Programmation Procédurale – Variables, Constantes et Types

Polytech'Nice Sophia Antipolis

Erick Gallesio

2015 – 2016

Identificateurs

- Les identificateurs permettent de construire des noms de variables, de constantes ou de types.
- Syntaxe:
  - séquence de lettres ou de chiffres (le premier caractère doit être une lettre)
  - le caractère '_' est considéré comme une lettre
  - La casse est importante

```
ident          Ident              IDENT
foo            x1                 x2
_2012          RacineCarree       racine_carree
MAX            __DATE__
```

Mots réservés

Les mots clés suivants sont réservés et ne peuvent être utilisés comme identificateurs.

`auto`	`double`	`int`	`switch`
`_Bool`	`else`	`long`	`typedef`
`break`	`enum`	`register`	`union`
`case`	`extern`	`restrict`	`unsigned`
`char`	`float`	`return`	`void`
`_Complex`	`for`	`short`	`volatile`
`const`	`goto`	`signed`	`while`
`continue`	`if`	`sizeof`	
`default`	`_Imaginary`	`static`	
`do`	`inline`	`struct`	

Mots clés C ANSI: `const enum signed void volatile`

Mots clés C99: `_Bool _Complex _Imaginary restrict`

Types simples (1 / 3)

- Plusieurs types d'entiers

type	déclaration C
caractère	`char`
short integer	`short int` *ou* `short`
integer	`int`
long integer	`long int` *ou* `long`
long long integer	`long long int`

- Chaque type d'entier peut être signé ou non:

```
[unsigned | signed] char
[unsigned | signed] [long] [short | long] [int]
```

Types simples (2 / 3)

Types entiers

- constantes caractères

```
'a'   '\012'    '\xff'
'\''  '\\'
'\n'          newline
'\t'          horizontal tab
'\v'          vertical tab
'\b'          backspace
'\r'          carriage return
'\f'          form feed
'\a'          audible bell
```

- constantes entières

```
100 (décimal)     0100 (octal)     0x100 (hexadécimal)
100UL (décimal et unsigned long)   100L (décimal et long)
```

Types simples (3 / 3)

- Nombre réels (`float` *ou* `double`)
  3.1415927 1.23e45
- Le type `void`
  - aucun objet ne peut être de ce type
  - utilisé pour spécifier qu'un fontion n'a pas de résultat (procédure)
  - utile aussi avec les pointeurs
- Le type booléen `_Bool` (apport C99)
  - souvent utilisé avec le fichier `<stdbool.h>`
  - `bool` est équivalent à `_Bool`
  - `false` est équivalent à `0`
  - `true` est équivalent à `1`
  - On peut se passer de ce type
    - 0 est faux
    - toute autre valeur est vraie

Déclarations de variables (1 / 2)

```
/* déclarations simples */
int x;
int a, b, c;

/* déclaration et initialisation */
unsigned int v1 = 0xabcd;
unsigned long int v2, v3 = 1234UL;
float v4 = 123.45,
      v5 = 0.0;

/* déclarations de constantes */
const int size = 100;
const double Pi = 3.14159;
```

Déclarations de variables (2 / 2)

Les tailles des types simples ne sont pas définies par la norme C. Toutefois:

- `sizeof(char) == 1` par définition
- `sizeof(short)` occupe au moins 16 bits
- sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)
- Exemples
  - 8, 16, 16, 32, ?? (Dos / 16 bits)
  - 8, 16, 32, 32, 64 (Linux 32 bits)
  - 8, 16, 32, 64, 64 (Linux 64 bits)

Type tableau (1 / 2)

Principales caractéristiques

- Une seule dimension (mais on peut faire des tableaux de tableaux)
- Indice entier
- Borne inférieure à l'indice 0
- Initialisation possible avec des agrégats
- Examples

```
int t[10];
        ==> t[0] t[1] ... t[9]

short int t[3][10];
        ==> t[0][0] t[0][1] ... t[0][9]
            t[1][0] t[1][1] ... t[1][9]
            t[2][0] t[2][1] ... t[2][9]
```

