Jusqu'à présent, la plupart des programmes que nous avons réalisés n'utilisaient que la fonction principale main. Mais alors que vos programmes deviennent progressivement plus complexes, vous pouvez simplifier votre tâche et améliorer la lisibilité en les subdivisant en sous-ensembles dénommés fonctions.

Une fonction désigne une entité de données et d'instructions qui fournie une solution à une (petite) partie bien définie d'un problème plus complexe. Elle peut faire appel à d'autres fonctions, leur transmettre des données ou bien en recevoir en retour. L'ensemble des fonctions ainsi reliées doit alors être capable de résoudre le problème global.

Il arrive que nos programmes utilisent très fréquemment un même groupe d'instructions. Il devient très avantageux alors d'en faire une ou plusieurs fonctions particulières, que l'on peut même regrouper dans une bibliothèque (en dehors du fichier du programme principal). Ainsi, avec cette démarche, nous réalisons une étude une fois pour toute, et ensuite chaque programme profite de cet effort avec une utilisation beaucoup plus simplifiée. Cette démarche est fondamentale. Le programme principal devient alors une boîte à outils, chaque outil étant représenté par une fonction.

Un programme est très souvent développé en équipe. L'identification des fonctions du programme permet de répartir le travail au sein de cette équipe. Chaque programmeur aura ensuite la charge du développement d'une ou de plusieurs fonctions. La tâche du programmeur sera alors de les coder et de les tester, afin de garantir leur fonctionnement.

Définition d'une fonction

Une fonction peut s'apparenter à une opération par l'utilisateur. Une fonction est représentée par un nom. Les opérandes d'une fonction, appelée **paramètres**, sont spécifiés dans une liste entourée de parenthèses, les paramètres étant séparés par des virgules. Le résultat d'une fonction se nomme **valeur de retour** et le type de la valeur de retour s'appelle **type de retour**. Une fonction ne renvoyant pas de valeur a un type de retour **void**, ce qui veut dire qu'elle ne renvoie rien. Dans ce cas particulier, cette fonction s'appelle une procédure. Les actions qu'exécute une fonction sont spécifiées dans le corps de la fonction. Celui-ci, entouré d'accolades, est parfois appelé **bloc de la fonction**. Le type de retour de la fonction suivi du nom de la fonction, la liste des paramètres et le corps de la fonction composent la définition de la fonction.

Si la fonction ne retourne rien, il faut utiliser le mot réservé **void**, c'est alors une procédure.

```
<Type de retour>  <nom de la fonction> (<liste des paramètres>)
{
      Instruction1 ;
      Instruction2 ;
      ....
      return <valeur de retour> ;
}
```

Les parenthèses sont obligatoires même si la fonction n'a pas besoin de paramètres

C'est vous qui décidez du nom de la fonction, c'est un identificateur.

Cette instruction permet de sortir de la fonction et renvoie la valeur vers l'extérieur. L'instruction **return** peut exister plusieurs fois dans une fonction. Si la fonction est une procédure, il n'est pas nécessaire d'utiliser l'instruction **return** ou alors uniquement si l'on désire sortir prématurément. Dans ce cas, il ne faut pas spécifier de valeur de retour

Bloc de la fonction, délimitée par les accolades.

Nous allons mettre en œuvre une fonction qui calcule les puissances entières de la forme : $z = y^x = \underbrace{y.y....y}_{X \text{ fois}}$ avec $y^0=1$
Nous appellerons la fonction du nom de **puissance** à laquelle nous associerons deux paramètres relatifs à **x** et **y**. Cette fonction doit renvoyer une valeur pour **z** puisse la récupérer.

```
//-------------------------------------
double puissance(double y, unsigned x)
{
  double resultat = y;
  if (x==0) return 1.0;
  for (int i=1; i<x; i++)
    resultat *= y;
  return resultat;
}
//-------------------------------------
int main()
{
  int binaire = puissance(2, 10);
  return 0;
}
//-------------------------------------
```

Chaque paramètre doit être séparé des autres par une virgule. Chaque paramètre doit également posséder un nom précédé de son type.

Il est possible de déclarer des variables au sein de la fonction. Elles sont appelées variables locales.

Il peut y avoir plusieurs **return**. Comme la fonction possède une valeur de retour, la valeur renvoyée doit avoir un type compatible avec le type de retour.

Utilisation de la fonction. Pour pouvoir être utilisée, une fonction doit être connue et donc être définie au préalable avant la fonction principale main.
binaire ← 1024 (2 x 2 x 2 x.....x 2) 10 fois.

Déclaration des fonctions

<Type de retour> <nom de la fonction> (<liste des paramètres>) ← Signature d'une fonction

Avant de pouvoir être utilisée (être appelée), il est nécessaire pour le compilateur, de connaître la signature d'une fonction. Cette signature informe le compilateur du type des paramètres et du résultat attendu de la fonction. A l'aide de ces données, le compilateur peut contrôler si le nombre et le type des paramètres d'une fonction sont corrects.

Pour que cette signature soit connue avant l'utilisation de la fonction, il est nécessaire qu'elle soit placée avant l'appel de cette fonction. Il existe deux façons de procéder.

1. La fonction se situe dans le même fichier, auquel cas, il suffit de placer le texte de sa définition avant le texte de son appel comme nous venons de le faire. L'appel de la fonction peut être dans la fonction principale main ou tout autre fonction, le tout c'est que cette dernière soit définie après la fonction appelée.
2. Vous décidez malgré tout de placer votre définition de fonction après la fonction principale ou bien même, vous placez la définition de la fonction dans un autre fichier source. Il est alors nécessaire dans ces cas là, de proposer ce que l'on appelle une déclaration pour avoir la signature requise.

Une déclaration de fonction consiste à placer la signature de la fonction suivie d'un point virgule sans le bloc de définition. Par ailleurs, il est possible d'omettre le nom des paramètres, le compilateur vérifie uniquement leurs types pour permettre la vérification de compatibilité lors de l'appel. Il peut être toutefois judicieux de nommer quand même ces paramètres afin d'informer l'utilisateur de se que l'on attend à priori comme valeurs possibles. Pour la définition de la fonction, par contre, il sera impératif de nommer les paramètres, sinon il serait impossible de les atteindre.

```
//--------------------------------------
double puissance(double y, unsigned x);
//--------------------------------------
int main()
{
  int binaire = puissance(2, 10);
  return 0;
}
//--------------------------------------
double puissance(double y, unsigned x)
{
  double resultat = y;
  if (x==0) return 1.0;
  for (int i=1; i<x; i++)
    resultat *= y;
  return resultat;
}
```

Déclaration de la fonction placée avant la fonction principale main qui exécute l'appel. Remarquez la présence du « ; ».

Appel de la fonction. A ce moment là, la signature est connue par la déclaration et nous pouvons donc l'utiliser correctement.

Définition de la fonction. Cette partie est bien sûr absolument nécessaire. Lorsque que la fonction **main** fait son appel, elle se connecte à ce niveau là pour effectivement réaliser le traitement. Remarquez la présence des accolades.

Quelques déclarations de fonction	*Commentaires associés*
int longueurChaine(char *) ;	longueurChaine attend comme paramètre une chaîne de caractères et renvoi la longueur sous forme d'un entier. Le nom du paramètre n'est pas spécifié.
void copieChaine(char *source, char *destination) ;	copieChaine est une procédure puisque, à priori, elle ne renvoie pas de valeur, si ce n'est au travers de son dernier paramètre puisque c'est un pointeur. Le nom des paramètres devient indispensable pour mieux maîtriser la fonction et surtout pour savoir dans quel ordre on doit placer les arguments.
void affiche(void) ; ou void affiche() ;	C'est une simple procédure sans paramètre, le but étant de morceler le programme en plusieurs parties pour avoir une meilleure lecture et une meilleure fiabilité.
int saisieClavier() ;	Cette fonction ne prend pas de paramètres, mais retourne une valeur entière qui peut être récupérée par une variable de même type.
int factoriel(int) ;	Fonction classique. Le nom du paramètre n'est pas indispensable. Nous comprenons parfaitement le rôle de la fonction.

