(masquer detail) (afficher detail)

Introduction au C++

1) Algorithmique et Programmation (Index)

Quelques repères...

algorithme
: suite d'actions claires et concises permettant de définir les étapes de la résolution un problème (dont on envisage généralement l'exécution automatique) (+) Exemple : comment déterminer le nombre de racines d'une équation du 2nd degré ? d'abord calculer le discriminant, puis si le discriminant est négatif, alors il n'y a pas de racine, sinon si le discriminant est nul, il y a une racine double, sinon deux racines. (+) Exemples de problèmes : retourner les cartes d'une couleur d'un jeu face sur la table, classer les cartes de la plus petite à la plus forte. ou classer un jeu de cartes complet par couleurs. ou calculer la distance entre 2 points d'un espace.

algorithmique
: ensemble de méthodes et techniques mises en oeuvre pour construire des algorithmes

notation algorithmique
: langage utilisé pour définir les algorithmes (écrit - pseudo-code - ou schéma - organigramme, arbre programmatique -)

ordinateur
: machine pouvant exécuter de manière automatique, mais programmée, un ensemble d'instructions

mémoire vive (RAM)
: équipement électronique permettant la mémorisation de données dans des cases de taille fixe, accessibles par leur adresse (position absolue - par rapport au début - ou relative - par rapport à une autre case)

processeur
: équipement électronique capable de réaliser des opérations élémentaires : calculs arithmétiques, comparaison, commandes de lecture et d'écriture dans la mémoire vive, etc.

périphériques
: équipements capables d'envoyer et/ou recevoir des flux d'informations de la mémoire vive.

langage machine
: jeu d'instructions élémentaires connues d'un processeur et exécutable par ce dernier pour effectuer des opérations

élémentaires

programme
: suite d'instructions écrites en langage machine et qu'un ordinateur peut interpréter et exécuter

programmation
: activité qui consiste à composer la suite d'instructions qui permettra le traitement automatique de la résolution d'un problème (+) Cela consiste, à partir d'un algorithme, à éditer le code source du futur programme dans un langage particulier, à compiler et puis tester le bon fonctionnement du programme.

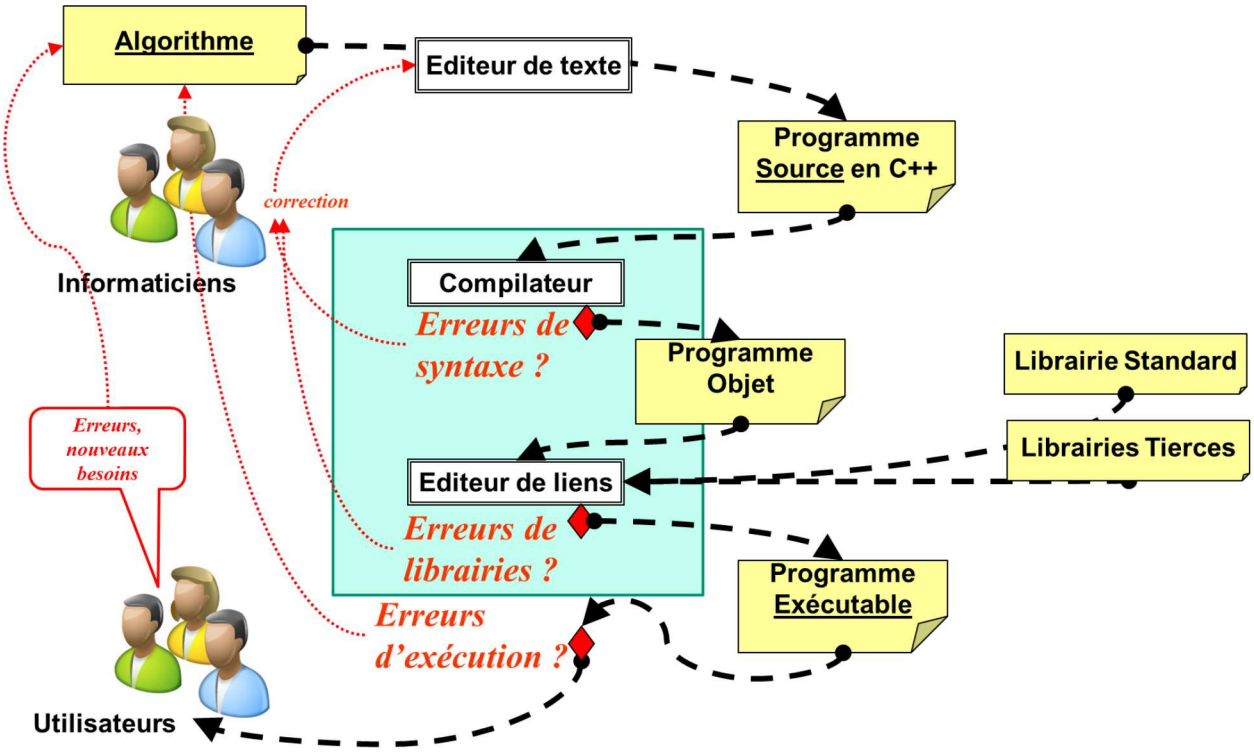
codage binaire
: système de représentation d'une abstraction d'un information élémentaire numérique (symboles 0 et 1) à partir des niveaux de voltage utilisés par les équiments électroniques de l'ordinateur (passage des signaux analogiques à des signaux discrets mesurés à une certaine fréquence de temps, mesurés en terme de niveau de courant électrique : par exemple, inférieur à 1.0V pour représenter 0, supérieur à 2.0V pour représenter 1).

