

INF147 : DEUXIÈME COURS

Les types Entiers vs. les types Réels

Les Entiers - Peuvent représenter soit :

- Des valeurs entières signés (positif/négatif) ou non (**unsigned**)
- Des vecteurs de bits
- Des valeurs booléennes (Vrai/Faux), en C « 0 » est la valeur de FAUX et toute autre valeur est la valeur de VRAI
- Des caractères (codes ASCII dans un **char**)

La Représentation des Nombres Entiers en Mémoire :

En informatique, les nombres sont représentés en binaire (base 2) dans une série d'un ou plusieurs octets (paquet de 8 *bits*) consécutifs.

Voici la décomposition d'une valeur binaire (en base 2) :

$$\begin{aligned}\text{Eg. } 100101 &= 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 1 \times 32 + 0 \times 16 + 0 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 \\ &= 32 + 0 + 0 + 4 + 0 + 1 = 37\end{aligned}$$

En Visual C++, les valeurs de type **char** sont représentés en binaire par des paquets de 8 bits (1 octet), et les valeurs de type **int** et **long** sont représentés en binaire par des paquets de 32 bits (sur 4 octets). Si le dernier bit – le bit d'ordre 7 (**char**) ou d'ordre 31 (**int**, **long**) à l'extrême gauche – est allumé, ceci indique une valeur négative. La représentation des entiers négatifs est faite en utilisant la méthode du **complément à 2**.

$$\begin{aligned}\text{Eg. } 00100101 &\rightarrow 37 \text{ (sur 8 bits)} \\ 11011010 &\rightarrow \text{complément restreint (inversion des bits)} \\ 1101101\underline{1} &\rightarrow \text{complément à 2} = \text{complément restreint} + 1 = -37\end{aligned}$$

$$\begin{array}{rcl}\text{Si on fait la somme de } & 00100101 & (37) \\ & + 11011011 & (-37) \\ \text{..on obtient } & 100000000 & (\text{la somme} = 0 \text{ sur 8 bits})\end{array}$$

Si le type d'une variable est précédé du préfixe **unsigned**, ceci indique que TOUS les (8 ou 32) bits seront utilisés pour représenter la valeur et elle sera toujours positive.

Tailles - **char** < **short int** ≤ **int** ≤ **long**

Valeurs permises - Voir « **exemples01/limites types entiers.c** »

- Sur certains compilateurs, le type **int** est limité de -32768 à 32767 (sur 16 bits)

Opérations permises :

- sur un seul entier : - ++ -- & ! **sizeof()**
- entre 2 entiers : * / % + -
- opérations binaires : ~ << >> & ^ |
- opérations de comparaisons : < <= >= > == != && (ET) || (OU)

Les Réels – Représentent des valeurs « floating point »

Les valeurs réelles sont codées en deux parties alors qu'une partie des bits sera utilisée pour représenter la *mantisse* et l'autre portion des bits représentera l'*exposant* (eg. avec 0,12345678 x 10⁵, 12345678 est la mantisse et 5 l'exposant). En format IEEE-754 la mantisse est normalisée à une valeur entre [1..2[et l'exposant est une puissance de 2.

Tailles - **float** ≤ **double** (≤ **long double**)

Valeurs permises - Voir « **exemples01/limites types reels.c** »

Opérations permises :

- les mêmes que pour les entiers sauf : ~ << >> & ^ | %

Les DANGERS avec les types numériques :

- Division par zéro : ne jamais faire « x / 0 » ou « x % 0 »
- Entiers : comparer une valeur signée avec une valeur non-signée

Voir « **probleme unsigned.c** »

- Réels : erreurs de précisions avec (==, !=)

Voir « **erreur precision.c** »

Utilisation d'un « forçage de type ». Syntaxe : (**nouveau_type**) *expression*;

- Réels vs. Entiers avec un « forçage de type » (*typecast*)

Voir : « **erreur de calcul.c** »

et « **exemples de conversions.c** »

➔ Site web : voir présentation powerpoint « **Conversions.pps** »

Les Opérateurs en C

- Un opérateur renvoie toujours une valeur
- Impossible de changer la priorité ou l'associativité d'un opérateur - sauf avec ()
- Impossible de surcharger un opérateur (programmation objet seulement)

L'Opérateur d'Affectation :

L'opérateur d'affectation « = » peut être utilisé dans une expression ou une comparaison.

