Méthodes numériques et langage C

Rappels et définitions

R. Flamary

3 septembre 2018

Méthodes numériques et langage C

Objectifs du Cours

- Utilisation pratique du langage C.
- Introduction aux méthodes numériques.
- Implémenter des méthodes complexes à partir d'opérations de base.

Méthodes numériques

- Séries entières.
- Équations non linéaires.
- Algèbre linéaire.
- Intégration et dérivation numérique.
- Interpolation polynomiale.

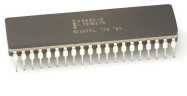
Plan du cours

Représentation des valeurs numériques

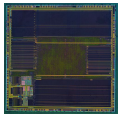
Principe

- Les nombres sont représentés en mémoire sur des bits.
- Le nombre de bits dédiés à chaque nombre définit ses limites.
- Les limites sont de natures différentes selon que l'on encode des nombres entiers ou réels.

Architectures





4 bits Intel 4004 (1971)

8 bits MOS Technology 6502 (NES 1985), Sharp x80 (GameBoy 1989)

16 bits Intel 8086 (1978), 65C816 (Super Nintendo 1990)

32 bits Intel x86 (1985), AMD Athlon K5 (1995), R3000A (Playstation 1994)

64 bits Athlon 6 (2003), Pentium 4 (2004), ARMv8-A (2011)

Nombres entiers

Entiers signés

Stocké en mémoire en base 2 :

$$x = s\sum_{i=0}^{p-2} d_i 2^i$$

- s signe (en binaire le signe moins est encodé par 1)
- p entier est le nombre de bits (souvent puissance de 2)
- $d_i \forall i \in 0, \ldots, p-2$ est un nombre binaire (0 ou 1).
- Représentation en mémoire (16 bits) :

signe	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
s	d_{14}	d_{13}	d_{12}	d_{11}	d_{10}	d_9	d_8	d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0

Exercice

Donner la représentation binaire des nombres suivants :

x	signe	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
5																
-10																
37																

Nombres entiers en C

Entiers signés

Bits	Usage	Intervalle	Déc.
8	**char***, **int8_t**	-128 à 127	3
16	**int*** (min), **int16_t**	-32 768 à 32 767	5
32	**int***, **long*** (min) , **int32_t**	-2 147 483 648 à 2 147 483 647	10
64	**long***, **long long**, **int64_t**	-9 223 372 036 854 775 808 à 9 223 372 036 854 775 807	19

- * Le nombre de bit pour **char**, **short**, **int**, **long** dépend du compilateur C et du système.
- Les types **int*_t** indépendant du système ont été définis dans le standard C99.
- Ces types nécessite la déclaration du header `<stdint.h>`

Entiers non signés

- Même intervalle mais commençant à 0.
- Déclarés en précédant l'identifieur de **unsigned** ou comme **uint*_t**.

Exemple de déclaration

```
1 #include <stdint.h>
2 int i;
3 int64_t j=2014;
4 unsigned long k;
```


Limitation des entiers en C

Taille et intervalle

- Entier sur p bits
- Signé entre -2^{p-1} et $2^{p-1}-1$
- Non signé entre 0 et 2^p-1
- Représentation exacte et règles arithmétiques respectées dans l'intervalle.

Code source

```
1  int8_t i=127;
2  int32_t j=127;
3  printf("i=%d\n",i);
4  i=i+1;
5  printf("i+1=%d\n",i);
6  printf("j=%d\n",j);
7  j=j+1;
8  printf("j+1=%d\n",j);
9  j=j+3000000000 ;
10 printf("j+3000000001=%d\n",j);
```

Sortie

```
1 $./ex_int
2 i=127
3 i+1=-128
4 j=127
5 j+1=128
6 j+3000000001=-1294967168
```


Nombres à virgule flottante

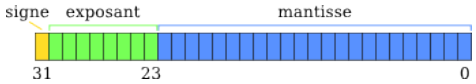
Virgule flottante

Un flottant est encodé comme :

$$x = sm2^{e-d}$$

- s signe
- $1 \leq m < 2$ mantisse (m)
- e exposant, d décalage de l'exposant

Norme IEEE 754



- Flottant 32 bits (s : 1 bits, m : 23 bits, e : 8 bits, d=$2^{8-1}-1$=127)
  - Chiffres significatifs : $2^{-23} \approx 1.10^{-7}$ → 7 chiffres
  - Min/max en valeur absolue : 1.10^{-38}, $3.4.10^{38}$
- Flottant 64 bits (s : 1 bits, m : 52 bits, e : 11 bits, d=$2^{11-1}-1$=1023)
  - Chiffres significatifs : $2^{-52} \approx 2.2.10^{-16}$ → 15 chiffres
  - Min/max en valeur absolue : $2.22.10^{-308}$, $1.79.10^{308}$

Opérations en virgule flottante

Soient x_1 et x_2 deux nombres flottants.

