LANGAGE C++

LES FONCTIONS

Notion de fonction

Règle : Ne jamais dupliquer de code en programmant

Pourquoi ne jamais dupliquer du code (notion de réutilisabilité)

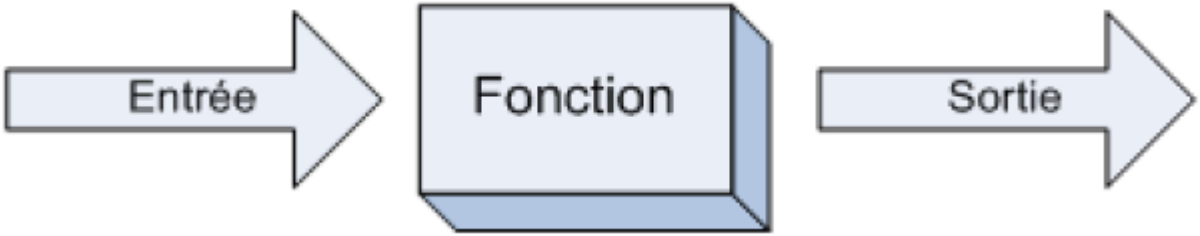
Cela rend le programme :

- inutilement long
- difficile à comprendre
- difficile à maintenir : reporter chaque modification dans chacune des copies

Tout bon langage de programmation fournit donc des moyens pour permettre la réutilisation de portions de programmes.

Fonction (en programmation)
fonction = portion de programme réutilisable ou importante en soi

Une fonction reçoit des informations en entrée, exécute des actions et renvoie un résultat.

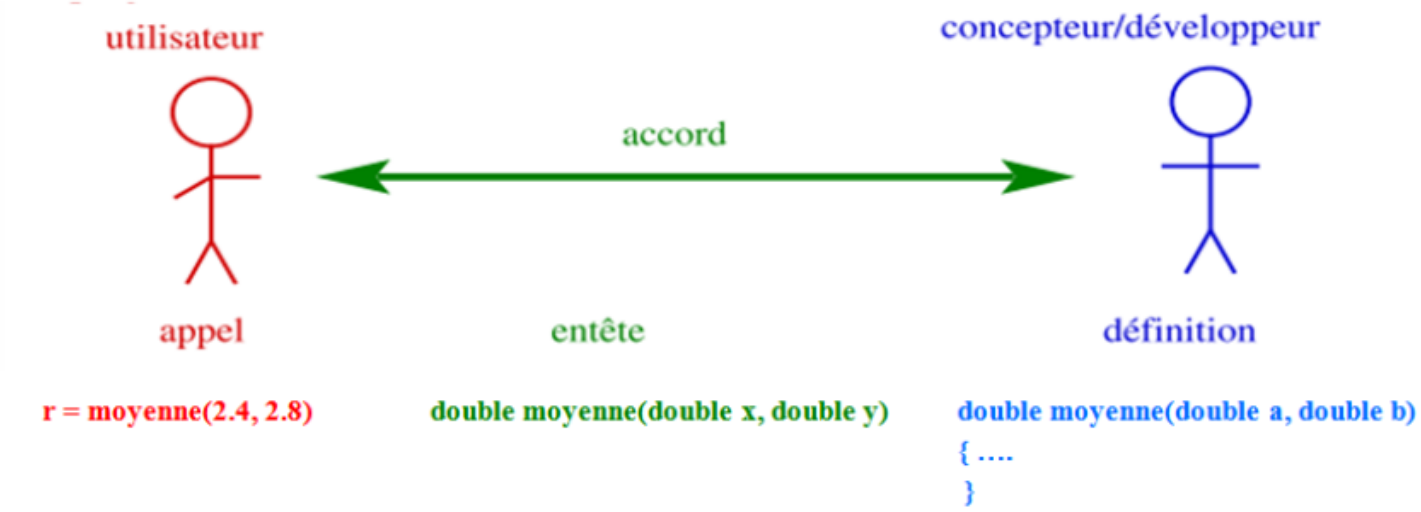


Caractéristiques d'une fonction

Les « 3 facettes » d'une fonction

1. Résumé / Contrat (« **prototype** »)
2. Implémentation / Création (« **définition** »)
3. Utilisation (« **appel** »)

Imaginons que nous développions un programme qui doit souvent calculer la moyenne de deux réels :



