Les bases du langage C++

Sommaire

Introduction

Historique

- En 1970, **Ken Thompson**, créa un nouveau langage : Le B, descendant du BCPL (*Basic Combined Programming Language*, créé en 1967 par Martin Richards). Son but était de créer un langage simple, malheureusement, son langage fût trop simple et trop dépendant de l'architecture utilisée.
- En 1971, **Dennis Ritchie** commence à mettre au point le successeur du B, le **C**. Le résultat est convaincant : Le C est **totalement portable** (il peut fonctionner sur tous les types de machines et de systèmes), il est de **bas niveau** (il peut créer du code aussi rapide que de l'assembleur) et il permet de traiter des **problèmes de haut niveau**. Le C permet de quasiment tout faire, du driver au jeu. Le C devient très vite populaire.
- En 1989, l'**ANSI** (*American National Standards Institute*) normalisa le C sous les dénominations **ANSI C** ou **C89**. Un programme écrit dans ce standard est compatible avec tous les compilateurs.
- En 1983, **Bjarne Stroustrup** des laboratoires Bell crée le **C++**. Il construit donc le C++ sur la base du C. Il garde une forte compatibilité avec le C.
- En 1999, l'**ISO** (*International Organization for Standardization*) proposa une nouvelle version de la norme, qui reprenait quelques bonnes idées du langage C++. Il ajouta aussi le type `long long` d'une taille minimale de 64 bits, les types complexes, l'initialisation des structures avec des champs nommés, parmi les modifications les plus visibles. Le nouveau document est celui ayant autorité aujourd'hui, est connu sous le sigle **C99**.


Les langages C et C++ sont les langages les plus utilisés dans le monde de la programmation.

Remarques :
- Certaines autres extensions du C ont elles aussi été standardisées, voire normalisées. Ainsi, par exemple, des fonctions spécifiques aux systèmes **UNIX**, sur lesquels ce langage est toujours très populaire, et qui n'ont pas été intégrées dans la norme du langage C, ont servi à définir une partie de la norme **POSIX**.
- Le langage C++ est normalisé par l'ISO. Sa première normalisation date de 1998 (**C++98**), puis une seconde en 2003. Le standard actuel a été ratifié et publié par ISO en septembre 2011 (**C++11**). Mais certains compilateurs ne la supportent pas encore complétement.


L'option `-std` des compilateurs gcc/g++ permet de choisir la norme à appliquer au moment de la compilation. Par exemple : `-std=c89` ou `c99` pour le C et `-std=c++98` ou `c++0x` pour le C++ (compilateur version 4.4.3)

Apports du C++ par rapport au C

Le C++ a apporté par rapport au langage C les notions suivantes :
- les **concepts orientés objet (encapsulation, héritage)** [1],
- les références, la vérification stricte des types,
- les valeurs par défaut des paramètres de fonctions,
- la surcharge de fonctions (plusieurs fonctions portant le même nom se distinguent par le nombre et/ou le type de leurs paramètres),
- la surcharge des opérateurs (pour utiliser les opérateurs avec les objets), les constantes typées,
- la possiblité de déclaration de variables entre deux instructions d'un même bloc

1. Les concepts orientés objet ne sont étudiés dans ce support de cours.

C → C++

- On peut passer progressivement de C à C++ : il suffit en premier lieu de **changer de compilateur (par exemple gcc → g++)**, sans nécessairement utiliser les nombreuses possibilités supplémentaires qu'offre C++.
- Le principal intérêt du passage de C à C++ est de profiter pleinement de la puissance de la **programmation orientée objet (POO)**.

```
#include <iostream> // pour cout, cin

int main (int argc, char **argv)
{
   int n;

   std::cout << "Donnez un entier : " << std::endl;
   std::cin >> n;

   for(int i = 0; i < n; i++) // remarque : déclaration de i seulement pour ce bloc
      std::cout << "Hello world !" << std::endl;

   return 0;
}
```

L'extension par défaut des fichiers écrits en langage C++ est `.cpp` ou `.cc`. Par défaut, gcc et g++ fabriquent un exécutable de nom `a.out` (ou `a.exe` sous Windows). Sinon, on peut lui indiquer le nom du fichier exécutable en utilisant l'option `-o <executable>`.

```
$ g++ <fichier.cpp>

$ ./a.out
Donnez un entier : 2
Hello world !
Hello world !
```

Étapes de fabrication

1. **Le préprocesseur** (pré-compilation)
   - Traitement des **directives** qui commencent toutes par le symbole dièse (`#`)
   - Inclusion de fichiers (`.h`) avec `#include`
   - Substitutions lexicales : les "macros" avec `#define`
2. **La compilation**
   - Vérification de la syntaxe
   - Traduction dans le langage d'assemblage de la machine cible
3. L'assemblage
   - Traduction finale en code machine (appelé ici code objet)
   - Production d'un fichier objet (`.o` ou `.obj`)
4. **L'édition de liens**
   - Unification des symboles internes
   - Étude et vérification des symboles externes (bibliothèques `.so` ou `.DLL`)
   - Production du programme exécutable

