Algorithme et langage C .

Michaël Quisquater (Maître de Conférences, UVSQ)

Première partie I

Quelques références

Quelques réfŕences utiles et bibliographie

Sur le C...

- The C programming Language. Brian W. Kernighan and Dennis M. Ritchie
- Programmation en langage C. Anne Canteaut. (disponible sur le web)

Astuces de programmation...

- Hackers's Delight. Henry S. Warren, Jr.
- Numerical Recipes in C : The Art of Scientific Computing. William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery.

Quelques réfŕences utiles (suite)

Un peu d'algorithmique...

- The art of computeer Programming (vol I-II-III). D. Knuth.
- Introduction à l'algorithmique. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest et Clifford Stein.
- Modern Computer Algebra. von zur Gathen and Gerhard.
- Algorithmic Cryptanalysis. A. Joux.

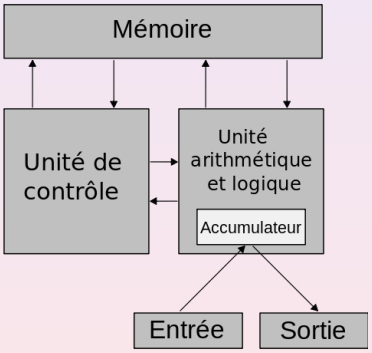
Deuxième partie II

Architecture des ordinateurs

Troisième partie III

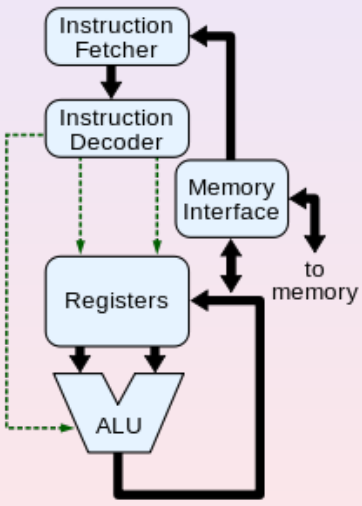
Architecture des ordinateurs et notion de programme informatique

Architecture de von Neumann



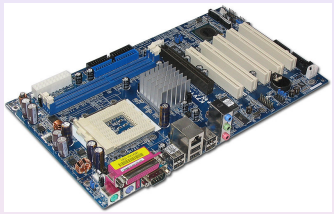
- Unité arithmétique et logique (UAL ou ALU) : effectue les opérations de base
- Unité de contrôle : chargé du séquençage des opérations
- Mémoire : contient à la fois les données et le programme qui dira à l'unité de contrôle quels calculs à effectuer sur ces données
- Mémoire=mémoire volatile + mémoire permanente
- Entrée-Sortie : permet de communiquer avec le monde extérieur.

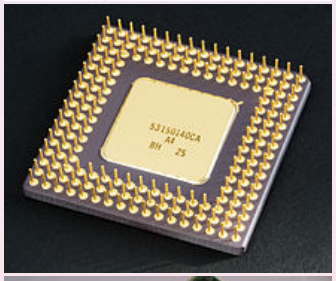
Étapes d'exécution d'une instruction

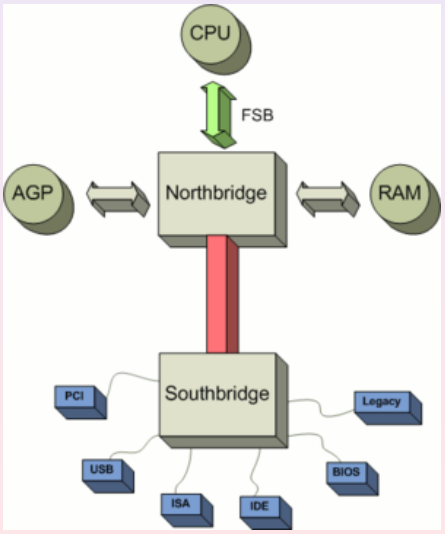


- Fetch : Rechercher instruction
- Decode : décodage de l'instruction (opération et opérandes)
- Execute : Exécution de l'opération
- Writeback : Écriture résultat

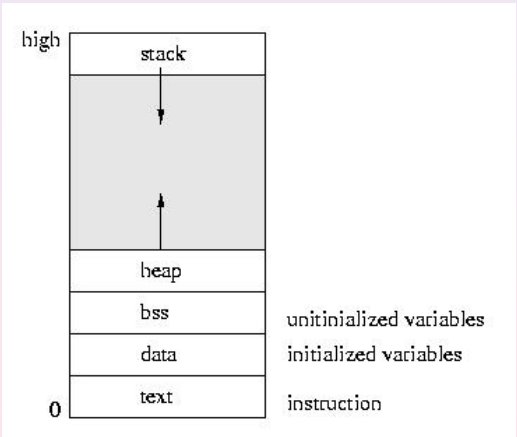
Carte mère/CPU/RAM/Cache





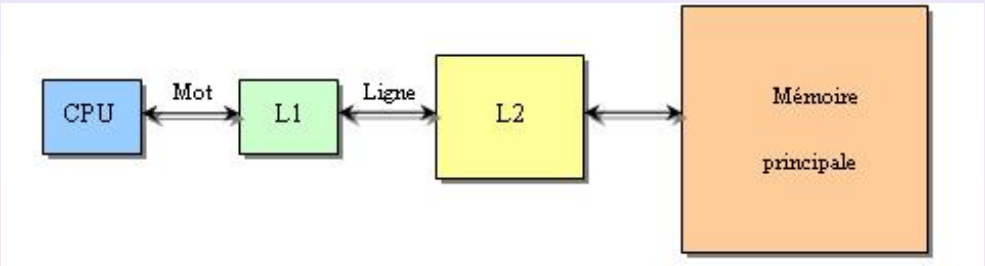


Mémoire RAM



- Stack (Pile) : zone dans laquelle s'exécute le programme.
  - paramètres, variables locales, adresse, valeur retour de fonction
  - croît à chaque appel, décroît à chaque retour
- Heap (Tas) : zone où le programme alloue toutes les données qui ne sont pas en pile. ex : malloc.
- Text : zone où se trouve le programme à exécuter (souvent accès en lecture seule).

Notion de mémoire cache



- En pratique, le processeur ne va pas chercher les données ou les instructions immédiatement dans la mémoire RAM. Les données par différentes mémoires appelées *caches*.
- Généralement, il y a 3 niveaux de caches ; L1, L2 et L3. L1 est le cache le plus rapide mais aussi le plus petit.
- Les mémoires caches sont intégrés dans le CPU.

Rôle du northbridge

- Une des deux puces du chipset de la carte mère.
- Gère les communications entre le microprocesseur, la RAM, les ports AGP ou PCI express et le southbridge.
- Parfois le northbridge intègre une carte graphique (peu performante mais de faible coût).

Remarques :

- le port AGP (Advance Graphics Port) permet de connecter une carte graphique mais remplacé par PCI express.
- Le port PCI express permet de connecter beaucoup de périphériques et est censé remplacer le PCI (Peripheral Component Interconnect).

Rôle du southbridge

- Une des deux puces du chipset de la carte mère.
- Définit et commande tout ce qui n'est pas géré par le northbridge (PCI, interface PS/2 pour le clavier et la souris, (le port série), (le port parallèle)). Certaines de ces fonctions sont souvent prises en charge par un contrôleur secondaire d'I/O. Dans ce cas, le southbridge fournit une interface à ce contrôleur.
- Il sert aussi de support pour l'interface parallèle ATA (IDE), SATA (disque dur, CDROM), pour l'interface ethernet (réseau), l'USB (Universal Serial Bus) et IEEE1994 (firewire). Parfois le southbridge intègre une carte son.

