Introduction au langage C

Ce document a été co-écrit à l'aide de ChatGPT 5.2 à partir du support de cours disponibles sur Moodle.

Table des matières

1 Le langage C, un langage compilé

1.1 Contexte et historique

Le langage C est historiquement situé à l'interface entre le matériel et le logiciel. Il a été conçu pour permettre l'écriture de programmes efficaces et portables, tout en conservant une proximité forte avec le fonctionnement interne des machines. Pour comprendre les choix de conception du langage, ses forces et ses contraintes, il est nécessaire de replacer son utilisation dans le contexte de l'architecture des ordinateurs et de l'évolution des microprocesseurs.

Comme l'indique la figure 1, un ordinateur est constitué de plusieurs composants matériels interconnectés. Le processeur occupe une place centrale : il exécute les instructions des programmes et effectue les opérations sur les données. La mémoire vive contient à la fois les données manipulées et le code des programmes en cours d'exécution. Les ports d'extension et les interfaces permettent la communication avec les périphériques externes, mais du point de vue du programmeur, ces éléments restent accessibles indirectement, principalement par l'intermédiaire du processeur et du système d'exploitation.

Le processeur est l'élément chargé d'exécuter les instructions élémentaires d'un programme. Un microprocesseur désigne un processeur dont l'ensemble des composants a été suffisamment miniaturisé pour être regroupé dans un unique boîtier. Les microprocesseurs se caractérisent notamment par leur jeu d'instructions, par la complexité de leur architecture interne, par le nombre de bits qu'ils peuvent traiter simultanément et par leur fréquence de fonctionnement. Ces caractéristiques influencent directement les performances, mais également la manière dont le code machine est généré et exécuté.

L'évolution des microprocesseurs au cours des dernières décennies met en évidence une augmentation considérable de la puissance de calcul et de la complexité des architectures. Le tableau 1 présente quelques microprocesseurs emblématiques et illustre cette évolution à travers des critères tels que l'année d'apparition, le nombre de transistors, la fréquence ou encore la largeur de mot.

Ce tableau met en évidence plusieurs tendances majeures, notamment l'augmentation spectaculaire du nombre de transistors et la généralisation des architectures 64 bits. Il illustre également la diversité des familles de microprocesseurs et des choix technologiques réalisés par les industriels. Le cas d'Apple est particulièrement représentatif de cette évolution. Au cours de son histoire, Apple a successivement utilisé des microprocesseurs Motorola, puis des processeurs PowerPC conçus avec IBM, avant de migrer vers les processeurs Intel, et plus récemment vers des microprocesseurs de type ARM développés en interne. Ces transitions, motivées par des contraintes de performance,

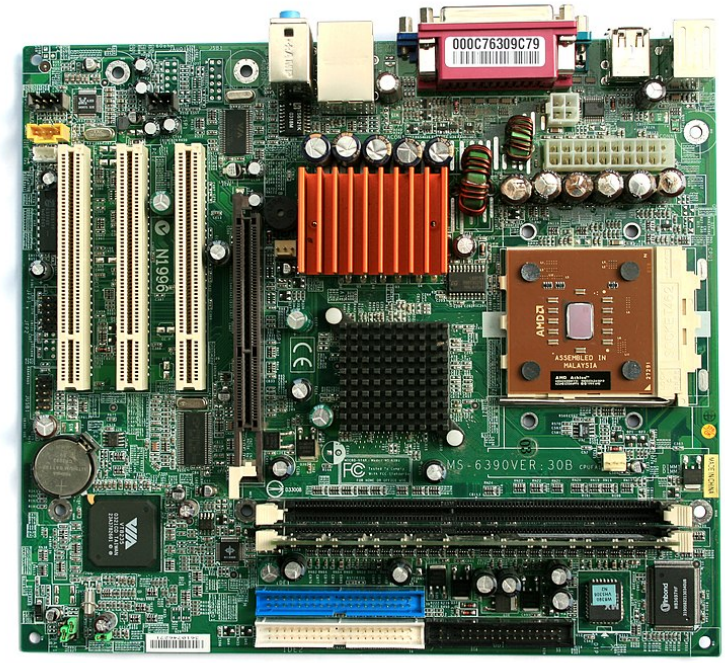

FIGURE 1 – Principaux composants matériels d'un ordinateur

de consommation énergétique et d'intégration matérielle, montrent qu'un même logiciel peut être amené à être recompilé sur des architectures très différentes, renforçant l'importance de la portabilité offerte par des langages comme le C.

Indépendamment de l'architecture matérielle, un principe fondamental demeure : tout est octet dans une machine. Les données manipulées par les programmes, qu'il s'agisse d'entiers, de réels, de caractères ou de structures plus complexes, sont représentées en mémoire sous forme de suites d'octets. Les programmes eux-mêmes sont également stockés sous forme d'octets, correspondant à des instructions codées selon le jeu d'instructions du microprocesseur. La distinction entre code et données est donc une distinction logique, mais pas une distinction physique, puisqu'ils partagent le même type de représentation en mémoire.

Le code machine, directement exécuté par le microprocesseur, est peu lisible pour un humain. Le langage assembleur fournit une représentation symbolique de ce code, dans laquelle chaque instruction machine est associée à un mnémonique plus compréhensible. L'opération qui consiste à traduire un programme écrit en assembleur vers son équivalent en code machine est appelée

Année	Fabricant	Nom	Nb instructions	Nb transistors	Fréquence	Nb bits
1979	Motorola	68000	82	68 000	2 à 12 MHz	16/32
1982	Intel	80286	99	134 000	6 à 16 MHz	16/16
1983	Acorn	ARM1	≈ 50	25 000	≤ 8 MHz	32/26
1985	Intel	80386	155	275 000	16 à 40 MHz	32/32
1992	IBM	PowerPC 601		2 800 000	50 à 120 MHz	32/32
1993	Intel	Pentium	168	3 100 000	60 à 233 MHz	32/64
1995	AMD	K5	168	4 300 000	75 à 100 MHz	32/64
⋮	⋮	⋮	⋮	⋮	⋮	⋮
2021	IBM	Power10		3 600 000 000	4 GHz	64
2022	Intel	i9 12900H	981	4 150 000 000	5 GHz	64
2022	Apple	M2 (ARMV8-5)	"232"	20 000 000 000	$\leq 3{,}4$ GHz	64

TABLE 1 – Historique de quelques microprocesseurs

assemblage. Inversement, le désassemblage consiste à reconstruire une représentation assembleur à partir d'un code machine existant.

Afin de faciliter le développement de logiciels complexes et portables, des langages de plus haut niveau ont été conçus pour exprimer les algorithmes de manière plus abstraite que l'assembleur. Le langage C appartient à cette catégorie de langages compilés. Le code source est analysé et traduit intégralement avant l'exécution, ce qui permet d'effectuer des vérifications syntaxiques et sémantiques et de produire un code machine efficace. La compilation permet également de séparer le programme de la machine cible, chaque compilation produisant un code adapté à une architecture donnée, tout en conservant une expression unique de l'algorithme dans le code source. La figure 2 illustre ce processus de compilation. Un même code C produit deux codes assembleurs à destination de deux types de microprocesseurs, en l'occurence un processeur Intel et un processeur ARM, qui sont ensuite assemblés pour produire les codes machines correspondants.

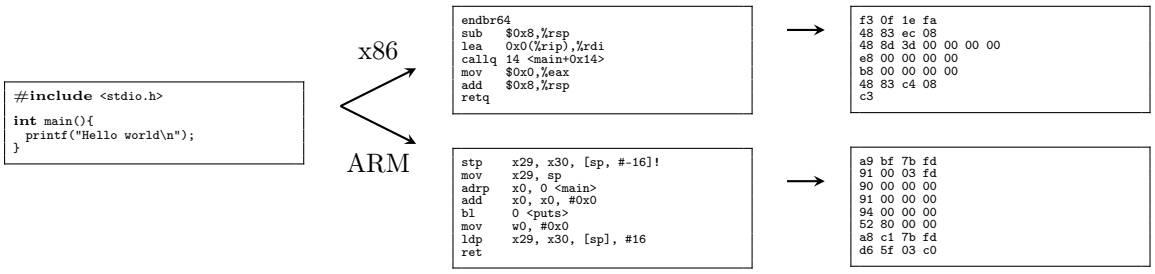

FIGURE 2 – Deux compilations d'un même programme pour des microprocesseurs différents

1.2 Du code source au programme exécutable

L'un des objectifs fondamentaux du langage C est de permettre l'écriture de programmes portables tout en produisant un code exécutable efficace. Cette efficacité repose sur un processus de traduction préalable, appelé compilation, qui transforme un programme écrit en langage C en un programme exécutable par le microprocesseur. Comprendre cette chaîne de compilation est essentiel pour maîtriser le langage, interpréter les messages du compilateur et structurer correctement des projets de taille non triviale.

Un programme écrit en C est contenu dans un ou plusieurs fichiers sources, dont l'extension est généralement `.c`. Comme nous l'avons vu, ces fichiers ne sont pas directement exécutables par la machine. Ils doivent d'abord être traduits en une représentation intermédiaire, puis combinés avec d'autres modules et bibliothèques afin de produire un exécutable. Cette transformation s'effectue en plusieurs étapes distinctes, qui peuvent être invoquées séparément ou regroupées par un compilateur moderne.

La première étape est le prétraitement. Elle concerne toutes les instructions qui commencent par le caractère `#`. Ces directives ne font pas partie du langage C à proprement parler, mais constituent un langage de macros intégré. Le préprocesseur se charge notamment d'inclure le contenu des fichiers d'en-tête, de remplacer les macros par leur définition et de supprimer certaines parties du code en fonction de conditions. À l'issue de cette étape, le compilateur obtient un fichier source C équivalent, dans lequel toutes les macros ont été résolues.

La seconde étape est la compilation proprement dite. Le compilateur analyse le code source afin de vérifier qu'il respecte la syntaxe et les règles sémantiques du langage. Il s'assure notamment que les types sont compatibles dans les expressions et les appels de fonctions. Lorsque le code est correct, il est traduit en un code de plus bas niveau, souvent sous forme de langage assembleur spécifique à l'architecture cible, puis converti en code machine. Le résultat de cette étape est un fichier objet, généralement identifié par l'extension `.o` [1].

1. Par défaut le code objet produit est compatible avec le microprocesseur de la machine sur laquelle à lieu la compilation. Il est tout fois possible de produire du code objet à destination d'un autre type de microprocesseur,

Un fichier objet contient du code machine relogeable ainsi que des informations décrivant les symboles définis et utilisés par le module. Il ne constitue pas un programme exécutable. En particulier, il peut contenir des références à des fonctions ou des variables définies dans d'autres fichiers objets ou dans des bibliothèques. Ces références doivent être résolues lors d'une étape ultérieure.

L'édition de liens est l'étape qui consiste à regrouper un ou plusieurs fichiers objets et à résoudre toutes les références externes. Lors de cette étape, le code des fonctions provenant de bibliothèques peut être intégré au programme ou référencé dynamiquement. Le résultat est un fichier exécutable, directement utilisable par le système d'exploitation. Cette séparation entre compilation et édition de liens permet une compilation modulaire et favorise la réutilisation de code.

Pour illustrer ce processus, on considère un programme C minimal affichant un message à l'écran.

```
#include <stdio.h>

int main(void) {
  printf("Hello world\n");
  return 0;
}
```

Pour obtenir le programme exécutable, il faut suivre les étapes suivantes (avec le compilateur *open source* `gcc`) :

1. compiler le programme pour obtenir un fichier objet : `gcc -c helloworld.c`. Le compilateur `gcc` invoque alors successivement le préprocesseur, le compilateur, l'assembleur et produit un fichier objet `helloworld.o` ;
2. lier le(s) fichier(s) objet avec les bibliothèques nécessaires (ici une, la `libc`) pour obtenir un exécutable : `gcc -o helloworld helloworld.o -lc`. Par défaut, le fichier exécutable produit se nomme `a.out`, mais l'option `-o` permet de spécifier un nom explicite.

Il est fortement recommandé d'activer les avertissements du compilateur, car ils signalent des constructions potentiellement dangereuses ou ambiguës.

```
gcc -Wall -pedantic helloworld.c
gcc -o helloworld helloworld.o -lc
```

La modularité du langage C repose sur l'utilisation de bibliothèques. Le cœur du langage ne fournit qu'un ensemble minimal de fonctionnalités. Les fonctionnalités supplémentaires sont regroupées dans des bibliothèques, accessibles via des fichiers d'en-tête. Lors de la compilation, ces fichiers d'en-tête doivent être accessibles afin de vérifier les déclarations. Lors de l'édition de liens, le code correspondant aux fonctions utilisées doit être disponible sous forme de bibliothèques statiques ou dynamiques. Cette organisation favorise la structuration des programmes et constitue l'un des fondements du développement logiciel en C.

1.3 Bibliothèques statiques et dynamiques

Le langage C repose sur un principe de modularité qui sépare le code source en différents modules et permet la réutilisation de fonctionnalités communes sous forme de bibliothèques. Une bibliothèque regroupe des fichiers objets, qui peuvent être utilisés par plusieurs programmes. Cette organisation évite la duplication de code et facilite la maintenance et l'évolution des logiciels.

Sous les systèmes de type UNIX, les bibliothèques suivent des conventions de nommage précises. Le nom du fichier commence généralement par le préfixe `lib`, et l'extension du fichier indique le type de bibliothèque. On distingue principalement les bibliothèques statiques, dont l'extension est `.a`, et les bibliothèques dynamiques, dont l'extension est `.so`. Ces deux types de bibliothèques se distinguent par le moment auquel le code qu'elles contiennent est intégré ou associé au programme.

on réalise alors une cross-compilation.

Pour les bibliothèques statiques, lors de l'édition de liens, le code des fonctions effectivement utilisées par le programme est copié depuis les bibliothèques et intégré directement dans l'exécutable final. Le lien entre l'appel d'une fonction et son implémentation est donc entièrement résolu à l'édition des liens. Chaque exécutable possède ainsi sa propre copie du code provenant de la bibliothèque. Cette approche garantit que le programme utilise exactement la version de la bibliothèque avec laquelle il a été lié, indépendamment de l'environnement d'exécution.

L'utilisation de bibliothèques statiques présente cependant des inconvénients. Les exécutables produits sont généralement plus volumineux, puisque le code de la bibliothèque est dupliqué dans chaque programme. De plus, si plusieurs programmes utilisent la même bibliothèque statique, le code correspondant peut être chargé plusieurs fois en mémoire, ce qui entraîne une utilisation moins efficace des ressources.

Une bibliothèque dynamique repose sur un principe différent. Le code de la bibliothèque n'est pas intégré dans l'exécutable lors de l'édition de liens. À la place, l'exécutable contient des références vers une bibliothèque externe qui sera chargée en mémoire au moment du lancement du programme, si ce n'est pas déjà le cas. Le lien entre l'appel d'une fonction et son implémentation est donc réalisé à l'exécution. Plusieurs programmes peuvent ainsi partager une même bibliothèque dynamique, qui n'est chargée qu'une seule fois en mémoire.

