

# Cours : Programmation en C++ avec Linux

Masters: Sciences et Techniques Nucléaires  
&&  
Physique de la Matière Condensée

Prof. Mohamed Gouighri

Année Universitaire 2019/2020

# Plan

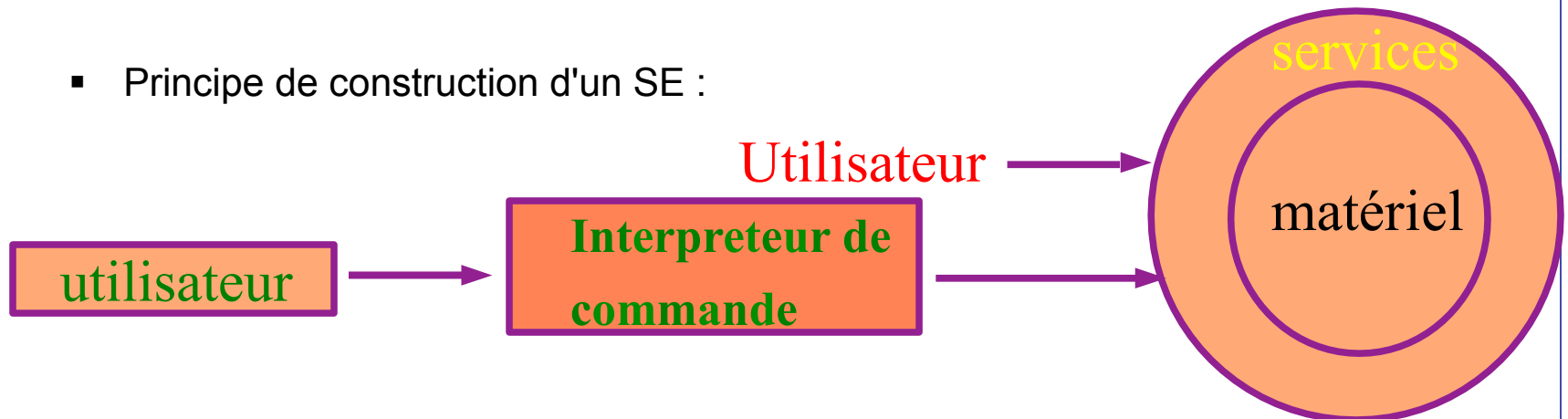
- **Introduction générale**
- **Système d'exploitation (Linux) && commandes shell**
- **Bref introduction en programmation C**
- **La programmation en C++**
- **Les fonctions en C++**
- **Les énoncés conditionnels, combinaison logique d'expressions booléennes**
- **Les itérations en C++**
- **Tableaux et Pointeurs**
- **Allocation statique de la mémoire**
- **La programmation Orientée Objet (POO)**
- ...

## Composants d'un ordinateur

- **Matériel** : Processeur, Disque dur, les périphériques, la mémoire
- **Système d'exploitation** : MS-DOS, Windows, Unix, Linux, , ...
- **Les applications** : Programmes, jeux, les utilitaires, ...
- **Les Utilisateurs**

## Introduction

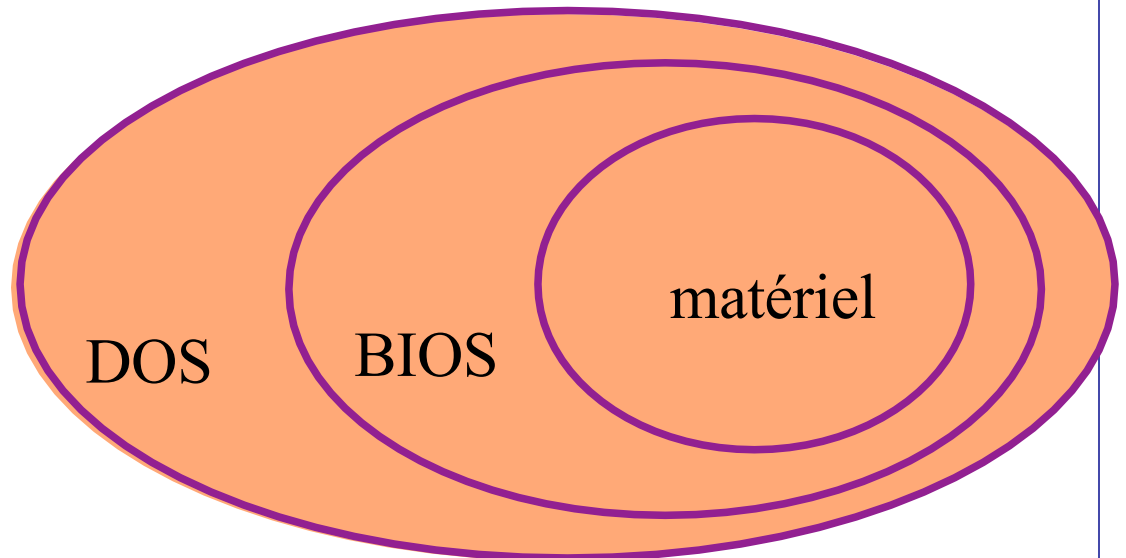
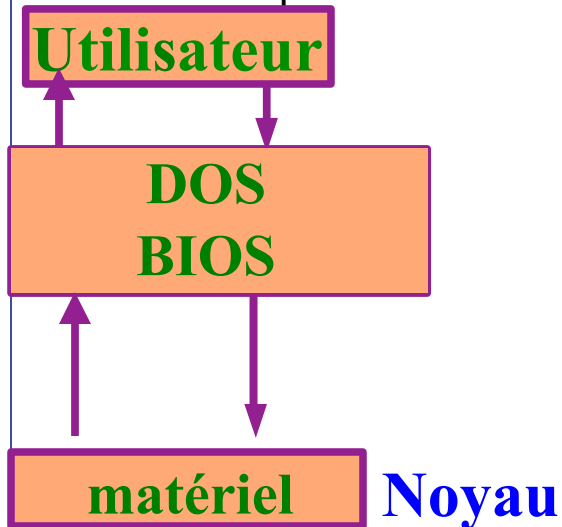
- Un système d'exploitation (SE ou OS) est un logiciel destiné à faciliter l'utilisation d'un ordinateur, il assure l'interface entre le matériel et l'utilisateur.
- La complexité d'un SE dépend :
  - Système mono-utilisateur
  - système mono-utilisateur mono-tâche : MS-DOS
  - système mono-utilisateur multitâche
- Principe de construction d'un SE :



# Introduction Générale

## Cas de MS-DOS:

- Un des premiers systèmes d'exploitation, il est mono-utilisateur et mono-tâche.
- La partie service se décompose en deux parties : le Dos et le BIOS.
- Le DOS (Disque Operating System) fait appel aux services du BIOS (Basic Input Output System) qui sont dépendants du matériel, le DOS en est indépendant.
- L'interpréteur de commande : command.com



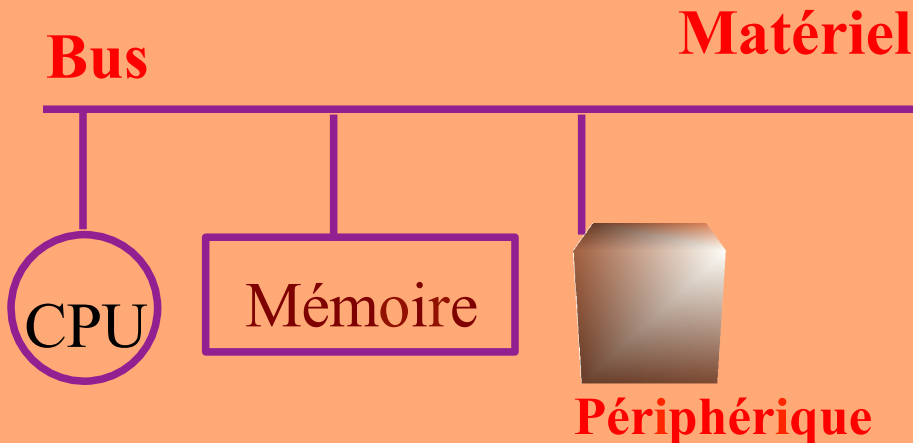
# Systeme d'exploitation

Utilisateurs



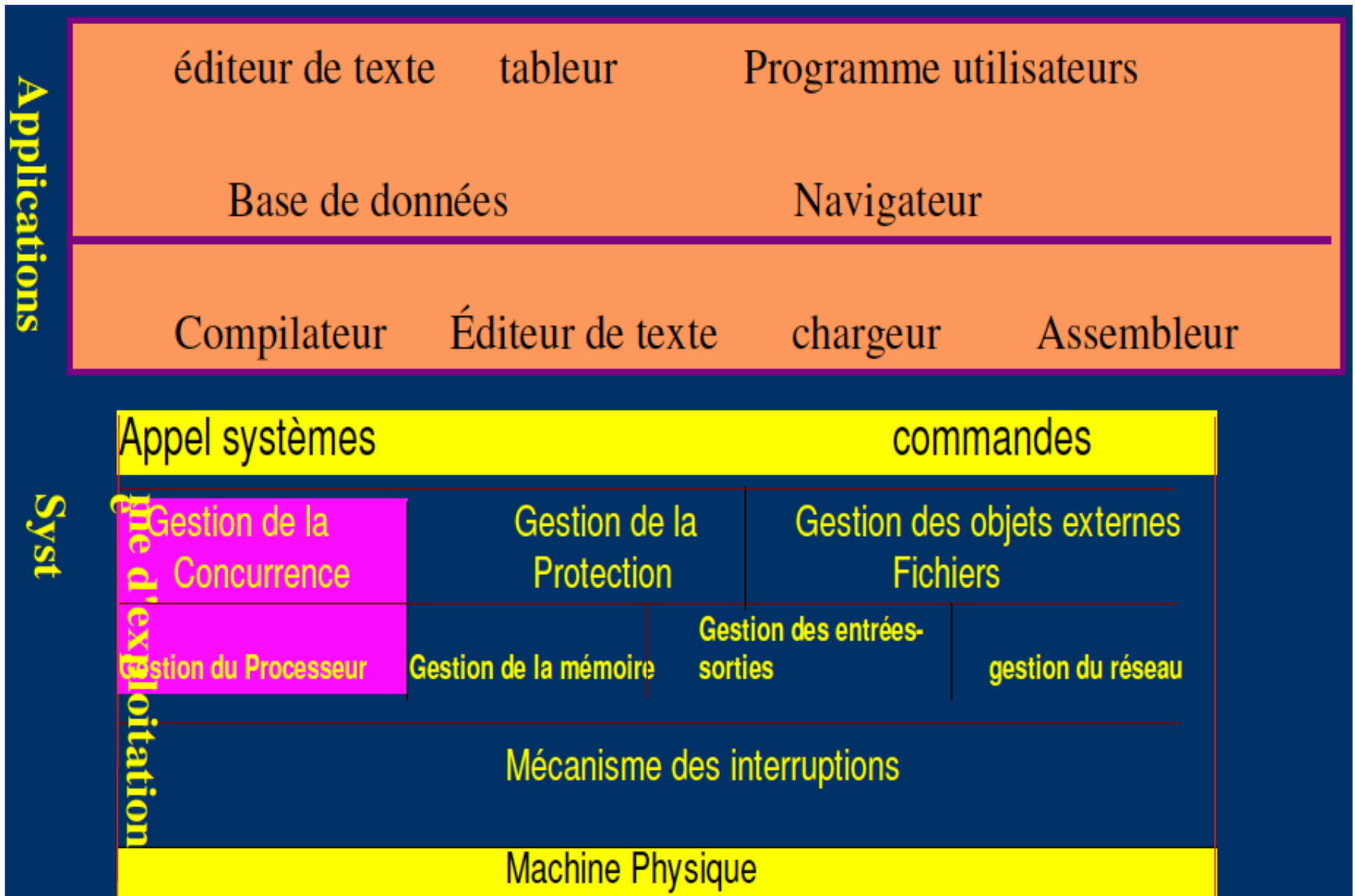
Machine Virtuelle

systeme d'exploitation



- Prise en charge et partage des ressources
  - Processeur, mémoire centrale et les périphériques
- Construction au-dessus du matériel d'une machine virtuelle plus facile d'emploi et plus conviviale
- Le SE réalise une couche logicielle placée entre la machine matérielle et les applications
- **allocation du processeur** aux différents processus par l'algorithme d'ordonnancement
- **La mémoire centrale** ; mémoire volatile, toutes les données doivent être stockés sur une mémoire de masse non volatile (disque dur, disquette, cédérom) **systeme de gestion de fichiers**

# **Système d'exploitation**



# Systeme d'exploitation

- **Le langage de commandes** : permet à un utilisateur de communiquer avec un ordinateur pour créer des répertoires, des fichiers, déclencher l'exécution d'un programme. L'interpréteur de commandes (command.com) lit les commandes au clavier ou dans un fichier de commandes et déclenche les services du noyau correspondant.
  - Exemple : type lettre.txt
  - dir
  - copy a:fichier1.txt c:fichier2.txt
- Le système de fichiers a une structure hiérarchique de répertoires et de fichiers.



# Systeme d'exploitation

- **Systeme multiutilisateur :**
- Accès aux données : un utilisateur doit indiquer ce qui est donnée privée et ce qui est donnée publique.
- La mise en oeuvre d'un système d'exploitation se fait à l'aide d'un langage de commandes, dont la syntaxe varie d'un système à un autre
  - Ordonnancement des accès
  - Partage équitable des ressources que constituent le processeur, la mémoire centrale et les périphériques

# Historique, présentation générale d'Unix

- Unix est né d'un échec : celui du développement d'un super-système d'exploitation appelé Multics. Le développement du langage C a permis de s'affranchir de l'écriture en langage d'assemblage.
- Les raisons du succès d'Unix sont :
  - Le SE est écrit dans un langage de haut niveau(plus lent, mais portable)
  - Les appels systèmes sont réutilisables pour l'écriture des commandes
  - Le système de Gestion des Fichiers est hiérarchique
  - Le SE est multi-utilisateur et multi-tâche
- Le langage de commandes : Il ne fait pas partie du noyau, il correspond à la couche la plus externe d'où son nom de shell. Les principaux shells sont :
  - le Bourne shell                      /bin/sh
  - le C-shell                              /bin/csh
  - le tcsh                                 /bin/tcsh
  - le Korn shell                         /bin/ksh
  - le bash                                 /bin/bash

# Commandes sous Unix

- Le shell est un programme qui gère l'interface utilisateur-noyau du système. Lors du login, l'utilisateur est connecté avec un répertoire et un shell par défaut.
- Syntaxe : **Nom de commande options paramètres**
- On peut écrire plusieurs commandes sur la même ligne séparées par ; ou s'étendant sur plusieurs lignes à l'aide de \.
- **Système de fichiers arborescents** : C' est un système arborescent constitué de répertoires et de fichiers. A la racine de l'arbre, on trouve le répertoire de nom /. Les différents disques logiques (partitions de disques physiques) constituent chacun un système de fichiers.
  - date
  - date + '%d/%m/%y %Hh%M'
  - cal

# Programmation shell

- **Les commandes externes** : Une commande externe est fichier localisé dans l'arborescence.
  - Exemple ls : /usr/bin/ls
- Sont considérés comme commandes externes les fichiers possédant l'un des formats suivants :
  - Fichiers au format binaire exécutable ;
  - Fichiers au format texte représentant un script de commandes (écrit en shell ou dans un langage comme Perl)
- **Les commandes internes** : Une commande interne est intégrée au processus shell, elle ne correspond pas un fichier sur le disque (cd, pwd, ...)

# Programmation shell

- Ecrire un script à l'aide d'un éditeur de texte : (nedit, emacs, vi, ....)
  - Fichier texte script.sh
  - `chmod a+x script.sh`

```
#!/usr/bin/sh
date
echo "Catalogue accueil =" $HOME
echo "Catalogue de travail = \C" ; pwd
echo "Nombre de fichiers = " ; ls | wc -l
```

Exécution de script.sh : `sh script.sh`

# Programming shell

```
Terminal
File Edit View Terminal Tabs Help
fpk02*14-->nedit script.sh &
[1] 6417
fpk02*15-->nedit: the current locale is utf8 (en_US.UTF-8)
nedit: changed locale to non-utf8 (en_US)

fpk02*15-->sh script.sh
Tue May 22 16:53:46 WET 2007
Catalogue accueil = /home/hoummada
Catalogue de travail = \C
/home/hoummada
Nombre de fichiers =
5
fpk02*16-->ls
Desktop hoummada hoummada.tar.gz latex2html-2002.tar.gz script.sh
fpk02*17-->□
```

```
script.sh - /home/hoummada/
File Edit Search Preferences Shell Macro Windows Help
#!/usr/bin/sh
date
echo "Catalogue accueil =" $HOME
echo "Catalogue de travail = \C" ; pwd
echo "Nombre de fichiers = " ; ls | wc -l
```

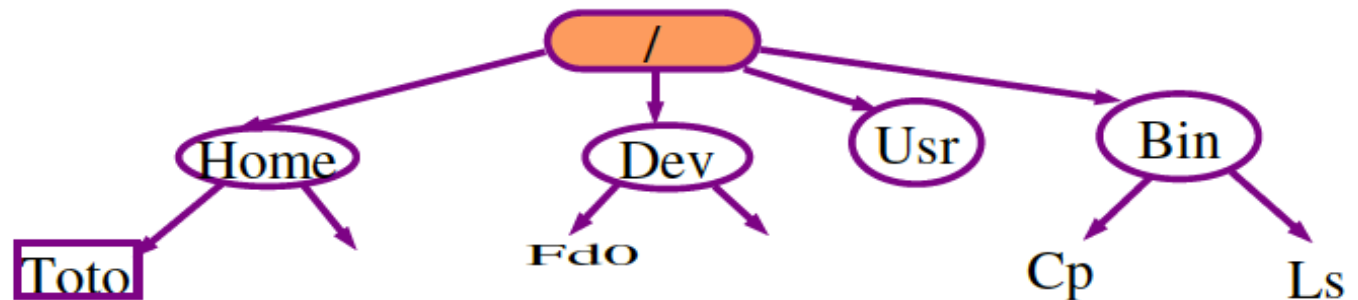
# Introduction Unix/Linux

- Linux est un système d'exploitation totalement gratuit, il est la propriété de Linus Torvalds (1991) ainsi que des autres contributeurs, il fait partie du “logiciel libre”. Linux fonctionne sur des machines 386/486/Pentium possédant un bus ISA, EISA, ou PCI,....
  
- **Caractéristiques de Linux :**
  - Multi-tâches: exécute plusieurs programmes en pseudo-parallélisme.
  - Multiutilisateurs: Plusieurs utilisateurs actifs sur la même machine en même temps et sans licence multiutilisateurs (**droits d'accès**).
  - Mémoire virtuelle utilisant la pagination sur disque (swap)
  - Consoles virtuelles multiples : Plusieurs sessions indépendantes accessibles par une combinaison de touches
  - Client serveur et Lan Manager
  - X Window System (X11R6) sous la forme Xfree86, gérant l'essentiel des cartes vidéo et des souris disponibles sur PC

# Configuration minimale

- Distribution : RedHat7.x... différence entre les noyaux (kernel) et les utilitaires qui les accompagnent.
- Disque dur : minimum 1 Go
- Ram (Random Access Memory) : 64 Mo
- Carte vidéo : 16 Mo

## Arborescence sous UNIX :



- Structure des répertoires de Linux : /bin, /boot, /dev, /etc, /home, /lib, /opt, /mnt, /proc, /root, /sbin, /tmp, /usr, /var

/ : root of the tree, racine de l'arborescence



# Les répertoires principaux de Linux

- / : Ce répertoire contient les sous répertoires il peut contenir aussi le noyau de Linux qui porte le nom de vmlinuz ou vmlinux
- /boot : Au démarrage du système, le programme d'amorçage examinera le répertoire boot (LILO, GRUB, ...)
- /bin : Contient les commandes les plus importantes, les moins importantes sont dans /usr/bin
- /dev : il contient les fichiers des périphériques (devices) par lesquels on communique avec les appareils raccordés à l'ordinateur.
- /etc : Fichiers de configuration et une série de commandes d'administration (hosts, passwd, group,...). Contient aussi /etc/rc.d qui contient des scripts de shell traités au démarrage de l'ordinateur. /etc/X11 contient les fichiers de configuration de l'interface graphique. Les principaux fichiers de /etc
- csh.login : fichier de démarrage du C shell. Les instructions qu'il contient sont exécutées au démarrage.
- fstab : liste de tous les systèmes de fichier, y compris leur point de montage (répertoire de démarrage), leur type et d'autres informations.
- inittab : liste des étapes à parcourir au lancement du système.

