Variables en C

Lycée Thiers

Crédits

- Variables : Comment ça marche ou Openclassroom

Crédits

- Variables : Comment ça marche ou Openclassroom
- Sur le formattage, un tuto du lycée des Arts et métiers du Luxembourg ou encore Wikipedia

Représentation grossière de la mémoire

- En simplifiant abusivement, la RAM peut être vue comme un tableau à deux entrées `Adresse` et `Valeur` :

Adresse	Valeur
0	12
110	12.35
⋮	⋮
adresse max	123.56

Représentation grossière de la mémoire

- En simplifiant abusivement, la RAM peut être vue comme un tableau à deux entrées `Adresse` et `Valeur` :

Adresse	Valeur
0	12
110	12.35
⋮	⋮
adresse max	123.56

- L'adresse la plus petite est 0, l'adresse maximale dépend de la quantité de RAM dans l'ordinateur.

Représentation grossière de la mémoire

- En simplifiant abusivement, la RAM peut être vue comme un tableau à deux entrées `Adresse` et `Valeur` :

Adresse	Valeur
0	12
110	12.35
⋮	⋮
adresse max	123.56

- L'adresse la plus petite est 0, l'adresse maximale dépend de la quantité de RAM dans l'ordinateur.
- Il s'agit d'un tableau de nombre, écrits ici en base 10, mais en fait plutôt codés en binaire. Pour la machine, tout n'est finalement qu'une suite de 0 ou de 1 qu'on interprète différemment.

Variables et mémoire

- En simplifiant : si la machine veut mémoriser une valeur comme 12, elle la met dans une case mémoire vide (s'il en existe) à une certaine adresse. Pour récupérer la valeur stockée, il faut son adresse.

Variables et mémoire

- En simplifiant : si la machine veut mémoriser une valeur comme 12, elle la met dans une case mémoire vide (s'il en existe) à une certaine adresse. Pour récupérer la valeur stockée, il faut son adresse.
- Comme les adresses (écrites souvent en hexadécimal) sont difficile à retenir pour le programmeur, on associe un nom aux adresses. Ainsi, une *variable* est une adresse représentée par un nom.

Variables et mémoire

- En simplifiant : si la machine veut mémoriser une valeur comme 12, elle la met dans une case mémoire vide (s'il en existe) à une certaine adresse. Pour récupérer la valeur stockée, il faut son adresse.
- Comme les adresses (écrites souvent en hexadécimal) sont difficile à retenir pour le programmeur, on associe un nom aux adresses. Ainsi, une *variable* est une adresse représentée par un nom.
- Le *contenu* d'une variable x est soit

Variables et mémoire

- En simplifiant : si la machine veut mémoriser une valeur comme 12, elle la met dans une case mémoire vide (s'il en existe) à une certaine adresse. Pour récupérer la valeur stockée, il faut son adresse.
- Comme les adresses (écrites souvent en hexadécimal) sont difficile à retenir pour le programmeur, on associe un nom aux adresses. Ainsi, une *variable* est une adresse représentée par un nom.
- Le *contenu* d'une variable x est soit
  - la contenu de l'adresse n qu'elle représente

Variables et mémoire

- En simplifiant : si la machine veut mémoriser une valeur comme 12, elle la met dans une case mémoire vide (s'il en existe) à une certaine adresse. Pour récupérer la valeur stockée, il faut son adresse.
- Comme les adresses (écrites souvent en hexadécimal) sont difficile à retenir pour le programmeur, on associe un nom aux adresses. Ainsi, une *variable* est une adresse représentée par un nom.
- Le *contenu* d'une variable x est soit
  - la contenu de l'adresse n qu'elle représente
  - soit la *concaténation* des contenus des adresses $n, n+1, \ldots, n+k$ (avec k dépendant du type de la variable x).

Noms de variables en C

- Un nom de variable doit commencer par une lettre (majuscule ou minuscule) ou un « _ » (pas par un chiffre)

Noms de variables en C

- Un nom de variable doit commencer par une lettre (majuscule ou minuscule) ou un « _ » (pas par un chiffre)
- un nom de variable peut comporter des lettres, des chiffres et le caractère underscore (les espaces ne sont pas autorisés !)

Noms de variables en C

- Un nom de variable doit commencer par une lettre (majuscule ou minuscule) ou un « _ » (pas par un chiffre)
- un nom de variable peut comporter des lettres, des chiffres et le caractère underscore (les espaces ne sont pas autorisés !)
- les noms de variables ne peuvent pas être les noms suivants (qui sont des noms réservés) : `auto, break, case, char, const, continue, default` etc.

