Programmation avancée en C :

Entrée-sortie avancée, manipulation de bits

Licence informatique 3e année

Université Gustave Eiffel

Section 1

Entrée-sortie avancée

E/S binaires

```
size_t fread(void *ptr, size_t size,
             size_t nmemb, FILE *stream);
```

- Elle lit `nmemb` éléments de taille `size` chacun, ...
- ... puis les met dans la zone d'adresse `ptr`,
- ... et retourne le nombre d'éléments de taille `size` lus
- $\neq$ nombre d'octets lus !
- `sizeof(size_t)` est machine dépendant (c'est souvent juste un alias de `unsigned int` sous système 32 bits et de `long unsigned int` sous 64 bits, quoi qu'il en soit, on adapte les `printf` en conséquence)

E/S binaires

- `fread` retourne le nombre d'éléments lu avec succès.
- Pourrait être moins que le `n` demandé (erreur, `EOF`, ...)
- Si on veut un nombre précis, on doit faire une boucle.

```
1 int read_n_ints(int* t, unsigned int n, FILE* f){
2   int i;
3   while (n && (i = fread(t, n, sizeof(int), f)) ){
4     t+=i;
5     n-=i;
6   }
7   return (n == 0);
8 }
```

E/S binaires

```
size_t fwrite(const void *ptr, size_t size,
              size_t nmemb, FILE *stream);
```

- Elle lit depuis la zone d'adresse `ptr`, ...
- ... écrit `nmemb` éléments de taille `size` chacun, ...
- ... et retourne le nombre d'éléments de taille `size` écrits.
- $\neq$ nombre d'octets écrits !

Fichiers textes ou binaires ?

- Différences d'efficacité en espace
- Au programmeur de choisir
- On peut mélanger les deux.

```
1  int main(int argc, char* argv[]){
2    int i = 708077092;
3    FILE* f = fopen("text", "w");
4    if (f == NULL) exit(1);
5    fprintf(f, "%d\n", i);
6    fclose(f);
7    f = fopen("bin", "wb");
8    if (f == NULL) exit(1);
9    fwrite(&i, sizeof(int), 1, f);
10   fclose(f);
11   return 0;
12 }
```

donne

```
nborie@perceval:~> ./test
nborie@perceval:~> cat text
708077092
nborie@perceval:~> cat bin
$f4*~>
```


Sérialisation

- Si un "objet" ne contient pas de pointeur, `fwrite` permet de le sauver en une seule fois.
- Idem en lecture

```
1  typedef struct{
2    int score;
3    int board[N][N];
4  } Game;
5
6  int save_game(char* n, Game* g){
7    FILE* f = fopen(n, "wb");
8    if (f == NULL) return -1;
9    fwrite(*g, sizeof(Game), 1, f);
10   fclose(f);
11   return 0;
12 }
```


Boutisme (endianness)

- Date en France : 14 juillet 1789
- Date ISO : 1789-07-14
- Boutisme : l'ordre des octets dans une structure multioctet (`int`, `long`, ...)
- Les résultats de `fread` et `fwrite` dépendent du boutisme.
- Fichiers potentiellement non portables !
- Sérialisation à éviter si besoin de portabilité

Encodage de texte

- Pour des fichiers textes, si on veut des codages sur plus d'un octet :

  - UTF8,

  - UTF16 (Little ou Big Endian),

  - ...,

  on doit gérer les E/S à la main, ou passer par une bibliothèque.

Positionnement

- `int fseek(FILE *stream, long offset, int origin);`
- Elle modifie la position courante dans le fichier.
- position = `origin` (point de départ) + `offset` (décalage)
- `origin` peut valoir :
  - `SEEK_SET` : début du fichier
  - `SEEK_CUR` : position courante
  - `SEEK_END` : fin du fichier
- Sur une machine n-bits, on ne peut pas adresser un fichier plus grand que 2^n avec `fseek` et `ftell`
- Équivalents plus portables :

```
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, fpos_t *pos);
```


Positionnement

- `void rewind(FILE* stream);`
- `long ftell(FILE* stream);`
- `rewind` revient au début du fichier
- `ftell` donne la position courante

```
1 long filesize(FILE* f){
2   long old_pos=ftell(f);
3   fseek(f, 0, SEEK_END);
4   long size=1+ftell(f);
5   fseek(f, old_pos, SEEK_SET);
6   return size;
7 }
```


Renommer et détruire

- `int rename(const char *oldpath, const char *newpath);`
- `int remove(const char *pathname);`
- Ces fonctions (`stdio.h`) fonctionnent aussi pour les répertoires.
- Elles renvoient 0 si OK, -1 si erreur.

Fichiers temporaires

- Comment être sûr qu'on ouvre un fichier qui n'existe pas déjà ?
- Pour obtenir un nom de fichier unique :
  - le faire à la main (pénible, fastidieux...)
  - utiliser une fonction qui sait le faire comme

```
char *tempnam(const char *dir, const char *pfx);
```

- Renvoie un pointeur sur une chaîne globale
  - pas thread-safe
- Le fichier peut avoir été créé entre l'obtention du nom et l'ouverture
- Ne pas être parano :
  - problème si et seulement si énormément de fichiers temporaires à gérer (application web, ...)

