Ksass'Peuk

CC BY NC SA

Zeste de savoir

Lire la norme C - exemple avec les valeurs non initialisées

4 avril 2021

Table des matières

TL ;DR : En C, initialisez les variables locales que vous créez. Si vous avez accès à C99 ou ultérieur, déclarez vos variables au plus proche possible de leur première utilisation. Ne lisez pas de mémoire que vous n'avez jamais écrite par ailleurs, ça vous évitera des surprises.

Ce billet à deux buts:

— lever quelques imprécisions sur la questions de la lecture de valeurs non-initialisées en C;
— montrer comment on procède pour extraire des informations d'un document comme la norme C.

On peut régulièrement lire qu'accéder à une valeur non-initialisée en C est un comportement indéterminé (*undefined behavior*), s'il n'est pas très grave d'avoir cette approximation en tête, en réalité ce n'est pas tout à fait vrai, et comme toujours avec C, c'est plus compliqué.

Sur cet exemple de question, à savoir «quel est le comportement d'un programme quand on lit une valeur initialisée», nous allons voir comment l'on peut extraire les informations de la norme, les raisons qui rendent ce travail fastidieux et complexe et les conséquences de tout cela sur la confiance que l'on a sur les connaissances extraites.

1. Quelques termes généraux de la norme C

Pour bien comprendre la suite, je vais faire quelques rappels (ou pas) de termes de la norme qui vont nous intéresser pour la suite.

1.0.0.1. Valeur non spécifiée

La norme nous dit:

> 3.19.3 - Unspecified value
> valid value of the relevant type where this International Standard imposes no requirements on which value is chosen in any instance.

C'est donc une valeur telle que le standard n'impose pas d'autre contrainte que le fait qu'elle doit être lisible. Un point important est que cela veut notamment dire qu'elle n'impose pas que celle-ci soit la même si le programme la lit deux fois d'affilée par exemple. Cela peut sembler obscur mais nous verrons que c'est important pour les questions d'initialisation.

1.0.0.2. Représentation piégée

NDLR : je n'ai pas trouvé de meilleure traduction

La norme nous dit:

> 3.19.4 - Trap representation
> an object representation that need not represent a value of the object type

Celle ci est encore un peu plus obscure au premier abord que la précédente définition, mais l'idée est en fait plutôt simple: une implémentation du langage C a le droit, selon la manière qu'elle a choisi d'implémenter certains types du langage (par exemple, les entiers, ou les flottants, …) d'avoir des valeurs considérées comme invalides.

Par exemple, le type `_Bool` ajouté en C99 a deux représentations valides: 0 et 1. Toute autre valeur dans une zone mémoire pour un booléen est une représentation piégée.

D'après la norme, lire une telle valeur est un comportement indéfini, dont nous parlerons un peu plus loin.

1.0.0.3. Valeur indéterminée

La norme nous dit :

> 3.19.2 - Indeterminate value
> either an unspecified value or a trap representation

Ici rien de complexe: une valeur indéterminée est une valeur de l'une des deux classes définies juste avant. À noter que comme une valeur non spécifiée est valide, elle ne peut pas être une représentation piégée et inversement.

1.0.0.4. Comportement indéfini

La norme nous dit:

> 3.4.3 - Undefined behavior
> behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements.

Pour faire simple: lorsqu'un comportement indéterminé est présent dans un programme, la sémantique (le sens) du C ne donne aucune information sur ce que va faire le programme si le code concerné est atteignable lors de l'exécution.

Un exemple simple et connu de cela, est l'accès hors bornes dans un tableau. Mais il est important de rappeler que la norme ne dit **aucunement** qu'une telle action mène nécessairement à un crash du programme. Et en pratique ce n'est effectivement pas nécessairement le cas, le programme peut tout à fait continuer son exécution en ayant fait silencieusement n'importe quoi (comme corrompre des données).

1.0.0.5. Comportement non spécifié

La norme nous dit:

> 3.4.4 Unspecified behavior
> **use of an unspecified value**, or other behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance.

Sans rentrer plus avant dans les détails, ce qui nous intéresse ici, c'est qu'utiliser une valeur non spécifiée entraîne un comportement non-spécifié, et que ce comportement doit être l'un de ceux proposés par le standard. Le programme ne peut pas faire complètement n'importe quoi comme cela peut être le cas pour un comportement indéterminé.

