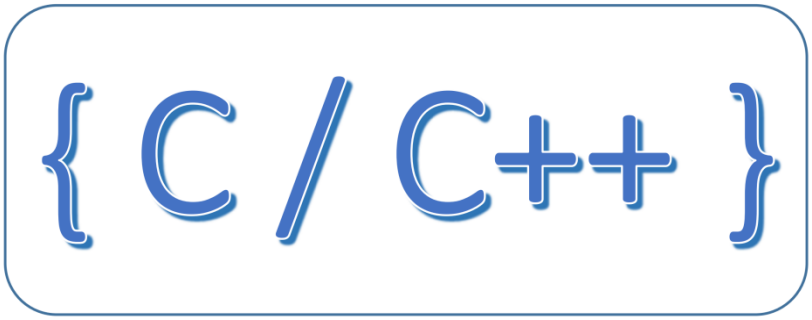
LYCÉE ST-JOSEPH HASPARREN BTS SN-IR	C++ FONCTIONS	COURS

Tâches professionnelles :	T7.2	Réaliser une maquette, un prototype logiciel/matériel
Compétences du référentiel :	C4.4	Développer un module logiciel.
Savoirs - Savoir faires :	S4.7	Langage de programmation C++

Table des matières



1 LES FONCTIONS

Les fonctions permettent de structurer les programmes en segments de code qui effectuent des tâches bien précises, et qu'on appelle quand on en a besoin.

Par exemple dans « …\06 - C++ Opérateurs\Emballages.cpp » on demande plusieurs saisies consécutives, et à chaque fois on a plusieurs lignes de programmes quasi identiques pour faire la vérification de la saisie :

- Cela alourdi le programme qui devient plus difficile à lire et comprendre
- En cas de modification, il va falloir faire cette modification en plusieurs endroits

Il serait préférable d'utiliser une fonction qui s'occupe de cette vérification, et qu'on appelle quand on en a besoin.

En C ++, une fonction est un groupe d'instructions auquel on a donné un nom, et qui peut être appelé depuis n'importe quel endroit du programme. La syntaxe la plus courante pour définir une fonction est:

type nom (paramètre1, paramètre2, ...) { instructions }

- Le **type** est le type de la **valeur retournée** par la fonction.
- Le **nom** est l'identifiant par lequel la fonction peut être appelée.
- Des **paramètres** (autant que nécessaire) : Chaque paramètre est constitué d'un **type suivi par un identificateur**, chaque paramètre étant séparé du suivant par une **virgule**. Chaque paramètre ressemble beaucoup à une déclaration normale de variable (par exemple : double x), et agit en fait au sein de la fonction comme une variable normale qui est locale (interne) à la fonction. Le but de **paramètres** est de permettre le **passage d'arguments à la fonction à partir de l'endroit où elle est appelée**.
- **instructions** est le corps de la fonction. C'est un bloc d'instructions entourées par des accolades **{}** , et donc c'est ce que fait la fonction.

Exemple :

```
#include <iostream>
#include <math.h>                    // pour les maths (sqrt,...)
using namespace std;

// définition de la fonction pythagore
double pythagore(double x, double y)
{
    double a;
    a = x * x + y * y;
    a = sqrt(a);
    return a ;
}

int main()
{
    double u, v, w, adj, opp, hypo;

    cout << "Tapez la valeur du coté adjacent: "; cin >> u;
    cout << "Tapez la valeur du coté opposé: "; cin >> v;
    w = pythagore(u, v); //appel de notre fonction
    cout << "L'hypothénuse vaut: " << w << endl;

    cout << "Tapez la valeur du coté adjacent: "; cin >> adj;
    cout << "Tapez la valeur du coté opposé: "; cin >> opp;
    hypo = pythagore(adj, opp); //appel de notre fonction
    cout << "L'hypothénuse vaut: " << hypo << endl;

    system("pause");
    return 0;
}
```

Ce programme est divisé en deux fonctions : **pythagore** et **main**. Rappelez - vous que peu importe l'ordre dans lequel elles sont définies, le programme C ++ démarre toujours en appelant **main**. En fait, **main** est la seule fonction appelée automatiquement et le code dans une autre fonction est exécutée seulement si sa fonction est appelée à partir main (directement ou indirectement).

