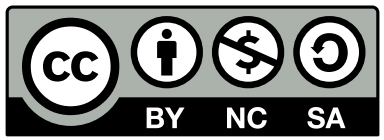
INF1135
Programmation procédurale
Chapitre 6 — Pointeurs

Alexandre Blondin Massé

Université du Québec à Montréal

v263



Plan

1. Rappel et motivation
2. Déclaration et déréférencement
3. Passage par adresse
4. Tableaux
5. Arithmétique des pointeurs
6. Pointeurs et `const`
7. Débogage

Plan

1. Rappel et motivation
2. Déclaration et déréférencement
3. Passage par adresse
4. Tableaux
5. Arithmétique des pointeurs
6. Pointeurs et `const`
7. Débogage

Rappel : une fonction ne modifie pas ses arguments

Fichier `swapfail.c` :

```c
#include <stdio.h>

void echanger(int a, int b) {
  int temp = a;
  a = b;
  b = temp;
}

int main(void) {
  int a = 1, b = 2;
  echanger(a, b);
  printf("a = %d, b = %d\n", a, b);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o swapfail
      swapfail.c
$ ./swapfail
a = 1, b = 2
```

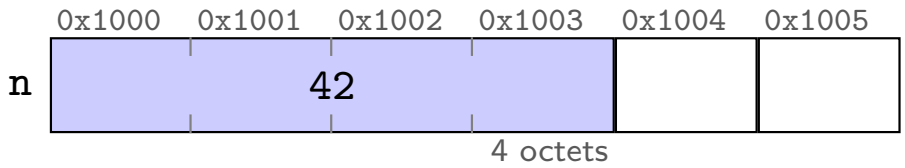
- Passage **par valeur** : `echanger` reçoit des **copies**
- Les copies sont échangées, puis disparaissent

Pourtant...

`scanf("%d", &n)` modifie bien `n` : on lui transmet l'**adresse** de `n`, et non une copie de sa valeur.

Rappel : adresses mémoire

- Mémoire : grande suite d'**octets** numérotés
- **Adresse** d'une variable : numéro de son premier octet
- `&n` : adresse de `n`, affichée avec `%p`



Questions

- Peut-on **conserver** une adresse dans une variable ?
- Peut-on ensuite lire ou modifier la variable qui se trouve à cette adresse ?

Réponse

Oui : c'est le rôle des **pointeurs**.

Plan

1. Rappel et motivation
2. Déclaration et déréférencement
3. Passage par adresse
4. Tableaux
5. Arithmétique des pointeurs
6. Pointeurs et `const`
7. Débogage

Exemple : modifier une variable par son adresse

Fichier `pointer.c` :

```c
#include <stdio.h>

int main(void) {
  int n = 42;
  int *p = &n;
  printf("n  = %d\n", n);
  printf("&n = %p\n", (void *) &n);
  printf("p  = %p\n", (void *) p);
  printf("*p = %d\n", *p);
  *p = 10;
  printf("n  = %d\n", n);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o pointer
    pointer.c
$ ./pointer
n  = 42
&n = 0x7ffe530987fc
p  = 0x7ffe530987fc
*p = 42
n  = 10
```

- `p` et `&n` : même adresse
- `n` est modifiée **sans** apparaître dans l'affectation `*p = 10`
- `void *` : pointeur **générique**, sans type pointé. `%p` l'exige, d'où la conversion `(void *)`. On y reviendra

Déclarer un pointeur

- **Pointeur** : variable qui contient une **adresse**
- On indique le **type** de la variable pointée, suivi de `*`

Exemples

```
int *p;          // p : pointeur vers un int
double *q;       // q : pointeur vers un double
int n = 42;
p = &n;          // p contient l'adresse de n
```

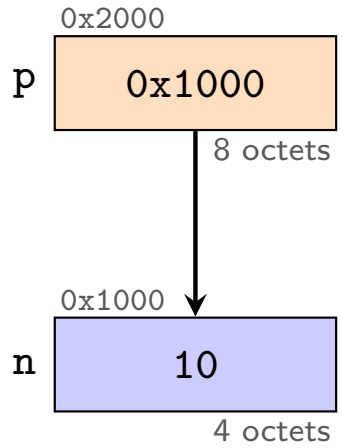
- On dit que `p` **pointe vers** `n`
- Le type de `p` est `int *` (« pointeur vers `int` »)

Piège

`int *p, q;` déclare un pointeur `p` et un **entier** `q` : l'étoile s'applique à un seul nom.