# Les répertoires principaux de Linux

- /home : Sous linux, le répertoire personnel des utilisateurs figurera le plus souvent sous le répertoire /home, permet de définir les privilèges d'accès, le quota, ...
- /lib : Contient les bibliothèques communes ou partagées, disponibles au démarrage. Les bibliothèques contiennent des fonctions standard nécessaires aux programmes qui correspondent aux DLL (Dynamic Link Libraries) de windows;
- /opt : Contient certains programmes complémentaires
- /proc : Pseudo-système de fichiers, contient les fichiers d'état du système, ces fichiers n'occupe aucune place sur le disque dur. Constructions logiques qui pointent vers des programmes en mémoire vive qui lisent directement des informations système.
- /root : Le répertoire de l'administrateur
- /sbin : Contient des fichiers de configuration et de démarrage du système, en plus de
- /vmlinuz et /etc. On peut distinguer trois types de commandes :
  - les commandes système générales : init, swapon, swapoff, mkswap, getty, etc
  - les commandes de démarrage et d'arrêt du système : shutdown, fastboot, fasthalt et reboot

# Les répertoires principaux de Linux

- les commandes gérant l'espace du disque dur : Au lancement du système, les systèmes de fichiers sont, entre autres, vérifiés avant d'être montés. Les commandes nécessaires s'appellent : fsck, e2fsck, mkfs, mke2fs, et fdisk.
- /tmp : Espace temporaire de stockage
- /var : Emplacement des données variables : accès en lecture et écriture (/var/www et /var/spool/lpd) par contre /usr accès en lecture uniquement.
- /usr : données sensibles : Le répertoire /usr contient une série de répertoires dans lesquels linux stocke des données très importantes.
- /usr/X11R6 : répertoire racine de toutes les données sur X Window
- /usr/bin : utilitaires, employés moins souvent (/usr/sbin)
- /usr/doc : fichiers de documentation linux
- /usr/games : jeux , très important !!
- /usr/include : Lors de la programmation et de la configuration du noyau du système d'exploitation, les fichiers d'en tête sont lus par le compilateur dans ce répertoire. Ils contiennent des constantes et des définitions de macros importantes. - /usr/lib : Bibliothèques
- /usr/local : fichiers spécifiques à l'ordinateur
- /usr/src : le texte source du noyau du système d'exploitation et éventuellement d'autres packages.

# Partition du disque

- Dans un premier temps avant d'installer linux on procède à un formatage du disque, on crée au moins une partition Linux et une partition swap (le swap est un système de mémoire virtuelle).
- Disques sous Linux sont des devices.
  - /dev/hdax premier x = 1, 2, ....
  - /dev/hdbx second Contrôleur IDE
  - /dev/sdax pour les contrôleurs SCSI
- Partition manuelle à l'aide de fdisk ou Druid
- LILO : permet d'avoir un système d'amorçage multiple : MSDOS, linux, ... premier secteur du disque Master Boot Record (MBR)
- Le fichier **/etc/fstab** contient toutes les informations concernant le montage des partitions

<b>LABEL=</b>	<b>/</b>	<b>ext3</b>	<b>defaults</b>	<b>1 1</b>
<b>none</b>	<b>/dev/pts</b>	<b>devpts</b>	<b>gid=5,mode=620</b>	<b>0 0</b>
<b>none</b>	<b>/proc</b>	<b>proc</b>	<b>defaults</b>	<b>0 0</b>
<b>none</b>	<b>/dev/shm</b>	<b>tmpfs</b>	<b>defaults</b>	<b>0 0</b>
<b>/dev/hda5</b>	<b>swap</b>	<b>swap</b>	<b>defaults</b>	<b>0 0</b>
<b>/dev/cdrom</b>	<b>/mnt/cdrom</b>	<b>udf,iso9660</b>	<b>noauto,owner,kudzu,ro</b>	<b>0 0</b>
<b>/dev/fd0</b>	<b>/mnt/floppy</b>	<b>auto</b>	<b>noauto,owner,kudzu</b>	<b>0 0</b>

# Partition du disque

## ▪ Description :

- Device (périphérique) de la partition 2 – Point de montage
- Type de partition
- options (lecture, écriture, ...)
- fréquence correspond au nombre de jours entre deux traitements du fichier.  
6 – ordre de tests des partitions (fsck), 0 aucune vérification automatique au démarrage

## ▪ Montage manuel des partitions :

- Disquette : `mount -t msdos /dev/fd0 /mnt/floppy` -t type de support : ext2, ext3, msdos, vfat, iso9660 : Cd-Rom, nfs pour démonter : `umount /mnt/floppy`
- CDROM : `mount -t iso9660 /dev/cdrom /mnt/cdrom`
- Clé usb : `mount -t vfat /dev/sda1 /mnt/usb`
- Disque externe : `mount -t ext2 /dev/sda1 /mnt/usbdisk`
- X window : Installation de la carte graphique à l'aide de XF86Config placé dans `/etc/X11/`, ou `/usr/X11R6/lib/X11/XF86Config`

# Interpréteur de commandes shell

- Le shell est un programme ayant pour fonction d'assurer l'interface entre l'utilisateur et le système Unix. C'est un interpréteur de commandes.
- Plusieurs shell sont disponibles sur les plateformes Unix.
- Principaux interpréteurs de commandes :
  - `/usr/bin/sh` : Correspond au shell POSIX ou shell Bourne
  - `/usr/bin/ksh` : Correspond au Korn Shell 88
  - `/usr/bin/bash` ou `/bin/bash` : correspond au Bourne Again Shell
  - `/usr/bin/csh` : correspond au Cshell

# Vue d'ensemble de C

## ▪ Origine de C :

- Dennis Ritchie : Concepteur du C sur DEC-PDP-11 sous UNIX – 1972
- Langage intermédiaire
  - Niveau le plus haut : Ada, Modula-2, Pascal, COBOL, FORTRAN, BASIC
  - Niveau intermédiaire : Java, C++, C, FORTH
  - Niveau le plus bas : Macro-assembleur, Assembleur

## ▪ Avantage du C :

- utilisable sur plusieurs plates-formes
- programmation de certains instruments (acquisition de données)
- ne se limite pas uniquement au domaine de la physique (FORTRAN)
- beaucoup plus utilisé que le FORTRAN
- compilateur gratuit sur plusieurs plates-formes
- Le système d'exploitation UNIX est écrit en C, facilité d'écriture des interfaces...

# Premiers pas en C

## Un exemple :

```
#include  
<stdio.h>  
  
main() {  
    printf('bonjour tout le monde C\n');  
}
```

-include <stdio.h> :

- main() : Tout programme admet une fonction main
- { } délimitent le corps de la fonction

-printf : fonction de la librairie standard, fonction permettant d'imprimer à l'écran.

printf(format [, paramètres ]\*); le premier paramètre indique le format qu'on désire utiliser pour l'impression, chaînes de caractères et codes de conversion introduit par %. Chaque code de conversion est associé à un paramètre de l'appel, qui est converti selon le format demandé puis affiché.

- Toutes les instructions en C se terminent par ;



# Principaux code de conversion de **printf**

%d : entier affiché en décimale

%f : Nombre flottant de type flottant ou double

%e,%E : Nombre flottant de type flottant ou double

%g,%G :Nombre flottant de type flottant ou double

%Lf : Nombre flottant de type flottant ou double, %Le,%LE,%Lg,%LG)

%c : caractère

%s : chaîne de caractères terminée par un NULL

%x : entier non signé affiché en hexadécimale avec les minuscules a à f

%X : idem avec les majuscules A à F

%o : entier non signé affiché en octale

%% : affiche la caractère %

\n : début d'une ligne

\b : recul vers la gauche

\f : saut de page

\\ : barre oblique inverse

\' : apostrophe

\a : sonnerie

\t : tabulation horizontale

# Identificateurs variables expressions

- **Les identificateurs** : noms des variables, fonctions ; étiquettes composées d'un ou de plusieurs caractères de longueur quelconque.
- **Variables** : désigne un emplacement de mémoire servant à stocker une valeur susceptible d'être modifiée par le programme, elles doivent être déclarées avant leur utilisation suivant la syntaxe :

**type** liste\_de\_variables;

Exemples

**int** i, j , l;

**short int** si;

**unsigned int** ui;

**double** balance , profit, perte;

- les variables **locales ou automatiques** sont déclarées au sein des fonctions
- Les variables **globales** sont déclarées en dehors des fonctions
- **Déclaration** : allocation d'un espace mémoire pour une variable et permettre au calculateur d'effectuer des opérations spécifiques correctement (a+b) entier ou à flottante.

# Types de données définis dans la norme ANSI/ISO

- 5 types de données : `char` (caractère), `int` (entier), `float` (nombre à virgule flottante en simple précision) , `double` (nombre à virgule flottante en double précision) et `void` (sans valeur). La taille et les valeurs dépendent des processeurs et compilateurs
- `signed short` est équivalent à `short`
- `signed int` est équivalent à `int`
- `signed long` est équivalent à `long`

# Types de données définis dans la norme ANSI/ISO

- **Les identificateurs** sont soumis à la restriction des noms réservés : **goto, default, case, if , operator,...**
- **Expression** : une expression représente une donnée simple qui est dans la majorité des cas un nombre. Elle peut être une constante, une variable ou une combinaison des deux. Elle peut représenter une valeur logique qui est vrai(1) ou faux(0).
  - $A + B$  : représente la somme de variables
  - $X = Y$  : affectation de y à x
  - $T = U + V$  : la valeur de l'expression  $u + v$  affectée à t
  - $X \leq Y$  : comparaison des valeurs de x et de y cette expression a la valeur 1 si c'est vrai et 0 si c'est faux
  - $++ J$  : expression incrémentant la valeur de j d'une unité

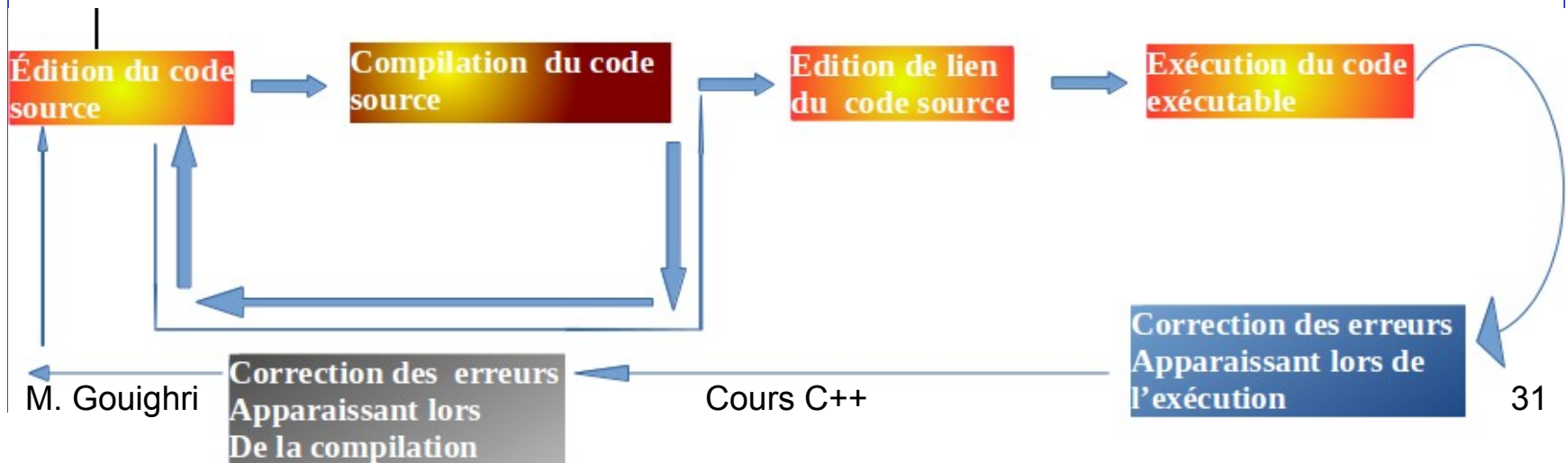
# **La programmation C++**

# Introduction Générale

- Le langage de programmation C++ est le résultat d'une évolution du langage de programmation C, l'un des langages les plus populaires dans le monde de l'informatique. Malgré qu'il est un descendant du langage C, le langage C++ est significativement différent parce qu'il s'articule autour des notions de programmation objet.
- En programmation, un objet est un groupe d'informations stockés dans une partie de la mémoire de l'ordinateur. A chaque objet est associé un type de données, ce type définissant l'utilisation qui sera faite des données. Tous les langages de programmation possèdent des types de données internes ou (built-in) tels les entiers ou les réels.
- Le langage C++ devient de plus en plus populaire pour les raisons suivantes :
  - C++ est relativement facile à apprendre
  - les programmes en C++ s'exécutent rapidement
  - les programmes en C++ sont concis
  - le C++ est disponible sur plusieurs plates-formes matérielles.

# Compilation et exécution d'un programme simple :

- Un programme source contient le texte décrivant les instructions du langage dans lequel le programme est écrit. L'éditeur permet d'entrer ce texte à l'ordinateur et le sauvegarder dans un fichier source sur le disque dur de l'ordinateur, ce qui permet d'en conserver le contenu à long terme et de le réutiliser ultérieurement.
- Une fois le programme source (code source) entré, il faut ensuite compiler avec un programme appelé compilateur, pour produire ce qu'on appelle code objet. Ce code objet a une forme incompréhensible pour l'utilisateur mais qui est pleine de sens pour l'ordinateur.
- Le code source peut être distribué dans plusieurs fichiers sources différents qui peuvent être compilés séparément pour mener à plusieurs fichiers de code objet. Pour joindre ces fichiers de code objet en un programme fonctionnel, il suffit de les lier ensemble à l'aide d'un programme appelé éditeur de liens.



# Exemple de programme

Un programme type en C++ est généralement composé de :

- Définitions de fonctions contenant des déclarations de variables informant le compilateur sur les différentes variables qui seront utilisées
- D'énoncés informant le compilateur des tâches qui seront effectuées par le programme.

Tout programme en C++ doit contenir la définition d'une fonction appelée *main*. Le programme ci-dessous, qui calcule la puissance dissipée dans une résistance de 10 ohms traversée par un courant de 20 ampères, ne contient qu'une seule fonction, la fonction *main*.

```
main()
{
10*20*20
}
```

ce petit programme est plutôt *hermétique* car il n'accepte aucune donnée d'entrée et ne produit aucune donnée de sortie dont l'utilisateur peut prendre connaissance.



# Exemple de programme

Pour rendre le programme moins hermétique, il suffit d'ajouter des énoncés informant le compilateur que l'utilisateur désire que les résultats du calcul de la puissance dissipée s'affichent à l'écran de l'ordinateur

```
main()  
{  
cout << " la puissance dissipée est : " ;  
cout << 10*20*20 ;  
cout << endl ;  
}
```

l'opérateur de sortie, <<, (insertion)

Dans ce programme, on compte trois instructions de sortie à l'écran

- La première instruction demande à l'opérateur de sortie << d'afficher une chaîne de caractères (délimitée à chaque extrémité par le symbole " ).
- La seconde instruction de sortie affiche le résultats d'un opérateur arithmétique
- La dernière instruction de sortie fait l'appel à l'acronyme endl signifiant **endline**, ce qui informe le compilateur de terminer la ligne courante et de démarrer une nouvelle ligne.

# Exemple de programme

- C++ considère la différence entre les lettres minuscules et les lettres majuscules
- L'opérateur \* et l'opérateur de sortie << sont inclus par défaut dans le langage et travaillent sur des opérandes.
- Il est important de donner des noms de fichiers qui sont suffisamment explicites pour permettre de les reconnaître facilement.
- Dans le cadre de ces notes de cours la compilation du fichier source *exemple1.cpp* s'effectue de la manière suivante (ubuntu 16.04 LTS) `g++ exemple1.cpp -o exemple1`
  - `ccp` signifie que l'on désire que le compilateur cpp traite le fichier source
  - `-o exemple1` signifie que l'on désire que le fichier contenant le code exécutable porte le nom **exemple1**. En général, **lorsque l'option "o" est omise**, le programme exécutable est stocké par défaut dans le fichier **a.out**

# Déclaration des variables en C++

- En C++, un identificateur est un nom formé de lettres et de chiffres, le premier symbole étant une lettre
- Une variable est un identificateur qui sert de nom à une région de la mémoire de l'ordinateur. Une variable identifie donc cet espace de mémoire.
- Le type de cette variable détermine les dimensions de l'espace de mémoire qu'elle sert à identifier.
- L'espace mémoire identifié par la variable renferme le contenu de cette variable.
- Lors de l'exécution d'un programme, le contenu d'une variable peut changer mais le type de la variable ne change jamais.
- Le C++ offre aussi plusieurs possibilités pour représenter des nombre réels : **float**, **double**, **long**, **long double**.

# Déclaration des variables en C++

- Lorsque l'on indique le type d'une variable au compilateur dans la fonction main, on procède alors à la déclaration de la variable. Par exemple, deux variables de type entier (int) sont déclarées dans l'Exemple (1) :
- On peut regrouper plusieurs déclarations de variables du même type en modifiant la syntaxe Exemple (2), le séparateur virgule sert à séparer les identificateurs des différentes variables.
- On peut initialiser une variable en procédant comme dans l'Exemple (3) :
- L'opérateur de stocker une valeur dans la mémoire réservée à une variable est appelée "affectation"

/\* Exemple (1) \*/

```
main() {  
int resistance;  
int courant;  
}
```

/\* Exemple (2) \*/

```
main() {  
int resistance, courant;  
}
```

/\* Exemple (3) \*/

```
main () {  
int resistance = 100;  
int courant = 20;  
}
```

## Question :

Parmi les trois exemples, quelle est l'exemple le plus performant ?

Pour les plus curieux, il est possible de connaître la quantité de mémoire qui est réservée pour un type de données grâce à l'opérateur `sizeof` du C++. Le code suivant montre l'utilisation de cet opérateur :

```
#include <iostream.h>

using namespace std;

main() {
    cout << "Type de donnees Octets " << endl;
    cout << "Char " << sizeof(char) << endl;
    cout << "short" << sizeof(short) << endl;
    cout << "int" << sizeof(int) << endl;
    cout << "long " << sizeof(long) << endl;
    cout << "float" << sizeof(float) << endl;
    cout << "double" << sizeof(double) << endl;
    cout << "long double" << sizeof(long double) << endl;
}
```

### Remarque :

Le C++ offre deux méthodes différentes pour l'inclusion de commentaire dans les programmes. Premièrement, lorsque le compilateur rencontre des barres obliques consécutives dans le code source, il ignore les deux barres obliques et tout ce qui les suit sur la ligne.

# Les expressions arithmétiques

Les quatre opérateurs arithmétiques fondamentaux du C++ sont +, -, \*, /, soit l'addition, la soustraction, la multiplication et la division.

- $5/3$ ; // *Resultat donne 1* : L'opérateur de division du C++ tronque le résultat entier plutôt que de l'arrondir.
- $5\%3$ ; // *Resultat donne 2* : L'opérateur *modulo* du C++, représenté par le symbole %, produit le reste de la division entière.
- $5.0/3.0$  // *Resultat donne 1.66667* : La division de nombre en format point flottant donne un nombre du même format.
- les opérateurs arithmétiques peuvent contenir aucun, un seul ou plusieurs opérateurs.
- Le C++ suit la pratique commune quand vient le temps d'évaluer la priorité des opérateurs entre eux.

## Exemple :

Programmer les trois opérations suivantes :  $6+3*2$ ,  $6*3/2$ ,  $6/3*2$   
à votre avis comment on peut contourner la priorité par défaut du C++ ?

# Priorité des opérateurs

- Le compilateur C++ donne la priorité à l'opérateur de multiplication sur l'opérateur d'addition.
- Les opérateurs de multiplication et de la division ont le même niveau de priorité, on peut dire que ces deux opérateurs obéissent à une règle d'associativité de gauche à droite.
- La plupart des opérateurs du C++ sont binaires, c'est-à-dire qu'ils requièrent deux opérandes qui sont placées immédiatement à gauche et à droite de l'opérateur. D'autres opérateurs, comme par exemple l'opérateur de négation arithmétique -, sont dits unaires parce qu'ils n'acceptent qu'une seule opérande placée immédiatement à droite de l'opérateur. La priorité de l'opérateur unaire " - " est supérieure à celle des opérateurs + , -(binaire), \* et / .
- Lorsque le compilateur rencontre une expression arithmétique contenant des éléments appartenant à plusieurs types, celui-ci convertit le nombre entier en notation point-flottant float avant d'effectuer la multiplication.
- $x = (\textit{double})i$  ; cette instruction force la conversion de la variable i en type double, c'est l'opérateur de casting
- $y=3$  c'est un **opérateur d'affectation**, produit la valeur 3 en plus d'affecter cette valeur à la variable y. Contrairement aux autres opérateurs du C++ , l'opérateur
- = procède à une associativité de droite à gauche.
- l'opérateur de sortie << applique l'associativité de gauche à droite. il a une **priorité inférieure** à celle de tous les opérateurs arithmétiques sauf l'affectation.

