Algmique

et

Programmation en Langage C

~

**Cours – Exercices – Mini projets**

**2018-2019**

**ING3 Nouveaux**

**Frédéric Ravaut**

Enseignant – Chercheur

**TRES IMPORTANT – ENSEIGNANTS ET ELEVES**

La préparation des exercices de fin de chapitre est OBLIGATOIRE, **même si vous enseignants n’en font pas la demande.** Elle pourra être contrôlée par le chargé de TD-TP ou validée par une interrogation écrite de 20 minutes chaque semaine en TD ou TP portant sur un ou plusieurs des exercices à préparer ou similaires. Le chargé de TD-TP ramassera l’intégralité des copies mais ne corrigera qu’un nombre limité de copies à chaque fois. Les notes collectées sur le semestre seront moyennées et constitueront la note de suivi unique du semestre pour le module informatique.

Les élèves utiliseront un outil de « versioning » (au choix) pour tous les projets en groupe. Cet outil permet le travail collaboratif et le contrôle d’investissement par les chargés de TD-TP.

La plupart des fonctionnalités décrites dans ce polycopié sont compatibles avec l’environnement WINDOWS et l’environnement LINUX. Toutefois, certaines (comme le getch par exemple) ne fonctionnent que sous WINDOWS. Utilisez internet ou questionnez votre chargé de cours pour trouver l’équivalent dans un autre système d’exploitation.

**Sous MAC, vous pourrez utiliser l’environnement de développement XCode, mais là aussi vous devrez chercher les équivalents des fonctionnalités WINDOWS. Il est vivement déconseillé d’investir dans un MAC pour cet enseignement. Vous pourrez néanmoins utiliser des outils comme DUALBOOT pour installer une partition ou une machine virtuelle sous Windows sur votre Mac et ainsi simuler un PC.**

Nous vous rappelons que l’entraide est encouragée à l’ECE, mais cela signifie un réel travail de réflexion en commun, notamment pour la partie conception de solution. La transformation de la conception en code devra ensuite donner lieu à un travail original pour chaque élève ou chaque binôme, en fonction des consignes.

**Le don de code déjà écrit est strictement interdit, sous peine de lourdes sanctions. Tout comme le plagiat, qui résulte en l’appropriation du travail d’autrui.**

**Algorithmique et programmation en langage C**

**SOMMAIRE**

1 Chapitre 1 - Objectifs de ce cours 13

1.1 Le rôle de l’ingénieur – ce qu’on attend de vous ! 13

1.2 Haut niveau / Bas niveau 13

1.3 Haut niveau : Le problème à résoudre, décrit dans le Cahier Des Charges - CDC 14

1.4 Bas niveau : La logique informatique 14

1.5 En résumé, la conduite de projet 15

1.5.1 Phase préliminaire collaborative avec une « équipe au complet » 15

1.5.2 Phase de répartition des tâches en « sous équipes » 15

1.5.3 Phase de « fusion » 16

1.6 Pour valider votre année… 16

1.7 Exercices d’application 17

1.7.1 Exercice 101 17

1.7.2 Exercice 102 (Réflexion – CDC) 17

1.7.3 Exercice 103 (Réflexion – CDC) 17

2 Chapitre 2 - Phase Conception : Données / Traitements / Interface 19

2.1 La schématisation Données/Traitements/Interface - DTI 19

2.2 Un mot sur le Cahier Des Charges (CDC) 20

2.2.1 Identifier les noms 20

2.2.2 Identifier les verbes 20

2.3 Première étape de la méthodologie DTI : le modèle de DONNEES 21

2.4 Deuxième étape : Les TRAITEMENTS et l’Analyse Chronologique et Descendante du problème 21

2.4.1 Analyse chronologique 21

2.4.2 Analyse descendante 21

2.5 Troisième étape : l’Interface Homme-Machine (IHM) pour les Entrées/Sorties externes 22

2.6 Illustration sur des exemples 22

2.6.1 Premier exemple : recherche d’un maximum 22

2.6.2 Deuxième exemple : calcul du PGCD 25

2.7 Retour sur la notion de ressource 26

2.8 Exercices d’application 26

2.8.1 Exercice 201 26

2.8.2 Exercice 202 (Réflexion) 27

2.8.3 Exercice 203 (Réflexion – Trace d’exécution) 28

3 Chapitre 3 - Conception (suite) : Algorithme et Organigramme 29

3.1 Algorithme 29

3.1.1 Conception de l’algorithme 29

3.1.2 Avantages 29

3.1.3 Exemple : Algorithme de calcul du PGCD par la méthode d’Euclide 29

3.2 Organigramme 30

3.3 Exemple du PGCD 31

3.4 Cas d’utilisations, jeux d’essai, tests et traces d’exécution 31

3.4.1 Cas d’utilisation et jeux d’essai 32

3.4.2 Trace d’exécution 32

3.5 Exercices d’application 32

3.5.1 Exercice 301 Données/Traitements/Interface 33

3.5.2 Exercice 302 DTI - Analyse Chronologique Descendante 33

3.5.3 Exercice 303 DTI – ACD + ALGO 33

3.5.4 Exercice 304 (DTI/ACD+ALGO+ORGA) 33

3.5.5 Exercice 305 (DTI/ACD+ALGO+ORGA) 34

3.5.6 Exercice 306 (ACD+ALGO/ORGA+TRACE) 34

3.5.7 Exercice 307 (ACD+ORGA) 34

4 Chapitre 4 - Phase de Réalisation : le Code 35

4.1 Programmation en langage dit « de haut niveau » 35

4.1.1 Le langage choisi pour la première année : Le C… Historique 36

4.1.2 Edition d’un programme C 36

4.1.3 Les instructions 38

4.1.4 Les commentaires 38

4.1.5 Un exemple sous CodeBlocks 38

4.1.6 Compilation ou Interprétation 39

4.1.7 Création du fichier « objet » .obj en compilation séparée 39

4.1.8 Création du fichier « exécutable » : Edition des liens 39

4.2 Dernière étape : Définition du protocole de test d’exécution 40

4.3 Qu’est-ce qu’un « bon » résultat ? 40

4.4 Exercices d’application 40

5 Chapitre 5 - L’ordinateur : Rôle et fonctionnement 41

5.1 Composition d’un ordinateur : le Hardware 41

5.1.1 Extérieur : l’Unité Centrale (UC) et ses périphériques d’Entrée/Sortie 41

5.1.2 Intérieur : les constituants de l’Unité Centrale : Mémoire vive, mémoire de masse, processeur… 41

5.2 Rôle de l’ordinateur : manipuler de l’information 42

5.2.1 La communication homme-machine : De l’électricité à l’homme : l’empilement des couches systèmes et logicielles 42

5.2.2 La mémoire vive et le processeur : le ying et le yang 43

5.2.3 Transfert des données mémoire 🡪 processeur : les bus 43

5.3 Stockage des données : mémoire vive, morte et mémoire de masse 43

5.3.1 Les données 43

5.3.2 Les mémoires 43

5.3.3 Retour sur les variables 45

5.3.4 La FAT (File Allocation Table), le NTFS (NT File System) 45

5.3.5 Le Pourquoi du formatage 45

5.4 Les traitements des données : le processeur 45

5.4.1 La notion de registre 45

5.4.2 Les bus de données 45

5.4.3 L’ordonnancement des instructions 45

5.5 Exercices d’application 46

5.5.1 Exercice 501 Présentation de l’ordinateur 46

5.5.2 Exercice 502 Présentation de l’ordinateur 46

5.5.3 Exercice 503 Fonctionnement de l’ordinateur 46

TP de Mise en route 47

\* Le noyau du système d’exploitation Windows 47

\* Le noyau du système d’exploitation LINUX 47

6 Chapitre 6 - Codage de l’information 53

6.1 Le binaire 53

6.1.1 0 ou 1 : le courant passe t’il ? 53

6.1.2 1 fil, 2 fils…n fils 53

6.2 Le codage/décodage de l’info : La Table d’Interprétation Sémantique - TIS 53

6.3 Exercices d’application 54

6.3.1 Exercice 601 Codage de l’information 54

7 Chapitre 7 - Stockage des données : variables de type scalaire 55

7.1 Rôle et convention d’écriture 55

7.2 Déclaration de variable 55

7.3 Le domaine de définition et la plage de valeurs 55

7.3.1 Les altérateurs de type : short, long, signed, unsigned 55

7.4 Les types scalaires de données en C : int, float, char, pointeur 56

7.4.1 Nombre à virgule : float 56

7.4.2 Nombre entier : int 57

7.4.3 Caractère : char 58

7.4.4 Pointeurs (int\*, char\*, float\*, int\*\*\*\*\*,…) 58

7.5 Le lieu de déclaration et portée 59

7.5.1 Analyse d’un exemple 59

7.5.2 Notion de variable globale 59

7.6 Créer un type de donnée 59

7.7 Conversion de type de données - transtypage 59

7.8 Les booléens VRAI et FAUX 60

7.9 Précisions sur les constantes 60

7.10 L’affectation 61

7.10.1 L’initialisation 61

7.10.2 L’affectation des variables scalaires 61

7.10.3 Les affectations cachées 62

7.11 Exercices d’application 63

7.11.1 Exercice 701 trace d’exécution 63

7.11.2 Exercice 702 bonnes pratiques 63

7.11.3 Exercice 703 Variable scalaires 63

7.11.4 Exercice 704 Variable scalaires 63

7.11.5 Exercice 705 Erreurs de compilation 63

7.11.6 Exercice 706 Affectation 64

7.11.7 Exercice 707 Booléens 64

7.11.8 Exercice 708 Variable scalaires, permutation 64

7.11.9 Exercice 709 (TRACE D’EXECUTION) 64

8 Chapitre 8 - Les pointeurs : type scalaire particulier 67

8.1 Définition 67

8.2 Déclaration et initialisation 67

8.3 Adressage direct 67

8.4 Adressage indirect 67

8.5 Usage 68

8.5.1 Opérateur unaire & 68

8.5.2 Déclaration des pointeurs 68

8.5.3 Opérateur unaire \* 68

8.6 Les pointeurs pour le passage de paramètres par adresse à une fonction 68

8.7 Exercices d’application 69

8.7.1 Exercice 801 (Pointeurs) 69

9 Chapitre 9 - Calculs, comparaisons et operateurs logiques 71

9.1 Les opérateurs mathématiques 71

9.2 Le CAST 71

9.3 Les opérateurs de comparaison 71

9.4 Les opérateurs logiques 71

Tables de vérité des opérateurs logiques 71

9.4.1 Priorité des opérateurs 72

9.5 Opérateurs binaires de traitement des bits 72

9.5.1 & (ET) 72

9.5.2 | (OU) 72

9.5.3 ^ (OU exclusif) 73

9.5.4 op1 << op2 73

(décalage binaire gauche de op1 un nombre de fois égal à 73

9.5.5 op1 >> op2 73

9.5.6 ~ (complément à 1 d’un opérande de type entier, inversion des bits) 73

9.6 Exercices d’application 74

9.6.1 Exercice 901 (LOGIQUE) 74

9.6.2 Exercice 902 (ACD+ALGO) 74

9.6.3 Exercice 903 (ACD+ALGO+TRACE) 74

9.6.4 Exercice 904 (opérateurs et booléens) 74

10 Chapitre 10 - Les entrées/sorties clavier/ecran 75

10.1 Les E/S formatées 75

10.1.1 Ecriture : printf (écrire, traitement de sortie) 75

10.1.2 Lecture : scanf (lire, traitement d’entrée) 75

10.1.3 Les formats 75

10.1.4 Les alignements 76

10.1.5 Les caractères spéciaux 76

10.2 Exercices d’application 76

10.2.1 Exercice 1001 (scalaires, TIS, transtypage) 76

10.2.2 Exercice 1002 (scalaires) 76

10.2.3 Exercice 1003 (E/S) 76

11 Chapitre 11 - Les entrées/sorties en Fichier Texte (ASCII) 77

11.1 Intérêt des fichiers : la permanence des données 77

11.2 Ecriture et lecture dans un fichier 77

11.2.1 Lecture dans le fichier 78

11.2.2 Ecriture dans le fichier 78

11.2.3 Progression dans le fichier 79

11.2.4 Fin de fichier 79

11.3 Droits d’accès aux fichiers permanents 79

11.3.1 Ouverture et droit d’accès 80

11.3.2 Fermeture du fichier 80

11.4 Exercices d’application 81

11.4.1 Exercice 1101 (Fichiers) 81

11.4.2 Exercice 1102 (Fichiers) 81

11.4.3 Exercice 1103 (Scalaires, fichiers) 81

11.4.4 Exercice 1104 (Opérateurs) 81

12 Chapitre 12 - Les Tests 83

12.1 Les structures de contrôle conditionnelles : les tests 83

12.1.1 Test If 83

12.1.2 Test If…Else… 83

12.1.3 Test « Switch » (branchement) 83

12.2 Exercices d’application 84

12.2.1 Exercice 1201 Modulo 84

12.2.2 Exercice 1202 Cast 84

12.2.3 Exercice 1203 Tests, intervalles, transtypage 84

13 Chapitre 13- Les Boucles de répétition 85

13.1 Les structures de contrôle répétitives : les boucles 85

13.1.1 Boucle FOR (nombre de répétitions connu) 85

13.1.2 Boucle DO…WHILE… (au moins une fois) 85

13.1.3 Boucle WHILE… (peut-être pas du tout) 85

13.2 Exercices d’application 86

13.2.1 Exercice 1301 Observation et correction 86

13.2.2 Exercice 1302 87

13.2.3 Exercice 1303 87

13.2.4 Exercice 1304 87

14 Chapitre 14 - Les sous-programmes en langage C 89

14.1 Modularité du code 89

14.1.1 Procédure ou fonction ? 89

14.1.2 Mécanisme d’appel 89

14.2 Visibilité des sous-programmes 90

14.2.1 Prototype d’un sous-programme 90

14.2.2 Déclaration en fichier bibliothèque .h 91

14.3 Les procédures : Le type void 91

14.4 Les fonctions : Le mécanisme de retour 91

14.5 Appel des sous-programmes 92

14.5.1 Les fonctionnalités prédéfinies du langage 92

14.5.2 Nos propres fonctionnalités 92

14.5.3 Appel de fonction, appel de procédure 93

14.6 Graphes d’appel 93

14.7 Exercices d’application 93

14.7.1 Exercice 1401 94

14.7.2 Exercice 1402 94

14.7.3 Exercice 1403 94

14.7.4 Exercice 1404 94

14.7.5 Exercice 1405 94

14.7.6 Exercice 1406 94

15 Chapitre 15 - Données de type complexe : Les tableaux 95

15.1 Les tableaux monodimensionnels 95

15.1.1 Accès 95

15.1.2 Schéma de la représentation classique 95

15.2 Un pointeur caché dans un tableau monodimensionnel 96

15.2.1 Principe d’adressage 96

15.3 Les tableaux multidimensionnels 96

15.3.1 Schéma de la représentation classique 96

15.3.2 Principe d’adressage 97

15.3.3 Accès 97

15.3.4 Simulation d’une matrice en tableau monodimensionnel 97

15.4 Exercices d’application 98

15.4.1 Exercice 1501 98

15.4.2 Exercice 1502 98

15.4.3 Exercice 1503 98

15.4.4 Exercice 1504 98

15.4.5 Exercice 1505 98

15.4.6 Exercice 1506 98

15.4.7 Exercice 1507 98

15.4.8 Exercice 1508 99

15.4.9 Exercice 1509 99

16 Chapitre 16 - Données complexes : les chaînes de caractères 101

16.1 Convention 101

16.2 Déclaration d’une chaîne 101

16.3 Saisie 101

16.3.1 Plusieurs fonctions de saisie possibles 102

16.3.2 Problème d’espace mémoire : la zone de stockage doit avoir été réservée. 102

16.4 Affichage à l’écran : printf(« %s », ch) ; 102

16.5 Autres fonctions dédiées 102

16.5.1 Affectation 102

16.5.2 Concaténation 102

16.5.3 Comparaison 102

16.5.4 Dimension 103

16.6 Exercices d’application 103

16.6.1 Exercice 1601 103

16.6.2 Exercice 1602 103

17 Chapitre 17 - Les structures et les unions 105

17.1 Les Structures 105

17.1.1 Déclaration d’un type structure 105

17.1.2 Déclaration d'une variable de type structure (ne pas confondre) 106

17.1.3 Les données membres d'une structure : les champs 106

17.1.4 Accès aux champs d’un objet structure. 106

17.1.5 Accès aux champs d’un objet structure via un pointeur 107

17.1.6 Accès aux données membres de structure dans le cas de structures imbriquées 108

17.1.7 Opération sur les structures 108

17.2 Les tableaux de structures 109

17.3 Le chaînage de structures 110

17.4 Les unions 110

17.5 Exercices d’application 112

17.5.1 Exercice 1701 112

17.5.2 Exercice 1702 112

18 Chapitre 18 – Génération de Nombres Aléatoires 113

18.1 Génération de nombres aléatoires 113

18.1.1 Inclusion des fichiers d’entêtes 113

18.1.2 Initialisation du générateur aléatoire 113

18.1.3 Génération du nombre par la fonction RAND 113

18.2 Exercices d’application 114

18.2.1 Exercice 1801 114

18.2.2 Exercice 1802 114

18.2.3 Exercice 1803 114

Image, histogramme et binarisation 115

18.2.4 Exercice 1105 115

18.2.5 Exercice 1106 115

18.2.6 Exercice 1107 115

19 Chapitre 19 - Programmation Evénementielle 117

19.1 Evénement, action, démon de surveillance 117

19.2 Exercices d’application 118

19.2.1 Exercice 1901 118

19.2.2 Exercice 1902 118

19.2.3 Exercice 1903 118

19.2.4 Exercice 1904 118

19.2.5 Exercice 1905 119

19.2.6 Exercice 1906 119

20 Chapitre 20 - Allocation dynamique de mémoire 121

20.1 Retour sur la déclaration automatique de données 121

20.1.1 Variables scalaires 121

20.1.2 Tableaux 121

20.1.3 Pointeurs 121

20.1.4 Chaînes de caractères constantes 121

20.2 Problème et solution 122

20.3 Principe de l’allocation 122

20.3.1 La fonction malloc 122

20.3.2 Echec de l’allocation 122

20.3.3 La fonction sizeof 122

20.3.4 Cas du tableau monodimensionnel 123

20.3.5 Allocation dynamique d’un tableau multidimensionnel 123

20.3.6 Cas des chaînes de caractères 124

20.4 La fonction free 125

20.5 Exercices d’application 125

20.5.1 Exercice 2001 126

20.5.2 Exercice 2002 126

20.5.3 Exercice 2003 126

20.5.4 Exercice 2004 126

20.5.5 Exercice 2005 126

20.5.6 Exercice 2006 126

20.5.7 Exercice 2007 127

20.5.8 Exercice 2008 127

20.5.9 Exercice 2009 127

20.5.10 Exercice 2010 127

20.5.11 Exercice 2011 127

21 Chapitre 21 - Passage des paramètres – Le tube 129

21.1 Le passage par valeur 129

21.2 Le passage par adresse 129

21.3 Cas particulier des tableaux 130

21.4 Passage d’arguments à la fonction main : la ligne de commande 130

21.5 Exercices d’application 131

21.5.1 Exercice 2101 131

21.5.2 Exercice 2102 131

21.5.3 Exercice 2103 132

21.5.4 Exercice 2104 132

21.5.5 Exercice 2105 132

21.5.6 Exercice 2106 : La ligne de commande 132

21.5.7 Exercice 2107 132

22 Chapitre 22 La récursivité 133

22.1 Définition 133

22.2 Utilité 133

22.3 Notion de clone 133

22.4 Condition d’arrêt 134

22.5 Procédure récursive 134

22.6 Fonction récursive 134

22.7 Exercices d’application 134

22.7.1 Exercice 2201 134

22.7.2 Exercice 2202 134

22.7.3 Exercice 2203 134

23 Chapitre 23 - Les fichiers : ascii et binaires 135

23.1 Rappels sur les fichiers ASCII 135

23.1.1 Création et Ouverture d'un fichier 135

23.1.2 Lecture/Ecriture dans un fichier texte 136

23.1.3 Fermeture d'un fichier 136

23.1.4 Principe de manipulation des fichiers 137

23.1.5 Le caractère de fin de fichier (EOF) 139

23.2 Fichiers binaires 139

23.3 Exercices d’application 141

23.3.1 Exercice 2301 141

23.3.2 Exercice 2302 141

23.3.3 Exercice 2303 141

23.3.4 Exercice 2304 141

23.3.5 Exercice 2305 141

23.3.6 Exercice 2306 141

23.3.7 Exercice 2307 142

23.3.8 Exercice 2308 142

23.3.9 Exercice 2309 142

24 Chapitre 24 - Structures de données complexes : listes chaînées, table de hachage, arbre, graphe… 143

24.1 Les listes chaînées 143

24.1.1 Définition 143

24.1.2 Construction 143

24.1.3 Parcours 144

24.1.4 Ajout 144

24.1.5 Liste doublement chaînée 144

24.1.6 Liste circulaire 144

24.1.7 Listes de listes 144

24.2 Les tables de hachage 145

24.3 Les arbres 145

24.4 Les graphes 145

24.5 Exercices d’application 146

24.5.1 Exercice 2401 146

24.5.2 Exercice 2402 146

24.5.3 Exercice 2403 146

24.5.4 Exercice 2404 146

25 Mini Projet Image : Visualisation, traitement et binarisation via histogramme 147

26 ANNEXE A : Règles d’écriture des programmes en C 152

27 ANNEXE B : Les séquences d'échappement 154

28 ANNEXE C : Les priorités des opérateurs 154

29 ANNEXE D : Classification, conversion de caractère : <ctype.h> 154

30 ANNEXE E : Traitement de chaînes de caractères : <string.h> 155

31 ANNEXE F : TABLE ASCII ET TABLE ASCII ETENDUE 156

# Chapitre 1 - Objectifs de ce cours

Le rôle de l’ingénieur – ce qu’on attend de vous !

Le rôle de l’ingénieur est difficile à définir du fait du **large spectre de ses responsabilités**. Il devra assumer des décisions, guider ses équipes, savoir dialoguer avec tous ses interlocuteurs. Il devra tout d’abord bien comprendre la problématique du client et les contraintes qui accompagnent le projet. L’ingénieur doit également bien maîtriser le fonctionnement des machines et leurs performances pour mettre en place la meilleure solution, tous critères confondus.

Il s’agit donc de démêler le **sac de nœuds** que représentele **cahier des charges -** **CDC**, voire d’aider à son écriture et ensuitede définir et suivre une **méthodologie de réalisation** et de proposer une **solution efficace** à votre client.

Proposer une solution, ça ne veut pas forcément dire « développer » une solution originale de A à Z. Ça veut dire **exploiter ses connaissances** personnelles du domaine concerné, exploiter **l’expérience acquise** au fil des années d’activité et de ses propres centres d’intérêt et mettre en place une **communication efficace** avec le client pour ne pas faire fausse route dans ses **choix**. Sans oublier les fameux **tests réguliers,** que tout débutant minimise malheureusement systématiquement, commettant ici sa plus grande erreur ! (ne faut-il pas « toucher » soi-même pour vérifier que c’est chaud, même si on était prévenu ? ;-)

Des connaissances personnelles, vous en avez sûrement. De l’expérience acquise, vous allez en accumuler tout au long de cette formation. Des erreurs, vous allez en faire beaucoup (j’espère !) et vous devriez être capable d’en tirer beaucoup d’enseignements. Vos choix, il faudra les justifier. Il vous faudra être « **ingénieux** » et choisir au mieux vos **traitements** et vos **tests**.

Haut niveau / Bas niveau

Si on précise ce qui vient d’être dit, on peut constater que votre travail va à la fois consister à :

1. Apprendre à communiquer avec le client en comprenant et utilisant un vocabulaire qui lui est familier ; pour lui poser les bonnes questions et bien comprendre son besoin : c’est l’écriture ou la finalisation du **cahier des charges ou CDC** ;
2. Apprendre le fonctionnement et les limites d’une machine pour en tirer le meilleur afin qu’elle réponde à la problématique, en la pilotant au moyen d’un langage de programmation subtilement choisi (il y en a plusieurs dizaines !).

Ces deux aspects, qui se positionnent aux deux extrêmes de votre travail, seront qualifiés dans la suite de ce cours de « haut niveau » et « bas niveau ».

*L’Homme et ses problèmes existentiels,*

*par exemple « Comment faire voler les avions avec encore plus de sécurité ? »*

*La machine et ses composants électroniques qui évoluent sans cesse.*

HAUT

BAS

Tout ça pour vous dire que si vous souhaitez apprendre à programmer, ou simplement goûter succinctement à **la programmation** dans une formation d’ingénieur beaucoup plus ambitieuse, il vous faudra être capable de **comprendre et** **faire le lien entre le bas-niveau et le haut-niveau**.

Cette liaison se fait par l’application d’une méthodologie rigoureuse (il en existe plusieurs) dont toutes les étapes sont réfléchies et importantes.

Nous apprendrons cette année la méthode dite **« Données / Traitements / Interface » (DTI)**.

Haut niveau : Le problème à résoudre, décrit dans le Cahier Des Charges - CDC

Plaçons-nous ici au haut-niveau, celui du client et de sa problématique. Vous allez voir, ce n’est pas si simple que ça.

Le haut-niveau, c’est le niveau où le problème à résoudre est exprimé dans un langage oral ou écrit. Prenons le français par exemple (ça facilitera un peu). Imaginez donc : Vous êtes déjà ingénieur (la formation passe vite) et à ce titre vous utilisez votre savoir pour résoudre le problème d’un « client ».

Il vous exprime sa problématique de la façon suivante :

*« J’ai un site de vente en ligne et je voudrais que vous me trouviez une solution pour vendre des articles en France et à l’international, sans prendre le risque de vendre en même temps 2 fois le même article»*

ou encore

*« J’ai 1500 avions qui se posent à Roissy Charles De Gaulle chaque jour, comment organiser au mieux les atterrissages pour qu’ils ne se télescopent pas ? »*

**Quelques conclusions immédiates :**

* Votre client n’est pas informaticien, il n’en a pas le vocabulaire.
* Il n’a pas tout dit, il y a un risque qu’il vous appelle dans quelques jours pour vous faire « Ah au fait, j’oubliais de vous dire…. ».
* Rien ne prouve que ce que vous lui proposerez, après des mois de travail, lui convienne…

Bas niveau : La logique informatique

L’informatique actuelle est organisée autour d’une **logique binaire** « 0/1 », « oui/non » ou encore « vrai/faux ». Pour exploiter cette logique, différents langages proposent des jeux d’instructions incluant affectations, test, boucles, entrées, sorties… De cette logique découle toutes les capacités d’un ordinateur. Les programmes sont créés afin d’exploiter au mieux cette logique et offrir une réponse à un problème donné.

Pour être capable d’exploiter à notre tour cette logique nous allons essayer de comprendre comment fonctionne un ordinateur et apprendre des « méthodes » de programmation :

* **Analyse du besoin, chronologie des étapes de réalisation,**
* **Conception de la solution : algorithme, organigramme, schéma mémoire,**
* **Codage : transformation de la conception en langage de programmation,**
* **Vérification de fiabilité : tests & traces d’exécution…**

**Ces méthodes devront être scrupuleusement respectées cette année.** Vous apprendrez à vous émanciper par la suite, notamment en découvrant dans les années à venir d’autres méthodes et d’autres langages.

Ce cours s’adresse à des étudiants n’ayant jamais pratiqué la programmation, mais ayant malgré tout déjà touché, voire « allumé » un ordinateur et curieux de comprendre un peu mieux comment fonctionne cet outil indispensable à notre vie de tous les jours.

Aujourd’hui vous utilisez l’ordinateur presque par réflexe, habitué à ce qu’il ait réponse à (quasi) tout ! Mais comment fait-il ?

La quantité d’informations qu’un ordinateur est aujourd’hui capable de traiter est quasi illimitée, notamment grâce à Internet qui permet de connecter entre elles les sources de connaissances mondiales. Mais pour bien exploiter ces connaissances il est nécessaire de **savoir aller les chercher**, leur opposer un **esprit critique** pour les **trier**, les **stocker** et les **traiter**. C’est le « D » (Données) et le « T » (Traitements) de notre méthode. Le « I », c’est pour l’Interface de **communication** entre le programme et son utilisateur. Aussi performant que soit votre programme, s’il n’est pas ergonomique et agréable à utiliser il a peu de chance de conquérir son public.

En résumé, la conduite de projet

Vous aurez donc besoin de **maîtriser le haut-niveau**, celui du besoin, celui du langage oral, celui du client qui parle en français sans avoir les idées toujours claires ; et vous aurez également besoin de **maîtriser le bas niveau**, celui de l’ordinateur, celui de l’électronique et des contraintes de la machine sur laquelle vous devrez exécuter le programme. Et bien évidemment, vous aurez besoin de maîtriser toute la chaîne entre les deux, détaillée ci-dessous :

* L’élaboration ou finalisation du **cahier des charges avec le client**,
* **La conception d’une solution :** après une analyse détaillée de la problématique, vous élaborez une solution (on parle ici de schémas et d’idées sur papier, pas encore de code), en respectant une méthodologie adaptée **Données/Traitements/Interface** cette année
  + - Modèle des Données
    - Traitements conduisant au résultat
    - Interface Homme Machine, gestion des Entrées/Sorties externes
* Les tests de validation de la solution développée pour en garantir l’adéquation avec le cahier des charges.
  + 1. Phase préliminaire collaborative avec une « équipe au complet »

Dans le cas d’un projet en équipe, cette vision en trois points D-T-I est tout d’abord réalisée par **l’équipe au complet** afin de bien mesurer l’envergure du projet.

Ce travail préliminaire vous conduira à écrire :

* Les **spécifications**, les « specs » (Qu’est-ce qu’on doit faire ?, comment… ?)
* **L’organisation de l’équipe** pour définir la répartition des tâches si vous travaillez à plusieurs,
* **Le planning prévisionnel** de l’ensemble des étapes**,**
* La définition des **scénarii** (toutes les situations susceptibles de survenir – les « use cases ») pour valider qu’on a bien pensé à tout.

Réaliser ces 4 étapes avec l’équipe au complet permet à tous de comprendre où va aller le projet et comment le « chef de projet » va gérer son avancement.

* + 1. Phase de répartition des tâches en « sous équipes »

Ensuite, l’équipe se divise selon ce qui a été décidé. Chaque « sous-groupe » va alors travailler indépendamment en respectant de nouveau la même méthodologie sur sa partie :

* DTI (eh oui, on recommence sur sa partie)
* rédaction des **algorithmes ou organigrammes**,
* **détermination des entrées, sorties et ressources mémoire locales** pour chaque algorithme (dessin des « boites noires » faisant apparaitre les E/S),
* **tests papiers** des algorithmes en traitant tous les scénarii envisagés, **schéma mémoire** et **trace d’exécution (pour tous les scénarii identifiés)**,
* **codage** dans un langage approprié (le C ?) choisi pour ses performances et son efficacité dans la gestion des **contraintes du bas niveau** (optimisation),
* **compilation** de ce code,
* **exécution** des **tests** intermédiaires
* **corrections éventuelles et retour au codage en créant un système de version qui garantit d’avoir toujours une version incomplète mais fonctionnelle du projet.** (Vouloir tout faire en même temps est la pire des erreurs).

Chaque sous équipe s’attache à fournir le résultat qui a été prévu dans la phase préliminaire de travail en commun, **dans le respect des formats prédéfinis**. Un point régulier permet de contrôler l’avancement de chaque sous-groupe et de redistribuer les forces vives au besoin.

* + 1. Phase de « fusion »

Chaque partie étant terminée, vient alors la phase dite « **d’intégration** » ou « fusion » où l’équipe se regroupe pour fusionner les parties. C’est une étape délicate qui n’a de chance de succès que si les spécifications ont été correctement rédigées et surtout, **respectées** !

* Création d’une nouvelle version du projet qui regroupe toutes les parties fusionnées
* Vérification des entrées/sorties et tests finaux,
* la **validation** de la solution finale avec le client (**recette**)…

Et tout ça en collaboration étroite avec le client, quel qu’il soit, pour optimiser le temps de développement.

Pour valider votre année…

Un dernier rappel schématique pour bien enfoncer le clou sur ces notions qui me semblent très importantes pour comprendre ce que nous attendons de vous cette année. Pour valider votre « Module informatique » vous devrez maîtriser :

HAUT NIVEAU

BAS NIVEAU

Le client, la problématique…

La machine, l’électronique, l’ordinateur, le binaire…

**VOUS ! Élaboration de la solution :**

* **la conception**
* **le programme**
* **les tests**

**Voilà ! Je viens de vous donner un petit aperçu du contenu de cette première** année d’étude. L’acquisition de tout ça ne sera pas forcément simple, vous aurez des difficultés mais on ne vous en voudra pas pour ça. C’est normal. **Ce qui ne l’est pas, en tous cas pour un ingénieur qui se respecte, c’est de ne pas chercher à dépasser ses difficultés. Et pour cela, beaucoup de solutions vous sont proposées à l’école :**

* + vos camarades,
  + l’association tutorat,
  + les professeurs à votre écoute et qui peuvent vous aider en séance individuelle sur rendez-vous...

**Utilisez-les !!!** On vous reprocherait de ne pas le faire et de vous laisser glisser, ce qui reflèterait de la mauvaise attitude que vous pourriez avoir face aux difficultés du monde professionnel.

C’est à vous, **futur ingénieur** de trouver **une** solution !

Exercices d’application

Vous n’êtes pas encore ingénieur ni même informaticien mais vous savez réfléchir.

Etre ingénieur, c’est quoi ? Apporter une solution à un besoin.

Comprendre le besoin : nécessite la compréhension du haut niveau

Élaborer la solution : nécessite la maîtrise du bas niveau.

Entre les 2, les phases **d'analyse**, de **conception** et de **test**.

**But de la formation : acquérir ces notions.**

**Commençons donc par réfléchir en français (avant le C) !!**

Un client vous exprime ses attentes. Il est rarement clair, rarement complet…

1. Identifiez ses attentes, faites préciser les points flous, posez des hypothèses…
2. Déterminez les ressources qui vous seront nécessaires (charge mémoire)
3. Déterminez les traitements (actions) à exécuter et leur ordre d’exécution (charge processeur)

**Rappelez-vous, ce qui nous intéresse ici c’est « comment vous y prendriez-vous pour… ? »**

**A VOUS DE JOUER…**

* + 1. Exercice 101

Redéfinissez en quelques mots le rôle de l’ingénieur.

* + 1. Exercice 102 (Réflexion – CDC)

Le client vous dit : « Je souhaiterais pouvoir afficher le plus petit d'une liste de nombres ».

Qu’avez-vous à faire en tant qu’ingénieur face à ce client ?

(Rappel : on en est au stade de la réflexion en français. Décrivez tout ce que ce genre d’énoncé vous suggère comme idées, interrogations, etc.)

* + 1. Exercice 103 (Réflexion – CDC)

L’ECE souhaite automatiser la réalisation des emplois du temps. Quelles sont les informations à prendre en compte ? (On ne s’intéressera ici qu’aux données en entrée à prendre en considération, pas à la réalisation des EDT)

# Chapitre 2 - Phase Conception : Données / Traitements / Interface

Nous considérons ici que le cahier des charges a été finalisé en étroite collaboration avec le client (ce qui ne veut pas dire que nous n’aurons plus d’interaction avant la recette) et que nous avons les idées claires sur ce qu’il y a à faire. Reste à s’organiser.

Plusieurs méthodologies de conduite de projet existent : le cycle en V, la méthode AGILE, le SCRUM, MVC… et la **méthodologie DTI !**

Aucune de ces méthodes ne répond aux questions suivantes :

* Comment instaurer un dialogue avec le client afin d’obtenir des précisions sur tous les points obscurs ?
* Dans quelle mesure sommes-nous libres de faire des choix lorsque le client n’a pas de précision à apporter ?
* Comment justifier ces choix ?
* Quels sont les bons critères de choix ?
* Comment organiser notre propre pensée ?
* Comment bien choisir notre méthode ?

Puisque ce cours s’adresse à des débutants en programmation, nous allons commencer par faire simple et utiliser une méthode qui distingue 3 aspects du travail à faire.

La schématisation Données/Traitements/Interface - DTI

Traitements

Données

Interface

Le modèle **Données/Traitement/Interface** est une représentation de l’architecture de la solution à mettre en place.

C’est une méthode de conception qui, si elle n’est pas parfaite, offre une vision claire séparant :

* Les structures de **données**, leur nature, leur quantité et leur mode de stockage,
* les **traitements**, leur organisation logique et chronologique,
* l’**interface**, encore appelée Interface-Homme-Machine (IHM) permettant les interactions entre l’utilisateur et le programme.

