TP C++ pré-requis : 3° partie

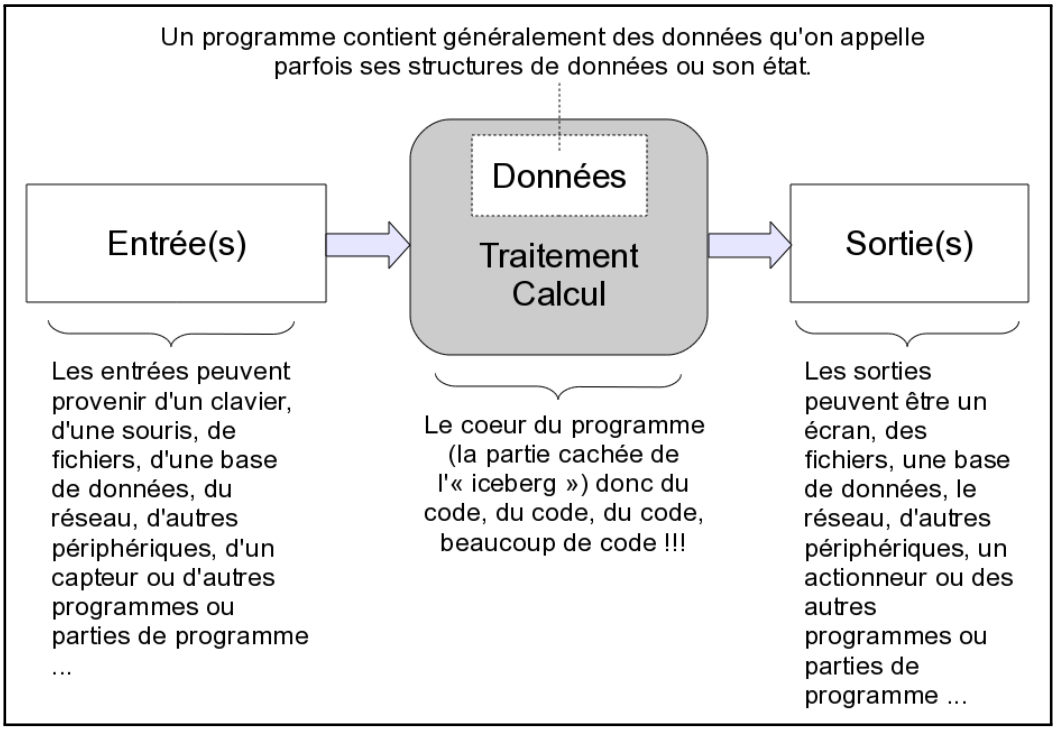
Sommaire

Les objectifs de ce tp sont de comprendre et mettre en oeuvre la fabrication d'un programme simple. Beaucoup de conseils sont issus du livre de référence de Bjarne Stroustrup (www.programmation.stroustrup.pearson.fr).

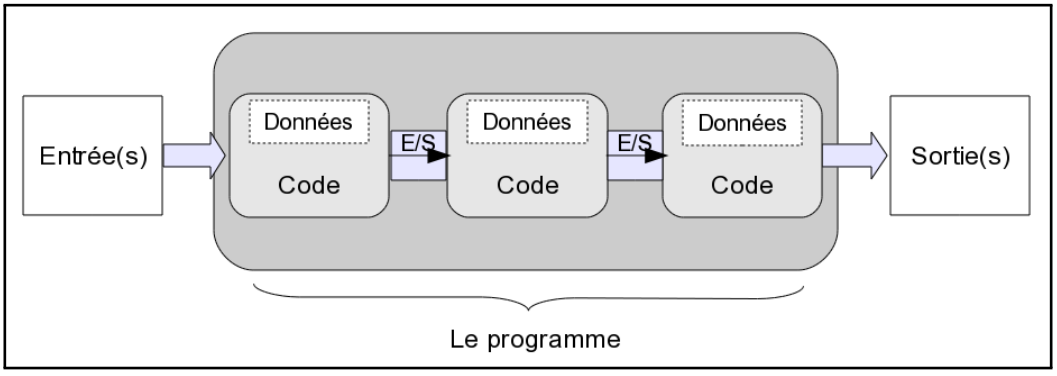
Remarque : les TP *ont pour but d'établir ou de renforcer vos compétences pratiques. Vous pouvez penser que vous comprenez tout ce que vous lisez ou tout ce que vous a dit votre enseignant mais la répétition et la pratique sont nécessaires pour développer des compétences en programmation. Ceci est comparable au sport ou à la musique ou à tout autre métier demandant un long entraînement pour acquérir l'habileté nécessaire. Imaginez quelqu'un qui voudrait disputer une compététion dans l'un de ces domaines sans pratique régulière. Vous savez bien quel serait le résultat.*

Troisième partie : Programmation modulaire

D'un certain point de vue, un programme informatique ne fait jamais rien d'autre que **traiter des données**. Comme on l'a déjà vu dans la deuxième partie, un programme accepte des entrées et produit des sorties :



La majeure partie du travail d'un programmeur est : comment exprimer un programme sous la forme d'un ensemble de parties qui coopèrent et comment peuvent-elles partager et échanger des données ?



Les E/S signifie évidemment "entrées/sorties" : la sortie d'une partie de code est l'entrée de la suivante.

Un **traitement est tout simplement l'action de produire des sorties à partir d'entrées**. Les entrées dans une partie de programme sont souvent appelées des **arguments** (ou parfois paramètres) et les sorties d'une partie de programme des **résultats**.

