**UE104 – Architecture des systèmes**

**HAUTE ÉCOLE DE NAMUR-LIÈGE-LUXEMBOURG**

**Bloc 1**

Atelier 3 : codage des nombres

Objectifs

* savoir comment sont codés les nombres entiers en mémoire
* savoir comment sont codés les nombres réels en mémoire

# Introduction

Les ordinateurs ne sont pas équipés pour traiter n’importe quel type de données. Les informations traitées par un ordinateur sont représentées sous forme de nombres. L'atelier précédent a permis de comprendre qu'un nombre peut être représenté selon différents systèmes de numération.

L'ordinateur, de par la technologie qui le caractérise, ne supporte qu'un seul de ces systèmes : le binaire. De plus, pour que les informations soient portables (utilisables sur différentes machines), il est nécessaire de respecter les mêmes conventions/normes.

Cet atelier a donc pour objectif de vous permettre de savoir comment sont codés les nombres afin d'être traités et mémorisés par un ordinateur.

# Codage des entiers

Les entiers peuvent être codés sur différentes tailles (nombres de bit occupés), et être signés ou non.   
S’ils sont non signés, aucune valeur négative ne peut être stockée.

Concrètement, il faut donc commencer par faire des choix :

* A-t-on besoin d’un entier signé ou non signé ?
* De quelle place mémoire doit-on disposer pour y stocker les valeurs à traiter ?

La réponse à ces questions dépend des valeurs qui devront être stockées dans la variable : l'âge des utilisateurs, les températures saisonnières au cours de l'année, le montant d'un emprunt hypothécaire…

Cependant, pour savoir quelle taille allouer à une variable, il faut connaître les valeurs maximum et minimum associées à cette taille.

Prenons une taille mémoire de 16 bits, combien de nombres différents peut-on représenter ?

216 = 65 536

On en revient à la question de savoir si on représente uniquement des nombres positifs ou si on représente des nombres qui peuvent être positifs ou négatifs.

* **Non signés** : on peut représenter les nombres de 0à216-1.   
  On aurait pu choisir de 1 à 216, mais c'est bien pratique de pouvoir représenter le zéro.
* **Signés** : si on veut le 0, il reste 216-1 = (2\*215)-1 = 215 + (215 – 1) possibilités à répartir entre les négatifs et les positifs.   
  Par convention, on choisit de représenter les nombres de -215à 215-1.

Par convention, trois tailles de nombre sont prévues, respectant les spécifications matérielles de la plupart des ordinateurs. Suivant la capacité maximale souhaitée, l’entier peut être *court*, *normal* ou *long*. On retrouve d'ailleurs ces trois tailles dans la syntaxe du langage C.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Nombre d’octets | Entier signé | Entier non signé |
| court | 2 | -32 768 à 32 767 | 0 à 65 535 |
| normal | Minimum 2  (4 le plus souvent) | -2 147 483 648 à 2 147 483 647 | 0 à  4 294 967 295 |
| long | Minimum 4 (8 accepté) |  |  |

En C, l’opérateur sizeof permet de vérifier la taille en octets du type employé.   
sizeof(int) renvoie la valeur de la taille occupée en mémoire par un entier de type int.   
La norme impose la règle suivante : sizeof(short) ≤ sizeof(int) ≤ sizeof(long).

La question suivante est de savoir comment **coder** **les entiers négatifs**. Trois codages sont possibles :

* bit de signe et valeur absolue
* complément à 1
* complément à 2

Par simplicité, les exemples sont présentés avec des nombres codés sur 8 bits.

## Bit de signe et valeur absolue

Le bit de poids fort (le plus à gauche) représente le signe et les autres correspondent à la valeur exprimée en valeur absolue.

Le bit de signe vaut

0 pour une valeur positive et

1 pour une valeur négative.

Sur 8 bits,

2710 est codé **0**001 10112 et

-2710 est codé **1**001 10112.

De la même façon, sur 8 bits,

010 est codé 0000 00002 et

-010 est codé 1000 00002.

Il y a donc deux façons de coder 010. Ce qui n'est pas pratique !

## Complément à 1

Pour le codage des entiers positifs, il suffit de convertir le nombre du système d'origine vers le système binaire.