Addition

- L'addition nécessite que les deux nombres aient le même exposant.
- Si $e_1 = e_2$, on additionne les mantisses.
- Si $e_1 \neq e_2$, on décale la mantisse du nombre le plus petit pour qu'ils aient le même exposant, ensuite on additionne les mantisses.
- Si après l'addition la mantisse $m > 2$, on la décale vers la droite et on ajoute 1 à l'exposant.
- Attention si les nombres sont d'un ordre de grandeur très différents !

Multiplication

$$x_1 x_2 = m_1 m_2 2^{e_1+e_2-2d}$$

- On effectue le produit des mantisses et la somme des exposants.
- si m sort de l'intervalle, on décale la mantisse et change l'exposant.

Flottants en C

Bits	Usage	absolute max/min	Déc.
32	**float***, **float_t**	1.10^{-38}, $3.4.10^{38}$	7
64	**double***, **double_t**	$2.22.10^{-308}$, $1.79.10^{308}$	15

- De la même manière que pour les entiers, les types **float** et **double** dépendent du compilateur et du système.
- Les types **float_t** et **double_t** sont définis dans le standard C99 (entête <math.h>).
- La précision numérique et les erreurs d'arrondis doivent être pris en compte dans le codage.

Code source

```
1  float_t x=1.;
2  printf("x=%2.8f\n",x);
3  x=x+1e-8;
4  printf("x+1e-8=%2.8f\n",x);
5  x=x+1e50;
6  printf("x+1e50=%2.8f\n",x);
7  double_t y=1.;
8  printf("y=%2.8f\n",y);
9  y=y+1e-8;
10 printf("y+1e-8=%2.15f\n",y);
11 y=y+1e50;
12 printf("y+1e50=%2.15e\n",y);
```

Sortie

```
1 $./ex_float
2 x=1.00000000
3 x+1e-8=1.00000000
4 x+1e50=inf
5 y=1.00000000
6 y+1e-8=1.000000010000000
7 y+1e50=1.000000000000000e+50
```


Exemples d'encodage

Code source

```
1 int nb=7;
2 float lst[] = { 3.14,0.125, 9.3, 0,-0.,atof("NaN"), atof("inf") };
3
4 for (int i=0; i<nb;i++)
5 {
6   printf("%2.6f -> ",lst[i]);
7   printBits_IEEE754(sz, lst+i);
8 };
```

Sortie

```
1 $./ex_float_enc
2 3.140000 -> 0 10000000 10010001111010111000011
3 0.125000 -> 0 01111100 00000000000000000000000
4 9.300000 -> 0 10000010 00101001100110011001101
5 0.000000 -> 0 00000000 00000000000000000000000
6 -0.000000 -> 1 00000000 00000000000000000000000
7 nan -> 0 11111111 10000000000000000000000
8 inf -> 0 11111111 00000000000000000000000
```


Les pointeurs

Définition

- Un pointeur est une variable dont la valeur (un entier) est l'adresse mémoire d'une autre variable (ou d'une série de valeurs).
- L'accès à la valeur pointée par l'adresse est fait avec l'opérateur *****.
- L'accès à l'adresse d'une variable se fait par l'intermédiaire de l'opérateur **&**.
- Les pointeurs peuvent être utilisés pour tout type d'objets.
- Ils doivent toujours pointer sur une zone mémoire allouée (segmentation fault).