# Lecture des informations au clavier

L'opérateur d'entrée, `>>`, aussi appelé **opérateur d'extraction**, est le complément de l'opérateur de sortie. lorsqu'il est utilisé de pair avec `cin`, correspondant à l'endroit où les données doivent être obtenues, cet opérateur reçoit une donnée du clavier de l'ordinateur et place cette information dans une variable.

```
#include <iostream.h>

using namespace std;
main() {
    int a, b, c;
    cout << "SVP, entrer des entiers au clavier " << endl;
    cin >> a;          // une énoncé d'entrée peut contenir plusieurs
    cin >> b;          // opérateurs d'entrée cin >> a >> b >> c;|
    cin >> c;
    cout << "Le produit est :" << a*b*c << endl;
}
```

- Si l'opérateur d'entrée `>>` est utilisé, il faut en informer le compilateur en incluant la ligne suivante au début du programme
- Si on désire rediriger l'entrée d'un programme du clavier vers un fichier de données, il suffit d'utiliser la redirection

*nom du programme < nom du fichier >*



# Les Fonctions

Cette partie présente un concept important en C++ soit la notion de fonction. Nous avons déjà abordé la fonction main essentielle à tout programme en C++.

```
#include <iostream.h>

using namespace std;
main() {

    cout <<"La puissance dissipée est :" <<endl;
    cout <<10*20*20 << endl;
}
```

Ce programme ne fonctionne que pour calculer la puissance dissipée dans une résistance de 10 Ohms traversée par un courant de 20 Ampères. Si l'on veut calculer la puissance dissipée dans des résistances différentes parcourues par différents courants, il serait intéressant de concevoir une fonction appelée *puissance – dissipée*.

```
#include <iostream.h>

using namespace std;

//...Placer ici le code de la fonction puissance_dissipee
main() {

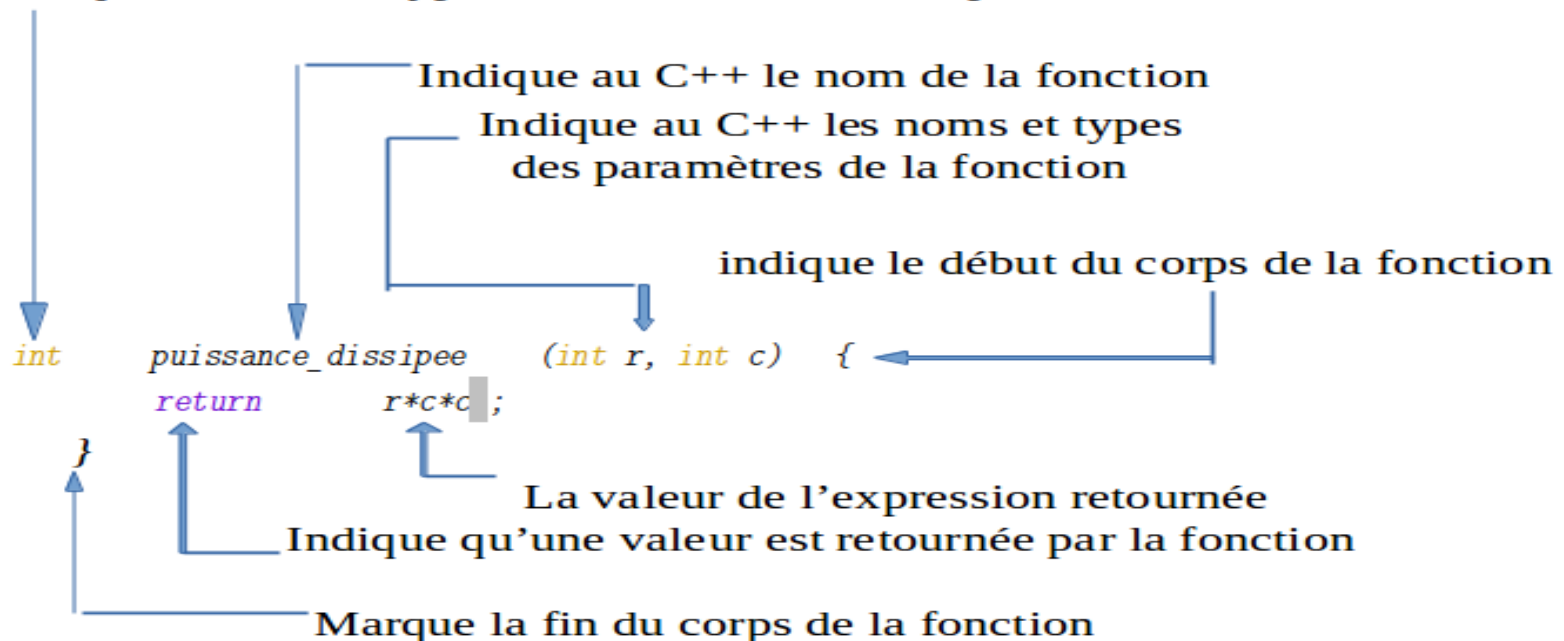
    cout <<"La puissance dissipée est :" <<endl;
    cout << puissance_dissipee(10,20) << endl;
}
```

- Dans l'exemple qui précède, les arguments sont des valeurs constantes (10 et 20). Cependant, les arguments d'une fonction peuvent aussi être des expressions contenant des variables, par exemple *puissance – dissipée(resistance, courant)*
- Nous voyons maintenant comment définir la fonction *puissance – dissipée*. La définition prend la forme suivante :

```
int puissance_dissipee (int r, int c) {
    return r*c*c
}
```

- La signification de chaque partie du code de la fonction puissance dissipée est donnée ci-dessous :

| Indique au C++ le type de la valeur retournée par la fonction



A chaque fois que le nom de la fonction *puissance\_dissipée* apparaît dans le programme, le compilateur C++ doit effectuer les tâches suivantes :

- écrire les valeurs de ces expressions dans l'espace mémoire réservé identifier les espaces mémoire réservés avec des noms de paramètres
- évaluer l'expression  $r * c * c$  qui est la puissance dissipée
- retourner la valeur de la puissance  $r * c * c$  pour l'utiliser dans d'autres calculs.
- Noter bien que les paramètres de la fonction ne sont que des variables qui sont initialisées à la valeur des arguments à chaque fois que la fonction est appelée.

```
#include <iostream.h>
```

```
using namespace std;
```

```
//...Placer ici le code de la fonction puissance_dissipee
```

```
int puissance_dissipee (int r, int c){
```

```
    return r*c*c ;
```

```
main() {
```

```
    cout <<"La puissance dissipée est :" <<endl;
```

```
    cout << puissance_dissipee(10,20) << endl;
```

```
    cout <<"La puissance dissipée est :" <<endl;
```

```
    cout << puissance_dissipee(100,50) << endl;
```

```
}
```

# Exercices

- Concevez un programme qui calcule le volume d'une boîte rectangulaire. Les longueurs des côtés de la boîte doivent être lus au clavier de l'ordinateur lors de l'exécution du programme.
- Concevez une fonction qui reçoit en argument la valeur de deux résistances et qui retourne la valeur de la combinaison en série de ces résistances. La valeur de la combinaison en série de deux résistances est simplement la somme de la valeur de chaque résistance.

- Quand la valeur de retour d'une fonction est de type entier (*int*), il n'est pas nécessaire d'écrire ce détail dans la déclaration de la fonction.
- Il peut arriver qu'une fonction ne retourne aucune valeur. Cela se produit lorsqu'on conçoit une fonction pour afficher le contenu de variables mais sans en faire nécessairement usage (*void*).
- Il est important de noter qu'une fonction peut aussi appeler une autre fonction.

```
#include <iostream>

using namespace std;

int fonction_1() {
    return 1;
}

void fonction_appel() {
    cout << "appel de la fonction " << fonction_1() << endl;
}

main() {
    fonction_appel();
}
```

- Il faut se rappeler que comme C++ tient compte des majuscules, les fonctions *puissance-dissipee* et *PUISSANCE-DISSIPEE* seraient différentes si elles appartenaient au même programme.

- Le C++ offre aux programmeurs un outil intéressant appelé *"surdéfinition de fonctions"* qui permet de définir des fonctions ayant le même nom mais ayant des variantes dans le type ou le nombre de paramètres ou dans le type de la valeur de retour.

Il serait potentiellement intéressant d'avoir deux fonctions :

```
#include <iostream>

using namespace std;

int puissance_dissipee (int r, int c) {
    cout << " version (int): " << endl;
    return r*c*c;
}

double puissance_dissipee (double r, double c) {
    cout << " version (double): " << endl;
    return r*c*c;
}

main() {

    int res_int = 10, cour_int = 20;
    double res_double = 100.35, cour_double = 5.23;
    cout << "La puissance dissipée est :";
    cout << puissance_dissipee(res_int, cour_int) << endl;
    cout << "La puissance dissipée est :";
    cout << puissance_dissipee(res_double, cour_double) << endl;

}
```

Cette partie présente les avantages liés à l'utilisation des fonctions :

- Lorsqu'un détail de calcul est déplacé du programme principal à une fonction, on cache ces détails derrière la barrière de la fonction.
- L'utilisation de fonction reporte les détails des calculs dans la fonction, hors de la vue du programmeur qui ne veut qu'utiliser la fonction pour faire un calcul sans nécessairement connaître en détail comment cette fonction est implantée.
- L'utilisation des fonctions permet de segmenter le code en petits segments indépendants.
- Le fait de réutiliser une fonction à plusieurs endroits dans un programme évite de recopier les instructions de cette fonction.

# Variables locales et globales

- La durée de vie d'une variable est la période de temps durant laquelle de l'espace mémoire est réservé pour cette variable. La portée d'une variable est la portion d'un programme pour laquelle elle peut être évaluée ou recevoir une valeur par le processus d'affectation.
  - Lorsqu'une fonction est appelée, la valeur des paramètres de celles-ci n'est disponible qu'à l'intérieur de la fonction.
  - Lorsque la fonction est appelée la valeur de variables est protégée.

```
#include <iostream>
```

```
using namespace std;
```

```
void fonction_1 (int a, double b) {  
    a = a + 10;  
    b = b + 20.2;  
    cout << " Dans la fonction " << endl;  
    cout << " a : " << a << "      b: " << b << endl;  
}
```

```
main() {
```

```
    int a = 5;    double b = 8.9;  
    cout << "Avant l'appel de la fonction :" << endl;  
    cout << " a : " << a << "      b: " << b << endl;  
    fonction_1(a,b);  
    cout << "Après l'appel de la fonction :" << endl;  
    cout << " a : " << a << "      b: " << b << endl;  
}
```



Le résultats de l'exemple qui précède nous permet d'arriver aux cinq conclusion suivantes :

- Les valeurs des paramètres d'une fonction ne sont pas accessibles en dehors de celle-ci.
- Lorsqu'une fonction appelle une autre fonction, la valeur des paramètres dans la fonction appelante ne sont pas accessibles durant l'exécution de la fonction appelée. On voit bien qu'il est impossible que la fonction *fonction-1* puisse utiliser les variables de la fonction appelante *main* puisque *fonction-1* est déclarée avant même que les variables de *main* ne soient elles-mêmes déclarées dans ce cas spécifique.
- Lorsqu'une variable apparaît en dehors du corps de toutes les fonctions d'un programme, elle est à la fois déclarée et définie parce que le compilateur est informé sur le type de la variable et parce que le compilateur réserve de la mémoire pour stocker le contenu de cette variable lors de la compilation.
- L'espace d'une variable d'une fonction ne sera réservé que lorsque la fonction sera appelée lors de l'exécution du programme.
- Une fonction est toujours à la fois définie et déclarée parce qu'on doit spécifier le type de retour de la fonction et parce que le compilateur réserve de l'espace pour le code de la fonction lors de la compilation.

# Variables locales et globales

- Une variable déclarée à l'intérieur d'une fonction est appelée **variable locale**, ou encore **automatique**. Les règles suivantes s'appliquent aux variables locales :
  - Les variables locales ne sont accessibles que dans la fonction à l'intérieur de laquelle elles sont déclarées.
  - La valeur des variables locales d'une fonction n'est plus accessible une fois que l'exécution de cette fonction est complétée.
  - Lorsqu'une fonction en appelle une autre, la valeur des variables de la fonction appelante ne sont pas accessibles lors de l'exécution de la fonction appelée.
- Une variable est définie à l'extérieur de toute fonction est appelée variable globale :
  - La valeur des variables globales est accessible de toutes les fonctions définies après la définition de la variable sauf dans le cas où un paramètre ou une variable locale de la fonction possède le même nom. (**masquage**)
  - aux endroits où une variable globale n'est pas masquée par une variable ou un paramètre local, sa valeur peut être modifiée par une opération d'affectation. Ce changement est permanent dans le sens où la nouvelle valeur stockée dans la variable globale efface l'ancienne valeur qui s'y trouvait et cette valeur est définitivement perdue.
- On qualifie les variables globales de **statiques** (l'espace mémoire réservé n'est jamais réaloué) elles ont une **portée universelle**, et les variables locales de **dynamique** (la mémoire est réalouée dès que l'exécution de la fonction est complétée) elle ont une **portée locale**

Comme exemple, supposons que vous désiriez calculer la valeur de la phase  $\omega t$  d'un phaseur  $e^{i\omega t}$  à l'instant  $t = 8.7 \text{ sec}$ . La fréquence d'oscillation du phaseur est  $f = 20 \text{ Hz}$ .

```
#include <iostream>

using namespace std;
const double pi=3.14159; /* variable globale accessible de toutes
                           les parties du programme*/

double phase (double f, double t) {
    return 2*pi*f*t;
}

main() {
    double f = 20, t= 8.7;
    cout << "***** : " << endl;
    cout << " phase : " << phase (f, t) << endl;
}
```

### Remarque :

Au lieu de définir nous-même la constante mathématique  $\pi$ , nous aurions pu procéder de façon plus judicieuse en utilisant directement les ressources mises à notre disposition par les bibliothèques de constantes et de fonctions mathématiques

```

#include <iostream>
#include <math.h>    /* La librairie dont les ressources (constante,
                    fonction mathématique) sont disponible*/

using namespace std;

double phase (double f, double t) {
    return 2*M_PI*f*t; // notez bien la majuscules de pi
}

main() {
    double f = 20, t= 8.7;
    cout << "***** : " << endl;
    cout << " phase : " << phase (f, t) << endl;
}

```

# Résumé (Partie I)

- C++ offre des opérateurs de négation, d'addition, de soustraction, de multiplication, de division, de sortie et d'affectation et suit les règles courantes de priorité et d'associativité
- pour rendre des expressions arithmétiques plus claires, l'usage des parenthèses est recommandé.
- l'opérateur de sortie à une priorité inférieure de celle des opérateurs arithmétiques. L'opérateur d'affectation a une priorité encore plus basse que tous ces opérateurs.
- Si l'opérateur d'entrée >> est utilisé, il faut en informer le compilateur en incluant la ligne suivante au début du programme `#include < iostream.h >`
- Si le programme doit utiliser des données fournies au clavier, il suffit d'utiliser un énoncé incluant `cin >>`.
- Lorsqu'une fonction est appelée dans un programme, ses arguments sont évalués et copiés dans les paramètres.
- La déclaration d'une fonction exige que le type de chaque argument soit spécifié de même que le type de la valeur de retour
- Le C++ supporte la surdéfinition de fonctions qui consiste à définir deux fonctions ayant le même nom mais qui diffèrent de par leur paramètres ou leur valeur de retour.
- Une variable locale est une variable qui est déclarée à l'intérieur d'une fonction. Une variable globale est pour sa part définie à l'extérieur de toutes les fonctions d'un programme.

# Exercice (Partie I)

## Questions :

- Ecrire un petit programme qui fait l'addition, soustraction, multiplication et la division
- Ecrire une fonction distance ayant comme paramètres 4 doubles xa, ya et xb, yb qui représente les coordonnées de deux points A et B et qui renvoie la distance AB
- Ecrire un petit programme qui fait appeler le theoreme de Pythagor.

```

#include <iostream>
#include <math.h>
using namespace std;

    void Pythagoras(void);    //Prototype de fonctions
    void Setvars(void);      //Prototype de fonctions

double a, b, c; /* Déclaration de variables globales
                  (Notez qu'elles sont déclarées avant main)*/

int main(){
    Setvars();
    Pythagoras();

    cout << " hypothénus est égale à " << c << endl;
}
void Pythagoras(void){
    c =sqrt((a*a)+(b*b));
}
void Setvars(void){
    cout << "entrez la valeur de a " << endl;
    cin >> a;
    cout << "entrez la valeur de b " << endl;
    cin >> b;
}

```

# Structures de controles

- Un programme est un flux d'instructions qui est executé dans l'ordre. Pour casser cette linéarité et donner au programme une relative intelligence, les langage de programmation permettent d'effectuer des choix et des boucles.
- On va parler de de blocs d'instructions :
- Il s'agit d'un ensemble d'instructions entouré d'accolades ouvrantes et fermantes.

```
{  
    x=val;  
    .....  
}
```



# Enoncés conditionnels

Les structures de contrôle nous permettent de prendre des décisions et d'exécuter des boucles d'une façon lisible, nous voyons comment on peut utiliser les énoncés conditionnels du C++ pour effectuer des calculs dont la nature dépend de la valeur d'une expression impliquant éventuellement un ou plusieurs prédicats.

- **Expression booléenne** est une expression qui produit la valeur **faux** ou **vrai**. En C++, cela signifie que l'expression produit 0 (associé à faux) ou tout entier différent de 0 (associé à vrai)
- Un énoncé **if** comprend une **expression booléenne** contenue entre des parenthèses, suivie d'un **énoncé associé** :  
**if (expression booléenne) énoncé associé**
- Lorsque l'expression booléenne d'un énoncé **if** produit une valeur entière différente de 0, C++ considère que cette expression est vraie et l'énoncé associé est alors exécuté. Autrement, si l'expression booléenne produit la valeur 0, le C++ considère cette expression comme étant fausse et l'énoncé associé n'est pas exécuté.

## Exemple :

Supposons qu'on désire écrire un programme qui affiche un message dépendant de la puissance dissipée dans une résistance. Si la puissance dissipée est plus grande que la puissance maximum (de 10 watts), on affiche "puissance trop élevée". Si la puissance dissipée est inférieure ou égale à 10 watts, on affiche " Puissance Ok"

# Structures de controles

- L'énoncé **if-else** ressemble à l'énoncé **if** sauf qu'il possède un second énoncé associé qui suit le mot **else** **if (expression booléenne) énoncé associé si vrai else énoncé associé si faux**
- Le premier énoncé associé est exécuté si l'expression booléenne est vraie. Autrement, c'est le second énoncé qui est exécuté.
- L'énoncé associé d'un énoncé **if** ou les énoncés associés d'un énoncé **if-else** peuvent eux-même contenir des énoncés **if** ou **if-else**
- L'utilisation de crochets rend possible l'exécution de plusieurs énoncés associés à l'énoncé **if** ou **if-else**.
- L'opérateur conditionnel permet de produire un résultat en fonction de la valeur d'un prédicat logique.

## Exemple :

- Supposons qu'on désire écrire un programme qui affiche un message dépendant de la puissance dissipée dans une résistance. Si la puissance dissipée est plus grande que la puissance maximum (de 20 watts), on affiche "puissance trop élevée". Si la puissance dissipée est entre 20 et on affiche "puissance reste toujours élevée" si la puissance inférieure ou égale à 10 watts, on affiche " Puissance Ok"
- Ecrire un programme qui va nous permettre de trouver la solution d'une équation de deuxième degré.

Le code suivant peut poser des problèmes d'interprétation :

```
#include <iostream>

using namespace std;

int main(){

    int puissance;
    cout << "entrez la valeur de la puissance" << endl;
    cin >> puissance;

    if (puissance > 2)
        if (puissance <=10)
            cout <<"puissance OK." << endl;
        else cout << "puissance ??????" << endl;
    }
```

**Question** : par quoi devrait-on remplacer le " ? ? ? ? " dans l'énoncé **else**

Le code suivant peut poser des problèmes d'interprétation :

```
#include <iostream>

using namespace std;

int main(){

    int puissance;
    cout << "entrez la valeur de la puissance" << endl;
    cin >> puissance;

    if (puissance > 2)
        if (puissance <=10)
            cout <<"puissance OK." << endl;
        else cout << "puissance ??????" << endl;
    }
```

**Question** : par quoi devrait-on remplacer le " ? ? ? ? " dans l'énoncé **else**

**Remarque** :

Le C++ assume que chaque **else** doit être païé avec le **if** le plus près qui n'a pas déjà un **else** qui lui est païé.

Afin d'éviter toute ambiguïté, il est préférable d'utiliser des crochets pour mieux délimiter les énoncés **if-else**

# Remarques

- Remarquez que, contrairement aux autres opérateurs, **l'opérateur conditionnel** possède une syntaxe combinant deux symboles (soit **" ?" et " :"**) séparant trois expressions. pour cette raison, on qualifie l'opérateur conditionnel d'opérateur ternaire.
- Remarquez aussi que seulement une des deux expressions (**vrai ou faux**) est **évaluée**. Toute opération ou affectation dans l'énoncé non évalué ne sera pas exécutée.
- Voyons comment nous pouvons tirer profit de l'opérateur conditionnel dans ce programme :

```
#include <iostream>

using namespace std;

int main(){

    int delta_puissance;
    cout << "entrez la valeur de la variation de la puissance" << endl;
    cin >> delta_puissance;
    cout << "L'augmentation de la puissance est de: " << delta_puissance
        << (delta_puissance ==1 ? " watt": " watts") << endl;
}
```

# Remarques

- Remarquez que, contrairement aux autres opérateurs, **l'opérateur conditionnel** possède une syntaxe combinant deux symboles (soit **" ?" et " :"**) séparant trois expressions. pour cette raison, on qualifie l'opérateur conditionnel d'opérateur ternaire.
- Remarquez aussi que seulement une des deux expressions (**vrai ou faux**) est **évaluée**. Toute opération ou affectation dans l'énoncé non évalué ne sera pas exécutée.
- Voyons comment nous pouvons tirer profit de l'opérateur conditionnel dans ce programme :

```
#include <iostream>

using namespace std;

int main(){

    int delta_puissance;
    cout << "entrez la valeur de la variation de la puissance" << endl;
    cin >> delta_puissance;
    cout << "L'augmentation de la puissance est de: " << delta_puissance
        << (delta_puissance ==1 ? " watt": " watts") << endl;
}
```

**Si la valeur de notre variable est de 1 watt, l'opérateur conditionnel produit la valeur "watt" autrement, il produit la valeur "watts".**

# Combinaison logique d'expressions booléennes

- L'opérateur logique **ET**, symbolisé par **&&** (les esperluettes), retourne la valeur 1 si ses deux opérandes ont une valeur entière différentes de 0 et retourne 0 autrement.
- L'opérateur logique **OU**, symbolisé par **||**, retourne la valeur 1 si au moins une de ses opérandes a une valeur différente de 0 et retourne 0 autrement.

