Programmation C avancée

Manipulation de bits

Licence informatique 3ème année

Université de Marne-la-Vallée

Le binaire

- représentation en base 2 :

128	64	32	16	8	4	2	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	1	0	0	1	0	0

$= 0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$

$= 64 + 32 + 4$

$= 100$

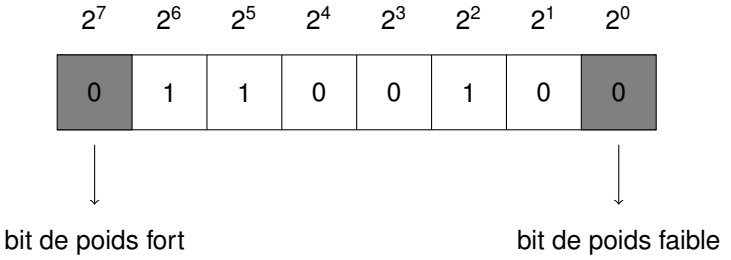
Le binaire

- premières puissances de 2 à connaître par cœur
- pratique pour détecter les valeurs spéciales (1023, 4097, ...)

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7
1	2	4	8	16	32	64	128
2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}
256	512	1024	2048	4096	8192	16384	32768

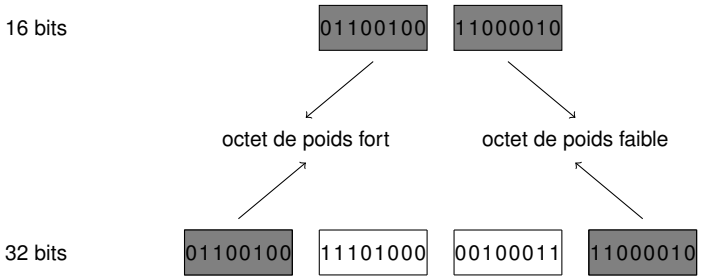
Poids fort / poids faible

- poids fort = grandes puissances de 2
- poids faible = petites puissances de 2



Poids fort / poids faible

- idem pour les types sur plusieurs octets



Entiers signés

- codage qui dépend de l'implémentation !
- bit de poids fort = signe (peu utilisé) :
  `00000010` = +2 en décimal
  `10000010` = -2 en décimal
- problèmes :
  - 0 a deux représentations (`00000000` et `10000000`)
  - l'addition ne marche pas :
  `10000100` + `00000011` = `10000111` (-4 + 3 = -7)

Complément à deux

- on prend la valeur absolue
- on inverse les bits
- on ajoute 1 en ignorant les dépassements
- exemple : -6 codé sur un octet
  6 = 00000110 ⇒ ~6 = 11111001
  ~6+1 = 11111010 ⇒ 250

```
1 int main(int argc, char* argv[]){
2   printf("%d\n", (unsigned char)(-6));
3   return 0;
4 }
```

affiche bien 250 !

Complément à deux

- l'addition fonctionne :
  `11111100` + `00000011` = `11111111` (-4 + 3 = -1)
- écriture unique de zéro : 0 = -0
- -(-x) = x

Complément à deux

- lors d'une conversion de type, une transformation est appliquée

```
1 int main(int argc, char* argv[]){
2   char c=-26;
3   int i=c;
4   unsigned char uc=c;
5   unsigned int ui=c;
6   printf("%d %d %u %u\n", c, i, uc, ui);
7   return 0;
8 }
```

donne

```
nborie@perceval:~> ./test
-26 -26 230 4294967270
```

Pas de problème lorsque les valeurs converties restent dans la plage de valeur du type cible...

Opérateurs bit à bit

- ne fonctionnent que sur les types entiers
- à ne pas confondre avec les opérateurs logiques || et &&

Bit 1	0	0	1	1
Bit 2	0	1	0	1
& (et)	0	0	0	1
| (ou)	0	1	1	1
^ (ou exclusif)	0	1	1	0

Opérateur &

a	1	0	1	1	0	1	1	1
b	1	1	0	1	0	1	0	1
a&b	1	0	0	1	0	1	0	1

Opérateur |

a	1	0	1	1	0	1	1	1
b	1	1	0	1	0	1	0	1
a|b	1	1	1	1	0	1	1	1

Opérateur ^

a	1	0	1	1	0	1	1	1
b	1	1	0	1	0	1	0	1
a^b	0	1	1	0	0	0	1	0

utile en cryptographie car réversible !