Type tableau (2 / 2)

- On peut déclarer **et** initialiser un tableau

```
int t[4] = {1, 2, 3, 4};      /* [1, 2, 3, 4] */
int t[4] = {1, 2};            /* [1, 2, 0, 0] */
```

- La dimension peut être calculée par le compilateur

```
int t[] = {1, 2, 3, 4};      /* dimension = 4 */

short int t[][3] = {
            { 0, 1, 2 },
            { 3, 4},
            { 5}
};                          // =>  0 1 2
                            //     3 4 0
                            //     5 0 0
```

Attention: Seule la première dimension peut être omise.

Exemple: Tableaux

```c
#include <stdio.h>

int main() {
  int c, i, spaces, others;
  int digits[10];

  spaces = 0; others = 0;
  for (i=0; i<10; i++) digits[i] = 0;

  while ((c = getchar()) != EOF) {
     if (c >= '0' && c <= '9') digits[c-'0'] += 1;
     else
       if (c == '\t' || c == '\n' || c == ' ')
         spaces += 1;
       else
         others += 1;
  }

  for (i=0; i<10; i++) printf("%c: %d\n", '0'+i, digits[i]);
  printf("spaces: %d\nothers: %d\n", spaces, others);

  return 0;
}
```

Chaînes de caractères (1 / 2)

Principales caractéristiques

- tableaux de caractères
- se terminent par le caractère nul (caractère '\0')
- n'est pas un type C à proprement parlé
- possibilité d'avoir des chaînes littérales avec des guillemets
- **Exemples**

```
char string1[100], string2[10];

"I'm a string"
"Another string with embedded \"quotes\""
"and another one with a \"\\\" !!!"
```

Chaînes de caractères (2 / 2)

Notes

- Le caractère nul n'apparaît pas dans une chaîne littérale (automatiquement mis par le compilateur)
- Pas d'opérateur pré-défini sur les chaînes (concaténation, sous-chaîne, . . .)
- Nombreuses fonctions dans la bibliothèque C standard
- *Attention*: ne pas oublier de réserver un caractère supplémentaire pour mettre le caractère nul.

```
char ch[10];      /* permet de stocker des chaînes
                     de longueur 0 -> 9 */
```

Exemple: Chaînes de caractères

```c
int strlen(char s[])
{
  int i = 0;

  while (s[i] != '\0') i += 1;
  return i;
}
```

```c
void strcat(char s1[], char s2[]) /* le strcat standard n'est pas void... */
{
  int i=0, j=0;

  while (s1[i] != '\0') i += 1;     /* parcours de s1 */

  while (s2[j] != '\0') {           /* parcours de s2 + copie */
    s1[i] = s2[j];
    i += 1; j += 1;
  }

  /* Ne pas oublier le caractère nul final */
  s1[i] = '\0';
}
```

Types énumérés (1 / 2)

- ils permettent de nommer des constantes

```
enum traffic_lights {green, orange, red} light1, light2;
enum traffic_lights light3, light4 = green;
enum traffic_lights town[1000];
```

- l'énumeration peut être aussi *anonyme*

```
enum {green, orange, red} light1, light2;
```

- Le compilateur affecte automatiquement des valeurs croissantes

```
green  = 0
orange = 1
red    = 2
```

Types énumérés (2 / 2)

- Les valeurs de constantes peuvent être spécifiées par l'utilisateur

```
enum start_address {mono = 0xb000, color = 0xb800};
enum escapes {bell    = '\a', backspace ='\b', tab    ='\t',
              newline = '\n', vtab      ='\v', return ='\r'};
```

- Pas de contrôle sur les valeurs utilisées (`'light1 = 20;'` est correct)
- On peut avoir des valeurs d'énumération identiques

```
enum color {blue=0, red =1, gray=2, grey=2};        /* OK */
```