## Programme (1)

```
#include <iostream>

using namespace std;

int main(){

    int puissance;
    cout << "Entrez la puissance" << endl;
    cin >> puissance;
    if (puissance > 10 && puissance <20)
        cout << "Puissance normale: " << endl;
}
```

## Programme (2)

```
#include <iostream>

using namespace std;

int main(){

    int puissance;
    cout << "Entrez la puissance" << endl;
    cin >> puissance;
    if (puissance > 10 || puissance <20)
        cout << "Puissance normale: " << endl;
}
```

**Question :** Donnez via ces deux programmes la manière dont le compilateur évalué des expressions comprenant **&&** ou **||**.

# Itérations: while & for

Cette partie présente les instructions **while** et **for**. Ces **opérations d'itération** permettent de répéter une séquence d'instructions ou d'opérations un certain nombre de fois (exactement jusqu'à ce qu'une condition logique soit vérifiée).

## L'instruction while :

- L'instruction **while** du C++ est composée d'une **expression booléenne** contenue entre parenthèses suivie d'un **énoncé associé**:

**while** (expression booléenne) **Enoncé associé**

- L'expression booléenne est d'abord évaluée et si elle retourne une valeur différente de 0, l'énoncé associé est exécuté. Autrement, le C++ passe outre à l'énoncé associé et exécute les instructions suivantes.

```
#include <iostream>

using namespace std;

int main(){

    int dose;
    cout << "Entrez la dose mSv" << endl;
    cin >> dose;

    while (dose <=10) {
        dose = dose + 1;
        cout << "Dose valide est: " << dose << endl;
    }
}
```



## Exercice avancé

En électronique numérique, il est souvent nécessaire d'évaluer la puissance  $n$  du nombre 2, notée  $2^n$ . L'instruction `while` permet d'implanter une fonction très intéressante pour le calcul de la  $n^{\text{ième}}$  puissance de 2. Concevez ce programme.

# Exercice avancé

En électronique numérique, il est souvent nécessaire d'évaluer la puissance  $n$  du nombre 2, notée  $2^n$ . L'instruction while permet d'implanter une fonction très intéressante pour le calcul de la  $n^{\text{ième}}$  puissance de 2. Concevez ce programme.

```
#include <iostream>

using namespace std;
int puissance_de_2(int n){
    int resultat=1;
    while (n!=0) {
        resultat = resultat*2;
        n=n-1;
    }

    return resultat;
}

int main(){
    int exposant;
    cout << "entrez l'exposant " << endl;
    cin >> exposant;

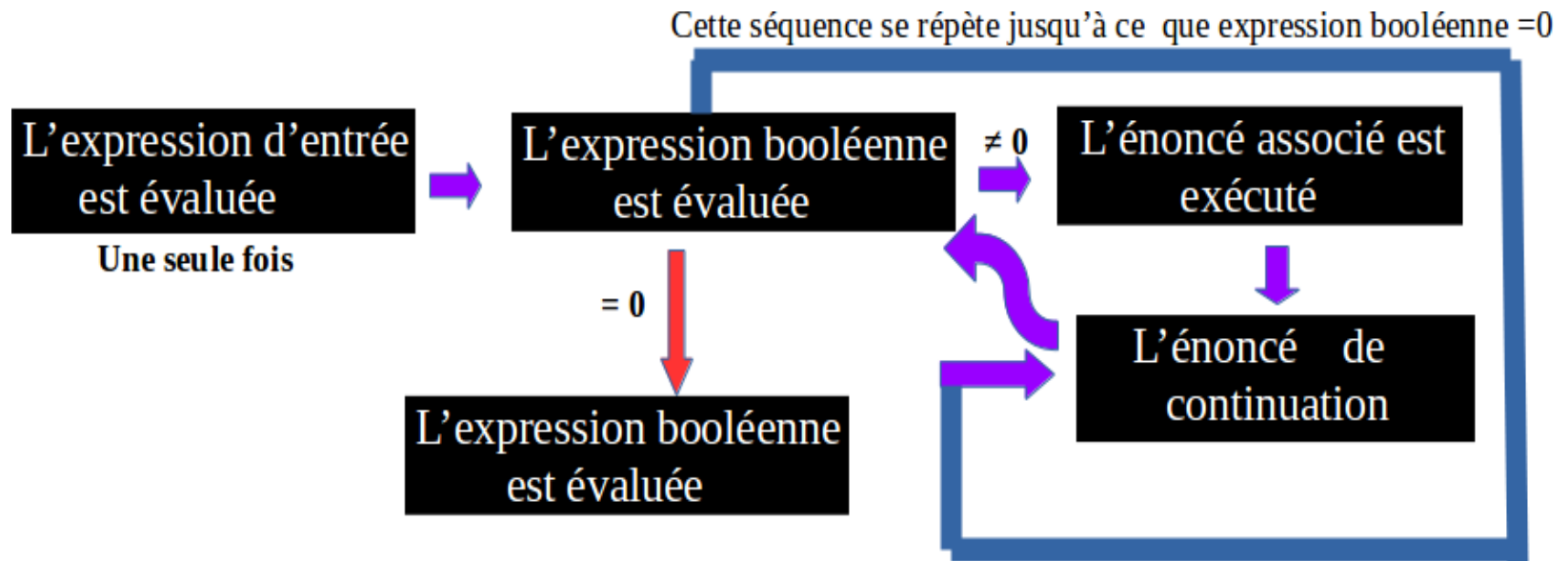
    while (exposant) { // <==> while (exposant!=0)
        cout << " 2 puissance " << exposant << "eagle: "
            << puissance_de_2(exposant) << endl;
        exposant = exposant -1;
    }
}
```

# Boucle for

- Le principal ennui avec l'instruction `while` est qu'il faut **trois lignes** pour implanter une itération : **l'initialisation du compteur, le test sur la valeur du compteur et troisièmement, l'incrément** (ou la décrémentation du compteur).
- L'instruction `for` est un moyen simple de faire des itérations en C++ sans avoir la lourdeur de l'instruction `while`. La syntaxe de l'instruction `for` est la suivante : `for (expression d'entrée ; expression booléenne ; expression de continuation) énoncé associé`

Nous allons maintenant tirer profit de la puissance de l'instruction `for` pour améliorer le programme de calcul des puissance de 2.

```
#include <iostream>
using namespace std;
int puissance_de_2(int n){
    int resultat=1;
    int compteur;
    for (compteur =n; compteur; compteur = compteur-1) {
        resultat = resultat*2;
    }
    return resultat;
}
int main(){
    int e, exposant;
    cout << "entrez l'exposant " << endl;
    cin >> exposant;
    for (e=exposant; e; e = e-1) {
        cout << " 2 puissance " << exposant << "eagle: "
            << puissance_de_2(e) << endl;
    }
}
```



### Des notions utiles :

- *resultat* = 2 \* *resultat* Comme ce type d'instruction se trouve souvent, le C++ offre une syntaxe abrégée appelée **opération d'affectation étendue** *resultat*\* = 2
- Nous pouvons donc utiliser la notation compacte, de la façon suivantes :
  - *compteur* = *compteur* - 1 par - - *compteur* (décrémenter comme préfixe)
  - *compteur* = *compteur* + 1 par + + *compteur* (incrémenter comme préfixe)
- Concevez un programme qui accepte deux entiers *n* et *m* au clavier et qui calcule  $n^m$ .

# Boucle for avec compteur

```
#include <iostream>
using namespace std;
int main(){
    for (int i = 0; i < 10; i++)
    {
        cout << "i = " << i << endl;
    }
}
```

- Ce programme, une fois compilé et exécuté affichera simplement à l'écran les nombre de 0 à 9.
- On aurait pu évidemment ce résultat avec une boucle while.

```
$ g++ ex.cpp
```

```
roland@DESKTOP-M1EA3EP ~
```

```
$ ./a.exe
```

```
i = 0
```

```
i = 1
```

```
i = 2
```

```
i = 3
```

```
i = 4
```

```
i = 5
```

```
i = 6
```

```
i = 7
```

```
i = 8
```

```
i = 9
```

# Break, continue et goto

## Instructions de branchement inconditionnel :

- **break et continue** : s'utilisent principalement dans des boucles afin de contrôler plus finement le flux d'exécution.
- **break** : permet de sortir de la boucle à n'importe quel moment (souvent une condition validée dans la boucle par un if)
- **continue** : va stopper prématurément le tour de boucle actuel et passer directement au suivant
- **goto** : est une instruction déconseillée, elle s'utilise conjointement à des étiquettes dans le code et permet d'y aller directement. Même si cela semble intéressant en première approche, son usage sera interdit lors de ce cours.

# Programmez le modèle de la goutte liquide

Von Weizsacker a proposé une approche semi-empirique de la masse des noyaux, donc de l'énergie de liaison, pour ceux appartenant à la vallée de stabilité. Avec un petit nombre de paramètres, obtenus par ajustement sur l'ensemble des valeurs expérimentales :

$$M(A, Z)c^2 = Zm_p c^2 + (A - Z)m_n c^2 - a_v A + a_s A^{2/3} + a_c \frac{Z^2}{A^{1/3}} + a_a \frac{A - 2Z}{A} + \delta$$

$$E_l(A, Z) = [Zm_p + (A - Z)m_n - M(A, Z)]c^2$$

En utilisant cette relation, on obtient :

$$E_l(A, Z) = a_v A + a_s A^{2/3} - a_c \frac{Z^2}{A^{1/3}} - a_a \frac{A - 2Z}{A} + \delta$$

Le dernier terme est une correction provenant du fait que les nucléons ont tendance à s'apparier (force de pairing).

$\delta = 0$	pour les noyaux pair-impair ou impair-pair (A impair)			
$\delta = \frac{a_p}{A^{1/2}}$	pour les noyaux pair-pair (Z et (A - Z) pairs)			
$\delta = -\frac{a_p}{A^{1/2}}$	pour les noyaux impair-impair (Z et (A - Z) impairs)			
$a_v (MeV)$	$a_s (MeV)$	$a_c (MeV)$	$a_a (MeV)$	$a_p (MeV)$
15,46	17,23	0,697	23,285	12

# Les pointeurs

Modes d'adressage de variables. Définition d'un pointeur. Opérateurs de base. Opérations élémentaires. Pointeurs et tableaux. Pointeurs et chaînes de caractères. Pointeurs et enregistrements. Tableaux de pointeurs. Allocation dynamique de la mémoire. Libération de l'espace mémoire.



# Tableaux et pointeurs

## Premier exemple

```
#include <iostream>
using namespace std;
main()
{
    int t[10];

    for (int i = 0; i < 10; i++)
        t[i] = i;
    for (int i = 0; i < 10; i++)
        cout << "t["<i><<"</i><<"</i> << " : " << t[i] << endl;
}
```

- La déclaration `int t[10]` réserve en mémoire l'emplacement pour 10 éléments de type entier.
- Dans la première boucle, on initialise chaque élément du tableau. Le premier étant conventionnellement numéroté 0.
- Dans la deuxième boucle, on parcourt chaque élément du tableau pour l'afficher.
- On notera que la notation `[]` s'emploie aussi bien pour la déclaration que pour l'accès à un élément du tableau.

- Il ne faut pas confondre les éléments d'un tableau avec le tableau lui-même.
- Ainsi, `t[2] = 3`, `tab[i]++` sont des écritures valides.
- Mais `t1 = t2`, si `t1` et `t2` sont des tableaux, n'est pas possible.

**Il n'existe pas en C++ de mécanisme d'affectation globale pour les tableaux.**

# L'importance des pointeurs

- On peut accéder aux données en mémoire à l'aide de pointeurs i.e. des variables pouvant contenir des adresses d'autres variables.
- Comme nous le verrons dans le chapitre suivant, en C, les pointeurs jouent un rôle primordial dans la définition de fonctions :

Les pointeurs sont le seul moyen de changer le contenu de variables déclarées dans d'autres fonctions.

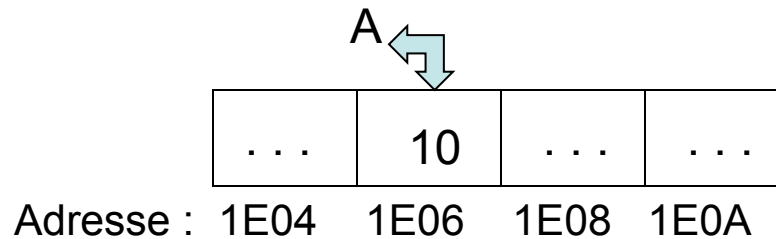
- Le traitement de tableaux et de chaînes de caractères dans des fonctions serait impossible sans l'utilisation de pointeurs.
- Les pointeurs nous permettent de définir de nouveaux types de données : les piles, les files, les listes, ....
- Les pointeurs nous permettent d'écrire des programmes plus compacts et plus efficaces.
- Mais si l'on n'y prend pas garde, les pointeurs sont une excellente technique permettant de formuler des programmes incompréhensibles.

# Mode d'adressage direct des variables

## Adressage direct :

- ❑ Jusqu'à maintenant, nous avons surtout utilisé des variables pour stocker des informations.
- ❑ La valeur d'une variable se trouve à un endroit spécifique dans la mémoire de l'ordinateur.

short A;  
A = 10;



- ❑ Le nom de la variable nous permet alors d'accéder directement à cette valeur.

Dans l'adressage direct, l'accès au contenu d'une variable se fait via le nom de la variable.

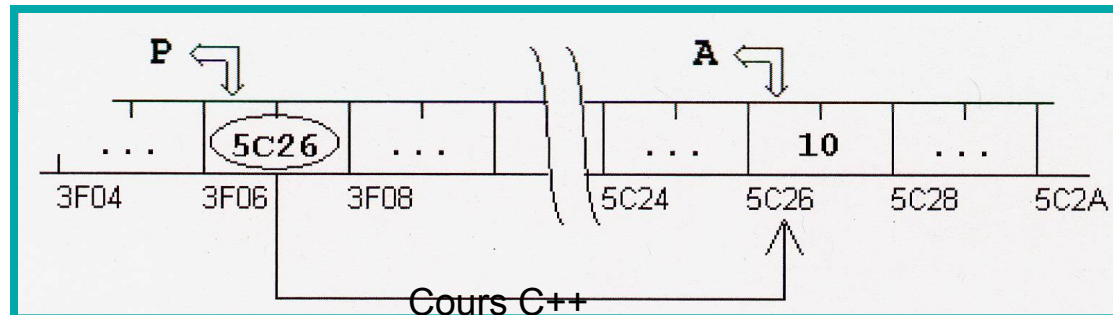
# Mode d'adressage indirect des variables

## Adressage indirect :

- ❑ Si nous ne voulons ou ne pouvons pas utiliser le nom d'une variable A, nous pouvons copier l'adresse de cette variable dans une variable spéciale, disons P, appelée pointeur.
- ❑ Nous pouvons alors retrouver l'information de la variable A en passant par le pointeur P.

Dans l'adressage indirect, l'accès au contenu d'une variable se fait via un pointeur qui renferme l'adresse de la variable.

**Exemple :** Soit A une variable renfermant la valeur 10, et P un pointeur qui contient l'adresse de A. En mémoire, A et P peuvent se présenter comme suit :



# Définition d' un pointeur

Un **pointeur** est une variable spéciale pouvant contenir l' adresse d' une autre variable.

- En C, chaque pointeur est limité à un type de données. Il ne peut contenir que l' adresse d' une variable de ce type. Cela élimine plusieurs sources d' erreurs.

Syntaxe permettant de déclarer un pointeur :

type de donnée \* identificateur de variable pointeur;

Ex. : `int * pNombre;` `pNombre` désigne une variable pointeur pouvant contenir uniquement l' adresse d' une variable de type `int`.

Si `pNombre` contient l' adresse d' une variable entière `A`, on dira alors que `pNombre` pointe vers `A`.

- Les pointeurs et les noms de variables ont le même rôle : ils donnent accès à un emplacement en mémoire.  
Par contre, un pointeur peut contenir différentes adresses mais le nom d' une variable (pointeur ou non) reste toujours lié à la même adresse.
- **Bonne pratique de programmation** : choisir des noms de variable appropriés (Ex. : `pNombre`, `NombrePtr`).

# Comment obtenir l'adresse d'une variable ?

- Pour obtenir l'adresse d'une variable, on utilise l'opérateur `&` précédant le nom de la variable.

Syntaxe permettant d'obtenir l'adresse d'une variable :

`& nom de la variable`

Ex. : `int A;`  
`int * pNombre = &A;`

ou encore,

`int A;`  
`int * pNombre;`  
`pNombre = &A;`

pNombre désigne une variable pointeur initialisée à l'adresse de la variable A de type `int`.

Ex. : `int N;`  
`printf("Entrez un nombre entier : ");`  
`scanf("%d", &N);`



`scanf` a besoin de l'adresse de chaque paramètre pour pouvoir lui attribuer une nouvelle valeur.

**Note :** L'opérateur `&` ne peut pas être appliqué à des constantes ou des expressions.

# Comment accéder au contenu d'une adresse ?

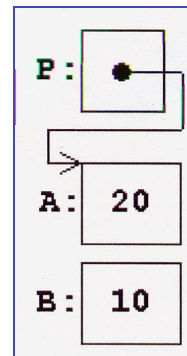
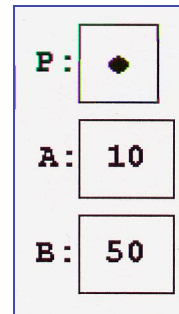
- Pour avoir accès au contenu d'une adresse, on utilise l'opérateur `*` précédant le nom du pointeur.

Syntaxe permettant d'avoir accès au contenu d'une adresse :

`* nom du pointeur`

Ex. :     `int A = 10, B = 50;`  
         `int * P;`

`P = &A;`  
`B = *P;`  
`*P = 20;`



`*P` et `A` désigne le même emplacement mémoire et `*P` peut être utilisé partout où on peut écrire `A` (ex. : `cin >> *P;`).

# Priorité des opérateurs \* et &

- Ces 2 opérateurs ont la même priorité que les autres opérateurs unaires (!, ++, --).
- Dans une même expression, les opérateurs unaires \*, &, !, ++, -- sont évalués de droite à gauche.

Après l'instruction

```
P = &X;
```

les expressions suivantes, sont équivalentes:

```
Y = *P+1    ⇔ Y = X+1
```

```
*P = *P+10  ⇔ X = X+10
```

```
*P += 2     ⇔ X += 2
```

```
++*P        ⇔ ++X
```

```
(*P)++     ⇔ X++
```

Parenthèses

obligatoires sans quoi, cela donne lieu à un accès non autorisé.



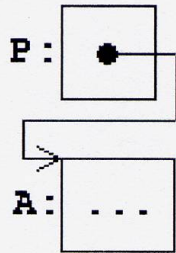
# Le pointeur NULL

- Pour indiquer qu'un pointeur pointe nulle part, on utilise l'identificateur NULL (On doit inclure `stdio.h` ou `iostream.h`).
- On peut aussi utiliser la valeur numérique 0 (zéro).

```
int * P = 0;
```

```
if (P == NULL) printf("P pointe nulle part");
```

# En résumé ...



Après les instructions:

```
int A;  
int *P;  
P = &A;
```

**A** désigne le contenu de A

**&A** désigne l'adresse de A

**P** désigne l'adresse de A

**\*P** désigne le contenu de A

En outre:

**&P** désigne l'adresse du pointeur P

**\*A** est illégal (puisque A n'est pas un pointeur)

$A == *P \Leftrightarrow P == \&A$

$A == *\&A$  et  $P == \&*P$

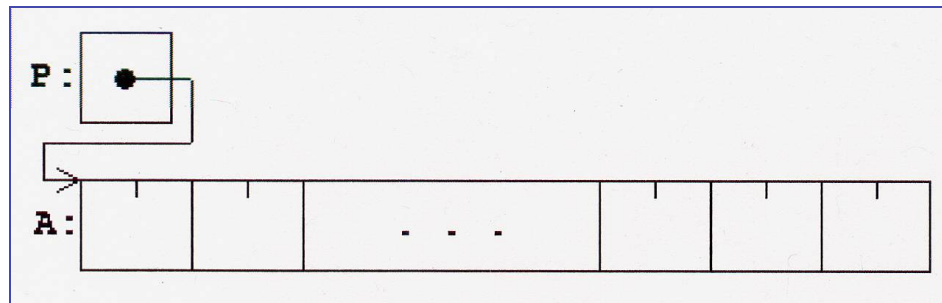
# Pointeurs et tableaux

- Chaque opération avec des indices de tableaux peut aussi être exprimée à l'aide de pointeurs.
- Comme nous l'avons déjà constaté, le nom d'un tableau représente l'adresse de la première composante.

`&tableau[0]` et `tableau` sont une seule et même adresse.

- Le nom d'un tableau est un pointeur **constant** sur le premier élément du tableau.

```
int A[10];  
int * P;  
P = A;    est équivalente à P = &A[0];
```



# Adressage des composantes d' un tableau

- Si P pointe sur une composante quelconque d' un tableau, alors P + 1 pointe sur la composante suivante.

P + i pointe sur la i<sup>ème</sup> composante à droite de \*P.

P - i pointe sur la i<sup>ème</sup> composante à gauche de \*P.

Ainsi, après l' instruction **P = A;**

**\*(P+1)** désigne le contenu de A[1]

**\*(P+2)** désigne le contenu de A[2]

... ..

**\*(P+i)** désigne le contenu de A[i]

- Incréméntation et décrémentation d' un pointeur

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

**P++;** P pointe sur A[i+1]

**P+=n;** P pointe sur A[i+n]

**P--;** P pointe sur A[i-1]

**P-=n;** P pointe sur A[i-n]

Ces opérateurs (+, -, ++, --, +=, -=) sont définis seulement à l' intérieur d' un tableau car on en peut pas présumer que 2 variables de même type sont stockées de façon contiguë en mémoire.

