

Programmation

Le langage C

K. Boudjelaba

BTS SNIR – CIEL, Carnus



Table des matières

Introduction

Programmation

Structure d'un programme en C

Structures de contrôle

Les boucles

Les tableaux

Les chaînes de caractères

Les fonctions

Les structures

Les énumérations

Compilation

Organisation d'un projet

La fonction `main`

Les directives de préprocesseur

Découper le programme en plusieurs fichiers

Les fichiers

Introduction

Un ordinateur est une machine qui ne comprend qu'un langage très simple constitué de 0 et de 1, appelé le langage machine.

→ Le programme doit être programmé en binaire : une suite de 0 et 1 (compliqué).

Afin de simplifier la programmation, les informaticiens ont développé des langages intermédiaires, plus simples que le binaire.

Étapes d'interprétation d'un programme par l'ordinateur

- ▶ Écriture du programme (instructions) dans un langage de programmation (C++, Python ...)
- ▶ Traduction des instructions en binaire grâce à un programme de traduction (compilateur : transformer le code, écrit dans un langage de programmation, en un programme exécutable par la machine)
- ▶ Lecture du code binaire et execution des instructions

Introduction

Langage de haut niveau et de bas niveau

Le langage de haut niveau est un langage éloigné du binaire (langage machine) et permet généralement de développer de façon plus souple et rapide.

Le langage de bas niveau est plus proche du langage machine, il demande en général un peu plus d'efforts et permet un contrôle plus simple des instructions.

- ▶ Programmes de haut niveau : Python, Matlab, Java ...
- ▶ Programmes de bas niveau : C++ ...
- ▶ Programme machine : assembleur.

Introduction - Installation

Il faut installer certains logiciels spécifiques pour programmer en C

Éditeur de texte

Permet d'écrire le code source du programme en C. L'idéal est d'avoir un éditeur de texte intelligent qui colore tout seul le code, ce qui permet de repérer bien plus facilement les différentes instructions.

Compilateur

Permet de compiler (transformer) le code source en binaire.

Debugger

Permet de détecter certaines erreurs de programmation.

Installation

- ▶ On récupère chacun de ces 3 programmes séparément (méthode compliquée)

Introduction - Installation

- ▶ Ou on utilise un programme qui combine éditeur de texte, compilateur et debugger. Ces programmes sont appelés IDE (EDI en français : Environnement de Développement Intégré).

Quand on code un programme, l'ordinateur génère plusieurs fichiers de code source : des fichiers.c, .h, les images du programme...

Le rôle d'un IDE est de rassembler tous ces fichiers au sein d'une même interface. → on a accès à tous les éléments du programme.

Un IDE est un logiciel informatique qui rassemble un certain nombre d'outils nécessaires ou commodes pour l'écriture de programmes informatiques, leur compilation, leur exécution et, si besoin est, le dépannage (débugage) quand le programme ne donne pas le résultat escompté.

Choix de l'IDE

- ▶ **Code::Blocks** : Gratuit et disponible pour la plupart des systèmes d'exploitation (Windows, Mac OS 32 bits et Linux).
- ▶ **XCode** : Gratuit et disponible sur Mac OS uniquement.
- ▶ **Visual Studio Code** : Nécessite l'installation d'extensions ...

Programmation

Programme

est une suite finie d'instructions élémentaires exécutables par ordinateur.

Traitement de l'information (données)

- ▶ Saisie des données (entrées)
- ▶ Mémorisation (stockage) des données
- ▶ Opérations sur les données : calcul, tri ...
- ▶ Restitution des résultats : affichage, impression, fichier ...

Instruction

Une instruction est un ordre qu'on demande à un ordinateur (programme) d'exécuter.

Exemple

Calculer $2 + 3$ est une instruction

+ est le nom de l'opération (opérateur)

2 et 3 sont les données (opérandes)

Programmation

Etapas de résolution d'un problème en programmation

- ▶ Etablir la liste des données en entrées (données à saisir et/ou à initialiser), la liste des données en sortie (résultats : données à afficher) et les liens entre elles.
Construire un schéma de résolution qui permet d'obtenir les données en sortie à partir des données en entrée : c'est l'algorithme.
- ▶ Décrire le schéma de résolution en terme d'instruction élémentaires acceptées par l'ordinateur : c'est le programme.

Programmation

Exemple 1 :

Problème : Calcul du périmètre d'un cercle ($P = 2 \times \pi \times R$)

1. Identification des données d'entrées et de sorties :
 - ▶ Données en entrées : le rayon (R), π
 - ▶ Données en sorties : le périmètre (P)
 - ▶ Relation entre les données : $P = 2\pi R$
2. Chemin (schéma) de résolution :
 - ▶ Donner une valeur au rayon (affectation ou lecture)
 - ▶ Calculer $2 * \pi * R$
 - ▶ Mettre $2 * \pi * R$ dans périmètre (affectation) : $P = 2 * \pi * R$
 - ▶ Afficher la valeur du périmètre (écriture)
3. Traduire le chemin (schéma) en programme.

```
double r = valeur_de_r, pi = valeur_de_pi, p;  
p = 2*pi*r;  
printf("%.5f Le périmètre est : ",p);
```

Programmation

Exemple 2 :

Problème : Permuter les valeurs de deux variables x et y (utilisation de la méthode difficile).

1. Identification des données d'entrées et de sorties :

- ▶ Données en entrées : x et y
- ▶ Données en sorties : x et y échangées

2. Chemin (schéma) de résolution :

- ▶ Donner des valeurs à x et y
- ▶ Echanger les valeurs de x et y
- ▶ Afficher x et y

3. Traduire le chemin (schéma) en programme.

```
double x = valeur_de_x, y = valeur_de_y;  
x = x - y;  
y = x + y; // y = x - y + y = x  
x = y - x; // x = x - (x - y) = x - x + y = y  
printf("%f x nouveau = \t %f y nouveau = ", x,y);
```

Structure d'un programme en C

La structure d'un programme C est proche de celle d'un algorithme. Le fichier, qui doit avoir l'extension `.c`, commence par l'importation (l'inclusion) des bibliothèques (librairies).

La pratique du C exige l'utilisation de bibliothèques de fonctions. Ces bibliothèques sont disponibles sous forme précompilées (`.lib`). Afin de pouvoir les utiliser, il faut inclure des fichiers en-tête (`.h`) dans nos programmes. Ces fichiers contiennent les prototypes des fonctions prédéfinies dans les bibliothèques et créent un lien entre les fonctions précompilées et nos programmes.

Pour inclure les fichiers en-tête : `include <fichier.h>`

Pour le compilateur que nous utiliserons, différents types de fichiers seront identifiés par leurs extensions:

- `.c` : fichier source
- `.obj` : fichier compilé
- `.exe` : fichier exécutable
- `.lib` : bibliothèque de fonctions précompilées
- `.h` : bibliothèque en-tête
- ...

Structure d'un programme en C

Les `#include` correspondent à des directives qui indiquent au compilateur (en fait au pré-processeur) d'inclure les fichiers (nommés par exemple `stdlib`). Ces fichiers font partie de la bibliothèque standard du C et donne accès à des fonctions déjà définies.

Par exemple les fonctions d'affichage (`printf`) et de lecture (`scanf`) sont définies dans `stdio`. Si on ne met pas `#include<stdio.h>`, on ne peut pas utiliser ces fonctions.

Finalement, les déclarations et les instructions sont regroupées entre les accolades qui suivent `int main()`, d'abord les déclarations, puis les instructions.

`main` est la fonction principale, c'est-à-dire que c'est elle qui est exécutée quand le programme sera lancé.

Les commentaires sont ignorés par le compilateur.

Structure d'un programme en C

- ▶ La fonction main : Elle constitue le programme principal

```
main()  
{  
    déclaration des variables  
    instructions  
}
```

- ▶ Les fonctions :

```
Type_du_resultat Nom_fonction (Type_param Nom_param,...)  
{  
    déclaration des variables locales  
    instructions  
}
```

- ▶ Les commentaires : Un commentaire commence toujours par les deux symboles /* et se termine par les deux symboles */.
- ▶ Les variables : Type_variable Nom_variable;

Structure d'un programme en C

Le classique "Hello word" (affiche Hello word à l'écran). Pour le faire, il faut :

- ▶ Inclure les bibliothèques
- ▶ Inclure le main

```
main()
{
    déclaration des variables : aucune
    instruction : écrire "bonjour"
}
```

La fonction prédéfinie qui permet d'écrire à l'écran est `printf`, elle est contenue dans le fichier en-tête `stdio.h`; sa syntaxe est :

```
printf("ce que l'on veut écrire");
```

Structure d'un programme en C

Voici donc le programme :

```
#include <stdio.h>
int main()
{
    printf("bonjour\n");
    /*toute instruction se termine par un point virgule*/
}
```

Séquences d'échappement :

\t : tabulation (horizontale)

\v : tabulation (verticale)

\n : nouvelle ligne

\b : batch (curseur arrière)

\r : retour (retour au début de ligne, sans saut de ligne :
retour chariot)

\f : saut de page

\a : attention (signal acoustique)

Structure d'un programme en C – Les variables

Une variable est une entité qui contient une information, elle possède :

- ▶ Un nom (identifiant)
- ▶ Une valeur
- ▶ Un type qui caractérise l'ensemble des valeurs que peut prendre la variable

L'ensemble des variables est stocké dans la mémoire de l'ordinateur.

Pour déclarer une variable en C, on commence par mettre le type suivi du nom de la variable, éventuellement sa valeur et un point-virgule.

Type Nom; ou Type Nom = Valeur; ou Type Nom(Valeur);

On peut déclarer dans une instruction plusieurs variables d'un même type

type var1, var2, var3, ...;

Exp : int a, b, c;

On peut initialiser une variable lors de sa déclaration :

type var = valeur; ou type var(valeur);

Exp : float x = 12.65;

Structure d'un programme en C – Les variables

| Déclaration | Type | Nom | Valeur |
|-----------------------------------|--------|-------------|-------------------------------|
| <code>int quantite;</code> | Entier | quantite | Non définie (non initialisée) |
| <code>int nbre_eleves(14);</code> | Entier | nbre_eleves | 14 |
| <code>int x = 20;</code> | Entier | x | 20 |
| <code>int a, b, c;</code> | Entier | a, b, c | Non définie (non initialisée) |
| <code>double note;</code> | Réel | note | Non définie (non initialisée) |
| <code>double y = 1.2, z;</code> | Réel | y, z | 1.2 pour y, z non initialisée |

Table 1: Déclaration de variables

Structure d'un programme en C – Les variables

En mathématiques, on distingue divers ensembles de nombres (entiers naturels, entiers relatifs, réels, complexes ...). L'ordre de grandeur des nombres est illimité, ils peuvent être exprimés sans perte de précision. Un ordinateur utilise le système binaire pour sauvegarder et calculer les nombres, il existe pour un ordinateur deux grands systèmes de nombres: les entiers et les rationnels.

