Introduction au Langage C

Paul HONEINE

— H201 —

Université de technologie de Troyes

– Automne 2014 –

version 1.4 *b*

Table des matières

0. Préambule

Classification des langages de programmation

Classification 1

- Langages interprétés (Perl, Python, Tcl, ...) ou Langages compilés (Pascal, C, ...)

Classification 2

- Première génération : langage machine
- Deuxième génération : langage assembleur
- Troisième génération : langages procéduraux (Fortran, Basic, Pascal, C, ...) ou encore à objets (Java, C++, Python, ...).
- Quatrième génération : avec interface utilisateur et langage plus proche de la syntaxe *naturelle* (Microsoft Access avec SQL, Labview, Matlab, Mathematica)

Classification 3

- La programmation structurée (Fortran, Pascal, C, Perl, Tcl)
- La programmation fonctionnelle (Lisp) ou logique (Prolog)
- La programmation objet (C++, Java, VB.net, C#, ...)

Classification 4

- Langages déclaratifs : le programmeur réfléchit en terme de valeurs des fonctions et de relations entre entités diverses. Il n'y a pas d'attribution de valeurs aux variables (programmation fonctionnelle et programmation logique)
- Langages impératifs : langages incluant des moyens pour le programmeur d'attribuer des valeurs a des locations en mémoire (les autres !)

Classification des langages de programmation

Classification 5

- Programmation Procédurale (Ada, Pascal, C) : Le programme est divises en blocs qui peuvent contenir leurs propres variables ainsi que d'autres blocs.
- Programmation Orientée Objet (C++, Java) : Programmation qui supporte l'interaction d'objets. Un objet contient des données ainsi que des fonctions qui peuvent s'appliquer à ces données.
- Programmation Concurrente (Ada 95) : Langages de programmation qui s'appliquent a plusieurs CPU qui opèrent en parallèle. Les données peuvent être partagées ou non.
- Programmation Fonctionnelle (Lisp) : Un programme est un appel de fonction avec un certain nombre de paramètres, qui eux-mêmes peuvent être des appels d'autres fonctions. Le programme renvoie donc un seul résultat, qui peut-être assez complexe (exemple : une nouvelle fonction).
- Programmation Logique (Prolog) : Un programme consiste à une série d'axiomes, de règles de déduction et à un théorème a prouver. Le programme renvoie la valeur "vraie" si les axiomes supportent le théorème. Il renvoie la valeur "fausse" autrement.

Classification 6

- niveau bas, niveau intermédiaire, niveau haut, niveau très haut

Historique

- Création en 1970 par D. Ritchie (Laboratoire BELL)
- Évolution chronologique : BCPL (Basic Combined Programming Language), B (Version simplifiée du BCPL), CPL (Combined Programming Language)
- A été conçu initialement pour le système UNIX
- Conséquence : le système UNIX est écrit principalement en langage C
- Aujourd'hui, le C est indépendant de l'UNIX. Il peut être utilisé sur tous les systèmes d'exploitation
- Ses applications sont de plus en plus nombreuses (le plus utilisé dans le domaine industriel)
- A partir du langage C, plusieurs langages dérivent (C++, C#, Objective-C, ...)

Propriétés du langage C

- Le langage C est également un langage de *bas niveaux*, dans le sens où il permet la manipulation de données élémentaires que manoeuvrent les ordinateurs : *Bit (binary digit), octet (8 bits), et adresse*
- Il est suffisamment général pour permettre de développer des applications de type scientifique ou de gestion basée sur l'accès aux bases de données (Word et Excel sont écrits à partir de C ou C++)
- Il est un des 1ers langages offrant des possibilités de programmation modulaire : un programme peut être constitué de plusieurs module (module = fichier .c)

Propriétés du langage C

Langage bas niveau

- Un langage de programmation a pour finalité de communiquer avec la machine. Le langage maternel de la machine n'utilise que deux symboles (0 et 1) : c'est le *langage machine*.
  Exemple : le nombre 5 est reconnu par une machine par la succession des symboles 101 : c'est la représentation du nombre en base 2.

- De même, les opérations qu'une machine est capable d'exécuter sont codées par des nombres, c'est-à-dire une succession de 0 et 1.
  Exemple : l'instruction machine 00011010 0001 0010 demande à la machine d'effectuer l'opération 1+2.

- Ce langage machine est le seul qui soit compris par l'ordinateur.
  Est-il alors le seul moyen pour communiquer avec celui-ci ?
  Réponse : Non, utilisation du langage assembleur : add $1 $2
  Mais, ce langage assembleur n'est pas portable

Caractéristiques du langage C

1/2

- Type d'application : programmation procédurale
  Le programme est divisé en blocs qui peuvent contenir leurs propres variables ainsi que d'autres blocs.
- Structures de contrôle évoluées :
  - Conditionnelles (`if`, `else`, `switch`)
  - Répétitives (`for`, `do`, `while`)
  - Utilisation en séquence et/ou en imbrication.
- Structures de (sous-) programmes :
  - Procédures
  - Fonctions
  - Appel à des bibliothèques qui peuvent être utilisées sur plusieurs types de machines.
- Types de données très variés et personnalisables :
  - Entiers (`int`, `short`, ...)
  - Réels (`float`, `double`, ...)
  - Caractères (`char`)
  - Pointeurs (adresse mémoire)
  - Structures et types (possibilité de définition)
  - Fichiers, ...

Caractéristiques du langage C

2/2

- Langage évolué
  - Structuration
  - Modularité : peut être découpé en modules qui peuvent être compilés séparément
  - Portabilité : sur n'importe quel système en possession d'un compilateur C
  - ...
- Permet aussi de réaliser des instructions proches du langage Assembleur (bas niveau) : manipulation des adresses, pointeurs, registres, ...
- Limité en traitement et en calcul mathématique mais offre la possibilité d'ajout des fonctions et des bibliothèques
- Code généré de taille très réduite
- Offre une grande liberté au programmeur mais nécessite en revanche une grande maîtrise de ses règles

Quelques références ...

Un programme en C

Ecrire un programme en C

Etapes de réalisation

Etapes de réalisation

1. **Ecrire le fichier .c** (ou .cpp) avec un éditeur texte (WordPad, Notepad, ...)
2. **Compiler** (Code Warrior, Code::Blocks, Borland C/C++, Visual C++, gcc, KDevelop, Xcode, ...)
3. Etape cachée : Assemblage (transformer le code assembleur en un fichier binaire)
4. Etape cachée : Linker (lier les fichier objets)
5. **Résultat** : un fichier exécutable

Un programme C est un texte écrit avec un éditeur de texte, respectant une certaine syntaxe et stocké sous forme d'un ou plusieurs fichiers (généralement avec l'extension .c). A l'opposé du langage assembleur, les instructions du langage C sont obligatoirement encapsulées dans des fonctions et il existe une fonction privilégiée appelée `main` qui est le point de départ de tout programme.

Ecrire un programme en C

Le plus important ...

Programmation en C

Le premier programme

```c
#include <stdio.h>  /* entête = header */

/* Ce programme permet d'afficher un texte
en utilisant une des fonctions d'entrée-sortie
de la bibliothèque standard stdio.h */

int main(void)
{

 printf("Salut UTT\n");  // une seule ligne suffit

 return 0;
}
```

Programmation en C

en ANSI C, C++, C#, Objective-C

[Hello, word!]

C (ANSI)

```c
#include <stdio.h>

int main(void)
{
    printf("Salut UTT\n");
    return 0;
}
```

C++ (ISO)

```cpp
#include <iostream>

using namespace std;

int main()
{
    cout << "Salut UTT" << endl;
    return 0;
}
```

C#

```csharp
public class HelloWorld
{
    static void Main()
    {
        System.Console.Write("Salut UTT\n");
    }
}
```

Objective-C (orienté objet)

```objc
#import <stdio.h>
#import <objc/Object.h>

@interface Salut : Object
{
}
- salut;
@end

@implementation Salut
- salut
{
   printf("Salut UTT\n");
}
@end

int main(void)
{
   id obj;
   obj = [Salut new];
   [obj salut];
   [obj free];
   return 0;
}
```

1. Les bases du langage C

Un programme en C : un aperçu

Les parties d'un programme en C

Appels aux bibliothèques, déclaration des types et des structures

Définition de la $1^{\text{ère}}$ fonction ou procédure

Définition de la 2^{nd} fonction ou procédure

⋮

Définition de la $n^{\text{ème}}$ fonction ou procédure

Définition du programme principal **main**

Présentation de quelques instructions du langage C

Exemple.c

```c
#include <stdio.h>
#include <math.h>
#define NFois 5

void main()
{
   int i;
   float x;
   float racinex;

   printf("Bonjour!\n");
   printf("Je vais vous calculer %d racines carrées\n", NFois);

   for(i=0;i<NFois;i++)
   {
        printf("Donnez un nombre: ");
        scanf("%f",&x);
        if (x<0)
           printf("Le nombre %f ne possède pas de racine carrée\n", x);
        else
        {
           racinex=sqrt(x);
           printf("Le nombre %f a pour racine carrée : %f\n", x, racinex);
        }
    }
    printf("Travail terminé - Au revoir!");
}
```

Exemple2.c

Exemple2.c

```c
#include <stdio.h>      // appel à une bibliothèque

void procedure()        // déclaration d'une procédure
{
   printf("la procédure a été exécutée\n");
}
void main()             // déclaration du programme principal
{
   int i;               // déclaration d'une variable
   i=0;                 // initialisation d'une variable

   printf("Bonjour, on va exécuter la procédure\n");

   procedure();         // premier appel de la procédure
   i=i+1;               // incrémentation de la variable i

   procedure();
   i=i+1;

   printf("On a exécuté la procédure %d fois\n",i);
}
```

Les bases du C : les fichiers

- On peut écrire un programme en langage C sur n'importe quel éditeur texte (de préférence un éditeur standard comme WordPad) à condition d'attribuer une extension du type .c ou .cpp
- Exécution du programme par un compilateur C
- Génération de l'exécutable en cas de succès et exécution du code

Les éléments constructifs d'un programme

Les éléments constructifs d'un programme : entête

- Les bibliothèques sont des fichiers du type `.h`. On peut les appeler grâce à la commande `#include`. Il existe déjà un ensemble de bibliothèques disponibles avec les compilateurs C et on a la possibilité de définir nos propres bibliothèques et de les ajouter
- Dans l'entête d'un programme C, on peut aussi définir des constantes grâce à la commande `#define` (exemple : `#define m 7`)
- On peut également définir de nouveaux types des données d'une manière personnalisée grâce à `struct` et `typedef` à partir des types de base (exemple : `int`)

```
#include <stdio.h>
#define m 7              //définition de la constante m

//définition de la structure couple
typedef struct {int l; int j; } couple;

void main()
{
   couple cup;           //déclaration de la variable cup
   cup.j=m; cup.l=2;     //initialisation de la variable cup
   printf("la somme est %d\n",cup.j+cup.l);
}
```

Les éléments constructifs d'un programme : fonctions et procédures

```
<type> NomFonction(<type1> var1, <type2> var2, , <typen> varn)
{
// Déclaration des variables locales
Instruction1;
InstructionN;
}
```

- NomFonction : est le nom de la fonction
- <type> : est le type de la donnée retournée par la fonction (exemple : int)
- <type1> : est le type de la donnée de la variable var1
- <typen> : est le type de la donnée de la variable varn
- On déclare les fonctions et les procédures en séquence. Attention : les fonctions devraient être déclarées avant d'être appelées par d'autres fonctions !
- La valeur retournée par la fonction varie en fonction des valeurs prises par ses variables arguments
- Les variables de la fonction sont indiquées entre parenthèses après le nom de la fonction. Deux variables sont séparées par une virgule ","
- Une fonction est composée de plusieurs instructions
- On peut définir des variables à l'intérieur de la fonction. Dans ce cas, elles sont dites locales car elles ne seront utilisées que par les instructions de la fonction
- Une fonction peut être appelée un nombre illimité de fois par une autre fonction
- Si la fonction ne retourne pas de valeur, il s'agit dans ce cas d'une procédure (un ensemble d'instructions à exécuter). Dans ce cas, le type de la donnée retournée par la procédure est void

Les éléments constructifs d'un programme : instructions

Une fonction est composée de plusieurs instructions

- Chaque instruction est terminée par un séparateur qui marque la fin de celle-ci, c'est le point-virgule ";"
- Le nombre des instructions dans une fonction n'est pas limité
- L'ensemble des instructions d'une fonction est délimité entre deux accolades marquant le début et la fin du bloc des instructions de la fonction
- Une instruction peut être écrite sur plusieurs lignes. Plusieurs instructions peuvent être écrites sur la même ligne à condition d'utiliser le séparateur (le point-virgule)

Les éléments constructifs d'un programme : fonction principale `main`

- La dernière partie d'un programme en C est le programme principal
- C'est une fonction qui porte obligatoirement le nom `main`. Elle est toujours exécutée en premier. Ses variables représentent les variables globales du programme
- Le programme principal a une structure similaire aux autres fonctions (déclarations, instructions, syntaxe, ...)

Ne pas oublier :

- Des fonctions prédéfinies peuvent être utilisées en faisant appel à des bibliothèques comme la fonction `printf` de la bibliothèque `stdio.h`
- Des commentaires peuvent être insérés à la suite de `//` sur la même ligne ou entre les délimiteurs `/*` `*/` pas nécessairement sur la même ligne

Application : écrire une fonction

```
/* Routine de calcul du maximum */
int imax(int n, int m)     // Déclaration de la fonction
{
   int max;                // Variable locale

   if (n>m)
         max = n;
   else
         max = m;
   return max;             // Valeur retournée par la fonction
}
```

Application : écrire un programme

Écrire un programme permettant de calculer l'aire d'un rectangle, et d'afficher le résultat.