Évolution

- CPU : Central Processing Unit
- NB : Northbridge
- SB : Southbridge
- GPU : Graphics Processing Unit

Remarque : Notons que l'architecture Northbridge/Southbridge a tendance à disparaître. Le Northbridge est parfois intégré au processeur ou bien il y a une seule puce pour NB/SB/GPU.

Exemple de processeur (CPU : Central Processing Unit) :

- Marque/modèle : Intel I5
- Fréquence : 2.8 Ghz
- Nombre de coeurs : 4
- taille des caches : L1=4*64Ko, L2=4*256Ko, L3=8Mo (partagé)

Exemple de mémoire RAM :

- Marque(s) : Kingstone ou Corsair
- Capacité : 4Go ou 8 Go parfois plus.

C’est quoi un programme informatique

Programme informatique

- séquence d’instructions qui spécifie étape par étape les opérations à effectuer pour obtenir un résultat
- exprimé sous une forme compréhensible par un ordinateur (éventuellement après une traduction adéquate).

Remarque : En 1842 la comtesse Ada Lovelace crée des diagrammes pour la machine analytique de Babbage, diagrammes qui sont considérés aujourd’hui comme étant les premiers programmes informatiques au monde.

Langage de programmation et notion de compilation d'un programme

- Un ordinateur ne peut exécuter que des programmes binaires (langage machine) qui utilisent des instructions propres au processeur de celui-ci.
- Un programme binaire est difficilement par l'homme.
- Un programme assembleur est la traduction d'un programme binaire en quelque chose d'un peu plus lisible. Exemple : l'instruction machine 10110000 01100001 s'écrit en langage assembleur "movb $0x61,%al". Ce qui signifie "écrire le nombre 97 dans le registre AL". L'assemblage consiste à traduire un programme écrit en assembleur en un programme binaire.

Langage de programmation et notion de compilation d'un programme

- Afin de faciliter l'écriture de programmes et l'éxécution d'un même programme sur différents types d'ordinateur, des langages de plus hauts niveaux ont été développés.
- Un programme spécifique, appelé compilateur, permettant de traduire un programme écrit en ce langage de haut niveau en langage assembleur a été developpé.
- Un autre programme, appelé préprocesseur, fonctionne souvent en amont du compilateur et permet une certaine automatisation de la génération du code à compiler. Exemple : répétition de morceaux de codes, inclure des fichiers de façon intelligente les uns dans les autres, enlever les commentaires etc

Langage de programmation et notion de compilation d'un programme

En résumé, la compilation se décompose en 4 phases :

- Traitement par le préprocesseur.
- Compilation.
- Assemblage.
- Edition de liens

Quatrième partie IV

Base du langage C

La compilation d'un programme en C : gcc

Le programme de compilation GNU est gcc.
Un programme écrit en C a un nom de la forme : fichier.c
La syntaxe d'utilisation

gcc [options] fichier.c [-l*librairies*]

où

[options] : les options permettent de modifier le comportement du compilateur (n'effectuer que certaines phases de la compilation, optimisation, etc)

[-l*librairies*] : il est parfois nécessaire de spécifier les librairies (pré-compilées) utilisées afin que l'édition de liens puisse être faite. Le compilateur cherche la librairie lib*libraries*.a, généralement dans /usr/lib.

La compilation d'un programme en C : gcc

Les options courantes :

- -E : seul le préprocesseur est appliqué. Le code fourni est dirigé vers la sortie standard (Shell)
- -o *nom_fichier* : permet de changer le nom de l'exécutable. Sans spécification particulière gcc produit un fichier exécutable nommé **a.out**
- -O, -O1, -O2, -O3 : permet d'optimiser l'efficacité du code binaire produit (au plus le chiffre est élevé, au plus la compilation prend du temps et le code binaire fourni est efficace).
- -S : produit un fichier assembleur.
- -W : affiche davantage de messages d'avertissement.
- -Wall : affiche tous les messages d'avertissement.

La compilation d'un programme en C : gcc

- Afin d'exécuter un exécutable *file* (ex. a.out), il suffit de taper *file* dans le Shell dans le repertoire où il se trouve.
- Si l'on souhaite connaître le temps d'exécution du programme, il suffit d'utiliser la commande *time*. Exemple : *time file*
  - Real : temps séparant le début du lancement du programme à sa terminaison (cela inclut les temps d'attente pour les I/O et le temps consacré aux autres processus).
  - User : temps consacré au processus lancé sans tenir compte des appels systèmes.
  - Sys : temps consacré aux appels systèmes initié par le processus.

Les composants élémentaires du C

- Les identificateurs (ex. nom de variable, d'un nouveau type etc),
- Les mots-clefs (ex. nom d'une intruction, d'un type pré-défini),
- Les constantes (ex. A qui représente 3),
- Les chaînes de caractères (ex. "Hello World"),
- Les opérateurs (ex. +,*,« etc),
- Les signes de ponctuation (ex. ;).

Il existe aussi les commentaires (ex. /*commentaire */).

Expression, instruction et déclaration

- Une *expression* est une suite de composants élémentaires syntaxiquement correcte. ex. x=0 , (a>0)&&(b=2)
- Une *instruction* est une expression suivie d'une point virgule (qui signifie "évaluer cette expression"). Plusieurs instructions peuvent être regroupées par des accolades pour former une *instruction composée*.

Exemple :

```
if (i!=1)
 {
   a=0;
   b=1;
 }
```


Les intructions composées ou bloc

Plusieurs intructions peuvent être rassemblées en une seule, appelée instruction composée ou bloc.
La syntaxe est :

```
{
  instruction-1
  instruction-2
    ....
}
```

Les intructions composées ou bloc (suite)

Exemple :

```
{
  a=6;
  b=7;
}
```


Expression, instruction et déclaration (suite)

- Le langage C est typé cela signifie que les variables et constantes ont un type. Ces types déterminent la façon dont ces objets sont représentés en mémoire. En début de programme, il faut déclarer les objets (variables/constantes) que l'on va utiliser dans la suite.
- Une *déclaration* est une instruction composée d'un spécificateur de type et d'une liste d'identificateurs séparés par une virgule. Une déclaration consiste à associer l'adresse mémoire de l'objet, sa représentation et son identificateur.

Exemple :

```
int a;
int b,c;
```

Un premier programme en C

```
#include<stdio.h>
int main()
{
printf("Hello world\n");
}
```


Structure d'un programme en C

```
[ directive au préprocesseur]
[ déclarations de variables externes]
[ prototypes des fonctions secondaires]
[ déclaration des fonctions secondaires]

int main()
{
déclaration de variables internes
intructions
}
```

Déclaration d'une fonction secondaire

La déclaration d'une fonction secondaire se place devant ou derrière la fonction principale main.
La syntaxe est :

```
type fonction_secondaire ( arguments )
{
 déclarations de variables internes
 instructions
}
```

- Cette fonction retrournera un objet de type *type* via le mot-clef **return**.
- La syntaxe des *arguments* est proche de celle des déclarations de variables.

