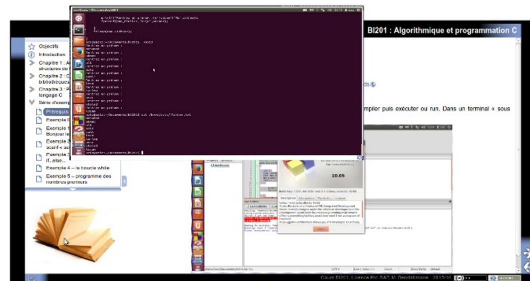
BI201 : Algorithmique et programmation C



El Hachemi BENDAHMANE

05/06/2016
v2.0

Table des matières

Objectifs **4**

Introduction **5**

I - Chapitre 1 : Algorithmes et structures de données **6**

1. Définitions ... 6
   - *1.1. Algorithme ... 6*
   - *1.2. Organigramme ... 7*
   - *1.3. Langage de description d'algorithme (Pseudocode) ... 11*
   - *1.4. Types : constantes, variables et structures ... 11*
   - *1.5. Opérateurs arithmétiques ... 13*
   - *1.6. Procédures et fonctions ... 13*
   - *1.7. Entrées/Sorties ... 14*
   - *1.8. Expressions conditionnelles complexes ... 14*
   - *1.9. Branchements multiples ... 15*
   - *1.10. Boucles ... 15*

2. Complexité des algorithmes ... 17
   - *2.1. Définitions ... 17*

3. Principes de conception d'algorithmes ... 18
   - *3.1. Conception d'algorithme ... 18*
   - *3.2. Pratique du développement d'algorithmes ... 20*

II - Chapitre 2 : Compilateurs, liens et bibliothèques **21**

1. Etapes de génération des fichiers binaires ... 21
   - *1.1. Préprocesseur ... 21*
   - *1.2. Compilation ... 21*
   - *1.3. Assemblage ... 22*
   - *1.4. Édition de liens (LINK) ... 22*

2. Types de fichier binaires exécutables ... 23
   - *2.1. Exécutable ... 23*
   - *2.2. Fichiers LIB (édition de lien Statique) ... 23*
   - *2.3. Dynamic link library (DLL) ... 23*
   - *2.4. COM (Component Object Model) ... 24*
   - *2.5. Les fichiers compilés .NET ... 24*

III - Chapitre 3 : Programmation en langage C **25**

1. Les atouts du C ... 26

2. Exécution d'un programme C ... 26

3. Notion de fonction en C ... 29

4. Définitions de base ... 29

5. Exercice : Calcul de n ! ... 33

6. Les structures de contrôle (Instructions conditionnelles) ... 33
   - *6.1. Les tests if ... 34*
   - *6.2. Instructions itératives (Les boucles) ... 34*

7. Les fonctions 37

8. Passage des arguments 39

9. Exercice : Puissances entières et type long 40

10. Tableaux, pointeurs et structures 40

10.1. Tableaux 40
10.2. Structures 42
10.3. Les énumérations de constantes 43
10.4. Pointeurs 43

11. Exercice : Tri des tableaux 45

12. Exercice : Pointeurs 46

13. Conseils pour écrire des programmes en C 47

13.1. Modularité 47
13.2. Pratique 47

IV - Série d'exemples en C **48**

1. Prérequis 49

2. Exemple 0 - rappels 50

3. Exemple 1 - programme Bonjour le monde 50

4. Exemple 2 – utilisation de scanf « saisie au clavier » 51

5. Exemple 3 – utilisation de if...else... 51

6. Exemple 4 – la boucle while 51

7. Exemple 5 – programme des nombres premiers 52

8. Exemple 6 - utilisation des arguments en ligne de commandes 53

9. Exemple 7 – les tableaux 54

10. Exemple 8 – les fonctions 55

11. Exemple 9 - les commentaires 55

12. Exemple 10 - les structures en C 56

13. Exemple 11 – la serie de Fibonacci 57

14. Exemple 12 – les fichiers 58

Bibliographie **59**

Objectifs

Pré-recquis :

Ce cours est destiné à des étudiants de 2ème année de licence pro en réseaux et télécoms du département d'électronique de l'université de Constantine 1. Il est souhaitable pour le suivre dans de bonnes conditions d'avoir quelques pré-requis en informatique de base, en algèbre de bool, en structure des ordinateurs et éventuellement des notions en algorithmique (avec le langage Pascal par exemple). Seuls de brefs rappels seront fait lorsque cela sera nécessaire.

Objectifs :

Les principaux objectifs de ce cours sont :

- apprendre à concevoir des algorithmes pour résoudre des problèmes réels ;
- transcrire ces algorithmes dans un langage structuré en l'occurrence le langage C ;
- compiler, le cas échéant corriger et tester des programmes C.

Introduction

Le présent cours fait partie du module BI201 qui est une initiation à l'informatique et à la programmation. Il permettra à l'étudiant de connaître le matériel et les systèmes informatiques largement utilisé dans le reste des modules de la formation R&T. Dans la continuité de ce module, nous nous concentrons dans le cadre de ce cours sur les algorithmes et comment résoudre des problèmes réels avec de la programmation et de l'algorithmique. On verra aussi les différents langages de programmation et particulièrement en détail la programmation en langage C, dans un environnement de programmation et de débogage. Nous choisirons l'IDE *CODE : :BLOCKS.*

Enfin, ce cours sera aussi utile à toute personne désireuse de se former par elle-même en informatique et en programmation.

Ce cours est organisé comme suit :

- Le premier chapitre introduit le concept d'algorithme et de structures de données en algorithmique. Nous parlerons d'organigrammes et de pseudocodes, nous aborderons aussi les notions de types de variables et de structures, de fonctions et procédures, d'opérateurs, de lectures et d'écriture, nous présenterons les expression conditionnelles, les branchements et les boucles. Nous parlerons aussi de la complexité algorithmique et conclurons par la présentation de quelques principes utiles et nécessaires en pratique pour la conception d'algorithmes.

- Dans *le chapitre 2*, nous verrons les étapes de compilation, d'assemblage, d'édition de liens et d'exécution de programmes. Nous parlerons aussi de la notion de bibliothèques et de fichiers binaires en informatique.

- Dans *le chapitre 3*, nous présenterons rapidement le langage de programmation C, ses atouts, les notions fondamentales pour le C, nous présenterons les structures de contrôle : les tests if et les boucles, nous verrons notamment les fonctions, le passage des arguments, les tableaux, les structures et les pointeurs. Nous donnons à chaque étape des exemples et des exercices pour approfondir la compréhension et la maîtrise des notions introduites. Nous finirons par donner quelques conseils utiles dans la pratique de l'écriture de programmes en C.

- *Un dernier chapitre* , est consacré au travaux dirigés sous forme d'une série d'exercices avec corrigés.

Chapitre 1 : Algorithmes et structures de données

Définitions	6
Complexité des algorithmes	17
Principes de conception d'algorithmes	18

En informatique, la solution unique à un problème n'existe pas. Il est très rare qu'il y ait une solution unique à un problème donné. Tout le monde a sa version d'un programme; contrairement aux mathématiques où une solution s'impose relativement facilement. En plus, un problème est traitable par informatique seulement si :

- L'on peut parfaitement définir les données et les résultats ;
- L'on peut décomposer le passage de ces données vers ces résultats en une suite d'opérations élémentaires dont chacune peut être exécutée par une machine.

Dans un cours de programmation, on ne travaille que sur des problèmes ayant déjà des algorithmes. C'est pourquoi dans la plupart des exercices, les algorithmes sont déjà spécifiés dans l'énoncé, ou leur spécification est triviale. Dans le travail de recherche, il est généralement demandé aux chercheurs de trouver ou d'inventer des algorithmes.

1. Définitions

1.1. Algorithme

Définition

Un algorithme est un ensemble de règles qui décrivent comment résoudre un problème donné. La définition précise d'un algorithme est "une suite finie de règles à appliquer dans un ordre déterminé à un nombre fini de données pour arriver, en un nombre fini d'étapes, à un certain résultat, et cela indépendamment des données".

Pour un ordinateur, un algorithme est une séquence bien définie d'opérations et de règles (calcul, manipulation de données, etc.) qui à partir d'un état initial, se termine à un état défini.

Complément

L'algorithme doit être défini rigoureusement : spécifier la façon d'appliquer les opérations dans toutes les circonstances qui peuvent se produire. Donc, les cas particuliers doivent être traités cas par cas et les critères de chaque cas doivent être spécifiés clairement.

L'ordre de calcul est aussi important pour un algorithme. Les instructions sont habituellement considérées comme listées explicitement et traitées comme commençant du début à la fin de l'algorithme.

Les algorithmes sont représentés par plusieurs notations : langage naturel, pseudocode, organigramme (flowcharts), et langages de programmation.

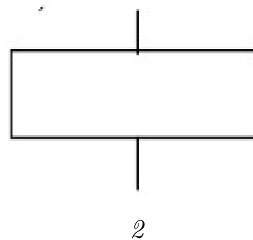
Pseudocode et organigrammes sont des façons structurés pour exprimer les algorithmes qui permettent d'éviter beaucoup d'ambiguïtés des langages naturels, tout en étant indépendant des langages de programmation utilisés pour implémenter l'algorithme.

1.2. Organigramme

Définition

Un organigramme est une représentation graphique de l'algorithme. Il utilise un ensemble de symboles normalisés (ISO 7805). Les étapes de l'algorithme sont représentés sous forme de boites reliés par des flèches. Le paragraphe suivant montre les principaux symboles :

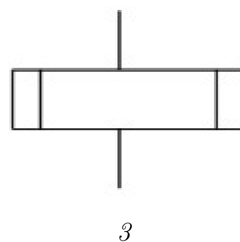
Complément : Procédé (symbole général de traitement)



2

On l'utilise pour représenter une opération (un groupe d'opérations) sur des données ou pour les instructions pour lesquelles il n y a pas de symbole normalisé.

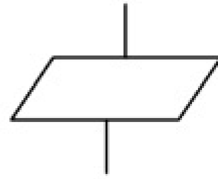
Complément : Sous programme (procédé défini)



3

Ce symbole est utilisé pour représenter une portion de programme considérée comme une simple opération (sous programme, procédure, fonction).

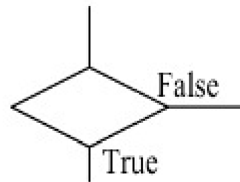
Complément : Donnée (Entrée/sortie)



4

Ce symbole est utilisé pour représenter la mise à la disposition du programme d'une donnée à traiter ou une opération d'enregistrement d'une donnée à effectuer.

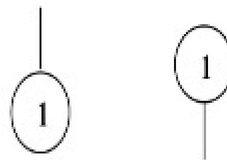
Complément : Branchement (décision)



5

Ce symbole est utilisé pour représenter le test et l'exploitation d'une condition pour faire la sélection entre deux choix. Les étiquettes False, True, Yes, No; doivent obligatoirement être présentes. Sinon un petit cercle indique le choix pour le résultat faux. Une autre convention aussi consiste à ce que la flèche sur le coté est le résultat du test lorsqu'il est faux et celui vers le bas est le résultat du test lorsqu'il est vrai.

Complément : Renvoi (connecter)



7

Ce symbole est utilisé pour assurer la continuité d'une ligne de liaison pour représenter un organigramme qui a une grande taille sur plusieurs pages par exemple. Ce symbole est utilisé deux fois avec le même numéro.

Complément : Début et fin d'un organigramme

8

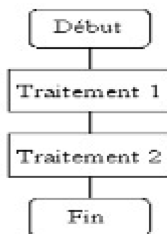
Ce symbole est utilisé pour représenter le debut ou lafin de l'algorithme. Il contient généralement le mot debut (Start) ou fin (End), ou toute autre phrase indiquant la même chose.

Structures complexes et boucles

Pour construire un organigramme, les symboles sont reliés par des lignes (flèches). La flèche indique que le contrôle passe du traitement duquel la flèche commence au traitement ou se termine la flèche.

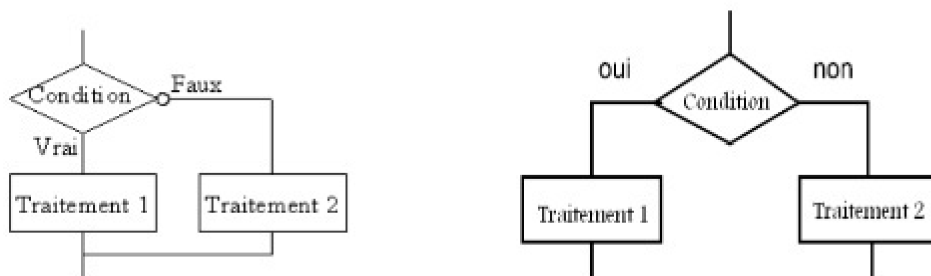
Les opérations élémentaires peuvent, en fonction de leur enchaînement, être organisées suivant quatre familles de structures algorithmiques fondamentales.

La structure de l'organigramme peut être une simple séquence linéaire. Elle consiste en un bloc d'opérations de traitement consécutives inconditionnelles.



