Cours de C - IR1
2007-2008

Manipulation de bits

Sébastien Paumier

MCours.com

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

$= 0\times2^7+1\times2^6+1\times2^5+0\times2^4+0\times2^3+1\times2^2+0\times2^1+0\times2^0$

$= 64 + 32 + 4$

$= 100$

Le binaire

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

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

Poids fort/poids faible

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

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	1	0	0	1	0	0

bit de poids fort

bit de poids faible

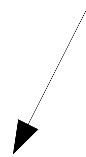
Poids fort/poids faible

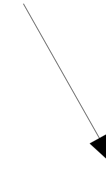
- idem pour les types sur plusieurs octets

16 bits

0	1	1	0	0	1	0	0

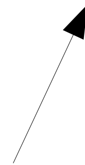
1	1	0	0	0	0	1	0

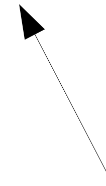




octet de poids fort

octet de poids faible





32 bits

0	1	1	0	0	1	0	0

1	1	1	0	1	0	0	0

0	0	1	0	0	0	1	1

1	1	0	0	0	0	1	0

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

$6 = 00000110 \Rightarrow \sim6 = 11111001$

$\sim6+1 = 11111010 \Rightarrow 250$

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

→

```
$>./a.out
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

```c
int main(int argc,char* argv[]) {
   char c=-26;
   int i=c;
   printf("%d %d\n",(unsigned char)c,i);
   return 0;
}
```

```
$>./a.out
230 -26
```

on obtient bien un **int** qui vaut -26 et non pas 230

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 inclusif)	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

- pas de XOR logique `a^^b` :(
- solution: transformer `a` et `b` en 0 ou 1

```
/* logical XOR */
int xor(int a,int b) {
   /* a&&a -> 0 if a==0
    *      -> 1 if a!=0 */
   return (a&&a)^(b&&b);
}

int main(int argc,char* argv[]) {
   printf("  0 XOR 0 = %d\n"
          "  0 XOR 4 = %d\n"
          "-17 XOR 0 = %d\n"
          " 13 XOR 9 = %d\n"
          ,xor(0,0),xor(0,4)
          ,xor(-17,0),xor(13,9));
   return 0;
}
```

```
$>./a.out
  0 XOR 0 = 0
  0 XOR 4 = 1
-17 XOR 0 = 1
 13 XOR 9 = 0
```

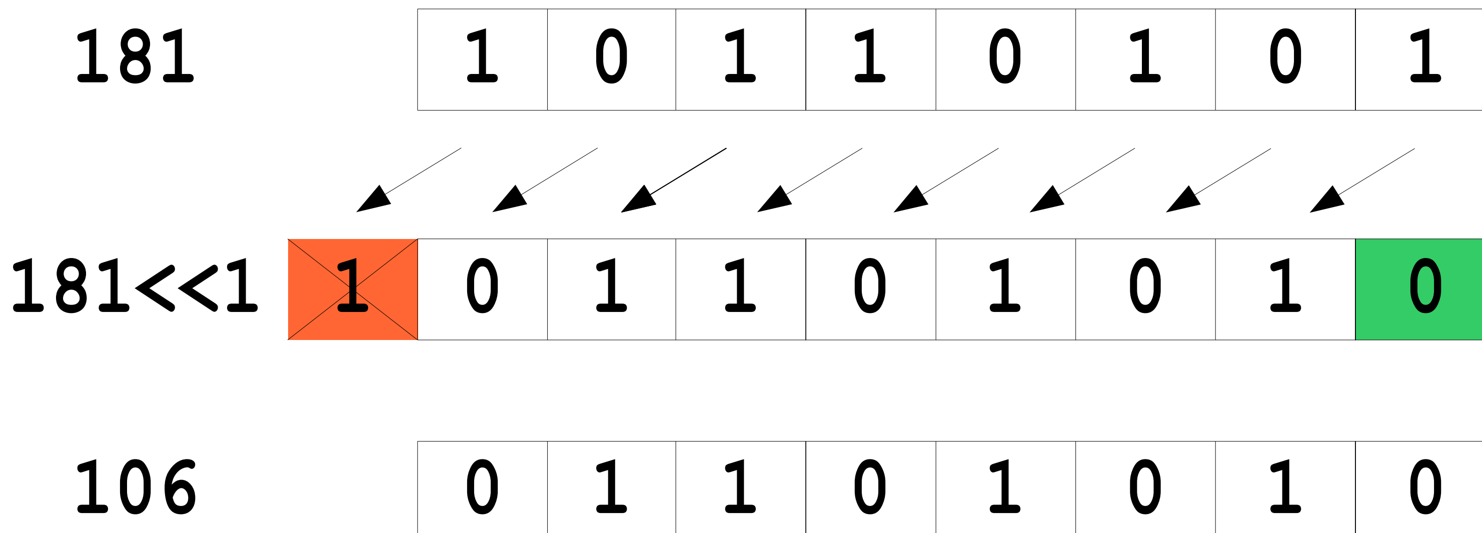
Opérateur unaire ~

- 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 fort sont perdus



paumier@univ-mlv.fr

Décalage à gauche

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

