## III.1. Rappels sur le langage C

Langage conçu en 1972 par Dennis Richie et Ken Thompson, le C a été Utilisé pour réécrire le système d'exploitation UNIX.

Normalisation du langage par l'ANSI dans les années 80 (norme reprise par l'ISO en 1990).

En réalité le C est une famille de langages assez proches avec en leur centre la norme ISO (pas toujours totalement respectée).

Et POSIX ? POSIX définit une utilisation normative du C pour les systèmes d'exploitation (C POSIX = C standard associé aux interfaces de programmation POSIX). Tous les systèmes d'exploitation ne prennent pas en charge POSIX (Windows en particulier).

En résumé, on peut donc utiliser le C comme un langage de haut niveau ou comme un langage système.

C est un langage compilé : à partir d'un ou plusieurs fichiers sources (langage C), le compilateur produit un fichier exécutable (langage machine)

Programme C minimaliste :

```
#include <stdio.h>
int main() {
    printf("Bonjour\n");
    return 0;
}
```

Note : tout programme doit comporter une fonction **main()** : c'est le point d'entrée du programme

Quatre étapes de compilation

1) Traitement par le préprocesseur : transformations purement textuelles ( remplacement de chaînes, inclusion de fichiers sources)

2) Compilation : le fichier généré par le préprocesseur est traduit en assembleur (instructions pour le microprocesseur)

3) Assemblage : transformation du code assembleur en langage binaire. Le fichier obtenu est appelé fichier objet. Normalement, les 3 premières étapes sont effectués dans la foulée, sauf si on spécifie que l'on veut le code intermédiaire (fichier préprocessé, code assembleur)

4) Edition de liens : un programme fait souvent appel à plusieurs fichiers sources (pour des question de clarté ou parce qu'il fait généralement appel à des librairies de fonctions déjà écrites). L'édition de lien permet lier entre eux les différents fichiers objets

Extensions : Fichier source (.c), prétraité par le préprocesseur (.i), assembleur (.s), objet (.o), objet correspondant à une librairie précompilée (.a)

Compilateur C sous UNIX : cc. Exemple : compilateur gcc du projet GNU.

**gcc -c mon_prog.c** : compilation (sans édition de liens) => création du .o

**gcc - o mon_prog mon_prog.o** : édition de liens => création de l'exécutable

**./mon_prog** : exécution du programme

## Constantes

Constante entière : notation décimale, octale (commence par **0**), hexadécimale (commence par **0x**)

Constante réelle : float (suffixe **f** ou **F**), double par défaut (sans suffixe), long double (suffixe **l** ou **L**)

Constante caractère : entre **‘ ’**, caractères spéciaux **‘\a’**, **‘\n’**, **‘\’t’**

Constante chaîne de caractères : entre **" "**

Constante énumérée : **enum modele {cte1, …, cteN}**

```
int nb=5;
char car = 'A';
float reel=1.2f
```

## Types de données simples

Nombre entier : **int** (mot sur la machine utilisée), **short**, **unsigned short**, **long**, **unsigned long**

Nombre flottant : **float**, **double**, **long double**

Caractère : **char** (dépend du codage utilisé par la machine : ASCII le plus souvent)

## Opérateurs

Opérateurs arithmétiques : **+**, **-**, *****, **/**, **%**

Opérateurs relationnels : **>**, **>=**, **<**, **<=**, **==**, **!=**

Opérateurs logiques : **&&**, **||**, **!**

Opérateur d'affectation : **=**

Opérateurs bits à bits : **&**, **|**, **^**, **~**, **>>**, **<<**

| opérateurs | |
|---|---|
| () [] -> . | → |
| ! ~ ++ -- -(unaire) (type) *(indirection) &(adresse) sizeof | ← |
| * / % | → |
| + -(binaire) | → |
| << >> | → |
| < <= > >= | → |
| == != | → |
| &(et bit-à-bit) | → |
| ^ | → |
| \| | → |
| && | → |
| \|\| | → |
| ? : | ← |
| = += -= *= /= %= &= ^= \|= <<= >>= | ← |
| , | → |

Opérateurs d'affectation composée : **+=**, **-=**, ***=**, **/=**, **%=**, **&=** …

Opérateurs d'incrémentation et de décrémentation : **++**, **--**

Séparateur d'expression : **,**

Expression conditionnelle : **exp1 ? exp2 : exp3**

Opérateur de calcul d'adresse **&** et d'indirection *****

Conversion de type (cast) : **(type)objet**

Opérateur sizeof : **sizeof(descripteur de type)**

Priorité des opérateurs : voir tableau

```
int a; int b;
b = ((a = 3), (a + 2));

m = ((a > b) ? a : b);

int i=10;
int j=3;
float reel=(float)i/j
```

## Instructions de branchements et boucles

Branchement conditionnel : **if…else**

```
if (expression1)
     liste instruction1
else if (expression2)
     liste instruction2
...
else
     liste instructionN
```

Branchement multiple : **switch**

```
switch (expression) {
   case constante1:
       liste instructions 1
       break;
   case constante2:
       liste instructions 2
       break;
   ...
   case constanteN:
       liste instructions N
       break;
   default:
       liste instructions
}
```

```
Int i;
…
switch (i) {
   case 1:
       printf (" impair" );
       break;
  case 2:
       printf (" pair" );
       break;
  default
       printf (" ni 1 ni 2" );
}
```

Boucles : **while, do…while, for**

```
while (expression)
   liste instructions
```

```
do
     liste instructions
while (expression)
```

```
for (expression1;expressions2;expression3)
   liste instructions
```

```
for (int i=0;i<9;i++){
    if (i%2==0) continue;
    printf ("%d\t" , i);
}
```

Branchement non conditionnel : **break, continue, goto**

## Tableau

Déclaration d'un tableau : **type nomTab [dimension];**

```
int tab1[4];
int tab2[]={1,2,3,4};
int matrice[2][3]={{1,2,3}{4,5,6}};
```

```
#define N 10
int tab1[N], tab2[N];
…
for (int i=0; i<N; i++)
    tab1[i] = tab2[i];
```

## Chaîne de caractères

C'est un tableau de caractère dont le dernier élément est le caractère de code ASCII nul

```
char ch1[20]={'b', 'o', 'n', 'j', 'o', 'u', 'r', '\0'}; /* tableau de 20 caractères */
char ch2[]={'b', 'o', 'n', 'j', 'o', 'u', 'r', '\0'}; /* tableau de 8 caractères */
char ch3[20]= "bonjour" ; /* déclaration identique à ch1 */
char ch4[]= "bonjour " ; /* déclaration identique à ch2 */
```

