INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Réinterprétation Cours Langage C

Octobre 2015

M. Ivan Leplumey

Table des matières

I. PRINCIPALES BASES

A. Un premier exemple

Le langage C est un langage né vers les années 1970 dans les laboratoires Bell, langage impératif, basé sur l'affectation de variables

```
int main(){        /* Le point d'entrée d'un programme C */
        int x,y,z;   /* Déclarations de trois variables typées */

        x=5;
        y=8;

        z=x; /* z = 5 */      /* Algorithme d'inversion */
        x=y; /* x = 8 */
        y=z; /* y = 5 */

        printf("x=%d y=%d\n",x,y);
        return 0;
}
```

Programme 1 : Première version d'inversion de variables

```
int main(){        /* Le point d'entrée d'un programme C */
        int x,y;     /* Déclarations de deux variables typées */

        x=5;
        y=8;

        x = x + y; /* x = 13 */     /* Algorithme d'inversion */
        y = x – y; /* y = 5 */
        x = x – y; /* x = 8 */

        printf("x=%d y=%d\n",x,y);
        return 0;
}
```

Programme 2 : Seconde version d'inversion de variables, un algorithme différent

```
void swap(int *x, int *y){  /* Code compliqué */
        *x = *x + *y;
        *y = *x - *y;
        *x = *x - *y;
}
int main(){                /* Le point d'entrée d'un programme C */
        int x,y;           /* Déclarations de deux variables typées */

        x=5; y=8;

        swap(&x, &y); swap(&x, &y);
```

```
        printf("x=%d y=%d\n",x,y);
        return 0;
}
```

Programme 3 : Inversion de variables par le biais d'une fonction

B. Structuration d'un programme et traduction en langage machine

i. Structuration d'un programme

Un fichier source est saisi par un éditeur de texte (et non un traitement de texte), puis sauvegardé avant d'être traduit dans un langage compréhensible par le processeur.

Pourquoi découper un programme en plusieurs fichiers sources ?

- Conserver un temps de compilation raisonnable
- Permettre une réutilisabilité des modules séparés
- Améliorer la lisibilité grâce à la création d'unités sémantiques
- Maitriser la gestion de la version imprimée

Quelle est l'utilité des fichiers d'entête ?

- Donner les informations minimales d'un fichier source pour qu'il soit utilisé par un autre fichier source (permettre la compilation du deuxième fichier)
- Centraliser des informations pour éviter leur duplication

ii. Traduction en langage machine

La conversion d'un programme écrit en langage C en un programme binaire adapté à un ordinateur est réalisé en 3 phases à l'aide d'un logiciel comme **gcc** :

- **Phase 1** : le **pré-processing** (modification textuelle du code : une constante est recopiée dans le code à chaque endroit où elle est utilisée)
- **Phase 2** : la **compilation** (un module, une pièce de l'exécutable, est traduit en langage binaire, le module objet)
- **Phase 3** : l'**édition de liens** rassemble les modules pour construire l'exécutable binaire final

C. Elements du langage

i. Les commentaires

Un programme quel que soit son langage, est **toujours commenté**, les commentaires explicitent le rôle d'une variable ou d'instructions.

Les commentaires du langage C débutent par les caractères **/*** et terminent par ***/**.

Les commentaires se mettent au fur et à mesure de la conception d'un programme et pas après pour faire plaisir à l'enseignant; le commentaire mis à la volée coûte moins cher en réflexion et en temps qu'un commentaire mis a posteriori.

ii. Le pré-processeur

Directive **#include**

#include <stdio.h>

Remplacement de la directive par le fichier système stdio.h. Le fichier stdio.h se situe à un endroit connu du pré-processeur (/usr/include).

#include "monHeader.h"

Remplacement de la directive par le fichier monHeader.h. Le fichier monHeader.h se situe dans le répertoire courant.

Directive **#define**

#define TVA 19.6

Remplacement dans le code source de toutes les occurrences de TVA par 19.6. TVA va correspondre à une constante.

#define DOUBLE(x) 2*x

Remplacement dans le code source de toutes les occurrences de DOUBLE(x) par 2*x, x correspondant à un paramètre passé dans le code source.

Directive **#ifdef**

La directive **#ifdef** permet de choisir d'insérer une partie de code de manière conditionnelle en fonction de l'existence d'une constante de type **#define** et donc de s'adapter en fonction de l'environnement.

Directive **#ifndef**

La directive **#ifndef** permet de choisir d'insérer une partie de code de manière conditionnelle en fonction de la non-existence d'une constante de type **#define**.

Cette directive est utilisée pour éviter la multiple inclusion d'un même fichier d'entête dans un programme C. Elle doit être utilisé systématiquement dans tous fichiers d'entête.

iii. Les identificateurs

Un identificateur permet de désigner une constante/variable/objet/entité/fonction par un mot/nom qui **se doit d'être bien choisi pour apporter du sens** à l'objet qu'il désigne.

Règles de construction d'un identificateur

- commence par une lettre
- peut contenir des lettres minuscules et majuscules, des chiffres et le caractère souligné _
- 31 premiers caractères significatifs (norme ANSI, 8 caractères avant la normalisation)

iv. Les variables

Les variables doivent être déclarées avant utilisation et possèdent un type représentatif de leur contenu. Les types simples sont les suivants : **char**, **short int**, **int**, **long int**, **float**, **double**, **void**.

On peut aussi manipuler des variables de type tableau, regroupant des valeurs de même type sous un identificateur unique, un indice entier permettant d'accéder à une case donnée.

