Rappels de C, C99, C11 et de programmation

Ed. 2019-2020, semestre 1 : Alexandre Ghiti, Florence *Maraninchi*, Grégory Mounié, Manuel Selva, Sébastien Viardot ; semestre 2 : Baptiste Jonglez, Grégory Mounié [1]

Ensimag, Grenoble-INP,
France

23 septembre 2019

1. en partie d'après les slides de Noël de Palma

Introduction

Cette présentation est :

- un rappel de ce que vous savez probablement déjà,
- des points supplémentaires, notamment [C99-11],
- quelques questions techniques annexes.

Types de base

Types d'entiers et virgules flottantes

char/**short**/**int**/**long** [**int**]/**long long** [**int**], signés par défaut, ajout de **unsigned** sinon
float/**double**/**long double** et
float /... complex [#include <complex.h>]

Divers

Le type caractère **char**, caractère long wchar_t (wchar.h) et le type booléen bool (stdbool.h)

Taille des types de base

Quelle est la taille de ces types de base sur une architecture 32 bits ? 64 bits ? Comment savoir ?

Types de base

Types d'entiers et virgules flottantes

char/**short**/**int**/**long** [**int**]/**long long** [**int**], signés par défaut, ajout de **unsigned** sinon
float/**double**/**long double** et
float /... complex [#include <complex.h>]

Divers

Le type caractère **char**, caractère long wchar_t (wchar.h) et le type booléen bool (stdbool.h)

Taille des types de base

Quelle est la taille de ces types de base sur une architecture 32 bits ? 64 bits ? Comment savoir ? **sizeof**() ou des entiers de taille fixée : int8_t, uint64_t, UINT32_MAX, INT32_MIN (stdint.h)

Tableau

Déclaration (1ère forme)

Type nomtableau [taille]

- **int** t0 [20]; 1 seule dimension indice de 0 à 19. (80 octets contiguës pré-alloués en mémoire)
  t0 [4] : accès au 5ème élément.
- **char** t1 [10][20]; 2 dimensions. Indice 0 à 9 et 0 à 19 (200 octets contiguës pré alloués en mémoire).

Initialisation à la création : **int** t0[3]= {2,1,4}; ou
int t0 [3] = { [2]= 4 }; [C99−11]

position en mémoire

Quelle est la position mémoire de l'élément t1 [2][1] par rapport au début du tableau ?

Pointeurs

Un pointeur = une adresse + un type

Types de données et mémoire

Les données en mémoire n'ont pas de types ! Les types n'existent que par les instructions qui manipulent ces données.

Un pointeur permet de définir pour le compilateur le type (et donc la taille en octet) des manipulations de la mémoire (lecture, écriture).

Pointeurs

Déclaration

```
Type *nom;
int *p0; // pointeur sur un int
void *p; // pointeur universel
```

Accès à la valeur pointée : *ptr

```
int n = 20;
int * ad_n;
ad_n = & n; // recupere l'adresse de n
*ad_n = 30; // n=30
(*ad_n)++; // n=31
*ad_n++; // !!! ad_n == ((& n) + 1)
```

Pointeurs

```
int b;
int n = 0b00010011; //[C99-11] notation binaire
int * ad_n = &n;
*(ad_n +1) = 32; // BUG ecrit b ou bien ad_n

// ad_n est un pointeur sur un int.
// ad_n+1 = @ de l'entier suivant !!!
// si ad_n  = 0x00000000 alors
// ad_n+1 = 0x00000004 (sizeof(int) == 4)
```

Tableaux et pointeurs

Un nom de tableau est un pointeur (ou presque).

```
int t[20];
```

La notation t est (globalement) équivalente à & t[0]

```
t + 1 // &t[1]
t + i // &t[i]
t[i] // *(t+i)
*(t+2) = 3; // t[2] = 3;
```

Tableaux et pointeurs

Un nom de tableau est un pointeur (ou presque).

```
int t[20];
```

La notation t est (globalement) équivalente à & t[0]

```
t + 1 // &t[1]
t + i // &t[i]
t[i] // *(t+i)
*(t+2) = 3; // t[2] = 3;
```

L'addition est commutative :-)

```
*(2+t) = 3; // t[2] = 3
2[t] = 3; // t[2] = 3
```