Tampon des E/S

Un léger parallèle :

- Vous acheminez régulièrement du poisson du Havre à Paris (On affiche souvent des choses à l'écran)...
- Le poisson a besoin d'être frais (On veut lire les choses rapidement à l'écran)...
- Il faut payer les chauffeurs (Solliciter l'affichage demande de la CPU (et d'autres choses coûteuses))...
- On peut stocker au Havre avant de faire partir un gros camion (On peut remplir un large buffer avant de solliciter l'affichage)...

L'enjeu :

Comment économiser de l'argent et livrer du poisson frais ?
Comment minimiser le coût en CPU sans que les messages attendent trop ?

Tampon des E/S

- 3 modes de tampon (*buffer*) :
  - tampon totale
  - tampon par ligne
  - sans tampon
- Vider les tampons avec `int fflush(File* stream)` (voir le manuel !)
- Tampon totale, avec l'espace du tampon déjà réservé :

```
void setbuf(FILE* stream, char* buf);
void setbuffer(FILE* stream, char* buf,
               size_t size);
```

- Avec `setbuf`, la taille du tampon est fixée à `BUFSIZ` (8192 octets par défaut pour une Lubuntu 64)
- Tampon par lignes :

```
void setlinebuf(FILE* stream);
```


Tampon des E/S

- Tampon générale :

```
int setvbuf(FILE *stream, char *buf, int
mode, size_t size);
```

- `mode` :
  - `_IONBF` : sans tampon
  - `_IOLBF` : tampon par lignes
  - `_IOFBF` : tampon complet (F=full)
- `size` : taille du tampon
- Elle renvoie 0 si OK, `EOF` sinon

Section 2

Manipulation des bits

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

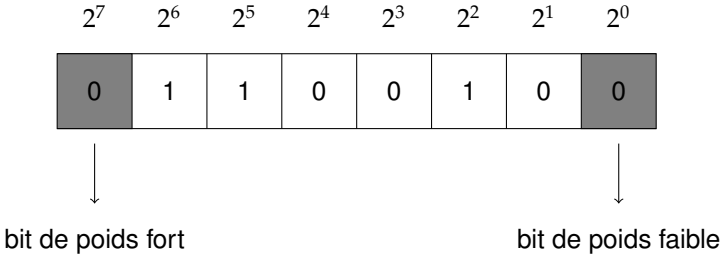
$= 0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
$= 64 + 32 + 4$
$= 100$

- 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
- Idem pour les types sur plusieurs octets



Entiers signés

- Leur codage 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

- Pour représenter un nombre négatif, on prend la valeur absolue,
- On inverse les bits,
- On ajoute 1 en ignorant les dépassements.
- exemple : -6 codé sur un octet

  6 = 00000110 $\Rightarrow$ ~6 = 11111001
  ~6+1 = 11111010 $\Rightarrow$ 250
- Pour les matheux : on fait modulo 2^n. ($-6 \equiv 250 \pmod{2^8}$)

```
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 `&&`
- Il n'y a pas de xor logique en C : `a^^b` :(
- Utiliser `(a && !b) || (!a && b)` pour le xor logique

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

Exemple : 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 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 droite (`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 5 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
```


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 ! Sinon, problème !

  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
```

  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 print_bin(unsigned 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(unsigned char* c, int bit){
2    *c = (*c) | (1<<bit);
3  }
4
5  int main(int argc, char* argv[]){
6    unsigned 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 `~(1 << n)`

```
1  void unset_bit(unsigned char *c, int bit){
2    *c = (*c) & ~(1<<bit);
3  }
4
5  int main(int argc, char* argv[]){
6    unsigned 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 $n^{\text{ème}}$ bit

- Mettre à 0 ou 1 en fonction de la valeur

```
1  void set(char *c, int bit, int value){
2    if (value) set_bit(c, bit);
3    else unset_bit(c, bit);
4  }
```

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`

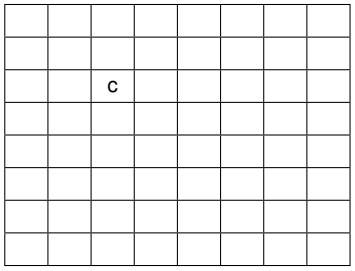
Représentation d'ensembles

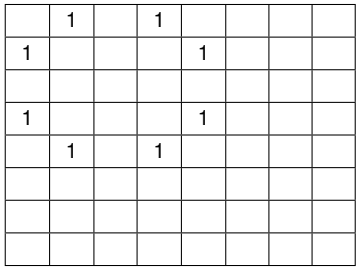
- Implémentation efficace d'ensemble
- L'intersection de A et B : `A & B`
- L'union de A et B : `A | B`
- Complément de A : `~A`
- Condition de "A inclu dans B" : `(A & B) == A`
- A \ B : `A & ~B`

