Le langage C

Rappels, C99-C11-C17-C23 et programmation

Semestre 1 : Louis Boulanger, Florence Maraninchi, Grégory Mounié, Alain Tchana, Frédéric Wagner. Semestre 2 : Grégory Mounié, Alain Tchana [1]

2022-2023

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

Introduction

Cette présentation est un rappel de ce que vous savez probablement déjà ; quelques points supplémentaires moins connus, notamment [C99-C11-C17/C18-C23] ; quelques questions techniques annexes.

1. Rappels de C
   - Les types de base
   - Les tableaux
   - Les pointeurs
   - Macro processeur
2. C avancé
   - Pointeurs de fonctions
   - C99, C11, C17, C23
3. Programmation
   - compilation séparée
   - Valgrind
   - Assertions

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/double/long double complex`
[`#include <complex.h>`]

Divers

Le type caractère `char` , caractère long `wchar_t` [`#include <wchar.h>`] et le type booléen `bool` [`#include <stdbool.h>`]

Modificateurs

- `static`, variables globales non exportés, variables locales de durée de vie égale au programme (comme les variables globales), paramètre de l'index des tableaux en argument indiquant la taille minimum ńgarantież,
- `const`, suffix, variables non modifiables après l'initialisation
- `restrict`, il n'existe pas d'autres pointeurs sur la même zone
- `thread_local` variable globale dupliquée, une par thread, au lieu d'être unique et partagée

Types de base

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

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 fixer la taille

L'opérateur `sizeof(nom_de_type_ou_nom_de_variable)` permet d'obtenir la taille d'un type en octet.
Il est aussi possible d'utiliser des types entiers de taille fixée : `int8_t`, `uint64_t`, `UINT32_MAX`, `INT32_MIN` [`#include 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 (10 blocs of 20 char). Indice 0 à 9 et 0 à 19 (200 octets contigus pré alloués en mémoire).

Initialisation à la création : `int t0[3]= {2,1,4};` ou `int t0[3] = { [2]= 4 };`

position en mémoire

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

Pointeurs : addresses typées

Un pointeur = une adresse + un type

Types de données et mémoire

La mémoire (RAM) est un tableau qui stocke des bits (0/1) rangés par octet (8 bits), indicé par un entier (l'adresse de l'octet dans la mémoire).
Les données en mémoire n'ont pas de types explicites !. Elles sont juste des suites d'octets. Les types n'existent que dans les instructions qui manipulent ces données.

Un pointeur permet de d'expliquer au compilateur le type (et donc sa taille, en octet) d'une suite d'octet et donc les manipulations à faire pour ce type (lecture, écriture, opérations).

Pointeurs

Déclaration

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

Accès à la valeur pointée : `*ptr`

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

Pointeurs

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

```
1  int t[20];
2  t + 1 // &t[1]
3  t + i // &t[i]
4  t[i] // *(t+i)
5  *(t+2) = 3; // t[2] = 3;
```

Tableaux et pointeurs

Un nom de tableau est un pointeur (ou presque). La notation `t` est (globalement) équivalente à `& t[0]`

```
1  int t[20];
2  t + 1 // &t[1]
3  t + i // &t[i]
4  t[i] // *(t+i)
5  *(t+2) = 3; // t[2] = 3;
```

L'addition est commutative (Oui, cela compile ! Sans warning !)

