Le C pour l'embarqué

Avant-propos

L'objectif de cette série d'articles sur le langage C, n'est pas de faire un cours exhaustif, mais de réaliser un focus sur certains aspects techniques, pas toujours suffisament développés dans les cours, mais nécessaires quand on travaille sur des systèmes embarqués.

Pour traiter les exemples présentés, nous pouvons utiliser un simulateur de code C en ligne https://www.onlinegdb.com/online_c_compiler.

Manipulation sur les Bits

La technique du masquage consiste à modifier individuellement des bits sans affecter les autres bits.

- sur un octet, on est sur 8 bits, bit#7 ... bit#0
- sur un int, on est sur 4 octets, bit#31 ... bit#0

Opérateurs de manipulation de bits

```
&      // AND bit à bit
|      // OR bit à bit
^      // OU Exclusif bit à bit
~      // NOT bit à bit (complément à 1)
<<     // décalage à gauche
>>     // décalage à droite
```

- Dans l'exemple ci-dessous, j'utilise le type uint8_t pour travailler sur 8 bits en non-signé
- Pour utiliser uint8_t, il faut inclure <stdint.h>
- 0x18 correspond à la valeur 18 en hexadécimal, soit 24 en décimal, ou 0b0001 1000 en binaire.
- Pour afficher le contenu d'un uint8_t dans un printf on utilise %hhx à la place de %d pour un int, pour afficher la valeur en hexa.

Exemple d'utilisation

A partir de ce programme C, essayez dans un premier temps d'anticiper les résultats à obtenir avant d'éxecuter le code.

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

int main()
{
  uint8_t A=0x18,B=0x57, Res;
  Res=A&B;
  printf("A=0x %hhx & B=0x %hhx donne Res=0x %hhx \n",A,B,Res);
  Res=B>>3;
  printf("Un décalage à droite de B>>3 donne Res=0x %hhx \n",Res);
  Res=~A;
  printf("Le complément à 1 de A donne  Res=0x %hhx \n",Res);
  return 0;
}
```

Solution

- A=0x18 & B=0x57 donne Res=0x10, en binaire 0b0001 1000 & 0b0101 0111 donne 0b0001 0000 soit 0x10
- Un décalage à droite de B>>3 donne Res=0xa, en binaire 0b0101 0111 décalé de 3 à droite donne 0b0000 1010 soit 0x0a ou 0xa
- Le complément à 1 de A donne Res=0xe7, en binaire, le complément à 1 de 0b0001 1000 donne 0b1110 0111 soit 0xe7

Rappel: un décalage de 1 bit vers la gauche correspond à une multiplication par 2, 2 bits à gauche correspond à une multiplication par 4 .... Pour un décalage à droite, le principe est identique mais engendre une division.

Bit SET

Pour faire une mise à 1 de bit (SET), nous utiliserons l'operateur OR qui est représenté par `|` et qui effectue une opération OR bit à bit.

```
// Nous voulons faire un SET du Bit#7 d'un registre nommé: REG
REG = REG |  0x80;

// Pour fair un SET du bit#31:
```

```
REG = REG |  0x80000000;

// Simplifions :
// (1 << 31) signifie que 1 sera décalé à gauche 31 fois pour produire 0x80000000
REG = REG | (1 << 31);

// Simplifions et compactons:
REG |= (1 << 31);

// Un SET du Bit#21 et du Bit#23:
REG |= (1 << 21) | (1 << 23);
```

Exemple

```
#include <stdio.h>

int main()
{
  int REG=5;
  printf("Au départ REG = %d\n",REG);
  REG |= (1<<7);
  printf("REG |= (1<<7) = %d",REG);
  return 0;
}
```

Solution

- REG = 8 soit en binaire 0b0000_0101
- on met le bit#7 à 1
- REG = 0b1000_0101 soit 133 en décimal

Bit CLEAR

Pour mettre un bit à 0, on parle de Reset ou de Clear.

- Une fonction AND (ET) notée `&` sera utilisée en association avec 0 pour effectuer une mise à 0.
- Une fonction AND associée avec 1 ne change rien (élément neutre)

L'opérateur tilde (`~`) peut nous aider pour simplifier cette opération

- pour rappel, l'opérateur tilde (`~`) correspond à la fonction NOT