Opérateur * : déréférencement

- `*p` : la variable **pointée** par `p`
- Lecture et écriture :




```
int n = 42;
int *p = &n;
int x = *p;   // x vaut 42
*p = 10;      // n vaut 10
```

Deux usages de *

- Dans une **déclaration** : `int *p;` indique que `p` est un pointeur
- Dans une **expression** : `*p` désigne la variable pointée

`&` et `*` sont **inverses** : `*&n` est `n`.

Taille d'un pointeur

```
$ ./pointersize
sizeof(char)      = 1
sizeof(double)    = 8
sizeof(char *)    = 8
sizeof(int *)     = 8
sizeof(double *) = 8
```

- Tous les pointeurs ont la **même taille** : celle d'une adresse
- 8 octets (64 bits) sur une machine 64 bits

Alors pourquoi indiquer le type pointé ?

- Pour savoir **combien d'octets** lire ou écrire avec `*p` : 4 pour un `int`, 8 pour un `double`
- Pour savoir comment **interpréter** ces octets
- Pour que le compilateur détecte les erreurs : affecter `&x`, où `x` est un `double`, à un `int *` provoque un avertissement

Initialisation et NULL

- Comme toute variable locale, un pointeur non initialisé contient une valeur **quelconque** : il pointe n'importe où
- NULL : pointeur **nul**, qui ne pointe vers aucune variable

Exemple

```
int *p = NULL;  // ne pointe vers rien, pour l'instant
...
if (p != NULL) {
  printf("%d\n", *p);
}
```

Bonne pratique

Toujours initialiser un pointeur : avec une adresse valide, ou avec NULL. NULL est défini dans <stdio.h>, entre autres. Depuis C23, on peut aussi écrire nullptr.

Plan

1. Rappel et motivation
2. Déclaration et déréférencement
3. Passage par adresse
4. Tableaux
5. Arithmétique des pointeurs
6. Pointeurs et `const`
7. Débogage

Échanger deux variables

Fichier `swap.c` :

```c
#include <stdio.h>

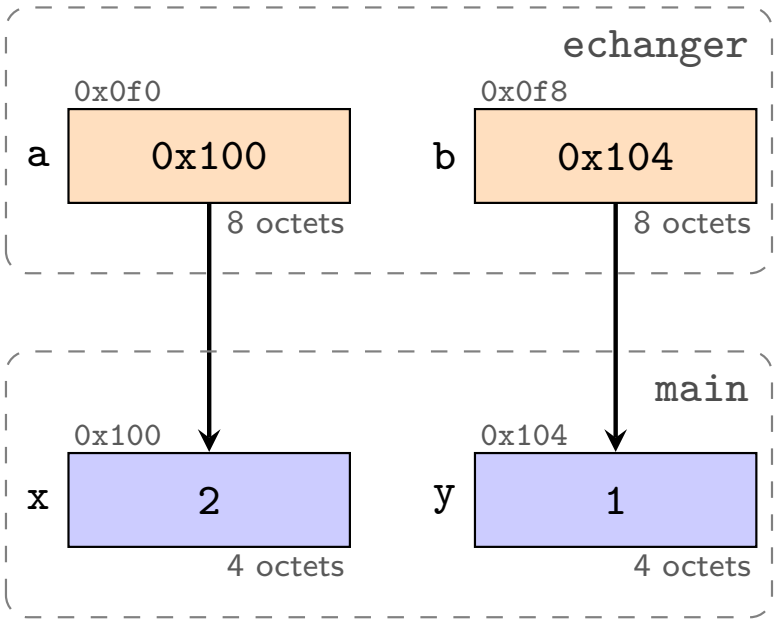
/**
 * Echange les valeurs de deux variables entieres.
 *
 * @param a  L'adresse de la premiere variable
 * @param b  L'adresse de la deuxieme variable
 */
void echanger(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

int main(void) {
  int x = 1, y = 2;
  echanger(&x, &y);
  printf("x = %d, y = %d\n", x, y);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o swap
      swap.c
$ ./swap
x = 2, y = 1
```