Longueur d'une chaîne : **int strlen (const char * chaine);**

## Définir un nouveau type

Il est possible de donner un nouveau nom à un type : **typedef type synonyme;**

```
typedef int LONGUEUR;
....
LONGUEUR l=10;
```

## Structure

Objet regroupant des objets de types potentiellement différents. Cela correspond aux enregistrements dans d'autres langages (Pascal …).

```
struct modele {
    type1 membre1;
    type2 membre2;
    ...
    typeN membreN;
};
```

```
struct point {
    char nom;
    int x;
    int y ;
};
```

```
typedef struct {
    double reel;
    double imaginaire;
} COMPLEXE;

COMPLEXE cplx={1.2, 1.4};
```

Accès aux éléments d'une structure : **.**

```
struct point pt1, pt2={'A', 1, 2};
pt1.nom='E'
pt1.x=5;
pt1.y=7;
```

## Union

Objet pouvant prendre un type au choix parmi un ensemble prédéfini de types. Il contient des objets de types potentiellement différents qui sont susceptibles d'occuper alternativement un même espace mémoire (l'espace prévu est donc celui de l'objet le plus grand). Un seul objet est donc valide à un moment donné.

```
union jour {
      char lettre;
      int numero;
};
…
union jour hier, demain;
hier.lettre = 'J';
hier.numero = 4;
printf("hier = %c \t %d \n",hier.lettre, hier.numero);       // seul le numéro s'affiche bien
```

## Enumération

Définir un type par la liste des valeurs qu'il peut prendre. Les valeurs sont en réalité représentées comme des int (0 pour le premier, 1 pour le second, etc …)

```
enum booleen {faux, vrai};
enum booleen b;
b = vrai;
```

```
typedef enum {pomme, orange, citron} FRUIT;
FRUIT F1;
F1 = pomme;
```

## Pointeur

Un pointeur est une variable qui contient l'adresse d'une autre variable (la variable pointée). Voici la déclaration d'un pointeur : **type *p**

p est un pointeur sur une variable de type **type**

*p est la variable de type **type** pointée par **p**

Pour allouer dynamiquement une adresse à un pointeur : **p= malloc (sizeof(type))**

Cette fonction alloue l'espace désigné en paramètre et retourne son adresse. On préfère souvent convertir cette adresse dans le type souhaité par un cast :

**p= (type*) malloc (sizeof(type))**

```
int x=1, y=2, *p, *q, ;
p=&x        /* p pointe sur x  */
y=*p        /* y=x=1 */
q= (int *)malloc(sizeof(int))   /* réservation d'un espace mémoire de 4 octets pour q */
*q=3;       /* valeur 3 à l'emplacement pointé par q */
```

Arithmétique des pointeurs

**p** est un pointeur sur une variable de type **type**

**p+n** a pour valeur en mémoire **p+ n*sizeof(type)**

**p++** a pour valeur en mémoire **p+ sizeof(type)** etc …

## Pointeurs et tableaux : tableaux dynamiques

Dans la déclaration de tableau vue précédemment (**type nomTab [dimension]**), l'allocation mémoire est réalisée à la compilation. Souvent, la taille du tableau n'est connue qu'à l'exécution. Pour réaliser une allocation dynamique, on déclare :

```
type *nomTab ;
nomTab=(type*) malloc (nb * sizeof(type));
```

ou **nb** est un entier correspondant au nombre d'éléments du tableau et évalué à l'exécution

## Pointeurs et chaînes de caractères

A la place de la déclaration vue précédemment (**char nomCh[dimension]**), on pourra utiliser :

```
char *nomTabCh;
nomTabCh=(char*) malloc (nb * sizeof(char));
```

ou **nb** est un entier correspondant au nombre de caractères de la chaîne et évalué à l'exécution et **sizeof(char)** vaut 1 en code ASCII simple

```
char * ch1, * ch2 , * ch3 ;
char ch4[40] ;
ch1= " je suis un pointeur sur une constante chaîne"  ;
ch2= ch1 ;  /* ch2 pointe sur la constante chaîne précédente */
ch2= ch4       /* ch2 pointe sur le premier élément de ch4 */
ch4= ch1;      /* PLANTE car ch4 est une constante*/
ch3= (char *) malloc(strlen(ch1))*sizeof(char))
strcpy (ch3, ch1);  /* Copie de l'emplacement pointé par ch1 à celui pointé par ch3*/
```

## Remarque

Les tableaux (et donc les chaînes de caractères) sont toujours manipulés à partir de leur adresse (qui est celle de leur premier élément). Mais dans une déclaration de type **type nomTab [dimension]**, **nomTab** est un pointeur constant (on ne pourra pas changer sa valeur) contrairement à une déclaration de type **type * nomTab**.

## Pointeurs et structures

On peut manipuler les structures avec des pointeurs. On accède ensuite aux différents éléments par **->** à la place de **.**

```
typedef struct {
      char lettre ;
      char libelle [20] ;
      int x;
      int y ;
} POINT ;

POINT pt1= {‘A’, "point A" , 1, 2}, *pt2;
pt2= &pt1 ;
printf("lettre:%c, libelle:%s, abs:%d, ord:%d", pt1.lettre, pt1.libelle, pt1.x, pt1.y ) ;
printf("lettre:%c, libelle:%s, abs :%d, ord:%d", pt2->lettre, pt2->libelle, pt2->x, pt2->y) ;
```

## Fonctions

Elles permettent d'écrire une série d'instructions qui sera exécutée plusieurs fois dans le programme (write once, run anywhere)

Déclaration d'une fonction

```
type_de_retour nomFonction (type1 argForm1, …, typeN argFormN){
    déclarations de variables locales
    liste d'instructions
}
```

Appel d'une fonction

```
nomFonction (argEff1, argEff2)
```

Passage de paramètres par valeur ou par adresse

Lors de l'appel de fonction, les paramètres effectifs sont copiés dans la pile. La fonction travaille alors sur la copie des variables du programme appelant. On parle de transmission par valeur. Pour que la fonction puisse modifier la valeur d'une variable, il faut qu'elle prenne en paramètre l'adresse de cette variable et non sa valeur. On parle alors de passage par adresse.

