Représentation des expressions en C 1

Salem BENFERHAT

Centre de Recherche en Informatique de Lens (CRIL-CNRS) email : benferhat@cril.fr

¹ Version préliminaire du cours. Tout retour sur la forme comme sur le fond est le bienvenu.

Moodle

- Les cours/TD/TP sont déposés sous Moodle juste après les cours.
- Les corrigés d'une partie des exercices des TDs ou TPs seront également déposés sous Moodle.
- Compilateurs C sous Windows.

La fonction printf est composée de deux parties :

■ Partie 1 obligatoire : une chaîne de caractères à imprimer (de gauche à droite). Pour chaque caractère c de la chaîne:

- Partie 1 obligatoire : une chaîne de caractères à imprimer (de gauche à droite). Pour chaque caractère c de la chaîne:
 - ▶ si $c \neq \$ et $c \neq \%$: on l'imprime tel quel;

- Partie 1 obligatoire : une chaîne de caractères à imprimer (de gauche à droite). Pour chaque caractère c de la chaîne:
 - ▶ si $c \neq \$ et $c \neq \%$: on l'imprime tel quel;
 - ▶ si $c = \setminus$, on imprime l'interprétation du caractère suivant (e.g., $\setminus n$)

- Partie 1 obligatoire : une chaîne de caractères à imprimer (de gauche à droite). Pour chaque caractère c de la chaîne:
 - ▶ si $c \neq \$ et $c \neq \%$: on l'imprime tel quel;
 - ▶ si $c = \setminus$, on imprime l'interprétation du caractère suivant (e.g., $\setminus n$)
 - si c = % on imprime la valeur donnée en argument. Le format d'impression est donné par le caractère qui suit %.

- Partie 1 obligatoire : une chaîne de caractères à imprimer (de gauche à droite). Pour chaque caractère c de la chaîne:
 - ▶ si $c \neq \$ et $c \neq \%$: on l'imprime tel quel;
 - ▶ si $c = \setminus$, on imprime l'interprétation du caractère suivant (e.g., $\setminus n$)
 - si c = % on imprime la valeur donnée en argument. Le format d'impression est donné par le caractère qui suit %.
 - Le "f" dans print signifie "formaté"

- Partie 1 obligatoire : une chaîne de caractères à imprimer (de gauche à droite). Pour chaque caractère c de la chaîne:
 - ▶ si $c \neq \$ et $c \neq \%$: on l'imprime tel quel;
 - ▶ si $c = \setminus$, on imprime l'interprétation du caractère suivant (e.g., $\setminus n$)
 - si c = % on imprime la valeur donnée en argument. Le format d'impression est donné par le caractère qui suit %.
 - Le "f" dans printf signifie "formaté"
- Partie 2 optionnelle : une liste d'arguments utilisée dans la partie
 1.

Remarque sur le Scanf

■ Même principe que le printif; mais au lieu d'imprimer la chaîne formatée, scanf la lit.

Remarque sur le Scanf

■ Même principe que le printif; mais au lieu d'imprimer la chaîne formatée, scanf la lit.

Remarque sur le Scanf

Même principe que le printif; mais au lieu d'imprimer la chaîne formatée, scanf la lit.

Dans un premier temps :

Utiliser uniquement des caractères "formatés" %x. (pas de caractères ordinaires)

Variables à valeurs entières

Par défaut : le type int

int nom_de_la_variable;

Variables à valeurs entières : deux questions à poser

Quelle est la nature des valeurs prises par la variable ?

- Des toutes petites valeurs (positives ou négatives) : char.
- Des petites valeurs : rajouter le préfixe short.
- Des valeurs ordinaires : pas de préfix.
 - Typiquement : le int est codé sur 4 octets. Les valeurs doivent être comprises entre [-2³¹, 2³¹ - 1].
- De grandes valeurs : rajouter le préfixe long.

Variables à valeurs entières : deux questions à poser

Est-ce que les valeurs prises par la variable sont toutes positives?

- Si oui : rajouter une autre préfixe unsigned à int ou à char.
- Si non, soit vous ne rajoutez rien soit vous rajoutez le préfixe signed.

Une dernière remarque

- Les deux déclarations suivantes :
 - unsigned short int et
 - short unsigned int
- sont équivalentes!

Opérateurs arithmétiques

■ Le C utilise les cinq opérateurs standards à deux arguments :

- Le C utilise les cinq opérateurs standards à deux arguments :
 - Addition (+).

- Le C utilise les cinq opérateurs standards à deux arguments :
 - ▶ Addition (+).
 - ► Soustraction (-).

- Le C utilise les cinq opérateurs standards à deux arguments :
 - ▶ Addition (+).
 - ► Soustraction (-).
 - ► Multiplication (*).

- Le C utilise les cinq opérateurs standards à deux arguments :
 - ▶ Addition (+).
 - Soustraction (-).
 - Multiplication (*).
 - Division (/).

- Le C utilise les cinq opérateurs standards à deux arguments :
 - ▶ Addition (+).
 - Soustraction (-).
 - Multiplication (*).
 - Division (/).
 - Reste de la division entière, appelé Modulo (%).

- Le C utilise les cinq opérateurs standards à deux arguments :
 - ▶ Addition (+).
 - Soustraction (-).
 - Multiplication (*).
 - ▶ Division (/).
 - Reste de la division entière, appelé Modulo (%).
- Les quatre premiers opérateurs s'appliquent sur tout type de données (char, int, float etc), alors que l'opérateur "%" ne s'applique que sur des entiers.

