Types entiers de taille connues

Definition (Entiers de taille connue)

La taille des types standard pouvant varier d'une plateforme à une autre, les derniers standards du langage C définissent l'existence possible de types entiers dont les tailles sont connues, ce qui aide à la portabilité de nombreux codes.
Ces types sont disponibles par inclusion du fichier d'en-tête `<stdint.h>`.

Les entiers sont classifiés en :

- les entiers dont la **taille** est **exacte** (*exact width*),
- les entiers dont la **taille** est **au moins** égale à la spécification (*least specified width*),
- les entiers dont la **taille** est **au moins** égale à la spécification et pour lesquels une certaine garantie de **rapidité** de fonctionnement existe (*fast integers*),
- les entiers garantissant le stockage d'une adresse (*wide enough*),
- les **plus grands entiers** disponibles (*greatest width*).

Entiers de taille exacte

S'ils sont disponibles, on trouve :

- les types signés `int8_t`, `int16_t`, `int 32_t` et `int64_t`,
- les types non-signés `uint8_t`, `uint16_t`, `uint 32_t` et `uint64_t`.

qui sont respectivement de taille 8, 16, 32 et 64 bits exactement.

Entiers de taille minimale

On trouve :

- les types signés `int_least8_t`, `int_least16_t`, `int_least32_t` et `int_least64_t`,
- les types non-signés `uint_least8_t`, `uint_least16_t`, `uint_least32_t` et `uint_least64_t`,

qui sont respectivement et au moins de taille 8, 16, 32 et 64 bits. L'existence de ces types est requise.

Entiers rapides

On trouve :

- les types signés `int_fast8_t`, `int_fast16_t`, `int_fast 32_t` et `int_fast64_t`,
- les types non-signés `uint_fast8_t`, `uint_fast16_t`, `uint_fast 32_t` et `uint_fast64_t`,

qui sont respectivement et au moins de taille 8, 16, 32 et 64 bits et dont la plateforme garantit que les calculs sont effectués rapidement.
L'existence de ces types est requise.

Grands entiers

Tout d'abord, les entiers permettant de stocker toute adresse de type `void *` : `intptr_t` et `uintptr_t` dont l'existence est optionnelle.
Et ensuite, les plus grands entiers disponibles : `intmax_t` et `uintmax_t`, dont l'existence est bien entendu requise.

Bornes pour les types de taille spécifiée

Il existe bien entendu des macro-définitions de valeurs pour les bornes des types précécents. On les citera pas toutes mais on trouvera par exemple :

- `INT`N`_MIN` qui représente la valeur -2^{N-1}, pour $N = 8, 16, 32$ ou 64,
- `INT`N`_MAX` qui représente la valeur $2^{N-1} - 1$,
- `UINT`N`_MAX` qui représente la valeur $2^N - 1$,

Énumérations

Definition (Énumération)

On appelle **énumération** (*enumeration*) la définition extensive (explicite) d'un ensemble de valeur. Le langage C permet de définir des énumérations qui définissent simplement des constantes entières (et uniquement) à chacune desquelles un identificateur est associé. La syntaxe de définition d'un type énuméré est :

```
enum [identificateur] { identificateur [= constante], ... };
```

Note

Le nom du type est `enum` *identificateur*.

Exemple

```
#include <stdio.h>

enum JOURS { LUNDI=0, MARDI=1,
    MERCREDI=2, JEUDI=3, VENDREDI=4,
    SAMEDI=5, DIMANCHE=6 };

int main(void) {
  printf("Lundi␣c'est␣%d\n",LUNDI);
  printf("Vendredi␣c'est␣%d\n",VENDREDI);
}
```

produit :

```
$ ./enum1
Lundi c'est 0
Vendredi c'est 4
$
```


⚠ Attention

Les énumérations ne définissent pas d'espace de nom! Il ne peut donc y a voir dans une même visibilité de définition de constantes d'énumérations de même nom.

Exemple

```
#include <stdio.h>
enum JOURS { LUNDI=0, MARDI=1 };
enum DAYS { LUNDI = 0 };
```

produit une erreur de compilation :

```
$ gcc -o enum2 enum2.c
enum2.c:3:13: error: redefinition of enumerator 'LUNDI'
enum DAYS { LUNDI = 0 };
            ^
enum2.c:2:14: note: previous definition is here
enum JOURS { LUNDI=0, MARDI=1 };
             ^
1 error generated.
$
```

Si à un identificateur ne correspond pas de valeur explicitement définie la valeur est celle de la constante précédente augmentée de 1. Pour la première constante, la valeur par défaut est 0.

Exemple

```
#include <stdio.h>
enum JOURS { LUNDI, MARDI , MERCREDI };

int main(void) {
  printf("Mercredi␣c'est␣%d\n",MERCREDI);
}
```

produit :

```
$ ./enum3
Mercredi c'est 2
$
```


Il n'y a pas de nécessité à ce que les valeurs des constantes soient croissantes, ni même à ce que les valeurs soient toutes différentes.