Passage par valeur

```
void echange (int a, int b) {
      int t;
      t = a;
      a = b;
      b = t;
      return;
}
....
....
int a=5, b=7;
echange(a,b);
printf("a=%d /t b=%d", a, b); /*a=5 b=7 */
```

Passage par adresse

```
void echange (int *a, int *b) {
      int t;
      t = *a;
      *a = *b;
      *b = t;
      return;
}
....
....
int a=5, b=7;
echange(&a,&b);
printf("a=%d /t b=%d", a, b);  /*a=7 b=5 */
```

La fonction main(…)

C'est la fonction principale. L'exécution d'un programme entraîne automatiquement l'appel de cette fonction. Elle retourne un entier qui indique si le programme s'est déroulé sans erreurs ou non. La fonction main(...) peut être utilisée sans arguments et a pour prototype **int main(void).** Elle peut aussi recevoir une liste d'arguments :
**int main ( int argc, char *argv[]).**

- **argc** (argument count) : nombre de mots composant la ligne de commande
- **argv** (argument vector) : tableau de chaînes de caractères, chacune correspondant à un mot de la ligne de commande (argv[0] contient le nom du programme lui même, argv[1] le premier paramètre …)

## Pointeurs sur fonctions

Un pointeur sur une fonction correspond à l'adresse de début de code de cette fonction.

Déclaration d'un pointeur f vers une fonction de paramètres de type1 … typeN et retournant un objet de type typeR : **typeR (*f) (type1, …, typeN);**

```
void direBonjour(void){
    printf("Bonjour");
}

int main(void){
    void (*fct)(void); // déclaration d'un pointeur sur fonction (du type précédent)
    fct =direBonjour; // initialisation du pointeur avec l'adresse de la fonction
    (*fct)(); // appel de la fonction
}
```

## Structures récursives en C (listes, arbres …)

Liste chainée :

```
struct element {
    T valeur ;
    struct element *suivant ;
}
typedef element *liste;
```

Arbre binaire :

```
struct nœud {
    T valeur ;
    struct nœud *filsGauche, *filsDroit ;
}
typedef nœud *arbreBinaire;
```

## Gestion des flux

Flux binaire ou flux texte (si les octets sont interprétés avec le code ASCII par ex)

Déclaration d'un flux : **FILE *nom_interne_fichier ;**

Ouverture : **FILE * fopen (const char *nom_externe_fichier, char *mode) ;**

Mode : "r", "r+", " w", "w+", "a", " a+ ", " r+b"

Fermeture : **int fclose (FILE *flux);** Renvoie *0* si succès ou l'entier *EOF* si erreur

Vidage immédiat du tampon : **int fflush (FILE *flux);** Renvoie *0* si succès ou *EOF* si erreur

Contrôle du flux : **int feof (FILE *flux);** Renvoie une valeur non nulle si fin de fichier
**int ferror(FILE *flux);** Renvoie une valeur non nulle si erreur

Lecture d'un caractère dans le flux texte : **int fgetc (FILE *flux);**

Renvoie le prochain caractère ou *EOF*

Ecriture d'un caractère dans le flux texte : **int fputc (int caractere, FILE *flux);**

Renvoie le caractère écrit ou *EOF*

Lecture d'une chaine dans le flux texte : **char *fgets (char *s, int n, FILE *flux);**

Lit au plus n-1 caractères (moins si elle rencontre *"\n"*), rajoute'\0' à la fin et renvoie la chaine correspondante ou *NULL* si erreur ou fin de fichier.

Ecriture d'une chaine dans le flux texte : **int fputs (const char *s, FILE *flux);**

Renvoie *EOF* si erreur, une valeur non négative sinon.

Lecture avec format dans le flux texte : **int fscanf (FILE *flux, const char *format, …);**

Renvoie le nombre d'éléments effectivement lus et convertis ou EOF si erreur.

Ecriture avec format dans le flux texte : **int fprintf (FILE *flux, const char *format, …);**

Renvoie le nombre de caractères effectivement écrits (sans compter le '\0' final), une valeur négative si erreur

Lecture dans le flux binaire :**size_t fread (void *dest, size_t taille, size_t nb, FILE * flux);**
Lecture de *nb* éléments de *taille* octets depuis *flux* et copie dans *dest*. Renvoie le nombre d'éléments effectivement lus

Ecriture dans le flux binaire:**size_t fwrite (const void *src, size_t taille, size_t nb, FILE *flux);**
Ecriture de *nb* éléments de *taille* octets depuis *src* dans *flux*. Renvoie le nombre d'éléments effectivement écrits

## Gestion des flux standards

Dans <stdio.h> on trouve la définition de 3 flux standards : **FILE *stdin, *stdout, *stderr ;**

stdin est le flux standard d'entrée (clavier par défaut) stdout et stderr sont respectivement le flux standard de sortie (écran par défaut) et le flux d'affichage des erreurs (écran par défaut)

Lecture d'un caractère depuis stdin : **int getchar (void) ;** Renvoie le caractère ou EOF si erreur ou fin de fichier

Ecriture d'un caractère sur stdout : **int putchar (int) ;** Renvoie le caractère ou EOF

Lecture d'une chaîne depuis stdin : **char *gets (char *s) ;** Renvoie la chaine lue ou EOF si erreur ou fin de flux

Ecriture d'une chaîne sur stdout : **int puts (const char *s);** Renvoie EOF si erreur, une valeur non négative sinon

Lecture d'une chaîne formatée depuis stdin : **int scanf (const char *format, …);**
Renvoie le nombre d'éléments effectivement lus et convertis ou EOF si erreur

Ecriture d'une chaîne formatée sur stdout : **int printf (const char *format, …);** Renvoie le nombre de caractères effectivement écrits (sans compter le '\0' final), une valeur négative si erreur

| format | conversion en | écriture |
|---|---|---|
| %d | int | décimale signée |
| %ld | long int | décimale signée |
| %u | unsigned int | décimale non signée |
| %lu | unsigned long int | décimale non signée |
| %o | unsigned int | octale non signée |
| %lo | unsigned long int | octale non signée |
| %x | unsigned int | hexadécimale non signée |
| %lx | unsigned long int | hexadécimale non signée |
| %f | double | décimale virgule fixe |
| %lf | long double | décimale virgule fixe |
| %e | double | décimale notation exponentielle |
| %le | long double | décimale notation exponentielle |
| %g | double | décimale, représentation la plus courte parmi %f et %e |
| %lg | long double | décimale, représentation la plus courte parmi %lf et %le |
| %c | unsigned char | caractère |
| %s | char* | chaîne de caractères |

**Formats d'impression pour printf**

| format | type d'objet pointé | représentation de la donnée saisie |
|---|---|---|
| %d | int | décimale signée |
| %hd | short int | décimale signée |
| %ld | long int | décimale signée |
| %u | unsigned int | décimale non signée |
| %hu | unsigned short int | décimale non signée |
| %lu | unsigned long int | décimale non signée |
| %o | int | octale |
| %ho | short int | octale |
| %lo | long int | octale |
| %x | int | hexadécimale |
| %hx | short int | hexadécimale |
| %lx | long int | hexadécimale |
| %f | float | flottante virgule fixe |
| %lf | double | flottante virgule fixe |
| %Lf | long double | flottante virgule fixe |
| %e | float | flottante notation exponentielle |
| %le | double | flottante notation exponentielle |
| %Le | long double | flottante notation exponentielle |
| %g | float | flottante virgule fixe ou notation exponentielle |
| %lg | double | flottante virgule fixe ou notation exponentielle |
| %Lg | long double | flottante virgule fixe ou notation exponentielle |
| %c | char | caractère |
| %s | char* | chaîne de caractères |

