

Langage C

I. Introduction

- I.1. D'où vient le langage C?
- I.2. Pourquoi apprendre le langage C?
- I.3. Caractéristiques du langage C
- I.4. Du fichier source à l'exécutable

II. Notions de bases

- II.1. Structure générale d'un programme
- II.2. Les composants d'un programme en C
 - a) Les identificateurs
 - b) Les commentaires
 - c) Les mots réservés
 - d) Les opérateurs
 - e) Les délimiteurs
 - f) Les constantes
 - g) Les chaînes de caractères

Langage C

III. Les types prédéfinis

III.1. Le type caractère

III.2. Les types entiers

III.3. Les types flottants

III.4. Le type void

IV. L'expression et les opérateurs

IV.1. L'expression

IV.2. Les opérateurs

IV.2.1. L'affectation

IV.2.2. Les opérateurs arithmétiques

IV.2.3. Les opérateurs relationnels

IV.2.4. Les opérateurs logiques booléens

IV.2.5. Les opérateurs logiques bit à bit

IV.2.6. Les opérateurs d'affectation composée

IV.2.7. Les opérateurs d'incrément et de décrémentation

IV.2.8. L'opérateur virgule

IV.2.9. L'opérateur conditionnel ternaire

IV.2.10. L'opérateur de conversion de type

IV.2.11. L'opérateur adresse

Langage C

V. Les instructions

V.1. Les branchements conditionnels

V.2. Les boucles

V.3. Les branchements non conditionnels

VI. Les fonctions et les procédures

VI.1. Les fonctions

VI.2. Appel d'une fonction

VI.3. La fonction main

VI.4. Les fonctions imbriquées

VI.5. Les fonctions récursives

VI.6. Les procédures

VII. Les objets structurés

VII.1. Les tableaux

VII.2. Les structures

VII.3. Les champs de bit

VII.4. Les unions

VII.5. Les énumérations

VII.6. La directive *typedef*

Langage C

VIII. Les entrées/sorties de bas niveau

VIII.1. Notion de descripteur de fichier

VIII.2. Fonctions d'ouverture et de fermeture de fichier

VIII.3. Fonctions de lecture et d'écriture

VIII.4. Accès direct

VIII.5. Relation entre flot et descripteur de fichier

IX. Les bibliothèques

IX.1. Définition

IX.2. La librairie standard

X. Les pointeurs

X.1. Définition

X.2. Les pointeurs et les tableaux

X.3. Les pointeurs et les chaînes de caractères

X.4. Les pointeurs et les structures

X.5. Les pointeurs et les fonctions

X.6. Les opérations sur les pointeurs

Langage C

I. Introduction

I.1. D'où vient le langage C?

I.2. Pourquoi apprendre le langage C?

I.3. Caractéristiques du langage C

I.4. Du fichier source à l'exécutable

Langage C

I. Introduction

I.1. D'où vient le langage C

- Le langage C a été inventé au début des années 70 par Dennis RITCHIE, dans les laboratoires BELL AT&R, en s'inspirant d'un langage moins connu : le langage B.
- L'histoire du langage C est intrinsèquement liée à celle d'UNIX (Linux compris) et d'internet, de TCP/IP ...
- Le langage C est standardisé (le même pour tout le monde), et est normalisé par l'ANSI (American National Standard Institute) depuis 1982.
- Mise à part la déplorable tentative de Microsoft de le dénaturer (avec le C#), le langage C est et reste standard, unique et portable.
- Malgré l'apparition du langage C++ (orienté objet), les puristes et les perfectionnistes continuent essentiellement à développer en C, car ce langage est beaucoup plus rapide et portable (d'un ordinateur à un autre) que tout langage à objet (sauf peut-être java qui est extrêmement lent).

Langage C

I.2. Pourquoi apprendre le langage C

- Le langage C est la base syntaxique qui a inspirée de nombreux langages comme C++, Java, Perl, HTML, XML, Javascript ..., ce qui fait que, si vous maîtrisez le langage C, il vous sera aisé d'apprendre de nombreux autres langages.
- Tous les UNIX, tout Linux et tout le projet GNU, ainsi qu'internet, sont essentiellement fondés sur le langage C. L'avis de Kernighan sur la durabilité du langage C (plusieurs décennies, et ce n'est pas fini !-) semble la meilleure : Le langage C représente un très bon compromis entre la compréhension humaine, et l'interprétation par l'ordinateur (citation approximative).
- Il est important de noter qu'en langage C, on peut TOUT faire, sans véritables limitations, ce qui n'est pas toujours le cas dans les langages dits "objets".
- De plus, on peut tout faire de manière efficace et rapide, à la compilation comme à l'exécution.

Langage C

I.3. Caractéristiques du langage C

a) Universalité

Langage de programmation par excellence, le C n'est pas confiné à un domaine particulier d'applications. En effet, le C est utilisé dans l'écriture:

- de systèmes d'exploitations (comme Windows, UNIX et Linux) ou de machines virtuelles (JVMs, Runtimes et Frameworks .NET, logiciels de virtualisation, etc.)
- de logiciels de calcul scientifique, de modélisation mathématique ou de CFAO (Matlab, R, Labview, Scilab, etc.)
- de bases de données (MySQL, Oracle, etc.)
- d'applications en réseau (applications intranet, internet, etc.)
- de jeux vidéo (notamment avec OpenGL ou la SDL et/ou les différents moteurs (de jeu, physique ou 3D) associés)

Langage C

- d'assembleurs, compilateurs, débogueurs, interpréteurs, de logiciels utilitaires et dans bien d'autres domaines encore.

Oui, le C permet tout simplement de tout faire. A cause de son caractère proche du langage de la machine, le C est cependant peu productif, ce qui signifie qu'il faut souvent écrire beaucoup pour faire peu. C'est donc le prix à payer pour l'utilisation de ce langage surpuissant !

b) Concision, souplesse

C'est un langage concis, très expressif, et les programmes écrits dans ce langage sont très compacts grâce à un jeu d'opérateurs puissant.

Langage C

c) Puissance

Le C est un langage de haut niveau mais qui permet d'effectuer facilement des opérations de bas niveau et d'accéder aux fonctionnalités du système, ce qui est la plupart du temps impossible ou difficile à réaliser dans les autres langages de haut niveau. C'est d'ailleurs ce qui fait l'originalité du C, et aussi la raison pour laquelle il est autant utilisé dans les domaines où la performance est cruciale comme la programmation système, le calcul numérique ou encore l'informatique embarquée.

d) Portabilité

C'est un langage qui ne dépend d'aucune plateforme matérielle ou logicielle, c'est-à-dire qui est entièrement portable. De plus, de par sa simplicité, écrire un compilateur C pour un processeur donné n'est pas beaucoup plus compliqué que d'écrire un assembleur pour ce même processeur. Ainsi, là où l'on dispose d'un assembleur pour programmer, on dispose aussi généralement d'un compilateur C, d'où l'on dit également que le C est un "assembleur portable".

Langage C

e) Omniprésence

La popularité du langage mais surtout l'élégance des programmes écrits en C est telle que son style et sa syntaxe ont influencé de nombreux langages :

- C++ et Objective C sont directement descendus du C (on les considère souvent comme des extensions du C)
- C++ a influencé Java
- Java a influencé JavaScript
- PHP est un mélange de C et de langage de shell-scripting sous Linux
- C# est principalement un mix de C++ et de Java, deux langages de style C, avec quelques influences fonctionnelles d'autres langages comme Delphi et Visual Basic (le # de C# vient-il dit-on de deux "++" superposés)

Connaître le C est donc très clairement un atout majeur pour quiconque désire apprendre un de ces langages.

Langage C

I.4. Du fichier source à l'exécutable

Le C est un langage de programmation compilé, c'est à dire qu'il est traduit en langage machine par un compilateur, qui transforme le code source en code exécutable. D'autres langages (par ex. certains Basic) sont interprétés, et échappent à la phase de compilation: chaque instruction du code source est traduite en code exécutable juste avant son exécution.

Les langages compilés sont

- plus rapides (les instructions sont déjà en langage machine).
- moins souples: avant d'exécuter le programme, une compilation est nécessaire si le programme a été modifié.
- dépendants de la machine: la compilation est spécifique à chaque système (alias plateforme matérielle). Pour utiliser le programme sur un autre système, il faudra reprendre le code source, et le recompiler spécifiquement pour ce nouveau système.

Langage C

Les langages interprétés sont

- plus souples (pas besoin de compilation)
- souvent indépendants de la machine (le code source identique sera traduit différemment pour toutes les machines pour lesquelles un interpréteur de ce langage a été prévu).
- plus lents (l'interprétation en temps réel consomme de la puissance)

Le C est compilé donc rapide, mais assez portable (on parle de portabilité multi-plates-formes), ceci dans sa version normalisée ANSI (bien sûr après recompilation).

Langage C

Avant de définir la compilation, il y a deux expressions qu'il faut déjà avoir défini. Il s'agit de "fichier source" et "fichier exécutable".

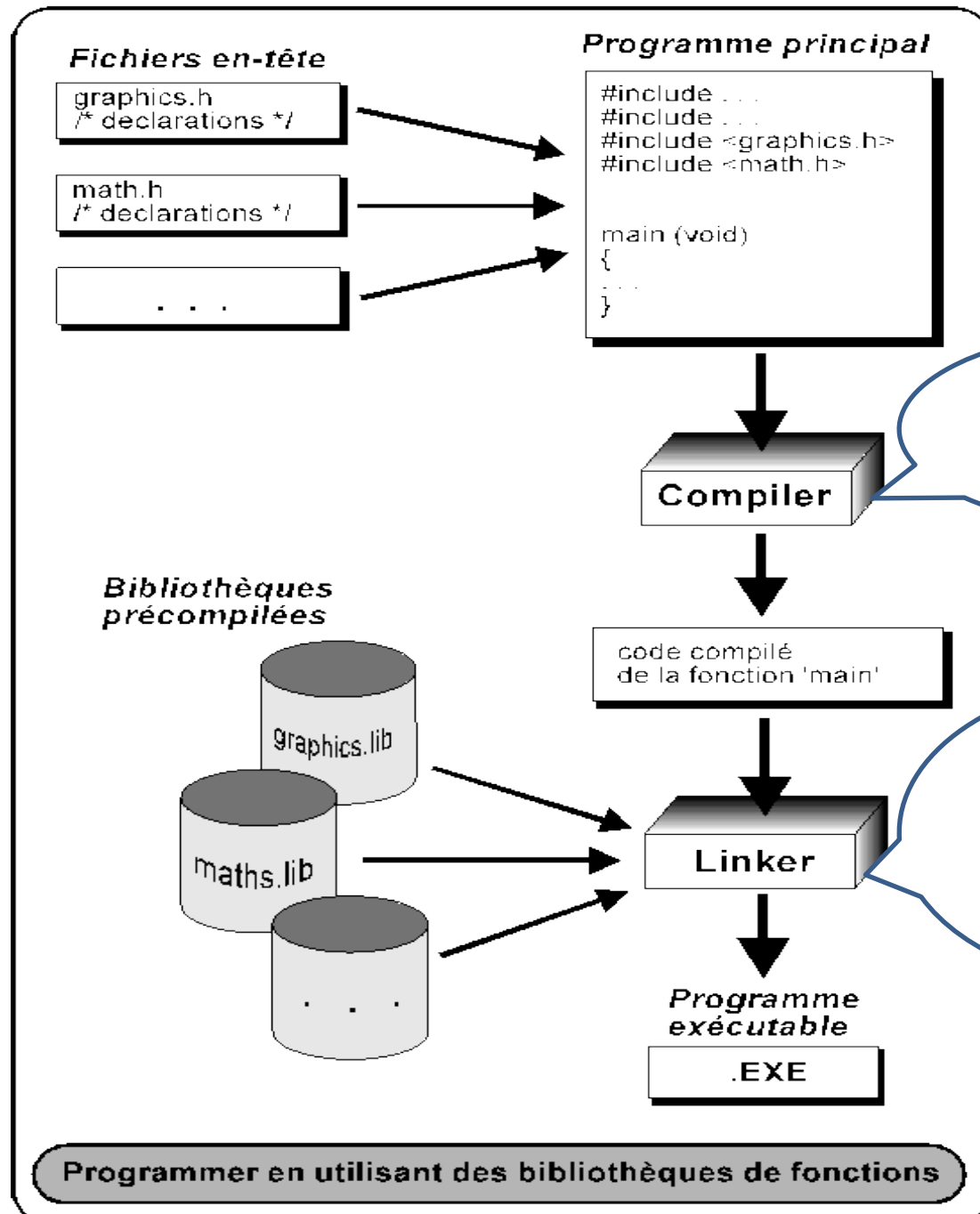
Un **fichier source** est un fichier contenant, dans le cas qui nous intéresse, des lignes de code écrits en C. Le code contenu dans le fichier source est appelé du **code source**. Les fichiers sources C portent généralement l'extension **".c"** (par exemple : hello.c).

Un **fichier exécutable** est un fichier contenant, entre autres, du code directement exécutable par le processeur. Ce code est appelé **code machine**. L'extension portée par les fichiers exécutables varient selon le système. Sous Windows par exemple, ils portent généralement l'extension **".exe"** (par exemple : hello.exe). Sous Linux, ils ne portent généralement pas d'extension.

Langage C

La **compilation** c'est, en première approximation, le processus pendant lequel le fichier source (qui ne contient que du texte) est transformé en fichier exécutable (qui contient du code compréhensible par le processeur).

Puisque les programmes C doivent être compilés avant de pouvoir être exécutés, on dit que le C est un **langage compilé**. Le programme qui effectue la compilation est appelé **compilateur**.



