Initiation au C
cours n°5

Antoine Miné

École normale supérieure

15 mars 2007

Plan du cours

- le débogage avec gdb,
- les chaînes de caractères.

Le débogueur gdb

Présentation de gdb

GDB = débogueur interactif permettant de :

- interrompre et reprendre l'exécution du programme,
- suivre l'exécution du programme pas à pas,
- poser des points d'arrêt,
- inspecter le contenu des variables,
- connaître la ligne exacte et le contenu des variables au moment d'un `Segmentation fault`.

Lancement de gdb

Préparation du programme

- compiler son programme avec l'option `-g`,
- éviter les options d'optimisation `-O2`, `-O3`,...

Exemple : `gcc toto.c -g -Wall -Wextra`

Lancement de gdb

Dans le shell, taper :

`gdb a.out`

gdb propose un *shell* interactif, d'invite `(gdb)`.

Exécution du programme sous gdb

Lancement du programme

```
(gdb) run
```

L'exécution se termine ou est suspendue dès que :

- le programme se termine normalement :
  `Program exited with code XXX`
- le programme se termine sur une erreur fatale :
  `Program received signal SIGSEGV, Segmentation fault`
- l'utilisateur tape contrôle+C :
  `Program received signal SIGINT, Interrupt`
- le programme passe par un point d'arrêt :
  `Breakpoint X, fun at file : line`

Points d'arrêt

Placement d'un point d'arrêt

(gdb) `break` *nom de fonction*
(gdb) `break` *n° de ligne*

Suspend l'exécution à chaque fois que le programme :

- entre dans la fonction indiquée, ou
- arrive au début de la ligne indiquée.

$\Rightarrow$ on peut alors entrer de nouvelles commandes gdb.

Notes :

- on peut placer des points d'arrêt avant `run`,
- on peut placer plusieurs points d'arrêt,
- un point d'arrêt reste actif jusqu'à sa destruction explicite : `delete` efface *tous* les points d'arrêt.

Reprise de l'exécution

Reprise de l'exécution

```
(gdb) continue
```

Possible uniquement si l'exécution n'est que suspendue :

Possible

- après contrôle+C,
- sur un point d'arrêt.

Impossible

- avant `run`,
- après terminaison normale,
- après terminaison sur erreur.

Note : `run` recommence l'exécution au début.

Exécution pas à pas

Exécution d'une ligne

```
(gdb) next
(gdb) step
```

Effet : exécute une seule ligne et rend la main.

L'exécution peut être suspendue avant la ligne suivante :

- par contrôle+C,
- en cas de point d'arrêt,
- en cas d'appel de procédure pour `step`.

Contraintes : identiques à celles de `continue`.

Voir aussi : `finish`, `next` *n*, `step` *n*.

Inspection des données

Affichage d'une expression

`(gdb) print expr`

Expressions autorisées :

- opérateurs C classiques, y compris déréférences `*`, `[]`,
- variables globales,
- variables locales de la fonction en cours d'exécution.

Contraintes : le programme doit :

- soit être suspendu (point d'arrêt, `next`, etc.),
- soit s'être terminé sur une erreur fatale.

Inspection de la pile

Inspection de la pile

```
(gdb) bt          (concis)
(gdb) bt full     (détaillé)
```

Effet : affiche la pile d'appel et les arguments des fonctions.

Déplacement dans la pile

On peut se "déplacer" dans la pile d'appel pour choisir une fonction.

Déplacement dans la pile

`(gdb) up`	(monte vers l'appelant)
`(gdb) down`	(descend vers l'appelé)

Effet :

- sur `print`
  indique de quelles variables locales on parle.
- sur `finish` :
  indique la fonction après laquelle gdb rend la main.

Aucun effet sur l'exécution du programme (`continue`, `next`,...).