Les bibliothèques dynamiques permettent de réduire la taille des exécutables et de mutualiser le code en mémoire. Elles facilitent également la mise à jour de bibliothèques partagées, sans nécessiter la recompilation de tous les programmes qui les utilisent. En contrepartie, elles introduisent des contraintes supplémentaires, notamment la gestion des versions et la nécessité de disposer de la bibliothèque adéquate sur le système d'exécution. L'absence ou l'incompatibilité d'une bibliothèque dynamique peut empêcher l'exécution d'un programme.

L'édition de liens avec le compilateur `gcc` s'effectue en précisant explicitement les bibliothèques à utiliser. L'option `-l` permet d'indiquer le nom logique de la bibliothèque, sans le préfixe `lib` ni l'extension. L'option `-L` permet de spécifier le répertoire dans lequel le compilateur doit rechercher les bibliothèques. Par exemple, lors de la création d'un exécutable utilisant la bibliothèque standard du C, l'éditeur de liens associe automatiquement la bibliothèque `libc`, qui fournit notamment les fonctions d'entrée et de sortie.

Dans la pratique, la bibliothèque standard du langage C est généralement liée par défaut sur les systèmes de type UNIX. Lorsqu'un programme n'utilise aucune autre bibliothèque externe, il est donc possible de compiler et de lier directement un fichier source en une seule commande. Dans l'exemple précédent, on peut obtenir l'exécutable de la façon suivante :

```
gcc -o helloworld helloworld.c
```

Il reste néanmoins essentiel de comprendre le mécanisme de l'édition de liens et la distinction entre bibliothèques statiques et dynamiques, car ces notions interviennent systématiquement dès que l'on développe des projets modulaires ou que l'on crée ses propres bibliothèques.

1.4 Principales bibliothèques standards

Le langage C repose sur un cœur volontairement minimal, qui ne fournit que les mécanismes fondamentaux du langage. Les fonctionnalités nécessaires à l'écriture de programmes complets sont regroupées dans un ensemble de bibliothèques standards, définies par la norme du langage. Ces bibliothèques sont disponibles sur toute implémentation conforme et constituent un socle commun garantissant la portabilité des programmes.

Une bibliothèque est constituée d'un ou plusieurs modules. Un module correspond à une unité de compilation et est formé d'un fichier source `.c`, de son fichier d'en-tête associé `.h`, et du fichier objet `.o` produit après compilation. Le fichier `.h` décrit l'interface du module, c'est-à-dire les types, constantes et fonctions accessibles aux autres parties du programme, tandis que le fichier `.c` contient l'implémentation de ces fonctionnalités.

Une bibliothèque peut regrouper plusieurs modules. Il est donc possible, et même fréquent, qu'une même bibliothèque soit associée à plusieurs fichiers d'en-tête. Chacun de ces fichiers décrit

une partie spécifique de l'interface de la bibliothèque. Cette organisation favorise la modularité et permet de structurer de grandes bibliothèques en sous-ensembles cohérents.

Les principales bibliothèques standards du langage C sont présentées dans le tableau 2, avec leurs fichiers d'en-tête associés et les fonctionnalités qu'elles proposent.

Bibliothèque	Fichiers d'en-tête	Fonctionnalités principales
libc.so	assert.h	macro `assert` pour l'expression de préconditions en mode debug
	errno.h	codes d'erreur utilisés par les fonctions standards
	limits.h	constantes définissant les bornes des types entiers
	stdarg.h	création de fonctions à nombre variable de paramètres
	stdlib.h	conversions, nombres aléatoires, allocation dynamique de mémoire
	stdio.h	gestion des entrées et sorties standard
libm.so	complex.h	manipulation des nombres complexes
	math.h	fonctions mathématiques usuelles (trigonométrie, exponentielle, etc.)

TABLE 2 – Deux bibliothèques standard

L'inclusion d'un fichier d'en-tête à l'aide de la directive `#include` permet au compilateur de connaître les déclarations nécessaires à la vérification des appels de fonctions et à l'utilisation des types. Le code correspondant à ces déclarations n'est pas inclus à ce stade. Il est fourni ultérieurement, lors de l'édition de liens, par les fichiers objets ou les bibliothèques associées.

Par abus de langage, on désigne souvent une bibliothèque par son ou ses fichiers d'en-tête. Il est ainsi courant de parler de la « bibliothèque `stdio.h` », alors que `stdio.h` ne constitue que l'interface d'un ou plusieurs modules dont le code est contenu dans une bibliothèque, ici la `libc`. Cette distinction entre interface et implémentation est fondamentale pour comprendre la compilation séparée, la modularité du langage C et le rôle des bibliothèques dans la chaîne de compilation.

1.5 Synthèse de la chaîne de compilation

Les différentes étapes de la compilation et de l'édition de liens peuvent être résumées par une vue synthétique de la transformation d'un programme écrit en C vers un exécutable (cf. figure 3). Cette vue d'ensemble permet de clarifier le rôle de chaque artefact intermédiaire et de mieux comprendre les erreurs rencontrées lors du développement.

Un ou plusieurs fichiers sources `.c` constituent le point de départ du processus. Chaque fichier est compilé séparément afin de produire un fichier objet `.o`, qui contient du code machine relogeable ainsi que des informations sur les symboles définis et utilisés. Ces fichiers objets peuvent ensuite être regroupés, éventuellement avec des bibliothèques statiques, lors de l'édition de liens afin de produire un programme exécutable.

Les bibliothèques statiques peuvent être intégrées directement à l'exécutable, tandis que les bibliothèques dynamiques restent externes et sont chargées au moment de l'exécution. Le programme final est ainsi le résultat de l'assemblage de plusieurs modules compilés indépendamment, ce qui permet une organisation modulaire du code et une compilation incrémentale efficace.

Cette synthèse met en évidence la séparation claire entre les différentes étapes du processus, ainsi que la distinction entre le code source, le code objet et le programme exécutable. Elle constitue un point d'ancrage essentiel pour la compréhension du langage C en tant que langage compilé et modulaire, et servira de référence pour la suite du cours.

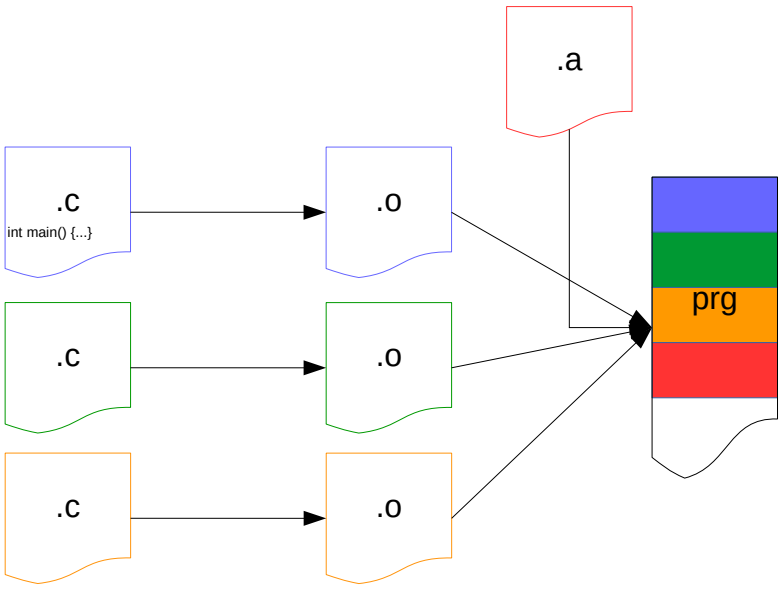

FIGURE 3 – Synthèse de la chaîne de compilation et d'édition de liens

2 Les bases du langage

2.1 Types simples

Le langage C propose un ensemble réduit de types simples, qui constituent la base de la représentation des données en mémoire. Ces types permettent de représenter des entiers, des nombres réels et des caractères, et sont directement liés aux capacités du microprocesseur. Contrairement à certains langages de plus haut niveau, le C ne fournit pas une grande variété de types prédéfinis, mais privilégie des types élémentaires combinables et proches de la machine.

Les principaux types simples du langage C sont les types entiers, les types flottants et le type caractère. Le type `int` est utilisé pour représenter des entiers signés, tandis que les types `float` et `double` permettent de représenter des nombres réels avec une précision simple ou double. Le type `char` permet de représenter un caractère, encodé sur un octet selon l'encodage ASCII ou un encodage compatible. Ces types constituent le socle à partir duquel sont construits les types plus complexes.

Le langage C permet d'affiner la représentation des types entiers à l'aide de qualificateurs. Les qualificateurs `short`, `long` et `long long` modifient la taille mémoire associée au type, et donc l'intervalle de valeurs représentables. Les qualificateurs `signed` et `unsigned` permettent de préciser si les valeurs négatives sont autorisées. La taille exacte des types dépend de l'architecture et du système d'exploitation (cf. tableau 3), mais la norme impose des relations d'ordre entre ces tailles. Cette dépendance au matériel impose une certaine prudence lors de l'écriture de programmes portables.

Historiquement, le langage C ne dispose pas de type booléen natif. Les valeurs de vérité sont représentées par des entiers, la valeur zéro correspondant au faux et toute valeur non nulle correspondant au vrai. Depuis la norme C99, le fichier d'en-tête `stdbool.h` définit un type `bool` et les constantes `true` et `false`, mais cette abstraction reste fondée sur une représentation entière. Il est donc essentiel de garder à l'esprit cette convention, notamment lors de l'écriture d'expressions conditionnelles et de tests.

Architecture	short int	int	long int	long long int
8 bits	2	2	4	8
16 bits	2	2	4	8
32 bits	2	4	4	8
64 bits (LP64 : Linux/macOS)	2	4	8	8
64 bits (LLP64 : Windows)	2	4	4	8

TABLE 3 – Nombre d'octets utilisés pour représenter des int

Le choix explicite des types simples en C a des conséquences importantes sur la précision des calculs, l'occupation mémoire et la portabilité des programmes. Comprendre ces types et leurs variantes est une étape indispensable avant d'aborder les variables, les expressions et les structures de contrôle du langage.

2.2 Variables et portée

Une variable en langage C représente une zone mémoire identifiée par un identifiant, ou son nom, et associée à un type. Avant de pouvoir être utilisée, une variable doit être déclarée, ce qui permet au compilateur de réserver l'espace mémoire nécessaire et de vérifier la cohérence des opérations effectuées sur cette variable. La déclaration d'une variable consiste à préciser son type, suivi de son identifiant, et éventuellement d'une initialisation.

```
int a;
float x = 1.0;
char c;
```

L'initialisation d'une variable permet de lui donner une valeur dès sa déclaration. Lorsqu'une variable n'est pas explicitement initialisée, son contenu est indéterminé dans le cas d'une variable locale, ce qui peut conduire à des comportements imprévisibles si elle est utilisée sans précaution. Il est donc fortement recommandé d'initialiser systématiquement les variables locales avant leur première utilisation.

Le langage C distingue les variables globales et les variables locales. Une variable globale est définie en dehors de toute fonction. Elle est accessible dans le fichier source à partir de sa définition et reste présente en mémoire pendant toute la durée d'exécution du programme. Sa durée de vie correspond donc à celle du programme entier. Lorsqu'aucune valeur initiale n'est fournie, une variable globale est automatiquement initialisée à zéro.

```
int compteurGlobal = 0;

int incrementer(void) {
  compteurGlobal++;
  return compteurGlobal;
}
```

Bien que les variables globales soient simples à utiliser, leur emploi est généralement déconseillé. Elles introduisent des dépendances implicites entre différentes parties du programme, rendent le code plus difficile à comprendre et à maintenir, et compliquent le raisonnement sur le comportement des fonctions. De plus, l'utilisation excessive de variables globales nuit à la modularité et à la réutilisabilité du code. Dans la mesure du possible, il est préférable de limiter leur usage et de privilégier le passage explicite de données par paramètres de fonctions ou l'utilisation de structures adaptées.

Une variable locale est définie à l'intérieur d'une fonction ou d'un bloc. Elle n'est accessible que dans le bloc dans lequel elle est déclarée et sa durée de vie est limitée à l'exécution de ce bloc. À chaque appel de la fonction, une nouvelle instance de la variable locale est créée, généralement sur la pile d'exécution, puis détruite à la fin de l'appel.

```
int carre(int x) {
  int resultat;
  resultat = x * x;
  return resultat;
}
```

La portée d'une variable désigne la partie du programme dans laquelle son identifiant est accessible. En C, la portée est déterminée par les blocs délimités par des accolades. Une variable déclarée dans un bloc n'est visible que dans ce bloc et dans les blocs imbriqués qu'il contient. Ce mécanisme permet de restreindre volontairement la visibilité des variables et de réduire les risques de conflits d'identifiants.

Lorsque plusieurs variables portent le même identifiant dans des portées différentes, la variable déclarée dans la portée la plus interne masque les autres. Ce phénomène, appelé masquage, peut être source d'erreurs s'il est utilisé sans précaution. Il est donc recommandé d'adopter des conventions de nommage claires et d'éviter de redéclarer des identifiants identiques dans des portées imbriquées.

La distinction entre variables globales et variables locales, ainsi que la compréhension de leur portée et de leur durée de vie, est essentielle pour écrire des programmes corrects et lisibles. Ces notions interviennent directement dans la compréhension du passage de paramètres aux fonctions, de l'utilisation des pointeurs et de la gestion de la mémoire, qui seront abordées dans les sections suivantes.

2.3 Types énumérés

Le langage C permet de définir des types énumérés à l'aide du mot-clé `enum`. Un type énuméré définit un ensemble fini de constantes symboliques, associées implicitement ou explicitement à des valeurs entières. Il permet de représenter des ensembles de valeurs discrètes de manière plus lisible et plus sûre qu'une utilisation directe d'entiers.

La définition d'un type énuméré consiste à lister les différents identifiants correspondant aux valeurs possibles. Par défaut, les constantes sont associées à des valeurs entières croissantes, en commençant par zéro, sauf si une valeur explicite est fournie.

```
enum Semaine {
  LUNDI,
  MARDI,
  MERCREDI,
  JEUDI,
  VENDREDI,
  SAMEDI,
  DIMANCHE
};
```

Dans cet exemple, le type `enum Semaine` permet de représenter les jours de la semaine de manière explicite. Une variable de ce type ne peut prendre que l'une des valeurs définies dans l'énumération, ce qui améliore la lisibilité du code et réduit les risques d'erreurs liées à l'utilisation de valeurs arbitraires.

```
enum Semaine unJour;
unJour = LUNDI;
```

En C, un type énuméré est représenté en interne par un type entier. Il n'existe pas de mécanisme empêchant strictement l'affectation d'une valeur entière quelconque à une variable de type énuméré, mais l'utilisation systématique des constantes symboliques associées à l'énumération permet de conserver une sémantique claire et cohérente.

2.4 Alias de type

Le langage C permet de définir des alias de type à l'aide du mot-clé `typedef`. Un alias de type introduit un nouvel identifiant qui désigne un type existant. Il ne crée pas un nouveau type au sens strict, mais fournit un nom alternatif permettant d'améliorer la lisibilité du code et d'exprimer plus clairement l'intention du programmeur.

La définition d'un alias s'effectue à partir d'une déclaration classique (variable ou fonction), précédée du mot-clé `typedef`. L'identifiant de la variable ou de la fonction définie devient alors l'identifiant de l'alias utilisable partout où un type est attendu.

```
typedef int Entier;
Entier a;
Entier b;
```

Dans cet exemple, `Entier` est un alias du type `int`. L'utilisation de cet alias ne modifie en rien la représentation mémoire ou le comportement du programme, mais elle permet de rapprocher le code de la terminologie algorithmique ou métier utilisée dans la spécification du problème.