Évaluer le résultat pour une largeur $l = 5$ et une longueur $L = 24$.

```c
#include <stdio.h>

void main()
{
      float l;
      float L;
      float Surface;

      l=5;
      L=24;
      Surface=l*L;
      printf("La surface est = %.2f\n",Surface);
}
```

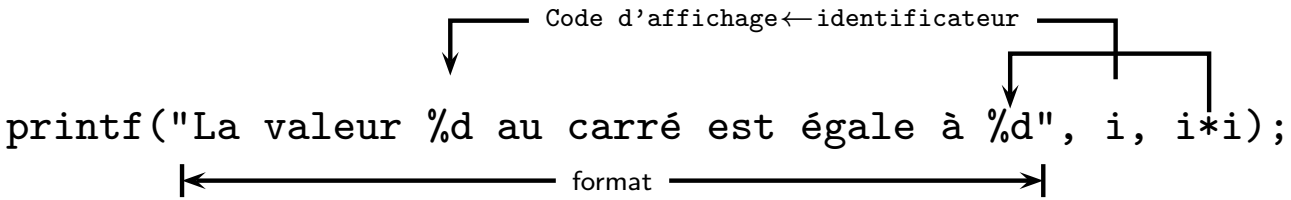
printf

printf

syntaxe

syntaxe de printf

```
printf(<format>, <identificateur1>, ... , <identificateurn>);
```



Le **format** indique comment vont être affichées les valeurs des variables. Il est composé de texte et de codes d'affichage suivant le type de variable

Format	Type de données	Exemple
d (ou i)	Nombre entier en décimal (signé)	392
u	Nombre entier en décimal (non signé)	7235
o	Nombre entier en octal (non signé)	610
x (ou X)	Nombre entier en hexadécimal (non signé) en minuscules (ou majuscules)	7fa ou 7FA
f	Nombre réel (Flottant)	392.65
e (ou %E)	Notation scientifique (mantisse/exponent) avec le caractère e (ou %E)	3.9265e+2
g (ou G)	Nombre réel sous la forme la plus courte entre %f et %e (ou %E)	392.65
c	Caractère	a
s	Chaîne de caractères	chaine
%	L'usage de %% affiche un %	%
p	Adresse pointeur	B800 :0000

printf

Exemple et Séquence d'échappement

```c
#include <stdio.h>

int i;                                  // Déclaration de variable globale

void main()
{
      i=23;                             // Initialisation de i
      printf(" i(dec) = %d\n",i);       // Affichage de i en décimal
      printf(" i(octal) =%o\n",i);      // Affichage de i en octal
      printf(" i(hex)= %x\n",i);        // Affichage de i en Hexadécimal
}
```

Séquence d'échappement	Affichage et signification
\\	affichage du caractère \
\'	affichage d'apostrophe '
\"	affichage de guillemet "
\Onn	Valeur octale (ASCII) de nn
\Xnn	Valeur hexadécimal de nn
\n	nouvelle ligne
\r	retour chariot
\b	backspace ; espace arrière
\0	caractère null
\f	saut de page
\t	tabulation horizontale
\v	tabulation verticale
\a	alerte ; bip (code ASCII 7)

printf

Exemple

```c
#include <stdio.h>

#define M_PI  3.14159265358979323846  /* pi */

#define YEAR 2007

int main()
{
   printf ("Caracteres : %c %c %c %c\n", 'a', 65, 0x30, '0');
   printf ("Entiers : %d %ld\n", YEAR, 650000);
   printf ("Affichage avec espaces : |%10d|\n", YEAR);
   printf ("Affichage avec zeros : |%010d|\n", YEAR);
   printf ("Differentes bases : %d %x %o %#x %#o \n", 100, 100, 100, 100, 100);
   printf ("Reels : |%4.2f| |%+.4e| |%E|\n", M_PI, M_PI*YEAR, M_PI);
   printf ("Largeur en argument : |%*d|\n", 5, 10);
   printf ("%s\n", "Debian GNU/Linux");
   return 0;
}
```

```
Caracteres : a A 0 0
Entiers : 2007 650000
Affichage avec espaces :       2007|
Affichage avec zeros : |0000002007|
Differentes bases : 100 64 144 0x64 0144
Reels : |3.14| |+6.3052e+03| |3.141593E+00|
Largeur en argument :    10|
Debian GNU/Linux
```

Les identificateurs et les mots réservés dans le langage C

Les identificateurs et les mots réservés dans le langage C

Identificateur :

- C'est l'expression utilisée pour identifier (nommer) une variable ou une fonction.
  - Un identificateur doit toujours commencer par un caractère (les lettres de l'alphabet en majuscule ou en minuscule et le caractère _).
  - Un identificateur peut être composé de caractères et de chiffres.
  - Attention : il existe aussi un ensemble d'expressions interdites pour jouer le rôle d'un identificateur (exemples : `int`, `float`, ...). Voir la liste des mots réservés

Mots réservés :

- Le langage C utilise des expressions spécifiques pour désigner des fonctions, des constantes ou des structures de contrôle. En conséquence, ces expressions ne peuvent être utilisées comme identificateurs. La liste de ces expressions sont mentionnées dans le tableau suivant :

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

2. Types de base, Instructions et Opérateurs

Rappel sur le codage de l'informatique

Codage de l'informatique

Ce rappel a pour objectif de comprendre le lien entre une donnée et sa représentation sur un support informatique ou un programme informatique (en conséquence, déterminer le type adéquat pour déclarer une variable dans un programme informatique...).

- En informatique, les machines sont électriques et ne savent qu'interpréter deux données :
  - Donnée 1 : passage d'un courant électrique ;
  - Donnée 2 : absence du courant électrique.
- A ces deux informations, les informaticiens ont associé deux éléments binaires : le 1 logique et le 0 logique
- L'alphabet binaire $\{0, 1\}$ a permis de construire toute une théorie mathématique permettant le codage de l'information avec des séquences formées à partir des 0 et des 1
- **Bit** : un bit est une variable binaire qui peut prendre l'une de deux valeurs 0 et 1
- **Octet** : un octet est une séquence formée de 8 bits. Il en résulte que nous pouvons représenter 2^8 états différents grâce à un octet soit 256 valeurs différentes

Question : à partir des bits (et des octets), comment peut-on représenter les chiffres, les nombres, les caractères, les réels ?
Réponse : selon le domaine de définition d'une variable donnée, on peut déterminer le nombre des bits (ou des octets) qui sont nécessaires pour représenter toutes les valeurs potentielles de la variable.

Codage de l'informatique

Théorème de conversion

Théorème

Pour tout x et y (entiers) tels que $y > 1$, il existe un entier n et une suite $(a_0, a_1, \ldots, a_n)$ tels que

$$x = a_0 \cdot y_0 + a_1 \cdot y_1 + \ldots + a_n \cdot y_n$$

et $a_i < y_i$ pour tout $i \leq n$.

Pour déterminer la suite $(a_0, a_1, \ldots, a_n)$, il suffit d'appliquer récursivement la division euclidienne de x par y et de remplacer x par le résultat de sa division par y si un tel résultat est supérieur à y.

Codage de l'informatique

Exemple de conversion

Exemple de conversion :
Si $x_{\text{binaire}} = 1101$, alors $x_{\text{décimal}} = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 = 13$

Pour l'opération inverse, c'est-à-dire convertir un nombre décimal en un nombre binaire, il suffit d'appliquer itérativement la division euclidienne. Nous proposons l'exemple suivant comme illustration.

$$135_{\text{décimal}} = 1000\ 0111_{\text{binaire}}$$

135	2	67	2	33	2	16	2	8	2	4	2	2	2	1	2
1	67	1	33	1	16	0	8	0	4	0	2	0	1	1	0

$$\begin{aligned}
135_{\text{décimal}} &= 2 \cdot 67 + 1 \\
&= 2 \cdot (2 \cdot 33 + 1) + 1 \\
&= 2^2 \cdot 33 + 1 \cdot 2^1 + 1 \cdot 2^0 \\
&\vdots \\
&= 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\
&= 1000\ 0111_{\text{binaire}}
\end{aligned}$$

Codage de l'informatique

Codage octal

Codage octal :
C'est le codage en base 8. Les chiffres utilisés sont compris entre 0 et 7.

Décimal	0	1	2	3	4	5	6	7	8	9	10	11	···
Octal	0	1	2	3	4	5	6	7	10	11	12	13	···

Le passage de la base octale à la base binaire est immédiat ; chaque chiffre est décomposé sur 3 bits.

A titre d'exemple, la conversion de $145_{\text{décimal}}$ en octal consiste à réaliser les divisions successives de ce nombre par 8.

145	8
1	18

18	8
2	2

2	8
2	0

Il s'ensuit que $145_{\text{décimal}} = 221_{\text{octal}}$.

Par la suite, $145_{\text{décimal}}$ vaut $010\ 010\ 001$ en binaire.

Codage de l'informatique

Codage hexadécimal

Codage hexadécimal :
L'alphabet est représenté ici sur 16 éléments.

Décimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Hexadécimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10

La conversion d'un nombre en hexadécimal au nombre binaire, qui lui correspond, est évidente. Chaque chiffre d'un nombre en hexadécimal peut être représenté par un groupe de 4 éléments binaires (bits).

Le nombre 45 en hexadécimal, donne en décimal $4 \cdot 16^1 + 5 \cdot 16^0 = 69$. Le même nombre donne en binaire $0100\ 0101$ (remarquons que $4 = 0100$ et que $5 = 0101$).
Pour l'opération inverse, il suffit d'appliquer itérativement la division euclidienne de 69 sur 16 pour le décodage en hexadécimal.

$$\begin{array}{r|l} 69 & 16 \\ \hline 5 & 4 \end{array} \qquad \begin{array}{r|l} 4 & 16 \\ \hline 4 & 0 \end{array}$$

Cherchons à convertir 177 à titre d'exemple :

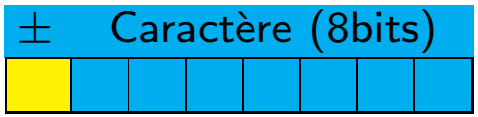
$$\begin{array}{r|l} 177 & 16 \\ \hline 1 & \mathrm{B} \end{array} \qquad \begin{array}{r|l} \mathrm{B} & 16 \\ \hline \mathrm{B} & 0 \end{array}$$

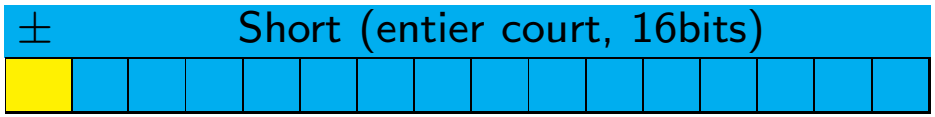
Il s'ensuit donc que $177_{\text{Décimal}} = \mathrm{B1}_{\text{Hexadécimal}}$.

Codage de l'informatique

Code ASCII

- Nous venons de voir qu'il est possible de coder des nombres entiers en utilisant exclusivement des 0 et des 1
- Notons aussi que les bases utilisées en informatique sont des bases représentant des puissances de la base binaire (base 8, base 16, ...)
- Les réels peuvent être aussi représentés par des 0 et des 1 (voir cours NF04)
- Les caractères aussi sont codés en informatique sur 7 bits (donc un octet suffit). C'est-à-dire, à chaque caractère, on associe un code entier entre 0 et 127 qui représente ce qu'on appelle le **CODE ASCII** du caractère (exemple : le caractère **B** possède un code ASCII égal à **66**).
- Les caractères et les entiers ont le même schéma de codage : un bit de signe suivi d'un champ indiquant la valeur :





- En revanche, les valeurs réelles (`float` et `double`) sont codées sous la formule :

$$\text{valeur} = (\text{signe})\ \text{mantisse} \cdot 2^{\text{exposant}}$$

  - le signe est représenté par un seul bit, le bit de poids fort
  - l'exposant est codé sur les 8 bits consécutifs au signe
  - la mantisse (les bits situés après la virgule) sur les 23 bits restants. Elle est composée de la partie décimale de la valeur en base 2 normalisée (enlever le bit le plus à gauche - voir NF04)

Codage de l'informatique

Code ASCII

binaire	oct	déc	hex	
0100000	040	32	20	espace
0100001	041	33	21	!
0100010	042	34	22	”
0100011	043	35	23	#
0100100	044	36	24	$
0100101	045	37	25	%
0100110	046	38	26	&
0100111	047	39	27	’
0101000	050	40	28	(
0101001	051	41	29	)
0101010	052	42	2A	*
0101011	053	43	2B	+
0101100	054	44	2C	,
0101101	055	45	2D	–
0101110	056	46	2E	.
0101111	057	47	2F	/
0110000	060	48	30	0
0110001	061	49	31	1
0110010	062	50	32	2
0110011	063	51	33	3
0110100	064	52	34	4
0110101	065	53	35	5
0110110	066	54	36	6
0110111	067	55	37	7
0111000	070	56	38	8
0111001	071	57	39	9
0111010	072	58	3A	:
0111011	073	59	3B	;
0111100	074	60	3C	<
0111101	075	61	3D	=
0111110	076	62	3E	>
0111111	077	63	3F	?

binaire	oct	déc	hex	
1000000	100	64	40	@
1000001	101	65	41	A
1000010	102	66	42	B
1000011	103	67	43	C
1000100	104	68	44	D
1000101	105	69	45	E
1000110	106	70	46	F
1000111	107	71	47	G
1001000	110	72	48	H
1001001	111	73	49	I
1001010	112	74	4A	J
1001011	113	75	4B	K
1001100	114	76	4C	L
1001101	115	77	4D	M
1001110	116	78	4E	N
1001111	117	79	4F	O
1010000	120	80	50	P
1010001	121	81	51	Q
1010010	122	82	52	R
1010011	123	83	53	S
1010100	124	84	54	T
1010101	125	85	55	U
1010110	126	86	56	V
1010111	127	87	57	W
1011000	130	88	58	X
1011001	131	89	59	Y
1011010	132	90	5A	Z
1011011	133	91	5B	[
1011100	134	92	5C	\
1011101	135	93	5D	]
1011110	136	94	5E	^
1011111	137	95	5F	_

binaire	oct	déc	hex	
1100000	140	96	60	‘
1100001	141	97	61	a
1100010	142	98	62	b
1100011	143	99	63	c
1100100	144	100	64	d
1100101	145	101	65	e
1100110	146	102	66	f
1100111	147	103	67	g
1101000	150	104	68	h
1101001	151	105	69	i
1101010	152	106	6A	j
1101011	153	107	6B	k
1101100	154	108	6C	l
1101101	155	109	6D	m
1101110	156	110	6E	n
1101111	157	111	6F	o
1110000	160	112	70	p
1110001	161	113	71	q
1110010	162	114	72	r
1110011	163	115	73	s
1110100	164	116	74	t
1110101	165	117	75	u
1110110	166	118	76	v
1110111	167	119	77	w
1111000	170	120	78	x
1111001	171	121	79	y
1111010	172	122	7A	z
1111011	173	123	7B	{
1111100	174	124	7C	|
1111101	175	125	7D	}
1111110	176	126	7E	~
1111111	177	127	7F	suppr

... et 32 caractères de contrôle, dont $10_{\text{déc}}$ pour nouvelle ligne, $13_{\text{déc}}$ pour retour chariot et $28_{\text{déc}}$ pour séparateur de fichier

Types de base

Les types standards

- `int`
  - convient aux variables entières
  - occupe généralement 2 ou 4 octets de mémoire (dépend de la machine)
- `char`
  - convient pour des chaînes de caractères
  - nécessite 1 octet
- `float`
  - convient aux variables réelles
  - nécessite 4 octets
- `double`
  - convient aux variables réelles en double précision
  - nécessite 8 octets
- `short`
  - convient aux variables entières
  - occupe 2 octets dans la mémoire
  - convient lorsque le type `int` n'est pas codé sur 2 octets sur la machine
- `long`
  - convient aux variables entières
  - occupe 4 octets dans la mémoire
- `unsigned`
  - permet d'affecter un signe toujours positif à la variable (conséquence : la valeur minimale et la valeur maximale changent)
- `signed`
  - le signe peut être positif ou négatif

Comment déclarer ?

