

# L3 MI — Programmation

Pierre Karpman

`pierre.karpman@univ-grenoble-alpes.fr`  
`https://membres-ljk.imag.fr/Pierre.Karpman/tea.html`

2023-09-13

## Éléments de langage (2)

Retour sur les types arithmétiques

Instructions bit (1)

Quelques types spéciaux (1)

## Compilation (2) : au sens très large

Le CPP

Débuggage (1)

Fichiers multiples (2) : visibilité

Bibliothèques

## Génération d'aléa (en C)

## Éléments de langage (2)

Retour sur les types arithmétiques

Instructions bit (1)

Quelques types spéciaux (1)

## Compilation (2) : au sens très large

Le CPP

Débuggage (1)

Fichiers multiples (2) : visibilité

Bibliothèques

## Génération d'aléa (en C)

# Rappel fixe, flottant (en fil d'araignée) ?

---

Un nombre non entier peut être représenté en machine en...

- ▶ Virgule fixe : la partie entière (resp. fractionnaire) d'un nombre est encodée sur un nombre fixe de bits
  - ▶ Conceptuellement simple
  - ▶ Mais inefficace : on ne peut pas avec un même format représenter des nombres de très grande et très petite magnitude
- ▶ Virgule flottante : utilisation d'une *mantisse*  $x \in \llbracket 0, s \rrbracket$ , d'un *exposant*  $e \in \llbracket -a, b \rrbracket$  et d'un signe pour représenter les nombres de la forme  $\pm 2^e \cdot m$ 
  - ▶ Flexible
  - ▶ Plus complexe ?

## Flottants IEEE 754

- ▶ Les types flottants en C (`float`, `double...`) suivent la norme IEEE 754 pour l'arithmétique flottante
- ▶  $\rightsquigarrow$  spécifie le format de représentation des nombres, des valeurs spéciales, des modes d'arrondi, etc.

Taille des champs :

- ▶ `float` : mantisse : 23 bits (+ 1 gratuit), exposant 8 bits ( $\in \llbracket -126, 127 \rrbracket$ )
- ▶ `double` : mantisse : 52 bits (+ 1 gratuit), exposant 11 bits ( $\in \llbracket -1022, 1023 \rrbracket$ )

Certaines valeurs de l'exposant sont réservées pour des usages spéciaux :

- ▶ Tout à 1 : représentation d'infini, NaN ("not a number")
- ▶ Tout à 0 : représentation *des* zéros ; de nombres « dénormalisés » (très petits, avec des zéros de tête dans la mantisse non nulle)

## Flottants (cont.)

---

Quelques conséquences du format :

- ▶ Les nombres *entiers* (relatifs) suffisamment petits (en valeur absolue) sont *toujours* représentés exactement
  - ▶ On peut parfois utiliser des flottants pour implémenter plus efficacement des opérations sur les entiers
- ▶ Les autres le sont peut-être, ou peut-être pas...
- ▶ Cela n'a pas de sens (par ex. dans un schéma d'approximation numérique) de chercher une précision supérieure à celle du format (par ex. approcher à  $2^{-100}$  pour un `double`)
- ▶ Attention aux tests d'égalité et leur interprétation, d'autant plus qu'il y a plusieurs zéros (`0.0`, `-0.0`) → ne pas tester l'égalité des flottants

Si besoin, des bibliothèques comme MPFR permettent de faire des calculs numériques en précision arbitraire

# Conversion de types

---

Il y a au moins trois types de conversion de types arithmétiques rencontrés couramment en C :

- ▶ Entier  $\leftrightarrow$  flottant
- ▶ Changement de précision, par ex. `uint64_t` vers `uint32_t`
- ▶ Entier signé  $\leftrightarrow$  non-signé

Toutes peuvent se faire implicitement, et toutes peuvent engendrer une perte de précision ou des erreurs

## Entier ↔ flottant

---

Quelques exemples :

- ▶ `double x = 1337 // tout va bien`
- ▶ `double x = 0x123456789ABCDEF0 // perte de précision`
- ▶ `double x = 12/5 // division entière`
- ▶ `int x = 14. // tout va bien`
- ▶ `int x = (int)14. // pareil, explicitement`
- ▶ `int x = 14./5. // troncation`
- ▶ `int x = 123456789123. // overflow/erreur`

# Changement de précision, signes

---

Quelques exemples :

- ▶ `uint32_t x = 2;`  
`uint8_t y = x; // tout va bien`
- ▶ `uint32_t x = 257;`  
`uint8_t y = x; // réduction modulo 256`
- ▶ `int32_t x = 128;`  
`int8_t y = x; // non défini`
- ▶ `int32_t x = 128;`  
`uint8_t y = x; // tout va bien`
- ▶ `int32_t x = -1;`  
`uint32_t y = x; // renormalisation ; dépend de`  
↪ *l'architecture (jusqu'à C23)*

## Quelques conseils

---

- ▶ Utilisez des types homogènes (taille, signe) pour éviter les conversions (implicites)
- ▶ Si une conversion spécifique est nécessaire faites la explicitement, par ex.

```
int8_t x = (int8_t)(y % 128); // y de type uint8_t
```

- ▶ Faites attention aux types des constantes numériques, par ex.
  - ▶ 1 est un entier signé « standard » (`int`)
  - ▶ 0x1 est un entier non signé « standard » (`unsigned`)
  - ▶ 1ULL est un long long entier non signé (`unsigned long long`)

## Une curiosité

---

Exemple fourni par @rep\_stosq\_void (valeurs d'affichage données pour un compilateur et une architecture 64 bits classique) :

```
#include <stdio.h>
int main() {
    printf("%d\n", 0 > -1); // 1
    printf("%d\n", 0U > -1); // 0
    printf("%d\n", 0U > -1L); // 1
    printf("%d\n", 0UL > -1L); // 0
}
```

