ALGORITHMIQUE ET PROGRAMMATION

On se trouve face à un problème :

- dont on connait un certain nombre de **données**.
- dont la résolution procurera un certain nombre de **résultats**.

Exemple :

Trouver la position de la chaîne de caractère ←*problème*
« **stéphane** » ←*donnée*
dans
« stéenastéphaneeanstéphane…….eeestéphane » ←*donnée*
réponse : positions 7, 18,… 6742 ←*résultats*

La résolution d'un problème passe par deux étapes :

1- l'algorithme : la description précise des opérations à faire
2- le programme : l'expression d'un algorithme dans un langage qu'un ordinateur donné comprend.

L'obtention du programme :

Dans la technique de **l'interprétation** (exemple langage python) **:**

Code source ► Interpréteur ► Résultat

L'interpréteur lit le code source …

…et le résultat apparaît sur l'écran

Dans la technique de **la compilation** (exemple langage C ou C++) **:**

Code source►Compilation►code objet►Edition de liens►

Le compilateur lit le code source …

…et produit un code objet….

…L'éditeur de lien crée le programme …

►Programme machine ►Exécution ► Résultat

…on exécute le programme…

…et le résultat apparaît sur l'écran

STRUCTURE D'UN PROGRAMME C

Un programme C comporte une série d'instructions structurées en blocs. Il peut y avoir une ou plusieurs fonctions dans un fichier. Cependant un fichier C contiendra toujours une fonction appelée **fonction principale**, elle correspond à l'algorithme principal et peut utiliser les autres fonctions définies dans ce fichier ou les fonctions prédéfinies du langage.

La fonction principale

```
int main(void)
```

La fonction principale s'appelle le `main`. Cette fonction retournera la valeur 0 pour indiquer que le programme s'est correctement déroulé elle peut avoir des arguments, mais nous n'en utiliserons pas et pour celà nous mettrons le mot `void` entre parenthèses. Le mot `int` placé devant la fonction indique le ***type*** de la valeur qui est retournée par la fonction `main`. Ce sera toujours un entier.

La structure du programme

La structure d'un programme est liée à plusieurs contraintes : les contraintes d'ordre syntaxique et les contraintes que l'on s'impose pour avoir une bonne lisibilité.

On peut structurer un programme de la façon suivante :

- Partie identification : en commentaire, nom du programme, auteur, date …..
- Partie directives : nouveaux types, déclaration des variables globales, prototypes globaux.
- Fonction principale : déclaration de variables, prototype de fonctions , corps de la fonction.
- Partie définition des fonctions.

↔

Exemple de programme:

```
// Partie identification
// nom du programme
// auteur, date


// Partie directives
// des inclusions de fichiers en-tete ,constantes…


#include <string.h>
#include <math.h>

using namespace std;
const int PI=3.14;
typedef float F[12];


// fonction principale


int main(void)
{
  // declaration de variables
  int i,j, m;
  float k;
...
  // prototype des fonctions
  int Mult(int , int );
...
  // corps de la fonction principale
  m=Mult(i,j);
...
  system("PAUSE");
  return 0
}


// definitions des fonctions


int Mult(int i, int j)
{
   return(i*j);
}
...
```

LES VARIABLES

L'essentiel du travail effectué par un programme d'ordinateur consiste à manipuler des données. Ces données peuvent être très diverses, mais dans la mémoire de l'ordinateur elles se ramènent toujours en définitive à **une suite finie de nombres binaires**.

Pour pouvoir accéder aux données, le programme (quel que soit le langage dans lequel il est écrit) fait usage d'un grand nombre de **variables** de différents types.

Une VARIABLE possède :

Un nom : Un variable apparaît en programmation sous un **nom de variable**.

Un type : Pour distinguer les uns des autres les divers contenus possibles, on utilise différents types de variables (entiers, réels, chaîne de caractères …).

Une valeur : Une fois la variable définie, une **valeur** lui est **assignée**, ou **affectée.**

Une adresse : C'est l'emplacement dans la mémoire de l'ordinateur, où est stockée la valeur de la variable.