Dans l'exemple ci - dessus, **main** commence en déclarant les variables u, v, w de type double, puis il demande à l'utilisateur de saisir 2 valeurs, et juste après il effectue l'appel de la fonction : il appelle la fonction **pythagore**. L'appel à une fonction a une structure très similaire à sa déclaration. Dans l'exemple ci - dessus, l'appel à **pythagore** peut être comparée à sa définition :

```
// définition de la fonction pythagore
double pythagore(double x, double y)
```

```
w = pythagore(u, v); //appel de notre fonction
```

L'appel **passe en arguments** deux valeurs de variables **u** et **v** à la fonction : ceux - ci correspondent aux **paramètres x** et **y**, déclarés pour la fonction pythagore (**en tout cas le type correspond, le nom n'est pas important**).

A l'endroit où la fonction est appelée à partir de **main**, le contrôle est passé à la fonction **pythagore** : ici, l'exécution de **main** est arrêtée, et ne reprendra qu'une fois la fonction **pythagore** terminée. Au moment de l'appel de la fonction, la valeur des deux arguments (**u** et **v**) sont copiés dans les variables **x** et **y** locales à la fonction.

Ensuite, à l'intérieur de **pythagore**, une autre variable locale est déclarée (**double a**) et au moyen des instructions suivantes on calcule la valeur de l'hypoténuse du triangle : le résultat se retrouve dans la variable locale **a**.

L'instruction finale de la fonction (`return a;`) permet de retourner le résultat à la fonction appelante **main**.

Comme nous retournons un double (**a**) et que le type de la fonction **pythagore** est aussi un double, on peut récupérer la valeur retournée (renvoyée) par la fonction pythagore : ici on la récupère dans **w**, qui est aussi un double.

```
// définition de la fonction pythagore
double pythagore(double x, double y)
```

```
w = pythagore(u, v); //appel de notre fonction
```

Vu de w l'appel à la fonction est remplacé par la valeur qu'elle retourne.

En résumé :

```
int main()
{
    double u, v, w;
    cout << "Tapez la valeur de u : "; cin >> u;
    cout << "Tapez la valeur de v : "; cin >> v;

    w = pythagore(u, v); //appel de notre fonction

    cout << "Le résultat est " << w << endl;
    return 0;
}
```

```
double pythagore(double x, double y)
{
    double a;
    a = x*x + y*y;
    a = sqrt (a);
    return a;
}
```

Utiliser des fonctions présente beaucoup d'avantages :

- Le programme principal est beaucoup plus lisible
- En cas de modification du contenu de la fonction, vous n'aurez qu'un endroit à retoucher
- Vous pouvez écrire une fonction qui sera utilisée par d'autres, vous aurez juste à leur dire :
  - Ce que fait la fonction
  - Le type et le nombre de paramètres (et leur utilisation dans la fonction)
  - Le type de la valeur retournée (et sa signification)
- Vous pouvez surtout utiliser des fonctions écrites et déjà testées par des professionnels, et c'est toujours un moyen qu'il faut privilégier (ne pas réinventer la roue) : mêmes choses à savoir que point précédent, donc lire la documentation fournie avec la fonction.

Exercice : Saisissez ce programme, testez-le, puis **rajouter** dans la fonction, et dans le main, une ligne qui affiche l'adresse physique en mémoire des variables **u, v, w, x, y, a** (voir cours C++ Variables et Types chap).

D'après les adresses est-ce que **u** et **x** sont la même variable ? ..... **v** et **y** ?..... **a** et **w** ?.....

2 FONCTION SANS RETURN: UTILISATION DE VOID

void nom (paramètre1, paramètre2, ...) { instructions }

void signifie que la fonction ne retourne pas de valeur. **void** est un type spécial pour représenter l'absence de valeur de retour.