a^b	0	1	1	0	0	0	1	0
a	1	0	1	1	0	1	1	1
b=(a^b)^a	1	1	0	1	0	1	0	1

XOR logique

- Il n'y a pas de xor logique en C : `a^^b` :(
- solution possible : transformer les conditions logiques en entier et utiliser un xor bit à bit

```
1  #define XOR(cond1, cond2) ((cond1) != 0)^((cond2) != 0)
2
3  int main(int argc, char* argv[]){
4    int a=atoi(argv[1]);
5    int b=atoi(argv[2]);
6    if (XOR(a%2, b%2))
7      printf("%d ou %d est divisible par 2 mais pas les deux\n", a, b);
8    else
9      printf("%d et %d sont pairs ou impairs tous les deux\n", a, b);
10   return 0;
11 }
```

se comporte ainsi

```
nborie@perceval:~> ./test 2 2
2 et 2 sont pairs ou impairs tous les deux
nborie@perceval:~> ./test 2 3
2 ou 3 est divisible par 2 mais pas les deux
nborie@perceval:~> ./test 5 4
5 ou 4 est divisible par 2 mais pas les deux
nborie@perceval:~> ./test 5 17
5 et 17 sont pairs ou impairs tous les deux
```

Opérateur unaire ~

- inversion des bits (complément à un)

a	1	0	1	1	0	1	0	1
~a	0	1	0	0	1	0	1	0

Décalage à gauche

- `x<<y` : décale les bits de `x` de `y` bits vers la gauche (`x` n'est pas modifié)
- remplissage avec des zéros
- les bits de poids forts sont perdus

181	1	0	1	1	0	1	0	1
181<<1	1	0	1	1	0	1	0	1	0
(181*2)-256 = 106	0	1	1	0	1	0	1	0

Décalage à gauche

- 181 est trop petit par rapport à la taille d'un `int` pour que la perte du bit de poids fort soit sensible, mais pas pour un `char`

```
1 int main(int argc, char* argv[]){
2   unsigned char c=181;
3   int i=c;
4   c = c<<1;
5   i = i<<1;
6   printf("%d %d\n", c, i);
7   return 0;
8 }
```

donne

```
nborie@perceval:~> ./test
106 362
```

Décalage à droite

- `x>>y` : décale les bits de `x` de `y` bits vers la gauche (`x` n'est pas modifié)
- remplissage avec :

  - des 0 si type non signé (ou type signé mais de valeur positive), version portable

  - dépend de l'implémentation sinon!, version non portable
- jamais de décalage sur les entiers signés sans une très bonne raison
- les bits de poids faibles sont perdus

Décalage circulaire à gauche

- pour décaler `x` de n (<SIZE) bits :
  - copier `x` dans `tmp`
  - décaler `tmp` de SIZE-n bits vers la droite
  - décaler `x` de n bits vers la gauche
  - faire `x` OU `tmp`

```
1 void shift_circular_left(unsigned char *c, unsigned int n){
2   n=n%CHAR_BIT; /* modulo 8 probablement */
3   unsigned char tmp = (*c)>>(CHAR_BIT-n);
4   (*c)=((*c)<<n)|tmp;
5 }
```

Avec un affichage binaire de 80 (en `unsigned char`) et les 7 décalages:

```
nborie@perceval:~> ./test
0 1 0 1 0 0 0 0
1 0 1 0 0 0 0 0
0 1 0 0 0 0 0 1
1 0 0 0 0 0 1 0
0 0 0 0 0 1 0 1
0 0 0 0 1 0 1 0
0 0 0 1 0 1 0 0
0 0 1 0 1 0 0 0
```

Décalage circulaire à droite

- pour décaler circulairement `x` de n (<SIZE) bits :

  - décaler circulairement `x` de SIZE-n bits à gauche

```
1 void shift_circular_right(unsigned char *c, unsigned int n){
2   shift_circular_left(c, (CHAR_BIT - n) % CHAR_BIT);
3 }
```

- travailler sur du non-signé ! (N. Borie : Vous allez alors au devant de graves déconvenues...)

  Même programme qu'avant avec des `char` :

```
nborie@perceval:~> ./test
0 1 0 1 0 0 0 0
1 0 1 0 0 0 0 0
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
```

  Aie, le complément à deux a introduit des 1 !