- Le C utilise les cinq opérateurs standards à deux arguments :
 - ▶ Addition (+).
 - ► Soustraction (-).
 - Multiplication (*).
 - ▶ Division (/).
 - Reste de la division entière, appelé Modulo (%).
- Les quatre premiers opérateurs s'appliquent sur tout type de données (char, int, float etc), alors que l'opérateur "%" ne s'applique que sur des entiers.
- Les opérateurs "+" et le "-" peuvent aussi être utilisés comme des opérateurs unaires (à un seul argument).

L'opérateur "/" appliqué sur deux valeurs entières donnerait un résultat entier : le résultat de la division euclidienne (ou entière).

- L'opérateur "/" appliqué sur deux valeurs entières donnerait un résultat entier : le résultat de la division euclidienne (ou entière).
- Par exemple, l'instruction :

```
printf (" %d \n", 32 / 3); afficherait (A votre avis?).
```

- L'opérateur "/" appliqué sur deux valeurs entières donnerait un résultat entier : le résultat de la division euclidienne (ou entière).
- Par exemple, l'instruction :

```
printf (" %d \n", 32 / 3);
```

afficherait la valeur 10.

L'opérateur de division "/" appliqué sur deux valeurs flottantes donnerait un résultat flottant.

- L'opérateur de division "/" appliqué sur deux valeurs flottantes donnerait un résultat flottant.
- Par exemple, l'instruction :

```
printf (" %f \n", 32. / 3.);
```

afficherait la valeur 10.666667.

Python utilise deux types d'opérateurs / et // pour distinguer la division réelle de la division entière.

- Python utilise deux types d'opérateurs / et // pour distinguer la division réelle de la division entière.
- En Langage C seul l'opérateur "/" est utilisé.

- Python utilise deux types d'opérateurs / et // pour distinguer la division réelle de la division entière.
- En Langage C seul l'opérateur "/" est utilisé.
 - Ce sont les composants de la division qui déterminent la nature du résultat de la division.

Remarques sur l'opérateur modulo %

Remarques sur le modulo %

■ Le modulo ne s'applique que sur des valeurs entières.

Remarques sur le modulo %

- Le modulo ne s'applique que sur des valeurs entières.
- Dans la plupart des compilateurs du Langage C, l'opérateur

représente le reste de la division de "a" par "b".

Remarques sur le modulo %

- Le modulo ne s'applique que sur des valeurs entières.
- Dans la plupart des compilateurs du Langage C, l'opérateur

représente le reste de la division de "a" par "b".

■ C'est-à-dire, pour calculer a % b:

- Le modulo ne s'applique que sur des valeurs entières.
- Dans la plupart des compilateurs du Langage C, l'opérateur

représente le reste de la division de "a" par "b".

- C'est-à-dire, pour calculer a % b:
 - on calcule dans un premier temps le résultat de la division entière :

$$k = a/b$$

- Le modulo ne s'applique que sur des valeurs entières.
- Dans la plupart des compilateurs du Langage C, l'opérateur

représente le reste de la division de "a" par "b".

- C'est-à-dire, pour calculer a % b:
 - on calcule dans un premier temps le résultat de la division entière :

$$k = a/b$$

ensuite, on prend le reste de la division pour obtenir le modulo :

$$a \% b = a - k * b$$

■ De ce fait, en présence de valeurs négatives, l'opérateur "%" ne se comporte pas **toujours** comme un modulo.

- De ce fait, en présence de valeurs négatives, l'opérateur "%" ne se comporte pas **toujours** comme un modulo.
- C'est-à-dire, dans des compilateurs C, le signe du résultat de la division par des nombres négatifs peut-être négatif.

- De ce fait, en présence de valeurs négatives, l'opérateur "%" ne se comporte pas **toujours** comme un modulo.
- C'est-à-dire, dans des compilateurs C, le signe du résultat de la division par des nombres négatifs peut-être négatif.
- Par exemple, en division réelle, -8/6 donnerait -1.333...

- De ce fait, en présence de valeurs négatives, l'opérateur "%" ne se comporte pas **toujours** comme un modulo.
- C'est-à-dire, dans des compilateurs C, le signe du résultat de la division par des nombres négatifs peut-être négatif.
- Par exemple, en division réelle, -8/6 donnerait -1.333...
- En division entière, deux options :

- De ce fait, en présence de valeurs négatives, l'opérateur "%" ne se comporte pas **toujours** comme un modulo.
- C'est-à-dire, dans des compilateurs C, le signe du résultat de la division par des nombres négatifs peut-être négatif.
- Par exemple, en division réelle, -8/6 donnerait -1.333...
- En division entière, deux options :
 - Le résultat est égal à -1 et le reste est égal à -2.

- De ce fait, en présence de valeurs négatives, l'opérateur "%" ne se comporte pas **toujours** comme un modulo.
- C'est-à-dire, dans des compilateurs C, le signe du résultat de la division par des nombres négatifs peut-être négatif.
- Par exemple, en division réelle, -8/6 donnerait -1.333...
- En division entière, deux options :
 - ▶ Le résultat est égal à -1 et le reste est égal à -2.
 - ▶ Le résultat est égal à -2 et le reste est égal à 4.