Caractéristiques d'une fonction

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

double moyenne(double nombre_1, double nombre_2);

int main()
{
  double note1(0.0), note2(0.0);
  cout << "Entrez vos deux notes : " << endl;
  cin >> note1 >> note2;
  cout << "Votre moyenne est : "
       << moyenne(note1, note2) << endl;
  return 0;
}

double moyenne(double x, double y)
{
   return (x + y) / 2.0;
}
```

prototype

appel

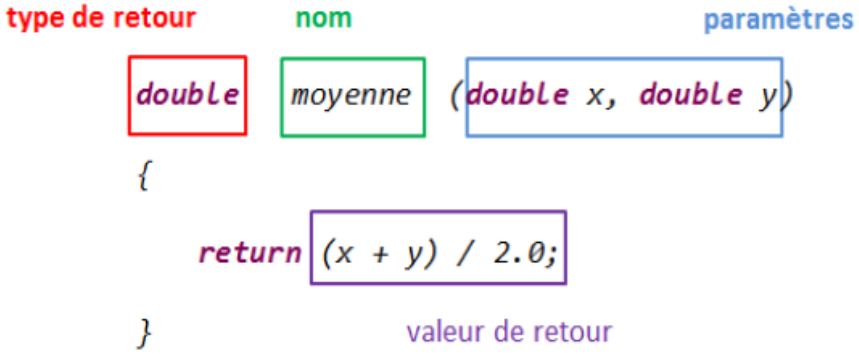
définition

Une fonction est un objet logiciel caractérisé par :

un corps : la portion de programme à réutiliser ou mettre en évidence, qui a justifié la création de la fonction ;

- **un nom** : par lequel on désignera cette fonction ;
- **des paramètres** : (les « entrées », on les appelle aussi « arguments ») ensemble de variables extérieures à la fonction dont le corps dépend pour opérer ;
- **un type et une valeur de retour** : (la « sortie ») ce que la fonction renvoie au reste du programme

L'utilisation de la fonction dans une autre partie du programme se nomme un appel de la fonction.



Note :

- Les arguments figurant dans l'en-tête de la définition d'une méthode se nomment ***arguments muets*** *(*rôle voisin de celui d'une variable locale à la méthode, avec cette seule différence que leur valeur sera fournie à la méthode au moment de son appel).
- Les arguments fournis lors de l'appel de la fonction portent quant à eux le nom d'***arguments effectifs***.

Evaluation de l'appel d'une fonction - contexte

Que se passe-t-il lors de l'appel suivant :