9

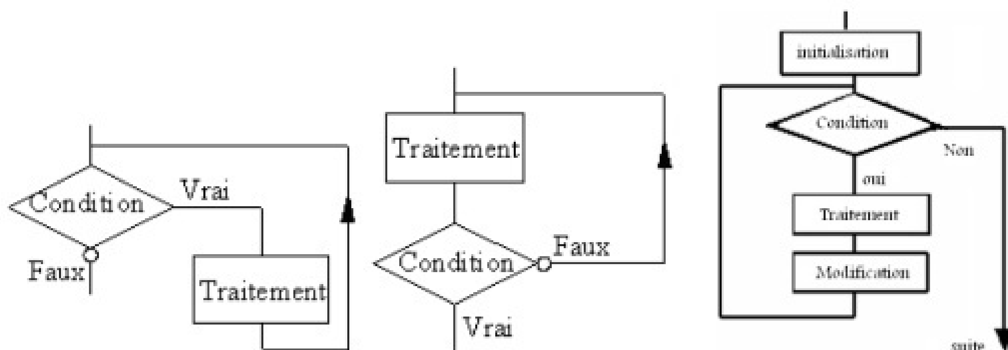
Quand il y a des séquences alternatives, la structure devient un peu complexe. Dans ce cas, une condition est testée. Si elle est vraie le traitement 1 est effectué sinon le traitement 2 est effectué. Ce type de structure est pratiquement toujours utilisé au moins pour voir est ce que le programme a terminer son traitement ou non.



10

La structure alternative réduite se distingue de la précédente par le fait que seule la situation correspondant à la validation de la condition entraîne l'exécution du traitement; l'autre situation conduisant systématiquement à la sortie de la structure.

La séquence d'instruction la plus complexe est la séquence répétitive ou boucle qui consiste à répéter un traitement plusieurs fois. On peut répéter un traitement donné tant qu'une certaine condition n'est pas satisfaite (boucle while à gauche). Il faut remarquer que puisque le test de la condition se fait avant le bloc d'instructions, celui-ci n'est pas forcément exécuté. Le traitement peut se faire aussi avant le test comme pour la boucle (do while au milieu). Dans ce cas, le test se fait après le bloc d'instructions, celui-ci est exécuté au moins une fois. La deuxième boucle consiste à répéter le traitement jusqu'à ce qu'une certaine condition est satisfaite (boucle For à droite).

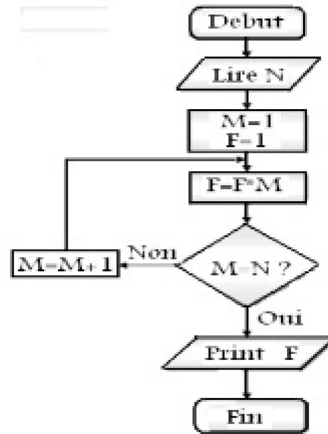


12

Exemple : Organigramme du calcul de N!

Le premier exemple est le calcul du factoriel d'un nombre naturel. $N! = 1x2x3\ x...x\ N$.

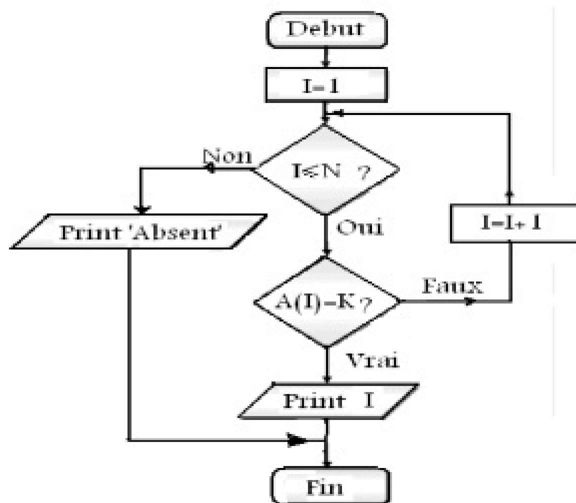
Il fonctionne comme suit (algorithme naïve): on commence par lire N, on utilise deux valeurs M et F. La valeur du factoriel F est initialisé à 1. La valeur M est utilisée comme compteur de boucle qui est incrémenté de 1 jusqu'à N et chaque fois elle est multipliée par F. le résultat de la multiplication est toujours stockée dans F. à la fin, le résultat F est affiché ou imprimé.



13

Exemple : Organigramme de recherche d'un nombre dans un tableau

Le deuxième organigramme est pour la recherche d'une valeur donnée (k) dans une suite représentée par un tableau (A) non vide à une dimension de longueur connue (N).



14

Ces problèmes, fort simples, conduisent à des organigrammes simples. Mais la complexité des problèmes et surtout des solutions trouvées induit souvent des organigrammes où les lignes s'entrecroisent rendant la lecture de l'organigramme très difficile.

1.3. Langage de description d'algorithme (Pseudocode)

Définition

Le pseudocode utilise un ensemble de mots clés (langage) et de structures permettant de décrire de manière complète et claire l'ensemble des opérations à exécuter sur des données pour obtenir des résultats. C'est une composition séquentielle finie d'opérations comme celle de l'organigramme. Donc, tout algorithme représenté par un organigramme peut être représenté à l'aide du pseudocode. L'avantage d'un tel langage est de pouvoir être facilement transcrit dans un langage de programmation procédurale (Pascal, C...)

Structure du pseudocode de l'algorithme

Un algorithme écrit en pseudocode est constitué de quatre parties :

1. *L'en-tête* : Il permet tout simplement d'identifier un algorithme.
2. *Les déclarations* : C'est une liste exhaustive des objets, grandeurs utilisés et manipulés dans le corps de l'algorithme; cette liste est placée en début d'algorithme. Elle contient les variables, les constantes et les fonctions.
3. *Le corps* : Dans cette partie de l'algorithme, on place les traitements à effectués.
4. *Les commentaires* : c'est du texte qui ne fait pas partie du programme final exécuté par l'ordinateur mais utilisé seulement pour permettre une lecture aisée de l'algorithme.

Exemple : Pseudocode d'un algorithme

```
Algorithme nom_de_l'algorithme ;
const
liste des constantes ;
var
liste des variables ;
struct
liste des structures ;
fonc
liste des fonctions ;
proc
liste des procédures ;
début
traitement 1 ;
traitement 2 ;
.
.
.
Traitement n ;
fin algorithme
```

1.4. Types : constantes, variables et structures

Définition : Constantes

Les constantes représentent des chiffres, des nombres, des caractères, des chaînes de caractères,... dont la valeur ne peut pas être modifiée au cours de l'exécution de l'algorithme.

Définition : Variable

Les variables peuvent stocker des chiffres, des nombres, des caractères, des chaînes de caractères,... dont la valeur peut être modifiée au cours de l'exécution de l'algorithme.

Une variable ou une constante est un "endroit" pour contenir une valeur ou plusieurs valeurs. Ils sont définis dans la partie déclarative de l'algorithme par deux caractéristiques essentielles, à savoir :

- *L'identificateur* : c'est le nom de la variable ou de la constante. Il est composé de lettres et de chiffres;
- *Le type* : il détermine la nature de la variable ou de la constante. Nous considérerons quatre types de base :
  - *L'entier* : pour représenter les nombres entiers. Exemple : 45, 36, −564, 0 ... en décimal, 45h, 0FBh, 64h ... en hexadécimal, % 10101111, %1011 ... en binaire ;
  - *Le réel* : pour représenter les nombre fractionnaires comme 3.1415 ou 3 e-9... ;
  - *Le booléen* : Il ne peut prendre que deux états : VRAI ou FAUX ;
  - *Le caractère et chaîne de caractères* : ‘a', ‘A','*','7','z',' !'...., 'mohamed', 'SOMME',...

Tous les langages, quels qu'ils soient, offrent un certain nombre de types de données, dont le détail est susceptible de varier légèrement d'un langage à l'autre. Grosso modo, on retrouve cependant les types suivants :

Type de donnée	Plage
Byte (octet)	0 à 255
Entier simple précision	-32 768 à 32 767
Entier long	-2 147 483 648 à 2 147 483 647
Réel simple précision	-3,40E38 à -1,40E-45 pour les valeurs négatives 1,40E-45 à 3,40E38 pour les valeurs positives
Réel double précision	1,79E308 à -4,94E-324 pour les valeurs négatives 4,94E-324 à 1,79E308 pour les valeurs positives

15

Il n'est cependant pas toujours nécessaire de déclarer les variables, c'est à dire de leur donner un type de données. Déclarer une variable par un type c'est présupposer que le langage est typé; or tous les langages ne le sont pas.

Typer une variable peut avoir des problèmes avec les langages et compilateurs car les types sont bornés. Par contre, il n'y a aucune raison de supposer dans l'algorithme que les mathématiques "normales" ne s'appliquent pas, et donc en algorithmique, 32767+1 doit toujours faire 32768.

L'affectation (ou assignation ←) qui s'écrit de la façon suivante :

Nom_de_variable valeur (ou expression calculable);

Permet de donner la valeur de l'expression de droite à la variable de gauche. Elle se fait toujours en deux temps :

1. évaluation de l'expression située à droite du symbole
2. affectation du résultat à l'identificateur de variable

Exemple : Équation du second degré

Cet exemple calcule les solutions de l'équation du second degré `ax2+bx+c = 0`, avec a et c de signe contraire :

```
delta  b*b - 4*a*c
rd  racinecarée( delta )
s1  ( - b - rd ) / ( 2*a )
s2  (c/a) / s1 //on peut écrire s2  ( -b + rd )/( 2*a ))
```

Exemple : Permutation de valeurs des variables

Comment permuter deux variables ? Par exemple, si la variable A vaut 5 et la variable B vaut 3, quelles instructions écrire pour que A contienne la valeur de B c'est à dire 3 et que B contienne la valeur de A c'est à dire 5?

```
Temp   A
A   B
B   Temp
```

Définition : Structure (types complexes)

Les *structures* permettent de rassembler plusieurs variables ou constantes *sous un même identificateur*; produisant un type de *variable ou constante complexe*. Ceci comporte particulièrement les *tableaux*. Un tableau est un ensemble de valeurs portant ainsi le même nom de variable et repérées par un nombre. Le nombre qui sert à repérer chaque valeur s'appelle *un indice*.

On peut créer des tableaux contenant des variables de tous types : tableaux de numériques, bien sûr, mais aussi tableaux de caractères, de booléens, de tout ce qui existe dans un langage donné comme type de variables.

La valeur d'un indice doit toujours :

- être égale au moins à 0 ou à 1 (dans certains langages, le premier élément d'un tableau porte l'indice zéro, dans d'autres langages l'indice 1)
- être un nombre entier
- être inférieure ou égale au nombre d'éléments du tableau.

1.5. Opérateurs arithmétiques

Définition : Opérateurs

Ce sont les symboles utilisés pour représenter les opérations arithmétique, logiques ou de comparaison entres les variables et les constantes. Une suite de variables (ou constantes) et d'opérations constitue une expression.

Exemple

- *(+)* addition ;
- *(-)* soustraction ;
- *(*)* multiplication ;
- *(/)* division ;
- *(>)* supérieur à ;
- *(<)* inférieur à ;
- *(=)* égal à ;
- *()* inférieur ou égal à ;

1.6. Procédures et fonctions

Définition : Procédure

Une procédure est un ensemble d'instructions référencé par un nom, et dont l'exécution est provoquée par le simple énoncé de ce nom.

Définition : Fonction

Comme pour la procédure, l'exécution d'une fonction est provoquée par la simple évocation de son nom, à la différence qu'elle se voit assigner une valeur dont le type doit être défini.

L'intérêt des procédures et fonctions est de permettre :

- Une lecture plus facile de l'algorithme principal ;
- De développer de manière indépendante des parties d'algorithmes dont l'emploi multiple au sein de l'algorithme principal est ainsi rendu plus aisé.

Pour fournir à une procédure les informations qui doivent être traitées, et pour que la procédure puisse fournir en contrepartie des résultats, on utilise des paramètres. On distinguera trois types de paramètres :

- *Les paramètres entrants (arguments)* peuvent être consultés (et éventuellement modifiés) à l'intérieur de la procédure ;
- *Les paramètres sortants (valeurs de retour)* dont la valeur sont déterminés à l'intérieur de la procédure et utilisable après l'appel à la procédure ;
- *Les paramètres mixtes* ont une valeur à l'entrée dans la procédure, valeur qui peut être modifiée à l'intérieur de celle-ci, la modification étant répercutée à l'extérieur de la procédure.

Une fonction est dite *récursive* si elle s'appelle elle-même. Par exemple pour calculer le factoriel, au lieu d'utiliser `fact(N)= N x (N-1)x (N-2) x ....2x1` on peut utiliser `fact(N)=N x fact(N-1)`.

1.7. Entrées/Sorties

Dans le but de traduire l'algorithme en un programme d'ordinateur, les entrées et les sorties ont le rôle de gérer l'interaction avec l'utilisateur, à savoir la sortie à l'écran et l'entrée au clavier (il y a aussi entrées et sorties sur imprimante, sur fichier), avec les actions correspondantes de lecture et écriture.

- Pour écrire sur l'écran, on utilise le mot ÉCRIRE (WRITE). Par exemple, l'instruction suivante : ÉCRIRE Val ; Affiche le contenu de la variable Val à l'écran.
- Pour lire sur le clavier, on utilise le mot LIRE (READ). Par exemple, l'instruction suivante : Lire A ; Donne au contenu de la variable A la valeur entée au clavier.

1.8. Expressions conditionnelles complexes

Nous utilisons *SI (IF)* pour indiquer le début de la conditionnelle simple et *FINSI (ENDIF)* pour en indiquer la fin. Le mot *ALORS (THEN)*, qui pourrait être facultatif, a pour but d'aider à la lecture. S'il y a quelque chose à faire si la condition n'est pas vérifiée, on utilise le mot *SINON (ELSEIF)*.