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

Tableaux alloués dynamiquement

Un tableau peut être créer dynamiquement à l'exécution en demandant un bloc de mémoire de la bonne taille. L'accès reste identique.

```
1 // l'équivalent dynamique de:  int t0[20];
2 int *t0;
3 t0 = malloc( sizeof( int[20] ) ); // or 20 * sizeof(int)
4
5 *t0 = 3; // t0[0] = 3
6 *(t0+1) = 4; // t0[1] = 4
7 t0[2] = 5; // *(t0+2) = 5
```

Paramètres des fonctions

Toujours un passage par valeur (l'argument est une autre variable, copie de celle passée en paramètre)

```
1  void swap(int a, int b) {
2   int sv;
3   sv = a; a = b; b = sv;
4   printf("swap a: %d, b: %d", a,b);
5  }
6  int main(int argc, char **argv) {
7   int a=5; int b = 10;
8
9   printf("a: %d, b: %d", a,b); // a: 5,  b: 10
10  swap (a,b);              // dans swap: a: 10, b: 5 (des copies !)
11  printf("a: %d, b: %d", a,b); // a: 5,  b: 10
12 }
```

Paramètres des fonctions

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

```c
void swap(int *a, int *b) {
  int sv;
  sv = *a; *a = *b; *b = sv;
  printf("swap a: %d, b: %d\n", *a,*b);
}
int main(int argc, char **argv) {
  int a=5; int b = 10;

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

Paramètres des fonctions

Aider l'analyseur du compilateur à trouver des bugs

```c
void swap(int a[static 1], int b[static 1]) { // a != NULL, b != NULL
  int sv;
  sv = *a; *a = *b; *b = sv;
  printf("swap a: %d, b: %d\n", *a,*b);
}
int main(int argc, char *argv[argc +1]) { // argv != NULL
  int a=5; int b = 10;

  printf("a: %d, b: %d\n", a,b); // a: 5,  b: 10
  swap (&a,&b);            // a[0]: 10, b[0]: 5
  printf("a: %d, b: %d\n", a,b); // a: 10, b: 5
}
```

Structure

Un type point

```
1  struct point {
2    int x;
3    int y; };
4  struct point p0 = { 1, 2 };
5  typedef struct point Point;
6  Point p1 = { .y= 10 }; //[C99-11]
7  // ...
8  p0.x = 2; p0.y = 3;
9  p1 = p0;
```

enum

Une énumération de constantes entières.

```
1 enum VAL { VAL0= 0, VAL3=3, VAL4, VAL5 };
2 // ...
3 enum VAL v = VAL4;
```

Union

Plusieurs valeurs de types différents, rangés dans les mêmes octets. Par exemple

```c
union Data { // renvoie la solution ou bien un code d'erreur
  int error_code; // 4 octets
  long double value; // 16 octets
}; // taille total: max(4, 16) == 16 octets
struct Result {
  bool ok;
  union Data value;
} r;
...
r = compute_first(); // note: compute renvoie la copie d'une struct
if (! r.ok)
  printf("error %d\n", r.error_code);
else
  compute_next(r.value);
```

Une liste de points

```
1 typedef struct liste_point {
2   struct point p0;
3   struct liste_point *suivant } ListePoint ;
4 ListePoint *p ;
5 // ...
6 (p->p0).x = 2; p= p->suivant;
```

Une liste de points

Ou bien

```
1 struct liste_point {
2   int x; int y;
3   struct liste_point *suivant; } ;
4 struct liste_point *p;
5 // ...
6 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

```
1 struct liste_point {
2   int x; int y;
3   struct liste_point *suivant; } ;
4 struct liste_point *p;
5 // ...
6 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`.

Solution : Parcours d'une liste chainée

```c
#include <stdio.h>
struct liste_point {
 int x;
 int y;
 struct liste_point *suivant;
} a[4] = {{.x = 1}, {.y = 2}, {.x = 3}, {.y = 4}};
struct liste_point *Tete_de_liste;

void affiche() {
 struct liste_point *p = Tete_de_liste;
 while (p != NULL) {
  printf("%d %d;", p->x, p->y);
  p = p->suivant;
 }
 printf("\n");
}
```

Solution : Parcours d'une liste chainée

```
18 int main() {
19   Tete_de_liste = &a[0];
20   for (int i = 0; i < 3; i++) {
21     a[i].suivant = &a[i + 1];
22   }
23   affiche();
24 }
```

Conversions de pointeur

```
1  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, les insérer dans une liste chainée, afficher la liste chainée, libérer le bloc complet.

