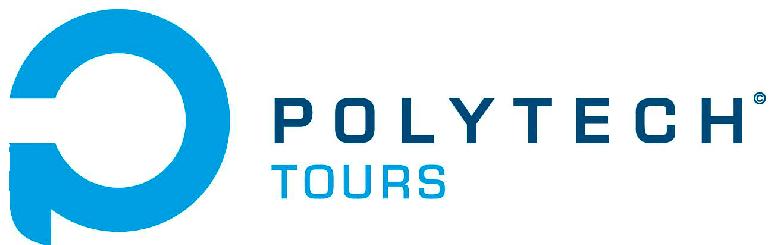

Polytech'Tours
Département Informatique
64 Avenue Jean Portalis
37200 Tours



Programmation en langage C

Sébastien Aupetit

sebastien.aupetit@univ-tours.fr - Bureau 209

Ce document est en cours de rédaction, signalez moi toutes erreurs ou omissions qui permettraient son amélioration pour les années à venir.

Ce document a été réalisé uniquement comme une aide pour les cours, TDs et TPs. Il ne peut se substituer à votre présence pendant les cours, TDs et TPs

Il ne doit pas être diffusé sous quelque forme que ce soit sans autorisation préalable.

Table des matières

1	Organisation de l'enseignement	17
I	Niveau initiation	19
2	Historique	21
1	Les origines dans les Laboratoires Bell	21
1.1	Les origines dans les Laboratoires Bell	21
2	Vers une standardisation du langage	21
2.1	Vers une standardisation du langage	21
3	De nos jours	22
3.1	De nos jours	22
3	Le langage C	23
1	Caractéristiques du langage C.....	23
1.1	Caractéristiques du langage C.....	23
2	Le langage C est un langage structuré.....	24
2.1	Le langage C est un langage structuré.....	24
3	Le langage C est un langage typé.....	24
3.1	Le langage C est un langage typé.....	24
4	Programmes en langage C	25
1	Structure générale d'un programme C	25
1.1	Structure générale d'un programme en C	25
2	Du code source à l'exécutable	25
2.1	Du code source à l'exécutable	25
3	Structure d'un programme en mémoire et appel de fonctions.....	26
3.1	Structure d'un programme en mémoire	26
4	Appel de fonctions	27
4.1	Appel de fonctions	27
5	Premiers éléments de syntaxe	29
1	La syntaxe.....	29
1.1	Mots clés ANSI-C réservés	29
1.2	Mots clés réservés spécifiques à un compilateur/SE/machine	29
1.3	Notation des caractères spéciaux.....	29
1.4	Casse et instructions.....	30
1.5	Quelques fichiers standards d'entête du langage C	30

TABLE DES MATIÈRES

2	Exemples	30
2.1	Exemple 1	30
2.2	Exemple 2	31
2.3	Exemple 3	31
2.4	Exemple 4	31
3	Résumé de la commande printf	32
3.1	Résumé de la commande printf	32
4	Man pages française de la commande printf	32
4.1	NOM	32
4.2	SYNOPSIS	32
4.3	DESCRIPTION	32
4.4	VALEUR RENVOYÉE	32
4.5	CHAÎNE DE FORMAT	33
4.6	CARACTÈRE D'ATTRIBUT	33
4.7	LARGEUR DE CHAMP	34
4.8	PRÉCISION	34
4.9	MODIFICATEUR DE LONGUEUR	34
4.10	INDICATEUR DE CONVERSION	34
4.11	EXEMPLES	35
4.12	NOTES	36
4.13	CONFORMITÉ	36
4.14	HISTORIQUE	37
4.15	BOGUES	37
4.16	VOIR AUSSI	37
4.17	TRADUCTION	37
II	Niveau débutant	39
6	Types de données, Opérateurs et Expressions	41
1	Les types de bases	41
1.1	Les types de bases	41
1.2	Taille et domaine de valeurs	41
1.3	Cas particuliers	42
1.4	Exemple	42
2	Les variables	42
2.1	Les identificateurs et la déclaration de variable	42
2.2	Portée d'une variable	43
2.3	Portée d'une variable locale	43
2.4	Portée des paramètres d'une fonction	44
2.5	Remarques sur les prototypes de fonctions	44
2.6	Portée des variables globales	44
2.7	Exemples	45
2.7.1	Exemple 1	45
2.7.2	Exemple 2	45
2.8	Initialisation de variables	45

TABLE DES MATIÈRES

2.9	Les modificateurs.....	46
2.9.1	Les modificateurs : stockage et accès.....	46
2.9.2	Modificateur d'accès const	46
2.9.3	Modificateur d'accès volatile	46
2.9.4	Modificateur d'accès auto	47
2.9.5	Modificateur d'accès register	47
2.9.6	Modificateur d'accès static	47
2.9.7	Modificateur d'accès extern	47
2.9.8	Résumé et remarque.....	48
2.9.9	Exemple	48
2.10	Les constantes	48
2.10.1	Les constantes : quatre catégories	48
2.10.2	Typage et codage des constantes numériques	49
2.10.3	Exemple	49
3	Les opérateurs	49
3.1	Généralités	49
3.2	L'opérateur d'affectation.....	50
3.3	Opérateurs arithmétiques.....	50
3.4	Pré et post incrémentation/décrémentation.....	51
3.5	Opérateurs logiques sur bits.....	51
3.6	Opérateurs de décalage	52
3.7	Opérateurs relationnels et logiques.....	52
3.8	Opérateurs d'affectation élargie.....	53
3.9	Opérateur sizeof	54
3.10	Opérateur ?	54
3.11	Opérateur séquentiel ,	54
3.12	Pointeurs et opérateurs associés	54
3.13	Les conversions	55
3.13.1	Conversion implicite de types au sein d'une expression.....	55
3.13.2	Conversion explicite de type : cast	55
3.14	Priorités des opérateurs	56
3.15	Exemple de pré/post incrémentation/décrémentation.....	56
4	Les tableaux, les chaînes et les structures.....	58
4.1	Les tableaux 1D.....	58
4.1.1	Généralités sur les tableaux	58
4.1.2	Tableaux à une dimension	58
4.2	Les chaînes de caractères	59
4.2.1	Convention	59
4.2.2	Déclaration.....	59
4.2.3	Caractères et chaînes de caractères	59
4.3	Les structures	60
4.3.1	Définitions, déclarations et utilisation	60
4.3.2	Occupation mémoire.....	60
4.3.3	Initialisation	61
4.3.4	Tableau de structure.....	61
4.3.5	Pointeurs sur une structure.....	61
4.3.6	Structures et fonctions.....	62
4.3.7	Exemple	62

TABLE DES MATIÈRES

7 Instructions, structures de contrôle et fonctions	63
1 Instructions et bloc d'instructions	63
2 Structures de contrôle.....	63
2.1 Structures de contrôle.....	63
2.2 Sélection.....	63
2.2.1 Si Alors Sinon	63
2.2.2 Switch	65
2.3 Itérations.....	67
2.3.1 for	67
2.3.2 while	68
2.3.3 do while	69
2.3.4 Les boucles d'itération.....	69
2.4 Sauts	70
2.4.1 goto	70
2.4.2 break	70
2.4.3 continue	70
2.4.4 Exemple.....	70
3 Fonctions.....	71
3.1 Définitions et syntaxe	71
3.1.1 Définition	71
3.1.2 Syntaxe	71
3.1.3 Prototype de fonctions.....	71
3.2 Passage de paramètres	72
3.2.1 Passage de paramètres par valeur	72
3.2.2 Passage de paramètres par adresse.....	72
3.3 Renvoyer des résultats	73
3.3.1 Retourner une valeur : return	73
3.3.2 Comment retourner plusieurs valeurs : passage par adresse	73
3.3.3 Appel de fonctions.....	73
3.3.4 La récurrence	73
3.3.5 Exemple 1	74
3.3.6 Exemple 2	74
3.4 La fonction main	74
3.4.1 Arguments de la fonction main	74
3.4.2 Exemple	75
3.4.3 Structures et prototype de fonctions	75
8 Quelques fonctions simples et étude de codes existants	77
1 L'écran et le clavier	77
1.1 Sorties de base : affichage écran	77
1.2 Entrées de base : saisie clavier	77
2 Fonctions de bases	78
2.1 Les principales fonctions mathématiques	78
2.2 Caractères et chaînes de caractères	78
2.3 Fonctions diverses.....	79

TABLE DES MATIÈRES

3	Etude de codes existants	79
3.1	Exemple 1	79
3.2	Exemple 2	80
3.3	Exemple 3	80
3.4	Exemple 4	81
3.5	Exemple 5	81
3.6	Exemple 6	82
3.7	Exemple 7	82
3.8	Exemple 8	82
3.9	Exemple 9	82
3.10	Exemple 10.....	83
9	Le C en pratique	85
1	Fichiers sources et fichiers d'entête.....	85
1.1	Structures des fichiers sources	85
1.2	Exemple de fichier avant la séparation	85
1.3	Exemple de fichiers après la séparation : produit.h	87
1.4	Exemple de fichiers après la séparation : produit.c	87
1.5	Exemple de fichiers après la séparation : catalogue.h	87
1.6	Exemple de fichiers après la séparation : catalogue.c	87
1.7	Exemple de fichiers après la séparation : facture .h	88
1.8	Exemple de fichiers après la séparation : facture .c	88
1.9	Exemple de fichiers après la séparation : main.c	89
1.10	Dépendances entre les modules	89
2	Mise en forme de fichiers sources.....	89
2.1	Mise en forme	89
2.1.1	Mise en forme de code source.....	89
2.1.2	Quelques exemples de mise en forme classique	90
2.2	Bonnes pratiques	92
2.2.1	Nommer les choses utilement	92
2.2.2	Quelques conventions de noms.....	93
2.2.3	Commenter utilement	93
2.3	Doxygen	94
2.3.1	Documenter utilement avec Doxygen	94
2.3.2	Exemple	94
2.3.3	Documentation générée.....	95
2.3.4	Quelques balises utiles	96
2.4	Conseils de codage en langage C	97
3	Bibliothèques de code	97
3.1	Bibliothèques de code	97
3.2	Bibliothèque statique	98
3.3	Bibliothèque dynamique	98
3.4	Bibliothèques	99
4	Compilation par la pratique	99
4.1	Visual Studio 6.....	99

TABLE DES MATIÈRES

4.1.1	Création du projet et des fichiers	99
4.1.2	Les menus	104
4.1.3	Le projet	106
4.1.4	Construire et debugger un projet	109
4.1.5	Configuration de l'IDE	112
4.1.6	Ouverture d'un projet	114
4.1.7	Configurations du projet et fichiers produits	115
4.2	Visual Studio .Net	117
4.3	Dev C++	117
4.3.1	Création du projet et des fichiers	117
4.3.2	Les Menus	120
4.4	GNU gcc	123
4.4.1	gcc	123
4.4.2	Les frontends	124
4.4.3	Attention	124
III	Niveau intermédiaire	125
10	Types avancés	127
1	Champs de bits, unions et énumérations	127
1.1	Champs de bits	127
1.2	Unions	128
1.3	Enumérations	128
2	Typedef	129
2.1	typedef	129
2.2	Exemple 1	130
2.3	Exemple 2	131
2.4	Structures récursives	131
3	Tableaux	132
3.1	Rappel sur les tableaux	132
3.2	Rappel sur les tableaux 1D	132
3.3	Tableau 2D	133
3.4	Tableaux à N dimensions	134
3.5	Tableaux 1D et fonctions	134
3.6	Tableaux 2D/ND et fonctions	135
11	Les pointeurs	137
1	Les pointeurs	137
1.1	Rappel de l'essentiel	137
1.2	Opérations sur les pointeurs	137
1.3	L'arithmétique des pointeurs	138
1.4	Exemples	138
2	Tableaux et pointeurs	138
2.1	Tableaux 1D et pointeurs	138
2.1.1	Dualité entre tableaux 1D et pointeurs	138

TABLE DES MATIÈRES

2.1.2	Arithmétique des pointeurs.....	139
2.1.3	Types.....	139
2.1.4	Fonctions, Tableaux 1D et pointeurs.....	139
2.2	Tableaux 2D et pointeurs.....	139
2.2.1	Dualité entre tableaux 2D et pointeurs	139
2.2.2	Arithmétique des pointeurs.....	140
2.2.3	Types.....	140
2.2.4	Fonctions, Tableaux 2D et pointeurs.....	140
2.3	Exemples.....	141
2.3.1	Exemple 1	141
2.3.2	Exemple 2	141
2.3.3	Exemple 3	142
3	Erreurs courantes (et souvent fatales)	142
3.1	Déclaration de pointeurs	142
3.2	Utilisation de pointeurs non initialisés.....	142
3.3	Affectation du contenu d'une variable à un pointeur.....	143
3.4	Valeur pointée par p+n	143
3.5	Priorité des opérateurs.....	143
4	Remarques.....	143
4.1	Les chaînes de caractères	143
4.2	Les pointeurs génériques void*	143

12 Allocation dynamique	145	
1	Allocation, libération et réallocation	145
1.1	Rappel sur l'organisation de la mémoire et les variables	145
1.2	Variables dynamiques	145
1.3	Allocation de mémoire	145
1.3.1	malloc	145
1.3.2	calloc	146
1.4	Libération de mémoire avec free	147
1.5	Réallocation mémoire avec realloc	148
1.6	SIZE_MAX en C99.....	148
2	Applications	148
2.1	Tableaux 1D dynamiques	148
2.1.1	Tableau 1D.....	148
2.1.2	Exemple	149
2.2	Tableaux 2D semi-dynamiques	149
2.2.1	Tableau 2D semi-dynamique	149
2.2.2	Exemple	150
2.3	Tableaux 2D dynamiques	150
2.3.1	Tableau 2D dynamique	150
2.3.2	Exemple	151
2.4	Listes simplement chaînées.....	151
2.4.1	Listes simplement chaînées	151
2.4.2	Insertion et suppression	152

TABLE DES MATIÈRES

2.4.3	Parcours.....	152
2.4.4	Recherche.....	153
2.4.5	Programme principal	153
3	Erreurs courantes avec les pointeurs.....	153
3.1	Il faut vérifier le résultat d'une allocation.....	153
3.2	Confusion entre le segment de données, la pile et le tas.....	154
3.3	Libération ou réallocation sur des pointeurs calculés	154
3.4	Utilisation d'un pointeur après libération	154
3.5	Utilisation d'un pointeur avant allocation	154
3.6	Pertes de mémoire/memory leaks.....	154
4	La bibliothèque string.h	154
4.1	La bibliothèque string.h	154
4.2	Fonctions de manipulation de zones mémoires en C99	155
4.3	Fonctions de manipulation de chaînes de caractères.....	155
13 Fichiers		157
1	Généralités	157
1.1	Fichiers et flux.....	157
1.2	Flux/fichiers en C.....	158
2	Ouverture et fermeture	158
2.1	Ouverture de flux/fichier.....	158
2.2	Fermeture de flux/fichier.....	159
3	Lectures et écritures	159
3.1	Lecture/écriture d'un caractère	159
3.2	Lecture/écriture d'une chaîne de caractères.....	159
3.3	Lecture/écriture d'octets.....	160
3.4	Padding/alignement de données en mémoire	160
3.5	Positionnement du curseur de lecture/écriture.....	161
3.6	Formattage des entrées/sorties	161
4	Notions avancées.....	161
4.1	Gestion des buffers d'E/S	161
4.2	Gestion des erreurs	162
4.3	Flux prédéfinis.....	162
4.4	Redirection de flux déjà ouverts.....	162
IV Niveau avancé		163
14 Compléments		165
1	Compléments sur les fonctions	165
1.1	Les pointeurs de fonctions.....	165
1.1.1	Pointeurs de fonctions	165
1.1.2	Exemple 1	165
1.1.3	Exemple 2	166
1.2	Les fonctions variadiques	166
1.2.1	Fonctions à nombre d'arguments variables	166

TABLE DES MATIÈRES

1.2.2	Exemples.....	167
2	Le préprocesseur	168
2.1	Rappel sur la compilation	168
2.2	#define : définition de constantes.....	168
2.3	#define : définition de macros.....	169
2.4	#include	170
2.5	Les directives de compilation conditionnelle	170
2.6	Tests de définition de symbole	171
2.7	Compilation conditionnelle et mise au point.....	171
2.8	Autres commandes	172
2.9	_LINE_ , _FILE_ et #line	172
2.10	Les opérateurs # et ##	172
2.11	Quelques symboles utiles.....	173
3	Etude de codes sources	173
3.1	Exemple 1	173
3.2	Exemple 2	174
3.3	Exemple 3	174
3.4	Exemple 4	174
3.5	Exemple 5	175
3.6	Exemple 6	176
3.7	Exemple 7	177
15	Compilation avancée et outils	181
1	Gestion des types de données interdépendants.....	181
1.1	Un exemple	181
1.2	Le problème.....	181
1.3	La solution	181
16	Mieux programmer	183
1	Gestion et détection d'erreurs	183
1.1	Compilation en mode debug/release.....	183
1.2	Mise en place de contrats avec assert	183
1.3	Gestion des erreurs avec errno	184
1.4	Valeur de retour d'un programme	185
1.5	Les options de compilation.....	185
2	Les principales nouveautés du C99	185
2.1	Les nouveaux types de données	185
2.1.1	Les entiers 64bits.....	185
2.1.2	Les réels à haute précision	185
2.1.3	Les booléens	185
2.1.4	Les nombres complexes.....	186
2.1.5	Le fichier d'entête stdint.h	186
2.2	Les nouveaux mots clés.....	187
2.2.1	inline	187
2.2.2	restrict	188
2.2.3	Les macros à nombre variable d'arguments	188

TABLE DES MATIÈRES

2.3	Amélioration de la syntaxe	188
2.3.1	Les commentaires	188
2.3.2	Mélange entre déclaration et code.....	188
2.3.3	Nécessité de spécifier le type de retour d'une fonction	189
2.3.4	Variable Length Array	189
2.4	La norme C99 en pratique	190
3	Les signaux.....	190
3.1	Rappels	190
3.2	Les processus en détail.....	190
3.2.1	Création et identification d'un processus.....	190
3.2.2	Remplacement de contexte	190
3.2.3	Mort naturelle d'un processus.....	191
3.2.4	Synchronisation	191
3.2.5	Mise en sommeil d'un processus.....	191
3.3	Communication inter-processus.....	191
3.3.1	Les fichiers	192
3.3.2	Les signaux	192
3.3.3	Les tubes ("pipes").....	193
3.4	Les processus légers ("Threads").....	193
3.4.1	Processus vs thread	193
3.4.2	La bibliothèque "pthread"	194
3.4.3	Création d'un thread	194
3.4.4	Fin d'un thread	194
3.4.5	Synchronisation sur la fin d'un thread.....	194
3.4.6	Les mutex	195
4	Mesurer le temps	196
4.1	Généralités	196
4.1.1	Temps d'horloge et temps processus.....	196
4.2	Temps processus	197
4.2.1	La fonction clock (C89,C99,POSIX)	197
4.2.2	La fonction getrusage (POSIX)	197
4.2.3	La fonction times (POSIX)	198
4.3	Temps d'horloge	198
4.3.1	Obtenir l'heure	198
4.3.2	Calculer des intervalles	198
4.3.3	Conversion de date en chaînes de caractères.....	198
5	Les nombres aléatoires	199
5.1	Les fonctions	199
5.2	Cas particuliers.....	199
5.3	Remarque sur la qualité du générateur aléatoire du C	199

TABLE DES MATIÈRES

17 Interface graphique avec GTK+	201
1 Origines et architecture générale	201
1.1 GTK+.....	201
1.2 Installation	201
2 Un exemple pour commencer	202
2.1 Un premier exemple.....	202
2.2 Compilation.....	202
3 Objets, Signaux et fonctions callback	203
3.1 Deuxième exemple	203
3.2 Fonctions de rappel/callback et signaux	203
3.3 Gestion des signaux	204
3.4 Troisième exemple	205
3.5 Création et utilisation de widget : 5 étapes	205
3.6 Hiérarchie des objets GTK+	206
3.7 Programmation orientée objet GTK+ et langage C.....	206
4 Quelques points d'entrée	207
4.1 Quelques widgets de base.....	207
4.2 Aller plus loin avec GTK+	207
V Niveau expert	209
18 Etes-vous prêt pour le niveau expert ?	211
1 Objectifs et principes.....	211
2 Questionnaire	211
VI Exemple complet	213
19 Un exemple complet	215
1 Gestion des erreurs	215
2 Gestion des instances et des structures.....	215
3 Les types classiques	216
4 Des tableaux génériques.....	217
5 Des chaînes de caractères	219
6 Correction	219
6.1 Fatal.h	219
6.2 Fatal.c	219
6.3 Instance.h	220
6.4 Instance.c	221
6.5 StdInstance.h	222
6.6 StdInstance.c	223
6.7 Array.h	223
6.8 Array.c	223
6.9 MyString.h	224
6.10 MyString.c	225
6.11 main.c	226

TABLE DES MATIÈRES

VII Exercices complémentaires	229
20 Les bases du langage C (structures de contrôle, fonctions, paramètres, syntaxe, .h)	231
1 Exercices sur les nombres.....	231
1.1 Troncature d'un nombre réel.....	231
1.2 Arrondi d'un nombre réel.....	231
1.3 Suppression de toutes les occurrences d'un caractère dans une chaîne de caractères.....	231
1.4 Suppression des caractères communs à deux chaînes de caractères.....	231
1.5 Concaténation de deux chaînes de caractères.....	231
2 Exercices sur les tableaux	231
2.1 Affectation pseudo-aléatoire des éléments d'un tableau d'entiers.....	232
2.2 Recherche de la valeur max au sein d'un tableau.....	232
2.3 Présence / absence d'un élément au sein d'un tableau.....	232
2.4 Recherche du 2ème plus petit élément au sein d'un tableau (exercice complémentaire)	232
3 Questions diverses.....	232
3.1 Question 1	232
3.2 Question 2	232
3.3 Question 3	232
3.4 Question 4	233
3.5 Question 5	233
3.6 Question 6	233
3.7 Question 7	233
3.8 Question 8	233
3.9 Question 9	234
3.10 Question 10	234
21 Types complexes et allocation mémoire	235
1 Les listes chaînées	235
1.1 Structure des données utilisées	235
1.2 Opérations sur listes simplement chaînées	235
1.2.1 Création des types de données	235
1.2.2 Saisie des champs d'information de I.....	235
1.2.3 Affichage des champs d'information de I	235
1.2.4 Création d'un élément	235
1.2.5 Insertion d'un élément en tête de liste.....	236
1.2.6 Retrait du premier élément de la liste.....	236
1.2.7 Insertion d'un élément à un emplacement quelconque	236
1.2.8 Retrait d'un élément à un emplacement quelconque	236
1.2.9 Parcours de la liste	236
1.2.10 Suppression de tous les éléments de la liste	236
1.2.11 Fonction main	236
2 Manipulation de tableaux dynamiques	236
2.1 Présentation du sujet	236
2.2 Travail demandé	237

TABLE DES MATIÈRES

22 Les caractères, les chaînes, les pointeurs et l'allocation dynamique	239
1 Traitement de chaînes de caractères	239
1.1 Comparaison de chaînes.....	239
1.2 Transcription d'algorithme.....	239
2 Évaluation de la distribution d'une série de tirages de dés	239
2.1 Lancé de Dés	239
2.2 Affichage d'histogramme	240
2.3 Normalisation de données	240
2.4 Fonction « main »	240
23 Les fichiers	241
1 Gestion d'un carnet d'adresses de clients/fournisseurs	241
1.1 La structure de données.....	241
1.2 Gestion non triée du carnet d'adresse.....	241
1.3 Gestion triée du carnet d'adresse	242
2 Création d'un fichier de log d'activités	242
3 Fusion du contenu de deux fichiers.....	243
24 Pointeurs de fonctions, programmation générique et callback, GLIB/GTK+	245
0.1 Une calculatrice en notation polonaise inverse.....	245
0.1.1 La notation polonaise inverse.....	245
0.1.2 Structure des données utilisées.....	245
0.1.3 Travail demandé.....	245
0.1.4 Exercices complémentaires.....	246
1 Premiers pas avec GTK+	246
2 Une petite application réelle : le chronomètre.....	246
25 Compilation avancée, utilisation de bibliothèques et fichiers .h complexes	247
1 Utilisation des directives du préprocesseur	247
2 Programmation modulaire	247
2.1 Module de gestion d'arbre binaire	247
2.2 Module de parcours d'arbre	247
2.3 Module de test et Makefile.....	247
2.4 Pour aller plus loin	247

1

Organisation de l'enseignement

Enseignement

- 18h de Cours
- 14h de TDs
- 12h de TPs

Structuration des cours/TDs/TPs

- niveau initiation : aperçu
- niveau débutant : 50% des besoins
 - TDs/TPs
- niveau intermédiaire : 90% des besoins
 - TDs/TPs
- niveau avancé : 110% des besoins
 - TDs/TPs/Projets/L'expérience
- niveau expert : 200% des besoins
 - Projets/L'expérience

Objectifs

- Cours : connaître la syntaxe, savoir lire et comprendre du C
- TDs/TPs : savoir expliquer, écrire, compiler et débugger des programmes

Evaluation : contrôle continu

- 100% ou 75% : Participation aux TDs/TPs, Projet TDs/TPs guidés (en monome), Prise en compte de l'absentéisme
- 0% ou 25% : Interrogation(s) surprise(s)

Projet de C (au S6)

- En binôme
- En liaison avec l'algorithme et le génie logiciel
- Des consignes spécifiques vous seront transmises plus tard

Première partie

Niveau initiation

2

Historique

1 Les origines dans les Laboratoires Bell

1.1 Les origines dans les Laboratoires Bell

Années 1970

Le système d'exploitation UNIX :

- trop proche de la machine
- difficile à adapter à de nouvelles plateformes
- les langages de programmation utilisés ne sont pas assez «portables»

1970

Kenneth Thompson crée le langage B inspiré de BCPL mais il est peu performant

1972

Dennis M. Ritchie crée le langage C :

- évolution du langage B
- beaucoup plus performant
- le système UNIX est réécrit en grande partie en C

2 Vers une standardisation du langage

2.1 Vers une standardisation du langage

Objectifs de la standardisation du langage C

- ne plus dépendre de la machine
- ne plus dépendre du système d'exploitation

1978

Brian W. Kernighan et Dennis M. Ritchie :

- publient « The C Programming Language »
- devient une « norme » de fait : **K&R-C**

ANSI-C

- 1983 : Début des travaux de normalisation de l'ANSI¹
- 1989 : l'ANSI publie la norme **ANSI-C** aussi connue comme norme **C89**
 - fortement inspiré de K&R-C
 - 99.9% du ANSI-C est compatible avec le K&R-C

1990

L'ISO² adopte sans changement la norme ANSI-C. Cette norme porte également le nom de **ISO-C**.

1999

L'ISO publie une nouvelle évolution du langage : la norme **C99**.

1. American National Standard Institute / Institut national américain de normalisation
2. International Standards Organization / Organisation internationale de standardisation

3 De nos jours

3.1 De nos jours

Le langage C a beaucoup de succès

- De nombreux compilateurs C ont été créés :
 - sur de nombreuses architectures matérielles
 - sur de nombreux systèmes d'exploitation
- Tous les compilateurs C respectent (au moins) la norme ANSI-C

Mais

- la description actuelle du langage est trop vague \implies les compilateurs sont incompatibles entre eux
- le bibliothèque standard de fonction est trop limitée \implies les compilateurs étendent les fonctions disponibles

\implies Malgré ses objectifs initiaux, les programmes ne sont pas aussi portables que prévus initialement

✓ L'essentiel à retenir :

1. Qu'est-ce que le K&R-C ?
2. Qu'est-ce que l'ANSI-C ?
3. Qu'est-ce que le C89 ?
4. Qu'est-ce que le C99 ?

3

Le langage C

1 Caractéristiques du langage C

1.1 Caractéristiques du langage C

Universel

- s'applique à tous les domaines d'application : système d'exploitation, programmation réseau, système de gestion, bureautique, web, petite application « proche du matériel », grosse application « indépendante » du matériel...
- standard ouvert depuis les années 1970
- le langage C est connu par de très nombreux informaticiens

Compact

- un nombre limité de concepts
- un nombre limité d'opérateurs
- un nombre limité de fonctions de base

Moderne

- structuré : syntaxe relativement rigide
- déclaratif : les variables doivent être déclarées avant d'être utilisées
- procédural : les instructions sont regroupées au sein de fonction (gestion d'une pile d'appel, paramètre de fonction, variables locales, valeur de retour)
- récursif : une fonction peut appeler une fonction (autre ou elle-même)

Proche de la « machine », langage de bas niveau

- les types de base dépendent des types de la machine (mots, entiers et flottants du processeur...)
- utilisation intensive des adresses mémoires
- pas de types de données de haut niveau (liste, fichier, chaîne de caractère...) et donc pas d'opérateur de haut niveau (concaténation...)
- de nombreuses vérifications sont confiées au programmeur \Rightarrow **de nombreux bogues !!**

Rapide

- Souvent aussi/plus rapide que du code assembleur
- ce n'est pas de l'assembleur mais reste très proche de la machine
 - les optimisations du compilateur sont souvent plus efficace que ce que fait l'être humain
 - les normes ne spécifient pas la forme finale des exécutables : ils tirent donc profit au mieux de la plateforme cible

Portable

- Des compilateurs C existent pour quasiment tous les types de machines et tous les systèmes d'exploitation
- Architectures x86, IA64, AMD64, PowerPC, RISC, CISC, microcontroller...
 - DOS, Windows, Linux, MacOS, BSD, Unix...
 - le langage C est simple donc le portage du compilateur vers une nouvelle plateforme est rapide
 - **Mais** les variabilités dans les programmes exécutables produits rendent le portage pas si direct que ça

3 Le langage C

2 Le langage C est un langage structuré

2.1 Le langage C est un langage structuré

Séparation

- du code
- et des données

Langage procédural

- le code est organisé sous forme de fonctions
- une fonction peut avoir des variables locales
- une fonction peut retourner une valeur
- une fonction peut s'appeler ou appeler d'autres fonctions

Programmation modulaire

- un programme peut être découpé en plusieurs fichiers sources
- du code source peut être réutilisé dans différents programmes grâce à l'utilisation de bibliothèque de code

3 Le langage C est un langage typé

3.1 Le langage C est un langage typé

Variable

zone mémoire pouvant contenir une information / une valeur

Type de données

ensemble des valeurs qu'une variable peut prendre

Propriétés

- Une variable possède un type de données
- Une variable ne peut changer de type de données
- On distingue les types simples (nombres...) et les types composés (struct)
- Des opérateurs sont associés à chaque type simple (+,-...)
- Toute variable doit être déclarée avant de pouvoir être utilisée
- La valeur initiale d'une variable est aléatoire : **les variables ne sont pas initialisées par le compilateur**

L'essentiel à retenir :

1. Le langage C est un langage structuré.
2. Le langage C est un langage typé.
3. Le langage C est un langage proche de la machine.

4

Programmes en langage C

1 Structure générale d'un programme C

1.1 Structure générale d'un programme en C

Pour un programme ne comportant qu'un fichier source

1 - Les directives du préprocesseur

- informe le compilateur qu'un ensemble de traitements doit être appliqué au code source avant la traduction en instruction machine

2 - La définition de types de données

- les nouveaux types de données composés sont déclarés
- les nouveaux noms de type de données existants sont déclarés
- les types partiellement définis sont déclarés

3 - Les prototypes des fonctions

- une fonction doit être déclarée ou définie avant de pouvoir être utilisée

4 - La déclaration de variables globales

- les variables globales du programme utilisables dans le fichier source sont importées
- les variables globales au fichier source sont déclarées

5 - Le code des fonctions

- Une fonction ne contient que des instructions et des variables locales
- Dans un programme, seule la fonction **main** est **nécessaire et suffisante**. C'est avec elle que tout commence !

Et quand on a plusieurs fichiers sources ?

- Chaque fichier obéit à la même structure
- Mais il ne peut y avoir qu'une seule fonction **main** sur l'ensemble des fichiers

La fonction **main**

- c'est une fonction comme les autres
- c'est le point de départ du programme
- mais le type de la valeur renvoyée et de ses paramètres éventuels sont définis par le langage C

2 Du code source à l'exécutable

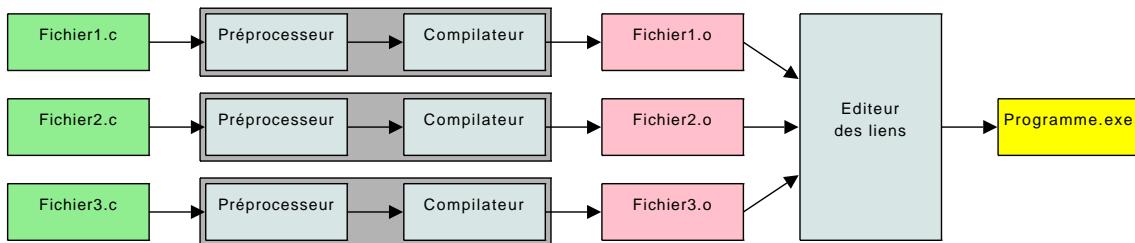
2.1 Du code source à l'exécutable

Trois étapes

1. **Création** du ou des fichiers sources (ex : **nom1.c**, **fichier2 .c**)
2. **Compilation** des fichiers sources :
 - Chaque fichier **.c** donne un fichier objet associé (ex : **nom1.o**, **fichier2 .obj**)
 - Un fichier objet contient du code machine
3. **Édition des liens**

4 Programmes en langage C

- Réunit tout le code objet des fichiers compilés et des bibliothèques de fonctions utilisées
- Fabrique un unique fichier exécutable (ex : **monprog.exe**)



Fichier source = source file

Fichier texte contenant les instructions C.

Code objet = object code

Suite d'instructions machine (en binaire). Les adresses en mémoire ne sont pas forcément toutes résolues.

Préprocesseur = preprocessor

Outil permettant de « rechercher/remplacer, insérer, supprimer du code source ». Le résultat est un fichier en langage C ne contenant plus d'instructions du préprocesseur.

Compilateur = compiler

Outil de traduction des instructions C en code machine. Le résultat est un fichier de code objet.

Editeur des liens = linker

Outil d'assemblage de fichiers objets avec résolution des adresses des fonctions et des variables non encore résolues. Le résultat est un programme exécutable.

Remarque

Un programme C utilise toujours la bibliothèque de fonctions C-ANSI

- assure la portabilité des programmes,
- assure l'indépendance par rapport au système d'exploitation ou de la machine

Remarque

Il existe des bibliothèques de fonction spécifiques :

- à un système d'exploitation
- ou à une machine (ex : fonction graphique, réseaux, ...)

⇒ on peut perdre la portabilité si la bibliothèque n'existe pas sur un SE ou une machine

Remarque

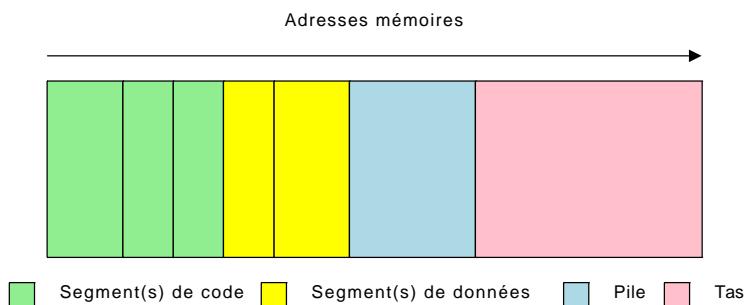
Pour utiliser des fonctions de bibliothèques ou des fonctions définies dans un autre fichiers sources :

- on utilise des fichiers d'entête (ex : **fichier2.h**)
- et l'instruction **#include <fichier2.h>** ou **#include "fichier2.h"**

3 Structure d'un programme en mémoire et appel de fonctions

3.1 Structure d'un programme en mémoire

- Chargement du programme en mémoire ⇒ cf. cours de SE
- Quatre zones en mémoire :



Segment de code = code segment

Zone contigue de la mémoire :

- en lecture seulement
- contient les instructions en code machine à exécuter
- peut contenir des constantes du type chaîne de caractères
- un programme peut en avoir plusieurs

Segment de données = data segment

Zone contigue de la mémoire :

- zones mémoires réservées dans le code source
- contient les variables globales
- un programme peut en avoir plusieurs

Tas = heap

Zone contigue de la mémoire :

- création dynamique (à l'exécution) de zones de données pendant l'exécution
- ex : enregistrements, files, piles, arbres, ...
- un programme peut en posséder en général qu'une seule

Pile d'appel = stack

Zone contigue de la mémoire :

- lieu où sont empilés des éléments lors de l'appel d'une fonction
- contient des variables locales
- un programme peut en posséder en général qu'une seule

4 Appel de fonctions

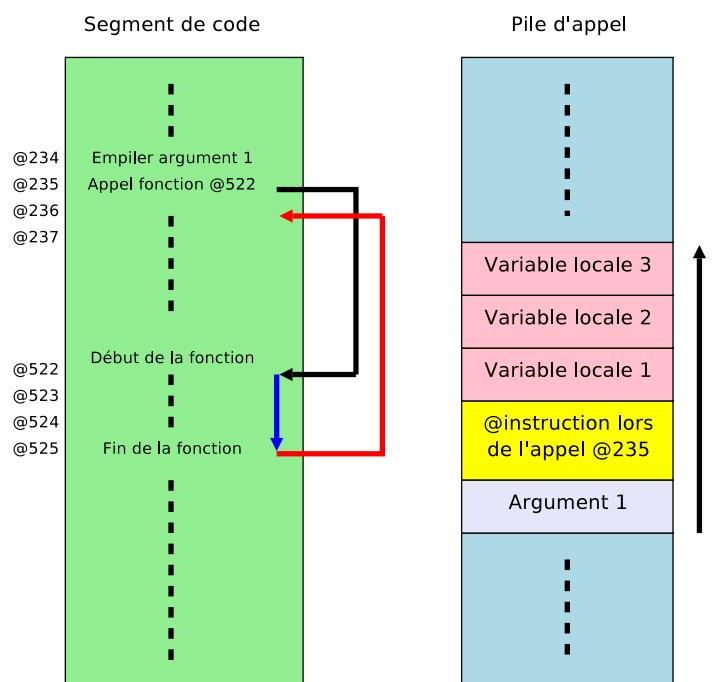
4.1 Appel de fonctions

Éléments empilés lors de l'appel d'une fonction

- les arguments transmis à la fonction
- l'emplacement du curseur d'instruction avant l'appel de la fonction
- les variables locales nécessaires à l'exécution de la fonction

Lorsque la fonction se termine

- les éléments sont dépiler en sens inverse
- le curseur d'instruction est restauré à la valeur dépiler et l'exécution continue à l'instruction suivante
- les éventuels paramètres transmis sont dépiler



 **L'essentiel à retenir :**

1. Quelle est l'organisation d'un fichier source en C ?
2. Quelles sont les étapes de transformation d'un code source en programme exécutable ?
3. Quelle est l'organisation d'un programme en mémoire ?
4. Comment se déroule l'appel d'une fonction ?

5

Premiers éléments de syntaxe

1 La syntaxe

1.1 Mots clés ANSI-C réservés

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

Il faut donc faire attention à ne pas déclarer de variable ou de fonction portant un nom réservé.

1.2 Mots clés réservés spécifiques à un compilateur/SE/machine

- Exemple de MS Visual Studio (liste incomplète)

__asm	__dlimport	__int8	__naked
__based	__except	__int16	__stdcall
__cdecl	__fastcall	__int32	__thread
__declspec	__finally	__int64	__try
__dllexport	__inline	__leave	

- Exemple de GCC (liste incomplète) :

__label__	__builtin_apply_args	__builtin_apply
__typeof__	ULL	__complex__
__builtin_return	VA_ARGS	

Vous pouvez obtenir la liste complète des extensions de GCC pour la langage C à l'adresse <http://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html>.

1.3 Notation des caractères spéciaux

Séquence	Valeur
\a	Signal sonore (ne fonctionne pas toujours)
\b	Effacement arrière
\f	Saut de page (ne fonctionne pas toujours)
\n	Changement de ligne
\r	Retour chariot
\t	Translation horizontale
\v	Translation verticale
\\	Barre inverse \
\'	Apostrophe '
\"	Guillemet "

5 Premiers éléments de syntaxe

Séquence	Valeur
\?	Point d'interrogation ?
\DDD	Caractère n°DDD (en octal, base 8) de la table ASCII
\xHH	Caractère n°HH (en hexadécimal, base 16) de la table ASCII

1.4 Casse et instructions

Le langage C distingue la casse

- majuscules ou minuscules c'est différent !
- **mavariable**, **MaVariable**, **MAVARIABLE**, **MaVaRiAbLe**

Marque de fin d'instruction

Toutes les instructions en C se terminent par \;

Une instruction

- Une déclaration
- Une commande
- Un bloc d'instructions

Un bloc d'instruction

- Suite d'instructions entre \{ et \}

1.5 Quelques fichiers standards d'entête du langage C

<assert.h>	Fournit une définition de la macro assert
<ctype.h>	macros de classification et de conversion de caractères
<errno.h>	définitions des mnémoniques des codes d'erreurs
<float.h>	paramètres pour les primitives de l'arithmétique flottante
<limits.h>	paramètres d'environnement, limitation de la compilation, limites des opérandes entières
<locale.h>	prise en compte de spécificités locales liées à un pays (pas toujours disponible)
<math.h>	fonctions mathématiques
<signal.h>	traitement des signaux (conditions peuvent survenir durant l'exécution d'un programme)
<stdarg.h>	fonctions à nombre variable d'arguments
<stddef.h>	définition de type de données globales et macros
<stdio.h>	types de données et macros liées aux entrées/sorties
<stdlib.h>	déclaration de primitives de conversion, de recherche et de tri
<string.h>	manipulation de chaîne de caractères et de zones mémoires
<time.h>	primitives liées à la conversion de temps

2 Exemples

2.1 Exemple 1

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Salut !\n");
6     return 0;
7 }
```

Remarques

- **#include <stdio.h>** : stdio.h est recherché dans les répertoires des bibliothèques
- **#include "monfichier.h"** : monfichier.h est dans le répertoire de travail
- **#include "c:\toto\monfichier.h"** : monfichier.h est dans le répertoire c :\toto\

2.2 Exemple 2

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     int fahr, celsius, min, max, pas;
6     min = 0;
7     max = 300;
8     pas = 20;
9     fahr = min;
10    while (fahr <= max)
11    {
12        celcius = 5 * (fahr - 32) / 9;
13        printf ("%d°F\t%d°C\n", fahr, celcius);
14        fahr += pas;
15    }
16    return 0;
17 }
```

Remarques

- `celcius = 5 * (fahr - 32) / 9` utilise la division entière ($5/9 == 0$)

2.3 Exemple 3

```

1 #include <stdio.h>
2 #define MIN 0
3 #define MAX 300
4 #define PAS 20
5
6 int main(void)
7 {
8     float fahr, celsius;
9     fahr = MIN;
10    while (fahr <= MAX)
11    {
12        celcius = 5.0 / 9.0 * (fahr - 32.0);
13        printf ("%5.1f°F\t%4.2f°C\n", fahr, celcius);
14        fahr += PAS;
15    }
16    return 0;
17 }
```

Remarques

- `celcius = 5.0 / 9.0 * (fahr - 32.0)` utilise l'arithmétique réelle
- `%4.2f` affichage sur au moins 4 caractères dont 2 chiffres décimaux

2.4 Exemple 4

```

1 #include <stdio.h>
2
3 #define MIN 0
4 #define MAX 300
5 #define PAS 20
6
7 int main(void)
8 {
9     int a, b, s;
10
11    printf("Entrer deux nombres\n");
12    scanf ("%d%d", &a, &b);
13    s = a + b;
14    printf ("La somme de %d et %d est : %d\n", a, b, s);
15
16    return 0;
17 }
```

Remarques

- Saisie de 2 valeurs entières : passage des paramètres par adresse

3 Résumé de la commande printf

3.1 Résumé de la commande printf

Syntaxe

```
printf( Format, Liste de paramètres );
```

Format

- Les caractères ordinaires sont affichés tels quels
- Les codes de caractères spéciaux sont traduits (\n, ...)
- %d : affiche un entier de type int
- %f, %lf : affiche un réel de type float ou de type double

Lisez la documentation !

4 Man pages française de la commande printf

Extrait du « Manuel du programmeur Linux ».

4.1 NOM

printf, fprintf, sprintf, snprintf, vprintf, vfprintf, vsprintf, vsnprintf - Formatage des sorties.

4.2 SYNOPSIS

```
1 #include <stdio.h>
2 int printf (const char *format, ...);
3 int fprintf (FILE *stream, const char *format, ...);
4 int sprintf (char *str, const char *format, ...);
5 int snprintf (char *str, size_t size, const char *format, ...);
6
7 #include <stdarg.h>
8 int vprintf (const char *format, va_list ap);
9 int vfprintf (FILE *stream, const char *format, va_list ap);
10 int vsprintf (char *str, const char *format, va_list ap);
11 int vsnprintf (char *str, size_t size, const char *format, va_list ap);
```

4.3 DESCRIPTION

Les fonctions de la famille printf produisent des sorties en accord avec le format décrit plus bas. Les fonctions printf et vprintf écrivent leur sortie sur stdout, le flux de sortie standard. fprintf et vfprintf écrivent sur le flux stream indiqué. sprintf, snprintf, vsprintf et vsnprintf écrivent leurs sorties dans la chaîne de caractères str. Les fonctions vprintf, vfprintf, vsprintf, vsnprintf sont équivalentes aux fonctions printf, fprintf, sprintf, snprintf, respectivement, mais elles emploient un tableau va_list à la place d'un nombre variable d'arguments. Ces fonctions n'appellent pas la macro va_end, aussi la valeur de ap est-elle indéfinie après l'appel. Les applications devraient invoquer va_end(ap) elles-mêmes à la suites de ces routines. Ces huit fonctions créent leurs sorties sous le contrôle d'une chaîne de format qui indique les conversions à apporter aux arguments suivants (ou accessibles à travers les arguments de taille variable de stdarg(3)).

4.4 VALEUR RENVOYÉE

Ces fonctions renvoient le nombre de caractères imprimés, sans compter le caractère nul '\0' final dans les chaînes. Les fonctions snprintf et vsnprintf n'écrivent pas plus de size octets (y compris le '\0' final). Si la sortie a été tronquée à cause de la limite, la valeur de retour est le nombre de caractères (sans le '\0' final) qui auraient été écrits dans la chaîne s'il y avait eu suffisamment de place. Ainsi une valeur de retour size ou plus signifie que la sortie a été tronquée. (Voir aussi la section NOTES plus bas). Si une erreur de sortie s'est produite, une valeur négative est renvoyée.

4.5 CHAÎNE DE FORMAT

Le format de conversion est indiqué par une chaîne de caractères, commençant et se terminant dans son état de décalage initial. La chaîne de format est composée d'indicateurs : les caractères ordinaires (différents de %), qui sont copiés sans modification sur la sortie, et les spécifications de conversion, qui sont mises en correspondances avec les arguments suivants. Les spécifications de conversion sont introduites par le caractère %, et se terminent par un indicateur de conversion. Entre eux peuvent se trouver (dans l'ordre), zéro ou plusieurs attributs, une valeur optionnelle de largeur minimal de champ, une valeur optionnelle de précision, et un éventuel modificateur de longueur. Les arguments doivent correspondre correctement (après les promotions de types) avec les indicateurs de conversion. Par défaut les arguments sont pris dans l'ordre indiqué, où chaque '*' et chaque indicateur de conversion réclament un nouvel argument (et où l'insuffisance en arguments est une erreur. On peut aussi préciser explicitement quel argument prendre, en écrivant, à chaque conversion, '%m\$' au lieu de '%', et '*m\$' au lieu de '*'. L'entier décimal m indique la position dans la liste d'arguments, l'indexation commençant à 1. Ainsi,

```
1 printf ("\%*d", width, num);
```

et

```
1 printf ("\%2\$*1\$d", width, num);
```

sont équivalents. La seconde notation permet de répéter plusieurs fois le même argument. Le standard C99 n'autorise pas le style utilisant '\$', qui provient des Spécifications Single Unix. Si le style avec '\$' est utilisé, il faut l'employer pour toutes conversions prenant un argument, et pour tous les arguments de largeur et de précision, mais on peut le mélanger avec des formats '%%' qui ne consomment pas d'arguments. Il ne doit pas y avoir de sauts dans les numéros des arguments spécifiés avec '\$'. Par exemple si les arguments 1 et 3 sont spécifiés, l'argument 2 doit aussi être mentionné quelque part dans la chaîne de format.

Pour certaines conversions numériques, un caractère de séparation décimale (le point par défaut) est utilisé, ainsi qu'un caractère de regroupement par milliers '. Les véritables caractères dépendent de la localisation **LC_NUMERIC**. La localisation POSIX utilise '.' comme séparateur décimal, et n'a pas de caractère de regroupement. Ainsi,

```
1 printf ("\%.2f", 1234567.89);
```

s'affichera comme 1234567.89 dans la localisation POSIX, 1234567,89 en localisation **fr_FR**, et 1.234.567,89 en localisation **da_DK**.

4.6 CARACTÈRE D'ATTRIBUT

Le caractère % peut être éventuellement suivi par un ou plusieurs attributs suivants :

indique que la valeur doit être convertie en une autre forme. Pour la conversion o le premier caractère de la chaîne de sortie vaudra zéro (en ajoutant un préfixe 0 si ce n'est pas déjà un zéro). Pour les conversions x et X une valeur non nulle reçoit le préfixe '0x' (ou '0X' pour l'indicateur X). Pour les conversions a, A, e, E, f, g, et G le résultat contiendra toujours un point décimal même si aucun chiffre ne le suit (normalement, un point décimal n'est présent avec ces conversions que si des décimales le suivent). Pour les conversions g et G les zéros en tête ne sont pas éliminés, contrairement au comportement habituel. Pour les autres conversions, cet attribut n'a pas d'effet.

0 indique le remplissage avec des zéros. Pour les conversions d, i, o, u, x, X, a, A, e, E, f, F, g, et G, la valeur est complétée à gauche avec des zéros plutôt qu'avec des espaces. Si les attributs 0 et - apparaissent ensemble, l'attribut 0 est ignoré. Si une précision est fournie avec une conversion numérique (d, i, o, u, x, et X), l'attribut 0 est ignoré. Pour les autres conversions, le comportement est indéfini.

- indique que la valeur doit être justifiée sur la limite gauche du champ (par défaut elle l'est à droite). Sauf pour la conversion n, les valeurs sont complétées à droite par des espaces, plutôt qu'à gauche par des zéros ou des blancs. Un attribut - surcharge un attribut 0 si les deux sont fournis.

' (un espace) indique qu'un espace doit être laissé avant un nombre positif (ou une chaîne vide) produit par une conversion signée

+ indique que le signe doit toujours être imprimé avant un nombre produit par une conversion signée. Un attribut + surcharge un attribut 'espace' si les deux sont fournis.

Les cinq caractères d'attributs ci-dessus sont définis dans le standard C, les spécifications SUSv2 en ajoute un :

5 Premiers éléments de syntaxe

- ’ Pour les conversions décimales (**i**, **d**, **u**, **f**, **g**, **G**) indique que les chiffres d'un argument numérique doivent être groupés par millier en fonction de la localisation. Remarquez que de nombreuses versions de gcc n'accepte pas cet attribut et déclencheront un avertissement (warning). SUSv2 n'inclue pas %’F.

La GlibC 2.2 ajoute un caractère d'attribut supplémentaire.

- I Pour les conversions décimales (**i**, **d**, **u**) la sortie emploie les chiffres alternatifs de la localisation s'il y en a.

4.7 LARGEUR DE CHAMP

Un nombre optionnel ne commençant pas par un zéro, peut indiquer une largeur minimale de champ. Si la valeur convertie occupe moins de caractères que cette largeur, elle sera complétée par des espaces à gauche (ou à droite si l'attribut d'alignement à gauche a été fourni). À la place de la chaîne représentant le nombre décimal, on peut écrire '*' ou '*m\$' (m étant entier) pour indiquer que la largeur du champ est fournie dans l'argument suivant, ou dans le m-ième argument, respectivement. L'argument fournissant la largeur doit être de type **int**. Une largeur négative est considéré comme l'attribut ‘-’ vu plus haut suivi d'une largeur positive. En aucun cas une largeur trop petite ne provoque la troncature du champ. Si le résultat de la conversion est plus grand que la largeur indiquée, le champ est élargi pour contenir le résultat.

4.8 PRÉCISION

Une précision eventuelle, sous la forme d'un point (‘.’) suivi par un nombre. À la place de la chaîne représentant le nombre décimal, on peut écrire '*' ou '*m\$' (m étant entier) pour indiquer que la précision est fournie dans l'argument suivant, ou dans le m-ième argument, respectivement. L'argument fournissant la précision doit être de type **int**. Si la précision ne contient que le caractère ‘.’, ou une valeur négative, elle est considérée comme nulle. Cette précision indique un nombre minimum de chiffres à faire apparaître lors des conversions **d**, **i**, **o**, **u**, **x**, et **X**, le nombre de décimales à faire apparaître pour les conversions **a**, **A**, **e**, **E**, **f**, et **F**, le nombre maximum de chiffres significatifs pour **g** et **G**, et le nombre maximum de caractères à imprimer depuis une chaîne pour les conversions **S** et **s**.

4.9 MODIFICATEUR DE LONGUEUR

Ici, une conversion entière correspond à **d**, **i**, **o**, **u**, **x**, ou **X**.

- hh** La conversion entière suivante correspond à un **signed char** ou **unsigned char**, ou la conversion **n** suivante correspond à un argument pointeur sur un **signed char**.
- h** La conversion entière suivante correspond à un **short int** ou **unsigned short int**, ou la conversion **n** suivante correspond à un argument pointeur sur un **short int**.
- l (elle)** La conversion entière suivante correspond à un **long int** ou **unsigned long int**, ou la conversion **n** suivante correspond à un pointeur sur un **long int**, ou la conversion **c** suivante correspond à un argument **wint_t**, ou encore la conversion **s** suivante correspond à un pointeur sur un **wchar_t**.
- ll (elle-elle)** La conversion entière suivante correspond à un **long long int**, ou **unsigned long long int**, ou la conversion **n** suivante correspond à un pointeur sur un **long long int**.
- L** La conversion **a**, **A**, **e**, **E**, **f**, **F**, **g**, ou **G** suivante correspond à un argument **long double**. (C99 autorise %LF mais pas SUSv2).
- q** (‘quad’ **BSD 4.4 et Linux sous libc5 seulement, ne pas utiliser**) Il s'agit d'un synonyme pour **ll**.
- j** La conversion entière suivante correspond à un argument **intmax_t** ou **uintmax_t**.
- z** La conversion entière suivante correspond à un argument **size_t** ou **ssize_t**. (La bibliothèque libc5 de Linux proposait l'argument Z pour cela, ne pas utiliser).
- t** La conversion entière suivante correspond à un argument **ptrdiff_t**.

Les spécifications SUSv2 ne mentionnent que les modificateurs de longueur **h** (dans **hd**, **hi**, **ho**, **hx**, **hX**, **hn**), **l** (dans **ld**, **li**, **lo**, **lx**, **lx**, **ln**, **lc**, **ls**) et **L** (dans **Le**, **LE**, **Lf**, **Lg**, **LG**).

4.10 INDICATEUR DE CONVERSION

Un caractère indique le type de conversion à apporter. Les indicateurs de conversion, et leurs significations sont :

- d**, **i** L'argument **int** est convertie en un chiffre décimal signé. La précision, si elle est mentionnée, correspond au nombre minimal de chiffres qui doivent apparaître. Si la conversion fournit moins de chiffres, le résultat est rempli à gauche avec des zéros. Par défaut la précision vaut 1. Lorsque 0 est converti avec une précision valant 0, la sortie est vide.

- o, u, x, X** L'argument **unsigned int** est converti en un chiffre octal non-signé (**o**), un chiffre décimal non-signé (**u**), un chiffre hexadécimal non-signé (**x** et **X**). Les lettres **abcdef** sont utilisées pour les conversions avec **x**, les lettres **ABCDEF** sont utilisées pour les conversions avec **X**. La précision, si elle est indiquée, donne un nombre minimal de chiffres à faire apparaître. Si la valeur convertie nécessite moins de chiffres, elle est complétée à gauche avec des zéros. La précision par défaut vaut 1. Lorsque 0 est converti avec une précision valant 0, la sortie est vide.
- e , E** L'argument réel, de type **double**, est arrondi et présenté avec la notation scientifique **[−]c.ccce*(Pmcc** dans lequel se trouve un chiffre avant le point, puis un nombre de décimales égal à la précision demandée. Si la précision n'est pas indiquée, l'affichage contiendra 6 décimales. Si la précision vaut zéro, il n'y a pas de point décimal. Une conversion **E** utilise la lettre **E** (plutôt que **e**) pour introduire l'exposant. Celui-ci contient toujours au moins deux chiffres. Si la valeur affichée est nulle, son exposant est **00**.
- f, F** L'argument réel, de type double, est arrondi, et présenté avec la notation classique **[−]ccc.ccc**, où le nombre de décimales est égal à la précision réclamée. Si la précision n'est pas indiquée, l'affichage se fera avec 6 décimales. Si la précision vaut zéro, aucun point n'est affiché. Lorsque le point est affiché, il y a toujours au moins un chiffre devant. SUSv2 ne mentionne pas **F** et dit qu'il existe une chaîne de caractères représentant l'infini ou **NaN**. Le standard C99 précise '**[−] inf**' ou '**[−] infinity**' pour les infinis, et une chaîne commençant par '**nan**' pour **NaN** dans le cas d'une conversion **f**, et les chaînes '**[−]INF**' '**[−]INFINITY**' '**NAN***' pour une conversion **F**.
- g, G** L'argument réel, de type double, est converti en style **f** ou **e** (ou **E** pour la conversion **G**) La précision indique le nombre de décimales significatives. Si la précision est absente, une valeur par défaut de 6 est utilisée. Si la précision vaut 0, elle est considérée comme valant 1. La notation scientifique **e** est utilisée si l'exposant est inférieur à -4 ou supérieur ou égal à la précision demandée. Les zéros en fin de partie décimale sont supprimés. Un point decimal n'est affiché que s'il est suivi d'au moins un chiffre.
- a, A** (C99 mais pas SUSv2). Pour la conversion **a**, l'argument de type **double** est transformé en notation hexadécimale (avec les lettres **abcdef**) dans le style **[−]0xh.hhhhp*(Pmd**; Pour la conversion **A**, le préfixe **0X**, les lettres **ABCDEF** et le séparateur d'exposant **P** sont utilisés. Il y a un chiffre hexadécimal avant la virgule, et le nombre de chiffres ensuite est égal à la précision. La précision par défaut suffit pour une représentation exacte de la valeur, si une représentation exacte est possible en base 2. Sinon elle est suffisamment grande pour distinguer les valeurs de type double. Le chiffre avant le point décimal n'est pas spécifié pour les nombres non-normalisés, et il non-nul pour les nombres normalisés.
- c** S'il n'y a pas de modificateur **1**, l'argument entier, de type **int**, est converti en un **unsigned char**, et le caractère correspondant est affiché. Si un modificateur **1** est présent, l'argument de type **wint_t** (caractère large) est converti en séquence multi-octet par un appel à **wctomb**, avec un état de conversion débutant dans l'état initial. La chaîne multi-octet résultante est écrite.
- s** S'il n'y a pas de modificateur **1**, l'argument de type **const char *** est supposé être un pointeur sur un tableau de caractères (pointeur sur une chaîne). Les caractères du tableau sont écrits jusqu'au caractère NUL final, non compris. Si une précision est indiquée, seul ce nombre de caractères sont écrits. Si une précision est fournie, il n'y a pas besoin de caractère nul. Si la précision n'est pas donnée, ou si elle est supérieure à la longueur de la chaîne, le caractère NUL final est nécessaire. Si un modificateur **1** est présent, l'argument de type **const wchar_t *** est supposé être un pointeur sur un tableau de caractères larges. Les caractères larges du tableau sont convertis en une séquence de caractères multi-octets (chacun par un appel de **wctomb**, avec un état de conversion dans l'état initial avant le premier caractère large), ceci jusqu'au caractère large nul final compris. Les caractères multi-octets résultants sont écrits jusqu'à l'octet nul final (non compris). Si une précision est fournie, il n'y a pas plus d'octets écrits que la précision indiquée, mais aucun caractère multi-octet n'est écrit partiellement. Remarquez que la précision concerne le nombre d'octets écrits, et non pas le nombre de caractères larges ou de positions d'écrans. La chaîne doit contenir un caractère large nul final, sauf si une précision est indiquée, suffisamment petite pour que le nombre d'octets écrits la remplisse avant la fin de la chaîne.
- C** (dans SUSv2 mais pas dans C99) Synonyme de **lc**. Ne pas utiliser.
- S** (dans SUSv2 mais pas dans C99) Synonyme de **ls**. Ne pas utiliser.
- p** L'argument pointeur, du type **void *** est affiché en hexadécimal, comme avec **%#x** ou **%#lx**.
- n** Le nombre de caractères déjà écrits est stocké dans l'entier indiqué par l'argument pointeur de type **int ***.
Aucun argument n'est converti.
- %** Un caractère '%' est écrit. Il n'y a pas de conversion. L'indicateur complet est '**%%**'.

4.11 EXEMPLES

Pour afficher avec cinq décimales :

```

1 #include <math.h>
2 #include <stdio.h>
3 fprintf (stdout, "pi = %.5f\n", 4 * atan (1.0));

```

Pour afficher une date et une heure sous la forme ‘Sunday, July 3, 23 :15’, où jour_semaine et mois sont des pointeurs sur des chaînes :

```

1 #include <stdio.h>
2 fprintf (stdout, "%s, %s %d, %.2d:%.2d\n",
3          jour_semaine, mois, jour, heure, minute);

```

De nombreux pays utilisent un format de date différent, comme jour-mois-année. Une version internationale doit donc être capable d'afficher les arguments dans l'ordre indiqué par le format :

```

1 #include <stdio.h>
2 fprintf(stdout, format,
3         jour_semaine, mois, day, hour, min);

```

où le format dépend de la localisation et peut permuter les argument. Avec la valeur “%1\$s, %3\$d. %2\$s, %4\$d:%5\$.2d\n” On peut obtenir ‘Dimanche, 3 Juillet, 23 :15’.

