Langage C

E. Lecolinet ENST

Aperçu

Un programme C comprend :

— un ou plusieurs **fichiers** C

— définissant des **fonctions**

— au moins une fonction nommée **main**()
(= la 1ère fonction appelée)

Un fichier C comprend :

— des inclusions de "**headers**"

— des **variables** globales (éventuellement)

— des définitions de **fonctions**

Exemple

un seul fichier "hello.c" :

```
#include <stdio.h>

int main()
{
   printf("hello word\n");
}
```

NB: stdio.h = header standard pour les E/S

fichier1.c	fichier2.c	fichier2.c
fonction1() fonction2() **main**()	fonction3() fonction4()	fonction5() fonction6() ....

Eric LECOLINET
ENST - Paris

Une fonction C comprend :

— un **en-tête** :
(valeur retournée, nom, arguments)

— un **bloc** :
délimité par les symboles **{** et **}**

Un bloc comprend :

— des **déclarations** (de variables ...)

— des **instructions**

— d'autres **blocs**

Eric LECOLINET
ENST - Paris

Exemple: programme "hello_you":

Fichier "hello_you.c" avec 2 fonctions :

```
#include <stdio.h>

void foo(const char *name){
   printf("hello %s \n", name);
}

int main(int argc, char *argv[])
{
  const char* name;

  name = "hubert";
  foo(name);
  dupondt();
}
```

Fichier "dupondt.c" avec 1 fonction :

```
void dupondt() {
   foo("dupondt");
}
```


Eric LECOLINET
ENST - Paris

Syntaxe

— commentaires entre: */* et */*

— nom de header entre: **<** *et* **>** ou **"** *et* **"**

— CHAÎNES de caractères entre: **"** *et* **"**

— UN caractère entre: **'** *et* **'**

— **{** et **}** pour délimiter les blocs

Toutes les variables doivent être déclarées avant utilisation

Fonction main = 1ere fonction appelée quand on exécute le programme

Eric LECOLINET
ENST - Paris

Types, Opérateurs et Expressions

Types de base

— **char** : caractère (1 octet)
— **int** : entier
— **float** : flottant simple précision
— **double** : flottant double précision

Les tailles varient suivant le compilateur.

Qualifieurs

— **short** : entier court
— **long** : entier long
— **long** double
— **signed** int
— **unsigned** float

Eric LECOLINET
ENST - Paris

Règles:

— short <= int <= long
— short >= 16 bits
— long >= 32 bits
— cf. <limits.h> et <float.h>
(headers standards dans: /usr/include)

Sur Sun 32 bits (avec cc et gcc) :

— short = 16 bits,
— int = long = 32 bits
— float = 32 bits
— double = 64 bits

Exemple:

— **unsigned** char : de 0 à 255
— **signed** char : de -128 à 127
— **char** : dépend du compilateur (de -128 à 127 sur Sun)

Eric LECOLINET
ENST - Paris

Constantes

— 1234 --> int
— **0**37 --> int (en octal)
— **0x**1f --> int (en hexadécimal)

— 1.234 --> double
— 1e-2 --> double

— 'a' --> caractère
— "abcd" --> string (chaîne de caractères)

Constante caractère = valeur entière : (en ASCII) :

— 'A' = le caractère A; valeur = 65
— '0' = le caractère 0 (zéro); valeur = 48

Constante chaîne de caractères =

— suite de caractères suivie d'un 0 (zéro)

Caractères spéciaux

— '\0' = le caractère NUL; valeur = 0
— '\n' = new line, '\t' = tabulation, etc ...
— '\ooo' = caractère défini en octal
— exemple : '\007' = caractère BELL

Definition de macros

```
#define MAX_VAL   255
#define BELL      '\007'
#define MESSAGE   "hello word"
```

— Substitution textuelle !!!
— Concaténation implicite:
"hello word" == "hello " "word"

Enumérations

```
enum boolean {NO, YES};
enum semaine {LUN=1, MAR, MER, ...};
```

— commence implicitement à 0
— on peut préciser les valeurs par =

Déclarations de variables

— doivent être déclarées **avant** utilisation

— déclaration terminée par un ;

— valeur indéfinie par défaut !!!

```
char c, rep;
int  i, res, truc;
int  vect[1000], mat[5][10];
```

- vecteur de 1000 entiers
- matrice de 5 lignes et 10 colonnes

Déclaration avec initialisation

```
int i = 5;
char new_line = '\n';
```

— **toujours** initialiser les variables !!!

Variables à valeur constante

```
const float pi = 3.14;
```

— variable qui ne peut pas changer de valeur

— à préférer aux macros

Affectation des variables

— variables préalablement déclarées

— affectation --> signe **=**

— toute instruction se termine par un ;

!Attention: ne pas confondre = avec ==

```
{
  int i, j, k, l;

  i = 5;
  j = 3;
  k = i + j;
  l = i + j * k;
  m = (i + j) * k;
  ....
}
```

Attention à la précédence des opérateurs !

Opérateurs arithmétiques

Par ordre de priorité:

- - (unaire) -x
- * / %
- + - (binaire) x - y

% = reste de la division entière

/ = division entière **ou** réelle suivant le cas

Attention:

2 / 4 vaut 0

mais 2. / 4 vaut 0.5

Langage C - Eric Lecolinet - ENST Paris

Opérateurs logiques et relationnels

Par ordre de priorité :

— ! (négation unaire)

— > >= < <=

— == !=

— && : signifie "et logique"

— || : signifie "ou logique"

Propriétes

— évalués de la gauche vers la droite

— priorité < aux opérateurs arithmétiques :
(sauf pour négation unaire **!**)

```
a < lim-1    ==    a < (lim-1)
```

— arrêt **dès que** la valeur de vérité est trouvée:

```
(k < tab_len-1 && tab[k] != 0)
(i <= 5 &&(c = getchar()) != 'n'))
```

NOTE: c doit entre un **int** (pas un char!)