Les variables de type pointeurs (ou adresses) sont aussi accessibles. Elles permettent d'indiquer la position d'une variable en mémoire pour permettre par exemple, de modifier cette variable par une fonction intermédiaire.

```
int a ;          /* Déclaration d'un entier */
float t[50] ;    /* Déclaration d'un tableau de 50 nombres réels */
char *ptc ;      /* Déclaration d'un pointeur sur un caractère */
```

Les constantes entières peuvent s'exprimer en décimal (base 10), en octal (base 8) avec le préfixe 0 et en hexadécimal (base 16) avec le préfixe (0x). On ne peut pas exprimer directement un nombre en base 2.

```
a=15 ;   /* Base 10 */
```

```
a=017 ; /* Base 8 */
a=0xF ; /* Base 16 */
```

Les caractères sont codés sur un octet, valeur de 0 à 255, mais classiquement seules les premières 128 valeurs sont employées en référence à la table des caractères ASCII.

```
char c ;
c=0x61 ;          /* Code ASCII du a */
c='a' ;
```

b7 b6 b5 / BITS / b4 b3 b2 b1	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
	CONTROL		SYMBOLS NUMBERS		UPPER CASE		LOWER CASE	
0 0 0 0	0 NUL 0	16 DLE 20	32 SP 40	48 0 60	64 @ 100	80 P 120	96 ‘ 140	112 p 160
0 0 0 1	1 SOH 1	17 DC1 21	33 ! 41	49 1 61	65 A 101	81 Q 121	97 a 141	113 q 161

Extrait de la table des caractères ASCII

v. Structure d’un programme

Toute variable doit être définie avant son usage

La fonction est le principal élément structurant de la programmation C, le regroupement de ces fonctions dans de multiples fichiers séparés est très structurant

On ne peut pas mélanger instructions et déclarations en C contrairement au Java et C++

Une fonction de type **void** qui ne retourne donc aucun résultat, s’appelle une procédure

Le découpage en fonction permet d’utiliser une stratégie de division pour résoudre un problème, de factorisation les fonctionnalités pour éviter la redondance.

Un bloc est un ensemble de lignes commençant par une accolade ouvrante **{** , se terminant par une accolade fermante **}** et contenant un ensemble d’instructions

On peut déclarer des variables locales au début d’un bloc, variables connues uniquement du bloc et des blocs internes à ce bloc.

Un bloc peut contenir des blocs (structure hiérarchique).

Un programme se décompose en fichiers, qui se décomposent eux-mêmes en fonctions, qui peuvent contenir de multiples blocs.

Une fonction est définie par un bloc dont elle possède les propriétés; elle possède un nom, prend des données en paramètres et rend la plupart du temps un résultat.

La fonction de nom **main** est le point d’entrée d’une application, cela implique l’unicité de cette fonction.

```
int                 /* Type du résultat */
  fct2              /* Nom de la fonction */
    (int a)         /* Nom et type des paramètres */
        {           /* Début du bloc de la fonction */
```

```
        int j=2*a;   /* Déclaration variable locale */
        return j;    /* Retour résultat de la fonction */
    }                /* Fin du bloc de la fonction */
```

Anatomie d'une fonction

```
/* Inclusion des fichiers d'entête système */
#include <stdio.h>
...
/* Puis, inclusion des fichiers d'entête utilisateur */
#include "monHeader.h"
 …
/* Déclaration des variables globales et/ou des fonctions */
int glob;
int fct2(int a){int j=2*a; return j;}
…
int main(){
    /* Déclaration des variables locales */
    int i;
    /* Corps du programme principal */
    …
}
```

Anatomie d'un source C

vi. Structures de contrôle

- **if… else…**

L'instruction **if… else…** permet de tester des conditions.

```
a=5;
/* Ne pas confondre = et == */
if (a==5)       printf("a est égal à 5\n");
if (a=5)        printf("Ce message s'affiche aussi\n");

a=0;
if (a==0)       printf("a est égal à 0\n");
if (a=0)        printf("Ce message ne s'affiche pas\n");
else            printf("Par contre, celui-là s'affiche\n");
```

Attention aux confusions entre = et ==

```
a=5;
b=3;
if ((a==5) && (b==3)){   /* Début de bloc */
  int c=a+b;                  /*Déclaration variable locale c */
  printf("a est égal à 5\n");
  printf("b est égal à 3\n");
  printf("Somme : %d\n",c);
}                              /* Fin de bloc */
else {   /* Début de bloc */
  if ((a==2) || (b==2)) { printf("Une des variables vaut 2\n");
  }
}        /* Fin de bloc */
```

Exemple d'utilisation du if… else…

- **while**

```
                                /* Afficher les 100 premiers entiers, un par ligne */
#include <stdio.h>
int main(){                     /* Traduction par une boucle while */
        int i=1;                /* Initialisation */
        while (i<=100){         /* Test d'arrêt */
            printf("%d\n" ,i);
            i++;                /* Modifcation de la variable de boucle */
        }
        return 0;
}
```

Exemple d'utilisation du while

- **do… while**

```
                                /* Afficher les 100 premiers entiers, un par ligne */
#include <stdio.h>
int main(){                     /* Traduction par une boucle do-while */
        int i=0;                /* Initialisation */
        do {
            i++;                /* Modification de la variable de boucle */
            printf("%d\n" ,i);
        } while(i<100);         /* Test d'arrêt (toujours le réfléchir) */
        return 0;
}
```

- **Exemple d'utilisation du do… while**

- **for**

```
                                /* Afficher les 100 premiers entiers, un par ligne */
#include <stdio.h>
#define NBT 100
int main(){         /* Traduction par une boucle do-while */
        int i;
        /* Initialisation / test /évolution variable de boucle */
        for (i=1; i<=NBT; i=i+1) {
            printf("%d\n" ,i);
        }
        return 0;
}
```

- **Exemple d'utilisation du for**

- **switch**

```
#include <stdio.h>
int main(){
        char c;
        printf("Veuillez taper o ou n :");
        scanf("%c",&c);
        switch(c){
```

```
        case 'o':
        case 'O':       printf("Vous avez répondu oui\n");
                        break;
        case 'n':
        case 'N':       printf("Vous avez répondu non\n");
                        break;
        default:        printf("Saisie erronée, il faut taper o ou n\n");
    }
    return(0);
}
```

