TOGO CODE RUN
TCR

Cours d'initiation

LANGAGE C ET ALGO

Stimuler l'innovation en permettant aux jeunes d'apprendre à développer des idées et solutions créatives à travers la programmation et les technologies numériques.

Rédiger par
DJEGNON Achille

SEQUENCE I : GENERALITES

Dans ce chapitre, nous vous proposons de voir les termes principaux lier à l'informatique basé sur 1 exemple commentés.

Q : Qu'es ce que la programmation ?
R : La programmation est l'ensemble des activités qui permettent de réaliser des programmes informatiques.
Vous en avez déjà rencontré par exemple dans les jeux vidéo, les sites web et même les applications mobiles.

Plus simplement un programme informatique c'est comme une recette de cuisine.

1. Présentation par un exemple, de quelque instruction

1.1 Exemple de programme

Programme informatique

```
Début program

    nombre nbOeufs = 4
    nombre quantiteFarine = 80

    nombre resultat = nbOeufs * quantiteFarine

    afficher resultat

Fin program
```

Recette de cuisine

Gâteau au chocolat

Ingrédients :

4 œufs
150g de sucre
80g de farine
100g de levure
100g de beurre
200g de chocolat pâtissier

Recette :

Réchauffez le four à 200°
Dans un saladier, mélangez les œufs et le sucre à l'aide d'un fouet
Ajoutez petit à petit la farine
Dans une casserole faites fondre le beurre avec le chocolat
Versez le chocolat dans le saladier en agitant le fouet
Beurrez puis farinez un moule, versez la préparation dans le moule
Faites cuire votre gâteau pendant 20 à 25 minutes.

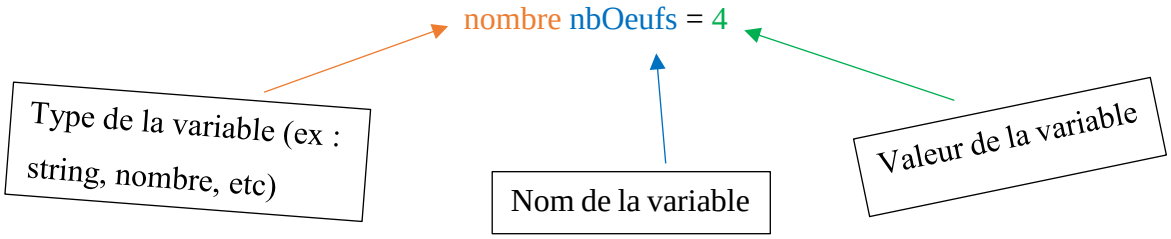
La suite d'étape, d'instruction ou chose à faire comme dans une recette, c'est la même chose en programmation.

Pour réaliser notre recette on aura besoin de certain élément dit ingrédient ; en programmation on les appellera des ***variables***.

1.2 Les variables

Une variable permet de stocker temporairement une valeur à l'intérieur sachant que cette valeur peut changer lors de l'exécution du programme.

Une variable est caractérisé par 2 chose : son nom et son type



Comme en mathématique on peut faire des opérations (addition, soustraction, multiplication, division, etc.) avec ses variables.

1.3 Les constantes

```
Réchauffez le four à 200°
```

Peu importe le nombre d'œufs utilisés dans la recette, le gateau doit être cuit à la même température. Ceci est appelé une ***constante***.

1.4 Les conditions

```
si il y a plus de 5g de sucre alors on double la quantité de lait.
```

Ceci est un appelé une ***condition***.

Une condition indique comment réagir en fonction de différent paramètres.

1.5 Les boucles

```
tant que le gâteau n'est pas cuit

        Alors je le laisse au four

sinon
        Le gâteau est prêt

fin tant que
```

Ceci est appelé une ***boucle***.

Une boucle est une suite d'instruction qui va s'effectuer tant qu'une condition est remplie

```
Début program
    nombre cuissonFinit = faux

    tant que cuissonFinit = faux faire
            cuire()
    fin tant que

    afficher  " le gâteau est prêt ! "

Fin program
```

Plus tard on souhaite faire un gâteau au chocolat et un autre à la vanille.
Pour faire les 2 gâteau à chaque fois on devra faire la même chose en se basant sur les différents paramètres qui sont ici les ingrédients ; c'est ce qu'on appelle une ***fonction***.

1.6 Les fonctions

Une fonction est un sous-programme qui permet d'effectuer une suite d'instruction.
Notre recette permet d'obtenir un gâteau à partir de ses ingrédients.
En informatique cela est appeler un ***algorithme***.

SEQUENCE II : L'ALGORITHME

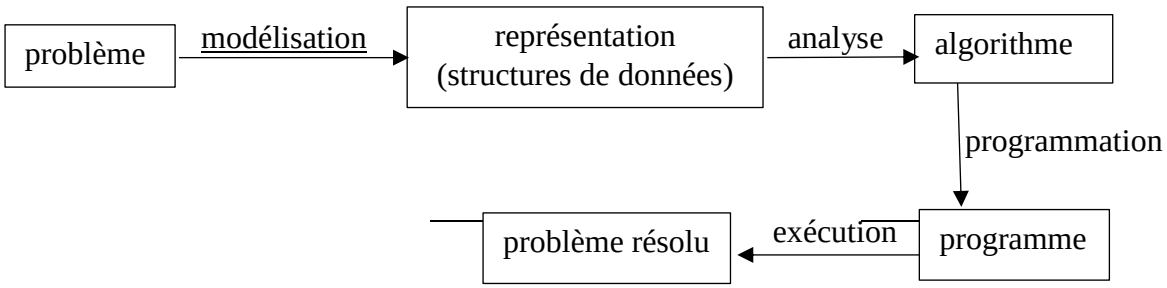
Q : Qu'est-ce qu'un algorithme ?

