CM 9 - MANIPULATION BINAIRE

G. Bianchi, G. Blin, A. Bugeau, S. Gueorguieva, R. Uricaru

2015-2016

Programmation 1 - uf-info.ue.prog1@diff.u-bordeaux.fr

université de BORDEAUX

LE BINAIRE

LA REPRÉSENTATION EN BASE 2

▷ Dans un mot binaire, chaque position correspond à une puissance de deux - c'est la représentation positionnelle vue en CP sur une base 2 plutôt que 10

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

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

LA BASE 2

▷ Il faut connaître les premières puissances de deux par coeur

▷ Pratique pour détecter des valeurs spéciales (e.g. 1023, 4097, ...)

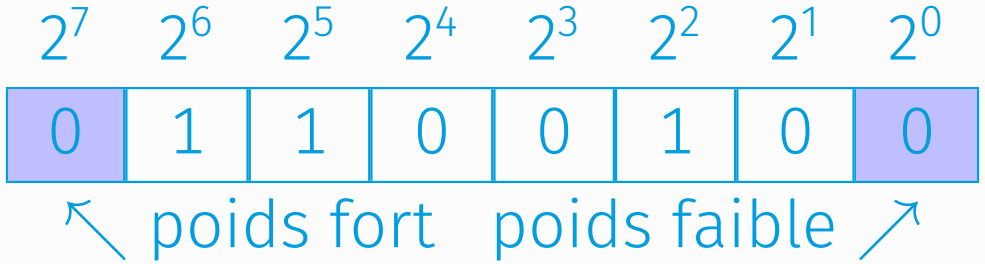
128	64	32	16	8	4	2	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

8192	4096	2048	1024	512	256
2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8

POIDS FORT/POIDS FAIBLE

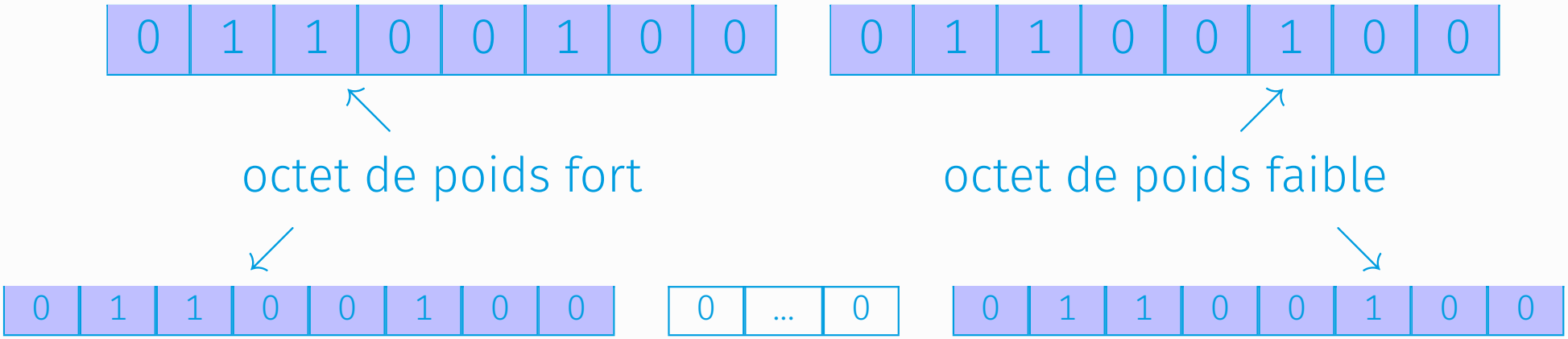
▷ On appelle poids fort, la position de la plus grande puissance de deux

▷ On appelle poids faible, la position de la plus petite puissance de deux



POIDS FORT/POIDS FAIBLE

▷ Il en est de même pour les types sur plusieurs octets



LES ENTIERS SIGNÉS

▷ ⚠Le codage d'un entier signé est dépendant de l'implémentation !

▷ Dans la représentation par magnitude signée, le bit de poids fort encode le signe de l'entier (0 =positif, 1 =négatif)

  ▷ $+25_{10} = 00011001_2$ et $-25_{10} = 10011001_2$

▷ Inconvénients:

  ▷ Le nombre 0 possède deux représentations que sont $+0_{10} = 0000000_2$ et $-0_{10} = 1000000_2$

▷ La simple addition ne fonctionne pas

  ▷ $10000100 + 00000011 = 10000111$ $(-4 + 3 = -7)$

▷ Du coup, très peu utilisée (mais utilisable)

LES ENTIERS SIGNÉS

▷ ⚠Le codage d'un entier signé est dépendant de l'implémentation !