Tableaux alloués dynamiquement

```
int t0[20];
```

Peut être créer dynamiquement à l'exécution :

```
int *t0;
t0 = (int *) malloc(20 * sizeof(int));
```

L'accès reste identique :

```
*t0 = 3; // t0[0] = 3
*(t0+1) = 4; // t0[1] = 4
t0[2] = 5; // *(t0+2) = 5
```

Paramètres des fonctions

Passage par valeur

```
void swap(int a, int b) {
  int sv;
  sv = a; a = b; b = sv;
  printf("swap a: %d, b: %d", a,b);
}

int main(int argc, char **argv) {
  int a=5; int b = 10;

  printf("a: %d, b: %d", a,b); // a: 5,  b: 10
  swap (a,b);                  // a: 10, b: 5
  printf("a: %d, b: %d", a,b); // a: 5,  b: 10
}
```

Paramètres des fonctions

Passage par adresse == passage par valeur de l'adresse

```
void swap(int *a, int *b) {
  int sv;
  sv = *a; *a = *b; *b = sv;
  printf("swap a: %d, b: %d", *a,*b);
}

int main(int argc, char **argv) {
  int a=5; int b = 10;

  printf("a: %d, b: %d", a,b); // a: 5,  b: 10
  swap (&a,&b);                // a: 10, b: 5
  printf("a: %d, b: %d", a,b); // a: 10, b: 5
}
```

Structure

Un type point

```
struct point {
  int x;
  int y; };
struct point p0 = { 1, 2 };
typedef struct point Point;
Point p1 = { .y= 10 }; //[C99-11]
...
p0.x = 2; p0.y = 3;
p1 = p0;
```

Une liste de points

```
typedef struct liste_point {
  struct point p0;
  struct liste_point *suivant } ListePoint ;
ListePoint *p ;
...
(p->p0).x = 2; p= p->suivant;
```

Une liste de points

Ou bien

```
struct liste_point {
  int x; int y;
  struct liste_point *suivant; } ;
struct liste_point *p;
...
p->x = 2; p = p->suivant;
```

Remplissage de la mémoire

Quelle est la différence entre les deux solutions du point de vue du positionnement mémoire des éléments ?

Une liste de points

Ou bien

```
struct liste_point {
  int x; int y;
  struct liste_point *suivant; } ;
struct liste_point *p;
...
p->x = 2; p = p->suivant;
```

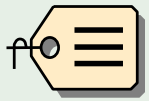
Remplissage de la mémoire

Quelle est la différence entre les deux solutions du point de vue du positionnement mémoire des éléments ? Aucune !

Parcours d'une liste chaînée

Exercice : parcours d'une liste

Parcourir et afficher les coordonnées des éléments de la liste de **struct** liste_point * Tete_de_liste .



Conversions de pointeur

```
struct liste_point *p ;
```

Arithmétique des pointeurs

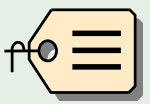
Quelle est la différence entre

- p+4
- (**int**) p + 4
- ((**int**)p) + 4

Pointeur

Exercice : pointeurs, arithmétique et liste chaînée

1. Allouer un bloc mémoire de taille 10000 octets
2. Écrire 3 **struct** liste_points dans le bloc, au début au milieu et à la fin, et les insérer dans une liste chainée.



Fichier d'en-tête

L'utilisation des **#ifndef**... permet d'éviter les définitions multiples et les boucles infinies d'inclusion.

```
#ifndef __MEM_ALLOC_H
#define __MEM_ALLOC_H
 ...
 blabla
 ...
#endif
```

Variables globales

Définitions multiples et variables globales

Attention à ne pas définir ET initialiser une variable globale dans un fichier d'entête !
Il vaut mieux définir la variable comme **extern** dans le fichier d'entête et la définir et l'initialiser dans un seul fichier .c

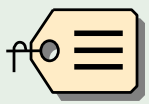
```
/// dans un fichier toto.h
extern int toto_globale;
// et surtout pas int toto_globale = 10

/// Dans un fichier toto.c
#include "toto.h"
int toto_globale= 10; // implantation
```

Opérateurs Binaires

Indispensable pour PSE

Les opérateurs classiques binaires :
ET & ;OU | ;NOT ~ ;XOR ^ ;DECALG << ;DECALD >> s'utilise plutôt sur des types entiers non signés.