Elle a été longuement discutée et choisie par l’équipe enseignante de l’ECE pour sa simplicité et sa structuration. Vous en découvrirez beaucoup d’autres dans la suite de votre formation d’ingénieur.

En séparant les données, les traitements et les écrans d’échange entre l’utilisateur et le programme, cette méthode nous permet de nous intéresser aux fondamentaux de l’ordinateur sur lequel l’application sera utilisée :

* la mémoire,
* le processeur,
* l’IHM permettant les échanges entre l’utilisateur et le programme, encore appelée les Entrées/Sorties externes.

INTERFACE

DONNEESS

TRAITEMENTS

Identification des traitements

Gestion des événements

Identification de la logique (la chronologie) des actions

Analyse des actions du CDC (les verbes)

**Entrées / Sorties**

IHM / Design

Obtention des données

Affichage des résultats

Validation des E/S externes

Identification des données

Analyse des ressources du CDC (les noms)

Pour préparer la déclaration des variables

Analyse chronologique et descendante (ACD)

Un mot sur le Cahier Des Charges (CDC)

Le **cahier des charges** CDC est un document « **contractuel** » qui vous lie à votre client. Il précise les attentes de ce dernier et vous fournit un cadre d’exécution.

Rien de ce qui n’est écrit au cahier des charges ne peut vous être exigé, et tout manquement à ce qui y figure, en termes de respect des contenus et des délais, peut entraîner des pénalités (en euros, centaines d’euros, voire millier d’euros dans la vie professionnelle ; en terme de points sur 20 à l’école.

Armez-vous d’un surligneur et lisez attentivement le cahier des charges. Si vous dialoguez avec le client, prenez un papier et un stylo et prenez des notes de ce qu’il vous dit…

* + 1. Identifier les noms

Les **noms** représentent la plupart du temps les données, les informations à manipuler. Ces données devant être stockées en mémoire, les identifier toutes permet d’évaluer les ressources « mémoire » nécessaires à votre programme (nature & quantité) : **la charge mémoire**. Cette charge regroupe :

* Les données en ENTREE
* Les données en SORTIE
* Les données internes de traitement

L’identification des données vous aide à construire le modèle de données, à choisir les structures de données les plus adaptées et à faire vos déclarations de variables.

* + 1. Identifier les verbes

Les verbes correspondent souvent aux actions, aux traitements à réaliser. Ces traitements étant effectuées par le processeur, les identifier permet d’évaluer **la charge processeur** et donc le temps d’exécution de votre programme.

L’identification des traitements vous aide à construire le code.

L’objectif final de l’analyse du cahier des charges est d’identifier :

* Les données à manipuler
* Les traitements à effectuer
* Les résultats à fournir

Première étape de la méthodologie DTI : le modèle de DONNEES

La première partie, la partie DONNEE, correspond schématiquement à ce que nous venons de faire pour le repérage des noms, i-e une première analyse du CDC pour y repérer les données à traiter et les résultats à fournir. Ces données seront regroupées par « catégories ».

INTERFACE

DONNEES

TRAITEMENTS

Identification des données

Analyse des ressources du CDC (les noms)

Analyse chronologique et descendante du CDC

Cette étape est importante car dans la conception de notre programme nous commencerons par réserver la place mémoire nécessaire au stockage des informations manipulées. On comprendra aisément qu’on ne construira pas de la même façon un programme qui manipule quelques dizaines de données stockées en mémoire vive de l’ordinateur et un programme de « trading » qui consulte des données réparties dans plusieurs bases de données connectées entre elles par les réseaux mondiaux.

Deuxième étape : Les TRAITEMENTS et l’Analyse Chronologique et Descendante du problème

La 2ème partie de l’acronyme DTI, la partie TRAITEMENT, s’intéresse à la logique des actions. Elle mélange 2 phases :

* + 1. Analyse chronologique

Analyse chronologique et descendante du programme

TRAITEMENTS

Maintenant que vous avez repéré et identifié l’ensemble des traitements à exécuter sur l’ensemble des données, il est important d’y mettre de l’ordre. En effet, rien ne dit qu’à la rédaction du cahier des charges ou lors du dialogue avec le client, les idées soient arrivées dans un ordre logique.

**L’analyse chronologique** consiste à définir la succession temporelle logique des étapes à réaliser pour résoudre le problème.

* + 1. Analyse descendante

Après avoir mis les différentes étapes de traitement dans l’ordre, il peut s’avérer nécessaire de décomposer le problème associé à chaque étape en sous problèmes, respectant eux-mêmes un ordre logique, jusqu’à atteindre des taches élémentaires en se mettant à la place de l’utilisateur ou de la machine.

Troisième étape : l’Interface Homme-Machine (IHM) pour les Entrées/Sorties externes

L’Interface, troisième aspect de notre méthodologie s’intéresse aux **Entrées/Sorties externes**.

Si on considère notre programme comme un « système » (nom générique que nous lui donnerons), les Entrées/Sorties externes du système sont :

* **Entrées** : Les informations externes dont le programme a besoin pour s’exécuter
* **Sorties** : Les résultats que le programme fournit

Si on décompose notre système en « sous-systèmes » (que nous appellerons bientôt sous-programmes), les notions d’Entrées/Sorties externes s’appliquent à toutes les informations externes nécessaires aux sous-programmes et qui seront collectées à l’extérieur du système, ainsi que les résultats que ce dernier fournit à l’extérieur du système.

On distinguera donc :

* Les Entrées/Sorties **externes,** données qui sont fournies au programme par l’utilisateur via l’Interface Homme-Machine ou qui sont collectées dans des sources externes (fichiers, bases de données…) et résultats que le programme calcule.
* Les Entrées/Sorties **internes** qui correspondent aux échanges d’informations entre les sous-programmes (paramètres, mécanisme de retour…) à l’intérieur du programme, nécessaires au fonctionnement du système mais non-communiquées à l’extérieur.

L’Interface Homme-Machine (IHM) gère ce que l’utilisateur du programme aura à l’écran. Elle va se préoccuper des données en entrée fournies par l’utilisateur et de la présentation des résultats. Elle va également gérer le design de l’application, la succession des écrans (menu, application, résultats, crédits…). Les parties de code qui gèrent l’interface vont également intégrer les fonctions de contrôle de validité des données externes entrées dans le système avant de les livrer aux traitements qui en ont besoin pour produire les résultats, et ce afin d’éviter de « planter » le programme.

Illustration sur deux exemples

* + 1. Premier exemple : recherche d’un maximum

Le cahier des charges est libellé de la façon suivante : « Chercher et afficher le maximum d’une suite de valeurs saisies par l’utilisateur ».

1ère partie DONNEES

Si nous appliquons notre méthode DTI, la première étape nous permet de nous intéresser à la partie DONNEES en analysant le CDC pour y relever les noms suivants :

* Le maximum,
* Les valeurs,
* L’utilisateur.

Ces noms propres devraient nous aider à déterminer les **ressources « mémoire »** nécessaires au stockage des données à manipuler. Décortiquons-les :

* L’utilisateur sera sollicité par l’IHM pour saisir des informations et pour observer des résultats, ce n’est donc pas une ressource informatique.
* Le maximum, par contre, est une valeur à déduire d’une série de valeurs. Il faudra le stocker en mémoire. Cela correspond donc à une ressource informatique, c'est-à-dire un emplacement en mémoire.
* De même, les valeurs saisies par l’utilisateur devront pouvoir être stockées en mémoire au fur et à mesure de la saisie afin d’être analysées ensuite pour en déterminer le maximum.

On voit ici que pour déterminer les ressources il nous faut faire un tri parmi les éléments mis en évidence par notre méthode, « l’utilisateur » étant faussement détecté comme une ressource mémoire. Rappelons-le, aucune méthode n’est parfaite. Elle apporte un protocole à suivre pour atteindre notre objectif ; mais elle doit s’accompagner de savoir-faire et celui-ci s’acquiert par la pratique.

Notre savoir-faire nous révèle un deuxième problème dans le résultat de cette première analyse. En effet, nous lisons dans le CDC qu’il va falloir stocker « **des** » valeurs. Un certain nombre de questions se pose à propos de ces valeurs :

* **Comment obtenez-vous les valeurs ?** Le CDC suggère deles demander à l’utilisateur !... OK !
* **Combien de valeurs va-t-il falloir stocker ? Peut-on en fixer le nombre ou doit-on permettre une saisie des valeurs une par une avec un moyen pour l’utilisateur de dire « stop » ?** A voir avec le client auteur du CDC ou hypothèse à poser… Le client dit qu’on se limitera à 10… OK !
* **Quelle est la nature de ces valeurs (on dira le « type ») ?** En effet, des entiers ne prennent pas la même place en mémoire que des nombres à virgule… Encore une précision à demander au client… Il veut des nombres à virgule… OK !

Nous avons illustré ici **les imperfections d’un cahier des charges**. Or je vous rappelle que c’est le contrat qui vous lie au client. A vous de ne pas laisser ce contrat contenir des zones de flou qui pourraient se retourner contre vous.

**Une méthode, quelle qu’elle soit, ne peut être efficace si le CDC a mal été rédigé.**

2ème partie TRAITEMENTS

Passons aux traitements et déterminons les actions à effectuer sur nos données.

Les verbes contenus dans le CDC sont « chercher », « afficher » et « saisir ». Les 2 premiers concernent le maximum, le 3ème concerne la suite de valeur.

On pourra synthétiser ces informations dans un tableau de la façon suivante :

|  |  |
| --- | --- |
| **Données** | **actions** |
| Maximum : flottant (nombre à virgule) | Chercher  Afficher |
| Suite de valeurs : 10 flottants | Saisir  Analyser pour trouver le maximum |

Tiens donc ! Une action est apparue pour la suite de valeurs, l’analyse afin de déterminer le maximum.

Tout comme pour les ressources, notre méthode n’est pas non plus infaillible pour les actions. Certaines ressources ou actions ne seront pas mentionnées dans le cahier des charges et devront être déduites de l’objectif à atteindre. Et cela nécessite à nouveau un savoir-faire informatique.

Il s’agit maintenant de déterminer le séquencement des actions pour en respecter la logique et par exemple, ne pas décider d’afficher le maximum avant de l’avoir trouvé !!!

Commençons par mettre de l’ordre, c'est-à-dire par déterminer la « chronologie » des événements.

On obtient :

|  |
| --- |
| **Analyse chronologique** |
| * 1. Obtenir les valeurs * 2. Déterminer la plus grande * 3. Annoncer la plus grande valeur trouvée |

Ensuite, nous pouvons préciser la façon dont nous devrons nous y prendre pour chacune de ces étapes. Pour cela nous allons « préciser » nos traitements (actions). Nous dirons que nous « descendons » en précision dans une « **analyse descendante** ».

|  |
| --- |
| **Analyse descendante** |
| * 1. Obtenir les valeurs en demandant à l’utilisateur de les saisir au clavier * 2. Déterminer la plus grande * 2.1 Comparer la première à la deuxième et conserver la plus grande * 2.2 Comparer la valeur trouvée précédemment à la 3ème et conserver la plus grande * 2.3 Répéter l’opération jusqu’à avoir comparé toutes les valeurs * 3. Annoncer la plus grande valeur trouvée * 3.1 Afficher à l’écran le maximum trouvé |

Nous aurions pu être encore plus précis dans cette analyse descendante, notamment en détaillant la façon dont nous allions obtenir les valeurs en précisant si nous allions demander les 10 d’un coup ou bien au contraire si nous allions les demander une par une.

La précision jusqu’à laquelle nous devons descendre est une notion un peu floue. Il faut éviter de dépasser les 10 étapes pour rester lisible, et éviter également de détailler des actions évidentes. Dans le cas où il vous semblerait nécessaire de préciser en de nombreuses étapes un traitement, pensez à le décomposer en quelques étapes, puis à refaire une analyse chronologique et descendante pour chacune de ces étapes.

3ème partie INTERFACE

Nous devons ici réfléchir à la façon dont nous allons présenter à l’utilisateur ce que nous attendons de lui.

Dans cet exemple, l’utilisateur doit nous fournir des valeurs mais pas n’importe lesquelles : des nombres à virgule qu’on appelle des flottants ; et pas 1 ou 2… mais 10 !

Il est donc nécessaire d’être précis dans notre demande pour que notre utilisateur soit efficace, tout en n’ayant aucune garantie qu’il ne fera pas d’erreur malgré nos consignes (il faudra donc contrôler les Entrées ; c’est une opération qu’on appelle **blindage des saisies**).

On peut donc envisager ici que l’interface consiste à afficher à l’écran le message suivant :

*« Bonjour, veuillez entrer au clavier 10 nombres décimaux séparés par un espace puis valider. »*

Et une fois les traitements de recherche du maximum terminés, l’interface affichera :

*« Le maximum des valeurs saisies est : 25.32 »* (par exemple).

* + 1. Deuxième exemple : calcul du PGCD

Le cahier des charges est cette fois libellé de la manière suivante : **« Calculer le PGCD par la méthode d’Euclide ».** Qu’en pensez-vous ?

1ère partie DONNEES

Ce CDC est assez pauvre. L’application de notre méthode nous permet d’identifier comme ressource mémoire : le PGCD, voire « la méthode d’Euclide ??? » ; et comme action : calculer. C’est peu !

Immédiatement il nous vient de nombreuses questions. La première : qu’est-ce que le PGCD ? La suivante : qu’est-ce que la méthode d’Euclide ? Et lorsque nous commençons à percevoir la logique informatique, les questions suivantes : il faut calculer le pgcd de combien de nombres ? Quelle est la nature (le type) de ces nombres ?...

Encore une fois, tout cela est à faire préciser à l’utilisateur ou, s’il n’est pas accessible, à poser sous forme d’hypothèses clairement spécifiées (les fameuses « specs »).

On considèrera donc qu’il s’agit ici de calculer le Plus Grand Commun Diviseur de 2 entiers.

2ème partie TRAITEMENTS

Notre méthode DTI nous permet cette fois d’identifier (ou de prévoir) les ressources mémoires et les actions associées suivantes :

|  |  |
| --- | --- |
| **Données** | **actions** |
| 2 valeurs A et B : entiers | Saisir |
| pgcd : 1 entier | Calculer  Afficher |

Quant à l’Analyse Chronologique/Descendante, je vous la résume dans le tableau ci-dessous, accompagnées de **la trace d’exécution** dans la colonne de droite.

|  |  |
| --- | --- |
| Obtenir A et B  Calculer la val. abs. de la différence entre A et B  Remplacer le plus grand de A et B par cette différence  Répéter jusqu’à ce qu’A égale B  Afficher résultat | **A B**  24 9  15  15 9…  6 3  3 3 |

3ème partie INTERFACE

Cette étape nous permet de préciser les modalités et l’aspect de **l’Interface Homme/Machine IHM** qui permet la communication du programme avec l’utilisateur, les ENTREES/SORTIES.

On pourra s’intéresser dans cette partie à des considérations esthétiques, fonctionnelles, sécuritaires… On pourra notamment préciser une notion importante : le **blindage des saisies**, qui consiste à contrôler les saisies utilisateurs pour ne pas accepter des entrées qui pourraient mettre en défaut le programme sans pour autant déclencher des erreurs de compilation ou d’exécution.

Par exemple, si vous attendez la saisie d’une note pour un élève, vous pourrez vérifier que cette note est bien entre 0 et 20 et demander une nouvelle saisie en cas d’erreur.

Tant que nous travaillons sur des programmes en mode console, le design ou l’esthétique ne sont pas très élaborés. On s’adresse à l’utilisateur avec un texte écrit en blanc sur un fond d’écran noir (sous Windows), dans une police de caractères unique… Cela ne nous empêche pas, comme pour l’exemple précédent, d’utiliser cet affichage à l’écran pour rappeler les consignes de saisie à l’utilisateur.

Dans le cas de notre PGCD, l’interface va nous permettre de demander les 2 entiers à manipuler puis d’afficher le PGCD des 2.

Retour sur la notion de ressource

Le mot « ressource » cache une notion de coût. Coût mémoire, coût processeur, coût en temps, coût en place… C’est l’analyse du cahier des charges qui va révéler les coûts à minimiser en priorité :

* en temps lorsqu’il s’agit d’écrire le programme qui doit gonfler un air-bag lors d’une collision,
* en place mémoire lorsqu’on doit minimiser la taille des barrettes mémoires pour embarquer un maximum de matériel sur un petit robot d’exploration…

Dans ce cours, nous parlerons principalement de **ressource** lorsqu’il s’agira de **place mémoire** **pour stocker une information.** Afin de définir la taille de la zone mémoire nécessaire au stockage de cette information (ainsi que le principe de codage et décodage ; nous y reviendrons), il faut **décider** le **type** de notre donnée et déclarer une **variable** de stockage. **C’est la déclaration des variables**.

**La notion de variable :** une variable est une ressource matérielle permettant de stocker une information. Cette ressource est prise dans les barrettes mémoire et doit être **dimensionnée et typée**. La notion de type sera détaillée dans les chapitres suivants.

Chaque programme, voire chaque sous-programme commencera systématiquement par la **déclaration des ressources mémoires**.

## Exercices d’application

* + 1. Exercice 201

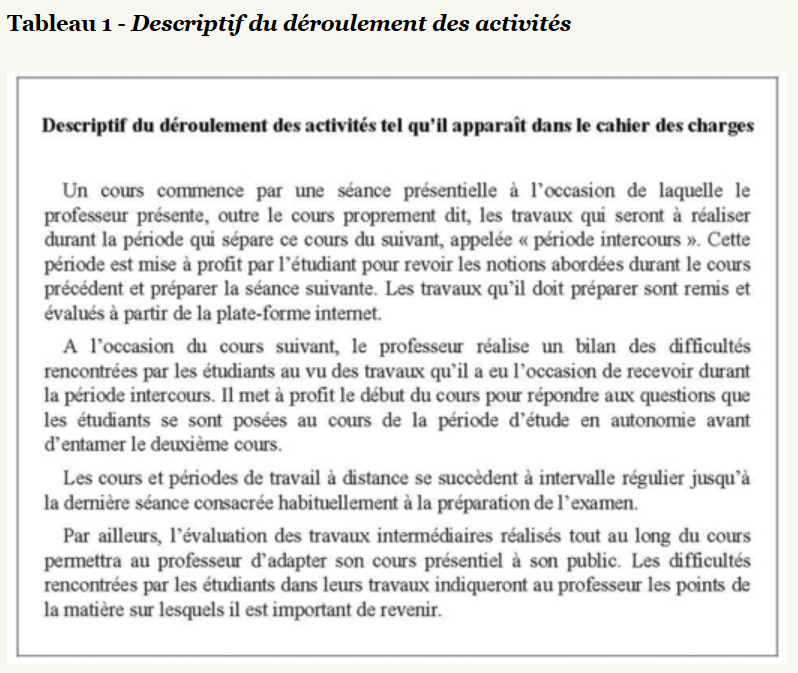
Voici un exemple de morceau de cahier des charges proposé par un directeur d’établissement scolaire qui joue ici le rôle de client.

**Vous devez en étudier toutes les informations que vous aurez éventuellement à stocker dans le système informatique qui permet de gérer l’établissement scolaire.**

En vous positionnant en tant que **maître d’ouvrage**, chargé de la partie **conception de la solution**, définissez les spécifications du projet (les « specs »).

La partie *réalisation*, travail du *maître d’œuvre*, ne sera pas étudiée ici.

En application de la partie « Modèle » de la méthodologie MVC, repérez les noms propres dans l’exemple ci-dessous et identifiez en face de chaque nom les traitements associés.



* + 1. Exercice 202 (Réflexion)

Vous avez devant vous un verre à pied contenant du vin et un verre droit contenant de l’eau.

**But** : échanger le contenu des 2 verres. Comment vous y prenez-vous ?

Situation de départ situation d’arrivée ou résultat

Vous constatez ici que nous ne faisons toujours pas de la programmation à proprement parler. En effet, la première chose à faire est de réfléchir à la façon dont on va s’y prendre : **la méthode**.

C’est la partie la plus importante ; celle qui différencie l’ingénieur du technicien. Le rôle du technicien sera de programmer la solution conçue par l’ingénieur (vision un peu simpliste, certes).

Cette méthode dite « des 2 verres », vous l’utiliserez en programmation. C’est une problématique classique de ressource « cachée ».

* + 1. Exercice 203 (Réflexion – Trace d’exécution)

1. Application de l’exercice précédent : Vous souhaitez échanger le contenu de 2 zones en mémoire, la partie de la barrette mémoire de l’ordinateur où se trouve stockée la valeur de A et celle où se trouve la valeur de B. Comment vous y prenez-vous ?
2. Afin de vérifier la méthode que vous venez d’exprimer en français, faites un dessin représentant toutes vos ressources, dont les 2 zones A et B, par des rectangles. Ecrivez des valeurs dedans puis échangez les contenus en respectant étape par étape votre méthode et en rayant le contenu de chaque zone à chaque changement pour écrire la nouvelle valeur **dessous** sans rien effacer. On appelle ce travail **une trace d’exécution**.

Exemple :

A B

~~?~~ ~~?~~ Le point d’interrogation symbolise le fait qu’avant d’y mettre

~~24~~ ~~9~~ une valeur, une zone mémoire contient… **n’importe quoi (mais pas RIEN)**.

…

# Chapitre 3 - Conception (suite) : Algorithme et Organigramme

## Algorithme

Continuité de l’Analyse Chronologique Descendante ACD, un **algorithme** correspond à l’élaboration, dans un pseudo langage proche du langage humain, d’une succession d’actions à exécuter par le processeur permettant d’atteindre un objectif donné.

**Charge mémoire (place)** et **charge processeur (temps d’exécution)** sont les 2 critères permettant de déterminer l’efficacité de votre solution.

* + 1. Conception de l’algorithme

Pour concevoir l’algorithme, nous allons reprendre une à une les taches élémentaires issues de **l’analyse chronologique descendante** réalisée dans la phase conceptuelle (phase qu’on appelle aussi la maitrise d’ouvrage) et écrire dans un pseudo-langage la succession d’instructions à faire exécuter par la machine pour réaliser chaque tache.

Cet algorithme sera commenté pour en expliquer les étapes complexes et le rendre compréhensible.

Nous respecterons une présentation rigoureuse (syntaxe et indentation) et choisirons des noms significatifs aux variables (valeur, moyenne, âge… plutôt que x, y, z…) lors des déclarations des zones de stockage des données.

* + 1. Avantages

Le pseudo code obtenu respecte la logique informatique tout en restant indépendant du langage **d’implémentation** (de programmation) final. La réalisation du programme ne sera plus qu’une étape de traduction dans le langage informatique que vous choisirez pour ses performances (cette année il s’agira du C).

* + 1. Exemple : Algorithme de calcul du PGCD par la méthode d’Euclide

|  |
| --- |
| Algorithme PGCD-Euclide  DEBUT  // declaration des variables  variables  A, B, Diff  : entiers  // obtention des valeurs A et B  Ecrire(″entrez la valeur de A″)  Lire(A)  Ecrire(″entrez la valeur de B″)  Lire(B)  // boucle de traitement  Tant que (A<>B) faire  Début  Diff ← absolu(A – B)  Si (A>B) alors A ←Diff  Sinon B ←Diff  Fin  // Affichage du resultat  Ecrire(″Le PGCD est ″,A)  FIN |

Les traits verticaux représentent les niveaux **d’indentation** (décalage à droite). Ils caractérisent les « blocs » ou « niveaux » d’instructions.

Vous remarquerez que cet algorithme ne respecte pas tout à fait notre modèle. En effet, la variable PGDC n’apparait pas. Mais il fait apparaitre une variable Diff. Pourquoi d’après vous ?

## Organigramme

Établir l'**organigramme**, c'est traduire l'algorithme sous forme schématique. Cette modélisation schématique est efficace et sera souvent reprise en électronique. Elle peut néanmoins paraitre un peu moins complète que l’algorithme, notamment dans l’absence de commentaire.

La symbolique utilisée dans les organigrammes est variable d’un domaine d’application à un autre. Nous utiliserons cette année une symbolique très simple composée de 4 symboles et de flèches.

|  |  |
| --- | --- |
| debutfin | Le **cartouche** symbolise le **début et la fin du bloc de traitement** |
| traitement | Le **rectangle** correspond à un **traitement**, c'est-à-dire une instruction ou une séquence d’instructions identifiée par l’analyse chronologique descendante |
| alternativealternative  FAUX  VRAI  Si (a<B) | Le **losange** représente un **test** (une **alternative)** qui, suivant le résultat du test traitera l'information suivant une voie ou l'autre. Nous entrerons dans le test par une des branches arrivant sur le losange et indiquerons sur les 2 autres branches le chemin correspondant à un test VRAI et celui pour un test FAUX. |
| Tant que (x<y)  Vrai  traitement 1  traitement 2  Faux | Les **flèches** symbolisent les **boucles** (ou **répétitives)** en revenant en arrière sur l’organigramme. Elles permettent la répétition d’un traitement (suite d'instructions écrites dans le(s) rectangle(s)) un nombre déterminé de fois ou encore tant qu’une condition est vérifiée.  Le **nombre de tours de boucle** ou la **condition de répétition** sera mentionné dans la forme particulière figurant ci-contre. |

## Exemple du PGCD

|  |
| --- |
| Début  Fin  Obtenir A et B  Diff 🡨 abs(A – B)  A 🡨 Diff  B 🡨 Diff  si A > B  vrai  faux  Ressources :  Diff, A et B : 3 entiers  Afficher « Le Pgcd est : »  Afficher la valeur de la variable A  Tant que A != B  vrai  faux |

Tout algorithme, aussi compliqué soit-il, peut se traduire à l'aide d’un organigramme. Lorsque ce dernier devient trop volumineux, on **décomposera** l’organigramme en plusieurs parties, de la plus générale à la plus précise. On retrouvera ainsi la notion d’analyse descendante (descendante en précision). Un traitement grossier mentionné dans un rectangle de l’organigramme général, par exemple « obtenir A et B » pourra donner lieu à un autre organigramme plus précis décrivant comment s’y prendre.

La solution complète correspondra donc à un ensemble d’algorithmes ou d’organigrammes, chacun étant la description détaillée du fonctionnement d’une « **boîte noire** » **dédiée à une tache particulière**. On approche déjà la notion de **programmation structurée** (et de sous-programme), voire la notion d’objet.

Cas d’utilisations, jeux d’essai, tests et traces d’exécution

Chacune des boites noires, chaque partie de l’organigramme ou de l’algorithme sera soumise à une phase de test pour en valider l’efficacité. Cette phase de test revient à **faire tourner le programme à la main** en jouant le rôle de l’ordinateur lorsqu’il exécute le code.

* + 1. Cas d’utilisation et jeux d’essai

Les cas d’utilisation (les « use case » en anglais) sont tous les cas de figure qui peuvent se présenter à l’utilisateur à l’exécution du programme. Les programmeurs débutants se contentent souvent des cas d’utilisation standards correspondant à la situation « **idéale** » d’utilisation de leur programme. Ils oublient que :

* L’utilisateur peut découvrir le programme pour la première fois et avoir besoin d’être un peu guidé,
* L’utilisateur peut être habitué au programme et ne pas avoir envie ni envie d’être systématiquement guidé,
* L’utilisateur peut faire des erreurs de saisies ou ne pas respecter les conditions idéales d’utilisation du programme…

Par conséquent, lors de la définition des jeux d’essai, vous ne devez pas vous contenter des situations standards et penser à tester les « **cas limites** ».

* + 1. Trace d’exécution

Exécuter à la main chacun des morceaux du code rédigé en pseudo langage algorithmique ou en organigramme permet de suivre l’évolution du contenu des différentes variables de stockage des informations et de s’assurer qu’on obtient bien le résultat recherché à la fin.

Toutes les variables seront listées les unes à côté des autres sur une même ligne. Un « ? » sera indiqué sous chacune d’elle pour symboliser le contenu incertain et **inutilisable** d’une variable **non-initialisée**, état de toute variable au moment de sa déclaration.

Au fur et à mesure de l’exécution de l’algorithme à la main, le contenu des variables variera en fonction des **affectations**. La nouvelle valeur d’une variable vient remplacer la précédente en rayant cette dernière et en se plaçant en dessous, **sans jamais rien effacer**.

Voici l’exemple d’une trace d’exécution de l’algorithme du PGCD. On suppose ici que l’utilisateur entre les valeurs 24 et 9.

|  |
| --- |
| A B Diff  Déclaration 🡪 ~~?~~ ~~?~~ ~~?~~  ~~24~~ ~~9~~ ~~15~~  ~~15~~ ~~9~~ ~~6~~  ~~6~~ ~~3~~ 3  3 3 |

## Exercices d’application

POUR VOUS ENTRAINER

**Analyse chronologique et descendante**

Au chapitre précédent, nous exprimions en français tout ce que notre analyse des besoins du client pouvait révéler. Cette fois-ci, formalisons un peu notre pensée et mettons un peu d’ordre et de logique dans tout ça.

Pour les premiers exercices, rédigez l’analyse chronologique et descendante des problèmes qui vous sont proposés. Vous ne dépasserez pas 10 étapes élémentaires dans vos analyses.

Vous préciserez, pour chaque exercice les variables (ressources) qui vous semblent nécessaires pour stocker les informations et les éléments de calcul en précisant leur type (entier, réel, texte…).

Enfin, dans les derniers exercices, réalisez également l’algorithme ou l’organigramme correspondant à votre analyse descendante.

**Les différentes étapes de votre analyse chronologique et descendante doivent vous servir de trame et être reprises comme commentaires dans votre algorithme.**

* + 1. Exercice 301 Données/Traitements/Interface

En application de la première partie de la méthode DTI, identifiez et présentez sous forme de tableau (cf cours) les données, les traitements sur ces données et les résultats à fournir dans le cahier des charges suivant :

*« Ecrire un programme qui permettent la saisie des notes obtenues par un élève ING1 et qui applique précisément le processus d’évaluation pour déterminer si cet élève est admis en année supérieure ou pas ».*

* + 1. Exercice 302 DTI - Analyse Chronologique Descendante

Réalisez la partie DONNEES (comme précédemment) puis la partie TRAITEMENTS en aboutissant à l’Analyse Chronologique Descendante (ACD) décrivant la programmation d’un **automate distributeur de boissons.**

**Hypothèses** : On considèrera que l’automate dispose de boissons en nombre suffisant. Donc pas de gestion de stock ni de boissons ni de pièces.

* + 1. Exercice 303 (DTI/ACD+ALGO+ORGA)

Ecrivez le résultat de DTI avec l’ACD, puis un **algorithme** et un **organigramme** d’un programme qui demande deux entiers et **indique lequel est le plus petit, lequel est le plus grand.**

* + 1. Exercice 304 DTI – ACD + ALGO

Appliquez la méthode DONNEES/TRAITEMENTS/INTERFACE en allant jusqu’àl’ACD puis donnez l’algorithme décrivant la programmation d’une **recherche du minimum de plusieurs valeurs.**

Vous fournirez un algorithme (inspirez-vous de l’exemple du PGCD vu en cours) dans lequel votre ACD sera reprise sous forme de commentaire (rappel : une ligne de commentaire commence par // dans un algorithme ou un programme en C).

* + 1. Exercice 305 (DTI/ACD+ALGO+ORGA)

Ecrivez le résultat de DTI avec l’ACD, puis un algorithme et un organigramme d’un programme qui demande un entier et indique s’il est pair ou impair.

* + 1. Exercice 306 (ACD+ALGO/ORGA+TRACE)

**Problème** : trier 5 entiers.

En plus du DTI avec l’ACD, écrivez l’algorithme **ou** l’organigramme et faites tourner votre programme à la main en listant les ressources et en fournissant la **trace d’exécution** simulant l’exécution de votre code.

Ex : nb1 nb2 nb3 nb4 nb5

4 ~~8~~ ~~2~~ …

4 2 8 la trace d’exécution montre l’évolution des variables

2…

* + 1. Exercice 307 (ACD+ORGA)

Ecrivez le DTI avec ACD, puis l’organigramme décrivant la programmation d’un automate distributeur de billets de banque.

**Hypothèses** : En l’absence, à vous de poser les vôtres.

# Chapitre 4 - Phase de Réalisation : le Code

Programmation en langage dit « de haut niveau »

Après l’étape de test de la solution proposée sous forme d’algorithme ou d’organigramme, vous savez que la solution envisagée apporte une solution au problème, en ayant imaginé tous les cas de figure (les « use cases »). En tous cas, sur le papier !

Il reste à vérifier que la machine va savoir exécuter les instructions attendues.

Pour cela, il faut en faire une **traduction** dans un **langage de programmation** spécifique. Cette traduction est une étape nommée « codage » qu’on réalise à l’aide d’un éditeur (nous utiliserons celui de l’IDE (Intergrated Development Environment) CodeBlocks). Le code créé sera soumis à **l’interpréteur** ou le **compilateur** qui sont des logiciels implantés dans votre PC. Ces programmes installés dans votre "machine" traduiront à leur tour vos instructions en **« langage machine »** (par exemple l’Assembleur) pour qu'elles soient exécutées. On trouve également un compilateur dans l’IDE CodeBlocks.

Le schéma ci-dessous retrace les différentes étapes et acteurs d’une solution complète.

**Client**

Spécifications (Cahier des charges)

Algorithme ou Organigramme

Code en langage de programmation

Langage machine (Assembleur)

Exécution

**L’IDE (interface de développement comme CodeBlocks)**

* **Edition de texte**
* **Compilation**
* **Liens Objet**
* **Exécution**

**Vous**

***Travail sur papier***

**Vous**

***Sur machine***

Pour réaliser la transcription du pseudo code dans un langage choisi pour ses spécificités, en respectant la syntaxe associée, vous pouvez utiliser un **éditeur** de texte genre « bloc note » pour taper votre traduction en code de haut niveau comme le langage C, puis soumettre ce code à un **compilateur** comme le célèbre logiciel GCC (GNU Compilateur C) puis **lancer** en double-cliquant sur son icone le fichier .exe que GCC aura généré... Ouf !!!

Vous pouvez également utiliser une interface de développement « toute prête » qui intègre chacun de ces rôles (éditeur, compilateur et exécuteur) comme le logiciel **CodeBlocks** (logiciel libre) ou encore les outils Visuals (payants).

* + 1. Le langage choisi pour la première année : Le C… Historique

Choisi pour ses vertus pédagogiques, le langage C est un langage puissant qui offre des possibilités dans de nombreux domaines comme le graphisme, le traitement d’images, le web, mais aussi la programmation système ou l’électronique embarquée et la robotique.

Langage à la fois de haut niveau et de bas niveau créé dans les années 70 par B.W. Kernighan et D.M. Ritchie pour le développement du système UNIX par les laboratoires BELL (prédécesseur : le langage B - 1970).

Nombreuses normalisations depuis : Norme IEEE X3-J11 (C natif), ou encore norme ANSI.

**Haut niveau** : Langage structuré (sous programmes, structures de contrôle, structures de données).

**Bas niveau** : Opérateurs permettant de manipuler les instructions et les données au niveau du langage-machine.

* + 1. Edition d’un programme C

Le travail préliminaire de réflexion sur papier étant terminé, détaillons un peu la première étape sur machine. « **Editer** » un programme veut dire « écrire » un programme, par référence aux éditeurs de texte bloc‑note, word, VI sous UNIX… Cette rédaction doit suivre un ordre et une syntaxe **définis par le langage choisi**. Par exemple en C vous respecterez l’ordre suivant : Bibliothèques d’abord, puis sous-programmes ; et dans chacun des sous-programmes : déclaration des variables d’abord, puis code.

Les bibliothèques

Elles contiennent les fonctionnalités livrées avec le langage que vous n’aurez ainsi pas besoin d’écrire, même si rien ne vous empêche de réécrire toutes les fonctionnalités préexistantes.

Il y en a des centaines ! Parcourez le répertoire « include » du système (/usr/include sur Linux, ou sous-répertoire du lieu d’installation de CodeBlocks sous windows) pour voir la multitude de fichiers thématiques qui s’y trouvent.

Même si les noms des fichiers sont souvent explicites (math.h, file.h, stdio.h…), bon courage pour y trouver la fonctionnalité dont vous avez besoin.

Les bibliothèques que vous aurez choisies seront précédées d’un # qui stipule qu’elles doivent être rapatriées dans votre code par le pré-compilateur. Nous y reviendrons.

Décomposition du corps en procédures ou fonctions thématiques

Plutôt que de coder un programme contenant plusieurs centaines de lignes, il est préférable de structurer son code en le décomposant en plusieurs parties assimilables à des « boites noires » dédiées chacune à un traitement distinct. Ces différents traitements seront pour l’instant séparés par des commentaires mais ils pourront devenir des sous-programmes qu’on appelle, en C, des procédures ou des fonctions. Nous découvrirons cela dans les chapitres suivants.