Les types prédéfinis

`void` utilisé pour spécifier le fait qu'il n'y a pas de type. Cela a une utilité notamment pour faire des *procédures* (fonctions ne renvoyant rien) ;

Les types prédéfinis

`void` utilisé pour spécifier le fait qu'il n'y a pas de type. Cela a une utilité notamment pour faire des *procédures* (fonctions ne renvoyant rien) ;

`char` les caractères : char. Les *chaînes de caractères* sont en fait des tableaux de caractères en C ;

Les types prédéfinis

`void` utilisé pour spécifier le fait qu'il n'y a pas de type. Cela a une utilité notamment pour faire des *procédures* (fonctions ne renvoyant rien) ;

`char` les caractères : char. Les *chaînes de caractères* sont en fait des tableaux de caractères en C ;

`int` les entiers ;

Les types prédéfinis

`void` utilisé pour spécifier le fait qu'il n'y a pas de type. Cela a une utilité notamment pour faire des *procédures* (fonctions ne renvoyant rien) ;

`char` les caractères : char. Les *chaînes de caractères* sont en fait des tableaux de caractères en C ;

`int` les entiers ;

`float` les « réels » (en fait une approximation des réels) (simple précision ?) ;

Les types prédéfinis

`void` utilisé pour spécifier le fait qu'il n'y a pas de type. Cela a une utilité notamment pour faire des *procédures* (fonctions ne renvoyant rien) ;

`char` les caractères : char. Les *chaînes de caractères* sont en fait des tableaux de caractères en C ;

`int` les entiers ;

`float` les « réels » (en fait une approximation des réels) (simple précision ?) ;

`bool` : bouléen avec les constantes `true` et `false`. Par défaut, pas de boulen en C. Les importer avec la directive `#include <stdbool.h>` ;

Les types prédéfinis

`void` utilisé pour spécifier le fait qu'il n'y a pas de type. Cela a une utilité notamment pour faire des *procédures* (fonctions ne renvoyant rien) ;

`char` les caractères : char. Les *chaînes de caractères* sont en fait des tableaux de caractères en C ;

`int` les entiers ;

`float` les « réels » (en fait une approximation des réels) (simple précision ?) ;

`bool` : bouléen avec les constantes `true` et `false`. Par défaut, pas de boulen en C. Les importer avec la directive `#include <stdbool.h>` ;

`double` les réels en double précision ;

Les types prédéfinis

`void` utilisé pour spécifier le fait qu'il n'y a pas de type. Cela a une utilité notamment pour faire des *procédures* (fonctions ne renvoyant rien) ;

`char` les caractères : char. Les *chaînes de caractères* sont en fait des tableaux de caractères en C ;

`int` les entiers ;

`float` les « réels » (en fait une approximation des réels) (simple précision ?) ;

`bool` : bouléen avec les constantes `true` et `false`. Par défaut, pas de boulen en C. Les importer avec la directive `#include <stdbool.h>` ;

`double` les réels en double précision ;

tableaux à une dimension : les indices sont spécifiés par des crochets ('[' et ']'). Pour les tableaux de dimension supérieure ou égale à 2, on utilise des tableaux de tableaux ;

Les types prédéfinis

`void` utilisé pour spécifier le fait qu'il n'y a pas de type. Cela a une utilité notamment pour faire des *procédures* (fonctions ne renvoyant rien) ;

`char` les caractères : char. Les *chaînes de caractères* sont en fait des tableaux de caractères en C ;

`int` les entiers ;

`float` les « réels » (en fait une approximation des réels) (simple précision ?) ;

`bool` : bouléen avec les constantes `true` et `false`. Par défaut, pas de boulen en C. Les importer avec la directive `#include <stdbool.h>` ;

`double` les réels en double précision ;

tableaux à une dimension : les indices sont spécifiés par des crochets ('[' et ']'). Pour les tableaux de dimension supérieure ou égale à 2, on utilise des tableaux de tableaux ;

types structurés : étudiés plus tard.

Types signés et non signés

- Les types entiers sont déclinés en deux versions `signed` (signé) et `unsigned` (non signé).

Types signés et non signés

- Les types entiers sont déclinés en deux versions `signed` (signé) et `unsigned` (non signé).
- Par défaut, un type est signé.

Types signés et non signés

- Les types entiers sont déclinés en deux versions `signed` (signé) et `unsigned` (non signé).
- Par défaut, un type est signé.
- Le type `int` utilise en général 4 octets (32 bits) sur beaucoup d'architectures.

Types signés et non signés

- Les types entiers sont déclinés en deux versions `signed` (signé) et `unsigned` (non signé).
- Par défaut, un type est signé.
- Le type `int` utilise en général 4 octets (32 bits) sur beaucoup d'architectures.
  - En version signée : minimum $-2^{32-1} = -2147483648$ et maximum $2^{32-1} - 1 = 2147483647$ (observer l'assymétrie).