Par parties de programme (ou de code), on entend des entités comme une **fonction** produisant un résultat à partir d'un ensemble d'arguments en entrée.

Exemple : un traitement comme produire le résultat (sortie) 7 à partir de l'argument (entrée) 49 au moyen de la fonction `racineCarree`.

Objectifs et outils

Le métier de programmeur consiste à écrire des programmes qui :

– donnent des résultats corrects
– sont simples
– sont efficaces

L'ordre donné ici est très important : peu importe qu'un programme soit rapide si ses résultats sont faux. De même, un programme correct et efficace peut être si compliqué et mal écrit qu'il faudra le jeter ou le récrire complètement pour en produire une nouvelle version. N'oubliez pas que les progremmes utiles seront toujours modifiés pour répondre à de nouveaux besoins.

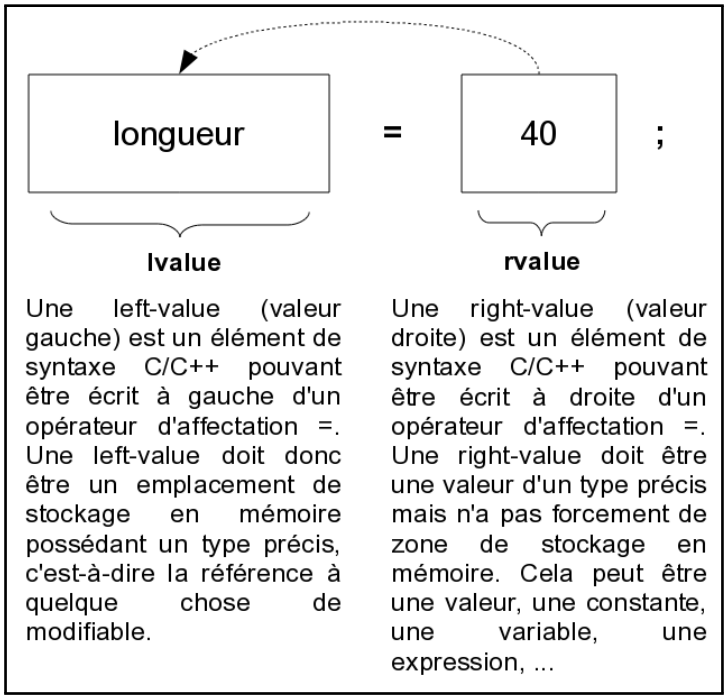
Un programme (ou une fonction) doit s'acquitter de sa tâche de façon aussi simple que possible.

Nous acceptons ces principes quand nous décidons de devenir des professionnels. En termes pratiques, cela signifie que nous ne pouvons pas nous contenter d'aligner du code jusqu'à ce qu'il ait l'air de fonctionner : nous devons nous soucier de sa structure. Paradoxalement, le fait de s'intéresser à la structure et à la "qualité du code" est souvent le moyen le plus facile de faire fonctionner un programme.

Notre principal outil pour organiser un programme est de décomposer un traitement de grande taille en plusieurs parties plus petites jusqu'à obtenir quelque chose de suffisamment simple à comprendre et à résoudre (cf. Descartes et son discours de la méthode).

Expressions et instructions

La brique de base la plus élémentaire d'un programme est une **expression**. Une expression calcule une **valeur** à partir d'un certain nombre d'opérandes. Cela peut être une **valeur littérale** comme 10, 'a', 3.14, "rouge" ou le **nom d'une variable**.




```
// calcule une aire
int longueur = 40;
int largeur = 20;
int aire = longueur * largeur;
```

Il y a plusieurs sortes d'instructions : les déclarations, les instructions d'expression, les instructions conditionnelles, les instructions itératives (boucles), etc ...

Une **instruction d'expression** est une expression suivie d'un point-virgule (;). Le point-virgule (;) est un élément syntaxique permettant au compilateur de "comprendre" ce que l'on veut faire dans le code (comme la ponctuation dans la langue française).

Dans les programmes comme dans la vie, il faut souvent choisir entre plusieurs possibilités. Le C/C++ propose plusieurs **instructions conditionnelles** : l'instruction `if` ou l'instruction `switch`.

Une instruction `if` choisit entre deux possibilités : si la condition est vraie, la première instruction (ou bloc d'instructions) est exécutée sinon c'est la seconde.

```
if(feuxPieton == vert) traverser();
else attendre();
```

La notion de base est donc simple mais il est également facile d'utiliser `if` de façon trop simpliste.

Voici un exemple simple de programme de conversion cm/*inch* qui utilise une instruction `if` :

```
int main()
{
    const double conversion = 2.54; // nombre de cm pour un pouce (inch)
    int longueur = 1;             // longueur (en cm ou en in)
    char unite = 0; // 'c' pour cm ou 'i' pour inch
    cout << "Donnez une longueur suivi de l'unité (c ou i):\n";
    cin >> longueur >> unite;

    if (unite == 'i')
        cout << longueur << " in == " << conversion*longueur << " cm\n";
    else
        cout << longueur << " cm == " << longueur/conversion << " in\n";
}
```

Conversion cm/inch (version 1)

En fait cet exemple semble seulement fonctionner comme annoncé. Ce programme dit que si ce n'est pas une conversion en *inch* c'est forcément une conversion en cm. Il y a ici une dérive sur le comportement de ce programme si l'utilisateur tape '`f`' car il convertira des cm en *inches* ce qui n'est probablement pas ce que l'utilisateur désirait. Un programme doit se comporter de manière sensée même si les utilisateurs ne le sont pas.