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(unsigned char *c, unsigned char flag){
7    *c = (*c) | flag;
8  }
9
10 void unset_flag(unsigned char *c, unsigned char flag){
11   *c = (*c) & ~flag;
12 }
13
14 int is_set_flag(unsigned char c, unsigned char flag){
15   return c & flag;
16 }
```

Champs de bits - avantage

- Même esprit dans les champs de bits que dans les drapeaux
- Plus pratique que les masques
- On peut utiliser des nombres signés et des champs anonymes.
- C'est le langage qui gère tout, y compris les positions et les débordements.

Champs de bits - inconvénients

- 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 ?)
- Peu normés :
  - souvent par bloc d'1 `int`, mais pas forcément
  - champs contigus si possible, mais pas forcément
  - ordre des bits d'un champ non normé
  - implémentation des nombres signés non normée
- Champs non adressables, donc accéder-les par leurs noms !
- Sauvegarde d'un champs par `fwrite` impossible

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, uint8_t: %lu\n",
3           sizeof(int8_t), sizeof(uint8_t));
4    printf("int16_t: %lu, uint16_t: %lu\n",
5           sizeof(int16_t), sizeof(uint16_t));
6    printf("int32_t: %lu, uint32_t: %lu\n",
7           sizeof(int32_t), sizeof(uint32_t));
8    printf("int64_t: %lu, uint64_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
```


Tampon de bits

```
1 struct bit_buffer{
2   FILE* file;
3   int pos; /* position du prochain bit libre */
4   uint8_t current;
5 };
```

- 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){
7    b->current |= (value != 0) << b->pos;
8    b->pos--;
9    if(b->pos == -1){
10     fputc(b->current, b->file);
11     b->current = 0;
12     b->pos = 7;
13   }
14 }
```

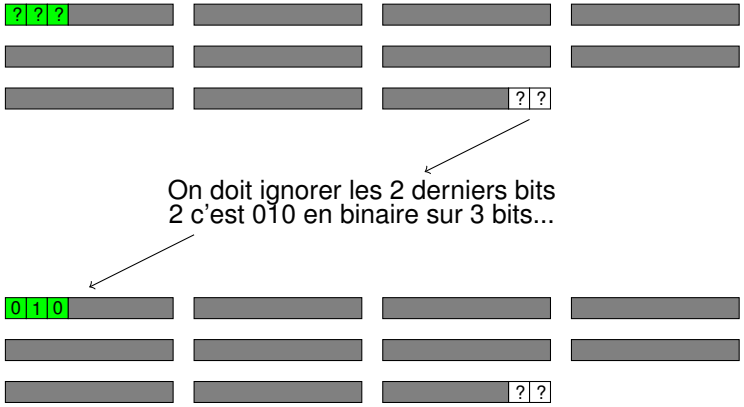

Sauver des bits dans un fichier

- Utiliser un tampon de bits
- Que faire si le dernier octet n'est pas rempli ?

Sauver des bits dans un fichier

- Sur le dernier octet on doit 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 :

  - 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/write */
3    FILE* f;
4
5    /* Bytes already read from / write to the file */
6    unsigned int bytepos;
7
8    /* Size of the file in bytes */
9    unsigned int bytesize;
10
11   /* Number of padding bits in the last byte */
12   char padding;
13
14   /* The current Byte */
15   uint8_t current;
16
17   /* Position of the next bit to be read/write */
18   char pos;
19 }
```


Lecture - Initialisation du tampon

```
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->file = f;
6
7    fseek(f, SEEK_END, 0);
8    b->bytesize = 1 + ftell(f);
9    if (b->bytesize == 0) return -1;
10
11   fseek(f, SEEK_SET, 0);
12   b->current = fgetc(f);
13   b->bytepos = 1;
14
15   b->padding = b->current >> 5; /* get padding size */
16   b->pos = 4; /* skip the 3 first bits */
17   return 0;
18 }
```


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->bytepos == b->bytesize && b->pos+1 == b->padding)
6      return -1; /* Reading ended */
7
8    int v = b->current & (1<<(b->pos));
9    v = (v!=0) ? 1 : 0;
10   (b->pos)--;
11   if (b->pos == -1 && b->bytepos != b->bytesize){
12     b->current = fgetc(b->file);
13     b->bytepos++;
14     b->pos = 7;
15   }
16   return v;
17 }
```

Lecture

```c
int main(int argc, char* argv[]){
  FILE* f=fopen("foo", "rb");
  struct bit_buffer b;
  init(&b, f);
  printf("%d bytes, %d padding bits\n", b.bytesize, b.padding);

  int i;
  printf("01100001 01110010 01110100\n"); /* "art" in ASCII */
  while ((i = read_bit(&b)) != -1){
    printf("%d", i);
    if (b.pos == 7) printf(" ");
  }
  fclose(f);
  return 0;
}
```

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