Fonctionnement bas niveau : programmation C

BUT Informatique S2

Arnaud GIERSCH

Département Informatique

IUT Nord Franche-Comté

Université Marie et Louis Pasteur

2025–2026

Thèmes abordés

Introduction

Organisation des séances

- Cours pendant les séances de TD
- Mise en pratique pendant les séances de TP

Organisation des séances

- Cours pendant les séances de TD
- Mise en pratique pendant les séances de TP
- Pré-requis
  - système Linux fortement conseillé
    (*éventuellement* macOS ; Windows avec WSL)
  - éditeur de textes
  - compilateur : gcc

```
$ sudo apt --install-suggests install gcc
```

Organisation des séances

- Cours pendant les séances de TD
- Mise en pratique pendant les séances de TP
- Pré-requis
  - système Linux fortement conseillé
    (*éventuellement* macOS ; Windows avec WSL)
  - éditeur de textes
  - compilateur : gcc

```
$ sudo apt --install-suggests install gcc
```

- Évaluation
  - examen final sur feuille

Séance 1 : le langage C

Le langage C

- Inventé vers 1972 par Brian Kernighan et Dennis Ritchie pour l'écriture du système Unix.

Le langage C

- Inventé vers 1972 par Brian Kernighan et Dennis Ritchie pour l'écriture du système Unix.
- Le langage C a ensuite été normalisé ($\sim$ 1989) et est depuis un des langages de programmation les plus utilisés.

Le langage C

- Inventé vers 1972 par Brian Kernighan et Dennis Ritchie pour l'écriture du système Unix.
- Le langage C a ensuite été normalisé ($\sim$ 1989) et est depuis un des langages de programmation les plus utilisés.
- C est un langage compilé, c.-à-d. que le programme ne peut pas être exécuté directement.

Le langage C

- Inventé vers 1972 par Brian Kernighan et Dennis Ritchie pour l'écriture du système Unix.
- Le langage C a ensuite été normalisé ($\sim$ 1989) et est depuis un des langages de programmation les plus utilisés.
- C est un langage compilé, c.-à-d. que le programme ne peut pas être exécuté directement.
- Il faut utiliser un compilateur pour traduire le programme en langage machine.

Compilation

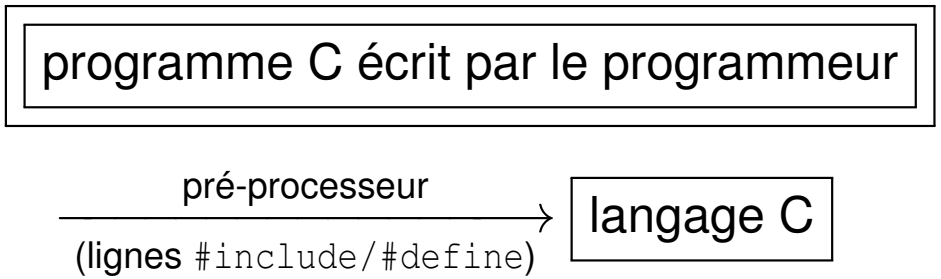
- Les étapes de compilation sont :

Compilation

- Les étapes de compilation sont :

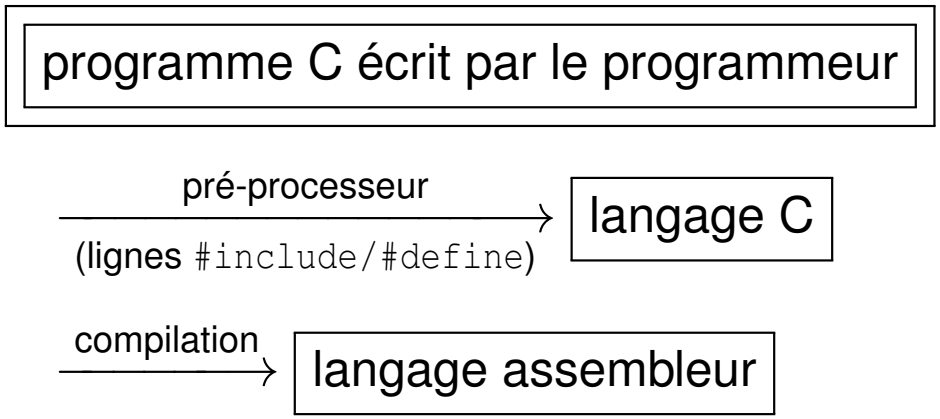
Compilation

- Les étapes de compilation sont :



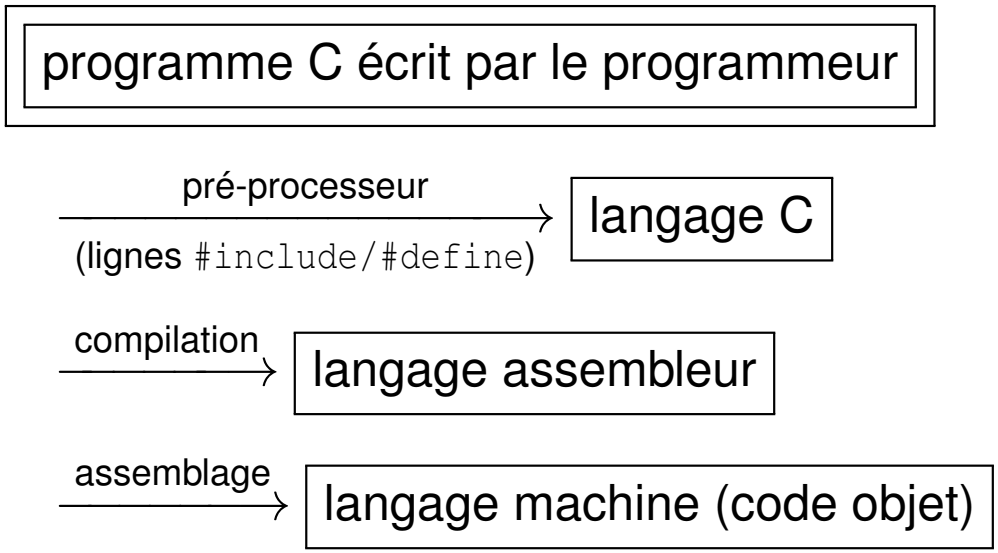
Compilation

- Les étapes de compilation sont :



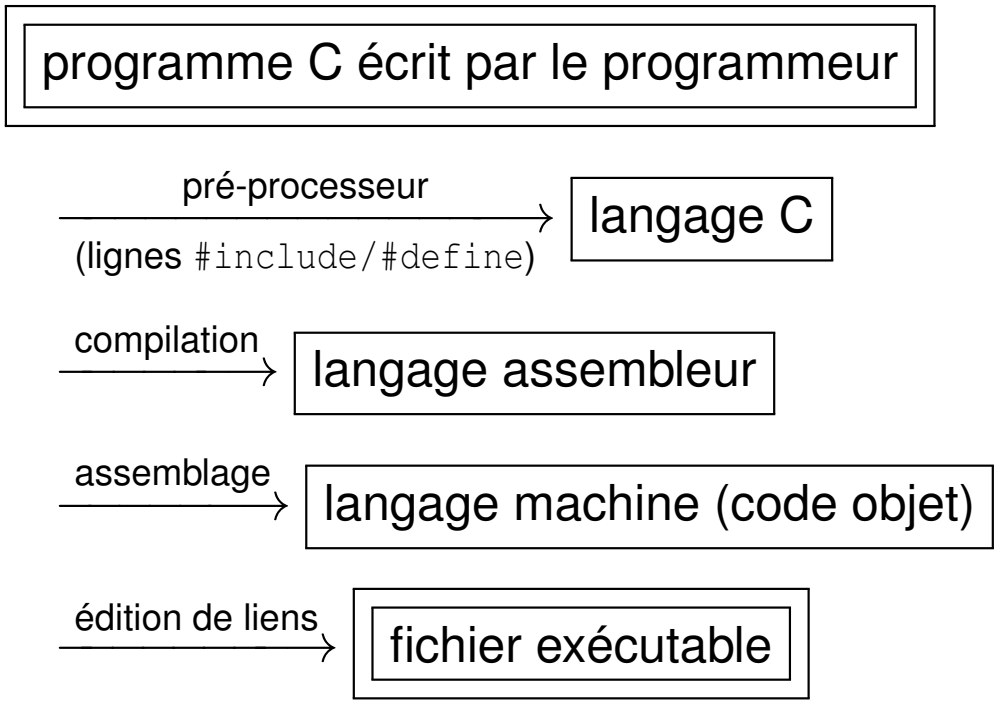
Compilation

- Les étapes de compilation sont :