Voici une version améliorée en utilisant une instruction `if` imbriquée dans une instruction `if` :

```
int main()
{
    const double conversion = 2.54; // nombre de cm pour un pouce (inch)
    int longueur = 1;             // longueur (en cm ou en in)
    char unite = 0; // 'c' pour cm ou 'i' pour inch
    cout << "Donnez une longueur suivi de l'unité (c ou i):\n";
    cin >> longueur >> unite;

    if (unite == 'i')
```



```
        cout << longueur << " in == " << conversion*longueur << " cm\n";
    else if (unite == 'c')
        cout << longueur << " cm == " << longueur/conversion << " in\n";
    else
        cout << "Désolé, je ne connais pas cette unité " << unite << endl;
}
```

Conversion cm/inch (version 2)

De cette manière, vous serez tenter d'écrire des tests complexes en associant une instruction `if` à chaque condition. Mais, rappelez-vous, le but est d'écrire du code simple et non complexe.

En réalité, la comparaison d'unité à 'i' et à 'c' est un exemple de la forme de sélection la plus courante : une sélection basée sur la comparaison d'une valeur avec plusieurs constantes. Le C/C++ fournit pour cela l'instruction `switch`.

```
int main()
{
    const double conversion = 2.54; // nombre de cm pour un pouce (inch)
    int longueur = 1;             // longueur (en cm ou en in)
    char unite = 0; // 'c' pour cm ou 'i' pour inch
    cout << "Donnez une longueur suivi de l'unité (c ou i):\n";
    cin >> longueur >> unite;

    switch (unite)
    {
      case 'i':
        cout << longueur << " in == " << conversion*longueur << " cm\n";
        break;
      case 'c':
        cout << longueur << " cm == " << longueur/conversion << " in\n";
        break;
      default:
        cout << "Désolé, je ne connais pas cette unité " << unite << endl;
        break;
    }
}
```

Conversion cm/inch (version 3)

L'instruction `switch` utilisée ici sera toujours plus claire que des instructions `if` imbriquées, surtout si l'on doit comparer à de nombreuses constantes.

Vous devez garder en mémoire ces particularités quand vous utilisez un `switch` :

- la valeur utilisée pour le `switch()` doit être un entier, un `char` ou une énumération (on verra cela plus tard). Vous ne pourrez pas utiliser un `string` par exemple.
- les valeurs des étiquettes utilisées dans les `case` doivent être des expressions constantes (voir plus loin). Vous ne pouvez pas utiliser de variables.
- vous ne pouvez pas utiliser la même valeur dans deux `case`
- vous pouvez utiliser plusieurs `case` menant à la même instruction
- l'erreur la plus fréquente dans un `switch` est l'oubli d'un `break` pour terminer un case. Comme ce n'est pas une obligation, le compilateur ne détectera pas ce type d'erreur.

Il est rare de faire quelque chose une seule fois. C'est pour cela que tous les langages de programmation fournissent des moyens pratiques de faire quelque chose plusieurs fois (on parle de traitement itératif). On appelle cela une **boucle** ou une **itération**.

Le C/C++ offrent plusieurs **instructions itératives** : la boucle `while` (et sa variante `do ... while`) et la boucle `for`.

Le premier programme jamais exécuté sur un ordinateur à programme stocké en mémoire (l'EDSAC) est un exemple d'itération. Il a été écrit et exécuté par David Wheeler au laboratoire informatique de Cambridge le 6 mai 1949 pour calculer et afficher une simple liste de carrés comme ceci :

```
0  0
1  1
2  4
3  9
4  16
...
98 9604
99 9801
```

Ce premier programme n'a pas été écrit en C/C++ mais le code devait ressembler à ceci :

```
// Calcule et affiche le carré d'un nombre

#include <iostream>
#include <cmath>

using namespace std;

int main()
{
   int i = 0; // commencer à 0

   // tant que i est inférieur strict à 100 : on s'arrête quand i a atteint la valeur 100
   while (i < 100)
   {
      cout << i << '\t' << pow(i, 2) << '\n'; // affiche i et son carré séparés par une
         tabulation
      ++i; // on incrémente le nombre et on recommence
   }
}
```

Le premier programme jamais écrit (version while)

Les accolades (`{}`) délimitent le **corps de la boucl**e : c'est-à-dire le bloc d'instructions à répéter. La condition pour la répétition est exprimée directement dans le `while`.

Donc écrire une boucle est simple. Mais cela peut s'avérer dangereux :
- Que se passerait-il si `i` n'était pas initialisé à `0` ? Voilà une première raison qui démontre que les variables non initialisées sont une source d'erreurs courante.
- Que se passerait-il si on oubliait l'instruction `++i` ? On obtient une boucle infinie (un programme qui ne "répond" plus). Il faut éviter au maximum d'écrire des boucles infinies. Il est conseillé de ne pas coder ce type de boucle : `while(1)` ou `while(true)` qui sont des boucles infinies.

Itérer sur une suite de nombres est si courant en C/C++ que l'on dispose d'une instruction spéciale pour le faire. C'est l'instruction `for` qui très semblable à `while` sauf que la gestion de la variable de contrôle

de boucle est concentrée sur une seule ligne plus facile à lire et à comprendre. Il existe toujours une instruction `while` équivalente à une instruction `for`.