Exemple de session

Programme débogué

```
int f(int x)
{
  int y = x+1;
  return y;
}
```

```
int main()
{
  int x;
  x = f(12);
  return 0;
}
```

Exemple de session

```
$ gcc toto.c -Wall -Wextra -g
$ gdb ./a.out
GNU gdb 6.6
Copyright (C) 2006 Free Software Foundation, Inc.
...
```

Exemple de session

Exemple de session (suite)

```
(gdb) break main
Breakpoint 1 at 0x4004b4: file toto.c, line 11.
(gdb) run
Starting program: /home/mine/ATER/2006/prog/a.out

Breakpoint 1, main () at toto.c:11
11          x = f(12);
(gdb) step
f (x=12) at toto.c:3
3           int y = x+1;
(gdb)
4           return y;
(gdb) print y
$1 = 13
```

Exemple de session

Exemple de session (fin)

```
(gdb) up
#1  0x0000000004004be in main () at toto.c:11
11          x = f(12);
(gdb) step
5       }
(gdb) print x
$2 = 12
(gdb) step
main () at toto.c:12
12          return 0;
(gdb) print x
$3 = 13
(gdb) quit
The program is running.  Exit anyway? (y or n) y
$
```

Récapitulatif des commandes gdb

`quit` (ou Ctrl+D)	quitte
`help`	aide intégrée
`run`	commence l'exécution
`continue`	reprend l'exécution
`next` (ou `next` *n*)	exécute une (ou *n*) ligne(s)
`step` (ou `step` *n*)	" (mais s'arrête aux fonctions)
`finish`	exécute jusqu'au retour de la fonction
Ctrl+C	suspend l'exécution
`break` *n° ligne*	place un point d'arrêt
`break` *fonction*	"
`delete`	efface tous les points d'arrêts
`print` *expr*	affiche la valeur d'une expression
`up`	remonte dans la pile d'appels
`down`	descend dans la pile d'appels
`bt` (ou `bt full`)	affiche la pile d'appels

Les caractères

Les caractères

Représentation : par un code de caractère :

- historiquement : sur 7 bits $\Rightarrow$ 128 codes,
- de nos jours : sur **8** bits $\Rightarrow$ **256 codes**.

Les caractères regroupent :

- codes 0 à 127 : standard **ASCII** :
  - 32 à 126 : 95 symboles affichables,
  - 0 à 31 et 127 : 33 caractères de contrôle,
- codes 128 à 255 : symboles affichables étendus :
  de nombreux standard existent :
  - **latin1** : ISO pour l'Europe de l'ouest,
  - **Mac-Roman** : Europe de l'ouest sur les Macintoshs,
  - **UTF-8** : standard universel Unicode,

En C : type `char`.

Codes ASCII

ASCII = *American Standard Code for Information Interchange* (1961).

Codes ASCII affichables

code	caractères															
32	*esp*	!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

Note : voir aussi man ascii.

Caractères de contrôle

Caractères de contrôle courants

code	sigle	signification	notation C
10	LF	saut de ligne	`\n`
13	CR	début de ligne	`\r`
8	BS	efface un caractère	`\b`
7	BEL	bip	`\a`
27	ESC	échappement	`\033`
0	NUL	fin de chaîne C	`\0` (ou `\000`)

Affichage d'un caractère de contrôle : l'effet dépend du terminal !

Constantes caractères en C

Constante `char` **:** un caractère entre ' ' :

- caractère affichable non '
  exemples : `'a'` `')'` `' '`
- séquence spéciale commençant par \
  exemples : `'\n'` `'\''` `'\\'`
- code numérique en octal (base 8) de 1 à 3 chiffres
  exemples : `'\033'` $(= 27)$ `'\0'` `'\000'`
- code numérique en hexadécimal (base 16) de 1 à 2 chiffres
  exemples : `'\x0a'` $(= 10)$ `'\x40'` $(= 65, = 'A')$

Note : on peut aussi utiliser un nombre tout bête
exemple : `char c = 13;`

Arithmétique sur les caractères

Remarques :

- `char` est un type entier,
- l'ASCII est organisé en plages logiques (`A` à `Z`, etc.)