Exemple

```
#include <stdio.h>
enum JOURS { LUNDI=0,MARDI=12,MERCREDI=3,JEUDI=12 };

int main(void) {
  printf("Jeudi␣c'est␣%d\n",JEUDI);
}
```

produit :

```
$ ./enum4
Jeudi c'est 12
$
```

Les énumérations constituent des types, on peut donc déclarer des variables de ce type :

Exemple

```
#include <stdio.h>
enum JOURS { LUNDI=0,MARDI=12,MERCREDI=3,JEUDI=12 };

int main(void) {
  enum JOURS unJour = JEUDI;
  if (unJour==JEUDI) printf("C'est␣jeudi\n");
  else               printf("Pas␣jeudi\n");
}
```

produit :

```
$ ./enum5
C'est jeudi
$
```


Si seules les constantes sont intéressantes alors il est possible d'omettre la **signature** (*tag*) du type énuméré.

Exemple

```
enum {
        LUNDI, MARDI, MERCREDI
};
```

Évidemment dans ce cas, il n'est plus possible de déclarer une variable du type puisqu'il ne porte pas de nom.

On peut définir une énumération anonyme et des variables de ce type en même temps :

Exemple

```
enum {
  OPEN, READ, WRITE, CLOSE
} mode;
```


Types dérivés

Les types dérivés sont :

- tableaux
- structure
- union
- fonction
- pointeur

Un type dérivé peut-être construit à partir d'autres types dérivés.

Tableaux

Les tableaux du C ont un statut particulier, on dit même parfois qu'«ils n'existent pas vraiment» car ils sont facilement «assimilables» à des pointeurs. C'est un peu exagéré mais les différences sont un peu subtiles.

Definition (Type tableau)

représente une **collection non vide d'objets contigus en mémoire**, d'un type dit type des éléments et dont dérive le type tableau. On parle, par exemple, de «tableaux d'`int`» (*array of* `int`*s*).

Il s'agit d'un type dit d'**agrégation**.

Définition d'un tableau

Definition (Définition d'un tableau)

La définition d'un tableau consiste à lui associer une taille, ainsi:

type *identificateur* `[` *taille* `];`

où la *taille* est une constante entière.
Il est possible de ne pas spécifier la taille mais en ce cas, le tableau doit être initialisé (voir p.389).
Il est aussi possible de ne pas utiliser une constante mais en ce cas, le tableau est appelé tableau de taille variable (voir p.386).

⚠ Attention

La taille doit être strictement positive.

Accès aux tableaux

Definition (accès à un tableau)

Une expression comme :

$e_1[e_2]$

dans laquelle l'une des deux sous-expressions est du type tableau ou pointeur et l'autre d'un type entier, permet d'obtenir l'élément d'indice correspondant à l'expression entière dans le tableau.

⚠ Attention

Les indices des tableaux commencent à 0, ainsi 0 désigne le premier élément, 1 le second, etc., le dernier est donc la taille du tableau - 1.

Exemple

```c
#include <stdio.h>

int main(void) {
  int t[5];
  for (int i=0; i<5; i++) t[i] = 2*i;
  for (int i=0; i<5; i++) printf("%d\n",t[i]);
}
```

```
$ ./arrayaccess
0
2
4
6
8
$
```

Attention

Les tableaux sont des objets non modifiables... On peut modifier les éléments du tableau mais pas le tableau lui-même. Ceci signifie que l'affectation d'un tableau est interdite.

Mauvais exemple

```
int main(void) {
  int t[5], t2[5];
  t = t2;
}
```

```
make arrayassign
cc --std=c11 -pedantic -Wall    arrayassign.c   -o
    arrayassign
arrayassign.c:3:5: error: array type 'int [5]' is not
     assignable
  t = t2;
  ~ ^
1 error generated.
make: *** [arrayassign] Error 1
```


Note

$e_1[e_2]$ est strictement équivalent à $(*((e_1)+(e_2)))$.

Cette remarque n'est pas anodine car elle autorise à écrire :

Exemple tordu

```
#include <stdio.h>

int main(void) {
  int t[50];
  2[t] = 666;
  printf("%d\n",t[2]);
}
```

Ce sport est assez couramment pratiqué parmi les programmeurs C.

Attention

Toute expression du type «tableau de *T*» est réduite en «pointeur sur *T*» (voir p. 255); à l'exception d'un usage aves les opérateurs **`sizeof`** et `_Alignof`.

Ce que l'on verra avec le passage d'arguments.

Taille d'un tableau

Definition (Taille d'un tableau)

Tout tableau a une taille fixe et déterminée à sa construction. L'opérateur :

`sizeof` *id_tableau*
`sizeof``(`*type_tableau*`)` ⟵`attention ()`

permet d'obtenir la **taille de l'espace mémoire occupé** par le tableau; cette taille est **exprimée en `char`**.

Attention

Le type **`char`** est généralement définit avec une représentation sur 8 bits, 1 octets, mais ce n'est pas obligatoire! Quelque soit cette taille **`sizeof`**`(`**`char`**`)` est égal à 1, par définition de l'opérateur.