Un bit tout seul

- $n^{\text{ème}}$ bit à 1 (en partant de zéro) et les autres à 0 = `1<<n` (c'est à dire la valeur 2^n).
- bit 0 : `00000001` = `1<<0` = 2^0
- bit 1 : `00000010` = `1<<1` = 2^1
- ...
- bit 7 : `10000000` = `1<<7` = 2^7

Tester le $n^{\text{ème}}$ bit

- ET bit à bit entre la valeur à tester et `1<<n`
- application : affichage en binaire

```
1  void printf_bin(char a){
2    int i;
3    for (i=CHAR_BIT-1 ; i>=0 ; i--)
4      printf("%c", (a&(1<<i))?'1':'0');
5    printf("\n");
6  }
7
8  int main(int argc, char* argv[]){
9    print_bin(79);
10   return 0;
11 }
```

affiche bien :

```
01001111
```

Mettre à 1 le $n^{\text{ème}}$ bit

- OU inclusif entre la valeur à modifier et `1<<n`

```
1  void set_bit(char* c, int bit){
2    *c=(*c)|(1<<bit);
3  }
4
5  int main(int argc, char* argv[]){
6    char c=79;
7    printf("  "); print_bin(c);
8    printf("| "); print_bin(1<<5);
9    set_bin(&c, 5);
10   printf("= "); print_bin(c);
11   return 0;
12 }
```

```
  01001111
| 00100000
= 01101111
```

Mettre le $n^{\text{ème}}$ bit à 0

- ET bit à bit entre la valeur à tester et $\sim (1 << n)$

```
1  void unset_bit(char *c, int bit){
2    *c=(*c)&~(1<<bit);
3  }
4
5  int main(int argc, char* argv[]){
6    char c=79;
7    printf("  "); print_bin(c);
8    printf("| "); print_bin(~(1<<2));
9    unset_bin(&c, 2);
10   printf("= "); print_bin(c);
11   return 0;
12 }
```

```
  01001111
& 11111011
= 01001011
```

Affecter le $n^{ème}$ bit

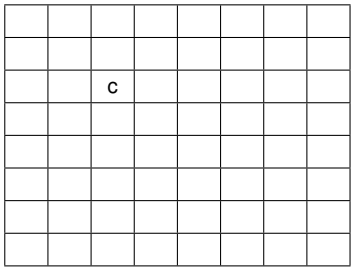
- méthode 1 : mettre à 0 ou 1 en fonction de la valeur
- méthode 2 : mise à 0, puis OU bit à bit avec 0 ou 1 décalé de n bits vers la gauche

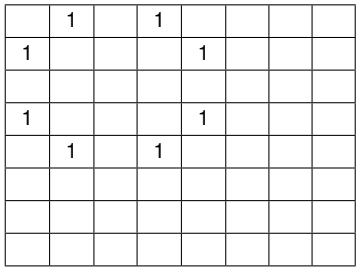
```
1  void set(char *c, int bit, int value){
2    if (value) set_bit(c, bit);
3    else unset_bit(c, bit);
4  }
5
6  void set2(char *c, int bit, int value){
7    unset_bit(c, bit);
8    *c=(*c)|(value!=0)<<bit;
9  }
```

Masques

- stocker plusieurs informations sur un entier
- exemple : les échiquiers de S.Giraudo (expert en échec et programmation même s'il prétend le contraire)

Les cases menacées par une pièce du jeu d'échec peuvent être représentées par un échiquier rempli de booléens.





Théorie des bitboard...

Drapeaux

- désigner les bits par des constantes qui servent de masques

```
1  #define INITIAL 1
2  #define FINAL 2
3  #define ACCESSIBLE 4
4  #define COACCESSIBLE 8
5
6  void set_flag(char *c, int flag){
7    *c=(*c)|flag;
8  }
9
10 void unset_flag(char *c, int flag){
11   *c=(*c)&~flag;
12 }
13
14 int is_set_flag(char c, int flag){
15   return c&flag;
16 }
```

Champs de bits

- même esprit dans les champs de bits que dans les drapeaux
- plus pratique que les masques
- c'est le langage qui gère les positions au lieu de tout programmer à la main
- les variables de champs de bits posent des problèmes de portabilité (on les lits dans quel sens ? où les bits sont-ils positionnés en mémoire ?)