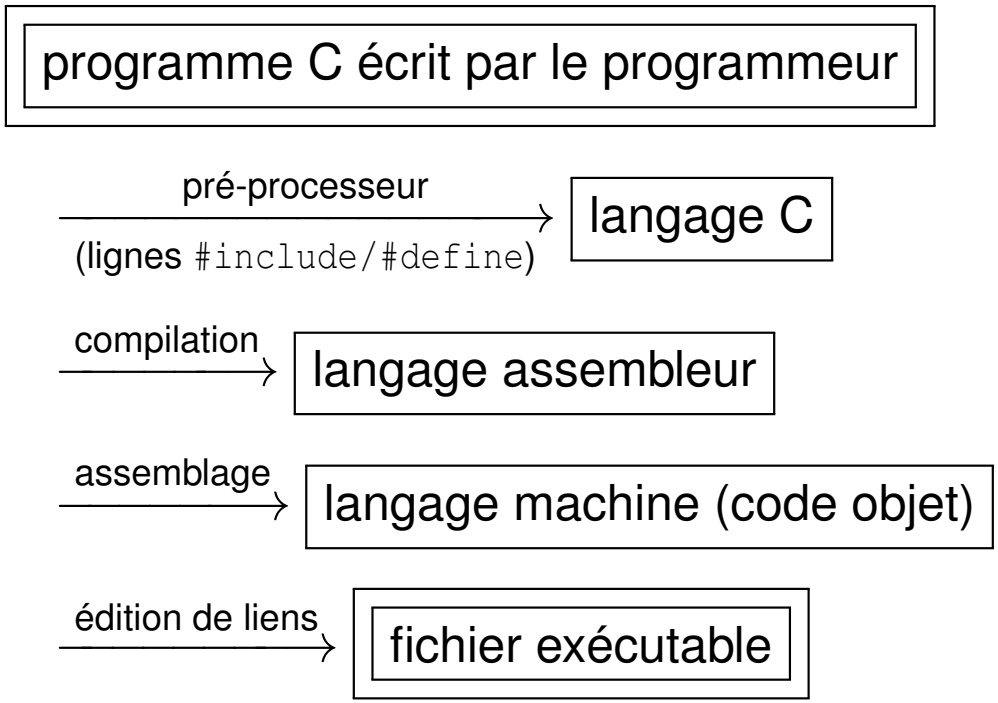
Compilation

- Les étapes de compilation sont :



Compilation

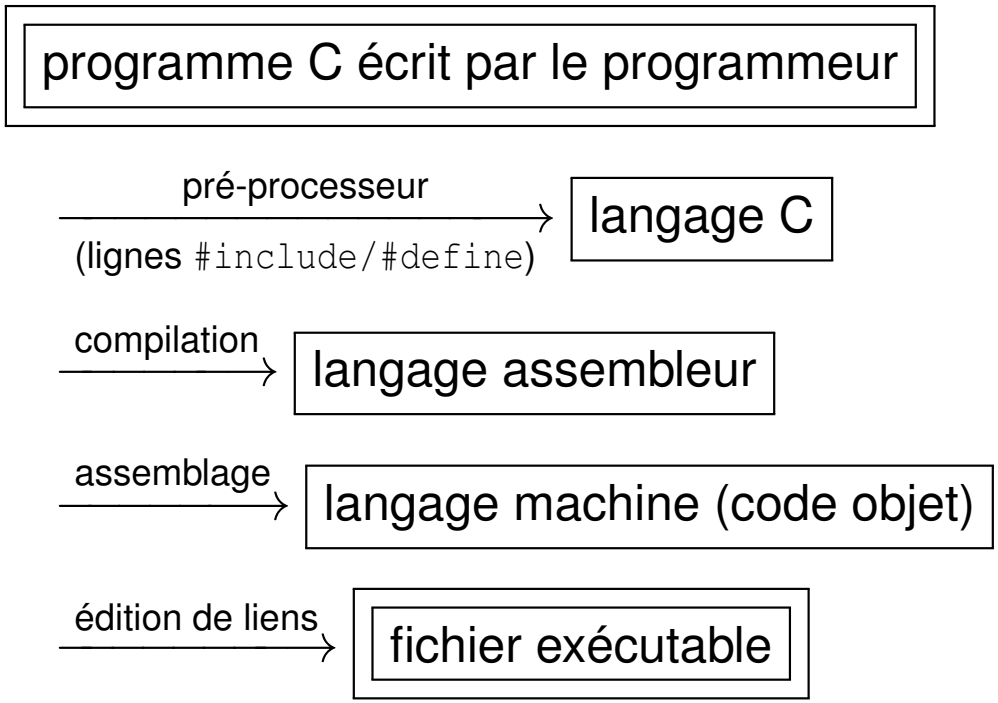
- Les étapes de compilation sont :



- En pratique, on appelle « compilateur » le programme qui exécute les différentes étapes.

Compilation

- Les étapes de compilation sont :



- En pratique, on appelle « compilateur » le programme qui exécute les différentes étapes.

NB : L'exécutable produit n'est valable que pour une architecture (ex : x86_64, arm, etc.) et un système donnés.

Exemple de programme C

- Enregistré avec l'extension `.c` (minuscule). Par exemple : `toto.c`

Exemple de programme C

- Enregistré avec l'extension `.c` (minuscule). Par exemple : `toto.c`

```
/* Un commentaire qui peut, si besoin,
   s'étendre sur plusieurs lignes */
// Une autre forme de commentaire

#include <stdio.h>      // ligne de préprocesseur
                        // Ici : inclure les déclarations du fichier d'en–tête stdio.h
                        //       (entrées/sorties standards)

/* Fonction principale (point d'entrée du programme). La signature est imposée.
 */
int main(void)
{
    printf ("Bonjour tout le monde !\n"); // écrire sur la sortie standard (terminal)
    return 0;                             // par convention 0 signifie OK
}
```

Compilation, exécution

- Pour compiler vers un exécutable :

```
$ gcc toto.c
```

Compilation, exécution

- Pour compiler vers un exécutable :

```
$ gcc toto.c
```

gcc compilateur

Compilation, exécution

- Pour compiler vers un exécutable :

```
$ gcc toto.c
```

gcc compilateur
toto.c code source

Compilation, exécution

- Pour compiler vers un exécutable :

```
$ gcc toto.c
```

gcc compilateur
toto.c code source

- Options recommandées (voire imposées !) :

```
$ gcc –g –Og –Wall –Wextra –o toto toto.c
```

Compilation, exécution

- Pour compiler vers un exécutable :

```
$ gcc toto.c
```

gcc compilateur
toto.c code source

- Options recommandées (voire imposées !) :

```
$ gcc –g –Og –Wall –Wextra –o toto toto.c
```

-o toto nom de l'exécutable à produire

Compilation, exécution

- Pour compiler vers un exécutable :

```
$ gcc toto.c
```

gcc compilateur
toto.c code source

- Options recommandées (voire imposées !) :

```
$ gcc –g –Og –Wall –Wextra –o toto toto.c
```

-o toto nom de l'exécutable à produire
-Wall -Wextra active les avertissements

Compilation, exécution

- Pour compiler vers un exécutable :

```
$ gcc toto.c
```

gcc compilateur
toto.c code source

- Options recommandées (voire imposées !) :

```
$ gcc –g –Og –Wall –Wextra –o toto toto.c
```

-o toto nom de l'exécutable à produire
-Wall -Wextra active les avertissements
-Og active des optimisations

Compilation, exécution

- Pour compiler vers un exécutable :

```
$ gcc toto.c
```

gcc compilateur
toto.c code source

- Options recommandées (voire imposées !) :

```
$ gcc –g –Og –Wall –Wextra –o toto toto.c
```

-o toto nom de l'exécutable à produire
-Wall -Wextra active les avertissements
-Og active des optimisations
-g inclure des informations de débogage

Compilation, exécution

- Pour compiler vers un exécutable :

```
$ gcc toto.c
```

gcc compilateur
toto.c code source

- Options recommandées (voire imposées !) :

```
$ gcc –g –Og –Wall –Wextra –o toto toto.c
```

-o toto nom de l'exécutable à produire
-Wall -Wextra active les avertissements
-Og active des optimisations
-g inclure des informations de débogage

- Exécution :

```
$ ./toto
```

Séance 2 : fonctions, tableaux, structures

Fonctions

Définition

La définition d'une fonction a la forme générale suivante :

```
type_de_retour nom_de_fonction(paramètres...)
{
    // corps de la fonction.
}
```

Fonctions

Définition

La définition d'une fonction a la forme générale suivante :

```
type_de_retour nom_de_fonction(paramètres...)
{
    // corps de la fonction.
}
```

Remarques

- Une fonction qui retourne une valeur le fera avec le mot clef **return**.

