

Les types de base

Introduction au C++ et à la programmation objet

E. Courcelle, P. ELYAKIME

CALMIP UMS 3669, IMFT UMR 5502

Juin 2022

- 1 Expressions et instructions
 - Expressions et instructions
 - Les instructions conditionnelles
 - Les boucles
 - break et continue
 - Ecrivons du code lisible
- 2 Le préprocesseur
 - Les constantes en langage C
 - Compilation conditionnelle
 - Les includes
- 3 Déclarations de variables
 - Les variables simples
 - Les types de base
 - Le type auto (c++11)
 - Les types dérivés
- 4 Les opérateurs

Les expressions

- Expressions et instructions
- Les instructions conditionnelles
- Les boucles
- **break** et **continue**

Les expressions

- Une expression pas trop compliquée :

```
7;
```

- Une expression renvoie toujours une valeur :

```
4 + 3; // Renvoie le 7
```

```
b = 4 + 3; // met 7 dans la variable b et renvoie aussi la valeur 7
```

```
a = (b = 4 + 3); // met 7 dans b et dans a:
```

Instructions simples et complexes

Une instruction peut être simple, dans ce cas **elle se termine par un ;**

```
a = 4 + 3;
```

Ou alors c'est un **bloc** et est délimité par { ou }

```
{  
    a = 4 + 3;  
    b = 4 - 3;  
}
```

Déclarations de variables en C

Elles se font **toujours** en début de bloc ! En C++, on déclare les variables n'importe où.

L'instruction conditionnelle if

- Permet de tester si une condition est vraie ou non

La forme générale:

```
if (expression)  
    instruction1;  
else  
    instruction2;
```

Commentaires

- **en C**: Si expression vaut $\neq 0$ instruction1, si elle vaut 0 instruction2
- **en C++**: Si expression vaut **true** instruction1, si elle vaut **false** instruction2
- instruction1 et instruction2 peut être un bloc
- else est optionnel

Plusieurs instructions conditionnelles

Si on veut tester plusieurs conditions, on peut **imbriquer les if** mais ce n'est pas très lisible:

```
if (expression1)
    instruction1;
else if (expression2)
    instruction2;
else if (expression3)
    instruction3;
else
    instruction4;
```

Le switch

Dans beaucoup de cas, on préférera utiliser l'instruction switch plutôt que des if imbriqués:

```
switch(expression) {  
    case 1: instruction1;  
    case 2: instruction2;  
    case 5: instruction3;  
    default: instruction4;  
}
```

ATTENTION, PIEGE !

Si expression vaut 1 on exécutera instruction1, instruction2, instruction3, puis instruction4 !

Si vous ne voulez pas cela, vous pouvez utiliser break:

```
switch(expression) {  
    case 1: instruction1; break;  
    case 2: instruction2; break;  
    case 5: instruction3; break;  
    default: instruction4;  
}
```

Ne pas en abuser !

L'héritage du C++ permettra de limiter de manière drastique l'utilisation des switches !

L'instruction ?;

C'est un if...else compact, qui peut être utilisé au milieu d'une instruction.
Très pratique, à condition de ne pas en abuser car le code risque d'être illisible !

```
// Ce code ...  
int a;  
if ( b == 1 )  
{  
    a = 3000;  
}  
else  
{  
    a = -10;  
}  
  
// ...est équivalent à celui-ci :  
int A = b==1 ? 3000 : -10;  
  
// Étonnant, non ?
```

Les boucles

Permet de répéter l'exécution d'une série d'instructions à l'intérieur d'un programme. Il existe plusieurs types de boucles en C++ :

- for
- while
- do ... while

for

- Permet de boucler en spécifiant la condition initiale, la condition finale, et incrémentation d'un compteur
- Généralisée en C++ pour balayer un conteneur (par exemple un tableau) avec des itérateur
- La boucle peut ne jamais être exécutée

```
float x = 1;  
for (int i=0; i<n; i++)  
    x = 2 * x;
```

ATTENTION, PIEGE !

La forme ci-dessus fonctionne en C++, pas en C.