Langage C - Eric Lecolinet - ENST Paris

Conversion implicite des types

— Principe général : conversion implicite vers le type "le plus grand"

— Cette règle est parfois dangereuse !!!

```
{
  int i = 2, j = 4;
  char c;
  float x = 4., y1, y2, y3;

  i = c;
  c = i;    /* entier tronqué */

  x = i;
  i = x;    /* réel tronqué */

  y1 = i + x;
  y2 = i / x;
  y3 = i / j;
  .....
}
```

!Attention: y3 vaut 0 (car i et j sont des int)

Langage C - Eric Lecolinet - ENST Paris

Conversion explicite des types

au moyen d'un "**cast**" :

```
int i = 2, j = 4;
float y, z;

y = i / j;              /* y = 0.0 */
z = (float) i / j;      /* y = 0.5 */
```

Incrémentation

— i++ ou ++i signifient: i = i + 1

— i += n signifie: i = i + n

— même chose avec: i--, --i, i -= n

Notation préfixe et postfixe:

— i = 5; a = ++i; --> i = 6 et ***a = 6***

— i = 5; a = i++; --> i = 6 et ***a = 5***

Attention aux ambiguités:

— un seul ++/-- par ligne!

Langage C - Eric Lecolinet - ENST Paris

Manipulation de bits

— & ET (à ne pas confondre avec &&)
— | OU inclusif (à ne pas confondre avec ||)
— ^ OU exclusif
— << décalage à gauche
— >> décalage à droite
— ~ complément à un

```
int n, m;
m = n & 0x10;
m = n << 2;     /* <->  m = n * 2 */
```

!Attention:

Ne pas confondre :
— & avec && (et logique)
— | avec || (ou logique)

Parenthèses nécessaires dans expressions composées (cf. priorité des opérateurs)

Eric LECOLINET
ENST - Paris

Priorité des opérateurs

Eric LECOLINET
ENST - Paris

Tests et Boucles

Expressions conditionnelles

Principe:

```
if (test)
  action1;
else
  action2 ;
```

Remarques:

— pas de "then"

— mais des **( )** autour du "test"

— **;** à la fin des DEUX instructions

Exemple:

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


Eric LECOLINET
ENST - Paris

Tests imbriqués

```
if (a > b && ( a < 5 || b < 10))
  z = a;
else z = b;
```

Rappel: arret de l'évaluation dès que :

— condition fausse pour: &&

— condition vraie pour: ||

Booléens

pas de type booléen en C => simulation :

```
#define False    0
#define True     1
#define Boolean  int

Boolean ok, fini, status;

if (ok)              /*(ok != 0)*/
   status = True;

if (! fini)          /*(fini == 0)*/
   status = False;
```


Eric LECOLINET
ENST - Paris

Actions imbriquées

```
if (n > 0)
  if (a > b)
     z = a;
  else z = b;


if (n > 0) {
  if (a > b)
     z = a;
}
else z = b;
```

Le "**else**" ne se rapporte pas au même **if** !!!

Règles:

— **toujours** mettre des **{ }** pour éviter les ambiguités

— indenter intelligemment le code

Cas particulier: else-if cascadés

```
if (test1)
  action1;
else if (test2)
  action2 ;
else if (test3)
  action3 ;
....
else action_par-défaut ;
```

Structure de tests en "rateau"

Ne pas mettre des **{ }** inutiles

Aternative: le **switch**

Switch

```
switch (expression){
  case const1:
    actions1;
    break;

  case const2:
  case const3:
    actions2;
    break;

  default:
    actions3;
    break;
}
```

— const1, const2 .... sont des **constantes** à valeur entière

— **break** ou **return** pour sortir du switch (sinon on continue jusqu'à la fin du switch)

— plusieurs actions possibles séparées par **;**

Langage C - Eric Lecolinet - ENST Paris

Opérateur ?

L'expression :

```
test ? expr1 : expr2
```

renvoie :

— expr1 si le test est vrai (cad. != 0)

— expr2 sinon

Exemple

```
z = (a > b) ? a : b ;
```

équivaut à :

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

Application

```
#define MAX(a,b) ((a)>(b) ? (a):(b))

int i = 5, j = 7, k;
........
k = MAX(i,j);
```

Langage C - Eric Lecolinet - ENST Paris

Boucles while et for

```
while (test) {
   actions;
}

for (init; test; incr) {
   actions;
}
```

— *init, test* et *incr* = expressions quelconques
— test effectué **avant** d'entrer dans la boucle
— continue tant que *test != 0*

Préférer la boucle **for**, équivalente à :

```
    init ;
    while (test) {
       actions;
       incr;
    }
```


Eric LECOLINET
ENST - Paris

Exemples

```
/* TOUJOURS définir les tailles
 * par des MACROS
 */
#define TAB_SIZE 20

int tab[TAB_SIZE];
int i;

/* en C les tableaux commencent a
 * l'indice 0 => balayer "tab"
 * de 0 a TAB_SIZE-1
 */

for (i = 0; i < TAB_SIZE; i++) {
   tab[i] = 0;
}

/* equivaut a : */

i = 0;
while (i < TAB_SIZE) tab[i++] = 0;


i = -1;
while (++i < TAB_SIZE) tab[i] = 0;
```


Eric LECOLINET
ENST - Paris

Boucle do-while

```
do {
    actions;
} while (test) ;
```

— action toujours exécutée au moins 1 fois
— continue tant que *test != 0*

Break et Continue

— **break** permet de sortir d'une boucle ou d'un switch
— **continue** passe à l'itération suivante