Le programme dit « principal » (ou « main » en C)

Le main est le **point d’entrée** dans l’exécution de votre programme (on dit que le système d’exploitation « appelle » le main). Il joue le rôle de chef d’orchestre et appelle dans l’ordre prévu les différents sous-programmes qui participent à l’exécution du programme.

**Premier programme : « Bonjour le monde ! »**

mabib.h (bibliothèque personnelle)

#ifndef BIB

#define BIB

#include <stdio.h>

#endif

monfic.c

|  |
| --- |
| #include ″mabib.h″  int main()  {  // ce programme affiche un message a l’ecran  printf(″Bonjour le monde !″) ;  system(« pause ») ;  return 0 ;  } |

**Autre exemple : Reprenons notre PGCD.**

|  |
| --- |
| // on commence par les bibliothèques  #include <stdio.h>  #include “mabib.h” // notez la différence de syntaxe ligne 2 et 3  // ici les fonctions et procedures éventuelles  …  // et le programme principal  int main()  {  // declaration des variables  int a,b,diff ;    // obtention des 2 valeurs  printf(″entrez la valeur de A″) ;  scanf(″%d″,&a) ;  printf(″entrez la valeur de B″) ;  scanf(″%d″,&b) ;  // boucle de calcul du pgcd  while (a != b)  {  diff = abs(a – b) ;  if (a>b) a = diff ;  else b = diff ;  }  // affichage du resultat  printf(″Le PGCD est %d\n″,a) ;  return 0 ;  } |

Vous noterez en C :

* Les débuts et fins de bloc sont des accolades
* Les instructions sont terminées par des points-virgules.
* Les instructions d’un même bloc sont décalées à droite par rapport aux accolades du bloc, on appelle ça **l’indentation du programme**. Elle n’est utile qu’au codeur mais permet une relecture efficace du code. Elle s’obtient très simplement sous CodeBlocks en cliquant dans le menu Plugins/SourceCode Formater.
  + 1. Les instructions

Instruction simple

Une instruction correspond à une action à faire exécuter par la machine. Elle coïncide en général avec une ligne de code, bien qu’il puisse y avoir plusieurs instructions sur une seule ligne.

Les instructions sont de nature différente en fonction du but recherché : Echange avec l’utilisateur, déclaration d’une variable, attribution d’une valeur à une variable, test sur cette valeur, répétition d’une série de taches, appel de sous-programme…

**Exemple** : afficher\_a\_l\_ecran(« bonjour ») ou encore printf(« bonjour ») ;

Bloc d’instructions

Un bloc d’instructions correspond à un ensemble d’instructions délimité par les termes **DEBUT et FIN** en algo, **BEGIN et END** en Pascal ou **{ et }** en C. Un programme est donc, en soi, un bloc d’instructions.

La notion de bloc permet de traiter un ensemble donné d’instructions en cas de réponse positive d’un test ou un autre bloc dans le cas négatif. Elle permet également de délimiter les instructions à répéter dans le cadre d’une boucle.

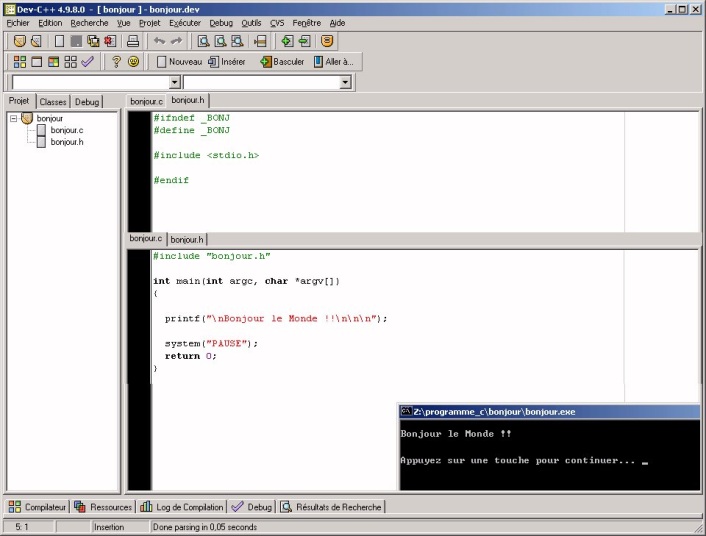
* + 1. Les commentaires

Les commentaires accompagnent chaque partie « logique » du code. Ils sont indispensables pour la bonne compréhension du programme…

* Par les différents membres de l’équipe de développement
* Par les correcteurs
* Par tout programmeur souhaitant s’inspirer d’un code déjà écrit
* **Par le programmeur du code lui-même, lorsqu’il reprend un code écrit il y a quelques temps**

**Ils seront automatiquement extraits par certains logiciels pour réaliser une documentation automatique.**

* + 1. Un exemple sous CodeBlocks



L’éditeur CodeBlocks, permet de paramétrer l’écran pour afficher 3 parties : l’arborescence de fichiers composants le projet, une fenêtre d’édition et une zone d’affichage des erreurs.

* + 1. Compilation ou Interprétation

Certains langages de programmation sont compilés (comme le C), d’autres sont interprétés (comme le Basic). Dans le cas des langages interprétés, chaque instruction jugée valide est aussitôt exécutée, tandis que dans le cas des langages compilés la vérification est globale sur l’ensemble du code avant que ce dernier ne soit exécuté.

Vérification syntaxique

D’abord visuelle de la part du programmeur, la vérification syntaxique est la première phase du compilateur. Elle consiste à contrôler le respect des règles d’écriture (la syntaxe) du langage choisi.

En C, attention aux « ; », au « = » (« prend la valeur » pour l’affectation) et au « == » (pour l’égalité).

Vérification des variables

La deuxième phase de compilation consiste a contrôler vos ressources afin de vérifier que vous n’avez pas utilisé de caractère interdit dans leur dénomination et de s’assurer que vous n’utilisez pas de ressources que vous auriez oublié de déclarer.

Vérification des instructions

Les contrôles sont multiples ici. Entre autres, le compilateur vérifie que les fonctionnalités que vous utilisez existent bien dans les bibliothèques que vous avez incluses. Il contrôle également les appels de sous-programmes.

* + 1. Création du fichier « objet » .obj en compilation séparée

Cette phase ne concerne que les langages compilés. Elle permet de faire de la **compilation séparée**, c'est-à-dire d’écrire un programme composé de plusieurs fichiers .c qui seront analysés chacun séparément par la machine (le compilateur) pour en valider le fonctionnement, puis transformés chacun en un fichier .o (sous Linux) ou .obj (sous Windows), format intermédiaire en langage machine genre assembleur, avant regroupement de l’ensemble sous un unique exécutable (extension .exe sous Windows, sans sous Linux) qui constituera le logiciel à vendre. En général, on ne vend pas les sources.

* + 1. Création du fichier « exécutable » : Edition des liens

Dernière étape vers le fichier exécutable. Il s’agit ici de fusionner les fichiers .o en un seul et même fichier en langage machine. C’est ce fichier qui sera vendu ou fourni au client.

Voici une illustration schématique du processus de compilation et d’édition des liens pour créer l’exécutable.

|  |
| --- |
| COMPIL |

Dernière étape : Définition du protocole de test d’exécution

Après les nombreux tests de l’algorithme sur papier avec traces d’exécution, c’est la seconde phase de test, voire la troisième ou la quatrième… Vous ne ferez jamais trop de tests. Après avoir validé votre méthode sur papier, après avoir fait tourner à la main chacune des boites noires (sous-programmes) qui composent le code, vous voici en possession de l’exécutable final et il faut maintenant s’assurer qu’il répond bien aux attentes du client.

Gros défaut des concepteurs : ils sont toujours persuadés d’avoir pensé à tout.

**Grave erreur !** Vous ne pensez souvent qu’aux situations standards. Le problème ne viendra évidemment pas des cas auxquels vous aurez pensé, mais plutôt de la petite exception qui n’arrive jamais… ou presque. Alors, **pensez aux cas limites**.

Reprenez tous les « use cases » ou « cas d’utilisation » que vous avez envisager lors du test de votre algorithme et soumettez-les à votre programme. Si les résultats obtenus sont identiques à ceux attendus « sur le papier », vous êtes en très bonne voie ☺.

Qu’est-ce qu’un « bon » résultat ?

Bien souvent, toute l’énergie de l’équipe de développement est consacrée à l’obtention du programme le plus performant, quitte à y passer la nuit la veille de la livraison au client.

Or un **bon** résultat ce n’est pas la solution la plus performante !!

**Un bon résultat c’est :**

* le résultat de la meilleure organisation d’équipe,
* une équipe où le « leader » est clairement identifié (il sera utile),
* une équipe où chacun a pu exprimer ses opinions et où les décisions ont été prises en concertation, sous la direction du « leader »
* une solution élaborée **après** une analyse performante et efficace,
* un découpage en morceau de la réalisation avec des **tests** de chacune des parties (inutiles de passer à la suite tant que la partie précédente n’est pas efficacement développée, sauf en cas de parallélisme),
* un respect du cahier des charges sous peine de sanction de la part du commanditaire,
* un respect du budget,
* un bon compromis charge mémoire/charge processeur, adapté aux exigences du CDC.

Un **bon résultat,** c’est celui qu’on obtient grâce à une **bonne conduite de projet.**

## Exercices d’application

Ce chapitre est une présentation de la méthodologie complète de la réalisation d’un programme en langage informatique, depuis l’écriture du code jusqu’à son exécution et ses tests.

Il ne donnera pas lieu à des exercices dans ce chapitre.

# Chapitre 5 - L’ordinateur : Rôle et fonctionnement

Pour aller plus loin et découvrir des notions théoriques plus complexes, il nous faut maintenant parler la même langue, enfin… disons plutôt le même langage.

L’idée ici, c’est d’acquérir un vocabulaire commun, de comprendre un peu mieux la terminologie utilisée par les « informaticiens », de comprendre le fonctionnement d’un ordinateur et de réaliser que tout part du binaire « 0/1 », « oui/non », « vrai/faux » …

Composition d’un ordinateur : le Hardware

* + 1. Extérieur : l’Unité Centrale (UC) et ses périphériques d’Entrée/Sortie

|  |  |  |
| --- | --- | --- |
| ordi-complet | [http://tse1.mm.bing.net/th?&id=OIP.Me6e80a3ad10ac44f1ae1e9f1d20d7691o0&w=300&h=298&c=0&pid=1.9&rs=0&p=0](http://www.bing.com/images/search?q=smartphone&view=detailv2&&id=968F91B61A92A6B714C814D9EE75427B0F9B6D0C&selectedIndex=12&ccid=5ugKOtEK&simid=608016745750201024&thid=OIP.Me6e80a3ad10ac44f1ae1e9f1d20d7691o0) | [Résultat d’images pour ordi portable](http://www.bing.com/images/search?q=ordi+portable&view=detailv2&&id=19C71AAECB441862AB0490CDEE33EDD8405E5D93&selectedIndex=16&ccid=PB3ALlOA&simid=608029707962942070&thid=OIP.M3c1dc02e5380d19dd0ca49bd1dfdb2cfH0) |

3 catégories d’ordinateur à part entière

* + 1. Intérieur : les constituants de l’Unité Centrale : Mémoire vive, mémoire de masse, processeur…

|  |  |
| --- | --- |
| carte_mere-autre | proc-dessous |

Que savez-vous de…

* La communication entre les différents organes.
* Les unités employées (Mhz, Méga-octets…).
* Les performances actuelles.

|  |
| --- |
|  |

Rôle de l’ordinateur : manipuler de l’information

* Obtenir et stocker de l’information, des données
* Effectuer des traitements sur ces données
* Permettre la visualisation et l’exploitation de résultats

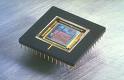
STOCKAGE DES DONNEES

TRAITEMENT DES DONNEES

Bus

[](http://images.google.fr/imgres?imgurl=http://www.bestofpc.com/v3/images/stories/bestofpc/barette_memoire.jpg&imgrefurl=http://www.bestofpc.com/v3/index.php?view=article&catid=1:latest-news&id=59:acheter-un-ordinateur-portable&tmpl=component&print=1&page=&h=413&w=550&sz=31&hl=fr&start=1&um=1&usg=__eZfqu16xy7kHDZJdpAcP7T5ypBI=&tbnid=vz6ifsBZwRYdDM:&tbnh=100&tbnw=133&prev=/images?q=barette+memoire&um=1&hl=fr&sa=N)

[](http://images.google.fr/imgres?imgurl=http://www.l-ordinateur.com/camilleinfo/114115.gif&imgrefurl=http://www.l-ordinateur.com/uc.html&h=600&w=800&sz=147&hl=fr&start=8&um=1&usg=__UZnxbd5UiV95sUHJHOGKTORLY0Q=&tbnid=zztrTvfYfiwsgM:&tbnh=107&tbnw=143&prev=/images?q=bus+de+donnees&um=1&hl=fr)

[](http://images.google.fr/imgres?imgurl=http://blog.concept-web.ch/images/processeur.jpg&imgrefurl=http://blog.concept-web.ch/index.php?2006/08/20/17-la-course-a-la-vitesse-engendre-des-puces-superpuissantes&h=259&w=400&sz=109&hl=fr&start=3&um=1&usg=__QlakuXqYpS2dM9A98ZiaAE0xX9k=&tbnid=ikgJ6eaiq1vPWM:&tbnh=80&tbnw=124&prev=/images?q=processeur&um=1&hl=fr)

* + 1. La communication homme-machine : De l’électricité à l’homme : l’empilement des couches systèmes et logicielles

Les couches

Peter

Pietro

Bob

Ali

Maria

Rapport 2

BIOS

WINDOWS

Word

Système d’exploitation

Programmes, applications

Winamp

MSN

Rapport 1

Benabar

Pierre

Paul

Documents

* + 1. La mémoire vive et le processeur : le ying et le yang

Vous l’aurez compris, les deux principales fonctions de l’ordinateur sont le **stockage** et le **traitement** des données. Sans minimiser le rôle des autres constituants, les deux organes qui vont retenir notre attention sont donc la **mémoire** et le **processeur** que l’on pourra respectivement associer à des préoccupations de **place** et de **vitesse**.

C’est leur capacité intrinsèque qui va déterminer les limites de la solution à mettre en œuvre. Il sera donc toujours intéressant de prendre connaissance de la configuration matérielle sur laquelle le client souhaite exploiter la solution qui lui sera proposée. Nous ne construirons pas de la même façon un programme devant gonfler un airbag en cas de choc qu’un programme devant guider un robot miniature.

* + 1. Transfert des données mémoire 🡪 processeur : les bus

Organe à ne pas négliger, les bus transfèrent l’information d’une partie à l’autre de la machine. Par analogie, vous comprendrez sans difficulté qu’il est plus efficace de faire transiter beaucoup de monde dans un avion rapide qu’une personne à la fois dans une barque à rames. C’est pareil dans un ordinateur. A quoi sert d’avoir des mémoires super performantes et un processeur ultra rapide si les informations circulent entre les deux sur des bus inefficaces ?

Dans un système, quel qu’il soit, c’est le maillon le plus faible qui détermine la puissance ou la capacité générale. Il vaut donc mieux privilégier une configuration équilibrée que des parties très performantes côtoyant des parties obsolètes.

Stockage des données : mémoire vive, morte et mémoire de masse

* + 1. Les données

Commençons par revenir sur une notion importante pour ce cours de programmation, la notion de **donnée**.

Une donnée est un nom générique pour traduire toute information utilisée par l’ordinateur :

* Une information saisie par l’utilisateur (ou Data)
* Une information qui provient du réseau, d’internet
* Un programme qui a déjà été installé dans l’ordinateur
* …

**Remarque** : le fait que les programmes soient eux-aussi considérés comme « données » est très important !

* + 1. Les mémoires

Pour que votre ordinateur puisse manipuler des **données** (informations mais aussi programmes, je vous le rappelle), il doit au préalable les **stocker**. On distinguera le stockage dit **permanent** qui conserve l’information même après extinction de la machine, du stockage dit **volatile** qui contient temporairement les données à traiter et se vide lorsqu’il n’est plus alimenté.

Seules les informations de la mémoire volatile ou **vive** peuvent être traitées. Elles doivent donc éventuellement être copiées de la mémoire permanente vers la mémoire vive (ou saisies directement par l’IHM vers la mémoire vive) avant le traitement par le processeur.

Il y a plusieurs types de mémoire :

* La mémoire vive (ou Random Acess Memory – RAM) qui est volatile
* Les mémoires de masse (les disques durs à plateaux magnétiques, mémoires flash SSD et clefs USB, supports optiques CD, DVD…) toujours persistants, réinscriptibles ou non.

[](http://fr.wikipedia.org/wiki/Image:RAM_n.jpg)**La mémoire vive** est constituée par les barrettes mémoires. Elles mettent à disposition de l’ordinateur une unité de stockage élémentaire appelée **octet** (ou Byte en anglais, à ne pas confondre avec les bits, sous-unité de l’octet : 1 octet (byte) = 8 bits). Une barrette sera donc caractérisée par sa capacité de stockage, par exemple 1 Go (Giga octet) (mais également par sa vitesse d’accès, son mode d’accès, etc.).

Pour en permettre l’accès, chaque octet est numéroté. Une barrette de 1 Go offre donc des **octets numérotés de 0 à 1 milliard**.

*Différents types de RAM, de haut en bas :*

[*DIP*](http://fr.wikipedia.org/wiki/Dual_Inline_Package)*,* [*SIP*](http://fr.wikipedia.org/wiki/Single_Inline_Package)*,* [*SIMM*](http://fr.wikipedia.org/wiki/SIMM) *30 broches,* [*SIMM*](http://fr.wikipedia.org/wiki/SIMM) *72 broches,* [*DIMM*](http://fr.wikipedia.org/wiki/DIMM)*,* [*RIMM*](http://fr.wikipedia.org/wiki/RIMM) *(Wikipédia)*

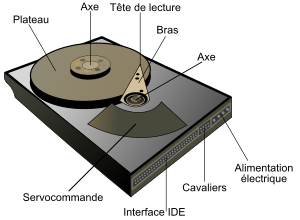
Une donnée peut être stockée dans un ou plusieurs octets en fonction de sa taille, les octets étant indivisibles. L’ensemble des octets occupés par une donnée s’appelle **une variable**. Le numéro du premier octet occupé par cette variable constituera **l’adresse** **de la variable**.

Une variable sera donc identifiée par la machine par son adresse, sa taille en octets et la nature du contenu (taille et nature étant définis par ce qu’on appelle son **type**).

Encore une bonne dose de vocabulaire, hein !?

La notion de **donnée** regroupant également les **programmes**, ces derniers doivent également être stockés dans la mémoire vive de l’ordinateur pour être utilisés. Lorsque vous « lancez » (exécutez) un programme installé sur votre ordinateur, celui-ci est « monté » dans la mémoire vive de l’ordinateur. La limite de capacité de la mémoire vive (rappelez-vous, 1Go dans mon exemple) explique que, même si vous avez installé des centaines de programmes dans votre machine, vous êtes limité dans le nombre de programmes que vous pouvez **exécuter simultanément**.

Après avoir ouvert Word sur 2 ou 3 documents, lancé un programme pour écouter de la musique, ouvert internet pour faire des recherches et évidemment MSN pour chatter avec 5 ou 6 copains tout en matant une vidéo sur Youtube…vous risquez d’approcher les limites de la mémoire vive de votre ordinateur (sans parler du processeur) et de déclencher les mécanismes de **swap** qui ralentissent considérablement l’ordinateur avant de geler définitivement tout l’ensemble.

[](http://fr.wikipedia.org/wiki/Image:Hard_drive-fr.svg)**Enfin, la mémoire de masse** : disques durs (à plateau magnétique ou en mémoire Flash SSD (Solid State Drive)), clef USB, CD, DVD... Ils constituent une zone de stockage permanent par opposition à la mémoire centrale (ou vive), volatile. C’est la zone de stockage des programmes, musiques, photos, vidéos, jeux… que vous installez et souhaitez conserver dans votre ordinateur, même lorsqu’ils ne sont pas en cours d’utilisation.

[](http://fr.wikipedia.org/wiki/Image:Usbdrive_icon.svg)[](http://www.bing.com/images/search?q=memoire+ssd&view=detailv2&&id=EAA65C62458DC78CCF5E465F182A2B8E78A6667B&selectedIndex=0&ccid=bc88wzgW&simid=607986152700642340&thid=OIP.M6dcf3cc33816759a194ac4f9a9f920b7o0)

*Clef USB, SSD et schéma d’un disque dur (Wikipédia)*

* + 1. Retour sur les variables

On n’a pas fini d’en parler, tellement cette notion est **importante et complexe**.

Nous avons vu que la mémoire stockait toutes sortes d’informations : des programmes et des données.

*Ah bon ? Mais je croyais que les programmes étaient aussi des données ? Je ne comprends plus rien !...*

Tout dépend du point de vue où on se situe. **Pour un ordinateur**, un programme est une donnée, une information est une donnée, etc. **Pour un programme**, un traitement est une instruction et les instructions manipulent des données (ou data, ou entrée…) stockées dans des variables (précisant ainsi leur type, leur taille et leur principe de codage).

Je vous ai dit que le plus gros problème c’est le vocabulaire employé !

* + 1. La FAT (File Allocation Table), le NTFS (NT File System)

Nous en avons déjà parlé, une donnée (information musicale, photo ou programme…) occupe un certains nombres d’octets sur son support de stockage (mémoire vive ou mémoire de masse). Pour que le système de votre ordinateur soit en mesure de retrouver cette donnée là où elle se trouve, sa position est référencée dans une **table d’allocation des octets** (ou des fichiers, un fichier pouvant contenir n’importe quel type de donnée : photo, vidéo, musique, programme, texte…) appelée FAT16, FAT32, NTFS… en fonction des différents systèmes.

Retenez que la table d’allocation des fichiers traduit à tout moment la localisation d’un fichier sur disque (en termes de piste et secteur ou encore de cluster ou bloc). Si vous abimez cette table, vos données sont inaccessibles bien que toujours présentes sur le disque.

* + 1. Le Pourquoi du formatage

Cette opération réorganise les particules magnétiques d’un support de mémoire de masse et à pour objectif d’en supprimer le contenu de manière définitive (même si des spécialistes sont en mesure de récupérer des données après formatage jusqu’à 7 fois !).

Les traitements des données : le processeur

Le processeur est l’organe de calcul de l’ordinateur. Il exécute les instructions du programme en mémoire vive et traite les données qui s’y trouvent également. Pour cela, il dispose de quelques opérations de base et de quelques accumulateurs (dépendant du type de processeur).

* + 1. La notion de registre

Les registres sont des zones temporaires de stockage de l’information avant leur utilisation par le processeur. Ils sont en nombre réduit, variable selon le type de processeur dans votre ordinateur.

* + 1. Les bus de données

Ce sont des circuits électroniques ou des fils servant au transport des informations de la mémoire vers le processeur, des périphériques vers la carte mère…

* + 1. L’ordonnancement des instructions

C’est le séquenceur qui garantit l’ordre logique d’exécution du programme amenant au résultat.

## Exercices d’application

* + 1. Exercice 501 Présentation de l’ordinateur

Listez les composants internes et externes d’un ordinateur. Donnez le ou les critère(s) de performance pour chacun de ces composants (exemple : « capacité » pour la mémoire). Précisez les unités utilisées pour mesurer chacune de ces performances. Donnez enfin une valeur « actuelle » moyenne pour chacune de ces mesures (Vous pourrez utiliser internet pour les obtenir).

Vous regrouperez dans un tableau l’ensemble de ces informations.

* + 1. Exercice 502 Présentation de l’ordinateur
* A quoi sert la mémoire ? que contient-elle ?
* Quel est le rôle du processeur ?
  + 1. Exercice 503 Fonctionnement de l’ordinateur

**(Si ces notions n’ont pas été abordées en cours, cherchez la réponse sur internet)**

* A quoi sert le formatage d’un disque dur ?
* Que se passe-t-il lorsque vous supprimez un fichier d’un support permanent ?
* Que contient la mémoire vive d’un ordinateur ?

POUR COMMENCER A DECOUVRIR LA MACHINE

# TP de Mise en route

# \* Le noyau du système d’exploitation Windows

# \* Le noyau du système d’exploitation LINUX

**1ère partie : WINDOWS**

**Premier objectif : découvrir quelques commandes de base du noyau Windows**

1. Comprendre l’organisation des fichiers en réseau à L’ECE et la notion de profil itinérant. (Expliquer pourquoi il ne faut pas travailler dans « mes documents » ou sur le « bureau »).
2. Découvrir Windows et son environnement graphique : le menu démarrer, les fenêtres (plein écran/flottante/réduite). Apprendre quelques raccourcis clavier : Win+E, Win+M, Alt+Tab, Alt+Ctrl+Suppr…
3. Apprendre à utiliser l’explorateur de Windows (Win+E) : Parcourir les unités de disque et les répertoires.
4. Créer son espace de travail sur Z : avec un répertoire par séance de TD-TP, voire par exercice.
5. Comprendre ce qu’est une extension de fichier (ex. « .txt »).Apprendre à Afficher/Masquer les extensions de fichier. (Organisation des dossier/Options d’affichage/Onglet affichage/Décocher « Masquer les extensions des fichiers… »). Observer la différence dans l’explorateur.
6. Découvrir la ligne de commande (Démarrer / Exécuter…) et lancer quelques applications via cette ligne (iexplore, calc…)
7. Lancer la commande « cmd » par la ligne de commande et découvrir une fenêtre « console » windows. Comprendre la notion de « noyau » : le DOS sous Windows.
8. Découvrir quelques commandes « shell » sous DOS.
   1. dir
   2. dir /p
   3. dir /w
   4. cd ..
   5. cd c:\ (expliquer la notion de chemin relatif et absolu)
   6. help
   7. echo bonjour
   8. move
   9. ren
   10. start calc

**Deuxième objectif : comment coder en langage de programmation**

Ci-dessus, vous avez découvert l’environnement système et appris à manipuler les rudiments des commandes du noyau qu’on appelle des commandes « shell ».

Nous nous proposons ici d’utiliser ces connaissances pour découvrir les outils de la programmation :

* Editeur
* Compilateur
* Editeur de liens
* Exécuteur

Allons-y !

1. **Edition du code**

**L’étape d’éditio**n correspond à la rédaction du code via un simple « bloc-note » ou n’importe quel autre éditeur, comme « notepad » sous Windows ou « vi » sous Linux par exemple.

Vous pouvez appliquer les mêmes principes sous Windows que sous UNIX, à savoir utiliser le bloc note pour taper votre code, lancer ensuite un compilateur puis un éditeur de liens, et enfin double-cliquer sur le fichier exécutable généré pour le lancer et l’utiliser.

Nous préférerons utiliser une plateforme de développement qui intègre tous ces outils en une seule et même application. Nous avons choisi le logiciel CodeBlocks pour sa gratuité et sa compatibilité multi-OS.

1. **Notion de projet**

Ce logiciel intègre la notion de « projet », associé à un répertoire, dans lequel CodeBlocks gérera automatiquement tous les fichiers intermédiaires à créer pour générer l’exécutable (fichiers objets et autres…).

Vous trouverez sur campus une video qui vous guide dans la création de projet sous CodeBlocks.

Testez CodeBlocks. Découvrez la notion de « projet » et comprenez l’intérêt de systématiquement créer un projet pour tout nouvel exercice, dans un répertoire indépendant, en regardant tous les fichiers qui ont été créés automatiquement. Si vous ne faites pas ça, les fichiers automatiques vont s’écraser les uns les autres au fur et à mesure de vos exercices.

Le premier projet que vous allez créer en mode console avec GNU GCC comme compilateur va transformer l’affichage de CodeBlocks et diviser la fenêtre en 3 parties : l’arborescence du projet à gauche, le code à droite et les messages d’erreur en bas à droite.

Si vous déroulez l’arborescence de ce premier projet en cliquant sur le ‘+’ qui précède « sources » à gauche, vous verrez apparaitre le fichier « main.c » qui contient le même code que celui testé sous UNIX. C’est le modèle par défaut « hello World ! ».

1. **Compilation/Edition des liens/Exécution**

Appuyez sur F9 pour le compiler, éditer les liens et l’exécuter. Une fenêtre console s’ouvre si le programme n’a pas d’erreur. Sinon, une fenêtre vous indique les erreurs de compilation.

Par la suite, pour chaque nouveau projet (chaque nouvel exercice), vous créerez un projet en suivant la même démarche et vous remplacerez l’instruction  ***printf(« Hello World ! ») ;***  par les commentaires issus de votre ACD puis le code résultant de votre travail de transcription de l’ACD.

**Cette partie ne concerne que les utilisateurs de LINUX. Elle ne sera pas à faire en TP à l’ECE.**

***2ème partie : LINUX***

***Premier objectif : découvrir quelques commandes de base du noyau LINUX***

1. *Découvrir un autre système d’exploitation : UNIX (gros système) / LINUX (version PC)*
2. *Accéder au noyau de LINUX via une fenêtre console sous Windows : le TERMINAL (Applications/Accessoires/Terminal)*
3. *Découvrir les commandes UNIX de base*
   1. *ls*
   2. *ls –la (expliquer ce qu’on voit. Présenter la notion de droit rwx pour soi, le groupe et les autres)*
   3. *mkdir info*
   4. *cd info*
   5. *cd .. (avec espace)*
   6. *cd ~/info*
   7. *cd (retour au Home directory)*
   8. *whoami*
   9. *hostname (la machine sur laquelle vous vous êtes logué)*
   10. *edit test.txt (taper la lettre ‘a’ puis un peu de texte et quitter en faisant ESC puis « :wq » (le fichier test.txt a été créé. Vérifier avec ls)*
   11. *more test.txt (pour afficher le contenu sans le modifier)*
   12. *cp test.txt toto.txt (réalise une copie)*
   13. *mv toto.txt titi.txt (renomme le fichier. Vérifiez !)*
   14. *chmod 700 titi.txt (pour changer les droits d’accès. Vérifier avec ls –l)*
   15. *rm titi.txt (supprime définitivement le fichier)*
   16. *exit (pour quitter le shell)*
4. *Découvrir VI (vi est le seul éditeur de texte au niveau du noyau d’UNIX. Les autres éditeurs, plus élaborés, se placent à un niveau plus élevé dans le système et ne sont plus fonctionnels en cas de problème système. Maitriser « vi » permet donc parfois de se sortir de quelques situations problématiques sous UNIX quand il ne reste que le noyau ;-)*
   1. *vi info.txt (vous retrouvez le texte tapé précédemment)*

***Quelques infos sur « vi »***

*vi à deux modes de fonctionnement : le mode commande et le mode édition.*

*Par défaut, vous entrez dans vi en mode commande, ce qui vous permet de vous déplacer dans le texte affiché à l’aide des touches directionnelles ou des touches ‘h,j,k,l’. Pour taper du texte il faut entrer en mode édition avec les touches ‘a’ ou ‘i’. Pour revenir en mode commande il faut appuyer sur ESC.*

*Certaines commandes passent automatiquement d’un mode à l’autre.*

*L’alternance des deux modes est parfois un peu pénible mais on s’habitue.*

*Pour quitter l’éditeur :*

*Esc puis :wq (pour quitter en sauvegardant les modifications)*

*Esc puis :q ! (pour quitter sans sauvegarder)*

***Quelques commandes vi utiles (en mode commande)***

*o (ajoute une ligne en dessous)*

*O (ajoute une ligne au-dessus)*

*x (supprime un caractère)*

*dd (supprime une ligne complète)*

*u (annule la dernière commande)*

*Il existe évidemment beaucoup d’autres commandes shell sous Windows ou UNIX / LINUX.*

*Mais l’usage courant est de préférer la version graphique des systèmes d’exploitation et d’utiliser plutôt la souris sur des icônes.*

*Si toutefois vous êtes intéressé par aller plus loin dans la manipulation système ou dans l’utilisation de vi, internet fourmille de tutoriels très bien construits.*

*D’autres éditeurs plus « conviviaux » existent. Vous découvrirez notamment* ***Gedit****.*

***Deuxième objectif : comment coder en langage de programmation***

1. ***Edition du code***

*A l’aide de l’application NX CLIENT FOR WINDOWS, connectez-vous sous LINUX et ouvrez une fenêtre TERMINAL.*

*Créez-vous un répertoire de travail nommé info (peut-être déjà fait) et un sous-répertoire nommé « bonjour ». Descendez dans le répertoire bonjour.*

*Tapez la commande suivante : vi hello.c*

*A l’aide de vi ou de l’éditeur Gedit, tapez le code suivant :*

*#include <stdio.h>*

*#include <stdlib.h>*

*int main()*

*{*

*// declaration des variables*

*// aucune variable nécessaire dans cet exemple*

*// code*

*printf("Hello world!\n");*

*return 0;*

*}*

*Rappels : ‘a’ pour entrer en mode frappe – Esc :wq pour sauvegarder et quitter vi*

*Vérifiez par la commande ls que votre fichier a bien été créé.*

1. ***Compilation du code***

*Une fois tapé, la syntaxe de votre code doit être vérifiée par le compilateur. Sous UNIX comme sous Windows, nous utiliserons le compilateur GCC développé en logiciel libre.*

*Dans la fenêtre TERMINAL, tapez la commande suivante : gcc hello.c*

*Avec la commande ls, regardez les fichiers qui ont été créés automatiquement par le compilateur. Vous devriez voir un fichier nommé hello.o (le compilateur crée un fichier .o par fichier .c)*

*Ce fichier est un fichier objet, étape intermédiaire avant la création du fichier exécutable.*

1. ***Edition des liens***

*Si votre programme ne contenait pas d’erreur, le compilateur a automatiquement fait appel à l’éditeur de liens pour que tous les fichiers objets soient réunis en un unique fichier exécutable. Dans notre exemple, il n’y a qu’un seul fichier objet, cette étape d’édition des liens n’est donc pas significative.*

1. ***Exécution***

*L’édition des liens terminée, vous devriez voir apparaître un fichier exécutable nommé a.out par défaut sous UNIX. En tapant a.out dans la ligne de commande puis enter, vous déclenchez l’exécution de votre programme et voyez affiché « hello World ! » à l’écran.*

*Voilà la chaîne de programmation est complète…*

*Supprimez le fichier a.out. Reprenez votre fichier hello.c avec vi et introduisez des erreurs volontaires (une à la fois). Vous pouvez par exemple supprimer un ; à la fin d’une ligne, changer la syntaxe de printf en pinf…etc.*

*Lancez la compilation et analysez les messages d’erreur affichés par gcc.*

*Essayez de vous familiariser avec leur signification.*

*Notez au passage que l’éditeur de lien n’est pas appelé et que les fichiers .o ou le fichier a.out n’ont pas été créés.*

# Chapitre 6 - Codage de l’information

## Le binaire

Un peu manichéen, le binaire c’est 0 ou 1, noir ou blanc, chaud ou froid, vrai ou faux…

Ces notions, qui s’opposent deux à deux sont **insuffisantes** pour traduire toutes les subtilités du haut niveau et de ses « plutôt joli », « assez grand », « entre gris clair et gris foncé »…

En attendant de passer à l’ordinateur quantique, qui n’est pas le propos de ce cours (et qui n’est pas encore vraiment au point, il faut bien l’avouer), nous allons découvrir comment le binaire, somme toute assez basique, nous permet malgré tout de coder des informations complexes.

* + 1. 0 ou 1 : le courant passe t’il ?

Si on fait passer par intermittence du courant dans un fil électrique, à un moment donné il est possible de dire si le courant passe ou s’il ne passe pas. C’est binaire !

A cette présence de courant on peut associer une signification, une information, et à l’absence une autre. On peut donc faire passer 2 informations sur 1 fil.

Combien sur 2 fils d’après vous ?

* + 1. 1 fil, 2 fils…n fils

Multiplions les fils, on peut multiplier les informations et presque en faire un discours. Attention cependant à bien parler le même langage d’un bout à l’autre des fils.

## Le codage/décodage de l’info : La Table d’Interprétation Sémantique - TIS

Les informations de haut niveau sont des informations **complexes**. Pour récupérer cette information, une phrase par exemple, le binaire nous permettra de coder une lettre dans une variable de type caractère, puis de regrouper ces caractères pour former des mots, de regrouper les mots pour former la phrase, etc.