Solution : pointeurs, arithmétique et liste chaînée

```c
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
struct liste_point {
 int x;
 int y;
 struct liste_point *suivant;
};
struct liste_point *tete;
void affiche() {
 struct liste_point *p = tete;
 while (p != NULL) {
  printf("%d %d;", p->x, p->y);
  p = p->suivant;
 }
 printf("\n");
}
```

Solution : pointeurs, arithmétique et liste chaînée

```
19 int main() {
20   void *p = malloc(10000);
21   assert(p);
22   struct liste_point *debut, *milieu, *fin;
23
24   debut = (struct liste_point *)p;
25   *debut = (struct liste_point){.x = 1};
26
27   milieu = (struct liste_point *)(((unsigned long)p) + 5000);
28   *milieu = (struct liste_point){};
29   milieu->y = 2;
```

Solution : pointeurs, arithmétique et liste chaînée

```
31    fin = (struct liste_point *)(((unsigned long)p) + 10000 -
32                        sizeof(struct liste_point));
33    *fin = (struct liste_point){};
34    fin->x = 3;
35
36    tete = debut;
37    debut->suivant = milieu;
38    milieu->suivant = fin;
39    fin->suivant = NULL;
40
41    affiche();
42    free(p);
43  }
```

Const

Modificateur **suffixé** (détail important pour les pointeurs) mais qui peut être mis en premier pour les types simples.

```
1  const int a=10; // en vrai: int const a=10;
2  int b= 10;
3  int *const pb = &b;
4  *pb = 12; // b == 12
```

Constantes

Le pré-processeur peut remplacer une chaine de caractère, c'est souvent utiliser pour définir des constantes.

```
1  // Exemples tirés de math.h
2  # define M_E            2.7182818284590452354   /* e */
3  # define M_LOG2E        1.4426950408889634074   /* log_2 e */
4  # define M_LOG10E       0.43429448190325182765  /* log_10 e */
5  # define M_LN2          0.69314718055994530942  /* log_e 2 */
6  # define M_LN10         2.30258509299404568402  /* log_e 10 */
7  # define M_PI           3.14159265358979323846  /* pi */
8  # define M_PI_2         1.57079632679489661923  /* pi/2 */
9  # define M_PI_4         0.78539816339744830962  /* pi/4 */
10 # define M_1_PI         0.31830988618379067154  /* 1/pi */
11 # define M_2_PI         0.63661977236758134308  /* 2/pi */
```

Fichier den-tête

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

```
1 #ifndef __MEM_ALLOC_H
2 #define __MEM_ALLOC_H
3 // ...
4 // blabla
5 // ...
6 #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`

```
1  /// dans un fichier toto.h
2  extern int toto_globale;
3  // et surtout pas int toto_globale = 10
4
5  /// Dans un fichier toto.c
6  #include "toto.h"
7  int toto_globale= 10; // implantation
```

Ne sous-estimez pas le pré-processeur !

Paquetage générique de liste

```c
#include "utlist.h"
typedef struct lp {
 int x;
 struct lp *next; } LP; // NB: link has 'next' name
LP *head=NULL;
// ....
LP *elem = malloc(sizeof(LP));
*elem = (LP){42, NULL}; // copie !
LL_APPEND(head, elem);
LL_FOREACH(head, elem) {
 printf("%d\n", elem->x);
}
```

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

Solution : Opérations bit à bit

```c
#include <stdio.h>

int main() {
  unsigned int a = 0;

  printf("%d %d\n", a, ((a & 0b1000) >> 3));
  a |= (1 << 3);
  printf("%d %d\n", a, ((a & 0b1000) >> 3));
  a &= ~(1 << 3);
  printf("%d %d\n", a, ((a & 0b1000) >> 3));
  a ^= (1 << 3);
  printf("%d %d\n", a, ((a & 0b1000) >> 3));
}
```

Une fonction est un pointeur comme les autres !

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

