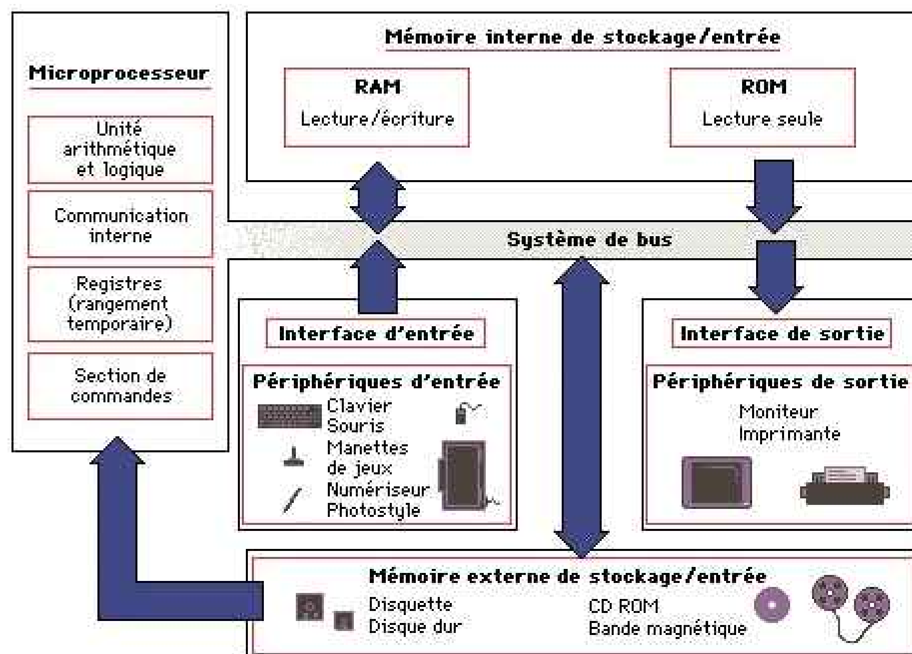
Cours de langage C - Marc Jachym, Lurpa – ENS de Cachan

Créé à l'occasion d'un cours de programmation en :
Licence professionnelle – Métrologie dimensionnelle et qualité
IUT de St Denis, Université Paris 13

1/ Introduction et généralité sur la programmation

1.1/ Structure d'un ordinateur



(origine de l'image : wsinfomatique.xooit.fr)

1.2/ Le programme

Un programme dans un ordinateur est une suite d'instructions, destinée à obtenir un certain résultat, et qui s'exécute grâce au microprocesseur .

Le programme est placé en mémoire vive. Le microprocesseur parcourt les instructions du programme en lisant séquentiellement la mémoire et exécute ces instructions.

L'exécution d'une instruction correspond souvent à la recherche d'une (ou plusieurs) donnée en mémoire, au stockage de cette donnée, à sa transformation éventuelle et à l'écriture du résultat en mémoire.

1.3/ Le microprocesseur

Il peut être simple ou multicoeur, parfois être spécialisé pour les opérations graphiques ou pour les entrées/ sorties (micro-contrôleurs), travailler seul ou en parallèle avec d'autres microprocesseurs mais à la base, pour une unité d'exécution donnée, un microprocesseur reconnaît et exécute **séquentiellement** des instructions prédéfinies grâce à un code instruction. Juste à coté du code instruction, en mémoire se trouvent les opérandes de cette instruction (il peut y en avoir 0, 1 ou 2). Pour l'unité d'exécution, l'instruction est donc le code instruction plus les opérandes éventuelles. Le processeur exécute l'instruction sur les opérandes et passe à l'instruction suivante.

1.4/ Les langages de programmation

Il existe différents types de langages informatiques et même différentes familles de langages informatiques. Ces langages mettent parfois en œuvre des concepts fort différents.

On peut déjà définir des catégories en fonction de la « distance » du langage considéré au langage machine qui est le langage de base compréhensible directement par un microprocesseur. Un langage proche du langage machine sera dit de « bas niveau », un langage proche du langage humain sera dit de « haut niveau ».

On peut distinguer les trois niveaux suivants :

- le langage machine,
- le langage assembleur
- les langages évolués.

Et il y a aussi plusieurs catégories de langages évolués basées sur des approches conceptuelles différentes.

a/ La programmation impérative correspond aux **programmes traditionnels séquentiels** (ou procéduraux) qui sont des suites d'instructions manipulant des variables.

Cette catégorie comprend notamment le langage FORTRAN, le langage Basic historique et le langage C.

b/ Les **langages orientés objets**

Ces langages sont devenus extrêmement populaires. Ce sont largement les plus utilisés.
Le programme n'est plus vu comme l'exécution purement séquentielle d'une suite d'instruction se traduisant assez rapidement en langage proche de la machine mais comme la manipulation d'**objets évolués.** Ces objets se rapportent à des entités identifiables du domaine considéré et ils associent des données avec les traitements sur ces données sous forme de services. Les objets reçoivent aussi (et répondent à) des événements, notamment dans la mise en œuvre d'interfaces graphiques.

La catégorie, des langages orientés objets comprend les langages suivants :

- C++
- Java
- Python
- Php

Le langage Java a mis en œuvre des mécanismes évolués de gestion de la mémoire destinés à affranchir le programmeur de la manipulation directe de cette mémoire comme c'était le cas avec C et C++.
Ces mécanismes basés sur la notion de « ramasse miette » (garbage collector) ont été repris en Python et Php.

De plus, à la différence des langages C et C++, les langages les plus modernes, comme Java, Python et Php, ne sont plus des langages compilés (du moins au sens strict). Ils sont à mi chemin entre compilation et interprétation. Leur compilation (explicite ou réalisée implicitement lors de la première exécution du programme) donne lieu à un **langage intermédiaire.** Le langage intermédiaire ressemble à du langage de bas niveau comme du code assembleur mais il est indépendant de toute plate-forme matérielle et il est mis en œuvre (interprété) par une machine virtuelle.

c/ les **langages fonctionnels**

Il sont basés sur la manipulation d'expressions mathématiques uniquement. Il sont mis en œuvre par emboîtement d'appels de fonctions.
Les langages fonctionnels sortent du cadre de ce cours

1.5/ Quelques éléments clés de programmation

On s'intéresse ici à la programmation impérative séquentielle.

Un programme utilise des constantes, des mots réservés, des variables

a/ Une constante est une valeur fixe, représentée par un symbole et utilisable partout dans le programme. Il faut utiliser les constantes autant que possible car leur définition peut être centralisée à un même endroit. Si au contraire on utilise des valeurs « en dur » directement dans le programme, on aura du mal à les retrouver ultérieurement s'il faut les modifier et il y a un risque d'erreur important.

b/ Un mot réservé est un élément réservé du langage qui a une signification est une fonction précise et qui ne peut pas être utilisé autrement. Par exemple il ne doit pas servir comme nom de constante ou de variable.

c/ Les variables

Ce sont les éléments essentiels d'un programme. C'est par l'intermédiaire des variables qu'un programme manipule les données, qu'elles soient sous forme numériques ou textuelles.

Une variables est représentée par un symbole (c'est le nom de la variable) et contient une valeur,

simple ou complexe. Une valeur simple peut-être un caractère unique, un entier, un nombre réel ou encore une valeur logique (valeur booléenne qui n'a que deux états possibles, vrai ou faux). Une valeur complexe est composée de plusieurs éléments simples et elle peut-être une chaîne de caractères, un tableau (d'entier, de réels), une structure (c'est un composé de divers éléments simples de type différents) .

En fait, une variable représente un **espace de stockage en mémoire vive de l'ordinateur.** Les variables permettent donc d'accéder à la mémoire de l'ordinateur pour y lire, stocker et manipuler des informations. Grâce aux variables, on accède à la mémoire par l'intermédiaire de symboles auxquels le programmeur attribue une signification précise, et cela dispense de devoir manipuler directement des adresses mémoire comme en langage assembleur.