Les types énumérés sont souvent utilisés conjointement avec `typedef`, afin d'éviter la manipulation explicite du mot-clé `enum` lors des déclarations. Ainsi l'exemple précédent pourrait être réécrit de la manière suivante :

```
typedef enum {
  LUNDI,
  MARDI,
  MERCREDI,
  JEUDI,
  VENDREDI,
  SAMEDI,
  DIMANCHE
} Semaine;
Semaine unJour;
unJour = LUNDI;
```

Les alias de type sont particulièrement utiles pour masquer des détails d'implémentation. Ils permettent par exemple de changer ultérieurement le type sous-jacent sans modifier l'ensemble du code qui l'utilise, à condition que les opérations effectuées restent compatibles. Ils sont également largement utilisés pour simplifier des déclarations complexes, notamment lorsque l'on manipule des pointeurs, des structures ou des types fonctionnels.

L'utilisation de `typedef` contribue ainsi à rendre les programmes plus lisibles, plus maintenables et plus proches des abstractions manipulées lors de la conception algorithmique.

2.5 Constantes

Une constante est une valeur littérale écrite directement dans le code source. Les constantes permettent d'exprimer des valeurs fixes utilisées dans un programme, comme un seuil, une valeur initiale, un caractère particulier ou un texte affiché à l'utilisateur. Le langage C propose plusieurs catégories de constantes, correspondant aux types de base, ainsi que différents mécanismes permettant de définir des constantes symboliques.

Comme nous l'avons vu, les constantes booléennes sont représentées historiquement par des entiers, la valeur zéro correspondant au faux et toute valeur non nulle correspondant au vrai. Depuis la norme C99, le fichier d'en-tête `stdbool.h` fournit le type `bool` ainsi que les constantes `true` et `false`. Même si cette abstraction améliore la lisibilité du code, il est important de retenir que la représentation sous-jacente reste entière.

Les constantes entières peuvent être écrites selon plusieurs notations. La notation décimale est la plus courante et correspond à la base dix. Le langage C permet également d'écrire des constantes en base huit et en base seize. Une constante écrite en base huit commence par un zéro,

tandis qu'une constante hexadécimale commence par `0x` ou `0X`. Ces notations sont particulièrement utiles lorsqu'on manipule des masques de bits, des adresses ou des valeurs issues de spécifications matérielles.

```
int a = 379;
int b = 0573;
int c = 0x17B;
```

Les constantes flottantes permettent de représenter des nombres réels. En C, la virgule décimale est représentée par le caractère . (point). Le langage permet également d'utiliser une notation scientifique, dans laquelle la lettre `e` représente une puissance de dix. Ces notations sont utilisées pour exprimer des valeurs très grandes ou très petites de manière compacte.

```
double x = 2.0;
double y = 0.3;
double z = 2.3e4;
```

Les constantes caractères sont entourées d'apostrophes. Elles représentent un caractère unique, encodé sur un octet (code ASCII étendu). Certains caractères ne peuvent pas être écrits directement, soit parce qu'ils ne sont pas imprimables, soit parce qu'ils entrent en conflit avec la syntaxe du langage. Ils doivent alors être écrits sous forme de séquences d'échappement, comme `\n` pour le retour à la ligne ou `\t` pour une tabulation. Il est également possible d'exprimer un caractère par son code numérique, notamment en notation octale.

```
char c1 = 'a';
char c2 = '\n';
char c3 = '\t';
char c4 = '\\';
```

Les constantes chaînes de caractères sont entourées de guillemets. Elles représentent une suite de caractères stockée en mémoire et terminée automatiquement par le caractère `\0`, appelé caractère nul. Ce caractère marque la fin de la chaîne et joue un rôle central dans la manière dont les chaînes sont manipulées en C. Une constante chaîne de caractères est donc, en pratique, un tableau de caractères terminé par un marqueur de fin.

```
char *s = "une chaine";
```

Il est important de noter que la représentation des caractères dépend de l'encodage du fichier source. En pratique, l'encodage ASCII ou des encodages compatibles sont souvent supposés, mais un encodage mal maîtrisé peut conduire à des comportements inattendus lors de l'affichage ou de la manipulation de caractères accentués.

Au-delà des constantes littérales, implicites, il est souvent nécessaire de définir des constantes explicites, afin d'éviter la duplication de valeurs dans le code et d'améliorer la lisibilité. Une approche historique consiste à utiliser le préprocesseur avec la directive `#define`, qui effectue un remplacement textuel avant compilation. Une approche plus robuste consiste à utiliser le mot-clé `const`, introduit dans la norme C90, qui permet de définir une constante typée.

```
#define MAX 100
```

```
const int TAILLE_MAX = 100;
```

L'utilisation de `const` présente plusieurs avantages. Elle permet au compilateur de vérifier que la constante est utilisée conformément à son type et elle exprime explicitement l'intention du programmeur. De plus, dans le cas des pointeurs passés en paramètres de fonctions, `const` permet d'indiquer qu'une fonction ne doit pas modifier la zone mémoire pointée, ce qui renforce la robustesse des interfaces.

Les constantes constituent ainsi un outil essentiel pour écrire un code lisible, maintenable et correct. Elles seront utilisées tout au long du cours, aussi bien pour les valeurs numériques que pour la gestion de chaînes de caractères et la définition de paramètres de fonctions.

2.6 Opérateurs

Le langage C propose un ensemble d'opérateurs permettant de construire des expressions, de combiner des valeurs et de produire des résultats. Les opérateurs du C couvrent notamment les opérations arithmétiques, les comparaisons, la logique booléenne et l'affectation. La maîtrise de ces opérateurs est indispensable pour écrire des instructions correctes et comprendre les effets produits lors de l'exécution d'un programme.

Les opérateurs relationnels permettent de comparer des valeurs. Ils produisent un résultat interprété comme une valeur de vérité, selon la convention du C. L'égalité et la différence s'écrivent respectivement `==` et `!=`, ce qui distingue clairement la comparaison de l'affectation. Les comparaisons d'ordre s'expriment avec `<`, `<=`, `>` et `>=`. Ces opérateurs sont utilisés dans les expressions conditionnelles et les tests de boucles.

```
int a = 3;
int b = 5;

if (a != b) {
  /* ... */
}
```

Les opérateurs logiques permettent de combiner des expressions booléennes. Le ET logique s'écrit `&&`, le OU logique s'écrit `||` et la négation s'écrit `!`. Ces opérateurs sont dits courts-circuités, ce qui signifie que le second opérande n'est évalué que si cela est nécessaire pour déterminer le résultat. Ce mécanisme est très utile, notamment pour éviter des erreurs, par exemple lorsqu'un test dépend de la validité d'un pointeur.

```
if (p != NULL && *p > 0) {
  /* ... */
}
```

Les opérateurs arithmétiques usuels sont disponibles. L'addition, la soustraction, la multiplication et la division s'écrivent `+`, `-`, `*` et `/`. L'opérateur `%` permet de calculer le reste de la division euclidienne entre deux entiers. Il est important de noter que la division entre deux entiers produit un entier, ce qui correspond à une division entière. Pour obtenir une division réelle, au moins l'un des opérandes doit être de type flottant.

```
int q = 7 / 2;
int r = 7 % 2;
double x = 7.0 / 2.0;
```

Un point fondamental du langage C est que l'affectation est elle-même une opération. L'opérateur `=` ne se contente pas d'assigner une valeur à une variable, il produit également une valeur, qui est la valeur affectée. Cette propriété permet d'écrire des expressions compactes, mais elle peut également conduire à des erreurs, notamment lorsque l'on confond `=` et `==` dans une condition.

```
int x, y;
x = y = 3;
```

Le langage C propose également les opérateurs unaires `++` et `--`, qui permettent respectivement d'incrémenter ou de décrémenter une variable. Lorsqu'ils sont utilisés isolément, `i++` et `++i` ont le même effet, car ils augmentent tous deux la valeur de `i` d'une unité. En revanche, lorsqu'ils sont utilisés dans une expression plus complexe, la position de l'opérateur par rapport à l'opérande modifie l'ordre dans lequel la valeur est produite.

```
int i = 3;
int j;

j = i++;
/* j vaut 3 et i vaut 4 */
```

```
i = 3;
j = ++i;
/* i vaut 4 et j vaut 4 */
```

Ces opérateurs introduisent des effets de bord, c'est-à-dire des modifications d'état en plus de la production d'une valeur. Dans la pratique, il est recommandé d'utiliser ces opérateurs avec prudence, en évitant de les combiner dans des expressions trop complexes. Un code plus explicite est souvent plus facile à lire et à maintenir, tout en réduisant les risques d'erreurs.

2.7 Instructions

Un programme en langage C est constitué d'une suite d'instructions. Une instruction représente une action élémentaire exécutée par la machine, comme une affectation, un appel de fonction ou une instruction de contrôle. Dans la syntaxe du langage, une instruction simple se termine systématiquement par un point-virgule, qui marque la fin de l'instruction. Ce point-virgule fait partie intégrante de la grammaire du langage et son oubli est une source fréquente d'erreurs de compilation.

```
a = a + 1;
```

Le langage C permet également de regrouper plusieurs instructions en une seule instruction composée. Une instruction composée est un bloc d'instructions délimité par des accolades. Elle est considérée syntaxiquement comme une instruction unique, ce qui permet de l'utiliser partout où une seule instruction est attendue, par exemple après un `if` ou dans le corps d'une boucle. L'utilisation de blocs est essentielle pour structurer un programme et pour contrôler précisément la portée des variables locales.

```
{
  a = a + 1;
  b = b + 2;
}
```

Un bloc peut contenir des déclarations de variables locales. Ces variables sont alors accessibles uniquement dans le bloc où elles sont déclarées, et leur durée de vie est limitée à l'exécution de ce bloc. Cette propriété permet de limiter volontairement la visibilité de certaines variables, d'éviter des conflits d'identifiants et de clarifier l'organisation du code.

La distinction entre instructions simples et instructions composées est particulièrement importante dans les structures de contrôle. En C, lorsqu'une structure comme `if` ou `while` est suivie d'une instruction, celle-ci peut être soit une instruction simple, soit un bloc. Lorsque l'on souhaite exécuter plusieurs instructions dans ce contexte, il est nécessaire d'utiliser un bloc. L'omission des accolades peut conduire à des erreurs logiques difficiles à détecter, car le compilateur ne signale pas nécessairement ce type de problème.

```
if (a < 10)
  a = a + 1;
  b = b + 2;
```

Dans cet exemple, seule la première affectation est contrôlée par la condition, tandis que la seconde est exécutée systématiquement. Pour exprimer l'intention correcte, il est nécessaire d'utiliser un bloc.

```
if (a < 10) {
  a = a + 1;
  b = b + 2;
}
```

2.8 Instructions conditionnelles

Les instructions conditionnelles permettent de modifier le flot d'exécution d'un programme en fonction d'une condition. En langage C, une condition est une expression évaluée selon la convention du langage, où la valeur zéro correspond au faux et toute valeur non nulle correspond au vrai. Les structures conditionnelles sont essentielles pour exprimer des alternatives, tester des cas particuliers et contrôler la logique d'un algorithme.

L'instruction conditionnelle la plus courante est `if`. Elle permet d'exécuter une instruction ou un bloc si une condition est vraie, et éventuellement d'exécuter une autre instruction ou un autre bloc dans le cas contraire, via le mot-clé `else`. La syntaxe générale est simple, mais l'utilisation de blocs est fortement recommandée dès que le corps contient plusieurs instructions.

```
if (a < 10)
  a = a + 1;
else
  a = a + 2;
```

Une particularité importante du langage C est que le `else` est associé au `if` le plus proche qui ne possède pas déjà de `else`. Ce comportement, conforme à la grammaire du langage, peut introduire des ambiguïtés lorsque des `if` sont imbriqués sans accolades. Même si le compilateur accepte ce code, il peut être difficile à lire et mener à des erreurs logiques.

```
if (a > b)
  if (c < d)
    u = v;
  else
    i = j;
```

Dans ce cas, le `else` est associé au second `if`. Si on veut associer le `else` au premier `if`, il est nécessaire d'utiliser des blocs :

```
if (a > b) {
  if (c < d)
    u = v;
} else {
  i = j;
}
```

Lorsque l'on souhaite choisir entre plusieurs cas en fonction de la valeur d'une expression entière ou d'un caractère, le langage C propose l'instruction `switch`. Elle permet de sélectionner un bloc d'instructions correspondant à une valeur donnée, via les étiquettes `case`. Le mot-clé `default` permet de définir un comportement par défaut lorsque aucun cas ne correspond. L'instruction `break` est généralement nécessaire pour éviter l'enchaînement involontaire des cas, car en l'absence de `break`, l'exécution se poursuit dans le cas suivant.

```
switch (choix) {
  case 't':
    printf("vous voulez un triangle\n");
    break;
  case 'c':
    printf("vous voulez un carre\n");
    break;
  case 'r':
    printf("vous voulez un rectangle\n");
    break;
  default:
    printf("erreur. recommencez !\n");
}
```

2.9 Instructions itératives

Les instructions itératives permettent de répéter l'exécution d'une instruction ou d'un bloc tant qu'une condition est satisfaite. Elles sont indispensables pour traiter des collections de données, réaliser des calculs successifs ou parcourir des structures. Le langage C propose trois formes principales de boucles : `for`, `while` et `do ... while`. Ces trois constructions expriment des intentions proches, mais chacune est adaptée à des situations spécifiques.

La boucle `for` est souvent utilisée lorsque le nombre d'itérations est connu ou lorsque l'itération suit une progression simple. Sa syntaxe regroupe, dans une même structure, une initialisation, une condition d'arrêt et une opération effectuée à chaque itération. Contrairement à l'écriture algorithmique classique, cette construction est très flexible et peut être utilisée de manière détournée, ce qui impose une certaine discipline pour conserver un code lisible.

```
int i;
for (i = 0; i < 10; i++)
  printf("%d\n", i);
```

La boucle `while` est adaptée aux situations où l'on souhaite répéter un traitement tant qu'une condition reste vraie. La condition est testée avant chaque itération, ce qui signifie que le corps de la boucle peut ne jamais être exécuté si la condition est fausse dès le départ. Cette boucle est souvent utilisée pour exprimer des algorithmes dont le nombre d'itérations dépend d'un calcul ou d'une convergence.

```
float g, d, m;
g = 1;
d = x;
while (d - g > epsilon) {
  m = (d + g) / 2.0;
  if (m * m > x)
    d = m;
  else
    g = m;
}
```

La boucle `do ... while` permet également de répéter un traitement tant qu'une condition est vraie, mais avec une différence essentielle : la condition est testée après l'exécution du corps de boucle. Le corps est donc exécuté au moins une fois. Cette construction est particulièrement utile lorsque l'on doit effectuer une première lecture ou une première action avant de pouvoir tester une condition de répétition.

```
char temp[MAX];
int motDePasseValide = 0;
int nbEssais = 0;

do {
  printf("Mot de passe : ");
  scanf("%s", temp);
  motDePasseValide = hash(temp) == hashMotPassseRef;
  nbEssais++;
} while (!motDePasseValide && nbEssais < nbEssaisMax);
```

Il est important de noter que, dans certains algorithmes exprimés en pseudo-code, la boucle est décrite sous la forme répéter ... jusqu'à ce que. En langage C, la condition du `do ... while` correspond alors à la négation de la condition d'arrêt exprimée en algorithmique. Cette inversion est une source fréquente d'erreurs, en particulier lors de la traduction d'un algorithme vers du code C.