Dans le second cas, un `int` ne peut pas représenter toutes les valeurs d'un `unsigned` : `-1` est converti en un `unsigned`; dans le troisième cas, un `long` peut représenter toutes les valeurs d'un `unsigned` : `0U` est converti en un `long`; dans le dernier cas un `long` ne peut représenter toutes les valeurs d'un `unsigned long` : `-1L` est converti en un `unsigned long`

Pour n'importe quel type de donnée, il est capital de distinguer une représentation :

- ▶ logique (par ex. un entier manipulé en base 10 (`int x = 3;`, `printf("%d\n", 3);`, `scanf("%d", &x);`))
- ▶ physique (par ex. un champ de 32 bits regroupés en 4 octets adressés en *little endian*)

## (Absence de) conversion

---

Une « mauvaise pratique » classique :

- ▶ Avoir besoin de travailler sur la représentation physique (binaire) d'un entier
- ▶ Le convertir en un tableau de (par ex.) 32 entiers dans  $\{0, 1\}$
- ▶ Travailler sur cette « représentation binaire »
- ▶ Reconvertir le résultat en un entier « décimal »

↪ Ces conversions sont en général inutiles :

- ▶ Le processeur travaille déjà avec la « représentation binaire »
- ▶ ... et fournit des instructions permettant de la manipuler directement

↪ Un bon programme utilise autant que possible l'arithmétique implémentée par les processeurs, et présente éventuellement une vision logique différente (par ex. affichage d'un nombre en binaire (relativement inutile))

# Exemple

---

Supposons qu'on souhaite savoir s'il y a une propagation de retenue dans l'addition binaire de deux variables `a`, `b` de type `unsigned`

# Exemple

---

Solution (excessivement) naïve : « convertir » a et b en binaire, puis implémenter et tester l'addition bit à bit

## Exemple

---

Mieux :

- ▶ Il y a une propagation ssi a et b valent tous les deux 1 sur un même bit (N.B. Dans le cas du bit de poids fort, une propagation serait « absorbée » par la réduction modulo  $2^{32}$ , et serait donc « silencieuse »)
- ▶ Donc absence de propagation si leurs écritures binaires sont de « supports » disjoints
- ▶ Pour un test sur un seul bit, la condition  $(x, y) \neq (1, 1)$  peut par exemple s'exprimer simplement comme  $x \text{ AND } y = 0$
- ▶ En C, ceci peut se calculer facilement et simultanément sur tous les bits d'un `unsigned` via l'expression `a & b == 0`
- ▶ (Presque alternative : tester l'égalité entre addition et XOR bit à bit : `a + b == a ^ b`)

# Endianness

---

L'endianness (“big” ou “little”) définit la façon dont une suite d'octets est interprétée comme un entier ; c'est une caractéristique du processeur

- ▶ Big endian :  $\{a, b\}$  (où  $a$  est le premier élément, par ex. se trouvant à l'adresse la plus petite (cf. prochain cours sur les pointeurs)) est interprété comme  $a \times 2^8 + b$
- ▶ Little endian :  $\{a, b\}$  est interprété comme  $b \times 2^8 + a$

La plupart des processeurs grand-public actuels sont en little endian

## Endianness (cont.)

---

Exemple (cf. un cours prochain pour plus de détails sur les pointeurs) :

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

int main() {
    uint8_t a[4] = {0x12, 0x34, 0x56, 0x78};
    printf("%X\n", *a); // 12
    printf("%X\n", *((uint16_t*)a)); // 3412 (sur un
    ↪ CPU LE)
    printf("%X\n", *((uint32_t*)a)); // 78563412
    ↪ (ditto)
    return 0;
}
```

L'endianness est importante quand on doit interpréter des données non structurées (e.g. fichier binaire) comme une suite d'entier, par ex. :

- ▶ transfert réseau (convention réseau : big endian)
- ▶ lecture/écriture (par ex. d'un tableau de nombres) depuis/vers un fichier
- ▶ fonctions prenant en entrée des données « brutes » (par ex. fonctions de hachage (cryptographiques ou non))

Mais en général, les entiers sont manipulés à suffisamment haut niveau pour que l'endianness n'importe pas

## Éléments de langage (2)

Retour sur les types arithmétiques

Instructions bit (1)

Quelques types spéciaux (1)

## Compilation (2) : au sens très large

Le CPP

Débuggage (1)

Fichiers multiples (2) : visibilité

Bibliothèques

## Génération d'aléa (en C)

## Premier exemple

---

Le langage C définit des opérateurs permettant de manipuler individuellement les bits d'entiers (généralement non signés)

- ▶ L'opérateur `^` calcule le XOR bit-à-bit

Exemple :