Table 2: Les entiers.

| Type | Signification | Intervalle | Taille |
|----------------|-----------------|---------------------------|----------|
| char | caractère | -128 à +127 | 1 Octet |
| short | entier court | -32768 à +32767 | 2 Octets |
| int | entier standard | -32768 à +32767 | 2 Octets |
| long | entier long | -2147483648 à +2147483647 | 4 Octets |
| unsigned char | caractère | 0 à +255 | 1 Octet |
| unsigned short | entier court | 0 à +65535 | 2 Octets |
| unsigned int | entier standard | 0 à +65535 | 2 Octets |
| unsigned long | entier long | 0 à +4294967295 | 4 Octets |

Structure d'un programme en C – Les variables

Table 3: Les rationnels.

| Type | Mantisse ¹ | Intervalle | Taille |
|-------------|-----------------------|---|-----------|
| float | 6 | 3.4×10^{-38} à $3.4 \times 10^{+38}$ | 4 Octets |
| double | 15 | 1.7×10^{-308} à $1.7 \times 10^{+308}$ | 8 Octets |
| long double | 19 | 3.4×10^{-4932} à $1.1 \times 10^{+4932}$ | 10 Octets |

¹ nombre de chiffres significatifs après la virgule

| Type | Syntaxe |
|-------------------------------------|---------|
| Booléen (vrai ou faux) | bool |
| Caractère (alphabétique, numérique) | char |

Table 4: Autres types de variables

Structure d'un programme en C – Les instructions

Table 5: L'opérateur d'affectation.

| Opérateur | Signification | Instruction | Exemples |
|-----------|---------------|---------------------------|--|
| = | Affectation | Non_variable = expression | x = 3.14; lettre = 'c'; y = a; z = a+2*b; a = a-b; |

Par exemple l'expression (l'instruction)

c = **a** + **b**;

se comprend de la façon suivante :

- ▶ On prend la valeur contenue dans la variable a
- ▶ On prend la valeur contenue dans la variable b
- ▶ On additionne ces deux valeurs
- ▶ On met ce résultat dans la variable c (si c avait auparavant une valeur, cette dernière est perdue (écrasée)).

Structure d'un programme en C : Opérateurs arithmétiques

| Opération | Symbole (opérateur) |
|---|---------------------|
| Addition | + |
| Soustraction | - |
| Multiplication | * |
| Division | / |
| Modulo (reste de la division euclidienne) | % |

Table 6: Opérateurs arithmétiques

Modulo (%)

L'opérateur % (exp : $a \% b$) renvoie, dans cet exemple, le reste de la division euclidienne de a par b .

Cette opérateur est utilisé uniquement sur des entiers.

Dans la division euclidienne de 7 par 3, le quotient est 2 et le reste est 1 ($7 = 2 \times 3 + 1$). Donc $7 \% 3$ renvoie 1.

Structure d'un programme en C : Opérateurs arithmétiques

Les raccourcis

| Opération | Raccourci |
|-------------------------|---|
| <code>a = a + b;</code> | <code>a += b;</code> |
| <code>a = a - b;</code> | <code>a -= b;</code> |
| <code>a = a * b</code> | <code>a *= b;</code> |
| <code>a = a / b;</code> | <code>a /= b;</code> |
| <code>a = a % b;</code> | <code>a %= b;</code> si a et b sont des entiers (a mod b) |
| <code>a = a + 1;</code> | <code>a++;</code> si a est un entier |
| <code>a = a - 1;</code> | <code>a--;</code> si a est un entier |

Table 7: Les raccourcis

Une variable peut être de type nombre, caractère, tableau, vecteur ...

- ▶ le nom de la variable doit être constitué uniquement de lettres, de chiffres et du tiret-bas "_"
- ▶ le premier caractère doit être une lettre
- ▶ on n'utilise pas d'accents dans le nom de la variable
- ▶ on n'utilise pas d'espaces dans le nom de la variable

Structure d'un programme en C - Opérateurs arithmétiques

Table 8: Les opérateurs d'incrément et de décrémentation.

| | |
|-------------|----------------|
| $i = i + 1$ | $i++$ ou $++i$ |
| $i = i - 1$ | $i--$ ou $--i$ |

- (a) Pour incrémenter ou décrémentation une variable, par exemple dans une boucle (dans ce cas, pas de différence entre la notation préfixe ($++i$, $--i$) et la notation postfixe ($i++$, $i--$)).
- (b) Pour incrémenter ou décrémentation une variable et en même temps affecter sa valeur à une autre variable. Dans ce cas, on choisit entre la notation préfixe et postfixe :

Structure d'un programme en C - Opérateurs arithmétiques

Table 9: Affectation avec les opérateurs d'incrément et de décrémentation.

| | |
|-----------|--|
| $x = i++$ | Passe d'abord la valeur de i à x , puis incrémente i (le $++$ est après i , on l'incrémente après) |
| $x = i--$ | Passe d'abord la valeur de i à x , puis décrémente i |
| $x = ++i$ | Incrémente d'abord i puis passe la valeur de i incrémentée à x (le $++$ est avant i , on l'incrémente avant) |
| $x = --i$ | Décrémente d'abord i puis passe la valeur de i décrémentée à x |

C : Opérateurs logiques et relationnels

| Opérateur | Symbole |
|-----------|---------|
| AND | && |
| OR | |
| NOT | ! |

Table 10: Opérateurs logiques en C

| Relation | Symbole |
|--------------------------------|-------------------------------|
| Egal à | == (mais pas = : affectation) |
| Différent de (\neq) | != |
| Supérieur à | > |
| Supérieur ou égal à (\geq) | >= |
| Inférieur à | < |
| Inférieur ou égal à (\leq) | <= |

Table 11: Opérateurs relationnels en C++

C : Opérateurs logiques et relationnels

Les résultats des opérations logiques sont de type int :

La valeur 0 correspond au booléen FAUX et
la valeur 1 correspond au booléen VRAI.

Remarque

Les opérateurs logiques considèrent toute valeur différente de 0 comme VRAI, toute valeur nulle comme FAUX.

- `32 && 2.3` → 1
- `!65.4` → 0
- `0 | (32>5)` → 1

C : Notion de condition

Une condition est une expression dont la valeur est :

- ▶ soit vraie
 - ▶ soit fausse
- (expression booléenne ou expression de type booléen).

Exemples :

$x > 1$

$x \neq 0$

$x \leq 10$

Structure d'un programme en C – Lecture et écriture des données

La bibliothèque standard `<stdio.h>` contient un ensemble de fonctions qui assurent la communication de la machine avec le monde extérieur. Les fonctions les plus importantes sont :

Pour la lecture :

- ▶ `printf()` : écriture formatée de données
- ▶ `putchar()` : écriture d'un caractère

Pour l'écriture :

- ▶ `scanf()` : lecture formatée de données
- ▶ `getchar()` : lecture d'un caractère

Structure d'un programme en C - Ecriture formatée

`printf` : cette fonction est utilisée pour transférer du texte, des valeurs de variables ou des résultats d'expression vers le fichier de sortie standard stdout (par défaut l'écran).

```
printf("format", expr1, expr2);
```

- ▶ `format` (format de représentation) : est une chaîne de caractères qui peut contenir :
 - du texte
 - des séquences d'échappement (Voir page 15)
 - des spécificateurs de format (un spécificateur pour chaque expression)

Structure d'un programme en C - Ecriture formatée

Table 12: Les spécificateurs de format : ils commencent toujours avec le symbole %.

| Symbole | Impression comme | Type ^{1,2} |
|----------|---|---------------------|
| %d ou %i | Entier relatif | int |
| %u | Entier naturel (non signé) | int |
| %o | Entier exprimé en octal | int |
| %x | Entier exprimé en hexadécimal | int |
| %f | Rationnel en notation décimale | float |
| %e | Rationnel en notation scientifique | float |
| %g | Rationnel en notation décimale/scientifique | float |
| %lf | Rationnel en notation décimale | double |
| %le | Rationnel en notation scientifique | double |
| %lg | Rationnel en notation décimale/scientifique | double |
| %c | Caractère | char |
| %s | Chaîne de caractères | char* |

¹ Entier long : on utilise %ld, %li, %lu, %lo, %lx

² Rationnel : long double, on utilise %Lf, %Le, %Lg

Structure d'un programme en C - Lecture formatée

`scanf` : La fonction `scanf` reçoit ses données à partir du fichier standard `stdin` (le clavier).

Syntaxe : `scanf("format", adrvar1, adrvar1);`

- ▶ `format` (format de données)
 - La chaîne de format détermine comment les données reçues doivent être interprétées
- ▶ Les données reçues correctement sont mémorisées aux adresses indiquées par `adrvar1, adrvar1 ...`

Exp : `scanf("On entre au clavier 12 \Rightarrow nombre = 12.`

On peut traiter plusieurs variables avec une seule instruction `scanf` : Lors de l'entrée des données, une suite de signes d'espacement (espace, tab, interligne) est évaluée comme un seul espace.

Structure d'un programme en C - Lecture formatée

Table 13: Les spécificateurs de format pour scanf.

| Symbole | Lecture d'un(e) | Type ^{1,2} |
|----------|---|---------------------|
| %d ou %i | Entier relatif | int |
| %u | Entier naturel (non signé) | int |
| %o | Entier exprimé en octal | int |
| %x | Entier exprimé en hexadécimal | int |
| %f | Rationnel en notation décimale | float |
| %e | Rationnel en notation scientifique | float |
| %g | Rationnel en notation décimale/scientifique | float |
| %lf | Rationnel en notation décimale | double |
| %le | Rationnel en notation scientifique | double |
| %lg | Rationnel en notation décimale/scientifique | double |
| %c | Caractère | char |
| %s | Chaîne de caractères | char* |

¹ Entier long : on utilise %ld, %li, %lu, %lo, %lx

² Rationnel : long double, on utilise %Lf, %Le, %Lg

Structure d'un programme en C - Ecriture d'un caractère

Syntaxe : `putchar(caractere);`

`putchar` transfère le caractère "caractere" vers le fichier de sortie standard `stdout` (l'écran), les arguments de `putchar` sont des caractères (type `char`, i.e. des nombres entiers entre 0 et 255).

| Instructions | Affichage |
|----------------------------|---|
| <code>putchar('x');</code> | x |
| <code>putchar('?');</code> | ? |
| <code>putchar(65);</code> | A (Voir la table ASCII) |
| <code>putchar(A);</code> | Valeur de la variable A si c'est un caractère |

Structure d'un programme en C - Lecture d'un caractère

Syntaxe : `getchar()`;

Les valeurs retournées par `getchar()` sont des caractères. Le type du résultat de `getchar` est `int`.

```
int C;  
C = getchar();
```

`getchar` lit les données de la zone tampon `stdin` (clavier) et fournit les données seulement après confirmation par "Enter".

Il existe dans la bibliothèque `<conio.h>` une fonction `getch()` qui fournit immédiatement le prochain caractère entré au clavier (sans validation).