Types signés et non signés

- Les types entiers sont déclinés en deux versions `signed` (signé) et `unsigned` (non signé).
- Par défaut, un type est signé.
- Le type `int` utilise en général 4 octets (32 bits) sur beaucoup d'architectures.
  - En version signée : minimum $-2^{32-1} = -2147483648$ et maximum $2^{32-1} - 1 = 2147483647$ (observer l'assymétrie).
  - En version non signée : minimum 0, maximum $2^{32} - 1 = 4294967295$.

Taille des types

- Les entiers peuvent être `short, int, long` par ordre croissant de *taille* (le nombre d'octets occupés par une variable de ces types).

Taille des types

- Les entiers peuvent être `short, int, long` par ordre croissant de *taille* (le nombre d'octets occupés par une variable de ces types).
- Le type `char` est le type entier le plus petit, sa taille est toujours un *byte* (la plus petite quantité de mémoire directement utilisable par un processeur). Le byte vaut le plus souvent un octet et c'est ce que nous adoptons pour la suite.

Taille des types

- Les entiers peuvent être `short, int, long` par ordre croissant de *taille* (le nombre d'octets occupés par une variable de ces types).
- Le type `char` est le type entier le plus petit, sa taille est toujours un *byte* (la plus petite quantité de mémoire directement utilisable par un processeur). Le byte vaut le plus souvent un octet et c'est ce que nous adoptons pour la suite.
- Malheureusement, la taille exacte est toujours un multiple de celle du type `char` mais elle dépend des architectures utilisées. Le type `int` devrait être le type des entiers natifs du processeur, donc 64 bits sur une architecture 64 bits, 32 bits sur une machine 32 bits etc.

Taille des types

- Pour rendre les codes plus portables, en général le type `int` est stocké sur 4 octets, `short` sur 2 et `long` sur 8. Nous adoptons cette convention à l'écrit.

Taille des types

- Pour rendre les codes plus portables, en général le type `int` est stocké sur 4 octets, `short` sur 2 et `long` sur 8. Nous adoptons cette convention à l'écrit.
- Quand on a besoin de précision (pour assurer la portabilité) on utilise les types `int8_t` (1 octet exactement), `int32_t` (4 octets) et `int64_t` (8) ainsi que leurs versions non signées `uint8_t`, `uint32_t` et `uint64_t`. On les importe avec la directive

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

Taille des types

La fonction `sizeof` permet de connaître en nombre de bytes la taille des types :

```
1  # include <stdio.h>
2
3  void main(){
4     printf("taille int %ld\n",sizeof(int));
5     printf("unsigned int %ld\n",sizeof(unsigned int));
6     printf("taille long %ld\n",sizeof(long));
7     printf("taille char %ld\n",sizeof(char));
8     printf("taille float %ld\n",sizeof(float));
9     printf("taille double %ld\n",sizeof(double));
10 }
```

Taille des types

Ce qui donne :

```
$ gcc size.c
$ ./a.out
taille  int 4
taille  unsigned int 4
taille  long 8
taille  char 1
taille  float 4
taille  double 8
```

Vision moins naïve de la mémoire

- Supposons qu'avec `int n = 1036` l'adresse de `n` soit 1451.

Vision moins naïve de la mémoire

- Supposons qu'avec `int n = 1036` l'adresse de `n` soit 1451.
- on est dans la situation suivante :

Adresse	Valeur	type	commentaire
⋮			
1451	00001100	int	début des quatre blocs de `n`
1452	00000100	int	toujours `n`
1453	00000000	int	toujours `n`
1454	00000000	int	fin des 4 blocs de `n`
1455	_		bloc libre
1456	_		bloc libre
1457	01000001	char	sur un seul bloc
⋮			

Vision moins naïve de la mémoire

- Supposons qu'avec `int n = 1036` l'adresse de `n` soit 1451.
- on est dans la situation suivante :

Adresse	Valeur	type	commentaire
⋮			
1451	00001100	int	début des quatre blocs de `n`
1452	00000100	int	toujours `n`
1453	00000000	int	toujours `n`
1454	00000000	int	fin des 4 blocs de `n`
1455	_		bloc libre
1456	_		bloc libre
1457	01000001	char	sur un seul bloc
⋮			

- On pourrait insérer une variable de type `short` aux adresses 1455 et 1456 ou encore deux `char`.

Déclaration

- Selon la syntaxe `type nom;` pour la déclaration

Déclaration

- Selon la syntaxe `type nom;` pour la déclaration
- Ex de déclaration : `int maVariable;`

Déclaration

- Selon la syntaxe `type nom;` pour la déclaration
- Ex de déclaration : `int maVariable;`
- La déclaration réserve un emplacement mémoire à la variable (la taille réservée dépend du type).