Code source

```
1 int i=10;
2 int *p=NULL;
3 printf("i=%d, p=%u\n",i,p);
4 p=&i;
5 printf("i=%d, p=%u\n",i,p);
6 *p=5;
7 printf("i=%d, p=%u\n",i,p);
8 p[0]=4;
9 printf("i=%d, p=%u\n",i,p);
```

Sortie

```
1 $./ex_ptr
2 i=10, p=0
3 i=10, p=1910622300
4 i=5, p=1910622300
5 i=4, p=1910622300
```

Chaines de caractères

- Ce sont des pointeurs vers une suite de caractères imprimables (type **char**).
- Nombreuses fonctions de gestion définies dans `<string.h>`.
- Se termine par le caractère \0. Elle a donc un octet de plus que la liste de caractères imprimables.
- Toujours possible de déterminer sa taille avec la fonction **strlen()**.
- Une chaîne de caractères est délimitée par " en C.
- Le caractère \n représente un retour à la ligne, \t une tabulation.
- On accède aux caractères avec l'opérateur **[]** (indexage à 0).

Code source

```
1 char *txt="Premier texte\n";
2 char *txt2;
3 int nb=26;
4 printf("%s",txt);
5 txt2=malloc(nb*sizeof(char));
6 for (int i=0;i<nb;i++)
7   txt2[i]='a'+i;
8 printf("%s",txt2);
```

Sortie

```
1 $./ex_str
2 Premier texte
3 abcdefghijklmnopqrstuvwxyz
```


Tableaux 1D et mémoire

Les tableaux utilisent les pointeurs pour accéder à une suite de valeurs voisines en mémoire. L'opérateur utilisé pour la déclaration et l'accès aux données est **[]**.

Allocation et déclaration

- Taille donnée : `type_t p[n];`
  Initialise un tableau vers n cases mémoires du type **type_t**
- Par valeur : `type_t p[]={val1,val2,...,valn};`
  Initialise un tableau vers n cases mémoire du type **type_t** avec les valeurs données dans {...}

Code source

```
1 int p1[10];
2 int p2[]={1,2,3,4,5,6,7,8,9,10};
3 printf("p1->");
4 for (int i=0;i<10;i++)
5 {
6   p1[i]=1+i*i;
7   printf("%d,",p1[i]);
8 }
9 printf("\np2->") ;
10 for (int i=0;i<10;i++)
11   printf("%d,",p2[i]);
```

Sortie

```
1 $./ex_tab1d
2 p1->1,2,5,10,17,26,37,50,65,82,
3 p2->1,2,3,4,5,6,7,8,9,10,
```


Tableaux N-dimensionnels

- Les tableaux peuvent être étendu à N dimensions.
- Les déclarations de font de manière similaire.

Code source

```
1 int m1[3][3];
2 int m2[3][3]={1,2,3,4,5,6,7,8,9};
3 printf("m1:\n");
4 for (int i=0;i<3;i++)
5 {
6   for (int j=0;j<3;j++)
7     {
8       m1[i][j]=i+j;
9       printf("%d,",m1[i][j]);
10    };
11  printf("\n");
12 }
13 printf("\nm2:\n") ;
14 for (int i=0;i<3;i++)
15 {
16   for (int j=0;j<3;j++)
17     printf("%d,",m2[i][j]);
18   printf("\n");
19 }
```

Sortie

```
1 $./ex_tab2d
2 m1:
3 0,1,2,
4 1,2,3,
5 2,3,4,
6
7 m2:
8 1,2,3,
9 4,5,6,
10 7,8,9,
```


Tableaux et pointeurs

- Les tableaux peuvent être vus comme des pointeurs à l'adresse fixée.
- Un pointeur peut récupérer l'adresse d'un tableau et avoir accès aux données mais doit utiliser un indexage linéaire (warning compilateur).

Code source

```
1 int m[3][3]={1,2,3,4,5,6,7,8,9};
2 int *p;
3 p=m;
4 printf("m[0][0]=%d\n",m[0][0]);
5 printf("*p=%d\n",*p);
6 printf("p[0]=%d\n",p[0]);
7 printf("m[2][1]=%d\n",m[2][1]);
8 printf("*(p+2*3+1)=%d\n",*(p+2*3+1));
9 printf("p[2*3+1]=%d\n",p[2*3+1]);
```

Sortie

```
1 $./ex_tabptr
2 m[0][0]=1
3 *p=1
4 p[0]=1
5 m[2][1]=8
6 *(p+2*3+1)=8
7 p[2*3+1]=8
```

Structures et définition de type

- `struct` permet de définir une structure contennant plusieurs variables.
- Les valeurs de ces variables sont obtenues avec l'opérateur .
- `typedef` permet de définir un type (évite de répéter le struct).
- Une structure peut être adressée avec un pointeur.