R : Un algorithme est la description d'une suite d'étapes permettant d'obtenir un résultat à partir d'élément fournis en entrée.

1. Instructions et types élémentaires

1.1. Introduction à l'algorithmique

Un algorithme est une étape préalable à l'écriture d'un programme informatique. Il décrit le résultat de l'analyse d'un problème (énoncé en français) dans un langage formel. Il se présente sous la forme d'une liste d'opérations permettant de réaliser un travail. L'ordre de ces opérations et leur enchaînement est très important. La figure suivante décrit les phases nécessaires à la résolution d'un problème.



Le « langage algorithmique» utilisé est un compromis entre un langage naturel et un langage de programmation. Un algorithme est une suite d'instructions présentées dans l'ordre des traitements. Les instructions sont des opérations composées de variables, de constantes et d'opérateurs. L'algorithme sera toujours accompagné d'un lexique qui indique, pour chaque variable, son type et son rôle. Un algorithme est délimité par les mots clés *début* et *fin*. Nous manipulerons les types couramment rencontrés dans les langages de programmation : *entier*, *réel*, *booléen*, *caractère*, *chaîne*, *tableau* et *type composite*.

1.2. Notions de variables, types et valeurs

Les variables d'un algorithme permettent de contenir les informations intermédiaires pour établir un raisonnement. Chaque variable a un nom (identifiant) et un type. Ce dernier correspond au genre d'information que l'on souhaite utiliser :

- **entier** pour manipuler des nombres entiers,
- **réel** pour manipuler des nombres réels,
- **booléen** pour manipuler des valeurs booléennes **vrai** ou **faux**,

- **caractère** pour manipuler des caractères alphabétiques et numériques,
- **chaîne** pour manipuler des chaînes de caractères permettant de représenter des mots ou des phrases.

Il faut noter qu'à un type donné, correspond un ensemble d'opérations définies pour ce type. Une variable est l'association d'un nom avec un type, permettant de mémoriser une valeur de ce type.

Le type entier

Les opérations utilisables sur les entiers sont :

- les opérateurs arithmétiques classiques : + (addition), - (soustraction), * (produit)
- la division entière, notée ÷, telle que $n \div p$ donne la partie entière du quotient de la division de n par p
- le modulo, notée mod, telle que $n \bmod p$ donne le reste de la division entière de n par p
- les opérateurs de comparaison classiques : <, >,=,≠, ≥, ≤

Le type réel

Les opérations utilisables sur les réels sont :

- les opérateurs arithmétiques classiques : + (addition), - (soustraction), * (produit), / (division)
- les opérateurs de comparaison classiques : <, >,=,≠, ≥, ≤

Le type booléen

Il s'agit du domaine dont les seules valeurs sont *vrai* ou *faux*. Les opérations utilisables sur les booléens sont réalisées à l'aide des connecteurs logiques : et (pour le *et* logique), ou (pour le *ou* logique), non (pour le *non* logique).

Rappel :

non	
vrai	faux
faux	vrai

et	*vrai*	*faux*
vrai	vrai	faux
faux	faux	faux

ou	*vrai*	*faux*
vrai	vrai	vrai
faux	vrai	faux

Le type caractère

Il s'agit du domaine constitué des caractères alphabétiques et numériques. Les opérations élémentaires réalisables sont les comparaisons : <, >,=,≠, ≥, ≤.

Le type chaîne

Une chaîne est une séquence de plusieurs caractères. Les opérations élémentaires réalisables sont les comparaisons : <, >,=,≠, ≥, ≤ selon l'ordre lexicographique.

1.3. Instructions d'affectation et expressions

Une instruction traduit une ou plusieurs actions portant sur une ou plusieurs variables. Ces actions sont relatives aux opérations admissibles sur les valeurs des variables. L'instruction la plus commune est l'affectation. Elle consiste à doter une variable d'une valeur appartenant à son domaine, c'est-à-dire à lui donner une première valeur ou à changer sa valeur courante. Elle se note par le signe ←.

Une expression est une suite finie bien formée d'opérateurs portant sur des variables ou des valeurs et qui a une valeur. La valeur de l'expression doit être conforme au domaine de la variable affectée.

Exemple d'algorithme

algorithme
début
x ← 12
y ← x + 4
x ← 3
fin
lexique
- x : entier
- y : entier

On remarque que les deux premières instructions ne sont pas permutables car *x* n'aurait alors pas de valeur au moment du calcul de l'expression.

Schéma de l'état des variables au cours de l'exécution :

~~12~~ 3		16
x		y

Schéma de l'évolution de l'état des variables instruction par instruction :

instructions / variables	1	2	3
x	12		3
y		16	

1.4. Instructions de lecture et d'écriture

Instruction de lecture

L'instruction de prise de données sur le périphérique d'entrée (en général le clavier) est :

variable ← **lire** ( )

L'exécution de cette instruction consiste à affecter une valeur à la variable en prenant cette valeur sur le périphérique d'entrée. Avant l'exécution de cette instruction, la variable avait ou n'avait pas de valeur. Après, elle a la valeur prise sur le périphérique d'entrée.

Instruction d'écriture

L'instruction de restitution de résultats sur le périphérique de sortie (en général l'écran) est:

écrire (liste d'expressions)

Cette instruction réalise simplement l'affichage des valeurs des expressions décrites dans la liste. Ces expressions peuvent être simplement des variables ayant des valeurs ou même des nombres ou des commentaires écrits sous forme de chaîne de caractères.

Exemple d'utilisation : écrire (x, y+2, "bonjour")

Exemple d'algorithme

Écrire un algorithme qui décompose une somme d'argents en billets de 100 euros, 50 euros et 10 euros, et de pièces de 2 euros et 1 euro. La somme sera lue au clavier, et les valeurs affichées une par ligne.