> *i* Relié à cette notion, on trouve aussi la notion de comportement défini par l'implémentation (*implementation-defined behavior*), qui est en gros un comportement non spécifié pour lequel la norme impose que l'implémentation documente son choix mais nous n'en aurons pas besoin dans la suite.

2. Usage de valeur non initialisée

2.1. Variable automatique non initialisée

Commençons par le cas le plus simple. Dans le programme suivant:

```c
int main(void){
  int i ;
  int j = i ; // undefined behavior
}
```

La lecture de `i` est un comportement indéterminé. Cependant, regardons plus précisément ce que nous dit la norme à ce sujet :

> 6.3.2.1 § 2
> If the lvalue designates an object of automatic storage duration that could have been declared with the register storage class (never had its address taken), and that object is uninitialized

> (not declared with an initializer and no assignment to it has been performed prior to use), the behavior is undefined.

Nous pouvons voir que c'est un peu plus compliqué que «la valeur n'a jamais été écrite». Ce comportement ne s'applique qu'aux objets C:

— avec un *automatic storage duration*,
— qui auraient pu être déclarées avec le mot clé `register`,
— qui n'a pas été initialisé,
— qui n'a pas eu d'opération en écriture.

Détaillons un peu cela.

> 6.2.4 § 5
> An object whose identifier is declared with no linkage and without the storage-class specifier `static` has automatic storage duration, as do some compound literals.

Pour éviter de nous enfoncer encore plus loin dans les méandres de la norme, coupons court: nous parlons ici des variables locales et des paramètres formels (les paramètres de fonctions). Et cela tombe bien c'est le sujet de cette section.

2.1.1. Variables « qui aurait pu être déclarées `register` »

Le second point est le plus intéressant, puisqu'il nous parle de `register`. Historiquement ce mot clé était utilisé pour demander au compilateur de s'assurer que la variable qualifiée ainsi soit placée dans un registre du processeur, à fin d'optimisation. Aujourd'hui les compilateurs ignorent poliment cette directive parce qu'ils sont beaucoup plus doués que les humains au petit jeu de «savoir qui doit aller dans un registre pour avoir les meilleures performances». Cependant, l'usage de ce mot clé a des implications importantes sur ce que l'on peut faire à une variable. En particulier.

> 6.7.1 § 5, footnote 121
> However, whether or not addressable storage is actually used, the address of any part of an object declared with storage-class specifier `register` cannot be computed, either explicitly (by use of the unary `&` operator as discussed in 6.5.3.2 or implicitly (by converting an array name to a pointer as discussed in 6.3.2.1).

On ne peut pas prendre l'adresse d'un élément (ou d'une sous-partie d'un élément) déclaré `register`. Cette règle est la seule qui peut nous interdire de placer le mot clé `register` sur une variable locale. Par conséquent, la règle que nous avons cité plus haut concerne les éléments dont l'adresse n'a pas été prise dans la fonction cible.

En particulier, dans le programme suivant :

```
1 int main(void){
2   int i ;
3   int j = i ;    // access
4   int * p = &i ; // the address of `i` is taken
5
6   int a[1];
```

```
7   int b = a[0];  // access + the address of `a[0]` is taken
8 }
```

Cette règle ne s'applique pas, il faut chercher ailleurs dans la norme pour traiter cet aspect.

2.1.2. Initialisation et affectation

La norme nous parle de deux opérations : initialisation et affectation. Ces deux opérations sont différentes en C.

```
1 int x = 0 ; // initialization
2 int y ;
3 y = 0 ;     // assignment
```

S'il n'est pas très utile de distinguer les deux lorsque nous parlons d'un entier, ce n'est pas la même chose pour le cas des structures ou pour les tableaux. Par exemple dans le code suivant:

```
1 struct S {
2   int x ;
3   int y ;
4 };
5
6 int main(void){
7   struct S s = { 1 } ;
8   int j = s.y ;
9 }
```

Notre «initialiseur» ne précise qu'une valeur tandis que la structure en possède 2. Dans ce cas, la norme nous dit :

> 6.7.9 § 21
> If there are fewer initializers in a brace-enclosed list than there are elements or membersof an aggregate, or fewer characters in a string literal used to initialize an array of known size than there are elements in the array, the remainder of the aggregate shall be initialized implicitly the same as objects that have static storage duration.