Ex 1. **int** a, b, c;

a = (b=2) + (c=3); est équivalent à :
$$\begin{cases} b = 2; \\ c = 3; \\ a = b + c; \end{cases}$$

Ex 2. Les Affectations Multiples (évaluées de gauche ← droite) :

a = b = c = 0; est équivalent à : a = (b = (c = 0));

Les Expressions d'Affectation :

Avec les opérateurs `+=`, `-=`, `*=`, `/=`, `%=`, etc.. ,

on peut remplacer une instruction : `variable = variable + expression;`

avec une expression d'affectation : `variable += expression;`

EXERCICE : Évaluez les expressions suivantes..

`int i=1, j=2, k=3, m=4, a=1;`

1) `i += j + k;` →

2) `j *= k = m + 5;` →

3) `a += a += a;` →

La Priorité :

Voici une liste (partielle) des priorités des opérateurs vus jusqu'à présent :

Priorité	Opérateurs	Ordre d'Évaluation (sans parenthèses)
1	(...)	gauche → droite
2	-x !x (avec <code>int x;</code>)	gauche ← droite
3	* / %	gauche → droite
4	+ -	gauche → droite
5	< > <= >=	gauche → droite
6	== !=	gauche → droite
7	&&	gauche → droite
8		gauche → droite
9	= += -= *= /= etc..	gauche ← droite

Les Opérateurs « ++ », « -- » :

Les opérateurs auto-incrément (`++`) et auto-décrément (`--`) ont des priorités variables. Tout dépend du placement de l'opérateur (*avant* la variable / *après* la variable) dans l'expression.

Avec ; `int x;`

`++x` est équivalent à : `x = x + 1;`

`--x` est équivalent à : `x = x - 1;`



Exécutée **avant** toute autre opérateur de la même expression

Ex. `z = ++x * y;` ⇔ `z = (++x) * y;` ⇔ `z = (x=x+1) * y;`

`x++` est équivalent à : `x = x + 1;`

`x--` est équivalent à : `x = x - 1;`



Exécutée **après** tous les autres opérateurs de la même expression

Ex. `int a, b, c=0;`
`a = ++c;`
`b = c++;`
`printf("%d %d %d\n", a, b, ++c);`

Sortie écran :

INCDEC.cpp: Donnez les valeurs que le programme suivant affichera à l'écran.

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>

int main(void)
{
    int a = 1, i = 1;

    a += i++;
    printf(" après a += i++ ... \n");
    printf(" a = %d i = %d \n\n", a, i);           → _____

    a=i=1;
    a += ++i;
    printf(" après a += ++i ... \n");
    printf(" a = %d i = %d \n\n", a, i);           → _____

    a=i=1;
    a += i--;
    printf(" après a += i-- ... \n");
    printf(" a = %d i = %d \n\n", a, i);           → _____

    a=i=1;
    a += --i;
    printf(" après a += --i ... \n");
    printf(" a = %d i = %d \n", a, i);             → _____

    return EXIT_SUCCESS;
}
```

Condition de boucle « Tant qu'une touche saisie n'est pas <ESC>, Faire ... » :

→ Quelle est l'erreur ? (avec : `#define CAR_ESC 27`)

`while (touche = _getch() != CAR_ESC) {...}` //avec `<conio.h>`

→ en cas de doute, ajouter des parenthèses!

L'Associativité :

Déterminer dans quel ordre les opérateurs de même priorité (ou niveau) seront évalués.

`val = a + b - c;` →

`x += y -= z;` →

`if (c1 || c2 && c3)` →

La structure itérative FOR

Syntaxe Générale : **for** (*expr1*; *expr2*; *expr3*) {
 instructions;
 }

expr1 → initialisation(s) à faire avant de commencer la boucle.

expr2 → condition pour exécuter la boucle (se terminera quand la condition == faux).

expr3 → instruction à exécuter à la fin de chaque itération de la boucle.

Exemple : Calcul d'un factoriel (N!) ..

 ..avec une boucle **for**

```
int i, N, facto=1;
```

```
... //demander la valeur de N
```

```
for (i=1; i<=N; i=i+1){  
    facto *= i;  
}
```

 ..avec une boucle **while**



```
int i, N, facto=1;
```

```
... //demander la valeur de N
```

```
i = 1; //expr1 du for  
while (i<=N){ //expr2 du for  
    facto *= i;  
    i = i + 1; //expr3 du for  
}
```

NOTEZ : dans un **for**, les *expr1*, *expr2*, *expr3* peuvent être vides!

Ex. **int** i, tot=0; //calculer la somme des valeurs de 1 à 10

```
i=1;   
for (i=1; i<=10; i=i+1) {  
    tot = tot + i;  
    i = i + 1;   
}
```

CALCULPI.cpp : Programme qui calcule une approximation de $Pi = \sqrt{6 \left(1 + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \dots \right)}$