$\Rightarrow$ on peut faire de l'arithmétique sur les caractères.

Exemple : mettre en majuscule

```
char en_majuscule(char c)
{
  if ( c >= 'a' && c <= 'z' ) return c - 'a' + 'A';
  else                        return c;
}
```

Les chaînes de caractères

Représentation des chaînes de caractères

Chaîne (string) = suite de caractères terminée par le caractère `\0`.

En C, pas de type *chaîne* spécifique :

- **déclaration :** on utilise un tableau de `char` : `char[]`,
- **en argument :** on passe un pointeur dans un tableau : `char*`.

Déclaration de chaînes

Déclaration sans initialisation (dangereux car pas de \0)

Exemples

```
char buf[1024];
```

Initialisation caractère par caractère

Exemples

```
char cc[] = { 'H', 'e', 'l', 'l', 'o', '!', 0 };
char toto[4] = { 'H', 'i', '\0' };
char titi[4] = { 'H', 'i' };
```

Rappel des règles sur les initialisations de tableaux :

- avec [], la taille est calculée automatiquement,
- sinon, les initialiseurs manquants sont mis à 0.

Déclaration de chaînes

Initialisation par une chaîne littérale entre " " :

Exemples

```
char cc[] = "Hello!";
char toto[4] = "Hi";
```

Avantages :

- plus léger et naturel,
- le \0 final est automatiquement ajouté,
- on peut utiliser les caractères spéciaux,
  Exemple : `"toto\na \033\a"`.

Rappels sur les pointeurs

`char*` : référence un objet `char` en mémoire.

Rappels : si on se donne `char x, t[5];` alors

- `&x` pointe sur `x`,
- `&t[5]` pointe sur la 6ème case de `t`,
- `t` est équivalent à `&t[0]`,
- ces objets ont tous pour type `char*`.

Déréférencement de pointeur : par `[`i`]` ou `*` (`=[0]`).

Arithmétique de pointeurs : si `p` pointe sur la $i$ème case de `t`,

- `p+`j pointe sur la $i+j$ème case de `t`,
- `p-`j pointe sur la $i-j$ème case de `t`,
- `p++`, `p--` décalent `p` d'une case, etc.

Passage en paramètre

Les chaînes sont passées aux fonctions par référence sous forme de pointeur vers le premier caractère.
Type de l'argument :

- `char*` si la chaîne risque d'être modifiée par la fonction,
- `const char*` si la chaîne n'est pas modifiée.

Exemple

```
int longueur(const char* s)
{
  int i = 0;
  while (s[i]) i++;
  return i;
}
```

Exemple typique : on boucle du premier au dernier caractère (0).

Passage en paramètre

Autre exemple

```
char en_majuscule(char c)
{
  if ( c >= 'a' && c <= 'z' )
    return c - 'a' + 'A';
  else
    return c;
}

void chaine_en_majuscule(char* s)
{
  for ( ; *s; s++ )
    *s = en_majuscule(*s);
}
```

Fonctions standards sur les chaînes

Fonctions standards : documentées dans `man string`.

Quelques fonctions utiles

`strlen`	longueur d'une chaîne
`strcmp`	compare deux chaînes lexicographiquement
`strcpy`	copie de chaîne
`strcat`	concaténation de chaînes
`strncpy` `strncat` `strncmp`	versions limitées à *n* caractères
`strchr`	recherche d'un caractère dans une chaîne
`strstr`	recherche d'une sous-chaîne dans une chaîne

Il faut faire `#include <string.h>`.

Utilisation des fonctions standards

Exemples de prototypes

```
size_t strlen(const char *s);
char *strcpy(char *dest, const char *src);
char *strcat(char *dest, const char *src);
```

Exemple d'utilisation

```
void make_path(const char* dir, const char* file)
{
  char buf[1024]
  if ( strlen(dir) + strlen(file) + 2 > sizeof(buf) ) return;
  strcpy(buf, dir);
  strcat(buf, "/");
  strcat(buf, file);
  ...
```

Chaînes constantes

On peut utiliser une chaîne entre " en dehors des initialisations de tableaux.