2/ Introduction au langage C

2.1/ Contexte historique

Le langage C a été crée au début des années 1970 par Dennis Ritchie (avec Ken Thompson et Brian Kernighan) aux laboratoires Bell (laboratoire des télécoms américains) à l'occasion de la création, à la même époque, du système d'exploitation UNIX.
Le langage C est donc intrinsèquement lié au système UNIX même si des compilateurs existent pour toutes les plates-formes (Mac, PC, etc). C'est un langage qui est utilisé pour développer toute sorte d'applications et, avec ses dérivés, notamment le langage orienté objet C++, il forme la famille de langages les plus utilisés.

2.2/ Les points remarquables du langage C

a/ C est un **langage compilé**

La compilation consiste à traduite le programme source écrit en langage C en un programme objet écrit en langage machine.
La compilation est effectuée par un programme spécial appelé **compilateur** qui dépend de l'architecture matérielle de la machine cible du programme. En effet le code généré (code objet) n'est pas le même sur un processeur de type Intel ou un processeur de type PowerPC ou ARM.

L'opération de compilation est suivie par une édition de liens qui permet à partir des différents fichiers sources du programme et des bibliothèques qu'il utilise de créer un fichier programme **exécutable**.

La compilation d'un programme comporte trois étapes :

deux phases d'analyses du code : l'analyse lexicale et l'analyse syntaxique durant lesquelles le compilateur produit des avertissements (warnings) et indique des erreurs (relatives à des types de variables non compatibles dans les opérations ou à une syntaxe erronée par exemple)

La phase de génération du code à proprement parler. Si le programme est correct du point de vue lexical et syntaxique, alors le compilateur est à même de générer le code objet (instructions en langage machine du programme).

Attention : La réussite de la compilation indique que le programme est cohérent du point de vue du compilateur. Cela n'indique en aucun cas que le programme est dénué d'erreurs de conception.
En particulier le compilateur ne peut pas vérifier la validité des écriture mémoire effectuées en grâces au pointeurs, ni que la mémoire allouée est correctement libérée.
Ainsi les erreurs mémoires et débordements sont fréquents dans les programmes écrits en langage C.

b/ C'est un langage qui est à la fois de « haut niveau » et de « bas niveau »

c/ la gestion de la mémoire en langage C
C'est un point crucial.
En langage C, le programmeur gère lui-même l'occupation mémoire de ses données. Il doit s'occuper des types de ses variables et réserver / libérer lui-même les espaces de mémoire dynamique dans lesquelles il placera les données (lors du chargement d'un fichier ou de la réception de flux à travers un lien réseau ou une liaison série par exemple).

Le langage C est différent en cela de langages plus récents comme Java, Python ou Php qui sont basés sur des variables au typage dynamique et sur un mécanisme de « ramasse miette».
Avec ces derniers langages, c'est l'**interpréteur** qui détermine lors des affectations quel est le type des variables et c'est l'interpréteur qui, de manière transparente, alloue, ajuste et libère l'espace mémoire nécessaire au stockage des données lors de leur chargement.

Avantages / inconvénients :
On se doute que les langages récents sont plus rapides à mettre en œuvre puisque le programmeur est débarrassé d'un certain nombre de tâches prises en compte automatiquement par l'interpréteur de ces langages . Ainsi le cycle de développement est nettement plus rapide.

En contrepartie, ces langages sont plus lents à l’exécution qu'un programme compilé à partir du langage C . Cela n'est pas un point crucial dans tous les cas et les processeurs étant de plus en plus rapides, ils pallient en partie à ce problème.

Mais pour certains usages, comme la programmation système de bas niveau (gestion directe du matériel pour écrire un driver par exemple) ou pour avoir un accès fin à certaines particularités d'un système (interface Windows ou X Window sur les systèmes Unix), le langage C reste le meilleur choix.

d/ Les pointeurs en langage C

Les pointeurs représentent à la fois la force et la faiblesse du langage C.
Ils permettent une gestion directe du contenu de la mémoire par le programmeur et cela est très efficace dans de nombreux cas. L'inconvénient étant que les opérations mémoires effectuées ne peuvent pas être contrôlées par le compilateur et, par expérience, cela conduit à des erreurs de programmation comme il a déjà été mentionné précédemment.

2.3/ Mon premier programme en langage C

Voici comment se présente un programme en lanage C

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

#define NBRE 5

main()
{
      /* déclaration des variables */
      char chaine[NBRE+1];
      int i;

      /* le programme */
      chaine[NBRE]=0;
      printf("Hello !\n");
      printf("Saissez %d caractères, puis appuyez sur entree: ",_NBRE);
      for (i=0; i<NBRE; i++) {
            chaine[i] = getchar();
      }
      printf("Vous avez entré: ");
      printf(chaine);
      exit(0);
}
```

Repérage des différents éléments et explications

2.4/ La structure d'un programme

a/ zone de déclaration

b/ zones d'instructions

Les instructions sont terminées par un point-virgule.
Elle peuvent être regroupées dans un bloc délimité par des accolades { }
Dans une boucle comme celle de la structure **for** , on peut mettre une instruction unique

```
for (c=0 ; c<val ; c++)
      printf ("%c", c ) ;
```

ou un bloc instruction terminé par l'accolade fermante.

```
for (i=0 ; i<val ; i++) {
      if (c>='a' && c<='z')
            c = toupper(c) ;
      printf ("%c", c ) ;
}
```

c/ des variables avec des types

d/ des opérateurs
affectation, comparaison

c/ des opérations d'entrée/sortie

d/ des structures de contrôle

Ex. n° 1

Présentation d'un environnement de développement (Mingw sous Windows)
avec un éditeur de texte (NotePad, Wordpad, Notepad++)
Mise en œuvre du programme exemple.

2.5/ Les instructions d'entrée/sortie console

2.5.1/ E/S non formatées

Lecture d'un caractère au clavier (entrée standard) :
car = getchar() ;

Ecriture d'un caractère à l'écran (sortir standard) :
putchar (car) ;

b/ E/S formatées

Lecture d'une chaîne de caractères avec spécifications de format :

scanf("chaine avec formats",   adr1, …, adrn) ;

Les arguments adr1, …, adrn sont les adresses des variables qui recevront les données saisies.

Affichage d'une chaîne avec formats :

printf("chaine avec formats", arg1, …, argn) ;

En fait les « instructions » d'E/S présentées ci-dessus ne sont pas des instructions du langage C à proprement parler, mais ce sont des **fonctions de la bibliothèque standard du langage C.**

L'utilisation de ces fonctions nécessitera donc l'inclusion au début du programme du fichier d'entête de la bibliothèque et ce sera lors de l'édition de liens (c'est l'étape qui vient juste après la compilation du programme) que le **code** des fonctions de la bibliothèque sera placé, au même titre que le code qui utilise ces fonctions, dans le fichier programme **exécutable**.

La bibliothèque standard est découpée en plusieurs parties, correspondant à autant de fichiers d'entêtes différents (voir : http://www.utas.edu.au/infosys/info/documentation/C/CStdLib.html)

L'entête pour les opérations d'Entrées / Sorties étudiées ici est le fichier : stdio.h
Donc notre programme source, au tout début du ficher, doit comporter la directive suivante :

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

Le caractère # indique que la directive doit être traitée par le **pré-compilateur**. La pré-compilation est la première étape de la compilation. Ici le code source des **déclarations** de constantes, variables globales éventuelles et fonctions sera inséré dans le fichier (après son chargement en mémoire vive) avant que celui-ci ne soit compilé.
Attention : dans le vocabulaire ci-dessus, il ne faut pas confondre la déclaration d'une fonction (nom de la fonction, nombres et types des arguments qu'elle reçoit, type de la valeur qu'elle retourne) et sa définition (le code programme à proprement parler de la fonction).

Pour les détails sur les chaînes avec formats des fonctions **scanf** et **printf** voir l'excellent cours de *Programmation en langage C*, d'Anne Canteaut (INRIA), disponible à partir du lien suivant :
https://www.rocq.inria.fr/secret/Anne.Canteaut/COURS_C/

exemple avec printf et scanf :