En C on écrira plutôt:

```
int i;  
float x = 1;  
for (i=0; i<n; i++)  
    x = 2 * x;
```

for (en C++11)

Syntaxe qui provient de la **STL** : permet d'itérer à travers un **conteneur**. C'est une instruction de haut niveau qui utilise la notion d'**itérateurs**, et qui se révélera très pratique et très lisible:

```
int somme = 0;
vector<int> mon_vecteur = { 1,2,3,4,5};
for ( auto element : mon_vecteur )
{
    somme += element;
}
```

while

- Permet de boucler jusqu'à ce que la **condition soit réalisée**
- La condition est évaluée en **début de boucle**
- La boucle peut ne **jamais être exécutée**

```
while (c!= ' ') {  
    c = getchar();  
    chaine[i++] = c;  
}
```

do... while

- Permet de boucler jusqu'à que la **condition soit réalisée**
- La condition est évaluée en **fin de boucle**
- En conséquence, la boucle est exécutée au moins une fois

```
do {  
    instruction ;  
} while (condition);
```

Sortir avec break

break permet de sortir prématurément d'une boucle.

Le code C suivant compte le nombre de zéros situés au début d'un tableau:

```
int cpt = 0;
for (cpt = 0; cpt < n; cpt++) {
    if (A[cpt]!=0)
        break;
}
```

ATTENTION

- On ne peut sortir avec break que de la boucle la plus interne.
- break est utilisé seulement dans les switches et dans les boucles

Sauter des itérations avec continue

continue permet de sauter des itérations.

Imaginons que le code C suivant fait un traitement coûteux pour toutes les cellules non nulles:

```
int cpt = 0;
for (cpt = 0; cpt < n; cpt++) {
    if (A[cpt]==0)
        continue;
    blabla();
    ... faire un traitement compliqué
}
```


Non au instructions compactées

Puisqu'une instruction renvoie toujours une valeur, n'importe quelle instruction peut servir de condition dans un if ou une boucle.

On peut donc faire plusieurs choses à la fois, par exemple:

- Exécuter une instruction en remplissant une variable
- Utiliser la valeur de la variable pour prendre une décision

Cela conduit rapidement à du code illisible !

```
// Plutôt que d'écrire cela :  
if (a=sin(theta))  
    instruction1;  
else  
    instruction2;  
  
// Ecrivez ce qui suit , un poil plus long mais plus lisible :  
a=sin(theta)  
if (a!=0)  
    instruction1;  
else  
    instruction2;
```

Le préprocesseur

Le préprocesseur est la 1^{ère} étape de la compilation qui procède à des opérations de transformations (rechercher/modifier) dans le code source, comme :

- l'inclusion des fichiers d'entêtes
- La définition/lecture des macros
- La compilation conditionnelle

Les fichiers sources en C++

Un code en C++ est composé des fichiers d'entête **.h** ou **.hpp** et des fichiers de codes sources **.cpp**.

Les fichiers d'entête **.hpp** ou **.h**, **utilisés par le compilateur**, contiennent:

- Des **déclarations** de constantes, de fonctions, de classes
- Le code des fonctions déclarées **inline**
- Des modèles (templates)

Les fichiers **.cpp**, **utilisés par l'éditeur de lien** contiennent:

- Des variables globales
- Les définitions (le code) des fonctions

Inclusion des fichiers d'entête

Tous les fichiers d'entête (les includes) se terminant par `.h` ou `.hpp` sont inclus au fichier principal pendant l'étape de préprocessing.

```
#include <iostream>
#include <biblio/chemin/vers/un/fichier.hpp>
#include "mes_fonctions.hpp"
#include "ma_biblio/mes_fonctions.hpp"
```

Où sont les include ?

Si les includes sont ceux du système (ou de bibliothèques ajoutées), on écrit entre `<` et `>` et le chemin est calculé:

- Soit à partir des répertoires systèmes
- Soit à partir des répertoires donnés par le switch de compilateur `-I`

Si les includes sont ceux de mon application, on écrit entre `" "` et le chemin est calculé à partir des répertoire courant.

Les #include en C++

Par convention, les include système en C++ n'ont pas d'extension:

```
#include <iostream>  
#include <vector>
```

Les includes de la bibliothèque standard du C sont utilisables en C++ en :

- ajoutant la lettre "c" au début
- retirant l'extension

```
#include <math.h>           // en langage C  
#include <cmath>            // la meme chose en langage C++
```

Unicité des headers

C'est au programmeur de faire en sorte qu'un header ne soit pas inclus plusieurs fois ! (hé oui) Cela se fait de la manière suivante:

```
#ifndef MON_MODULE  
#define MON_MODULE  
...  
#endif
```

Les macros

Le préprocesseur du C effectue des recherches/remplacements tout au long du code à partir de l'instruction `#define` et remplace le symbole par sa valeur.
Dans l'exemple ci-dessous, `PI` sera remplacé par `3.14`

```
#define PI 3.14
```

Il permet aussi de définir des fonctions, comme la fonction `MAX` qui sera remplacé par l'instruction qu'il définit dans le code :

```
#define MAX(a, b) a>b?a:b
```

Les constantes en langage C

Il n'est pas possible de déclarer en langage C des constantes, à part des constantes littérales (cf doc.) pendant la phase de compilation.

