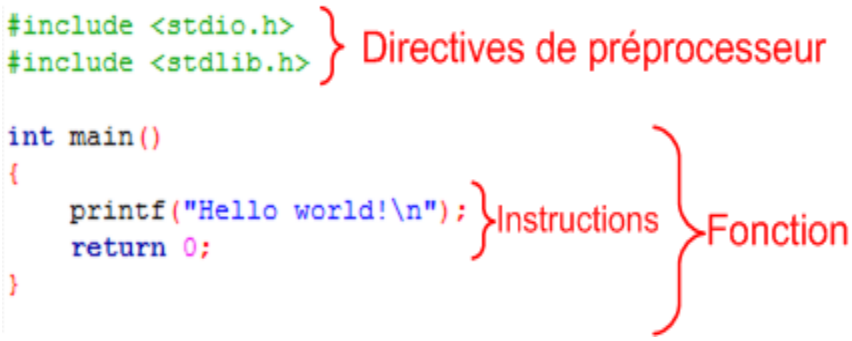
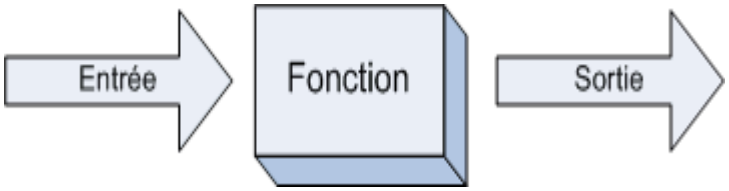
1. Définitions

Tous les gros programmes sont en fait des assemblages de petits bouts de code, et ces petits bouts de code sont justement ce qu'on appelle... **des fonctions.** Un programme en C commençait par une fonction appelée "main".



On dit qu'une fonction possède une entrée et une sortie:



Lorsqu'on appelle une fonction, il y a 3 étapes :

1. L'entrée: on fait "rentrer" des informations dans la fonction (en lui donnant des informations avec lesquelles travailler)
2. Les calculs : grâce aux informations qu'elle a reçues en entrée, la fonction travaille.
3. La sortie : une fois qu'elle a fini ses calculs, la fonction renvoie un résultat. C'est ce qu'on appelle la sortie, ou encore le retour.

Schéma d'une fonction :Voici le schéma d'une fonction:

```
<type> nomFonction(parametres)
{
    // Insérez vos instructions ici
}
```

type : c'est le type de la fonction. Comme les variables, les fonctions ont un type. Ce type dépend du résultat que la fonction renvoie : si la fonction renvoie un nombre décimal, vous mettrez sûrement double, si elle renvoie un entier vous mettrez int ou long par exemple. Mais il est aussi possible de créer des fonctions qui ne renvoient rien . Il y a donc 2 sortes de fonctions :

- Les fonctions qui **renvoient une valeur** : on leur met un des types que l'on connaît (char, int, double...)
- Les fonctions qui **ne renvoient pas de valeur** : on leur met un type spécial "void" (qui signifie "vide").

nomFonction : c'est le nom de votre fonction (pas d'accents, pas d'espaces etc).

parametres : entre parenthèses, vous pouvez envoyer des paramètres à la fonction. Ce sont des valeurs avec lesquelles la fonction va travailler.
Par exemple, pour une fonction "triple", vous envoyez un nombre en paramètre. La fonction "récupère" ce nombre et en calcule le triple, en le multipliant par 3. Elle renvoie ensuite le résultat de ses calculs.

- Vous pouvez envoyer autant de paramètres que vous le voulez.
- Vous pouvez aussi n'envoyer aucun paramètre à la fonction, mais ça se fait plus rarement.

Ensuite vous avez les **accolades** qui indiquent le début et la fin de la fonction.

2. Exemple de Fonction

```
int triple(int nombre)
{
    int resultat = 0;

    resultat = 3 * nombre;  // On multiplie le nombre qu'on nous a
transmis par 3
    return resultat;       // On retourne la variable resultat qui vaut
le triple de nombre
}
```

La fonction suivante fait exactement la même chose que la fonction précédente, elle est juste plus rapide à écrire

```
int triple(int nombre)
{
    return 3 * nombre;
}
```

2.1. Plusieurs paramètres

Il est possible de créer des fonctions prenant plusieurs paramètres. Par exemple, une fonction addition qui additionne deux nombres a et b :

```
int addition(int a, int b)
{
    return a + b;
}
```

2.2. Aucun Paramètre

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

Fonction "bonjour" n'a pas de "return". Elle ne retourne rien. Une fonction qui ne retourne rien est de type void.

