

# Objectif 02

## Éléments de base du langage C

Aziz Salah  
salah.aziz@uqam.ca

Département d'informatique  
UQÀM

Automne 2013

- 1 Le identificateurs
- 2 Déclaration d'une variable
- 3 Les types entiers
- 4 Le type des caractères
- 5 Les chaines de caractères
- 6 Les types flottants
- 7 Le type énuméré
- 8 Le type booléen
- 9 Les expressions
- 10 Les instructions
- 11 Entrée/sortie formatées de base
- 12 Instructions de contrôle
- 13 Initiation aux tableaux
- 14 Initiation aux fonctions
- 15 Conclusions

- 1 **Le identificateurs**
- 2 Déclaration d'une variable
- 3 Les types entiers
- 4 Le type des caractères
- 5 Les chaines de caractères
- 6 Les types flottants
- 7 Le type énuméré
- 8 Le type booléen
- 9 Les expressions
- 10 Les instructions
- 11 Entrée/sortie formatées de base
- 12 Instructions de contrôle
- 13 Initiation aux tableaux
- 14 Initiation aux fonctions
- 15 Conclusions

- Les identificateurs sont utilisés pour donner des noms à certaines entités d'un programme comme les variables, les fonctions, les macros du préprocesseur, ...
- Un identificateur est mot qui commence par une lettre ou par '\_' et pouvant contenir des chiffres
- C99 considère les premiers 63 caractères pour les identificateurs internes et seulement jusqu'à 31 caractères pour les externes
- Les majuscules sont distinctes des minuscules dans le langage C
- C99 permet des identificateurs avec des caractères universelles mais ceci n'est pas encore supporté par `gcc`. Donc à éviter pour avoir un code portable
- Un identificateur ne doit pas être un mot clé du Langage C

## Liste des mots clés dans C99

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

## Choisir un identificateur qui révèle son rôle

Facilite la lisibilité et augmente la documentation du code source

## Exemples

- canadiens, Canadiens, CanAdiens , CANADIENS et \_CANADIENS sont des identificateurs valides et tous différents
- De même While et while0
- prix\_unitair, num\_carte\_credit (**pas d'accent**)

- 1 Le identificateurs
- 2 Déclaration d'une variable**
- 3 Les types entiers
- 4 Le type des caractères
- 5 Les chaines de caractères
- 6 Les types flottants
- 7 Le type énuméré
- 8 Le type booléen

- 9 Les expressions
- 10 Les instructions
- 11 Entrée/sortie formatées de base
- 12 Instructions de contrôle
- 13 Initiation aux tableaux
- 14 Initiation aux fonctions
- 15 Conclusions

# Déclaration d'une variable (cas simple)

La déclaration d'une variable consiste à lui donner un nom et un type

- Le nom de la variable est identificateur
- Le type est l'un des types définis dans le langage C, par exemple
  - int
  - char ...

## Déclaration sans initialisation

```
<type> <identificateur>;
```

## Déclaration avec initialisation

```
<type> <identificateur> = <expression>;
```

## Exemple

```
int nombre_produit; //valeur initiale inconnue  
char choix = 'K';
```



- 1 Le identificateurs
- 2 Déclaration d'une variable
- 3 Les types entiers**
- 4 Le type des caractères
- 5 Les chaines de caractères
- 6 Les types flottants
- 7 Le type énuméré
- 8 Le type booléen
- 9 Les expressions
- 10 Les instructions
- 11 Entrée/sortie formatées de base
- 12 Instructions de contrôle
- 13 Initiation aux tableaux
- 14 Initiation aux fonctions
- 15 Conclusions

## Exigence de précision **minimale** selon C99

- `short` ou `short int` (`SHRT_MIN` : -32767) (`SHRT_MAX` : 32767) ( $2^{15} - 1$ ) (2 octets)
- `unsigned short` ou `unsigned short int` (`USHRT_MAX` : 65535) ( $2^{16} - 1$ ) (2 octets)
- `int` (`INT_MIN` : -32767) (`INT_MAX` : 32767) ( $2^{15} - 1$ ) (2 octets)
- `unsigned` ou `unsigned int` (`UINT_MAX` : 65535) ( $2^{16} - 1$ ) (2 octets)
- `long` (`LONG_MIN` : -2147483647) (`LONG_MAX` : 2147483647) ( $2^{31} - 1$ ) (4 octets)
- `unsigned long` (`ULONG_MAX` : 4294967295) ( $2^{32} - 1$ ) (4 octets)
- `long long` (`LLONG_MIN` : -9223372036854775807) (`LLONG_MAX` : 9223372036854775807) ( $2^{63} - 1$ ) (8 octets)
- `unsigned long long` (`ULLONG_MAX` : 18446744073709551615) ( $2^{64} - 1$ ) (8 octets)

- Les constantes de suffixes `_MAX` et `_MIN` sont définies dans `<limits.h>` et leurs valeurs dépendent de la plate-forme
- Pour des raisons de portabilité, C99 définit les entiers étendus<sup>a</sup> à travers `<stdint.h>` : `intN_t` et `uintN_t` avec  $N = 8, 16, 32, 64$  et `MAX`
- `intN_t` est un type d'entiers de  $N$  bits quelque soit la plate-forme (`int32_t` est un entier avec signe sur 32 bits)
- `<stdint.h>` définit aussi `int_leastN_t` garantissant au moins  $N$  bits et `int_fastN_t` pour les entiers les plus rapides sur  $N$  bits avec  $N = 8, 16, 32, 64$

---

a. La norme les appelle entiers étendus mais veut dire de taille fixée

## Définitions

- C99 considère qu'une constante entière sans suffixe qu'elle est de type `int`, `long` ou `long long`
- Suffixes
  - `L` ou `l` (risque confusion avec `1` (un)) : `long`
  - `UL` ou `ul` : `unsigned long`
  - `LL` ou `ll` : `long long`
  - `ULL` ou `ull` : `unsigned long long`
- Préfixes
  - `0` (zéro) : représentation en base octale (8)
  - `0x` ou `0X` : représentation en base hexadécimale (16)