Goto et Labels

— **à éviter** sauf cas particuliers

Exemples

```
#define TAB_SIZE 20

char line[TAB_SIZE];
int i;

/* trouver le 1er char non blanc */

for(i = 0; i < TAB_SIZE; i++){
  if (line[i] != ' ') break;
}
```

```
/* une justification de goto */

for (....)
  for (....)
    for (....) {
       ....
       if (Touché(iceberg) == True)
          goto FATAL;
    }
   ....

FATAL :
  printf("sauve qui peut ...\n");
```

Fonctions

Objectif:

Découper un programme en petites entités :

— indépendantes

— réutilisables

— plus lisibles

Librairies et compilation séparée

Les fonctions peuvent être définies :

— dans plusieurs fichiers

— dans des librairies

Syntaxe

```
type nom_fonction(parametres) {
   declarations;
   instructions;
   return type;
}
```


Exemple: *fichier main.c :*

```
#include <stdio.h>

int ChercheVal( float tab[ ], int taille, float val ) {
   int k;
   for (k = 0; k < taille; k++) {
      if (tab[k] == val) return k;
   }
   return -1;
}

int main( int argc, char *argv[] ) {
   float donnees[ ] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10.};
   int indice = -1;

   indice = ChercheVal(donnees, 10, 4.);

  /* que manque t'il ici ? */
   printf( "indice = %d , valeur = %f \n",
            indice,  donnees[indice] );
}
```

— valeur de l'indice ?

— noter l'initialisation des tableaux

— il manque un test : pourquoi ?

Argc,argv

```
int main( int argc, char *argv[] )
{
  if (argc > 1) printf("%s %s \n", argv[0], argv[1]);
  else return 1;
  ....
}
```

— argv : arguments de la ligne de commande

— argc : nombre d'arguments

— return : retourne un code d'erreur (0 = OK)

Déclarations et instructions

```
int main( int argc, char *argv[] )
{
  if (argc > 1) printf("premier argument %s\n, argv[1]);
  int i, j;     // FAUX!
  ....
}
```

— déclarations de variables AVANT les instructions (contrairement à Java et C++)

— mais on peut avoir des sous-blocs **{ }**

— // pas accepté par tous les compilateurs

Eric LECOLINET
ENST - Paris

Return

```
int ChercheVal( float tab[ ], int taille, float val ) {
   ...
  return -1;
}
```

— ne pas oublier le **return** !

— **void** si la fonction ne retourne rien :

```
void foo(int i, int j) {
   if (i > j) return;
   .....
}
```

Fonction sans paramètre

```
int foo(void) {
   .....
}
```

Syntaxe obsolète (aucune vérification !)

```
void foo(i, j)
int i, j;
{
   .....
}
```


Eric LECOLINET
ENST - Paris

Passage des arguments

Toujours par **valeur :**

— les arguments transmis sont **recopiés**

— dans les **paramètres** des fonctions

— sauf pour les **tableaux**

Exemple: quelles sont les valeurs affichées ?

```
void swap( int i, int j ) {
   int  aux;
   aux = i;   i = j;   j = aux;
}

int main() {
   int  i = 5,  j = 7;
   swap( i, j );
   printf( " i = %d , j = %d \n", i , j );
}
```


Passage des tableaux en argument

```
int CherchеVal( float tab[ ], int taille, float val ) {
   ...
}

int main() {
   float donnees[ ] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10.};
   int indice = ChercheVal(donnees, 10, 4.);
   ....
}
```

Le contenu des **tableaux** n'est pas recopié

— car c'est leur adresse qui est recopiée !

Autres cas où les arguments ne sont pas recopiés

Via une indirection par un **pointeur**

— à suivre ...

Nombre d'éléments d'un tableau

```
#define CARD(T)      ( sizeof(T) / sizeof(T[0]) )

int main() {
   float donnees[ ] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10.};
   int indice = ChercheVal(donnees,
                           CARD(donnees), 4.);
   ....
}
```

— CARD(donnees) vaut 10 dans main()

mais attention:

```
int ChercheVal( float tab[ ], int taille, float val ) {
   ...
}
```

— CARD(tab) != 10 dans ChercheVal() !!!

— car le paramètre **tab** n'est pas un tableau mais un **pointeur** !

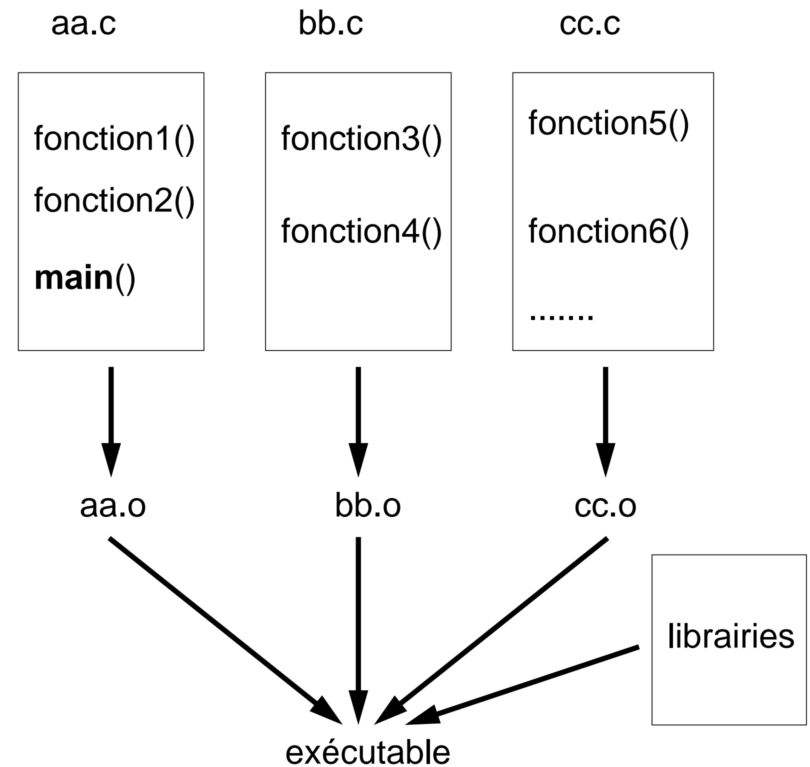
— ce qui revient à :

```
int ChercheVal( float *tab, int taille, float val ) {
   ...
}
```