Étapes de fabrication avec `g++` :

Pré-compilation :

```
$ g++ -E <fichier.cpp> -o <fichier_precompile.cpp>
```

Compilation :

```
g++ -S <fichier_precompile.cpp> -o <fichier.s>
// ou
$ g++ -c <fichier.cpp> -o <fichier.o>
```

Assemblage :

```
$ as <fichier.s> -o <fichier.o>
```

Édition des liens :

```
$ g++ <fichier.o> -o <executable>
// ou
$ g++ <fichier1.o> <fichier2.o> <fichiern.o> -o <executable>
// voir aussi : $ ld -dynamic-linker <fichier.o> -o <executable>
```

Décomposition des étapes :

```
$ g++ -v -save-temps <fichier.cpp> -o <executable>
```

Fabrication :

Combiner en une seule ligne de commande toutes les étapes (préprocesseur, compilation, assemblage et édition des liens) : `$ g++ <fichier.cpp> -o <executable>`

Généralement dans le cas d'une **programmation modulaire** [2], on applique les instructions suivantes :

```
// Compilation modulaire (chaque fichier séparement) :
$ g++ -c <fichier1.cpp> -o <fichier1.o>
$ g++ -c <fichier2.cpp> -o <fichier2.o>
$ g++ -c <fichierN.cpp> -o <fichierN.o>

// Édition des liens :
$ g++ <fichier1.o> <fichier2.o> <fichierN.o> -o <executable>

// ou si on doit lier des bibliothèques (ici la bibliothèque math) :
$ g++ <fichier1.o> <fichier2.o> <fichierN.o> -o <executable> -lm
```

2. cf. `make` et `Makefile`

Manipuler des variables

Rappels :
- Une variable est un **espace de stockage pour un résultat**.
- Une variable est un **symbole** (habituellement un **nom** qui sert d'identifiant) qui renvoie à une **position de mémoire** (**adresse**) dont le contenu peut prendre successivement différentes valeurs pendant l'exécution d'un programme.

Règles de codage : Un nom de variable est un **nom** principal (surtout pas un verbe) suffisamment éloquent, éventuellement complété par :
- une caractéristique d'organisation ou d'usage
- un qualificatif ou d'autres noms

Exemples : `distance`, `distanceMax`, `consigneCourante`, `etatBoutonGaucheSouris`, `nbreDEssais`, ...

Exemple

```
#include <iostream> /* pour cout */
using namespace std;
int main() /* la fonction principale appelée automatiquement au lancement de l'exécutable */
{
   bool reussie = true; // true est une valeur booléenne
   int nombreDOeufs = 3; // 3 est une valeur entière
   unsigned long int jeSuisUnLong = 12345678UL; // U pour unsigned et L pour long
   float quantiteDeFarine = 350.0; // ".0" rend la valeur réelle
   char unite = 'g'; // ne pas oublier les quotes : ' '
   double const PI = 3.14159265358979323846; // une constante de type double
   cout << "La variable nombreDOeufs a pour valeur " << nombreDOeufs << endl;
   cout << "La variable jeSuisUnLong a pour valeur " << jeSuisUnLong << endl;
   cout << "La variable quantiteDeFarine a pour valeur " << quantiteDeFarine << endl;
   cout << "La variable unite a pour valeur " << unite << endl;
   cout << "Recette " << reussie << " : il faut " << nombreDOeufs << " oeufs et " <<
      quantiteDeFarine << " " << unite << " de farine" << endl;
   cout << "La constante PI a pour valeur " << PI << endl;
   return 0; /* fin normale du programme */
}
```

Les types entiers

- `bool` : false ou true → booléen (**seulement en C++**)
- `unsigned char` : 0 à 255 ($2^8 - 1$) → entier très court (1 octet ou 8 bits)
- `[signed] char` : -128 (-2^7) à 127 ($2^7 - 1$) → idem mais en entier relatif
- `unsigned short [int]` : 0 à 65535 ($2^{16} - 1$) → entier court (2 octets ou 16 bits)
- `[signed] short [int]` : -32768 (-2^{15}) à +32767 ($2^{15} - 1$) → idem mais en entier relatif
- `unsigned int` : 0 à 4.295e9 ($2^{32} - 1$) → entier sur 4 octets ; **taille "normale" actuelle**
- `[signed] int` : -2.147e9 (-2^{31}) à +2.147e9 ($2^{31} - 1$) → idem mais en entier relatif
- `unsigned long [int]` : 0 à 4.295e9 → entier sur 4 octets ou plus ; sur PC identique à "int" (hélas...)
- `[signed] long [int]` : -2.147e9 à -2.147e9 → idem mais en entier relatif
- `unsigned long long [int]` : 0 à 18.4e18 ($2^{64} - 1$) → entier (très gros !) sur 8 octets sur PC
- `[signed] long long [int]` : -9.2e18 (-2^{63}) à -9.2e18 ($2^{63} - 1$) → idem mais en entier relatif