Déclaration d'une fonction secondaire (suite)

```
int addition(int a, int b)
{
   int temp;

   temp=a+b;
   return temp;
}
```

Appel de fonction

Pour utiliser une fonction, on fait un appel de fonction en mentionnant la fonction muni de la spécification de ses arguments.
La syntaxe est :

Nom_Fonction(expression_1,...,expression_n)

Remarque :

- L'ordre dans lequel les expressions sont évaluées n'est pas déterminé. Il faut donc utiliser des variables temporaires pour éviter toute confusion.

Appel de fonction (suite)

```
#include<stdio.h>

int max(int a, int b)
{
  if (a>=b)
    return a;
  else
    return b;
}

int main()
{
  int i=2, i=3;
  printf("le maximum vaut %d\n", max(i,j));
}
```

Prototypes des fonctions secondaires

Il est recommandé de placer avant la fonction main d'un programme les prototypes des fonctions secondaires.
La syntaxe d'un prototype pour une fonction est :

type fonction_secondaire (type_des_arguments) ;

- Il s'agit de mentionner le format d'une fonction en spécifiant les types des arguments et de la valeur de retour.
- Si ces prototypes ne sont pas présents il est possible que certaines erreurs de conversion de type se produisent (voir plus loin).

Prototypes des fonctions secondaires (suite)

Exemple :

```
#include<stdio.h>
int max(int, int);
int max(int a, int b)
{
  if (a>=b)
    return a;
  else
    return b;
}
int main()
{
  int i=2, i=3;
  printf("le maximum vaut %d\n", max(i,j));
}
```

Variable locale et globale

- Une variable globale est sauvergardée dans la partie "initialized variable" ou dans le bss "uninitialized variable" en fonction de son initialisation. Cette variable est déclarée en dehors de toute fonction c-à-d au dessus du main.
- Une variable locale est déclarée à l'intérieur du fonction et n'est accessible qu'à l'intérieur de celle-ci. Elle peut-être statique ou non.

Variable globale : Exemple

```
#include<stdio.h>
int c=4;
int increment()
{
    c++;
    return c;
}
int main()
{
  increment();
  increment();
  printf("c vaut %d\n",increment());
}
```

c vaut 7.

Variable locale : Exemple

```
#include<stdio.h>
int increment()
{
    int c=4;
    c++;
    return c;
}
int main()
{
  increment();
  increment();
  printf("c vaut %d\n",increment());
}
```

c vaut 5.

Variables statiques et temporaires

- Une variable statique est une variable qui est attribuée à une fonction au moment de la compilation. Si elle est initialisée lors de la déclaration elle est sauvergardée dans la partie "initialized variable" (en dessous du tas généralement) sinon elle se trouve dans bss "uninitialized variable" (également en dessous du tas généralement). Un variable statique non initialisée spécifiquement est initialisée à zéro automatiquement. La valeur d'une variable statique est conservée entre deux appels de fonction puisqu'il s'agit du même espace mémoire.
- Une variable temporaire est une variable qui est créée dans la pile lors de l'appel de fonction. Elle est détruite en sortie de la fonction.

Variables statiques et temporaires : Exemple

```
#include<stdio.h>
int increment()
{
    static int c=4;
    c++;
    return c;
}
int main()
{
  increment();
  increment();
  printf("c vaut %d\n",increment());
}
```

c vaut 7.

Variables statiques et temporaires : Exemple

```
#include<stdio.h>
int increment()
{
    int c=4;
    c++;
    return c;
}
int main()
{
  increment();
  increment();
  printf("c vaut %d\n",increment());
}
```

c vaut 5.

Représentation d'une valeur en C

- Valeurs entières
- Valeurs en virgule flottante
- Valeurs caractères
- Valeurs chaînes de caractères

Représentation d'une valeur en C : Valeurs entières

- Décimale : 12321
- Octale : 01717
- Hexadécimale : 0xa4e12

Représentation d'une valeur en C : Valeurs en virgule flottante

Une valeur d'un nombre en virgule flottante est représenté par son signe, sa mantisse et son exposant.
Exemple :

Valeur	type
1.25	double
1.25e-5 ou 1.25E-5	double
1.25F ou 1.25f	float
1.25L ou 1.25l	long double

Représentation d'une valeur en C : Valeurs caractères

- Un caractère imprimable est mis entre apostrophes afin de le distinguer de l'identificateur d'une variable. Par exemple, 'a' représente le caractère **a**.
- Antislash est désigné par \\
- l'apostrophe est désigné par \'

Il existe aussi des caractères non-imprimables :

\n	nouvelle ligne
\t	tabulation horizontale
\v	tabulation verticale
\b	retour en arrière
\f	saut de page

Représentation d'une valeur en C : Valeurs chaînes de caractères

- Une chaîne de caractères est une suite de caractères entourés par des guillemets.
  Exemple : "Hello world"
- Une chaîne de caractères peut contenir des caractères non-imprimables.
  Exemple : "Hello `\n` world"

Représentation d'une valeur en C : Valeurs chaînes de caractères (suite)

Remarques :

- le caractère " doit être désigné par \"
- le caractère \ suivi d'un retour chariot est ignoré.

Exemple :

```
"alibaba \
et les 40 voleurs"
```

est équivalent à

```
"alibaba et les 40 voleurs"
```

Par contre, la chaîne suivante provoquera une erreur :

```
"alibaba
et les 40 voleurs"
```

Les types prédéfinis

- Une variable ou une constante est d'un type précis.
- Le type d'un objet (variable ou constante) définit la façon dont il est représenté en mémoire.

Il existe trois ensembles de types prédéfinis

- les caractères,
- les entiers,
- les flottants.

Les types prédéfinis : les caractères

- Le type caractère est dénommé par le mot clef **char**.
- Un objet de type char est codé sur un octet.
- Le jeu de caractère utilisé en C est le code ASCII (codage sur 7 bits).

Exemple :

```
int main()
{
char a;
}
```

Les types prédéfinis : les entiers

- Le type entier est dénommé par le mot clef **int**.
- Sur un ordinateur 32 bits, un entier est codé sur 4 octets. Le bit de poids fort d'un entier est son signe
- Le type int peut être précédé d'un attribut de précision (short ou long)
- Le type int peut être précédé d'un attribut de représentation (unsigned)

Les types prédéfinis : les entiers (suite)

	Intel 32bits
int	32 bits
short (int)	16 bits
long (int)	64 bits

Les types prédéfinis : les entiers (suite)

Exemple :

```
int main()
{
int a;
short b;
unsigned int c;
unsigned short d;
}
```


Les types prédéfinis : les flottants

- Un nombre en virgule flottante :

$$\text{signe } 0, \text{mantisse} * B^{\text{exposant}}$$

où B est une base définie (exemple B=2 ou B=10).
- Les types flottants sont dénommés par le mots clef **float**, **double** et **long double**.
- La mantisse et l'exposant étant de longueur finie, un nombre en virgule flottant n'est ni un rationnel ni un réel.
- Le bit de poids fort d'un flottant est son signe, le champ du milieu correspond à l'exposant et les bits de poids faible à la mantisse.

Les types prédéfinis : les flottants (suite)

	Intel 32bits
float	32 bits
double	64 bits
long double	128 bits

Les types prédéfinis : les flottants (suite)

Exemple :

```
int main()
{
float a;
double b;
long double c;
}
```

La fonction **sizeof**

La fonction **sizeof** prend en argument l'identificateur d'une variable d'un certain type ou le mot clef correspondant à un certain type et retourne le nombre d'octets nécessaires pour représenter un objet de ce type.