- Paramètres de type `int *` : `echanger` reçoit des **adresses**
- À l'appel : `&x` et `&y`
- Dans la fonction : `*a` et `*b` désignent `x` et `y`

Ce qui se passe en mémoire



- a et b sont des variables **locales** de echanger
- Elles reçoivent des **copies** des adresses &x et &y : c'est toujours un passage par valeur
- Mais ces copies désignent les variables de main : *a = *b modifie x

Passage par adresse

Pour qu'une fonction modifie une variable de l'appelant, on lui transmet l'**adresse** de cette variable.

Retourner plusieurs résultats

Fichier `divmod.c` :

```c
#include <stdio.h>

/**
 * Calcule le quotient et le reste de a / b.
 *
 * L'entier b doit etre non nul.
 *
 * @param a         Le dividende
 * @param b         Le diviseur
 * @param quotient  L'adresse ou ecrire le quotient
 * @param reste     L'adresse ou ecrire le reste
 */
void diviser(int a, int b, int *quotient, int *reste) {
  *quotient = a / b;
  *reste = a % b;
}

int main(void) {
  int q, r;
  diviser(17, 5, &q, &r);
  printf("17 = %d * 5 + %d\n", q, r);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o
      divmod divmod.c
$ ./divmod
17 = 3 * 5 + 2
```

- `return` ne retourne qu'**une** valeur
- `quotient` et `reste` : paramètres de **sortie**
- C'est exactement ce que fait `scanf`

Plan

1. Rappel et motivation
2. Déclaration et déréférencement
3. Passage par adresse
4. Tableaux
5. Arithmétique des pointeurs
6. Pointeurs et `const`
7. Débogage

Déclarer et parcourir un tableau

- **Tableau** : suite de variables de même type, désignées par un seul nom et un **indice**
- Les indices vont de 0 à `taille - 1`

Exemple

```
int t[5] = {3, 1, 4, 1, 5};   // 5 entiers
t[2] = 9;                     // 3e element
for (int i = 0; i < 5; i++) {
  printf("%d\n", t[i]);
}
```

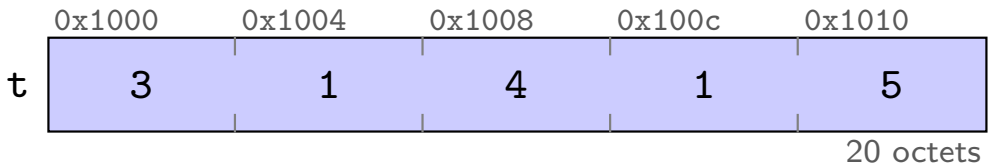
Premier contact

Juste ce qu'il faut pour comprendre le lien avec les pointeurs. On reviendra en détail sur les tableaux et sur les chaînes de caractères.

Le nom d'un tableau est une adresse

```c
int t[5] = {3, 1, 4, 1, 5};
for (int i = 0; i < 5; i++) {
  printf("t[%d] = %d  (adresse %p)\n", i, t[i], (void *) &t[i]);
}
printf("t         : %p\n", (void *) t);
```

```
$ ./arrayintro
t[0] = 3  (adresse 0x7ffcd8c23250)
t[1] = 1  (adresse 0x7ffcd8c23254)
t[2] = 4  (adresse 0x7ffcd8c23258)
t[3] = 1  (adresse 0x7ffcd8c2325c)
t[4] = 5  (adresse 0x7ffcd8c23260)
t         : 0x7ffcd8c23250
```



- Éléments **contigus** : les adresses augmentent de `sizeof(int)` $= 4$
- `t` vaut `&t[0]` : l'adresse du **premier** élément

Plan

1. Rappel et motivation
2. Déclaration et déréférencement
3. Passage par adresse
4. Tableaux
5. Arithmétique des pointeurs
6. Pointeurs et const
7. Débogage