Eric LECOLINET
ENST - Paris

Compilation séparée

Fonctions réparties dans plusieurs fichiers.



Eric LECOLINET
ENST - Paris

Principales options du compilateur C

— `-g` pour deboguer

— `-O/-O1/-O2`... pour optimiser

— `-Wall` pour afficher plus d'infos (avec gcc) **(TOUJOURS METTRE CETTE OPTION!)**

— `-I`*`directory`* : chercher les headers dans ce répertoire (dans l'ordre s'il y a plusieurs `-I`)

Principales options de l'éditeur de liens

— `-L`*`directory`* : chercher les librairies dans ce répertoire (dans l'ordre s'il y a plusieurs `-L`)

— `-l`*`library`* : chercher les fonctions dans cette librairie dans un des répertoires précédemment spécifiés par `-L`

— attention l'ordre importe !

Répertoires standard sous Unix

– `/usr/include et /usr/local/includ`

— `/usr/lib et /usr/local/lib` (suivant système et compilateur)

Eric LECOLINET
ENST - Paris

Makefile

Fichier de règles indiquant:

— les fichiers à compiler

— les compilateurs et leurs options

— les librairies, etc.

La commande **make**

— lit le **Makefile**

— appelle automatiquement les outils adéquats

— vérifie les dates et ne recompile **QUE** ce qui doit l'être

On a typiquement 1 Makefile par répertoire

Règle d'or:

— toujours utiliser un Makefile (ou, au choix, un outil plus évolué)

— ne pas s'amuser à retaper les commandes de compilations à la main (toujours fausses dès que l'application devient importante !)

Eric LECOLINET
ENST - Paris

Exemple de Makefile simplifié

```
# spécifique à Sun
.KEEP_STATE:

# le compilateur C et ses options
CC= gcc
CFLAGS= -g -Wall -I/usr/local/qt/include

# les librairies utilisees
LDLIBS = -L/usr/local/qt/lib -lqt

# les fichiers objet de l'application et le nom de l'executable
OBJS= tri.o donnees.o
EXEC= tri

# regle de production de l'application
# ATTENTION: la 2e ligne commence par une tabulation
$(EXEC) : $(OBJS)
	$(CC) $(CFLAGS) -o $@ $(OBJS) $(LDLIBS)

# nettoyage
clean:
	-@$(RM) $(EXEC) $(OBJS)
```

make : lance la 1ere règle (qui compile tout)
make clean : lance la règle "clean"

Compléments sur les Makefiles

Le Makefile précédent est incomplet :

— pas de règles pour créer les .o
— => fait implicitement
— mais sans tenir compte des .h !!!

Il faut soit:

— mettre les règles à la main
— utiliser makedepend (ou .KEEP_STATE: sous Sun)
— utiliser configure ou un outil plus évolué

Autres outils utiles

— xemacs, etc.
— gdb, xxgdb, ddd
— grep, nm
— ar
— tar, gtar, gzip, gunzip
— workshop

Compléments sur les Librairies

Librairies statiques :

— extension .a

— simples "archives" de fichiers .o

— le code est inséré dans l'exécutable à la compilation

Librairies dynamiques :

— extension .so, dylib, etc.

— le code n'est PAS inséré dans l'exécutable

— il est chargé DYNAMIQUEMENT à l'exécution

Avantages/inconvénients

— les programmes sont beaucoup moins gros

— moins de swapping car le binaire est partagé

— mais la librairie doit exister à l'exécution !

  — => éventuels problèmes de licences

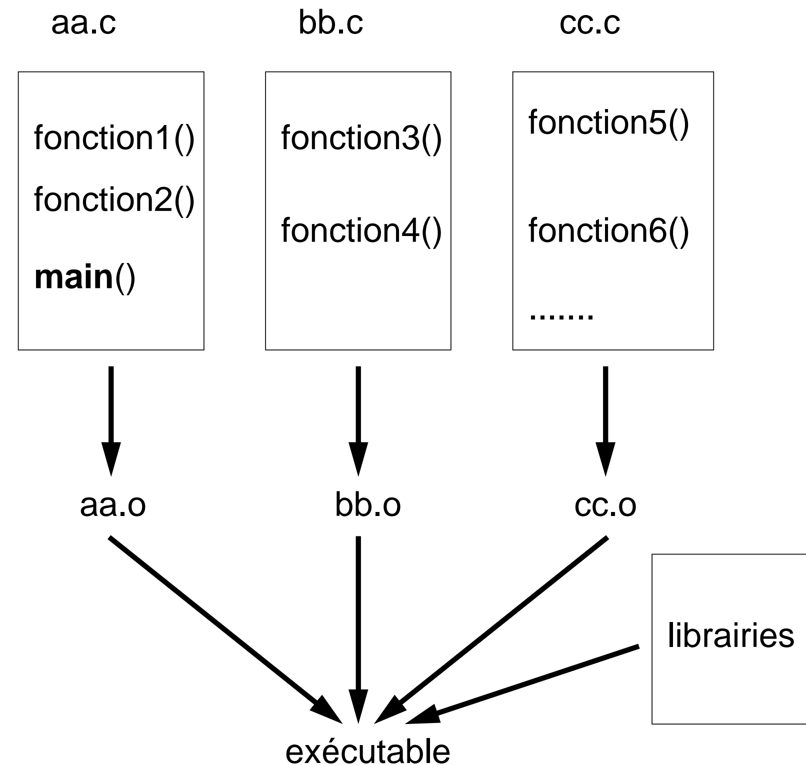
— et il faut trouver la bonne !