Pour allouer une chaîne de taille suffisante et écrire dedans (code correct aussi bien pour GlibC 2.0 que GlibC 2.1) :

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdarg.h>
4 char * make_message(const char *fmt, ...){
5     /* Supposons que 100 octets suffisent. */
6     int n, size = 100;
7     char *p;
8     va_list ap;
9
10    if ((p = malloc (size)) == NULL)
11        return NULL;
12
13    while (1) {
14        /* Essayons avec l'espace alloué. */
15        va_start(ap, fmt);
16        n = vsnprintf (p, size, fmt, ap);
17        va_end(ap);
18
19        /* Si ça marche, renvoyer la chaîne. */
20        if (n > -1 && n < size)
21            return p;
22
23        /* Sinon réessayer avec plus de place */
24        if (n > -1) /* GlibC 2.1 */
25            size = n+1; /* ce qu'il fallait */
26        else /* GlibC 2.0 */
27            size *= 2; /* deux fois plus */
28
29        if ((p = realloc (p, size)) == NULL)
30            return NULL;
31    }
32}

```

4.12 NOTES

L'implémentation des fonctions `snprintf` et `vsnprintf` de la GlibC se conforme au standard C99, et se comporte comme décrit plus haut depuis la GlibC 2.1. Jusqu'à la GlibC 2.0.6, elles renvoyaient -1 si la sortie avait été tronquée.

4.13 CONFORMITÉ

Les fonctions `fprintf`, `printf`, `sprintf`, `vprintf`, `vfprintf`, et `vsprintf` sont conformes à ANSI X3.159-1989 (“ANSI C”) et ISO/IEC 9899 :1999 (“ISO C99”). Les fonctions `snprintf`, et `vsnprintf` sont conformes à ISO/IEC 9899 :1999.

En ce qui concerne la valeur de retour de `snprintf`, SUSv2 et C99 sont en contradiction : lorsque `snprintf` est appelée avec un argument `size=0` lors SUSv2 précise une valeur de retour indéterminée, autre que 1, alors

que C99 autorise str à être NULL dans ce cas, et réclame en valeur de retour (comme toujours) le nombre de caractères qui auraient été écrits si la chaîne de sortie avait été assez grande.

La bibliothèque libc4 de Linux connaissait les 5 attributs standards du C. Elle connaissait les modificateurs de longueur **h**, **l**, **L** et les conversions **cdeEfFgGinopsuxX**, où **F** était synonyme de **f**. De plus elle acceptait **D**, **O**, **U** comme synonymes de **ld**, **lo** et **lu**. (Ce qui causa de sérieux bogues par la suite lorsque le support de **%D** disparut). Il n'y avait pas de séparateur décimal dépendant de la localisation, pas de séparateur des milliers, par de NaN ou d'infinis, et pas de **%m\$** ni ***m\$**.

La bibliothèque libc5 de Linux connaissait les 5 attributs standards C, l'attribut ', la localisation, **%m\$** et ***m\$**. Elle connaissait les modificateurs de longueur **h**, **l**, **L**, **Z**, **q**, mais acceptait **L** et **q** pour les "long double" et les "long long integer" (ce qui est un bogue). Elle ne reconnaissait plus **FDOU**, mais ajoutait le caractère de conversion **m**, qui affiche **strerror (errno)**.

La bibliothèque GlibC 2.0 ajoute les caractères de conversion **C** et **S**.

La bibliothèque GlibC 2.1 ajoute les modificateurs de longueur **hh**, **t**, **z**, et les caractères de conversion **a**, **A**.

La bibliothèque GlibC 2.2. ajoute le caractère de conversion **F** avec la sémantique C99, et le caractère d'attribut **I**.

4.14 HISTORIQUE

Unix V7 définissait les trois routines **printf**, **fprintf**, **sprintf**, et l'attribut -, la largeur ou la précision *, le modificateur de longueur 1, et les conversions **doxfegcsu**, ainsi que **D**, **O**, **U**, **X** comme synonymes de **ld**, **lo**, **lu**, **lx**. Ceci est vrai pour BSD 2.9.1, mais BSD 2.10 avait les attributs #, + et <espace> mais ne mentionnait plus **D**, **O**, **U**, **X**. BSD 2.11 avait **vprintf**, **vfprintf**, **vsprintf**, et signalait le problème de **D**, **O**, **U**, **X**. BSD 4.3 Reno avait l'attribut **0**, les modificateurs de longueur **h** et **L**, ainsi que les conversions **n**, **p**, **E**, **G**, **X** (avec sa signification actuelle), et dénonçait **D**, **O**, **U**. BSD 4.4 introduisit les fonctions **snprintf** et **vsnprintf**, et le modificateur de longueur **q**. FreeBSD avait aussi les fonctions **asprintf** et **vasprintf**, qui allouaient un buffer assez grand pour **sprintf**. Dans la GlibC, il existe des fonctions **dprintf** et **vdprintf** qui affichent leur résultat sur un descripteur de fichier plutôt qu'un flux.

4.15 BOGUES

Comme **sprintf** et **vsprintf** ne font pas de suppositions sur la longueur des chaînes, le programme appelant doit s'assurer de ne pas déborder l'espace d'adressage. C'est souvent difficile. Notez que la longueur des chaînes peut varier avec la localisation et être difficilement prévisible. Il faut alors utiliser **snprintf** ou **vsnprintf** à la place (ou encore **asprintf** et **vasprintf**).

La libc4.[45] de Linux n'avait pas **snprintf**, mais proposait une bibliothèque **libbsd** qui contenait un **snprintf** équivalent à **sprintf**, c'est à dire qui ignorait l'argument **size**. Ainsi, l'utilisation de **snprintf** avec les anciennes **libc4** pouvait conduire à de sérieux problèmes de sécurité.

Un code tel que **printf(foo);** indique souvent un bogue, car foo peut contenir un caractère %. Si foo vient d'une saisie non sécurisée, il peut contenir %n, ce qui autorise **printf** à écrire dans la mémoire, et crée une faille de sécurité.

4.16 VOIR AUSSI

printf(1), **asprintf(3)**, **dprintf(3)**, **wcrtomb(3)**, **wprintf(3)**, **scanf(3)**, **locale(5)**

4.17 TRADUCTION

Christophe Blaess, 1996-2003.

✓ L'essentiel à retenir :

1. Comment code-t-on un saut de ligne ?
2. Comment code-t-on un \ ?
3. Quelle est la différence entre la division entière et la division réelle ? Comment se matérialise-t-elle en C ?
4. Quels sont les paramètres de la fonction **printf** ? Comment fait-on pour afficher un entier de type **int** ?

Deuxième partie

Niveau débutant

6

Types de données, Opérateurs et Expressions

1 Les types de bases

1.1 Les types de bases

Les types

- Entiers : `char`, `short`, `int` et `long / long int`
- Réels : `float`, `double` et `long double`
- Non typé / Sans type : `void`
- Les pointeurs : on les verra un peu plus tard

Les modificateurs de signe

- `signed` : l'entier est signé

```
1 signed int
2 signed short
```

- `unsigned` : l'entier est non signé

```
1 unsigned int
2 unsigned short
```

1.2 Taille et domaine de valeurs

Type	Taille en bits	Domaine
<code>unsigned char</code>	8	0..255
<code>signed char</code>	8	-128..127 ou -127..127
<code>unsigned short</code>	≥ 16	0..65535
<code>signed short, short</code>	≥ 16	-32768..32767 ou -32767..32767
<code>unsigned int</code>	≥ 16	0..65535 ou 0..4294967295
<code>signed int, int</code>	≥ 16	-32768..32767 ou -32767..32767 ou -2147483648..2147483647 ou -2147483647..2147483647 ou plus
<code>unsigned long</code>	≥ 32	0..4294967295 ou 0..18446744073709551615 ou plus
<code>signed long, long</code>	≥ 32	-2147483648..2147483647 ou -2147483647..2147483647 ou -9223372036854775808..9223372036854775807 ou -9223372036854775807..9223372036854775807 ou plus
<code>float</code>	32	précision ≥ 6 chiffres décimaux, -1E+37..1E+37 au minimum -3.4E+38..3.4E+38
<code>double</code>	64	précision ≥ 10 chiffres décimaux, -1E+37..1E+37 au minimum -1.7E+308..1.7E+308

Remarque

Les limites ou tailles peuvent changer en fonction de :

- du compilateur
- de la machine

⇒ utiliser les constantes de **limits.h** et **float.h** telles que

- **INT_MIN**,
- **INT_MAX**,
- **DBL_MAX**,
- **DBL_EPSILON**,
- **CHAR_MIN** ou
- **SCHAR_MIN**.

1.3 Cas particuliers

Variabilité du type **char**

Le type **char** peut être signé ou non signé en fonction de la configuration du compilateur

Le type caractère

Le type entier **char** est également le type caractère : ⇒ on peut calculer sur des caractères ! (cf. table ASCII et TDs)

Les chaînes de caractères

Il n'y a pas de type « chaîne de caractère » : ⇒ on utilise une convention de représentation et des tableaux
⇒ on verra ça plus loin

Les booléens

Il n'y a pas de type « booléen » : ⇒ on utilise une convention de représentation sur les entiers ! ⇒ 0 = Faux, différent de 0 = Vrai

1.4 Exemple

```
1 #include <limits.h>
2 #include <stdio.h>
3
4 int main( void )
5 {
6     printf( "Type char [min;max] = [%d;%d]\n", (int)CHAR_MIN, (int)CHAR_MAX );
7     printf( "Type signed char [min;max] = [%d;%d]\n", (int)SCHAR_MIN,
8             (int)SCHAR_MAX );
9     printf( "Type unsigned char [min;max] = [%d;%d]\n", (int)0, (int)UCHAR_MAX );
10
11    printf( "Type short [min;max] = [%d;%d]\n", (int)SHRT_MIN, (int)SHRT_MAX )
12    printf( "Type unsigned short [min;max] = [%d;%d]\n", (int)0,
13            (int)USHRT_MAX );
14
15    printf( "Type int [min;max] = [%d;%d]\n", (int)INT_MIN, (int)INT_MAX );
16    printf( "Type unsigned int [min;max] = [%u;%u]\n",
17            (unsigned int)0, (unsigned int)UINT_MAX );
18
19    printf( "Type long int [min;max] = [%ld;%ld]\n", (long int)LONG_MIN,
20            (long int)LONG_MAX );
21    printf( "Type unsigned long int [min;max] = [%lu;%lu]\n",
22            (unsigned long int)0, (unsigned long int)ULONG_MAX );
23
24    return 0;
25 }
```

2 Les variables

2.1 Les identificateurs et la déclaration de variable

Variable

Zone de stockage d'informations. Cette zone est réservée en mémoire.

Identificateur de variable

Nom donné à la zone mémoire associée à une variable.

Constituant de l'identificateur

- Suite de lettres (maj et min), chiffres et caractères soulignés
- Premier caractère : une lettre ou le caractère souligné
- Pas un mot clé !
- Taille des identificateurs C-ANSI : 31 caractères max
- **ATTENTION** : distinction minuscule/majuscule

Déclaration d'une variable

Obligatoire !

```
1 Type Identificateur1 , Identificateur2 , ... , IdentificateurN ;
```

```
1 int n, __mon_entier_pour_calculer ;
2 float x1, x2, mon_reel_a_moi ;
```

2.2 Portée d'une variable

Portée d'une variable

Zones d'un programme où la variable et son contenu sont accessibles

Trois catégories de variables

- Variables locales
- Variables paramètres de fonction
- Variables globales

2.3 Portée d'une variable locale

Variable locale

- Toute variable déclarée au sein d'une fonction
- Toute variable déclarée au sein d'un bloc d'instruction { }

Portée

En dehors du bloc où une variable locale est déclarée, elle est inconnue.

Création / destruction

A l'exécution d'un bloc d'instruction / d'une fonction :

- Entrée dans le bloc : création des variables
- Sortie du bloc : destruction des variables

Ce mécanisme utilise la pile d'appel : les variables sont stockées dans la pile d'appel.

Propriétés

- Le contenu n'est pas conservé entre 2 exécutions successives
- Aucun effet de bord et aucune modification non explicite d'une variable locale ne sont possibles

```
1 void f(void)
2 {
3     int x;
4     // utilisation de x
5     {
6         double y, z;
7         // utilisation de x, y et z
8     }
9     // utilisation de x
10}
```

6 Types de données, Opérateurs et Expressions

Lisibilité des fichiers sources

La norme C99 autorise la déclaration de variable à n'importe quel endroit de la fonction. **Toute variable doit être déclarée en début de bloc**

2.4 Portée des paramètres d'une fonction

Paramètre de fonction

Toute variable qui est paramètre d'une fonction.

Portée

Même comportement que les variables locales :

- Connus dans le corps de la fonction **uniquement** !
- Utilisation de la pile : cf. mécanisme d'appel d'une fonction

Remarque

La tête de la fonction définie :

- Le type des arguments
- Les identificateurs des variables paramètres

```
1 void fct1( int n, double x)
2 {
3     // utilisation de n et x
4 }
```

2.5 Remarques sur les prototypes de fonctions

Prototype

Signature / entête de la fonction + ;

```
1 void fct1( int n, double x);
```

Rôle de vérification

Permet de vérifier adéquation paramètres formels/arguments transmis

```
1 fct1 (3.2 , 1) ;
```

Les arguments transmis sont du type (double, int) \Rightarrow le compilateur va devoir faire des conversions

Rôle de déclaration

Permet de rendre une fonction connue avant qu'elle soit implémentée

```
1 void f( int x); // déclaration de f pour l'utiliser dans g
2
3 void g( int y )
4 {
5     // traitement utilisant f
6 }
7
8 void f( void )
9 {
10    // traitement utilisant g
11 }
```

2.6 Portée des variables globales

Variable globale

Variables déclarées hors de toute fonction

Portée

Visibilité étendue à toutes les fonctions déclarées après la variable

```

1 void g(void)
2 {
3     // on ne peut pas utiliser x ici !!
4 }
5
6 int x; // c'est une variable globale
7
8 void f(void)
9 {
10    // on peut utiliser x !!
11 }
```

Propriétés

La zone mémoire associée à une variable globale :

- Crée automatiquement au début du programme
- Existe du début à la fin du programme en mémoire

2.7 Exemples

2.7.1 Exemple 1

```

1 #include <stdio.h>
2
3 void afficher(void);
4
5 int main(void)
6 {
7     int mavariable = 1;
8
9     printf("(1) Ma variable vaut %d\n", mavariable);
10    afficher(void);
11
12    return 0;
13 }
14
15 void afficher(void)
16 {
17     printf("(2) Ma variable vaut %d\n", mavariable);
18 }
```

2.7.2 Exemple 2

```

1 #include <stdio.h>
2
3 void afficher(void);
4
5 int mavariable = 1;
6
7 int main(void)
8 {
9     printf("(1) Ma variable vaut %d\n", mavariable);
10    afficher(void);
11
12    return 0;
13 }
14
15 void afficher(void)
16 {
17     printf("(2) Ma variable vaut %d\n", mavariable);
18 }
```

2.8 Initialisation de variables

Cas des variables numériques entières ou réelles :

6 Types de données, Opérateurs et Expressions

Initialisation à la déclaration

```
1 Type Identificateur = Valeur;
```

- Les réels se notent avec un point (.) : **102.5**, **102.**, **.52**

```
1 int mavariable = 10;
2 double monautrevariable = 3.1415;
```

Déclaration puis affectation

```
1 int mavariable;
2 mavariable = 10;
```

2.9 Les modificateurs

2.9.1 Les modificateurs : stockage et accès

Deux types de modificateurs

- d'accès : **const** et **volatile**
- de stockage : **auto**, **register**, **static** et **extern**

Utilisation

- A la déclaration de la variable

```
1 Modificateur Type Identificateur;
```

- Exemple :

```
1 volatile int mavar1;
2 register long mavar2;
```

2.9.2 Modificateur d'accès **const**

Propriétés

- Déclare une variable constante
- Lorsque que la variable a une valeur, elle ne peut plus en changer

Formes

- Variables locales et globales : \Rightarrow initialisation obligatoire à la déclaration !

```
1 const int mavariable = 10;
2 const double monautrevariable = 3.1415;
```

- Variables paramètres de fonction : \Rightarrow pas d'initialisation, pas le droit de modifier la valeur !

```
1 void f( const int x )
2 {
3     x = 43;    // on a pas le droit de modifier x !
4 }
```

2.9.3 Modificateur d'accès **volatile**

Propriétés

- Indique que le contenu d'une variable peut être modifié par une cause extérieure au programme telle qu'une interruption
- Agit sur le code machine généré par le compilateur
- Aucun effet visible du point de vue du programmeur

Forme

```
1 volatile int mavariable;
```

2.9.4 Modificateur d'accès auto

Propriété

Etat par défaut d'une variable

Forme

```
1 auto int mavariable;
```

Remarque

Ne sert à rien en pratique

2.9.5 Modificateur d'accès register

Propriétés

- ANSI-C : la variable est stocké dans un registre du CPU plutôt qu'en RAM lorsque c'est possible
- Applicable aux variables locales de type **int** ou **pointeur** (indice de boucle, ...)
- Permet en théorie d'accélérer l'exécution du programme

Forme

```
1 register int mavariable;
```

Remarques

- De moins en moins respecté par les compilateurs
- Les compilateurs essayent automatiquement de mettre un maximum de variables dans les registres CPU pour optimiser les performances

2.9.6 Modificateur d'accès static

Deux cas d'utilisation

Conserver la valeur d'une variable locale entre deux appels d'une fonction

- Variable locale statique

```
1 static Type Identificateur = Valeur;
```

- L'initialisation est obligatoire et est réalisée une seule fois, lors du premier appel de la fonction

```
1 void f(void)
2 {
3     static int compteur = 0;
4     compteur = compteur + 1;
5     printf("Valeur du compteur : %d\n", compteur);
6 }
```

Réduire la portée d'une variable globale

- Réduction de la visibilité de la variable au fichier où elle est déclarée

2.9.7 Modificateur d'accès extern

Propriétés

- Utilisé avec plusieurs fichiers sources et des variables globales communes
- La variable est créée une seule fois mais elle est utilisée dans plusieurs fichiers sources

Forme

```
1 extern int mavariable;
```

Principes

```

1  /* module1.c */
2  int x, y;
3
4  int main(void)
5  {
6  }
```

```

/* modules2.c */
extern int x;

void fct1(void)
{
}
```

```

/* module3.c */
extern int x, y;

void fct2(void)
{
}
```

- x est visible dans les trois modules
- y n'est visible que dans les modules module1.c et module3.c
- Visibilité implique que la variable a une adresse unique en RAM
- **extern** indique que la variable est déclarée au sein d'un autre fichier \Rightarrow pas de réservation mémoire

2.9.8 Résumé et remarque

Résumé

Classe de stockage	Mot clé	Définition	Durée de vie	Portée
Automatique	auto ou rien	Au sein d'une fonction	Temporaire	Locale
Statique	static	Au sein d'une fonction	Permanent	Locale
Externe	extern	En dehors d'une fonction	Permanent	Globale à tous les fichiers
Externe statique	static	En dehors d'une fonction	Permanent	Globale au fichier de déclaration

Remarque

Il faut éviter autant que possible les variables globales.

2.9.9 Exemple

```

1 #include <stdio.h>
2
3 void mafonction( int compteur );
4
5 int main(void)
6 {
7     int compt;
8
9     for( compt = 0; compt < 20; compt++ )
10    {
11        printf("Itération de la boucle #%d\n", compt);
12        mafonction( compt );
13    }
14    return 0;
15 }
16 void mafonction( int compteur )
17 {
18     static int nombreappel = 0;
19     int uncompteur = 0;
20
21     nombreappel = nombreappel + 1;
22     uncompteur = uncompteur + 1;
23     printf( "(compteur ,nombreappel ,uncompteur)=(%d,%d,%d)\n" , compteur ,
24             nombreappel ,uncompteur );
25 }
```

2.10 Les constantes

2.10.1 Les constantes : quatre catégories

Les constantes littérales

- Valeur numérique directement introduite dans le code :
- ex : 100, 100., 27.4
- Le mécanisme de conversion de types s'applique (cf. plus loin)

Les constantes symboliques

- Représentées par un nom
- ```
1 #define NOM Valeur
```
- Par convention, NOM est en majuscule
  - Par convention, la définition de constantes symboliques se fait en début de fichier source

## Les variables constantes

- Utilisation du modificateur **const**
- Attention aux règles de portée : locale, globale
- Utilisation de pointeurs possible

```
1 const double PI = 3.1415;
```

## Les énumérations

- On verra ça plus tard

### 2.10.2 Typage et codage des constantes numériques

#### Typage des constantes numériques : suffixe de typage

| Type                     | Format numérique | Suffixe   | Exemple |
|--------------------------|------------------|-----------|---------|
| <b>int</b>               | entier           | aucun     | 31415   |
| <b>unsigned int</b>      | entier           | <b>U</b>  | 31415U  |
| <b>long int</b>          | entier           | <b>L</b>  | 31415L  |
| <b>unsigned long int</b> | entier           | <b>UL</b> | 31415UL |
| <b>float</b>             | décimal          | <b>F</b>  | 3.1415F |
| <b>double</b>            | décimal          | aucun     | 3.1415  |
| <b>long double</b>       | décimal          | <b>L</b>  | 3.1415L |

#### Les constantes entières octales et hexadécimales

- **0nnn** : nnn est le nombre en octal (base 8, 0-7)
- **0xnnn** : nnn est le nombre en hexadécimal (base 16, 0-9,A-F,a-f)

### 2.10.3 Exemple

```
1 #include <stdio.h>
2
3 #define MACONSTANTE1 32L
4
5 const int maconstante2 = 64;
6
7 int main(void)
8 {
9 int total = MACONSTANTE1 + maconstante2 + 16;
10 printf("La somme de %ld + %d + %d est %d\n", MACONSTANTE1,
11 maconstante2 , 16, total);
12 return 0;
13 }
```

#### Remarque

- MACONSTANTE1 est de type **long int** donc **%ld**

## 3 Les opérateurs

### 3.1 Généralités

#### Définitions

- lvalue : partie gauche d'une opération = variable
- rvalue : partie droite d'une opération = expression

### Catégories d'opérateurs

- Opérateurs d'affectation et d'affectation élargie
- Opérateurs d'arithmétiques
- Opérateurs de logiques sur bits
- Opérateurs de décalage
- Opérateurs relationnels et logiques
- Opérateurs `sizeof`, ?, ,
- Opérateurs de changement de type

### 3.2 L'opérateur d'affectation

---

#### Syntaxe

```
1 lvalue = rvalue;
```

#### Mécanisme de conversion

Le mécanisme de conversion de type s'applique :  $\Rightarrow$  le type de rvalue est converti dans le type de lvalue

#### Si le type de lvalue est plus large que celui de rvalue

Conversion automatique sans perte d'information :

1. de signé vers signé
2. de non signé vers non signé

`char < short <= int <= long < float < double <= long double`  
—————> Conversion sans perte

#### Si le type de lvalue est plus petit que celui de rvalue

| <              |                        |                                                                                                                             |
|----------------|------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| <b>lvalue</b>  | <b>rvalue</b>          | <b>Commentaires</b>                                                                                                         |
| type signé     | même type non signé    | Passage en représentation signée<br>Possibilité de dépassement de capacité<br>Possibilité de changement de signe            |
| type non signé | même type signé        | Passage en représentation non signée<br>Possibilité de dépassement de capacité<br>Possibilité de changement de signe        |
| type entier    | type entier plus grand | Conservation des octets de poids faible<br>Possibilité de troncature de la rvalue                                           |
| type réel      | type réel plus grand   | Perte de précision de la partie décimale<br>Possibilité de troncature de la partie entière                                  |
| type entier    | type réel              | Perte de la partie décimale<br>Possibilité d'arrondi de la partie entière<br>Possibilité de troncature de la partie entière |

### 3.3 Opérateurs arithmétiques

---

#### Les opérateurs

| <b>Opérateur</b>   | <b>Symbol</b> | <b>Description</b>                                                  | <b>Exemple</b> |
|--------------------|---------------|---------------------------------------------------------------------|----------------|
| Addition           | +             | Addition de deux opérandes                                          | $x+y$          |
| Soustraction       | -             | Soustraction de la seconde opérande à celle de la première          | $x-y$          |
| Multiplication     | *             | Multiplication de deux opérandes                                    | $x*y$          |
| Division           | /             | Division de la première opérande par la seconde                     | $x/y$          |
| Modulo             | %             | Reste de la division entière de la première opérande par la seconde | $x\%y$         |
| Inversion de signe | -             | Inversion de signe                                                  | $-x$           |

### L'opérateur modulo

**x % y** : les deux opérandes doivent être entières

### L'opérateur de division

**x / y** : le résultat dépend du type des opérandes

```
1 float x = 1/3; // x = 0.0
2 float x = 1./3; // x = 0.333333333
3 float x = 1/3.; // x = 0.333333333
```

Deux opérandes entières  $\Rightarrow$  division entière

## 3.4 Pré et post incrémentation/décrémentation

### Contexte d'application

S'applique à des variables réelles ou entières

### Syntaxes

Pré-incrémentation

```
1 ++Variable
```

Post-incrémentation

```
1 Variable++
```

Pré-décrémentation

```
1 --Variable
```

Post-décrémentation

```
1 Variable--
```

### Exemple avec $x=2$

| Exemple  | Équivalence   | Valeur de x après | Valeur de y après |
|----------|---------------|-------------------|-------------------|
| $x++;$   | $x=x+1;$      | 3                 |                   |
| $++x;$   | $x=x+1;$      | 3                 |                   |
| $y=x++;$ | $y=x; x=x+1;$ | 3                 | 2                 |
| $y=++x;$ | $x=x+1; y=x;$ | 3                 | 3                 |

## 3.5 Opérateurs logiques sur bits

### Contexte d'application

Opérandes de types entier uniquement

### Table de vérité sur les bits a et b

|   |   | ou  | et  | ou exclusif | négation |
|---|---|-----|-----|-------------|----------|
| a | b | a b | a&b | a^b         | $\sim a$ |
| 0 | 0 | 0   | 0   | 0           | 1        |
| 0 | 1 | 1   | 0   | 1           | 1        |
| 1 | 0 | 1   | 0   | 1           | 0        |
| 1 | 1 | 1   | 1   | 0           | 0        |

### Mode d'emploi

- Sur des entiers : on applique sur chacun des bits correspondant des entiers
- Opérateur **|** : forcer un bit à 1
- Opérateur **&** : forcer un bit à 0

## 6 Types de données, Opérateurs et Expressions

### 3.6 Opérateurs de décalage

#### Contexte d'application

Opérandes de types entier uniquement

#### Syntaxe

- n décalages des bits vers la gauche

```
1 x << n
```

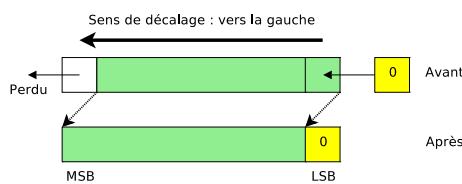
- n décalages des bits vers la droite

```
1 x >> n
```

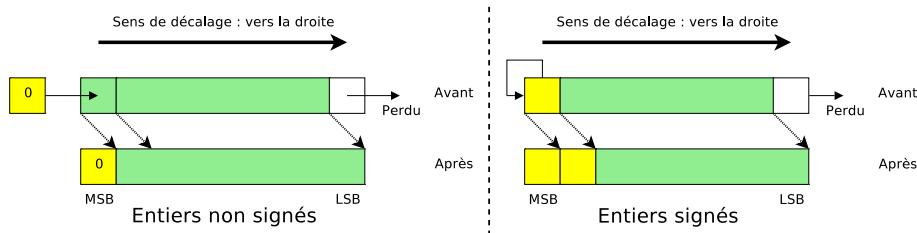
#### Remarques

- Décalage à gauche  $\Leftrightarrow$  multiplication par 2
- Décalage à droite  $\Leftrightarrow$  division par 2
- Résultat exacte s'il n'y a pas de débordement avec des entiers non signés (ou positifs)

#### Décalage à gauche sur nombre signé et non signé

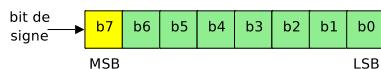


#### Décalage à droite sur nombre signé et non signé



#### Remarque sur les compléments à deux et les nombres négatifs

- si  $MSB=0$  alors  $n \geq 0$  et  $n = \sum_{i=0}^6 b_i 2^i$
- si  $MSB=1$  alors  $n < 0$  et  $n = -2^7 + \sum_{i=0}^6 b_i 2^i$



### 3.7 Opérateurs relationnels et logiques

#### En C, il n'y a pas de type booléen $\Rightarrow$ convention

- Vrai : toute valeur non nulle
- Faux : toute valeur nulle

#### Valeur de retour des opérateurs logiques et relationnels

- 1 si Vrai
- 0 si Faux

## Les opérateurs de comparaison

| Opérateur         | Symbol             | Description                      | Exemple              |
|-------------------|--------------------|----------------------------------|----------------------|
| Egalité           | <code>==</code>    | Test d'égalité                   | <code>x==y</code>    |
| Inférieur         | <code>&lt;</code>  | Test d'infériorité               | <code>x&lt;y</code>  |
| Inférieur ou égal | <code>&lt;=</code> | Test d'égalité ou d'infériorité  | <code>x&lt;=y</code> |
| Supérieur         | <code>&gt;</code>  | Test de supériorité              | <code>x&gt;y</code>  |
| Supérieur ou égal | <code>&gt;=</code> | Test d'égalité ou de supériorité | <code>x&gt;=y</code> |
| Different         | <code>!=</code>    | Test de différence               | <code>x!=y</code>    |

## Remarques

- Ne pas confondre l'affectation `=` et la comparaison `==` !
- Il ne faut pas utiliser `==` ou `!=` avec deux nombres réels. En remplacement, on utilise `fabs(x-y)<epsilon` ou `fabs(x-y)>epsilon`.

## Les opérateurs relationnels/logiques

| Opérateur | Symbol                  | Description                             | Exemple                             |
|-----------|-------------------------|-----------------------------------------|-------------------------------------|
| et        | <code>&amp;&amp;</code> | Les deux expressions sont-elles vraies  | <code>expr1 &amp;&amp; expr2</code> |
| ou        | <code>  </code>         | Une des deux expressions est-elle vraie | <code>expr1    expr2</code>         |
| non       | <code>!</code>          | L'expression est-elle fausse            | <code>!expr</code>                  |

## Evaluation des opérateurs logiques en ANSI-C

- `expr1 || expr2` : expr2 n'est évaluée que si expr1 est fausse
- `expr1 && expr2` : expr2 n'est évaluée que si expr1 est vraie

## 3.8 Opérateurs d'affectation élargie

### Syntaxe

```
1 lvalue Opérateur= rvalue;
```

### Équivalence

```
1 lvalue = lvalue Opérateur rvalue;
```

## Les opérateurs

| Affectation et ... | Symbol                 | Équivalence               | Exemple                  |
|--------------------|------------------------|---------------------------|--------------------------|
| Addition           | <code>+=</code>        | <code>x=x+y</code>        | <code>x+=y</code>        |
| Soustraction       | <code>-=</code>        | <code>x=x-y</code>        | <code>x-=y</code>        |
| Multiplication     | <code>*=</code>        | <code>x=x*y</code>        | <code>x*=y</code>        |
| Division           | <code>/=</code>        | <code>x=x/y</code>        | <code>x/=y</code>        |
| Modulo             | <code>%=</code>        | <code>x=x%y</code>        | <code>x%-=y</code>       |
| Décalage à droite  | <code>&gt;&gt;=</code> | <code>x=x&gt;&gt;y</code> | <code>x&gt;&gt;=y</code> |
| Décalage à gauche  | <code>&lt;&lt;=</code> | <code>x=x&lt;&lt;y</code> | <code>x&lt;&lt;=y</code> |
| Complémentation    | <code>^=</code>        | <code>x=x^y</code>        | <code>x^-=y</code>       |
| Et bit à bit       | <code>&amp;=</code>    | <code>x=x&amp;y</code>    | <code>x&amp;-=y</code>   |
| Ou bit à bit       | <code> =</code>        | <code>x=x y</code>        | <code>x =y</code>        |

## 6 Types de données, Opérateurs et Expressions

### 3.9 Opérateur sizeof

#### Principe

**sizeof (Type)** : fournit la taille en octet d'un type de données  $\Rightarrow$  permet de réaliser du code portable

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5 int tailleentier = sizeof(unsigned long int);
6 printf("Un entier long non signé occupe %d octets en mémoire\n",
7 tailleentier);
8 return 0;
9 }
```

### 3.10 Opérateur ?

#### Principe

Implémente un « Si Alors Sinon » sous forme d'expression

#### Syntaxe

```
1 condition ? rvalue1 : rvalue2
```

#### Exemple

```
1 int valeurabsolue;
2 valeurabsolue = (y>0)?y:-y;
```

#### Équivalence

```
1 int valeurabsolue;
2 if (y>0)
3 valeurabsolue = y;
4 else
5 valeurabsolue = -y;
```

#### Avantages et inconvénients

- Avantage : code plus compact
- Inconvénient : lisibilité discutable  $\Rightarrow$  ne pas en abuser

### 3.11 Opérateur séquentiel ,

#### Principe

Permet de former des expressions composées d'expressions évaluées séquentiellement de gauche à droite

#### Syntaxe

```
1 expr1 , expr2 , expr3;
```

#### Équivalence

```
1 expr1 ; expr2 ; expr3 ;
```

#### Intérêt

cf. boucle **for**

### 3.12 Pointeurs et opérateurs associés

#### Pointeur

Variable contenant l'adresse d'une donnée en RAM

## Variable pointeur

Variable contenant un pointeur sur une donnée typée

### Syntaxe

```
1 Type * Identificateur;
```

```
1 char * p;
2 float * f;
3 int * x, y;
```

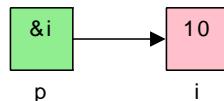
### Attention

La notation pointeur ne s'applique qu'à la variable adjacente !

### Opérateurs spécifiques

- **&** : adresse de l'opérande
- **\*** : valeur de l'opérande accessible par le pointeur (indirection)

```
1 int * p, i, j;
2 i = 10;
3 p = &i;
4 j = *p;
```



### Pointeurs de pointeurs

- Pointeur sur une variable contenant un pointeur :

```
1 TypePointé ** Identificateur;
```

- Niveau 3 :

```
1 TypePointé *** Identificateur;
```

## 3.13 Les conversions

### 3.13.1 Conversion implicite de types au sein d'une expression

#### Conversion implicite

- Conversion/promotion de type vers le type le plus grand
- Pas de perte d'information
- Conversion implicite par paires
- Si les opérandes sont de même type : pas de conversion

```
1 int main(void)
2 {
3 int i = 4;
4 float f;
5
6 f = 2.0 / i;
7 f = 2 / i;
8 return 0;
9 }
```

### 3.13.2 Conversion explicite de type : cast

#### Objectif

Forcer le compilateur à faire une conversion de type

## 6 Types de données, Opérateurs et Expressions

---

### Syntaxe

```
1 (Type) opérande
```

```
1 int main(void)
2 {
3 int i = 4;
4 float f;
5
6 f = 2 / (float) i;
7 return 0;
8 }
```

### Attention

Ne pas hésiter à parenthèser en cas de doute sur les priorités

### 3.14 Priorités des opérateurs

---

| Priorité (1=max) | Opérateurs                                                                |
|------------------|---------------------------------------------------------------------------|
| 1                | () [] ->                                                                  |
| 2                | ! ~ ++ -- *(indirection)&(adresse de)(type)<br>sizeof +(unaire) -(unaire) |
| 3                | *(multiplication) / %                                                     |
| 4                | + -                                                                       |
| 5                | << >>                                                                     |
| 6                | < <= > >=                                                                 |
| 7                | == !=                                                                     |
| 8                | & (et bit à bit)                                                          |
| 9                | ^                                                                         |
| 10               |                                                                           |
| 11               | &&                                                                        |
| 12               |                                                                           |
| 13               | ?                                                                         |
| 14               | = += -= *= /= %= &= ^=  = <<= >>=                                         |
| 15               | ,                                                                         |

### Attention

En cas de doutes : ajoutez des parenthèses !

### 3.15 Exemple de pré/post incrémentation/décrémentation

---

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5 int i = 5;
6 int j = 10;
7 int k = 20;
8 int l = 0;
9
10 printf("i=%d j=%d k=%d l=%d\n", i, j, k, l);
11
12 l = 3 * i++ * j-- + k++;
13 printf("i=%d j=%d k=%d l=%d\n", i, j, k, l);
14
15 l += -i++ + ++j;
16 printf("i=%d j=%d k=%d l=%d\n", i, j, k, l);
17
18 l = ++i - j;
19 printf("i=%d j=%d k=%d l=%d\n", i, j, k, l);
20
21 return 0;
22 }
```

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5 int i = 5;
6 int j = 10;
7 int k = 20;
8 int l = 0;
9
10 printf("i=%d j=%d k=%d l=%d\n" , i , j , k , l); // (i,j,k,l)=(5,10,20,0)
11
12 l = 3 * i++ * j-- + k++;
13 printf("i=%d j=%d k=%d l=%d\n" , i , j , k , l);
14
15 l += -i++ + ++j;
16 printf("i=%d j=%d k=%d l=%d\n" , i , j , k , l);
17
18 l = ++i - j;
19 printf("i=%d j=%d k=%d l=%d\n" , i , j , k , l);
20
21 return 0;
22 }
```

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5 int i = 5;
6 int j = 10;
7 int k = 20;
8 int l = 0;
9
10 printf("i=%d j=%d k=%d l=%d\n" , i , j , k , l); // (i,j,k,l)=(5,10,20,0)
11
12 l = 3 * i++ * j-- + k++;
13 printf("i=%d j=%d k=%d l=%d\n" , i , j , k , l); // (i,j,k,l)=(6,9,21,3*5*10+20=170)
14
15 l += -i++ + ++j;
16 printf("i=%d j=%d k=%d l=%d\n" , i , j , k , l);
17
18 l = ++i - j;
19 printf("i=%d j=%d k=%d l=%d\n" , i , j , k , l);
20
21 return 0;
22 }
```

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5 int i = 5;
6 int j = 10;
7 int k = 20;
8 int l = 0;
9
10 printf("i=%d j=%d k=%d l=%d\n" , i , j , k , l); // (i,j,k,l)=(5,10,20,0)
11
12 l = 3 * i++ * j-- + k++;
13 printf("i=%d j=%d k=%d l=%d\n" , i , j , k , l); // (i,j,k,l)=(6,9,21,3*5*10+20=170)
14
15 l += -i++ + ++j;
16 printf("i=%d j=%d k=%d l=%d\n" , i , j , k , l); // (i,j,k,l)=(7,10,21,170 + (-6+10)=174)
17
18 l = ++i - j;
19 printf("i=%d j=%d k=%d l=%d\n" , i , j , k , l);
20
21 return 0;
22 }
```

```

1 #include <stdio.h>
2
3 int main(void)
4 {
```

```
5 int i = 5;
6 int j = 10;
7 int k = 20;
8 int l = 0;
9
10 printf("i=%d j=%d k=%d l=%d\n", i, j, k, l); // (i,j,k,l)=(5,10,20,0)
11
12 l = 3 * i++ * j-- + k++;
13 printf("i=%d j=%d k=%d l=%d\n", i, j, k, l); // (i,j,k,l)=(6,9,21,3*5*10+20=170)
14
15 l += -i++ + ++j;
16 printf("i=%d j=%d k=%d l=%d\n", i, j, k, l); // (i,j,k,l)=(7,10,21,170 + (-6+10)=174)
17 l = ++i - j;
18 printf("i=%d j=%d k=%d l=%d\n", i, j, k, l); // (i,j,k,l)=(8,10,21,8-10=-2)
19
20 return 0;
21
22 }
```

## 4 Les tableaux, les chaînes et les structures

---

### 4.1 Les tableaux 1D

---

#### 4.1.1 Généralités sur les tableaux

---

Pour l'instant, on se limite

- aux tableaux numériques
- aux tableaux 1D

#### Tableau

- Ensemble d'emplacements mémoires contigus
- Portent tous le même nom
- Contiennent tous le même type de données

#### Elément d'un tableau

Un emplacement spécifique d'un tableau

#### 4.1.2 Tableaux à une dimension

---

#### Syntaxe

```
1 TypeDesElements Identificateur [NombreDElements];
```

- Type élément : le type de **tous** les éléments du tableau
- Identificateur : le nom de la variable représentant le tableau
- Nombre éléments : le nombre d'éléments du tableau : **C'est une constante (littérale/symbolique) de type entier**

#### Taille réelle d'un tableau en mémoire

```
1 NombreDElements * sizeof(TypeDesElements)
2 sizeof(NomTableau)
```

Sur certain système/compilateur : il peut y avoir des limites de taille  $\Rightarrow$  allocation dynamique

#### Accès à un élément du tableau

- Syntaxe :

```
1 Identificateur [IndiceDeLElement]
```

- IndiceDeElement : peut être une expression qui renvoie un entier
- Indice du premier élément : 0
- Indice du dernier élément : NombreDElements - 1
- Il n'y a pas de vérification des indices  $\Rightarrow$  **Il est possible de sortir de la zone mémoire réservée pour le tableau !!**

## Initialisation des éléments d'un tableau par affectation

```
1 tab [2]=10;
```

## Initialisation lors de la déclaration (en utilisant des constantes)

```
1 int tab [4] = {2,4,6,8}; // contenu = 2 4 6 8
2 int tab [4] = {2,4}; // contenu = 2 4 ???
3 int tab [4] = { ,4, ,8}; // contenu = ?? 4 ?? 8
4 int tab [] = {2,4,6,8}; // contenu = 2 4 6 8 et dimension = 4
```

Lorsque les éléments sont connus, la dimension du tableau n'est pas nécessaire !

## 4.2 Les chaînes de caractères

### 4.2.1 Convention

#### Attention

Ca n'existe pas en C : on utilise une convention de représentation.

#### Convention

- C'est un tableau 1D contenant des **char**
- La fin de la chaîne est le caractère '**\0**'
- On parle de chaînes de caractères AZT (à zéro terminal)

### 4.2.2 Déclaration

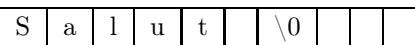
#### Déclaration par une constante chaîne de caractères

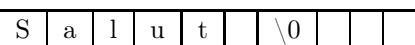
```
1 printf("Salut !");
2 char * s = "Salut !";
```

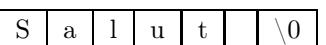
Remarque :

- La chaîne de caractères ne doit/peut pas être modifiée
- Dans le pire des cas : risque d'effets de bord
- Dans le meilleur des cas : violation d'accès mémoire et plantage du programme

#### Déclaration par des variables

1.    
`1 char s[10] = "Salut !";`

2.    
`1 char s[10] = { 'S', 'a', 'l', 'u', 't', '\0', ' ', '!', '\0' };`

3.    
`1 char s[] = "Salut !";`

Remarques

1&2 Taille du tableau  $\geq$  nombre de caractères + 1

3 Taille du tableau = nombre de caractères + 1

- La chaîne peut être modifiée

### 4.2.3 Caractères et chaînes de caractères

#### Variable de type char (ex : char c = 'a';)

1. c contient la caractère a
2. c contient le code ASCII de a c'est-à-dire 97

### Remarque

Pour manipuler des caractères ASCII > 128 : **unsigned char**

### Tableau de chaînes de caractères

- **char \* NomTableau[NombreElements];**
- char \* t [4] = { "Lundi", "Mardi", "Mercredi" };**

## 4.3 Les structures

---

### 4.3.1 Définitions, déclarations et utilisation

---

#### Définition

- type de données,
- contenant une ou plusieurs variables (champs)
- regroupées au sein d'une même entité
- pouvant être de types différents

#### Déclaration du nouveau type de données

```
1 struct <Nom de structure> {
2 Type1 NomChamps1;
3 Type2 NomChamps2;
4 ...
5 TypeN NomChampsN;
6 };
```

#### Utilisation du nouveau type de données

```
1 struct NomDeStructure NomVar1, NomVar2;
```

#### Syntaxe pour la déclaration et l'utilisation de structures anonymes

```
1 struct {
2 Type1 NomChamps1;
3 Type2 NomChamps2;
4 ...
5 TypeN NomChampsN;
6 } NomVar1, NomVar2;
7
8 struct NomDeStructure {
9 Type1 NomChamps1;
10 Type2 NomChamps2;
11 ...
12 TypeN NomChampsN;
13 } NomVar1, NomVar2;
```

#### Accès à un champ

```
1 MaVariableDeStructure .MonChamps
```

### 4.3.2 Occupation mémoire

---

#### Occupation mémoire d'une structure

- **sizeof(struct MaStructure)**
- **sizeof(MaVariableDeStructure)**
- $\geq \sum_{\text{champ} \in \text{MaStructure}} \text{sizeof(champs)}$
- Le compilateur peut introduire des octets non utilisés pour améliorer les performances (cf. padding plus tard)

### 4.3.3 Initialisation

#### Initialisation par affectation

```

1 struct coord3D {
2 double x;
3 double y;
4 double z;
5 };
6 struct coord3D point1;
7 point1.x = 10;

```

#### Initialisation à la déclaration

```

1 struct coord3D {
2 double x;
3 double y;
4 double z;
5 };
6 struct coord3D point1 = {10, 0, -1};

```

#### Remarques

- Structures imbriquées : un champ peut être une structure !
- Structures contenant un tableau : un champ peut être un tableau !
- Structures contenant un pointeur : un champ peut être un pointeur !

```

1 struct Personne
2 {
3 char nom[30];
4 char prenom[30];
5 struct Adresse adresse;
6 struct DetailsSupplementaire * detailsSupplementaire;
7 }

```

### 4.3.4 Tableau de structure

#### Déclaration

```

1 struct NomStructure NomTableau [NombreElements];

```

#### Utilisation

```

1 NomTableau [Indice]. Champs

```

#### Copie d'éléments du tableau

```

1 NomTableau [Indice1] = NomTableau2 [Indice2];

```

### 4.3.5 Pointeurs sur une structure

#### Déclaration

```

1 struct NomStructure * NomVariable;

```

#### Accès aux champs

```

1 NomVariable->NomChamps
2 (* NomVariable). NomChamps

```

### 4.3.6 Structures et fonctions

#### Passage par valeur

```
1 void fct(struct MaStructure ms)
```

Le contenu des champs est copié

#### Passage par adresse

```
1 void fct struct MaStructure * pms)
```

cf. plus loin

### 4.3.7 Exemple

```
1 #include <stdio.h>
2
3 struct Agenda {
4 int jour;
5 int mois;
6 int annee;
7 int heure;
8 int minute;
9 char detail[50];
10 }
11 void afficher(struct Agenda rdv)
12 {
13 printf("Rendez-vous le %d/%d/%d à %d:%d pour \"%s\"\n",
14 rdv.jour, rdv.mois, rdv.annee, rdv.heure, rdv.minute,
15 rdv.detail);
16 }
17 int main(void)
18 {
19 struct Agenda rdv1 = { 07, 09, 2006, 9, 0, "Réunion de rentrée" };
20 struct Agenda rdv2 = { 08, 09, 2006, 14, 0, "Premier cours de C" };
21 afficher(rdv1);
22 afficher(rdv2);
23 return 0;
24 }
```

#### ✓ L'essentiel à retenir :

1. Quels sont les principaux types de données entiers et réels ?
2. Quels sont les tailles minimales de chacun des types de base ?
3. Quels sont les contraintes sur un identificateur ?
4. Qu'est-ce que la portée d'une variable ? Quelles sont les portées des variables ?
5. Qu'est-ce qu'un modificateur ? Quelles sont les différences entre les modificateurs ?
6. Comment définit-on une constante ?
7. Quels sont les différents opérateurs ?
8. Quels sont les principes des conversions implicite et explicites ?
9. Qu'est-ce qu'un tableau 1D ?
10. Qu'est-ce qu'une chaîne de caractère AZT ?
11. Qu'est-ce qu'une structure ?

# 7

# Instructions, structures de contrôle et fonctions

## 1 Instructions et bloc d'instructions

Une instruction c'est

- Soit une déclaration de variable locale terminée par ;
  - N'effectue aucun traitement « visible »
  - Avec ou sans initialisation
- Soit une affectation terminée par ;
  - Affectation simple ou étendue
- Soit un appel de fonction terminée par ;
- Soit une structure de contrôle (cf. ci-après)
- Soit un bloc d'instructions { }

Un bloc d'instruction

- Commence par {
- Se termine par }
- Contient une séquence d'instructions

## 2 Structures de contrôle

### 2.1 Structures de contrôle

Trois catégories de structures de contrôle

- Sélection
- Itération
- Sauts

Rappel

- Pas de booléen en C mais une convention
- 0  $\implies$  Faux
- $\neq 0 \implies$  Vrai

### 2.2 Sélection

#### 2.2.1 Si Alors Sinon

Mots clés

if et else

Syntaxe

```
1 if (expression)
2 instruction;
```

```
1 if (expression)
2 instruction1;
3 else
4 instruction2;
```

```
1 if (expression)
2 {
3 instruction1;
4 instruction2;
5 }
6 else
7 instruction3;
```

### Cas de plusieurs if else imbriqués

- **else** est toujours associé au **if** le précédent (non déjà associé à un **else** dans un même bloc d'instructions)
- Mettre en forme le code pour une meilleure lisibilité

### Exemple 1

```
1 #include <stdio.h>
2 #include <stdlib.h> // pour la fonction rand()
3
4 int main(void)
5 {
6 int magique, devine;
7
8 magique = rand();
9
10 printf("Devinez le nombre magique\n");
11 scanf("%d", &devine);
12
13 if (devine == magique)
14 printf("\n ** Gagne ! **\n");
15
16 return 0;
17 }
```

### Exemple 2

```
1 #include <stdio.h>
2 #include <stdlib.h> // pour la fonction rand() et srand()
3
4 int main(void)
5 {
6 int magique, devine;
7
8 srand(1); /* Initialisation du générateur de nombres pseudo-aléatoires */
9 magique = rand();
10
11 printf("Devinez le nombre magique\n");
12 scanf("%d", &devine);
13
14 if (devine == magique)
15 printf("\n ** Gagne ! **\n");
16 else
17 {
18 printf("\n ** Perdu ! **\n");
19 printf("Le nombre était : %d\n", magique);
20 }
21
22 return 0;
23 }
```

### Exemple 3

```
1 #include <stdio.h>
2 #include <stdlib.h> // pour la fonction rand() et srand()
3
4 int main(void)
```

```

5 {
6 int magique, devine;
7 srand(1); /* Initialisation du générateur de nombres pseudo-aléatoires */
8 magique = rand();
9 printf("Devinez le nombre magique\n");
10 scanf("%d", &devine);
11
12 if (devine == magique)
13 printf("\n ** Gagne ! **\n");
14 else
15 {
16 printf("\n ** Perdu ! **\n");
17 if (devine > magique)
18 printf("Trop grand\n");
19 else
20 printf("Trop petit\n");
21 printf("Le nombre était : %d\n", magique);
22 }
23
24 return 0;
25 }
```

#### Exemple 4

```

1 #include <stdio.h>
2
3 #define TVA 0.206
4 #define TAUX1 0.01
5 #define TAUX2 0.03
6 #define TAUX3 0.05
7
8 int main(void)
9 {
10 float montant_ht, montant_ttc, prix;
11 float taux_remise;
12
13 printf("Montant hors taxe : ");
14 scanf("%f", &montant_ht);
15
16 montant_ttc = montant_ht * (1.0 + TVA);
17
18 if (montant_ttc < 1000.0)
19 taux_remise = 0.0;
20 else
21 if (montant_ttc >= 1000.0 && montant_ttc < 2000.0)
22 taux_remise = TAUX1;
23 else
24 if (montant_ttc >= 2000.0 && montant_ttc < 5000.0)
25 taux_remise = TAUX2;
26 else
27 if (montant_ttc >= 5000.0)
28 taux_remise = TAUX3;
29
30 prix = montant_ttc * (1.0 - taux_remise);
31
32 printf("Montant HT : %.2f\n", montant_ht);
33 printf("Montant TTC : %.2f\n", montant_ttc);
34 printf("Taux de la remise : %.1f et prix payé : %.2f\n", taux_remise,
35 prix);
36
37 }
```

### 2.2.2 Switch

#### A quoi ça sert

- Remplace plusieurs **if**
- Teste une valeur **entière** (char inclus, enum inclus)
- Test de type **égalité**
- Compare le résultat d'une expression à des constantes

## 7 Instructions, structures de contrôle et fonctions

---

### Mots clés

switch, case, break et default

#### Exemple

```
1 if (x==1)
2 instruction1;
3 else
4 if (x==2)
5 instruction2;
6 else
7 if (x==3)
8 instruction3;
```

```
1 switch (x)
2 {
3 case 1 : instruction1;
4 break;
5 case 2 : instruction2;
6 break;
7 case 3 : instruction3;
8 break;
9 }
```

#### case

– Syntaxe

```
1 case Valeur : instruction;
```

- La valeur est une constante
- Les case sont exécutés séquentiellement

#### break

- Instruction particulière d'un case
- Optionnel
- Termine un cas particulier (case)
- Sort du switch

#### Attention

Sans break, l'exécution des cases se poursuit avec les cases dont valeur de la condition est supérieure à celle du cas examiné et qui sont située après le cas examiné.

#### default

- Optionnel
- Cas par défaut
- S'exécute si aucun cas particulier n'est vérifié

```
1 switch (expression)
2 {
3 case cstel : instruction1;
4 break;
5 case cste2 :
6 {
7 instruction2;
8 instruction3;
9 }
10 break;
11 case cste3 :
12 {
13 instruction4;
14 break;
15 }
16 case cste4 : instruction5;
17 case cste5 : instruction6;
18 break;
19 case cste6 :
20 case cste7 : instruction7;
21 default : instruction8;
22 }
```

## Instructions exécutées en fonction de la valeur de l'expression

- cste1 : instruction1
- cste2 : instruction2, instruction3
- cste3 : instruction4
- cste4 : instruction5, instruction6
- cste5 : instruction6
- cste6 : instruction7, instruction8
- cste7 : instruction7, instruction8
- autre valeur : instruction8

### Exemple

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5 int nombre;
6 printf("Choisir un nombre entre 1 et 3 :\n");
7 scanf("%d", &nombre);
8
9 printf("Vous avez choisi : ");
10 switch (nombre)
11 {
12 case 1 : printf("Un\n");
13 break;
14 case 2 : printf("Deux\n");
15 break;
16 case 3 : printf("Trois\n");
17 break;
18 default : printf("Erreur ! Mauvais choix !\n");
19 }
20 return 0;
21 }
```

## 2.3 Itérations

### 2.3.1 for

#### Syntaxe

```

1 for(initialisation; test; incrément)
2 instruction;
```

#### Initialisation

- Expression séquentielle (,)
- Utilisé pour initialiser des variables

#### Test

- Condition qui doit être vraie pour que instruction soit exécutée
- Si test == Faux Alors sortie de la boucle

#### Incrément

- Expression séquentielle (,)
- Utilisé pour faire évoluer des variables

#### Comportement

1. Evaluation de initialisation
2. Si test est vrai Alors      Exécuter instruction Sinon      Sortir de la boucle
3. Exécuter incrément
4. Aller en 2

## 7 Instructions, structures de contrôle et fonctions

---

### Remarque

Initialisation et incrément peuvent être déportés avant ou dans la boucle

```
1 initialisation;
2 for(; test;)
3 {
4 instruction;
5 incrément;
6 }
```

### Exemple

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main(void)
5 {
6 int i, j, k;
7 int puissance = 10;
8 int nombre;
9
10 for(i=0, j=2, k=0; i < 300; ++k)
11 {
12 i += j - k;
13 j += i - k;
14 }
15
16 for(nombre = 1, k=1; k < puissance; ++k)
17 nombre = nombre * 2 + 1
18
19 return 0;
20 }
```

### 2.3.2 while

---

#### Syntaxe

```
1 while (condition)
2 instruction;
```

#### Comportement

1. Si condition est vrai Alors      Exécuter instruction
2. Sinon      sortir de la boucle
3. Aller en 1

### Remarque

**while**  $\iff$  **for** sans initialisation ni incrément

```
1 for(; condition ;)
2 instruction;
3
4 while (condition)
5 instruction;
```

### Exemple

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5 int count;
```

```

7 count = 1;
8 while (count <= 20)
9 {
10 printf("%d\n", count);
11 count++;
12 }
13 return 0;
14 }
```

### 2.3.3 do while

#### Syntaxe

```

1 do
2 {
3 instruction;
4 }
5 while (condition);
```

#### Comportement

1. Exécuter les instructions
2. Si condition est vraie Alors      Aller en 1
3. Sinon      Sortir de la boucle

#### Remarque

Répéter instruction Jusqu'à condition :

```

1 do
2 {
3 instruction;
4 }
5 while (!(condition));
```

#### Exemple

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5 int choix;
6
7 do
8 {
9 printf("1. Saisir le nom\n");
10 printf("2. Saisir le prénom\n");
11 printf("3. Saisir la ville\n");
12 printf("4. Saisir le code postal\n");
13 printf("Entrer votre choix :\n");
14 scanf("%d", &choix);
15 }
16 while (choix < 1 && choix > 4);
17 printf("Vous avez sélectionné le numéro %d\n", choix);
18 }
```

### 2.3.4 Les boucles d'itération

#### Remarques

- Les boucles **for** et **while** sont équivalentes
- Une boucle **for** permet en une seule ligne d'initialiser, contrôler et incrémenter
- Une boucle **do while** s'utilise quand on veut exécuter au moins 1 fois le corps de la boucle
- Les boucles peuvent être imbriquées

### 2.4 Sauts

---

#### 2.4.1 goto

---

##### Branchement sur une étiquette

- Usage interdit !
  - Rend les programmes plus difficiles à comprendre
  - Rend les programmes plus difficiles à maintenir
- Les autres structures de contrôle remplacent cette instruction avantageusement
- Projet de C :
  - présence d'un **goto**  $\Rightarrow$  0 !!
  - exception possible mais très largement justifiée (1p de rapport min)

#### 2.4.2 break

---

##### Utilisation

- Sortie d'un **switch**
- Sortie immédiate d'une boucle (**for**, **while**, **do while**)  $\Rightarrow$  dans le corps d'une boucle

##### Remarque

- Dans les boucles imbriquées : seule la boucle la plus interne est interrompue
- Avec de bonnes pratiques, **break** n'est jamais nécessaire

#### 2.4.3 continue

---

##### Utilisation

- Toujours dans le corps d'une boucle
- Comportement :
  - Arrête l'exécution des instructions de la boucle **et**
  - Force une nouvelle itération de la boucle
- Attention : comme **break**, cela n'agit que sur la boucle la plus interne

#### 2.4.4 Exemple

---

```
1 #include <stdio.h>
2 int main(void)
3 {
4 int index;
5 for(index = 0; index < 9; index++)
6 {
7 if (index == 6)
8 break;
9 printf("Valeur de index dans la boucle1 : %d\n", index);
10 if (index == 5)
11 printf("\tInterruption par break\n\n");
12 }
13 for(index = 0; index < 9; index++)
14 {
15 if (index == 6) continue;
16 printf("Valeur de index dans la boucle1 : %d\n", index);
17 if (index == 5)
18 {
19 printf("\tInterruption par continue\n\n");
20 printf("Les intégrations reprennent à 7");
21 }
22 }
23 return 0;
24 }
```

### 3 Fonctions

#### 3.1 Définitions et syntaxe

##### 3.1.1 Définition

###### Une fonction en C

- bloc de code en C
- indépendant
- référencé par un nom
- réalise une tâche précise
- peut admettre des paramètres
- peut renvoyer une valeur

##### 3.1.2 Syntaxe

###### Organisation syntaxique

```

1 TypeRetourné NomDeLaFonction(ListeDesParamètres)
2 {
3 instructions;
4 }
```

###### TypeRetourné

- tout type de données valide en langage C
- ou **void** si pas de valeur à retourner

###### NomDeLaFonction

identificateur, même convention que pour les variables

###### Corps de fonction

- c'est un bloc d'instructions !

###### ListeDesParamètres

- **void** : aucun paramètre
- **Type Nom, Type1 Nom1, Type2 Nom2, ...**

###### Remarques

- Le type retourné peut être omis : par défaut, c'est un **int**  $\Rightarrow$  **Très fortement déconseillé !**
- Les paramètres se comportent comme des variables locales de la fonction

##### 3.1.3 Prototype de fonctions

###### Définition

Fourni au compilateur la description d'une fonction définie plus loin, dans un autre fichier source ou dans une bibliothèque

###### Syntaxe

```

1 TypeRetourné NomDeLaFonction(ListeDesParamètres);
```

Cette notation s'appelle l'entête ou le prototype de la fonction

```

1 void fct1(int n);
2 double carre(double x);
3 double puissance(double x, double y);
4 int mean(void);
```

### Remarque

Habituellement situé en début de programme ou dans un fichier d'extension .h

## 3.2 Passage de paramètres

---

### 3.2.1 Passage de paramètres par valeur

---

#### Les paramètres sont transmis exclusivement par valeur

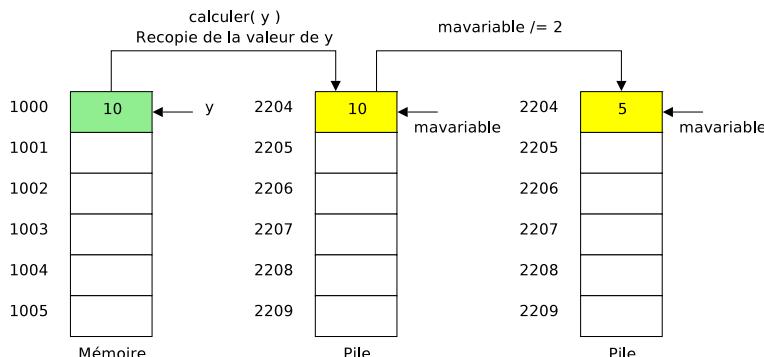
1. Les paramètres transmis à l'appel de la fonction sont évalués
2. Le résultat des expressions sont **copiés** dans la pile
3. La fonction est appelée

### Remarque

La fonction récupère les **copies** des paramètres transmis via la pile d'appel. La fonction peut modifier ces variables **mais** seule les copies sont modifiées.

```

1 #include <stdio.h>
2
3 void calculer(int mavariable)
4 {
5 mavariable /= 2;
6 }
7
8 int main(void)
9 {
10 int y = 10;
11
12 calculer(y);
13
14 printf("y vaut %d\n", y);
15
16 }
```



### 3.2.2 Passage de paramètres par adresse

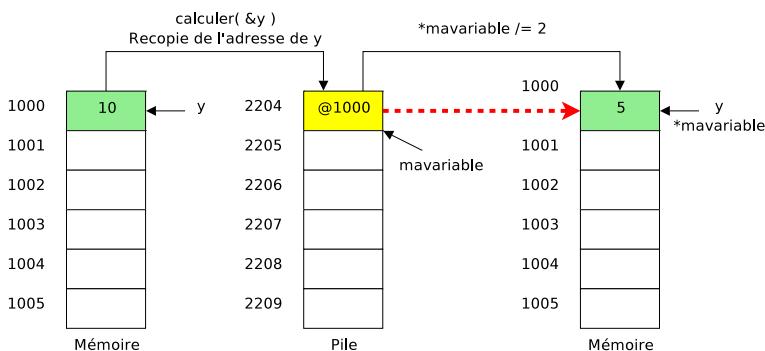
---

#### On peut transmettre l'adresse de la variable

A l'aide de l'adresse d'une variable, on peut accéder et modifier la variable transmise  $\Rightarrow$  On peut modifier la variable qui est hors de la fonction

```

1 #include <stdio.h>
2
3 void calculer(int * mavariable)
4 {
5 *mavariable /= 2;
6 }
7
8 int main(void)
9 {
10 int y = 10;
11
12 calculer(&y);
13
14 printf("y vaut %d\n", *y);
15
16 }
```



### 3.3 Renvoyer des résultats

#### 3.3.1 Retourner une valeur : return

##### Syntaxe

```
1 return expression;
```

##### Comportement

- Expression est évaluée
- La valeur est alors éventuellement convertie en **TypeRetourné** : même règle qu'une affectation
- On sort de la fonction

##### Remarque

- Si pas de valeur à retourner (**void**)
- **return;** permet de forcer la sortie de la fonction

#### 3.3.2 Comment retourner plusieurs valeurs : passage par adresse

##### Pour retourner plusieurs valeurs simultanément

- On utilise la passage par adresse
- Un exemple : **scanf**
- On peut utiliser cette technique en combinaison ou non avec **return**

#### 3.3.3 Appel de fonctions

##### Appel d'une fonction sans paramètre

```
1 NomDeLaFonction();
2 lvalue = NomDeLaFonction();
```

##### Appel d'une fonction avec des paramètres

```
1 NomDeLaFonction(ListeDesArguments);
2 lvalue = NomDeLaFonction(ListeDesArguments);
```

#### 3.3.4 La récurrence

```
1 int Fibonacci(int n)
2 {
3 if (n == 0)
4 return 0;
5 else
6 if (n == 1)
7 return 1;
8 else
9 return Fibonacci(n-1)+Fibonacci(n-2);
10 }
```

## 7 Instructions, structures de contrôle et fonctions

---

### Remarques

- Les paramètres sont transmis par valeur / copie
- Pour modifier une variable extérieure à une fonction  $\Rightarrow$  transmettre l'adresse de la variable !

#### 3.3.5 Exemple 1

```
1 #include <stdio.h>
2
3 int carre(int i);
4
5 int main(void)
6 {
7 int i = 10, j;
8
9 printf("Valeur de i dans main avant l'appel de la fonction carré : %d\n",
10 i);
11 puts("Appel de la fonction carré\n");
12 j = carre(i);
13 printf("Valeur de i dans main après l'appel de la fonction carré : %d\n",
14 i);
15 printf("Valeur de j : %d\n", j);
16 return 0;
17 }
18 int carre(int i)
19 {
20 printf("La valeur de i dans la fonction carré est : %d\n", i);
21 i *= i;
22 printf("Valeur retournée par la fonction carre : i = %d\n", i);
23 return i;
24 }
```

#### 3.3.6 Exemple 2

```
1 #include <stdio.h>
2
3 void echange(int * a, int * b);
4
5 int main(void)
6 {
7 int a = 10, b = 15;
8 printf("Avant l'appel de echange : (a,b,&a,&b)=(%d,%d,%p,%p)\n",
9 a, b, &a, &b);
10 echange(&a,&b);
11 printf("Après l'appel de echange : (a,b,&a,&b)=(%d,%d,%p,%p)\n",
12 a, b, &a, &b);
13 return 0;
14 }
15 void echange(int * x, int * y)
16 {
17 int temp;
18 printf("Adresse de x : %p, adresse de y : %p\n", &x, &y);
19 printf("Contenu des variables x et y : %p, %p\n", x, y);
20 printf("Valeurs pointées par x et y : %d, %d\n", *x, *y);
21 temp = *x; *x = *y; *y = temp;
22 puts("Fin de la fonction echange\n");
23 printf("Valeurs pointées par x et y : %d, %d\n", *x, *y);
24 }
```

### 3.4 La fonction main

---

#### 3.4.1 Arguments de la fonction main

##### Pas d'argument

```
1 int main(void)
```

## Récupération des paramètres de la ligne de commande

```
1 int main(int argc, char * argv [])
2 int main(int argc, char ** argv)
```

- Permet de passer des paramètres au programme au moment de l'exécution
- Exemple : nom de fichier, nombre d'itérations, ...
- **argc** le nombre d'argument de la ligne de commande
- **argv** isole les différents arguments
- **argv[0]** correspond au nom du programme
- **argv[i]** est une chaîne de caractères  $\Rightarrow$  même les nombres sont sous forme de chaînes de caractères !

