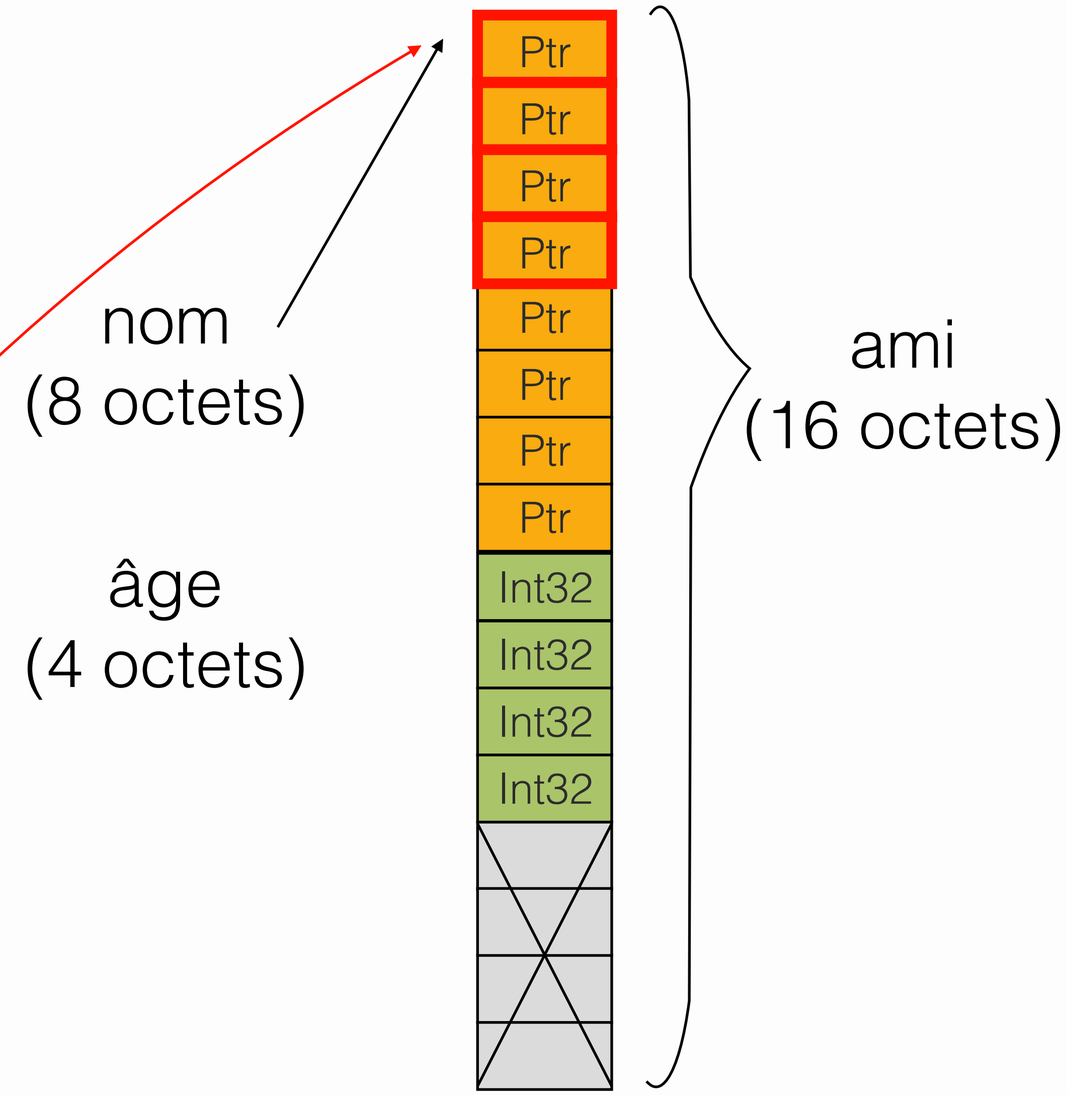
person sur x86-64. Chaque case correspond à un octet.