Exemple

```
#include <stdio.h>

int main(void) {
  size_t s = sizeof(int [50]);
  printf("%lu\n",s);
  int t[49];
  s = sizeof t;
  printf("%lu\n",s);
}
```

```
$ make siz
cc -Wall -pedantic --std=c11 siz.c -o siz
$ ./siz
200
196
$
```


tableaux et arithmétique des pointeurs

⚠ Attention

On insiste bien sur le fait que le type tableau n'est pas le même que le type pointeur correspondant...

Definition

adresses et tableaux Pour un tableau $T\ t[N]$:

- t est l'adresse du premier élément du tableau (valeur assimilable à un $T*$);
- $\&t$ est l'adresse du tableau lui-même (valeur assimilable à un $T(*)[N]$, un pointeur vers un tableau de taille N).

L'opérateur **`sizeof`** appliqué à t renvoie la taille totale du tableau (la taille de 50 entiers), mais appliqué à $\&t$ renvoie la taille d'un pointeur sur tableau de taille N (la taille d'un pointeur donc)...

Exemple

```
#include <stdio.h>

int main(void) {
  int t[50];
  int *p = t;
  printf("%4ld␣%p␣%p\n",sizeof t,
         (void *)t,
         (void *)(t+1));
  printf("␣␣␣␣␣%p␣%p\n",(void *)p,
         (void *)(p+1));
  printf("%4ld␣%p␣%p\n",sizeof &t,
         (void *)&t,
         (void *)(&t+1));
  int (*pt)[50] = &t;
  printf("␣␣␣␣␣%p␣%p\n",(void *)pt,
         (void *)(pt+1));
}
```



```
$ ./siztab
 200 0x7fff584f9930 0x7fff584f9934
     0x7fff584f9930 0x7fff584f9934
   8 0x7fff584f9930 0x7fff584f99f8
     0x7fff584f9930 0x7fff584f99f8
$
```

Paramètre tableau

Definition (tableau en paramètre)

Le véritable type d'un paramètre «tableau de *T*» est «pointeur sur *T*». Par conséquent, les tableaux ne sont pas transmis en paramètre, ce sont simplement leur adresse qui l'est.

⚠ Attention

Si vous ne connaissez pas la taille d'un tableau, il n'existe aucun moyen de la déterminer!
La syntaxe est donc trompeuse et c'est ici que les ambiguïtés et les confusions surviennent entre tableaux et pointeurs.


```c
#include <stdio.h>

void f(int t[50]) { // ce n'est pas tableau de 50 entiers!!!
  size_t s = sizeof(t);
  printf("%lu\n",s);
}

int main(void) {
  int t[49];
  f(t);
}
```

```
$ gcc -o siz2 siz2.c
siz2.c:4:20: warning: sizeof on array function parameter
    will return size of 'int *' instead of 'int [50]' [-
    Wsizeof-array-argument]
  size_t s = sizeof(t);
                    ^
siz2.c:3:12: note: declared here
void f(int t[50]) {
           ^
1 warning generated.
$ ./siz2
8                ⟵ Aïe, aïe!
$
```

```c
#include <stdio.h>

void f(int t[50]) { // ce n'est pas tableau de 50 entiers!!!
  size_t s = sizeof(t);
  printf("%lu\n",s);
}

int main(void) {
  int t[49];
  f(t);
}
```

Remarquez que la taille `50` ne sert à rien! On a même pu passer un tableau de 49 entiers...
Il existe un maigre palliatif à ce problème : la contrainte de taille...

Taille statique

Definition (paramètre tableau et taille statique)

La déclaration d'un paramètre tableau de la forme suivante :

type id`(___,` *type id*`[`**`static`** *taille*`],` `___)`

permet de spécifier que l'argument passé est l'adresse d'un tableau contenant **au moins** *taille* > 0 éléments.

⚠ Attention

Il n'est pas toujours possible au compilateur de vérifier cette contrainte! Vous n'êtes donc pas intégralement protégé...

```c
#include <stdio.h>

void f(int t[static 10]) {
  printf("%lu\n",sizeof t);
}

void g(int t[]) {
  f(t);
}

int main(void) {
  int t[10], t2[11], t3[9];
  f(t); f(t2); f(t3); f(NULL);
  g(t3);
}
```



```
$ gcc -Wall --std=c11 -o stattab stattab.c
stattab.c:4:25: warning: sizeof on array function parameter will
     return size of 'int *' instead of 'int [static 10]' [-Wsizeof-
     array-argument]
  printf("%lu\n",sizeof t);
                        ^
stattab.c:3:12: note: declared here
void f(int t[static 10]) {
           ^
stattab.c:13:16: warning: array argument is too small; contains 9
     elements, callee requires at least 10 [-Warray-bounds]
  f(t); f(t2); f(t3); f(NULL);
               ^ ~~
stattab.c:3:12: note: callee declares array parameter as static here
void f(int t[static 10]) {
           ^~~~~~~~~~~~
stattab.c:13:23: warning: null passed to a callee that requires a non
     -null argument [-Wnonnull]
  f(t); f(t2); f(t3); f(NULL);
                      ^ ~~~~
stattab.c:3:12: note: callee declares array parameter as static here
void f(int t[static 10]) {
           ^~~~~~~~~~~~
3 warnings generated.
$
$ ./stattab
8
8
8
8
8
$
```

Tableaux de taille variable - *aka VLA*

Cette construction n"est pas nécessairement implantée, le standard la définit comme optionnelle.