## Exemples

- 137 est de type `int`, `long` ou `long long`
- 6149<sub>L</sub> est de type `long`
- 30498<sub>ULL</sub> est de type `unsigned long long`
- 015 est la représentation en base octale de l'entier  $1 * 8^1 + 5 * 8^0 = 13$  en décimal
- 0x26 est la représentation en base hexadécimale de l'entier  $2 * 16^1 + 6 * 16^0 = 38$  en décimal

## Macros d'instanciation des types d'entiers étendus

- Assurent la portabilité
- Pour `intN_t` et `uintN_t` avec `N = 8, 16, 32, 64` et `MAX` on utilise les macros `INTN_C()` `UINTN_C()`
- De même pour `int_leastN_t`, `int_fastN_t` on a `INT_LEASTN_C()` et `INT_FASTN_C()`, etc

## Exemples

- `INTMAX_C(-3135)` donne lieu à une expression de type `intmax_t` dont la valeur est `-3135`
- `INT_LEAST32_C(117)` donne lieu à une expression de type `int_least32_t` dont la valeur est `117`
- `UINT_LEAST32_C(117)` donne lieu à une expression de type `uint_least32_t` dont la valeur est `117`

# Plan

- 1 Le identificateurs
- 2 Déclaration d'une variable
- 3 Les types entiers
- 4 Le type des caractères**
- 5 Les chaines de caractères
- 6 Les types flottants
- 7 Le type énuméré
- 8 Le type booléen
- 9 Les expressions
- 10 Les instructions
- 11 Entrée/sortie formatées de base
- 12 Instructions de contrôle
- 13 Initiation aux tableaux
- 14 Initiation aux fonctions
- 15 Conclusions

# Un petit aperçu sur le codage des caractères

- Un caractère est un symbole qui peut être soit une lettre (majuscule ou minuscule), soit un chiffre, soit un caractère graphique comme ceux de la ponctuation ou un caractère blanc comme le caractère de fin de ligne, espace, tabulation . . .
- Ici on parle des caractères qui sont des données du programme et non pas ceux utilisé pour les identificateurs
- Un caractère est codé en mémoire par un entier selon une table de correspondance
- La table `ascii` est la plus utilisée (comportant 128 caractères codés sur 7 bits)
- La norme `iso latin 1` (ISO 8859-1) étend la table `ascii` en y introduisant les caractères accentués et autres des langues européennes pour les coder sur 8 bits
- Les normes Unicode et ISO10646 unifiée permettent coder tous les caractères du monde en utilisant 16 bits ou 32 bits



## ■ Type `char` :

- Il est selon la norme ISO646 qui coïncide avec le code `ascii` pour la majorité de ses caractères
- En pratique, ce type codés selon le code des caractères adopté par la plateforme d'implémentation (en général `iso latin 1` (8bits) qui est un superset du code `ascii`)
- La taille du type `char` pour une plate-forme est donnée en bits par la macro `CHAR_BIT` définie dans `<limits.h>`

## ■ Type `wchar_t` (*wide char* ou caractère large)

- implémente les normes Unicode et ISO10646 nécessitant 32 bits
- utilise une librairie spécifique pour la manipulation des chaînes de caractères

## ■ Le type `char` a deux facettes

- C'est un entier codé sur `CHAR_BIT` bits et peut être interprété avec signe ou sans signe
- C'est le caractère dont le code est représenté sur `CHAR_BIT` bits

## ■ Le langage C définit

- `signed char`
- `unsigned char`

## ■ La norme laisse le choix à l'implémentation de considérer le type `char` comme étant un `signed char` ou bien `unsigned char`

# Exigence minimale sur les types `char`

| <i>Macro</i>           | <i>Exigence minimale</i>                 | <i>Valeur</i>          |
|------------------------|--|------------------------|
| <code>CHAR_BIT</code>  | Min de nombre de bit dans un octet       | 8                      |
| <code>SCHAR_MIN</code> | valeur min de <code>signed char</code>   | -127                   |
| <code>SCHAR_MAX</code> | valeur max de <code>signed char</code>   | 127                    |
| <code>UCHAR_MAX</code> | valeur max de <code>unsigned char</code> | 255                    |
| <code>CHAR_MIN</code>  | valeur min de <code>char</code>          | <code>SCHAR_MIN</code> |
|                        |  | 0                      |
| <code>CHAR_MAX</code>  | valeur max <code>char</code>             | <code>SCHAR_MAX</code> |
|                        |  | <code>UCHAR_MAX</code> |

Les valeurs adoptées par une implémentation sont définis dans `<limits.h>`

## Par l'exemple

- `'a' 'A' '0' '6' '.' ' '`
- Attention `"a"` est de type chaîne de caractères de même  
`"Il fait 35 degrés ..."`
- `'\x41'` représente le caractère dont le code est 41 en  
hexadécimal
- `'\107'` représente le caractère dont le code est 107 en base  
octale

## Ordre des caractères

- Le code de 'A' est 65, 'B' 66, 'C' 67 ... 'Y' 89 'Z' 90
- 'a' 97 'b' 98 ... 'z' 122
- '0' 48 '1' 49 '2' 50 ...
- utiliser `(c - '0')` pour convertir le char (chiffre) en nombre correspondant
- Les codes consécutifs des caractères A-Z et a-z facilitent la conversion "manuelle" de MAJ vers MIN et inversement. (penser utiliser les fonctions spécifiques `tolower()` et `toupper()`)