- **MAIS** les constantes doivent être différentes entre les énumérations

```
enum traffic_lights {green, orange, red};
enum fruit {apple, banana, orange};                 /* Erreur */
```

- **Assez peu utilisé en fait**

Structures (1 / 2)

Une structure C:

- objet composite constitué d'éléments qui peuvent être de types différents
- peut être manipulée comme un tout

```
struct {                              /* structure anonyme */
    short int day, month;
    int year;
} date1, date2;


/* Accès par champs */
date1.day = 25;
date1.month = 12;

/* Affectation de structures  */
date1 = date2;
```

Structures (2 / 2)

```
struct person {                    /* structure nommée */
  char Name[30];
  struct date birth_date;
};

struct person me, employees[100];
date1 = me.birth_date;
employees[i]. birth_date.day=12;
c = employees[0].Name[0];
```

Les structures peuvent être initialisées avec des agrégats:

```
struct coords {double x,y;} p = {0.0, 1.2};
struct coords rectangle[2]    = { {0.0, 0.0},
                                  {1.0, 4.0}};
```

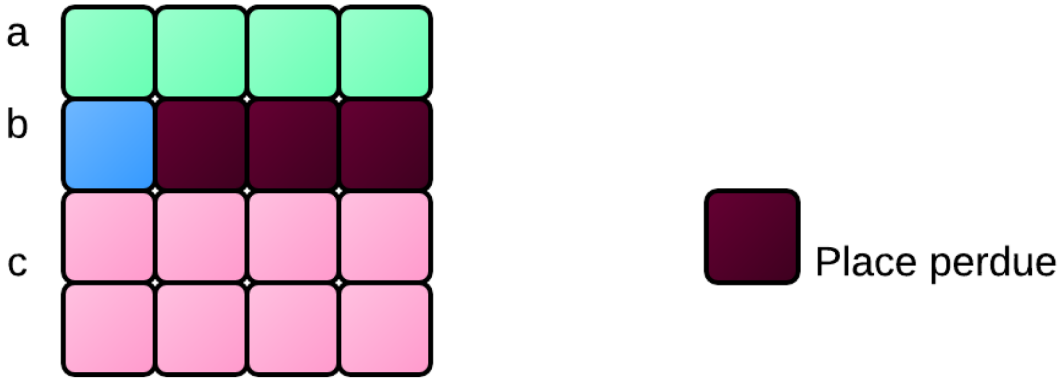
Unions (1 / 3)

Semblables aux structures mais où un seul champ n'est valide à un instant donné.

Utile pour:

- partager de la mémoire entre des objets qui sont accédés exclusivement
- interpréter la représentation interne d'un objet comme s'il était d'un autre type

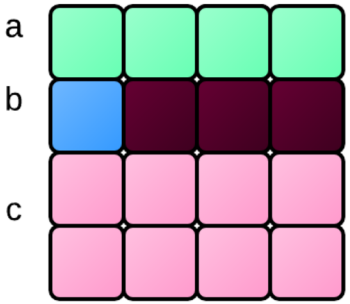
```
struct {
  int  a;
  char b;
  double c;
}
```



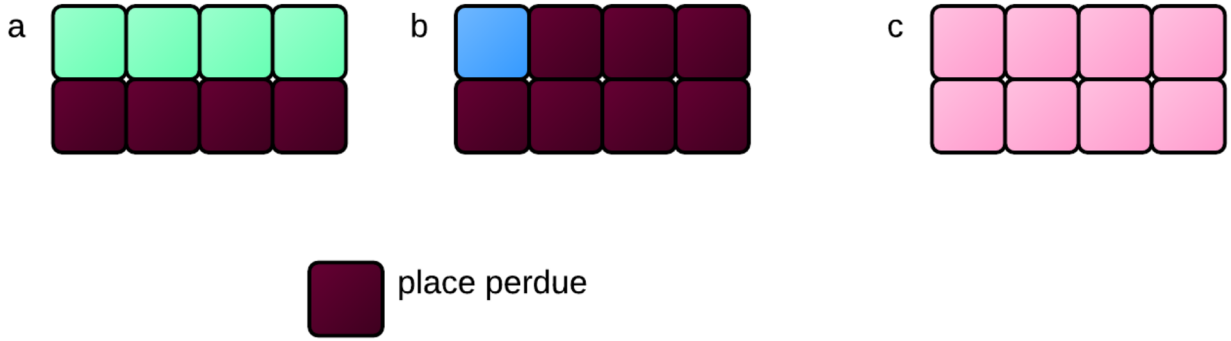
Occupation mémoire (Linux x86)