```
// Nous voulons faire un reset du Bit#7 d'un registre nommé: REG
REG = REG &   0x7F;
REG = REG & ~(0x80); // Même chose, mais en utilisant ~ pour simplifier

// Faison un reset du bit#31:
REG = REG & ~(0x80000000);

// Simplifions :
REG = REG & ~(1 << 31);

// Simplify et compactons :
REG &= ~(1 << 31);

// Reset Bit#21 et Bit# 23:
REG &= ~( (1 << 21) | (1 << 23) );
```

Exemple

```
#include <stdio.h>

int main()
{
  int REG=133;
  printf("Au départ REG = %d\n",REG);
REG &= ~(1<<7);
printf("REG &= ~(1<<7) = %d",REG);
  return 0;
}
```

Solution

- REG = 133 soit en binaire 0b1000_0101
- on met le bit#7 à 0
- REG = 0b0000_0101 soit 5 en décimal

Bit TOGGLE

Réaliser un Toggle d'un bit consiste à inverser l'état actuel du bit. On utiliser la fonction XOR notée `^`

```
// On utilise XOR pour faire un toggle du bit#5
REG ^= (1 << 5);

// Inversion du bit#3 et du bit#5
REG ^= ((1 << 3) | (1 << 5));
```

Exemple

```
#include <stdio.h>

int main()
{
  int REG=133;
  printf("Au départ REG = %d\n",REG);
  REG ^= (1 << 7);
  printf("Toggle du bit#7 de REG  = %d\n",REG);
  REG ^= (1 << 7);
  printf("on refait un Toggle du bit#7 de REG  = %d\n",REG);
  return 0;
}
```

Solution

- REG = 133 soit en binaire 0b1000_0101
- Le bit#7 est à 1, si l'on fait un Toggle, le bit#7 passe à 0 càd REG = 0b0000_0101 = 5
- REG = 5 soit en binaire 0b0000_0101
- Le bit#7 est à 0, si l'on fait un Toggle, le bit#7 passe à 1 càd REG = 0b1000_0101 = 133

Bit CHECK

Supposons que nous souhaitons vérifier que le bit 7 d'un registre est à 1 (set) :

- deux métodes

```
// Si bit#7 est à 1, on lance la fonction DoAThing()
if(REG & (1 << 7))
{
```

```
 DoAThing();
}

// On reste bloqué dans la boucle while tant que bit#7 est à 0
while( ! (REG & (1 << 7)) ) {
  ;
}
```

Voici un autre exemple dans lequel nous souhaitons attendre jusqu'à ce que le bit#9 soit à 0

```
// Tant que bit#9 n'est pas à zero (tant que bit#9 est set)
while((REG & (1 << 9)) != 0) {
  ;
}

// Tant que bit#9 est set
while(REG & (1 << 9)) {
  ;
}
```

Brainstorming

Combien de manière pouvons nous trouver pour tester si l'entier value est une puissance de deux en utilisant les manipulations de bits.

```
(value | (value + 1)) == value
(value & (value + 1)) == value
(value & (value - 1)) == 0
(value | (value + 1)) == 0
(value >> 1) == (value/2)
((value >> 1) << 1) == value
```

Exercice

Que réalise cette fonction ?

```
boolean foo(int x, int y) {
  return ((x & (1 << y)) != 0);
}
```

Solution

- la fonction foo reçoit deux entiers x et y en paramètres
- on créer un masque avec 1 qui est décalé y fois
  - exemple y=7 alors 0b1000 0000 ou 0x80 (le bit#7 est mis à 1)
- on fait un AND avec x, ce qui veut dire que tous les bits de x sont mis à 0, sauf le bit#7 qui conserve sa valeur (0 ou 1).
- deux cas de figure :
  - bit#7 de x vaut 0
    - la fonction foo retourne 0
  - bit#7 de x vaut 1
    - la fonction foo retourne 1

Cette fonction permet de tester si le y ème bit du nombre x est Set ou Reset.

Opérateurs logiques