▷ Dans la représentation à complément à deux, (1) on prend la valeur absolue, (2) on inverse les bits (complément à 1) et (3) on ajoute 1 au résultat en ignorant les dépassements de capacités

▷ Le nombre -6 codé sur un octet s'obtient donc ainsi

- ▷ $6 = 00000110 \quad \Rightarrow \quad \sim(6) = 11111001$
- ▷ $\sim(6) + 1 = 11111010 \quad \Rightarrow \quad 250$

```
int main(void){
    char c = -6;
    printf("%u\n",(unsigned char) c);
    return EXIT_SUCCESS;
}
```

```
$> ./a.out
250
```

LES ENTIERS SIGNÉS

▷ ⚠Le codage d'un entier signé est dépendant de l'implémentation !

▷ Dans la représentation à complément à deux, (1) on prend la valeur absolue, (2) on inverse les bits (complément à 1) et (3) on ajoute 1 au résultat en ignorant les dépassements de capacités

▷ L'addition fonctionne correctement

- ▷ $11111100 + 00000011 = 11111111$ $(-4 + 3 = -1)$
- ▷ Il y a une écriture unique de $0_{10} = 00000000_2$
- ▷ $-(-x) = x$

CONVERSION, PROMOTION ET COERCITION

▷ Retour sur l'exemple précédent

```
int main(void){
    char c = -6;
    printf("%u\n",(unsigned char) c);
    return EXIT_SUCCESS;
}
```

▷ La chaîne de format spécifie qu'elle attend un `unsigned int` et l'argument passé est de type `char` transformé en `unsigned char`

▷ Que s'est-il passé pour que cela fonctionne ?

  ▷ Une conversion entre deux types compatibles (`char` et `unsigned char`)

  ▷ Suivie d'une transformation : la promotion de la valeur

CONVERSION, PROMOTION ET COERCITION

▷ Rappel : type = (taille, interpretation)

▷ Un type de départ $t_d = (\text{taille}_d, \text{interpretation}_d)$ est compatible avec un type cible $t_c = (\text{taille}_c, \text{interpretation}_c)$ si $\text{taille}_c \geq \text{taille}_d$

▷ Si $\text{taille}_c == \text{taille}_d$, les bits de l'expression de type de départ sont recopiés tels quels dans l'expression de type cible. C'est donc l'interprétation qui change mais pas les données.

▷ Si $\text{taille}_c > \text{taille}_d$, les bits de l'expression de type de départ sont recopiés tels quels dans les octets de poids faibles de l'expression de type cible.

▷ Que se passe-t-il pour les bits non "affectés" par cette recopie ?

CONVERSION, PROMOTION ET COERCITION

▷ Rappel : type = (taille, interpretation)

▷ Un type de départ $t_d = (\text{taille}_d, \text{interpretation}_d)$ est compatible avec un type cible $t_c = (\text{taille}_c, \text{interpretation}_c)$ si $\text{taille}_c \geq \text{taille}_d$

▷ Si $\text{taille}_c == \text{taille}_d$, les bits de l'expression de type de départ sont recopiés tels quels dans l'expression de type cible. C'est donc l'interprétation qui change mais pas les données.

▷ Si $\text{taille}_c > \text{taille}_d$, les bits de l'expression de type de départ sont recopiés tels quels dans les octets de poids faibles de l'expression de type cible.

▷ Que se passe-t-il pour les bits non "affectés" par cette recopie ? Cela dépend du type cible ...

CONVERSION, PROMOTION ET COERCITION

▷ C'est le mécanisme de promotion qui est en jeu

▷ Si $\text{taille}_c > \text{taille}_d$ et le type cible type_c est non signé

  ▷ Des 0 sont placés dans les bits non "affectés" par la recopie

▷ Si $\text{taille}_c > \text{taille}_d$ et le type cible type_c est signé

  ▷ C'est le bit de poids fort de l'expression de type type_d qui est placé dans l'ensemble des bits non "affectés" par la recopie

  ▷ Cela permet de "conserver" le signe et la valeur de départ si le type type_c est signé

  ▷ Cela peut changer le signe et la valeur de départ si le type type_c est non-signé