```
int main(int argc,char* argv[]) {
   unsigned char c=181;
   /*   10110101<<1
    * = 01101010 */
   c=c<<1;
   int i=181;
   /*   000...010110101<<1
    * = 000...101101010 */
   i=i<<1;
   printf("%d %d\n",c,i);
   return 0;
}
```

→

```
$>./a.out
106 362
```

$362=106+256$
$=2\times181$

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 valeur positive)
  - dépend de l'implémentation sinon!
- jamais de décalage sur des entiers signés sans une très bonne raison ⚡
- les bits de poids faible 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**

```
void shift_left(unsigned char *c,
                unsigned int n) {
   n=n%CHAR_BIT; /* make sure that n<CHAR_BIT */
   unsigned char tmp=(*c)>>(CHAR_BIT-n);
   *c=((*c)<<n)|tmp;
}
```

```
$>./a.out
00101111
10111100
```

Décalage circulaire à droite

- pour décaler **x** de n (<SIZE) bits:
  - décaler **x** de SIZE-n bits à gauche

```
void shift_right(unsigned char *c,
                 unsigned int n) {
   shift_left(c,CHAR_BIT-n%CHAR_BIT);
}
```

→

```
$>./a.out
10111100
00101111
```

- travailler sur du non signé!
  - si **shift_left** travaillait sur des **signed char**:

remplissage avec des 1 dû au complément à deux ←

```
$>./a.out
10111100
11101111
```

Un bit tout seul

- `n`ème bit (en partant de zéro) à 1 et les autres à 0 = `1<<n` (2^n)
-
- bit 0: `00000001` = `1<<0` = 2^0
- bit 1: `00000010` = `1<<1` = 2^1
- ...
- bit 7: `10000000` = `1<<7` = 2^7

MCours.com

Tester le $n^{ème}$ bit

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

```
void print_bin(char a) {
   int i;
   for (i=CHAR_BIT-1;i>=0;i--) {
      printf("%c",(a&(1<<i))?'1':'0');
   }
   printf("\n");
}

int main(int argc,char* argv[]) {
   print_bin(79);
   return 0;
}
```

→

```
$>./a.out
01001111
```

Mettre à 1 le n^ème^ bit

- OU inclusif bit à bit entre la valeur à tester et 1<<n

```
void set_bit(char *c,int bit) {
   *c=(*c)|(1<<bit);
}

int main(int argc,char* argv[]) {
   char c=79;
   printf("  "); print_bin(c);
   printf("| "); print_bin(1<<5);
   set_bit(&c,5);
   printf("= "); print_bin(c);
   return 0;
}
```

```
$>./a.out
  01001111
| 00100000
= 01101111
```

Mettre à 0 le n^{ème} bit

- ET bit à bit entre la valeur à tester et ~(1<<n)

```c
void unset_bit(char *c,int bit) {
   *c=(*c)&~(1<<bit);
}

int main(int argc,char* argv[]) {
   char c=79;
   printf("  "); print_bin(c);
   printf("& "); print_bin(~(1<<2));
   unset_bit(&c,2);
   printf("= "); print_bin(c);
   return 0;
}
```

```
$>./a.out
  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

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

void set2(char *c,int bit,int value) {
   unset_bit(c,bit);
   *c=(*c)|(value&&value)<<bit;
}
```

Drapeaux

- désigner les bits par des constantes qui servent de masques

```c
#define INITIAL 1
#define FINAL 2
#define ACCESSIBLE 4
#define COACCESSIBLE 8

void set_flag(char *c,int flag) {
    *c=(*c)|flag;
}

void unset_flag(char *c,int flag) {
    *c=(*c)&~flag;
}

int is_set_flag(char c,int flag) {
    return c&flag;
}
```

Masques

- stocker plusieurs informations sur un entier

1023=10 bits à 1

constantes non signées pour éviter le problème de la page 20