Addition et soustraction

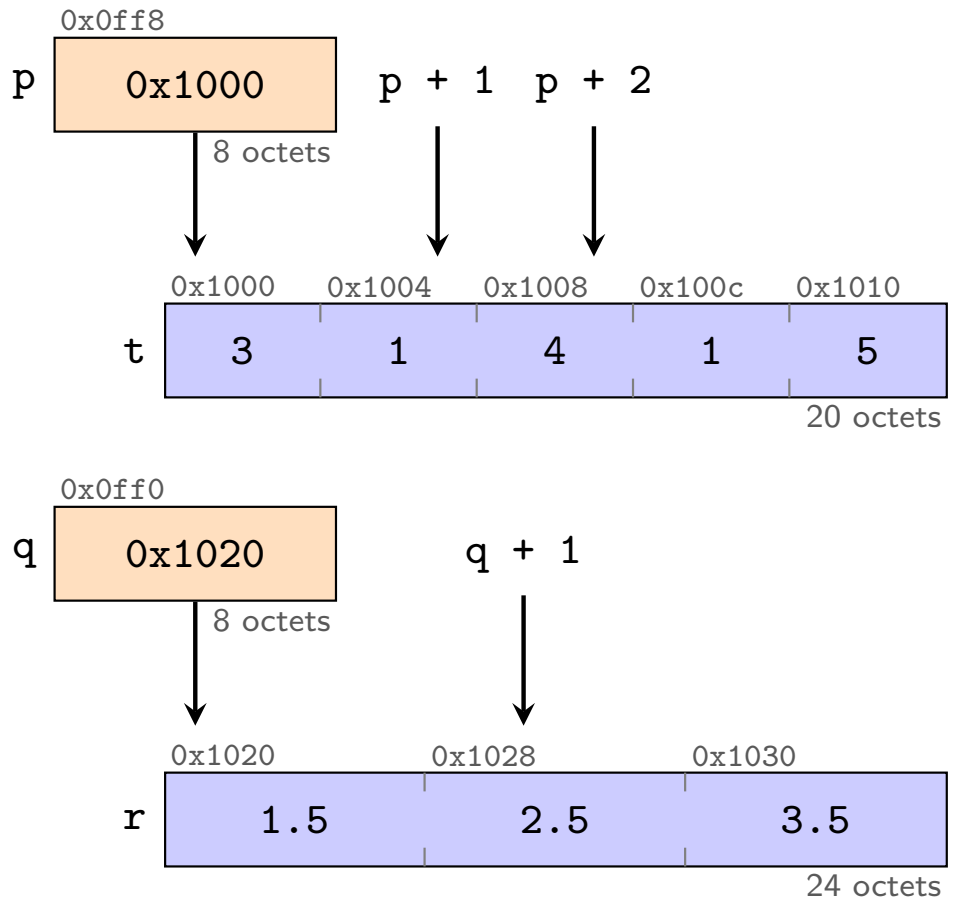
Fichier `arith.c` :

```
int t[5] = {3, 1, 4, 1, 5};
int *p = t;
printf("p     : %p -> %d\n",
       (void *) p, *p);
printf("p + 1 : %p -> %d\n",
       (void *) (p + 1), *(p + 1));
printf("p + 2 : %p -> %d\n",
       (void *) (p + 2), *(p + 2));

double r[3] = {1.5, 2.5, 3.5};
double *q = r;
printf("q     : %p\n", (void *) q);
printf("q + 1 : %p\n", (void *) (q + 1));

int *fin = &t[4];
printf("fin - p : %td\n", fin - p);
```

```
$ ./arith
p     : 0x7ffdb04d2c40 -> 3
p + 1 : 0x7ffdb04d2c44 -> 1
p + 2 : 0x7ffdb04d2c48 -> 4
q     : 0x7ffdb04d2c60
q + 1 : 0x7ffdb04d2c68
fin - p : 4
```



Règles de calcul

- `p + i` : adresse de l'élément situé `i` éléments plus loin, soit `i * sizeof(*p)` octets ($+4$ par `int`, $+8$ par `double`)
- `p - i`, `p++`, `p--` : même principe
- `fin - p` : nombre d'**éléments** entre deux pointeurs d'un même tableau (format `%td`)