L'instruction `for` concentre : une zone d'initialisation, une zone de condition et une zone d'opération d'incrémentation. N'utilisez `while` que lorsque ce n'est pas le cas.

```
// Calcule et affiche le carré d'un nombre

#include <iostream>
#include <cmath>

using namespace std;

int main()
{
   // exécute le bloc d'instructions de la boucle :
   // avec i commençant à 0 (initialisation)
   // tant que i est inférieur strict à 100 (condition)
   // en incrémentant i après chaque exécution du bloc d'instruction (opération d'
      incrémentation)
   for (int i = 0; i < 100; i++)
      cout << i << '\t' << pow(i, 2) << '\n'; // affiche i et son carré séparés par une
         tabulation
}
```

Le premier programme jamais écrit (version for)

Constantes

Les programmes utilisent généralement beaucoup de constantes. Elles sont très importantes pour conserver un code lisible.

Le C++ offre la notion de **constante symbolique**, c'est-à-dire un objet nommé auquel on ne peut pas donner de nouvelle valeur une fois qu'il a été initialisé.

```
const double pi = 3.14159;
pi = 22/7; // Erreur : affectation d'une constante
const double v = 7*pi/r; // Ok : on ne fait que de lire la constante pi
```

Règle de codage : on évite d'utiliser des valeurs littérales dans le code et on s'oblige le plus possible à utiliser des constantes portant des noms significatifs.

Remarque : les valeurs littérales dans le code (en dehors des définitions const) sont qualifiées par dérision de constantes magiques.

Il est aussi possible de créer des constantes en utilisant l'instruction du préprocesseur `#define` :

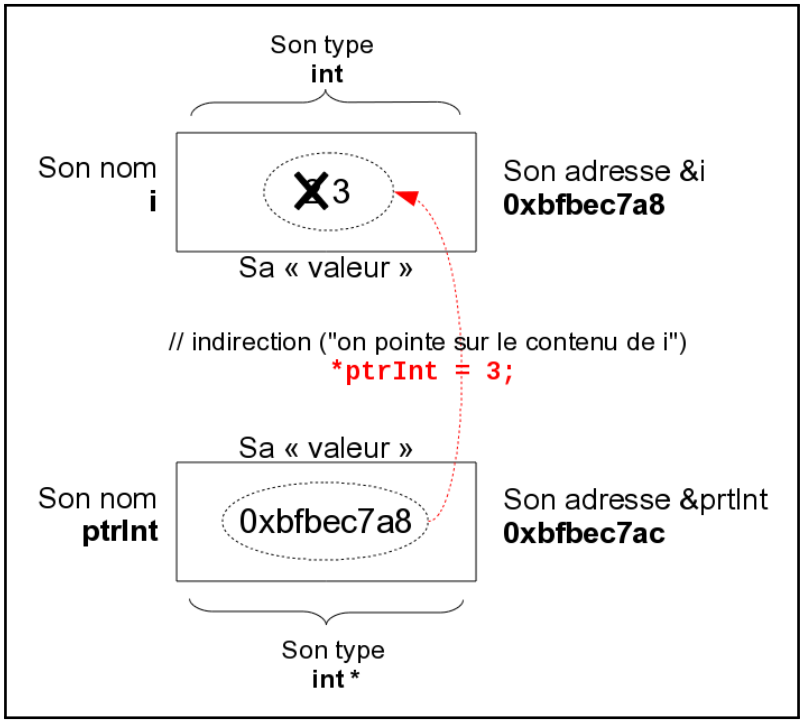
```
#define PI 3.14159
```

C'est simplement une **substitution de texte** (une sorte de copier/coller) qui est réalisée avant la compilation. Il n'y a aucun typage de la constante.

L'utilisation de `#define` améliore surtout la lisibilité du code source. La convention usuelle est d'utiliser des MAJUSCULES (pour les distinguer des variables).

Références et pointeurs

Les **pointeurs** sont des variables spéciales permettant de stocker une adresse (pour la manipuler ensuite). L'adresse représente généralement l'emplacement mémoire d'une variable (ou d'une autre adresse). Comme la variable adressée a un type, le pointeur qui stockera son adresse doit être du même type pour la manipuler convenablement.




```
// On utilise l'étoile (*) définir un pointeur
int *ptrInt; // ptrInt est un pointeur sur un entier (int)

// On utilise le & devant une variable pour avoir la valeur de son adresse et s'en servir
   pour initialiser ou affecter un pointeur
int i = 2;
ptrInt = &i; // affectation de ptrInt avec l'adresse de la variable i (&i)

// On utilise l'étoile devant le pointeur (*) pour accéder à l'adresse stockée
*ptrInt = 3; // indirection ("on pointe sur le contenu de i")

// et maintenant la variable i contient 3
```

Utilisation d'un pointeur

Les pointeurs peuvent être incrémentés, décrémentés, additionnés ou soustraits. Dans ce cas, leur nouvelle valeur d'adresse dépend du type sur lequel ils "pointent". Cela peut servir par exemple pour se déplacer sur la prochaine variable du même type dans la mémoire.