langage de programmation
: langage à mi-chemin entre notation algorithmique et langage machine, qui permet au programmeur la description des tâches que l'ordinateur devra exécuter, mais en utilisant des instructions plus proches des préoccupation du programmeur que des contraintes de l'ordinateur.

programme source
: résultat de traduction d'un algorithme dans un langage de programmation. (+) Il est stocké dans un fichier au format texte brut, dont l'extension est .h et .cpp. (+) L'ASCII (American Standard Code for Information Interchange) : tableau d'encodage des caractères en fonction de la valeur numérique d'un octet. Au départ, codage de 128 caractères - chiffres 0 à 9, lettres a-z et A-Z, caractères de ponctuation, caractères fde contrôle des transmission de données, etc. - , puis extension à 128 supplémentaires en fonction des zones géographiques (Europe de l'ouest, ou de l'est, etc.)

programme exécutable
: résultat de la traduction d'un programme source (exprimé dans un langage de programmation) en un langage exécutable par l'ordinateur, le langage machine. Il est stocké dans un fichier binaire, son extension est .exe (sous Windows).

compilation
: enchainement d'un certain nombre de programmes qui, à partir de codes sources et de codes déjà partiellement compilés, produisent un programme directement exécutable par l'ordinateur (+) La compilation enchaine 3 phases principales, réalisées par les programmes suivants : (a)Le pré-processeur (pre-processor) qui analyse des directives de compilation (ex : #include , etc.) et supprime les commentaires, (b) Le compilateur (compiler) qui assure la traduction du code source en code machine en tenant compte du jeu d'instruction de la famille de processeur, du mode d'adressage (analyse la syntaxe du programme source et produit un programme compilé, mais non exécutable, (c) L'éditeur de liens (linker) qui fusionne le programme avec les librairies externes utilisées et construit le programme exécutable.



Les outils de la programmation

Pour mener à bien la programmation, on doit disposer au minimum des outils suivants :

1. un éditeur de texte (+) un éditeur de texte, logiciel permettant la création et la modification d'un fichier texte (suite de caractères sans mise en forme, non formaté). Un bon éditeur de texte, pour la programmation, doit offrir (a)la coloration syntaxique, coloration des mots-clefs du langage de programmation, (b)la complétion de code : proposition de syntaxe des

instructions en fonction du contexte de saisie. Par exemple : (a) pour Windows : Bloc-notes, NotePad++, PsPad, Gvim, Scite, etc.(b) pour Linux : Gedit, emacs, Vim, etc.
2. un compilateur (+) Compilateur du langage C++ : (a) sous Windows : MinGw (Minimalist GNU for Windows), (b) sous Linux : GCC (GNU Compiler Collection), incluant des possibilités de débogage permettant la recherche et la correction des défauts de conception du programme conduisant à des disfonctionnements ou bogues (en anglais : bugs)