Les libs dyn sont recherchées suivant la variable:

— **LD_LIBRARY_PATH** (ou équivalent)

Retour à la compilation séparée

Fonctions réparties dans plusieurs fichiers.



Appel d'une fonction définie dans un autre fichier

Le langage C :

— permet d'appeler une fonction **indéclarée**

— mais ne vérifie pas ses arguments !

— car le compilateur n'examine qu'un seul fichier à la fois !

=> source d'erreurs considérable !!!

Pour imposer une vérification

— la fonction doit être **déclarée**

— dans le fichier où elle est **appelée** (avant l'appel)

Déclaration de fonction

```
extern int ChercheVal(float tab[ ], int taille, float val) ;
```

Différence avec la **définition:**

— pas de corps de fonction mais un **;**

— le mot clé (optionnel) **extern**

Première version (pas satisfaisante)

fichier cherche.c :

```
/* définition de la fonction */
int ChercheVal( float tab[ ], int taille, float val ) {
    int k;
    for (k = 0; k < taille; k++)
        if (tab[k] == val) return k;
    return -1;
}
```

---

fichier main.c :

```
#include <stdio.h>

/* déclaration de la fonction (insuffisant !) */
int ChercheVal( float tab[ ], int taille, float val ) ;

int main() {
    float donnees[ ] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10.};

    int indice = ChercheVal (donnees, 10, 4.);

    if (indice >= 0)
        printf( "indice = %d , valeur = %f \n",
                  indice,  donnees[indice] );
}
```

Problème

— cohérence de la déclaration et de la définition

Solution

— inclure la **même déclaration** dans les fichiers
- — où la fonction est définie
- — et ceux où elle est appelée

— via des fichiers partagés appelés "**headers**"

Principe

- `cherche.c` --> **définition** de *ChercheVal*
- `cherche.h` --> **déclaration** de *ChercheVal*

inclure `cherche.h`
- — dans main.c
- — et dans cherche.h !

Eric LECOLINET
ENST - Paris

header cherche.h :

```
extern int ChercheVal( float tab[ ], int taille, float val );
```

---

header utile.h :

```
#define CARD(tab)       ( sizeof(tab) / sizeof(tab[0]) )
```

---

fichier cherche.c :

```
#include "cherche.h"

int ChercheVal( float tab[ ], int taille, float val ) {
    .......
}
```

---

fichier main.c :

```
#include <stdio.h>
#include "utile.h"
#include "cherche.h"

int main() {
    .......
    indice = ChercheVal (donnees, CARD(donnees), 4.);
    .......
}
```


Eric LECOLINET
ENST - Paris

Headers standards

Déclaration des fonctions des bibliothèques (printf( ), cos( ), strcpy( ) ...)

— stdio.h : fonctions d'entrées-sorties

— math.h : fonctions mathématiques

— string.h : chaînes de caractères

— etc.

Il y a un header pour chaque librairie

Recherche des headers

— header **utilisateur** #include "`truc.h`"

— header **standard** #include <`truc.h`>

< > recherche dans

— répertoires standard (/usr/include ...)

— répertoires indiqués en option -I de **cc**

```
cc -I.. -I../.. truc.c -o truc
```


! Attention

— fonction externe non déclarée

— qui retourne autre chose qu'un **int**

=> PROGRAMME FAUX !!!

Ne PAS oublier d'inclure les headers

y compris pour les fonctions des librairies

Exemple: faux sans le `#include`

```
#include <math.h>

int main(){
  double x = cos(0.5);
  printf("cos(0.5) = %f \n", x);
}
```

Remarque importante

— l'édition de liens est indépendante de la compilation

— => un programme peut "compiler" (en trouvant les bonnes librairies) même s'il n'utilise pas les bons headers !

Variables

Variable automatique

— variable **locale** à une fonction

— accessible uniquement dans cette fonction

— valeur initiale indéterminée
(sauf si initialisation explicite)

Variable globale

— déclarée hors fonction

— accessible dans toutes les fonctions

Variable externe

— déclare une variable **globale**

— **définie** dans un **autre** fichier
(même principe que pour les fonctions)

Eric LECOLINET
ENST - Paris

header main.h :

```
#define TAILLE 10
extern float donnees[TAILLE];        /* déclaration */
```

---

fichier main.c :

```
#include <stdio.h>
#include "main.h"
#include "cherche.h"

float donnees[TAILLE] = {              /* définition */
    1., 2., 3., 4., 5., 6., 7., 8., 9., 10.
};

void Affiche() {
    int k;
    for (k = 0; k < TAILLE; k++)
        printf (" tab[%d] = %f \n",  k,  donnees[k]);
}

int main() {
    int indice = -1;
    Affiche();
    indice = ChercheVal( 4.);
    if (indice >= 0)
        printf( "indice = %d , valeur = %f \n",
                  indice,  donnees[indice] );
}
```


Eric LECOLINET
ENST - Paris

header cherche.h :

```
extern int ChercheVal(float val);
```

---

fichier cherche.c :

```
#include "main.h"
#include "cherche.h"

/* donnees = var. externe déclarée dans main.h */

int ChercheVal(float val) {
   int k;
   for (k = 0; k < TAILLE; k++)
      if (donnees[k] == val) return k;
   return -1;
}
```

La variable "`donnees`" est :

— **définie** dans `main.c`

— **déclarée** (via **extern**) dans `main.h`
et donc dans tous les fichiers incluant `main.h`

Règles

— **une seule** définition

— déclaration **cohérente** avec définition

— sinon le programme est faux !