Les types à virgule flottante

- `float` : Environ 6 chiffres de précision et un exposant qui va jusqu'à $\pm 10^{\pm 38}$ → Codage IEEE754 sur 4 octets
- `double` : Environ 10 chiffres de précision et un exposant qui va jusqu'à $\pm 10^{\pm 308}$ → Codage IEEE754 sur 8 octets
- `long double` → Codé sur 10 octets

Les constantes

- Celles définies **pour le préprocesseur** : c'est simplement une **substitution syntaxique pure** (une sorte de copier/coller). Il n'y a aucun typage de la constante.

```
#define PI 3.1415 /* en C traditionnel */
```

- Celles définies **pour le compilateur** : c'est une **valeur typée**, ce qui permet des contrôles lors de la compilation.

```
const double pi = 3.1415; // en C++ et en C ISO
```

Les pointeurs (rappel)

- 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 a un type, le pointeur qui stockera son adresse doit être du même type pour la manipuler convenablement.
- Le type `void*` représentera un **type générique** de pointeur : en fait cela permet d'indiquer sagement que l'on ne sait pas encore sur quel type il pointe.

Utilisation des pointeurs :

- On utilise l'étoile (*) pour **déclarer un pointeur**.

  déclaration d'un pointeur (*) sur un entier (int) : `int *ptrInt;`

- On utilise le & devant une variable pour **initialiser ou affecter un pointeur avec une adresse**.

  déclaration d'un entier i qui a pour valeur 2 : `int i = 2;`

  affectation avec l'adresse de la variable i (&i) : `ptrInt = &i;`

- On utilise l'étoile devant le pointeur (*) pour **accéder à l'adresse stockée**.

  indirection ("pointe sur le contenu de i") : `*ptrInt = 3;`

Maintenant la variable i contient 3

Les références en C++

- En C++, il est possible de déclarer une référence `j` sur une variable `i` : cela permet de créer un **nouveau nom `j` qui devient synonyme de `i`** (**un alias**).
- On pourra donc modifier le contenu de la variable en utilisant une référence.
- La déclaration d'une référence se fait en précisant le type de l'objet référencé, puis le symbole `&`, et le nom de la variable référence qu'on crée.
- Une référence ne peut être initialisée qu'une seule fois : à la déclaration.
- Une référence ne peut donc référencer qu'une seule variable tout au long de sa durée de vie.

Exemple :

```
#include <iostream>

int main (int argc, char **argv)
{
   int i = 10; // i est un entier valant 10
   int &j = i; // j est une référence sur un entier, cet entier est i.
   //int &k = 44; // ligne7 : illégal

   std::cout << "i = " << i << std::endl;
   std::cout << "j = " << j << std::endl;

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

   std::cout << "i = " << i << std::endl;
   std::cout << "j = " << j << std::endl;

   return 0;
}
```

```
i = 10
j = 10
i = 11
j = 11
```

Si on dé-commente la ligne 7, on obtient cette erreur à la compilation :

```
ligne 7: erreur: invalid initialization of non-const référence of type 'int&' from a
    temporary of type 'int'
```

le = dans la déclaration de la référence n'est pas réellement une affectation puisqu'on ne copie pas la valeur de `i`. En fait, on affirme plutôt le lien entre `i` et `j`. En conséquence, la ligne 7 est donc parfaitement illégal ce que signale le compilateur.

Intérêt des références :
- Comme une référence établit un lien entre deux noms, leur utilisation est efficace dans le cas de variable de grosse taille car cela évitera toute copie.
- Les références sont (systématiquement) utilisées dans le passage des paramètres d'une fonction (ou d'une méthode) dès que le coût d'une recopie par valeur est trop important ("gros" objet).
- Exemple :

```
void truc(const grosObjet& rgo);
```

Références et pointeurs constants

L'utilisation des mots clés `const` (et `volatile`) avec les pointeurs et les références est un peu plus compliquée qu'avec les types simples. En effet, il est possible de déclarer des pointeurs sur des variables mais aussi :
- des pointeurs constants sur des variables,
- des pointeurs sur des variables constantes et
- des pointeurs constants sur des variables constantes (idem avec les références).

La position des mots clés `const` (et `volatile`) dans les déclarations des types complexes est donc extrêmement importante.