(Et `6.7.9 § 10` nous dit que pour le *static storage duration* on met des valeurs à 0 qui correspondent aux types cibles, c'est long donc je résume).

Donc dans notre code, l'accès à `s.y` se fait sur une valeur initialisée à 0. En revanche, si nous n'utilisons plus une initialisation mais une affectation partielle de la structure:

```c
int main(void){
  struct S s ;
  s.x = 1 ;
  int j = s.y ;
}
```

L'accès que nous faisons à `s.y` se fait cette fois sur une valeur non initialisée. Pour autant une affectation a bien été réalisée sur l'objet qui représente la structure. Ce code n'est donc pas non plus traité par la règle plus haut.

Un raisonnement similaire peut être fait pour le cas des tableaux.

2.2. Mémoire non initialisée

Le cas des variables dont l'adresse est prise ou qui a été partiellement affecté avant usage est traitée par la règle suivante de la norme :

> 6.7.9 § 10
> If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate.

Ici, nous pouvons faire un parallèle avec le cas d'une mémoire que l'on aurait récupéré via une allocation dynamique:

> 7.22.3.4 § 2
> The malloc function allocates space for an object whose size is specified by size and whose value is indeterminate.

Que l'on reçoive un pointeur sur une zone de mémoire ou que l'on crée un pointeur sur une zone de mémoire automatique existante, le résultat est le même: la mémoire contient des valeurs indéterminées.

Nous avons donc deux possibilités:

— la valeur lue est non spécifiée, et le comportement est alors non spécifié,
— la valeur lue est une représentation piégée, et le comportement est alors indéfini.

Il va donc falloir s'intéresser de plus près aux représentations piégées.

3. Représentation piégée

Rappel, la norme nous dit:

> an object representation that need not represent a value of the object type

Autant dire qu'elle est un peu avare en détails. Et pour cause, c'est quelque chose d'assez spécifique aux implémentations. Mais regardons cela de plus près.

En fouillant un peu, nous pouvons trouver quelques informations supplémentaires. Tout d'abord:

> 6.2.6.1 § 5
> Certain object representations need not represent a value of the object type. If the stored value of an object has such a representation and is read by an lvalue expression that does not have character type, the behavior is undefined. [...] Such a representation is called a trap representation.

Ce qu'on apprend ici, c'est qu'a priori, jusqu'à ce que la norme nous dise le contraire les objets C peuvent avoir des représentations piégées. Cela tendrait à nous dire que les exemples que nous avons montré plus tôt peuvent en générer.

Nous allons voir que le fait qu'un type puisse avoir ou non une représentation piégée est lié aux bits de remplissage (*padding*). À savoir des bits qui ne sont là que pour «compléter» l'espace dans les multiplets (que je simplifierai en octet dans la suite parce que des architectures avec autre chose que des octets, on n'en a pas tout le tour du ventre) utilisés par l'objet s'il a besoin de moins de bits que ce qu'ils peuvent contenir. Par exemple, le type `_Bool` n'a fondamentalement besoin que de 1 bit pour faire son travail. Cependant, le standard impose qu'il fasse au moins `CHAR_BIT`, dont le minimum est 8 dans la norme. Nous avons donc au minimum 7 bits de «remplissage».

3.1. Bits de remplissage

Que nous dit la norme à propos des bits de remplissage?

> 6.2.6.2 § 5
> The values of any padding bits are unspecified. (54) [...]
> (54) Some combinations of padding bits might generate trap representations, for example, if one padding bit is a parity bit.

Leur valeur est dite non-spécifiée. Cela peut sembler ajouter un peu de confusion à tout cela puisque la norme nous disait plus tôt que ces valeurs sont censées être valides, et ne pas être des représentations piégées. En fait la raison est relativement simple: la valeur de chacun de ces bits est effectivement non spécifiée et il n'est pas interdit de les lire. En revanche quand ils forment le remplissage d'un objet d'un type donné, en lisant l'objet en question, c'est la combinaison de ces bits qui peut donner lieu à une représentation piégée.