Comparaison avec les fonctions

— même principe que pour les fonctions

sauf que:

— une variable doit forcément être définie ou déclarée

— le mot clé **extern** est obligatoire pour déclarer une variable externe (il est optionnel pour les fonctions)

Conseils

— éviter les variables globales

— toujours les déclarer dans un header

Variables statiques

— variables **permanentes**

— connues seulement du **fichier** ou de la **fonction** où elles sont définies

fichier main.c :

```
#include <stdio.h>
#include "main.h"
#include "cherche.h"

static float donnees[TAILLE] = {        /* var. static */
   1., 2., 3., 4., 5., 6., 7., 8., 9., 10.
};

void Affiche( ) {
   static int n_ieme_fois = 1;
   int k;

   printf("J'affiche pour la %d e fois\n", n_ieme_fois);
   n_ieme_fois++;

   for (k = 0; k < TAILLE; k++)
      printf (" tab[%d] = %f \n",  k,  donnees[k]);
}
```



```
int main( ) {
   int indice = -1;

   Affiche();
   Affiche();

   indice = ChercheVal(donnees, TAILLE, 4.);

   printf ("indice = %d , valeur = %f \n",
           indice, donnees[indice] );
   Affiche();
}
```

---

fichier cherche.c :

```
#include "cherche.h"
/* la var. donnees n'est pas accessible car static */

int ChercheVal(float tab[ ], int taille, float val) {
   int k;
   for (k = 0; k < taille; k++) {
      if (tab[k] == val) return k;
   }
   return -1;
}
```

Registres

— variables utilisées très fréquemment

— pour optimiser l'exécution

— inutiles avec compilateurs actuels

Structure de bloc

— code entre **{** et **}**

— peuvent être imbriqués

— chaque bloc peut contenir des définitions de variables

```
{
   int i, taille;
   ..........

   if (i > taille)
      return -1;
   else {
      int k;
      for (k = 0; k < taille; k++)
         ......;
   }
}
```


Eric LECOLINET
ENST - Paris

Tableaux et Pointeurs

Définition des tableaux

syntaxe :

type nom[*taille*];

type nom[*nb_lignes*][*nb_colonnes*];

etc ....

— éléments contigus

— indices allant de ***0*** à ***taille-1***

— tableaux multi-dimensionnels

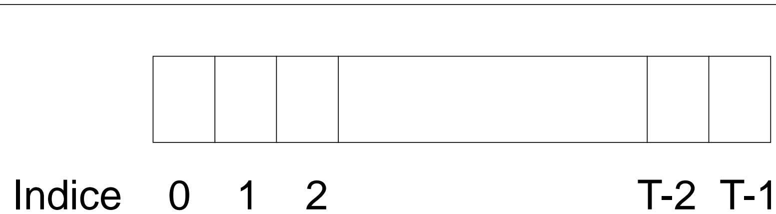
--> éléments placés en séquence

Exemples

```
int vect[10];
float mat[5][10];
double cube[5][10][20];
```


Eric LECOLINET
ENST - Paris

Vecteur :



Matrice 2D :

Ligne

0 | *0* | *1* | *2* | *C-2* | *C-1*

1 | *C* | *C+1* | *C+2*

L-1

Colonne 0 1 2 C-2 C-1

Adresses et Valeurs

Operateurs:

— **&***c* ---> ***adresse*** d'un objet *c*

— ******a* ---> ***valeur*** pointée par une adresse *a*

- *Adresse* du ième élément du tableau *vect*:

  &vect[i] == **&**(vect[i])

- *Adresse* du 1er elt. de *vect* :

  &vect[0] == vect

  (NB: cf. passage des tableaux en argument des fonctions)

- *Valeur* du ième elt. de *vect* :

  vect[i] == *(vect + i)

- *Adresse* de l'elt. (x,y) de *mat* :

  &mat[y][x]

- *Valeur* de l'elt. (x,y) de *mat* :

  mat[y][x] == *(mat + y * nb_col + x)

Initialisation des tableaux

initialisation possible lors de la définition :

```
int vect[10] = {1, 2, 3};
int toto[] = {1, 2, 3};
```

— les initialiseurs doivent être des constantes

— cardinal = 10 dans le 1er cas (mais seuls les trois premiers éléments sont initialisés)

— cardinal ***implicite*** = 3 dans le second cas

Tableaux multi-dimensionnels

float mat[5][10] = {{1, 2, 3, 4}, {5, 6}, {7, 8, 9}};

float truc[][10] = {{1, 2, 3, 4}, {5, 6}, {7, 8, 9}};

— initialisation ligne par ligne

— nombre de ***lignes*** éventuellement implicite (mais le nb. de colonnes doit être défini)

Paramètres des fonctions

```
void foo(int tab[], float mat[][10])
```

Omission possible :

— du cardinal de *tab*

— du nb. de lignes de *mat*

Définition en fait équivalente à:

```
void foo(int *tab, float *mat[10])
```

Pointeurs

— **variables** contenant une ***adresse***

— les pointeurs sont (généralement) ***typés***

Exemple

```
int i, *p;          /* p = pointeur d'entier */

p = &i;
*p = 5;
printf("%d %d \n",  *p, i);
i = 7;
printf("%d %d \n",  *p, i);
```

