Ch. 2 - Pointeurs

Langage C / C++

R. Absil

Haute École Bruxelles-Brabant
École supérieure d'Informatique

20 septembre 2021

Table des matières

Table des matières

Table des matières

Table des matières

Introduction

Overview

- Les pointeurs sont utilisés comme des étiquettes pour désigner d'autres objets
- On les utilise
  - si l'on veut propager en écriture des effets de bords (modifier un paramètre de fonction)
  - si on ne veut pas que des copies implicites de données soient effectuées
- Il y a des cas où l'on ne peut pas se passer de pointeurs

Idée

- Pointeur = adresse de « qqch »

Overview

- Les pointeurs sont utilisés comme des étiquettes pour désigner d'autres objets
- On les utilise
  - si l'on veut propager en écriture des effets de bords (modifier un paramètre de fonction)
  - si on ne veut pas que des copies implicites de données soient effectuées
- Il y a des cas où l'on ne peut pas se passer de pointeurs

Idée

- Pointeur = adresse de « qqch »

Overview

- Les pointeurs sont utilisés comme des étiquettes pour désigner d'autres objets
- On les utilise
  - si l'on veut propager en écriture des effets de bords (modifier un paramètre de fonction)
  - si on ne veut pas que des copies implicites de données soient effectuées
- Il y a des cas où l'on ne peut pas se passer de pointeurs

Idée

- Pointeur = adresse de « qqch »

Overview

- Les pointeurs sont utilisés comme des étiquettes pour désigner d'autres objets
- On les utilise
  - si l'on veut propager en écriture des effets de bords (modifier un paramètre de fonction)
  - si on ne veut pas que des copies implicites de données soient effectuées
- Il y a des cas où l'on ne peut pas se passer de pointeurs

Idée

- Pointeur = adresse de « qqch »

Overview

- Les pointeurs sont utilisés comme des étiquettes pour désigner d'autres objets
- On les utilise
  - si l'on veut propager en écriture des effets de bords (modifier un paramètre de fonction)
  - si on ne veut pas que des copies implicites de données soient effectuées
- Il y a des cas où l'on ne peut pas se passer de pointeurs

Idée

- Pointeur = adresse de « qqch »

Overview

- Les pointeurs sont utilisés comme des étiquettes pour désigner d'autres objets
- On les utilise
  - si l'on veut propager en écriture des effets de bords (modifier un paramètre de fonction)
  - si on ne veut pas que des copies implicites de données soient effectuées
- Il y a des cas où l'on ne peut pas se passer de pointeurs

Idée

- Pointeur = adresse de « qqch »

Différents types de pointeurs

- On peut avoir des pointeurs
  - de types de base
  - de structures
  - de fonctions
  - `void*`
- Les pointeurs `void*` sont des pointeurs « génériques » permettant de représenter des pointeurs de « n'importe quoi »
- En C, les pointeurs sont également utilisés pour manipuler des tableaux
  - En C++, on privilégie d'autres mécanismes

Différents types de pointeurs

- On peut avoir des pointeurs
  - de types de base
  - de structures
  - de fonctions
  - `void*`
- Les pointeurs `void*` sont des pointeurs « génériques » permettant de représenter des pointeurs de « n'importe quoi »
- En C, les pointeurs sont également utilisés pour manipuler des tableaux
  - En C++, on privilégie d'autres mécanismes

Différents types de pointeurs

- On peut avoir des pointeurs
  - de types de base
  - de structures
  - de fonctions
  - `void*`
- Les pointeurs `void*` sont des pointeurs « génériques » permettant de représenter des pointeurs de « n'importe quoi »
- En C, les pointeurs sont également utilisés pour manipuler des tableaux
  - En C++, on privilégie d'autres mécanismes

Différents types de pointeurs

- On peut avoir des pointeurs
  - de types de base
  - de structures
  - de fonctions
  - `void*`
- Les pointeurs `void*` sont des pointeurs « génériques » permettant de représenter des pointeurs de « n'importe quoi »
- En C, les pointeurs sont également utilisés pour manipuler des tableaux
  - En C++, on privilégie d'autres mécanismes

Différents types de pointeurs

- On peut avoir des pointeurs
  - de types de base
  - de structures
  - de fonctions
  - `void*`
- Les pointeurs `void*` sont des pointeurs « génériques » permettant de représenter des pointeurs de « n'importe quoi »
- En C, les pointeurs sont également utilisés pour manipuler des tableaux
  - En C++, on privilégie d'autres mécanismes

Différents types de pointeurs

- On peut avoir des pointeurs
  - de types de base
  - de structures
  - de fonctions
  - `void*`
- Les pointeurs `void*` sont des pointeurs « génériques » permettant de représenter des pointeurs de « n'importe quoi »
- En C, les pointeurs sont également utilisés pour manipuler des tableaux
  - En C++, on privilégie d'autres mécanismes

Différents types de pointeurs

- On peut avoir des pointeurs
  - de types de base
  - de structures
  - de fonctions
  - `void*`
- Les pointeurs `void*` sont des pointeurs « génériques » permettant de représenter des pointeurs de « n'importe quoi »
- En C, les pointeurs sont également utilisés pour manipuler des tableaux
  - En C++, on privilégie d'autres mécanismes

Différents types de pointeurs

- On peut avoir des pointeurs
  - de types de base
  - de structures
  - de fonctions
  - `void*`
- Les pointeurs `void*` sont des pointeurs « génériques » permettant de représenter des pointeurs de « n'importe quoi »
- En C, les pointeurs sont également utilisés pour manipuler des tableaux
  - En C++, on privilégie d'autres mécanismes

Syntaxe

Concept de pointeur

- Un pointeur vers un type `T` contient l'adresse d'un élément de type `T`
- On peut construire un pointeur en
  - l'affectant à un autre pointeur : `int * pt2 = pt1;`
  - prenant l'adresse d'une variable : `int * pt = &i;`
- On accède au contenu d'un pointeur avec `*`
  - On dit qu'on déférence le pointeur : `int j = *pt;`
- Un pointeur a une adresse, et est systématiquement de la taille du bus d'adresse
  - En 32 bits, `sizeof(T*)` est égal à 4
  - En 64 bits, `sizeof(T*)` est égal à 8
- On peut créer des `void*`
  - Pointeur vers un type `void`, incomplet
  - On ne peut pas déférencer un `void*`

Concept de pointeur

- Un pointeur vers un type `T` contient l'adresse d'un élément de type `T`
- On peut construire un pointeur en
  - l'affectant à un autre pointeur : `int * pt2 = pt1;`
  - prenant l'adresse d'une variable : `int * pt = &i;`
- On accède au contenu d'un pointeur avec `*`
  - On dit qu'on déférence le pointeur : `int j = *pt;`
- Un pointeur a une adresse, et est systématiquement de la taille du bus d'adresse
  - En 32 bits, `sizeof(T*)` est égal à 4
  - En 64 bits, `sizeof(T*)` est égal à 8
- On peut créer des `void*`
  - Pointeur vers un type `void`, incomplet
  - On ne peut pas déférencer un `void*`

Concept de pointeur

- Un pointeur vers un type `T` contient l'adresse d'un élément de type `T`
- On peut construire un pointeur en
  - l'affectant à un autre pointeur : `int * pt2 = pt1;`
  - prenant l'adresse d'une variable : `int * pt = &i;`
- On accède au contenu d'un pointeur avec `*`
  - On dit qu'on déférence le pointeur : `int j = *pt;`
- Un pointeur a une adresse, et est systématiquement de la taille du bus d'adresse
  - En 32 bits, `sizeof(T*)` est égal à 4
  - En 64 bits, `sizeof(T*)` est égal à 8
- On peut créer des `void*`
  - Pointeur vers un type `void`, incomplet
  - On ne peut pas déférencer un `void*`