(plus)
Dans le cadre de projets informatiques plus importants, certains outils seront plus adaptés :

- un EDI (Environnement de Développement Intégré, en anglais : IDE, Integrated Development Environment) est un outil intégrant un éditeur de texte et des facilités pour lancer la compilation. Il inclut également des outils permettant la construction d'interfaces graphiques et de nombreuses bibliothèques de procédures et fonctions permettant de produire rapidement des applications. Exemples :
  - Code::Blocks, Dev C++, etc.
  - Eclipse (Open Source), IBM WSAD (commercial)
- un AGL (Atelier de Génie Logiciel, en anglais : outils CASE, Computer Aided Software Engineering) intègre toutes les phases de développement des applications à l'échelle de l'entreprise :
  - Analyse et modélisation, conception des bases de données
  - Un environnement de développement intégré permettant le travail en équipes de programmeur
  - Des outils de tests et de distribution de logiciels.
  - Il utilise souvent un langage propriétaire (spécifique). Exemple : Windev

Algorithmes et notations algorithmiques

L'algorithme est une sequence d'actions qui décrivent pas-à-pas la résolution de problèmes.

algorithmes de la "vie courante"

Par exemple :

- permutter 2 tableaux accrochés côte à côte au mur en tenant compte des contraintes suivantes : on ne peut en prendre qu'un à la fois, on ne peut poser un tableau au sol, un clou planté au mur est disponible
- classer les cartes d'un jeu en ordre croissant de valeur (2-10, valet, dame, roi et as) et dans l'ordre des couleurs : piques, trèfles, carreaux et coeur
- classer les pages d'un tas non classé mais qui portent chacune un numéro de page.

(solution)
Exercice 1:

1. prendre le tableau de gauche et l'accrocher au clou libre
2. prendre le tableau de droite et l'accrocher au clou de gauche
3. prendre le tableau sur le clou libre et l'accrochezr sur le clou droit

Exercice 2:

1. répartir les cartes selon 4 tas, un par couleur
2. classer chaque tas dans l'ordre croissant des valeurs
3. réunir les 4 tas dans l'ordre des couleurs en un nouveau tas unique

Exercice 3:

1. comparer les 2 premieres pages du tas
2. conserver la page avec le plus grand numéro et la remettre sur le tas, et poser l'autre sur un tas "à trier"
3. repeter les étapes 1) et 2) jusqu'à ce qu'il ne reste qu'une feuille sur le tas : elle correspond au numéro de page le plus élévé du tas. La déplacer vers un tas "trié".
4. reprendre le tas "à trier" et répéter les étapes 1) à 3) jusqu'à ce qu'il n'y ait plus de feuilles dans le tas "à trier"

Certains algorithmes vous sont déjà proposés couramment dans des livres de recettes

Algorithmes calculatoires

Les algorithmes informatiques sont du même ordre que les algorithmes de la vie courante, mais les objets manipulés sont immatériels et représentés par des données numériques

De plus, un algorithme calculatoire pourrait être exprimé dans un langage naturel, mais les langages que nous utilisont peuvent être ambigus ou manquent de précision et de rigueur.

```
prendre le rayon,
```

```
calculer la circonference
```

Ainsi des notations plus strictes ont été établies pour écrire un algorithme, parmi lesquelles :

- le pseudo-code : notation écrite
- l'organigramme et l'arbre programmatique : notation graphique

(plus)

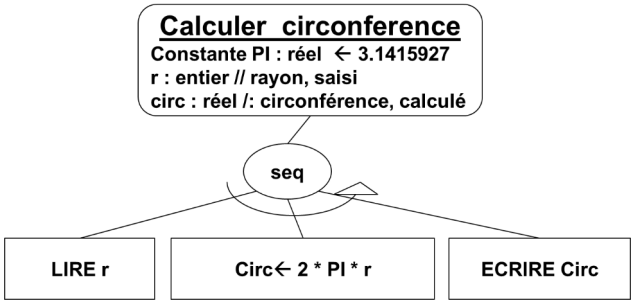
Le pseudo-code

Le pseudo-code est un langage minimaliste, exprimé dans la langue usuelle, avec un nombre de mots réduits, mais ayant un rôle bien précis.

```
Algorithme calculerCirconference
Donnees
  Constante
    PI : reel <-- 3.1415927
  Variables
    r : entier // rayon, saisi
    circ : reel // circonférence, calculée
Debut Traitement
  lire r
  circ <-- 2 * PI * r
  ecrire circ
Fin Traitement
```