```
#include <stdio.h>

main()
{
  int i,j;
  double d;
  char tab[81];

  printf("Saisir un entier: ");
  scanf("%d", &i);
  printf("l'entier saisi est %d\n", i);
  printf("Saisir 2 entiers et 1 double: ");
  scanf("%d%d%lf", &i, &j, &d);
  printf("les deux entiers saisis sont : %d et %d, le reel est : %f\n", i,j, d);
  printf("Saisir une chaine de caractères (sans espace): ");
  scanf("%80s", tab);
  /* le caractère '\0' est automatiquement ajouté à la fin de la chaîne tab*/
  printf("voici la chaine saisie : %s", tab);
}
```

scanf permet d'interpréter directement les valeurs saisies au clavier en un certain type de données.
Les mêmes choses peuvent être aussi réalisées avec getchar() mais le programmeur doit alors écrire des instructions pour transformer les caractères saisies en entiers, flottants, etc.

Pour placer dans des variables du programme les valeurs saisies par l'utilisateur au moyen de scanf on utilise un symbole spécial : **&**

En effet, le langage C n'utilise qu'un seul type de passage d'argument lors de l'appel à une fonction (ici appel à scanf) qui est le **passage par valeur**. Cela signifie que le **contenu** des variables est

recopié dans le contexte d’exécution de la fonction appelée. La fonction travaille sur une copie des variables et ne peut donc pas mettre à jour directement une variable du programme appelant.

Nous reverrons cela quand nous étudierons de plus près les sous-programmes en langage C. Nous reverrons également le symbole & lors de l'étude des pointeurs.

Le symbole **&** permet d'indiquer, non pas le contenu d'une variable mais son **adresse.**
Ainsi on passe à la fonction scanf **les adresses des variables** du programme appelant dans lesquelles on veut que les valeurs saisies sur « l'entrée standard » soient placées.

3/ Plus en détails : le langage C

3.1. Les variables et les types

3.1.1/ Les types simples

Les variables de type simple représentent une valeur unique comme un caractère, un nombre entier ou un nombre réel. Lors de la déclaration d'une variable de type simple, le compilateur connaît exactement la taille mémoire occupée par la variable.

Voici un tableau résumant les types simples du langage C

Type	Description	Propriété
char	nombre entier (8 bits) signé, caractère	De -128 à 127
int	nombre entier (32 bits) signé	de -2^{31} à $2^{31}-1$
unsigned int	nombre entier (32 bits) non signé	de 0 à $2^{32}-1$
long	nombre entier (64 bits) signé	De 0 à $2^{64}-1$
unsigned char	nombre entier (8 bits) signé	de 0 à $2^{8}-1$
float	nombre réel (flottant) simple précision	$1,2 \times 10^{-38}$ à $3,4 \times 10^{+38}$
double	nombre réel (flottant) double précision	$1.7*10^{-308}$ à $1.7*10^{308}$

On a donc en C plusieurs types pour représenter des nombres, comme :
Les types int et long pour les entiers
Les types float et double pour les réels

Les constantes de type réel s'écrivent sous l'une des formes suivantes :

3.1415
7.252e+2 (ou avec le E majuscule 7.252E+2, ce qui vaut $7,252*10^2 = 725,2$)
25.0e-2 (ce qui vaut 0,25)
-14.524

Le compilateur effectue un certain nombre de **conversions automatiques** de type quand c'est nécessaire. Par exemple, la valeur renvoyée pour l'expression : 2.0 + 1 sera de type réel.

Les nombres réels sont des nombres à **virgule flottante,** c'est-à-dire des nombres pour lesquels la position de la virgule n'est pas fixe. Le codage en mémoire est réalisé par trois parties distinctes : un bit pour le signe, une partie des bits pour représenter l'exposant, une autre partie pour représenter la mantisse (c'est à dire l'ensemble des chiffres significatifs).

Les nombres de type **float** sont codés sur 32 bits dont :

23 bits pour la mantisse
8 bits pour l'exposant
1 bit pour le signe

Les nombres de type **double** sont codés sur 64 bits dont :

52 bits pour la mantisse
11 bits pour l'exposant
1 bit pour le signe

3.1.2/ L'opérateur sizeof()

Cet opérateur particulier du langage C est très important. sizeof() permet de connaître la taille en octets d'un type ou d'une variable.

Il s'utilise ainsi :

```
taille = sizeof(une_variable) ;
```

ou

```
taille = sizeof(un_type) ;
```

par exemple :

```
int i = 1500 ;
int taille ;
taille = sizeof(i) ;
taille = sizeof(int) ; // cette instruction fournit le même résultat que la précédente
```

L'opérateur sizeof est utile dans plusieurs situations et en particulier il permet d'écrire des programmes portables car on n'est pas obligé par exemple, de présupposer de l'occupation d'une variable de type entier (16, 23, ou 64 bits?).

Nous reverrons ce point plus loin avec le calcul de l'occupation mémoire d'une variable de type tableau.

3.1.3/ La modification dynamique d'un type

Il est possible de modifier dynamiquement le type d'une expression lors d'une opération d'affectation, pour peu que l'opération ait un sens.

On utilise un opérateur appelé « cast » qui consiste à indiquer le nouveau type désiré entre parenthèses :

une_variable = (type) expression

on peut « transtyper » des entiers, de long vers int ou réciproquement, de unsigned int vers int, etc et des pointeurs.

La conversion d'un pointeur affectera l'adresse d'un pointeur à un autre pointeur d'un autre type.

3.1.4/ Les types complexes

Les types complexes représentent des variables qui contiennent **plusieurs éléments** de type simple. L'identifiant de la variable complexe représente l'adresse du premier élément de la « variable composée » et les éléments sont accédés au moyen d'un indice ou à l'aide d'un pointeur (qui représente une adresse mémoire).

a/Les tableaux

Déclaration: **type nom[nb];**

Cette déclaration signifie que le compilateur réserve nb places en mémoire pour ranger
les éléments du tableau.

Exemples:

int compteur[10]; //le compilateur réserve des places en mémoire pour 10 entiers, soit 40 octets.

float nombre[20]; // le compilateur réserve des places en mémoire pour 20 réels, soit 80 octets.

Remarque: le nombre donné comme dimension du tableau est nécessairement une EXPRESSION CONSTANTE (expression qui peut contenir des valeurs ou des variables constantes – c.f. modificateur const). Ce ne peut être en aucun cas une combinaison des variables du programme1.

Utilisation: Un élément du tableau est repéré par son indice. En langage C et C++ les tableaux commencent à l'indice 0. L'indice maximum est donc dim-1.

Exemples:
compteur[2] = 5;
nombre[i] = 6.789;

Il n’est pas nécessaire de définir tous les éléments d'un tableau. Toutefois, les valeurs
non initialisées contiennent alors des valeurs quelconques.

Exercice :
Saisir 10 réels, les ranger dans un tableau. Calculer et afficher leur moyenne et leur écart-type.

L'opérateur **sizeof** et les tableaux :

Comment connaître la taille mémoire d'un tableau, c'est à dire le nombre d'octets pris en mémoire pour un tableau ?
Pour cela nous pouvons utiliser l'opérateur sizeof de différentes manières.

Soit un tableau définit par :

```
int myTab[50] ;
```

on peut obtenir sa taille de plusieurs manières :

```
taille = 50*sizeof(int] ;
```

ou :

```
taille = 50*sizeof(myTab[0]) ;
```

mais aussi :

```
taille = sizeof(myTab) ;
```

La dernière possibilité est évidemment la plus simple mais elle ne sera pas applicable pour les types dynamiques que nous verrons plus loin.

Les tableaux à plusieurs dimensions

Tableaux à deux dimensions:

Déclaration: type nom [dim1] [dim2] ;

Utilisation: Un élément du tableau est repéré par ses indices. En langage C les tableaux
commencent aux indices 0. Les indices maximums sont donc dim1-1, dim2-1.

Exemples:

```
compteur[2][4] = 5;

nombre[i][j] = 6.789;

c = compteur[i][j];
```

Exercice 3.6 :
Saisir une matrice d'entiers 2x2, calculer et afficher son déterminant.

Tableaux à plus de deux dimensions

On procède de la même façon en ajoutant les éléments de dimensionnement ou les indices nécessaires.

b/ Les chaînes de caractères.

Il n'y a pas de type spécial « chaîne de caractères » en langage C.
Une chaîne littérale, ou chaîne de caractère constante est représentée à l'aide de double cotes. Elle correspond à un tableau de caractères.

ex. :