```
z = moyenne( 1.5 + 0.8, 3.4 * 1.25 );
```

```
double moyenne (double x, double y) {
    return (x + y) / 2.0;
}
```

1. évaluation des expressions passées en arguments :

1.5 + 0.8 → 2.3

3.4 * 1.25 → 4.25

2. affectation des paramètres :

x = 2.3

y = 4.25

3. exécution du corps de la fonction :

rien dans ce cas (corps réduit au simple return)

4. évaluation de la valeur de retour (expression derrière return)

(x + y) / 2.0 → 3.275

5. replacement de l'expression de l'appel par la valeur retournée :

z = 3.275;

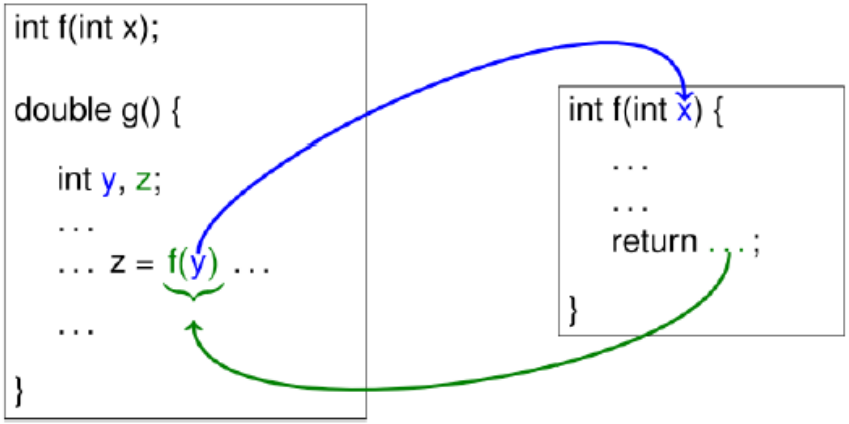
Évaluation d'un appel de fonction

L'évaluation de l'appel d'une fonction s'effectue de la façon suivante

1. les expressions passées en argument sont **évaluées**
2. les valeurs correspondantes sont **affectées** aux paramètres de la fonction
3. le corps de la méthode est exécuté
4. l'expression suivant la première commande return rencontrée est évaluée...
5. ...et retournée comme résultat de de l'appel : cette valeur remplace l'expression de l'appel

Remarques :

- Les étapes 1 et 2 n'ont pas lieu pour une fonction sans arguments.
- Les étapes 4 et 5 n'ont pas lieu pour une fonction sans valeur de retour (void).
- L'étape 2 n'a pas lieu lors d'un passage par référence (voir plus loin).



Le passage des arguments – contexte

Contexte

Que vaut val ? Qu'affiche le programme ?

```
#include <iostream>
using namespace std;
void f(int x);
int main()
{
    int val(1);
    f(val);
    cout << " val=" << val << endl;
    return 0;
}
void f(int x) {
    x = x + 1;
    cout << "x=" << x;
}
```

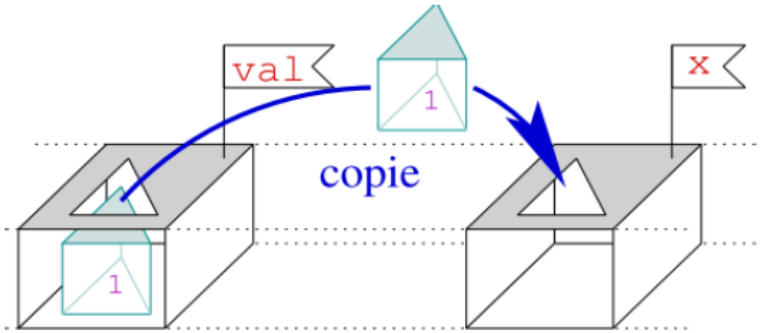
A : x=1 val=2
B : x=2 val=0
C : x=2 val=1
D : x=2 val=2

Que peut-on garantir sur la valeur d'une variable passée en argument lors d'un appel de fonction ?

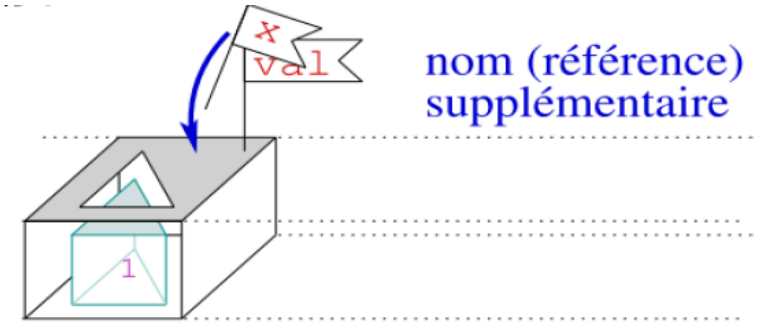
Passage par valeur / référence

On distingue 2 types de passages d'arguments :

1. **Passage par valeur :** la variable locale associée à un argument passé par valeur correspond à une copie de l'argument (i.e. un objet distinct mais de même valeur littérale).
=> Une modification effectuée à l'intérieur de la fonction ne sont donc pas répercutées à l'extérieur



2. **passage par référence :** la variable locale associée à un argument passé par référence correspond à une référence sur l'objet associé à l'argument lors de l'appel.
=> Une modification effectuée à l'intérieur de la fonction se répercute alors à l'extérieur de la fonction.



Passage par valeur / référence

Exemple de passage par valeur