Appel d'une fonction

Lors de l'appel d'une fonction il y a suspension de l'exécution de la fonction en cours. On « saute » à l'exécution de la fonction appelée. Quand l'évaluation de la fonction appelée est terminée, la fonction suspendue reprend son exécution à l'endroit qui suit immédiatement l'appel. L'exécution d'une fonction se termine une fois exécutée la dernière instruction du corps de la fonction ou quand une instruction **return** est rencontrée dans le corps de la fonction.

```
void Affiche();
int Calcul(int valeur);

void main()
{
   int temp,resultat;
   char caract;
   float var;
   ...
   resultat = Calcul(temp);
   ...
   ...
   Affiche();
   ...
   ...
   ...
}
```

Lors de l'appel de **Calcul()**, l'exécution de la fonction principale est interrompue. La fonction **Calcul()** est alors exécutée jusqu'à la rencontre de l'instruction **return**. La fonction principale reprend alors son cours et récupère la valeur retournée pour la placer dans la variable **resultat**.

```
int Calcul(int valeur)
{
   ...
   return quelqueChose;
}
void Affiche()
{
   ...
}
```

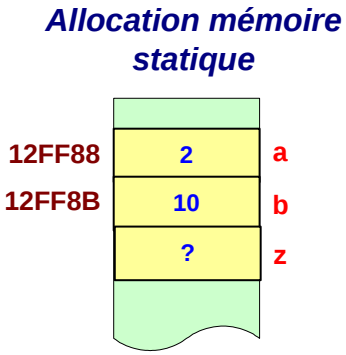
Passage des arguments et variables locales

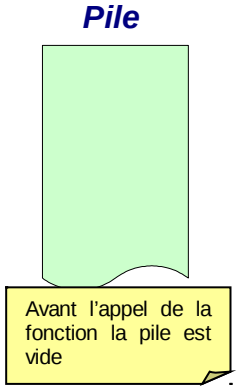
Les fonctions utilisent un espace d'allocation de mémoire située sur la pile d'exécution du programme. Cet espace d'allocation reste associé à la fonction jusqu'à ce que celle-ci se termine. Dès lors, l'espace devient automatiquement disponible pour être réutilisé.

Chaque paramètre de fonction, ainsi que les variables internes, sont stockés sur cet espace d'allocation. Ces deux valeurs sont alors appelés, variables locales. Cette pile est différente de l'allocation mémoire statique, ce qui sous-entends que les valeurs des arguments passées à la fonction vont être copiées dans les paramètres et se retrouvent donc sur la pile (si c'est un passage par valeur – voir plus loin).

Les changements effectués sur ces variables locales (donc sur la pile), ne sont pas répercutées sur les valeurs des arguments. Chaque entité possède son propre espace mémoire. Une fois la fonction terminée, l'espace d'allocation de la pile est supprimée pour cette fonction, et donc, les valeurs locales sont définitivement perdues. Les valeurs locales sont donc des variables dynamiques qui possèdent, malgré tout, une identité, c'est-à-dire un nom.

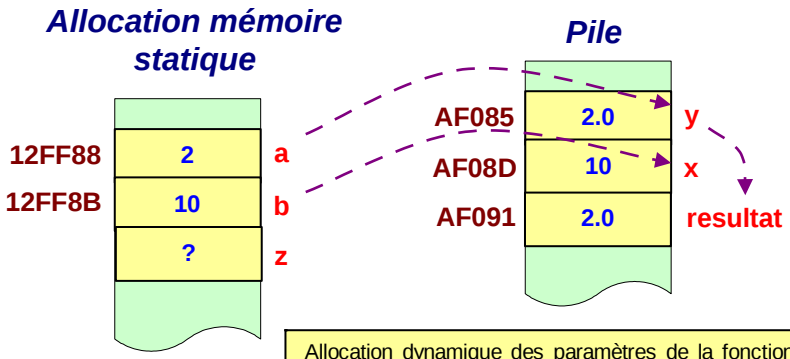
```
//-----------------------------------------
double puissance(double y, unsigned x);
//-----------------------------------------
int main()
{
  int a = 2, b = 10;
  int z;
  z = puissance(a, b);
  return 0;
}
//-----------------------------------------
double puissance(double y, unsigned x)
{
  double resultat = y;
  if (x==0) return 1.0;
  for (int i=1; i<x; i++) resultat *= y;
  return resultat;
}
//-----------------------------------------
```





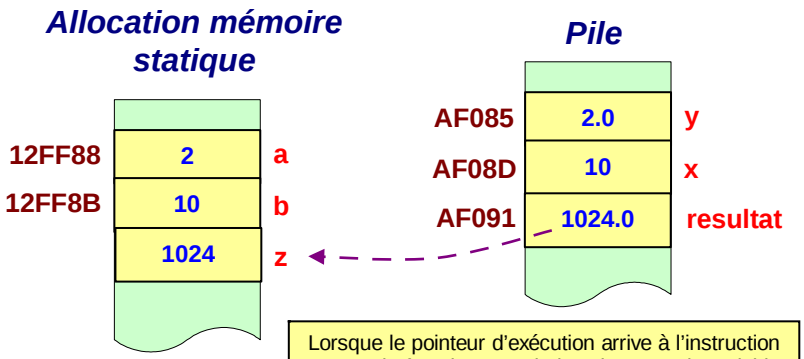
Prochaine ligne d'exécution

```
//-----------------------------------------
double puissance(double y, unsigned x);
//-----------------------------------------
int main()
{
  int a = 2, b = 10;
  int z;
  z = puissance(a, b);
  return 0;
}
//-----------------------------------------
double puissance(double y, unsigned x)
{
  double resultat = y;
  if (x==0) return 1.0;
  for (int i=1; i<x; i++) resultat *= y;
  return resultat;
}
//-----------------------------------------
```



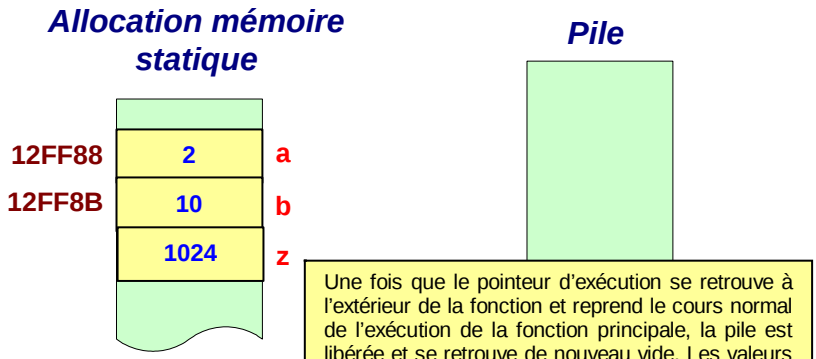
Allocation dynamique des paramètres de la fonction ainsi que la variable locale **resultat**. Ensuite, le système copie les valeurs des arguments **a** et **b** vers leurs paramètres respectifs **y** et **x**. Enfin, **resultat** prend la valeur de **y**.

```
//-----------------------------------
double puissance(double y, unsigned x);
//-----------------------------------
int main()
{
  int a = 2, b = 10;
  int z;
  z = puissance(a, b);
  return 0;
}
//-----------------------------------
double puissance(double y, unsigned x)
{
  double resultat = y;
  if (x==0) return 1.0;
  for (int i=1; i<x; i++) resultat *= y;
  return resultat;
}
//-----------------------------------
```