Exemple

```
SI condition1 ALORS action1
FINSI
```

```
SI condition1 ALORS action1
SINON action2
FINSI
```

```
SI condition1 ALORS action1
SI condition2 ALORS action2
SINON action3
FINSI
FINSI
```

1.9. Branchements multiples

S'il y a beaucoup de tests à effectuer sur une même variable ou sur plusieurs, on peut alors recourir à une structure de cas (choix). Cette structure permet, aussi en fonction de plusieurs conditions de type booléen, d'effectuer des actions différentes suivant les valeurs que peut prendre une même variable.

Exemple

```
AUX CAS OU
CAS [condition 1] Action1
CAS [condition 2] Action2
...
AUTRES CAS Action
FIN DES CAS
```

On peut aussi trouver la notation suivante :

```
SUIVANT valeur FAIRE
valeur1 : action1 ;
valeur2 : action2 ;
:
:
valeurN : actionN ;
sinon action N+1 ;
FINSUIVANT;
```

1.10. Boucles

La boucle POUR (FOR)

Une boucle POUR (FOR) (itérative déterminée) suit la syntaxe suivante :

```
POUR [variable] DE 1 A [constante ou variable]
Suite d'instructions
FINPOUR [rappel de la variable]
```

Dans cette structure, la sortie de la boucle d'itération s'effectue lorsque le nombre souhaité de répétition est atteint.

Si la valeur finale de l'indice est inférieure à sa valeur initiale le pas de variation est négatif, la structure est dite *«pour décroissante»*; dans le cas contraire, le pas est positif et la structure est dite *«pour croissante»*.

Exemple

Somme <--- 0

POUR x DE 1 A 1000

Somme <--- Somme + x

FINPOUR x

Remarque

Afin que les algorithmes donnent des résultats "intuitifs", il faut imposer trois règles à respecter dans le corps de la boucle :

1. on n'a pas le droit de modifier l'indice de la boucle
2. on n'a pas le droit de modifier la borne de la boucle
3. on ne peut pas réutiliser la dernière valeur d'indice de la boucle.

La boucle TANT QUE (WHILE)

La boucle TANT QUE (WHILE) (itérative indéterminée) a pour syntaxe:

```
[Initialisation des variables du test]
...
TANT QUE [test ou variable logique]
[Suite d'instructions]
[Modification des variables du test]
FINTANT QUE [rappel du test]
```

Dans cette structure, on commence par tester la condition ; si elle est vérifiée, le traitement est exécuté.

Exemple

```
k <--- 1
TANT QUE k <= n
[Suite d'instructions]
k <--- k+1
FINTANTQUE
```

La structure de boucle RÉPÉTER JUSQU'À

Dans la structure RÉPÉTER JUSQU'À, le traitement est exécuté une première fois puis sa répétition se poursuit jusqu'à ce que la condition soit vérifiée.

```
REPETER
action;
JUSQU'A [condition vraie]
```

Exemple : Recherche d'une valeur dans un tableau

L'organigramme de la recherche d'une valeur dans un tableau peut être écrit :

```
I <--- 1;
TANTQUE ( I  N et A(I)  K)
I <--- I+1
FINTANTQUE;
SI (I = N+1)
ALORS ECRIRE 'ABSENT'
SINON ECRIRE I
FINSI
```

Exemple : Tri par sélection d'un tableau

On veut trier un tableau de n éléments en ordre croissant :

On commence par parcourir le tableau pour trouver la plus petite valeur, on la place à l'indice 0. Ensuite, on recommence à parcourir le tableau à partir de l'indice 1, pour trouver la plus petite valeur que l'on stocke à l'indice 1. Et ainsi de suite pour l'indice 2, 3 jusqu'à n-2. Le pseudocode est :

```
Fonction trierSelection(ELEMENT * t, ENTIER n):
i <--- 0;
tant que (i < n - 1) faire
j <--- i + 1;
tant que (j < n) faire
si (t[j] < t[i]) alors
tmp <--- t[j];
t[j] <--- t[i];
t[i] <--- tmp;
fin si;
j <--- j + 1;
fin tant que;
i <--- i + 1;
fin tant que;
fin fonction;
```

La fonction se déroule de la manière suivante. Le tableau est parcouru du premier élément (indice 0) à l'avant dernier (indice n-2). On note i l'indice de l'élément visité à une itération donnée. On compare l'élément i avec chaque élément j qui suit dans le tableau, c'est-à-dire de l'indice i+1 jusqu'à l'indice n-1. Si l'élément d'indice j est plus petit que l'élément d'indice i alors on permute i et j dans le tableau.

2. Complexité des algorithmes

Il y a des algorithmes qui complètent le traitement en un temps linéaire proportionnel à la taille des données, d'autres en un temps exponentiellement proportionnel et d'autres ne se terminent jamais. Il y a aussi des problèmes qui ont plusieurs algorithmes, d'autres peuvent ne pas avoir d'algorithmes et d'autres n'ont pas d'algorithme efficace connu. Il est donc important de connaître combien de ressources (temps et stockage) sont nécessaires pour un algorithme donné.

2.1. Définitions

Définition : Complexité d'un algorithme

La complexité d'un algorithme exprime ce dont on a besoin en ressources pour aboutir au résultat. C'est le nombre d'opérations élémentaires (affectations, comparaisons, opérations arithmétiques) effectuées par l'algorithme. Elle s'exprime en fonction de la taille n des données.

On dit que la complexité de l'algorithme est O(f(n)) où f est d'habitude une combinaison de polynômes, logarithmes ou exponentielles. Elle signifie que le nombre d'opérations effectuées est borné par cf(n), où c est une constante, lorsque n tend vers l'infini. Ceci est justifié par le fait que les données des algorithmes sont de grande taille et qu'on se préoccupe surtout de la croissance de cette complexité en fonction de la taille des données.

Complément

Dans la pratique :

- les opérations qui consomment peu de temps, telles que les tests, les affectations et les incrémentations ne sont pas comptabilisées.
- les opérations ne faisant pas partie de l'algorithme, comme les E/S ne sont pas comptabilisées.

Les algorithmes usuels peuvent être classés en un certain nombre de grandes classes de complexité :

- les algorithmes sous-linéaires, dont la complexité est en général en *O(log n)*. C'est le cas de la recherche d'un élément dans un ensemble ordonné fini de taille n.
- Les algorithmes linéaires en complexité *O(n)* ou en *O(nlog n)* sont considérés comme rapides. C'est le cas par exemple de l'évaluation de la valeur d'une expression composée de n symboles ou les algorithmes optimaux de tri.
- Plus lents sont les algorithmes de complexité située entre *O(n2)* et *O(n3)*, c'est le cas de la
  multiplication des matrices et du parcours dans les graphes.
- Au delà, les algorithmes polynomiaux en *O(nk)* pour $k > 3$ sont considérés comme lents, sans parler des algorithmes exponentiels (dont la complexité est supérieure à tout polynôme en n) qui sont impraticables dès que la taille des données est supérieure à quelques dizaines d'unités.

3. Principes de conception d'algorithmes

3.1. Conception d'algorithme

Définition

La conception (et la vérification) d'un algorithme se constitue des étapes suivantes :

- Analyse : définition du problème en termes de séquences d'opérations de calcul de stockage de données, etc. ;
- Conception : définition précise (proche du langage de programmation) des données, des traitements et de leur séquencement ;
- Implémentation : traduction et réalisation de l'algorithme dans le langage cible ;
- Test : vérification du bon fonctionnement de l'algorithme.

Exemple : Somme des N premiers entiers positifs

- *Analyse*

1 Déterminer ou préciser N

2 Calculer la somme des N premiers entiers positifs

3 Renvoyer du résultat

- Premier raffinement :

1 lire N

2 boucles sur les N premiers entiers positifs ; calcul de la somme

3 afficher somme

- Deuxième raffinement de la phase 2

somme <--- 0

pour i = 1 à N

somme <--- somme + i

- *Conception*

Entiers N, i, somme

```
ECRIRE "Donner N"
LIRE N
SI N  0 alors ERREUR
POUR i = 1 à N
somme <--- somme + i
FINPOUR x
ECRIRE "La somme est : " somme
```

- *Implémentation* : (en langage C dans un fichier appelé "sommeNentiers.c")

```
#include<stdio.h>
int main(void)
{
int n, i, somme = 0;
printf("Donner n : \n");
scanf("%d", &n);
if(n  0)
{
printf(" n doit etre > 0 !\n");
return(-1); /* code erreur*/
}
for (i = 1; i  n; i++)
somme += i;
printf("Somme: %d\n", somme);
return(0); // bon fonctionnement
}
```

- *Test*

```
>sommeNentiers
Donner n :
-1
n doit être > 0 !
>sommeNentiers
Donner n :
7845
Somme : 30775935
```

3.2. Pratique du développement d'algorithmes

Principe de raffinement

Procéder par raffinements successifs, en particulier pour la phase d'analyse, en précisant les éléments manquants ou ambigus de l'énoncé. Dans le cas de l'exemple :

- S'agit-il d'une interaction avec l'utilisateur ou d'une fonction ?
- Peut-on utiliser un théorème permettant d'obtenir directement le résultat ?

À cause de l'éventuelle complexité du processus, il est souvent nécessaire de faire des *retours en arrière* pour corriger ou compléter la phase en cours ou une phase précédente. Dans tous les cas de figure, il faut veiller à bien maintenir la cohérence; prévoir les erreurs de calcul et d'utilisation. Il faut vérifier le bon fonctionnement de l'algorithme dans tous les cas. Il existe plusieurs façons de procéder :

- *Preuve mathématique* : moyen le plus sûr mais assez difficile à mettre en œuvre ;
- *Test de tous les cas* : preuve " pratique" du bon fonctionnement, mais trop long voire même impossible à mettre en œuvre ;
- *Test de tous les chemins logiques*, des conditions limites et des cas d'erreur. Dans le cas de l'exemple précédent :
    - N0;
    - plusieurs valeurs valables de N;
    - N =1 ;
    - N = INT_MAX.

Chapitre 2 : Compilateurs, liens et bibliothèques

Etapes de génération des fichiers binaires	21
Types de fichier binaires exécutables	23

Les programmes écrits en un langage de programmation sont généralement enregistrés dans des fichiers texte. Ce code, appelé code source, n'est pas directement exécutable. La génération d'un fichier binaire du programme exécutable, à partir des fichiers sources, se fait en plusieurs étapes. Ces étapes sont souvent automatisées à l'aide d'outils comme Make, Cmake ou SCons (Python) ou bien encore des outils spécifiques à l'environnement de développement intégré (IDE) utilisé.

1. Etapes de génération des fichiers binaires

1.1. Préprocesseur

Durant cette étape, le préprocesseur effectue plusieurs opérations sur les fichiers sources, dont les instructions (les directives du préprocesseur) sont au sein même de ces fichiers. Le préprocesseur produit alors des fichiers intermédiaires pour chaque fichier source, qui seront utilisés dans l'étape suivante.

Le préprocesseur effectue des remplacements de textes, des inclusions de fichiers avec la possibilité d'effectuer certaines opérations uniquement si certaines conditions sont remplies. C'est également durant cette étape que les commentaires sont supprimés.

1.2. Compilation

Un compilateur transforme un programme écrit en langage évolué en une suite d'instructions machine (en passant par la génération du code assembleur). La compilation d'un programme est réalisée en trois phases :

1. *l'analyse lexicale*, qui est la reconnaissance des mots clés du langage. Elle consiste à découper le programme en petites entités : opérateurs, mots réservés, variables, constantes numériques, alphabétiques, etc.
2. *L'analyse syntaxique*, qui analyse la structure du programme et sa conformité avec la norme d'écriture d'un code qui ressemble à celui de l'assembleur. Elle consiste à expliciter la structure du programme sous forme d'un arbre, appelé arbre de syntaxe, chaque noeud de cet arbre correspond à un opérateur et ses fils aux opérandes sur lesquels il agit.
3. *La génération de code*, est une opération qui consiste à construire la suite d'instructions du microprocesseur à partir de l'arbre de syntaxe.

Complément

Pour des programmes de plusieurs milliers de lignes, il est bon de les découper en des fichiers compilés séparément. Pour chaque fichier source, on obtient un fichier en langage d'assemblage.

Certains compilateurs (comme C) fonctionnent en deux phases, la première générant un fichier compilé dans un langage intermédiaire destiné à une machine virtuelle idéale (P-Code) portable d'une plateforme à l'autre, la seconde convertissant le langage intermédiaire en langage d'assemblage dépendant du microprocesseur utilisé sur la plateforme cible.

D'autres compilateurs permettent de ne pas générer de langage d'assemblage, mais seulement le fichier compilé en langage intermédiaire, qui sera interprété ou compilé automatiquement en code natif à l'exécution sur la machine cible (par une machine virtuelle).

1.3. Assemblage

Cette étape consiste en la génération, à partir du code assembleur, d'un fichier objet pour chaque fichier assembleur. Ce fichier objet est généralement d'extension *".o"* sous Linux, ou *".obj"* sous Windows. Cette phase est parfois regroupée avec la précédente, dans ce cas le compilateur génère directement un fichier objet binaire.

Pour les compilateurs qui génèrent du code intermédiaire, cette phase d'assemblage peut aussi être totalement supprimée : c'est la machine virtuelle qui interprétera ou compilera ce langage en code machine natif directement sur la machine hôte. Dans ce cas, la machine virtuelle qui interprète le langage intermédiaire ou le compile en code natif optimisé pour la machine hôte, peut être un composant du système d'exploitation ou une bibliothèque partagée installée sur celui-ci, et cette machine virtuelle ne sera même pas incluse dans le programme final.

1.4. Édition de liens (LINK)

L'édition de liens est la dernière étape et a pour but de réunir tous les éléments d'un programme. Les différents fichiers objets sont alors réunis, ainsi que les bibliothèques statiques, pour ne produire qu'un seul fichier exécutable. Le but de l'édition de lien est de sélectionner les éléments de code utiles présents dans un ensemble de codes compilés et de bibliothèques, et de résoudre les références mutuelles entre ces différents éléments afin de permettre à ceux-ci de se référencer directement à l'exécution du programme.