Principe :

L'algorithme commence par lire sur l'entrée standard l'entier qui représente la somme d'argent et affecte la valeur à une variable somme. Pour obtenir la décomposition en nombre de billets et de pièces de la somme d'argent , on procède par des divisions successives en conservant chaque fois le reste.

algorithme

```
début
 somme ← lire ( )                  // 1
 b100 ← somme ÷ 100                // 2
 r100 ← somme mod 100              // 3
 b50 ← r100 ÷ 50                   // 4
 r50 ← r100 mod 50                 // 5
 b10 ← r50 ÷ 10                    // 6
 r10 ← r50 mod 10                  // 7
 p2 ← r10 ÷ 2                      // 8
 r2 ← r10 mod 2                    // 9
 p1 ← r2                           // 10
écrire (b100, b50, b10, p2, p1)    // 11
fin
```

lexique

- somme : entier, la somme d'argent à décomposer
- b100: entier, le nombre de billets de 100 euros
- b50: entier, le nombre de billets de 50 euros
- b10: entier, le nombre de billets de 10 euros
- p2: entier, le nombre de pièces de 2 euros
- p1: entier, le nombre de pièces de 1 euro
- r100: entier, reste de la division entière de somme par 100
- r50: entier, reste de la division entière de r100 par 50
- r10: entier, reste de la division entière de r50 par 10
- r2: entier, reste de la division entière de r10 par 2

Schéma de l'évolution de l'état des variables instruction par instruction :

instructions / variables	1	2	3	4	5	6	7	8	9	10	11
somme	988										
b100		9									écrire
b50				1							écrire
b10						3					écrire
p2								4			écrire
p1										0	écrire
r100			88								
r50					38						
r10							8				
r2									0		

1.5. Notion de fonction

Une fonction est un algorithme autonome, réalisant une tâche précise. Il reçoit des paramètres (valeurs) en entrée et retourne une valeur en sortie (résultat). La notion de fonction est très intéressante car elle permet, pour résoudre un problème, d'employer une méthode de décomposition en sous-problèmes distincts. Elle facilite aussi la réutilisation d'algorithmes déjà développés par ailleurs.

Une fonction est introduite par un *en-tête*, appelé aussi *signature* ou *prototype*, qui spécifie:

- le nom de la fonction,
- les paramètres donnés et leur type,
- le type du résultat.

La syntaxe retenue pour l'en-tête est la suivante :

fonction nomFonction (liste des paramètres) : type du résultat

La liste des paramètres précise, pour chaque paramètre, son nom et son type. La dernière instruction de la fonction indique la valeur retournée, nous la noterons:

retourne expression

Exemple de fonction

Ecrire une fonction calculant le périmètre d'un rectangle dont on donne la longueur et la largeur.

```
fonction calculerPérimètreRectangle (longueur: réel, largeur : réel) : réel
   début
      périmètre ← 2*(longueur + largeur)
      retourne périmètre
   fin
lexique
```

- longueur : réel, longueur du rectangle
- largeur : réel, largeur du rectangle
- périmètre : réel, périmètre du rectangle

1.6. Instructions conditionnelles

Les exemples précédents montrent des algorithmes dont les instructions doivent s'exécuter dans l'ordre, de la première à la dernière. Nous allons introduire une instruction précisant que le déroulement ne sera plus séquentiel. Cette instruction est appelée une **conditionnelle**. Il s'agit de représenter une alternative où, selon les cas, un bloc d'instructions est exécuté plutôt qu'un autre. La syntaxe de cette instruction est :

```
si condition
        alors liste d'instructions
        sinon liste d'instructions
 fsi
```

Cette instruction est composée de trois parties distinctes : la condition introduite par *si*, la clause *alors* et la clause *sinon*. La condition est une expression dont la valeur est de type booléen. Elle est évaluée. Si elle est vraie, les instructions de la clause *alors* sont exécutées. Dans le cas contraire, les instructions de la clause *sinon* sont exécutées.

On peut utiliser une forme simplifiée de la conditionnelle, sans clause *sinon*. La syntaxe est alors :

si condition
    alors liste d'instructions
fsi

Exemple 1 (conditionnelle simple)

Ecrire un algorithme qui permet d'imprimer le résultat d'un étudiant à un module sachant que ce module est sanctionné par une note d'oral de coefficient 1 et une note d'écrit de coefficient 2. La moyenne obtenue doit être supérieure ou égale à 10 pour valider le module.

données : la note d'oral et la note d'écrit

résultat : impression du résultat pour le module (*reçu* ou *refusé*)

principe : on calcule la moyenne et on la compare à 10.

```
algorithme :
        début
        ne, no ← lire ( )
        moy← (ne * 2 + no) / 3
        si moy ≥ 10
            alors   écrire ("reçu")
            sinon   écrire ("refusé")
        fsi
        fin
lexique :
-   moy : réel, moyenne
-   ne : réel,  note d'écrit
-   no : réel,  note d'oral
```

Exemple 2 (conditionnelles imbriquées)

On veut écrire une fonction permettant de calculer le salaire d'un employé payé à l'heure à partir de son salaire horaire et du nombre d'heures de travail.

Les règles de calcul sont les suivantes : le taux horaire est majoré pour les heures supplémentaires : 25% au-delà de 160 h et 50% au-delà de 200 h.

```
fonction calculerSalaire ( sh : réel, nbh : entier) : réel
      début
      si nbh ≤ 160
      alors  salaire ← nbh * sh
      sinon   si  nbh ≤ 200
                        alors salaire ← 160 * sh + (nbh - 160) * sh * 1.25
                        sinon salaire ←160 * sh + 40 * sh * 1,25 + (nbh - 200) * sh * 1.5
              fsi
```

fsi
retourne salaire
fin

lexique

- sh : réel, salaire horaire de l'employé
- nbh : entier, nombre d'heures de travail de l'employé
- salaire : réel, salaire de l'employé

SEQUENCE III : GENERALITES SUR LE LANGAGE C

Dans ce chapitre, nous vous proposons une première approche d'un programme en langage C, basée sur deux exemples commentés. Vous y découvrirez (pour l'instant, de façon encore informelle) comment s'expriment les instructions de base (déclaration, affectation, lecture et écriture), ainsi que deux des structures fondamentales (boucle avec compteur, choix).