# Calcul d'adresse des composantes d'un tableau

**Note :** Il peut paraître surprenant que  $P + i$  n'adresse pas le i<sup>ème</sup> octet après  $P$ , mais la i<sup>ème</sup> composante après  $P$ .

**Pourquoi ?** Pour tenter d'éviter des erreurs dans le calcul d'adresses.

**Comment ?** Le calcul automatique de l'adresse  $P + i$  est possible car, chaque pointeur est limité à un seul type de données, et le compilateur connaît le # d'octets des différents types.

Soit  $A$  un tableau contenant des éléments du type **float** et  $P$  un pointeur sur **float**:

```
float A[20], X;  
float *P;
```

Après les instructions,

```
P = A;  
X = *(P+9);
```

$X$  contient la valeur du dixième élément de  $A$ , i.e. celle de  $A[9]$ .

Une donnée de type **float** ayant besoin de 4 octets, le compilateur obtient l'adresse  $P + 9$  en ajoutant  $9 * 4 = 36$  octets à l'adresse dans  $P$ .

# Soustraction et comparaison de 2 pointeurs

## - Soustraction de deux pointeurs

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

**P1 - P2** fournit le nombre de composantes comprises entre P1 et P2.

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

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

## - Comparaison de deux pointeurs

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

Mêmes tableaux : Comparaison des indices correspondants.

Tableaux différents : Comparaison des positions relatives en mémoire.



## Différence entre un pointeur et le nom d'un tableau

Comme **A** représente l'adresse de **A[0]**,

**\* (A+1)** désigne le contenu de **A[1]**

**\* (A+2)** désigne le contenu de **A[2]**

...

**\* (A+i)** désigne le contenu de **A[i]**

- Un *pointeur* est une variable,  
donc des opérations comme **P = A** ou **P++** sont permises.

- Le *nom d'un tableau* est une constante,  
donc des opérations comme **A = P** ou **A++** sont impossibles.

## Résumons ...

Soit un tableau A de type quelconque et i un indice d'une composante de A,

**A** désigne l'adresse de **A[0]**

**A+i** désigne l'adresse de **A[i]**

**\*(A+i)** désigne le contenu de **A[i]**

Si  $P = A$ , alors

**P** pointe sur l'élément **A[0]**

**P+i** pointe sur l'élément **A[i]**

**\*(P+i)** désigne le contenu de **A[i]**



## Copie des éléments positifs d'un tableau S dans un tableau T

```
#include <iostream.h>

void main()
{
    int S[10] = { -3, 4, 0, -7, 3 , 8, 0, -1, 4, -9};
    int T[10];
    int i, j;

    for (i = 0, j = 0; i < 10; i++)
        if (*(S + i) > 0)
        {
            *(T + j) = *(S + i);
            j++;
        }

    for (i = 0; i < j; i++) cout << *(T + i) << " ";
    cout << endl;
}
```

## Rangement des éléments d'un tableau dans l'ordre inverse

```
#include <iostream.h>
void main()
{
    int N;
    int tab[50];
    int somme = 0;
    cout << "Entrez la dimension N du tableau : ";
    cin >> N;
    for (int i = 0; i < N; i++)
    {
        cout << "Entrez la " << i << " ieme composante : ";
        cin >> *(tab+i);
        somme += *(tab+i);
    }
    for (int k = 0; k < N / 2; k++)
    {
        int echange = *(tab+k);
        *(tab+k) = *(tab + N - k - 1);
        *(tab + N - k - 1) = echange;
    }
}
```

## Rangement des éléments d'un tableau dans l'ordre inverse

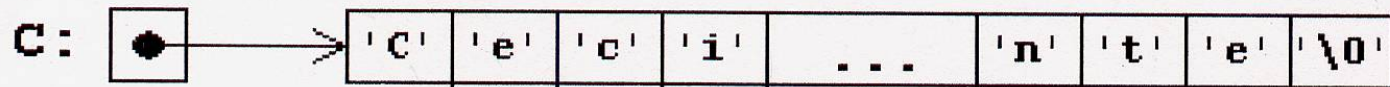
```
cout << endl << endl << "Affichage du tableau inverse." << endl;
for (int j = 0; j < N; j++)
{
    if((j % 3) == 0) cout << endl;
    cout << "tab[ " << j << " ] = " << *(tab+j) << "\t";
}
}
```

# Pointeurs et chaînes de caractères

- Tout ce qui a été mentionné concernant les pointeurs et les tableaux reste vrai pour les pointeurs et les chaînes de caractères.
- En plus, un pointeur vers une variable de type char peut aussi contenir l'adresse d'une chaîne de caractères constante et peut même être initialisé avec une telle adresse.

## Exemple

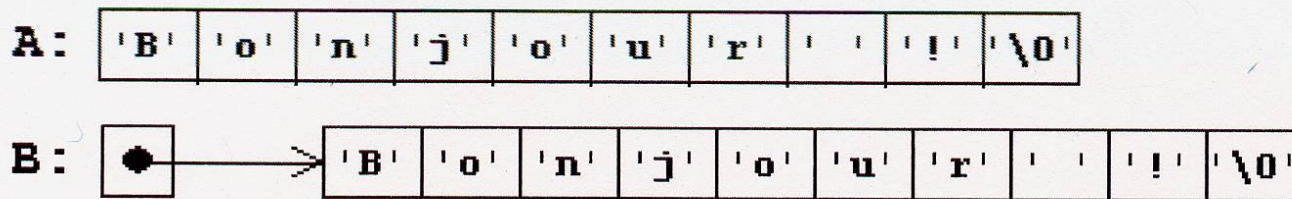
```
char *C;  
C = "Ceci est une chaîne de caractères constante";
```



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

# Distinction entre un tableau et un pointeur vers une chaîne constante

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



**A** a exactement la grandeur pour contenir la chaîne de caractères et \0.  
Les caractères peuvent être changés mais **A** va toujours pointer sur la même adresse en mémoire (pointeur constant).

## Exemple

```
char A[45] = "Petite chaîne";  
char B[45] = "Deuxième chaîne un peu plus longue";  
char C[30];  
A = B;          /* IMPOSSIBLE -> ERREUR !!! */  
C = "Bonjour !"; /* IMPOSSIBLE -> ERREUR !!! */
```

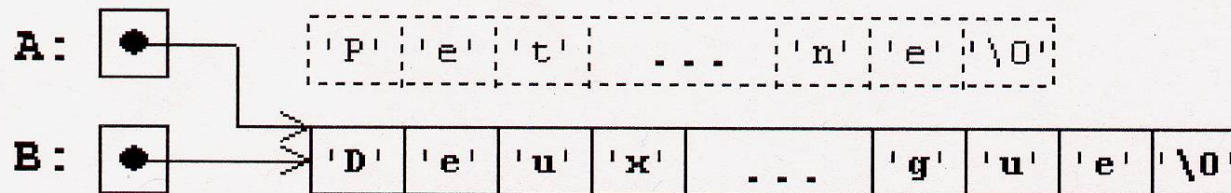
# Distinction entre un tableau et un pointeur vers une chaîne constante

**B** pointe sur une chaîne de caractères constante. Le pointeur peut être modifié et pointer sur autre chose (la chaîne de caractères constante originale sera perdue). La chaîne constante peut être lue, copiée ou affichée, mais pas modifiée (`B[1] = 'o';` est illégal).

## Exemple

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

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



Un pointeur sur `char` a l'avantage de pouvoir pointer sur des chaînes de n'importe quelle longueur.

# Avantage des pointeurs sur char

- Un pointeur vers char fait en sorte que nous n'avons pas besoin de connaître la longueur des chaînes de caractères grâce au symbole \0.
- Pour illustrer ceci, considérons une portion de code qui copie la chaîne CH2 vers CH1.

```
char * CH1;  
char * CH2;
```

...

1<sup>ière</sup> version :

```
int I;  
I=0;  
while ((CH1[I]=CH2[I]) != '\0')  
    I++;
```

2<sup>ième</sup> version :

Un simple changement de notation nous donne ceci :

```
int I;  
I=0;  
while ((* (CH1+I)=* (CH2+I)) != '\0')  
    I++;
```

# Avantage des pointeurs sur char

Exploitions davantage le concept de pointeur.

```
while ((*CH1=*CH2) != '\0')  
{  
    CH1++;  
    CH2++;  
}
```

Un professionnel en C obtiendrait finalement :

```
while (*CH1++ = *CH2++)  
    ;
```



# Pointeurs et tableaux à deux dimensions

Soit `int M[4][10] = {`    `{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},`  
                                  `{10,11,12,13,14,15,16,17,18,19},`  
                                  `{20,21,22,23,24,25,26,27,28,29},`  
                                  `{30,31,32,33,34,35,36,37,38,39}};`

M représente l'adresse du 1<sup>e</sup> élément du tableau et pointe vers le tableau M[0] dont la valeur est : { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. De même, M + i est l'adresse du i<sup>ème</sup> élément du tableau et pointe vers M[i] dont la valeur est la i<sup>ème</sup> ligne de la matrice.

```
cout << (*(M+2))[3];           // 23
```

## Explication :

Un tableau 2D est un tableau unidimensionnel dont chaque composante est un tableau unidimensionnel. Ainsi, M + i désigne l'adresse du tableau M[i].

## Question :

Comment accéder à l'aide de pointeurs uniquement à une composante M[i][j] ?

Il s'agit de convertir la valeur de M qui est un pointeur sur un tableau de type `int` en un pointeur de type `int`.

# Pointeurs et tableaux à deux dimensions

Solution :

```
int M[4][10] = {{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
                 {10,11,12,13,14,15,16,17,18,19},
                 {20,21,22,23,24,25,26,27,28,29},
                 {30,31,32,33,34,35,36,37,38,39}};

int *P;
P = (int *)M;  /* conversion forcée */
```

Puisque le tableau 2D est mémorisé ligne par ligne et que cette dernière affectation entraîne une conversion de l'adresse &M[0] à &M[0][0] \*, il nous est maintenant possible de traiter M à l'aide du pointeur P comme un tableau unidimensionnel de dimension 40.

- \* P et M renferme la même adresse mais elle est interprétée de deux façons différentes.

# Pointeurs et tableaux à deux dimensions

Exemple : Calcul de la somme de tous les éléments du tableau 2D M.

```
int M[4][10] = {{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
                 {10,11,12,13,14,15,16,17,18,19},
                 {20,21,22,23,24,25,26,27,28,29},
                 {30,31,32,33,34,35,36,37,38,39}};

int *P;
int I, SOM;
P = (int*)M;
SOM = 0;
for (I=0; I<40; I++)
    SOM += *(P+I);
```

**Note :** Dans cet exemple, toutes les lignes et toutes les colonnes du tableau sont utilisées. Autrement, on doit prendre en compte

- le nombre de colonnes réservé en mémoire,
- le nombre de colonnes effectivement utilisé dans une ligne,
- le nombre de lignes effectivement utilisé.

# Tableaux de pointeurs

## Syntaxe :

type \* identificateur du tableau[nombre de composantes];

Exemple : `int * A[10];` // un tableau de 10 pointeurs  
// vers des valeurs de type `int`.

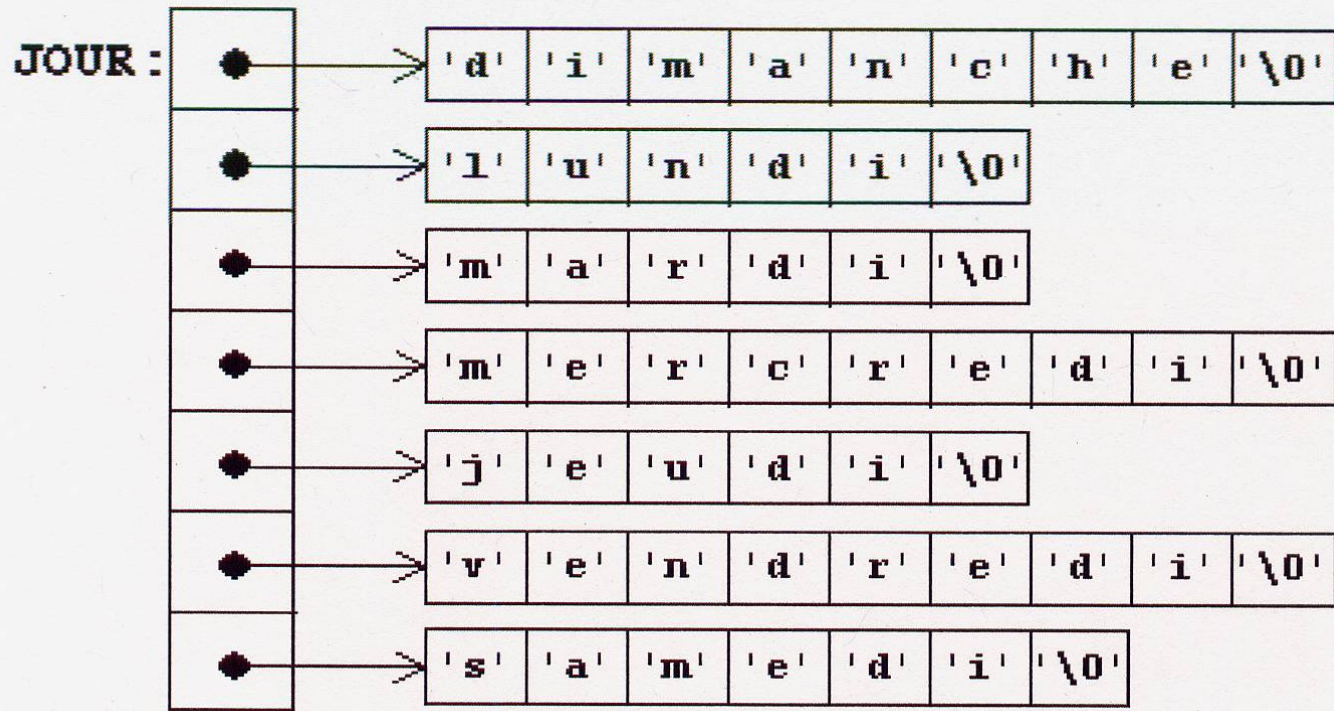
Tableaux de pointeurs vers des chaînes de caractères de différentes longueurs

### *Exemple*

```
char *JOUR[] = {"dimanche", "lundi", "mardi",  
               "mercredi", "jeudi", "vendredi",  
               "samedi"};
```

Nous avons déclaré un tableau JOUR[] de 7 pointeurs de type char, chacun étant initialisé avec l'adresse de l'une des 7 chaînes de caractères.

# Tableaux de pointeurs



Affichage :

```
int I;  
for (I=0; I<7; I++) printf("%s\n", JOUR[I]);
```

Pour afficher la 1<sup>e</sup> lettre de chaque jour de la semaine, on a :

```
int I;  
for (I=0; I<7; I++) printf("%c\n", *JOUR[I]);
```



# Tableaux de pointeurs

Si  $D[j]$  pointe dans un tableau,

$D[i]$	désigne l'adresse de la première composante
$D[i] + j$	désigne l'adresse de la j-ième composante
$*(D[i] + j)$	désigne le contenu de la j-ième composante

Les tableaux de pointeurs vers des chaînes de caractères de différentes longueurs sont d'un grand intérêt mais ce n'est pas le seul (à suivre).

# Allocation statique de la mémoire

- Jusqu' à maintenant, la déclaration d' une variable entraîne automatiquement la réservation de l' espace mémoire nécessaire.
- Le nombre d' octets nécessaires était connu au temps de compilation; le compilateur calcule cette valeur à partir du type de données de la variable.

## Exemples d' allocation statique de la mémoire

```
float A, B, C;           /* réservation de 12  octets */
short D[10][20];        /* réservation de 400 octets */
char E[] = {"Bonjour !"};
                        /* réservation de 10  octets */
char F[][10] = {"un", "deux", "trois", "quatre"};
                        /* réservation de 40  octets */
```

Il en est de même des pointeurs ( $p = 4$ ).

```
double *G;              /* réservation de p    octets */
char *H;                /* réservation de p    octets */
float *I[10];           /* réservation de 10*p octets */
```

# Allocation statique de la mémoire

Il en est de même des chaînes de caractères constantes ( $p = 4$ ).

## *Exemples*

```
char *J = "Bonjour !";  
        /* réservation de    p+10      octets */  
float *K[] = {"un", "deux", "trois", "quatre"};  
        /* réservation de 4*p+3+5+6+7 octets */
```



# Allocation dynamique de la mémoire

## Problématique :

Souvent, nous devons travailler avec des données dont nous ne pouvons prévoir le nombre et la grandeur lors de l'écriture du programme.

La taille des données est connue au temps d'exécution seulement.

Il faut éviter le gaspillage qui consiste à réserver l'espace maximal prévisible.

**But :** Nous cherchons un moyen de réserver ou de libérer de l'espace mémoire au fur et à mesure que nous en avons besoin pendant l'exécution du programme.

**Exemples :** La mémoire sera allouée au temps d'exécution.

```
char * P;           // P pointera vers une chaîne de caractères
                    // dont la longueur sera connue au temps d'exécution.
```

```
int * M[10];        // M permet de représenter une matrice de 10 lignes
                    // où le nombre de colonnes varie pour chaque ligne.
```

# La fonction malloc et l'opérateur sizeof

- ❑ La fonction malloc de la bibliothèque stdlib nous aide à localiser et à réserver de la mémoire au cours de l'exécution d'un programme.
- ❑ La fonction malloc fournit l'adresse d'un bloc en mémoire disponible de N octets.

```
char * T = malloc(4000);
```

Cela fournit l'adresse d'un bloc de 4000 octets disponibles et l'affecte à T. S'il n'y a plus assez de mémoire, T obtient la valeur zéro.

- ❑ Si nous voulons réserver de l'espace pour des données d'un type dont la grandeur varie d'une machine à l'autre, on peut se servir de `sizeof` pour connaître la grandeur effective afin de préserver la portabilité du programme.

```
sizeof <var>
    fournit la grandeur de la variable <var>
sizeof <const>
    fournit la grandeur de la constante <const>
sizeof (<type>)
    fournit la grandeur pour un objet du type <type>
```

# La fonction malloc et l'opérateur sizeof

Exemple : `#include <stdio.h>`  
`void main()`  
`{`

205081048

```
    short A[10];  
    char B[5][10];  
    printf("%d%d%d%d%d%d", sizeof A, sizeof B,  
           sizeof 4.25,  
           sizeof "Bonjour !",  
           sizeof(float),  
           sizeof(double));  
}
```

Exemple :

Réserver de la mémoire pour X valeurs de type `int` où X est lue au clavier.

```
int X;  
int *PNum;  
printf("Introduire le nombre de valeurs :");  
scanf("%d", &X);  
PNum = malloc(X*sizeof(int));
```

# La fonction malloc et l'opérateur sizeof

## Note :

S'il n'y a pas assez de mémoire pour satisfaire une requête, il est conseillé d'interrompre l'exécution du programme à l'aide de la commande `exit` de `stdlib` et de renvoyer une valeur non nulle comme code d'erreur.

## Exemple :

Lire 10 phrases au clavier et ranger le texte.  
La longueur maximale d'une phrase est fixée à 500 caractères.

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

main()
{
    /* Déclarations */
    char INTRO[500];
    char *TEXTE[10];
    int I;
    /* Traitement */
    for (I=0; I<10; I++)
    {
```

# La fonction malloc et l'opérateur sizeof

```
gets(INTRO);
/* Réserve de la mémoire */
TEXTE[I] = malloc(strlen(INTRO)+1);
/* S'il y a assez de mémoire, ... */
if (TEXTE[I])
    /* copier la phrase à l'adresse */
    /* fournie par malloc, ... */
    strcpy(TEXTE[I], INTRO);
else
{
    /* sinon quitter le programme */
    /* après un message d'erreur. */
    printf("ERREUR: Pas assez de mémoire \n");
    exit(-1);
}
}
return 0;
}
```

## Exple : Matrice triangulaire inférieure (partie I)

Exemple :

$$\begin{pmatrix} 12 & & & \\ -2 & 4 & & \\ 9 & -17 & 50 & \\ -98 & 19 & 25 & 75 \end{pmatrix}$$

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int * M[9];
    int i, j;

    //      Allocation dynamique de la mémoire.

    for (i = 0; i < 9; i++) M[i] = malloc((i+1) * sizeof(int));
```

## Exple : Matrice triangulaire inférieure (partie I)

```
//      Initialisation de la matrice.

for (i = 0; i < 9; i++)
    for (j = 0; j <= i; j++)
        *(M[i] + j) = (i + 1) * 10 + j + 1;

//      Affichage des éléments de la matrice.

for (i = 0; i < 9; i++)
{
    for (j = 0; j <= i; j++)
        printf("\t%d", *(M[i] + j));
    for (j = i+1; j < 9; j++)
        printf("\t0");
    printf("\n");
}
}
```

Reprenons le même exemple où, cette fois, le # de lignes de la matrice est lu.

## Exple : Matrice triangulaire inférieure (partie II)

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

```
void main()
{
```

```
    int * * M; 
    int N;
    int i, j;
```

```
    printf("Entrez le nombre de lignes de la matrice : ");
    scanf("%d", &N);
```

```
    //      Allocation dynamique de la mémoire.
```

```
    M = malloc(N * sizeof(int *)); 
```

```
    for (i = 0; i < N; i++) M[i] = malloc((i+1) * sizeof(int));
```






## Exple : Matrice triangulaire inférieure (partie II)

```
//      Initialisation de la matrice.

for (i = 0; i < N; i++)
    for (j = 0; j <= i; j++)
        *((*(M+i)) + j) = (i + 1) * 10 + j + 1;

//      Affichage des éléments de la matrice.

for (i = 0; i < N; i++)
{
    for (j = 0; j <= i; j++)
        printf("\t%d", M[i][j]);
    for (j = i+1; j < N; j++)
        printf("\t0");
    printf("\n");
}
}
```



The diagram illustrates the reuse of code. A red dotted arrow points from the underlined expression `*((*(M+i)) + j)` in the initialization loop to the underlined expression `M[i][j]` in the display loop. The word "idem" is written in red between the two arrows, indicating that the same memory access pattern is used in both contexts.