```
&&      // Logical AND
||      // Logical OR
!       // Logical NOT
```

Attention

Historiquement, C ne possède pas de type Booléan (True or False)

- En C, 0 signifie "False", et non-zéro signifie "True"
- Ainsi 1 est True. Tout comme -37 est True et 3.1415 est True. Seul 0 est False.

Quand on débute en C, on fait souvent l'erreur d'oublier que -37 est équivalent à True sur une opération logique. Vigilence !

Une possibilité d'introduire une notion de booléen plus lisible est d'inclure `#include <stdbool.h>` On peut ainsi accéder à des noms symboliques de type true ou false, mais c'est uniquement de l'affichage, dans une opération logique en C, -54 sera toujours vue comme True...

```
#include <stdio.h>
#include <stdbool.h>
int main(void) {
  bool x = true;
  if (x) {
    printf("x is true!\n");
  }
}
```

Logical AND

De manière simple `&&` retourne vrai (1) si les deux côtés de l'expression sont différents de 0.

L'expression est évaluée de 'Gauche vers Droite'. Si une des partie de l'expression vaut ZERO - l'évaluation se termine.

Par exemple :

```
/* These all return TRUE (1) */

if (4 && 5) return();

i=3;
j=2;
return( i && j);
```

Exemple

```
//For example:
#include <stdio.h>

int main()
{
  int k=0;
int i=3;
int j=2;
if ( i-i && j++) k=1;
  printf("i=%d, j=%d, k=%d",i,j,k);
  return 0;
}
```

Solution

- Le côté gauche (i-i) donne 0, l'évaluation se termine, ce qui fait que j n'est pas postincrémentée et k n'est pas changé.
- i=3, j=2, k=0

WARNING : ici on sort directement de l'évaluation sans incrémenter j. C'est spécifique à &&

Si à la place de la condition `( i-i && j++)`, on avait `( i && j++)`

- le côté gauche est différent de 0, le côté droit est différent de 0, la condition de boucle est valide, on incrémente j, on affecte 1 à k
- i=3, j=3, k=1

Logical OR

De manière simple `||` retourne vrai (1) quand un des deux côté vaut vrai. OR s'évalue également de 'gauche à droite' et va stopper quand une expression returne true.

Exemple

```
//For example:
#include <stdio.h>

int main()
{
  int k=0;
int i=3;
int j=2;
if ( i-i || j++) k=1;
  printf("i=%d, j=%d, k=%d",i,j,k);
  return 0;
}
```

Solution

- Le côté gauche (i-i) donne 0, on évalue à droite j=2, l'opération || retourne True, la variable j est post-incrémentée, la boucle if est validée, et k=1
- i=3, j=3, k=1

Exemple avec subtilité

```
//For example:
#include <stdio.h>

int main()
{
  int k=0;
 int i=3;
 int j=0;
 if ( i-i || j++) k=1;
  printf("i=%d, j=%d, k=%d",i,j,k);
  return 0;
}
```

Solution

- Le côté gauche (i-i) donne 0, on évalue à droite j=0, l'opération || retourne False, la variable j n'est pas post-incrémentée, la boucle if n'est pas validée.
- i=3, j=0, k=0

Autre exemple avec subtilité

```
//For example:
#include <stdio.h>

int main()
{
  int k=0;
 int i=3;
 int j=0;
 if ( i-i || ++j) k=1;
  printf("i=%d, j=%d, k=%d",i,j,k);
  return 0;
}
```

Solution

- Le côté gauche (i-i) donne 0, on pré-incrémente à droite j, j vaut 1, l'opération || retourne True, la boucle if est validée.
- i=3, j=1, k=1

Logical NOT

NOT inverse l'état logique de son opérande. Si l'opérande est 0, 1 est retourné, sinon 0 est retouné. Si l'opérande est différent de 0, il est considéré comme True, et 0 est retourné.