Nous dégagerons ensuite quelques règles générales concernant l'écriture d'un programme.

Enfin, nous vous montrerons comment s'organise le développement d'un programme en vous rappelant ce que sont **l'édition, la compilation, l'édition de liens et l'exécution.**

I. PRESENTATION PAR UN EXEMPLE, DE QUELQUES INSTRUCTIONS DU C

I.1. Exemple de programme

Voici un exemple de programme en langage C, accompagné d'un exemple d'exécution. Avant d'en lire les explications qui suivent, essayez d'en percevoir plus ou moins le fonctionnement.

```
#include<stdio.h>
#include<math.h>
#define NFOIS 5
main()
{
int i;
float x ;
float racx ;

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 < 0.0)
     printf("Le nombre %f ne possède pas de racine carrée \n", x) ;
  else
  {
    racx = sqrt(x) ;
```

```
    printf("Le nombre %f a pour racine carrée : %f\n", x, racx) ;
  }
 }
Printf("Travail terminé – Au revoir") ;
}
```

```
Bonjour
Je vais vous calculer 5 racines carrées
Donnez un nombre : 4
Le nombre 4.000000 a pour racine carrée : 2.000000
Donnez un nombre : 2
Le nombre 2.000000 a pour racine carrée : 1.414214
Donnez un nombre : -3
Le nombre -3.000000 ne possède pas de racine carrée
Donnez un nombre : 5.8
Le nombre 5.800000 a pour racine carrée : 2.408319
Donnez un nombre : 12.58
Le nombre 12.580000 a pour racine carrée : 3.546829 Travail
terminé - Au revoir
```

Nous reviendrons un peu plus loin sur le rôle des *trois premières lignes*. Pour l'instant, admettez simplement que le symbole **NFOIS** est équivalent à la valeur **5**.

I.2. Structure d'un programme en Langage C

La ligne :

main()

se nomme un « **en-tête** ». Elle précise que ce qui sera décrit à sa suite est en fait le ***programme principal*** **(main)**. Lorsque nous aborderons l'écriture des fonctions en C, nous verrons que celles-ci possèdent également un tel en-tête; ainsi, en C, le programme principal apparaîtra en fait comme une fonction dont le nom (main) est imposé.

Le programme (principal) proprement dit vient à la suite de cet en-tête. Il est délimité par les accolades **«{»** et **«}».** On dit que les instructions situées entre ces accolades forment un « bloc ». Ainsi peut-on dire que la fonction main est constituée d'un entête et d'un bloc; il en ira de

même pour toute fonction C. Notez qu'un bloc peut lui-même contenir d'autres blocs (c'est le cas de notre exemple). **En revanche, nous verrons qu'une fonction ne peut jamais contenir d'autres fonctions.**

I.3. Déclarations

Les trois instructions :

int i;

float x;

float racx;

sont des « déclarations ».

- La première précise que la variable nommée **i** est de type **int**, c'est-à-dire qu'elle est destinée à contenir des nombres entiers (relatifs). Nous verrons qu'en C il existe plusieurs types d'entiers.
- Les deux autres déclarations précisent que les variables **x** et **racx** sont de type **float**, c'est-à-dire qu'elles sont destinées à contenir des nombres flottants (approximation de nombres réels). Là encore, nous verrons qu'en C il existe plusieurs types flottants.

En C, comme dans la plupart des langages actuels, **les déclarations des types des variables sont obligatoires et doivent être regroupées au début du programme** (on devrait plutôt dire : au début de la fonction main). Il en ira de même pour toutes les variables définies dans une fonction; on les appelle « variables locales » (en toute rigueur, les variables définies dans notre exemple sont des variables locales de la fonction main). Nous verrons également (dans le chapitre consacré aux fonctions) qu'on peut définir des variables en dehors de toute fonction; on parlera alors de variables globales.

Remarque :

Une déclaration peut figurer à n'importe quel emplacement, pour peu qu'elle apparaisse avant que la variable correspondante ne soit utilisée.

I.4. Directives à destination du préprocesseur

Les trois premières lignes de notre programme :

#include<stdio.h>

#include<math.h>

#define NFOIS 5

sont en fait un peu particulières. Il s'agit de directives qui seront prises en compte avant la traduction (compilation) du programme. Ces directives, contrairement au reste du programme, doivent être écrites à raison d'une par ligne et elles doivent obligatoirement commencer en début de ligne. Leur emplacement au sein du programme n'est soumis à aucune contrainte (mais une directive ne s'applique qu'à la partie du programme qui lui succède). D'une manière générale, il est préférable de les placer au début, comme nous l'avons fait ici.

Les deux premières directives demandent en fait d'introduire (avant compilation) des instructions (en langage C) situées dans les fichiers ***stdio.h*** et ***math.h*****.** Leur rôle ne sera complètement compréhensible qu'ultérieurement. Pour l'instant, notez que, dès lors que vous faites appel à une fonction prédéfinie, il est nécessaire d'incorporer de tels fichiers, nommés « fichiers en-têtes », qui contiennent des déclarations appropriées concernant cette fonction :
stdio.h pour printf et
scanf, math.h pour sqrt.