## Caractères spéciaux

| Réprésentation | Signification   |
|----------------|---|
| '\n'           | fin de ligne  |
| '\t'           | tabulation horizontale  |
| '\a'           | beep  |
| '\"'           | les deux guillemets "   |
| '\hhh'         | représente le caractère dont le code est hhh en base octale. Les h sont des chiffres entre 0 et 7                                     |
| '\xnn'         | représente le caractère dont le code est nn en base hexadécimal. Les n sont des chiffres entre 0 et 9 en plus de a , b, c, d , e et f |

- 1 Le identificateurs
- 2 Déclaration d'une variable
- 3 Les types entiers
- 4 Le type des caractères
- 5 Les chaines de caractères**
- 6 Les types flottants
- 7 Le type énuméré
- 8 Le type booléen
- 9 Les expressions
- 10 Les instructions
- 11 Entrée/sortie formatées de base
- 12 Instructions de contrôle
- 13 Initiation aux tableaux
- 14 Initiation aux fonctions
- 15 Conclusions

# Un aperçue sur les chaines de caractères

## ■ Déclaration et initialisation d'une chaine de caractères

```
char *chaine = "L'université du Québec"
```

## ■ Affichage

```
printf("Je suis à [%s].",chaine);
```

```
---
```

```
Je suis à [L'université du Québec].
```



- 1 Le identificateurs
- 2 Déclaration d'une variable
- 3 Les types entiers
- 4 Le type des caractères
- 5 Les chaines de caractères
- 6 Les types flottants**
- 7 Le type énuméré
- 8 Le type booléen
- 9 Les expressions
- 10 Les instructions
- 11 Entrée/sortie formatées de base
- 12 Instructions de contrôle
- 13 Initiation aux tableaux
- 14 Initiation aux fonctions
- 15 Conclusions

## Définition

- En programmation, un flottant est une approximation d'un nombre réel
- Le C utilise les normes IEC 60559 ou IEEE 754 définissant les opérations arithmétiques sur les flottants et leurs codages (en mémoire)
- Un flottant est codé par un triplet  $(\text{signe}, m, e)$  qui représente le nombre  $\text{signe } m \cdot 10^e$  où :
  - $\text{signe}$  représente le signe du flottant
  - $m$  s'appelle la mantisse ( $0.1 \leq m < 1$ )
  - $e$  s'appelle l'exposant

## Exemple

Le nombre réel  $2.5 = + 0.25 \cdot 10^1$  est représenté par le flottant  $(+, 0.25, 1)$

- `float`, `double` et `long double` sont les types des flottants en C
- Le C impose des exigences minimales sur les tailles des types flottants
- Chaque plateforme déclare les tailles adoptées pour chaque type flottant dans `<float.h>` sous forme de macros (constantes)
- `<float.h>` définit des macros avec comme préfixe `FLT_` pour `float`, `DBL_` pour `double` et `LDBL_` pour `long double`

# Aperçu des caractéristiques définies dans `<float.h>` avec les préfixes `FLT_`, `DBL_` et `LDBL_`

## Suffixes de macros

- `DIG` : le nombre min de chiffres qui sont précis après la virgule
- `MIN_10_EXP` : l'exposant minimal (négatif)
- `MAX_10_EXP` : l'exposant maximal (positif)
- `MIN` : La valeur minimale positive représentable
- `MAX` : La valeur maximale positive représentable
- `EPSILON` : la plus petite valeur à ajouter à 1 pour obtenir le plus petit nombre représentable plus grand que 1

## Exemple

`DBL_MAX_10_EXP` représente l'exposant maximal pour le type de flottant `double`

# Exigences minimales du langage C

|            | FLT_       | DBL_       | LDBL_      |
|------------|------------|------------|------------|
| DIG        | 6          | 10         | 10         |
| MIN_10_EXP | -37        | -37        | -37        |
| MAX_10_EXP | 37         | 37         | 37         |
| MIN        | $10^{-37}$ | $10^{-37}$ | $10^{-37}$ |
| MAX        | $10^{37}$  | $10^{37}$  | $10^{37}$  |
| EPSILON    | $10^{-5}$  | $10^{-9}$  | $10^{-9}$  |

Ce tableau définit les valeurs min en valeurs absolues de `FLT_DIG`, `DBL_DIG`, `LDBL_DIG`, ...

## À noter

En pratique la majorité des implémentations code `float` sur 4 octets et `double` sur 8

## Les suffixes

- F ou f pour float
- L ou l pour long double
- Sans suffixe, c'est considéré de type double par défaut afin que le C99 préserve une compatibilité rétroactive les autres versions.

## Exemples de valeurs

- 123.45, 1e-3, 1E-3 sont tous de type double (doit contenir au moins le point ou E)
- 123.45F ou 123.45f, -1e-3F sont de type float
- 123.45L ou +1E-3L sont de type long double

- 1 Le identificateurs
- 2 Déclaration d'une variable
- 3 Les types entiers
- 4 Le type des caractères
- 5 Les chaines de caractères
- 6 Les types flottants
- 7 Le type énuméré**
- 8 Le type booléen
- 9 Les expressions
- 10 Les instructions
- 11 Entrée/sortie formatées de base
- 12 Instructions de contrôle
- 13 Initiation aux tableaux
- 14 Initiation aux fonctions
- 15 Conclusions

# Type énuméré I

- On peut définir un nouveau type en énumérant l'ensemble de ses valeurs symboliques

- Syntaxe

```
enum <type_enum>
{
    <idenificateur1> [= <exp_contante_entière>] [, ...]
};
```

- ▶ `enum <type_enum>` : nom du nouveau type
- ▶ `<idenificateur1>` : représente une valeur symbolique du nouveau type
- ▶ `<exp_contante_entière>` : dans l'arrière scène, le C associe une valeur entière à chaque valeur symbolique. On peut décider que cette valeur entière soit la valeur `<exp_contante_entière>`