2.3. Quelques exemples:

1-

```
int max (int i, int j)
/* cette fonction retourne le plus */
/* grand des deux entiers i et j   */
 {
   if (i>j)
      return (i);
   else
      return (j);
 }
```

2-

```
void mise_a_zero (double t[], int n)
/* cette fonction ne renvoie aucune valeur */
/* mise a zero du tableau de reels t       */
/* de dimension n                          */
 {
   /* variable locale */
   int i;
   for (i=0; i<n; i++)
      t[i]=0.0;
 }
```

3. Appel d'une Fonction

Code :

```
#include <stdio.h>
#include <stdlib.h>
int triple(int nombre)
{
  return 3 * nombre;
}
int main(int argc, char *argv[])
{
    int NEntre = 0, NTriple = 0;

    printf("Entrez un nombre... ");
    scanf("%d", &NEntre);
   NTriple = triple(NEntre);
    printf("Le triple de ce nombre est %d\n", NTriple);
    return 0;
}
```

Notre programme commence par la fonction main.
On demande à l'utilisateur de rentrer un nombre. On envoie ce nombre qu'il a rentré à la fonction triple, et on récupère le résultat dans la variable NTriple.

4. Passage de paramètres

On appelle *arguments formels* d'une fonction ceux qui sont utilisés dans sa définition ainsi:

```
int fact (int n)
 {
   ... /* n est l'argument formel */
 }
```

Les variables utilisés à l'appel de la fonction sont appelés *arguments réels*:

```
...
int main()
 {
   int res, i;
   ...
   res=fact(i); /* i est l'argument reel */
   ...
 }
```

Le lien entre arguments réels et formels est défini par le *mode de passage* des arguments. En langage C, **tous les arguments sont passés par valeur**. Dans ce mode, la fonction ne travaille pas directement sur l'argument réel mais sur une variable locale à la fonction, déclarée implicitement et initialisée à la valeur de l'argument réel (d'où le nom de passage par valeur). Ceci signifie que la fonction **ne peut pas modifier la valeur de ses arguments réels**.

La fonction mise_a_zero du paragraphe précédent:

```
void mise_a_zero (double t[], int n)
 {
   /* variable locale */
   int i;
   /* t[i] sera bien modifie   */
   /* en dehors de la fonction */
   for (i=0; i<n; i++)
     t[i]=0.0;
 }
```

l'idée ici, est que si l'**on ne peut modifier le tableau lui-même, il est possible de modifier ses éléments**.

```
int main ()
{
double tt[100];
......
mise_a_zero (tt,100);
.....
}
```

En effet, **passer à la fonction le nom du tableau revient à lui passer son adresse en mémoire et c'est elle qui ne peut être modifiée**.

Remarque :

Pour passer un tableau comme argument, il est inutile de préciser sa première dimension. Par exemple:

- double t[]; tableau à une dimension;
- int itt[][100]; tableau à deux dimensions...

5. Règles de visibilité

La portion d'un programme où une variable existe est appelée portée (scope) de la variable.

A l'intérieur d'une fonction, il est possible d'utiliser des variables locales aussi bien que des variables globales, mais il est recommandé soit de déclarer les variables localement autant que possible, soit de passer les variables globales en paramètres, car ceci évite de manipuler par erreur des variables déclarées dans le programme principal et utilisées également avant et après l'appel à la fonction. Normalement, une fonction ne devrait utiliser pour ses calculs que les valeurs de ses paramètres et de ses variables locales. Cela demande un peu plus de travail au programmeur, mais évite nombre d'erreurs, et rend la fonction réutilisable d'un programme à l'autre.

Deux variables déclarées dans deux fonctions différentes peuvent avoir le même nom: ce seront cependant deux variables différentes. Une variable de programme peut avoir le même nom qu'une variable locale. Dans la fonction, après la déclaration de la variable locale, seule la variable locale reste visible.