Exemple :

```
unsigned short a;

sizeof(a)
sizeof(unsigned short)
```


Cinquième partie V

Les opérateurs

L'affectation

- L'affectation consiste à attribuer le résultat d'une expression à une variable. Il s'agit donc de mettre dans la case mémoire dont l'adresse correspond à l'identificateur une valeur déterminée, valeur qui est obtenue en évaluant une expression.
- Notons qu'il est possible de combiner la déclaration d'une variable et son affectation.

L'affectation est symbolisée par le signe $=$.

$$\textit{variable} = \textit{expression};$$

L'affectation (suite)

Exemple :

```
int main()
{
int a;  /*déclare une variable de type int*/
unsigned int b=0x2; /*déclare une variable de type int
                      et lui affecte la valeur 0x2*/
unsigned int c=(b^0x1); /*déclare une variable de type
                          int et lui affecte la valeur
                          résultant de l'évaluation de
                          l'expression (b^0x1)*/
a=3;  /*affecte la valeur 3 à la variable dont
        l'identificateur est a*/
a=a+3; /*affecte le résultat de l'expression (a+3) à
         la variable dont l'indentificateur est a*/
}
```

L'affectation (suite)

Remarque : lorsque la déclaration et l'affectation sont combinées il est déconseillé que l'expression à évaluer soit autre chose qu'une valeur (constante).

L'opérateur adresse

L'opérateur adresse s'applique à une variable et renvoie l'adresse mémoire de celle-ci.
L'adresse d'une variable **var** s'obtient via la syntaxe :

&**var**

Remarque : Comme on le verra plus tard il est possible de faire des opérations sur les adresses.

L'opérateur de conversion de type

- L'opérateur de conversion de type permet de changer le type d'un objet ce qui signifie changer sa représentation.
- L'opérateur de conversion de type est appelé *cast*.

Lorque l'on souhaite modifier le type d'un objet en le type **type**, on écrit :

(**type**)*objet*

L'opérateur de conversion de type (suite)

```
int main()
{
float a=1.5,b=1.0;
int c;

c=(int)b; /*la valeur contenue dans la variable b est
         convertie en un int et affectée à la variable c*/

c=(int)a; /*la valeur contenue dans la variable a est
            convertie en un int (lequel?) et affectée
            à la variable c*/
}
```

Remarque : l'opérateur de conversion de type n'est pas toujours bijectif ! Attention !

Conversion de type

Lorsque qu'une des deux opérandes est un float, double ou long double, l'autre opérande est convertie automatiquement en le type le plus précis.

Conversion de type (suite)

Exemple :

```
int main()
{
int a=2,b=3;
float quotient;

quotient=(float)(a/b); /*quotient=0 car division entière
                        puis conversion en flottant*/
quotient=a/b;          /*quotient=0 car division entière
                       et conversion automatique de type
                       en flottant*/
quotient=(float)a/b);  /*quotient=0.66... car conversion
                       de  a en flottant puis
                       division flottante*/
}
```

Conversion de type : promotion intégrale

- Lorsque l'on manipule des opérateurs ou des fonctions (définis sur des int ou unsigned int) sur des char ou des short, les valeurs sont d'abord converties en des int ou unsigned int avant d'être appliquées à l'opérateur.

Exemple : char a=0xff ; a«1 ; produit un int car **a** est d'abord converti en int et puis l'opérateur shift est appliqué ce qui produit un int.

Les opérateurs arithmétiques

Les opérateurs arithmétiques sont :

- (opérateur unaire) :

  $$-\mathbf{var}$$

  modifie le signe de la valeur contenue dans la variable **var**. Le type de la valeur renvoyée peut être changé (si la variable est unsigned int par exemple).
- (opérateur binaire)

+	addition
-	soustraction
*	multiplication
/	division
%	reste de la division (modulo)

Les opérateurs arithmétiques (suite)

Remarques :

- le symbole / désigne à la fois la division entière et entre flottants. Si les deux opérandes sont de même type, elle produira un résultat de ce type. Si une des deux opérandes est flottante, le résultat est flottant.
- % ne s'applique que sur des integer's (signed et unsigned). Le signe est généralement celui du dividende mais cela peut dépendre de l'implémentation.

Les opérateurs logiques bit à bit

&	et logique
^	ou logique exclusif (XOR)
|	ou logique non-exclusif
<<	décalage à gauche
>>	décalage à droite
~	complément à 1

- Ces opérateurs s'appliquent aux entiers de toute longueur (short, int ou long) signé ou non.

Les opérateurs : opération relationnels/de comparaison

>	strictement supérieur
>=	supérieur ou égal
<	strictement inférieur
<=	inférieur ou égal
==	teste l'égalité
!=	teste la non-égalité

La syntaxe est :

expression1 operateur expression2

- Les deux expressions sont évaluées puis comparées.
- La valeur retourné est de type int, 1 si la relation est vérifiée et 0 sinon.

Les opérateurs logiques booléens

&&	et logique : vérifie que les deux conditions sont vérifiées
||	ou logique : vérifie qu'une des deux conditions est vérifiées
!	négation logique

La syntaxe est :

expression-1 operateur-1 expression-2 operateur-2 $\cdots$ operateur-n expression-n

- Les deux expression sont évaluées puis comparées.
- La valeur retourné est de type int, 1 si la relation est vérifiée et 0 sinon.
- L'évaluation se fait de gauche à droite.

Les opérateurs d'affectation composée

Pour tout opérateur **op** :

expression-1 op=expression-2

est équivalent à

expression-1= expression-1 op expression-2

- Les expressions 1 et 2 sont évaluées et puis l'opérateur **op** est appliqué. Le résulat est affecté à l'expression 1 (qui est une variable)

Les opérateurs d'affectation composée

+=	mise à jour d'une variable par addition
-=	mise à jour d'une variable par soustraction
*=	mise à jour d'une variable par multiplication
/=	mise à jour d'une variable par division
%=	mise à jour d'une variable par réduction modulo
&=	mise à jour d'une variable par ET logique
^=	mise à jour d'une variable par XOR
|=	mise à jour d'une variable par OU non-exclusif
<<=	mise à jour d'une variable par décalage à gauche
>>=	mise à jour d'une variable par décalage à droite

Les opérateurs d'incrémentation et de décrémentation

- Les opérateurs d'incrémentation (++) et décrémentation (–) permet d'ajouter ou soustraire 1 à une variable.
- Ces opérateurs peuvent s'utiliser en suffixe (i++ ou i–) ou en préfixe (++i ou –i).

Les opérateurs d'incrémentation et de décrémentation (suite)

Exemple :

```
int main()
{
int a,b;

a=2; /*affecte la valeur 2 à la variable  a*/
++a; /*incrémente de 1 la variable  a*/
a++;  /*incrémente de 1 la variable  a*/
b=++a; /*incrémente de 1 la variable  a
         et affecte la valeur obtenue à la variable  b*/
b=a++; /*affecte la valeur de  a à la variable  b et
         incrémente de 1 la variable  a */
}
```

L'opérateur virgule

- L'opérateur virgule permet de construire une expression à partir de plusieurs autres.
- La syntaxe :

  expression-1, expression-2, $\cdots$, expression-n

  constitue une expression qui est évaluée de la gauche vers la droite.

L'opérateur virgule

Exemple :

```
int main()
{
int a,b,c;

c=(a=2,a++); /*affecte la valeur 2 à la variable a
               et ensuite incrémente la variable a*/
}
```

Remarque : les arguments d'une fonction séparés par des virgules ne constituent pas une expression.