### 3.4.2 Exemple

```
1 c:\> monprog.exe fichier.txt 43
```

**argc = 3, argv[0] = "monprog.exe", argv[1] = "fichier.txt", argv[2] = "43"**

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main(int argc, char * argv [])
5 {
6 char * Fichier;
7 int NumeroLigne;
8
9 if (argc < 3)
10 printf("Nombre de paramètres insuffisants !\n");
11 else
12 {
13 Fichier = argv [1];
14 NumeroLigne = atoi(argv [2]);
15 printf("Le fichier à traiter est %s et la ligne est %d\n", Fichier ,
16 NumeroLigne);
17 }
18 return 0;
19 }
```

### 3.4.3 Structures et prototype de fonctions

#### Passage par valeur

```
1 void fct(struct MaStructure ms)
```

Le contenu des champs est copié

#### Passage par adresse

```
1 void fct struct MaStructure * pms)
```

#### ✓ L'essentiel à retenir :

1. Qu'est-ce qu'une instruction ?
2. Quelle est la syntaxe d'un si alors sinon et d'un branchement conditionnel multiple ?
3. Comment écrit-on une boucle ? Quel est l'ordre d'évaluation des instructions dans une boucle **for** ?
4. Qu'est-ce qu'une fonction ? Qu'est-ce qu'un prototype ? A quoi ça sert ?
5. Quelle est la différence entre un passage de paramètres par valeur ou par adresse ?
6. Quelle est la syntaxe complète de la fonction **main** ?



# 8

# Quelques fonctions simples et étude de codes existants

## 1 L'écran et le clavier

### 1.1 Sorties de base : affichage écran

#### Sortie écran (stdio.h)

- printf : affichage formaté

```
1 int printf(const char *format, ...);
```

- puts : affichage d'une chaîne de caractères **suivit d'un saut de ligne**

```
1 int puts (const char * s);
```

- putchar : affichage d'un caractère

```
1 int putchar(int c);
```

```
1 #include <stdio.h>
2 void f()
3 {
4 char * s = "Le premier paramètre est %s";
5 printf(""); printf(s); printf("\n");
6 printf(""); printf("%s", s); printf("\n");
7 printf(""); puts(s); printf("\n");
8 }
```

### 1.2 Entrées de base : saisie clavier

#### Entrée clavier (stdio.h)

- scanf : saisie formatée et affectation à des variables

```
1 int scanf(const char * format, ...);
```

%s : les caractères sont pris en compte tant qu'ils ne sont pas des « blancs »

- gets : saisie d'une chaîne de caractères jusqu'au retour chariot (écrit dans s, retourne s mais ne fait aucune vérification de débordement)

```
1 char * gets(char * s);
```

- getchar : saisie d'un caractère

```
1 int getchar(void);
```

```
1 #include <stdio.h>
2 void f()
3 {
4 char buf[1024];
5 gets(buf);
6 scanf("%s", buf);
7 }
```

## 2 Fonctions de bases

### 2.1 Les principales fonctions mathématiques

La valeur absolue

```
1 int abs (int j); // stdlib.h
2 double fabs (double x); // math.h
```

Fonctions trigonométrique (math.h)

```
1 double cos (double x);
2 double sin (double x);
3 double tan (double x);
4 double acos (double x);
5 double asin (double x);
6 double atan (double x);
7 double cosh (double x);
8 double sinh (double x);
9 double tanh (double x);
```

Fonctions exponentielles, logarithmiques et puissances (math.h)

```
1 double exp (double x);
2 double exp2 (double x);
3 double exp10 (double x);
4 double log (double x);
5 double log2 (double x);
6 double log10 (double x);
7 double sqrt (double x);
8 double pow (double x, double y);
```

Les arrondis (math.h)

```
1 // arrondi à l'entier supérieur
2 double ceil (double x);
3 // arrondi à l'entier inférieur
4 double floor (double x);
5 // arrondi à l'entier le plus proche
6 double round (double x);
7 // calcul le reste de la division entière de x par y
8 double fmod (double x, double y);
```

Remarques

- Il existe des fonctions qui admettent des float / long double et qui renvoient des float / long double
- Pour utiliser la bibliothèque mathématique avec gcc il faut ajouter "-lm" à la ligne de commande

### 2.2 Caractères et chaînes de caractères

Fonctions pour les caractères (ctype.h)

```
1 int isalnum (int c);
2 int isalpha (int c);
3 int isblank (int c); // espace ou tabulation
4 int isdigit (int c);
5 int islower (int c);
6 int isspace (int c); // blanc (espace, tab, saut de ligne)
7 int isupper (int c);
8 int isxdigit (int c); // 0-9, A-F, a-f
9 int toupper (int c);
10 int tolower (int c);
```

## Conversion et formattage

```

1 int atoi (const char *nptr); // stdlib.h
2 long atol(const char *nptr); // stdlib.h
3 double atof (const char *nptr); // stdlib.h
4 int sprintf (char *str, const char *format, ...); // stdio.h
5 int snprintf (char *str, size_t size, const char *format, ...); // stdio.h
6 int sscanf (const char * str, const char * format, ...); // stdio.h

```

## 2.3 Fonctions diverses

### Les nombres aléatoires (stdlib.h)

```

1 void srand (unsigned int seed);
2 int rand (void);
3 #define RAND_MAX

```

### Outils systems

```

1 void assert (int expression); // assert.h, désactivée si NDEBUG
2 void abort (void); // stdlib.h
3 void exit (int status); // stdlib.h
4 int system (const char * commande); // stdlib.h
5 // temps du processeur utilisé par le processus
6 // diviser par CLK_TCK pour obtenir des secondes
7 clock_t clock (void); // time.h, CLK_TCK
8 // temps écoulé depuis le 1/01/1970
9 time_t time(time_t *t); // time.h

```

## 3 Etude de codes existants

### 3.1 Exemple 1

```

1 #include <stdio.h>
2 #include <math.h>
3
4 float delta(float a, float b, float c)
5 {
6 return b*b-4*a*c;
7 }
8
9 int main(void)
10 {
11 float a, b, c,d;
12 printf(" Résolution de A X^2+B X+C =0\n ");
13
14 printf(" A ? ");
15 scanf("%f",&a);
16 printf(" B ? ");
17 scanf("%f",&b);
18 printf(" C ? ");
19 scanf("%f",&c);
20
21 d = delta(a,b,c);
22 printf(" Le descriminant vaut :%f\n ",d);
23
24 if (d >=0)
25 {
26 float x1, x2;
27 x1 = (-b+sqrt(d))/2./a;
28 x2 = (-b-sqrt(d))/2./a;
29 printf(" Les solutions sont :\n X1= %f \n X2= %f\n ",x1,x2);
30 }
31 else
32 {
33 float x_r, x_i;
34 x_r=-b/2./a;
35 x_i = sqrt(-d)/2./a;
36 printf(" Les solutions sont imaginaires :\n x1= %f +i %f \n x2= %f -i %f\n ",
37 x_r,x_i,x_r,x_i);
38 }
39 return 0;
40 }

```

### 3.2 Exemple 2

```

1 #include <stdio.h>
2 #include <math.h>
3
4 double x ;
5
6 void carre_de_x1 (void) ;
7 void carre_de_x2 (void) ;
8
9 void carre_de_x1 (void)
10 {
11 x = x * x ;
12 return ;
13 }
14
15 void carre_de_x2 (void)
16 {
17 double x ;
18 x = x * x ;
19 return ;
20 }
21
22 int main (void)
23 {
24 x = 2.0 ;
25
26 carre_de_x1 () ;
27 printf ("x = %f\n",x) ;
28
29 carre_de_x2 () ;
30 printf ("x = %f\n",x) ;
31
32 return 1 ;
33 }
```

### 3.3 Exemple 3

```

1 #include <stdio.h>
2
3 #define MAXDONNEES 8
4
5 int tab[MAXDONNEES];
6
7 int lire(void);
8 void trier(int);
9 void afficher(int);
10
11 void main(void)
12 {
13 int NbrDonnees;
14
15 NbrDonnees = lire();
16 trier(NbrDonnees);
17 afficher(NbrDonnees);
18 }
19
20 int lire(void)
21 {
22 int donnee, nbr = 0;
23 printf("Entrez les données entières positives à trier.\n"
24 "Terminez la saisie en tapant -1.\n"
25 "Vous avez droit à au plus %d valeurs\n",MAXDONNEES);
26 scanf("%d",&donnee);
27 while((nbr <= MAXDONNEES) && (donnee != -1))
28 {
29 if (nbr == MAXDONNEES)
30 {
31 printf("\nvous avez donné plus de %d entiers, seuls les %d "
32 "premiers seront triés\n", MAXDONNEES, MAXDONNEES);
33 break;
34 }
35 else
36 {
```

```

37 tab[nbr] = donnee ;
38 nbr++;
39 scanf("%d" , &donnee);
40 }
41 }
42 return nbr;
43 }

44
45 void trier(int nombre)
46 {
47 int i , j , cle ;
48
49 for (i = 1; i < nombre; i++)
50 {
51 cle = tab[i];
52 for (j = i-1; (j >= 0) && (cle<tab[j]); j--) tab[j+1] = tab[j];
53 tab[j+1] = cle ;
54 }
55 }
56
57 void afficher(int nombre)
58 {
59 int i ;
60
61 printf("\nvoici le tableau trié\n");
62 for(i = 0; i < nombre; i++) printf("%5d" ,tab[i]);
63 printf("\n");
64 }
```

### 3.4 Exemple 4

```

1 #include <stdio .h>
2
3 int main ()
4 {
5 char n;
6 int i , fact = 1;
7
8 printf("entrez une valeur entière positive inférieure ou égale à 7 : ");
9 scanf("%d" ,&n);
10 while((n<=0) || (n>7))
11 {
12 if (n<=0) printf("j'ai demandé une valeur positive , redonnez la valeur :");
13 else printf("j'ai demandé une valeur inférieure à 8 , redonnez la valeur :");
14 scanf("%d" ,&n);
15 }
16 for(i = 2; i <= n; i++) fact*=i;
17 printf("la valeur de %d! est %d\n" ,n ,fact);
18 return 0;
19 }
```

### 3.5 Exemple 5

```

1 #include <stdio .h>
2
3 struct st _ date
4 {
5 int jour , annee;
6 char *mois;
7 };
8 struct st _ personne
9 {
10 char *nom, *prenom;
11 int age;
12 struct st _ date date;
13 };
14 int main()
15 {
16 struct st _ personne paul = {
17 "Dupont" , "Paul" , 45 ,
18 {29, 1962, "Janvier"} };
19
20 printf("%s est né en %d\n" , paul.prenom , paul.date.annee);
```

## 8 Quelques fonctions simples et étude de codes existants

```
21 return 0;
22 }
```

### 3.6 Exemple 6

```
1 #include <stdio.h>
2
3 #define N 3
4
5 void copie_tab(int orig[N], int dest[N])
6 {
7 int i;
8 for (i=0;i<N; i=i+1)
9 dest[i]=orig[i];
10 }
11
12 int main()
13 {
14 int tab1[N]={1,2,3}, tab2[N];
15 copie_tab(tab1,tab2);
16 printf("%d\t%d\t%d\n", tab2[0], tab2[1], tab2[2]);
17 return 0;
18 }
```

### 3.7 Exemple 7

```
1 #include <stdio.h>
2
3 int main()
4 {
5 char * s="abghjahjartarytaamliaoaoipa";
6 int i = 0, compteur=0;
7 char c = 'a';
8
9 while (s[i] != '\0')
10 {
11 if (s[i] == c)
12 compteur = compteur + 1;
13 i = i + 1;
14 }
15 printf("\'%c\' apparaît %d fois dans \"%s\".\n", c, compteur, s);
16 return 0;
17 }
```

### 3.8 Exemple 8

```
1 #include <stdio.h>
2
3 int main()
4 {
5 int a=1, b=4, c=100;
6
7 while (c!=0)
8 {
9 if ((b - 2*a) == 0)
10 break;
11 c = c/(b - 2*a);
12 b = b / 2;
13 }
14 printf("%d\n", c);
15 return 0;
16 }
```

### 3.9 Exemple 9

```
1 #include <stdio.h>
2
3 int main()
4 {
5 struct
```

```

6 {
7 char *nom, *p9renom;
8 short age;
9 } paul = { "Dupont", "Paul", 45 }, clone;
10
11 clone=paul;
12 printf("paul: %d clone: %d.\n", paul.age, clone.age);
13
14 clone.age=clone.age+1;
15 printf("paul: %d clone: %d.\n", paul.age, clone.age);
16 return 0;
17 }
```

### 3.10 Exemple 10

```

1 #ifndef __fs_h
2 #define __fs_h
3
4 struct FS {
5 const char* fsname; /* file system name (e. g. 'ext2') */
6 const char* options; /* standard mount options (must not be empty) */
7 int support_ugid; /* whether the fs supports uid and gid options */
8 const char* umask; /* umask value (NULL if umask is not supported) */
9 int support_ioccharset; /* whether the fs supports the ioccharset option */
10 };
11
12 const struct FS* get_fs_info(const char* fsname);
13 const struct FS* get_supported_fs();
14
15#endif
```

```

1 #include "fs.h"
2 #include <string.h>
3
4 static struct FS supported_fs[] = {
5 { "udf", "nosuid,nodev,user", 1, "007", 1 },
6 { "iso9660", "nosuid,nodev,user", 1, NULL, 1 },
7 { "vfat", "nosuid,nodev,user,quiet,shortname=mixed", 1, "077", 1 },
8 { "ntfs", "nosuid,nodev,user", 1, "077", 1 },
9 { "hfsplus", "nosuid,nodev,user", 1, NULL, 0 },
10 { "hfs", "nosuid,nodev,user", 1, NULL, 0 },
11 { "ext3", "nodev,noauto,nosuid,user", 0, NULL, 0 },
12 { "ext2", "nodev,noauto,nosuid,user", 0, NULL, 0 },
13 { "reiserfs", "nodev,noauto,nosuid,user", 0, NULL, 0 },
14 { "reiser4", "nodev,noauto,nosuid,user", 0, NULL, 0 },
15 { "xfs", "nodev,noauto,nosuid,user", 0, NULL, 0 },
16 { "jfs", "nodev,noauto,nosuid,user", 0, NULL, 1 },
17 { NULL, NULL, 0, NULL }
18 };
19
20 const struct FS*
21 get_supported_fs() {
22 return supported_fs;
23 }
24
25 const struct FS*
26 get_fs_info(const char* fsname) {
27 struct FS* i;
28
29 for(i = supported_fs; i->fsname; ++i)
30 if(!strcmp(i->fsname, fsname))
31 return i;
32 return NULL;
33 }
```

#### ✓ L'essentiel à retenir :

1. Suis-je capable de lire et comprendre un code C ?



# 9

# Le C en pratique

## 1 Fichiers sources et fichiers d'entête

### 1.1 Structures des fichiers sources

Pour compiler un programme, deux types de fichiers

- Fichier d'entête : .h
- Fichier de code : .c

Fonctionnement par paire

- **fichier .h** : Déclare ce qui est publique
- **fichier .c** : Implémente ce qui est publique
- Définissent un module

Pour utiliser ce qui est publique d'un module implementé dans module.c

```
1 #include "module.h"
2 #include <module.h>
```

Le contenu du fichier est copié/collé à la place de la commande

**Fichier d'entête contient**

- Des inclusions d'autres fichiers d'entête
- Déclaration des constantes publiques du module
- Déclaration des types publiques du module
- Déclaration des variables globales publiques du module (**extern**)
- Déclaration des fonctions publiques du module

**Fichier de code contient**

- L'inclusion du fichier d'entête correspondant
- Des inclusions d'autres fichiers d'entête
- Déclaration des constantes privées du module
- Déclaration des types privés du module
- Déclaration/implémentation des variables globales privées du module (**static**)
- Déclaration/implémentation des variables globales publiques du module
- Implémentation des fonctions publiques du module
- Implémentation des fonctions privées du module (**static**)

### 1.2 Exemple de fichier avant la séparation

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define TAILLE_MAX_NOM_PRODUIT 20
5 #define TAILLE_MAX_CODE_PRODUIT 5
6 #define TAILLE_MAX_CATALOGUE 100
7 #define TAILLE_MAX_NOM_CLIENT 50
8 #define TAILLE_MAX_FACTURE 20
9
10 struct Produit
11 {
12 char Nom[TAILLE_MAX_NOM_PRODUIT];
```

## 9 Le C en pratique

---

```
13 char Code[TAILLE_MAX_CODE_PRODUT] ;
14 double PrixHT;
15 double TVA;
16 }
17
18 struct Ligne
19 {
20 struct Produit LeProduit;
21 int Quantite;
22 };
23
24 struct Facture
25 {
26 char Nom[TAILLE_MAX_NOM_CLIENT];
27 int NbLigne;
28 struct Ligne LesLignes [TAILLE_MAX_FACTURE];
29 };
30
31 static int TailleCatalogue = 0;
32 static struct Produit Catalogue [TAILLE_MAX_CATALOGUE];
33
34 int NombreProduitsCrees = 0;
35
36 void AjouterProduitAuCatalogue(struct Produit LeProduit)
37 {
38 ...
39 }
40
41 void EffacerProduitAuCatalogue(char LeCode[TAILLE_MAX_CODE])
42 {
43 ...
44 }
45
46 void AfficherCatalogue ()
47 {
48 ...
49 }
50
51 struct Produit Rechercher (char LeCode[TAILLE_MAX_CODE])
52 {
53 ...
54 }
55
56 struct Produit CreerProduit(char LeNom[TAILLE_MAX_NOM],
57 char LeCode[TAILLE_MAX_CODE],
58 double LePrixHT, double LaTVA)
59 {
60 ...
61 }
62
63 void AfficherProduit(struct Produit * LeProduit)
64 {
65 ...
66 }
67
68 struct Facture CreerFactureVierge ()
69 {
70 ...
71 }
72
73 void AjouterLigne(struct Facture * LaFacture , struct Produit LeProduit ,
74 int LaQuantite)
75 {
76 ...
77 }
78
79 void AfficherFacture (struct Facture * LaFacture)
80 {
81 ...
82 }
83
84 int main(void)
85 {
86 ...
87 }
```

### 1.3 Exemple de fichiers après la séparation : produit.h

```

1 #ifndef PRODUIT_H
2 #define PRODUIT_H
3
4 #define TAILLE_MAX_NOM_PRODUIT 20
5 #define TAILLE_MAX_CODE_PRODUIT 5
6
7 struct Produit
8 {
9 char Nom[TAILLE_MAX_NOM_PRODUIT];
10 char Code[TAILLE_MAX_CODE_PRODUIT];
11 double PrixHT;
12 double TVA;
13 };
14
15 extern int NombreProduitsCrees;
16
17 struct Produit CreerProduit(char LeNom[TAILLE_MAX_NOM],
18 char LeCode[TAILLE_MAX_CODE],
19 double LePrixHT, double LaTVA);
20
21 void AfficherProduit(struct Produit * LeProduit);
22
23 #endif

```

### 1.4 Exemple de fichiers après la séparation : produit.c

```

1 #include "produit.h"
2 #include <stdio.h>
3
4 int NombreProduitsCrees = 0;
5
6 static void CopierChaine(char * Source, char * Dest)
7 {
8 ...
9 }
10
11 struct Produit CreerProduit(char LeNom[TAILLE_MAX_NOM],
12 char LeCode[TAILLE_MAX_CODE],
13 double LePrixHT, double LaTVA)
14 {
15 ...
16 NombreProduitsCrees++;
17 }
18
19 void AfficherProduit(struct Produit * LeProduit)
20 {
21 ...
22 }

```

### 1.5 Exemple de fichiers après la séparation : catalogue.h

```

1 #ifndef CATALOGUE_H
2 #define CATALOGUE_H
3
4 #include "produit.h"
5
6 void AjouterProduitAuCatalogue(struct Produit LeProduit);
7 void EffacerProduitAuCatalogue(char LeCode[TAILLE_MAX_CODE]);
8 void AfficherCatalogue();
9 struct Produit Rechercher(char LeCode[TAILLE_MAX_CODE]);
10
11 #endif

```

### 1.6 Exemple de fichiers après la séparation : catalogue.c

```

1 #include "catalogue.h"
2 #include <stdio.h>
3
4 #define TAILLE_MAX_CATALOGUE 100

```

## 9 Le C en pratique

---

```
5 static int TailleCatalogue = 0;
6 static struct Produit Catalogue[TAILLE_MAX_CATALOGUE];
7
8 void AjouterProduitAuCatalogue(struct Produit LeProduit)
9 {
10 }
11 ...
12 }
13 void EffacerProduitAuCatalogue(char LeCode[TAILLE_MAX_CODE])
14 {
15 }
16 ...
17 void AfficherCatalogue()
18 {
19 }
20 ...
21 struct Produit Rechercher(char LeCode[TAILLE_MAX_CODE])
22 {
23 }
24 }
```

### 1.7 Exemple de fichiers après la séparation : facture.h

---

```
1 #ifndef FACTURE_H
2 #define FACTURE_H
3 #include "produit.h"
4
5 #define TAILLE_MAX_NOM_CLIENT 50
6 #define TAILLE_MAX_FACTURE 20
7
8 struct Ligne
9 {
10 struct Produit LeProduit;
11 int Quantite;
12 };
13 struct Facture
14 {
15 char Nom[TAILLE_MAX_NOM_CLIENT];
16 int NbLigne;
17 struct Ligne LesLignes[TAILLE_MAX_FACTURE];
18 };
19
20 struct Facture CreerFactureVierge();
21 void AjouterLigne(struct Facture * LaFacture , struct Produit LeProduit ,
22 int LaQuantite);
23 void AfficherFacture(struct Facture * LaFacture);
24
25#endif
```

### 1.8 Exemple de fichiers après la séparation : facture.c

---

```
1 #include "facture.h"
2 #include <stdio.h>
3
4 struct Facture CreerFactureVierge()
5 {
6 ...
7 }
8
9 void AjouterLigne(struct Facture * LaFacture , struct Produit LeProduit ,
10 int LaQuantite)
11 {
12 ...
13 }
14
15 void AfficherFacture(struct Facture * LaFacture)
16 {
17 ...
18 }
```

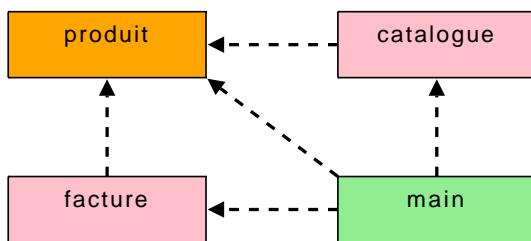
## 1.9 Exemple de fichiers après la séparation : main.c

```

1 #include "produit.h"
2 #include "catalogue.h"
3 #include "facture.h"
4 #include <stdlib.h>
5 #include <stdio.h>
6
7 void f()
8 {
9 ...
10}
11
12 int main(void)
13 {
14 ...
15 printf("Produits crées : %d\n", NombreProduitsCrees);
16 return 0;
17 }

```

## 1.10 Dépendances entre les modules



## 2 Mise en forme de fichiers sources

### 2.1 Mise en forme

#### 2.1.1 Mise en forme de code source

```

1 if (prix < 1000.0) taux_remise = 0.0; else
2 if (prix < 2000.0)
3 taux_remise = TAUX1;
4 else
5 if (prix < 5000.0)
6 taux_remise = TAUX2; else
7 if (prix >= 5000.0 || prix < 2000.0)
8 taux_remise = TAUX3;

```

```

1 int i, j = 0;
2 for(i = 0; i < 100; ++i) {
3 while (i > 100) if (i < 30)
4 j++; if (i > 30) j++;

```

#### Intérêt

- Le compilateur ne se préoccupe pas de l'indentation/mise en forme
- On peut donner du sens à l'indentation/mise en forme pour améliorer la clareté du code

#### Caractéristiques importantes

- Il faut choisir une convention et s'y tenir du début à la fin !
- La convention doit mettre en évidence les hiérarchies/imbrications d'instructions

## 9 Le C en pratique

---

### Remarque

ne pas hésiter à éclaircir son code avec des lignes vides !

```
1 if (prix < 1000.0)
2 taux_remise = 0.0;
3 else
4 if (prix < 2000.0)
5 taux_remise = TAUX1;
6 else
7 if (prix < 5000.0)
8 taux_remise = TAUX2;
9 else
10 if (prix >= 5000.0 || prix < 2000.0)
11 taux_remise = TAUX3;
```

```
1 int i, j = 0;
2
3 for(i = 0; i < 100; ++i)
4 {
5 while (i > 100)
6 if (i<30)
7 j++;
8 if (i>30)
9 j++;
10 }
```

### 2.1.2 Quelques exemples de mise en forme classique

---

```
1 if (x > 0) {
2 x--;
3 }
4
5 if (x > 0)
6 {
7 x--;
8 }
9
10 if (x > 0)
11 {
12 x--;
13 }
14
15 if (x > 0)
16 x--;
17
18 if (x > 0) {
19 x--;
20 } else {
21 x++;
22 }
```

```
1 if (x > 0)
2 {
3 x--;
4 }
5 else
6 {
7 x++;
8 }
9
10 if (x > 0)
11 x--;
12 else
13 x++;
14
15 do {
16 x--;
17 } while (x);
18
19 do {
20 x--;
21 }
22 while (x);
```

```

1 do
2 {
3 x--;
4 }
5 while (x);
6
7 switch (i)
8 {
9 case 0:
10 break;
11 case 1:
12 {
13 ++i;
14 }
15 default:
16 break;
17 }
```

```

1 switch (i)
2 {
3 case 0:
4 break;
5 case 1:
6 {
7 ++i;
8 }
9 default:
10 break;
11 }
12
13 switch (i)
14 {
15 case 0:
16 break;
17 case 1:
18 {
19 ++i;
20 }
21 default:
22 break;
23 }
```

```

1 while ((e_code - s_code) < (dec_ind - 1))
2 {
3 set_buf_break (bb_dec_ind);
4 *e_code++ = ',';
5 }
6
7 while ((e_code-s_code) < (dec_ind-1))
8 {
9 set_buf_break(bb_dec_ind);
10 *e_code++ = ',';
11 }
12
13 calculer(x);
14 calculer(x);
15
16 void calculer(int x);
17 void calculer(int x);
18
19 void fonction(char* x);
20 void fonction(char * x);
```

```

1 void foo (int arg1, char arg2, int *arg3, long arg4, char arg5);
2 void foo (
3 int arg1,
4 char arg2,
5 int *arg3,
6 long arg4,
7 char arg5);
8 void foo (
9 int arg1,
10 char arg2,
```

## 9 Le C en pratique

---

```
11 int *arg3 ,
12 long arg4 ,
13 char arg5
14);
15
16 struct foo {
17 int x;
18 };
19
20 struct foo
21 {
22 int x;
23 };
```

```
1 p1 = first_procedure (second_procedure (p2, p3),
2 third_procedure (p4, p5));
3 p1 = first_procedure (second_procedure (p2, p3),
4 third_procedure (p4, p5));
```

### Remarque

- Il n'y a pas de véritable norme de mise en forme en C !
- C'est une affaire de goût ! Faites votre choix !

## 2.2 Bonnes pratiques

---

### 2.2.1 Nommer les choses utilement

```
1 #include <stdio.h>
2
3 #define C1 5.5
4 #define C2 19.6
5
6 int main(void)
7 {
8 double a, b;
9 scanf("%lf", &a);
10 if (a < 10000)
11 b = a * (1.0+C1);
12 else
13 b = a * (1.0+C2);
14 printf("\n%lf\n", b);
15 return 0;
16 }
```

### Question

A quoi sert ce programme ???

```
1 #include <stdio.h>
2
3 #define TVA1 5.5
4 #define TVA2 19.6
5
6 int main(void)
7 {
8 double PrixHT, PrixTTC;
9 scanf("%lf", &PrixHT);
10 if (PrixHT < 10000)
11 PrixTTC = PrixHT * (1.0+TVA1);
12 else
13 PrixTTC = PrixHT * (1.0+TVA2);
14 printf("\n%lf\n", PrixTTC);
15 return 0;
16 }
```

### Question

A quoi sert ce programme ???

## 2.2.2 Quelques conventions de nommages

- Les identificateurs :

```

1 int mavariableamoi;
2 int ma_variable_a_moi;
3 int MaVariableAMoi;
4 int maVariableAMoi;
```

- Les paramètres de fonctions :

```
1 void f(int Param1, int a_Param2, int _Param3, int the_Param4, int The_Param5);
```

- Les membres de structures, ... :

```

1 struct MaStructure
2 {
3 int m_Champ1;
4 int _Champ2;
5 int champ3;
6 int Champ4;
7 };
```

## 2.2.3 Commenter utilement

- Sans commentaire

```

1 #include <stdio.h>
2
3 #define TVA1 5.5
4 #define TVA2 19.6
5
6 int main(void)
7 {
8 double PrixHT, PrixTTC;
9 scanf("%lf", &PrixHT);
10 if (PrixHT < 10000)
11 PrixTTC = PrixHT * (1.0+TVA1);
12 else
13 PrixTTC = PrixHT * (1.0+TVA2);
14 printf("\n%lf\n", PrixTTC);
15 return 0;
16 }
```

- Avec des commentaires

```

1 /* Inclusion de stdio.h pour pouvoir utiliser les fonctions : printf et scanf */
2 #include <stdio.h>
3
4 /* Les deux constantes de TVA à 5.5% et à 19.6% */
5 #define TVA1 5.5
6 #define TVA2 19.6
7
8 /* La fonction main : c'est là que tout commence
9 * Pas de paramètres parce qu'on en a pas besoin
10 */
11 int main(void)
12 {
13 /* Deux variables pour stocker le prix HT et le prix TTC */
14 double PrixHT, PrixTTC;
15
16 /* Saisie du prix HT */
17 scanf("%lf", &PrixHT);
```

```

1 /* Calcul du prix TTC en fonction du prix HT
2 * Lorsque le prix HT est inférieur à 10000, on applique la TVA à 5.5%
3 * Sinon on applique la TVA à 19.6% */
4 if (PrixHT < 10000)
5 PrixTTC = PrixHT * (1.0+TVA1); // on applique la TVA à 5.5%
6 else
7 PrixTTC = PrixHT * (1.0+TVA2); // on applique la TVA à 19.6%
8
9 // on affiche le prix TTC
```

## 9 Le C en pratique

---

```
10 printf("\n%lf\n", PrixTTC);
11
12 return 0; // et on termine le programme
13 }
```

### Attention

- Trop commenter peut nuire à la compréhension du code
- Ne pas assez commenter est aussi problématique  $\Rightarrow$  **Il faut trouver le juste milieu !**

### Remarques

- Les conventions de mise en forme réduisent le besoin de commentaires
- Les conventions de nommage réduisent le besoin de commentaires
- Aérer son code réduit le besoin de commentaires
- Il faut pouvoir comprendre le code très rapidement mais sans le polluer de trop de commentaires

## 2.3 Doxygen

---

### 2.3.1 Documenter utilement avec Doxygen

---

#### Doxygen est un logiciel de documentation de code source

- <http://www.doxygen.org>
- Sous licence GNU GPL
- Fonctionne sous Windows, Linux, Mac
- Langages supportés : C, C++, Java, Objective-C, Python, IDL, PHP, C#, D

#### Intérêts

- La documentation est intégrée au code source
- La documentation peut être très riche
- Le document généré est toujours à jour si les commentaires sont à jour !

#### En entrée

- Des fichiers sources (extraction d'infos non documentées)
- Des commentaires ordinaires (extraction d'infos partiellement documentées)
- Des commentaires contenant des balises spéciales (extraction d'infos documentées)

#### En sortie

- Une documentation de code / d'API
- Formats : HTML, Latex RTF, PostScript, PDF, Unix man pages

### 2.3.2 Exemple

---

```
1 #ifndef DBUS_ERROR_H
2 #define DBUS_ERROR_H
3
4 #include <dbus/dbus-macros.h>
5 #include <dbus/dbus-types.h>
6
7 DBUS_BEGIN_DECLS
8
9 typedef struct DBusError DBusError;
10
11 struct DBusError {
12 const char *name;
13 const char *message;
14 unsigned int dummy1 : 1;
15 unsigned int dummy2 : 1;
16 unsigned int dummy3 : 1;
17 unsigned int dummy4 : 1;
18 unsigned int dummy5 : 1;
19 void *padding1;
20 };
21
22
```

```

23 void dbus_error_init (DBusError *error);
24 void dbus_error_free (DBusError *error);
25 void dbus_set_error (DBusError *error,
26 const char *name,
27 const char *message,
28 ...);
29 void dbus_set_error_const (DBusError *error,
30 const char *name,
31 const char *message);
32 void dbus_move_error (DBusError *src,
33 DBusError *dest);
34 dbus_bool_t dbus_error_has_name (const DBusError *error,
35 const char *name);
36 dbus_bool_t dbus_error_is_set (const DBusError *error);

37 DBUS_END_DECLS
38
39 #endif /* DBUS_ERROR_H */

```

### 2.3.3 Documentation générée

[Main Page](#) [Modules](#) [Data Structures](#) [Files](#) [Related Pages](#)  
[Data Structures](#) [Class Hierarchy](#) [Data Fields](#)

## DBusError Struct Reference

Object representing an exception. [More...](#)

#include <[dbus-errors.h](#)>

### Data Fields

|              |                   |               |
|--------------|-------------------|---------------|
| const char * | <b>name</b>       | error name    |
| const char * | <b>message</b>    | error message |
| unsigned int | <b>dummy1</b> : 1 | placeholder   |
| unsigned int | <b>dummy2</b> : 1 | placeholder   |
| unsigned int | <b>dummy3</b> : 1 | placeholder   |
| unsigned int | <b>dummy4</b> : 1 | placeholder   |
| unsigned int | <b>dummy5</b> : 1 | placeholder   |
| void *       | <b>padding1</b>   | placeholder   |

---

### Detailed Description

Object representing an exception.

Definition at line 41 of file **dbus-errors.h**.

---

The documentation for this struct was generated from the following file:

- [dbus-errors.h](#)

Main Page

Modules

Data Structures

Files

Related Pages

## Error reporting

[D-Bus low-level public API]

Error reporting. [More...](#)

**Functions**

```
void dbus_error_init (DBusError *error)
 Initializes a DBusError structure.

void dbus_error_free (DBusError *error)
 Frees an error that's been set (or just initialized), then reinitializes the error as in dbus_error_init().

void dbus_set_error_const (DBusError *error, const char *name, const char *message)
 Assigns an error name and message to a DBusError.

void dbus_move_error (DBusError *src, DBusError *dest)
 Moves an error src into dest, freeing src and overwriting dest.

dbus_bool_t dbus_error_has_name (const DBusError *error, const char *name)
 Checks whether the error is set and has the given name.

dbus_bool_t dbus_error_is_set (const DBusError *error)
 Checks whether an error occurred (the error is set).

void dbus_set_error (DBusError *error, const char *name, const char *format,...)
 Assigns an error name and message to a DBusError.
```

### Detailed Description

Error reporting.

Types and functions related to reporting errors.

In essence D-Bus error reporting works as follows:

```
DBusError error;
dbus_error_init (&error);
dbus_something (arg1, arg2, &error);
if (dbus_error_is_set (&error))
{
 fprintf (stderr, "an error occurred: %s\n", error.message);
 dbus_error_free (&error);
}
```

There are some rules. An error passed to a D-Bus function must always be unset; you can't pass in an error that's already set. If a function has a return code indicating whether an error occurred, and also a **DBusError** parameter, then the error will always be set if and only if the return code indicates an error occurred. i.e. the return code and the error are never going to disagree.

An error only needs to be freed if it's been set, not if it's merely been initialized.

You can check the specific error that occurred using **dbus\_error\_has\_name()**.

#### **void dbus\_error\_init ( DBusError \* error )**

Initializes a **DBusError** structure.

Does not allocate any memory; the error only needs to be freed if it is set at some point.

**Parameters:**

error the **DBusError**.

Definition at line 151 of file **dbus-errors.c**.

References **\_dbus\_assert**, **DBusRealError::const\_message**, **DBusRealError::message**, **DBusRealError::name**, **NULL**, and **TRUE**.

Referenced by **\_dbus\_auth\_script\_run()**, **\_dbus\_keyring\_new\_homedir()**, **dbus\_error\_free()**, and **dbus\_move\_error()**.

### 2.3.4 Quelques balises utiles

#### Commentaires extraits

```
1 /**
2 * Commentaires
3 */
4
```

```

5 /**
6 * Commentaires
7 */
8
9 /// Commentaires
10
11 //! Commentaires
12
13 /*!
14 * Commentaires
15 */

```

## Commentaires courts

```

1 /*! | brief Voici une courte description
2 * sur plusieurs lignes.
3 *
4 * Et ici la description longue normale.
5 */

```

## Commentaires rapides

```

1 /// Un commentaire avant pour MaStructure
2 struct MaStructure
3 {
4 /// Un commentaire avant la variable pour MaVariable
5 int MaVariable;
6
7 int MonAutreVariable; ///< Un commentaire situé après pour MonAutreVariable
8 }

```

## Balisage

- **@param nom descriptiton** : décrire un paramètre de fonction
- **@return description** : décrire la valeur retournée par la fonction
- **@b mot, @c mot, @e mot** : le mot suivqnt est écrit en gras, courier ou italique
- **@code @endcode** : encadre une portion de code qui sera formatté comme du code
- **@n** : force un saut de ligne

## Il existe beaucoup d'autres tags :

@a @addindex @addtogroup @anchor @arg @attention @author @b @brief @bug @c @callgraph @callergraph @category @class @code @cond @copydoc @date @def @defgroup @deprecated @dir @dontinclude @dot @dotfile @e @else @elseif @em @endcode @endcond @enddot @endhtmlonly @endif @endlatexonly @endlink @endmanonly @endverbatim @endxmlonly @enum @example @exception @f\$ @f[ @f] @file @fn @hideinitializer @htmlinclude @htmlonly @if @ifnot @image @include @includeleno @ingroup @internal @invariant @interface @latexonly @li @line @link @mainpage @manonly @n @name @namespace @nosubgrouping @note @overload @p @package @page @par @paragraph @param @post @pre @private (PHP only) @privatesection @property @protected @protectedsection @protocol @public @publicsection @ref @relates @remarks @return @retval @sa @section @see @showinitializer @since @skip @skipline @struct @subpage @subsection @subsubsection @test @throw @todo @typedef @union @until @var @version @warning @weakgroup @xmlonly @xrefitem @\\$ @@ \\& \\< \\> \\# \\% Consultez la documentation de doxygen (réalisée avec doxygen !)

## 2.4 Conseils de codage en langage C

### Pour aller plus loin

[http://fr.wikibooks.org/wiki/Conseils\\_de\\_codage\\_en\\_C](http://fr.wikibooks.org/wiki/Conseils_de_codage_en_C)

## 3 Bibliothèques de code

### 3.1 Bibliothèques de code

#### Bibliothèque

- ensemble de fichiers objets
- regroupés dans un seul fichier
- issus de la compilation de fichiers sources (Asm, C, C++...)

## 9 Le C en pratique

### Exemples

- La bibliothèque standard du C
- Les bibliothèques du système d'exploitation

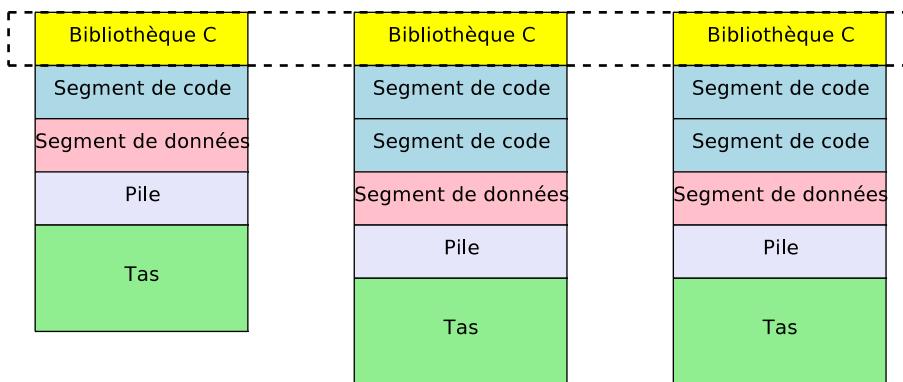
### Deux types de bibliothèque

- Les bibliothèques statiques
- Les bibliothèques dynamiques

#### 3.2 Bibliothèque statique

##### Principes

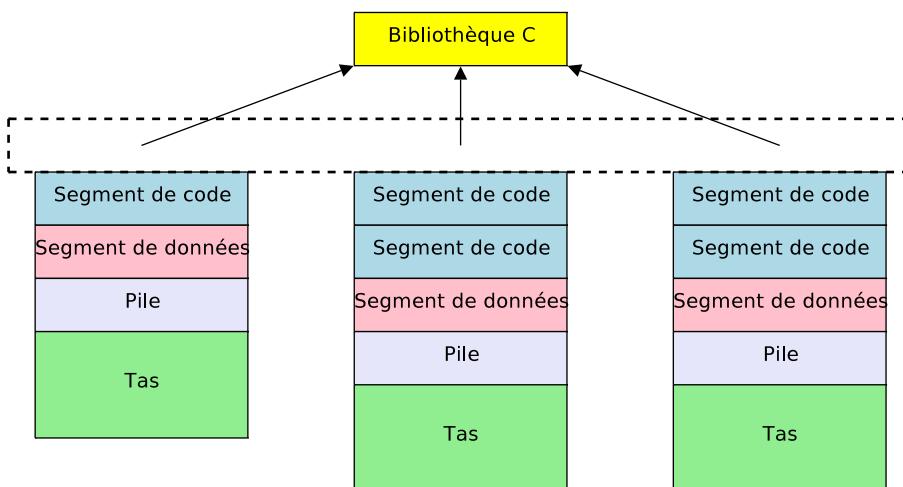
- Le code de la bibliothèque est intégré dans l'exécutable
- Plusieurs copies en mémoire !



#### 3.3 Bibliothèque dynamique

##### Principes

- Le code de la bibliothèque n'est pas intégré dans l'exécutable
- Une seule copie en mémoire !



### Intérêt des bibliothèques dynamiques

- Le gain en occupation mémoire peut être très conséquent !
- 1 bibliothèque c'est souvent plusieurs Mo !
  1. libc sous linux : 2.5 Mo
  2. environ 100 programmes utilisant la libc
  3. Si libc bibliothèque statique Alors 250 Mo de libc!!!
- 1 programme utilise souvent plusieurs bibliothèques !
- Certains systèmes d'exploitation ne permettent pas l'utilisation de bibliothèques dynamiques

### 3.4 Bibliothèques

#### Utilisation d'une bibliothèque en C

- Il faut le fichier de bibliothèque
- Il faut les entêtes .h correspondant
- Il faut indiquer au compilateur d'utiliser la bibliothèque

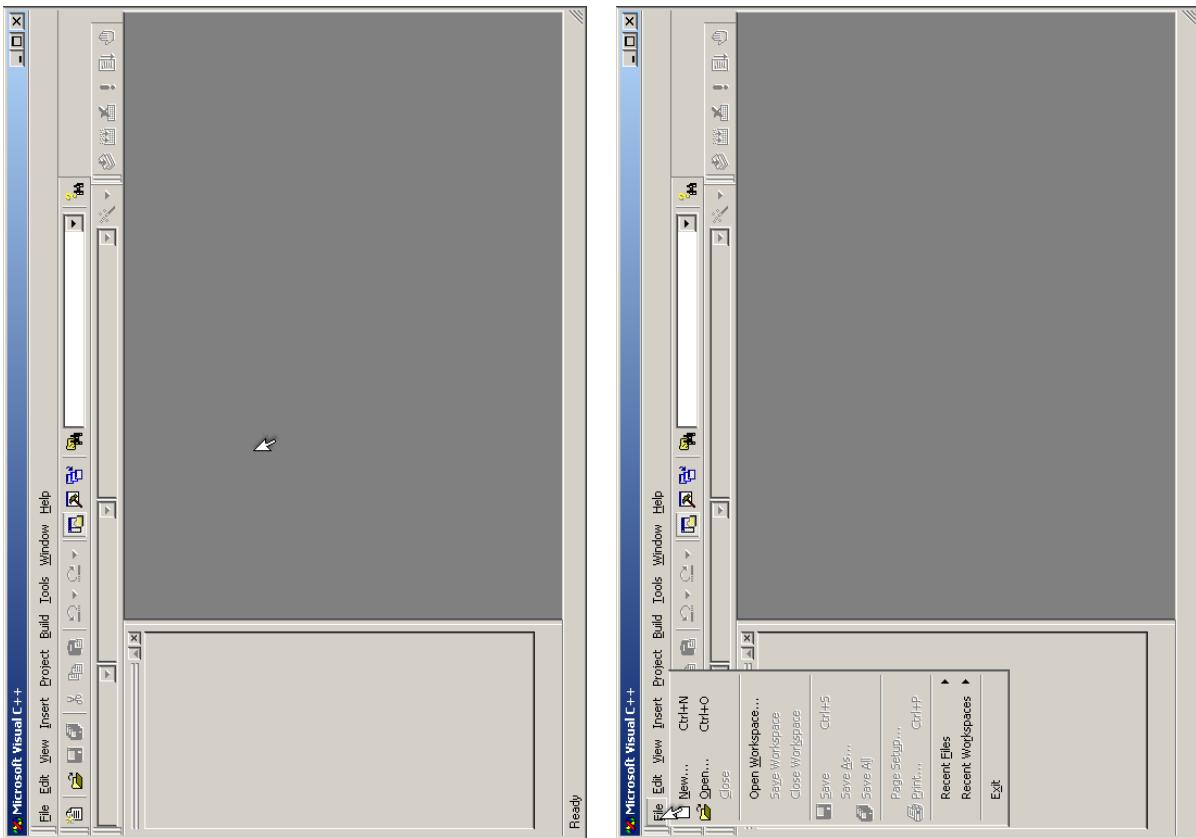
#### Création d'une bibliothèque en C

- Spécifique au système d'exploitation
- Spécifique au compilateur ==> consultez la documentation de votre compilateur !

## 4 Compilation par la pratique

### 4.1 Visual Studio 6

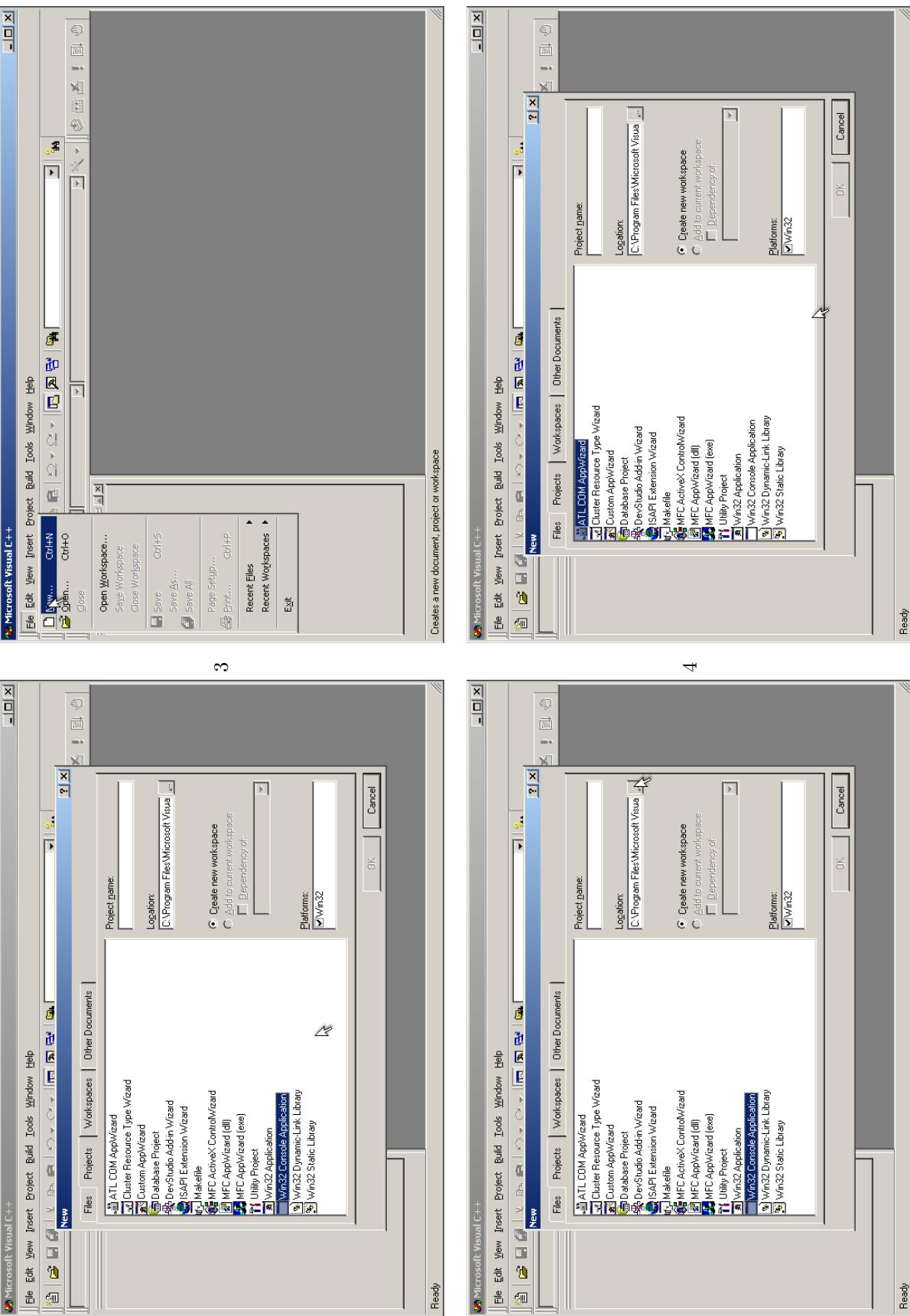
#### 4.1.1 Création du projet et des fichiers

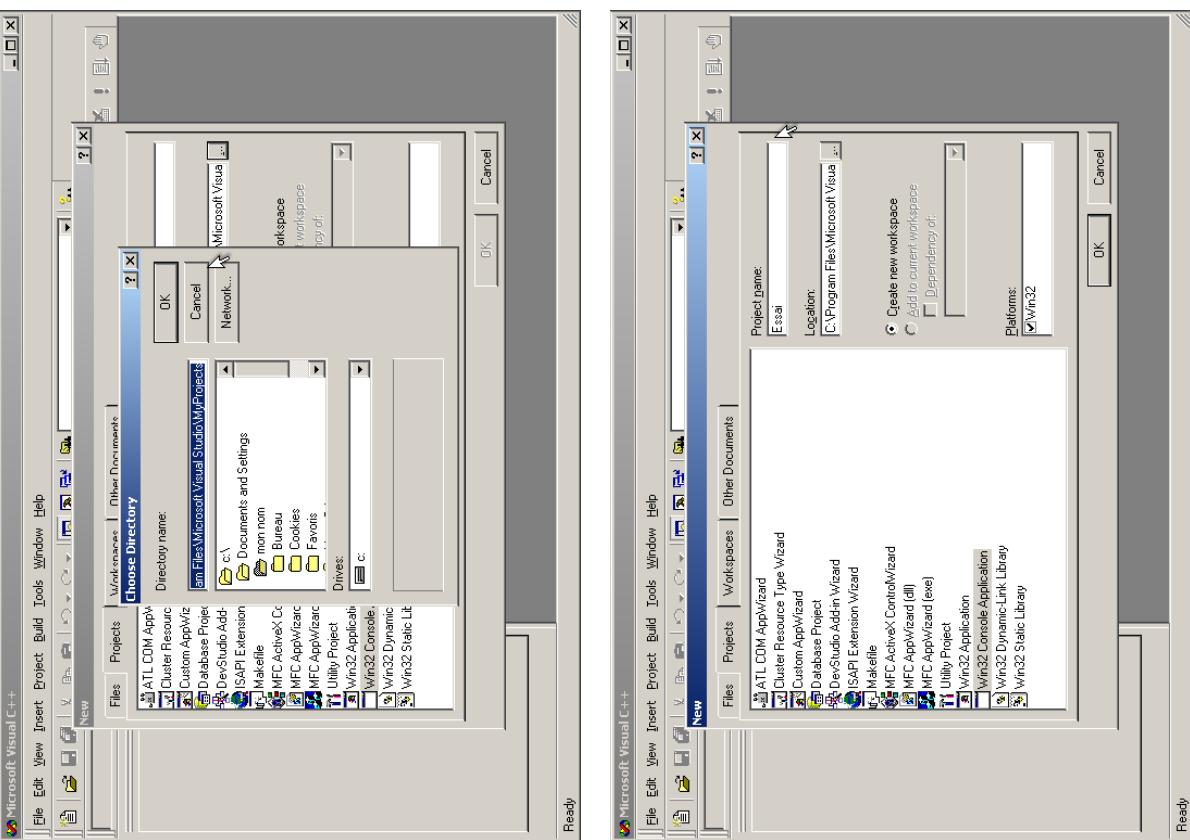


1

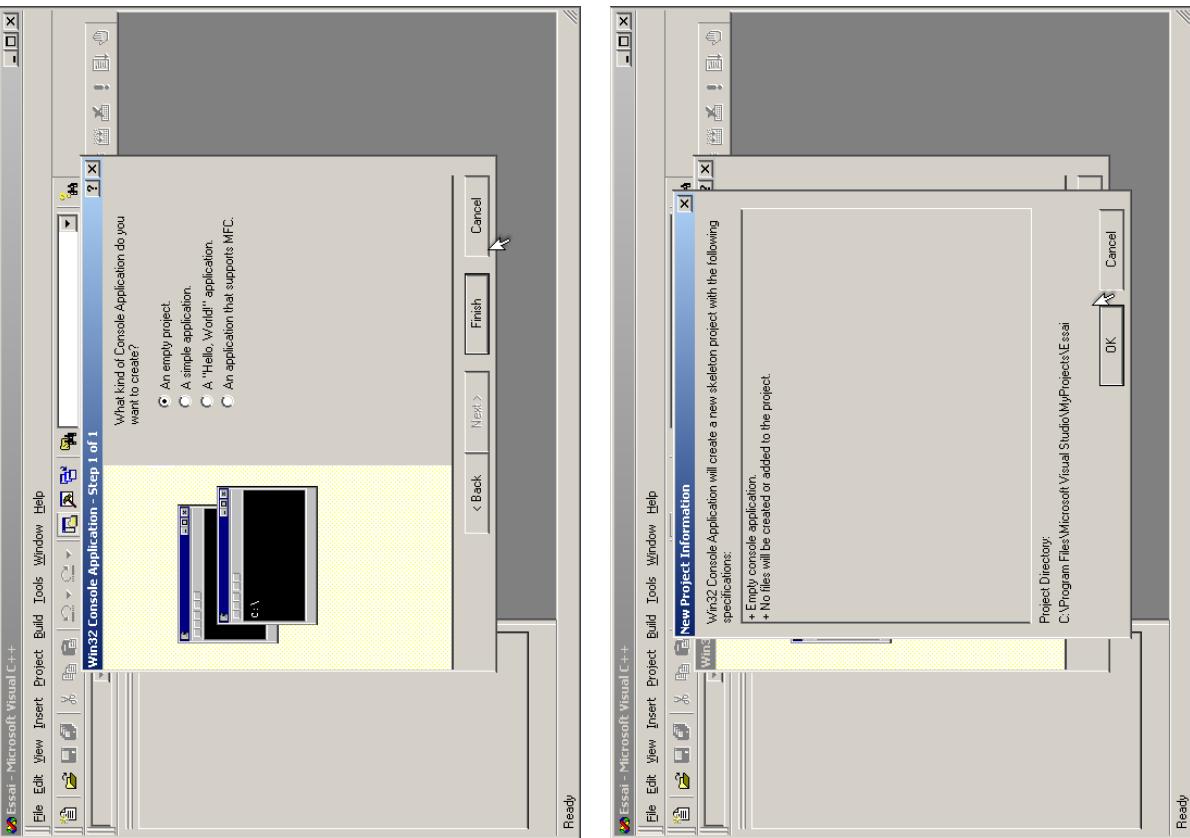
2

## 9 Le C en pratique





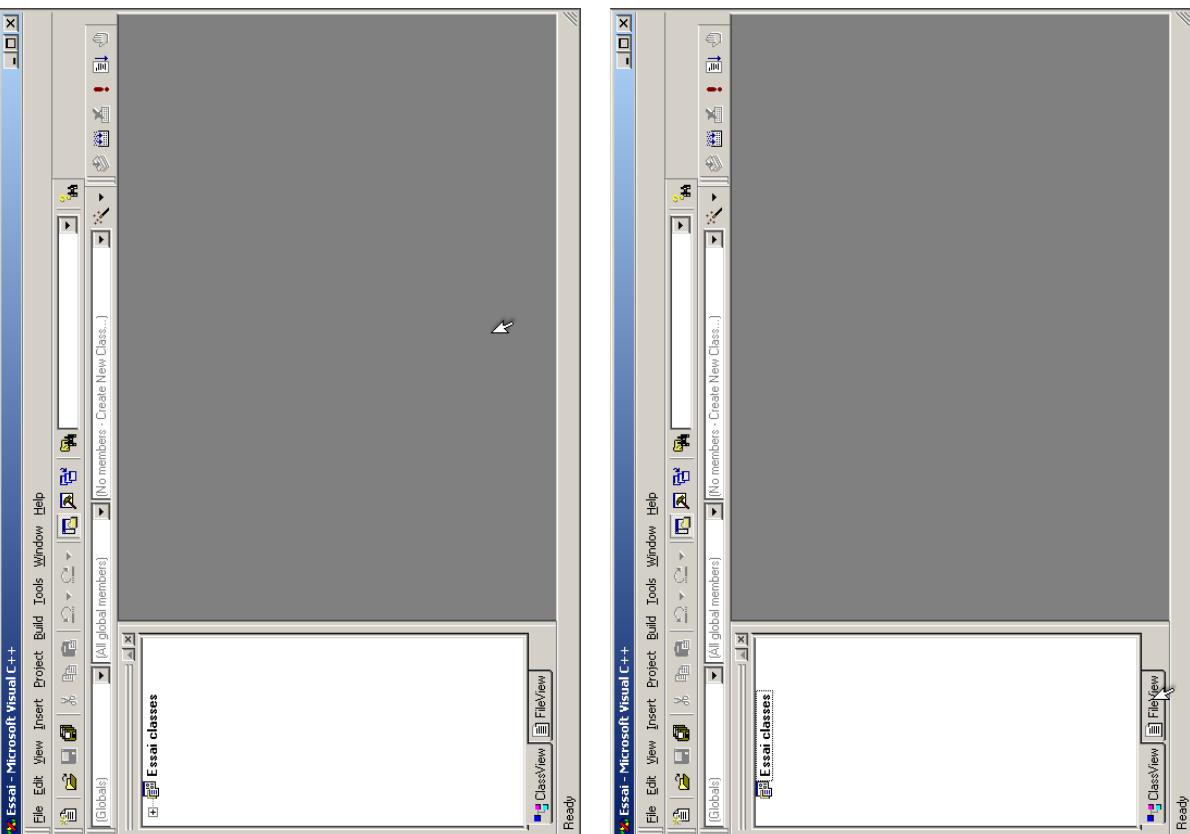
7



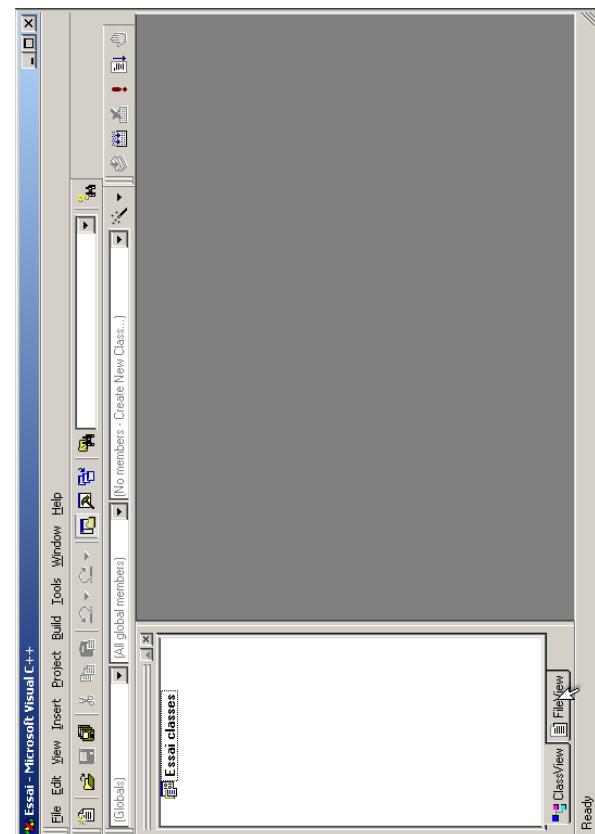
9

10

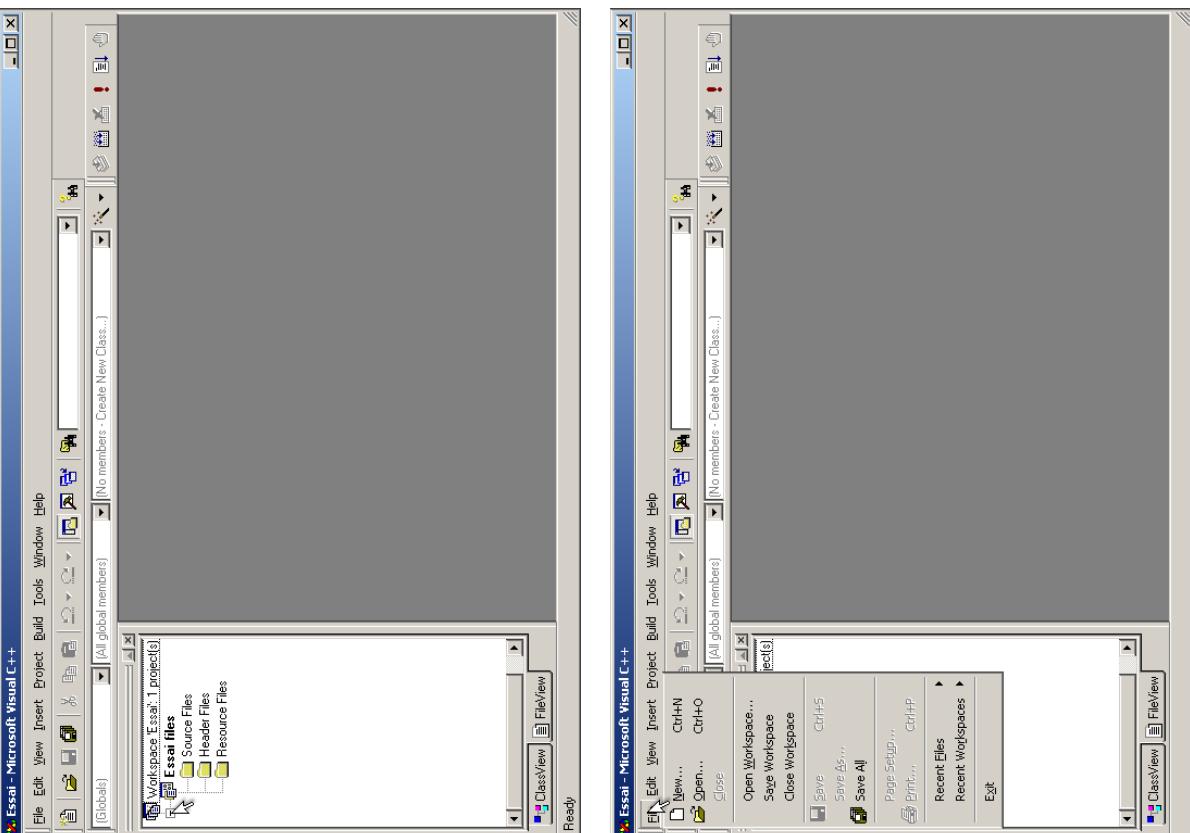
## 9 Le C en pratique



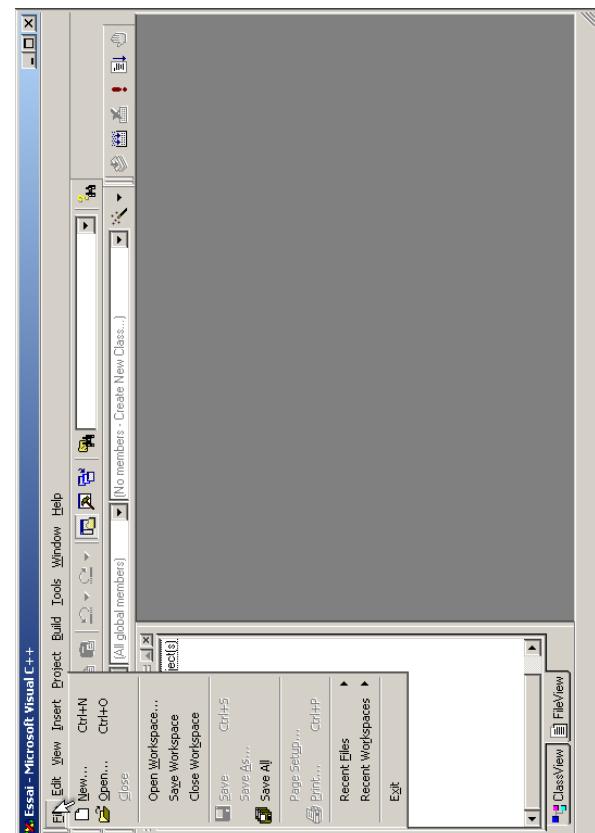
11



12



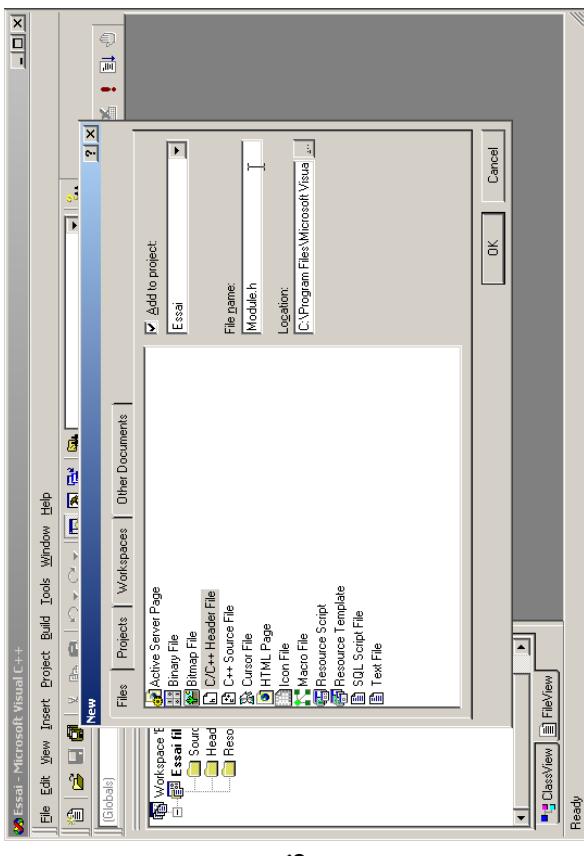
13



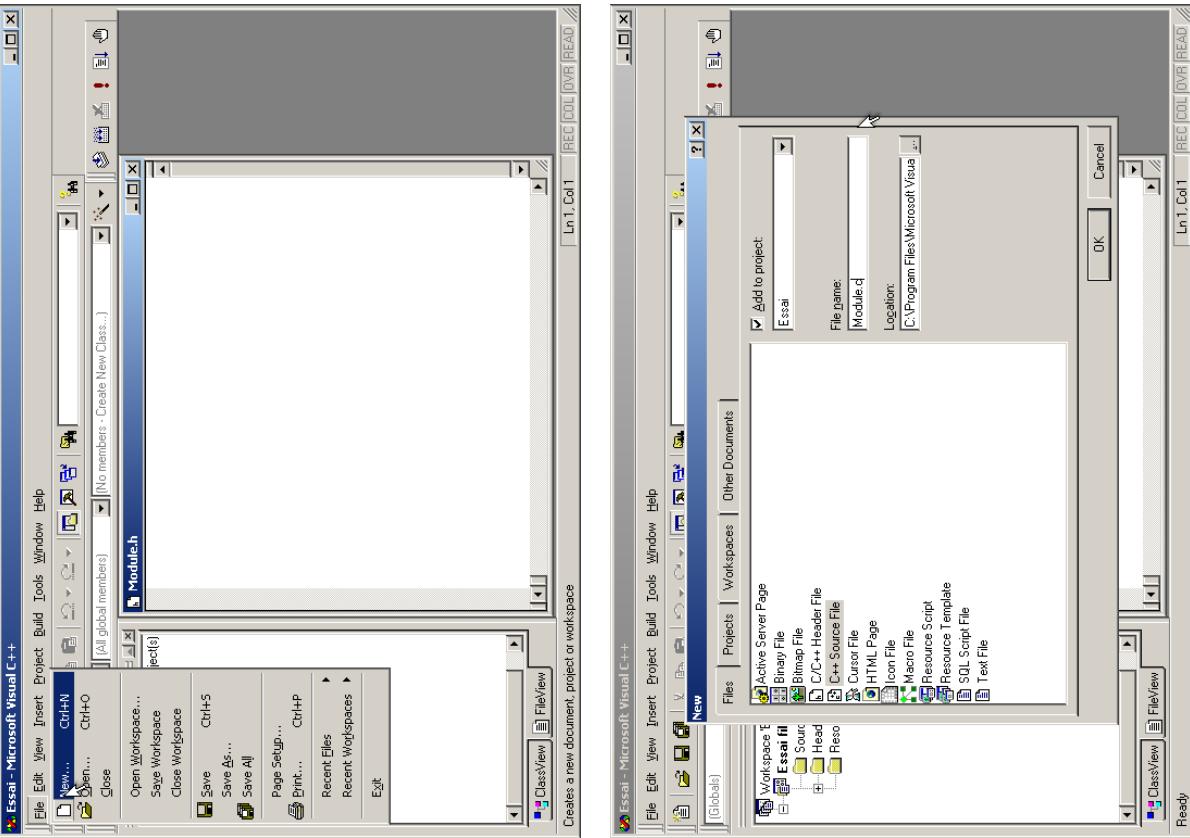
14



15

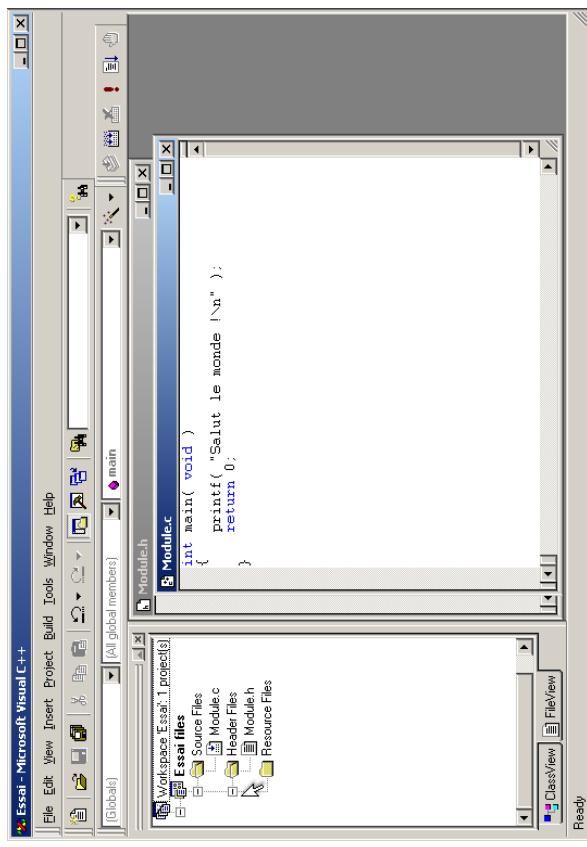
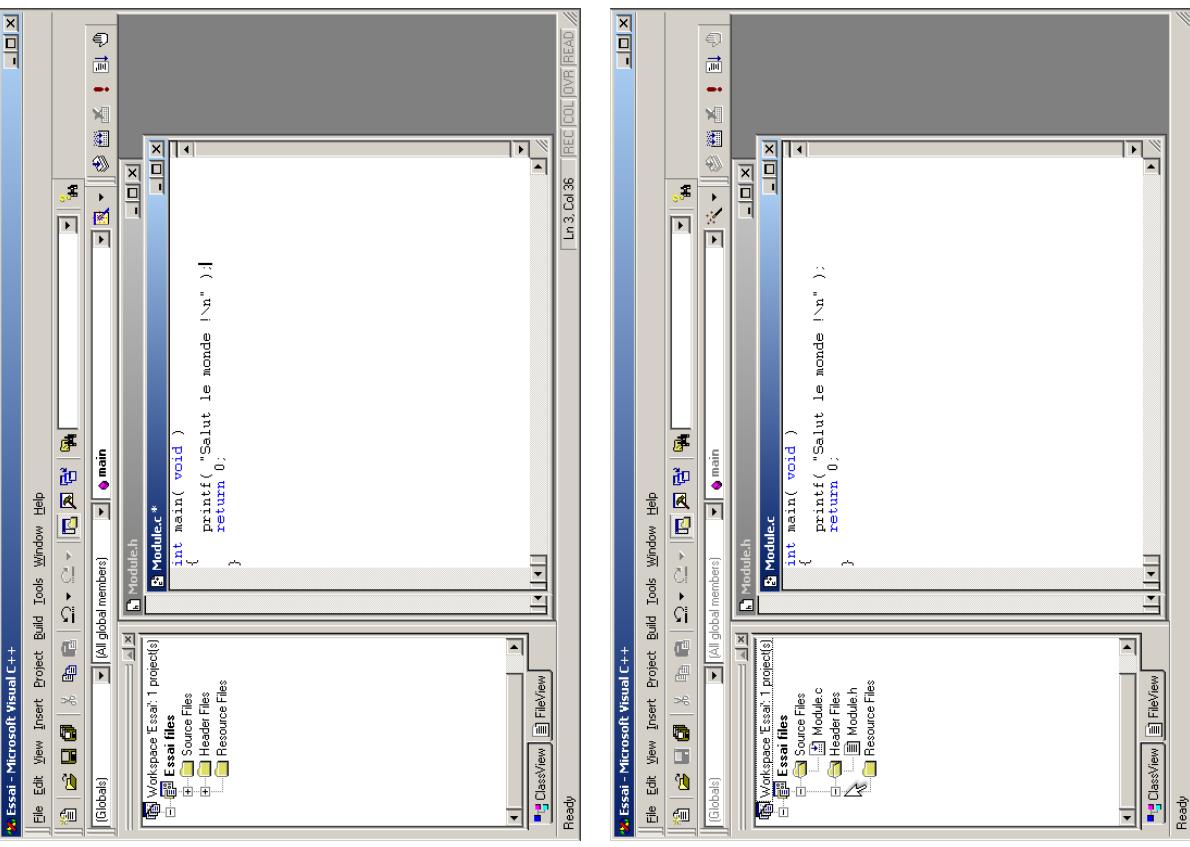