Langage C

Exécution d'un programme

1. La compilation

L'utilisation d'un langage évolué nécessite une traduction du programme source en langage machine. Le résultat est appelé un fichier exécutable. Cette « traduction » s'effectue en plusieurs étapes. Ces étapes sont décrites ici de manière succincte. **Le préprocesseur**

Le préprocesseur : le préprocesseur effectue des remplacements de texte dans le fichier comprenant le programme : il enlève les commentaires, remplace les parties de code si une instruction d'inclusion était demandée.

Langage C

La compilation

La compilation : la phase de compilation consiste en la génération d'un fichier objet (fichier codé en binaire selon le langage machine). Pour chaque fichier source, on obtient un fichier objet binaire. La compilation comprend une analyse lexicale (mots clés du langage), une analyse syntaxique (structure et grammaire du langage), et enfin la traduction en langage machine (parfois cette étape est décomposée en deux et passe par une phase de traduction en assembleur avant d'être codé en binaire).

Par abus de langage, on appelle compilation toute la phase de génération d'un fichier exécutable à partir des fichiers sources. Mais c'est seulement une des étapes menant à la création d'un exécutable.

L'édition de liens

L'édition de liens est la dernière étape et a pour but de réunir tous les éléments d'un programme. Les différents fichiers objets sont alors réunis, ainsi que les bibliothèques statiques, pour ne produire qu'un fichier exécutable.

Langage C

2.L'exécution

L'exécution d'un programme consiste à indiquer au système d'exploitation de l'ordinateur qu'il faut exécuter les instructions générées après les étapes de compilation. Selon le système d'exploitation utilisé (Windows, Unix, Linux.....) et le compilateur utilisé (VisualC++, GCC, ...), cette commande peut se faire de différentes façons (double clic sur une icône, ligne d'instruction dans une fenêtre de commande, appui sur un bouton....)

Langage C

Le compilateur C sous UNIX s'appelle **cc**. On utilisera de préférence le compilateur **gcc** du projet GNU. Ce compilateur est livré gratuitement avec sa documentation et ses sources. Par défaut, **gcc** active toutes les étapes de la compilation. On le lance par la commande

gcc [options] fichier.c [-llibrairies]

Par défaut, le fichier exécutable s'appelle *a.out*. Le nom de l'exécutable peut être modifié à l'aide de l'option **-o**.

Les éventuelles librairies sont déclarées par la chaîne **-llibrairie**. Dans ce cas, le système recherche le fichier **liblibrairie.a** dans le répertoire contenant les librairies pré-compilées (généralement **/usr/lib/**). Par exemple, pour lier le programme avec la librairie mathématique, on spécifie **-lm**. Le fichier objet correspondant est **libm.a**. Lorsque les librairies pré-compilées ne se trouvent pas dans le répertoire usuel, on spécifie leur chemin d'accès par l'option **-L**.

Langage C

Le compilateur C sous UNIX s'appelle **cc**. On utilisera de préférence le compilateur **gcc** du projet GNU. Ce compilateur est livré gratuitement avec sa documentation et ses sources. Par défaut, **gcc** active toutes les étapes de la compilation. On le lance par la commande

gcc [options] fichier.c [-llibrairies]

Par défaut, le fichier exécutable s'appelle *a.out*. Le nom de l'exécutable peut être modifié à l'aide de l'option **-o**.

Les éventuelles librairies sont déclarées par la chaîne **-llibrairie**. Dans ce cas, le système recherche le fichier **liblibrairie.a** dans le répertoire contenant les librairies pré-compilées (généralement **/usr/lib/**). Par exemple, pour lier le programme avec la librairie mathématique, on spécifie **-lm**. Le fichier objet correspondant est **libm.a**. Lorsque les librairies pré-compilées ne se trouvent pas dans le répertoire usuel, on spécifie leur chemin d'accès par l'option **-L**.

Langage C

Les options les plus importantes du compilateur **gcc** sont les suivantes :

- c : supprime l'édition de liens ; produit un fichier objet.
- E : n'active que le préprocesseur (le résultat est envoyé sur la sortie standard).
- g : produit des informations symboliques nécessaires au débogueur.
- I *nom-de-répertoire* : spécifie le répertoire dans lequel doivent être recherchés les fichiers en-têtes à inclure (en plus du répertoire courant).
- L *nom-de-répertoire* : spécifie le répertoire dans lequel doivent être recherchées les bibliothèques précompilées (en plus du répertoire usuel).
- o *nom-de-fichier* : spécifie le nom du fichier produit. Par défaut, le exécutable fichier s'appelle a.out.
- O, -O1, -O2, -O3 : options d'optimisations. Sans ces options, le but du compilateur est de minimiser le coût de la compilation. En rajoutant l'une de ces options, le compilateur tente de réduire la taille du code exécutable et le temps d'exécution. Les options correspondent à différents niveaux d'optimisation : -O1 (similaire à -O) correspond à une faible optimisation, -O3 à l'optimisation maximale.
- S : n'active que le préprocesseur et le compilateur ; produit un fichier assembleur.
- v : imprime la liste des commandes exécutées par les différentes étapes de la compilation.
- W : imprime des messages d'avertissement (warning) supplémentaires.
- Wall : imprime tous les messages d'avertissement.

Langage C

Nombreux programmeurs utilisent cependant un **EDI** (Environnement de Développement Intégré) au lieu d'un simple compilateur pour compiler leurs programmes. Un EDI est un logiciel qui intègre un éditeur de texte pour taper les codes sources, un compilateur pour les traduire en exécutable ainsi que d'autres outils aidant à la mise au point et à la distribution des programmes. Il existe de nombreux EDIs pour le langage C. Certains sont gratuits (Code::Blocks Studio, Visual C++ Express, etc.), d'autres payants (CodeGear RAD Studio, Microsoft Visual Studio, etc.).

Langage C

Télécharger Code::Blocks ➔ <http://www.codeblocks.org/downloads/binaries>

Download binary - Windows Internet Explorer

http://www.codeblocks.org/downloads/binaries

Code::Blocks

Code::Blocks - The IDE with all the features you need, having a consistent look, feel and operation across platforms.

Home Features Downloads Forums Wiki

Main

- Home
- Features
- Screenshots
- Downloads
 - Binaries
 - Source
 - SVN
- Plugins
- User manual
- Licensing
- Donations

Quick links

- FAQ
- Forums
- Wiki
- Nightlies
- BugTracker
- PatchTracker
- Browse SVN
- Browse SVN log

Please select a setup package depending on your platform:

- Windows 2000/XP/Vista/7
- Linux 32-bit
- Linux 64-bit
- Mac OS X

NOTE: There are also more recent nightly builds available in the **forums** or (for Debian users) in **Jens' repository**.

IMPORTANT NOTE: If you try to download from BerliOS and get a "Too many clients" - error, you should retry to download the file. According to a BerliOS - admin, this can happen several times, before the download starts.
If it still does not work, search our **forum** for "alternate mirror", you will find at least alternatives for the windows and debian downloads.

Windows 2000 / XP / Vista / 7:

File	Date	Size	Download from
codeblocks-10.05-setup.exe	27 May 2010	23.3 MB	BerliOS or Sourceforge.net
codeblocks-10.05mingw-setup.exe	27 May 2010	74.0 MB	BerliOS or Sourceforge.net

NOTE: The codeblocks-10.05mingw-setup.exe file includes the GCC compiler and GDB debugger from MinGW.

Internet | Mode protégé : activé

FR 22:18 13/02/2011

Langage C

II. Notions de bases

II.1. Structure générale d'un programme

II.2. Les composants d'un programme en C

a) Les identificateurs

b) Les commentaires

c) Les mots réservés

d) Les opérateurs

e) Les délimiteurs

f) Les constantes

g) Les chaînes de caractères

Langage C

II. Notions de base

II.1. Structure générale d'un programme

Chaque fichier source entrant dans la composition d'un programme exécutable est fait d'une succession d'un nombre quelconque d'éléments indépendants, qui sont :

- des directives pour le préprocesseur (lignes commençant par#),
- des constructions de types (struct, union, enum, typedef),
- des déclarations de variables et de fonctions externes,
- des définitions de variables et
- des définitions de fonctions.

Un programme C se présente de la façon suivante :

```
[directives au préprocesseur]
[déclarations de variables externes]
[fonctions secondaires]
main()
{
    déclarations de variables internes
    instructions
}
```

Langage C

Seules les expressions des deux dernières catégories (variables et fonctions) font grossir le fichier objet: les définitions de fonctions laissent leur traduction en langage machine, tandis que les définitions de variables se traduisent par des réservations d'espace, éventuellement garni de valeurs initiales. Les autres directives et déclarations s'adressent au compilateur et il n'en reste pas de trace lorsque la compilation est finie.

En C on n'a donc pas une structure syntaxique englobant tout, comme la construction «Program...end.» du langage Pascal; un programme n'est qu'une collection de fonctions assortie d'un ensemble de variables globales.

D'où la question: par où l'exécution doit-elle commencer? La règle généralement suivie par l'éditeur de liens est la suivante: parmi les fonctions données il doit en exister une dont le nom est *main*. C'est par elle que l'exécution commencera; le lancement du programme équivaut à l'appel de cette fonction par le système d'exploitation.

Notez bien que, à part cela, *main* est une fonction comme les autres, sans aucune autre propriété spécifique; en particulier, les variables internes à *main* sont locales, tout comme celles des autres fonctions.

Langage C

Pour finir cette entrée en matière, voici la version C du célèbre programme-qui-dit-bonjour, sans lequel on ne saurait commencer un cours de programmation:

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

```
int main() {  
    printf("Bonjour\n");  
    return 0;  
}
```

Langage C

II.2. Les composants d'un programme en C

a) Les identificateurs

Ils permettent de référencer les différents objets du C:

- Constantes
- Variables
- Les fonctions.

Les noms des fonctions et des variables en C sont composés d'une suite de lettres et de chiffres. Le premier caractère doit être une lettre. Le symbole '_' est aussi considéré comme une lettre.

* L'ensemble des symboles utilisables est donc:

{0,1,2,...,9,A,B,...,Z,_,a,b,...,z}

* Le premier caractère doit être une lettre (ou le symbole '_')

* C distingue les majuscules et les minuscules, ainsi:

'Nom_de_variable' est différent de **'nom_de_variable'**

* La longueur des identificateurs n'est pas limitée, mais C distingue 'seulement' les 31 premiers caractères.

Langage C

Remarques:

- Il est déconseillé d'utiliser le symbole '_' comme premier caractère pour un identificateur, car il est souvent employé pour définir les variables globales de l'environnement C.
- Le standard dit que la validité de noms externes (p.ex. noms de fonctions ou var. globales) peut être limité à 6 caractères (même sans tenir compte des majuscules et minuscules) par l'implémentation du compilateur, mais tous les compilateurs modernes distinguent au moins 31 caractères de façon à ce que nous pouvons généraliser qu'en pratique les règles ci-dessus s'appliquent à tous les identificateurs.

b) Les commentaires

Un commentaire commence toujours par les deux symboles '/*' et se termine par les symboles '*/'. Il est interdit d'utiliser des commentaires imbriqués.

Exemples

/* Ceci est un commentaire correct */

/* Ceci est /* évidemment */ défendu */

Langage C

c) Les mots réservés ou mots-clés

Symboles terminaux du langage, ils sont nécessaires à la sémantique du langage.

- spécificateur de type d'objet : int, char, float, double, struct ...
- spécificateur de classe d'allocation d'un objet : auto, extern, typedef...
- opérateurs symboliques : if, else, while, switch...
- étiquettes : default...

d) Les opérateurs

- L'affectation
- Les opérateurs arithmétiques
- Les opérateurs relationnels
- Les opérateurs logiques booléens
- Les opérateurs logiques bit à bit
- Les opérateurs d'affectation composée
- Les opérateurs d'incrément et de décrémentation
- L'opérateur virgule
- L'opérateur conditionnel ternaire
- L'opérateur de conversion de type
- L'opérateur adresse

Langage C

e) Les délimiteurs ou les signes de ponctuation

- ;** : termine une déclaration de variables ou une instruction
- ,** : sépare deux éléments dans une liste
- ()** : encadre une liste d'arguments ou de paramètres
- []** : encadre une dimension ou l'indice d'un tableau
- { }** : encadre un bloc d'instructions ou une liste de valeurs d'initialisations

f) Les constantes

Une constante est une valeur qui apparaît littéralement dans le code source d'un programme, le type de la constante étant déterminé par la façon dont la constante est écrite. Les constantes peuvent être de 4 types : entier, flottant (nombre réel), caractère, et des constantes nommées.

Par exemple:

const int n=10;

Langage C

Ces constantes nommées sont données soit par le préprocesseur soit par des énumérations.

1) Les **#define**

Lorsque le préprocesseur lit une ligne du type

#define identificateur reste-de-la-ligne

il remplace dans toute la suite du source, toute nouvelle occurrence de *identificateur* par *reste-de-ligne*.

Par exemple: ***#define PI 3.1415***

on peut donc utiliser le nom PI pour désigner la constante 3.1415

Le préprocesseur ne compile pas, il fait des transformations d'ordre purement textuel.

Par exemple si on écrit ***#define PI 3.1415;***

Faites attention, car le préprocesseur va remplacer PI/2 par 3.1415;/2 et là une erreur va être générée à l'instruction PI/2 qui paraît correct.

Langage C

2) Les énumérations

On peut définir des constantes de la manière suivante:

enum { liste d'identificateurs};

Par exemple:

enum {Janvier, Février, Mars, Avril, Mai, Juin, Juillet, Aout, Septembre, Octobre, Novembre, Décembre};

Ces identificateurs sont des constantes de type *int*, et leur donné les valeurs 0,1,2, ..,11.

g) Les chaines de caractères

Une chaîne de caractères est une suite de caractères entourés par des guillemets. Par exemple, "Ceci est une chaîne de caractères ".