Pour le codage des négatifs, on convertit la valeur absolue de l'entier à coder en binaire et on passe à son opposée en inversant les bits. En fait, il s'agit de complémenter chaque bit par rapport à 1 en lui ajoutant 1 ou 0 selon le cas.

Les nombres positifs commencent donc par 0 et les nombres négatifs commencent par 1.

Sur 8 bits,

2710 est codé **0**001 10112 et

-2710 est codé **1**110 01002.

De la même façon, sur 8 bits,

010 est codé **0**000 00002 et

-010 est codé **1**111 11112.

Il y a donc à nouveau deux façons de coder 010. Ce qui n'est toujours pas pratique !

Pour décoder un nombre **négatif** exprimé en complément à 1, on inverse chaque bit et on convertit en base 10 pour en connaître la valeur dans le système décimal.

## Complément à 2

Comme pour le complément à 1, pour le codage des positifs, une simple conversion suffit.

Pour le codage des négatifs, on calcule le complément à 2.

Sur 8 bits,

2710 est codé **0**001 10112 et

-2710 est codé **1**110 01002 + 12 = **1**110 01012

De la même façon, sur 8 bits,

010 est codé **0**000 00002 et

-010 est codé **1**111 11112 + 12 = **~~1~~** **0**000 00002.

En effet, comme le nombre de bits occupés par le codage d'un nombre est fixé dès le départ, dans cet exemple à 8 bits, le dernier report est "perdu" ! On parle de dépassement de capacité ou *overflow*. C'est donc le fait d'ajouter 1 qui permet d'obtenir une seule façon de coder 010.

### Exemple 1

Soit 1910 en base 10 à stocker en mémoire sur 8 bits :

Représentation binaire : 0001 0011

### ExemplE 2

Soit -1910 en base 10 à stocker en mémoire sur 16 bits :

1. Représentation binaire de 19 : 0000 0000 0001 0011
2. Complément à 1 : 1111 1111 1110 1100
3. Complément à 2 : 1111 1111 1110 1101

### Exemple 3

Soit -13210 en base 10 à stocker en mémoire sur 16 bits :

1. Représentation binaire de 132 : 0000 0000 1000 0100
2. Complément à 1 : 1111 1111 0111 1011
3. Complément à 2 : 1111 1111 0111 1100

Ce système est à la fois plus compliqué pour l'humain (mais heureusement, il ne doit pas faire toutes ces conversions à la main... d'habitude) et plus simple pour la machine (la somme de deux valeurs codées en compléments à 2 est plus facile à reproduire en électronique).

En effet, dans le cas du complément à 1 avec un nombre codé sur 8 bits, la somme de deux nombres opposés ne donne pas le résultat auquel on s'attend :

En décimal En binaire

27 0001 1011

+ -27 + 1110 0100

0 1111 1111

En décimal, on obtient 0, comme attendu, mais en binaire codé en complément à 1, le résultat n'est pas 0, puisqu'on obtient comme résultat 1111 1111 !

Par contre, dans le cas du complément à 2, on obtient bien le résultat espéré :

En décimal En binaire

*11111 111*

27 0001 1011

+ -27 + 1110 0101

0 10000 0000

Le résultat de la somme de deux nombres opposés donne 0 car on perd la dernière retenue, puisque le nombre est codé sur 8 bits.

C'est donc la méthode du complément à 2 qui est actuellement utilisée pour coder les entiers.

Pour rappel,

* cette façon de faire est plus facile à réaliser au niveau des circuits,
* l’entier 010 est codé d'une seule façon : 0000 00002, et
* la somme d'un nombre et de son opposé donne 0.

Compléments d’explications :

• <http://villemin.gerard.free.fr/Wwwgvmm/Numerati/BINAIRE/Negatif.htm>

## Capacités minimale et maximale des entiers

En conclusion de l'explication ci-dessus, les capacités maximales pour des entiers sont :

* sur une zone de n bits, en ne considérant que des entiers **non signés**, le nombre le plus élevé vaut 2n-1.
* sur cette même zone, si on considère les entiers **signés**, les valeurs sont comprises entre -2n-1 à 2n-1-1.

### Exemple 1

Soit l'entier **signé** 1111 11112 sur 8 bits. Quel est ce nombre ?

Démarche inverse… **en n’oubliant pas que, comme le premier bit est à 1, il s'agit d'un négatif**.

