

Programmation Orientée Objet: C++ - Cours 2

M1 IM/MF-MPA - Année 2023-2024

Structures de contrôle

Structures de contrôle

Un programme est un **flux d'instructions** qui est exécuté dans l'ordre.

Pour casser cette linéarité et donner au programme une relative intelligence, les langages de programmation permettent d'effectuer des choix et des boucles.

On utilise très souvent un bloc d'instructions: il s'agit d'un ensemble d'instructions entouré de deux accolades, l'une ouvrante et l'autre fermante.

```
{  
...  
int a = 5;  
/* des commentaires */  
double c = a + 5;  
...  
}
```

Les instructions placées entre les accolades ouvrante '{' et fermante '}' font partie du même bloc d'instructions.

Les conditions - l'instruction if

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

int main()
{
    int a;

    a = 0;
    if (a == 0)
    {
        cout << "a est nul" << endl;
    }
    else
    {
        cout << "a est non nul" << endl;
    }
    return 0;
}
```

Listing 1: exempleif.cpp

L'instruction *if* permet de choisir si une partie du code sera exécutée ou pas.

Son utilisation est fondamentale lors de l'écriture d'un programme.

Sa syntaxe est :
if (expression)
instruction_1
else
instruction_2

Les conditions - l'instruction if

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

int main()
{
    int a;

    a = 0;
    if (a == 0)
    {
        cout << "a est nul" << endl;
    }
    else
    {
        cout << "a est non nul" << endl;
    }
    return 0;
}
```

Listing 1: exempleif.cpp

expression est une expression quelconque avec la convention:
 Différent de 0 → vrai
 Egal à 0 → faux

instruction_1 et **instruction_2**
 sont des instructions quelconques, à savoir:

- une instruction simple (terminée par un point virgule)
- un bloc d'instructions
- une instruction structurée

Les conditions - l'instruction if

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

int main()
{
    int a;

    a = 0;
    if (a == 0)
    {
        cout << "a est nul" << endl;
    }
    else
    {
        cout << "a est non nul" << endl;
    }
    return 0;
}
```

Listing 1: exempleif.cpp

else est un mot-clé du langage permettant d'exécuter le code dans le cas où la condition n'est pas vérifiée.

Son utilisation est facultative.

L'instruction switch - Syntaxe

```
switch (variable)
{
    case constante_1 : [
        instruction_1]
    case constante_2 : [
        instruction_2]
    ...
    case constante_n : [
        instruction_n]
    [default :
        suite_instruction]
}
```

- L'instruction *switch* permet dans certains cas d'éviter une abondance d'instruction *if* imbriquées.
- **variable** est une variable quelconque de type entier, dont la valeur va être testée contre les constantes.
- **constante_1** : expression constante de type entier (*char* est accepté car converti en *int*) (idem pour constante_2, ..., constante_n)
- **instruction_1** : suite d'instructions quelconques (idem pour instruction_2, ..., instruction_n et suite_instruction)

Petite subtilité : Une fois un cas positif trouvé, les instructions suivantes sont exécutées, même si elles appartiennent à un autre cas. Ce peut être pratique, mais pas toujours. Pour éviter cela, on utilisera l'instruction *break* qui stoppe le flot d'exécution.

L'instruction switch - Syntaxe

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

int main()
{
    unsigned int a;
    cout << "Valeur de a ?" << endl;
    cin >> a;
    switch (a)
    {
        case 0 :
            cout << "a est nul" << endl;
            break;
        case 1 :
            cout << "a vaut 1" << endl;
            break;
        default:
            cout << "a est > 1" << endl;
            break;
    }
    return 0;
}
```

Voici un exemple d'utilisation de l'instruction *switch*.

Tout d'abord le programme demande à l'utilisateur d'entrer un nombre (entier non signé) au clavier.

Ensuite, en fonction de la valeur de *a* entrée, un affichage différent est obtenu.

Listing 2: exempleSwitch.cpp

Structures de contrôle - l'instruction for - boucle avec compteur

L'instruction *for* permet de répéter une ou plusieurs instructions avec une syntaxe parfois plus pratique que les boucles *while*.

```
for (expression_declaration; expression_2; expression_3)
    instruction
```

- **expression_declaration** → va permettre d'initialiser le compteur de boucle.
- **expression_2** → une condition sur le compteur pour arrêter la boucle.
- **expression_3** → l'incrémentation du compteur.
- **instruction** → il s'agit d'une instruction simple, d'un bloc, d'une structure de contrôle ...