Remarque : lors de l'analyse de la déclaration d'un identificateur X, il faut toujours commencer par une phrase du type « X est un ... ». Pour trouver la suite de la phrase, il suffit de lire la déclaration en partant de l'identificateur et de suivre l'ordre imposé par les priorités des opérateurs. Cet ordre peut être modifié par la présence de parenthèses.

Définition de synonymes de types `typdedef`

Le mot réservé `typedef` permet simplement la définition de **synonyme de type** qui peut ensuite être utilisé à la place d'un nom de type :

```
typedef int              entier;
typedef float            reel;
typedef enum{FALSE,TRUE} booleen;
entier a; // a de type entier donc de type int
reel   x; // x de type réel donc de type float
```

typedef est très utilisé car cela rend les code sources portables et lisibles, donc plus facile à maintenir et à faire évoluer.

Énumération `enum`

Une énumération est un type de données dont les valeurs sont des constantes nommées.

- `enum` permet de déclarer un **type énuméré** constitué d'un ensemble de constantes appelées **énumérateurs**.
- Une variable de type énuméré peut recevoir n'importe quel énumérateur (lié à ce type énuméré) comme valeur.
- Le premier énumérateur vaut zéro (par défaut), tandis que tous les suivants correspondent à leur précédent incrémenté de un.

Exemple d'utilisation de `enum` :

```
enum couleur_carte
{
    TREFLE = 1, /* un énumérateur */
    CARREAU, /* 1+1 donc CARREAU = 2 */
    COEUR  = 4, /* en C, les énumérateurs sont équivalents à des entiers (int) */
    PIQUE  = 8 /* il est possible de choisir explicitement les valeurs (ou de certaines d'
       entre elles). */
};

enum JourDeSemaine { LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE };
```

Exemple d'utilisation des `typedef` et `enum` précédents :

```
int main()
{
   entier e = 1;
   reel r = 2.5;
   booleen fini = FALSE;
   enum couleur_carte carte = CARREAU;
```



```
   printf("Le nouveau type entier possède une taille de %d octets (ou %d bits)\n", sizeof(
      entier), sizeof(entier)*8);
   printf("La variable e a pour valeur %d et occupe %d octets\n", e, sizeof(e));
   printf("La variable r a pour valeur %.1f et occupe %d octets\n", r, sizeof(r));
   printf("La variable fini a pour valeur %d et occupe %d octets\n", fini, sizeof(fini));
   printf("La variable carte a pour valeur %d et occupe %d octets\n", carte, sizeof(carte));
   return 0;
}
```

```
Le nouveau type entier possède une taille de 4 octets (ou 32 bits)
La variable e a pour valeur 1 et occupe 4 octets
La variable r a pour valeur 2.5 et occupe 4 octets
La variable fini a pour valeur 0 et occupe 4 octets
La variable carte a pour valeur 2 et occupe 4 octets
```

Les tableaux

Rappels :

- Un tableau est un **ensemble d'éléments de même type désignés par un identificateur unique** (un nom).
- Chaque élément est repéré par une valeur entière appelée **indice** (ou index) indiquant sa position dans l'ensemble.
- Les tableaux sont toujours à **bornes statiques** et leur indiçage démarre toujours à partir de **0**.

Exemple :

```
#define MAX 20  // définit l'étiquette MAX egale à 20

int  t[10];     // tableau de 10 éléments entiers (int)

// tableau à 2 dimensions de 2 lignes et 5 colonnes :
int  m[2][5] = { 2, 6, -4, 8, 11,  // initialise avec des valeurs
                 3, -1, 0, 9, 2 };

int  x[5][12][7]; // tableau a 3 dimensions, rarement au-delà de cette dimension

float f[MAX]; // tableau de MAX éléments de type float

t[2] = 6; // accès en écriture au 3eme élément du tableau t
printf("%d", m[1][3]); // affiche la valeur 9

int t[] = {2, 7, 4}; // tableau de 3 éléments (la dimension d'un tableau peut être omise dans ce cas)

char msg[] = "Bonjour"; // chaîne de caractères (la dimension d'un tableau peut être omise dans ce cas mais attention la
    taille est ici de 8 car il y a la valeur nulle qui a été ajoutée pour marquer la fin de chaîne)

int t[5] = {0, 2, 3, 6, 8}; // un tableau de 5 entiers
int *p1 = NULL; // le pointeur est initialisé à NULL (précaution obligatoire)
int *p2; // pointeur non initialisé : il pointe donc sur n'importe quoi (gros danger)

p1 = t;             // p1 pointe sur t c'est-a-dire la première case du tableau
// identique a : p1 = &t[0];

p2 = &t[1];         // p2 pointe sur le 2eme élément du tableau

*p1 = 4;             // la première case du tableau est modifiée
printf("%d ou %d\n", *p1, t[0]); // affiche 4 ou 4

printf("%d ou %d\n", *p2, t[1]); // affiche 2 ou 2
p2 += 2;             // p2 pointe sur le 4eme élément du tableau (indice 3)
printf("%d ou %d\n", *p2, t[3]); // affiche 6 ou 6

// on peut utiliser les [] sur un pointeur :
p1[1] = 8;           // identique à : *(p1+1) = 8; ou a : t1[1] = 8;
printf("%d\n", t[1]); // affiche 8
```