Union en C

```
person paul = {"Paul", 23};
chien wouf = {2, 245};
ami a;
a.humain = paul;

printf("age : %i\n", a.canin.age);
// age : 29101926
```

Code C : mauvaise utilisation de l'union ami



person sur x86-64. Chaque case correspond à un octet.

Union en C

```
void foo(ami a){
  // a est une personne ou un chien ?
};
```

Union en C

```
typedef struct person person;
struct person {
   char* nom;
   int age;
};

typedef struct chien chien;
struct chien {
  int age;
  int nbPuces;
};
```

Code C pour définir person et chien

```
typedef struct ami2 ami2;
struct ami2{
  enum {HUMAIN, CANIN} tag;
  union ami ami;
};
```

Code C pour définir ami avec un tag

Union en C

```
typedef struct person person;
struct person {
   char* nom;
   int age;
};

typedef struct chien chien;
struct chien {
  int age;
  int nbPuces;
};
```

Code C pour définir person et chien

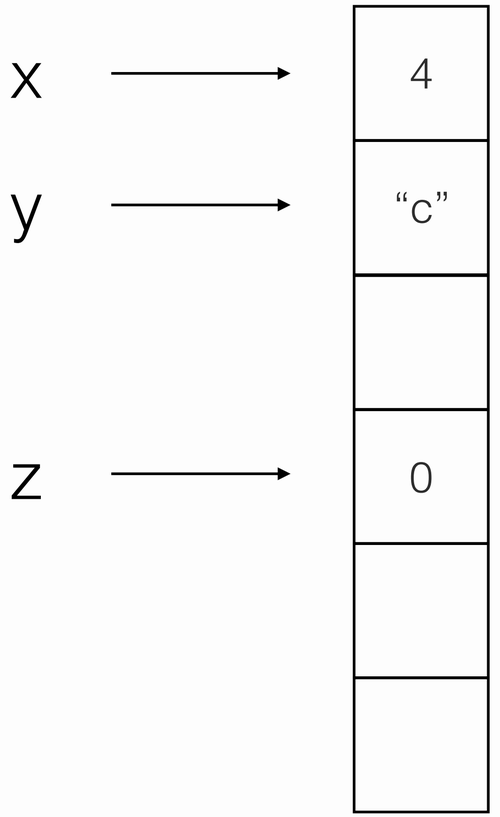
```
typedef struct ami3 ami3;
struct ami3{
  enum {HUMAIN, CANIN} tag;
  union {
  person humain;
  chien  canin;
  } ami;
};
```