Structures de contrôle - l'instruction for - boucle avec compteur

```
#include <iostream>

using namespace std;

int main()
{
    for (int i=0; i<10; i++)
    {
        cout << "i = "
            << i << endl;
    }
    return 0;
}
```

Listing 3: exempleFor.cpp

Ce programme, une fois compilé et exécuté, affichera simplement à l'écran les nombres de 0 à 9.
On aurait pu évidemment obtenir ce résultat avec une boucle *while*.

```
g++ exempleFor.cpp
./a.out
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
```

Structures de contrôle - l'instruction do .. .while

```
do  
instruction  
while ( expression );
```

L'instruction *do ... while* permet de répéter une ou plusieurs instructions tant que la condition **expression** est vraie.

A noter que:

- La série d'instructions dans **instruction** est exécutée au moins une fois.
- Il faut s'assurer que **expression** peut devenir fausse (sinon on ne sort jamais de la boucle !!)

Structures de contrôle - l'instruction while

```
while (expression)
    instruction
```

L'instruction *while* permet de répéter une ou plusieurs instructions tant que la condition **expression** est vraie.

A noter que:

- Il faut s'assurer que **expression** peut devenir fausse (sinon on ne sort jamais de la boucle !!)
- L'expression est évaluée avant l'exécution des instructions de **instruction**. Celles-ci ne sont donc pas forcément exécutées.

Structures de contrôle - l'instruction while

```
#include <iostream>
using namespace std;
int main()
{
    int a = 0;

    while (a < 10)
    {
        cout << a << endl;
        a = a + 1;
    }

    do
    {
        cout << a << endl;
        a = a - 1;
    }
    while (a > 0);

    return 0;
}
```

Voici un exemple de l'utilisation de *while* puis de *do...while*.

La première boucle affiche les nombres de 0 à 9 et se termine lorsque *a* vaut 10 (pas d'affichage).

La deuxième boucle affiche *a*, puis le décrémente tant que *a* est supérieur à 0.

Que vaut *a* lorsque la deuxième boucle se termine ?

Listing 4: exempleWhile.cpp

Structures de contrôle - break et continue

Instructions de branchement inconditionnel :

- *break* et *continue* s'utilisent principalement dans des boucles afin de contrôler plus finement le flux d'exécution.
- *break* permet de sortir de la boucle à n'importe quel moment (souvent une condition validée dans la boucle par un *if*)
- *continue* va stopper prématurément le tour de boucle actuel et passer directement au suivant.

Les fonctions

Pour structurer un programme, un des moyens les plus courants est de diviser le code en briques appelées **fonctions**.

Le terme n'est pas strictement équivalent au terme mathématique.

En effet, une fonction permet de renvoyer un résultat, mais pas seulement: elle peut modifier les valeurs de ses arguments, ne rien renvoyer, agir autrement qu'en renvoyant une valeur: affichage, ouverture et écriture dans un fichier, etc.

Les fonctions - syntaxe

```
type_de_retour nom_de_la_fonction(paramètres)
{
    instructions ...
    return ...
}
```

Il est tout d'abord nécessaire de faire la différence entre la déclaration d'une fonction et sa définition.

Dans un premier temps, une fonction peut être déclarée - c'est-à-dire qu'on signifie au compilateur que cette fonction existe et on lui indique les types des arguments de la fonction et son type de retour - et être définie, dans un second temps : c'est-à-dire qu'on définit l'ensemble des instructions qui vont donner un comportement particulier à la fonction.

La syntaxe ci-dessus présente la définition d'une fonction.

Les fonctions - syntaxe

```
type_de_retour nom_de_la_fonction( paramètres )
{
    instructions ...
    return ...
}
```

type_de_retour → Une fonction peut renvoyer une valeur. Le compilateur doit connaître le type de la valeur afin de pouvoir vérifier la cohérence de l'utilisation de cette valeur de retour dans le reste du programme.

Les fonctions - syntaxe

```
type_de_retour nom_de_la_fonction(paramètres)
{
    instructions ...
    return ...
}
```

nom_de_la_fonction → il s'agit du nom de notre fonction. Il doit bien sûr être cohérent avec ce que fait la fonction en question ...

Les fonctions - syntaxe

```
type_de_retour nom_de_la_fonction(paramètres)
{
    instructions ...
    return ...
}
```

paramètres → Il peut s'agir d'un ou de plusieurs paramètres, séparés par des virgules et qui doivent être précédés par leur type respectif.

Les fonctions - syntaxe

```
type_de_retour nom_de_la_fonction( paramètres )
{
    instructions ...
    return ...
}
```

Vient ensuite le corps de la fonction: il s'agit d'un bloc d'instructions.