Fonctions

Définition

La définition d'une fonction a la forme générale suivante :

```
type_de_retour nom_de_fonction(paramètres...)
{
    // corps de la fonction.
}
```

Remarques

- Une fonction qui retourne une valeur le fera avec le mot clef **return**.
- Pour une fonction ne retournant rien, on utiliser **void** comme type de retour.

Fonctions

Définition

La définition d'une fonction a la forme générale suivante :

```
type_de_retour nom_de_fonction(paramètres...)
{
    // corps de la fonction.
}
```

Remarques

- Une fonction qui retourne une valeur le fera avec le mot clef **return**.
- Pour une fonction ne retournant rien, on utiliser **void** comme type de retour.
- Pour une fonction qui ne prend aucun paramètre, on indiquera **void**.

Exemples

Retour **void**

```
void affiche_entier(int x)
{
    printf("%i\n", x);
}
```

Exemples

Retour **void**

```
void affiche_entier(int x)
{
    printf("%i\n", x);
}
```

Paramètres **void**

```
int alea10(void)
{
    return rand() % 10;
}
```

Déclaration des fonctions

- Avant d'utiliser une fonction, celle-ci doit être déclarée.
  $\Rightarrow$ précédemment dans le code source

Déclaration des fonctions

- Avant d'utiliser une fonction, celle-ci doit être déclarée.
  $\Rightarrow$ précédemment dans le code source
- La déclaration est normalement faite par la définition de la fonction.

Déclaration des fonctions

- Avant d'utiliser une fonction, celle-ci doit être déclarée.
  $\Rightarrow$ précédemment dans le code source
- La déclaration est normalement faite par la définition de la fonction.
- Elle peut cependant aussi se faire en amont.

Déclaration des fonctions

- Avant d'utiliser une fonction, celle-ci doit être déclarée.
  $\Rightarrow$ précédemment dans le code source
- La déclaration est normalement faite par la définition de la fonction.
- Elle peut cependant aussi se faire en amont.

Exemple

```
#include <stdio.h>

int main(void)
{
    ma_jolie_fonction(33); // erreur : fonction non déclarée
}
void ma_jolie_fonction(int n) // définition de la fonction
{
    printf("*** %d ***\n", n);
}
```

Déclaration des fonctions

- Avant d'utiliser une fonction, celle-ci doit être déclarée.
  $\Rightarrow$ précédemment dans le code source
- La déclaration est normalement faite par la définition de la fonction.
- Elle peut cependant aussi se faire en amont.

Exemple

```
#include <stdio.h>
void ma_jolie_fonction(int n) ; // déclaration de la fonction
int main(void)
{
    ma_jolie_fonction(33); // erreur ok, la fonction est connue
}
void ma_jolie_fonction(int n) // définition de la fonction
{
    printf("*** %d ***\n", n);
}
```

Tableaux

Définition

Un tableau est déclaré de la manière suivante :

```
type nom[taille];
```

Tableaux

Définition

Un tableau est déclaré de la manière suivante :

```
type nom[taille];
```

Exemple

Déclaration d'un tableau de 42 entiers : tab[0], tab[1], . . ., tab[41] :

```
int tab[42];
```

Tableaux

Définition

Un tableau est déclaré de la manière suivante :

```
type nom[taille];
```

Exemple

Déclaration d'un tableau de 42 entiers : tab[0], tab[1], . . ., tab[41] :

```
int tab[42];
```

Attention

À l'usage, il n'y a aucune vérification des indices, c'est au programmeur de vérifier qu'il ne fait pas d'erreur.

Tableaux à plusieurs dimensions

Définition

Les tableaux à plusieurs dimensions sont définis comme des tableaux de tableaux.

Tableaux à plusieurs dimensions

Définition

Les tableaux à plusieurs dimensions sont définis comme des tableaux de tableaux.

Exemple

Un tableau de 5 lignes et 3 colonnes de nombres réels :

```
float matrice[5][3];
```

Tableaux et fonctions

- Une fonction peut prendre un tableau en paramètre.
- Celui-ci est passé par adresse.

Tableaux et fonctions

- Une fonction peut prendre un tableau en paramètre.
- Celui-ci est passé par adresse.

Exemple

```
void affiche_tab(int tab[], int n)
{
    for (int i = 0; i < n; i++) printf("tab[%i] = %i\n", i, tab[i]);
}
void somme(int a[], int n)
{
    for (int i = 1; i < n; i++) a[i] = a[i − 1] + a[i];
}
int main(void)
{
    int valeurs[5] = {1, 2, 3, 4, 5};
    somme(valeurs, 5); // ici, le contenu du tableau est modifié
    affiche_tab(valeurs, 5);
}
```

Tableaux et fonctions

- Une fonction peut prendre un tableau en paramètre.
- Celui-ci est passé par adresse.

Exemple

```
void affiche_tab(int tab[], int n)
{
    for (int i = 0; i < n; i++) printf("tab[%i] = %i\n", i, tab[i]);
}
void somme(int a[], int n)
{
    for (int i = 1; i < n; i++) a[i] = a[i – 1] + a[i];
}
int main(void)
{
    int valeurs[5] = {1, 2, 3, 4, 5};
    somme(valeurs, 5); // ici, le contenu du tableau est modifié
    affiche_tab(valeurs, 5);
}
```

Sortie

```
tab[0] = 1
tab[1] = 3
tab[2] = 6
tab[3] = 10
tab[4] = 15
```

Remarques

- Il faut en général passer la taille du tableau à l'aide d'un paramètre supplémentaire (*n* dans l'exemple).

Remarques

- Il faut en général passer la taille du tableau à l'aide d'un paramètre supplémentaire (*n* dans l'exemple).
- Dans la listes des paramètres l'indication de taille entre crochets est inutile.

Remarques

- Il faut en général passer la taille du tableau à l'aide d'un paramètre supplémentaire (*n* dans l'exemple).
- Dans la listes des paramètres l'indication de taille entre crochets est inutile.
- Pour les tableaux à plusieurs dimensions, les tailles des dimensions autres que la première doivent cependant être indiquées.

Remarques

- Il faut en général passer la taille du tableau à l'aide d'un paramètre supplémentaire (*n* dans l'exemple).
- Dans la listes des paramètres l'indication de taille entre crochets est inutile.
- Pour les tableaux à plusieurs dimensions, les tailles des dimensions autres que la première doivent cependant être indiquées.

Exemple

```
void affiche_mat(float mat[][3], int lignes)
{
    for (int i = 0; i < lignes; i++) {
        for (int j = 0; j < 3; j++)
            printf("%f,  ", mat[i][j]);
        printf("\n");
    }
}
```

Chaînes de caractères

Définition

En C, les chaînes de caractères sont représentées par des tableaux particuliers :

- tableau de *char*
- un des éléments du tableau doit avoir la valeur particulière 0 (ou '\0') qui indique la fin de la chaîne

Chaînes de caractères

Définition

En C, les chaînes de caractères sont représentées par des tableaux particuliers :

- tableau de *char*
- un des éléments du tableau doit avoir la valeur particulière 0 (ou '\0') qui indique la fin de la chaîne

Exemple

```
char nom[] = "Toto";
```

Chaînes de caractères

Définition

En C, les chaînes de caractères sont représentées par des tableaux particuliers :

- tableau de *char*
- un des éléments du tableau doit avoir la valeur particulière 0 (ou '\0') qui indique la fin de la chaîne

Exemple

```
char nom[] = "Toto";
```

- La taille n'est pas précisée (`nom[]`) : elle est calculée automatiquement par l'initialisation (5 pour l'exemple)

Chaînes de caractères

Définition

En C, les chaînes de caractères sont représentées par des tableaux particuliers :

- tableau de *char*
- un des éléments du tableau doit avoir la valeur particulière 0 (ou '\0') qui indique la fin de la chaîne

Exemple

```
char nom[] = "Toto";
```

- La taille n'est pas précisée (`nom[]`) : elle est calculée automatiquement par l'initialisation (5 pour l'exemple)
- En mémoire, le tableau a la forme suivante :

nom :

'T'	'o'	't'	'o'	'\0'

Structures

On peut définir des enregistrements (ou structures) pour gérer plusieurs données de types (éventuellement) différents.