Structures de contrôle – Instructions conditionnelles *if*

```
if (condition)
    une seule instruction;
```

```
if (condition)
{
    instruction 1;
    ...
    instruction n;
}
```

La deuxième forme est préférable car dans la première, on ne peut mettre qu'une seule instruction contrôlée par le *if* alors que dans la deuxième, on peut mettre autant d'instructions qu'on veut grâce aux accolades.

Structures de contrôle – Instructions conditionnelles *if...else*

```
if (condition)
{
    instruction 1.1;
    instruction 1.2;
    ...
}
else
{
    instruction 2.1;
    instruction 2.2;
    ...
}
```

L'opérateur conditionnel

`resultat = expr1 ? expr2 : expr3;`

Si `expr1` est vraie, `resultat = expr2`, sinon `resultat = expr3`.

Structures de contrôle – L'instruction switch

L'instruction `switch` est une instruction multidirectionnelle utilisée pour gérer les décisions. Cela fonctionne presque exactement comme la déclaration `if-else`. La différence est que l'instruction `switch` génère un code plus lisible par rapport à l'instruction `if-else`. De plus, elle s'exécute parfois plus rapidement que `if-else`.

Structures de contrôle – L'instruction switch

"expression" dans l'instruction `switch` peut être toute expression valide qui donne une valeur entière. L'expression peut également être un caractère (car tous les caractères sont finalement convertis en un entier avant toute opération), mais il ne peut s'agir ni de virgule flottante (`float`, `double`) ni de chaîne.

`val1`, `val2` ... après le mot clé "`case`" doit être de type entier (comme `int`, `long int` ...) ou de type caractère. Elle peut aussi être une expression qui donne une valeur entière. Chaque cas doit avoir une seule valeur. Les valeurs multiples dans la déclaration `case` ne sont pas autorisées. de plus, toutes les valeurs doivent être uniques.

```
switch(expression)
{
    case val1:
        instruction1;
        instruction2;
        ...
        [break;]
    case val2:
        instruction3;
        instruction4;
        ...
        [break;]
    case val3:
        instruction5;
        instruction6;
        ...
        [break;]
    default:
        instruction7;
        ...
}
```

Structures de contrôle – L'instruction switch

Exemple

```
#include <stdio.h>

int main(void){
    int jour;

    printf("saisir le numéro du jour : ");
    scanf("%d",&jour);

    switch(jour){
        case 1 : printf("Lundi");
        case 2 : printf("Mardi");
        case 3 : printf("Mercredi");
        case 4 : printf("Jeudi");
        case 5 : printf("Vendredi");
        case 6 : printf("Samedi");
        case 7 : printf("Dimanche");
        default: printf("jour invalide");
    }
    return 0;
}
```

Les boucles

Il arrive souvent dans un programme qu'une même action (instruction) soit répétée plusieurs fois, avec éventuellement quelques variantes. Il est alors fastidieux d'écrire un algorithme qui contient de nombreuses fois la même instruction. Pour gérer ces cas, on fait appel à des instructions en boucle qui ont pour effet de répéter plusieurs fois une même instruction.

Deux formes existent :

- ▶ la première, si le nombre de répétitions est connu avant l'exécution de l'instruction de répétition (On utilise la boucle for),
- ▶ la seconde s'il n'est pas connu (on utilise la boucle while).

L'exécution de la liste des instructions se nomme itération.

Les boucles – La boucle *for*

```
for (expr1; expr2; expr3)
{
    Instructions;
}
```

- ▶ **expr1** est évaluée une fois avant le passage dans la boucle, elle est utilisée pour initialiser les données de la boucle.
- ▶ **expr2** est évaluée à chaque passage de la boucle, elle est utilisée pour savoir si la boucle est répétée ou non (c'est une condition de répétition, et non d'arrêt).
- ▶ **exp3** est évaluée à la fin de chaque passage de la boucle, elle est utilisée pour réinitialiser les données de la boucle.

```
for (int i=0; i<5; i++)
{
    instructions;
}
```

Les boucles – La boucle *while*

```
while (i<5)
{
    instructions;
    i++;
}
```

Les tableaux

Pour déclarer un tableau à une dimension, on utilise l'instruction :

```
Type Nom[Taille];
```

Ou

```
Type Nom[Taille]{Valeur1, Valeur2, ...};
```

Exemple :

```
int mon_tableau[6]{134, 46, 12, 20, 51, 17};
```

| Rang (indice) | 0 | 1 | 2 | 3 | 4 | 5 |
|---------------|-----|----|----|----|----|----|
| Valeur | 134 | 46 | 12 | 20 | 51 | 17 |

- ▶ Ce tableau est de longueur 6 car il contient 6 emplacements (cases).
- ▶ Chacun des éléments du tableau est repéré par son rang (appelé indice).
- ▶ Pour accéder à un élément du tableau, il suffit de préciser entre crochets l'indice de la case contenant cet élément :
 - Pour accéder au troisième élément (12), on écrit `mon_tableau[2]` ;
 - `x = mon_tableau[1]` ; La variable `x` prend la valeur du deuxième élément du tableau (46).
 - `mon_tableau[4] = 79` ; on remplace la valeur du cinquième élément (51) par 79.

Les tableaux

Déclaration : `Type(simple) NomDuTableau[Dimension];`

Exemples :

- ▶ `int Tableau1[6];` Déclaration d'un tableau nommé `Tableau1`, de type `int` et de dimension 6
- ▶ `char MonTableau[50];` Déclaration d'un tableau nommé `MonTableau`, de type `char` et de dimension 50
- ▶ `float Table[12];` Déclaration d'un tableau nommé `Table`, de type `float` et de dimension 12

Lors de la déclaration d'un tableau, on peut initialiser ses composantes en indiquant la liste des valeurs entre accolades

```
int B[5] = {10, 20, 30, 40, 50};
```

On peut accéder aux composantes du tableau par :

```
B[0], B[1], ..., B[4]
```

Les tableaux

La boucle `for` se prête particulièrement bien pour accéder et afficher les éléments du tableau.

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

Affectation avec des valeurs saisies au clavier :

```
int main()
{
    int B[5];
    for (int i=0; i<5; i++)
    {
        scanf("%d", &B[i]);
    }
    return 0;
}
```

Les tableaux

Pour déclarer un tableau à deux dimensions, on utilise l'instruction :

```
Type Nom[Nombre_de_lignes] [Nombre_de_colonnes];
```

Ou

```
Type Nom[Nombre_de_lignes] [Nombre_de_colonnes] {{Val1,  
Val2, ...},{...},{...},...};
```

Exemple :

```
double table[3][2]{{5.1, 3.6}, {3.14, 0.25}, {10.3, 7.5}};
```

| Indices | 0 | 1 |
|---------|------|------|
| 0 | 5.1 | 3.6 |
| 1 | 3.14 | 0.25 |
| 2 | 10.3 | 7.5 |

- ▶ Ce tableau contient 3 lignes et 2 colonnes.
- ▶ Les éléments du tableau sont repérés par leur numéro de ligne et leur numéro de colonne :
 - `Y = table[1][0]` ; La variable y prend la valeur située à la deuxième ligne et à la première colonne, c-à-d, 3.14

Les tableaux

Les tableaux à deux dimensions

Déclaration :

Type(simple) NomTableau[DimensionLigne][DimensionColonne];

Exemples :

- ▶ `double A[2][5];` Déclaration d'un tableau à deux dimensions nommé A et de type `double`
- ▶ `char B[4][2];` Déclaration d'un tableau à deux dimensions nommé B et de type `char`

Les composantes d'un tableau à deux dimensions sont stockées ligne par ligne dans la mémoire.

```
short A[2][3] = {{1,3,5},  
                 {10,30,50}};
```

Pour accéder à un élément du tableau :

```
NomTableau[Ligne][colonne];
```

Les tableaux

Les tableaux à deux dimensions

Exemple :

```
int main()
{
    int A[5][10];
    int i, j;
    for (i=0; i<5; i++)
    {
        for (j=0; j<10; j++)
        {
            printf("%7d \n", A[i][j]);
        }
    }
    return 0;
}
```


Les chaînes de caractères

Il n'existe pas de type spécial chaîne ou string en C. Une chaîne de caractères est traitée comme un tableau à une dimension de caractères. La déclaration d'une chaîne de caractères se fait avec l'une des méthodes suivantes :

- ▶ `char NomChaine[Longueur];`
- ▶ `char *NomChaine;`

Exemple :

```
char Nom[26]; //Déclaration sans initialisation
char Prenom[]="Abcd"; //Déclaration avec initialisation
```

Les chaînes de caractères

Les fonctions de stdio.h

```
char Texte[6] = "Hello";  
char CH[5];  
printf("%s",Texte); // Affiche Hello sans retour à la ligne  
printf("%s",CH); // CH contient l'adresse du premier caractère de la  
    chaîne  
puts(Texte); // Affiche Hello avec un retour à la ligne  
puts("Bonjour"); // Affiche Bonjour avec un retour à la ligne  
gets(CH); // Lit une ligne de caractères de stdin
```

Les chaînes de caractères

Les fonctions de string.h

```
char CH1[] = "Hello";  
char CH2[] = "Bonjour";  
char *chaine = "Bonsoir";  
int k = 2;  
char c = 'a';  
strlen(CH1); // Fournit la longueur de CH1  
strcpy(CH1, CH2); // Copie CH1 dans CH2  
strcat(CH1, CH2); // Ajoute CH2 à la fin de CH1  
strcmp(CH1, CH2); // Compare CH1 et CH2  
strcpy(CH1, "Bonsoir");
```

Les chaînes de caractères

Les fonctions de stdlib.h

```
char CH[] = "125";  
atoi(CH); // Retourne la valeur numérique représentée par CH comme  
            int  
atol(CH); // Retourne la valeur numérique représentée par CH comme  
            long  
atof(CH); // Retourne la valeur numérique représentée par CH comme  
            double
```

Les chaînes de caractères

Les fonctions de ctype.h

```
// c représente une valeur de type int qui peut être représentée
// comme caractère.
isupper(c); // Retourne une valeur différente de 0 si c'est une
// majuscule
islower(c); // Retourne une valeur différente de 0 si c'est une
// minuscule
isspace(c); // Retourne une valeur différente de 0 si c'est un signe
// d'espacement
tolower(c) ; // Retourne c converti en minuscule si c'est une
// majuscule
toupper(c) ; // Retourne c converti en majuscule si c'est une
// minuscule
```

Les chaînes de caractères

Tableaux de chaîne de caractères

```
// Déclaration
char JOUR[7][9];
char *mois[12];

// Initialisation
char JOUR[7][9] = {"Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche"};
char *mois[12] = {"Jan", "Fev", "Mars", "Avr", "Mai", "Juin", "Juil", "Aout", "Sep", "Oct", "Nov", "Dec"};

// Accès aux différents éléments
printf("%s", JOUR[2]); // Affiche Mercredi
printf("%s", mois[0]); // Affiche Jan

// Accès aux caractères
for (int i=0; i<7; i++)
    printf("%c", JOUR[1][0]);

/* Affectation :
 * Pas d'affectation directe (JOUR[2]="Jeudi");.
 * On utilise : */
strcpy(JOUR[2], "Jeudi");
```

Les fonctions

Les fonctions ont la forme suivante :

```
type nom_de_la_fonction(arguments)
{
    //Instructions effectuées par la fonction
}
```

- ▶ **type** : indique le type de variable renvoyée par la fonction (string, int, double ...)
- ▶ **nom_de_la_fonction** : permet de donner un nom à la fonction
- ▶ **arguments** : les différentes variables d'entrée de la fonction

Exemple :

```
int multiplicationDix(int nbre)
{
    int res(nbre * 10);

    return res;
}
```

Les fonctions - Les fonctions récursives

Une fonction récursive est une fonction qui pour fournir un résultat, s'appelle elle-même un certain nombre de fois.