Particularités des tableaux :

- L'identificateur du tableau désigne non pas le tableau dans son ensemble, mais plus précisément l'**adresse en mémoire du début du tableau**.
- Ceci implique qu'il est impossible d'affecter un tableau à un autre : `int a[10], b[10]; a = b; // cette affectation est interdite`
- L'identificateur d'un tableau sera donc "vu" comme un **pointeur constant**.

Le plus grand danger dans la manipulation des tableaux est d'accéder en écriture en dehors du tableau. Cela provoque un accès mémoire interdit qui n'est pas contrôlé au moment de la compilation. Par contre, lors de l'exécution, cela provoquera une exception de violation mémoire (*segmentation fault*) qui se traduit généralement par une sortie prématurée du programme avec un message "Erreur de segmentation".

Les structures (`struct`)

Une structure est un **objet agrégé comprenant un ou plusieurs champs (membres)** que l'on regroupe sous un seul nom afin d'en faciliter la manipulation et le traitement. Chacun des champs peut avoir n'importe quel type, y compris une structure, à l'exception de celle à laquelle il appartient.

Pour **accéder aux champs dune structure**, il faut distinguer 2 cas :

1. la structure est délivrée par une variable : cet accès se fait à l'aide de l'opérateur `.` (point)

   Exemple : `naissanceMarie.mois` désigne le champ `mois` de la variable `naissanceMarie` (soit 11 dans notre exemple).

2. la structure est délivrée par un pointeur p : cet accès se fait à l'aide de l'opérateur `->` (indirection)

   Exemple : `p->jour` désigne le champ `jour` de la variable `p`

   Ou : `(*p).jour` désigne le champ `jour` de la variable `p`

Exemples :

```
// Déclaration d'une structure date :
struct date
{
   int  jour,
        mois,
        annee;
};

// Utilisation de typdef :
typedef struct
{
   int  jour,
        mois,
        annee;
} DATE; // le type DATE

// ou apres :
typedef struct date DATE_NAISSANCE; // le type DATE_NAISSANCE

int main()
{
   struct date naissanceMarie = {7, 11, 1867}; // initialisation de la structure naissanceMarie
   struct date *p_naissanceMarie = &naissanceMarie;
   DATE_NAISSANCE naissancePierre = {15, 5, 1859}; // initialisation de la structure naissancePierre
   DATE mortColuche = {19, 6, 1986};

   printf("Marie Curie est née le %02d/%02d/%4d\n", naissanceMarie.jour, naissanceMarie.mois, naissanceMarie.annee);

   printf("Marie Curie est née le %02d/%02d/%4d\n", p_naissanceMarie->jour, p_naissanceMarie->mois, p_naissanceMarie->annee);


   printf("Pierre Curie est né le %02d/%02d/%4d\n", naissancePierre.jour, naissancePierre.mois, naissancePierre.annee);
```

```
   printf("Coluche est mort le %02d/%02d/%4d\n\n", mortColuche.jour, mortColuche.mois, mortColuche.annee);

   printf("La structure struct date occupe une taille de %d octets\n", sizeof(struct date)); // affiche une taille de 12
        octets

   return 0;
}
```

Il existe aussi les **unions** qui sont conceptuellement identiques aux structures mais peut, à tout moment, contenir n'importe lequel des différents champs. Une **union**, d'un autre côté, définit plusieurs manières de regarder le même emplacement mémoire. A l'exception de ceci, la façon dont sont déclarés et référencées les structures et les unions est identique.

Allocation dynamique

Rappels : La mémoire dans un ordinateur est une **succession d'octets (soit 8 bits)**, organisés les uns à la suite des autres et **directement accessibles par une adresse**.

En C/C++, la mémoire pour stocker des variables est organisée en deux catégories :

1. la pile (*stack*)
2. le tas (*heap*)

Dans la plupart des langages de programmation compilés, la pile (*stack*) est l'endroit où sont stockés les paramètres d'appel et les variables locales des fonctions.

La pile (stack) :

- La pile (*stack*) est un **espace mémoire réservé au stockage des variables désallouées automatiquement**.
- Sa taille est limitée mais on peut la régler (appel POSIX `setrlimit`).
- La pile est bâtie sur le modèle **LIFO** (*Last In First Out*) ce qui signifie "Dernier Entré Premier Sorti". Il faut voir cet espace mémoire comme une pile d'assiettes où on a le droit d'empiler/dépiler qu'une seule assiette à la fois. Par contre on a le droit d'empiler des assiettes de taille différente. Lorsque l'on ajoute des assiettes on les empile par le haut, les unes au dessus des autres. Quand on les "dépile" on le fait en commençant aussi par le haut, soit par la dernière posée. Lorsqu'une valeur est dépilée elle est effacée de la mémoire.

