Programmation « orientée système »
LANGAGE C – BIBLIOTHÈQUES

Jean-Cédric Chappelier

Laboratoire d'Intelligence Artificielle
Faculté I&C

Objectifs du cours d'aujourd'hui

- présentation générale (survol) des bibliothèques C

Bibliothèques standard

Les bibliothèques standard (d'outils) C **facilitent la programmation**.

Les outils standards C(89 et 95) sont constitués des 18 « paquets » suivants :

`<assert.h>`	tests d'assertions pendant l'exécution (désactivés par `#define NDEBUG`)
`<ctype.h>`	divers tests/traitements sur les caractères
`<errno.h>`	code d'erreurs retournés dans la bibliothèque standard
`<float.h>`	diverses informations sur la représentation des réels
`<iso646.h>` (C95)	écriture texte des opérateurs logiques (`and`, `or`, ...), digraphes et trigraphes
`<limits.h>`	divers tests sur les entiers
`<locale.h>`	adaptation à diverses langues
`<math.h>`	divers définitions mathématiques
`<setjmp.h>`	branchement non locaux
`<signal.h>`	contrôle des signaux (processus)
`<stdarg.h>`	nombre variables d'arguments
`<stddef.h>`	diverses définitions (types et macros)
`<stdio.h>`	entrées sorties de base
`<stdlib.h>`	diverses opérations de base utiles
`<string.h>`	manipulation des chaînes de caractères à la C
`<time.h>`	diverses conversions de date et heures
`<wchar.h>` (C95)	caractères étendus (« wide chars »), genre UTF-8 (et leurs chaînes)
`<wctype.h>` (C95)	classification des codes de caractères étendus

Bibliothèques standard

Les standards C99 et C11 ajoutent les 11 « paquets » suivants :

`<complex.h>` (C99)	nombres complexes
`<fenv.h>` (C99)	outils pour tester/manipuler les états des nombres à virgule flottante
`<inttypes.h>` (C99)	macros pour les `stdint`
`<stdbool.h>` (C99)	booléens
`<stdint.h>` (C99)	entiers de tailles définies (`int8_t`, etc.)
`<tgmath.h>` (C99)	« type generic » macro wrapper pour `math.h` et `complex.h`
`<stdalign.h>` (C11)	macro pour alignement mémoire de `struct`
`<stdatomic.h>` (C11)	types atomiques (multi-threading)
`<stdnoreturn.h>` (C11)	`noreturn` macro pour fonctions
`<threads.h>` (C11)	multi-threading
`<uchar.h>` (C11)	UTF-16 & UTF-32

Quelques outils de `stdlib.h`

`<stdlib.h>` est la « bibliothèque standard » de C.

En plus de `malloc`, `realloc` et `free` (déjà vus), de nombreux utilitaires y sont définis, dont les

- nombres aléatoires,
- des fonctions de conversion de chaînes,
- et une fonction de tri `qsort` (déjà vue deux fois).

Nombres aléatoires

La génération de nombres au hasard sur ordinateur se fait avec des générateurs dit « pseudo-aléatoires » qui pour une valeur initiale donnée (appelée « **graine** » [« seed » en anglais]) donnent toujours la **même** séquence « aléatoire ».
(Cela peut être utile pour déverminer un programme utilisant des nombres aléatoires.)

`int rand()` retourne un nombre au hasard entre `0` et `RAND_MAX`

et `void srand(unsigned int seed)` sert à initialiser la graine.

Pour avoir une série de nombres aléatoires différente à chaque utilisation du programme, il faut utiliser une graine différente à chaque fois.

Cela se fait souvent en utilisant comme graine la valeur de l'horloge de l'ordinateur à cet instant :

```
#include <time.h>
#include <stdlib.h>
// ...
    srand(time(0));
// ...
```


Nombres aléatoires (2)

MAIS `rand` est connu pour être un **mauvais générateur aléatoire**

Dans des applications critiques (e.g. cryptographie), on cherchera donc à utiliser d'autres outils (non standards) [vaste sujet]

Si la machine le permet (non standard non plus), on préfèrera également utiliser `random`, un générateur uniforme de nombres **entiers**, et `drand48` un générateur uniforme de nombres **réels** dans [0, 1[.

Autres outils de `stdlib.h`

On trouve aussi dans `stdlib` (au lieu de `math` ! !), les fonctions « valeur absolue » pour les entiers :

`int abs(int)`
et `long int labs(long int)`

et diverses fonctions de conversion de chaînes (au lieu d'être dans `string` ! !)