LES TYPES DE VARIABLES EN C

Pour pouvoir accéder aux données, le programme (quel que soit le langage dans lequel il est écrit) fait usage d'un grand nombre de **variables** de différents types.

Pour distinguer les uns des autres les divers contenus possibles, on utilise différents types de variables (entiers, réels, chaîne de caractères …).

En langage C, comme dans d'autres langages, il faut toujours, par des instructions précises, déclarer (définir) le nom et le type d'une variable, avant de pouvoir lui affecter (assigner) une valeur (un contenu) qui doit être compatible avec le type déclaré. Si par la suite une autre valeur est affectée à une variable, la précédente est perdue.

En langage C les principaux types de variables sont les suivants :

Booléen :

Ensemble de définition :	{FAUX , VRAI}
Déclaration algorithmique :	a , b : booléen
Déclaration C :	bool a ,b ;

Entier :

Ensemble de définition :	9
Déclaration algorithmique :	i , j : entier
Déclaration C :	int i , j ;

Flottant :

Ensemble de définition :	IF
Déclaration algorithmique :	x,y : flottant
Déclaration C :	float x , y ; ou double x , y ;

Caractère :

Ensemble de définition :	La table ASCII
Déclaration algorithmique :	c, g : caractère
Déclaration C :	char c , g ;

LES OPERATEURS EN C

Opérateurs arithmétiques en langage C :

Dans tout programme on manipule les **valeurs** et les **variables** qui les référencent, en les combinant avec des **opérateurs** pour former des **expressions**. Exemple :
y= 3*a + b/5
Ici, l'exemple consiste à affecter à la variable **y**, le résultat d'une expression qui combine les **opérateurs =,** *, + et / avec les **opérandes** a, b, 3 et 5.

Lorsqu'il y a plus d'un opérateur dans une expression, l'ordre dans lequel les opérations doivent être effectuées dépend des **règles de priorité**. En C, les règles de priorité sont les mêmes que celles qui vous ont été enseignées au cours de mathématique.

Opérateur		Notation algorithmique	Notation C
addition	+	**a+b**	**a+b**
soustraction	-	**a-b**	**a-b**
opposé	-	**-a**	**-a**
produit	*	**a * b**	**a * b**
division	/	**a / b**	**a / b** (division entière ou décimale selon les types de variables)
modulo	%	**a%b**	**a%b** (variables de type entier)

Opérateurs relationnels (de comparaison) en langage C:

La condition évaluée après l'instruction if (entre autres !) peut contenir les opérateurs de comparaison suivants :

Notez bien que l'opérateur de comparaison pour l'égalité de deux valeurs est constitué de deux signes " = " et non d'un seul. Le signe " = " utilisé seul est un opérateur d'affectation et non un opérateur de comparaison.

↔

Opérateurs booléens en langage C:

Opérateur	Notation algorithmique	Notation C

négation	**NON(a)**	**!a**
ou	**a OU b**	**a || b**
et	**a ET b**	**a && b**

Opérateurs binaires en langage C :

Ils servent à manipuler les mots bit à bit, par exemple pour faire des masques :

Tables de vérité des opérateurs logiques :

a	b	a&b	a|b	a^b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

LES DIFFERENTS TYPES DE BOUCLES

Le langage C fournit trois instructions pour réaliser des boucles :

- **while**
- **do …while**
- **for**

1-Tant que (condition) faire...

En langage C :

```
while (condition) {
   instructions;
}
```

Algorithme :

```
TantQue(condition)
   bloc_instructions
FinTantQue
```

Tant que la valeur de la condition est différente de zéro (faux), le contenu de la boucle est exécuté. Attention ! Etant donné que la condition est testée en début de boucle, il est donc possible de ne jamais entrer dans le corps de la boucle.