# Libération de l'espace mémoire

- Si nous n'avons plus besoin d'un bloc de mémoire que nous avons réservé à l'aide de `malloc`, nous pouvons le libérer à l'aide de la fonction `free` de la librairie `stdlib`.

```
free(pointeur);
```



Pointe vers le bloc à libérer.

## À éviter :

Tenter de libérer de la mémoire avec `free` laquelle n'a pas été allouée par `malloc`.

## Attention :

La fonction `free` ne change pas le contenu du pointeur.

Il est conseillé d'affecter la valeur zéro au pointeur immédiatement après avoir libéré le bloc de mémoire qui y était rattaché.

## Note :

Si la mémoire n'est pas libérée explicitement à l'aide de `free`, alors elle l'est automatiquement à la fin de l'exécution du programme.

# Allocation dynamique, libération de l'espace mémoire en C++

- Le mot clé `new` permet de réserver de l'espace selon le type de donnée fourni.

**Syntaxe :** `new type de donnée`

L'opérateur renvoie l'adresse du bloc de mémoire allouée.  
Si l'espace n'est pas disponible, l'opérateur renvoie 0.

**Exemple :** `unsigned short int * pPointeur;`  
`pPointeur = new unsigned short int;`

ou encore,

`unsigned short int * pPointeur = new unsigned short int;`

- On peut également affecter une valeur à cette zone. Ex. : `*pPointeur = 25;`
- Pour libérer l'espace alloué avec `new`, on utilise l'opérateur `delete` une seule fois.  
Ex.: `delete pPointeur;`

Autrement, il se produira une erreur à l'exécution. Pour éviter ceci, mettez le pointeur à 0 après avoir utilisé l'opérateur `delete`.

# Allocation dynamique, libération de l'espace mémoire en C++

- Libérer le contenu d'un pointeur nul est sans incidence.

```
int * p = new int;  
delete p;           // libérer la mémoire.  
p = 0;  
...  
delete p;           // sans incidence sur le programme.
```

- Réaffecter une valeur à un pointeur alors que celui-ci n'est pas nul génère une perte de mémoire.

```
unsigned short int * pPointeur = new unsigned short int;  
*pPointeur = 72;  
pPointeur = new unsigned short int;  
*pPointeur = 36;
```

À chaque instruction `new` devrait correspondre une instruction `delete`.

- Tenter de libérer le contenu d'un pointeur vers une constante ou une variable allouée de manière statique est une erreur.

Ex. : `const int N = 5;`  
`int * p = &N;`  
`delete p;`

## Allocation dynamique, libération de l'espace mémoire en C++

- Comment libérer l'espace mémoire d'un tableau ?

```
float * p = new float[10];
```

```
...
```

```
delete [] p;
```

```
// libérer la mémoire.
```

```
p = 0;
```

 car, nous sommes en présence d'un tableau.

# Matrice de réels - exemple

```
#include <iostream.h>

void main()
{
    // Saisie de la dimension de la matrice.

    int M, N;
    cout << "Nombre de lignes : ";
    cin >> M;
    cout << "Nombre de colonnes : ";
    cin >> N;

    // Construction et initialisation d'une matrice réelle M x N.

    typedef float * pReel;
    pReel * P;
    P = new pReel[M];
```

# Matrice de réels - exemple

```
for (int i = 0; i < M; i++)  
{  
    P[i] = new float[N];  
    for (int j = 0; j < N; j++)    P[i][j] = (float) 10*i + j;  
}
```

// Affichage d'une matrice M x N.

```
for (i = 0; i < M; i++)  
{  
    cout << endl;  
    for (int j = 0; j < N; j++)    cout << P[i][j] << " ";  
}  
cout << endl;
```

// Libération de l'espace.

```
for (i = 0; i < M; i++)    delete [] P[i];  
delete [] P;
```

# Pointeur générique

Une variable de type `void *` est un pointeur générique capable de représenter n'importe quel type de pointeur.

```
#include <stdio.h>
void main()
{
    int A = 5;          void * P = &A;
    if ((* (int *) P) == 5)           // *P est invalide car on ne connaît
        printf("%d", (*(int *) P));  // pas le type pointé par P.
}
```

On peut affecter un pointeur à un autre si les 2 sont de même type. S'ils ne le sont pas, il faut effectuer une conversion explicite.

La seule exception est le type `void *`. On ne peut toutefois pas affecter un pointeur `void *` directement à un pointeur d'un autre type.

```
#include <iostream.h>
void main()
{
    void * P;          float * R;          int Q = 5;
    P = &Q;            R = (float *)P;
    cout << *R;        // Donne des résultats erronés.
}
```



# Usage de const avec les pointeurs

- Un pointeur constant est différent d'un pointeur à une constante.

```
int n = 44;  
int* p = &n;  
++(*p);
```

```
++p;
```

```
int* const cp = &n;
```

un pointeur constant

```
++(*cp);
```

```
++cp;
```

illégal

```
const int k = 88;
```

```
const int * pc = &k;
```

un pointeur à une constante

```
++(*pc);
```

illégal

```
++pc;
```

```
const int* const cpc = &k;
```

un pointeur constant  
à une constante

```
++(*cpc);
```

illégal

```
++cpc;
```

illégal

# Avant-goût des structures de données

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    // Déclaration des types de données.
    struct Fiche_etudiant
    {
        char nom[25];
        char prenom[25];
        int age;
        bool sexe;
        int matricule;
        struct Fiche_etudiant * suivant;
    };

    struct Fiche_etudiant * pEnsemble_des_fiches = NULL;
    struct Fiche_etudiant * pointeur = NULL;

    int Nombre_de_fiches = 0;
    char test;
```

# Avant-goût des structures de données

// Saisie des fiches des étudiants.

```
for (Nombre_de_fiches = 0; Nombre_de_fiches < 50; Nombre_de_fiches++)
{
    printf("\nVoulez-vous entrer les donnees d'une %s fiche (O ou N) ? ",
           Nombre_de_fiches ? "autre " : "");
    scanf(" %c", &test);
    if(test == 'N' || test == 'n') break;

    pointeur=(struct Fiche_etudiant *) malloc(sizeof(struct Fiche_etudiant));
    (*pointeur).suivant = pEnsemble_des_fiches;
    pEnsemble_des_fiches = pointeur;

    scanf("%s%s%i%i%i",
          (*pEnsemble_des_fiches).nom,
          (*pEnsemble_des_fiches).prenom,
          &(*pEnsemble_des_fiches).age,
          &(*pEnsemble_des_fiches).sexe,
          &(*pEnsemble_des_fiches).matricule);
}
```

# Avant-goût des structures de données

// Affichage de l'ensemble des fiches d'étudiants.

```
pointeur = pEnsemble_des_fiches;
while (pointeur != NULL)
{
    printf("\n%s %s %i %i %i ", (*pointeur).nom,
                                                (*pointeur).prenom,
                                                (*pointeur).age,
                                                (*pointeur).sexe,
                                                (*pointeur).matricule);
    pointeur = (*pointeur).suivant;
}
}
```

La structure de données utilisée est une pile.

# La programmation Orientée Objet

La programmation Orientée Objet est .

# La création des classes et des objets en C++

- Une classe représente un type de données, un gabarit qui permet de créer des variables que l'on appelle objets ou instances de classe ainsi que les méthodes.
- Les types de données standards tel int, double, long, char, float, sont également des classes, ce sont les classes intrinsèques au C++.

Exemple :

```
class    NAND_2
{
    public:
    int in_1;    /*entrée 1*/
    int in_2;    /*entrée 2*/
    int out_1;   /*sortie */
};
```

# La création des classes et des objets en C++

Exemple :

```
class    NAND_2
{
    public:
    int in_1;    /*entrée 1*/
    int in_2;    /*entrée 2*/
    int out_1;   /*sortie */
};
```

cette classe ne décrit que les variables, il n'y a pas de définition de fonction.

- **Public:** toutes les données membres d'une classe définies avec le mot clé public sont utilisables par toutes les fonctions, accès pour la lecture et l'écriture.
- **Private:** tous les membres d'une classe définis avec le mot clé private sont utilisables uniquement par les fonctions membres de cette classe.

# La création des classes et des objets en C++

- **Protected:** tous les membres d'une classe définis avec le mot `protected` sont utilisables uniquement par les fonctions membres des classes dérivées.
- **Déclaration des données (variables ) membres :** Les données membres sont les variables définies à l'intérieur d'une classe, la portée de ces variables est définie avec les étiquettes **public, private, protected**.
- **Déclaration des fonctions membres :** Les fonctions membres correspondent aux fonctions que vous définissez au sein de la classe. Les fonctions représentent l'ensemble des traitements que vous pouvez mettre en œuvre avec les objets de cette base. **Les termes méthodes et fonctions membres sont synonymes.**



# La création des classes et des objets en C++

- Une classe est un type de données qui n'a qu'un seul objectif : permettre la création d'objets. Ces objets permettront d'accéder aux données et fonctions membres d'une classe.

**NAND\_2** **nnd**; (création statique)

- La variable **nnd** correspond à un objet de type **NAND\_2**
- Pour accéder aux variables membres de l'objet **nnd** on utilise l'opérateur d'accès aux membres d'une classe « . » **nnd.in\_1**
- On peut lire le contenu des variables membres ou en modifier le contenu par des opérations d'affectations à ces variables. (**Exemple ex\_class1.cpp**)
- Une définition de classe doit se terminer par « ; » afin de permettre de définir éventuellement des variables globales dans le même énoncé.

# La création des classes et des objets en C++

- Classe pour une porte NOR\_2 (deux entrées) exemple `ex_class2.cpp`, la fonction `set_out_1` est surdéfinie.
- Exercice : 131 et 132.

# La création des classes et des objets en C++

- Une classe est un type de données qui n'a qu'un seul objectif : permettre la création d'objets. Ces objets permettront d'accéder aux données et fonctions membres d'une classe.

**NAND\_2** **nnd**; (création statique)

- La variable **nnd** correspond à un objet de type **NAND\_2**
- Pour accéder aux variables membres de l'objet **nnd** on utilise l'opérateur d'accès aux membres d'une classe « **.** » **nnd.in\_1**
- On peut lire le contenu des variables membres ou en modifier le contenu par des opérations d'affectations à ces variables. (**Exemple ex\_class1.cpp**)
- Une définition de classe doit se terminer par « **;** » afin de permettre de définir éventuellement des variables globales dans le même énoncé.

# Définition des fonctions membres

- La fonction membre fait partie du corps de la classe.
- Chaque fonction membre possède un argument spécial :
  - la valeur de cet argument spécial est un objet appartenant à la même classe que la fonction membre.
  - cet argument n'apparaît pas entre les parenthèses, il est associé, via l'opérateur d'accès aux membres de la classe c'est un argument implicite via l'objet `nnd`.  
« . » `.nnd.set_out_1();`
- Les fonctions membres n'ont aucun paramètre correspondant à l'objet pour lequel elles sont appelées.
- Dans les fonctions membres il n'y a ni paramètre ni variable associés aux variables membres via l'opérateur d'accès « . ». **Exemple : `ex_class3.cpp`**

# Définition des fonctions membres

- Les fonctions membres peuvent recevoir des arguments ordinaires autres que l'argument spécial de l'objet de la classe qui les appelle.

Exemple : ex\_class5.cpp

- **Fonction prototype** : une définition de classe peut contenir plusieurs fonctions membres. Pour plus de visibilité on fait appel au **prototype de la fonction membre**.
- Pour définir les fonctions membres d'une classe à l'extérieur du corps de celle-ci on utilise l'opérateur d'évaluation de portée « :: ».

Exemple : ex\_class4.cpp

- Deux classes différentes peuvent avoir des fonctions membres qui ont le même nom.
- **Syntaxe** : `type nom_de_la_classe nom_de_la_fonction_membre`

# Définition des fonctions membres

- La déclaration des fonctions membres peut se faire de la manière suivante :
    - On définit le prototype de la fonction à l'intérieur de la classe.
    - Le corps de la fonction est défini en dehors de la classe en utilisant le nom complet de la fonction et l'opérateur de résolution de portée « :: ».
- ✓ **Type\_de\_retour : classe::fonction()**

# Les constructeurs

- Ce sont des fonctions membres spéciales, appelées lorsque des objets d'une classe sont créés.
- Les constructeurs permettent l'initialisation des variables membres lors de la création d'objets d'une classe
- Un constructeur porte le nom de sa classe et ne retourne aucune valeur.
- Le constructeur par défaut ne reçoit aucun paramètre Il est conseillé de toujours définir un constructeur par défaut explicitement dans une classe.
- Si on veut initialiser les objets aux valeurs par défaut, il suffit de déclarer ces objets en respectant la syntaxe suivante :

**Nom\_de\_la\_classe    nom\_de\_l'objet;**

Exemple : ex\_class6.cpp

- Si on veut initialiser un objet à des valeurs spécifiques lors de sa création, il suffit de définir un constructeur avec paramètres et de déclarer l'objet en respectant la syntaxe suivante :

**✓ Nom\_de\_la\_classe    nom\_de\_l'objet(arg1,arg2, ...,argn);**

Exercice : 156-p.77

# Les fonctions membres de lecture et d'écriture

- Ces fonctions permettent d'accéder aux valeurs des variables membres et d'affecter des valeurs aux variables membres après leur initialisation avec le constructeur

Exemple : `ex_class7.cpp`

- Une fonction membre de lecture permet d'extraire de l'information d'un objet.
- On peut aussi affecter une valeur à une variable membre par le biais d'une fonction membre d'écriture plutôt que par l'opérateur d'accès aux membres « . ».

Exemple : `ex_class8.cpp`



# Laboratoire

- Rappel sur la logique combinatoire :
- États logiques (0 ou 1), Variables logiques, Fonctions logiques .
- Fonction NON (NOT): la fonction  $S = \text{NON } X$
- Table de vérité :

entrée	sortie
0	1
1	0

- Fonction **ET (AND)**:

X	Y	sortie
0	0	0
1	0	0
0	1	0
1	1	1

# Laboratoire

- Fonction **OU (OR)**:

X	Y	sortie
0	0	0
1	0	1
0	1	1
1	1	1

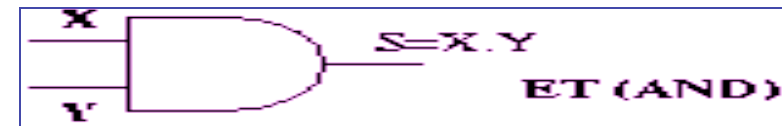
- Fonction **NON (NOR)**:

X	Y	sortie
0	0	1
1	0	0
0	1	0
1	1	0

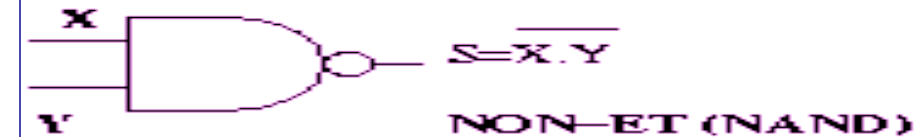
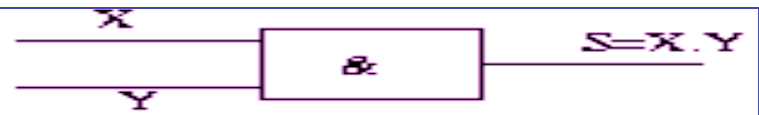
- Fonction **NON ET (NAND)**:

X	Y	sortie
0	0	1
1	0	1
0	1	1
1	1	0

# Fonctions combinatoire de base



ET (AND)



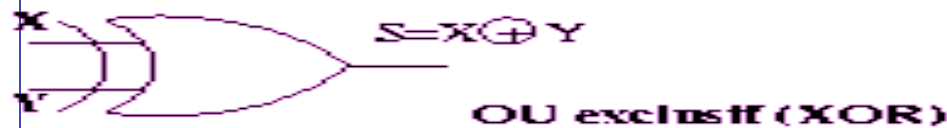
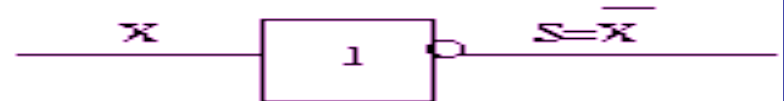
NON-ET (NAND)



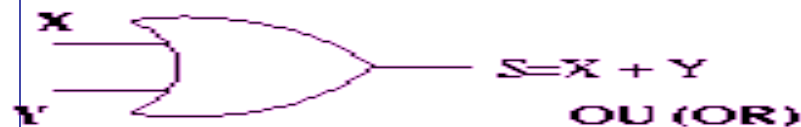
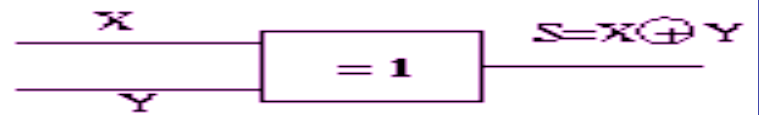
NON-OU (NOR)



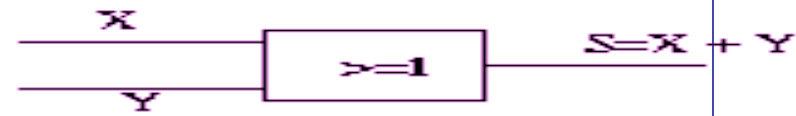
NON (NOT)



OU exclusif (XOR)



OU (OR)



# Comment tirer profit de l'abstraction des données

- Les constructeurs et les fonctions membres d'écriture et de lecture sont appelés fonctions d'accès
- Ces fonctions d'accès permettent de faire de l'abstraction des données
- Les avantages de l'abstraction de données sont de faciliter la réutilisation du code, de faciliter la lecture du code, de permettre de modifier une classe sans avoir à modifier le programme qui l'utilise et, enfin, de permettre d'améliorer la façon dont les données d'une classe sont manipulées

## Protection contre l'accès accidentels aux membres des classes

- L'accès à une variable membre par erreur via l'opérateur d'accès aux membres détruit souvent les efforts d'abstraction des données `nnd.out_1=1;`

Exemple : `class14-1.cpp`

- On peut empêcher ces accès en plaçant les membres dans la partie privée d'une classe et en créant une interface publique adéquate
- Les fonctions membres en général, les constructeurs, les fonctions membres d'écriture et de lecture plus particulièrement ont accès aux variables membres (interface publique), qu'elles soient public ou privées
- Les variables et fonctions membres situées dans la partie publique de la classe forment l'interface publique de la classe.
- L'opposition **public:** est **private:** permet d'implanter l'abstraction des données de façon plus systématique

# Les instructions de préprocesseur

- Avant la compilation d'un programme, il y a d'abord l'étape du preprocessing pour inclure les fichiers du type (\*.h) à l'aide de l'instruction **#include**.
- Le programmeur peut inclure ses propres fichiers à l'aide de l'instruction **#include**
- Avec include on peut utiliser soit < > ou " " pour indiquer le nom du fichier d'inclusion.

## Exemple : ex\_class15-1.cpp et class15-1.cpp

- Instruction de préprocesseur : **#ifndef**, **#undef**, **#ifdef**, **#define** et **#endif**
- Pour éviter d'inclure plusieurs fois le même fichier on utilise le mécanisme de drapeaux (flags).

## Exemple : ex\_class15-1.cpp et class15-2.cpp

- Les instructions du préprocesseur peuvent être utiles pour debugger.

# La notion d'héritage en c++

- Une classe dérivé est une classe obtenue à partir d'une ou de plusieurs classes existantes
- Une classe dont les membres tant données que fonctions s'intègrent dans une autre classe est appelée classe mère ou classe de base (superclasse). Une classe de base transmet tous ses constituants à la classe dérivé à l'exception des constructeurs et opérateurs d'affectation.
- Une classe **B** dérivé d'une classe **A** peut être considérée comme une extension et une spécialisation de la classe **A**. La classe dérivé **B** constitue une généralisation de la classe **A**.

**classe** **nom\_classe\_dérivée** : **[type\_d'accès]** **nom\_classe\_de\_base**

- Il est intéressant d'utiliser une hiérarchie parce que cela permet : de décrire des catégories d'objets réels, d'éviter la duplication de variables membres, d'ajouter des bugs dans du code déjà déverminé et d'utiliser du code disponible commercialement.

# La notion d'héritage en c++

- La création d'un objet d'une classe dérivé entraîne non seulement l'appel au constructeur de cette classe, mais également l'exécution du constructeur de la classe de base.
- Le concept d'héritage est aussi connu sous le nom de dérivation. On peut dire qu'une classe dérive ou hérite d'une autre classe.
- Si B dérive de A, si on crée un objet de B cela entraîne implicitement la création d'un objet de A. Un objet de type B est en réalité un objet de A avec des éléments supplémentaires.

## Exemple d'héritage : classe16-1.cpp

- Quelles que soient les étiquettes de protection, une fonction membre d'une classe peut accéder à toutes les données membres de cette classe.

## Exercice : 212-p121

- **Class NOM\_CLASSE : public NOM\_SUPERCLASSE\_1, public NOM\_SUPERCLASSE\_2 {.....};**



# La notion d'héritage en c++

- Un constructeur, comme toute autre fonction membre, doit être intégré à une étiquette de protection. Dans la plupart des cas, les constructeurs sont déclarés dans la section public, ce qui permet à quiconque de créer les objets de cette classe en utilisant ces constructeurs.

## **Exemple d'héritage : classe16-2.cpp**

- L'étiquette de protection private utilisée par la classe de base ne peut pas être modifiée par les classes dérivées. En d'autres termes, par l'héritage on ne peut qu'augmenter le niveau de sécurité, jamais le diminuer. Les données privées ne sont manipulables que par les fonctions membres de la classe qui les a définies.

## **Exemple d'héritage : classe16-3.cpp, classe16-4.cpp**

- Si la fonction d'une classe masque la fonction de la classe de base, c'est la fonction membre de la classe qui a la priorité.
- **Exemple : chap16/exemple-1.cpp, exemple-2.cpp**

# CIRCUIT

v\_num\_plus  
v\_num\_moins  
v\_analog\_plus  
numero

Tension d'alim. numerique pos.  
Tension d'alim. numerique neg.  
Tension d'alim. analogique pos.  
Numero d'une composante

## NUMERIQUE

Fan\_out Nb. max sorties

## NAND 2

In\_1 Entrée  
in\_2 entrée  
out\_1 sortie

## NOR 2

In\_1 Entrée  
in\_2 entrée  
out\_1 sortie

## AMPLIFICATEUR

gain Gain de l'ampli.

## ANALOGIQUE

Puissance\_max  
type

Puissance max dissipée  
type de composante  
(active ou passive)

## RESISTANCE

Valeur  
tolerance Valeur  
resistance  
en %

## CONDENSATEUR

Capacité  
v\_claquage Valeur du  
condensateur  
en volts

# Les testes utilisant les prédicats numériques

- Un Les fonctions retournant une valeur représentant « vrai » ou faux sont appelés prédicats. C++ offre des prédicats intrinsèques permettant de vérifier la relation entre une paire de nombres.
- **Prédicats intrinsèques : Utilité**
- `==` Est-ce que les deux nombres sont égaux
- `!=` Est-ce que les deux nombres ne sont pas égaux
- `>` Est ce que le premier nombre est strictement supérieur au second
- `<` Est ce que le premier nombre est strictement inférieur au second
- `>=` Est ce que le premier nombre est supérieur ou égal au second
- `<=` Est ce que le premier nombre est inférieur ou égal au second

# Les testes utilisant les prédicats numériques

- La valeur de l'expression `6 == 3`, est `0`, i.e « faux » en C/C++.
- La valeur de l'expression `6 != 3`, est `1`, i.e « vrai » en C/C++.
- L'opérateur « `!` » seul représente l'opérateur de négation, la valeur `!0` est `1` et la valeur `!1` est `0`, ou `!(6==3)` est `1` tandis que la valeur de `!(6!=3)` est `0`.
- C++ interprète tout entier différent de 0 comme étant vrai
- `!` Change tout entier différent de 0 en 0.
- Pour comparer deux nombres de type différent il faut utiliser un casting. Comparaison `int i` et `double d` : `(float) i == d`.
- Exercice :
  - Écrire un programme qui accepte un nombre et qui affiche 1 si le nombre est plus petit que 100 et 0 s'il est plus grand ou égal à 100.
  - Exercice : `exo18-1.cpp` (exercice 232-p.129)

# Les énoncés conditionnels

- Une expression booléenne est une expression qui produit la valeur **vrai** ou **faux**. En C++ cela signifie que l'expression produit **0** (associé à **faux**) ou tout entier **différent de 0** (associé à **vrai**).
- Un énoncé if comprend une expression booléenne contenue entre des parenthèses, suivie d'un énoncé associé:  
**if (expression booléenne ) énoncé associé**  
**Exemple : exam19-1.cpp**
- L'énoncé **if-else** : **if (expression booléenne ) énoncé associé si vrai**  
**else énoncé associé si faux**
- Si on veut exécuter plus d'un énoncé lorsque l'expression booléenne d'un énoncé **if** ou **if-else** est **vraie** ou **fausse**, il suffit de créer un énoncé associé dont les énoncés sont compris entre crochets.
  - **Exemples : exam19-2.cpp exam19-3.cpp**

# Les énoncés conditionnels

- **Opérateur conditionnel** (**? :**) du C++ : cet opérateur permet de **calculer une valeur** à partir de deux expressions produisant chacune une valeur. (opérateur ternaire).
- (**Expression booléenne** **?** **Expression si vraie** **:** **expression si faux**)
- L'opérateur conditionnel permet de produire un résultat en fonction de la valeur d'un prédicat
  - **Exemple** : **Exam19-4.cpp**

# La combinaison logique d'expressions booléennes

- Comment combiner des expressions booléennes simples pour former des expressions plus complexes.
- L'opérateur logique **ET**, symbolisé par **&&**, retourne la valeur 1 si ses deux opérandes ont une valeur entière différente de 0 et retourne 0 autrement.
- L'opérateur logique **OU**, symbolisé par **||**, retourne la valeur 1 si au moins une de ses opérandes a une valeur entière différente de 0 et retourne 0 autrement.
- Les opérateurs **&&** et **||** ont une priorité inférieure aux prédicats logiques ( <, >, == ...).

**Exemple : exam20-1.cpp**

- L'instruction **switch** : permet de tester le contenu d'une variable par rapport à une série de valeurs (remplace une suite d'instructions **if**)

# La combinaison logique d'expressions booléennes

- **Syntaxe de l'instruction switch :**

