

Cours de Réseau et communication Unix n°9

Edouard THIEL

Faculté des Sciences

Université d'Aix-Marseille (AMU)

Septembre 2016

Les transparents de ce cours sont téléchargeables ici :

<http://pageperso.lif.univ-mrs.fr/~edouard.thiel/ens/rezo/>

Lien court : <http://j.mp/rezocom>

Plan du cours n°9

1. Outils binaires pour l'analyse de trames
2. Analyse de chaînes de caractères

1 - Outils binaires pour l'analyse de trames

Opérateurs binaires, macros et champs de bits.

Représentation en bases

Le C standardise 3 représentations des entiers :

décimale `i = 42;`

octale `i = 052;`


hexadécimale `i = 0x2a;`

Le C autorise des extensions de formats :

binaire `i = 0b101010;` extension gcc

Affichage en bases

Formats d'affichages de printf :

<code>%d</code>	décimal signé
<code>%u</code>	décimal non signé
<code>0%o</code> ou <code> %#o</code>	octal non signé
<code>0x%x</code> ou <code> %#x</code>	hexadécimal non signé
	pas d'affichage standard en base 2

Modificateurs :

<code>l</code>	long int
<code>ll</code>	long long int
<code>h</code>	short int
<code>hh</code>	short short int (\simeq char)
<code>z</code>	size_t, ssize_t, off_t (fstat)

Exemple : `size_t len = strlen (argv[1]);`
`printf ("Taille = %zu = %#zx\n", len, len);`

Opérateurs du C

- Opérateurs binaires classiques :

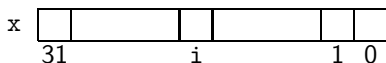
et	1001	&	0101	=	0001
ou	1001		0101	=	1101
xor	1001	^	0101	=	1100
non		~	0101	=	1010

- Décalages :

décalage à gauche	1001	<< 1	=	10010
	1001	<< 2	=	100100
	x	<< n	=	$x 2^n$
décalage à droite	1001	>> 1	=	100
	1001	>> 4	=	0
	x	>> n	=	$x 2^{-n}$

Utilisation

```
#include <stdint.h>
uint32_t x;
```

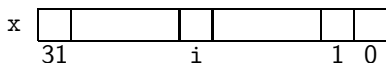


- Mettre tous les bits de x à 0 : ?
- Allumer le bit n° i et mettre les autres à 0 :
- Allumer le bit n° i sans modifier les autres :
- Éteindre le bit n° i :
- Tester si le bit n° i est allumé :

Voir aussi : Bit Twiddling Hacks by S.E. Anderson

Utilisation

```
#include <stdint.h>
uint32_t x;
```

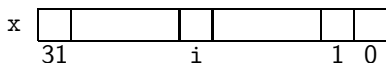


- Mettre tous les bits de x à 0 : $x = 0;$
- Allumer le bit n° i et mettre les autres à 0 :
- Allumer le bit n° i sans modifier les autres :
- Éteindre le bit n° i :
- Tester si le bit n° i est allumé :

Voir aussi : Bit Twiddling Hacks by S.E. Anderson

Utilisation

```
#include <stdint.h>
uint32_t x;
```

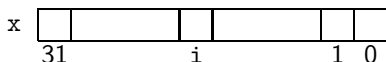


- Mettre tous les bits de x à 0 : $x = 0;$
- Allumer le bit n° i et mettre les autres à 0 : $x = 1 \ll i;$
- Allumer le bit n° i sans modifier les autres :
- Éteindre le bit n° i :
- Tester si le bit n° i est allumé :

Voir aussi : [Bit Twiddling Hacks](#) by S.E. Anderson

Utilisation

```
#include <stdint.h>
uint32_t x;
```

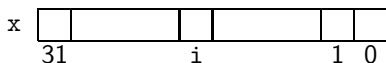


- Mettre tous les bits de x à 0 : $x = 0;$
- Allumer le bit n° i et mettre les autres à 0 : $x = 1 \ll i;$
- Allumer le bit n° i sans modifier les autres : $x |= 1 \ll i;$
- Éteindre le bit n° i :
- Tester si le bit n° i est allumé :