Par exemple, une fonction qui affiche simplement un message n'a pas besoin de retourner une valeur quelconque :

```
// void function example
#include <iostream>
#include <string>
using namespace std;

void printmessage(string texte) // retour void
{
    cout << texte;
                    // pas de return
}

int main()
{
    // pas de valeur de retour à récupérer
    printmessage("du texte à afficher");
}
```

3 FONCTION SANS PARAMETRES

void peut également être utilisé dans la liste des paramètres de la fonction pour spécifier explicitement que la fonction ne prend aucun paramètre lorsqu'elle est appelée.

En C ++, une liste de paramètres vide peut être utilisé à la place du void avec la même signification, mais l'utilisation de void dans la liste d'arguments est fréquente (habitude du langage C pour lequel c'est obligatoire).

```
// void function example
#include <iostream>
#include <string>
using namespace std;

bool printAlarme()  // pas de paramètres
{
    cout << "Attention alarme !";
    return true;
}

int main()
{
        // pas de paramètres
    if ( printAlarme() )    cout << "OK";
}
```

Ce qui n'est en aucun cas facultatif ce sont les **parenthèses** qui suivent le nom de la fonction, aussi bien dans sa déclaration, que lors de l'appel. Et même si la fonction ne prend aucun paramètre, au moins une paire de parenthèses vide doit toujours être ajoutée au nom de la fonction.

Les parenthèses sont ce qui différencie les fonctions des autres types de déclarations ou d'instructions.

```
printAlarme();  //OK

printAlarme;    // pas OK
```

4 LA VALEUR DE RETOUR DE MAIN

Vous avez peut - être remarqué que le type de retour main est **int**, donc normalement à la fin du main il doit y avoir un instruction **return**.

Pour le **main**, si vous ne mettez pas d'instruction **return**, le compilateur rajoute automatiquement `return 0;`

Cela signifie en général que le programme s'est terminé correctement.

5 ARGUMENTS PASSES PAR VALEUR OU PAR REFERENCE

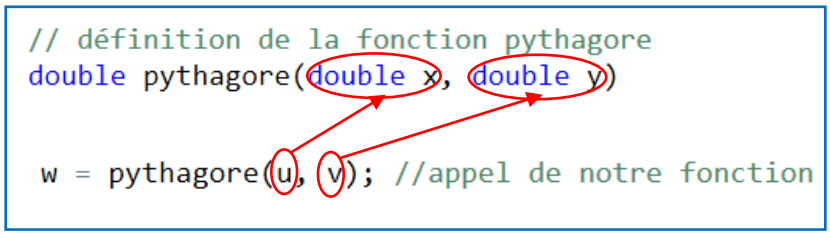
Dans les fonctions vues précédemment, les arguments ont toujours été passé ***par valeur***.

Cela signifie que, lors de l'appel d'une fonction, ce qui est passé à la fonction sont les valeurs de ces arguments qui sont copiées dans les variables locales représentées par les paramètres de la fonction.

Dans l'exemple précédent nous avions :

La valeur de u (variable de main) est copiée dans x (variable interne de pythagore).

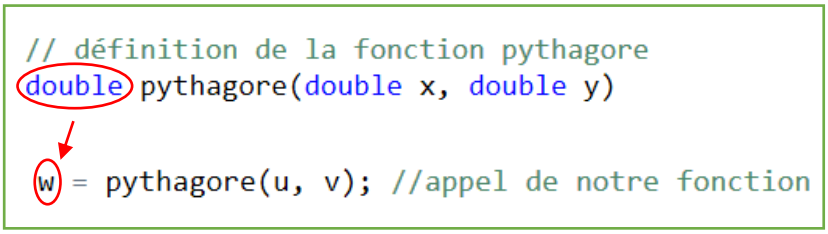
(**de même pour v copiée dans y**)



Cela fonctionne toujours, mais, si on doit passer des variables de grande taille (par exemple chaînes de caractères (string) longues, etc.) alors apparaissent 2 inconvénients :

- la copie peut prendre du temps,
- on utilise 2 fois de la mémoire (une fois dans main, une autre fois en local dans la fonction appelée)

De même le return retournait une valeur qui est le résultat de la fonction :



L'inconvénient principal est que :

- la fonction ne peut donner qu'un seul résultat (return d'une seule valeur).