Exemple

La formule de calcul de la factorielle d'un nombre entier est donnée par :

$$n! = 1 * 2 * 3 * \dots * (n-1) * n$$

Ce calcul peut être réalisé avec une boucle *for*.

Une autre manière de réaliser ce calcul, serait de dire que :

$$n! = n * (n-1)!$$

c.à.d. la factorielle d'un nombre, c'est ce nombre multiplié par la factorielle du nombre précédent. Pour coder cela, il faut une fonction *Facto*, chargée de calculer la factorielle. Cette fonction effectue la multiplication du nombre passé en argument par la factorielle du nombre précédent.

Pour terminer, il manque la condition d'arrêt de ces auto-appels de la fonction *Facto*. On s'arrête quand on arrive au nombre 1, pour lequel la factorielle est par définition 1.

Les fonctions - Les fonctions récursives

```
Fonction Facto (n : Entier)
Si n = 1 alors
    Renvoyer 1
Sinon
    Renvoyer Facto(n - 1) * n
```

Le processus récursif remplace en quelque sorte la boucle, c'est-à-dire un processus itératif.

Les structures

Une structure est une suite finie d'objets de types différents.

Contrairement aux tableaux, les différents éléments d'une structure n'occupent pas nécessairement des zones contiguës en mémoire. Chaque élément de la structure, appelé membre ou champ, est désigné par un identificateur.

On distingue la déclaration d'un modèle de structure de celle d'un objet de type structure correspondant à un modèle donné. La déclaration d'un modèle de structure dont l'identificateur est `modele` suit la syntaxe suivante :

```
struct modele
{
    type-1 membre-1;
    type-2 membre-2;
    ...
    type-n membre-n;
};
```

Les structures

Pour déclarer un objet de type structure correspondant au modèle précédent, on utilise la syntaxe : `struct modele objet;`
Ou bien, si le modèle n'a pas été déclaré au préalable :

```
struct modele
{
    type-1 membre-1;
    type-2 membre-2;
    ...
    type-n membre-n;
} objet;
```

On accède aux différents membres d'une structure grâce à l'opérateur membre de structure, noté ".". Le i-ème membre de objet est désigné par l'expression `objet.membre-i`

Les structures

On peut effectuer sur le i -ème membre de la structure toutes les opérations valides sur des données de type `type-i`. Par exemple, le programme suivant définit la structure `complexe`, composée de deux champs de type `double` ; il calcule la norme d'un nombre complexe.

```
#include <stdio.h>
#include <math.h>
struct complexe
{
    double reelle;
    double imaginaire;
};

int main()
{
    struct complexe z = {3. , 4.};
    double norme;
    norme = sqrt(z.reelle * z.reelle + z.imaginaire * z.imaginaire);
    printf("norme de (%f + i %f) = %f \n", z.reelle, z.imaginaire,
        norme);
    return 0;
}
```

Les énumérations

Les énumérations permettent de définir un type par la liste des valeurs qu'il peut prendre. Un objet de type énumération est défini par le mot-clef `enum` et un identificateur de modèle, suivis de la liste des valeurs que peut prendre cet objet : `enum modele constante-1, constante-2,...,constante-n`};


En réalité, les objets de type `enum` sont représentés comme des `int`. Les valeurs possibles `constante-1`, `constante-2`,...,`constante-n` sont codées par des entiers de 0 à $n-1$. Par exemple, le type `enum` `booléen` défini dans le programme suivant associe l'entier 0 à la valeur "faux" et l'entier 1 à la valeur "vrai".

```
#include <stdio.h>


int main()
{
    enum booléen {faux, vrai};
    enum booléen b;
    b = vrai;
    printf("b = %d\n",b);
}
```

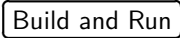
Compilation — A partir de l'IDE

Lancement de la phase de compilation

Pour compiler le fichier source, utilisez l'icône . Si vous n'avez pas d'erreur, vous pouvez passer à la phase suivante. Sinon, corrigez les erreurs . . .

Exécution du programme

■ L'exécution n'est possible que si la compilation a été faite sans erreurs. Les messages d'erreurs de compilation permettent de corriger les fautes de syntaxe. Corrigez les erreurs et relancez jusqu'à obtenir une compilation sans erreurs. Les warnings sont de simples avertissements et n'empêchent pas l'exécution. Faites attention néanmoins à ces warnings. Le lancement de l'exécution se fait par l'icône .

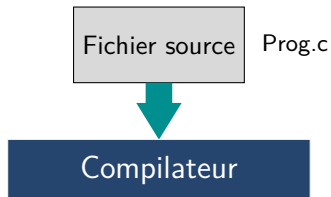
■ **Remarque :** l'icône  (compiler et exécuter) lance la compilation et l'exécution par la suite sauf en cas d'erreur de compilation bien sûr.

Dans la section Build log, l'IDE affiche quelques messages en bas de l'IDE. Et si tout va bien, une console apparaît avec le résultat de l'exécution du programme.

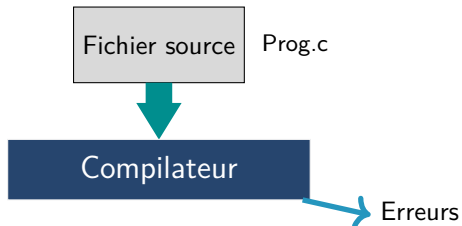
Compilation – Les étapes de la compilation

Fichier source Prog.c

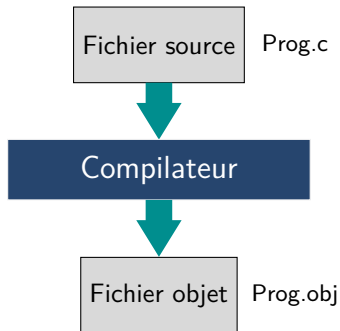
Compilation — Les étapes de la compilation



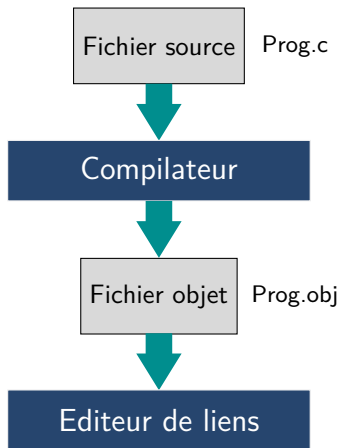
Compilation — Les étapes de la compilation



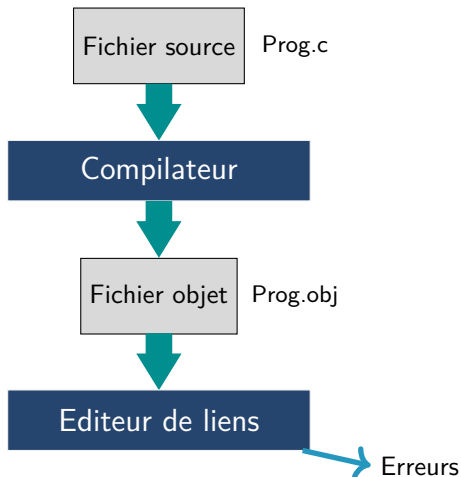
Compilation – Les étapes de la compilation



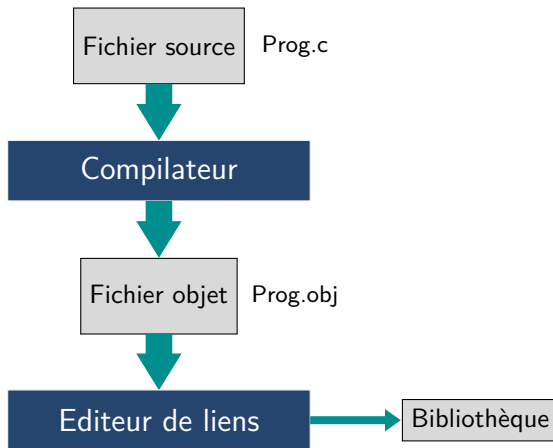
Compilation — Les étapes de la compilation



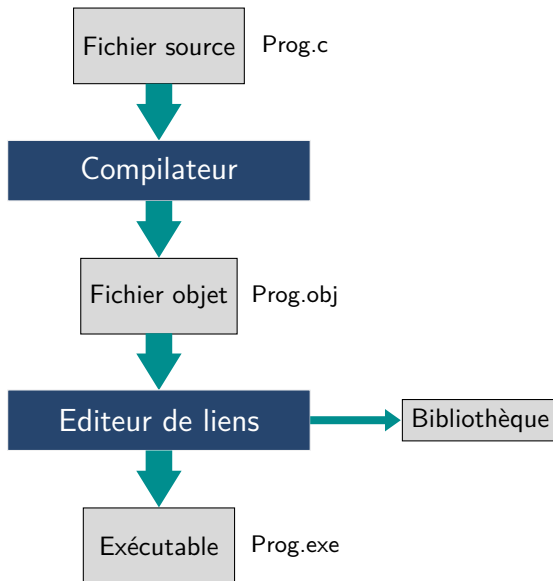
Compilation – Les étapes de la compilation



Compilation – Les étapes de la compilation



Compilation – Les étapes de la compilation

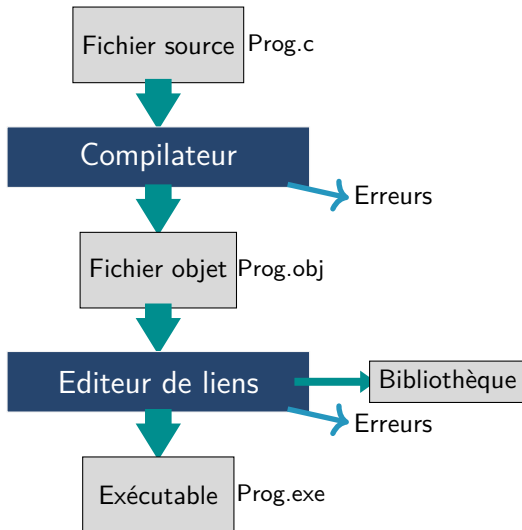


Compilation – Les étapes de la compilation

La compilation consiste en une série d'étapes de transformation du code source en du code machine exécutable sur un processeur cible.

Le langage C fait partie des langages compilés : le fichier exécutable est produit à partir de fichiers sources par un compilateur.