Déclaration

- Selon la syntaxe `type nom;` pour la déclaration
- Ex de déclaration : `int maVariable;`
- La déclaration réserve un emplacement mémoire à la variable (la taille réservée dépend du type).
- L'adresse de cet emplacement mémoire est récupérable en `C` par `&nomDeLaVariable`.

Déclaration

- Selon la syntaxe `type nom;` pour la déclaration
- Ex de déclaration : `int maVariable;`
- La déclaration réserve un emplacement mémoire à la variable (la taille réservée dépend du type).
- L'adresse de cet emplacement mémoire est récupérable en `C` par `&nomDeLaVariable`.
- Déclaration simultanées de variables de même type
  `int a,b,c,d,e;`

Déclaration

- Selon la syntaxe `type nom;` pour la déclaration
- Ex de déclaration : `int maVariable;`
- La déclaration réserve un emplacement mémoire à la variable (la taille réservée dépend du type).
- L'adresse de cet emplacement mémoire est récupérable en `C` par `&nomDeLaVariable`.
- Déclaration simultanées de variables de même type
  `int a,b,c,d,e;`
- Cependant, il n'y a pas de valeur par défaut. La variable `maVariable` prend la valeur du contenu de l'adresse au moment de la déclaration. Cela peut être n'importe quoi et, possiblement, autre chose qu'un entier.

Affectation

- `nomDeVariable = valeur` pour l'affectation. Ex : `a = 5;`

Affectation

- `nomDeVariable = valeur` pour l'affectation. Ex : `a = 5;`
- Pour éviter qu'une variable ait un contenu indésirable, on peut lui affecter une valeur au moment de sa déclaration : `int a =5`

Affectation

- `nomDeVariable = valeur` pour l'affectation. Ex : `a = 5;`
- Pour éviter qu'une variable ait un contenu indésirable, on peut lui affecter une valeur au moment de sa déclaration : `int a =5`
- Déclarations et affectations simultanées : `int a = 10, b = 5;`

Affectation

- `nomDeVariable = valeur` pour l'affectation. Ex : `a = 5;`
- Pour éviter qu'une variable ait un contenu indésirable, on peut lui affecter une valeur au moment de sa déclaration : `int a =5`
- Déclarations et affectations simultanées : `int a = 10, b = 5;`
- On peut changer plusieurs fois la valeur d'une variable

```
1       int v1;// déclaration
2       v1 = 5;// affectation
3       printf("v1 vaut %d; mais on peut changer\n", v1);
4       v1=8;
5       printf("v1 vaut maintenant %d\n", v1);
```

Variables globales/locales

Portée Selon l'endroit où on déclare une variable, celle-ci pourra être accessible (visible) partout dans le code ou bien uniquement dans une portion confinée de celui-ci.

Variables globales/locales

Portée Selon l'endroit où on déclare une variable, celle-ci pourra être accessible (visible) partout dans le code ou bien uniquement dans une portion confinée de celui-ci.

Variable globale Une variable accessible partout dans le code est dite *globale*. Ce sont des variables définies hors de toute fonction ou bloc d'instruction.

Variables globales/locales

Portée Selon l'endroit où on déclare une variable, celle-ci pourra être accessible (visible) partout dans le code ou bien uniquement dans une portion confinée de celui-ci.

Variable globale Une variable accessible partout dans le code est dite *globale*. Ce sont des variables définies hors de toute fonction ou bloc d'instruction.

variable locale Les variables qui ne sont pas globales sont *locales*.

```
1        int a = 5 // variable globale
2        int affine(int nombre)
3            {
4             int r = 0;// r est locale
5             r = 3 * nombre + a ; // OK car a globale
6             return r;
7             }
8        a = a + r ; //erreur car r est locale
9
```

Les constantes

- Définies selon la syntaxe `const type nom = valeur`, par exemple : `const float pi = 3.14`.

Les constantes

- Définies selon la syntaxe `const type nom = valeur`, par exemple : `const float pi = 3.14`.
- La valeur et le type de la constante ne sont pas destinés à changer.

Les constantes

- Définies selon la syntaxe `const type nom = valeur`, par exemple : `const float pi = 3.14`.
- La valeur et le type de la constante ne sont pas destinés à changer.
- Déclarée ainsi, une constante est typée et occupe de la place mémoire.

La directive `define`

Pour info uniquement

- La syntaxe est la suivante : `#define NOM valeur`, ceci devant être écrit au début du fichier. Par exemple : `#define INC 1` fixe la valeur d'un incrément.