Exemple d'utilisation du switch

vii. Entrées-sorties

L'affichage à l'écran se fait à l'aide de la fonction : **int printf(const char *format,…);**

printf une fonction possédant un nombre variable de paramètres, le premier étant la chaine de format et … représente les paramètres à afficher.

```
/* Affichage chaine constante */    printf("Test d'affichage");
/* Affichage entier */              {int i=5; printf("%d",i);}
/* Affichage nombre réel */         {float fl=5.; printf("%f",fl);}
```

Exemple d'appel du printf

Les formats les plus courants sont le suivants :

- **%d** : entier en décimal
- **%c** : caractère
- **%f** : réel en simple précision
- **%lf** : réel en double précision
- **%x** : entier en hexadécimal
- **%O** : entier en octal
- **%4d** : entier sur 4 caractères
- **%5.2f** : réel sur 5 caractères et 2 chiffres après la virgule
- **%s** : chaine de caractères

L'entrée de données au clavier se fait à l'aide de la fonction : **int scanf(const char *format,…);**

scanf une fonction possédant un nombre variable de paramètres, le premier étant la chaine de format, … représentent les adresses des paramètres à entrer au clavier.

La valeur résultat de **scanf** permet de savoir si celui-ci s'est bien passé. Il rend le nombre de valeurs lues ; 0 en résultat indique souvent un problème.

```
    float f;
    scanf("%f",&f);
```

Exemple d'appel du scanf

L'oubli du **&** devant l'identificateur de la variable à entrer conduit le plus souvent à la catastrophe

Un espace devant le format permet de sauter espaces et sauts de ligne : **scanf(" %f",&f);**

On peut aussi faire des lectures multiples : **scanf("%d %d",&a,&b);**

D. Exemple complet

L'exemple qui suit, calcule le tableau d'amortissement d'un prêt bancaire.

L'utilisateur fournit préalablement les informations suivantes :

- Montant de l'emprunt
- Taux d'intérêt annuel
- Mensualité de remboursement

La structure du programme est la suivante :

- Programme principal ⇒ lecture des données utilisateur et lancement de la fonction de calcul
- Fonction de calcul du tableau d'amortissement

La compilation séparée n'est pas employée vu la taille de l'application, mais cela serait souhaitable si l'on désire factoriser ou réutiliser la fonction de calcul du tableau d'amortissement.

	A	B	C	D	E	F
1	Mois	Mensualité	Amortissement	Intérêts mensuels	Capital restant du	Intérêts cumulés
2	0				93000	
3	1	1250	928.38	321.63	92071.63	321.63
4	2	1250	931.59	318.41	91140.04	640.04
5	3	1250	934.81	315.19	90205.23	955.23
6	4	1250	938.04	311.96	89267.19	1267.19
7	5	1250	941.28	308.72	88325.91	1575.91
8	6	1250	944.54	305.46	87381.37	1881.37
9	7	1250	947.81	302.19	86433.56	2183.56

Résultats attendus après lecture dans un tableur comme Excel

```
#include <stdio.h>
#include <stdlib.h>

#define MIN(x,y) ((x)<(y)?(x):(y))
#define NBMOISPARAN 12

/* Fonction de conversion d'un nombre de mois en années et mois restants */
int convertirAnnee(int nbMois){ return nbMois/NBMOISPARAN;}
int resteMois(int nbMois){ return nbMois%NBMOISPARAN;}

void calculerTableauAmortissement(  double montantInitial, double tauxAnnuel,
                                    double mensualite, char *nom){
    FILE *sortie=NULL;
    double tauxMensuel;             /* Taux d'intérêt mensuel */
    double interetsMensuel;         /* Intérêt mensuel */
    double amortissementMensuel; /* Remboursement mensuel du capital */
    double mensualiteEffective;     /* Mensualité effective pour gérer les conditions d'arrêt */
    double resteARembourser;        /* Capital restant à rembourser */
    double coutPret;                /* Cout total du prêt au niveau intérêt */
    int nbMois;                     /* Nb mois de remboursement */

    tauxMensuel=tauxAnnuel/(NBMOISPARAN*100);

    /* Vérication de la pertinence de la mensualité */
    interetsMensuel=montantInitial*tauxMensuel;
```

```
        if (interetsMensuel>=mensualite){
            printf("Mensualité insuffisante (mensualite > %lf)\n",interetsMensuel);
        }

        if (nom==NULL){
            sortie=stdout;
        }
        else{
            sortie=fopen(nom,"w");
            if (sortie==NULL){
                    fprintf(stderr,"Echec d'ouverture du fichier de sortie (%s)\n",nom);
                    sortie=stdout;
            }
        }

        nbMois=0;
        coutPret=0;
        resteARembourser=montantInitial;
        fprintf(sortie,     "Mois;Mensualité;Amortissement;Intérêts mensuels;"
                            "Capital restant du;Intérêts cumulés\n");
        fprintf(sortie,"0;%.2lf;;;%lf;\n",tauxAnnuel,resteARembourser);
        while(resteARembourser>0){ /* Boucle sur les mois */
            nbMois++;
            interetsMensuel=resteARembourser*tauxMensuel;
            mensualiteEffective=MIN(mensualite,resteARembourser+interetsMensuel);
            amortissementMensuel=mensualiteEffective-interetsMensuel;
            resteARembourser-=amortissementMensuel;
            coutPret+=interetsMensuel;
            fprintf(sortie,"%d;%.2lf;%.2lf;%.2lf;%.2lf\n",
                    nbMois,mensualiteEffective,amortissementMensuel,interetsMensuel,
                    resteARembourser,coutPret);
        }

        printf("Duree de remboursement (%d mois): %d ans %d mois\n",
                    nbMois,convertirAnnee(nbMois),resteMois(nbMois));
        printf("Cout total du pret     : %.2lf euros\n",coutPret);

        if (sortie!=stdout) fclose(sortie);
}

int main(){  /* Programme principal */
        double montantInitial;      /* Capital emprunté */
        double tauxAnnuel;          /* Taux d'intérêts annuel */
        double mensualite;          /* Mensualité de remboursement */

        /* Interrogation utilisateur */
        printf("Montant pret       : ");      scanf("%lf",&montantInitial);
        printf("Taux interet annuel : ");   scanf("%lf",&tauxAnnuel);
        printf("Montant mensualite  : ");  scanf("%lf",&mensualite);
```

```
        /* Calcul du tableau d'amortissement */
        calculerTableauAmortissement(montantInitial, tauxAnnuel,  mensualite,
                                        "amortissement.csv");
        return 0;
}
```