```
uint8_t a = 0x55;  
uint8_t b = 0xAA;  
uint8_t c = a ^ b; // 0xFF
```

Si l'on interprète  $a$  et  $b$  comme des vecteurs  $\mathbf{a}, \mathbf{b} \in \mathbb{F}_2^8$ ,  $a \wedge b$  calcule  $\mathbf{a} + \mathbf{b}$

*Remarque* : Il est pratique (et usuel) d'écrire les opérandes avec une base compatible avec une action bit-à-bit, c-à-d la base 16

On dispose aussi des opérateurs suivants

- ▶ L'opérateur  $\&$  ; calcule le AND bit-à-bit (le produit terme à terme de vecteurs de  $\mathbb{F}_2^n$ )
- ▶ L'opérateur  $|$  ; calcule le OR bit-à-bit
- ▶ L'opérateur unaire  $\sim$  ; calcule le NOT bit-à-bit
  - ▶ Exercice : comment peut-on calculer  $\sim x$  grâce à  $\wedge$  ?
  - ▶ comment peut-on calculer  $a | b$  grâce à  $\sim$  et  $\&$  ?
  - ▶ comment peut-on calculer  $a \wedge b$  grâce à  $\sim$  et  $\&$  ?
  - ▶ proposition : l'opérateur NAND est universel

## Exercice : inclusion d'ensembles

---

Soit un ensemble  $\mathcal{S}$  d'au plus  $n$  éléments  $x_i$ ,  $i \in \llbracket 0, n - 1 \rrbracket$ . On représente un sous-ensemble  $\mathcal{S}'$  de  $\mathcal{S}$  par l'entier  $s' := \sum_{i=0}^{n-1} [x_i \in \mathcal{S}'] \times 2^i$  (où  $[P]$  est un entier qui vaut 1 si le prédicat  $P$  est vrai, et 0 sinon)

Supposons que  $n \leq 64$  ; comment peut-on calculer efficacement l'inclusion de `a` dans `b`, où chaque ensemble est représenté par un `uint64_t` ?

## Opérateurs de décalage

---

Il est également possible de décaler les bits d'un unique entier :

- ▶ L'opérateur `<<` agit sur un entier de  $n$  bits (à gauche) et un entier entre 0 et  $n$  (exclu) à droite
- ▶ Le résultat de `a << b` est égal à `a` dont les bits sont décalés de `b` positions vers la « gauche » (c-à-d vers les bits de poids fort), et autant de zéros sont introduits à droite

▶ `1 << 1; // 2`

`1 << 2; // 4`

`uint64_t a = 1 << 40; // comportement non défini`  
↳ (UB), car '1' est un littéral sur 32 bits...

`uint64_t a = 1ULL << 40; // 0x10000000000`

`1ULL << 64; // UB`

`0x5 << 1; // 0xA`

`0xF << 1; // 0x1E`

## Opérateurs de décalage (cont.)

---

L'opérateur `>>` fonctionne de façon analogue, avec deux variantes :

- ▶ Décalage *logique* ; le résultat de `a >> b` est égal à `a` dont les bits sont décalés de `b` positions vers la « droite » (c-à-d vers les bits de poids faible), et autant de zéros sont introduits à gauche
- ▶ Décalage *arithmétique* pour entiers signés ; le résultat de `a >> b` est [...], et autant de zéros sont introduits à gauche, *excepté le bit de poids fort* dont la valeur conserve le signe de `a` avant le décalage
- ▶ En C, `>>` implémente *habituellement* le décalage logique (resp. arithmétique) pour les entiers non signés (resp. signés)
- ▶ `2 >> 1; // 1`  
`-4 >> 2; // -1 (en général)`  
`0xA >> 1; // 0x5`

Les opérateurs de décalage (et l'adressage des bits en général) fonctionnent de façon logique, indépendamment de l'endianness.

Par ex.  $(x \gg 7) \& 1$  est une expression qui vaut 0 ou 1 en fonction de la valeur du 7<sup>ième</sup> bit “ $x_7$ ” de  $x$  vu comme l'entier  $x = \sum_{i=0}^{31} x_i \times 2^i$ ; ce n'est pas (forcément) le 7<sup>ième</sup> bit de la représentation mémoire de  $x$  !

## Exercice : décalage circulaire

---

Écrivez une fonction qui implémente le décalage circulaire (la « rotation ») vers la gauche d'un entier non signé de 64 bits par  $r \in \llbracket 0, 63 \rrbracket$  positions. Faites attention aux UBs!

# Arithmétique efficace avec des opérandes de la forme $2^x$

---

D'un point de vue logique, on peut interpréter :

- ▶  $a \ll b$  comme la multiplication de  $a$  par  $2^b$
- ▶  $a \gg b$  comme la division (entière) de  $a$  par  $2^b$
- ▶  $a \& ((1 \ll b) - 1)$  comme le reste de la division (entière ; non signée) de  $a$  par  $2^b$

~>

- ▶ Les divisions par des puissances de 2 sont beaucoup plus efficaces que celles par des nombres arbitraires
- ▶ Ditto les multiplications, ainsi que (en général, mais pas toujours) les multiplications par des nombres dont l'écriture en base 2 a un poids faible (par ex.  $(a \ll 3) + a$  ;  $// a*9$ )

- ▶ Un bon compilateur remplacera une expression  $a * / \% b$  par la variante appropriée quand  $b$  est une puissance de 2 *constante*
- ▶ Le programmeur doit lui-même faire les substitutions si  $b$  est variable (ou inconnu à la compilation)
- ▶ La programmeure doit choisir avec sagesse la valeur des constantes pour lesquelles une puissance de 2 est admissible
- ▶ Ex. : certains algorithmes de réduction modulaire

## Éléments de langage (2)

Retour sur les types arithmétiques

Instructions bit (1)

Quelques types spéciaux (1)

## Compilation (2) : au sens très large

Le CPP

Débuggage (1)

Fichiers multiples (2) : visibilité

Bibliothèques

## Génération d'aléa (en C)

## Types spéciaux (1) : les tableaux (1)

---

Soit un type `type`, on peut déclarer une variable de type *tableau* sur ce type des façons suivantes :