La directive `define`

Pour info uniquement

- La syntaxe est la suivante : `#define NOM valeur`, ceci devant être écrit au début du fichier. Par exemple : `#define INC 1` fixe la valeur d'un incrément.
- Une constante définie ainsi n'est pas typée.

La directive `define`

Pour info uniquement

- La syntaxe est la suivante : `#define NOM valeur`, ceci devant être écrit au début du fichier. Par exemple : `#define INC 1` fixe la valeur d'un incrément.
- Une constante définie ainsi n'est pas typée.
- Le préprocesseur change dans le fichier source et ceux qui l'utilisent toutes les occurences du mot `INC` par la valeur 1.

La directive `define`

Pour info uniquement

- La syntaxe est la suivante : `#define NOM valeur`, ceci devant être écrit au début du fichier. Par exemple : `#define INC 1` fixe la valeur d'un incrément.
- Une constante définie ainsi n'est pas typée.
- Le préprocesseur change dans le fichier source et ceux qui l'utilisent toutes les occurences du mot `INC` par la valeur 1.
- En conséquence, une constante défini par `define` n'occupe pas de place en mémoire.

La directive `define`

Pour info uniquement

- La syntaxe est la suivante : `#define NOM valeur`, ceci devant être écrit au début du fichier. Par exemple : `#define INC 1` fixe la valeur d'un incrément.
- Une constante définie ainsi n'est pas typée.
- Le préprocesseur change dans le fichier source et ceux qui l'utilisent toutes les occurences du mot `INC` par la valeur 1.
- En conséquence, une constante défini par `define` n'occupe pas de place en mémoire.
- Cette méthode n'est pas recommandée en CPGE.

La directive `define`

Pour info uniquement.
Considérons

```
1 # define INC 1 // définition de la valeur de l'incrément
2 int myfunc(int i) {
3   return i + INC;// incrémente i de INC
4 }
```

La directive `define`

Pour info uniquement.
L'option `-E` de `gcc` permet de voir le contenu du fichier après traitement par le préprocesseur :

```
$ gcc -E define.c
# 1 "define.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "define.c"

int myfunc(int i) {
    return i + 1;
}
```

Les commentaires ont disparu ; le mot `INC` a été remplacé par 1.

Débordements

- Sur une machine donnée, un type entier a un domaine de valeur fixe, par exemple $[\![-2^{31}, 2^{31}-1]\!]$ pour les entiers `int` sur ma machine.

Débordements

- Sur une machine donnée, un type entier a un domaine de valeur fixe, par exemple $[\![-2^{31}, 2^{31}-1]\!]$ pour les entiers `int` sur ma machine.
- Que se passe-t-il si on essaye de stocker un entier plus grand que le maximum théorique autorisé pour son type ?

Débordements

- Sur une machine donnée, un type entier a un domaine de valeur fixe, par exemple $[\![-2^{31}, 2^{31}-1]\!]$ pour les entiers `int` sur ma machine.
- Que se passe-t-il si on essaye de stocker un entier plus grand que le maximum théorique autorisé pour son type ?
- La réponse dépend du caractère `signed` ou `unsigned` du type :

Débordements

- Sur une machine donnée, un type entier a un domaine de valeur fixe, par exemple $[\![-2^{31}, 2^{31}-1]\!]$ pour les entiers `int` sur ma machine.
- Que se passe-t-il si on essaye de stocker un entier plus grand que le maximum théorique autorisé pour son type ?
- La réponse dépend du caractère `signed` ou `unsigned` du type :

  signed Si on essaye d'enregistrer une valeur hors domaine dans une variable de type signé, la conversion n'est pas définie par le langage. Cela signifie que tout peut arriver.

Débordements

- Sur une machine donnée, un type entier a un domaine de valeur fixe, par exemple $[\![-2^{31}, 2^{31}-1]\!]$ pour les entiers `int` sur ma machine.
- Que se passe-t-il si on essaye de stocker un entier plus grand que le maximum théorique autorisé pour son type ?
- La réponse dépend du caractère `signed` ou `unsigned` du type :

  signed Si on essaye d'enregistrer une valeur hors domaine dans une variable de type signé, la conversion n'est pas définie par le langage. Cela signifie que tout peut arriver.

  unsigned Si on essaye d'enregistrer une valeur hors domaine dans une variable de type non signé (unsigned long, par exemple), la conversion se fait modulo la valeur maximale représentable par ce type + 1.