16

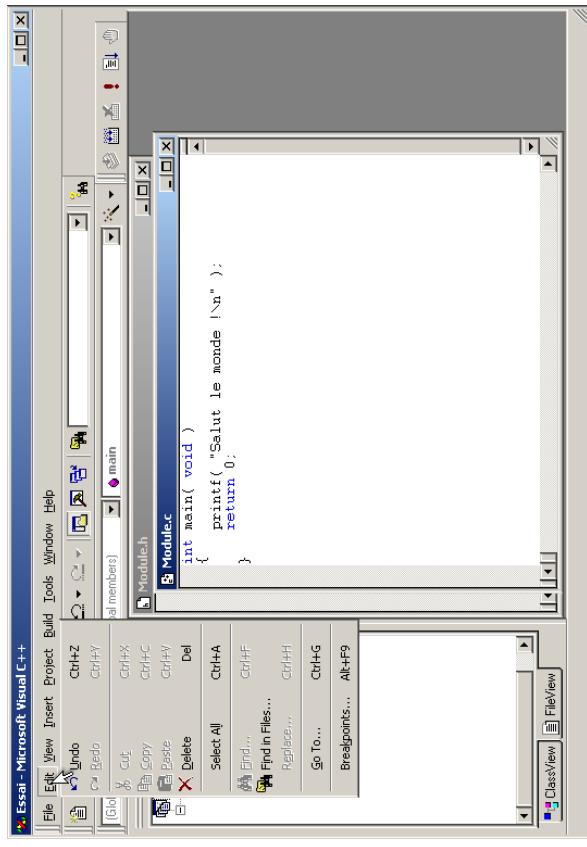
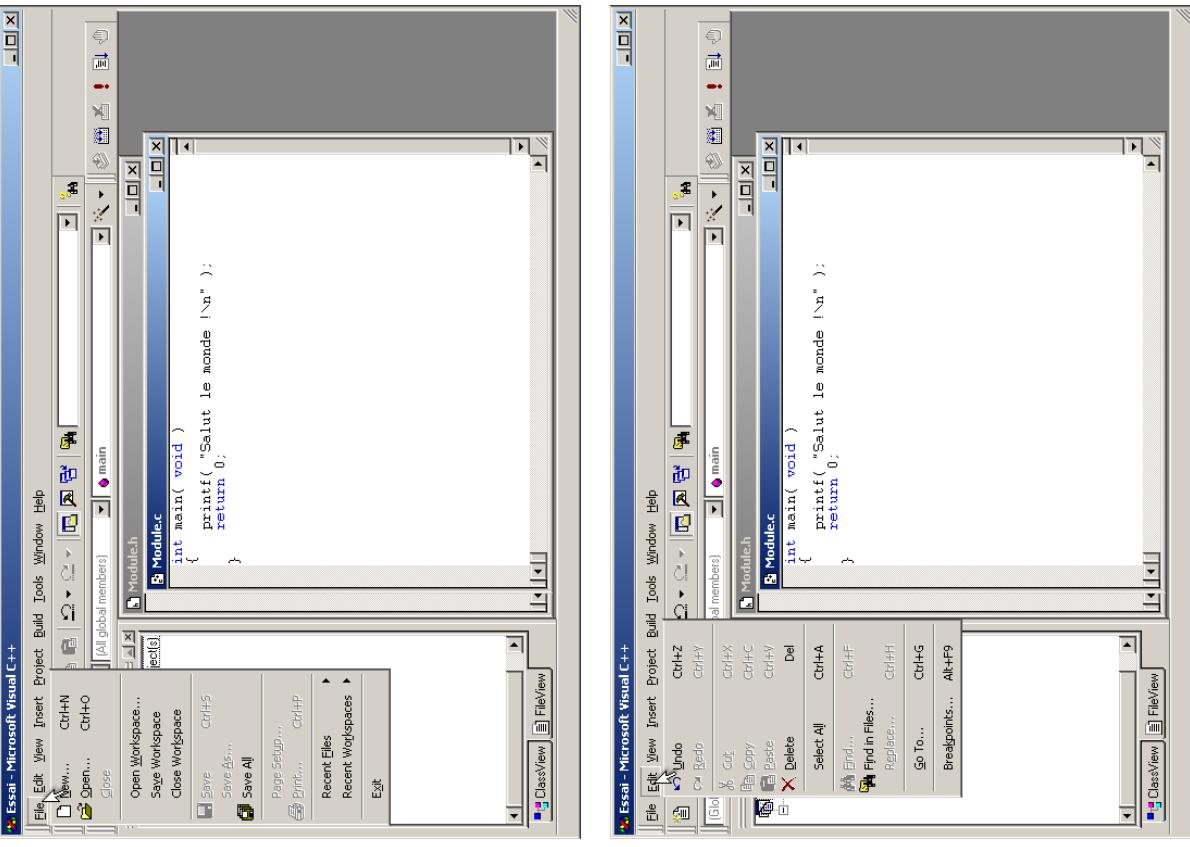


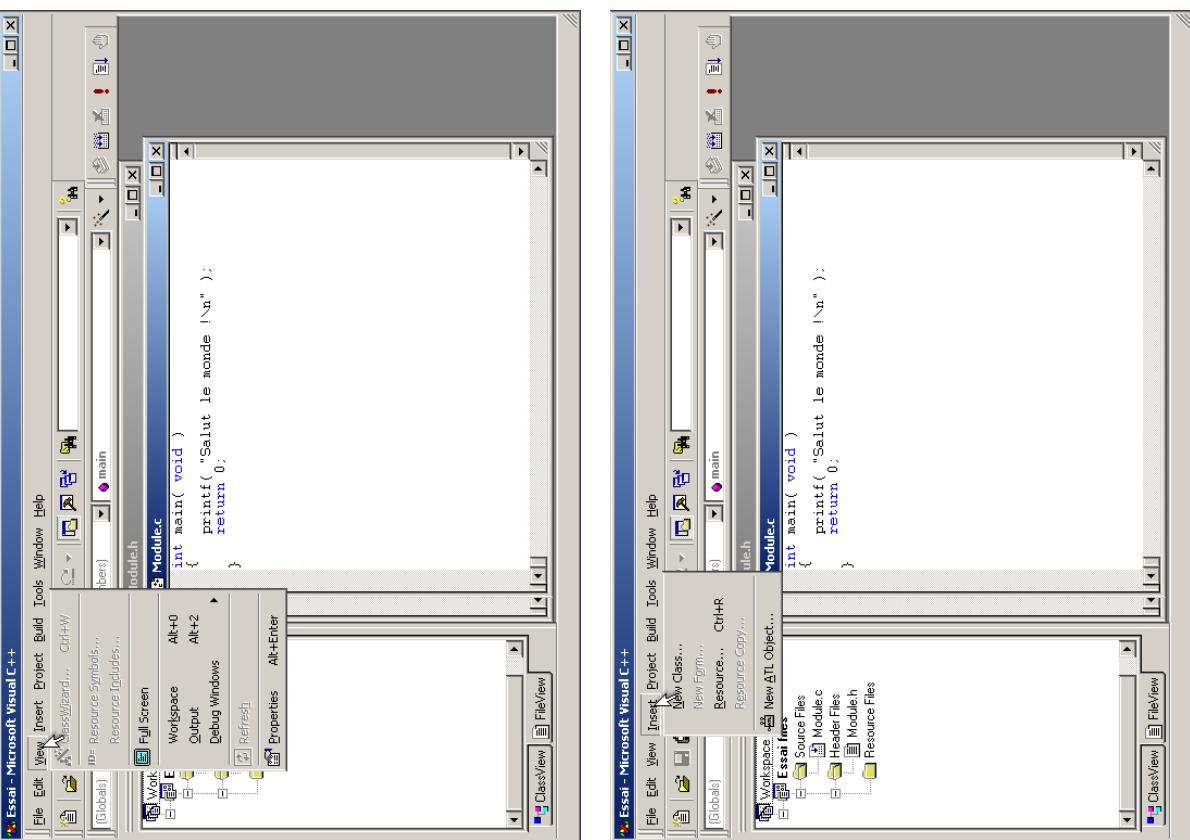
17

18



### 4.1.2 Les menus

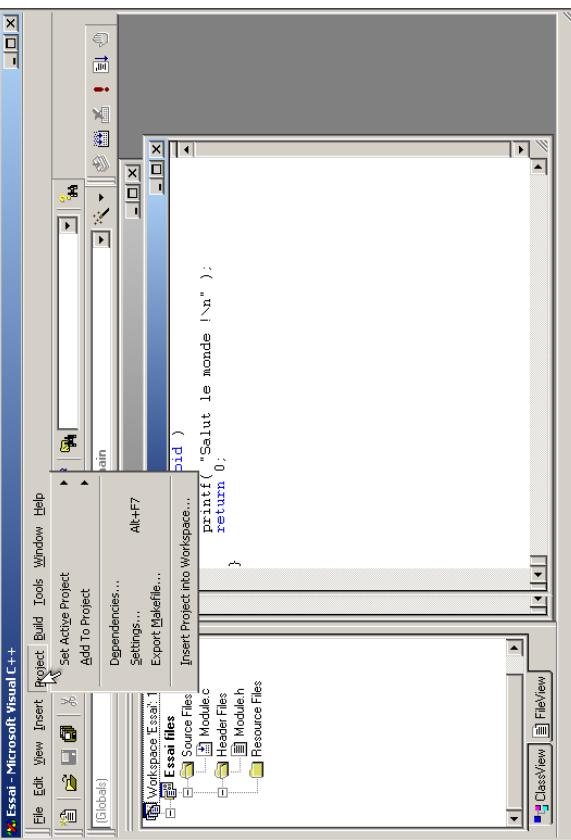




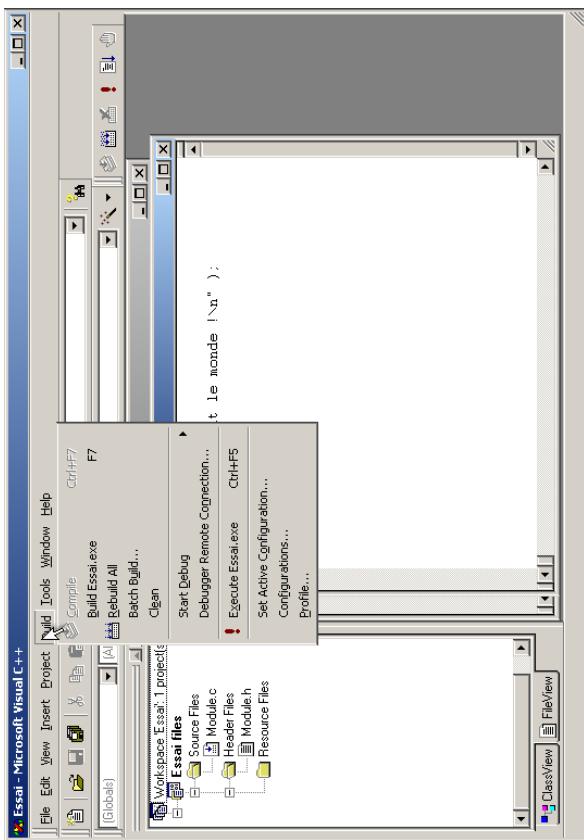
3



4

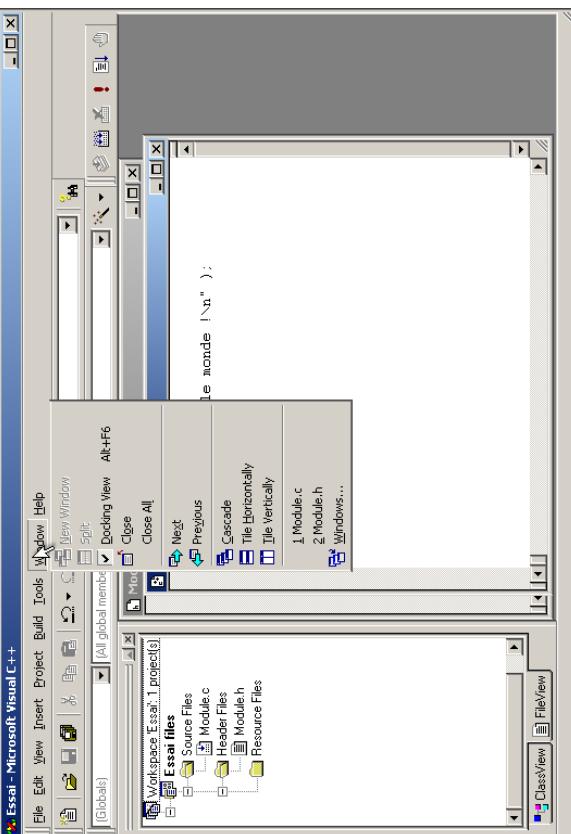


5

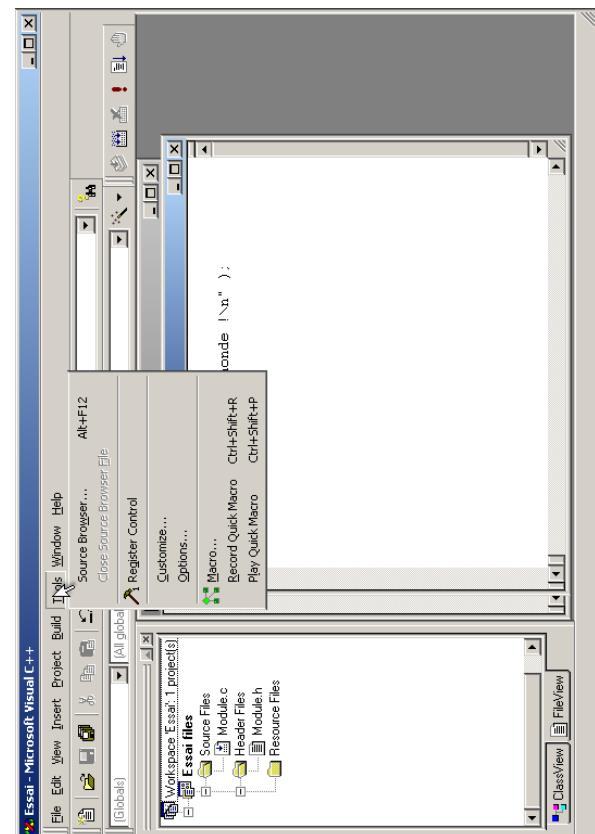


6

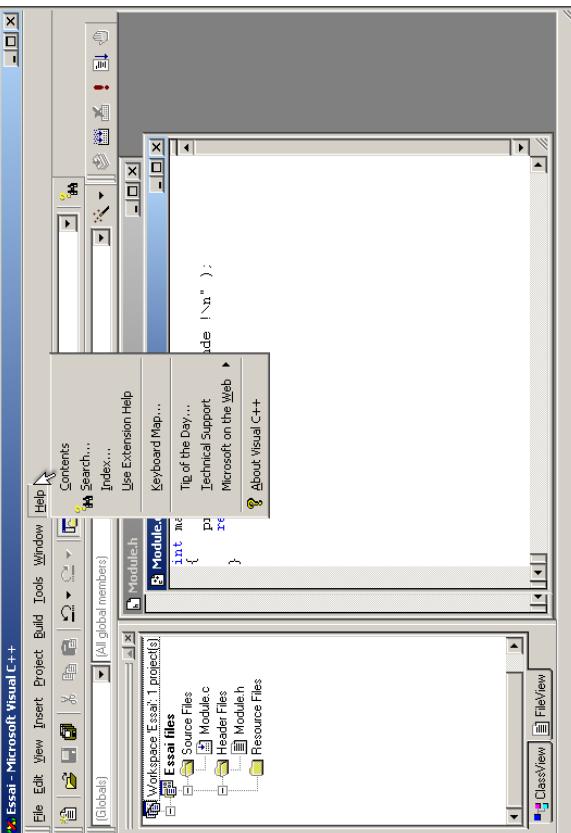
## 9 Le C en pratique



7

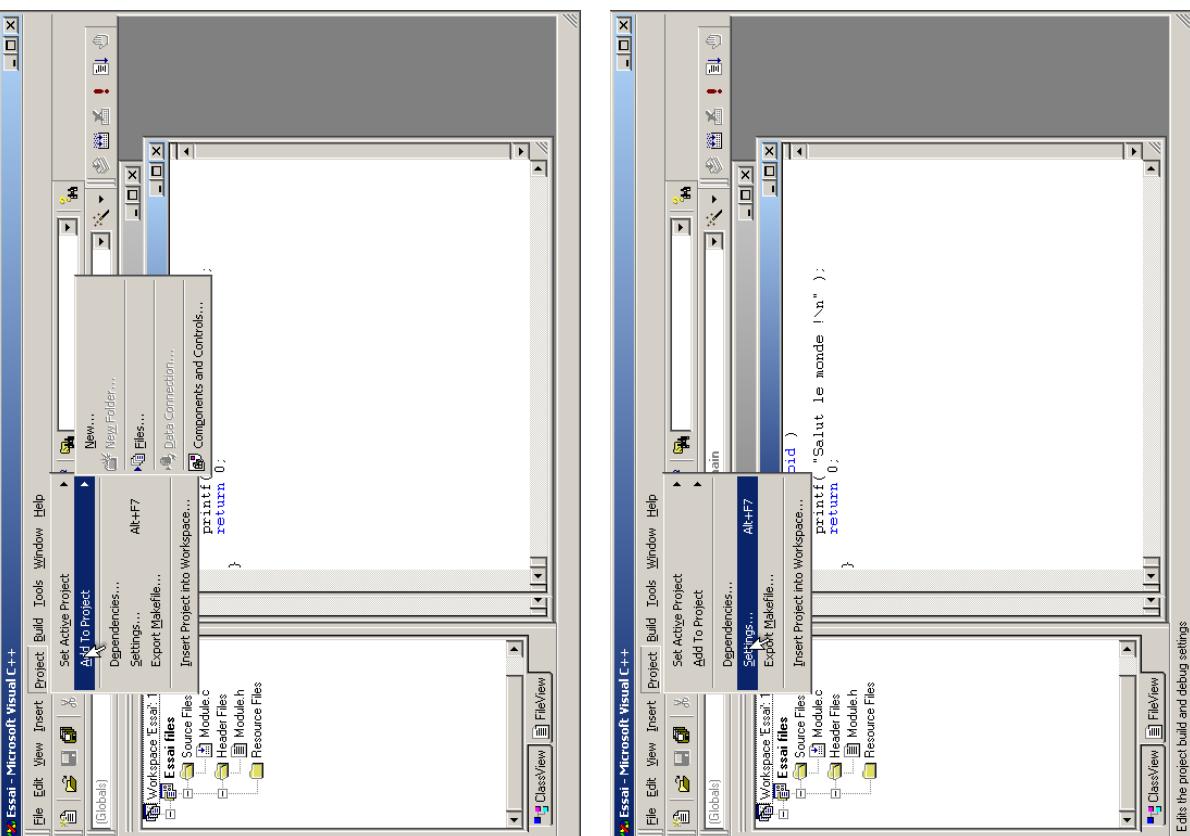


8

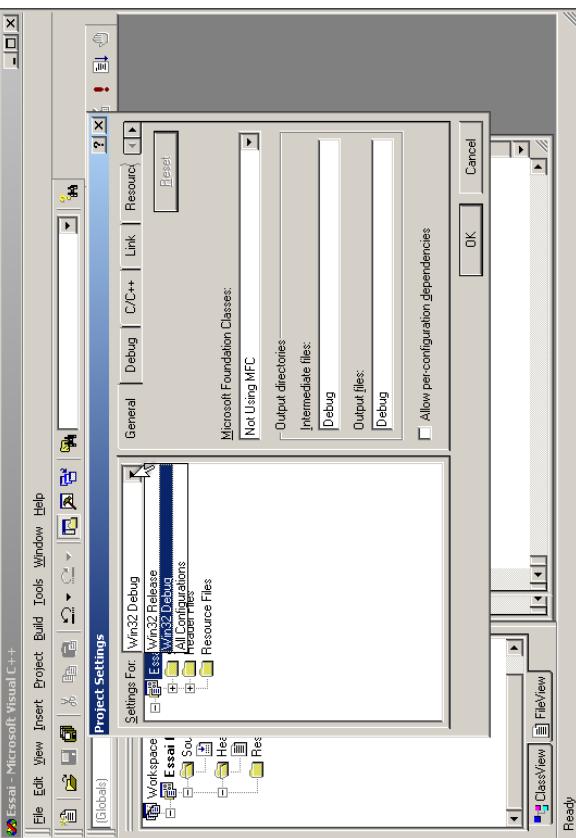


9

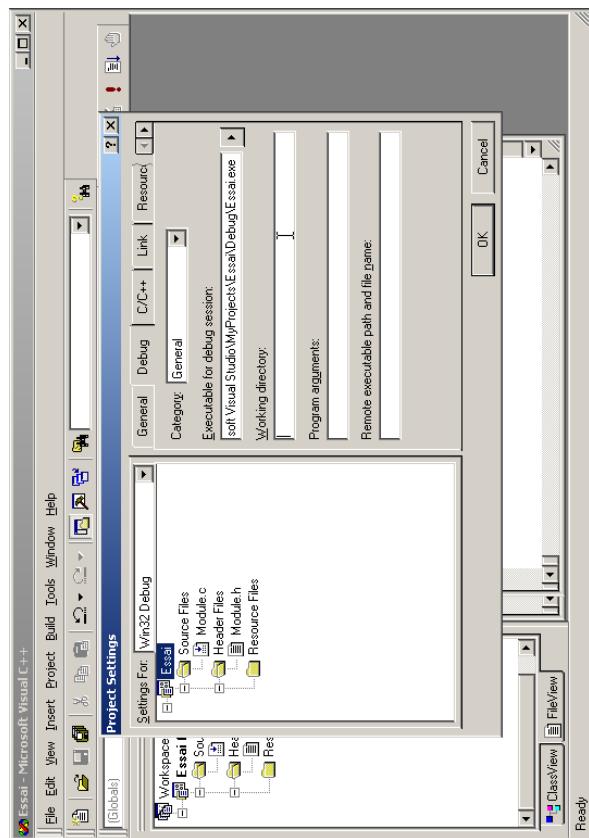
### 4.1.3 Le projet



1

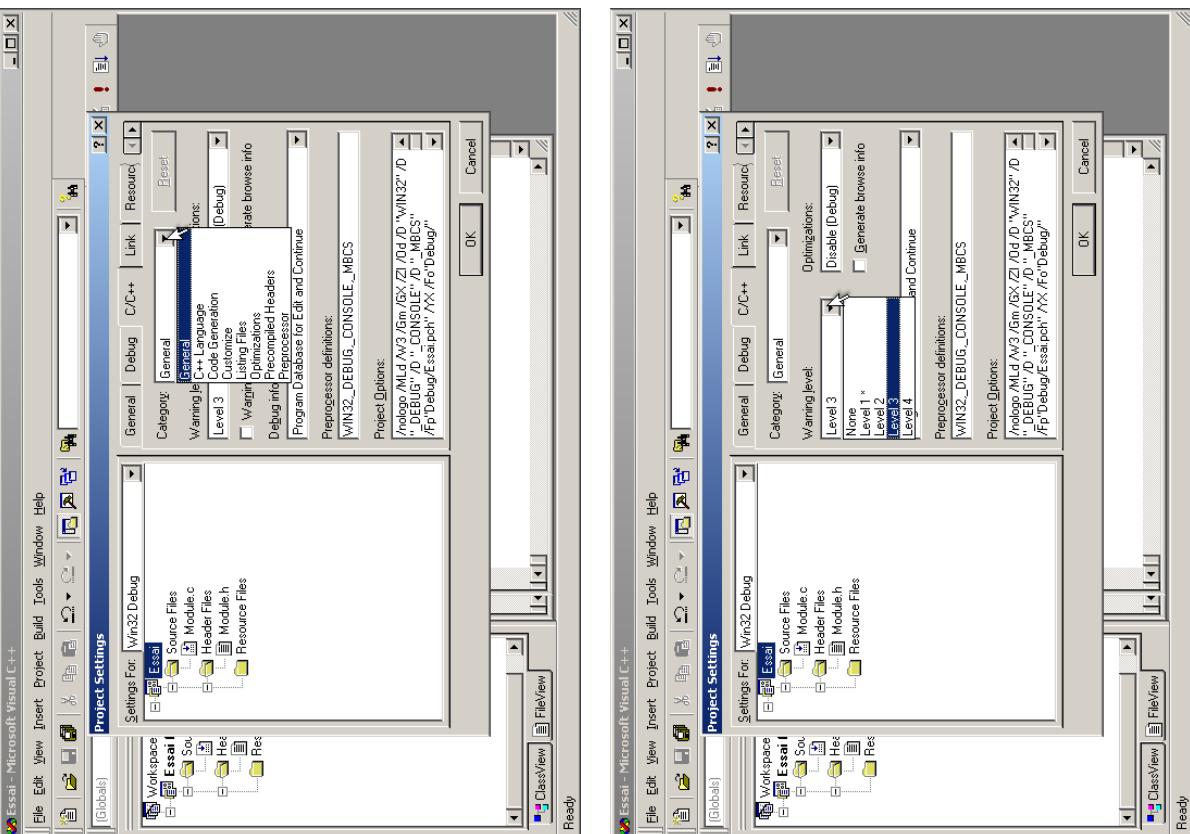


3

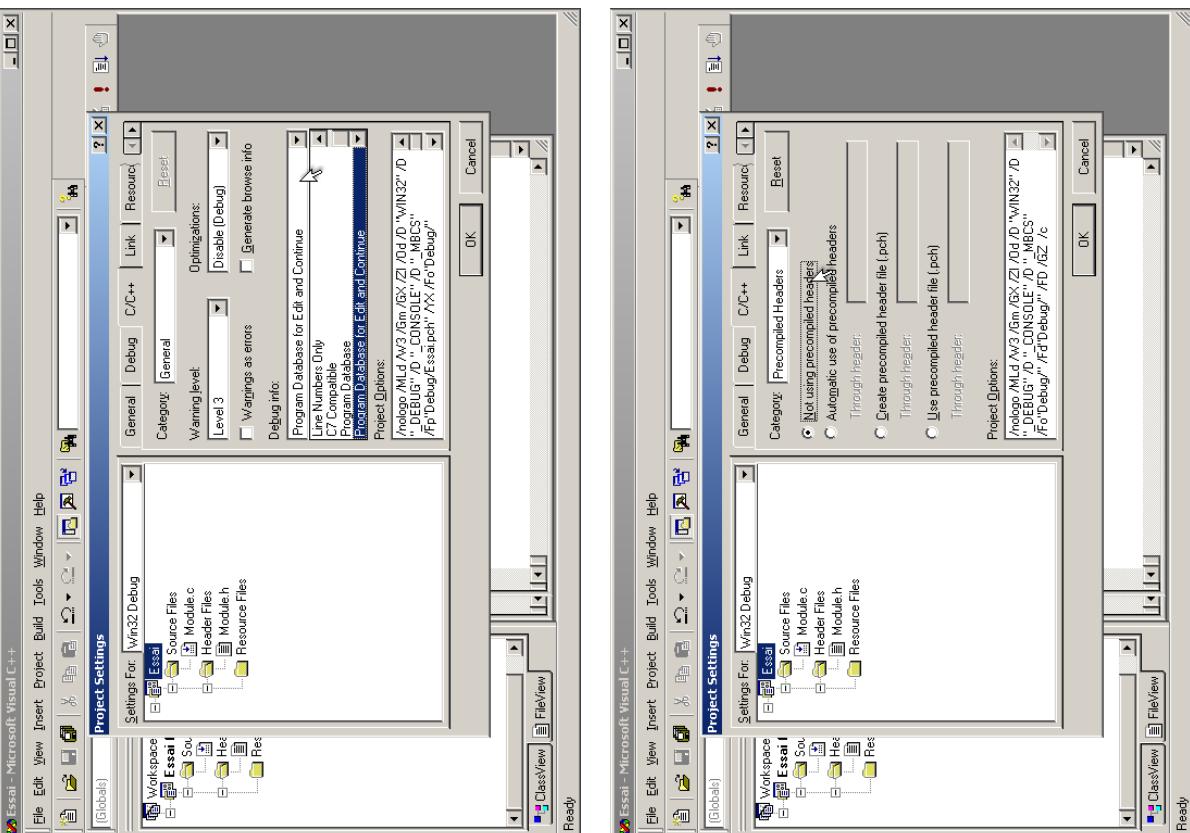


4

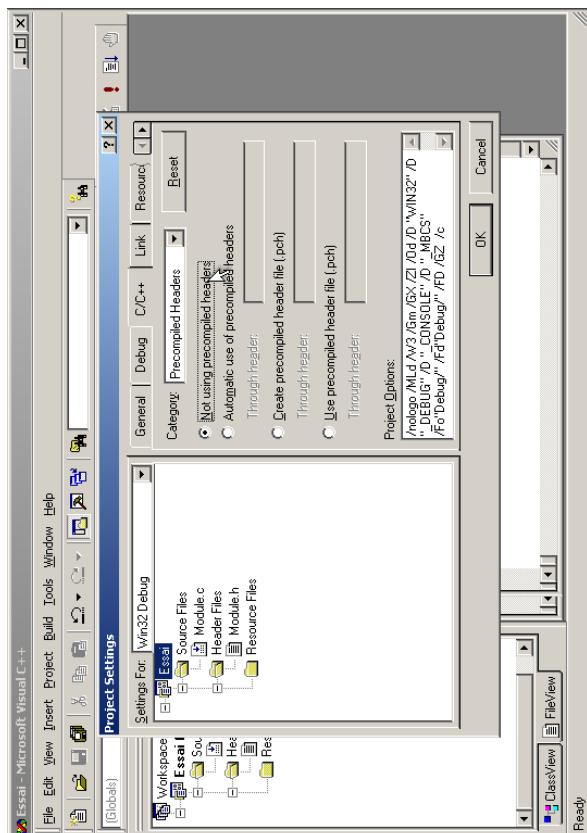
## 9 Le C en pratique



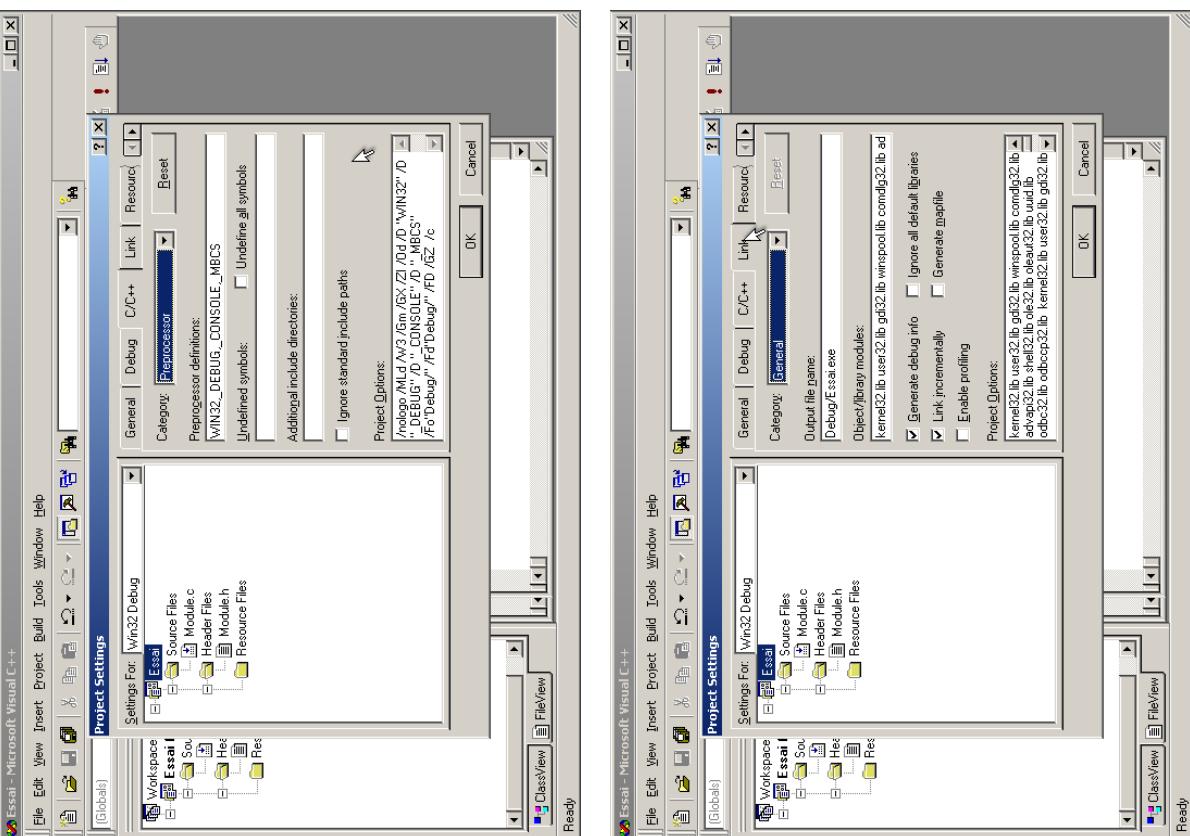
5



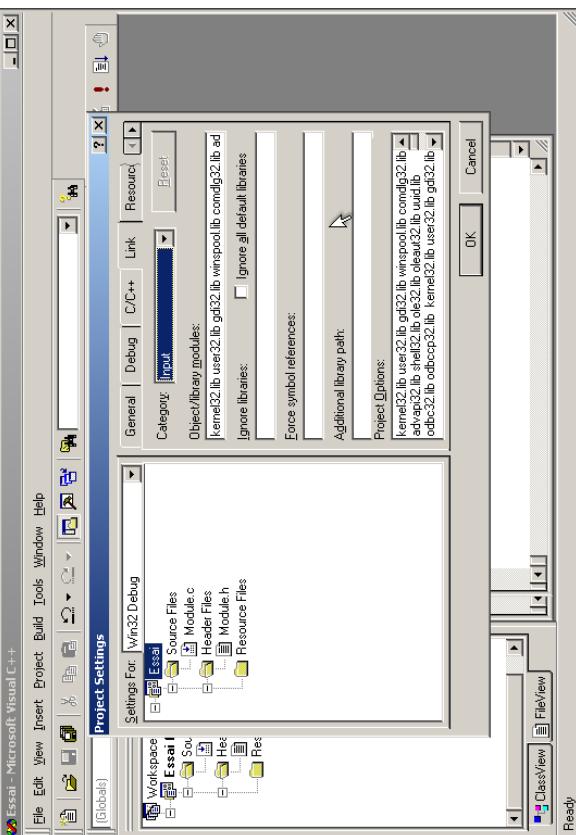
7



8



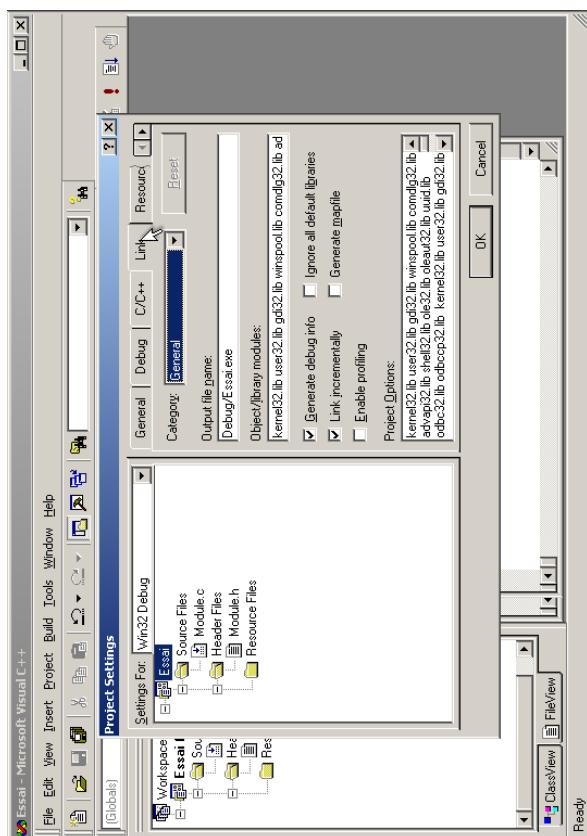
9

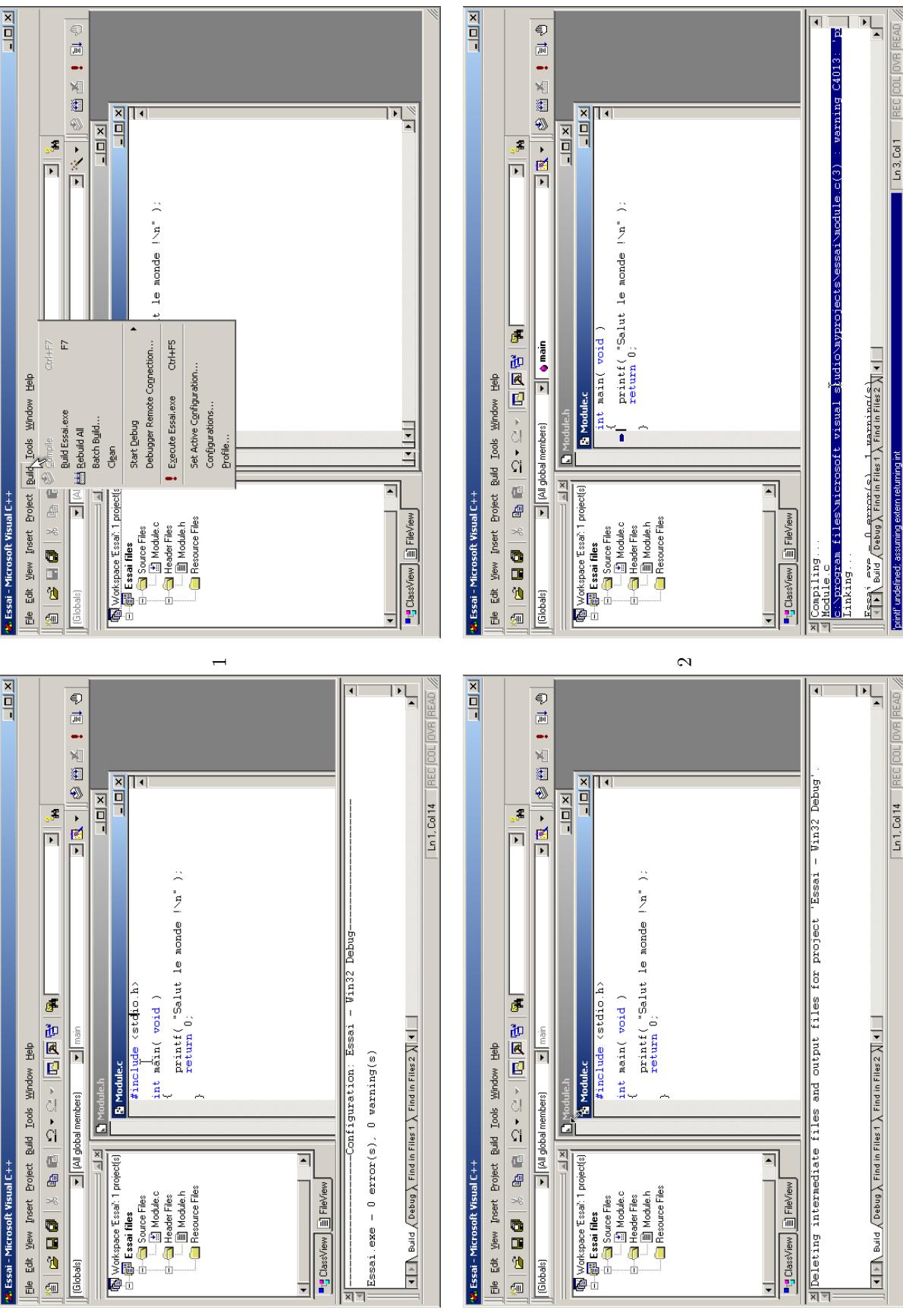


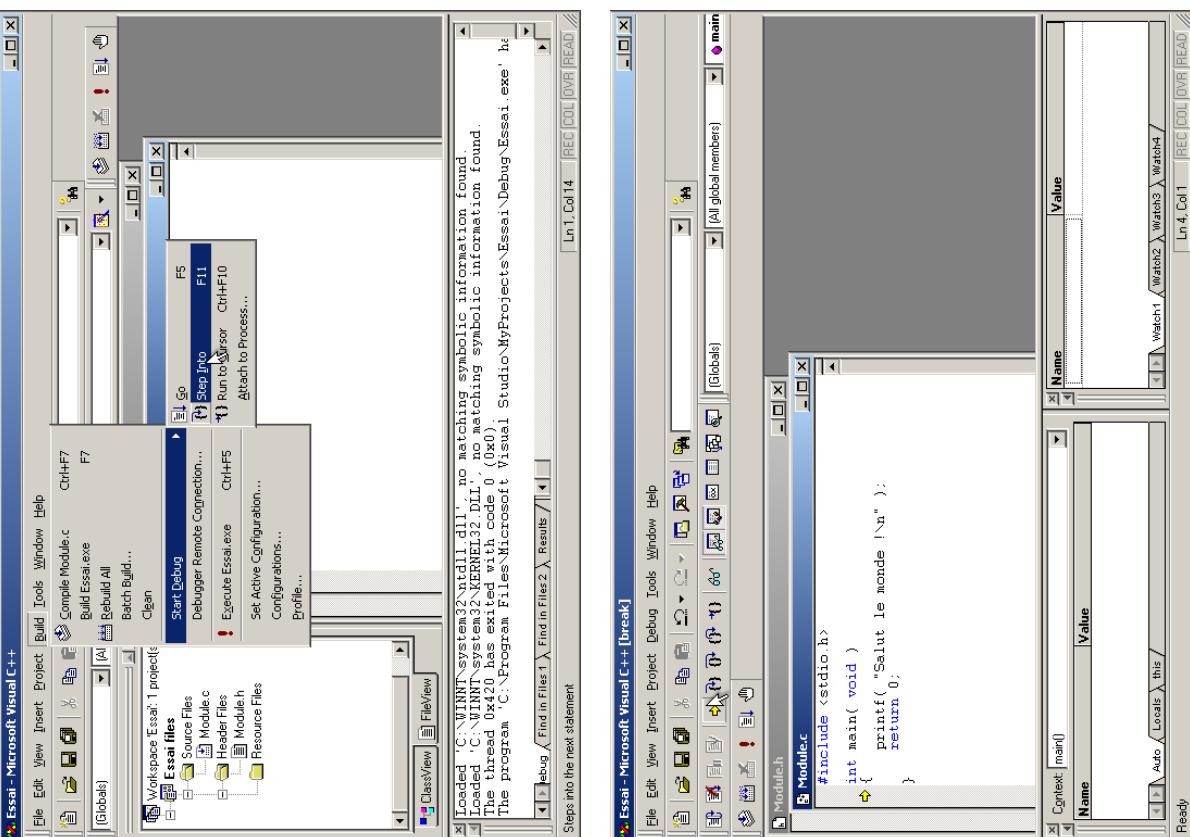
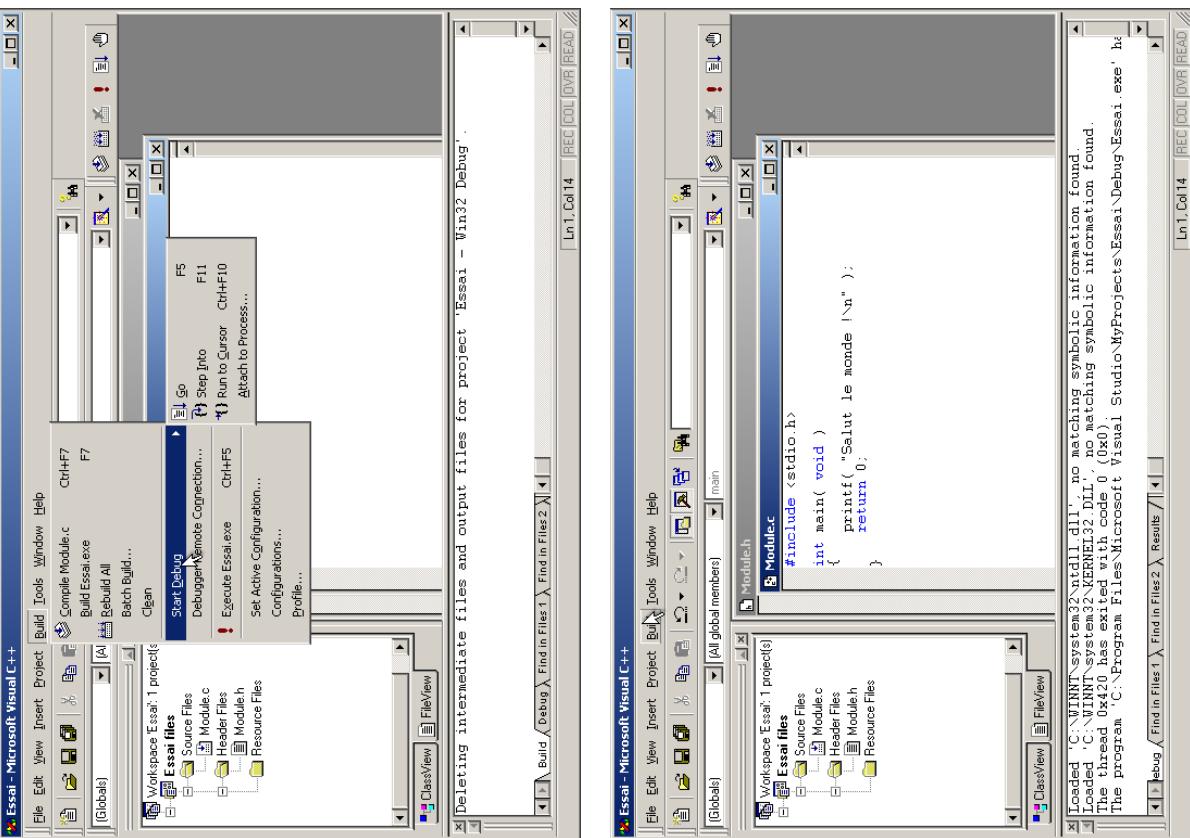
11

#### 4.1.4 Construire et debugger un projet

10







## 9 Le C en pratique

9

Code:

```
#include <stdio.h>
int main(void)
{
 printf("Salut le monde !\n");
 return 0;
}
```

Locals window:

| Name                               | Value |
|------------------------------------|-------|
| Auto / Locals / this /             |       |
| Watch / Watch2 / Watch3 / Watch4 / |       |

10

Registers window:

| Name | Value              |
|------|--------------------|
| rax  | 0x0000000000401000 |
| rbx  | 0x0000000000401000 |
| rcx  | 0x0000000000401000 |
| rdx  | 0x0000000000401000 |
| rsi  | 0x0000000000401000 |
| rdi  | 0x0000000000401000 |
| rbp  | 0x0000000000401000 |
| rsp  | 0x0000000000401000 |
| r8   | 0x0000000000401000 |
| r9   | 0x0000000000401000 |
| r10  | 0x0000000000401000 |
| r11  | 0x0000000000401000 |
| r12  | 0x0000000000401000 |
| r13  | 0x0000000000401000 |
| r14  | 0x0000000000401000 |
| r15  | 0x0000000000401000 |
| rip  | 0x0000000000401000 |
| cs   | 0x0000000000401000 |
| ss   | 0x0000000000401000 |
| ds   | 0x0000000000401000 |
| fs   | 0x0000000000401000 |
| gs   | 0x0000000000401000 |

11

Registers window:

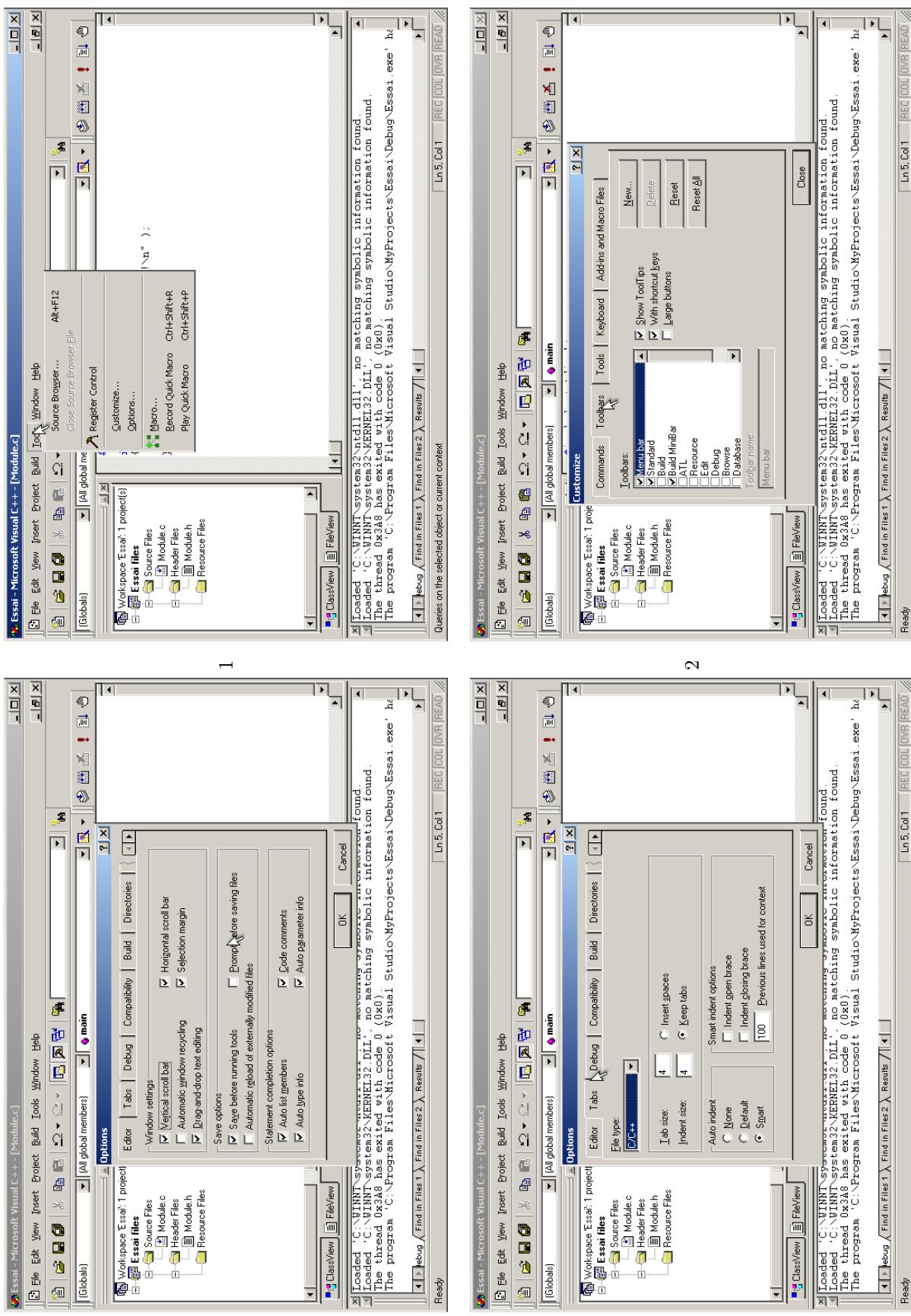
| Name | Value              |
|------|--------------------|
| rax  | 0x0000000000401000 |
| rbx  | 0x0000000000401000 |
| rcx  | 0x0000000000401000 |
| rdx  | 0x0000000000401000 |
| rsi  | 0x0000000000401000 |
| rdi  | 0x0000000000401000 |
| rbp  | 0x0000000000401000 |
| rsp  | 0x0000000000401000 |
| r8   | 0x0000000000401000 |
| r9   | 0x0000000000401000 |
| r10  | 0x0000000000401000 |
| r11  | 0x0000000000401000 |
| r12  | 0x0000000000401000 |
| r13  | 0x0000000000401000 |
| r14  | 0x0000000000401000 |
| r15  | 0x0000000000401000 |
| rip  | 0x0000000000401000 |
| cs   | 0x0000000000401000 |
| ss   | 0x0000000000401000 |
| ds   | 0x0000000000401000 |
| fs   | 0x0000000000401000 |
| gs   | 0x0000000000401000 |

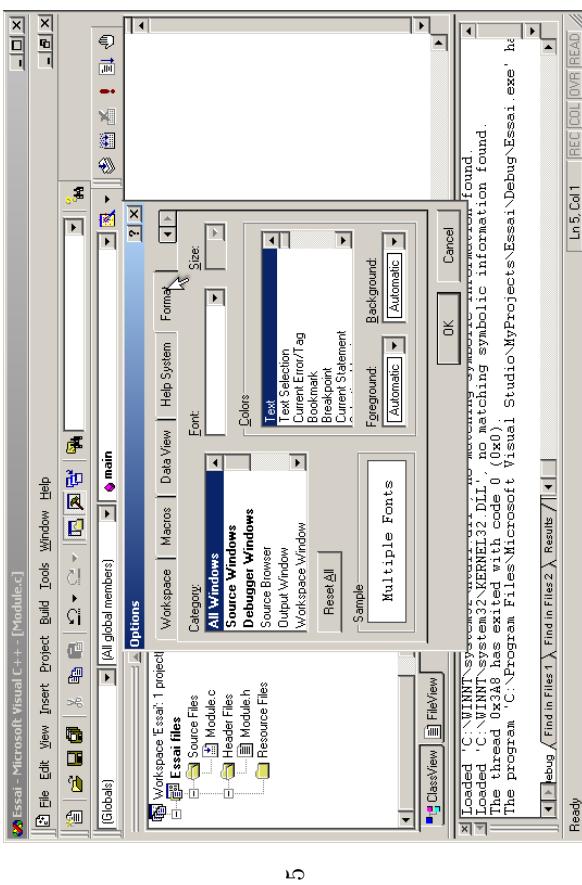
12

Registers window:

| Name | Value              |
|------|--------------------|
| rax  | 0x0000000000401000 |
| rbx  | 0x0000000000401000 |
| rcx  | 0x0000000000401000 |
| rdx  | 0x0000000000401000 |
| rsi  | 0x0000000000401000 |
| rdi  | 0x0000000000401000 |
| rbp  | 0x0000000000401000 |
| rsp  | 0x0000000000401000 |
| r8   | 0x0000000000401000 |
| r9   | 0x0000000000401000 |
| r10  | 0x0000000000401000 |
| r11  | 0x0000000000401000 |
| r12  | 0x0000000000401000 |
| r13  | 0x0000000000401000 |
| r14  | 0x0000000000401000 |
| r15  | 0x0000000000401000 |
| rip  | 0x0000000000401000 |
| cs   | 0x0000000000401000 |
| ss   | 0x0000000000401000 |
| ds   | 0x0000000000401000 |
| fs   | 0x0000000000401000 |
| gs   | 0x0000000000401000 |