**Formats d'enregistrement pour scanf**

Création et écriture dans un fichier texte de caractères saisis au clavier

```
int main(void){
      char nom[20], car;
      FILE *fic;
      printf("entrez le nom du fichier");
      scanf ("%s", nom);
      if ((fic=fopen(nom, "w"))==NULL)
            printf("Erreur de création");
      else
            while(car=getchar() != EOF)
                  fputc(car, fic);
      fclose(fic);
}
```

Lecture depuis un fichier texte et affichage à l'écran

```
 int main(void){
       char nom[20], car;
       FILE *fic;
       printf("entrez le nom du fichier");
       scanf ("%s", nom);
       if ((fic=fopen(nom, "r"))==NULL)
             printf("Erreur d'ouverture");
       else
             while(car=fgetc(f) != EOF)
                   putchar(car);
       fclose(fic);
 }
```

## Duplication de processus : la commande fork()

La commande fork() permet de dupliquer à un processus de se dupliquer. On parle alors de processus père (celui qui est à l'origine de la duplication) et de processus fils (le processus nouvellement créé).

Exemple

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
const char * quiSuisJe=null;
int main(){
    pid_t pid ;
    quiSuisJe= "le père";
    pid=fork();
    if (pid==0){
        quiSuisJe="le fils";
        printf("je suis %s", quiSuisJe);
    }
    else
        printf("je suis %s", quiSuisJe);
    }
    return 0;
}
```

➡ Première instruction exécutée par le processus fils à sa création

**pid_t fork (void)** retourne :
- -1 si erreur
- 0 au fils
- le pid du fils au père.

Les variables (locales, globales) du fils sont au départ obtenues par duplication mais évoluent ensuite différemment de celles du père.

Les processus père et fils partagent les mêmes descripteurs de fichiers (0/stdin, 1/stdout, 2/stderr).

Obtenir le pid du processus courant : **pid_t getpid (void)**

Obtenir le pid du processus père : **pid_t getppid (void)**

## Communication entre processus

Les signaux permettent d'informer de manière évènementielle (asynchrone) les processus. Ils ont des origines diverses : ils peuvent résulter de causes internes au processus (division par zéro/SIGFPE, erreur d'adressage/SIGSEGV …) , résulter d'une action utilisateur (ctrl-c/SIGINT, quit/SIGQUIT …), être envoyés par le processus lui-même (primitive sleep()/SIGALRM …) ou encore provenir d'autres processus (primitive kill() …)

Chaque type de signal est associé à un traitement (handler) par défaut appelé *SIG_DFL* :
- Terminaison (avec/sans image mémoire appelé fichier core)
- Signal ignoré
- Suspension du processus
- Continuation (reprise d'un processus stoppé)

Un processus ignorera un signal si on associe à ce dernier le handler *SIG_IGN* (pas valable pour les signaux *SIGKILL*, *SIGCONT* et *SIGSTOP)*

Un processus redéfinir son propre handler (pas valable pour les signaux *SIGKILL*, *SIGCONT* et *SIGSTOP)*

L'association d'un signal à un handler se fait grâce à la primitive *signal().*

| Nom | Signification | Comportement |
|---|---|---|
| SIGHUP | Hang-up (fin de connexion) | T(erminaison) |
| SIGINT | Interruption (Ctrl-c) | T |
| SIGQUIT | Interruption forte (Ctrl-\) | T + core |
| SIGFPE | Erreur arithmétique | T + core |
| SIGKILL | Interruption immédiate et absolue | T + core |
| SIGSEGV | Violation des protections mémoire | T + core |
| SIGPIPE | Écriture sur un pipe sans lecteurs | T |
| SIGTSTP | Arrêt temporaire(Ctrl-z) | Suspension |
| SIGCONT | Redémarrage d'un fils arrêté | Ignoré |
| SIGCHLD | un des fils est mort ou arrêté | Ignoré |
| SIGALRM | Interruption d'horloge | Ignoré |
| SIGSTOP | Arrêt temporaire | Suspension |
| SIGUSR1 | Émis par un processus utilisateur | T |
| SIGUSR2 | Émis par un processus utilisateur | T |

Liste signaux : commande **kill -l**

## Communication entre processus (suite)

Terminer son exécution : **void exit (int statut);** Si le processus s'est terminé normalement, *statut* doit normalement être mis à 0

Se mettre en attente de la terminaison d'un fils : **pid_t wait (int *statut);** Le processus est réveillé par un signal *SIGCHLD* et ne fait rien par défaut. S'il n'existe aucun processus fils, la fonction retourne -1 ; sinon elle retourne le pid du fils qui a provoqué la fin de l'attente. Si le processus fils s'est terminé normalement, alors l'octet 0 de son statut est écrit dans le deuxième octet de poids faible de l'entier pointé par *statut* …

Pareil mais pour un fils précis : **pid_t waitpid (pid_t pid, int *statut, int options);**

Se mettre en sommeil : **int sleep (int n);** Suspend l'exécution du processus appelant pendant une durée de N secondes ; il est alors réveillé (par un signal *SIGALRM*). La fonction renvoie 0 si le temps prévu s'est écoulé ou le nombre de secondes restantes si l'appel a été interrompu par un gestionnaire de signal.

Se mettre en attente d'un signal quelconque : **int pause();** Suspend le processus jusqu'à la réception d'un signal qui peut alors soit terminer le processus soit provoquer l'exécution d'une fonction gestionnaire ; dans ce deuxième cas, la fonction pause() retourne -1

Envoyer un signal à un (ou plusieurs) processus : **int kill (pid_t pid, int signal);** signal envoyé au processus pid (si pid>0), à tous les processus du même groupe que le processus courant (si pid=0) ou à tous ceux du groupe |pid| (si pid<0). La fonction retourne 0 en cas de succès, -1 en cas d'échec

Définir un gestionnaire de signal : **void (*signal(int sig, void (*action)(int)))(int);** ...

Exécution d'un nouveau programme à la place de celui qui fait l'appel : **execXXX (...)**

Exemple : **int execv(char *ref, char *argv[]);** Lancement du fichier exécutable *ref* avec les arguments passés dans *argv.* Si la fonction revient c'est qu'il y a eu une erreur (valeur -1 dans ce cas)

## Directives pour le préprocesseur

Directive #include : elle permet d'incorporer dans le fichier source le texte provenant d'un autre fichier. Ce fichier peut être une en-tête de librairie standard (stdio.h, math.h,...) ou n'importe quel autre fichier.

