LES FONCTIONS

- **Déclaration des fonctions**
- **Passage par référence**
- **Valeur par défaut des paramètres**
- **Fonction *inline***
- **Surcharge de fonctions**
- **Retour d'une référence**

DECLARATION DES FONCTIONS

Le langage C++ impose au programmeur de déclarer le nombre et le type des arguments de la fonction.

Ces déclarations sont identiques aux prototypes de fonctions de la norme C−ANSI.

Cette déclaration est par ailleurs obligatoire avant utilisation (contrairement à la norme C−ANSI).

La déclaration suivante *int f1();* où *f1* est déclarée avec une liste d'arguments vide est interprétée en C++ comme la déclaration *int f1(void);*

Alors que la norme C−ANSI considère que *f1* est une fonction qui peut recevoir un nombre quelconque d'arguments, eux mêmes de type quelconques, comme si elle était déclarée *int f1(...);* .

PASSAGE PAR REFERENCE

En plus du passage par valeur, le C++ définit le passage par référence. Lorsque l'on passe à une fonction un paramètre par référence, cette fonction reçoit un "synonyme" du paramètre réel.

Toute modification du paramètre référence est répercutée sur le paramètre réel.

Exemple :

```
void echange( int & n1, int & n2) {
  int temp = n1; n1 = n2; n2 = temp;
}

void main() {
  int i=2, j=3;
  cout << "i= " << i << " j= " << j << endl;
  // affichage de :  i= 2  j= 3

  echange(i, j);
  cout << "i= " << i << " j= " << j << endl;
  // affichage de :  i= 3  j= 2
}
```

Comme vous le remarquez, l'appel se fait de manière très simple. Cette facilité augmente la puissance du langage mais doit être utilisée avec précaution, car elle ne protège plus la valeur du paramètre réel transmis par référence.

L'utilisation du mot réservé *const* permet d'annuler ces risques pour les arguments de grande taille ne devant pas être modifiés dans la fonction.

```
struct FICHE {
  char nom[30], prenom[20], email[256];
};

void affiche( const FICHE  & f) {
  cout << f.nom << " " << f.prenom;
  cout <<  " " << f.adresse << endl;
}

void main() {
  FICHE user = {
    "Dancel", "Alain", "dancel@enac.fr"
  };

  affiche(user);
}
```

VALEUR PAR DEFAUT DES PARAMETRES

Certains arguments d'une fonction peuvent prendre souvent la même valeur.

Pour ne pas avoir à spécifier ces valeurs à chaque appel de la fonction, le C++ permet de déclarer des valeurs par défaut dans le prototype de la fonction.

Exemple :

```
void print(long valeur, int base = 10);

void main() {
    print(16);    // affiche 16    (16 en base 10)
    print(16, 2); // affiche 10000 (16 en base 2)
}

void print(long valeur, int base){
    cout << ltostr(valeur, base) << endl;
}
```

- Les paramètres par défaut sont obligatoirement les derniers de la liste.
- Ils ne sont déclarés que dans le prototype de la fonction et pas dans sa définition.

FONCTION *inline*

Le mot clé *inline* remplace avantageusement l'utilisation de *#define* du préprocesseur pour définir des pseudo−fonctions.

Afin de rendre l'exécution plus rapide d'une fonction et à condition que celle ci soit de courte taille, on peut définir une fonction avec le mot réservé *inline*.

Le compilateur génèrera, à chaque appel de la fonction, le code de celle ci. Les fonctions *inline* se comportent comme des fonctions normales et donc, présente l'avantage de vérifier les types de leurs arguments, ce que ne fait par la directive *#define*.

Exemple :

```
#include <iostream.h>

inline int carre(int n); // déclaration

void main() {
  cout << carre(10) << endl;
```

```
}

// inline facultatif à la définition, mais préférable
inline int carre(int n) {
  return n * n;
}
```

contrairement à une fonction normale, la portée d'une fonction *inline* est réduite au module dans lequel elle est déclarée.

SURCHARGE DE FONCTIONS

Une fonction se définit par :

- son nom,
- sa liste typée de paramètres formels,
- le type de la valeur qu'elle retourne.

Mais seuls les deux premiers critères sont discriminants. On dit qu'ils constituent la **signature** de la fonction.

On peut utiliser cette propriété pour donner un même nom à des fonctions qui ont des paramètres différents :

```
int somme( int n1, int n2)
       { return n1 + n2; }

int somme( int n1, int n2, int n3)
       { return n1 + n2 + n3; }

double somme( double n1, double n2)
       { return n1 + n2; }

void main() {
  cout << "1 + 2 = " << somme(1, 2) << endl;
  cout << "1 + 2 + 3 = " << somme(1, 2, 3) << endl;
  cout << "1.2 + 2.3 = " << somme(1.2, 2.3) << endl;
}
```

Le compilateur sélectionnera la fonction à appeler en fonction du type et du nombre des arguments qui figurent dans l'appel de la fonction.

Ce choix se faisant à la compilation, fait que l'appel d'une fonction surchargée procure des performances identiques à un appel de fonction "classique".

On dit que l'appel de la fonction est résolu de manière statique.

Autres exemples :

```
int somme1(int n1, int n2) {return n1 + n2;}
int somme1(const int n1, const int n2) {return n1 + n2;}
// Erreur: la liste de paramètres dans les déclarations
//         des deux fonctions n'est pas assez divergente
//         pour les différencier.

int somme2(int n1, int n2) {return n1 + n2;}
int somme2(int & n1, int & n2) {return n1 + n2;}
// Erreur: la liste de paramètres dans les déclarations
//         des deux fonctions n'est pas assez divergente
//         pour les différencier.

int    somme3(int n1, int n2) {return n1 + n2;}
double somme3(int n1, int n2) {return (double) n1 + n2;}
// Erreur :
// Seul le type des paramètres permet de faire la distinction
// entre les fonctions et non pas la valeur retournée.

int somme4(int n1, int n2) {return n1 + n2;}
int somme4(int n1, int n2=8) {return n1 + n2;}
// Erreur: la liste de paramètres dans les déclarations
//         des deux fonctions n'est pas assez divergente
//         pour les différencier.
```

RETOUR D'UNE REFERENCE

Fonction retournant une référence

Une fonction peut retourner une valeur par référence et on peut donc agir, à l'extérieur de cette fonction, sur cette valeur de retour.

La syntaxe qui en découle est plutôt inhabituelle et déroutante au début.

```
#include <iostream.h>

int t[20];  // variable globale -> beurk !

int & nIeme(int i) {
  return t[i];
}

void main() {
  nIeme(0) = 123;
```

```
  cout << t[0] << " " << ++nIeme(0);
}
// -- résultat de l'exécution ----------
// 123 124
```

- Tout se passe comme si *nIeme(0)* était remplacé par *t[0]*.
- Une fonction retournant une référence (non constante) peut être une *lvalue*.
- La variable dont la référence est retournée doit avoir une durée de vie permanente.

Retour d'une référence constante

Afin d'éviter la création d'une copie, dans la pile, de la valeur retournée lorsque cette valeur est de taille importante, il est possible de retourner une valeur par référence constante.

Le préfixe *const*, devant le type de la valeur retournée, signifie au compilateur que la valeur retournée est constante.

```
const Big & f1() {
  static Big b;    // notez le static ...
  // ...
  return b;
}

void main() {
  f1() = 12;  // erreur
}
```