```
char chaine[6] = "paris";
printf(chaine) ;
```

Attention : un caractère nul ('\0') est automatiquement ajouté à la fin d'une constante chaîne. Ainsi, dans l'exemple précédent, la déclaration/initialisation

```
char chaine[6] = "paris";
```
est équivalente à
```
char chaine[6] = {'p', 'a', 'r', 'i', 's', '\0' }
```

C'est pourquoi il faut prévoir un tableau de 6 caractères pour stocker la constante.

Dans la bibliothèque standard, il y a des fonctions spéciales pour traiter les chaînes. Il faut inclure le fichier d'entête <string.h> pour y avoir accès.

ex. :

strcpy(dest, src) ;
/* copie la chaine *src dans dest.* Il faut s'assurer que le tables *dest* contient assez d'espace */

pos = strstr(str1, str2) ;

/* renvoie un pointeur de la première occurrence de str2 dans str1 */
Nous verrons ultérieurement ce que sont les pointeurs.

size = strlen(chaine)
/* renvoie la talle de chaine, terminaison ('\0') non comprise.

c/ Les structures

Les structures permettent de définir des objets composés de plusieurs éléments de types différents que l'on peut appeler champs.

Voici la syntaxe de définition d'une structure :

```
struct identificateur {

	type identificateur ;
	…
	…
	type identificateur ;
} ;
```

Le premier identificateur est celui de la structure, qui représente un nouveau type de données. Il ne s'agit pas d'une variable mais d'un type à partir duquel on pourra créer des variables (des instances de la structure nouvellement définie).

La suite de la définition entre les accolades permet de définir les noms des différents champs de la structure ainsi que leur type.

exemple :

Voici un type de structure pour mémoriser des adresses de personnes .

```
struct adresse {
	int id_personne ;
	char rue[40] ;
	char code[10] ;
	char ville[20] ;
} ;
```

Voici maintenant la déclarations de variables avec ce nouveau type de données :

```
struct adresses adr1, adr2, adr3 ;
```

L'accès à un champ d'un objet structuré est réalisé au moyen d'un opérateur particulier, le point. Ainsi :

adr1.code permet d'accéder au champ code de l'objet adr1.
adr3.ville permet d'accéder au champ ville de l'objet adr3

d/ La définition de nouveaux types

Plutôt que de devoir réutiliser le mot réservé **struct** dans la déclaration de variables de type structure, on peut définir des types nouveaux grâce à la directive **typedef**.

Par exemple :

```
typedef struct {
	float reel ;
	float imag ;
} complexe ;
```

Cela définit « complexe » comme un nouveau type de donnée, qui est composé de deux champs, reel et imag. La déclaration d'une variable pour ce type se fait alors de la manière suivante :

complexe v1 ;

et l'on pourra référencer ses champs par la notation avec point : v1.reel et v1.imag

e/ Les énumérations

Il y a moyen de construire un type par énumération de différentes valeurs. Une variable de ce type pourra alors prendre l'une des valeurs de l'énumération. Ce type est introduit par la directive **enum**.

Exemple :

enum quadrilatere { carre, rectangle, losange, parallelogramme} q1, q2 ;

Cela définit un type énumération quadrilatère et déclare deux variables de ce type q1 et q2 qui pourront prendre l'une des 4 valeurs définies (1 seule à la fois). Par exemple :

q1 = rectangle ;
q2 = losange ;

3.2/ Opérateurs et Expressions

Les opérateurs s'appliquent aux types entiers, flottants, caractères ou pointeurs. On ne peut pas recopier un tableau ou une chaîne de caractère avec une simple opération d'affectation par exemple.

3.2.1/ L'opérateur d'affectation

L'opérateur d'affectation est le signe =. La syntaxe d'affectation est

variable = expression ;

expression peut être une variable simple, une constante, un élément de tableau ou une expression complexe arithmétique, logique ou relative à des pointeurs. L'expression est d'abord évaluée puis l'affection a lieu avec la valeur retournée par l'expression.

3.2.2/ Les opérateurs arithmétiques

Ces opérateurs réalisent des opérations sur des variables de type entier ou flottant. Ce sont :

+ pour l'addition
- pour la soustraction
/ pour la division
% opérateur modulo (reste de la division)

Pour la division, si l'un des opérandes est un flottant, le résultat retourné est un flottant.

Pour réaffecter dans une variable une valeur calculée à partir de cette même variable, on peut utiliser les opérateurs suivants :

+=, -=, *=, /=

par exemple, pour deux entiers a et b,

a = a+b ;

est équivalent à a += b ;

Très utilisés sont également les **opérateurs d'incrémentation** ++ et --
Ceux-ci peuvent être placés devant ou derrière la variable avec un résultat différent dans les deux cas : ++a , a++

La différence est la suivante :
Si une variable est incrémentée (ou décrémentée) à l'intérieur d'une expression composée ou dans une affectation (ou aussi dans une boucle) c'est la **valeur après incrémentation** qui est utilisée dans l'expression (l'instruction, la boucle) dans le cas de **++a** (ou –a) et c'est la **valeur avant incrémentation** qui est utilisée dans l'expression (l'instruction, la boucle) dans le cas de **a++** (ou a--).

par exemple, après :

a = 4 ;
b = --a ;

a et b valent 3 car dans la seconde instruction, a est d'abord décrémentée, puis b est affectée avec la nouvelle valeur de a.

mais après :
a = 4 ;
b = a-- ;

a vaut 3, b vaut 4. En effet, a est d'abord utilisée avec sa valeur (4) pour l'affectation dans b, puis a est décrémentée.

Plus compliqué : nous étudierons les boucles ci-dessous mais d'ores et déjà considérons la boucle while suivante :

```
while (i-- > 0) {
	res = res*i ;
}
```

la valeur de i utilisée dans le test, comme la valeur de i utilisée dans l'expression, est celle **avant incrémentation.**

3.2.3/ Opérateurs de comparaison

Les opérateurs suivants évaluent un résultat logique entre deux opérandes. Le résultat est 1 (vrai) ou 0 (faux).

Remarque : il n'y a pas en C de type booléen explicite. La valeur « faux » correspond à 0 tandis que tout ce qui est différent de 0 est évalué comme « vrai » dans une expression logique.

Égalité : ==

a = 1
b = 2
res = a == b ; /* le résultat de la comparaison est faux, donc res est affecté avec la valeur 0 */

Différence :!=

res = a!=b ; /* res vaudra 1 */

Inférieur, Supérieur : < , >

res = a<b ; /* vrai, donc res vaudra 1 */

Inférieur ou égal, Supérieur ou égal : <= , >=

res = a >= b ; /* res vaudra 0 */

Et logique, Ou logique : && , ||

Opérateurs bits à bits :

Et bits à bits : **&**

Ou bits à bits : **|**

Décallage à droite : >>

Décallage à gauche : <<

3.2.4/ Opérateurs bit à bit

Ces opérateurs considèrent individuellement les bits à l'intérieur des octets et permettent d'effectuer des opérations de décalage, des inversions, des ET et des OU bit à bit.

Ces opérateurs s'appliquent aux types entiers (short, int, long). Ce sont :

\>> décalage à droite

a >> n décalage à droite de n positions des bits à l'intérieur de la variable entière a

par exemple si a vaut 16, l'expression a>>2 retournera la valeur 4, ce qui équivaut à une division par 2^2 , soit 4.

<<   décalage à gauche

~   complément à 1 ou inversion bit à bit

exercice : si une variable a type unsigned short vaut 16, quelle sera la valeur de ~a ?

&   ET bit à bit

| OU bit à bt

exercice : a et b sont de type unsigned int. A vaut 16 et B vaut 33. Quelles seront les valeurs des expressions a & b et a | b ?

Remarque : les opérations bits à bits sont principalement utilisées en programmation système.

3.2.5/ Les règles de priorité dans l'évaluation des expressions

Une expression peut être composées de multiples opérateurs de différentes catégories (addition, multiplication, combinaison logique, comparaison, etc). Il y a donc des **règles précises de priorité** entre les opérateurs qui permettent de répondre à la question suivante : dans quel ordre vont s'effectuer les opérations ?

Dans ce contexte,l'**utilisation de parenthèses** permet d'une part de clarifier les expressions et de rendre leur compréhension plus rapide à l'œil humain et d'autre part elle permet au programmeur de choisir lui-même l'ordre d'évaluation désiré.

<u>Un exemple simple :</u>

soient trois variables a, b et c de type int :
int a=2, b=3, c=4 ;

a/ la multiplication est prioritaire sur l'addition, donc :

l'expression a + b*c renvoie 14.

par contre l'expression (a+b) * 4 renvoie 20

b/ l'affectation est le moins prioritaire des opérateurs tandis qu'un appel de fonction sera effectué en tout premier lieu et que les comparaisons logiques sont de priorité « moyenne ».

Donc si j'écris un test de la manière suivante :