```
void f(int x);
int main()
{
     int val(1);
     f(val);
     cout << " val=" << val << endl;
     return 0;
}
void f(int x) {
     x = x + 1;
     cout << "x=" << x;
}
```

```
x=2 val=1

Process returned 0 (0x0)
Press any key to continue.
```

Exemple de passage par référence

```
void f(int &x);
int main()
{
     int val(1);
     f(val);
     cout << " val=" << val << endl;
     return 0;
}
void f(int &x) {
     x = x + 1;
     cout << "x=" << x;
}
```

```
x=2 val=2

Process returned 0 (0x0)
Press any key to continue.
```

Quand utiliser un passage par référence ?

lorsque l'on souhaite modifier une variable

Par exemple :

pour saisir une valeur

```
void saisie_entier(int& a_lire);
...
int i(0);
...
saisie_entier(i);
```

Alternative : retourner la valeur

```
int saisie_entier();
...
i = saisie_entier();
```

pour « retourner » plusieurs valeurs

```
void echangerDeuxVariable(int& a, int& b);
```

Alternative : utiliser les structures (futur cours)

Quand utiliser un passage par référence ?

Passage par valeur

```
#include <iostream>
using namespace std;
void ajouteDeux(int a);
int main()
{
    int nombre(4);
    cout << "AVANT : " << nombre << endl;
    ajouteDeux(nombre);
    cout << "APRES : " << nombre << endl;
    return 0;
}
void ajouteDeux(int a) //Notez le petit & !!!
{
    a+=2;
}
```

```
AVANT : 4
APRES : 4

Process returned 0 (0x0)
Press any key to continue.
```

Passage par référence

```
#include <iostream>
using namespace std;
void ajouteDeux(int& a);
int main()
{
    int nombre(4);
    cout << "AVANT : " << nombre << endl;
    ajouteDeux(nombre);
    cout << "APRES : " << nombre << endl;
    return 0;
}
void ajouteDeux(int& a) //Notez le petit & !!!
{
    a+=2;
}
```

```
AVANT : 4
APRES : 6

Process returned 0 (0x0)
Press any key to continue.
```

Prototypage

Toute fonction doit être annoncée avant d'être utilisée : prototype

prototype = déclaration de la fonction, sans en définir le corps :

- nom
- paramètres
- type de (la valeur de) retour

Syntaxe

$$type\ nom\ (\ \overbrace{type_1\ id\_param_1,\ \ldots,\ type_N\ id\_param_N}^{liste\ de\ paramètres}\ )\ ;$$

Exemples de prototypes

```
double moyenne(double x, double y);
int nbHasard();
void afficheMessageBienvenu() ;
void echangerDeuxVariable(int& a, int& b);
void cartesiennes_vers_polaires(double x, double y, double& distance, double& rayon);
```

Bonnes pratiques

- Une fonction ne doit faire que ce pour quoi elle est prévue
- Choisissez des noms pertinents pour vos fonctions et vos paramètres
- Commencez toujours par faire le prototype de votre fonction : Demandez-vous ce qu'elle doit recevoir et retourner.

Prototypage

Attention à la syntaxe !

- **int a;** : déclaration de variable non initialisée
- **int a();** : prototype de fonction sans paramètre
- **int a(5);** : déclaration/initialisation de variable
- **a(5);** : appel de fonction à un argument

Définition des fonctions

La définition d'une fonction sert à définir ce que fait la fonction
=> spécification du corps de la fonction

Syntaxe

```
type nom ( liste de paramètres )
{
    instructions du corps de la fonction;
    return expression;
}
```

Exemple