```
1  int fibo(int n) {
2   return (n < 2)?n:fibo(n-1)+fibo(n-2);
3  }
4  int (*g)(int) = fibo;
5  struct pf { int (*g)(int); } toto = { &fibo };
6
7  void h( struct pf a, int (*b)(int)) {
8   a.g(10) == b(10);
9  }
10 h( toto, fibo );
```

Exercice : qsort d'un tableau de complex

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

```
1  float complex tab[1000];
2
3  #include <stdlib.h> // defini qsort
4  // void qsort(void *base, size_t nmemb,
5        size_t size,
6  //        int (*compar)(const void *, const void *));
7  // compar: renvoie un entier <=> 0 si arg1 <=> arg2
```

C99, C11, C17

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`, tableau variable, restrict, complex, `//`
- le `C11`, unicode, threads, `_Generic` (used in tgmath.h), `_Static_assert`
- le `C17`, (`C18`), static dans les arguments tableaux, correctifs et clarifications
- le futur `C23` (to appear), `nullptr`, `constexpr`, `typeof`, paramètres de fonctions non nommés (car inutilisé), tableaux constants, attributs, flottants à exposants décimaux ; etc.

C99, C11, C17

extensions

Vous utilisez déjà naturellement de nombreuses extensions de `C99`, `C11` et `C17` (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. Attention à la portée !

```
1  struct point { int x; int y; };
2
3  int f(struct point p) {...};
4  int g(struct point *p) {...};
5
6  // ...
7  f( (struct point){ 1, 1 } );
8  g( & (struct point){ 2, 2 } );
9  // ...
```

designated initializers

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

```
1  struct point { int x; int y; };
2
3  struct point p = { .y= 10 };
4
5  int a[10] = { [4]= 10 };
6
7  float identite[4][4]= {[0][0]= 1.0, [1][1]= 1.0,
8                         [2][2]= 1.0, [3][3]= 1.0};
```

Inline functions

De petites fonctions peuvent être *inlinée. Le code est écrit dans un fichier d'entête, contrairement aux autres. Mais elle doit aussi être instanciée dans un unique .c*

```
1  // dans MYADD.h
2  inline int add1(int a) { return a + 1; }
3
4  // dans SINGLE .c
5  #include <MYADD.h>
6  int add1(int);
```

Les caractères accentués

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

int main() {
  wchar_t string[100]={};
  setlocale(LC_ALL, ""); // user locale
  printf(u8"éàî\n"); // [C11]
  scanf("%99ls", string);
  printf("%ls est une chaîne de %ld glyphes de taille %ld octets\n",
       string, wcsnlen(string, 100), wcstombs(NULL, string, 100));
}
```

Types génériques

C fournit en standard un mécanisme de généricité. Il est utilisé par exemple dans `tgmath.h`. En fonction du type de l'argument, c'est la bonne fonction mathématique qui est appelée.

```
1  #include <tgmath.h>
2  #include <stdio.h>
3
4  int main()
5  {
6   float f= 1;
7   long double complex fc = -1.0 - I;
8   printf("%e\n", sqrt(f));
9   printf("%Lf + %Lf i = %Lf e^{i \%Lf}\n",
10       creal(sqrt(fc)), cimag(sqrt(fc)),
11       fabs(sqrt(fc)), carg(sqrt(fc)));
12 }
```

Types génériques

Le principe est d'avoir l'équivalent d'un `switch`, mais sur les types. Il est utilisé en combinaison avec les macros.

```
1  #define sqrt(X) _Generic((X), \ //switch sur le type de (X)
2  long double: cbrtl, \
3  default: cbrt, \
4  float: cbrtf, \
5  long double complex: csqrtl, \
6  double complex: csqrt, \
7  float complex: csqrtf, \
8  )(X)
```