Definition (VLA)

Un tableau de taille variable — *variable length array* est un tableau dont la taille est déterminée à l'exécution, c'est-à-dire que la taille à sa construction n'est pas déterminée par une constante mais par la valeur d'une expression calculée durant l'exécution. Ces tableaux doivent nécessairement avoir la visibilité d'un bloc.

⚠ Attention

Le terme VLA est ambigu puisqu'un tableau a une taille fixée et non modifiable pour toute sa durée de vie.

Exemple

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

void f() {
  int t[random()%10+1];
  printf("%lu\n",sizeof t/sizeof t[0]);
  int t2[] = { [random()%10]= 666 };
}

int main(void) {
  srandom((unsigned)time(NULL));
  f();
  f();
  f();
}
```

```
$ gcc -Wall -o tabvla tabvla.c
$ ./tabvla
6
4
9
$
```

Pour une piqûre de rappel à propos de `srandom`, `random` et `time`, voir p. 300.

Initialisation d'un tableau

Definition (Initialisation d'un tableau)

L'initialisation d'un tableau consiste à fournir des valeurs pour ses éléments :

type id`[`*taille*`]` `=` `{` v_0`,` v_1`,` `___`, v_k `};`

Si $k < \mathit{taille} - 1$, les «derniers» éléments sont initialisés selon la règle applicable aux données **`static`** (donc `0` ou `NULL`).

```c
#include <stdio.h>

void print(float *t,int size) {
  for (int i=0; i<size; i++)
    printf("t[%d]=%f ",i,t[i]);
  putchar('\n');
}

int main(void) {
  float t[5] = { 3.14, 1.432, 299E6 };
  print(t,sizeof(t)/sizeof(t[0]));
}
```

```
$ make inittab
cc -Wall -pedantic --std=c11    inittab.c   -o inittab
...quelques warnings à propos des initialisations...
8 warnings generated.
$ ./inittab
./inittab
t[0]=3.140000 t[1]=1.414000 t[2]=299000000.000000 t
    [3]=0.000000 t[4]=0.000000
$
```


Definition (Initialisation d'un tableau (bis))

L'initialisation peut aussi servir à déterminer la taille d'un tableau, ainsi :

type id[] = { V_0, V_1, ____, $V_{taille-1}$ };

permet d'obtenir un tableau de *taille* indiquée, qui est intégralement et explicitement initialisé.

Initialisation désignée d'un tableau

Definition (Initialisation d'un tableau (ter))

L'initialisation peut aussi servir à déterminer la taille d'un tableau, ainsi :

type id`[`*taille*`]` `=` `{` `[`i_1`]=`v_{i_1}`,` `[`i_2`]=`v_{i_2}`,` `____` `};`

permet d'obtenir un tableau de *taille* indiquée, et dont les éléments d'indices spécifiés sont initialisés explicitement (les autres sont sujets à la règle **`static`**).

⚠ Attention

La syntaxe d'initialisation désignée par intervalle est propre à `gcc` et donc non portable :

```
int t[] = { [10 ... 20]=666, [55]=999 };
```


Exemple

```
#include <stdio.h>

int t[] = { [1]=111, [5]=666, [3]=333 };

int main(void) {
  for (int i=0; i<sizeof t/sizeof t[0]; i++) {
    printf("t[%d]=%d\n",i,t[i]);
  }
}
```

```
$ gcc -o tabdesign tabdesign.c
$ ./tabdesign
t[0]=0
t[1]=111
t[2]=0
t[3]=333
t[4]=0
t[5]=666
$
```


Tableau multidimensionnel

Definition (Définition d'un tableau)

La définition d'un tableau multidimensionnel consiste à lui associer une suite de tailles, ainsi :

type *identificateur* [$entier_1$] [$entier_2$] ____ [$entier_n$] ;

Le tableau possède n dimensions, et doit être interprété comme un tableau de tableaux de tableaux...d'éléments. De la sorte les éléments sont contigus en mémoire et rangés dans l'ordre inverse de leurs dimensions.

Structures

Definition (Structure)

Décrit (**structure**) une **zone non-vide et contiguë de mémoire** comme un ensemble de valeurs de possiblement différents types appelés **membres**.
Cela correspond à la définition d'un produit cartésien.
La définition courante a la forme générale suivante :

```
struct id {
  type identificateur;
  ____
};
```

Cela définit un nouveau type de nom **`struct`** *id*. *id* n'est pas un type, techniquement il s'agit d'un **tag**.

Exemple