Voir aussi : [Bit Twiddling Hacks](#) by S.E. Anderson

Utilisation

```
#include <stdint.h>
uint32_t x;
```

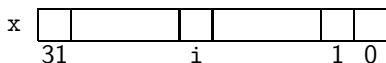


- Mettre tous les bits de x à 0 : $x = 0;$
- Allumer le bit n° i et mettre les autres à 0 : $x = 1 \ll i;$
- Allumer le bit n° i sans modifier les autres : $x |= 1 \ll i;$
- Éteindre le bit n° i : $x \&= \sim(1 \ll i);$
- Tester si le bit n° i est allumé :

Voir aussi : [Bit Twiddling Hacks](#) by S.E. Anderson

Utilisation

```
#include <stdint.h>
uint32_t x;
```



- Mettre tous les bits de x à 0 : $x = 0;$
- Allumer le bit n° i et mettre les autres à 0 : $x = 1 \ll i;$
- Allumer le bit n° i sans modifier les autres : $x |= 1 \ll i;$
- Éteindre le bit n° i : $x \&= \sim(1 \ll i);$
- Tester si le bit n° i est allumé : $x \& 1 \ll i$

Voir aussi : [Bit Twiddling Hacks](#) by S.E. Anderson

Exemple : affichage binaire

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

void printbits (int v) {
    for (int i = (sizeof(v)*8)-1; i >= 0; i--) {
        putchar('0' + ((v >> i) & 1));
        if (i % 8 == 0) putchar (' ');
    }
}

int main (int argc, char *argv[]) {
    if (argc-1 != 1) return 1;
    printbits (atoi(argv[1])); puts (""); return 0;
}
```

```
<> ./printbits 519
00000000 00000000 00000010 00000111
<> ./printbits -519
11111111 11111111 11111101 11111001
```

fd_set, un peu nettoyé (1/2)

Dans <bits/typesizes.h> :

```
/* Number of descriptors that can fit in an 'fd_set'. */  
#define FD_SETSIZE          1024
```

Dans <sys/select.h> :

```
typedef long int fd_mask;  
#define NFDBITS      (8 * (int) sizeof (fd_mask))  
  
typedef struct {  
    fd_mask fds_bits[FD_SETSIZE / NFDBITS];  
} fd_set;  
  
# define FDS_BITS(set) ((set)->fds_bits)
```

fd_set, un peu nettoyé (2/2)

Dans <sys/select.h> :

```
#define FD_ELT(d)    ((d) / NFDBITS)
#define FD_MASK(d)  ((fd_mask) 1 << ((d) % NFDBITS))
```

Dans <bits/select.h> :

```
# define FD_ZERO(set) \
do {
    unsigned int i;
    fd_set *arr = (set);
    for (i = 0; i < sizeof (fd_set) / sizeof (fd_mask); ++i) \
        FDS_BITS(arr)[i] = 0;
} while (0)

#define FD_SET(d, set) \
    ((void) (FDS_BITS(set)[FD_ELT(d)] |= FD_MASK (d)))
#define FD_CLR(d, set) \
    ((void) (FDS_BITS(set)[FD_ELT(d)] &= ~FD_MASK (d)))
#define FD_ISSET(d, set) \
    ((FDS_BITS(set)[FD_ELT(d)] & FD_MASK (d)) != 0)
```

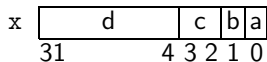
Champs de bits (bitfields)

Permet de grouper des bits de façon plus agréable à manipuler que les opérateurs binaires.

Déclaration : `type_entier : nombre_de_bits;`

Le type de base est entier, signé ou non signé, de taille \leq taille mot machine.