```
#include <nomFichier>
#include "nomFichier "

#include <stdio.h>
```

**<stdio.h>**, pour "Standard Input/Output Header" ou "En-tête Standard d'Entrée/Sortie", est l'en-tête de la bibliothèque standard du C déclarant les macros, les constantes et les définitions de fonctions utilisées dans les opérations d'entrée/sortie.

Il y en a beaucoup d'autres, **<string.h>** pour la manipulation des chaînes, **<time.h>** pour manipuler les formats de date et heure etc.

Directive #define : elle permet de définir des constantes symboliques et des macros avec paramètres.

```
#define NB_LIGNES 10

#define CARRE(a) a*a
```

## La compilation conditionnelle

La compilation conditionnelle à pour but de compiler ou d'ignorer des parties de code, le choix étant basé sur un test effectué à la compilation.

### Tester une condition (#if, #elif, #else)

```
#if condition1
     partie du programme 1
#elif condition2
 partie du programme 2
...
#else
    partie du programme N
#endif
```

```
#define PROCESSEUR "32bits"
#if PROCESSEUR == "32bits"
    ...
#elif PROCESSEUR == "64bits"
    ...
#endif
```

### Tester l'existence d'un symbole (#ifdef) ou sa non-existence (#ifndef)

```
#ifdef macro
    partie du programme 1
#else
   partie-du-programme 2
#endif
```

```
#ifndef macro
    partie du programme 1
#else
    partie-du-programme-2
#endif
```

```
#define _Win32
#ifdef __unix__ /* si le programme tourne sous Unix */
    # include <unistd.h>
#elif defined _WIN32 /* si le programme tourne sous Windows */
    # include <windows.h>
#endif
```

# III.2. Rappels sur le langage Java

## Généralités

Java est interprété : byte code interprété par une JVM.

Java est indépendant de toute plate forme (portable) : code source et byte code indépendant de la machine

Java est Orienté Objet : chaque fichier source contient la définition d'une ou plusieurs classes ; il n'est pas complètement OO car il gère les types primitifs

Java est simple : pas de pointeurs comme en C / C++

Java est fortement typé : chaque variable a un type

Java gère automatiquement la mémoire par le Garbage collector : allocation mémoire lors de la création des objets, désallocation à la destruction

Java est multitâches : Les threads sont des unités d'exécution isolés qui se partagent une même zone mémoire (processus légers)

Java est sûr : impossible d'accéder à des fichiers systèmes, de lancer d'autres programmes, impossible d'imiter d'autres programmes non Java.

## Développer en Java

La JRE (Java Runtime Environment) contient uniquement l'environnement d'exécution de programmes Java (machine virtuelle - JVM - capable d'exécuter le byte-code et les bibliothèques standards de Java).

JDK (Java Developpement Kit) est un ensemble d'outils pour développer ses applications java (JDK est téléchargeable gratuitement sur le site d'Oracle) : JRE + compilateur + bibliothèques utiles.

Compilation : **javac MonPgm.java** Exécution : **java MonPgm**

Variable **PATH** pour compiler/exécuter depuis n'importe où sur le disque

Environnements de développement intégré (éditeur de texte incorporé, outils de déboguage, de création d'interface graphiques … pour faciliter le travail du programmeur) : Eclipse, Netbeans …

Trois éditions de développement en Java :

- Java 2 Standard Edition (J2SE) : contient les APIs nécessaires au développement d'applications de bureau (lang, util, swing …).

- Java 2 Micro Edition (J2ME) : petites applications capables de fonctionner en environnement limité (smartphones, agendas éléectroniques …). Contient des API de J2SE et des API spécifiques.

- Java 2 Enterprise Edition (J2EE) : contient Les API de J2SE + un ensemble de d'API permettant le développement d'applications serveur d'entreprises (EJB, Servlet / JSP, ...).

## APIs de Java

Java contient de nombreuses API (Application Programming Interface), plus ou moins nombreuses selon les éditions :

- IO : gérer les entrées sorties
- Swing, AWT : interfaces graphiques
- RMI : appels de procédures distantes
- Servlets : gèrent des requêtes HTTP et génèrent des pages HTML dynamiques (travaillent avec un serveur Web).
- Applets : applications Web côté client
- EJB pour développer des composants distribués
- JMS pour la communication par messages asynchrones
- Java mail : gérer les mails
- JDBC : accès aux bases de données
- Java 3D : graphisme en 3 dimensions
- Beaucoup d'autres …..

## Programmation Orientée Objet

Les concepts de base de la Programmation Orientée Objet (POO) sont ceux de classe et d'objet.

Une classe est une entité qui comporte à la fois des champs (variables …) et des méthodes (fonctions qui s'appliquent sur les champs).

Une fois définie, une classe constitue un nouveau type. Il est alors possible de créer des objets de ce type. Chaque objet possède ses propres valeurs de champ.

Un constructeur est une méthode particulière, sans valeur de retour, appelée à la création de l'objet et qui porte le même nom que la classe. Un constructeur est généralement utilisé pour l'affectation de valeurs initiales, mais peut effectuer n'importe quelle action utile pour l'objet.

Un objet est une instance de classe, une classe correspond à une collection d'objets.

Le champ d'une classe peut être d'un type primitif habituel (int, float …) mais peut être aussi de type classe. Une classe **Voiture** peut par exemple définir un objet de type **Moteur**, la classe Moteur ayant elle-même ses propres champs et méthodes.

## Définition d'une classe

Nous allons définir une classe **Point**. Un point est repéré par ses coordonnées **x** et **y**, et propose les méthodes **deplace(…)** et **affiche()**.