Langage C

III. Les types prédéfinis

III.1. Le type caractère

III.2. Les types entiers

III.3. Les types flottants

III.4. Le type void

Langage C

III. Les types prédéfinis

Au niveau du processeur, toutes les données sont représentées sous leur forme binaire et la notion de **type n'a pas** de sens. Cette notion n'a été introduite que par les langages de haut niveau dans le but de rendre les programmes plus rationnels et structurés. Mais même parmi les langages de haut niveau, on distingue ceux qui sont dits fortement typés (qui offrent une grande variété de types), comme le Pascal par exemple, de ceux qui sont dits faiblement ou moyennement typés (et plus proches de la machine) comme le C par exemple.

En résumé, le C est un langage typé. Cela signifie en particulier que toute variable, constante ou fonction est d'un type précis. Le type d'un objet définit la façon dont il est représenté en mémoire.

La mémoire de l'ordinateur se décompose en une suite continue d'octets. Chaque octet de la mémoire est caractérisé par son *adresse*, qui est un entier. Deux octets contigus en mémoire ont des adresses qui diffèrent d'une unité. Quand une variable est définie, il lui est attribué une adresse. Cette variable correspondra à une zone mémoire dont la longueur (le nombre d'octets) est fixée par le type.

Langage C

La taille mémoire correspondant aux différents types dépend des compilateurs; toutefois, la norme ANSI spécifie un certain nombre de contraintes.

Les types de base en C concernent les caractères, les entiers et les flottants (nombres réels). Ils sont désignés par les mots-clefs suivants:

<u>Type C</u>	<u>Type correspondant</u>
<i>char</i>	caractère (entier de petite taille)
<i>int</i>	entier
<i>float</i>	nombre flottant (réel) en simple précision
<i>double</i>	nombre flottant (réel) en double précision

Devant *char* ou *int*, on peut mettre le modificateur *signed* ou *unsigned* selon que l'on veut avoir un entier signé (par défaut) ou non signé.

Langage C

Types de base

- *Nombres entiers*

- Anonymes

- Petite taille

- signés
 - non signés

char
unsigned char

- Taille moyenne

- signés
 - non signés

short
unsigned short

- Grande taille

- signés
 - non signés

long
unsigned long

- Nommés

enum

- *Nombres flottants*

- Simple

float

- Grande précision

double

- Précision encore plus grande

long double

Langage C

La classification des types numériques obéit à deux critères :

- Si on cherche à représenter un ensemble de nombres tous positifs on pourra adopter un type non signé;
au contraire si on doit représenter un ensemble contenant des nombres positifs et des nombres négatifs on devra utiliser un type signé.
- Le deuxième critère de classification des données numériques est la taille requise par leur représentation.

Comme précédemment, c'est un attribut d'un ensemble, et donc d'une variable devant représenter tout élément de l'ensemble, non d'une valeur particulière.

Par exemple, le nombre 123 considéré comme un élément de l'ensemble $\{0 \dots 65535\}$ est plus encombrant que le même nombre 123 quand il est considéré comme un élément de l'ensemble $\{0 \dots 255\}$.

Langage C

A propos des booléens: En C il n'existe donc pas de type booléen spécifique. Il faut savoir qu'à tout endroit où une expression booléenne est requise (typiquement, dans des instructions comme if ou while) on peut faire figurer n'importe quelle expression; elle sera tenue pour vraie si elle est non nulle, elle sera considérée fausse sinon.

Ainsi, dans un contexte conditionnel,

<i>expr</i>	(c'est-à-dire <i>expr</i> <<vraie>>) équivaut à
<i>expr</i> != 0	(<i>expr</i> différente de 0).

Inversement, lorsqu'un opérateur (égalité, comparaison, etc.) produit une valeur booléenne, il rend 0 pour faux et 1 pour vrai.

Signalons aux esthètes que le fichier <types.h> comporte les déclarations :

```
enum{ false, true };  
typedef unsigned char Boolean;
```

qui introduisent la constante false valant 0, la constante true valant 1 et le type Boolean comme le type le moins encombrant dans lequel on peut représenter ces deux valeurs.

Langage C

III.1. Le type caractère

Un objet de type *char* peut être défini, au choix, comme:

- un nombre entier pouvant représenter n'importe quel caractère du jeu de caractères de la machine utilisée;
- un nombre entier occupant la plus petite cellule de mémoire adressable séparément.

Sur les machines actuelles les plus répandues cela signifie généralement un octet (8 bits). Le plus souvent, un *char* est un entier signé; un *unsigned char* est alors un entier non signé. Lorsque les *char* sont par défaut non signés, la norme ANSI prévoit la possibilité de déclarer des *signed char*.

Les types *char* sont des sous ensembles des types entier.

Un caractère est assimilé à son code ASCII.

Langage C

Une constante de type caractère se note en écrivant le caractère entre apostrophes.

Une constante de type chaîne de caractères se note en écrivant ses caractères entre guillemets.

Exemples:

- trois caractères : 'A' '2' '''
- Quatre chaînes de caractères : "A" "Bonjour à tous !" "" ""

Une des particularités du type *char* en C est qu'il peut être assimilé à un entier : tout objet de type *char* peut être utilisé dans une expression qui utilise des objets de type entier.

Par exemple, si *c* est de type *char*, l'expression *c + 1* est valide. Elle désigne le caractère suivant dans le code ASCII. Ainsi, le programme suivant imprime le caractère 'B'.

```
main() {  
    char c = 'A';  
    printf("%c", c + 1);  
}
```

Langage C

Certain caractères non affichables peuvent être représentés par des séquences particulières de caractères, toujours précédés de la barre gauche (\).

Nous connaissons déjà \n (new line) qui est le retour à la ligne et \t qui commande un tabulation horizontale. Il en existe d'autres:

Séquence d'échappement	Code ASCII	Signification
\a	7	sonnerie
\b	8	retour arrière
\t	9	tabulation horizontale
\n	10	retour à la ligne
\v	11	tabulation verticale
\f	12	nouvelle page
\r	13	retour chariot
\"	34	guillemet anglais
\'	39	apostrophe
\?	63	point d'interrogation
\\	92	barre gauche
\0	0	caractère nul

Langage C

III.2. Les types entiers

En principe, le type *int* correspond à la taille d'entier la plus efficace, c'est-à-dire la plus adaptée à la machine utilisée. Sur certains systèmes et compilateurs *int* est synonyme de short, sur d'autres il est synonyme de long.

Le type *int* peut donc poser un problème de portabilité: le même programme, compilé sur deux machines distinctes, peut avoir des comportements différents. D'où un conseil important: n'utilisez le type *int* que pour des variables locales destinées à contenir des valeurs raisonnablement petites (inférieures en valeur absolue à 32767). Dans les autres cas il vaut mieux expliciter *char*, *short* ou *long* selon le besoin.

Les trois types de variables entières:

- le type *short* (*signed* ou *unsigned*) qui est codé avec 2 octets
- le type *int* (*signed* ou *unsigned*) qui est codé avec 2 ou 4 octets,
- le type *long* (*signed* ou *unsigned*) qui est codé avec 4 ou 8 octets.

Langage C

la taille exacte des données de ces types n'est pas fixée par la norme du langage. Sur une machine 16 bits, le type *int* est codé sur 2 octets, sur une machine 32 bits actuelle, il est codé sur 4 octets, et sur les futures architectures 64 bits, il pourra atteindre 8 octets.

De nos jours on trouve souvent :

- unsigned short* : 16 bits pour représenter un nombre entier compris entre 0 et 65.535
- Short* : 16 bits pour représenter un nombre entier compris entre -32.768 et 32.767
- unsigned long* : 32 bits pour représenter un nombre entier entre 0 et 4.294.967.296
- long* : 32 bits pour représenter un entier entre -2.147.483.648 et 2.147.483.647

Plus généralement, les valeurs maximales et minimales des différents types entiers sont définies dans la librairie standard *limits.h*.

Langage C

III.3. Les types flottants

La norme ANSI prévoit trois types de nombres flottants: *float* (simple précision), *double* (double précision) et *long double* (précision étendue dans certains compilateurs). La norme ne spécifie pas les caractéristiques de tous ces types. Il est garanti que toute valeur représentable dans le type *float* est représentable sans perte d'information dans le type *double*, et toute valeur représentable dans le type *double* l'est dans le type *long double*.

Les nombres réels sont des nombres à virgule flottante, c'est-à-dire des nombres dans lesquels la position de la virgule n'est pas fixe. Une partie des bits d'un réel sert à coder l'exposant, le reste des bits permettent de coder le nombre sans virgule, la mantisse.

Dans le code source nous écrirons par exemple : 3.4e-38 sur une seule ligne pour un littéral réel.

Langage C

En résumé, voici un tableau récapitulatif des types simples:

Type de donnée	Signification	Taille (en octets)	Plage des littéraux
char	Caractère	1	-128 à 127
unsigned char	Caractère non signé	1	0 à 255
short int	Entier court	2	-32 768 à 32 767
unsigned short int	Entier court non signé	2	0 à 65 535
int	Entier	2 (sur processeur 16 bits) 4 (sur processeur 32 bits)	-32 768 à 32 767 -2 147 483 648 à 2 147 483 647
unsigned int	Entier non signé	2 (sur processeur 16 bits) 4 (sur processeur 32 bits)	0 à 65 535 0 à 4 294 967 295
long int	Entier long	4	-2 147 483 648 à 2 147 483 647
unsigned long int	Entier long non signé	4	0 à 4 294 967 295
float	Flottant (réel)	4	3.4×10^{-38} à 3.4×10^{38}
double	Flottant double	8	1.7×10^{-308} à 1.7×10^{308}

Langage C

III.4. Le type *void*

Le type ***void*** est le type de résultat d'une fonction qui ne produit aucun résultat.

Par exemple:

```
void main()  
{  
    ...  
}
```

Il est impossible de déclarer directement une variable ou une constante de type ***void***.

Langage C

IV. L'expression et les opérateurs

IV.1. L'expression

IV.2. Les opérateurs

IV.2.1. L'affectation

IV.2.2. Les opérateurs arithmétiques

IV.2.3. Les opérateurs relationnels

IV.2.4. Les opérateurs logiques booléens

IV.2.5. Les opérateurs logiques bit à bit

IV.2.6. Les opérateurs d'affectation composée

IV.2.7. Les opérateurs d'incrémentement et de décrémentement

IV.2.8. L'opérateur virgule

IV.2.9. L'opérateur conditionnel ternaire

IV.2.10. L'opérateur de conversion de type

IV.2.11. L'opérateur adresse

Langage C

IV.1. L'expression

- Une expression représente une donnée qui possède une valeur. On parle d'évaluation d'une expression.
- Une expression peut être simplement un identificateur de constante, un identificateur de variable, ou un littéral constant.
- Elle peut être également un ensemble d'identificateurs de constantes, de variables, ou des littéraux constants reliés par des opérateurs.

Langage C

IV.2. Les opérateurs

IV.2.1. L'affectation

En C, l'affectation est un opérateur à part entière. Elle est symbolisée par le signe **=**.

Sa syntaxe est la suivante : ***variable = expression***

Le terme de gauche de l'affectation peut être une variable simple, un élément de tableau mais pas une constante. Cette expression a pour effet d'évaluer *expression* et d'affecter la valeur obtenue à *variable*. De plus, cette expression possède une valeur, qui est celle *expression*. Ainsi, l'expression ***i = 5*** vaut 5.

L'affectation effectue une *conversion de type implicite* : la valeur de l'expression (terme de droite) est convertie dans le type du terme de gauche.

Langage C

Par exemple, le programme suivant:

```
main() {  
    int i, j = 2;  
    float x = 2.5;  
    i = j + x;  
    x = x + i;  
    printf("|n %f |n",x);  
}
```

imprime pour x la valeur 6.5 (et non 7), car dans l'instruction $i = j + x$, l'expression $j + x$ a été convertie en entier.

Langage C

IV.2.2. Les opérateurs arithmétiques

Les opérateurs arithmétiques classiques sont:

- l'opérateur unaire - (changement de signe)
- ainsi que les opérateurs binaires:
 - + addition
 - soustraction
 - * multiplication
 - / division
 - % reste de la division (modulo)

Ces opérateurs agissent de la façon attendue sur les entiers comme sur les flottants.

Leurs seules spécificités sont les suivantes : Contrairement à d'autres langages, le C ne dispose que de la notation / pour désigner à la fois la division entière et la division entre flottants.

Si les deux opérandes sont de type entier, l'opérateur / produira une division entière (quotient de la division). Par contre, il délivrera une valeur flottante dès que l'un des opérandes est un flottant.

Langage C

Par exemple,

```
float x;
```

```
x = 3 / 2;
```

affecte à x la valeur 1.

Par contre

```
x = 3 / 2.0;
```

affecte à x la valeur 1.5.

L'opérateur % ne s'applique qu'à des opérandes de type entier. Si l'un des deux opérandes est négatif, le signe du reste dépend de l'implémentation, mais il est en général le même que celui du dividende.

Notons enfin qu'il n'y a pas en C d'opérateur effectuant l'élévation à la puissance. De façon générale, il faut utiliser la fonction ***pow(x,y)*** de la librairie ***math.h*** pour calculer x^y .

Langage C

IV.2.3. Les opérateurs relationnels (de comparaison)

- `>` strictement supérieur
- `>=` supérieur ou égal
- `<` strictement inférieur
- `<=` inférieur ou égal
- `==` égal
- `!=` différent

Leur syntaxe est ***expression1 op expression2***

Les deux expressions sont évaluées puis comparées. La valeur rendue est de type *int* (il n'y a pas de type booléen en C); elle vaut 1 si la condition est vraie, et 0 sinon.

Attention à ne pas confondre l'opérateur de test d'égalité `==` avec l'opérateur d'affectation `=`.

Langage C

Ainsi, le programme:

```
main() {  
    int a = 0;  
    int b = 1;  
    if (a = b)  
        printf("|n a et b sont egaux |n");  
    else  
        printf("|n a et b sont differents |n");  
}
```

imprime à l'écran a et b sont égaux !