```
struct point {
  int abscisse;
  int ordonnee;
};
```

Une telle structure possède deux champs de type **`int`** respectivement dénommés `abscisse` et `ordonnee`.

⚠ Attention

La taille d'une structure est **au moins** égale à la somme des tailles de ses membres (des règles d'alignement peuvent être à l'œuvre et modifier la donne, voir p.445).

Déclaration d'une variable structurée

`struct` *id identificateur*`;`

permet d'obtenir la définition d'une variable d'un type structuré définit auparavant.

Accès aux champs d'une structure `.` et `->`

Definition (opérateur d'accès)

Les opérateurs `.` et `->` permettant à partir d'une variable d'un type structuré, respectivement d'un pointeur sur, d'accéder à un membre.
Ces opérateurs vérifient :

- `s.c` ⇔ `(&s)->c`
- `p->c` ⇔ `(*p).c`

Exemple

```
struct point {
  int abscisse;
  int ordonnee;
};
struct point un_point;
un_point.abscisse = 0;
un_point.ordonnee = 0;
struct point *p_point = &un_point;
p_point->abscisse = 10;
p_point->ordonnee = 10;
```


Définition et instanciation d'une structure

Exemple

Il est possible de définir une structure et des variables du type dans le même temps :

struct *nom* {

} var_1, var_2;

Structures anonymes

Definition

Il est possible de définir des structures anonymes, c'est-à-dire dont le type ne porte pas de nom. Il suffit de ne pas spécifier d'identificateur pour la structure.

Il y a deux usages possibles :

1. définition instanciée d'un type à caractère «local», c'est-à-dire qui n'est pas destiné à être utilisé ailleurs,
2. définition d'une sous-structure sans nom.

Exemple

```
struct {
  int abscisse;
  int ordonnee;
} v1;
v1.abscisse = 0;
v1.ordonnee = 0;
```

Exemple

```
struct s {
  struct {
    int c1;
    int c2;
  };
};
```

dans ce cas, les membres de la structure interne anonyme sont considérés comme des membres de la structure directement englobante :

```
struct s v;
v.c1 = 0;
```


le C c'est un poil tordu, non ?

⚠ Attention

Il n'y a pas d'imbrication des types! La définition d'un type en C ne correspond pas à celle d'un espace de noms...

La définition :

```
struct s1 {
  struct s2 { ___ } c;
};
```

est strictement équivalente à la suivante :

```
struct s1;
struct s2 { ___ };
struct s1 {
  struct s2 c;
};
```

Question

```
struct x {
  struct y { int y; } z;
  struct { int x; } y;
  struct { int x; };
} x;
```

Décrivez cette définition...

Initialisation des structures

L'initialisation des structures peut être effectuée à leur définition :

```
struct S {
  int m1;
  int m2;
};
struct S s = { 12, 34 };
```

On peut aussi, comme pour les tableaux, utiliser une initialisation «partielle» (voir p. 389) qui aura pour effet d'initialiser à la valeur par défaut les membres restants :

```
struct S {
  int m1;
  int m2;
  int m3;
};
struct S s = { 12 };
```

Il existe aussi une initialisation désignée comme pour les tableaux (voir p. 392) :

```
struct S {
  int m1;
  int m2;
  int m3;
};
struct S s = { .m3=12, .m1=5 };
```


rl-values et structures

On rappelle qu'un identificateur placé en position de r-value désigne la valeur de la variable.

Definition (r-value d'une structure)

c'est la valeur qui représente le tuple constitué des valeurs des membres.

Lorsqu'un identificateur est placé en l-value il désigne le contenant.

Definition (affectation)

L'affectation d'une variable d'un type à une variable du même type correspond à la recopie de la zone mémoire de la première dans celui de la seconde.

Par conséquent l'affectation de variables de structure fonctionne :

```
#include <stdio.h>

struct point { int a; int o; };

int main(void) {
  struct point p1 = { 0, 0 };
  struct point p2 = { 10 };
  printf("%d,%d\n",p1.a,p1.o);
  printf("%d,%d\n",p2.a,p2.o);
  p2 = p1;
  printf("%d,%d\n",p1.a,p1.o);
  printf("%d,%d\n",p2.a,p2.o);
}
```



```
$ ./affectstru
0,0
10,0
0,0
0,0
$
```

ainsi que le passage de structures en paramètre :

```
#include <stdio.h>

struct point { int a; int o; };

void print_point(struct point p) {
  printf("%d,%d\n",p.a,p.o);
}

int main(void) {
  struct point p1 = { 0, 0 };
  struct point p2 = { 10 };
  print_point(p1);
  print_point(p2);
  p2 = p1;
  print_point(p1);
  print_point(p2);
}
```

On insiste sur le fait que les arguments sont passés par valeur (voir p. 218).

Littéraux composés

Definition (Littéral composé)

permet la définition d'un «objet» anonyme (en général un type agrégat). Une expression définissant un littéral composé est constituée d'un type spécifié entre parenthèses, suivi d'une initialisation.

Exemple