3 Pointeurs et fonctions

3.1 Mémoire, adresses et représentation

Les pointeurs constituent l'un des mécanismes centraux du langage C. Ils permettent de manipuler directement des adresses mémoire, c'est-à-dire de désigner l'emplacement où sont stockées les données. Cette capacité est à la fois une force et une source fréquente d'erreurs. Elle explique en grande partie pourquoi le C est considéré comme un langage proche de la machine, et pourquoi la compréhension de la mémoire est indispensable pour programmer correctement.

Lorsqu'un programme s'exécute, il manipule des données qui sont stockées en mémoire vive. Chaque octet de cette mémoire possède une adresse, qui est un entier permettant d'identifier de manière unique l'emplacement correspondant. Ainsi, lorsqu'une variable est déclarée, le compilateur réserve une zone mémoire suffisamment grande pour contenir une valeur du type associé. Cette zone mémoire est repérée par une adresse, que l'on peut interpréter comme la position du premier octet de la variable.

Le langage C expose explicitement cette notion d'adresse. Une variable ne se réduit pas à sa valeur : elle possède également une localisation en mémoire. Cette localisation devient essentielle dès que l'on souhaite modifier une donnée depuis une autre fonction, partager une zone mémoire entre plusieurs parties du programme, ou manipuler des structures de données dynamiques comme des listes chaînées, des arbres ou des tableaux alloués dynamiquement.

Un pointeur est une variable dont la valeur est une adresse. Autrement dit, un pointeur contient l'adresse d'une donnée. Le type d'un pointeur n'indique pas seulement qu'il s'agit d'une adresse, il indique également le type de la donnée pointée. Cette information est essentielle car elle permet au compilateur de connaître la taille des objets manipulés et de vérifier, partiellement, la cohérence des opérations effectuées.

La représentation mémoire des données dépend de leur type. Un `char` occupe un octet, tandis qu'un `int` occupe plusieurs octets (cf. tableau 3). Un pointeur occupe également un certain nombre d'octets, qui dépend de l'architecture. Sur une architecture 64 bits, un pointeur occupe typiquement 8 octets, tandis que sur une architecture 32 bits, il occupe typiquement 4 octets. La taille d'un pointeur ne dépend pas du type pointé, mais uniquement de l'architecture.

Cette uniformité de représentation des adresses est un élément important pour comprendre le langage C. Quel que soit l'objet pointé, un pointeur reste une valeur interprétée comme une adresse. En revanche, le type associé au pointeur influence la manière dont le compilateur interprète cette adresse lorsqu'il faut accéder à la donnée pointée. Cette distinction entre l'adresse elle-même et le type de la donnée pointée est une notion clé, qui sera utilisée dans toute la suite, notamment pour le passage de paramètres aux fonctions et pour la manipulation de tableaux.

Déréférencer un pointeur signifie accéder à la donnée située à l'adresse contenue dans le pointeur. Un déréférencement peut être en lecture ou en écriture, selon que l'on souhaite lire la valeur pointée ou la modifier.

Enfin, il est important de rappeler qu'un pointeur peut contenir une adresse invalide. Le langage C ne vérifie pas automatiquement que l'adresse contenue dans un pointeur correspond à une zone mémoire autorisée. Déréférencer un pointeur invalide conduit à un comportement indéfini, pouvant se manifester par un crash, une corruption de données ou un résultat incohérent. La programmation en C nécessite donc une discipline particulière, notamment dans l'initialisation des pointeurs et la gestion de la durée de vie des objets pointés.

3.2 Déclaration et initialisation d'un pointeur

Un pointeur est une variable dont la valeur est une adresse mémoire. Comme toute variable en C, un pointeur doit être déclaré avant d'être utilisé. La déclaration d'un pointeur consiste à préciser le type de la donnée pointée, suivi du symbole `*`, puis de l'identifiant du pointeur. Le symbole `*` fait partie du type et indique que la variable ne contient pas directement une valeur de ce type, mais l'adresse d'une valeur de ce type.

```
int *p;
```

```
double *q;
char *s;
```

Dans ces exemples, `p` est un pointeur vers un entier, `q` est un pointeur vers un flottant en double précision et `s` est un pointeur vers un caractère. Le type du pointeur est essentiel car il permet au compilateur de connaître la taille des objets pointés et d'interpréter correctement les accès mémoire.

Comme pour toute variable locale, un pointeur déclaré sans initialisation contient une valeur indéterminée. Dans le cas d'un pointeur, cela signifie qu'il contient une adresse arbitraire, qui ne correspond pas nécessairement à une zone mémoire valide. L'utilisation d'un tel pointeur, en particulier son déréférencement, conduit à un comportement indéfini. Il est donc indispensable d'initialiser systématiquement un pointeur avant sa première utilisation.

L'initialisation la plus simple consiste à faire pointer un pointeur vers une variable existante, dont la durée de vie est connue. Dans ce cas, le pointeur reçoit l'adresse de la variable, obtenue grâce à l'opérateur `&`. Le pointeur et la variable désignent alors la même zone mémoire.

```
int a = 10;
int *p = &a;
```

Une autre forme d'initialisation courante consiste à initialiser un pointeur à `NULL`. La constante `NULL` représente une adresse invalide particulière, utilisée par convention pour indiquer qu'un pointeur ne pointe sur aucune donnée. Initialiser un pointeur à `NULL` est une bonne pratique, car cela permet de tester explicitement si le pointeur est utilisable avant de l'employer.

```
int *p = NULL;
```

Il est important de noter que `NULL` n'est pas une adresse mémoire réelle utilisable. Déréférencer un pointeur nul conduit à un comportement indéfini. L'intérêt de `NULL` est précisément de fournir une valeur reconnaissable, que l'on peut tester dans des conditions et utiliser comme marqueur d'absence de donnée.

```
if (p != NULL) {
  /* utilisation possible de p */
}
```

La déclaration d'un pointeur doit également être comprise correctement lorsque plusieurs variables sont déclarées sur une même ligne. En C, le symbole `*` est associé à l'identifiant et non au type global de la ligne. Ainsi, dans une déclaration multiple, il est possible qu'une variable soit un pointeur et une autre non, ce qui peut être source de confusion. Pour éviter ce type d'erreur, il est recommandé d'écrire une déclaration par ligne lorsque l'on manipule des pointeurs.

```
int* p1, p2;
```

Dans cet exemple, `p1` est un pointeur vers `int`, mais `p2` est un entier. Cette syntaxe est légale, mais elle est trompeuse. Une écriture plus claire consiste à séparer les déclarations.

```
int *p1;
int p2;
```

La déclaration et l'initialisation correcte des pointeurs constituent une étape fondamentale. Elles conditionnent la validité des accès mémoire et la robustesse des programmes. Dans la suite, on introduira les opérateurs permettant de manipuler explicitement les adresses et d'accéder à la valeur pointée.

3.3 Les opérateurs `&` et `*`

Les pointeurs sont manipulés en langage C à l'aide de deux opérateurs fondamentaux. L'opérateur `&` permet d'obtenir l'adresse d'une variable, tandis que l'opérateur `*` permet de déréférencer

un pointeur, d'accéder à la valeur stockée à l'adresse contenue dans un pointeur. Ces deux opérateurs sont complémentaires et constituent le mécanisme de base permettant de passer de la notion de variable à la notion d'adresse, puis de revenir à la valeur.

L'opérateur `&` est appelé opérateur d'adressage. Appliqué à une variable, il produit l'adresse de la zone mémoire dans laquelle cette variable est stockée. Le résultat de `&` est donc une valeur de type pointeur, dont le type dépend du type de la variable. Ainsi, si `a` est un entier, alors `&a` est de type `int*`.

```
int a = 42;
int *p = &a;
```

L'opérateur `*` est appelé opérateur de déréférencement. Appliqué à un pointeur, il permet d'accéder à la valeur stockée à l'adresse contenue dans ce pointeur. Le résultat de `*p` est une valeur du type pointé par `p`. Dans l'exemple précédent, `p` est un pointeur vers `int`, donc `*p` est un entier.

```
int a = 42;
int *p = &a;
int b = *p; /* b vaut 42 */
```

Le déréférencement ne sert pas uniquement à lire une valeur. Il peut également apparaître à gauche d'une affectation, ce qui permet de modifier la valeur stockée en mémoire à l'adresse pointée. Dans ce cas, la modification ne concerne pas le pointeur lui-même, mais bien la donnée pointée.

```
int a = 42;
int *p = &a;
*p = 100; /* a vaut maintenant 100 */
```

L'opérateur `*` possède deux rôles distincts en langage C, selon le contexte. Dans une déclaration, `*` indique qu'une variable est un pointeur. Dans une expression, `*` correspond au déréférencement. Il s'agit du même symbole, mais son sens dépend du contexte syntaxique. Cette particularité est une source fréquente de confusion pour les débutants, et il est important de distinguer clairement ces deux usages.

L'utilisation de ces opérateurs nécessite cependant une condition essentielle : le pointeur doit contenir une adresse valide. Déréférencer un pointeur non initialisé, un pointeur nul ou un pointeur dont la zone mémoire pointée n'existe plus conduit à un comportement indéfini. Ce point est crucial en C, car le langage ne fournit pas de mécanisme automatique empêchant ces erreurs. Dans la suite, on verra comment les pointeurs interviennent dans la manipulation de tableaux, dans la représentation des chaînes de caractères et dans le passage de paramètres aux fonctions.

3.4 Fonctions en langage C

Les fonctions permettent de structurer un programme en décomposant un problème en sous-problèmes. Elles rendent le code plus lisible, facilitent sa maintenance et favorisent la réutilisation de traitements. En langage C, une fonction est définie par un type de retour, un identifiant, une liste de paramètres et un corps, qui contient les instructions exécutées lors de l'appel.

La syntaxe générale d'une définition de fonction peut être illustrée par l'exemple suivant.

```
int somme(int a, int b) {
  return a + b;
}
```

Dans cet exemple, `somme` est l'identifiant de la fonction, `a` et `b` sont ses paramètres, et `int` est son type de retour. Lorsqu'elle est appelée, la fonction calcule une valeur entière et la renvoie au code appelant. Le mot-clé `return` permet de terminer l'exécution de la fonction et de transmettre une valeur au point d'appel. Dans une fonction dont le type de retour n'est pas `void`, il est

indispensable que le programme exécute un `return` fournissant une valeur compatible avec le type annoncé.

Le langage C permet également de définir des fonctions qui ne renvoient aucune valeur. Dans ce cas, le type de retour est `void`. Une fonction `void` peut être utilisée pour effectuer une action, comme afficher un message, modifier une structure de données ou mettre à jour des variables accessibles par pointeurs.

```
void afficherBonjour(void) {
  printf("Bonjour\n");
}
```

Dans cet exemple, la fonction ne renvoie aucune valeur. Le paramètre `void` dans la liste des paramètres indique explicitement que la fonction ne prend aucun argument. Cette écriture est recommandée, car elle rend l'interface de la fonction non ambiguë. Dans un code C moderne, il est préférable d'écrire `f(void)` plutôt que `f()` lorsque la fonction ne prend aucun paramètre.

Une fonction peut comporter plusieurs instructions `return`. L'exécution de la fonction s'arrête dès que l'un de ces `return` est atteint. Cette possibilité est utile pour gérer des cas particuliers, mais elle doit être utilisée avec modération afin de conserver une structure de code lisible.

Enfin, il est important de distinguer la déclaration et la définition d'une fonction. Une déclaration, appelée prototype, décrit l'interface de la fonction, c'est-à-dire son type de retour, son identifiant et les types de ses paramètres. La définition fournit l'implémentation complète. Les prototypes sont généralement placés dans des fichiers d'en-tête `.h`, tandis que les définitions sont placées dans des fichiers sources `.c`. Cette organisation permet la compilation séparée et la modularité des programmes, ce qui constitue un principe central du développement en langage C.

3.5 Passage de paramètres aux fonctions

Lors de l'appel d'une fonction, les valeurs fournies par le code appelant sont transmises aux paramètres formels définis dans l'interface de la fonction. Ces paramètres jouent le rôle de variables locales initialisées au début de l'exécution de la fonction. Ils permettent de fournir à la fonction les données nécessaires à son calcul, et de spécialiser son comportement en fonction des arguments passés.

En langage C, les paramètres sont toujours transmis par valeur. Cela signifie que la fonction reçoit une copie des valeurs fournies lors de l'appel. Cette règle s'applique aussi bien aux types simples, comme les entiers ou les réels, qu'aux pointeurs. Ainsi, lorsqu'un argument est transmis, la fonction ne reçoit pas directement la variable de l'appelant, mais une copie de son contenu.

```
int max2(int a, int b) {
  if (a > b)
    return a;
  return b;
}
```

Dans cet exemple, `a` et `b` sont des paramètres formels de type `int`. À l'intérieur de la fonction, ils peuvent être utilisés comme des variables locales. Toute modification de `a` ou `b` n'a aucun effet sur les variables passées lors de l'appel, car seule une copie a été transmise.

Le passage par valeur a une conséquence importante. Une fonction ne peut pas modifier directement une variable locale appartenant à une autre fonction. Si l'on souhaite qu'une fonction puisse modifier une donnée définie dans la fonction appelante, il est nécessaire de transmettre l'adresse de cette donnée. Cette adresse est stockée dans un pointeur, et la fonction peut alors accéder à la zone mémoire correspondante en déréférençant ce pointeur.

```
void incrementer(int *p) {
  *p = *p + 1;
}
```

Dans cet exemple, la fonction `incrementer` reçoit un pointeur vers `int`. Lors de l'appel, l'argument transmis est l'adresse d'une variable entière, obtenue avec l'opérateur `&`. Le pointeur est transmis par valeur, mais il désigne la zone mémoire originale. Le déréférencement permet donc de modifier directement la valeur de la variable de l'appelant.

```
int main(void) {
  int a = 10;
  incrementer(&a);
  /* a vaut maintenant 11 */
  return 0;
}
```

Ce mécanisme constitue le fondement du passage par adresse en C. Il est omniprésent dans les interfaces des bibliothèques standards, notamment pour les fonctions d'entrée et sortie, ainsi que pour de nombreuses fonctions qui produisent plusieurs résultats ou qui modifient des structures de données.

4 Entrée et sortie standard

Les entrées et sorties standard constituent le mécanisme le plus simple pour interagir avec l'utilisateur en langage C. Elles permettent d'afficher par défaut des informations à l'écran et de lire des données saisies au clavier. Ces opérations sont réalisées à l'aide de fonctions de la bibliothèque standard, principalement définies dans le fichier d'en-tête `stdio.h`. Dans ce cours, on se limite dans un premier temps aux entrées et sorties en mode texte, qui suffisent pour écrire de nombreux programmes et illustrer les concepts essentiels.