Fréquemment, ces déclarations permettront au compilateur d'effectuer des contrôles sur le nombre et le type des arguments que vous mentionnerez dans l'appel de votre fonction.

Notez qu'un même fichier en-tête contient des déclarations relatives à plusieurs fonctions. ***En général, il est indispensable d'incorporer stdio.h.***

La troisième directive demande simplement de remplacer systématiquement, dans toute la suite du programme, le symbole NFOIS par 5. Autrement dit, le programme qui sera réellement compilé comportera ces instructions :

```
printf("Je vais vous calculer %d racines carrées\n", 5) ;
for (i=0; i<5; i++)
```

Notez toutefois que le programme proposé est plus facile à adapter lorsque l'on emploie une directive *define*.

II. QUELQUES REGLES D'ECRITURE

Ce paragraphe expose un certain nombre de règles générales intervenant dans l'écriture d'un programme en langage C. Nous y parlerons précisément de ce que l'on appelle les « **identificateurs** » et les « **mots-clés** », du format libre dans lequel on écrit les instructions, ainsi que de l'usage des **séparateurs** et des **commentaires**.

II.1. Identificateurs

Les identificateurs servent à désigner les différents « objets » manipulés par le programme : variables, fonctions, etc. (Nous rencontrerons ultérieurement les autres objets manipulés par le langage C : constantes, étiquettes de structure, d'union ou d'énumération, membres de structure ou d'union, types, étiquettes d'instruction GOTO, macros). Comme dans la plupart des langages, ils sont formés d'une suite de caractères choisis parmi les **lettres** ou les **chiffres**; **le premier d'entre eux étant nécessairement une lettre**.

En ce qui concerne les lettres :

- le caractère souligné (_) est considéré comme une lettre. Il peut donc apparaître au début d'un identificateur.

Voici quelques identificateurs corrects : lg_lig, valeur_5, _total, _89.

- les majuscules et les minuscules sont autorisées mais ne sont pas équivalentes. Ainsi, en C, les identificateurs **ligne** et **Ligne** désignent deux objets différents.

En ce qui concerne la longueur des identificateurs, la norme ANSI prévoit qu'au moins les 31 premiers caractères soient « significatifs » (autrement dit, deux identificateurs qui diffèrent, par leurs 31 premières lettres désigneront deux objets différents).

II.2. Mots-clés

Certains « mots-clés » sont réservés par le langage à un usage bien défini et ne peuvent pas être utilisés comme identificateurs. En voici la liste de quelques-uns:

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

II.3. Séparateurs

Dans notre langue écrite, les différents mots sont séparés par un espace, un signe de ponctuation ou une fin de ligne. Il en va quasiment de même en langage C dans lequel les règles vont donc paraître naturelles. Ainsi, dans un programme, deux identificateurs successifs entre lesquels la syntaxe n'impose aucun signe particulier (tel que : , =; * ( ) [ ] { }) doivent impérativement être séparés soit par un espace, soit par une fin de ligne. Par contre, dès que la syntaxe impose un

séparateur quelconque, il n'est alors pas nécessaire de prévoir d'espaces supplémentaires (bien qu'en pratique cela améliore la lisibilité du programme).

Ainsi, vous devrez impérativement écrire :
int x, y et non : **intx,y**

En revanche, vous pourrez écrire indifféremment :
int n,compte,total,p ou plus lisiblement : **int n, compte, total, p**

II.4. Format libre

Le langage C autorise une mise en page parfaitement libre. En particulier, une instruction peut s'étendre sur un nombre quelconque de lignes, et une même ligne peut comporter autant d'instructions que vous le souhaitez. Les fins de ligne ne jouent pas de rôle particulier, si ce n'est celui de séparateur, au même titre qu'un espace, sauf dans les « constantes chaînes » où elles sont interdites; de telles constantes doivent impérativement être écrites à l'intérieur d'une seule ligne. Un identificateur ne peut être coupé en deux par une fin de ligne, ce qui semble évident.

Bien entendu, cette liberté de mise en page possède des contreparties. Notamment, le risque existe, si l'on n'y prend garde, d'aboutir à des programmes peu lisibles.

À titre d'exemple, voyez comment pourrait être (mal) présenté notre programme précédent :

```
#include <stdio.h>
#include      <math.h>
#define NFOIS 5
main() { int i; float
x
; float racx; 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 < 0.0)
printf ("Le nombre %f ne possède pas de racine carrée\n", x); else {
racx = sqrt (x); printf ("Le nombre %f a pour racine carrée : %f\n", x,
racx); } } printf ("Travail terminé - Au revoir");}
```

II.5. Commentaires

Comme tout langage évolué, le langage C autorise la présence de commentaires dans vos programmes source. Il s'agit de textes explicatifs destinés aux lecteurs du programme et qui n'ont aucune incidence sur sa compilation.

Ils sont formés de caractères quelconques placés entre les symboles /* et */. Ils peuvent apparaître à tout endroit du programme où un espace est autorisé. En général, on se limitera à des emplacements propices à une bonne lisibilité du programme.

Voici quelques exemples de commentaires :

```
/* Programme de calcul de racines carrées */

/* commentaire fantaisiste &ç§{<>} ?%!!!!!! */

/* commentaire s'étendant
sur plusieurs lignes
de programme source*/

/* ============================================
* commentaire quelque peu esthétique *
* et encadré, pouvant servir,        *
* par exemple, d'en-tête de programme  *
============================================ */
```

Voici un exemple de commentaires qui, situés au sein d'une instruction de déclaration, permettent de définir le rôle des différentes variables :

```
int i;                          /* compteur de boucle */
float x;                        /* nombre dont on veut la racine carrée */ float
racx;           /* racine carrée du nombre */
```

Remarque :
Il est autorisé une seconde forme de commentaire, dit « de fin de ligne », que l'on retrouve également en C++. Un tel commentaire est introduit par //. Et tout ce qui suit ces deux caractères jusqu'à la fin de la ligne est considéré comme un commentaire. En voici un exemple :

```
printf ("Bonjour\n") ;                  // formule de politesse
```

III. LA CREATION D'UN PROGRAMME EN LANGAGE C

La manière de développer et d'utiliser un programme en langage C dépend naturellement de l'environnement de programmation dans lequel vous travaillez. Nous vous fournissons ici quelques indications générales (s'appliquant à n'importe quel environnement) concernant ce que l'on pourrait appeler les grandes étapes de la création d'un programme, à savoir : édition du programme, compilation et édition de liens.