Débordement

Exemple avec un type `unsigned`

Le domaine de `unsigned char` est $[\![0, 255]\!]$. Les conversions se font modulo 256. Le compilateur nous indique qu'il y a un problème.

```
1  #include <stdio.h>// dans un fichier debordement_unsigned.c
2
3  int main(void)
4  {
5      unsigned char c = 300;
6      /* %hhu indique à printf qu'on veut afficher un unsigned
       char */
7      printf("La variable c vaut %hhu.\n", c);
8      c = -5;
9      printf("La variable c vaut %hhu.\n", c);
10     return 0;  }
```

Débordement

Exemple avec un type `unsigned`

- A la compilation avec `gcc debordement_unsigned.c`, on récupère un seul warning (-5 ne semble pas poser de problèmee) :

```
debordement_unsigned.c: In function 'main':
debordement_unsigned.c:5:23: warning:
large integer implicitly truncated to unsigned type
[-Woverflow]
     unsigned char c = 300;
```

Débordement

Exemple avec un type `unsigned`

- A la compilation avec `gcc debordement_unsigned.c`, on récupère un seul warning (-5 ne semble pas poser de problèmee) :

```
debordement_unsigned.c: In function 'main':
debordement_unsigned.c:5:23: warning:
large integer implicitly truncated to unsigned type
[-Woverflow]
     unsigned char c = 300;
```

- A l'exécution (avec `./debordement_unsigned`) :

```
La variable c vaut 44.
La variable c vaut 251.
```

Débordement

Exemple avec un type `signed`

Le domaine de `signed char` est $[\![-64, 63]\!]$. Le compilateur nous indique qu'il y a un problème.

```
1  #include <stdio.h>// dans un fichier debordement_signed.c
2
3  int main(void)
4  {
5      signed char c = 300;
6      /* %hhd sert à dire à printf() qu'on veut afficher un
       signed char */
7      printf("La variable c vaut %hhd.\n", c);
8      c = -300;
9      printf("La variable c vaut %hhd.\n", c);
10     return 0;}
```

Débordement

Exemple avec un type `signed`

- A la compilation avec `gcc debordement_signed.c`, on récupère deux warning pour 300 et -300.

```
debordement_signed.c:8:9: warning:
overflow in implicit constant conversion [-Woverflow]
      c = -300;
```

Débordement

Exemple avec un type `signed`

- A la compilation avec `gcc debordement_signed.c`, on récupère deux warning pour 300 et -300.

```
debordement_signed.c:8:9: warning:
overflow in implicit constant conversion [-Woverflow]
      c = -300;
```

- A l'exécution (avec `./debordement_signed`) :

```
La variable c vaut 44.
La variable c vaut -44.
```

La conversion, sur cette machine se fait aussi avec des modulos pour des entiers signés de type `char`.

Un exemple amusant

- Les conventions automatiques de conversion peuvent générer l'incompréhension du développeur.

Un exemple amusant

- Les conventions automatiques de conversion peuvent générer l'incompréhension du développeur.
- Voici un exemple amusant communiqué par François Fayard :

```c
#include <stdio.h>

int main() {
    int a = -1;
    unsigned int b = 0;
    if (b < a) {
        printf("b<a\n");
    }
    else printf("a<b\n");
    return 0;
}
```

Un exemple amusant (suite)

- Après compilation et exécution, on s'attend à obtenir `a<b`, mais :

```
1 $ ./a.out
2 b<a
3
```

Un exemple amusant (suite)

- Après compilation et exécution, on s'attend à obtenir `a<b`, mais :

```
1 $ ./a.out
2 b<a
3
```

- Lorsqu'on compare un type signé et un type non signé, avant la comparaison, le type signé est converti en type non signé. Du coup, -1 est converti en `unsigned int`. Cette conversion se fait modulo 2^{32} ce qui fait qu'il est converti en $2^{32} - 1 = 4294967295$. Comme ce nombre est plus grand que 0, le test s'évalue en true.

Un exemple amusant (suite)

- Après compilation et exécution, on s'attend à obtenir `a<b`, mais :

```
$ ./a.out
b<a

```

- Lorsqu'on compare un type signé et un type non signé, avant la comparaison, le type signé est converti en type non signé. Du coup, -1 est converti en `unsigned int`. Cette conversion se fait modulo 2^{32} ce qui fait qu'il est converti en $2^{32} - 1 = 4294967295$. Comme ce nombre est plus grand que 0, le test s'évalue en true.
- On retient cette règle de bon sens : il faut rester le plus éloigné possible des types non signés en C SAUF si on souhaite un comportement cyclique

La fonction `printf`

- `printf` (**print formatted**, ≪ imprimer formaté ≫) est unnom de fonction de C permettant d'afficher une ou plusieurs variables de façon formatée dans le flux de sortie.