Concept de pointeur

- Un pointeur vers un type `T` contient l'adresse d'un élément de type `T`
- On peut construire un pointeur en
  - l'affectant à un autre pointeur : `int * pt2 = pt1;`
  - prenant l'adresse d'une variable : `int * pt = &i;`
- On accède au contenu d'un pointeur avec `*`
  - On dit qu'on déférence le pointeur : `int j = *pt;`
- Un pointeur a une adresse, et est systématiquement de la taille du bus d'adresse
  - En 32 bits, `sizeof(T*)` est égal à 4
  - En 64 bits, `sizeof(T*)` est égal à 8
- On peut créer des `void*`
  - Pointeur vers un type `void`, incomplet
  - On ne peut pas déférencer un `void*`

Concept de pointeur

- Un pointeur vers un type `T` contient l'adresse d'un élément de type `T`
- On peut construire un pointeur en
  - l'affectant à un autre pointeur : `int * pt2 = pt1;`
  - prenant l'adresse d'une variable : `int * pt = &i;`
- On accède au contenu d'un pointeur avec `*`
  - On dit qu'on déférence le pointeur : `int j = *pt;`
- Un pointeur a une adresse, et est systématiquement de la taille du bus d'adresse
  - En 32 bits, `sizeof(T*)` est égal à 4
  - En 64 bits, `sizeof(T*)` est égal à 8
- On peut créer des `void*`
  - Pointeur vers un type `void`, incomplet
  - On ne peut pas déférencer un `void*`

Concept de pointeur

- Un pointeur vers un type `T` contient l'adresse d'un élément de type `T`
- On peut construire un pointeur en
  - l'affectant à un autre pointeur : `int * pt2 = pt1;`
  - prenant l'adresse d'une variable : `int * pt = &i;`
- On accède au contenu d'un pointeur avec `*`
  - On dit qu'on déférence le pointeur : `int j = *pt;`
- Un pointeur a une adresse, et est systématiquement de la taille du bus d'adresse
  - En 32 bits, `sizeof(T*)` est égal à 4
  - En 64 bits, `sizeof(T*)` est égal à 8
- On peut créer des `void*`
  - Pointeur vers un type `void`, incomplet
  - On ne peut pas déférencer un `void*`

Concept de pointeur

- Un pointeur vers un type `T` contient l'adresse d'un élément de type `T`
- On peut construire un pointeur en
  - l'affectant à un autre pointeur : `int * pt2 = pt1;`
  - prenant l'adresse d'une variable : `int * pt = &i;`
- On accède au contenu d'un pointeur avec `*`
  - On dit qu'on déférence le pointeur : `int j = *pt;`
- Un pointeur a une adresse, et est systématiquement de la taille du bus d'adresse
  - En 32 bits, `sizeof(T*)` est égal à 4
  - En 64 bits, `sizeof(T*)` est égal à 8
- On peut créer des `void*`
  - Pointeur vers un type `void`, incomplet
  - On ne peut pas déférencer un `void*`

Concept de pointeur

- Un pointeur vers un type `T` contient l'adresse d'un élément de type `T`
- On peut construire un pointeur en
  - l'affectant à un autre pointeur : `int * pt2 = pt1;`
  - prenant l'adresse d'une variable : `int * pt = &i;`
- On accède au contenu d'un pointeur avec `*`
  - On dit qu'on déférence le pointeur : `int j = *pt;`
- Un pointeur a une adresse, et est systématiquement de la taille du bus d'adresse
  - En 32 bits, `sizeof(T*)` est égal à 4
  - En 64 bits, `sizeof(T*)` est égal à 8
- On peut créer des `void*`
  - Pointeur vers un type `void`, incomplet
  - On ne peut pas déférencer un `void*`

Concept de pointeur

- Un pointeur vers un type `T` contient l'adresse d'un élément de type `T`
- On peut construire un pointeur en
  - l'affectant à un autre pointeur : `int * pt2 = pt1;`
  - prenant l'adresse d'une variable : `int * pt = &i;`
- On accède au contenu d'un pointeur avec `*`
  - On dit qu'on déférence le pointeur : `int j = *pt;`
- Un pointeur a une adresse, et est systématiquement de la taille du bus d'adresse
  - En 32 bits, `sizeof(T*)` est égal à 4
  - En 64 bits, `sizeof(T*)` est égal à 8
- On peut créer des `void*`
  - Pointeur vers un type `void`, incomplet
  - On ne peut pas déférencer un `void*`

Concept de pointeur

- Un pointeur vers un type `T` contient l'adresse d'un élément de type `T`
- On peut construire un pointeur en
  - l'affectant à un autre pointeur : `int * pt2 = pt1;`
  - prenant l'adresse d'une variable : `int * pt = &i;`
- On accède au contenu d'un pointeur avec `*`
  - On dit qu'on déférence le pointeur : `int j = *pt;`
- Un pointeur a une adresse, et est systématiquement de la taille du bus d'adresse
  - En 32 bits, `sizeof(T*)` est égal à 4
  - En 64 bits, `sizeof(T*)` est égal à 8
- On peut créer des `void*`
  - Pointeur vers un type `void`, incomplet
  - On ne peut pas déférencer un `void*`

Concept de pointeur

- Un pointeur vers un type `T` contient l'adresse d'un élément de type `T`
- On peut construire un pointeur en
  - l'affectant à un autre pointeur : `int * pt2 = pt1;`
  - prenant l'adresse d'une variable : `int * pt = &i;`
- On accède au contenu d'un pointeur avec `*`
  - On dit qu'on déférence le pointeur : `int j = *pt;`
- Un pointeur a une adresse, et est systématiquement de la taille du bus d'adresse
  - En 32 bits, `sizeof(T*)` est égal à 4
  - En 64 bits, `sizeof(T*)` est égal à 8
- On peut créer des `void*`
  - Pointeur vers un type `void`, incomplet
  - On ne peut pas déférencer un `void*`