- La déclaration se fait généralement de la manière suivante : `<type> NomVariables;`
- Attention : la taille occupée par un type de variable dépend de la machine (l'ordinateur). C'est pourquoi on peut/doit interroger sa machine pour connaître cette taille grâce à la fonction `sizeof`.

```c
#include <stdio.h>
int main()
{
    int A=40000;
    short B=32768;
    printf("A = %d B= %d\n", A,B);
    printf("Taille int = %d octets\n", sizeof(int));
    printf("Taille char = %d octets\n", sizeof(char));
    printf("Taille short = %d octets\n", sizeof(short));
    printf("Taille long = %d octets\n", sizeof(long));
    printf("Taille float = %d octets\n", sizeof(float));
    printf("Taille double = %d octets\n", sizeof(double));
    printf("Taille long double = %d octets\n", sizeof(long double));
    return 0;
}
```

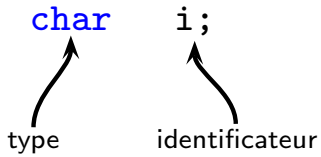
```
Taille int = 4 octets
Taille char = 1 octets
Taille short = 2 octets
Taille long = 4 octets
Taille float = 4 octets
Taille double = 8 octets
Taille long double = 12 octets
```

Déclaration d'une variable

syntaxe

syntaxe de déclaration d'une variable

```
<type> <identificateur1>, ... , <identificateurn>;
```



- Le **type** détermine la nature de la variable ainsi que les opérations pouvant être effectuées. Il dépend de la machine ou du compilateur utilisé. En précédant le type de la variable par le mot `unsigned` (à l'exception de `float` et `double`), la variable devient non-signée et cela permet d'étendre la plage de valeurs.
- L'**identificateur** est le nom affecté à la variable. Le nombre de caractères est souvent limité selon le compilateur utilisé.
  - Attention à la casse :
    MonIdentificateur $\neq$ monidentificateur $\neq$ MONIDENTIFICATEUR
  - Le premier caractère ne doit pas être un chiffre
  - Les lettres accentuées ne sont pas autorisées

Déclaration d'une variable

Exemple

```
/* Déclaration de réel */
float rayon;
/*Déclaration d'entier */
int i,j;
/* Déclaration de caractère */
char t;
/*Déclaration de réel double */
double pi;
/* Déclaration d'un octet */
unsigned char octet;
/*Déclaration d'un octet avec la classe registre */
register unsigned char port;

void main()
{
   rayon=10.14;
   i=2;
   j=3;
   t='A';          /* t=65 Code ASCII de A */
   pi=3.14159;
   octet=129;      /* On peut aller au dessus de +127 */
   port=34;
}
```

Déclaration d'une variable

Octale et hexadécimale

Remarque :
Remarque : Lors de l'affectation des variables si on met `0` avant la valeur, elle sera en Octal et si on met `0x` devant une valeur elle sera hexadécimale.

```
/* Déclaration d'entier */
int i;

void main()
{
    i=21;       /* Décimal */

    i=025;      /* Octal */

    i=0x15;     /* Hexadécimal */
}
```

Déclaration d'une variable

Initialisation

Deux solutions sont possibles pour l'initialisation

```
/* Déclaration */
int i;

void main()
{
    /* Initialisation */
    i=15;
    .
    .
}
```

```
/* Déclaration */
int i=15;

void main()
{
    .
    .
    .
    .
}
```

Pointeur

- Toute variable manipulée dans un programme est stockée quelque part en mémoire centrale. Cette mémoire est constituée d'octets qui sont identifiés par un numéro qu'on appelle adresse
- Pour retrouver une variable, il suffit de connaître l'adresse où elle est stockée
- Un pointeur est une variable contenant une adresse, donc peut être défini sur tous les types de données (variables et fonctions).

syntaxe de déclaration d'un pointeur

```
<type> *Variable;
```

- `Variable` est un pointeur qui contient l'adresse mémoire de la variable `*Variable`

```
#include <stdio.h>
int main()
{
   float *ptr;
   float a;
   a=3.4;
   ptr=&a;
   printf("la valeur contenue dans l'adresse est %1.2f \n",*ptr);
   printf("l'adresse est %d \n",ptr);
   printf("l'adresse de a est %d \n",&a);
   *ptr=a+1;
   printf("la valeur contenue dans l'adresse est %1.2f \n",*ptr);
   printf("l'adresse est %d \n",ptr);
   printf("l'adresse de a est %d \n",&a);
   printf("la valeur de a est %1.2f \n",a);
   return 0;
}
```

Les constantes

- Après avoir défini les types de variables, il est nécessaire de définir les constantes. Elles vont simplement représenter les valeurs prises par les variables.
- On peut lister les catégories des bases suivantes :
  - Les constantes entières : les entiers (exemples : 4, 77, -98, ...)
  - Les constantes réelles : les flottants (exemples : 332.88, 15, 0.094, - 12.06, ...).
  - Les constantes caractères : elles sont délimitées par deux apostrophes : un seul caractère (exemples : 'a', 'E', ' ', '{', ...).
  - Les constantes chaînes de caractères : il s'agit d'une séquence de caractères délimitée par deux guillemets (exemples : " 1a7", " 123o0"...).

```c
#include <stdio.h>
int main()
{
    char car;
    int i;
    float r;
    char *chaine;
    car='a';
    i=4;
    i=-5;
    r=3.98;
    chaine="reau";
    printf("la chaine est : %s\n",chaine);
    printf("le caractere est : %c\n",car);
    printf("le code ASCII du caractere est : %d\n",car);
    printf("l'entier est : %d\n",i);
    printf("le reel est : %f\n",r);
    return 0;
}
```

```
la chaine est : reau
le charactere est : a
le code ASCII du caractere est : 97
l'entier est : -5
le reel est : 3.980000
```

Les tableaux

- Les tableaux sont des structures pratiques et nécessaires pour traiter plusieurs variables du même type et pour les ranger.
- En C, on peut définir des tableaux pour tous les types de base (entier, réel, caractère, ...)
- En C, l'indexation d'un tableau commence à partir de 0

syntaxe de déclaration d'un tableau

```
<type> tableau[Nombre_elements];
```

- Exemple : `int tab[5]`, alors la taille du tableau est 5 (nombre d'éléments)
- `tab[0]` représente le premier élément du tableau et `tab[1]` est le deuxième élément et ainsi de suite ... Le dernier élément est `tab[4]` (`tab[5]` n'existe pas)

- On peut aussi créer un tableau à l'aide d'un pointeur. Dans ce cas, la taille n'est pas spécifiée immédiatement

syntaxe de déclaration d'un tableau à l'aide d'un pointeur

```
<type> *tableau;
```

- Exemple : `int *table;`
- `table[0]` représente le premier élément, `table[2]` est le troisième élément ...
- Le même principe reste valable pour les caractères.

Les tableaux

Exemple

```c
#include <stdio.h>
int main()
{
    char car[4];
    int tab[4];
    int *table;
    char *chaine;
    chaine="oiseau";
    car[0]=chaine[0];
    car[1]=chaine[1];
    car[2]=chaine[2];
    car[3]=chaine[3];
    printf("%c %c %c %c \n",car[0],car[1],car[2], car[3]);
    printf("taille de car =%d octets\n",sizeof(car));
    printf("taille de chaine =%d octets\n",sizeof(chaine));
    printf("taille de chaine =%d octets\n",sizeof(*chaine));
    tab[0]=9;
    tab[1]=2;
    tab[2]=3;
    tab[3]=4;
    table=tab;
    printf("%d %d %d %d\n",table[0],table[1],table[2], table[3]);
    printf("taille de table =%d octets\n",sizeof(table));
    printf("taille de table =%d octets\n",sizeof(*table));
    printf("taille de tab =%d octets\n",sizeof(tab));
    return 0;
}
```

```
o i s e
taille de car =4 octets
taille de chaine =4 octets
taille de chaine =1 octets
9 2 3 4
taille de table =4 octets
taille de table =4 octets
taille de tab =16 octets
```

Les chaînes de caractères

- En C, une chaîne de caractères est un tableau de caractères se terminant par un code de fin appelé le caractère nul `\0`.

syntaxe de déclaration d'une chaîne de caractères

```
char NomVariable[NombreDeCaractères + 1];
```

```
char messageA[4] = "utt";

char messageB[] = "utt";

char messageC[4];

void main()
{
    messageC[0]='u';
    messageC[1]='t';
    messageC[2]='t';
    messageC[3]='\0'; /* Caractère de fin */
}
```

- `message` est considérée comme une adresse par le compilateur. On ne peut pas écrire `message="utt"`. Soit on initialise chaque caractère comme dans l'exemple ci-dessus, ou bien on utilise la fonction `strcpy` de la bibliothèque `string.h`.

Instructi ons

Affectation

- En C, l'affectation se fait par l'intermédiaire de l'opérateur =

syntaxe d'affectation

```
variable=valeur;
```

A l'issue de cette affectation, la `variable` prend la valeur `valeur`

```
int var;
var=7;
```

- L'affectation peut se faire aussi par l'intermédiaire d'une autre variable

syntaxe d'affectation

```
variable1=variable2;
```

Dans ce cas, la variable `variable1` prend la même valeur prise par `variable2`

```
int var1, var2;
var2=5;
var1=var2;
```

Affectation

```c
#include <stdio.h>
int main()
{
    char car1, car2;
    float a=5, b=7;

    car1='6';
    car2='z';
    printf("%c et %c \n",car1, car2);
    car1=car2;
    printf("%c et %c \n",car1, car2);

    printf("%1.2f et %1.2f \n",a, b);
    a=a+b;
    b=a-b;
    printf("%1.2f et %1.2f \n",a, b);

    return 0;
}
```

```
6 et z
z et z
5.00 et 7.00
12.00 et 5.00
```

(Affectation par) conversion

L'affectation en C peut se faire entre variables de types différents sous réserve de plusieurs conditions :

- `int i; double x=4.3; i=x;` Ici, `i` prendra la valeur 4
  On a alors une perte d'information
- `int j; double y=6.1; j=(int)y;` La valeur 6 est affectée à `j`
  Dans ce cas, la conversion est commandée (conversion explicite, ou `cast`).
- `int j=5; char y='d'; j=j+y;`
  Dans ce cas, la conversion se faire dans une expression (conversion implicite).

```c
#include <stdio.h>
int main()
{
  int i;
  double x=4.3;
  i=x;
  printf("x=%1.1f et i=%d \n",x,i)
       ;
  return 0;
}
```

```c
#include <stdio.h>
void main()
{
  char ca;
  int x=114;
  ca=(char)x;
  x=x+ca;
  printf("ca=%c et x=%d \n",ca, x);
  x=0;
  ca='d'+14;
  x=x+ca;
  printf("ca=%c et x=%d \n",ca, x);
}
```

```
ca=r et x=228
ca=r et x=114
```

Conversion

Il existe deux conversions possibles

- La **conversion implicite** est effectuée pour évaluer le même type de données lors d'évaluation d'expressions. Les conversions systématiques se font toujours du type le plus petit vers le plus long. Exemple : de `char` en `int`, en `float`, en `double`
- La **conversion explicite** consiste à changer le type d'une variable vers un autre type en utilisant l'opérateur `cast` en mettant `(type)` devant l'identificateur de variable à convertir.

```c
#include <stdio.h>
/* Déclaration des variables */
char car;
int a,b,c;
float g;

void main()
{
   a=4;
   b=7;
   c=0x41; /* Code Ascii de 'A' */

   /* Conversion implicite de a et b en float */
   g= (a+b)/100.;
   printf("g= %f\n",g);

   /* Conversion explicite c en char */
   car = (char) c +3;
   /* c est de type entier et sera converti en char */
   printf("car = %c\n",car);
}
```

```
g=0.110000
car=D
```

Opérateurs

Opérateurs

Opérateurs arthmétiques +,-,*,/,%

Opérateur +

- Il peut opérer en incrémentation : `x++;` et `++x;` sont équivalentes à `x=x+1;`
- Il peut opérer sur une seule expression :
  - `x++;` produit une incrémentation de `x` après son traitement
    Exemple : `y=x++;` est équivalent à `y=x; x=x+1;` (évaluation puis incrémentation)
  - `++x;` permet d'incrémenter `x` avant son traitement
    Exemple : `y=++x;` est équivalent à `x=x+1; y=x;` (incrémentation puis évaluation)
- Il peut également opérer sur plusieurs expressions
  - Exemple : addition de plusieurs expressions : `y+=z+d;` est équivalent à `y=y+z+d;`

Opérateur -

- On peut l'utiliser pour donner un signe négatif à une expression (une valeur ou une variable). Exemple `x=-x;` Ici, `x` prend la valeur de son opposé
- Il peut opérer en décrémentation : `x--;` et `--x;` sont équivalentes à `x=x-1;`
- Il peut opérer sur une seule expression :
  - `x--;` produit une décrémentation de `x` après son traitement
    Exemple : `y=x--;` est équivalent à `y=x; x=x-1;` (évaluation puis décrémentation)
  - `--x;` permet de décrémenter `x` avant son traitement
    Exemple : `y=--x;` est équivalent à `x=x-1; y=x;` (décrémentation puis évaluation)
- Il peut également opérer sur plusieurs expressions
  - Exemple : soustraction : `y-=z;` est équivalent à `y=y-z;`

Opérateurs

Opérateur `*`

- C'est l'opérateur de la multiplication classique.
  - Exemple : multiplication : `x*=y;` est équivalent à `x=x*y;`
- Dans une déclaration, il permet de créer un pointeur
  Exemple : pointeur `int *pointeur`

Opérateur `/`

- C'est l'opérateur de la division classique.
  - Exemple : division : `x/=y;` est équivalent à `x=x/y;`

Opérateur `%`

- Permet de calculer le reste d'une division euclidienne (modulo)
  - Exemple : modulo : `int x=303, y=4, z; z=x%y;` donne la valeur du reste de la division de `x/y` dans `z`

Opérateurs à résultat booléen

En C, toute valeur non-nulle (la valeur 1, par convention) est considérée "vrai", alors que 0 est considéré comme "faux"

Opérateurs de relation

opérateur	signification	exemple
==	est égale à	7==7
!=	est différente de	5!=7
<	est strictement inférieure à	5<7
>	est strictement supérieure à	5>7
<=	est inférieure (au sens large) à	5<=7
>=	est supérieure (au sens large) à	5>=7

Opérateurs logiques (algèbre de Boole)

opérateur	signification	exemple
!	négation logique	`int x=2, y=4, z; z=!((y>x)&&(y*x>7));`
&&	ET logique	`int x=2, y=4, z; z=((y>x)&&(y*x>7));`
||	OU logique	`int x=2, y=4, z; z=!((y>x)||!(y*x>7));`

Opérateurs combinatoires (sur les bits)

Opérateurs combinatoires (sur les bits)

opérateur	signification	exemple
~	Calcule le complément à 1	`unsigned short x=7, y; y=~x;` 7=0000000000000111 alors `y` prend la valeur 65528 = 1111111111111000
&	ET logique (combinatoire)	`short i=5,j=11; i=i&j;` `i` prendra la valeur 1
|	OU logique (combinatoire)	`short i=5,j=10; i=i|j;` `i` prendra la valeur 15
^	OU exclusif (combinatoire)	`short i=5,j=11; i=i^j;` `i` prendra la valeur 14
<<	shift à gauche	`int i, j; j=4; i=j<<2;` `i` prendra la valeur 16
>>	shift à droite	`int i, j; j=4; i=j>>2;` `i` prendra la valeur 1

Shift à gauche : Affecte un multiple de puissance de 2 : `0x01<<4;` devient `0x10`
Shift à droite : Affecte un multiple de puissance inverse de 2 : `0x010>>4;` devient `0x0`

Opérateur conditionnel ternaire

Opérateur conditionnel ternaire

```
x = condition ? <expression1> : <expression2>;
```

est équivalent à

```
if (condition)
   x=<expression1>;
else
   x=<expression2>;
```

Exemples :

```
signe=x>0?1:-1;
// qui est équivalent à
if (x>0)
   sign=1;
else
   sign=-1;
```

```
est_impaire=x%2==1?1:0;
// qui est équivalent à
if (x%2==1)
   est_impaire=1;
else
  est_impaire=0;
```

Opérateur conditionnel ternaire

Exemples

```
    int i, j;
    j=4;
    i=j>>2;
    i=(i<j)?i+1:i-1;
```

```
#include <stdio.h>

int main()
{ int x=4, y=6, z;
    int i, j;
    z=x!=y;
    j=4;
    i=j<<2;
    printf("i=%d et z=%d\n",i,z);
    i=(i<=j)?i+1:i-3;
    printf("i=%d et j=%d \n",i, j);
    i+=j%i;
    z+=!((i>j)||!(j*i>7));
    printf("i=%d et z=%d \n",i, z);
    --i=i+(j>z);
    printf("%i=d et j=%d \n",i, j);

    return 0;
}
```

```
i=16 et z=1
i=13 et j=4
i=17 et z=1
17=d et j=4
```

Entrées/Sorties

Entrées/Sorties

Entrées/Sorties

Entrées/Sorties

Entrées/Sorties

Entrées/Sorties

Entrées/Sorties

Entrées/Sorties

Entrées/Sorties

Entrées/Sorties

Entrées/Sorties

Entrées/Sorties

4. Structures de contrôle

Structures de contrôle

Structures de contrôle

- Les structures de contrôle offrent beaucoup de possibilités au programmeur
- Facilite sa tâche et permet de rendre le code (le programme) plus lisible
- Exemple : on doit exécuter la même tâche 100 fois. Il est donc plus profitable d'utiliser des structures itératives qu'écrire 100 instructions équivalentes !