Exercice : Opérations bit à bit

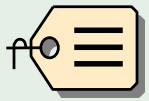
Sur un **unsigned int** : mettre le 4ème bit (en partant des poids faibles) à 1 ; le mettre à 0 ; lire sa valeur ; inverser sa valeur.
Il y a plusieurs solutions pertinentes pour chaque cas. Essayer de les trouver toutes. 
Que faut-il changer à votre code si le processeur est *little endian* ou *big endian* ?

Opérateurs Binaires

Indispensable pour PSE

Les opérateurs classiques binaires :
ET & ;OU | ;NOT ~ ;XOR ^ ;DECALG << ;DECALD >> s'utilise plutôt sur des types entiers non signés.

Exercice : Opérations bit à bit

Sur un **unsigned int** : mettre le 4ème bit (en partant des poids faibles) à 1 ; le mettre à 0 ; lire sa valeur ; inverser sa valeur.
Il y a plusieurs solutions pertinentes pour chaque cas. Essayer de les trouver toutes. 
Que faut-il changer à votre code si le processeur est *little endian* ou *big endian* ? Rien, tant que cela ne concerne que le programme

Une fonction est un pointeur comme les autres !

```
int fibo(int n) {
  return (n < 2)?1:fibo(n-1)+fibo(n-2);
}
int (*g)(int) = fibo;
```

On peut donc les passer en paramètre de fonction ou les stocker dans une structure, comme les autres

```
struct pf { int (*g)(int); } toto = { &fibo };

void h( struct pf a, int (*b)(int)) {
  a.g(10) == b(10);
}
h( toto, fibo ); ...
```

Exercice : qsort d'un tableau de complex

Faire le tri d'un tableau de float complex (réel puis imaginaire).

```
float complex tab[10];

#include <stdlib.h> // defini qsort
// void qsort(void *base, size_t nmemb,
              size_t size,
//            int (*compar)(const void *, const void *))
// compar: renvoye un entier <=> 0 si arg1 <=> arg2
```

C99, C11

Il existe plusieurs normes de `C` :

- le `C KetR`, l'original (1972)
- le `C ANSI` ou `C89` ; + corrections (celui de votre poly de *Bernard Cassagne*)
- le `C99`
- le `C11` (incomplet dans les compilateurs (ex. threads))

C99, C11

extensions

Vous utilisez déjà de nombreuses extensions de `C99` et `C11` en plus de celles déjà citées (en partie provenant de `C++`) :

- la déclaration des variables n'importe où dans un bloc,
- les commentaires mono-lignes `//`,
- le type `long long int`,
- les fonctions `inline`,
- les `variadic macro`,
- `restrict type *p` pour indiquer que `p` est le seul accès à la zone mémoire pointée (pas d'alias).

Les variables anonymes à la Java

compounds litterals

```
struct point { int x; int y; };

int f(struct point p) {...};
int g(struct point *p) {...};

...
f( (struct point){ 1, 1 } );
g( & (struct point){ 2, 2 } );
...
```

designated initializers

L'initialisation partielle "random" d'un tableau ou d'une structure.

```
struct point { int x; int y; };

struct point p = { .y= 10 };

int a[10] = { [4]= 10 };

float identite[4][4]= {[0][0]= 1.0, [1][1]= 1.0,
                       [2][2]= 1.0, [3][3]= 1.0};
```

Les caractères accentuées

```
#include <stdio.h>
#include <wchar.h>
#include <locale.h>

int main() {
    wchar_t string[100];
    setlocale(LC_ALL, ""); // user locale
    printf(u8"éàî\n""); // [C11]
    scanf("%ls", string);
    printf("%ls est une chaîne de longueur %d\n",
            string, wcslen(string));
}
```

Compilation séparée : le makefile

```
# Options generales de compilation
# Les warning et les debogueurs sont vos amis
CFLAGS = -Wall -g  -Werror -Wextras

# Pour compiler le shell de test de l'allocateur
memshell: alloc.o memshell.o
	$(CC) $(CFLAGS) -o memshell alloc.o memshell.o

# Pour creer le fichier objet de l'allocateur
alloc.o: alloc.c alloc.h
	$(CC) $(CFLAGS) -c alloc.c
```