```
#define _CRT_SECURE_NO_WARNINGS
```

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

```
#include<stdlib.h>
```

```
#include<math.h>
```

```
#define MAX 10000
```

```
int main(void)
```

```
{ double terme = 1, pi = 1.0;
```

```
int compteur;
```

```
for (compteur = 2; compteur <= MAX; ++compteur) {
```

```
    terme = 1.0 / (compteur*compteur); //expression de type double
```

```
    pi += terme;
```

```
}
```

```
pi = sqrt(6*pi);
```

```
printf("pi = %.14lf", pi);
```

```
return EXIT_SUCCESS;
```

```
}
```

Les Fonctions

En premier lieu, rappelons-nous quelques considérations générales sur les blocs en C :

- Les instructions en C doivent être dans des blocs.
- On peut faire des déclarations de variables locales dans tout bloc avant les instructions contenues dans ce bloc.
- Les blocs sont imbriquables « à volonté ».
- Les blocs les plus externes – un bloc qui n'est pas dans un bloc – sont différents, ils possèdent un identificateur -- un nom -- ce sont des fonctions.

Définition : une fonction est un sous-programme contenant une ou plusieurs instructions qui réalisent une tâche unique.

Quel est l'intérêt d'avoir des fonctions dans un programme ?

- Réutilisation de code (au lieu de copier-coller).
- Décomposer un problème complexe en petits problèmes simples.
- Augmenter la lisibilité de votre code.
- Aide le processus de « *debuggage* ».

Trois règles fondamentales sur les fonctions

Règle #1 : La déclaration d'une fonction

Comme tout identificateur doit être déclaré avant d'être utilisé dans une expression Une fonction doit être déclarée avec un « **prototype** » dont voici la syntaxe :

```
/*=====*/
/*  ...le commentaire d'entête à la déclaration...  */
/*=====*/
type_retour  identificateur (liste de paramètres formels);
```

- **type_retour** indique un type connu du compilateur auquel appartient la valeur retournée comme résultat d'un appel de la fonction (le type de la réponse).
- **identificateur** sera le nom d'usage normal lors d'un appel de la fonction.
- On aura le même nombre de *paramètres formels* que de *paramètres actuels*. Un paramètre *formel*, c'est la déclaration d'une variable locale à la fonction qui sera initialisée par son paramètre *actuel* correspondant lors d'un appel.
- Si la liste de paramètres formels est vide, on l'indique avec le mot réservé **void**.
- Si la fonction ne retourne rien – comme une fonction qui va seulement afficher des résultats – on l'indique avec le mot réservé **void** comme type de retour.
- Le terme **prototype** fait référence à l'action du compilateur qui chaque fois qu'il rencontre l'appel d'une fonction va vérifier la compatibilité du type des paramètres actuels avec le type des paramètres formels correspondants.
- Une déclaration de fonction se termine toujours avec un « ; » (pas de bloc ici !).

Exemples de déclarations de fonctions :

```
int  pgcd(int n1, int n2); //Déclaration d'une fonction plus grand commun diviseur
double  sin(double);      //La déclaration de la fonction sinus dans <math.h>
```

➔ Remarquez ici que le nom du paramètre formel n'a pas été spécifié. Dans un prototype, les noms des paramètres formels ne sont pas obligatoires !

Les commentaires à la déclaration :

- ❖ **Ce sont les commentaires les plus importants.** Ils s'adressent à tout « client » éventuel de la fonction et lui expliquent le service offert par cette fonction ainsi que les limites normales prévues à son utilisation.
- ❖ Toute déclaration de fonction suit un modèle que vous devez obligatoirement suivre :
 1. Un séparateur seul ou avec l'identificateur en majuscules à l'intérieur.
 2. Une description succincte du but visé par cette fonction.
 3. Une description du rôle donné à chaque paramètre formel reçu par la fonction.
 4. Une description des valeurs possibles au retour.
 5. Les spécifications qui donnent réponse aux interrogations légitimes de l'utilisateur.
 6. La déclaration proprement dite (le *prototype*).