- Exemple : on veut calculer la somme suivante : 1*1+2*2+3*3+...+100*100
  - Solution 1 : écrire les 100 termes de la suite ! (c'est idiot !)
  - Solution 2 : utiliser une structure de contrôle itérative ou répétitive

Structures de contrôle de base

Les **boucles** permettent de répéter une série d'instructions

- Boucle `while`
- Boucle `do --- while`
- Boucle `for`

Les **structures de branchement conditionnel** permettent de déterminer quelles instructions seront exécutées et dans quel ordre

- Branchement `if`
- Branchement `if --- else`
- Branchement multiple `switch`

Les **structures de branchement non conditionnel**

- Branchement `break`
- Branchement `continue`
- Branchement `goto`

Structures de contrôle : while

Structures de contrôle : while

La structure while permet d'exécuter une boucle itérative et elle est utilisée selon la forme syntaxique suivante :

```
while (condition) {instruction1; instruction2; instructionN;};
```

- Tant que la condition est vérifiée, les instructions sont exécutées
- Si la condition n'est pas vérifiée au départ, aucune instruction sera exécutée
- Exemple : calculer la somme 1*1+2*2+3*3+...+100*100

```
#include <stdio.h>
int main()
{
    int S=0, i=1;
    while(i<=100) {S+=i*i; i++;};
    printf("La somme S = %d\n", S);
    return 0;
}
```

Structures de contrôle : do --- while

La structure `while` équivalente, d'un point de vue algorithmique, à la structure (faire — tant que). Elle est utilisée selon la forme syntaxique suivante :

```
do {instruction1; instruction2; instructionN;} while (condition);
```

Les instructions sont toujours exécutées au moins une fois même si la condition n'est pas vérifiée au départ.

- L'algorithme suivant résout le problème : calculer la somme 1*1+2*2+3*3+...+100*100
  - S, i, entiers ;
  - S $\leftarrow$ 0 ; i $\leftarrow$ 1 ;
  - Faire S $\leftarrow$ S + i^2 ; i $\leftarrow$ i+1 ; tant que (i$\leq$100)
- et en C

```c
#include <stdio.h>
int main()
{
    int S=0, i=1;
    do {S+=i*i; i++;} while (i<=100);
    printf("La somme S = %d\n", S);
    return 0;
}
```

Structures de contrôle : for

La structure `for` permet d'exécuter une boucle itérative

```
for (expression1_init; expression2_cond; expression3_update;)
{instruction1; instruction2; instructionN;}
```

- `expression1_init` : initialisation de la variable
- `expression2_cond` : condition à vérifier pour la variable
- `expression3_update` : changement de la variable

Une version équivalente en utilisant la structure `while` :

```
expression1
while (expression2)
   { instruction1; instruction2; instructionN;
     expression 3;
   }
```

Structures de contrôle : for

Exemple

Exemple : calculer la somme 1*1+2*2+3*3+...+100*100

```
#include <stdio.h>
int main()
{
   int S=0,i;
   for(i=1;i<=100;i++)
      S+=i*i;
   printf("La somme S = %d\n", S);
   return 0;
}
```

Les trois expressions utilisées dans une boucle `for` peuvent être constituées de plusieurs expressions séparées par des virgules.

```
#include <stdio.h>
int main()
{
   int S,i;
   for(i=1,S=0; i<=100; S+=i*i,i++);
   printf("La somme S = %d\n", S);
   return 0;
}
```

Structures de contrôle : Exercice

Exercice : Écrire un programme pour saisir au clavier un entier entre 1 et 10 en utilisant la structure `while`, la structure `do --- while`, et la structure `for`

structure `for`

```
#include <stdio.h>
int main()
{
   int S,i;
   for(printf("Entrer un entier entre 1 et 10:"), scanf("%d",&i);
     (i<=0||i>10);
        printf("\nEntrer un entier entre 1 et 10:"), scanf("%d",&i))
             ;
   printf("i = %d\n", i);
   return 0;
}
```

Structures de contrôle : Exercice

structure do --- while

```
#include <stdio.h>
int main()
{
   int i;
   do {
        printf("\nEntrer un entier entre 1 et 10:");
        scanf("%d",&i);
        }  while ((i<=0)||(i>10));
   printf("Entier saisi est = %d\n", i);
   return 0;
}
```

structure while

```
#include <stdio.h>
int main()
{
   int i;
   printf("\nEntrer un entier entre 1 et 10:"); scanf("%d",&i);
   while ((i<=0)||(i>10))
        {
              printf("\nEntrer un entier entre 1 et 10:");
              scanf("%d",&i);
        }
   printf("Entier saisi est = %d\n", i);
   return 0;
}
```

Structures de contrôle : if

Structures de contrôle : if

La structure `if` est une structure conditionnelle permettant d'exécuter une instruction ou plusieurs si une condition est vérifiée. Elle est utilisée selon la forme syntaxique suivante :

```
if (condition) {instruction1; instruction2; instructionN;};
```

Exemple : valeur absolue d'un entier saisi par l'utilisateur

```
#include <stdio.h>
int main()
{
    int i;
    printf("Entrer un entier:");
    scanf("%d",&i);
    if(i<0) i=-i;
    printf("valeur absolue est %d\n", i);
    return 0;
}
```

Structures de contrôle : if --- else

La structure if --- else est une structure conditionnelle permettant d'exécuter une instruction ou plusieurs si une condition est vérifiée. Sinon, elle exécute d'autres instructions selon la forme syntaxique suivante :

```
if (condition) {instruction1; instruction2; instructionN;}
else {instruction1; instruction2; instructionK;};
```

Exemple : afficher la nature du entier saisi par l'utilisateur

```
#include <stdio.h>
int main()
{
    int i;
    printf("Entrer un entier:");
    scanf("%d",&i);
    if(i<0) printf("C'est un entier negatif\n");
    else printf("C'est un entier positif ou null\n");
    return 0;
}
```

Structures de contrôle : if --- else if --- else

```
if (condition1) {instruction1; instruction2; instruction N1;}
else if (condition2) {instruction1; instruction2; instructionN2;}
else if (condition3) {instruction1; instruction2; instructionN3;}
---------
else {instruction1; instruction2; instructionK;};
```

Exemple : afficher la nature du entier saisi par l'utilisateur

```
#include <stdio.h>
int main()
{
   int i;
   printf("Entrer un entier:");
   scanf("%d",&i);
   if(i<0) printf("C'est un entier negatif\n");
   else if (i==0) printf("C'est un zero\n");
   else printf("C'est un entier trictement positif\n");
   return 0;
}
```

Structures de contrôle : switch

La structure switch est une structure conditionnelle permettant d'exécuter une instruction ou plusieurs suivant la valeur prise par une variable.

```
switch (variable)
   {case val1 : instruction1; break;
    case val2 : instruction2; break;
    default : instructionN;}
```

Si aucune des valeurs ne correspond, alors l'instruction par défaut qui sera exécutée.

Exemple : saisir au clavier un entier et vérifier ensuite si l'entier est un multiple de 5

```
#include <stdio.h>
int main()
{
   int i,j;
   printf("Saisir un entier:");
   scanf("%d",&i);
   j=i%5;
   switch(j)
   { case 1 : printf("i=5n+1\n"); break;
     case 2 : printf("i=5n+2\n"); break;
     case 3 : printf("i=5n+3\n"); break;
     case 4 : printf("i=5n+4\n"); break;
     default : printf("c'est un multiple de 5\n"); break;
   }
   return 0;
}
```

Structures de contrôle : break

Structures de contrôle : break

La structure break permet de quitter une boucle sans avoir terminé toutes ses instructions et de passer à la première instruction qui suit la boucle.

Elle peut être utilisée dans toutes les structures de contrôle précédentes.

Exemple : calculer la somme 1*1+2*2+3*3+...+100*100 en utilisant la structure for et if --- else

```
#include <stdio.h>
int main()
{
    int S=0,i;
    for(i=1;;i++)
       { if (i<=100) S+=i*i; else break;}
    printf("La somme S = %d\n", S);
    return 0;
}
```

Structures de contrôle : continue

La structure `continue` permet de quitter une itération dans une boucle et passer à l'itération suivante sans avoir terminé toutes les instructions de l'itération quittée.

Elle peut être utilisée dans toutes les structures de contrôle itératives précédentes.

Exemple : calculer la somme 2*2+4*4+...+100*100 en utilisant la structure `for` et `if`

```
#include <stdio.h>
int main()
{
    int S=0,i;
    for(i=1;i<=100;i++)
        {
          if (i%2!=0) continue;
        S+=i*i;
        }
    printf("La somme S = %d\n", S);
    return 0;
}
```

Structures de contrôle : goto

La structure goto permet d'effectuer un saut jusqu'à l'instruction étiquette correspondante.

Elle peut être utilisée partout, mais ...

Exemple : Déterminer PPCM (plus petit commun multiple) de deux entiers saisis par l'utilisateur

```
#include <stdio.h>
int main()
{
    int i,j,max,ppcm;
    printf("\nSaisir le premier un entier:");
    scanf("%d",&i);
    printf("\nSaisir le deuxiem entier:");
    scanf("%d",&j);
    max =i;
    if (j>max) max =j;
    for(ppcm=max;;ppcm+=max)
        {
        if ((ppcm%i==0)&&(ppcm%j==0)) goto affichage;
        }
    i+=j;
    affichage: printf("Le ppcm de %d et %d est %d\n", i,j,ppcm);
    return 0;
}
```

5. Fonctions et procédures

Fonctions

Fonctions

- Les fonctions et procédures servent à rendre les programmes plus modulaires et plus lisibles
- Elles permettent aussi de les rendre plus compacts
- Une fonction / procédure peut être appelée plusieurs fois dans un programme

```
<type> Nom_fonction(<type1> arg1, <type2> arg2, <typeN> argN)
// paramètres formels arg1, arg2, argN
{
   instruction1;
   instruction2;
   instructionN;

   return val;
}
```

Une fonction retourne une valeur via `return`. l'effet de `return` sur le reste des instructions d'une fonction est celui d'un `break` dans une boucle

```
/* la fonction surface délivre un int,
   ses paramètres formels sont l'int largeur et l'int longueur */
int surface(int largeur, int longueur)
{
   int res;        /* déclaration des variables locales */
   res=largeur*longueur;
   return res;     /* retourne à l'appelant en délivrant res */
}
```

Fonctions

```c
#include <stdio.h>

int surface(int largeur, int longueur)
{
   int res;
   res=largeur*longueur;
   return res;
}

void main()
{
   int l,L,surf;
   printf("\n Entrer la largeur:"); scanf("%d", &l);
   printf("\n Entrer la longueur:"); scanf("%d", &L);
   surf=surface(l,L);   // appel de la fonction surface
   printf("\n Surface = %d\n",surf);
}
```

Procédures

```
void Nom_procedure(<type1> arg1, <type2> arg2, <typeN> argN)
// paramètres formels arg1, arg2, argN
{
   instruction1;
   instructionN;
}
```

Une procédure est une fonction qui ne retourne pas de valeur. Elle peut modifier éventuellement la valeur de certains paramètres transmis par adresse.

```
void surface(int largeur, int longueur, int *res)
{
   *res=largeur*longueur;
}
```

```
#include <stdio.h>

void surface(int largeur, int longueur, int *res)
{
   *res=largeur*longueur;
}
void main()
{
   int l,L,surf;
   printf("\n Entrer la largeur:"); scanf("%d", &l);
   printf("\n Entrer la longueur:"); scanf("%d", &L);
   surface(l,L, &surf);   // appel de la fonction surface
   printf("\n Surface = %d\n",surf);
}
```

Appel d'une fonction/procédure

- Une fonction peut être déclarée grâce à un prototype (une entête) sans indiquer ses instructions. On peut dans ce cas appeler la fonction avant sa déclaration complète
- Appel d'une fonction : `Variable = Nom_fonction(arg1, arg2, argN);`
- Appel d'une procédure : `Nom_procedure(arg1, arg2, argN);`
- Une fonction/procédure peut ne contenir aucun paramètre formel (argument)

```
void affichage () {printf("\n Entrer une valeur: ");}
```

```
double pi() { return(3.14159); }
```

```
#include <stdio.h>
int surface(int largeur, int longueur);
void message();

void main()
{
   int l,L,surf;
   message(); scanf("%d", &l);
   message(); scanf("%d", &L);
   surf=surface(l,L);      /*appel avant déclaration de la fonction surface*/
   printf("\n Surface = %d\n",surf);
}

int surface(int largeur, int longueur)
{
   return largeur*longueur;
}
void message() { printf(" \n Saisir une valeur:");}
```

Variables globales

Variables globales

- Une variable globale est une variable déclarée en dehors de toutes fonctions (entête du programme)
- Les variables globales sont statiques ou permanentes (une variable permanente occupe un emplacement en mémoire qui reste le même durant tout l'exécution du programme)
- Les variables permanentes sont initialisées à zéro par le compilateur

```c
#include <stdio.h>

int n;  // Variable globale

void Appel()
{
   n++;
   printf("Appel numero %d\n",n);
}

int main()
{
   int i;   // Variable locale
   printf("Valeur initiale de n = %d\n",n);
   for (i = 0; i < 5; i++) Appel();
   return 0;
}
```

Variables locales

- Une variable locale est une variable déclarée à l'intérieur d'une fonction/procédure
- Les variables locales sont temporaires. Quand une fonction/procédure est appelée, elle place ses variables locales. Ces variables sont dépilées et donc perdues à la sortie de la fonction/procédure

```c
#include <stdio.h>
int n;  // Variable globale
void Appel()
{
    int n=0;
    n++;
    printf("Appel numero %d\n",n);
}
int main()
{
    int i;   // Variable locale
    for (i = 0; i < 5; i++) Appel();
    return 0;
}
```

Variables locales statique

```
static <type> variable_locale;
```

- La valeur de la variable locale statique est conservée d'un appel au suivant.
- Les variables statiques sont initialisées à zéro par le compilateur

```
#include <stdio.h>
int n;  // Variable globale
void Appel()
{
   static int n;
   n++;
   printf("Appel numero %d\n",n);
}
int main()
{
   int i;   // Variable locale
   printf("Valeur initiale de n = %d\n",n);
   for (i = 0; i < 5; i++) Appel();
   return 0;
}
```

Passage d'arguments par valeur / par adresse

Passage d'arguments

Pour appeler une fonction, on doit indiquer ses arguments. Après l'exécution de la fonction, ces arguments peuvent changer sous l'effet de ses instructions,
Deux modes de passage existent :

- Passage d'arguments par valeur : les arguments sont des valeurs et ne changent pas après l'exécution de la fonction
- Passage d'arguments par adresse : les arguments sont des adresses et changent sous l'effet des instructions de la fonction

Passage d'arguments par valeur / par adresse

Passage d'arguments par valeur

```
#include <stdio.h>
void surface_effective
   (float largeur, float longueur)
{
  float res;
  largeur*=0.9;
  longueur*=0.8;
  res=largeur*longueur;
  printf("surface = %f\n",res);
}

void main()
{
  float l,L,surf;
  printf("\n Entrer la largeur: ");
  scanf("%f", &l);
  printf("\n Entrer la longueur:");
  scanf("%f", &L);
  surface_effective(l,L);
  printf("largeur = %f\n",l);
  printf("longueur = %f\n",L);
}
```

Passage d'arguments par adresse

```
#include <stdio.h>
void surface_effective
   (float *largeur, float *longueur)
{
  float res;
  *largeur*=0.9;
  *longueur*=0.8;
  res=*largeur*(*longueur);
  printf("surface = %f\n",res);
}

void main()
{
  float l,L,surf;
  printf("\n Entrer la largeur: ");
  scanf("%f", &l);
  printf("\n Entrer la longueur:");
  scanf("%f", &L);
  surface_effective(&l,&L);
  printf("largeur = %f\n",l);
  printf("longueur = %f\n",L);
}
```

Récursivité

Récursivité

Une fonction est dite récursive si on fait appel à cette même fonction dans le bloc de sa définition.