```
#include <stdio.h>

struct pt { int x; int y; };

void print(struct pt p) {
  printf("(%d,%d)\n",p.x,p.y);
}

int main(void) {
  struct pt p = { .x = 10, .y = 20 };
  print(p);
  print((struct pt){.x=666,.y=42});
}
```



```
$ ./compound
(10,20)
(666,42)
$
```

Exemple

```
#include <stdio.h>

void print(int t[static 2]) {
  for (int i=0; i<2; i++)
    printf("%d ",t[i]);
  putchar('\n');
}

void printInt(int v) {
  printf("%d\n",v);
}

int main(void) {
  print((int [2]){666,42});
  printInt((int){2});
}
```



```
$ ./compound2
666 42
$
```

On peut prendre l'adresse d'un littéral composé :

Exemple

```
#include <stdio.h>

void printInt(int *v) {
  printf("%p %d\n",(void *)v,*v);
}

int main(void) {
  printInt(&(int){2});
}
```

```
$ ./compound3
0x7fff5ac769dc 2
$
```


taille d'une structure

Definition (Taille d'une structure)

La taille d'une structure est la taille de l'emplacement mémoire permettant de stocker une valeur du type. Elle est au moins égale à la somme des tailles de chaque constituant (il ne faut pas oublier l'alignement, voir p. 445).

Exemple

```
#include <stdio.h>

struct point_2d {
  double abscisse;
  double ordonnée;
};

struct point_3d {
  int x, y, z;
};

int main(void) {
  printf("%lu\n",sizeof(struct point_2d));
  printf("%lu\n",sizeof(struct point_3d));
}
```



```
$ ./sizestruct
16
12
$
```

Structures et tableaux

Rappelons qu'il est autorisé d'assigner le contenu d'une structure à une autre (affectation entre objets du même type, exception faite des tableaux).
Par conséquent il est possible de recopier des tableaux l'un dans l'autre, à condition qu'ils soient placés dans une structure.

Exemple

```
#include <stdio.h>

struct S { int t[5]; };

void f(int *t) {
  t[0] = 666;
}

void g(struct S s) {
  s.t[0] = 666;
}

int main(void) {
  int t[5] = { 0, 0, 0, 0, 0 };
  printf("t[0]=%d\n",t[0]);
  f(t);
  printf("t[0]=%d\n",t[0]);
  struct S s = { .t={0, 0, 0, 0, 0} };
  printf("s.t[0]=%d\n",s.t[0]);
  g(s);
  printf("s.t[0]=%d\n",s.t[0]);
}
```

```
$ ./argstructarray
t[0]=0
t[0]=666
s.t[0]=0
s.t[0]=0
$
```


flexible array member

Definition (flexible array member)

Est ainsi appelé le dernier membre d'une structure, membre de type tableau et dont la taille est non spécifiée (il s'agit alors d'un type incomplet).

Exemple

```
struct T {
  int t;
  int v[];
};
```

Ce type de définition n'est utile qu'avec l'allocation dynamique. Le dernier membre se comporte alors comme un tableau dont la taille correspond à ce qui a été obtenu avec l'allocation.

Attention

La gestion du dernier élément est de la responsabilité du programmeur. Pour le compilateur, la structure est dans la plupart des cas, identique à celle ne contenant pas le membre flexible.

Exemple

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

struct T {
  int taille;
  int m[];
};

void print(struct T *t) {
  for (int i=0; i<t->taille; i++) printf("%d␣",t->m[i]);
  putchar('\n');
}

int main(void) {
  struct T *t = calloc(1,sizeof(struct T) + 4*sizeof t->m
      [0]);
  t->taille = 4;
  for (int i=0; i<t->taille; i++) t->m[i] = i;
  print(t);
  struct T *t2 = calloc(1,sizeof(struct T) + 4*sizeof t2->m
      [0]);
  *t2 = *t;
  print(t2);
  free(t2);
  free(t);
}
```

```
$ ./fam
0 1 2 3
0 0 0 0
$
```


Les unions

Definition (union)

Dans sa forme syntaxique les **union**s sont similaires aux **struct**ures (y compris en ce qui concerne les membres anonymes), mais contrairement aux structures les espaces de stockage de ses membres se recouvrent. L'adresse de chacun des membres est l'adresse de l'union elle-même.
La taille d'une union est au moins aussi grande que la taille du plus grand de ses membres (attention au bourrage, voir p. **??**).

Exemple

```
#include <stdio.h>

union U {
  int i;
  int j;
  short s;
  double d;
};

int main(void) {
  union U u;
  u.i = 12;
  printf("%d\n",u.i);
  printf("%d\n",u.j);
  u.d = 5.3;
  printf("%f\n",u.d);
}
```


produit :

```
$ ./union
12
12
5.300000
$
```

Attention

Lorsqu'une valeur est assignée à l'un des membres d'une union, la seule lecture autorisée est celle de ce membre ou d'un membre du même type (autrement qualifié ou non). Sinon le comportement est indéfini.