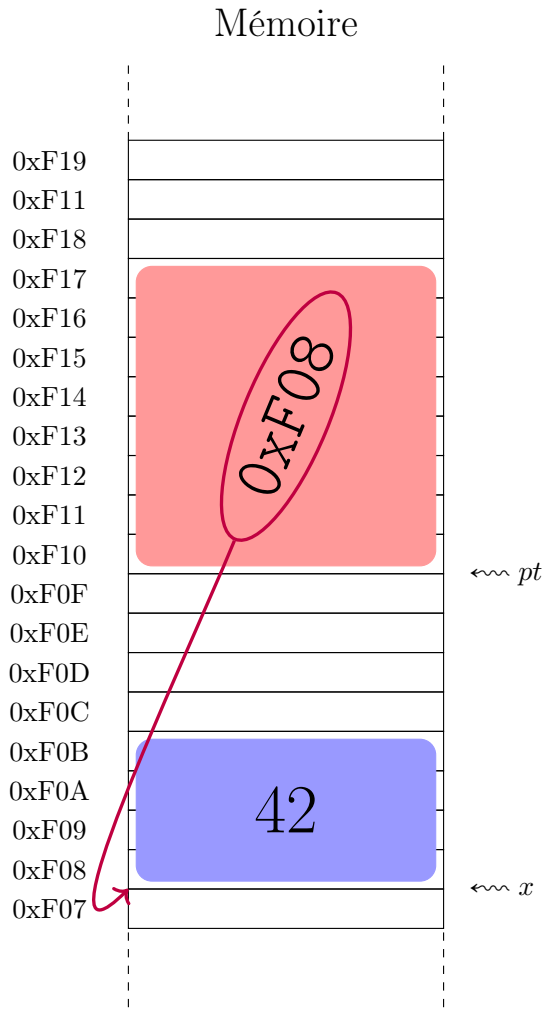
Concept de pointeur

- Un pointeur vers un type `T` contient l'adresse d'un élément de type `T`
- On peut construire un pointeur en
  - l'affectant à un autre pointeur : `int * pt2 = pt1;`
  - prenant l'adresse d'une variable : `int * pt = &i;`
- On accède au contenu d'un pointeur avec `*`
  - On dit qu'on déférence le pointeur : `int j = *pt;`
- Un pointeur a une adresse, et est systématiquement de la taille du bus d'adresse
  - En 32 bits, `sizeof(T*)` est égal à 4
  - En 64 bits, `sizeof(T*)` est égal à 8
- On peut créer des `void*`
  - Pointeur vers un type `void`, incomplet
  - On ne peut pas déférencer un `void*`

Illustration

```
{
   int x = 42;      //&x  = 0xF08
   int * pt = &x;   //&pt = 0xF10
   ...
}
```

```
sizeof(int) = 4
sizeof(int*) = 8 //x64
```



Pointeur nuls

- On peut créer des pointeurs nuls
  - En `C` : `NULL`
  - En `C++` : `nullptr`
- `NULL` est une macro : `#define NULL 0`
- `nullptr` est un immédiat
- Déférencer un pointeur nul a un comportement indéterminé
- Créer un pointeur n'alloue pas de mémoire vers la zone pointée
  - `int * p;` n'alloue pas `sizeof(int)` bytes en mémoire pour `*p`
- Pour qu'un pointeur référence de la mémoire allouée, il faut soit
  - que l'emplacement référencé soit pré-alloué
  - allouer un espace dynamiquement

Pointeur nuls

- On peut créer des pointeurs nuls
  - En `C` : `NULL`
  - En `C++` : `nullptr`
- `NULL` est une macro : `#define NULL 0`
- `nullptr` est un immédiat
- Déférencer un pointeur nul a un comportement indéterminé
- Créer un pointeur n'alloue pas de mémoire vers la zone pointée
  - `int * p;` n'alloue pas `sizeof(int)` bytes en mémoire pour `*p`
- Pour qu'un pointeur référence de la mémoire allouée, il faut soit
  - que l'emplacement référencé soit pré-alloué
  - allouer un espace dynamiquement

Pointeur nuls

- On peut créer des pointeurs nuls
  - En `C` : `NULL`
  - En `C++` : `nullptr`
- `NULL` est une macro : `#define NULL 0`
- `nullptr` est un immédiat
- Déférencer un pointeur nul a un comportement indéterminé
- Créer un pointeur n'alloue pas de mémoire vers la zone pointée
  - `int * p;` n'alloue pas `sizeof(int)` bytes en mémoire pour `*p`
- Pour qu'un pointeur référence de la mémoire allouée, il faut soit
  - que l'emplacement référencé soit pré-alloué
  - allouer un espace dynamiquement

Pointeur nuls

- On peut créer des pointeurs nuls
  - En `C` : `NULL`
  - En `C++` : `nullptr`
- `NULL` est une macro : `#define NULL 0`
- `nullptr` est un immédiat
- Déférencer un pointeur nul a un comportement indéterminé
- Créer un pointeur n'alloue pas de mémoire vers la zone pointée
  - `int * p;` n'alloue pas `sizeof(int)` bytes en mémoire pour `*p`
- Pour qu'un pointeur référence de la mémoire allouée, il faut soit
  - que l'emplacement référencé soit pré-alloué
  - allouer un espace dynamiquement

Pointeur nuls

- On peut créer des pointeurs nuls
  - En `C` : `NULL`
  - En `C++` : `nullptr`
- `NULL` est une macro : `#define NULL 0`
- `nullptr` est un immédiat
- Déférencer un pointeur nul a un comportement indéterminé
- Créer un pointeur n'alloue pas de mémoire vers la zone pointée
  - `int * p;` n'alloue pas `sizeof(int)` bytes en mémoire pour `*p`
- Pour qu'un pointeur référence de la mémoire allouée, il faut soit
  - que l'emplacement référencé soit pré-alloué
  - allouer un espace dynamiquement

Pointeur nuls

- On peut créer des pointeurs nuls
  - En `C` : `NULL`
  - En `C++` : `nullptr`
- `NULL` est une macro : `#define NULL 0`
- `nullptr` est un immédiat
- Déférencer un pointeur nul a un comportement indéterminé
- Créer un pointeur n'alloue pas de mémoire vers la zone pointée
  - `int * p;` n'alloue pas `sizeof(int)` bytes en mémoire pour `*p`
- Pour qu'un pointeur référence de la mémoire allouée, il faut soit
  - que l'emplacement référencé soit pré-alloué
  - allouer un espace dynamiquement

Pointeur nuls

- On peut créer des pointeurs nuls
  - En `C` : `NULL`
  - En `C++` : `nullptr`
- `NULL` est une macro : `#define NULL 0`
- `nullptr` est un immédiat
- Déférencer un pointeur nul a un comportement indéterminé
- Créer un pointeur n'alloue pas de mémoire vers la zone pointée
  1. `int * p;` n'alloue pas `sizeof(int)` bytes en mémoire pour `*p`
- Pour qu'un pointeur référence de la mémoire allouée, il faut soit
  - que l'emplacement référencé soit pré-alloué
  - allouer un espace dynamiquement

Pointeur nuls

- On peut créer des pointeurs nuls
  - En `C` : `NULL`
  - En `C++` : `nullptr`
- `NULL` est une macro : `#define NULL 0`
- `nullptr` est un immédiat
- Déférencer un pointeur nul a un comportement indéterminé
- Créer un pointeur n'alloue pas de mémoire vers la zone pointée
  1. `int * p;` n'alloue pas `sizeof(int)` bytes en mémoire pour `*p`
- Pour qu'un pointeur référence de la mémoire allouée, il faut soit
  - que l'emplacement référencé soit pré-alloué
  - allouer un espace dynamiquement

Pointeur nuls

- On peut créer des pointeurs nuls
  - En `C` : `NULL`
  - En `C++` : `nullptr`
- `NULL` est une macro : `#define NULL 0`
- `nullptr` est un immédiat
- Déférencer un pointeur nul a un comportement indéterminé
- Créer un pointeur n'alloue pas de mémoire vers la zone pointée
  1. `int * p;` n'alloue pas `sizeof(int)` bytes en mémoire pour `*p`
- Pour qu'un pointeur référence de la mémoire allouée, il faut soit
  - que l'emplacement référencé soit pré-alloué
  - allouer un espace dynamiquement

Pointeur nuls

- On peut créer des pointeurs nuls
  - En `C` : `NULL`
  - En `C++` : `nullptr`
- `NULL` est une macro : `#define NULL 0`
- `nullptr` est un immédiat
- Déférencer un pointeur nul a un comportement indéterminé
- Créer un pointeur n'alloue pas de mémoire vers la zone pointée
  1. `int * p;` n'alloue pas `sizeof(int)` bytes en mémoire pour `*p`
- Pour qu'un pointeur référence de la mémoire allouée, il faut soit
  - que l'emplacement référencé soit pré-alloué
  - allouer un espace dynamiquement