L'édition de lien peut avoir lieu aussi avec les compilateurs générant du langage intermédiaire, afin de générer un seul en langage intermédiaire où toutes les références sont résolues et un fichier facilement installable est généré. Mais même cette dernière phase d'édition de liens en langage intermédiaire est parfois aussi supprimée, celle-ci étant alors réalisée uniquement lors du chargement du programme directement par la machine virtuelle hôte. Dans ce cas, elle est le plus souvent remplacée par la création d'un paquetage.

On note un très vif regain d'intérêt pour les machines virtuelles, en raison de la grande variété des systèmes hôtes et des très rapides évolutions technologiques de celles-ci, car cela facilite très nettement le déploiement des programmes, le programmeur n'ayant plus alors besoin de générer autant de programmes exécutables que de types de systèmes cibles.

2. Types de fichier binaires exécutables

2.1. Exécutable

Sous DOS et Windows, l'extension *(.EXE, .COM)* indique que le fichier est un programme exécutable. Sous linux, il n'y a pas d'extension particulière. Le terme "*exécutable*" s'applique généralement aux programmes compilés et traduits en code machine, dans un format qui peut être cherché en mémoire et exécuté par le microprocesseur. Tout code supplémentaire utilisé dans le programme (par exemple, les fonctions de bibliothèque standard de C) sera inséré dans le fichier exécutable lui-même.

Ce code est généralement fourni sous forme de fichier objet (LIB) par l'environnement de développement. Le programme exécutable sera ainsi autosuffisant et on pourra l'installer facilement sur n'importe quelle machine, en le copiant dans un répertoire. Par contre, le problème est que cette façon rend le partage des fichiers binaires difficile. Pour utiliser aussi le code de ce programme, dans une autre application, il est nécessaire d'avoir accès au code source.

2.2. Fichiers LIB (édition de lien Statique)

Si le développement d'un gros programme devient très complexe, il y a un besoin d'employer du code développé par d'autres personnes pour des taches particulières. Par exemple, on veut développer un programme qui fonctionne sur le réseau, on peut employer une bibliothèque qui se charge des sockets. De telles bibliothèques sont disponibles sous forme de fichier bibliothèque (LIB). Un fichier LIB contient le code objet de la bibliothèque. L'édition de lien a la tâche de s'assurer que le code objet sera fusionné avec l'EXE. Cette opération s'appelle "édition de lien Statique". L'avantage des fichiers LIB est la possibilité de réutiliser le binaire directement sans la nécessité d'avoir le code source. Tout dont on a besoin est un fichier LIB et d'un peu de documentation (ou un fichier d'en-tête .h) pour vérifier la syntaxe de la fonction. Les problèmes sont que l'EXE sera d'une très grande taille parce que le code de la bibliothèque est reproduit dans chaque EXE. En plus, dans un environnement multitâche, deux copies de la même fonction de bibliothèque seraient en mémoire inutilement. Finalement, n'importe quel changement dans une fonction de la bibliothèque exige la recompilation de l'EXE, pour être inclut.

2.3. Dynamic link library (DLL)

Une DLL (fichier avec extension *.DLL)* est une bibliothèque liée dynamiquement utilisée par Windows. C'est un ensemble de modules qui contient un ensemble de fonctions ou d'autres DLLs appelées. Le code de la bibliothèque est isolé de celui des fichiers exécutables pour permettre de les stockées séparément et d'être chargé seulement une fois nécessaire, par un programme. En plus, une DLL a plusieurs avantages. D'abord, elle est facile à créer comparée aux applications et elle ne consomme aucune mémoire jusqu'à ce qu'elle soit employée. Deuxièmement, puisqu'une DLL est un fichier séparé, un programmeur peut apporter des corrections ou des améliorations seulement à ce module sans affecter l'opération du programme d'appel ou de n'importe quelle autre DLL. Enfin, un programmeur peut employer la même DLL avec d'autres programmes et partager seulement une copie d'elle, dans la mémoire.

Le problème avec les DLLs est que quand un programme a été testé avec une version particulière de la DLL et qu'une autre installation du programme met à jour la DLL avec une nouvelle version, le programme peut ne pas fonctionner bien ou du tout *(DLL hell)*.

Dans l'environnement Linux/UNIX, le même concept est mis en application dans les fichiers d'objets partagés *(.so)*.

2.4. COM (Component Object Model)

Les *COM* sont des spécifications développées par *Microsoft* pour construire des composants de logiciel qui peuvent être assemblés aux programmes ou ajouter des fonctionnalités aux programmes existants fonctionnant sur des plateformes Windows. A la différence des DLLs, au moment de compilation, il y a une dépendance très faible aux objets de bibliothèque, quand le programme utilise un objet COM. Ils sont aussi indépendants du langage. Ainsi ils doivent suivre la même disposition en mémoire exigé par les spécifications. Les composants COM peuvent être enlevés (débranchés) d'un programme au moment de l'exécution sans recompiler le programme. COM est la base de OLE (object linking and embedding), ActiveX, et DirectX.

Le problème des COM est leur complexité parce qu'ils sont difficiles à apprendre, comprendre et ont besoin d'être stockés dans la base des registres. Une référence à un objet COM sera résolue avec une relation GUID (globally unique identifier)-location, stocké dans la base des registres.

2.5. Les fichiers compilés .NET

Le code compilé pour *.NET* est stocké en tant qu'assembleur. L'assembleur stocke : IL code, Metadata d'Assembleur (manifests), Identité : nom, de version et Information de culture, Noms des fichiers dans l'assembleur, Données de type d'accès : privé ou autremen, Permissions de sécurité, Type metadata, Détails des types, des méthodes et des propriétés dans l'assembleur, et Ressources : un code assembleur est auto-descriptif. Il ne dépend pas d'objets externes comme liens de da base de registres ou des fichiers de bibliothèque pour la réutilisation.

Les avantages de .NET sont que le fichier est complètement auto-descriptif et indépendant des langages. Deux versions du même code assembleur peuvent être chargées : c'est possible parce que le manifeste stocke également l'information de version (finit DLL hell.). Enfin, l'installation des programmes est facile parce qu'un programme utilisateur peut juste avoir le code assembleur dans le même répertoire ou GAC (Global Assembly Cache) pour pouvoir utiliser les objets définis. Par contre, .NET framework doit être installé (comme Java exige JVM) et les programmes deviendront plus volumineux pour le téléchargement ou l'installation. Enfin, puisque l'assembleur stocke le metadata très détaillé, la décompilation du code assembleur est possible.

Chapitre 3 : Programmation en langage C

Les atouts du C	26
Exécution d'un programme C	26
Notion de fonction en C	29
Définitions de base	29
Exercice : Calcul de n !	33
Les structures de contrôle (Instructions conditionnelles)	33
Les fonctions	37
Passage des arguments	39
Exercice : Puissances entières et type long	40
Tableaux, pointeurs et structures	40
Exercice : Tri des tableaux	45
Exercice : Pointeurs	46
Conseils pour écrire des programmes en C	47

Le langage C est apparu au cours de l'année 1972 dans les Laboratoires Bell. Il était développé en même temps qu'UNIX par Dennis Ritchie et Ken Thompson. Sa définition rigoureuse fut en 1978. Il est appelé C parce qu'il a un prédécesseur appelé le langage B.

Le langage C est suffisamment général pour permettre de développer des applications variées type scientifique ou encore pour l'accès aux réseaux et aux bases de données. De nombreux logiciels du domaine des PC tels que Microsoft Word ou Excel, sont eux aussi écrits en langage C ou de son successeur *orienté objet C++*.

1. Les atouts du C

- L'un des langages les plus utilisés; il est devenu un standard. Normalisation ANSI (American National Standard Institut) en 1988, puis ISO (International Standardisation Organization).
- Instructions et structures de haut niveau (presque un langage parlé) : Le langage C n'a pas l'ambition d'obliger les programmeurs à respecter un quelconque style de programmation, il est en effet peu contraignant au niveau de la compilation et il offre peu de restrictions sémantiques. En particulier, la mise en page est libre, ce qui permet d'écrire des programmes dont la mise en page reflète la structure.
- un code rapide grâce à :
  1. un compilateur performant. Les concepteurs du C l'ont créé pour leur propre utilisation, ils ont donc préféré favoriser l'expressivité du langage (d'où la richesse des expressions) que d'éviter les erreurs de programmation en multipliant les tests à la compilation.
  2. Des instructions proches du langage machine. Le langage C a été conçu et réalisé pour écrire un système d'exploitation et les logiciels de base de ce système (interpréteur de commande, compilateur, ...). Pour ce faire, il doit être capable de faire les mêmes choses que l'assembleur.
- Le langage C est populaire chez les électroniciens (on peut même dire que c'est leurs langage) :
  1. Il est utilisé pour développer avec les programmes pour les micro microprocesseurs (autres que ceux du PC) et micro contrôleurs : mplab du pic de microchip, Avr ......
  2. Il est utilisé pour les DSP. Voir par exemple code composer studio de Texas instrument.
  3. Il est le langage naturel de Unix et Linux (voir le code source de Linux).
  4. Il y a même des tentatives comme (SystemC) qui proposent de traduire directement le langage en portes logiques ou de généré du HDL pour programmer un FPGA.
  5. Il a une interface avec les langages script comme python et Tcl.
  6. Il a une interface avec Matlab.

2. Exécution d'un programme C

Programme C (Ecrire le code source)

↓

Compilateur (Créer un code exécutable)

↓

Programme objet

↓

Editeur de liens

↓

Données → Programme exécutable → Résultat

Exemple : Le programme (Hello world) en C

Selon la tradition, on va commencer la découverte du C par l'inévitable programme "Hello world". Ce programme ne fait rien d'autre qu'imprimer le message "Hello world" sur l'écran :

```
# include <stdio.h>
main ( )
// Notre premier programme en C
{
Printf ("Hello, world\n");
return 0;
}
```

Dans la suite, nous allons discuter les détails de cette implémentation :

- *Le Préprocesseur*

Un Préprocesseur traite le code avant la compilation et effectue des substitutions de macros, une compilation conditionnelle, et l'inclusion de fichiers par leurs noms. En plus, le Préprocesseur permet :

1. Simple substitution de texte.
2. Augmente la lisibilité du code source.
3. Augmente la vitesse d'exécution du programme.
4. Facilite le débogage.

Les principales commandes du préprocesseur sont :

- # *define* identificateur chaîne :

Le Préprocesseur remplace identificateur par chaîne dans le code source.

- # *include* <nom-de-fichier> :

On inclut le fichier texte "nom-de-fichier". Le texte du fichier est alors recopié à l'endroit de la commande. Le Préprocesseur recherche "nom-de-ficher" dans les répertoires standards et l'insère dans le code source.

- # *include* "nom-de-fichier" :

La même chose que précédemment sauf que le Préprocesseur recherche "nom-de-ficher" dans le répertoire courant, et l'insère dans le code source.

- *La compilation conditionnelle*

Il y a plusieurs instructions de compilation conditionnelle propres au Préprocesseur dont la syntaxe est la suivante :

- # *if expression-constante* ou # *ifdef indentificateur* ou # *ifndef identificateur* :

ifdef identificateur équivaut à # if defined (identificateur);

de même

ifndef équivaut à if !defined.

ifdef (identificateur).

- texte

- # *elif* expression-constante

- texte

- # *else*

- texte

- # *endif*

La compilation conditionnelle permet d'éviter une redondance du code lors des multiples appels *#include "module.h"*. Par ailleurs, l'inclusion de *fichier.h* dans *fichier.c* (qui n'est à priori pas nécessaire) permet au Préprocesseur de vérifier la cohérence entre les déclarations et les définitions.

Les fichiers h standards les plus utilisés sont :

La bibliothèque standard *stdio.h*

La bibliothèque standard *stdlib.h*

La bibliothèque standard *math.h*

3. Notion de fonction en C

Les programmes en C sont composés essentiellement de fonctions et de variables. Il est donc indispensable de se familiariser avec les caractéristiques fondamentales de ces éléments.

En C, le programme principal et les sous-programmes sont définis comme fonctions. Il n'existe pas de structures spéciales pour le programme principal ni les procédures (comme en Pascal). Le programme principal étant aussi une 'fonction'.

La fonction principale des programmes C est appelée *main*. Elle se trouve obligatoirement dans tous les programmes. L'exécution d'un programme entraîne automatiquement l'appel de main.

Par définition, toute fonctions en C (y compris main) doit avoir des arguments d'*entrée* et fournit un résultat (de *sortie*) dont le *type* doit être défini. Si aucun type n'est défini explicitement, C suppose par défaut que le type du résultat est *int* (integer). Le retour du résultat se fait en général à la fin de la fonction par l'instruction *return*. Le type d'une fonction qui ne fournit pas de résultat est déclaré comme *void* (vide).

Enfin, si nous utilisons des fonctions prédéfinies (par exemple : *printf*), il faut faire précéder la définition de main par les instructions *#include* correspondantes.

4. Définitions de base

Définition : Déclarations

Le rôle d'une déclaration est de préciser le nom (identifiant) et le type d'une variable (ou une fonction, etc).

Définition : Instructions

La liste des opérations à effectuer et qui définit l'action qui doit être exécutée. On appelle instruction, une instruction simple se terminant par un point-virgule.

Par exemple : printf("Hello, world\n");

Définition : Bloc d'instructions

Délimité par des accolades {...} regroupe plusieurs instructions. En C, *toute instruction simple est terminée par un point-virgule*; même si elle se trouve en dernière position dans un bloc d'instructions.