Lorsque le pointeur d'exécution arrive à l'instruction **return**, la fonction renvoie la valeur vers la variable qui est en connexion par le signe d'affectation proposé lors de l'appel, donc ici **z**.

```
//-----------------------------------
double puissance(double y, unsigned x);
//-----------------------------------
int main()
{
  int a = 2, b = 10;
  int z;
  z = puissance(a, b);
  return 0;
}
//-----------------------------------
double puissance(double y, unsigned x)
{
  double resultat = y;
  if (x==0) return 1.0;
  for (int i=1; i<x; i++) resultat *= y;
  return resultat;
}
//-----------------------------------
```



Une fois que le pointeur d'exécution se retrouve à l'extérieur de la fonction et reprend le cours normal de l'exécution de la fonction principale, la pile est libérée et se retrouve de nouveau vide. Les valeurs locales sont définitivement perdues.

Transmission par valeur et par variable

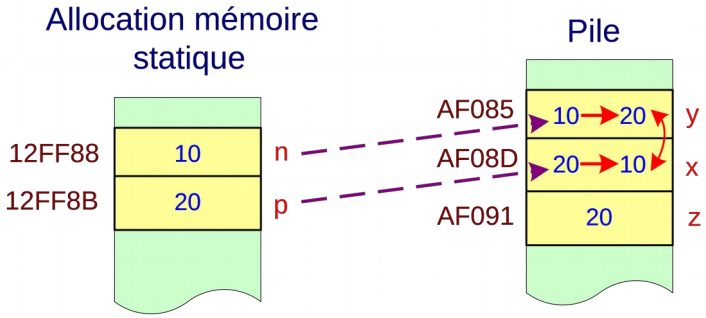
Nous venons de voir, que le passage de paramètres permet à une fonction de pouvoir traiter des données qui ne sont pas définies dans son corps. Ces données sont passées à la fonction lors de son appel. Il existe globalement deux techniques de passage de paramètres :

- **Soit par valeur**, et c'est la technique que nous venons d'utiliser.
- **Soit par variable,** ce qui permet dans ce cas là, de se connecter directement (ou indirectement) aux variables de la fonction principale, c'est-à-dire aux arguments.

Transmission par valeur :
C'est le type de transmission qui est le plus couramment utilisé. Avec ce système, la fonction manipule les copies locales des arguments. Ainsi, les fonctions n'obtiennent que les valeurs de leurs paramètres passés et elles n'ont pas accès au contenu des variables elles-mêmes. Les paramètres d'une fonction sont des variables locales qui sont initialisées automatiquement par les valeurs indiquées par les arguments lors de l'appel.

A l'intérieur de la fonction, nous pouvons donc changer les valeurs des paramètres sans influencer les valeurs originales dans les fonctions appelantes, ce qui procure une protection maximale pour les arguments. Dans certain cas Toutefois, il peut être nécessaire d'atteindre l'argument lui-même pour permettre le changement de sa valeur. C'est là qu'intervient la transmission par variable.

```
//-------------------------
void echange(int x, int y)
{
  int z=x;
  x=y; y=z;
}
//-------------------------
int main()
{
  int n=10, p=20;
  echange(n, p);
  return 0;
}
//-------------------------
```



Dans cet exemple, nous proposons de fabriquer une fonction qui permet d'échanger le contenu des variables.

En prenant le passage par valeurs comme c'est le cas ici, les seuls échanges proposés se situent au niveau des variables locales à la fonction, sans qu'il y ait de répercutions sur les arguments **n** et **p**. Dans la plupart des cas, c'est très bien, puisque

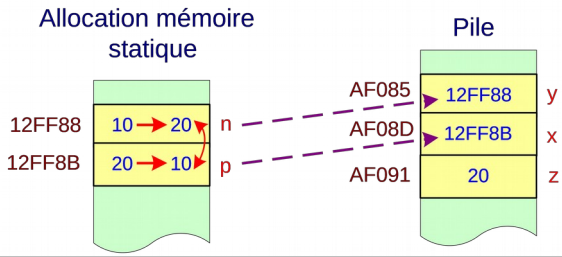
les arguments sont protégés de toute mauvaise utilisation. Dans le cas qui nous préoccupe, cela ne correspond spécialement pas à notre attente, puisque nous désirons échanger les valeurs entre les deux arguments.

Transmission par variable :
Le passage par variable permet à la fonction appelée de pouvoir modifier le contenu de la variable passée en paramètre. Il existe deux techniques pour résoudre ce problème :

- soit **indirectement** en utilisant les **pointeurs**.
- soit **directement** en utilisant les **références**.

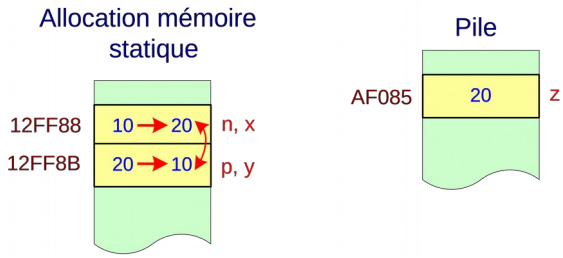
Envisageons les deux cas de figure sur notre problème pour arriver à réaliser l'échange proposé et commençons par la technique des pointeurs.

```
//---------------------------
void echange(int *x, int *y)
{
  int z = *x;
  *x = *y; *y = z;
}
//---------------------------
int main()
{
  int n=10, p=20;
  echange(&n, &p);
  return 0;
}
//---------------------------
```



L'échange s'effectue bien sur les bonnes variables, c'est-à-dire sur les arguments de la fonction principale. Toutefois, la syntaxe demeure relativement lourde puisqu'il faut déréférencer systématiquement. Enfin, à l'appel de la fonction, il est nécessaire de donner les adresses des variables à traiter puisqu'il s'agit de pointeurs.

```
//---------------------------
void echange(int &x, int &y)
{
  int z = x;
  x = y; y = z;
}
//---------------------------
int main()
{
  int n=10, p=20;
  echange(n, p);
  return 0;
}
//---------------------------
```



Ici aussi, l'échange entre les arguments s'effectue bien. De plus, la syntaxe est extrêmement simple. Par ailleurs, les références se connectent directement sur les arguments (alias), ce qui évite d'utiliser des variables locales supplémentaires. Le gain de temps et la souplesse d'emploi sont prépondérants.