Langage C

IV.2.4. Les opérateurs logiques booléens

&&	et logique
	ou logique
!	négation logique

Comme pour les opérateurs de comparaison, la valeur retournée par ces opérateurs est un *int* qui vaut 1 si la condition est vraie et 0 sinon.

Dans une expression de type *expression1 op1 expression2 op2 ...expressionn* l'évaluation se fait de gauche à droite et s'arrête dès que le résultat final est déterminé.

Par exemple dans

```
int i;  
int p[10];  
if ((i >= 0) && (i <= 9) && !(p[i] == 0))  
...
```

la dernière clause ne sera pas évaluée si *i* n'est pas entre 0 et 9.

Langage C

IV.2.5. Les opérateurs logiques bit à bit

Les six opérateurs suivants permettent de manipuler des entiers au niveau du bit. Ils s'appliquent aux entiers de toute longueur (short, int ou long), signés ou non.

&	et
	ou inclusif
^	ou exclusif
~	complément à 1
<<	décalage à gauche
>>	décalage à droite

Considérons par exemple les entiers $a=77$ et $b=23$ de type *unsigned char* (*i.e.* 8 bits). En base 2 il s'écrivent respectivement 01001101 et 00010111.

$a = 01001101$	➔ valeur décimale = 77
$b = 00010111$	➔ valeur décimale = 23
$a \& b = 00000101$	➔ valeur décimale = 5
$\sim a = 10110010$	➔ valeur décimale = 178
$b \ll 5 = 11100000$	➔ valeur décimale = 112
$b \gg 1 = 00001011$	➔ valeur décimale = 11

Langage C

IV.2.6. Les opérateurs d'affectation composée

Les opérateurs d'affectation composée sont

+=	-=	*=	/=	%=
&=	^=	=	<<=	>>=

Pour tout opérateur *op*, l'expression

expression1 op= expression2

est équivalente à

expression1 = expression1 op expression2

Toutefois, avec l'affectation composée, *expression1* n'est évaluée qu'une seule fois.

Langage C

IV.2.7. Les opérateurs d'incrémentation et de décrémentation

Les opérateurs

d'incrémentation **++**

et de décrémentation **--**

s'utilisent aussi bien en suffixe (**i++**) qu'en préfixe (**++i**). Dans les deux cas la variable **i** sera incrémentée, toutefois dans la notation suffixe la valeur retournée sera l'ancienne valeur de **i** alors que dans la notation préfixe se sera la nouvelle.

Par exemple,

```
int a = 3, b, c;
```

```
b = ++a;                /* a et b valent 4 */
```

```
c = b++;               /* c vaut 4 et b vaut 5 */
```

Langage C

IV.2.8. L'opérateur virgule

Une expression peut être constituée d'une suite d'expressions séparées par des virgules :

expression1, expression2, ... , expressionn

Cette expression est alors évaluée de gauche à droite. Sa valeur sera la valeur de l'expression de droite.

Par exemple, le programme

```
main() {  
    int a, b;  
    b = ((a = 3), (a + 2));  
    printf("|n b = %d |n",b);  
}
```

imprime b = 5.

Langage C

La virgule séparant les arguments d'une fonction ou les déclarations de variables n'est pas l'opérateur virgule. En particulier l'évaluation de gauche à droite n'est pas garantie.

Par exemple l'instruction composée

```
{      int a=1;  
  printf("|%d |%d",++a,a);  
}
```

(compilée avec gcc) produira la sortie:

2 1 sur un PC Intel/Linux

et la sortie 2 2 sur un DEC Alpha/OSF1.

Langage C

IV.2.9. L'opérateur conditionnel ternaire

L'opérateur conditionnel `?` est un opérateur ternaire. Sa syntaxe est la suivante :
condition ? expression1 : expression2

Cette expression est égale à
expression1 si *condition* est satisfaite, et à *expression2* sinon.

Par exemple, l'expression
`x >= 0 ? x : -x` correspond à la valeur absolue d'un nombre.

De même l'instruction
`m = ((a > b) ? a : b);` affecte à m le maximum de a et de b.

Langage C

IV.2.10. L'opérateur de conversion de type

L'opérateur de conversion de type, appelé ***cast***, permet de modifier explicitement le type d'un objet. On écrit ***(type) objet***

Par exemple,

```
main() {  
    int i = 3, j = 2;  
    printf("%f \n", (float)i/j);  
}
```

retourne la valeur 1.5.

IV.2.11. L'opérateur adresse

L'opérateur d'adresse ***&*** appliqué à une variable retourne l'adresse mémoire de cette variable. La syntaxe est ***&objet***

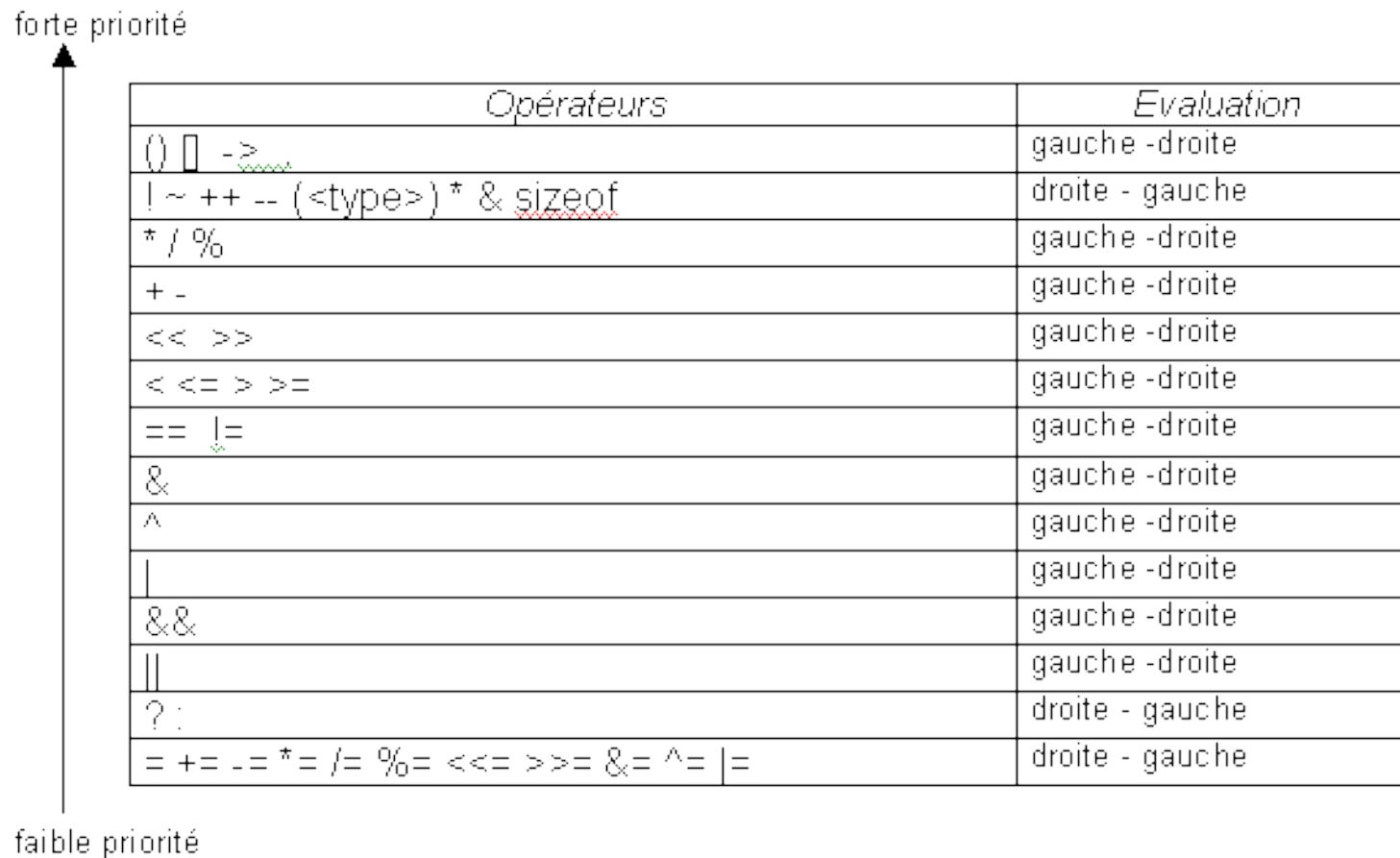
Langage C

Classes de priorités

Priorité 1 (la plus forte)	: ()								
Priorité 2	: !	++	--						
Priorité 3	: *	/	%						
Priorité 4	: +	-							
Priorité 5	: < <=	> >=							
Priorité 6	: ==	!=							
Priorité 7	: &&								
Priorité 8	:								
Priorité 9 (la plus faible)	: =	+=	-=	*=	/=	%=			

Langage C

Ordre de priorité des opérateurs



Opérateurs	Evaluation
() [] ->	gauche - droite
! ~ ++ -- (<type>)* & sizeof	droite - gauche
* / %	gauche - droite
+ -	gauche - droite
<< >>	gauche - droite
< <= > >=	gauche - droite
== !=	gauche - droite
&	gauche - droite
^	gauche - droite
	gauche - droite
&&	gauche - droite
	gauche - droite
? :	droite - gauche
= += -= *= /= %= <<= >>= &= ^= =	droite - gauche

pour ceux de même niveau => de gauche à droite.

Langage C

V. Les instructions

V.1. Les branchements conditionnels

V.2. Les boucles

V.3. Les branchements non conditionnels

Langage C

V. Les instructions

On appelle instruction de contrôle toute instruction qui permet de contrôler le fonctionnement d'un programme.

V.1. Les branchements conditionnels

1) Les branchements conditionnels **if---else**

La forme la plus générale est celle-ci :

if (expression_1)

instruction_1

else

if (expression_2)

instruction_2

...

else

if (expression_n)

instruction_n

else

instruction_∞

Langage C

avec un nombre quelconque de *else if* (...). Le dernier *else* est toujours facultatif.

La forme la plus simple est:

if (expression)
instruction

Chaque *instruction* peut être un bloc d'instructions.

Langage C

2) Branchement multiple **switch**

L'instruction « switch » permet d'effectuer une suite de tests d'égalité consécutifs pour une valeur donnée et de déclencher des instructions selon la valeur.

```
switch (expression )  
    {case constante_1:  
        liste d'instructions 1  
        break;  
    case constante_2:  
        liste d'instructions 2  
        break;  
    ...  
    case constante_n:  
        liste d'instructions n  
        break;  
    default:  
        liste d'instructions ∞  
        break;  
    }
```

Langage C

Si la valeur de *expression* est égale à l'une des *constantes*, la *liste d'instructions* correspondant est exécutée. Sinon la *liste d'instructions* ∞ correspondant à default est exécutée. L'instruction default est facultative.

Exemple:

```
int main()
{
    int c;
    printf("\n entrer un nombre ");
    scanf("%d",&c);
    switch (c)
    {
        case 1: printf("\n un"); break;
        case 2 : printf("\n deux"); break;
        case 3 : printf("\n trois"); break;
        default : printf("\n autre chose");
    }
    printf("\n");
}
```

Langage C

V.2. Les boucles

Les *boucles* permettent de répéter une série d'instructions tant qu'une certaine condition n'est pas vérifiée.

1) Boucle **while**

La syntaxe de *while* est la suivante :

while (expression)
instruction

Tant que *expression* est vérifiée (*i.e.*, non nulle), *instruction* est exécutée. Si *expression* est nulle au départ, *instruction* ne sera jamais exécutée. *instruction* peut évidemment être une instruction composée.

Par exemple, le programme suivant imprime les entiers de 1 à 9.

```
i = 1;  
while (i < 10)  
{  
    printf("|n i = %d",i);  
    i++;  
}
```


Langage C

2) Boucle **do---while**

Il peut arriver que l'on ne veuille effectuer le test de continuation qu'après avoir exécuté l'instruction. Dans ce cas, on utilise la boucle *do---while*. Sa syntaxe est:

```
do  
    instruction  
while (expression );
```

Ici, *instruction* sera exécutée tant que *expression* est non nulle. Cela signifie donc que *instruction* est toujours exécutée au moins une fois. Par exemple, pour saisir au clavier un entier entre 1 et 10 :

```
int a;  
do  
{  
    printf("\n Entrez un entier entre 1 et 10 : ");  
    scanf("%d",&a);  
} while ((a <= 0) || (a > 10));
```

Langage C

3) Boucle **for**

La syntaxe de for est :

```
for (expr 1 ;expr 2 ;expr 3)  
instruction
```

Une version équivalente plus intuitive est :

```
expr 1;  
while (expr 2 )  
{  
instruction expr 3;  
}
```

Par exemple, pour imprimer tous les entiers de 0 à 9, on écrit :

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

A la fin de cette boucle, i vaudra 10. Les trois expressions utilisées dans une boucle for peuvent être constituées de plusieurs expressions séparées par des virgules. Cela permet par exemple de faire plusieurs initialisations à la fois.

Langage C

Par exemple, pour calculer la factorielle d'un entier, on peut écrire :

```
int n, i, fact;  
for (i = 1, fact = 1; i <= n; i++)  
    fact *= i;  
    printf("%d != %d |n",n,fact);
```

On peut également insérer l'instruction `fact *= i;` dans la boucle `for` ce qui donne :

```
int n, i, fact;  
for (i = 1, fact = 1; i <= n; fact *= i, i++);  
    printf("%d != %d |n",n,fact);
```

On évitera toutefois ce type d'acrobaties qui n'apportent rien et rendent le programme difficilement lisible.

Langage C

V.3. Les branchements non conditionnels

1) Les branchements non conditionnels **break**

On a vu le rôle de l'instruction `break`; au sein d'une instruction de branchement multiple *switch*. L'instruction *break* peut, plus généralement, être employée à l'intérieur de n'importe quelle boucle. Elle permet d'interrompre le déroulement de la boucle, et passe à la première instruction qui suit la boucle. En cas de boucles imbriquées, `break` fait sortir de la boucle la plus interne.