```
while ( c = getchar() != '\n' ) {
        instructions
}
```

je n'obtiendrai pas le résultat voulu ! En effet, j'essaye visiblement de lire des caractères au clavier jusqu'à recevoir un ordre '\n' (fin de ligne).

L'appel getchar() sera effectué en premier mais ensuite la comparaison sera effectuée avant l'affection dans la variable c. Donc la variable c recevra finalement le résultat de la comparaison (la valeur 1 en général et la valeur 0 quand « Entrée »sera tapé au clavier).

La boucle fonctionnera donc dans une certaine mesure sauf que je n'obtiendrai pas la valeur du caractère saisi au clavier dans la variable c pour traitement dans les instructions de la boucle. Or on peut supposer que c'est ce que je voulais obtenir !

Donc il aurait fallu écrire :

```
while ( (c = getchar()) != '\n' ) {
        instructions
}
```

Cette fois-ci, grâce à l'ajout d'un jeu de parenthèses, j'affecte d'abord le caractère dans la variable c et je fais le test ensuite.

Pour voir un tableau des priorités comparées pour l'ensemble des opérateurs, on se reportera au cours de langage C d'Anne Canteaut, INRIA, p.24 :

https://www.rocq.inria.fr/secret/Anne.Canteaut/COURS_C/cours.pdf

3.3/ Les structures de contrôle

On appelle structure de contrôle les commandes du langage qui permettent de modifier le flux séquentiel d'un programme en testant des conditions, en faisant répéter une série d'instructions jusqu'à obtention d'un certain résultat.

3.3.1/ La condition if

C'est la structure de contrôle la plus simple :

```
if (condition)
        instruction-1
else
        instruction-2
```

l'instruction n'est exécutée que si la condition est vraie, sinon l'instruction est sautée et le programme continue sont exécution à l'instruction suivante.

<u>Ex :</u>

```
if ( c >= 'a' && c <= 'z')
        c+= 26 ;
```

autres variantes du test if :

avec un bloc d'instructions :

```
if (condition) {
	instruction ;
	….
	….
}
```

avec une alternative :

```
if (condition) {

	…
	bloc instruction 1
	...
}
else {
	…
	bloc instruction 2
	...
}
```

on peut combiner les instructions if, else :

```
if (condition 1) {

	…
	bloc instruction 1
	...
}
else if (condition 2) {
	…
	bloc instruction 2
	...
}
else if (condition 3) {
	…
	bloc instruction 3
	...
}
else {
	…
	bloc instruction 4
	...
}
```

3.3.2/ Switch

Pour faire un choix entre différentes alternatives , on peut utiliser la structure **switch** :

```
switch (expression)
{
case  valeur1 :
        instructions 1  ;
        break ;

case valeur2 :
        instructions2 ;
        break ;
...
...

default :
        instructions ;
}
```

Sans les instructions break, c'est l'ensemble des blocs instructions qui sont exécutés.

ex. : afficher le nom d’un mois en fonction de son numéro :

```
switch ( numMois )
{
case 1 : printf ( ”janvier” ) ; break ;
case 2 : printf ( ”fevrier” ) ; break ;
case 3 : printf ( ”mars” ) ; break ;
case 4 : printf ( ”avril” ) ; break ;
…
…
default : printf ( ”Numéro invalide” ) ;
}
```

3.3.3/ Les boucles avec while

Il y a deux variantes de boucles « tant que ... »

Avec la boucle **do**.... **while** le bloc instruction est exécuté au moins une fois avant le test de la condition.

```
do {
  ...
  instructions
  ...
} while (condition) ;
```

Avec la boucle **while** le test est effectué avant le bloc instructions

```
while (condition)
{
  ...
  instructions
  ...
}
```

ex. : afficher les 26 lettres de l'alphabet, séparées par des virgules.

```
#define TAILLE_ALPHABET 26
char lettre='a';
int compteur = 0;
while(compteur<26) {
  printf("%c", lettre);
  if (lettre++ == 'z')
          printf("\n");
  else
          printf(" , ");
  compteur ++ ;
}
```

3.3.4/ La boucle for

Cette boucle a une forme condensée qui est surtout adaptée aux boucles avec un compteur de type entier. La forme générale est la suivante :

```
for ( instruction-prealable ; condition ; instruction-finale) {
    ...
    instructions
    ...
}
```

Cela est équivalent à la structure while suivante :

```
instruction-prealable ;
while (condition) {
   …
   instructions ;
   …
   instruction-finale ;
}
```

Par exemple, la boucle while précédente peut s'écrire

```
int compteur;
for (compteur=0 ; compteur < 26 ; compteur++ ) {
  printf("%c", lettre);
  if (lettre++ == 'z')
        printf("\n");
  else
        printf(" , ");
  compteur ++ ;
}
```

3.4/ Les sous-programmes

Traditionnellement on distingue deux sortes de sous-programmes, ceux qui renvoient une valeur ou fonctions et ceux qui n'en renvoient pas ou procédures.

Le langage Pascal distinguait explicitement procédures et fonctions mais en langage C, tout sous programme est une fonction.

3.4.1/ Syntaxe générale de définition d'une fonction

Voici la syntaxe générale de définition d'une fonction :

```
type-renvoi  NomFonction (typeArg1 nomArg1, typeArg2 nomArg2, …)
{
  zone de déclaration des variables locales

  zone des instructions de la fonctions
}
```

Les variables définies entre parenthèses avec indication de leur type et de leur nom sont appelées **arguments**. Les arguments sont les valeurs passées du programme appelant à la fonction (le programme appelé) et à partir desquelles il va travailler (faire un calcul, effectuer des opérations d'E/S, etc).

Important : En langage C, les arguments sont passés **par valeurs**. Le programme appelant fournit des valeurs qui sont recopiées dans le contexte du programme appelé (dans la pile : voir plus loin *Pile et récursivité.)*. Il n'y a donc pas modification des variables correspondantes du programme appelant (dans le cas où se sont des variables qui sont utilisées pour le passage d'arguments. Mais on peut aussi utiliser des constantes).

Pour modifier des variables du programme appelant depuis l'intérieur d'une fonction, il faut utiliser l'opérateur **&**, pour passer à la fonction, non les variables (leur contenu) **mais leur adresse**.

Pour renvoyer une valeur, il y a le mot réservé **return** :
return valeur ;

et valeur doit correspondre au type déclaré pour la fonction.
Il peut y avoir plusieurs instructions return dans une fonction ou éventuellement aucune.

Ex1 :

```
int moyenne (int a, int b) {
	int moy ;

	moy = (a+b) / 2 ;
	return moy ;
}
```

Cette fonction renvoie la moyenne de deux nombres donnés en argument

Ex 2 :

```
void affiche_car (char car) {
  printf ("%c",car);
}
```

Cette fonction qui renvoie une valeur vide, grâce au mot réservé void, est l'équivalent d'une procédure qui affiche un caractère.

3.4.2/ Utilisation d'une fonction

L'utilisation d'une fonction (appel) se fait en invoquant son nom dans le contexte d'un **programme appelant**. La signature (ou prototype) de la fonction doit être connue du compilateur quand il rencontre l'appel et le programme appelant doit fournir des symboles pour chaque argument requis, soit des variables, soit des constantes et il doit aussi éventuellement récupérer la valeur renvoyée par la fonction.

Exemples :

a/ appel avec renvoi de valeur