Conclusion

A l'aide de ces différents scénarii, nous pouvons presque conclure que :

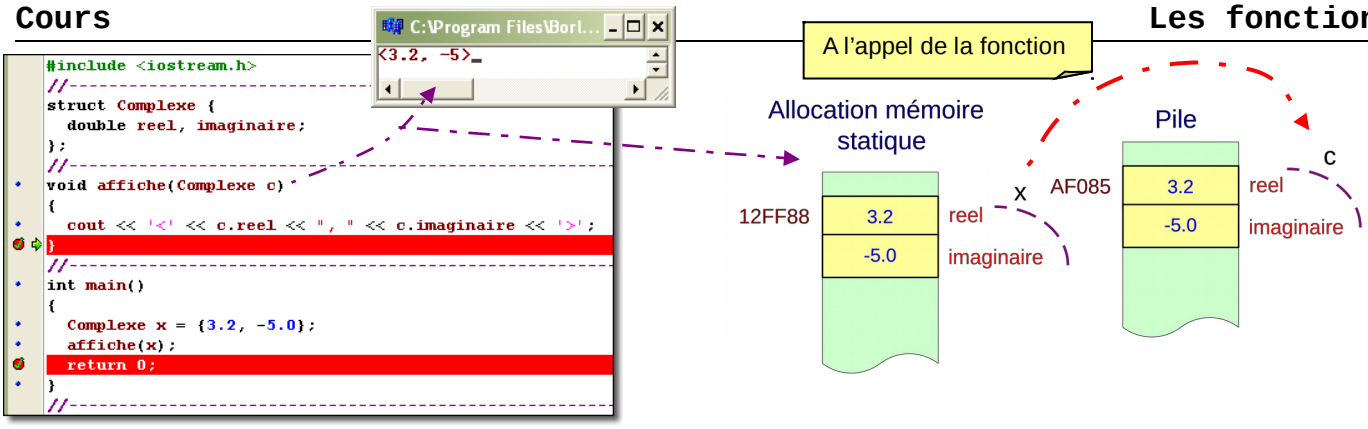
- Lorsque nous devons récupérer une valeur sans changer le contenu de l'argument, il faut alors proposer une transmission par valeur, et il suffit alors de faire une déclaration classique des paramètres.
- Lorsque nous devons modifier directement le contenu de l'argument, il faut cette fois-ci proposer une transmission par variable en prenant si possible une référence pour que l'argument soit directement connecté.

La plupart du temps, ces principes demeurent vrais. Toutefois, il existe des situations où nous devons réaliser une analyse plus fine pour répondre parfaitement à l'attente de l'utilisateur. En fait, tout dépend du type d'argument.

Les types d'argument

Jusqu'à présent, nous nous sommes contenté de prendre des types simples comme paramètres de fonction. Dans ce chapitre, nous allons nous intéresser à des types définis par l'utilisateur ainsi que les tableaux et les chaînes de caractères.

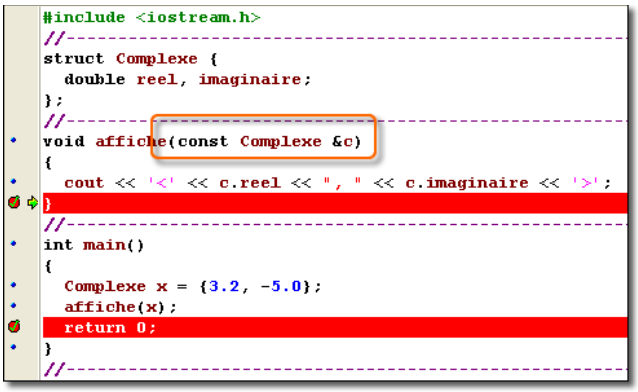
Les structures :
Nous pouvons utiliser les principes évoqués plus haut au sujet des types de transmission. La déclaration des paramètres est alors identique à la déclaration que nous réalisons lorsque nous avons besoin de variables classiques. Prenons l'exemple de la structure sur les nombres complexe pour visualiser le comportement au sein de la pile.

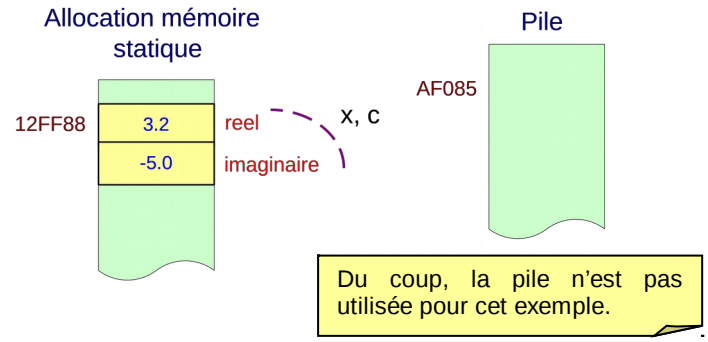


La fonction affiche a juste besoin de récupéré la valeur de la structure. C'est pour cela, qu'à priori, nous utilisons le passage par valeur. Le résultat du programme est d'ailleurs le comportement attendu. Toutefois, avec ce système, nous copions tout le contenu de la structure sur la pile. Imaginons que nous ayons une structure volumineuse avec pas moins d'une dizaine de champs. Dans ce cas de figure, nous prenons alors une bonne partie de la pile pour récupérer tous ces champs. Par ailleurs, la copie demande un temps considérable par rapport à l'utilisation que nous en faisons. Il est alors judicieux de proposer une autre solution plus adaptée.

Plutôt que de copier tout le contenu de la structure, il serait préférable de se connecter directement à l'argument **x** en utilisant le principe des références. Cette démarche est judicieuse puisque nous obtenons un gain de temps considérable. Nous avons toutefois un problème. En effet, lorsque nous utilisons une référence sur un argument, cela veut normalement dire que nous désirons modifier son contenu, et l'utilisateur s'attend justement à ce comportement. Dans cet exemple, ce n'est pas ce que nous voulons.

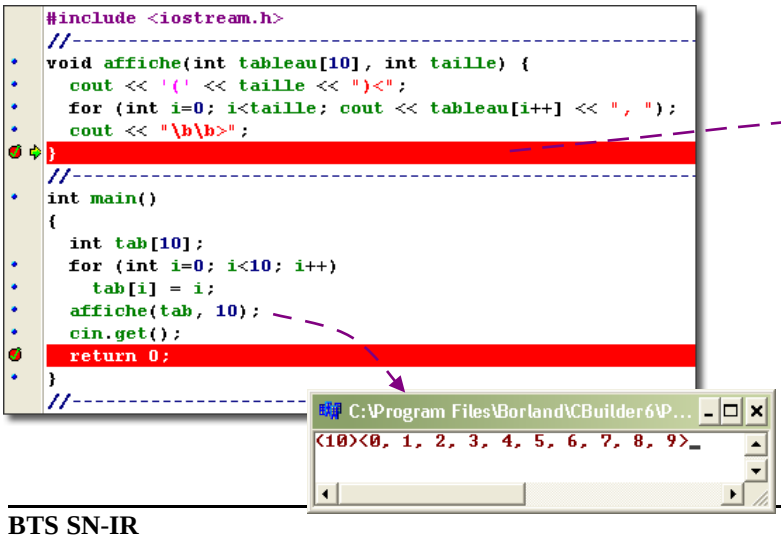
Il faut alors impérativement avertir l'utilisateur que nous désirons juste lire l'argument sans modifier son contenu et que malgré tout nous proposons une connexion directe par une référence par soucis de performance. Nous avons déjà traité ce genre de principe. Une lecture sans modification du contenu correspond à une constante. Il faudra donc proposer une référence constante.