Le tas (heap) :

- Le tas (*heap*) est l'autre **segment de mémoire utilisé lors de l'allocation dynamique** de mémoire durant l'exécution d'un programme informatique.
- Sa taille est souvent considére comme illimitée mais elle est en réalité limitée.
- Les fonctions `malloc` et `free`, ainsi que les opérateurs du langage C++ `new` et `delete` permettent, respectivement, d'allouer et désallouer la mémoire sur le tas.
- La mémoire allouée dans le tas doit être désallouée explicitement.

Les opérateurs `new` et `delete` :

- Pour allouer dynamiquement en C++, on utilisera l'opérateur `new`.
- Celui-ci renvoyant une adresse où est crée la variable en question, il nous faudra un pointeur pour la conserver.
- Manipuler ce pointeur, reviendra à manipuler la variable allouée dynamiquement.
- Pour libérer de la mémoire allouée dynamiquement en C++, on utilisera l'opérateur `delete`.

Exemple : allocation dynamique

```
#include <iostream>
#include <iostream>
#include <new>

using namespace std;

int main ()
{
   int * p1 = new int; // pointeur sur un entier

   *p1 = 1; // ecrit 1 dans la zone mémoire allouée
   cout << *p1 << endl; // lit et affiche le contenu de la zone mémoire allouée

   delete p1; // libère la zone mémoire allouée

   int * p2 = new int[5]; // alloue un tableau de 5 entiers en mémoire

   // initialise le tableau avec des 0 (cf. la fonction memset)
   for(int i=0;i<5;i++)
   {
      *(p2 + i) = 0;  // les 2 écritures sont possibles
      p2[i]     = 0;   // identique à la ligne précèdente
      cout << "p2[" << i << "] = " << p2[i] << endl;
   }

   delete [] p2; // libère la mémoire allouée

   return 0;
}
```

Fuite de mémoire : L'allocation dynamique dans le tas **ne permet pas la désallocation automatique**. Chaque allocation avec "new" doit impérativement être libérée (détruite) avec "delete" sous peine de créer une **fuite de mémoire**. La fuite de mémoire est une zone mémoire qui a été allouée dans le tas par un programme qui a omis de la désallouer avant de se terminer. Cela rend la zone inaccessible à toute application (y compris le système d'exploitation) jusqu'au redémarrage du système. Si ce phénomène se produit trop fréquemment la mémoire se remplit de fuites et le système finit par tomber faute de mémoire. Ce problème est évité en Java en introduisant le mécanisme de "ramasse-miettes" (*Garbage Collector*).

Conversion de type (transtypage)

Conversion "forcée" par la lvalue

– Les opérateurs d'affectation (=, -=, += ...), appliqués à des valeurs de type numérique, provoquent la conversion de leur opérande de droite dans le type de leur opérande de gauche. Cette conversion "forcée" peut être "dégradante" (avec perte).

Exemple : transtypage "forcée" avec perte

```
#include <iostream>

int main()
{
   int x = 5; float y = 1.5; int res;

   res = (x + y); // cela revient à faire int(x + y) ou (int)(x + y) car res est de type int
   std::cout << "res = " << res << "\n"; // Affiche : res = 6
   return 0;
}
```


Conversion automatique

- Soit l'opération suivante : `short int a = 2; a + 2;`
- L'opération `a + 2` revient à faire l'addition entre un `short int` (a) et un `int` (2). Cela est impossible car on ne peut réaliser que des opérations entre **type identique**.
- Une conversion implicite (automatique) sera faite (pouvant donner lieu à un *warning* de la part du compilateur).
- Les conversions d'ajustement de type automatique réalisées suivant la hiérarchie ci-dessous sont réalisées **sans perte** :
  1. `char` → `short int` → `int` → `long` → `float` → `double` → `long double`
  2. `unsigned int` → `unsigned long` → `float` → `double` → `long double`

Conversion forcée ou cast

- Une conversion de type (ou de promotion de type) peut être implicite (automatique) ou **explicite** (c'est-à-dire forcée par le programmeur).
- Lorsqu'elle est explicite, on utilise l'**opérateur de *cast*** : `(float)a` permet de forcer le `short int` a en `float` en C.
- Nouvelle syntaxe en C++ : **type(expression à transtyper)** soit par exemple `float(a)`.
- Les conversions forcées peuvent être des **conversions dégradantes (avec perte)**. Par exemple : `int b = (int)2.5;`
- En effet, le cast `(int)b` donnera `2` : perte de la partie décimale. Cela peut être dangereux (source d'erreur).