1. Retirer 1 : 1111 1110
2. Inverser : 0000 0001
3. Convertir en base 10 : -1

### Exemple 2

Soit l'entier **signé** 1000 00002, sur 8 bits. Quel est ce nombre ?

1. Retirer 1 : 0111 1111
2. Inverser : 1000 0000
3. Convertir en base 10 : -27 = -128

Le plus petit nombre codable sur 8 bits : -2n-1

### Exemple 3

Soit l'entier **signé** 0111 11112, sur 8 bits. Quel est ce nombre ?

26+ 25 + 24 + 23 + 22 + 21 + 20 = 27 – 1 = 127

Le plus grand nombre codable sur 8 bits : 2n-1-1

## Propagation du signe

Dans certains cas, un nombre codé sur un certain nombre de bits doit être mémorisé dans un espace mémoire plus grand. Il faut donc veiller à ce que la valeur codée reste celle de départ ! Après avoir encodé le nombre en complément à 2, il faut donc **propager le signe**.

Pour étendre la taille d’un nombre non signé, on ajoute des 0 à sa gauche.

Pour étendre la taille d’un nombre signé, on ajoute sur la gauche des bits identiques au bit de signe.

### Exemple

(-4)10

* codé sur 1 octet = FC16 = **1**111 11002
* codé sur 2 octets = FFFC16 = **1111 1111** 1111 11002

(+4)10

* codé sur 1 octet = 0416 = **0**000 01002
* codé sur 2 octets = 000416 = **0000 0000** 0000 01002

## Exercices

1. Calculer le complément à 1 des nombres suivants codés sur 1 octet.
   1. 1001 11012
   2. 0011 00112
2. Donner le codage en complément à 2 des entiers signés suivants, sur 1 octets.
   1. 3710
   2. -1310
   3. -12710
3. Donner la représentation décimale et hexadécimale des entiers signés suivants, codés en complément à 2.
   1. 1100 11012
   2. 0000 11012
   3. 1001 0001 0011 00102
   4. 0000 0000 0111 0101*2*
4. Quelle valeur minimum et quelle valeur maximum peut-on coder sur 32 bits, en considérant des entiers signés ?
5. Vrai ou faux ? Les nombres signés suivants peuvent-ils être encodés sur un seul octet ?

12810 ❑ Vrai ❑ Faux

-12810 ❑ Vrai ❑ Faux

25510 ❑ Vrai ❑ Faux

12710 ❑ Vrai ❑ Faux

# Codage des réels

Comme pour les nombres entiers, il existe plusieurs façons de procéder pour mémoriser un nombre réel :

* réels à virgule fixe
* réels à virgule flottante

## Codage des réels à virgule fixe

Le principe est de fixer un nombre de bits pour coder la partie décimale du nombre, c'est-à-dire celle correspondant aux **puissances de 2** **négatives**, les autres bits représentent la partie entière.

Par exemple, si on dispose de N bits, on peut décider que seuls les 3 derniers sont ceux alloués à la partie décimale.

Sur 8 bits,

01011,0112 = 1\*23 + 1\*21 + 1\*20 + 1\*2-2 + 1\*2-3

= 8 + 2 + 1 + 0,25 + 0,125

= 11,37510

Cependant, ce codage ne permet pas de gérer des grands nombres et des petits nombres simultanément. De plus, il provoque des **erreurs d’arrondis**.

En effet, sur base de 2 nombres binaires non signés codés sur 8 bits avec 2 bits pour la partie décimale:

1110 = 001011,002 et 810 = 001000,002

si on divise 11 par 8, on devrait avoir

1,37510 = 000001,0112

mais on obtient

1,2510 = 000001,012

**car on a perdu le dernier bit** !

## Codage des réels à virgule flottante

Le nombre 1010112 peut s’écrire sous plusieurs formes :

10101,12 \* 21

1010,112 \* 22

101,0112 \* 23

10,10112 \* 24

1,010112 \* 25

En fonction de l'endroit où on place le séparateur décimal, on doit multiplier par une puissance de la base, ici 2, afin de conserver la valeur initiale.   
On parle de virgule flottante : la virgule est placée selon la valeur de l’exposant.

Cette façon de faire permet de manipuler des grands nombres et/ou des petits nombres simultanément, car il y a une séparation entre la précision (**mantisse**/**significande**) et l’ordre de grandeur (**exposant**).