```
type t[7]; // variable tableau `t' de taille 7
type t[7][90]; // variable tableau `t' à deux
  ↪ dim. de taille 7 × 90
type t[n]; // variable tableau `t' de taille
  ↪ variable `n' (à partir de C99)
```

- ▶ Cette déclaration s'associe d'une *allocation mémoire* sur la *pile* (cf. un prochain cours pour plus de détails)
- ▶  $\rightsquigarrow$  il ne faut pas déclarer des tableaux « trop grands » (par ex. de taille  $10^8$ ).

## Types spéciaux (1) : les tableaux (2)

---

Une variable de type tableau (de taille fixe ou variable) n'est pas exactement un pointeur (cf. prochain cours); elle n'est notamment pas assignable

- ▶ `int t[1]; t = 3; ~\r\n`  
error: array type 'int [1]' is not assignable
- ▶ `int t[a]; printf("%p\\n", t++); ~\r\n`  
error: cannot increment value of type 'int [a]'

En pratique, hors allocation (cf. prochain cours) pointeurs et tableaux se manipulent généralement de la même façon.

## Types spéciaux (1) : les tableaux (3)

---

Plusieurs options existent pour déclarer la signature d'une fonction prenant un tableau en argument. Si celui-ci est de taille fixe (connue à la compilation) on peut par exemple faire :

```
void f(int t[]);
```

```
void f(int t[10]); // équivalent au précédent,
```

→ *«auto-documenté»*

```
void f(int t[static 10]); // t doit pointer vers un
```

→ *tableau d'int d'au moins 10 éléments (possible*

→ *warning à la compilation si ce n'est pas le cas)*

Les deux dernières options documentent naturellement le code, et sont à préférer

## Types spéciaux (1) : les tableaux (4)

---

Si un tableau en argument est de taille variable (inconnue à la compilation), celle-ci doit être passée en argument, par exemple comme :

```
void f(int t[], size_t nelem);  
void f(size_t nelem, int t[nelem]); // équivalent au  
↪ précédent  
void f(int t[nelem], size_t nelem); // ne marche pas,  
↪ nelem «inconnu»
```

Encore une fois, la seconde option documente naturellement le code et est à préférer

## Types spéciaux (2) : enum

---

On peut utiliser un type `enum` pour associer un nom à une valeur entière `int`, par ex. :

```
enum status
{
    on,
    off,
};
enum status a = on;
if (a == 0) { a += off; }
```

↪ Aucune contrainte de typage, purement « visuel »

## Types spéciaux (3) : union (1)

---

Un type `union` sert à référencer un emplacement mémoire avec différents types  $\rightsquigarrow$  comme une `struct` (qu'on (re)verra plus en détails dans un prochain cours), mais les emplacements mémoire se chevauchent. Ex. :

```
union uint64d
{
    double d;
    uint64_t i;
};
union uint64d x;
x.d = M_SQRT2;
```

permet d'accéder à la représentation binaire (mantisse et exposant) du `double` `M_SQRT2` via `x.i`

## Types spéciaux (3) : union (2)

---

Autre exemple :

```
union uint64st
{
    uint64_t i;
    uint8_t t[8];
};
union uint64st x;
x.i = 0x0123456789ABCDEF;
```

permet d'accéder aux octets de `x.i` via `x.t[]` (dans ce cas, un résultat similaire pourrait être obtenu par cast de pointeurs)

## Éléments de langage (2)

Retour sur les types arithmétiques

Instructions bit (1)

Quelques types spéciaux (1)

## Compilation (2) : au sens très large

Le CPP

Débuggage (1)

Fichiers multiples (2) : visibilité

Bibliothèques

## Génération d'aléa (en C)

Un programme lisible est un programme plus facile à débogger

- ▶ Utilisez un mécanisme d'auto-indentation...
- ▶ un *prettifier* (« enjoliveur »), par ex. `clang-format`
- ▶ Utilisez un nombre de fichiers raisonnable
- ▶ Commentez a minima les points clefs de vos programmes, par ex. les contraintes sur les arguments de fonction

## Éléments de langage (2)

Retour sur les types arithmétiques

Instructions bit (1)

Quelques types spéciaux (1)

## Compilation (2) : au sens très large

**Le CPP**

Débuggage (1)

Fichiers multiples (2) : visibilité

Bibliothèques

## Génération d'aléa (en C)

# Préprocesseur

---

- ▶ Le préprocesseur est appelé en début de compilation avant les différentes phases de traduction
- ▶ Il exécute notamment les directives *#include*
- ▶ Il dispose aussi d'un langage de macros avancé, et de symboles prédéfinis

## Quelques points sur les symboles

---

- ▶ On peut définir des symboles, avec ou sans valeurs, par ex. :  
*#define MTHREAD*  
*#define MAX\_THREADS 64*
- ▶ On peut faire appel à ces symboles dans tout fichier où ils sont définis ; chaque occurrence (isolée, hors d'une chaîne de caractères) de la chaîne MAX\_THREADS sera remplacée par 64 par le préprocesseur
- ▶ On peut annuler la définition d'un symbole pour la suite d'un fichier :

*#undef SIMD*

- ▶ Il existe plusieurs symboles prédéfinis, dont notamment deux parfois utiles pour le débogage
    - ▶ `__LINE__` est remplacé par le numéro de la ligne où il se trouve
    - ▶ `__func__` est remplacé par le nom de la fonction où il se trouve (si pertinent)
  - ▶ Exemple d'utilisation :
- ```
printf("Coucou from %s @%d\n", __func__, __LINE__);
```

# Compilation conditionnelle

---

- ▶ Le préprocesseur possède aussi des tests *#if*, *#ifdef*, *#ifndef*, *#else*, *#elif*, *#endif*...
- ▶ Permet de facilement commenter un gros bloc de code :

```
#if 0  
.....  
#endif
```

- ▶ Permet d'activer du code en fonction de contraintes extérieures :