III.1. Edition du programme

L'édition du programme (on dit aussi parfois « saisie ») consiste à créer, à partir d'un clavier, tout ou partie du texte d'un programme qu'on nomme « programme source ». En général, ce texte sera conservé dans un fichier que l'on nommera « fichier source».

Chaque système possède ses propres conventions de dénomination des fichiers. En général, un fichier peut, en plus de son nom, être caractérisé par un groupe de caractères (au moins 3) qu'on appelle une « extension » (ou, parfois un « type »); la plupart du temps, en langage C, les fichiers source porteront l'extension C.

III.2. Compilation

Elle consiste à traduire le programme source (ou le contenu d'un fichier source) en langage machine, en faisant appel à un programme nommé compilateur. En langage C, compte tenu de l'existence d'un préprocesseur, cette opération de compilation comporte en fait deux étapes :

- **traitement par le préprocesseur** : ce dernier exécute simplement les directives qui le concernent (il les reconnaît au fait qu'elles commencent par un caractère #). Il produit, en résultat, un programme source en langage C pur. Notez bien qu'il s'agit toujours d'un vrai texte, au même titre qu'un programme source : la plupart des environnements de programmation vous permettent d'ailleurs, si vous le souhaitez, de connaître le résultat fourni par le préprocesseur.
- **compilation proprement dite**, c'est-à-dire traduction en langage machine du texte en langage C fourni par le préprocesseur.

Le résultat de la compilation porte le nom de module objet.

III.3. Edition de liens

Le module objet créé par le compilateur n'est pas directement exécutable. Il lui manque, au moins, les différents modules objet correspondant aux fonctions prédéfinies (on dit aussi « fonctions standard ») utilisées par votre programme (comme printf, scanf, sqrt).

C’est effectivement le rôle de l’éditeur de liens que d’aller rechercher dans la bibliothèque standard les modules objet nécessaires. Notez que cette bibliothèque est une collection de modules objet organisée, suivant l’implémentation concernée, en un ou plusieurs fichiers.

Le résultat de l’édition de liens est ce que l’on nomme un programme exécutable, c’est-à-dire un ensemble autonome d’instructions en langage machine. Si ce programme exécutable est rangé dans un fichier, il pourra ultérieurement être exécuté sans qu’il soit nécessaire de faire appel à un quelconque composant de l’environnement de programmation en C.

III.4. Fichiers en-tête

Nous avons vu que, grâce à la directive **#include**, vous pouviez demander au préprocesseur d’introduire des instructions (en langage C) provenant de ce que l’on appelle des fichiers « en-tête ». De tels fichiers comportent, entre autres choses :

- des déclarations relatives aux fonctions prédéfinies,
- des définitions de macros prédéfinies.

Lorsqu’on écrit un programme, on ne fait pas toujours la différence entre fonction et macro, puisque celles-ci s’utilisent de la même manière. Toutefois, les fonctions et les macros sont traitées de façon totalement différente par l’ensemble « préprocesseur + compilateur + éditeur de liens ».

En effet, les appels de macros sont remplacés (par du C) par le préprocesseur, du moins si vous avez incorporé le fichier en-tête correspondant. Si vous ne l’avez pas fait, aucun remplacement ne sera effectué, mais aucune erreur de compilation ne sera détectée : le compilateur croira simplement avoir à faire à un appel de fonction; ce n’est que l’éditeur de liens qui, ne la trouvant pas dans la bibliothèque standard, vous fournira un message.

Les fonctions, quant à elles, sont incorporées par l’éditeur de liens. Cela reste vrai, même si vous omettez la directive #include correspondante; dans ce cas, simplement, le compilateur n’aura pas disposé d’informations appropriées permettant d’effectuer des contrôles d’arguments (nombre et type) et de mettre en place d’éventuelles conversions. Aucune erreur ne sera signalée à la compilation ni à l’édition de liens. Les conséquences n’apparaîtront que lors de l’exécution : elles peuvent être invisibles dans le cas de fonctions comme **printf** ou, au contraire, conduire à des résultats erronés dans le cas de fonctions comme **sqrt**.

IV. OPERATEURS ET EXPRESSION

IV.1. Présentation des opérateurs

Comme tous les langages, C dispose :

- d'opérateurs classiques « binaires » (c'est-à-dire portant sur deux « opérandes »), à savoir l'addition (+), la soustraction (-), la multiplication (*) et la division (/), ainsi que
- d'un opérateur « unaire » (c'est-à-dire ne portant que sur un seul opérande) correspondant à l'opposé noté - (comme dans -n ou dans -x+y).

Les opérateurs **binaires** ne sont à priori définis que pour deux opérandes ayant le même type parmi : **int**, **long int**, **float**, **double**, **long double** et ils fournissent un résultat de même type que leurs opérandes.