La notion d'entrée et sortie standard repose sur trois flux prédéfinis. Le flux de sortie standard correspond aux affichages normaux du programme, le flux d'entrée standard correspond aux données lues au clavier ou redirigées depuis un fichier, et le flux d'erreur standard est utilisé pour afficher des messages d'erreur. Ces flux peuvent être redirigés par le système d'exploitation, ce qui permet de chaîner des programmes ou de sauvegarder des sorties dans des fichiers sans modifier le code source.

Dans la suite, on présente les deux fonctions les plus courantes pour interagir avec ces flux. La fonction `printf` permet d'écrire sur la sortie standard, tandis que la fonction `scanf` permet de lire depuis l'entrée standard. Ces fonctions reposent sur un mécanisme de format, qui décrit comment interpréter les données à afficher ou à lire. La compréhension de ce mécanisme est essentielle, car une erreur de format peut provoquer des résultats incorrects, voire des comportements indéfinis.

4.1 Sortie standard : `printf`

La fonction `printf` permet d'écrire sur la sortie standard, généralement l'écran. Elle est définie dans le fichier d'en-tête `stdio.h` et constitue l'outil le plus courant pour produire un affichage en mode texte. La fonction repose sur une chaîne de format, qui décrit à la fois le texte à afficher et la manière dont les valeurs des arguments doivent être converties en une représentation lisible.

```
#include <stdio.h>

int main(void) {
  int a = 12;
  printf("a vaut %d\n", a);
  return 0;
}
```

La chaîne de format peut contenir du texte ordinaire, ainsi que des spécificateurs introduits par le caractère `%`. Chaque spécificateur indique le type attendu pour l'argument correspondant et la manière dont il doit être affiché. Les arguments supplémentaires sont fournis après la chaîne de

format et doivent apparaître dans le même ordre que les spécificateurs. Si le nombre d'arguments ou leurs types ne correspondent pas à la chaîne de format, le programme peut produire un résultat incohérent ou provoquer un comportement indéfini.

Les spécificateurs les plus couramment utilisés sont présentés dans le tableau 4. Ils suffisent pour la majorité des programmes de ce cours et seront réutilisés dans les exemples et les exercices.

Spécificateur	Type attendu	Remarque
`%d`	`int`	entier signé en base dix
`%u`	`unsigned int`	entier non signé en base dix
`%ld`	`long int`	entier signé de type `long`
`%lu`	`unsigned long int`	entier non signé de type `unsigned long`
`%lld`	`long long int`	entier signé de type `long long`
`%llu`	`unsigned long long int`	entier non signé de type `unsigned long long`
`%x`	`unsigned int`	entier en hexadécimal (lettres minuscules)
`%X`	`unsigned int`	entier en hexadécimal (lettres majuscules)
`%f`	`double`	réel, `float` promu en `double`
`%e`	`double`	réel en notation scientifique
`%c`	`int`	caractère, transmis sous forme d'entier
`%s`	`char*`	chaîne de caractères terminée par `\0`
`%p`	`void*`	adresse mémoire, utile pour les pointeurs
`%%`	aucun	affiche le caractère `%`

TABLE 4 – Les spécificateurs les plus utilisés

Le spécificateur `%p` permet d'afficher une adresse mémoire. Il est particulièrement utile lors du débogage, notamment lorsque l'on manipule des pointeurs. Le type attendu est `void*`, ce qui conduit souvent à effectuer un transtypage explicite afin de respecter le format.

```
int a = 42;
printf("adresse de a = %p\n", (void*)&a);
```

Il est important de noter que, dans le cas de `printf`, certains types sont automatiquement promus lors du passage des arguments. En particulier, un `float` est automatiquement converti en `double`. C'est pourquoi le spécificateur `%f` attend un `double`, même si l'on souhaite afficher une variable de type `float`. Cette règle fait partie des conversions automatiques appliquées lors de l'appel de fonctions à nombre variable d'arguments.

La fonction `printf` renvoie un entier correspondant au nombre de caractères effectivement affichés. Cette valeur peut être utilisée pour détecter certaines erreurs d'écriture, mais elle est rarement exploitée dans les programmes simples. Dans ce cours, `printf` sera principalement utilisée pour produire des affichages destinés à l'utilisateur ou pour faciliter l'observation et la compréhension du comportement d'un programme.

4.2 Entrée standard : `scanf`

La fonction `scanf` permet de lire des données depuis l'entrée standard, typiquement saisies au clavier. Elle est définie dans `stdio.h` et repose également sur une chaîne de format. Contrairement à `printf`, qui reçoit des valeurs à afficher, `scanf` doit écrire les valeurs lues dans des variables. Pour cette raison, les arguments passés à `scanf` sont généralement des adresses de variables.

```
#include <stdio.h>

int main(void) {
  int a;
  scanf("%d", &a);
  printf("a = %d\n", a);
```

```
  return 0;
}
```

L'utilisation de `scanf` met en évidence un point essentiel du langage C. Pour modifier une variable définie dans la fonction appelante, il est nécessaire de transmettre son adresse. Ainsi, pour lire un entier, il faut fournir l'adresse d'une variable de type `int`, obtenue avec l'opérateur `&`. Oublier cet opérateur conduit à une erreur grave, car `scanf` interprète alors la valeur fournie comme une adresse, ce qui provoque un comportement indéfini.

Comme pour `printf`, la chaîne de format doit correspondre exactement aux types des variables associées. Par exemple, `%d` est utilisé pour un entier, `%f` pour un `float` et `%lf` pour un `double`. Une incohérence entre le format et le type de la variable peut produire des résultats incorrects ou corrompre la mémoire.

La fonction `scanf` renvoie le nombre de valeurs effectivement lues et affectées. Cette valeur permet de vérifier que la lecture a réussi. Dans un programme robuste, il est recommandé de tester cette valeur afin de détecter les erreurs de saisie. Dans la suite du cours, on insistera sur l'importance de ces tests, car une entrée utilisateur incorrecte est une source fréquente de bugs.

Enfin, l'utilisation de `scanf` pour lire des chaînes de caractères doit être réalisée avec prudence, car une lecture non bornée peut provoquer un dépassement de tampon. Ce point sera détaillé dans la partie consacrée aux chaînes de caractères, une fois les tableaux introduits.

4.3 Fichiers

Les entrées et sorties standard permettent d'interagir simplement avec l'utilisateur, mais un programme doit souvent lire ou écrire des données de manière persistante. Le langage C fournit pour cela un mécanisme de manipulation de fichiers, fondé sur la bibliothèque standard `stdio.h`. Ce mécanisme repose sur la notion de flux, qui représente une source ou une destination de données. Un fichier est ainsi vu comme un flux associé à un support externe, généralement le disque.

En langage C, un flux est manipulé à travers un pointeur de type `FILE*`. Ce pointeur désigne une structure interne gérée par la bibliothèque standard, qui contient notamment des informations sur l'état du flux, la position courante dans le fichier, ainsi que des buffers utilisés pour optimiser les accès. Le programmeur ne manipule pas directement cette structure, mais utilise des fonctions standards pour ouvrir, lire, écrire et fermer les fichiers.

Le langage C définit trois flux standards, disponibles dans `stdio.h`. Le flux `stdin` correspond à l'entrée standard, généralement le clavier, le flux `stdout` correspond à la sortie standard, généralement l'écran, et le flux `stderr` correspond à la sortie d'erreur standard. Le flux `stderr` est destiné aux messages d'erreur ou de diagnostic. Cette séparation est particulièrement utile, car le système d'exploitation peut rediriger `stdout` et `stderr` séparément. Un programme peut ainsi produire un résultat normal redirigeable dans un fichier, tout en conservant les erreurs affichées à l'écran.

Les fonctions `printf` et `scanf` sont des cas particuliers de fonctions plus générales. Conceptuellement, `printf` correspond à un appel à `fprintf` sur le flux `stdout`, et `scanf` correspond à un appel à `fscanf` sur le flux `stdin`. Cette remarque est essentielle, car elle montre que les entrées et sorties standard et les entrées et sorties sur fichiers reposent sur le même mécanisme de flux.

```
printf("x = %d\n", x);
/* equivalent conceptuellement */
fprintf(stdout, "x = %d\n", x);

scanf("%d", &x);
/* equivalent conceptuellement */
fscanf(stdin, "%d", &x);
```

L'ouverture d'un fichier se fait avec la fonction `fopen`. Cette fonction prend deux arguments, le nom du fichier et un mode d'ouverture. Le mode `"r"` ouvre un fichier en lecture, le mode `"w"` ouvre un fichier en écriture en le créant ou en l'écrasant, et le mode `"a"` ouvre un fichier en ajout,

c'est-à-dire en écrivant à la fin. Si l'ouverture échoue, `fopen` renvoie `NULL`. Ce test est indispensable pour éviter un comportement indéfini et pour produire un message d'erreur exploitable.

```
#include <stdio.h>

int main(void) {
  FILE *f;

  f = fopen("donnees.txt", "r");
  if (f == NULL) {
    fprintf(stderr, "erreur: impossible d'ouvrir donnees.txt\n");
    return 1;
  }

  fclose(f);
  return 0;
}
```

Une fois le fichier ouvert, il est possible d'écrire des données dans ce fichier avec `fprintf`. Cette fonction fonctionne comme `printf`, mais elle écrit dans un flux donné au lieu d'écrire sur `stdout`. Elle utilise la même notion de chaîne de format et les mêmes spécificateurs. De manière analogue, la lecture de données dans un fichier texte peut être réalisée avec `fscanf`, qui fonctionne comme `scanf`, mais lit dans un flux explicite. Comme `scanf`, `fscanf` attend des adresses de variables, car la fonction doit écrire les valeurs lues dans ces variables.

```
#include <stdio.h>

int main(void) {
  FILE *f;
  int x;

  f = fopen("donnees.txt", "r");
  if (f == NULL) {
    fprintf(stderr, "erreur: fichier introuvable\n");
    return 1;
  }

  if (fscanf(f, "%d", &x) == 1)
    printf("x = %d\n", x);

  fclose(f);
  return 0;
}
```

Les fonctions de manipulation de fichiers sont nombreuses, mais un petit ensemble suffit dans la majorité des cas. Le tableau 5 présente les fonctions les plus utilisées dans ce cours, ainsi que leur rôle principal.

Lorsqu'un programme a terminé d'utiliser un fichier, il est indispensable de le fermer avec `fclose`. Cette opération libère les ressources associées au flux et garantit que les données en attente dans les buffers sont effectivement écrites sur le disque. Oublier de fermer un fichier peut conduire à une perte de données ou à une consommation inutile de ressources.

Le modèle des flux en C unifie ainsi l'entrée standard, la sortie standard, la sortie d'erreur et les fichiers. Les fonctions `printf` et `scanf` utilisent implicitement `stdout` et `stdin`, tandis que `fprintf` et `fscanf` permettent de choisir explicitement le flux utilisé. Cette cohérence facilite l'apprentissage et permet de passer progressivement de programmes interactifs à des programmes capables de traiter des fichiers de données.

Fonction	Prototype simplifié	Rôle principal
`fopen`	`FILE* fopen(const char*, const char*)`	ouvre un fichier et renvoie un flux
`fclose`	`int fclose(FILE*)`	ferme un flux et libère les ressources associées
`fprintf`	`int fprintf(FILE*, const char*, ...)`	écrit des données formatées dans un flux
`fscanf`	`int fscanf(FILE*, const char*, ...)`	lit des données formatées depuis un flux
`fgetc`	`int fgetc(FILE*)`	lit un caractère, renvoie `EOF` en fin de fichier
`fputc`	`int fputc(int, FILE*)`	écrit un caractère dans un flux
`fgets`	`char* fgets(char*, int, FILE*)`	lit une ligne ou un bloc de caractères
`fputs`	`int fputs(const char*, FILE*)`	écrit une chaîne dans un flux
`feof`	`int feof(FILE*)`	teste si la fin de fichier a été atteinte
`ferror`	`int ferror(FILE*)`	teste si une erreur de lecture ou écriture est survenue

TABLE 5 – Principales fonctions d'utilisation des flux

5 Types composites

Les types simples du langage C permettent de représenter des valeurs élémentaires, comme des entiers, des réels ou des caractères. Cependant, la majorité des programmes manipulent des ensembles de données structurées, composées de plusieurs valeurs. Le langage C fournit pour cela des types composites, qui permettent de regrouper plusieurs éléments en mémoire et de construire des structures de données plus riches.

Dans cette section, on introduit quatre familles essentielles de types composites. Les tableaux permettent de représenter une séquence contiguë d'éléments de même type. Les chaînes de caractères reposent sur des tableaux de caractères terminés par un caractère nul et constituent une convention fondamentale du langage. Les structures permettent de regrouper plusieurs champs de types potentiellement différents au sein d'un même objet. Enfin, les unions permettent de partager une même zone mémoire entre plusieurs représentations possibles, ce qui est utile dans certains contextes bas niveau.

Ces types composites sont directement liés à la représentation mémoire des données. Ils permettent d'exprimer des structures de données proches de celles utilisées en algorithmique, tout en conservant les caractéristiques du langage C, notamment sa proximité avec la machine et sa gestion explicite de la mémoire.

5.1 Tableaux

En langage C, un tableau est une structure de données permettant de regrouper plusieurs valeurs de même type sous un même identificateur. Les éléments sont stockés de manière contiguë en mémoire, ce qui permet un accès direct à n'importe quel élément à partir de son indice.

On déclare une variable de type tableau en ajoutant, après l'identificateur, la taille de chaque dimension entre crochets. Par exemple, un tableau d'entiers de taille 10 se déclare ainsi :

```
int tab1[10];
```

Le C autorise également les tableaux multidimensionnels. Un tableau de réels à deux dimensions peut par exemple s'écrire :

```
float tab2[2][5];
```

Le premier indice d'une dimension est toujours 0. Ainsi, pour un tableau de taille n, les indices valides vont de 0 à $n-1$. Dans l'exemple `tab2[2][5]`, la première dimension accepte les indices 0 et 1, et la seconde les indices 0 à 4.

Le langage C ne donne pas de sémantique particulière aux dimensions. C'est au programmeur de décider si une dimension correspond à des lignes, des colonnes, des vecteurs, ou toute autre organisation logique des données.

Une fois déclaré, un élément d'un tableau s'utilise comme une variable classique. Il est donc possible de lire ou d'écrire une case à l'aide de l'opérateur d'indice `[ ]`. Par exemple, la lecture d'un tableau d'entiers peut être réalisée par une boucle :

```
int t[10];
int i;
for (i = 0; i < 10; i++)
    scanf("%d", &t[i]);
```

On remarque ici que l'on transmet à `scanf` l'adresse de la case `t[i]`. En effet, la fonction `scanf` écrit directement dans la mémoire de la variable cible, et attend donc un pointeur.

En C, le nom d'un tableau est considéré comme un pointeur constant vers son premier élément. Autrement dit, si `t` est un tableau, l'expression `t` représente l'adresse de `t[0]`. Cette propriété est essentielle car elle explique de nombreux comportements du langage.

Le qualificatif constant signifie que l'adresse associée au tableau est fixe. Contrairement à une variable pointeur classique, il est impossible de modifier l'adresse portée par le tableau. Par exemple, une instruction du type `t = t + 1` est interdite : un tableau n'est pas une variable pointeur, mais une zone mémoire dont l'adresse de départ ne peut pas être modifiée.