Structures

On peut définir des enregistrements (ou structures) pour gérer plusieurs données de types (éventuellement) différents.

Exemple

```
struct point {
    int x; // champs ou attributs de la structure
    int y;
};
int main(void)
{
    struct point p; // déclaration d'une variable
    p.x = 1; // accès aux champs avec l'opérateur .
    p.y = 2;
    struct point p2;
    p2 = p; // affectation (copie)
    // ...
}
```

Séance 3 : pointeurs et allocation dynamique

Pointeurs

Définition

- Un pointeur est une variable particulière qui contient une adresse mémoire (par exemple l'adresse d'une autre variable).
- Un pointeur est typé : cela indique le type de la donnée à l'adresse pointée.
- Par exemple : pointeur vers int, pointeur vers float, etc.

Pointeurs

Définition

- Un pointeur est une variable particulière qui contient une adresse mémoire (par exemple l'adresse d'une autre variable).
- Un pointeur est typé : cela indique le type de la donnée à l'adresse pointée.
- Par exemple : pointeur vers int, pointeur vers float, etc.

Syntaxe

Pour déclarer un pointeur vers un type X, on notera :

```
X *nom_du_pointeur;    // X* est le type « pointeur vers X »
```

Exemples de déclarations

```
int *ptr_i ;        // pointeur vers int
int* ptr_j ;        // Variations : l'espace n'a pas d'importance
int * ptr_k;        //

double *ptr_d;      // pointeur vers double
char **tab2D;       // pointeur vers pointeur vers char
```

Valeurs d'un pointeur

- Définir un pointeur revient à l'initialiser avec une adresse a priori valide :

Valeurs d'un pointeur

- Définir un pointeur revient à l'initialiser avec une adresse a priori valide :
  - soit une valeur retournée par une fonction (par exemple, `malloc()`)

Valeurs d'un pointeur

- Définir un pointeur revient à l'initialiser avec une adresse a priori valide :
  - soit une valeur retournée par une fonction (par exemple, `malloc()`)
  - soit l'adresse d'une variable, obtenue avec l'opérateur d'adressage : **&**

Valeurs d'un pointeur

- Définir un pointeur revient à l'initialiser avec une adresse a priori valide :
  - soit une valeur retournée par une fonction (par exemple, `malloc()`)
  - soit l'adresse d'une variable, obtenue avec l'opérateur d'adressage : **&**
- On peut aussi initialiser un pointeur avec la valeur spéciale **0** (ou **NULL**) qui spécifie un pointeur toujours invalide.

Valeurs d'un pointeur

- Définir un pointeur revient à l'initialiser avec une adresse a priori valide :
  - soit une valeur retournée par une fonction (par exemple, `malloc()`)
  - soit l'adresse d'une variable, obtenue avec l'opérateur d'adressage : **&**
- On peut aussi initialiser un pointeur avec la valeur spéciale **0** (ou **NULL**) qui spécifie un pointeur toujours invalide.

Exemples

```
int entier = 12;
int *pi = &entier;     // adresse de la variable « entier »

int *pj = pi;          // copie de pointeur
int *pk;
pk = pi;               // et hop, une deuxième copie

float *pf = &entier;   // erreur : types non cohérents

char *x = NULL;        // pointeur initialisé à NULL
```

Valeur pointée

- Pour accéder à la valeur stockée à l'adresse donnée par le pointeur, on utilisera l'opérateur de déréférencement : *

Exemple

```
int entier = 12;
int *pi = &entier;

printf ("%d\n", entier); // entier vaut 12
printf ("%d\n", *pi);    // *pi est la valeur à l'adresse pi : la même valeur 12
```

En mémoire :

Autre exemple

```
float reel = 3.14, x = 42.0;
float *ptr = &reel;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
reel = 1.618;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
*ptr = 2.718;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
ptr = &x;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
ptr = NULL;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
```

Sortie

Autre exemple

```
float reel = 3.14, x = 42.0;
float *ptr = &reel;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
reel = 1.618;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
*ptr = 2.718;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
ptr = &x;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
ptr = NULL;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
```

Sortie

```
reel = 3.14 ; *ptr = 3.14
```

Autre exemple

```
float reel = 3.14, x = 42.0;
float *ptr = &reel;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
reel = 1.618;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
*ptr = 2.718;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
ptr = &x;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
ptr = NULL;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
```

Sortie

```
reel = 3.14 ; *ptr = 3.14
reel = 1.618 ; *ptr = 1.618
```

Autre exemple

```
float reel = 3.14, x = 42.0;
float *ptr = &reel;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
reel = 1.618;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
*ptr = 2.718;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
ptr = &x;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
ptr = NULL;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
```

Sortie

```
reel = 3.14 ; *ptr = 3.14
reel = 1.618 ; *ptr = 1.618
reel = 2.718 ; *ptr = 2.718
```

Autre exemple

```
float reel = 3.14, x = 42.0;
float *ptr = &reel;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
reel = 1.618;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
*ptr = 2.718;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
ptr = &x;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
ptr = NULL;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
```

Sortie

```
reel = 3.14 ; *ptr = 3.14
reel = 1.618 ; *ptr = 1.618
reel = 2.718 ; *ptr = 2.718
reel = 2.718 ; *ptr = 42
```

Autre exemple

```
float reel = 3.14, x = 42.0;
float *ptr = &reel;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
reel = 1.618;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
*ptr = 2.718;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
ptr = &x;
printf("reel = %f ; *ptr = %f\n", reel, *ptr);
ptr = NULL;
printf("reel = %f ; *ptr = %f\n", reel, *ptr); // ERREUR !
```

Sortie

```
reel = 3.14 ; *ptr = 3.14
reel = 1.618 ; *ptr = 1.618
reel = 2.718 ; *ptr = 2.718
reel = 2.718 ; *ptr = 42
Erreur de segmentation (core dumped)
```

Notation [ ]

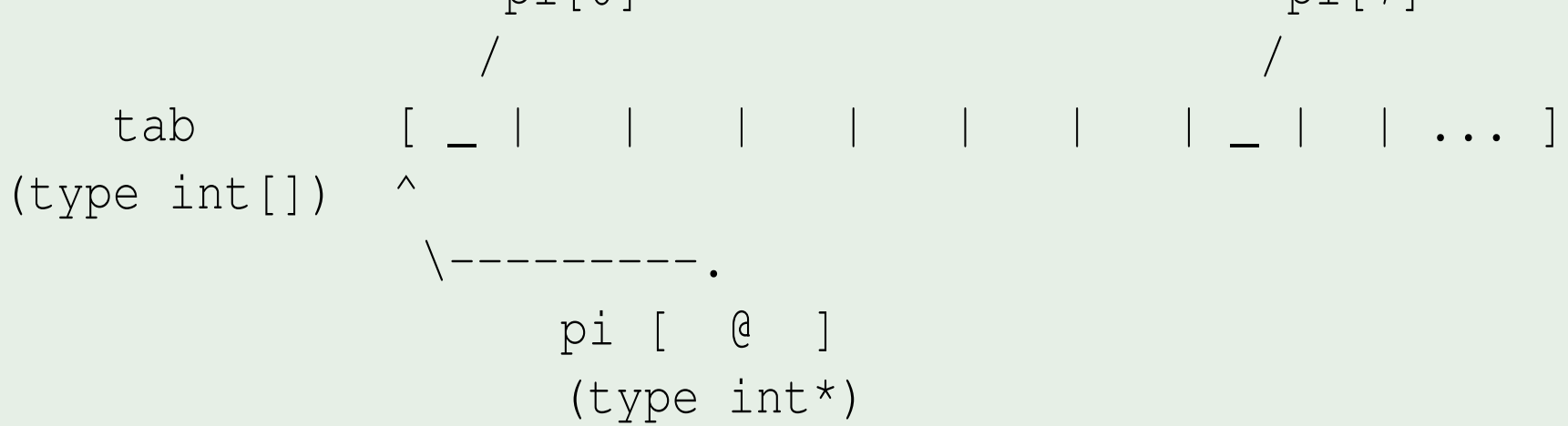
- On peut accéder à la i^e valeur à partir d'un pointeur avec les crochets **[ ]**