## Exemple

### ■ Définition du type

```
enum couleur  
{  
    rouge, vert, bleu = 96, noir // 0, 1, 96 et 97  
};
```

### ■ Déclaration d'une variable

```
enum couleur choix = vert;  
choix = rouge;
```

# Plan

- 1 Le identificateurs
- 2 Déclaration d'une variable
- 3 Les types entiers
- 4 Le type des caractères
- 5 Les chaines de caractères
- 6 Les types flottants
- 7 Le type énuméré
- 8 Le type booléen**
- 9 Les expressions
- 10 Les instructions
- 11 Entrée/sortie formatées de base
- 12 Instructions de contrôle
- 13 Initiation aux tableaux
- 14 Initiation aux fonctions
- 15 Conclusions

# Le type booléen :true false

- Le langage C suppose que 0 est false et donc tout ce qui est différent de 0 est true
- Le C99 propose le type `_Bool` défini dans `<stdbool.h>`

```
typedef enum
{
    false = 0, true = 1
} _Bool;
#define bool _Bool
```
- Le type `_Bool` n'est pas vraiment nécessaire mais pour s'approcher du C++

- 1 Le identificateurs
- 2 Déclaration d'une variable
- 3 Les types entiers
- 4 Le type des caractères
- 5 Les chaines de caractères
- 6 Les types flottants
- 7 Le type énuméré
- 8 Le type booléen

- 9 **Les expressions**
- 10 Les instructions
- 11 Entrée/sortie formatées de base
- 12 Instructions de contrôle
- 13 Initiation aux tableaux
- 14 Initiation aux fonctions
- 15 Conclusions

# Une expression ?

Expression est l'une des deux options

① Expression élémentaire est une des trois options

- ① un littéral constant
- ② une variable
- ③ un appel de fonction

② Expression composée est une des trois options

- ① Expression\_1, Opérateur\_unaire
- ② Expression\_1, Expression\_2, Opérateur\_binaire
- ③ Expression\_1, Expression\_2, Expression\_3, Opérateur\_tertiaire

Toute expression dans C a un type statique et une valeur

# Déterminer le type d'une expression

Le type d'une expression se déduit de celui de ses sous-expressions et de son opérateur en utilisant

- Les règles de priorité et d'associativité des opérateurs
- Les règles de conversions implicites de types

# Tableau de priorité (décroissante) et d'associativité

| Opérateur                         | Associativité | Arité |
|-----------------------------------|---------------|-------|
| ()                                | GD            | 1     |
| [] -> .                           |               | 2     |
| ! ~ ++ - + * & (type) sizeof      | DG            | 1     |
| * / %                             | GD            | 2     |
| + -                               | GD            | 2     |
| » «                               | GD            | 2     |
| >= > <= <                         | GD            | 2     |
| == !=                             | GD            | 2     |
| &                                 | GD            | 2     |
| ^                                 | GD            | 2     |
|                                   | GD            | 2     |
| &&                                | GD            | 2     |
|                                   | GD            | 2     |
| ? :                               | DG            | 2     |
| = += -= *= /= %= <<= <<= &= ^=  = | DG            | 2     |
| ,                                 | GD            | 2     |

# Application des règles de priorité et d'associativité des opérateurs

| Expression                   | Expression équivalente avec parenthèses |
|------------------------------|---|
| $a + b + c$                  | $(a + b) + c$                           |
| $a + b - c + d$              | $((a + b) - c) + d$                     |
| $a + b * c - 2$              | $(a + (b * c)) - 2$                     |
| $a + b * c - 7 * 3 / c \% 7$ | $(a + (b * c)) - (((7 * 3) / c) \% 7)$  |
| $a \parallel b \&\& c == 3$  | $a \parallel (b \&\& (c == 3))$         |
| $\odot a = b = c + 2$        | $a = (b = (c + 2))$                     |
| $\odot a = b, b ++$          | $(a = b), (b ++)$                       |
| $\odot a = b ++ + 1$         | $a = ((b ++ ) + 1)$                     |

Soyez explicite car ça va être toujours plus facile à lire ☺☺☺



# Les règles de conversions implicites de types I

- **Promotion entière** : les opérandes de types (`signed unsigned char` et (`unsigned`) `short` sont toujours implicitement convertis en `int` ou `unsigned` (si `int` pas suffisant). (À tester : `char c = 3; c+c` est de type `int`)
- **Les opérations d'arité 2 (binaires)** sont définies pour des opérandes de même type. Toute opération binaire sur des opérandes de types différents mais compatibles nécessite la conversion de l'opérande de plus petit type vers l'autre. (Exception de l'affectation )
- **Ordre entre les types** : `long double > double > float > unsigned long long > long long > unsigned long > long > unsigned int > int`  
(Une petite exception si `long == int` alors `unsigned int > long`)

# Les règles de conversions implicites de types II

- Certains cas de conversions implicites sont indéterminés alors il faut les préciser en utilisant la conversion explicite de type (cast)
- La conversion (cas d'affectation) d'un type entier vers un autre type plus petit se fait par troncature de bits de plus haut poids sans avertissement. ☹
- La conversion d'un type flottant vers un type entier se fait par la suppression de la partie décimale.
- Éviter de combiner un type entier sans signe et un avec signe car le résultat risque d'être inattendue ( $3U > -3$  n'est pas ? c'est faux !!)