La fonction `printf`

- `printf` (**print formatted**, « imprimer formaté ») est unnom de fonction de C permettant d'afficher une ou plusieurs variables de façon formatée dans le flux de sortie.
- il faut inclure l'en-tête standard `<stdio.h>` au début du code source du programme.

La fonction `printf`

- `printf` (**print formatted**, « imprimer formaté ») est unnom de fonction de C permettant d'afficher une ou plusieurs variables de façon formatée dans le flux de sortie.
- il faut inclure l'en-tête standard `<stdio.h>` au début du code source du programme.
- signature :

```c
int printf(const char* format, ...);
```

La fonction `printf`

- `printf` (**print formatted**, « imprimer formaté ») est unnom de fonction de C permettant d'afficher une ou plusieurs variables de façon formatée dans le flux de sortie.
- il faut inclure l'en-tête standard `<stdio.h>` au début du code source du programme.
- signature :

```c
int printf(const char* format, ...);
```

- Les pointillés signifient que c'est une fonction *variadique* (prend un nombre variable de paramètres).

La fonction `printf`

- `printf` (**print formatted**, « imprimer formaté ») est unnom de fonction de C permettant d'afficher une ou plusieurs variables de façon formatée dans le flux de sortie.
- il faut inclure l'en-tête standard `<stdio.h>` au début du code source du programme.
- signature :

```c
int printf(const char* format, ...);
```

- Les pointillés signifient que c'est une fonction *variadique* (prend un nombre variable de paramètres).
- Le terme `format` représente, de quoi sera faite la sortie (Entier, Double...). Principe : pour chaque « % », `printf` regarde la lettre qui suit ce % et écrit la variable correspondante (*i.e.* dans l'ordre de saisie) dans les paramètres.

Tableau de spécificateurs de formats

Symbole	Type	Impression comme
%d ou %i	`int`	entier relatif
%u	`int`	entier naturel non signé
%o	`int`	entier exprimé en octal
%x	`int`	entier exprimé en héxadécimal
%c	`char`	caractère
%s	`char *`	chaîne de caractères
%f	`double`	flottants et doubles en notation décimale
%e	`double`	flottant en notation scientifique

Affichage des entiers

- Les spécificateurs `%d,%i,%u,%o,%x` peuvent seulement représenter des valeurs du type `int` ou `unsigned int`.

Affichage des entiers

- Les spécificateurs `%d,%i,%u,%o,%x` peuvent seulement représenter des valeurs du type `int` ou `unsigned int`.
- Une valeur trop grande pour être stockée dans deux octets est coupée sans avertissement si nous utilisons `%d`.

Affichage des entiers

- Les spécificateurs `%d,%i,%u,%o,%x` peuvent seulement représenter des valeurs du type `int` ou `unsigned int`.
- Une valeur trop grande pour être stockée dans deux octets est coupée sans avertissement si nous utilisons `%d`.
- Pour pouvoir afficher correctement les valeurs de type `long`, il faut utiliser les spécificateurs `%ld,%li,%lu,%lo,%lx`.

Affichage des entiers

Exemple

- Avec :

```
1  long N = 1500000;
2  printf("%ld , %lx\n", N, N);
3  printf("%lo\n" , N);
4  printf("%li\n",N);
```

Affichage des entiers

Exemple

- Avec :

```
1  long N = 1500000;
2  printf("%ld , %lx\n", N, N);
3  printf("%lo\n" , N);
4  printf("%li\n",N);
```

- On obtient :

```
1500000, 16e360
5561540
1500000
```

Affichage des flottants

- Les spécificateurs `%f` et `%e` peuvent être utilisés pour représenter des arguments du type `float` ou `double`.

Affichage des flottants

- Les spécificateurs `%f` et `%e` peuvent être utilisés pour représenter des arguments du type `float` ou `double`.
- La mantisse des nombres représentés par `%e` contient exactement un chiffre (non nul) devant le point décimal : c'est la notation scientifique.

Affichage des flottants

- Les spécificateurs `%f` et `%e` peuvent être utilisés pour représenter des arguments du type `float` ou `double`.
- La mantisse des nombres représentés par `%e` contient exactement un chiffre (non nul) devant le point décimal : c'est la notation scientifique.
- Pour pouvoir traiter correctement les arguments du type long double, il faut utiliser les spécificateurs `%Lf` et `%Le`.

Affichage des flottants

- Le code :

```
1  float N = 12.1234;
2  double M = 12.123456789;
3  long double P = 15.5;
4  printf("N=%f\n", N);
5  printf("M=%f\n" , M);
6  printf("N=%e\n",N);
7  printf("M=%e",M);
8  printf(" Et enfin P : %Le, %Lf\n",P, P);
9
```

, affiche :

```
N=12.123400
M=12.123457
N=1.212340e+01
M=1.212346e+01 Et enfin P : 1.550000e+01, 15.500000
```

Affichage des entiers

Largeur minimale

- On peut indiquer la largeur minimale de la valeur à afficher.