Par exemple, le programme suivant :

```
main()
{
    int i;
    for (i = 0; i < 5; i++)
    {
        printf("i = %d\n",i);
        if (i == 3)
            break;
    }
    printf("valeur de i a la sortie de la boucle = %d\n",i);
}
```

```
i = 0
i = 1
i = 2
i = 3
valeur de i a la sortie de la boucle = 3
```

imprime à l'écran

Langage C

2) Branchement non conditionnel **continue**

L'instruction *continue* permet de passer directement au tour de boucle suivant, sans exécuter les autres instructions de la boucle. Ainsi le programme

```
main()
{
    int i;
    for (i = 0; i < 5; i++)
    {
        if (i == 3)
            continue;
        printf("i = %d\n",i);
    }
    printf("valeur de i a la sortie de la boucle = %d\n",i);
}
```

imprime

```
i = 0
i = 1
i = 2
i = 4
valeur de i a la sortie de la boucle = 5
```

Langage C

3) Branchement non conditionnel **goto**

L'instruction *goto* permet d'effectuer un saut jusqu'à l'instruction *etiquette* correspondant. Elle est à proscrire de tout programme C digne de ce nom. Elle n'est jamais recommandée.

4) Branchement non conditionnel **return**

L'instruction *return* provoque l'abandon de la fonction en cours et le retour à la fonction appelante.

La syntaxe de cette instruction est la suivante:

return *<expression>*; /* termine la fonction et retourne <expression> */

ou:

return; /* termine la fonction sans spécifier de valeur de retour */

Langage C

VI. Les fonctions et les procédures

VI.1. Les fonctions

VI.2. Appel d'une fonction

VI.3. La fonction main

VI.4. Les fonctions imbriquées

VI.5. Les fonctions récursives

VI.6. Les procédures

Langage C

VI. Les fonctions et les procédures

VI.1. Les fonctions

Une fonction est un bloc d'instructions auquel on donne un nom pour pouvoir l'utiliser. Eventuellement, ce bloc d'instructions peut avoir besoin de valeurs pour pouvoir travailler sur des données qu'on lui transmet. Dans ce cas, on dit que la fonction a besoin de paramètres.

Comme dans la plupart des langages, on peut en C découper un programme en plusieurs fonctions. Une seule de ces fonctions existe obligatoirement; c'est la fonction principale appelée ***main***. Cette fonction principale peut, éventuellement, appeler une ou plusieurs fonctions secondaires. De même, chaque fonction secondaire peut appeler d'autres fonctions secondaires ou s'appeler elle-même (dans ce dernier cas, on dit que la fonction est réursive).

```
type nom_fonction(type1 arg1,..., typen argn)  
{  
    [déclarations de variables locales]  
    liste d'instructions  
}
```


Langage C

- Une fonction qui ne renvoie pas de valeur est une fonction dont le type est spécifié par le mot clé *void*.
- Les arguments de la fonction sont appelés paramètres formels, par opposition aux paramètres effectifs qui sont les paramètres avec lesquels la fonction est effectivement appelée.
- Si la fonction ne possède pas de paramètres, on remplace la liste de paramètres formels par le mot-clé *void*.
- La valeur de l'expression de

return(expression);

est la valeur que retourne la fonction. Son type doit être le même que celui qui a été spécifié dans l'en-tête de la fonction. Si la fonction ne retourne pas de valeur (fonction de type *void*), sa définition s'achève par

return;

Langage C

Exemple:

```
int produit (int a, int b)
{
    return (a*b);
}

int puissance (int a, int n)
{
    if (n == 0)
        return(1);
    return (a* puissance(a, n-1));
}

void imprime_tab (int *tab, int nb_elements)
{
    int i;
    for(i = 0; i < nb_elements; i++)
        printf("%d\t", tab[i]);
    printf("\n");
    return;
}
```

Langage C

VI.2. Appel d'une fonction

- L'appel d'une fonction se fait en écrivant son nom, suivi d'une paire de parenthèses contenant éventuellement une liste d'arguments effectifs.

-Le passage des arguments se fait par valeur. Cela signifie que les arguments formels de la fonction représentent d'authentiques variables locales initialisées, lors de l'appel de la fonction, par les valeurs des arguments effectifs correspondants.

Exemple:

```
void main()  
{  
    prod = produit(4,5);  
    pui = puissance(2,3);  
    imprime_tab(int *tab, 10);  
}
```

Langage C

VI.3. La fonction *main*

La fonction principale *main* est une fonction comme les autres. Nous avons jusqu'à présent considéré qu'elle était de type *void*, ce qui est toléré par le compilateur. Toutefois l'écriture:

main()

provoque un message d'avertissement lorsqu'on utilise l'option `-Wall` de gcc:

```
%gcc -Wall prog.c
```

```
prog.c:5:warning: return-type defaults to 'int'
```

```
prog.c:In function 'main':
```

```
prog.c:11:warning: control reaches end of non-void function
```

En fait, la fonction *main* est de type *int*. Elle doit retourner un entier dont la valeur est transmise à l'environnement d'exécution. Cet entier indique si le programme s'est ou non déroulé sans erreur. La valeur de retour 0 correspond à une terminaison correcte, toute valeur de retour non nulle correspond à une terminaison sur une erreur.

Langage C

La fonction *main* peut également posséder des paramètres formels. En effet, un programme C peut recevoir une liste d'arguments au lancement de son exécution. La ligne de commande qui sert à lancer le programme est, dans ce cas, composée du nom du fichier exécutable suivi par des paramètres. La fonction *main* reçoit tous ces éléments de la part de l'interpréteur de commandes. En fait, la fonction *main* possède deux paramètres formels, appelés par convention *argc* (argument count) et *argv* (argument vector).

argc est une variable de type ***int*** dont la valeur est égale au nombre de mots composant la ligne de commande (y compris le nom de l'exécutable). Elle est donc égale au nombre de paramètres effectifs de la fonction + 1.

argv est un tableau de chaînes de caractères correspondant chacune à un mot de la ligne de commande. Le premier élément *argv[0]* contient donc le nom de la commande (du fichier exécutable), le second *argv[1]* contient le premier paramètre....

Langage C

Ainsi si on a le programme: **MyProg.c**

```
#include <stdio.h>  
int main(int argc, char *argv[])  
{  
    int i;  
    for (i = 0; i < argc; i++)  
    {  
        printf("parameter %d value is %s\n", i, argv[i]);  
    }  
    return 0;  
}
```

Si j'exécute MyProg de la façon suivante:

MyProg param1 param2 param3

J'aurai comme résultat

parameter 0 value is MyProg
parameter 1 value is param1
parameter 2 value is param2
parameter 3 value is param3

- ➔ nom de la commande argv[0]
- ➔ premier paramètre argv[1]
- ➔ second paramètre argv[2]
- ➔ troisième paramètre argv[3]

Langage C

VI.4. Les fonctions imbriquées

À l'inverse de certains langages, les fonctions imbriquées n'existent pas dans le langage C. Il n'est donc pas possible qu'une fonction ne soit connue qu'à l'intérieur d'une autre fonction.

<div>function main -> F1</div>		main() { ... F1(...); ... }
<div>function F1 -> F2</div>		... F1(...) { ... F2(...); ... }
<div>function F2</div>		... F2(...) { ... }

en langage C

<div>program P <div>function F1 <div>function F2</div></div></div>		program P(...) function F1(...):... function F2(...):... begin {F2} ... end {F2}; begin {F1} ... end {F1}; begin {P} ... end {P}.
--	--	--

en langage pascal

Langage C

VI.5. Les fonctions récursives

Une fonction récursive est une fonction qui s'appelle elle-même. La récursivité permet de résoudre certains problèmes d'une manière très rapide, alors que si on devait les résoudre de manière itérative, il nous faudrait beaucoup plus de temps et de structures de données intermédiaires.

En résumé, une fonction récursive comporte un appel à elle-même, alors qu'une fonction non récursive ne comporte que des appels à d'autres fonctions.

Exemple de fonction factorielle :

```
int factorielle(int n)  
{  
    if (n == 1)  
        return(1);  
    else  
        return(n * factorielle(n-1));  
}
```


Langage C

VI.6. Les procédures

Le langage C ne comporte pas à strictement parler le concept de procédure. Cependant, les fonctions pouvant réaliser sans aucune restriction tout effet de bord qu'elles désirent, le programmeur peut réaliser une procédure à l'aide d'une fonction qui ne rendra aucune valeur. Pour exprimer l'idée de « aucune valeur », on utilise le mot-clé ***void***. Une procédure sera donc implémentée sous la forme d'une fonction retournant *void* et dont la partie *liste-d'instructions* ne comportera pas d'instruction *return*.

Lors de l'appel de la procédure, il faudra ignorer la valeur rendue c'est à dire ne pas l'englober dans une expression.

Problème de vocabulaire

Nous utiliserons le terme de *fonction* pour désigner indifféremment une *procédure* ou une *fonction*, chaque fois qu'il ne sera pas nécessaire de faire la distinction entre les deux.

Langage C

Exemple :

```
void print_add(int i,int j) /* la procédure et ses paramètres formels */  
{  
    int r; /* une variable locale à print_add */  
    r = i + j;  
    ... /* instruction pour imprimer la valeur de r */  
}
```

```
void prints(void) /* une procédure sans paramètres */  
{  
    int a,b; /* variables locales à prints */  
    a = 12;  
    b = 45;  
    print_add(a,b); /* appel de print_add */  
    print_add(13,67); /* un autre appel à print_add */  
}
```

Langage C

VII. Les objets structurés

VII.1. Les tableaux

VII.2. Les structures

VII.3. Les champs de bit

VII.4. Les unions

VII.5. Les énumérations

VII.6. La directive *typedef*

Langage C

VII. Les objets structurés

VII.1. Les tableaux

Les tableaux permettent de gérer un ensemble de variables de même type.

Un tableau est une zone mémoire constituée de cases contiguës où sont rangées des données de même type. Les cases ont donc une taille identique. Un tableau possède un nombre fixé de cases qui doit être connu quand il est créé. La zone mémoire possède donc un début et une fin. Pour accéder à une case, nous utilisons un indice qui repère le numéro de la case à laquelle on fait référence. L'indice d'un tableau est un index, il y a ainsi un lien étroit entre tableau et pointeur

Comme toute variable, une case d'un tableau doit être initialisée avant d'être utilisée.

- La déclaration d'un tableau à une dimension:

type nom-du-tableau[nombre_éléments];

où *nombre_éléments* est une expression constante entière positive.

Langage C

Par exemple, la déclaration

```
int tab[10];
```

indique que `tab` est un tableau de 10 éléments de type *int*. Cette déclaration alloue donc en mémoire pour l'objet `tab` un espace de 10×4 octets consécutifs.

Pour plus de clarté, il est recommandé de donner un nom à la constante `nombre_éléments` par une directive au préprocesseur, par exemple:

```
#define nombre_éléments 10
```

On accède à un élément du tableau en lui appliquant l'opérateur `[]`. Les éléments d'un tableau sont toujours numérotés de 0 à *nombre_éléments-1*. Le programme suivant imprime les éléments du tableau `tab`:

```
#define N 10
```

```
main()
```

```
{
```

```
    int tab[N];
```

```
    int i;
```

```
    ...
```

```
    for (i = 0; i < N; i++)
```

```
        printf("tab[%d] = %d\n", i, tab[i]);
```

```
}
```

Langage C

- De manière similaire, on peut déclarer un tableau à plusieurs dimensions. Par exemple, pour un tableau à deux dimensions :

type nom_du_tableau[nombre_lignes][nombre_colonnes]

En fait, un tableau à deux dimensions est un tableau unidimensionnel dont chaque élément est lui-même un tableau. On accède à un élément du tableau par l'expression "tableau[i][j]".

Pour initialiser un tableau à plusieurs dimensions à la compilation, on utilise une liste dont chaque élément est une liste de constantes :

#define M 2

#define N 3

int tab[M][N] = {{1, 2, 3}, {4, 5, 6}};

main()

{

int i, j;

for (i = 0 ; i < M; i++)

{

for (j = 0; j < N; j++)

printf("tab[%d][%d]=%d\n",i,j,tab[i][j]);

}

}

Langage C

- En langage C, les chaînes de caractères sont représentées comme des tableaux de caractères. Un caractère nul ‘\0’ suit le dernier caractère utile de la chaîne et en indique la fin. Cette convention est suivie par le compilateur (qui range les chaînes constantes en leur ajoutant un caractère nul), ainsi que par les fonctions de la librairie standard qui construisent des chaînes.

Par exemple, si vous avez besoin de manipuler des chaînes de 7 caractères, il faudra déclarer des tableaux de caractères de taille 8.

```
#include <string.h>  
#define taille 8  
char tab[taille];  
strcpy(tab, "bonjour");
```

où la fonction *strcpy* copie le mot « bonjour » dans la chaîne de caractères *tab*.

Il est préférable d'utiliser les fonctions prédéfinies dans la bibliothèque *<string.h>*.

Langage C

Remarques:

1) Le tableau est probablement le concept le plus difficile à définir correctement en C. Il y a en effet beaucoup de confusion sur les termes : tableau, adresse, pointeur, indices...

- Un tableau est une séquence d'éléments de types identiques.
- Le nom du tableau est invariant. Il a la valeur et le type de l'adresse du premier élément du tableau. C'est donc une adresse typée (encore appelée pointeur constant). Etant de la même nature qu'un pointeur, les mêmes règles d'adressage s'appliquent, à savoir que le premier élément est en `tab`, soit `tab + 0` et que son contenu est donc `*(tab + 0)`. De même le contenu du deuxième élément est `*(tab + 1)` etc.
- Sémantique du nom d'un tableau:

En général lorsque le nom d'un tableau apparaît dans une expression il y joue le même rôle qu'une constante de type adresse ayant pour valeur l'adresse du premier élément du tableau. Autrement dit, si `t` est de type tableau, les deux expressions

`t`

`&t[0]`

sont équivalentes.