Code C pour définir ami avec un tag

Les mêmes concepts, mais sans modèle de la mémoire

Haskell: variable pour 1 espace mémoire

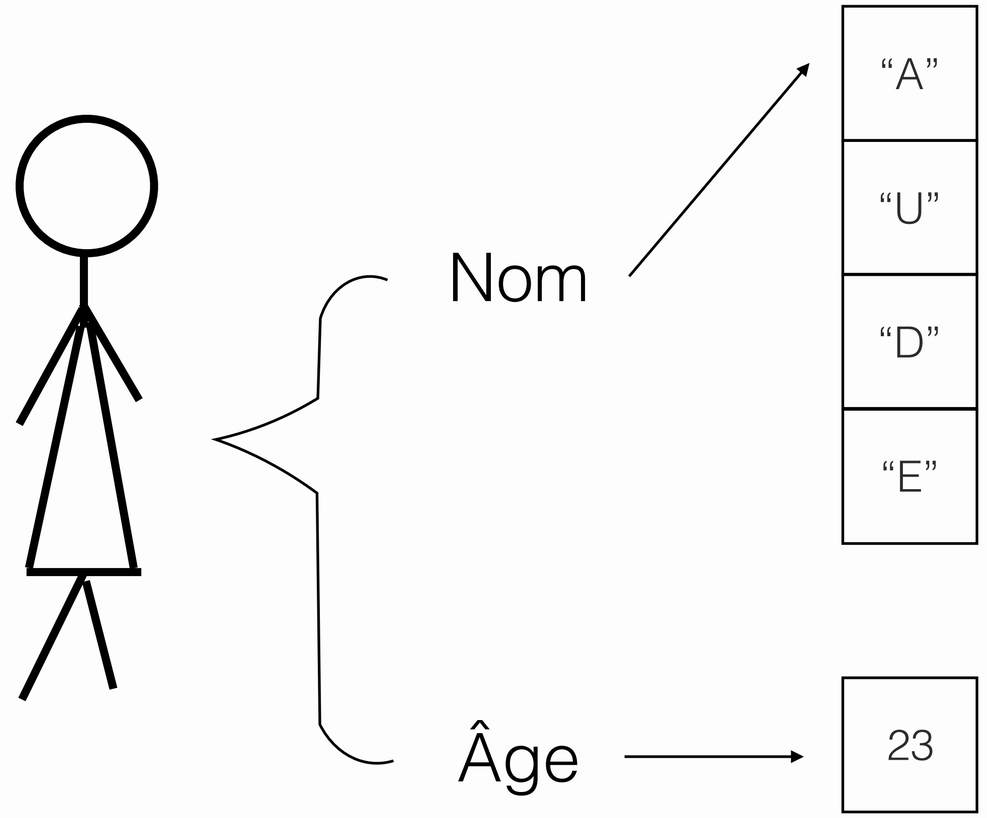
```
x = 4
y = "c"
z = 0
```



Haskell: variable pour person

```
data Person = Person String Int

x = Person "Aude" 23
```