Programme 4 : Application de calcul d'un tableau d'amortissement

II. STRUCTURES, TABLEAUX, ADRESSES

A. Structurer les données

Il existe plusieurs façons pour organiser les données :

- regrouper des constantes entières partageant une même sémantique (enum)
- regrouper des variables de type différents pour créer des entités complexes (struct)
- positionner des variables de type différents sur un même espace mémoire pour économiser cet espace (union)
- créer de nouveaux types pour simplifier la syntaxe (typedef)

i. Enumérations

Une variable énumérée est une variable entière regroupant un ensemble de constantes partageant une même sémantique. Plusieurs constantes peuvent posséder la même valeur entière.

```
/* Déclaration d'une variable jour de la semaine */
enum { lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche } jour;
/* Affectation d'une valeur à cette variable */
jour = mercredi;
printf("Jour : %d\n",jour); /* Jour : 2 */
```

Exemple d'énumération

```
/* Forme équivalente à la ligne enum de l'exemple précédent */
enum { lundi=0, mardi=1, mercredi=2, jeudi=3, vendredi=4, samedi=5, dimanche=6 } jour;
```

Enumération équivalente

```
/* Déclaration d'une variable jour de la semaine */
enum {lundi=12, mardi, mercredi=8, jeudi, vendredi=15, samedi=25, dimanche} jour;
/* Affectation d'une valeur à cette variable */
/* Pas de test de validité */
jour = 64;
/* Mais cette affectation n'a pas lieu d'être ! */
printf("Jour : %d %d %d %d\n", jour, mardi, jeudi, dimanche); /* Jour : 64 13 9 26*/
```

Autre exemple d'énumération

ii. Structures

Une structure est un ensemble complexe de variables regroupées sous un même identificateur.

```
/* Déclaration d'une variable pixel */
struct {
        int x;                  /* Coordonnées x,y du point */
        int y;
        unsigned char couleur_r;        /* Rouge */
        unsigned char couleur_v;        /* Vert */
```

```
        unsigned char couleur_b;        /* Bleu */
} pixel;
pixel.x = 100; /* Accès à un champ : . */
```

Exemple de structure

```
Pixel milieu (Pixel p1, Pixel p2) {
  /* Initialisation du résultat à p1 pour la couleur */
  /* Recopie complète de la structure avec = */
  Pixel resultat = p1;
  /* Calcul des coordonnées du point milieu */
  resultat.x = (p1.x + p2.x) / 2;
  resultat.y = (p1.y + p2.y) / 2;
  /* Le résultat est une structure complète */
  return (resultat);
}
```

Exemple d'utilisation d'une structure dans une fonction

iii. Union

Une variable de type union est un ensemble de variables occupant le même espace mémoire (utilisation rare).

```
/* Déclaration d'une variable de type union */
union {
        int choix1;
        float choix2;
} variable ;
variable.choix1 = 234;   /* Ecrasement réel */
printf("%d\n", variable.choix1);
variable.choix2 = 3.14;   /* Ecrasement entier */
printf("%f\n", variable.choix2);
```

Exemple d'union

iv. Définition de types

La définition d'un nouveau type permet de simplifier l'écriture d'un programme et d'ajouter des nommages des structures manipulées, plus explicites, plus proches de la sémantique du problème.

```
/* Définition d'un type Boolean */
/* Déclaration d'un nouveau type */
typedef enum {faux=0, vrai, FAUX=0, VRAI, false=0, true, FALSE=0, TRUE} Boolean;
/* - Une même valeur est utilisable plusieurs fois */
/* - Incrémentation si valeur non présente */
/* Déclaration et initialisation d'une variable */
Boolean jour = FALSE;
```

Exemple de définition de type

```
typedef struct {
        int x,y;
        unsigned char couleur_r;
        unsigned char couleur_v;
        unsigned char couleur_b;
} Pixel;
```

```
Pixel monpixel; /* Déclaration d'une variable de type Pixel */
/* Pixel vert en x= 245, y = 654 */
Pixel monpixel = {245, 654, 0, 255, 0};
/* Même initialisation champ par champ */
monpixel.x = 245;
monpixel.y = 654;
```

Second exemple de définition de type

B. Tableaux

Un tableau correspond à un ensemble de d'éléments (valeurs, structures) regroupés autour d'un identificateur unique. L'accès à l'une des valeurs s'effectue à l'aide d'un indice.

```
#define NBENT 15
#define NBLIG 25
#define NBCOL 80

int main(){
        /* Tableau de NBENT entiers : */
        int liste[NBENT];
        /* Tableau 2 dimensions : NBLIG lignes/NBCOL caractères */
        char ecran[NBLIG][NBCOL];

        /* Affecte la valeur 18 au 6ème élément */
        liste[5] = 18 ;
        /* La 5ème case est égale à la somme des cases 8 et 10*/
        liste[4] = liste[7] + liste[9] ;
        /* Le caractère de la 9ème ligne, 19ème colonne est un 'c' */
        ecran[8][18] = 'c' ;
        return(0);
}
```

Exemple de déclaration et d'utilisation de tableaux

Une chaine de caractères est un tableau de caractères se terminant par un marqueur de fin de chaine '\0' (ou 0).

```
char message[50]; /* 49 caractères disponibles */
scanf("%s",&message[0]);
scanf("%s",message);
scanf("%49s",message);
/* Le format %s de scanf ajoute automatiquement le marqueur en fin de chaîne */
printf("Message : %s\n", message);
/* Le format %s de printf s'appuie sur le marqueur pour arrêter l'affichage de la chaîne */
```

Exemple de déclaration et d'utilisation d'une chaine de caractères

C. Adresses

La mémoire d'un ordinateur correspond à un tableau d'octets possédant tous un indice (adresse mémoire de l'octet).
Un pointeur correspond à l'adresse mémoire du premier octet d'un objet (variable ou fonction).