Pointeur nuls

- On peut créer des pointeurs nuls
  - En `C` : `NULL`
  - En `C++` : `nullptr`
- `NULL` est une macro : `#define NULL 0`
- `nullptr` est un immédiat
- Déférencer un pointeur nul a un comportement indéterminé
- Créer un pointeur n'alloue pas de mémoire vers la zone pointée
  1. `int * p;` n'alloue pas `sizeof(int)` bytes en mémoire pour `*p`
- Pour qu'un pointeur référence de la mémoire allouée, il faut soit
  - que l'emplacement référencé soit pré-alloué
  - allouer un espace dynamiquement

Conversions

- Conversions implicites de `T*` vers `void*`
- Aucune autre conversion implicite entre pointeurs n'est possible
- Pas de conversion de `int` vers `T*`, « sauf » avec `NULL`

Hygiène de programmation

- N'utilisez jamais `NULL` en C++

Conversions

- Conversions implicites de `T*` vers `void*`
- Aucune autre conversion implicite entre pointeurs n'est possible
- Pas de conversion de `int` vers `T*`, « sauf » avec `NULL`

Hygiène de programmation

- N'utilisez jamais `NULL` en C++

Conversions

- Conversions implicites de `T*` vers `void*`
- Aucune autre conversion implicite entre pointeurs n'est possible
- Pas de conversion de `int` vers `T*`, « sauf » avec `NULL`

Hygiène de programmation

- N'utilisez jamais `NULL` en C++

Conversions

- Conversions implicites de `T*` vers `void*`
- Aucune autre conversion implicite entre pointeurs n'est possible
- Pas de conversion de `int` vers `T*`, « sauf » avec `NULL`

Hygiène de programmation

- N'utilisez jamais `NULL` en `C++`
  - `nullptr` ne permet pas de conversions implicites

Conversions

- Conversions implicites de `T*` vers `void*`
- Aucune autre conversion implicite entre pointeurs n'est possible
- Pas de conversion de `int` vers `T*`, « sauf » avec `NULL`

Hygiène de programmation

- N'utilisez jamais `NULL` en `C++`
  - `nullptr` ne permet pas de conversions implicites

Conversions

- Conversions implicites de `T*` vers `void*`
- Aucune autre conversion implicite entre pointeurs n'est possible
- Pas de conversion de `int` vers `T*`, « sauf » avec `NULL`

Hygiène de programmation

- N'utilisez jamais `NULL` en `C++`
  - `nullptr` ne permet pas de conversions implicites

Exemple

- Fichier `ptr.c`

```
1  int main() {
2      int i = 3; int * pti = &i;
3
4      printf("i = %d, pti ( %p ) : %d\n", i, pti, *pti);
5      printf("Pointer address : %p of size %lu\n", &pti, sizeof(pti));
6
7      i++;
8      printf("i = %d, pti ( %p ) : %d\n", i, pti, *pti);
9
10     double d = 2.5; double * ptd = &d;
11
12     //pti = ptd; //Error
13     pti = (int*)ptd; //Ok, but bad idea
14     f(NULL); //Ok
15     *pti = *ptd;
16
17     int * ptn = NULL; //int * ptn = 0; //same stuff
18     int * ptinv1;
19     int * ptinv2 = 3; //Ok, but bad idea
20
21     printf("%d\n", *ptn);
22     printf("%d\n", *ptinv1); //bad idea
23 }
```

- Quelques différences en `C++` (`ptr.cpp`)

Syntaxe pointeurs et constantes

Pointeur constant de `double`

- `double d = 2;`
- `double * const pt = &d;`

Pointeur de `double` constant

- `const double d = 2;`
- `const double * pt = &d;`

Pointeur constant de `double` constant

- `const double d = 2;`
- `const double * const pt = &d;`

Syntaxe pointeurs et constantes

Pointeur constant de `double`

- `double d = 2;`
- `double * const pt = &d;`

Pointeur de `double` constant

- `const double d = 2;`
- `const double * pt = &d;`

Pointeur constant de `double` constant

- `const double d = 2;`
- `const double * const pt = &d;`

Syntaxe pointeurs et constantes

Pointeur constant de `double`

- `double d = 2;`
- `double * const pt = &d;`

Pointeur de `double` constant

- `const double d = 2;`
- `const double * pt = &d;`

Pointeur constant de `double` constant

- `const double d = 2;`
- `const double * const pt = &d;`

Syntaxe pointeurs et constantes

Pointeur constant de `double`

- `double d = 2;`
- `double * const pt = &d;`

Pointeur de `double` constant

- `const double d = 2;`
- `const double * pt = &d;`

Pointeur constant de `double` constant

- `const double d = 2;`
- `const double * const pt = &d;`

Syntaxe pointeurs et constantes

Pointeur constant de `double`

- `double d = 2;`
- `double * const pt = &d;`

Pointeur de `double` constant

- `const double d = 2;`
- `const double * pt = &d;`

Pointeur constant de `double` constant

- `const double d = 2;`
- `const double * const pt = &d;`

Syntaxe pointeurs et constantes

Pointeur constant de `double`

- `double d = 2;`
- `double * const pt = &d;`

Pointeur de `double` constant

- `const double d = 2;`
- `const double * pt = &d;`

Pointeur constant de `double` constant

- `const double d = 2;`
- `const double * const pt = &d;`

Syntaxe pointeurs et constantes

Pointeur constant de `double`

- `double d = 2;`
- `double * const pt = &d;`

Pointeur de `double` constant

- `const double d = 2;`
- `const double * pt = &d;`

Pointeur constant de `double` constant

- `const double d = 2;`
- `const double * const pt = &d;`

Syntaxe pointeurs et constantes

Pointeur constant de `double`

- `double d = 2;`
- `double * const pt = &d;`

Pointeur de `double` constant

- `const double d = 2;`
- `const double * pt = &d;`

Pointeur constant de `double` constant

- `const double d = 2;`
- `const double * const pt = &d;`

Syntaxe pointeurs et constantes

Pointeur constant de `double`

- `double d = 2;`
- `double * const pt = &d;`

Pointeur de `double` constant

- `const double d = 2;`
- `const double * pt = &d;`

Pointeur constant de `double` constant

- `const double d = 2;`
- `const double * const pt = &d;`

Illustration

- Fichier `ptr-cst.c`

```c
int main()
{
  int i = 2;
  const int ci = 3; //int const ci = 3 similaire
  i += 2;
  //ci += 2; //ko

  int * pi = &i;
  printf("%p : %d\n", pi, i);
  *pi = 3;
  printf("%p : %d\n", pi, i);

  int * const cpi = &i; //ptr constant
  *cpi = 5;
  //cpi++;

  const int * pic = &ci; //ptr d'entier constant
  //*pic = 4;
  pic++;

  const int * const cpic = &ci; //ptr cst d'entier cst
  //*cpic = 4;
  //cpic++;
}
```

Arithmétique

- On peut « déplacer » un pointeur
  - C'est une adresse : on se déplace en mémoire
- Si `pt` est un pointeur de type `T`, alors `t + k` déplace l'adresse de `k * sizeof(T)` bytes
  - `pt++` déplace `pt` de `sizeof(T)`