2-Faire ... tant que (condition)

En langage C :

```
do {
   instructions;
} while (condition);
```

Algorithme :

```
Faire
   bloc_instructions
TantQue(condition)
```

La condition est testée à la fin de la boucle, donc avec une boucle do...while, on passe toujours au moins une fois dans le corps de la boucle.

3-Boucle for

En langage C :

```
for(expression1;expression2;expression3){
   instructions;
}
```

Algorithme :

```
Pour cpt Dans[a…b] ParPasDe n
  bloc_instructions
FinPOur
```

expression1 réalise des initialisations avant l'entrée dans la boucle, expression2 est le test de continuation de boucle (test réalisé avant chaque itération), expression3 est évaluée à la fin de chaque itération.

C'est l'équivalent de :

```
expression1;
while(expression2)
{
  instructions;
  expression3;
}
```

LES FONCTIONS ET PROCEDURES (1)

La notion de **fonction** est très importante en programmation. Les fonctions et les **procédures** permettent de décomposer un programme complexe en une série de sous-programmes plus simples qui peuvent eux-mêmes être décomposés en éléments plus petits, et ainsi de suite…
Une **fonction** (ou une **procédure**) est réutilisable. Par exemple, si nous disposons d'une **fonction** capable de calculer la moyenne d'un tableau, nous pouvons l'utiliser partout dans notre programme. Utiliser des **fonctions** permet donc d'obtenir un code plus clair, plus lisible, et moins redondant.
Une **fonction** reçoit des données ou arguments et elle retourne un résultat et un seul. Lorsqu' aucun résultat n'est retourné on parle de **procédure**.

Une fonction possède trois aspects :

1- Le **prototype :** C'est la déclaration indispensable pour utiliser la fonction. Un prototype donne des informations indispensables pour le programmeur et pour le compilateur. Il précise les arguments attendus (nombre et type), et ce que la fonction retourne.

La notation est la suivante :

```
void
int       Demande( int, float, int[],….,char) ;
float…
```

↑	↑	↑	↑
type de retour	nom de la fonction	type des arguments, séparés par des virgules et entre parenthèses	point virgule

2- L'**appel :** C'est l'utilisation de la fonction dans le programme principal, ou dans une autre fonction. Les variables en argument doivent obligatoirement être du même type et en même nombre que dans le **prototype**. Exemple :

```
Demande( i, x, tab1,….,c)    ;
```

↑	↑	↑
nom de la fonction	arguments séparés par des virgules entre parenthèses	point virgule

↔

3- La **définition :** C'est l'écriture de la fonction. Elle comporte un **en-tête** et un **corps :**

- L'**en-tête** comprend le type de retour, le nom de la fonction suivit d'une liste d'arguments (type + nom de la variable) entre parenthèses :

```
void
int      Demande( int i, float x, int tab1[],….,char c)
float…
```

↑	↑	↑
type de retour	nom de la fonction	type et nom des arguments, séparés par des virgules et entre parenthèses

- Le **corps** de la fonction, c'est un bloc d'instructions, suivi ou non d'une instruction de retour (selon qu'il s'agit d'une fonction ou d'une procédure).

Exemple de **définition** d'une fonction retournant la somme de deux entiers :

```
int  Somme( int i, int j)                    ← en-tête
{
  int k;                                     }
  k=i+j;                                     } ← corps

  return k;
}
```

Remarque importante :

Le **nom** des variables n'est pas obligatoirement le même dans l'appel et dans l'en-tête de la fonction.

FONCTIONS ET PROCEDURES (2)

- Le passage des arguments :

En C il y a trois modes possibles pour le passage des arguments :

1- **Le passage par valeur** : Dans ce cas, la fonction travaille sur une copie des variables passées en arguments, copie affectée de la même valeur. On dit que les arguments sont **en entrée** et la modification d'une copie ne modifie pas l'original. La fonction ne modifiera donc jamais le contenu de la variable qu'on lui passe.

2- Le passage par référence : En plus du passage par valeur, le C définit le passage par référence. Lorsque l'on passe à une fonction un paramètre par référence, cette fonction reçoit un **"synonyme"**, un alias du paramètre. On dit que les arguments sont **en entrée/sortie** et dans ce cas toute modification du **"**synonyme**"** est répercutée sur le paramètre réel.