```
#ifdef __SIMD_AVX  
.....  
#elif defined(__SIMD_SSSE3) || defined(__X86_64)  
.....  
#endif
```

# Inclusion conditionnelle

---

- ▶ En C, on ne doit pas déclarer plusieurs fois une même fonction
- ▶ Mais avant d'inclure un fichier `.h`, on ne sait pas forcément s'il a déjà été inclus ou pas, ce qui peut mener à des erreurs
- ▶ Une solution classique :

```
#ifndef __HAI_H  
#define __HAI_H  
    void print_hai();  
#endif
```

- ▶ Nécessite une absence de conflit des symboles, et une « coopération » des développeurs/ses

# Définitions en ligne de commande

---

- ▶ On peut définir un symbole, y compris avec une valeur à l'appel au compilateur :

```
cc -DMAX_THREADS=128 p.c
```

- ▶ Nécessite éventuellement un test *#ifndef* dans le source pour ne pas être écrasé

# Macros à arguments

---

- ▶ On peut aussi définir des macros à argument, *qui ne sont pas des fonctions*
- ▶ L'expression correspondant au résultat est calculée par le préprocesseur et substituée à l'appel
- ▶ Par ex.

```
#define MIN(X,Y) X < Y ? X : Y
```

```
...
```

```
    MIN(NTHREADS, 12);
```

```
...
```

## Macros à arguments : quelques pièges

---

- ▶ Dans le cas suivant :

```
#define SQ(X) X*X
```

```
...
```

```
    SQ(a+b);
```

```
    SQ(i++);
```

```
...
```

la première expression sera traduite en  $a+b*a+b$  qui n'a pas la valeur attendue, et dans la seconde `i++` sera évalué deux fois et `i` incrémenté deux fois

- ▶ Le premier cas peut se régler en (sur)parenthésant la définition : *#define* SQ(X) ((X)\*(X))
- ▶ Le second en s'abstenant d'utiliser des effets de bords dans les arguments des macros

## Éléments de langage (2)

Retour sur les types arithmétiques

Instructions bit (1)

Quelques types spéciaux (1)

## Compilation (2) : au sens très large

Le CPP

Débuggage (1)

Fichiers multiples (2) : visibilité

Bibliothèques

## Génération d'aléa (en C)

Les compilateurs C proposent de nombreuses options qui permettent de détecter efficacement des erreurs (d'étourderie ou non). Idéalement, utilisez toujours au moins :

- ▶ `-Wall` (tous les *warnings*)
- ▶ `-Wextra` (encore plus de warnings)

Il faut **lire** et **corriger** les warnings remontés par le compilateur. Pour vous-y forcer, utilisez :

- ▶ `-Werror` (un avertissement vaut comme une erreur)

En cas de débogage « actif » (cf. plus loin) :

- ▶ `-g` ou `-g3` (ajoute des informations de suivi dans le code)
- ▶ `-fno-omit-frame-pointer` (garde des des informations utiles)

Remarque : le bug ne vient (très probablement) pas de la libc...

## En prévention (2)

---

Il peut parfois être utile d'instrumenter le programme pour tester des conditions lors de l'exécution. Un outil utile pour cela est la macro `assert` :

```
#include <assert.h>
```

```
...
```

```
    assert(d > 0);
```

```
    x = y % d;
```

```
...
```

```
↪
```

```
Assertion failed: (d > 0), function main, file
```

```
↪ toto.c, line 11.
```

```
zsh: abort      ./a.out
```

Ces assertions sont plutôt un outil de développement/debug. Pour un programme fini, on peut les laisser dans le code (utile comme documentation) et les désactiver en compilant avec `-DNDEBUG`

Les bugs étant malgré tout fréquents, on peut aussi utiliser des outils afin de faciliter leur détection & correction, notamment :

- ▶ Valgrind : <https://valgrind.org>
  - ▶ Utilisation : installer le logiciel et invoquer un programme `a.out` comme `> valgrind ./a.out`
- ▶ ASan/UBSan : <https://github.com/google/sanitizers/wiki/AddressSanitizer>
  - ▶ Utilisation : compiler (et lier) avec les options `-fsanitize=address` et `-fsanitize=undefined`

## Valgrind : exemple de sortie

---

```
$ valgrind ./a.out
...
==4854==
==4854== Conditional jump or move depends on
↳ uninitialised value(s)
==4854==      at 0x100000F19: main (toto.c:8)
...
```

## Valgrind : exemple de sortie (2)

---

```
$ valgrind ./a.out
...
==2564== Invalid write of size 4
==2564==    at 0x100000F2B: main (toto.c:8)
==2564==   Address 0x100dea800 is 0 bytes after a
↪  block of size 80 alloc'd
==2564==    at 0x1000AC086: malloc
==2564==   by 0x100000F0A: main (toto.c:5)
```

# ASan : exemple de sortie (extrait)

---

```
$ ./a.out
==4970==ERROR: AddressSanitizer: stack-buffer-overflow on address
↳ 0x7ffeee06f8f0 at pc 0x000101b90cbd bp 0x7ffeee06f7d0 sp
↳ 0x7ffeee06f7c8
READ of size 4 at 0x7ffeee06f8f0 thread T0
Address 0x7ffeee06f8f0 is located in stack of thread T0 at offset 272
↳ in frame
  #0 0x101b90b5f in main toto.c:4
This frame has 1 object(s):
  [32, 272) 't' (line 5) <== Memory access at offset 272 overflows
  ↳ this variable
SUMMARY: AddressSanitizer: stack-buffer-overflow toto.c:7 in main
Shadow bytes around the buggy address:
  0x1fffddc0ded0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1fffddc0dee0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1fffddc0def0: 00 00 00 00 00 00 00 00 00 00 00 00 00 f1 f1 f1 f1
  0x1fffddc0df00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x1fffddc0df10: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00[f3]f3
  0x1fffddc0df20: f3 f3 f3 f3 f3 f3 f3 f3 00 00 00 00 00 00 00 00
  0x1fffddc0df30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

- ▶ Utilisez systématiquement ASan (et UBSan) lors du processus de développement (sans attendre la présence visible d'un bug)
- ▶ En cas de bug observé non détecté par ASan/UBSan, utiliser Valgrind (mieux vaut dans ce cas désactiver ASan et UBSan).