On utilise donc l'instruction `#define` pour construire une constante préprocesseur

```
#define PI 3.14
```

Depuis C++11

En C++, on peut définir de "vraies" constantes.

A partir de la norme C++11, `#define` peut être remplacé par `constexpr`.

`constexpr` est beaucoup plus riche, et permet notamment d'évaluer une fonction à l'étape de la compilation

```
constexpr int a=56;  
constexpr float pi=3.14;
```


Les constantes Chaines de caractères

Le code suivant définit une chaîne de caractères:

```
"Bonjour tout le monde"
```

ATTENTION ! PIEGE !

Une chaîne de caractères est un tableau qui se termine par 0

Une constante caractère s'écrit 'a', elle contient **un** octet.

Une chaîne de caractères n'ayant qu'**un seul** caractère s'écrit "**a**", elle contient **deux octets** (le a et le 0).

En C++...

En C++, on peut définir de "vraies" chaînes de caractère (string)

Compilation conditionnelle

La compilation conditionnelle permet de compiler ou pas un passage en fonction d'une condition:

```
#ifdef GNULINUX
... Version de code pour gnu/linux
#else
... Version de code pour les autres
#endif
#ifdef WINDOWS
... code ajoute pour Windows
#endif
```

La portée d'une variable

La portée d'une variable est la portion de code depuis laquelle on peut utiliser cette variable.

- Elle commence lorsque la variable est déclarée et se termine à la fin du bloc
- Les blocs internes ont accès à la variable

```
...  
{  
    ...           // Ici , on ne peut pas utiliser A  
    int A = 5;    // Debut de la portee de A  
    ...  
    {  
        ...       // Ici , on peut utiliser A  
    }  
    ...  
}                // A partir d'ici , on ne peut plus utiliser A  
...
```

Variable globale

Une variable globale est définie à l'extérieur de toute fonction, de toute classe, ... et est accessible dans tout le programme :

```
..  
int A;           // variable globale  
  
int main() {  
    int B;       // variable locale a main  
    int C=f1();  
};  
  
int f1() {  
    int C=A;     // pas de pb, A est accessible  
    C += B;      // ERREUR B n'est pas connu ici  
    return C;  
}
```

Différences entre C et C++

- En C, les variables **doivent** être déclarées en début de bloc
- En C++, on peut les déclarer n'importe où

Par exemple : en C++, on peut déclarer des indices de boucle for dans la boucle elle-même:

```
for (int i=0; i<10; i++) {  
    ...  
}
```

La structure d'une déclaration de variables

La forme générale :

[descripteur] [type de base] [déclarateur] [initialisateur]

- un descripteur optionnel : const, extern, virtual, ...
- un type de base obligatoire : int, float, double, ...
- un déclarateur obligatoire : nom choisi par le programmeur, un ou plusieurs
- un initialiseur optionnel

```
char* ctbl [] = {"bleu", "blanc", "rouge"};  
const int A = 2;  
int B;
```

Les types entiers

```
short A = 1;  
int B = 10;  
long B = 100;  
long long C = 1000;  
  
unsigned int = A;
```

Remarques

Le type **short** prend moins d'octets que **int**, qui en prend moins que **long**, qui lui-même en prend moins que **long long**...

Dépend du processeurs 32 ou 64 bits: par exemple pour **int** 4 ou 8 octets.

Il existe des entiers signés (**signed**) et non signé (**unsigned**) qui ne sont **jamais** négatifs.

D'autres types entiers

```
char c = 'a';

enum jour_t {lundi=1, mardi, mercredi, jeudi, vendredi, samedi,
             dimanche};
jour_t jour;
...
if (jour == mercredi) {
    ...
}
```

Les booléens en C

Il n'y a pas de type booléen en C.

Faux s'écrit "entier nul", et vrai s'écrit "entier non nul".

Les booléens en C++

En C++, il existe un type booléen.

Une variable booléenne peut prendre les valeurs true ou false, ce qui conduit à des codes plus lisibles.