```
struct { unsigned a: 1;
         signed  b: 1;
         unsigned c: 2;
         unsigned d: 28; } x;
```



Se manipulent comme les entiers :

```
x.a = x.b;
if (x.c == 3) ..
x.d += 7;
```


Propriétés des bitfields

- ▶ Ne peuvent être déclarés que dans des structs ou unions.
- ▶ Ils n'ont pas d'adresse : pas de `&`, pas de tableau.
- ▶ L'ordre des bits est implémentation-dépendant (endianness).
- ▶ Il peut y avoir des alignements mémoire.
- ▶ Un bitfield signé de largeur 1 vaut 0 ou -1.

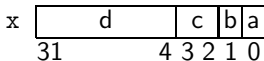
Exemple de bitfield

```
#include <stdio.h>
int main ()
{
    union {
        struct { unsigned a: 1;
                 signed  b: 1;
                 unsigned c: 2;
                 unsigned d: 28; } x;

        int y;
    } u;

    for (u.y = 0; u.y < 18; u.y++)
        printf ("y = %2d    a = %d    b = %2d    c = %d    d = %d\n",
                u.y, u.x.a, u.x.b, u.x.c, u.x.d);

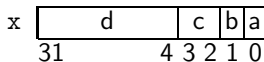
    return 0;
}
```



←
LSB First
(little endian)

Trace

y = 0	a = 0	b = 0	c = 0	d = 0
y = 1	a = 1	b = 0	c = 0	d = 0
y = 2	a = 0	b = -1	c = 0	d = 0
y = 3	a = 1	b = -1	c = 0	d = 0
y = 4	a = 0	b = 0	c = 1	d = 0
y = 5	a = 1	b = 0	c = 1	d = 0
y = 6	a = 0	b = -1	c = 1	d = 0
y = 7	a = 1	b = -1	c = 1	d = 0
y = 8	a = 0	b = 0	c = 2	d = 0
y = 9	a = 1	b = 0	c = 2	d = 0
y = 10	a = 0	b = -1	c = 2	d = 0
y = 11	a = 1	b = -1	c = 2	d = 0
y = 12	a = 0	b = 0	c = 3	d = 0
y = 13	a = 1	b = 0	c = 3	d = 0
y = 14	a = 0	b = -1	c = 3	d = 0
y = 15	a = 1	b = -1	c = 3	d = 0
y = 16	a = 0	b = 0	c = 0	d = 1
y = 17	a = 1	b = 0	c = 0	d = 1



Exemple réel : entête TCP

Défini dans /usr/include/netinet/tcp.h

```
struct tcphdr
{
    u_int16_t source;
    u_int16_t dest;
    u_int32_t seq;
    u_int32_t ack_seq;
#   if __BYTE_ORDER == \
        __LITTLE_ENDIAN
    u_int16_t res1:4;
    u_int16_t doff:4;
    u_int16_t fin:1;
    u_int16_t syn:1;
    u_int16_t rst:1;
    u_int16_t psh:1;
    u_int16_t ack:1;
    u_int16_t urg:1;
    u_int16_t res2:2;
#   elif __BYTE_ORDER == \
        __BIG_ENDIAN
    u_int16_t doff:4;
    u_int16_t res1:4;
    u_int16_t res2:2;
    u_int16_t urg:1;
    u_int16_t ack:1;
    u_int16_t psh:1;
    u_int16_t rst:1;
    u_int16_t syn:1;
    u_int16_t fin:1;
#   else
#       error "Adjust <bits/endian.h>"
#   endif
    u_int16_t window;
    u_int16_t check;
    u_int16_t urg_ptr;
};
```

Où est défini le boutisme (endianness) ?

Linux : dans /usr/include/endian.h

```
#define __LITTLE_ENDIAN 1234
#define __BIG_ENDIAN    4321
#define __PDP_ENDIAN    3412

#include <bits/endian.h>
```

Et (pour Linux x86_64) dans :

/usr/include/x86_64-linux-gnu/bits/endian.h

```
/* x86_64 is little-endian. */

#define __BYTE_ORDER __LITTLE_ENDIAN
```

Où sont définies les conversions d'ordre réseau ?

Dans /usr/include/netinet/in.h

