Programmation et structures de données en C cours 1: Introduction et rappels

Jean-Lou Desbarbieux, Stéphane Doncieux
et Mathilde Carpentier
LU2IN018 SU 2022/2023

Sommaire

Présentation du module

- Module avancé (niveau 200) de **3 ECTS**
- Objectifs principaux :
  - Consolidation des notions de gestion explicite de la mémoire
  - Structure de données autoréférentielles : listes, arbres, ...
  - Bonnes habitudes de programmation (tests, structuration, débogage)
  - Entrées/sorties et manipulation de fichiers
  - Introduction à la généricité
- Itinéraire :
  - Notions d'algorithmique

Pensez à vous inscrire...

... si ce n'est pas encore fait

Évaluation

- 50% pour l'examen final
- 25% pour le partiel
- 25% pour une évaluation des TME (participation et mini-soutenances)

Les annales des examens et partiels sont sur le site de l'UE. Attention, l'UE a évolué au cours des dernières années et si le programme est resté globalement le même, il y a tout de même eu quelques changements. Un de ces changements est que, avant 2015, les documents étaient autorisés, depuis cette date, **seul le memento est autorisé**.

Calendrier

- 6 semaines de cours, 6 semaines de TD, 6 semaine de TME
- Cours une semaine sur 2
- TME une semaine et TD l'autre, sur le même créneau horaire
- Les TME commencent cette semaine !
- Les TD commencent la semaine du 26 septembre
- Le partiel aura lieu la semaine du 7 novembre
- L'examen aura lieu la semaine du 9 janvier

Tutorat : à venir, surveiller sur Moodle.

Calendrier (à titre indicatif)

Séance 1. Rappels pointeurs/structures/listes chaînées
Séance 2. Débogage, compilation séparée et outils de développement
Séance 3. Chaînes de caractères et entrées/sorties
Séance 4. Arbres
Séance 5. LibC et introduction à la généricité
Séance 6. Conclusion et Révisions

Bibliographie et outils

- Le langage C : norme ANSI. Brian W. Kernighan & Denis M. Ritchie, Dunod
- Programmer en langage C. Claude Delannoy, Eyrolles
- C : a reference manual. Samuel P. Harbison & Guy L. Steele Jr., Prentice Hall
- C : a software engineering approach. Peter A. Darnell & Philip E. Margolis, Springer
- ...

Environnement : Linux.
Outils informatiques utilisés :

- éditeurs : emacs/vi/gvim/gedit
- compilateur : gcc
- débogueur : gdb, ddd
- automatisation de la compilation : Makefile

Le Langage C : historique

- Le langage C a été inventé en 1972 par Dennis Ritchie et Ken Thompson (AT&T Bell Laboratories) pour réécrire Unix et développer des programmes sous Unix.
- En 1978, Brian Kernighan et Dennis Ritchie publient la définition classique du C dans le livre The C Programming language .
- C est une norme ANSI (ANSI-C) depuis 1989 et un standard ISO depuis 1990, standard étendu en 1999 (C99), 2011 (C11) et en 2018 (C17), future version à venir (C23 ?).
- Toutes les distributions Linux fournissent des compilateurs C.

Caractéristiques du langage :

- impératif
- bas-niveau
- typé

Comparaison de différents langages

	Energy
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) Jruby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

	Time
(c) C	1.00
(c) Rust	1.04
(c) C++	1.56
(c) Ada	1.85
(v) Java	1.89
(c) Chapel	2.14
(c) Go	2.83
(c) Pascal	3.02
(c) Ocaml	3.09
(v) C#	3.14
(v) Lisp	3.40
(c) Haskell	3.55
(c) Swift	4.20
(c) Fortran	4.20
(v) F#	6.30
(i) JavaScript	6.52
(i) Dart	6.67
(v) Racket	11.27
(i) Hack	26.99
(i) PHP	27.64
(v) Erlang	36.71
(i) Jruby	43.44
(i) TypeScript	46.20
(i) Ruby	59.34
(i) Perl	65.79
(i) Python	71.90
(i) Lua	82.91