`double atof(char const* s)`	convertit en `double` le nombre écrit dans `s`
`int atoi(const char* s)`	convertit en `int` le nombre écrit dans `s`
`long int atol(const char* s)`	convertit en `long int` le nombre écrit dans `s`

Détails de `math.h`

Voici quelques fonctions définies dans la bibliothèque `math`.
Toutes ces fonctions prennent un/des argument(s) `double` et retournent un `double`

`acos`	arccos
`asin`	arcsin
`atan`	arctan
`ceil`	$\lceil x \rceil$, entier supérieur
`cos`	cos
`cosh`	cosinus hyperbolique
`exp`	exp
`fabs`	valeur absolue
`floor`	$\lfloor x \rfloor$, entier inférieur
`fmod(x,y)`	`x % y` mais pour des `double`
`log`	ln, logarithme népérien
`log10`	log, logarithme en base 10
`pow(x,y)`	$x^y = \exp(y \ln x)$
`sin`	sin

EPFL

`math.h` (2)

`sinh`	sinus hyperbolique
`sqrt`	$\sqrt{}$
`tan`	tan
`tanh`	tangente hyperbolique

En cas d'erreur (domaine d'application, dépassement dans la capacité par excès (valeurs trop grandes ou « overflow ») ou par défaut (valeurs trop petites, ou « underflow »), arrondi), divers comportements sont possibles suivant la configuration de la machine.

Depuis C99, deux mécanismes de report d'erreur sont possibles :

- `errno` (dont nous avons déjà parlé)
- et les « *floating-point exceptions* » de la bibliothèque `fenv`.

Le résultat en cas d'erreur dépend de la fonction (et est souvent non spécifié (« *implementation-defined* »)).

Si c'est un « *overflow* », la valeur `HUGE_VAL` est retournée.

EPFL

`math.h` : erreurs

La configuration de report de messages d'erreur peut être testée à l'aide de la macro `math_errhandling` definie dans `math.h` :

- `math_errhandling & MATH_ERRNO` est vrai si `errno` est utilisé
- `math_errhandling & MATH_ERREXCEPT` est vrai si `fenv` est utilisé

(les deux peuvent être vrais).

cas d'erreur	exemples	`errno`	`fetestexcept()`
pôle	`1.0 / 0.0`, `log(0.0)`	`ERANGE`	`FE_DIVBYZERO`
argument hors du domaine de définition	`sqrt(-1.0)` `asin(1.5)`	`EDOM`	`FE_INVALID`
« *overflow* »	`DBL_MAX + 1`	`ERANGE`	`FE_OVERFLOW`
« *underflow* »	`DBL_TRUE_MIN / 2.0`	inchangée ou `ERANGE` (« *i.-d.* »)	`FE_UNDERFLOW` ou pas (« *i.-d.* »)
arrondi (valeur non exactement représentable)	`1.0 / 10.0` `sqrt(2.0)`	inchangée	`FE_INEXACT` ou pas (« *i.-d.* »)

i.-d. = *implementation-defined*

EPFL

`math.h` : exemple `errno` et `HUGE_VAL`

```c
/* sur une machine où  math_errhandling & MATH_ERRNO
 *                 et  math_errhandling & MATH_ERREXCEPT sont vrais. */

void f(double x) {
  // toujours remettre à 0 avant de les utiliser (pour les maths)
  feclearexcept(FE_ALL_EXCEPT);
  errno = 0;

  printf("1/x = %g\n", 1.0 / x);
  show_errno();
  show_fe_exceptions();
}
// ...            // résultat  errno   fetestexcept(?)
f(2.0);           // 0.5       0       all 0
f(10.0);          // 0.1       0       FE_INEXACT
f(DBL_MAX);       // 5.6e-309  0       FE_INEXACT FE_UNDERFLOW
f(DBL_MIN);       // 4.5e+307  0       all 0
f(1.0/DBL_MAX);   // inf       0       FE_INEXACT FE_OVERFLOW
f(DBL_TRUE_MIN); // inf       0       FE_INEXACT FE_OVERFLOW
f(0.0);           // inf       0 [??]  FE_DIVBYZERO

log( 0.0);        // -inf      ERANGE  FE_DIVBYZERO
log(-1.0);        // -NaN      EDOM    FE_INVALID
```


EPFL

float.h (Rappel)

constante	description	exemple de valeur
DBL_DIG	nombre de caractères significatifs	15
DBL_EPSILON	plus petite valeur x telle que $1 + x \neq 1$	2.22045e-16
DBL_MAX	plus grande valeur représentable sur un double	1.79769e+308
DBL_MIN	plus petite valeur *normalisée* représentable sur un double	2.22507e-308
DBL_TRUE_MIN (C11)	plus petite valeur représentable sur un double	4.94066e-324

limits.h (Rappel)

constante	description	exemple de valeur
CHAR_BIT	nombre de bits dans un char	8
UCHAR_MAX	plus grande valeur représentable dans un unsigned char	255
SCHAR_MAX	idem pour un signed char	127
SCHAR_MIN	plus petite valeur signed char	-128
LONG_MAX		2147483647
LONG_MIN		-2147483648
INT_MAX		2147483647
INT_MIN		-2147483648
SHRT_MAX		32767
SHRT_MIN		-32768
ULONG_MAX		4294967295
UINT_MAX		4294967295
USHRT_MAX		65535