Il doit donc débuter avec une accolade ouvrante et se terminer avec une accolade fermante.

La fonction se termine lorsqu'une instruction *return* est exécutée.

Il peut y avoir plusieurs instructions *return* en différents points de la fonction (par exemple dans le cas de l'utilisation d'une condition *if ...*)

Les fonctions - syntaxe

```
type_de_retour nom_de_la_fonction(paramètres)
{
    instructions ...
    return ...
}
```

Il existe un type de retour un peu particulier: *void*

Ce type signifie que la fonction ne renvoie rien. C'est par exemple le cas si elle doit uniquement provoquer un affichage.

Dans ce cas, l'instruction *return* seule (sans argument) est autorisée (par exemple pour sortir de la fonction avant d'atteindre la dernière instruction) car elle ne doit pas retourner de valeur.

Les fonctions - Un exemple de fonction

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

unsigned int mySum(unsigned int N)
{
    unsigned int resu = 0;

    for(unsigned int i=0; i<N+1; i++)
        resu += i;
    return resu;
}

int main()
{
    cout << "Somme jusqu'à 5 inclus = "
        << mySum(5) << endl;
    return 0;
}
```

Listing 5: mySum.cpp

Voici un exemple de fonction.

La fonction *mySum* prend en argument un entier *N* non signé et retourne une valeur de type entier non signé calculant la somme des entiers jusqu'à *N* inclus.

On notera que la fonction *main* est bien sûr également une fonction.

```
./mySum.exe
Somme jusqu'à 5 inclus
= 15
```

Les fonctions - Déclaration de fonctions

Avant de pouvoir utiliser une fonction, c'est-à-dire de l'appeler, il est nécessaire que le compilateur "connaisse" la fonction. Il pourra ainsi réaliser les contrôles nécessaires qui pourront donner lieu à des erreurs de compilation le cas échéant.

Ainsi, on prendra soin d'écrire le "prototype" de la fonction. Pour la fonction *my_pow*,

```
double my_pow(double, unsigned int);
```

- Il n'est pas nécessaire de préciser le nom des paramètres dans ce cas.
- La déclaration se termine par un point virgule

Les fonctions - Passage par valeur

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

/*
 Cette fonction doit échanger la valeur des
 deux entiers passés en paramètres */
void my_swap(int, int);

int main()
{
    int a = 2, b = 3;
    cout << "a : " << a << " b : "
        << b << endl;
    my_swap(a, b);
    cout << "a : " << a << " b : "
        << b << endl;
    return 0;
}

void my_swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}
```

Listing 6: mySwapV0.cpp

Quand on exécute ce programme, on remarque qu'il ne fait pas ce qu'on veut.

Les valeurs de a et de b sont les mêmes avant et après l'appel à la fonction *my_swap*.

Pourquoi?

Par défaut en C++, le passage des arguments à une fonction se fait "par valeur": c'est-à-dire que la valeur du paramètre est copiée en mémoire, et une modification sur la copie n'entraîne évidemment pas la modification de l'original.

Les fonctions - Passage par référence

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

/*
    Cette fonction doit échanger la valeur des
    deux entiers passés en paramètres */
void my_swap(int &, int &);

int main()
{
    int a = 2, b= 3;
    cout << "a : " << a << " b : "
        << b << endl;
    my_swap(a, b);
    cout << "a : " << a << " b : "
        << b << endl;
}

void my_swap(int &a, int &b) {
    int tmp = a;
    a = b;
    b = tmp;
}
```

Listing 7: mySwapV1.cpp

Modifions la fonction *my_swap*.

Cette fois-ci, le programme a bien l'effet désiré!

Pourquoi?

La notation 'int &' signifie qu'on ne passe plus un entier par valeur mais par référence. Il n'y a donc plus copie de la valeur. On passe directement la valeur elle-même.

Si les arguments sont modifiés dans la fonction, ils sont donc modifiés après l'appel à la fonction.

Les fonctions - Passage par référence

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

/*
 Cette fonction doit échanger la valeur des
 deux entiers passés en paramètres */
void my_swap(int &, int &);

int main()
{
    my_swap(2, 3);
}

void my_swap(int &a, int &b) {
    int tmp = a;
    a = b;
    b = tmp;
}
```

Listing 8: mySwapV2.cpp

Quand on tente de compiler ce programme, le compilateur termine en erreur.

Pourquoi?

A la lecture du message, on comprend qu'on ne fournit pas à la fonction un paramètre du bon type.