■ Par exemple, l'instruction suivante :

```
printf ("-8 %% 6 = %d \n", -8 % 6);
```

donnerait :

à votre avis?

■ Par exemple, l'instruction suivante :

```
printf ("-8 %% 6 = %d \n", -8 % 6);
```

- donnerait le reste négatif -2. En effet :
 - Le résultat de la division entière de k = -8/6 est k = -1

Par exemple, l'instruction suivante :

```
printf ("-8 %% 6 = %d \n", -8 % 6);
```

- donnerait le reste négatif -2. En effet :
 - Le résultat de la division entière de k = -8/6 est k = -1
 - De ce fait:

$$-8\%6 = -8 - (-1 * 6) = -2$$

Par exemple, l'instruction suivante :

```
printf ("-8 %% 6 = %d \n", -8 % 6);
```

- donnerait le reste négatif -2. En effet :
 - Le résultat de la division entière de k = -8/6 est k = -1
 - De ce fait:

$$-8\%6 = -8 - (-1 * 6) = -2$$

Alors que le -8 modulo 6 donnerait 4!

■ De même, la séquence des deux instructions suivantes :

```
printf ("8 %% -5 = %d \n", 8 % -5);
printf ("-5 %% -3 = %d \n", -5 % -3);
```

afficherait respectivement à votre avis ?

■ De même, la séquence des deux instructions suivantes :

```
printf ("8 %% -5 = %d \n", 8 % -5);
printf ("-5 %% -3 = %d \n", -5 % -3);
```

afficherait respectivement 3 (8=-1x-5+3) et -2 (-5=1x-3-2).

■ Le modulo est défini à partir du résultat de la division.

- Le modulo est défini à partir du résultat de la division.
- Supposons que *a* et *b* sont deux entiers relatifs (de type int).

- Le modulo est défini à partir du résultat de la division.
- Supposons que *a* et *b* sont deux entiers relatifs (de type int).
- Soit k le résultat de la division entière de *a* par *b* :

$$k = a/b$$

- Le modulo est défini à partir du résultat de la division.
- Supposons que *a* et *b* sont deux entiers relatifs (de type int).
- Soit k le résultat de la division entière de *a* par *b* :

$$k = a/b$$

Alors:

$$a \% b = a - k * b$$

■ Attention : le définition de la division par un nombre négatif n'est pas la même en C et en Python

- Attention : le définition de la division par un nombre négatif n'est pas la même en C et en Python
- Avec des valeurs négatives, le C arrondit vers la valeur supérieure alors que Python vers la valeur inférieure. Par exemple :

- Attention : le définition de la division par un nombre négatif n'est pas la même en C et en Python
- Avec des valeurs négatives, le C arrondit vers la valeur supérieure alors que Python vers la valeur inférieure. Par exemple :
 - ► En langage C, 6/-4=-1

- Attention : le définition de la division par un nombre négatif n'est pas la même en C et en Python
- Avec des valeurs négatives, le C arrondit vers la valeur supérieure alors que Python vers la valeur inférieure. Par exemple :
 - ► En langage C, 6/-4=-1
 - ▶ alors qu'en python, 6//-4=-2.

- Attention : le définition de la division par un nombre négatif n'est pas la même en C et en Python
- Avec des valeurs négatives, le C arrondit vers la valeur supérieure alors que Python vers la valeur inférieure. Par exemple :
 - ► En langage C, 6/-4=-1
 - ▶ alors qu'en python, 6//-4=-2.
- De ce fait,

- Attention : le définition de la division par un nombre négatif n'est pas la même en C et en Python
- Avec des valeurs négatives, le C arrondit vers la valeur supérieure alors que Python vers la valeur inférieure. Par exemple :
 - ► En langage C, 6/-4=-1
 - ▶ alors qu'en python, 6//-4=-2.
- De ce fait,
 - le modulo du langage C et celui utilisé en Python diffèrent sur les résultats négatives.

- Attention : le définition de la division par un nombre négatif n'est pas la même en C et en Python
- Avec des valeurs négatives, le C arrondit vers la valeur supérieure alors que Python vers la valeur inférieure. Par exemple :
 - ► En langage C, 6/-4=-1
 - ▶ alors qu'en python, 6//-4=-2.
- De ce fait,
 - le modulo du langage C et celui utilisé en Python diffèrent sur les résultats négatives.
 - En réalité, Python utilise ce qui est appelée "la floor division" (division réelle, suivie d'une opération d'arrondi).

■ Le langage C offre la possibilité de travailler directement sur la représentation binaire des entiers (char et int).

- Le langage C offre la possibilité de travailler directement sur la représentation binaire des entiers (char et int).
- Ces opérateurs ne s'appliquent pas sur les variables de type float ou double.

Le langage C offre les opérateurs bits à bits suivants :

Opérateurs logiques bits à bits

complément à 1 (changer les 0 en 1 et les 1 en 0).

Le langage C offre les opérateurs bits à bits suivants :

Opérateurs logiques bits à bits

- complément à 1 (changer les 0 en 1 et les 1 en 0).
- & : "et" logique appliqué bit par bit.

Le langage C offre les opérateurs bits à bits suivants :

Opérateurs logiques bits à bits

- complément à 1 (changer les 0 en 1 et les 1 en 0).
- & : "et" logique appliqué bit par bit.
- | : "ou" logique appliqué bit par bit.