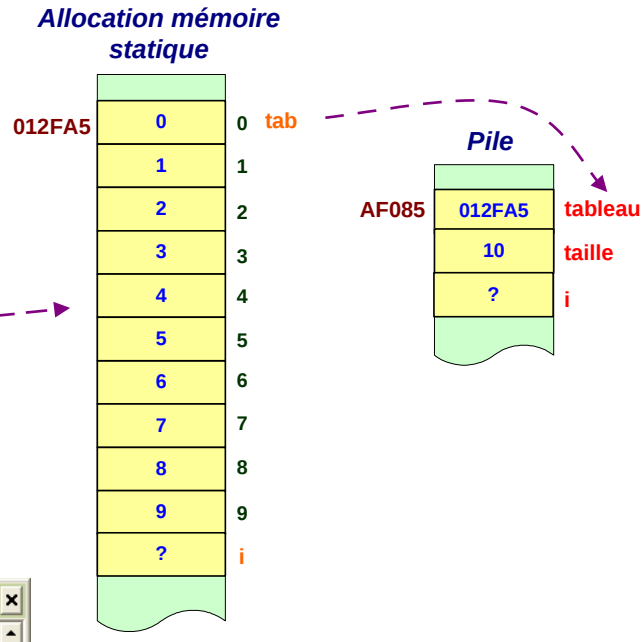




Les tableaux :

Encore une fois, n'oubliez pas qu'un tableau est un pointeur constant, et que du coup, pour ce type de variable, la transmission par valeur n'existe pas. C'est donc une transmission par variable qui est proposée et qui s'effectue à l'aide de pointeur et non pas une référence.





Dans l'exemple proposé, nous avons une fonction qui permet d'afficher un tableau d'entier. Nous passons donc un tableau en paramètre, seulement, nous récupérons uniquement l'adresse de l'argument (puisqu'il s'agit d'un pointeur). Du coup, la dimension d'un tableau n'est plus significative puisque seule l'adresse est passée. Les trois déclarations suivantes sont alors équivalentes :

```
void affiche(int tableau[10], int taille) ;
void affiche(int tableau[], int taille) ;
void affiche(int *tableau, int taille) ;
```

C++11

```
void affiche(vector<int> tableau) ;          // déclaration
....
affiche( {12, 5, -89, 18} ) ;      // utilisation de la fonction
```

Comme la dimension n'est pas connue lorsque nous passons un tableau en paramètre, il est alors nécessaire de rajouter un paramètre supplémentaire pour réellement connaître la taille effective du tableau.

Pour résumer, lorsque nous passons un tableau en paramètre d'une fonction, il s'agit d'une transmission par variable de type pointeur ce qui amène des conséquences. Comme toutes les transmissions par variable, les changements effectués sur un tableau dans la fonction appelée sont faits sur l'argument lui-même, non sur la copie locale (puisqu'il n'y a pas de copie locale de l'ensemble du tableau).

```
#include <iostream.h>
//-----------------------------------------------------
void affiche(const int tableau[], int taille) {
  cout << "( " << taille << ")<";
  for (int i=0; i<taille; cout << tableau[i++] << ", ");
  cout << "\b\b>";
}
//-----------------------------------------------------
int main()
{
  int tab[10];
  for (int i=0; i<10; i++)
    tab[i] = i;
  affiche(tab, 10);
  cin.get();
  return 0;
}
//-----------------------------------------------------
```

Comme pour les structures, si vous n'avez pas l'intention de modifier les éléments du tableau, il faut avertir l'utilisateur de la fonction en proposant l'attribut **const** sur le paramètre de type tableau. Avec cette nouvelle signature, toute tentative de modification d'un des éléments du tableau se traduit par une erreur de compilation.

Les chaînes de caractères :

Comme vous le savez, une chaîne de caractères est un cas particulier du tableau. Toute la discussion sur les tableaux s'applique intégralement sur les chaînes de caractères et voici juste un exemple pour illustrer ces propos.

```
#include <iostream.h>
//-----------------------------------------------------
int longueurChaine(const char chaine[]) {
  int longueur = -1;
  while (chaine[++longueur]);
  return longueur;
}
//-----------------------------------------------------
int main()
{
  char *message = "bonjour";
  int nombre = longueurChaine(message);
  cout << "Il y a "<< nombre <<" caracteres dans "<< message;
  return 0;
}
//-----------------------------------------------------
```

```
C:\Program Files\Borland\CBuilde...
Il y a 7 caracteres dans bonjour
```

Les arguments par défaut

Il est assez fréquent qu'un des paramètres d'une fonction prenne souvent la même valeur. Une fonction peut gérer ce genre de situation en spécifiant un argument par défaut pour un de ses paramètres, ou plus, en utilisant la syntaxe d'initialisation à l'intérieur de la liste des paramètres. Une fonction donnant un argument par défaut pour un paramètre peut être invoqué avec ou sans argument pour ce paramètre. Si un argument est fourni, il remplace la valeur par défaut ; sinon, l'argument par défaut est utilisé.

A titre d'exemple, créons une fonction qui affiche un caractère quelconque à l'écran. Sans précision particulière, elle doit afficher le caractère 'espace'.

```
#include <iostream.h>
//-----------------------------------------------------
void affiche(char caractere = ' ') {
  cout << caractere;
}
//-----------------------------------------------------
int main()
{
  affiche('A');
  affiche();
  affiche('B');
  return 0;
}
//-----------------------------------------------------
```

```
C:\Pro...
A B
```

Dans cet appel, 'A' remplace l'argument par défaut ' ', et c'est bien 'A' qui est affiché.

Dans cet appel, aucun argument n'est proposé. C'est alors la valeur par défaut qui est fourni à la fonction d'affichage

Lorsqu'une déclaration prévoit des valeurs par défaut, les arguments concernés doivent obligatoirement être les derniers de la liste. La déclaration suivante est interdite :

double exemple(int = 5, long, int = 3) ; // notez qu'il s'agit d'une déclaration de fonction et non d'une définition.

En fait, une telle interdiction relève du pur bon sens. En effet, si cette déclaration était acceptée, l'appel suivant :

exemple(10, 20) ; pourrait être interprété aussi bien comme :
exemple(5, 10, 20) ; que comme : **exemple(10, 20, 3) ;**

Une partie de la conception d'une fonction avec des arguments par défaut consiste à trier les paramètres à l'intérieur de leur liste afin que ceux, amenés le plus souvent à recevoir une valeur spécifiée par l'utilisateur, soient définis en premier et que ceux devant fréquemment utiliser les arguments par défaut le soient en dernier.

```
#include <iostream.h>
//-----------------------------------------
void affiche(char = ' ');
//-----------------------------------------
int main()
{
  affiche('A');
  affiche();
  affiche('B');
  return 0;
}
//-----------------------------------------
void affiche(char caractere) {
  cout << caractere;
}
//-----------------------------------------
```

Attention

L'argument par défaut d'un paramètre ne peut être spécifié qu'une seule fois dans un fichier. En conséquence, si nous avons à la fois une déclaration et une définition de fonction, la spécification de l'argument par défaut doit plutôt être placé sur la déclaration puisque c'est elle qui représente la fonction.

La sur-définition de fonctions

Imaginons que nous ayons besoin de fabriquer une fonction qui détermine la valeur minimale entre deux entités. Il vient donc tout de suite à l'esprit qu'il s'agit de déclarer une fonction que nous pouvons simplement appeler ***minimum()***. Une question se pose toutefois quant au choix des types des paramètres de cette fonction. Il faudrait que cette fonction puisse accepter aussi bien des réels que des entiers. Le meilleurs choix semble être des paramètres de type réels, puisque les réels englobent les entiers.

```
#include <iostream.h>
//-----------------------------------------
double minimum(double x, double y)
{
  if (x<=y) return x;
  else return y;
}
//-----------------------------------------
int main()
{
  double h = 3.2, k = -8.6;
  int a = 3, b = -9;
  double i = minimum(h, k);    // arguments réels
  int j = minimum(a, b);       // arguments entiers
  i = minimum(-5.0, 2.3);      // arguments réels
  j = minimum(5, 3);           // arguments entiers
  return 0;
}
//-----------------------------------------
```

Toutefois, cette situation n'est pas satisfaisante. En effet, si nous proposons à une telle fonction des entiers comme arguments, le système devra faire une conversion pour passer des entiers vers les réels et effectuer ensuite les calculs nécessaires, ce qui représente une perte de temps considérable vu la dimension de cette fonction.