Mauvais exemple

```
union U {
  int          i;
  double d;
};
...
union U u;
u.i = 12;
printf("%f\n",u.d);
```


Exemple

```
#include <stdio.h>

union U {
  int i;
  short s;
  double d;
  char c[11];
};

int main(void) {
  union U u;
  printf("%lu\n",sizeof u.i);
  printf("%lu\n",sizeof u.s);
  printf("%lu\n",sizeof u.d);
  printf("%lu\n",sizeof u.c);
  printf("%lu\n",sizeof(union U));
}
```

produit :

```
$ ./sizeunion
4
2
8
11
16
$
```

En effet, le plus grand des membres est `c` de taille 11 octets, mais l'union est sujette au bourrage afin d'assurer l'alignement, la taille est donc de 16 octets.

déclarations anticipées

Dans le cas où l'on souhaite obtenir une définition croisée de types pointant l'un vers l'autre, il est nécessaire de faire des déclarations anticipées.

Definition (déclaration d'un type incomplet)

```
struct S;
enum E;
union U;
typedef T A;
```

déclarent un type incomplet (sauf dans le cas où le type considéré est déjà défini). Un type incomplet n'a pas de taille et aucune variable du type considéré ne peut être définie; il est par contre possible de définir un pointeur sur un type incomplet. Un type incomplet peut-être complété par une définition du type.

Note

Durant la définition d'un type, ce type lui même est incomplet.

Note

typedef permet d'obtenir la déclaration d'un alias de type, voir p. 457.

Exemple

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

struct liste {
  int valeur;
  struct liste *next;
};

struct T;

int main(void) {
  struct liste t = { 0, NULL };
}
```

Exemple

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

struct liste *t;

struct liste {
  int valeur;
  struct liste *next;
};

struct T;

int main(void) {
}
```


Exemple

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

typedef struct liste liste;
liste *l;

struct liste {
  int valeur;
  liste *next;
};

struct T;

int main(void) {
}
```

Exemple

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

typedef struct liste {
  int valeur;
  struct liste *next;
} liste;

struct T;

int main(void) {
  liste l;
}
```


Exemple (structures croisées)

```
typedef struct T T;

struct S {
  T *p;
};

typedef struct S S;

struct T {
  S *p;
};
```

champs de bits

Definition (champ de bits)

est la définition d'un type entier dont la taille est spécifiée. Cette dernière ne peut être plus grande que la taille du type considéré.
Un **champ de bit** (*bit-field*) ne peut apparaître que comme membre d'une structure ou d'une union.

Exemple

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

struct mon_type {
  unsigned int b1: 1;
  unsigned int b4: 4;
};


int main(void) {
  struct mon_type v;
  v.b1 = v.b4 = 0;
  for (int i=0; i<10; i++) printf("%d␣",v.b1++);
  putchar('\n');
  for (int i=0; i<20; i++) printf("%d␣",v.b4++);
  putchar('\n');
}
```

```
$ ./bit
0 1 0 1 0 1 0 1 0 1
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2
   3
$
```


alignement

Definition (Alignement)

On appelle **alignement** (*alignment*) d'un type la **contrainte sur les adresses mémoires valides** en vue stocker des objets du type.
Un **mauvais alignement** conduit à un **comportement non défini**.
Les contraintes d'alignement en C sont nécessairement des contraintes de multiplicité sur la valeur brute des adresses, *i.e.* tel type est toujours aligné sur des adresses multiples de l'alignement.
Le seul type qui a la plus faible contrainte est le type `char` pour lequel toutes les adresses sont potentiellement valides (sauf le pointeur nul), son alignement est 1.

La raison de l'existence d'un alignement strict est souvent la performance (pour les processeurs modernes les plus courants) ou l'impossibilité physique (pour certaines architectures embarquées). Si la violation de l'alignement est possible, la pénalité induite peut-être rédhibitoire en cas d'accès fréquent à des variables mal alignées.

Note

Les fonctions d'allocation mémoire garantissent l'alignement de toutes les allocations par le choix d'un alignement particulier dit **alignement fondamental**.

L'alignement en tant que tel ne pose pas de réel problème, c'est sur la constitution de types agrégés qu'il a une influence : le bourrage.

bourrage

Definition (Bourrage)

Le **bourrage** (*padding*) est le procédé consistant à ajouter de l'espace supplémentaire à un type donné afin d'assurer la correction de son propre alignement ou de données qui lui succèderaient.

Par exemple, on sait que le placement mémoire des membres d'une structure suit celui de leur déclaration, mais les contraintes d'alignement peuvent alors imposer de procéder à du bourrage.

Exemple

Prenons le cas d'une structure :

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

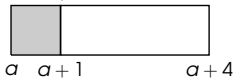
Question

Si la taille de `char` est 1 et celle des `int` est 4 on pourrait s'attendre à ce que la structure soit de taille 5, mais une expérience mène (par exemple) à la valeur 8. Pourquoi ?

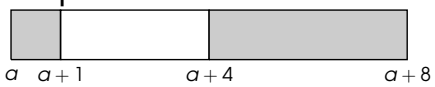
Il faut d'abord s'enquérir de l'alignement des types : pour `char` et pour `int` c'est 4.
Ensuite rappelons qu'un tel objet est normalement fondamentalement aligné, disons qu'il s'agit d'une adresse a :

Ensuite et tout naturellement, le premier membre de la structure est placé à cette adresse, a, l'espace libre suivant est $a+1$:

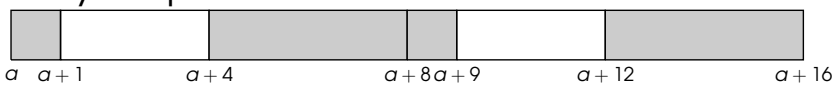
Le second membre de type `int` ne peut être placé en $a+1$ car il violerait sa contrainte d'alignement (rappelons que c'est 4), il est donc nécessaire d'ajouter 3 octets de bourrage :



puis d'y placer l'entier, donc à l'adresse $a+4$, l'espace libre suivant est en $a+8$:



Essayons pour un tableau :



Notre structure ne pose pas d'autres problèmes, sa taille est donc 8 octets.

Exemple

Prenons un autre cas :