Exemple

```
int  tab[42];
int  *pi  = tab;          // pi pointe vers le début du tableau « tab »
                          // on aurait aussi pu écrire : pi = &tab[0]

printf ("%d\n", pi[7]); // équivalent à *(pi + 7)
printf ("%d\n", pi[0]); // équivalent à *pi
```

En mémoire :



Pointeurs et fonctions

- Les pointeurs peuvent être utilisés pour passer l'adresse d'une variable à une fonction. C'est utile par exemple :

Pointeurs et fonctions

- Les pointeurs peuvent être utilisés pour passer l'adresse d'une variable à une fonction. C'est utile par exemple :
  - pour pouvoir modifier la valeur de la variable depuis la fonction ;

Pointeurs et fonctions

- Les pointeurs peuvent être utilisés pour passer l'adresse d'une variable à une fonction. C'est utile par exemple :
  - pour pouvoir modifier la valeur de la variable depuis la fonction ;
  - lorsqu'on souhaite éviter une copie de la valeur de la variable ;

Pointeurs et fonctions

- Les pointeurs peuvent être utilisés pour passer l'adresse d'une variable à une fonction. C'est utile par exemple :
  - pour pouvoir modifier la valeur de la variable depuis la fonction ;
  - lorsqu'on souhaite éviter une copie de la valeur de la variable ;
  - pour passer un tableau en paramètre à une fonction.

Pointeurs et fonctions

- Les pointeurs peuvent être utilisés pour passer l'adresse d'une variable à une fonction. C'est utile par exemple :
  - pour pouvoir modifier la valeur de la variable depuis la fonction ;
  - lorsqu'on souhaite éviter une copie de la valeur de la variable ;
  - pour passer un tableau en paramètre à une fonction.
- **Remarque :** la notation [ ] introduite la séance précédente pour déclarer un paramètre de fonction n'est que du sucre syntaxique pour la notation * :

```
// Ces deux déclarations sont strictement équivalentes
int ma_jolie_fonction(int tab[], int n);
int ma_jolie_fonction(int *tab, int n);
```

Pointeurs et fonctions

- Les pointeurs peuvent être utilisés pour passer l'adresse d'une variable à une fonction. C'est utile par exemple :
  - pour pouvoir modifier la valeur de la variable depuis la fonction ;
  - lorsqu'on souhaite éviter une copie de la valeur de la variable ;
  - pour passer un tableau en paramètre à une fonction.
- **Remarque :** la notation [ ] introduite la séance précédente pour déclarer un paramètre de fonction n'est que du sucre syntaxique pour la notation * :

```
// Ces deux déclarations sont strictement équivalentes
int ma_jolie_fonction(int tab[], int n);
int ma_jolie_fonction(int *tab, int n);
```

- Ainsi, lorsqu'un tableau est passé en paramètre à une fonction, on dit qu'il *se dégrade* en pointeur.

Exemple qui ne fonctionne pas

```
void echanger(int a, int b)
{
    int c = a;
    a = b;
    b = c;
}
int main(void)
{
  int valeur1 = 5;
  int valeur2 = 7;
  printf("AVANT : valeur1 = %d ; valeur2 = %d\n", valeur1, valeur2);
  echanger(valeur1, valeur2); // ne fonctionne pas !
  printf("APRES : valeur1 = %d ; valeur2 = %d\n", valeur1, valeur2);
}
```

Sortie

Exemple qui ne fonctionne pas

```
void echanger(int a, int b)
{
    int c = a;
    a = b;
    b = c;
}
int main(void)
{
  int valeur1 = 5;
  int valeur2 = 7;
  printf("AVANT : valeur1 = %d ; valeur2 = %d\n", valeur1, valeur2);
  echanger(valeur1, valeur2); // ne fonctionne pas !
  printf("APRES : valeur1 = %d ; valeur2 = %d\n", valeur1, valeur2);
}
```

Sortie

```
AVANT : valeur1 = 5 ; valeur2 = 7
```

Exemple qui ne fonctionne pas

```
void echanger(int a, int b)
{
    int c = a;
    a = b;
    b = c;
}
int main(void)
{
  int valeur1 = 5;
  int valeur2 = 7;
  printf("AVANT : valeur1 = %d ; valeur2 = %d\n", valeur1, valeur2);
  echanger(valeur1, valeur2); // ne fonctionne pas !
  printf("APRES : valeur1 = %d ; valeur2 = %d\n", valeur1, valeur2);
}
```

Sortie

```
AVANT : valeur1 = 5 ; valeur2 = 7
APRES : valeur1 = 5 ; valeur2 = 7
```

Exemple qui fonctionne

```
void echanger(int *a, int *b)
{
    int c = *a;
    *a = *b;
    *b = c;
}
int main(void)
{
  int valeur1 = 5;
  int valeur2 = 7;
  printf("AVANT : valeur1 = %d ; valeur2 = %d\n", valeur1, valeur2);
  echanger(&valeur1, &valeur2); // ok
  printf("APRES : valeur1 = %d ; valeur2 = %d\n", valeur1, valeur2);
}
```

Sortie

Exemple qui fonctionne

```
void echanger(int *a, int *b)
{
    int c = *a;
    *a = *b;
    *b = c;
}
int main(void)
{
  int valeur1 = 5;
  int valeur2 = 7;
  printf("AVANT : valeur1 = %d ; valeur2 = %d\n", valeur1, valeur2);
  echanger(&valeur1, &valeur2); // ok
  printf("APRES : valeur1 = %d ; valeur2 = %d\n", valeur1, valeur2);
}
```

Sortie

```
AVANT : valeur1 = 5 ; valeur2 = 7
```

Exemple qui fonctionne

```
void echanger(int *a, int *b)
{
    int c = *a;
    *a = *b;
    *b = c;
}
int main(void)
{
  int valeur1 = 5;
  int valeur2 = 7;
  printf("AVANT : valeur1 = %d ; valeur2 = %d\n", valeur1, valeur2);
  echanger(&valeur1, &valeur2); // ok
  printf("APRES : valeur1 = %d ; valeur2 = %d\n", valeur1, valeur2);
}
```

Sortie

```
AVANT : valeur1 = 5 ; valeur2 = 7
APRES : valeur1 = 7 ; valeur2 = 5
```

Allocation dynamique

- L'allocation mémoire (réservation) statique ou automatique pour les variables a des inconvénients :

Allocation dynamique

- L'allocation mémoire (réservation) statique ou automatique pour les variables a des inconvénients :
  - le nombre de variables est fixé à l'écriture du programme ;

Allocation dynamique

- L'allocation mémoire (réservation) statique ou automatique pour les variables a des inconvénients :
  - le nombre de variables est fixé à l'écriture du programme ;
  - les tableaux sont de taille fixe ;

Allocation dynamique

- L'allocation mémoire (réservation) statique ou automatique pour les variables a des inconvénients :
  - le nombre de variables est fixé à l'écriture du programme ;
  - les tableaux sont de taille fixe ;
  - une variable déclarée est détruite lorsqu'on sort du bloc (par exemple à la fin d'une fonction) ;

Allocation dynamique

- L'allocation mémoire (réservation) statique ou automatique pour les variables a des inconvénients :
  - le nombre de variables est fixé à l'écriture du programme ;
  - les tableaux sont de taille fixe ;
  - une variable déclarée est détruite lorsqu'on sort du bloc (par exemple à la fin d'une fonction) ;
  - on ne peut pas libérer la mémoire lorsqu'on n'en a plus besoin ; etc.

