L3 MI — Programmation

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

2022–09–14/21

Principe

Un programme C peut être écrit sur plusieurs fichiers, si :

- Exactement un fichier contient une fonction `main`
- Toute fonction utilisée dans un fichier a été *déclarée* au préalable *dans ce fichier*
- Toute fonction utilisée a été *définie* dans un des fichiers (ou une bibliothèque externe)

Structure habituelle

Un programme écrit sur plusieurs fichiers comporte généralement :

- Plusieurs fichiers `.c`
- Pour chaque fichier `file.c`, un fichier `file.h` correspondant qui déclare les fonctions (pas forcément toutes) de `file.c` qui peuvent être visibles dans d'autres fichiers `.c`
  - Un fichier `.c` « inclura » les fichiers `.h` nécessaires (pas forcément tous)

Compilation séparée (pour de faux)

Un programme écrit sur deux fichiers `prog.c`, `file2.c` peut être compilé ainsi :

```
# inutile d'ajouter les fichiers .h éventuels
> cc -o prog prog.c file2.c
```

qui produira un fichier de sortie `prog`

- Chaque fichier est (re)compilé à chaque appel à `cc`
- Relativement peu d'intérêt

Compilation séparée en plusieurs étapes

On procède habituellement comme suit :

1. Compilation séparée des fichiers `.c` en fichiers `.o`

   ```
   > cc -c prog.c
   > cc -c file2.c
   ```

2. Édition des liens des fichiers `.o` pour produire un exécutable

   ```
   > cc -o prog prog.o file2.o
   ```

3. Si seul `prog.c` est modifié, on peut reproduire un exécutable en faisant uniquement

   ```
   > cc -c prog.c
   > cc -o prog prog.o file2.o
   ```

Compilation séparée en plusieurs étapes (suite)

Quelques avantages

- Permet de ne pas systématiquement tout recompiler $\rightsquigarrow$ gain de temps pour les gros programmes
- Permet de fournir un bout de programme uniquement sous forme de `.o` (en pratique on utilisera plutôt un format de bibliothèque partagée)

Quelques inconvénients

- Peut conduire à une multiplication des fichiers
- Complexifie le processus de compilation $\rightsquigarrow$ utilisation d'outils dédiés

Un outils pour la compilation : make

Objectifs de make :

- Permettre une compilation modulaire efficace (faire le strict nécessaire) et facilement configurable
- Passe par un langage simple exprimant des cibles, des dépendances, et des règles de dérivation

Exemple

Dans notre exemple précédent, on avait les dépendances suivantes

- `prog.o` ne peut être créé que si `prog.c` existe
- `file2.o` ne peut être créé que si `file2.c` existe
- `prog` ne peut être créé que si `prog.o` et `file2.o` existent

Ce qui se traduit par le Makefile suivant (attention, il faut utiliser des *tabulations* ⁉) :

```
prog.o: prog.c # cible : dépendance
        cc -c prog.c # commande à exécuter pour
        ↪  produire la cible

file2.o: file2.c
        cc -c file2.c

prog: prog.o file2.o
        cc -o prog prog.o file2.o
```

Make : gestion des dépendances

La résolution d'une dépendance comme :

```
prog.o: prog.c
        cc -c prog.c
```

se fait en exécutant la commande si :

- le fichier `prog.o` n'existe pas ;
- le fichier `prog.c` est plus récent que `prog.o` (il faut recompiler `prog.o` pour prendre en compte des changements éventuels dans `prog.c`)

Dans le cas contraire (`prog.o` existe et est plus récent que `prog.c`), rien n'est fait

Pour forcer une recompilation : `> rm prog.o` ou `> touch prog.c`

Utilisation de make

- En ligne de commande, `> make target` si les instructions sont dans un fichier appelé `Makefile`, ou
  `> make -f otherfile target` sinon
- Avec `target` un des labels se trouvant dans le fichier (par ex. `prog` dans l'exemple précédent)
- Si on ne spécifie pas de `target`, la cible en début du fichier est utilisée par défaut

Quelques cibles classiques

On ajoute souvent quelques ≪ fausses ≫ cibles à un makefile :

```
all: prog1 prog2 prog3 # tous les exécutables finaux

clean:
	rm *.o

run: prog1
	./prog1

check:
	# lance des tests...
```

Quelques options de compilation

Nous avons déjà vu `-c` et `-o` ; d'autres options courantes sont :

- `-Wall`, `-Wextra`... : options d'avertissement
- `-std=c89`, `-std=c99`... : options de standard
- `-S` : émission du code assembleur intermédiaire
- `-O2`, `-O3`... : options d'optimisation
- `-mavx`, `-mpclmul`, `-march=native`... : options d'architecture

Ainsi que :

- `-I` : option du préprocesseur permettant d'ajouter des dossiers explorés pour la directive `#include <....>`
- `-L` : option du *linker* permettant d'ajouter des dossiers explorés pour la recherche de bibliothèques partagées
- `-l...` : option du linker spécifiant une bibliothèque (par ex. `-lm` pour la bibliothèque mathématique standard) à utiliser

Make : règles implicites

Pour des programmes C, make dispose d'un certain nombre de règles implicites ; le makefile précédent peut se simplifier en :

```
prog.o: prog.c

file2.o: file2.c

prog: prog.o file2.o
```