```
switch (variable)
{
    case valeur1 : // Instructions break ;
    case  valeur2 :
        // instructions
    break ;

    default :           // instructions
```

L'expression à évaluer est indiquée entre parenthèses et peut être de type **short**, **int**, **long** et **char**.

- Un énoncé Deux mots clés sont associés à l'utilisation du switch : **break** et **default**. Le premier indique que l'on doit quitter le test, le second précise le traitement à effectuer dans le cas où l'expression n'est égale à aucune valeur répertoriée par les directives de case. L'instruction default est optionnelle et doit être placée à la fin du switch.



# Les itérations en C++

- Une Les instructions **while** , **for** et **do while** permettent de répéter une séquence un certain nombre de fois.
- While (expression booléenne) énoncé associé  
while (condition): {  
    **code de la boucle**  
}
- **Exemples** : exam21-1.cpp && exam21-2.cpp
- **while (n != 0 ) n = n-1    [ while (n) n = n-1 ]**
- Boucle for  
    **for (expression1; expression2; expression3) {**  
        **code de la boucle**  
    **}**
- La boucle **for** permet de réaliser un traitement un certain nombre de fois.
- **Exemple** : **Exam21-3.cpp**

# Les itérations en C++

- Le **do while** est une autre forme de boucle de type tant que. La seule différence avec le while réside dans le fait qu'avec l'instruction do while on commence par exécuter le corps de la boucle, et seulement ensuite on teste la condition. Le corps de la boucle est exécutée au moins une fois.

```
do {  
    corps de la boucle  
} while (conditions);
```

- La condition est testée à la fin de chaque passage.
- Les instructions de saut :

**break, continue, goto et return**

**La permettent le transfert du contrôle d'exécution à l'intérieur d'une même fonction.**

# Les opérateurs en C++

- Les opérateurs arithmétiques

Opérateurs	Opérateurs	Opérateurs
+	Plus unaire	+X
-	Moins unaire	-X
*	Multiplication	X * y
/	Division	X / y
%	Reste de la division (modulo)	X % y
+	Plus binaire	X + y
-	Moins binaire	X - y

- Incrément et décrétement :

**a += 1;**      // augmente de 1 la valeur de a

**a -= 1;**      // diminue de 1 la valeur de a

# Les opérateurs en C++

- Opérateur ++ : x++ (suffixe) , ++x (préfixe)
- Opérateur -- : x-- (suffixe) , --x (préfixe)
- a++ ou ++a augmente la valeur de a d'une unité
- a-- ou --a diminue la valeur de a d'une unité
- b= ++a; //a est incrémenté **avant** l'affectation à b
- b= a++; //a est incrémenté **après** l'affectation à b
- B= a; //a est décrémenté **avant** l'affectation à b
- b= a; //a est décrémenté **après** l'affectation à b

# Traitement des données contenues dans des fichiers

- On peut tirer profit des instructions **while** et **for** pour traiter des données contenues dans des fichiers
- Entrée au clavier
- While (cin >> variable\_1>>variable\_2>>variable\_3...)
  - **Exemples** : [exam22-1.cpp](#) [exam22-2.cpp](#)
- **Les tableaux de nombres** : Ensemble de variables de même type (int, long, char, etc..).
- Type NomTableau[nombre élément]
  - int Tabentier[5]; float Revenus[12];
- Tabentier[1]=10; //initialise le deuxième élément a 10
  - int NombreJours[12] = {31,28,31,30,31,30,31,31,30,31,30,31}
  - int NombreJours[] = {31,28,31,30,31,30,31,31,30,31,30,31}

Tableaux : int matrice [2][3]

([exam22-3.cpp](#))

# Traitement des données contenues dans des fichiers

- On peut utiliser l'utilitaire de redirection « < » pour lire les données du fichier nombre.dat. `exam224.cpp`

- **Exemple :** `exam22-4.cpp`

- `exam224 < nombre.dat`
- `cat nombre.dat | exam224`

# Tableaux d'objets

- Les tableaux peuvent être utilisés pour stocker des objets appartenant à des classes.
  - **Exemple :** `exam24-1.cpp`
- La manipulation des membres d'un objet faisant partie d'un tableau se fait de la même manière que pour un objet seul. Il suffit simplement d'identifier l'élément du tableau pour lequel on désire accéder au nombre via l'opérateur d'accès aux membres « . »
- Le constructeur par défaut est appelé pour chaque élément d'un tableau d'objets.

# Création de flots de données d'entrée et sortie

- Accès direct aux données contenues dans des fichiers, sans faire appel à l'utilitaire de direction.
- **Clavier >---- flot cin---- programme ----- flot cout----> écran**
- Pour lire des données contenues dans un fichier, il suffit de créer un flot de données d'entrée au travers duquel les données transitent du fichier vers le programme (`ifstream`) (lecture).
- Pour écrire des données dans un fichier, il suffit de créer un flot de données de sortie au travers duquel les données transitent du programme vers le fichier (`ofstream`) (écriture).
- La création de flux de données `#include <fstream .h>`.
- **Pour ouvrir un flot de données d'entrée :**
  - **Syntaxe** : `ifstream nom_flot(« nom_du_fichier_d'entrée »,ios::in)`



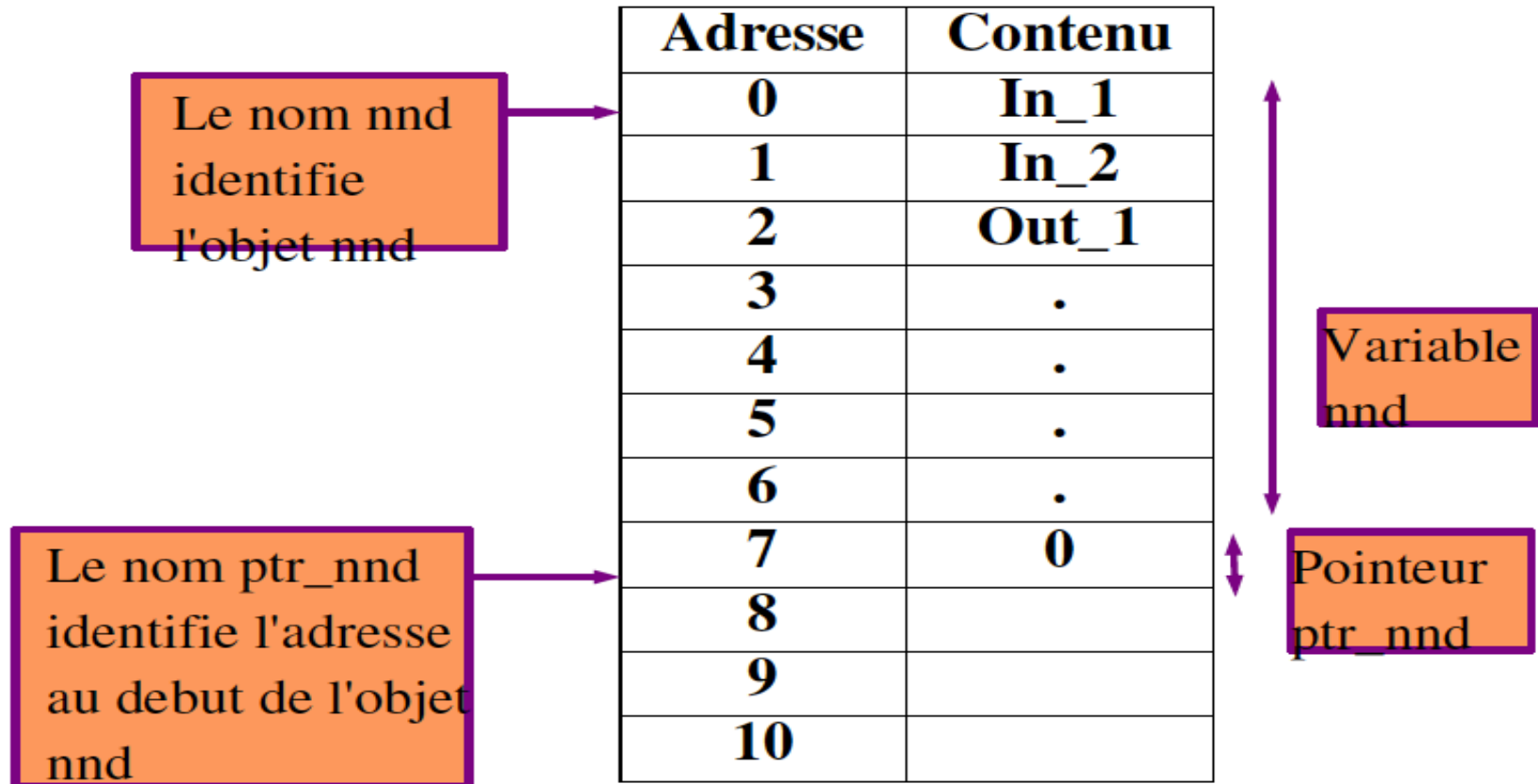
# Création de flots de données d'entrée et sortie

- Pour ouvrir un flot de données de sortie :
  - **Syntaxe** : `ifstream nom_flot(« nom_du_fichier_de_sortie »,ios::out)`
- Pour accéder aux données d'entrée :
  - **Syntaxe** : `nom_flot >> variables d'entrée;`
- Pour ouvrir un flot de données de sortie :
  - **Syntaxe** : `ofstream nom_flot(« nom_du_fichier_de_sortie »,ios::out)`
- Pour stocker des données dans le flot de sortie :
  - **syntaxe** : `nom_flot << données;`
  - Exemple : **exam25-1.cpp exam25-2.cpp**
- Pour fermer les flots de données : `nom_flot.close();`
  - Exemple : **exam25-3.cpp**

# Création d'objets au run-time

- Un pointeur est un espace mémoire qui contient l'adresse d'un objet. Le nom d'un pointeur identifie l'espace mémoire où est stockée l'adresse de l'objet
- Un pointeur est une variable qui contient l'adresse d'une autre variable. Accès indirectement au contenu d'une variable en passant par son adresse et manipuler les contenus de zone mémoire sans nom. Un pointeur est une donnée qui enregistre ou représente l'adresse d'une autre donnée : le pointeur renvoie ou « **point** » par son contenu vers la donnée dont il contient l'adresse, i.e vers la zone mémoire où elle se trouve.
- **Syntaxe** : `type *nompointeur;`
- Type désigne un type de variable (int, char, etc.) mais aussi aux types que nous avons créés : `typedef`, `struct`, `class`, `NAND_2`, etc.
- Un pointeur doit être initialisé de la même manière que toute variable. On doit affecter au pointeur l'adresse d'une variable d'un type adéquat.
  - **Exemple** : `exam26-1.cpp`

# Mémoire de l'ordinateur



- L'opérateur new : permet d'allouer de l'espace mémoire pour stocker un objet au runtime.

```
Double *ptr_double;  
ptr_double = new double;
```

# Mémoire de l'ordinateur

- L'opérateur new alloue de la mémoire pour stocker un type (double, int, etc.) et retourne l'adresse de cet espace de mémoire.

```
NAND_2 *ptr_nand_2;  
ptr_nand_2 = new NAND_2;
```

- l'opérateur new réserve l'espace pour stocker l'objet de type NAND\_2, l'adresse de cet espace est stockée dans le pointeur ptr\_nand\_2, le constructeur par défaut de la classe est appelé.

- \*ptr\_nand\_2 est un pointeur (adresse) par contre ptr\_nand\_2 (espace mémoire) est un objet de type NAND\_2. Pour accéder aux membres d'un objet via le pointeur, il suffit d'utiliser : (\*ptr\_nand\_2).ecrire\_in\_1(1); ou en utilisant l'opérateur de pointeurs aux classes « -> » ptr\_nand\_2 -> écrire\_in\_1(1);

- Pour assigner une valeur à une variable membre d'un objet :

```
(*ptr_nand_2).variable_membre = valeur;
```

ou

```
ptr_nand_2 -> variable_membre = valeur
```

# Stockage des pointeurs aux objets d'une classe

- La déclaration des tableaux occupe beaucoup d'espace mémoire, si les éléments du tableau sont des objets, l'espace mémoire occupé est encore plus significatif.
- Par contre un pointeur à un objet occupe souvent moins d'espace que l'objet lui-même. Au lieu de déclarer un tableau d'objets il est plus judicieux de déclarer un tableau de pointeurs (**exam25-3.cpp**)
  - **Exemple : exam27-1.cpp (2,3)**, on réserve un tableau de 100 objets pour n'utiliser que 5, de l'espace réservé inutilement.
- Une solution plus judicieuse consiste à réserver de l'espace pour des pointeurs à des objets et d'allouer dynamiquement au run-time de l'espace à l'aide de l'opérateur new. (**exam27-2.cpp**)

# Stockage des pointeurs aux objets d'une classe

- La L'opérateur **delete** permet de libérer de l'espace mémoire préalablement alloué avec **new**.

```
delete nom_pointeur;  
delete nom_tableau[indice];  
delete [] nom_tableau;
```

- Par L'opérateur `sizeof` calcule le nombre d'octets requis pour stocker un objet.  
(exam27-3.cpp)

# Récupération de la mémoire

- La récupération de la mémoire (désallocation) grâce à l'opérateur **delete** et aux fonctions membres appelées **destructeurs**
- **Exemple :** [chap47/exam37-2.cpp](#)
- Quand un programme produit de la mémoire détritue, on dit qu'il possède des fuites (leaks).
- Quand on désire colmater une fuite de mémoire, il suffit d'adopter l'une des syntaxes suivantes :  
  

```
delete nom_pointeur;  
delete nom_tableau[indice];  
delete [ ] nom_tableau;
```
- Un destructeur est une fonction membre qui est appelée lorsque la mémoire allouée à un objet est récupérée par le programme.

# Récupération de la mémoire

- **Syntaxe :**

```
~constructeur  
~classe(); ~NAND_2();  
virtual ~NAND_2();
```

- Il est nécessaire que les destructeurs récupèrent de la mémoire additionnelle ( allouée dynamiquement et accédée par des variables membres de type pointeur contenue dans des objets) à celle réservée pour l'objet en appliquant l'opérateur **delete** aux variables membres (globales) qui sont des pointeurs à des données qui ont été allouées dynamiquement lors de la création d'un objet.
- **Exemple :** [chap47-1/exam37-2.cpp](#)



# Introduction aux fonctions virtuelles

- Étudions le programme suivant : [exam28-1.cpp](#)
- Ce programme permet de lire le fichier ci-après :  
**1 1 1 2 0 1 1 1 0 : type\_porte entrée\_1 entrée\_2 ...**
- Le programme exam28-1.cpp ne peut traiter qu'un nombre de cas limité. On ne peut créer un vecteur du même type contenant l'information sur les deux types de portes. On doit déclarer deux vecteurs de pointeurs :  
**NAND\_2 \*nand\_pointeur[100];**  
**NOR\_2 \*nor\_pointeur[100];**  
cette méthode est peut élégante
- Or la classe NAND\_2 et NOR\_2 héritent de la classe NUMERIQUE : Lorsqu'on déclare un tableau d'objets d'une classe donnée, les pointeurs peuvent contenir l'adresse d'un objet de la classe à laquelle appartient l'objet mais également l'adresse de tout objet appartenant à une sous-classe..

# Introduction aux fonctions virtuelles

- On peut créer un tableau de pointeurs de type NUMERIQUE :

**NUMERIQUE \*num\_circuit[100];**

- On peut par la suite allouer de l'espace pour un objet NAND\_2 et stocker le pointeur à cet espace dans l'élément **num\_circuit[0];**

num\_circuit[0] = new NAND\_2; **(exam28-2.cpp)**

- Un pointeur qui est défini pour pointer à un objet d'une superclasse peut aussi pointer à un objet appartenant à une sous-classe de cette classe.
- Modifions la définition des classes circuit, NUMERIQUE, NAND\_2 et NOR\_2 pour afficher le type de composants au moment de sa création. Il faut ajouter une fonction membre responsable d'afficher le type de composante à laquelle la classe appartient. **exam28-3.cpp**
- Si on définit un tableau de pointeurs à des objets d'une classe mère et que ces pointeurs pointent en réalité à des objets d'une sous-classe, il faut définir des fonctions membres virtuelles si on désire appeler ces fonctions membres.

**Une fonction membre virtuelle définie dans une classe est automatiquement considérée comme étant virtuelle dans toutes les sous-classes**

# Les énoncés conditionnels multiples

- **If** pour décider d'une action en fonction du résultat d'un prédicat logique
- **Switch** exécution d'une séquence d'énoncés en fonction du résultat d'une expression produisant une valeur entière (int)
- Syntaxe de l'énoncé switch :