Allocation dynamique

- L'allocation mémoire (réservation) statique ou automatique pour les variables a des inconvénients :
  - le nombre de variables est fixé à l'écriture du programme ;
  - les tableaux sont de taille fixe ;
  - une variable déclarée est détruite lorsqu'on sort du bloc (par exemple à la fin d'une fonction) ;
  - on ne peut pas libérer la mémoire lorsqu'on n'en a plus besoin ; etc.
- L'allocation dynamique est un mécanisme qui permet de résoudre ces problèmes.

Allocation dynamique en C

- La mémoire est allouée (réservée) à la demande avec la fonction :

```
#include <stdlib.h>
void *malloc(size_t taille);
```

Allocation dynamique en C

- La mémoire est allouée (réservée) à la demande avec la fonction :

```
#include <stdlib.h>
void *malloc(size_t taille);
```

- Cette fonction réserve un espace de *taille* octets en mémoire et en retourne l'adresse (`void*` est le type « pointeur générique »).

Allocation dynamique en C

- La mémoire est allouée (réservée) à la demande avec la fonction :

```
#include <stdlib.h>
void *malloc(size_t taille);
```

- Cette fonction réserve un espace de *taille* octets en mémoire et en retourne l'adresse (`void*` est le type « pointeur générique »).
- **Remarque :** `malloc()` retourne NULL en cas d'échec (la demande d'allocation ne peut pas être satisfaite).

Allocation dynamique en C

- La mémoire est allouée (réservée) à la demande avec la fonction :

```
#include <stdlib.h>
void *malloc(size_t taille);
```

- Cette fonction réserve un espace de *taille* octets en mémoire et en retourne l'adresse (`void*` est le type « pointeur générique »).
- **Remarque :** `malloc()` retourne NULL en cas d'échec (la demande d'allocation ne peut pas être satisfaite).
- Avant la fin du programme, lorsqu'on n'en a plus besoin, la mémoire allouée doit être libérée avec la fonction :

```
#include <stdlib.h>
void free(void *ptr);
```

Exemple : tableau dynamique

```
int *tab;
tab = malloc(47235 * sizeof (int)); // allocation d'un tableau de 47235 entiers

if (tab == NULL) {                  // on vérifie que l'allocation a réussi
    printf ("Erreur!\n");
    exit (1);
}
// ... on peut maintenant utiliser tab[0] ... tab[47234]
// ...
free(tab); // libération de la mémoire
```

Exemple : tableau dynamique

```
int *tab;
tab = malloc(47235 * sizeof (int)); // allocation d'un tableau de 47235 entiers

if (tab == NULL) {                  // on vérifie que l'allocation a réussi
    printf ("Erreur!\n");
    exit (1);
}
// ... on peut maintenant utiliser tab[0] ... tab[47234]
// ...
free(tab); // libération de la mémoire
```

- La mémoire réservée par `malloc()` n'est pas initialisée.

Exemple : tableau dynamique

```
int *tab;
tab = malloc(47235 * sizeof (int)); // allocation d'un tableau de 47235 entiers

if (tab == NULL) {                  // on vérifie que l'allocation a réussi
    printf ("Erreur!\n");
    exit (1);
}
// ... on peut maintenant utiliser tab[0] ... tab[47234]
// ...
free(tab); // libération de la mémoire
```

- La mémoire réservée par `malloc()` n'est pas initialisée.
- Après l'appel à `free()`, on n'a plus le droit d'utiliser la mémoire (`*tab`, `tab[i]`, ..., sont interdits).

Exemple : tableau dynamique

```
int *tab;
tab = malloc(47235 * sizeof (int)); // allocation d'un tableau de 47235 entiers

if (tab == NULL) {                  // on vérifie que l'allocation a réussi
    printf ("Erreur!\n");
    exit (1);
}
// ... on peut maintenant utiliser tab[0] ... tab[47234]
// ...
free(tab); // libération de la mémoire
tab = NULL;
```

- La mémoire réservée par `malloc()` n'est pas initialisée.
- Après l'appel à `free()`, on n'a plus le droit d'utiliser la mémoire (`*tab`, `tab[i]`, ..., sont interdits).
- C'est une bonne pratique d'ajouter `tab = NULL;` après l'appel à `free()`.

Exemple : variable de type structure

```
struct Point {
    int x;
    int y;
};
// ...
struct Point *ptr_point;
ptr_point = malloc(sizeof (struct Point));
if (ptr_point != NULL) {
    // ... utilisation de *ptr_point
    // ...
    free(ptr_point);
}
```

Remarque (syntaxe) pour accéder aux champs d'une structure par un pointeur, on peut utiliser l'opérateur **->**

ptr_point->x	*est équivalent à*	(*ptr_point).x

Séance 4 : compilation séparée, Makefile

Retour sur les étapes de la compilation

- On peut observer les différentes étapes de compilation avec l'option de *gcc* « `-v` ». Par exemple :

```
$ gcc –v –o hello hello.c
```

Retour sur les étapes de la compilation

- On peut observer les différentes étapes de compilation avec l'option de *gcc* « `-v` ». Par exemple :

```
$ gcc –v –o hello hello.c
```

- Les principales étapes sont :

```
$ gcc -no-integrated-cpp -v -Wl,-v -o hello hello.c
```

Retour sur les étapes de la compilation

- On peut observer les différentes étapes de compilation avec l'option de *gcc* « `-v` ». Par exemple :

```
$ gcc –v –o hello hello.c
```

- Les principales étapes sont :

```
$ gcc -no-integrated-cpp -v -Wl,-v -o hello hello.c
```

1. passage du pré-processeur

```
cc1 -E [...] hello.c [...] -march=x86-64 -o /tmp/cc8tzfJe.i
```

Retour sur les étapes de la compilation

- On peut observer les différentes étapes de compilation avec l'option de *gcc* « `-v` ». Par exemple :

```
$ gcc –v –o hello hello.c
```

- Les principales étapes sont :

```
$ gcc -no-integrated-cpp -v -Wl,-v -o hello hello.c
```

1. passage du pré-processeur

   ```
   cc1 -E [...] hello.c [...] -march=x86-64 -o /tmp/cc8tzfJe.i
   ```

2. compilation

   ```
   cc1 -fpreprocessed /tmp/cc8tzfJe.i -march=x86-64 [...] -o /tmp/cct4savT.s
   ```

Retour sur les étapes de la compilation

- On peut observer les différentes étapes de compilation avec l'option de *gcc* « `-v` ». Par exemple :

```
$ gcc –v –o hello hello.c
```

- Les principales étapes sont :

```
$ gcc -no-integrated-cpp -v -Wl,-v -o hello hello.c
```

1. passage du pré-processeur

   ```
   cc1 -E [...] hello.c [...] -march=x86-64 -o /tmp/cc8tzfJe.i
   ```

2. compilation

   ```
   cc1 -fpreprocessed /tmp/cc8tzfJe.i -march=x86-64 [...] -o /tmp/cct4savT.s
   ```

3. assemblage

   ```
   as -64 -o /tmp/ccJlR2ux.o /tmp/cct4savT.s
   ```

Retour sur les étapes de la compilation

- On peut observer les différentes étapes de compilation avec l'option de *gcc* « `-v` ». Par exemple :

```
$ gcc –v –o hello hello.c
```

- Les principales étapes sont :

```
$ gcc -no-integrated-cpp -v -Wl,-v -o hello hello.c
```

1. passage du pré-processeur

   ```
   cc1 -E [...] hello.c [...] -march=x86-64 -o /tmp/cc8tzfJe.i
   ```

2. compilation

   ```
   cc1 -fpreprocessed /tmp/cc8tzfJe.i -march=x86-64 [...] -o /tmp/cct4savT.s
   ```

3. assemblage

   ```
   as -64 -o /tmp/ccJlR2ux.o /tmp/cct4savT.s
   ```

4. édition de liens

   ```
   collect2 [...] -m elf_x86_64 -o hello [...] /tmp/ccJlR2ux.o -lc -lgcc [...]
   ld [...] -m elf_x86_64 [...] -o hello [...] /tmp/ccJlR2ux.o -lc -lgcc [...]
   ```

Compilation, résultats intermédiaires

- Il est possible d'interrompre le compilateur après chacune des étapes pour observer le résultat.