À appliquer en TP (et en particulier lors du TP « débuggage ») !

Il existe aussi des analyseurs *statiques*, qui n'exécutent pas (« réellement ») le code analysé. Trois exemples (plus ou moins faciles d'utilisation, et sans plus de détails) :

- ▶ T-Snippet : <https://tsnippet.trust-in-soft.com/#>
- ▶ Frama-C : <https://frama-c.com/>
- ▶ Clang Static Analyzer :  
<https://clang-analyzer.llvm.org/>

- ▶ En cas de bug non immédiatement trouvé par les méthodes précédentes, on peut utiliser un débogueur comme gdb
- ▶ C'est la bonne alternative aux débugeage par `printf` ; apprenez à vous en servir
- ▶ Pour un meilleur fonctionnement de gdb, compilez votre programme avec les options mentionnées plus haut, et sans optimisation (pas d'option `-O`, ou alors `-O1` ou `-O0`) (Attention : cela peut changer la manifestation du bug par rapport un autre niveau d'optimisation)

gdb permet de faire tourner un programme dans un environnement contrôlé, et d'interrompre l'exécution en cas :

- ▶ de signal envoyé par le système, capturé par gdb, par ex :
  - ▶ SIGSEGV (erreur de segmentation ; courant en cas de bug « mémoire »)
  - ▶ SIGABRT (assert évalué à faux)  $\rightsquigarrow$  gdb s'utilise bien en conjonction avec les assert
- ▶ d'atteinte d'un *breakpoint* défini par l'utilisateur/utilisatrice

On trouve beaucoup de *gdb cheat sheets* sur internet, qui listent de nombreuses commandes et permettent d'entrevoir les possibilités de gdb

## gdb : quelques commandes

---

- ▶ Dans un shell, préfixez l'invocation de votre programme par gdb, par ex. `> gdb prog <args>`
- ▶ Ajoutez un ou plusieurs *breakpoints* (une ligne du source ; une fonction...), par ex. `(gdb) break main.c:756` ou `(gdb) break fun1`
- ▶ Lancez votre programme avec `run`
- ▶ En cas d'interruption du programme : `n(ext)` ou `s(tep)` ou `finish` ou `c(ontinue)`... Mode *visuel* avec `-` ou `Ctrl + x + a`
- ▶ Si besoin, affichez une expression avec `p(rint)`, par ex. `(gdb) p toto`, `(gdb) p toto + titi`
- ▶ Si besoin, modifiez une lvalue avec `p(rint)`, par ex. `(gdb) p toto=4`
- ▶ En cas d'interruption d'un programme : `bt` pour afficher la pile d'appels ; `up` et `down` pour naviguer dedans

- ▶ Pour déboguer un programme dont les sources ne sont pas disponibles, un outil puissant est `strace`, dont l'utilisation est simplement `> strace toto`
- ▶  $\rightsquigarrow$  liste les appels systèmes effectués par le programme, avec leurs arguments, leurs valeurs de retour, les éventuelles erreurs... ainsi que les signaux qui sont levés
- ▶ Cf. la documentation pour plus d'information (“Students, hackers and the overly-curious will find that a great deal can be learned about a system and its system calls by tracing even ordinary programs”)

Un programme correct n'est pas forcément un programme efficace. Pour mesurer les performances d'un programme on peut par exemple :

- ▶ Mesurer le temps réel ("wall time") d'exécution, par ex. via `time` (dans un shell) ou `clock_gettime` dans un programme (Mais attention à la fiabilité/précision de cette approche)
- ▶ Inspecter les compteurs de performance, par ex. avec `perf stat (-d (-d (-d))) prog`
- ▶ Échantillonner le temps passé dans chaque fonction, par ex. avec `perf record prog` puis `perf report` ou `perf script`

## Éléments de langage (2)

Retour sur les types arithmétiques

Instructions bit (1)

Quelques types spéciaux (1)

## Compilation (2) : au sens très large

Le CPP

Débuggage (1)

Fichiers multiples (2) : visibilité

Bibliothèques

## Génération d'aléa (en C)

# Symboles et importation

---

- ▶ Lors de la compilation d'un fichier `.c` vers un fichier `.o`, le compilateur doit gérer l'absence possible de certains symboles (fonctions, variables) dans le `.c`
- ▶ Si ces symboles ne sont pas trouvés par le linker  $\rightsquigarrow$  échec de la compilation

```
$ cc -c toto.c titi.c
```

```
$ cc toto.o titi.o
```

```
Undefined symbols for architecture x86_64:
```

```
"_foo", referenced from: _main in toto.o
```

```
ld: symbol(s) not found for architecture x86_64
```

```
clang: error: linker command failed with exit code 1
```

```
↪ (use -v to see invocation)
```

## Symboles et importation (2)

---

La portée d'un symbole détermine s'il est accessible depuis un autre fichier/une autre fonction... :

- ▶ Portée externe : accessible de tout le monde (défaut pour les identifiants globaux : fonctions, variables globales)
- ▶ Portée interne : seulement accessible du fichier
- ▶ Pas de portée : différent à chaque occurrence (arguments de fonction, variables locales...)

Cette portée se contrôle explicitement (ou se modifie) via les mots-clefs `extern` et `static`.