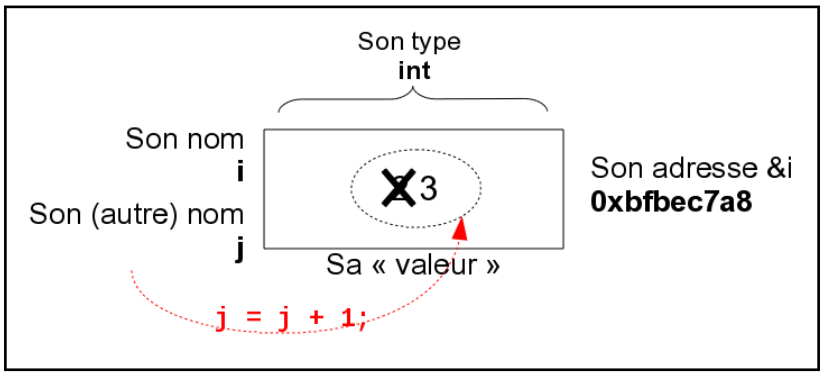
Règle de codage : on peut préfixer ses variables pointeurs avec `ptr`.

Remarque : les pointeurs font parties des variables les plus délicates à manipuler car, si on les utilise mal, on peut provoquer des erreurs d'accès mémoire. Certains langages (comme le Java) les ont interdits.

En C++ (pas en C), il est possible d'utiliser des **références** sur des objets (variables) définis ailleurs : cela permet de créer un nouveau nom qui sera un synonyme de cette variable (comme un alias ou un raccourci). On pourra donc modifier le contenu de la variable en utilisant une référence sur celle-ci. Une référence ne peut être initialisée qu'une seule fois. Elle ne peut donc référencer qu'une seule variable tout au long de sa durée de vie.

```
int i = 2; // i est un entier valant 2

// On indique le & devant le nom de la variable et cela signifie ‘‘référence’’
int &j = i; // j est une référence sur un entier et cet entier est i

//&j = x; // Erreur : illégal (on ne peut pas affecter une nouvelle variable à une référence
    déjà initialisée)
//int &k = 44; // Erreur : illégal (on ne peut pas créer une référence sur une valeur)

// A partir d'ici j est ‘‘synonyme‘‘ de i, ainsi :
j = j + 1; // est équivalent à i = i + 1 !

// et maintenant la variable i contient 3
```

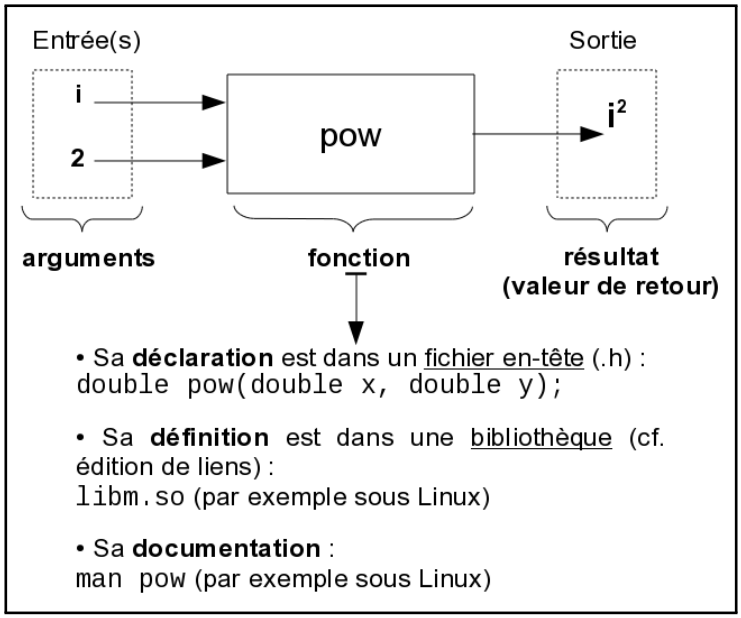
Utilisation d'une référence



Remarque : les références sont très utilisées en C++, notamment dans le passage des arguments d'une fonction. On peut aussi ajouter le mot clé `const` *pour interdire la fonction de modifier (accidentellement) la variable passée en argument.*

Fonctions

Dans le programme précédent (celui qui calcule et affiche le carré d'un nombre), `pow(i, 2)` est un **appel de fonction**. Plus précisément, c'est un appel à une fonction nommée **pow** avec les **arguments** `i` et **2** et qui retourne le **résultat** de `i` élévé à la puissance 2.



L'appel `pow(i, 2)`

La **bibliothèque standard** fournit beaucoup de fonctions utiles, comme la fonction `pow()`. Toutefois, nous écrirons de nombreuses fonctions nous-mêmes.

Voici une **définition** plausible d'une fonction `carre` :

```
// Calcule et affiche le carré d'un nombre

#include <iostream>

using namespace std;

int carre(int x)
{
   return x*x;
}

int main()
{
   for (int i = 0; i < 100; i++)
      cout << i << '\t' << carre(i) << '\n'; // affiche i et son carré séparés par une
         tabulation

   return 0;
}
```

Notre fonction carre()

Le programmeur professionnel se doit de maîtriser les concepts sur les fonctions :

La **déclaration** d'une fonction est une **instruction** fournissant au compilateur un certain nombre d'informations concernant une fonction. Elle sert à dire au compilateur qu'une fonction de nom `carre` existe, qu'elle doit recevoir un entier (`int`) et qu'elle retournera un entier (`int`).

Il existe une forme recommandée dite **prototype** qui comprend sa **signature** et son **type de retour** :

```
int carre ( int );
```

type de retour Si on ne veut pas retourner de résultat, on spécifie **void** (qui signifie « rien »)

le nom de la fonction

liste de paramètres Ses éléments se nomment des paramètres. On indique leur type mais leur nom est facultatif dans une déclaration. Et la liste peut être **vide** ().

Déclaration de fonction

La **définition** d'une fonction revient à écrire le **corps** de la fonction (qui définit donc les traitements effectués dans le bloc {}). Les paramètres de la liste doivent être nommées comme des variables.