L'opérateur conditionnel ternaire

L'opérateur ? est un opérateur qui permet de retourner une expression parmi deux en fonction de la réalisation ou non d'une condition.
La syntaxe est :

condition expression-1 : expression-2

Exemple :

```
int main()
{
int a,b=2,c=3;

a=(b<=c)? b : c; /*affecte à la variable  a
                   le minimum de  b et  c */
}
```


Sixième partie VI

Les instructions de contrôle

Les intructions de contrôle

Instruction de contrôle

Une instruction de contrôle permet de contrôler le fonctionnement d'un programme. On distingue

- Les instructions de branchement : permet de déterminer quel instruction doit être exécutée en fonction de la réalisation de conditions.
- Les boucles : permet de répéter une instruction un certain nombre de fois en fonction de la réalisation de conditions.

Les intructions de contrôle (suite)

- Branchement conditionnel :
  - if/else et else/if
  - branchements multiples : switch
- Boucle :
  - while
  - do/while
  - for
- Branchement non conditionnel :
  - break
  - continue

Instruction de branchement conditionnel : if/else

Une instruction de branchement consiste à exécuter une instruction en fonction de la réalisation d'une condition.

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

Si l'expression-1 vaut 1 l'instruction-1 est exécutée.
Sinon l'instruction-2 est exécutée.

Notons que la partie *else* de l'instruction de branchement est facultative.

Instruction de branchement conditionnel : if/else (suite)

```
int main()
{
int a,b,c;

a=2;
b=3;

if (a<b)
  c=a;
else
  c=b;
}
```

Instruction de branchement conditionnel : else/if

Il est possible de combiner plusieurs instruction de branchement ce qui permet de créer un arbre de branchement.

```
if (expression-1)
  instruction-1
else if (expression-2)
  instruction-2
    ....
```

Remarque : le dernier *else* est toujours facultatif.

Instruction de branchement conditionnel : else/if (suite)

```
int main()
{
int a,signe;

a=2;
if (a==0)
  signe=0;
else if (a<0)
  signe=-1;
else
  signe=1;
}
```

Instruction de branchement conditionnel : switch

- L'instruction *switch* est une instruction de branchement multiple.
- Cette instruction est plus efficace que des else/if imbriqués mais ne fonctionne que si l'instruction que l'on souhaite exécuter ne dépend que de la valeur d'une seule variable.
- Elle s'exprime comme une succession de listes d'instructions indexées par des constantes.

Instruction de branchement conditionnel : switch (suite)

- Lorsque l'expression de l'argument du switch correspond à une des contantes, toutes les instructions correspondants aux constantes suivates sont exécutés également.
- L'usage de l'instruction *break* est donc fréquent (voir plus loin).

Instruction de branchement conditionnel : switch (suite)

La syntaxe est :

```
switch( expression )
{
   case constant-1:
      liste d'instruction-1
   case constant-2:
      liste d'instruction-2
        ...
   case constant-n:
      liste d'instruction-n
   default:
      liste d'instruction
 }
```


Inst. de branchement conditionnel : switch (suite)

Exemple :

```
int main()
{
int a=0;
switch(a)
  {
     case 0:
        printf("a=0\n");
     case 1:
        printf("a=1\n");
     default:
        printf("a>1\n");
  }
}
```

Boucle : while

Une boucle permet de répéter une série d'intructions tant qu'une condition est vérifiée.
La syntaxe est :

```
while (expression)
  instruction
```

Tant que expression est non-nulle, instruction est exécutée.

Boucle : while (suite)

Exemple :

```
int main()
{
int i=0;

while (i<=10)
   {
       printf("i=%d\n",i);
       i+=2;
   }
}
```

Boucle : do/while

Une forme alternative à la boucle while consiste à d'abord exécuter l'instruction et puis réaliser le test.

La syntaxe est :

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


Boucle : do/while

```
int main()
{
int i=10;

do
  {
      printf("i=%d\n",i);
      i+=2;
  }
while (i<10);
}
```

Boucle : for

La boucle for est un cas particulier des boucles while.
La syntaxte est :

```
for (expression1 ; expression2 ; expression 3)
 instruction
```

Cette instruction est formellement équivalente ` :

```
expression1;
while(expression2)
 {
     instruction
     expression3
 }
```

Note : les expressions peuvent être des instructions composées.

Boucle : for (suite)

Exemple :

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

```
for(i=10;i>=0;i-=2)
 {
     printf("i=%d\n",i);
 }
```

Branchement non conditionnel : continue

L'instruction continue permet de passer à la boucle suivante sans exécuter les instructions de la boucle courante succédant au continue.

Branchement non conditionnel : continue (suite)

Exemple :

```
int main()
{
int i;
for(i=0;i<10;i++)
 {
   if ((i%2)==0)
     continue;
   printf("i=%d\n",i);
 }
}
```

Affiche tous les nombres impairs de 1 à 9. Il était possible de faire ceci en jouant sur l'initialisation et l'incrément d'un for classique.

Branchement non conditionnel : break

L'instruction break permet de sortir d'une boucle (while, do/while, for) ou d'un switch.

Le programme suivant énumère tous les couples (i, j) tels que $i \leq j$. Il aurait également été possible de réaliser ceci en jouant sur les bornes des boucles.

Branchement non conditionnel : break (suite)

Exemple :

```
int main()
{
int i,j;

for(i=0;i<10;i++)
 {
    for(j=0 ; j<10 ; j++)
      printf("i=%d , j=%d \n",i,j);
      if (i<=j)
        break;
 }
}
```

Septième partie VII

Les fonctions d'entrées-sorties classiques

La fonction printf

La fonction **printf** permet l'affichage de message sur la sortie standard (affichée dans le shell).
L'usage de la fonction **printf** nécessite souvent d'utiliser la librairie strandard **stdio.h**
#include <stdio.h>
La syntaxe est :

printf("chaîne de contrôle",expression-1,·,expression-n) ;

La chaîne de contrôle contient le texte à afficher ainsi que le résultat des expressions sous un certain format.

La fonction printf

Exemple :

```
#include<stdio.h>
int main()
{
int i;

i=3;

printf("La valeur de i=%d\n",i);
}
```

Le %d dans la chaîne de contrôle sert à insérer la valeur de la variable i. La lettre d permet de spécifier le format d'impression (int dans ce cas).

La fonction printf

Format	type de l'expression	représentation
%d	int	décimale signée
%ld	long int	décimale signée
%u	unsigned int	décimale non signée
%lu	unsigned long int	décimale non signée
%o	unsigned int	octale non signée
%lo	unsigned long int	octale non signée
%x	unsigned int	hexadécimale non signée
%lx	unsigned long int	hexadécimale non signée
%f	double/float	décimale virgule fixe
%lf	long double	décimale virgule fixe
%e	double/float	mantisse/expo
%le	long double	mantisse/expo
%c	unsigned char	caractère

La fonction scanf

La fonction **scanf** permet de saisir des données au clavier et les affecter à des variables.
L'usage de la fonction **scanf** nécessite souvent d'utiliser la librairie strandard **stdio.h**
La syntaxe est :

scanf("chaîne de contrôle",argument-1,·,argument-n) ;

La fonction scanf

Exemple :

```
#include<stdio.h>
int main()
{
int i;

scanf("%d",&i);

printf("La valeur de i en octale est %o\n",i);
}
```

Le %d dans la chaîne de contrôle sert à stocker la valeur tapée dans la variable dont l'adresse est donnée en argument. Si il y a plusieurs arguments dans **scanf**, il faudra entrer les différents valeurs au clavier en tapant "entrée" entre chacune d'elles.