```
double moyenne (double x, double y) {
    return (x + y) / 2.0;
}
```

Corps de la fonction

- Le corps de la fonction est donc un **bloc** dans lequel on peut utiliser les paramètres de la fonction (en plus des variables qui lui sont propres).
- La valeur retournée par la fonction est indiquée par l'instruction :
  return expression; // où l'expression a le même type que celui retourné par la fonction.

L'instruction return fait deux choses:

- elle précise la valeur qui sera fournie par la fonction en résultat
- elle met fin à l'exécution des instructions de la fonction.

Remarques sur l'instruction return

1. Il est possible de placer plusieurs instructions return dans une même fonction.
Par exemple, une fonction déterminant le maximum de deux valeurs peut s'écrire

avec une instruction return :

```
double max1(double a, double b) {
    double m;
    if (a > b) {
        m = a;
    } else {
        m = b;
    }
    return m;
}
```

ou deux :

```
double max2double a, double b)
{
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

2. Le type de la valeur retournée doit correspondre au type dans l'en-tête :

```
double f() {
    bool b(true);
    //...
    return b; // Erreur : mauvais type
}
```

Remarques sur l'instruction return

3. return doit être la toute dernière instruction exécutée :

```
double lire() {
  cout << "Entrez un nombre : ";
  double n(0.0);
  cin >> n;
  return n;
  cout << "entré : " << n << endl; // Erreur : jamais exécuté
}
```

4. Le compilateur doit être sûr de toujours pouvoir exécuter un return

```
double lire() {
    cout << "Entrez un nombre : ";
    double n(0.0);
    cin >> n;
    if (n > 0.0) {
        return n;
    }
    // Erreur : pas de return si n <= 0 !
}
```

Fonctions sans valeur de retour

Quand une fonction ne doit fournir aucun résultat (on appelle de telles fonctions des « procédures »)

- On utilise alors le type particulier **void** comme type de retour.

Dans ce cas la commande de retour return est optionnelle :

- soit on ne place aucun return dans le corps de la fonction
- soit on utilise l'instruction return sans la faire suivre d'une expression: **return;** => sortir de la fonction

```
void affiche_racine(double a)
{
    if (a < 0.0) {
        return; /* Grâce à ce return, on quitte la fonction avant *
        * de calculer sqrt(a) si a est négatif. */
    }
    cout << sqrt(a);
    // il n'est pas nécessaire de mettre un return ici
}
```

Fonctions sans paramètre

```
double saisie()
{
    double nb_points(0.0);
    do {
        cout << "Entrez le nombre de points (0-100) : ";
        cin >> nb_points;
    } while ((nb_points < 0.0) or (nb_points > 100.0));
    return nb_points;
}
```

La fonction main()

main est aussi une fonction avec un nom et un prototype imposés.

- Par convention, tout programme C++ doit avoir une fonction main, qui est appelée automatiquement quand on exécute le programme.
- Cette fonction doit retourner une valeur de type int. La valeur 0 indique par convention que le programme s'est bien déroulé.

- Les deux seuls prototypes autorisés pour main sont :

```
int main(); // Seul le premier sera utilisé dans ce cours.
int main(int argc, char** argv);
```

Méthodologie pour construire une fonction

1 clairement identifier ce que doit faire la fonction

(ce point n'est en fait que conceptuel, on n'écrit aucun code ici !)

⇒ ne pas se préoccuper ici du comment, mais bel et bien du **quoi** !

2 quels arguments ?

⇒ que doit recevoir la fonctionpour faire ce qu'elle doit ?

3 passage(s) par valeur(s) / référence(s) ?

⇒ pour chaque argument : doit-il être modifié par la fonction ? (si oui : passage par référence)

4 quel type de retour ?

⇒ que doit « retourner » la fonction?