En effet, on ne peut pas modifier la constante 2 ou 3! Heureusement!

```
$ g++ mySwapV2.cpp
mySwapV2.cpp: Dans la fonction 'int main()':
mySwapV2.cpp:12:15: erreur : invalid initialization of
my_swap(2, 3);
^
mySwapV2.cpp:8:6: note : initializing argument 1 of
void my_swap(int &, int &);
```

Les fonctions - Passage par référence

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

/*
 Cette fonction doit échanger la valeur des
 deux entiers passés en paramètres */
void my_swap(int &, int &);

int main()
{
    my_swap(2, 3);
}

void my_swap(int &a, int &b) {
    int tmp = a;
    a = b;
    b = tmp;
}
```

Listing 8: mySwapV2.cpp

La fonction `my_swap` modifie ses paramètres. On ne peut donc évidemment pas l'appeler avec des arguments constants.

Pour lever cette ambiguïté, on considère qu'une fonction qui ne modifie pas ses arguments doit le spécifier dans sa déclaration en ajoutant le mot-clé **const** au type de ses arguments. Sinon, on considère qu'ils sont modifiables.

```
$ g++ mySwapV2.cpp
mySwapV2.cpp: Dans la fonction 'int main()':
mySwapV2.cpp:12:15: erreur : invalid initialization of
my_swap(2, 3);
^
mySwapV2.cpp:8:6: note : initializing argument 1 of
void my_swap(int &, int &);
```

Les fonctions - Mot-clé const

On a déjà vu ce mot-clé.

- Il permet de définir des variables par une seule instruction et qui ne peuvent pas être modifiées dans la suite du programme, elles restent donc constantes tout au long de leur existence.
- L'attribut *const* permet de protéger les variables.
- Une variable *const* est déclarée et définie en même temps.

```
...
const int N=20; // entier constant
N += 2; // erreur de compilation!
....
```

Les fonctions - Variables globales

La portée d'une variable peut varier.

On dit qu'une variable est **globale** lorsque la portée de celle-ci s'étend sur une portion de code ou de programme groupant plusieurs fonctions. On les utilise en général pour définir des constantes qui seront utilisées dans l'ensemble du programme, par exemple si nous devions définir dans une bibliothèque de maths, la valeur PI. Elles sont définies hors de toute fonction, ou dans un fichier header, et sont connues par le compilateur dans le code qui suit cette déclaration.

Leur utilisation est cependant **déconseillée** tant elles peuvent rendre un code compliqué à comprendre et à maintenir.

Nous ne nous attarderons pas sur elles pour l'instant, il faut juste savoir que cela existe.

Les fonctions - Variables locales

Les variables locales sont les variables les plus couramment utilisées dans un programme informatique impératif (de loin !)

Elles sont déclarées dans une fonction et n'existent que dans celle-ci.

Elles disparaissent (=leur espace mémoire est libéré) une fois que la fonction se termine.

L'appel des fonctions et la création des variables locales reposent sur un système LIFO (Last In - First Out) ou de pile.

Lors de l'appel d'une fonction, les valeurs des variables, des paramètres, etc. sont "empilées" en mémoire et "dépilées" lors de la sortie de la fonction.

Le système considère donc que cet espace mémoire est réutilisable pour d'autres usages !!

Les fonctions - Surcharge

- Aussi appelé overloading ou surdéfinition.
- Un même symbole peut posséder plusieurs définitions. On choisit l'une ou l'autre de ces définitions en fonction du contexte.
- On a en fait déjà rencontré des opérateurs qui étaient surchargés. Par exemple, `+` peut être une addition d'entiers ou de flottants en fonction du type de ses opérandes.
- Pour choisir quelle fonction utiliser, le C++ se base sur le type des arguments.

Les fonctions - Surcharge - Un exemple

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

void printMe(int a)
{
    cout << "Hello ! I'm an integer ! : "
        << a << endl;
}

void printMe(double a)
{
    cout << "Hello ! I'm a double ! : "
        << a << endl;
}

int main()
{
    printMe(2);
    printMe(2.0);
    return 0;
}
```

Listing 9: printMe.cpp

La fonction *printMe* est définie deux fois. Son nom et sa valeur de retour ne changent pas.

Le type de son paramètre change.

Lorsque l'on appelle la fonction, le compilateur se base sur le type de l'argument pour choisir quelle fonction il va appeler.