Langage C

2) Le langage C n'autorise pas le passage d'un tableau en paramètre à une fonction. La raison est probablement une recherche d'efficacité, afin d'éviter des copies inutiles.

Le but de 'passer un tableau' à une fonction est en fait de permettre à celle-ci d'accéder aux éléments du tableau en lecture ou en écriture. Pour se faire, l'adresse du début du tableau et le type des éléments suffisent à mettre en oeuvre l'arithmétique des pointeurs. Un paramètre 'pointeur' est donc exactement ce qu'il faut.

3) En C une fonction ne sait pas 'retourner un tableau'.

Ce qu'elle sait faire, c'est retourner une valeur. La pratique courante est de retourner l'adresse du premier élément du tableau. Pour cela, on définit le type retourné comme un pointeur sur le type d'un élément du tableau.

4) Un tableau ne peut pas figurer à gauche d'un opérateur d'affectation.

Par exemple, on ne peut pas écrire

tab1 = tab2;

Il faut effectuer l'affectation pour chacun des éléments du tableau.

Langage C

5) Débordement par excès et par défaut

- Le débordement par excès survient quand on cherche à accéder à une case en dehors du tableau en utilisant un indice plus grand que l'indice maximal possible.

Exemple:

```
#include <stdio.h>  
#define MAX 10  
int main()  
{  
    int tab[MAX];  
    tab[15] = 1;  
}
```

La case d'indice 15 n'existe pas puisque l'indice maximal pour le tableau qui a été déclaré est 9.

- Le débordement par défaut survient quand on cherche à accéder à une case en dehors du tableau en utilisant un indice plus petit que l'indice minimal possible.

Exemple:

```
tab[-1] = 35;
```

La case d'indice -1 n'existe pas puisque l'indice minimal est 0.

Langage C

Cela peut sembler très surprenant que le compilateur accepte ces instructions, mais c'est le cas. Plus surprenant encore, le programme compilé peut aussi bien fonctionner que ne pas fonctionner.

Les débordements constituent évidemment des erreurs graves de programmation.

Car il est très dangereux d'accéder ainsi à une donnée qui n'existe pas et de la modifier.

En effet, on accède ainsi à des octets dans la mémoire et on les modifie. Cela peut faire "planter" le code, comme cela peut ne pas le faire "planter".

Et puisque le C ne vérifie pas ce genre de chose, ce sont des erreurs très difficiles à détecter quand on les commet.

Langage C

VII.2. Les structures

Une structure est une variable qui regroupe une ou plusieurs variables, non nécessairement de même type (contrairement aux tableaux), appelées champs.

La déclaration d'une structure se fait par le mot réservé ***struct*** suivi d'une liste de déclaration de variables entre accolades.

La syntaxe est :

```
struct modele  
{  
    type champ1;  
    type champ2;  
    ...  
    type champn;  
};
```

Contrairement aux tableaux, les différents éléments d'une structure n'occupent pas nécessairement des zones contiguës en mémoire.

Pour déclarer un objet de type structure correspondant au modèle précédent, on utilise la syntaxe:

```
struct modele objet;
```

Langage C

ou bien, si le modèle n'a pas été déclaré au préalable:

```
struct modele  
{  
    type champ1;  
    type champ2;  
    ...  
    type champn;  
  
    } objet;
```

Par exemple:

```
struct fiche  
{  
    int numero;  
    char nom[32], prenom[32];  
    }a, b, c;
```

Cette déclaration introduit trois variables a, b et c, chacune constituée des trois champs numero, nom et prenom; en même temps elle donne à cette structure le nom fiche.

Langage C

VII.3. Les champs de bit

Il est possible en C de spécifier la longueur des champs d'une structure au bit près si ce champ est de type entier (int ou unsigned int). Cela se fait en précisant le nombre de bits du champ avant le ; qui suit sa déclaration. Par exemple, la structure suivante:

```
struct adresse_virt
{
    unsigned depl           : 9;
    unsigned numpagv       : 16;
                           : 5;
    unsigned tp            : 2;
};
```

découpe un mot de 32 bits en quatre champs,

Langage C

Attention:

Dans tous les cas, les champs de bits sont très dépendants de l'implantation. Par exemple, certaines machines rangent les champs de gauche à droite, d'autres de droite à gauche. Ainsi, la portabilité des structures contenant des champs de bits n'est pas assurée; cela limite leur usage à une certaine catégorie de programmes, comme ceux qui communiquent avec leur machine-hôte à un niveau très bas (noyau d'un système d'exploitation, gestion des périphériques, etc.). Ces programmes sont rarement destinés à être portés d'un système à un autre.

Langage C

VII.4. Les unions

Une union désigne un ensemble de variables de types différents susceptibles d'occuper alternativement une même zone mémoire. Une union permet donc de définir un objet comme pouvant être d'un type au choix parmi un ensemble fini de types. Si les membres d'une union sont de longueurs différentes, la place réservée en mémoire pour la représenter correspond à la taille du membre le plus grand.

Les déclarations et les opérations sur les objets de type union sont les mêmes que celles sur les objets de type struct.

Dans l'exemple suivant, la variable *hier* de type *union jour* peut être soit un entier, soit un caractère.

```
union jour
{
    char lettre;
    int numero;
};
```


Langage C

```
main()
{
    union jour hier, demain;
    hier.lettre= 'J';
    printf("hier= %c\n", hier.lettre);
    hier.numero= 4;
    demain.numero= (hier.numero + 2) % 7;
    printf("demain= %d\n", demain.numero);
}
```

Les unions peuvent être utiles lorsqu'on a besoin de voir un objet sous des types différents (mais en général de même taille). Par exemple, le programme suivant permet de manipuler en même temps les deux champs de type *unsigned int* d'une structure en les identifiant à un objet de type *unsigned long* (en supposant que la taille d'un entier long est deux fois celle d'un int).

Langage C

```
struct coordonnees
{
    unsigned int x;
    unsigned int y;
};
union point
{
    struct coordonnees coord;
    unsigned long mot;
};
main()
{
    union point p1, p2, p3;
    p1.coord.x= 0xf;
    p1.coord.y= 0x1;
    p2.coord.x= 0x8;
    p2.coord.y= 0x8;
    p3.mot= p1.mot ^ p2.mot;
    printf("p3.coord.x= %x |t p3.coord.y = %x\n", p3.coord.x, p3.coord.y);
}
```

Langage C

VII.5. Les énumérations

Une énumération permet de définir des constantes pour une liste de valeurs. Le compilateur assigne une valeur par défaut à chaque élément de la liste en commençant par 0 et en incrémentant à chaque fois.

Un objet de type énumération est défini par le mot-clé *enum* et un identificateur de modèle, suivis de la liste des valeurs que peut prendre cet objet:

enum modele {constante1, constante2, ..., constanten};

Par exemple:

enum Couleur { rouge, vert, bleu, jaune };

Il est aussi possible d'assigner explicitement des valeurs :

enum Couleur { rouge = 2, vert = 3, bleu = 1, jaune = 0 };

La variable s'utilise comme suit :

enum Couleur couleur_pixel;
couleur_pixel = rouge;

Langage C

En réalité, les objets de type *enum* sont représentés comme des *int*. Les valeurs possibles constante1, constante2, ..., constanten sont codées par des entiers de 0 à n-1.

Par exemple, le type *enum booleen* défini dans le programme suivant associe l'entier 0 à la valeur faux et l'entier 1 à la valeur vrai.

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

Langage C

VII.6. La directive *typedef*

Cette directive permet d'assigner un nouveau nom à un type de données. On l'utilise par commodité.

Dans l'exemple suivant, on peut utiliser *Navire* à la place de *struct Bateau* :

```
typedef struct Bateau Navire;
```

De même pour COULEUR à la place de enum couleur :

```
typedef enum Couleur  
{  
    rouge = 2,  
    vert = 3,  
    bleu = 1,  
    jaune = 0  
} COULEUR;
```

```
COULEUR couleur_pixel = rouge;
```

Langage C

VIII. Les entrées/sorties de bas niveau

VIII.1. Notion de descripteur de fichier

VIII.2. Fonctions d'ouverture et de fermeture de fichier

VIII.3. Fonctions de lecture et d'écriture

VIII.4. Accès direct

VIII.5. Relation entre flot et descripteur de fichier

Langage C

VIII. Les entrées/sorties de bas niveau

Les entrées/sorties (E/S) ne font pas vraiment partie du langage C car ces opérations sont dépendantes du système. Cela signifie que pour réaliser des opérations d'entrée/sortie en C, il faut en principe passer par les fonctionnalités offertes par le système. Néanmoins sa bibliothèque standard est fournie avec des fonctions permettant d'effectuer de telles opérations afin de faciliter l'écriture de code portable. Les fonctions et types de données liées aux entrées/sorties sont principalement déclarés dans le fichier ***stdio.h*** (standard input/output).

VIII.1. Notion de descripteur de fichier

Une entrée-sortie de bas niveau est identifiée par un **descripteur de fichier** («file descriptor»). C'est un entier positif ou nul. Les flots stdin, stdout et stderr ont comme descripteur de fichier respectif **0**, **1** et **2**. Il existe une table des descripteurs de fichiers rattachée à chaque processus. La première entrée libre dans cette table est affectée lors de la création d'un descripteur de fichier.

Le nombre d'entrées de cette table, correspondant au nombre maximum de fichiers que peut ouvrir simultanément un processus, est donné par la pseudo-constante **NOFILE** définie dans le fichier en-tête «*sys/param.h*».

Langage C

Lorsque le système exécute un programme, il commence avec trois flots de texte automatiquement ouverts par le système. Ils sont connectés (on dit affectés) aux organes d'entrée-sortie les plus «naturels» par rapport à la manière dont on utilise l'ordinateur. Ces fichiers sont déclarés dans `<stdio.h>`, de la manière suivante :

FILE *stdin, *stdout, *stderr;

stdin ➔ est l'unité standard d'entrée. Elle est habituellement affectée au clavier du poste de travail.

stdout ➔ est l'unité standard de sortie. Elle est habituellement affectée à l'écran du poste de travail.

stderr ➔ est l'unité standard d'affichage des erreurs. Elle est aussi affectée à l'écran du poste de travail.

Langage C

VIII.2. Fonctions d'ouverture et de fermeture de fichier

L'affectation d'un descripteur de fichier, c'est-à-dire l'initialisation d'une entrée-sortie, s'effectue par l'appel aux fonctions **open** et **creat** qui sont déclarées dans le fichier en-tête **fcntl.h**. La version ANSI de l'ordre **open** contient dorénavant l'appel **creat**. Sa syntaxe est:

int open(const char *fichier, int mode, mode_t acces)

int creat(const char *fichier, mode_t acces)

Le type *mode_t* est un alias du type *unsigned long*, défini dans le fichier en-tête *sys/types.h*. L'argument *fichier* indique le fichier à ouvrir.

L'argument *mode* indique le mode d'ouverture du fichier que l'on spécifie à l'aide de pseudo-constantes définies dans le fichier en-tête *fcntl.h*:

Langage C

L'argument *mode* indique le mode d'ouverture du fichier que l'on spécifie à l'aide de pseudo-constantes définies dans le fichier en-tête *fcntl.h*:

O_RDONLY	: lecture seulement,
O_WRONLY	: écriture seulement,
O_RDWR	: lecture et écriture,
O_APPEND	: écriture en fin de fichier,
O_CREAT	: si le fichier n'existe pas il sera créé,
O_TRUNC	: si le fichier existe déjà il est ramené à une taille nulle,
O_EXCL	: provoque une erreur si l'option de création a été indiquée et si le fichier existe déjà.

Ces pseudo-constantes peuvent être combinées à l'aide de l'opérateur booléen `|`.

L'appel `creat` est équivalent à `open` avec `O_WRONLY | O_CREAT | O_TRUNC` comme mode d'accès.

L'appel à `creat`, ainsi qu'à `open` dans le cas où le mode `O_CREAT` a été indiqué, s'effectue avec un autre argument décrivant les accès UNIX du fichier à créer qui se combinent avec ceux définis au niveau de la commande *umask* du SHELL.

Ces fonctions retournent le descripteur de fichier ou `-1` en cas d'erreur.

La fonction `close` permet de libérer le descripteur de fichier passé en argument.

Langage C

Exemple : *#include <stdio.h>*

#include <fcntl.h>

main()

{

int fd1, fd2;

if ((fd1 = open("Fichier1", O_WRONLY/O_CREAT/O_TRUNC,
0644)) == -1)

{

perror("open");

exit(1);

}

if ((fd2 = creat("Fichier2", 0644)) == -1)

{

perror("creat");

exit(2);

}

...

close(fd1);

close(fd2);

return 0;

}

Langage C

VIII.3. Fonctions de lecture et d'écriture

Les fonctions **read** et **write** permettent de lire et écrire dans des fichiers par l'intermédiaire de leur descripteur.

Sa syntaxe :

int read(int fd, char *buffer, int NbOctets)

int write(int fd, char *buffer, int NbOctets)

Ces fonctions lisent ou écrivent, sur le descripteur de fichier **fd**, **NbOctets** à partir de l'adresse **buffer**. Elles retournent le nombre d'octets effectivement transmis. Une valeur de retour de **read** égale à **0** signifie **fin de fichier**.

Une valeur de retour égale à **-1** correspond à la détection d'une erreur d'entrée-sortie.