```
public class Point {

    public int x ;
    public int y ;

    public Point (int abs, int ord){ // constructeur
        x=abs ;
        y=ord ;
    }

    public void deplace(int newAbs, int newOrd){ // changement de coordonnées
        x=newAbs;
        y=newOrd;
    }

    public void affiche(){ // affichage des coordonnées
        System.out.println("abscisse "+x+" et ordonnée "+y);
    }
}
```

Selon le principe d'encapsulation, il ne faut pas permettre un accès direct aux champs d'une classe directement depuis l'extérieur de celle-ci. Les champs sont généralement déclarés **private**.

Les méthodes de la classe peuvent accéder aux champs. Si elles sont déclarées **public** on pourra alors appeler ces méthodes depuis l'extérieur. Cela permet donc indirectement d'accéder aux données.

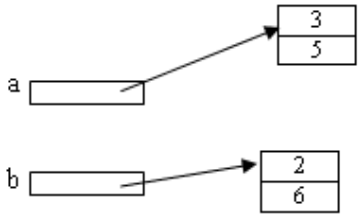
On parle de méthodes d'accès et d'altération pour désigner les méthodes permettant de lire ou de mettre à jour ces champs

```
public class Point {
    private int x, y ;
    public int getX (){              // méthode d'accès à l'abscisse
          return x ;
    }
    public int getY (){              // méthode d'accès à l'ordonnée
        return y ;
    }
    public void setX (int abs){  // méthode de modification de l'abscisse
        x=abs ;
    }
    public void setY (int ord){  // méthode de modification de l'ordonnée
        y=ord ;
    }
}
```

## Manipulation d'objets

Lorsque l'on créé une variable de type classe (un objet), on affecte à la variable non seulement l'objet mais une référence à cet objet.

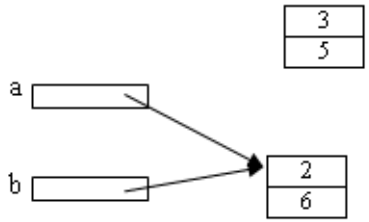
```
Point a, b;
a = new Point (3,5);
b = new Point (2,6);
```




**a** et **b** sont des références qui pointent chacune sur leur propre objet.

Affectation de références

```
a=b;
/* l'objet initialement
référencé par a est
candidat pour le GC */
```




C'est la référence contenue dans **b** qui a été recopié dans **a**. Les variables **a** et **b** désignent maintenant un seul et même objet. Modifier **a** revient à modifier **b** et inversement.

Comparaison de références

```
System.out.println(a==b); // affiche true
```

Les opérateurs **==** et **!=** s'appliquent aux objets, mais ils portent sur des références non sur le contenu de leur champs.

Comparaison de contenu

```
System.out.println(a.equals(b)); // affiche true
```

La méthode **equals(…)** de la classe **Object** compare le contenu des objets.

Référence Null

```
Point a  = new Point (5,2);
a=null ; /* l'objet référencé par a devient
candidat pour le GC */
```

Avant l'appel du constructeur, une référence ne pointe sur rien. Le mot clé **null** (utilisé dans des comparaisons ou affectations) permet de gérer les références nulles.

## Garbage Collector (ramasse miette)

Lorsque l'opérateur **new** est appelé, java alloue un emplacement mémoire pour l'objet et l'initialise

Lorsqu'un objet n'est plus référencé, son emplacement peut alors être récupéré pour autre chose. En pratique, l'objet n'est pas forcément supprimé immédiatement mais il devient candidat au ramasse miette (Garbage Collector).

## Champs de type classe (objets membres)

La classe **Cercle** suivante contient le champ **p** de type **Point** :

```
public class Cercle {
    private float rayon;
    private Point p ; // classe Point définie précédemment
    public Cercle (int abs, int ord, float ray){ // constructeur
        p= new Point (abs, ord) ;
        rayon=ray ;
    }
    public void deplace(int newAbs, int newOrd){ // changement de coordonnées
        p.setX(newAbs);
        p.setY(newOrd);
    }
    public void affiche(){ // affichage des coordonnées et du rayon
        System.out.println("abscisse "+p.getX()+", ordonnée "+p.getY()+" et rayon" +rayon);
    }
}
```

## Champs déclarés 'final'

Les champs déclarés **final** n'ont le droit qu'à une seule initialisation.

Cette affectation doit être faite au plus tard par le constructeur.

L'exemple suivant fournit une erreur de compilation à cause de **c** qui devrait être initialisé au plus tard par le constructeur ;

```
public class Constantes {
    private final int a=10 ;
    private final int b ;
    private final int c ;
    public Constantes(int val) {
        b= val;
    }
}
```

## Surdéfinition de méthodes

Plusieurs méthodes peuvent porter le même noms, pour peu que le nombre ou le type des arguments permette au compilateur d'effectuer son choix.

Voici 3 définitions de la méthode **deplace(…)** au sein d'une même classe :

```
public void deplace (int dx, int dy) { …}
public void deplace (float dx, float dy) {…}
public void deplace (int dx) {…}
```

## Champs et méthodes de classe ‘static’

En java, on peut définir des champs qui au lieu d’exister pour chacun des objets de la classe, n’existent qu’en un seul exemplaire pour tous les objets de cette classe. On parle alors de champ de classe ou de champ statique.

De même, on peut définir des méthodes de classe (ou méthodes statiques) qui peuvent être appelées indépendamment de toute existence d’objet.

```
public class Point {
      private int x, y ;
      public static int nbPoints=0 ;
      public Point (int abs, int ord){
            x=abs ;
            y=ord ;
            nbPoints ++;
      }
}
……. // depuis une méthode xxx d’une classe xxx
Point p1=newPoint(3,5);
Point p2=newPoint(1,4);
System.out.println(“Nombre d’objets créés“+Point.nbPoints);
```

---

```
public class Utilitaires {
     public static float moyenne(int [] tab){
          …….
     }
}
……. // depuis une méthode xxx d’une classe xxx
int tableauEntier[] = {0,1,2,3,4,5,6,7,8,9};
float moy=Utilitaires.moyenne(tableauEntier);
```

## Passage de paramètre par valeur ou par référence

Pour un argument formel de type primitif, l'argument effectif correspondant est une valeur. En fait, c'est une copie de la valeur de la variable qui est fournie à l'appel. La variable initiale ne pourra ainsi pas être modifiée par la méthode.
.
Pour un argument formel de type classe, l'argument effectif correspondant est la référence à un objet de cette classe. Il s'agit certes d'une copie de référence, mais qui désigne toujours le même objet qui peut donc être modifié par la méthode.

Rajoutons dans la classe Point la méthode **clone(…)** suivante :

```
public class Point {
    private int x, y ;
    public Point (int abs, int ord){
        x=abs ; y=ord ;
    }
    public void setX (int abs){
        x=abs ;
    }
    public void setY (int ord){
        y=ord ;
    }
    public void clonerPoint (Point p){
        p.setX(x) ; p.setY(y) ;
    }
    public void affiche(){
        System.out.println("abscisse "+x+", ordonnée "+y);
    }
}
……. // depuis une méthode xxx d'une classe xxx
Point p1=new Point (3,5) ; Point p2=new Point (4,6) ;
p1.clonerPoint(p2) ;p2.affiche() ; // affiche "abscisse 3 et ordonnée 5"
```

## Héritage

A partir d'une classe existante dite classe de base, il est possible de définir une nouvelle classe, dite classe dérivée.

Cette nouvelle classe hérite automatiquement des champs et méthodes de la classe de base qu'elle pourra modifier ou compléter à volonté sans que cela ne change quoi que ce soit pour la classe de base.

Une classe dérivée peut elle-même servir de classe de base pour une nouvelle classe dérivée, ainsi de suite …

Une classe dérivée n'a pas accès aux membres privés de sa classe de base.

Le constructeur de la classe dérivée doit prendre en charge l'intégralité de la construction de l'objet. On peut appeler le constructeur de la classe de base grâce au mot-clé **super(…)**.

En Java, toutes les classes héritent de la super-classe **java.lang.Object**