Nouveaux opérateurs de transtypage en C++

- `static_cast` : opérateur de transtypage à tout faire. Ne permet pas de supprimer le caractère `const` ou `volatile`.
- `const_cast` : opérateur spécialisé et limité au traitement des caractères `const` et `volatile`
- `dynamic_cast` : opérateur spécialisé et limité au traitement des *downcast*
- `reinterpret_cast` : opérateur spécialisé dans le traitement des conversions de pointeurs peu portables (permet de réinterpréter les données d'un type en un autre type. Aucune vérification de la validité de cette opération n'est faite).

La syntaxe est la suivante : `op_cast<expression type>(expression à transtyper)` où `op` prend l'une des valeurs (`static`, `const`, `dynamic` ou `reinterpret`)

L'opérateur `static_cast`

- C'est l'opérateur de transtypage à tout faire qui remplace dans la plupart des cas l'opérateur hérité du C.
- Toutefois il est limité dans les cas suivants : il ne peut convertir un type constant en type non constant ET il ne peut pas effectuer de promotion (*downcast*).

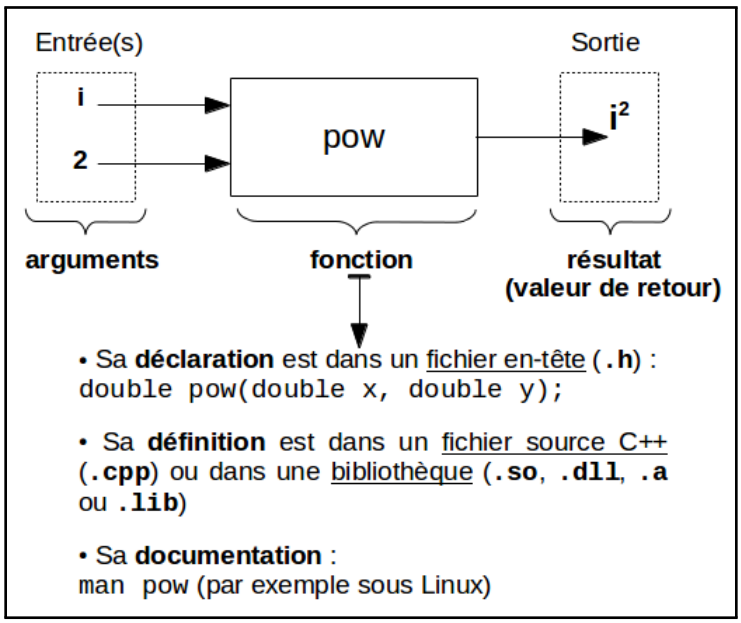
Exemple :

```
int i;
double d;

i = static_cast<int>(d);
```


Les fonctions

Rappels :
– Une fonction réalise un **traitement** en exécutant un ensemble d'instructions.
– Un traitement est tout simplement l'**action** de produire des sorties à partir d'entrées.
– Une fonction produit donc un **résultat en sortie à partir d'un ensemble d'arguments en entrée**.



L'appel `pow(i, 2)`

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

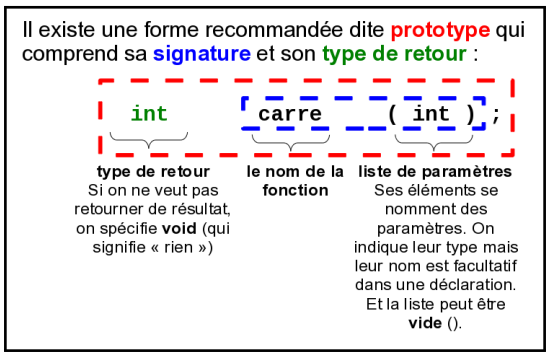
Exemples : `void ajouter()`, `void sauveValeur()`, `estPresent()`, `estVide()`, `viderAMoitieLeReservoir()`, ...

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.

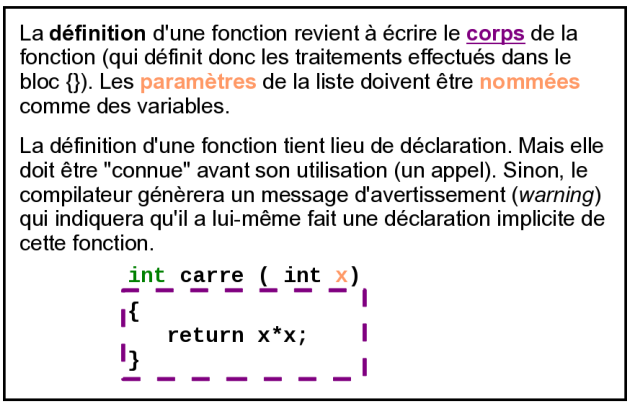
On définit une fonction lorsqu'on souhaite un traitement distinct et nommé parce que procéder ainsi :
– rend le traitement distinct du point de vue logique
– rend le programme plus clair et plus lisible
– permet d'utiliser la fonction à plusieurs endroits dans un programme (à chaque fois qu'on en a besoin)
– 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**. Pour cela, on limitera la taille des fonctions à une valeur comprise entre 10 à 15 lignes maximum.