```
int val1 = 5 ;
int val2 = 10 ;
int moy ;

moy = moyenne(5, 10) ;
```

b/ appel sans renvoi de valeur

```
affiche_car('A') ;
```

c/ autre exemple avec renvoi de valeur

```
….
if ( moyenne(u,v) > w ) {
  ….
}
```

Ici la valeur renvoyée la par fonction n'est pas mémorisée dans une variable mais elle utilisée directement dans un test if.

3.4.3/ Déclaration d'une fonction.

Il se peut qu'on utilise une fonction définie ailleurs que dans le module en cours d'écriture (module = fichier source C). Dans ce cas, pour pouvoir utiliser la fonction il faut au préalable la déclarer.

Déclarer une fonction consiste à écrire sa signature ou entête.
Par exemple :

```
int moyenne (int a, int b) ;

void affiche_car (char car) ;
```

C'est un peu comme déclarer une variable (mais à ceci prêt qu'il n'y a pas de place mémoire réservée à cette occasion).
Grâce à la déclaration, le compilateur connaît le nom de la fonction, le type de valeur éventuellement retournée, le nombre et le type des arguments.

Ainsi le compilateur aura toutes les informations nécessaires pour générer le code objet quand il rencontrera une instruction d'appel de la fonction.

Exercice : Écrire une fonction pour calculer la factorielle n! d'un entier n.

Quel est la signature de la fonction ?
Écrire la définition de la fonction

3.4.4/ Fonctions avec nombre d'arguments variables

Nous avons déjà rencontré le cas lors de l'utilisation des fonctions d'Entrées / Sorties formatées printf et scanf, il est possible en langage C de définir une fonction admettant i un nombre variables d'arguments.

La syntaxe de déclaration est la suivante :

```
type-renvoi  MaFonction (type-arg1 arg1, …)
```

Les **trois points …** indiquent un nombre variable d'arguments. Il faut qu'au moins un argument formel soit défini pour la fonction.

Dans la définition de la fonction, pour l'accès à la liste des arguments effectivement saisis, il faut utiliser des macros définies dans le fichier **stdarg.h.**
Pour plus de détails sur les fonctions à nombre variable d'argument, on se reportera au document *Programmation en langage C*, p. 74, d'Anne Canteaut (INRIA) :
https://www.rocq.inria.fr/secret/Anne.Canteaut/COURS_C/

Remarque : On a rarement besoin de cette possibilité de nombre variable d'arguments et il ne faut y recourir qu'en cas de besoin vraiment justifié si d'autres solutions plus simples ne sont pas suffisantes. Le cas des fonctions printf et scanf est cependant un bon exemple de leur mise en œuvre car cela donne une grande souplesse à l'écriture de la chaîne formatée que représente le premier argument.

3.5/ Les pointeurs

3.5.1/Définition

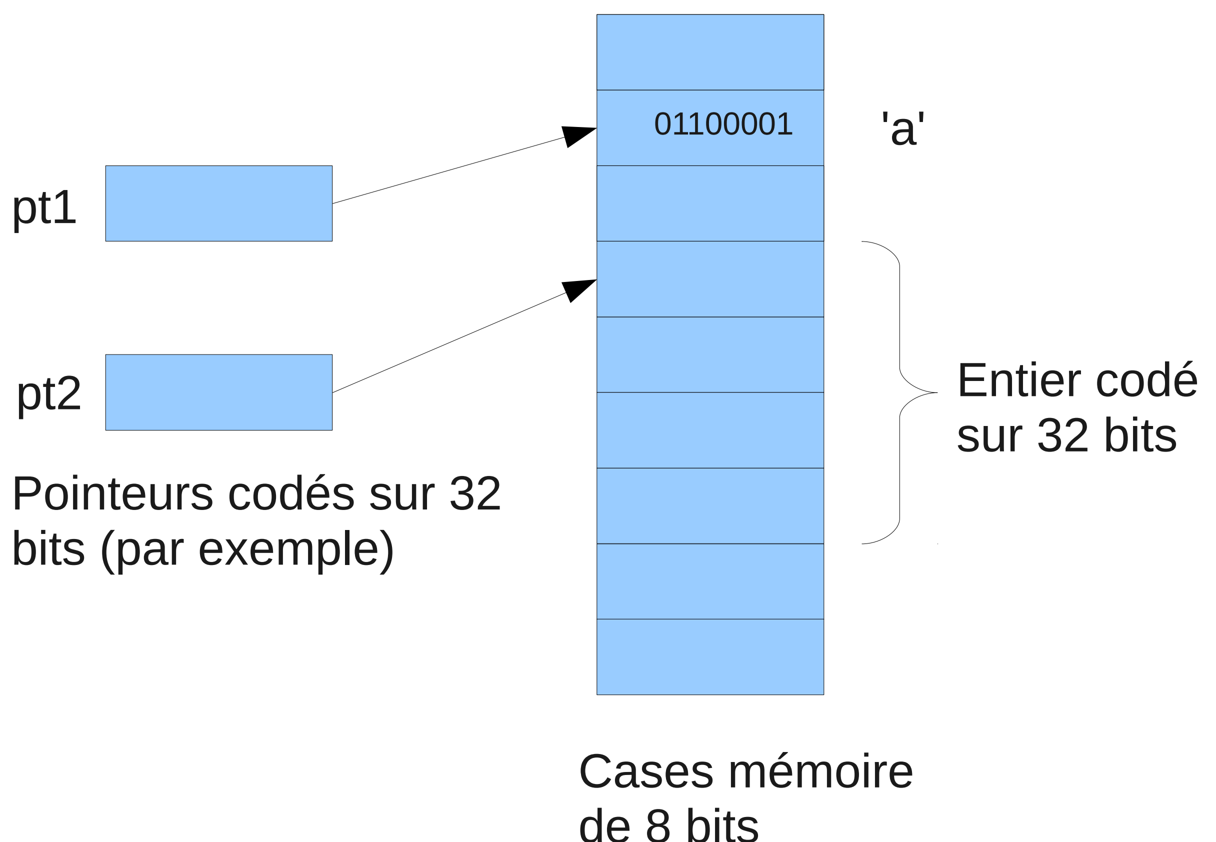
Les pointeurs sont une particularité du langage C.
Un pointeur représente une adresse de la mémoire vive accessible par le programme. C'est un peu comme le nom d'une variable puisque qu'une variable représente aussi un espace de stockage en mémoire. Mais avec une variable, on manipule un contenu et non pas directement les adresses mémoire au contraire des pointeurs.

Voici la syntaxe de déclaration d'un pointeur :

type * un_pointeur ;

Un pointeur est toujours associé à un type (le type de l'élément pointé).
Exemples :
char *pt1 ; /* pointeur sur un caractère */
int *pt2 ; /* pointeur sur entier */

En fait le pointeur est lui-même une variable. C'est une variable qui contient une adresse : l'adresse d'un autre élément associé à un type. On parle aussi d'**indirection** pour qualifier les pointeurs.



En langage C, les pointeurs jouent un rôle central :

- pour transmettre à une fonction un gros volume de données : on passera un pointeur sur la zone de données qui n'aura pas besoin d'être recopiée. Les pointeurs permettent ainsi de pallier aux limitations induites par le seul mode de passage de paramètres (passage par valeur) offert par le langage et de « simuler » d'une certaine manière le passage par référence;
- pour l'accès aux tableaux : tout tableau est de fait un pointeur (sur son début) et tout accès à un élément d'un tableau est traduit en un déplacement en mémoire par rapport à ce pointeur;
- la définition de structures de données dynamiques récursives telles que les listes et les arbres.

3.5.2/ Affectation et manipulation d'un pointeur

Comment placer l'adresse d'un élément dans un pointeur ?

a/ cas des variables simples

On accède à l'adresse d'une variable grâce à le symbole **&** placé devant le nom de la variable.
Ainsi après les deux lignes de déclarations suivantes :

```
int a=4 ;
int *pt  ;
```

l'instruction :

```
pt = &a ;
```

affecte le pointeur pt avec l'adresse de l'entier a.

Ensuite, pour accéder à la valeur de l'élément pointé, un faut utiliser le symbole ***** placé devant le nom du pointeur.

Dans l'exemple précédent, puisque pt contient l'adresse de l'entier a, *pt représente la valeur de a.
Ainsi après :

```
int a=4 ;
int b ;
int *pt=&a  ;

b = *pt ;
```

l'entier b contient la même valeur que a.

b/ cas des tableaux

En langage C, **le nom d'un tableau est un pointeur constant**. C'est en fait l'adresse du premier élément du tableau.
Lors de la déclaration :