Du coup, il peut être judicieux de proposer une fonction par type d'arguments ; une fonction minimum pour les entiers et une fonction minimum pour les réels. Maintenant se pose le problème de la désignation de chacune de ces fonctions. On pourrait par exemple appeler la première fonction ***minimumInt()*** et la suivante ***minimumDouble()***. L'écriture est très lourde, et de plus nous ne sommes par sûr de nous en souvenir.

Sur-définition de fonctions :

Tout ce que nous désirons, c'est avoir le minimum de deux valeurs, et qu'à l'utilisation, le système soit capable de faire la différence entre les entiers et les réels sans se soucier de donner un nom spécifique à chacune des fonctions. Heureusement, le langage C++ permet d'avoir cette approche. C'est ce qui s'appelle la sur-définition (ou alors la surcharge) de fonctions. Deux fonctions sont sur-définies (ou surchargées) si elles ont le même nom, mais disposent d'une liste de paramètres différente.

La fonction minimum() est surchargée puisqu'elle est définie deux fois (avec le même nom) tout en ayant une liste de paramètres différente.

```
#include <iostream.h>
//-----------------------------------------
int minimum(int x, int y)
{
  return x<=y ? x : y;
}
//-----------------------------------------
double minimum(double x, double y)
{
  return x<=y ? x : y;
}
//-----------------------------------------
int main()
{
  double h = 3.2, k = -8.6;
  int a = 3, b = -9;
  double i = minimum(h, k);    // arguments réels
  int j = minimum(a, b);       // arguments entiers
  i = minimum(-5.0, 2.3);      // arguments réels
  j = minimum(5, 3);           // arguments entiers
  return 0;
}
//-----------------------------------------
```

Résolution de surcharge :

La résolution de surcharge est le mécanisme par lequel une fonction sur-définie est choisie plutôt qu'une autre. En effet, comme les fonctions sur-définies portent, par définition, le même nom, le compilateur doit être suffisamment compétent pour sélectionner la bonne fonction au moment de l'appel. Si les arguments proposés correspondent parfaitement à l'une des signatures d'une fonction, l'appel ne posera aucun problème. Par contre, si les arguments ne correspondent à aucune des signatures proposées dans l'ensemble des fonctions sur-définies, il faut

que le compilateur sélectionne la fonction qui correspond le mieux à l'appel. S'il n'arrive pas à choisir, le compilateur provoque alors une erreur de compilation.

Dans tous ces appels, il existe une fonction qui correspond parfaitement aux types des arguments proposés.

Pour cet appel, il n'existe pas de fonction qui corresponde exactement aux arguments proposés de type caractère. Du coup le système sélectionne celle qui correspond le mieux. Comme le caractère est une sous-famille du type entier, le choix se porte sur la fonction :
int minimum(int x, int y) ;
Le compilateur transforme alors les caractères passés à la fonction pour qu'ils deviennent des entiers.

```
#include <iostream.h>
//-----------------------------------------------
int minimum(int x, int y)
{
  return x<=y ? x : y;
}
//-----------------------------------------------
double minimum(double x, double y)
{
  return x<=y ? x : y;
}
//-----------------------------------------------
int main()
{
  double h = 3.2, k = -8.6;
  int a = 3, b = -9;
  double i = minimum(h, k);     // arguments réels
  int j = minimum(a, b);        // arguments entiers
  i = minimum(-5.0, 2.3);       // arguments réels
  j = minimum(5, 3);            // arguments entiers
  char c = minimum('A', 'B');
  j = minimum(5.0, 3);
  return 0;
}
//-----------------------------------------------
```

Pour cette ligne, le compilateur est confronté à un dilemme. En effet, deux solutions sont possibles :
- Soit le premier argument est transformé en valeur entière auquel cas, c'est la fonction minimum avec des paramètres entiers qui est sollicitée.
- Soit le deuxième argument est transformé en valeur réelle auquel cas, c'est la fonction minimum avec des paramètres réels qui est sollicitée.

Le compilateur provoque une erreur de compilation puisqu'il est incapable de résoudre un dilemme.

Récursivité

Une fonction qui s'appelle elle-même, directement ou indirectement, est appelée une fonction récursive. Pour illustrer l'intérêt de la récursivité, nous allons nous intéresser à la fonction factorielle. Toutefois, commençons par une étude plus classique en utilisant une itérative pour résoudre le problème.

Structure itérative :
Rappelons que la fonction factorielle de **n** est :
n ! = n (n-1) (n-2) ..... 3.2.1 avec **0 ! = 1** et **n ∈ N**
Par exemple, la factorielle de **5** est : **5 ! = 5.4.3.2.1 = 120.**

Structure récursive :
Reprenons l'exemple précédent : **5 ! = 5.4.3.2.1** peut s'écrire aussi :
5 ! = 5. (4.3.2.1) = 5. 4 !, autrement dit, la factorielle de 5, c'est cinq fois la factorielle de 4. Finalement, pour définir une factorielle, je fais référence à une autre factorielle. C'est une écriture récursive. D'une façon générale, nous avons : **n ! = n. (n-1) !** avec **0 ! = 1**

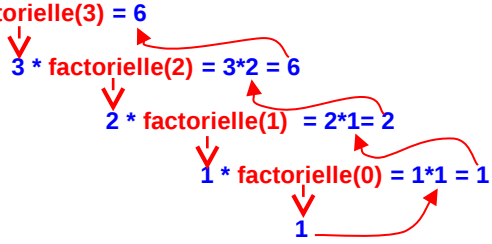
```
#include <iostream.h>
//-----------------------------------------------
int factorielle(unsigned n)
{
  int resultat = 1;
  for (int i=2; i<=n; resultat *= i++);
  return resultat;
}
//-----------------------------------------------
int main()
{
  int y = factorielle(5);    // y <-- 120
  y = factorielle(0);        // y <-- 1
  return 0;
}
//-----------------------------------------------
```

```
//-----------------------------------------------
int factorielle(unsigned n)
{
  return n==0 ? 1 : n * factorielle(n-1);
}
//-----------------------------------------------
int main()
{
  int y = factorielle(5);    // y <-- 120
  y = factorielle(0);        // y <-- 1
  return 0;
}
//-----------------------------------------------
```

Une fonction récursive doit toujours définir une condition d'arrêt ; sinon la fonction s'appelle à l'infini. On parle alors d'erreur de récursivité infinie.

Une fonction récursive s'exécute souvent plus lentement que la version itérative équivalente à cause du surcoût associé à l'appel d'une fonction. Cependant, la version récursive est toujours beaucoup plus concise et donc plus facilement compréhensible.

Si nous avons : **y = factorielle(3) = 6**



Fonctions en ligne (Macros)

```
//------------------------
int minimum(int x, int y)
{
  return x<=y ? x : y;
}
//------------------------
int main()
{
  int z = minimum(3, -5);
  z = minimum(-2, 8);
  return 0;
}
//------------------------
```

Fonction normale :
Il est très fréquent d'écrire de petites fonctions de quelques instructions seulement, qui sont essentiellement des commodités d'écriture. Il est en effet plus facile d'écrire, par exemple, ***minimum(a, b)***, plutôt que ***x<=y ? x : y***. Par ailleurs, si un jour nous changeons l'implémentation de cette fonction, cela n'aura aucune répercussion pour les utilisateurs. Ce serait plus problématique si cette fonction n'existait pas, il faudrait en effet modifier plusieurs fois dans l'application les lignes équivalente à cet appel de fonction. Si jamais vous avez **300** occurrences de ce type, il serait fastidieux de tout remettre en place.