La fonction scanf

Format	type de l'expression	représentation
%d	int	décimale signée
%hd	short int	décimale signée
%ld	long int	décimale signée
%u	unsigned int	décimale non signée
%hu	unsigned short int	décimale non signée
%lu	unsigned long int	décimale non signée
%o	int	octale
%ho	short int	octale
%lo	long int	octale
%x	int	hexadécimale
%hx	short int	hexadécimale
%lx	long int	hexadécimale
%f	float	flottante à virgule fixe
%lf	double	flottante à virgule fixe

Huitième partie VIII

Les types composés

Les types composés

- Les types "simples" ne permettent pas d'implémenter beaucoup d'algorithmes.
- Le langage C met à disposition des types qui permettent de sauvergarder des données sous forme structurée.

Les tableaux

Un tableau permet de stocker des données de même type dont les adresses mémoire sont contiguës.

La déclaration d'un tableau de N élements dont le type est **type** :

type nom_tab[N] ;

Le tableau peut être initialisé lors de sa déclaration :

type nom_tab[N]={constante_1, ..., constante_N} ;

Les élements du tableau sont numérotés de 0 à (N-1).

Les tableaux

Exemple :

```
#include<stdio.h>
int main()
{ int i,j;  int tab_1[5]; int tab_2[5]={0,1,2,3,4};
    for(i=0;i<5;i++)
     {
          tab_1[i]=0;
          for(j=0;j<=i;j++)
            {
                 tab_1[i]+=tab_2[j];
            }
          printf("tab_1[%d]=%d\n",i,tab_1[i]);
    }
}
```


Les tableaux (suite)

Remarque :

- Si l'on souhaite recopier un tableau dans un autre, il faut le faire élément par élément.
- On ne peut pas écrire tab_1=tab_2.

Les tableaux

- Lorsque l'on initialise un tableau lors de sa déclaration, il n'est pas nécessaire de mentionner sa taille. Exemple : char tab[]="bonjour" ;
- Il est également possible de déclarer des tableaux à N dimensions. Il s'agit simplement d'un tableau dont les éléments sont des tableaux à N-1 dimensions. La syntaxe pour N=2 est :

  type tab[nbre_lignes][nbre_colonnes]

  L'accès à un élément se fait via tab[i][j]

Les tableaux multi-dimensionnels

Exemple :

```
#include<stdio.h>
int main()
{
int tab_1[2][3];  int tab_2[2][3]={{0,1,2},{3,4,5}}; int i,j;
for(i=0;i<2;i++)
  {
      for(j=0;j<3;j++)
       {
           tab_1[i][j]=tab_2[i][j];
           printf("tab_2[%d][%d]=%d\n",i,j,tab_2[i][j]);
       }
  }
}
```

Les structures

- Les structures permettent de définir des objets composés d'objets de types différents.
- Chaque élément d'une structure est appelé champ.
- Avant de déclarer (et utiliser) un objet qui est une structure, il faut déclarer la composition de la structure qui correspond à un nouveau type :

```
struct framework
 {
    type_1 champ_1;
    ...
    type_n champ_n;
 };
```


Les structures (suite)

Afin de déclarer un objet correspondant à la structure précédente, il faut déclarer l'objet de la façon suivante :

```
struct framework objet;
```

Les deux étapes précédents peuvent se faire en une seule :

```
struct framework
 {
    type_1 champ_1;
    ...
    type_n champ_n;
 } objet;
```

Il est possible ensuite de déclarer d'autres variables de type *struct framework*.

Les structures (suites)

Remarques :

- On peut recopier un struct dans un autre via une simple affectation (pas besoin de descendre au niveau des champs).
- Il est possible d'initialiser un struct lors de sa déclaration. Exemple : struct framework objet={val_1,...,val_n} ;
- On désigne un champ en ajoutant un point suivi du nom de l'objet. Exemple : objet.champ_1. Ces expressions s'utilisent comme des variables classiques.

Les champs de bits

- Il est possible de découper un mot en plusieurs groupements de bits, cela se fait via une structure en spécifiant la taille des champs.
- La correspondance entre les bits du mot et les différents champs dépend de l'implémentation.
- Ces champs ne possèderont pas d'adresse, il ne sera donc pas possible d'utiliser l'opérateur & sur ceux-ci (mais bien sur la structure).

Les champs de bits

Exemple :

```
struct framework
 {
    unsigned int gauche : 8;
    unsigned int droite: 24;
 } objet;
```

objet.gauche contiendra 8 bits alors que objet.droite en contiendra 24.

Les unions

Les unions permettent d'attribuer plusieurs types une même case mémoire. Sa syntaxe est identique à celle de la structure.

```
union framework
 {
    type_1 champ_1;
    ...
};
```

Afin de déclarer un objet correspondant à l'union précédente, il faut déclarer l'objet de la façon suivante :

```
union framework objet;
```

Les unions

Les deux étapes précédents peuvent se faire en une seule :

```
union framework
 {
    type_1 champ_1;
    ...
    type_n champ_n;
 } objet;
```


Les unions

Exemple :

```
#include<stdio.h>
int main()
{
union framework
 {
    int a;
    float b;
 }objet;
objet.a=1;
printf("objet=%d\n",objet.a);
objet.b=1.5;
printf("objet=%f\n",objet.b);
}
```

Les énumérations

Une énumération permet de créer un type en spécifiant les différentes valeurs qu'il peut prendre.
Déclaration d'une énumération :

```
enum framework {constante_1, ..., constante_n};
```

Afin de déclarer un objet correspondant à l'énumération précédente, il faut déclarer l'objet de la façon suivante :

```
enum framework objet;
```


Les énumérations

Nativement, il n'existe pas de type booléen en C. Créons-le.
Exemple :

```
#include<stdio.h>
int main()
{
enum booleen {vrai, faux};
enum booleen variable;
variable=vrai;
variable=faux;
if (variable==vrai)
   printf("vrai\n");
if (variable==faux)
   printf("faux\n");
}
```

Les énumérations (suite)

Remarque :

- Le compilateur va associer une valeur à chaque donnée définissant le type. Il est possible de définir cette correspondance.

Exemple : enum booleen {vrai=1, faux=0} ;

Renommage de type

Les écritures suivantes sont relativement lourdes :

- enum booleen variable
- struct framework objet

Il est possible de donner un nom à "enum booleen" et "struct framework" afin d'alléger les notations.

typedef type synonyme ;

Renommage de type

Exemple :

```
#include<stdio.h>
int main()
{
enum booleen {vrai, faux};
typedef enum booleen booleen;
booleen variable;
variable=vrai;
variable=faux;
if (variable==vrai)
   printf("vrai\n");
if (variable==faux)
   printf("faux\n");
}
```


Neuvième partie IX

Les pointeurs

Nom d'une variable et adresse mémoire

- Une variable possède un nom et réfère à un espace mémoire de la mémoire centrale.
- Cette case mémoire possède une adresse.
- Une variable se manipule soit via nom nom, soit via son adresse mémoire.