Exemple : la fonction factorielle est définie de la manière suivante :

$$\texttt{fact}(0) = 1 \text{ et } \texttt{fact}(n) = n \times \texttt{fact}(n-1) \text{ pour } n > 0$$

```
#include <stdio.h>
int fact(int n)
{
   if(n==0) return 1;
   else return (n*fact(n-1));
}
void main()
{
   printf("fact(7)=%d\n",fact(7));
}
```

Exemple : Calculer x^n par récursivité

```
#include <stdio.h>
int puissance(float x, int n)
{
   if(n==0) return 1;
   else return (x*puissance(x,n-1));
}
void main()
{
   printf("Puissance(2,2)=%f\n",puissance(1.5,2));
}
```

Récursivité

Inconvénient de la récursivité : temps de calcul énorme (car besoin de tout recalculer à chaque étape)

```
#include <stdio.h>
int premier(int n)
{
   if(n==2) return 1;
   else
      {
         int k=2;int z=1;
         while(k<n)
           {
              while ((premier(k)==0)&&(k<n-1)) {k++;};
              if (n%k==0) {z=0;k++;} else k++;
           };
         return z;
      };
}
void main()
{
   int i;
   for(i=2;i<100;i++) printf("premier(%d)=%d\n",i,premier(i));
}
```

Fonctions mathématiques

Fonctions mathématiques

- La plupart des compilateurs en C disposent de bibliothèques mathématiques avec de nombreuses fonctions prédéfinies :
- Pour pouvoir les utiliser, on peut inclure la librairie `<math.h>` pour certaines de ces fonctions. D'autres fonctions de la librairie `<stdlib.h>` peuvent être employées,

Fonction	librairie	description
fabs	`<math.h>`	Renvoie la valeur absolue d'une valeur flottante
exp	`<math.h>`	Renvoie l'exponentielle de la valeur d'entrée
cos	`<math.h>`	Renvoie le cosinus de la valeur d'entrée
sin	`<math.h>`	Renvoie le sinus de la valeur d'entrée
acos	`<math.h>`	Renvoie l'arccosinus
asin	`<math.h>`	Renvoie l'arcsinus
ceil	`<math.h>`	Renvoie la partie entière supérieure d'un réel
log	`<math.h>`	Renvoie le logarithme
min	`<stdlib.h>`	Renvoie le minimum de deux expressions
max	`<stdlib.h>`	Renvoie le maximum de deux expressions
pow	`<math.h>`	Renvoie la puissance de la forme x^n
rand	`<stdlib.h>`	Génère aléatoirement un entier entre 0 et la valeur d'entrée

main

main

- Fonction/procédure `main` sans paramètre formel :
  - `void main (void)`
  - `int main () return 0;`
    La valeur de retour 0 correspond à une terminaison correcte EXIT_SUCCESS
  - `int main () return 1;`
    La valeur de retour 0 correspond à une terminaison sur une erreur EXIT_FAILLURE
- La fonction main peut être possédée des paramètres formels. Un programme C peut recevoir une liste d'arguments au lancement de son exécution
- La ligne de commande qui sert à lancer le programme est composée du nom du fichier exécutable suivi par des paramètres

main

```
int main (int argc, char *argv[]) { return 0; /* ou 1 */ }
```

- `argc` (argument count) : variable de type `int` donc la valeur est égale au nombre de mots composant la ligne de commande (y compris le nom de commande)
- `argv` (argument vector) : tableau de chaines de caractères correspondant chacune à un mot de la ligne de commande
- `argv[0]` contient le nom de la commande (du fichier exécutable)
- `argv[1]` contient le premier paramètre
- `argv[2]` contient le deuxième paramètre

Exemple 1

Exemple 1

Exemple 1 : Calculer la somme : 1*1+2*2+3*3+...+N*N. La valeur de N est déterminée à partir d'un autre programme (l'élément maximal d'une suite saisie par l'utilisateur)

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

int main(int argc, char *argv[])
{
   int a;
   printf("\nNombre de mots de commande =%d",argc);
   if (argc != 2)
   {
      printf("\nErreur : nombre invalide d'arguments");
      printf("\nUsage: %s int int\n",argv[0]);
      return(1);
   }
   printf("\n argc= %d; argv[0]= %s; argv[1]=%s\n", argc,argv[0],argv[1]);
   a = atoi(argv[1]);// Convertir en entier
   int S=0, i=1;
   while(i<=a) {S+=i*i; i++;};
   printf("La somme (n=%d): S = %d\n",a, S);
   return(1);
}
```

Exemple 2

Exemple 2 : Programme pour saisir une suite d'entier, chercher ensuite le maximale et finalement calculer la somme en appelant le programme Calcul.exe

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

int main()
{
    int i,n,j,k=0;
    printf("Entrer le nombre elements de la suite:");
    scanf("%d",&n);
    for (i=0;i<n;i++)
    {
        printf("Entrer %d ieme entier: ",i+1);
        scanf("%d",&j); if (j>k) k=j;
    }
    char chaine[6];
    sprintf(chaine,"%d",k);  // convertir un entier en chaine de caractere
    execl("C:\\Code", "Calcul.exe", chaine, NULL);
     // execl execute "Calcul.exe" dans "C:\Code", avec l'argument "chaine"
     // (char*)0 ou NULL pour indiquer la fin de la liste des arguments
    return 0;
}
```

Fonctions génériques

Fonctions génériques

Astuce : écrire des fonctions génériques (sans préciser les types)

```c
#include <stdio.h>
#define maxi(a,b) (a>b)?a:b

int main()
{
   float l=2,L=4,m;
   int i=2,j=9,mij;
   char ca='r',cb='Z',mc;

   m=maxi(l,L);
   mij=maxi(i,j);
   mc=maxi(ca,cb);

   printf("maxi_float = %f\n",m);
   printf("maxi_int = %d\n",mij);
   printf("maxi_char = %c\n",mc);
   return 0;
}
```

6. Tableaux

Généralités sur les tableaux

Généralités sur les tableaux

- Les tableaux sont des structures pratiques et nécessaires pour traiter plusieurs variables du même type et pour les ranger.
- En C, on peut définir des tableaux pour tous les types de base :
  entier (`int`), réel (`float`, `double`), caractère (`char`), ...
- Type de tableaux
  - Tableaux à une dimension
  - Tableaux multi-dimentionnels
  - Chaînes de caractères

Tableaux à une dimension

Tableaux à une dimension

Tableau

```
<type> tableau[Nombre_elements];
```

- La taille d'une case du tableau est le nombre d'octets sur lequel la donnée est codée
- En C, l'indexation d'un tableau commence à partir de 0

Exemple : `int tab[5];`

- `tab` est un pointeur constant (non modifiable) dont la valeur est l'adresse du premier élément du tableau. Autrement dit, `tab` a pour valeur `&tab[0]` :
  - `tab` est équivalent à `&tab[0]` : adresse
  - `tab+i` est équivalent à `&tab[i]` : adresse
  - `*(tab+i)` est équivalent à `tab[i]` : valeur !
- Le tableau `tab` est distribué sur 5 adresses de mémoire consécutives dont chacune occupe 4 octets de mémoire

```
#include <stdio.h>
void main()
{
   char tab[]={'2', '5', '7', 'r'};
   int i;
   printf("\nTAILLE = %d",sizeof(tab));
   printf("\nAdresse de memoire de tab[0]: %d, tab[1]:%d",&tab[0], &tab[1]);
   printf("\nAdresse de memoire du pointeur tab: %d, tab+2: %d\n",&tab[0], &tab[2]);
   for (i=0;i<4;i++)
      printf("\nValeur de tab[%d]= %c, de *(tab+%d)=%c ",i,tab[i],i, *(tab+i));
}
```

Tableaux à une dimension

Initialisation

Initialisation d'un tableau :

- Lors de sa déclaration :
  - `char tab[4]={'2', '5', '7', 'r'};` permet de préciser la taille
  - `char tab[]={'2', '5', '7', 'r'};` sans préciser la taille
  - `char tab[10]={'2', '5', '7', 'r'};` permet d'initialiser que les 4 premiers éléments
  - `char tab[10]={[4] = '1',[7]= '8'};` initialiser que le $5^{ème}$ et le $8^{ème}$ éléments

- Après sa déclaration, en utilisant une boucle (`for`, `while`, ...) :

```
#include <stdio.h>
void main()
{
    int tab[10]={0,1,4,9,16};
    int i;
    for(i=5;i<10;i++) tab[i]=i*i;
    for(i=0;i<10;i++)
       printf("tab[%d]=%d\n",i,tab[i]);
}
```

Tableaux à une dimension

Pointeurs et tableaux

Pointeurs et tableaux

- Le langage C autorise aussi la déclaration d'un tableau sans spécifier sa taille en utilisant un pointeur :

Tableau

```
<type> *tableau;
```

Exemple : `int *point;`

- `point` est un pointeur et la zone mémoire affectée est identique à celle de `tab` dans l'exemple suivant :

```
#include <stdio.h>
void main()
{
   char tab[]={'y', '5', '7', 'r'};
   char *point;
   point=tab;
   printf("point[3] = %c\n",point[3]);
}
```

Tableaux à une dimension

Accès

Accès :
En C, on accède au (k+1)$^{\text{ème}}$ élément du tableau `tab` par l'expression `tab[k]`. Cette expression représente une variable indépendante et se manipule comme les autres variables du même type que le tableau `tab` (affectation, lecture, affichage, ...)

```c
#include <stdio.h>
void main()
{
    char tab[7];
    int i;
    for(i=0;i<=6;i++)
    {
        tab[i]=(char)i+100;
        printf("tab[%d]=%c\n",i,tab[i]);
    }
}
```

Tableaux multi-dimensionels

Tableaux multi-dimensionels

- La déclaration d'un tableau multi-dimensionnel est similaire à celle d'un tableau à une dimension :

```
<type> tableau[N1][N2][Nn];
```

- Les déclarations suivantes sont aussi autorisées :

```
<type> tableau[][N2][Nn];
```

```
#include <stdio.h>
void main()
{
    int t[][2]={1, 2, 3, 4};
    int i,j;
    t[2][0]=5;
    t[2][1]=5;
    for(i=0;i<2;i++)
    {
        for(j=0;j<2;j++) printf("t[%d][%d]=%d\n",i,j,t[i][j]);
    }
}
```

Tableaux multi-dimensionels

Exercice

Exercice :
Gestion d'une classe : vous avez N étudiants (N est fourni par l'utilisateur) et vous devrez saisir leurs notes du médian, du partiel et du final. Vous devrez calculer pour chaque étudiant sa moyenne selon la pondération (30%, 25%, 45%). Vous devrez ensuite afficher la liste des étudiants (avec leurs moyennes respectives) dans l'ordre décroissant.
Écrire le programme nécessaire.

Tableaux et fonctions/procédures

Tableaux et fonctions/procédures

Tableaux et fonctions :
Les éléments d'un tableau peuvent être considérés comme des variables indépendantes (entrées/sorties) et on peut les utiliser comme les autres variables dans une fonction/-procédure

```c
#include <stdio.h>
int somme(int k, int tab[]) // ou encore int somme(int k, int *tab)
{
   int i, s=0;
   for(i=0;i<k;i++)
      { s+=tab[i];tab[i]+=1;}
   return s;
}
int main()
{
   int i, tableau[5]={1, 2, 3, 4, 5};
   printf("Tableau entre:");
   for(i=0;i<5;i++)
      printf(" %d ",tableau[i]);
   printf(". Somme =%d \n",somme(5,tableau));
   printf(" Tableau sorti:");
   for(i=0;i<5;i++)
      printf(" %d ",tableau[i]);
   printf(". Somme =%d \n",somme(5,tableau));
   return 0;
}
```

Tableaux et fonctions/procédures

Exercice

Exercice : Ecrire une fonction/procédure pour saisir un tableau d'entiers de taille N (la valeur de N sera demandée à l'utilisateur). Ecrire une fonction/procédure pour trier par ordre croisant ce tableau.

```c
#include <stdio.h>
#include <stdlib.h>
int *saisir(int *k)
{
   int *res, i;
   printf("Saisir le nombre des
      elements de la suite entiers:");
   scanf("%d",k);
   res = (int*)malloc(*k*sizeof(int));
   printf("\n Saisir une suite entiers:"
        );
   for(i=0;i<*k;i++) scanf("%d",&res[i])
        ;
   return res;
}
void tri(int k, int *tab)
{
   int i, j, aux;
   for(i=0;i<k-1;i++)
      {
      for(j=i+1;j<k;j++)
         {
         if(tab[i]>tab[j])
            {
               aux=tab[i];
               tab[i]=tab[j];
               tab[j]=aux;
            }
 } } }
```

```c
int main()
{
   int *ta;
   int k=1,i;
   ta=saisir(&k);
   printf("\nTableau entre:");
   for(i=0;i<k;i++)
      printf(" %d ",ta[i]);
   tri(k,ta);
   printf("\nTableau trie:");
   for(i=0;i<k;i++)
      printf(" %d ", ta[i]);
   return 0;
}
```

Chaînes de caractères

Chaînes de caractères

- Les chaînes de caractères représentent des tableaux de caractères qui se terminent par le caractère `NULL` ou `\0`,

u	n		e	x	e	m	p	l	e	\0

- Prendre l'habitude de réserver une case pour le caractère NULL et de s'assurer de l'affectation de cette valeur dans la bonne case
- Le traitement des fonctions sur les chaînes de caractères en dépend. Ce caractère marque la fin de la chaîne.
- Il n'a plus besoin de connaître la taille de la chaîne car grâce à la marque de fin `\0`, on peut savoir à quel moment on doit stopper un traitement.
- Attention : en conséquence, si vous affectez ce caractère au milieu de la chaîne, vous cassez votre chaîne car les éléments qui suivent deviennent inaccessible pour des fonctions comme `printf`.
- Dans certains cas, le compilateur ajoute automatiquement le caractère `NULL`.