```
# if __BYTE_ORDER == __BIG_ENDIAN
# define ntohl(x) (x)
# define ntohs(x) (x)
# define htonl(x) (x)
# define htons(x) (x)
# else
#   if __BYTE_ORDER == __LITTLE_ENDIAN
#     define ntohl(x) __bswap_32 (x)
#     define ntohs(x) __bswap_16 (x)
#     define htonl(x) __bswap_32 (x)
#     define htons(x) __bswap_16 (x)
#   endif
# endif
```

Où sont définies les fonctions de byteswap ?

Linux x86_64 : dans

`/usr/include/x86_64-linux-gnu/bits/byteswap.h`

Implémentation en assembleur, ou par macros :

```
#define __bswap_constant_16(x) \  
    ((unsigned short int) ( ((x) >> 8) & 0xff) \  
        | (((x) & 0xff) << 8) ))
```

```
#define __bswap_constant_32(x) \  
    ( (((x) & 0xff000000) >> 24) | (((x) & 0x00ff0000) >> 8) \  
    | (((x) & 0x0000ff00) << 8) | (((x) & 0x000000ff) << 24) )
```

Byteswap en 64 bits

Également dans `byteswap.h` :

```
# define __bswap_constant_64(x) \  
  ( ((x) & 0xff00000000000000ull) >> 56) \  
  | ((x) & 0x00ff000000000000ull) >> 40) \  
  | ((x) & 0x0000ff0000000000ull) >> 24) \  
  | ((x) & 0x000000ff00000000ull) >> 8) \  
  | ((x) & 0x00000000ff000000ull) << 8) \  
  | ((x) & 0x0000000000ff0000ull) << 24) \  
  | ((x) & 0x000000000000ff00ull) << 40) \  
  | ((x) & 0x00000000000000ffull) << 56) )
```


2 - Analyse de chaînes de caractères

Outils standards

Pour encoder

- Écrire dans une chaîne

```
#include <stdio.h>
#define CAPACITE 2048
char s[CAPACITE];

sprintf (s, format, ...);
snprintf (s, CAPACITE, format, ...);
```

Renvoient le nombre de caractères écrits dans la chaîne (sans '\0')

format : man 3 sprintf

- Écrire en concaténant

```
int pos = snprintf (s, CAPACITE, format1, ...);
pos += snprintf (s+pos, CAPACITE-pos, format2, ...);
```

Pour décoder

Un outil plus puissant qu'il n'en a l'air : `sscanf`

```
#include <stdio.h>
#define CAPACITE 2048
char s[CAPACITE];

int k = sscanf (s, format, ...);
```

Le format a un sens différent pour `sscanf` :

- ▶ Blanc = espace, tabulation, retour chariot : correspondent à 0, 1 ou plusieurs blancs dans `s`.
- ▶ Toutes les conversions commencent par %
- ▶ Les autres caractères sont des littéraux.
- ▶ `sscanf` s'arrête dès qu'un caractère ne correspond pas au format ou qu'une conversion est impossible.

Conversions (1)

<code>%%</code>	% littéral
<code>%d, %f</code>	entier, flottant
<code>%s</code>	Séquence de caractères non blanc (les blancs situés devant sont supprimés)
<code>%c</code>	1 caractère, y compris blanc
<code>%α...</code>	de taille maximale α caractères (α entier)
<code>%*...</code>	ne pas stocker le résultat <code>sscanf ("10 20", "%*d %d", &x);</code> $\rightarrow x = 20$
<code>%n</code>	Position courante (nombre de caractères consommés) <code>char *s = "789 65";</code> <code>sscanf (s, "%d%n", &x, &p);</code> $\rightarrow x = 789, p = 3$ <code>sscanf (s+p, "%d", &y);</code> $\rightarrow y = 65$

Conversions (2)

- `%[. .]` Séquence non vide de caractères appartenant ou
- `%[^. .]` n'appartenant pas à un ensemble
- `%* α [. .]` variantes : de taille maximale α , sans mémoriser

Cas particuliers :

- `%[] . .`, `%[^] . .` pour inclure ou exclure ']' ,
- `%[. ^ .]`, `%[^ ^ . .]` pour inclure ou exclure '^'
- `%[a-z]`, `%[^a-z]` pour inclure ou exclure un intervalle
- `%[. . -]`, `%[^. . -]` pour inclure ou exclure '-'

Valeur retournée

`sscanf` renvoie le nombre de conversions mémorisées

Les conversions `%*..` et `%n` ne sont pas comptées

Tester le retour de `sscanf` est obligatoire : sans cela on ne discerne pas les variables indéfinies.

Exemple : propriété dans un entête HTTP :