Passage d'arguments dans les fonctions

— RAPPEL: passage par **valeur**

— les arguments transmis sont **recopiés**

— sauf pour les tableaux

— pour éviter une recopie

— utiliser les ***pointeurs***

Exemple

```
void swap1( int i, int j ) {            /* ne fait rien */
   int  aux;
   aux = i;  i = j;  j = aux;
}

main() {
  int a = 5, b = 0;
  swap1(a, b);
  printf(" a = %d, b = %d \n", a, b);
}
```

```
void swap2( int *pi, int *pj ) {     /* echange les valeurs */
   int  aux;
   aux = *pi;  *pi = *pj;  *pj = aux;
}

main() {
  int a = 5, b = 0;
  swap2(&a, &b);
  printf(" a = %d, b = %d \n", a, b);
}
```

Un autre exemple: **scanf()**

```
   int i; float x;
   scanf("%d %f");
```

Pointeurs et Tableaux

Les pointeurs et tableaux sont intimement liés

```
float *p, *q, tab[] = {1, 2, 3, 4, 5};

p = &tab[2];
printf("%f %f \n",  *p, tab[2]);

p = tab + 2;
printf("%f %f \n",  *p, tab[2]);

*p = 0.0;
printf("%f %f \n",  *p, tab[2]);

q = p++;
*(q+2) = 777.77;
printf("%f %f \n",  *p, *q);
```

Valeur des éléments de tab ?

Remarques

— les notations: tab[i] et *(tab + i) sont équivalentes

— tab == &tab[0] == ***adresse*** du tableau

! Attention:

— un pointeur ***est*** une ***variable***

— un tableau ***n'est pas*** une variable !!!

```
p = tab;    p++;        /* CORRECT */
tab = p;    tab++;      ARCHI-FAUX !!!!
```

Arithmétique des pointeurs

```
char  tabchar[10], *pc = tabchar;
int     tabint[10],    *pi = tabint;
```

pc = pc + 5;   ---> 5 **caractères** plus loin (= 5 octets)

pi = pi + 5;   ---> 5 **entiers** plus loin (= 20 octets)

L'arithmétique sur les pointeurs n'a de sens que lorsque qu'ils sont typés (et de même type)

Type des pointeurs et void*

— **void *p** : pointeur non typé

— affectation entre pointeurs de types différents

- — possible via des "casts"
- — sport à haut risque !

```
int i;                    /* un "int" occupe 4 octets */
char *pc;                 /* un "char" occupe 1 octet */
pc = (char *)&i;          /* pc pointe sur le 1er octet de i */
```

— les tailles dépendent de l'OS et du processeur (parfois du compilateur)

— les int, float, etc. doivent commencer à certaines adresses (souvent un début de mot)

Chaînes de Caractères (Strings)

Suite de caractères terminée par 0 (valeur zéro)

On peut les définir de 2 manières:

— par un **tableau**
char s[] = "abcd";

— par un pointeur pointant sur un "**littéral**"
char *p ="abcd";

— dans les 2 cas le 0 final est rajouté automatiquement

un litteral est une sorte de tableau crée implicitement en mémoire statique

Attention:

— sizeof(s) = 5 : nombre de caractères + 0 final

— sizeof(p) = 4 : taille du pointeur (en 32 bits)

— le contenu des littéraux ne doit pas être modifié (utiliser des tableaux dans ce cas)

Exemples

Calcul de la longueur d'une chaîne de caractères

```
int strlen(const char *s) {              /* Version 1 */
    int n;
    for (n = 0; *s != 0; s++) n++;
    return n;
}

int strlen(const char *s) {              /* Version 2 */
    char *p = s;
    while (*p) p++;
    return p - s;           /* nombre de char entre s et p*/
}
```

Lire une chaîne depuis le terminal:

```
char *s;            // FAUX!!!
scanf("%s", s);

char s[20];         // DANGEREUX
scanf("%s", s);

char s[500];        // ACCEPTABLE
scanf("%s", s);
```

Pourquoi ?

Eric LECOLINET
ENST - Paris

Tableaux de pointeurs

Tableaux pointant sur des objets de taille quelconque

Exemple : tableau de chaines de caractères

```
char *jour[ ] = {
    "Lundi",
    "Mardi",
    "Mercredi",
    "Jeudi",
    "Vendredi",
    "Samedi",
    "Dimanche",
    NULL          // indique la fin du tableau
};
```

Autres exemples

— tableau d'objets que l'on veut trier

— tableau avec des lignes de longueur différentes

Eric LECOLINET
ENST - Paris

Mémoire dynamique

Pour créer, agrandir ou détruire **dynamiquement** des objets pendant l'éxécution.

Principe

— faire pointer un pointeur

— sur une "zone mémoire" spécialement créé

Fonctions principales

— **malloc**(), **calloc**() : allouent de la mémoire

— **free**() : libère la mémoire pointée

— **realloc**() : réalloue de la mémoire

— **sizeof**() : taille (en octets) d'un objet C

Pointeur indéfini

— un pointeur indéfini doit valoir **NULL**

— (NULL est une macro qui vaut 0)

Règle d'or

— toujours initialiser les pointeurs !

Eric LECOLINET
ENST - Paris


```
#include <stdio.h>      /* pour printf */
#include <stdlib.h>     /* pour malloc */

float *NewVect(int card) {
   float *v = (float*) malloc(card * sizeof(float));
   if (v == NULL)
        fprintf(stderr, "NewVect: No more memory\n");
   return v;
}

float *AddVect(float *v1, float *v2, int card) {
   float *v = NULL;
   if (!v1 || !v2) {
       fprintf(stderr, "AddVect: Null argument! \n");
   }
   else if ((v = NewVect(card))) {
       int k;
       for (k = 0; k < card; k++)  v[k] = v1[k] + v2[k];
   }
   return v;
}

int main() {
   float a[3] = {1., 2., 3.},  b[3] = {4., 5., 6.};
   float *x = NULL, *y = NULL;

   x = AddVect(a, b, 3);
   y = AddVect(x, a, 3);
}
```


Eric LECOLINET
ENST - Paris

Remarques

— l'exemple précédent présente deux inconvénient majeurs

— lequels ?