Définition : Les variables

Une *variable* est désignée par un nom qui est une chaîne de caractères appelée identificateur (commence par une lettre). Une variable a une *adresse*, un *type* et une *valeur*.

Définition : Les types

À toutes données doit correspondre un type, qui autorise la réservation d'une zone mémoire de taille prédéfinie. Chaque variable a un type. Elle est déclarée avec son type par :

type_ de _variable nom_ de_ variable;

Exemple : `int n ;`

Complément

Les types de base en C sont :

- char caractères
- int entiers
- float réel en virgule flottante
- double float double précision

Les précisions des types du C sont les suivantes :

- short int 2 octets -32 768 ... 32 767
- unsigned short int 2 octets 0 ... 65 535
- long int 4 octets -2 147 483 647 ... 2 147 483 648
- unsigned long int 4 octets 0 ... 4 294 967 295
- float 4 octets simple précision, 1.18 E +/- 38
- double 8 octets double précision, 9.46 E -308 ... 1.8 E 308
- long double 10 octets
- char 1 octet -128 ... 127
- unsigned char 1 octet 0 ... 255

Pour le cas des variables *booliennes*, en C, une expression conditionnelle est fausse, quand elle prend la valeur (entière) 0, et vraie dans tous les autres cas.

Il existe aussi des types plus compliqués comme les *tableaux* et les *structures*.

Définition : Initialisation des variables

L'initialisation consiste à spécifier la valeur d'une variable au moment de la création de sa zone mémoire. En l'absence d'initialisation explicite :

- les variables externes et statiques sont toujours initialisées à zéro.
- les variables automatiques et les variables-registres ont des valeurs initiales indéfinies.

On peut initialiser les variables scalaires simplement en écrivant :

- int x = 1 ; char c = 'a' ; long j = 1000L * 100L ;

Pour les variables externes et statiques, l'initialisateur doit être une expression constante;

Définition : Les expressions

Une *expression* exprime un calcul, et possède une valeur typée. Les éléments de base de construction d'une expression sont d'une part les données (les constantes littérales, les variables) et d'autre part les traitements (les fonctions, les opérateurs). On construit une expression en utilisant des variables et des constantes puis des opérateurs choisis parmi :

- Les opérateurs arithmétiques : + , - , * , / , ...
- Les comparateurs : > , >= , == , ! = , ...
- Les opérateurs logiques : && (AND), | | (OR) ! (NOT)

Elle peut être (le résultat de l'expression) de type char, int ou float...

Définition : Les opérateurs

Les *opérateurs* sont des traitements prédéfinis, propres au langage, à la différence des fonctions qui sont des traitements définis par le programmeur.

On distingue, selon la place de l'opérateur par rapport aux opérandes, les opérateurs :

- Préfixes : op d1 d2
- infixes : d1 op d2
- postfixes : d1 d2 op.

Pour évaluer ces expressions sans ambiguïtés, on définit un ordre de priorité entre les opérateurs. Les parenthèses possèdent l'ordre de priorité maximum. L'associativité définit le sens du traitement gauche-doite, ou droite-gauche, des expressions constituées d'opérateurs de même priorité. Le tableau suivant résume les ordres de priorités des différents opérateurs arithmétiques et logiques du langage C.

1	fonction/sélection	( ) [ ] . ->	GD
2	unaire	* & - ! ~++ -- sizeof	DG
3	multiplicatif	* / %	GD
4	additif	+ -	GD
5	décalage	<< >>	GD
6	relationnel	< > <= >=	GD
7	inégalité/égalité	== !=	GD
8	ET logique	&	GD
9	XOR logique	^	GD
10	OU logique	|	GD
11	ET relationnel	&&	GD
12	OU relationnel	||	GD
13	conditionnel	?	GD
14	affectation	= <op>=	DG
15	virgule	,	GD

Les opérateurs en C

Complément

En plus des opérateurs arithmétiques usuels le C définit aussi :

- l'opérateur *sizeof* (type ou variable) qui retourne *la taille en octet du type ou de la variable* passée en argument. Cet opérateur est très utilisé pour l'allocation dynamique de mémoire.

- les opérateurs d'*incrémentation et de décrémentation* servent à incrémenter et décrémenter les variables (et non toutes expressions). L'opérateur d'incrémentation *++* ajoute 1 à son opérande, alors que l'opérateur de décrémentation -- lui retranche 1. L'aspect inhabituel de ces opérateurs est qu'on peut les mettre sous forme préfixée (++n) ou postfixée (n++). Dans les deux cas, n est incrémenté. Mais l'expression ++n incrémente n avant de prendre sa valeur, alors que n++ l'incrémente après avoir pris sa valeur.

- l'expression conditionnelle, qui utilise l'opérateur conditionnel ? s'écrit :
  expression1 ? expression2 : expression3
  Cet opérateur est ternaire. On commence par évaluer l'expression expression1. Si elle est vrai (c'est-à-dire 0), alors on évalue expression2 et c'est elle qui donne sa valeur à l'expression conditionnelle; sinon on évalue expression3 et c'est elle qui donne sa valeur.
  Ainsi pour affecter le maximum de a et b à max, il suffit d'écrire :
  `max = ( a > b ) ? a : b ;`
  Il n'est pas obligatoire de mettre des parenthèses autour de la première expression, car la priorité de ? : est très faible, immédiatement supérieur à celle de l'affectation.
- Les opérateurs d'affectation : On remplace souvent en raccourci, et pour plus de clarté, `i = i + 2 ;` par l'expression équivalente utilisant l'opérateur d'affectation *+=* , c.a.d. : `i + = 2 ;`
- Le C associe à la plupart des opérateurs binaires op un opérateur d'affectation op=, où op fait partie de + - * / % << >> & ^ |
- Les opérateurs de traitement des bits :
  le C fournit six opérateurs qui réalisent des manipulations au niveau des bits des entiers, (char, short, int et long).
  ~ complément à un (opérateur unaire)
  >> décalage à droite
  << décalage à gauche
  & ET bit à bit
  ^ OU exclusif bit à bit
  | OU inclusif bit à bit
  Par exemple, on se sert souvent du ET bit à bit & pour masquer certains bits ; Les opérateurs << et >> décale leur opérande de gauche du nombres de bits indiqué par leur opérande de droite, qui doit être positif. Ainsi x << 2 décale la valeur de x de deux bits vers la gauche, en remplissant à droite par des 0, ce qui revient à faire une multiplication par 4.

Définition : Les conversions de type

Lorsqu'un opérateur a des opérandes de types différents, ils sont convertis en un type commun d'après quelques règles. En général, les seules conversions automatiques sont celles qui convertissent un opérande "étroit" en un opérande plus "large", sans qu'il y ait perte d'information.

Enfin, dans toute expression, on peut forcer explicitement des conversions grâce à un opérateur unaire appelé cast. Dans la construction : (int) expression, l'expression est convertie dans le type précisé.

Exemple : Discussion de l'exemple 'Hello World'

- La fonction main ne reçoit pas de données, donc la liste des paramètres est vide.
- La fonction main fournit un code d'erreur numérique à l'environnement, donc le type du résultat est int et n'a pas besoin d'être déclaré explicitement.
- Le programme ne contient pas de variables, donc le bloc de déclarations est vide.
- La fonction main contient deux instructions :

1. l'appel de la fonction printf avec l'argument "hello, world\n"; son effet est d'afficher la chaîne de caractères "hello world\n" sur l'écran.
2. la commande return avec l'argument 0; son effet est de retourner la valeur 0 comme code d'erreur à l'environnement.

- L'argument de la fonction printf est une chaîne de caractères indiquée entre les guillemets. Une telle suite de caractères est appelée chaîne de caractères constante (constant string).
- La suite de symboles '\n' à la fin de la chaîne de caractères "hello, world\n" est la notation C pour 'passage à la ligne' (new line). En C, il existe plusieurs couples de symboles qui contrôlent l'affichage ou l'impression de texte. Ces séquences d'échappement sont toujours précédées par le caractère d'échappement '\'.
  printf et la bibliothèque <stdio> : La fonction printf fait partie de la bibliothèque de fonctions standard <stdio> qui gère les entrées et les sorties de données. La première ligne du programme :
  # include <stdio.h>
  instruit le compilateur d'inclure le fichier en-tête '*stdoi.h*' dans le texte du programme. Le fichier 'stdio.h' contient les informations nécessaires pour pouvoir utiliser les fonctions de la bibliothèque standard *<stdio>*.

5. Exercice : Calcul de n !

Calculer *n!* pour une valeur de n *fournie par l'utilisateur et comprise entre 1 et 7*. Si la valeur entière fournie par l'utilisateur n'est pas dans cet intervalle, il lui sera demandé une nouvelle valeur.

Solution

```
#include <stdio.h>
void main ()
{
char n;
int i ,
fact = 1;
printf("entrez une valeur entière positive inférieure ou égale à 7 :
");
scanf("%d",&n);
while((n<=0)||(n>7))
{
if (n<=0) printf("j'ai demandé une valeur positive, redonnez la valeur
: ");
else printf("j'ai demandé une valeur inférieure à 8, redonnez la
valeur : ");
scanf("%d",&n);
}
for(i = 2; i <= n; i++) fact*=i;
printf("la valeur de %d! est %d\n",n,fact);
}
```

6. Les structures de contrôle (Instructions conditionnelles)

Les instructions de contrôle servent à contrôler le déroulement de l'enchaînement des instructions à l'intérieur d'un programme, ces instructions peuvent être des instructions conditionnelles ou itératives.

6.1. Les tests if

Instructions conditionnelles

Comme on a vu en algorithmique, les instructions conditionnelles permettent de réaliser des tests et d'exécuter, suivant le résultat de ces tests, des parties de code différentes.

Les tests if

forme incomplète

- `if (test) instruction;` Si test est vrai, alors instruction est exécuté, sinon instruction est ignoré.

ou aussi forme complète:

- `if (test) instruction1 ; else instruction2 ;` Si test est vrai, alors instruction1 est exécuté, sinon instruction2 est exécuté.

Le test est une expression de type entier comme a<b construite en général avec les comparateurs <, = =, ! =, ... et les opérateurs logiques &&, | |, ...

On peut aussi effectuer plusieurs tests comme :

```
if (test) instruction1

else if (test2) instruction2

else if (test3) instruction3

...

else if (testN) instructionN

else instructionN+1
```

Une et une seule des N instructions sera exécutée. L'instruction qui est exécutée correspond à la première expression vraie. Dans le cas, où il n'y a aucune expression vraie, et si l'instruction else est présente (car elle est facultative) c'est instructionN+1 qui est exécutée.

Exemple : Calcul du minimum de deux valeurs a et b :

```
if (a < b)
{
min = a ;
}
else
{
min = b ;
}
```

6.2. Instructions itératives (Les boucles)

Les instructions itératives sont commandées par trois types de boucles : le while, le for et le do while.

6.2.1. Boucle For

```
for (initialisation ; test ; incrémentation)
{
liste d'instructions
}
Exemple :
for (i = 0 ; i<n ; i++)
{
x = x+1 ;
}
```

Au début de la boucle, on exécute les instructions d'initialisation de la boucle une seule fois. Si test est vrai, alors on exécute la liste d'instructions, puis les expressions d'incrémentation ou décrémentation. Ces deux dernières opérations continuent jusqu'à ce que test deviennent faux.

Exemple : Somme des n premiers entiers

```
s=0;
for (i =1 ; i<= n ; i++) s = s+i ;
```

Après l'exécution de cette instruction, s vaut :

1+2+ ... + n = n (n+1)/2

Exemple : Somme des n premiers carrés :

```
s=0;
for (i = 1 ; i<= n ; i++) s = s+i*i ;
```

Après l'exécution de cette instruction, s vaut :

1+4+9+ ... + n2 (= ?)

6.2.2. Boucle While

La syntaxe est :

`while (test)`

`instruction`

instruction est exécuté tant que test est vraie

Exemple : Calcul de la première puissance de 2 excédant un nombre N

```
p=1;
while (p < N) do
{
p = 2*p ;
}
```

Résultats :

Pour N = 100 on a p = ?

Pour N = 200 on a p = ?

6.2.3. Boucle do

La syntaxe est :

```
do
{
liste d'instructions
}
while (test) ;
```

On exécute pour commencer la liste d'instructions, puis si test est vrai on exécute la liste d'instructions à nouveau, sinon on quitte la structure, et on passe à l'instruction suivante.

Exemple : Calcul de la somme des n premiers entiers

```
i=0;
do
{
i=i+1;
}
while (i <= n) ;
```

6.2.4. L'instruction switch

La syntaxe est :

```
switch (test)
{
case expression-constante1 :
instructions
break;
case expression-constante2 :
instructions
break;
...
case expression-constanteN :
instructions
break;
default :
instructions
break ;
}
```

test est est évalué et permet de faire le branchement sur l'une des constantes (entiers ou caractères, énumérations) de case correspondant à la valeur de test. Il faut noter que expression-constante ne peut en aucun cas être une variable. Les constantes à l'intérieur d'un switch doivent être toutes de valeurs distinctes.

L'étiquette default est facultative et l'instruction correspondante à celle-ci est exécutée lorsque aucun branchement n'a été effectué.

Les breaks sont à priori facultatifs, mais il est préférable de toujours les garder, pour forcer la sortie du switch, et prévenir un fonctionnement hasardeux.

6.2.5. Ruptures de séquence

Dans le cas où une boucle commande l'exécution d'un bloc d'instructions, il peut être intéressant de vouloir sortir de cette boucle alors que la condition de passage est encore valide. Ce type d'opération est appelé une rupture de séquence. Les ruptures de séquence sont utilisées lors des conditions multiples peuvent conditionner l'exécution d'un ensemble d'instructions. Dans le cas de switch, break est utilisée pour faire une rupture de séquence. Elle exécute une sortie d'un bloc d'instructions dépendant de l'une des instructions suivantes do, for, while, ou switch. Il y a aussi :

- *continue* : Instruction forçant le passage à l'itération suivante de la boucle la plus proche.
- *goto label*; ... *label : instruction* : l'instruction goto permet de se brancher inconditionnellement à instruction. A éviter absolument.
- *return [expression]* : Exécute une sortie d'une fonction en rendant le contrôle à la fonction appelante, tout en retournant une valeur si la fonction appelée l'autorise.
- *exit [expression] :* Permet de quitter le programme avec une valeur, avec "*flush*" et fermeture des fichiers, libération de la mémoire...

7. Les fonctions

Définition

Les fonctions (procédures) sont des parties de code source qui permettent de réaliser le même type de traitement plusieurs fois et/ou sur des variables différentes.

Complément

Une fonction en langage C peut : modifier des données globales. Ces données sont dans une zone de mémoire qui peut être modifiée par le reste du programme. Une fonction peut dans ces conditions réaliser plusieurs fois le même traitement sur un ensemble de variables définies à la compilation; communiquer avec le reste du programme par une interface. Cette interface est spécifiée à la compilation. L'appel de la fonction correspond à un échange de données à travers cette interface, au traitement de ces données (dans le corps de fonction), et à un retour de résultat via cette interface.

Rappel

- Un programme C est constitué de fonctions. Elles ont toutes la même structure.
- La fonction principale s'appelle main.
- Toutes les fonctions sont au "même niveau". On ne peut pas déclarer une fonction à l'intérieur d'une autre.

Fondamental

La syntaxe d'une fonction est comme suit :

```
Type retourné Nom (Type des paramètres)
{
déclarations
instructions
l'instruction return (expression);
}
```

Chaque fonction comprend :

1. Une déclaration qui précise :
    - Le type de ses paramètres.
    - Le type de la valeur de retour.
    - comment on la calcule.
2. Un ou plusieurs appels : c'est l'utilisation de la fonction.
3. Un ou plusieurs paramètres : ce sont les arguments de la fonction.
4. Un type et une valeur.

Généralement, la définition de la fonction (prototype) joue aussi le rôle de la déclaration;

mais on peut très bien séparer la déclaration. Par exemple, si l'on souhaite réaliser un

module exportable, ou encore si l'on souhaite utiliser une fonction défini plus bas.

La syntaxe du prototype est seulement :

```
Type retourné Nom (Type des paramètres);
```

Si l'on ne précise pas le type de retour, c'est le type int qui est pris par défaut. Les fonctions communiquent par le mécanisme des arguments et des valeurs de retour, et encore via le mécanisme des variables externes.

L'instruction return est le mécanisme par lequel une fonction appelée retourne une valeur à la fonction appelante.

```
return expression ;
```

```
Fonction sans valeur de retour :
Void fonct ( ... . )
{
} ; fonct ne retourne aucune valeur
Fonction sans paramètre :
float Pi ( void )
{
return 3.1416 ; }
Exemple de la fonction factorielle :
int fact ( int n )
{
int i, f ;
f=1;
for ( i = 2 ; i<= n ; i++ )
f=f*i;
return f ;
}
```

Exemple : Suite de Fibonacci

f0 = 0,

f1 = 1,

f n = f n-1 + f n-2, f i = nombre de Fibonacci

```
int fibonacci (int n)
{
int f , f 0, f 1, i ;
f0=0;
f1=1;
i=2;
while ( i <= n)
{
f = f 0+ f 1 ;
f0=f1;
f1=f;
i = i+ 1
}
return ( f )
}
```

Attention

Que se passe t'il si le paramètre d'appel n = 1 ou 0 ? Comment y remédier ?

8. Passage des arguments

Dans le cas général (et simple), une fonction est déclarée avec des paramètres formels; elle est appelée avec des paramètres réels de même nombre. Tous les paramètres sont transmis par valeur. La fonction ne travaille pas directement sur le paramètre réel mais sur une variable locale à la fonction, déclarée implicitement et initialisée à la valeur du paramètre réel. La fonction ne peut pas modifier la valeur de ses paramètres réels. Pour modifier la valeur de ses paramètres réels, il faut passer comme paramètres les adresses (pointeurs sur) de ces derniers.

Exemple : Fonction echanger(a,b) : Exemple faux

La fonction ci-dessous ne réalise pas l'échange des entiers a et b.

```
void echanger ( int x, int y)
{
int temp ;
temp = x ;
x=y;
y = temp ;
}
```

Exemple : Fonction echanger(a,b) : Exemple correcte

Le moyen d'obtenir le résultat voulu est de faire en sorte que le programme appelant passe en arguments des pointeurs sur les valeurs à modifier (passage par adresse) :

`echanger (&a , &b).`

On propose la solution :

```
void echanger ( int * px, int * py)
{
int temp ;
temp = * px ;
* px = * py ;
* py = temp ;
}
```

Les arguments de type pointeur permettent à une fonction d'accéder aux objets de la fonction appelante, et de les modifier. Le passage par valeur d'un tableau n'existe pas en langage C : seule l'adresse de début de zone mémoire où sont stockées les valeurs est passé à la fonction appelée.

9. Exercice : Puissances entières et type long

Écrire un programme permettant d'imprimer les 12 premières puissances des entiers -2 et 3.

Attention : le résultat pouvant excéder 35000 il faut utiliser un entier long pour le stocker.

Solution

```
#include<stdio.h>
long puis(int,int);
void main(void)
{
int i;
for(i=0;i<12;++i)
printf("%d %ld %ld\n",i,puis(-2,i),puis(3,i));
}
long puis(int base,int exp)
{
int i;
long p=1;
for(i=1;i<=exp;++i) p=p*base;
return p;
}
```

10. Tableaux, pointeurs et structures

10.1. Tableaux

Définition

Un tableau est un type de données complexe le plus simple à construire : Suite ordonnée d'éléments. Il faut que tous les éléments soient du même type. Ce qui permet d'accéder facilement à chaque élément.

Complément

Le tableau est caractérisé par :

- Un nom
- Un type
- Une dimension
- Une classe d'allocation

Déclaration :

```
float a [5] ;
```

Tous les indices commencent à 0. Le nom de l'élément d'indice i est : *a [i]*

Le nom du tableau est *a [ ]*.

On utilise souvent une constante pour désigner la dimension d'un tableau. Cette dimension doit être connue à la compilation : allocation statique de mémoire.

La déclaration d'un tableau à une dimension réserve un espace de mémoire contiguë dans lequel éléments du tableau peuvent être rangés.

Le nom du tableau seul est une constante dont la valeur est l'adresse du début du tableau.

Les éléments sont accessibles par : le nom du tableau, un crochet ouvrant, l'indice de l'élément et un crochet fermant.

Attention

Les indices non valides ne sont pas détectés.

Exemple

#define N 50 // dimension des tableaux

..

float R [N] ; déclare un tableau de 50 nombres réels de *R[0]* à *R [49]*.

Exemple

On peut initialiser un tableau en le déclarant.

int a [5] = {1, 1, 1, 1, 1} ; // initialise tous les éléments à 1.

Exemple

Recherche du minimum du tableau a [0] , a [1] , ..., a [N-1] de nombres entiers. (On garde dans la variable m la valeur provisoire du minimum et on balaye le tableau de 0 à N – 1)

```
int Min (int a [ ] )
{
int i, m ;
m = a [ 0] ;
for (i = 2 ; i < N ; i ++)
if ( a [ i ] < m ) m = a [ i ] ;
return m ;
}
```

Définition : Tableaux à n dimensions

Les tableaux à deux dimensions sont des tableaux de tableaux. Les indices de droite sont les plus internes. Les tableaux à n dimensions sont des tableaux de tableaux à n-1 dimensions.

Exemple

int tab[8][5];

déclare un tableau de 8 lignes et 5 colonnes.

10.2. Structures

Définition

Une structure est un type complexe (‘tableau') rassemblant plusieurs variables, mais chaque case peut contenir un type différent. La syntaxe est la suivante :

```
struct [ nom ] {
< liste de déclarations >
};
```

Exemple

```
struct point {
int x;
int y;
};
déclare une structure contenant les données d'un point de
l'espace.
struct point p1,p2;
déclare deux points.
```

Les structures sont un exemple de définition de nouveaux types. Sa taille est au moins égale à la somme des tailles de ses membres du fait d'éventuels alignements mémoires.

L'opérateur sizeof permet d'en connaître la taille.

Exemple

```
struct {
char
c;
unsigned int i;
float tab[10];
char *p;
} a, b;
```

10.3. Les énumérations de constantes

Définition

Les énumérations sont des types définissant un ensemble de constantes qui portent un nom que l'on appelle énumérateur. Elles servent à rajouter du sens à de simples numéros, à définir des variables qui ne peuvent prendre leur valeur que dans un ensemble fini de valeurs possibles identifiées par un nom symbolique. La syntaxe est la suivante :

```
enum [nom] {
énumérateur1,
énumérateur2,
...
énumérateur n
};
```

Les constantes figurant dans les énumérations ont une valeur entière affectée de façon automatique par le compilateur en partant de 0 par défaut et avec une progression de 1.

Les valeurs initiales peuvent être forcées lors de la définition.

Exemple

```
enum couleurs {noir, bleu, vert, rouge, blanc, jaune};
enum couleurs {
noir = -1,
bleu,
vert,
rouge = 5,
blanc,
jaune
};
```

Complément

Il exite deux autres types complexes qui sont les :

- champ de bits : un ensemble de bits contigus à l'intérieur d'un même mot.
- Union : permet de définir des données de type différent ayant la même adresse mémoire.

10.4. Pointeurs

Définition

Un pointeur est une variable ou une constante dont la valeur est une adresse. Il est reconnu syntaxiquement par le * lors de sa déclaration. L'adresse d'une variable (ou d'une fonction) est indissociable de son type. On pourra donc définir, par exemple, des pointeurs de caractères, des pointeurs d'entiers voire des pointeurs d'objets plus complexes.

Complément

L'opération fondamentale effectuée sur les pointeurs est l'*indirection*, c'est-à-dire l'évaluation de l'objet pointé. Le résultat de cette indirection dépend du type de l'objet pointé.

Par exemple, si *p_car* et *p_reel* sont respectivement un pointeur de caractères et un pointeur de réel simple précision référençant la même adresse *add*, une indirection effectuée sur p_car désignera le caractère situé à l'adresse add, tandis qu'une indirection effectuée sur p_reel désignera le réel simple précision située à la même adresse. Donc, bien qu'ayant le même contenu (l'adresse add), ces deux pointeurs ne sont pas identiques !

Le symbole associé à la construction des pointeurs est celui d'indirection *(*)*.

La syntaxe est : `type * nom_pointeur`

Remarque

La déclaration d'un pointeur n'implique pas la déclaration implicite d'une variable associée et l'affectation de l'adresse de la variable au pointeur. Il faut donc déclarer une variable du type correspondant et initialiser le pointeur avec l'adresse de cette variable.

Fondamental

Les opérations possibles sur un pointeur sont les suivantes :

- affectation d'une adresse au pointeur;
- utilisation du pointeur pour accéder à l'objet dont il contient l'adresse;
- addition d'un entier (n) à un pointeur; la nouvelle adresse est celle du neme objet à partir de l'adresse initiale;
- soustraction de deux pointeurs du même type. Ce qui permet de calculer le nombre de variables entre les adresses contenues dans les pointeurs.

Enfin le pointeur peut être combiner avec les tableaux [] et les fonctions ( ) pour définir des objets encore plus complexes.

Exemple

- *int *ptint;* //pointeur sur un entier
- *char *ptchar;* //pointeur sur un caractère.
- *char *chaines[100];* //un tableau de 100 pointeurs de caractère,
- *char **argv;* //un pointeur de pointeur de caractère.

L'utilisation de parenthèses permet de modifier la priorité et donc l'ordre d'évaluation.

Exemple

- `int (*tab)[10];`
- `char (*f)();`
- `char *(*g)();`
- `float *(*tabf[20])();`

Cet exemple permet de définir respectivement :

- Un pointeur de tableau de 10 entiers,
- Un pointeur de fonction retournant un caractère,
- Un pointeur de fonction retournant un pointeur de caractère,
- Un tableau de 20 pointeurs de fonction retournant un pointeur de réel simple précision.

Les tableaux peuvent être manipulés à travers l'utilisation de pointeurs. Cette utilisation des pointeurs pour accéder aux contenus des tableaux est l'une des techniques les plus utilisées par les programmeurs expérimentés.

Le nom d'un tableau, correspond à l'adresse du premier élément tableau, de plus les éléments d'un tableau sont du même type. Ces éléments ont donc tous la même taille, et ils ont tous une adresse qui correspond au même type d'objet.

Considérant l'exemple suivant : *int tab[10];*

Les adresses de tous les éléments du tableaux sont comprises entre *&tab[0]* et *&tab[9]*. Il est ainsi possible d'additionner ou de soustraire un entier (n) à une adresse. Cette opération calcule une nouvelle adresse. L'adresse de départ est &tab[0] ou tab qui peut être considérée comme un pointeur. tab + n est l'adresse du nième entier à partir du début du tableau.

Remarque

Le programmeur doit être conscient des risques de dépassement des bornes du tableau.

11. Exercice : Tri des tableaux

Écrire un programme permettant de saisir des entiers (au plus 8), et de les voir affichés triés par ordre croissant. L'algorithme de tri utilisé est le tri par insertion.

Les entiers seront stockés dans un tableau prévu pour 8 entiers. Ils seront ensuite triés dans ce même tableau, qui servira à l'affichage. Ce tableau sera déclaré comme variable globale du programme. Ce tableau est rempli grâce à une fonction nommée lire().

On rappel le principe du tri insertion. On considère que la gauche du tableau est déjà triée. On considère alors l'élément suivant la partie triée et on le fait "descendre" à sa place en le comparant à l'élément qui le précède. On commence par "sauver" l'élément à mettre en place dans une variable "clé". On pousse ensuite d'une position vers la droite les éléments qui le précèdent et qui sont plus grands que lui. On le range dans le tableau lorsqu'on a trouvé sa place.

Solution

```
#include <stdio.h>
enum {MAXDONNEES = 8};
int tab[MAXDONNEES]; //le tableau tab est global
int lire(void); //prototype de la fonction lire
void trier(int);
void afficher(int);
void main(void)
{
int NbrDonnees;
NbrDonnees = lire();
trier(NbrDonnees);
afficher(NbrDonnees);
}
int lire(void)
{
int donnee, nbr = 0;
printf("Entrez les données entières positives à trier.\n"
"Terminez la saisie en tapant -1.\n"
"Vous avez droit à au plus %d valeurs\n",MAXDONNEES);
scanf("%d",&donnee);
while((nbr <= MAXDONNEES) && (donnee != -1))
{
if (nbr == MAXDONNEES)
{
printf("\nvous avez donné plus de %d entiers, seuls les %d "
"premiers seront triés\n", MAXDONNEES, MAXDONNEES);
break;
}
else
{
```

```
tab[nbr] = donnee;
nbr++;
scanf("%d", &donnee);
}
}
return nbr;
}
void trier(int nombre)
{
int i, j, cle;
for (i = 1; i < nombre; i++)
{
cle = tab[i];
for (j = i-1; (j >= 0) && (cle<tab[j]); j--) tab[j+1] = tab[j];
tab[j+1] = cle;
}
}
void afficher(int nombre)
{
int i;
printf("\nvoici le tableau trié\n");
for(i = 0; i < nombre; i++) printf("%5d",tab[i]);
printf("\n");
}
```

12. Exercice : Pointeurs

Séparer les nombres pairs et les impairs en les mettant dans deux tableaux, puis trier chacun des deux tableaux. Les tableaux ont une taille arbitraire de 100. La lecture des entiers se termine lors de la saisie de 0. La séparation entre nombres pairs et impairs est faite dans une fonction qui reçoit en arguments les adresses des trois tableaux (adresse de l'élément de rang 0 du tableau), cette fonction termine les tableaux pairs et impairs par des zéros.

Le tri de chaque tableau est réalisé par une fonction tri(int *) qui utilise un algorithme de tri simple.

Solution

```
#include <stdio.h>
int tab[100], imp[100], pair[100];
int count;
// fonction de saisie du tableau a manipuler
int saisie (int ti[ ])
{
int *ptab = ti;
printf (" Entrer des entiers \n");
printf (" Terminer la liste avec 0\n\n");
do
scanf ("%d", ptab);
while (*ptab++ != 0);
return ptab - ti;
}
// separation des pairs et des impairs en deux tableaux
int separ (int t[], int tp[], int ti[])
{
register int *pttab = t, *ptimp = ti, *ptpair = tp;
while (*pttab)
if (*pttab & 0x01) *ptimp++ = *pttab++;
else *ptpair++ = *pttab++;
*ptimp = *ptpair = 0; /* chien de garde de fin de tableau */
return pttab - t; /* nombre d entiers traites */
}
// fonction d'impression du tableau
void imptab (int tri[], int type)
{
register int *ptri = tri, i;
if (type == 0)
```

```
printf ("\n\tImpression du tableau initial \n");
else if (type == 1)
printf ("\n\tImpression du tableau impair \n");
else
printf ("\n\tImpression du tableau pair \n");
i = 0;
while (*ptri)
{
printf (" %6d ", *ptri++);
if (!*ptri) break;
if (i++ && !(i % 8)) printf ("\n");
}
}
// programme principal realisant les operations suivantes
int main (int argc, char *argv[], char **envp)
{
register int count; // definition de 3 pointeurs
// saisie du tableau a manipuler
count = saisie (tab);
printf (" Vous avez saisi %d nombres dans le tableau\n", count);
// separation des pairs et des impairs en deux tableaux
count = separ (tab, pair, imp);
printf (" Nous avons séparé %d nombres dans le tableau\n", count);
// tri et impression
tri (tab);
imptab (tab, 0);
tri (imp);
imptab (imp, 1);
tri (pair);
imptab (pair, 2);
return 0;
}
```

13. Conseils pour écrire des programmes en C

13.1. Modularité

- Séparer le programme en plusieurs sous-modules chacun s'occupant d'un sous-domaine du programme initial ;
  - Il est plus facile de résoudre plusieurs problèmes simples qu'un seul gros problème ;
  - Un sous-module peut être réutilisé pour autre chose ;
- Usage des fonctions ;
- Usage du Préprocesseur

13.2. Pratique

- Chaque sous-module est constitué de fichiers *.h, *.c, plus un fichier *.c pour le programme principal;
- Indentez les programmes (facile à faire dans un environnement de développement)
  - Introduire une tabulation pour chaque niveau d'imbrication des instructions
  - Commentez vos programmes /* blabla....*/ ou // blabla....
  - En-tête de fichier

```
/* fonction truc
Paramètres entrée: paramètres sortie :
fonctionnalité:
fonctions appelées:
Auteurs : Date:
autres : */
```

Série d'exemples en C

Prérequis	49
Exemple 0 - rappels	50
Exemple 1 - programme Bonjour le monde	50
Exemple 2 – utilisation de scanf « saisie au clavier »	51
Exemple 3 – utilisation de if...else...	51
Exemple 4 – la boucle while	51
Exemple 5 – programme des nombres premiers	52
Exemple 6 - utilisation des arguments en ligne de commandes	53
Exemple 7 – les tableaux	54
Exemple 8 – les fonctions	55
Exemple 9 - les commentaires	55
Exemple 10 - les structures en C	56
Exemple 11 – la serie de Fibonacci	57
Exemple 12 – les fichiers	58

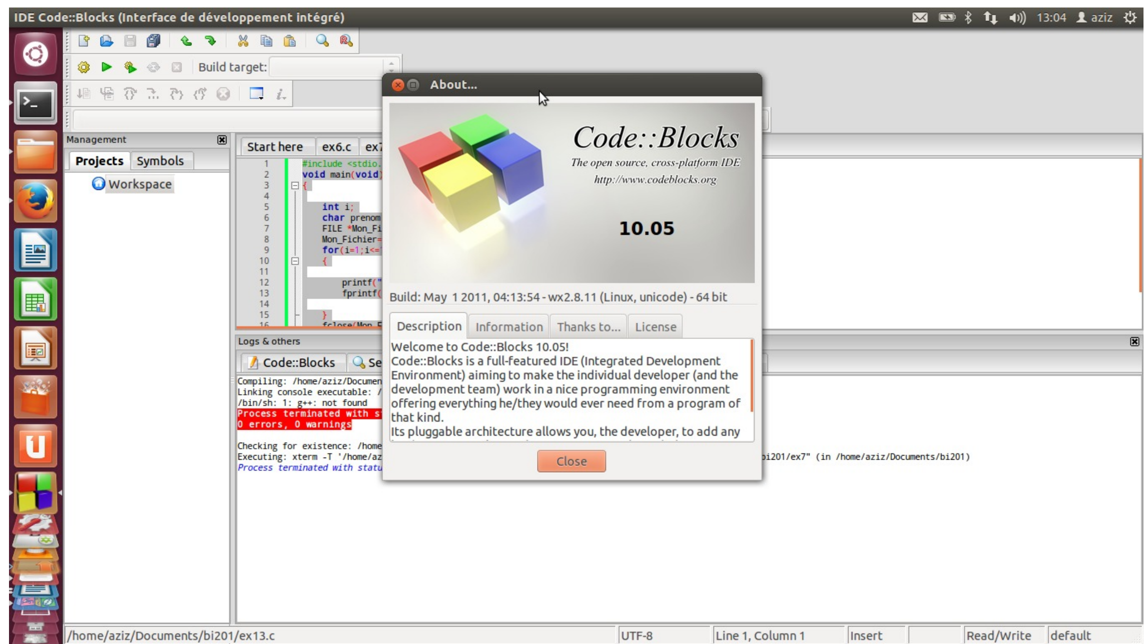
Objectifs

En complément à ce que nous avons vu jusque là, voici quelques exemples avec corrigés de programmes en C, résumant l'ensemble de notions et de concepts nécessaires pour démarrer sérieusement sur des bases solides le développement en langage C.

Il vous sera demandé de copier, de lire et de comprendre, puis de compiler et d'exécuter ces programmes dans votre IDE préféré " je vous recommande *Code: :Blocks* ".

1. Prérequis

- Installation de *Code::Blocks* :
  - pour Windows : télécharger l'IDE depuis : *http://www.codeblocks.org/downloads*
  - pour Ubuntu : *sudo apt-get install codeblocks build-essential*
- Pour compiler et exécuter les programmes ; aller dans le menu build, choisir compiler puis exécuter ou run. Dans un terminal « sous ubuntu » : *gcc program1.c -o program1*, ensuite pour exécuer : *./program1*



2. Exemple 0 - rappels

Ques tion :

Distinguer et classifier les éléments (commentaires, variables, déclarations, instructions, etc, ...) qui composent le programme suivant :

```
#include <stdio.h>
main()
{
int a, b, somme;
printf("Veuillez saisir un entier au clavier\n");
scanf("%d", &a);
printf("Veuillez saisir un autre entier\n");
scanf("%d", &b);
somme = a + b;
printf("La somme de %d et %d est %d \n", a, b, somme);
return 0;
}
```

Réponse :

- `#include<stdio.h>` : commande au compilateur, pour pouvoir utiliser les fonctions printf et scanf.
- `main()` : fonction prinicpale du programme, elle n'a pas de paramètres (la liste des paramètres est vide) et fournit par défaut un résultat du type int (à l'environnement).
- Les variables : a , b et somme déclarées comme entiers (type int).
- Les fonctions : printf et scanf de la bibliothèque <stdio>.
- Les opérateurs : = opérateur d'affectation, + opérateur d'addition
- return 0 : retourne la valeur zéro comme code d'erreur à l'environnement après l'exécution du programme .
- Les mots clef : int, return

3. Exemple 1 - programme Bonjour le monde

```
/* Affichage d'une chaine de caractères à l'écran*/
#include <stdio.h>
main()
{
printf("Bonjour le monde\n");
return 0;
}
```

Sortie du programme :

`Bonjour le monde`

4. Exemple 2 – utilisation de scanf « saisie au clavier »

```
#include <stdio.h>
main()
{
int number;
printf("Saisissez un entier\n");
scanf("%d",&number);
printf("Vous avez saisi %d\n", number);
}
return 0
```

Sortie du programme :

```
Saisissez un entier

7

Vous avez saisi 7
```

5. Exemple 3 – utilisation de if...else...

```
#include <stdio.h>
main()
{
int x = 1;
if ( x == 1 )
printf("x est egal à un.\n");
else
printf("x est différent de un.\n");
}
return 0;
```

Sortie :

```
x est egale à un.
```

6. Exemple 4 – la boucle while

```
#include <stdio.h>
main()
{
int value = 1;
while(value<=3)
{
printf("La valeur de value est %d\n", value);
value++;
}
return 0;
}
```

Sortie :

```
La valeur de value est 1

La valeur de value est 2

La valeur de value est 3
```

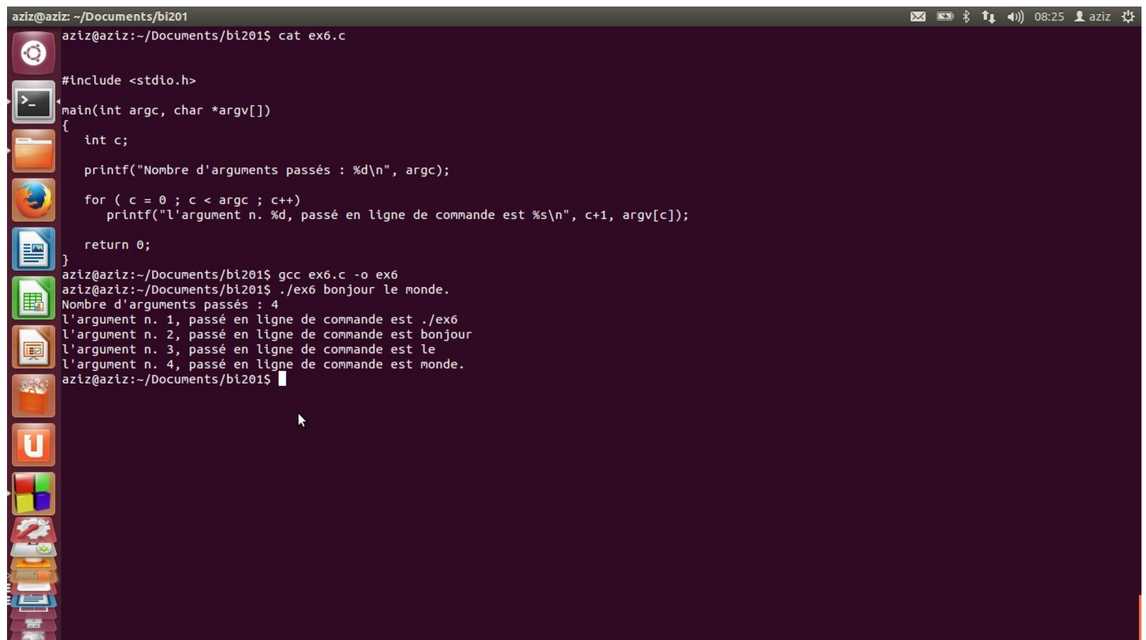
7. Exemple 5 – programme des nombres premiers

```
#include <stdio.h>
main()
{
int n, c ;
printf("Entrez un nombre\n");
scanf("%d", &n);
if ( n == 2 )
printf("Nombre premier.\n");
else
{
for ( c = 2 ; c <= n - 1 ; c++ )
{
if ( n % c == 0 )
break;
}
if ( c != n )
printf("N'est pas premier.\n");
else
printf("Nombre premier.\n");
}
return 0;
}
```

```
aziz@aziz: ~/Documents/bi201
aziz@aziz:~/Documents/bi201$ cat ex5.c

#include <stdio.h>

main()
{
   int n, c ;
   printf("Entrez un nombre\n");
   scanf("%d", &n);
   if ( n == 2 )
      printf("Nombre premier.\n");
   else
   {
       for ( c = 2 ; c <= n - 1 ; c++ )
       {
           if ( n % c == 0 )
              break;
       }
       if ( c != n )
          printf("N'est pas premier.\n");
       else
          printf("Nombre premier.\n");
   }
   return 0;
}
aziz@aziz:~/Documents/bi201$ gcc ex5.c -o ex5
aziz@aziz:~/Documents/bi201$ ./ex5
Entrez un nombre
7
Nombre premier.
aziz@aziz:~/Documents/bi201$ ./ex5
Entrez un nombre
12
N'est pas premier.
aziz@aziz:~/Documents/bi201$
```