CONVERSION, PROMOTION ET COERCITION

▷ C'est le mécanisme de promotion qui est en jeu

```c
int main(void){
    char c = -6; // 11111010
    unsigned char d = 250; // 11111010
    printf("%u\n",(unsigned char) c); //0...0 11111010 (250)
    printf("%u\n", c); //1...1 11111010 (4 294 967 290)
    printf("%d\n", c); //1...1 11111010 (-6)
    printf("%u\n",(char) d); //1...1 11111010 (4 294 967 290)
    printf("%u\n", d); //0...0 11111010 (250)
    printf("%d\n", d); //0...0 11111010 (250)
    return EXIT_SUCCESS;
}
```

CONVERSION, PROMOTION ET COERCITION

▷ Il est possible de forcer une conversion d'un type $type_d$ vers un type $type_c$ non compatible (i.e., $taille_c < taille_d$)

▷ C'est alors le mécanisme de coercition qui est mis en oeuvre

▷ Seuls les octets de poids faibles jusqu'à obtention de $taille_c$ octets sont recopiés ; les autres sont ignorés

▷ ⚠ Il faut être sûr de ce que l'on fait

```
int main(void){
    int c = -6; // 1...1 11111010
    unsigned int d = 270; // 0...01 00001110
    printf("%d\n",(char) c); //11111010 (-6)
    printf("%u\n",(unsigned char) c); //11111010 (250)
    printf("%u\n",(char) d); //00001110 (14)
    printf("%u\n",(unsigned char) d); //00001110 (14)
    return EXIT_SUCCESS;
}
```

OPÉRATEURS BIT À BIT

OPÉRATEURS BIT À BIT

▷ Les opérateurs bit à bit ne fonctionnent que sur les entiers

▷ ⚠Ces opérateurs sont à différencier des 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

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

OPÉRATEURS BIT À BIT

▷ Les opérateurs bit à bit ne fonctionnent que sur les entiers

▷ ⚠Ces opérateurs sont à différencier des 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

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

OPÉRATEURS BIT À BIT

▷ Les opérateurs bit à bit ne fonctionnent que sur les entiers

▷ ⚠Ces opérateurs sont à différencier des 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

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

Opérateurs bit à bit

▷ Les opérateurs bit à bit ne fonctionnent que sur les entiers

▷ ⚠Ces opérateurs sont à différencier des opérateurs logiques `&&` et `||`

▷ L'opérateur `~` permet d'inverser les bits (complément à un)

a	1	0	1	1	0	1	1	1

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

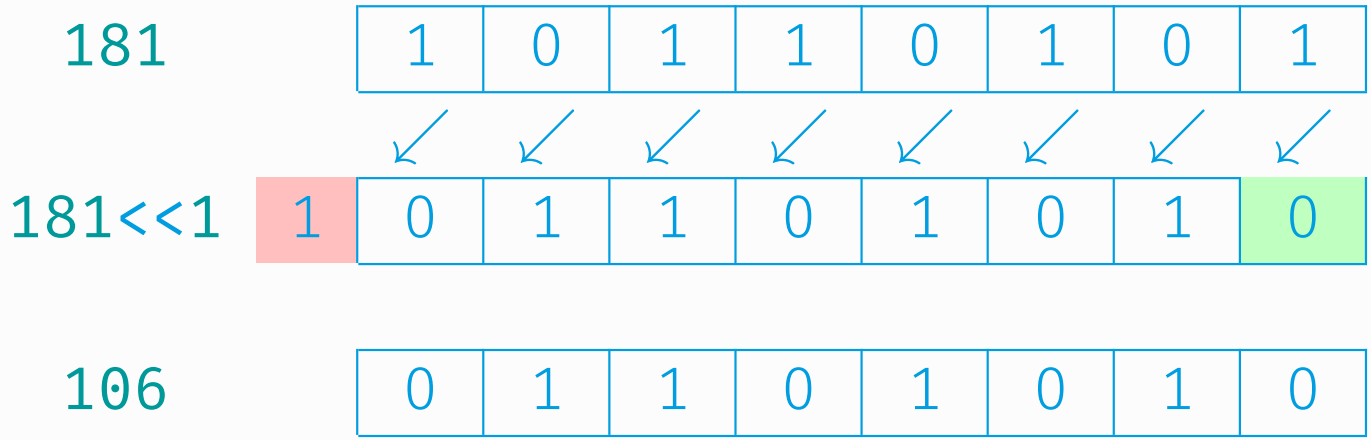
OPÉRATEURS DE DÉCALAGE À GAUCHE

▷ En langage C, l'opérateur `<<` permet de décaler vers la gauche les bits d'un entier

▷ `x<<y` décale les bits de `x` de `y` bits vers la gauche (`x` n'est pas modifié)