Dans certains cas, le compilateur n'arrive pas à faire un choix. Il se terminera alors en erreur.

```
Hello ! I'm an integer ! : 2
Hello ! I'm a double ! : 2
```

Tableaux & pointeurs

Exemple

```
#include <iostream>

using namespace std;

int main()
{
    int t[10];

    for(int i=0; i<10; i++)
        t[i] = i;

    for(int i=0; i<10; i++)
        cout << "t[" << i << "]": " " << t[i]
        << endl;

    return 0;
}
```

Listing 10: code14.cpp

- La déclaration `int t[10];` réserve en mémoire l'emplacement pour 10 éléments de type entier.
- Dans la première boucle, on initialise chaque élément du tableau, le premier étant conventionnellement numéroté 0.
- Dans la deuxième boucle, on parcourt le tableau pour afficher chaque élément.
- On notera qu'on utilise la notation `[]` pour l'accès à un élément du tableau.

Quelques règles

Il ne faut pas confondre les éléments d'un tableau avec le tableau lui-même.

Ainsi, par exemple, $t[2] = 3$; $t[0]++$; sont des écritures valides.

Mais écrire $t1 = t2$, si $t1$ et $t2$ sont des tableaux, n'est pas possible!

Il n'existe pas, en C++, de mécanisme d'affectation globale pour les tableaux.

Quelques règles

Les indices peuvent prendre la forme de n'importe quelle expression arithmétique d'un type entier.

Par exemple, si n , p , k et j sont de type *int*, il est valide d'écrire:

```
t[n-3], t[3*p-2*k+j%4]
```

Il n'existe pas de mécanisme de contrôle des indices! Il revient au programmeur de ne pas écrire dans des zones mémoire qu'il n'a pas allouées.

Source de nombreux bugs ...

Quelques règles

En C ANSI et en iso C++, la dimension d'un tableau (=son nombre d'éléments) ne peut être qu'une constante, ou une expression constante. Certains compilateurs acceptent néanmoins le contraire en tant qu'extension du langage.

```
const int N = 50;
int t[N]; // Valide quels que soient la norme et le
           compilateur
int n = 50;
int t[n]; // n'est pas valide systématiquement -
           utilisation déconseillée!
```

Tableaux à plusieurs indices

On peut écrire:

```
int t [5][3];
```

pour réserver un tableau de 15 éléments (5×3) de type entier.

On accède alors à un élément en jouant sur les deux indices du tableau.

Le nombre d'indices peut être quelconque en C++. On prendra néanmoins en compte les limitations de la machine, comme la quantité de mémoire à disposition.

Initialisation d'un tableau

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

int main()
{
    int t[6] = {0, 3, 4, 7, 9, 13};

    for (int i=0; i<6; i++)
        cout << t[i] << ";" ;
    cout << endl;

    return 0;
}
```

Listing 11: code16.cpp

Nous avons déjà initialisé des tableaux grâce à des boucles.

On peut en fait les initialiser "en dur" lors de leur déclaration.

On utilisera alors la notation {} comme dans l'exemple ci-contre.

Pointeurs

Un pointeur est une adresse mémoire. C'est un nombre hexadécimal et il se définit à partir d'un type (*int*, *float*, ...).

Exemples de déclarations de pointeurs:

```
int *v; // pointeur sur int
float* q; // pointeur sur float
double* p; // pointeur sur double
```

Pour écrire la déclaration d'un seul pointeur, on peut placer '*' comme on veut: le coller au type, au nom de la variable ou laisser un espace entre les deux.

Mais pour déclarer simultanément deux pointeurs sur double, p1 et p2, on écrira:

```
double *p1, *p2; // p1 et p2 pointeurs sur double
double* q1, q2; //q1 pointeur sur double, q2 double!
```

Pointeurs - Les opérateurs * et &

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

int main()
{
    int *ptr;
    int i = 42;

    ptr = &i;
    cout << "ptr = " << ptr << endl;
    cout << "*ptr = " << *ptr
        << endl;
    return 0;
}
```

Listing 12: code17.cpp

On commence par déclarer une variable *ptr* de type *int **: un pointeur sur entier.

Puis une variable *i* de type entier.

On assigne à *ptr* l'**adresse** en mémoire de la variable *i*, grâce à l'opérateur **&**.

On affiche ensuite *ptr*: une **adresse**, une valeur qui sera affichée en hexadécimal.