ctype.h

Quelques fonctions de ctype :

```
int isalnum(int c);     isalpha(c)||isdigit(c)
int isalpha(int c);     isupper(c)||islower(c)
int iscntrl(int c);     caractère de contrôle
int isdigit(int c);     un chiffre ([0-9])
int isgraph(int c);     isprint(c)&&!isspace(c)
int islower(int c);     lettre minuscule
int isprint(int c);     caractère affichable
int ispunct(int c);     isgraph(c)&&!isalpha(c)
int isspace(int c);     une « espace » (TAB, à-la-ligne, blanc)
int isupper(int c);     lettre majuscule
int isxdigit(int c);    chiffre ou [a-fA-F]
int tolower(int c);     convertit c en minuscule (si approprié)
int toupper(int c);     convertit c en majuscule
```


locale.h

Cette bibliothèque sert à gérer les spécificités de format et caractères propres aux langues
par exemple pour modifier le comportement des fonctions précédentes (is...)

```
char* setlocale(int category, const char* locale);
```

permet de définir le type de langue utilisé
category peut valoir

LC_ALL	tous les aspects à la fois
LC_COLLATE	règles pour caractères composés (e.g. accents)
LC_CTYPE	classe de caractères, conversion, comparaison majuscules/minuscules, ...
LC_MESSAGES	messages (dans la langue précisée)
LC_MONETARY	format monétaire
LC_NUMERIC	format des nombres (e.g. 1'234)
LC_TIME	formats date et heure

« locale » usuels : "C", "en", "french", "en.utf8"

locale.h (2)

Exemple d'utilisation :

```
char* test;
char locale[] = "french";

test = setlocale(LC_ALL, locale);
if (test == NULL) {
    fprintf(stderr, "setlocale
failed for \%s.\bsn", locale);
    perror("Message");
    return 1;
}
```

Exemples d'effets :

en locale `"C"` :
- `isalpha('é')` ☞ faux
- `toupper('é')` ☞ é

en locale `"french"` :
- `isalpha('é')` ☞ vrai
- `toupper('é')` ☞ É

stdarg.h : introduction

La surchage n'existant pas en C, c'est bien la **MÊME** fonction `printf()` que vous utilisez quand vous faites :

```
printf("Bonjour !\n");                  // 1 argument
printf("J'ai %u ans.\n", age);          // 2 arguments
printf("J'habite \%s, %u\n", rue, num); // 3 arguments
// ...
```

Comment est-ce possible ?

stdarg.h

La bibliothèque `stdarg` permet de définir des fonctions avec un nombre variable d'arguments.

Attention ! On peut vite faire des bêtises avec cela. À **n'utiliser qu'avec précaution !**

`va_list` est le type « nombre variable d'arguments »

`va_start(va_list, nom)` permet de définir où ils commencent (après l'argument *nom* de la fonction)
et on doit utiliser `va_end(va_list)` une fois le traitement des arguments terminé

type `va_arg(va_list, type)` renvoit la valeur de l'argument désigné, lu comme un pointeur sur *type*
et passe à l'argument suivant

Attention ! Ce sont des **macros !** (avec tout ce que cela implique sur l'évaluation de leur arguments)

stdarg.h (2)

Exemple :

```
void affiche_doubles(size_t nb_args, ...)
{
    va_list arguments;
    va_start(arguments, nb_args);

    for (size_t i = 0; i < nb_args; ++i) {
        printf(" %f\n", va_arg(arguments, double));
    }

    va_end(arguments);
}
// ...
  f(2, 3.1, 4.5);
  f(3, 233.11, 333.22, 444.18);
```

stdarg.h (3)

Autre exemple classique : faire sa propre variante de `printf`

```
int monprintln(const char* format, ...)
{
    va_list arguments;
    va_start(arguments, format);

    int retour = printf("dans mon printf on écrit : ");
    retour += vfprintf(stdout, format, arguments);
    putchar('\n'); ++retour;

    va_end(arguments);
    return retour;
}
```


Conclusion

Ici se termine donc notre apprentissage du C.

Il reste encore de nombreux aspects à appronfondir, notamment la gestion des processus, des threads, ...

mais cela sera vu dans le cours de « systèmes d'exploitation ».

Pour continuer à progresser en C

Sujets non (ou peu) abordés dans ce cours d'introduction :

- toutes les bibliothèques non présentées
- les `bitfields` et les opérateurs `&`, `|`, `^`, `<<`, `>>`
- autres modificateurs de déclaration (on a vu `const` et `extern`) : `static`, `register`, (`auto`,) `inline`, `restrict`, `volatile`
- les directives `#if`, `#line`, `#error`, `#pragma`
- les digraphes (C99) : `<:`, `:>`, `<%`, `%>`, `%:`, `%:%`