Affichage des entiers

Largeur minimale

- On peut indiquer la largeur minimale de la valeur à afficher.
- Dans le champ réservé, les nombres sont justifiés à droite.

```
1  printf("%4d\n",123);// largeur min de 4 caractères
2  printf("%4d\n",1234);
3  printf("%4d\n", 12345);
4  printf("%4u\n", 0);
5  printf("%4x\n", 123);
```

Affichage des entiers

Largeur minimale

- On peut indiquer la largeur minimale de la valeur à afficher.
- Dans le champ réservé, les nombres sont justifiés à droite.

```
1  printf("%4d\n",123);// largeur min de 4 caractères
2  printf("%4d\n",1234);
3  printf("%4d\n", 12345);
4  printf("%4u\n", 0);
5  printf("%4x\n", 123);
```

- Et on obtient (la flèche a été ajoutée) :

```
   ↓
 123
1234
12345
   0
  7b
```

Affichage des flottants

Largeur, nombre de décimales

- On peut indiquer la largeur minimale de la valeur à afficher et la précision. La précision par défaut est fixée à six décimales.

```
1  printf("%f\n", 100.123);// 6 décimales par défaut
2  printf("%12f\n", 100.123); // largeur de 12
3  printf("%.2f\n", 100.123); // 2 décimales
4  printf("%5.0f\n", 100.123); // largeur 5, 0 décimale
5  printf("%10.3f\n", 100.123);// largeur 10, 3 décimale
6  printf("%.4f\n", 1.23456); // 4 décimales
7
```

- On obtient (avec parfois des arrondis) :

```
100.123000
  100.123000
100.12
  100
   100.123
1.2346
```

La fonction `scanf`

Lire, depuis l'entrée standard, des données formatées

- Lit les données depuis `stdin` et les stockes selon la valeur du paramètre `format` dans la ou les adresses pointées par les arguments additionnels.
  Ces arguments additionnels doivent pointer sur des objets déjà alloués dans le type spécifié par l'identifieur de format.

La fonction `scanf`

Lire, depuis l'entrée standard, des données formatées

- Lit les données depuis `stdin` et les stockes selon la valeur du paramètre `format` dans la ou les adresses pointées par les arguments additionnels.
  Ces arguments additionnels doivent pointer sur des objets déjà alloués dans le type spécifié par l'identifieur de format.
- Signature

```
1    int scanf ( const char * format, ... );
2
```

La fonction `scanf`

Lire, depuis l'entrée standard, des données formatées

- Lit les données depuis `stdin` et les stockes selon la valeur du paramètre `format` dans la ou les adresses pointées par les arguments additionnels.
  Ces arguments additionnels doivent pointer sur des objets déjà alloués dans le type spécifié par l'identifieur de format.
- Signature

```
1    int scanf ( const char * format, ... );
2
```

- La fonction retourne le nombre de variables affectées par la saisie, et permet donc de vérifier si la saisie s'est bien passée. En cas d'erreur de lecture, cette fonction renvoie `EOF` et indique la raison de l'erreur dans la variable `errno`.

Descripteurs de formats de saisie

Voici les principaux :

format	description
`%d`	Une donnée entière de type int
`%ld`	Une donnée entière de type long
`%f` ou `%lf`	Une donnée décimale flottante ou double
`%c`	Une donnée de type caractère
`%p`	donnée entière (en hexadécimal) de type adresse en m
`%s`	donnée type chaîne caractères.
`%[characters]`	chaîne de caractères, uniquement caractères spécifiés
`%[^characters]`	chaîne de caractères, aucun des spécifiés

Pour les chaînes de caractères, tout *séparateur* (comme un espace) interrompt la lecture. On peut préciser qu'on veut ou ne veut pas certains caractères.

Exemple

```c
#include <stdio.h>

int main(){
 int nombre = 0;
 int res;
 printf("Entrez une valeur : ");
 res = scanf("%d", &nombre);
 if (res == 1)
    printf("Votre valeur est %d.\n", nombre);
 else
    printf("Vous avez fait une erreur de saisie\n");
 return 0;
}
```

Exemple

```c
#include <stdio.h>

int main(){
 int nombre = 0;
 int res;
 printf("Entrez une valeur : ");
 res = scanf("%d", &nombre);
 if (res == 1)
    printf("Votre valeur est %d.\n", nombre);
 else
    printf("Vous avez fait une erreur de saisie\n");
 return 0;
}
```

- On obtient :

```
Entrez une valeur : 23
Votre valeur est 23.
```