Dans le prototype, le fait que le passage s'effectue par référence est indiqué par « & » après le type de variable. Dans la fonction, pour accéder au contenu de la variable on utilise directement le nom de la variable.

3- Le passage par adresse : Il sera revu dans la fiche sur les pointeurs. Retenons seulement que c'est le mode implicite de passage des arguments de type tableau. Dans ce cas on passe l'adresse de la variable, ce qui permet à la fonction, connaissant l'adresse de la variable en mémoire, de modifier son contenu.

L'appel s'effectue en plaçant un & devant les noms des variables, sauf si c'est un tableau. Dans le prototype, le fait que le passage s'effectue par adresse est indiqué par * devant le nom de la variable. Dans la fonction, pour accéder au contenu de la variable on utilise * devant le nom de la variable, sauf si c'est un tableau.

↔

- L'utilisation des fonctions prédéfinies :

Dans le langage C, on utilise régulièrement un certain nombre de fonctions prédéfinies appartenant à la bibliothèque standard. Ces fonctions peuvent être de différents types, fonctions mathématiques (racine carrée *sqrt*, cosinus *cos* ...), fonctions d'affichage ou de saisie (saisie *cin*, affichage *cout* …) etc.

Pour utiliser une fonction prédéfinie, comme pour toute sorte de fonction, il est nécessaire de connaître le type de la fonction, le type et le nombre d'arguments, en bref le **prototype** de la fonction.

Ces prototypes sont regroupés dans les **fichiers .h**. Par exemple, le fichier `stdlib.h` contient le prototype de la fonction *atoi* qui permet de transformer une chaîne de caractères en un entier :

```
int atoi (char * ) ;
```

C'est pourquoi il faut donc inclure les **fichiers .h** nécessaires (c'est-à-dire ceux dont on utilise les fonctions) en tête de programme, par une instruction de ce type :

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

LES STRUCTURES EN C

Les **structures** permettent de regrouper sous un même nom, des objets de **types hétérogènes**, contrairement aux tableaux qui regroupent sous un même nom un ensemble de valeurs de même type.

Une variable de type structure est une variable composée de champs, eux-mêmes déclarés d'un certain type.

Déclaration de type structure :

Une déclaration de type structure ne définit aucune variable, elle permet de définir un modèle de structure. Déclarer une structure c'est définir un nouveau type de variable.

Syntaxe :

```
struct identificateur_de_type
        { type1 var1 ;        // premier champ
          type2 var2 ;        // deuxième champ
          ...
        };
```

Exemple :

```
struct eleve {
        char nom[30];
        char prenom[30];
        char sexe;
        int  age;
        float moyenne ;
 };
```

Ici on définit une nouvelle structure *eleve* composée de 5 champs :
Les deux premiers champs sont des chaînes de caractères (tableaux).
Le troisième est de type caractère.
Le quatrième est de type entier.
Le cinquième est de type flottant.

↔

Déclaration et utilisation d'une variable structure.

Si *eleve1* est une variable de type *eleve* (définie ci-dessus) :

- Déclaration d'une variable de type élève :

```
struct eleve eleve1 ;
```

- Affectation d'une valeur à chacun des champs :

Pour faire référence à un champ particulier de la structure, on accole le nom de la structure concernée avec le nom du champ visé, et on insère le caractère **"."** (point) entre les 2 termes.

```
strcpy(eleve1.nom, "De Musset");
strcpy(eleve1.prenom, "Alfred");
eleve1.sexe='m';
eleve1.age=19;
eleve1.moyenne=20;
```

- Remarque :

Si on utilise un pointeur sur une structure, l'accès aux champs s'écrit :
`*(eleve1.age);`
ou plus simplement on utilise la notation fléchée :
`eleve1→age;`

ENTREE/SORTIE SUR FICHIERS ASCII

Rappelons qu'un fichier texte (ou ASCII) est un fichier qui est lisible, il peut être édité dans votre éditeur de texte.