Il existe une possibilité pour contourner ces inconvénients : **les arguments passés à la fonction peuvent lui être passés *par référence*, plutôt que *par valeur*.**

C'est-à-dire que ce qu'on va passer à la fonction ce n'est pas la valeur de la variable, mais sa référence c'est-à-dire un lien vers son adresse physique dans la mémoire.

Par conséquent, de l'intérieur de la fonction on peut aller **lire et modifier** ces variables, qui ne sont pourtant pas locales à la fonction, car la fonction connait leur adresse physique !

Ce qui fait que l'on n'a plus les inconvénients cités plus haut :

- Même pour des variables de grande taille l'adresse a la même taille (pas de perte de temps)
- On ne duplique pas la variable de grande taille, mais son adresse (pas de gaspillage de mémoire)
- On peut modifier les variables, donc on n'a plus besoin du return, sauf pour dire si la fonction c'est bien finie

Exemple :

On écrit une fonction qui transforme une valeur de température donnée en degrés Fahrenheit °F en valeur en degrés Celsius °C.

Cette fonction vérifie aussi que la valeur en °F n'est pas inférieure au minimum dans cette échelle (-459.67°F). Si c'est le cas la fonction retourne -1 (erreur), sinon elle retourne 0 (ok).

La valeur de retour étant utilisée pour ce test de validité, on va utiliser faire le passage d'argument par référence pour que la variable qui passe la valeur en °F contienne la valeur en °C après la fin de la fonction.

```
///Fahrenheit --> Celsius: conversion d'une température
//////////////////////////////////////////////////////

#include <iostream>
#include <clocale>              // pour gestion des caractères accentués
#include <limits>               // pour cin.ignore

using namespace std;

// définition de la fonction fahrenheitCelsius
int fahrenheitCelsius(double& temper) // avec en paramètre une référence vers un double
{
    if (temper < -459.67) return -1 ;       // return erreur inférieur à valeur min
    else {
        temper = (temper - 32.0) / 1.8;     // conversion et résultat dans la même variable
        return 0 ;                      // return ok
    }

}
```

```
int main()
{
    setlocale(LC_CTYPE, "fra");     // pour gestion des caractères accentués

    double temp;

    cout << "Tapez la valeur d'une température en °F: "; cin >> temp;

    if (fahrenheitCelsius(temp) == -1)  //appel fonction, identique pour passage par valeur et référence
        cout << "Erreur saisie température <  -459.67 \n" ;
    else
        cout << "La température en °C est:  " << temp << endl;

    cin.ignore(numeric_limits<streamsize>::max(), '\n');      // efface ce qui pourrait trainer dans le b
    cout << "Taper Entrée pour continuer";                    // équivalent à system("pause")
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    return 0;
}
```

Dans la définition de la fonction, le paramètre est de type **référence vers un double** : **double &** (le & peut-être collé ou pas).

La variable **temper** locale à la fonction recevra le lien vers le double (appelé ***temp*** dans le main) qui sera passé en argument lors de l'appel de la fonction depuis le main.

La variable ***temper*** locale à la fonction, devient la **même variable** que celle qui s'appelle ***temp*** dans le main. Ces deux noms de variables désignent la **même case mémoire**, on dit aussi que les deux noms sont des **alias** (nous reverrons cela plus en détail bientôt). On aurait pu aussi donner le même nom aux deux.

La fonction teste aussi l'argument qui lui est passé, et si ce n'est pas inférieur au min, transforme les °F en °C, dans la même variable.

Dans le main, on appelle la fonction en lui passant en argument le double **temp** (qui contient ce que l'on a saisi en °F), et dans la même instruction teste la valeur retournée.

L'affichage dépend de la valeur retournée.

Exercice : Saisissez ce programme, testez-le, puis **rajouter** dans la fonction, et dans le main, une ligne qui affiche l'adresse des variables **temp** et **temper** (voir cours C++ Variables et Types chap 4 pour exemple).

6 PASSAGE DE PARAMETRES PAR REFERENCES CONST

Supposons que vous vouliez passer un argument par référence car la variable est de grande taille, mais que vous ne vouliez pas qu'elle soit modifiée dans la fonction.