Déclaration et définition



Déclaration de fonction



Définition de fonction

La déclaration est nécessaire pour la phase de compilation et la définition pour l'édition des liens.

Paramètres par défaut

Le langage C++ offre la possibilité d'**avoir des valeurs par défaut pour les paramètres** d'une fonction (ou d'une méthode), qui peuvent alors être sous-entendus au moment de l'appel.

```
int mult(int a=2, int b=3)
{
   return a*b;
}

mult(3,4); // donne 12
mult(3);   // équivaut à mult(3,3) qui donne 9
mult();    // équivaut à mult(2,3) qui donne 6
```

Surcharge ou surdéfinition

- Il est généralement conseillé d'attribuer des noms distincts à des fonctions différentes.
- Cependant, lorsque des fonctions effectuent la même tâche sur des variables de type différent, il peut être pratique de leur **attribuer des noms identiques**.
- Ce n'est pas possible de réaliser cela en C mais seulement en C++.

– L'utilisation d'un même nom pour des fonctions (ou méthodes) s'appliquant à des types différents est nommée **surcharge** ou surdéfinition.

Cette technique est déjà utilisée dans le langage C++ pour les opérateurs de base. L'addition, par exemple, ne possède qu'une seul nom (+) alors qu'il est possible de l'appliquer à des valeurs entières, virgule flottante, etc ...

```
#include <iostream>

using namespace std;

// Un seul nom au lieu de print_int, print_float, ...
void print(int);
void print(float);

int main (int argc, char **argv)
{
   int n = 2; float x = 2.5;

   print(n); print(x);

   return 0;
}

void print(int a)
{
   cout << "je suis print et j'affiche un int : " << a << endl;
}

void print(float a)
{
   cout << "je suis print et j'affiche un float : " << a << endl;
}
```

On obtient :

```
je suis print et j'affiche un int : 2
je suis print et j'affiche un float : 2.5
```

Signature et Prototype

Remarque importante : la surcharge ne fonctionne que pour des ***signatures différentes*** *et le type de retour d'une fonction ne fait pas partie de la signature.*

On distingue :
– La **signature d'une fonction est le nombre et le type de chacun de ses arguments**.
– Le **prototype d'une fonction est le nombre et le type de ses arguments (signature) et aussi de sa valeur de retour**.

Aller plus loin : il est aussi possible de **surcharger les opérateurs de base** avec des signatures différentes pour ses propres types.

Espace de nom

– En C++, un **espace de nom (*namespace*)** est une notion permettant de lever une ambiguïté sur des termes qui pourraient être homonymes sans cela.
– Il est matérialisé par un préfixe identifiant de manière unique la signification d'un terme. On utilise alors l'**opérateur de résolution de portée** ::.
– Le terme espace de noms (*namespace*) désigne un lieu abstrait conçu pour accueillir (encapsuler) des ensembles de termes (constantes, variables, ...) appartenant à un même domaine. Au sein d'un même espace de noms, il n'y a pas d'homonymes.

La notion d'espace de noms est aussi utilisée en Java, C# et dans les technologies XML.

Utilisation d'un `namespace` :

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

const int UneConstanteGlobale = 1;
int UneVariableGlobale;

namespace MonEspaceDeNom
{
   const int MaConstanteDePorteeNommee = 2;
   int MaVariableDePorteeNommee;
   int UneVariable;
}

int main(int argc, char* argv[])
{
   int UneVariable = 3;
   MonEspaceDeNom::MaVariableDePorteeNommee = UneConstanteGlobale;
   UneVariableGlobale = MonEspaceDeNom::MaConstanteDePorteeNommee;
   MonEspaceDeNom::UneVariable = 4;

   cout << "MonEspaceDeNom::MaVariableDePorteeNommee = " << MonEspaceDeNom::
      MaVariableDePorteeNommee << endl;
   cout << "UneVariableGlobale = " << UneVariableGlobale << endl;
   cout << "UneVariable = " << UneVariable << endl;
   cout << "MonEspaceDeNom::UneVariable = " << MonEspaceDeNom::UneVariable << endl;
   return 0;
}
```

```
MonEspaceDeNom::MaVariableDePorteeNommee = 1
UneVariableGlobale = 2
UneVariable = 3
MonEspaceDeNom::UneVariable = 4
```