# Restriction de portée

---

- ▶ Par défaut, une fonction a une portée externe : elle est accessible depuis n'importe quel autre fichier sur simple déclaration
- ▶  $\rightsquigarrow$  utilisation classique des couples de fichiers `.c` et `.h`
- ▶ On peut changer ce comportement en déclarant/définissant la fonction comme `static` :

```
static int fun(void)
{
    ...
}
```

- ▶ Intérêts : éviter des conflits de nom ; limiter l'exportation de symboles inutilisés ; empêcher des contournements de l'API
- ▶ (Même comportement pour les variables globales)

## Importation de variables globales

---

- ▶ Si une variable globale à portée externe est définie dans un fichier, on peut l'« importer » dans un autre en la déclarant `extern` (comme variable globale ou locale)
- ▶ Ne marche évidemment pas s'il n'y a pas de telle variable ou si elles sont `static`

```
// a.c
static int toto = 1;
// b.c
extern int toto;
...
```

Undefined symbols for architecture x86\_64:

- ↪ `"_toto"`, referenced from: `_main` in `b.o` ld:
- ↪ symbol(s) not found for architecture x86\_64

## Les variables locales `static`

---

Attention, `static` a un sens identique/différent pour les variables *locales* ?

- ▶ Une variable locale à une fonction peut être déclarée `static` afin d'être allouée une unique fois  $\rightsquigarrow$  « variable globale » visible uniquement dans la fonction
- ▶ Si la variable est initialisée à la déclaration (par ex. `static int a = 0`), cette initialisation n'est faite qu'une unique fois au début de l'exécution du programme
- ▶ Utile pour par ex. implémenter des compteurs de ressource

On a déjà vu plus tôt encore un autre (vraiment) différent (!)

## Éléments de langage (2)

Retour sur les types arithmétiques

Instructions bit (1)

Quelques types spéciaux (1)

## Compilation (2) : au sens très large

Le CPP

Débuggage (1)

Fichiers multiples (2) : visibilité

**Bibliothèques**

## Génération d'aléa (en C)

N.B. : Nous ne parlerons que brièvement des *bibliothèques partagées chargées dynamiquement*

Une telle bibliothèque regroupe un ensemble de fonctions qui peuvent ensuite être utilisées par un exécutable qui est lié dynamiquement à celle-ci. Quelques intérêts de cette approche :

- ▶ Le code de la bibliothèque n'est stocké qu'une fois (il n'est pas inclus dans les executables qui l'utilisent)  $\rightsquigarrow$  gain de place
- ▶ Une nouvelle version de la bibliothèque peut être utilisée par l'exécutable sans le recompiler (à condition que...)
- ▶ Un exécutable peut facilement changer de bibliothèque implémentant une certaine API (à condition que...)

Le principe même du chargement dynamique peut aussi être source de problèmes ou de limitations :

- ▶ Un exécutable utilisant une vieille version d'une bibliothèque peut ne plus fonctionner avec une nouvelle version
- ▶  $\rightsquigarrow$  au niveau d'un système complet, il peut être nécessaire de maintenir plusieurs versions d'une même bibliothèque
- ▶ L'essentiel du coût du liage (voire plus ? !) est payé à chaque fois que l'exécutable est lancé, plutôt qu'une seule fois à la compilation

# Bibliothèques partagées : exemples

---

Quelques bibliothèques courantes généralement disponibles sous forme dynamique :

- ▶ La bibliothèque C standard
- ▶ La bibliothèque C de maths
- ▶ Les bibliothèques GMP et MPFR de calcul en précision arbitraire

Des outils comme `objdump` ou `ldd` permettent d'obtenir des informations sur les bibliothèques utilisées par un exécutable

Pour lier dynamiquement une bibliothèque `libtoto.so` à un exécutable, il faut :

- ▶ (Généralement) ajouter une option `-ltoto` à la phase de liage de l'exécutable (soit `-l` suivi du nom de la bibliothèque, sans le préfixe `lib` ni le suffixe `.so`)
- ▶ (Exception : certaines bibliothèques comme la bibliothèque C standard n'ont généralement pas besoin d'être liées explicitement (elle peut néanmoins l'être via `-lc`))
- ▶ Attention : il faut que la bibliothèque soit connue du linker (comme un `.h` doit être connu du préprocesseur)  $\rightsquigarrow$  option `-L`
- ▶ Attention : il faut que la bibliothèque soit présente sur le système et trouvée par le loader au moment de l'exécution

Il est très simple de créer une bibliothèque partagée : il suffit d'ajouter l'option `-shared` lors de la phase de liage

Quelques conseils / conventions

- ▶ Une bibliothèque est généralement accompagnée d'un fichier `.h` compagnon qui déclare les fonctions (et autres symboles) de l'API de la bibliothèque
- ▶ Limitez les déclarations aux fonctions nécessaires ; déclarez toutes les autres comme `static` (cf. supra)
- ▶ Pour un projet de longue durée, numérotez chaque version différente et présentez un nom « par défaut » via un lien symbolique  $\rightsquigarrow$  simple d'utilisation, et laissez la possibilité de lier avec une version spécifique

## Éléments de langage (2)

Retour sur les types arithmétiques

Instructions bit (1)

Quelques types spéciaux (1)

## Compilation (2) : au sens très large

Le CPP

Débuggage (1)

Fichiers multiples (2) : visibilité

Bibliothèques

## Génération d'aléa (en C)

Il est souvent nécessaire d'utiliser des nombres « aléatoires » en informatique ; quelques cas d'usages :

- ▶ algorithmes probabilistes (cf. le cours d'algo, le cours d'AAE, ...)
- ▶ simulation
- ▶ tests logiciels
- ▶ cryptographie
- ▶ jeux vidéos (cf., hum, un prochain TP ?)