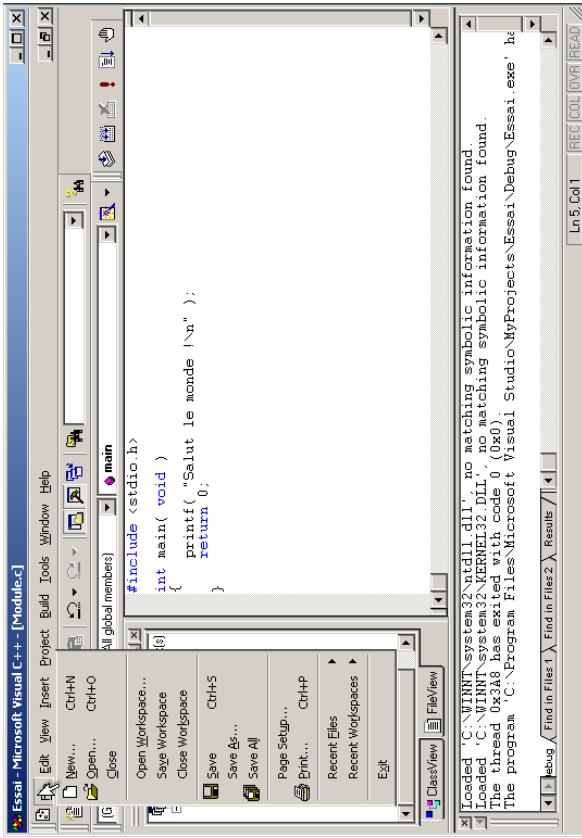
### 4.1.5 Configuration de l'IDE



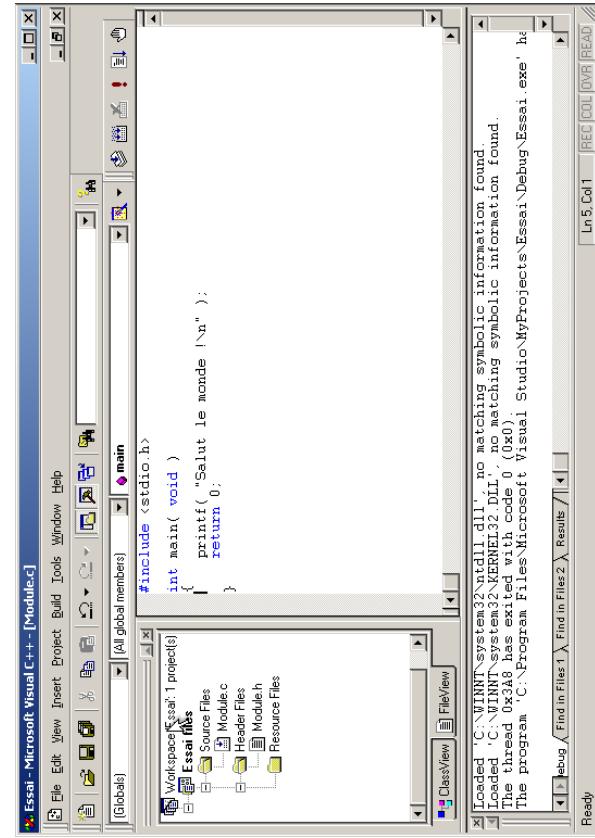


5

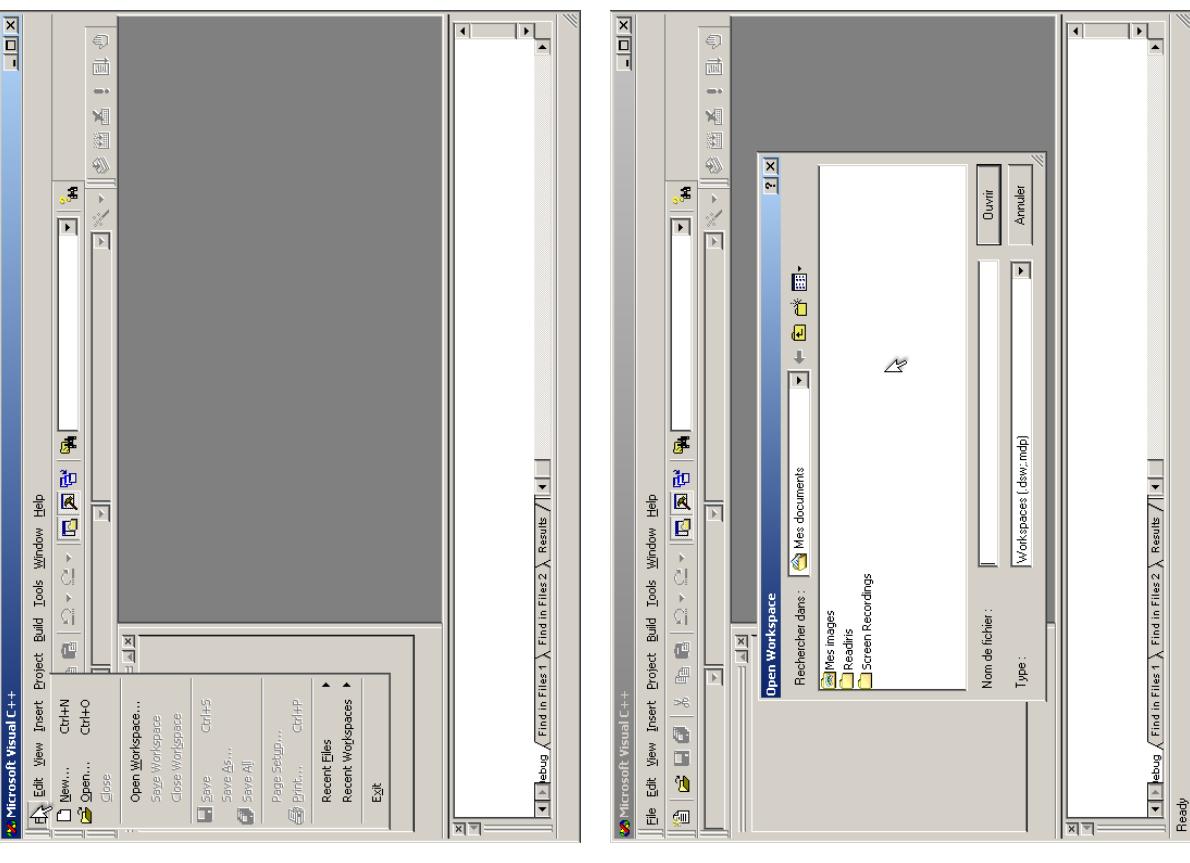
### 4.1.6 Ouverture d'un projet



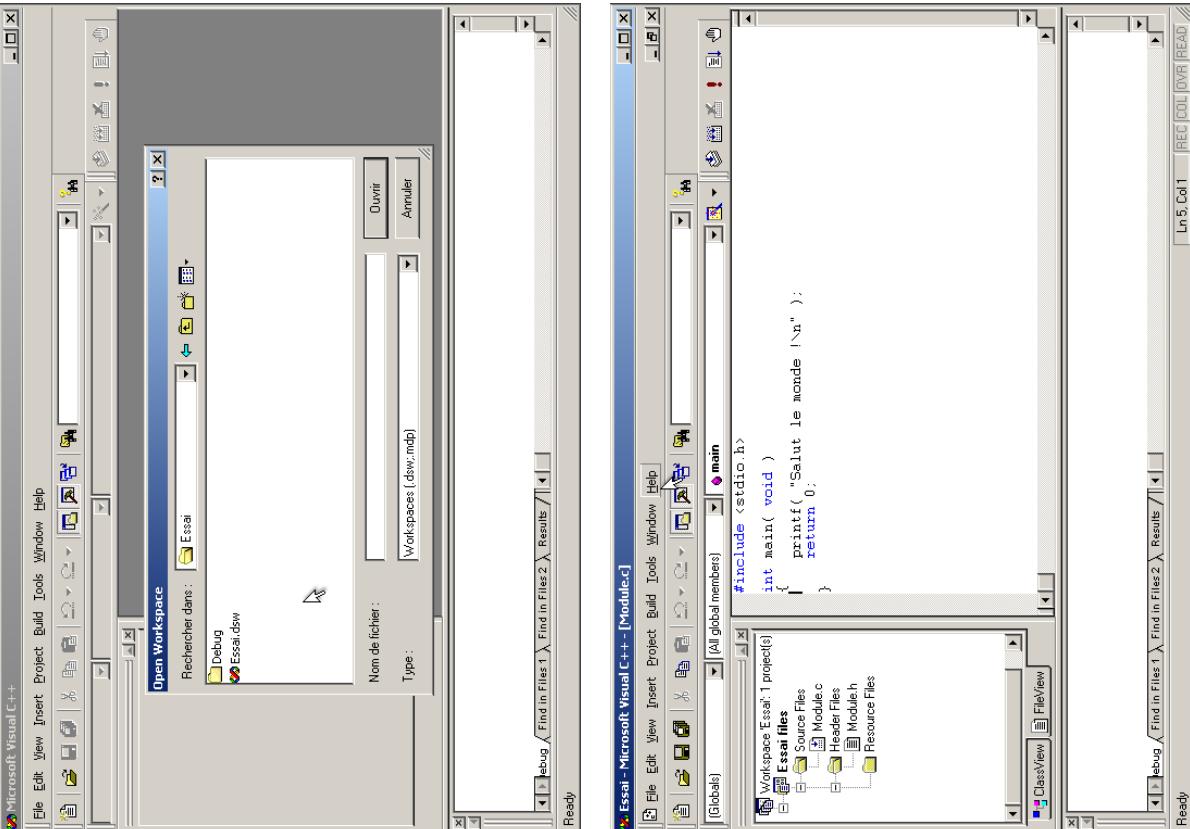
1



2



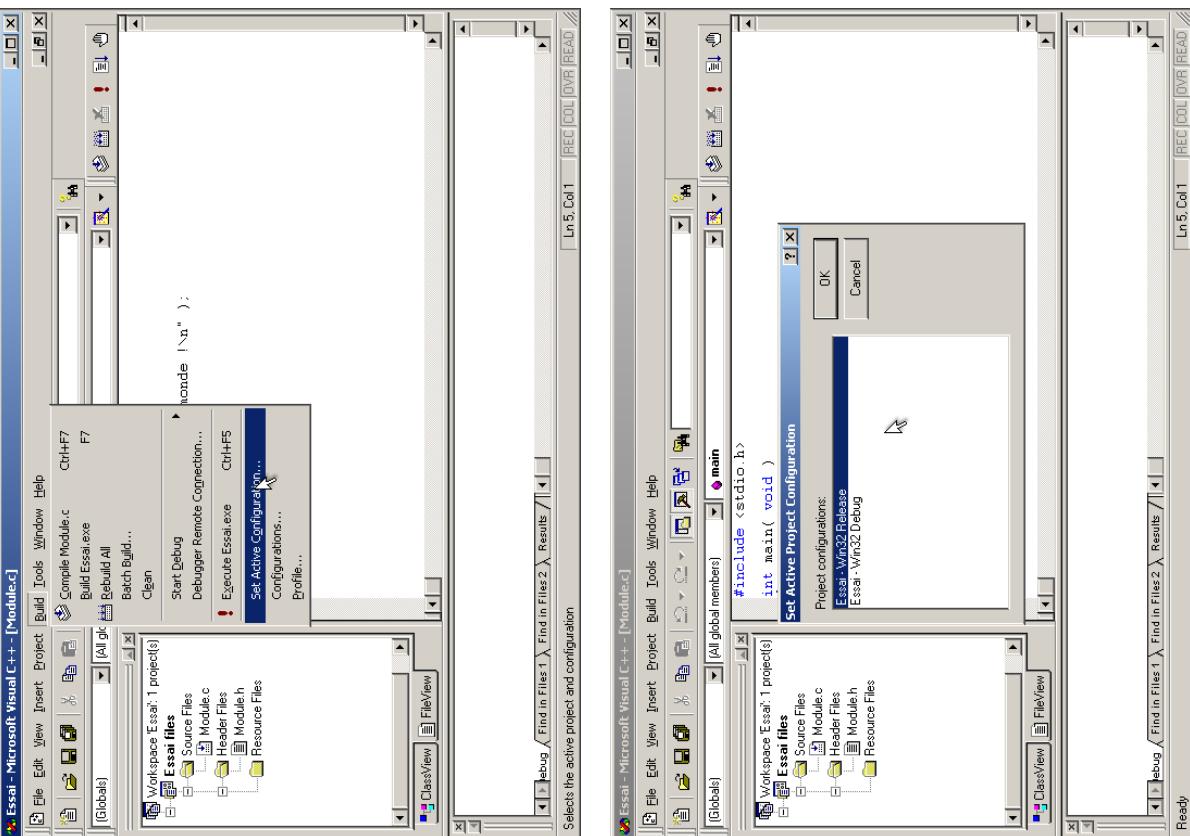
#### 4.1.7 Configurations du projet et fichiers produits



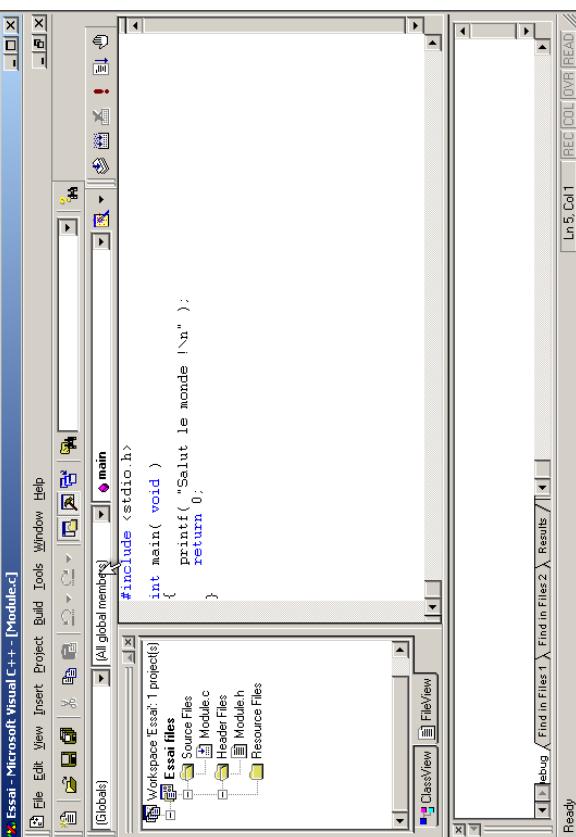
1

2

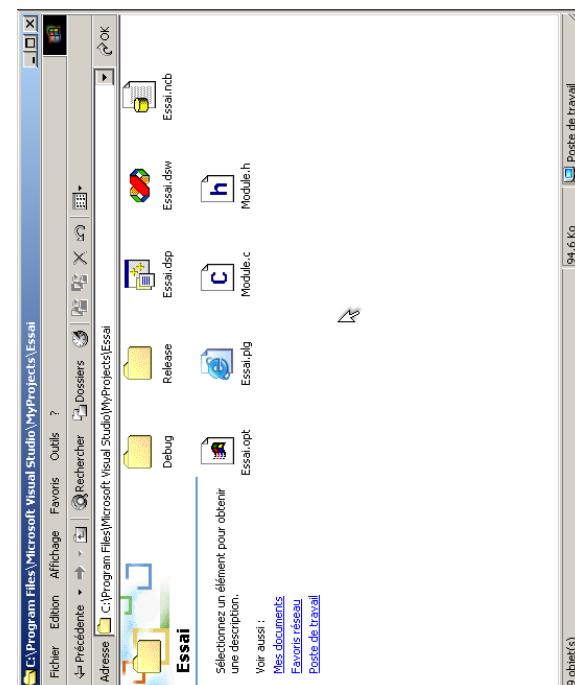
## 9 Le C en pratique



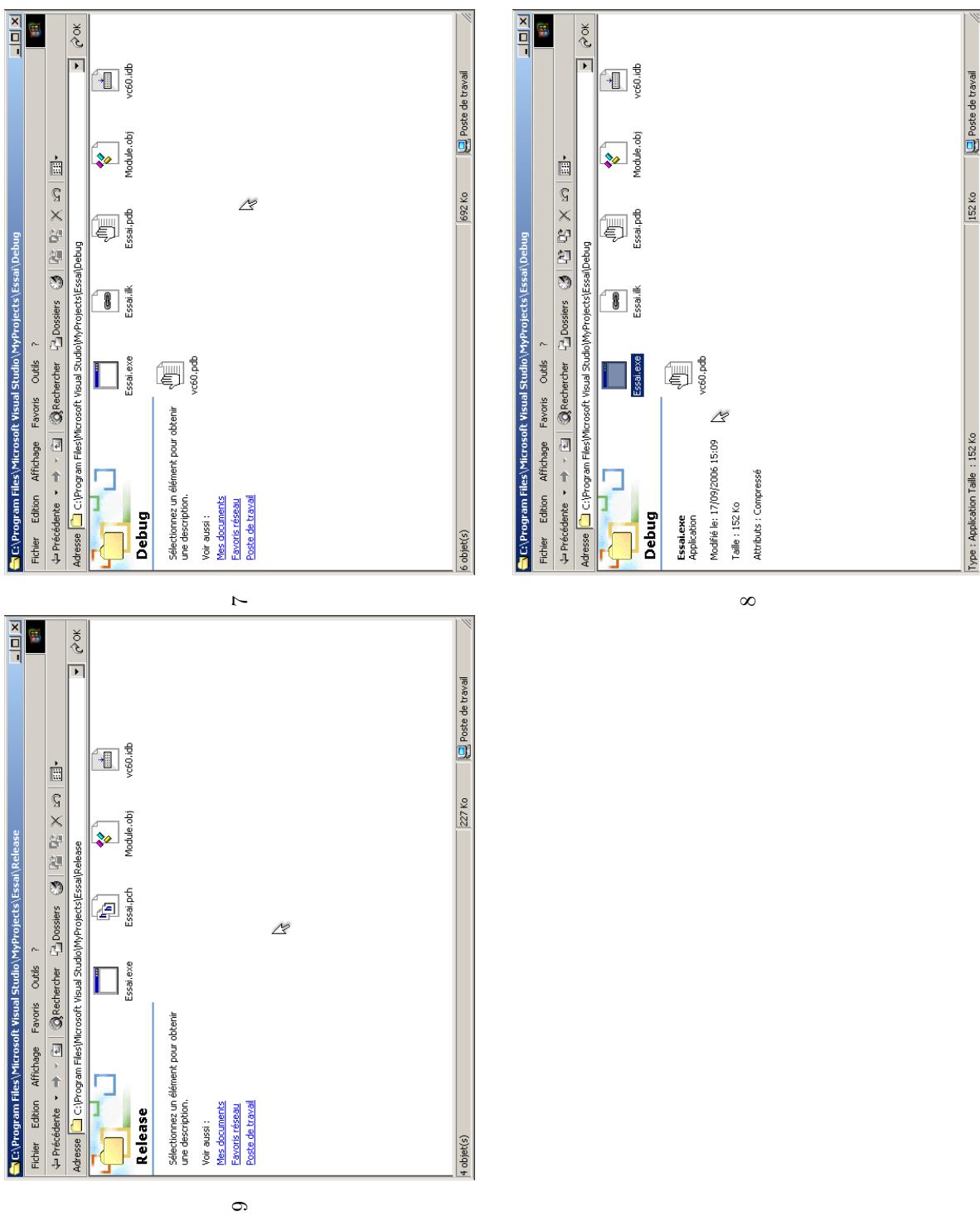
3



5



6



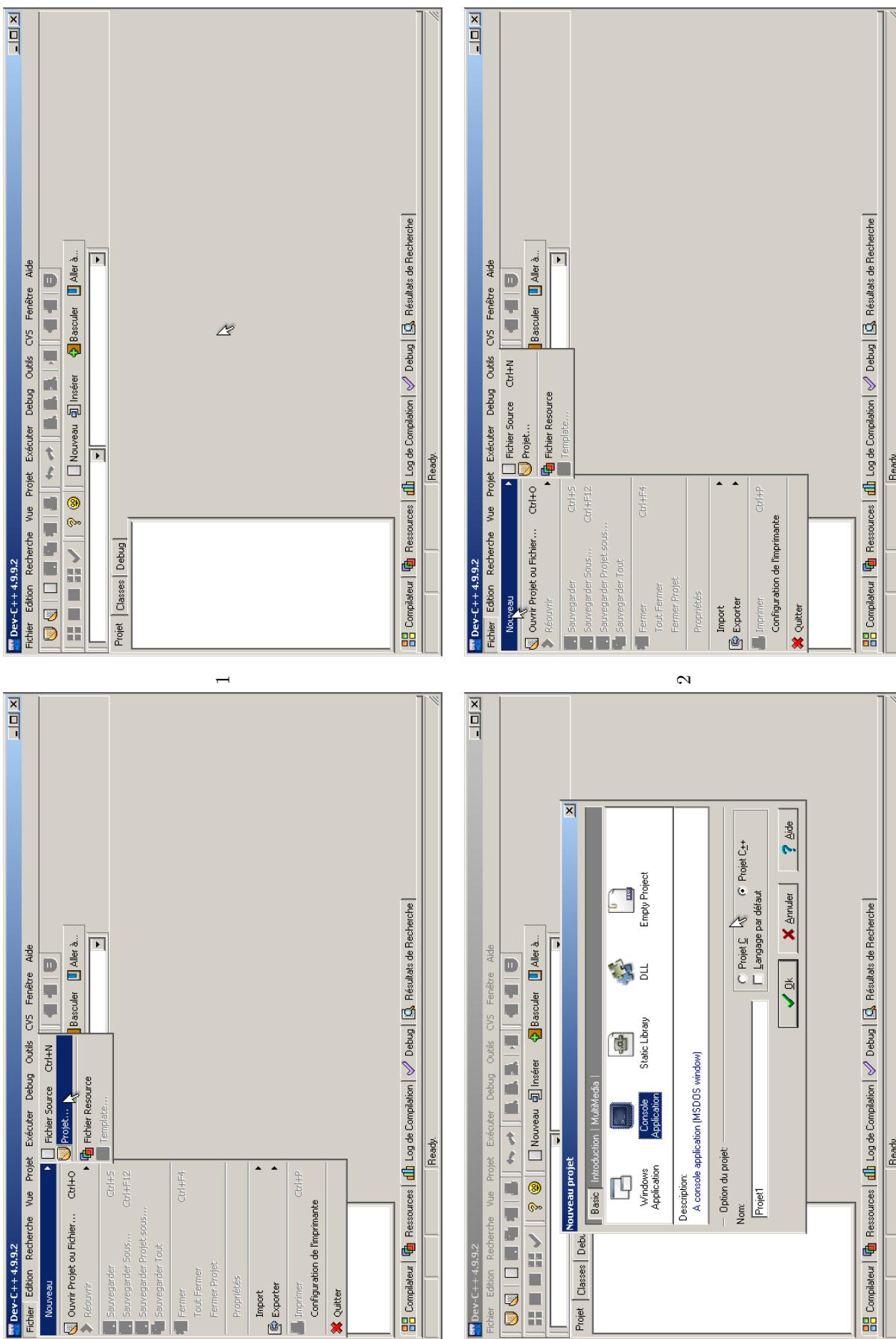
## 4.2 Visual Studio .Net

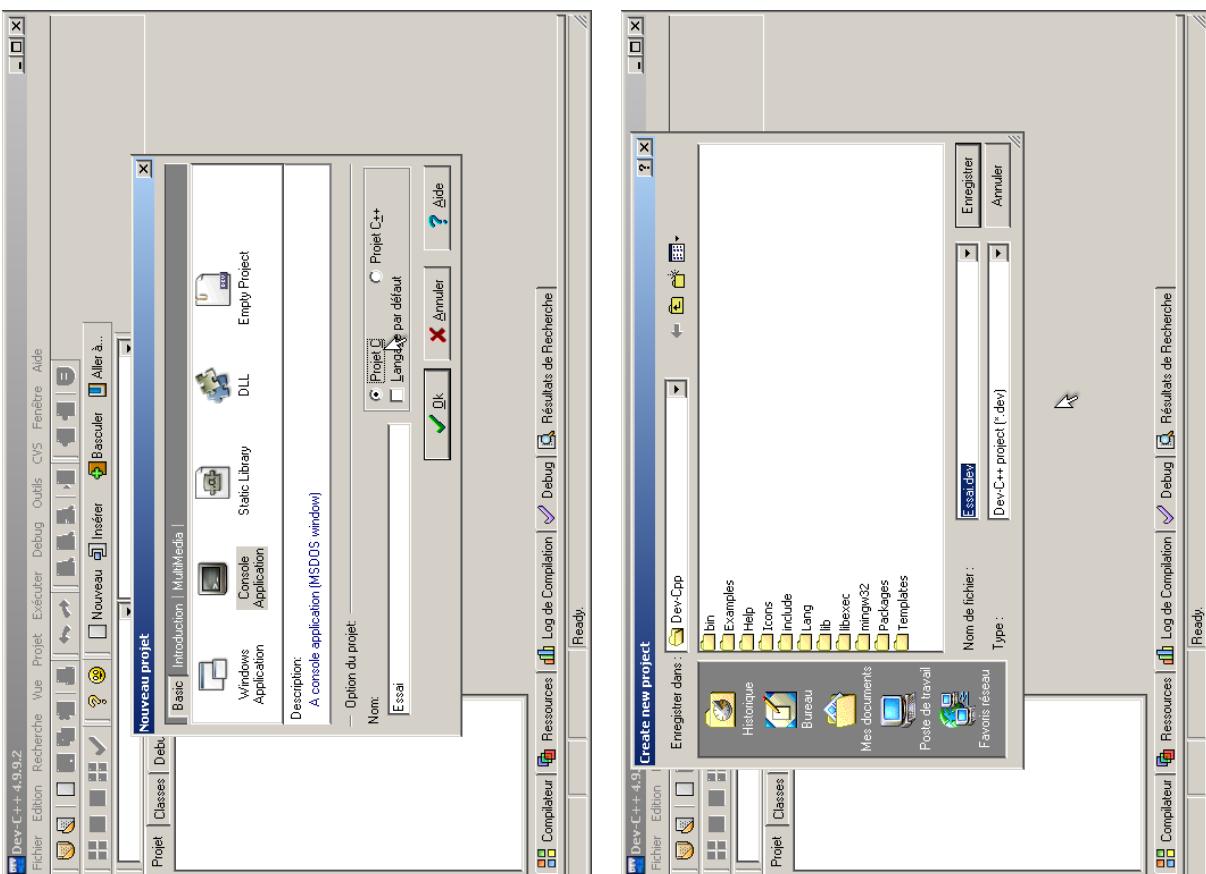
Similaire à VS6

## 4.3 Dev C++

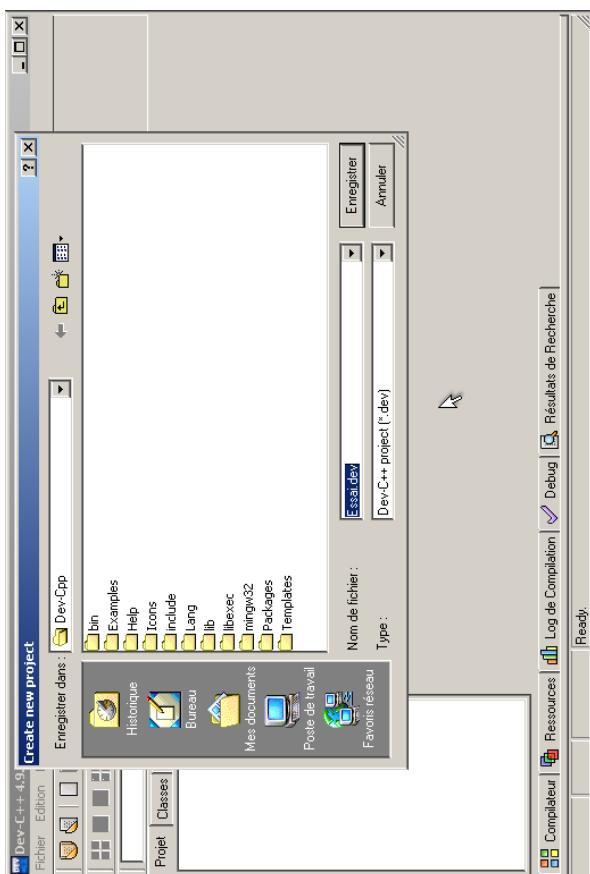
### 4.3.1 Création du projet et des fichiers

## 9 Le C en pratique

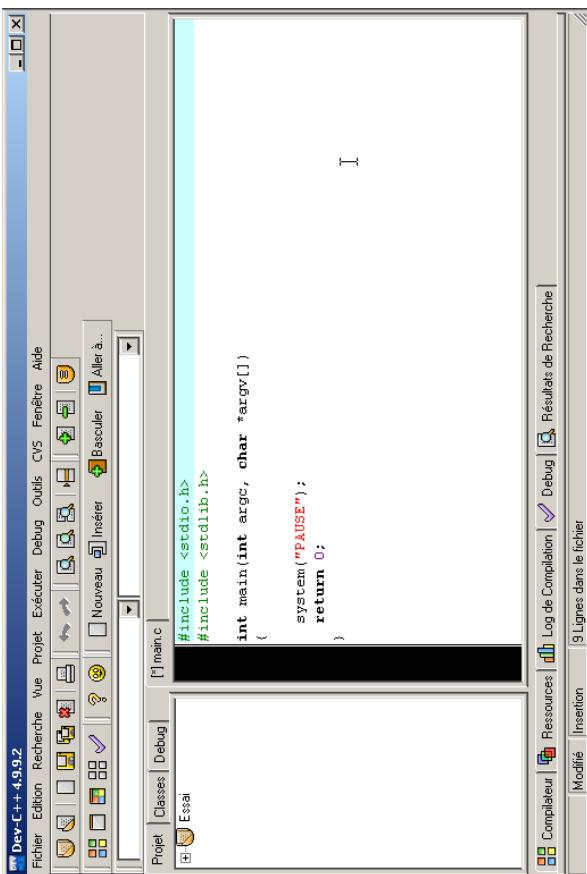




5



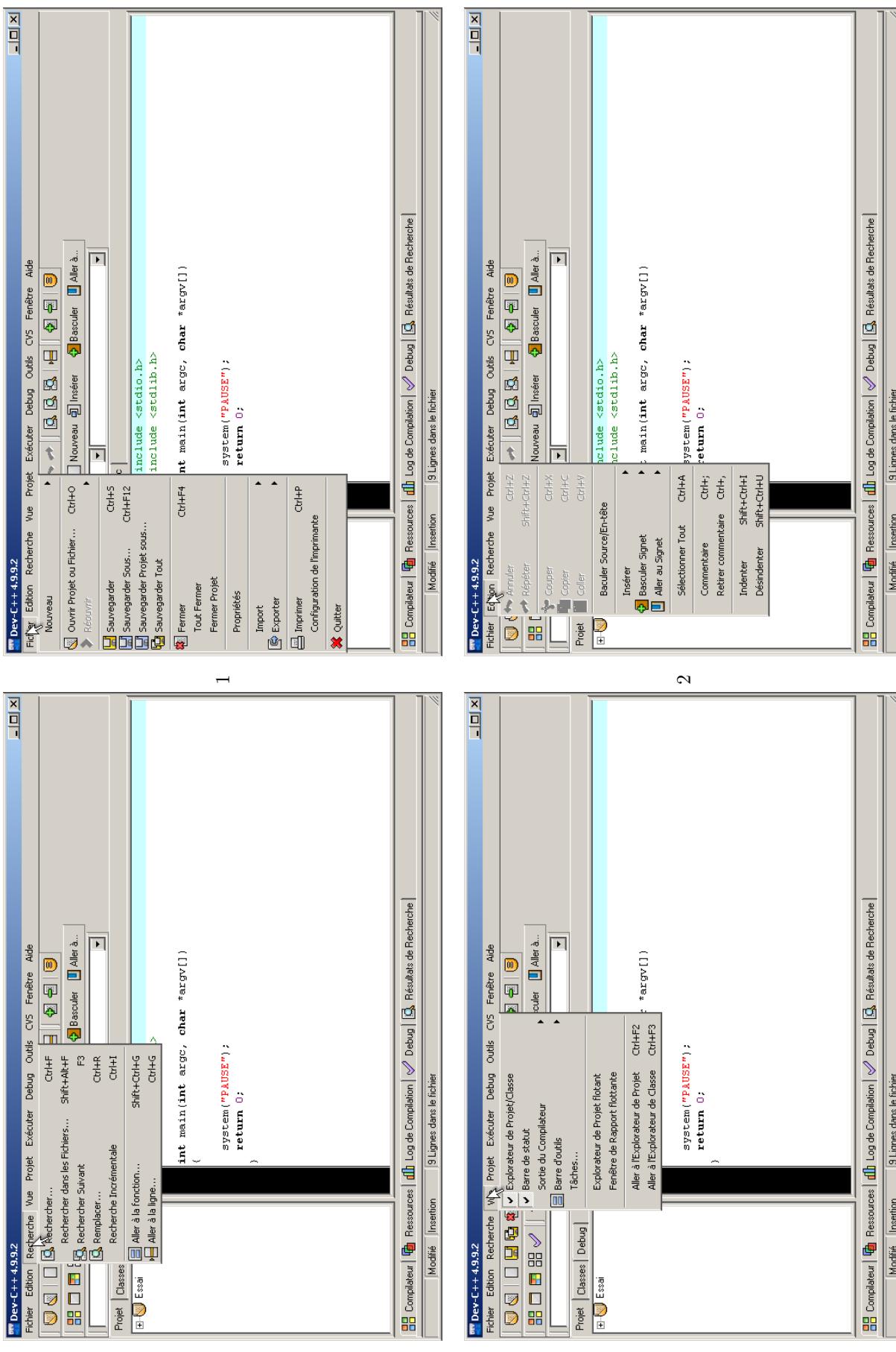
6

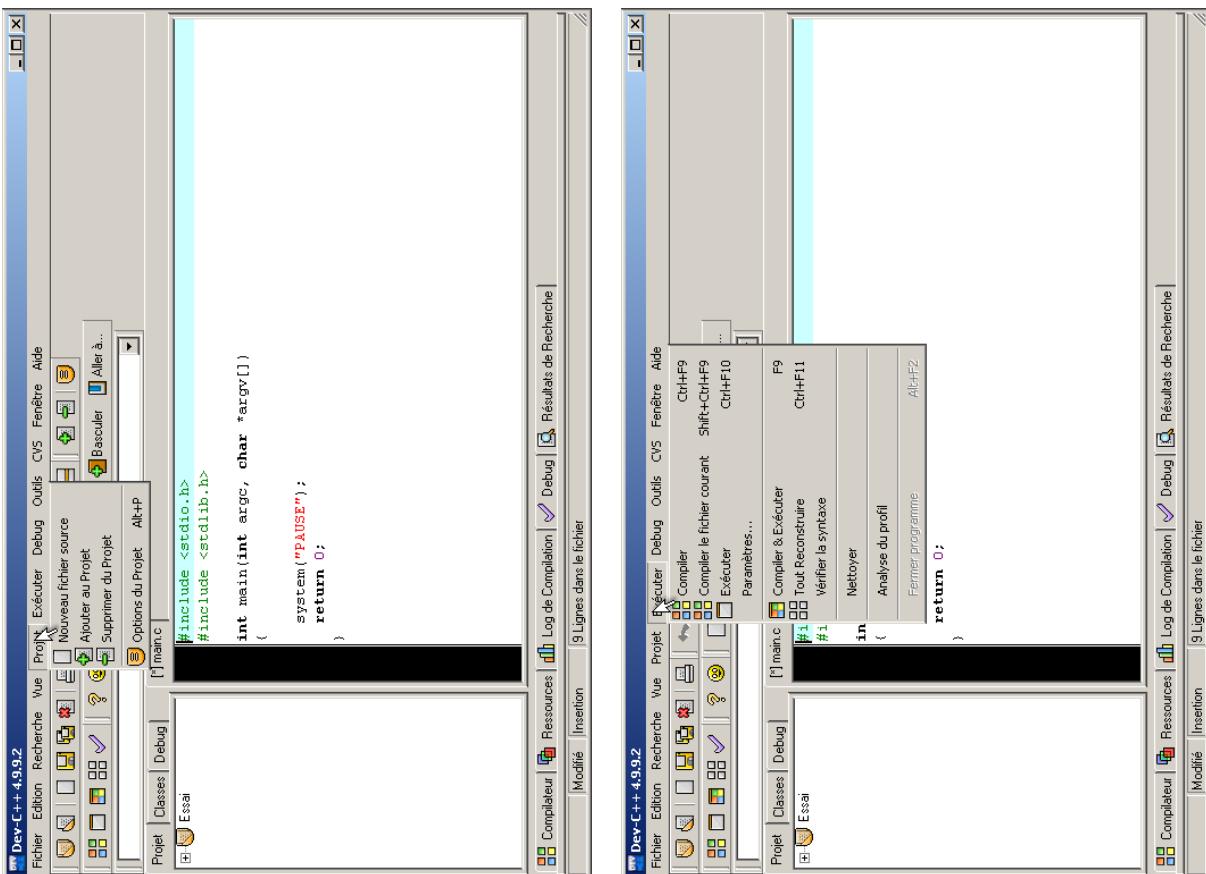


7

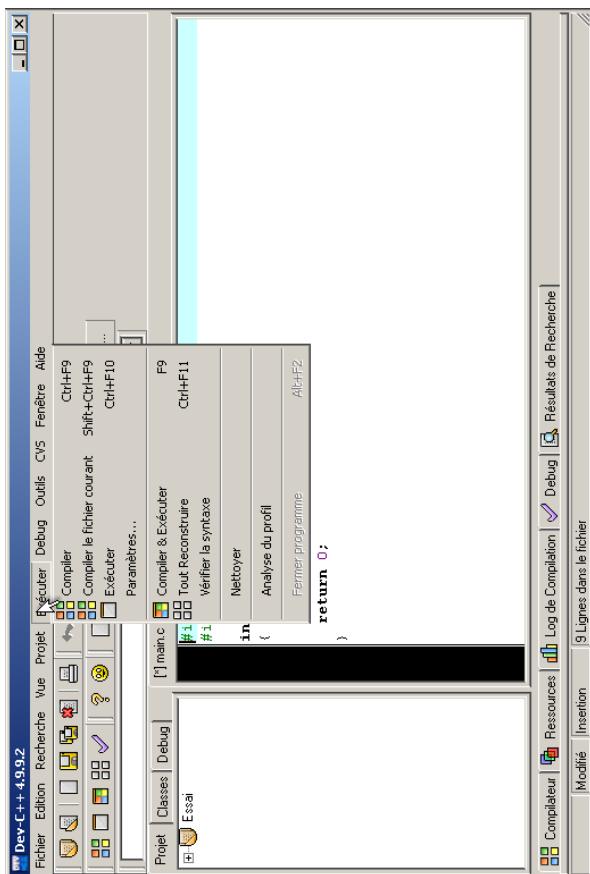
## 9 Le C en pratique

### 4.3.2 Les Menus

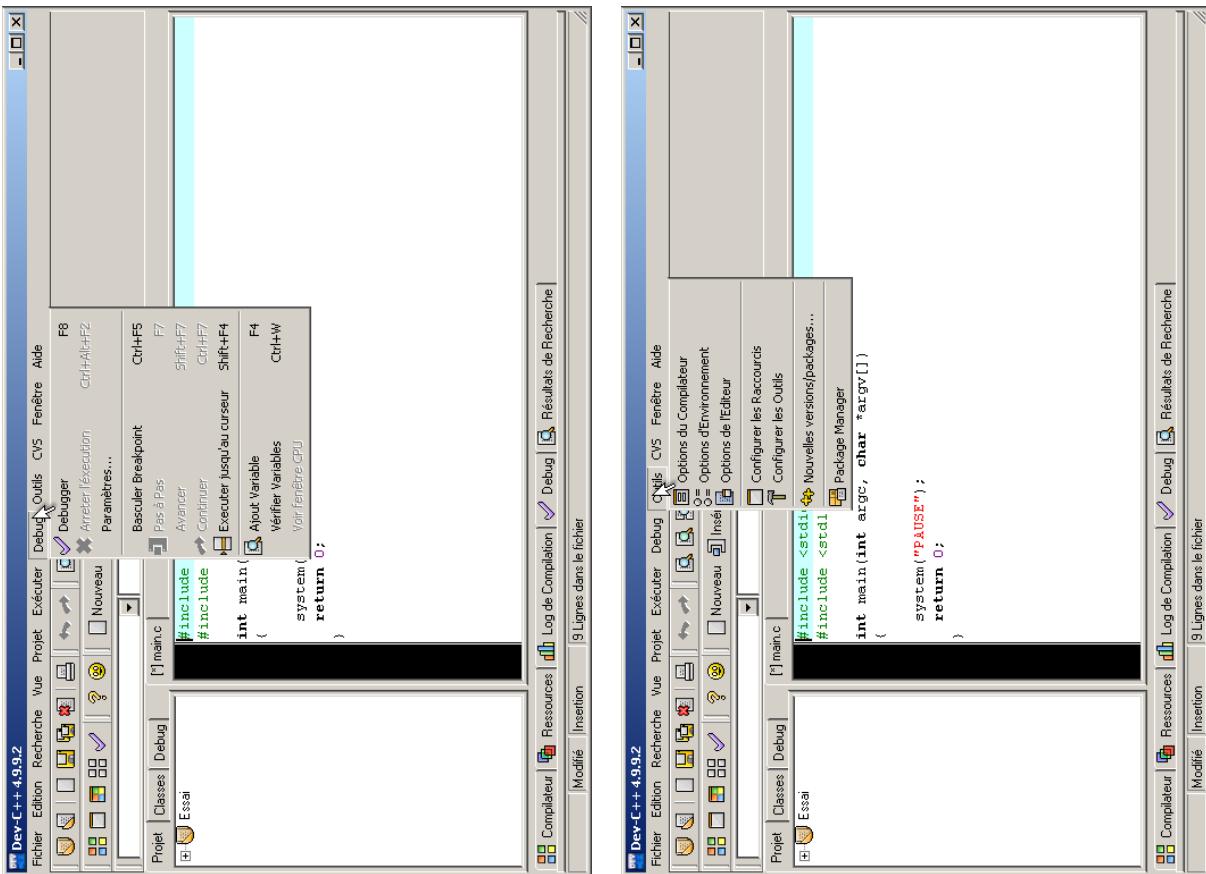




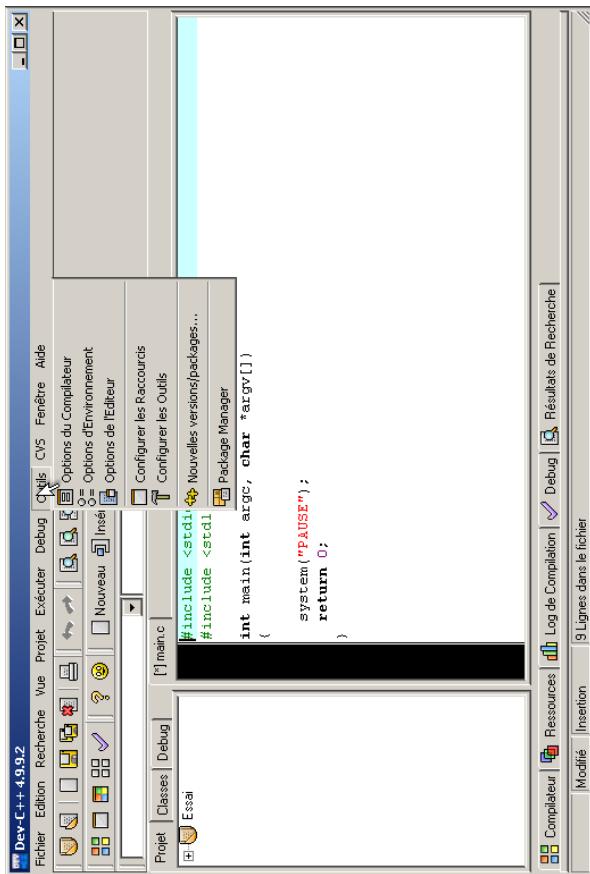
5



6

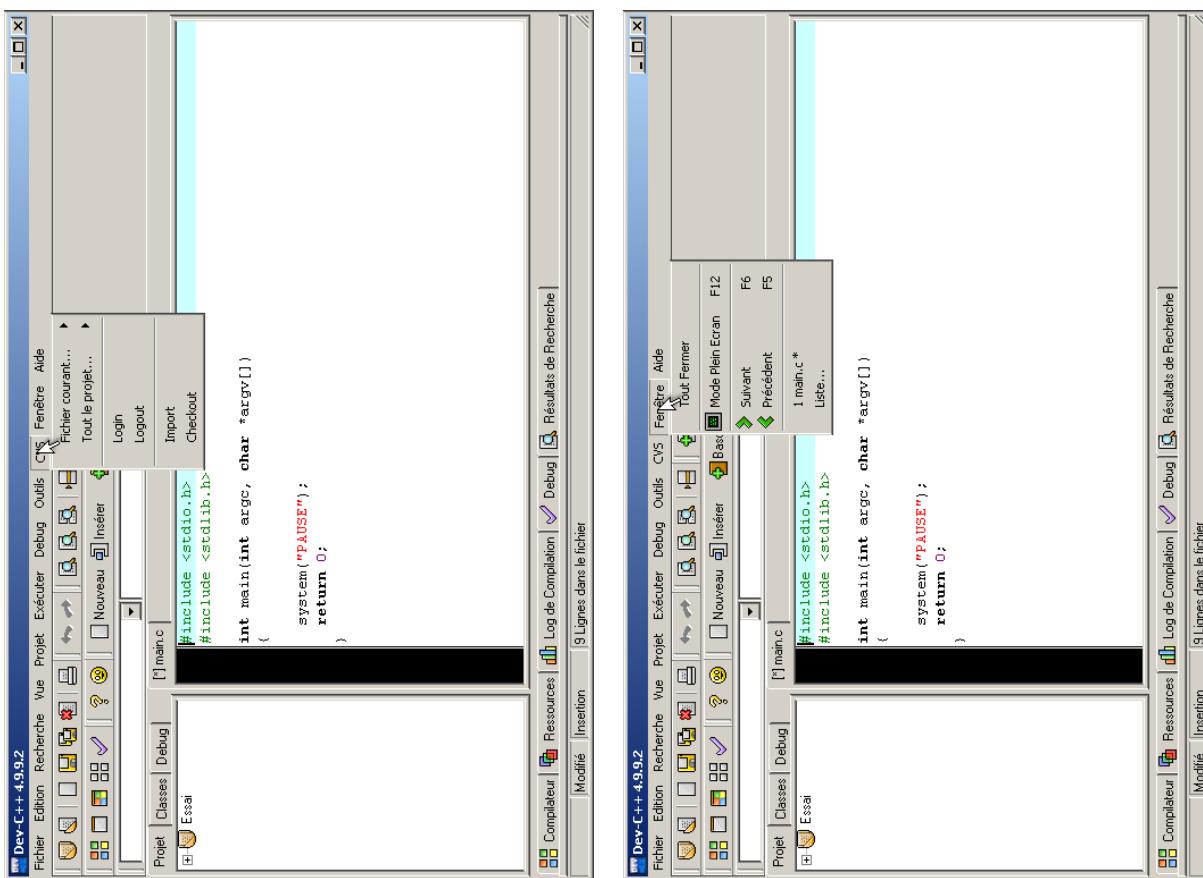


7

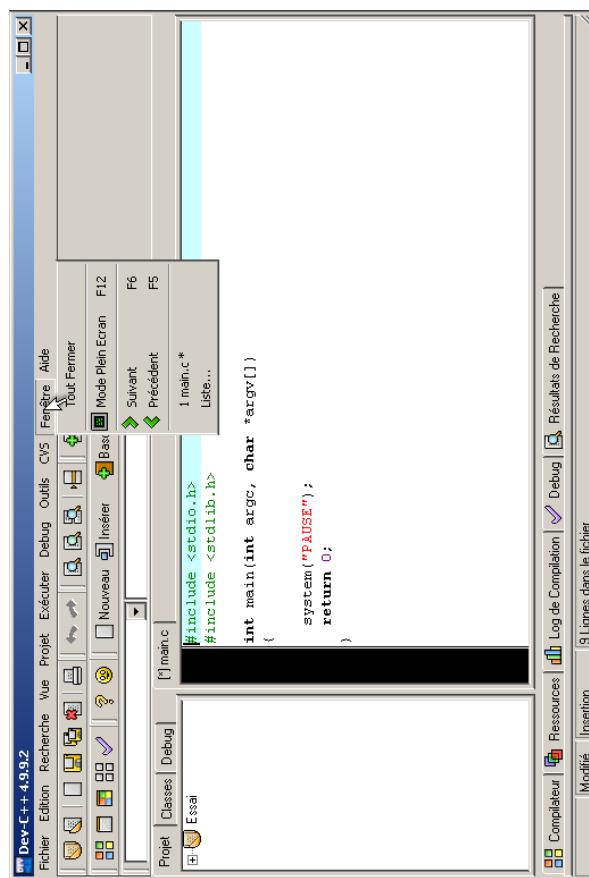


8

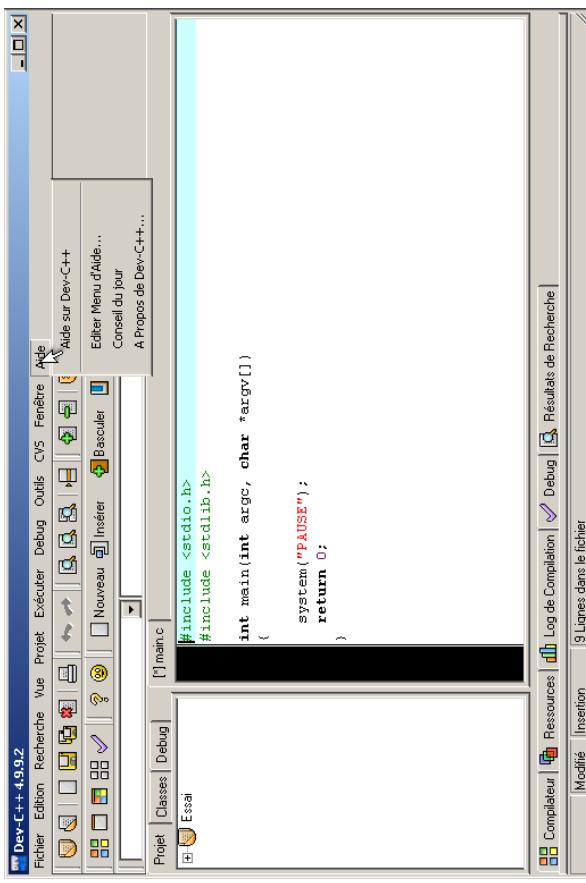
## 9 Le C en pratique



9



10



11

## 4.4 GNU gcc

### 4.4.1 gcc

#### Caractéristiques

- Compilateur en ligne de commande : **gcc**
- Respecte très bien la norme ANSI-C

#### Compilation et édition des liens

- Compiler sans édition des liens :

```
1 gcc -c fichier.c -o fichier.o
```

- Compiler avec édition des liens :

```
1 gcc fichier.c -o fichier
```

- Edition des liens :

```
1 gcc fichier1.o fichier2.o -o fichier
```

#### Options pour l'optimisation du code produit

- O0** : pas d'optimisation (debug possible)
- O1** : optimise taille du code et temps d'exécution (debug possible)
- O2** : optimise taille du code et temps d'exécution (debug impossible)
- O3** : optimise temps d'exécution (debug impossible)
- Os** : optimise taille du code
- f???????** : flag particulier d'optimisation
- mcpu=pentium** : optimise pour exécution sur un pentium mais peut tourner sur autre
- march=pentium** : optimise pour pentium mais plus exécutable sur i386

#### Options du préprocesseur

- DSYMBOL** : #ifdef SYMBOL
- DSYMBOL=toto** : #if SYMBOL=toto
- USYMBOL** : #ifndef SYMBOL
- Idir** : où chercher les fichiers #include <fichier.h>

#### Linker avec des bibliothèques

- lefence** : link avec la bibliothèque efence
- Ldir** : où chercher les bibliothèques à linker
- static** / –**shared** : pour faire des bibliothèque statique ou partagée
- pthread** : utiliser les threads

#### Options de compilation

- Activer les avertissements : –**Wall** –**Wimplicit** –**Wunused** –**Wuninitialized** –**Wshadow** –**Wunreachable\_code**
- Desactiver les avertissements : –**w**
- Rendre les constantes de chaînes de caractères écrivables : –**f writable-strings**
- Profiler le code (à la compilation comme à l'édition des liens) : –**p** –**pg**
- Activer le déboggage (à la compilation comme à l'édition des liens) : –**g3** –**ggdb3**
- Création d'un exécutable Position Independent Code (pour l'édition dynamique des liens (dll, so)) : –**fPIC**

#### De nombreuses options existent

Consultez la documentation de gcc !!

### 4.4.2 Les frontends

---

#### Les principaux frontends à gcc

- Dev-C++, code : :blocks : sous windows
- Kdevelop, Anjuta : sous linux

#### Le débugger en ligne de commande : gdb

Les principaux frontends à gdb : Dev-C++, code : :blocks, Kdevelop, Anjuta

#### Fonctionnement similaire à Visual Studio

- Création d'un projet et des fichiers
- Modification des options du projets
- Compilation
- Déboggage

S'appuie sur la notion de Makefile : script pour la compilation du programme

### 4.4.3 Attention

---

#### Mode langage C

- La plupart des compilateurs évoqués plus haut prennent en charge le C et le C++
- Il est donc important de régler le compilateur en mode C lorsque vous l'utilisez (extension .c, options spécifiques...)
- Les compilateurs évoqués supportent au moins la norme C89. Certains implémentent totalement ou partiellement la norme C99. Des petites différences peuvent donc apparaître entre les différents compilateurs.

#### ✓ L'essentiel à retenir :

---

1. Comment fait-on pour structurer ses codes sources en fichiers sources et fichiers d'entête ? A quoi ça sert ?
2. Quel est l'intérêt de la mise en forme du code ? A quoi sert Doxygen ? Comment s'en sert-on ?
3. Quelle est la différence entre une bibliothèque statique et une bibliothèque dynamique ?
4. De quoi a-t-on besoin pour utiliser une bibliothèque ?
5. Suis-je capable de créer un projet, éditer, compiler et debugger avec un IDE ?

## **Troisième partie**

**Niveau intermédiaire**



# 10

# Types avancés

## 1 Champs de bits, unions et énumérations

### 1.1 Champs de bits

#### Définition

Type de données similaire à une structure. Les champs ne sont pas des variables mais des bits.

#### Syntaxe

```
1 struct {
2 Type1 NomChamps1 : Taille1;
3 Type2 NomChamps2 : Taille2;
4 ...
5 TypeN NomChampsN : TailleN;
6 } NomVar1, NomVar2;
7 struct NomDuChampsDeBits {
8 Type1 NomChamps1 : Taille1;
9 Type2 NomChamps2 : Taille2;
10 ...
11 TypeN NomChampsN : TailleN;
12 } NomVar1, NomVar2;
13 struct NomDuChampsDeBits {
14 Type1 NomChamps1 : Taille1;
15 Type2 NomChamps2 : Taille2;
16 ...
17 TypeN NomChampsN : TailleN;
18 };
19 struct NomDuChampsDeBits NomVar1, NomVar2;
```

#### Principes

- Chaque taille correspond à un nombre de bits
- Les bits des champs sont mis bout à bout
- Permet de manipuler facilement et de façon transparente des groupes de bits

#### Occupation mémoire

$$\left\lceil \frac{\text{nombre total de bits des champs}}{8} \right\rceil$$

#### Exemple

```
1 struct MaStructure
2 {
3 signed int id : 8;
4 unsigned short color : 4;
5 unsigned int icone : 2;
6 } MaVariable;
7 MaVariable.icone = 1;
8 MaVariable.color = 3;
9 MaVariable.id = 10;
```

## 10 Types avancés



### 1.2 Unions

#### Définition

Type de données similaire à une structure. Les champs partagent la même zone mémoire et se recouvrent.

#### Occupation mémoire

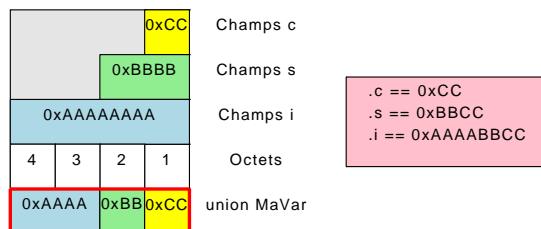
$$\max\{\text{sizeof}(champs)\}$$

#### Syntaxe

```
1 union
2 {
3 Type1 Champs1;
4 ...
5 TypeN ChampsN;
6 } MaVariable;
7 union MonUnion
8 {
9 Type1 Champs1;
10 ...
11 TypeN ChampsN;
12 } MaVariable;
13 union MonUnion
14 {
15 Type1 Champs1;
16 ...
17 TypeN ChampsN;
18 };
19 union MonUnion MaVariable;
```

#### Exemple

```
1 union LesNombres
2 {
3 unsigned char c;
4 unsigned short s;
5 unsigned int i;
6 } MaVar;
7 MaVar.i = 0xAAAAAA;
8 MaVar.s = 0xBBB;
9 MaVar.c = 0xCC;
```



### 1.3 Enumérations

#### Définition

- Définir des constantes entières
- Définir un type de données dont le domaine de valeur est les constantes

## Remarques

- Les constantes ne doivent pas être en conflit avec d'autres identifiants
- Les constantes sont de type **int**
- Une variable d'un type **enum** peut contenir :
  1. les constantes associées au type **enum**
  2. tout entier signé (néanmoins certains compilateurs émettent des avertissements)

## Syntaxe

```

1 enum
2 {
3 Id1 , Id2 ,
4 Id3 = Valeur1 ,
5 Id4
6 } MaVar;
7 enum NomEnumeration
8 {
9 Id1 , Id2 ,
10 Id3 = Valeur1 ,
11 Id4
12 } MaVar;
13 enum NomEnumeration
14 {
15 Id1 , Id2 ,
16 Id3 = Valeur1 ,
17 Id4
18 };
19 enum NomEnumeration MaVar;
```

## Valeurs des constantes

- Par défaut, la valeur d'une constante = valeur de la précédente +1
- Par défaut, la première constante est 0
- Pour chaque constante, on peut préciser la valeur associée
- On peut utiliser plusieurs fois la même valeur
- Les constantes peuvent être négatives

## Exemple

```

1 enum MonEnum
2 {
3 rouge ,
4 vert = 2 ,
5 bleu = 5 ,
6 marron ,
7 noir
8 };
9
10 enum MonEnum MaVar;
11
12 MaVar = rouge ;
13 if (MaVar == noir)
14 ...
15 int val = (int) MaVar;
16 MaVar = (enum MonEnum) val;
```

## 2 Typedef

### 2.1 typedef

Lorsqu'on utilise les structures, les unions, les énumérations, les pointeurs, ...

- l'écriture peut devenir lourde
- on ne peut pas s'abstraire du type de l'objet manipulé

## Une solution

L'opérateur **typedef** permet de définir de nouveaux types de données en renommant des types de données

### Syntaxe

```
1 typedef AncienType NomDuNouveauType ;
```

Syntaxe identique à la déclaration d'une variable du type considéré mais précédée de **typedef**

### Exemple

```
1 typedef struct
2 {
3 int champ1;
4 char champ2;
5 } MonType1;
6 typedef enum
7 {
8 cst1 ,
9 cst2
10} MonType2;
11 typedef int MonTypeEntier;
12 typedef int * MonPointeurSurEntier;
13
14 void f(void)
15 {
16 MonType1 var1;
17 MonType2 var2;
18 MonTypeEntier entier = 10;
19 MonPointeurSurentier pentier = &entier;
20 }
```

## 2.2 Exemple 1

---

```
1 #include <stdio.h>
2
3 typedef enum
4 {
5 character ,
6 integer ,
7 real
8 } TypeElement ;
9
10 typedef union
11 {
12 char c ;
13 int i ;
14 double d ;
15 } TypeStockage ;
16
17 typedef struct
18 {
19 TypeElement Type;
20 TypeStockage Valeur;
21 } TypeCouple ;
22
23 void afficher(TypeCouple couple);
24
25 int main(void)
26 {
27 TypeCouple c ;
28
29 c.Type = character ;
30 c.Valeur.c = 'A' ;
31 afficher(c) ;
32
33 c.Type = integer ;
34 c.Valeur.i = 10 ;
35 afficher(c) ;
36
37 c.Type = real ;
38 c.Valeur.d = 3.1415 ;
39 afficher(c) ;
40
41 return 0 ;
42 }
```

```

43 void afficher(TypeCouple couple)
44 {
45 printf("Le type de l'élément est %d et sa valeur est ", couple.Type);
46 switch (couple.Type)
47 {
48 case character : printf("%c", couple.Valeur.c);
49 break;
50 case integer : printf("%d", couple.Valeur.i);
51 break;
52 case real : printf("%lf", couple.Valeur.d);
53 break;
54 default : printf("*** Type d'élément inconnue ***");
55 break;
56 }
57 }
58 }
```

## 2.3 Exemple 2

```

1 #include <stdio.h>
2
3 enum { TAILLE = 50 };
4
5 typedef struct
6 {
7 char Nom[TAILLE];
8 char Prenom[TAILLE];
9 } Personne;
10
11 typedef struct
12 {
13 Personne NomPrenom;
14 int AnneeNaissance;
15 } EtatCivil;
16
17 void SaisiePersonne(Personne * p)
18 {
19 puts("\nNom : ");
20 gets(p->Nom);
21 puts("\nPrénom : ");
22 gets(p->Prenom);
23 puts("\n");
24 }
25 void AfficherPersonne(Personne p)
26 {
27 printf("%s %s", p.Nom, p.Prenom);
28 }
29 void SaisieEtatCivil(EtatCivil * ec)
30 {
31 SaisiePersonne(&ec->NomPrenom);
32 puts("Année de naissance : ");
33 scanf("%d", &ec->AnneeNaissance);
34 puts("\n");
35 }
36 void AfficherEtatCivil(EtatCivil ec)
37 {
38 AfficherPersonne(ec.NomPrenom);
39 printf(" né en %d\n", ec.AnneeNaissance);
40 }
41
42 int main(void)
43 {
44 EtatCivil ec;
45 SaisieEtatCivil(&ec);
46 AfficherEtatCivil(ec);
47 return 0;
48 }
```

## 2.4 Structures récursives

```

1 struct SListe
2 {
3 int data;
4 struct Liste * next;
5 };
```

```
6 typedef struct _Liste
7 {
8 int data;
9 struct _Liste * next;
10 } Liste;
11
12
13 struct SListe MaListe;
14 Liste MaDeuxiemeListe;
```

## 3 Tableaux

---

### 3.1 Rappel sur les tableaux

---

#### Tableau

- Ensemble d'emplacements mémoires contigus
- Portent tous le même nom
- Contiennent tous le même type de données

#### Elément d'un tableau

Un emplacement spécifique d'un tableau

### 3.2 Rappel sur les tableaux 1D

---

#### Syntaxe

```
1 TypeDesElements Identificateur [NombreDElements];
```

- Type élément : le type de **tous** les éléments du tableau
- Identificateur : le nom de la variable représentant le tableau
- Nombre éléments : le nombre d'éléments du tableau : **C'est une constante (littérale/symbolique) de type entier**

#### Taille réelle d'un tableau en mémoire

```
1 NombreDElements * sizeof(TypeDesElements)
2 sizeof(Identificateur)
```

Sur certain système/compilateur : il peut y avoir des limites de taille  $\Rightarrow$  allocation dynamique (cf. plus loin)

#### Accès à un élément du tableau

- Syntaxe :

```
1 Identificateur [IndiceDeLElement]
```

- IndiceDeLElement : peut être une expression qui renvoie un entier
- Indice du premier élément : 0
- Indice du dernier élément : NombreDElements - 1
- Il n'y a pas de vérification des indices  $\Rightarrow$  **Il est possible de sortir de la zone mémoire réservée pour le tableau !!**

#### Initialisation des éléments d'un tableau par affectation

```
1 tab [2]=10;
```

#### Initialisation lors de la déclaration (en utilisant des constantes)

```
1 int tab [4] = {2,4,6,8}; // contenu = 2 4 6 8
2 int tab [4] = {2,4}; // contenu = 2 4 ? ?
3 int tab [4] = { ,4, ,8}; // contenu = ? 4 ? 8
4 int tab [] = {2,4,6,8}; // contenu = 2 4 6 8 et dimension = 4
```

Lorsque les éléments sont connus, la dimension du tableau n'est pas nécessaire !

## Organisation en mémoire

```
1 int tab[6];
```

### Adresses en mémoire



## 3.3 Tableau 2D

### Syntaxe

```
1 TypeDesElements Identificateur [NombreLigne][NombreColonne];
```

### Tableau 2D

Fonctionne comme une matrice

### Occupation mémoire

Tableau 2D  $\iff$  tableau 1D de tableaux 1D

```
1 NombreLigne * NombreColonne * sizeof(TypeDesElements)
2 sizeof(Identificateur)
```

### Accès à un élément

```
1 Identificateur [IndiceLigne][IndiceColonne]
```

Indice du premier élément : **[0][0]**

### Initialisation des éléments par affectation

```
1 tab[i][j] = 10;
```

### Lors de la déclaration (avec des constantes)

```
1 int tab[2][3] = {1, 2, 3, 4, 5, 6};
2 int tab[2][3] = {{1, 2, 3}, {4, 5, 6}};
3 int tab[][3] = {1, 2, 3, 4, 5, 6};
```

Dans le dernier cas, le nombre de lignes est déterminé automatiquement via la formule :

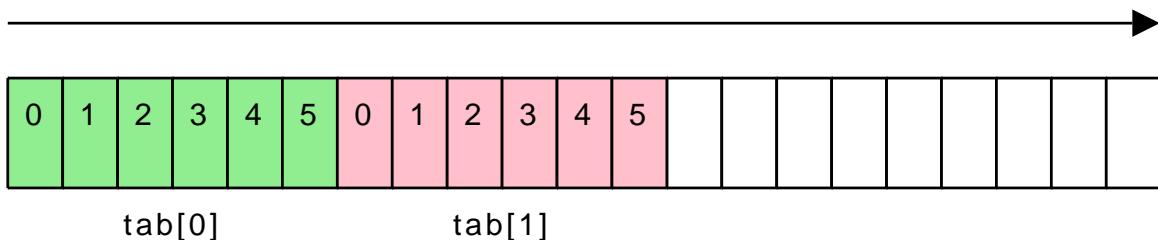
$$\frac{\text{Nombre d'élément du tableau}}{\text{Nombre d'élément d'une ligne}}$$

|   | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 1 | 4 | 5 | 6 |

### Organisation en mémoire

```
1 int tab [2][6];
```

Adresses en mémoire



### 3.4 Tableaux à N dimensions

---

#### Tableaux à N dimensions

On applique les mêmes principes

#### Syntaxe

```
1 TypeDesElements Identificateur [Taille1][Taille2][Taille3]...[TailleN];
```

#### Occupation mémoire

Tableau ND  $\iff$  tableau 1D de tableaux (N-1)D

```
1 Taille1 * Taille2 * ... * TailleN * sizeof(TypeDesElements)
2 sizeof(Identificateur)
```

#### Accès à un élément

```
1 tab [i] [j] [k]
```

#### L'initialisation suit les mêmes principes

```
1 int tab [2][2][2] = { 1, 2, 3, 4, 5, 6, 7, 8};
2 int tab [2][2][2] = { {1, 2}, {3, 4}, {5, 6}, {7, 8} };
3 int tab [2][2][2] = { { {1, 2}, {3, 4} }, { {5, 6}, {7, 8} } };
4 int tab [] [2][2] = { { {1, 2}, {3, 4} }, { {5, 6}, {7, 8} } };
```

### 3.5 Tableaux 1D et fonctions

---

#### Paramètres

La liste des paramètres de la fonction contient :

```
1 TypeElement IdTableau [Taille]
2 TypeElement IdTableau []
```

La taille est optionnelle et n'est pas pris en compte par le compilateur.

## Appel de la fonction

- On utilise le nom du tableau
- **Les éléments du tableau ne sont pas recopiés, seul l'adresse du tableau est transmise par valeur**  $\Rightarrow$  Il n'existe pas de mécanisme de passage par valeur des éléments d'un tableau !

```

1 #define N 100
2
3 void fct(int t[N], int n);
4
5 int main(void)
6 {
7 int tab[N];
8 fct(tab, 100);
9 return 0;
10 }
11
12 void fct(int montableau[N], int longueur)
13 {
14 int i;
15 for(i = 0; i < longueur; ++i)
16 montableau[i] = i;
17 }
```

## 3.6 Tableaux 2D/ND et fonctions

### Paramètres

Tableau 2D et ND : mêmes principes que tableaux 1D

```

1 TypeElement IdTableau[Taille1][Taille2]
2 TypeElement IdTableau[][Taille2]
```

La première taille est optionnelle et n'est pas pris en compte par le compilateur. Les autres tailles sont obligatoires et constantes.

### Remarques

- **Les éléments du tableau ne sont pas recopiés, seul l'adresse du tableau est transmise par valeur**  $\Rightarrow$  Il n'existe pas de mécanisme de passage par valeur des éléments d'un tableau !
- Identificateur de tableau  $\Leftrightarrow$  pointeur sur le premier élément
- Il existe une syntaxe de passage des tableaux utilisant des pointeurs !  $\Rightarrow$  On verra ça plus loin.

### ✓ L'essentiel à retenir :

1. Quelles sont les différences entre une structure, un champs de bits et une union ?
2. A quoi sert la commande **typedef** ?
3. Quelle est l'organisation mémoire des tableaux ? Comment transmet-on un tableau à une fonction ?
4. Si je modifie les éléments d'un tableau dans une fonction, quels sont les effets de bord ?



# 11

# Les pointeurs

## 1 Les pointeurs

### 1.1 Rappel de l'essentiel

#### Pointeur

- Variable qui contient l'adresse d'une données en RAM
- Pointe sur une données typée

#### Syntaxe

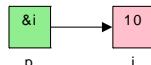
```
1 Type * NomVariable;
```

#### Opérateurs associés

- **&** : adresse de l'opérande (du premier octet)
- **\*** : déréférencement / indirection / valeur de la zone pointée

#### Exemple

```
1 int * p, i, j;
2 i = 10;
3 p = &i;
4 j = *p;
```



### 1.2 Opérations sur les pointeurs

#### Affectation

```
1 pointeur = rvalue;
```

```
1 int * ptr1, * ptr2, i;
2 ptr1 = &i;
3 ptr2 = ptr2;
```

#### Comparaisons : <, <=, >, >=, == et !=

Le résultat dépend de la machine et du SE

#### Comparaisons : cas de l'x86 en mode réel (ex : DOS)

1. Comparaison du segment et de l'offset
2. Les segments se recouvrent  $\implies$  1 zone mémoire  $\equiv$  plusieurs adresses
  - le numéro de case mémoire est : segment \* 16 + offset
  - Seg:Off+16 est donc égal à Seg+1:Off

#### Comparaisons : cas de l'x86 en mode protégé (ex : Windows, Linux)

1. L'adressage de la mémoire est linéaire  $\implies$  1 zone mémoire  $\equiv$  1 adresse
2. Comparaison de deux nombres !

## 11 Les pointeurs

### Fiabilité des comparaisons

- La comparaison est fiable dans ces deux cas
- Mais Exemple de Turbo C (mode réel) + pointeur court :
  - Un pointeur court ne contient que l'offset  $\Rightarrow$  compare que les offsets !  $\equiv$  danger !
  - Turbo C et le mode réel ne sont quasiment plus utilisés !
- Sur toutes nouvelles plateformes, il faut se renseigner sur la fiabilité de la comparaison des pointeurs.

### 1.3 L'arithmétique des pointeurs

#### Opérateurs valides

+, -, ++ et --

#### Sémantique

Actions liées au **type de données** pointé.