Le type énumération en C++

En C, le type **enum** est un sous-ensemble d'entiers (janvier et lundi valent 1, février et mardi 2, etc), on peut donc comparer des jours avec des mois !

```
enum Jour {lundi=1, mardi, mercredi, jeudi, vendredi, samedi,
           dimanche};
enum Mois {janvier=1, fevrier, mars, avril, mai, juin, juillet, aout,
           septembre, octobre, novembre, decembre};
}
```

En C++11, le type **enum class** permet d'exprimer que les jours et les mois sont bien deux objets différents : le code suivant ne compile pas !

```
#include <stdio.h>
enum class Jour {lundi, mardi, mercredi, jeudi, vendredi, samedi,
                 dimanche};
enum class Mois {janvier, fevrier, mars, avril, mai, juin, juillet,
                 aout, septembre, octobre, novembre, decembre};

int main() {
    Jour j = Jour::lundi;
    Mois m = Mois::janvier;
    if (j==m) printf ( "hoho\n");
}
```

Les types float

Il y a quatre types float. Contrairement aux types entiers, leur taille est normalisée, ainsi que la manière de coder mantisse et exposant (norme IEE 754)

- **float**. Réel simple précision, codé sur 32 bits
- **double**. Réel double précision, codé sur 64 bits
- **long double**. Réel quadruple précision, codé sur 128 bits

Le type void

void signifie "vide".

On ne peut pas donner ce type à une variable.

Mais on peut donner ce type à un pointeur, cela signifie
"pointeur vers n'importe quelle variable".

On peut aussi donner void comme type de retour d'une fonction, cela signifie
"fonction qui ne renvoie rien"

```
void* ptr;  
void f(float x);
```

Taille d'une variable d'un type donné

On a vu à propos des entiers que suivant le système la taille d'un entier peut varier.

La fonction **sizeof** permet de savoir quelle place prend:

- une variable
- un type

```
cout << "Taille d'un tres long entier = " << sizeof( long long int );  
  
short s;  
cout << "Taille de s = " << sizeof(s);
```

Des types entiers dont on connaît la taille

On peut avoir besoin dans certains cas d'utiliser des entiers dont on connaisse précisément l'occupation mémoire, par exemple pour des questions de portabilité.

```
#include <stdint.h>
int8_t a;
uint8_t au;

int16_t b;
uint16_t bu;

int32_t c;
uint32_t cu;

int64_t d;
uint64_t du;
```

Le type auto

Depuis **C++11**, il existe le type **auto utilisé avec un initialiseur** et qui donne à la variable initialisée le même type que la variable à droite du signe `=`.

```
int A = 4;  
auto B = A; // B est initialise a partir de A, et a le meme type
```

Types dérivés

- Tableaux
- Pointeurs
- Structures
- Unions
- Toute combinaison de tous ces types

Les tableaux

Une variable ne permet de stocker qu'une seule valeur. Si on veut en stocker plusieurs, on utilise la structure de données appelée **tableau** qui peut être :

- **statique** si la taille est fixée une fois pour toute
- **dynamique** si la taille varie au cours de l'exécution du programme

Syntaxe de déclaration d'un tableau statique

```
type identificateur[taille];  
identificateur[i]; // pour accéder à l'élément i
```

où i est un entier, et vaut 0, ..., taille-1

Remarques

- Un tableau statique est une adresse de base vers une zone de la mémoire pour stocker le tableau (= ensemble de "cases" mémoires)
- Selon le type de ses éléments, une case ne fait pas la même taille

Les tableaux statiques - Exemple

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

int main() {

    int i;
    int const tailleTableau=10;
    int tableau[tailleTableau];

    for (int i=0; i<tailleTableau; i++ ) {
        tableau[i]=i*i;
        cout<<"Le tableau ["<<i<<" contient la valeur "<<tableau[i]<<
        endl;
    }

    return 0;
}
```

Les tableaux

Pointeurs et tableaux

Les notions de tableau et de pointeur sont interchangeables

Les tableaux en C++

En C++, la STL propose le type `vector` ou `array`, bien plus souples à utiliser que les tableaux C

Initialiser un tableau statique

Le tableau **doit être initialisé**, le C ne le fera pas automatiquement

```
int a[5]    = {34,35,36,37,12};  
float x[5]  = {3.14,1.5};    // Les 3 derniers elements sont = 0  
int b[]     = {1,2,3,4};     // b est de dimension 4
```

Passer un tableau en argument à une fonction

Comme le C ne connaît que l'adresse de base, le développeur doit **passer explicitement la taille du tableau**

Deux manières de procéder:

```
// Declaration de la fonction: 1ere maniere
void fonction_1(int tab[],int taille) {
    ...
}

// Declaration de la fonction: 2nde maniere
void fonction_2(int* tab,int taille) {
    ...
}

// Utilisation de ces fonctions: pas de difference !
int main() {
    int t[TAILLE];

    fonction_1(t,TAILLE);
    fonction_2(t,TAILLE);
}
```

Les structures

Un tableau permet de regrouper des données de même type, si nous souhaitons regrouper des données de types différents, nous utiliserons une structure.