```
char tab[5] ;
```

le compilateur réserve la place de 5 caractères en mémoire et place l'adresse du premier caractère dans une zone mémoire référencé par le nom tab.

On peut écrire le code suivant :

```
char *pt ;
pt = tab ;
```

le pointeur pt « pointe » alors sur le premier élément élément du tableau est les deux écritures *pt et tab[0] sont équivalentes.

De même *(pt+1) et tab[1] sont équivalents. On peut même écrire pt[1].
On peut aussi dire que pt et &pt[0] sont équivalents, ils représentent tous les deux la même adresse.

c/ Le pointeur NULL

Il existe une constante NULL qui permet souvent à des fonctions renvoyant un pointeur de retourner cette valeur NULL comme code d'erreur.
On peut l'utiliser aussi pour initialiser un pointeur non encore affecté :
char * pt = NULL ;

L'intérêt est qu'ainsi on pourra tester le pointeur qui n'est pas laissé à une valeur aléatoire.

3.5.3/ Arithmétique des pointeurs

L'intérêt des pointeurs est de pouvoir les traiter comme des variables, c'est à dire faire sur eux des opérations « arithmétiques ». En fait il s'agit toujours d'ajouter un certain offset (décalage mémoire) à un pointeur. Cela permet de parcourir des structures comme les tableaux, de réaliser certaines opérations fines en mémoire et les pointeurs sont très utilisés pour les opérations d'**allocation dynamique** de mémoire.

Les opérations autorisées sur les pointeurs sont :

l'affectation (pt1 = pt2), l'incrémentation (pt++ ou ++pt), la décrémentation (pt-- ou –pt) , l'addition d'un entier (pt+5), la soustraction d'un entier (pt-5).

Remarque très importante :

Il s'agit d'une arithmétique contrôlée (c'est le compilateur qui s'en charge) par le type de la variable déclarée pour le pointeur.

Soit les déclarions :

```
int tab1[5] = {1, 2, 3, 4, 5} ;
long tab2[5] = {1L , 2L, 3L, 4L, 5L} ;
int *pt1=tab1 ;
long *pt2=tab2 ;
```

si maintenant on écrit :

pt1=pt1+3 ;

pt1 représente le quatrième élément du tableau tab1 (ici *pt1 vaut 4).

de même

pt2 = pt2+3 représente le quatrième élément du tableau tab2 (ici *pt2 vaut 4).

Dans le premier cas, pt1 aura été incrémenté de 3 * 4 octets, c'est à dire 3 * sizeof(int)
Dans le second cas, pt2 aura été incrémenté de 3 * 8 octets, c'est à dire 3 * sizeof(long).

Si maintenant j'utilise un pointeur sur char pour parcourir mes tableaux ;
char *pt ;
j'ai le droit d'affecter mon pointeur de la manière suivante :

pt = (char *)tab1 ; / *casting du pointeur constant tab1 pour l'affecter à pt qui est d'un autre type */

et alors (pt+12) représente l'adresse du quatrième élément.

Par contre avec :

pt = (char *)tab2 ;

l'adresse du quatrième élément est (pt+24) ;

Exemple d'application : copie d'une chaîne de caractères dans une autre.

Rappel : une chaîne de caractères n'est autre qu'un tableau de caractères terminé par le caractère spécial nul.

```
char str1[1000] ;
char str2[1000] ;
char *pt1, *pt2 ;

pt1 = str1 ;
pt2 = str2 ;
while(*pt2++ = *pt1++) ;
```

3.5.4/ Pointeurs et tableaux à plusieurs dimensions

Il est possible d'avoir des tableaux à plusieurs dimensions. Mais en langage C, il ne faut pas penser un tableau à deux dimensions comme une matrice avec des valeurs rangées en lignes et colonnes. Un tableau à plusieurs dimensions en langage C est un **tableau de tableaux.**

Voici la déclaration d'un tableau à deux dimensions contenat des entiers.

```
int tab[5][6] ;
```

tab représente une **adresse** mémoire fixe (contrairement à un pointeur qu'on pourrait incrémenter) et le compilateur réservera 5 places pour les adresses de 5 tableaux de 6 entiers. On a donc un **tableau de pointeurs vers des entiers.**
tab[0], tab[1],..., tab[4] sont des **pointeurs** vers les premiers éléments de 5 tableaux de 6 entiers.
tab[1][2] est le troisième **entier** du second tableau.

La déclaration suivante :

```
char * tab[5] ;
```

représente la déclaration d'un tableau de 5 pointeurs sur des caractères. C'est aussi un tableau à deux dimensions, donc un tableau de chaînes de caractères. **Mais attention**, ici seul l'espace pour les 5 pointeurs sera réservé par le compilateur. L'espace pour les caractères à stocker devra être réservé dynamiquement (sauf les les chaînes existent déjà par ailleurs).

Cela rappelle la déclaration de la fonction main du langage C :

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

Autre cas :

```
char **pt ;
```

Cette dernière déclaration représente un pointeur de pointeurs sur des caractères. Il y a une double indirection comme dans une déclaration tab[][]. Le pointeur pt peut donc permettre de parcourir un tableau à deux dimensions.

3.5.6/ Pointeurs sur des structures

Ce point est très important car les objets de type struture sont le plus souvent manipulés par l'intermédiaire de pointeurs en langage C.

Nous avons vu le type structure précédemment dans notre étude des types de données complexes (composés) du langage C. Grâce à l'opérateur typedef, on peut même créer de nouveaux types basés sur des structures.

a/ Rappel

Reprenons l'exemple précédent de la structure servant à mémoriser l'adresse d'un individu :

```
struct adresse {
        int id_personne ;
        char rue[40] ;
        char code[10] ;
        char ville[20] ;
} ;
```

Pour définir *adresse* comme un nouveau type de données, plutôt que de devoir déclarer des variables en répétant *struct adresse* à chaque fois, je vais déclarer *adresse* comme suit :

```
typedef struct {
	int id_personne ;
	char rue[40] ;
	char code[10] ;
	char ville[20] ;
} adresse ;
```

Maintenant adresse est un **véritable type** de données. Je peux déclarer mes variables de type adresse ainsi :

```
adresse adr1, adr2, adr3 ;
```

b / Avec des pointeurs

En pratique, il est plus courant de manipuler les structures à l'aide de pointeurs. En effet, une structure pouvant être un objet très complexe, les passages en argument à des fonctions ou les recopies de structures peuvent demander des traitements particuliers. Donc il est plus naturel de passer à une fonction **l'adresse d'une structure,** d(où l'utilisation d'un pointeur.

Suivant l'exemple précédent, voici la déclaration d'un pointeur sur une adresse :

```
adresse *p_adr ;
```

p_adr est un pointeur sur un objet de type adresse. Mais attention, après cette déclaration, seule la place pour le pointeur (l'adresse mémoire) a été réservée, et non pas la place pour une variable de type adresse.

Plus conséquent serait donc :

```
adresse adr1 ;
adresse *p_adr = &adr1 ;
```

La première déclaration créé une variable de type adresse avec réservation de la mémoire (statique) nécessaire. La seconde déclaration créé un pointeur sur l'adresse.

Pour accéder au champs de la structure adresse, il faut utiliser la notation suivante :