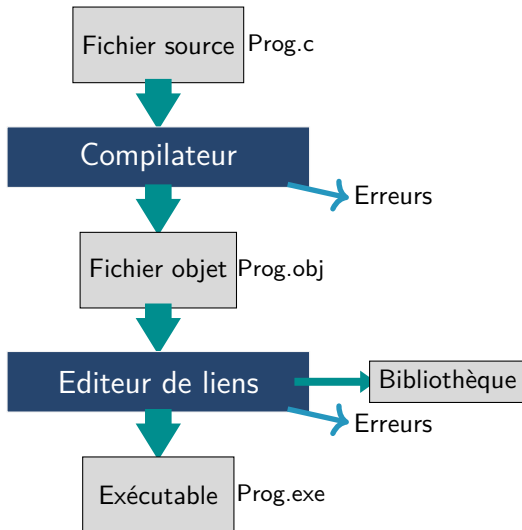
La compilation passe par différentes phases :



Compilation — Les étapes de la compilation

Le preprocessing :

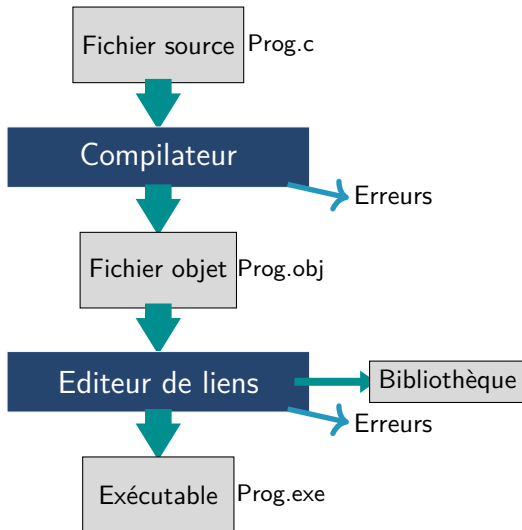
Le compilateur analyse le langage source afin de vérifier la syntaxe et de générer un code source brut (s'il y a des erreurs de syntaxe, le compilateur est incapable de générer le fichier objet). Les commentaires sont enlevés et les directives de compilation commençant par `#` sont d'abord traités pour obtenir le code source brut.



Compilation — Les étapes de la compilation

La compilation en fichier objet :

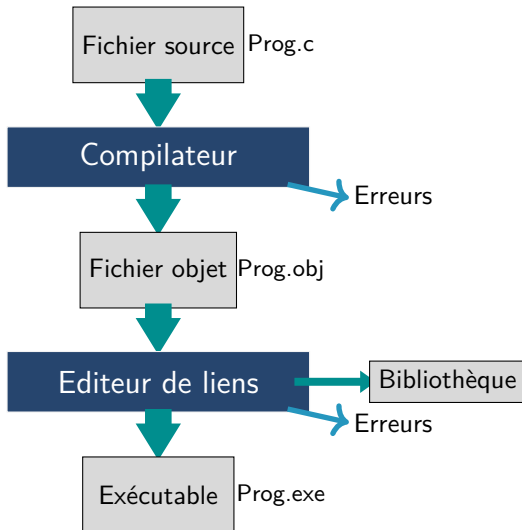
Les fichiers de code source brut sont transformés en un fichier dit objet, c'est-à-dire un fichier contenant du code machine ainsi que toutes les informations nécessaires pour l'étape suivante (édition de liens).



Compilation — Les étapes de la compilation

L'édition de liens :

L'éditeur de liens (linker) s'occupe d'assembler les fichiers objet en une entité exécutable et doit pour ce faire résoudre toutes les adresses non encore résolues. C'est à dire que lorsqu'il fait appel dans le fichier objet à des fonctions ou des variables externes, l'éditeur de liens recherche les objets ou bibliothèques concernés et génère l'exécutable (Il se produit une erreur lorsque l'éditeur de liens ne trouve pas ces références).



Compilation – Les étapes de la compilation

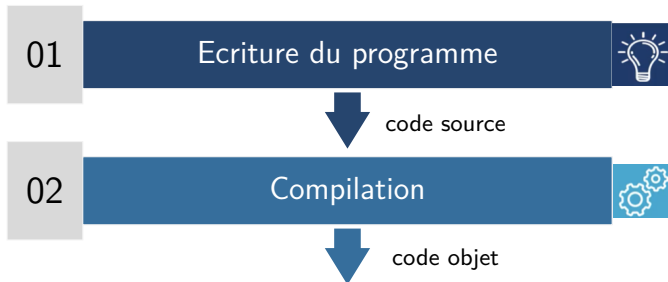
01

Ecriture du programme

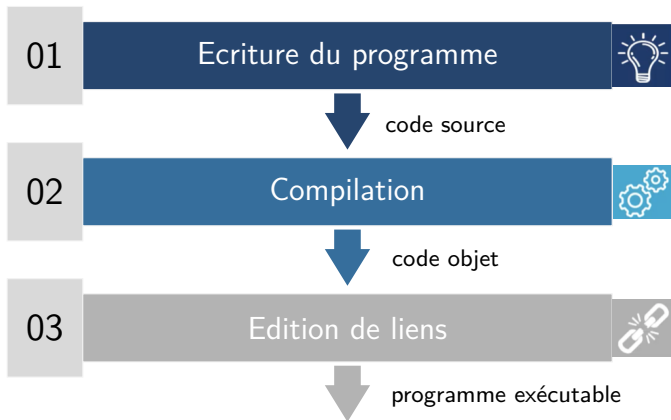


code source

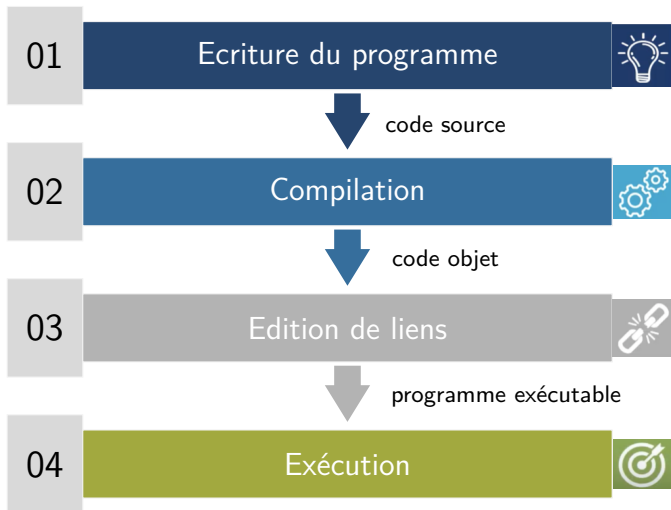
Compilation — Les étapes de la compilation



Compilation — Les étapes de la compilation



Compilation — Les étapes de la compilation



Compilation — Les étapes de la compilation

01

Ecriture du programme

Prog.c

Compilation — Les étapes de la compilation

01

Ecriture du programme

Prog.c

02

Compilation

```
gcc -o ProgObjet -c Prog.c
```


Compilation — Les étapes de la compilation

01

Ecriture du programme

Prog.c

02

Compilation

`gcc -o ProgObjet -c Prog.c`

03

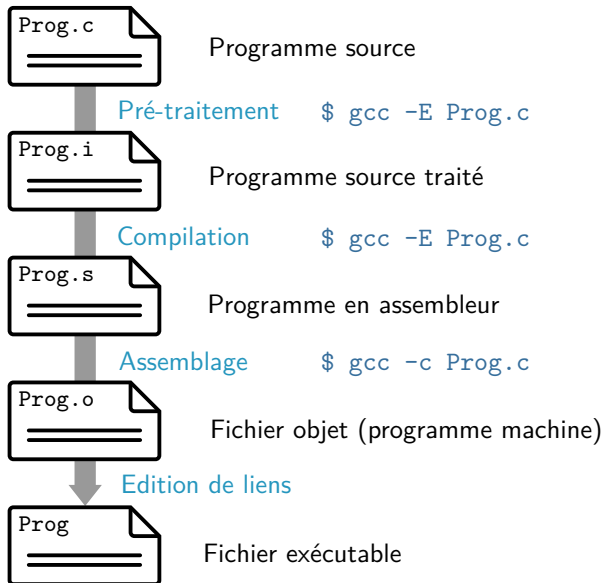
Edition de liens

`gcc -o ProgExe ProgObjet`

Compilation — Les étapes de la compilation

| | | |
|----|-----------------------|---|
| 01 | Ecriture du programme | Prog.c |
| 02 | Compilation | <code>gcc -o ProgObjet -c Prog.c</code> |
| 03 | Edition de liens | <code>gcc -o ProgExe ProgObjet</code> |
| 04 | Exécution | <code>./ProgExe</code> |

Compilation — Les étapes de la compilation



Compilation – Les étapes de la compilation

Table 14: Méthode 1 - Commandes pour exécuter un programme en C

| Commande ¹ | Type du fichier généré | Nom du fichier généré |
|---------------------------------|----------------------------|-----------------------|
| <code>gcc Prog.c -o Prog</code> | Exécutable | Prog |
| <code>./Prog</code> | Pour exécuter le programme | |

¹ Les commandes doivent être réalisées dans l'ordre.

Compilation – Les étapes de la compilation

Table 15: Méthode 2 - Commandes pour exécuter un programme en C

| Commande ² | Type du fichier généré | Nom du fichier généré |
|---|----------------------------|-----------------------|
| <code>gcc -o ProgObjet -c Prog.c</code> | Objet | ProgObjet |
| <code>gcc -o ProgExe ProgObjet</code> | Exécutable | ProgExe |
| <code>./ProgExe</code> | Pour exécuter le programme | |

² Les commandes doivent être réalisées dans l'ordre.

Compilation – Les étapes de la compilation

Table 16: Méthode 3 - Commandes pour exécuter un programme en C

| Commande ³ | Type du fichier généré | Nom du fichier généré |
|--|--|---------------------------------------|
| <code>gcc -save-temps Prog.c -o ProgExe</code> | PS traité Assembleur Objet Exécutable | Prog.i Prog.s Prog.o ProgExe |

| | |
|------------------------|----------------------------|
| <code>./ProgExe</code> | Pour exécuter le programme |
|------------------------|----------------------------|

³ Les commandes doivent être réalisées dans l'ordre.

Compilation – Les étapes de la compilation

Table 17: Méthode 4 - Commandes pour exécuter un programme en C

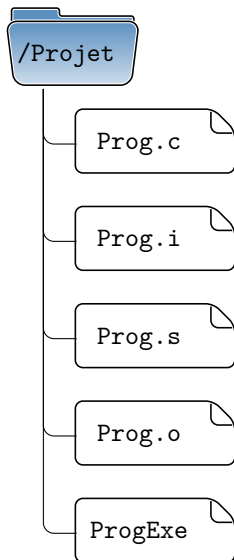
| Commande ⁴ | Type du fichier généré | Nom du fichier généré |
|---------------------------------|------------------------------------|-----------------------|
| <code>gcc -E Prog.c</code> | Affiche le programme source traité | |
| <code>gcc -c Prog.c</code> | Objet | Prog.o |
| <code>gcc -o Prog Prog.o</code> | Exécutable | Prog |
| <code>./Prog</code> | Pour exécuter le programme | |

⁴ Les commandes doivent être réalisées dans l'ordre.