```
// Declaration d'une personne
struct personne {
    char nom[20];
    char prenom[20];
    int no_ss;
}

// Initialisation
personne moi = {"Emmanuel", "Courcelle", 1};

// Accéder aux champs
cout << "Bonjour je m'appelle " << moi.prenom << "\n";
```

Les objets en C++

Pour déclarer des objets nous utiliserons des **classes**, qui dérivent directement des structures.

Les opérateurs

- Les opérateurs d'affectations
- Les opérateurs arithmétique
- Les opérateurs d'incrémentation ou de décrémentation
- Les opérateurs de comparaison
- Les opérateurs logiques
- Les opérateurs pour nombres binaires

Affectation

=

Copie une variable sur une autre.

En C++: Peut être coûteux si la variable est grosse, c'est à dire **si la variable est un objet**.

En C: Peut être un peu coûteux également dans le cas de structures.

A = B

ATTENTION, PIEGE !

Si A et B sont des tableaux, B ne sera pas copié sur A !

Simplement l'adresse de base de A sera remplacée par l'adresse de base de B !

Arithmétique

```
+ - * / %  
+= -= *= /= %=
```

- Binaires: + - * / %
- Unaires: += -= *= /= %=
- Incrémentation ++ ou décrémentation --

```
A = B + C;  
  
A = A + 4;  
A += 4;      // Meme chose !
```


Incrémentation, décrémentation

`++i , i++ , --i , i--`

- Définis sur le type pointeur ou entier
- Servent à itérer dans une boucle: compteurs, indices de tableaux, pointeurs
- En C++, ils sont aussi définis sur les itérateurs

```
for (int i=0; i<1000; ++i) {  
    a[i] *= 2;  
}
```

ATTENTION ! PIEGE !

La valeur retournée par ++ ou -- est différente suivant qu'on itère **avant** ou **après** la variable.

```
int i = 0;  
int j = i++; // j contient 0, i contient 1  
  
int i = 0;  
int k = ++i; // k et i contiennent 1
```

Opérateur de comparaison

```
= , !=  
< , <= , > , >=
```

C ou C++

- Ils renvoient 0 ou 1 **en C**
- Ils renvoient false ou true **en C++**
- Ils sont utilisés dans les boucles, dans les structures if, etc.

! , && , ||

- non
- et
- ou

Opérateurs pour nombres binaires

& | ^ << >>

- ET bit à bit
- OU bit à bit
- NON bit à bit
- Décalage à gauche, donc multiplication par deux
- Décalage à droite, donc division par deux

C ou C++

En **C++**, les opérateurs de décalage prennent le plus souvent une autre signification: sortie ou entrée

Les fonctions

- Déclaration et définition d'une fonction
- Récursivité
- Utiliser les fonctions des bibliothèques systèmes
- Fonction main dans un .cpp
- La compilation

Les fonctions - déclaration

La déclaration de l'interface d'une fonction est optionnelle mais très conseillée. On le fait dans les fichiers d'entête **.hpp** (en C++) ou **.h** (en C)

Syntaxe d'une déclaration d'interface

[type de retour] NomFonction (Liste des types des arguments)

Le type de retour peut être **void** si elle ne renvoie rien.

Le fichier hyperbolique.hpp

```
// Une declaration de fonction qui prend un parametre flottant ,  
// renvoie un parametre flottant  
float sinh( float );  
float cosh( float );  
...
```

Les fonctions - définition

La définition d'une fonction s'écrit dans le fichier **.cpp** (en C++) ou **.c** (en C) :

- La déclaration, doit être **la même** que dans le fichier **.h**
- Le corps de la fonction contient le code

Le fichier hyperbolique.cpp

```
// en debut de fichier, on inclue le header
#include "hyperbolique.hpp"

// La definition de sinh
float sinh(float x) {
    result = exp(x) - exp(-x);
    result /= 2;
    return result;
}

// La definition de cosh
float cosh(float x) {
    result = exp(x) + exp(-x);
    result /= 2;
    return result;
}
```

Les fonctions sont récursives

Elles peuvent donc s'appeler elles-mêmes:

```
// Version recursive de la fonction factorielle
int factorielle(unsigned int x) {
    if (x==0)
        return 1;
    else
        return x * factorielle(x - 1);
}
```


Les bibliothèques système

Les include permettent d'utiliser les fonctions définies par le système ou par des éditeurs tiers:

Les bibliothèques système en C