Vous pouvez déclarer la fonction comme ceci :

```
// définition de la fonction fahrenheitCelsius
void maFonction(const string & phrase) // avec en paramètre une référence constante vers un double
{
      …        //ici on peut lire le contenu du string phrase, mais pas le modifier car const

}
```

```
int main()
{
    setlocale(LC_CTYPE, "fra");     // pour gestion des caractères accentués

    string phrase = "Nom Prénom etc . bla bla " ;

    maFonction (phrase) ;
```

On passe l'argument **phrase** par **référence constante,** plutôt que par valeur pour gagner du temps et de la place (en supposant que phrase soit longue, on ne passe que son adresse et pas toute la chaîne), donc l'alias de phrase (qu'on a appelé phrase aussi dans la fonction) pourra utiliser le contenu de phrase, mais pas le modifier car référence constante.

7 VALEURS PAR DEFAUT DES PARAMETRES

En C ++, les fonctions peuvent aussi avoir des **paramètres optionnels**, pour lesquels aucun argument n'est requis dans l'appel.

Par exemple, une fonction de trois paramètres peut être appelée avec seulement deux arguments. Pour cela, la fonction doit inclure une valeur par défaut pour son dernier paramètre, qui est utilisée par la fonction lorsqu'elle est appelée avec moins d'arguments.

Par exemple :

```
// default values in functions
#include <iostream>
using namespace std;

int divide (int a, int b=2)
{
  int r;
  r=a/b;
  return  r ;
}

int main ()
{
  cout << divide (12) << '\n';
  cout << divide (20,4) << '\n';
  return 0;
}
```

```
6
5
```

Dans cet exemple, il y a deux appels pour la fonction **divide**. Dans le premier : `divide (12)` l'appel ne passe qu'un argument à la fonction, même si la fonction a deux paramètres.

Dans ce cas, la fonction suppose que le second paramètre vaut 2 (notez la définition de fonction, qui déclare son deuxième paramètre comme **int b=2**). Par conséquent, le résultat est 6.

Dans le deuxième appel : `divide (20,4)` l'appel passe deux arguments à la fonction. Par conséquent, la valeur par défaut pour b (int b=2) est ignorée, et b prend la valeur passée en argument, qui est 4, ce qui donne un résultat de 5.

8 DECLARATION DE FONCTIONS

Exercice : reprenez le programme de conversion °F °C vu un peu plus haut, vous voyez que l'on a d'abord la définition de la fonction **fahrenheitCelsius, puis la définition du main.**

Faites un couper/coller pour inverser les 2, puis recompilez. Que constate-t-on ? .....

.....

En fait, au moment de la compilation pour pouvoir appeler la fonction, il faut que le compilateur la connaisse, au moins de nom, qu'il sache qu'elle existe, que ce n'est pas une erreur.

C'est à l'édition de lien (link, opération que nous allons voir de plus près bientôt) que se fait la vérification finale et l'assemblage de tous les morceaux de notre programme.

On va donc distinguer 2 niveaux, 2 étapes pour la prise en compte de notre fonction :

- **La déclaration de la fonction** : le prototype minimum pour dire au compilateur qu'elle existe
- **La définition de la fonction** : tout le code de la fonction, comme dans le programme ci-dessus

La déclaration de la fonction doit inclure tous les types concernés : le type de retour et le type de ses paramètres, en utilisant la même syntaxe que celle utilisée dans la définition de la fonction, mais en remplaçant le corps de la fonction (le bloc d'instructions) par un point-virgule de fin.

La liste des paramètres n'a pas besoin d'inclure les noms de paramètres, mais seulement leurs types. Les noms de paramètres peuvent néanmoins être précisés, mais ils sont facultatifs et ne doivent pas correspondre nécessairement à ceux de la définition de la fonction.

Par exemple, une fonction appelée **protofunction** avec deux paramètres int peut être déclarée avec l'une ou l'autre de ces **déclarations** :

```
1 int protofunction (int first, int second);
2 int protofunction (int, int);
```

Donner un nom bien choisi pour chaque paramètre améliore aussi la lisibilité du programme, donc c'est conseillé.