	Mb
(c) Pascal	1.00
(c) Go	1.05
(c) C	1.17
(c) Fortran	1.24
(c) C++	1.34
(c) Ada	1.47
(c) Rust	1.54
(v) Lisp	1.92
(c) Haskell	2.45
(i) PHP	2.57
(c) Swift	2.71
(i) Python	2.80
(c) Ocaml	2.82
(v) C#	2.85
(i) Hack	3.34
(v) Racket	3.52
(i) Ruby	3.97
(c) Chapel	4.00
(v) F#	4.25
(i) JavaScript	4.59
(i) TypeScript	4.69
(v) Java	6.01
(i) Perl	6.62
(i) Lua	6.72
(v) Erlang	7.20
(i) Dart	8.64
(i) Jruby	19.84

Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., & Saraiva, J. (2017). Energy efficiency across programming languages : how do energy, time, and memory relate ?. In Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (pp. 256-267).

Rappels au travers d’exemples

Exemple de programme (1)

Fichier "HelloWorld.c" (éditeur de texte)

```
#include <stdio.h>

int main(void) {
   printf("Hello Sorbonne !\n");
   return 0;
}
```

Compilation (dans un terminal)

```
gcc -o HelloWorld HelloWorld.c
```

Exécution (dans un terminal)

```
./HelloWorld
```

Exemple de programme (2)

```c
#include <stdio.h>

#define NBGROUPES 12

int groupes[NBGROUPES]={27, 24, 30, 25, 28, 29, 30, 29, 30,
                        30, 29, 30};
void afficher_groupes(void) {
  int i;
  for(i=0;i<NBGROUPES; i++) {
    printf("Groupe %d: %d etudiants\n",i,groupes[i]);
  }
}
int total_etudiants(void) {
  int i;
  int nb=0;
  for(i=0;i<NBGROUPES; i++) {
    nb+=groupes[i];
  }
  return nb;
}
int main(void) {
  afficher_groupes();
  printf("Nombre total d'etudiants: %d\n",total_etudiants());
  return 0;
}
```

Exemple de programme (3)

```
#include <stdio.h>
#define NBGROUPES 12
#define MAXSIZE 34
int groupes[NBGROUPES]={27, 24, 30, 25, 28, 29, 30, 29, 30,
                        30, 29, 30};
int ajouter_etudiant(void) {
  int i=0;
  while(i<NBGROUPES) {
    if (groupes[i]<MAXSIZE) {
      groupes[i]++;
      return i;}
    i++;
  }
  return -1;
}
int main(void) {
  int g=ajouter_etudiant();
  if (g==-1) {
    printf("Tous les groupes sont pleins, l'ajout a echoue\n");
  }
  else {
    printf("L'etudiant.e a ete ajoute.e dans le groupe %d\n",g);
  }
  return 0;
}
```

Exemple de programme (4)

```
void afficher_filiere(int n) {
  printf("Le groupe %d est dans la filiere : ",n);
  switch(n) {
  case 3: case 6:
    printf("mono\n");
    break;
  case 2: case 7: case 8:
    printf("majeure\n");
    break;
  case 1: case 4: case 5:
    printf("mono + double majeure/cursus\n");
    break;
  case 9:
    printf("DANT + double majeure\n");
    break;
  case 12:
    printf("double majeure\n");
    break;
  default:
    printf("<filiere inconnue !>");
  }
}
int main(void) {
  afficher_filiere(3);
  return 0;
}
```

Pointeurs et gestion de la mémoire

Pointeurs et gestion dynamique de la mémoire

Chaque variable occupe un certain espace en mémoire (en couleur ce qu'il faut retenir)

Type	Signification	Taille (o)	Plage de valeurs
char	Caractère	1	-128 à 127
unsigned char	Caractère	1	0 à 255
short int	Entier court	2	-32768 à 32767
uns. short int	Entier court non s.	2	0 à 65535
int	Entier	2 (16 b) 4(32 et 64 b)	-32768 à 32767 -2 147 483 648 à 2 147 483 647
unsigned int	Entier non signé	2 (16 b) 4 (32 et 64 b)	0 à 65 535 0 à 4 294 967 295
long int	Entier long	4 8 (64 b)	-2 147 483 648 à 2 147 483 647 -9 223 372 036 854 775 080 à 9 223 372 036 854 775 807
uns. long int	Entier long non s.	4	0 à 4 294 967 295