Équivalences

$$\begin{aligned} \texttt{t[i]} &\equiv \texttt{*(t + i)} \\ \texttt{\&t[i]} &\equiv \texttt{t + i} \\ \texttt{t} &\equiv \texttt{\&t[0]} \end{aligned}$$

- La notation `t[i]` est un **raccourci** pour `*(t + i)`
- Les crochets s'utilisent donc aussi avec un pointeur : si `p = t`, alors `p[2]` est `t[2]`
- Les indices commencent à `0` parce qu'un indice est un **décalage** par rapport au début

Exemple : parcourir un tableau avec un pointeur

Fichier `arraywalk.c` :

```c
#include <stdio.h>

int main(void) {
  int t[5] = {3, 1, 4, 1, 5};

  int somme = 0;
  for (int i = 0; i < 5; i++) {
    somme += t[i];
  }
  printf("avec indices  : %d\n", somme);

  somme = 0;
  for (int *p = t; p < t + 5; p++) {
    somme += *p;
  }
  printf("avec pointeur : %d\n", somme);
  return 0;
}
```

```
$ ./arraywalk
avec indices  : 14
avec pointeur : 14
```

- p commence au premier élément et avance d'un élément à chaque tour
- `t + 5` : adresse **juste après** le dernier élément, qui sert de borne
- Les deux versions sont équivalentes ; celle avec indices est souvent plus lisible

Passer un tableau à une fonction

Fichier `arrayparam.c` :

```c
/**
 * Calcule la somme des elements de t.
 *
 * @param t  L'adresse du premier element
 * @param n  Le nombre d'elements
 * @return   La somme des n elements
 */
int somme(int *t, int n) {
  int s = 0;
  for (int i = 0; i < n; i++) {
    s += t[i];
  }
  return s;
}

/**
 * Double chacun des elements de t.
 *
 * @param t  L'adresse du premier element
 * @param n  Le nombre d'elements
 */
void doubler(int *t, int n) {
  for (int i = 0; i < n; i++) {
    t[i] = 2 * t[i];
  }
}
```

```c
int main(void) {
  int t[5] = {3, 1, 4, 1, 5};
  printf("somme = %d\n", somme(t, 5));
  doubler(t, 5);
  printf("somme = %d\n", somme(t, 5));
  printf("3 derniers = %d\n", somme(t + 2, 3));
  return 0;
}
```

```
$ ./arrayparam
somme = 14
somme = 28
3 derniers = 20
```

- La fonction reçoit l'**adresse** du premier élément, pas une copie : elle peut donc **modifier** les éléments
- Elle ignore la taille : il faut la transmettre (`n`)
- `t + 2` : le « tableau » qui commence au 3e élément

Débordements

```
int t[5] = {3, 1, 4, 1, 5};
printf("%d\n", t[5]);    // hors du tableau
*(t + 10) = 0;           // hors du tableau
```

- Aucune vérification des indices, ni à la compilation, ni à l'exécution
- On lit ou on écrit dans une zone mémoire qui appartient à **autre chose** : comportement **indéfini**
- Selon les cas : valeur absurde, autre variable modifiée, plantage, ou rien de visible... pour l'instant

À retenir

C'est au programmeur de garantir que chaque accès reste entre `t` et `t + n - 1`. Seule l'adresse `t + n` peut être **calculée** (comme borne), mais jamais déréférencée.

Plan

1. Rappel et motivation
2. Déclaration et déréférencement
3. Passage par adresse
4. Tableaux
5. Arithmétique des pointeurs
6. Pointeurs et `const`
7. Débogage

Rappel : le mot-clé const

```
const double v_ref = 3.3;
v_ref = 5.0;

error: assignment of read-only variable 'v_ref'
```

- Vu précédemment : const déclare une variable dont la valeur ne peut **pas** être modifiée après l'initialisation
- Le compilateur **refuse** toute modification : une erreur détectée tôt plutôt qu'un bogue à l'exécution
- Le code **documente** l'intention : cette valeur ne doit pas changer

Et avec les pointeurs ?