Compilation, résultats intermédiaires

- Il est possible d'interrompre le compilateur après chacune des étapes pour observer le résultat.
  - option « `-E` » pour arrêter après le passage du pré-processeur

Compilation, résultats intermédiaires

- Il est possible d'interrompre le compilateur après chacune des étapes pour observer le résultat.
  - option « `-E` » pour arrêter après le passage du pré-processeur
  - option « `-S` » pour arrêter avec le code assembleur

Compilation, résultats intermédiaires

- Il est possible d'interrompre le compilateur après chacune des étapes pour observer le résultat.
  - option « `-E` » pour arrêter après le passage du pré-processeur
  - option « `-S` » pour arrêter avec le code assembleur
  - option « `-c` » pour arrêter avec le code objet (binaire)

Compilation, résultats intermédiaires

- Il est possible d'interrompre le compilateur après chacune des étapes pour observer le résultat.
  - option « `-E` » pour arrêter après le passage du pré-processeur
  - option « `-S` » pour arrêter avec le code assembleur
  - option « `-c` » pour arrêter avec le code objet (binaire)
- Exemple avec le programme suivant : `hello.c`

```
#include <stdio.h>
#define MESSAGE "Hello World!\n"

/*********************************
 * Un joli programme d'exemple *
 *********************************/
int main(void)
{
    printf(MESSAGE);
    return 0;
}
```

Après passage du pré-processeur, fichier `hello.i`

```
$ gcc –E hello.c –o hello.i
```

```
# 1 "hello.c"
// [...]
# 143 "/usr/include/stdio.h" 3 4
extern FILE *stdin;
extern FILE *stdout;
extern FILE *stderr;
// [...]
# 328 "/usr/include/stdio.h" 3 4
// [...]
extern int printf (const char *__restrict __format, ...);
// [...]
extern int scanf (const char *__restrict __format, ...) ;
// [...]
# 909 "/usr/include/stdio.h" 3 4

# 2 "hello.c" 2


# 7 "hello.c"
int main(void)
{
    printf("Hello World!\n");
    return 0;
}
```

Après la compilation, code assembleur, fichier `hello.s`

$ gcc –S hello.c *# Note : tester avec l'option -fverbose-asm*

```
        .file "hello.c"
        .text
        .section .rodata
.LC0:
        .string "Hello World!"
        .text
        .globl main
        .type main, @function
main:
.LFB0:
        .cfi_startproc
        pushq %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, –16
        movq %rsp, %rbp
        .cfi_def_cfa_register 6
        leaq .LC0(%rip), %rax
        movq %rax, %rdi
        call puts@PLT
        movl $0, %eax
        popq %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size main, .–main
        .ident "GCC: (Debian 12.2.0-14+deb12u1) 12.2.0"
        .section .note.GNU–stack,"",@progbits
```

Après assemblage, code objet (binaire), fichier `hello.o`

$ gcc –c hello.c

```
$ file hello.o
hello.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
$ hd hello.o
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  01 00 3e 00 01 00 00 00  00 00 00 00 00 00 00 00  |..>.............|
00000020  00 00 00 00 00 00 00 00  20 02 00 00 00 00 00 00  |........ .......|
00000030  00 00 00 00 40 00 00 00  00 00 40 00 0d 00 0c 00  |....@.....@.....|
00000040  55 48 89 e5 48 8d 05 00  00 00 00 48 89 c7 e8 00  |UH..H......H....|
00000050  00 00 00 b8 00 00 00 00  5d c3 48 65 6c 6c 6f 20  |........].Hello |
00000060  57 6f 72 6c 64 21 00 00  47 43 43 3a 20 28 44 65  |World!..GCC: (De|
00000070  62 69 61 6e 20 31 32 2e  32 2e 30 2d 31 34 2b 64  |bian 12.2.0-14+d|
00000080  65 62 31 32 75 31 29 20  31 32 2e 32 2e 30 00 00  |eb12u1) 12.2.0..|
00000090  14 00 00 00 00 00 00 00  01 7a 52 00 01 78 10 01  |.........zR..x..|
000000a0  1b 0c 07 08 90 01 00 00  1c 00 00 00 1c 00 00 00  |................|
000000b0  00 00 00 00 1a 00 00 00  00 41 0e 10 86 02 43 0d  |.........A....C.|
000000c0  06 55 0c 07 08 00 00 00  00 00 00 00 00 00 00 00  |.U..............|
000000d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000e0  01 00 00 00 04 00 f1 ff  00 00 00 00 00 00 00 00  |................|
000000f0  00 00 00 00 00 00 00 00  00 00 00 00 03 00 01 00  |................|
00000100  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000110  00 00 00 00 03 00 05 00  00 00 00 00 00 00 00 00  |................|
00000120  00 00 00 00 00 00 00 00  09 00 00 00 12 00 01 00  |................|
00000130  00 00 00 00 00 00 00 00  1a 00 00 00 00 00 00 00  |................|
00000140  0e 00 00 00 10 00 00 00  00 00 00 00 00 00 00 00  |................|
00000150  00 00 00 00 00 00 00 00  00 68 65 6c 6c 6f 2e 63  |.........hello.c|
00000160  00 6d 61 69 6e 00 70 75  74 73 00 00 00 00 00 00  |.main.puts......|
00000170  07 00 00 00 00 00 00 00  02 00 00 00 03 00 00 00  |................|
00000180  fc ff ff ff ff ff ff ff  0f 00 00 00 00 00 00 00  |................|
```

Édition de liens vers un exécutable, fichier `hello`

- Les différents codes objets (`.o`) peuvent ensuite être combinés en un programme exécutable avec l'éditeur de liens : *ld*.

Édition de liens vers un exécutable, fichier `hello`

- Les différents codes objets (`.o`) peuvent ensuite être combinés en un programme exécutable avec l'éditeur de liens : *ld*.
- En pratique, on utilisera *gcc* qui invoquera *ld* avec les bonnes options.

Édition de liens vers un exécutable, fichier `hello`

- Les différents codes objets (`.o`) peuvent ensuite être combinés en un programme exécutable avec l'éditeur de liens : *ld*.
- En pratique, on utilisera *gcc* qui invoquera *ld* avec les bonnes options.
- Exemple :

```
$ gcc -o hello hello.o
$ file hello
hello: ELF 64-bit LSB pie executable, x86-64, version 1
(SYSV), dynamically linked, interpreter /lib64/ld-linux-
x86-64.so.2, BuildID[sha1]=c6fe8ac2e5be3c78cf31179a5cb31
df3978ca160, for GNU/Linux 3.2.0, not stripped

$ ./hello
Hello World!
```

Compilation séparée

- Pour de gros projets, le code source est généralement réparti dans plusieurs fichiers C (unités de compilation).

Compilation séparée

- Pour de gros projets, le code source est généralement réparti dans plusieurs fichiers C (unités de compilation).
- Pour construire l'exécutable, il faut assembler l'ensemble des unités de compilation. Deux solutions :

Compilation séparée

- Pour de gros projets, le code source est généralement réparti dans plusieurs fichiers C (unités de compilation).
- Pour construire l'exécutable, il faut assembler l'ensemble des unités de compilation. Deux solutions :
  1. Tout compiler en une seule fois :

```
$ gcc –g –Og –Wall –Wextra –o mon_programme fichier1.c fichier2.c fichier3.c
```

Inconvénient : tous les fichiers sont recompilés à chaque fois

Compilation séparée

- Pour de gros projets, le code source est généralement réparti dans plusieurs fichiers C (unités de compilation).
- Pour construire l'exécutable, il faut assembler l'ensemble des unités de compilation. Deux solutions :
  1. Tout compiler en une seule fois :

```
$ gcc –g –Og –Wall –Wextra –o mon_programme fichier1.c fichier2.c fichier3.c
```

Inconvénient : tous les fichiers sont recompilés à chaque fois

  2. Compiler chaque fichier séparément et assembler le tout :

```
$ gcc –g –Og –Wall –Wextra –c fichier1.c
$ gcc –g –Og –Wall –Wextra –c fichier2.c
$ gcc –g –Og –Wall –Wextra –c fichier3.c
```

(⇒ les fichiers objet `fichier1.o`, `fichier2.o`, `fichier3.o` sont générés)

```
$ gcc –o mon_programme fichier1.o fichier2.o fichier3.o
```

Question

Comment répartir le code dans plusieurs fichiers ?