Arithmétique

- On peut « déplacer » un pointeur
  - C'est une adresse : on se déplace en mémoire
- Si `pt` est un pointeur de type `T`, alors `t + k` déplace l'adresse de `k * sizeof(T)` bytes
  - `pt++` déplace `pt` de `sizeof(T)`

Arithmétique

- On peut « déplacer » un pointeur
  - C'est une adresse : on se déplace en mémoire
- Si `pt` est un pointeur de type `T`, alors `t + k` déplace l'adresse de `k * sizeof(T)` bytes
  - `pt++` déplace `pt` de `sizeof(T)`

Arithmétique

- On peut « déplacer » un pointeur
  - C'est une adresse : on se déplace en mémoire
- Si `pt` est un pointeur de type `T`, alors `t + k` déplace l'adresse de `k * sizeof(T)` bytes
  - `pt++` déplace `pt` de `sizeof(T)`

Illustration

- Fichier `adv-ptr.c`

```c
#include <stdio.h>

void increment_and_print(int* ptr, unsigned count)
{
    for(unsigned i = 0; i < count; i++)
    {
        long long unsigned before = (long long)ptr;
        ptr++;
        printf("adress : %p shifted by %llu bytes", ptr, (long long)ptr - before);
        printf("because sizeof(int) is %zu bytes\n", sizeof(int));
    }
}

int main()
{
    int i = 0;
    int* ptr = &i;
    printf("adress before : %p\n", ptr);

    increment_and_print(ptr, 3); //DO NOT deference anymore
}
```

Application

- Fichier `print-str.c`

```
1  void print_str(const char* s)
2  {
3    const char* pt = s;
4    while(*pt != '\0')
5    {
6      printf("%c", *pt);
7      pt++;
8    }
9  }
10
11 int main()
12 {
13   const char* s = "Hello World!\n";
14   print_str(s);
15 }
```

Pointeurs vers des temporaires

Attention

- Ne retournez pas de pointeurs vers une variable locale

```
1  int * f()
2  {
3    int i = 2;
4    return &i;
5  }
6
7  int main()
8  {
9    printf("%d\n", *f()); //undefined behaviour
10 }
```

Conclusion

- Les pointeurs sont des adresses
- On peut construire un pointeur par affectation ou par prise d'adresse (`&` préfixé)
- On accède au contenu pointé par déférencement (`*` préfixé)
- On dispose d'une arithmétique de pointeur pour avancer en mémoire
- Un pointeur peut être nul

Hygiène de programmation

- N'utilisez *jamais* `NULL` en `C++`

Conclusion

- Les pointeurs sont des adresses
- On peut construire un pointeur par affectation ou par prise d'adresse (`&` préfixé)
- On accède au contenu pointé par déférencement (`*` préfixé)
- On dispose d'une arithmétique de pointeur pour avancer en mémoire
- Un pointeur peut être nul

Hygiène de programmation

- N'utilisez *jamais* `NULL` en C++

Conclusion

- Les pointeurs sont des adresses
- On peut construire un pointeur par affectation ou par prise d'adresse (`&` préfixé)
- On accède au contenu pointé par déférencement (`*` préfixé)
- On dispose d'une arithmétique de pointeur pour avancer en mémoire
- Un pointeur peut être nul

Hygiène de programmation

- N'utilisez *jamais* `NULL` en C++

Conclusion

- Les pointeurs sont des adresses
- On peut construire un pointeur par affectation ou par prise d'adresse (`&` préfixé)
- On accède au contenu pointé par déférencement (`*` préfixé)
- On dispose d'une arithmétique de pointeur pour avancer en mémoire
- Un pointeur peut être nul

Hygiène de programmation

- N'utilisez *jamais* `NULL` en C++

Conclusion

- Les pointeurs sont des adresses
- On peut construire un pointeur par affectation ou par prise d'adresse (`&` préfixé)
- On accède au contenu pointé par déférencement (`*` préfixé)
- On dispose d'une arithmétique de pointeur pour avancer en mémoire
- Un pointeur peut être nul

Hygiène de programmation

- N'utilisez *jamais* `NULL` en C++

Conclusion

- Les pointeurs sont des adresses
- On peut construire un pointeur par affectation ou par prise d'adresse (`&` préfixé)
- On accède au contenu pointé par déférencement (`*` préfixé)
- On dispose d'une arithmétique de pointeur pour avancer en mémoire
- Un pointeur peut être nul

Hygiène de programmation

- N'utilisez *jamais* `NULL` en `C++`

Conclusion

- Les pointeurs sont des adresses
- On peut construire un pointeur par affectation ou par prise d'adresse (`&` préfixé)
- On accède au contenu pointé par déférencement (`*` préfixé)
- On dispose d'une arithmétique de pointeur pour avancer en mémoire
- Un pointeur peut être nul

Hygiène de programmation

- N'utilisez *jamais* `NULL` en `C++`

Pointeurs et tableaux

Absence de conteneur

- En C, il n'existe pas de conteneurs standards
- Utilisation de tableaux « en dur », avec des pointeurs
- La taille du tableau doit pouvoir être déterminée à la compilation
  - Mention explicite
- Le type `type[]` est implicitement converti vers `type*` quand
  - ce n'est pas un opérande de `sizeof` et `&` (prise d'adresse)
  - ce n'est pas un littéral de chaîne de caractère
- En C, les tableaux « n'embarquant pas leur taille »
  - Il faut la spécifier quand on en a besoin
  - En C++, on utilise des conteneurs
- Les éléments « non spécifiés » sont initialisés à zéro
  - Si on spécifie plus d'éléments que la taille explicite : erreur

Absence de conteneur

- En C, il n'existe pas de conteneurs standards
- Utilisation de tableaux « en dur », avec des pointeurs
- La taille du tableau doit pouvoir être déterminée à la compilation
  - Mention explicite
- Le type `type[]` est implicitement converti vers `type*` quand
  - ce n'est pas un opérande de `sizeof` et `&` (prise d'adresse)
  - ce n'est pas un littéral de chaîne de caractère
- En C, les tableaux « n'embarquant pas leur taille »
  - Il faut la spécifier quand on en a besoin
  - En C++, on utilise des conteneurs
- Les éléments « non spécifiés » sont initialisés à zéro
  - Si on spécifie plus d'éléments que la taille explicite : erreur

Absence de conteneur

- En C, il n'existe pas de conteneurs standards
- Utilisation de tableaux « en dur », avec des pointeurs
- La taille du tableau doit pouvoir être déterminée à la compilation
  - Mention explicite
- Le type `type[]` est implicitement converti vers `type*` quand
  - ce n'est pas un opérande de `sizeof` et `&` (prise d'adresse)
  - ce n'est pas un littéral de chaîne de caractère
- En C, les tableaux « n'embarquant pas leur taille »
  - Il faut la spécifier quand on en a besoin
  - En C++, on utilise des conteneurs
- Les éléments « non spécifiés » sont initialisés à zéro
  - Si on spécifie plus d'éléments que la taille explicite : erreur

Absence de conteneur

- En C, il n'existe pas de conteneurs standards
- Utilisation de tableaux « en dur », avec des pointeurs
- La taille du tableau doit pouvoir être déterminée à la compilation
  - Mention explicite
- Le type `type[]` est implicitement converti vers `type*` quand
  - ce n'est pas un opérande de `sizeof` et `&` (prise d'adresse)
  - ce n'est pas un littéral de chaîne de caractère
- En C, les tableaux « n'embarquant pas leur taille »
  - Il faut la spécifier quand on en a besoin
  - En C++, on utilise des conteneurs
- Les éléments « non spécifiés » sont initialisés à zéro
  - Si on spécifie plus d'éléments que la taille explicite : erreur