Chaînes de caractères

```
#include <stdio.h>
void main()
{
    char chaine1[20];
    char *chaine2="et voila un deuxième exemple";
    int i;

    for(i=0;i<19;i++)
        chaine1[i]=i+100;
    chaine1[i]='\0';

    printf("chaine1 = %s\n", chaine1);
    printf("chaine2 = %s\n", chaine2);

    chaine1[10]='\0';
    printf("chaine1 = %s\n", chaine1);

}
```

Fonctions prédéfinies sur les chaînes de caractères

Fonction	librairie	description
strlen	`<string.h>`	Renvoie la longueur de la chaîne
strcpy	`<string.h>`	Copie une chaîne dans une autre
strcat	`<string.h>`	Permet d'ajouter une expression à une chaîne
strcmp	`<string.h>`	Compare deux chaînes
atof	`<stdlib.h>`	Convertit la chaîne en valeur du type float
atol	`<stdlib.h>`	Convertit la chaîne en valeur du type long int
atoi	`<stdlib.h>`	Convertit la chaîne en valeur du type int
isalpha	`<ctype.h>`	Vérifie si le caractère est un chiffre
isprint	`<ctype.h>`	Vérifie si le caractère est imprimable
ispunct	`<ctype.h>`	Vérifie si le caractère est un signe de ponctuation
isascii	`<ctype.h>`	Vérifie si le caractère est ASCII (son code est entre 0 et 127)

Tableaux et chaînes de caractères

Déterminer la longueur

Déterminer la longueur :

- Pour un tableau, utiliser la fonction `sizeof()`
- Pour une chaine de caractère, utiliser la fonction `strlen()`, sans oublier de déclarer la librairie `<string.h>`

7. Types composées

Structures

Structures

- En plus de ses types standards, le langage C offre la possibilité de créer les types composés : les **structures**.
- Une structure est une suite finie d'objets de types différents.
- Chaque élément de la structure, appelé **membre** ou **champ**, est désigné par un identificateur
- Une structure peut être déclarée selon le schéma suivant :

Structure

```
struct Nom_structure{<type1> var1; <type2> var2; <typeN> varN;};
```

```
struct etudiant
{
    char *nom;
    float note_median;
    float note_final;
    float moyenne;
};
```

Remarque : Les différents éléments d'une structure n'occupent pas nécessairement des zones contiguës en mémoire.

Structures

Variable structurée

```
struct Nom_structure Variable;
```

Pour accéder à une composante de la structure, on utilise l'opérateur ·

```
#include <stdio.h>
struct etudiant
   {
      char *nom;
      float note_median;
      float note_final;
      float moyenne;
   };

int main()
{
   struct etudiant Dupont, Ami;
   Dupont.nom="DUPONT";
   Dupont.note_median=17;
   Dupont.note_final=8.5;
   Dupont.moyenne=0.4*Dupont.note_median+0.6*Dupont.note_final;
   Ami=Dupont;
   Ami.nom="RAYAN";
   printf("NOM : %s Moyenne : %1.2f\n",Dupont.nom, Dupont.moyenne);
   printf("NOM : %s Moyenne : %1.2f\n",Ami.nom, Ami.moyenne);
   return 0;
}
```

Structures

```
struct Nom_structure Variable;
```

Pour accéder à une composante d'un pointeur sur la structure, on utilise l'opérateur ->

```
#include <stdio.h>
struct etudiant
   {
      char *nom;
      float note_median;
      float note_final;
      float moyenne;
   };
void modifier_notes(float a,float b, struct etudiant *etu)
{
   etu->note_median=a; etu->note_final=b; etu->moyenne=0.4*a+0.6*b;
}
int main()
{
   struct etudiant Dupont={"DUPONT", 17, 8.5, 0};
   struct etudiant Ami;
   Dupont.moyenne=0.4*Dupont.note_median+0.6*Dupont.note_final;
   Ami=Dupont; Ami.nom="RAYAN";
   modifier_notes(9,14,&Ami);
   printf("NOM : %s Moyenne : %1.2f\n",Dupont.nom, Dupont.moyenne);
   printf("NOM : %s Moyenne : %1.2f\n",Ami.nom, Ami.moyenne);
   return 0;
}
```

Types personnalisés

Types personnalisés

- Le langage C permet aussi de créer ses propres types personnalisés.

```
typedef <type> NouveauType;
```

- Exemple : `typedef int entier;` Dans ce cas, `entier` est défini comme le type `int`

```
#include <stdio.h>
typedef int entier;
int main()
{
   entier test;
   printf("\n Saisir un nombre entier:");
   scanf("%d",&test);
   printf("Nombre entier saisi=%d\n",test);
   return 0;
}
```

Structures et types personnalisés

- On peut utiliser `typedef` pour la définition d'une structure

```
typedef struct { <type1> var1; <type2> var2; <typeN> varN;
} Nom_structure;
```

- Définition d'une variable structurée dans ce cas est plus simple : `Nom_structure Variable;`

```
#include <stdio.h>
struct etudiant
      {
         char *nom;
         float note_median;
         float note_final;
         float moyenne;
      };
int main()
{
   struct etudiant Dupont, Ami;
   Dupont.nom="DUPONT";
   Dupont.note_median=17;
   Dupont.note_final=8.5;
   Dupont.moyenne=0.4*Dupont.note_median+0.6*Dupont.note_final;
   Ami=Dupont;
   Ami.nom="RAYAN";
   printf("NOM : %s Moyenne : %1.2f\n",Dupont.nom, Dupont.moyenne);
   printf("NOM : %s Moyenne : %1.2f\n",Ami.nom, Ami.moyenne);
   return 0;
}
```

Structures et types personnalisés

Définition et utilisation d'une structure : résumé

Il y a deux définitions possibles

```
struct Nom_structure
      {
         <type1> var1;
         <type2> var2;
         <typeN> varN;
      };

// Définition d'une variable :
struct Nom_structure Variable;
```

```
typedef struct
        {
          <type1> var1;
          <type2> var2;
          <typeN> varN;
        } Nom_structure;

// Définition d'une variable :
Nom_structure Variable;
```

Si on ajoute une définition supplémentaire à la définition 1 (à gauche) :

`typedef struct Nom_structure Nom_structure ;`

Définition d'une variable : `Nom_structure Variable;`

Structures et types personnalisés

Exercice : Nombres complexes

Exercice : Nombres complexes

Un nombre complexe peut être représenté sous forme cartésienne ($a + ib$, où $i = \sqrt{-1}$)

1. Ecrire le type *CplxCartesien* correspondant
2. Un nombre complexe peut aussi être représenté sous forme polaire ($re^{i\theta}$).
   Ecrire le type *CplxPolaire* correspondant
3. Ecrire un type *Cplx* permettant de représenter un complexe quelle que soit sa forme.
4. Définir en C les opérations classiques s'appliquant aux variables complexes de type *Cplx* :
   - *PartieReelle*
   - *PartieImaginaire*
   - *Module*
   - *Argument*
   - *ConvertirEnPolaire*
   - *ConvertirEnCartesien*
   - *ChangerDeForme*
   - *Conjuguer*
   - *Additionner*
   - *Soustraire*
   - *Multiplier*
   - *Diviser*

Types d'énumération

Types d'énumération

- Les énumérations permettent de définir un type par la liste noms symboliques qu'il peut prendre.
- Le langage C permet aussi de créer ses propres types d'énumération.
- La commande à utiliser pour définir un type d'énumération est la suivante :

```
enum Nom_type_enumeration {expression1, expression 2, expressionN
    };
```

```
#include <stdio.h>

enum Couleur{bleu, blanc, rouge, vert, jaune};

int main()
{
   Couleur C;
   C=vert;
   printf("\nLe code du couleur = %d\n", C);
   return 0;
}
```

Types d'énumération

Chaque nom symbolique corresponds à une valeurs entière.

- Par défaut, cette valeur est égale à l'indexation nom symbolique

```
enum BOOLEEN
{
    faux,      // faux = 0, vrai = 1
    vrai
};

enum BOOLEEN mondrapeau;
```

- On peut cependant définir une valeur pour chaque nom symbolique

```
#include <stdio.h>

enum Couleur{bleu=100, blanc=20, rouge, vert=1000, jaune};

int main()
{
   Couleur C;
   C=vert;
   printf("\nLe code du couleur = %d\n", C);
   return 0;
}
```

Types d'énumération

Exemples

```
enum { oui, non } reponse;
```

```
enum JOUR
{
    samedi,
    dimanche = 0,
    lundi,
    mardi,
    mercredi, // ici, mercredi est associé à 3
    jeudi,
    vendredi
} jour_ouvrable;

enum JOUR demain = jeudi;
```

```
enum lescouleurs {pique, coeur, carreau, trèfle};

struct carte {
   enum lescouleurs couleur;
   short int valeur;
} main[13];
```

Unions

Unions

Le langage C permet aussi de créer des types spécifiques qui regroupent plusieurs membres mais en occupant la même zone mémoire : les unions.
Une variable de type `union` mémorise une seule des valeurs définies par le type

```
union Nom_union {expression1 , expression 2, expressionN};
```

8. Pointeur et allocation dynamique de la mémoire

Pointeurs

Pointeurs

- Un pointeur est un *lien*
- C'est une variable dont la valeur est l'adresse mémoire d'une autre variable
- La déclaration se fait généralement de la manière suivante :

```
<type> *Variable;
```

- `Variable` est un pointeur qui contient l'adresse mémoire de la variable `*Variable`
- Exemple : `int *x;` Dans ce cas, `x` est un pointeur sur une variable entière
- Principaux usages des pointeurs :
  - Passage des paramètres par adresse
  - Construction de structure dynamique (tableaux, listes, ...)

Pointeurs

- Exemple : `int *x; int y=7; x=&y; *x=9; //x reçoit l'adresse de y`

```
#include <stdio.h>
int main()
{
    int *x;
    int y=7;
    x=&y;
    printf("\nValeur de y = %d et son adresse = %d",y,&y);
    printf("\n Va valueur de x = %d et son adresse = %d",x,&x);
    printf("\n La valeur de la variable *x = %d",*x);
    *x=9;
    printf("\n La valeur de x = %d et son adresse = %d.
            Valeur de y = %d\n",x,&x,y);
    x++;
    printf("\n la nouvelle valueur de x = %d
            et son adresse = %d",x,&x);
    printf("\n La nouvelle valeur de la variable *x = %d",*x);
    return 0;
}
```

- un pointeur peut être incrémenté à la nouvelle adresse, on ne sait pas l'état ni le contenu de la mémoire : `int *x; int y=7; x=&y; x++;`

Pointeurs

- `NULL` : c'est le pointeur nul (adresse=0), il ne pointe sur rien. Il permet d'initialiser un pointeur mais ne réserve pas la mémoire.
- Un pointeur occupe habituellement toujours la même taille (occupe la même place en mémoire), quelque soit l'objet se trouvant à cet emplacement.
- Sur une architecture (système) 32 bits, un pointeur occupe 4 octets ; il occupe 8 octets sur une architecture 64 bits, etc.

```
#include <stdio.h>
int main()
{
   double *a, b=7.5;a=&b;
   int *x, i=5;x=&i;
   char *y, c='c';y=&c;
   printf("\nTAILLE de la variable b = %d;
   celle du poiteur a= %d",sizeof(b),sizeof(a));
   printf("\nTAILLE de la variable i = %d;
   celle du poiteur x= %d",sizeof(i),sizeof(x));
   printf("\nTAILLE de la variable c = %d;
            celle du pointeur y= %d",sizeof(c),sizeof(y));
   return 0;
}
```

Allocation dynamique de la mémoire

Allocation dynamique de la mémoire

- De quoi s'agit-il ?
  Réserver explicitement de l'espace mémoire en cours d'exécution

- Intérêts :
  - Adapter l'espace mémoire réservé aux besoins
  - Déclarer un tableau de taille variable
  - Occuper l'espace mémoire juste nécessaire
  - Créer des structures de données dynamiques

- Quand ?
  On ne connaît pas le volume de données à traiter lors de l'écriture de l'algorithme

Allocation dynamique de la mémoire

La mémoire est automatiquement réservée à la suite d'une déclaration simple (exemple : `int i;` où 4 octets sont alloués)

L'allocation dynamique permet de gérer la mémoire (réserver et libérer) par le programmeur selon le besoin du programme

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *tab;
    int i;
    tab=(int*)malloc(5*sizeof(int));
    for(i=0;i<5;i++)
    {
        printf("entrer tab[%d]:\n",i);
        scanf("%d",&tab[i]);
    }
    return 0;
}
```

Allocation dynamique de la mémoire

- Permet la maîtrise de la mémoire et l'optimisation de son utilisation
- Très utile dans le cas d'un programme nécessitant de manipuler ou de stocker une grande quantité de données
- L'allocation dynamique peut se faire par les fonctions de la librairie `<stdlib.h>`

Quatre étapes sont distinguées :

- Déclaration du pointeur qui va représenter l'adresse de la mémoire réservée
- Réservation dynamique de la mémoire (fonction `malloc`, `calloc`)
- Utilisation de la mémoire par les instructions du programme
- Libération de la mémoire allouée (fonction `free`)

Allocation dynamique de la mémoire

Syntaxe d'allocation

```
pointeur (type *) malloc (nombre-octets);
```

- `nombre-octets` est souvent donné à l'aide de la fonction `sizeof(type_variable)`. Cette fonction retourne le nombre d'octets requis pour stoker une valeur de type `type_variable`.
- La mémoire est automatiquement libérée grâce à la fonction `free`.
- Cette fonction est très utile pour mieux exploiter la mémoire pendant l'exécution du programme.

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int *tab;
    int i;
    tab=(int*)malloc(5*sizeof(int));
    for(i=0;i<5;i++)
    {
        printf("\n Entrer tab[%d]= ",i);
        scanf("%d",&tab[i]);
    }
    free(tab);
}
```

Allocation dynamique de la mémoire

La fonction `malloc`

La fonction `malloc`

- `malloc` permet de réserver une zone de mémoire dont l'adresse sera donnée par le pointeur. La taille de la mémoire allouée est l'argument de cette fonction.
- `malloc` renvoie l'adresse d'une zone mémoire en cas de succès ou 0 en cas d'échec.