### 1.4 Exemples

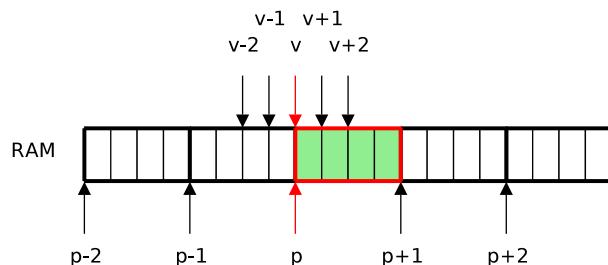
#### Exemple pour la déclaration Type \* p;

```
1 p++;
```

1. Incrémente l'adresse contenue dans **p** de **sizeof (Type)** octets
2. **p** pointe sur l'enregistrement de type **Type** suivant en RAM

#### Exemple

```
1 int i;
2 int * p = &i;
3 char * v = (char*) p;
```



## 2 Tableaux et pointeurs

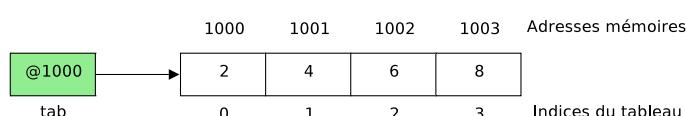
### 2.1 Tableaux 1D et pointeurs

#### 2.1.1 Dualité entre tableaux 1D et pointeurs

#### Dualité

L'identificateur est également un pointeur sur le premier élément du tableau

```
1 char tab[4] = { 2, 4, 6, 8 };
```



```
1 tab == &tab[0]
```

## 2.1.2 Arithmétique des pointeurs

### Arithmétique des pointeurs

`tab+1` : pointeur sur le 2ième élément (incrémenté le pointeur de `sizeof(Type)` octets) `tab+i` : pointeur sur le  $(i+1)$ -ième élément `tab[i]` et `*(t+i)` : le  $(i+1)$ -ième élément

#### Exemple

```

1 int tab [4] = {2, 4, 6, 8};
2 int * p1 = tab;
3 int * p2 = &tab [0];
4
5 tab [0] = 1;
6 p1 [0] = 1;
7 p2 [0] = 1;
8 *tab = 1;
9 *p1 = 1;
10 *p2 = 1;
11 tab [1] = 2;
12 *(p1+1) = 2;
13 *(p2+1) = 2;
```

## 2.1.3 Types

### Conséquences

Tout pointeur peut être manipulé comme un tableau 1D.

#### Type, type du pointeur et type pointé

- Type du tableau : `TypeDUnElement[Taille]` ou `TypeDUnElement *`
- Type pointé : `TypeDUnElement`

## 2.1.4 Fonctions, Tableaux 1D et pointeurs

### Soit les déclarations suivantes

```

1 #define N 10
2 int t[N];
```

### Les prototypes suivants sont équivalents

```

1 void fct(int * t);
2 void fct(int t[]);
3 void fct(int t[N]);
```

Dans le dernier cas, N peut être remplacé par n'importe quelle constante. Celle-ci n'est pas prise en compte.

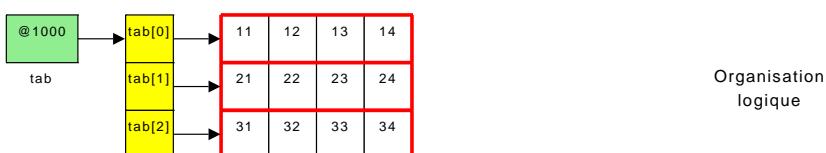
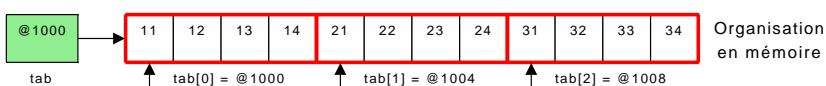
## 2.2 Tableaux 2D et pointeurs

### 2.2.1 Dualité entre tableaux 2D et pointeurs

#### Dualité

L'identificateur est également un pointeur sur le premier élément de la matrice

Adresses mémoires 1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011



```

1 char tab [3][4] = { 11, 12, 13, 14, 21, 22, 23, 24, 31, 32, 33, 34 };
2 tab == &tab [0] tab [0] == &tab [0][0] tab [1] == &tab [1][0]
3 *(tab) == *tab [0] == tab [0][0] *tab [1] == tab [1][0]
4 tab+i == &tab [i]
5 tab [i]+j == &tab [i][j] == (*(tab+i)+j)
```

## 11 Les pointeurs

### 2.2.2 Arithmétique des pointeurs

#### Remarque

Un tableau 2D est un tableau 1D de tableaux 1D

#### Arithmétique des pointeurs

**tab+1** : pointeur sur le premier **octet** de la 2*ième* ligne (incrémenté le pointeur de **sizeof(Type)\*NbColonne octets**)  
**tab+i** : pointeur sur le premier **octet** de la (i+1)-*ième* ligne **tab[i]** et **\*(t+i)** : pointeur sur le premier élément de la (i+1)-*ième* ligne **tab[i]+j** et **\*(t+i)+j** : pointeur sur le j-*ième* élément de la i-*ième* ligne **tab[i][j]**, **\*(tab[i]+j)** et **\*(\*(tab+i)+j)** : le j-*ième* élément de la i-*ième* ligne

#### Exemple

```
1 int tab[10][20];
2 int (*p1)[20];
3 int * p2;
4
5 p1 = tab;
6 p2 = *(tab+1);
7 p2 = tab[1]+3;
```

#### Remarque

Attention, il ne faut pas confondre **int (\*t)[M]** et **int \*t[M]** !

### 2.2.3 Types

#### Type, type du pointeur et type pointé

- Type du tableau **tab** : **TypeDUnElement[Taille1][Taille2]** ou **TypeDUnElement[Taille2] \***
- Type de l'élément du tableau **tab[i]** : **TypeDUnElement (\*)[Taille2]**
- Type de l'élément pointé du tableau **tab[i]** : **TypeDUnElement[Taille2]** ou **TypeDUnElement \***
- Type de l'élément du tableau **tab[i][j]** : **TypeDUnElement**

### 2.2.4 Fonctions, Tableaux 2D et pointeurs

#### Soit les déclarations suivantes

```
1 #define N 10
2 #define M 100
3 int t[N][M];
```

#### Les prototypes suivants sont équivalents

```
1 void fct(int t[][M]);
2 void fct(int t[N][M]);
3 void fct(int (*t)[M]);
```

#### Remarques

- **Attention** : **int (\*t)[M]** est différent de **int \*t[M]** !
- Les tableaux 1D du tableau 2D sont contigus en mémoire  $\Rightarrow$  Il est donc possible d'utiliser la fonction :

```
1 void fctbis(int * tp);
```

avec l'appel :

```
1 fctbis((int*)t);
```

Dans ce cas, l'accès aux éléments s'effectue par :

```
1 x = *(tp + i * M + j)
```

**Attention** : ca ne marche que pour ce type de tableau !!! Il ne faut pas généraliser et surtout ne pas en abuser !

## 2.3 Exemples

### 2.3.1 Exemple 1

```

1 #include <stdio.h>
2 #include <math.h>
3
4 #define NMAX 11
5
6 void init1(long * t)
7 {
8 int i;
9 for(i = 0; i < NMAX; ++i)
10 t[i] = (long)pow(2.0, i);
11 }
12 void init2(long t[])
13 {
14 int i;
15 for(i = 0; i < NMAX; ++i)
16 t[i] = (long)pow(3.0, i);
17 }
18 void init3(long t[NMAX])
19 {
20 int i;
21 for(int i = 0; i < NMAX; ++i)
22 t[i] = (long)pow(4.0, i);
23 }
24
25 void afficher(long * t)
26 {
27 int i;
28 for(i = 0; i < NMAX; ++i)
29 printf("(i, val)=(%d,%ld)\n", i, *(t+i));
30 }
31
32 int main(void)
33 {
34 long t[NMAX];
35 init1(t); afficher(t);
36 init2(t); afficher(t);
37 init3(t); afficher(t);
38 return 0;
39 }
```

### 2.3.2 Exemple 2

```

1 #include <stdio.h>
2 #include <math.h>
3
4 #define NLIGNE 3
5 #define NCOL 5
6
7 void init1(long (*t)[NCOL])
8 {
9 int i, j;
10 for(i = 0; i < NLIGNE; ++i)
11 for(j = 0; j < NCOL; ++j)
12 *((t+i)+j) = (long)pow(2.0, i*NCOL+j);
13 }
14
15 void init2(long t[][NCOL])
16 {
17 int i, j;
18 for(i = 0; i < NLIGNE; ++i)
19 for(j = 0; j < NCOL; ++j)
20 *(t[i]+j) = (long)pow(3.0, i*NCOL+j);
21 }
22
23 void init3(long t[NLIGNE][NCOL])
24 {
25 int i, j;
26 for(i = 0; i < NLIGNE; ++i)
27 for(j = 0; j < NCOL; ++j)
28 t[i][j] = (long)pow(3.0, i*NCOL+j);
```

## 11 Les pointeurs

```
29 }
30
31 void afficher1(long * t)
32 {
33 int i , j ;
34 for(i = 0; i < NLIGNE; ++i)
35 for(j = 0; j < NCOL; ++j)
36 printf("t[%d][%d] = %ld\n" , i , j , *(t+i*NCOL+j));
37 }
38
39 void afficher2(long (*t)[NCOL])
40 {
41 int i , j ;
42 for(i = 0; i < NLIGNE; ++i)
43 for(j = 0; j < NCOL; ++j)
44 printf("t[%d][%d] = %ld\n" , i , j , t[i][j]);
45 }
46
47 int main(void)
48 {
49 long t[NLIGNE][NCOL];
50
51 init1(t); afficher1((long*)t);
52 init2(t); afficher2(t);
53 init3(t); afficher2(t);
54 return 0;
55 }
```

### 2.3.3 Exemple 3

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5 int tableau[50];
6 int * p = tableau;
7 char * v = (char*) tableau;
8 int i;
9
10 for(i = 0; i < 50; ++i)
11 tableau[i] = i+1;
12
13 printf("L'entier situé à la position 10 par tableau est %d\n", tableau[10]);
14 printf("L'entier situé à la position 10 par p est %d\n", *(p+10));
15 printf("L'entier situé à la position 10 par v est %d\n",
16 *((int*)(v+10*sizeof(int))));
```

## 3 Erreurs courantes (et souvent fatales)

### 3.1 Déclaration de pointeurs

```
1 int main(void)
2 {
3 int * p, q;
4 int i, j;
5 p = &i;
6 q = &j;
7
8 return 0;
9 }
```

### 3.2 Utilisation de pointeurs non initialisés

```
1 int main(void)
2 {
3 int x, * p;
4 x = 10;
5 *p = x;
6 return 0;
7 }
```

### 3.3 Affectation du contenu d'une variable à un pointeur

```

1 int main(void)
2 {
3 int x, * p;
4 x = 10;
5 p = x;
6 return 0;
7 }
```

#### Remarque

La syntaxe est correcte !

### 3.4 Valeur pointée par $p+n$

```

1 int main(void)
2 {
3 int tab[10];
4 int * p = tab;
5 *(p+20) = 10;
6 return 0;
7 }
```

### 3.5 Priorité des opérateurs

```

1 int tab[10];
2 int * p = tab;
3 int res;
4 res = *p+2;
5 res = *(p+2);
```

#### Remarque

$*$  est plus prioritaire que  $+$  !

## 4 Remarques

### 4.1 Les chaînes de caractères

#### Les chaînes de caractères vues comme des pointeurs

Une chaîne de caractère en C est un tableau dont la fin est marquée par le caractère '\0'. Il est donc possible de manipuler une chaîne de caractères à l'aide de pointeurs.

```

1 char chaine[] = "Voici un exemple";
2 char * p = chaine;
3
4 while (*p != '\0')
5 p++;
6 printf("La longueur de la chaîne '%s' est %d", p, (int)(p-chaine));
```

### 4.2 Les pointeurs génériques void\*

#### Définition

Un pointeur générique est un pointeur sur une zone mémoire dont le contenu est non typé. Un tel pointeur est défini de la façon suivante :

```
1 void * p;
```

#### Limitations des opérateurs

- Le déréférencement  $*$  n'est pas possible car le type pointé n'est pas défini
- L'arithmétique des pointeurs est utilisable mais celle-ci suppose que les éléments pointés sont des octets (donc de taille 1).

### Utilisation

Les pointeurs génériques s'utilisent habituellement lorsqu'une variable doit pouvoir contenir un pointeur quelque soit son type. Pour l'utiliser, un changement du type du pointeur est généralement utilisé.

#### Exemple

```
1 void ajouter(enum Type type, void * result, void * source, int length)
2 {
3 int * pint1, * pint2;
4 double * pdouble1, * pdouble2;
5 switch (type)
6 {
7 case Entier : {
8 pint1 = (int*)result;
9 pint2 = (int*)source;
10 while (length-- > 0)
11 *pint1++ += *pint2++;
12 }
13 case Reel : {
14 pdouble1 = (double*)result;
15 pdouble2 = (double*)source;
16 while (length-- > 0)
17 *pdouble1++ += *pdouble2++;
18 }
19 default : exit(1);
20 }
21 }
```

#### ✓ L'essentiel à retenir :

---

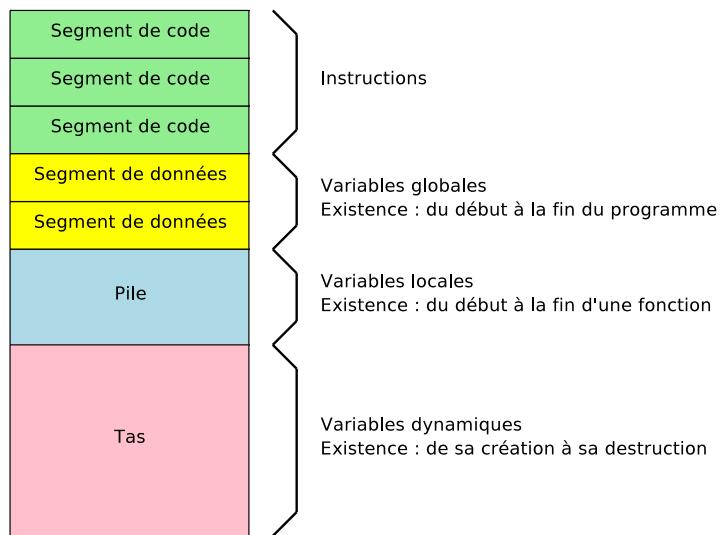
1. Qu'est-ce qu'un pointeur ? Quelles opérations peut-on faire avec ? Quels sont les principes de l'arithmétique des pointeurs ?
2. Quel est le lien entre un tableau 1D et un pointeur ? Quelles écritures sont équivalentes ? Quels sont les types associés ?
3. Quel est le lien entre un tableau 2D et les pointeurs ? Quelles écritures sont équivalentes ? Quels sont les types associés ?
4. Suis-je capable de généraliser ces informations aux tableaux ND ?
5. Est-ce que je comprend les erreurs courantes ?
6. Je suis capable de manipuler une chaîne de caractères sous la forme d'un pointeur.
7. Je connais les limitations des pointeurs génériques.

# 12

# Allocation dynamique

## 1 Allocation, libération et réallocation

### 1.1 Rappel sur l'organisation de la mémoire et les variables



### 1.2 Variables dynamiques

#### Créateur

Crée par l'utilisateur dans le tas

#### Existence

- Indépendante de la notion de portée
- Le programmeur décide quand la créer et quand la détruire

#### Fonctionnement

- Réserver une zone mémoire dans le tas
- Affecter l'adresse de cette zone à un pointeur
- Utiliser la zone de mémoire via le pointeur
- Libérer la zone mémoire allouée

### 1.3 Allocation de mémoire

#### 1.3.1 malloc

#### Objectif

Permet de réserver/allouer une zone mémoire du tas

#### Fichiers d'entête

stdlib.h (ou alloc.h ou malloc.h)

## 12 Allocation dynamique

---

### Prototype

```
1 void * malloc (size_t size);
```

- **size** : la taille en octet de la zone mémoire à allouer/réserver
- Résultat :
  - Un pointeur non typé vers le premier octet de la zone mémoire allouée
  - Ou le pointeur **NULL** si échec de l'allocation

### Remarques

- La zone mémoire n'est pas initialisée. Son contenu est quelconque !
- En cas d'impossibilité à allouer la mémoire  $\Rightarrow$  Pointeur **NULL**  $\Rightarrow$  **Test nécessaire et impératif !**
- **size\_t** : c'est un **typedef** sur un type entier
- Des contraintes de taille maximale peuvent exister :
  - En fonction du SE
  - En fonction du compilateur
  - En fonction de la machineEx : DOS sur x86 limité à 64ko !
- Le pointeur obtenu est non typé  $\Rightarrow$  **Cast obligatoire !**

### Utilisation

```
1 float * ptrf;
2 ptrf = (float *) malloc(50 * sizeof(float));
3 if (ptrf == NULL)
4 exit(-1);
```

```
1 float * ptrf;
2 if ((ptrf = (float *) malloc(50 * sizeof(float))) == NULL)
3 exit(-1);
```

```
1 struct MaStructure
2 {
3 int i;
4 char c;
5 float f;
6 };
7 struct MaStructure * ptrf;
8 ptrf = (struct MaStructure *) malloc(100 * sizeof(struct MaStructure));
9 if (ptrf == NULL)
10 exit(-1);
```

### 1.3.2 calloc

---

### Objectif

Permet de réserver/allouer des enregistrements consécutifs dans le tas

### Fichiers d'entête

stdlib.h (ou alloc.h ou malloc.h)

### Prototype

```
1 void * calloc (size_t nmemb, size_t size);
```

- **nmemb** : le nombre d'enregistrement à allouer
- **size** : la taille en octet de chaque enregistrement
- Résultat :
  - Un pointeur non typé vers le premier octet de la zone mémoire allouée
  - Ou le pointeur **NULL** si échec de l'allocation

### Remarques

- La zone mémoire est initialisée avec des 0 (zéros)
- Les mêmes contraintes que **malloc** existent

## Utilisation

```

1 float * ptrf;
2 ptrf = (float *) calloc(50, sizeof(float));
3 if (ptrf == NULL)
4 exit(-1);

1 float * ptrf;
2 if ((ptrf = (float *) calloc(50, sizeof(float))) == NULL)
3 exit(-1);

```

```

1 struct MaStructure
2 {
3 int i;
4 char c;
5 float f;
6 };
7 struct MaStructure * ptr;
8 ptr = (struct MaStructure *) calloc(100, sizeof(struct MaStructure));
9 if (ptr == NULL)
10 exit(-1);

```

## 1.4 Libération de mémoire avec free

### Objectif

Libérer de la mémoire dynamique

- Quand : Lorsqu'une zone mémoire allouée dynamiquement n'est plus utile
- Comment : utilisation de la fonction **free**

### Fichiers d'entête

stdlib.h (ou alloc.h ou malloc.h)

### Prototype

```
1 void free (void * ptr);
```

- **ptr** : un pointeur sur le premier octet d'une zone mémoire dynamique allouée par **malloc** ou **calloc** ou **realloc**
- Si **ptr** est **NULL** alors aucune zone mémoire n'est libérée

### Remarque

- Si **ptr** n'est pas un pointeur obtenu par **malloc** ou **calloc** ou **realloc** alors le comportement de **free** est indéterminé.

## Utilisation

```

1 float * ptrf;
2 ptrf = (float *) malloc(50 * sizeof(float));
3 if (ptrf == NULL)
4 exit(-1);
5 ...
6 free(ptrf);

```

```

1 struct MaStructure
2 {
3 int i;
4 char c;
5 float f;
6 };
7 struct MaStructure * ptr;
8 ptr = (struct MaStructure *) calloc(100, sizeof(struct MaStructure));
9 if (ptr == NULL)
10 exit(-1);
11 ...
12 free(ptr);

```

## 12 Allocation dynamique

---

### 1.5 Réallocation mémoire avec realloc

---

#### Objectif

Permet de redimensionner une zone mémoire allouée dynamiquement

#### Fichiers d'entête

stdlib.h (ou alloc.h ou malloc.h)

#### Prototype

```
1 void * realloc (void * ptr, size_t size);
```

- **ptr** : Un pointeur vers la zone mémoire dynamique à redimensionner
- **size** : La nouvelle taille en octet de la zone mémoire dynamique
- Résultat :
  - Un pointeur vers la nouvelle zone mémoire
  - Ou **NULL** en cas d'échec du redimensionnement et **la zone ptr n'est pas modifiée !**

#### Remarques

- Si **ptr** n'est pas un pointeur obtenu par **malloc** ou **calloc** ou **realloc** alors le comportement de **realloc** est indéterminé.
- Le contenu de la zone mémoire commune est préservée.
- Si **ptr** est **NULL** alors **realloc** se comporte comme **malloc**
- Si **size** est 0 alors **realloc** se comporte comme **free**. La valeur renvoyée peut alors être **NULL** ou un pointeur acceptable par **free**.
- Comparer le résultat à **NULL** n'est pas suffisant pour détecter des erreurs. **NULL** peut apparaître quand **size** est 0.

### 1.6 SIZE\_MAX en C99

---

#### Remarque

La norme C99 définit une nouvelle constante : **SIZE\_MAX**. Elle correspond à la plus grande valeur numérique qui peut être stockée dans une variable de type **size\_t**. Elle ne donne pas d'information sur la taille maximale d'une zone allouable mais elle permet de gérer les dépassements de capacités du type **size\_t**.

## 2 Applications

---

### 2.1 Tableaux 1D dynamiques

---

#### 2.1.1 Tableau 1D

---

#### Rappel

```
1 TypeDesElements IdTab[NbDElements];
```

#### Déclarer le pointeur sur le 1er élément

```
1 TypeDesElements * ptr;
```

#### Réserver la mémoire

```
1 ptr = (TypeDesElements *) malloc(NbDElements * sizeof(TypeDesElements));
```

#### Accéder aux éléments

```
1 *(ptr+i)
2 ptr[i]
```

## Libération du tableau

```
1 free(ptr);
```

### 2.1.2 Exemple

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef struct _Personne
5 {
6 char nom[20];
7 char prenom[20];
8 } Personne;
9
10 typedef struct _Liste
11 {
12 Personne * date;
13 int combien;
14 } Liste;
15
16 Liste * CreerListe()
17 {
18 Liste * l;
19 l = (Liste *) malloc(sizeof(Liste));
20 l->date = NULL;
21 l->combien = 0;
22 }
23
24 void Ajouter(Liste * l, const char * nom, const char * prenom)
25 {
26 Personne * p;
27
28 if (l == NULL)
29 exit(-1);
30 p = (Personne *) realloc(l->date, (l->combien+1) * sizeof(Personne));
31 if (p==NULL)
32 exit(-1);
33 l->date = p;
34 l->combien++;
35 strcpy(l->date[l->combien-1].nom, nom);
36 strcpy(l->date[l->combien-1].prenom, prenom);
37 }
38
39 void Detruire(Liste * l)
40 {
41 if (l != NULL)
42 {
43 free(l->date);
44 free(l);
45 }
46 }
47
48 int main(void)
49 {
50 Liste * l;
51 l = CreerListe();
52 Ajouter(l, "Dupond", "Jean");
53 Ajouter(l, "Marcel", "Andrée");
54 return 0;
55 }
```

## 2.2 Tableaux 2D semi-dynamiques

### 2.2.1 Tableau 2D semi-dynamique

#### Rappel

```
1 TypeDesElements IdTab[NbLigne][NbCol];
```

Tableaux 2D semi-dynamiques : NbCol est connu et fixe. Chaque ligne est du type **TypeDesElements (\*ptr)[NbCol];**

## 12 Allocation dynamique

---

Déclarer le pointeur sur le 1er élément

```
1 TypeDesElements (* ptr) [NbCol];
```

Réserver la mémoire

```
1 ptr = (TypeDesElements (*) [NbCol]) malloc(NbLigne * sizeof(TypeDesElements [NbCol]));
```

Accéder aux éléments

```
1 *(*(ptr+i)+j)
2 ptr [i] [j]
```

Libération du tableau

```
1 free(ptr);
```

### 2.2.2 Exemple

```
1 int main(int argc, char * argv[])
2 {
3 int t[20][10];
4 int (*tab)[10];
5
6 tab = t;
7
8 printf(" %d", sizeof(int[10]));
9
10 tab = (int (*)[10]) malloc(sizeof(int[10]) * 30);
11 tab[25][4] = 1;
12
13 return 0;
14 }
```

## 2.3 Tableaux 2D dynamiques

---

### 2.3.1 Tableau 2D dynamique

Rappel

```
1 TypeDesElements * IdTab[NbLigne];
```

Déclarer le pointeur sur le 1er élément

```
1 TypeDesElements ** ptr;
```

Réserver la mémoire

```
1 ptr = (TypeDesElements **) malloc(NbLigne * sizeof(TypeDesElements *));
2 for(i = 0; i < NbLigne; ++i)
3 ptr [i] = (TypeDesElements *) malloc(sizeof(TypeDesElements));
```

Accéder aux éléments

```
1 *(*(ptr+i)+j)
2 ptr [i] [j]
```

Libération du tableau

```
1 for(i = 0; i < NbLigne; ++i)
2 free(ptr [i]);
3 free(ptr);
```

### 2.3.2 Exemple

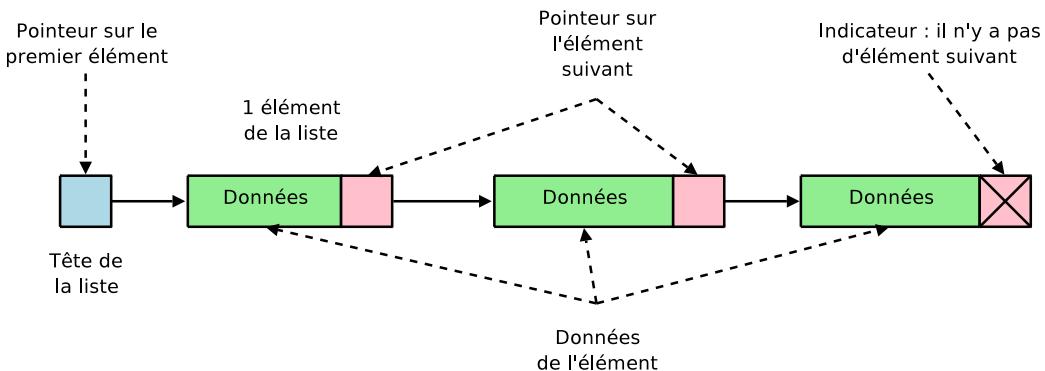
```

1 int main(int argc, char * argv[])
2 {
3 int i;
4 int t[20][10];
5 int ** tab;
6
7 tab = t; // IMPOSSIBLE !!!
8
9 tab = (int **) malloc(sizeof(int*) * 30);
10 for(i = 0; i < 30; ++i)
11 tab[i] = (int *) malloc(sizeof(int) * 100);
12 tab[30][50] = 1;
13
14 }
```

## 2.4 Listes simplement chaînées

### 2.4.1 Listes simplement chaînées

#### Principes



- Liste vide : La tête est le pointeur NULL
- Un élément de liste :
  - un champ de données
  - un pointeur vers l'élément suivant

#### Définition d'un élément

```

1 typedef struct _IdStructure
2 {
3 // données
4 struct _IdStructure * suivant;
5 } Nom;
```

#### Exemple

```

1 typedef struct _Personne
2 {
3 char nom[20];
4 char prenom[20];
5 struct _Personne * suiv;
6 } Personne;
```

#### Déclarer la tête de liste

```
1 Nom * tete;
```

#### Exemple :

```
1 Personne * tete1;
```

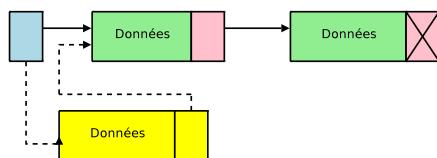
```

1 typedef Personne ListePersonne;
2
3 ListePersonne * tete2;
```

## 12 Allocation dynamique

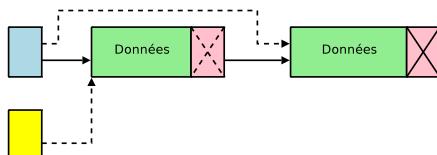
### 2.4.2 Insertion et suppression

#### Insertion d'un nouvel élément en tête de liste



```
1 void inserer_tete_liste(Personne ** tete)
2 {
3 Personne * p;
4 p = (Personne *) malloc(sizeof(Personne));
5 if (p == NULL)
6 exit(-1);
7 gets(p->nom);
8 gets(p->prenom);
9 p->suit = *tete;
10 *tete = p;
11 }
```

#### Suppression d'un élément en tête de liste



```
1 void supprimer_tete_liste(Personne ** tete)
2 {
3 Personne * temp;
4 if (*tete != NULL)
5 {
6 temp = *tete;
7 *tete = (*tete)->suit;
8 free(temp);
9 }
10 }
```

### 2.4.3 Parcours

#### Parcourir une liste

```
1 typedef void (*fct_traiter)(Personne * p);
2
3 void parcourir_liste(Personne * tete , fct_traiter fct)
4 {
5 if (tete != NULL)
6 {
7 fct(tete);
8 parcourir_liste(tete->suit);
9 }
10 }
11 void parcourir_liste(Personne * tete , fct_traiter fct)
12 {
13 Personne * p;
14 p = tete;
15
16 while (p != NULL)
17 {
18 fct(p);
19 p = p->suit;
20 }
21 }
```

### 2.4.4 Recherche

#### Recherche du dernier élément d'une liste

```

1 Personne * dernier_element(Personne * tete)
2 {
3 Personne * p;
4
5 if (tete == NULL)
6 return NULL;
7
8 p = tete;
9 while (p->suiv != NULL)
10 p = p->suiv;
11 return p;
12 }
```

### 2.4.5 Programme principal

#### Fonction main

```

1 void afficher(Personne * p)
2 {
3 printf("%s %s\n", p->nom, p->prenom);
4 }
5
6 int main(void)
7 {
8 Personne * tete = NULL;
9 Personne * dernier;
10
11 inserer_tete_liste(&tete);
12 inserer_tete_liste(&tete);
13 inserer_tete_liste(&tete);
14 inserer_tete_liste(&tete);
15 supprimer_tete_liste(&tete);
16 parcourir_liste(tete, afficher);
17 dernier = dernier_element(tete);
18
19 return 0;
}
```

## 3 Erreurs courantes avec les pointeurs

### 3.1 Il faut vérifier le résultat d'une allocation

#### Bonnes pratiques

Lors de l'utilisation de toutes fonctions allouant ou modifiant la taille d'une zone mémoire, il est nécessaire de vérifier que l'opération s'est bien déroulée.

#### Mauvaises pratiques

La machine sur laquelle j'exécute mon programme a beaucoup plus de mémoire que je n'en ai besoin. Je peux donc supposer que les allocations réussiront toujours.

- Vrai dans 99.9% des cas
- **mais** dans 0.01% des cas, vous vous retrouverez à utiliser des pointeurs **NULL**.
- Or, **\*NULL** désigne la zone mémoire située à l'adresse 0.
  - Si vous avez de la chance, tout accès à cette mémoire terminera immédiatement votre programme en raison d'une violation d'accès.
  - Sinon vous accèderez à une zone mémoire de façon involontaire (table d'interruptions, ...) avec effets indirectes immédiats ou non immédiats.

## 12 Allocation dynamique

---

### 3.2 Confusion entre le segment de données, la pile et le tas

---

#### Exemple

```
1 char * s1 = "Bonjour !";
2 char s2 [] = "Bonsoir !";
3 int tab[20];
4 char * p;
5 int * pi = tab;
6 ...
7 p = (char *) realloc(s2, sizeof(char) * LongeurMessage);
8 pi = (int *) realloc(pi, sizeof(int) * 30);
9 ...
10 free(s1);
11 free(p);
12 free(pi);
```

Ca peut vite devenir dangereux et difficile à débugger si les libérations/réallocations sont effectuées dans une fonction recevant un pointeur.

### 3.3 Libération ou réallocation sur des pointeurs calculés

---

#### Exemple

```
1 int * tab = NULL;
2 int * tab2;
3 tab = (int*) malloc(sizeof(int) * 10);
4 tab2 = tab + 1;
5 free(tab2); // ou realloc(tab2, 0);
```

### 3.4 Utilisation d'un pointeur après libération

---

#### Exemple

```
1 int * tab;
2 tab = (int*) malloc(sizeof(int) * 10);
3 ...
4 free(tab);
5 ...
6 tab[0] = 10;
```

### 3.5 Utilisation d'un pointeur avant allocation

---

#### Exemple

```
1 int * tab;
2 tab[0] = 10;
```

### 3.6 Pertes de mémoire/memory leaks

---

#### Exemple

```
1 int * p = malloc(sizeof(int) * 10);
2 if (p != NULL)
3 {
4 ...
5 p = realloc(p, sizeof(int) * 20);
6 ...
7 }
```

## 4 La bibliothèque string.h

---

### 4.1 La bibliothèque string.h

---

#### Contenu

- Fonctions de manipulation de zones mémoires
- Fonctions de manipulation de chaînes de caractères

## 4.2 Fonctions de manipulation de zones mémoires en C99

### Copier une zone mémoire

```
1 void *memcpy (void *dest, const void *src, size_t n);
```

Copie n octets de src vers dest. Retourne dest. **Les deux zones mémoires ne doivent pas se chevaucher !**

### Copier une zone mémoire

```
1 void *memmove (void *dest, const void *src, size_t n);
```

Copie n octets de src vers dest. Retourne dest. **Les deux zones mémoires peuvent se chevaucher !**

### Initialisation d'une zone mémoire

```
1 void *memset (void *s, int c, size_t n);
```

Initialise les n octets commençant à s à la valeur c.

### Comparaison de zones mémoires

```
1 int memcmp (const void *s1, const void *s2, size_t n);
```

Compare les n premiers octets de s1 et s2. Retourne :

- un entier négatif si  $s1[] < s2[]$
- 0 si  $s1[] == s2[]$
- un entier positif si  $s1[] > s2[]$

### Recherche de caractère

```
1 void *memchr (const void *s, int c, size_t n);
2 void *memrchr (const void *s, int c, size_t n);
```

Recherche la première/dernière occurrence de c dans les n premiers/derniers octets de s[]. Retourne un pointeur vers l'occurrence trouvée ou NULL si non trouvée

## 4.3 Fonctions de manipulation de chaînes de caractères

### Longueur

```
1 size_t strlen (const char *s);
```

Renvoie la taille de la chaîne sans tenir compte du '\0' terminal.

### Duplicer une chaîne de caractère

```
1 char * strdup (const char *s);
```

- La nouvelle chaîne est allouée par malloc
- Elle devra donc être libérée par free

### Comparaisons sensibles à la casse

```
1 int strcmp (const char *s1, const char *s2);
2 int strncmp (const char *s1, const char *s2, size_t n);
```

Retourne :

- un entier négatif si  $s1[] < s2[]$
- 0 si  $s1[] == s2[]$
- un entier positif si  $s1[] > s2[]$

## 12 Allocation dynamique

### Comparaisons insensibles à la casse

```
1 int strcasecmp (const char *s1, const char *s2);
2 int strncasecmp (const char *s1, const char *s2, size_t n);
```

Retourne :

- un entier négatif si  $s1[] < s2[]$
- 0 si  $s1[] == s2[]$
- un entier positif si  $s1[] > s2[]$

### Remarques sur les comparaisons

- Deux chaînes ne peuvent pas être comparées à l'aide de l'opérateur `==`. Pourquoi ?
- Pourquoi ne peut-on pas utiliser `if (strcmp(s1,s2)) action;` pour faire l'action en cas d'égalité ?
- Les méthodes strXXXcmp ne tiennent pas compte de la locale active mais utilise l'ordre ASCII (ex : z<é). Pour effectuer une comparaison tenant compte de la locale (ex : e<é<f), il faut utiliser :

```
1 int strcoll (const char *s1, const char *s2);
```

### Concaténation

```
1 char *strcat (char *dest, const char *src);
```

Ajoute les caractères de src à la suite de ceux de dest. Renvoie dest.

```
1 char *strncat (char *dest, const char *src, size_t n);
```

Ajoute au plus n caractères de src à la suite de ceux de dest. Renvoie dest. Dans les deux cas, le caractère '\0' est ajouté en fin de chaîne.

### Copier une chaîne

```
1 char *strcpy (char *dest, const char *src);
```

Copie src[] dans dest[]. Renvoie dest. Aucune vérification de dépassement n'est effectuée.

```
1 char *strncpy (char *dest, const char *src, size_t n);
```

Copie au plus n caractères de src[] dans dest[]. Si src a une longueur supérieure à n alors le caractère '\0' n'est pas recopié. Le résultat n'est donc pas une chaîne.

### Recherche de caractère

```
1 char *strchr (const char *s, int c);
2 char * strrchr (const char *s, int c);
```

Recherche c dans la chaîne s[] à partir du début ou de la fin. Retourne un pointeur sur la première occurrence du caractère ou NULL si le caractère n'a pas été trouvé.

### Recherche de chaîne

```
1 char *strstr (const char *meule_de_foin, const char *aiguille);
```

Recherche l'aiguille dans le meule de foin. Renvoie un pointeur vers la première occurrence de l'aiguille ou NULL si elle n'a pas été trouvée.

### L'essentiel à retenir :

1. Je sais allouer, réallouer et libérer de la mémoire sur le tas.
2. Je sais allouer et utiliser des tableaux dynamiques : 1D, 2D semi-dynamique et 2D-dynamique.
3. Je suis capable de généraliser aux tableaux dynamiques en ND.
4. Je connais et comprend les erreurs classiques avec l'allocation dynamique.
5. Je connais les principales fonctions de la bibliothèque string.h.

# 13 Fichiers

## 1 Généralités

### 1.1 Fichiers et flux

#### La norme ANSI-C

L'ANSI-C différencie les fichiers et les flux (fLOTS).

#### Fichiers

entité physique d'E/S (écran, clavier, disque, ...)

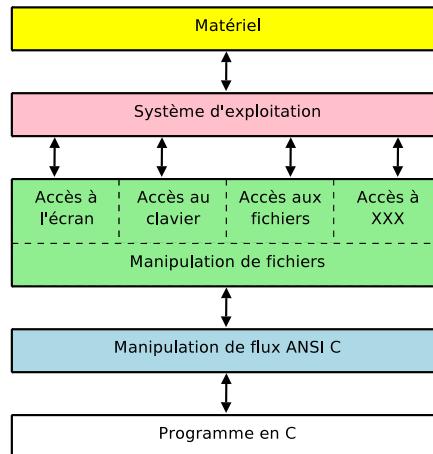
#### Flux

entité logique d'E/S utilisant des fichiers et définissent la nature des données manipulées

#### Manipuler un fichier

- Associer 1 fichier à 1 flux (à l'ouverture du fichier)
- Gestion transparente et unifiée

#### Fichiers et flux



#### Remarque

Il existe d'autre type de fonction pour manipuler les fichiers mais ce n'est pas de l'ANSI C

- POSIX
- BSD
- ...

#### Deux types de flux

- Texte : séquence de caractères
- Binaire : séquence d'octets

#### Fichier d'entête

stdio.h

## 13 Fichiers

### 1.2 Flux/fichiers en C

#### FILE

- Tous les flux/fichiers sont associés à une structure opaque **FILE**
- Définie dans stdio.h
- Instance créée à l'ouverture d'un flux/fichier
- Détruite à la fermeture d'un flux/fichier
- En pratique, on définit une variable comme suit pour chaque fichier :

```
1 FILE * Id ;
```

#### Les fonctions

|       |        |         |          |
|-------|--------|---------|----------|
| fopen | fclose | fputc   | fgetc    |
| fputs | fgets  | fread   | fwrite   |
| fseek | ftell  | rewind  | clearerr |
| feof  | ferror | fprintf | fscanf   |

## 2 Ouverture et fermeture

### 2.1 Ouverture de flux/fichier

#### Prototype

```
1 FILE *fopen (const char *path, const char *mode);
```

#### path

- Chaîne de caractères contenant le nom du fichier à ouvrir (unité, répertoire, nom de fichier...)
- ATTENTION : c:\monrep\monfichier.ext se code

```
1 "c:\\\\monrep\\\\monfichier . ext"
```

#### mode

Chaîne de caractères définissant le mode d'accès (lecture/écriture) et le type de flux (texte/binaire)

#### Mode d'accès

| Mode | Lecture / écriture                          | Position du curseur de flux |
|------|---------------------------------------------|-----------------------------|
| r    | Lecture                                     | Au début                    |
| r+   | Lecture/écriture                            | Au début                    |
| w    | Écriture, créé ou tronqué à 0 octet         | Au début                    |
| w+   | Lecture/écriture, créé ou tronqué à 0 octet | Au début                    |
| a    | Écriture, créé                              | A la fin                    |
| a+   | Lecture/écriture, créé                      | A la fin                    |

#### Type de flux

| Mode | Type de flux |
|------|--------------|
| t    | texte        |
| b    | binaire      |

A partie du C99, le type de flux n'est plus nécessaire par la norme, n'a plus d'impact et est ignoré sur la majorité des plateformes.

#### Valeur renournée

- Un pointeur sur une structure **FILE** représentant le flux/fichier
- Ou **NULL** en cas d'erreur. La variable **extern int errno;** de errno.h contient une description de l'erreur.

## Exemple

```
1 FILE * fp;
2 fp = fopen("donnees.txt", "rt");
3 if (fp == NULL)
4 exit(-1);
```

```
1 FILE * fp;
2 if ((fp = fopen("donnees.txt", "rt")) == NULL)
3 exit(-1);
```

## 2.2 Fermeture de flux/fichier

### Prototype

```
1 int fclose (FILE *stream);
```

### Valeur renournée

- 0 en cas de réussite
- EOF en cas d'échec et errno contient une description de l'erreur

## 3 Lectures et écritures

### 3.1 Lecture/écriture d'un caractère

#### Lecture d'un caractère

```
1 int fgetc (FILE *stream);
```

- Renvoie la valeur **unsigned char** du caractère lu
- Ou **EOF** en cas d'erreur ou de fin de fichier

#### Ecriture d'un caractère

```
1 int fputc (int c, FILE *stream);
```

- Renvoie la valeur **unsigned char** du caractère écrit
- Ou **EOF** en cas d'erreur

#### Condition d'utilisation

Utilisable pour des fichiers textes et binaires

### 3.2 Lecture/écriture d'une chaîne de caractères

#### Lecture d'une chaîne de caractères

```
1 char * fgets (char * s, int size, FILE * stream);
```

- Lit au plus **size-1** caractères, les met dans **s** et ajoute '\0'
- S'arrête dès qu'un retour-chariot est lu ou que la fin du fichier est atteinte
- Remarque : le retour-chariot lu est mis dans **s**
- Renvoie **s** en cas de réussite ou **NULL** en cas d'échec ou de fin de fichier

#### Ecriture d'une chaîne de caractères

```
1 int fputs (const char *s, FILE *stream);
```

- Renvoie un nombre non négatif en cas de réussite ou **EOF** en cas d'échec

#### Conditions d'utilisation et remarque

- Utilisable avec des fichiers textes uniquement
- Retour-chariot : \n\r ou \n

## 13 Fichiers

### 3.3 Lecture/écriture d'octets

#### Lecture d'octets

```
1 size_t fread (void *ptr, size_t size, size_t nmemb, FILE *stream);
```

- Lecture de **nmemb** objets de taille **size** et stocke le résultat à partir de l'octet pointé par **ptr**
- Renvoie le nombre d'objets lus
- Déetecter une erreur : **valeur renvoyée < nmemb**

#### Ecriture d'octets

```
1 size_t fwrite (const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

- Ecriture de **nmemb** objets de taille **size** et lit les données à partir de l'octet pointé par **ptr**
- Renvoie le nombre d'objets écrits
- Déetecter une erreur : **valeur renvoyée < nmemb**

#### Exemple

```
1 FILE * fp;
2 double t[4]; int n;
3 n = fwrite(t, sizeof(double), 4, fp);
4 n = fread(t, sizeof(double), 4, fp);
```

#### Mise en garde

Attention lorsque vous utilisez des structures ou des types complexes. Des octets de padding (ou d'alignement) peuvent être ajoutés en fonction du compilateur et de la plateforme.

### 3.4 Padding/alignement de données en mémoire

#### Exemple

```
1 struct StructureAlignee
2 {
3 char c1;
4 short s;
5 int i;
6 char c2;
7 };
```

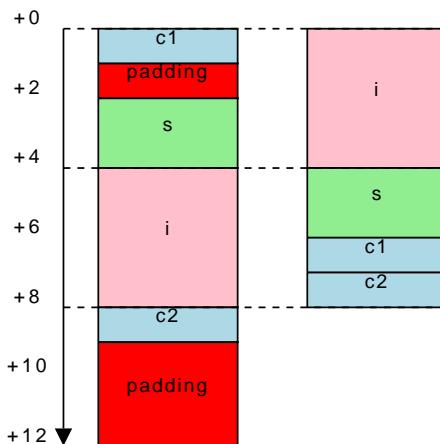
  

```
1 struct StructureNonAlignee
2 {
3 int i;
4 short s;
5 char c1;
6 char c2;
7 };
```

Sous Linux sur x86 avec gcc, la première structure occupe une place mémoire de 12 octets et la seconde de 8 octets.

#### Pourquoi ?

Sur cette architecture, les alignements s'effectuent sur des multiples de 4 octets (32 bits). Ainsi on obtient l'organisation suivante :



Les données sont alignées pour permettre au processeur d'accéder plus rapidement aux différentes structures (cas d'un tableau) et aux champs de la structure.

## Remarques

- La prise en compte du padding est très importante si vous êtes contraint en terme d'occupation mémoire.
- Pour minimiser l'occupation mémoire, il suffit de déclarer les champs dans l'ordre des tailles décroissantes.
- Il y a des cas où ça ne sert à rien. Ex : rajouter un caractère c3 à la fin des deux structures : les deux structures auront une taille de 12 octets.
- Sous gcc, l'option **-Wpadded** permet au compilateur de vous avertir lorsqu'il fait du padding.
- Directives des compilateurs VC++/GCC pour forcer l'alignement d'une structure dans le code source : **#pragma pack**

## 3.5 Positionnement du curseur de lecture/écriture

### Obtenir le position du curseur

```
1 long ftell (FILE *stream);
```

- Renvoie la position du curseur
- Ou -1 en cas d'erreur. errno contient une description de l'erreur.

### Placer le curseur au début du flux

```
1 void rewind (FILE *stream);
```

### Positionnement du curseur

```
1 int fseek (FILE *stream, long offset, int whence);
```

- Positionne le curseur
- Nouvelle position = origine + offset octet (positif ou négatif)
- Origine dépend de whence :
  - **SEEK\_SET** : origine = début du fichier
  - **SEEK\_CUR** : origine = position courante
  - **SEEK\_END** : origine = fin du fichier
- Renvoie la nouvelle position ou -1 en cas d'erreur (errno contient une description de l'erreur)

## 3.6 Formattage des entrées/sorties

### Ecriture formattée

```
1 int fprintf (FILE *stream, const char *format, ...);
```

Fonctionnement identique à **printf**

### Lecture formatée

```
1 int fscanf (FILE * stream, const char * format, ...);
```

Fonctionnement identique à **scanf**

## 4 Notions avancées

### 4.1 Gestion des buffers d'E/S

#### Les fonctions de stdio.h pour la manipulation de flux utilise des tampons

- pour réduire le nombre d'accès au périphérique
- pour accélérer les opérations d'E/S

#### Pour forcer le vidage du tampon en écriture

```
1 int fflush (FILE *flux);
```

- Renvoie 0 en cas de réussite
- Ou EOF en cas d'erreur. errno contient une description de l'erreur.

### 4.2 Gestion des erreurs

---

Effacer les indicateurs d'erreurs du fichier et l'indicateur de fin de fichier

```
1 void clearerr (FILE *stream);
```

Retourner un booléen à vrai si la fin du fichier est atteinte

```
1 int feof (FILE *stream);
```

Retourner un booléen à vrai si une erreur a eu lieu sur le fichier

```
1 int ferror (FILE *stream);
```

Afficher un message d'erreur d'E/S

```
1 void perror (const char *s);
```

Affiche la chaîne `s` si elle n'est pas vide ou `NULL` suivit de deux points, puis du message d'erreur correspondant à l'erreur contenu dans `errno`.

### 4.3 Flux prédéfinis

---

Il existe des flux prédéfinis en ANSI-C

- `stdin` : mode texte, lecture uniquement, entrée standard (clavier)
- `stdout` : mode texte, écriture uniquement, sortie standard (écran)
- `stderr` : mode texte, écriture uniquement, sortie standard (écran)

Certains compilateurs définissent d'autres flux (ex : Turbo C)

- `stdaux` : mode binaire, lecture/écriture, port série COM1 ou COM2
- `stdprn` : mode binaire, écriture, imprimante

### 4.4 Redirection de flux déjà ouverts

---

Redirection de flux déjà ouverts

```
1 FILE *freopen (const char *path, const char *mode, FILE *stream);
```

- Ferme le flux initial
- Ouvre le nouveau fichier en utilisant la structure de données pointée par `stream`

Remarques

- N'a d'intérêt que pour les flux standards (`stdin`, `stdout`, `stderr`)
- Permet par exemple de rediriger la sortie standard vers un fichier depuis le programme

#### ✓ L'essentiel à retenir :

1. Je sais ouvrir et fermer un flux.
2. Je sais lire et écrire dans un flux.
3. Je comprend ce qu'est la padding et ses dangers.
4. Je sais me déplacer dans un fichier.
5. Je sais tester si la fin du fichier est atteinte.

## **Quatrième partie**

**Niveau avancé**



# 14

# Compléments

## 1 Compléments sur les fonctions

### 1.1 Les pointeurs de fonctions

#### 1.1.1 Pointeurs de fonctions

##### Objectif

- appel de fonctions plus fléxible
- pouvoir transmettre le nom d'une fonction à appeler à une autre fonction

##### Règle générale

le nom d'une fonction constitue le pointeur sur la fonction (principe similaire aux tableaux)

##### Type de données

```
1 TypeRetourné (*NomDeLaVariable)(ListeDesParamètres);
```

Si Nom est une variable (locale, globale, paramètre, membre) alors on déclare une variable de type fonction

##### Typedef

```
1 typedef TypeRetourné (*NomDuType)(ListeDesParamètres);
```

Il est alors possible de déclarer une variable du type NomDuType. Une telle variable peut contenir un pointeur vers une fonction.

#### 1.1.2 Exemple 1

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int calculer1()
5 {
6 return 0;
7 }
8
9 int calculer2()
10 {
11 return rand();
12 }
13
14 void afficher1(int x)
15 {
16 printf("Tatam ! x=%d\n", x);
17 }
18
19 void afficher2(int x)
20 {
21 printf("*** %d ***\n", x);
22 }
23
24 int main(void)
25 {
26 int (*sansargument)(void);
27 void (*avecargument)(int x);
```

## 14 Compléments

```
29 sansargument = calculer1;
30 avecargument = afficher2;
31
32 avevargument(sansargument());
33
34 return 0;
35 }
```

### 1.1.3 Exemple 2

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef int (*Traitement)(int index, int v, void * data);
5
6 int appliquer(int * liste, int N, Traitement t, void * data)
7 {
8 int res = 0, i;
9 for(i = 0; i < N; ++i)
10 if (t(i, liste[i], data))
11 res = 1;
12 return res;
13 }
14
15 int plusPetit(int index, int v, void * data)
16 {
17 if (index == 0)
18 *(int*)data = v;
19 else
20 if (*(int*)data > v)
21 *(int*)data = v;
22 return 1;
23 }
24
25 int existe(int index, int v, void * data)
26 {
27 if (*(int*)data == v)
28 return 1;
29 else
30 return 0;
31 }
32
33 int afficher(int index, int v, void * data)
34 {
35 printf("%d => %d\n", index, v);
36 }
37
38 int main(int argc, char ** argv)
39 {
40 int liste[100];
41 int i, min, v;
42
43 for(i = 0; i < 100; ++i)
44 liste[i] = rand();
45
46 if (appliquer(liste, 100, plusPetit, &min))
47 printf("Le plus petit élément est : %d\n", min);
48
49 v = 25;
50 if (appliquer(liste, 100, existe, &v))
51 puts("25 est dans la liste\n");
52 else
53 puts("25 n'est pas dans la liste\n");
54
55 appliquer(liste, 100, afficher, NULL);
56
57 return 0;
58 }
```

## 1.2 Les fonctions variadiques

### 1.2.1 Fonctions à nombre d'arguments variables

## Remarque

La fonction `printf` n'a pas toujours le même nombre d'arguments. Ce n'est pas une astuce du compilateur. C'est une fonction variadique !

## Contraintes pour définir une fonction variadique

- Il faut utiliser le fichier d'entête `stdarg.h`
- La fonction doit avoir au moins 1 argument fixe

## Prototype

```
1 TypeRetourné NomFct(ListeParam , Type NomDernierParam , ...);
```

## Récupération des paramètres en nombre variable

- Déclarer une variable X du type `va_list`
- Initialiser cette variable X avec `va_start(X, NomDernierParam);`
- Récupérer l'argument courant du type `TypeArgument` et se déplacer sur l'argument suivant :

```
1 maval = va_arg(X, TypeArgument)
```

- Terminer le traitement avec `va_end(X);`

## Remarques très très importantes

- Les paramètres optionnels (...) sont promus automatiquement
- Les entiers sont systématiquement promus en `int` sauf si la taille du type est plus grande ( $\Rightarrow$  pas de conversion)
- Les réels sont systématiquement promus en `double` sauf si la taille du type est plus grande ( $\Rightarrow$  pas de conversion)
- `char, short, float` ne peuvent donc pas être utilisé avec `va_arg`

### 1.2.2 Exemples

```
1 #include <stdarg.h>
2 #include <stdio.h>
3
4 int MaxInt(int Combien , ...)
5 {
6 va_list ap;
7 int max = INT_MIN;
8 int i , cur;
9 va_start(ap , Combien);
10 for(i = 0; i < Combien; ++i)
11 {
12 cur = va_arg(ap , int);
13 if (cur > max)
14 max = cur;
15 }
16 va_end(ap);
17 return max;
18 }
19 int main(void)
20 {
21 printf("Le max est %d\n" , MaxInt(5 , 1 , 4 , 34 , 'a' , 23));
22 printf("Le 2ème max est %d\n" , MaxInt(6 , 13 , 24 , -34 , 54 , 98 , 10));
23 return 0;
24 }
```

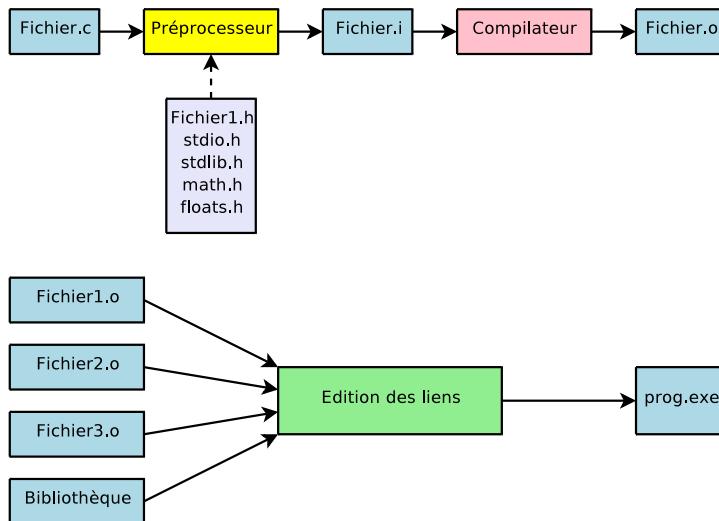
```
1 #include <stdio.h>
2 #include <stdarg.h>
3
4 void AfficherArg(int p1 , int p2 , ...)
5 {
6 int arg , count = 0;
7 va_list ap;
8 printf("%d => %d\n" , 1 , p1);
9 printf("%d => %d\n" , 2 , p2);
10 count = 3;
```

```

11 va_start(ap, p2);
12 while ((arg = va_arg(ap, int)) != -1)
13 printf("%d =>%d\n", count++, arg);
14 va_end(ap);
15 }
16 int main(void)
17 {
18 AfficherArg(1, 2, 3, 4, 5, -1);
19 AfficherArg(6, 2, 1, -2, -1);
20 return 0;
21 }
```

## 2 Le préprocesseur

### 2.1 Rappel sur la compilation



### 2.2 #define : définition de constantes

#### Objectif

Définir une constante du préprocesseur

#### Syntaxe

```
1 #define NomConstante Valeur
```

#### Principe

Toutes les occurrences du nom de la constante sont remplacées par la valeur associée. Les constantes dans les constantes sont également substituées.

#### Exemple

```

1 #define BLEU 0
2 #define ROUGE 1
3 #define VERT 2
4 #define JAUNE BLEU+VERT
5 #define NOMPROG "Le nom du programme !"
6 #define UNE_GROSSE_CONSTANTE 0xFFFFFFFFUL
7 #define UNE_TOUTE_PETITE_CONSTANTE 1.0e-300
8 #define UNE_CONSTANTTE_VIDE
```

#### Attention

les constantes ne sont pas substituées dans les chaînes de caractères.

```

1 #define XYZ this is a test
2 #define ATEST "this is a test"
3 puts("XYZ");
4 puts(ATEST);
```

## Attention

Il s'agit de substitution littérale : aucun calcul et aucune simplification ne sont effectués.

## Constantes multilignes

Les constantes longues peuvent être coupées sur plusieurs lignes (Attention les espaces en début de ligne compte !)