Le langage C offre les opérateurs bits à bits suivants :

Opérateurs logiques bits à bits

- complément à 1 (changer les 0 en 1 et les 1 en 0).
- & : "et" logique appliqué bit par bit.
- | : "ou" logique appliqué bit par bit.
- `: "xor" ou bien le **ou exclusif** toujours appliqué bit par bit.

Le langage C offre les opérateurs bits à bits suivants :

Opérateurs de décalage de bits

■ a << n : décalage (de n-bits) à gauche de la variable entière a.

Le langage C offre les opérateurs bits à bits suivants :

Opérateurs de décalage de bits

- a << n : décalage (de n-bits) à gauche de la variable entière a.
 - ▶ On ignore les *n*-bits les plus forts (se trouvant à gauche).

Le langage C offre les opérateurs bits à bits suivants :

Opérateurs de décalage de bits

- a << n : décalage (de n-bits) à gauche de la variable entière a.
 - ▶ On ignore les *n*-bits les plus forts (se trouvant à gauche).
 - ► On insère *n* "0" dans les positions les plus faibles (le plus à droite).

Le langage C offre les opérateurs bits à bits suivants :

Opérateurs de décalage de bits

- a << n : décalage (de n-bits) à gauche de la variable entière a.
 - ► On ignore les *n*-bits les plus forts (se trouvant à gauche).
 - ▶ On insère *n* "0" dans les positions les plus faibles (le plus à droite).
- a >> n : décalage (de n-bits) à droite de la variable entière a.

Opérateurs bits à bits

Le langage C offre les opérateurs bits à bits suivants :

Opérateurs de décalage de bits

- a << n : décalage (de n-bits) à gauche de la variable entière a.
 - ▶ On ignore les *n*-bits les plus forts (se trouvant à gauche).
 - ▶ On insère *n* "0" dans les positions les plus faibles (le plus à droite).
- a >> n : décalage (de n-bits) à droite de la variable entière a.
 - ▶ On ignore les n-bits les plus faible (se trouvant à droite).

Opérateurs bits à bits

Le langage C offre les opérateurs bits à bits suivants :

Opérateurs de décalage de bits

- a << n : décalage (de n-bits) à gauche de la variable entière a.
 - ▶ On ignore les *n*-bits les plus forts (se trouvant à gauche).
 - ▶ On insère *n* "0" dans les positions les plus faibles (le plus à droite).
- a >> n : décalage (de n-bits) à droite de la variable entière a.
 - ▶ On ignore les *n*-bits les plus faible (se trouvant à droite).
 - On insère *n* "0" dans les positions les plus fortes (le plus à gauche).

Opérateurs bits à bits

Le langage C offre les opérateurs bits à bits suivants :

Opérateurs de décalage de bits

- a << n : décalage (de n-bits) à gauche de la variable entière a.
 - ▶ On ignore les *n*-bits les plus forts (se trouvant à gauche).
 - ▶ On insère *n* "0" dans les positions les plus faibles (le plus à droite).
- a >> n : décalage (de n-bits) à droite de la variable entière a.
 - ► On ignore les *n*-bits les plus faible (se trouvant à droite).
 - ▶ On insère n "0" dans les positions les plus fortes (le plus à gauche).
- N'utiliser pas le décalage avec un *n* négatif!

opérateurs bits à bits : exemple

opérateurs bits à bits : exemple

■ Considérons la séquence d'instructions suivante :

```
1  unsigned char i, j;
2  i=125;
3  i=i>>2;
4  printf ("i=%d \n", i);
5  i=i<<1;
6  printf ("i=%d \n", i);
7  j=3;
8  printf ("i&j=%d, i|j=%d, \n", i&j, i|j);</pre>
```

■ Donner le résultat de l'exécution de cette séquence d'instructions. Justifier les résultats obtenus.

opérateurs bits à bits : exemple

Considérons la séquence d'instructions suivante :

```
1   unsigned char i, j;
2   i=125;
3   i=i>>2;
4   printf ("i=%d \n", i);
5   i=i<<1;
6   printf ("i=%d \n", i);
7   j=3;
8   printf ("i&j=%d, i|j=%d, \n", i&j, i|j);</pre>
```

L'exécution de cette séquence donne :

| Instruction | valeur de la variable i | | | | | |
|-------------------|-------------------------|--|--|--|--|--|
| i=125; | 125 | | | | | |
| i= <i>i</i> >>2; | 31 | | | | | |
| i= <i>i</i> <<1; | 62 | | | | | |
| i & j (avec j=11) | 2 | | | | | |
| i (avec j=11) | 63 | | | | | |

Justifier ces résultats

■ Comme *i* est déclaré en unsigned char, il est représenté sur 8 bits et a comme domaine de valeurs [0, 255].

- Comme *i* est déclaré en unsigned char, il est représenté sur 8 bits et a comme domaine de valeurs [0, 255].
- Après l'instruction "i=125", en binaire, i a comme valeur :



■ Comme *i* est déclaré en unsigned char, il est représenté sur 8 bits et a comme domaine de valeurs [0, 255].