Type	Signification	Taille (o)	Plage de valeurs
float	Simple précision	4	+/- 1.175494e-38 à 3.402823e+38
double	Double précision	8	+/- 2.225074e-308 à 1.797693e+308
long double	Double préc. long	12	+/- 3.362103e-4932 à 1.189731e+4932

Notion de pointeur (rappel)

- Chaque variable correspond à un bloc mémoire d'une taille connue et à un emplacement donné.
- Cet emplacement est l'adresse de la variable.
- Un pointeur est une variable dont le contenu est une valeur qui est elle-même une adresse mémoire.
- Déclaration : on ajoute "*" au type de la donnée pointée. Exemples : `int *i; char *c;`
- L'adresse d'une variable est récupérée grâce à l'opérateur "&" :

```
int i=2;
int *l; /* l est un pointeur sur entier */
l=&i; /* l pointe sur i */
```

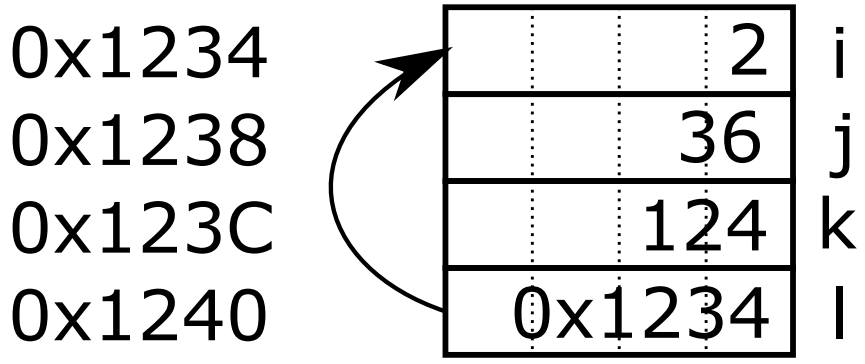
- La valeur stockée en mémoire à l'adresse indiquée par un pointeur peut être récupérée grâce à l'opérateur "*" (on parle de déréférenciation). Ex :

```
int i=2;
int *l=&i; /* l est un pointeur sur i */
*l=3; /* la valeur de i passe a 3 */
```

- NULL (ou 0) est une valeur spéciale qui ne pointe pas sur une zone mémoire valide (utilisé pour indiquer qu'un pointeur n'est pas encore

Exemple

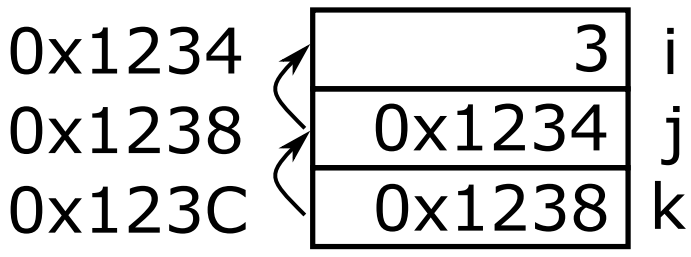
```
int  i=2,j=36,k=124;
int  *l=&i;
```



Dans cet exemple, l contient l'adresse de la variable i.

Les pointeurs de pointeurs de ...

```
int  i=3;
int  *j=&i;
int  **k=&j;
```



k est un pointeur de pointeur sur entier. Il contient donc l'adresse d'une variable contenant l'adresse d'une variable de type int.

Il est également possible de définir des pointeurs de pointeurs de pointeurs sur un type donné et ainsi de suite...

Tableaux & Pointeurs

Une référence à un tableau est équivalente à un pointeur constant sur son premier élément !

```
int a[10];
int *b=a;
```

$a \iff b \iff \&a[0] \iff \&b[0]$

La notation avec [] est équivalente à l'addition de l'indice :

```
int a[10];
int *b=a;
```

$a[2] \iff *(a+2) \iff b[2] \iff *(b+2)$

Passage de paramètres à une fonction

- Toujours par copie ... mais copie de quoi ?
- Copie du contenu de la variable transmise

```
void f(int n) {
  /* n est une variable interne a f initialisee
  avec la valeur transmise a la fonction.
  Modifier n n'a aucun impact en dehors de f.
  */
  n=5;
}

int i=4;
f(i);
/* i vaut toujours 4 */
```