```

1 #define MAVALEUR \
2 1 + (\
3 2+3+4 \
4)
5 #define MA_LONGUE_CHAINE "Ceci est une chaîne très longue ! \
6 Si c'est vrai ! Elle est très longue car sa longueur est grande \
7 et non pas petite donc ce n'est pas une petite chaîne !"

```

Attention : il ne doit pas y avoir d'espace après le \

## 2.3 #define : définition de macros

### Objectif

Définir une macro du préprocesseur.

### Macro

Une macro joue le même rôle qu'une fonction :

- Elle possède des paramètres
- Elle fonctionne comme un copier-coller paramétré

### Syntaxe

```
1 #define NomMacro(Liste des paramètres) Expression
```

### Remarque

La liste des paramètres est placée derrière le nom de la macro SANS espace

```

1 #define Moitier(x) x/2
2 #define Carre(x) x*x
3 ...
4 resultat = Moitier(10);
5 resultat = Moitier(x[1] + x[2]);
6 resultat = Carre(x+y);

```

## Attention

Pour éviter les effets indésirables, les paramètres doivent être placés entre parenthèses dans l'expression

```

1 #define Moitier(x) (x)/2
2 #define Carre(x) (x)*(x)
3 ...
4 resultat = Moitier(10);
5 resultat = Moitier(x[1] + x[2]);
6 resultat = Carre(x+y);

```

```
1 #define PlusGrand(x, y) ((x)>(y)?(x):(y))
```

```

1 #include <stdio.h>
2
3 #define Carre1(x) (x)*(x)
4 double Carre2(double x)
5 {
6 return x*x;
7 }
8 double f()
9 {
10 static int compteur = 0;

```

## 14 Compléments

---

```
11 compteur++;
12 return 2;
13 }
14 int main(void)
15 {
16 double y;
17 y = Carre1(f());
18 y = Carre2(f());
19 return 0;
20 }
```

```
1 #define ABS(valeur) valeur <0?-valeur:valeur
2 ...
3 x = ABS(10-20);
```

### 2.4 #include

---

#### Objectif

Inclure le contenu du fichier spécifié à la place de l'instruction

#### Syntaxe

```
1 #include <fichier.h>
```

Inclu le contenu du fichier fichier.h se situant dans l'un des répertoires globaux de fichiers d'entête du compilateur

```
1 #include "fichier.h"
```

Inclu le contenu du fichier fichier.h se situant dans le répertoire courant

**Remarque : Un chemin de fichier complet ou partiel peut être spécifié**

- #include <malib/monmodule/fichier1.h>
- #include "c:\monrep\fichier.h"

### 2.5 Les directives de compilation conditionnelle

---

#### Syntaxe

```
1 #if condition1
2 Bloc d'instruction1
3 #elif condition2
4 Bloc d'instruction2
5 #elif condition3
6 Bloc d'instruction3
7 #else
8 Bloc d'instruction par défaut
9 #endif
```

#### Syntaxe

```
1 #if condition1
2 Bloc d'instruction1
3 #else
4 #if condition2
5 Bloc d'instruction2
6 #else
7 #if condition3
8 Bloc d'instruction3
9 #else
10 Bloc d'instruction par défaut
11 #endif
12 #endif
13 #endif
```

## Les conditions

- Toute expression constante
- N'utilisant pas `sizeof`, un cast ou un type réel (float, double, ...)<sup>1</sup>
- Fait en général intervenir des constantes symboliques

## Exemples

```

1 #if GB == 1
2 #include "GB.h"
3 #elif FR == 1
4 #include "FR.h"
5 #else
6 #include "US.h"
7 #endif

```

```

1 #if TAILLEINT == 32
2 typedef short int16;
3 typedef int int32;
4 #else
5 typedef int int16;
6 typedef long int32;
7 #endif

```

## 2.6 Tests de définition de symbole

### Syntaxe

```

1 #if defined(MONSYMBOL)
2
3 #endif

```

```

1 #ifdef MONSYMBOL
2
3 #endif

```

```

1 #if !defined(MONSYMBOL)
2
3 #endif

```

```

1 #ifndef MONSYMBOL
2
3 #endif

```

## 2.7 Compilation conditionnelle et mise au point

### Activer certaines instructions que pendant la mise au point du programme

```

1 #if DEBUG == 1
2 printf("Debug : x = %d \n", x);
3 #endif

```

Pour activer ces instructions : `#define DEBUG 1`

### Autre solution

```

1 #if defined(DEBUG)
2 printf("Debug : x = %d \n", x);
3 #endif
4 #ifdef DEBUG
5 printf("Debug : x = %d \n", x);
6 #endif

```

Pour activer ces instructions : `#define DEBUG`

1. Inconnus du préprocesseur.

## 14 Compléments

### Définition conditionnelle d'un symbole

```
1 #if !defined(TOTO)
2 #define TOTO 1
3 #endif
4 #ifndef TOTO
5 #define TOTO 1
6 #endif
```

### 2.8 Autres commandes

#### #undef : Effacer un symbole du préprocesseur

```
1 #undef DEBUG
```

#### #error : Terminer la compilation avec affichage d'un message

```
1 #error Voici le message d'erreur !
```

- Attention : pas de guillemet!!!
- D'autres informations peuvent d'afficher en plus

#### #pragma en C99

Directive permettant de donner des instructions complémentaires au compilateur (optimisation, alignement des données, ...). Attention, les paramètres de pragma sont spécifiques au compilateur et non normalisés.

#### #warning (non standard)

```
1 #warning Voici le message d'avertissement !
```

Instruction non standardisée dont le fonctionnement est similaire à **#error** et qui affiche un message d'avertissement sans arrêter la compilation.

### 2.9 LINE , FILE et #line

#### --LINE--

symbole contenant le numéro de la ligne du fichier source où est employé le symbole

#### --FILE--

symbole contenant le nom du fichier source où est employé le symbole

#### #line nombre "fichier"

spécifie le contenu de **--LINE--** et **--FILE--** pour la ligne suivante du fichier source

```
1 #include <stdio.h>
2
3 #line 100
4 int main(void)
5 {
6 printf("%d\n", __LINE__); // 102 !
7 return 0;
8 }
```

### 2.10 Les opérateurs # et ##

#### L'opérateur #

- Il s'utilisent à l'intérieur des macros **#define**
- Il transforme l'argument le suivant en chaîne de caractères

```
1 #include <stdio.h>
2
3 #define mkstr(s) # s
4
5 int main(void)
6 {
7 printf(mkstr(J'adore le C !)); // printf("J'adore le C !");
8 return 0;
9 }
```

## L'opérateur ##

concatène l'argument de gauche à l'argument de droite

```

1 #include <stdio.h>
2
3 #define concat(a, b) a ## b
4
5 int main(void)
6 {
7 int xy = 10;
8 printf("%d", concat(x, y)); // printf("%d", xy);
9 return 0;
10 }
```

## 2.11 Quelques symboles utiles

### --DATE--

La date de compilation du fichier courant sous la forme d'une chaîne de caractères

### --TIME--

L'heure de compilation du fichier courant sous la forme d'une chaîne de caractères

### --STDC--

Indique que le compilateur est (sensé être) conforme à la norme ANSI-C/C89

## 3 Etude de codes sources

### 3.1 Exemple 1

```

1 #include <stdio.h>
2
3 #define N 5
4 #define M 3
5
6 int main()
7 {
8 int a[N][M]={{1,3,7}, {5,2,6}, {3,1,4}, {2,1,6}, {5,1,9}};
9 int b[M][N]={{3,1,7,8,1}, {2,5,5,2,6}, {9,4,3,1,4}};
10 int c[N][N];
11 int i,j,k;
12
13 for (i=0; i<N; i=i+1)
14 {
15 for (j=0; j<N; j=j+1)
16 {
17 c[i][j]=0;
18 for (k=0; k<M; k=k+1)
19 c[i][j]=c[i][j]+a[i][k]*b[k][j];
20 }
21 }
22
23 for (i=0; i<N; i=i+1)
24 {
25 for (j=0; j<N; j=j+1)
26 printf("%d\t",c[i][j]);
27 printf("\n");
28 }
29 return 0;
30 }
```

## 14 Compléments

### 3.2 Exemple 2

```
1 int is_word_str(const char* s)
2 {
3 const char* i;
4
5 if(!s || !*s)
6 return 0;
7
8 for(i = s; *i; ++i) {
9 if((*i >= 'a' && *i <= 'z') ||
10 (*i >= 'A' && *i <= 'Z') ||
11 (*i >= '0' && *i <= '9') ||
12 (*i == '-') ||
13 (*i == '_'))
14 continue;
15 return 0;
16 }
17
18 return 1;
19 }
```

### 3.3 Exemple 3

```
1 void error(const char* s, ...)
2 {
3 va_list ap;
4
5 va_start(ap, s);
6 fprintf(stderr, "winegcc: ");
7 vfprintf(stderr, s, ap);
8 fprintf(stderr, "\n");
9 va_end(ap);
10 exit(2);
11 }
12 void* xmalloc(size_t size)
13 {
14 void* p;
15
16 if ((p = malloc(size)) == NULL)
17 error("Could not malloc %d bytes.", size);
18
19 return p;
20 }
21
22 int strendswith(const char* str, const char* end)
23 {
24 int l = strlen(str);
25 int m = strlen(end);
26
27 return l >= m && strcmp(str + l - m, end) == 0;
28 }
```

### 3.4 Exemple 4

```
1 const char *symbol_get_spec_type (const parsed_symbol *sym, size_t arg)
2 {
3 assert (arg < sym->argc);
4 switch (sym->arg_type [arg])
5 {
6 case ARG_STRING: return "str";
7 case ARG_WIDE STRING: return "wstr";
8 case ARG_POINTER: return "ptr";
9 case ARG_DOUBLE: return "double";
10 case ARG_STRUCT:
11 case ARG_FLOAT:
12 case ARG_LONG: return "long";
13 }
14 assert (0);
15 return NULL;
16 }
```

### 3.5 Exemple 5

```

1 void fatal (const char *message)
2 {
3 if (errno)
4 perror (message);
5 else
6 puts (message);
7 exit(1);
8 }
9
10 char *str_substring (const char *start , const char *end)
11 {
12 char *newstr;
13
14 assert (start && end && end > start);
15
16 if (!(newstr = (char *) malloc (end - start + 1)))
17 fatal ("Out of memory");
18
19 memcpy (newstr , start , end - start);
20 newstr [end - start] = '\0';
21
22 return newstr;
23 }
24
25 char *str_create (size_t num_str , ...)
26 {
27 va_list args;
28 size_t len = 1, i = 0;
29 char *tmp, *t;
30
31 va_start (args , num_str);
32 for (i = 0; i < num_str; i++)
33 if ((t = va_arg(args , char *)))
34 len += strlen (t);
35 va_end (args);
36
37 if (!(tmp = (char *) malloc (len)))
38 fatal ("Out of memory");
39
40 tmp[0] = '\0';
41
42 va_start (args , num_str);
43 for (i = 0; i < num_str; i++)
44 if ((t = va_arg(args , char *)))
45 strcat (tmp, t);
46 va_end (args);
47 return tmp;
48 }
49 char *str_replace (char *str , const char *oldstr , const char *newstr)
50 {
51 int oldlen , newlen;
52 char *p, *q;
53
54 if (!(p = strstr(str , oldstr)))
55 return p;
56 oldlen = strlen (oldstr);
57 newlen = strlen (newstr);
58 memmove (q = p + newlen , p + oldlen , strlen (p + oldlen) + 1);
59 memcpy (p, newstr , newlen);
60 return q;
61 }
62
63 const char *str_find_set (const char *str , const char *findset)
64 {
65 assert (str && findset);
66
67 while (*str)
68 {
69 const char *p = findset;
70 while (*p)
71 if (*p++ == *str)
72 return str;
73 str++;
74 }
75 }
```

```

74 }
75 return NULL;
76 }
77
78 char *str_toupper (char *str)
79 {
80 char *save = str;
81 while (*str)
82 {
83 *str = toupper (*str);
84 str++;
85 }
86 return save;
87 }
```

### 3.6 Exemple 6

---

```

1 static inline unsigned short wpa_swap_16(unsigned short v)
2 {
3 return ((v & 0xff) << 8) | (v >> 8);
4 }
5
6 static inline unsigned int wpa_swap_32(unsigned int v)
7 {
8 return ((v & 0xff) << 24) | ((v & 0xff00) << 8) |
9 ((v & 0xff0000) >> 8) | (v >> 24);
10 }
11
12 static int hex2num(char c)
13 {
14 if (c >= '0' && c <= '9')
15 return c - '0';
16 if (c >= 'a' && c <= 'f')
17 return c - 'a' + 10;
18 if (c >= 'A' && c <= 'F')
19 return c - 'A' + 10;
20 return -1;
21 }
22
23 static int hex2byte(const char *hex)
24 {
25 int a, b;
26 a = hex2num(*hex++);
27 if (a < 0)
28 return -1;
29 b = hex2num(*hex++);
30 if (b < 0)
31 return -1;
32 return (a << 4) | b;
33 }
34
35 #ifdef CONFIG_DEBUG_FILE
36 static FILE *out_file = NULL;
37 #endif /* CONFIG_DEBUG_FILE */
38
39 int wpa_debug_use_file = 0;
40 int wpa_debug_level = MSG_INFO;
41 int wpa_debug_show_keys = 0;
42 int wpa_debug_timestamp = 0;
43
44 void wpa_debug_print_timestamp (void)
45 {
46 struct os_time tv;
47
48 if (!wpa_debug_timestamp)
49 return;
50
51 os_get_time(&tv);
52 #ifdef CONFIG_DEBUG_FILE
53 if (out_file) {
54 fprintf(out_file, "%ld.%06u: ", (long) tv.sec,
55 (unsigned int) tv.usec);
56 } else
57 #endif /* CONFIG_DEBUG_FILE */
```

```

58 printf("%ld.%06u: ", (long) tv.sec, (unsigned int) tv.usec);
59 }
60
61 void wpa_printf(int level, char *fmt, ...)
62 {
63 va_list ap;
64
65 va_start(ap, fmt);
66 if (level >= wpa_debug_level) {
67 wpa_debug_print_timestamp();
68 #ifdef CONFIG_DEBUG_FILE
69 if (out_file) {
70 vfprintf(out_file, fmt, ap);
71 fprintf(out_file, "\n");
72 } else {
73 #endif /* CONFIG_DEBUG_FILE */
74 vprintf(fmt, ap);
75 printf("\n");
76 #ifdef CONFIG_DEBUG_FILE
77 }
78 #endif /* CONFIG_DEBUG_FILE */
79 }
80 va_end(ap);
81 }
82
83 int wpa_debug_open_file(void)
84 {
85 #ifdef CONFIG_DEBUG_FILE
86 static int count = 0;
87 char fname[64];
88 if (!wpa_debug_use_file)
89 return 0;
90 #ifdef _WIN32
91 snprintf(fname, sizeof(fname), "\\Temp\\wpa_supplicant-log-%d.txt",
92 count++);
93 #else /* _WIN32 */
94 snprintf(fname, sizeof(fname), "/tmp/wpa_supplicant-log-%d.txt",
95 count++);
96 #endif /* _WIN32 */
97 out_file = fopen(fname, "w");
98 return out_file == NULL ? -1 : 0;
99 #else /* CONFIG_DEBUG_FILE */
100 return 0;
101 #endif /* CONFIG_DEBUG_FILE */
102 }
103
104 void wpa_debug_close_file(void)
105 {
106 #ifdef CONFIG_DEBUG_FILE
107 if (!wpa_debug_use_file)
108 return;
109 fclose(out_file);
110 out_file = NULL;
111 #endif /* CONFIG_DEBUG_FILE */
112 }
```

### 3.7 Exemple 7

```

1 #if !defined(_MSC_VER) || _MSC_VER < 1400
2 #ifndef _MSC_VER
3 #undef memcpy
4 void *memcpy(void *dest, const void *src, size_t n)
5 {
6 unsigned char *d = dest;
7 const unsigned char *s = src;
8 while (n--)
9 *d++ = *s++;
10 return dest;
11 }
12 #endif
13 #undef memmove
14 #define memmove
15 void *memmove(void *dest, const void *src, size_t n)
16 {
```

```

17 | if (dest < src)
18 | memcpy(dest, src, n);
19 | else {
20 | /* overlapping areas */
21 | unsigned char *d = (unsigned char *) dest + n;
22 | const unsigned char *s = (const unsigned char *) src + n;
23 | while (n--)
24 | *--d = *--s;
25 | }
26 | return dest;
27 |
28
29 #ifndef _MSC_VER
30 #undef memset
31 void *memset(void *s, int c, size_t n)
32 {
33 unsigned char *p = s;
34 while (n--)
35 *p++ = c;
36 return s;
37 }
38#endif
39
40 #undef strchr
41 char *strchr(const char *s, int c)
42 {
43 while (*s) {
44 if (*s == c)
45 return (char *) s;
46 s++;
47 }
48 return NULL;
49 }
50
51 #ifndef _MSC_VER
52 #undef memcmp
53 int memcmp(const void *s1, const void *s2, size_t n)
54 {
55 const unsigned char *p1 = s1, *p2 = s2;
56
57 if (n == 0)
58 return 0;
59
60 while (*p1 == *p2) {
61 p1++;
62 p2++;
63 n--;
64 if (n == 0)
65 return 0;
66 }
67
68 return *p1 - *p2;
69 }
70#endif
71
72 #undef strrchr
73 char *strrchr(const char *s, int c)
74 {
75 const char *p = s;
76 while (*p)
77 p++;
78 p--;
79 while (p >= s) {
80 if (*p == c)
81 return (char *) p;
82 p--;
83 }
84 return NULL;
85 }
86
87 #ifndef _MSC_VER
88 #undef strcmp
89 int strcmp(const char *s1, const char *s2)
90 {
91 while (*s1 == *s2) {

```

```

92 if (*s1 == '\0')
93 break;
94 s1++;
95 s2++;
96 }
97
98 return *s1 - *s2;
99 }
100 #endif
101
102 #undef strncmp
103 int strncmp(const char *s1, const char *s2, size_t n)
104 {
105 if (n == 0)
106 return 0;
107
108 while (*s1 == *s2) {
109 if (*s1 == '\0')
110 break;
111 s1++;
112 s2++;
113 n--;
114 if (n == 0)
115 return 0;
116 }
117
118 return *s1 - *s2;
119 }
120
121 #ifndef _MSC_VER
122 #undef strlen
123 size_t strlen(const char *s)
124 {
125 const char *p = s;
126 while (*p)
127 p++;
128 return p - s;
129 }
130 #endif
131
132 #undef strcpy
133 char *strcpy(char *dest, const char *src, size_t n)
134 {
135 char *d = dest;
136
137 while (n--) {
138 *d = *src;
139 if (*src == '\0')
140 break;
141 d++;
142 src++;
143 }
144
145 return dest;
146 }
147
148 #undef strstr
149 char *strstr(const char *haystack, const char *needle)
150 {
151 size_t len = strlen(needle);
152 while (*haystack) {
153 if (strncmp(haystack, needle, len) == 0)
154 return (char *) haystack;
155 haystack++;
156 }
157
158 return NULL;
159 }
160
161 #undef strdup
162 char * strdup(const char *s)
163 {
164 char *res;
165 size_t len;
166 if (s == NULL)

```

```
167 return NULL;
168 len = strlen(s);
169 res = malloc(len + 1);
170 if (res)
171 memcpy(res, s, len + 1);
172 return res;
173 }
174 #endif /* !defined (_MSC_VER) || _MSC_VER < 1400 */
175
176 void *wpa_zalloc(size_t size)
177 {
178 void *b = malloc(size);
179 if (b)
180 memset(b, 0, size);
181 return b;
182 }
```

✓ L'essentiel à retenir :

---

1. Je sais ce qu'est un pointeur de fonction et ce que je peux faire avec.
2. Je sais ce qu'est une fonction variadique.
3. Je connais les principales instructions du préprocesseur et leurs fonctionnements respectifs.
4. Je sais lire et comprendre des codes sources complexes.

# 15

# Compilation avancée et outils

## 1 Gestion des types de données interdépendants

### 1.1 Un exemple

#### Objectifs

On souhaite définir une structure de données **Data** générique. Pour cela, un pointeur de fonction est ajouté à la structure. La fonction aura pour objectif de comparer deux structures de données. Pour des raisons de simplicité de lecture, nous choisissons de définir un type de données pour le pointeur de fonction.

On obtient les deux morceaux de code suivants

```
1 typedef int (*TCompare)(struct Data * d1, struct Data * d2);

1 struct Data
2 {
3 void * data;
4 TCompare compare;
5 }
```

### 1.2 Le problème

#### Dans un fichier d'entête

- Si TCompare est défini avant la structure Data, le compilateur se plaint que la structure Data n'est pas connue lorsque TCompare est défini
- Si la structure Data est définie avant TCompare, le compilateur se plaint que TCompare n'est pas connu lorsque la structure est définie.

Que faut-il faire ?

### 1.3 La solution

#### Ajouter une définition partielle de la structure Data

```
1 /* Il existe une structure Data */
2 struct Data;
3 /* On définit TCompare */
4 typedef int (*TCompare)(struct Data * d1, struct Data * d2);
5 /* On définit la structure */
6 struct Data
7 {
8 void * data;
9 TCompare compare;
10 }
```

#### Conditions d'utilisation

L'utilisation de déclaration partielle anticipée est utilisable à l'unique condition que la définition complète du type ne soit pas requise. C'est le cas des pointeurs ou des variables paramètres d'un prototype ou d'un type de pointeur de fonction.



# 16

# Mieux programmer

## 1 Gestion et détection d'erreurs

### 1.1 Compilation en mode debug/release

#### Mode de compilation d'un programme

Un programme peut être compilé de différentes façons. En fonction des options du compilateur qui sont activées, vous pouvez réaliser des actions différentes.

##### Mode Debug

- Les informations de débogage sont ajoutées à l'exécutable.
- Aucune optimisation du code n'est effectuée.
- L'utilisation d'un débuggeur est donc possible (pas à pas, inspection des variables...)
- Le symbole **DEBUG** du préprocesseur est en général défini conjointement avec d'autres symboles spécifiques afin d'activer des tests et vérifications supplémentaires du code.

##### Mode Release

- Les informations de débogage ne sont pas ajoutées à l'exécutable.
- Toutes les optimisations du codes sont activées.
- Un débuggeur est en général inutilisable.
- Le symbole **NDEBUG** du préprocesseur est défini et les actions optionnelles de tests et de vérifications du code sont désactivées.

#### Conditions d'utilisation

- Mode Debug : à utiliser pendant la mise au point du programme.
- Mode Release : à utiliser pour la mise en production du programme.

#### Remarque

Un programme compilé en mode Debug s'exécute beaucoup lentement que le même programme en mode Release. **C'est normal ! Il ne faut pas mettre au point en mode Release : c'est une mauvaise pratique !**

### 1.2 Mise en place de contrats avec assert

#### Principes

```
1 void assert (int expression);
```

- Lorsque le symbole **NDEBUG** n'est pas défini :
  - Si expression est évaluée comme vrai, alors l'exécution se poursuit.
  - Si expression est évaluée comme faux, alors le programme s'arrête avec une erreur d'assertion indiquant où l'assertion a échouée.
- Lorsque le symbole **NDEBUG** est défini :
  - L'expression n'est pas évaluée et l'exécution du programme se poursuit.

Dans la pratique, cette fonction est implémentée sous la forme d'une macro du préprocesseur.

#### Condition d'emploi

- S'utilise pour vérifier des invariants dans un code.
- S'utilise pour vérifier des préconditions et des postconditions (des contrats).
- **assert** ne doit pas se substituer à une gestion d'erreur car la fonction sera désactivée en mode Release.

### La condition transmise à assert ne doit pas avoir d'effet de bord

```
1 #include <assert.h>
2
3 int compteur = 0;
4
5 int tester()
6 {
7 compteur++;
8 return 1;
9 }
10 int main(int argc, char ** argv)
11 {
12 assert(tester());
13 return 0;
14 }
```

### 1.3 Gestion des erreurs avec `errno`

---

#### Les erreurs de la bibliothèque C

- Les fonctions de la bibliothèque C utilisent un mécanisme simple pour indiquer des erreurs.
- Deux cas de figure :
  - La fonction retourne une valeur indiquant qu'une erreur est survenue. La variable `errno` donne le détail de l'erreur.
  - La fonction ne peut pas indiquer qu'une erreur est survenue en retournant une valeur. Elle utilise néanmoins `errno` pour indiquer qu'il y a eu une erreur.

#### `errno`

```
1 extern int errno;
```

- la variable est positionnée à l'une des constantes `EXXX` possibles (très nombreuses, non standard et spécifique à la plateforme).
- Les constantes `EXXX` ont des valeurs non nulles.
- La variable `errno` n'est jamais mise à 0. Il est donc nécessaire de la mettre à zéro soit même.

#### Tout appel de fonction peut modifier la variable

Il ne faut donc jamais utiliser un code tel que

```
1 if (!lafonction()) {
2 // La fonction a entraînée une erreur
3 printf("Il y a eu une erreur\n");
4 if (errno == ...) { ... }
5 }
```

mais plutôt

```
1 if (!lafonction()) {
2 // La fonction a entraînée une erreur
3 int saveerrno = errno;
4 printf("Il y a eu une erreur\n");
5 if (saveerrno == ...) { ... }
6 }
```

#### Mode d'emploi

- Mettre `errno` à 0
- Appeler la fonction
- Si la fonction a signalée une erreur explicitement alors tester la valeur `errno`
- Si la fonction ne peut pas signaler une erreur explicitement alors tester la valeur `errno`

#### Affichage d'un message d'erreur

Pour afficher un message d'erreur dépendant de la valeur de `errno` :

```
1 char *strerror (int errnum);
2 void perror (const char *s);
```

## Attention

En pratique, `errno` peut ne pas être une variable (ex : une macro). Il ne faut donc pas utiliser son adresse dans un programme.

### 1.4 Valeur de retour d'un programme

#### Valeur retournée par un programme

Lorsqu'un programme se termine, il renvoie un nombre au programme appelant ou au système d'exploitation. Ce code de retour est défini par convention.

- 0 indique que le programme s'est déroulé correctement : il s'agit d'une convention
- Toutes autres valeurs peuvent indiquer une erreur ou une réussite : chaque programme définit l'interprétation qu'il faut donner à ces valeurs

#### Comment retourner une valeur spécifique

- Via l'instruction `return` dans la fonction `main`
- Via l'appel de la fonction `exit`

### 1.5 Les options de compilation

#### Utiliser les capacités du compilateur

Les compilateurs possèdent de nombreuses options activables à la compilation. Parmi celles-ci, certaines permettent d'activer un grand nombre d'avertissements. Ces avertissements permettent souvent de déceler certains bugs à la compilation. Il est donc conseillé de les activer dès le début du développement.

## 2 Les principales nouveautés du C99

### 2.1 Les nouveaux types de données

#### 2.1.1 Les entiers 64bits

`long long int` ou `long long`

| Type                            | Taille en bits | Domaine                                                                        |
|---------------------------------|----------------|--------------------------------------------------------------------------------|
| <code>long long int</code>      | 64             | <code>-9 223 372 036 854 775 807..+9 223 372 036 854 775 807</code><br>minimum |
| <code>long long</code>          |                |                                                                                |
| <code>unsigned long long</code> | 64             | <code>0..+18 446 744 073 709 551 615</code> minimum                            |

Pour définir des constantes de type `long long` ou `unsigned long long`, les suffixes **LL** et **ULL** peuvent être utilisés.

#### 2.1.2 Les réels à haute précision

`long double`

| Type                     | Taille en bits | Domaine                                                                                     |
|--------------------------|----------------|---------------------------------------------------------------------------------------------|
| <code>long double</code> | 80             | précision $\geq 10$ chiffres décimaux, $-1E+37..1E+37$ au minimum<br>$3.4E+4932..3.4E+4932$ |

#### 2.1.3 Les booléens

#### Jusqu'à la norme C99

- le type booléen n'existe pas
- les valeurs '0' et 'différent de 0' jouent le rôle de faux et vrai

## Le C99

- introduction du type `_Bool` pouvant contenir les valeurs 0 et 1
- le fichier `stdbool.h` définit le type `bool` comme étant un alias du type `_Bool`
- le fichier `stdbool.h` définit les constantes `true` (1) et `false` (0)

Remarque : les opérateurs booléens du C renvoie les valeurs 0 ou 1. Par conséquent, le type `_Bool` est peu utile donc très peu utilisé.

### 2.1.4 Les nombres complexes

---

#### Les types

```
1 float complex MaVariable; // simple précision
2 double complex MaVariable; // double précision
3 long double complex MaVariable; // précision maximale
4 MaVariable = 1 + 2 * I; // I macro utilisable pour la partie imaginaire
```

#### Les opérateurs

```
1 +, - (unaire), - (binaire), *, /, I (imaginaire pur)
```

#### Les fonctions

```
1 double creal(double complex z);
2 double cimag (double complex z);
3 double cabs(double complex z);
4 double carg(double complex z);
5 double complex cexp(double complex z);
6 double complex cexp2(double complex z);
7 double complex cexp10(double complex z);
8 double complex clog(double complex z);
9 double complex clog2(double complex z);
10 double complex clog10(double complex z);
11 double complex cproj(double complex z);
```

### 2.1.5 Le fichier d'entête stdint.h

---

#### stdint.h

Le C99 crée le fichier d'entête `stdint.h`. De nouveaux types entiers y sont définis avec pour caractéristique la taille en bits des entiers. Ces entiers ont pour objectif d'accroître la portabilité lorsque des tailles précises sont exigées. Certains sont optionnels et d'autres sont obligatoires pour respecter la norme.

#### Les types optionnels

| Type                   | Signe     | Taille en bit                |
|------------------------|-----------|------------------------------|
| <code>int8_t</code>    | Signé     | 8                            |
| <code>int16_t</code>   | Signé     | 16                           |
| <code>int32_t</code>   | Signé     | 32                           |
| <code>int64_t</code>   | Signé     | 64                           |
| <code>uint8_t</code>   | Non signé | 8                            |
| <code>uint16_t</code>  | Non signé | 16                           |
| <code>uint32_t</code>  | Non signé | 32                           |
| <code>uint64_t</code>  | Non signé | 64                           |
| <code>intptr_t</code>  | Signé     | Pouvant contenir un pointeur |
| <code>uintptr_t</code> | Non signé | Pouvant contenir un pointeur |

## Les types obligatoires

| Type                        | Signe     | Taille en bit                             |
|-----------------------------|-----------|-------------------------------------------|
| <code>intmax_t</code>       | Signé     | Taille maximale                           |
| <code>uintmax_t</code>      | Non signé | Taille maximale                           |
| <code>int_least8_t</code>   | Signé     | Au moins 8                                |
| <code>int_least16_t</code>  | Signé     | Au moins 16                               |
| <code>int_least32_t</code>  | Signé     | Au moins 32                               |
| <code>int_least64_t</code>  | Signé     | Au moins 64                               |
| <code>uint_least8_t</code>  | Non signé | Au moins 8                                |
| <code>uint_least16_t</code> | Non signé | Au moins 16                               |
| <code>uint_least32_t</code> | Non signé | Au moins 32                               |
| <code>uint_least64_t</code> | Non signé | Au moins 64                               |
| <code>int_fastN_t</code>    | Signé     | Taille N choisie pour des calculs rapides |
| <code>uint_fastN_t</code>   | Non signé | Taille N choisie pour des calculs rapides |

## Les constantes maximales et minimales

Le fichier stdint.h définit aussi les constantes représentant les valeurs minimales et maximales admissibles pour chacun de ces types.

## 2.2 Les nouveaux mots clés

### 2.2.1 inline

#### Mot clé

Le mot clé `inline` permet de qualifier une fonction.

#### Effet

Lorsqu'une fonction inline est appellée, le compilateur peut (s'il le veut) remplacer l'appel de la fonction par le code de la fonction. Cela permet d'éviter d'effectuer des appels de fonction et donc d'accélérer l'exécution du programme.

#### inline vs macro

Une fonction inline peut avantageusement remplacer une macro :

- les paramètres ne sont évalués qu'une fois à la différence d'une macro
- la lisibilité d'une fonction inline est plus grande qu'une macro

Mais :

- une fonction inline type ses paramètres et en C, il n'est pas possible de créer deux fonctions de même nom ayant des paramètres différents
- le compilateur n'est pas obligé de satisfaire le demande inline si le compilateur le juge opportun : une perte d'efficacité par rapport à une macro est possible

Remarque : il est conseillé d'utiliser le mot clé `static` avec une fonction inline de manière à éviter la définition de la fonction dans plusieurs fichiers objets et de risquer un conflit de nom de fonction.

#### Exemple

```

1 #define macrocarre(x) ((x)*(x))
2 static inline int inlinecarre(int x)
3 {
4 return x*x;
5 }
6 int main(void)
7 {
8 int a = 10;
9 double b = 10.5;
10 int resint;
11 double resdouble
12
13 resint = macrocarre(a); // 110

```

```
14 resdouble = macrocarre(b); // 110.25
15 resint = inlinecarre(a); // 110
16 resdouble = inlinecarre(b); // conversion de 10.5 en 10 => 110 !!
17 return 0;
18 }
```

### 2.2.2 restrict

---

#### Objectif

Indiquer au compilateur qu'un pointeur est l'unique pointeur paramètre de la fonction à pointer sur la zone mémoire. Grâce à cette information le compilateur peut avoir plus d'hypothèse pour effectuer une optimisation du code.

#### Exemple

Pour la fonction memcpy, l'hypothèse est faite que dst et src désignent des zones mémoires ne se recouvrant pas. Il est donc possible d'aider le compilateur en écrivant :

```
1 void *memcpy (void *restrict dst, const void *restrict src, size_t size);
```

restrict se place entre le nom de la variable et \*.

### 2.2.3 Les macros à nombre variable d'arguments

---

#### Les principes des fonctions variadiques sont généralisés aux macros variadiques

```
1 #define LOG_ERROR(format, ...) fprintf(stderr, (s), __VA_ARGS__)
```

## 2.3 Amélioration de la syntaxe

---

### 2.3.1 Les commentaires

---

#### Jusqu'en C99

Les commentaires se notent entre /\* et \*/.

#### En C99

Les commentaires courts font leur apparition.

```
1 /* Ceci est
2 un commentaire
3 long */
4 // Ceci est un commentaire court (C99)
```

#### Remarque

Plusieurs compilateurs, tel que Visual Studio, supportent les commentaires courts sans pour autant supporter la norme C99.

### 2.3.2 Mélange entre déclaration et code

---

#### Jusqu'au C99

Les déclarations de variables doivent être réalisées en début de blocs d'instructions.

#### En C99

Les déclarations de variables peuvent être effectuées à n'importe quel endroit d'un bloc d'instructions.

### Exemple : à ne pas faire

Ce type d'utilisation est à éviter car nuisible à la lisibilité du code.

```

1 int fibonacci(int n)
2 {
3 if (n <= 1)
4 return 0;
5 else
6 {
7 int res ;
8 res = fibonacci(n-1) + fibonacci(n-2);
9 return res;
10 }
11 }
```

### Exemple : tolérable

L'utilisation pour les variables de boucles peut améliorer la lisibilité.

```

1 int fibonacci(int n)
2 {
3 int fn1 = 1;
4 int fn = 0;
5 int t;
6 for(int i = 1; i <= n; ++i)
7 {
8 t = fn;
9 fn = fn + fn1;
10 fn1 = t;
11 }
12 return fn;
13 }
```

## 2.3.3 Nécessité de spécifier le type de retour d'une fonction

### Jusqu'en C99

Lorsque le type de retour d'une fonction n'est pas spécifié, le compilateur suppose qu'il est de type **int**.

### En C99

Aucune hypothèse n'est faite par le compilateur lorsque le type de retour n'est pas spécifié. Il est donc obligatoire d'indiquer un type de retour.

## 2.3.4 Variable Length Array

### Jusqu'en C99

Lors de la déclaration d'un tableau, la dimension spécifiée doit être une constante connues à la compilation.

### En C99

La dimension spécifiée peut être une variable connue à l'exécution.

```

1 void f(int taille)
2 {
3 double tab[taille];
4 ...
5 }
```

### Limitations

- Un tel tableau dynamique ne peut être défini que dans une fonction.
- Il ne peut donc pas être associé au mot clé **extern**.
- Il ne peut pas être associé au mot clé **static**.

## 2.4 La norme C99 en pratique

### Peu de compilateur supporte entièrement la norme C99

Si vous souhaitez exploiter les capacités de la norme C99, l'idéal est de se limiter aux fonctionnalités les plus couramment supportées :

- Les commentaires court //
- Le mot clé **inline**
- Les Variable Length Arrays
- Les types **long long** et **long double**

Renseignez vous sur le support de la norme C99 pa vos compilateurs

GCC : <http://gcc.gnu.org/c99status.html>

## 3 Les signaux

*Extrait de « Programmation Unix, Support de cours MIMaTS, Arnaud Puret, 2008-2009 » avec la permission de l'auteur.*

### 3.1 Rappels

L'appel système "fork" permet de créer un nouveau processus qui est la copie conforme du processus appelant. C'est le seul appel système qui peut **créer** un processus. Lors d'un "fork", le processus fils est une copie quasi-conforme du père (au PID près). Mais le fils est indépendant du père. Si l'on veut exécuter un autre programme on utilise l'appel système "exec", ce dernier remplace le processus courant par le nouveau programme, en gardant le même PID. **En aucun cas "exec" ne peut créer un nouveau processus.**

L'intérêt des processus est de pouvoir faire plusieurs travaux en même temps. Cependant, le mécanisme de création des processus est assez lourd et peut poser des problèmes de performances et de facilité de communication (pas de partage de l'espace mémoire entre deux processus). Dans ce genre de cas on peut utiliser des "processus légers", aussi appelés "Threads".

### 3.2 Les processus en détail

#### 3.2.1 Crédation et identification d'un processus

Pour créer un processus, on utilise l'appel système "fork". Les appels systèmes "getpid()" et "getppid()" permettent respectivement de connaître le PID du processus courant et le PID du processus père. Mis à part le processus INIT de PID "1", tout processus à un père. L'exemple suivant montre la création d'un nouveau processus et l'affichage des PID père et fils :

```
1 #include <unistd.h>
2 #include <stdio.h>
3
4 int main(int argc, char* argv[])
5 {
6 int pid ;
7
8 pid = fork() ;
9 if(pid!=0)
10 {
11 printf("On est dans le pere, pid %d, fils %d\n", getpid(), pid) ;
12 }
13 else
14 {
15 printf("On est dans le fils, pid %d, pere %d\n", getpid(), getppid()) ;
16 }
17
18 return(0) ;
19 }
```

#### 3.2.2 Remplacement de contexte

L'appel système "exec" remplace le contexte (zones mémoires, code exécutable) d'un processus par celui d'un autre programme. "exec" ne s'utilise pas directement, on utilise un ensemble de fonctions C qui encapsulent cette fonction. Par exemple, pour remplacer le contexte du processus courant avec la commande "ls -l /usr" on utilise :

```
1 execvp("ls" , { "ls" , "-l" , "/usr" , NULL });
```

"execvp" renvoie "-1" en cas d'échec et ne renvoie rien en cas de succès : le code n'est plus en mémoire. D'autres variantes d'"exec" sont utilisables (voir le manuel de "exec").

### 3.2.3 Mort naturelle d'un processus

"exit" termine le processus courant. Sa syntaxe est la suivante :

```
1 #include <stdlib.h>
2 void exit(int status);
```

L'argument statut est un entier qui permet d'indiquer au shell (ou au père de façon générale) qu'une erreur s'est produite. On le laisse à zéro pour indiquer une fin normale.

### 3.2.4 Synchronisation

Lorsqu'un processus se termine, il envoie le signal "SIGCHLD" à son père. Le processus fils entre alors dans l'état "zombie", c'est-à-dire que la mémoire est libérée mais que le processus apparaît toujours dans la table des processus du noyau. Pour que le processus fils soit supprimé complètement, il faut que son père lise le code de retour.

L'appel système "wait" permet de synchroniser le processus père sur la fin de ses processus fils. Lors de l'exécution de cette fonction, le code de retour du fils est lu, ce qui permet de le supprimer.

Le comportement de "wait" dépend de l'état des processus fils :

- si au moins un processus fils est dans l'état "zombie", "wait" revient immédiatement avec le code de retour du fils ;
- sinon si il y a des processus fils en cours d'exécution, "wait" attend que l'un de ses fils se termine (le code du processus père ne s'exécute plus durant ce temps) ;
- si le processus courant n'a aucun processus fils, "wait" retourne "-1".

En C on a deux variantes de l'appel système "wait" :

```
1 #include <sys/types.h>
2 #include <sys/wait.h>
3 pid_t wait(int * status);
4 pid_t waitpid(pid_t pid, int * status, int options);
```

"wait" correspond à la description précédente. "waitpid" permet d'attendre spécifiquement un des processus fils identifié par son PID.

Le paramètre "status" doit être l'adresse mémoire d'une variable de type "int". Il contient le code de retour du fils ainsi que le code du signal qui a tué le fils (s'il n'a pas été terminé par un "exit").

Il existe des macros pour faciliter le traitement du statut (extrait du manuel) :

- WIFEXITED(status) : non nul si le fils s'est terminé normalement ;
- WEXITSTATUS(status) : donne le code de retour tel qu'il a été mentionné dans l'appel "exit()" ou dans le "return" de la routine main. Cette macro ne peut être évaluée que si WIFEXITED est non nul ;
- WIFSIGNALED(status) : non nul si le fils s'est terminé à cause d'un signal non intercepté ;
- WTERMSIG(status) : donne le numéro du signal qui a causé la fin du fils. Cette macro ne peut être évaluée que si WIFSIGNALED est non nul ;
- WIFSTOPPED(status) : indique que le fils est actuellement arrêté. Cette macro n'a de sens que si l'on a effectué l'appel à "wait" avec l'option WUNTRACED ;
- WSTOPSIG(status) : donne le numéro du signal qui a causé l'arrêt du fils. Cette macro ne peut être évaluée que si WIFSTOPPED est non nul.

On peut mettre "NULL" à la place de "status" si le code de retour ne nous intéresse pas.

Si le père n'a pas à être synchronisé avec ses fils, on peut utiliser l'instruction **signal(SIGCHLD, SIG\_IGN)** pour que le signal soit ignoré (voir Section 3.3.2). Il faut éviter d'avoir des processus à l'état zombie.

### 3.2.5 Mise en sommeil d'un processus

L'appel système "sleep" permet d'endormir un processus pendant une certaine durée :

```
1 #include <unistd.h>
2 int sleep(int seconds);
```

"sleep" peut avoir de multiples intérêts, notamment le processus ne consomme pas de temps processeur pendant qu'il dort. **En aucun cas on ne doit utiliser "sleep" pour synchroniser des processus.**

## 3.3 Communication inter-processus

Créer des processus c'est bien, mais s'ils sont incapables de s'échanger des informations, l'intérêt est limité.

### 3.3.1 Les fichiers

Un moyen simple pour faire communiquer deux processus est l'utilisation de fichiers. Cela pose plusieurs problèmes. Premièrement, la communication est asynchrone, ensuite les fichiers ordinaires sont prévus pour stocker des données, enfin un fichier est écrit sur un support de stockage qui est considéré comme lent. Le principal avantage est que c'est simple à mettre en place.

### 3.3.2 Les signaux

On peut envoyer des signaux pour indiquer certaines choses à un processus. C'est notamment le cas quand on cherche à tuer un processus : on lui envoie un signal de type "SIGKILL".

Pour qu'un processus puisse recevoir des signaux, il doit configurer son "gestionnaire de signaux" grâce à la fonction "signal" :

```
1 #include <signal.h>
2 typedef void (*sighandler_t)(int);
3 sighandler_t signal(int signum, sighandler_t handler);
```

Le premier paramètre est le numéro de signal à intercepter. Le second paramètre est l'adresse de la fonction à appeler.

Si on veut que le processus s'endorme jusqu'à l'arrivée d'un signal, on utilise la fonction "pause" :

```
1 #include <unistd.h>
2 int pause(void);
```

Voici un exemple d'utilisation :

```
1 #include <unistd.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <signal.h>
5 #include <errno.h>
6
7 void traitement(int numero_signal) {
8 switch(numero_signal) {
9 case SIGINT :
10 printf("Reception d'un ctrl+c\n");
11 break ;
12 case SIGTERM :
13 printf("Reception d'une demande d'arret\n");
14 break ;
15 case SIGKILL :
16 printf("Reception d'un kill\n");
17 break ;
18 default :
19 printf("Signal inconnu\n");
20 }
21 }
22
23 int main(int argc, char* argv[]) {
24 signal(SIGINT, traitement) ; // interceptions des signaux
25 signal(SIGTERM, traitement) ;
26 signal(SIGKILL, traitement) ;
27 while(1) {
28 pause();
29 }
30 return(0);
31 }
```

Ce programme intercepte les "ctrl+c" et les "SIGTERM" (arrêt propre) et affiche un message correspondant.

Le nombre de signaux sur un système UNIX dépend de la version du système d'exploitation. Notons parmi les signaux classiques :

- SIGINT : interruption depuis le clavier ;
- SIGKILL : tuer le processus ;
- SIGSTOP : arrêt du processus ;
- SIGCONT : continuer si arrêté ;
- SIGUSR1 : signal utilisateur 1 ;
- SIGUSR2 : signal utilisateur 2 ;

– SIGCHLD : fils arrêté ou terminé.

La liste complète des signaux figure dans le manuel de "signal". Il faut bien remarquer que SIGKILL et SIGSTOP ne peuvent ni être interceptés, ni ignorés (voir programme précédent).

Pour envoyer un signal, on utilise la fonction "kill" :

```

1 #include <sys/types.h>
2 #include <signal.h>
3 int kill(pid_t pid, int sig);
```

La variable "sig" est le signal à envoyer et "pid" est le PID du processus destinataire. Si "pid" vaut "-1", alors le signal est envoyé à tous les processus fils.

### 3.3.3 Les tubes ("pipes")

On a vu en Bash l'utilisation des tubes pour transmettre des données entre deux programmes. A ce moment là on était limité à connecter la sortie standard sur l'entrée standard.

Dans un programme en C, on peut utiliser autant de tubes que l'on veut. On a à notre disposition la fonction :

```

1 #include <unistd.h>
2 int pipe(int filedes[2]);
```

Cette fonction va créer 2 descripteurs de fichiers : "filedes[0]" pour lire les données envoyées dans le tube et "filedes[1]" pour écrire les données dans le tube.

Exemple :

```

1 #include <unistd.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <signal.h>
5 #include <errno.h>
6 #include <string.h>
7
8 int main(int argc, char* argv[]) {
9 int pid ;
10 int val ;
11 int tube[2] ;
12 char buffer[255] ;
13 char buffer2[255] ;
14
15 pipe(tube) ; // creation du tube
16
17 pid = fork() ;
18 if(pid!=0) {
19 wait() ; // on attend que le fils se termine;
20 read(tube[0], buffer2, 10) ; // lecture de 10 octets dans le tube
21 printf("buffer : %s\n", buffer2) ;
22 }
23 else {
24 strcpy(buffer, "hello") ;
25 write(tube[1], buffer, 10) ; // ecriture de 10 octets dans le tube.
26 }
27 close(tube[1]) ; // fermeture des fd (dans le pere et le fils)
28 close(tube[0]) ;
29
30 return(0) ;
31}
```

## 3.4 Les processus légers ("Threads")

### 3.4.1 Processus vs thread

Nous avons vu que les processus permettent de réaliser des tâches en "parallèle". Cependant, la création d'un processus est un mécanisme lourd (il faut dupliquer les zones mémoires) et la communication inter-processus n'est pas des plus simples. Avec l'arrivée des machines multi-processeurs et les approches clients-serveurs, les processus ont montré leurs limites.

Les threads sont là pour combler le fossé entre la programmation mono-tâche (communication facile entre les différentes parties du programme) et l'utilisation de plusieurs processus (exécution en parallèle de plusieurs

travaux). Un thread ne peut pas exister en dehors d'un processus, mais un processus peut avoir plusieurs threads. Au sein d'un processus, les threads se partagent les différentes zones mémoires mais leur "fil" d'exécution est indépendant.

La création d'un thread est bien plus rapide que la création d'un processus car on n'a pas à dupliquer les zones mémoires ni à créer une nouvelle entrée dans la table des processus du noyau.

### 3.4.2 La bibliothèque "pthread"

---

A l'inverse de "fork", la création de threads ne se fait pas via un appel système mais via l'utilisation d'une fonction dans une bibliothèque (en pratique cela ne change rien). Sous Unix, une bibliothèque est "POSIX thread" abrégée en "pthread".

Pour utiliser les fonctions de "pthread", il faut inclure le fichier "pthread.h" à son code source. Il faut aussi spécifier au linker que l'on utilise cette bibliothèque avec l'option "-lpthread".

Les fonctions de cette bibliothèque sont préfixées par "pthread".

### 3.4.3 Crédation d'un thread

---

Le principe de création d'un thread est différent de celui d'un processus. Lorsque l'on crée un thread, on indique quelle sera la fonction qui sera exécutée dans le thread nouvellement créé (le point d'entrée).

Syntaxe :

```
1 int pthread_create(pthread_t *p_pid,
2 pthread_attr_t attr,
3 void *(*fonction)(void*),
4 void *arg) ;
```

La valeur "0" est renvoyée en cas de réussite et "-1" en cas d'échec.

- "p\_pid" est un pointeur vers un objet "pthread\_t" qui identifie le thread ;
- "attr" n'est plus utilisé, on met sa valeur à NULL ;
- "void \*(\*fonction)(void\*)" est l'adresse de la fonction qui sera le point d'entrée du thread. Cette fonction renvoie un pointeur générique (void\*) et prend comme argument un pointeur générique qui permet de passer tous types de variables ;
- "void \*arg" est un pointeur vers l'argument passé en paramètre à la fonction.

Pour connaître l'identificateur du thread courant, on utilise la fonction :

```
1 pthread_t pthread_self(void);
```

### 3.4.4 Fin d'un thread

---

Pour interrompre un thread, on utilise la fonction :

```
1 int pthread_exit(void *pstatus);
```

"pstatus" contient la valeur de retour du thread, lisible avec la fonction :

```
1 int pthread_join(pthread_t thread, void **value_ptr);
```

Si la fonction appelée lors du lancement du thread se termine, alors il y a un appel automatique à "pthread\_exit".

Attention : si le thread principal se termine, cela termine aussi le processus, même si d'autres threads sont en cours d'exécution.

### 3.4.5 Synchronisation sur la fin d'un thread

---

Tout comme avec les processus, on peut synchroniser les threads sur la terminaison d'un ou plusieurs threads. La fonction est :

```
1 int pthread_join(pthread_t tid, void **status);
```

Le thread appelant est mis en attente jusqu'à ce que le thread identifié par 'tid' se termine. Le paramètre "status" contient la valeur de retour du thread terminé. La valeur de retour de la fonction est "0" en cas de succès ou "-1" en cas d'échec.

### 3.4.6 Les mutex

Comme les threads se partagent les mêmes zones mémoires, il est capital de bien synchroniser les threads pour éviter les corruptions mémoires. Un objet peut nous aider à effectuer cette syncronisation : le "mutex".

Un mutex ("MUTual EXclusion") est un objet qui permet l'exclusion mutelle entre threads. On l'utilise, entre autres, pour limiter l'accès à une ressource à un seul thread à la fois. Un mutex peut être dans deux états : déverrouillé (accessible) ou verrouillé (pris par un thread). Par définition un thread ne peut pas verrouiller un mutex déjà verrouillé.

#### Création d'un mutex

Un mutex étant un objet, il doit être initialisé par la fonction :

```
1 int pthread_mutex_init(pthread_mutex *mutex_pt, pthread_mutexattr_t *attr);
```

Avant l'appel à la fonction, "mutex\_pt" doit pointer vers une zone allouée de la mémoire et "attr" doit contenir les attributs du mutex (on se contentera de mettre cette valeur à NULL pour avoir les options par défaut).

#### Réserver le mutex

Quand un thread entre dans une zone d'exclusion mutuelle (code qui ne peut être exécuté que par un unique thread à la fois) il doit verrouiller le mutex. Pour cela on a deux fonctions.

- L'appel bloquant : avec cette fonction, si le mutex est déjà verrouillé, le thread appelant est mis en attente, sinon le thread continue et le mutex est verrouillé ;

```
1 int pthread_mutex_lock(pthread_mutex *mutex_pt);
```

- L'appel non bloquant : avec cette fonction, si le mutex est déjà verrouillé, le thread appelant n'est pas mis en attente. si le mutex est déverrouillé, alors il devient verrouillé.

```
1 int pthread_mutex_trylock(pthread_mutex *mutex_pt);
```

La valeur de retour est "1" en cas de réservation réussie, "0" en cas d'échec de la réservation ou " $\neq 0$ " en cas d'erreur. Attention à ne pas accéder aux ressources critiques si le retour est 1 !

#### Relâcher le mutex

Une fois que le thread n'utilise plus la ressource critique, il faut relâcher le mutex avec la fonction :

```
1 int pthread_mutex_destroy(pthread_mutex_t *mutex_pt);
```

#### Le problème de l'interbloquage

Un interblocage ("deadlock") se produit lorsque deux threads s'attendent l'un et l'autre. Le blocage est définitif, la seule solution est l'intervention d'un utilisateur pour tuer le processus parent.

Un interblocage se produire (entre autre) lorsque deux threads réservent deux mutex dans un ordre différent. Prenons l'exemple suivant :

```
1 Thread 1 :
2 Reserver le mutex A puis le mutex B
3 programme utilisant la memoire protegee par les deux mutex
4 Relacher le mutex B puis le mutex A
5
6 Thread 2 :
7 Reserver le mutex B puis le mutex A
8 programme utilisant la memoire protegee par les deux mutex
9 Relacher le mutex A puis le mutex B
```

Un interblocage se crée lors de l'exécution parallèle des deux threads :

1. le thread 1 obtient le mutex A ;
2. le thread 2 obtient le mutex B ;
3. le thread 1 attend pour obtenir le mutex B ;
4. le thread 2 attend pour obtenir le mutex A.

Il n'y a pas de solution pour éviter tous les interblocages mais il est possible de limiter les risques. Il faut toujours acquérir les mutex dans le même ordre. Si plusieurs threads utilisent plusieurs mutex et qu'ils les réservent dans un ordre différent, un interblocage peut se produire.

La priorité des threads est aussi importante. Si un thread utilise le même mutex qu'un thread de plus basse priorité, un interblocage peut survenir<sup>1</sup>. Une solution est de ne pas utiliser de mutex entre des processus de priorités différentes.

## 4 Mesurer le temps

---

### 4.1 Généralités

---

#### 4.1.1 Temps d'horloge et temps processus

---

##### Temps d'horloge

Il s'agit de la valeur de l'horloge (au sens ordinaire) du système.

##### Temps d'horloge écoulé

C'est la différence entre deux temps d'horloge. Elle s'exprime en unité en temps (secondes, minutes, heures...).

##### Temps processus

Il s'agit de la quantité de temps du processeur utilisé par le processus pour s'exécuter. Il s'exprime généralement en tics d'horloge du processeur.

##### Temps processus écoulé

C'est la différence entre deux temps processus. Elle s'exprime en tics d'horloge du processeur.

##### Cas des systèmes monotâches

Dans un système monotâche (exécutant un unique processus à la fois), les temps d'horloge écoulés et les temps processus écoulés sont deux mesures de temps identiques.

##### Cas des systèmes multitâches

Dans un système multitâche, différents processus se partagent le processeur. Les temps d'horloge écoulés et les temps processus écoulés sont deux mesures de temps différentes.

##### Exemples

Considérons un système multitâche monoprocesseur sur lequel on exécute un processus monothreadé.

1. Le processus s'est exécuté pendant 10s et a occupé 10% du temps du processeur. La durée d'exécution du processus est :
  - 10s en temps d'horloge écoulé (de 10h03m20s à 10h03m30s)
  - 1s en temps processus écoulé ( $10s * 10\%$ )Si le système n'avait à exécuter que ce processus, il lui aurait donc fallu 1s pour le faire.
2. Si on réexécute le processus, celui-ci s'exécute pendant 15s et occupe 6.6% du temps processeur. La durée d'exécution du processus est :
  - 15s en temps d'horloge écoulé (de 10h06m10s à 10h06m25s)
  - 1s en temps processus écoulé ( $15s * 6.6\%$ )

1. ce type d'interblocage est appelé une "inversion de priorité" : comme le thread moins prioritaire a réservé le mutex, il empêche le thread plus prioritaire de s'exécuter. Le thread moins prioritaire obtient virtuellement une priorité plus élevée mais il ne peut pas être exécuté pour autant...

## Temps d'horloge écoulé vs temps processus écoulé

Pour mesurer la durée d'exécution d'un processus, il faut donc utiliser le temps processus et non pas le temps horloge.

### 4.2 Temps processus

#### 4.2.1 La fonction `clock` (C89,C99,POSIX)

##### Prototype

```
1 #include <time.h>
2 clock_t clock (void);
```

Retourne la durée en unité d'horloge du temps du processeur utilisé par le processus depuis le début de son exécution. En divisant la valeur par `CLOCKS_PER_SEC`, on obtient des secondes.

##### Remarque sur `clock`

- La fonction peut revenir à 0 environ toutes les 72 minutes. Il ne faut donc pas l'utiliser pour mesurer des temps trop longs.
- Le temps initial au début du processus n'est pas forcément 0. Il ne faut donc l'utiliser qu'en faisant des différences entre deux temps.

```
1 t1 = clock ();
2 for(i = 0; i < 100; ++i)
3 mafonction ();
4 t2 = clock ();
5 printf("Temps processus écoulé : %d", (int)((t2-t1)/CLOCKS_PER_SEC));
```

#### 4.2.2 La fonction `getrusage` (POSIX)

##### Prototype

```
1 #include <sys/time.h>
2 #include <sys/resource.h>
3 int getrusage(int who, struct rusage *usage);
```

Si `who` vaut `RUSAGE_SELF`, renvoie l'utilisation des ressources courantes par le processus dans la structure `usage` et 0 en cas de réussite.

##### La structure `rusage`

```
1 struct rusage {
2 struct timeval ru_utime; /* Temps utilisateur écoulé (POSIX) */
3 struct timeval ru_stime; /* Temps système écoulé (POSIX) */
4 long ru_maxrss; /* Taille résidente maximale */
5 long ru_ixrss; /* Taille de mémoire partagée */
6 long ru_idrss; /* Taille des données non partagées */
7 long ru_isrss; /* Taille de pile */
8 long ru_minflt; /* Demandes de pages */
9 long ru_majflt; /* Nombre de fautes de pages */
10 long ru_nswap; /* Nombre de swaps */
11 long ru_inblock; /* Nombre de lectures de blocs */
12 long ru_oublock; /* Nombre d'écritures de blocs */
13 long ru_msgrnd; /* Nombre de messages émis */
14 long ru_msgrcv; /* Nombre de messages reçus */
15 long ru_nssignals; /* Nombre de signaux reçus */
16 long ru_nvcs; /* Chgmnts de contexte volontaires */
17 long ru_nivcs; /* Chgmnts de contexte involontaires */
18 };
```

Seul `ru_utime` et `ru_stime` sont imposés par la norme POSIX. Les autres champs dépendent du système d'exploitation.

### 4.2.3 La fonction times (POSIX)

---

#### Prototype

```
1 #include <sys/types.h>
2 clock_t times(struct tms *buf);
```

Enregistre dans la structure buf les statistiques du processus.

#### La structure tms

```
1 struct tms {
2 clock_t tms_utime; /* durée utilisateur */
3 clock_t tms_stime; /* durée système */
4 clock_t tms_cutime; /* durée utilisateur des fils */
5 clock_t tms_cstime; /* durée système des fils */
6 };
```

Les durées sont exprimées en tics d'horloge. Pour obtenir des secondes, il faut les diviser par `sysconf(_SC_CLK_TCK)`.

#### Remarque

Les unités de temps de `clock` et `times` ne sont pas les mêmes.

### 4.3 Temps d'horloge

---

#### 4.3.1 Obtenir l'heure

---

##### Obtenir l'heure actuelle (POSIX)

```
1 #include <time.h>
2 time_t time(time_t *t);
```

L'heure est exprimée en seconde depuis le 01/01/1970 0h0m0s GMT. La valeur est retournée et stockée dans t si t est non NULL.

##### Obtenir l'heure actuelle (POSIX)

```
1 #include <sys/time.h>
2 int gettimeofday(struct timeval *tv, struct timezone *tz);
```

L'heure est exprimée en seconde depuis le 01/01/1970 0h0m0s GMT. tz doit être NULL.

```
1 struct timeval {
2 time_t tv_sec; /* secondes */
3 suseconds_t tv_usec; /* microsecondes */
4 };
```

#### 4.3.2 Calculer des intervalles

---

#### Prototype

```
1 double difftime (time_t timedefin, time_t timedebut);
```

Calcule la longueur en seconde de [timedefin ; timedefin].

#### 4.3.3 Conversion de date en chaînes de caractères

---

#### Prototypes

```
1 char *asctime (const struct tm *tm);
2 char *asctime_r (const struct tm *tm, char *buf);
3 char *ctime (const time_t *timep);
4 char *ctime_r (const time_t *timep, char *buf);
5 struct tm *gmtime (const time_t *timep);
6 struct tm *gmtime_r (const time_t *timep, struct tm *result);
7 struct tm *localtime (const time_t *timep);
8 struct tm *localtime_r (const time_t *timep, struct tm *result);
9 time_t mktime (struct tm *tm);
10 size_t strftime (char *s, size_t max, const char *format, const struct tm *tm);
```

## 5 Les nombres aléatoires

### 5.1 Les fonctions

#### Fichier d'entête

`stdlib.h`

#### Initialisation

- `void srand (unsigned int seed);`
- Utilisation classique :
  - `srand(time(NULL));`
  - Initialise le générateur avec une graine dépendante de l'heure de la machine.
  - N'est en généralisé réalisé qu'une seule fois dans un programme

#### Génération de nombre aléatoire

- `int rand (void);`
- Génère une nombre aléatoire dans l'intervalle  $[0; RAND\_MAX]$  selon une distribution uniforme

### 5.2 Cas particuliers

#### Générer des entiers dans l'intervalle $[0; RAND\_MAX]$

`rand()`

#### Générer des entiers dans l'intervalle $[0; MAX]$

- $(rand() * MAX) / RAND\_MAX$  ou  $(int)(MAX * (rand() / (double)RAND\_MAX))$
- Déconseillé car  $MAX$  à une probabilité inférieur d'apparaître  
 $P(0) = \dots = P(MAX - 1) \approx \epsilon \frac{MAX}{RAND\_MAX}$  et  $P(MAX) \approx \epsilon \frac{1}{RAND\_MAX}$

#### Générer des entiers dans l'intervalle $[0; MAX[$

- $(rand() * MAX) / (RAND\_MAX + 1)$  ou  $(int)(MAX * (rand() / (1. + RAND\_MAX)))$
- Uniforme :  $P(i) \approx \epsilon \frac{MAX}{MAX + RAND}$

#### Générer des réels dans l'intervalle $[0; 1]$

- `rand() / (double)RAND\_MAX`
- Ne peut générer que les réels de la forme  $\frac{1}{RAND\_MAX}$

#### Générer des entiers dans l'intervalle $[0; 1[$

- `rand() / (1. + RAND\_MAX)`
- Ne peut générer que les réels de la forme  $\frac{1}{1. + RAND\_MAX}$

### 5.3 Remarque sur la qualité du générateur aléatoire du C

#### Initialisation

- Une graine produit toujours la même séquence de nombres aléatoires
- `time` a une précision à la seconde : deux initialisations à moins d'une seconde d'intervalle produisent les mêmes séquences de nombres aléatoires

#### L'opérateur modulo % et les nombres aléatoires

- L'opérateur modulo ne permet pas d'obtenir une distribution uniforme
- `rand() % MAX` génère des nombres aléatoires dans  $[0; MAX - 1]$  mais ils ne sont pas uniformes !

#### Le générateur aléatoire standard du C

Il est connut pour ne pas être de bonne qualité. Mais il est rapide et suffisant dans de nombreux cas où la qualité n'est pas primordiale.

### Génération de nombres aléatoires

- Il faut faire attention à la façon dont vous initialisez votre générateur aléatoire
- Il faut faire attention à la précision du générateur et vérifier que les nombres à atteindre le sont effectivement (cf. réels)
- Il faut faire attention à l'uniformité de la distribution des nombres générés
- Si vous n'avez pas besoin de précision ou de respects stricts de l'uniformité de la distribution, **rand** est suffisant. Sinon, il faut utiliser un autre générateur aléatoire (ex : Mersene Twister).

#### ✓ L'essentiel à retenir :

---

1. Je connais la différence entre le mode Debug et le mode Release.
2. Je sais mettre en place des contrats avec **assert**.
3. Je sais éviter les pièges de la gestion des erreurs avec **errno**.
4. J'ai compris à quoi servait la valeur retournée par la fonction **main**.
5. Je connais les principales nouveautés du C99, celles que je peux utiliser et celles qu'il ne vaut mieux pas utiliser.
6. Je sais comment fonctionne les signaux et ce que je peux faire avec.
7. Je connais la différence entre temps d'horloge et temps processus. Je sais utiliser les mesures de temps appropriées à ce que je veux faire.
8. Je sais générer des nombres aléatoires correctement.

# 17

# Interface graphique avec GTK+

## 1 Origines et architecture générale

### 1.1 GTK+

#### GTK+

- GTK+ signifie The Gimp Tool Kit
- GTK+ est une bibliothèque d'interface graphique
- GTK+ a été créée lors de la création de The GIMP

#### Caractéristiques

- Libre (licence GNU LGPL)
- Multi-plateforme (GNU/Linux, Windows, Unix, ...)
- Ecrit en C
- Utilise le paradigme de la programmation orienté objet

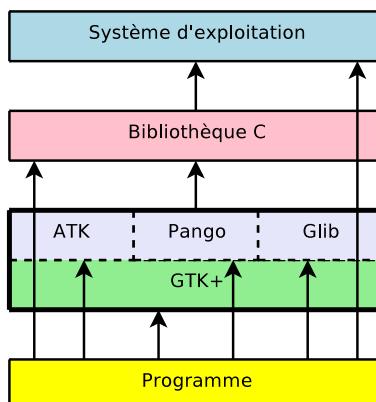
#### GTK+ ce n'est pas que pour le C

- Peut s'utiliser dans de nombreux langages
- Grâce à des bindings (C++, Perl, Ruby, Java, Python, Mono, ...)

#### GTK+ s'appuie sur trois bibliothèques principales

- GLib : bibliothèque de bas niveau (structures de données, portabilité, boucle d'événements, threads, ...)
- Pango : gestion des polices de caractères et du tracé de texte
- ATK : support de fonctionnalités pour l'accessibilité (lecteurs écrans, loupe, ...)

#### Architecture



### 1.2 Installation

#### Installation pour Visual Studio

<http://gladewin32.sourceforge.net/> Vous pouvez utiliser le script suivant pour déterminer les informations que vous devez fournir à votre IDE. Il faut néanmoins ajuster la variable **PKG\_CONFIG\_PATH** pour qu'elle soit correcte.

```
1 set PKG_CONFIG_PATH=c:\Program Files\GIMP\lib\pkgconfig
2
3 %PKG_CONFIG_PATH%\..\..\bin\pkg-config --msvc-syntax --cflags gtk+-2.0 > cflags
4 for /F "delims==" %i in ('%PKG_CONFIG_PATH%\..\..\bin\pkg-config \
5 --msvc-syntax --libs gtk+-2.0') do echo /link %i > libs
```

### Installation sous GNU/Linux et gcc

- Installez un paquet contenant GTK+
- Ou compilez et installez à partir des sources : <http://www.gtk.org/>
- Utilisez les commandes suivantes pour obtenir les informations à transmettre au compilateur et à l'éditeur des liens.

```
1 pkg-config --cflags gtk+-2.0
2 pkg-config --libs gtk+-2.0
```

## 2 Un exemple pour commencer

---

### 2.1 Un premier exemple

```
1 #include <gtk/gtk.h>
2
3 int main (int argc, char * argv[])
4 {
5 GtkWidget * window;
6
7 gtk_init (&argc, &argv);
8
9 window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
10 gtk_widget_show (window);
11
12 gtk_main ();
13
14 return 0;
15 }
```

#### Etape 1 : initialisation de GTK+

```
1 void gtk_init(gint *argc, gchar ***argv);
```

- Initialise la bibliothèque
- Supprime de argv/argc les paramètres spécifiques à GTK+

#### Etape 2 : Création d'une fenêtre

```
1 window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
2 gtk_widget_show (window);
```

- Crée une nouvelle fenêtre ordinaire (GTK\_WINDOW\_TOPLEVEL)
- Et l'affiche

#### Etape 3 : Démarrage de la boucle événement de GTK+

```
1 void gtk_main ();
```

- On se met en attente d'événements (souris, clavier, ...)

#### Résultat



### 2.2 Compilation

#### Compilation avec gcc

```
1 gcc test.c -o test `pkg-config --cflags --libs gtk+-2.0`
```

## Compilation avec Visual Studio

- Consultez la documentation !!!

## 3 Objets, Signaux et fonctions callback

### 3.1 Deuxième exemple

```

1 #include <gtk/gtk.h>
2
3 void hello (GtkWidget *widget, gpointer data)
4 {
5 g_print ("Bonjour tout le monde.\n");
6 }
7
8 gint delete_event (GtkWidget *widget, GdkEvent *event, gpointer data)
9 {
10 g_print ("le signal delete_event est survenu.\n");
11 return (FALSE); /* Ne pas tenir compte de cet événement */
12 }
13
14 void destroy (GtkWidget *widget, gpointer data)
15 {
16 gtk_main_quit ();
17 }
18
19 int main (int argc, char *argv[])
20 {
21 GtkWidget *window;
22 GtkWidget *button;
23
24 gtk_init (&argc, &argv);
25
26 window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
27
28 gtk_signal_connect (GTK_OBJECT (window), "delete_event",
29 GTK_SIGNAL_FUNC (delete_event), NULL);
30
31 gtk_signal_connect (GTK_OBJECT (window), "destroy",
32 GTK_SIGNAL_FUNC (destroy), NULL);
33
34 gtk_container_border_width (GTK_CONTAINER (window), 10);
35
36 button = gtk_button_new_with_label ("Bonjour tout le monde");
37
38 gtk_signal_connect (GTK_OBJECT (button), "clicked",
39 GTK_SIGNAL_FUNC (hello), NULL);
40
41 gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
42 GTK_SIGNAL_FUNC (gtk_widget_destroy),
43 GTK_OBJECT (window));
44
45 gtk_container_add (GTK_CONTAINER (window), button);
46
47 gtk_widget_show (button);
48 gtk_widget_show (window);
49
50 gtk_main ();
51
52 return 0;
53 }
```

### 3.2 Fonctions de rappel/callback et signaux

#### Signal

Événement déclanchant une ou plusieurs actions (click sur un bouton, ...)

#### Fonction rappel/callback

Fonction appelée lorsque survient un signal

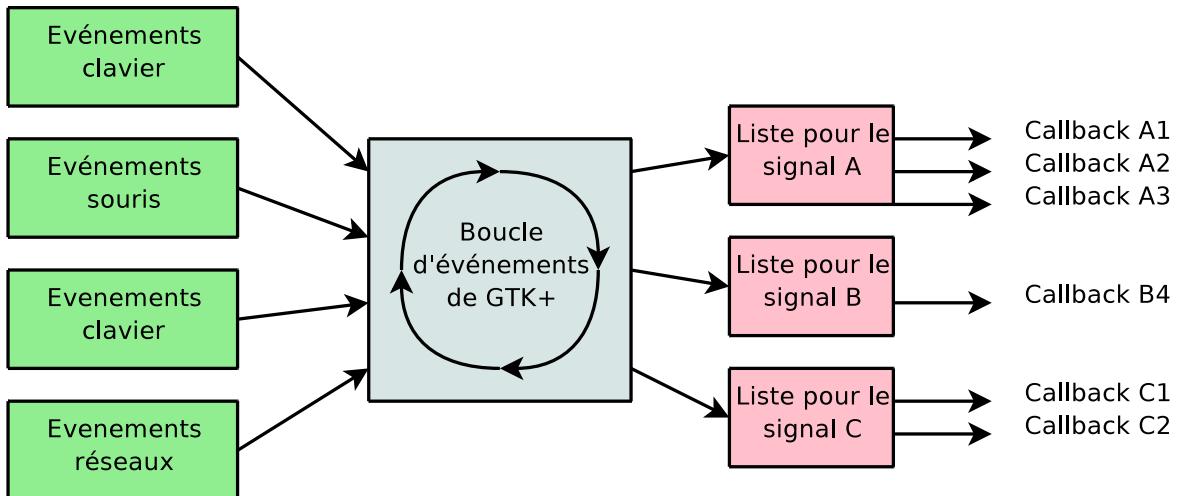
### Widget

Elément de l'interface (bouton, zone de saisie, ...)

### Remarques

- Plusieurs fonctions callback peuvent être associées à un signal
- Les fonctions sont appellées dans l'ordre de leurs ajouts
- Un widget (ou un objet) émet un signal qui appelle les fonctions callback associées

### Boucle des événements



### Connection d'une fonction callback à un signal

```

1 gint gtk_signal_connect (GtkObject *object , gchar *name ,
2 GtkSignalFunc func , gpointer func_data);

```

- object : objet/widget qui émettra le signal
- name : le nom du signal
- func : le pointeur vers la fonction callback
- func\_data : pointeur vers des données à transmettre à la fonction callback

Fonction callback :

```

1 void callback_func (GtkWidget *widget , gpointer *callback_data);

```

- widget : l'émetteur du signal
- callback\_data : le func\_data de gtk\_signal\_connect

### Alternative à gtk\_signal\_connect

```

1 gint gtk_signal_connect_object (GtkObject *object , gchar *name ,
2 GtkSignalFunc func , GtkObject *slot_object);

```

- object : objet/widget qui émettra le signal
- name : le nom du signal
- func : le pointeur vers la fonction callback
- slot\_object : l'objet/widget à transmettre à la fonction

```

1 void callback_func (GtkObject *object);

```

- Intérêt : utiliser des fonctions standards de GTK+ (ex : gtk\_widget\_destroy)

### 3.3 Gestion des signaux

#### Connecter une fonction callback à un signal

```

1 gint gtk_signal_connect (GtkObject *object , gchar *name ,
2 GtkSignalFunc func , gpointer func_data);
3 gint gtk_signal_connect_object (GtkObject *object , gchar *name ,
4 GtkSignalFunc func , GtkObject *slot_object);

```

L'entier retourné identifie la fonction callback par rapport à l'objet

## Déconnecter une fonction callback d'un objet

```
1 void gtk_signal_disconnect (GtkObject *object , gint id);
```

## Déconnecter toutes les fonctions callback d'un objet

```
1 void gtk_signal_handlers_destroy (GtkObject *object);
```

### 3.4 Troisième exemple

```

1 #include <gtk/gtk.h>
2
3 void rappel (GtkWidget *widget , gpointer *data)
4 {
5 g_print ("Re-Bonjour - %s a été pressé\n" , (char *) data);
6 }
7 void delete_event (GtkWidget *widget , GdkEvent *event , gpointer *data)
8 {
9 gtk_main_quit ();
10 }
11 int main (int argc , char *argv [])
12 {
13 GtkWidget *window;
14 GtkWidget *button;
15 GtkWidget *box1;
16
17 gtk_init (&argc , &argv);
18
19 window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
20 gtk_window_set_title (GTK_WINDOW (window) , "Salut les boutons !");
21 gtk_signal_connect (GTK_OBJECT (window) , "delete_event",
22 GTK_SIGNAL_FUNC (delete_event) , NULL);
23 gtk_container_border_width (GTK_CONTAINER (window) , 10);
24 box1 = gtk_hbox_new (FALSE, 0);
25 gtk_container_add (GTK_CONTAINER (window) , box1);
26
27 button = gtk_button_new_with_label ("Bouton 1");
28 gtk_signal_connect (GTK_OBJECT (button) , "clicked",
29 GTK_SIGNAL_FUNC (rappel) , (gpointer) "Bouton 1");
30 gtk_box_pack_start (GTK_BOX (box1) , button , TRUE, TRUE, 0);
31 gtk_widget_show (button);
32
33 button = gtk_button_new_with_label ("Bouton 2");
34 gtk_signal_connect (GTK_OBJECT (button) , "clicked",
35 GTK_SIGNAL_FUNC (rappel) , (gpointer) "Bouton 2");
36 gtk_box_pack_start (GTK_BOX (box1) , button , TRUE, TRUE, 0);
37 gtk_widget_show (button);
38
39 gtk_widget_show (box1);
40
41 gtk_widget_show (window);
42
43 gtk_main ();
44
45 return 0;
46 }
```

### 3.5 Crédit et utilisation de widget : 5 étapes

#### Les 5 étapes

1. Créer le widget via `gtk_*_new()`
2. Connexion des fonctions callback aux signaux
3. Configuration des attributs du widget
4. Placement du widget dans un container via `gtk_container_add()` ou `gtk_box_pack_start()`
5. Affichage du widget via `gtk_widget_show()`

### 3.6 Hiérarchie des objets GTK+

---

#### Paradigme de programmation de GTK+

Les objets/widgets GTK+ sont organisés selon des principes de programmation orientée object

#### Exemple

- Un GtkWidget est un GtkContainer
- Un GtkContainer est un GtkWidget
- Un GtkWidget est un GtkObjet

#### Un petit extrait de cette hiérarchie

```
1 GObject
2 GInitiallyUnowned
3 GtkObject
4 GtkWidget
5 GtkContainer
6 GtkBin
7 GtkWidget
8 GtkDialog
9 GtkAboutDialog
10 GtkColorSelectionDialog
11 GtkFileChooserDialog
12 GtkFileSelection
13 GtkInputDialog
14 GtkMessageDialog
15 GtkPlug
16 GtkAlignment
17 GtkFrame
18 GtkAspectFrame
19 GtkButton
20 GtkToggleButton
21 GtkCheckButton
```

#### Conséquences

- Un objet GTK+ peut être utilisé partout où un de ses parents peut être utilisé
  - ex : Un GtkWidget peut être transmis à la fonction `gtk_signal_connect` qui admet un GtkObject
- Un objet GTK+ hérite du comportement et des attributs de ses ancêtres
  - ex : Un GtkWidget peut contenir des objets comme un GtkContainer

### 3.7 Programmation orientée objet GTK+ et langage C

---

#### Attention

La compatibilité des types ne fait pas partie du langage C  $\Rightarrow$  Il faut le gérer soit même

#### Une solution

Les fonctions/macros de coercition (cast amélioré) :

- Vérifie le type de l'objet paramètre
- Effectue la conversion

:

#### Exemple de fonctions de coercition de GTK+

```
1 GTK_WIDGET(widget)
2 GTK_OBJECT(object)
3 GTK_SIGNAL_FUNC(function)
4 GTK_CONTAINER(container)
5 GTK_WINDOW(window)
6 GTK_BOX(box)
```

## 4 Quelques points d'entrée

### 4.1 Quelques widgets de base

#### Quelques widgets de base

- GtkWidget : une fenêtre ordinaire
- GtkDialog : une fenêtre de dialogue
- GtkHBox, GtkVBox : positionnement de widgets dans un container
- GtkButton, GtkToggleButton, GtkCheckButton, GtkRadioButton : des boutons
- GtkMenuBar, GtkMenu, GtkMenuItem : les menus
- GtkNotebook : affichage par onglets
- GtkTree : affichage d'arbres
- GtkLabel : une étiquette
- GtkFileChooserDialog, GtkColorSelectionDialog, GtkMessageDialog : des boîtes de dialogues classiques
- et plein d'autres ...

### 4.2 Aller plus loin avec GTK+

#### Références bibliographiques sur GTK+

- <http://www.gtk.org/>
- <http://www.gtk.org/tutorial/>
- <http://www.gtk.org/api/>
- [http://www.linux-france.org/article/devl/gtk/gtk\\_tut.html](http://www.linux-france.org/article/devl/gtk/gtk_tut.html)
- <http://nicolasj.developpez.com/gtk/cours/>
- <http://fr.wikipedia.org/wiki/GTK>
- <http://developer.gnome.org/doc/API/2.0/gtk/>
- <http://www.google.fr/search?q=gtk>
- [http://fr.wikibooks.org/wiki/Programmation\\_GTK2\\_en\\_C](http://fr.wikibooks.org/wiki/Programmation_GTK2_en_C)

#### L'essentiel à retenir :

1. Je connais l'architecture générale d'un programme GTK+
2. Je connais et comprend le mécanisme des callbacks et leurs rôles
3. Je suis capable de réaliser une interface minimale en GTK+



## **Cinquième partie**

**Niveau expert**



# 18

## Etes-vous prêt pour le niveau expert ?

### 1 Objectifs et principes

- Objectifs :
  - Réviser !
  - Evaluer votre maîtrise du cours !
- Principes :
  - Questionnaire à choix uniques
  - Chaque bonne réponse rapport 1 point

### 2 Questionnaire

1. Lorsqu'un programme est chargée en mémoire : de combien de zones mémoires/segments différent(e)s se compose-t-il ?
  - 2
  - 3
  - 4
  - 5
  - 6
2. Quel est le nom de chacun de ces segments ?
3. Dans quel segment sont créées les variables globales ?
4. Dans quel segment sont créées les variables dynamiques ?
5. Dans quel segment est enregistrée l'adresse de retour d'une fonction ?
6. Citer trois façons de définir des constantes
7. Quelle est la valeur de  $10 * (5\%2==1)$  ?
8. Quelle est la valeur de  $11/2$  ?
9. Quelle est la valeur de  $11./2$  ?
10. Quelle est la valeur de  $11/2+0$  ?
11. Quelle est la valeur de  $11/2+0.$  ?
12. Quelle est la valeur de  $11/(2+0.)$  ?
13. Comment code-t-on le caractère n°213 en C ?
14. Quel est l'occupation mémoire minimum d'une chaîne de caractères ?
15. Quel est l'occupation mémoire maximum d'une chaîne de caractères ?
16. Où sont les intrus ?
  - bool, char, short, integer, long
  - float, double, longdouble
  - void x ; int y ; long z ; double t ;
17. Qu'est-ce que la portée d'une variable ?
18. Qu'est-ce qu'une variable globale statique ?
19. Qu'est-ce qu'une variable locale statique ?
20. A quoi sert le mot clés extern ?
21. Trouvez les erreurs :

```
1 int x;
2
3 int main()
4 {
5 int * x;
6 int y;
7 x = &y;
8 y = 1;
9 if (y <= 2)
10 printf("%d %lf ", y, x);
11 }
```

22. A quoi sert l'opérateur sizeof?  
23. Qu'affiche la fonction suivante en fonction du paramètre ?