Il existe cependant un sérieux inconvénient à utiliser tel quel des fonctions aussi petites.

- En effet, un appel de fonction casse l'effet de séquence dans un programme. Les microprocesseurs actuels anticipent les instructions et préparent déjà l'exécution de l'instruction suivante avant même quelle ne soit réclamée. En cas d'appel de fonction, l'instruction suivante n'est pas celle qui était prévue. Tout le travail d'anticipation se révèle inutile et le chargement de l'instruction à l'intérieur du microprocesseur doit recommencer à zéro.
- N'oubliez pas également que les deux arguments doivent être copiés sur la pile, et même plus, les registres du microprocesseur doivent être sauvegardés puisque le programme continu son exécution à un nouvel emplacement.
- Une fois que tout ceci est réalisé, le corps de la fonction est enfin traité. Lorsque la fonction a fini son traitement, elle doit restituer la pile ainsi que les registres du microprocesseur.

On se rend compte du coup que l'appel d'une fonction prend du temps rien que pour se connecter. Si dans le corps, nous avons beaucoup d'instructions, ou bien si nous avons des itératives, ce temps de connexion est infime par rapport au temps passé pour traiter le contenu. Par contre, pour de toutes petites fonction, le système passe pratiquement tout son temps à se connecter et se déconnecter.

Fonction inline :
Une fonction « en ligne » permet de palier à tous ces problèmes. Une fonction « en ligne » est développée au moment de la compilation à chaque endroit du programme ou elle est invoquée. Une fonction « en ligne » utilise le préfixe ***inline*** devant la déclaration ou la définition normale de la fonction.

```
//------------------------
inline int minimum(int x, int y)
{
  return x<=y ? x : y;
}
//------------------------
int main()
{
  int z = minimum(3, -5);
  z = minimum(-2, 8);
  return 0;
}
//------------------------
```

```
int main()
{
  int z = 3 <= -5 ? 3 : -5;
  z = -2 <= 8 ? -2 : 8;
  return 0;
}
//------------------------
```

Lorsque le compilateur parcoure le code, à chaque endroit où minimum est trouvé, il le remplace par le contenu de la fonction.

Par ce système, nous n'avons plus de mécanisme d'appel de fonction, il s'agit tout simplement du changement d'un texte par un autre. Toutefois, ces fonctions « en ligne » ne doivent être utilisée que pour de toute petites fonctions, sinon, la taille de l'exécutable pourrait devenir relativement conséquente.

Manipulation des options de la ligne de commande dans la fonction main()

Lorsque nous demandons un nouveau projet, vous remarquez que le système nous propose couramment, pour la fonction principale main, une signature étendue avec deux paramètres passés à la fonction.

```
//------------------------
int main(int argc, char* argv[])
{
  return 0;
}
//------------------------
```

Ces paramètres sont utiles lorsque dans un programme, on doit passer des options sur la ligne de commande comme, par exemple : **ls –al *.cpp**

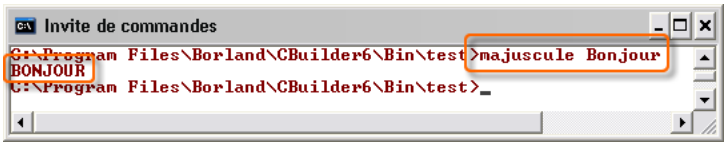
Ici, la commande **ls** (un « programme » qui permet de lister le contenu du répertoire courant) comporte deux arguments, **« -al »** et **« *.cpp »**. Ces arguments sont récupérés par les paramètres de la fonction main correspondant à l'application **« ls »**.

Le paramètre **argc** récupère le nombre d'options sur la ligne de commande, alors que **argv** est le tableau de chaînes de caractères représentant les options de commande séparées par des espaces.

```
#include <iostream.h>
//------------------------
void majuscule(char *chaine)
{
  for (int i=0; chaine[i]; i++)
    if (chaine[i]>='a' && chaine[i]<='z')  chaine[i] += 'A'-'a';
}
//------------------------
int main(int argc, char* argv[])
{
  if (argc==2) majuscule(argv[1]);
  cout << argv[1];
  return 0;
}
//------------------------
```

ls -al *.cpp

argc = 3
argv[0] = " ls "
argv[1] = " –al "
argv[2] = " *.cpp "



Expériences sur différentes utilisations de fonctions élaborées durant ce cours

main.cpp

```cpp
#include <iostream>
#include <string>
#include <vector>
using namespace std;

struct Complexe
{
    double reel, imaginaire;
};

Complexe addition(const Complexe &x, const Complexe &y)
{
    Complexe z;
    z.reel = x.reel + y.reel;
    z.imaginaire = x.imaginaire + y.imaginaire;
    //return z;
    return {x.reel+y.reel, x.imaginaire+y.imaginaire};
}

inline void affiche(const Complexe &c)
{
    cout << '<' << c.reel << ", " << c.imaginaire << ">\n";
}

void affiche(int entiers[], unsigned nombre)
{
    cout << '[';
    for (unsigned i=0; i<nombre; i++) cout << entiers[i] << ' ';
    cout << "\b]\n";
}

void affiche(vector<int> entiers)
{
    cout << '[';
    for (int i=0; i<entiers.size(); i++) cout << entiers[i] << ' ';
//    for (int entier : entiers) cout << entier << ' ';
    cout << "\b]\n";
}

inline int maximum(int x, int y)
{
  return x>=y ? x : y;
}

inline double maximum(double x, double y)
{
  return x>=y ? x : y;
}

int maximum(vector<int> entiers)
{
  int max = 0;
  for (int entier : entiers) if (entier>max) max = entier;
  return max;
}

inline void affiche(string nom, double valeur)
{
  cout << nom << " : " << valeur << endl;
}

int factoriel(int n)
{
  return n==0 ? 1 : n*factoriel(n-1);
}

int main()
{
    Complexe a={2.0, -8.9}, b={-5.2, 6.4};
    affiche(addition(a, b));
    int tableau[] = {5, 3, -89};
    vector<int> entiers = {7, 6,-8, -9, 12, -4};
    affiche(tableau, 3);
    affiche({5, 2, -8, 4});
    affiche("Entier", maximum(5, -3));
    affiche("Réel", maximum(8.3, 9.6));
    affiche("Entiers", maximum(entiers));
    affiche("5!", factoriel(5));
    return 0;
}
```

Transformer une chaîne de caractères en une valeur numérique correspondante

Nous allons élaborer un programme qui permet de retrouver le factoriel d'une valeur proposée en option de la ligne de commande à la suite du nom du programme :

factoriel.cpp

```
#include <iostream>
using namespace std;

/**
 * @brief entier transforme un nombre sous forme de chaîne de caractères
 * en une valeur entière équivalente
 * @param chaine nombre écrit sous forme de chaîne de caractère
 * @return même nombre correspondant à une valeur entière de type int
 */
int entier(char chaine[])
{
  bool negatif = chaine[0]=='-';
  int entier=0, i = negatif ? 1 : 0;
  while (chaine[i])
  {
    entier *= 10;
    entier += chaine[i++]-0x30;
  }
  return negatif ? -entier : entier;
}

unsigned factoriel(unsigned n)
{
  return n==0 ? 1 : n*factoriel(n-1);
}

int main(int nombre, char *arguments[])
{
  if (nombre==2)
  {
    int n = entier(arguments[1]);
 // int n = stoi(arguments[1]);

    if (n<0) cout << "ATTENTION, nombre négatif" << endl;
    else cout << n << "! = " << factoriel(n) << endl;
  }
  else cout << "Le nombre d'argument n'est pas valide" << endl;
  return 0;
}
```

Dans ce programme, la fonction ***premier()*** permet de retourner une valeur entière signée à partir d'une valeur numérique exprimée sous forme de chaîne de caractères passée en argument de la fonction. L'objectif de cette fonction est de voir comment réaliser une analyse de chaîne de caractères et de retrouver une valeur particulière. Il faut savoir qu'une telle fonction existe déjà et se nomme ***stoi()*** ***(string to int)***. De la même façon, il existe une fonction qui permet de retrouver une valeur réelle exprimée sous forme de chaîne de caractères et qui se nomme ***stod()*** ***(string to double)***. Des exemples sur l'utilisation de ces fonctions sont proposés page 14 à la fin de cette étude.