En revanche, il est tout à fait possible de déclarer un pointeur et de lui affecter l'adresse du premier élément d'un tableau :

```
int t[10];
int *p;
p = t;
```

Dans ce cas, `p` est une variable pointeur, et il est possible de la modifier, par exemple avec de l'arithmétique sur les pointeurs.

Lorsqu'un tableau est utilisé comme argument dans un appel de fonction, il n'est pas recopié. À la place, seule l'adresse de sa première case est transmise. Le passage de paramètre d'un tableau se fait donc par adresse.

Une conséquence directe est que si une fonction modifie les éléments du tableau reçu en paramètre, ces modifications sont visibles dans la fonction appelante. Cette propriété est très utile, car elle évite la copie potentiellement coûteuse de grandes structures de données.

Dans la signature d'une fonction, la dimension du tableau est facultative pour un tableau à une dimension. Les deux écritures suivantes sont donc équivalentes :

```
void initialiser(int t[], int n);
void initialiser(int *t, int n);
```

Dans les deux cas, `t` est traité comme un pointeur vers des entiers. La taille réelle du tableau n'est pas connue automatiquement par la fonction. Il est donc nécessaire de transmettre explicitement cette taille, généralement sous la forme d'un paramètre supplémentaire.

Le cas des tableaux multidimensionnels est plus délicat. Lorsqu'un tableau à deux dimensions est passé en paramètre, la taille de toutes les dimensions sauf la première doit être précisée, afin que le compilateur puisse calculer correctement l'adresse d'un élément.

Par exemple, pour un tableau déclaré par `float tab2[2][5]`, une fonction peut être déclarée de la manière suivante :

```
void traiter(float t[][5], int nbLignes);
```

Ici, la valeur `5` doit obligatoirement apparaître dans le type du paramètre, car elle conditionne la disposition mémoire d'une ligne.

Le langage C ne réalise pas de vérification automatique des dépassements de tableaux. Un accès avec un indice en dehors des bornes constitue une erreur de programmation, et peut conduire à des comportements imprévisibles, parfois difficiles à diagnostiquer.

Il est donc essentiel de manipuler les indices avec rigueur, et de toujours conserver la taille d'un tableau lorsqu'elle n'est pas fixée explicitement et de manière évidente dans le code.

5.2 Chaînes de caractères

Le langage C ne propose pas de type chaîne de caractères au sens où on l'entend dans des langages plus récents. Il n'existe donc pas de type `string` en C. À la place, une chaîne est représentée par un tableau de caractères.

Une chaîne de caractères est un tableau de `char` dont le dernier élément significatif est un caractère spécial, appelé caractère nul, noté `'\0'`. Ce caractère marque la fin de la chaîne.

Par exemple, la déclaration suivante réserve un tableau de 30 caractères :

```
char nom[30];
```

Ce tableau peut contenir au plus 29 caractères visibles, car une case doit être réservée pour le caractère `'\0'`.

Ainsi, si l'utilisateur saisit n caractères, le tableau utilisera en réalité $n + 1$ cases, la dernière contenant `'\0'`. Cette convention est utilisée par toutes les fonctions classiques de manipulation de chaînes du C.

Une lecture simple au clavier peut être réalisée avec `scanf`. Par exemple :

```
char nom[30];
printf("Votre nom: ");
scanf("%s", nom);
```

On remarque que l'on ne place pas l'opérateur `&` devant `nom`. Cela s'explique par le fait que le nom du tableau `nom` est déjà l'adresse de sa première case, c'est-à-dire un pointeur constant vers `nom[0]`.

Il faut cependant être prudent : avec le spécificateur `%s`, la fonction `scanf` lit une suite de caractères jusqu'au premier séparateur (espace, tabulation ou retour à la ligne). De plus, `scanf` ne vérifie pas automatiquement que la chaîne saisie tient dans le tableau. Une saisie trop longue peut donc provoquer un dépassement de mémoire, ce qui constitue une erreur grave.

Dans un programme robuste, il est préférable d'utiliser des fonctions plus sûres, comme `fgets`, qui permet de limiter explicitement le nombre de caractères lus.

Le fait que les chaînes soient des tableaux implique plusieurs conséquences importantes.

Tout d'abord, une chaîne n'est pas un type valeur. Il n'est pas possible d'affecter directement une chaîne à une autre avec l'opérateur `=`. Par exemple, si `a` et `b` sont deux tableaux de caractères, une instruction du type `a = b` est interdite. Pour copier une chaîne, on doit utiliser une fonction dédiée, typiquement `strcpy` ou `strncpy`.

Ensuite, la taille maximale d'une chaîne stockée dans un tableau est fixée lors de la déclaration. Cette contrainte peut être levée en utilisant de l'allocation dynamique, ce qui sera abordé plus loin, notamment dans le cadre des pointeurs et de la gestion mémoire.

Enfin, il est important de distinguer deux notions souvent confondues : un tableau de caractères, et un pointeur vers des caractères. Un tableau `char nom[30]` réserve directement 30 cases mémoire. En revanche, une variable `char *p` ne réserve qu'un emplacement pour stocker une adresse. Pour que `p` puisse représenter une chaîne, il faut qu'il pointe vers une zone mémoire valide contenant une suite de caractères terminée par `'\0'`.

En C, une chaîne de caractères est donc une structure construite sur des tableaux et des pointeurs. Cette approche offre une grande efficacité, mais impose au programmeur de gérer explicitement la taille des zones mémoire et de prévenir les dépassements. Cette rigueur est indispensable pour produire un code correct et sûr.

5.3 Structures

Les structures permettent de regrouper, sous un même type, plusieurs champs pouvant être de types différents. Elles sont essentielles en C car elles permettent de construire des données plus riches que les types simples ou les tableaux, en donnant une représentation structurée à une entité du programme.

Une structure se définit avec le mot clé `struct`. La syntaxe générale est la suivante :

```
struct NomStructure {
    type1 champ1;
    type2 champ2;
    ...
};
```

Par exemple, une structure représentant une personne peut être définie ainsi :

```
struct Personne {
    char nom[30];
    char prenom[30];
};
```

Une variable de ce type peut ensuite être déclarée avec le nom complet du type :

```
struct Personne p;
```

Un point important du langage C est que la définition d'une structure crée un nouveau type dont le nom commence par `struct`. Autrement dit, le type n'est pas `Personne`, mais `struct Personne`.

Cette particularité est parfois surprenante au début, mais elle reflète l'approche historique du langage C, dans laquelle les mots clés `struct`, `union` et `enum` introduisent des espaces de noms distincts.

Pour éviter de devoir écrire `struct Personne` partout dans le code, on associe très souvent la déclaration de structure à un `typedef`. Cela permet de créer un alias de type plus pratique.

Par exemple :

```
typedef struct {
    char nom[30];
    char prenom[30];
} Personne;
```

Grâce à cet alias, on peut ensuite déclarer des variables simplement avec :

```
Personne p;
```

Cette écriture est très courante, notamment dans les bibliothèques et dans les programmes organisés en modules, car elle améliore nettement la lisibilité du code.

Les champs d'une structure sont accessibles via l'opérateur `.` (point). Par exemple :

```
Personne p;
scanf("%s", p.nom);
scanf("%s", p.prenom);
```

Lorsque l'on manipule un pointeur sur une structure, l'accès aux champs peut se faire avec l'opérateur `->`. Par exemple :

```
Personne *pp;
pp = &p;
printf("%s\n", pp->nom);
```

Cet opérateur est simplement une notation équivalente à une déréférence suivie d'un accès champ. Autrement dit, `pp->nom` est équivalent à `(*pp).nom`.

Une structure occupe en mémoire un bloc dont la taille est au moins la somme des tailles de ses champs. En pratique, la taille réelle peut être légèrement supérieure, car le compilateur peut insérer des octets de remplissage afin de respecter les contraintes d'alignement mémoire.

L'ordre des champs dans une structure a donc un impact sur l'organisation mémoire, et parfois sur la taille totale. Dans un cadre applicatif classique, ce détail n'est pas problématique, mais il devient important lorsqu'on travaille sur des données binaires, des formats de fichiers, ou des communications réseau.

Les structures constituent l'outil principal du langage C pour définir des types composites riches. Elles permettent de modéliser des objets du monde réel ou des entités logiques d'un programme, tout en restant compatibles avec la philosophie du C, fondée sur un contrôle explicite de la mémoire et des représentations.

5.4 Unions

Une union est un type composite proche d'une structure, mais dont la logique mémoire est très différente. Comme pour les structures, une union permet de regrouper plusieurs champs potentiellement de types différents. La différence essentielle est qu'une union ne réserve pas un espace mémoire pour chacun de ses champs. Au contraire, tous les champs partagent la même zone mémoire.

On définit une union avec le mot clé `union`. La syntaxe est similaire à celle des structures :

```
union NomUnion {
    type1 champ1;
    type2 champ2;
    ...
};
```

Par exemple :

```
union Nombre {
    int unEntier;
    float unReel;
};
```

La taille mémoire d'une union correspond à la taille de son plus grand champ. En effet, comme tous les champs partagent la même zone mémoire, l'union doit être suffisamment grande pour contenir le champ le plus volumineux.

Cette propriété rend les unions intéressantes dans certains contextes où l'on souhaite stocker une valeur qui peut prendre plusieurs formes possibles, sans allouer plus de mémoire que nécessaire. Une union peut par exemple représenter une valeur numérique qui est, selon le cas, un entier ou un réel.

Cependant, une union impose une contrainte importante : à un instant donné, un seul champ doit être considéré comme valide. Autrement dit, si l'on écrit dans le champ `unEntier`, puis que l'on lit le champ `unReel`, l'interprétation des bits en mémoire change complètement. Le langage n'effectue aucune conversion automatique. Il ne s'agit pas d'un transtypage, mais d'une relecture de la même représentation binaire avec un type différent.

Dans un code correct, il est donc indispensable de conserver, séparément, l'information indiquant quel champ est actuellement actif. Cette information est généralement stockée sous la forme d'un entier, ou plus proprement d'un type énuméré. L'association d'une union avec un champ indiquant son type effectif est un motif classique, souvent appelé union taguée.

Enfin, comme pour les structures, la taille réelle d'une union peut dépendre de l'alignement mémoire imposé par le compilateur. Dans la majorité des cas, ce détail n'a pas d'impact sur l'écriture du programme, mais il est important de savoir qu'une union ne garantit pas une taille strictement égale à celle de son plus grand champ.

Les unions sont donc un outil puissant mais délicat. Elles sont principalement utilisées dans des bibliothèques bas niveau, dans la manipulation de données binaires, ou lorsqu'une contrainte

mémoire est critique. Dans un cadre applicatif classique, leur utilisation doit rester maîtrisée, car elles peuvent rendre le code plus difficile à comprendre et à maintenir.

6 Particularités du langage C

6.1 Le programme principal

En langage C, l'exécution d'un programme commence toujours dans une fonction particulière : la fonction `main`. C'est le point d'entrée du programme, c'est-à-dire la première fonction appelée lorsque l'exécutable est lancé par le système d'exploitation.

Contrairement à ce que l'on voit parfois dans des exemples simplifiés, la signature réelle de la fonction `main` n'est pas seulement `int main(void)`. La forme la plus générale est :

```
int main(int argc, char **argv)
```

Le type de retour `int` indique que `main` renvoie un code de retour au système d'exploitation. Par convention, une valeur nulle signifie que le programme s'est terminé correctement. Une valeur non nulle indique qu'une erreur a été détectée durant l'exécution.

Le paramètre `argc` (*argument count*) est un entier qui représente le nombre d'arguments passés au programme lors de son lancement. Le paramètre `argv` (*argument vector*) est un tableau de chaînes de caractères, qui contient les arguments eux-mêmes.

L'argument `argv[0]` est toujours le nom du programme tel qu'il a été invoqué. Les arguments suivants, s'ils existent, sont accessibles via `argv[1]`, `argv[2]`, etc.

Le code suivant illustre ce mécanisme en affichant le nombre d'arguments et leur contenu :

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

int main(int argc, char **argv) {
    int i;

    printf("argc: %d\n", argc);

    for (i = 0; i < argc; i++)
        printf("argv[%d]: %s\n", i, argv[i]);

    return EXIT_SUCCESS;
}
```

Dans cet exemple, la constante `EXIT_SUCCESS` provient de l'en-tête `stdlib.h`. Elle représente une terminaison correcte du programme. Il existe également la constante `EXIT_FAILURE` pour indiquer une terminaison en erreur.

L'accès aux arguments de la ligne de commande est particulièrement utile lorsque l'on souhaite paramétrer un programme sans interaction avec l'utilisateur. C'est un mécanisme très courant sous les systèmes de type UNIX, notamment pour les programmes de traitement de fichiers, les outils de compilation, ou les scripts automatisés.

Enfin, il est important de noter que `argv` est un tableau de pointeurs. Chaque élément `argv[i]` est un pointeur vers une chaîne de caractères terminée par `'\0'`. Cette représentation est cohérente avec la manière dont le langage C manipule les chaînes : il ne s'agit pas d'un type dédié, mais d'une convention fondée sur les tableaux de caractères.

6.2 Pointeurs de fonction

En langage C, une fonction n'est pas uniquement un bloc d'instructions que l'on appelle. Une fonction possède également une adresse en mémoire, comme une variable. Cette propriété permet

de manipuler une fonction comme une valeur, c'est-à-dire de stocker son adresse dans une variable, puis de transmettre cette adresse à une autre fonction.

On parle alors de pointeur de fonction. Un pointeur de fonction est une variable dont la valeur est l'adresse d'une fonction, et dont le type décrit précisément la signature de cette fonction, c'est-à-dire le type de retour ainsi que les types de ses paramètres.

Cette fonctionnalité est particulièrement importante car elle permet d'écrire des fonctions génériques, capables de recevoir un comportement en paramètre. Par exemple, une fonction de tri peut recevoir en paramètre une fonction de comparaison, ce qui permet de changer la manière de trier sans modifier l'algorithme de tri lui-même.

Pour illustrer l'idée, considérons deux fonctions qui comparent deux entiers. L'une indique si le premier entier doit être placé avant le second pour un tri croissant, l'autre pour un tri décroissant :

```
int plusPetit(int a, int b) {
    return a > b;
}

int plusGrand(int a, int b) {
    return a < b;
}
```

Ces fonctions ont la même signature : elles prennent deux entiers et retournent un entier. Elles peuvent donc être stockées dans un même type de pointeur de fonction.

Supposons maintenant que l'on dispose d'une fonction `trierEtAfficherTableauDEntiers` qui prend en paramètre un pointeur de fonction correspondant à un comparateur. On peut alors choisir le comportement de tri au moment de l'appel :