Unions (2 / 3)

```
struct {
   int a;
   char b;
   double c;
}
```




```
union {
   int a;
   char b;
   double c;
}
```



Occupation mémoire (Linux x86)

.

Unions (3 / 3)

```
struct person {
  char name[30], first_name[20];
  struct date birth_date;
  enum {female, male} gender;
  union {
     char maiden_name[30];
     enum boolean {false, true} national_service;
  } info;
};

struct person bob  = {"Smith", "Bob", {1,1,1940}, male, false},
              mary = {"Smith", "Mary", {2,2,1940}, female, "Brown"};

i = Mary.birth_date.day;
bob.birth_date = mary.birth_date;
bob.info. national_service = true;
```

MAIS l'accès à `bob.info.maiden_name` est permis (résultat indéfini)

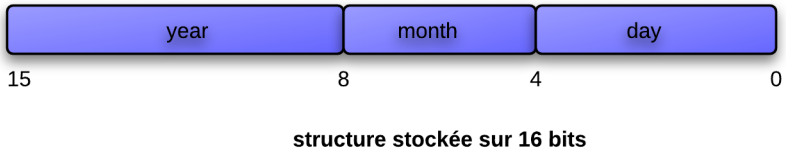
Champs de bits (1 / 2)

Les champs de bits peuvent être utilisés pour des accès bas niveau.

C'est utile:

- quand la mémoire est contrainte
- pour l'accès aux périphériques au niveau le plus bas.

```
struct Date {
   unsigned int day  : 5;
   unsigned int month: 4;
   unsigned int year : 7;
};
```



structure stockée sur 16 bits

.

```
struct Date d1 = {31, 1, 12};
struct Date d2;

d2.day = 1; d2.month = 1;
d2 = d1;
```

Champs de bits (2 / 2)

```
union date {
  struct {
     unsigned int day  : 5;
     unsigned int month: 4;
     unsigned int year : 7;
  } d;
  short n;
};

union date d1, d2;

/* utilisation comme une structure */
d1.d.day = ...
d2.d.day = ...

/*utilisation comme un entier */
if (d1.n < d2.n)
  ....
```

Complexes

- Les complexes sont un apport de C99
- On peut avoir des `_Complex float` ou des `_Complex double` ...
- Le header `<complex.h>` définit `complex`

```c
#include <stdio.h>
#include <complex.h>

int main()
{
  complex float a = 3+2i, b = 4+5i, c;

  c = a + b;
  printf("la somme de a et b = %f + %fi\n", creaf(c), cimagf(c));

  return 0;
}
```

A l'exécution: `la somme de a et b = 7.000000 + 7.000000i`

Définition de types (typedef)

- Possibilité de donner un nom à un type.
- permet de simplifier l'écriture
- utilisation du mot-clé **typedef**
- *similitude* avec une déclaration de variable

```
typedef int Integer;       /* Integer et int sont synonymes */
typedef int Table[100];    /* Table est un tableau de 100 entiers */
typedef struct {
    int x, y;
} Position;                /* Position est une struct. de 2 entiers */
typedef enum {false, true} Boolean;  /* ~ _Bool */

Integer i, j, k;
Table my_table;
Position Origin = {0, 0};
Boolean b = true;

int z;
z = i;     /* est autorisé (typedef est une aide pas vraiment un type) */
```