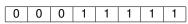
- Comme *i* est déclaré en unsigned char, il est représenté sur 8 bits et a comme domaine de valeurs [0, 255].
- Après l'instruction "i=125", en binaire, i a comme valeur :



- Comme *i* est déclaré en unsigned char, il est représenté sur 8 bits et a comme domaine de valeurs [0, 255].
- Après l'instruction "i=125", en binaire, i a comme valeur :

| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

■ Le décalage à droite de deux positions, *i* >>2, donnerait :



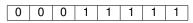
qui est égal à 31 en décimal.

■ L'instruction, *i* >>2, a donné :

| Γ | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|
| _ | | | | | | | | |

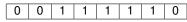
qui est égal à 31 en décimal.

■ L'instruction, *i* >>2, a donné :



qui est égal à 31 en décimal.

■ Le décalage à gauche d'une position, $i \ll 1$, donnerait :

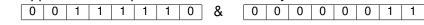


qui est égal à 62 en décimal.

L'application de l'opérateur bit à bit & "conjonction bit à bit" donne :

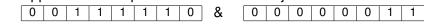
| 0 0 1 1 1 1 1 0 & | 0 & | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | |
|-----------------------------------|-------|---|---|---|---|---|---|---|---|--|
|-----------------------------------|-------|---|---|---|---|---|---|---|---|--|

L'application de l'opérateur bit à bit & "conjonction bit à bit" donne :



qui est égal à 2.

L'application de l'opérateur bit à bit & "conjonction bit à bit" donne :

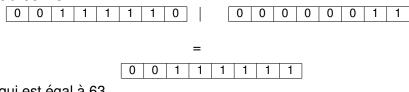


qui est égal à 2.

■ Finalement, l'application de l'opérateur bit à bit | "disjonction bit à bit" donne :

| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

■ Finalement, l'application de l'opérateur bit à bit | "disjonction bit à bit" donne :



qui est égal à 63.

Un autre exercice : Opérations bits à bit

En utilisant les opérations bits à bit, écrire un programme C qui :

- lit un entier postif et
- affiche le troisième bit, de poids le plus faible, de cet entier.

Par exemple, si le nombre lu est 12 le programme affichera à votre avis ?.

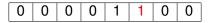
Un autre exercice : Opérations bits à bit

En utilisant les opérations bits à bit, écrire un programme C qui :

- lit un entier postif et
- affiche le troisième bit, de poids le plus faible, de cet entier.

Par exemple, si le nombre lu est 12 le programme affichera 1.

En effet, l'écriture binaire du nombre décimal 12 donne :



La première étape consiste à se débarrasser des deux premiers bits :



La première étape consiste à se débarrasser des deux premiers bits :



Cette étape est implémentée avec l'instruction suivante (a est modifiée) :

$$a = a >> 2$$
; ou encore $a >>= 2$;

La première étape consiste à se débarrasser des deux premiers bits :



Cette étape est implémentée avec l'instruction suivante (a est modifiée) :

$$a = a >> 2$$
; ou encore $a >> = 2$;

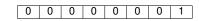
ou encore (a n'est pas modifiée) :

$$a \gg 2$$

■ La deuxième étape consiste à appliquer un "&" avec le nombre 1 :



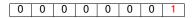




■ La deuxième étape consiste à appliquer un "&" avec le nombre 1 :



Le résultat obtenu est :



■ La deuxième étape consiste à appliquer un "&" avec le nombre 1 :



Le résultat obtenu est :

Ces deux étapes s'implémentent avec l'instruction suivante :

$$(a >> 2)\&1;$$

```
#include <stdio.h>
int main (void)
{
   unsigned int a;
   printf ("Merci d'introduire un entier positif : ");
   scanf("%u", &a);
   printf ("\n Le troisième bit de a est : %d.\n", (a>>2) & 1);
   return(0);
}
```

■ Combiner la valeur de "a" avec le nombre 4 (en binaire 100)

- Combiner la valeur de "a" avec le nombre 4 (en binaire 100)
 - Le résultat est "0" ou A votre avis

■ Combiner la valeur de "a" avec le nombre 4 (en binaire 100)

- Combiner la valeur de "a" avec le nombre 4 (en binaire 100)
 - Le résultat est "0" ou 4.

- Combiner la valeur de "a" avec le nombre 4 (en binaire 100)
 - Le résultat est "0" ou 4.
- Décaler le résultat vers la gauche de deux positions.

```
#include <stdio.h>
int main (void)
{
  unsigned int a;
  printf ("Merci d'introduire un entier positif : ");
  scanf("%u", &a);
  printf ("\n Avec une autre solution, on obtient : %d.\n", (a&4)>>2);
  return(0);
}
```

Un exercice un peu difficile

Un autre exercice : un peu difficle

```
#include <stdio.h>
int main (void)
{
    unsigned char a;
    printf ("\n Merci de saisir un petit entier positif");
    printf (" compris entre 0 et 255 : ");
    scanf("%hhu", &a);
    a = (a&0x55) + ((a>>1) &0x55);
    a = (a&0x33) + ((a>>2) &0x33);
    a = (a&0x30) + ((a>>4) &0x00);
    printf ("xxxxxxx %d \n", a);
    return(0);
}
```

Opérateurs de comparaison

> (strictement supérieur).

- > (strictement supérieur).
- >= (supérieur ou égal).