Nous pouvons donc déjà lister deux catégories de types susceptibles de produire des représentations piégées:

— le type `_Bool`
— les types `enum` (dont les valeurs sont une sous-plage d'un type entier)

3.2. Types entiers

Les entiers non-signés sont décrits dans cette section :

> 6.2.6.2 § 1
> For unsigned integer types **other than `unsigned char`**, the bits of the object representation shall be divided into two groups : value bits **and padding bits** (there need not be any of the latter). […]

On y apprend que le type `unsigned char` ne peut pas avoir de bits de remplissage. Toutes les valeurs doivent donc être valides: ce type n'a pas de représentation piégée.

En revanche rien ne garantit dans ce paragraphe qu'un autre type non signé n'ait pas de représentation piégée.

Les entiers signés sont décrits dans cette section.

> 6.2.6.2 § 2
> For signed integer types, the bits of the object representation shall be divided into three groups : value bits, **padding bits, and the sign bit**. There need not be any padding bits ; **`signed char` shall not have any padding bits**. […]
> If the sign bit is one, the value shall be modified in one of the following ways :
> — the corresponding value with sign bit 0 is negated (sign and magnitude) ;
> — the sign bit has the value $-(2^M)$ (two's complement) ;
> — the sign bit has the value $-(2^M-1)$ (ones' complement).
> Which of these applies is implementation-defined, as is whether the value with the value with sign bit 1 and all value bits zero (for the first two), or with sign bit and all value bits 1 (for ones' complement), **is a trap representation** or a normal value.

Le cas est un peu plus complexe. À nouveau, il est possible d'avoir des bits de remplissages (et donc des représentations piégées), mais ce n'est pas tout. En effet, la norme indique que le bit de signe peut intervenir aussi dans la possibilité ou non d'avoir une représentation piégée. En conséquence, même si, comme la norme l'indique, le type `signed char` n'a pas de bits de remplissage, rien n'empêche qu'il puisse avoir une représentation piégée.

Depuis C99, la norme propose d'avoir des entiers à taille exacte:

> 7.20.1.1
> §1: The typedef name `intN_t` designates a signed integer type with width N, no padding bits, and a two's complement representation. Thus, `int8_t` denotes such a signed integer type with a width of exactly 8 bits.

Ici, nous voyons que la norme interdit les bits de remplissages pour ces types mais ne dit rien à propos du cas du bit de signe pour exclure ou non la représentation piégée sur ce point, dommage.

> §2: The typedef name `uintN_t` designates an unsigned integer type with width N and nopadding bits. Thus, `uint24_t` denotes such an unsigned integer type with a width of exactly 24 bits.

En revanche pour les non-signés, exclure les bits de remplissage exclus la présence de représentation piégée. Tout les types de taille fixe non signés excluent dont cette possibilité.

Les autres types entiers spécifiques (`int_leastN_t`, `int_fastN_t`, etc) ne donnent pas de contraintes particulières sur les bits de remplissage, on peut donc considérer qu'ils ont les mêmes propriétés que les entiers habituels.

3.2.1. Résumé pour les entiers

Les types suivants ne peuvent pas avoir de représentation piégée:

— `unsigned char`
— `uintN_t`

Le comportement des types non signés restant est conditionné par les bits de remplissage.

Le comportement des types signés est principalement conditionné par le comportement du bit de signe, mais aussi par les bits de remplissage (pour les types qui ne sont pas de taille fixe).

3.3. Types `float`

La norme est très indirecte lorsqu'elle parle du comportement des nombres `float`. La seule occurrence que j'ai pu trouver est la suivante:

> F.2.1
> This specification does not define the behavior of signaling NaNs. It generally uses the term NaN to denote quiet NaNs. The `NAN` and `INFINITY` macros and the nan functions in `<math.h>` provide designations for IEC 60559 NaNs and infinities.

Comme le comportement lié au fait de lire un *signaling NaN* n'est pas un comportement défini, cela rapproche ce comportement de la notion de représentation piégée (dont la lecture est aussi un comportement indéfini).

3.4. Pointeurs

La norme est à nouveau indirecte ici :

> 6.3.2.3 § 5
> An integer may be converted to any pointer type. Except as previously specified, the result is implementation-defined, might not be correctly aligned, might not point to an entity of the referenced type, and might be a trap representation.