- 1 Le identificateurs
- 2 Déclaration d'une variable
- 3 Les types entiers
- 4 Le type des caractères
- 5 Les chaines de caractères
- 6 Les types flottants
- 7 Le type énuméré
- 8 Le type booléen
- 9 Les expressions
- 10 Les instructions**
- 11 Entrée/sortie formatées de base
- 12 Instructions de contrôle
- 13 Initiation aux tableaux
- 14 Initiation aux fonctions
- 15 Conclusions

# Une instruction ?

## Une instruction

### ① une instruction simple

- ① Une expression suivie d'un point virgule ( ; )
- ② une instruction vide, c'est un point virgule ( ; ) tout court

### ② une instruction composée

#### ① Une instruction conditionnelle

- ① if
- ② switch

#### ② Une instruction itératif

- ① for
- ② while
- ③ do while

#### ③ Un bloc d'instructions (Une liste d'instructions en entre { et })

#### ④ Une instruction de branchement inconditionnel (goto)

- 1 Le identificateurs
- 2 Déclaration d'une variable
- 3 Les types entiers
- 4 Le type des caractères
- 5 Les chaines de caractères
- 6 Les types flottants
- 7 Le type énuméré
- 8 Le type booléen
- 9 Les expressions
- 10 Les instructions
- 11 Entrée/sortie formatées de base**
- 12 Instructions de contrôle
- 13 Initiation aux tableaux
- 14 Initiation aux fonctions
- 15 Conclusions

## Les types entiers (signé)

```
char entier_char = 'A'; //code : 65
short entier_court = 18;
int entier = 27;
long entier_long = 77777;
long long entier_llong = 99999;
printf("0- entier_char = [%c]\n",entier_char);
printf("1- entier_char = [%hhd]\n",entier_char);
printf("2- entier_court = [%hd]\n", entier_court);
printf("3- entier = [%d]\n", entier);
printf("3bis- entier = [%o] en octal\n", entier);
printf("3bisbis- entier = [%x] en hexadecimal\n", entier);
printf("4- entier_long = [%ld]\n", entier_long);
printf("5- entier_llong = [%lld]\n", entier_llong);
---Affichage obtenu
0- entier_char = [A]
1- entier_char = [65]
2- entier_court = [18]
3- entier = [27]
3bis- entier = [33] en octal
3bisbis- entier = [1b] en hexadecimal
4- entier_long = [77777]
5- entier_llong = [99999]
```

## Les types entiers non signé

```
unsigned int entier = 27;

printf("1- entier = [%u]\n", entier);
printf("2- entier = [%o] en octal\n", entier);
printf("3- entier = [%x] en hexadécimal\n", entier);
printf("4- entier = [%X] en HEXADECIMAL\n", entier);

---Affichage obtenu
1- entier = [27]
2- entier = [33] en octal
3- entier = [1b] en hexadécimal
4- entier = [1B] en HEXADECIMAL
```

Les conversions `hhu` (unsigned char), `hu` (unsigned short), `Lu` (unsigned long) et `LLu` ((unsigned long long) sont aussi possibles

## Les types flottants

```
float reel = 105.555555555;  
double reel_double = 3487.555555555555e36;  
long double reel_ldouble = 0.55e-136 ;  
printf("1- reel = [%f]\n",reel);  
printf("2- reel_double = [%lf]\n",reel_double);  
printf("3- reel_ldouble = [%Lf]\n",reel_ldouble);  
printf("4- reel = [%e]\n",reel);  
printf("5- reel_double = [%le]\n",reel_double);  
printf("6- reel_ldouble = [%Le]\n",reel_ldouble);  
printf("7- reel = [%g]\n",reel);  
printf("8- reel_double = [%lg]\n",reel_double);  
printf("9- reel_ldouble = [%Lg]\n",reel_ldouble);  
---  
1- reel = [105.555557]  
2- reel_double = [3487555555555554728265617963564606160896.000000]  
3- reel_ldouble = [0.000000]  
4- reel = [1.055556e+02]  
5- reel_double = [3.487556e+39]  
6- reel_ldouble = [5.500000e-137]  
7- reel = [105.556]  
8- reel_double = [3.48756e+39]  
9- reel_ldouble = [5.5e-137]
```



## Les chaines de caractères

```
char *conseil = "Bla bla.";

printf("0- Un conseil = [%s]\n",conseil);
printf("1- Un conseil = [%15s] sur 15 justifié à droite\n",conseil);
printf("2- Un conseil = [%-15s]sur 15 justifié à gauche\n",conseil);
printf("3- Un conseil = [%4s] sur 4 poutant\n",conseil);
---
```

0- Un conseil = [Bla bla.]  
1- Un conseil = [ Bla bla.] sur 15 justifié à droite  
2- Un conseil = [Bla bla. ]sur 15 justifié à gauche  
3- Un conseil = [Bla bla.] sur 4 poutant

## Sortie formatée

```
int printf(const char *format, ...);
```

- `format` est une chaîne de caractères à afficher telle quelle sauf pour les portions débutant par un `%` et se terminant par un indicateur de conversion ; celles-là correspondent des spécifications d'affichage se rapportant aux paramètres restant de `printf`
- Le 1er `%` se rapporte à l'expression donnée comme 2ième paramètres, le 2ième `%` au à la 3ième expression, etc
- Chaque spécification d'affichage est remplacée par l'expression correspondante au moment de l'affichage

# Spécification d'affichage dans `printf`

`%[<attribut>][<largeurMin>][.<precision>][<modifLong>]<conversion>`

- **<conversion>** : Indicateur de conversion précisant le type de l'expression (`d u f e E ...`)
- **<modifLong>** : pour les conversions entières `h, hh, l, ll` et pour les conversions des flottants (`l, L, ...`)
- **<precision>** : un nombre entier représentant le nombre de chiffres après la virgule pour les flottants. (par défaut c'est 6)
- **<largeurMin>** : entier représentant le nombre de caractères min réserver où afficher l'expression (compléter avec des espaces)
- **<attribut>** : un ou plusieurs caractères comme
  - `0` : les espaces devant un nombre seront remplis de `0` sur **<largeurMin>** spécifié
  - `-` : justification à gauche de l'expression dans l'espace réservé avec **<largeurMin>**
  - une espace : les nombres positifs ont une espace au lieu du signe
  - `+` : pour forcer l'affichage de `+` devant les nombres positifs