Passage de paramètres à une fonction

- Toujours par copie ... mais copie de quoi ?
- Copie du contenu de la variable transmise

```
void f(int *p) {
  /* p est une variable interne a f initialisee
  avec la valeur transmise a la fonction
  Modifier p n'a aucun impact en dehors de f...
  mais modifier *p a un impact...
  */
  *p=5;
}

int i=4;
f(&i);
/* i vaut maintenant 5 */
```

Passage de paramètres à une fonction

- Toujours par copie ... mais copie de quoi ?
- Copie du contenu de la variable transmise

```
void f(int t[10]) {
  /* t est un tableau, il est donc equivalent
  a un pointeur... donc modifier ce sur quoi
  il pointe a un impact hors de f
  */
  t[3]=42;
}

int tab[10]={0,1,2,3,4,5,6,7,8,9};
f(tab);
/* tab[3] vaut maintenant 42 */
```

Allouer dynamiquement la mémoire

```
#include <stdlib.h>

        void *malloc(size_t taille)
```

- Argument : `taille` en octets
- `size_t` : entier non signé
- Valeur renvoyée de type `void *`
- Déclaré dans `stdlib.h`
- ATTENTION : la zone mémoire allouée n'est pas initialisée...

Autres fonctions d'allocation : realloc

```
#include <stdlib.h>
void *realloc(void *adr, size_t taille)}
```

- Changement de taille d'un emplacement mémoire précédemment alloué
- La nouvelle taille est `taille`, elle peut être inférieure ou supérieure
- Les octets conservés de l'ancien tableau sont identiques
- L'emplacement en mémoire peut changer
- Exemple :

```
int n=10,m, *tab=malloc(n*sizeof(int)), *tab2;
... /* code initialisant m */
tab2=realloc(tab,m);
if(!tab2) {
   printf("Erreur de reallocation\n");
   exit(1);
}
tab=tab2;
...
```

Libération de la mémoire : free

```
#include <stdlib.h>
void free(void *adr);
```

- Libération de la mémoire
- Libère, d'un coup, un bloc précédemment alloué par malloc, calloc ou realloc
- Il n'est pas possible de ne libérer la mémoire que partiellement
- `free(NULL)` ne fait rien
- Libérer deux fois de suite la mémoire conduit à un comportement indéterminé : généralement un seg fault...

Copie de blocs mémoire : memcpy

```
#include <string.h>
void *memcpy(void *s1,
        const void *s2, size_t n);
```

- Copie `n` octets à partir de l'adresse contenue dans le pointeur `s2` vers l'adresse stockée dans `s1`
- Remarque : `s2` doit pointer vers une zone mémoire de taille suffisante.
- la fonction renvoie la valeur de `s1`

Structures

Vous avez dit structures ?

- Les structures sont des types composés définis par le programmeur.
- Les structures regroupent à l'intérieur d'une même variable des variables sémantiquement liées. Exemple : une fiche d'un répertoire contient le nom, le prénom, l'adresse, le numéro de téléphone...

Exemple, point dans un espace 3D :

```
struct point {
  float x;
  float y;
  float z;
};
struct point p1;
p1.x = 12.0;
p1.y = 24.2;
p1.z = 42.0;
```

Déclaration :

```
struct <nom de type>
{
   type1 var1;
   type2 var2;
   ...
   typen varn;
};
```

Utilisation :

- Déclaration d'une variable du type défini

  `struct <nom de type> <nom de variable>;`

  Exemple : `struct point p1;`

  `struct point *pp1;`

  ATTENTION, il faut bien préciser le mot-clé `struct`

- Récupération d'un champ :

  `var.<nom du champ>`

  Exemple :`p1.x=44.20;`

  `pvar-><nom du champ>`

  Exemple :`pp1->x=44.20;`

typedef

- Définition de types synonymes. Exemple :

```
typedef int entier;
entier i=3;
```

- cas des tableaux :

```
typedef int vect[3];
vect v1,v2;
```

- cas des struct

```
typedef struct _point {int x;int y;} point;
point A;
```

Initialisation & affectation

Initialisation semblable à celle de tableaux :

```
typedef struct _ma_struct {
        float x;
        char c;
} ma_struct;
ma_struct s={ 1.12, 'a'};
```

Possibilité de copier directement une structure par affectation :

```
ma_struct s1={1.12, 'a'},s2;
```

`s2=s1;` équivalent à

```
s2.x=s1.x;
s2.c=s1.c;
```

ATTENTION aux pointeurs ...