On apprend ici que la conversion d'un entier en pointeur peut amener à une représentation piégée, de telles représentations peuvent donc bien exister pour les pointeurs. Rien n'exclut donc que la valeur indéterminée d'un pointeur soit une représentation piégée.

Un pointeur peut avoir une représentation piégée.

3.5. Structures

A propos des structures et des unions, nous apprenons:

> 6.2.6.1 § 6
> […] The value of a structure or union object is never a trap representation, even though the value of a member of the structure or union object may be a trap representation.

J'ai enlevé la première partie qui ne nous intéresse pas pour le moment nous y reviendrons pour les unions. En revanche, nous apprenons ici que la valeur d'une structure ou d'une union n'est jamais une représentation piégée. Donc ici:

```c
struct S {
  int x ;
  int y ;
};

int main(void){
  struct S s ;
  struct S *ptr = &s ; // we take the address of s
  struct S s2 = s ;    // s is not a trap representation
}
```

Copier une structure (ou une union) de valeur indéterminée est toujours défini.

Pour ce qui est d'accéder aux champs, on réapplique le raisonnement aux types des champs utilisés. S'ils ont un type structure (ou union), le cas présent s'applique, sinon c'est le cas de l'un des types fondamentaux définis plus tôt qui s'appliquera. Comme ici:

```c
struct S {
  int x ;
  int y ;
};
struct T {
  unsigned char v1 ;
  struct S v2 ;
  int v3 ;
};

int main(void){
  struct T t ;
  struct T *ptr = &t ;      // we take the address of t
  unsigned char v1 = t.v1 ; // no trap representation
  struct S v2 = t.v2 ;      // no trap representation
  int v3 = s.v3 ;           // potential trap represensaiton
}
```

3.6. Unions

Le dernier cas est celui des unions. Dans les grandes lignes, les unions ont le comportement spécifié précédemment pour les structures. Mais il ne faut pas oublier que lorsque l'on déclare un type union comme:

```c
union U {
  int x ;
  unsigned char a[4];
};
```

Les champs `x` et `a` ne sont pas séparés, ce sont deux vues possibles du contenu de la mémoire à cet emplacement. En conséquence quelques cas particuliers s'appliquent comme le dit ce passage de la norme:

> 6.5.2.3 § 3
> (95) If the member used to read the contents of a union object is not the same as the member last used to store a value in the object, the appropriate part of the object representation of the value is reinterpreted as an object representation in the new type as described in 6.2.6. This might be a trap representation.

Nous sommes donc dans un cas où une initialisation faite sur l'un des membres de l'union, tout en n'étant pas une représentation piégée pour ce type, peut être une représentation piégée si l'on interprète cette valeur à travers le type d'un autre champ de l'union. Par exemple:

```c
union X {
  unsigned char e1 ;
  signed char   e2 ;
};

int main(void){
  union X x ;
  x.e1 = <SOME BIT PATTERN THAT MIGHT BE A TRAP FOR signed char> ;
  unsigned char e = x.e1 ; // never a trap representation
  signed char f = x.e2 ;   // might be a trap representation
}
```

À cela s'ajoute les questions d'alignements de champs. Nous avons parlé des bits de remplissage pour les types fondamentaux, mais il faut aussi considérer le cas des octets utilisés pour aligner les champs de structure. Sans rentrer dans les détails. Si l'on prend une structure comme :

```c
struct S {
  char x ;
  int  i ;
};
```

Si le type `int` prend 4 octets, la structure sera de taille 8 octets. En effet, on s'arrangera pour mettre `x` dans une zone de 4 octets, même s'il n'en utilise qu'un pour que `i` soit aligné en mémoire sur une adresse multiple de 4, pour des questions de performances ou de compatibilité matérielle.

Le passage qui nous intéresse maintenant est le suivant:

> 6.2.6.1 § 6
> When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values. […]

Donc si nous mettons une `struct S` dans une union avec une autre structure qui n'a pas de tels octets de remplissage, par exemple:

```
struct T {
  int f ;
  int g ;
};
union U {
  struct S s ;
  struct T t ;
};
```

Initialiser la partie `x` de `S` entraîne que les octets de remplissage juste après lui ont une valeur non-spécifiée, d'où le programme suivant:

```
int main(void){
  union U u = { .s = { 'a', 0 } } ;
  int v = u.t.f ; // migth be a trap representation
```

Et cela peut se produire aussi lors de l'écriture d'un champ simple d'une union, mais la formulation est plus insidieuse:

> 6.2.6.1 § 7
> When a value is stored in a member of an object of union type, the bytes of the object representation that do not correspond to that member but do correspond to other members take unspecified values.