En base 10, une autre façon d'exprimer un réel, appelée notation scientifique, est la suivante : **mantisse** \* **10exposant**

## Norme IEEE754

Actuellement, tous les constructeurs utilisent cette façon de coder les réels, publiée en **1985**, à savoir qu’un nombre en binaire et en virgule flottante s’écrira sous la forme

 (1 + mantisse) \* 2exposant

Il y a quelques différences avec la notation scientifique originale qui sont précisées ci-après.

Afin de mémoriser un nombre réel, il faut déterminer les 3 champs qui permettent son codage :

* le signe,
* la mantisse et
* l'exposant

De plus, la norme propose 3 modes de codage :

* les nombres « simple précision » codés sur 32 bits
* les nombres « double précision » codés sur 64 bits
* les nombres « précision étendue » codés sur 80 bits

En C, chacun de ces modes correspond à un type : float, double et long double.

En mémoire, chaque élément occupe un certain nombre de bits selon le mode choisi :

|  |  |  |
| --- | --- | --- |
| Signe | Exposant | Mantisse |
| 1 bit | 8 bits | 23 bits |
| 1 bit | 11 bits | 52 bits |
| 1 bit | 15 bits | 64 bits |

Dans le cadre de cet atelier, seuls des nombres codés en simple précision seront utilisés.

Comme dans le cas des entiers, le bit de **signe** vaut 0 pour une valeur positive et 1 pour une valeur négative.

Comme expliqué précédemment, il est possible de représenter une même valeur de différentes façons. Dans le cas de la norme, l'écriture d'un nombre réel est dite **normalisée**, c'est-à-dire qu'il n'y a qu'un seul chiffre non nul avant la virgule. En considérant l'exemple ci-dessus, c'est la notation 1,010112 \* 25 qui est normalisée. Cette notation assure l'unicité du codage et permet de gagner une position dans la mantisse…

En effet, comme le codage du nombre se fait en binaire, le chiffre avant la virgule est nécessairement 1 (puisqu'il ne peut être 0). L'idée est donc de ne pas coder ce 1 afin de gagner une position dans la mantisse et donc un bit de précision… Ce bit est appelé le bit caché… et explique le fait d'ajouter 1 dans la formule " (1 + mantisse) \* 2exposant".

La **mantisse** est donc constituée de la partie décimale du nombre normalisé.

L’**exposant** est dit **polarisé**, en ce sens qu’on lui ajoute systématiquement 127, dans le cas de la simple précision, avant de le coder. L'exposant peut en effet être positif ou négatif, ce qui nécessiterait de le coder de façon particulière. Afin d'éviter des complications au niveau des circuits en utilisant le complément à 2, il a été décidé d'utiliser la technique du **biais**. En effet, un biais est ajouté à l'encodage et est soustrait lors du décodage.

Question. Pourquoi 127 ?

En simple précision, l'exposant est codé sur 8 bits. Les valeurs d'exposant 0 (0000 00002) et 255 (1111 11112) sont réservées au codage de valeurs particulières.   
Les valeurs d'exposant possibles sont donc comprises entre -126 et +127. En ajoutant 127, l'exposant est toujours positif et situé entre 1 et 254.

### Exemple 1

Voici un nombre « très grand » :

+9352710 = +1011 0110 1010 10111,02 = 1,0110110101010111 \* 216

signe : positif 🡪 bit de signe = 0

Exposant : 16 🡪 exposant polarisé = 16 + 127 = 143 = 100011112

Mantisse : 0110110101010111

Représentation : 0 10001111 01101101010101110000000

### Exemple 2

Voici un nombre « très petit » :

-0,0551757812510 = -0,00001110001 = - 1,110001 \* 2-5

Signe : négatif 🡪 bit du signe = 1

Exposant : -5 🡪 exposant polarisé = -5 + 127 = 122 = 011110102

Mantisse : 110001

Représentation : 1 01111010 11000100000000000000000

## Capacités minimale et maximale des réels

En langage C, il y a 3 types de réels : float, double et long double (de plus en plus de précision dans les décimales de la mantisse).

La norme impose une règle : sizeof(float) ≤ sizeof(double) ≤ sizeof(long double).

Un réel est toujours signé.