Un pointeur non initialisé en langage C est source de dysfonctionnement, c'est une raison fréquente d'erreur dans les programme écrits en langage C.

```
char caract;
char *ptcar =&caract; /* & : opérateur adresse */
int nbent;
int *ptent = &nbent; /* & : opérateur adresse */
*ptcar='A';          /* * : opérateur de déférencement */
*ptent=18;           /* * : opérateur de déférencement */
printf("%c %d\n",caract,nbent); /* Affiche A 18 */
```

Exemple d'utilisation des pointeurs

Un pointeur de fonctions correspond à une variable contenant l'adresse de la première instruction d'une fonction, cela permet la construction d'architectures logicielles extensibles.

```
/* Déclaration d'un pointeur sur une fonction de deux paramètres et rendant un entier */
int (*ptrfct)( int a, int b);

int decroissant(int a, int b) {return (a<b);}
int croissant(int a, int b) {return (a>b);}

ptrfct=croissant ; /* Initialisation du pointeur de fonction */
```

Exemple d'utilisation d'un pointeur de fonction

Une variable en mémoire correspond à plusieurs réalités :

- un contenu (c1),
- une adresse pour la localiser (a1),
- un nom de variable (v1).

Le nom de la variable v1 est interprété suivant le contexte, soit comme un contenu c1, soit comme une adresse a1.

L'opérateur & appliqué à un nom de variable force l'interprétation de type adresse a1.

Pour permettre à une fonction A de modifier une variable appartenant à une fonction B appelant la fonction A, il est nécessaire de passer en paramètre l'adresse de cette variable.

```
#include <stdio.h>
/* Un premier programme sans réel intérêt */
void egal(int *ptval, int val){
  /* La variable adressée par ptval est initialisée à val */
  *ptval=val;     /* * opérateur de déférencement */
}

int main(){
      int v;                  /* v, la variable de travail */
      egal(&v,5);             /* ≡ v=5 */
      printf("v=%d\n",v);     /* Affiche v=5 */
      …
}
```

Exemple de fonction egal modifiant la variable v du programme principal

D. Les classes d'allocation

Les classes d'allocation sont un élément essentiel de la structuration d'une application.

Elles permettent de réduire la visibilité des variables/fonctions pour diminuer au maximum les interactions non désirées entre les différentes parties d'une application.

3 principaux mots-clés permettent de contrôler la classe d'allocation d'une variable : **auto** (classe par défaut) , **static**, **extern**

Les propriétés intéressantes d'une variable sont les suivantes :

- Quelle est la valeur initiale d'une variable ?
- Quelle est sa visibilité/portée dans le code ?
- Quelle est sa durée de vie ?
- A-t-elle des propriétés de rémanence qui lui permettent de conserver sa valeur entre deux utilisations ?

Les niveaux de visibilité se résument à trois niveaux :

- le niveau application qui correspond à l'ensemble du code,
- le niveau module qui concerne un seul module source,
- le niveau bloc, qui correspond à une région délimitée par des accolades.

Les variables **globales** de type **static** voient leur visibilité limitée au module dans lequel elles sont déclarées. On limite ainsi la visibilité et les modifications intempestives de ces variables. Ces variables globales sont initialisées automatiquement à 0.

Les variables **locales** non **static** à un bloc ne sont pas initialisées automatiquement (source de problème potentiel). Les variables **locales static** bénéficie de l'initialisation à 0 et possède la propriété de rémanence entre chaque utilisation du bloc auquel elles appartiennent.

E. Exemple

L'exemple qui suit, montre une application réalisant le tri d'un tableau de chaines de caractères suivant un critère variable exprimé à l'aide des pointeurs de fonctions.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TLMAX 100   /* Longueur maximum chaine */
#define NBC   6     /* Nombre de chaines */

/* Trois critères de tri pour les chaines */
int decroissant(char *a, char *b){   /* Décroissant suivant ordre ASCII */
  return(strcmp(a,b)<0);
}
int croissant(char *a, char *b){     /* Croissant suivant ordre ASCII */
  return(strcmp(b,a)<0);
}
int croissantI(char *a, char *b){    /* Croissant, indifférence min./maj. */
  char ac[TLMAX], bc[TLMAX];
  strcpy(ac,a); strlwr(ac);   /* Recopie + transformation en minuscules */
  strcpy(bc,b); strlwr(bc);   /* Recopie + transformation en minuscules */
  return(strcmp(bc,ac)<0);
}

/* Fonction effectuant le tri d'un tbleau de chaines de caractères */
void trier(char tb[ ][TLMAX], int nb, int (*critere)(char *, char *)){
  char temp[TLMAX];  /* Chaine temporaire */
  int i, j, irec;    /* Indices de balayage/repère des chaines */
```