Le premier exemple auquel nous pensons est effectivement intuitif par rapport à ce que nous avons vu jusqu'ici :

```
union U {
  unsigned char x ;
  int  y ;
};
int main(void){
  union U u = { .x = 0 };
  int v = u.y ; // might be a trap representation
}
```

Mais la formulation implique aussi que le code suivant présente un problème.

```c
union U {
  unsigned char x ;
  int  y ;
};
int main(void){
  union U u = { .y = 0 }; // y is not a trap representation
  u.x = 1 ;               // but writing x bytes
  int i = u.y ;           // makes y a potential trap
      representation
}
```

Parce que la norme nous dit qu'écrire `x` entraîne que les octets non utilisés par cette écriture prennent une valeur non spécifiée quand bien même ces octets étaient définis juste avant!

À cause de cela, on pourrait trop rapidement résumer par le fait que l'on ne peut simplement pas lire un champ si ce n'est pas le dernier à avoir été écrit, sauf que ce n'est bien entendu pas le cas. Si les types sont compatibles (sans rentrer dans tous les détails) nous avons certaines garanties. Notamment:

> 6.5.2.3 § 6
> One special guarantee is made in order to simplify the use of unions : if a union contains several structures that share a common initial sequence (see below), and if the union object currently contains one of these structures, it is permitted to inspect the common initial part of any of them anywhere that a declaration of the completed type of the union is visible. Two structures share a common initial sequence if corresponding members have compatible types (and, for bit-fields, the same widths) for a sequence of one or more initial members.

Nous ne pouvons donc pas faire ce raccourci.

4. Conséquences du concept de valeur indéterminée

4.1. Les représentations piégées, en vrai

Nous l'avons dit les valeurs indéterminées sont de deux catégories:

— les valeurs non spécifiées,
— les représentations piégées.

Seules les secondes entraînent un comportement indéterminé en cas d'usage.

Mais des représentations piégées, en réalité, en trouve-t-on souvent dans les implémentations de C? Il semble que ce ne soit pas vraiment le cas.

La proposition de changement N2091 pour la norme C nous apprend notamment que:

— pour les entiers ce n'est globalement pas le cas (exceptions, les booléen, et on pourra noter les énumérations, qui ne sont pas mentionnées dans ce rapport);

— pour les flottant, les signaling NaN existent mais pas partout, et ne sont généralement pas actifs, et quand ils sont actifs, leurs utilisateurs attendent un comportement particulier de l'implémentation et pas un comportement indéterminé;
— pour les pointeurs, certaines architectures plus vraiment fabriquées ont cela.

En conséquence, sur la majorité des implémentations, ces valeurs ne seront que des valeurs non-spécifiées. Cependant avant de dire que l'on pourrait simplement considérer que ces cas ne sont plus assez nombreux aujourd'hui pour les considérer, demandons nous quand même: «quel est le comportement d'un programme qui utilise des valeurs non spécifiées?».

4.2. Les valeurs non spécifiées d'après la norme

4.2.1. Comportement d'un programme simple

Reprenons un exemple.

```c
int main(void){
  unsigned t[1];
  unsigned a = t[0];
  unsigned b = t[0];

  if(a == b){
    return 0 ;
  } else {
    return 1 ;
  }
}
```

Nous savons que le programme n'a pas de comportement indéterminé, mais que pouvons nous dire à propos du comportement de ce programme d'après la norme C.

Eh bien, tristement, rien du tout mis à part qu'il va s'exécuter jusqu'à atteindre un `return`. En effet, `a` et `b` vont recevoir chacun une valeur non spécifiée. Dès lors, rien ne garantit que la condition sera évaluée à vraie. Car la norme nous a dit:

> valid value of the relevant type where [the norm] imposes no requirements on which value is chosen **in any instance**.

Donc pire encore, on ne sait pas non plus ce que retourne le programme suivant :

```c
int main(void){
  unsigned t[1];

  if(t[0] == t[0]) return 0 ;
  else return 1 ;
}
```

Même si notre programme n'a pas de comportement indéterminé, il ne semble pas raisonnable de reposer sur de telles valeurs pour écrire un programme.