Remarque :

Pour l'exécution des programmes C++, il suffit de remplacer `gcc` par `g++`

Compilation — Le contenu du dossier



Note : Fichiers générés en utilisant les commandes
du tableau 16

Compilation — Les fichiers générés

Fichiers générés en utilisant les commandes du tableau 16

Programme source : Prog.c

```
#include <stdio.h>
#define J 5

// Fonction main
int main()
{
    // Déclaration des variables
    int i = 10, k;
    k = i+J;
    printf("k = %d\n", k);
    return 0;
}
```

Taille : 170 octets

Compilation — Les fichiers générés

Programme source traité : Prog.i

```
# 1 "Prog.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 363 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "Prog.c" 2
# 1 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/stdio.h" 1 3 4
.
.
500 lignes plus bas
.
.
extern int __vsnprintf_chk (char * restrict, size_t, int, size_t,
    const char * restrict, va_list);
# 408 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/stdio.h" 2 3 4
# 2 "Prog.c" 2


int main()
{

    int i = 10, k;
    k = i+5;
    printf("k = %d\n", k);
    return 0;
}
```

Taille : 23 ko

Compilation — Les fichiers générés

Programme en assembleur : Prog.s

```
.section __TEXT,__text,regular,pure_instructions
.build_version macos, 10, 15, 4 sdk_version 10, 15, 4
.globl _main                ## -- Begin function main
.p2align 4, 0x90
_main:                      ## @main
.cfi_startproc
## %bb.0:
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq %rsp, %rbp
.cfi_def_cfa_register %rbp
subq $16, %rsp
movl $0, -4(%rbp)
movl $10, -8(%rbp)
movl -8(%rbp), %eax
addl $5, %eax
movl %eax, -12(%rbp)
movl -12(%rbp), %esi
leaq L_.str(%rip), %rdi
movb $0, %al
callq _printf
xorl %ecx, %ecx
movl %eax, -16(%rbp)        ## 4-byte Spill
movl %ecx, %eax
addq $16, %rsp
popq %rbp
retq
.cfi_endproc

                                ## -- End function

.section __TEXT,__cstring,cstring_literals
L_.str:                      ## @.str
.asciz "k = %d\n"

.subsections_via_symbols
```

Taille : 474 octets

Compilation — Les fichiers générés

Programme objet : Prog.o

```
cffa edfe 0700 0001 0300 0000 0100 0000
0400 0000 0802 0000 0020 0000 0000 0000
1900 0000 8801 0000 0000 0000 0000 0000
.
.
40 lignes plus bas
.
.
0000 0000 0100 0006 0100 0000 0f01 0000
0000 0000 0000 0000 0700 0000 0100 0000
0000 0000 0000 0000 005f 6d61 696e 005f
7072 696e 7466 0000
```

Taille : 792 octets

Compilation — Les fichiers générés

Programme Exécutable : ProgExe.out

```
cffa edfe 0700 0001 0300 0000 0200 0000
1000 0000 5805 0000 8500 2000 0000 0000
1900 0000 4800 0000 5f5f 5041 4745 5a45
524f 0000 0000 0000 0000 0000 0000 0000
```

.

.

800 lignes plus bas

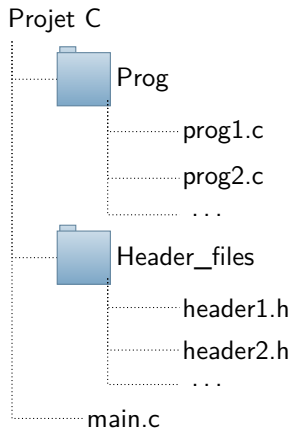
.

.

```
0300 0000 2000 5f5f 6d68 5f65 7865 6375
7465 5f68 6561 6465 7200 5f6d 6169 6e00
5f70 7269 6e74 6600 6479 6c64 5f73 7475
625f 6269 6e64 6572 005f 5f64 796c 645f
7072 6976 6174 6500 0000 0000
```

Taille : 13 ko

Organisation d'un projet — Exemple



La fonction main

La fonction principale main est une fonction comme les autres et de type int. Elle doit retourner un entier dont la valeur est transmise à l'environnement d'exécution. Cet entier indique si le programme s'est ou non déroulé sans erreur :

- ▶ La valeur de retour 0 correspond à une terminaison correcte
- ▶ toute valeur de retour non nulle correspond à une terminaison sur une erreur.

La fonction main peut également posséder des paramètres formels. En effet, un programme C ou C++ peut recevoir une liste d'arguments au lancement de son exécution. La ligne de commande qui sert à lancer le programme est, dans ce cas, composée du nom du fichier exécutable suivi par des paramètres. La fonction main reçoit tous ces éléments de la part de l'interpréteur de commandes.

La fonction main

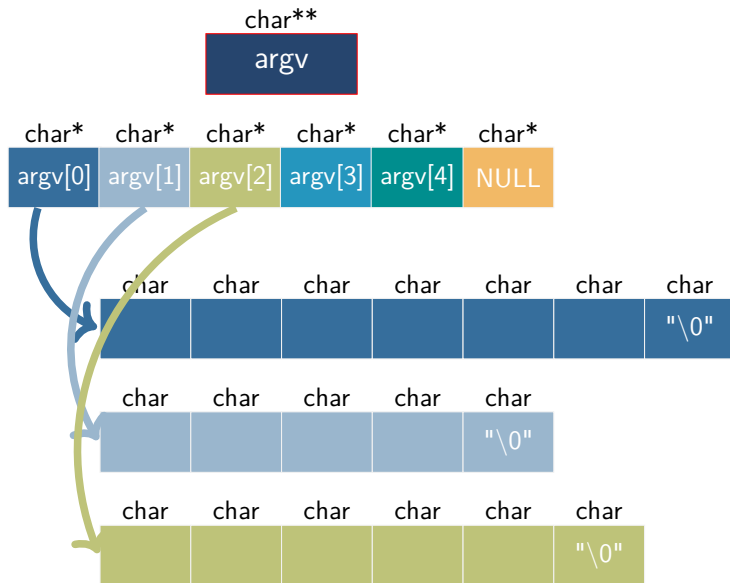
La fonction main possède deux paramètres formels, appelés `argc` (argument count) et `argv` (argument vector (values)).

- ▶ `argc` est une variable de type `int` dont la valeur est égale au nombre de mots composant la ligne de commande (y compris le nom de l'exécutable). Elle est donc égale au nombre de paramètres effectifs de la fonction +1.
- ▶ `argv` est un tableau de chaînes de caractères correspondant chacune à un mot de la ligne de commande. Le premier élément `argv[0]` contient donc le nom de la commande (du fichier exécutable), le second `argv[1]` contient le premier paramètre ...

Le second prototype valide de la fonction main est :

```
int main(int argc, char *argv[]);
```


La fonction main



La fonction main

Exemple

Ce programme calcule la somme de deux entiers, entrés en arguments de l'exécutable.

```
#include <stdio.h>
#include <stdlib.h> // Pour la fonction atoi

int main(int argc, char *argv[])
{
    int a, b;
    if (argc != 3)
    {
        printf("\nErreur : nombre invalide d'arguments");
        printf("\nUsage: %s int int\n", argv[0]);
        return(EXIT_FAILURE);
    }
    a = atoi(argv[1]); // prend en argument une chaîne de caractères
                       // et retourne l'entier dont elle est l'écriture décimale
    b = atoi(argv[2]);
    printf("\nLa somme de %d et %d vaut : %d\n", a, b, a + b);
    return(EXIT_SUCCESS);
}
```

Dans le terminal, taper (après compilation) : ./NomProExe 15 10

La console affiche : La somme de 15 et 10 vaut : 25

La fonction main

Exemple (Affichage des arguments)

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

int main(int argc, char *argv[])
{
    int a, b;
    if (argc != 3)
    {
        printf("\nErreur : nombre invalide d'arguments");
        printf("\nUsage: %s int int\n", argv[0]);
        return(EXIT_FAILURE);
    }
    a = atoi(argv[1]);
    b = atoi(argv[2]);
    printf("\nLa somme de %d et %d vaut : %d\n", a, b, a + b);
    int i;
    for (i=0; i < argc; i++)
        printf("Argument %d : %s\n", i+1, argv[i]);
    return(EXIT_SUCCESS);
}
```

La console affiche : (après la commande ./NomProExe 15 10)

La somme de 15 et 10 vaut : 25

Argument 1 : ./NomProExe

Argument 2 : 15

Argument 3 : 10

Les directives de préprocesseur

Le préprocesseur est un programme exécuté lors de la première phase de la compilation. Il effectue des modifications textuelles sur le fichier source à partir de directives. Les différentes directives au préprocesseur, introduites par le caractère `#`, ont pour but :

- ▶ l'incorporation de fichiers source (`#include`)
- ▶ la définition de constantes symboliques et de macros (`#define`)
- ▶ la compilation conditionnelle (`#if`, `#ifdef`, `#ifndef`, ...)

Les directives de préprocesseur — La directive `#include`

`#include`

Elle permet d'incorporer dans le fichier source, le texte figurant dans un autre fichier. Ce dernier peut être un fichier en-tête de la librairie standard (`iostream`, `cmath`, ...) ou n'importe quel autre fichier. La directive `#include` possède deux syntaxes voisines :

`#include <nom-de-fichier>` Recherche le fichier mentionné dans un ou plusieurs répertoires systèmes définis par l'implémentation (par exemple, `/usr/include/`).

`#include "nom-de-fichier"` Recherche le fichier dans le répertoire courant (celui où se trouve le fichier source). On peut spécifier d'autres répertoires à l'aide de l'option `-I` du compilateur.

La première syntaxe est généralement utilisée pour les fichiers en-tête de la librairie standard, tandis que la seconde est plutôt destinée aux fichiers créés par l'utilisateur.

Les directives de préprocesseur — La directive #define

La directive `#define` permet de définir :

- ▶ Des constantes symboliques
- ▶ Des macros avec paramètres

Remarque : `#undef` supprime une macro (constante) définie.

Définition des constantes symboliques

La directive `#define nom reste-de-la-ligne` demande au préprocesseur de substituer toute occurrence de `nom` par la chaîne de caractères `reste-de-la-ligne` dans la suite du fichier source. Son utilité principale est de donner un nom parlant à une constante, qui pourra être aisément modifiée.