```
  for (i=0; i<NBC-1; i++){    /* Recherche chaine resp. fonction critere*/
    irec=i;
    for (j=i+1; j<NBC; j++){
      if (critere(tb[irec],tb[j])) { irec=j; }
    } /* for */
    if (irec!=i){ /* Inversion des chaines si nécessaire */
      strcpy(temp,tb[irec]);
      strcpy(tb[irec],tb[i]);
      strcpy(tb[i],temp);
    } /* if */
  } /* for */
} /* trier */

int main(){ /* Programme principal */
    char t[NBC][TLMAX]={"Rennes", "Paris", "Lyon", "Saint Etienne", "Verdun", "marseille"};
    trier(t, NBC,croissant);  /* Trier le tableau */
    imprimer(t, NBC);          /* Imprimer le tableau */
    return 0;
}
```

Programme 5 : Tri d'un tableau de chaine de caractères suivant des critères variables

L'utilité des pointeurs de fonctions dans une architecture logicielle permet de :

- minimiser la taille du code en factorisant les parties communes (il n' y a pas une fonction **trier** par critère),
- minimiser les dépendances et les connaissances entre les fonctions ou modules (la fonction **trier** ne connait rien sur ce qui se passe dans la fonction de comparaison **critere**),
- maximiser la réutilisabilité d'un module (d'autres applications peuvent réutiliser le module **trier** avec d'autres critères) ; il faudrait cependant exporter cette fonction dans un module C séparé et créer le fichier d'entête correspondant,
- enrichir les données si nécessaire en y introduisant des références aux fonctions ; ce dernier point n'est pas présent dans l'exemple.

III. FICHIERS, GESTION MEMOIRE

A. Entrées-sorties dans un fichier

Il existe 2 types de fichiers couramment utilisés :

- Fichier texte observable par un éditeur de texte,
- Fichier binaire exploitable par une application dédiée (jpg, wav…).

Leur utilisation se passe 3 phases :

- Ouverture du fichier ⇒ **fopen**
- Lire/écrire des données
  - dans un fichier texte ⇒ **fprintf, fscanf**
  - dans un fichier binaire⇒ **fwrite, fread**
- Fermeture du fichier ⇒ **fclose**

Le type FILE encapsule un descripteur (correspondant à une structure **struct**) permettant l'utilisation d'un fichier texte ou binaire. Il n'est pas nécessaire de connaitre les champs de cette structure; les fonctions fournies suffisent à opérer toutes les manipulations nécessaires.

```
#include <stdio.h>
/* Déclaration d'un pointeur sur un descripteur de fichier */
FILE * monfichier;
```

Déclaration d'un descripteur de fichier

Les mots de **flots**, **flux** ou **streams** sont employés pour parler des entrées/sorties en cours
Il existe 3 flots ouverts automatiquement lors de l'exécution d'un programme (**stdio.h**) :

- **stdin** : le flot d'entrée attaché par défaut au clavier
- **stdout** : le flot de sortie attaché par défaut à l'écran
- **stderr** : le flot de sortie d'erreur attaché par défaut à l'écran (qui doit être systématiquement utilisé pour les erreurs)

```
#include <stdio.h>
#define ERROUV      567     /* Centralisation constantes */
#define NOM    "donnees.txt"
FILE *monfichier=NULL;      /* Ouverture du fichier */
if ((monfichier = fopen(NOM,"r")) == (FILE *) NULL) {
   fprintf(stderr,"erreur ouverture fichier %s\n",NOM);
   exit(ERROUV);
} /* Effectuer systématiquement le test */
...
/* Fermeture du fichier */
if (fclose(monfichier)) {
   fprintf(stderr,"problème à la fermeture du fichier\n");
}
```

Exemple d'ouverture et de fermeture d'un fichier

```
int main(){
  char tampon[TMAX], trash[TMAX];
  FILE * pFile=NULL;
  /* Ouverture du fichier */
  pFile = fopen ("leplumey150730.ged","r");
  if (pFile==NULL){fprintf(stderr,"Erreur\n"); return 1;}
 /* Balayage du fichier */
  while (!feof(pFile)){ /*  1 NAME Jean Marie/BOULLET/  */
     if (fscanf(pFile," 1 NAME %*[^/\r\n]/%[^/]/%[ \r\n]",
           tampon,trash)==2) {printf("Nom: %s\n", tampon);}
     else fscanf(pFile,"%*[^\r\n]");    /*  Nom: BOULLET  */
  }
  fclose(pFile);
  return 0;
}
```

Exemple d'extraction de données dans un fichier texte au format Gedcom

B. Organisation de la mémoire, appel de fonctions

Une **pile** est une structure favorisant l'utilisation des données situées en sommet.

Lors de l'exécution d'un programme, la mémorisation des informations pour aller dans une fonction et en revenir se situe dans une structure de pile qui mémorise dans l'ordre suivant :

- les variables locales (en sommet de pile),
- l'adresse de retour,
- les paramètres,
- le résultat.

On peut ainsi enchainer les appels de fonction de manière imbriquée voire récursive.

```
float puissance (int nb,  float val) {      /* Fonction de calcul d'une puissance d'un nombre */
        int i;
        float res = 1.0;
        for (i=0; i< nb; i++)
            res *= val;
        return(res);
}
Int main(){
        float resf=puissance(3,12.1) ;      /* Calcul de 12.1 à la puissance 3 */
        return 0 ;
}
```

Exemple d'un appel à une fonction

Dans l'exemple précédent, on commence par la réservation d'une zone pour y insérer le résultat effectif , la place pour un nombre réel. Une copie des paramètres effectifs, 3 et 12.1, (pour être associé aux paramètres formels val, nb) est disposée dans la pile. L'adresse de retour permettant de revenir au programme principal est ensuite empilée. La réservation dans la pile des variables locales i et res est effectuée. Lors de la fin de l'appel à cette fonction, les zones allouées dans la pile seront libérées dans l'ordre inverse.