Allocation dynamique de la mémoire

La fonction calloc

La fonction calloc

- calloc a le même rôle que la fonction malloc mais elle initialise en plus l'objet pointé à zéro

```
pointeur (type *) calloc (nombre_objet, taille d'un objet);
```

```c
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int *tab;
    int i;
    tab=(int*)calloc(5, sizeof(int));
    for(i=0;i<5;i++)
    {
        printf("\n Entrer tab[%d]= ",i);
        scanf("%d",&tab[i]);
    }
    free(tab);
}
```

Allocation dynamique de la mémoire

La fonction `free`

La fonction `free`

- `free` permet de libérer la mémoire qui a été allouée dynamiquement et sur laquelle le pointeur est désigné.
- Attention : ne jamais libérer une mémoire qui n'a pas été allouée dynamiquement ! ! L'adresse du pointeur ne change pas après son exécution

Allocation dynamique de la mémoire

Le pointeur NULL

Le pointeur `NULL`

- `NULL` est le pointeur nul (adresse 0). Il ne pointe sur rien. Affecter `NULL` à un pointeur ne permet pas de libérer la mémoire qui a été allouée dynamiquement !
- Attention : il s'agit simplement d'un déplacement de son adresse ! ! L'adresse du pointeur est l'adresse 0 après l'affectation

Allocation dynamique de la mémoire

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *tab;
    int i;
    int *tab2;
    tab=(int*)malloc(3*sizeof(int));
    for(i=0;i<3;i++)
    {
       printf("\n Eentrer tab[%d] =",i);
       scanf("%d",&tab[i]);
    } for(i=0;i<3;i++) printf("tab[%d]=%d\n",i,tab[i]);
    tab2=tab;
    for(i=0;i<3;i++) printf("tab2[%d]=%d\n",i,tab2[i]);
    printf("ADRESSE de tab avant affectation=%d\n",tab);
    printf("ADRESSE de tab2 avant affectation=%d\n",tab2);
    tab=NULL;
    tab2[2]=777;
    for(i=0;i<3;i++) printf("tab2[%d]=%d\n",i,tab2[i]);
    printf("ADRESSE de tab apres affectation=%d\n",tab);
    printf("ADRESSE de tab2 apres affectation=%d\n",tab2);
    return 0;
}
```

Pointeurs et tableaux

Pointeurs et tableaux

Tableaux multi dimensionnel

- Un tableaux statique, la taille de chaque dimension est fixée
- Exemple : `int tab[M][N]; //un tableaux d'entier de M lignes et N colonnes`
- Tableaux dynamique ⇒ pointeur ! !

```
type **nom-pointeur; // pour un tableau à deux dimensions
type ***nom-pointeur; // pour un tableau à trois dimensions
```

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int k=2, n=2,m=3,i;
    int **tab;
    tab = (int**)malloc(k * sizeof(int*));
       tab[0] = (int*)malloc(n*sizeof(int));
       tab[1] = (int*)malloc(m*sizeof(int));
       tab[0][0]=1;tab[0][1]=2;
       tab[1][0]=3;tab[1][1]=4;tab[1][2]=5;
    printf("\n%d %d",tab[0][0],tab[0][1]);
    printf("\n%d %d %d\n",tab[1][0],tab[1][1],tab[1][2]);
    for (i = 0; i < k; i++)
       free(tab[i]);
    free(tab);
    return 0;
}
```

Exemples de pointeurs

Exemples de pointeurs

Pointeurs et chaînes de caractères

```
#include <stdio.h>
#include <string.h>
int main()
{
    char *chaine;
    chaine="chaine de cacacters";
    printf("%s\n",chaine);
    printf("\nNombre de caracters = %d\n",strlen(chaine));
    return 0;
}
```

Exemples de pointeurs

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int *tab;
   int i;
   int *tab2;
   tab=(int*)malloc(3*sizeof(int));
   for(i=0;i<3;i++) {
      printf("\n Entrer tab[%d] = ",i);
      scanf("%d",&tab[i]);
   }
   for(i=0;i<3;i++) printf("tab[%d]=%d\n",i,tab[i]);
   tab2=tab;
   for(i=0;i<3;i++) printf("tab2[%d]=%d\n",i,tab2[i]);
   printf("ADRESSE de tab avant free=%d\n",tab);
   printf("ADRESSE de tab2 avant free=%d\n",tab2);
   free(tab);
   for(i=0;i<3;i++) printf("tab[%d]=%d\n",i,tab[i]);
   for(i=0;i<3;i++) printf("tab2[%d]=%d\n",i,tab2[i]);
   printf("ADRESSE de tab apres free=%d\n",tab);
   printf("ADRESSE de tab2 apres free=%d\n",tab2);
   return 0;
}
```

Exemples de pointeurs

Pointeurs et structures

```c
#include <stdio.h>
typedef struct
 { char *nom;
   float note_median;
   float note_final;
   float moyenne;
  } etudiant;

void modifier_notes(float a,float b, etudiant *etu)
{
   etu->note_median=a;
   etu->note_final=b;
   etu->moyenne=0.4*a+0.6*b;
}

int main()
{
   etudiant Dupont={"DUPONT", 17, 8.5, 0};
   etudiant Ami;

   Dupont.moyenne=0.4*Dupont.note_median+0.6*Dupont.note_final;
   Ami=Dupont;Ami.nom="RAYAN";modifier_notes(9,14,&Ami);
   printf("NOM : %s Moyenne : %1.2f\n",Dupont.nom, Dupont.moyenne);
   printf("NOM : %s Moyenne : %1.2f\n",Ami.nom, Ami.moyenne);
   return 0;
}
```

9. Fichiers

Généralités sur les fichiers

Généralités

Introduction

- Permettent de stocker les diverses données selon des extensions différentes
- Les fichiers peuvent être stockés sur plusieurs types de mémoire (disque dur, clé USB, CD, disquette...)
- Chaque fichier est identifié par son nom et son chemin d'accès
- Propriété des fichiers : Volume variable et Volume non défini à l'ouverture
- Les entrées/sorties d'un programme en C sont gérées par des fichiers

Généralités

Règle d'utilisation

- Le nom d'un fichier est composé de deux rubriques (une chaîne de caractères suivie d'une extension) :
  exemple d'un fichier texte : *titi.txt*
  exemple d'un fichier binaire : *toto.bin*
- Les fichiers sont ouverts avant toute manipulation
- Les éléments d'un fichier sont lus ou écrits séquentiellement
- Les fichiers sont fermés lorsque les opérations de lecture ou d'écriture sont terminées

- Le clavier de l'ordinateur est le fichier des entrées par défaut (`stdin`)
- L'écran de l'ordinateur est le fichier de sorties par défaut (`stdout`)
  L'écran est aussi par défaut l'unité d'affichage des messages d'erreur (`stderr`)

fopen

Création d'un fichier

- Déclaration d'un pointeur sur une structure FILE :

```
FILE *ptr;
```

- FILE est une structure définie dans stdio.h et elle regroupe les caractéristiques d'un fichier
- Saisie du nom du fichier : char name[20]= {'t', 'o', 't', 'o'} ;
- Ouverture du fichier :

```
ptr=fopen(name, "w");
```

- La fonction fopen permet de réserver une zone dans la mémoire centrale. Le résultat retourné est l'adresse de cette zone en cas de succès et NULL en cas d'échec.

fopen

Création d'un fichier

- Le résultat de la fonction sera donc affecté au pointeur `ptr` qui représentera l'adresse de la zone mémoire correspondante au fichier
- Le deuxième argument de la fonction `fopen` permet de spécifier le mode d'ouverture : le mode écriture, le mode lecture, le mode ajout
- A partir de ces trois catégories principales, d'autres modes dérivés peuvent être utilisés.
- Il existe aussi le mode *TEXTE* (traduction de certains caractères de contrôle : retour à la ligne,...) et le mode *BINAIRE* (aucune traduction)

fopen

Modes d'ouverture

Modes d'ouverture (en texte) d'un fichier :

- `r` : ouverture en mode lecture s'il existe. Le fichier doit exister.
- `w` : ouverture en mode écriture. Si ce fichier existe déjà, il sera supprimé et recréé.
- `a` : ouverture en mode ajout. Si le fichier n'existe pas, il sera créé.
- `r+` : ouverture en mode lecture et écriture s'il existe. Le fichier doit exister.
- `w+` : ouverture en mode lecture et écriture. S'il existe déjà, il sera supprimé et recréé.
- `a+` : ouverture en mode lecture et ajout. S'il n'existe pas, il sera créé.

En mode ajout, l'écriture se fait à la fin.

Modes d'ouverture (en binaire) d'un fichier :

- `rb`, `wb`, `ab`, `r+b`, `w+b`, `a+b` pour une lecture et/ou écriture.

fclose

Fermeture d'un fichier

- Après la réalisation de différentes actions sur un fichier, on doit le fermer avec la fonction `fclose`
- Elle permet de libérer la mémoire tampon réservée pour le fichier et sur laquelle le pointeur associé pointe.

```
#include <stdio.h>
void main()
{
    FILE *ptr;
    char name[20]={'t','o','t','o','.','t','x','t'};
    char nom[20];
    printf("Saisir un nom:");
    scanf("%s",nom);
    ptr=fopen(name,"w");
    ptr=fopen(nom,"w");
    fclose(ptr);
}
```

Quelques fonctions utiles

Quelques fonctions

Écrire dans un fichier - entrées formatées : fprintf

La fonction `fprintf` permet d'écrire des données, en précisant leur format dans un fichier qui est associé au pointeur.

```
#include <stdio.h>
int main()
{
   FILE *ptr;
   char nom[20];
   printf("Saisir un nom:");
   scanf("%s",nom);
   ptr=fopen(nom,"w");
   fprintf(ptr, "%s", "Je ecris dans ptr!!!\n");
   fclose(ptr);
   return 0;
}
```

Quelques fonctions

Lire dans un fichier - sorties formatées : fprintf

La fonction `fscanf` permet de lire des données, en précisant leur format dans un fichier qui est associé au pointeur.

```
char chaine; fscanf(ptr, "%c\n", &chaine);
```

```
#include <stdio.h>
int main()
{
   FILE *ptr;
   char name[20]={'t','o','t','o','.','t','x','t'};
   char chaine[20];

   ptr=fopen(name,"r");
   int i=0,z=0;
   do
   {
      fscanf(ptr, "%c", &chaine[i]);
      printf("%c",chaine[i]);
      if(chaine[i]=='\n') z=1;
      i++;
   } while (z==0);
   fclose(ptr);
   return 0;
}
```

Quelques fonctions

Détecter la fin d'un fichier : feof

La fonction feof permet de détecter la fin du fichier auquel le pointeur est associé : EOF

```
int test; test=feof(ptr);
```

```
#include <stdio.h>
int main()
{
   FILE *ptr;
   char name[20]={'t','o','t','o','.','t','x','t'};
   char car;

   ptr=fopen(name,"r");
   while (feof(ptr)==0)
   {
      fscanf(ptr, "%c", &car);
      printf("%c",car);
   };
   fclose(ptr);
   return 0;
}
```

Quelques fonctions

Lecture de caractères : fgetc

La fonction `fgetc` permet de lire le caractère dans le fichier associé au pointeur suivant la position du curseur qui sera incrémentée.

```
fgetc(ptr);
```

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int i, j;
    FILE *ptr;
    char car,name[20]={'t','o','t','o','.','t','x','t'};

    ptr=fopen(name,"r");
    while(feof(ptr)==0)
    {
        car=fgetc(ptr);
        printf("%c",car);
    };
    fclose(ptr);

}
```

Quelques fonctions

Ecriture de caractères : fputc

La fonction fputc permet d'écrire un caractère dans le fichier associé au pointeur selon la position du curseur qui sera incrémentée.

```
char ca='\n'; fputc(ca,ptr);
```

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int j=32;
    FILE *ptr;
    char car, name[20]={'t','o','t','o','.','t','x','t'};

    ptr=fopen(name,"w");
    while(j<128)
    {
        fputc((char)j,ptr);
        fputc('\n',ptr);
        printf("%c\n",(char)j);
        j++;
    }
    fclose(ptr);

}
```

Quelques fonctions

Exemple

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

#define ENTREE "entree.txt"
#define SORTIE "sortie.txt"

int main(void)
{
   FILE *f_in, *f_out;
   int c;

   if ((f_in = fopen(ENTREE,"r")) == NULL)
   {
      fprintf(stderr, "\n Erreur: Impossible de lire le fichier %s\n",ENTREE);
      return(EXIT_FAILURE);
   }
   if ((f_out = fopen(SORTIE,"w")) == NULL)
   {
      fprintf(stderr, "\n Erreur : Impossible d'ecrire dans le fichier %s\n", \
            SORTIE);
      return(EXIT_FAILURE);
   }
   while ((c = fgetc(f_in)) != EOF)
      fputc(c, f_out);
   fclose(f_in);
   fclose(f_out);
   return(EXIT_SUCCESS);
}
```

Quelques fonctions

fwrite

La fonction `fwrite` permet d'écrire des données dans le fichier à partir d'une zone mémoire précise

```
fwrite(tab,sizeof(int),n_enreg, ptr);
```

C'est une fonction du mode binaire, mais reste valable dans le cas du mode texte.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   FILE *ptr;
   char name[20]={'t','o','t','o','.','t','x','t'};
   char *tab;
   int i;
   tab=(char*)malloc(10*sizeof(char));
   for(i=0;i<10;i++)
      tab[i]=(char)48+i;
   ptr=fopen(name,"wb");
   fwrite(tab,sizeof(char),10,ptr);
   fclose(ptr);
   return 0;
}
```

Quelques fonctions

fread

La fonction `fread` permet de lire des données du fichier et de les sauvegarder dans une zone mémoire précise.

```
fread(tab,sizeof(int),n_enreg, ptr);
```

C'est une fonction du mode binaire, mais reste valable dans le cas du mode texte.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   FILE *ptr;
   char name[20]={'t','o','t','o','.','t','x','t'};
   char *tab;
   int i;
   tab=(char*)malloc(10*sizeof(char));
   ptr=fopen(name,"r");
   fread(tab,1,10,ptr);
   for(i=0;i<10;i++)
      printf("%c\n",tab[i]);
   fclose(ptr);
   return 0;
}
```

Quelques fonctions

Déplacement du curseur : fseek

La fonction fseek permet de déplacer le curseur du fichier à partir d'une position initiale d'un certain nombre d'octets.

```
fseek(ptr,n_octets, pos_init);
```

pos_init peut être SEEK_SET (début), SEEK_CUR (courant) ou SEEK_END (fin)

C'est une fonction du mode binaire, mais reste valable dans le cas du mode texte.

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
   int i,j;
   FILE *ptr;
   char name[20]={'t','o','t','o','.','t','x','t'};
   ptr=fopen(name,"r+");
   for(j=0;j<10;j++)
   {
      fseek(ptr,sizeof(int),SEEK_CUR);
      fscanf(ptr,"%d",&i);
      printf("\n%d",i);
   }
   fclose(ptr);
}
```

Quelques fonctions

Position courante : ftell

La fonction ftell permet de donner la position courante du curseur en octets.

```
int position; position=ftell(ptr);
```

C'est une fonction du mode binaire, mais reste valable dans le cas du mode texte.