Cet assemblage, ce codage, ne sera compréhensible que s’il est correctement décodé. Pour cela, il sera nécessaire d’utiliser la **table d’interprétation sémantique** (T.I.S.) adaptée.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Codage binaire sur 1 octet | Table d’interprétation sémantique des entiers | Table d’interprétation sémantique ASCII | … | Table d’interprétation sémantique des plongeurs sous-marins |
| 0000 0000  0000 0001  .  .  0000 0101  0000 0110  0000 0111  .  .  .  0010 0000  0010 0001  0010 0010  0010 0011  .  .  0010 1000  0010 1001  0010 1010  .  .  .  0011 1100  0011 1101  0011 1110  0011 1111  .  .  1111 1111 | 0  1  .  .  5  6  7  .  .  .  32  33  34  35  .  .  40  41  42  .  .  .  60  61  62  63  .  .  255 | .  .  .  .  A  B  C  .  .  .  Y  Z  0  1  .  .  9  a  b  .  .  .  Z  #  %  \*  .  .  . |  | .  .  Raie manta  Mérou  Barracuda  .  .  Sardine  .  Poisson clown  .  .  Poisson lune  Murène  Congre  .  .  .  .  J’ai plus d’air !  Il me reste 50 bars  .  .  Je suis niveau 1  .  .  .  J’ai de l’eau dans mon masque |

Mais ce n’est pas encore terminé ! Une fois la phrase récupérée, il faut encore en comprendre le sens !!!

## Exercices d’application

* + 1. Exercice 601 Codage de l’information
* Combien d’informations pouvez-vous coder sur 2 bits ? sur 8 bits ?
* Quels sont les codes (ou configurations) associés à ces informations ?
* Quelles sont les valeurs numériques entières associées à ces codes ? en signé ? en non-signé ?
* Si vous souhaitez associer un code à chaque étudiant de l’ECE (environ 1400), combien de bits avez-vous besoin ? d’octets ?
* Qu’est-ce qu’une table d’interprétation sémantique TIS ?

# Chapitre 7 - Stockage des données : variables de type scalaire

Rôle et convention d’écriture

**Un programme informatique manipule de l’information. Chaque information est stockée dans une variable.**

Une **variable** est une entité, une ressource, un objet, une zone mémoire, une instance (oui, tout ca…) destiné à stocker une **information : une donnée**. Cette information contenue dans cette variable peut varier au fil du temps… ou pas ! Une variable dont le contenu ne doit pas bouger (un comble !) pourra s’appeler une **constante**.

Pour demander cette ressource à la machine, il faut la réserver auprès du système d’exploitation (on dira la « **déclarer ») en fournissant un certain nombre d’informations** (taille, nature du contenu…).

Quel que soit le contenu d’une variable, l’information qu’elle contient est codée, dans nos machines actuelles en binaire ?

Déclaration de variable

Une déclaration de variable consiste à **réserver et à nommer** la zone mémoire nécessaire au stockage d’une information. Pour préciser la **taille** de la zone à réserver, la **nature du contenu** de cette zone et le **mode de codage/décodage** de la zone, il faut en préciser le **type**.

Pour donner un nom d’identification à la zone réservée, il suffit de le préciser **après** le type.

La seule chose que le programmeur ne décide pas, c’est l’emplacement mémoire de cette variable. C’est l’ordinateur qui gère sa mémoire de manière autonome et qui définit cet emplacement qu’on appellera **adresse**.

Règles impératives pour la lisibilité du code :

* Nom de variable, nom de fichier, nom de répertoire doivent être **explicites** ! (évitez x,y,z…préférez age, sexe, nom…)
* Pas d’accent, pas d’espace (préférez le « \_ »), pas de caractère spécial, pas de ponctuation !!!

Le domaine de définition et la plage de valeurs

Le mode de codage/décodage, qui est certainement le plus complexe à comprendre dans la notion de variable, dépend du **domaine de définition,** de la **plage de valeur** et de la **Table d’interprétation sémantique (TIS)**. Pour comprendre les 2 premières notions (la troisième a déjà été expliquée précédemment), prenons un exemple :

* En mathématiques les entiers sont définis sur le **domaine de définition** noté\mathbb Z.
* En C, le type entier respecte ce domaine de définition. Mais il existe plusieurs types entiers, dépendant du nombre d'octets sur lesquels ils sont codés ainsi que de leur format, c'est-à-dire s'ils sont signés (négatifs et positifs) ou non (positifs exclusivement), ce qui détermine **la plage de valeurs de la variable** (centrée sur 0). Par défaut les données sont signées.
  + 1. Les altérateurs de type : short, long, signed, unsigned

Les altérateurs de type vont permettre de modifier la plage de valeur des données.

Nous venons de le dire, les variables sont signées par défaut (**signed** est donc facultatif). Pour avoir une représentation **non signée**, il faut ajouter un altérateur de type : **unsigned**

**Exemple**

* Pour un « char », la plage de valeur initiale (puisque le type est signé) est -128 à +127

Pour un « unsigned char » la plage de valeur ira de 0 à 256

* Pour un « [unsigned](http://www.lri.fr/~aze/page_c/aide_c/unsigned.html) int », la valeur maximale, UINT\_MAX est définie dans le fichier <limits.h> et vaut:

Si le type est sur 16 bits, UINT\_MAX   [=](http://www.lri.fr/~aze/page_c/aide_c/affectat.html)   65535   [=](http://www.lri.fr/~aze/page_c/aide_c/affectat.html)   2\*\*16-1

Si le type est sur 32 bits : de 0 à 4 294 967 295.

(UINT\_MAX   [=](http://www.lri.fr/~aze/page_c/aide_c/affectat.html)   2 147 483 648   [=](http://www.lri.fr/~aze/page_c/aide_c/affectat.html)   2\*\*31).

L’altérateur **short** permet de réduire le nombre d’octets utilisés pour un type. L’altérateur **long** permet de l’augmenter. Dans une architecture 32 bits (4 octets), une variable déclarée [**short**](http://www.lri.fr/~aze/page_c/aide_c/short.html) **int**, est représentée sur 16 bits (2 octets).

Les types scalaires de données en C : int, float, char, pointeur

Les données manipulées en langage C sont typées, c'est-à-dire que pour chaque donnée que l'on utilise (dans les variables par exemple) il faut préciser le type de donnée, ce qui permet de connaître l'occupation mémoire (le nombre d'octets) de la donnée ainsi que sa représentation et son mode d’interprétation :

* **les nombres** : entiers (**int**), ou réels c'est-à-dire à virgule (**float**),
* **les caractères**: une lettre (et une seule) peut être stockée dans une variable de type **char**,
* **les pointeurs** : permettent de stocker **l'adresse** d'une autre donnée, ils « pointent » vers une autre donnée.

Il y a 4 donc types scalaires (simples) en C : **Entier** : int

**Réel** : float

**Caractère** : char

**Pointeurs** : int\*, char\*\*\*, float\* …

Voici un tableau **indicatif** décrivant les types de données en langage C et l’effet des altérateurs :

|  |  |  |  |
| --- | --- | --- | --- |
| **Type de donnée** | **Signification** | **Taille (en octets)** | **Plage de valeurs acceptée** |
| char | Caractère | 1 | -128 à 127 |
| unsigned char | Caractère non signé | 1 | 0 à 255 |
| short int | Entier court | 2 | -32 768 à 32 767 |
| unsigned short int | Entier court non signé | 2 | 0 à 65 535 |
| int | Entier | 2 (sur processeur 16 bits) 4 (sur processeur 32 bits) | -32 768 à 32 767 -2 147 483 648 à 2 147 483 647 |
| unsigned int | Entier non signé | 2 (sur processeur 16 bits) 4 (sur processeur 32 bits) | 0 à 65 535 0 à 4 294 967 295 |
| long int | Entier long | 4 | -2 147 483 648 à 2 147 483 647 |
| unsigned long int | Entier long non signé | 4 | 0 à 4 294 967 295 |
| float | Flottant (réel) | 4 | 3.4\*10-38 à 3.4\*1038 |
| double | Flottant double | 8 | 1.7\*10-308 à 1.7\*10308 |
| long double | Flottant double long | 10 | 3.4\*10-4932 à 3.4\*104932 |

* + 1. Nombre à virgule : float

Un nombre réel à virgule flottante, bien que nombre à virgule, peut toutefois être représenté de différentes façons :

* un entier décimal sans partie flottante : 895.00 (ca ressemble à un entier, non ?)
* un nombre à partie décimale séparée par un point (et non une virgule) : 845.32
* une fraction : 27/11
* un nombre exponentiel, c'est-à-dire un nombre (éventuellement à virgule) suivi de la lettre *e* (ou *E*), puis d'un entier correspondant à la puissance de 10 (signé ou non, c'est-à-dire précédé d'un + ou d'un -) :

2.75e-2 35.8E+10 .25e-2

En réalité, les nombres réels sont des nombres binaires dans lequel la position de la virgule est codée par une partie de ses bits (appelée l'exposant), le reste des bits permettent de coder le nombre sans virgule (la mantisse).

**Exemple** : 237.27 mantisse : 23727 exposant : -2 (qui veut dire 10-2)

Les nombres de type **float** sont codés sur 32 bits dont :

* 23 bits pour la mantisse
* 8 bits pour l'exposant
* 1 bit pour le signe

Exemple : 010011110100010101100011-11110010-1

Les nombres de type **double** sont codés sur 64 bits dont :

* 52 bits pour la mantisse
* 11 bits pour l'exposant
* 1 bit pour le signe

Les nombres de type **long double** sont codés sur 80 bits dont :

* 64 bits pour la mantisse
* 15 bits pour l'exposant
* 1 bit pour le signe

La précision des nombres réels est approchée. Elle dépend du nombre de positions décimales, suivant le type de réel elle sera au moins :

* de 6 chiffres pour le type **float**
* de 15 chiffres pour le type **double**
* de 17 chiffres pour le type **long double**
  + 1. Nombre entier : int

Les variables de type int permettent le stockage des entiers.

En général, la taille en octet d’un int est de 2 octets=16 bits sur une machine 16 bits (les anciennes) et sur les microcontrôleurs utilisés pour des applications électroniques numériques.

Mais un int aura une longueur de 4 octets=32 bits sur une machine à processeur 32 bits (quelques PC, smartphones et tablettes) ou les machines à 64 bits (les machines standard d’aujourd’hui).

Un int peut donc être défini sur 2 ou 4 octets en fonction des architectures.

La valeur stockée dans un int (sur 4 octets) va de :

-2 147 483 648 (INT\_MIN)  à 2 147 483 647 (INT\_MAX)   
(cf. tableau ci-dessus)

INT\_MAX, INT\_MIN sont définis dans le fichier <limits.h> et représentent les valeurs limites représentables par le type int.

Un nombre entier est un nombre sans virgule qui peut être exprimé dans différentes bases :

* **Base décimale** : l'entier est représenté par une suite de chiffres unitaires (de 0 à 9) ne devant pas commencer par le chiffre 0
* **Base hexadécimale** : l'entier est représenté par une suite d'unités (de 0 à 9 ou de A à F (ou a à f)) devant commencer par *0x* ou *0X*
* **Base octale** : l'entier est représenté par une suite d'unités (incluant uniquement des chiffres de 0 à 7) devant commencer par *0*

Rappelez-vous, les entiers sont signés par défaut, cela signifie qu'ils comportent un signe. Vous trouverez de nombreux ouvrages ou documentation décrivant le stockage binaire des nombres négatifs.

* + 1. Caractère : char

Le type char (provenant de l'anglais character, avec un « h » de plus qu’en français) permet de stocker **une et une seule lettre, symbole ou ponctuation**. Cette lettre sera codée en binaire par la valeur [ASCII](http://www.commentcamarche.net/base/ascii.php3) du caractère qu’elle représente, c'est-à-dire un nombre entier entre 0 et 256 !

**Pour info** : Par défaut les **variables** sont **signées** (toutes, quel que soit le type !), cela signifie qu'ils comportent un signe (avec ça, si vous n’avez pas compris ;-). Pour stocker l'information concernant le signe (en binaire), les ordinateurs utilisent le [complément à deux](http://www.commentcamarche.net/base/representation.php3#relatif) (voir cours d’archi). Une donnée de type char est donc signée, cela ne signifie bien sûr pas que la lettre possède un signe mais tout simplement que dans la mémoire la valeur codant le caractère peut avoir un équivalent entier de valeur négative...

**Heureusement, cela n’a pas beaucoup de conséquence pour nous**. Mais quand même…Si on désire par exemple stocker la lettre ‘B’ dans une variable de type char (son code ASCII est 66), on pourra définir cette donnée soit en mettant le nombre 66 dans la variable (c’est possible car le C est **faiblement typé** ou encore transtypable), soit en y mettant 'B' où les apostrophes simples signifient code ascii de...

char toto ; // déclaration

toto = 66 ; ou toto = ‘B’ ; // affectation

Il n'existe pas de type de données pour les chaînes de caractères (suite de caractères) en langage C. Pour créer une chaîne de caractères on utilisera donc des tableaux contenant dans chacune de ses cases un caractère avec une case particulière pour marquer la **fin de chaîne**... **‘\0’**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ‘b’ | ‘o’ | ‘n’ | ‘j’ | ‘o’ | ‘u’ | ‘r’ | ‘\0’ | ??? | ??? |

* + 1. Pointeurs (int\*, char\*, float\*, int\*\*\*\*\*,…)

Imaginez un parking de supermarché un samedi après-midi ! Chaque voiture représente une variable stockant une information. Il y a des petites voitures pour les petites données comme les scalaires (données de base) et des camionnettes, des vans, des 4x4…voire des camions ou des bus pour les données plus grosses (contenues dans les variables dites complexes : les tableaux, les listes, les structures…).

Lorsque vous réservez (ou déclarez) une variable, la machine vous attribue une voiture sur le parking (le parking représente les barrettes mémoires), de la taille que vous avez demandé (via le type de la variable) **mais vous ne savez pas laquelle**. Heureusement **le pointeur** est là ! Le pointeur c’est la **clef du véhicule**… et la machine sait toujours retrouver le véhicule dont vous avez la clef. Miracle !

Le lieu de déclaration et portée

La portée d’une variable c’est la portion de code où cette variable est utilisable. Elle est limitée à son **bloc de définition**.

* + 1. Analyse d’un exemple

|  |
| --- |
| #include “mabib.h”  #include <math.h>  int main()  {  // declaration des ressources  int a,b,diff ;    // début du code  printf(″entrez la valeur de A″) ; // obtention des valeurs  scanf(″%d″,&a) ;  printf(″entrez la valeur de B″) ;  scanf(″%d″,&b) ;  while (a != b) // calcul du pgcd  {  int x; // la variable x n’existe que dans ce bloc local. TRES DANGEUREUX !  diff = abs(a – b) ;  if (a>b) a = diff ;  else b = diff ;  }  printf(″Le PGCD est %d \n″,a) ;  system(« pause ») ;  return 0 ;  } |

* + 1. Notion de variable globale

Une variable sera dite **globale** si elle est déclarée dès le début tout en haut avant tout bloc { } (le plus souvent on les place dans la bibliothèque personnelle, nommée mabib.h dans l’exemple ci-dessus). Elle sera alors visible et accessible en lecture/écriture par toutes les lignes de code situées en-dessous et tous les sous-programmes. Attention au mauvais usage des globales !

Créer un type de donnée

Il est possible en C de définir un nouveau type de données grâce au mot clef ***typedef***. Celui-ci admet la syntaxe suivante :

typedef caracteristiques\_du\_type Nom\_du\_type

où

* *Caracteristiques\_du\_type* représente un type de données existant (par exemple *float*, *short int*, ...)
* *Nom\_du\_type* définit le nom que vous donnez au nouveau type de donnée

Ainsi l'instruction suivante crée un type de donnée ***lettre*** calqué sur le type ***char*** :

typedef char lettre ; (ensuite à la déclaration : lettre toto ; toto est un char…magique !)

Nous utiliserons cette création de type pour approcher la notion d’objet en C via les structures.

Conversion de type de données - transtypage

Le langage C est un langage **faiblement typé**, ce qui veut dire qu’un type de donnée peut être choisi pour contenir autre chose que ce pour quoi il est fait, dans un but de réduire la place occupée en mémoire par exemple.

On appelle *conversion de type de données* le fait de modifier le type d'une donnée en un autre. Il peut arriver par exemple que l'on veuille travailler sur un type de variable, puis l'utiliser sous un autre type. Imaginons que l'on travaille par exemple sur une variable en virgule flottante (type *float*) ; il se peut que l'on veuille « supprimer les chiffres après la virgule », c'est-à-dire convertir un *float* en *int*.

La **table d’interprétation sémantique** permettra de préciser le mode d’interprétation que le processeur doit utiliser pour le traitement de la donnée. On retrouve cette notion de table d’interprétation sémantique dans les formats %... utilisés dans les entrées / sorties formatées.

L’opération de conversion peut être réalisée de deux manières :

* **Conversion implicite** : une conversion implicite consiste en une modification du type de donnée effectuée automatiquement par le compilateur. Cela signifie que lorsque l'on va stocker un type de donnée dans une variable déclarée avec un autre type, le compilateur ne retournera pas d'erreur mais effectuera une conversion *implicite* de la donnée avant de l'affecter à la variable.

int x;

x = ‘A’;

// x contiendra après affectation la valeur binaire 01000001, qui peut se traduire en valeur décimale 65, qui correspond également au code ASCII de la lettre A.

* **Conversion explicite** : une conversion explicite (appelée aussi *opération de* ***cast***) consiste en une modification du type de donnée forcée. Cela signifie que l'on utilise un opérateur dit *de cast* pour spécifier la conversion. L'opérateur de cast est tout simplement le type de donnée, dans lequel on désire convertir une variable, entre des parenthèses précédant la variable.

int x;

x = (int)8.324; // x contiendra après affectation la valeur 8.

*Pour info : Lors d'un passage de paramètres, un argument de type* [*char*](http://www.lri.fr/~aze/page_c/aide_c/char.html)*est converti pour être représenté sur la longueur d'un int. Il en est de même pour un argument de type* [*short.*](http://www.lri.fr/~aze/page_c/aide_c/short.html)

Les booléens VRAI et FAUX

! Pas de booléen en C !

Ils sont remplacés par la convention 0 pour FAUX, tout ce qui est différent de 0 pour VRAI.

Ainsi 1 ou 2.85 ou ‘A’ ou -152 ou « il fait beau » correspondent à la valeur booléenne VRAI.

Précisions sur les constantes

La définition la plus classique d’une constante se fait par appel d’une macro-définition.

**Exemple :** #define NB 5 // pas de ;

Le préprocesseur remplace alors toutes occurrences de NB dans le code par la valeur 5.

**Problème :**

* Aucune place mémoire n’est réservée pour cette constante, son adresse est donc inaccessible.
* Le remplacement se fait sans contrôle de l’utilisation.

Pour remédier à cela, la norme ANSI permet d’utiliser le concept de « variable constante », c’est-à-dire une variable qu’on ne pourra plus modifier après l’initialisation.

**Syntaxe** : const int Nb = 5 ;

L’affectation

L’affectation consiste à attribuer une valeur à une variable. Sans affectation, une variable contient **les résidus binaires de sa dernière utilisation.**

* + 1. L’initialisation

L’initialisation n’est autre que la première affectation d’une variable **préalablement déclarée**. Elle lui donne sa première valeur, elle peut se faire lors de la déclaration. Elle est **parfois** indispensable et **toujours** souhaitable.

**Exemple : 4 façons de faire…**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| a) | int age = 27 ; | b) | int age ; | c) | int age ; | d) | int age, val = 5 ; |
|  |  |  | age = 27 ; |  | scanf(« %d »,&age) ; |  | age = val+10 ; |

**Attention** danger : int val, age ;

**!**

val = age +10 ; // ERREUR ! car age n’a pas été initialisé !!!

* + 1. L’affectation des variables scalaires

C’est l’instruction permettant de ranger une valeur dans une variable. Le symbole d’affectation des variables scalaires en C est le **symbole « = »**  qu’on appelle « **prend la valeur** » (ne pas confondre avec l’égalité == qu’on appelle « égal-égal »).

La partie gauche du symbole reçoit le résultat de l’évaluation de la partie droite (val = 12 ; val 🡨 12).

Les données non-scalaires (c’est à dire de type complexe) ont leur propre mode d’affectation !

**Exemple** : int val, nb, a, tableau1[10], tableau2[10] ;

val = 8 ; La zone mémoire identifiée par le nom val reçoit la valeur 8.

val = nb ; La zone mémoire identifiée par le nom val reçoit la valeur de la zone identifiée par le nom nb.

a = a + 10 ; La machine ajoute 10 au contenu de la variable a et range le résultat dans la variable a en écrasant l’ancienne valeur.

tableau2 = tableau1 ; **Attention** ! ! ! Erreur car tableau1 n’est pas scalaire

**!**

Vous agissez ici sur les adresses.

En reprenant l’image du parking, vous aviez 2 bus « tableau1 » et « tableau2 » et donc 2 clefs. Vous vouliez recopier le contenu du bus 1 dans le bus 2 mais avec cette instruction vous n’avez agi que sur les clefs. Maintenant vous avez écrasez la clef du bus 2 par une copie de la clef du bus 1.

Résultat, vous avez deux clefs du bus 1 (tableau1 et tableau2) et vous avez perdu la clef du bus 2. Tant pis pour ce qu’il y avait dans le bus 2, c’est définitivement perdu dans une mémoire RAM inaccessible !

* + 1. Les affectations cachées

Opérateurs unaires spécifiques incluant l’affectation

i++, i--, i+=2, i\*=2 …

i++ ; correspond à i=i+1 ;

++i ; correspond à i=i+1 ;

i-- ; correspond à i=i-1 ;

i\*=2 ; correspond à i=i\*2 ;

Différence entre ++i et i++

Exemple de code :

// 0. Déclaration des variables

int i;

// 1. Début du code

i=10 ;

printf(« blablabla %d »,++i) ;

printf(« blabla…%d »,i++) ;

**Remarque** : La fonction *sizeof (variable);* retourne la taille en octets de votre variable. Cela vous permet de connaitre la puissance de votre microprocesseur afin de savoir si c’est un processeur 16 bits, 32 bits ou encore 64 bits.

## Exercices d’application

POUR VOUS ENTRAINER

* + 1. Exercice 701 trace d’exécution

Si vous faites la trace d’exécution du code ci-dessous, qu’est-ce que ça donne d’après vous ?

// 0. Déclaration des variables

int i;

// 1. Début du code

i=10 ;

printf(« blablabla %d »,++i) ;

printf(« blabla…%d »,i++) ;

* + 1. Exercice 702 bonnes pratiques

Quelle est la place de la déclaration des variables locales dans un code bien écrit ?

A quoi sert l’indentation ?

* + 1. Exercice 703 Variable scalaires
* Combien y a t’il de types scalaires en C et quels sont-ils ?
* Qu'est ce qui les différencie ?
* Donner leur plage de valeurs.
* Par défaut sont-ils signés ou non signés ?
  + 1. Exercice 704 Variable scalaires

Quel type choisir pour coder les nombres :

* 45.876
* 56.0
* 77
* 650987
* 32769
* -32765
* 450009996
  + 1. Exercice 705 Erreurs de compilation

Soit dans un programme les déclarations suivantes. Indiquez, expliquez et corrigez les erreurs.

int 0t, ti, p0;

freste float;

double div-total;

float tata, t2345, char c, cc :

short Err\_, \_E\_,

* + 1. Exercice 706 Affectation

Qu’est-ce qu’une affectation ?

Quelles sont les différentes façons d’affecter une valeur à une variable ?

Peut-on affecter un caractère à un entier ? Un entier à un caractère ? Pourquoi ?

* + 1. Exercice 707 Booléens

Comment simule-t-on un BOOLEEN en C ?

Quelle est la valeur booléenne de la lettre ‘k’ ? Du caractère ‘0’ (zéro) ? Et du chiffre 0 ?

* + 1. Exercice 708 Variable scalaires, permutation

Soit deux variables a et b de type char. Ecrivez le code en C permettant de saisir n’importe quelle lettre dans ces variables, puis réalise l’échange des contenus des 2 variables en mémoire avant d’afficher ce contenu à l’écran.

* + 1. Exercice 709 (TRACE D’EXECUTION)

Vous savez que la machine ne se préoccupe pas de l’indentation de vos programmes et que cette dernière apporte un confort pour les relecteurs de votre code. Pour vous en convaincre, voici des algorithmes écrits avec une **indentation un peu vague**. Essayez de vous y retrouver et d’en comprendre l’importance de respecter une indentation correcte dans vos programmes.

**Rappelez-vous que sans bloc, la machine ne conditionne ou ne répète qu’une seule instruction.**

**Appliquez ces règles sur l’exemple ci-dessous en tentant de suivre l’évolution du contenu des variables X, Y et Z au fur et à mesure de l’exécution du code en réalisant une trace d’exécution (ce qui revient à faire tourner le code à la main).**

X,Y, et Z étant des variables numériques, on considère les deux séquences algorithmiques S1 et S2 :

|  |  |
| --- | --- |
| Séquence S1  {  if ((X<5 ou Y>2) &&( Z>3))  {  X =1 ;  if (Z-Y) >0  Z =10 ;  Y =Y+Z ;  }  else  X =2 ;  Z =Y+Z ;  } | Séquence S2  {  if ((X<5) || ((Y>2) && (Z>3)))  {  X =1 ;  }  if ((Z-Y) >0)  {  Z =10 ;  Y =Y+Z ;  }  else  {  X =2 ;  Z =Y+Z ;  }  } |

Pour chacune des deux séquences, donner les traces d’exécution de X, Y, et Z si l’on suppose qu’à l’état initial ces trois variables ont les valeurs :

a) X :=4 Y :=1 Z :=4

b) X :=4 Y :=5 Z :=4

c) X :=1 Y :=3 Z :=1

Testez ces codes sous Codeblocks pour vérifier vos résultats.

**Rappel : en l’absence de bloc début-fin ou {-} les boucles et les tests ne portent que sur l’instruction qui suit immédiatement.**

**Soyez sûrs, à la fin de ce chapitre, d’avoir bien compris la notion d’information, d’Entrée/Sortie, de codage, de CAST... N’hésitez pas à questionner le chargé de TD si ce n’est pas le cas. Ces notions sont indispensables pour la suite de l’enseignement.**

# Chapitre 8 - Les pointeurs : type scalaire particulier

Définition

Un pointeur est une variable de type scalaire destinée à contenir **une adresse**, par exemple l’adresse mémoire d’une autre variable.

Les pointeurs permettent l’écriture de programme manipulant directement la mémoire de l’ordinateur, ce qui conduit à des programmes plus compacts et plus efficaces mais aussi plus fragiles (**dangereux ?)**.

Déclaration et initialisation

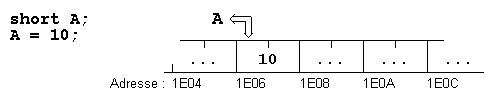
La déclaration des pointeurs dépend de l’objet qu’ils vont pointer. Un pointeur destiné à contenir l’adresse d’un entier sera déclaré de type int\* tandis que celui destiné à contenir l’adresse d’un « pointeur de caractère » sera déclaré char\*\*. Vous noterez qu’il y a toujours une étoile de plus que le type de l’objet pointé.

Immédiatement après leur déclaration, les pointeurs doivent être initialisés à NULL.

Adressage direct

Dans la programmation, nous utilisons des variables pour stocker des informations. La valeur d'une variable se trouve à un endroit spécifique dans la mémoire interne de l'ordinateur. Le nom de la variable nous permet alors d'accéder *directement* à cette valeur.

***Adressage direct :*** Accès au contenu d'une variable par le nom de la variable.

***Exemple***

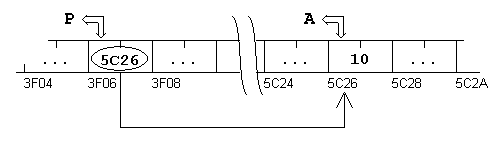
Adressage indirect

Si nous ne voulons ou ne pouvons pas utiliser le nom d'une variable A (variable dynamique par exemple), *nous pouvons copier l'adresse de cette variable dans une variable spéciale P, appelée* ***pointeur*.** Ensuite, nous pouvons retrouver l'information de la variable A en passant par le pointeur P.

***Adressage indirect:*** Accès au contenu d'une variable, en passant par un pointeur qui contient l'adresse de la variable.

**Exemple**

Soit A une variable contenant la valeur 10 et P un pointeur qui contient l'adresse de A. En mémoire, A et P peuvent se présenter comme suit:



Usage

* + 1. Opérateur unaire &

L’opérateur **unaire** **&** renvoie l’adresse de la variable sur laquelle il s’applique.

L’opérateur unaire & ne peut s’appliquer que sur des variables et des éléments de tableaux. L’écriture &(x+1) ou &3 est interdite.

* + 1. Déclaration des pointeurs

// déclaration des variables

int\* px ; int x ;

Soit ***x*** une variable de type **int** et ***px*** une variable de type **pointeur d’entier**.

L’instruction **px = &x ;** affecte l’adresse de la variable ***x*** à la variable ***px***.

On dira alors que ***px* pointe sur *x***.

* + 1. Opérateur unaire \*

L’opérateur unaire \* permet d’accéder au contenu pointé par la variable de type adresse sur laquelle il s’applique.

Soit ***y*** de type entier, l’instruction **y = \*px ;** range dans ***y*** le contenu de l’adresse pointée par ***x***.

La suite d’instructions **px = &x ;**

**y = \*px ;**

est équivalente à **y = x ;**

Les pointeurs pour le passage de paramètres par adresse à une fonction

Les pointeurs interviendront également lors de l’échange de données entre les différentes parties d’un programme structuré. Ils permettront de rendre les données échangées « **modifiables** » par le récipiendaire, ou au contraire « **non modifiables** » par le récipiendaire, afin de gérer des aspects sécurité.

Nous reviendrons sur ces mécanismes complexes dans les chapitres sur les sous-programmes et les passages de paramètres.

## Exercices d’application

* + 1. Exercice 801 (Pointeurs)
* Déclarez un entier val1, un caractère non-signé val2, et un caractère nommé lettre.
* Déclarez un pointeur sur chacune de ces trois variables nommés pt1, pt2 et pt3.
* Déclarez un pointeur sur chacun des 3 pointeurs précédents que vous nommerez ppt1, ppt2 et ppt3.
* Réalisez les affectations permettant de « raccrocher » tous les pointeurs sur leur variable respective.
* Initialisez les 3 premières variables par une affectation directe et affichez leur contenu à l’écran.
* Modifiez les 3 premières variables en utilisant cette fois une affectation indirecte à 1 indirection.
* Affichez de nouveau la valeur des 3 variables val1, val2 et lettre en utilisant cette fois la première indirection
* Modifiez les 3 variables par une affectation utilisant la 2ème indirection.
* Affichez de nouveau la valeur des 3 variables val1, val2 et lettre en utilisant cette fois la deuxième indirection.

# Chapitre 9 - Calculs, comparaisons et operateurs logiques

Les opérateurs mathématiques

Les opérateurs sont soit unaires, binaires ou ternaires, c’est à dire qu’ils acceptent un, deux ou trois opérandes.

**+ - \* / %**

**Règle 1** : Dans le cas où les opérandes ne sont pas de même type, le compilateur « agrandi » en général l’opérande le plus étroit.

**Règle 2** : L’affectation d’un type étroit à un type plus large n’entraîne aucun problème et se fait implicitement (conversion implicite). Dans le cas contraire, il y a risque de perte d’information.

### Cas particulier de la division

La division en C est Type-dépendante, c’est-à-dire que la division d’un int par un int donnera un int, la division d’un float par un float donnera un float…

En respect de la règle 1 précédemment citée, que donnera la division d’un int par un float ? D’un flaot par un int ? D’un int par un char ?

Comment obtenir un float quand on divise un int par un int ?

### Le Modulo

Le Modulo « % » correspond au reste de la division entière. 7%3 vaudra donc 1. Que vaut 7/3 ?

Le CAST

La notion de **CAST** (conversion explicite) permet d’effectuer un changement de type de manière explicite en précisant la conversion.

**Il est nécessaire de maîtriser la notion de CAST pour comprendre que la DIVISION en C est**

**« Type-dépendante » !**

Attention aux « cast » automatiques !

**Exemple** : float C ;

int A=2, B=6 ;

C = A / B ; // donne 0.0 (division de deux entiers)

C = **(float)** A / **(float)** B ; // donne 0.33333 (division de deux flottants)

Les opérateurs de comparaison

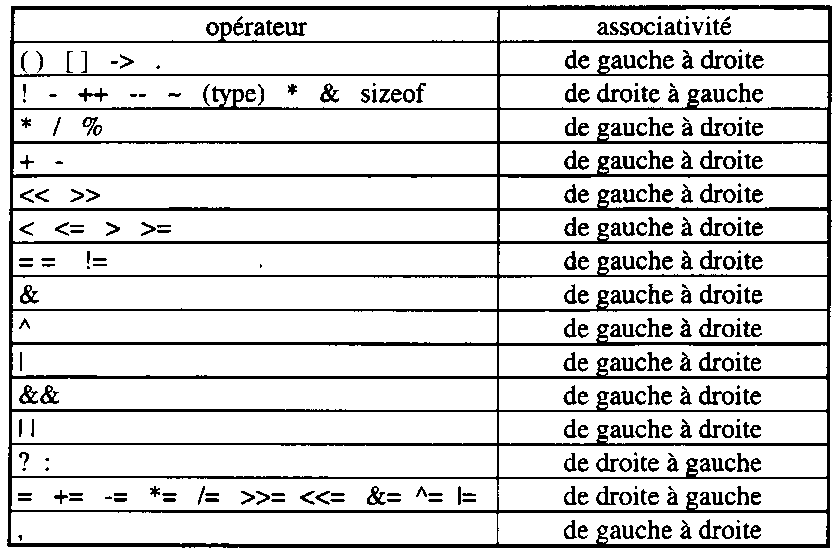
= = > < ! = >= <=

## Les opérateurs logiques

&& (ET) || (OU) ! (NON)

### Tables de vérité des opérateurs logiques

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **A** | **B** | **Not A** | **A et B** | **A ou B** | **A xou B** |
| **V**  **V**  **F**  **F** | **V**  **F**  **V**  **F** | **F**  **F**  **V**  **V** | **V**  **F**  **F**  **F** | **V**  **V**  **V**  **F** | **F**  **V**  **V**  **F** |

* + 1. Priorité des opérateurs

Opérateurs binaires de traitement des bits

Certains opérateurs agissent directement sur les bits composant le codage de la variable. Ces opérateurs spécifiques ne sont applicables qu’aux opérandes signés ou non de type : char, short, int et double.

* + 1. & (ET)

Exemple : 6 0 0 0 0 0 1 1 0

&

10 0 0 0 0 1 0 1 0

2 0 0 0 0 0 0 1 0 à ne pas confondre avec (6 && 10) → VRAI

* + 1. | (OU)

Exemple : 1 0 0 0 0 0 0 0 1

|

121 0 1 1 1 1 0 0 1

121 0 1 1 1 1 0 0 1

* + 1. ^ (OU exclusif, ou XOU)

Exemple : 1 0 0 0 0 0 0 0 1

^

121 0 1 1 1 1 0 0 1

120 0 1 1 1 1 0 0 0

* + 1. op1 << op2

(décalage binaire gauche de op1 un nombre de fois égal àop2)

Exemple : x=1 0 0 0 0 0 0 0 1

X<<2

4 0 0 0 0 0 1 0 0 eq. (X\*4)

* + 1. op1 >> op2

(Décalage binaire droite de op1 un nombre de fois égal à op2)

Exemple : x=8 0 0 0 0 1 0 0 0

X>>2

2 0 0 0 0 0 0 1 0 eq. (X/4)

* + 1. ~ (complément à 1 d’un opérande de type entier, inversion des bits)

Exemple : x=28 0 0 0 1 1 1 0 0

~X

227 1 1 1 0 0 0 1 1

## Exercices d’application

* + 1. Exercice 901 (LOGIQUE)

Afin d’analyser des résultats d’examen, 4 variables permettent de décrire l’environnement : les variables numériques Nlv, Nf, Nm, Np qui indiquent respectivement, pour un candidat donné :

* des notes littéraires : langue vivante (Nlv), de français (Nf)
* des notes scientifiques : mathématiques (Nm), et physique (Np).

On suppose que les notes sont calculées sur 20 et qu’elles ont toutes le même coefficient.

Formez les expressions logiques (et seulement elles) correspondant aux situations suivantes :

1. la moyenne des quatre notes est supérieure à 10
2. les notes de mathématiques et de français sont supérieures à la moyenne des quatre notes
3. il y a au moins une note supérieure à 10
4. toutes les notes sont supérieures à 10
5. la moyenne (10) est obtenue pour l’un des deux types (littéraire et scientifique)
6. la moyenne des quatre notes est supérieure ou égale à 10 et la moyenne (10) est obtenue pour l’un des deux types
   * 1. Exercice 902 (ACD+ALGO)

Donnez le modèle, l’analyse chrono-descendante, l’algorithme (pas le code C car vous ne connaissez pas encore la syntaxe des tests et des boucles) et tous les scénarii possibles pour les jeux d’essai d’un programme qui détermine le nombre de valeurs distinctes parmi trois variables à faire saisir par l’utilisateur. (ex : 8, 8 et 8 saisi par l’utilisateur donne 1 valeur distincte ; 8,1,8 en donne 2 et 8,2,5 en donne 3).