## Exemples généraux

```
float reel_pos= 125.555555555;  
float reel_neg = -125.555555555;  
printf("0- Pour afficher %% c'est %%%\n");  
printf("1- %%f : [%f]\n",reel_pos);  
printf("2- %g : [%g]\n",reel_pos);  
printf("3- %g : [%g]\n",reel_neg);  
printf("4- %10.2f : [%10.2f]\n",reel_pos);  
printf("5-      %10.2f : [%10.2f]\n",reel_pos);  
printf("5bis- %10.2f : [%10.2f]\n",reel_pos);  
printf("6- %10.2f : [%10.2f]\n",reel_pos);  
--  
0- Pour afficher % c'est %  
1- %f : [125.555557]  
2- %g : [+125.555557]  
3- %g : [-125.555557]  
4- %10.2f : [   +125.56]  
5-      %10.2f : [+125.56   ]  
5bis- %10.2f : [+125.56   ]  
6- %10.2f : [+000125.56]
```

Une mauvaise spécification peut donner lieu à des résultats erronés

# Saisie d'un caractère : `int getchar(void) ;`

## Listing 1 – `getchar.c`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int
5 main()
6 {
7     int i;
8     char c;
9     for(i = 0 ; (c = getchar()) != EOF ; i++ )
10    //EOF : fin de fichier
11    {
12        putchar(c); // printf("%c",c);
13    }
14    printf("Le_nombre_de_caractères_lus_:_%d\n",i);
15
16    return EXIT_SUCCESS;
17 }
```

# Saisie d'un caractère : `int getchar(void) ;` ||

## Exécution de `getchar.c`

```
$ gcc -Wall -std=c99 getchar.c -o getchar
$ ./getchar
abc
abc
dfe
dfe
^D
Le nombre de caractères lus : 8
$
```

## Interprétation

la touche entrer introduit un caractère de fin de ligne après `abc` et après `dfe`. Le `^D` (fin de fichier) ne compte pas.

# Saisie formatée des données : `scanf()`

```
int scanf(const char * format,...)
```

- la fonction `scanf()` permet de saisir des données sur le canal de de l'entrée standard et d'affecter les valeurs saisies aux variables fournies comme paramètres
- La fonction `scanf()` ressemble à la fonction `printf()` dans sa manière de faire la correspondance entre `format` et ses autres paramètres restants
- `format` peut avoir plusieurs spécification

**`%[<longMax>][<modifLong><conversion>`**

- ▶ **`<longMax>`** : un entier représentant le nombre de caractères max à considérer dans le canal de l'entrée standard
- ▶ **`<modifLong>`** : idem que `printf()` (`hh, h, l, ll, ...`)
- ▶ **`<conversion>`** : idem que `printf()` (`d u f e E ...`)

# Exemples courants avec `scanf`

```
//Code source
char entier_char;
short entier_court ;
scanf("%c",&entier_char);
printf("[%c]\n",entier_char);

scanf("%hhd",&entier_char);
printf("[%c]\n",entier_char);

scanf("%hd", &entier_court);
printf("[%hd]\n", entier_court);

scanf("%hX", &entier_court);
printf("[%hX]\n", entier_court);
printf("[%hd]\n", entier_court);
printf("[%hu]\n", entier_court);
```

Exec1

```
-
-
A
[A]

65
[A]

136
[136]

AB00
[AB00]
[-21760]
[43776]
```

Exec2

```
-
-
B
[B]

66
[B]

-136
[-136]

1B0002
[2]
[2]
[2]
```

Exec3

```
-
-
B
[B]

h
[?]

[?]
[?]
[?]
```



# Spécification de conversion dans `scanf()`

| Spécification            | Signification                            | Parsing   |
|--------------------------|--|---|
| <code>%c</code>          | <code>char</code>                        | accepte tout caractère  |
| <code>%hhd (%hhu)</code> | <code>signed char (unsigned char)</code> | ignore les blancs<br>jusqu'à un entier                              |
| <code>%hd (%hu)</code>   | <code>short (unsigned short)</code>      |   |
| <code>%d (%u)</code>     | <code>signed (unsigned)</code>           |   |
| <code>%ld (%lu)</code>   | <code>long (unsigned long)</code>        |   |
| <code>%lld (%llu)</code> | <code>long (unsigned long long)</code>   |   |
| <code>%f</code>          | <code>float</code>                       | ignore les blancs<br>jusqu'à un flottant                            |
| <code>%lf</code>         | <code>double</code>                      |   |
| <code>%Lf</code>         | <code>long double</code>                 |   |
| <code>%s</code>          | chaîne de caractères                     | prend tous jusqu'au<br>blanc et met ' <code>\0</code> ' à<br>la fin |

Une mauvaise spécification donne lieu à des valeurs inattendues !

# Comportement du buffer

Les caractères non consommés restent dans le buffer (zone tampon ) pour les lectures futures

```
int entier1, entier2;  
char car;  
scanf ("%d%d%c", &entier1, &entier2, &car);
```

Saisir 

|     |
|-----|
| 15↵ |
| 12↵ |

 ou 

|        |
|--------|
| 15_12↵ |
|--------|

 donne le même résultat

- entier1 : 15
- entier2 : 12
- car : '\n' (provient de ↵)

Pour lire un caractère spécifique dans `car` il faudrait consommer le '\n'

```
scanf ("%d%d%c", &entier1, &entier2, &car);  
scanf ("%c", &car);
```

# Saisie d'une chaîne de caractères