▷ Les bits de poids faibles décalés sont remplacés par des `0`

▷ Les bits de poids forts décalés sont perdus



Opérateurs de décalage à gauche

▷ Le décalage à gauche ne conduit pas forcément à un résultat inapproprié

▷ En effet, le décalage de `181` stocké dans un `char` provoque la perte d'un bit significatif mais ce n'est pas le cas si ce dernier est stocké dans un `int`

```
int main(void){
    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 EXIT_SUCCESS;
}
```

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

▷ $362 = 106 + 256 = 2 * 181$

Opérateurs de décalage à droite

▷ En langage C, l'opérateur `>>` permet de décaler vers la droite les bits d'un entier

▷ `x>>y` décale les bits de `x` de `y` bits vers la droite (`x` n'est pas modifié)

▷ Les bits de poids forts décalés sont remplacés par

  ▷ des `0` si le type est non signé

  ▷ ⚠des `0` ou des `1` suivant l'implémentation si le type est signé

▷ ⚠Il ne faut donc jamais effectuer de décalage à droite d'entiers signés sans une très bonne raison

▷ Les bits de poids faibles sont perdus

DÉCALAGE CIRCULAIRE À GAUCHE

▷ Le décalage circulaire consiste à ré-injecter les bits sortant en tant que bits entrant

▷ Pour décaler `x` – de taille `SIZE` – de `n` bits vers la gauche

  ▷ On copie `x` dans une variable `tmp`

  ▷ On décale `tmp` de `SIZE-n` bits vers la droite

  ▷ On décale `x` de `n` bits vers la gauche

  ▷ On effectue un ”OU bit à bit” (`|`) entre les résultats des décalages

```
void shiftLeft(unsigned char * c, unsigned int n){
    n=n%CHAR_BIT;
    unsigned char tmp= (*c) >> (CHAR_BIT-n);
    *c=((*c)<<n)|tmp;
}
```

```
$> ./a.out
shiftLeft(x,2)
00101111
10111100
```

DÉCALAGE CIRCULAIRE À DROITE

▷ Le décalage circulaire consiste à ré-injecter les bits sortant en tant que bits entrant

▷ Pour décaler `x` – de taille `SIZE` – de `n` bits vers la droite

  ▷ On décale circulairement `x` de `SIZE-n` bits vers la gauche

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

```
$> ./a.out
shiftRight(x,2)
10111100
00101111
```

MASQUE BINAIRE

▷ Il est possible de générer un mot binaire avec un `1` uniquement au `n`ème bit avec les autres bits à `0`

  ▷ `1<<n` – qui correspond à 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 NÈME BIT

▷ Pour récupérer la valeur du nème bit d'une valeur à tester, il suffit d'effectuer un "ET bit à bit" (&) entre la valeur à tester et le masque 1<<n

▷ Cela permet, par exemple, d'afficher en binaire une valeur

```
void printBinary(char a){
    int i;
    for(i=CHAR_BIT-1;i>=0;i=i-1){
        if((a&(1<<i))){
            printf("1");
        }else{
            printf("0");
        }
    }
    printf("\n");
}
```

```
int main(void){
    printBinary(79);
    return EXIT_SUCCESS;
}
```

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

METTRE LE NÈME BIT À 1

▷ Pour fixer à 1 le n**è**me bit d'une valeur, il suffit d'effectuer un "OU bit à bit" (|) entre la valeur à tester et le masque 1<<n

```
void setBit(char *c, int bit){
    *c=((*c)|(1<<bit));
}
int main(void){
    char c=79;
    printf("  ");
    printBinary(c);
    printf("| ");
    printBinary(1<<5);
    setBit(&c,5);
    printf("= ");
    printBinary(c);
    return EXIT_SUCCESS;
}
```

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

METTRE LE NÈME BIT À 0

▷ Pour fixer à 0 le nème bit d'une valeur, il suffit d'effectuer un "ET bit à bit" (&) entre la valeur à tester et le masque `~(1<<n)`

```
void unsetBit(char *c, int bit){
    *c=((*c)&~(1<<bit));
}
int main(void){
    char c=79;
    printf("  ");
    printBinary(c);
    printf("& ");
    printBinary(~(1<<2));
    setBit(&c,2);
    printf("= ");
    printBinary(c);
    return EXIT_SUCCESS;
}
```

```
$>./a.out
  01001111
& 11111011
= 01001011
```