- > (strictement supérieur).
- >= (supérieur ou égal).
- < (strictement inférieur).</p>

- > (strictement supérieur).
- >= (supérieur ou égal).
- < (strictement inférieur).</p>
- <= (inférieur ou égal).</p>

- > (strictement supérieur).
- >= (supérieur ou égal).
- < (strictement inférieur).</p>
- <= (inférieur ou égal).</p>
- == (égal).

- > (strictement supérieur).
- >= (supérieur ou égal).
- < (strictement inférieur).</p>
- <= (inférieur ou égal).</p>
- == (égal).
- ! = (différent).

- > (strictement supérieur).
- >= (supérieur ou égal).
- < (strictement inférieur).</p>
- <= (inférieur ou égal).</p>
- == (égal).
- ! = (différent).

- > (strictement supérieur).
- >= (supérieur ou égal).
- < (strictement inférieur).</p>
- <= (inférieur ou égal).</p>
- == (égal).
- ! = (différent).

Remarque

La valeur entière (intuitivement booléenne) retournée à l'issue de chaque comparaison est :

- 0 : si la relation est fausse.
- 1 : si la relation est vraie.

Rappels

 Il n'existe pas de type booléen propre en langage C (contrairement aux autres langages),

Rappels

- Il n'existe pas de type booléen propre en langage C (contrairement aux autres langages),
- Par convention (même interprétation en Python) :

Rappels

- Il n'existe pas de type booléen propre en langage C (contrairement aux autres langages),
- Par convention (même interprétation en Python) :
 - le 0 (chaîne vide, pointeur NULL, etc.) est interprété comme faux et

Rappels

- Il n'existe pas de type booléen propre en langage C (contrairement aux autres langages),
- Par convention (même interprétation en Python) :
 - le 0 (chaîne vide, pointeur NULL, etc.) est interprété comme faux et
 - le reste des valeurs est interprété comme vrai.

■ && (ET).

- && (ET).
 - La valeur retournée est de "0" si la valeur de l'un des deux composants du ET est égale à "0".

- && (ET).
 - La valeur retournée est de "0" si la valeur de l'un des deux composants du ET est égale à "0".
 - Sinon la valeur "1" est retournée.

- && (ET).
 - La valeur retournée est de "0" si la valeur de l'un des deux composants du ET est égale à "0".
 - Sinon la valeur "1" est retournée.
- || (OU).

- && (ET).
 - La valeur retournée est de "0" si la valeur de l'un des deux composants du ET est égale à "0".
 - Sinon la valeur "1" est retournée.
- || (OU).
 - La valeur retournée est de "0" si la valeur des deux composants du OU est égale à "0".

- && (ET).
 - La valeur retournée est de "0" si la valeur de l'un des deux composants du ET est égale à "0".
 - Sinon la valeur "1" est retournée.
- || (OU).
 - La valeur retournée est de "0" si la valeur des deux composants du OU est égale à "0".
 - Sinon la valeur "1" est retournée.

- && (ET).
 - La valeur retournée est de "0" si la valeur de l'un des deux composants du ET est égale à "0".
 - Sinon la valeur "1" est retournée.
- || (OU).
 - La valeur retournée est de "0" si la valeur des deux composants du OU est égale à "0".
 - Sinon la valeur "1" est retournée.
- ! (Négation).

- && (ET).
 - La valeur retournée est de "0" si la valeur de l'un des deux composants du ET est égale à "0".
 - Sinon la valeur "1" est retournée.
- || (OU).
 - La valeur retournée est de "0" si la valeur des deux composants du OU est égale à "0".
 - Sinon la valeur "1" est retournée.
- ! (Négation).
 - La valeur retournée est de "1" si la valeur du composant du NON est égale à "0".

- && (ET).
 - La valeur retournée est de "0" si la valeur de l'un des deux composants du ET est égale à "0".
 - Sinon la valeur "1" est retournée.
- || (OU).
 - La valeur retournée est de "0" si la valeur des deux composants du OU est égale à "0".
 - Sinon la valeur "1" est retournée.
- ! (Négation).
 - La valeur retournée est de "1" si la valeur du composant du NON est égale à "0".
 - Sinon la valeur "0" est retournée.

Opérateurs bits à bit et opérateurs booléens

■ Ne confondez pas : & et &&.

- Ne confondez pas : & et &&.
 - L'opérateur & est appliqué (bit à bit) sur la représentation binaire des données entières.

- Ne confondez pas : & et &&.
 - L'opérateur & est appliqué (bit à bit) sur la représentation binaire des données entières.
 - L'opérateur && est appliqué sur deux expressions booléennes.

- Ne confondez pas : & et &&.
 - L'opérateur & est appliqué (bit à bit) sur la représentation binaire des données entières.
 - L'opérateur && est appliqué sur deux expressions booléennes.
- De même, ne confondez pas : | et ||

Remarques: exemple

Qu'afficherait le programme suivant :

```
#include <stdio.h>
int main (void)
{
  unsigned int a, b;
  printf ("Merci d'introduire deux entier positifs : ");
  scanf("%u%u", &a, &b);
  printf ("\n Voici les valeurs rentrées : a=%u et b=%u.\n", a, b);
  printf ("\n Affichage 1 : %d.\n", a & b);
  printf ("\n Affichage 2 : %d.\n", a && b);
  return(0);
}
```

si l'utilisateur rentre les valeurs suivantes :

- **124**
- **8** 5

Remarques: exemple

```
#include <stdio.h>
int main (void)
{
  unsigned int a, b;
  printf ("Merci d'introduire deux entier positifs : ");
  scanf("%u&u", &a, &b);
  printf ("\n Voici les valeurs rentrées : a=%u et b=%u.\n", a, b);
  printf ("\n Affichage 1 : %d.\n", a & b);
  printf ("\n Affichage 2 : %d.\n", a && b);
  return(0);
}
```

Avec les valeurs "12 4" le programme affiche :

- > Voici les valeurs rentrées : a=12 et b=4.
- > Affichage 1:4.
- > Affichage 2:1.