Tableaux et structures

Tableaux dans une structure

```
struct personne {
char nom[30];
char prenom[30];
unsigned int nb_commande;
unsigned int n_commande[10];
};
...
struct personne client1={"Dupond","Albert",0,
  {0,0,0,0,0,0,0,0,0,0}};
...
client1.n_commande[client1.nb_commande] = 3406;
client1.nb_commande++;
```

Tableaux de structure

```
struct personne clients[100];
...
clients[20].n_commande[clients[20].nb_commande] = 4807;
clients[20].nb_commande++;
```

Structures dans une structure

```
struct date {
  unsigned char jour;
  unsigned char mois;
  unsigned int annee;
};

struct personne {
  char nom[30];
  char prenom[30];
  struct date date_naissance;
};
...
struct personne client={"Dupond","Albert",
  {24,5,1950}};
if (client.date_naissance.annee>1940) {
  ...
}
```

Structure et pointeurs

```
typedef struct _personne {
  char * nom;
  char * prenom;
} personne;


personne p1,p2;

p1.nom = (char *)malloc(7+1);
strcpy(p1.nom,"Deckard");

p1.prenom = (char *)malloc(4+1);
strcpy(p1.prenom,"Rick");

p2=p1; // cela fait quoi ?

printf("p2.nom=%s p2.prenom=%s\n",p2.nom,p2.prenom);
free(p1.nom);
free(p1.prenom);

// Quel est le resultat de cet appel ?
printf("p2.nom=%s p2.prenom=%s\n",p2.nom,p2.prenom);
```

Listes chaînées

Listes chaînées : introduction

Principe :

- stockage d'un ensemble d'éléments
- chaque élément contient l'emplacement du suivant

Avantages par rapport aux tableaux :

- insérer un élément sans avoir avoir à recopier la suite du tableau,
- retirer un élément sans avoir avoir à recopier la suite du tableau,
- ne pas nécessiter l'utilisation de mémoire contiguë,
- permet de faire varier dynamiquement la taille,

Inconvénients :

- prend plus de place qu'un tableau
- pas d'accès immédiat à un élément

Structure de liste

```
typedef struct _un_element
  {
   int data;
   struct _un_element *suivant;
  } Un_element;
```

ou bien :

```
typedef struct _un_element *P_un_element;

typedef struct _un_element
  {
   int data;
   P_un_element suivant;
  } Un_element;
```

Création d'un élément

```
P_un_element creer_element(int v)
  {
   P_un_element el;

   el = (P_un_element) malloc(sizeof(Un_element));
   if (el == NULL) return NULL;
   el->data=v;
   el->suivant = NULL;
   return el;
  }
```

Déclaration d'une liste :

```
P_un_element   ma_liste = NULL;
```

Insertion en début de liste

```
P_un_element inserer_element_debut(P_un_element pliste,
                                   P_un_element el)
  {
   el->suivant = pliste;
   return el;
  }
```

Insertion en fin de liste

```
P_un_element inserer_element_fin(P_un_element pliste,
                                 P_un_element el)
  {
   P_un_element pl = pliste;

   if (pliste == NULL)
      return el;

   while (pl->suivant)
     {
      pl = pl->suivant;
     }
   pl->suivant = el;

   return pliste;
  }
```

Suppression d'un élément dans une liste

```
P_un_element supprimer_element(P_un_element liste,
                               P_un_element *el) {
    if ( (*el)==liste) {
        P_un_element nl=(*el)->suivant;
        free(*el);
        *el=NULL;
        return nl;
    }
    P_un_element tmp=liste;
    while ((tmp) && (tmp->suivant!=*el)) {
        tmp=tmp->suivant;
    }
    if (tmp->suivant==*el) {
        tmp->suivant = (*el)->suivant;
        free(*el);
        *el=NULL;
    }
    else {
        printf("Attention, el n'est pas dans la liste !\n");
    }
    return liste;
}
```