Pointeur

- Un pointeur est une variable dont la valeur est l'adresse (un entier) d'une case mémoire qui possède un type.
- On accède à la valeur d'un pointeur via l'opérateur *.
- Par défaut, un pointeur est initialisé à la valeur NULL (qui vaut souvent 0) et défini dans <stdio.h>.

On déclare un pointeur via la syntaxe :

```
type *nom_du_pointeur;
```

Pointeur (suite)

Exemple :

```
#include<stdio.h>
int main()
{
    int i=3;
    int *pt;

    pt=&i;
    *pt+=1;

   printf("i=%d\n",i);
}
```


Arithmétique des pointeurs

Le type vers lequel un pointeur pointe permet de déterminer la taille de la case vers laquelle il pointe.

- Si on additionne (resp. soustrait) un entier i à un pointeur pointant vers un objet dont le type est **type**, la valeur du pointeur résultant est un entier dont la valeur est la somme (différence) de celui-ci et de sizeof(type)*i.
- Si on soustrait deux pointeurs p et q de même type, on obtient un pointeur dont la valeur est (p-q)/sizeof(type).

Arithmétique des pointeurs

Exemple :

```
#include<stdio.h>
int main()
{
    int tab[]=1,2,3;
    int *pt;

    pt=&tab[0];
    pt++;
    printf("tab[1]=%d\n",*pt);
}
```


Allocation de mémoire et pointeurs

- Il arrive fréquemment que l'on ne connaisse pas la taille des tableaux à utiliser avant l'exécution d'un programme.
- Le C met à disposer des fonctions qui permettent d'allouer et de libérer de la mémoire dynamiquement c-à-d en cours d'exécution du programme. Ces fonctions se trouvent dans <stdlib.h>.
- Ces zones mémoires que l'on peut allouer sont référencées par des pointeurs.

Allocation de mémoire et pointeurs (suite)

La fonction malloc permet d'allouer de la mémoire :

malloc(nombre_octets)

- La fonction malloc retourne un pointeur de type *char correspondant à l'adresse du premier octet de la zone allouée.
- Il faudra utiliser un cast pour convertir ce pointeur en un pointeur d'un autre type.

Allocation de mémoire et pointeurs (suite)

La fonction calloc permet d'allouer de la mémoire et d'initialiser les octets de la zone allouée à zéro :

calloc(nombre_octets)

La fonction free permet de libérer la mémoire pointée par un pointeur.

Allocation de mémoire et pointeurs(suite)

Exemple :

```
#include<stdio.h>
int main()
{
    int *pt;
    int n=4,i;

    pt=(int*)malloc(n*sizeof(int));
    for(i=0;i<n;i++)
     {
       *(pt+i)=i;
        printf("*(pt+%d)=%d\n",i,*(pt+i));
     }
    free(pt);
}
```


Allocation de mémoire et pointeurs(suite)

Remarques :

- on ne peut libérer la mémoire qu'en mentionnant le début de la zone. Exemple : free(pt+1) ; n'est pas valide.

Pointeurs et tableaux

- Un tableau est un pointeur constant, son adresse ne peut donc être modifiée (ex. tab++, tab1=tab2).
- La taille d'un tableau ne peut dépendre d'une variable du programme (ATTENTION À PRÉCISER).
- Un tableau bidimenstionnel est nécessairement rectangulaire (toutes les lignes ont même longueur).
- Au lieu d'écrire *(pt+i) on peut également utiliser l'instruction pt[i] et inversément (pour les pointeurs et les tableaux).

Pointeurs et tableaux à plusieurs dimensions

- Un tableau à n dimensions est un tableau à une dimension dont les éléments sont des tableaux à (n-1) dimensions.
- En terme de pointeur, un tableau à deux dimensions est un pointeur dont les éléments sont des pointeurs. Un pointeur de pointeur est déclaré via l'instruction :

  type **pt ;
- Lorsque l'on souhaite construire un tableau à deux dimensions via des pointeurs de pointeurs, il faut d'abord allouer l'espace mémoire contenant les pointeurs et ensuite, pour chacun de ces pointeurs, l'espace pour chaque "ligne" du "tableau". La désallocation nécessite de faire les étapes dans l'ordre inverse.

Pointeurs et tableaux à plusieurs dimensions (suite)

Exemple :

```
#include<stdio.h>
int main()
{
    int **pt;
    int n=4,i;
    pt=(int**)malloc(n*sizeof(int*));
    for(i=0;i<n;i++)
      {
       pt[i]=(int*)malloc(2*n*sizeof(int));
          ...
      }
    for(i=0;i<n;i++)
       free(pt[i]);
    free(pt);
}
```


Pointeurs et fonctions

- Les passages des valeurs dans une fonction se font par valeur.
- Lorsque l'on déclare une fonction int fct(int a, int b) cette fonction s'attend à recevoir deux int en argument.
- Lorsque l'on appelle une fonction fct(3,5), cela signifie que la valeur 3 et 5 sont transmise à la fonction. Cela signifie que deux autres variables sont crées lors de l'appel, placées sur la pile et les valeurs 3 et 5 leurs sont affectés. Lors de la sortie de la fonction ces variables sont détruites.
- Si on souhaite que la fonction modifie une valeur extérieure à celle-ci, il faut fournir l'adresse de cette variable, c'est-à-dire un pointeur (constant ou pas).

Pointeurs et fonctions : passage par valeur

Exemple :

```
#include<stdio.h>
int fonction(int a)
{
  return a+1;
}
int main()
{
    int i=5;
    i=fonction(i);
    printf("i vaut %d\n",i);
}
```

i vaut 6.

Pointeurs et fonctions : transmission de l'adresse

Exemple :

```
#include<stdio.h>
int fonction(int *a)
{
  (*a)++;
}
int main()
{
    int i=5;
    fonction(&i);
    printf("i vaut %d\n",i);
}
```

i vaut 6.

Pointeurs et chaînes de caractères

Exemple :

```
#include<stdio.h>
int main()
{
char *chaine;
int i=0;
chaine="Hello";
while(chaine[i]!='\0')
 {
      printf("caractère=%c\n",chaine[i]);
      i++;
 }
}
```


Pointeur vers une structure

- Il arrive souvent qu'un pointeur pointe vers une structure. Il existe alors un formalisme particulier pour accéder aux champs de celle-ci. Si pt est un pointeur vers une structure dont un champ est **a**, on écrit (*pt).a pour désigner le champ a de la structure. La notation pt->a est synonyme.

Pointeur vers une structure (suite)

Exemple :

```
#include<stdio.h>
struct framework
{
int a;
int b;
}
typedef struct framework framework;
int main()
{
framework *pt;
pt=(framework*)malloc(sizeof(framework));
(*pt).a=1;
(*pt).b=2;
printf("le champ a vaut %d et b vaut %d\n",pt->a,pt->b);
}
```


Pointeurs et structures : liste chaînée

- Un tableau possède taille qui doit être fixée indépendamment d'une variable du programme, sa taille doit donc être déterminée en fonction d'une constante.
- Un pointeur vers un espace mémoire alloué dynamiquement permet de créer un tableau dont la taille peut dépendre d'une variable du programme.
- La manipulation d'un tableau ou d'un espace alloué dynamiquement peut cependant poser quelques problèmes, notamment si l'on veut insérer des éléments au début.