Remarque :
En machine, il n'existe, par exemple, que des additions de deux entiers de même taille ou de flottants de même taille. Il n'existe pas d'addition d'un entier et d'un flottant ou de deux flottants de taille différente.
De plus, il existe un opérateur de modulo noté % qui ne peut porter que sur des entiers et qui fournit le reste de la division de son premier opérande par son second. Par exemples, 11%4 vaut 3; 23%6 vaut 5.
La norme ANSI ne définit les opérateurs % et / que pour des valeurs positives de leurs deux opérandes. Dans les autres cas, le résultat dépend de l'implémentation.
Notez bien qu'en C le quotient de deux entiers fournit un entier. Ainsi, 5/2 vaut 2; en revanche, le quotient de deux flottants (noté, lui aussi, /) est bien un flottant (5.0/2.0 vaut bien approximativement 2.5).

Il n'existe pas d'opérateur d'élévation à la puissance. Il est nécessaire de faire appel soit à des produits successifs pour des puissances entières pas trop grandes (par exemple, on calculera x3 comme x*x*x), soit à la fonction power de la bibliothèque standard.

IV.2. Priorités relatives des opérateurs

Les opérateurs unaires **+** et **-** ont la priorité la plus élevée. On trouve ensuite, à un même niveau, les opérateurs *, **/ et %**. Enfin, sur un dernier niveau, apparaissent les opérateurs binaires + et -.

En cas de priorités identiques, les calculs s'effectuent de gauche à droite. On dit que l'on a à faire à une associativité de gauche à droite.

Enfin, des parenthèses permettent d'outrepasser ces règles de priorité, en forçant le calcul préalable de l'expression qu'elles contiennent. Notez que ces parenthèses peuvent également être employées pour assurer une meilleure lisibilité d'une expression.

Voici quelques exemples dans lesquels l'expression de droite, où ont été introduites des parenthèses superflues, montre dans quel ordre s'effectuent les calculs (les deux expressions proposées conduisent donc aux mêmes résultats) :

a + b * c	a + (b * c)
a * b + c % d	(a * b) + (c % d)
- c % d	(- c) % d
- a + c % d	(- a) + (c % d)
- a / - b + c	((- a) / (- b)) + c
- a / - (b + c)	(- a) / (- (b + c))

IV.3. Conversions d'ajustement de type

Une conversion telle que **int → float** se nomme une « **conversion d'ajustement de type** ». Une telle conversion ne peut se faire que suivant une hiérarchie qui permet de ne pas dénaturer la valeur initiale (on dit parfois que de telles conversions respectent l'intégrité des données). A savoir : **int → long → float → double → long double**

On peut bien sûr convertir directement un int en double. Par contre, on ne pourra pas convertir un double en float ou en int.

IV.4. Opérateurs relationnels

Comme tout langage, le C permet de comparer des expressions à l'aide d'opérateurs classiques de comparaison. En voici un exemple : **2 * a > b + 5**

Mais le C se distingue de la plupart des autres langages sur deux points :

le résultat de la comparaison est, non pas une valeur booléenne (on dit aussi logique) prenant l'une des deux valeurs vrai ou faux, mais un entier valant :

- 0 si le résultat de la comparaison est faux, o 1 si le résultat de la comparaison est vrai. Ainsi, la comparaison ci-dessus devient en fait une expression de type entier. Cela signifie qu'elle pourra éventuellement intervenir dans des calculs arithmétiques

OPERATEUR	SIGNIFICATION
<	inférieur à
< =	inférieur ou égale à
>	supérieur à
> =	supérieur ou égale à
= =	égale à
! =	différent de

IV.5. Opérateurs logiques

Le C dispose de trois opérateurs logiques classiques : **et** (noté **&&**), **ou** (noté ||) et **non** (noté **!**).

Par exemples :

- **(a<b) && (c<d)**

  prend la valeur 1 (vrai) si les deux expressions a < b et c < d sont toutes deux vraies (de valeur 1), et prend la valeur 0 (faux) dans le cas contraire.

- **(a<b) | | (c<d)**

  prend la valeur 1 (vrai) si l'une au moins des deux conditions a < b et c < d est vrai (de valeur 1), et prend la valeur 0 (faux) dans le cas contraire.

- **! (a<b)**

  Prend la valeur 1 (vrai) si la condition a < b est fausse (de valeur 0) et prend la valeur 0 (faux) dans le cas contraire. Cette expression est équivalente à : a >= b.

Il est important de constater que, ne disposant pas de type logique, le C se contente de représenter vrai par 1 et faux par 0. C'est pourquoi ces opérateurs produisent un résultat numérique (de type int).

IV.6. Opérateurs d'affectation ordinaire

Nous avons déjà eu l'occasion de remarquer que : **i = 5** était une expression qui :

- réalisait une action : **l'affectation de la valeur 5 à i**,
- possédait une valeur : celle de i après affectation, c'est-à-dire 5.

Cet opérateur d'affectation (=) peut faire intervenir d'autres expressions comme dans: **c=b + 3**

La faible priorité de cet opérateur = (elle est inférieure à celle de tous les opérateurs arithmétiques et de comparaison) fait qu'il y a d'abord évaluation de l'expression b + 3. La valeur ainsi obtenue est ensuite affectée à c.

Il n'est par contre pas possible de faire apparaître une expression comme premier opérande de cet opérateur =. Ainsi, l'expression suivante n'aurait pas de sens : **c + 5 = x**

IV.7. Opérateurs d'incrémentation et de décrémentation

Dans des programmes écrits dans un langage autre que C, on rencontre souvent des expressions (ou des instructions) telles que :
i = i + 1
n = n - 1

qui incrémentent ou qui décrémentent de 1 la valeur d'une variable. En C, ces actions peuvent être réalisées par des opérateurs « unaires ». Ainsi, l'expression : **++i** a pour effet d'incrémenter de 1 la valeur de i, et sa valeur est celle de i **après incrémentation**. Là encore, comme pour l'affectation, nous avons à faire à une expression qui non seulement possède une valeur, mais qui, de surcroît, réalise une action (incrémentation de i).