```
#include <stdio.h>
int main(){
        int i=666, liste[15]; /* Taille liste : 15 entiers */
        /* Débordement de 1 : écrasement de i */
        liste[15]=78;
         /* Écrasement de l'adresse de retour */
        liste[22]=78;
        return(0);
}
```

Utilisation de débordement de tableaux variables locales pour détruire les données connexes

C. Allocation dynamique

i. Notions de base

L'utilité principale de l'allocation dynamique est d'allouer des zones mémoires dont la taille n'est connue que lors de l'exécution, et qui peuvent de plus varier au cours de l'exécution.

La durée de vie d'une telle zone mémoire commence lors de l'instruction d'allocation et se termine lors de l'instruction de libération, elle est donc sous le contrôle direct du programme.

L'opérateur **sizeof** fournit la taille en octets d'une variable ou d'un type.

```
float val = 3.67;
printf("%d\n", sizeof(val));      /* 4 */
printf("%d\n", sizeof(float));    /* 4 */
```

Exemple d'utilisation de l'opérateur sizeof

La fonction **malloc** (memory allocation) effectue l'allocation d'une zone mémoire d'une taille exprimée en octets. **Dans un programme fonctionnel, il est primordial de tester que l'allocation mémoire s'est bien déroulée.**

```
/* Définition d'un code d'erreur */
#define ERROR_ALLOC 43567
/* Déclaration du pointeur */
float * ptrval;
/* Allocation du nombre réel */
if (( ptrval = (float *) malloc (sizeof(float))) == NULL){
        fprintf(stderr, "Problème allocation mémoire\n");
        exit(ERROR_ALLOC);
}
```

Exemple d'utilisation de la fonction malloc

La fonction **free** effectue la libération d'une zone mémoire allouée préalablement par **malloc** ou une autre fonction d'allocation de mémoire dynamique. Si le paramètre passé à la fonction **free** n'est pas une valeur issue d'un appel à une fonction d'allocation de mémoire dynamique (**malloc**…), le programme peut avoir un **comportement imprévisible**.

```
/* Libération de mémoire allouée */
free(ptrval) ;
```

Exemple d'utilisation de la fonction free

Il faut libérer (**free**) la mémoire allouée (**malloc**) sous peine de créer des fuites mémoires et de créer de futurs dysfonctionnements d'allocation (programme dont la durée d'exécution est longue : par exemple, serveurs).

Une mauvaise manipulation de l'allocation de mémoire dynamique conduit à des erreurs difficiles à détecter, l''erreur ne se produit classiquement pas immédiatement.

La mémoire libérée n'est pas réinitialisée ; on peut ainsi donc toujours accéder à la zone libérée en ayant l'impression que tout va bien, et une nouvelle allocation ultérieure suivie d'une modification des données allouées conduit alors au dysfonctionnement prévisible.

ii. Les tableaux à plusieurs dimensions

Les tableaux à une ou plusieurs dimensions sont l'une des structures utilisant très souvent l'allocation dynamique. Lors d'une exécution, par exemple suite à une lecture d'un fichier, on obtient l'information nécessaire au dimensionnement des tableaux manipulés.

Utiliser des tableaux alloués statiquement de taille importante pour être sûr d'avoir assez de place quelle que soit la taille réelle nécessaire **n'est pas la bonne solution** : **un jour ou l'autre, cette taille se révèlera insuffisante… L'allocation dynamique est la bonne solution.**

La fonction **calloc** effectue l'allocation d'une zone mémoire **initialisée à 0** correspondant à un tableau.

```
/* Définition d'un code d'erreur */
```

```
#define ERROR_ALLOC 43567
/* Définition de la constante de taille */
#define TAILLE 1000
/* Déclaration du pointeur */
char * ptrchar;
/* Allocation du nombre tableau de caractères */
if (( ptrchar = (char *) calloc (TAILLE, sizeof(char)))==NULL) {
        fprintf(stderr,"Problème allocation mémoire\n");
        exit(ERROR_ALLOC);
}           /* Equivalent à : char ptrchar[TAILLE]; */
```

Exemple peu pertinent d'utilisation de la fonction calloc

```
/* Définition d'un code d'erreur */
#define ERROR_ALLOC 43567
/* Déclarations */
char * ptrchar; int taille;
printf("Taille du tableau : ");     /* Saisie de la taille */
scanf(" %d",&taille);
/* Allocation du nombre tableau de caractères */
if (( ptrchar = (char *) calloc (taille, sizeof(char))) == NULL) {
        fprintf(stderr,"Problème allocation mémoire\n");
        exit(ERROR_ALLOC);
}
```

Exemple pertinent d'utilisation de la fonction calloc

La fonction **realloc** effectue la ré-allocation d'une zone mémoire d'une taille exprimée en octets. Cette fonction permet l'augmentation ou la diminution de taille d'un tableau en effectuant si nécessaire, la recopie de la partie déjà allouée.

La fonction **realloc** fonctionne de la façon suivante :

- Si la taille est en diminution, on libère l'espace réservé en trop.
- A l'opposé, si la taille est en augmentation, on a alors deux possibilités. Soit il y a suffisamment de place libre dans la continuité de la zone préalablement allouée pour satisfaire à l'accroissement de taille, on peut alors effectuer le redimensionnement sans recopie. Soit on alloue une nouvelle zone de la taille demandée, on recopie alors la zone précédente en début de cette nouvelle zone allouée, le coût de ces recopies peut diminuer de manière très significative les performances d'une application.

Un tableau bidimensionnel d'entiers de largeur lg et de hauteur ht occupe une place de lg*ht*sizeof(int) octets.

Si les lignes sont rangées consécutivement en mémoire, il peut être simulé par un tableau monodimensionnel avec une formule d'accès : j * lg + i, où j représente l'indice de la ligne et i l'indice de colonne.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
        int i,j; int lg,ht; int *t2D;
```

```
        printf("Largeur/hauteur : "); scanf("%d/%d",&lg,&ht);
        if ((t2D=(int *) calloc(lg*ht,sizeof(int))) == NULL){ …}
        /* Accès au point i,j par une formule de conversion */
        j=2; i=5; t2D[j*lg+i]=5; /* Faible lisibilité des accès */
        free(t2D); /* Libération de la zone mémoire */
        return(0);
}
```