```
char *s = "Accept-Encoding: gzip, deflate";
char key[80], value[256];

int k = sscanf (s, "%79[^: ]: %255[^\n]", key, value);

if (k != 2) printf ("mauvaise syntaxe\n");
```

Exemple : conversion d'adresse IP

chaîne de caractères "129.43.02.01" → adresse uint32_t.

```
char ip[4];
struct sockaddr_in a;

if (sscanf(chaine, "%hhu.%hhu.%hhu.%hhu",
           ip, ip+1, ip+2, ip+3) == 4)           // MSB first
    memcpy(&a.sin_addr.s_addr, ip, 4);
```

... ou appeler `inet_addr()`

Autre outil : fonctions regex

Fonctions POSIX.1-2001 de la libc :

```
#include <sys/types.h>
#include <regex.h>

int regcomp (regex_t *preg, const char *regex, int cflags);

int regexec (const regex_t *preg, const char *string,
             size_t nmatch, regmatch_t pmatch[], int eflags);

size_t regerror (int errcode, const regex_t *preg,
                char *errbuf, size_t errbuf_size);

void regfree (regex_t *preg);
```


Fonction regcomp

```
int regcomp (regex_t *preg, const char *regex,  
             int cflags);
```

regex expression régulière

preg expression compilée

cflags REG_EXTENDED : syntaxe POSIX étendue
 REG_ICASE : ignorer casse

Renvoie 0 succès, sinon code d'erreur.

Fonction regexec

```
int regexec (const regex_t *preg, const char *string,  
             size_t nmatch, regmatch_t pmatch[],  
             int eflags);
```

preg expression compilée par regcomp

string chaîne à analyser

nmatch nombre d'éléments à trouver

pmatch tableaux de nmatch éléments

eflags voir man

Renvoie 0 succès, sinon REG_NOMATCH.

Élément trouvé : index [rm_so,rm_eo[dans string :

```
typedef struct {  
    regoff_t rm_so;  
    regoff_t rm_eo;  
} regmatch_t;
```

Fonctions regerror et regfree

```
size_t regerror (int errcode, const regex_t *preg,  
                char *errbuf, size_t errbuf_size);
```

Transforme un code d'erreur en message.

errcode	résultat de regcomp ou regexec
preg	expression compilée par regcomp
errbuf	buffer qui contiendra le message
errbuf_size	taille du buffer

```
void regfree (regex_t *preg);
```

preg	expression compilée par regcomp
------	---------------------------------

Exemple

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

int main (int argc, char *argv[]) {
    if (argc-1 != 2) { fprintf (stderr,
        "Usage: %s regexp chaîne\n", argv[0]); return 1; }
    regex_t regex; int k; char *re = argv[1], *ch = argv[2];

    k = regcomp (&regex, re, REG_EXTENDED);
    if (k != 0) {
        char msg[200]; regerror (k, &regex, msg, sizeof msg);
        fprintf (stderr, "Erreur dans regex: %s\n", msg);
    } else {
        k = regexec (&regex, ch, 0, NULL, 0);
        if (k == 0) puts("Correspond");
        else puts("Ne correspond pas");
    }
    regfree (&regex); return !!k;
}
```

Trace

```
$ gcc -W -Wall -std=c99 regex1.c -o regex1
$ ./regex1 "ch(at|ien)" "chat"
Erreur dans regex: Unmatched ( or \
$ ./regex1 "ch(at|ien)" "chat"
Correspond
$ ./regex1 "ch(at|ien)" "chute"
Ne correspond pas
$ ./regex1 "^a[[:digit:]]E+bc" "a342E+5bc"
Correspond
```

Syntaxe : [voir expression rationnelles POSIX](#)