Exemple

L'exemple suivant va vous permettre de vérifier si vous avez compris les règles de visibilité précédentes.

```
#include <stdio.h>
#include <math.h>
float x, fact;

float factorielle(float x)
{
int i;
float z = 1.0;
for (i=2; i<=floor(x); i++) {
z = z * i;
}
x = 0;
fact = z;
return fact;
}

void main()
{
x = 12;
fact = factorielle(x);
printf("Factorielle de %1.0f = %1.0f\n", x, fact);
fact = 4;
factorielle(4.0);
printf("Factorielle de 4.0 = %1.0f\n", fact);
}
```

Les noms x et z déclarés localement dans la fonction factorielle ne sont connus que dans cette fonction. Les variables du programme principal, x et fact, déclarées après les directives d'inclusion sont connues partout, y compris à l'intérieur de factorielle. Cependant le paramètre x se comporte comme une variable locale à la fonction factorielle, distincte de la variable du même nom déclarée globalement.

On en tire les conclusions suivantes: l'instruction `printf("Factorielle de %1.0f = %1.0f\n", x, fact);` placée dans la fonction main affiche la valeur du x global. Cette valeur n'est pas changée par l'appel factorielle(x) car la variable x visible à l'intérieur de la fonction factorielle est locale et masque la variable globale du même nom. L'affectation x=0 n'a donc pas d'incidence sur la variable x globale. La valeur du x affichée par le premier printf du programme principal est donc 12.0.
D'autre part comme fact est une variable globale, elle est connue aussi bien à l'intérieur de la fonction factorielle qu'à l'intérieur de la fonction main. Comme il n'y a pas, à l'intérieur de factorielle, de variable de même nom, l'affectation fact=z; modifie la variable fact globale et le deuxième printf affiche donc bien la valeur de la factorielle de 4 (24) et non pas 4.0 comme on pourrait s'y attendre.

Remarque
Une façon d'éviter toutes ces subtilités est de choisir des noms différents pour toutes les variables de votre programme, quel que soit l'endroit de leur déclaration et de limiter au maximum l'emploi de variables globales.

Exercice

Ecrire une fonction qui calcule et renvoie la valeur du polynôme $P(X)$, représenté par le tableau de ses coefficients, pour une valeur donnée de la variable X.

Correction de l'exercice

```
include<stdio.h>
#define TM 100

float calcul_poly(float coeff[],float x,int n)
{
   float x0,som;
   int i;

   som=coeff[0];
   x0=x;
   for(i=1;i<n;i++)
   {
      som=som+coeff[i]*x0;
      x0=x0*x;
   }
   return(som);
}
int main(void)
{
   float x,val,coeff[TM];
   int i,n;

   printf("Entrez le degre du polynome ");
   scanf("%d",&n);
   for(i=0;i<=n;i++)
   {
      printf("Entrez le coefficient de degre %d du polynome ",i);
```

```
      scanf("%f",&coeff[i]);
   }
   printf("Entrez la valeur de x ");
   scanf("%f",&x);

   val=calcul_poly(coeff,x,(n+1));

   printf("L'evaluation donne %f",val);
   printf("\n");
   return(0);
}
```

6. La Récursivité

Le langage C supporte la récursivité: une fonction peut s'appeler elle-même. L'exemple le plus courant est la fonction qui calcule *factorielle n* (pour n>=0):

```
double factoriel(int n)
{
   if(n<=1)
     return(1);
   else
      return(n*factoriel(n-1));
}
```

Ecrire une fonction calculant C_n^p à partir de *n* et *p*. Constatez que la formule de base

$$C_n^p = \frac{n!}{p!(n-p)!}$$

```
#include<stdio.h>
#define TM 100

double factoriel(int n)
{
   if(n<=1)
     return(1);
   else
      return(n*factoriel(n-1));
}

int main(void)
{
   int n,p;
   double c;

   printf("Calculons C(n,p)\n");
   printf("Entrez la valeur de n  ");
   scanf("%d",&n);
   printf("Entrez la valeur de p  ");
   scanf("%d",&p);
```

```
   c=factoriel(n)/(factoriel(p)*factoriel(n-p));
   printf(" C(%d,%d) vaut %lf", n,p,c);
   printf("\n");
   return(0);
}
```