Compilation séparée : cmake

Autoconf/automake, cmake, scons, ... génèrent automatiquement un makefile (ou des projets XCode ou Visual Studio) à partir d'une description succincte des besoins. Les TPs utilisent `cmake`.
Un fichier `CMakeList.txt` minimaliste

```
projet(Memshell)
set(CMAKE_BUILD_TYPE Debug)
add_executable(memshell alloc.c memshell.c)
```

Compiler dans un sous-répertoire permet de nettoyer plus facilement

```
> ls
CMakeList.txt build/
> cd build
> cmake .. # cree le makefile
> make
```

Valgrind

Valgrind est un outil de débogage (et plus) qui est capable de vérifier la pertinence des accès mémoires et surtout de donner des indications sur les erreurs :

- débordement de tableaux,
- variable non initialisée,
- réutilisation d'un pointeur déjà libéré...

Il fonctionne en instrumentant le code à l'exécution pour tracer les accès mémoires.
Mais il ne fait pas des miracles : les accès mémoire valides sont valides ! (eg. débordement de mémoire allouée statiquement)

Valgrind

Exemple de code faux

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

int main(int argc, char **argv) {
  char tampon[256]; char *copie;
  scanf("%255s", tampon);
  copie = malloc(strnlen(tampon, 256));
  strncpy(copie, tampon, 256); return 0; }
```

Tampon de taille limitée en mémoire locale

Pourquoi faut-il faire particulièrement attention à la taille des entrées (`%255s`) ? (Indication : que font `call toto` et `ret` en assembleur x86 ?)

Valgrind

Messages d'erreurs

```
==20137== Memcheck, a memory error detector
==20137== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Sewa
==20137== Using Valgrind-3.5.0-Debian and LibVEX; rerun with -h
==20137== Command: ./a.out
==20137== Invalid write of size 1
==20137==    at 0x4025DB0: strncpy (mc_replace_strmem.c:329)
==20137==    by 0x80484DE: main (exemple_valgrind.c:10)
==20137==  Address 0x41a202c is 0 bytes after a block of size 4
==20137==    at 0x4024C4C: malloc (vg_replace_malloc.c:195)
==20137==    by 0x80484B8: main (exemple_valgrind.c:9)
==20137==
==20137== Invalid write of size 1
==20137==    at 0x4025DBD: strncpy (mc_replace_strmem.c:329)
==20137==    by 0x80484DE: main (exemple_valgrind.c:10)
==20137==  Address 0x41a202e is 2 bytes after a block of size 4
==20137==    at 0x4024C4C: malloc (vg_replace_malloc.c:195)
==20137==    by 0x80484B8: main (exemple_valgrind.c:9)
```

Valgrind

Messages d'erreurs

```
==20137==
==20137==
==20137== HEAP SUMMARY:
==20137==     in use at exit: 4 bytes in 1 blocks
==20137==   total heap usage: 1 allocs, 0 frees, 4 bytes allocat
==20137==
==20137== LEAK SUMMARY:
==20137==    definitely lost: 4 bytes in 1 blocks
==20137==    indirectly lost: 0 bytes in 0 blocks
==20137==      possibly lost: 0 bytes in 0 blocks
==20137==    still reachable: 0 bytes in 0 blocks
==20137==         suppressed: 0 bytes in 0 blocks
==20137== Rerun with --leak-check=full to see details of leaked
==20137==
==20137== For counts of detected and suppressed errors, rerun wi
==20137== ERROR SUMMARY: 252 errors from 2 contexts (suppressed:
```

Assertion

C fournit en standard une gestion des assertions. On peut aussi utiliser des macros, avec quelques précautions.

```
#include <assert.h>

#define handle_error(msg) \
      do { perror(msg); exit(-1); } while (0)

int f(int *a)
{
   assert( a != NULL); // a DOIT etre non null
   if (un_appel_systeme(...)) // 0: OK,
                              // != 0: probleme
      handle_error("appel rate. La raison est:");
}
```