AFFECTER LE NÈME BIT

▷ Il existe deux méthodes pour affecter le **n**ème bit

  ▷ Méthode 1: Mettre à **0** ou à **1** en fonction de la valeur

  ▷ Méthode 2: Mettre à **0** puis effectuer un "OU bit à bit" (**|**) avec la valeur décalée de **n** bits vers la gauche

```
void setBit1(char *c, int bit, int value){
    if(value){
        setBit(c,bit);
    }else{
        unsetBit(c,bit);
    }
}

void setBit2(char *c, int bit, int value){
    unsetBit(c,bit);
    *c=((*c) | ((value && value)<<bit));
}
```

UTILISATION DES BITS

Stockage d'informations

▷ Il est par exemple possible de stocker les valeurs de propriétés binaires – appelées "drapeaux" (flags en anglais) – en désignant des constantes servant de masques pour chaque bit

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

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

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

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

```c
int main(void){
    char c='\0';
    setFlag(&c,FINAL);
    printBinary(c);
    setFlag(&c,(INITIAL|ACCESSIBLE));
    printBinary(c);
    if(isFlagSet(c,FINAL)){
        unsetFlag(&c,INITIAL);
    }
    printBinary(c);
    return EXIT_SUCCESS;
}
$>./a.out
00000010
00000111
00000110
```

Stockage d'informations

▷ Il est également possible d'utiliser un entier pour stocker plusieurs informations

```
/*
* A color representation with 2 bits for alpha and 10 bits for each
* of red, green and blue: aarr rrrr rrrr gggg gggg ggbb bbbb bbbb
*/
typedef unsigned int color;

void setRed(color * c, unsigned int value){
    /* removing extra bits if any */
    value=value&1023; //0...011 1111 1111
    *c=((*c)&(~(1023u<<20u)));//aarr rrrr rrrr gggg gggg ggbb bbbb bbbb
                              //1100 0000 0000 1111 1111 1111 1111 1111
    *c=((*c)|(value<<20u));   //aa00 0000 0000 gggg gggg ggbb bbbb bbbb
                              //00vv vvvv vvvv 0000 0000 0000 0000 0000
}
...
```

Stockage d'informations

▷ Il est également possible d'utiliser un entier pour stocker plusieurs informations

```
/*
* A color representation with 2 bits for alpha and 10 bits for each
* of red, green and blue: aarr rrrr rrrr gggg gggg ggbb bbbb bbbb
*/
typedef unsigned int color;

void setRed(color * c, unsigned int value){
    ...
}

unsigned int getRed(color c){
    return ((1023u<<20u)&c)>>20;
}           //0011 1111 1111 0000 0000 0000 0000 0000
            //aarr rrrr rrrr gggg gggg ggbb bbbb bbbb
            //00rr rrrr rrrr 0000 0000 0000 0000 0000
            //0000 0000 0000 0000 0000 00rr rrrr rrrr
```

CHAMPS DE BITS

▷ Les champs de bits correspondent à un type particulier de champ d'une structure où ce dernier est suivi du nombre de bits qu'il doit avoir

```
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

CONTRAINTES SUR LES CHAMPS DE BITS

▷ Les seuls types acceptés pour les champs de bits sont `int`, `unsigned int` et `signed int`

▷ L'ordre des champs est respecté

▷ L'ordre des bits d'un champ n'est pas normé ⚠

▷ Un champ de bits déclaré comme étant de type `int` peut en fait se comporter comme un `signed int` ou un `unsigned int` suivant l'implémentation ⚠

  ▷ Toujours utiliser des champs de bits de type `unsigned int`

▷ La taille d'un champ de bits ne doit pas excéder celle d'un entier

▷ Un champ de bits n'a pas d'adresse ; seule sa valeur est accessible (comme tout autre champ de la structure)

CHAMPS DE BITS

▷ Les champs de bits correspondent à un type particulier de champs d'une structure où ce dernier est suivi du nombre de bits qu'il doit avoir

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

void setRed(struct color * c, int value){
   (*c).red=value;
}

int getRed(const struct color * c){
    return (*c).red;
}
```

```
int main(void){
    struct color c={0,0,0,0};
    setRed(&c,255);
    printf("%d",c.red);
    setRed(&c,1025);
    printf("%d",c.red);
    return EXIT_SUCCESS;
}

$>./a.out
255
1
```

DOGGY BAG

TO TAKE AWAY ...

▷ Il est risqué de s'appuyer sur une norme de codage des entiers pour faire des calculs ou de l'analyse car elle dépend de l'implémentation

▷ La conversion de type fait appel à deux mécanismes possibles: la promotion et la coercition

▷ Les opérateurs bit à bit permettent la gestion fine d'une valeur au niveau du bit

▷ Cette gestion fine permet de stocker des propriétés ou informations avec un codage binaire personnel et ceci en optimisant l'espace mémoire

Questions?