Exemple d'utilisation d'un tableau bidimensionnel géré de manière monodimensionnel

```
#include <stdlib.h>
/* Allocation tableau 2D */
int **allouerTableau2D(int lg, int ht){
        int i;
        int **h; /* Tableau à deux dimensions */
        /* Allocation de la table d'accès aux lignes */
        if ((h=(int **) calloc(ht,sizeof(int *)))==NULL){…}
        /* Allocation des tables correspondant aux lignes */
        for (i=0; i<ht; i++)
            if ((h[i]=(int *) calloc(lg,sizeof(int)))==NULL){… }
        return(h);
}
/* Libération tableau 2D */
void libererTableau2D(int **h, int lg, int ht){
        int i;
        /* Libération des tables correspondant aux lignes */
        for (i=0; i<ht; i++)
            free(h[i]);
        /* Libération de la table d'accès aux lignes */
        free(h);
}
int main(){
        int i,j; int lg,ht; int **t2D;
        printf("Largeur/hauteur : ");
        scanf("%d/%d",&lg,&ht);
        /* Allocation du tableau bidimensionnel */
        t2D=allouerTableau2D(lg,ht);
        /* Accès au point i,j */
        j=2; i=5; t2D[j][i]=5;
        libererTableau(t2D, lg, ht);
        return(0);
}
```

Exemple d'utilisation d'un tableau bidimensionnel géré de manière bidimensionnel

D. Qualité logicielle

Un programme est toujours commenté.

Un commentaire décrivant le rôle est systématiquement associé à :

- chaque déclaration de variable
- chaque début de fonction/procédure

On ne doit pas trouver 10 lignes consécutives de code sans commentaire

Le commentaire se met avant d'écrire le code, et pas deux heures après.

La structure d'un programme doit être visuellement accessible.

Le code est indenté pour permettre une plus grande lisibilité entre autre de la structure.

Des lignes blanches peuvent être insérées pour séparer les régions.

Il est conseillé de pouvoir différencier visuellement variables, constantes (tout en majuscule) et types (majuscule en première lettre).

Un programme doit être lisible.

Toute variable/ constante/ fonction/ méthode/ module possède un identificateur significatif de son rôle/contenu (toto est à proscrire).

Un programme doit être structuré.

Les constantes sont centralisés, et non dispersées, ni dupliquées dans le code.

Une fonctionnalité est centralisée, la duplication de code doit être proscrite. Les fonctions/méthodes permettent d'obtenir cette propriété.

Les modules (fichiers sources .c) doivent être utilisés pour structurer un logiciel. Un logiciel dont le source dépasse les 20 pages peut être suspecté de ne pas utiliser suffisamment la notion de modules.

Une gestion fine des erreurs est au cœur d'un programme de qualité.

Les messages d'erreur sont affichés sur STDERR et non pas sur STDOUT.

Le résultat de l'appel d'une fonction système est systématiquement testé pour détecter au plus vite les erreurs (*malloc*, *fopen*…).

Initialiser systématiquement les pointeurs à NULL est une bonne précaution.

IV. QUELQUES QUESTIONS DE QCM

Préprocesseur

Soit la macro suivante :

```
#define MT(x) x*2
```

qu'affiche le programme suivant ?

```
x=8 ;
printf("%d %d %d\n",MT(x),MT(x+5),MT(1+x));
```

- A☐ : `16 18 17`
- B☐ : `16 18 18`
- C☐ : `8 13 9`
- D☐ : `16 26 18`

Préprocesseur **(difficulté moyenne)**

Soit le programme suivant :

```
#include <stdio.h>          /* Ligne 1 */
#define MAC1 2;             /* Ligne 2 */
#define MAC2 x);            /* Ligne 3 */
int main(){                 /* Ligne 4 */
        int x=MAC1;         /* Ligne 5 */
        printf("%d\n",MAC2  /* Ligne 6 */
        return MAC1         /* Ligne 7 */
}
```

que se passe-t-il lors de la compilation ?

- A☐ : pas d'erreur de compilation
- B☐ : erreur de compilation sur la ligne 5
- C☐ : erreur de compilation sur la ligne 6
- D☐ : erreur de compilation sur la ligne 7

Les boucles

Qu'affiche la séquence suivante :

```
int k;
for(k=0;k<5;);
{       k++;
        printf("%d ",k);
}
```

- A☐ : `0 1 2 3 4`
- B☐ : `1 2 3 4 5`
- C☐ : elle n'affiche rien
- D☐ : autre

Les choix

Qu'affiche la séquence suivante :

```
int a=5,b=2,r;
switch(a/b){
        case 0: r=0;break;
        case 1: r=1;break;
        case 2: r=2;break;
        default: r=3;
}
printf("%d",r);
```

- A☐ : `0`
- B☐ : `1`
- C☐ : `2`
- D☐ : `3`

Les fichiers

Soit la fonction `lireEntier` :

```
int lireEntier(FILE *f, int *nb){
   int res=fscanf(f," %d",nb);
   return(res);
}
```

quelle assertion vous semble fausse ?

- A☐ : le fichier désigné par `f` est ouvert en lecture
- B☐ : le fichier désigné par `f` est en partie lisible par un éditeur de texte
- C☐ : la fonction `lireEntier` rend en résultat la variable `res` qui correspond à la valeur lue dans le fichier
- D☐ : `res=0` implique une lecture défaillante

INSA Rennes
20 Avenue des Buttes de Coësmes
CS 70839
35708 Rennes Cedex 7

Tél. +33 (0) 2 23 23 82 00
Fax +33 (0) 2 23 23 83 96

www.insa-rennes.fr

INSA

UNIVERSITÉ EUROPÉENNE DE BRETAGNE

Commission des Titres d'Ingénieur

RÉPUBLIQUE FRANÇAISE

MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE