Introduction

- De façon schématique, la mémoire d'un ordinateur s'apparente à un immense tableau dont chaque case a une adresse.

Introduction

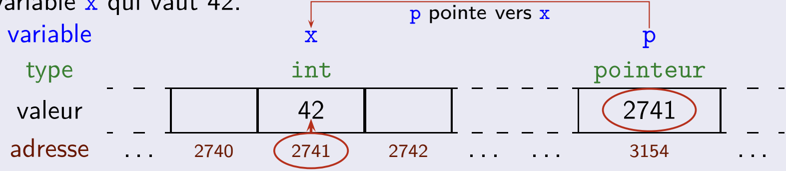
- De façon schématique, la mémoire d'un ordinateur s'apparente à un immense tableau dont chaque case a une adresse.
- Un pointeur est une variable contenant une de ces adresses.

Introduction

- De façon schématique, la mémoire d'un ordinateur s'apparente à un immense tableau dont chaque case a une adresse.
- Un pointeur est une variable contenant une de ces adresses.
- Le schéma ci-dessous représente un pointeur `p` contenant l'adresse d'une variable `x` qui vaut 42.

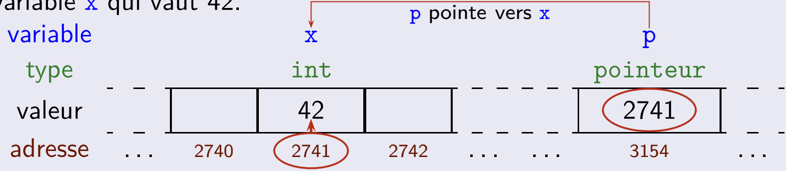
Introduction

- De façon schématique, la mémoire d'un ordinateur s'apparente à un immense tableau dont chaque case a une adresse.
- Un pointeur est une variable contenant une de ces adresses.
- Le schéma ci-dessous représente un pointeur `p` contenant l'adresse d'une variable `x` qui vaut 42.



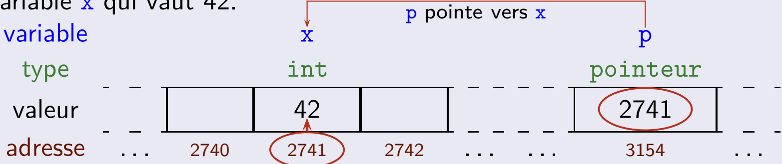
Introduction

- De façon schématique, la mémoire d'un ordinateur s'apparente à un immense tableau dont chaque case a une adresse.
- Un pointeur est une variable contenant une de ces adresses.
- Le schéma ci-dessous représente un pointeur `p` contenant l'adresse d'une variable `x` qui vaut 42.



Introduction

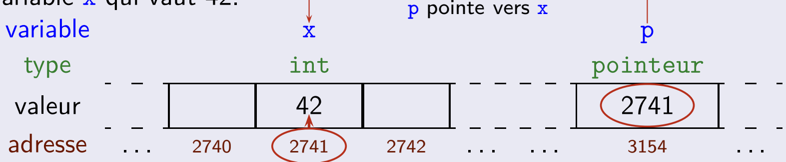
- De façon schématique, la mémoire d'un ordinateur s'apparente à un immense tableau dont chaque case a une adresse.
- Un pointeur est une variable contenant une de ces adresses.
- Le schéma ci-dessous représente un pointeur `p` contenant l'adresse d'une variable `x` qui vaut 42.



- En C, on peut directement manipuler des adresses mémoires (des pointeurs) :

Introduction

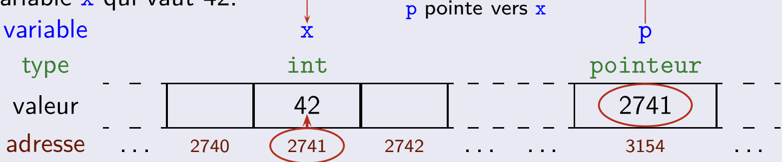
- De façon schématique, la mémoire d'un ordinateur s'apparente à un immense tableau dont chaque case a une adresse.
- Un pointeur est une variable contenant une de ces adresses.
- Le schéma ci-dessous représente un pointeur `p` contenant l'adresse d'une variable `x` qui vaut 42.



- En C, on peut directement manipuler des adresses mémoires (des pointeurs) :
  - `&x` est l'adresse de la variable `x` (dans l'exemple précédent `&x = p (=2741)`), le type de `p` est `int *` c'est à dire *adresse d'un entier*.

Introduction

- De façon schématique, la mémoire d'un ordinateur s'apparente à un immense tableau dont chaque case a une adresse.
- Un pointeur est une variable contenant une de ces adresses.
- Le schéma ci-dessous représente un pointeur `p` contenant l'adresse d'une variable `x` qui vaut 42.



- En C, on peut directement manipuler des adresses mémoires (des pointeurs) :
  - `&x` est l'adresse de la variable `x` (dans l'exemple précédent `&x = p (=2741)`), le type de `p` est `int *` c'est à dire *adresse d'un entier*.
  - `*p` est le contenu de l'adresse `p` (dans l'exemple précédent `*p = x (=42)`), le type de `*p` est `int`.

Schéma de l'organisation de la mémoire en C

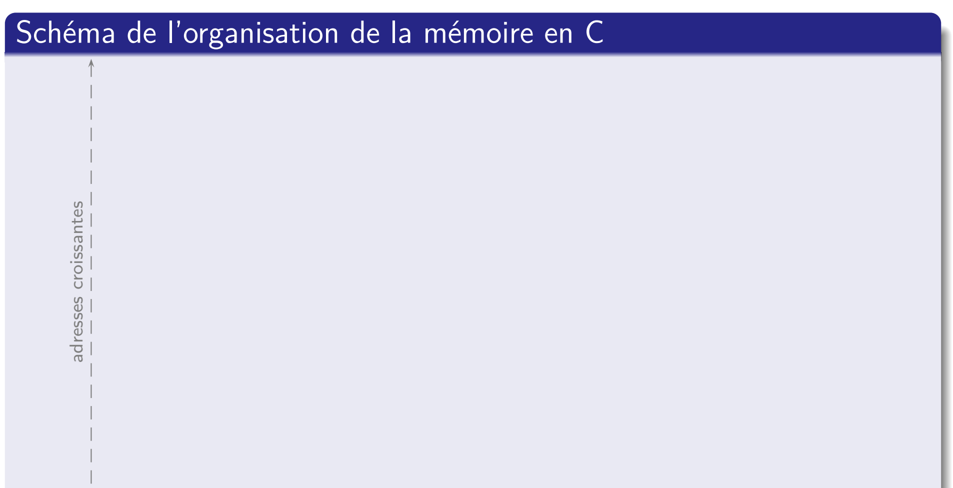


Schéma de l'organisation de la mémoire en C
adresses croissantes

Schéma de l'organisation de la mémoire en C

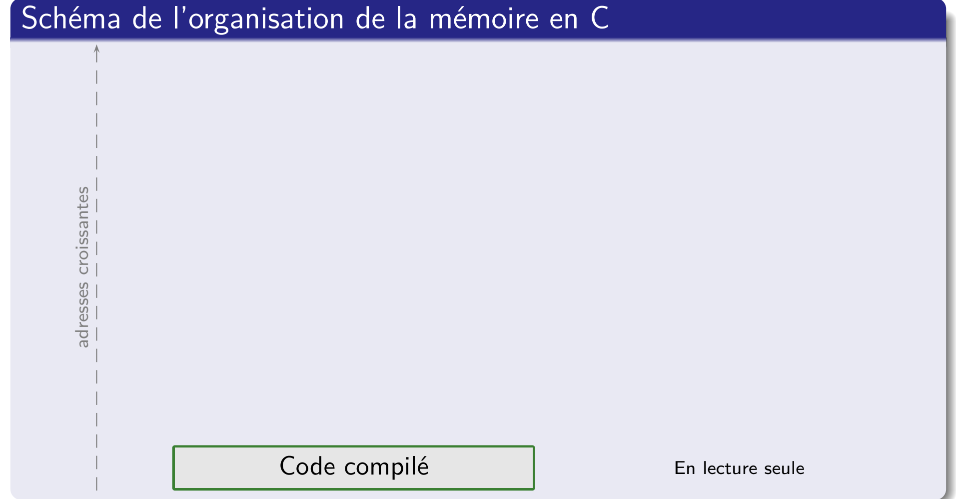


Schéma de l'organisation de la mémoire en C

adresses croissantes

Données statiques	Taille connue à la compilation initialisées ou non.
Code compilé	En lecture seule

Schéma de l'organisation de la mémoire en C

adresses croissantes

Zone	Description
Tas (*heap*)	Taille variable Allouée dynamiquement par le programmeur avec `malloc` et libéré avec `free`
Données statiques	Taille connue à la compilation initialisées ou non.
Code compilé	En lecture seule

Schéma de l'organisation de la mémoire en C

Schéma de l'organisation de la mémoire en C

adresses croissantes

Zone	Description
Pile (*stack*)	Empile les contextes d'appel des fonctions (variables locales, adresse de retour, ...)
Mémoire libre	
Tas (*heap*)	Taille variable Allouée dynamiquement par le programmeur avec `malloc` et libéré avec `free`
Données statiques	Taille connue à la compilation initialisées ou non.
Code compilé	En lecture seule

Schéma de l'organisation de la mémoire en C

adresses croissantes

Zone	Description
Pile (*stack*)	Empile les contextes d'appel des fonctions (variables locales, adresse de retour, ...)
Mémoire libre	
Tas (*heap*)	Taille variable Allouée dynamiquement par le programmeur avec `malloc` et libéré avec `free`
Données statiques	Taille connue à la compilation initialisées ou non.
Code compilé	En lecture seule

Schéma de l'organisation de la mémoire en C

Zone	Catégorie	Description
Pile (*stack*)	dynamique	Empile les contextes d'appel des fonctions (variables locales, adresse de retour, ...)
Mémoire libre	dynamique	
Tas (*heap*)	dynamique	Taille variable Allouée dynamiquement par le programmeur avec `malloc` et libéré avec `free`
Données statiques	statique	Taille connue à la compilation initialisées ou non.
Code compilé	statique	En lecture seule

adresses croissantes ↑

Conséquences

Cette organisation de la mémoire a des conséquences importantes

- La taille de la pile est limitée (bien plus que celle du tas), donc une variable locale de taille importante risque de provoquer un débordement de pile (*stackoverflow*). Il est nettement préférable de l'allouer dans le tas.

Conséquences

Cette organisation de la mémoire a des conséquences importantes

- La taille de la pile est limitée (bien plus que celle du tas), donc une variable locale de taille importante risque de provoquer un débordement de pile (*stackoverflow*). Il est nettement préférable de l'allouer dans le tas.
- Lors de l'appel à une fonction, les variables locales (et autres informations) sont stockées dans la pile. A la fin de l'exécution, ces informations sont supprimées de la pile. Conserver des pointeurs vers des adresses de variables locales est donc problématique.

Conséquences

Cette organisation de la mémoire a des conséquences importantes

- La taille de la pile est limitée (bien plus que celle du tas), donc une variable locale de taille importante risque de provoquer un débordement de pile (*stackoverflow*). Il est nettement préférable de l'allouer dans le tas.
- Lors de l'appel à une fonction, les variables locales (et autres informations) sont stockées dans la pile. A la fin de l'exécution, ces informations sont supprimées de la pile. Conserver des pointeurs vers des adresses de variables locales est donc problématique.
- De la mémoire alloué par le programmeur dans le tas et non libérée est considérée comme non disponible, créant des fuites mémoires (*memory leak*).

Un premier exemple

```c
int main()
{
    double big_array[1500000];
    return 0;
}
```

1. Rappeler la taille d'un `double`, en déduire la taille du tableau `big_array`

Un premier exemple

```c
int main()
{
    double big_array[1500000];
    return 0;
}
```

1. Rappeler la taille d'un `double`, en déduire la taille du tableau `big_array`

2. Comment expliquer que programme a provoqué une `erreur de segmentation`, alors qu'il a été exécuté sur une machine possédant 8 Go de mémoire vive ?

Un premier exemple

```
1  int main()
2  {
3      double big_array[1500000];
4      return 0;
5  }
```

1. Rappeler la taille d'un `double`, en déduire la taille du tableau `big_array`

2. Comment expliquer que programme a provoqué une `erreur de segmentation`, alors qu'il a été exécuté sur une machine possédant 8 Go de mémoire vive ?

3. En déduire une information sur la taille de la pile d'appel sur cet ordinateur.

Un premier exemple

```c
int main()
{
    double big_array[1500000];
    return 0;
}
```

1. Rappeler la taille d'un double, en déduire la taille du tableau big_array
2. Comment expliquer que programme a provoqué une erreur de segmentation, alors qu'il a été exécuté sur une machine possédant 8 Go de mémoire vive ?
3. En déduire une information sur la taille de la pile d'appel sur cet ordinateur.
4. Comment résoudre ce problème ?

Un premier exemple

```c
int main()
{
    double big_array[1500000];
    return 0;
}
```

1. Rappeler la taille d'un `double`, en déduire la taille du tableau `big_array`
   Un double occupe 8 octets, donc ce tableau $8 \times 1,5 = 12$ Mo.
2. Comment expliquer que programme a provoqué une `erreur de segmentation`, alors qu'il a été exécuté sur une machine possédant 8 Go de mémoire vive ?
3. En déduire une information sur la taille de la pile d'appel sur cet ordinateur.
4. Comment résoudre ce problème ?

Un premier exemple

```c
int main()
{
    double big_array[1500000];
    return 0;
}
```

1. Rappeler la taille d'un `double`, en déduire la taille du tableau `big_array`
   Un double occupe 8 octets, donc ce tableau $8 \times 1,5 = 12$ Mo.
2. Comment expliquer que programme a provoqué une `erreur de segmentation`, alors qu'il a été exécuté sur une machine possédant 8 Go de mémoire vive ?
   La taille du tableau dépasse celle de la pile, or le tableau étant une variable locale est alloué dans la pile.
3. En déduire une information sur la taille de la pile d'appel sur cet ordinateur.
4. Comment résoudre ce problème ?

Un premier exemple

```c
int main()
{
    double big_array[1500000];
    return 0;
}
```

1. Rappeler la taille d'un `double`, en déduire la taille du tableau `big_array`
   Un double occupe 8 octets, donc ce tableau $8 \times 1,5 = 12$ Mo.
2. Comment expliquer que programme a provoqué une `erreur de segmentation`, alors qu'il a été exécuté sur une machine possédant 8 Go de mémoire vive ?
   La taille du tableau dépasse celle de la pile, or le tableau étant une variable locale est alloué dans la pile.
3. En déduire une information sur la taille de la pile d'appel sur cet ordinateur.
   La pile fait moins de 12Mo (sa taille est de l'ordre de 8Mo sur l'ordinateur utilisé)
4. Comment résoudre ce problème ?

Un premier exemple

```c
int main()
{
    double big_array[1500000];
    return 0;
}
```

1. Rappeler la taille d'un `double`, en déduire la taille du tableau `big_array`
   Un double occupe 8 octets, donc ce tableau $8 \times 1,5 = 12$ Mo.
2. Comment expliquer que programme a provoqué une `erreur de segmentation`, alors qu'il a été exécuté sur une machine possédant 8 Go de mémoire vive ?
   La taille du tableau dépasse celle de la pile, or le tableau étant une variable locale est alloué dans la pile.
3. En déduire une information sur la taille de la pile d'appel sur cet ordinateur.
   La pile fait moins de 12Mo (sa taille est de l'ordre de 8Mo sur l'ordinateur utilisé)
4. Comment résoudre ce problème ?
   La mémoire occupée par le tableau doit être alloué sur le tas.

Opérateurs & et *

- Si `t` est un type du C, par exemple (`int`, `char`, ...), alors `t *` est du type pointeur vers une valeur de type `t`. Par exemple :

Opérateurs & et *

- Si t est un type du C, par exemple (int, char, ...), alors t * est du type pointeur vers une valeur de type t. Par exemple :

```c
int *p; //déclare un pointeur p vers un entier
```

Opérateurs & et *

- Si `t` est un type du C, par exemple (`int`, `char`, ...), alors `t *` est du type pointeur vers une valeur de type `t`. Par exemple :

```
int *p; //déclare un pointeur p vers un entier
```

- Une valeur spéciale `NULL`, indique un pointeur qui ne pointe vers rien. Par exemple,

Opérateurs & et *

- Si `t` est un type du C, par exemple (`int`, `char`, ...), alors `t *` est du type pointeur vers une valeur de type `t`. Par exemple :
```c
int *p; //déclare un pointeur p vers un entier
```
- Une valeur spéciale `NULL`, indique un pointeur qui ne pointe vers rien. Par exemple,
```c
char *c = NULL; // pointeur vers caractère initialisé à NULL
```

Opérateurs & et *

- Si `t` est un type du C, par exemple (`int`, `char`, ...), alors `t *` est du type pointeur vers une valeur de type `t`. Par exemple :
```
int *p; //déclare un pointeur p vers un entier
```
- Une valeur spéciale `NULL`, indique un pointeur qui ne pointe vers rien. Par exemple,
```
char *c = NULL; // pointeur vers caractère initialisé à NULL
```
- L'opérateur unaire `&`, appelé opérateur d'adresse, permet en C de récupérer l'adresse mémoire d'une variable.

Opérateurs & et *

- Si `t` est un type du C, par exemple (`int`, `char`, ...), alors `t *` est du type pointeur vers une valeur de type `t`. Par exemple :
  ```c
  int *p; //déclare un pointeur p vers un entier
  ```
- Une valeur spéciale `NULL`, indique un pointeur qui ne pointe vers rien. Par exemple,
  ```c
  char *c = NULL; // pointeur vers caractère initialisé à NULL
  ```
- L'opérateur unaire `&`, appelé opérateur d'adresse, permet en C de récupérer l'adresse mémoire d'une variable.
- L'opérateur unaire `*`, appelé opérateur de déréférencement, permet en C de récupérer la valeur stockée dans une adresse mémoire.
  ! Déréférencer un pointeur `NULL` est un comportement indéfini.

Opérateurs & et *

- Si `t` est un type du C, par exemple (`int`, `char`, ...), alors `t *` est du type pointeur vers une valeur de type `t`. Par exemple :
  ```
  int *p; //déclare un pointeur p vers un entier
  ```
- Une valeur spéciale `NULL`, indique un pointeur qui ne pointe vers rien. Par exemple,
  ```
  char *c = NULL; // pointeur vers caractère initialisé à NULL
  ```
- L'opérateur unaire `&`, appelé opérateur d'adresse, permet en C de récupérer l'adresse mémoire d'une variable.
- L'opérateur unaire `*`, appelé opérateur de déréférencement, permet en C de récupérer la valeur stockée dans une adresse mémoire.
  ❗ Déréférencer un pointeur `NULL` est un comportement indéfini.
- Pour résumer :

	S'applique à	Permet de
`&`	à une variable	récupérer son adresse
`*`	à un pointeur	récupérer la valeur à l'emplacement mémoire désigné

Exemple 1

```
1  int main()
2  {
3      int n = 42;
4      int *p = &n;
5      *p = 2024; // On modifie n (via le contenu de son adresse)
6      printf("Valeur de n = %d\n",n); // affiche 2024
7  }
```

Exemple 2

Exemple 1

```c
int main()
{
    int n = 42;
    int *p = &n;
    *p = 2024; // On modifie n (via le contenu de son adresse)
    printf("Valeur de n = %d\n",n); // affiche 2024
}
```

Dans l'exemple ci-dessus, `p` est un pointeur qui contient l'adresse de `n`.

Exemple 2

Exemple 1

```c
int main()
{
    int n = 42;
    int *p = &n;
    *p = 2024; // On modifie n (via le contenu de son adresse)
    printf("Valeur de n = %d\n",n); // affiche 2024
}
```

Dans l'exemple ci-dessus, `p` est un pointeur qui contient l'adresse de `n`. On modifie `n`, en modifiant le contenu de son adresse.

Exemple 2

Exemple 1

```
1  int main()
2  {
3      int n = 42;
4      int *p = &n;
5      *p = 2024; // On modifie n (via le contenu de son adresse)
6      printf("Valeur de n = %d\n",n); // affiche 2024
7  }
```

Dans l'exemple ci-dessus, `p` est un pointeur qui contient l'adresse de `n`. On modifie `n`, en modifiant le contenu de son adresse.

Exemple 2

1. Ecrire une fonction echange qui prend en argument deux adresses vers des entiers et échange les valeurs de ces deux entiers.

Exemple 1

```
1  int main()
2  {
3      int n = 42;
4      int *p = &n;
5      *p = 2024; // On modifie n (via le contenu de son adresse)
6      printf("Valeur de n = %d\n",n); // affiche 2024
7  }
```

Dans l'exemple ci-dessus, `p` est un pointeur qui contient l'adresse de `n`. On modifie `n`, en modifiant le contenu de son adresse.

Exemple 2

1. Ecrire une fonction `echange` qui prend en argument deux adresses vers des entiers et échange les valeurs de ces deux entiers.
2. Comment utiliser cette fonction pour échanger les valeurs de deux entiers `a` et `b` ?

Correction de l'exemple

Echanger les valeurs stockées aux adresses `p1` et `p2`.

Correction de l'exemple

Echanger les valeurs stockées aux adresses `p1` et `p2`.

Correction de l'exemple

Echanger les valeurs stockées aux adresses `p1` et `p2`.

Correction de l'exemple

Echanger les valeurs stockées aux adresses `p1` et `p2`.
Pour l'appel, récupérer les adresses de `a` et `b` afin de les passer en paramètres.

Correction de l'exemple

Echanger les valeurs stockées aux adresses p1 et p2.
Pour l'appel, récupérer les adresses de a et b afin de les passer en paramètres.

```c
void echange(int *p1, int *p2)
{
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
int main()
{
    int a = 42;
    int b = 2024;
    echange(&a, &b);
}
```

malloc

- La fonction `malloc` (disponible, comme `free`, après `#include <stdlib.h>`) permet d'allouer sur le tas, un bloc mémoire dont on donne la taille

Exemples

`malloc`

- La fonction `malloc` (disponible, comme `free`, après `#include <stdlib.h>`) permet d'allouer sur le tas, un bloc mémoire dont on donne la taille
- Elle s'utilise donc souvent conjointement à `sizeof` qui donne la taille d'un objet en C.

Exemples

malloc

- La fonction `malloc` (disponible, comme `free`, après `#include <stdlib.h>`) permet d'allouer sur le tas, un bloc mémoire dont on donne la taille
- Elle s'utilise donc souvent conjointement à `sizeof` qui donne la taille d'un objet en C.
- Comme pour les tableaux, accéder en dehors des limites du bloc alloué est un comportement indéfini.

Exemples

malloc

- La fonction `malloc` (disponible, comme `free`, après `#include <stdlib.h>`) permet d'allouer sur le tas, un bloc mémoire dont on donne la taille
- Elle s'utilise donc souvent conjointement à `sizeof` qui donne la taille d'un objet en C.
- Comme pour les tableaux, accéder en dehors des limites du bloc alloué est un comportement indéfini.

Exemples

- `int *t =  malloc(sizeof(int)*100); //alloue le bloc mémoire`

malloc

- La fonction `malloc` (disponible, comme `free`, après `#include <stdlib.h>`) permet d'allouer sur le tas, un bloc mémoire dont on donne la taille
- Elle s'utilise donc souvent conjointement à `sizeof` qui donne la taille d'un objet en C.
- Comme pour les tableaux, accéder en dehors des limites du bloc alloué est un comportement indéfini.

Exemples

- `int *t =  malloc(sizeof(int)*100); //alloue le bloc mémoire`
- `t[5] = 12; // affecte la valeur 12 au 6eme élément du bloc`

malloc

- La fonction `malloc` (disponible, comme `free`, après `#include <stdlib.h>`) permet d'allouer sur le tas, un bloc mémoire dont on donne la taille
- Elle s'utilise donc souvent conjointement à `sizeof` qui donne la taille d'un objet en C.
- Comme pour les tableaux, accéder en dehors des limites du bloc alloué est un comportement indéfini.

Exemples

- `int *t =  malloc(sizeof(int)*100); //alloue le bloc mémoire`
- `t[5] = 12; // affecte la valeur 12 au 6eme élément du bloc`
- `t[113] = 27; // Comportement indéfini`

free

- La fonction free permet de libérer un bloc mémoire alloué grâce à malloc

free

- La fonction free permet de libérer un bloc mémoire alloué grâce à malloc
- On appelle donc free sur un pointeur p crée par malloc.
  ⚠ Il est donc *impératif* de *toujours* disposer d'une référence vers un bloc mémoire qu'on a alloué. Sinon, on ne peut pas libérer ce bloc ce qui conduit à une fuite mémoire.

```c
void memory_leak()
{
    int *tab = malloc(sizeof(int)*100);
    tab[0] = 42;
    printf("On quitte sans libérer tab !\n");
}
int main()
{
    memory_leak();
    printf("Il devient ici impossible de libérer tab\n");
}
```

free

- La fonction `free` permet de libérer un bloc mémoire alloué grâce à `malloc`
- On appelle donc `free` sur un pointeur `p` crée par `malloc`.
  ⚠ Il est donc *impératif* de *toujours* disposer d'une référence vers un bloc mémoire qu'on a alloué. Sinon, on ne peut pas libérer ce bloc ce qui conduit à une fuite mémoire.

```c
void memory_leak()
{
    int *tab = malloc(sizeof(int)*100);
    tab[0] = 42;
    printf("On quitte sans libérer tab !\n");
}
int main()
{
    memory_leak();
    printf("Il devient ici impossible de libérer tab\n");
}
```

free

- La fonction `free` permet de libérer un bloc mémoire alloué grâce à `malloc`
- On appelle donc `free` sur un pointeur `p` crée par `malloc`.
  ⚠ Il est donc *impératif* de *toujours* disposer d'une référence vers un bloc mémoire qu'on a alloué. Sinon, on ne peut pas libérer ce bloc ce qui conduit à une fuite mémoire.

```
1  void memory_leak()
2  {
3      int *tab = malloc(sizeof(int)*100);
4      tab[0] = 42;
5      printf("On quitte sans libérer tab !\n");
6  }
7  int main()
8  {
9      memory_leak();
10     printf("Il devient ici impossible de libérer tab\n");
11 }
```

- On utilisera toujours l'option `fsanitize = adress` du compilateur pour détecter ces fuites mémoires.

Exemple : fonction renvoyant un « tableau »

Ecrire une fonction `int* make_tab(int size, int init)` qui renvoie un pointeur vers un bloc mémoire de `size` entiers tous initialisés à la valeur `init`.

Exemple : fonction renvoyant un « tableau »

Ecrire une fonction `int* make_tab(int size, int init)` qui renvoie un pointeur vers un bloc mémoire de `size` entiers tous initialisés à la valeur `init`.

⚠ L'allocation doit se faire avec `malloc`, sinon le bloc mémoire est alloué sur la pile, dans la zone de contexte d'appel de la fonction, cette zone est dépilé dès qu'on quitte la fonction et donc la zone mémoire du tableau n'est plus accessible.

Exemple : fonction renvoyant un « tableau »

Ecrire une fonction `int* make_tab(int size, int init)` qui renvoie un pointeur vers un bloc mémoire de `size` entiers tous initialisés à la valeur `init`.

- ⚠ L'allocation doit se faire avec `malloc`, sinon le bloc mémoire est alloué sur la pile, dans la zone de contexte d'appel de la fonction, cette zone est dépilé dès qu'on quitte la fonction et donc la zone mémoire du tableau n'est plus accessible.
- ⚠ La fonction appelante doit libérer la mémoire allouée avec `free` (sous peine de fuites mémoires.)

Correction

```c
// Fonction renvoyant un tableau d'entiers de
// taille size initialisé avec la valeur init
int *make_tab(int size, int init)
{
    int *tab = malloc(sizeof(int) * size);
    for (int i = 0; i < size; i++)
    {
        tab[i] = init;
    }
    return tab;
}

int main()
{
    int *test_tab = make_tab(100, 42);
    test_tab[0] = 34;
    free(test_tab);
}
```

Arguments de `main`

- La fonction `main` d'un programme C peut prendre en arguments un entier habituellement noté `argc` (*argument count*) et un tableau habituellement noté `argv[]` (*argument vector*) de chaines de caractères.

Arguments de `main`

- La fonction `main` d'un programme C peut prendre en arguments un entier habituellement noté `argc` (*argument count*) et un tableau habituellement noté `argv[]` (*argument vector*) de chaines de caractères.
- Ces arguments doivent alors être fournis à l'exécutable produit lors de la compilation.

Arguments de `main`

- La fonction `main` d'un programme C peut prendre en arguments un entier habituellement noté `argc` (*argument count*) et un tableau habituellement noté `argv[]` (*argument vector*) de chaines de caractères.
- Ces arguments doivent alors être fournis à l'exécutable produit lors de la compilation.
- Ces arguments sont traités comme des chaines de caractères et doivent donc être convertis dans le type adéquat si besoin grâce aux fonctions suivantes disponibles dans `stdlib`.

Arguments de `main`

- La fonction `main` d'un programme C peut prendre en arguments un entier habituellement noté `argc` (*argument count*) et un tableau habituellement noté `argv[]` (*argument vector*) de chaines de caractères.
- Ces arguments doivent alors être fournis à l'exécutable produit lors de la compilation.
- Ces arguments sont traités comme des chaines de caractères et doivent donc être convertis dans le type adéquat si besoin grâce aux fonctions suivantes disponibles dans `stdlib`.
  - La fonction `atoi` (ASCII *to integer*) permet de convertir une chaine de caractères en un `int`

Arguments de `main`

- La fonction `main` d'un programme C peut prendre en arguments un entier habituellement noté `argc` (*argument count*) et un tableau habituellement noté `argv[]` (*argument vector*) de chaines de caractères.
- Ces arguments doivent alors être fournis à l'exécutable produit lors de la compilation.
- Ces arguments sont traités comme des chaines de caractères et doivent donc être convertis dans le type adéquat si besoin grâce aux fonctions suivantes disponibles dans `stdlib`.
  - La fonction `atoi` (ASCII *to integer*) permet de convertir une chaine de caractères en un `int`
  - La fonction `atof` (ASCII *to float*) permet de convertir une chaine de caractères en un `double`

Exemple

Ecrire un exécutable `moyenne.exe` en C qui prend en argument sur la ligne de commande des flottants et écrit dans le terminal la moyenne de ces nombres. Par exemple `./moyenne.exe 12 10.5 16.5` doit écrire `13.0`.

Exemple

Ecrire un exécutable `moyenne.exe` en C qui prend en argument sur la ligne de commande des flottants et écrit dans le terminal la moyenne de ces nombres. Par exemple `./moyenne.exe 12 10.5 16.5` doit écrire `13.0`.

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

int main(int argc, char *argv[])
{
    double somme = 0.0;
    for (int i = 1; i < argc; i++)
    {
        somme = somme + atof(argv[i]);
    }
    printf("%f\n", somme / (argc - 1));
}
```

Définitions

- On peut définir en C, des types structurés, appelé `struct` composés de plusieurs champs.

Définitions

- On peut définir en C, des types structurés, appelé `struct` composés de plusieurs champs.
- La syntaxe générale de définition d'un type structuré est :

```c
struct nom_type_struct {
    type1 elt1;
    type2 elt2;
    ...};
```

Définitions

- On peut définir en C, des types structurés, appelé `struct` composés de plusieurs champs.
- La syntaxe générale de définition d'un type structuré est :

```
struct nom_type_struct {
    type1 elt1;
    type2 elt2;
    ...};
```

- Un nom de type (qui peut être celui du `struct`) peut être associé à un type structuré de façon à y faire référence plus rapidement.
  `typedef struct nom_type_struct nom_type`
  Le programme de MP2I recommande d'utiliser `nom_type_s` pour le nom du `struct` et `nom_type` pour le type.

Exemple

Pour créer le type structuré `personne_s` contenant les trois champs nom (chaine de caractères), poids et taille (float) :

```
struct personne_s {
    char nom[50]; //Le nom est limité à 50 caractères max
    poids float;
    taille float;
};
```

On peut donner un nom à ce type :

`typedef struct personne_s personne;`

Déclaration, lecture et écriture d'un champ

- La déclaration d'un variable de type personne peut se faire maintenant avec :
  `personne bruce_banner;`

Déclaration, lecture et écriture d'un champ

- La déclaration d'un variable de type personne peut se faire maintenant avec :
```
personne bruce_banner;
```
- Eventuellement avec initialisation immédiate avec la notation { et } déjà rencontré sur les tableaux :
```
personne hulk = {.nom="Hulk",.poids = 635,.taille=2.43};
```

Déclaration, lecture et écriture d'un champ

- La déclaration d'un variable de type personne peut se faire maintenant avec :
  `personne bruce_banner;`
- Eventuellement avec initialisation immédiate avec la notation { et } déjà rencontré sur les tableaux :
  `personne hulk = {.nom="Hulk",.poids = 635,.taille=2.43};`
- On accède aux champs avec la notion `.`, pour les lire comme par exemple :
  `imc_hulk = hulk.poids / (hulk.taille*hulk.taille);`

Déclaration, lecture et écriture d'un champ

- La déclaration d'un variable de type personne peut se faire maintenant avec :
  `personne bruce_banner;`
- Eventuellement avec initialisation immédiate avec la notation { et } déjà rencontré sur les tableaux :
  `personne hulk = {.nom="Hulk",.poids = 635,.taille=2.43};`
- On accède aux champs avec la notion `.`, pour les lire comme par exemple :
  `imc_hulk = hulk.poids / (hulk.taille*hulk.taille);`
- Ou pour les modifier, comme par exemple :
  `bruce_banner.taille = 1.75;`

Exemple : Hulk fait un régime

Quel sera l'affichage produit par le programme ci-dessous, pourquoi ?

```c
struct personne_s
{
    char nom[50];
    float taille;
    float poids;
};
typedef struct personne_s personne;

void change_poids(personne p, float modification)
{
    p.poids = p.poids + modification;
}

int main()
{
    personne hulk = {.nom = "Hulk", .poids = 650, .taille = 2.50};
    change_poids(hulk, -100.0);
    printf("Le poids de %s est  : %f\n", hulk.nom, hulk.poids);
    return 0;
}
```

Pointeur sur un `struct`

- L'utilisation de pointeurs sur des `struct` est courante en C.
  Lorsqu'on passe un pointeur vers un `struct` on passe une unique valeur (l'adresse), alors que si on passe le struct, on doit passer une copie de chacun de ses champs. Donc on peut passer par référence, même si on ne modifie pas le struct afin d'économiser l'espace mémoire et le temps de recopie des champs.

Exemple

Pointeur sur un `struct`

- L'utilisation de pointeurs sur des `struct` est courante en C.
  Lorsqu'on passe un pointeur vers un `struct` on passe une unique valeur (l'adresse), alors que si on passe le struct, on doit passer une copie de chacun de ses champs. Donc on peut passer par référence, même si on ne modifie pas le struct afin d'économiser l'espace mémoire et le temps de recopie des champs.
- Si `p` est un pointeur sur un `struct` (c'est-à-dire `*p` est un `struct`) alors on accède aux champs avec la notation : `(*p).nom_champ`

Exemple

Pointeur sur un `struct`

- L'utilisation de pointeurs sur des `struct` est courante en C.
  Lorsqu'on passe un pointeur vers un `struct` on passe une unique valeur (l'adresse), alors que si on passe le struct, on doit passer une copie de chacun de ses champs. Donc on peut passer par référence, même si on ne modifie pas le struct afin d'économiser l'espace mémoire et le temps de recopie des champs.
- Si `p` est un pointeur sur un `struct` (c'est-à-dire `*p` est un `struct`) alors on accède aux champs avec la notation : `(*p).nom_champ`
- La notation précédente est raccourcie en `p->nom_champ`.

Exemple

Pointeur sur un struct

- L'utilisation de pointeurs sur des `struct` est courante en C.
  Lorsqu'on passe un pointeur vers un `struct` on passe une unique valeur (l'adresse), alors que si on passe le struct, on doit passer une copie de chacun de ses champs. Donc on peut passer par référence, même si on ne modifie pas le struct afin d'économiser l'espace mémoire et le temps de recopie des champs.
- Si `p` est un pointeur sur un `struct` (c'est-à-dire `*p` est un `struct`) alors on accède aux champs avec la notation : `(*p).nom_champ`
- La notation précédente est raccourcie en `p->nom_champ`.

Exemple

Proposer une version correcte de la fonction modifiant le champ poids d'une variable de type `struct personne`.

Correction

```c
#include <stdio.h>

struct personne_struct{
    char nom[50];
    float taille;
    float poids;};
typedef  struct personne_struct personne;

void change_poids(personne *p, float modification){
    p->poids = p->poids + modification;}

int main(){
    personne hulk = {.nom = "Hulk",.poids=650,.taille=2.50};
    change_poids(&hulk,-100.0);
    printf("Le poids de %s est  : %f\n",hulk.nom,hulk.poids);
    return 0;}
```

Manipulation des fichiers en C

- L'ouverture d'un fichier se fait à l'aide de `fopen` qui prend comme arguments le nom du fichier et le mode d'ouverture :

Manipulation des fichiers en C

- L'ouverture d'un fichier se fait à l'aide de `fopen` qui prend comme arguments le nom du fichier et le mode d'ouverture :
  - `"r"` pour un accès en lecture

Manipulation des fichiers en C

- L'ouverture d'un fichier se fait à l'aide de `fopen` qui prend comme arguments le nom du fichier et le mode d'ouverture :
  - `"r"` pour un accès en lecture
  - `"w"` pour un accès en écriture (le fichier est détruit s'il existait)

Manipulation des fichiers en C

- L'ouverture d'un fichier se fait à l'aide de `fopen` qui prend comme arguments le nom du fichier et le mode d'ouverture :
  - `"r"` pour un accès en lecture
  - `"w"` pour un accès en écriture (le fichier est détruit s'il existait)
- Cette fonction renvoie un pointeur vers un objet de type `FILE` qui correspond à un flux de données sur lequel on peut lire ou écrire.

Manipulation des fichiers en C

- L'ouverture d'un fichier se fait à l'aide de `fopen` qui prend comme arguments le nom du fichier et le mode d'ouverture :
  - `"r"` pour un accès en lecture
  - `"w"` pour un accès en écriture (le fichier est détruit s'il existait)
- Cette fonction renvoie un pointeur vers un objet de type `FILE` qui correspond à un flux de données sur lequel on peut lire ou écrire.
- Les fonctions `fscanf` et `fprint` permet respectivement de lire et d'écrire sur le flux de données.

Manipulation des fichiers en C

- L'ouverture d'un fichier se fait à l'aide de `fopen` qui prend comme arguments le nom du fichier et le mode d'ouverture :
  - `"r"` pour un accès en lecture
  - `"w"` pour un accès en écriture (le fichier est détruit s'il existait)
- Cette fonction renvoie un pointeur vers un objet de type `FILE` qui correspond à un flux de données sur lequel on peut lire ou écrire.
- Les fonctions `fscanf` et `fprint` permet respectivement de lire et d'écrire sur le flux de données.
- Dans les deux cas, il faut fournir en argument le flux de données ainsi que les spécificateurs de format des données à lire et un pointeur vers les variables lues/écrites.

Manipulation des fichiers en C

- L'ouverture d'un fichier se fait à l'aide de `fopen` qui prend comme arguments le nom du fichier et le mode d'ouverture :
  - `"r"` pour un accès en lecture
  - `"w"` pour un accès en écriture (le fichier est détruit s'il existait)
- Cette fonction renvoie un pointeur vers un objet de type `FILE` qui correspond à un flux de données sur lequel on peut lire ou écrire.
- Les fonctions `fscanf` et `fprint` permet respectivement de lire et d'écrire sur le flux de données.
- Dans les deux cas, il faut fournir en argument le flux de données ainsi que les spécificateurs de format des données à lire et un pointeur vers les variables lues/écrites.
- La valeur spéciale `EOF` est renvoyée par `scanf` lorsque la fin du fichier est atteinte.

Manipulation des fichiers en C

- L'ouverture d'un fichier se fait à l'aide de `fopen` qui prend comme arguments le nom du fichier et le mode d'ouverture :
  - `"r"` pour un accès en lecture
  - `"w"` pour un accès en écriture (le fichier est détruit s'il existait)
- Cette fonction renvoie un pointeur vers un objet de type `FILE` qui correspond à un flux de données sur lequel on peut lire ou écrire.
- Les fonctions `fscanf` et `fprint` permet respectivement de lire et d'écrire sur le flux de données.
- Dans les deux cas, il faut fournir en argument le flux de données ainsi que les spécificateurs de format des données à lire et un pointeur vers les variables lues/écrites.
- La valeur spéciale `EOF` est renvoyée par `scanf` lorsque la fin du fichier est atteinte.
- Pour fermer un fichier, on utilise `fclose`.

Somme des entiers

On suppose qu'un fichier `entiers.txt` contient des entiers séparés par des espaces.

- Ecrire l'instruction permettant d'ouvrir ce fichier en mode lecture

Somme des entiers

On suppose qu'un fichier `entiers.txt` contient des entiers séparés par des espaces.

- Ecrire l'instruction permettant d'ouvrir ce fichier en mode lecture

Somme des entiers

On suppose qu'un fichier `entiers.txt` contient des entiers séparés par des espaces.

- Ecrire l'instruction permettant d'ouvrir ce fichier en mode lecture

Somme des entiers

On suppose qu'un fichier `entiers.txt` contient des entiers séparés par des espaces.

- Ecrire l'instruction permettant d'ouvrir ce fichier en mode lecture

Somme des entiers

On suppose qu'un fichier `entiers.txt` contient des entiers séparés par des espaces.

- Ecrire l'instruction permettant d'ouvrir ce fichier en mode lecture

```
1  FILE *fichier = fopen("entiers.txt","r");
```

- Déclarer deux entiers `n` (qui va contenir les entiers lus) et `somme` initialisé à 0.

Somme des entiers

On suppose qu'un fichier `entiers.txt` contient des entiers séparés par des espaces.

- Ecrire l'instruction permettant d'ouvrir ce fichier en mode lecture

```c
FILE *fichier = fopen("entiers.txt","r");
```

- Déclarer deux entiers `n` (qui va contenir les entiers lus) et `somme` initialisé à 0.

```c
int n, somme = 0;
```

- Ecrire une boucle while permettant de lire chacun des entiers jusqu'à la fin du fichier et d'en faire la somme dans la variable `somme`

Somme des entiers

On suppose qu'un fichier `entiers.txt` contient des entiers séparés par des espaces.

- Ecrire l'instruction permettant d'ouvrir ce fichier en mode lecture

```
FILE *fichier = fopen("entiers.txt","r");
```

- Déclarer deux entiers `n` (qui va contenir les entiers lus) et `somme` initialisé à 0.

```
int n, somme = 0;
```

- Ecrire une boucle while permettant de lire chacun des entiers jusqu'à la fin du fichier et d'en faire la somme dans la variable `somme`

```
while (fscanf(fichier,"%d",&n)!=EOF) {
    somme = somme + n;}
```

- Ecrire l'instruction permettant de fermer le fichier

Somme des entiers

On suppose qu'un fichier `entiers.txt` contient des entiers séparés par des espaces.

- Ecrire l'instruction permettant d'ouvrir ce fichier en mode lecture

```c
FILE *fichier = fopen("entiers.txt","r");
```

- Déclarer deux entiers `n` (qui va contenir les entiers lus) et `somme` initialisé à 0.

```c
int n, somme = 0;
```

- Ecrire une boucle while permettant de lire chacun des entiers jusqu'à la fin du fichier et d'en faire la somme dans la variable `somme`

```c
while (fscanf(fichier,"%d",&n)!=EOF) {
    somme = somme + n;}
```

- Ecrire l'instruction permettant de fermer le fichier

```c
fclose(fichier);
```

Programme complet

```c
#include <stdio.h>

int main(){
    FILE *fichier = fopen("entiers.txt","r");
    int n, somme = 0;
    while (fscanf(fichier,"%d",&n)!=EOF) {
        somme = somme + n;}
    fclose(fichier);
    printf("Somme = %d\n",somme);
    return 0;}
```

Principe

- De la même façon que certaines fonctions du langage C sont écrites dans des modules séparés et inclus à l'aide de la directive `#include` en début de programme, on peut écrire ses propres modules et y écrire des fonctions destinées à être réutilisées dans différents programmes.

Principe

- De la même façon que certaines fonctions du langage C sont écrites dans des modules séparés et inclus à l'aide de la directive `#include` en début de programme, on peut écrire ses propres modules et y écrire des fonctions destinées à être réutilisées dans différents programmes.
- Ces modules peuvent être compilés séparément, ce qui induit de nombreux avantages (réduction de la taille du programme principal, structuration de l'application, maintenance facilitée du programme, ...)

Méthode

Méthode

- On commence par simplement écrire les signatures des fonctions dans un fichier `<module.h>`. C'est le fichier d'en-tête (le `h` vient de l'anglais *header*).

Méthode

- On commence par simplement écrire les signatures des fonctions dans un fichier `<module.h>`. C'est le fichier d'en-tête (le `h` vient de l'anglais *header*).
- Les corps des fonctions sont écrites dans `<module.c>` et ce fichier est compilé de façon à obtenir un fichier objet `module.o` grâce à l'option `-c` de gcc.

Méthode

- On commence par simplement écrire les signatures des fonctions dans un fichier `<module.h>`. C'est le fichier d'en-tête (le `h` vient de l'anglais *header*).
- Les corps des fonctions sont écrites dans `<module.c>` et ce fichier est compilé de façon à obtenir un fichier objet `module.o` grâce à l'option `-c` de `gcc`.
- Dans le programme principal utilisant le module, on écrit au début `#include "module.h"` ce qui permet de faire référence à ces fonctions sans déclencher de *warning* à la compilation.

Méthode

- On commence par simplement écrire les signatures des fonctions dans un fichier `<module.h>`. C'est le fichier d'en-tête (le `h` vient de l'anglais *header*).
- Les corps des fonctions sont écrites dans `<module.c>` et ce fichier est compilé de façon à obtenir un fichier objet `module.o` grâce à l'option `-c` de `gcc`.
- Dans le programme principal utilisant le module, on écrit au début `#include "module.h"` ce qui permet de faire référence à ces fonctions sans déclencher de *warning* à la compilation.
  ⚠ Ce sont bien des guillemets et pas `<` et `>`.
- Le programme principal est lui aussi compilé avec l'option `-c` de `gcc` afin de produire un fichier `main.o`.

Méthode

- On commence par simplement écrire les signatures des fonctions dans un fichier `<module.h>`. C'est le fichier d'en-tête (le `h` vient de l'anglais *header*).
- Les corps des fonctions sont écrites dans `<module.c>` et ce fichier est compilé de façon à obtenir un fichier objet `module.o` grâce à l'option `-c` de `gcc`.
- Dans le programme principal utilisant le module, on écrit au début `#include "module.h"` ce qui permet de faire référence à ces fonctions sans déclencher de *warning* à la compilation.
  ⚠ Ce sont bien des guillemets et pas `<` et `>`.
- Le programme principal est lui aussi compilé avec l'option `-c` de `gcc` afin de produire un fichier `main.o`.
- Enfin, on lie ensemble les deux fichiers objets pour produire l'exécutable grâce à la ligne de compilation :

```
gcc module.o main.o -o main.exe
```

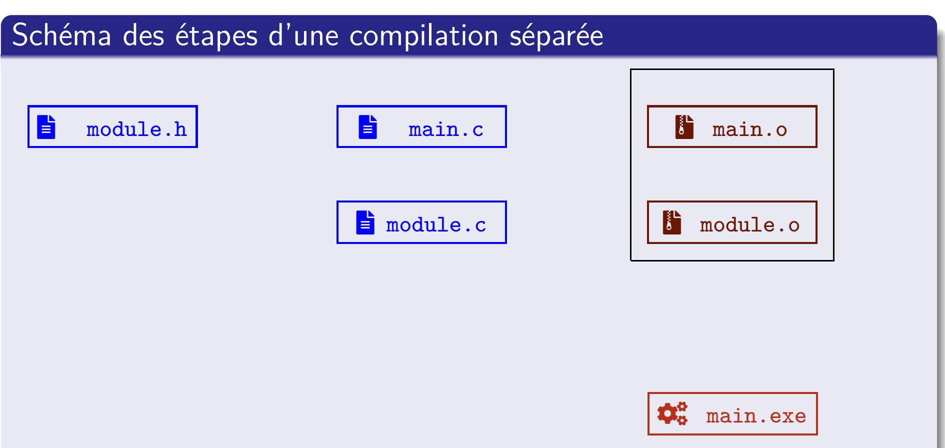


Schéma des étapes d'une compilation séparée
module.h
main.c
module.c
main.o
module.o
main.exe

Schéma des étapes d'une compilation séparée

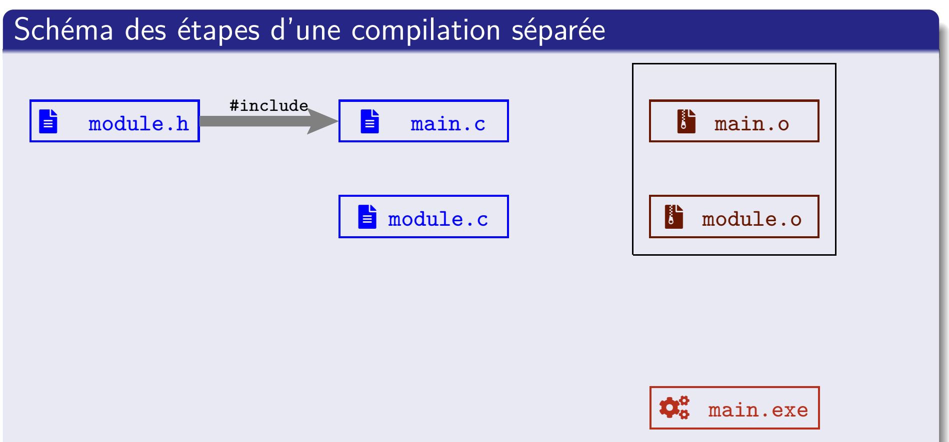


Schéma des étapes d'une compilation séparée

Schéma des étapes d'une compilation séparée

module.h —#include→ main.c —gcc -c→ main.o

module.c —gcc -c→ module.o

main.exe

Schéma des étapes d'une compilation séparée

Remarques

- Des types composés `struct` peuvent être définis dans le fichier d'en-tête sans donner leur structure. Ce point sera détaillé plus loin dans le cours lorsqu'on abordera les structures de données.

Remarques

- Des types composés `struct` peuvent être définis dans le fichier d'en-tête sans donner leur structure. Ce point sera détaillé plus loin dans le cours lorsqu'on abordera les structures de données.
- Afin d'éviter des redondances, on peut spécifier dans le fichier d'en-tête

```
#ifndef NONMODULE
#define NONMODULE
// ici définition des signatures des fonctions
// et ici des types structurés éventuels
#endif
```

Cela évite d'avoir des problèmes en cas de double inclusion malheureuse du même fichier.

Remarques

- Des types composés `struct` peuvent être définis dans le fichier d'en-tête sans donner leur structure. Ce point sera détaillé plus loin dans le cours lorsqu'on abordera les structures de données.
- Afin d'éviter des redondances, on peut spécifier dans le fichier d'en-tête

```
#ifndef NONMODULE
#define NONMODULE
// ici définition des signatures des fonctions
// et ici des types structurés éventuels
#endif
```

  Cela évite d'avoir des problèmes en cas de double inclusion malheureuse du même fichier.
- Afin d'éviter des conflits entre des fonctions portant le même nom et définies dans des modules ou dans le programme principale, on peut préfixer tous les noms de fonctions ou de type d'un module par le nom de ce module.

Un exemple

On décide de créer un module `entiers` afin d'y définir des fonctions usuelles sur les entiers non présentes en C. Par exemple le calcul de puissances ou encore le calcul du PGCD :

Un exemple

On décide de créer un module `entiers` afin d'y définir des fonctions usuelles sur les entiers non présentes en C. Par exemple le calcul de puissances ou encore le calcul du PGCD :

1. On commence par créer `entiers.h` et y écrire les signatures de nos fonctions

Un exemple

On décide de créer un module `entiers` afin d'y définir des fonctions usuelles sur les entiers non présentes en C. Par exemple le calcul de puissances ou encore le calcul du PGCD :

1. On commence par créer `entiers.h` et y écrire les signatures de nos fonctions On y trouvera par exemple
   ```
   int entiers_puissance(int a, int n);
   ```
2. On crée le fichier `entiers.c` qui contient ces fonctions et on le compile avec `gcc -c entiers.c`, cela produit un fichier `entiers.o`

Un exemple

On décide de créer un module `entiers` afin d'y définir des fonctions usuelles sur les entiers non présentes en C. Par exemple le calcul de puissances ou encore le calcul du PGCD :

1. On commence par créer `entiers.h` et y écrire les signatures de nos fonctions On y trouvera par exemple
   ```
   int entiers_puissance(int a, int n);
   ```
2. On crée le fichier `entiers.c` qui contient ces fonctions et on le compile avec `gcc -c entiers.c`, cela produit un fichier `entiers.o`
3. Dans le programme principal on inclus le fichier `entiers.h` avec :
   ```
   #include "entiers.h"
   ```

Un exemple

On décide de créer un module `entiers` afin d'y définir des fonctions usuelles sur les entiers non présentes en C. Par exemple le calcul de puissances ou encore le calcul du PGCD :

1. On commence par créer `entiers.h` et y écrire les signatures de nos fonctions On y trouvera par exemple
   ```
   int entiers_puissance(int a, int n);
   ```
2. On crée le fichier `entiers.c` qui contient ces fonctions et on le compile avec `gcc -c entiers.c`, cela produit un fichier `entiers.o`
3. Dans le programme principal on inclus le fichier `entiers.h` avec :
   ```
   #include "entiers.h"
   ```
4. On compile le programme principal avec `gcc -c main.c`, cela produit un fichier `main.o`

Un exemple

On décide de créer un module `entiers` afin d'y définir des fonctions usuelles sur les entiers non présentes en C. Par exemple le calcul de puissances ou encore le calcul du PGCD :

1. On commence par créer `entiers.h` et y écrire les signatures de nos fonctions On y trouvera par exemple
   ```
   int entiers_puissance(int a, int n);
   ```
2. On crée le fichier `entiers.c` qui contient ces fonctions et on le compile avec `gcc -c entiers.c`, cela produit un fichier `entiers.o`
3. Dans le programme principal on inclus le fichier `entiers.h` avec :
   ```
   #include "entiers.h"
   ```
4. On compile le programme principal avec `gcc -c main.c`, cela produit un fichier `main.o`
5. Enfin, on produit l'executable grâce à la compilation `gcc main.o entiers.o -o main.exe`

Allocation de l'espace mémoire

- En C, on peut allouer des tableaux multidimensionnels sur la pile en précisant les tailles attendues pour chaque dimension. Par exemple :

Allocation de l'espace mémoire

- En C, on peut allouer des tableaux multidimensionnels sur la pile en précisant les tailles attendues pour chaque dimension. Par exemple :
  ```
  float matrice[4][4];
  ```
  déclare un tableau de flottants de 4 lignes sur 4 colonnes. L'accès aux élément se fait alors avec la notation `matrice[i][j]`.

Allocation de l'espace mémoire

- En C, on peut allouer des tableaux multidimensionnels sur la pile en précisant les tailles attendues pour chaque dimension. Par exemple :

```
float matrice[4][4];
```

déclare un tableau de flottants de 4 lignes sur 4 colonnes. L'accès aux élément se fait alors avec la notation `matrice[i][j]`.
La taille de la pile étant limitée, cela est réservé aux tableaux de petites tailles.

Allocation de l'espace mémoire

- En C, on peut allouer des tableaux multidimensionnels sur la pile en précisant les tailles attendues pour chaque dimension. Par exemple :
  ```
  float matrice[4][4];
  ```
  déclare un tableau de flottants de 4 lignes sur 4 colonnes. L'accès aux élément se fait alors avec la notation `matrice[i][j]`.
  La taille de la pile étant limitée, cela est réservé aux tableaux de petites tailles.
- Pour une allocation sur le tas d'un tableau t, de n lignes et m colonnes, (qui sera nécessaire pour les tableaux de taille importante), trois options sont envisageables :

Allocation de l'espace mémoire

- En C, on peut allouer des tableaux multidimensionnels sur la pile en précisant les tailles attendues pour chaque dimension. Par exemple :
  ```
  float matrice[4][4];
  ```
  déclare un tableau de flottants de 4 lignes sur 4 colonnes. L'accès aux élément se fait alors avec la notation `matrice[i][j]`.
  La taille de la pile étant limitée, cela est réservé aux tableaux de petites tailles.
- Pour une allocation sur le tas d'un tableau t, de n lignes et m colonnes, (qui sera nécessaire pour les tableaux de taille importante), trois options sont envisageables :
  - V_0 Allouer un tableau à une dimension de taille $n \times m$.

Allocation de l'espace mémoire

- En C, on peut allouer des tableaux multidimensionnels sur la pile en précisant les tailles attendues pour chaque dimension. Par exemple :
  `float matrice[4][4];`
  déclare un tableau de flottants de 4 lignes sur 4 colonnes. L'accès aux élément se fait alors avec la notation `matrice[i][j]`.
  La taille de la pile étant limitée, cela est réservé aux tableaux de petites tailles.
- Pour une allocation sur le tas d'un tableau t, de n lignes et m colonnes, (qui sera nécessaire pour les tableaux de taille importante), trois options sont envisageables :
  - V_0 Allouer un tableau à une dimension de taille $n \times m$.
  - V_1 Allouer un tableau de tableau à une dimension.

Allocation de l'espace mémoire

- En C, on peut allouer des tableaux multidimensionnels sur la pile en précisant les tailles attendues pour chaque dimension. Par exemple :
  ```
  float matrice[4][4];
  ```
  déclare un tableau de flottants de 4 lignes sur 4 colonnes. L'accès aux élément se fait alors avec la notation `matrice[i][j]`.
  La taille de la pile étant limitée, cela est réservé aux tableaux de petites tailles.
- Pour une allocation sur le tas d'un tableau t, de n lignes et m colonnes, (qui sera nécessaire pour les tableaux de taille importante), trois options sont envisageables :
  - V_0 Allouer un tableau à une dimension de taille $n \times m$.
  - V_1 Allouer un tableau de tableau à une dimension.
  - V_2 Allouer un tableau de pointeurs, chacun des pointeurs pointe alors vers une ligne du tableau.

V_0 : Allocation d'un tableau à une dimension

- On alloue un tableau d'entiers de tailles $n \times m$ et on considère que les m premiers forment la premières ligne, que les m suivant sont la deuxième, ...

V_0 : Allocation d'un tableau à une dimension

- On alloue un tableau d'entiers de tailles $n \times m$ et on considère que les m premiers forment la premières ligne, que les m suivant sont la deuxième, ...
- La structure de données correspondante et "à visualiser" est alors la suivante :

$$\underbrace{\boxed{t(0,0) \mid \ldots \mid t(0,m-1)}}_{\text{ligne } 0 \ (m \text{ cases})} \ldots\ldots\ldots\ldots \underbrace{\boxed{t(n-1,0) \mid \ldots \mid t(n-1,m-1)}}_{\text{ligne } n-1 \ (m \text{ cases})}$$

V_0 : Allocation d'un tableau à une dimension

- On alloue un tableau d'entiers de tailles $n \times m$ et on considère que les m premiers forment la premières ligne, que les m suivant sont la deuxième, ...
- La structure de données correspondante et "à visualiser" est alors la suivante :

$$\underbrace{\boxed{t(0,0)}\boxed{\ldots}\boxed{t(0,m-1)}}_{\text{ligne } 0 \ (m \text{ cases})} \ldots\ldots\ldots\ldots \underbrace{\boxed{t(n-1,0)}\boxed{\ldots}\boxed{t(n-1,m-1)}}_{\text{ligne } n-1 \ (m \text{ cases})}$$

- Pour accéder à l'élément situé en colonne `j` et ligne `i` on doit donc faire `tab[m*i+j]`, c'est-à-dire qu'on perd la notation (pourtant pratique) `tab[i][j]` des tableaux à deux dimensions, c'est le seul inconvénient de cette méthode.

V_0 : Allocation d'un tableau à une dimension

- On alloue un tableau d'entiers de tailles $n \times m$ et on considère que les m premiers forment la premières ligne, que les m suivant sont la deuxième, ...
- La structure de données correspondante et "à visualiser" est alors la suivante :

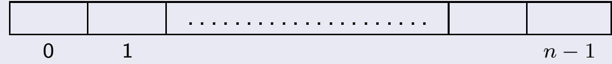
$t(0,0)$	...	$t(0,m-1)$	....	$t(n-1,0)$	...	$t(n-1,m-1)$

ligne 0 (m cases) ... ligne $n-1$ (m cases)

- Pour accéder à l'élément situé en colonne `j` et ligne `i` on doit donc faire `tab[m*i+j]`, c'est-à-dire qu'on perd la notation (pourtant pratique) `tab[i][j]` des tableaux à deux dimensions, c'est le seul inconvénient de cette méthode.
- Cette solution est la plus simple et sera celle à privilégier cette année. En effet, on ne fait qu'un `malloc` et un `free` et on manipule en fait un tableau à une dimension (ce qu'on doit normalement savoir faire !)

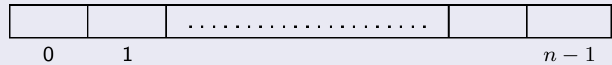
V_1 :Allocation d'un tableau de tableaux

- Un tableau d'entier de type `int` et de taille n à la structure suivante en mémoire (où chaque case à la taille prévue pour accueillir un `int`)



V_1 :Allocation d'un tableau de tableaux

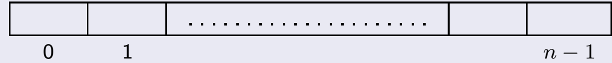
- Un tableau d'entier de type `int` et de taille n à la structure suivante en mémoire (où chaque case à la taille prévue pour accueillir un `int`)



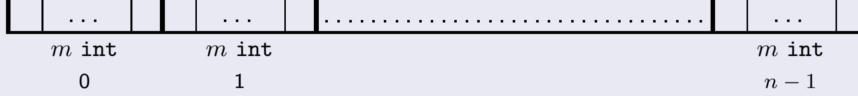
- Pour allouer une matrice de dimension $n \times m$, on peut "linéariser" le tableau et prévoir que chaque case à maintenant la capacité d'accueillir m entiers. *c'est-à-dire que c'est la solution précédente mais avec une gestion automatique des indices par le langage C.*

V_1 :Allocation d'un tableau de tableaux

- Un tableau d'entier de type `int` et de taille n à la structure suivante en mémoire (où chaque case à la taille prévue pour accueillir un `int`)



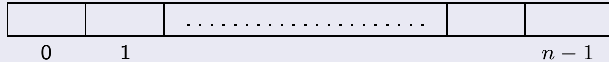
- Pour allouer une matrice de dimension $n \times m$, on peut "linéariser" le tableau et prévoir que chaque case à maintenant la capacité d'accueillir m entiers. *c'est-à-dire que c'est la solution précédente mais avec une gestion automatique des indices par le langage C.*
- La structure de données correspondante et "à visualiser" est alors la suivante :



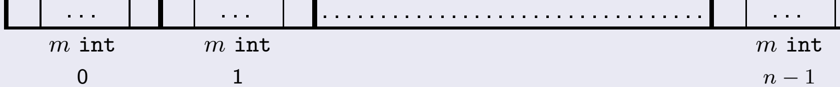
Chaque case du "grand" tableau contient un tableau de m entiers.

V_1 :Allocation d'un tableau de tableaux

- Un tableau d'entier de type `int` et de taille n à la structure suivante en mémoire (où chaque case à la taille prévue pour accueillir un `int`)



- Pour allouer une matrice de dimension $n \times m$, on peut "linéariser" le tableau et prévoir que chaque case à maintenant la capacité d'accueillir m entiers. *c'est-à-dire que c'est la solution précédente mais avec une gestion automatique des indices par le langage C.*
- La structure de données correspondante et "à visualiser" est alors la suivante :



Chaque case du "grand" tableau contient un tableau de m entiers.

- Cette allocation s'effectue avec :

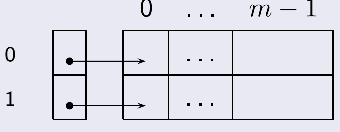
```
int (*mat)[n] =  malloc(sizeof(*mat)*m);
```

V_2 : Allocation d'un tableau de pointeurs

- On peut aussi, allouer un tableau de pointeurs, chaque pointeur pointe sur une ligne de la matrice. La structure de données correspondante est alors :

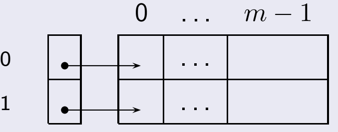
V_2 : Allocation d'un tableau de pointeurs

- On peut aussi, allouer un tableau de pointeurs, chaque pointeur pointe sur une ligne de la matrice. La structure de données correspondante est alors :



V_2 : Allocation d'un tableau de pointeurs

- On peut aussi, allouer un tableau de pointeurs, chaque pointeur pointe sur une ligne de la matrice. La structure de données correspondante est alors :

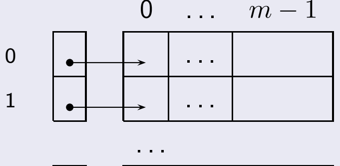


- Dans ce cas il faut allouer le tableau de pointeurs :

```c
int **mat = malloc(sizeof(int*)*n);
```

V_2 : Allocation d'un tableau de pointeurs

- On peut aussi, allouer un tableau de pointeurs, chaque pointeur pointe sur une ligne de la matrice. La structure de données correspondante est alors :



- Dans ce cas il faut allouer le tableau de pointeurs :
```
int **mat = malloc(sizeof(int*)*n);
```
- Puis chacune des lignes de la matrice individuellement :
`mat[i] = malloc(sizeof(int) * m);` (à faire pour $0 \leq i \leq n$)

Le bilan ...

- Chacune des méthodes a ses avantages et ses inconvénients

Le bilan ...

- Chacune des méthodes a ses avantages et ses inconvénients
- La seule qui doit être maitrisée est la méthode V_0 (puisqu'on reste en fait dans le cas de la dimension 1 en manipulant soi-même les indices)

Le bilan ...

- Chacune des méthodes a ses avantages et ses inconvénients
- La seule qui doit être maitrisée est la méthode V_0 (puisqu'on reste en fait dans le cas de la dimension 1 en manipulant soi-même les indices)
- Pour aller un peu plus loin dans l'apprentissage du C, les élèves intéressés peuvent essayer de coder une version utilisant la V_1 ou la V_2.