* + 1. Exercice 903 (ACD+ALGO+TRACE)

Ecrivez le modèle, l’ACD, l’algorithme et les scénarii d’un programme calculant la moyenne de nombres entiers positifs saisis par l’utilisateur. La saisie d’un nombre négatif entraîne l’affichage du nombre de valeurs positives saisies et de la moyenne calculée.

Décrivez les ressources.

Faites tourner votre solution à la main sur des exemples en indiquant la trace d’exécution listant l’évolution du contenu de chacune des ressources. Vous penserez à tous les cas possibles.

* + 1. Exercice 904 (opérateurs et booléens)

Ecrivez un programme en algorithmique qui utilise les opérateurs vus dans ce chapitre et les booléens pour déterminer si une variable saisie au clavier par l’utilisateur est paire ou non. Vous afficherez un message « la valeur est paire » ou « la valeur est impaire » en conséquence à l’écran.

# Chapitre 10 - Les entrées/sorties clavier/ecran

Dans les premiers chapitres de ce cours, nous avons différencié les E/S externes qui, pour simplifier, s’adressent à l’utilisateur, et les E/S internes qui correspondent à la transmission d’informations aux différentes parties d’un programme complexe.

Etudions dans ce chapitre les E/S externes clavier/écran.

## Les E/S formatées

* + 1. Ecriture : printf (écrire, traitement de sortie)

La fonction « printf » permet au programme d’afficher une donnée à l’écran.

Cette fonction, contenue dans la bibliothèque standard STDIO.H, attend comme premier paramètre la chaîne de caractère à imprimer avec éventuellement la description des formats d’affichage (cf. table d’interprétation sémantique) des variables à afficher dans cette chaîne. Les paramètres suivants correspondent, **dans l’ordre**, à chaque variable dont on souhaite afficher la valeur, aux positionnements respectifs des formats.

**Exemple :** printf(« La valeur des variables x et y sont : %d et %f \n »,x,y) ;

Les deux formats d’affichage %d et %f seront respectivement remplacés à l’écran par la valeur de x au format entier et de y au format flottant.

* + 1. Lecture : scanf (lire, traitement d’entrée)

La fonction « scanf » permet à l’utilisateur de saisir une donnée au clavier.

Cette fonction, également contenue dans la bibliothèque standard STDIO.H, attend comme premier paramètre la chaîne décrivant les formats d’interprétation des variables à lire au **clavier**. Les paramètres suivants sont, **dans l’ordre**, **l’adresse mémoire** (zone de stockage) des variables dont on souhaite faire saisir la valeur.

**Exemple :** scanf(« %d%f \n »,&x, &y) ;

Le & permet d’accéder à l’adresse des variables. Attention à ne pas mettre d’espace entre l’opérateur « % » et le format « d » (entier) ou « f » (flottant).

* + 1. Les formats

Les fonctions printf et scanf utilisent les mêmes formats.

* Entier : %d
* Réel : %f
* Caractère : %c
* Entier/Réel long : %ld, %lf
* Hexadécimal : %x
* Chaîne de caractères %s

**Attention !!** fflush(stdin) parfois indispensable sous WINDOWS.

**!**

* + 1. Les alignements

Le printf offre en plus la possibilité de définir le nombre d’emplacements à utiliser pour l’affichage d’une valeur, dans le but d’aligner des valeurs à l’écran par exemple.

**Exemple** : printf(« **valeur de X :%**7.3f », 3.5261) ;

donnera à l’écran >>>valeur de X :¤¤ 3.526 (chaque ¤ représentant un espace)

* + 1. Les caractères spéciaux

Les fonctions printf et scanf utilisent certains caractères spéciaux précédés par un anti-slash (\n, \t…).

A la ligne : \n

tabulation : \t

beep : \a

Exemple :

*heure = 13 ;*

*min = 27 ;*

*printf (« aujourd’hui\nil fait beau\net il est :\t%d :%d. »,heure,min) ;*

donnera à l’écran :

*>aujourd’hui*

*>il fait beau*

*>et il est : 13 :27.*

! Attention à l’utilisation de ces caractères spéciaux dans un scanf ! Problématique !!

## Exercices d’application

* + 1. Exercice 1001 (scalaires, TIS, transtypage)

Dans un programme, déclarer un char et lui affecter une valeur choisie au hasard.

Afficher la valeur entrée en utilisant les deux formats %d et %c, qu'est-ce que ça donne ?

Réessayer en demandant cette fois une valeur comprise entre 97 et 122. Que remarquez-vous ? Quelles valeurs permettent d'afficher les caractères \*, @, #, $ ?

* + 1. Exercice 1002 (scalaires)

Ecrivez le modèle, l’analyse descendante, l’algorithme, l’organigramme puis les scénarii des différents jeux d’essai d’un programme C qui permet de convertir caractère par caractère, des minuscules en majuscules. Chaque caractère est saisi par l’utilisateur puis converti.

Vous n’êtes pas autorisé ici à utiliser les fonctions prédéfinies du C comme « toupper ».

Vous devez baser votre raisonnement sur le codage de l’information dans la table d’interprétation sémantique ASCII (mais sans entrer de valeur « en dur » genre 32) et sur le décalage Min/Maj.

* + 1. Exercice 1003 (E/S)

Testez les fonctions scanf, printf sur des entiers, des flottants et des caractères.

# Chapitre 11 - Les entrées/sorties en Fichier Texte (ASCII)

Les données externes ne viennent pas toujours de l’utilisateur et les résultats ne sont pas toujours à afficher à l’écran. On peut en effet supposer que notre programme aille lire des données sur une clef USB, dans le fichier qui lui aura été spécifié.

De même, les résultats obtenus par le programme peuvent certes être affichés à l’écran mais également stocké sur la clef.

Cette possibilité d’utiliser les fichiers offre des avantages :

Intérêt des fichiers : la permanence des données

Les fichiers permettent de stocker de manière **permanente** des données sur un disque dur, une clef USB ou un disque réseau. On distingue deux types de fichier :

* Les fichiers texte ASCII dont le contenu n’a pas de formatage particulier et qui sont souvent visualisables à l’aide d’un simple bloc-notes.
* Les fichiers binaires (images, vidéo, son) mais aussi les fichiers Word, Excel… et les pages web qui utilisent un formatage spécifique traduit pas leur extension.

On utilisera donc souvent les fichiers pour se substituer à l’utilisateur dans la fourniture de données en entrée, ou encore pour stocker des résultats, des scores à des jeux, des niveaux de progression pour reprendre à l’endroit où on s’est arrêté…etc.

Ce chapitre ne concerne que la première catégorie : les fichiers ASCII.

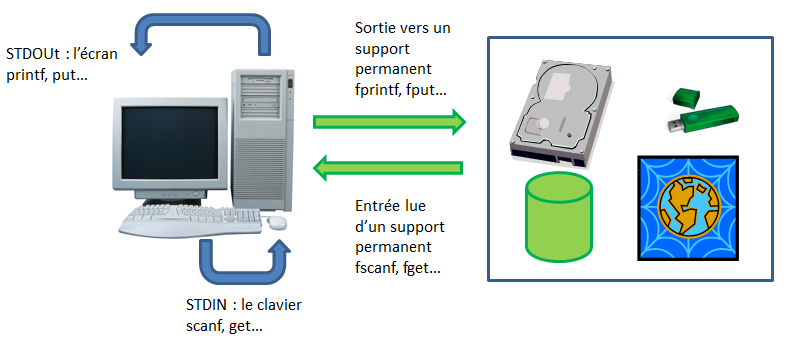
Un des chapitres de la fin de ce polycopié de cours détaille le fonctionnement des fichiers binaires et complète ce chapitre.

## Ecriture et lecture dans un fichier

Par défaut, le compilateur considère que la source standard des entrées (STDIN) est le **clavier** et que la sortie standard (STDOUT) est **l’écran**.

Il est possible de contourner cette association standard en utilisant des fonctions qui permettent de spécifier le flux (file) d’entrée ou de sortie.

Ces fonctions spécifiques commencent en général par un ‘f’ (comme « file ») suivi du nom de la fonction standard.



* + 1. Lecture dans le fichier

Pour lire une valeur entière au clavier et la stocker dans la variable val on écrira :

*int val ; scanf (« %d », &val) ;*

Pour lire une valeur entière dans un fichier et la stocker dans la variable val on écrira :

*int val ; fscanf (fic, « %d », &val) ;*

Dans les deux cas, la valeur lue sera stockée en mémoire dans la variable val. C’est l’origine de la valeur qui varie ici.

* Dans le premier cas, la valeur est lue sur le clavier après la frappe de l’utilisateur.
* Dans le second cas, la valeur est lue dans un fichier texte.
  + 1. Ecriture dans le fichier

Pour écrire à l’écran une valeur de type char contenue dans une variable val, on écrira :

*char val=’m’ ; printf (« %c », val) ; ou putc(val) ;*

Pour écrire dans un fichier une valeur de type char contenue dans la variable val, on écrira :

*char val=’m’ ; fprintf (fic, « %c », val) ; ou fputc(val, fic) ;*

* Dans le premier cas, la valeur stockée en mémoire dans la variable val sera affichée à l’écran.
* Dans le second cas, la valeur stockée en mémoire dans la variable val sera stockée dans un fichier permanent, sans apparaître à l’écran.

Vous aurez noté dans ces exemples que dans le cas d’un accès à un fichier, un paramètre supplémentaire est passé aux fonctions appelées. Ce paramètre s’appelle « *fic »* dans les lignes de code ci-dessus.

Ce paramètre « fic » permet de préciser au programme de quel fichier il s’agit parmi les centaines de fichiers présents sur votre ordinateur ou sur votre clef USB…

* + 1. Progression dans le fichier

Il est inutile de se déplacer dans un fichier. Toute opération de lecture ou d’écriture fait automatiquement avancer le positionnement dans le fichier pour se placer sur la prochaine valeur à lire ou la prochaine position pour écrire.

Cependant il existe des fonctions de positionnement forcé qui peuvent parfois être utiles.

*rewind (fic) ; // permet de se replacer au début du fichier*

*fseek ( fic, position, SEEK\_SET) ; // permet de se placer à l’octet “position” à partir du début*

*fseek ( fic, -position, SEEK\_END) ; // permet de se placer à l’octet “position” avant la fin du  
 // fichier. La valeur de position étant négative, on recule à partir de la fin*

*fseek ( fic, position, SEEK\_CUR) ; // permet de se placer à l’octet “position” à partir de la   
 // position courante*

Selon ce principe, *fseek( fic, 0, SEEK\_END) ;* vous positionne à la fin du fichier, donc sur le EOF.

* + 1. Fin de fichier

Lorsqu’on accède à un fichier, en lecture par exemple, il peut être nécessaire de savoir qu’on a atteint la fin pour éviter de continuer à chercher à y lire des valeurs.

Il existe 2 méthodes pour cela.

La première consiste à faire appel à la fonction feof(fic) ; qui renvoie VRAI (1) si la fin du fichier est atteinte, FAUX (0) sinon.

Exemple : *while ( ! feof(fic) ) printf( « %c », fgetc(fic) );*

La seconde consiste à tester la valeur lue en sachant que la dernière valeur du fichier est un marqueur de fin de type entire qui vaut la constant EOF (pour End Of File).

Exemple : *int val ;*

*while ( (val = fgetc(fic)) != EOF) printf(« %c », val) ; // avec transtypage du int en char ici*

Droits d’accès aux fichiers permanents

Votre disque dur, votre clef USB, ou encore votre espace partagée sur le web (ou Cloud) sont susceptibles de contenir des centaines de fichiers. Pour vous y retrouver vous avez créé des répertoires et des sous-répertoires pour organiser de manières logiques vos données.

Un fichier sera donc identifié par l’ordinateur selon son nom (exemple « monfic.txt »), et sa localisation ou son chemin d’accès (exemple « c:\tmp » (WIN) ou « /home/dupont/ » (Linux)). Dans le cas des fichiers binaires, la machine à également besoin de son mode de cryptage, de compression ou d’organisation qui sont décrites via son extension… .doc… .xls…. .html…

**Rappel** : les fichiers ASCII contiennent des caractères bruts et n’ont aucun cryptage particulier.

Les informations d’accès au fichier devront être stockées dans une structure spéciale accessible via un pointeur dans votre programme. Cette structure, de type FILE, est automatiquement créée par l’appel à la fonction « *fopen* »

* + 1. Ouverture et droit d’accès

Pour manipuler un fichier il faut tout d’abord se déclarer un pointeur sur structure de type FILE au niveau des déclarations de variables, c'est-à-dire au tout début du sous-programme, dans la zone de déclaration des ressources.

Pour ouvrir un fichier et y avoir ainsi accès, il faut ensuite préciser son chemin d’accès, son nom et le mode d’accès que l’on souhaite obtenir parmi lecture ou écriture.

La fonction d’ouverture du fichier est la fonction « *fopen* ».

**Exemple** : *// déclaration des ressources*

*int val ;*

*FILE \* fic = NULL ;*

*// ouverture du fichier*

*fic = fopen (« c:\\home\\tmp\\monfic.txt », « r ») ; // le “r” correspond au*

*// droit d’accès* **read**

*// autre syntaxe possible(choisir l’une ou l’autre)*

*fic = fopen (« /home/tmp/monfic.txt », « r ») ;*

*// accès en lecture au fichier et récupération de l’entier qui s’y trouve*

*// pour le stocker dans la variable val*

*fscanf ( fic, « %d », &val) ;*

*// affichage de la valeur lue à l’écran*

*printf (« la valeur lue dans le fichier est %d\n »,val) ;*

* Pour accéder en lecture à un fichier texte, il faut utiliser le droit « r »
* Pour accéder en écriture à un fichier texte et écrire au début en écrasant le contenu éventuel, il faut utiliser le droit « w ».
* Pour accéder en écriture à un fichier texte et ajouter du contenu à la fin d=sans écraser le contenu déjà présent, il faut utiliser le droit « a ».
  + 1. Fermeture du fichier

Une fois qu’il n’est plus nécessaire de travailler dans le fichier, il est important de penser à le fermer pour autoriser d’autres utilisateurs à y accéder. Pour cela, il suffit de faire appel à la fonction « fclose ».

**Exemple** : *// fermeture du fichier*

*fclose (fic) ;*

Dans le cas où le programme doit accéder plusieurs fois au fichier, avec des droits d’accès différents d’une fois sur l’autre, il est préférable de fermer le fichier après chaque « groupe d’accès ».

## Exercices d’application

* + 1. Exercice 1101 (Fichiers)

Ecrivez l’analyse un programme C qui permet de lire des entiers dans un fichier ASCII créé via le bloc-notes. Vous afficherez les nombres lus pour vérifier l’efficacité de votre programme.

* + 1. Exercice 1102 (Fichiers)

A l’inverse, écrivez un programme C qui permet de stocker dans un fichier ASCII le contenu de quelques entiers, sur une même ligne ou bien les uns sous les autres. Vous ouvrirez ensuite ce fichier à l’aide du bloc-notes pour en vérifier le contenu.

* + 1. Exercice 1103 (Scalaires, fichiers)

Ecrivez maintenant un programme C qui permet lire des entiers dans un fichier texte (ASCII) pour les stocker dans des variables en mémoire. Vous modifierez ensuite ces entiers (en leur ajoutant 10 par exemple). Vous stockerez enfin ces entiers dans un **autre** fichier ASCII sur une même ligne ou bien les uns sous les autres. Vous ouvrirez ensuite le fichier source et le fichier destination à l’aide du bloc-notes pour en vérifier le contenu.

* + 1. Exercice 1104 (Opérateurs)

Ecrivez l’analyse descendante, l’algorithme ou l’organigramme puis le programme C permettant de décomposer un nombre **à 4 chiffres** obligatoirement saisi par l’utilisateur en unité-dizaine-centaine-millier….

Exemple de résultat à afficher:

Nombre saisi : 4127

127 se décompose en :

7 unités

2 dizaines

1 centaine

4 milliers

**Précision** : vous devrez trouver une solution qui ne fait intervenir ni les boucles, ni les tests, ni les tableaux, **ni la division, ni le modulo**… Réfléchissez !

Vous stockerez dans un fichier le résultat de votre décomposition en respectant l’affichage décrit ci-dessus.

# Chapitre 12 - Les Tests

Les tests sont utilisés pour « protéger » (autoriser ou non) l’exécution d’une partie du code et éviter qu’elle ne s’exécute sur des données erronées. Il est en effet préférable que l’interface et les sous-programmes qui la gèrent se préoccupent du blindage des saisies et garantissent ainsi que l’exécution de la suite du programme se fasse sur des données fiables.

Les tests sont aussi utiles pour distinguer les traitements à effectuer sur telle ou telle catégorie de valeur. Ils sont traduits en C par le mot-clef ‘if » suivi d’une condition entre parenthèses. Ils peuvent parfois être complétés du mot-clef « else » qui permet de créer 2 branches dans le code qui passera alors obligatoirement par l’une ou l’autre.

Les structures de contrôle conditionnelles : les tests

* + 1. Test If

Exemple : if (age < 25) printf(″Vous avez droit à la réduction Jeune″) ;

if (age > 60)

{

tarif = tarif – 20 ;

printf(″Le prix de votre billet est %d″, tarif) ;

}

* + 1. Test If…Else…

Exemple : if ( (age > 60) && (age < 80) )

{

tarif = tarif – 20 ;

printf(″Le prix de votre billet est : %d \n″, tarif) ;

}

else printf(″Le prix de votre billet est : %d \n″, tarif) ;

* + 1. Test « Switch » (branchement)

Exemple : char choix, choix2 ; // choix sera de type caractère ici

… // Coder ici l’affichage du menu d’actions et récupérer le choix utilisateur

// printf…scanf…

// Analyser le choix

switch (choix)

{

case ‘1’ : printf (″Vous avez fait le choix 1″) ;

// coder ici le traitement correspondant au choix 1

break ;

case ‘2’ : printf(″Vous avez fait le choix 2″) ;

// idem pour le choix 2

break ;

case ‘a’ : printf(″Vous avez fait le choix a″) ;

/\* afficher un sous menu \*/

scanf(« %c », &choix2) ;

break ;

default : printf(″Votre saisie est erronée\n″)

}

Exemple : Afficher un menu d’actions a l’utilisateur et récupérer son choix

// ressources

int choix ; // choix sera de type entier

… // coder ici l’affichage du menu et l’obtention du choix

// analyse du choix

switch (choix)

{

case 1 : printf (″Vous avez fait le choix 1″) ;

break ;

case 2 : printf(″Vous avez fait le choix 2″) ;

break ;

default : printf(″Votre saisie est erronée\n″) ;

}

Le break permet de ne pas enchaîner tous les cas à partir du branchement.

Dans le premier exemple de switch de la page précédente, la variable choix est de type char.

On peut donc tester si l’utilisateur y a stocké un caractère comme la lettre ‘a’, mais également un caractère comme le chiffre ‘1’ ou le chiffre’2’…, considérés ici comme des éléments de la table ASCII. On aurait donc aussi pu tester le caractère ‘#’ ou ‘€’…

Dans le second exemple de switch, la variable choix est de type int, il n’est donc pas possible de lui donner comme valeur le caractère ‘a’. L’usage des quotes ‘ ‘ n’est plus possible et seules les valeurs numériques entières peuvent être testées dans chaque case.

**Le switch n’est valable que sur les types int et char. Il est impossible de faire un switch sur des flottants ou sur des chaines de caractères ou des mots.**

## Exercices d’application

* + 1. Exercice 1201 Modulo

Ecrivez l’analyse descendante, l’algorithme ou l’organigramme puis le code C permettant de déterminer si une valeur saisie par l’utilisateur est paire ou non. Un multiple de 5 ?

* + 1. Exercice 1202 Cast

Ecrire un programme permettant de tester tous les cas de figure de la division en C afin d’illustrer le problème des CAST.

* division d’un int par un int, rangé dans un int et affiché en %d
* division d’un int par un int, rangé dans un float et affiché en %f
* division d’un float par un int, rangé dans un float et affiché en %d
* division castée
* …

Jouez sur les types des opérandes, les types de la variable de récupération du résultat, le format d’affichage du résultat à l’écran…

* + 1. Exercice 1203 Tests, intervalles, transtypage

En vous efforçant à nouveau d’exploiter la table ASCII mais sans entrer de valeur « en dur » (genre 32), écrivez l’analyse descendante, l’algorithme ou l’organigramme puis le programme C permettant de déterminer si le caractère entré par l’utilisateur est un chiffre, une lettre majuscule, une lettre minuscule ou autre chose… Après analyse, vous afficherez à l’écran la nature du caractère saisi.

# Chapitre 13- Les Boucles de répétition

A ne pas confondre avec les tests qui autorisent ou interdisent une seule fois l’exécution d’une partie de code.

Les boucles quant à elles sont utilisées lorsqu’une partie du code doit être exécutée de façon répétitive (donc plus d’une fois). Les boucles contiennent une condition d’exécution (une **condition de continuité** en C, une condition d’arrêt dans les autres langages) qui détermine le nombre de tours de la boucle.

Les structures de contrôle répétitives : les boucles

Elles permettent la répétition d’une ou d’un bloc d’instructions. Elles seront choisies en fonction du nombre de répétition à effectuer.

* + 1. Boucle FOR (nombre de répétitions connu)

Exemple : for (i=0 ; i<9 ; i++) // i += 2 , i-- , i=i+3 …

{

printf(″La valeur ″, i ,″ est : %d″, i ) ;

}

Modification du compteur i interdite dans le corps de boucle. Si nécessaire, changez de boucle !

* + 1. Boucle DO…WHILE… (au moins une fois)

Exemple : magic = 6 ;

do

{

printf(″Donnez un nombre (0 pour arrêter) : ″) ;

scanf(« %d », &nb); // la variable du test évolue ici

if (nb = = magic) printf(″Gagné !\n″) ;

} while (nb != 0) ;

Modification obligatoire des variables intervenant dans le test à l’intérieur du corps de boucle sous peine de rester bloqué dans une boucle infinie.

* + 1. Boucle WHILE… (peut-être pas du tout)

Exemple : int i = 0 ; int note = -1;

scanf(« %d », &note) ;

while ( (note < 0) || (note > 20) )

{

printf(″La note saisie n’est pas valide, recommencez.\n″) ;

scanf(« %d », &note) ;

i++ ; // i sert ici à compter le nombre d’erreurs de saisie si on //veut limiter le nb d‘essais

}

Modification obligatoire des variables intervenant dans le test à l’intérieur du corps de boucle sous peine de rester bloqué dans une boucle infinie.

**Attention !!** Vous l’aurez compris, chaque boucle à un rôle précis. **Vous devez les connaître**. C’est ça « être rigoureux » !

**!**

## Exercices d’application

* + 1. Exercice 1301 Observation et correction

Maintenant que vous êtes capable de réaliser des programmes en C un peu plus évolués, voici un autre genre d’exercice qu’il vous faut apprendre à maîtriser : **la correction de programmes déjà écrits**.

Relisez et corrigez les erreurs que vous repérez dans le code ci-dessous en jouant le rôle du compilateur. Copiez-collez le code corrigé puis compilez-le pour vérifier que vous les aviez bien toutes détectées.

#includ <stdio.h>

int main()

{

/ declaration des ressources

int essai==0 ;

int a\_trouver ;

// initialisation (en dur) de la valeur que le joueur doit rechercher

a-trouver = 18 ;

// boucle de jeu

while (essai < >a\_trouver) // différent de…

{

printf (« entrez votre valeur)

scanf (« %f », essai) ;

if (essai = 0) print (« \nValeur nulle interdite\n ») ;

if (essai > a\_trouver) printf (« /nTrop grand !\n ») ;

else printf (« \nTrop petit) ;

}

pintf (« \n\nBravo !!\n ») ;

return 0 ;

}

* + 1. Exercice 1302

Ecrire le modèle, l’ACD, l’algorithme et les scénarii d’un programme C qui convertit un entier naturel en chiffres romains, en utilisant l’ancienne notation **uniquement additive** : exemple : 4 ( IIII), 9 ( VIIII), 900 ( DCCCC)

La valeur convertie sera affichée à l’écran mais ne sera pas stockée en mémoire.

Rappelons les éléments de base :

I : 1, V :5, X : 10, L : 50, C : 100, D : 500, M : 1000.

* + 1. Exercice 1303

Ecrire un programme C qui propose à l’utilisateur de dessiner à l’écran certaines figures composées d’étoiles. La figure **ne sera pas stockée** dans une matrice en mémoire.

Les figures seront proposées par un menu (triangle, carré, sablier…). La hauteur de la figure sera saisie par l’utilisateur.

Exemple : hauteur=4

|  |  |  |
| --- | --- | --- |
| \*  \*\*\*  \*\*\*\*\*  \*\*\*\*\*\*\* | \*\*\*\*  \*\*\*\*  \*\*\*\*  \*\*\*\* | \*\*\*\*\*\*\*  \*\*\*\*\*  \*\*\*  \*  \*  \*\*\*  \*\*\*\*\*  \*\*\*\*\*\*\* |

* + 1. Exercice 1304

Simulation d’épargne.

Vous épargnez une somme de 30€ tous les 2 de chaque mois sur un compte vous rapportant 0.3% mensuel, avec des intérêts calculés tous les 30 jours, le premier de chaque mois. Les intérêts se cumulent.

Ecrivez l’analyse descendante, l’algorithme **ou** l’organigramme puis le code C permettant d’afficher le solde de votre épargne au bout d’un nombre de mois saisi par l’utilisateur.

# Chapitre 14 - Les sous-programmes en langage C

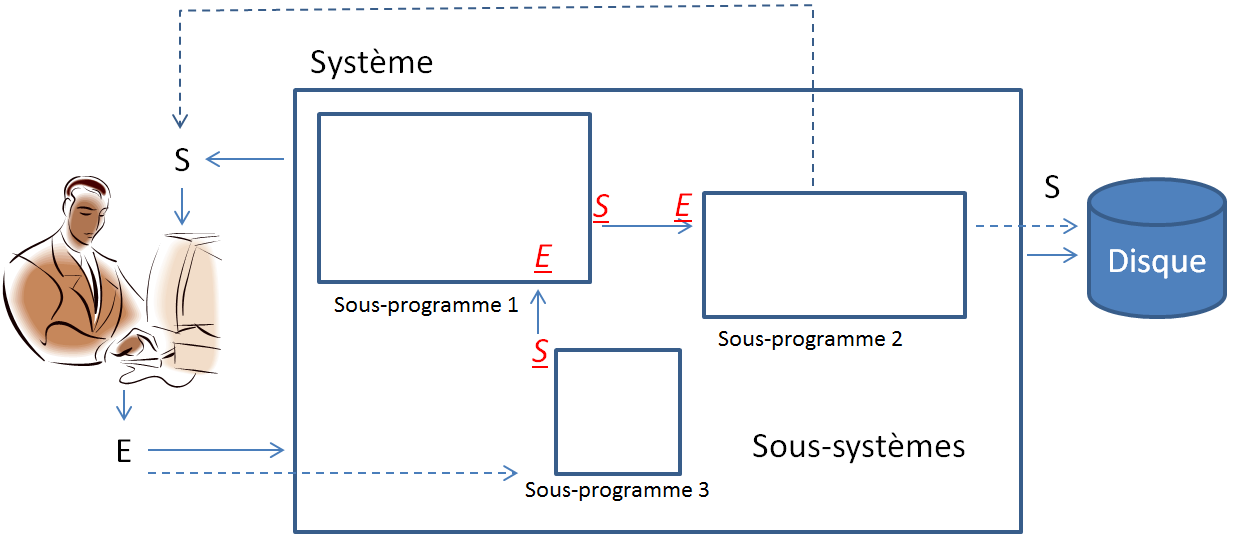
Modularité du code

Un programme bien écrit doit être structuré et décomposé en sous-programmes (ou modules, ou sous-systèmes), chacun associé à une tâche précise. L’Analyse Chronologique Descendante, aura permis d’identifier ces étapes.

Les sous-programmes sont donc des outils d’organisation et de structuration du code.

Ils facilitent l’exécution de tâches redondantes par le principe d’appel des modules correspondants.

Le schéma ci-dessous montre l’organisation d’un programme (système) complexe en sous-programmes (sous-systèmes), les E/S externes au système (en noir) (qui peuvent être fournit par les sous-programmes) et les E/S internes (en rouge, souligné) au système qui permettent l’échange d’informations entre les sous-programmes.



* + 1. Procédure ou fonction ?

Bien qu’en C tout soit fonction, on distingue en général deux types de sous-programme : Les **procédures** et les **fonctions**. Les procédures communiquent avec les autres sous-programmes via leur zone de paramètres (ou arguments) que l’on appellera le **tube**. Les fonctions disposent également du mécanisme de tube, mais profite, en plus, d’un **mécanisme** **de retour** leur permettant de retourner **une et une seule valeur de type scalaire** au sous-programme qui a fait appel à elles (qu’on appelle « l’appelant »).

Nous venons de le dire, le langage C ne fait pas la distinction entre les procédures et les fonctions. Pour ce langage, tout est considéré comme une fonction. Une procédure sera alors vue comme une fonction qui fait son traitement mais ne retourne rien. La non-utilisation du mécanisme de retour se traduira par le mot clef **void** placé avant le nom du sous-programme.

Le **programme principal** est une fonction particulière (le plus souvent de type *int*, d’où le *return 0 ;)* qui doit obligatoirement figurer dans tout programme car il en constitue le **point d’entrée** appelé par le **système** **d’exploitation**.

* + 1. Mécanisme d’appel

Pour qu’un sous-programme soit rendu actif et exécuté (c'est-à-dire qu’il prenne le contrôle du processeur pour exécuter ses tâches), il doit être appelé par un autre sous-programme. On distinguera alors le sous-programme **appelé** et le sous-programme **appelant.** L’appelant enregistre l’adresse de sa prochaine instruction à exécuter dans une pile d’adresse spécifique puis libère le processeur au profit de l’appelé. Après exécution, l’appelé rend le processeur à l’appelant qui dépile son adresse de retour (ou de reprise) et continue alors son exécution là où il s’était arrêté.

Visibilité des sous-programmes

Le compilateur n’autorise l’appel d’un sous-programme que s’il en connaît déjà l’existence ainsi que le nombre et le type des paramètres et de la valeur de retour dans le cas d’une fonction.

void paf()

{

}

void pif()

{

paf() ;

}

void main()

{

pif() ;

paf() ;

}

Dans l’exemple de code précédent, le programme principal « connait l’existence » des 2 sous-programmes pif et paf.

Mais pif ne « connait l’existence » que de paf, et paf ne connait l’existence d’aucun sous-programme.

Paf ne peut donc « appeler » aucun des sous-programmes écrits dans ce code.

**Cela justifie que le « main » soit toujours placé tout en bas du code.**

**Problème** : lorsque le sous-programme n’a pas été défini dans le même fichier que le programme appelant ni dans un fichier d’entêtes « .h » que l’on peut inclure.

**Une solution :** l’utilisation des prototypes.

* + 1. Prototype d’un sous-programme

Le prototype d’un sous-programme contient les informations suivantes :

* le type de valeur retournée dans le cas d’une fonction
* le nom du sous-programme
* la liste des paramètres du sous-programme sous forme : type [+ nom] (optionnel)

**Exemple** : pour la fonction int minimum (int A, int B)

{

…

}

Le prototype est  **int minimum (int val1, int val2) ;**

Comme toute déclaration, le prototype est suivi d’un point-virgule.

Le prototype permet au compilateur de vérifier la validité des appels ultérieurs au sous-programme dans le programme ; lorsque le sous-programme n’a pas été écrit dans le même fichier ou lors d’appels croisés entre deux sous-programmes.

En règle générale, les prototypes de fonctions et de procédures sont écrits dans un fichier d’entêtes avec extension « .h » qui sera inclus dans les fichiers manipulant les sous-programmes.

Nommer les variables dans le prototype est optionnel mais il correspond à une bonne pratique qui permet de comprendre le rôle des paramètres à la relecture des fichiers .h

* + 1. Déclaration en fichier bibliothèque .h

Les fichiers d’entêtes d’extension **.h** sont destinés à contenir des lignes de code à inclure dans un autre fichier source au moment de la compilation.

L’instruction d’inclusion sera par exemple : **#include ″mabib.h ″**

Remarque :

Notez la différence avec les bibliothèques de fonctions prédéfinies offertes par le langage que vous allez inclure en écrivant : **#include <stdio.h>**

Mettre dans ces fichiers les prototypes de vos fonctions ainsi que la déclaration de vos constantes permet de les rendre accessibles depuis tous les programmes C qui auront inclus la bibliothèque. Exactement comme vous faisiez pour profiter des fonctions prédéfinies du langage.

Les procédures : Le type void

On dira d’un sous-programme qu’il est de type *void* si il n’utilise pas son mécanisme de retour. On peut alors le considérer comme un équivalent d’une procédure en algorithmique. Il ne peut alors pas contenir le mot clef ***return*** puisqu’il ne retourne aucun résultat. *(Très exceptionnellement, dans une mauvaise programmation ;-), dans le cas où on souhaiterait malgré tout utiliser l’instruction return pour sortir du sous-programme, celle-ci sera appelée sans paramètre)*.

Une procédure est constituée : - d’une **entête de déclaration**

- d’un corps de procédure

**Exemple** :

void affiche(int val) ← entête

{

printf(« la valeur du paramètre est %d. », val) ; ← corps

}

L’entête de la procédure contient :

* Le mot clef void et le nom de la procédure
* La déclaration des types et des noms des paramètres formels entre parenthèses

Les fonctions : Le mécanisme de retour

Une fonction est constituée : - d’une **entête de déclaration**

- d’un corps de fonction

**Exemple :**

int add (int a, int b) ← entête

{

int c ;

c = a+b ; ← corps

return c ;

}

L’entête de la fonction contient :

* La déclaration du type et du nom de la fonction
* La déclaration des types et des noms des paramètres formels entre parenthèses

Le corps de la fonction est un bloc encadré par des accolades qui contient :

* La déclaration des **variables** **locales** au bloc constitué par la fonction
* Les instructions du bloc
* L’instruction particulière **return** provoquant la sortie et permettant de transmettre à l’appelant le résultat d’exécution de la fonction. **La valeur retournée doit être du type déclaré dans l’entête de la fonction.**

Appel des sous-programmes

* + 1. Les fonctionnalités prédéfinies du langage

Heureusement pour le programmeur, toutes les fonctionnalités n’ont pas à être redéveloppées. Lorsqu’on installe un compilateur sur une machine, ce dernier est livré avec un ensemble de librairies (ou bibliothèques de fonctions) qui contiennent un grand nombre de traitement prédéfinis.

Ces librairies (souvent .lib ou .a) sont associées à des fichiers d’entêtes dont l’extension .h en C. Vous pouvez les ouvrir à l’aide du bloc-notes pour en visualiser le contenu.

Pour pouvoir utiliser ces fonctionnalités, il est nécessaire d’inclure les librairies dont nous allons avoir besoin. Comme il a été décrit ci-dessus, cette inclusion se fait par l’instruction précompilée #include…

Dans les premiers programmes vus dans ce cours, nous avons fait des affichages de texte et de données à l’écran en utilisant la fonction « printf ». Le prototype de cette fonction se trouve dans le fichier d’entêtes « stdio.h », d’où le #include <stdio.h>

* + 1. Nos propres fonctionnalités

Pour compléter les fonctionnalités prédéfinies et coder nos propres traitements, nous avons nous aussi codé des procédures et des fonctions. Ces modules pourront être prototypés dans fichier d’entête .h « personnelle » que nous pouvons créer pour compléter celles du compilateur. Les librairies « personnelles » devront également être incluses dans nos fichiers .c nécessitant l’utilisation des fonctionnalités qu’elles renferment.

Une fois incluses les librairies « personnelles » (avec des " ") et les librairies « standards » (avec des < >), nous pouvons exécuter les fonctionnalités voulues en les « **appelant »**.

Appeler un sous-programme (une fonctionnalité) revient à écrire son nom dans le code et à préciser les paramètres éventuellement nécessaires dans les parenthèses qui suivent ce nom.

**Exemple** : printf (« il fait beau ») ;

* + 1. Appel de fonction, appel de procédure

Supposons que nous ayons écrit la fonction « add » qui réalise une addition et la procédure « affiche » qui affiche une valeur à l’écran.

L’appel de la fonction au niveau du sous-programme appelant doit prévoir la récupération de la valeur retournée par la fonction dans une variable de type approprié.

res = add ( 3, 4) ;

Dans le cas d’une fonction de type void, le mécanisme de retour de valeur n’existe pas.

affiche (val) ;

Graphes d’appel

Ce graphe décrit l’ordre des appels de sous-programmes. Il permet de vérifier la cohérence des enchaînements de procédures et de fonctions.

Main

Proc1

Proc2

Fonc7

Main

Fonc3

Proc9

Proc6

tube :

int A, int B

retour :

int res

tube :

float val

retour:

int erreur

**Efforcez-vous dès maintenant de structurer votre code en le modularisant à l’aide des sous-programmes. Utilisez l’ACD pour déterminer les tâches élémentaires à accomplir et l’ordre d’exécution des celles-ci et surtout constituer la trame de commentaires obligatoires à la bonne compréhension de votre programme. Faites de chacune de ces tâches un sous-programme et dessinez le ou les graphes(s) d’appels pour faciliter la compréhension de votre organisation.**

## Exercices d’application

Rappels

Lorsqu’un sous-programme (ou le main) appelle un autre sous-programme en lui transmettant des informations (des variables) par le tube, ce dernier joue son rôle de duplicateur et crée une copie de chacun des paramètres pour le sous-programme appelé. Ce mécanisme de création de copies a pour but de protéger le contenu des originaux. Donc si vous transmettez le contenu d’une variable par le tube, ce contenu est dupliqué dans une autre zone mémoire appartenant au sous-programme appelé. La modification de cette copie n’aura pas de répercussion sur l’original appartenant à l’appelant.

On appelle ce mode de passage de paramètre, un **passage par valeur**.

Cependant, lorsque l’on veut que la modification du contenu de la copie par le sous-programme appelé se répercute sur l’original détenu par l’appelant, il est possible de faire un autre mode de passage de paramètre : un **passage de paramètre par adresse**. Il consiste cette fois, non pas à passer le contenu mais l’adresse de la variable par le tube. Celui-ci joue alors son rôle de duplicateur et crée une copie de ce paramètre pour l’appelé qui dispose alors d’une clef d’accès (un pointeur) à la zone mémoire de l’appelant. L’appelé dispose d’une indirection vers la variable de l’appelant. Cette indirection lui permet de modifier le contenu de la zone mémoire de l’appelant (en utilisant l’opérateur « \* » - contenu pointé par - devant le pointeur).

POUR VOUS ENTRAINER

* + 1. Exercice 1401

Ecrire le sous-programme adapté à l’affichage d’un menu utilisateur, à la récupération et à la validation (blindage) du choix de ce dernier. Ecrire le programme principal appelant ce sous-programme.

* + 1. Exercice 1402

Ecrire le programme (modulaire !) qui affiche le carré d’un entier saisi au clavier.

* + 1. Exercice 1403

Ecrire le programme qui affiche le maximum d’une série de nombres flottants positifs saisis au clavier (sans tableau).

* + 1. Exercice 1404

Ecrire le sous-programme qui complète l’exercice 3 en permettant l’affichage du nombre de valeurs saisies à l’exercice précédent.

* + 1. Exercice 1405
* Qu’est-ce qui différencie procédure et fonction ?
* A quoi sert le tube ?
  + 1. Exercice 1406

Ecrire un sous-programme recevant 2 entiers x1 et x2 par valeur. Le sous-programme affiche la valeur des paramètres reçus, ajoute **10** à l’entier x1 puis affiche de nouveau la valeur des paramètres reçus.

Ecrire ensuite le main qui demande à l’utilisateur la saisie de 2 entiers val1 et val2, puis affiche ces valeurs, appelle le sous-programme précédent puis les affiche de nouveau après l’appel pour vérifier leur contenu.

* Que constatez-vous au sujet de val1 et val2 ?
* Qu’aurait-il fallu faire pour que la modification effectuée dans le sous-programme se répercute au niveau du main ?

Faites la modification proposée et vérifiez son efficacité.

# Chapitre 15 - Données de type complexe : Les tableaux

Nous avons vu les types de données scalaires. Ces derniers ne sont pas suffisants pour stocker toutes les natures de données que les programmeurs sont amenés à manipuler.

Les types complexes ou structurés permettent de pallier ce manque. Ils permettent notamment de **regrouper les briques de base** que constituent les données scalaires, de manière **homogène ou hétérogène,** c’est-à-dire en ne contenant que des données de même type ou au contraire des données de types différents.

Les tableaux, qui représentent le premier type complexe que nous utiliserons, sont constitués de case (ou zone mémoire) **consécutives (c**e détail aura son importance, retenez-le) contenant des données de même type, donc homogène.

Les tableaux monodimensionnels

Pour réserver un nombre donné d’éléments de même type il est possible de définir un tableau de la manière suivante :

int toto[10] ; réserve un emplacement mémoire permettant de stocker 10 entiers numérotés de 0 à 9. Attention ! Chaque case contient ce qui trainait dans la mémoire de l’ordinateur.

int jours[12] = {31,28,31,30,31,30,31,31,30,31,30,31};

réserve un emplacement mémoire permettant de stocker 12 entiers numérotés de 0 à 11 et initialise ces 12 entiers avec les valeurs entre accolades. Entier0 = 31, entier1 = 28…

int day[10] = {0} ; idem à précédemment avec la valeur 0 dans toutes les cases.

char tab\_car[50] ; réserve un emplacement mémoire permettant de stocker 50 caractères numérotés de 0 à 49.

* + 1. Accès

toto[8] = -32 ; ou toto[i] = 23 ;

L’accès à chaque élément se fait ainsi :

i = 5 ; // par exemple

tab\_car[i] = ‘a’ ;

printf(« %c », tab\_car[i]) ; // **i** représente l’indice de la case

i++ ; // passage à la case suivante

scanf(« %c », &(tab\_car[i])) ; // **i** représente l’indice de la case

printf(« %c », tab\_car[i]) ; // **i** représente l’indice de la case

…

* + 1. Schéma de la représentation classique

int toto [10] ;

toto

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ? | ? | ? | ? | ? | ? | ? | ? | ? | ? |

0 1 2 3 4 5 6 7 8 9

Un pointeur caché dans un tableau monodimensionnel

* + 1. Principe d’adressage

La définition automatique d’un tableau engendre un pointeur caché.

*int tab[25] ;*

|  |  |  |
| --- | --- | --- |
| tab | → | tab[0] |
|  |  | tab[1] |
|  |  | tab[2] |
|  |  | tab[…] |

*Tab* contient donc l’adresse du tableau. Il est équivalent à l’écriture *&(tab[0])* qui correspond à l’adresse du premier élément.

Les tableaux multidimensionnels

* + 1. Schéma de la représentation classique

**Exemple 1** : int tab\_bis[4][3] ;

|  |  |  |  |
| --- | --- | --- | --- |
|  | 0 | 1 | 2 |
| 0 | int | int |  |
| 1 |  |  |  |
| 2 |  |  |  |
| 3 |  |  |  |

**Exemple** 2 : int mat[10][30] ;

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | ... |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 0 | int | int |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 5 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 6 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 7 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 8 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 9 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

* + 1. Principe d’adressage

La réalité de l’organisation des données en mémoire est plus complexe qu’on ne l’imagine dans l représentation classique. La machine gère un tableau à 2 dimensions comme un ensemble de vecteurs à une dimension. Elle stocke l’adresse de chaque vecteur dans un tableau de pointeurs.

Nous obtenons l’organisation mémoire comme décrit dans le schéma ci-dessous.

**Exemple 1** : int tab\_bis[4][3] ;

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| tab | → | tab[0] | → | tab[0][0] | tab[0][1] | tab[0][2] |
|  |  | tab[1] | → | tab[1][0] | tab[1][1] | tab[1][2] |
|  |  | tab[2] | → | tab[2][0] | tab[2][1] | tab[2][2] |
|  |  | tab[…] | → | tab[3][0] | tab[3][1] | tab[3][2] |

**Exemple** 2 : int mat[10][30] ;

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| mat |  |  | mat[i] |  | 0 | 1 | 2 | 3 | 4 | … |
| pointeur  int \*\* | 🡪 | 0 | pointeur  int \* | 🡪 | int | int |  |  |  |  |
|  |  | 1 |  | 🡪 |  |  |  |  |  |  |
|  |  | 2 |  | 🡪 |  |  |  |  |  |  |
|  |  | 3 |  | 🡪 |  |  |  |  |  |  |
|  |  | … |  | 🡪 |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |

* + 1. Accès

mat[8][25] = -32 ; mat[ i ][ j ] = 23 ;

* + 1. Simulation d’une matrice en tableau monodimensionnel

Il est toujours possible de gérer un tableau multidimensionnel comme un vecteur simple (ou tableau monodimensionnel). Cette gestion nécessite alors un calcul du type *( mat[ i\*LARGEUR + j ] )* pour retrouver la bonne case *mat[ i ][ j ].*

**! Attention : Les tableaux ne peuvent contenir qu’un seul type de données !**

## Exercices d’application

POUR VOUS ENTRAINER

* + 1. Exercice 1501

Remplir un tableau de 10 entiers saisis par l’utilisateur.

* + 1. Exercice 1502

Réalisez un programme permettant la saisie d’un tableau de nombres entiers et réalisant ensuite un tri sur ce tableau. Le programme principal se chargera de l’affichage du tableau avant et après le tri.

* + 1. Exercice 1503

Ecrire un programme permettant de remplir un tableau de 10 entiers à partir des saisies de l’utilisateur. L’utilisateur pourra arrêter sa saisie avant 10 nombres en entrant un nombre négatif.

* + 1. Exercice 1504

Ecrire un programme permettant de recopier un tableau déjà rempli dans un autre de même dimension.

* + 1. Exercice 1505

Dessiner la représentation schématique d’un tableau défini comme suit :

float tab[3][5] ;

* + 1. Exercice 1506

Même exercice que précédemment, mais cette fois dans un tableau à 3 dimensions : float tab[3][4][2] ;

* Nombre de lignes 3
* Nombre de colonnes 4
* Profondeur 2
  + 1. Exercice 1507

1. Ecrivez l’algorithme permettant de saisir puis afficher un tableau de 10 nombres entiers.

2. Ecrivez l’analyse descendante puis l’algorithme ou l’organigramme et le code C de la recherche du minimum et du maximum de ce tableau.

3. Ecrivez l’analyse descendante puis l’algorithme ou l’organigramme puis le code en C du tri en ordre croissant de ce tableau.

* + 1. Exercice 1508

Saisissez, affichez et recherchez le minimum et le maximum, mais cette fois dans un tableau à 3 dimensions : int toto[3][4][2] ;

**Passage de paramètres et tableaux**

**Faites ces étapes les unes après les autres dans le même projet, le main étant la dernière étape. Vous respecterez ainsi les étapes logiques de l’écriture d’un code structuré.**

* + 1. Exercice 1509

1. Ecrire un sous-programme permettant le remplissage d’un tableau de 10 notes (blindées entre 0 et 20). Le tableau sera reçu en paramètre par le tube.
2. Ecrire un sous-programme permettant l’affichage d’un tableau de 10 notes. Le tableau sera reçu en paramètre par le tube.
3. En appelant les sous-programmes précédant, écrire un programme permettant la saisie et l’affichage de 10 notes stockées dans un tableau et calculant dans un nouveau sous-programme la moyenne de ces 10 notes. L’affichage de cette moyenne sera fait dans le sous-programme.
4. Quelle serait la conséquence d’un affichage de cette moyenne dans le main ? Ecrivez le nouveau sous-programme correspondant à ce nouveau cahier des charges (sans supprimer l’ancien).
5. Ecrire un sous-programme permettant la recherche du minimum dans un tableau de 10 notes.
6. Ecrire un sous-programme permettant la recherche du maximum dans un tableau de 10 notes.
7. Ecrire le main tel qu’il joue son rôle de chef d’orchestre.
8. Vous dessinerez le graphe d’appel de votre programme en faisant apparaître le fonctionnement des tubes et des mécanismes de retour.
9. (Facultatif) Ecrire et inclure à votre projet la bibliothèque personnelle au format .h

# Chapitre 16 - Données complexes : les chaînes de caractères

Les tableaux monodimensionnels, type complexe de données homogène, peuvent être détournés pour accueillir un autre type complexe : **la chaîne de caractères**.

Composée uniquement d’un ensemble de caractères de la table ASCII (donc homogène), la chaîne de caractère va contenir (et surtout, se terminer par) un caractère spécial (\0) sur lequel va s’appuyer tout le fonctionnement spécifique des chaînes de caractères.

Convention

Le C n'a pas, contrairement à d'autres langages, de type spécifique pour manipuler les chaînes de caractères. Le type string n’existe pas en natif.

Une chaîne de caractères est gérée en langage C comme un **tableau contenant des caractères** mais avec la particularité que la case du tableau juste après la dernière case utilisée pour la chaîne, contient le **caractère spécial ‘\0’**. Ce caractère représente la fin de la chaîne. Il n’est pas pris en compte par la fonction **strlen(chaine)** qui compte le nombre de caractères « utiles » d’une chaîne.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| b | o | n | j | o | u | r | ? | ? | ? |

Tableau de caractères **ne pouvant pas** être considéré comme une chaîne

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| b | o | n | j | o | u | r | \0 | ? | ? |

Tableau de caractères pouvant être considéré comme une chaîne car terminé par \0

**Exemple** : lg = strlen(« bonjour ») renvoie 7 dans lg. La fonction strlen ne compte pas le \0

On manipule ensuite les chaînes à l'aide d'un pointeur qui contient son adresse, ce qui permet d'accéder à la chaîne en entier. Ainsi, toutes les fonctions C servant à manipuler les chaînes de caractères utilisent comme arguments des pointeurs sur des variables de type char. Ces fonctions sont définies dans le fichier en-tête string.h. Le paragraphe 5 de ce chapitre en présente une sélection non-exhaustive.

Déclaration d’une chaîne

Rappelons-le, une chaîne en C n’est autre qu’un tableau de caractères.

Une chaîne de caractère appelée « toto » peut être réservée *en dur* (en automatique) par l’instruction suivante :

**char toto[50] ;** Elle est dans ce cas limitée à 49 caractères (+ ’\0’) et toto (le nom de la chaîne) est en fait un « pointeur caché » de type char\* (voir la notion de pointeur caché dans le chapitre précédent sur les tableaux).

Vous reconnaîtrez ici la déclaration d’un tableau de caractères. En effet, une chaîne de caractères et un tableau de caractères ne se différencient pas par leur déclaration mais par leur contenu (un ‘\0’ à la fin d’une chaîne, pas de ‘\0’ dans un tableau).

**Notez qu’une chaîne de caractères se représente entre guillemets « » tandis qu’un caractère seul se représente entre quote ‘ ’. Et ne confondez pas le « nom » de la chaîne et son « contenu ».**

« a » »

a

‘a’

a

\o

Saisie

Pour stocker une chaîne de caractères il faut réserver un emplacement mémoire de réception. Cet emplacement se déclare comme un tableau de caractères en ajoutant une case pour le caractère fin de chaîne ‘\0’.

* + 1. Plusieurs fonctions de saisie possibles

La lecture d’une chaîne au clavier peut se faire à l’aide d’un **scanf(« %s », ch) ;**

La fonction **scanf** possède un format adapté à la saisie des chaînes de caractères : « **%s** ».

Notez qu’il n’y a pas de **&** devant la variable **ch** car celle-ci est un pointeur qui contient déjà l’adresse de la zone de stockage de la chaîne.

Le défaut de cette fonction est de couper la récupération du texte saisi au premier espace. Cela peut néanmoins être utile pour séparer les différents mots d’une phrase. La fonction **gets(ch)** corrige ce problème et permet de saisir une chaîne jusqu’au « enter ».

char ch[100] ;

gets(ch) ;

Ces deux fonctions, scanf et gets, se chargent de positionner le ‘\0’ dans le tableau de stockage en mémoire.

* + 1. Problème d’espace mémoire : la zone de stockage doit avoir été réservée.

**Attention !** char \*ch ; // ici il n’y a aucune mémoire réservée, juste un pointeur

gets(ch) ;

**N’est pas correct** car vous n’avez pas réservé de place mémoire pour le stockage de votre chaîne.

En fonction de la valeur de votre pointeur vous obtiendrez **une erreur d’exécution** ou vous abîmerez la mémoire associée à vos autres variables.

Affichage à l’écran : printf(« %s », ch) ;

Le format d’affichage d’une chaîne par un **printf** est « %s ». La fonction printf affiche à l’écran les caractères contenus en mémoire à partir de l’adresse de début de la chaîne et s’arrête lorsqu’elle rencontre le caractère ‘\0’.

En l’absence de celui-ci (dans le cas d’un printf(« %s »…) appliqué sur un tableau sans ‘\0’ par exemple), la fonction interprète chaque octet de la mémoire comme un caractère (code ASCII) et ne s’arrête pas, ce qui provoque un affichage incohérent.

La fonction **puts** permet d’obtenir un résultat identique au printf.

Autres fonctions dédiées

**Ces fonctions se trouvent dans la bibliothèque standard STRING.H.** Elles s’appuient toutes sur la présence du caractère fin de chaîne ‘\0’. *Soit la déclaration char ch[100] ;*

* + 1. Affectation

Après la déclaration, l’affectation d’une chaîne se fera par la fonction STRCPY : strcpy(ch, « hello ») ;

* + 1. Concaténation

La concaténation de deux chaînes se fait par la fonction STRCAT : strcat(ch, « world ») ;

la chaîne ch contient alors *hello world*

* + 1. Comparaison

La comparaison de deux chaînes se fait par la fonction STRCMP : res = strcmp(chaine1, chaine2) ;

**Res** contiendra :

- un nombre négatif si chaine1 avant chaine2 au sens lexicographique (dans un dictionnaire)

- 0 si chaine1 et chaine2 sont identiques

- un nombre positif si chaine1 après chaine2

* + 1. Dimension

La fonction STRLEN retourne le nombre de caractères d’une chaîne sans compter le ‘\0’.

**Exemple** : strcpy(toto, « coucou ») ;

printf(« %d »,strlen(toto)) ; affichera 6 à l’écran.

Les fonctions de manipulation de chaînes prennent en général pour paramètre **l’adresse** des chaînes à traiter. Ces adresses seront stockées dans **des pointeurs**.

## Exercices d’application

Rappels

Les chaînes de caractères sont des tableaux de caractères dont la particularité est d’avoir, après la dernière case de la zone dite « utile » le caractère ‘\0’ qui marque la fin de la chaîne.

Ce caractère spécifique du langage C n’est pas pris en compte par la fonction strlen() mais doit être pris en compte lorsque vous déclarez votre chaîne de caractères en ajoutant 1 à la taille « utile » de la chaîne.

* + 1. Exercice 1601

Ecrivez le modèle puis le code d’un programme C affichant un menu utilisateur proposant :

* de saisir une chaîne de caractères
* de convertir la chaîne saisie en majuscules (vous ne modifierez que les caractères minuscules et ne toucherez pas aux autres
* de convertir la chaîne saisie en minuscules
* de comparer 2 chaînes à saisir (utilisez strcmp)
* de compter la longueur d'une chaîne
* de concaténer 2 chaînes dans la première (attention aux débordements)
* de crypter une chaîne en appliquant un décalage circulaire dont la valeur est entrée par l’utilisateur
* de décrypter une chaîne de caractère cryptée en appliquant un décalage circulaire inverse à partir d’une valeur entrée par l’utilisateur
* De compter le nombre de voyelles dans une chaîne

Chaque fonctionnalité sera gérée dans un sous-programme.

Vous blinderez votre programme pour garantir que vous travaillez sur des chaînes non vides et redemanderez la saisie dans le cas contraire.

Vous ferez attention aux débordements de tableau.

**Passage de paramètres et chaînes de caractères**

* + 1. Exercice 1602

Écrivez le modèle, l’ACD puis le code C d’un programme dont le main affiche un menu permettant les actions décrites dans chacun des exercices suivants :

*(Attention, vous blinderez les choix des utilisateurs dont l'ordre serait incohérent, ou encore vous proposerez d’autres solutions pour maintenir la cohérence des choix)*

1. L'utilisateur peut saisir une phrase contenant espaces et ponctuation. La phrase sera affichée pour valider la saisie.
2. Le sous-programme "voyelle" reçoit une phrase quelconque en paramètre et retourne le nombre de voyelles qu'elle contient. L'appelant affiche ce nombre.
3. Le sous-programme « miroir » permet de concaténer à la phrase saisie au travers de l’option de l’exercice 1 son équivalent inversé avec un ‘ : ’ en séparateur. Voici un exemple : si la phrase saisie en 1 est « il fait beau », le résultat de cette option sera « il fait beau : uaeb tiaf li ». Ici, le sous-programme reçoit un pointeur sur une phrase et retourne le pointeur sur une nouvelle phrase allouée dynamiquement. L’appelant affiche la phrase résultat.
4. Le sous-programme "consonne" reçoit une phrase quelconque en paramètre et retourne le nombre de consonnes qu'elle contient. L'appelant affiche ce nombre.

# Chapitre 17 - Les structures et les unions

Autre type complexe, cette fois **hétérogène**, les structures et les unions offrent un mode de stockage des données plus adaptées aux programmes de grande envergure et surtout à une approche objet de la solution à développer.

Les Structures

Il arrive très souvent qu'un objet manipulé par le programme ne se décrive pas seulement par un entier, un réel ou une autre valeur simple mais que sa description demande plusieurs informations de type différent. Par exemple, pour décrire une personne on utilisera des chaînes de caractères (pour le nom et le prénom), des entiers (pour sa date de naissance), …

Si on a plusieurs individus, on peut regrouper les données par type dans des tableaux homogènes, les informations d’un même individu étant alors rangées au même indice de tous les tableaux. Mais il peut être aussi intéressant d'avoir des mécanismes permettant de regrouper un certain nombre de variables de type différents. C'est le concept de structure.

Une structure est un ensemble de données regroupées sous un même nom. Les éléments d'une structure, ou membre d'une structure, peuvent être de types différents. (tableau, autre structure, etc.....).

Cette représentation des données (cette structure de données) est plus logique, plus naturelle et plus proche de la conception objet.

**Rappel sur le typedef :**

Nous avons déjà eu l’occasion de voir le mot clef **typedef** qui s’utilise en bibliothèque personnelle et permet de se définir de nouveau type.

**typedef description du type nom\_type ;**

**Attention :** typedef ne réserve pas d'espace mémoire. Le nom est un type, il est donc inaccessible comme variable.

* + 1. Déclaration d’un type structure

La déclaration d'un type de structure se fait dans la bibliothèque personnelle comme suit :

typedef struct nomDeStructure

{

type1 nomChamp1;

type2 nomChamp2;

type3 nomChamp3;

type4 nomChamp4;

..........

typeN nomChampN;

}nom\_du\_type;

**Exemple :**

typedef struct personne

{

char nom[30];

char prenom[40];

Synonyme (alias)

int age;

char sexe;

}t\_pers;

Un type de structure est donc déclaré par le mot clé « typedef » (définition de type) suivi de "struct"suivi d'un nom facultatif*,* ici "personne"*.* Ce nom est une étiquette qui pourra être ensuite utilisée pour désigner le type de structure. Le nom de type « t\_pers » est un alias du type « struct personne » décrit entre accolades. Il permet de faire plus simplement référence au type « struct personne ».

Cette déclaration ne provoque pas d'allocation mémoire; il s'agit uniquement de la définition d'un nouveau type (aussi appelé nouveau concept) dont le nom est « struct personne » ou encore « t\_pers ».

Une fois le type « abstrait » t\_pers défini, il nous faut désormais réserver de la mémoire pour stocker nos données. On va donc faire une déclaration de variables.

* + 1. Déclaration d'une variable de type structure (ne pas confondre)

La déclaration d'une variable de type structure personne ou t\_pers se fait ainsi :

struct personne toto1;

ou encore pour une deuxième variable :

t\_pers toto2 ; // t\_pers est un alias de struct personne, son utilisation remplace

// struc personne

Cela permet de créer une variable (on dit aussi instance de type ou objet) nommée toto1 ou toto2 de type "structure personne" précédemment définie. Cette fois ci, contrairement au point 16.1.1, le compilateur alloue de la mémoire pour ces 2 variables.

* + 1. Les données membres d'une structure : les champs

Les membres d'une structure peuvent être de types quelconques, type scalaire de base (char, int, float...), pointeurs, mais aussi tableaux ou encore autres structures. Une structure peut donc comporter plusieurs niveaux hiérarchiques de structures.

**Exemple**

typedef struct date

{

int jour;

int mois;

int annee;

}t\_date;

typedef struct adresse

{

int numero;

char rue[40];

long codePostal;

char localite[16];

}t\_adresse;

typedef struct identité

{

char nom[16];

char prenom[20];

struct adresse adr; // utilisation du type struct adresse précédemment défini

t\_date naissance; // utilisation de l’alias du type struct date (plus court :-)

};

Vous noterez dans cet exemple de structures imbriquées que l’ordre de définition des types est important.

* + 1. Accès aux champs d’un objet structure.

L'accès aux membres de structure se fait en utilisant l'opérateur "." :

L'opérateur "." relie le nom d'une variable de type structure à un de ses membres:

nom\_de\_structure. nom\_membre

**Exemple** : Soit le programme suivant

#ifndef COUCOU

#define COUCOU

typedef struct personne

{

char nom[30];

char prenom[40];

int age;

char sexe;

}t\_pers ;

#endif

#include « … »

void main( )

{

/\* déclaration de pers2 \*/

t\_pers pers2 ; // cette variable non-initialisée ne sera pas utilisée dans cet exemple

/\* déclaration et initialisation immédiate de pers1 \*/

t\_pers pers1 = {"DUPONT","RENE",28,'M'};

printf("\n voici les données de pers1 ");

/\* désignation de l'élément "nom" et de l'élément "prenom" de la variable pers1 \*/

printf("\n\n NOM :%s PRENOM : %s", pers1.nom, pers1.prenom);

printf("\n AGE %d", pers1.age);

printf("\n SEXE : %s\n",( pers1.sexe=='M') ? "MASCULIN":"FEMININ");

}

* + 1. Accès aux champs d’un objet structure via un pointeur

On peut utiliser l'opérateur "->" pour accéder aux membres d'une structure par l'intermédiaire d'un pointeur. Cet opérateur relie le pointeur de la structure au membre que l'on désire atteindre :

pointeur\_de\_structure -> nom\_membre

Exemple :

// dans la biblio perso

typedef struct adresse

{

int numero;

char rue[40];

long codePostal;

char localite[16];

} t\_adr;

…

// dans le code C

struct adresse adr1, \*ptad*; // ici adr1 est une variable (un objet)de type t\_adr,*

*// ptad est un pointeur sur variable (objet)de type t\_adr*

**Attention :**

Adr1.numero *est équivalent à* ptad->numero

Adr1.rue[0] *est équivalent à* ptad->rue[0]

(\*ptad).codePostal *est équivalent à* ptad->code\_postal

**Remarque:**

Pour adresser par pointeur le code postal de la structure adresse, les parenthèses sont nécessaires car la priorité de l'opérateur "." est plus élevée que celle de l'opérateur \*.

* + 1. Accès aux données membres de structure dans le cas de structures imbriquées

Dans le cas où plusieurs structures sont imbriquées, l'accès à un membre de structure de bas niveau, se fait en utilisant plusieurs fois l'opérateur "." .

Exemple:

typedef struct date

{

int jour;

int mois;

int annee;

};

typedef struct identite

{

char nom[16];

char prenom[20];

struct date naissance;

} t\_ident;

main ( )

{

t\_ident homme = {"Durand","Marcel",23,12,1915};

if(2000 - homme.naissance.annee >=18)

printf("%s %s est majeur \n", homme.nom, homme.prenom);

else printf("%s %s n'est pas majeur \n", homme.nom, homme.prenom);

}

* + 1. Opération sur les structures

Les opérateurs classiques (mathématiques, comparaisons…), utilisés pour les variables simples (scalaires), ne s'appliquent pas aux structures. Seul l'opérateur "&" permet d'obtenir l'adresse de tout objet en C et donc d'une structure.

Les comparaisons se feront donc champ par champ.

## Les tableaux de structures

On utilise fréquemment les tableaux de structures; l'instruction ci-dessous permet de déclarer un tableau de 100 structures de type "personne".

struct personne

{

char nom[30];

char prenom[40];

int age;

char sexe;

} tabDePersonnes[100];

L'accès à un élément du tableau peut se faire en utilisant un indice compris entre 0 et 99, ou bien encore un pointeur de tableau.

Exemple:

typedef struct personne

{

char nom[30];

char prenom[40];

int age;

char sexe;

} t\_pers;

main ()

{

t\_pers tabDePersonnes[100];

int i;

struct personne \*ptr = tabDePersonnes;

/\* lecture du nom de chaque personne en utilisant l'indice du tableau \*/

for ( i = 0; i < 100 i++)

scanf("%s", tabDePersonnes[i].nom);

/\* impression du nom de chaque personne en utilisant le pointeur \*/

for ( i = 0; i < 100; i++)

printf ("%s\n",ptr++ -> nom);

}

## Le chaînage de structures

Il est possible de définir un champ à l’intérieur d'une structure pour référencer une autre structure *du même type* à l'aide d'un *pointeur* vers cette structure. **C’est le principe des listes chaînées.**

struct liste {

char nom[16];

char prenom[20];

struct date naissance;

**struct liste \* suiv;**

};

où **suiv** est le pointeur vers une structure de type liste.

**Exemple** : La structure liste déclarée précédemment permet de construire les listes chaînées comme sur le schéma ci-dessous. Chaque élément de cette liste est de type struct liste.

"Ritchie"

"Dennis"

1952

"Kernighan"

"Brian"

1950

Nous reviendrons en détail sur les listes chaînées dans un prochain chapitre.

## Les unions

Une union est une structure de données cousine des structures qui permet de ranger à la même adresse en mémoire des variables de types différents.

Sur le plan syntaxique, la déclaration d'une union est parfaitement identique à celle d'une structure. Cette déclaration permet au compilateur de prendre connaissance des différents types de données que cette union peut contenir. Contrairement à la structure, toutes les données de l'union sont rangées à la même adresse. La taille mémoire (nombre d'octets) réservée par le compilateur est la taille de la donnée la plus grande.

Exemple:

union unionDef

{

char cvar;

int ivar;

long lvar;

double dvar;

} unionVar;

De même que pour les structures, c'est la déclaration d'une variable de type union qui déclenche l'allocation de mémoire par le compilateur. La déclaration d'un type union ne génère pas d'allocation mémoire. Dans l'exemple ci-dessus, le nom du type "unionDef" est facultatif.

unionDef définit un type d'union composée de différents types d'objets; unionVar représente une variable de type unionDef.

Le schéma ci-dessous, nous montre les différences d'implantations en mémoire entre une structure et une union.

|  |  |
| --- | --- |
| struct structDef {  char cvar;  int ivar;  long lvar;  double dvar;  }structVar; | union unionDef {  char cvar;  int ivar;  long lvar;  double dvar;  }unionVar; |



## Exercices d’application

* + 1. Exercice 1701

Fan de téléchargement, vous souhaitez gérer votre vidéothèque en regroupant quelques informations dans un fichier texte.

Ecrivez la définition du type **t\_film** contenant :

* le titre du film
* son année de réalisation
* le réalisateur
* la durée en minutes
* un booléen disant si vous l’avez vu ou pas
* une note sur 10

* + 1. Exercice 1702

Ecrivez un sous-programme chargé de remplir une instance de cette structure dont l’adresse est reçue en paramètre.

# Chapitre 18 – Génération de Nombres Aléatoires

Génération de nombres aléatoires

L’obtention d’un nombre aléatoire en C est un peu **surprenante** !!

La fonction RAND (une des nombreuses existantes) génère en effet, lors de ses utilisations successives dans un même programme, une séquence de nombres ALEATOIRES toujours IDENTIQUES d’une exécution du programme à la suivante !

Pour pallier cet inconvénient, il est indispensable d’initialiser le générateur aléatoire sur l’horloge du système de la machine grâce à la fonction SRAND (pour SEED RAND). Cette initialisation ne devra se faire qu’une seule fois dans le programme, au début du MAIN par exemple si vous savez que votre programme utilisera les nombres aléatoires quelques part.

* + 1. Inclusion des fichiers d’entêtes

#include <stdlib.h>

#include <time.h>

* + 1. Initialisation du générateur aléatoire

Elle se fait par appel de la procédure srand() pour start-rand, une seule fois au début du main comme dans l’exemple ci-dessous. Cette procédure recevra en paramètre la valeur de l’horloge interne récupérée par la fonction time().

* + 1. Génération du nombre par la fonction RAND

int main ()

{

int alea , n; /\* n est la borne MAX du nombre désiré \*/

srand ( time(NULL) ); // ne doit se faire qu’une seule fois dans le // programme principal pour initialiser   
 // le générateur

// puis, a chaque fois qu’un nombre aléatoire est requis…

alea = rand() % n ; // la plus grande valeur générée par rand()

// vaut RAND\_MAX. Avec le modulo, cette

// valeur est ramenée entre 0 et N-1

/\*RAND\_MAX définie dans /usr/include/stdlib.h et vaut (2^15)-1 \*/

return 0 ;

}

La valeur générée sera comprise entre 0 et n-1, bornes comprises.

**ATTENTION !**

**Le *srand (time (NULL)) ;* ne doit se faire que dans le MAIN, même si l’appel à *rand() ;* se fait dans un sous-programme.**

## Exercices d’application

* + 1. Exercice 1801

Dichotomie. Ecrivez l’analyse descendante, l’algorithme, l’organigramme qui génère aléatoirement un nombre que l’utilisateur devra découvrir. A chaque saisie, le programme dira si le nombre est plus petit ou plus grand. Si le nombre est trouvé par l’utilisateur, le programme lui annoncera qu’il a gagné en indiquant le nombre d’essai qui ont été nécessaires.

* + 1. Exercice 1802

Ecrire un programme affichant un menu proposant différents cas de génération de nombres aléatoires.

1. affichage d’un nombre aléatoire entier dans la plage de valeur globale du générateur aléatoire
2. affichage d’un nombre aléatoire entier compris entre 0 et une valeur « seuil haut » saisie par l’utilisateur
3. affichage d’un nombre aléatoire entier compris entre la valeur « seuil bas » et « seuil haut » saisies par l’utilisateur
4. affichage de **n** nombres aléatoires (n saisi par l’utilisateur) entre les seuils bas et haut saisis par l’utilisateur
5. affichage de n nombres aléatoires flottants à deux décimales entre 0 et 1 (bornes comprises).
   * 1. Exercice 1803

Ecrivez un programme qui offre un menu utilisateur permettant :

* De dire si un nombre aléatoire est pair ou impair
* De dire si un nombre aléatoire est positif ou négatif
* De générer un nombre entier entre 13 et 27
* De générer un nombre aléatoire décimal à 2 chiffres après la virgule entre 10,00 et 20,00
* De quitter

**TP FACULTATIF**

# Image, histogramme et binarisation

**A la demande de votre chargé de TP, cette série d’exercices pourra être faite sous forme de mini-projet à faire chez soi, seul ou en binôme.**

L’objectif de la suite de ce TP est de manipuler le tableau de valeurs correspondant à chaque point (appelé PIXEL) d’une image. Vous ne manipulerez pas une vraie image ; n’ayant pas d’outil de visualisation des images en mode console, vous ne pourrez manipuler que des nombres sans visualiser le résultat de vos traitements.

* + 1. Exercice 1105

Ecrire un programme remplissant aléatoirement un tableau à deux dimensions (matrice) de 10 lignes et 20 colonnes avec des valeurs comprises entre 0 et 255 (bornes comprises).

Afficher cette matrice à l’écran en respectant l’aspect rectangulaire de la matrice.

* + 1. Exercice 1106

L’histogramme d’une image étudie la répartition statistique de chaque valeur de niveau de gris dans une image.

Son principe consiste, dans un tableau histogramme de 256 cases de type entier, à compter combien l’image contient de pixels de niveau de gris 0 et à stocker cette valeur dans la case 0 du tableau histogramme, puis combien l’image contient de pixels de valeur 1 à ranger dans la case 1… ainsi de suite jusqu’à compter le nombre de pixels à 255.

**Attention** **!!** Il y a une méthode de calcul rapide et une méthode très lente. Réfléchissez !

Compléter le programme précédent pour qu’il calcule l’histogramme de l’image après son remplissage aléatoire.

Affichez cet histogramme de manière lisible à l’écran.

* + 1. Exercice 1107

La binarisation d’une image consiste, **dans une image secondaire afin de ne pas modifier l’image originale**, à mettre à 0 tous les pixels de l’image originale inférieurs à une valeur seuil entrée par l’utilisateur ; et à mettre à 255 tous les pixels de l’image originale supérieurs ou égaux à cette valeur seuil.

Compléter à nouveau le programme précédent pour y inclure la binarisation et affichez la valeur des pixels de l’image binarisée en respectant l’aspect de l’image.

# Chapitre 19 - Programmation Evénementielle

Evénement, action, démon de surveillance

Contrairement à la programmation « classique », la programmation événementielle est dictée par la détection d’événements déclenchés par l’utilisateur.

Le rôle du programmeur devient alors le suivant :

* Lister les **événements** autorisés
* Développer les **actions** associées à chacun des événements
* « Ecouter » le système en attente de la détection d’un des événements (boucle sans fin appelée **démon de surveillance**) et, au besoin, déclencher l’action associée à l’événement détecté

Vous remarquerez ici l’utilisation d’une boucle sans fin, ce qui est totalement proscrit dans la programmation classique. Mais dans ce cas particulier, la fin du programme n’est pas décidée par le programmeur ; elle est dépendante de l’événement indispensable « fin du programme ». Il est cependant possible de faire de l’événementiel sans boucle sans fin ; en décidant simplement de l’événement terminal.

Il existe des bibliothèques de fonctions écrites en C spécifiquement adaptées à la programmation événementielle. C’est le cas du GTK+.

Toutefois, on peut soi-même écrire ses propres fonctions dédiées pour obtenir ce type de programmation, comme l’illustre l’exemple suivant.

int main ()

{

char key = ‘a’; // initialisation de la variable

// partie initialisation séquentielle du programme ici

…

// partie événementielle ci-dessous

while (key !=’q’) // événement terminal

{

// partie des traitements redondants, vie autonome du programme

…

// si événement…

if (kbhit()) // si une touche est appuyée (un événement)

{

key = getch() ; // récupération du code de la touche

switch(key) // analyse du code de la touche (de l’événement)

{ // et déclenchement de l’action associée

case ‘a’…

case ‘m’…

…

case ‘q’ : printf (“a bientot”);

}

}

}

}

Ici, la fonction kbhit() écoute les événements et renvoie vrai si une touche de clavier a été touchée. Cette fonction renvoie en permanence faux sinon. Attention, kbhit et getch ne sont valables que sous Windows.

## Exercices d’application

Rappels :

*La programmation évènementielle diffère de la programmation séquentielle classique.*

*Au lieu d’exécuter des instructions en séquences les unes après les autres selon un ordre prédéterminé par l’écriture de votre code, vous allez prévoir des bouts de code adapté à chacune des actions autorisées à l’utilisateur et vous mettre en attente d’évènements.*

*Lorsqu’un évènement autorisé sera détecté, vous analyserez cet évènement pour en déterminer la nature et exécuter le morceau de code correspondant à l’action souhaitée par l’utilisateur.*

**PACMAN**

* + 1. Exercice 1901

Ecrivez un programme C affichant un X qui part du bord gauche de l’écran et qui se déplace **seul** jusqu’au bord droit, avant de disparaître.

En mémoire, le X sera stocké dans une matrice de caractères à 2 dimensions 10 lignes x 20 colonnes. La position initiale du X est aléatoire. L’affichage de chaque étape du déplacement du X correspond à l’affichage complet de la matrice après effacement de l’écran.

(Utilisez la fonction system(« cls ») ; ou system(« clear ») ; clrscr() ; (en fonction des systèmes) pour nettoyer l’écran et éviter le déplacement diagonal)

* + 1. Exercice 1902

Améliorez le programme précédent pour que le X réapparaisse à gauche lorsqu’il est sorti à droite.

* + 1. Exercice 1903

Modifiez encore ce programme pour que l’utilisateur puisse cette fois changer la direction de déplacement du X en utilisant les touches 2, 4, 6 et 8 du pavé numérique ( ! Assurez-vous que le verrouillage numérique du pavé soit actif). (Vous pourrez, à votre guise, choisir d’autres touches).

Le X réapparaîtra en haut ou en bas, à gauche ou à droite, à l’opposé de sa sortie de l’écran.

* + 1. Exercice 1904

Ajoutez une bordure à votre cadre de jeu pour observer l’effet stroboscopique à l’écran.

* + 1. Exercice 1905

Pour éviter cet effet clignotant de l’écran, vous pouvez remplacer l’opération d’effacement complet de l’écran suivi du réaffichage complet de la matrice terrain avec ses bordures par une opération de mise à jour des parties mobiles uniquement à l’écran. Les bordures n’étant plus réaffichées à chaque fois, l’effet clignotant disparaît.

**SOUS WINSOWS UNIQUEMENT**, améliorez votre programme en y intégrant la fonction

gotoligcol( lig , col )

ci-dessous qui utilise elle-même des fonctions de positionnement du curseur à l’écran qui se trouvent dans la bibliothèque windows.h (un petit coup de google pour comprendre comment marche cette fonction).

Cette fonction emmène le curseur à la position lig, col à l’écran, le coin haut gauche étant en 0,0. Vous pouvez alors y afficher le caractère de votre choix.

void gotoligcol( int lig, int col )

{

COORD mycoord;

mycoord.X = col;

mycoord.Y = lig;

SetConsoleCursorPosition( GetStdHandle( STD\_OUTPUT\_HANDLE ), mycoord );

}

(sans oublier bien sûr d'inclure windows.h)

* + 1. Exercice 1906

Modifiez encore une fois votre programme pour faire apparaître aléatoirement des objets dans le terrain de jeu et augmenter un score à chaque fois que l’utilisateur « mange » l’objet ; ou encore des ennemis qu’il faut à tout prix éviter.

Pour les plus forts en programmation parmi vous, vous pouvez même faire en sorte que le PACMAN devienne un SNAKE en accolant un nouveau X au serpent à chaque fois que vous réussissez à manger un objet.

Rappels :

(**Aléatoire** : « srand(time(NULL)) ; » à appeler une fois au début du programme puis « rand()%x ; » à chaque fois que vous avez besoin d’un entier entre 0 et x-1, bornes comprises.

Vous ferez attention aux débordements mémoire.

Vous ferez valider chaque étape par votre chargé de TD/TP.

# Chapitre 20 - Allocation dynamique de mémoire

Retour sur la déclaration automatique de données

Revoyons d'abord de quelle façon la mémoire a été réservée dans les programmes que nous avons écrits jusqu'ici.

* + 1. Variables scalaires

Chaque variable dans un programme a besoin d'un certain nombre d'octets en mémoire. Jusqu'ici, la réservation de la mémoire s'est déroulée automatiquement par l'emploi des déclarations automatiques des données. Dans tous ces cas, le nombre d'octets à réserver était déjà connu pendant la compilation.

La déclaration des variables se fait **avant la première instruction**, en tout début de chaque sous-programme. Puisqu’il faut réserver la mémoire de stockage AVANT de traiter les données. Nous parlons alors de la déclaration automatique des variables.

***Exemples***

float A, B, C; /\* réservation de 3x4 = 12 octets \*/

* + 1. Tableaux

De même, les déclarations de tableaux vues dans les chapitres précédents ne sont valables que si on connaît par avance le nombre de cases du tableau.

int D[10][20]; /\* réservation de 400 octets \*/

char E[10]; /\* réservation de 10 octets \*/

* + 1. Pointeurs

Le nombre d'octets « p » à réserver pour un pointeur dépend de la machine et du 'modèle' de mémoire choisi, mais il est déjà connu lors de la compilation **et correspond en principe au nombre d’octets nécessaire pour le codage d’un entier**. Un pointeur est donc aussi déclaré automatiquement. Supposons dans la suite qu'un pointeur ait besoin de p octets en mémoire.

***Exemples***

double \*G; /\* réserv. de p octets pour un pointeur sur double\*/

char \*H; /\* réserv. de p octets pour un pointeur sur char \*/

float \*I[10]; /\* réserv. de 10\*p octets pour un tableau de 10 pointeurs sur flottant\*/

* + 1. Chaînes de caractères constantes

L'espace pour les chaînes de caractères constantes qui sont affectées à des pointeurs ou utilisées pour initialiser des pointeurs sur char est aussi réservé automatiquement:

***Exemples***

char \*J = "Bonjour !";

/\* réservation de p+10 octets \*/

Problème et solution

Si le type de la variable, le nombre de variables ou encore le nombre de cases d’un tableau ne sont pas connus au moment de la déclaration des ressources mais sont des informations saisies par l’utilisateur de votre programme en cours d’exécution, il n’est alors pas possible de faire une réservation de place par une déclaration automatique en début de code.

Dans le cas du tableau, vous pouvez bien sûr réserver un tableau de taille maximale très importante (tableau surdimensionné) même si l’utilisateur n’en exploite qu’une petite partie. Vous risquez alors d’occuper beaucoup de mémoire pour rien.

Il existe une solution plus adaptée qui consiste à faire une allocation dynamique de mémoire.

Si nous générons ces données pendant l'exécution du programme, il nous faut des moyens pour réserver et libérer de la mémoire au fur et à mesure que nous en avons besoin. Nous parlons alors de l'allocation dynamique de la mémoire.

## Principe de l’allocation

* + 1. La fonction malloc

La fonction malloc de la bibliothèque <stdlib> nous aide à localiser et à réserver de la mémoire libre au cours d'un programme. Elle nous donne accès au tas (heap); c.-à-d. à l'espace en mémoire laissé libre une fois mis en place le système d’exploitation, les gestionnaires, les programmes résidents, les autres applications actives et leurs données, le programme lui-même et la pile (stack).

malloc( <N> )

fournit l'adresse d'un bloc en mémoire de <N> octets libres ou la valeur zéro s'il n'y a pas assez de mémoire.

**Exemple**

Supposons que nous ayons besoin d'un bloc en mémoire pour un texte de 4000 caractères. Nous disposons d'un pointeur T sur char (char \*T). Alors l'instruction…

T = malloc(4000);

…fournit l'adresse d'un bloc de 4000 octets libres et l'affecte à T. S'il n'y a plus assez de mémoire, T obtient la valeur NULL.

* + 1. Echec de l’allocation

La valeur NULL retournée parfois par la fonction malloc traduit un **échec** lors de la tentative d’allocation de l’espace mémoire requis. Il est donc **indispensable de tester** la valeur du pointeur de récupération de l’adresse retournée par malloc pour éviter de travailler dans un pointeur NULL, ce qui engendrerait inévitablement une erreur d’exécution. La suite de l’exécution du programme ayant besoin de l’espace mémoire alloué doit donc être protégée par un if. En cas d’échec, afficher un message d’erreur et arrêter le programme.

* + 1. La fonction sizeof

L’appel de la fonction SIZEOF permet l’écriture d’un code indépendant du compilateur. Que votre système réserve 2 ou 4 octets pour le codage d’un entier est alors transparents pour vous puisque vous demandez explicitement la place pour [25 \* (la taille d’un entier)].

Si nous voulons réserver de la mémoire pour des données d'un type dont la grandeur varie d'une machine à l'autre, nous avons besoin de la grandeur effective d'une donnée de ce type. L'opérateur sizeof nous aide alors à préserver la portabilité du programme.

sizeof <var>

fournit la grandeur de la variable <var>

sizeof <const>

fournit la grandeur de la constante <const>

sizeof (<type>)

fournit la grandeur pour un objet du type <type>

***Exemple*** 1

Après la déclaration,

int tab1[10];

char tab2[5][10];

nous obtenons les résultats suivants sur un IBM-PC (ou compatible) :

|  |  |  |
| --- | --- | --- |
|  | sizeof tab1 | s'évalue à 20 (pour un int sur 2 octets, 40 sinon) |
|  | sizeof tab2 | s'évalue à 50 (ou 200) |
|  | sizeof 4.25 | s'évalue à 8 |
|  | sizeof "Bonjour !" | s'évalue à 10 |
|  | sizeof(float) | s'évalue à 4 |
|  | sizeof(double) | s'évalue à 8 |

***Exemple*** 2

Nous voulons réserver de la mémoire pour X valeurs du type **int**; la valeur de X étant lue au clavier :

int x;

int \*pnum;

printf("Introduire le nombre de valeurs :");

scanf("%d", &x);

pnum = (int\*)malloc(x\*sizeof(int)); //pnum devient le nom du tableau de x entiers

Entrons dans le détail de cette nouvelle façon « dynamique » de réserver un tableau d’entiers.

* + 1. Cas du tableau monodimensionnel

**Exemple** : int\* tab ;

tab = (int\*) malloc (25 \* sizeof(int)) ;

Pour se réserver une zone de 25 entiers il est nécessaire de déclarer un pointeur sur entier (tab). Quand l’utilisateur aura fourni le nombre d’éléments effectifs du tableau, l’appel de la fonction MALLOC (contenue dans la bibliothèque ALLOC.H) selon la syntaxe ci-dessus permettra la réservation de la zone mémoire appropriée. La fonction Malloc retourne l’adresse de la zone allouée. Cette adresse est castée en int\* pour correspondre à la variable de récupération tab.

* + 1. Allocation dynamique d’un tableau multidimensionnel

Prenons l’exemple d’un tableau que l’utilisateur souhaite de 4 lignes et 3 colonnes.

L’allocation dynamique d’un tel tableau se fera en 3 temps :

* Tout d’abord la déclaration d’un pointeur sur pointeur d’entiers,
* puis l’allocation d’un tableau de 4 pointeurs sur entiers,
* enfin, l’allocation des 4 tableaux de 3 entiers.

**Exemple** : int\*\* tab ;

Tab = (int\*\*) malloc (4\* sizeof(int\*)) ;

For (i=0 ; i<3 ; i++) tab[i] = (int\*) malloc (3\*sizeof(int)) ;

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Tab  (int\*\*) | → | tab[0]  (int\*) | → | tab[0][0]  (int) | tab[0][1] | tab[0][2] |
|  |  | tab[1] | → | tab[1][0] | tab[1][1] | tab[1][2] |
|  |  | tab[2] | → | tab[2][0] | tab[2][1] | tab[2][2] |
|  |  | Tab[3] | → | tab[3][0] | tab[3][1] | tab[3][2] |

* + 1. Cas des chaînes de caractères

***Exemple***

Nous voulons lire 10 phrases au clavier et mémoriser les phrases en utilisant un tableau de pointeurs sur **char**. Nous déclarons ce tableau de pointeurs par:

char \*texte[10];

Pour les 10 pointeurs, nous avons besoin de 10\*p octets. Ce nombre est connu dès le départ et les octets sont réservés automatiquement. Il nous est cependant impossible de prévoir à l'avance le nombre d'octets à réserver pour les phrases elles-mêmes qui seront introduites lors de l'exécution du programme ...

Le programme à la page suivante lit 10 phrases au clavier, recherche des blocs de mémoire libres assez grands pour la mémorisation et passe (ou stocke) les adresses aux composantes (ou cases) du tableau TEXTE[]. S'il n'y a pas assez de mémoire pour une chaîne, le programme affiche un message d'erreur et interrompt le programme avec le code d'erreur -1.

Nous devons utiliser une variable d'aide INTRO comme zone intermédiaire (non dynamique, donc automatique et pré dimensionnée, voire surdimensionnée). Pour cette raison, la longueur maximale d'une phrase est fixée à 500 caractères.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

main()

{

/\* Déclarations \*/

char INTRO[500]; /\* une chaîne de 499 car. utiles \*/

char \*TEXTE[10]; /\* un tableau de 10 pointeurs sur chaînes \*/

int I;

/\* Traitement \*/

for (I=0; I<10; I++)

{

gets(INTRO);

/\* Réservation de la mémoire pour chaque chaîne TEXTE[I]\*/

TEXTE[I] = (char\*)malloc(strlen(INTRO)+1);

if (TEXTE[I]) /\* Si la mémoire est correctement allouée \*/

/\* copier la phrase à l'adresse \*/

/\* fournie par malloc, ... \*/

strcpy(TEXTE[I], INTRO);

else

{

/\* sinon quitter le programme \*/

/\* après un message d'erreur. \*/

printf("ERREUR: Pas assez de mémoire \n");

}

}

return 0;

}

La fonction free

La particularité de la mémoire réservée dynamiquement est d’être prise dans une zone mémoire indépendante des réservations automatiques. Contrairement aux zones de mémoire automatique qui sont libérées dès qu’un sous-programme n’en a plus besoin, la mémoire dynamique n’est jamais automatiquement libérée.

Si nous n'avons plus besoin d'un bloc de mémoire que nous avons réservé à l'aide de **malloc**, il est nécessaire de le libérer à l'aide de la fonction **free** de la bibliothèque *<stdlib>*.

free( <Pointeur> )

libère le bloc de mémoire désigné par le <Pointeur>; n'a pas d'effet si le pointeur a la valeur zéro.

***Attention !***

* La fonction **free** peut aboutir à un désastre si on essaie de libérer de la mémoire qui n'a pas été correctement allouée par **malloc**.
* La fonction **free** ne change pas le contenu du pointeur; il est conseillé d'affecter la valeur NULL au pointeur immédiatement après avoir libéré le bloc de mémoire qui y était attaché.
* Si nous ne libérons pas explicitement la mémoire à l'aide **free**, alors nous risquons un débordement de la capacité maximale de mémoire de la machine (et donc un plantage grave). Sachez tout de même que la mémoire dynamique est libérée automatiquement à la fin du programme.

## Exercices d’application

POUR VOUS ENTRAINER

* + 1. Exercice 2001

Ecrire un programme qui lit 10 mots au clavier (longueur maximale: 50 caractères) et attribue leurs adresses à un tableau de pointeurs MOT. Effacer les 10 mots un à un, en suivant l'ordre lexicographique et en libérant leur espace en mémoire. Afficher à chaque fois les mots restants en attendant la confirmation de l'utilisateur (par 'Enter').

* + 1. Exercice 2002

Ecrire un programme qui lit 10 mots au clavier (longueur maximale: 50 caractères) et attribue leurs adresses à un tableau de pointeurs MOT. Copier les mots selon l'ordre lexicographique en une seule 'phrase' dont l'adresse est affectée à un pointeur PHRASE. Réserver l'espace nécessaire à la PHRASE avant de copier les mots. Libérer la mémoire occupée par chaque mot après l'avoir copié. Utiliser les fonctions de *<string>*.

* + 1. Exercice 2003

Ecrire un programme qui lit 10 phrases au clavier (longueur maximale: 50 caractères) et attribue leurs adresses à un tableau de pointeurs PHRASE. Réserver dynamiquement l'emplacement en mémoire pour les phrases. Trier les phrases lexicographiquement en n'échangeant que les pointeurs.

* + 1. Exercice 2004

Ecrire un programme qui lit 10 phrases d'une longueur maximale de 200 caractères au clavier et qui les mémorise dans un tableau de pointeurs sur **char** en réservant dynamiquement l'emplacement en mémoire ajusté pour chaque chaîne. Ensuite, l'ordre des phrases est inversé en modifiant les pointeurs et le tableau résultant est affiché.

**Dans cette seconde partie des exercices, vous réaliserez un menu de manipulation des chaînes de caractères.**

**Petite nouveauté, vous gérerez la mémoire en dynamique afin d’optimiser la place requise.**

Écrivez un programme dont le main affiche un menu permettant les actions décrites dans chacun des exercices suivants. Testez chacune des étapes avant de passer à la suivante.

*(Attention, vous blinderez les choix des utilisateurs dont l'ordre serait incohérent, ou encore vous proposerez d’autres solutions pour maintenir la cohérence des choix)*

* + 1. Exercice 2005

L'utilisateur peut saisir une phrase contenant espaces et ponctuations. La phrase sera affichée pour valider la saisie. La mémoire ajustée au contenu.

* + 1. Exercice 2006

Vous afficherez « palindrome : vrai » si la phrase saisie précédemment, puis transférée dans une chaîne ajustée est un palindrome, « palindrome : faux » sinon.

(rappel : « anna », « ivi », « abc cba »… sont des palindromes, c'est-à-dire chaînes symétriques par rapport au milieu)

* + 1. Exercice 2007

Vous appliquerez un décalage **circulaire** de cryptage sur cette phrase ajustée (rappel : circulaire veut dire ‘z’+1 donne ‘a’ … et ‘a’-1 donne ‘z’).

Le pas de cryptage sera demandé à l’utilisateur.

Le décryptage se fera par "décalage arrière'' (soustraction) de l'entier reçu.

* + 1. Exercice 2008

Vous permettrez la saisie d’une seconde phrase et concaténerez les 2 dans une troisième qui sera affichée à l’écran.

Attention, dans ce TD-TP toutes les chaînes doivent être dynamiques et ajustées au contenu.

* + 1. Exercice 2009

**Histogramme** : Vous compterez, dans la phrase concaténée, le nombre d’occurrences de chacune des lettres (le nombre de ‘a’, de ‘b’,… de ‘z’) (majuscule et minuscule comptées dans la même case). Le tableau de comptage sera affiché à l’écran.

* + 1. Exercice 2010

Vous permettrez la saisie d’une chaîne « numérique » (entière) que vous convertirez en nombre à l’aide de la fonction atoi().

Idem avec une chaîne numérique flottante converti avec ftoat().

Les nombres seront affichés à l’écran. Si la saisie n’est pas correcte, un message sera affiché.

Ex : 1265.566 (correct) 1a6dfgs24.525 (incorrect)

**Allocation dynamique en sous-programme**

**Faites ces exercices les uns après les autres dans un autre projet, le main étant de nouveau la dernière étape à écrire.**

**Choisissez à chaque fois le bon type de sous-programme.**

* + 1. Exercice 2011

1. Ecrire une fonction permettant l’allocation dynamique d’une matrice d’entiers après avoir demandé la valeur des 2 dimensions à l’utilisateur.
2. Ecrire une procédure permettant le remplissage **aléatoire** de la matrice précédente avec des valeurs entre 0 et 255 (bornes comprises).
3. Ecrire un sous-programme permettant le tri en ordre croissant des valeurs de la matrice d’entiers.

Exemple :

|  |  |  |
| --- | --- | --- |
| 8 | 10 | 12 |
| 15 | 23 | 45 |
| 51 | 52 | 56 |

1. Ecrire un sous-programme permettant l’affichage de la matrice d’entiers.
2. Ecrire le main tel qu’il joue son rôle de chef d’orchestre.  
   Vous dessinerez le graphe d’appel de votre programme en faisant apparaître le fonctionnement des tubes et des mécanismes de retour.
3. Ecrire et inclure à votre projet la bibliothèque personnelle au format .h

# Chapitre 21 - Passage des paramètres – Le tube

Rentrons maintenant dans le détail des appels de sous-programmes.

Pour fonctionner correctement, les sous-programmes ont parfois besoin d’informations provenant des autres sous-programmes ou du programme principal.

Le sous-programme appelant peut ainsi déposer des informations à l’usage de l’appelé dans le mécanisme de communication appelé « tube » ou « zone des paramètres », symbolisé par des parenthèses.

Il existe deux manières de passer des paramètres à un sous-programme par le tube (schémas).

Le passage par valeur

Le sous-programme **appelé** reçoit une copie des paramètres passés par la fonction **appelante** et manipule uniquement cette copie considérée alors comme une variable locale. La valeur de la variable transmise par l’appelant ne peut pas être modifiée par le sous-programme appelé.

Exemple :

int addition (int a, int b)

{

a = a+b ;

return(a) ;

}

Le passage par adresse

Les pointeurs, vu dans un chapitre précédent, sont également utilisés lors du passage de paramètres à une fonction. Ils permettent le mécanisme dit de « **données modifiées** », c'est-à-dire la modification d’une variable appartenant au sous-programme appelant.

**! A retenir : Le tube est un duplicateur.**

Cela signifie que tous les paramètres envoyés à une fonction sont dupliqués dans une zone mémoire nouvelle et porteront le nom associé aux paramètres en réception (schéma). La fonction travaille donc sur une **copie** de l’original et toute modification de la copie n’est pas répercutée à l’appelant.

Cependant, si vous envoyez en paramètre à l’appel une variable de type pointeur contenant une adresse (par exemple X de type int\*, contenant 0x0033), celle-ci sera reçue dans une copie de type pointeur, par exemple Y de type int\* qui contiendra également 0x0033.

Le paramètre est bien dupliqué et son contenu recopié. Par contre, ce contenu étant une adresse de zone mémoire, on dira que X et Y **pointent** désormais tous les deux sur la même zone. Toute modification de la zone pointée par Y modifie également la zone pointée par X (c’est la même).

Nous avons donc réussi à faire un passage dit ***par adresse*** (ou par référence) correspondant aux données modifiées.

|  |  |  |  |
| --- | --- | --- | --- |
| 0x033 | X |  |  |
|  |  |  |  |
| 0x033 | Y |  |  |
|  |  |  |  |
| 12 | 22 | 47 | 325 |

Le sous-programme, appelé reçoit les adresses des paramètres et travaille par **pointeur** directement sur la variable transmise en paramètre. Cette variable est alors **partagée** par l’appelant et l’appelé, toute modification étant donc répercutée.

Exemple :

int addition (int \*a, int \*b)

{

// 0. Declaration des variables

int c ;

// 1. traitements

\*a = \*a+10 ;

c = \*a+\*b ;

return(c) ;

}

Cas particulier des tableaux

En langage C, les tableaux sont automatiquement passés par adresse afin de ne pas faire de copie mémoire de structure de données trop lourde. Ce passage se fait sans l’utilisation de l’\*.

Toute modification d’une des cases du tableau par le sous-programme appelé est donc automatiquement répercutée.

Exemple :

void saisie (int tab[10])

{

int i ;

for (i=0 ; i<10 ; i++) scanf(« %d »,&(tab[i])) ;

}

Passage d’arguments à la fonction main : la ligne de commande

Le programme principal, qui correspond à la fonction MAIN en C, peut recevoir des paramètres venant de l’extérieur au moment du lancement du programme. Ces paramètres sont transmis au MAIN par l’intermédiaire de la ligne de commande.

Par exemple, si votre exécutable s’appelle prog\_test, la ligne de commande …

prog\_test 1 25 durant

… envoie les chaînes de caractères « 1 », « 25 », et « durant » au programme principal.

Pour ce mécanisme, le MAIN doit être écrit de la manière suivante :

void main (int argc, char\* argv[])

**argc** : nombre d’arguments passés au main, nom du programme compris

**argv** : tableau de pointeurs sur chaîne de caractères contenant chacun des arguments.

Le premier élément du tableau ARGV (c’est à dire argv[0]) contient le nom du programme exécutable, ce qui correspond dans notre exemple à **prog\_test**.

## Exercices d’application

Rappels de cours

Lorsqu’un sous-programme (ou le main) appelle un autre sous-programme en lui transmettant des informations (des variables), le tube joue son rôle de duplicateur et crée une copie de chacun des paramètres pour le sous-programme appelé. Ce mécanisme de création de copies a pour but de protéger le contenu des originaux. Donc si vous transmettez le contenu d’une variable par le tube, ce contenu est dupliqué dans une autre zone mémoire appartenant au sous-programme appelé. La modification de cette copie n’aura pas de répercussion sur l’original appartenant à l’appelant.

On appelle ce mode de passage de paramètre, un **passage par valeur**.

Cependant, lorsque l’on veut que la modification du contenu de la copie par le sous-programme appelé se répercute sur l’original détenu par l’appelant, il est possible de faire un autre mode de passage de paramètre : un **passage de paramètre par adresse**. Il consiste cette fois, non pas à passer le contenu mais l’adresse de la variable par le tube. Celui-ci joue alors son rôle de duplicateur et crée une copie de ce paramètre pour l’appelé qui dispose alors d’une clef d’accès (un pointeur) à la zone mémoire de l’appelant. L’appelé dispose d’une indirection vers la variable de l’appelant. Cette indirection lui permet de modifier le contenu de la zone m »moire de l’appelant (en utilisant l’opérateur « \* » - contenu pointé par - devant le pointeur).

**Question de cours**

* + 1. Exercice 2101
* Quels sont les différents modes de passage de paramètres ?
* Qu’est-ce qui les différencie ?

**Analyse et programmation**

* + 1. Exercice 2102

Ecrire un programme composé d’un main et d’une procédure unique qui reçoit en paramètre tous les types de passages de paramètres par adresse et par valeur.

Dans la procédure, vous afficherez chacun des paramètres reçus **avant et après** les avoir modifiés.

Dans le main, vous afficherez également l’ensemble des variables à transmettre par le tube, **avant et après** l’appel de la procédure pour constater les modifications de paramètre répercutées sur les originaux et celles qui ne le sont pas.

Vous afficherez les valeurs de la (ou des) variable(s) utilisée(s) dans le main et dans la procédure pour observer le comportement du programme et comprendre la notion de « modifiable » (passage par adresse) ou « non-modifiable » (passage par valeur).

Vous afficherez les informations suivantes :

* Valeurs des paramètres dans le main avant l’appel de procédure
* Valeurs des paramètres dans la procédure en réception avant modification
* Type de modification effectuée
* Valeurs des paramètres dans la procédure après modification
* Valeurs des paramètres dans le main après terminaison de la procédure
  + 1. Exercice 2103

Ecrire un programme composé d’un main et de deux procédures. La première reçoit en paramètre un tableau de 10 entiers « vide » (déclaré au niveau du Main) et demande à l’utilisateur de le remplir. La deuxième reçoit en paramètre un tableau de 10 entiers (déjà rempli) et transmet au Main le minimum et le maximum de ce tableau.

* + 1. Exercice 2104

Compléter l’exercice 1503 en ajoutant une fonction dont le rôle est d’allouer dynamiquement le tableau avec un nombre de cases obtenu au niveau du Main et transmis par le tube.

* + 1. Exercice 2105

Même exercice que le 1504, mais en faisant l’allocation dynamique du tableau dans une **procédure** cette fois.

**Ligne de commandes**

* + 1. Exercice 2106 : La ligne de commande

Créez un nouveau projet et recopiez ce Main afin de tester le comportement de la ligne de commandes et du programme suivant :

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char \*argv[])

{

printf("\n argv de 0 : %s\n",argv[0]);

printf("\n argv de 1 : %s\n",argv[1]);

printf("\n argv de 2 : %s\n",argv[2]);

printf("\n argv de 3 : %s\n",argv[3]);

system("PAUSE");

return 0;

}

1. Avant de lancer l’exécution, vous écrirez sur la ligne de commande (menu démarrer/exécuter de windows, ou dans les options « paramètres du projet » de Codeblocks) les valeurs suivantes : **<chemin>\monprog.exe bonjour 127**

(Où <chemin> est le chemin d’accès absolu à votre exécutable supposé nommé ici monprog.exe)

1. Modifiez ce code afin de récupérer la partie numérique de la ligne de commande et de la transformer en nombre à l’aide de la fonction *atoi*. Affichez cette valeur numérique avec le format %d de la fonction printf.
   * 1. Exercice 2107

Ecrire un programme qui récupère une chaine de caractères tapée par l’utilisateur dans la ligne de commande et qui stocke cette chaîne en ajustant dynamiquement au contenu.

# Chapitre 22 La récursivité

Définition

De même que chaque sous-programme peut en appeler un autre, chaque sous-programme peut s’appeler lui-même.

En langage C, toute fonction a la possibilité de s’appeler elle-même, ce qui en fait une fonction **récursive**. Il est donc possible d’écrire des fonctions récursives comme par exemple la fonction factorielle ci-dessous.

Exemple :

long int fact(int n)

{

if (n= =0) return 1 ;

else return (n\*fact(n-1)) ;

}

Cette fonction peut également être écrite de manière plus concise :

long int fact(int n) { return( n ? n\*fact(n-1) : 1) ;}

Utilité

Le récursif est un mode de pensée. Tout programme récursif aura un algorithme équivalent en mode itératif (classique). L’inverse n’est pas vrai.

Un algorithme récursif est souvent très concis en termes de lignes de code. Il est par contre plus gourmand en place mémoire.

La récursivité s’applique pour un traitement sur un ensemble d’objets ou de données. On ne va coder que le traitement d’un des objets (le premier en l’occurrence), puis on va rappeler le traitement récursivement sur l’ensemble privé du premier objet déjà traité. Et ainsi de suite jusqu’au traitement du dernier objet de l’ensemble ou jusqu’à l’ensemble vide.

La définition de la récursivité sera donc : **Méthode de programmation spécifique au traitement d’un ensemble de données de cardinalité décroissante.**

(Relisez bien le paragraphe précédent pour en digérer le contenu ;-)

Un exemple pour vous aider :

Trier en récursif un ensemble d’entiers en ordre croissant consiste à rechercher le plus petit entier de l’ensemble, puis à trier l’ensemble des entiers privé de cet élément. Et ce jusqu’à ce qu’il n’y ait plus d’entier à trier.

Notion de clone

De même qu’appeler un sous-programme revient à se mettre en attente et à lui confier le processeur, lorsqu’un sous-programme s’appelle lui-même, il se met en attente (pause) et génère un « clone » qui prend la main du processeur. A sa terminaison, le clone rendra la main à son parent afin que celui-ci poursuive son exécution à partir de son point de mise en attente.

Chaque clone étant identique à son parent, il va lui-même générer un autre clone après s’être mis en pause. La chaîne de clone va ainsi s’agrandir indéfiniment si elle n’est pas contrôlée par une condition d’arrêt.

Chaque point de mise en attente de chacun des clones est stockée dans une « pile d’adresse » afin de permettre à chacun de reprendre correctement son exécution lorsqu’il récupèrera la main sur le processeur.

Condition d’arrêt

Cette condition est une instruction (souvent un test) permettant à un clone de ne pas généré de successeur. Il va alors se terminer normalement sans se mettre en attente. Il va donc rendre la main à son parent qui pourra à son tour poursuivre son exécution et se terminer… rendant ainsi la main à son parent…etc.

Le dépilement des clones peut alors commencer et se dérouler correctement jusqu’à l’initiateur de la chaîne de récursion.

Procédure récursive

**Exemple** : 1. Binarisation récursive à 4 voisins. 2. Tri de tableau d’entiers.

## Fonction récursive

**Exercice** : (à corriger en TD)

Ecrivez un programme permettant de calculer l’élévation d’un nombre « a » à une puissance « b » : (appel : *puissance (a,b) ;*) **. Vous le décomposerez en sous-programmes.**

## Exercices d’application

* + 1. Exercice 2201

Ecrire un programme contenant **une procédure récursive** chargée d’afficher le contenu d’un tableau dynamique d’entiers dont le pointeur est reçu en paramètre.

Le tableau aura été alloué et rempli dans le main ou dans d’autres sous-programmes (voir séance de TD-TP précédente).

* + 1. Exercice 2202

Ecrire un programme contenant **une fonction récursive** permettant de calculer la longueur d’une chaîne de caractères (comme le ferait la fonction STRLEN).

**La chaîne de caractères à « mesurer » sera fournie obligatoirement par la ligne de commande.**

* + 1. Exercice 2203

Ecrivez un programme permettant de trier un tableau de flottants.

* La taille du tableau ainsi que les nombres à trier pourront être donnés par la ligne de commande ou demandés à l’utilisateur
* Le main affichera et gérera un menu proposant, entre autres, l’appel du sous-programme de saisie **si les nombres à trier ne sont pas fournis par la ligne de commande** ; il appellera également l’affichage du tableau non-trié puis trié.
* Le sous-programme « saisie » sera **récursif** et permettra de saisir n flottants s’ils ne sont pas donnés par la ligne de commande
* Le sous-programme « Affiche », **récursif** **également**, permettra de les afficher
* Le sous-programme « Tri », qui devra lui encore être **récursif**, permettra de trier le tableau de flottants dans un ordre croissant

# Chapitre 23 - Les fichiers : ascii et binaires

Les données stockées en mémoire sont perdues dès la sortie du programme. Les fichiers sur support magnétique (bande, disquette, disque) sont par contre permanents, mais au prix d'un temps d'accès aux données très supérieur. On peut distinguer les fichiers séquentiels (on accède au contenu dans l'ordre du stockage) ou à accès direct (on peut directement accéder à n'importe quel endroit du fichier).

Les fichiers que nous allons considérer ici sont des fichiers séquentiels. Les fichiers sont soit binaires (un float sera stocké comme il est codé en mémoire, d'où gain de place mais incompatibilité entre logiciels), soit formaté ASCII (un float binaire sera transformé en décimal puis on écrira le caractère correspondant à chaque chiffre).

Les fichiers étant dépendants du matériel, ils ne sont pas prévus dans la syntaxe du C mais par l'intermédiaire de fonctions spécifiques.

Rappels sur les fichiers ASCII

Ils ont déjà été présentés au chapitre 7. Quelques rappels…

* + 1. Création et Ouverture d'un fichier

Un fichier peut être représenté de 2 manières:

Son nom externe

C'est le nom que lui a donné son propriétaire au moment de sa création sous la forme d'une chaîne de caractères.

Son nom interne

C'est le nom que lui a donné le système au moment de l'ouverture de ce fichier. Ce nom interne est une structure de fichier de type **FILE** (obligatoirement en majuscules) et qui contient les informations suivantes: *mode d'ouverture* du fichier, numéro du descripteur de fichier, informations concernant le tampon associé au fichier pour les opérations de lecture/écriture, pointeur courant dans le fichier.

A chaque fois que l'on prévoit d'appeler des fonctions d'E/S de la bibliothèque standard, il convient d'inclure le fichier **stdio.h** qui contient les définitions et les prototypes des fonctions d'E/S.

Il existe 3 fichiers *standards* associés à l'écran du terminal :

**stdin**: fichier d'entrée (nom interne = 0)

**stdout**: fichier de sortie (nom interne = 1)

**stderr**: fichier de sortie erreur (nom interne = 2)

Pour ouvrir un fichier il faut déclarer un pointeur de type **FILE \*** et faire appel à la fonction fopen dont le prototype est le suivant :

FILE \* fopen (const char \*path, const char \*mode);

La fonction fopen( ) retourne un pointeur sur une structure de type FILE ou bien un pointeur NULL si l'opération a échoué.

**Exemple :**

FILE \* fp;

fp = fopen ("chemin + nom-de-fichier", "w");

Il existe plusieurs modes d'ouverture d'un fichier :

|  |  |
| --- | --- |
| mode | Fonction |
| "r" | Ouvre un fichier existant pour la lecture |
| "w" | Crée et ouvre un fichier pour l'écriture. Tout fichier existant est remplacé. Un nouveau fichier est créé, s'il n'en existe pas. |
| "a" | Ouvre un fichier en ajout. L'ajout se fait à la fin d'un fichier existant. Un nouveau fichier est créé, s'il n'en existe pas. |
| "r+" | Ouvre un fichier existant pour la lecture et l'écriture |
| "w+" | Crée et ouvre un fichier existant pour la lecture et l'écriture. Tout fichier existant est remplacé. |
| "a+" | Ouvre un fichier pour la lecture et l'ajout de données à la fin d'un fichier existant. Un fichier est créé s'il n'en existe pas |

Chacun de ces droits peut être suivi d’un ‘b’ pour préciser le mode binaire (ex : « wb »).

* + 1. Lecture/Ecriture dans un fichier texte

De nombreuses fonctions permettent l'écriture ou la lecture d'un fichier texte. Voici leur prototype.

**Lecture:**

int  **fscanf** (FILE \*stream, const char \*format[,argument,...];

int **fgets** ( char \*string, int n, FILE \*stream);

int **fgetc** (FILE \*stream);

**Ecriture:**

int **fprintf** (FILE \*stream, const char \*format[,argument,...];

int  **fputs** (const char \*string, FILE \*stream);

int **fputc** (int c, FILE \*stream);

* + 1. Fermeture d'un fichier

Tout fichier ouvert doit être fermé. La fonction fclose( ) permet de fermer le fichier repéré par son pointeur.

int fclose (FILE \*stream);

* + 1. Principe de manipulation des fichiers

Pour manipuler un fichier, il faut toujours avoir ouvert ce fichier au préalable.

Il vaut mieux refermer un fichier inutilisé. L'instruction exit( ) referme également tous les fichiers ouverts et vide les mémoires tampons.

**Exemple**:

/\* creer le fichier "toto.txt" puis écrire dans le fichier (sous Windows) \*/

#include <stdio.h>

main( )

{

FILE \*fp;

static char t[ ]="j'ecris ce que je veux dans le fichier nomme fichier";

**/\* attention : ci-dessous le \ est doublé \\ car sinon '\f' sera considéré comme un caractère d'échappement genre \n, \t, \b….\*/**

fp = fopen (".\\toto.txt","w");

fprintf(fp,"%s",t); // ouverture

if (fp != NULL)

{ // 3 façons d’écrire dans le fichier

fputs(" et ceci de plusieurs facons", fp);

fputc('!',fp);

fprintf(fp,"%s","au revoir...");

fclose(fp);

}

}

**Exemple**:

/\* lecture du fichier "fichier" (sous Linux) \*/

#include <stdio.h>

main( )

{ // declaration des ressources

FILE \*fp;

char c;

char t[100]; char s[100];

fp = fopen ("./fichier","r");

if (fp != NULL)

{ // 3 façons de lire dans le fichier

fgets(s,52,fp);

c = fgetc(fp);

fscanf(fp,"%s",t);

fclose(fp);

printf("s=%s c=%c t=%s",s,c,t);

}

}

Pour être parfaitement rigoureux, il faut toujours tester le retour de la fonction qui ouvre (ou crée) le fichier. La fonction fopen( ) rend un pointeur nul lorsque l'opération de création ou d'ouverture de fichier a échoué.

En C condensé, dans le dernier exemple on peut écrire :

if ((fp = fopen ("fichier","r")) != NULL) {...}

else

{

printf("l'opération d'ouverture du fichier a échoué\n");

}

L'écriture ou la lecture dans un fichier se fait suivant le **pointeur courant** positionné à l'adresse où la dernière opération de lecture ou d'écriture a eu lieu. Pour se repositionner en début de fichier, on utilise la fonction rewind( ) déclarée par:

int rewind(FILE \* stream);

Pour se repositionner n'importe où dans le fichier, on utilise la fonction fseek( ) déclarée par:

int fseek(FILE \*stream, long offset, int origin);

Exemple:

#include <stdio.h>

main( )

{

// declarationd es ressources

FILE \*fp;

char c;

char t[100]; char s[100]; char tab1[100]; char tab2[100];

/\* ouvrir le fichier en lecture \*/

fp = fopen ("temp\\fichier.txt","r");

if (fp) // equivalent à if (fp != NULL)

{

/\* lire les 51 premiers caractères dans le tableau s \*/

fgets(s,52,fp);

/\* lire le caractère suivant dans c \*/

c = fgetc(fp);

/\* lire le mot suivant dans le tableau t \*/

fscanf(fp,"%s",t);

/\* afficher le contenu des variables s, c, t à l'écran \*/

printf("s=%s c=%c t=%s",s,c,t);

/\* se positionner en début de fichier \*/

**rewind(fp);**

/\* lire dans tab1 les 99 premiers caractères \*/

fgets (tab1, 100, fp);

/\* afficher à l'écran le contenu de tab1 \*/

printf("\ntab1=%s",tab1);

/\* se positionner à un déplacement de -40 octets par rapport au pointeur courant \*/

**fseek (fp, -40L, SEEK\_CUR);** /\* (SEEK\_CUR, SEEK\_SET, SEEK\_END)\*/

/\* lire les 99 caractères à partir du pointeur courant dans tab2 \*/

fgets (tab2, 100, fp);

/\* afficher le contenu de tab2 à l'écran \*/

printf("\ntab2=%s",tab2);

/\* fermer le fichier \*/

fclose(fp);

}

}

* + 1. Le caractère de fin de fichier (EOF)

Tous les fichiers texte ou binaire possède un caractère qui marque la fin du fichier. Cette marque de fin de fichier appelée génériquement EOF est souvent différent selon le type de machine utilisée. La valeur EOF est définie dans le fichier d'inclusion *stdio.h*. Pour un fichier texte, cette valeur est le plus souvent égale à 0 ou à -1.

**Exemple :**

/\* lire le fichier c:\toto.txt caractère après caractère jusqu'à la marque de fin de fichier \*/

#include <stdio.h>

main()

{

// declaration des ressources

int i;

FILE \*fp;

// ouverture et lescture du fichier

if (fp = fopen("toto.txt","r")) // si le fichier est bien ouvert…

{

**while ((i = fgetc(fp)) != EOF)** printf("%c", i);

/\* afficher le contenu du fichier à l'écran \*/

}

}

Fichiers binaires

Certaines applications nécessitent de mémoriser les données par blocs dans les fichiers. Par exemples si dans une application un ensemble de structures ayant la même composition est utilisé, il est intéressant de pouvoir écrire dans le fichier le contenu d'une structure entière (sans être obligé d'écrire le contenu de la structure champ par champ).

Les fonctions **fread** et **fwrite** sont destinées à ce type d'entrée/sorties et sont désignées sous le terme fonctions **d'entrées/sorties binaires**.

Ces fonctions requièrent quatre arguments :

* un pointeur sur le bloc de données (par exemple un pointeur de structure)
* la taille du bloc
* le nombre de blocs transférés
* le pointeur de fichier.

Les autres fonctions de manipulation de fichiers (fopen, fclose) ont la même utilisation avec les fichiers binaires. Sous certains systèmes d’exploitation, il est nécessaire d’ajouter un ‘b’ aux droits requis à l’ouverture (ex : « rb+ »).

**Exemple** :

#include <stdio.h>

*typedef struct*

*{*

*char nom[30];*

*char prenom[40];*

*int age;*

*char sexe;*

*} t\_personne;*

/////////////////////////////////////////////////////////////////////

void main( )

{

**// 0. Déclaration des variables locales**

**/\* déclaration de pointeur de fichier \*/**

FILE \* fp1;

FILE \* fp2;

**/\* déclaration et initialisation de maPomme \*/**

t\_personne maPomme = {"DUPONT","RENE",28,'M'};

t\_personne uneAutrePomme;

// 1. Ouverture et accès aux fichiers

if (fp1 = fopen("essai.dat","wb"))

{

**/\*ecriture de ma Pomme dans le fichier \*/**

**fwrite**(&maPomme, sizeof(t\_personne),1,fp1);

fclose(fp1);

}

if (fp2 = fopen("essai.dat","rb"))

{

**/\*lecture de ma Pomme dans le fichier \*/**

**fread**(&uneAutrePomme, sizeof(t\_personne),1,fp2);

fclose(fp2);

printf("\n\n NOM :%s PRENOM : %s",uneAutrePomme.nom, maPomme.prenom);

printf("\n AGE %d", uneAutrePomme.age);

printf("\n SEXE : %s\n",(uneAutrePomme.sexe=='M') ? "MASCULIN":"FEMININ");

}

}

## Exercices d’application

* + 1. Exercice 2301

Soit la structure **t\_film** contenant le titre, le réalisateur, l’année, le genre…

* Ecrivez le typedef correspondant à cette structure.
* Ecrivez la fonction de remplissage par l’utilisateur d’une instance dynamique de t\_film.
* Ecrivez un sous-programme chargé de sauvegarder dans un fichier texte (dont le nom sera reçu en paramètre) **une** instance déjà remplie de cette structure dont l’adresse est reçue en paramètre.
  + 1. Exercice 2302

Ecrivez un sous-programme recevant en paramètres le nom d’un fichier et dont le rôle est **de lire** la totalité du contenu du fichier pour construire le tableau des films archivés. Vous afficherez le contenu de ce tableau à l’écran.

* + 1. Exercice 2303

Ecrivez le Main jouant son rôle de chef d’orchestre et dont les fonctionnalités permettent également de demander à l’utilisateur combien de films il souhaite enregistrer cette fois pour les **ajouter** au fichier d’archive, puis alloue dynamiquement le tableau de stockage des films (tableau dynamique de pointeurs sur structures dynamiques).

Le Main appellera ensuite les sous-programmes ci-dessus pour réaliser/compléter votre archivage.

Même idée sur un autre thème : les rendez-vous

**A FAIRE EVENTUELLEMENT SOUS FORME DE MINI-PROJET**

* + 1. Exercice 2304

Créez le type correspondant à la structure d’un rendez-vous. Vous y placerez :

* un champ date au format jour/mois/année
* un champ heure début
* un champ heure fin
* un champ libellé
* un booléen rappel

Vous compléterez cette description par tous les champs que vous jugeriez nécessaire d’ajouter.

* + 1. Exercice 2305

Vous définirez la structure de données la plus adaptée à vos choix de stockage : tableau de rendez-vous ou tableau de pointeurs sur rendez-vous.

* + 1. Exercice 2306

Il s’agit maintenant de réaliser notre programme, avec un maximum de structuration en sous-programmes, en proposant à l’utilisateur les options suivantes :

* ajouter un RDV (attention à la place mémoire disponible)
* afficher tous les RDV
* rechercher un RDV correspondant à un critère donné
* afficher un RDV correspondant à un critère donné
* supprimer un rendez-vous correspondant à un critère donné
* supprimer tous les RDV
  + 1. Exercice 2307

Afin de ne pas être obligé de tout ressaisir à chaque fois et de ne pas perdre les rendez-vous créés, vous coderez maintenant la sauvegarde sur disque. Vous créerez les sous-programmes :

* sauvegarder les RDV dans un fichier (vous choisirez texte ou binaire)
* charger les RDV à partir d’un fichier de sauvegarde

Le sous-programme de chargement sera lancé systématiquement à l’ouverture de votre programme afin de monter en mémoire tous les RDV contenus dans le fichier. (Vous pourrez discuter avec votre chargé de TD-TP du bienfondé de cette méthode). Au passage, vous afficherez à l’écran tous les rendez-vous du jour. Si parmi ces rendez-vous certains ont le champ booléen rappel à vrai, vous afficherez une fenêtre pop-up de rappel affichant le libellé du RDV 15 minutes avant celui-ci.

* + 1. Exercice 2308

Pour optimiser la gestion de la mémoire, vous adapterez votre définition de type(s) afin de pouvoir stocker vos rendez-vous en liste chaînée.

* + 1. Exercice 2309

Codez de nouveau votre bibliothèque de sous-programmes de manipulation des rendez-vous afin de les adapter aux listes chaînées (ne supprimez pas les autres sous-programmes).

# Chapitre 24 - Structures de données complexes : listes, piles, files, listes chaînées, table de hachage, arbre, graphe…

Listes, Piles, Files

* + 1. Définition d’une liste

Une liste est une collection d’éléments dont la nature nous guidera dans le mode de stockage à choisir : monotype = tableaux prioritairement, polytype = structures en tableaux ou en liste chaînée.

Il y a plusieurs façons de manipuler cette collection. Deux d’entre elles sont remarquables par les nombreux problèmes qui suivent l’une ou l’autre : les Piles et le Files, LIFO et FIFO

* + 1. Usage en Pile

Une Pile, c’est comme une pile d’assiettes. Il est logique d’ajouter une nouvelle assiette en sommet de pile, il est également logique de retirer la première assiette en sommet de Pile. On dit alors que la dernière entrée est la première sortie de la Pile (Last In First Out ou LIFO en anglais).

* + 1. Usage en File

Une File, c’est comme une file d’attente à un guichet. Il est logique d’ajouter une nouvelle personne en queue de File, il est également logique de servir la personne en tête de File qui fait donc la queue depuis le plus longtemps. On dit alors que la première entrée est la première sortie de la File (Firt In First Out ou FIFO en anglais).

Cherchez quels problèmes de la vie courantes se gèrent en LIFO ou en FIFO ?

Les listes chaînées

* + 1. Définition d’une liste chaînée

Les listes chaînées sont des structures de même type liées entre elles par des pointeurs.

Elles permettent de lier des éléments de même nature (complexe) entre eux comme le ferait un tableau.

Elles offrent une souplesse supplémentaire par rapport au tableau en permettant d’ajouter ou supprimer les éléments un à un, sans aucune limite de taille (si la place mémoire est disponible évidemment).

* + 1. Construction

Définition des types en bibliothèque : les concepts

On définira tout d’abord dans une bibliothèque le concept de la ou des structure(s) contenant les informations dites « utiles » en respectant la logique des inclusions (logique du compilateur).

typedef struct naiss {

int jour;

int mois ;

int annee ;

} t\_naissance;

typedef struct data {

char nom[16];

char prenom[20];

t\_naissance date\_naissance ;

} t\_data;

Puis on définira la structure permettant le chaînage en contenant un champ de liaison sur une structure de même type.

typedef **struct cellule** {

t\_data data ;

**struct cellule \*suiv ;**

} t\_cell;

Ces concepts sont purement **déclaratifs** et ne prennent aucune place en mémoire. On ne peut donc pas y stocker de l’information. Pour stocker de l’information en mémoire, il faudra **instancier** ces concepts, par exemple en réalisant une allocation dynamique de ces structures.

Déclaration du pointeur initial

La liste chaînée sera rattachée à son propriétaire par un pointeur initial nommé par convention « ancre ». Cette ancre pointera sur la première cellule de la liste. Elle sera à NULL si la liste est vide.

t\_cell \*ancre ;

Chaque maillon de la liste sera ensuite alloué dynamiquement et rattaché un à un à son prédécesseur

t\_cell \* maillon ;

maillon = (t\_cell\*) malloc (1\*sizeof(t\_cell) ; // création d’une instance de structure

ancre = maillon ;

Marquage de fin

Afin de localiser la fin de la liste dans toutes les opérations de manipulation, le dernier maillon de la liste (la dernière instance) contiendra la valeur NULL dans son champ de chaînage.

maillon->suiv = NULL ;

* + 1. Parcours

Pour parcourir la liste, on définira un pointeur temporaire qui sera initialisé sur l’ancre et qui se déplacera de maillon en maillon tant qu’il en subsiste. Attention à ne pas dépasser la fin !

t\_cell \* pt ;

pt = ancre ;

…

while (pt != NULL) pt = pt-> suiv ; // permet d’avancer le pointeur d’un cran

* + 1. Ajout

Ajout en tête

Ajout en queue

Ajout à la bonne place

* + 1. Liste doublement chaînée
    2. Liste circulaire
    3. Listes de listes

Les tables de hachage

Les tables de hachage sont utilisées dans des opérations de classement, d’ordonnancement ou de classification

Les tables de hachages sont un mélange de tableau et de liste chaînée. Le tableau contient dans chacune de ses cases un pointeur pouvant servir d’ancre à un morceau de la liste chaînée globale, permettant ainsi un accès plus rapide à une partie de la liste. Chaque case peut par exemple représenter une catégorie du classement.

L’ajout dans la liste chaînée globale se fera de manière classique.

Le nombre de cases du tableau correspond au nombre de catégories du classement.

Ancre

Table\_hach

Les arbres

Les arbres sont des structures de données complexes qui utilisent les listes chaînées. Ils permettent une organisation des données adaptée à certains types de problèmes comme les analyses grammaticales ou lexicographiques, les arbres généalogiques...

Nous étudierons en cours le cas particulier des Arbres Binaires de Recherche pour en découvrir les modalités de construction et les avantages.

Les graphes

Les graphes sont des structures de données complexes qui utilisent les listes chaînées. Ils permettent une organisation des données adaptée à certains types de problèmes comme les recherches de plus court chemin dans un réseau.

Voilà, nous arrivons au terme de ce cours. Il a été riche en nouveautés… en difficultés aussi.

Rappelez-vous que notre objectif n’a jamais été de faire de vous des pros du C, même si certains se débrouillent pas mal ;-)

Le véritable challenge c’est de vous avoir amené à réfléchir avant de coder en utilisant une méthode, parmi tant d’autres que vous aurez l’occasion de découvrir dans vos stages et votre vie professionnelle. A l’image d’un architecte qui réalise les plans AVANT de commencer la construction d’un bâtiment.

J’espère que vous avez pu y prendre du plaisir parfois, même si je suis conscient que certains d’entre vous auront profité de ce cours pour décider que ce n’est pas « à coder » qu’ils destinent leur vie professionnelle. ;-)

Bon courage pour la suite.

FR

## Exercices d’application

Dans les chapitres précédents sur les structures, vous avez vu que celles-ci permettaient de regrouper des données de nature différentes dans un même « objet » (une instance de structure). Vous avez également vu qu’on pouvait mettre des structures dans des structures (structures imbriquées, par exemple un champ « date » dans une structure « rendez-vous »).

Les organisations de stockage complexes « Listes chaînées », « Arbres » et « Graphes » ne sont rien d’autres qu’un assemblage particulier de structures de données plus simple vues en cours.

Votre rôle est déjà de choisir cet assemblage : tableau ? Structures ?... Pour cela vous devez donc maitriser les avantages et inconvénients de toutes les organisations de données à votre disposition.

Dans ce TD-TP, le but est de manipuler les listes chaînées. Vous êtes donc contraints dans ce choix.

* Une liste chainée n’est rien d’autre qu’un assemblage de structures contenant chacune **un** **pointeur** qui pointe vers une autre structure de même type.
* Un arbre binaire n’est rien d’autre qu’un assemblage de structures contenant chacune **deux** **pointeurs** vers d’autres structures.
* Un graphe n’est rien d’autre qu’un assemblage de structures contenant chacune **N pointeurs** vers d’autres structures…

===================

* + 1. Exercice 2401

Ecrire la définition de type (typedef) pour une structure contenant les informations relatives à un **étudiants** ECE.

Ecrire la définition de type (typedef) pour une structure contenant les informations relatives à un **enseignant** ECE.

Y a-t-il des informations communes aux 2 types que vous venez de définir ?

Auriez-vous pu organiser autrement vos définitions de type pour éviter les redondances ?

* + 1. Exercice 2402

Ecrire la définition de type (typedef) pour une structure permettant le chaînage des étudiants ECE comme définis précédemment.

* + 1. Exercice 2403

Ecrire une fonction de « comptage » qui reçoit l’ancre d’une liste chaînée **quelconque** (attention à la liste vide) et qui retourne le nombre de cellules dans la liste.

* + 1. Exercice 2404

Ecrire toutes les fonctions « classiques » qui permettent les manipulations des listes chaînées :

* Ajout en tête
* Ajout en queue
* Suppression
* Recherche
* Modification
* Etc.

(vous intégrerez la fonction de comptage de l’exercice précédent à cette libraire de fonctions)

**VOS CHARGES DE TD-TP VOUS PROPOSERONT DES EXERCICES COMPLEMENTAIRES EN FONCTION DE VOTRE PROGRESSION.**

**MINI-PROJET ou TP FACULTATIF**

# Mini Projet Image : Visualisation, traitement et binarisation via histogramme

**Ce TD-TP de manipulation des images est à faire individuellement !**

L’objectif de ce TP est de manipuler des images en 256 niveaux de gris au **format RAW** (sans entête)contenant uniquement le tableau de valeurs correspondant à chaque point (appelé PIXEL) d’une image qui peuvent donc se stocker chacune dans un **unsigned char**.

Les formats d’image plus classique comme BMP, GIF ou JPG contiennent un entête qui précise les dimensions de l’image, sa nature (couleur ou niveaux de gris) ainsi que son éventuel mode de compression. Dans le cas des images RAW, ces informations sont à renseigner à la main à l’ouverture du fichier.

Vous visualiserez à l’écran vos images et le résultat des traitements que vous appliquerez à l’aide d’une application nommée **IRFANVIEW**.

Cette application vous est fournie sur campus sous forme de fichier compressé, qu’il vous suffit de décompresser, sans autre procédure d’installation.

Vous pouvez également utiliser des applications de visualisation d’images comme PAINT sous WINDOWS ou GIMP sous LINUX.

Vous manipulerez dans un premier temps des images créées par vous-même, avant de voir des images « réelles ».

Les traitements qui peuvent s’appliquer aux images dans un but « professionnels » dans le cadre de l’imagerie médicale ou biologique par exemple, ou dans un but amateur comme la suppression des yeux rouges, peuvent se répartir en deux catégories :

* Les traitements dits de **« bas niveau »** ont pour but d’améliorer visuellement une image pour le confort de l’œil humain. C’est le cas de l’amélioration de contraste, la réduction de flou, la modification de luminosité...
* Les traitements de **« haut niveau »** sont destinés à interpréter l’image pour en extraire de l’information. On retrouve dans cette catégorie les outils de segmentation, d’extraction de contours, de dénombrements... pour localiser un cancer dans une IRM par exemple.

Nous allons, dans ce TD-TP nous limiter à des manipulations de bas niveau.

**Exercice 1**

Ecrire un programme, structuré en sous-programmes, remplissant aléatoirement un tableau à deux dimensions (matrice) de 100 lignes et 200 colonnes avec des valeurs comprises entre 0 et 255 (bornes comprises).

Sauvegarder cette image dans un fichier en utilisant les fonctions *fprintf* ou *fputc*,mais en ouvrant le fichier en mode binaire (« rb », « wb »…). Vous nommerez ce fichier « image1\_100\_200.raw ».

Afficher cette matrice à l’écran en utilisant l’application IRFANVIEW. Que constatez-vous visuellement ?

**Pourquoi le mode binaire ?** Le mode binaire permet de gérer proprement l’absence de « retour à la ligne » dans une image. En effet, le logiciel de visualisation assure l’aspect « carré » de l’image grâce aux dimensions que vous renseignez à l’ouverture, mais en mémoire tous les pixels sont en continu.

**Pourquoi une écriture en mode texte ?** En fait, le mode texte (fprintf, fputc…) fonctionne aussi bien que le mode bloc (fread, fwrite…) pour sauvegarder les pixels. Il est même possible de sauvegarder tous les pixels d’un coup en un seul fwrite puisqu’ils sont continus en mémoire.

**Exercice 2**

Ouvrez maintenant avec Irfanview l’image « cellule » à disposition sur campus. Les dimensions de l’image sont dans le nom, elle est en niveaux de gris sur 8 bits, sans entête.

Contrairement à l’image aléatoire générée à l’exercice 1 qui montre un fond uniformément gris moucheté du fait de la répartition quasi équiprobable des 256 niveaux de gris sur l’ensemble de l’image, une image « réelle » comme celle des cellules montre des « objets » qui se détachent plus nettement sur le fond, parfois en clair sur fond foncé, parfois en foncé sur fond clair.

Ce « contraste » (écart des valeurs de niveau de gris) entre objets et fond permet à l’œil humain d’interpréter correctement l’image.

***Différence entre œil humain et machine :***

***La sensibilité de l’œil humain nécessite que le contraste entre les objets et le fond soit suffisamment important pour que les différences soient perçues.***

***L’ordinateur, quant à lui, se moque de la notion de contraste fort, et différencie aussi bien des éléments de l’image dont les niveaux de gris différent de 1 que s’ils différaient de 100.***

Chargez cette image de cellules dans votre programme (lecture du fichier en mode texte et récupération des pixels avec fgetc ou fscan %c sans tenir compte des passages à la ligne).

Dupliquez l’image de cellule dans une autre image en mémoire, en divisant par 10 la valeur de chaque pixel (division entière).

Sauvegardez cette nouvelle image dans le fichier « cellule2\_xxx\_xxx.raw » (où les xxx représentent les dimensions) et visualisez-là avec Irfanview. Que constatez-vous ?

Ouvrez « cellule2\_xxx\_xxx.raw », dans votre programme, dupliquez-là dans une troisième image en mémoire en multipliant chaque valeur de pixel par 10 et sauvegardez le résultat dans un fichier « cellule3\_xxx\_xxx.raw ».

Visualisez cette 3ème image avec Irfanview et comparez-là avec l’image de cellule d’origine récupérée sur campus. Sont-elles identiques ? Expliquez.

**Exercice 3**

L’histogramme d’une image étudie la répartition statistique de chaque valeur de niveau de gris dans une image.

Son principe consiste à compter le nombre de fois où chaque valeur de niveau de gris est représentée dans l’image.

Pour cela, dans un tableau nommé histogramme de 256 cases de type entier, vous compterez combien l’image contient de pixels de niveau de gris 0 et à stocker cette valeur dans la case 0 du tableau histogramme, puis combien l’image contient de pixels de valeur 1 à ranger dans la case 1… ainsi de suite jusqu’à compter le nombre de pixels à 255.

**Attention** **!!** Il y a une méthode de calcul rapide et une méthode très lente. Réfléchissez !

Compléter le programme précédent pour qu’il calcule l’histogramme de l’image1 (de l’exercice 1401) après son remplissage aléatoire, ainsi que les histogrammes des images cellule1, cellule2 et cellule3.

Affichez cet histogramme de manière lisible à l’écran.

Ex : Niveau de gris 0 : 15

Niveau de gris 1 : 27

Niveau de gris 2 : 48

…

Comparez les différents histogrammes et essayez de décrire en quelques mots les caractéristiques de chacun.

**Exercice 4**

La binarisation d’une image consiste, **dans une image secondaire afin de ne pas modifier l’image originale**, à mettre à 0 tous les pixels de l’image originale inférieurs à une valeur seuil entrée par l’utilisateur ; et à mettre à 255 tous les pixels de l’image originale supérieurs ou égaux à cette valeur seuil.

Compléter à nouveau le programme précédent pour y inclure la binarisation.

Utilisez les histogrammes pour identifier les valeurs seuil « idéales ». Cette valeur correspond au niveau de gris médian entre 2 « pics » de valeurs (1 pic pour le fond, 1 pic pour les formes).

**Exercice 5**

Utilisez vos outils de binarisation et de modification de contraste sur les images de plaque d’immatriculation et de visage de mon ami Olivier. Jouez aux experts comme cela vous est suggéré sur campus.

Amusez-vous bien !!

**ANNEXES**

# ANNEXE A : Règles d’écriture des programmes en C

**Règles de base à l’ECE**

Vous êtes ici pour apprendre à « bien » programmer. Or, chaque entreprise à ses propres règles de « bonne programmation ».

Vous allez donc commencer par apprendre à respecter les règles de ceux pour qui vous travaillez. Il sera toujours bien assez tôt, le jour où vous pourrez faire ce que bon vous semble.

A l’ECE, nous appliquons les règles suivantes, parfois différentes de l’entreprise.

* Les fonctions ne devraient pas excéder 25 lignes. Les décomposer sinon.
* Il faut éviter plus de 5 paramètres par fonction.
* Les noms de variables ou de fonctions doivent être explicites.
* Les fonctions et les variables sont écrites en minuscule ou commencent par une minuscule (voir notation hongroise).
* Les goto sont **interdits** (pas les gotoxy ou gotoligcol qui servent à positionner le curseur à l’écran)
* Les programmes doivent être commentés à chaque traitement ou fonction et correctement indentés.

**Ecriture Hongroise – Règles dans l’entreprise**

L’écriture hongroise est très utilisée aujourd’hui dans de nombreuses API (Interface de programmation) comme Windows, X11 et bien d’autres.

Son but est de permettre une normalisation et une meilleure lisibilité des programmes.

Les applications actuelles sont souvent le travail de plusieurs développeurs et cela nécessite que chacun comprenne ce que les autres ont fait.

**Règles :**

* Les noms de symboles (variables et fonctions) doivent être explicites (je vous l’avais dit ;-).
* Les variables commencent par une minuscule et peuvent contenir des majuscules.
* Les noms de fonctions commencent par une majuscule (celle là, on ne la respecte pas).
* Les fonctions sont précédées d’une zone de commentaire qui en précise l’utilité.
* On ne doit pas utiliser des caractères spéciaux ou des caractères accentués dans les identificateurs (noms) de variables ou de fonctions.
* Les noms sont constitués de plusieurs mots ou mots abrégés séparés par des « \_ » (AltGr + 8).
* Les programmes sont commentés.

Rien ne vaut quelques exemples :

**Ecriture classique : (mauvaise)**

#include <stdio.h>

int addk (int a, int b); /\* prototype de la fonction addk \*/

/\* la fonction addk retourne une valeur entière\*/

/\* et accepte 2 entiers en arguments\*/

main()

**{**

**int a = 5;**

**int b = 9;**

int résultat = addk (a , b); // beurk, un accent

printf("%d",résultat); /\* impression de la valeur 14 (0 + 5 + 9)\*/

résultat = addk (a , b);

printf("%d",résultat); /\* impression de la valeur 28 (14 + 5 + 9) \*/

**}**

int addk (int u, int v)

**{** **static** int k = 0; /\* la variable k est automatique \*/

k = k + u + v;

return(k); **}**

**Version écriture hongroise : La bonne**

#include <stdio.h>

int ***AjouteK*** (int valA, int valB); /\* prototype de la fonction ***AjouteK*** \*/

/\* la fonction ***AjouteK*** retourne une valeur entière\*/

/\* et accepte 2 entiers en arguments\*/

**int** ***AjouteK*** (**int** valU, **int** valW)

**{**

**static** **int** valK = 0; /\* la variable valK est automatique \*/

valK = valK + valU + valW;

**return**(valK);

**}**

**// Et le main, à la fin ce qui le rend facile à trouver**

int main()

**{**

// 0. Déclaration des variables  
 int valA = 5;

int valB = 9;

int resultat = 0 ;

// 1. traitements

resultat = ***AjouteK*** (valA , valB);

printf("%d",resultat); /\* impression de la valeur 14 (0 + 5 + 9)\*/

resultat = ***AjouteK***(valA , valB);

printf("%d",resultat); /\* impression de la valeur 28 (14 + 5 + 9) \*/

return 0 ;

**}**

# ANNEXE B : Les séquences d'échappement

|  |  |  |
| --- | --- | --- |
| \n | NL(LF) | nouvelle ligne |
| \t | HT | tabulation horizontale |
| \v | VT | tabulation verticale (descendre d'une ligne) |
| \a | BEL | sonnerie |
| \b | BS | curseur arrière |
| \r | CR | retour au début de ligne |
| \f | FF | saut de page |
| \\ | \ | trait oblique (back-slash) |
| \? | ? | point d'interrogation |
| \' | ' | apostrophe |
| \" | " | guillemets |
| \0 | NUL | fin de chaîne |

# ANNEXE C : Les priorités des opérateurs

|  |  |  |
| --- | --- | --- |
|  | ***Classes de priorités :*** | ***Ordre de l'évaluation :*** |
| Priorité 1 (la plus forte) | () | -> |
| Priorité 2 | ! ++ -- | <- |
| Priorité 3 | \* / % | -> |
| Priorité 4 | + - | -> |
| Priorité 5 | < <= > >= | -> |
| Priorité 6 | == != | -> |
| Priorité 7 | && | -> |
| Priorité 8 | || | -> |
| Priorité 9 (la plus faible) | = += -= \*= /= %= | <- |

# ANNEXE D : Classification, conversion de caractère : <ctype.h>

***Fonctions de classification et de conversion de caractères***

*Les fonctions suivantes ont des arguments du type* ***int****, dont la valeur est* ***EOF*** *ou peut être représentée comme* ***unsigned char****.*

int isupper(int C)

retourne une valeur différente de zéro, si *C* est une majuscule

int islower(int C)

retourne une valeur différente de zéro, si *C* est une minuscule

int isdigit(int C)

retourne une valeur différente de zéro, si *C* est un chiffre décimal

int isalpha(int C)

retourne une valeur différente de zéro, si **islower(C)** ou **isupper(C)**

int isalnum(int C)

retourne une valeur différente de zéro, si **isalpha(C)** ou **isdigit(C)**

int isxdigit(int C)

retourne une valeur différente de zéro, si *C* est un chiffre hexadécimal

int isspace(int C)

retourne une valeur différente de zéro, si *C* est un signe d'espacement

*Les fonctions de* ***conversion*** *suivantes fournissent une valeur du type* ***int*** *qui peut être représentée comme caractère; la valeur originale de C reste inchangée:*

int tolower(int C)

retourne *C* converti en minuscule si *C* est une majuscule, sinon *C*

int toupper(int C)

retourne *C* converti en majuscule si *C* est une minuscule, sinon *C*

# ANNEXE E : Traitement de chaînes de caractères : <string.h>

int strlen(const char \*CH1) 8.6.2.

fournit la longueur de *CH1 sans* compter le '\0' final

char \*strcpy(char \*CH1, const char \*CH2) 8.6.2.

copie *CH2* vers *CH1* ('\0' inclus); retourne *CH1*

char \*strncpy(char \*CH1, const char \*CH1, int N)

8.6.2.

copie au plus *N* caractères de *CH2* vers *CH1*; retourne *CH1.* Remplit la fin de *CH1* par des '\0' si *CH2* a moins que *N* caractères

char \*strcat(char \*CH1, const char \*CH1) 8.6.2.

ajoute *CH2* à la fin de *CH1*; retourne *CH1*

char \*strncat(char \*CH1, const char \*CH1, int N)

8.6.2.

ajoute au plus *N* caractères de *CH2* à la fin de *CH1* et termine *CH1* par '\0'; retourne *CH1*

int strcmp(const char \*CH1, const char \*CH1)

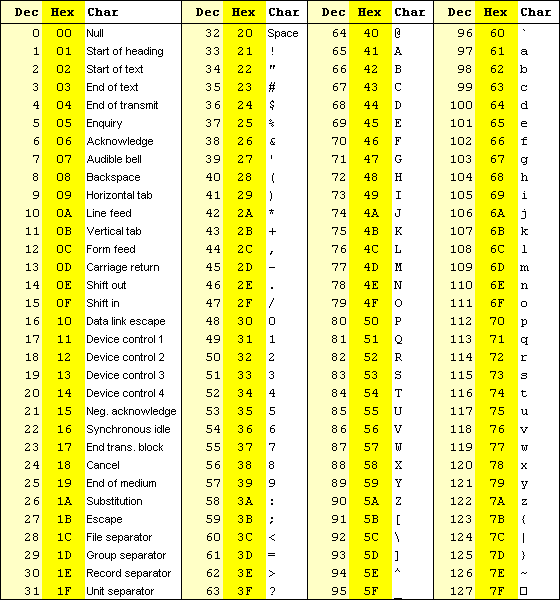
8.5. / 8.6.2.

compare *CH1* et *CH2* lexicographiquement et fournit un résultat:

|  |  |  |
| --- | --- | --- |
|  | négatif | si *CH1* précède *CH2* |
|  | zéro | si *CH1* est égal à *CH2* |
|  | positif | si *CH1* suit *CH2* |

# ANNEXE F : TABLE ASCII ET TABLE ASCII ETENDUE

La table ASCII traduit en caractères les 128 premiers codes.



La table ASCII **étendue** traduit en caractères les 128 codes suivants.