Les flux et les fichiers

Nous nous intéressons maintenant à la mise en œuvre et à la gestion des fichiers. Des classes sont spécialisées dans ce domaine. De plus, elles héritent directement ou indirectement de ***istream*** et ***ostream***, (classes représentant les objets que nous connaissons bien, respectivement ***cin*** et ***cout***), ce qui permet notamment d'utiliser les opérateurs de redirection pour écrire ou lire directement dans un fichier (de la même façon que ***cin*** et ***cout***). Voici le nom de ces trois classes :

Classes spécifiques à la gestion des fichiers

1. ***ofstream*** *: flot de sortie associé à un fichier. Permet d'écrire des informations* ***de type quelconque*** *dans un fichier.*
2. ***ifstream*** *: flot d'entrée associé à un fichier. Permet de lire des informations* ***de type quelconque*** *issues d'un fichier.*
3. ***fstream*** *: flot bidirectionnel associé à un fichier. Permet d'écrire ou de lire dans un fichier.*

Pour utiliser ou générer un fichier, il suffit de créer un objet associé au mode d'ouverture voulu en spécifiant le nom du fichier désiré en argument du constructeur. Dès que l'objet est créé, le fichier s'ouvre automatiquement suivant le mode donné par le type de la classe sélectionné. Enfin, lorsque l'objet est détruit, le fichier est automatiquement fermé, sans spécification particulière.

Entre ces deux phases, vous pouvez, suivant le cas, inscrire ou lire des informations en utilisant simplement les opérateurs de redirection. Comme ces opérateurs ont été redéfinis pour tous les types prédéfinis, il est donc possible d'envoyer ou de lire n'importe quelle information, comme des valeurs entières, des valeurs réelles, des chaînes de caractères, etc.

Les différents modes d'ouverture

Le mode d'ouverture est défini par un mot d'état ***open_mode***, dans lequel chaque bit correspond à une signification particulière. La valeur correspondant à chaque bit est définie par des constantes déclarées dans la classe de base ***ios***. Pour activer plusieurs bits, il suffit de faire appel à l'opérateur ou logique « | ».

Bits d'ouverture de fichier dans le mot d'état open_mode

- ***in*** *:Ouverture en lecture. Le fichier doit exister.*
- ***out*** *:Ouverture en écriture. Écrase l'ancien contenu. Si le fichier n'existe pas, il est automatiquement créé.*
- ***app*** *: Ouverture en ajout de données (écriture en fin de fichier).*
- ***trunc*** *: Si le fichier existe, son contenu est définitivement perdu.*
- ***binary*** *: Pour les précédents modes, l'information été systématiquement transformée en une suite de caractères. Au moment du transfert vers le fichier, ces valeurs subissent une transformation pour devenir une suite de caractères. Il est alors possible de contrôler le contenu du fichier avec* ***un simple éditeur de texte****. Toutefois, vous pouvez désirer conserver le type original et donc demander à avoir* ***un stockage sous forme binaire****. C'est ce que permet ce mode d'ouverture.*

Exemples de mise en œuvre

Le programme suivant permet d'enregistrer dans un fichier texte « notes.txt » un ensemble de notes saisies au clavier. Ce fichier est ensuite interprété dans un autre programme pour calculer automatiquement la moyenne. Le résultat est affiché sur un terminal spécifique au lieu de prendre la sortie standard.

enregistrer.cpp

```
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int main()
{
  ofstream fichier("/home/manu/notes.txt");
  unsigned nombre;
  cout << "Nombre de notes : "; cin >> nombre;

  fichier << nombre << endl; // élément séparateur '\n'.
  // Il faut un caractère blanc (' ', '\t', '\n') pour séparer chacune des données.

  for (unsigned i=0; i<nombre; i++)
  {
    double saisie;
    cout << "Note n°" << (i+1) << " : ";
    cin >> saisie;
    fichier << saisie << endl;
  }

  return 0;
}
```

moyenne.cpp

```
#include <fstream>
#include <vector>
using namespace std;

int main()
{
  unsigned nombre;
  ifstream fichier("/home/manu/notes.txt");
  ofstream cout("/dev/pts/2");
  fichier >> nombre;
  cout << nombre << " notes {";
  double somme = 0, note;

  for (unsigned i=0; i<nombre; i++)
    {
      fichier >> note;
      somme += note;
      cout << note << ' ';
    }

  cout << "\b}, moyenne = " << (somme/nombre) << endl;

  return 0;
}
```

Manipuler une chaîne de caractères comme un flux

Grâce aux flux, il est possible de transformer n'importe quel type primitif vers une chaîne de caractères et inversement. C'est ce que nous venons de mettre en œuvre au travers d'un ***fichier texte***. Dans le même ordre d'idée, il est également possible d'appliquer à une zone mémoire (au lieu d'un fichier disque) les opérations généralement destinées à des flots. Il existe pour cela deux classes spécialisées :

Flux chaînes de caractères en mémoire centrale

- ***ostringstream*** *: un objet de cette classe peut recevoir des caractères, au même titre qu'un flot de sortie. La fonction membre* ***str()*** *permet d'obtenir une chaîne (objet de type* ***string****) contenant une copie instantanée de ces caractères.*
- ***istringstream*** *: un objet de cette classe peut être créé à partir d'un tableau de caractères, d'une chaîne constante ou d'un objet de type* ***string****.*

Reprenons l'exemple du factoriel, mais cette fois-ci au travers d'un flux de chaînes de type ***istringstream***.

factoriel.cpp

```
#include <iostream>
#include <sstream>
using namespace std;

unsigned factoriel(unsigned n)
{
  return n==0 ? 1 : n*factoriel(n-1);
}

int main(int nombre, char *arguments[])
{
  if (nombre==2)
  {
    int n;
    istringstream option(arguments[1]);
    option >> n;
    if (n<0) cout << "ATTENTION, nombre négatif" << endl;
    else cout << n << "! = " << factoriel(n) << endl;
  }
  else cout << "Le nombre d'argument n'est pas valide" << endl;

  return 0;
}
```

C'est une solution, mais elle est un peu trop sophistiqué. Le plus simple, à mon avis, est d'utiliser la fonction **stoi()** présente depuis C++11.

factoriel.cpp

```
#include <iostream>
#include <sstream>
using namespace std;

unsigned factoriel(unsigned n)
{
  return n==0 ? 1 : n*factoriel(n-1);
}

int main(int nombre, char *arguments[])
{
  if (nombre==2)
  {
    int n = stoi(arguments[1]);
    if (n<0) cout << "ATTENTION, nombre négatif" << endl;
    else cout << n << "! = " << factoriel(n) << endl;
  }
  else cout << "Le nombre d'argument n'est pas valide" << endl;

  return 0;
}
```