Absence de conteneur

- En C, il n'existe pas de conteneurs standards
- Utilisation de tableaux « en dur », avec des pointeurs
- La taille du tableau doit pouvoir être déterminée à la compilation
  - Mention explicite
- Le type `type[]` est implicitement converti vers `type*` quand
  - ce n'est pas un opérande de `sizeof` et `&` (prise d'adresse)
  - ce n'est pas un littéral de chaîne de caractère
- En C, les tableaux « n'embarquant pas leur taille »
  - Il faut la spécifier quand on en a besoin
  - En C++, on utilise des conteneurs
- Les éléments « non spécifiés » sont initialisés à zéro
  - Si on spécifie plus d'éléments que la taille explicite : erreur

Absence de conteneur

- En C, il n'existe pas de conteneurs standards
- Utilisation de tableaux « en dur », avec des pointeurs
- La taille du tableau doit pouvoir être déterminée à la compilation
  - Mention explicite
- Le type `type[]` est implicitement converti vers `type*` quand
  - ce n'est pas un opérande de `sizeof` et `&` (prise d'adresse)
  - ce n'est pas un littéral de chaîne de caractère
- En C, les tableaux « n'embarquant pas leur taille »
  - Il faut la spécifier quand on en a besoin
  - En C++, on utilise des conteneurs
- Les éléments « non spécifiés » sont initialisés à zéro
  - Si on spécifie plus d'éléments que la taille explicite : erreur

Absence de conteneur

- En C, il n'existe pas de conteneurs standards
- Utilisation de tableaux « en dur », avec des pointeurs
- La taille du tableau doit pouvoir être déterminée à la compilation
  - Mention explicite
- Le type `type[]` est implicitement converti vers `type*` quand
  - ce n'est pas un opérande de `sizeof` et `&` (prise d'adresse)
  - ce n'est pas un littéral de chaîne de caractère
- En C, les tableaux « n'embarquant pas leur taille »
  - Il faut la spécifier quand on en a besoin
  - En C++, on utilise des conteneurs
- Les éléments « non spécifiés » sont initialisés à zéro
  - Si on spécifie plus d'éléments que la taille explicite : erreur

Absence de conteneur

- En `C`, il n'existe pas de conteneurs standards
- Utilisation de tableaux « en dur », avec des pointeurs
- La taille du tableau doit pouvoir être déterminée à la compilation
  - Mention explicite
- Le type `type[]` est implicitement converti vers `type*` quand
  - ce n'est pas un opérande de `sizeof` et `&` (prise d'adresse)
  - ce n'est pas un littéral de chaîne de caractère
- En `C`, les tableaux « n'embarquant pas leur taille »
  - Il faut la spécifier quand on en a besoin
  - En `C++`, on utilise des conteneurs
- Les éléments « non spécifiés » sont initialisés à zéro
  - Si on spécifie plus d'éléments que la taille explicite : erreur

Absence de conteneur

- En C, il n'existe pas de conteneurs standards
- Utilisation de tableaux « en dur », avec des pointeurs
- La taille du tableau doit pouvoir être déterminée à la compilation
  - Mention explicite
- Le type `type[]` est implicitement converti vers `type*` quand
  - ce n'est pas un opérande de `sizeof` et `&` (prise d'adresse)
  - ce n'est pas un littéral de chaîne de caractère
- En C, les tableaux « n'embarquant pas leur taille »
  - Il faut la spécifier quand on en a besoin
  - En C++, on utilise des conteneurs
- Les éléments « non spécifiés » sont initialisés à zéro
  - Si on spécifie plus d'éléments que la taille explicite : erreur

Absence de conteneur

- En `C`, il n'existe pas de conteneurs standards
- Utilisation de tableaux « en dur », avec des pointeurs
- La taille du tableau doit pouvoir être déterminée à la compilation
  - Mention explicite
- Le type `type[]` est implicitement converti vers `type*` quand
  - ce n'est pas un opérande de `sizeof` et `&` (prise d'adresse)
  - ce n'est pas un littéral de chaîne de caractère
- En `C`, les tableaux « n'embarquant pas leur taille »
  - Il faut la spécifier quand on en a besoin
  - En `C++`, on utilise des conteneurs
- Les éléments « non spécifiés » sont initialisés à zéro
  - Si on spécifie plus d'éléments que la taille explicite : erreur

Absence de conteneur

- En `C`, il n'existe pas de conteneurs standards
- Utilisation de tableaux « en dur », avec des pointeurs
- La taille du tableau doit pouvoir être déterminée à la compilation
  - Mention explicite
- Le type `type[]` est implicitement converti vers `type*` quand
  - ce n'est pas un opérande de `sizeof` et `&` (prise d'adresse)
  - ce n'est pas un littéral de chaîne de caractère
- En `C`, les tableaux « n'embarquant pas leur taille »
  - Il faut la spécifier quand on en a besoin
  - En `C++`, on utilise des conteneurs
- Les éléments « non spécifiés » sont initialisés à zéro
  - Si on spécifie plus d'éléments que la taille explicite : erreur

Absence de conteneur

- En C, il n'existe pas de conteneurs standards
- Utilisation de tableaux « en dur », avec des pointeurs
- La taille du tableau doit pouvoir être déterminée à la compilation
  - Mention explicite
- Le type `type[]` est implicitement converti vers `type*` quand
  - ce n'est pas un opérande de `sizeof` et `&` (prise d'adresse)
  - ce n'est pas un littéral de chaîne de caractère
- En C, les tableaux « n'embarquant pas leur taille »
  - Il faut la spécifier quand on en a besoin
  - En C++, on utilise des conteneurs
- Les éléments « non spécifiés » sont initialisés à zéro
  - Si on spécifie plus d'éléments que la taille explicite : erreur

Syntaxe : exemple

- Fichier `tab.c`

```
1  long unsigned sneaky(int array[])
2  {
3    return sizeof(array) / sizeof(*array);
4  }
5
6  int main()
7  {
8    int t1[5] = {1,2,3,4};
9    int t2[] = {1,2,3,4,5};
10   int * t3 = t2;
11   //int t4[];
12   //int t5[5] = {1,2,3,4,5,6};
13   int t6[8];
14
15   for(int i = 0; i < 5; i++)
16     printf("%d %d %d %d\n", t1[i], t2[i], t3[i], t6[i]);
17
18   printf("%lu\n", sizeof(t6) / sizeof(*t6));
19   printf("%lu\n", sneaky(t6));
20 }
```

Tableau en mémoire

- Les données d'un tableau sont allouées de manière contiguë en mémoire
- Même dans un tableau à plusieurs dimensions
- On peut y accéder à partir de l'adresse du premier élément
  - `int * t = {2,3,1,0,9,4,7,8};`
  - L'adresse du 4 (6e élément) est égal à `(t + 5)`

Tableau en mémoire

- Les données d'un tableau sont allouées de manière contiguë en mémoire
- Même dans un tableau à plusieurs dimensions
- On peut y accéder à partir de l'adresse du premier élément
  - `int * t = {2,3,1,0,9,4,7,8};`
  - L'adresse du 4 (6e élément) est égal à `(t + 5)`

Tableau en mémoire

- Les données d'un tableau sont allouées de manière contiguë en mémoire
- Même dans un tableau à plusieurs dimensions
- On peut y accéder à partir de l'adresse du premier élément
  - `int * t = {2,3,1,0,9,4,7,8};`
  - L'adresse du 4 (6e élément) est égal à `(t + 5)`