Code source

```
1   struct etud
2   {
3       char nom[50];
4       float note;
5   };
6
7   typedef struct etud Etudiant;
8
9   Etudiant e;
10  strcpy(e.nom,"Bibi");
11  e.note=12.5;
12
13  printf("Etudiant:\nNom: %s\nNote:
        %f",e.nom,e.note);
```

Sortie

```
1 $./ex_struct
2 Etudiant:
3  Nom: Bibi
4 Note: 12.500000
```


Structure d'un programme C

Code source

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5   printf("Hello, world!\n");
6   return 0;
7 }
```

Sortie

```
1 $./ex_hello
2 Hello, world!
```

- `#include <stdio.h>`, en-tête standard contenant les déclarations des fonctions d'entrées-sorties (dont `printf`).
- `int` est le type renvoyé par la fonction `main`.
- `main` est le nom de la fonction principale, aussi appelée point d'entrée du programme.
- Les parenthèses après main indiquent que main est une fonction.
- Les accolades { et } entourent les instructions constituant le corps de la fonction `main`.
- Un point-virgule ; termine l'instruction.

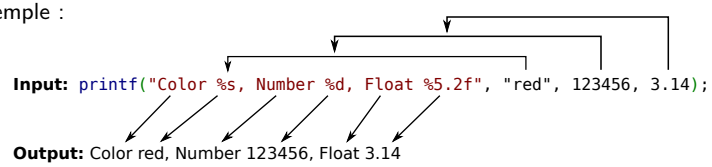
Source: Wikipedia

Entrée/Sortie écran

`printf`

- Imprime une chaîne de caractères à l'écran (terminal).
- Chaînes de caractères formatées en insérant des valeurs contenues dans des variables.
- Exemple :

Type	Format
int, long	%d,%ld
unsigned int	%u
float,double	%f, %lf
char	%c
char*	%s



`scanf`

- Permet de demander à l'utilisateur du programme d'entrée des données.
- Même format que `printf`, passage de la variable à modifier par pointeur.
- Préférer le passage d'information par ligne de commande.

Entrée/Sortie par ligne de commande

- Des paramètres (séparés par des espaces) peuvent être donnés lors de l'exécution du programme.
- Les paramètres donnés en ligne de commande sont passés à la fonction `main`.
- `argc` est un entier donnant le nombre de paramètres et `argv` est un tableau de pointeurs vers ces paramètres.
- Le premier paramètre (index 0) est toujours le nom du programme.

Code source

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[])
4 {
5   printf("Nombre de params: %d\n",
        argc);
6   for (int i=0;i<argc;i++)
7     printf("Param. %d: %s\n",i,argv[i
        ]);
8   return 0;
9 }
```

Sortie

```
1 $./ex_command
2 Nombre de params: 1
3 Param. 0: ./ex_command
4 $./ex_command 1 "Hello world"
5 Nombre de params: 3
6 Param. 0: ./ex_command
7 Param. 1: 1
8 Param. 2: Hello world
```

Entrée/Sortie fichier

Écriture de fichier avec `fputs`

- Le type **FILE** permet de gérér la lecture/écriture dans un fichier à l'aide des fonctions `fopen, fputs, fgets, fscanf, fclose` .
- Un fichier doit être ouvert avec `fopen` et fermé avec `fclose` pour être écrit sur le disque.

Code source

```
1 char txt[]="Texte de test\nSur deux
      lignes";
2 char fname[]="fichier_exemple.txt";
3 FILE *fp;
4 fp = fopen(fname, "w+");
5 fputs(txt, fp);
6 fclose(fp);
```

Sortie

```
1 $./ex_file
2 $cat fichier_example.txt
3 Texte de test
4 Sur deux lignes
```

Lecture de fichier

- `fgets` permet de lire une chaine de caractères de taille connue à partir d'un pointeur fichier.
- `fscanf` permet de lire une chaine de caractères formatée.

Fonctions en C

- Les fonctions permettent d'exécuter simplement une suite d'instructions (avec des paramètres).
- Elles sont définies par leur nom, liste de paramètres d'entrée et type de sortie.
- Si on veut modifier plusieurs variables dans une fonction on passe en paramètre des pointeurs vers les variables à modifier.