Puis on affiche la valeur pointée par *ptr* (la même que la valeur de *i*). On dit que l'on a **déréférencé** le pointeur *ptr*.

```
$ ./a.out
ptr = 0x7ffdde3edb19c
*ptr = 42
```

Pointeurs - Les opérateurs * et &

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

int main()
{
    int *ptr;
    int i = 42;

    ptr = &i;
    cout << "ptr = " << ptr << endl;
    cout << "*ptr = " << *ptr << endl;
    i += 3;
    cout << "i = " << i << ", *ptr = " << *ptr
        << endl;
    *ptr = 122;
    cout << "*ptr = " << *ptr << ", i = " << i
        << endl;
    return 0;
}
```

```
$ ./a.out
ptr = 0x7ffdde3edb19c
*ptr = 42
i = 45, *ptr = 45
*ptr = 122, i = 122
```

Listing 13: code17b.cpp

Pointeurs - Les opérateurs * et &

	adresses	valeurs	variables
m é m o i r e	0x0		
	:		
	0x42		
	0x43	0xffffcbf4	ptr
	:		
	0xffffcbf4		
	0xffffcbf5		
	0xffffcbf6		
	0xffffcbf7		
	:		

Voici une représentation schématisée de l'exemple précédent.

On voit bien que la valeur de la variable *ptr* est l'adresse en mémoire de la variable *i*.

Le type du pointeur est important: il permet de connaître la taille en mémoire de la valeur pointée!

Pour un type entier, il s'agira des 4 octets suivant l'adresse *0xffffcbf4*.

La taille du pointeur lui-même varie en fonction du nombre de bits du système : 16, 32, ou pour les machines actuelles: 64 bits.

Relation tableaux et pointeurs

En C++, l'identificateur d'un tableau (càd son nom, sans indice à sa suite) est considéré comme un pointeur.

Par exemple, lorsqu'on déclare le tableau de 10 entiers

```
int t [10];
```

La notation *t* est équivalente à *&t[0]*, c'est-à-dire à l'adresse de son premier élément.

La notation *&t* est possible aussi (c'est équivalent à *t*).

Pointeurs - Arithmétique des pointeurs

Une adresse est une valeur entière. Il paraît donc raisonnable de pouvoir lui additionner ou lui soustraire un entier, en suivant toutefois des règles particulières.

Que signifie ajouter 1 à un pointeur sur entier ? Est-ce la même chose que pour un pointeur sur char par exemple ?

Non

Ajouter 1 à un pointeur a pour effet de le décaler en mémoire du nombre d'octets correspondant à la taille du type pointé.

En ajoutant (soustrayant) 1 à un pointeur sur *int* (*float*, *double*, *char* ...), on le décale en mémoire de la taille d'un *int* (resp. *float*, *double*, *char* ...).

On appelle ce mécanisme l'arithmétique des pointeurs.

Relation tableaux et pointeurs

On sait maintenant qu'un tableau peut être considéré comme un pointeur.
Plus précisément, il s'agit d'un pointeur constant.

Pour accéder aux éléments d'un tableau, on a donc deux possibilités :

- La notation indicelle : $t[5]$
- La notation pointeur : $*(t+5)$

Attention:

- La priorité des opérateurs est importante : $*(t+5) \neq *t + 5$
- Un nom de tableau est un pointeur constant! On ne peut pas écrire $tab+=1$ ou $tab=tab+1$ ou encore $tab++$ pour parcourir les éléments d'un tableau.

Pointeurs particuliers

- **Le pointeur nul** (noté `NULL`), dont la valeur vaut 0.
Il est utile car il permet de désigner un pointeur ne pointant sur rien.
Evidemment déréférencer le pointeur nul conduit irrémédiablement à une erreur de segmentation.
- **Le pointeur générique `void *`.**
Un pointeur est caractérisé par deux informations: la valeur de l'adresse pointée et la taille du type pointé.
`void *` ne contient que l'adresse. Il permet donc de manipuler n'importe quelle valeur sans souci de la taille du type. C'était un type très utile en C, notamment pour écrire des fonctions génériques valables quel que soit le type des données.

Allocation statique et dynamique

MÉMOIRE	PILE (stack)	main	variables de la fonction main
		fct_1	variables et arguments de la fonction fct_1 appelée dans main
		fct_2	variables et arguments de la fonction fct_2 appelée dans fct_1
	La pile peut grandir en occupant la mémoire libre		
	mémoire libre		
	Le tas peut grandir en occupant la mémoire libre		
TAS (heap)	Le tas offre un espace de mémoire dite d'allocation dynamique. C'est un espace mémoire qui est géré par le programmeur, en faisant appel aux opérateurs d'allocation new pour allouer un espace et delete pour libérer cet espace.		

Ceci est une représentation schématisée de la mémoire occupée par un processus au cours de son exécution.

On connaît déjà la **pile** (ou stack en anglais) qui contient les variables et les tableaux que l'on a déclarés jusqu'à présent.

Le **tas** (ou heap) est une zone de la mémoire qui peut grandir au fil de l'exécution et dont le contenu est géré par le programmeur. Mais, "Un grand pouvoir implique de grandes responsabilités!"

Les opérateurs new et delete

- *new* est un opérateur unaire prenant comme argument un type.
- La syntaxe est: *new montype* où *montype* représente un type quelconque.
- Il renvoie un pointeur de type *montype** dont la valeur est l'adresse de la zone mémoire allouée pour notre donnée de type *montype*. Par exemple:

```
int *ptr = new int;
```

- On peut maintenant utiliser notre pointeur pour accéder à un entier dont on a alloué l'espace mémoire.
- (**Important**) Une autre syntaxe permet d'allouer un espace mémoire contigü pour plusieurs données à la fois. Le pointeur renvoyé, toujours de type *montype**, pointe vers la première valeur allouée.

```
int* ptr2 = new int[10];
```

On peut donc utiliser *new* pour manipuler des tableaux.

Les opérateurs new et delete

- Attention car on ne peut évidemment pas allouer indéfiniment de la mémoire, celle-ci étant finie. Un programme trop gourmand risque de mettre le système entier en danger et bien souvent celui-ci préférera le terminer de manière brutale.
- *delete* est l'opérateur permettant de faire le ménage dans la mémoire en libérant l'espace qui ne sert plus.
- Lorsqu'un pointeur *ptr* a été alloué par *new*, on écrira alors *delete ptr* pour le libérer.
- S'il s'agit d'un tableau, on écrira *delete [] ptr;*

Les opérateurs new et delete

Remarques:

- Des précautions doivent être prises lors de l'utilisation de *delete*.
- *delete* ne doit pas être utilisé pour des pointeurs déjà détruits.
- *delete* ne doit pas être utilisé pour des pointeurs obtenus autrement que par l'utilisation de *new*.
- Une fois un pointeur détruit, on doit évidemment arrêter de l'utiliser.

Exemple

```
#include <iostream>
using namespace std;
double sum(double *val, int n)
{
    double res = 0;
    for (int i=0; i<n; i++)
        res += val[i];
    return res;
}

int main()
{
    int n;
    double *val;
    cout << "nombres de valeurs: ";
    cin >> n;
    val = new double[n];
    for (int i = 0; i<n; i++) {
        cout << i << ":" " ";
        cin >> val[i];
    }
    cout << "Moyenne de ces valeurs: " <<
        sum(val,n)/n << endl;
    delete [] val;
    return 0;
}
```

Listing 14: code18.cpp

Ce programme calcule la moyenne des valeurs que l'utilisateur entre au clavier durant l'exécution. Mais le nombre de ces valeurs varie aussi! Si nous avions alloué un tableau statiquement, sur la pile, comme jusqu'à présent, nous aurions dû entrer une valeur constante pour sa taille qui aurait pu être soit trop grande, soit trop petite.

On notera

- l'utilisation de *delete* qui permet de libérer notre pointeur proprement à la fin de notre programme.
- L'utilisation du pointeur *val* avec la notation indicelle *[]*, au lieu d'utiliser l'arithmétique des pointeurs.

Pointeurs sur fonctions

Lorsqu'un exécutable est chargé en mémoire, ses fonctions le sont évidemment aussi. Par voie de conséquence, elles ont donc une adresse, que C++ permet de pointer.

Si nous avons une fonction dont le prototype est le suivant :

```
int fct(double, double);
```

Un pointeur sur cette fonction sera déclaré de la façon suivante :

```
int (* fct_ptr)(double, double); // le pointeur s'appellera fct_ptr
```

On notera l'utilisation des parenthèses.

En effet, écrire

```
int * fct(double, double) //même chose que int* fct(double, double)
```

ne signifie pas du tout la même chose.

Pointeurs sur fonctions

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

double fct1(double x){
    return x*x;
}
double fct2(double x){
    return 2*x;
}
void apply(double *val, int n, double (*fct)
    )(double)){
    for (int i=0; i<n; i++)
        val[i] = (*fct)(val[i]);
}
void aff_tab(double *val, int n){
    for (int i=0; i<n; i++)
        cout << i << ":" << val[i] << endl;
}
int main()
{
    double t[10] = {1,2,3,4,5,6,7,8,9,10};
    aff_tab(t, 10);
    apply(t, 10, fct1);
    cout << "Apres apply (elevation au carre)
        :" << endl;
    aff_tab(t, 10);
}
```

Listing 15: code19.cpp

On définit deux fonctions ayant même valeur de retour et même type d'argument, *fct1* et *fct2*. La fonction *aff_tab* n'est là que pour aider et affiche un tableau de doubles donné en paramètre.