Compilation séparée : le makefile

```
1  # Options generales de compilation
2  # Les warning et les debogueurs sont vos amis
3  CFLAGS = -Wall -g  -Werror -Wextras
4
5  # Pour compiler le shell de test de l'allocateur
6  memshell: alloc.o memshell.o
7       $(CC) $(CFLAGS) -o memshell alloc.o memshell.o
8
9  # Pour creer le fichier objet de l'allocateur
10 alloc.o: alloc.c alloc.h
11      $(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

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

Compiler dans un sous-répertoire permet de nettoyer plus facilement

```
1     $ ls
2     CMakeList.txt build/
3     $ cd build
4     $ cmake .. # cree le makefile
5     $ 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

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4
5  int main(int argc, char **argv) {
6   char tampon[256]={}; char *copie;
7   scanf("%255s", tampon); // do not overfill tampon
8   copie = calloc(strnlen(tampon, 256),sizeof(char)); // malloc + init à 0
9   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` et `ret` en assembleur x86 ?)

Valgrind

Messages d'erreurs

```
1  ==20137== Memcheck, a memory error detector
2  ==20137== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
3  ==20137== Using Valgrind-3.5.0-Debian and LibVEX; rerun with -h for copyright info
4  ==20137== Command: ./a.out
5  ==20137== Invalid write of size 1
6  ==20137==    at 0x4025DB0: strncpy (mc_replace_strmem.c:329)
7  ==20137==    by 0x80484DE: main (exemple_valgrind.c:10)
8  ==20137==  Address 0x41a202c is 0 bytes after a block of size 4 alloc'd
9  ==20137==    at 0x4024C4C: malloc (vg_replace_malloc.c:195)
10 ==20137==    by 0x80484B8: main (exemple_valgrind.c:9)
11 ==20137== Invalid write of size 1
12 ==20137==    at 0x4025DBD: strncpy (mc_replace_strmem.c:329)
13 ==20137==    by 0x80484DE: main (exemple_valgrind.c:10)
14 ==20137==  Address 0x41a202e is 2 bytes after a block of size 4 alloc'd
15 ==20137==    at 0x4024C4C: malloc (vg_replace_malloc.c:195)
16 ==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 allocated
==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 memory
==20137==
==20137== For counts of detected and suppressed errors, rerun with: -v
==20137== ERROR SUMMARY: 252 errors from 2 contexts (suppressed: 11 from 6)
```

Assertion dynamique

C fournit en des assertions vérifiées à l'exécution. Utiliser aussi les macros !

```c
#include <assert.h>
#define handle_error(msg) \
    do { perror(msg); exit(-1); } while (0)
int f(int *a) {
 assert( a != NULL); // a MUST NOT be NULL (eg. 0)

 //
 if ((fd= open(...)) < 0) // 0+: OK, < 0: problem
    handle_error(u8"open failed. Reason:");

 // assert_perror is a GNU Only macro of assert.h
 fd = open(...); // any syscall fills errno thread_local variable
 assert_perror(errno);
}
```

Assertion statique

C fournit aussi des assertions pouvant être vérifiées à la compilation.

```c
#include <assert.h>

int f(int *a)
{
  // macro defined in assert.h
  static_assert(sizeof(int) == 4, "32 bits integer required");

  // in Gnu libc, macro identical to call directly the C keyword
  _Static_assert(sizeof(int) == 4, "32 bits integer required");
}
```

Analyseur statique

Votre compilateur (gcc-10+ ou clang) peut passer plus de temps sur la compilation de votre code afin de détecter des manques ou des bugs.

```
$ gcc -fanalyzer -Wall -Wextra elempool.c
... # ou en utilisant clang
$ scan-build-15 clang-15 -c -Wall -Wextra elempool.c
$ scan-view-15 /tmp/scan-build-2022-08-29-183802-19055-1
```