```
#include <stdio.h>
unsigned int taille(FILE *pointeur)
{
   int size;
   fseek(pointeur,0,SEEK_END);
   size=ftell(pointeur);
   fseek(pointeur,0,SEEK_SET);
   size-=ftell(pointeur);
   return size;
}
void main()
{
   int i;
   FILE *ptr;
   char name[20]={'t','o','t','o','.','t','x','t'};
   ptr=fopen(name,"r");
   i=taille(ptr);
   printf("la taille du fichier est %d octets\n",i);
   fclose(ptr);
}
```

10. Listes chaînées

Listes chaînées

Listes chaînées

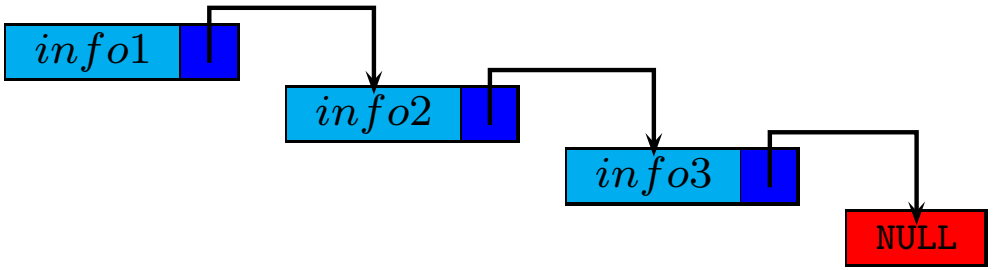
- Une liste chaînée est une structure récursive. C'est-à-dire, dans la définition de la structure, on fait appel au même type de structure
- D'un point de vue algorithmique, une telle structure peut être créée comme suit :

```
struct cellule
{
   int infos;
   ...
   struct cellule *pointeur;
};

typedef cellule brick;
typedef cellule *liste;
```

```
typedef struct cellule
{
   int infos;
   ...
   cellule *pointeur;
} cellule;

cellule brick;
cellule *liste;
```



Listes chaînées

```c
#include <stdio.h>

typedef struct cellule
{
   int info;
   struct cellule *suivant;
} cellule;

int main()
{
   cellule exemple, part, last;
   int a, b, c;

   last.info=4;
   last.suivant=NULL;

   part.info=5;
   part.suivant=&last;

   exemple.info=7;
   exemple.suivant=&part;

   a=exemple.info;
   b=(exemple.suivant)->info;
   c=((exemple.suivant)->suivant)->info;
   printf("%d %d %d\n",a,b,c);

   b=(*exemple.suivant).info;
   c=(*(*exemple.suivant).suivant).info;
   printf("%d %d\n",b,c);
   return 0;
}
```

Listes chaînées : Quelques fonctions associées

Affichage d'une liste chaînée

```c
#include <stdio.h>

typedef struct cellule
{
    int info;
    struct cellule *suivant;
} cellule;

void afficher(cellule l)
{
    cellule *aux;
    aux=&l;
    while (aux!=NULL)
    {
        printf("%d ",(*aux).info);
        aux=aux->suivant;
    };
}

int main()
{
    cellule exemple, part;
    exemple.info=4;
    part.info=6;
    part.suivant=NULL;
    exemple.suivant=&part;
    printf("Les elements:");
    afficher(exemple);
    return 0;
}
```

Listes chaînées : Quelques fonctions associées

Fonction `first_element`

La fonction `first_element` crée une liste qui contient le premier élément de la chaîne traitée et retourne un pointeur sur la liste créée

```
#include <stdio.h>

typedef struct cellule
{
   int info;
   struct cellule *suivant;
} cellule;

// Copier la fonction afficher ici
cellule first_element(cellule l)
{
   cellule *aux;  aux=&l;
   if(aux==NULL) return l;
   else { aux->suivant=NULL; return (*aux); };
}
int main()
{
   cellule exemple, part, last, premier;
   last.info=4;  last.suivant=NULL;
   part.info=5;  part.suivant=&last;
   exemple.info=7;  exemple.suivant=&part;
   printf("Les elements:");
   afficher(exemple);
   premier=first_element(exemple);
   printf("\n Premier element:");
   afficher(premier);
   return 0;
}
```

Listes chaînées : Quelques fonctions associées

Fonction `taille`

La fonction `taille` retourne un entier qui représente la longueur de la chaîne traitée (le nombre d'éléments qu'elle contient)

```
#include <stdio.h>

typedef struct cellule
{
   int info;
   struct cellule *suivant;
} cellule;

int taille(cellule *l)
{
   cellule *aux;  aux=l;
   if(aux==NULL) return 0;
   else
   {
      int z=0;
      while (aux!=NULL) { aux=aux->suivant; z++; };
      return z;
   };
}
int main()
{
   cellule exemple, part, last;
   last.info=4;  last.suivant=NULL;
   part.info=5;  part.suivant=&last;
   exemple.info=7;  exemple.suivant=&part;
   printf("taille d'exemple= %d\n", taille(&exemple));
   return 0;
}
```

Listes chaînées : Quelques fonctions associées

Fonction remove_first

La fonction remove_first supprime le premier élément de la liste si elle n'est pas vide et retourne un pointeur sur la nouvelle liste

```
#include <stdio.h>

typedef struct cellule
{
   int info;
   struct cellule *suivant;
} cellule;

// Copier la fonction afficher ici
cellule *remove_first(cellule *l)
{
    if(l!=NULL) return (l->suivant); else return l;
}
int main()
{
   cellule *chaine,exemple, part, last;
   last.info=4;  last.suivant=NULL;
   part.info=5;  part.suivant=&last;
   exemple.info=7;  exemple.suivant=&part;
   chaine=&exemple;
   printf("Liste:");
   afficher(*chaine);
   printf("\nListe apres la suppression:");

   chaine=remove_first(chaine);
   afficher(*chaine);
   return 0;
}
```

Listes chaînées : Quelques fonctions associées

Fonction add_first

La fonction add_first ajoute un élément *à la tête* de la chaîne traitée et retourne un pointeur sur la nouvelle liste

```
#include <stdio.h>
typedef struct cellule
{
   int info;
   struct cellule *suivant;
} cellule;

//  Copier la fonction afficher ici
cellule *add_first(int a,cellule *l)
{
   cellule *aux;
   aux=(cellule *)malloc(sizeof(cellule));
   aux->info=a;
   aux->suivant=l;
   return aux;
}
int main()
{
   cellule *chaine, *ch;
   int i;
   chaine=(cellule *)malloc(sizeof(cellule));
   ch=(cellule *)malloc(sizeof(cellule));
   chaine->info=0;
   chaine->suivant=NULL;
   for(i=0;i<5;i++) chaine=add_first(i+1,chaine);
   afficher(*chaine);
   return 0;
}
```

Listes chaînées : Quelques fonctions associées

Fonction liberer

La fonction liberer permet de libérer la zone mémoire occupée par les éléments de la chaîne.

```
#include <stdio.h>

typedef struct cellule
{
   int info;
   struct cellule *suivant;
} cellule;

//  Copier la fonction add_first ici

void liberer(cellule *l)
{
   cellule *aux;
   aux=(cellule *)malloc(sizeof(cellule));
   while (l!=NULL) {aux=l->suivant;free(l);l=aux;};
}
int main()
{
   cellule *chaine;
   int i;
   chaine=(cellule *)malloc(sizeof(cellule));
   chaine->info=0;
   chaine->suivant=NULL;
   for(i=0;i<5;i++) chaine=add_first(i+1,chaine);
   liberer(chaine);
   return 0;
}
```

Listes chaînées : Quelques fonctions associées

Fonction acceder

La fonction acceder permet d'accéder au pointeur sur un élément précis dans la chaîne traitée et retourne ce pointeur.

```
#include <stdio.h>
typedef struct cellule
{
   int info;  struct cellule *suivant;
} cellule;
//Copier les fonctions afficher et add_first ici
cellule *acceder(int a,cellule *l)
{
  if(l==NULL) return l;
  else
  {
     cellule *aux;
     aux=(cellule *)malloc(sizeof(cellule));
     aux=l;
     while (aux!=NULL) { if(aux->info==a) return aux;  else aux=aux->suivant; };
     return aux;
  };
}
int main()
{  cellule *chaine;  int i;
   chaine=(cellule *)malloc(sizeof(cellule));  chaine->info=0;  chaine->suivant=
        NULL;
   for(i=0;i<10;i++) chaine=add_first(i+1,chaine);
   printf("cellulee:");  afficher(*chaine);
   chaine=acceder(5,chaine);
   printf("\nPossition 6:"); afficher(*chaine);
   return 0;
}
```

Liste chaînées : Quelques fonctions associées

Fonction supprimer

La fonction supprimer permet de supprimer un élément précis dans la chaîne traitée et retourne un pointeur sur la nouvelle chaîne.

```
#include <stdio.h>
typedef struct cellule
{
   int info;
   struct cellule *suivant;
} cellule;

// Copier les fonctions afficher et *add_first ici
cellule *supprimer(int a,cellule *l)
{
   if(l==NULL) return l;
   else { if(l->info==a) return (l->suivant);
          else return (add_first(l->info,supprimer(a,l->suivant)));};
}
int main()
{
   cellule *chaine;  int i;
   chaine=(cellule *)malloc(sizeof(cellule));
   chaine->info=0;  chaine->suivant=NULL;
   for(i=0;i<9;i++) chaine=add_first(i+1,chaine);
   printf("Liste:");  afficher(*chaine);
   chaine=supprimer(8,chaine);
   printf("\nApres la suppression element 8:");  afficher(*chaine);
   chaine=supprimer(7,chaine);
   printf("\nApres la suppression element 7:");  afficher(*chaine);
   return 0;
}
```

Listes chaînées : Quelques fonctions associées

Fonction `remove_last`

La fonction `remove_last` permet de supprimer le dernier élément de la chaîne traitée qui sera modifiée en conséquence.

```
#include <stdio.h>
typedef struct cellule
{ int info;  struct cellule *suivant; } cellule;

void remove_last(cellule **l)
{
   if(((*l)==NULL)||((*l)->suivant)==NULL) (*l)=NULL;
   else { cellule *aux1;  aux1=(cellule *)malloc(sizeof(cellule));
          cellule *aux;  aux=(cellule *)malloc(sizeof(cellule));
          aux=*l;
          while ((aux->suivant)!=NULL) {aux=aux->suivant;};
          aux1=aux;
          aux=NULL;
          free(aux1);
        };
}
void main()
{
   cellule *chaine,*exemple, *part, *last;
   exemple=(cellule *)malloc(sizeof(cellule));
   part=(cellule *)malloc(sizeof(cellule));
   last=(cellule *)malloc(sizeof(cellule));
   (*last).info=4;  last->suivant=NULL;
   (*part).info=5;  part->suivant=last;
   exemple->info=7;  exemple->suivant=part;
   chaine=exemple;
   remove_last(&chaine);
}
```

Listes chaînées : Quelques fonctions associées

Quelques applications

- [Exo1] Créer une liste de contact avec les éléments : nom, prénom, tel, ...
  Modifier les fonctions précédentes pour ajouter, modifier, supprimer un contact
- [Exo2] Déterminer l'ensemble de nombres premiers entre 2 et N (N est donné par l'utilisateur) en utilisant une liste chaînée
- [Exo3] Représentation d'un polynôme, exemple $3X^3$-$2X$+$5X^2$+1 :
  - Ecrire un programme effectuant la lecture d'un polynôme et sa représentation en machine sous la forme d'une liste chaînée dont chaque maillon représente un monôme.
  - Modifiez le programme précédent afin qu'un polynôme soit représenté par une liste de monômes rangée par ordre (strictement) décroissant des degrés.
  - Ecrire une fonction qui renvoie la valeur d'un polynôme donné P pour une valeur donnée de X.

11. Notions sur la complexité & la qualité

Notions sur la complexité et la qualité

Notions sur la complexité et la qualité

La qualité d'un logiciel ou d'un programme

La qualité d'un logiciel ou d'un programme est une question multicritère car elle dépend de plusieurs paramètres :

- Est-ce que le programme répond au besoin ?
- À quel prix ?
- Est-ce que le programme est compréhensible ?
- Est-ce que le programme est facilement modifiable ?
- Est-ce que le programme peut être facilement étendu à d'autres besoins ?
- Est-ce que le programme peut tomber en panne ?
- En cas de panne, offre-t-il une aide à l'utilisateur ?

Notions sur la complexité et la qualité

Définition de la qualité

Définition de la qualité :

La qualité d'un programme obéit à des normes, selon l'afnor, sa définition est la suivante :

- "La qualité est l'ensemble des caractéristiques d'une entité qui lui confère l'aptitude à satisfaire des besoins exprimés ou implicites" [NORME ISO 8402]
- "Le terme qualité n'est pas utilisé pour exprimer un degré d'excellence dans un sens comparatif ; il n'est pas non plus utilisé dans un sens quantitatif pour des évaluations techniques" [NORME NF X50.120]

Notions sur la complexité et la qualité

Quelques conseils pratiques

Quelques conseils pratiques :

- Analysez bien votre problème
- Posez-vous la question : que doit faire le programme ?
- Identifiez les entrées, les sorties et les variables de décision (cahier de charges !)
- Décomposez les actions nécessaires pour atteindre l'objectif
- Cherchez des solutions pour chaque action
- Critiquez ces solutions et sélectionnez la plus satisfaisante pour chaque action
- Évaluez globalement vos choix et rectifiez en cas de nécessité

Notions sur la complexité et la qualité

Validation des algorithmes

Validation des algorithmes :

- En cas d'un projet important, la notion de la planification, la conduite du projet et le travail en équipe sont nécessaires pour bien organiser les choses
- Dialoguez souvent avec le client pour valider votre démarche
- Écrivez toujours vos procédures et vos algorithmes et évaluer leurs complexités et leurs capacités à atteindre l'objectif avant de passer au codage (l'écriture des programmes)
- Évaluez l'efficacité des algorithmes et les validez en cas de satisfaction

Notions sur la complexité et la qualité

Gestion et conflits

Gestion et conflits :

- Pensez aussi aux ressources nécessaires à l'exécution de votre programme (mémoire nécessaire, processeur, système d'exploitation, ...)
- Optimisez l'utilisation de la mémoire et du processeur
- Évitez les conflits avec les autres programmes
- Fournissez un mode d'emploi de votre programme
- Expliquez et commentez vos programmes
- Codez d'une manière structurée et claire ; vos programmes doivent être modulaires
- Pensez toujours à l'étape de la maintenance et de l'extension

Notions sur la complexité et la qualité

Complexité et rapidité

Complexité et rapidité :

- En dehors de l'objectif visé, la performance d'un programme peut être évaluée par sa rapidité à atteindre cet objectif
- La complexité d'un algorithme est, en réalité, l'indicateur qui peut nous renseigner sur la rapidité d'un programme
- La complexité d'un algorithme peut être définie simplement comme le nombre des actions nécessaires à son exécution en fonction du nombre de données
- Exemple : trier un tableau de n entiers selon la méthode classique est d'une complexité de $n \times n/2$
- Le temps nécessaire pour exécuter un programme augmente avec une complexité croissante
- En conséquence, la complexité est un coût important à prendre en compte dans la recherche d'algorithmes performants
- Un bon programmeur cherche toujours des algorithmes à faible complexité
- L'efficacité calculatoire est fortement corrélée avec le mode de gestion de la mémoire du programme
- Exploitez les possibilités offertes par le langage C en terme de gestion de mémoire en mode dynamique

Notions sur la complexité et la qualité

Récursivité

Récursivité :

- Évitez également les appels récursifs inutiles et préférez l'itération
- Notez bien que la récursivité est très gourmande et coûteuse en terme du temps de calcul
- La récursivité oblige généralement le compilateur à refaire des instructions déjà effectuées
- Pour vous convaincre, faites l'expérience : testez la version récursive de la fonction "premier" et la comparez avec la version itérative. Déterminez la taille limite atteinte par chaque version. Conclure.

Notions sur la complexité et la qualité

Notion de la complexité

Notion de la complexité :

Et si vous n'êtes pas toujours convaincu, observez le temps de calcul nécessaire pour deux complexités différentes (10^{-3} s/action) :

n	temps (complexité $= n^2$)	temps (complexité $= n\log(n)$)
10	0.1 s	0.01 s
100	10 s	0.2 s
1000	17 min	3 s
10000	27.8 h	40 s
100000	116 j	8.3 min