Passer une adresse à une fonction lui donne le droit de **modifier** la variable pointée. Souvent, on veut lui transmettre une adresse (par exemple celle d'un tableau) sans lui accorder ce droit : c'est le rôle de const.

Paramètres en lecture seule

Fichier `constparam.c` :

```c
/**
 * Calcule la somme des elements d'un tableau.
 *
 * @param t  L'adresse du premier element
 * @param n  Le nombre d'elements
 * @return   La somme des n elements
 */
int somme(const int *t, int n) {
  int s = 0;
  for (int i = 0; i < n; i++) {
    s += t[i];
    t[i] = 0;  // erreur volontaire
  }
  return s;
}

int main(void) {
  int t[5] = {3, 1, 4, 1, 5};
  printf("somme = %d\n", somme(t, 5));
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o constparam
      constparam.c
constparam.c: In function 'somme':
constparam.c:14:10: error: assignment of
      read-only location
    14 |     t[i] = 0;  // erreur
          volontaire
       |          ^
```

- `const int *t` : somme s'engage à **ne pas modifier** le tableau
- Le compilateur fait respecter cet engagement
- Le prototype **documente** aussi l'intention : on sait, sans lire le code, que le tableau sera intact

Pointeur vers une valeur constante

Fichier `constptr.c` :

```c
#include <stdio.h>

int main(void) {
  const double v_ref = 3.3;

  const double *p = &v_ref;
  printf("*p    = %.1f\n", *p);

  double *q = &v_ref;  // erreur
  *q = 5.0;
  printf("v_ref = %.1f\n", v_ref);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o constptr
    constptr.c
constptr.c: In function 'main':
constptr.c:9:15: warning: initialization
    discards 'const'
qualifier from pointer target type
    9 |   double *q = &v_ref;  // erreur
      |               ^
$ ./constptr
*p    = 3.3
v_ref = 5.0
```

- `p` est un `const double *` : on peut lire `*p`, mais `*p = 5.0;` serait une **erreur** de compilation
- `q` est un `double *` : il permettrait de modifier `v_ref`, d'où l'avertissement
- Ignorer l'avertissement : `v_ref`, pourtant `const`, vaut `5.0` ! Comportement **indéfini**
- À retenir : l'adresse d'une variable `const` se range dans un pointeur vers `const`

Trois combinaisons

Avec deux variables `int n` et `int m`, on peut déclarer `p` de trois façons :

Déclaration	`*p = 5;`	`p = &m;`
`const int *p = &n;`	interdit	permis
`int *const p = &n;`	permis	interdit
`const int *const p = &n;`	interdit	interdit

- `const int *p` : la valeur **pointée** ne peut pas être modifiée **via** `p`
- `int *const p` : le **pointeur** lui-même ne peut pas changer
- `n` reste modifiable directement : `const` ne protège que l'accès par ce pointeur

Truc de lecture

Lire de droite à gauche : `int *const p` se lit « `p` est un pointeur constant vers un `int` » ; `const int *p`, « `p` est un pointeur vers un `int` constant ».

Plan

1. Rappel et motivation
2. Déclaration et déréférencement
3. Passage par adresse
4. Tableaux
5. Arithmétique des pointeurs
6. Pointeurs et `const`
7. Débogage

Pointeur non initialisé et pointeur nul

```c
int main(void) {
  int *p;
  *p = 5;
  printf("*p = %d\n", *p);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o uninit uninit.
      c
uninit.c: In function 'main':
uninit.c:5:6: warning: 'p' is used
      uninitialized
    5 |   *p = 5;
      |   ~~~^~~
$ ./uninit
*p = 5
```

```c
int main(void) {
  int *p = NULL;
  printf("avant\n");
  *p = 5;
  printf("apres\n");
  return 0;
}
```

```
$ ./nullderef
avant
Segmentation fault (core dumped)
```

- À gauche : p pointe n'importe où. Le programme **semble** fonctionner : on a écrit 5 à un endroit inconnu
- À droite : déréférencer `NULL` provoque **toujours** un plantage, facile à repérer
- Ne pas ignorer l'avertissement `is used uninitialized` !