```
/*=====*/
/* X_ALA_N                                     */
/* Algorithme très rapide pour obtenir         */
/* une puissance entière d'un réel quelconque. */
/*                                             */
/* PARAMETRES : x le nombre réel et la puissance entière */
/* RETOUR : la puissance calculée (type double) */
/*                                             */
/* SPÉCIFICATIONS :                           */
/* On traite correctement le signe de la puissance. */
/* Le résultat d'un appel avec 0 à une puissance négative est */
/* imprévisible.                               */
double x_ala_n(double x, int puissance);
```

Règle #2 : La définition d'une fonction

C'est le code de la fonction. Une fonction doit être définie avec la syntaxe suivante :

```
type_retour  identificateur(liste de paramètres formels) {
    déclaration des variables locales;

    toutes les instructions nécessaires;

    return (expression-réponse); //si type_retour ≠ void
}
```

- Où on répète la déclaration de la fonction **sans** le point virgule à la fin.
- On y trouve le bloc de code à exécuter pour obtenir le résultat voulu. Dans le bloc de code, on retrouve **au moins une** instruction comportant le mot réservé **return** permettant de terminer l'exécution de la fonction et de retourner la valeur de l'expression qui suit immédiatement ce return. Cette valeur remplacera littéralement l'appel de la fonction.
- Si le **type_retour** de la fonction est **void**, l'instruction **return**; n'est PAS obligatoire et cette fonction n'aura pas de type.

Voici la définition complète de la fonction « `x_ala_n()` » qui calcule ($X^{\text{puissance}}$) :

```
/*=====*/
/* Permet d'obtenir une puissance entière d'un réel quelconque. */
double x_ala_n(double x, int puissance)
{
    double resul = 1;    /* accumulateur de la puissance à rendre */
    int positif = 1;     /* on y gardera le signe de la puissance */

    if ( puissance < 0 ) {
        positif = 0;
        puissance *= -1;
    }

    /* maintenant une descente à 0 pour la puissance positive */
    while ( puissance ) {
        /* si le bit de droite de puissance est = 1 (impair) */
        /* on accumule la puissance de x actuelle */
        if (puissance % 2 == 1) {
            resul *= x;
        }
        x *= x;           /* on passe de (x^k) à (x^2k) */
        puissance /= 2;   /* on coupe le bit de droite de puissance */
    }

    /* on prend l'inverse si la puissance était négative */
    if (!positif)
        return 1/resul;
    return resul;
}
```

Commentaires dans la définition :

- ❖ **Ce sont les commentaires du maintien ou de l'amélioration de l'algorithme.** Ils ne s'adressent absolument pas au client. Êtes-vous intéressé à savoir comment le calcul du sinus se fait dans la librairie `<math.h>`? On décrit dans ce commentaire la façon dont on parvient au résultat de l'algorithme. C'est ici qu'apparaît les commentaires qui expliquent les expressions qui forment le code. (ces commentaires n'ont rien à voir avec ceux de la déclaration de fonction).

Qu'est-ce qu'une variable locale ?

- ❖ Une variable locale à une fonction est déclarée dans le bloc de la fonction.
- ❖ Toute variable locale possède une place réservée dans la mémoire de la machine que durant l'exécution de cette fonction. Les variables locales aux blocs-fonctions seront automatiquement détruites à la sortie du bloc de sa définition. Ce sont des variables temporaires.
- ❖ Une variable locale n'utilise pas nécessairement la même position dans la mémoire RAM d'appel en appel de la fonction. La fonction ne peut évidemment pas mémoriser la valeur prise lors d'un appel antérieur par une de ses variables locales. On dit qu'une fonction n'a pas mémoire de ses exécutions antérieures.

Mise en garde importante :

Une variable globale, c'est une variable définie hors de tout bloc et qui est accessible dans toutes les unités de compilation d'un projet. **L'usage des variables globales en programmation moderne et dans ce cours est interdit !**

Règle #3 : L'appel d'une fonction

Dans une expression, un identificateur suivi d'une parenthèse ouvrante indique l'appel de la fonction portant ce nom. Les variables ou expressions séparées par des virgules à l'intérieur des parenthèses sont **évaluées** et forment les **paramètres actuels** de l'appel. L'appel d'une fonction est considéré comme une expression.

Exemples d'appels :

```
expo = x_ala_n(2, 3); //avec: double expo;
```

➔ Un appel correct de la fonction « x_ala_n » présentée plus haut.