```
#include <stdio.h>
```

Les bibliothèques système en C++

```
#include <iostream>
```

La fonction main dans un .cpp

Tout exécutable doit avoir une fonction main !

Les fonctions appelées:

- doivent être déclarées en incluant les fichiers d'en-têtes
- doivent être résolues à l'édition de liens

Le fichier cosinh2.cpp

```
#include <iostream>
#include "hyperbolique.hpp"

int main() {
    float s = sinh(2.0);
    float t = cosh(2.0);
    std::cout << "sinh=" << s << "   cosh=" << t << '\n';
}
```

Pour compiler...

Compiler et linker tout d'un coup

```
g++ -o cosinh2 hyperbolique.cpp cosinh2.cpp
```

Compiler en plusieurs fois

```
g++ -c -o hyperbolique.o hyperbolique.cpp  
g++ -c -o cosinh2.o cosinh2.cpp  
g++ -o cosinh2 cosinh2.o hyperbolique.o
```

Faire une bibliothèque et utiliser la bibliothèque

```
g++ -c -o hyperbolique.o hyperbolique.cpp  
ar -c libhyper.a hyperbolique.o  
g++ -o cosinh2 cosinh2.cpp -l hyper
```

Les pointeurs

- Les références: un alias
- Les pointeurs
- Le pointeur nul
- Passage de paramètres valeur ou par référence
- Renvoyer une valeur ou une référence

Les références: un alias

```
int A=3;  
int& a=A;  
A = 2 * a;
```

a et **A** ont toujours la même valeur car il s'agit de la même variable.
Elles sont identiques

ATTENTION, Piège !

```
int &a;    // Ne compile pas !
```

Les pointeurs: des variables qui pointent sur d'autres

```
int A=3;  
int* a;  
a = &A;  
A += 1;
```

a contient l'adresse de **A**

***a** est la valeur se trouvant à cette adresse, soit **A**

***a et A sont égales**

ATTENTION, Piège !

```
int *a;    // ca compile, mais risque de plantage !
```

Le pointeur nul: fier de ne pointer sur rien !

Un pointeur qui ne pointe sur rien, **c'est très dangereux**. Le pointeur nul ne pointe sur rien, **mais on peut le savoir**.

ATTENTION, Piège !

Un pointeur doit **toujours** pointer sur un bloc de variables valide, ou alors être **nul**.

Comment ça s'écrit ?

- en C et en C++ avant C++11: `NULL`
- en C++11: `nullptr`

```
int* x=nullptr;  
...  
if (x != nullptr)  
{  
    ... traiter le cas pointeur nul  
} else {  
    ... traiter le cas pointeur valide  
}
```

Attention, terrain miné !

Ne pas confondre...

```
int* x=NULL;      // Declaration d'un pointeur vers un entier
...
y = *x + 2;       // Operateur d'indirection !
```

Ne pas confondre non plus...

```
int& x=A;         // Declaration d'une reference vers un entier
...
int* y = &x;      // Operateur reference !
```


Passage de paramètres par valeur

Par défaut, les paramètres sont passés **par valeur**
Cela signifie que:

- ils sont copiés à l'entrée de la fonction
- **Cela peut être très lourd !!!!**
- Il ne peut pas y avoir d'effets de bord
- Ce sont des paramètres d'entrée seulement

Un exemple pas trop dur

```
void f(int x) {  
    x = 0;  
}  
x = 5;  
f(x);    // il y a toujours 5 dans x
```

Passage de paramètres par référence

On peut faire en sorte de passer les paramètres par **référence**
Cela signifie que:

- Seule leur adresse est copiée à l'entrée de la fonction
- **Or une adresse c'est tout petit !!!!**
- Il peut y avoir des effets de bord
- Ils peuvent être des paramètres d'entrée ou de sortie

Passer un paramètre par pointeur en C

bouh que c'est laid !

A chaque fois qu'on appelle f, il faut se rappeler que le paramètre est passé par référence.

```
void f(int* x) {  
    *x = 0;  
}  
x = 5;  
f(&x);    // maintenant il y a 0 dans x
```

Passage de paramètres par référence

Passer un paramètre par référence en C++

Bien plus joli !

L'information "passée par référence" est codée dans le prototype de la fonction
et c'est tout !