```
int main() {
    trierEtAfficherTableauDEntiers(plusPetit);
    trierEtAfficherTableauDEntiers(&plusGrand);
    return 0;
}
```

On remarque que les deux écritures `plusPetit` et `&plusGrand` sont acceptées. En effet, le nom d'une fonction est interprété comme l'adresse de cette fonction. L'opérateur `&` est donc facultatif dans ce contexte.

L'intérêt des pointeurs de fonction est double. D'une part, ils permettent d'écrire du code plus modulaire, dans lequel un algorithme est séparé de la politique de traitement. D'autre part, ils permettent de factoriser du code, en évitant de dupliquer plusieurs versions d'un même algorithme pour des comportements légèrement différents.

Ce mécanisme est très utilisé dans les bibliothèques standards du C. Un exemple classique est la fonction `qsort`, qui réalise un tri générique en recevant en paramètre une fonction de comparaison. Les pointeurs de fonction sont également utilisés pour implémenter des tables de fonctions, des callbacks, ou encore des mécanismes proches du polymorphisme dans certains codes bas niveau.

6.3 Les bibliothèques

Le langage C est un langage modulaire. Le noyau du langage reste volontairement réduit, et l'on obtient la plupart des fonctionnalités supplémentaires en s'appuyant sur des bibliothèques. Cette organisation pousse naturellement à structurer un programme en plusieurs modules, plutôt que d'écrire un unique fichier source. Dans un projet bien organisé, le programme principal est souvent court et se contente d'orchestrer l'appel à des fonctions fournies par ces modules.

En pratique, un module est généralement réparti en deux fichiers. Le fichier d'en-tête, de type `.h`, décrit ce que le module rend disponible aux autres parties du programme. Il contient notamment les prototypes des fonctions publiques et, lorsqu'il y a lieu, les définitions des types nécessaires à l'utilisation de ces fonctions. Le fichier source, de type `.c`, contient l'implémentation, c'est-à-dire le code des fonctions déclarées dans le `.h`, ainsi que d'éventuels éléments internes au

module. Cette séparation est importante car elle permet de compiler indépendamment chaque fichier source tout en conservant un typage cohérent et vérifiable par le compilateur.

La compilation d'un fichier `.c` produit un fichier objet `.o`. Un fichier objet contient du code machine, mais il n'est pas directement exécutable, notamment parce qu'il peut référencer des symboles définis ailleurs. La commande suivante illustre une compilation typique d'un module, avec activation des avertissements, vérification de conformité au standard et création du fichier objet dans le répertoire `src` :

```
gcc -osrc/module.o -Wall -pedantic -c -Iinclude src/module.c
```

Lorsque plusieurs fichiers objets sont disponibles, il devient possible de les regrouper dans une bibliothèque afin de faciliter leur réutilisation. Il existe deux grandes familles de bibliothèques en C, qui se distinguent par leur moment de liaison avec le programme. Une bibliothèque statique, généralement un fichier `.a` (`.lib` sous Windows), est intégrée à l'exécutable lors de l'édition de liens. Le code nécessaire est alors copié dans le binaire final, ce qui rend l'exécutable plus autonome, mais potentiellement plus volumineux. Une bibliothèque statique se construit classiquement à l'aide de l'outil `ar`, par exemple :

```
ar -r libexemple.a module1.o module2.o
```

À l'inverse, une bibliothèque dynamique, généralement un fichier `.so` (ou `.dll` sous Windows), n'est pas recopiée dans l'exécutable. Le binaire final contient des références vers cette bibliothèque, et le chargement effectif est réalisé à l'exécution. Cette stratégie permet à plusieurs programmes de partager une même bibliothèque, et facilite la mise à jour d'un composant sans recompiler tous les exécutables qui l'utilisent, à condition que l'interface reste compatible. Une bibliothèque dynamique peut être produite avec `gcc` via une commande de la forme :

```
gcc -o libexemple.so -shared module1.o module2.o
```

Pour qu'une bibliothèque soit réellement utilisable, il ne suffit pas de fournir le fichier compilé `.a` ou `.so`. Il faut également fournir les fichiers d'en-tête associés, car ce sont eux qui décrivent l'interface et permettent au compilateur de vérifier les types et les signatures lors de la compilation du programme utilisateur. Par abus de langage, on appelle souvent bibliothèque l'ensemble constitué des fichiers `.h` et du fichier compilé correspondant. Il faut enfin garder à l'esprit qu'une bibliothèque peut regrouper plusieurs fichiers objets, et qu'inversement plusieurs fichiers d'en-tête peuvent être associés à une même bibliothèque, selon la manière dont l'interface est organisée.

6.4 Inclusions multiples et ordre définition / inclusion

Dans un projet C modulaire, les fichiers d'en-tête sont souvent inclus dans plusieurs fichiers, parfois directement dans un `.c`, parfois indirectement via d'autres `.h`. Cette organisation est normale, mais elle peut conduire à des inclusions multiples, car `#include` correspond à une inclusion textuelle effectuée par le préprocesseur : le contenu du fichier est copié à l'endroit de l'inclusion.

Le cas devient particulièrement instructif lorsque deux fichiers d'en-tête définissent deux types `TypeA` et `TypeB`, et que chacun déclare une fonction de transtypage vers l'autre type. Par exemple, on souhaite déclarer dans `TypeA.h` une fonction `TypeBEnTypeA` qui prend un `TypeB`, et dans `TypeB.h` une fonction `TypeAEnTypeB` qui prend un `TypeA`. Dans ce contexte, on est tenté d'écrire une première version où chaque fichier inclut l'autre avant de faire ses propres définitions.

Une version incorrecte typique est la suivante :

```
/* TypeA.h : version incorrecte */
#include "TypeB.h"
typedef int TypeA;
TypeA TypeBEnTypeA(TypeB);

/* TypeB.h : version incorrecte */
#include "TypeA.h"
```

```
typedef int TypeB;
TypeB TypeAEnTypeB(TypeA);
```

Cette organisation déclenche un enchaînement d'inclusions. Lorsque `TypeA.h` est inclus, il inclut `TypeB.h`, qui inclut à son tour `TypeA.h`, et ainsi de suite. Même si le compilateur finit par signaler une redéfinition, le problème de fond est plus général : au moment où `TypeA.h` déclare `TypeBEnTypeA(TypeB)`, le type `TypeB` doit déjà être connu. Or, dans la version incorrecte, `TypeB` n'est défini qu'après l'inclusion de `TypeA.h`. On se retrouve donc avec des prototypes qui utilisent un type non défini au moment où le compilateur les lit.

La résolution repose sur deux idées complémentaires. D'abord, on protège chaque fichier d'en-tête contre les inclusions multiples à l'aide d'include guards. Ensuite, et c'est le point essentiel dans cet exemple, on place la définition du type du fichier courant avant l'inclusion de l'autre fichier. Ainsi, dès qu'un des fichiers est rencontré, son type devient immédiatement connu, ce qui permet à l'autre fichier de déclarer ses prototypes de transtypage.

On obtient alors une version correcte :

```
/* TypeA.h : version correcte */
#ifndef __TYPE_A__
#define __TYPE_A__

typedef int TypeA;
#include "TypeB.h"
TypeA TypeBEnTypeA(TypeB);

#endif

/* TypeB.h : version correcte */
#ifndef __TYPE_B__
#define __TYPE_B__

typedef int TypeB;
#include "TypeA.h"
TypeB TypeAEnTypeB(TypeA);

#endif
```

Dans cette version, la définition `typedef int TypeA;` est placée avant `#include "TypeB.h"`. Ainsi, lorsque `TypeB.h` inclut `TypeA.h`, le type `TypeA` est défini immédiatement, ce qui permet ensuite à `TypeB.h` de déclarer correctement `TypeAEnTypeB(TypeA)`. Le même raisonnement s'applique symétriquement à `TypeB.h`.

Cet exemple montre donc deux règles de base pour écrire des fichiers d'en-tête robustes. Un fichier `.h` doit toujours être protégé contre les inclusions multiples. De plus, lorsqu'un en-tête doit exposer un type et des fonctions qui manipulent ce type, la définition du type doit apparaître suffisamment tôt pour que les autres en-têtes qui en dépendent puissent l'utiliser au moment de la lecture des prototypes. L'ordre entre définition et inclusion n'est donc pas un détail de style : c'est une contrainte de compilation.

6.5 Macros paramétrées (macro-fonctions)

Le préprocesseur du langage C permet de définir des macros, c'est-à-dire des règles de substitution textuelle appliquées avant la compilation. Une macro paramétrée, parfois appelée macro-fonction, ressemble à une fonction car elle possède des paramètres, mais son mécanisme est très différent : il ne s'agit pas d'un appel de fonction, il s'agit d'un remplacement de texte.

Une macro paramétrée est définie avec la directive `#define`. Sa forme générale consiste à donner un nom, une liste de paramètres, puis un corps qui sera substitué. La substitution intervient partout où le nom de la macro apparaît ensuite dans le code, après analyse par le préprocesseur.

Les macros paramétrées sont parfois utilisées pour éviter le coût d'un appel de fonction dans des portions critiques, ou pour exprimer des opérations génériques sans recourir au typage. En contrepartie, elles sont plus délicates à maîtriser, car la substitution textuelle ignore les règles habituelles d'évaluation des expressions et de contrôle de portée.

Dans l'exemple suivant, on définit une macro paramétrée sur plusieurs lignes. Elle affiche une addition, calcule un résultat, puis affiche ce résultat :

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

#define ADD(a, b, c) \
printf("%d + %d ", a, b); \
c = a + b; \
printf("= %d\n", c);

int main(int argc, char *argv[]) {
    int i = 0;
    int r = 0;

    while (argv[i++])
        ADD(r, 1, r);

    return EXIT_SUCCESS;
}
```

Cet exemple illustre un premier piège : une macro paramétrée peut contenir plusieurs instructions, mais, du point de vue du langage C, son utilisation reste une simple substitution dans le code. Ici, l'appel `ADD(r, 1, r)` remplace textuellement l'appel par trois instructions consécutives. Si l'on utilise cette macro dans un contexte où une seule instruction est attendue, par exemple juste après un `if` sans accolades, le comportement peut devenir incorrect.

Un autre piège fréquent est l'utilisation multiple d'un même paramètre dans le corps de la macro. Si un paramètre est remplacé plusieurs fois, et si l'utilisateur fournit une expression avec effets de bord, alors ces effets peuvent être déclenchés plusieurs fois. Par exemple, passer `i++` à une macro qui utilise son paramètre deux fois conduit à deux incrémentations, ce qui n'est généralement pas souhaité.

Les macros paramétrées sont donc puissantes, mais elles doivent être utilisées avec rigueur. Lorsqu'une fonctionnalité peut être exprimée correctement avec une fonction, cette solution est souvent préférable car elle bénéficie du typage, du débogage et d'un comportement bien défini. Les macros paramétrées restent néanmoins utiles dans certains cas, en particulier pour des constructions génériques, des constantes symboliques, ou des mécanismes conditionnels contrôlés par le préprocesseur.

6.6 Mots clés `static` et `extern`

Le langage C propose plusieurs mots clés qui ne modifient pas directement le déroulement des instructions, mais qui influencent la durée de vie, la visibilité et l'édition de liens des variables et des fonctions. Parmi eux, `static` et `extern` jouent un rôle central dans l'écriture de programmes modulaires.

Le mot clé `static` possède deux usages principaux, qu'il est indispensable de distinguer. Lorsqu'il est utilisé devant une variable locale d'une fonction, `static` modifie la durée de vie de cette variable. Une variable locale classique est créée à chaque appel de la fonction, puis détruite à la fin de l'exécution. À l'inverse, une variable locale déclarée `static` n'est initialisée qu'une seule fois et conserve sa valeur entre deux appels. La variable reste locale au sens de sa portée, car elle n'est accessible que dans la fonction, mais sa durée de vie devient celle du programme entier.

Par exemple :

```
void foo() {
    static int compteurDAppels = 0;
    compteurDAppels++;
}
```

Dans cet exemple, `compteurDAppels` permet de mémoriser combien de fois la fonction `foo` a été appelée. Si la variable n'était pas déclarée `static`, elle serait recréée à chaque appel et sa valeur serait réinitialisée.

Le second usage de `static` concerne les variables globales et les fonctions. Lorsqu'il est placé devant la définition d'une variable globale ou d'une fonction, `static` limite la visibilité du symbole au fichier source courant. La variable ou la fonction devient alors privée au module, c'est-à-dire qu'elle ne peut pas être utilisée depuis un autre fichier `.c`. Ce mécanisme est essentiel pour éviter les conflits de noms entre modules et pour empêcher qu'un utilisateur d'une bibliothèque accède à des éléments internes qui ne font pas partie de l'interface publique.

Le mot clé `extern` joue un rôle complémentaire. Il est utilisé pour indiquer qu'un symbole est défini ailleurs, généralement dans un autre fichier source. Il s'applique principalement aux variables globales. Lorsqu'une variable globale est définie dans un fichier, on peut la rendre accessible dans un autre fichier en écrivant une déclaration `extern` correspondante. Le rôle de `extern` est donc de dire au compilateur que le symbole existe, mais que sa définition réelle sera trouvée lors de l'édition de liens.

Pour les fonctions, l'utilisation de `extern` est généralement implicite. En effet, une fonction déclarée dans un fichier d'en-tête est par défaut considérée comme définie dans un autre fichier. Le mot clé `extern` peut être écrit explicitement, mais il est facultatif.

Les mots clés `static` et `extern` sont donc fondamentaux pour comprendre la différence entre portée, durée de vie et visibilité entre fichiers. Ils permettent de contrôler précisément ce qui doit rester interne à un module, ce qui doit être partagé entre plusieurs fichiers, et ce qui doit être accessible depuis l'extérieur.

6.7 Allocation statique et allocation dynamique

Lorsqu'un programme en C manipule des données, il doit nécessairement réserver de la mémoire pour les stocker. Cette réservation peut être réalisée de plusieurs manières, qui correspondent à des durées de vie et des usages différents. On distingue principalement l'allocation statique et l'allocation dynamique.

L'allocation statique correspond à une réservation mémoire dont la taille est connue à la compilation, et dont la durée de vie est déterminée par la structure du programme. Par exemple, une variable locale déclarée dans une fonction est allouée automatiquement lors de l'appel de la fonction, puis libérée automatiquement à la fin de l'exécution de cette fonction. On parle souvent d'allocation automatique, car le programmeur n'a rien à gérer explicitement. Une variable globale, quant à elle, est allouée une seule fois pour toute la durée du programme.

Ces allocations sont simples à utiliser, sûres du point de vue de la gestion mémoire, et très efficaces. En revanche, elles ont une limitation importante : la taille des structures de données doit être fixée lors de l'écriture du code, ou au moins déterminable sans intervention de l'utilisateur au moment de l'exécution.

L'allocation dynamique permet de lever cette contrainte. Elle consiste à demander explicitement au système une zone mémoire dont la taille est calculée pendant l'exécution. Cette zone est obtenue dans le tas (*heap*). L'allocation dynamique est indispensable dès que l'on souhaite créer des structures de taille variable, par exemple un tableau dont la taille dépend d'une saisie utilisateur, une liste chaînée, un arbre, ou encore une structure de données construite progressivement.

En C, l'allocation dynamique est réalisée principalement à l'aide de la fonction `malloc`, définie dans l'en-tête `stdlib.h`. La fonction `malloc` prend en paramètre une taille en octets, et retourne un pointeur vers une zone mémoire suffisamment grande, ou `NULL` si l'allocation échoue.

Par exemple, pour allouer un tableau de n entiers :

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

```
int *t;
int n = 100;

t = (int*) malloc(n * sizeof(int));
```

L'expression `sizeof(int)` permet de rendre le code portable, car la taille d'un `int` dépend de l'architecture et du compilateur. On obtient ainsi un tableau dynamique contenant n cases de type `int`.