### Limites au niveau des exposants

Valeurs particulières :

* si tous les bits valent 1 : overflow
* si tous les bits valent 0 : underflow

Le plus **petitexposant polarisé** possible est :

000000012 = 110 soit un plus petit exposant réel : 1 - 127 = -126

Le plus **grandexposantpolarisé** possible est :

111111102 = 25410 soit un plus grand exposant réel : 254 - 127 = 127

### Limites au niveau de la mantisse

La **mantisse minimale** est : 000000000000000000000002 = 0

Valeur minimale pour (1 + mantisse) : 1

La **mantisse maximale** est : 111111111111111111111112 = 1 - 2-23

Valeur maximale pour (1 + mantisse) : 1 + (1 - 2-23) = 2 - 2-23

= 1.999999881

### Valeurs minimales et maximales

Pour les nombres positifs,

le plus petit nombre représentable est

**1**\*2**-126** = 1.17\*10-38

le plus grand nombre représentable est

**1.999999881**\*2**127** = 3.4\*1038

Pour les nombres négatifs,

le plus petit nombre représentable est

**-1.999999881**\*2**127** = -3.4\*1038

le plus grand nombre représentable est

**-1**\*2**-126** = -1.17\*10-38

Les valeurs minimum et maximum d'un réel sont donc connues, mais ce n'est pas pour autant que tous les nombres entre ces deux valeurs peuvent être représentés. En effet, certaines valeurs qui paraissent toutes simples à représenter en décimal ne sont pas représentables en binaire. Par exemple, 0,1 ou 0,7, qu'il est impossible de représenter exactement en binaire.

## Configurations particulières

### Infini

Pour coder l’infini, il faut remplir le champ exposant avec des 1 et le champ mantisse avec des 0, on a alors :

* moins l’infini (–∞) si le bit de signe est à 1

1 11111111 00000000000000000000000

* plus l’infini (+∞) si le bit de signe est à 0

0 11111111 00000000000000000000000

### Zéro

La notation scientifique ne permet pas d’écrire un « vrai » 0.

Pour coder le zéro, il faut remplir le champ exposant avec des 0 et le champ mantisse avec des 0, on a alors :

* moins zéro (–0) si le bit de signe est à 1

1 00000000 00000000000000000000000

* plus zéro (+0) si le bit de signe est à 0

0 00000000 00000000000000000000000

### Not a Number

Si une opération ne peut pas retourner un nombre réel, elle produit un code particulier. Ce code est appelé « **NaN** » pour « **Not a Number** »

* le bit de signe n’est pas significatif (mais souvent à 1)
* tous les bits du champ exposant sont à 1
* le champ mantisse est différent de 0 (mais tous les bits sont souvent à 1)

1 11111111 11111111111111111111111

### Nombres dénormalisés

On utilise les nombres dénormalisés pour coder les petites valeurs. Le champ exposant ne contient que des 0 (l’exposant réel est donc égal à –biais) et le champ mantisse est différent de 0.

## Exercices

1. Chacune des valeurs réelles suivantes est codée selon la norme IEEE754. Quelle est leur valeur en décimal ?
   1. C441000016
   2. 3EE0000016
   3. 1 10000001 010010011100010000000002
   4. 0 11001100 100001111101001000000002
2. Codez des valeurs réelles suivantes selon la norme IEEE754.
   1. 23,4375
   2. 0,99

## Et pour de vrai ?

Le 25 février 1991, pendant la Guerre du Golfe, une batterie américaine de missiles Patriot, à Dharan (Arabie Saoudite), a échoué dans l'interception d'un missile Scud irakien. Le Scud a frappé un baraquement de l'armée américaine et a tué 28 soldats. La commission d'enquête a conclu à un calcul incorrect du temps de parcours, dû à un problème d'arrondi.

En effet, les nombres étaient représentés en virgule fixe sur 24 bits avec 1 seul bit en partie entière et 23 bits pour la partie réelle, et l’horloge interne du missile Patriot mesure le temps en dixièmes de seconde (0.1s).

À votre avis quel était l’erreur de temps en secondes si on sait qu’au moment de l'attaque, la batterie de missile Patriot était allumée depuis environ 100 heures ?

**Réponse**  
Pour obtenir le temps en seconde, le système multiplie 0.1 par 10 en utilisant un registre de 24 bits en virgule fixe. Or 0.1 n’est pas un nombre codable de façon précise en binaire, donc a été arrondi : le registre de 24 bits contient 0,000110011001100110011002 et induit une erreur binaire de 0,0000000000000000000000011001100…2, soit approximativement 0,00000009510.