Code source

```
1 int carre(int i)
2 {
3   return i*i;
4 }
5
6 int main()
7 {
8 int nb=5;
9 for (int i=0;i<nb;i++)
10   printf("i=%d, carre(i)=%d\n",i,
       carre(i));
11 return 0;
12 }
```

Sortie

```
1 $./ex_func
2 i=0, carre(i)=0
3 i=1, carre(i)=1
4 i=2, carre(i)=4
5 i=3, carre(i)=9
6 i=4, carre(i)=16
```


Entête et définition de fonction

- Une fonction peut être déclarée sans sa définition. Ceci permet son utilisation sans avoir sa définition (dans un autre fichier *.c par exemple)
- Il faut un point virgule à la fin de la déclaration (pas de la définition).
- Les directives compilateur `#include "fichier.h"` permettent de déclarer les fonctions pour une utilisation dans le code (parfois utilisé pour la définition des fonctions).
- La définition des fonctions est souvent faite dans un fichier `fichier.c` associé au fichier `fichier.h`.

Fichier `carre.h`

```
1 #ifndef CARRE_H
2 #define CARRE_H
3
4 int carre(int i);
5
6 #endif
```

Fichier `carre.c`

```
1 #include "carre.h"
2
3 int carre(int i)
4 {
5   return i*i;
6 }
```

Fichier `ex_funch.c`

```
1 #include <stdio.h>
2 #include "carre.h"
3
4 int main()
5 {
6 int nb=5;
7 for (int i=0;i<nb;i++)
8   printf("i=%d, carre(i)
        =%d\n",i,carre(i))
        ;
9 return 0;
10 }
```

Sortie

```
1 $gcc ex_funch.c carre
      .c -o ex_funch -
      std=c99
2 $./ex_funch
3 i=0, carre(i)=0
4 i=1, carre(i)=1
5 i=2, carre(i)=4
6 i=3, carre(i)=9
7 i=4, carre(i)=16
```


Exercice 1

Coder une fonction qui retourne le cube d'un nombre flottant :

$$cube(x) = x^3$$

Analyse du problème

- Nom : cube
- Entrée :
- Sortie :
- Mise en oeuvre :

Solution

Pointeur de fonction en C

- Utiliser une variable comme une fonction.
- Utile pour résoudre des équations ou en intégration numérique.
- On utilise des pointeurs de fonction.
- Un pointeur de fonction est déclaré par :

```
type_ret (*nomfunc)(params);
```

Code source

```
1  double carre(double x)
2  {return x*x;}
3
4  int main()
5  {
6  double (*f1)(double),(*f2)(double);
7  f1=carre;
8  f2=sin;
9  printf("f1(2)=%f\n", f1(2));
10 printf("f2(2)=%f\n", f2(2));
11 }
```

Sortie

```
1 $./ex_pfunc0
2 f1(2)=4.000000
3 f2(2)=0.909297
```


Bibliothèque standard du C

- Ce sont un ensemble de fonctions et de fichiers d'entête qui sont présents sur tous les systèmes.
- Ils forment la bibliothèque standard ANSI C.
- Liste des entêtes/fonctions les plus utiles, mais documentation détaillée sur le web.

Principaux fichiers d'entête :

`<stdio.h>`	Entrées/Sorties standards (terminal et fichiers).
`<stdlib.h>`	Allocation mémoire et générateur aléatoire.
`<strings.h>`	Gestion des chaînes de caractères.
`<math.h>`	Fonctions mathématiques courantes.
`<float.h>`	Constantes pour les nombres à virgule flottante (ϵ, min, max).
`<limits.h>`	Constantes pour les nombres entiers (min, max).
`<complex.h>`	Nouveaux types pour les nombres complexes (C99).
`<time.h>`	Temps et affichage de dates.

Bibliothèque standard : `<stdio.h>`

Fonctions d'entrée/sortie de base :

Entrée/sortie terminal

printf	Écrire une chaîne de caractères dans le terminal.
scanf	Lire une chaîne de caractères sur le terminal.

Entrée/sortie fichier

fopen	Ouverture de fichier.
fclose	Fermeture de fichier et écriture définitive.
fprintf	Écrire une chaîne de caractères dans un fichier.
fscanf	Lire une chaîne de caractères dans un fichier.