```
switch (expression produisant une valeur entière ) {  
    case constante entière 1 : énoncé 1 break;  
    case constante entière 2 : énoncé 2 break;  
    case constante entière 3 : énoncé 3 break;  
    case constante entière 4 : énoncé 4 break;  
    ....  
default : énoncé par défaut;  
}
```

**exam29-1.cpp**

# Les énumération en C++

- pour L'énoncé d'énumération enum permet d'associer un code numérique à un mnémonique (exemple switch (type)) : `enum{nand_2_code, nor_2-code};`
- L'instruction enum assigne alors la valeur 0 à nand\_2\_code et la valeur 1 pour nor\_2\_code. `enum{nand_2_code=1, nor_2-code};`  
(exemple: exam30-1.cpp)
- On peut généraliser la syntaxe de l'énoncé enum :  
`enum {code_a, code_b, code_c, code_d=5, code_e}`  
`code_a=0, code_b=1, code_c=2, code_d=5, code_e=6`
- On peut définir aussi des types de données d'énumération qui permettent de définir des variables d'énumération  
`enum type_enum {code_a, code_b, code_c, code_d, code_e}`
- Les énumérations facilitent la lecture des programmes en C++ en associant des constantes à des valeurs numériques entières.

(exemple : exam30-2.cpp)

## Fonctions membres appelant d'autres fonctions membres

```
NAND_2 () : NUMERIQUE(5)    {
```

```
    - in_1=0;
```

```
    - in_2=0;
```

```
    out_1=set_out_1(); }  
}
```

- Appel de la fonction set\_out() (return 1-(in\_1\*in\_2))
- le constructeur par défaut NAND\_2 () est une fonction membre de la classe NAND\_2. La fonction set\_out\_1 est une fonction de la même classe. Nous n'avons pas eu besoin de passer par l'opérateur d'accès au membre d'une classe « . »
- L'objet par défaut pour lequel set\_out\_1 est appelé est le même que celui pour lequel NAND\_2() est appelé, soit l'objet courant. Cet objet est un argument implicite. L'opérateur d'accès aux membres est lui aussi implicite dans l'appel de la fonction membre set\_out\_1 par NAND\_2().
- L'argument implicite est le pointeur **this**. Chaque fonction membre d'une classe possède un argument implicite appelé **this** dont la valeur est un pointeur à l'objet correspondant à la classe de cette fonction membre.

# Fonctions membres appelant d'autres fonctions membres

- Le pointeur **this** pointe simplement à l'adresse de l'objet NAND\_2.
- La syntaxe suivante est aussi valable :

```
NAND_2 () : NUMERIQUE(5)          {  
    in_1=0; in_2=0;  
    out_1=(*this).set_out_1();      } // set_out_1 est appelée pour l'objet *this ,  
    l'objet *this est implicitement l'objet NAND_2
```

- ```
NAND_2() : NUMERIQUE(5)          {  
    in_1=0;  in_2=0;  
    out_1= this -> set_out_1(); } (exam31-2.cpp)
```

- De manière implicite, un argument est toujours passé lors d'un appel d'une fonction. Il s'agit d'un pointeur que l'on nomme **this**, et qui permet de désigner l'objet effectuant l'appel de la fonction.

• Exemples : [exemple32-10.cpp](#), [exemple32-11.cpp](#)

# Variables privées (private) et protégées (protected)

- Pour éviter de corrompre les données ou d'utiliser des fonctions incorrectement on utilise les mots réservés **private** et **protected**.
- Pour limiter l'accès aux variables membres et fonctions membres d'une classe seulement aux fonctions définies dans la même classe, on place ces variables et fonctions membres dans la partie **private** de la classe.
- Pour limiter l'accès aux variables et fonctions membres d'une classe seulement aux fonctions définies dans la même classe ou dans une sous-classe de la classe, il suffit de placer ces variables et fonctions membres dans la partie **protected** de la classe.
- Pour ne pas limiter l'accès aux variables et fonctions membres d'une classe, il suffit de placer ces variables et fonctions dans la partie **public** de la classe.
- Pour éviter une modification des variables membres d'une classe par une fonction d'une classe dérivée tout en permettant la consultation du contenu de ces variables, il suffit de définir les variables membres dans la partie **private** de la classe et de placer les fonctions membres de lecture dans la partie **protected**.

# Fonctions qui retournent des chaînes de caractères

- Les chaînes de caractères sont stockées dans un tableau à une dimension et se terminent par le caractère nul.
- Pour déclarer une variable dont la valeur est une chaîne de caractères il suffit d'adopter la syntaxe suivante :

**char \*var\_str = « chaine »**

- Pour qu'une fonction retourne une chaîne de caractères, il suffit d'adopter la syntaxe suivante :

**char \*nom\_fonction...**

- **Exemple :** [chap35/exam31-2.cpp](#)



# Passage des paramètres par référence

- Transmission par valeur, la valeur passée comme paramètre est d'abord copiée dans une variable temporaire (formelle), cette variable est utilisée par la fonction puis elle est détruite au retour de la fonction. La fonction ne travaille en réalité que sur une copie de la variable.
- Passage par variable : Il n'y a pas de copie ni de variable locale(formelle). Toute modification du paramètre dans la fonction appelée entraîne la modification de la variable passée en paramètre.
- En plus des pointeurs, le C++ permet de créer des références. Les références sont des synonymes d'identificateurs. Elles permettent de manipuler une variable sous un autre nom que celui sous lequel cette dernière a été déclarée. Il est recommandé de passer par référence tous les paramètres.

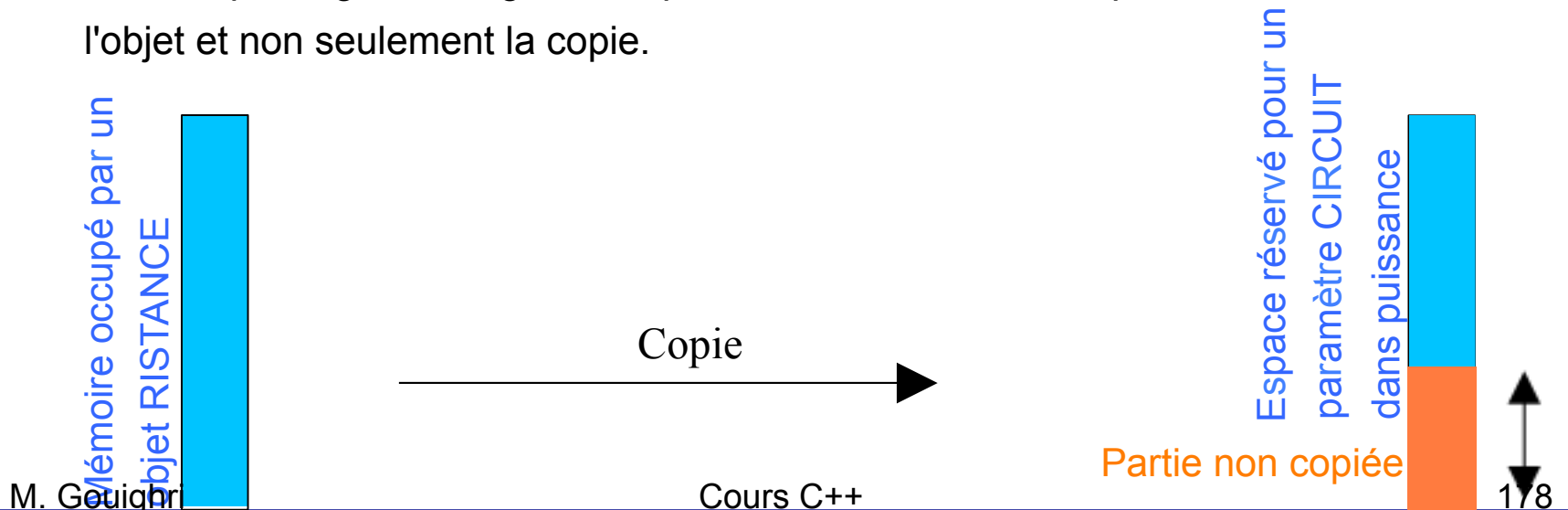
**type &référence = identificateur; (exam36-1.cpp)**

- Pour Si on veut passer un objet à une fonction membre on utilise les opérateurs d'accès aux membres « > » et « . », par contre si on veut passer plusieurs objets à une fonction, un seul peut passer via les opérateurs d'accès aux membres.
- Transmission par adresse ou par référence

**type\_de\_retour nom\_fonction(..., type\_arg & nom\_parametre,...) {}**

# Passage des paramètres par référence

- **Exemple :** exam36-2.cpp et exam36-3.cpp
- Le passage par référence permet de nous assurer que toutes les parties d'un objet sont accessibles à la fonction
- Le passage par référence est plus rapide car seule l'adresse passe en argument à la fonction est copiée dans le paramètre.
- Lors du passage des arguments par référence, la fonction peut modifier directement l'objet et non seulement la copie.



# La surdéfinition de l'opérateur d'insertion <<

- Si on veut surdéfinir un opérateur binaire (i.e à deux opérandes), on adopte le patron suivant :

```
type_retour opérateur      symbole
( type_gauche nom, type_droite_nom) { énoncé }
```

- Pour surdéfinir l'opérateur d'insertion << , on adopte le patron suivant :

```
ostream&      operator symbole ( ostream& output_stream ,
type_droite nom) {
                énoncé                                     return
                output_stream;
            }
```

**Exemple :** chap37/exam37-1.cpp et exam37-2.cpp

# La surdéfinition des opérateurs unaires

- La surcharge des opérateurs ne permet pas de modifier la signification des opérateurs dans le cas des types primaires du C++ (short, int, etc..)
- La surdéfinition d'un opérateur permet de l'adapter au contexte auquel il est exposé puisqu'il réagit différemment, selon la façon dont il est programmé, en fonction des arguments qui lui sont fournis.
- La surcharge d'un opérateur s'applique à un type particulier de données. Cela signifie qu'on doit définir cette surcharge pour toutes les classes concernées, mais aussi tenir compte du type d'argument (pointeur, référence, valeur) qui sera appliqué à cet opérateur. Il est souvent nécessaire de redéfinir deux surcharges d'opérateurs ( par pointeur et par référence).
- Un opérateur est surdéfini en écrivant une fonction incluant un entête et un corps comme toute fonction standard en C++. Le nom de la fonction est :

**type\_de\_retour opérateur symbole\_de\_l'opérateur**

# La surdéfinition des opérateurs unaires

- La surdéfinition d'opérateurs binaires comme fonctions membres d'une classe comprend un argument implicite (opérande de gauche), un argument ordinaire passé par référence (opérande de droite) et une valeur de retour. Le passage de paramètres par référence doit être effectué en utilisant la notion de référence constante (const) pour éviter le risque que la fonction modifie l'argument.

**Exemple :** chap40/exam402.cpp et exam401.cpp

- Un opérateur unaire d'une classe peut être surdéfini via une fonction membre sans argument ou via une fonction ordinaire avec un argument correspondant à un objet de la classe ou une référence à un objet de la classe.
- **La surdéfinition d'un opérateur unaire comme fonction membre d'une classe suit la syntaxe suivante:** `type_valeur_retour operator symbole()`  
`{ énoncé ... return valeur_retour; }`

**Exemple :** exam39-2.cpp

Combinaison des résistances et condensateurs : exam40-3.cpp

## La conception d'un programme à partir de plusieurs fichiers

- Les fonctions membres sont déclarées à l'intérieur des classes, ce qui limite le principe de lisibilité d'un programme.

### Les classes \*.h et fonctions \*.C

- Les classes contiennent que la déclaration des fonctions membres avec la valeur de retour, le nom de la fonction et la liste des paramètres avec leur type.
- Les fichiers d'extension .C contiennent le code des fonctions membres associées à une classe spécifique, grâce à l'opérateur d'évaluation de portée ( :: ).
- Les fichiers d'extension .C sont compilés pour produire des fichiers objets d'extension .o  
`c++ -c analogique.C, c++ c resistance.C`
- `C++ exam403.cpp -o exam403 résistance.o circuit.o analogique.o`
- L'édition des liens entre le programme principal et les différents modules se fait lors de la compilation du programme en donnant les modules avec lesquels il doit être lié

# L'utilitaire make notions élémentaires

- Le principe de visibilité : un programme se divise en plusieurs modules : classes (\*.h), fonctions (\*.c), objets (\*.o) qui dépendent les uns des autres, toutes modifications se répercutent sur l'ensemble des modules.
- L'utilitaire **make** traite les dépendances et plus dans un fichier appelé **makefile** résident dans le répertoire courant. Les dépendances expriment les relations existantes entre un fichier cible et les fichiers prérequis desquels il dépend.

**Cible : liste des                      prérequis                      ligne de dépendance**  
**(tabulation) commandes de construction**

**Exemple : chap42**

- L'utilitaire make notions avancées

**Exemple : chap43**

## La lecture et l'écriture de chaînes de caractères dans un fichier

- Pour lire une chaîne de caractères dans un fichier, il faut d'abord créer un tableau dans lequel elle sera stockée temporairement.

**char input\_chaine[100];**

- Le nom de la chaîne devient une constante dont la valeur est l'adresse du premier élément du tableau : pointeur.
- Le nom de la chaîne est donc le nom d'un pointeur qui pointe au début de la chaîne.
- On ne peut réassigner quelque chose au pointeur d'une chaîne parce que C++ considère cet objet comme une constante.
- Pour lire une chaîne de caractères dans un fichier, il faut d'abord créer un tableau assez grand pour stocker la plus longue chaîne qu'il contient.
- Pour lire la chaîne de caractères il suffit d'adopter la syntaxe suivante :

**cin >> nom\_chaine**

- L'opération d'extraction >> lit une chaîne de caractères jusqu'à ce qu'il rencontre un



# Les testes sur les chaînes de caractères

- Pour lire une chaîne de caractères dans un fichier, il suffit de déclarer un vecteur de caractères suffisamment long pour stocker la plus grande chaîne de caractères attendues. Ensuite, l'opérateur d'extraction >>, qui reçoit le nom de la chaîne en argument, lit les caractères jusqu'au premier caractères blanc (espace, retour de chariot, nouvelle ligne ou tabulation).
- Le nom d'une chaîne de caractères est un pointeur à son premier élément.
- La chaîne se termine par le caractère nul **\0**.
- Pour accéder à un caractère d'une chaîne de caractères, il suffit simplement d'adresser l'élément du vecteur servant à stocker cette chaîne.
- L'origine du vecteur est 0, c'est à dire que le premier élément est situé à l'indice 0.

**Exemple :** chap45/ exam37-2.cpp

# Stockage des chaînes de caractères dans les objets

- Lorsqu'on désire créer un tableau au run-time pour stocker une chaîne de caractères, il suffit d'adopter la syntaxe suivante :

**variable de type pointeur de caractère = new char[nb caractères + 1]**

- Si on désire utiliser les fonctions de traitement de chaînes de caractères du C++, il suffit d'ajouter la ligne :

**#include <string.h>**

- Pour connaître la longueur d'une chaîne de caractères en excluant le caractère de fin de chaînes, il suffit d'adopter la syntaxe suivante : **strlen(nom\_de\_la\_chaine)**
- Pour copier une chaîne dans une autre, il suffit d'adopter la syntaxe suivante.

**strcpy(nom\_chaine\_destination, nom\_chaine\_source)**

**Exemple :** chap46/ exam37-2.cpp

# Récupération de la mémoire

- La récupération de la mémoire (désallocation) grâce à l'opérateur **delete** et aux fonctions membres appelées **destructeurs**
- **Exemple :** [chap47/exam37-2.cpp](#)
- Quand un programme produit de la mémoire détritue, on dit qu'il possède des fuites (leaks).
- Quand on désire colmater une fuite de mémoire, il suffit d'adopter l'une des syntaxes suivantes :  

```
delete nom_pointeur;  
delete nom_tableau[indice];  
delete [ ] nom_tableau;
```
- Un destructeur est une fonction membre qui est appelée lorsque la mémoire allouée à un objet est récupérée par le programme.

# Récupération de la mémoire

- **Syntaxe :**

```
~constructeur  
~classe(); ~NAND_2();  
virtual ~NAND_2();
```

- Il est nécessaire que les destructeurs récupèrent de la mémoire additionnelle ( allouée dynamiquement et accédée par des variables membres de type pointeur contenue dans des objets) à celle réservée pour l'objet en appliquant l'opérateur **delete** aux variables membres (globales) qui sont des pointeurs à des données qui ont été allouées dynamiquement lors de la création d'un objet.
- **Exemple :** [chap47-1/exam37-2.cpp](#)

# Le constructeur par recopie (copy constructor)

- En général, les objets passés en argument par valeur à des fonctions peuvent conduire à des problèmes subtils de désallocation de mémoire. Il faut éviter de passer des objets par valeur à des fonctions. Un moyen de s'assurer que les objets sont passés par référence est de définir une fonction membre appelée constructeur par recopie :

**nom\_classe(nom\_classe &);**

**nom\_constructeur(const nom\_classe &);** et de la placer

dans la partie private de la classe. Ainsi, le compilateur signalera une erreur lorsque le programmeur tentera de passer un objet par valeur à une référence.

- Lorsque le C++ copie un argument dans un paramètre, il le fait en appelant une fonction membre appelée constructeur par recopie, s'il n'est pas défini le C++ adopte une par défaut. Le constructeur par recopie défaut copie la valeur des pointeurs de l'argument dans les pointeurs du paramètre mais ne copie pas le contenu de la mémoire à laquelle pointe le pointeur de l'argument.

# Le constructeur par recopie (copy constructor)

- Lorsque il n'est pas possible de passer les objets uniquement par référence pour éviter la recopie dans le paramètre, il faut définir une fonction membre appelée constructeur par recopie et veiller à ce que les variables membres de l'argument soient copiées correctement dans le paramètre, incluant les pointeurs et ce à quoi ils pointent.
- L'argument du constructeur par recopie est passé par référence, **nom\_constructeur(const nom\_classe &);** ici on choisit une référence à une constante pour s'assurer que l'argument ne sera pas modifié par la fonction.
- **Exemple :** chap48/exam48-1.cpp

# Les classes et fonctions amies

- Le principe d'encapsulation interdit à une fonction membre d'une classe d'avoir accès directement aux variables membres private d'une autre classe. Un moyen d'accéder aux variables private d'une classe est d'utiliser les fonctions membres de lecture et d'écriture.
- Un autre moyen est de passer par les fonctions amies de la classe.
- Lors de la création d'une classe on peut spécifier qu'une ou plusieurs fonctions sont amies de la classe
- La surdéfinition des opérateurs peut être implantées via les fonctions amies (**voir exemple p.397**).
- **Syntaxe :** `friend type function( classe &variable, classe &variable,...)`

# Les classes et fonctions amies

```
Class A {  
    private:  
        int xa;  
    public:  
        A() {}  
        int f_membre_a();  
        friend int funct(A &a, B &b); /* funct amie de la classe A  
et retourne un entier */  
};  
  
Class B {  
    private:  
        int xb;  
    public:  
        B() {}  
        int f_membre_b();  
};
```



# Les classes et fonctions amies

```
Class A {  
    private:  
        int xa;  
    public:  
        A() {}  
        int f_membre_a();  
        friend int funct(A &a, B &b); /* funct amie de la classe A  
et retourne un entier */  
};  
  
Class B {  
    private:  
        int xb;  
    public:  
        B() {}  
        int f_membre_b();  
        friend int funct(A &a, B &b); /* funct amie de la classe B  
et retourne un entier */  
};
```

# Les classes et fonctions amies

```
Class A {  
    private:  
        int xa;  
    public:  
        A() {}  
        int f_membre_a();  
        friend int funct(A &a, B &b); // funct amie de la classe A  
};  
  
Class B {  
    private:  
        int xb;  
    public:  
        B() {}  
        int f_membre_b();  
        friend int funct(A &a, B &b); // funct amie de la classe B  
        friend int A::f_membre_a(); //f_membre_a() de A amie de B  
};
```

# Les classes et fonctions amies

```
Class A {  
    private:  
        int xa;  
    public:  
        A() {}  
        int f_membre_a();  
        friend int funct(A &a, B &b); // funct amie de la classe A  
};  
  
Class B {  
    private:  
        int xb;  
    public:  
        B() {}  
        int f_membre_b();  
        friend int funct(A &a, B &b); // funct amie de la classe B  
        friend class A; // toutes les fonctions de A sont amies de B  
};
```

## La réutilisation des fonctions : la notion de fonction générique (template)

- La surdéfinition des fonctions peut être remplacée par le mécanisme appelé **fonction générique**, **patron** ou **template** qui permet de définir une seule fonction (patron) qui peut être appliquée à plusieurs types de données.
- **Syntaxe** : `template <class T, class U,..., class P> U fct(T a, V *b, Uc) { U interne; interne = a + (*b); return interne;}`
- `template <class T, class U,..., class P> retour nom_fonction(paramètres) {.....}`
- Un patron de fonction peut être surdefini
- Un patron de fonction peut être spécialisé
- **Exemple** : `exam50-1.cpp`

## La réutilisation des classes : la notion de patrons de classes (template)

- On peut réutiliser des classes en créant des patrons de classe aussi appelés templates .
- Template <class Ttype> class class-name  
    {.....}
- Template <class Ttype1, class Ttype2, class Ttype3,...> class class-name  
    {.....}
- Instanciation, spécialisation : class-name <type> ob;
- Un patron de fonction peut être spécialisé
- **Exemple :** exam51-1.cpp, exam51-2.cpp et exam51-3.cpp