Le flux :
En langage C, lorsque l'on utilise un fichier on utilise une variable appelée flux. C'est une variable de type `FILE*` déclarée dans l'en-tête `<stdio.h>`.

Ouverture d'un fichier

Pour ouvrir un fichier on utilise la fonction fopen, dont le prototype est le suivant :

```
FILE *fopen(char *, char *);
```

Cette fonction ouvre le fichier dont le nom est donné comme premier argument, selon le mode d'ouverture précisé en second argument (w = écriture, r = lecture, a = ajout en fin de fichier) et l'assigne à un flux, c'est-à-dire à une variable de type `FILE *`.

Exemple : Ouverture du fichier de nom "monfichier" en mode écriture

```
#include<stdio.h>
using namespace std ;
int main(void)
{
    FILE* fic ;

    fic=fopen( "monfichier", "w") ;
    return 0
}
```

↔

Fermeture d'un fichier

Pour fermer un fichier on utilise la fonction fclose , dont le prototype est le suivant :

```
void fclose(FILE *);
```

Exemple : Ouverture en mode écriture puis fermeture du fichier de nom "monfichier" en mode écriture

```
#include<stdio.h>
using namespace std ;
int main(void)
{
    FILE* fic ;
    fic=fopen("monfichier", "w") ;
    fclose(fic);          // fermeture du flux
    return 0
}
```

Lecture des données d'un fichier

Pour saisir des données dans un fichier on peut par exemple utiliser la fonction **fscanf** :
La syntaxe est la suivante :

```
fscanf(fic, format, adr_var_1, adr_var_2, ...);
```

Elle prend comme premier argument le flux (de type FILE*) dans lequel elle doit lire et stocker les données lues selon le format défini par la chaîne `format` dans les variables `adr_var_xx` passées par adresse.

Écriture de données dans un fichier

Pour écrire des données dans un fichier on peut par exemple utiliser la fonction **fprintf** :
La syntaxe est la suivante :

```
fprintf(fic, format, var_1, var_2, ...) ;
```

Cette fonction écrit les données `var_x` (passées par valeur) dans le flux `fic` en respectant le format spécifié par la chaîne `format`.

LES POINTEURS

- Qu'est-ce qu'un pointeur : C'est une variable dont la valeur est l'adresse d'une autre variable de n'importe quel type. Plus simplement, c'est une variable qui contient l'adresse d'une autre variable.

 La syntaxe générale d'une définition ou déclaration de variable pointeur est de la forme :

```
type *identificateur ;
```

Le type est celui de la variable pointée, identificateur est le nom de la variable pointeur .

 Exemple :

```
char *pt1;
```

pt1 est une variable pointeur de char. Elle contiendra donc l'adresse d'une zone mémoire capable de contenir elle même un objet de type char.

-Les opérateurs associés :

 L'opérateur unaire & fournit l'adresse en mémoire d'une variable.

Exemple :

 Programme C :

```
int main(void)
{
   int i=32; // i: entier
   int *ptr ; //ptr: pointeur d'entier

   ptr=&i ; //ptr contient l'adresse de i
   return 0 ;
}
```

↔

- En mémoire, on obtient :

adresse de la variable ↓	valeur ↓	nom de la variable ↓
…..	…..	…..
…..	…..	…..
12569	42153	ptr
…..	…..	…..
…..	…..	…..
42153	65	i
…..	…..	…..

- L'opérateur unaire d'indirection ***** (étoile) permet d'obtenir la valeur d'une variable dont l'adresse est contenue dans le pointeur.

- Exemple :

```
int main(void)
{
int   i=12,*ptr;      // i: entier , ptr: pointeur d'entier

    ptr = &i;       // ptr contient maintenant l'adresse de i
   *ptr = (*ptr)/4; // equivalent a :     i = i / 4;
   cout i;          // affiche 3 car i vaut maintenant 12/4
    return 0;
}
```