Programmation C et Système
Environnement d'exécution des processus (depuis un programme en C)

Régis Barbanchon

L2 Informatique

En C : accès aux arguments de la ligne de commande

En C, on accède aux arguments de la ligne de commande via les arguments de `int main (int argc, char * argv[])`, où `argv` est un tableau de chaînes, et l'entier `argc` est sa longueur :

```
// File: print-args.c
#include <stdio.h>
int main (int argc, char * argv[]) {
  printf ("argc: %d\n", argc);
  for (int k= 0; k < argc; k++)
    printf ("argv[%d]: <%s>\n", k, argv[k]);
  return 0;
}
```

```
$ clang -std=c99 -W -Wall -pedantic print-args.c -o print-args
$ ./print-args hello world "good bye"
argc: 4
argv[0]: <./print-args>
argv[1]: <hello>
argv[2]: <world>
argv[3]: <good bye>
```

- `argv[0]` contient le nom de la commande,
- ce nom compte comme 1 argument pour la longueur `argc`,
- les "vrais" arguments sont dans `argv[1]..argv[argc-1]`,
- `argv[argc]` vaut toujours `NULL`.

En C : terminaison normale avec status d'erreur

En C, on termine normalement le programme avec le status `s` via la fonction `void exit(int s)` de `<stdlib.h>`.
Si l'on est dans `main()`, on peut aussi retourner `s` via `return`.
La macro `EXIT_SUCCESS` vaut 0, et `EXIT_FAILURE` vaut 1.

```c
// File: args-are-equal.c
#include <stdlib.h>
#include <string.h>
int main (int argc, char * argv[]) {
  for (int k= 2; k < argc; k++)
    if (strcmp (argv[k-1], argv[k]) != 0)
      exit (EXIT_FAILURE); // = 1
  return EXIT_SUCCESS; // = 0
}
```

```
$ clang -std=c99 -W -Wall -pedantic args-are-equal.c -o args-are-equal
$ ./args-are-equal bla bla ; echo $?
0
$ ./args-are-equal bla bli ; echo $?
1
```

Les status d'erreur utilisent la convention inverse des Booléens : le succès est valeur *vrai*, et donc 0 est la valeur *vrai*.

```
$ if ./args-are-equal bla bla ; then echo "equal" ; else echo "not equal" ; fi
equal
$ if ./args-are-equal bla bli ; then echo "equal" ; else echo "not equal" ; fi
not equal
```

En C : terminaison anormale par envoi de signal

- La fonction `int raise(int signum)` de `<signal.h>` envoie le signal `signum` au programme.
- La fonction `void abort()` de `<stdlib.h>` est essentiellement équivalente à `raise(SIGABRT)`.
- Elle est invoquée via la macro `assert(cond)` de `<assert.h>` lorsque la condition `cond` est violée (= bug dans le source).

```c
// File: hexa.c
#include <assert.h>
#include <stdio.h>
char Int_digitToHexa (int digit) {
  assert (0 <= digit && digit < 16);
  char symbols[]= "0123456789ABCDEF";
  return symbols [digit];
}
```

```c
// ... (hexa.c continued) ...
int main (void) {
  for (int k= 0; k <= 16; k++) { // oops!
    char hexa= Int_digitToHexa (k);
    printf ("hexa(%d)= %c\n", k, hexa);
  }
  return 0;
}
```

Le programme avorte avec un message sur `stderr` pour `k=16` :

```
$ clang -std=c99 -W -Wall -pedantic hexa.c -o boom
$ ./boom 1> /dev/null
boom: hexa.c:6: char Int_digitToHexaSymbol(int):
Assertion ‘0 <= digit && digit < 16’ failed.
Aborted (core dumped)
```

```
$ echo $?            # SIGABRT = 6, and 128 + 6 = 134, hence $? = 134
134
```

En C : écriture sur la sortie standard et la sortie d'erreur

Pour l'écriture, 2 vars globlales sont déclarées dans `<stdio.h>` :

- `extern FILE * stdout` : la sortie standard,
- `extern FILE * stderr` : la sortie d'erreur.