Remarques: exemple

```
#include <stdio.h>
int main (void)
{
   unsigned int a, b;
   printf ("Merci d'introduire deux entier positifs : ");
   scanf("%u&u", &a, &b);
   printf ("\n Voici les valeurs rentrées : a=%u et b=%u.\n", a, b);
   printf ("\n Affichage 1 : %d.\n", a & b);
   printf ("\n Affichage 2 : %d.\n", a && b);
   return(0);
}
```

Avec les valeurs "8 5" le programme affiche :

- > Voici les valeurs rentrées : a=8 et b=5.
- > Affichage 1:0.
- > Affichage 2 : 1.

Opérateurs d'affectations

Les opérations d'affectations simples sont de la forme :

²Une erreur de syntaxe très courante.

Les opérations d'affectations simples sont de la forme :

οù

var est une variable (L-value).

²Une erreur de syntaxe très courante.

Les opérations d'affectations simples sont de la forme :

- var est une variable (L-value).
- "=" est le symbole d'affectation (à ne pas confondre avec "==" ²).

²Une erreur de syntaxe très courante.

Les opérations d'affectations simples sont de la forme :

- var est une variable (L-value).
- "=" est le symbole d'affectation (à ne pas confondre avec "==" 2).
- "expression" est une expression arithmétique (ou booléenne) impliquant des variables (simples ou complexes) et/ou des fonctions.

²Une erreur de syntaxe très courante.

Les opérations d'affectations simples sont de la forme :

- var est une variable (L-value).
- "=" est le symbole d'affectation (à ne pas confondre avec "==" 2).
- "expression" est une expression arithmétique (ou booléenne) impliquant des variables (simples ou complexes) et/ou des fonctions.
- Le point-virgule ";" est utilisé pour indiquer la fin de l'instruction.

²Une erreur de syntaxe très courante.

■ Le langage C offre la possibilité d'avoir des affectations en cascade :

■ Le langage C offre la possibilité d'avoir des affectations en cascade :

Par exemple,

$$a = b = 1 + d$$
;

■ Le langage C offre la possibilité d'avoir des affectations en cascade :

Par exemple,

$$a = b = 1 + d$$
;

▶ Dans un premier temps, l'affectation b = 1 + d est exécutée.

■ Le langage C offre la possibilité d'avoir des affectations en cascade :

Par exemple,

$$a = b = 1 + d$$
:

- ▶ Dans un premier temps, l'affectation b = 1 + d est exécutée.
- ► Ensuite, l'affectation *a* = *b* est exécutée.

Le langage C offre aussi la possibilité d'avoir des séquences d'expressions.

- Le langage C offre aussi la possibilité d'avoir des séquences d'expressions.
- Chaque expression est séparée par une virgule ",".

- Le langage C offre aussi la possibilité d'avoir des séquences d'expressions.
- Chaque expression est séparée par une virgule ",".
- La séquence d'expressions se termine par contre par un point virgule.

- Le langage C offre aussi la possibilité d'avoir des séquences d'expressions.
- Chaque expression est séparée par une virgule ",".
- La séquence d'expressions se termine par contre par un point virgule.
- Voici un exemple de séquence d'expressions :

$$a = 2, b = a + 2;$$

- Le langage C offre aussi la possibilité d'avoir des séquences d'expressions.
- Chaque expression est séparée par une virgule ",".
- La séquence d'expressions se termine par contre par un point virgule.
- Voici un exemple de séquence d'expressions :

$$a = 2, b = a + 2;$$

L'évaluation des expressions se fait de gauche à droite.

Les affectations compactes sont de la forme :

var operateur = expression;

Les affectations compactes sont de la forme :

```
var operateur = expression;
```

L'instruction compacte d'affectation suivante :

est au fait équivalente l'instruction "classique" :

Les affectations compactes sont de la forme :

```
var operateur = expression;
```

L'instruction compacte d'affectation suivante :

Les instructions d'affectation compactes sont :

Les affectations compactes sont de la forme :

L'instruction compacte d'affectation suivante :

$$var = var + expression;$$

- Les instructions d'affectation compactes sont :
 - Opérations compactes arithmétiques :

Les affectations compactes sont de la forme :

L'instruction compacte d'affectation suivante :

est au fait équivalente l'instruction "classique" :

- Les instructions d'affectation compactes sont :
 - Opérations compactes arithmétiques :

Opérations compactes binaires :

Opérateurs d'incrémentation

■ Le langage C offre une méthode compacte et efficace d'incrémenter une variable grâce à l'opérateur "++" qui peut être en préfixe ou en suffixe d'une variable.

Opérateurs d'incrémentation

 Cet opérateur peut s'appliquer sur une variable seule, comme par exemple

```
i++;
```

ou encore à l'intérieur d'une expression, comme :

```
i=5;
c=++i - 2;
d=i++ - 2;
```

- Lorsque l'opérateur d'incrémentation est utilisé en préfixe dans une expression, alors :
 - la variable concernée est augmentée de "1" avant l'évaluation de l'expression.