En multipliant cette quantité par 100 h (le temps écoulé entre la mise en marche du système et le lancement du missile Patriot), on obtient le décalage entre l’horloge interne de missile et le temps réel, soit : 0.000000095s × 100 × 3600 × 10 ≈ 0.34s

Or un missile Scud vole à la vitesse approximative de 1.676 m/s donc parcourt plus de 500 m en 0.34 s, ce qui le fait largement sortir de la zone d’acquisition de sa cible par le missile d’interception.

# Utilisation du binaire pour encoder des informations spécifiques

Le binaire est donc le langage de base d'un ordinateur. Nous avons vu comment les informations que nous manipulons généralement (nombres et caractères) sont codées pour être traitées et mémorisées par la machine. Mais la représentation binaire peut également servir à mémoriser des informations dont le codage aurait été établi par une équipe de programmeurs…

## Sorts

Par exemple, dans le cadre de la création de "Pathfinder : Kingmaker" (un RPG sur PC), afin de gagner de la place mémoire, les programmeurs décident de coder certaines informations en binaire.

Parmi ces informations, on retrouve les sorts disponibles. Chaque sort est décrit, en plus de son nom, par un ensemble de caractéristiques telles que l'école, le registre, le niveau du sort et les composantes nécessaires. Au lieu d'associer un libellé à chaque école (c'est-à-dire un tableau de *N* caractères, d'où *N* fois 8 bits) ou un numéro à chaque niveau de sort (c'est-à-dire un entier, d'où au moins16 bits), on désire les coder en binaire. Le gain de place est considérable !

Pour coder ces informations, il faut connaître le nombre de valeurs que chaque caractéristique prend afin de déterminer le nombre de bits qu'il est nécessaire de réserver pour son encodage.

Il y a 8 écoles, 24 registres, 10 niveaux et 4 composantes possibles. Un sort ne peut appartenir qu'à une seule école, un seul registre et est d'un niveau unique, mais les composantes peuvent être combinées.

De combien de bits a-t-on besoin pour coder l’école ?

De combien de bits a-t-on besoin pour coder le registre ?

De combien de bits a-t-on besoin pour coder le niveau ?

De combien de bits a-t-on besoin pour coder la composante ?

En C, de quel type, le plus adéquat, doit-on déclarer la variable qui permettrait de mémoriser ces informations ?

## Classe d’armure – examen janvier 2019

Dans un jeu de rôle, chaque personnage a une classe d’armure (CA). La classe d’armure représente la difficulté que les adversaires ont à toucher le personnage. Elle indique également la valeur que l’ennemi doit atteindre pour porter un coup infligeant des dégâts.

Dans cet exercice, la CA se calcule selon la formule suivante :

**CA** = 10 + [niveau/2]ENT + bonus d’armure

Le **niveau** est un entier compris entre 1 et 20.

Combien de bits sont nécessaires à l’encodage du niveau ?

Il y a différent type d’**armure**, chacune apportant un bonus spécifique, comme proposé ci-dessous :

|  |  |
| --- | --- |
| **Type** | **Bonus** |
| Aucune | +0 |
| Peau | +1 |
| Cuir | +2 |
| Matelassée | +3 |
| Maille | +4 |
| Plate | +5 |

Combien de bits sont nécessaires à l’encodage du bonus d’armure ?

En C, de quel type, le plus adéquat, doit-on déclarer la variable afin de mémoriser l’ensemble de ces informations sous la forme d’un mot binaire ?

## Cartes – examen août 2019

Dans un jeu de carte, chaque carte est représentée par un champ de bits. Pour pouvoir représenter une carte, il faut mémoriser trois informations :

* La couleur : rouge ou noir ;
* La famille : cœur, carreau, trèfle et pique ;
* La valeur : as, deux, trois, …, dix, valet, dame, roi.

Combien de bits sont nécessaires à l’encodage de la couleur ?

Combien de bits sont nécessaires à l’encodage de la famille ?

Combien de bits sont nécessaires à l’encodage de la valeur ?

En C, de quel type, le plus adéquat, doit-on déclarer la variable afin de mémoriser l’ensemble de ces informations sous la forme d’un mot binaire ?