5 (maintenant, et seulement maintenant) Se préoccuper du comment :

=> écrire le code du corps de la fonction

Exercice 1

```cpp
#include <iostream>
using namespace std;
int ajouteDeux(int a);
int main() {
    int nombre(4);
    cout << "AVANT : " << nombre << endl;
    ajouteDeux(nombre);
    cout << "APRES : " << nombre << endl;
    return 0;
}
int ajouteDeux(int a) {
    a+=2;
    return a;
}
```

Exercice 2

```cpp
#include <iostream>
using namespace std;
int ajouteDeux(int& a);
int main() {
    int nombre(4);
    cout << "AVANT : " << nombre << endl;
    ajouteDeux(nombre);
    cout << "APRES : " << nombre << endl;
    return 0;
}
int ajouteDeux(int& a) {
    a+=2;
    return a;
}
```

Exercice 1 :

A partir des étapes suivantes écrire la fonction lireEntier

1. Ecrire une fonction qui lit à partir du clavier un entier tant que la valeur saisit n'est pas comprise entre deux bornes passées en paramètres.
2. Les deux bornes
3. Passage par référence, nous ne souhaitons pas modifier les bornes mais lire un entier.
4. Le résultat de la fonction est un entier

Exercice 2 : dessiner un rectangle d'étoiles *

Ecrire une fonction dessinerRectangle qui deux paramètres, la largeur et la hauteur du rectangle

```
int main()
{
    int largeur, hauteur;
    cout << "Largeur du rectangle : ";
    cin >> largeur;
    cout << "Hauteur du rectangle : ";
    cin >> hauteur;
    dessineRectangle(largeur, hauteur);
    return 0;
}
```

Arguments par défaut

- Lors de son prototypage, une fonction peut donner des valeurs par défaut à ses paramètres.
- Il n'est alors pas nécessaire de fournir d'argument à ces paramètres lors de l'appel de la fonction.

Syntaxe d'un paramètre avec valeur par défaut

type identificateur = valeur

Attention : Les paramètres avec valeur par défaut doivent apparaître en dernier dans la liste des paramètres d'une fonction.

```
#include <iostream>
#include <cmath>
using namespace std;
void affiche_ligne(char elt, int nb = 5);
int main() {
    affiche_ligne('*');
    affiche_ligne('+', 8);
    return 0;
}
void affiche_ligne(char elt, int nb) {
    for(int i(0); i < nb; ++i) {
        cout << elt;
    }
    cout << endl;
}
```

Les arguments par défaut se spécifient dans le prototype et non pas dans la définition de la fonction

Arguments par défaut

Remarque

- Lors de l'appel à une fonction avec plusieurs paramètres ayant des valeurs par défaut, les arguments omis doivent être les derniers et omis dans l'ordre de la liste des paramètres.

Exemples

```
void f(int i, char c = 'a', double x = 0.0);
```

```
f(1)          → correct (vaut f(1, 'a', 0.0))
f(1, 'b')     → correct (vaut f(1, 'b', 0.0))
f(1, 3.0)     → incorrect !
f(1, , 3.0)   → incorrect !
f(1, 'b', 3.0) → correct
```

```cpp
#include <iostream>
#include <vector>
using namespace std;
int nombreDeSecondes(int heures, int minutes = 0, int secondes = 0);
int main()
{
    cout << nombreDeSecondes(1) << endl;
    return 0;
}
int nombreDeSecondes(int heures, int minutes, int secondes)
{
    int total = 0;
    total = heures * 60 * 60;
    total += minutes * 60;
    total += secondes;
    return total;
}
```

```
3600

Process returned 0 (0x0)
Press any key to continue.
```

La surcharge de fonctions

Le mécanisme surcharge de fonctions permet de définir plusieurs fonctions de même nom si ces fonctions n'ont pas les mêmes listes de paramètres (le type de retour n'est pas pris en compte)

=> nombre ou types de paramètres différents.

Exemple

```
void affiche(int x) {
    cout << "entier : " << x << endl;
}
void affiche(double x) {
    cout << "reel : " << x << endl;
}
void affiche(int x1, int x2) {
    cout << "couple : " << x1 << x2 << endl;
}
```

Exercice : exécuter les appels affiche(1), affiche(1.0) et affiche(1,1)

Remarque :

```
void affiche(int x);
void affiche(int x1, int x2 = 1);    // ambiguité !
```