La définition d'une fonction tient lieu de déclaration. Mais elle doit être "connue" avant son utilisation (un appel). Sinon, le compilateur génèrera un message d'avertissement (*warning*) qui indiquera qu'il a lui-même fait une déclaration implicite de cette fonction.

```
int carre ( int x)
{
   return x*x;
}
```

Définition de fonction

Remarque : une fonction qui ne retourne pas de résultat (donc void) se nomme une ***procédure****.*

```
// Un appel à une fonction signifie qu'on lui demande d'exécuter son traitement
// Il doit correspondre à la déclaration faite au compilateur qui vérifie :
// on n'est pas obligé d'utiliser le résultat de retour mais on doit donner à la fonction
    exactement les arguments qu'elle exige

carre(2); // Probablement une erreur car la valeur retour n'est pas utilisée

int c1 = carre(); // Erreur : argument manquant

int c2 = carre; // Erreur : parenthèses manquantes

int c3 = carre(1, 2); // Erreur : trop d'arguments

int c4 = carre("deux"); // Erreur : mauvais type d'argument car int attendu

int c5 = carre(2); // Ok : enfin !
```

L'appel à notre fonction carre()

On définit une fonction lorsqu'on souhaite un traitement distinct et nommé parce que procéder ainsi :

- rend le traitement distinct du point de vue logique
- rend le programme plus clair et plus lisible
- permet d'utiliser la fonction à plusieurs endroits dans un programme (à chaque fois qu'on en a besoin)
- facilite les tests (on simule des entrées et on compare le résultat obtenu à celui attendu)

Les programmes sont généralement plus facile à écrire, à comprendre et à maintenir lorsque chaque fonction réalise **une SEULE ACTION logique et bien évidemment celle qui correspond à son nom**.

Règle de codage : on évite d'utiliser des saisies et des affichages dans les fonctions pour permettre notamment leur ré-utilisation. Les fonctions se concentrent sur le traitement et n'ont pas à réaliser la saisie de leurs entrées et l'affichage de leur sortie. C'est une "bonne pratique" à respecter dès maintenant.

Voici ce qu'il ne faut pas faire :

```
// Calcule et affiche le carré d'un nombre saisi par l'utilisateur

#include <iostream>
using namespace std;

void saisirUnNombreEtAfficherSonCarre()
{
   int x;
   cout << "Donnez un nombre : ";
   cin >> x;
   cout << "Le carré de ce nombre est " << x*x;
}

int main()
{
   saisirUnNombreEtAfficherSonCarre();

   return 0;
}
```

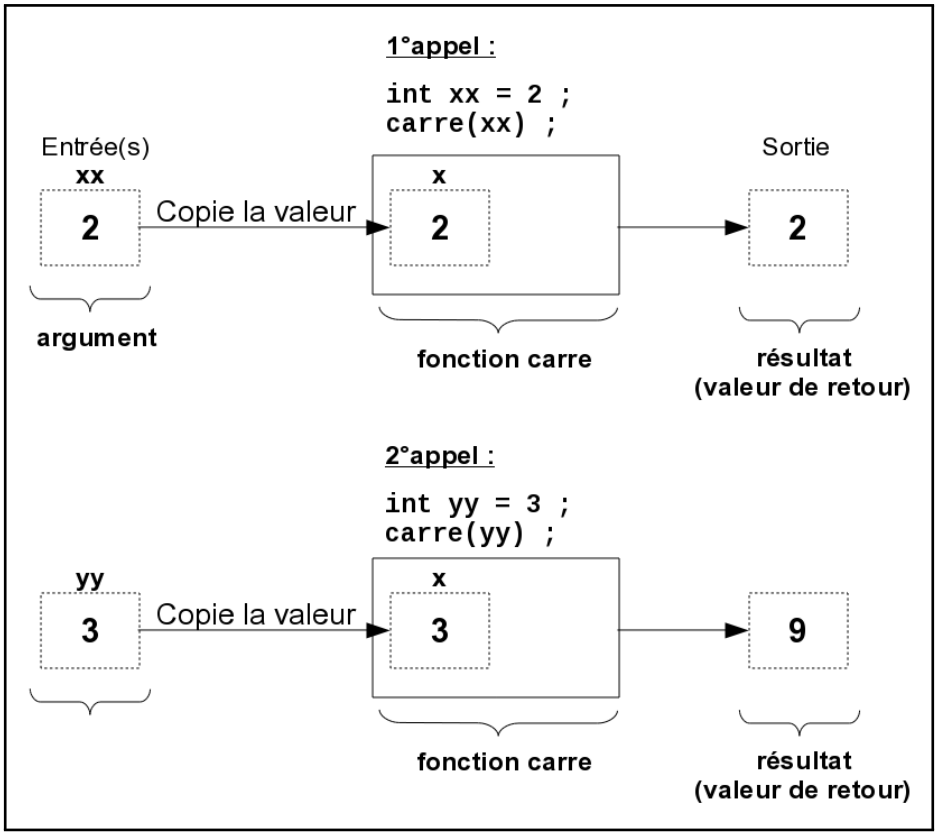
Mauvais exemple

Remarque : les fonctions qui ne recoivent aucune entrée et qui ne produisent aucun résultat en retour sont intellectuellement suspicieuses (des "trous noirs" de code !). Il faut les éviter au maximum.