```
!4 /* Returns 0 */
!-4 /* Returns 0 */
!1 /* Returns 0 */
!0 /* Returns 1 */
```

il ne faut pas confondre l'opérateur `~` et l'opérateur `!`

~ vs !

- Dans l'exemple ci-dessous, j'utilise le type uint8_t pour travailler sur 8 bits en non-signé
- Pour utiliser uint8_t, il faut inclure <stdint.h>
- Pour afficher le contenu d'un uint8_t dans un printf on utilise %hhu à la place de %d pour un int.

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

int main()
{
  uint8_t REG=0;
 printf("~REG = %hhu (Not bit à bit)\n",~REG);
 printf("!REG = %hhu (Not logique)\n",!REG);
  return 0;
}
```

Solution

- ~REG = 255 (Not bit à bit) car REG = 0 = 0b0000_0000 -> ~REG = 0b1111_1111 = 255
- !REG = 1 (Not logique) car REG = 0 et !REG = 1

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

int main()
{
```

```
  uint8_t REG=156;
printf("~REG = %hhu (Not bit à bit)\n",~REG);
printf("!REG = %hhu (Not logique)\n",!REG);
  return 0;
}
```

Solution

- ~REG = 99 (Not bit à bit) car REG = 156 = 0b1001_1100 -> ~REG = 0b0110_0011= 99
- !REG = 0 (Not logique) car !256 = 0

Opérateurs relationnels (de test)

```
==      // Egal à
!=      // Différend de
>       // supérieur
<       // inférieur
>=      // supérieur ou égal
<=      // inférieur ou égal
```

Une des erreurs classique en C est de confondre `= et ==`

- `i = i + 1;` correspond à une fonction d'affectation de valeur, on affecte la valeur i+1 à la variable i,
  - `i += 1;` correspond à l'écriture compacte,
  - `i++;` on parle également de post incrément de i,
- `if (i==9)` == correspond à une fonction de test, si i égale 9, alors on exécute le contenu de la boucle,

Post incrémentation et Pré incrémentation

Une des difficulté quand on débute, est de comprendre la différence entre une pré-incrémentation `++i` et une post-incrémentation `i++`

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

```
int main()
{
  int i=41;
  int k=0;
  if( 42 == i++) k=1;
  printf("i= %d, k= %d",i,k);
  return 0;
}
```

Le résultat de ce code est i=42 et k=0, alors que k devrait valoir 1 d'après le code.

Solution

- dans `( 42 == i++)` 42 est évalué, puis i, i=41, comme 42 n'est pas égale à 41, la boucle n'est pas évaluée, on post incrémente i qui passe à 42
- k reste à 0

```
#include <stdio.h>

int main()
{
  int i=41;
  int k=0;
  if( 42 == ++i) k=1;
  printf("i= %d, k= %d",i,k);
  return 0;
}
```

Le résultat de ce code est i=42 et k=1.

Solution

- dans `( 42 == i++)` 42 est évalué, puis on pré incrémente i, i=42, comme 42 est égale à 42, la boucle est évaluée
- k passe à 1

Expressions conditionnelles

On peut compacter les écritures de boucles if-else par des expressions conditionnelles.

```
// if - else  en C
  if ( x == 1 )
    y = 10;
  else
    y = 20;

// l'expression conditionnelle équivalente
  y = (x == 1) ? 10 : 20;
```

à droite, on évalue la première expression `(x ==1 )` et si elle est vraie, on évalue la seconde `10`. Si fausse, la troisième expression est évaluée `20`.

```
// if - else  en C
 if ( x == 1 )
   printf("take car");
 else
   printf("take bike");

// l'expression conditionnelle équivalente
 (x == 1) ? printf("take car") : printf("take
bike");
// ou bien
 printf( (x == 1) ? "take car" : "take bike");
```

Il paraîtrait que les structures conditionnelles soient plus efficaces au niveau de la compilation pour les sections de code critique en temps réel. On retrouve les expressions conditionnelles chez de nombreux programmeurs adeptes de la compacité du code au détriment de la lisibilité.

Revision #2
Created 5 July 2023 15:10:09 by Philippe Celka
Updated 5 July 2023 15:15:25 by Philippe Celka