Le nombre de paramètres de la fonction et le type de ces paramètres est appelé signature de la fonction.

Exemple :

```
void odd (int x);   //déclaration de la fonction odd (impair)

int main()
{
  int i;
  do {
    cout << "Please, enter number (0 to exit): ";
    cin >> i;
    odd (i);        // appel de la fonction odd
  } while (i!=0);
  return 0;
}

void odd (int x)    //définition de la fonction odd
{
  if ((x%2)!=0) cout << "It is odd.\n";
  else cout << "It is even.\n";
}
```

Il faut donc faire comme cela : **déclarer le prototype des fonctions**. Il contient déjà tout ce qui est nécessaire pour les utiliser : leur nom, les types de leurs arguments, et leur type de retour. Après la déclaration de leur prototype, elles peuvent être appelées avant qu'elles ne soient entièrement définies, et on peut donc écrire la définition complète à un autre endroit : après le main, dans un autre fichier, etc. Nous verrons la meilleure méthode d'organisation des programmes et fonctions dans un cours prochain.

9 EXERCICES :

9.1 Exercice 1

Dans le programme ci-dessous on va remplacer les lignes encadrées par un appel d'une fonction **calculEmballages()**.

Cette fonction calcule le nombre d'emballages (**emballages**) et le nombre d'articles qui restent (**reste**).

Cette fonction a besoin de connaître le nombre d'articles (**articles**) et le nombre d'articles par emballage (**articlesParEmballages**).

On utilisera le return pour le nombre d'emballages, et pour les autres variables le passage par les paramètres.

```
int main()
{
    int reste;                      // déclaration sans initialisation
    int emballages = 0;             // déclaration et initialisation style C
    int articles(1);                // déclaration et initialisation style C++ constructeur
    int articlesParEmballage{ 1 };  // déclaration et initialisation style C++ uniformisé
                                    // les 3 formes de déclaration-init sont équivalentes

    cout << "Entrez le nombres d'articles à emballer (au moins 1) : ";
    cin >> articles;
    cin.ignore();

    cout << "Entrez le nombres d'articles par emballage (au moins 1) : ";
    cin >> articlesParEmballage;
    cin.ignore();

    reste = articles % articlesParEmballage;
    emballages = (articles - reste) / articlesParEmballage;

    cout << "Il faut commander " << emballages << " emballages\n";
    cout << "Il va rester " << reste << " article(s) \n";

    system("pause");
}
```

9.2 Exercice 2

Dans une usine de fabrication de vis on a besoin de calculer la masse d'acier nécessaire pour la fabrication de vis sans tête de longueurs différentes. Il existe 3 diamètres possibles (6 ,8, 10mm), et pour chaque diamètre 3 longueurs (50, 55, 60mm). Donc l'utilisateur saisit le diamètre qu'il veut fabriquer, et le programme lui donne la masse d'acier totale pour fabriquer 1000 vis de chaque longueur pour le diamètre donné. Masse volumique de l'acier : 8000Kg/m^3.

Contraintes :

- Pour le calcul proprement dit (qui est toujours le même pour un diamètre et une longueur donnés) on utilisera une fonction qui sera appelée plusieurs fois dans le programme.
- La fonction est d'abord déclarée en début de programme, et n'est définie qu'après le main en fin de programme.
- Les 3 longueurs possibles sont des variables constantes définies en début de main
- La fonction retourne la masse d'acier nécessaire pour 1000 vis de chaque longueur pour un diamètre donné (approximativement un cylindre)
- Les paramètres de la fonction sont :
    - Le diamètre en mm (passage par valeur)
    - La première longueur (passage par valeur)
    - La deuxième longueur (passage par valeur)
    - La troisième longueur (passage par valeur)
    - La masse d'une vis en première longueur (calculé par la fonction, donc passage par référence)
    - La masse d'une vis en deuxième longueur (calculé par la fonction, donc passage par référence)
    - La masse d'une vis en troisième longueur (calculé par la fonction, donc passage par référence)
- La fonction principale affiche : le diamètre choisi, la première longueur et la masse de la vis, la deuxième longueur et la masse de la vis, la troisième longueur et la masse de la vis, la masse totale d'acier.