Tableau en mémoire

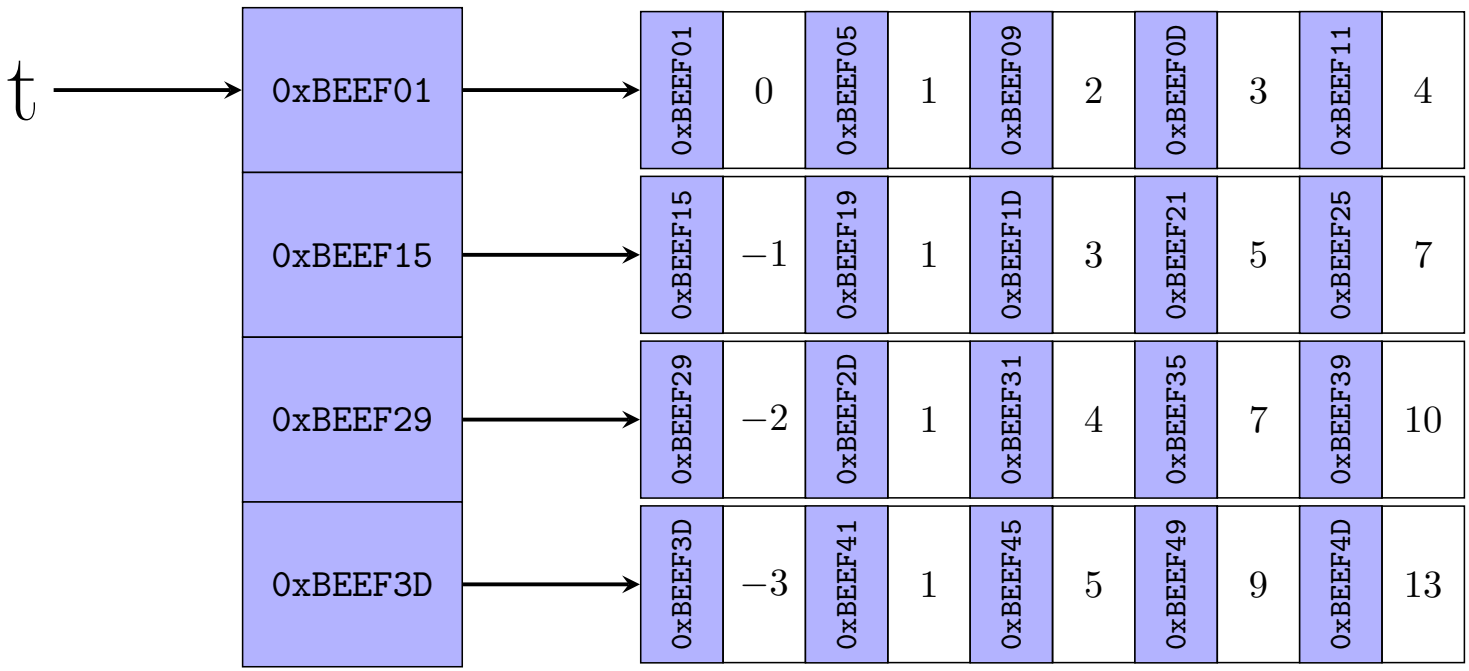
- Les données d'un tableau sont allouées de manière contiguë en mémoire
- Même dans un tableau à plusieurs dimensions
- On peut y accéder à partir de l'adresse du premier élément
  - `int * t = {2,3,1,0,9,4,7,8};`
  - L'adresse du 4 (6e élément) est égal à `(t + 5)`

Tableau en mémoire

- Les données d'un tableau sont allouées de manière contiguë en mémoire
- Même dans un tableau à plusieurs dimensions
- On peut y accéder à partir de l'adresse du premier élément
  - `int * t = {2,3,1,0,9,4,7,8};`
  - L'adresse du 4 (6e élément) est égal à `(t + 5)`

Illustration

```
int t[4][5]; //&t = 0xBEEF01
for(int i = 0; i < m; i++)
  for(int j = 0; j < n; j++)
    t[i][j] = i * j + (j - i);
```



Conclusion

- On peut modéliser des tableaux à l'aide de pointeurs

Hygiène de programmation

- On utilise cela uniquement en C pur

- On accéder aux éléments à l'aide de l'opérateur `[]`, ou à partir de l'adresse du premier élément
- La taille des tableaux doit être connue à la compilation

Conclusion

- On peut modéliser des tableaux à l'aide de pointeurs

Hygiène de programmation

- On utilise cela uniquement en C pur

- On accéder aux éléments à l'aide de l'opérateur `[]`, ou à partir de l'adresse du premier élément
- La taille des tableaux doit être connue à la compilation

Conclusion

- On peut modéliser des tableaux à l'aide de pointeurs

Hygiène de programmation

- On utilise cela uniquement en C pur
- On accéder aux éléments à l'aide de l'opérateur `[]`, ou à partir de l'adresse du premier élément
- La taille des tableaux doit être connue à la compilation

Conclusion

- On peut modéliser des tableaux à l'aide de pointeurs

Hygiène de programmation

- On utilise cela uniquement en C pur

- On accéder aux éléments à l'aide de l'opérateur `[]`, ou à partir de l'adresse du premier élément
- La taille des tableaux doit être connue à la compilation

Conclusion

- On peut modéliser des tableaux à l'aide de pointeurs

Hygiène de programmation

- On utilise cela uniquement en C pur

- On accéder aux éléments à l'aide de l'opérateur `[]`, ou à partir de l'adresse du premier élément
- La taille des tableaux doit être connue à la compilation

Pointeurs de fonctions

Adresse d'une fonction

- Les fonctions ont des adresses dans le segment `.text`
- On peut donc créer des pointeurs de fonction
- Utile pour passer une fonction en paramètre d'une autre fonction

Syntaxe

- `ReturnType (*Name)(parameters)`
- `void (*my_ptr)(int)` : `my_ptr` est un pointeur de fonction prenant en paramètre un `int` et retournant un `void`

- En `C++`, il existe divers wrappers

Adresse d'une fonction

- Les fonctions ont des adresses dans le segment `.text`
- On peut donc créer des pointeurs de fonction
- Utile pour passer une fonction en paramètre d'une autre fonction

Syntaxe

- `ReturnType (*Name)(parameters)`
- `void (*my_ptr)(int)` : `my_ptr` est un pointeur de fonction prenant en paramètre un `int` et retournant un `void`

- En C++, il existe divers wrappers

Adresse d'une fonction

- Les fonctions ont des adresses dans le segment `.text`
- On peut donc créer des pointeurs de fonction
- Utile pour passer une fonction en paramètre d'une autre fonction

Syntaxe

- `ReturnType (*Name)(parameters)`
- `void (*my_ptr)(int)` : `my_ptr` est un pointeur de fonction prenant en paramètre un `int` et retournant un `void`

- En C++, il existe divers wrappers

Adresse d'une fonction

- Les fonctions ont des adresses dans le segment `.text`
- On peut donc créer des pointeurs de fonction
- Utile pour passer une fonction en paramètre d'une autre fonction

Syntaxe

- `ReturnType (*Name)(parameters)`
- `void (*my_ptr)(int)` : `my_ptr` est un pointeur de fonction prenant en paramètre un `int` et retournant un `void`