Langage C

VIII.4. Accès direct

Les fonctions **tell** et **lseek** permettent de récupérer et de positionner le pointeur de position courante. Elles sont déclarées dans le fichier en-tête **unistd.h**. Sa syntaxe:

off_t lseek(int fd, off_t decalage, int origine)

int tell(int fd)

Le type **off_t** est un alias du type long défini dans le fichier en-tête «*sys/types.h*».

Langage C

VIII.5. Relation entre flot et descripteur de fichier

Il est possible de changer de mode de traitement d'un fichier c'est-à-dire passer du mode flot («stream») au mode descripteur («bas niveau») ou vice-versa.

Pour ce faire, il existe :

la pseudo-fonction **fileno** qui permet de récupérer le descripteur à partir d'une structure de type FILE,

la fonction **fdopen** qui permet de créer une structure de type FILE et de l'associer à un descripteur.

Sa syntaxe:

int fileno(FILE *f)

FILE *fdopen(const int fd, const char *type)

Le mode d'ouverture indiqué au niveau de l'argument type de la fonction **fdopen**, doit être compatible avec celui spécifié à l'appel de la fonction **open** qui a servi à créer le descripteur. Ces deux fonctions sont déclarées dans le fichier en-tête **stdio.h**.

Langage C

IX. Les bibliothèques

IX.1. Définition

IX.2. La librairie standard

Langage C

IX. Les bibliothèques

IX.1. Définition

Une bibliothèque (library) est une collection de fonctions mise à la disposition des programmeurs. Elle se compose d'une interface matérialisée par un ou plusieurs fichiers d'entêtes (.h), et d'un fichier d'implémentation qui contient le corps des fonctions (.lib, .a, .dll, .so etc.) sous forme exécutable.

Il est aussi possible à un programmeur de 'capitaliser' son travail en réalisant des fonctions réutilisables qu'il peut ensuite organiser en bibliothèques. Cette pratique est courante et encouragée.

La création physique d'une bibliothèque est assez simple si on respecte quelques règles de conception, comme l'absence de globales, la souplesse et l'autonomie.

Une bibliothèque peut être statique ou dynamique (partagée).

Langage C

Bibliothèque statique :

Une bibliothèque à édition de lien statique (.lib, .a etc.) est liée à l'application pour ne former qu'un seul exécutable. La taille de celui-ci peut être importante, mais il a l'avantage d'être autonome. Cette pratique a l'avantage de la simplicité, et elle ne requiert aucune action particulière de la part d'un éventuel système lors de l'exécution du programme. Elle est très utilisée en programmation embarquée (*embedded*).

Bibliothèque dynamique :

Une bibliothèque à édition de lien dynamique, est un fichier séparé (.dll, .so, ...) qui doit être livré avec l'exécutable. L'intérêt est que plusieurs applications peuvent se partager la même bibliothèque, ce qui est intéressant, surtout si sa taille est importante. Dans ce cas, les exécutables sont plus petits. Autre avantage, les défauts de la bibliothèque peuvent être corrigés indépendamment des applications. Ensuite, la correction est répercutée immédiatement sur toutes les applications concernées par simple mise à jour de la bibliothèque dynamique.

Une application qui utilise une bibliothèque dynamique doit réaliser le lien avec la bibliothèque à l'exécution. Pour cela, elle utilise des appels à des fonctions système spécifiques dans sa phase d'initialisation. Les détails dépendent de la plate-forme.

Langage C

IX.2. La librairie standard

- Entrées-sorties **<stdio.h>**
 - Manipulation de fichiers
 - Entrées et sorties formatées
 - Impression et lecture de caractères
- Manipulation de caractères **<ctype.h>**
- Manipulation de chaînes de caractères **<string.h>**
- Fonctions mathématiques **<math.h>**
- Utilitaires divers **<stdlib.h>**
 - Allocation dynamique
 - Conversion de chaînes de caractères en nombres
 - Génération de nombres pseudo-aléatoires
 - Arithmétique sur les entiers
 - Recherche et tri
 - Communication avec l'environnement
- Date et heure **<time.h>**

Langage C

- Créer des bibliothèques avec Code::Blocks

<http://www.siteduzero.com/tutoriel-3-35449-creer-des-bibliotheques-avec-code-blocks.html>

- Utilisation de makefile

<http://gl.developpez.com/tutoriel/outil/makefile/>

Langage C

X. Les pointeurs

X.1. Définition

X.2. Les pointeurs et les tableaux

X.3. Les pointeurs et les chaînes de caractères

X.4. Les pointeurs et les structures

X.5. Les pointeurs et les fonctions

X.6. Les opérations sur les pointeurs

Langage C

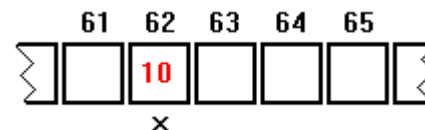
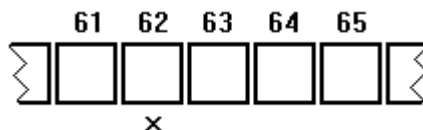
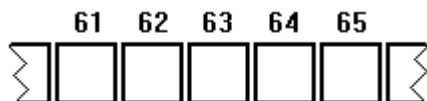
X. Les pointeurs

X.1. Définition

- Une variable est destinée à contenir une valeur du type avec lequel elle est déclarée. Physiquement cette valeur se situe en mémoire. Prenons comme exemple un entier nommé *x* :

int x; // Réserve un emplacement pour un entier en mémoire.

x = 10; // Ecrit la valeur 10 dans l'emplacement réservé.



La valeur de *x* est située physiquement à l'emplacement **&x** (adresse de *x*) dans la mémoire (62 dans le contexte du schéma). Pour obtenir l'adresse d'une variable on fait précéder son nom avec l'opérateur '**&**' (adresse de) :

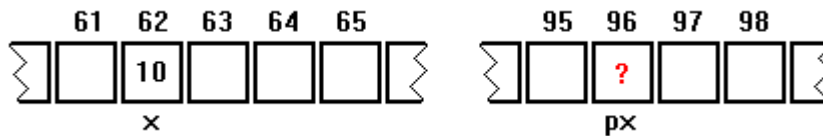
printf("%p",&x);

Ce qui dans le cas du schéma ci-dessus, afficherait 62

Langage C

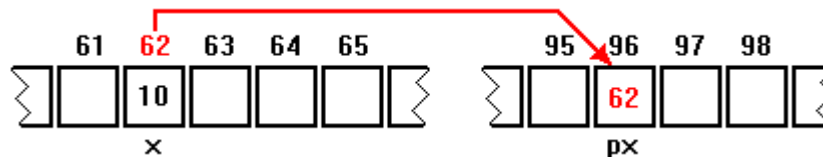
- Un pointeur est aussi une variable, il est destiné à contenir une adresse mémoire, c'est à dire une valeur identifiant un emplacement en mémoire. Pour différencier un pointeur d'une variable ordinaire, on fait précéder son nom du signe '*' lors de sa déclaration. Poursuivons notre exemple :

int *px; *// Réserve un emplacement pour stocker une adresse mémoire.*



Je réserve un emplacement en mémoire pour le pointeur *px* (case numéro 96 dans le cas du schéma). Jusqu'à ici il n'y a donc pas de différence avec une variable ordinaire. Quand j'écris :

px = &x; *// Ecrit l'adresse de x dans cet emplacement.*

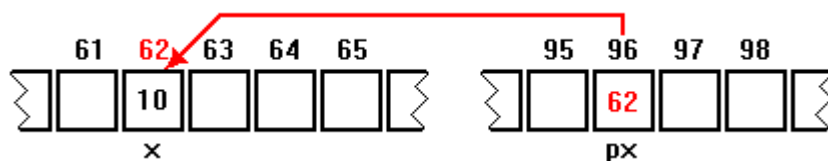


Langage C

Donc l'instruction suivante :

```
printf("%d", *px);
```

Affiche la valeur de x par pointeur déréférencé (10 dans le cas du schéma).



On peut donc de la même façon modifier la valeur de x :

```
*px = 20;           // maintenant x est égal à 20.
```

En résumé, un pointeur est une variable qui doit être définie en précisant le type de variable pointée, de la façon suivante :

```
type *nom_du_pointeur
```

Le type de variable pointée peut être aussi bien un type primaire (tel que *int*, *char*...) qu'un type complexe (tel que *struct*...).

Langage C

Remarques:

- 1) Les pointeurs permettent de manipuler de façon simple des données pouvant être importantes (au lieu de passer à une fonction un élément très grand (en taille) on pourra par exemple lui fournir un pointeur vers cet élément...)
- 2) Les tableaux ne permettent de stocker qu'un nombre fixé d'éléments de même type. En stockant des pointeurs dans les cases d'un tableau, il sera possible de stocker des éléments de taille diverse, et même de rajouter des éléments au tableau en cours d'utilisation (c'est la notion de tableau dynamique qui est très étroitement liée à celle de pointeur)
- 3) Il est possible de créer des structures chaînées, c'est-à-dire comportant des maillons
- 4) un pointeur doit être initialisé avec une adresse valide, c'est à dire qui a été réservée en mémoire (allouée) par le programme pour être utilisé. Autrement dit, si vous n'initialisez pas votre pointeur, celui-ci risque de pointer vers une zone hasardeuse de votre mémoire, ce qui peut être un morceau de votre programme ou... de votre système d'exploitation !

Langage C

5) Un pointeur doit **préférentiellement** être typé !

Il est toutefois possible de définir un pointeur sur *void*, c'est-à-dire sur quelque chose qui n'a pas de type prédéfini (*void * toto*). Ce genre de pointeur sert généralement de pointeur de transition, dans une fonction générique, avant un transtypage permettant d'accéder effectivement aux données pointées.

Langage C

X.2. Les pointeurs et les tableaux

Tout tableau en C est en fait un pointeur constant. Dans la déclaration

int tab[10];

tab est un pointeur constant (non modifiable) dont la valeur est l'adresse du premier élément du tableau. Autrement dit, *tab* a pour valeur *&tab[0]*. On peut donc utiliser un pointeur initialisé à *tab* pour parcourir les éléments du tableau.

#define N 5

int tab[5] = {0, 1, 4, 20, 7};

main()

{

int i;

int *p;

p = tab;

for (i = 0; i < N; i++)

{

printf(" %d |n", *p);

p++;

}

}

Langage C

On accède à l'élément d'indice i du tableau tab grâce à l'opérateur d'indexation $[]$, par l'expression $tab[i]$. Cet opérateur d'indexation peut en fait s'appliquer à tout objet p de type pointeur. Il est lié à l'opérateur d'indirection $*$ par la formule

$$p[i] = *(p + i)$$

Les pointeurs et tableaux se manipulent donc exactement de la même manière. Par exemple, le programme précédent peut aussi s'écrire

```
#define N 5  
int tab[5] = {0, 1, 4, 20, 7};  
main()  
{  
    int i;  
    int *p;  
    p = tab;  
    for (i = 0; i < N; i++)  
        printf(" %d \n", p[i]);  
}
```

Langage C

Cependant, la manipulation de tableaux, et non de pointeurs, possède certains inconvénients dus au fait qu'un tableau est un pointeur constant. Ainsi

- on ne peut pas créer de tableaux dont la taille est une variable du programme,
- on ne peut pas créer de tableaux bidimensionnels dont les lignes n'ont pas toutes le même nombre d'éléments.

Ces opérations deviennent possibles dès que l'on manipule des pointeurs alloués dynamiquement. Ainsi, pour créer un tableau d'entiers à n éléments où n est une variable du programme, on écrit :

```
#include <stdlib.h>  
main()  
{  
    int n;  
    int *tab;  
    ...  
    tab = (int*)malloc(n * sizeof(int));  
    ...  
    free(tab);  
}
```

Langage C

Si on veut en plus que tous les éléments du tableau `tab` soient initialisés à zéro, on remplace l'allocation dynamique avec *malloc* par:

tab = (int*)calloc(n, sizeof(int));

Les éléments de `tab` sont manipulés avec l'opérateur d'indexation `[]`, exactement comme pour les tableaux.

Les deux différences principales entre un tableau et un pointeur sont:

- un pointeur doit toujours être initialisé, soit par une allocation dynamique, soit par affectation d'une expression adresse, par exemple ***p = &i;***;
- un tableau n'est pas une Lvalue ; il ne peut donc pas figurer à gauche d'un opérateur d'affectation. En particulier, un tableau ne supporte pas l'arithmétique (on ne peut pas écrire `tab++;`).

Langage C

X.3. Les pointeurs et les chaînes de caractères

On a vu précédemment qu'une chaîne de caractères était un tableau à une dimension d'objets de type *char*, se terminant par le caractère nul '\0'. On peut donc manipuler toute chaîne de caractères à l'aide d'un pointeur sur un objet de type *char*. On peut faire subir à une chaîne définie par:

char *chaine;

des affectations comme:

chaine = "ceci est une chaine";

et toute opération valide sur les pointeurs, comme l'instruction:

chaine++;

Langage C

Un pointeur sur *char* peut être initialisé lors de la déclaration si on lui affecte l'adresse d'une chaîne de caractères constante:

```
char *B = "Bonjour !";
```

Attention !

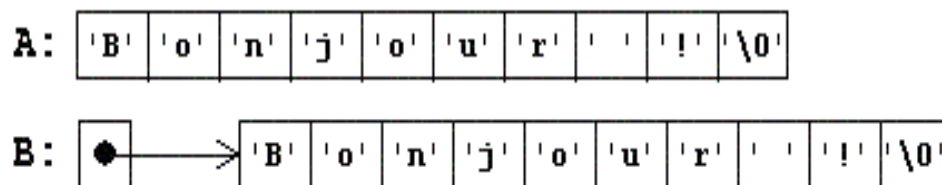
Il existe une différence importante entre les deux déclarations:

```
char A[] = "Bonjour !"; /* un tableau */
```

```
char *B = "Bonjour !"; /* un pointeur */
```

A est un tableau qui a exactement la grandeur pour contenir la chaîne de caractères et la terminaison `\0`. Les caractères de la chaîne peuvent être changés, mais le nom *A* va toujours pointer sur la même adresse en mémoire.