```
char chaine[30]; //taille max : 29 caractères + '\0'  
scanf("%s",chaine); //notez pas de & devant chaine
```

Saisir  donne chaine : "Salut"

- La fonction `scanf()` s'arrête au premier caractère blanc rencontré
- `scanf()` se charge de mettre '`\0`' à la fin
- C'est au programmeur de prévoir la bonne taille en mémoire pour la chaîne avant d'appeler `scanf()`

- 1 Le identificateurs
- 2 Déclaration d'une variable
- 3 Les types entiers
- 4 Le type des caractères
- 5 Les chaines de caractères
- 6 Les types flottants
- 7 Le type énuméré
- 8 Le type booléen
- 9 Les expressions
- 10 Les instructions
- 11 Entrée/sortie formatées de base
- 12 Instructions de contrôle**
- 13 Initiation aux tableaux
- 14 Initiation aux fonctions
- 15 Conclusions

# Expression conditionnelle (opérateur tertiaire)

`expression1 ? expression2 : expression3`

- retourne `expression2` si `expression1` est vraie (différente de 0).
- retourne `expression3` si `expression1` est fausse (égale à 0).
- Exemple : `max = (x > y) ? x : y;`

```
1
2 x = r ? r+1 : r-1 ;
3
4 ou bien
5
6 if (r)
7     x = r + 1;
8 else
9     x = r - 1;
```

```
1 float r = 6.3;
2 int x,y ;
3
4 x = r ? r+1 : r-1 ;
5 // x:7.3
6
7 y = r-1 < 5 ? r*r : r-1;
8 // y :5.3
9
10 y = (r-1 < 5) ? (r*r) : (r-1);
```

# L'instruction if

```
if (<expression>
    <instruction>
[else
    <instruction>]
```

La clause `else` est optionnelle

```
1 int note;
2 char decision;
3
4 printf("Donnez_note_:");
5 scanf("%d",&decision)
6 if ( note >= 60 )
7 {//accolades non necessaires
8     decision = 'R';
9 }
10 else
11 {
12     decision = 'E';
13     printf("Échec");
14 }
```

# L'instruction switch I

```
switch (<expression_entière>)  
{  
    case <expression_constante_entière> :  
        <liste_instructions>  
    .  
    .  
    .  
    [default :  
        <liste_instructions>]  
}
```

- La liste d'instructions peut être vide
- La liste d'instructions peut contenir `break` (à sa fin)
- Le premier `break` exécuté cause un saut vers `}` du `switch`
- La clause `default` est optionnelle

# L'instruction switch II

```
3 #define BE 'B'
4 int
5 main()
6 {
7     char choix;
8     printf("Donnez_choix_a/A,b/B:_");
9     scanf("%c",&choix);
10    switch (choix) //pas break
11    {
12        case 'a' :
13        case 'A' :
14            printf("Traitement_A");
15        case 'b' :
16        case BE : //celle-ci est une macro
17            printf("Traitement_B");
18        default :
19            printf("Je_ne_sais_pas_quoi_faire!");
20    }
```



# L'instruction switch (ne pas oublier `break` au besoin)

```
1 switch (choix) //avec break
2 {
3     case 'a' :
4     case 'A' :
5         printf("Traitement_A\n");
6         break;
7
8     case 'b' :
9     case 'B' :
10        printf("Traitement_B\n");
11        break;
12
13    default :
14        printf("Je_ne_sais_pas_quoi_faire!");
15        break;
16 }
```

# Les boucles `while` et `do while`

```
while (<expression>)  
{// accolades au besoin  
  
    <instruction>  
  
}
```

```
do  
{ // accolades au besoin  
  
    <instruction>  
  
}  
while (<expression>;
```

- Le comportement des boucles `while` et `do while` peut être altéré avec les instructions d'échappement `continue;` et `break;`
  - ▶ L'exécution de `break;` termine la boucle
  - ▶ L'exécution de `continue;` cause un saut vers le test de la condition de la boucle

# Illustration de break et continue

```
8  char choix = 0;
9  do
10 {
11     printf("choix__:_%hhhd\n", choix);
12     choix ++;
13     if (!(choix%5)) //vrai : 0 5 10
14         ...
15         continue; //sauter à la ligne
16         19
17     choix = choix + 1;
18     if (choix > 6)
19         break; //sauter à la ligne 20
20 }
21 while (true);
22 printf("choix_fin__:_%hhhd\n",choix);
```

```
choix    : 0
choix    : 2
choix    : 4
choix    : 5
choix fin : 7
```

# La boucle for

```
for ([<exp_init>] ; [<exp_condition>] ; [<expr_incr>] )  
{  
    // accolades au besoin  
  
    <instruction_boucle>  
  
}  
  
/**Avec C99**/  
for ([<exp_decla_init>] ; [<exp_condition>] ; [<expr_incr>] )  
{  
    // accolades au besoin  
  
    <instruction_boucle>  
  
}
```

Le déroulement de l'exécution d'une boucle for

- 1 Exécution de <exp\_init>; ou de <exp\_decla\_init>; au début seulement
- 2 Ensuite tant que <exp\_condition> est vraie (ou absente), il y a exécution de <instruction\_boucle> suivie de <exp\_incr>;

# Illustration de break et continue avec la boucle for

```
8  char choix = 'a';
9  for (char c = choix , i = 0 ; i<10 ; i++ , c++)
10 {
11     c = c + 1;
12
13     printf("choix__:_%c\n", choix);
14     choix ++;
15     if (c < 'c')
16         continue; //exécuter i++, c++; puis
                     //sauter à la ligne 10
17     choix = choix + 1;
18     if (choix > 'f')
19         break; //sauter à la ligne 22
20     choix = c;
21 }
22 printf("choix_fin__:_%c\n",choix);
```

```
choix : a
choix : b
choix : d
choix : f
choix fin : h
```

# Branchement inconditionnel : goto

## Syntaxe de goto

```
goto <identificateur_étiquette> ;
```

## Syntaxe d'une déclaration d'étiquette