Les flux associés sont ouverts en écriture, et on peut y écrire un caractère, une chaîne, ou des données formatées via :

```
int fputc   (int c, FILE *stream);
int fputs   (const char *s, FILE *stream);
int fprintf (FILE *stream, const char *format, ...);
```

`printf(...)` est un raccourci pour `fprintf(stdout, ...)`.

```
// File: hello.c
#include <stdlib.h>
#include <stdio.h>
int main (int argc, char * argv[]) {
  if (argc != 1+1) {
    fprintf (stderr, "Usage: %s name\n", argv[0]);
    exit (1);
  }
  fprintf (stdout, "Hello %s, how are you?\n", argv[1]);
  return 0;
}
```

```
$ clang -std=c99 -W -Wall -pedantic hello.c -o hello
```

```
$ ./hello Regis   1> output.log
$ cat ouput.log
Hello Regis, how are you?
```

```
$ ./hello   2> error.log
$ cat error.log
Usage: ./hello name
```

En C : lecture sur l'entrée standard

Pour la saisie, 1 var globlale est déclarée dans `<stdio.h>` :

- `extern FILE * stdin` : l'entrée standard.

Le flux associé est ouvert en lecture, et on peut y lire un caractère, une ligne, ou des données formatées via :

```
int    fgetc  (FILE *stream);
char * fgets  (char *s, int size, FILE *stream);
int    fscanf (FILE *stream, const char *format, ...);
```

`scanf(...)` est un raccourci pour `fscanf(stdin, ...)`.

```
// File: count-digits.c
#include <stdio.h>
#include <ctype.h>
int main (void) {
  int digitCount= 0;
  for (;;) {
    int ch= fgetc (stdin);
    if (ch == EOF) break;
    if (isdigit(ch)) digitCount++;
  }
```

```
// ... (count-digits.c continued) ...

  char * digitWord= (digitCount >= 2) ?
    "digits" : "digit";

  fprintf (stdout, "counted %d %s.\n",
           digitCount, digitWord);

  return 0;
}
```

```
$ clang -std=c99 -W -Wall -pedantic count-digits.c -o count-digits
```

```
$ ./count-digits
I have 3 cats.
[CTRL-D]
counted 1 digit.
```

```
$ echo "I have 15 cats." | ./count-digits
counted 2 digits.
```

En C : détection de la connection au terminal

L'entête <unistd.h> déclare la fonction isatty()
qui teste si le descripteur d'un fichier est connecté à un terminal.

```
int isatty (int fd);
```

Les fd de stdin, stdout, stderr sont respectivement 0, 1, 2.

```
// File: isatty.c
#include <stdio.h>
#include <unistd.h>

int main (void) {
  char * fdNames[3]= { "stdin", "stdout", "stderr"};
  for (int fd= 0; fd <= 2; fd++) {
    char * fdType= isatty (fd) ? "terminal" : "file or pipe";
    printf ("%s (fd %d) is a %s\n", fdNames[fd], fd, fdType);
  }
  return 0;
}
```

```
$ clang -stdc99 -W -Wall -pedantic isatty.c -o isatty
```

```
$ ./isatty
stdin (fd 0) is a terminal
stdout (fd 1) is a terminal
stderr (fd 2) is a terminal
```

```
$ echo lol | ./isatty
stdin (fd 0) is a file or pipe
stdout (fd 1) is a terminal
stderr (fd 2) is a terminal
```

```
$ ./isatty | cat
stdin (fd 0) is a terminal
stdout (fd 1) is a file or pipe
stderr (fd 2) is a terminal
```

```
$ ./isatty 2> /dev/null
stdin (fd 0) is a terminal
stdout (fd 1) is a terminal
stderr (fd 2) is a file or pipe
```