```
public class Point {
    private int x, y ;
    public point (int abc, int ord){
        x=abc;
        y=ord ;
    }
    public void afficheCoord(){
        System.out.println("abs "+x+", ord "+y);
    }
     …….
}
```

```
public class PointCol extends Point {
    private byte couleur;
    public pointCol (int abc, int ord, byte coul){
        super(abc, ord) ; couleur=coul ;
    }
    public void colore (byte coul){
        couleur=coul ;
    }
}
…….. // depuis une méthode xxx d'une classe xxx
PointCol pc=new PointCol (2, 5, 120);
pc.afficheCoord();
pc.colore(134);
```

## Redéfinition de méthodes

Une classe dérivée peut fournir une nouvelle définition d'une classe ascendante.

La méthode doit être de même nom, de même signature et de même type de valeur de retour.

Une redéfinition d'une méthode ne doit pas diminuer les droits d'accès mais peut les augmenter.

## Polymorphisme

Le polymorphisme permet de manipuler des variables sans en connaître tout à fait le type.

Le polymorphisme utilise la redéfinition de méthode et le principe selon lequel une variable de type P peut référencer un objet de type P' dérivé de P.

Le polymorphisme permet alors d'appliquer la méthode la plus appropriée au type de l'objet référencé.

Ce choix est fait à l'exécution (plus à la compilation) et porte le nom de ligature dynamique.

```
public class Point {
   private int x, y ;
   public Point (int abc, int ord){
      x=abc; y=ord ;
   }
   public int getX (){ return x ; }
   public int getY (){ return y ; }
   public void affiche(){
      System.out.println("abs"+x+",ord"+y);
   }
}
```

```
public class PointCol extends Point{
   private byte couleur;
   public PointCol (int abc, int ord, byte coul){
      super(abc, ord) ; couleur=coul ;
   }
   public void affiche(){
      System.out.println("abs"+getX()+",ord"+getY()+",coul"+couleur);
   }
}
```

```
…. // depuis une méthode xxx d'une classe xxx
Point p1=new Point(5,22); p1.affiche();  // appelle affiche() de Point
Point p2 =new PointCol(15,2,(byte)8); p2.affiche(); // appelle affiche() de PointCol
```

## Interfaces

Une interface définit les signatures d'un certain nombre de méthodes ainsi que des constantes.

Lorsqu'une classe implémente une interface (mot-clé **implements**), elle doit implémenter toutes les méthodes déclarées dans l'interface.

Une même interface peut être implémentée par plusieurs classes, comme une classe peut implémenter plusieurs interfaces.

Les interfaces peuvent utiliser l'héritage.

```
public interface I1 {
    public void f (int n) ;
}

public interface I2 {
    public void g ();
}

public class A implements I1, I2 { // A doit obligatoirement définir f(…) et g()
    public void f (int n){…}
    public void g (){…}
    public void h (){…}
}
```

## Classes abstraites

Une classe abstraite est une classe qui ne peut pas servir à instancier des objets. Elle ne peut servir que de classe de base pour une dérivation.

Dans une classe abstraite, on peut trouver des champs et méthodes dont héritera la classe dérivée. On peut aussi trouver des méthodes dites abstraites (dont on ne fournit que le type et la valeur de retour).

Remarque : dès qu'une classe comporte une ou plusieurs méthodes abstraites, elle est abstraite même si on n'indique pas le mot-clé abstract devant sa déclaration (ce qui reste quand même conseillé).

Une méthode abstraite doit être déclarée public, ce qui est logique puisque sa vocation est d'être redéfinie dans une classe dérivée.

```
public abstract class A {
        public void f() {…}
        public abstract void g (int n);
}
```

```
public class B extends A {
        public void g (int n) { …}
        …
}
```

```
… // depuis une méthode xxx d'une classe xxx
A a= new A (…);     // erreur
B b =new B (…);     // OK
A a =new B (…);     // OK (polymorphisme)
```

## Types primitifs et instructions de base

Syntaxe proche du C : **int**, **float**, **char** …

## Tableaux

- Création par l'opérateur **new** : **int t1[] =new int [10];**
  ou utilisation d'un initialiseur : **int t2[]={1,5,3,8};**
- Affectation de tableaux : **t1= t2; // affectation de référence, sinon copier case par case**
- Taille d'un tableau : champ **length** : **t2.length; // retourne 4**
- En argument d'une méthode : passage par référence, modifiable par la méthode
- Tableaux à plusieurs indices : **int t [][] ; t= {new int[3], new int[2]};**

## Chaînes de caractères

- Création par l'opérateur **new** : **String ch1= new String("bonjour");** (nombreuses surcharges du constructeur) ou directement : **String ch2= " tout le monde";**
- Accès aux caractères : **ch1.charAt(0) // vaut 'b'**
- Concaténation de chaînes : **ch1+ch2 // vaut "bonjour tout le monde";**
- Extraction d'une sous-chaîne : **ch1.substring(2,4); // vaut "njo"**
- Taille d'une chaîne : **ch1.length() // vaut 7**

Attention : comparaison de référence (**==**) ou de contenu (**equals()**)

## Exceptions

Une exception est une erreur se produisant dans un programme et qui conduit souvent son arrêt.

En java, les exceptions peuvent être capturées puis traitées (blocs **try** et **catch**).

Java contient une classe **Exception** et un ensemble de classes dérivées dans lesquelles sont répertoriées de nombreux types d'exception.

On peut ainsi essayer de capturer toutes les exceptions **catch (Exception …)** ou des types d'exceptions plus précis **catch (ArithmeticException …)** ; on peut mettre plusieurs blocs **catch** qui se suivent pour fournir un traitement spécifique pour chaque type d'exceptions.

Le programmeur peut aussi créer ses propres exceptions : création d'une classe héritant de **Exception** + utilisation des mots-clés **throws** et **throw**.

Un bloc **finally** peut être utilisé pour exécuter une série d'instructions quoi qu'il advienne (qu'une exception se produise ou non).