```
/**
 * A color representation with
 * 2 bits for alpha and 10 bits
 * for red, green and blue:
 * aarrrrrr rrrrgggg ggggggbb bbbbbbbb
 */
typedef unsigned int color;

void set_red(color *c,int value) {
   /* removing extra bits, if any */
   value=value&1023;
   *c=(*c)&~(1023u<<20u);
   *c=(*c)|(value<<20);
}

int get_red(color c) {
   return ((1023u<<20u)&c)>>20;
}
```

Champs de bits

- type particulier de structure:

```
struct color {
   unsigned int alpha: 2;
   unsigned int red: 10;
   unsigned int green: 10;
   unsigned int blue: 10;
};
```

- plus pratique que les masques
- mais non portables

Champs de bits

- peu normés:
  - souvent par blocs 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:

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

Champs de bits

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

```
struct color {
   unsigned int alpha:2,  red:10,
                green:10, blue:10;
};

/**
 * Safe saving of a color bit field.
 */
void write_color(struct color* c,FILE* f) {
   unsigned int i;
   i= (c->alpha<<30) | (c->red<<20)
    | (c->green<<10) | c->blue;
   fwrite(&i,sizeof(unsigned int),1,f);
}
```

Portabilité 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`**

```
int main(int argc,char* argv[]) {
printf("int8_t:   %d\nuint8_t:  %d\n"
       "int16_t:  %d\nuint16_t: %d\n"
       "int32_t:  %d\nuint32_t: %d\n"
       "int64_t:  %d\nuint64_t: %d\n"
       ,sizeof(int8_t),sizeof(uint8_t)
       ,sizeof(int16_t),sizeof(uint16_t)
       ,sizeof(int32_t),sizeof(uint32_t)
       ,sizeof(int64_t),sizeof(uint64_t));
return 0;
}
```

→

```
$>./a.out
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*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 [0;n-1]

```
/**
 * Numbers the chars in [a-zA-Z0-9]
 * from 0 to 61. Returns -1 if c is
 * not in the correct char set.
 */
int get_index(char c) {
   if (c>='a' && c<='z') return c-'a';
   if (c>='A' && c<='Z') return 26+c-'A';
   if (c>='0' && c<='9') return 52+c-'0';
   return -1;
}
```

Buffer de bits

```
struct bit_buffer {
   unsigned char c;
   char n; /* position du prochain bit libre */
};
```

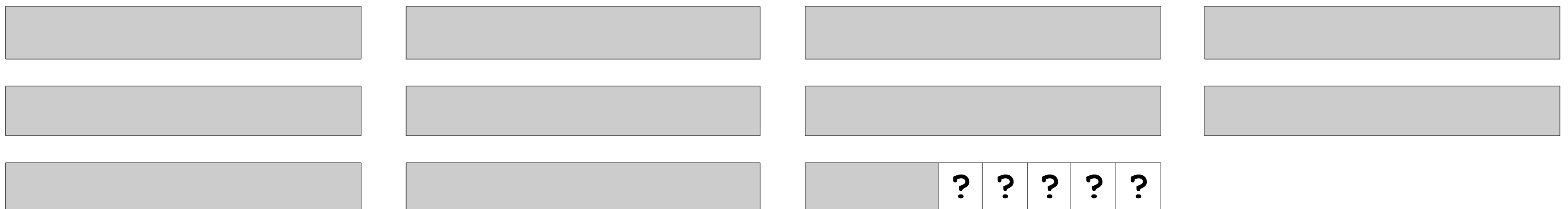
- à sauver et réinitialiser quand on a écrit 8 bits
- ordre de remplissage des bits au choix (poids fort vers poids faible ou l'inverse)

Buffer de bits

```
/**
 * Adds a 1 or a 0 to the given bit buffer,
 * depending on the given value. If the bit
 * buffer is filled up, it is flushed into the
 * given file and reseted.
 * Note that the bit buffer must have been
 * initialized to b->c=0 and b->n=7 before
 * the first call to this function.
 */
void write_bit(struct bit_buffer* b,int value,FILE* f) {
b->c=b->c|(value&&value)<<b->n;
(b->n)--;
if (b->n==-1) {
   fputc(b->c,f);
   b->c=0;
   b->n=7;
}
}
```