# Types de générateurs : TRNG

---

Idéalement, l'aléa utilisé dans un programme devrait être issu d'un « vrai » générateur de nombres aléatoires (*true random number generator*, ou TRNG) :

- ▶ un processus utilisant des données physiques imprévisibles (par ex. un dé, du bruit (thermique, acoustique, électromagnétique, radioactif...), ...) ensuite numérisées (et éventuellement retraitées)

Avantage :

- ▶ *En principe* capable de générer des bits suivant une distribution (proche de l') uniforme

Inconvénients :

- ▶ Nécessite un support matériel (le générateur lui-même...)
- ▶ Généralement relativement lent

# TRNG : sur vos machines ?

---

Les processeurs Intel relativement récents embarquent un TRNG accessible (indirectement) via deux instructions :

- ▶ `rdrand`, par ex. accessible via l'intrinsèque `_rdrand64_step` (nécessite le support RDRAND par le processeur)
- ▶ `rdseed`, par ex. accessible via l'intrinsèque `_rdseed32_step` (nécessite le support RDSEED par le processeur)

Cf. programme d'exemple `rdrand.c`

Pour plus d'informations sur :

- ▶ Les instructions disponibles sur processeurs Intel : <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>
- ▶ Les intrinsèques : <https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html>

## Types de générateurs : PRNG

---

En général, les nombres aléatoires utilisés dans des programmes proviennent de générateurs « pseudo-aléatoires » (*pseudo-random number generator*, ou PRNG)

- ▶ Un algorithme déterministe prenant en entrée une *graine* (seed) et produisant en sortie une suite de nombres « apparemment aléatoires » (toujours la même pour une même graine)

Avantages :

- ▶ Généralement plus rapide qu'un TRNG
- ▶ Permet de facilement générer plusieurs fois la même suite « aléatoire » (parfois utile, par ex. pour des tests)

Inconvénients :

- ▶ Aléa de possible mauvaise qualité (ex. `rand` historique en C)
- ▶ Nécessite une initialisation de la graine pour commencer...

# PRNG : initialisation

## Approches classiques pour initialiser un PRNG

« aléatoirement » (sous UNIX / en C)

- ▶ Avec le temps UNIX  
(`uint64_t seed = (uint64_t)time(NULL)`)  $\rightsquigarrow$  TRÈS MAUVAISE IDÉE (seulement 86 400 secondes en un jour ;  $\approx 10^9$  en 30 ans ; facile à prédire) (à peine mieux : utiliser `clock_gettime` ou autre)
- ▶ (Pour une étude de cas problématique en crypto, cf. par ex. <https://eprint.iacr.org/2023/912>)
- ▶ En lisant le pseudo-fichier `/dev/urandom` (raccourci sous Linux : `getrandom`, sous MacOS : `getentropy`)
- ▶ En utilisant `rdrand` ou `rdseed` (si disponible sur le CPU)
- ▶ Sinon ???

N.B. : En usage typique, l'initialisation d'un PRNG se fait *une seule fois* au début du programme (ou éventuellement d'une fonction)

# PRNG : quelles options en C

---

Générateurs de la bibliothèque standard (omission : `rand48`) :

- ▶ `rand` : extrêmement biaisé par rapport à l'uniforme, NE PAS UTILISER (n'est généralement plus disponible)
- ▶ `random` (à initialiser avec `srandom`) : moins biaisé que `rand` mais :
  - ▶ retourne seulement **31** bits d'aléa ?
  - ▶ s'initialise avec une graine de seulement 32 bits (pour les instantiations habituelles d'`unsigned`)

Générateurs externes « statistiques », par ex. :

- ▶ Mersenne Twister & cie.
- ▶ Xorshift & cie.

Générateurs externes « cryptographiques », par ex. :

- ▶ HMAC\_DRBG-SHA-512
- ▶ ChaCha20/8

## PRNG : un exemple « xorshift »

---

xoshiro256starstar (<https://prng.di.unimi.it/>) :

```
static uint64_t s[4];
uint64_t next(void)
{
    const uint64_t result = rotl(s[1] * 5, 7) * 9;
    const uint64_t t = s[1] << 17;
    s[2] ^= s[0];
    s[3] ^= s[1];
    s[1] ^= s[2];
    s[0] ^= s[3];
    s[2] ^= t;
    s[3] = rotl(s[3], 45);
    return result;
}
```

- ▶ Utiliser au moins `random` (mais attention aux 31 bits!)
- ▶ Initialiser les graines avec une bonne source d'aléa système (par ex. `getrandom`) ou matérielle (par ex. `_rdrand64_step`), et rien de basé sur le temps
- ▶ Si possible, utiliser un générateur rapide moderne (par ex. `xoshiro256starstar`)
- ▶ (Utiliser uniquement `_rdrand64_step` ← simple mais lent)

## Aléa : quelles distributions ?

---

- ▶ Les (bons) PRNGs renvoient (généralement) des nombres aléatoires (à peu près) uniformes sur  $\llbracket 0, 2^b - 1 \rrbracket$  (par ex.  $b = 31$  pour `random`)
- ▶ On peut avoir besoin de nombres uniformes sur  $\llbracket 0, N - 1 \rrbracket$ ,  $N$  qui n'est pas (toujours) une puissance de 2
- ▶ Une solution biaisée : `prng() % N`
  - ▶ Exercice : pourquoi un biais ?
  - ▶ Mais le biais est faible si  $(2^b \bmod N)/2^b \ll 1$  (Exercice : quantifiez-le grâce à votre distance statistique préférée)
- ▶ Une solution non-biaisée : l'échantillonnage par rejet (cf. remise à niveau en maths pour l'info ; programme d'exemple `randmod.c`)

Plus de détails en cours de Modèles proba-stats au S6 !