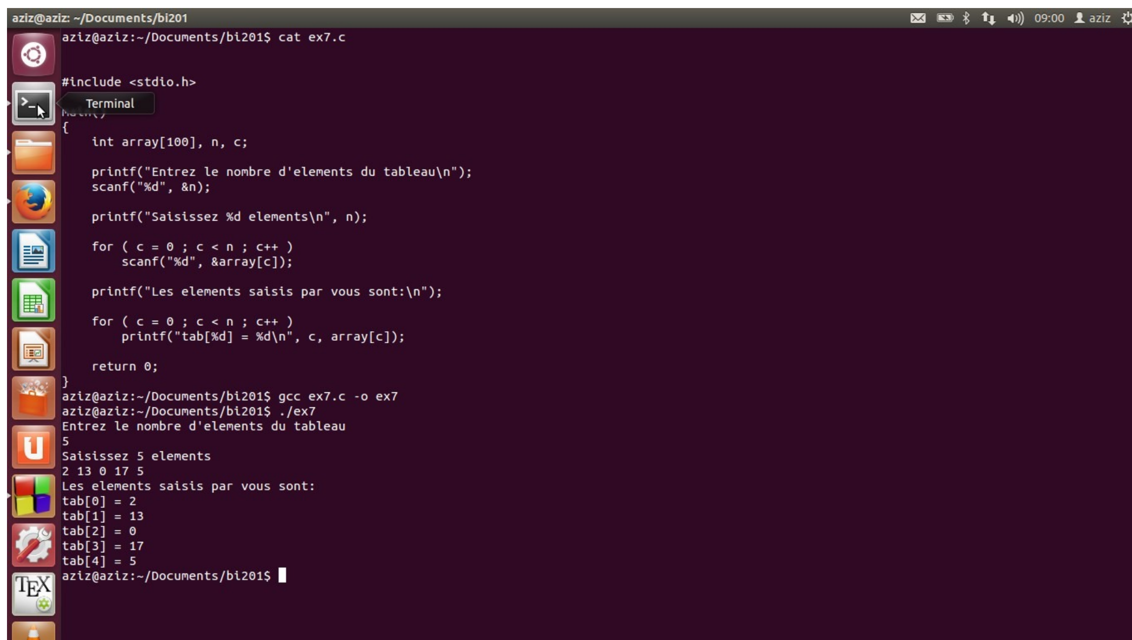
8. Exemple 6 - utilisation des arguments en ligne de commandes

```
#include <stdio.h>
main(int argc, char *argv[])
{
int c;
printf("Nombre d'arguments passés : %d\n", argc);
for ( c = 0 ; c < argc ; c++)
printf("l'argument n. %d, passé en ligne de commande est %s\n",
c+1, argv[c]);
return 0;
}
```



9. Exemple 7 – les tableaux

```
#include <stdio.h>
main()
{
int array[100], n, c;
printf("Entrez le nombre d'elements du tableau\n");
scanf("%d", &n);
printf("Saisissez %d elements\n", n);
for ( c = 0 ; c < n ; c++ )
scanf("%d", &array[c]);
printf("Les elements saisis par vous sont:\n");
for ( c = 0 ; c < n ; c++ )
printf("tab[%d] = %d\n", c, array[c]);
return 0;
}
```



10. Exemple 8 – les fonctions

```
#include <stdio.h>
void my_function();
main()
{
printf("La fonction main.\n");
my_function();
printf("Retour à la fonction main.\n");
return 0;
}
void my_function()
{
printf("Bienvenue à my_function.\n");
}
```

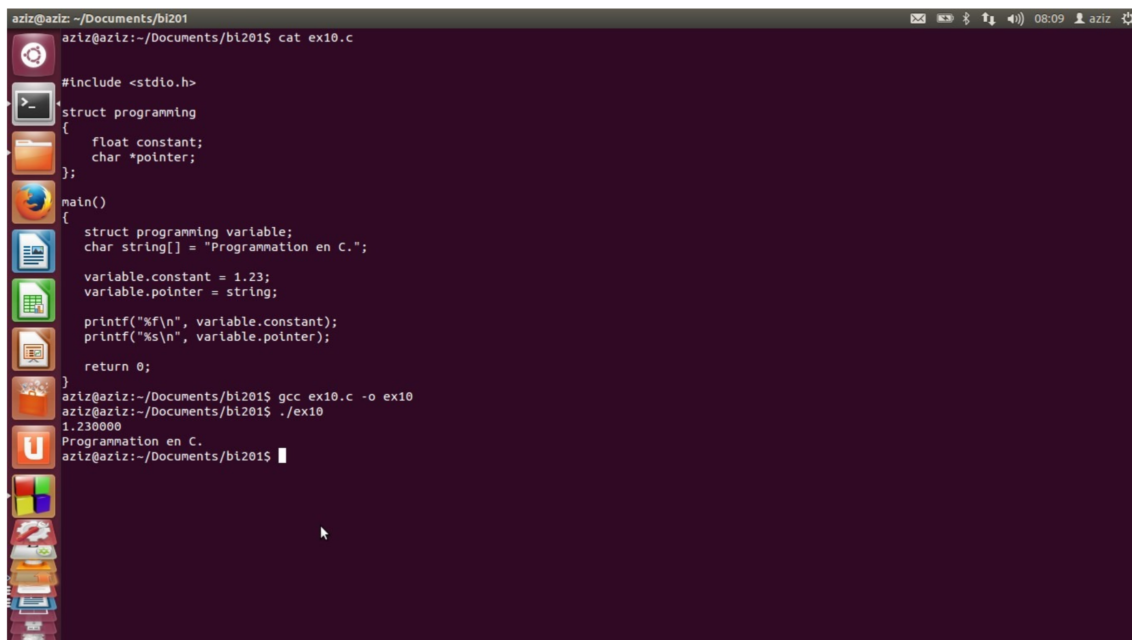
```
aziz@aziz: ~/Documents/bi201
aziz@aziz:~/Documents/bi201$ cat ex8.c

#include <stdio.h>

void my_function();

main()
{
   printf("La fonction main.\n");

   my_function();

   printf("Retour à la fonction main.\n");

   return 0;
}

void my_function()
{
   printf("Bienvenue à my_function.\n");
}
aziz@aziz:~/Documents/bi201$ gcc ex8.c -o ex8
aziz@aziz:~/Documents/bi201$ ./ex8
La fonction main.
Bienvenue à my_function.
Retour à la fonction main.
aziz@aziz:~/Documents/bi201$
```

11. Exemple 9 - les commentaires

```
#include <stdio.h>
main()
{
// Un commentaire en une seule ligne
printf("Ceci est un commentaire en une seule ligne.\n");
/*
* La syntaxe d'un bloc de commentaires,
* elle nous aide à mieux comprendre et facilement maintenir le
code.
* Avez vous déjà écrit des commentaires en plusieurs lignes ?
*/
printf("Et ceci est un bloc de commentaires.\n");
}
return 0;
```

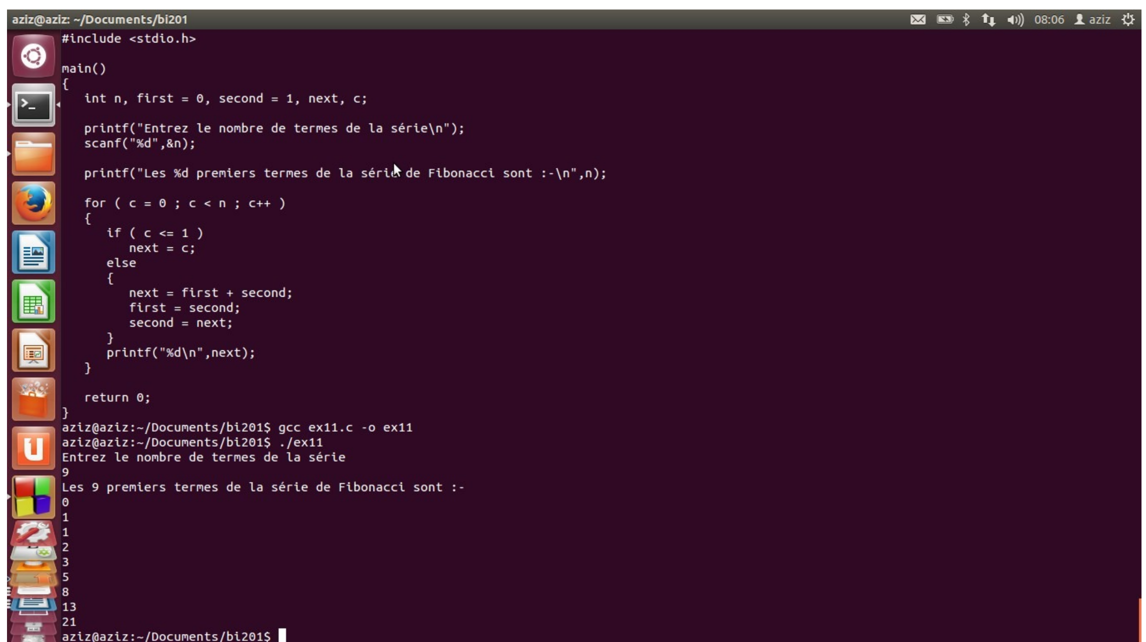
12. Exemple 10 - les structures en C

```
#include <stdio.h>
struct programming
{
float constant;
char *pointer;
};
main()
{
struct programming variable;
char string[] = "Programmation en C.";
variable.constant = 1.23;
variable.pointer = string;
printf("%f\n", variable.constant);
printf("%s\n", variable.pointer);
return 0;
}
```



13. Exemple 11 – la serie de Fibonacci

```
#include <stdio.h>
main()
{
int n, first = 0, second = 1, next, c;
printf("Entrez le nombre de termes de la série\n");
scanf("%d",&n);
printf("Les %d premiers termes de la série de Fibonacci sont
:-\n",n);
for ( c = 0 ; c < n ; c++ )
{
if ( c <= 1 )
next = c;
else
{
next = first + second;
first = second;
second = next;
}
printf("%d\n",next);
}
return 0;
}
```



14. Exemple 12 – les fichiers

```
#include <stdio.h>
void main(void)
{
int i;
char prenom[25];
FILE *Mon_Fichier;
Mon_Fichier=fopen("/home/aziz/fichier.txt","w");
for(i=1;i<=10;i++)
{
printf("Rentrez un prénom :\n");scanf("%s",prenom);
fprintf(Mon_Fichier,"%s\n",prenom);
}
fclose(Mon_Fichier);
}
```

```
aziz@aziz: ~/Documents/bi201
        printf("Rentrez un prénom :\n");scanf("%s",prenom);
        fprintf(Mon_Fichier,"%s\n",prenom);

    }
    fclose(Mon_Fichier);

}
aziz@aziz:~/Documents/bi201$ ./ex13
Rentrez un prénom :
mohamed
Rentrez un prénom :
ahmed
Rentrez un prénom :
ali
Rentrez un prénom :
aziz
Rentrez un prénom :
samir
Rentrez un prénom :
imen
Rentrez un prénom :
karima
Rentrez un prénom :
omar
Rentrez un prénom :
khaled
Rentrez un prénom :
issam
aziz@aziz:~/Documents/bi201$ cat /home/aziz/fichier.txt
mohamed
ahmed
ali
aziz
samir
imen
karima
omar
khaled
issam
aziz@aziz:~/Documents/bi201$
```

Bibliographie

[1] Cours d'Informatique de base 1 V2.0, Benierbah Said, Université Mentouri de Constantine, 2013

[2] Notes de cours d'Informatique, Jacques Tisseau, Ecole Nationale d'Ingénieurs de Brest (ENIB), 2009

[3] Cours d'Algorithmique, Structures de données et langage C, J. M. Enjalbert, Université de Toulouse 3, 2005

[4] Notes de cours Initiation à l'algorithmique, M. Delest, Université de Bordeaux1, 2007