Question

Comment répartir le code dans plusieurs fichiers ?

Exemple

- On définit une fonction :

```
int age(int annee) { /* ... */ }
```

- Cette fonction est utilisée dans un programme.
- Les fonctions `age()` et `main()` seront dans des fichiers séparés.

Fichier d'en-tête

- On définit un fichier d'en-tête contenant la déclaration des fonctions.

Fichier d'en-tête

- On définit un fichier d'en-tête contenant la déclaration des fonctions.
- Ce fichier porte l'extension : `.h`

Fichier d'en-tête

- On définit un fichier d'en-tête contenant la déclaration des fonctions.
- Ce fichier porte l'extension : `.h`
- Il est à placer dans le répertoire des fichiers source.

Fichier d'en-tête

- On définit un fichier d'en-tête contenant la déclaration des fonctions.
- Ce fichier porte l'extension : `.h`
- Il est à placer dans le répertoire des fichiers source.

age.h

```
/* Fichier d'en-tête.
 * (ne contient normalement que des déclarations)
 */
#ifndef AGE_H              // garde contre les inclusions multiples
#define AGE_H              // le symbole est généralement construit à partir du nom du fichier

int age(int annee);        // déclaration de la fonction

#endif
```

Code de la fonction

- Les définitions sont placées dans un ficher `.c`

Code de la fonction

- Les définitions sont placées dans un ficher `.c`
- C'est une bonne pratique d'inclure le fichier d'en-tête pour s'assurer de la cohérence de la déclaration.

Code de la fonction

- Les définitions sont placées dans un ficher `.c`
- C'est une bonne pratique d'inclure le fichier d'en-tête pour s'assurer de la cohérence de la déclaration.

age.c

```
#include "age.h" // noter l'utilisation des guillemets (double quotes)

/* Définition de la fonction.
 * (la signature doit être cohérente avec la déclaration dans age.h)
 */
int age(int annee)
{
    return annee - 1903;
}
```

Programme principal

- Il faut inclure les fichier d'en-tête pour les fonctions utilisées.

Programme principal

- Il faut inclure les fichier d'en-tête pour les fonctions utilisées.

main.c

```c
#include <stdio.h> // pour la fonction printf()
#include "age.h" // pour la fonction age()

int main(void)
{
    printf("L'âge du capitaine est %d\n", age(2026));
    return 0;
}
```

Programme principal

- Il faut inclure les fichier d'en-tête pour les fonctions utilisées.

main.c

```
#include <stdio.h> // pour la fonction printf()
#include "age.h" // pour la fonction age()

int main(void)
{
    printf("L'âge du capitaine est %d\n", age(2026));
    return 0;
}
```

- Lors de la compilation, on compilera uniquement les fichier `.c` :

```
$ gcc –g –Og –Wall –Wextra –c age.c
$ gcc –g –Og –Wall –Wextra –c main.c
$ gcc –o mon_programme main.o age.o
```

Automatiser la compilation

- L'outil **make** permet d'automatiser la compilation et de ne reconstruire que ce qui est nécessaire.

Automatiser la compilation

- L'outil **make** permet d'automatiser la compilation et de ne reconstruire que ce qui est nécessaire.
- Pour cela, on décrit le projet dans un fichier nommé **Makefile**

Automatiser la compilation

- L'outil **make** permet d'automatiser la compilation et de ne reconstruire que ce qui est nécessaire.
- Pour cela, on décrit le projet dans un fichier nommé **Makefile**
  - on y liste les choses à construire (par exemple le programme exécutable) ;

Automatiser la compilation

- L'outil **make** permet d'automatiser la compilation et de ne reconstruire que ce qui est nécessaire.
- Pour cela, on décrit le projet dans un fichier nommé **Makefile**
  - on y liste les choses à construire (par exemple le programme exécutable) ;
  - on décrit les dépendances entre les fichiers, ainsi que les commandes permettant d'arriver au résultat.

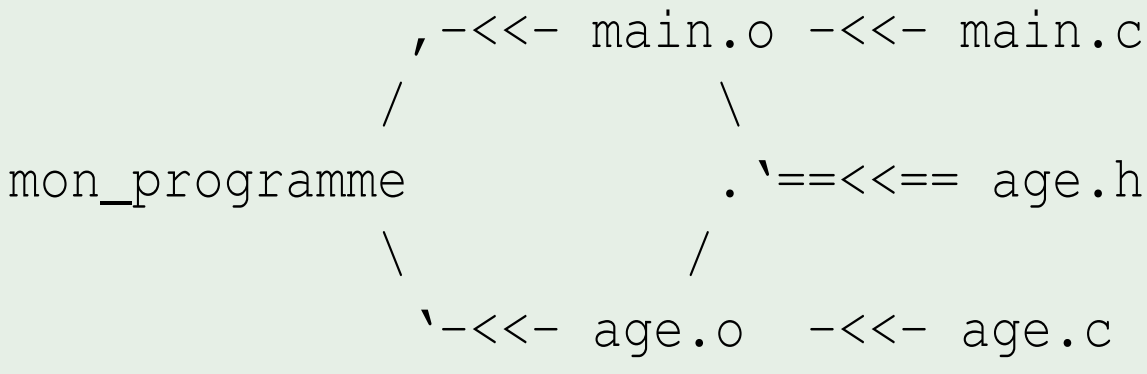
Automatiser la compilation

- L'outil **make** permet d'automatiser la compilation et de ne reconstruire que ce qui est nécessaire.
- Pour cela, on décrit le projet dans un fichier nommé **Makefile**
  - on y liste les choses à construire (par exemple le programme exécutable) ;
  - on décrit les dépendances entre les fichiers, ainsi que les commandes permettant d'arriver au résultat.
- L'outil *make* se base sur les dates des fichiers pour décider lesquels reconstruire.

Automatiser la compilation

- L'outil **make** permet d'automatiser la compilation et de ne reconstruire que ce qui est nécessaire.
- Pour cela, on décrit le projet dans un fichier nommé **Makefile**
  - on y liste les choses à construire (par exemple le programme exécutable) ;
  - on décrit les dépendances entre les fichiers, ainsi que les commandes permettant d'arriver au résultat.
- L'outil *make* se base sur les dates des fichiers pour décider lesquels reconstruire.

Exemple : les dépendances



Exemple de fichier `Makefile`

Makefile

```
# Les commentaires sont introduits par le  caractère #

mon_programme: main.o age.o
	gcc –o mon_programme main.o age.o
#    ^–– (tabulation)

main.o: main.c age.h
	gcc –g –Og –Wall –Wextra –c main.c

age.o: age.c age.h
	gcc –g –Og –Wall –Wextra –c age.c
```

Exemple de fichier `Makefile`

Makefile

```
# Les commentaires sont introduits par le  caractère #

mon_programme: main.o age.o
	gcc –o mon_programme main.o age.o
#    ^–– (tabulation)

main.o: main.c age.h
	gcc –g –Og –Wall –Wextra –c main.c

age.o: age.c age.h
	gcc –g –Og –Wall –Wextra –c age.c
```

Utilisation :

```
$ make mon_programme
```

Exemple de fichier `Makefile`

Makefile

```
# Les commentaires sont introduits par le  caractère #

mon_programme: main.o age.o
	gcc –o mon_programme main.o age.o
#    ^–– (tabulation)

main.o: main.c age.h
	gcc –g –Og –Wall –Wextra –c main.c

age.o: age.c age.h
	gcc –g –Og –Wall –Wextra –c age.c
```

Utilisation :

```
$ make mon_programme
```

```
$ make      # construction de la cible par défaut (la première)
```

Exemple avec des règles génériques

Makefile

```
CC = gcc
CFLAGS = –g –Og –Wall –Wextra

mon_programme: main.o age.o
	$(CC) –o $@ $^

main.o: main.c age.h
age.o: age.c age.h

%.o: %.c
	$(CC) $(CFLAGS) –c $<
```

Remarque : l'utilisation de *make* ne se limite pas à la compilation de programmes en C !

Conclusion