Exemples

```
printf("toto\n");
char* s = "toto\n";
```

Effet :

- à la compilation : créé un tableau anonyme bien initialisé, le tableau est global et constant,
  (*e.g.* : `const char anonymeXXX[] = "toto";`)
- à l'exécution : renvoie un pointeur sur ce tableau.
  (*e.g.* : `printf(anonymeXXX); char* s = anonymeXXX;`)

Notes :

- à l'exécution, aucune allocation ni copie de chaîne n'a lieu,
- la tableau ne doit pas être modifié.

Affichage de chaînes

On utilise le format `%s` avec `printf`.

Exemple

```
char nom[] = "personne";
printf("mon nom est %s\n",nom);
```

Attention

Ne jamais faire `printf(s)`...
...et si `s` contient un caractère `%` ?

Extensions :

- `%`*n*`s` : affiche au moins *n* caractères (complète par des blancs),
- `%.`*m*`s` : affiche au plus *m* caractères.

Erreurs classiques

Cause principale

Tout accès en dehors des bornes d'un tableau provoque une erreur non déterministe.

Exemples d'erreurs

```
char buf[4];
char c[2] = { 'a', 'b' };
char* s;

strcpy( buf, s );        /* s pointe dans l'espace */
strcpy( buf, "abcd" );   /* buf trop petit */
strcpy( buf, c );        /* c non terminé par \0 */
s = "toto";
s[1] = 'a';    /* s pointe sur dans un tableau constant */
```

Le latin1

latin1 (ou **ISO 8859-1**) : l'ASCII étendu pour l'Europe de l'ouest.

- codes 128 à 159 : codes de contrôle,
- codes 160 à 255 : symboles, lettres accentuées

Codes latin1 étendus

160		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬		®	¯
176	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
192	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
208	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
240	∂	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

latin9 (ou **ISO 8859-15**) : ajout de Œ, œ, Ÿ, Š, š, Ž, ž, €.

Voir : `man latin1`, `man latin9`.

Utiliser le latin1

Pour utiliser des caractères latin1 étendus dans " et ',
il faut s'assurer que :

- le source `.c` est sauvegardé au format latin1,
  (Options -> Mule dans Emacs)
- le terminal est configuré en latin1.

Exemple : le retour des majuscules

```
char en_majuscule(char c)
{
  if ( c >= 'a' && c <= 'z' ||
       c >= 'à' && c <= 'ý' && c != 247 && c != 248 )
      return c - 32;
  return c;
}
```

Unicode et UTF-8

Unicode (ou **ISO 10646**) : standard universel (**UCS**), 1M codes.

UTF-8 : encodage de taille variable pour Unicode :

- codes 0 à 127 : sur 1 octet, identique à l'ASCII (`0xxxxxxx`)
- codes 128 à 2047 : sur 2 octets (`110xxxxx 10xxxxxx`) (11 bits)
- codes 2048 à 65535 : sur 3 octets (16 bits)
  (`1110xxxx 10xxxxxx 10xxxxxx`)
- à partir de 65536 : sur 4 octets (21 bits)
  (`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`)

Avantages : entre autres :

- compatible avec l'**ASCII**,
- compatible avec le C (pas d'ajout de `\0`),
- codage non ambigu, parcours de droite à gauche possible.

Unicode et UTF-8

Pour utiliser l'UTF-8 en C, il faut s'assurer que :

- le source `.c` est sauvegardé au format UTF-8,
- le terminal est configuré en UTF-8.

Attention

Un "caractère" unicode peut correspondre à plusieurs `char` en C.

Conséquences :

- `'à'` n'a pas de sens (`à` occupe deux `char` en UTF-8),
- `strlen` et `strchr` ne fonctionnent pas comme on s'y attend,
- `strcat` et `strcpy` fonctionnent,
  mais attention à évaluer la taille des tableaux en `char` !