Champs de bits

- peu normés:
  - souvent par bloc d'1 `int`, mais pas forcément
  - champs contigus si possible, mais pas forcément
  - ordre des champs respecté
  - ordre des bits d'un champ non normé
  - implémentation des nombres signés non normée
- champs non adressables

Champs de bits

- le compilateur gère les débordements de champs
- on peut utiliser des nombres signés et des champs anonymes:

```
1  struct foo{
2    signed char x: 2;
3    unsigned char : 1; /* pour l'alignement */
4    signed char y: 3;
5    signed char z: 2;
6  }
```

Champs de bits

- OK pour simplifier les calculs en interne
- mais ne pas sauver directement un champ de bits en binaire !

```
1  struct color{
2    unsigned int alpha:2, red:10,
3                 green:10, blue:10;
4  };
5
6  /* Safe saving */
7  void write_color(struct color* c, FILE* f){
8    unsigned int i;
9    i=(c->alpha<<30) | (c->red<<20)
10   | (c->green<<10) | c->blue;
11   fwrite(&i, sizeof(unsigned int), 1, f);
12 }
```

Portabilités des types

- pour des opérations bit à bit portables, il faut être sûr de la taille des types entiers
- ⇒ types absolus définis dans `stdint.h`

```
1  int main(int argc, char* argv[]){
2    printf("int8_t: %lu\nuint8_t: %lu\n",
3           sizeof(int8_t), sizeof(uint8_t));
4    printf("int16_t: %lu\nuint16_t: %lu\n",
5           sizeof(int16_t), sizeof(uint16_t));
6    printf("int32_t: %lu\nuint32_t: %lu\n",
7           sizeof(int32_t), sizeof(uint32_t));
8    printf("int64_t: %lu\nuint64_t: %lu\n",
9           sizeof(int64_t), sizeof(uint64_t));
10   return 0;
11 }
```

```
nborie@perceval:~> ./test
int8_t: 1
uint8_t: 1
int16_t: 2
uint16_t: 2
int32_t: 4
uint32_t: 4
int64_t: 8
uint64_t: 8
```

Tableaux de bits

- pratiques pour gérer beaucoup d'informations binaires
- pour n informations, il faut $n/8$ octets, +1 si n n'est pas multiple de 8 (`CHAR_BIT` si on est parano)
  - taille = n/8 + (n%8 != 0)
- accès au $n^{\text{ème}}$ élément:
  - octet = n/8
  - bit = n%8

Représentation d'ensembles

- implémentation efficace d'ensemble
- A inter B = `A & B`
- A union B = `A | B`
- négation A = `~A`
- A inclus dans B = `(A & B) == A`
- A - B = A inter (négation B) = `A & ~ B`

Représentation d'ensembles

- seule condition : savoir énumérer les éléments (pas toujours simple!)
- utiliser une fonction qui projette les éléments sur l'ensemble [0, n-1]

```
1 /* Projette les intervalles [a-z], [A-Z] et [0-9] */
2 /* sur l'ensemble [0-61], retourne -1 si on n'est */
3 /* pas dans la plage des lettres ou chiffres */
4 int get_index(char c){
5   if (c>='a' && c<='z') return c-'a';
6   if (c>='A' && c<='Z') return 26+c-'A';
7   if (c>='0' && c<='9') return 52+c-'0';
8   return -1;
9 }
```

Buffer de bits

```
1 struct bit_buffer{
2   unsigned char c;
3   int n; /* position du prochain bit libre */
4 };
```

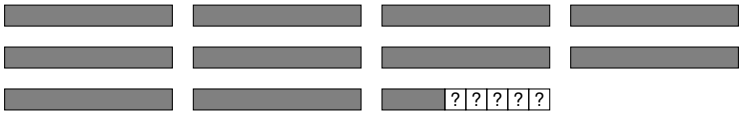
- à sauver et réinitialiser quand on écrit 8 bits
- ordre de remplissage des bits au choix (poids fort vers poids faible ou l'inverse)

Buffer de bits

```
1  /* Adds 1 or 0 to the given bit buffer, depending on
2     the given value. If the bit buffer is filled up, it
3     is flushed into the given file and reseted. Note
4     that the buffer must have been initialized properly
5     before the first call to this function. */
6  void write_bit(struct bit_buffer* b, int value, FILE* f){
7    b->c|=(value != 0)<<b->n;
8    b->n--;
9    if (b->n==-1){
10     fputc(b->c, f);
11     b->c=0;
12     b->n=7;
13   }
14 }
```