— que faudrait-il faire ?

Structures

Agrégat de données de types quelconques.

Définition d'un type structure

```
struct nom_struct {
    type1 variable1;
    type2 variable2;
    ......
};
```

Définition de variables de type structure

```
struct nom_struct var, tab[10], *p;
```

Il faut répéter le mot clé **struct**

Exemple: points dans le plan

```
struct POINT {
   double x;
   double y;
};

int main() {
   struct POINT p1 = {4.1, 7.3} , p2, ptab[10];

   p2 = p1;
   p2.x = 18.;
   p2.y = 21.;
   ptab[7] = p1;
   ptab[8].x = p2.x;
   .......
}
```

— initialisation similaire aux tableaux

— accès aux membres de la struct par **.**

— l'affectation entre 2 structs. est permise (si elles sont de même type)

— les structs. peuvent être paramètres des fonctions (et sont transmises par valeur)

Eric LECOLINET
ENST - Paris

Typedef

définit un nouveau nom de type

```
struct POINT {
   double x, y;
};

typedef struct POINT Point;

int main() {
   Point p1 = {4.1, 7.3} , p2, ptab[10];
   .......
}
```

Formes contractées:

```
typedef struct POINT {
   double x, y;
} Point;


typedef struct {
   double x, y;
} Point;
```


Eric LECOLINET
ENST - Paris

Structures imbriquées

```
#include <stdio.h>

typedef struct {
      Point p1;
      Point p2;
} Rect;

int main() {
      Point a = {1., 2.}, b = {3., 4.}
      Rect r1;
      Rect r2 = {{1., 2.}, {3., 4.}};

      r1.p1 = a;
      r1.p2 = b;

      printf(" rect1 = (%d, %d) x (%d, %d) \n",
            r1.p1.x, r1.p1.y, r1.p2.x, r1.p2.y);
}
```

Structures et Pointeurs

la notation: ***p->a*** équivaut à: ***(*p).a***


```
#include <stdio.h>
#include <stdlib.h>
#define BRIQUES_COUNT  6

typedef struct {
   char  *nom, *prenom;
   short  age;
   short  notes[BRIQUES_COUNT];
} Eleve;

int main() {
   Eleve *promo1 = NULL, *promo2 = NULL;
   int count1 = 0, count2 = 0;

   printf("Entrer le nombre d'élèves en Promo1 : ");
   scanf("%d", &count1);

   promo1 = (Eleve*)malloc(count1 * sizeof(Eleve));
   ... lire les données ...

   if (promo1) {                /* afficher les données */
     Eleve* p;
     for (p = promo1; p < promo1 + count1; p++) {
      printf("Eleve %s %s\n",  p->nom, p->prenom);
      printf("age : %d \n", p->age);
      ........
     }
   }
```

Remarque:

```
typedef struct {
   char  *nom, *prenom;
   short  age;
   short  notes[BRIQUES_COUNT];
} Eleve;
```

dans cet exemple, nom et prenom sont des pointeurs, ce qui suppose d’allouer la mémoire dynamiquement avec malloc() à la lecture

Exemple:

```
#include <string.h>

int k;
for (k = 0; k < count1; k++) {
    char nom[200], prenom[200];
    int age;
    scanf("%s %s %d", nom, prenom, &age);
    promo1[k].nom = strdup(nom);
    promo1[k].prenom = strdup(prenom);
    promo1[k].age = age;
 }
```


Récursivité

Les variables automatiques et les arguments sont stockés dans la **pile**

(sauf les tableaux passés en arguments)

Exemples

```
int fact(int n) {
   if (n <= 1)  return 1;
   else return n * fact(n - 1);
}

int fact2(int n) {
   return (n <= 1) ? 1 : n * fact2(n - 1);
}

void printd(int n) {
   if (n < 0) {
      putchar('-');
      n = -n;
   }
   if (n / 10)  printd(n / 10);
   putchar(n % 10 + '0');
}
```

Que fait printd ?

Parcours d'arbre

```
#include <stdio.h>

typedef struct NODE {
   struct NODE *left;     /* self-référence à NODE */
   char* val;
   struct NODE* right;
} Node;

void prefix(Node* n) {
   printf(" %s ", n->val);
   if (n->left)   prefix(n->left);
   if (n->right) prefix(n->right);
}

void infix(Node* n) {
   if (n->left)   infix(n->left);
   printf(" %s ", n->val);
   if (n->right) infix(n->right);
}

void postfix(Node* n) {
   if (n->left)   postfix(n->left);
   if (n->right) postfix(n->right);
   printf(" %s ", n->val);
}
```


Eric LECOLINET
ENST - Paris


```
int main() {
    /* feuilles */
    Node a = {NULL, "a", NULL};
    Node b = {NULL, "b", NULL};
    Node c = {NULL, "c", NULL};
    Node d = {NULL, "d", NULL};

    /* noeuds intermediaires */
    Node plus = {&a, "+", &b};
    Node div  = {&c, "/", &d};

    /* racine */
    Node star = {&plus, "*", &div};

    prefix(&star);
    printf("\n");
    infix(&star);
    printf("\n");
    postfix(&star);
    printf("\n");
    return 0;
}
```

Résultat ?

Eric LECOLINET
ENST - Paris