Une fois la zone mémoire allouée, elle se manipule comme un tableau classique. Par exemple, `t[i]` est valide pour tout indice compris entre 0 et $n - 1$. Il faut cependant garder à l'esprit que, contrairement à un tableau statique, la taille n'est pas connue du compilateur. Il est donc nécessaire de conserver la valeur de `n` pour éviter tout dépassement.

Une allocation dynamique doit être libérée explicitement. En C, la libération se fait avec la fonction `free`. Oublier de libérer une zone mémoire conduit à une fuite mémoire, c'est-à-dire une perte progressive de mémoire disponible. À l'inverse, libérer deux fois la même zone mémoire, ou libérer une adresse qui n'a pas été obtenue par `malloc`, est une erreur grave qui peut conduire à un crash ou à un comportement imprévisible.

La fonction `calloc` est une variante de `malloc` qui alloue une zone mémoire et initialise tous les octets à zéro. Elle est utile lorsque l'on souhaite obtenir une zone mémoire propre, notamment pour des tableaux. La fonction `realloc` permet quant à elle de redimensionner une zone déjà allouée, ce qui est pratique lorsqu'on veut agrandir progressivement une structure de données.

L'allocation dynamique donne donc une grande flexibilité, mais elle impose une discipline stricte. Le programmeur doit vérifier le résultat de `malloc`, conserver les pointeurs retournés, libérer toutes les zones allouées, et s'assurer de ne jamais utiliser une zone après l'avoir libérée. La maîtrise de ces règles est indispensable pour écrire des programmes fiables en C.

Enfin, il est important de souligner que l'allocation dynamique est intimement liée à l'utilisation des pointeurs. La plupart des structures de données avancées en algorithmique, comme les listes chaînées, les piles, les files, ou les arbres, reposent sur des allocations dynamiques et sur la manipulation rigoureuse de pointeurs.

6.8 La bibliothèque standard

Le langage C fournit un ensemble de bibliothèques standardisées, accessibles via des fichiers d'en-tête, qui complètent le noyau du langage. Ces bibliothèques définissent des types, des constantes et des fonctions qui permettent de réaliser les opérations usuelles, comme l'affichage, la gestion de chaînes de caractères, l'allocation mémoire, ou encore l'accès aux fichiers.

Dans cette section, on s'intéresse plus particulièrement à deux fichiers d'en-tête importants, qui permettent d'écrire des programmes plus robustes : `errno.h` et `assert.h`. Le premier est lié à la gestion des erreurs dans les appels de fonctions, et le second permet de vérifier des propriétés du programme pendant son exécution.

6.8.1 `errno.h`

De nombreuses fonctions de la bibliothèque standard, et plus généralement de l'interface POSIX, signalent une erreur en renvoyant une valeur particulière. Par exemple, certaines fonctions retournent `NULL`, d'autres retournent `-1`, et d'autres encore retournent une valeur spéciale indiquant un échec.

Lorsqu'une erreur survient, ces fonctions peuvent également mettre à jour une variable globale nommée `errno`. Cette variable contient un code numérique représentant la cause de l'erreur. Le fichier d'en-tête `errno.h` définit cette variable ainsi que les constantes associées aux codes d'erreurs les plus courants.

L'utilisation de `errno` permet donc d'obtenir une information plus précise que le simple test de la valeur de retour. Elle est particulièrement utile lorsqu'une fonction peut échouer pour plusieurs raisons différentes.

Par exemple, la fonction `fopen` retourne `NULL` si l'ouverture d'un fichier échoue. Dans ce cas, `errno` est mis à jour et indique la cause, par exemple un fichier inexistant, un problème de permission, ou un chemin invalide. Une manière classique de traiter cette situation consiste à tester le retour, puis à utiliser une fonction d'affichage d'erreur.

La fonction `perror`, disponible via `stdio.h`, affiche un message correspondant à la valeur courante de `errno`. On peut également utiliser `strerror` (dans `string.h`) pour obtenir une chaîne de caractères décrivant l'erreur.

L'exemple suivant illustre une ouverture de fichier robuste :

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

int main() {
    FILE *f;

    f = fopen("donnees.txt", "r");
    if (f == NULL) {
        perror("Erreur fopen");
        return EXIT_FAILURE;
    }

    fclose(f);
    return EXIT_SUCCESS;
}
```

Dans cet exemple, la valeur de retour de `fopen` est testée immédiatement. Si l'ouverture échoue, le programme affiche un message explicite puis se termine proprement.

Il est important de noter que `errno` n'a de sens que juste après un appel de fonction ayant échoué. En effet, `errno` n'est pas automatiquement remis à zéro après un succès. De plus, certaines fonctions peuvent modifier `errno` même en cas de réussite. Une bonne pratique consiste donc à ne consulter `errno` qu'après avoir détecté une valeur de retour indiquant un échec.

6.8.2 `assert.h`

Le fichier d'en-tête `assert.h` fournit une macro nommée `assert`. Cette macro permet de vérifier, à l'exécution, qu'une condition est vraie. Si la condition est fausse, le programme s'arrête immédiatement et affiche un message d'erreur contenant la condition testée, ainsi que le fichier et la ligne où l'erreur a été détectée.

La macro `assert` est particulièrement utile pour exprimer des hypothèses internes au programme, c'est-à-dire des propriétés qui doivent être vraies si le code est correct. Il ne s'agit pas d'un mécanisme de gestion d'erreur destiné à l'utilisateur, mais d'un outil de vérification pour le programmeur.

Par exemple, si une fonction suppose qu'un pointeur reçu en paramètre n'est jamais `NULL`, on peut écrire :

```
#include <assert.h>

void traiter(int *p) {
    assert(p != NULL);
    /* utilisation de p */
}
```

Si la fonction `traiter` est appelée avec un pointeur nul, le programme s'arrête immédiatement. Cette détection précoce permet d'éviter des erreurs plus graves, comme une déréférence de pointeur nul, qui conduirait à un crash moins explicite.

L'utilisation de `assert` est également utile pour vérifier des invariants d'algorithmes ou des préconditions de fonctions. Par exemple, une fonction de recherche dans un tableau peut vérifier que la taille est positive, ou qu'un indice est bien dans les bornes attendues.

Enfin, il est important de souligner que `assert` est une macro contrôlée par le préprocesseur. Si le symbole `NDEBUG` est défini lors de la compilation, alors toutes les assertions sont désactivées et ne produisent plus aucun code. Cela permet de conserver les assertions durant le développement et les tests, puis de les retirer automatiquement dans une version finale destinée à la production.

Ainsi, `assert` constitue un outil simple mais puissant pour renforcer la fiabilité d'un programme C, en détectant tôt les incohérences et en documentant explicitement les hypothèses faites par le code.

7 Compilation d'un grand projet

Lorsque l'on passe d'un exercice court à un projet organisé en modules, la compilation ne consiste plus à appeler une seule commande `gcc` sur un fichier. On souhaite compiler séparément chaque module, regrouper certains objets dans une bibliothèque, puis lier l'exécutable final. L'objectif est double : obtenir une compilation reproductible et, surtout, éviter de recompiler inutilement tout le projet lorsqu'un seul fichier change.

Prenons comme exemple un programme de tri par minimum successif. Le projet est organisé de manière classique, avec un répertoire `src` contenant les fichiers `.c`, un répertoire `include` pour les fichiers `.h`, un répertoire `lib` pour les bibliothèques produites, et un répertoire `bin` pour l'exécutable final. L'exécutable `bin/tri` dépend de plusieurs modules, dont un module utilitaire `echanger` fourni sous la forme d'une bibliothèque statique `lib/libechanger.a`.

7.1 Une première solution : un script `bash`

Une première manière de compiler un tel projet consiste à écrire un script `bash` qui enchaîne toutes les commandes nécessaires.

```
#!/bin/bash
gcc -o src/echanger.o -Wall -pedantic -c -Iinclude src/echanger.c
ar -r lib/libechanger.a src/echanger.o
gcc -o src/triParMinSuccessif.o -Wall -pedantic -c -Iinclude src/triParMinSuccessif.c
gcc -o src/main.o -Wall -pedantic -c -Iinclude src/main.c
gcc -o bin/tri src/main.o src/triParMinSuccessif.o -Llib -lechanger
```

Il compile d'abord `echanger.c` en fichier objet, construit la bibliothèque `libechanger.a`, compile ensuite les autres modules en objets, puis réalise l'édition de liens de l'exécutable :

Ce script met en évidence les étapes réelles de construction. Les compilations avec l'option `-c` produisent des fichiers objets `.o`. La commande `ar` regroupe certains objets dans une bibliothèque statique `.a`. Enfin, l'édition de liens produit l'exécutable en liant les objets du projet avec la bibliothèque `echanger` via `-Llib` et `-lechanger`.

Cette approche fonctionne, mais elle présente une limitation importante : le script exécute toutes les commandes à chaque fois, même si un seul fichier a été modifié.

7.2 Pourquoi passer à `make`

L'outil `make` est conçu pour automatiser ce processus en tenant compte des dépendances. L'idée est de décrire ce que l'on veut produire, de préciser de quoi cela dépend, et de laisser `make` décider quoi recompiler. Si un fichier source n'a pas changé depuis la dernière compilation, il n'est pas recompilé. Si, au contraire, un fichier source ou un en-tête impactant un module a changé, seuls les éléments nécessaires sont reconstruits.

Un fichier `Makefile` est composé de deux types d'éléments : des variables, qui factorisent les commandes et options, et des règles, qui expriment les dépendances entre fichiers. Une règle indique une cible, ses dépendances, et les actions à exécuter pour construire la cible.

7.3 Un premier `Makefile` équivalent au script

On peut commencer par écrire un `Makefile` qui reproduit exactement les étapes du script. La différence est que l'on exprime des cibles et des dépendances, ce qui permet à `make` de ne construire que ce qui est nécessaire.

Le fichier suivant correspond à une première version simple :

```
all: bin/tri

bin/tri: lib/libechanger.a src/triParMinSuccessif.o src/main.o
	gcc -o bin/tri src/main.o src/triParMinSuccessif.o -Llib -lechanger

lib/libechanger.a: src/echanger.o
	ar -r lib/libechanger.a src/echanger.o

src/echanger.o: src/echanger.c
	gcc -o src/echanger.o -Wall -pedantic -c -Iinclude src/echanger.c

src/triParMinSuccessif.o: src/triParMinSuccessif.c
	gcc -o src/triParMinSuccessif.o -Wall -pedantic -c -Iinclude src/triParMinSuccessif.c

src/main.o: src/main.c
	gcc -o src/main.o -Wall -pedantic -c -Iinclude src/main.c
```

Cette version décrit déjà correctement les dépendances. Si l'on modifie uniquement `src/main.c`, `make` recompilera `src/main.o` puis relancera l'édition de liens de `bin/tri`, mais ne recompilera pas `echanger.c` et ne reconstruira pas `libechanger.a`.

7.4 Factorisation avec des variables de `make`

La version précédente fonctionne, mais elle duplique de nombreuses options. `make` permet de factoriser ces éléments en variables, ce qui rend le fichier plus lisible et plus simple à maintenir.

On utilise typiquement une variable pour le compilateur, des variables pour les options, et des variables décrivant les fichiers sources et objets. Une convention courante consiste à utiliser `CC` pour le compilateur C, `CFLAGS` pour les options de compilation, `CPPFLAGS` pour les options de préprocesseur comme `-I`, et `LDFLAGS` ou `LDLIBS` pour l'édition de liens.

Une version plus propre du même fichier est la suivante :

```
CC = gcc
AR = ar
ARFLAGS = -r
CFLAGS = -Wall -pedantic
CPPFLAGS = -Iinclude
BIN = bin/tri
LIB = lib/libechanger.a

OBJ_MAIN = src/main.o src/triParMinSuccessif.o
OBJ_LIB = src/echanger.o

all: $(BIN)

$(BIN): $(LIB) $(OBJ_MAIN)
	$(CC) -o $@ $(OBJ_MAIN) -Llib -lechanger

$(LIB): $(OBJ_LIB)
	$(AR) $(ARFLAGS) $@ $(OBJ_LIB)
```

```
src/main.o: src/main.c
	$(CC) $(CFLAGS) $(CPPFLAGS) -c -o $@ $<

src/triParMinSuccessif.o: src/triParMinSuccessif.c
	$(CC) $(CFLAGS) $(CPPFLAGS) -c -o $@ $<

src/echanger.o: src/echanger.c
	$(CC) $(CFLAGS) $(CPPFLAGS) -c -o $@ $<
```

Cette version introduit aussi des variables automatiques de `make`. Dans une règle, `$@` représente la cible, et `$<` représente la première dépendance. Cela évite de recopier les noms de fichiers dans les commandes, ce qui limite les erreurs lors des modifications.

7.5 Règles d'inférence et variables automatiques

La majorité des règles de compilation des fichiers objets ont exactement la même forme : un fichier `.o` se construit à partir d'un fichier `.c` en appelant `gcc` avec les bons drapeaux. Écrire une règle distincte pour chaque fichier devient vite inutilement répétitif.

`make` propose alors deux mécanismes complémentaires. Le premier consiste à écrire une règle de motif, qui décrit une famille entière de règles. Le second consiste à utiliser les variables automatiques pour récupérer, dans la commande, le nom de la cible et celui de la dépendance.

Une règle de motif classique pour la compilation est :

```
%.o: %.c
	$(CC) $(CFLAGS) $(CPPFLAGS) -c -o $@ $<
```

Avec cette seule règle, `make` sait comment construire n'importe quel fichier `.o` à partir du `.c` correspondant, tant que les chemins correspondent. Dans ce cadre, on peut simplifier le `Makefile` en listant uniquement les objets nécessaires et en laissant `make` déduire la manière de les produire.

On obtient alors un fichier plus concis :

```
CC = gcc
AR = ar
ARFLAGS = -r
CFLAGS = -Wall -pedantic
CPPFLAGS = -Iinclude
BIN = bin/tri
LIB = lib/libechanger.a

OBJ_MAIN = src/main.o src/triParMinSuccessif.o
OBJ_LIB = src/echanger.o

all: $(BIN)

$(BIN): $(LIB) $(OBJ_MAIN)
	$(CC) -o $@ $(OBJ_MAIN) -Llib -lechanger

$(LIB): $(OBJ_LIB)
	$(AR) $(ARFLAGS) $@ $^

%.o: %.c
	$(CC) $(CFLAGS) $(CPPFLAGS) -c -o $@ $<

clean:
	rm -f $(OBJ_MAIN) $(OBJ_LIB) $(BIN) $(LIB)
```

Dans la règle de construction de la bibliothèque, on a utilisé `$^`, qui représente l'ensemble des dépendances. Cette variable est pratique lorsqu'une cible dépend de plusieurs fichiers objets. La cible `clean` permet quant à elle de supprimer les fichiers générés pour repartir d'un état propre avant une recompilation.

Avec cette organisation, on obtient un mécanisme de compilation plus fiable et plus efficace que le script initial. La compilation devient incrémentale, la structure du projet est explicitée par des dépendances, et les options sont centralisées dans des variables, ce qui rend le projet plus simple à faire évoluer.