Voici un code d'utilisation de la fonction `carre()` bien mieux structuré :

```
// Calcule et affiche le carré d'un nombre saisi par l'utilisateur

#include <iostream>

using namespace std;

int carre(int x)
{
   return x*x;
}

int main()
{
   int x;
   cout << "Donnez un nombre : ";
   cin >> x;
   cout << "Le carré de ce nombre est " << carre(x);
   return 0;
}
```

Bon exemple



Passage par valeur

Remarque : bien évidemment, on peut passer les arguments à une fonction par ***référence*** *ou par* ***pointeur*** *(voir cours et exercices plus loin).*

Règles de codage

Un nom de fonction est construit à l'aide d'un **verbe** (pas un nom), et éventuellement d'éléments supplémentaires :
– une quantité
– un complément d'objet
– un adjectif représentatif d'un état

On utilisera la convention suivante : **un nom de fonction commence par une lettre minuscule puis les différents mots sont repérés en mettant en majuscule la première lettre d'un nouveau mot. Un nom de fonction peut commencer par une majuscule dans des cas que l'on précisera plus tard.**

Le verbe peut être au présent de l'indicatif ou à l'infinitif. L'adjectif représentatif d'un état concerne surtout les fonctions booléennes. La quantité peut, le cas échéant, enrichir le sens du complément.

Exemples : `void ajouter()`, `void sauverValeur()`, `estPresent()`, `estVide()`, `viderAMoitieLeReservoir()`, ...

Rappel : Les mots clés du langage sont interdits comme noms.

Les noms des paramètres d'une fonction sont construits comme les noms de variables : ils commencent, notamment par une minuscule. L'ordre de définition des paramètres doit respecter la règle suivante :

```
nomFonction(parametrePrincipal, listeParametres)
```

où `parametrePrincipal` est la donnée principale sur laquelle porte la fonction, la `listeParametres` ne comportant que des données secondaires, nécessaires à la réalisation du traitement réalisé par la fonction.

Exemple : `ajouter(EnsembleMesures mesures, float uneMesure)`

La sémantique de cette fonction est d'ajouter une `mesure` à un ensemble de `mesures` (qui est la donnée principale) et non, d'ajouter un ensemble de `mesures` à une `mesure`.

Remarque : l'objectif de respecter des règles de codage est d'augmenter la lisibilité des programmes en se rapprochant le plus possible d'expressions en langage naturel.

Pour finir, on va s'ajouter une règle qui couvre l'objectif majeur suivant : décomposer un traitement de grande taille en plusieurs parties plus petites jusqu'à obtenir quelque chose de suffisamment simple à comprendre et à résoudre (cf. Descartes et son discours de la méthode). Pour cela, on va **s'obliger à limiter la taille des fonctions à une valeur comprise entre 10 à 15 lignes maximum**.

Questions de révision

L'idée de base des questions de révision est de vous donner une chance de voir si vous avez identifié et compris les points clés de cette partie.

Question 1. Qu'est-ce qu'un traitement ?

Question 2. Qu'entend-on par entrées et sorties d'un traitement. Donnez des exemples.

Question 3. Quelles sont les trois exigences qu'un programmeur doit garder présentes à l'esprit en exprimant des traitements ?

Question 4. Qu'est-ce qu'une *lvalue* ?

Question 5. Qu'est-ce qu'une constante symbolique et pourquoi les utilise-t-on ?

Question 6. Quand un programmeur préférerait-il une instruction `switch` à une instruction `if` ?

Question 7. Quelle est la fonction de chaque partie de la ligne d'en-tête d'une boucle `for` et dans quel ordre sont-elles exécutées ?

Question 8. Quand doit-on utiliser la boucle `for` et quand doit-on utiliser la boucle `while` ?

Question 9. Êtes-vous capable de faire la différence entre une déclaration et une définition de fonction ?

Question 10. Décrivez ce que la ligne `char foo(int x, int y)` signifie dans une définition de fonction.

Exercice 4 : programmation modulaire

Éditez le programme `somme1-100.cpp` dans votre répertoire de travail, étudiez son fonctionnement, puis fabriquez-le, et testez l'exécutable ainsi produit.

```
#include <iostream> // Inclusion des déclarations de cout, endl, etc.

using namespace std;

//-------------------------------------------------------------//
//  somme1-100.cpp                                             //
//                                                             //
//  Calcul de la somme des entiers de 1 à 100                  //
//-------------------------------------------------------------//
int main ()
{
 unsigned int i=0U; // variable de contrôle, valeur initiale : zéro.
 unsigned int accumulateur=0U; // la somme des entiers, initialement zéro.

 // calcul : utilise un traitement itératif : la boucle
 // while est l'une des instructions possibles en C/C++
 while (i <= 100U)
 {
    accumulateur = accumulateur + i;
    i = i + 1; // Avec un peu d'habitude, on écrira plutot ++i
 }

 // affichage du résultat
 cout << "La somme des entiers de 1 a 100 est " << accumulateur << endl;

 return 0;
} // fin du main()
```

Question 11. Remplacez la valeur "100" utilisée par le programme par une constante nommée `MAXIMUM` définie avant le début du programme et de même valeur. Une habitude courante des programmeurs C/C++, est de nommer les constantes en majuscules ; pensez à respecter cette règle vous aussi.

Question 12. Modifiez de nouveau le programme pour ne plus utiliser une constante mais une variable nommée désormais `maximum` (en minuscule puisque ce n'est plus une constante !). Modifiez le programme pour qu'il demande à l'utilisateur de donner une valeur pour le maximum avant le début du calcul de somme.