```
<identificateur_étiquette> : <instruction>
```

- L'exécution de `goto` donne lieu à un saut vers l'étiquette indiquée
- L'utilisation de `goto` peut rapidement dégrader la qualité du code
- L'usage `goto` devrait se limiter seulement à des cas où `break` et `continue` ne suffisent pas comme :
  - ▶ Échapper d'une boucle à partir d'un `switch` dans boucle (`goto` vers étiquette d'une instruction vide placée juste après la boucle)
  - ▶ Échapper de plusieurs boucles imbriquées d'un coup (`goto` vers étiquette placée juste après la boucle périphérique)
  - ▶ Échapper d'une situation d'erreur soit pour la reprise depuis le début ou bien traiter l'erreur

- 1 Le identificateurs
- 2 Déclaration d'une variable
- 3 Les types entiers
- 4 Le type des caractères
- 5 Les chaines de caractères
- 6 Les types flottants
- 7 Le type énuméré
- 8 Le type booléen
- 9 Les expressions
- 10 Les instructions
- 11 Entrée/sortie formatées de base
- 12 Instructions de contrôle
- 13 Initiation aux tableaux**
- 14 Initiation aux fonctions
- 15 Conclusions

# Les tableaux (vecteurs) : déclaration

Syntaxe : `<type> <identificateur>[<constante_entière>];`

- `<type>` est l'un des types reconnus (`int`, `short`, ... ou défini dans le programme)
- `<identificateur>` est le nom de la variable référençant le tableau
- `<constante_entière>` est la taille du tableau (à partir de C99 la taille d'un tableau peut être définie par une expression entière contenant des variables (`gcc` depuis C90 comme une extension de la norme))

## Exemple

```
char cars[5];
```

|   |   |   |   |   |
|---|---|---|---|---|
|   |   |   |   |   |
| 0 | 1 | 2 | 3 | 4 |

```
int taille = 3;
```

```
unsigned int vect[taille];
```

|   |   |   |
|---|---|---|
|   |   |   |
| 0 | 1 | 2 |



# Manipulation de tableau

```
#define DIM 5  
int valeurs1[DIM] = {77, 80, 47, 89, 100};  
for (int i = 0 ; i < DIM ; i++)  
    printf("%d ", valeurs1[i]);  
  
int valeurs2[]={7,0,8,2,4,0,6,7,8,9};  
//La taille sera déterminée à la compilation  
  
int valeurs3[10]={7,0,8};  
//Le compilateur l'accepte (valeurs restantes à 0)
```

# Tableau à deux dimensions

```
int mat[2][3]={ {0,1,2}, {3,4,5}};
```

mat[1][2] : ligne 1 colonne 2 (de valeur 5)

- 1 Le identificateurs
- 2 Déclaration d'une variable
- 3 Les types entiers
- 4 Le type des caractères
- 5 Les chaines de caractères
- 6 Les types flottants
- 7 Le type énuméré
- 8 Le type booléen
- 9 Les expressions
- 10 Les instructions
- 11 Entrée/sortie formatées de base
- 12 Instructions de contrôle
- 13 Initiation aux tableaux
- 14 Initiation aux fonctions**
- 15 Conclusions

# Les fonctions, un exemple, un rappel

```
1 int maximum (int a, int b); //Déclaration de la fonction ou prototype
2
3 int
4 main()
5 {
6     int n=4, resultat;
7     resultat = maximum(n,1);
8 }
9 int
10 maximum (int a, int b) //Définition de la fonction
11 {
12     int m;
13     if ( a < b )
14         m = b;
15     else
16         m = a;
17     return m;
18 }
```

# utilisation d'une librairie

```
1 #include <math.h>
2
3 int
4 main()
5 {
6     double resultat;
7     resultat = pow(4,0.5);
8 }
```

```
1 double pow(double x, double y);
2
3 int
4 main()
5 {
6     double resultat;
7     resultat = pow(4,0.5);
8 }
```

## Compilation

```
$ gcc -lm prog_lib_math.c prog
```

- `#include <math.h>` permet la déclaration de fonction `pow()`  
`double pow(double x, double y);`
- `-lm` permet l'édition des liens avec la librairie `m` (fichier `libm.so...`) des fonctions mathématiques

## compilation

```
$ gcc -Wall -stdc99 -lm prog_lib_math.c prog
```

## Liste des librairies requises

```
$ objdump -x prog | grep NEEDED
NEEDED      libm.so.6
NEEDED      libc.so.6

$ ldd prog
linux-vdso.so.1 => (0x00007fff483fd000)
libm.so.6 => /lib64/libm.so.6 (0x000000392fe00000)
libc.so.6 => /lib64/libc.so.6 (0x000000392fa00000)
/lib64/ld-linux-x86-64.so.2 (0x000000392f600000)
```

# Les fonctions, les cas à distinguer

- ❶ Fonction de librairie standard de C  
([http://en.wikipedia.org/wiki/C\\_standard\\_library](http://en.wikipedia.org/wiki/C_standard_library))
  - `<ctype.h>`, `<time.h>`
  - inclure le fichier `.h` déclarant la fonction
- ❷ Fonction d'une librairie autre
  - inclure le fichier `.h` déclarant la fonction
  - utiliser `-l<nom_lib>` pour l'édition des liens
    - ★ inclure `<math.h>` pour pouvoir appeler la fonction `sqrt()` dans `prog.c`
    - ★ compilation et édition des liens : `gcc -lm prog.c`
- ❸ Fonction définie par le programme
  - déclarer la fonction ou utiliser `#include` le fichier d'entête `.h` déclarant la fonction

- 1 Le identificateurs
- 2 Déclaration d'une variable
- 3 Les types entiers
- 4 Le type des caractères
- 5 Les chaines de caractères
- 6 Les types flottants
- 7 Le type énuméré
- 8 Le type booléen
- 9 Les expressions
- 10 Les instructions
- 11 Entrée/sortie formatées de base
- 12 Instructions de contrôle
- 13 Initiation aux tableaux
- 14 Initiation aux fonctions
- 15 Conclusions**



- Les notions couvertes jusqu'à présent sont très similaires à ce que vous avez vu en Java dans INF1120 et INF2120
- Une différence majeure : le C donne souvent raison au programmeur et ne force pas toujours des vérifications de type strictes