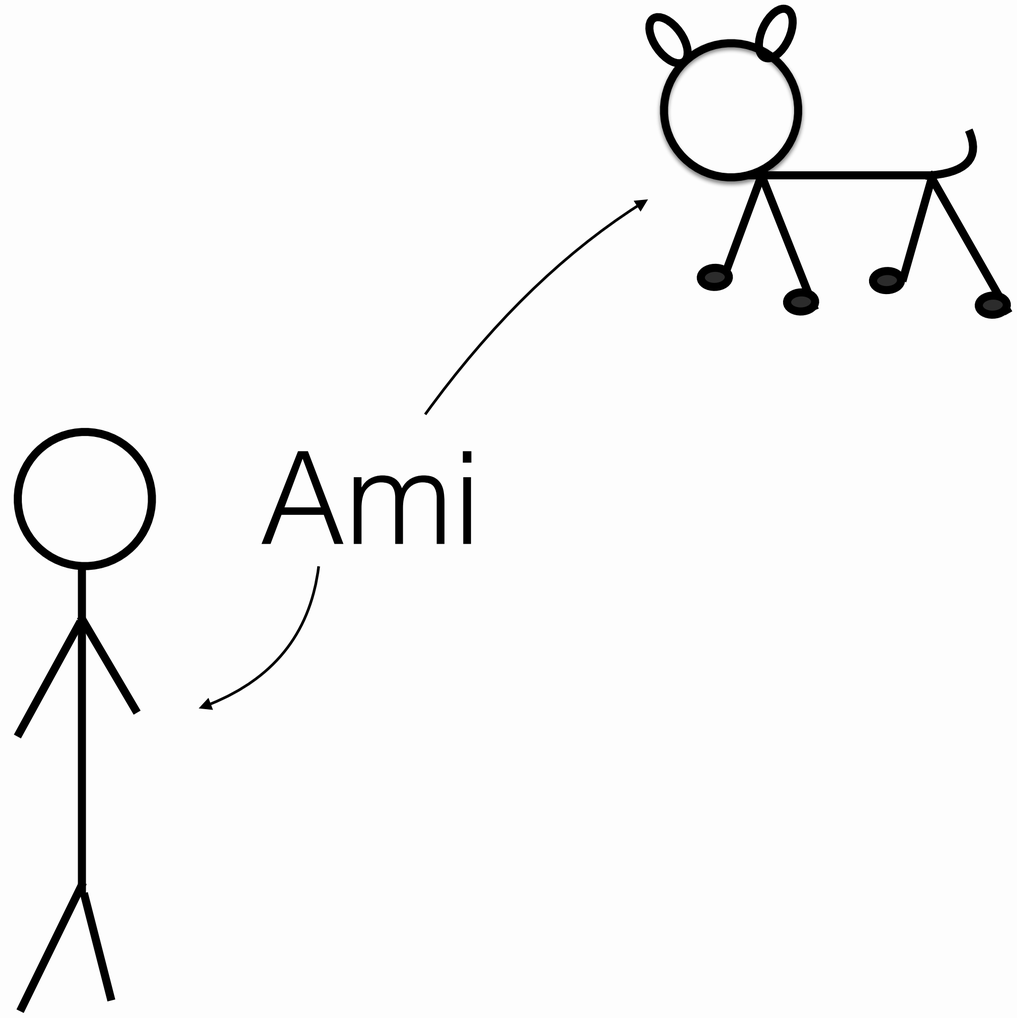
Haskell, définition de Person

Haskell: variable pour ami

```
data Ami = Person String Int
         | Chien Int Int

x :: Ami
x = Person "Aude" 23

y :: Ami
y = Chien 2 245
```