Code source

```
1 printf("i=%d, j=%u\n",-10,3);
2 printf("x=%1.3f ou x=%e\n"
      ,3.14159,3.14159);
3 printf("Nom: %s\n","bibi");
```

Sortie

```
1 $./ex_printf
2 i=-10, j=3
3 x=3.142 ou x=3.141590e+00
4 Nom: bibi
```


Bibliothèque standard : `<string.h>`

Fonctions de gestion de chaînes de caractères :

Gestion de chaînes de caractères

strcpy	Copie de chaîne.
strcat	Concaténation de deux chaînes.
strlen	Longueur d'une chaîne.

Conversion de chaines de caractères

atof	Conversion de chaîne en flottant.
atoi	Conversion de chaîne en entier.

Code source

```
1 char txt[80]="Prenom, ";
2 char txt2[]="Nom";
3 strcat(txt,txt2);
4 printf("%s",txt);
```

Sortie

```
1 $./ex_str2
2 Prenom, Nom
```

Bibliothèque standard : <math.h>

Fonctions mathématiques de base :

Nombres entiers

abs Valeur absolue.

div Division entière (quotient et reste).

Nombres à virgule flottante

round Arrondi d'un flottant.

fabs Valeur absolue.

exp Exponentielle.

log, log10 Logarithme népérien et décimal.

pow Puissance.

sqrt Racine carrée.

sin, cos, tan Fonctions trigonométriques.

Code source

```
printf("round(3.14)=%f\n",round(3.14)
    );
printf("log(2)=%f\n",log(2.0));
printf("exp(1)=%f\n",exp(1.0));
printf("sqrt(2)=%f\n",sqrt(2.));
printf("sin(1)=%f\n",sin(1.));
```

Sortie

```
$./ex_math
round(3.14)=3.000000
log(2)=0.693147
exp(1)=2.718282
sqrt(2)=1.414214
sin(1)=0.841471
```


Algorithmique

- Un algorithme est une suite finie d'instructions permettant de donner la réponse à un problème.
- Il existe différentes représentations d'algorithmes (schéma, texte), dans ce cours nous donnerons des algorithmes sous forme de fonction C.
- Pour résoudre un problème complexe on cherche à découper ce problèmes en plusieurs sous-problèmes plus simples.
- Un algorithme est défini par ses entrées, sorties et son nom.
- Les algorithmes utilisent typiquement des opérateurs de branchement (condition) et des boucles.

Fichier `carre.c`

```
#include "carre.h"

int carre(int i)
{
  return i*i;
}
```

Algorithme correspondant

- Nom : **carre**
- Entrée : entier **i**
- Sortie : entier **s** (valeur $\mathbf{i}^2$)

Début

s ← **i*i**

Fin

Opérateurs logiques en C

Ces opérateurs sont utilisés pour tester une condition booléenne (branchement ou boucle).

Comparaisons

`==, !=` Égalité, inégalité.

`<,>` Stricte comparaison.

`<=, >=` Comparaison avec égalité

Opérations booléennes

`!` Négation booléenne.

`&` ET booléen.

`|` OU booléen.

`^` OU exclusif booléen.

Exercice

Donner la valeur binaire retournée par ces expressions :

- `(1>=0)&!(10==10)` :
- `4==9/2` :
- `(1<=0)|!(10)` :
- `i=3.; (i<=4)&(i>=1)` :
- `i=3.; (i>=4)|(i<=1)` :

Condition `if`

- Exécute une série d'instructions si une condition est vérifiée.
- Format standard : `if (condition){instructions};`
- Le mot clé `else` permet de définir des instructions exécutées si la condition est fausse :

  `if (condition){instructions1}; else {instructions2};`

Code source

```
int a=3;
if (a >= 5)
    printf("Le nombre a est superieur
        ou egal a 5 \n");
else
    printf("Le nombre a est inferieur
        a 5 \n");
```

Sortie

```
$./ex_if
Le nombre a est inferieur a 5
```

Condition `switch`

- Permet de tester l'égalité d'une variable avec plusieurs valeurs.