```
void f(int& x) {  
    x = 0;  
}  
x = 5;  
f(x);    // maintenant il y a 0 dans x
```

Passage de paramètres par référence constante

Avoir le beurre et l'argent du beurre, c'est possible **en C++** !

- Passer une variable par référence, **c'est bien pour la performance**
- Spécifier que cette variable est constante, **ça évite les effets de bord**
- Du coup, la variable redevient une **variable d'entrée** uniquement

Supprimer les effets de bord

```
void f(const int& x) {  
    x = 0; // NOOOOOOOOOON x est contant, ne compile pas !  
}
```

Je passe comment alors ? Pointeur, référence, valeur ?

Pour des variables en entrée

- Si la variable est **petite**, le passage par valeur est suffisant
- Si c'est un **gros objet**, préférer le passage par const &

Pour des variables en entrée et en sortie

- A chaque fois qu'on peut, utiliser des **références**
- Quand on n'a pas le choix (c'est-à-dire **jamais**), utiliser des **pointeurs**

Renvoyer une valeur

Une fonction peut renvoyer une valeur.

- Cela peut être très lourd si:
 - vous utilisez un compilateur tout pourri
 - vos objets n'ont pas de constructeur de déplacement (en C++, voir plus loin)
 - Car cela signifie une copie de l'objet pour passer "à l'extérieur" de la fonction
- C'est une rvalue (un truc qu'on ne peut pas mettre à **gauche** de `=`, du coup on le met à droite.)
- Renvoyer une variable ou un objet par valeur est le moyen normal de travailler, **avec des compilateurs modernes**. Les compilateurs actuels implémentent en effet le RVO (Return Value Optimization).

Renvoyer une référence

Une fonction en C++ peut renvoyer une référence.

- Cela signifie qu'on renvoie juste l'adresse d'un objet
- **Or une adresse c'est tout petit !!!!**
- **Attention** à ne pas renvoyer par référence un objet créé dans la fonction !
 - ça ne compilera pas, et si ça compile ça plantera
- **Attention** à ne pas renvoyer par référence un paramètre passé en entrée
 - ça ne sert à rien
 - Si le paramètre est un littéral... boum !
- Une référence est une lvalue (un truc qu'on peut mettre à **gauche** de =)

Renvoyer une lvalue en C++

En C++ (mais pas en C) on peut écrire cela, aussi bizarre que ça paraisse:

```
f(x) = y; // ok (en c++)
```

Renvoyer un pointeur

Une fonction peut renvoyer un pointeur. Un motif de programmation courant:

- Allouer dynamiquement de la mémoire dans le corps de la fonction
- Renvoyer le pointeur contenant l'adresse de base

C'est la seule manière de faire en C.

Mais si vous faites du C++ (et surtout du C++11):

- c'est mal
- c'est moche
- c'est ringard

En C++ moderne:

- On ne fait pas d'allocation dynamique, ou alors on la cache (voir plus loin)
- On renvoie des objets par valeur, **pas** des pointeurs

Pointeur sur une structure

Pointer sur les champs d'une structure (1)

```
struct personne {  
    string nom;  
    string prenom;  
    int age;  
};  
personne *p;  
(*p).nom    = "Dupont";    // Les () sont indispensables !  
(*p).prenom = "Claude";  
(*p).age    = 20;
```

Pointer sur les champs d'une structure (2)

```
struct personne {  
    string nom;  
    string prenom;  
    int age;  
};  
personne *p;  
p->nom      = "Dupont";    // c'est bien plus joli !  
p->prenom   = "Claude";  
p->age      = 20;
```

Les tableaux et pointeurs

- Pointeurs et tableaux
- Initialiser, copier des tableaux
- Allocation et désallocation dynamique de la mémoire en C
- Allocation et désallocation dynamique de la mémoire en C++
- les tableaux de la STL

Pointeurs et tableaux

Les pointeurs permettent de faire du calcul d'adresse:

```
#define TAILLE 100
int tab[TAILLE];
int* p = tab;      // p pointe vers tab[0]
int *q = tab + 1;  // p pointe vers tab[1]
q -= 2;            // HOULALA !!!! q POINTE AVANT LE TABLEAU !!!
q = p + TAILLE - 1; // q pointe vers le dernier element
q = p + TAILLE;    // HOULALA !!!! q POINTE APRES LE TABLEAU !!!
q = p + 5;         // q pointe vers tab[5]
q++;              // q pointe vers tab[6]
cout << q - p;    // 6
```

Initialiser un tableau à 0

Première méthode

```
for (int i=0; i<TAILLE; i++) tab[i] = 0;
```

Seconde méthode

```
int* q = tab + TAILLE;  
for (int* r=tab; r<q; r++) *r = 0;
```

Copier deux tableaux

Première méthode