```
(*p_adr).ville
```

On prend le contenu de ce qui est pointé grâce à l'opérateur *étoile* puis on accède au champ grâce à

l'opérateur *point.* Les parenthèses sont indispensables car l'opérateur *point* est plus prioritaire que l'étoile.

Pour placer quelque chose dans le champ ville qui est un tableau de caractères, je peux utiliser une instruction du genre :

strcpy((*p_adr).ville , "Bordeaux") ;

(strcpy est une fonction de la bibliothèque standard du C qui demande l'inclusion du fichier d'entête <string.h>. Le premier argument est la destination de la copie, le second argument est la chaîne source à copier)

c/ L'opérateur « flèche »

La notation (*p_adr).ville n'étant pas très lisible, un opérateur existe dans le langage C qui donne exactement le même résultat : c'est l'opérateur **->**

Ainsi, l'accès au champ ville à partir du pointeur p_adr s'écrit :

p_adr->ville

par exemple :

strcpy(p_adr->ville , "Bordeaux") ;

3.6/ La portée des variables et l'allocation dynamique de mémoire

La durée de vie d'une variable en mémoire, sa visibilité par les différents éléments d'un programme dépendent de la **portée** de la variable. Celle-ci est en particulier déterminée par l'endroit où une variable est déclarée.

3.6.1/ Les variables locales

Elles ne sont accessibles que dans le corps d'une fonction et leur durée de vie n'excède pas le temps d'exécution de la fonction.
Exception : les variables **statique**
Une variable locale dont la déclaration est précédée du mot réservé **static** est conservée après retour de la fonction. Lors d'un nouvel appel de la fonction, la valeur de la variable sera celle de la fin de l'appel précédent de la fonction.
Une variable statique est donc comme une variable globale mais sa portée, c'est à dire son accessibilité, est limitée à l'intérieur de la fonction.

3.6.2/ Les arguments

Les arguments fonctionnent un peu comme des variables locales mais dans lesquelles des valeurs initiales sont recopiées. Ce sont les valeurs passées lors de l'appel de la fonction. Mais ensuite on travaille sur une **copie** de ces valeurs. Donc les variables (éventuelles) servant à passer ces valeurs ne sont pas modifiées dans le programme appelant quand on affecte les variables correspondant aux arguments.

Remarque : Si l'on veut modifier, dans une fonction, le contenu d'une variable donnée en argument , il faut passer l'**adresse** de cette variable, c'est à dire un pointeur sur cette variable. Il faut bien sûr que la fonction ait été définie avec un pointeur pour l'argument correspondant.

exercice :
Voici l'entête d'une fonction écrite pour permuter deux nombres entiers :
void permutte (int *a, int *b) ;

écrire la définition de la fonction et l'appeler dans un programme

3.6.5/ La pile et la récursivité

Les variables locales et les arguments sont placés dans la **pile**. C'est une zone mémoire de données temporaire propre à chaque programme. Les données y sont placées (empilées) et retirées (dépilées) suivant un mécanisme particulier de **dernier arrivé, premier reparti** (FIFO : first in, first out).

L'utilisation de la pile est fondamentale lors de l'appel des fonctions. Les arguments et données locales y sont empilés mais préalablement, c'est l'adresse de retour du **pointeur d'instruction** (c'est le contenu d'un registre particulier du microprocesseur qui contient en permanence l'adresse de la prochaine instruction à exécuter du programme en cours) qui y est placée.

Ce mécanisme autorise le langage C à définir des **fonctions récursives**.
Une fonction récursive est une fonctions dont la définition fait appel à elle même. Par exemple je peux définir une fonction qui modifie les attributs de tous les fichiers d'un répertoire. Si je rencontre un répertoire dans le répertoire en cours de traitement, la fonction peut faire appel à elle même pour traiter ce sous répertoire.

Ainsi :

```
fonction ModifierAttributRépertoire(répertoire) {

        pour chaque élément de répertoire faire
                si élément est un fichier alors
                        modifier attribut fichier
                sinon si élément est un sous_répertoire alors
                        ModifierAttributRépertoire(sous_répertoire)
                fsi
        fpour
}
```

C'est une fonction récursive qui s'appelle elle-même pour traiter les sous-répertoires. Or la hiérarchie peut comporter plusieurs niveaux, voire un nombre indéfini de niveaux.
Or la fonction, à chaque retour à un niveau après avoir traité un niveau inférieur, doit retrouver son contexte d'exécution, c'est à dire l'instruction courante et les valeurs des variables locales.

Cela est possible grâce aux empilements successifs des contextes d'exécution de la fonction lors de chaque appel.

Exercice : écrire une fonction récursive qui calcule la factorielle d'un entier positif ou nul.
Pour rappel la factorielle d'un entier n, notée n!, est définie comme suit :
0! = 1
n! = n x (n-1)! si n >= 1

3.6.4/ Les variables globales

Elles sont définies à l'extérieur des fonctions et accessibles partout dans le programme, à l'intérieur de toutes les fonctions. Un programme comportant plusieurs fichiers doit définir une variable globale dans l'un des fichiers et les autres fichiers doivent la référencer par le mot réservé **extern**.

Par exemple :

extern int temps ;

En général , les déclaration extern sont placées dans les fichiers .h définissant les constantes, variables et fonctions exportées par un modules.

Les variables globales sont placées en mémoire dans la **zone statique**. C'est la zone où est placée tout ce qui est connu au moment de la compilation, dont le code du programme compilé lui-même. Et précisément l'espace utilisé par une variables globale est connu au moment de la compilation, il peut donc être placé en zone statique.

3.6.5/ La mémoire dynamique

Outre la zone «programme » qui contient les instructions et les variables globales, la « pile » qui contient les arguments et des variables locales, il existe une troisième zone mémoire où peuvent être placées les données utilisées par un programme : l**e tas** (heap en anglais).

Le tas est la zone de la **mémoire dynamique.** La mémoire dynamique est réservée et libérée par le système d'exploitation à la demande du programme.
Le langage C dispose pour cela de deux fonctions spéciales présentes dans la librairie standard <stdlib.h> : **malloc** (allocation dynamique de mémoire) et **free** (libération de mémoire précédemment allouées) dont les signatures sont :

char * malloc(int size) ;
void free (char * pt) ;

3.7/ Travailler avec les fichiers

La librairie standard du C offre un mécanisme d'entrées/sorties sur fichiers. Les accès aux fichiers sont réalisées au travers d'un « flux » (« stream » dans les documentations en anglais) qui correspond à une mémoire tampon (buffer). Le système créé donc une abstraction entre les fichiers réels sur disque dur (ou d'autres dispositifs qui sont présentés comme des fichiers, avec le système UNIX notamment).

La première fonction à appeler pour ouvrir un flux sur un fichier est fopen dont voici le prototype :

FILE * fopen(char * chemin_du_fichier, char * mode) ;

FILE est une structure, représentant un flux , définie dans le le fichier d'entête stdio.h

Le mode, exprimé sous la forme d'une chaîne de caractères, détermine ce que l'on veut faire avec le fichier.
Les modes sont les suivants :

"r" : ouverture d'un fichier « texte » en lecture seule
"w" : ouverture d'un fichier « texte » en écriture seule (avec écrasement de tout contenu préexistant).
"a" : mode ajout en mode texte, c'est à dire écriture à la fin du fichier, le contenu existant étant conservé.
"r+" : ouverture d'un fichier « texte » en lecture / écriture
"w+" : ouverture d'un fichier « texte » en lecture / écriture avec écrasement de tout contenu préexistant.
"a+" : mode ajout d'un fichier « texte » en lecture / écriture à partir de la fin du fichier.

Ces modes correspondent aux fichiers « textes » c'est à dire à des fichiers composés de lignes séparées par le caractère spécial '\r '. Ce caractère spécial du langage C est interprété différemment sur les différentes plates-formes (carriage return + line feed sur système Unix).

Les mêmes modes existent pour les fichiers binaires, c'est à dire les fichiers pour lesquels on n’interprète à priori aucun caractère de manière spéciale. Les modes précédents, appliqués aux fichiers binaires, s'écrivent ainsi :
"rb", "wb", "ab", "r+b", "w+b", "a+b"

Toutes les fonctions de travail sur les flux sont également déclarées dans stdio.h. Voici les principales :

int fclose(FILE *stream) ;
int fgetc(FILE *stream) ;
char *fgets(char *s, int n, FILE *stream) ;
int fputc(int c, FILE *stream) ;
int fputs(const char *s, FILE *stream) ;
int fscanf(FILE *stream, const char *format, …) ;
int fprintf(FILE *stream, const char *format, …) ;
size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream) ;
size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *stream) ;

```
int fseek(FILE *stream, long offset, int origin) ;
long ftell(FILE *stream) ;
```

Voici un exemple minimaliste de création et d'écriture dans un fichier binaire :

```
FILE *f;
f=fopen("c:\\test.bin", "wb");
char x[25]="Ma chaîne de caractères";
fwrite(x, sizeof(x[0]), sizeof(x)/sizeof(x[0]), f);
fclose(fp) ;
```