Code source

```
1  int a=1;
2  switch (a)
3  {
4    case 1:
5    case 2:
6      printf("Le nombre a est egal a
           1 ou 2\n");
7      break;
8    case 3:
9      printf("Le nombre a est egal a
           3\n");
10     break;
11   default:
12     printf("Le nombre a est egal a
           une autre valeur\n");
13 }
```

Sortie

```
1 $./ex_switch
2 Le nombre a est egal a 1 ou 2
```

- Attention à ne pas oublier le mot clé `break` pour éviter d'exécuter tous les cas.
- Le dernier cas `default` permet de gérer les cas qui n'ont pas été définis.

Boucle `for`

- Format en C : `for (initialisation;condition;iteration){instructions};`
- La boucle exécute les `instructions` tant que `condition` est vraie, `iteration` est exécuté à chaque tour de la boucle.
- La boucle for est en général utilisée lorsque l'on connait le nombre d'itérations à priori.
- Exemple d'itérations avec une variable entière :

Code source

```
1 int nb=5;
2 for (int i=0;i<nb;i++)
3   printf("i=%d, i*i=%d\n",i,i*i);
```

Sortie

```
1 $./ex_for
2 i=0, i*i=0
3 i=1, i*i=1
4 i=2, i*i=4
5 i=3, i*i=9
6 i=4, i*i=16
```

- Exemple d'itérations avec une variable flottante :

Code source

```
1 float pas=.2;
2 float max=1.0;
3 for (float x=0;x<max;x=x+pas)
4   printf("x=%f, x*x=%f\n",x,x*x);
```

Sortie

```
1 $./ex_for2
2 x=0.000000, x*x=0.000000
3 x=0.200000, x*x=0.040000
4 x=0.400000, x*x=0.160000
5 x=0.600000, x*x=0.360000
6 x=0.800000, x*x=0.640000
```


Boucle `while`

- Format en C : `while (condition){instructions};`
- La boucle exécute les `instructions` tant que `condition` est vraie.
- Exemple :

Code source

```
1 int s=0;
2 int i=0;
3 int m=21;
4 while(s<m)
5 {
6   i=i+1;
7   s=i*i;
8   printf("s=%d, i=%d\n",s,i);
9 };
```

Sortie

```
1 $./ex_while
2 s=1, i=1
3 s=4, i=2
4 s=9, i=3
5 s=16, i=4
6 s=25, i=5
```

- Attention à l'initialisation et à bien vérifier que la boucle se termine.
- Souvent remplacée par une boucle for en C car cette dernière est extrêmement puissante.

Exercice 2 : Somme

Coder une fonction C calculant la valeur entière

$$sommecarre(n) = \sum_{i=0}^{n} i^2$$

Analyse du problème

- Nom :
- Entrée :
- Sortie :
- Mise en oeuvre :

Solution

Exercice 3 : Produit

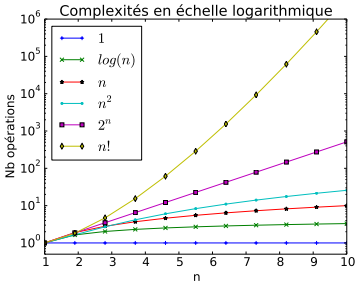
Coder une fonction C calculant la puissance n d'un flottant x :

$$puissance(x, n) = \prod_{i=1}^{n} x = x^n$$

Analyse du problème **Solution**

- Nom : `puissance`
- Entrée :
  - flottant `x`
  - entier `n`
- Sortie : flottant `f`
- Mise en oeuvre :
  - Création d'une variable d'accumulation.
  - Boucle `for`

Complexité d'un algorithme



Complexités typiques

Notation	Type
$O(1)$	constante
$O(log(n))$	logarithmique
$O(n)$	linéaire
$O(nlog(n))$	quasi-linéaire
$O(n^2)$	quadratique
$O(n^p)$	polynomiale
$O(2^n)$	exponentielle
$O(n!)$	factorielle

- La complexité en temps d'un algorithme est définie par le nombre d'opérations nécessaires à son exécution (Notation T(algo)).
- La complexité en espace d'un algorithme est définie par la taille mémoire nécessaire à son exécution (Notation S(algo)).
- Il est souvent difficile de connaître le nombre exact d'opérations et de la taille mémoire, on utilise donc la notation $O(f(n))$ qui permet de se concentrer sur les opérations/variables les plus complexes.
- n est une mesure de la taille des données (par exemple la taille d'un tableau ou le nombre de termes calculés dans une suite).