```
#define NB_LIGNES 10
#define NB_COLONNES 6
#define TAILLE_TABLEAU NB_LIGNES*NB_COLONNES
```

Les directives de préprocesseur — La directive #define

Définition des macros

Une macro avec paramètres se définit de la manière suivante :

```
#define nom(liste-de-paramètres) corps-de-la-macro
```

Par exemple :

```
#define MAX(a,b) (a>b?a:b)
```

Le processeur remplacera dans la suite du code toutes les occurrences du type **MAX(x,y)** où *x* et *y* sont des symboles quelconques par exemple **(x>y?x:y)**

Une macro a donc une syntaxe similaire à celle d'une fonction, mais son emploi permet en général d'obtenir de meilleures performances en temps d'exécution.

Les directives de préprocesseur — La directive #define

La distinction entre une définition de constante symbolique et celle d'une macro avec paramètres se fait sur le caractère qui suit immédiatement le nom de la macro :

- ▶ si ce caractère est une parenthèse ouvrante, c'est une macro avec paramètres,
- ▶ sinon c'est une constante symbolique.

Il ne faut donc jamais mettre d'espace entre le nom de la macro et la parenthèse ouvrante.

A noter que le préprocesseur n'effectue que des remplacements de chaînes de caractères. En particulier, il est conseillé de toujours mettre entre parenthèses le corps de la macro et les paramètres formels qui y sont utilisés. Par exemple, si l'on écrit sans parenthèses :

`#define CARRE(a) a*a` le préprocesseur remplacera `CARRE(a+b)` par `a + b * a + b` et non par `(a + b) * (a + b)`.

De même `!CARRE(x)` sera remplacé par `!x * x` et non par `!(x * x)`.

Les directives de préprocesseur — Compilation conditionnelle

La compilation conditionnelle a pour but d'incorporer ou d'exclure des parties du code source dans le texte qui sera généré par le préprocesseur. Elle permet d'adapter le programme au matériel ou à l'environnement sur lequel il s'exécute, ou d'introduire dans le programme des instructions de débogage.

Les directives de compilation conditionnelle se répartissent en deux catégories, suivant le type de condition invoquée :

- ▶ la valeur d'une expression
- ▶ l'existence ou l'inexistence de symboles

Les directives de préprocesseur — Compilation conditionnelle

Condition liée à la valeur d'une expression

```
#if condition-1
    partie-du-programme -1
#elif condition-2
    partie-du-programme -2
...
#elif condition-n
    partie-du-programme -n
#else
    partie-du-programme -N
#endif
```

Le nombre de `#elif` est quelconque et le `#else` est facultatif. Chaque condition-*i* doit être une expression constante. Une seule partie-du-programme sera compilée : celle qui correspond à la première condition-*i* non nulle, ou bien la partie-du-programme-*N* si toutes les conditions sont nulles.

Par exemple, on peut écrire :

```
#define PROCESSEUR ALPHA
#if PROCESSEUR == ALPHA
    taille_long = 64;
#elif PROCESSEUR == PC
    taille_long = 32;
#endif
```

Les directives de préprocesseur — Compilation conditionnelle

Condition liée à l'existence d'un symbole

```
#ifdef symbole
    partie-du-programme -1
#else condition-2
    partie-du-programme -2
#endif
```

La directive `#else` est évidemment facultative.

De façon similaire, on peut tester la non-existence d'un symbole par :

Ce type de directive est utile pour rajouter des instructions destinées au débogage du programme :

Si `symbole` est défini au moment où l'on rencontre la directive `#ifdef`, alors `partie-du-programme-1` sera compilée et `partie-du-programme-2` sera ignorée. Dans le cas contraire, c'est `partie-du-programme-2` qui sera compilée.

```
#ifndef symbole
    partie-du-programme -1
#else condition-2
    partie-du-programme -2
#endif
```

```
#define DEBUG
...
#ifdef DEBUG
    for (i = 0; i < N; i++)
        printf("%d\n", i);
#endif /* DEBUG */
```

Il suffit alors de supprimer la directive `#define DEBUG` pour que les instructions liées au débogage ne soient pas compilées.

Découper le programme en plusieurs fichiers

Le C (C++) permet de découper le programme en plusieurs fichiers source. Chaque fichier contient une ou plusieurs fonctions. On peut ensuite inclure les fichiers dont on a besoin dans différents projets. Les fichiers d'en-tête contiennent les déclarations des types et fonctions que l'on souhaite créer.

Un programme écrit en C se compose généralement de plusieurs fichiers-sources. Il y a deux sortes de fichiers-sources :

- ▶ ceux qui contiennent effectivement des instructions ; leur nom possède l'extension `.c`,
- ▶ ceux qui ne contiennent que des déclarations ; leur nom possède l'extension `.h` (header ou en-tête).

Un fichier `.h` sert à regrouper des déclarations qui sont communes à plusieurs fichiers `.c`, et permet une compilation correcte de ceux-ci. Dans un fichier `.c` on prévoit l'inclusion automatique des fichiers `.h` qui lui sont nécessaires, grâce aux directives de compilation `#include`.

En supposant que le fichier à inclure s'appelle `entete.h`, on écrira `#include <entete.h>` s'il s'agit d'un fichier de la bibliothèque standard du C, ou `#include "entete.h"` s'il s'agit d'un fichier écrit par nous-mêmes.

Découper le programme en plusieurs fichiers – Exemples

```
/***** Prog.c original *****/
#include <stdio.h>
#define J 5

// Fonction main
int main()
{
    // Déclaration des variables
    int i = 10, k;
    k = i+J;
    printf("k = %d\n", k);
    return 0;
}
```

```
/****** Prog.c *****/
#include <stdio.h>
#define J 5

// Fonction main
int main()
{
    // Déclaration des variables
    int i = 10, k;
    k = i+J;
    printf("k = %d\n", k);
    return 0;
#include "entete.h"
```

```
/****** entete.h *****/
}
```

Découper le programme en plusieurs fichiers – Exemples

```
/****** main.cpp *****/
#include <iostream>
#include "test.hpp"

int main()
{
    int const x(5);
    std::cout << "Voici un calcul
: " << x + fonction() << std
::endl;

    return 0;
}
```

```
/****** test.cpp *****/
#include "test.hpp"

int fonction()
{
    return 12;
}
```

```
/****** test.hpp *****/
#ifndef TEST_HPP
#define TEST_HPP

int fonction();

#endif
```

Générer les fichiers objet : `g++ -std=c++version -c test.cpp main.cpp`

Lier les fichiers : `g++ test.o main.o -o mon_programme.out`

Exécuter : `./mon_programme.out`

Ou

En une seule étape : `g++ -std=c++version test.cpp main.cpp -o programme.out`

Exécuter : `./programme.out`

Les fichiers - Introduction

Le C offre la possibilité de lire et d'écrire des données dans un fichier. Pour des raisons d'efficacité, les accès à un fichier se font par l'intermédiaire d'une mémoire-tampon (buffer), ce qui permet de réduire le nombre d'accès aux périphériques (disque, clé ...). Pour pouvoir manipuler un fichier, un programme a besoin d'un certain nombre d'informations :

- ▶ l'adresse de l'endroit de la mémoire-tampon où se trouve le fichier
- ▶ la position de la tête de lecture
- ▶ le mode d'accès au fichier (lecture ou écriture)
- ▶ ...

Ces informations sont rassemblées dans une structure dont le type, **FILE ***, est défini dans `stdio.h`. Un objet de type **FILE *** est appelé flot (flux) de données (stream). Avant de lire ou d'écrire dans un fichier, on notifie son accès par la commande **fopen**. Cette fonction prend comme argument le nom du fichier et initialise un flot de données, qui sera ensuite utilisé lors de l'écriture ou de la lecture. Après les traitements, on annule la liaison entre le fichier et le flot de données grâce à la fonction **fclose**.

Les fichiers - Ouverture et fermeture d'un fichier

La fonction `fopen`

Cette fonction, de type `FILE*` ouvre un fichier et lui associe un flot de données. Sa syntaxe est : `fopen("nom_de_fichier", "mode")`

La valeur retournée par `fopen` est un flot de données. Si l'exécution de cette fonction ne se déroule pas normalement, la valeur retournée est le pointeur `NULL`. Il est donc recommandé de toujours tester si la valeur renvoyée par la fonction `fopen` est égale à `NULL` afin de détecter les erreurs (lecture d'un fichier inexistant ...).

Le premier argument de `fopen` est le nom du fichier, fourni sous forme d'une chaîne de caractères. On peut aussi définir le nom du fichier par une constante symbolique au moyen de la directive `#define` plutôt que d'explicitement le nom de fichier dans le corps du programme.

Le second argument, `mode`, est une chaîne de caractères qui spécifie le mode d'accès au fichier. Les spécificateurs de mode d'accès diffèrent suivant le type de fichier considéré :

- ▶ les fichiers textes, pour lesquels les caractères de contrôle (retour à la ligne ...) seront interprétés en tant que tels lors de la lecture et de l'écriture.
- ▶ les fichiers binaires, pour lesquels les caractères de contrôle se sont pas interprétés.

Les fichiers - Ouverture et fermeture d'un fichier

La fonction `fopen`

Les différents modes d'accès sont les suivants :

| Mode | Signification |
|-------|--|
| "r" | ouverture d'un fichier texte en lecture |
| "w" | ouverture d'un fichier texte en écriture |
| "a" | ouverture d'un fichier texte en écriture à la fin |
| "rb" | ouverture d'un fichier binaire en lecture |
| "wb" | ouverture d'un fichier binaire en écriture |
| "ab" | ouverture d'un fichier binaire en écriture à la fin (en mode append) |
| "r+" | ouverture d'un fichier texte en lecture/écriture |
| "w+" | ouverture d'un fichier texte en lecture/écriture |
| "a+" | ouverture d'un fichier texte en lecture/écriture à la fin (en mode append) |
| "r+b" | ouverture d'un fichier binaire en lecture/écriture |
| "w+b" | ouverture d'un fichier binaire en lecture/écriture |
| "a+b" | ouverture d'un fichier binaire en lecture/écriture à la fin (en mode append) |

Les fichiers - Ouverture et fermeture d'un fichier

La fonction `fopen`

Remarque : particularités de ces modes d'accès

- ▶ Si le mode contient la lettre `r`, le fichier doit exister
- ▶ Si le mode contient la lettre `w`, le fichier peut ne pas exister. Dans ce cas, il sera créé. Si le fichier existe déjà, son ancien contenu sera perdu
- ▶ Si le mode contient la lettre `a`, le fichier peut ne pas exister. Dans ce cas, il sera créé. Si le fichier existe déjà, les nouvelles données seront ajoutées à la fin du fichier précédent.

Trois flots standard peuvent être utilisés en C sans qu'il soit nécessaire de les ouvrir ou de les fermer :

- ▶ `stdin` (standard input) : unité d'entrée (par défaut, le clavier)
- ▶ `stdout` (standard output) : unité de sortie (par défaut, l'écran)
- ▶ `stderr` (standard error) : unité d'affichage des messages d'erreur (par défaut, l'écran)

Il est conseillé d'afficher systématiquement les messages d'erreur sur `stderr` afin que ces messages apparaissent à l'écran même lorsque la sortie standard est redirigée.

Les fichiers - Ouverture et fermeture d'un fichier

La fonction `fclose`

Elle permet de fermer le flot qui a été associé à un fichier par la fonction `fopen`.

Sa syntaxe est :

`fclose(flott)`

où : `flott` est le flot de type `FILE *` retourné par la fonction `fopen` correspondant.

La fonction `fclose` retourne un entier qui vaut zéro si l'opération s'est déroulée normalement (et une valeur non nulle en cas d'erreur).

Les fichiers - Les entrées-sorties formatées

La fonction d'écriture `fprintf`

La fonction `fprintf`, analogue à `printf`, permet d'écrire des données dans un fichier. Sa syntaxe est :