Adresse d'une variable disparue

Fichier `dangling.c` :

```c
#include <stdio.h>

int *creer_compteur(void) {
  int compteur = 0;
  return &compteur;
}

int main(void) {
  int *p = creer_compteur();
  printf("*p = %d\n", *p);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o dangling dangling.
      c
dangling.c: In function 'creer_compteur':
dangling.c:5:10: warning: function returns address
       of local variable
    5 |   return &compteur;
      |          ^~~~~~~~~
$ ./dangling
Segmentation fault (core dumped)
```

- `compteur` a une durée de vie **automatique** : elle disparaît à la fin de l'appel
- L'adresse retournée ne désigne plus aucune variable : pointeur **pendant**
- Ici, `gcc` remplace même l'adresse par `NULL`, d'où le plantage
- Solution : retourner la **valeur**, ou écrire le résultat à une adresse reçue en paramètre

Inspecter un pointeur

```
$ gdb -q pointer
Reading symbols from pointer...
(gdb) break 10
Breakpoint 1 at 0x11fe: file pointer.c, line 10.
(gdb) run
Breakpoint 1, main () at pointer.c:10
10          *p = 10;
(gdb) print n
$1 = 42
(gdb) print &n
$2 = (int *) 0x7fffffffd5bc
(gdb) print p
$3 = (int *) 0x7fffffffd5bc
(gdb) print *p
$4 = 42
(gdb) next
11          printf("n  = %d\n", n);
(gdb) print n
$5 = 10
```

- `print` accepte n'importe quelle expression C : `&n`, `*p`, `p - t`, ...
- gdb affiche le **type** d'un pointeur avec sa valeur : `(int *)`
- Après `next`, `n` a changé sans être nommée

Tableaux et mémoire : @ et x

```
$ gdb -q arraywalk
Reading symbols from arraywalk...
(gdb) break 14
Breakpoint 1 at 0x11f7: file arraywalk.c, line 14.
(gdb) run
Breakpoint 1, main () at arraywalk.c:14
14          somme += *p;
(gdb) print *t@5
$1 = {3, 1, 4, 1, 5}
(gdb) x/5dw t
0x7fffffffd5b0: 3       1       4       1
0x7fffffffd5c0: 5
(gdb) print p
$2 = (int *) 0x7fffffffd5b0
(gdb) continue
Breakpoint 1, main () at arraywalk.c:14
14          somme += *p;
(gdb) print p
$3 = (int *) 0x7fffffffd5b4
(gdb) print p - t
$4 = 1
```

- `print *t@5` : 5 éléments à partir de `*t`
- `x/5dw t` : examine la mémoire à partir de l'adresse `t`
  - 5 : nombre d'unités
  - `d` : format décimal (`x` : hexadécimal, `c` : caractère)
  - `w` : unités de 4 octets (`b` : 1, `h` : 2, `g` : 8)
- `p` avance de 4 octets à chaque tour

Après un plantage

Fichier `nullfunc.c` :

```c
#include <stdio.h>

void remettre_a_zero(int *n) {
  *n = 0;
}

int main(void) {
  int *p = NULL;
  printf("avant\n");
  remettre_a_zero(p);
  printf("apres\n");
  return 0;
}
```

```
$ gdb -q nullfunc
Reading symbols from nullfunc...
(gdb) run
avant

Program received signal SIGSEGV, Segmentation fault.
0x0000555555555159 in remettre_a_zero (n=0x0) at nullfunc.c
      :4
4           *n = 0;
(gdb) print n
$1 = (int *) 0x0
(gdb) backtrace
#0  0x0000555555555159 in remettre_a_zero (n=0x0) at
      nullfunc.c:4
#1  0x0000555555555191 in main () at nullfunc.c:10
(gdb) frame 1
#1  0x0000555555555191 in main () at nullfunc.c:10
10          remettre_a_zero(p);
(gdb) print p
$2 = (int *) 0x0
```

- Arrêt sur la ligne **fautive** : `n` vaut 0x0, soit `NULL`
- `backtrace` : `remettre_a_zero` a été appelée par `main`, à la ligne 10
- `frame 1` : on remonte dans `main` pour trouver l'**origine** du pointeur nul