Pointeurs et structures : liste chaînée

- Les listes chaînées consistent à chaîner des structures les unes aux autres.
- On commence par déclarer un framework de structure :

```
struct cellule
{
     int donnee;
     struct cellule *pt_suivant;
}
```

- Une liste chaînée est simplement un pointeur vers une structure cellule (qui elle-même peut pointer vers une structure cellule etc).
- Il peut être commode de définir un nouveau type :

```
typedef struct cellule *liste;
```


Pointeurs et structures : arbre binaire

- Un arbre binaire fonctionne sur le même principe si ce n'est que chaque cellule pointe vers deux cellules.
- On commence par déclarer un framework de structure :

```
struct noeud
{
     int donnee;
     struct cellule *pt_gauche;
     struct cellule *pt_droite;
}
```

- Un arbre binaire est simplement un pointeur vers une structure noeud (qui elle-même peut pointer vers deux structures noeud etc).
- Il peut être commode de définir un nouveau type :

```
typedef struct noeud *arbre;
```

Dixième partie X

La gestions des fichiers

Introduction

- La librairie <stdio.h> met à disposition des fonctions qui permet de manipuler des fichiers (ouverture, fermeture, lecture, écriture).
- Lorsque l'on manipule un fichier depuis un programme C, on le manipule toujours par l'intermédiaire d'une mémoire tampon pour des raisons d'efficacité (même principe que le cache pour la manipulation d'objets en mémoire centrale).
- L'adresse de la mémoire tampon ainsi que d'autres informations (mode d'accès etc) se trouve dans une structure dont le type est FILE.

Ouverture d'un fichier : fopen

- Pour ouvrir un fichier, on utilise la fonction **fopen** en lui spécifiant le fichier à ouvrir ainsi que le mode d'accès (écriture, lecture etc).
- La fonction renvoie un pointeur vers un objet de type FILE qui spéficie le fichier en question. Si l'ouverture n'est pas possible le pointeur renvoyé est NULL.
- Un pointeur de type *FILE aura dû être déclarer avant l'utilisation de la fonction afin de sauvegarder le pointeur renvoyé par fopen.
- La syntaxe est :

fopen("nom_du_fichier","mode")

Ouverture d'un fichier : fopen

Les modes pour un fichier texte sont :

Symbole	Mode
"r"	ouverture d'un fichier texte en lecture. Erreur si inexistant. Stream au début.
"r+"	ouverture d'un fichier texte en lecture/écriture. Erreur si inexistant. Stream au début.
"w"	ouverture d'un fichier texte en écriture. Création dans le répertoire courant si inexistant ou écrasement du fichier. Stream au début.
"w+"	ouverture d'un fichier texte en lecture/écriture. Création si inexistant ou écrasement du fichier. Stream au début.
"a"	ouverture d'un fichier texte en écriture. Création dans le répertoire courant si inexistant. Stream à la fin.
"a+"	ouverture d'un fichier texte en lecture/écriture. Stream au à la fin

Ouverture d'un fichier : fopen

Exemple :

small

```
#include<stdio.h>
int main()
{
FILE *fp;

fp=fopen(".\fichier.txt","r");
}
```


Ouverture d'un fichier : fclose

- La fonction fclose permet de fermer un fichier, ce qui est indispensable afin de vider le tampon-mémoire sur le disque (même principe que pour retirer une clé USB).
- La fonction fclose prend en argument le pointeur de type *FILE renvoyé par fopen.
- La syntaxte est :

fclose(pointeur_fichier)

Fermeture d'un fichier : fclose

Exemple :

small

```
#include<stdio.h>
int main()
{
FILE *fp;

fp=fopen(".\fichier.txt","r");
fclose(fp);
}
```


Lecture d'un fichier : fscanf

- La fonction fscanf permet de lire des données dans un fichier.
- La fonction prend en argument le pointeur sur le fichier de type *FILE, une chaîne de contrôle permettant de savoir comment il faut interpréter les données lues et de addresses de variable permettant de sauvegarder les données lues.
- La fonction renvoie le nombre d'éléments lus et -1 si on est à la fin du fichier. Notons que -1 est contenu dans la constante EOF.
- La syntaxe est :

fscanf(pointeur_fichier,"chaîne de contrôle", adresse_1, ..., adresse_n)

Lecture d'un fichier : fscanf

On suppose que le fichier fichier.txt contient un nombre pair d'entiers.
Exemple :

```
#include<stdio.h>
int main()
{
FILE *fp;
int i,j;
fp=fopen(".\fichier.txt","r");
while (fscanf(fp,"%d%d",&i,&j)!=EOF)
 {
     printf("i=%d  et j=%d \n",i,j);
 }
fclose(fp);
}
```


Lecture d'un fichier : fscanf

On suppose que le fichier fichier.txt contient 4 chaînes de caractères de 10 caractères chacunes maximum.
Exemple :

```
#include<stdio.h>
int main()
{
FILE *fp;
char str1[10],str2[10],str3[10],str4[10];
int i,j;
fp=fopen(".\fichier.txt","r");
while (fscanf(fp,"%s%s%s%s",str1,str2,str3,str4)!=EOF)
 {
     printf("%s  %s  %s  %s\n",str1,str2,str3,str4);
 }
fclose(fp);
}
```

Ecriture dans un fichier : fprintf

- La fonction fprintf permet d'écrire des données dans un fichier.
- La fonction prend en argument le pointeur sur le fichier de type *FILE, une chaîne de contrôle permettant de savoir comment il faut interpréter les données à écrire et les variables contenant les données à écrire.
- La fonction renvoie le nombre d'éléments écrits dans le fichier et un nombre négatif en cas d'erreur.
- La syntaxe est :

fprintf(pointeur_fichier,"chaîne de contrôle", variable_1, ..., variable_n)

Ecriture dans un fichier : fprintf

Exemple :

```
#include<stdio.h>
int main()
{
FILE *fp;
int i;

fp=fopen(".\fichier.txt","w");
for(i=0;i<10;i++)
 {
   fprintf(fp,"i=%d\n",i);
 }

fclose(fp);
}
```

Onzième partie XI

Le préprocesseur

La directive **#define**

- Il peut être pratique de définir en début de programme un morceau de code, que l'on appelle **macro** que l'on souhaite pourvoir intégrer de multiples fois dans le code.
- Ce morceau de code peut être aussi simple qu'une valeur constante.
- Une macro se définit avant la fonction main en utilisant la syntaxe (par convention le nom d'une macro est écrit en majuscules) :

```
#define NOM_MACRO Code_A_Substituer
```

La directive **#define** (suite)

Exemple :

small

```
#include<stdio.h>
#define VAL 10
int main()
{
int i;

i+=VAL;

printf("La valeur de i en octale est %o\n",i);
}
```


La directive **#define** : macros imbriquées

- Il est possible d'exprimer une macro en fonction d'autres macros.

Exemple :

```
#include<stdio.h>
#define HAUTEUR 10
#define LONGUEUR 20
#define SURFACE HAUTEUR*LONGUEUR
int main()
{

printf("La surface vaut=%d\n",SURFACE);
}
```

La directive **#define** : morceau de code

- Il est définir une macro pour un morceau de code.

Exemple :

```
#include<stdio.h>
#define HELLO printf("Hello World\n")
int main()
{
HELLO;
}
```


La directive **#define** : morceau de code

- Il est possible de définir une macro avec un (ou plusieurs) paramètre(s). La ou les variables doivent apparaître avec des parenthèses dans la définition de la macro.

Exemple :

```
#include<stdio.h>
#define HELLO(x,y) printf("Hello World%d et %d\n",(x),(y))
int main()
{
HELLO(3,4);
}
```