Sauver des bits dans un fichier

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

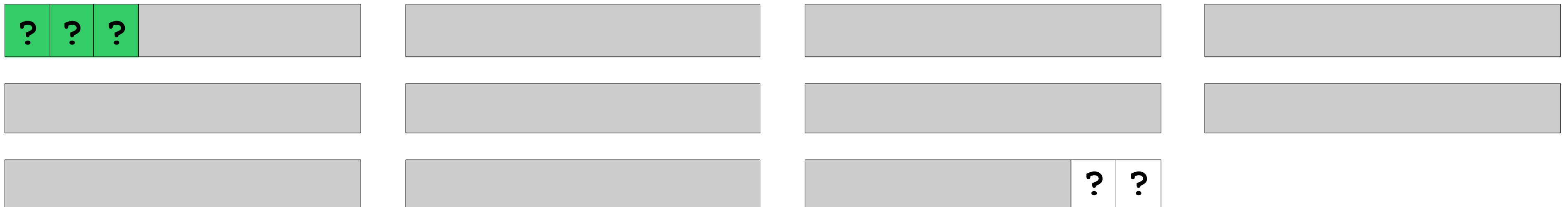


Sauver des bits dans un fichier

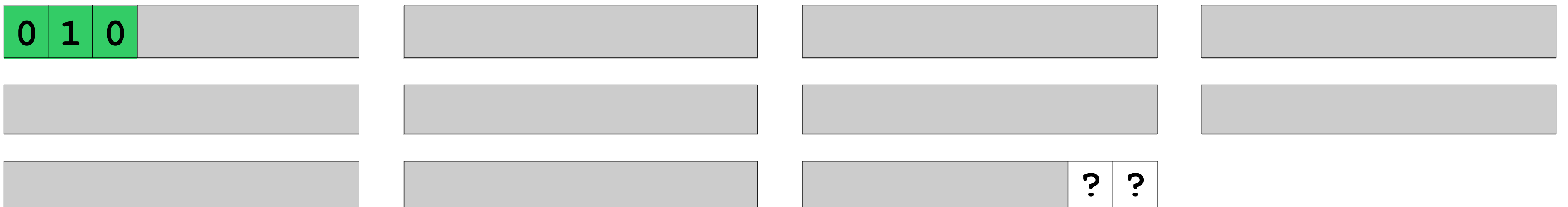
- sur le dernier octet on peut ignorer de 0 (octet plein) à 7 bits (un seul bit utilisé)
- pour coder 8 valeurs, il faut 3 bits
- on réserve les 3 premiers bits du fichier
- quand on a fini d'écrire, on revient au début du fichier avec `fseek` pour ajuster les 3 premiers bits

Sauver des bits dans un fichier

? ? ? on réserve les 3 premiers bits, puis on écrit les données



on doit ignorer les 2 derniers bits, soit 010 en binaire



Lecture

- on doit lire les 3 premiers bits
- on doit savoir si on est au dernier octet ou non
- deux moyens de tests:
  - avoir un octet d'avance (lourd)
  - comparer la position courante et la taille du fichier

Lecture

- on doit ajouter des champs à notre buffer

```
struct bit_buffer {
    /* The file to read from */
    FILE* f;
    /* Number of padding bits on the last byte */
    char n_padding_bits;
    /* Size of the file in bytes */
    unsigned int size;
    /* Bytes already read from the file */
    unsigned int n_read;
    /* The current byte */
    unsigned char c;
    /* Position of the next bit to be read
     * (from 7 to 0) */
    char n;
};
```

Lecture

- initialisation du buffer

```
/**
 * Prepares bit per bit reading from the given
 * file. Return 1 in case of success; 0 if the file
 * is empty.
 */
int init(struct bit_buffer* b,FILE* f) {
   b->f=f;
   fseek(f,SEEK_END,0);
   b->size=1+ftell(f);
   if (b->size==0) return 0;
   fseek(f,SEEK_SET,0); /* returning at the beginning */
   b->c=fgetc(f);
   b->n_read=1;
   b->n_padding_bits=(b->c&0xFF)>>5;
   b->n=4; /* we skip the 3 first bits */
   return 1;
}
```

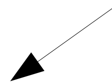
Lecture

- lecture d'un bit

```
/**
 * Reads and returns one bit from the file associated
 * to the given bit buffer. Returns -1 when the last
 * significant bit of the file has been read.
 */
int read_bit(struct bit_buffer* b) {
   if (b->n_read==b->size && b->n+1==b->n_padding_bits) {
      return -1; /* We have finished */
   }
   int v=b->c&1<<(b->n);
   v=v&&v;
   (b->n)--;
   if (b->n==-1 && b->n_read!=b->size) {
      /* reading next byte */
      b->c=fgetc(b->f);
      (b->n_read)++;
      b->n=7;
   }
   return v;
}
```

Lecture

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



```
$>echo -n art > foo
$>./a.out
size=3 bytes, 3 padding bits
01100001 01110010 01110100
   00001 01110010 01110
```