Le compilateur utilisé est celui spécifié dans la variable d'environnement `CC`, ou `cc` par défaut

Personnalisation de l'environnement

On peut spécifier dans un makefile la valeur de variables d'environnement, par ex. :

```
# variables utilisées dans les règles implicites
# = : redéfinition complète, += : ≪augmentation≫
CC=clang
CPPFLAGS= -I/home/karpman/sw/soft/include
CFLAGS= -O3 -mavx2 -mavx512vl -maxv512bw
LDFLAGS+= -L/usr/local/lib -lm4ri
# variable quelconque
LOL=cat Makefile

printmk:
        $(LOL)
```

Personnalisation à l'invocation

Les variables d'un Makefile peuvent être redéfinies à l'invocation :

```
# suffisant pour une variable d'environnement prédéfinie
> CC=gcc-9 make
# -e : utilise la définition de l'environnement
> LOL="echo 'hai'" make -e printmk
```

Quelques options pratiques :

```
# affiche les commandes sans les exécuter
> make --dry-run
# exécute les commandes en parallèle (32 au plus)
> make -j 32
```

Compilation multi-fichiers

Préprocesseur & macros

Génération d'aléa (en C)

Retour sur les expressions

Retour sur les types arithmétiques

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
```

Symboles prédéfinis

- Il existe plusieurs symboles prédéfinis, dont notamment deux utiles pour le débuggage
  - `__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

Compilation multi-fichiers

Préprocesseur & macros

Génération d'aléa (en C)

Retour sur les expressions

Retour sur les types arithmétiques

L'aléa dans la programmation

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

- algorithmes probabilistes (cf. peut-être le cours d'AAE)
- simulation
- tests logiciels
- cryptographie
- jeux vidéos (cf., hum, TP3 ?)

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` 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 `gettimeofday` ou autre)
- 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;
}
```

Bonne pratique / conseils dans le cadre du cours

- 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` $\leftarrow$ simple mais lent)

Aléa : quelles distributions ?

- Les (bons) PRNGs renvoient (généralement) des nombres aléatoires (à peu près) uniformes sur $[\![0, 2^b - 1]\!]$ (par ex. $b = 31$ pour `random`)
- On peut avoir besoin de nombres uniformes sur $[\![0, N - 1]\!]$, 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 $(2^b \mod N)/2^b \ll 1$ (Exercice : quantifiez-le grâce à votre distance statistique préférée)
- Une solution non-biaisée : l'échantillonnage par rejet (cf. programme d'exemple `randmod.c`)

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

Compilation multi-fichiers

Préprocesseur & macros

Génération d'aléa (en C)

Retour sur les expressions

Retour sur les types arithmétiques

Dualité expressions/instructions

On dispose en C :

- d'expressions, par ex. `3*x + 15`
- d'instructions, par ex. `x = 3` ou encore `fun(6)`

Une certaine particularité du langage est que :

- certaines expressions ont des effets de bord, par ex. d'affectation comme `i++`
- les instructions retournent une valeur, comme les expressions ; on peut par ex. faire `a = (b = 3)`

Exemple : les opérateurs unaires (pour référence)

- `i++` : la valeur de l'expression est `i`, qui est ensuite incrémenté
- `++i` : i est incrémenté, et la valeur de l'expression est `i`
- `i--` : la valeur de l'expression est `i`, qui est ensuite décrémenté
- `--i` : i est décrémenté, et la valeur de l'expression est `i`

Une conséquence piégeuse classique

- L'instruction d'affectation est aussi une expression, dont la valeur est la quantité affectée (si c'est un nombre)
- Il est donc licite d'écrire `if (x = 1) { ... }` mais le résultat n'est pas forcément celui attendu
- (Un compilateur moderne émettra généralement un avertissement)

La virgule

- Les instructions/expressions s'enchaînent généralement avec un `;`
- On peut aussi combiner plusieurs expressions avec une `,` : toutes les instructions sont exécutées, mais seule la valeur de la dernière expression sera retournée
- Utilisation principalement dans les blocs de `for`, etc. par ex :

```
for (int i = 0, j = 0; i < n; i++, j += i)
{ ... }
```

Compilation multi-fichiers

Préprocesseur & macros

Génération d'aléa (en C)

Retour sur les expressions

Retour sur les types arithmétiques

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* $x \in [\![0, s]\!]$, d'un *exposant* $e \in [\![-a, b]\!]$ 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 ($\in [\![-126, 127]\!]$)
- `double` : mantisse : 52 bits ($+$ 1 gratuit), exposant 11 bits ($\in [\![-1022, 1023]\!]$)

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 (cf. TP0)
- 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`) $\rightarrow$ 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 $\leftrightarrow$ 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
  ```
  ```

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`

Représentation logique ; physique

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 erreur 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 »

$\rightsquigarrow$ Ces conversions sont en général inutiles :

- Le processeur travaille déjà avec la « représentation binaire »
- ... et fournit des instructions permettant de la manipuler directement (cf. un prochain cours)

$\rightsquigarrow$ 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 $x \ \texttt{AND} \ y = 0$
- En C, ceci peut se calculer facilement et simultanément sur tous les bits d'un `unsigned` via l'expression `a & b == 0` (plus de détails dans un prochain cours)

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) :

```c
#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;
}
```

Endianness (cont.)

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