```
1 void mafonction(int * p)
2 {
3 if (p&&*p)
4 printf("Premier choix");
5 else
6 printf("Deuxième choix");
7 }
```

24. Qu'affiche la fonction suivante en fonction du paramètre ?

```
1 void mafonction(int x)
2 {
3 int val = 0;
4 switch (x==0)
5 {
6 case 0 : val = 1; break;
7 case 1 : val = 2;
8 case 2 : val = 3;
9 default : val = 4; break;
10 }
11 printf("%d", val)
12 }
```

25. Que fait la fonction suivante ?

```
1 void mafonction(int x, int y)
2 {
3 int z;
4 z = x;
5 x = y;
6 y = z;
7 }
```

26. Quel est la différence entre une bibliothèque statique et dynamique ?  
27. Dans quel ordre ces composants sont-ils utilisés pour créer un programme ? Quel est le rôle de chacun ?
  - Le compilateur
  - L'éditeur des liens
  - Le préprocesseur

28. Quelle doit être l'entête de la fonction mafonction pour que l'on puisse lui transmettre la fonction chercher ?

```
1 struct Liste * chercher(Liste * l, Element e);
```

29. Quel est la différence entre une structure et une union ?  
30. A quoi sert le mot clé typedef ?  
31. Avec quelles fonctions ouvre-t-on et ferme-t-on un fichier ?  
32. A quoi sert la fonction fseek ?  
33. A quoi sert la fonction rewind ?  
34. A quoi sert la fonction feof ?  
35. A quoi sert la fonction perror ?  
36. A quoi sert la fonction calloc ?

## **Sixième partie**

**Exemple complet**



# 19

## Un exemple complet

On souhaite mettre en place une structure de programmation C permettant de stocker tout types de données (entier, réel, pointeur, enregistrement, arbre, file, tableau dynamique, ...) afin par exemple de pouvoir la transmettre à une fonction. Une approche classique consisterait à utiliser une **union** ou un pointeur **void \***. L'inconvénient de ces deux méthodes est qu'elles ne permettent pas facilement à un programmeur d'ajouter de nouveau type de données ou de gérer les spécificités de chaque type de données (ex : copier une chaîne de caractères ne signifie pas copier le pointeur mais plutôt copier ce qui est pointé alors que la copie d'un entier consiste juste à copier la valeur de cet entier).

Une approche alternative consiste à concevoir tout un ensemble de fonctions/types de données extensibles à volonté. C'est ce qu'on vous propose de faire ici. A la fin de cet exercice, vous serez capable d'ajouter facilement des types de données et de définir par exemple un tableau dynamique dont les éléments peuvent être de types différents (entier, chaîne, tableau, ...).

### 1 Gestion des erreurs

Soit le fichier fatal.h suivant :

```
1 #ifndef FATAL_H_
2 #define FATAL_H_
3
4 #include <stdlib.h>
5
6 void fatal(const char * message);
7 void * fatalMalloc(size_t size);
8
9 #define FATAL_CHECK(condition, message) if (!(condition)) fatal(message);
10
11#endif /*FATAL_H_*/
```

Soit le fichier fatal.c suivant :

```
1 #include "Fatal.h"
2 #include <stdio.h>
3
4 void fatal(const char * message)
5 {
6 puts(message);
7 exit(1);
8 }
9
10 void * fatalMalloc(size_t size)
11 {
12 void * p = malloc(size);
13 if (p == NULL)
14 fatal("Not enough memory");
15 return p;
16 }
```

1. A quoi servent les fonctions **fatal** et **fatalMalloc** ?
2. Donnez un exemple d'utilisation de **FATAL\_CHECK**. Comment devrait-on modifier **fatalMalloc** pour utiliser **FATAL\_CHECK** ?

### 2 Gestion des instances et des structures

Pour pouvoir définir un système générique de gestion de types de données, il est nécessaire de différencier le type d'une donnée et ses instances particulières. Une instance est caractérisée par deux choses : le type de

## 19 Un exemple complet

données et la zone mémoire associée. Un type de données est définie par quatre informations : la taille de la zone mémoire associée à une instance, une opération permettant d'initialiser une instance, une opération permettant de désinitialiser (finaliser) une instance et une opération permettant de copier le contenu d'une instance dans une autre. Toutes ces informations peuvent être définies pour tous les types de données que pourraient gérer notre système générique.

1. Définir deux types d'enregistrements destinées à contenir les informations sur le type de données (TStructure) et sur les instances (TInstance). On s'assurera que les enregistrements «type de données» soient chainables.
2. Écrire les fonctions `int RegisterStructure(int size, TCreateMethod create, TCopyMethod copy, TDestroyMethod destroy)` et `TStructure * findStructure(int typeID)` qui respectivement permettent : (1) d'ajouter un type de données et de retourner un identifiant de ce type de données, (2) d'obtenir un pointeur vers le TInstance correspondant au type de données dont l'identifiant est fourni.
3. Écrire les fonctions `void initInstance(int typeID, TInstance * instance)` et `void destroyInstance(TInstance * instance)` qui respectivement initialisent une instance et détruisent une instance.
4. Écrire la fonction `void copyInstance(TInstance * left, TInstance * right)` qui copie le contenu d'une instance dans une autre du même type.
5. Écrire la fonction `void convertInstance(TInstance * instance, int newTypeID)` qui permet de changer le type de données associé à une instance.
6. On définit la fonction suivante :

```
1 void convertAndCopyInstance(TInstance * left, TInstance * right)
2 {
3 convertInstance(left, right->type->typeID);
4 copyInstance(left, right);
5 }
```

7. Soit le code suivant (fichiers Instance.h et Instance.c) :

```
1 #ifdef DEBUG
2 void * getInstanceData(TInstance * instance, int size);
3 #else
4 #define getInstanceData(instance, size) ((instance)->data)
5 #endif
6
7 #define InstanceAs(instance, type) (*((type*)getInstanceData(&instance, >
8 sizeof(type))))
```

```
1 #ifdef DEBUG
2 void * getInstanceData(TInstance * instance, int size)
3 {
4 if (instance->type->size != size)
5 fatal("Types incompatibles");
6 return instance->data;
7 }
8 #endif
```

Que fait la macro InstanceAs ? Donnez un exemple d'utilisation.

## 3 Les types classiques

Pour gérer les types de données classiques, on vous propose le code suivant (`StdInstance.h` et `StdInstance.c`).

```
1 #ifndef STDINSTANCE_H_
2 #define STDINSTANCE_H_
3
4 #include "Instance.h"
5
6 int defaultCreate(TInstance * instance);
7
8 void defaultDestroy(TInstance * instance);
9
10 void defaultCopy(TInstance * left, TInstance * right);
11
12 void RegisterStdInstances();
```

```

14 extern int None_TYPEID;
15 extern int char_TYPEID;
16 extern int short_TYPEID;
17 extern int int_TYPEID;
18 extern int long_TYPEID;
19
20 #define AsInt(v) (InstanceAs(v, int))
21 #define AsChar(v) (InstanceAs(v, char))
22 #define AsLong(v) (InstanceAs(v, long))
23 #define AsShort(v) (InstanceAs(v, short))
24
25 #endif /*STDINSTANCE_H_*/

```

```

1 #include "StdInstance.h"
2 #include <string.h>
3
4 int defaultCreate(TInstance * instance)
5 {
6 memset(instance->data, 0, instance->type->size);
7 return 1;
8 }
9
10 void defaultDestroy(TInstance * instance)
11 {
12 ;
13 }
14
15 void defaultCopy(TInstance * left, TInstance * right)
16 {
17 memmove(left->data, right->data, left->type->size);
18 }
19
20 int None_TYPEID;
21 int char_TYPEID;
22 int short_TYPEID;
23 int int_TYPEID;
24 int long_TYPEID;
25
26 void RegisterStdInstances()
27 {
28 None_TYPEID = RegisterStructure(0, defaultCreate, defaultCopy,
29 defaultDestroy);
30 char_TYPEID = RegisterStructure(sizeof(char), defaultCreate, defaultCopy,
31 defaultDestroy);
32 short_TYPEID = RegisterStructure(sizeof(short), defaultCreate, defaultCopy,
33 defaultDestroy);
34 int_TYPEID = RegisterStructure(sizeof(int), defaultCreate, defaultCopy,
35 defaultDestroy);
36 long_TYPEID = RegisterStructure(sizeof(long), defaultCreate, defaultCopy,
37 defaultDestroy);
38 }

```

1. Donnez un exemple de fonction **main**.

## 4 Des tableaux génériques

On souhaite pourvoir manipuler des tableaux de données génériques.

1. On vous propose d'ajouter le code suivant à **instance.h** :

```

1 #define DECLARE_INSTANCE(name) \
2 extern int name ## _TypeID; \
3 void Register ## name ## Instance();
4
5 #define IMPLEMENT_INSTANCE(name) \
6 int name ## _TypeID; \
7 static int name ## Create(TInstance * instance) \
8 { \
9 init ## name((name*)instance->data); \
10 return 1; \
11 } \
12 \
13 static void name ## Copy(TInstance * left, TInstance * right) \
14 { \

```

## 19 Un exemple complet

```
15 copy ## name((name*) left->data , (name*) right->data) ; \
16 \
17 \
18 static void name ## Destroy(TInstance * instance) \
19 { \
20 destroy ## name((name *) instance->data); \
21 } \
22 \
23 void Register ## name ## Instance() \
24 { \
25 name ## _TYPEID = RegisterStructure(sizeof(name) , name ## Create , name ## >= \
26 <- Copy , name ## Destroy); \
27 }
```

Quel est le code produit par **DECLARE\_INSTANCE(Array)** et **IMPLEMENT\_INSTANCE(Array)** ?

2. On vous propose le code suivant pour gérer des tableaux génériques (**Array.h** et **Array.c**) :

```
1 #ifndef ARRAY_H_
2 #define ARRAY_H_
3
4 #include "Instance.h"
5
6 typedef struct
7 {
8 int count;
9 TInstance * data;
10} Array;
11
12 void initArray(Array * a);
13
14 void resizeArray(Array * a, int newCount, int preserveContent);
15
16 void copyArray(Array * left, Array * right);
17
18 void destroyArray(Array * a);
19
20 #define AsArray(v) (InstanceAs(v, Array))
21
22 DECLARE_INSTANCE(Array);
23
24 #endif /*ARRAY_H_*/
```

```
1 #include "Array.h"
2 #include "StdInstance.h"
3
4 int ARRAY_TYPEID;
5
6 void initArray(Array * a)
7 {
8 a->count = 0;
9 a->data = NULL;
10}
11
12 void resizeArray(Array * a, int newCount, int preserveContent)
13 {
14 int i, typeID;
15 int min = a->count > newCount ? newCount : a->count;
16 TInstance * p = (TInstance *) malloc(sizeof(TInstance) * newCount);
17
18 for (i = 0; i < min; ++i)
19 {
20 typeID = a->data[i].type->typeID;
21 initInstance(typeID, &p[i]);
22 }
23
24 if (preserveContent)
25 for (i = 0; i < min; ++i)
26 convertAndCopyInstance(&p[i], &a->data[i]);
27
28 for (i = min; i < newCount; ++i)
29 initInstance(None_TYPEID, &p[i]);
30
31 for (i = 0; i < a->count; ++i)
```

```

32 destroyInstance(&a->data[i]);
33
34 free(a->data);
35 a->data = p;
36 a->count = newCount;
37 }
38
39 void copyArray(Array * left , Array * right)
40 {
41 int i ;
42
43 if (left->count != right->count)
44 resizeArray(left , right->count , 0);
45
46 for (i = 0; i < right->count; ++i)
47 convertAndCopyInstance(&left->data[i] , &right->data[i]);
48 }
49
50 void destroyArray(Array * a)
51 {
52 int i ;
53 for (i = 0; i < a->count; ++i)
54 destroyInstance(&a->data[i]);
55 free(a->data);
56 }
57
58 IMPLEMENT_INSTANCE(Array);

```

3. Donnez un exemple d'utilisation de ces tableaux génériques.

## 5 Des chaînes de caractères

On souhaite maintenant ajouter un nouveau type de données : la chaîne de caractère.

1. Ecrire le code nécessaire à la création de ce type de données.
2. Ecrire une fonction main utilisant ce nouveau type de données générique

## 6 Correction

### 6.1 Fatal.h

```

1 #ifndef FATAL_H_
2 #define FATAL_H_
3
4 #include <stdlib.h>
5
6 void fatal(const char * message);
7
8 void * fatalMalloc(size_t size);
9
10 #define FATAL_CHECK(condition , message) if (! (condition)) fatal(message);
11
12 #endif /*FATAL_H_*/

```

### 6.2 Fatal.c

```

1 #include "Fatal.h"
2 #include <stdio.h>
3
4 void fatal(const char * message)
5 {
6 puts(message);
7 exit(1);
8 }
9
10 void * fatalMalloc(size_t size)
11 {
12 void * p = malloc(size);
13 if (p == NULL)
14 fatal("Not enough memory");
15 return p;
16 }

```

### 6.3 Instance.h

```

1 #ifndef INSTANCE_H_
2 #define INSTANCE_H_
3
4 #include "Fatal.h"
5
6 struct TInstance_;
7
8 typedef int (*TCreateMethod)(struct TInstance_*);
9 typedef void (*TCopyMethod)(struct TInstance_*, struct TInstance_*);
10 typedef void (*TDestroyMethod)(struct TInstance_*);
11
12 typedef struct TStructure_
13 {
14 int typeID;
15 int size;
16 TCreateMethod create; // construit une instance sur la zone memoire specifiee
17 TCopyMethod copy; // recopie
18 TDestroyMethod destroy; // detruit une instance mais ne libere pas la zone memoire =>
19 // passee en parametre
20
21 struct TStructure_* next;
22 } TStructure;
23
24 int RegisterStructure(int size, TCreateMethod create, TCopyMethod copy,
25 TDestroyMethod destroy);
26
27 TStructure* findStructure(int typeID);
28
29 typedef struct TInstance_
30 {
31 TStructure* type;
32 void* data;
33 } TInstance;
34
35 void initInstance(int typeID, TInstance* instance);
36
37 TInstance* newInstance(int typeID);
38
39 void destroyInstance(TInstance* instance);
40
41 void freeInstance(TInstance** instance);
42
43 void convertInstance(TInstance* instance, int newTypeID);
44
45 void copyInstance(TInstance* left, TInstance* right);
46
47 void convertAndCopyInstance(TInstance* left, TInstance* right);
48
49 #ifdef DEBUG
50 void* getInstanceData(TInstance* instance, int size);
51 #else
52 #define getInstanceData(instance, size) ((instance)->data)
53 #endif
54
55 #define InstanceAs(instance, type) (* (type*) getInstanceData(&instance, =>
56 sizeof(type)))
57
58 #define DECLARE_INSTANCE(name) \
59 extern int name ## _TYPEID; \
60 void Register ## name ## Instance();
61
62 #define IMPLEMENT_INSTANCE(name) \
63 int name ## _TYPEID; \
64 static int name ## Create(TInstance* instance) \
65 { \
66 init ## name((name*)instance->data); \
67 return 1; \
68 } \
69 static void name ## Copy(TInstance* left, TInstance* right) \
70 { \
71 copy ## name((name*)left->data, (name*)right->data); \
72 }

```

```

72 \
73 static void name ## Destroy(TInstance * instance) \
74 { \
75 destroy ## name((name *) instance->data); \
76 } \
77 \
78 void Register ## name ## Instance() \
79 { \
80 name ## _TYPEID = RegisterStructure(sizeof(name), name ## Create, name ## Copy, \
81 &name ## Destroy); \
82 } \
83 #endif /*INSTANCE_H*/

```

## 6.4 Instance.c

```

1 #include "Instance.h"
2 #include <string.h>
3
4 static TStructure * firstStructure = NULL;
5
6 int RegisterStructure(int size, TCreateMethod create,
7 TCopyMethod copy, TDestroyMethod destroy)
8 {
9 int typeID = (firstStructure != NULL)?firstStructure->typeID+1:0;
10 TStructure * p = (TStructure *) malloc(sizeof(TStructure));
11 FATAL_CHECK(p != NULL, "Out of memory");
12
13 p->typeID = typeID;
14 p->size = size;
15 p->create = create;
16 p->copy = copy;
17 p->destroy = destroy;
18 p->next = firstStructure;
19 firstStructure = p;
20 return typeID;
21 }
22
23 TStructure * findStructure(int typeID)
24 {
25 TStructure * p = firstStructure;
26 while (p != NULL && typeID != p->typeID)
27 p = p->next;
28
29 return p;
30 }
31
32 void initInstance(int typeID, TInstance * instance)
33 {
34 instance->type = findStructure(typeID);
35 FATAL_CHECK(instance->type != NULL, "Cannot find the TStructure");
36 if (instance->type->size > 0)
37 {
38 instance->data = malloc(instance->type->size);
39 FATAL_CHECK(instance->data != NULL, "Out of memory");
40 }
41 else
42 instance->data = NULL;
43 FATAL_CHECK(instance->type->create(instance), "Create return an error : cannot \
44 & create the instance");
45 }
46
47 TInstance * newInstance(int typeID)
48 {
49 TInstance * p = (TInstance *) malloc(sizeof(TInstance));
50 FATAL_CHECK(p != NULL, "Cannot malloc an instance");
51 initInstance(typeID, p);
52 return p;
53 }
54
55 void destroyInstance(TInstance * instance)
56 {
57 FATAL_CHECK(instance != NULL, "Cannot destroy a NULL instance");
58 instance->type->destroy(instance);

```

## 19 Un exemple complet

```
58 if (instance->data != NULL)
59 free (instance->data);
60 }
61
62 void freeInstance(TInstance ** instance)
63 {
64 if (*instance != NULL)
65 {
66 destroyInstance(*instance);
67 free(instance);
68 *instance = NULL;
69 }
70 }
71
72 void convertInstance(TInstance * instance, int newTypeID)
73 {
74 if (instance->type->typeID != newTypeID)
75 {
76 destroyInstance(instance);
77 initInstance(newTypeID, instance);
78 }
79 }
80
81 void copyInstance(TInstance * left, TInstance * right)
82 {
83 FATAL_CHECK(left != NULL, "Cannot copy NULL instance");
84 FATAL_CHECK(right != NULL, "Cannot copy NULL instance");
85 FATAL_CHECK(left->type == right->type, "Cannot copy instances that are not similar");
86 left->type->copy(left, right);
87 }
88
89 void convertAndCopyInstance(TInstance * left, TInstance * right)
90 {
91 convertInstance(left, right->type->typeID);
92 copyInstance(left, right);
93 }
94
95 #ifdef DEBUG
96 void * getInstanceData(TInstance * instance, int size)
97 {
98 if (instance->type->size != size)
99 fatal("Types incompatible");
100 return instance->data;
101 }
102#endif
```

## 6.5 StdInstance.h

```
1 #ifndef STDINSTANCE_H_
2 #define STDINSTANCE_H_
3
4 #include "Instance.h"
5
6 int defaultCreate(TInstance * instance);
7
8 void defaultDestroy(TInstance * instance);
9
10 void defaultCopy(TInstance * left, TInstance * right);
11
12 void RegisterStdInstances();
13
14 extern int None_TYPEID;
15 extern int char_TYPEID;
16 extern int short_TYPEID;
17 extern int int_TYPEID;
18 extern int long_TYPEID;
19
20 #define AsInt(v) (InstanceAs(v, int))
21 #define AsChar(v) (InstanceAs(v, char))
22 #define AsLong(v) (InstanceAs(v, long))
23 #define AsShort(v) (InstanceAs(v, short))
24
25#endif /*STDINSTANCE_H_*/
```

## 6.6 StdInstance.c

```

1 #include "StdInstance.h"
2 #include <string.h>
3
4 int defaultCreate(TInstance * instance)
5 {
6 memset(instance->data, 0, instance->type->size);
7 return 1;
8 }
9
10 void defaultDestroy(TInstance * instance)
11 {
12 ;
13 }
14
15 void defaultCopy(TInstance * left, TInstance * right)
16 {
17 memmove(left->data, right->data, left->type->size);
18 }
19
20 int None_TYPEID;
21 int char_TYPEID;
22 int short_TYPEID;
23 int int_TYPEID;
24 int long_TYPEID;
25
26 void RegisterStdInstances()
27 {
28 None_TYPEID = RegisterStructure(0, defaultCreate, defaultCopy,
29 defaultDestroy);
30 char_TYPEID = RegisterStructure(sizeof(char), defaultCreate, defaultCopy,
31 defaultDestroy);
32 short_TYPEID = RegisterStructure(sizeof(short), defaultCreate, defaultCopy,
33 defaultDestroy);
34 int_TYPEID = RegisterStructure(sizeof(int), defaultCreate, defaultCopy,
35 defaultDestroy);
36 long_TYPEID = RegisterStructure(sizeof(long), defaultCreate, defaultCopy,
37 defaultDestroy);
38 }
```

## 6.7 Array.h

```

1 #ifndef ARRAY_H_
2 #define ARRAY_H_
3
4 #include "Instance.h"
5
6 typedef struct
7 {
8 int count;
9 TInstance * data;
10 } Array;
11
12 void initArray(Array * a);
13
14 void resizeArray(Array * a, int newCount, int preserveContent);
15
16 void copyArray(Array * left, Array * right);
17
18 void destroyArray(Array * a);
19
20 #define AsArray(v) (InstanceAs(v, Array))
21
22 DECLARE_INSTANCE(Array);
23
24 #endif /*ARRAY_H_*/
```

## 6.8 Array.c

## 19 Un exemple complet

---

```
1 #include "Array.h"
2 #include "StdInstance.h"
3
4 int ARRAY_TYPEID;
5
6 void initArray (Array * a)
7 {
8 a->count = 0;
9 a->data = NULL;
10 }
11
12 void resizeArray (Array * a, int newCount, int preserveContent)
13 {
14 int i, typeID;
15 int min = a->count > newCount ? newCount : a->count;
16 TInstance * p = (TInstance *) malloc(sizeof(TInstance) * newCount);
17
18 for (i = 0; i < min; ++i)
19 {
20 typeID = a->data[i].type->typeID;
21 initInstance(typeID, &p[i]);
22 }
23
24 if (preserveContent)
25 for (i = 0; i < min; ++i)
26 convertAndCopyInstance(&p[i], &a->data[i]);
27
28 for (i = min; i < newCount; ++i)
29 initInstance(None_TYPEID, &p[i]);
30
31 for (i = 0; i < a->count; ++i)
32 destroyInstance(&a->data[i]);
33
34 free(a->data);
35 a->data = p;
36 a->count = newCount;
37 }
38
39 void copyArray (Array * left, Array * right)
40 {
41 int i;
42
43 if (left->count != right->count)
44 resizeArray(left, right->count, 0);
45
46 for (i = 0; i < right->count; ++i)
47 convertAndCopyInstance(&left->data[i], &right->data[i]);
48 }
49
50 void destroyArray (Array * a)
51 {
52 int i;
53 for (i = 0; i < a->count; ++i)
54 destroyInstance(&a->data[i]);
55 free(a->data);
56 }
57
58 IMPLEMENT_INSTANCE(Array);
```

### 6.9 MyString.h

---

```
1 #ifndef MYSTRING_H_
2 #define MYSTRING_H_
3
4 #include "Instance.h"
5
6 typedef struct
7 {
8 int len;
9 int maxlen;
10 char * buf;
11 } String;
```

```

13 void initString(String * s);
14 void destroyString(String * s);
15 void copyString(String * s1, String * s2);
16 void copyCString(String * s1, char * s2);
17 void pushBack(String * s, char c);
18 void concatString(String * s1, String * s2);
19 DECLARE_INSTANCE(String);
20
21 #define AsString(v) (InstanceAs(v, String))
22
23 #endif /*MYSTRING_H*/

```

## 6.10 MyString.c

```

1 #include <string.h>
2 #include <stdlib.h>
3 #include "Fatal.h"
4 #include "MyString.h"
5
6 void initString(String * s)
7 {
8 s->len = 0;
9 s->maxlen = 0;
10 s->buf = NULL;
11 }
12
13 void destroyString(String * s)
14 {
15 if (s->buf != NULL)
16 free(s->buf);
17 s->len = 0;
18 s->maxlen = 0;
19 s->buf = NULL;
20 }
21
22 // attention ne modifie pas le contenu de la chaine i.e. les '\0'
23 void resizeString(String * s, int size)
24 {
25 if (s->maxlen >= size+1)
26 s->len = size;
27 else
28 {
29 s->buf = (char*) realloc(s->buf, size+1);
30 FATAL_CHECK(s->buf != NULL || size+1 == 0, "Out of memory");
31 s->maxlen = size+1;
32 s->len = size;
33 }
34 }
35
36 void copyString(String * s1, String * s2)
37 {
38 resizeString(s1, s2->len);
39 memcpy(s1->buf, s2->buf, s2->len+1);
40 }
41
42 void copyCString(String * s1, char * s2)
43 {
44 resizeString(s1, strlen(s2));
45 memcpy(s1->buf, s2, s1->len+1);
46 }
47
48 void pushBack(String * s, char c)
49 {
50 resizeString(s, s->len+1);
51 s->buf[s->len-1] = c;
52 s->buf[s->len] = '\0';
53 }

```

## 19 Un exemple complet

```
55 void concatString(String * s1, String * s2)
56 {
57 int oldlen = s1->len;
58 resizeString(s1, s1->len + s2->len);
59 memcpy(s1->buf + oldlen, s2->buf, s2->len);
60 s1->buf[s1->len] = '\0';
61 }
62
63 IMPLEMENT_INSTANCE(String);
```

### 6.11 main.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #include "Fatal.h"
6 #include "Instance.h"
7 #include "StdInstance.h"
8 #include "Array.h"
9 #include "MyString.h"
10
11 void f(void)
12 {
13 char tmp[100];
14 int i;
15 TInstance collection;
16 TInstance * ins;
17
18 initInstance(Array_TYPEID, &collection);
19 resizeArray(&AsArray(collection), 5, 0);
20
21 convertInstance(&AsArray(collection).data[0], char_TYPEID);
22 convertInstance(&AsArray(collection).data[1], short_TYPEID);
23 convertInstance(&AsArray(collection).data[2], int_TYPEID);
24 convertInstance(&AsArray(collection).data[3], long_TYPEID);
25 convertInstance(&AsArray(collection).data[4], Array_TYPEID);
26
27 resizeArray(&AsArray(AsArray(collection).data[4]), 10, 0);
28
29 for (i = 0; i < 10; ++i)
30 {
31 ins = &AsArray(AsArray(collection).data[4]).data[i];
32 convertInstance(ins, int_TYPEID);
33 AsInt(*ins) = i;
34 }
35
36 printf("[0] = %c\n[1] = %hd\n[2] = %d\n[3] = %ld\n",
37 AsChar(AsArray(collection).data[0]),
38 AsShort(AsArray(collection).data[1]),
39 AsInt(AsArray(collection).data[2]),
40 AsLong(AsArray(collection).data[3]));
41 for (i = 0; i < 10; ++i)
42 {
43 ins = &AsArray(AsArray(collection).data[4]).data[i];
44 printf("[4][%d] = %d\n", i, AsInt(*ins));
45 }
46
47 for (i = 0; i < 10; ++i)
48 {
49 ins = &AsArray(AsArray(collection).data[4]).data[i];
50 convertInstance(ins, String_TYPEID);
51 sprintf(tmp, "String #%d", i);
52 copyCString(&AsString(*ins), tmp);
53 }
54
55 for (i = 0; i < 10; ++i)
56 {
57 ins = &AsArray(AsArray(collection).data[4]).data[i];
58 puts(AsString(*ins).buf);
59 }
60 }
61
62 int main(void)
```

```
63 {
64 RegisterStdInstances();
65 RegisterArrayInstance();
66 RegisterStringInstance();
67
68 TInstance v;
69 initInstance(short_TYPEID, &v);
70
71 InstanceAs(v, short) = 10;
72 InstanceAs(v, short)++;
73
74 destroyInstance(&v);
75
76 f();
77
78 return 0;
79 }
```



## **Septième partie**

### **Exercices complémentaires**



# 20

# Les bases du langage C (structures de contrôle, fonctions, paramètres, syntaxe, .h)

## 1 Exercices sur les nombres

### 1.1 Troncature d'un nombre réel

Soit le nombre réel  $x$  de type double.

- Écrire la fonction **Troncature** qui renvoie un entier long, formé à partir du réel  $x$ , tronqué de sa partie décimale. Si le résultat de la troncature ne se trouve pas dans l'intervalle de définition du type **long int**, la fonction retourne -1. Écrire la fonction **main**.
- Écrire la fonction **Troncature2** similaire à la fonction **Troncature** mais qui retourne deux valeurs : la résultat de la troncature et un indicateur d'erreur booléen. Écrire la fonction **main**.

### 1.2 Arrondi d'un nombre réel

Soit  $x$  un réel de type double.

- Écrire la fonction **Arrondi** qui retourne deux valeurs :
  - Un entier long contenant l'entier le plus proche de  $x$ . Lorsque la partie décimale est égale à 0.5, la valeur renournée est arrondie à l'entier dont la valeur absolue est la plus grande.
  - Un indicateur d'erreur booléen qui est positionné lorsque le résultat n'est pas dans l'intervalle de valeur d'un entier long.
- Écrire la fonction **main**

### 1.3 Suppression de toutes les occurrences d'un caractère dans une chaîne de caractères

On considère une chaîne de caractères  $s$ . Soit  $c$  un caractère choisi arbitrairement. Écrire une fonction qui supprime dans la chaîne  $s$  toutes les occurrences du caractère  $c$  et renvoie la chaîne modifiée. On s'attachera à ne pas dupliquer le contenu de la chaîne  $s$ .

### 1.4 Suppression des caractères communs à deux chaînes de caractères

Soient deux chaînes de caractères  $s1$  et  $s2$ . On veut supprimer de la chaîne  $s1$  tous les caractères communs à  $s1$  et  $s2$ . Écrire une fonction qui admet comme argument les deux chaînes de caractères  $s1$  et  $s2$  et renvoie la chaîne  $s1$  modifiée. La chaîne initiale  $s1$  contiendra le résultat du traitement effectué. On veillera à ne pas dupliquer les chaînes en mémoire. On supposera que la longueur de  $s2$  est nettement inférieure à celle de  $s1$ . Ecrire une fonction qui est plus efficace que plusieurs appels à la fonction de l'exercice précédent.

### 1.5 Concaténation de deux chaînes de caractères

Soient deux chaînes de caractères  $s1$  et  $s2$ . Concaténer  $s2$  à  $s1$  consiste à ne former qu'une seule chaîne de caractères qui contient tous les caractères de  $s1$  et  $s2$ .

- Écrire une fonction qui admet comme paramètres les deux chaînes  $s1$  et  $s2$ .
- La chaîne  $s2$  ne doit pas pouvoir être modifiée.
- La chaîne  $s1$  constitue la destination tandis que  $s2$  constitue la source de la concaténation. Le résultat est donc placé dans  $s1$ , dont la taille doit être suffisante pour contenir l'ensemble des caractères de  $s1$  et  $s2$ . Si la concaténation s'effectue avec succès, la fonction renvoie la chaîne  $s1$ . Dans le cas contraire le pointeur NULL est renournée.

## 2 Exercices sur les tableaux

Remarque : dans cette partie, tous les tableaux manipulés contiennent des entiers de type **int**.

## 2.1 Affectation pseudo-aléatoire des éléments d'un tableau d'entiers

---

Soit un tableau d'entiers qui peut contenir n valeurs. On souhaite initialiser tous les éléments de ce tableau en utilisant un générateur de nombres pseudo-aléatoires.

Écrire une fonction qui accepte comme argument le tableau dont on veut initialiser les éléments, le domaine de valeur des éléments à générer et qui renvoie ce tableau. Ecrire une fonction **main** dans laquelle on prévoira d'initialiser ou non le générateur de nombres pseudo-aléatoires avec une valeur donnée ou avec l'heure. Le choix se fera par une saisie au clavier.

Remarques :

- Initialisation : **void srand(unsigned seed);** <stdlib.h>
- Initialisation automatique : **srand( (unsigned)time( NULL ));** avec <time.h>, <stdlib.h>
- Loi uniforme générant des entiers sur [0, RAND\_MAX] : **int rand(void);** <stdlib.h>
- Générer un nombre sur [0, x] : **x \* (double) rand() / (double) RAND\_MAX**, ATTENTION au cast.

## 2.2 Recherche de la valeur max au sein d'un tableau

---

Écrire une fonction qui permet de rechercher au sein d'un tableau d'entiers le plus grand élément. Cette fonction devra permettre de connaître la plus grande valeur ainsi que l'indice correspondant de ce plus grand élément.

## 2.3 Présence / absence d'un élément au sein d'un tableau

---

Écrire une fonction qui permet de savoir si au sein d'un tableau d'entiers, l'élément e est présent ou non. Dans le cas où il est présent, la fonction renvoie la valeur 1, dans le cas contraire la valeur 0.

## 2.4 Recherche du 2ème plus petit élément au sein d'un tableau (exercice complémentaire)

---

Écrire une fonction qui renvoie la deuxième plus petite valeur d'un tableau d'entiers et son indice.

## 3 Questions diverses

---

### 3.1 Question 1

---

Quelles sont les valeurs prises par i et j tout au long de cette fonction ?

```
1 void f(void)
2 {
3 int i = 0;
4 int j = 1;
5
6 j = ++i;
7 j = i++;
8 }
```

### 3.2 Question 2

---

Quelles sont les valeurs prises par \*p et c ?

```
1 void f(void)
2 {
3 char s[] = "abcd";
4 char * p = s+1;
5 char c;
6 c = *(p++);
7 c = *(++p);
8 }
```

### 3.3 Question 3

---

Quels sont le type et la taille de ces deux constantes : '\0' et "\0" ?

**3.4 Question 4**

Quel est le contenu de buf à la fin de la fonction ?

```

1 void f(void)
2 {
3 int buf[20];
4 int i;
5
6 buf[0] = 1;
7 for(i = 1; i < 20; ++i)
8 buf[i] = buf[i-1]++;
9 }
```

**3.5 Question 5**

Quel est le contenu de buf à la fin de la fonction ?

```

1 void f(void)
2 {
3 int buf[20];
4 int i;
5
6 buf[0] = 1;
7 for(i = 1; i < 20; ++i)
8 buf[i] = ++buf[i-1];
9 }
```

**3.6 Question 6**

Quelles sont les valeurs de i, j, k et l à la fin de la fonction ?

```

1 void f(void)
2 {
3 int i, j, k, l;
4
5 l = 0;
6 for(i=1, j=1, k=i+j-1; ((k < 20) && (j<5)); i = j+1, j=k++, k=++k -1)
7 l++;
8 }
```

**3.7 Question 7**

Quelles sont les valeurs de c1, c2, c3 et c4 ?

```

1 void f(void)
2 {
3 double c1 = 3.0*(2/3);
4 double c2 = 3*(2.0/3);
5 double c3 = 3*(2/3.0);
6 double c4 = 3*(2/3);
7 }
```

**3.8 Question 8**

La fonction suivante a été écrite pour calculer la suite de Fibonacci définie par :

$$f(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ f(n-1) + f(n-2) & \text{si } n > 1 \end{cases}$$

```

1 int f(int n)
2 {
3 int v;
4 switch (n)
5 {
6 case 0 : v = 0;
7 case 1 : v = 1;
8 default: v = f(n-1) + f(n-2);
9 }
10 return v;
11 }
```

Ce programme possède des erreurs : quelles sont-elles ? Corrigez le programme.

### 3.9 Question 9

---

Que fait le programme suivant ?

```
1 void f()
2 {
3 int i = 0;
4 int j = 1;
5
6 while (i < 100)
7 {
8 if (j == i+1)
9 j = i++;
10 }
11 }
```

### 3.10 Question 10

---

Quelles sont les valeurs prises par i ?

```
1 void f()
2 {
3 int i;
4 int j = 10;
5 int k = 2;
6
7 i = j & k;
8 i = j && k;
9 i = j | k;
10 i = j || k;
11 i = !j;
12 }
```

# 21

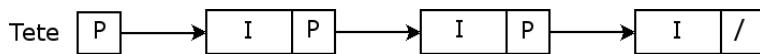
# Types complexes et allocation mémoire

## 1 Les listes chaînées

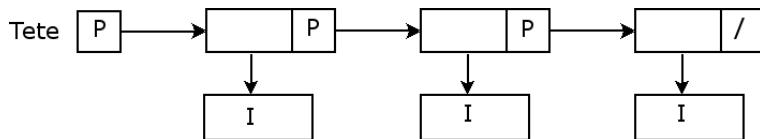
### 1.1 Structure des données utilisées

Une liste simplement chaînée est constituée :

- d'une adresse ou d'un pointeur sur le premier élément de la liste ;
- d'une adresse de fin NULL ;
- d'un ensemble de couples (I, P) où :
- I : information simple ou structurée ;
- P : pointeur sur le couple suivant (I, P).



Afin de garder une structure de liste tout à fait générale, le champs d'information I sera accessible par un pointeur. La structure de liste simple manipulée par la suite sera donc :



L'information manipulée I sera constituée des champs suivants :

- Nom : chaîne de caractères
- Prénom : chaîne de caractères
- Âge : entier
- Ville : chaîne de caractères.

### 1.2 Opérations sur listes simplement chaînées

#### 1.2.1 Création des types de données

Écrire *la/les définition(s)* nécessaire(s) à l'utilisation d'un élément d'une liste.

#### 1.2.2 Saisie des champs d'information de I

On dispose au préalable du pointeur sur le champ d'information I. Écrire la fonction *saisie* qui admet comme paramètre le pointeur sur I.

#### 1.2.3 Affichage des champs d'information de I

Écrire la fonction *affiche* qui admet comme paramètre le pointeur sur I. Cette fonction permet d'afficher les nom, prénom, âge, et ville du champ I considéré.

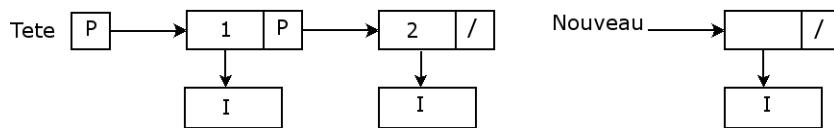
#### 1.2.4 Création d'un élément

Écrire la fonction *creeElement* qui crée un nouvel élément avec les informations correspondantes et qui retourne un pointeur vers ce nouvel élément.

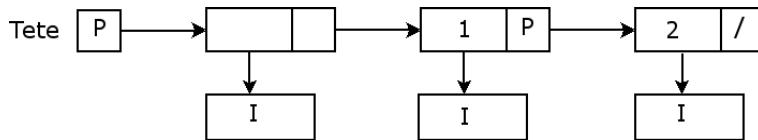
## 21 Types complexes et allocation mémoire

### 1.2.5 Insertion d'un élément en tête de liste

On veut insérer un nouvel élément accessible par le pointeur nouveau selon le schéma ci-après :



Après insertion on aura donc :



Écrire la fonction *insereTeteListe* qui admettra comme paramètres le pointeur sur le 1er élément de la liste ainsi que le pointeur sur l'élément à insérer.

### 1.2.6 Retrait du premier élément de la liste

Écrire la fonction *supprimeTeteListe* qui permet de retirer l'élément situé en tête de liste. Cette fonction admet comme seul paramètre le pointeur sur le 1er élément de la liste.

### 1.2.7 Insertion d'un élément à un emplacement quelconque

On souhaite pouvoir insérer un nouvel élément en n'importe quel point de la liste. Le point d'insertion est caractérisé par le pointeur p. L'insertion s'effectue après l'élément pointé par p. Ecrire la fonction *insereListe* dont les arguments seront le pointeur sur l'emplacement dans la liste (le pointeur p) ainsi que le pointeur sur le nouvel élément à insérer.

### 1.2.8 Retrait d'un élément à un emplacement quelconque

Le point de retrait de l'élément est caractérisé par le pointeur p. On effectue la suppression de l'élément pointé par p. Ecrire la fonction *supprimeListe* dont les arguments sont le pointeur p et le pointeur sur le premier élément de la liste.

### 1.2.9 Parcours de la liste

A partir du début de la liste, on veut se positionner sur le ième élément de la liste. Si i est supérieur à la taille de la liste, on se positionne sur le dernier élément. Ecrire la fonction *parcoursListe* dont les arguments sont le pointeur sur le premier élément ainsi que i. Cette fonction renverra le pointeur sur l'élément adressé lorsque le parcours est achevé.

### 1.2.10 Suppression de tous les éléments de la liste

Écrire la fonction *libereListe* qui permet de supprimer tous les éléments d'une liste. Le seul paramètre de cette fonction est le pointeur sur le premier élément de la liste.

### 1.2.11 Fonction main

Écrire la fonction *main* qui permet d'invoquer les fonctions précédentes afin de réaliser un gestionnaire de la liste chaînée. Initialement, la liste est vide et le pointeur sur le premier élément est initialisé à NULL.

## 2 Manipulation de tableaux dynamiques

### 2.1 Présentation du sujet

**But** : créer un ensemble de fonctions pour manipuler des tableaux de taille variable.

**Principe** : nous utiliserons les fonctions de manipulation de mémoire (malloc, calloc, realloc, memcpy, memmove, memset) pour créer une structure de données de taille quelconque.

**Structure des données** : Une structure contiendra le tableau de données (un 'char' ou un autre type de base) et sa taille. Un nouveau type de données sera créé pour cette structure.

## 2.2 Travail demandé

---

1. Créer la structure de données et le type associé.
2. Écrire la fonction **nouveau** qui alloue, initialise et renvoi un nouvel élément. Un paramètre sera utilisé pour la valeur d'initialisation.
3. Écrire la fonction **ajoute** qui ajoute un élément en fin de tableau.
4. Écrire la fonction **insere** qui insère un élément à un emplacement donné.
5. Écrire la fonction **copie** qui crée un nouveau tableau à partir d'une portion d'un autre tableau. Le bloc à copier sera délimité par la position de départ et sa longueur.
6. Écrire la fonction **supprime** qui efface et désalloue un élément indiqué par sa position.
7. Écrire la fonction **concatene** qui ajoute un tableau à la suite d'un autre.

Les consignes suivantes sont valables pour toutes les questions :

- La librairie 'string.h' ne devra pas être utilisée.
- Veiller à traiter convenablement les erreurs.
- Veiller à désallouer toutes les données créées.



# 22

# Les caractères, les chaînes, les pointeurs et l'allocation dynamique

## 1 Traitement de chaînes de caractères

### 1.1 Comparaison de chaînes

Écrire une fonction de comparaison de deux chaînes de caractères ne prenant pas en compte la casse. Cette fonction renverra :

- 0 si les deux chaînes sont identiques (même taille, même contenu),
- la différence entre les deux premiers caractères qui diffèrent sinon.

Son prototype est :

```
1 int compareInsensible(char *str1, char *str2);
```

### 1.2 Transcription d'algorithme

L'algorithme suivant prend un tableau en entrée et en affiche une version modifiée :

```
1 iBoucle1 <- 0
2 fini <- Faux
3 Tantque (iBoucle1 < Taille du Tableau) ET (fini = Faux) Faire
4 fini <- Vrai
5 iBoucle2 <- 1
6 Tantque (iBoucle2 < Taille du Tableau - iBoucle1) Faire
7 Si Tableau[iBoucle2] est inférieur à Tableau[iBoucle2 - 1] Alors
8 Inverser Tableau[iBoucle2] et Tableau[iBoucle2 - 1]
9 fini <- Faux
10 FinSi
11 incrementer iBoucle2
12 FinTantque
13 incrementer iBoucle1
14 FinTantque
15 Affiche le Tableau
```

La comparaison des deux chaînes de caractères se fera avec la fonction `compareInsensible`.

Implémentez cet algorithme sur un tableau de chaînes de caractères. Vous commenterez précisément son fonctionnement et décrirez son action.

Vous l'utiliserez ensuite dans un programme principal sur le Table 22.1 :

|          |           |          |          |
|----------|-----------|----------|----------|
| McIntosh | Jonagared | Braeburn | Jubile   |
| Gloster  | Jonagold  | Cybèle   | Calville |
| Elstar   | Juliet    | Tohoku   | Idared   |

Table 22.1 – Tableau de chaînes de caractères

## 2 Évaluation de la distribution d'une série de tirages de dés

### 2.1 Lancé de Dés

Écrire une fonction de génération d'un entier aléatoire compris entre 1 et un entier donné en paramètre d'entrée de la fonction.

```
1 int rand(void);
```

Renvoie une valeur pseudo-aléatoire comprise entre 0 et `RAND_MAX`. Appartient à la librairie `stdlib.h`.

**2.2 Affichage d'histogramme**

---

Écrire une fonction d'affichage d'un tableau d'entiers sous la forme d'un histogramme vertical avec la quantité stockée en abscisse et l'indice du tableau en ordonnée.

*Exemple avec les valeurs {1, 4, 9, 3, 8} :*

|   |              |
|---|--------------|
| 1 | 0 X          |
| 2 | 1 XXXX       |
| 3 | 2 XXXXXXXXXX |
| 4 | 3 XXX        |
| 5 | 4 XXXXXXXXXX |

**2.3 Normalisation de données**

---

Écrire une fonction de normalisation d'un tableau d'entiers.

La normalisation consiste à réduire les valeurs en conservant leurs rapports.

Cette fonction acceptera un entier en paramètre d'entrée : le paramètre de normalisation  $n$ , et un tableau  $\mathcal{X}$  d'entiers à normaliser. On remplace donc chaque valeur  $X \in \mathcal{X}$  dans le tableau par le rapport :

$$X_{\text{norm}} = X \times \frac{n}{\sum_{\forall Y \in \mathcal{X}} Y}$$

**2.4 Fonction « main »**

---

Écrire le programme principal qui demande à l'utilisateur :

1. Un nombre X de dés,
2. Le nombre Y de faces sur chaque dé,
3. Le nombre Z de tirage à réaliser.

et qui trace l'histogramme correspondant aux Z tirages de X dés à Y faces (nombre d'occurrences pour chaque somme des faces de l'ensemble des dés).

*Par exemple, on souhaite connaître un répartition du résultat de 10000 tirages de 2 dés à 6 faces.*

# 23

## Les fichiers

### 1 Gestion d'un carnet d'adresses de clients/fournisseurs

Dans une PME qui fabrique des rotisseries, vous avez des ingénieurs commerciaux. Ces commerciaux sillonnent quasiment toute l'année la France entière. Lors de leur déplacement, ils sont amené à rencontrer de nombreux fournisseurs et clients potentiels. Au fil des ans, ces commerciaux se sont constitués un carnet d'adresse conséquent. Chaque contact présent dans ce carnet d'adresse voit ses coordonnées inscrites sur une fiche cartonnée. De nombreux commerciaux ne supportent plus de se déplacer avec un classeur de fiches cartonnées. Il est grand temps d'infomatiser la gestion de ces fiches. On vous confie ce travail.

Chaque fiche contact contient les informations suivantes :

- Un indicateur spécifiant s'il s'agit d'une fiche client ou d'une fiche fournisseur,
- Le nom de l'entreprise,
- L'adresse de l'entreprise
- Le nom et le prénom du principal interlocuteur,
- La qualité du principal interlocuteur (directeur des achats, ...)
- Le mail du principal interlocuteur,
- Le téléphone du principal interlocuteur

#### 1.1 La structure de données

1. Spécifier la structure de données Contact.
2. Implémenter les fonctions CreerContact et DetruireContact permettant, respectivement, d'allouer un contact en mémoire et de désallouer un contact en mémoire.
3. Ecrire la fonction InitContact qui initialise les attributs de la structure à des valeurs vides.
4. Ecrire la fonction AfficherContact qui affiche le contact qui lui est fourni en paramètre.

#### 1.2 Gestion non triée du carnet d'adresse

On suppose maintenant que l'organisation du fichier de carnet d'adresse est la suivante :

- La première information présente dans le fichier est le nombre d'enregistrement Contact présent dans le fichier
- Les informations suivantes correspondent au contenu des enregistrements mis bout à bout. On supposera que la taille d'un enregistrement est fiche dans tout le fichier.

1. Créer la structure Carnet qui contient toutes les informations utiles du carnet.
2. Implémenter la fonction CreerCarnet qui crée un fichier de carnet d'adresse et retourne un pointeur sur une structure Carnet. Ce pointeur sera NULL en cas d'échec.
3. Implémenter la fonction OuvrirCarnet qui ouvre un fichier de carnet d'adresse et retourne un pointeur sur une structure Carnet. Ce pointeur sera NULL en cas d'échec.
4. Implémenter la fonction FermerCarnet qui ferme un fichier de carnet et désalloue la structure Carnet fournie en paramètre.
5. Implémenter la fonction LireContact qui permet de récupérer dans une structure Contact l'enregistrement dont le numéro est fourni en paramètre de la fonction. La structure Contact sera transmise via un passage par adresse.
6. Ecrire la fonction AfficherCarnet qui affiche le contenu du carnet d'adresse. La structure Contact sera transmise via un passage par adresse.
7. Implémenter la fonction EcrireContact similaire à LireContact mais qui écrit un Contact dans l'enregistrement dont le numéro est spécifié en paramètre de la fonction. La structure Contact sera transmise via un passage par adresse.

## 23 Les fichiers

---

8. Implémenter la fonction AjouterContact qui ajoute un contact au carnet. La structure Contact sera transmise via un passage par adresse. La valeur retournée par la fonction est le numéro de l'enregistrement dans le fichier de carnet.
9. Implémenter la fonction SupprimerContact qui supprime le contact spécifié en paramètre du carnet. Les enregistrements valides sont tassés au début du fichier.
10. Implémenter la fonction ChercherEntreprise qui retourne le numéro du contact qui correspond exactement au nom de l'entreprise spécifiée en paramètre. Si cette entreprise n'existe pas, la valeur -1 est retournée.
11. Ecrire une fonction main qui crée un carnet, y ajoute 4 Contacts, effectue une recherche d'un contact par le nom de l'entreprise et supprime le contact trouvée.

### 1.3 Gestion triée du carnet d'adresse

---

Supposons maintenant que les enregistrements du fichier de carnet sont triés dans l'ordre alphanumérique de l'attribut nom de l'entreprise. On vous fournit la fonction suivante.

```
1 int Dichotomie(struct Carnet * carnet , const char * cle , int * position)
2 {
3 int L = 0;
4 int R = carnet->NbEnr-1;
5 int M, res;
6 struct Contact contact;
7
8 while (L <= R)
9 {
10 M = (L+R)/2;
11 LireContact(carnet , M, &contact);
12 res = strcmp(contact.NomEntreprise, cle);
13 if (res == -1)
14 R = M-1;
15 else
16 if (res == 1)
17 L = M+1;
18 else
19 {
20 *position = M;
21 return 1;
22 }
23 }
24 *position = L;
25 return 0;
26 }
```

1. Que fait la fonction Dichotomie lorsque la clé est présente dans le carnet d'adresse ?
2. Que fait la fonction Dichotomie lorsque la clé n'est pas présente dans le carnet d'adresse ?
3. Implémenter la fonction AjouterContactAvecTri qui ajoute un contact au carnet trié. La structure Contact sera transmise via un passage par adresse. La valeur retournée par la fonction est le numéro de l'enregistrement dans le fichier de carnet. Le fichier Carnet sera encore trié à la fin de l'opération.
4. Implémenter la fonction ChercherEntrepriseAvecTri qui retourne le numéro du contact qui correspond exactement au nom de l'entreprise spécifiée en paramètre. Si cette entreprise n'existe pas, la valeur -1 est retournée.
5. Les fonctions XXXXAvecTri sont-elles plus avantageuses ? Pourquoi ?

## 2 Crédit d'un fichier de log d'activités

---

Lors de l'exécution d'un programme C, on souhaite garder une trace de l'activité du programme dans un fichier de logs.

1. Ecrire la fonction InitLog qui admet en paramètre le nom du fichier de logs ainsi que le type d'accès au fichier (création du fichier ou ajout à la fin du fichier).
2. Ecrire la fonction LogIt qui admet un message et l'ajoute à la fin du fichier de logs.
3. Ecrire la fonction CloseLog qui ferme le fichier de logs.
4. Ecrire une fonction main utilisant ces fonctions.

**3 Fusion du contenu de deux fichiers**

---

On considère deux fichiers textes dont les lignes sont triés dans l'ordre alphanumérique.

1. Ecrire la fonction FusionNonTrie qui fusionne le contenu de deux fichiers et met le résultat dans un troisième fichier.
2. Ecrire la fonction FusionTrie qui fusionne le contenu de deux fichiers et met le résultat dans un troisième fichier. Le troisième fichier doit contenir les lignes de textes des deux fichiers mais triées dans l'ordre alphanumérique.



# 24

# Pointeurs de fonctions, programmation générique et callback, GLIB/GTK+

## 0.1 Une calculatrice en notation polonaise inverse

### 0.1.1 La notation polonaise inverse

Cette notation, aussi appelée « post-fixée », permet de noter les formules arithmétiques sans utiliser de parenthèses : les opérandes (les nombres) sont précisés avant les opérateurs. Voici quelques exemples :

| Expression          | Notations polonaise inverse        |
|---------------------|------------------------------------|
| $3 * (4 + 7)$       | $4 7 + 3 *$ ou $3 4 7 + *$         |
| $((1 + 2) * 4) + 3$ | $1 2 + 4 * 3 +$ ou $3 4 1 2 + * +$ |

L'expression est évaluée à l'aide d'une pile. Les opérandes sont empilées dans l'ordre où elles sont saisies puis elles sont dépliées lorsqu'un opérateur est rencontré. Le résultat du calcul est ajouté à la pile. Avec le calcul  $(3 4 7 + *)$  on obtient, à chaque étape, la pile suivante :

| Etape | Opérande / opérateur | Actions                    | Pile  |
|-------|----------------------|----------------------------|-------|
| 1     | 3                    | Empiler                    | 3     |
| 2     | 4                    | Empiler                    | 3 4   |
| 3     | 7                    | Empiler                    | 3 4 7 |
| 4     | +                    | Dépiler, Calculer, Empiler | 3 11  |
| 5     | *                    | Dépiler, Calculer, Empiler | 33    |

### 0.1.2 Structure des données utilisées

Les fonctions mathématiques utilisables dans la calculatrice seront de type binaire : elles prendront deux nombres réels en entrée et renverront un nombre réel.

Une même structure de données stockera une opérande ou un opérateur. Une opérande sera codée comme un réel et un opérateur sera un pointeur de fonction. Pour stocker ces informations, nous utiliserons une *union* de variables. Comme il n'est pas possible de connaître le type de donnée contenu dans une *union* à un instant donné, nous englobons l'*union* dans une *structure*. Un champ supplémentaire de la structure, indiquera quel est le type de donnée. Le type « opérande » ou « opérateur » sera codé sur un entier de taille minimale. Nous définirons des constantes pour manipuler cette information.

La taille de la pile de calcul sera limitée via une constante.

### 0.1.3 Travail demandé

1. Définir les structures de données nécessaires pour stocker un élément de calcul, ajouter les types correspondants.
2. Définir un type de données pour le pointeur de fonctions « opérateur ».
3. Écrire une fonction qui crée, initialise et renvoi une nouvelle opérande.
4. Écrire une fonction qui crée, initialise et renvoi un nouvel opérateur.
5. Écrire une fonction permettant de lire la valeur d'une opérande.
6. Écrire une fonction permettant de récupérer le pointeur de fonction d'un opérateur.
7. Écrire une fonction permettant d'écrire la valeur d'une opérande.
8. Écrire une fonction permettant d'écrire le pointeur de fonction dans un opérateur.
9. Écrire une fonction qui appelle la fonction précisée en paramètres avec les valeurs indiquées.

10. Écrire les fonctions mathématique réalisant les opérations souhaitées (addition, soustraction, multiplication, division...)
11. Écrire la fonction qui réalise le calcul à partir d'une liste l'opérandes / opérateurs pré-existante. Penser à vérifier que la pile ne déborde pas et que suffisamment d'opérandes sont disponibles pour exécuter une opération.

### 0.1.4 Exercices complémentaires

---

1. Créer une fonction permettant de saisir les opérandes et les opérateurs au clavier. Modifier la calculatrice en conséquence pour utiliser la fonction de saisie.
2. Créer une *structure* de pile et les fonctions 'push' et 'pop' associées. Utiliser cette nouvelle pile pour remplacer le tableau de stockage temporaire des opérandes.
3. Nous souhaitons utiliser des fonctions mathématiques ne comportant qu'un seul paramètre (négation, sinus, exponentielle...). Quelles sont les modifications à apporter aux structures de données ? Transformer la calculatrice pour accepter de telles fonctions.

## 1 Premiers pas avec GTK+

---

La tradition veut que le premier programme que l'on réalise avec un langage soit un « Hello World ». Ne manquons donc pas à ce principe. Ce premier programme GTK+ est une fenêtre graphique correctement initialisée dans laquelle s'affiche le texte « Bonjour le monde ».

Rappel : *Il faut indiquer au compilateur l'utilisation des bibliothèques GTK+ avec l'option : 'pkg-config --cflags --libs gtk+-2.0'*.

## 2 Une petite application réelle : le chronomètre

---

On veut maintenant écrire un chronomètre graphique. Un chronomètre classique comporte un affichage du temps décompté ainsi que trois boutons :

1. Un bouton servant à démarrer , continuer ou recommencer le décompte
2. Un bouton servant à stopper le décompte
3. Un bouton pour remettre le compteur à zéro

Rappels :

- La GLib contient un type GTimer qui implémente des fonctions de chronomètre
- Pensez à l'utilisation de structure pour passer plusieurs paramètres avec un seul pointeur

# 25

# Compilation avancée, utilisation de bibliothèques et fichiers .h complexes

## 1 Utilisation des directives du préprocesseur

Ecrire un petit programme permettant d'afficher les différentes informations relatives à sa compilation :

1. Nom du fichier source
2. Sa version
3. La date et l'heure de sa compilation
4. Le système d'exploitation sur lequel il a été compilé

## 2 Programmation modulaire

### 2.1 Module de gestion d'arbre binaire

L'arrangement des données selon une structure d'arbre est couramment utilisé en informatique. Le module à développer doit servir de boîte à outil pour gérer une catégorie particulière d'arbre, les arbres binaires (chaque noeud possède au plus 2 noeuds fils). Il s'agit donc de définir une structure de données pour un arbre binaire d'entiers. On l'assortira des fonctions suivantes :

- Création de l'arbre
- Ajout d'un fils gauche à un noeud donné (et son équivalent pour le fils droit)
- Récupération du fils gauche d'un noeud (et son équivalent pour le fils droit)
- Récupération de la valeur stockée dans un noeud

### 2.2 Module de parcours d'arbre

Ce second module implémente des fonctions de parcours d'arbre générique. Il propose deux fonctions d'affichage du contenu d'un arbre binaire :

- Suivant un parcours en profondeur
- Suivant un parcours en largeur.

Notes :

- *Vous allez sans doute avoir besoin d'une gestion de liste dans ce module. Vous pouvez réutiliser celle que vous avez développé précédemment dans ces TD, ou utiliser celle fournie par la glib*
- *Il faut indiquer au compilateur l'utilisation de la bibliothèque GLib avec l'option : 'pkg-config --cflags --libs glib-2.0'*

### 2.3 Module de test et Makefile

Pour tester ces modules, on leur ajoute un programme principal simple visant à créer un arbre binaire, puis à l'afficher en profondeur, puis en largeur. L'utilisation d'un makefile permet une compilation plus simple d'un programme à plusieurs modules. Vous mettrez donc en place un makefile permettant la compilation, le nettoyage et l'exécution du projet.

### 2.4 Pour aller plus loin ...

L'idée de la programmation modulaire est de pouvoir découper le code en parties indépendantes. Vous pouvez donc :

- Imaginez un autre module implémentant une gestion différente des arbres binaires (par des listes par exemple), puis utilisez-le à la place de l'autre.
- Compilez vos modules et les fournir à votre voisin pour qu'il les utilise comme bibliothèque.