B est un pointeur qui est initialisé de façon à ce qu'il pointe sur une chaîne de caractères constante stockée quelque part en mémoire. Le pointeur peut être modifié et pointer sur autre chose. La chaîne constante peut être lue, copiée ou affichée, mais pas modifiée.



Langage C

Remarques:

1) Si nous affectons une nouvelle valeur à un pointeur sur une chaîne de caractères constante, nous risquons de perdre la chaîne constante. D'autre part, un pointeur sur char a l'avantage de pouvoir pointer sur des chaînes de n'importe quelle longueur:

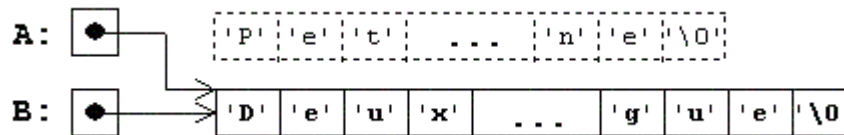
Exemple:

```
char *A = "Petite chaîne";
```

```
char *B = "Deuxième chaîne un peu plus longue";
```

```
A = B;
```

Maintenant *A* et *B* pointent sur la même chaîne; la "Petite chaîne" est perdue:



Langage C

2) Les affectations discutées ci-dessus ne peuvent pas être effectuées avec des tableaux de caractères:

Exemple:

```
char A[45] = "Petite chaîne";
```

```
char B[45] = "Deuxième chaîne un peu plus longue";
```

```
char C[30];
```

```
A = B; /* IMPOSSIBLE -> ERREUR !!! */
```

```
C = "Bonjour !"; /* IMPOSSIBLE -> ERREUR !!! */
```

A:

'P'	'e'	't'	...	'n'	'e'	'\0'
-----	-----	-----	-----	-----	-----	------

B:

'D'	'e'	'u'	'x'	...	'g'	'u'	'e'	'\0'
-----	-----	-----	-----	-----	-----	-----	-----	------

Dans cet exemple, nous essayons de copier l'adresse de *B* dans *A*, respectivement l'adresse de la chaîne constante dans *C*. Ces opérations sont impossibles et illégales parce que *l'adresse représentée par le nom d'un tableau reste toujours constante*.

Pour changer le contenu d'un tableau, nous devons changer les composantes du tableau l'une après l'autre.

Langage C

Ainsi, le programme suivant imprime le nombre de caractères d'une chaîne (sans compter le caractère nul).

```
#include <stdio.h>  
main()  
{  
    int i;  
    char *chaine;  
    chaine = "chaine de caracteres";  
    for (i = 0; *chaine != '\0'; i++)  
        chaine++;  
    printf("nombre de caracteres = %d\n",i);  
}
```

Langage C

La fonction donnant la longueur d'une chaîne de caractères, définie dans la librairie standard *string.h*, procède de manière identique. Il s'agit de la fonction *strlen* dont la syntaxe est *strlen(chaine)*; où *chaine* est un pointeur sur un objet de type *char*.

Cette fonction renvoie un entier dont la valeur est égale à la longueur de la chaîne passée en argument (moins le caractère '\0').

L'utilisation de pointeurs de caractère et non de tableaux permet par exemple de créer une chaîne correspondant à la concaténation de deux chaînes de caractères :

Langage C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main()
{

    int i;
    char *chaine1, *chaine2, *res, *p;
    chaine1 = "chaine ";
    chaine2 = "de caracteres";
    res = (char*)malloc((strlen(chaine1) + strlen(chaine2)) * sizeof(char));
    p = res;
    for (i = 0; i < strlen(chaine1); i++)
        *p++ = chaine1[i];
    for (i = 0; i < strlen(chaine2); i++)
        *p++ = chaine2[i];
    printf("%s\n",res);
}
```

Remarquons l'indispensable utilisation d'un pointeur intermédiaire *p* dès que l'on effectue des opérations de type incrémentation. En effet, si on avait incrémenté directement la valeur de *res*, on aurait évidemment perdu la référence sur le premier caractère de la chaîne.

Langage C

En résumé:

- Utilisons des tableaux de caractères pour déclarer les chaînes de caractères que nous voulons modifier.
- Utilisons des pointeurs sur *char* pour manipuler des chaînes de caractères constantes (dont le contenu ne change pas).
- Utilisons de préférence des pointeurs pour effectuer les manipulations à l'intérieur des tableaux de caractères.

Langage C

X.4. Les pointeurs et les structures

Contrairement aux tableaux, les objets de type structure en C sont des Lvalues. Ils possèdent une adresse, correspondant à l'adresse du premier élément du premier membre de la structure. On peut donc manipuler des pointeurs sur des structures.

Par exemple:

```
struct personne  
{
```

```
    ...
```

```
};
```

```
int main()
```

```
{
```

```
    struct personne pers;
```

```
    /* pers est une variable de type struct personne*/
```

```
    struct personne *p;
```

```
    /* p est un pointeur vers une struct personne */
```

```
    p = &pers;
```

```
}
```

```

#include <stdlib.h>
#include <stdio.h>
struct eleve {
    char nom[20];
    int date;
};
typedef struct eleve *classe;
main()
{
    int n, i;
    classe tab;
    printf("nombre d'eleves de la classe = ");
    scanf("%d",&n);
    tab = (classe) malloc(n * sizeof(struct eleve));
    for (i=0 ; i < n; i++)
    {
        printf("|n saisie de l'eleve numero %d|n",i);
        printf("nom de l'eleve = ");
        scanf("%s",&tab[i].nom);
        printf("|n date de naissance JJMMAA = ");
        scanf("%d",&tab[i].date);
    }
    printf("|n Entrez un numero ");
    scanf("%d",&i);
    printf("|n Eleve numero %d:",i);
    printf("|n nom = %s",tab[i].nom);
    printf("|n date de naissance = %d|n",tab[i].date);
    free(tab);
}

```

Langage C

Si p est un pointeur sur une structure, on peut accéder à un membre de la structure pointé par l'expression $(*p).membre$

L'utilisation de parenthèses est ici indispensable car l'opérateur d'indirection $*$ à une priorité plus élevée que l'opérateur de membre de structure. Cette notation peut être simplifiée grâce à l'opérateur pointeur de membre de structure, noté $->$.

L'expression précédente est strictement équivalente à $p->membre$

Ainsi, dans le programme précédent, on peut remplacer

$tab[i].nom$	et	$tab[i].date$	respectivement par
$(tab + i)>nom$	et	$(tab + i)->date$	

Langage C

X.5. Les pointeurs et les fonctions

- Pointeurs comme paramètres de fonctions

Une autre utilité des pointeurs est de permettre à des fonctions d'accéder aux données elles même et non à des copies.

Prenons pour exemple une fonction qui échange la valeur de deux entiers :

```
void exchange(int *x, int *y)  
{  
    int tmp;  
    tmp = *x;  
    *x = *y;  
    *y = tmp;  
}
```

Pour que la fonction puisse affecter leurs nouvelles valeurs à chaque variable elle doit y avoir accès, le passage des variables par pointeur permet donc à la fonction d'accéder aux variables par pointeur déréférencé.

Langage C

Voyons l'exemple d'utilisation de la fonction :

```
int a;  
int b;  
a = 5;  
b = 10;  
printf("a = %d\n b = %d\n",a,b);  
exchange(&a, &b);  
printf("a = %d\n b = %d\n",a,b);
```

On passe donc l'adresse des variables a et b comme paramètres puisque la fonction attend des pointeurs comme paramètres. Pour le pointeur les paramètres sont en fait passés par valeur, puisque l'adresse de la variable est bien copiée dans le pointeur créé afin d'être utilisé par la fonction

Le passage de paramètres sous forme de pointeur est aussi utilisé pour passer un tableau en tant que paramètre de fonction, c'est d'ailleurs la seule solution possible dans le cas d'un tableau. La fonction reçoit donc un pointeur sur le premier élément du tableau, mais la fonction ne devant pas accéder en dehors des limites du tableau, elle doit pouvoir en contrôler le traitement dans ce cas. Pour une chaîne de caractères, on peut tester la présence du caractère de fin de chaîne '\0', mais dans la majorité des autres cas il faudra fournir à la fonction la taille du tableau.

Langage C

Prenons comme exemple une fonction qui retourne la plus grande valeur d'un tableau d'entier:

```
int max(int *tab, int n)
{
    int x;
    int nmax;
    nmax = 0;
    for (x=0; x<n; x++)
        if (tab[x]>nmax) nmax=tab[x];
    return nmax;
}
```

Nous lui fournissons donc l'adresse du premier élément du tableau comme premier paramètre et la taille du tableau en second paramètre. Dans l'implémentation de la fonction nous utilisons le pointeur comme s'il était un tableau d'entier (utilisation de l'opérateur crochet "[]" pour accéder à ses éléments). Voici un code d'utilisation de la fonction ***max***:

```
int Tab[] = {12,5,16,7,3,11,14,6,11,4};
printf("%d",max(Tab,10));
```

Langage C

On aurait pu écrire l'entête de la fonction comme ceci :

int max(int tab[], int n)

ou

int max(int tab[10], int n)

Ces écritures étant équivalentes à la première, la fonction recevra dans tous les cas une copie du pointeur sur le tableau.

Langage C

- Les pointeurs de fonctions

En langage C, le nom d'une fonction est considéré comme une adresse de manière identique au nom d'un tableau. Le nom d'une fonction correspond à l'adresse de la première instruction à l'intérieur du code exécutable, une fois l'édition de liens réalisée. Dans l'exemple ci-dessous ou nous créons un pointeur de fonction sur la fonction *max*.

```
int (*pmax)(int*, int);  
pmax = max;  
printf("%d",pmax(Tab,10));
```

Le pointeur doit être déclaré avec la signature de la fonction, c'est-à-dire dans le cas de notre exemple le pointeur *pmax* est un pointeur sur des fonctions recevant en paramètre un pointeur sur un entier puis un entier et retournant un entier. Au pointeur ainsi déclaré doit ensuite être affectée l'adresse d'une fonction (*max* dans notre exemple) ayant la même signature. Le pointeur s'utilise alors avec la même syntaxe que la fonction.

Langage C

X.6. Les opérations sur les pointeurs

Comme les pointeurs jouent un rôle si important, le langage C soutient une série d'opérations arithmétiques sur les pointeurs que l'on ne rencontre en général que dans les langages machines. Le confort de ces opérations en C est basé sur le principe suivant:

Toutes les opérations avec les pointeurs tiennent compte automatiquement du type et de la grandeur des objets pointés.

- Affectation par un pointeur sur le même type soient *P1* et *P2* deux pointeurs sur le même type de données, alors l'instruction **P1 = P2;** fait pointer *P1* sur le même objet que *P2*
- Addition et soustraction d'un nombre entier.
- Incrémentation et décrémentation d'un pointeur.

Langage C

Si p pointe sur l'élément $A[i]$ d'un tableau, alors après l'instruction :

$p++ ;$	p pointe sur $A[i+1]$
$p+=n ;$	p pointe sur $A[i+n]$
$p-- ;$	p pointe sur $A[i-1]$
$p-=n ;$	p pointe sur $A[i-n]$
$p+n$	p pointe sur $A[i+n]$
$p-n$	p pointe sur $A[i-n]$

Domaine des opérations :

L'addition, la soustraction, l'incrémentation et la décrémentation sur les pointeurs sont seulement définies à l'intérieur d'un tableau. Si l'adresse formée par le pointeur et l'indice sort du domaine du tableau, alors le résultat n'est pas défini.

Langage C

Seule exception

Il est permis de 'pointer' sur le premier octet derrière un tableau (à condition que cet octet se trouve dans le même segment de mémoire que le tableau). Cette règle, introduite avec le standard ANSI-C, légalise la définition de boucles qui incrémentent le pointeur avant l'évaluation de la condition d'arrêt.

Exemples :

```
int A[10];
```

```
int *P;
```

```
P = A + 9; /* dernier élément → légal */
```

```
P = A + 10; /* dernier élément + 1 → légal */
```

```
P = A + 11; /* dernier élément + 2 → illégal */
```

```
P = A - 1; /* premier élément - 1 → illégal */
```


Langage C

Soustraction de deux pointeurs :

Soient $P1$ et $P2$ deux pointeurs qui pointent dans le même tableau:

$P1 - P2$ fournit le nombre de composantes comprises entre $P1$ et $P2$

Le résultat de la soustraction **$P1 - P2$** est :

négatif	si $P1$ précède $P2$
zéro	si $P1 = P2$
positif	si $P2$ précède $P1$
indéfini	si $P1$ et $P2$ ne pointent pas dans le même tableau

Plus généralement, la soustraction de deux pointeurs qui pointent dans le même tableau est équivalente à la soustraction des indices correspondants.

Comparaison de deux pointeurs :

On peut comparer deux pointeurs par $<$, $>$, $<=$, $>=$, $==$, $!=$.

La comparaison de deux pointeurs qui pointent dans le même tableau est équivalente à la comparaison des indices correspondants. (Si les pointeurs ne pointent pas dans le même tableau, alors le résultat est donné par leurs positions relatives dans la mémoire).

Langage C

Seule exception

Il est permis de 'pointer' sur le premier octet derrière un tableau (à condition que cet octet se trouve dans le même segment de mémoire que le tableau). Cette règle, introduite avec le standard ANSI-C, légalise la définition de boucles qui incrémentent le pointeur avant l'évaluation de la condition d'arrêt.

Exemples :

```
int A[10];
```

```
int *P;
```

```
P = A + 9; /* dernier élément → légal */
```

```
P = A + 10; /* dernier élément + 1 → légal */
```

```
P = A + 11; /* dernier élément + 2 → illégal */
```

```
P = A - 1; /* premier élément - 1 → illégal */
```