- Lorsque l'opérateur d'incrémentation est utilisé en préfixe dans une expression, alors la variable concernée est augmentée de "1" avant l'évaluation de l'expression.
- Question: Qu'affiche le programme suivant ?

```
i=5;
c=++i - 2;
```

- Lorsque l'opérateur d'incrémentation est utilisé en préfixe dans une expression, alors la variable concernée est augmentée de "1" avant l'évaluation de l'expression.
- Ainsi, la séquence suivante :

```
i=5;
c=++i - 2;
est équivalente à:
i=5;
i=i+1;
c=i - 2;
```

Et la valeur retournée est égale à 4.

- Lorsque l'opérateur d'incrémentation est utilisé en suffixe dans une expression, alors la variable concernée est augmentée de "1" après l'évaluation de l'expression.
- Ainsi, la séquence suivante :

```
i=5;
c=i++ - 2;
est équivalente à:
i=5;
c=i - 2;
i=i+1;
```

Et la valeur retournée est égale à 3.

Opérateurs de décrémentation

- De manière symétrique, on peut utiliser l'opérateur de décrémentation :
 - en préfixe d'une variable, et
 - en suffixe d'une variable.

Remarque

Les opérateurs d'incrémentation et de décrémentation ne s'appliquent pas aux expressions. Par exemple, l'instruction suivante :

$$a = (b+c)++;$$

n'est pas valide.

■ Une expression arithmétique (ou booléenne) est dite mixte si elle implique des variables ayant des types différents.

- Une expression arithmétique (ou booléenne) est dite mixte si elle implique des variables ayant des types différents.
- Dans ce cas, des opérations de conversion sont utilisées.

- Une expression arithmétique (ou booléenne) est dite mixte si elle implique des variables ayant des types différents.
- Dans ce cas, des opérations de conversion sont utilisées.
- On distingue deux types de conversion :

- Une expression arithmétique (ou booléenne) est dite mixte si elle implique des variables ayant des types différents.
- Dans ce cas, des opérations de conversion sont utilisées.
- On distingue deux types de conversion :
 - Conversions explicites données par l'utilisateur.

- Une expression arithmétique (ou booléenne) est dite mixte si elle implique des variables ayant des types différents.
- Dans ce cas, des opérations de conversion sont utilisées.
- On distingue deux types de conversion :
 - Conversions explicites données par l'utilisateur.
 - Conversions implicites utilisées par le compilateur.

Voici quelques règles de conversions implicites :

■ La première règle explicite est donnée par le schéma suivant :



Voici quelques règles de conversions implicites :

■ La première règle explicite est donnée par le schéma suivant :



Lorsqu'une expression implique une variable de type char ou short, celle-ci est systématiquement "promue" en int.

Voici quelques règles de conversions implicites :

La première règle explicite est donnée par le schéma suivant :



- Lorsqu'une expression implique une variable de type char ou short, celle-ci est systématiquement "promue" en int.
- Lorsqu'un opérateur (arithmétique ou de comparison) implique une variable signée et une variable non-signée, une conversion vers le type non-signé est effectuée.

Voici quelques règles de conversions implicites :

■ La première règle explicite est donnée par le schéma suivant :



- Lorsqu'une expression implique une variable de type char ou short, celle-ci est systématiquement "promue" en int.
- Lorsqu'un opérateur (arithmétique ou de comparison) implique une variable signée et une variable non-signée, une conversion vers le type non-signé est effectuée.
- Les conversions implicites peuvent conduire à des résultats inattendus (voir TD/TP). Mieux vaut utiliser des conversions explicites.

Sa syntaxe est:

(type_à_convertir) expression

Son effet est de convertir le résultat de l'expression vers type_à_convertir.

Sa syntaxe est:

(type_à_convertir) expression

- Son effet est de convertir le résultat de l'expression vers type_à_convertir.
- Par exemple, si b est de type float et a est de type int, alors :
 a=(int) b
 permet de récupérer la partie entière de b.

Sa syntaxe est:

(type_à_convertir) expression

- Son effet est de convertir le résultat de l'expression vers type_à_convertir.
- Par exemple, si b est de type float et a est de type int, alors :
 a=(int) b
 permet de récupérer la partie entière de b.
- Il est important de noter que l'opération cast est appliquée à l'expression et non aux variables utilisées dans l'expression.

Sa syntaxe est:

(type_à_convertir) expression

- Son effet est de convertir le résultat de l'expression vers type_à_convertir.
- Par exemple, si b est de type float et a est de type int, alors :
 a=(int) b
 permet de récupérer la partie entière de b.
- Il est important de noter que l'opération cast est appliquée à l'expression et non aux variables utilisées dans l'expression.

Sa syntaxe est:

(type_à_convertir) expression

- Son effet est de convertir le résultat de l'expression vers type_à_convertir.
- Par exemple, si b est de type float et a est de type int, alors :
 a=(int) b
 permet de récupérer la partie entière de b.
- Il est important de noter que l'opération cast est appliquée à l'expression et non aux variables utilisées dans l'expression.

Remarques

Il est recommandé d'utiliser au maximum la conversion explicite dans des expressions mixtes.