Sauver des bits dans un fichier

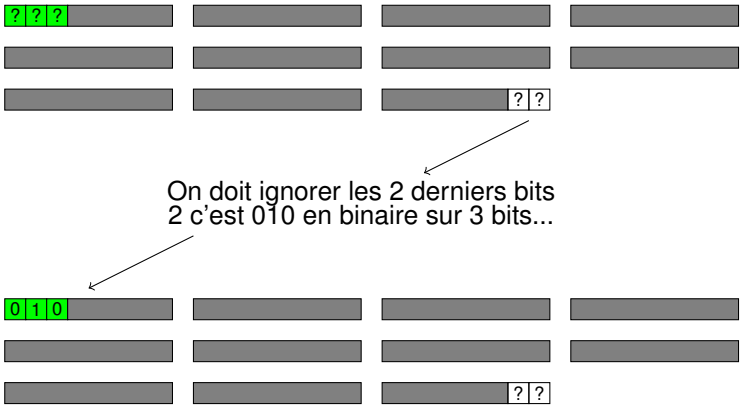
- utiliser un buffer de bits
- que faire si le dernier octet n'est pas rempli ?



Sauver des bits dans un fichier

- sur le dernier octet on peut ignorer de 0 (octet plein) à 7 bits (un seul bit utilisé)
- pour coder 8 valeurs différentes, il faut 3 bits
- on réserve les 3 premiers bits du fichier
- quand on a fini d'écrire, on revient au début du fichier avec `fseek` pour ajuster les trois premiers bits

Sauver des bits dans un fichier



Lecture

- On doit lire les trois premiers bits
- On doit savoir si on est au dernier octet ou non
- deux moyens de tester cela :

  - avoir un octet d'avance (lourd)

  - comparer la position courante et la taille du fichier

Lecture

- On doit ainsi rajouter des champs à notre fichier

```
1  struct bit_buffer{
2    /* The file to read from */
3    FILE* f;
4    /* Number of padding bits in the last byte */
5    char n_padding_bits;
6    /* Size of the file in bytes */
7    unsigned int size;
8    /* Bytes already read from the file */
9    unsigned int n_read;
10   /* The current Byte */
11   unsigned char c;
12   /* Position of the next bit to be read */
13   char n;
14 }
```

Lecture

- initialisation du buffer

```
1  /* Prepare a bit per bit reading of the given file.
2     Return 1 in case of succes and 0 if the file is
3     empty. */
4  int init(struct bit_buffer* b, FILE* f){
5    b->f=f;
6    fseek(f, SEEK_END, 0);
7    b->size=1+ftell(f);
8    if (b->size == 0) return 0;
9    fseek(f, SEEK_SET, 0);
10   b->c=fgetc(f);
11   b->n_read=1;
12   b->n_padding_bits=b->c>>5;
13   b->n=4; /* skip the 3 first bits */
14   return 1;
15 }
```

Lecture

- lecture d'un bit

```
1  /* Read one bit inside the given bit buffer.
2     Return -1 if the last signifiant bit has
3     been read */
4  int read_bit(struct bit_buffer* b){
5    if (b->n_read==b->size && b->n+1==b->n_padding_bits)
6      return -1; /* Reading ended */
7    int v=b->c&1<<(b->n);
8    v=(v!=0);
9    (b->n)--;
10   if (b->n==-1 && b->n_read!=b->size){
11     b->c=fgetc(b->f);
12     (b->n_read)++;
13     b->n=7;
14   }
15   return v;
16 }
```

Lecture

```
1  int main(int argc, char* argv[]){
2    FILE* f=fopen("foo", "rb");
3    struct bit_buffer b;
4    init(&b, f);
5    printf("size=%d bytes, %d padding bits\n",
6           b.size, b.n_padding_bits);
7    int i;
8    /* 'a' 'r' 't' <=> 01100001 01110010 01110100 */
9    printf("01100001 01110010 01110100\n");
10   while ((i=read_bit(&b))!=-1){
11     printf("%d", i);
12     if (b.n==7) printf(" ");
13   }
14   fclose(f);
15   return 0;
16 }
```

```
nborie@perceval:~> echo -n art > foo
nborie@perceval:~> ./test
size=3 bytes, 3 padding bits
01100001 01110010 01110100
   00001 01110010 01110
```