En C : accès aux variables d'environnement (1/2)

L'utilisateur peut déclarer la variable globale suivante :

```
extern char ** environ;
```

Elle pointe sur un tableau de chaînes de la forme "NAME=VALUE".
On n'en connaît pas la longueur mais il se termine par NULL.

```
// File: environ.c
#include <stdio.h>
extern char ** environ;
int main (void) {
  for (int k= 0; environ[k] != NULL; k++) {
    char * nameAndValue= environ[k];
    printf ("%2d: %s\n", k, nameAndValue);
  }
  return 0;
}
```

```
$ clang -std=c99 -W -Wall -pedantic environ.c -o environ
$ ./environ | fgrep ": LC_"
```

```
 1: LC_PAPER=fr_FR.UTF-8
 2: LC_ADDRESS=fr_FR.UTF-8
 5: LC_MONETARY=fr_FR.UTF-8
20: LC_NUMERIC=fr_FR.UTF-8
28: LC_TELEPHONE=fr_FR.UTF-8
43: LC_IDENTIFICATION=fr_FR.UTF-8
53: LC_MEASUREMENT=fr_FR.UTF-8
78: LC_TIME=fr_FR.UTF-8
79: LC_NAME=fr_FR.UTF-8
```

En C : accès aux variables d'environnement (2/2)

L'entête <stdlib.h> déclare la fonction :

```
char * getenv (const char * name);
```

Elle retourne la valeur d'une var d'env à partir de son nom.
Elle retourne NULL si la var d'env demandée n'existe pas :

```
// File: getenv.c
#include <stdlib.h>
#include <stdio.h>

int main (int argc, char * argv[]) {
  if (argc != 1+1) {
    fprintf (stderr, "Usage: %s name\n", argv[0]);
    exit (1);
  }
  char * name= argv[1];
  char * value= getenv (name);
  printf ("%s is %s\n", name, (value == NULL) ? "undefined" : value);
  return 0;
}
```

```
$ clang -std=c99 -W -Wall -pedantic getenv.c -o getenv
```

```
$ ./getenv
Usage: ./getenv name
```

```
$ ./getenv TOTO
TOTO is undefined
```

```
$ ./getenv LC_TIME
LC_TIME is fr_FR.UTF-8
```

(Les résultats dépendent évidemment de la config de votre shell.)

En C : accès à la locale (les paramètres régionnaux)

L'entête `<locale.h>` déclare une fonction `setlocale()` pour changer une catégorie de la locale (un paramètre régionnal) :

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

Pour `category`, on utilise des macros de `<locale.h>` telles que `LC_TIME`, `LC_CTYPE`, `LC_NUMERIC`, `LC_COLLATE`, `LC_ALL`, et :

- si `value` vaut `""`, la valeur est prise depuis la var d'env.
- si `value` vaut `NULL`, la valeur actuelle ne change pas (getter).
- la fonction retourne la nouvelle valeur de la catégorie.

```
// File: setlocale.c
#include <stdio.h>
#include <locale.h>
#include <time.h>

int main (void) {
  setlocale (LC_NUMERIC, "");
  printf ("PI: %f\n", 3.141593);
  setlocale (LC_TIME, "");
  time_t today= time (NULL);
  struct tm * todayTm= localtime (& today);
  char todayStr [1024];
  strftime (todayStr, 1024, "%c", todayTm);
  printf ("DATE: %s\n", todayStr);
  return 0;
}
```

```
$ clang -std=c99 -W -Wall -pedantic setlocale.c -o setlocale
```

```
$ LC_ALL=en_US.UTF-8 ./setlocale
PI: 3.141593
DATE: Tue 22 Jan 2019 08:23:06 PM CET
```

```
$ LC_ALL=fr_FR.UTF-8 ./setlocale
PI: 3,141593
DATE: mar. 22 janv. 2019 20:23:24 CET
```