Il est important de voir que la valeur de cette expression est celle de i après incrémentation.
Ainsi, si la valeur de i est 5, l'expression : **n = ++i – 5**
affectera à **i la valeur 6** et **à n la valeur 1**.

En revanche, lorsque cet opérateur est placé après la variable sur laquelle il porte, la valeur de l'expression correspondante est celle de la variable **avant incrémentation**.

Ainsi, si i vaut 5, l'expression : **n = i++ - 5**
affectera à **i la valeur 6** et à **n la valeur 0** (car ici la valeur de l'expression i++ est 5).
On dit que ++ est :

- un opérateur de **préincrémentation** lorsqu'il est placé à gauche de la *lvalue* sur laquelle il porte,
- un opérateur de **postincrémentation** lorsqu'il est placé à droite de la *lvalue* sur laquelle il porte.

De la même manière, il existe un opérateur de décrémentation noté -- qui, suivant les cas, sera :

- un opérateur de **prédécrémentation** lorsqu'il est placé à gauche de la *lvalue* sur laquelle il porte,
- un opérateur de **postdécrémentation** lorsqu'il est placé à droite de la *lvalue* sur laquelle il porte.

IV.8. Opérateurs conditionnel

Considérons l'instruction suivante :

```
if (a>b)
        max = a;
else
        max = b;
```

Elle attribue à la variable max la plus grande des deux valeurs de a et de b. La valeur de max pourrait être définie par cette phrase : **Si a>b alors a sinon b**

En langage C, il est possible, grâce à l'aide de **l'opérateur conditionnel**, de traduire presque littéralement la phrase ci-dessus de la manière suivante : **max = a>b? a : b**

Voici un autre exemple d'une expression calculant la valeur absolue de 3*a + 1 : 3*a+1 > 0 ? 3*a+1 : -3*a-1

De même, rien n'empêche que l'expression conditionnelle soit évaluée sans que sa valeur soit utilisée comme dans cette instruction : a>b ? i++ : i--;

Ici, suivant que la condition a>b est vraie ou fausse, on incrémentera ou on décrémentera la variable i.

Exercice

1-) Soit les déclarations suivantes :

int n = 10 , p = 4;

long q = 2; float x = 1.75;

Donner le type et la valeur de chacune des expressions suivantes :

a) n + q

b) n + x

c) n % p + q

d) n < p

e) n >= p

f) n > q

g) q + 3 * (n > p)

h) q && n

i) (q-2) && (n-10)

j) x * (q==2)

k) x *(q=5)

2-) Écrire plus simplement l'instruction suivante :

z = (a>b ? a : b) + (a <= b ? a : b);

3-) n étant de type int, écrire une expression qui prend la valeur :

-1 si n est négatif,

0 si n est nul,

1 si n est positif.

4-) Quels résultats fournit le programme suivant ?

```
#include <stdio.h>
main()
{
int n=10, p=5, q=10, r;
r = n == (p = q);
printf ("A : n = %d p = %d q = %d r = %d\n", n, p, q, r);
n = p = q = 5;
n += p += q;
printf ("B : n = %d p = %d q = %d\n", n, p, q);
q = n < p ? n++ : p++;
printf ("C : n = %d p = %d q = %d\n", n, p, q);
q = n > p ? n++ : p++;
printf ("D : n = %d p = %d q = %d\n", n, p, q);
```

5-) Vous devez développer une calculatrice simple en langage C qui peut effectuer les opérations d'addition, de soustraction, de multiplication et de division sur deux nombres saisis par l'utilisateur. La calculatrice devrait afficher un menu avec les options disponibles et permettre à l'utilisateur de choisir l'opération qu'il souhaite effectuer. Une fois l'opération

sélectionnée, la calculatrice devra demander à l'utilisateur d'entrer deux nombres, effectuer l'opération choisie et afficher le résultat.

Remarque :

- Le menu devrait inclure les options suivantes :

Addition
Soustraction
Multiplication
Division
Quitter

- Les opérations doivent être réalisées dans des fonctions distinctes.

- Gérez le cas de la division par zéro.

- Utilisez des fonctions pour effectuer les opérations.

*__Utiliser ses fonctions__ :

```
// Fonction pour effectuer l'addition
float addition(float a, float b) {
    return a + b;
}
// Fonction pour effectuer la soustraction
float soustraction(float a, float b) {
    return a - b;
}
// Fonction pour effectuer la multiplication
float multiplication(float a, float b) {
    return a * b;
}
// Fonction pour effectuer la division
float division(float a, float b) {
    if (b != 0) {
        return a / b;
    } else {
        printf("Division par zero impossible.\n");
        return 0;
    }
}
```

***Solution** :

```
#include <stdio.h>
int main() {
   int choix;
   float nombre1, nombre2;

   do {
      printf("Menu :\n");
      printf("1. Addition\n");
      printf("2. Soustraction\n");
      printf("3. Multiplication\n");
      printf("4. Division\n");
      printf("5. Quitter\n");
      printf("Choisissez une option : ");
      scanf("%d", &choix);

      if (choix >= 1 && choix <= 4) {
         printf("Entrez deux nombres : ");
         scanf("%f %f", &nombre1, &nombre2);

         switch (choix) {
            case 1:
               printf("Resultat : %.2f\n", addition(nombre1, nombre2));
               break;
            case 2:
               printf("Resultat : %.2f\n", soustraction(nombre1, nombre2));
               break;
            case 3:
               printf("Resultat : %.2f\n", multiplication(nombre1, nombre2));
               break;
            case 4:
               printf("Resultat : %.2f\n", division(nombre1, nombre2));
               break;
         }
      } else if (choix != 5) {
         printf("Option invalide. Veuillez reessayer.\n");
      }

   } while (choix != 5);

   printf("Au revoir !\n");

   return 0;
}
```