La nouveauté se situe dans la fonction *apply* qui va appliquer la fonction passée en paramètre sur chaque élément d'un tableau de *n* éléments, à l'aide d'un pointeur.

On notera que, pour appeler la fonction pointée, il faut la déréférencer, toujours à l'aide de l'opérateur ***.

Cela permet d'écrire des fonctions génériques puissantes et se passer de l'écriture de code redondants!

En effet, on aurait pu appliquer des fonctions de la librairie *cmath* comme *cos* ou *sqrt*, sans réécrire pour chaque cas une boucle pour l'appliquer à chaque élément de ce tableau.

Chaînes de caractères en C

```
#include <iostream>

using namespace std;

int main()
{
    char *str = "Hello World";
    char str2[10] = {'C', 'o', 'u', 'c', 'o', 'u', '\0'};
    int i = 0;

    cout << str << endl;
    while(str2[i++]) cout << str2[i-1];
    return 0;
}
```

Listing 16: code20.cpp

```
./code20
Hello World
Coucou
```

Les chaînes de caractères telles qu'elles sont représentées en C sont toujours valables en C++.

Bien que C++ offre d'autres mécanismes de plus haut niveau pour la manipulation de chaînes, nous allons en parler.

- D'une part car c'est un bon exercice sur les pointeurs.
- Pour comprendre ce qu'est une chaîne de caractères.
- Car elles sont encore utilisées en C++.

Chaînes de caractères en C

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

int main()
{
    char *str = "Hello World";
    char str2[10] = {'C', 'o', 'u', 'c', 'o', 'u', '\0'};
    int i = 0;

    cout << str << endl;
    while(str2[i++]) cout << str2[i-1];
    return 0;
}
```

Listing 16: code20.cpp

```
./code20
Hello World
Coucou
```

En C, les chaînes de caractères peuvent être vues comme des tableaux de `char`.

Il y a néanmoins une convention supplémentaire.

Une chaîne de caractères est donc un ensemble d'octets contigus se terminant par le caractère nul noté `\0`, ceci afin de donner une fin à la chaîne.

La notation "`Hello World`" définit donc un pointeur sur caractères vers une zone de la mémoire où est définie la constante "`Hello World`". On récupère cette adresse dans le pointeur `str`.

Chaînes de caractères en C

```
#include <iostream>

using namespace std;

int main()
{
    char *str = "Hello World";
    char str2[10] = {'C', 'o', 'u', 'c', 'o', 'u', '\0'};
    int i = 0;

    cout << str << endl;
    while(str2[i++]) cout << str2[i-1];
    return 0;
}
```

Listing 16: code20.cpp

```
./code20
Hello World
Coucou
```

On peut tout aussi bien définir une chaîne en déclarant un tableau et en l'initialisant avec la notation {} comme dans l'exemple.

Pour les afficher, on utilise *cout*, ou une boucle qui parcourt le tableau tant que le caractère nul n'est pas rencontré.

Ces deux méthodes ne sont pas tout à fait équivalentes. En effet on peut modifier *str2[0]* mais pas *str[0]*.

Chaînes de caractères en C

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

int main(int nb, char *args[])
{
    cout << "Mon programme possède "
        << nb << " arguments" << endl;
    for (int i=0; i<nb; i++)
        cout << args[i] << endl;
    return 0;
}
```

Listing 17: code21.cpp

```
$ ./a.out salut tout le monde
Mon programme possède 5 arguments
./a.out
salut
tout
le
monde
```

On peut passer à un programme des valeurs, lorsqu'on l'appelle sur la ligne de commande, dans le terminal.

Le programme les reçoit comme des arguments de la fonction *main* (dont nous n'avons pas parlé jusqu'ici).

- Le premier argument de la fonction est conventionnellement un entier qui correspond au nombre d'arguments fournis au programme.
- Le deuxième paramètre est un peu plus complexe. Il s'agit d'un tableau de pointeurs sur *char* de taille non définie.

Chaque élément de ce tableau est donc un pointeur vers une chaîne de caractères qui existe quelque part en mémoire. On peut donc le balayer, comme dans la boucle de l'exemple. Chaque élément *args[i]* est donc de type *char** et peut être considéré comme une chaîne de caractères.

Conventionnellement, le premier élément *args[0]* est le nom du programme exécuté.