- En `C++`, il existe divers wrappers

Adresse d'une fonction

- Les fonctions ont des adresses dans le segment `.text`
- On peut donc créer des pointeurs de fonction
- Utile pour passer une fonction en paramètre d'une autre fonction

Syntaxe

- `ReturnType (*Name)(parameters)`
- `void (*my_ptr)(int)` : `my_ptr` est un pointeur de fonction prenant en paramètre un `int` et retournant un `void`

- En `C++`, il existe divers wrappers

Adresse d'une fonction

- Les fonctions ont des adresses dans le segment `.text`
- On peut donc créer des pointeurs de fonction
- Utile pour passer une fonction en paramètre d'une autre fonction

Syntaxe

- `ReturnType (*Name)(parameters)`
- `void (*my_ptr)(int)` : `my_ptr` est un pointeur de fonction prenant en paramètre un `int` et retournant un `void`

- En C++, il existe divers wrappers

Adresse d'une fonction

- Les fonctions ont des adresses dans le segment `.text`
- On peut donc créer des pointeurs de fonction
- Utile pour passer une fonction en paramètre d'une autre fonction

Syntaxe

- `ReturnType (*Name)(parameters)`
- `void (*my_ptr)(int)` : `my_ptr` est un pointeur de fonction prenant en paramètre un `int` et retournant un `void`

- En C++, il existe divers wrappers

Exemple

- Fichier `fct-ptr.c`

```
1  void f(int a)
2  {
3    printf("a = %d\n", a);
4  }
5
6  int main()
7  {
8    void (*ptr)(int) = &f;
9    //void (*ptr)(int) = f; //similar
10
11   //Function call
12   (*ptr)(10);
13   //ptr(10);
14 }
```

Exemple d'utilisation

- Fichier `foreach.c`

```
1  void foreach_ro(int tab[], int size, void (*f)(int)) {
2    for(int i = 0; i < size; i++)
3      f(tab[i]);
4  }
5
6  void foreach_rw(int tab[], int size, int (*f)(int)) {
7    for(int i = 0; i < size; i++)
8      tab[i] = f(tab[i]);
9  }
10
11 void print(int i) {
12   printf("%d ", i);
13 }
14
15 int increment(int i) {
16   return i + 1;
17 }
18
19 int main() {
20   int tab[] = {1,2,3,4,5};
21
22   foreach_rw(tab, 5, increment);
23   foreach_ro(tab, 5, print);
24   printf("\n");
25 }
```

Remarques

- On ne peut pas allouer ou désallouer de la mémoire à l'aide d'un pointeur de fonction
  - Le segment `.text` est en lecture seule
- On peut créer des tableaux de pointeurs de fonction
  - `void (*my_fct_array[])(int)`
- Les pointeurs de fonctions permettent d'éviter la redondance de code
  - `qsort`
  - Comparateurs
- Si on imprime les bytes correspondant au pointeur de fonction, on obtient le langage machine
  - Très utilisé en cybersécurité

Remarques

- On ne peut pas allouer ou désallouer de la mémoire à l'aide d'un pointeur de fonction
  - Le segment `.text` est en lecture seule
- On peut créer des tableaux de pointeurs de fonction
  - `void (*my_fct_array[])(int)`
- Les pointeurs de fonctions permettent d'éviter la redondance de code
  - `qsort`
  - Comparateurs
- Si on imprime les bytes correspondant au pointeur de fonction, on obtient le langage machine
  - Très utilisé en cybersécurité

Remarques

- On ne peut pas allouer ou désallouer de la mémoire à l'aide d'un pointeur de fonction
  - Le segment `.text` est en lecture seule
- On peut créer des tableaux de pointeurs de fonction
  - `void (*my_fct_array[])(int)`
- Les pointeurs de fonctions permettent d'éviter la redondance de code
  - `qsort`
  - Comparateurs
- Si on imprime les bytes correspondant au pointeur de fonction, on obtient le langage machine
  - Très utilisé en cybersécurité

Remarques

- On ne peut pas allouer ou désallouer de la mémoire à l'aide d'un pointeur de fonction
  - Le segment `.text` est en lecture seule
- On peut créer des tableaux de pointeurs de fonction
  - `void (*my_fct_array[])(int)`
- Les pointeurs de fonctions permettent d'éviter la redondance de code
  - `qsort`
  - Comparateurs
- Si on imprime les bytes correspondant au pointeur de fonction, on obtient le langage machine
  - Très utilisé en cybersécurité

Remarques

- On ne peut pas allouer ou désallouer de la mémoire à l'aide d'un pointeur de fonction
  - Le segment `.text` est en lecture seule
- On peut créer des tableaux de pointeurs de fonction
  - `void (*my_fct_array[])(int)`
- Les pointeurs de fonctions permettent d'éviter la redondance de code
  - `qsort`
  - Comparateurs
- Si on imprime les bytes correspondant au pointeur de fonction, on obtient le langage machine
  - Très utilisé en cybersécurité

Remarques

- On ne peut pas allouer ou désallouer de la mémoire à l'aide d'un pointeur de fonction
  - Le segment `.text` est en lecture seule
- On peut créer des tableaux de pointeurs de fonction
  - `void (*my_fct_array[])(int)`
- Les pointeurs de fonctions permettent d'éviter la redondance de code
  - `qsort`
  - Comparateurs
- Si on imprime les bytes correspondant au pointeur de fonction, on obtient le langage machine
  - Très utilisé en cybersécurité

Remarques

- On ne peut pas allouer ou désallouer de la mémoire à l'aide d'un pointeur de fonction
  - Le segment `.text` est en lecture seule
- On peut créer des tableaux de pointeurs de fonction
  - `void (*my_fct_array[])(int)`
- Les pointeurs de fonctions permettent d'éviter la redondance de code
  - `qsort`
  - Comparateurs
- Si on imprime les bytes correspondant au pointeur de fonction, on obtient le langage machine
  - Très utilisé en cybersécurité

Remarques

- On ne peut pas allouer ou désallouer de la mémoire à l'aide d'un pointeur de fonction
  - Le segment `.text` est en lecture seule
- On peut créer des tableaux de pointeurs de fonction
  - `void (*my_fct_array[])(int)`
- Les pointeurs de fonctions permettent d'éviter la redondance de code
  - `qsort`
  - Comparateurs
- Si on imprime les bytes correspondant au pointeur de fonction, on obtient le langage machine
  - Très utilisé en cybersécurité

Remarques

- On ne peut pas allouer ou désallouer de la mémoire à l'aide d'un pointeur de fonction
  - Le segment `.text` est en lecture seule
- On peut créer des tableaux de pointeurs de fonction
  - `void (*my_fct_array[])(int)`
- Les pointeurs de fonctions permettent d'éviter la redondance de code
  - `qsort`
  - Comparateurs
- Si on imprime les bytes correspondant au pointeur de fonction, on obtient le langage machine
  - Très utilisé en cybersécurité

Exemple

- Fichier `machine.c`

```c
#include <string.h>

//code assembleur pour l'appel système execve sur /bin/sh
const char assembly[] =
  "\x31\xc0\x50\x68//sh\x68/bin"
  "\x89\xe3\x50\x53\x89\xe1\x99"
  "\xb0\x0b\xcd\x80";

int main()
{
  char buffer[sizeof(assembly)];
  strcpy(buffer, assembly);

  void (*f)() = (void (*)())buffer; //living the dream
  f();
}
```

- Il faut compiler comme `gcc -z execstack -o machine machine.c` sur une machine non protégée