Question 13. En déplaçant la portion de programme qui réalise la somme (et uniquement le calcul), réalisez une fonction `unsigned int somme1aN(unsigned int n)` qui calcule la somme des entiers de 1 à la valeur n fournie en paramètre et qui renvoie cette somme comme valeur de retour de la fonction. Définissez cette fonction au dessus de votre `main()`. Pour que votre réponse au problème soit la bonne, il faut que vous n'ayez pas ni de `cin` ni de `cout` dans votre fonction : elle ne fait que le calcul. Modifiez votre programme principal (c'est-à-dire la fonction `int main()`) pour que la fonction `somme1aN()` soit maintenant appelée et que le `main()` affiche son résultat.

Question 14. Rendez votre programme modulaire : déplacez la fonction `unsigned int somme1aN(unsigned int n)` dans un nouveau fichier `somme.cpp` ; créez un fichier `somme.h` contenant la déclaration de cette même fonction ; incluez ce fichier `somme.h` dans le fichier qui contient `int main()`. Compilez séparemment chaque fichier `.cpp` en un fichier `.o`, puis faites l'éditions de liens de tous les `.o` en un exécutable. Testez.

Question 15. En vous inspirant de la fonction `unsigned int somme1aN(unsigned int n)`, créez maintenant une fonction `unsigned int factorielle(unsigned int n)`.
On rapelle que `somme1aN(n)=0+0+1+2+3+...+(n-1)+n` et que `factorielle(n)=1*1*2*3*...*(n-1)*n`. On remarquera que les termes soulignés sont les valeurs initiales de l'accumulateur. Les termes qui suivent cette valeur initiale correspondent chacun à un tour de boucle. Pensez bien que la valeur de n peut être 0, et qu'on peut très bien ne faire aucun tour de la boucle !

Question 16. Rajoutez une boucle `for()` dans le `main()` qui appelle la fonction `factorielle()` pour toutes les valeurs comprises entre 1 et celle choisie par l'utilisateur et affiche la valeur de la factorielle pour chaque valeur de n. Éditez ces modifications dans un fichier `main.cpp`. Fabriquez et testez. Que se passe t'il quand n dépasse la valeur 20 ? Pourquoi ?

Les étapes de fabrication peuvent être automatisées en utilisant l'outil `make` et en écrivant les règles à appliquer dans un fichier `Makefile` :

```
main: main.o somme.o
   g++ -o main main.o somme.o
main.o: main.cpp somme.h
   g++ -c main.cpp
somme.o: somme.cpp somme.h
   g++ -c somme.cpp
```

Attention : il faut une TABULATION et non des espaces devant les lignes commençant par `g++`.

Question 17. Modifiez un après l'autre les fichiers concernés (`main.cpp`, `somme.h` et `somme.cpp`) et exécutez `make` à chaque fois pour observer ses actions. Quel est l'intérêt de `make` dans la programmation modulaire ?

Exercice 5 : passage par adresse

On définit trois nombres réels `a`, `b`, `c` simple précision (`float`) dans le `main()`, et on leur attribue des valeurs arbitraires, par exemple `a=11.5`, `b=-2.1` et `c=0.0`. On désire écrire une fonction `ordonne3()` qui puisse manipuler ces trois variables passées en paramètre, et faire en sorte qu'après son appel, on ait toujours la situation $a \leq b \leq c$, éventuellement en permutant leurs valeurs.

Question 18. Afin de simplifier le problème, on décide d'écrire d'abord une fonction `echange2ParAdresse()` qui effectue le tri sur seulement deux paramètres à la fois. Ecrivez cette fonction puis testez la sur deux variables du `main()`.

Question 19. Une fois que `echange2ParAdresse()` est au point, écrivez `ordonne3()` de telle façon qu'elle fasse un certain nombre d'appels à `echange2ParAdresse()`. Combien faut-il d'appel (au minimum) à `echange2ParAdresse()` pour être certain de la justesse du résultat ?

Pour être certain que votre solution soit juste, il faut que vous ne fassiez les affichages que dans le `main()`. Ce qui interdit en particulier d'en faire dans `echange2ParAdresse()` ou dans `ordonne3()`.

Exercice 6 : passage par référence

Question 20. On reprend la même question qu'à l'exercice 2, mais on décide d'utiliser des passages par référence à la place des passages par adresse. Laquelle des deux solutions vous parait la plus simple ? Peut-on utiliser des références en C ?

Bilan

D'un point de vue théorique, vous pouvez désormais faire tout ce qu'on peut faire avec un ordinateur, le reste n'est que détails ! Cela montre tout de même la valeur des "détails" et l'importance des compétences pratiques parce qu'il est clair que vous débutez à peine votre carrière de programmeur.

Conclusion : Il reste "seulement" à apprendre à écrire de bons programmes, c'est-à-dire des programmes corrects, simples et efficaces.

Rob Pike (ancien chercheur des Laboratoires Bell et maintenant ingénieur chez Google) :

> « Règle n°4 : Les algorithmes élégants comportent plus d'erreurs que ceux qui sont plus simples, et ils sont plus difficiles à appliquer. Utilisez des algorithmes simples ainsi que des structures de données simples. »

Cette règle n°4 est une des instances de la philosophie de conception KISS (*Keep it Simple, Stupid* dans le sens de « Ne complique pas les choses ») ou Principe KISS, dont la ligne directrice de conception préconise de rechercher la simplicité dans la conception et que toute complexité non nécessaire devrait être évitée.