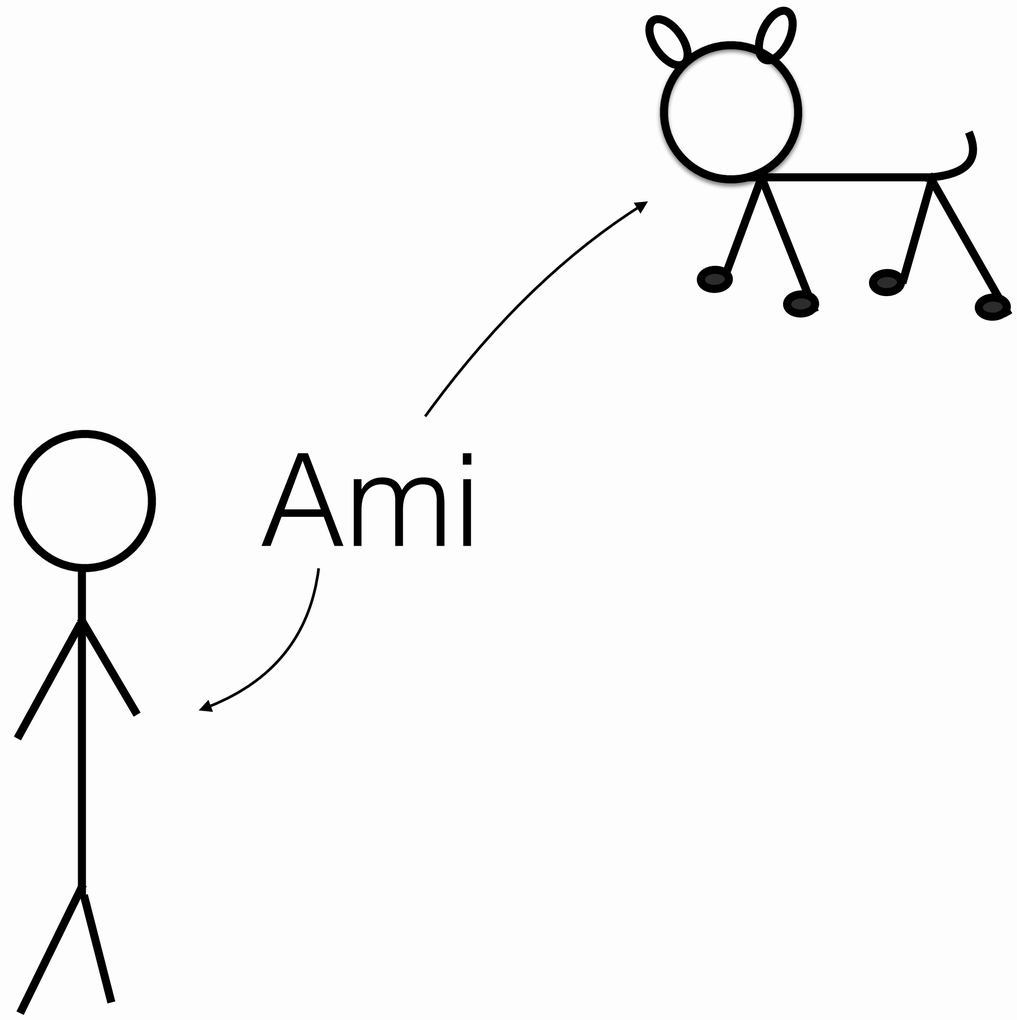


Haskell, définition de Ami

Haskell: variable pour ami

```
data Ami = Person String Int
         | Chien Int Int

age :: Ami -> Int
age x = case x of
         Person _ a -> a
         Chien a _  -> a
```



Haskell, utilisation du case au lieu d’un tag

Haskell ne permet pas de sous-typage

```c
typedef struct chien chien;
struct chien {
  int age;
  int nbPuces;
};

typedef union ami ami;
union ami{
  person humain;
  chien  canin;
};
chien wouf = {2, 245};

ami a;
a.canin = wouf;

chien c = a.canin;
```

C: chien dans ami peut devenir une struct chien

```haskell
data Chien = Chien Int Int

data Ami = Person String Int
         | Chien Int Int


c :: ???
c = Chien 2 245
```

Haskell : Erreur, c serait de type Ami et Chien

Haskell ne permet pas de sous-typage

```c
typedef struct chien chien;
struct chien {
  int age;
  int nbPuces;
};

typedef union ami ami;
union ami{
  person humain;
  chien  canin;
};
chien wouf = {2, 245};

ami a;
a.canin = wouf;

chien c = a.canin;
```

C : Chien dans l'union ami = struct chien

```haskell
data Chien = Chien Int Int

data Ami = Person String Int
         | Chien2 Int Int


c :: Chien
c = Chien 2 245

c2 :: Ami
c2 = Chien2 2 245
```

Haskell : Deux valeurs avec des types différences

Haskell ne permet pas de sous-typage

```c
typedef struct chien chien;
struct chien {
  int age;
  int nbPuces;
};

typedef union ami ami;
union ami{
  person humain;
  chien  canin;
};
chien wouf = {2, 245};

ami a;
a.canin = wouf;

chien c = a.canin;
```

C : Chien dans l'union ami = struct chien

```haskell
data Chien = Chien Int Int

data Ami = Person String Int
         | Chien2 Int Int


c2 :: Ami
c2 = Chien2 2 245

c :: Chien
c = case c2 of
     Person _ _ -> Chien 0 0 -- Bogus
     Chien2 x y -> Chien x y
```

Haskell : Conversion de chien2 vers chien

Haskell ne permet pas de sous-typage

```c
typedef struct chien chien;
struct chien {
  int age;
  int nbPuces;
};

typedef union ami ami;
union ami{
  person humain;
  chien  canin;
};
chien wouf = {2, 245};

ami a;
a.canin = wouf;

chien c = a.canin;
```

C : Chien dans l'union ami = struct chien

```haskell
data Chien = Chien Int Int

data Ami = Person String Int
         | Chien2 Int Int


c2 :: Chien
c2 = Chien2 2 245

c2 :: Ami
c = case c2 of
     Chien2 x y -> Chien x y
```

Haskell : Conversion de chien2 vers chien