Notation $O(\cdot)$

- Cette notation permet d'illustrer la dominance d'une fonction par une autre.
- Soient f et g deux fonctions de la variable réelle x. On dit que f est dominée par g en $+\infty$, et on note

  $$f(x) = O(g(x))(x \to +\infty)$$

  lorsqu'il existe des constantes N et C telles que

  $$\forall x > N \qquad |f(x)| \leq C\,|g(x)|\,.$$

- Intuitivement, cela signifie que f ne croit pas plus vite que g.
- Utilisation pour borner asymptotiquement la complexité algorithmique.
- Aussi communément utilisée (parfois avec un petit o) pour les séries de Taylor :

  $$e^x = 1 + x + \frac{1}{2}x^2 + O(x^2) \text{ quand } x \to 0$$

Calcul de complexité

- Les algorithmes de complexité déterministe ont un nombre d'opérations qui peut être déterminé à l'avance.
- Les programmes non déterministes ont un nombre d'opérations qui dépendent de manière complexe des données d'entrée.

Opération	Nom	Complexité $T()$
+,-,*,/	Opérations mathématiques	$O(1)$
=	Affectation	$O(1)$
`for (i=0;i<n;i++)`	Boucle `for`	$O(n)$
`if (cond)`	Test `if`	$O(1)$

- Les complexités de boucles imbriquées se multiplient.
- La complexité d'une boucle `while` peut parfois être estimée (séries entières, vitesse de convergence).

Exercice 4 : Complexité

Calculer la complexité des fonctions suivantes :

Code

```
1 int carre(int i)
2 {
3   return i*i;
4 }
```

Complexité

- Nb opérations :
- Complexité $T()$:
- Complexité $S()$:

Code

```
1 int sommecarre(int n)
2 {
3   int f=0;
4   for (int i=1;i<=n;i++)
5     f=f+i*i;
6   return f;
7 }
```

Complexité

- Nb opérations :
- Complexité $T()$:
- Complexité $S()$:

Code

```
1 float puissance(float x,int n)
2 {
3   float f=x;
4   for (int i=1;i<n;i++)
5     f=f*x;
6   return f;
7 }
```

Complexité

- Nb opérations :
- Complexité $T()$:
- Complexité $S()$:

Exercice 5 : Boucles imbriquées

Donner la complexité des fonctions suivantes visant à calculer :

$$func(x,n) = \sum_{i=1}^{n} x^i$$

Code

```
1  float puissance(float x,int n)
2  {
3    float f=x;
4    for (int i=1;i<n;i++)
5      f=f*x;
6    return f;
7  }
8
9  float func(float x,int n)
10 {
11   float f=0;
12   for (int i=1;i<=n;i++)
13     f=f+puissance(x,i);
14   return f;
15 }
```

Complexité

- Nb opérations :
- Complexité $T()$:
- Complexité $S()$:

Ressources bibliographiques

- La représentation numérique en machine est introduite dans [3, 5, 6].
- Référence intéressante en anglais [4].
- Une bonne introduction au langage C est disponible sur Wikibook [1] (livre en anglais plus complet ou en français), une petit précis [7].
- La bibliothèque standard C est très bien documentée avec des exemples sur le web [2].
- Une discussion très poussée sur la complexité algorithmique (attention aux notations) [5, Chap 1.2.11].

Ressources bibliographiques I

[1] "C programing," https://en.wikibooks.org/wiki/C_Programming, version du 2015-08-28.

[2] "Standard c library," http://www.tutorialspoint.com/c_standard_library/, version du 2015-08-28.

[3] J. Bastien and J.-N. Martin, *Introduction à l'analyse numérique : applications sous Matlab : cours et exercices corrigés*. Dunod, 2003.

[4] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991.

[5] D. E. Knuth, *The Art of Computer Programming, Volume 1 (3rd Ed.) : Fundamental Algorithms*. Redwood City, CA, USA : Addison Wesley Longman Publishing Co., Inc., 1997.

[6] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical recipies in C*. Cambridge university press Cambridge, UK, 1992.

[7] P. Prinz and U. Kirch-Prinz, *C précis et concis*. O'Reilly, 2003.