```
for (int i=0; i<TAILLE; i++) dst[i] = src[i];
```

Seconde méthode

```
int* s = src;  
int* d = dst;  
for (int i=0; i<TAILLE; i++) *d++ = *s++;
```

Les void *

Ce sont des pointeurs vers un type de données non précisé.

- Utiles pour échanger des adresses brutes
- Ne permettent pas de faire du calcul d'adresse !

Allocation dynamique de la mémoire

L'entête `<stdlib.h>` fournit 4 fonctions pour allouer/libérer la mémoire:

- `malloc` pour allouer de la mémoire
- `calloc` pour allouer et initialiser de la mémoire
- `realloc` pour réallouer (plus de) mémoire
- `free` pour rendre la mémoire

ATTENTION !

- `malloc`, `calloc` et `realloc` renvoient un `void *`
- Il faudra le convertir pour l'utiliser !
- Si on utilise `malloc`, Il faudra initialiser le tableau !
- Si l'allocation de mémoire n'a pas fonctionné, renvoie `NULL`: **il faut faire le test !**

malloc

Allocation de mémoire, sans initialisation.

```
size_t dimension = 1000;  
int* tab = (int*) malloc(dimension * sizeof(int));  
if ( tab==NULL ) { ... erreur ... };  
cout << tab[0] << '\n';      // renvoie n'importe quoi
```


calloc

Allocation de mémoire, suivi d'initialisation

ATTENTION, PIEGE !

Ne s'utilise pas tout-à-fait comme malloc !

```
size_t dimension = 1000;  
int* tab = (int*) calloc(dimension, sizeof(int));  
if ( tab==NULL ) { ... erreur ... };  
cout << tab[0] << '\n';    // renvoie 0
```

malloc ou calloc ?

- Dans la vie ordinaire: ni l'un ni l'autre, en C++ on a d'autres solutions.
- Pour un code "HPC" parallèle: préférer malloc car il ne fait pas d'initialisation, donc il permet de tirer partie de la "first touch policy": il suffit d'initialiser la mémoire dans les threads pour que la mémoire allouée soit "bien placée" par rapport aux processeurs

realloc

Nouvelle allocation de mémoire, plus ou moins grande que l'allocation initiale.

ATTENTION

- Il est possible que la nouvelle zone soit à un endroit différent !
- Si la seconde zone est plus grande que la première, pas d'initialisation
- Les données sont recopiées de sorte qu'on garde les données qui étaient déjà en mémoire

```
size_t dimension = 1000;
int* tab = (int*) calloc(dimension, sizeof(int));
if ( tab==NULL ) { ... erreur ... };
...
tab = realloc((void *) tab, 2 * dimension);
if ( tab==NULL ) { ... erreur ... };
cout << tab[0] << '\n';    // tab[0] pareil avant et apres realloc
```

Allocation dynamique de la mémoire en C++

Un tableau dynamique est un tableau dont le nombre de cases peut varier au cours de l'exécution du programme. Il permet d'ajuster la taille du tableau au besoin du programmeur.

Le C++ fournit les opérateurs **new** et **delete** pour allouer et désallouer de la mémoire dynamiquement.

Syntaxe de **new** et **delete**

```
pointeur = new type[taille];  
delete[] pointeur;
```

L'opérateur `new[]` permet d'allouer une zone mémoire pouvant stocker **taille** éléments de type **type**, et retourne l'adresse de cette zone.

Attention

N'oubliez surtout pas les crochets juste après `delete`, sinon le tableau ne sera pas correctement libéré

Tableaux à 2 dimensions - Exemple

```
#include <iostream>
using std::cout;
using std::cin;

int **t;
int nColonnes;
int nLignes;

void Free_Tab();

int main(){

    cout << "Nombre de colonnes : "; cin >> nColonnes;
    cout << "Nombre de lignes : "; cin >> nLignes;

    t = new int* [ nLignes ];
    for (int i=0; i < nLignes; i++)
        t[i] = new int[ nColonnes ];

    for (int i=0; i < nLignes; i++)
        for (int j=0; j < nColonnes; j++)
            t[i][j] = i*j;
```

Tableaux à 2 dimensions - Exemple

```
for (int i=0; i < nLignes; i++) {  
    for (int j=0; j < nColonnes; j++)  
        cout << t[i][j] << " ";  
    cout << std::endl;  
}  
  
Free_Tab();  
system("pause>nul");  
return 0;  
}  
  
void Free_Tab(){  
  
    for (int i=0; i < nLignes; i++)  
        delete [] t[i];  
    delete [] t;  
}
```

Les tableaux de la STL

La STL offre des tableaux à 1 dimension (conteneurs) beaucoup plus pratiques à utiliser et qui gèrent eux même la mémoire :

- vector
- array
- ...

et pour les tableaux à plusieurs dimensions voir avec la librairie **BOOST**