```
struct S {
  char c;
  long l;
  int i;
};
```

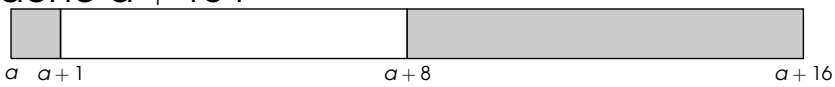
Question

Si la taille de `char` est 1, celle des `int` est 4 et celle des `long` est 8 on pourrait s'attendre à ce que la structure soit de taille 13, une expérience mène (par exemple) à la valeur 24. Pourquoi ?

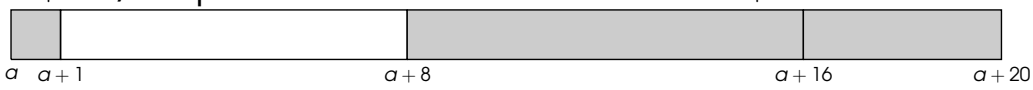
Il faut d'abord s'enquérir de l'alignement des types : pour `char` c'est 1, pour `int` c'est 4 et `long` 8.
Partons de l'adresse fondamentalement alignée a :

Le premier membre est placé en a, l'espace libre suivant est donc $a + 1$:

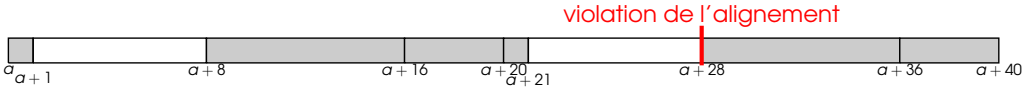
Le second membre de type `long` ne peut être placé en $a + 1$, il faut ajouter un bourrage de 7 octets et le placer à l'adresse $a + 8$, l'espace libre suivant est donc $a + 16$:



Le troisième membre est un `int` et peut-être placé en $a + 16$, l'espace libre suivant est donc $a + 20$:

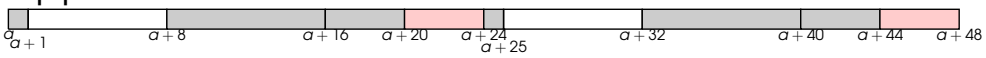


Regardons pour un tableau :



Si on laisse les choses en l'état, on aurait donc pour un élément qui suivrait (on rappelle que les éléments d'un tableau sont consécutifs en mémoire) : le premier membre à l'adresse $a + 20$ et le second membre à l'adresse $a + 28$ ce qui viole l'alignement requis pour un `long` (28 n'est pas divisible par 8).

Il est donc nécessaire d'assurer l'alignement de celui-ci, en continuant à assurer l'alignement des autres, ce qui requiert de choisir l'adresse $a + 32$, il fallait donc ajouter un bourrage de 4 octets supplémentaire à la fin de la structure!



C'est pourquoi la taille de la structure est de 24 octets et non pas 20.

Note

On remarquera que le bourrage est dépendant de l'ordre des membres (ou plutôt du type de ceux-ci). L'ordre qui fournit le plus petit bourrage est l'ordre inverse des alignements des types des membres. Diminuer la taille d'une structure n'est pas anodin, c'est même l'art du **packing**.

Le problème est qu'il n'est pas toujours possible de modifier l'ordre des membres, songeons par exemple à la sérialization de données dans un format spécifique.

Il y a deux façons de s'en sortir :

- la première est simplement d'utiliser des fonctions spécifiques de lecture/décodage afin de remplir «manuellement» la structure;
- la seconde est de demander au compilateur de ne pas utiliser de bourrage ainsi que de violer les contraintes d'alignement; cette technique est efficace mais évidemment non portable. Pour cela il faut consulter le manuel de son compilateur relativement au packing (`#pragma pack`, `-fpack-struct` ou `__attribute__((packed))` par exemple pour `gcc` ou `LLVM`).