Le rapport de défaut DR451 explique cela plus longuement.

4.2.2. Cas particulier lié au comportement des implémentations

Le même rapport nous apprend également:

> The answer to question 3 is that © library functions will exhibit undefined behavior when used on indeterminate values.

Donc, impossible d'utiliser la bibliothèque standard avec des valeurs indéterminées sans invoquer un comportement indéterminé? La réponse fournie est insuffisamment précise et mériterait beaucoup plus de détails (et rien ne le précise dans la norme actuelle).

En effet, pourquoi ce programme:

```c
int main(void){
  struct S s ;
  struct S a[1];
  memcpy(a, &s, sizeof(S));
}
```

devrait il entraîner un comportement indéterminé? C'est d'autant plus étrange qu'il est très facile de trouver «à l'état sauvage» des programmes travaillant pendant un temps avec des structures partiellement initialisées, qui remplissent progressivement les données calculées et qui font des manipulations comme celle ci-dessus entre temps.

4.3. Faisons un point

Pour résumer rapidement cette section, nous apprenons que:

— les représentations piégées sont peu communes;
— elles sont peu comprises[1];
— les valeurs non spécifiées donnent au mieux des programmes sans sémantique claire;
— au pire des programmes avec des comportement indéterminé.

Au final, qu'avons-nous appris au sujet de l'initialisation? Grossièrement, que :

— si l'on lit une variable qui aurait pu être `register` sans l'initialiser, c'est un comportement indéterminé;
— sinon on lit une valeur indéterminée, qui peut:
  — être une valeur non spécifiée, dans ce cas, au mieux son usage produira un comportement imprévisible d'après la norme, au pire c'est un comportement indéterminé;

1. Je n'ai cité qu'une proposition, mais en cherchant, on trouve beaucoup de questions au comité sur ce sujet.

  - être une représentation piégée, et c'est un comportement indéterminé;
- il est possible de manipuler des blocs de mémoire qui contiennent des valeurs non-initialisées de manière définie.

D'un point de vue développeur, les bonnes pratiques sont donc: ne jamais lire une valeur non-initialisée explicitement, seule exception: on n'a rien à craindre en copiant des blocs de mémoire (soit via des opérations de copie comme `memcpy`, soit via des copies de structures).

Venons en maintenant à quelques constatations à propos de la norme.

Tout d'abord, nous avons pu voir que pour répondre à une question très simple à propos du comportement d'un programme, il nous a fallu beaucoup de traval. Si cela nous a permis d'aboutir à quelques règles simples pour le développeur, ces règles ne s'appliquent pas à quelqu'un qui devrait développer un compilateur ou un analyseur.

Cela crée un décalage important entre ce qu'attend un développeur et ce qu'un fournisseur d'outil sera en mesure de lui fournir s'il décide de se focaliser sur la norme. L'outil pourrait donc s'avérer correct d'après la norme et imparfait d'après l'utilisateur. Cela peut s'apparenter à un outil qui passe sa vérification … mais pas sa validation!

Par ailleurs quelle confiance aurions nous si nous devions demain réaliser une partie de l'implémentation d'un compilateur qui dépend de ces informations, ou un analyseur? En effet nous avons dû lire:

- de *trop nombreux* paragraphes,
- ces paragraphes étaient *très courts et très imprécis*,
- ces paragraphes sont ventilés dans tout le manuel.

Dès lors, a-t-on:

- bien trouvé tous les paragraphes importants?
- bien compris tout le sens des paragraphes importants?
- bien pris en compte les conséquences de notions connexes?

Pour ma part, après plusieurs jours passés à collecter et analyser ces informations, j'ai une confiance toute relative en tout cela, alors que nous avons traité une question extrêmement simple. Comment s'assurer que les cas complexes sont bien compris? La norme avec ses annexes est quand même un pavé de plus de 600 pages de définitions.

Si ces documents normatifs sont des ressources précieuses, elles sont très loin d'être exemptes de défauts. Et ces défauts sont des bugs potentiels un jour ou l'autre. Murphy nous dit que ça arrivera toujours au pire moment. Peut-être devrait-on trouver de meilleurs moyens de fournir ces informations?