```
public class MaClasse{
    public static void main(String[] args) {
        try {
            System.out.println(10/0);
        } catch (ArithmeticException e) {
            System.out.println("Division par 0 !");
        }
        finally{
            System.out.println("Il ne faut pas !");
        }
    }
}
```

```
public class MesErreurs extends Exception{
    public MonErreur(…){ …}
    …
}

……. // dans une classe quelconque

public … maFonction(…) throws MonErreur {
    if (…)
        throw new MonErreur ();
}
```

## Threads (processus légers)

La manière la plus connue de créer un thread consiste à créer une classe qui hérite de la classe **Thread**.

Il faut implémenter une méthode **run()**. Lorsque le thread démarre, Java appelle cette méthode run(). Un thread se termine lorsque sa méthode run() se termine.

Une fois qu'un objet thread créé, il faut appeler sa méthode **start()** (définie dans la classe parente **Thread**) ; le thread passe alors à l'état prêt ce qui lui permet de démarrer dès que possible (mais pas forcément de suite)

Le thread principal d'une application est sa méthode **main()**

```
class MonThread extends Thread {
    MonThread() {
        ..... // code du constructeur
    }
    public void run() {
        ..... // code a exécuter une fois le thread lancé
    }
}

public class MaClasse{
    public static void main(String[] args) {
        MonThread p = new MonThread();
        p.start();
    }
}
```

## Synchronisation des Threads

Lorsque plusieurs threads accèdent en même temps à une ressource commune, cela peut la laisser dans un état incohérent. Pour éviter cela, on peut utiliser le principe d'exclusion mutuelle en plaçant un verrou sur cette ressource : mot-clé **synchronized**.

Lorsqu'on synchronise une méthode d'instance, le premier thread qui y accède obtient un verrou sur l'objet. Pendant ce temps, les autres threads ne pourront alors accéder ni à cette méthode ni aux autres méthodes synchronisées de l'objet. De même, un verrou sur une méthode de classe (statique) verrouillera les autres méthodes statiques synchronisées de la classe.

```
public synchronized void majDonnees() {
        // Méthode synchronisé sur l'instance courante
}
```

Il est possible de synchroniser un bloc de code plutôt qu'une méthode, on pourra alors spécifier l'instance que l'on souhaite synchroniser.

```
public void majDonnees() {
        synchronized (this){
    // bloc de code synchronisé : verrouillage de l'instance courante (this)
        }
}
```

## Méthodes associées aux Threads

La méthode statique **void sleep (int delai)** (de la classe **Thread**) met le Thread courant en sommeil pendant une durée de temps exprimée en ms. Les verrous ne sont pas libérés.

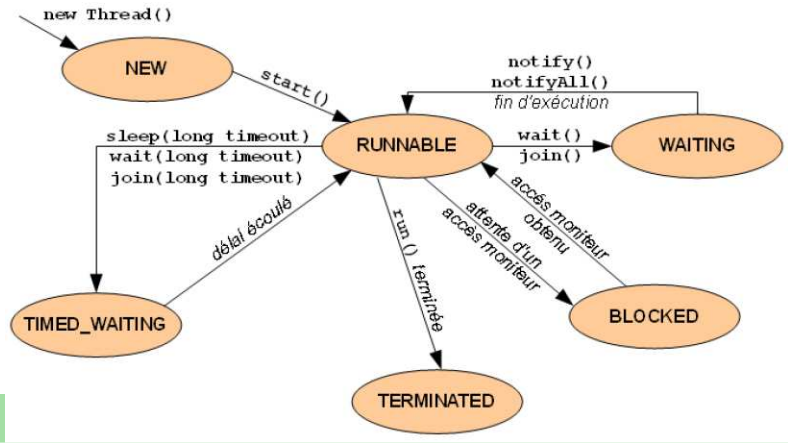
La méthode **void join()** (de la classe **Thread**) attend la fin de l'exécution d'un autre Thread pour continuer.

Les méthodes **void wait()**, **void notify()** et **void notifyAll()** (de la classe **Object**) ne s'utilisent que dans des méthodes **synchronized**.

La méthode **void wait()** bloque le thread courant et le met en attente. Il doit rendre le verrou. Il sera réveillé par **void notify()** ou **void notifyAll()**.

Les méthodes **void notify()** et **void notifyAll()** permettent de débloquer respectivement un Thread ou l'ensemble des Threads bloqués sur un objet.

Les méthodes **void wait()** et **void join()** peuvent aussi être appelée avec un délai maximal d'attente en paramètre.



## Questions de cours sur le langage C

1) Quelles sont les 3 valeurs successives de *y* affichées par le programme suivant ?

```
#include <stdio.h>

int main(){
    int x=1, y=2, *p, *q ;
    printf ("y=>%d\n", y);
    p= &x ;
    y= *p ;
    printf ("y=>%d\n", y);
    q=(int *) malloc(sizeof(int));
    *q=3 ;
    y= *q ;
    printf ("y=>%d\n", y);
}
```

2) Ecrire un petit programme qui affiche la liste des paramètres passés en ligne de commande.

```
int main (argc, char ** argv){
    // Compléter
}
```

3) Même chose mais en partant de la déclaration suivante :

```
int main (argc, char * argv[]){
   // Compléter
}
```

## Questions de cours sur le langage Java

1) Les Threads sont souvent associés aux notions de "ressource critique" et de "synchronisation des accès" contrairement aux processus créés par la commande *fork()*. Pourquoi ?

2) Deux processus accèdent simultanément à la méthode Java suivante, *printMultiple()*. Donner les traces possibles de cette exécution. Comment introduire un accès exclusif à cette méthode ? Donner toutes les traces possibles après cette modification.

```
public void printMultiple(){
     for (int i=0; i<2; i++){
          System.out.println("i="+i);
     }
}
```

3) Soit une classe C contenant les deux méthodes suivantes (remarque : P est une classe définie par ailleurs) appelées depuis la même JVM (pas à distance) :

```
class C {
    public void maMethode1(P p){
        ...
    }
    public synchronized void maMethode2(int i){
    ...
    }
}
```

a) Ces deux méthodes sont-elles appelables sur la classe C elle-même ou bien sur un objet de cette classe (en d'autres termes, pourrais-je avoir un appel de type : *C.maMethode2(4);* par exemple) ?

b) Si *maMethode1 (...)* modifie son paramètre p (les valeurs de ses champs), est-ce que cette modification est répercutée dans le programme appelant ?

c) Même question pour *maMethode2 (...)* et son paramètre *i* ?

d) Est-il possible (par principe) de passer en paramètre de *maMethode1 (...)* un objet de type *PSub* où *PSub* est une classe dérivée de *P.*

e) Et un objet de type *PSup* où *PSup* est une classe dont *P* hérite.

d) Si un thread est en train d'exécuter *maMethode2 (...) sur un objet O1*, est ce qu'un autre thread peut exécuter la même méthode (*maMethode2 (...))* sur le même objet (O1) ? Et *maMethode1(…) sur O1 ?* Et *maMethode2(…)* sur un autre objet *(O2)* ?