```
fprintf(flout, "chaîne de contrôle", expression_1, ..., expression_n)
```

où `flout` est le flot de données retourné par la fonction `fopen`. Les spécifications de format utilisées pour la fonction `fprintf` sont les mêmes que pour `printf`.

La fonction de saisie `fscanf`

La fonction `fscanf`, analogue à `scanf`, permet de lire des données dans un fichier. Sa syntaxe est :

```
fscanf(flout, "chaîne de contrôle", argument_1, ..., argument_n)
```

où `flout` est le flot de données retourné par la fonction `fopen`. Les spécifications de format utilisées pour la fonction `fscanf` sont les mêmes que pour `scanf`.

Les fichiers - Impression et lecture de caractères

Similaires aux fonctions `getchar` et `putchar`, les fonctions `fgetc` et `fputc` permettent respectivement de lire et d'écrire un caractère dans un fichier. La fonction `fgetc`, de type `int`, retourne le caractère lu dans le fichier. Elle retourne la constante `EOF` lorsqu'elle détecte la fin du fichier.

Son prototype est :

```
int fgetc(FILE*, flot)
```

où `flot` est le flot de type `FILE*` retourné par la fonction `fopen`.

Comme pour la fonction `getchar`, il est conseillé de déclarer de type `int` la variable destinée à recevoir la valeur de retour de `fgetc` pour pouvoir détecter correctement la fin de fichier.

La fonction `fputc` écrit "caractere" dans le flot de données :

```
int fputc(int caractere, FILE *flot)
```

Elle retourne l'entier correspondant au caractère lu (ou la constante `EOF` en cas d'erreur).

Il existe également deux versions optimisées des fonctions `fgetc` et `fputc` qui sont implémentées par des macros. Il s'agit respectivement de `getc` et `putc`. Leur syntaxe est similaire à celle de `fgetc` et `fputc` :

```
int getc(FILE* flot);  
int putc(int caractere, FILE *flot)
```

Les fichiers - Impression et lecture de caractères

Le programme suivant lit le contenu du fichier texte `entree.txt`, et le recopie caractère par caractère dans le fichier `sortie.txt` :

```
#include <stdio.h>
#include <stdlib.h>
#define ENTREE "entree.txt"
#define SORTIE "sortie.txt"

int main(void) {
    FILE *f_in, *f_out;
    int c;
    if ((f_in = fopen(ENTREE, "r")) == NULL) {
        fprintf(stderr, "\nErreur: Impossible de lire le fichier %s\n",
            ENTREE);
        return(EXIT_FAILURE); }
    if ((f_out = fopen(SORTIE, "w")) == NULL) {
        fprintf(stderr, "\nErreur: Impossible d'écrire dans le fichier
        %s\n", SORTIE);
        return(EXIT_FAILURE); }
    while ((c = fgetc(f_in)) != EOF)
        fputc(c, f_out);
    fclose(f_in);
    fclose(f_out);
    return(EXIT_SUCCESS);
}
```

Les fichiers - Relecture d'un caractère

Il est possible de replacer un caractère dans un flot au moyen de la fonction `ungetc` :

```
int ungetc(int caractere, FILE *flot);
```

Cette fonction place le caractère `caractere` (converti en unsigned char) dans le flot `flot`. En particulier, si `caractere` est égal au dernier caractère lu dans le flot, elle annule le déplacement provoqué par la lecture précédente. Toutefois, `ungetc` peut être utilisée avec n'importe quel caractère (sauf EOF). Par exemple, l'exécution du programme suivant sur le fichier `entree.txt` dont le contenu est `097023` affiche à l'écran `0.970.23`

Les fichiers - Relecture d'un caractère

```
#include <stdio.h>
#include <stdlib.h>
#define ENTREE "entree.txt"
int main(void)
{
    FILE *f_in;
    int c;
    if ((f_in = fopen(ENTREE,"r")) == NULL)
    {
        fprintf(stderr, "\nErreur: Impossible de lire le fichier %s\n",
        ENTREE);
        return(EXIT_FAILURE);
    }
    while ((c = fgetc(f_in)) != EOF)
    {
        if (c == '0')
            ungetc('.',f_in);
        putchar(c);
    }
    fclose(f_in);
    return(EXIT_SUCCESS);
}
```


Les fichiers - Les entrées-sorties binaires

Les fonctions d'entrées-sorties binaires permettent de transférer des données dans un fichier sans transcodage. Elles sont donc plus efficaces que les fonctions d'entrée-sortie standard, mais les fichiers produits ne sont pas portables puisque le codage des données dépend des machines. Elles sont notamment utiles pour manipuler des données de grande taille ou ayant un type composé.

Leurs prototypes sont :

```
size_t fread(void *pointeur, size_t taille, size_t nombre, FILE *flot);  
size_t fwrite(void *pointeur, size_t taille, size_t nombre, FILE *flot);
```

où **pointeur** est l'adresse du début des données à transférer, **taille** la taille des objets à transférer, **nombre** leur nombre. Rappelons que le type **size_t**, défini dans **stddef.h**, correspond au type du résultat de l'évaluation de **sizeof**. Il s'agit du plus grand type entier non signé. La fonction **fread** lit les données sur le flot **flot** et la fonction **fwrite** les écrit. Elles retournent toutes deux le nombre de données transférées.

Les fichiers - Les entrées-sorties binaires

Par exemple, le programme suivant écrit un tableau d'entiers (contenant les 50 premiers entiers) avec `fwrite` dans le fichier `sortie`, puis lit ce fichier avec `fread` et imprime les éléments du tableau.

```
#include <stdio.h>
#include <stdlib.h>
#define NB 50
#define F_SORTIE "sortie"
int main(void)
{
    FILE *f_in, *f_out;
    int *tab1, *tab2;
    int i;
    tab1 = (int*)malloc(NB * sizeof(int));
    tab2 = (int*)malloc(NB * sizeof(int));
    for (i = 0 ; i < NB; i++)
        tab1[i] = i;
    /* Ecriture du tableau dans F_SORTIE */
    if ((f_out = fopen(F_SORTIE, "w")) == NULL)
    {
        fprintf(stderr, "\nImpossible d'écrire dans le fichier %s\n",
            F_SORTIE);
        return(EXIT_FAILURE);
    }
}
```

Les fichiers - Les entrées-sorties binaires

```
fwrite(tab1, NB * sizeof(int), 1, f_out);
fclose(f_out);
/* Lecture dans F_SORTIE */
if ((f_in = fopen(F_SORTIE, "r")) == NULL)
{
    fprintf(stderr, "\nImpossible de lire dans le fichier %s\n",
F_SORTIE);
    return(EXIT_FAILURE);
}
fread(tab2, NB * sizeof(int), 1, f_in);
fclose(f_in);
for (i = 0 ; i < NB; i++)
    printf("%d\t", tab2[i]);
printf("\n");
return(EXIT_SUCCESS);
}
```

Les éléments du tableau sont bien affichés à l'écran. Par contre, on constate que le contenu du fichier `sortie` n'est pas encodé.

Les fichiers - Positionnement dans un fichier

Les différentes fonctions d'entrées-sorties permettent d'accéder à un fichier en mode séquentiel : les données du fichier sont lues ou écrites les unes à la suite des autres. Il est également possible d'accéder à un fichier en mode direct, c'est-à-dire que l'on peut se positionner à n'importe quel endroit du fichier. La fonction `fseek` permet de se positionner à un endroit précis ; elle a pour prototype :

```
int fseek(FILE *flot, long déplacement, int origine);
```

La variable `déplacement` détermine la nouvelle position dans le fichier. Il s'agit d'un déplacement relatif par rapport à l'origine; il est compté en nombre d'octets. La variable `origine` peut prendre trois valeurs :

- ▶ `SEEK_SET` (égale à 0) : début du fichier ;
- ▶ `SEEK_CUR` (égale à 1) : position courante ;
- ▶ `SEEK_END` (égale à 2) : fin du fichier.

La fonction `int rewind(FILE *flot)` permet de se positionner au début du fichier. Elle est équivalente à `fseek(flott, 0, SEEK_SET)`. La fonction `long ftell(FILE *flot)` ; retourne la position courante dans le fichier (en nombre d'octets depuis l'origine).

Les fichiers - Positionnement dans un fichier

```
#include <stdio.h>
#include <stdlib.h>
#define NB 50
#define F_SORTIE "sortie"
int main(void)
{
    FILE *f_in, *f_out;
    int *tab;
    int i;
    tab = (int*)malloc(NB * sizeof(int));
    for (i = 0 ; i < NB; i++)
        tab[i] = i;
    /* Ecriture du tableau dans F_SORTIE */
    if ((f_out = fopen(F_SORTIE, "w")) == NULL)
    {
        fprintf(stderr, "\nImpossible d'écrire dans le fichier %s\n",
            F_SORTIE);
        return(EXIT_FAILURE);
    }
    fwrite(tab, NB * sizeof(int), 1, f_out);
    fclose(f_out);
}
```

Les fichiers - Positionnement dans un fichier

```
/* Lecture dans F_SORTIE */
if ((f_in = fopen(F_SORTIE, "r")) == NULL)
{
    fprintf(stderr, "\nImpossible de lire dans le fichier %s\n",
    F_SORTIE);
    return(EXIT_FAILURE);
}
/* On se positionne à la fin du fichier */
fseek(f_in, 0, SEEK_END);
printf("\n position %ld", ftell(f_in));
/* Déplacement de 10 int en arrière */
fseek(f_in, -10 * sizeof(int), SEEK_END);
printf("\n position %ld", ftell(f_in));
fread(&i, sizeof(i), 1, f_in);
printf("\t i = %d", i);
/* Retour au debut du fichier */
rewind(f_in);
printf("\n position %ld", ftell(f_in));
fread(&i, sizeof(i), 1, f_in);
printf("\t i = %d", i);
```

Les fichiers - Positionnement dans un fichier

```
/* Déplacement de 5 int en avant */
fseek(f_in, 5 * sizeof(int), SEEK_CUR);
printf("\n position %ld", ftell(f_in));
fread(&i, sizeof(i), 1, f_in);
printf("\t i = %d\n", i);
fclose(f_in);
return(EXIT_SUCCESS);
}
```

L'exécution de ce programme affiche à l'écran :

position 200

position 160 i = 40

position 0 i = 0

position 24 i = 6

On constate en particulier que l'emploi de la fonction **fread** provoque un déplacement correspondant à la taille de l'objet lu à partir de la position courante.