```
divi = pgcd(1236, a); //avec: int divi, a;
```

➔ La fonction « pgcd » retourne le plus grand commun diviseur de deux entiers.

➤ En C, toute valeur transmise à une fonction est passé par valeur (par copie).

➤ L'ordre des paramètres effectifs de gauche à droite est important.

➤ Les noms des paramètres actuels ne sont pas importants, mais leurs types, oui !

Passage par valeur.c :

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdlib.h>
#include <stdio.h>

void fonction_quelconque(int x); //déclaration (ou prototype)

/*****
int main(void)
{ int x = 0;

    printf("main(): Adresse de x AVANT l'appel : %x\n", &x); //"%x" = hexadécimal
    printf("main(): contenu de x AVANT l'appel : %i\n\n", x);

    fonction_quelconque(x); //appel de la fonction
```

```

printf("\nmain(): Adresse de x APRES l'appel : %x\n", &x); //"%x" = hexadecimal
printf("main() : contenu de x APRES l'appel : %i\n", x);

return EXIT_SUCCESS;
}

/*****
void fonction_quelconque(int x) //définition de la fonction
{
    printf(" fonction_quelconque: adresse de x: %x\n", &x); //"%x" = hexadecimal
    printf(" fonction_quelconque: contenu du x au debut de la fonction: %i\n", x);
    ++x;
    printf(" fonction_quelconque: contenu du x a la fin de la fonction: %i\n", x);
}

```

Les Opérateurs de Bits en C

Description des Différents Opérateurs de Bits en C :

OPERATEUR DE BITS UNAIRE (une seule opérande) :

~ : Operateur d'inversion des bits (complément restreint).

~ 01011101 → 10100010

- : Opérateur de négation (complément vrai).

- a == (~a + 1) (avec char a;)

- 00010111 → 11101001

Caractéristique du complément vrai :

$$\begin{array}{r}
 a \rightarrow 00010111 \quad (= 23) \\
 + (-a) \rightarrow 11101001 \quad (= -23) \\
 \hline
 = 00000000
 \end{array}$$

NOTEZ : En C, avec des char (ou int) les 7 premier (ou 31 premiers) bits sont utilisés pour indiquer la valeur numérique. Le dernier bit (à gauche) indiquera le signe (+/-). Avec les types unsigned, tous les bits sont utilisés pour construire la valeur numérique.

OPERATEURS DE BITS BINAIRES (deux opérandes) :

& : ET

| : OU

^ : OU exclusif

A	B	A & B	A B	A ^ B
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	1	0

& : Operateur ET.

01011101 & 11101001 → 01001001

| : Operateur OU.

01011101 | 11101001 → 11111101

^ : Operateur OU EXCLUSIF (xor).

01011101 ^ 11101001 → 10110100

>>: Operateur de décalage vers la droite (divisions par 2).

a = 00011010 (=26)

a >> 3 = 00000011 (= 26 / 8 == 3)

<<: Operateur de décalage vers la gauche (multiplications par 2).

a = 00011010 (=26)

a << 2 = 01101000 (= 26 * 4 == 104)

Voici les fonctions de bits offertes dans le programme « **exemple_bits.c** » :

```
/* get_bit(): un test qui dit si un (and) entre la valeur reçue
   et un train de bits tout à 0 sauf un bit 1 en position voulue,
   obtenu avec l'opérateur <<, est différent de 0. */
int get_bit(unsigned int nombre, int ordre) {
    return (nombre & (1 << ordre)) != 0;
}

/* set_bit(): un (or) suffit entre la valeur reçue et un train de bits
   tout à 0 sauf pour un bit 1 en position voulue, obtenu avec l'opérateur <<. */
unsigned int set_bit(unsigned int nombre, int ordre) {
    unsigned int masque = (1 << ordre);
    unsigned int nombre_ou_masque = (nombre | masque);
    return nombre_ou_masque;
}

/* clear_bit(): un (and) suffit entre la valeur reçue et un train de
   bits tout à 1 sauf un bit 0 à la position voulue, obtenu avec les opérateurs <<, ~. */
unsigned int clear_bit(unsigned int nombre, int ordre) {
    unsigned int masque_inv = ~(1 << ordre);
    unsigned int nombre_et_masque = (nombre & masque_inv);
    return nombre_et_masque;
}

/* flip_bit(): réutilisation des deux fonctions précédentes
   si le bit est 1, on retourne clear_bit(), sinon, on retourne set_bit(). */
unsigned int flip_bit(unsigned int nombre, int ordre) {
    return get_bit(nombre, ordre) ? clear_bit(nombre, ordre)
    : set_bit(nombre, ordre);
}

/* voir_bits(): avec réutilisation de get_bit()
   on affiche le résultat de get_bit() pour chaque position de 31 à 0. */
void voir_bits(unsigned int nombre) {
    // pour respecter l'ordre naturel des bits on les
    // affiche du plus représentatif en descendant (ordre 31 à 0)
    for (int i = INT_BIT - 1; i > -1; i -= 1)
        printf("%d", get_bit(nombre, i));
    printf("\n"); // affiche un ENTER après la fin de la boucle for()
}
```