L'arbre programmatique

Les représentations graphiques reprennent les éléments du pseudo en leur attribuant un symbole visuel.



Programmes et langages de programmation : pourquoi tant de langages ?

Les langages de programmation sont apparus avec l'ordinateur, dans les années 1950. Les langages de programmation ont été conçus afin de répondre :

- A l'évolution des technologies liées à l'informatique :
  - au départ très proches de l'aspect technique des ordinateurs (0 et 1, le courant passe ou ne passe pas),
  - ensuite plus proche de l'expression en langage naturel et de la réflexion permettant la résolution d'un problème
- A des besoins particuliers (gestion, calcul scientifique, enseignement, systèmes d'exploitation, graphisme, intelligence artificielle, etc.)
- A l'évolution dans la manière d'appréhender les problèmes et de concevoir les programmes (notion de " paradigme de programmation ").

(plus)

Générations de langages

génération	caractéristiques	exemples

L1G, Langage de 1ère Génération	Correspond au langage machine : chaque instruction machine est codée sous forme binaire, une succession de 0 et de 1, sous forme plus compacte en hexadécimal	C'est le langage de base des microprocesseurs composé d'un jeu d'instructions. Voilà par exemple une séquence d'instructions codées en binaire :
		`0010 0100 0001 0000 0000 0000 0000 0011`
		`0010 0100 0001 0001 0000 0000 0000 0101`
		`0000 0010 0001 0001 1000 0000 0010 0000`
		ou pour en "simplifier" l'écriture, codage en hexadécimal :

		0x24100003 **0x24110005** **0x02118020**
L2G : Langages de 2nde génération	à chaque instruction machine, on associe un code mnémonique, plus simple à retenir qu'une suite de 0 et de 1.	Le langage Assembleur (spécifique à une architecture donnée) est plus lisible : **li $s0,3 # charger la valeur 3 dans reg. $s0** **li $s1,5 # charger la valeur 5 dans reg. $s1** **add $s0,$s0,$s1 # additionner $s0 et $s1 , ranger dans $s0** (cf. simulateur MARS
L3G :Langage de 3ème génération	La syntaxe des L3G se rapproche beaucoup plus du langage naturel et du pseudo-code. On parle alors de langage de haut niveau. Des compilateurs assurent la traduction en langage machine.	Cobol, C, Pascal, C++, Java, C#, Visual Basic, Ada, etc. Exemple en Cobol87 : **compute total = total + montant.** Exemple en C90 : **total+=montant;**
L4G : Langages de 4ème génération	La syntaxe est de plus haut niveau encore : quelques (macros)instructions suffisent à construire des programmes complets de manière très rapide. On parle de RAD pour Rapid Application Development.	W-langage, le langage de Windev / ICL Application Master : L4G mainframes ICL)

Paradigmes de programmation

paradigme	**caractéristique**	**Exemples de langages**
PROGRAMMATION IMPERATIVE des instructions manipulent des données constituant l'état du programme (effets de bord) C, VB, PHP, C++, etc.		
Programmation procédurale	On conçoit un programme comme un ensemble de procédures et fonctions qui vont modifier les variables d'état du programme.	C, Cobol, Pascal, Basic Fortran PHP C++, Cobol, Pascal, Java, C#, Visual Basic Bash
Programmation à objets	On associe des variables, des procédures et des fonctions pour former une classe d'objets ; un programme sera constitué d'un ensemble de ces classes. A l'exécution, le programme créé des objets (instanciation) qui s'envoient des messages (appels de méthodes d'instances)	Eiffel, Smalltalk Java Avec possibilité de créer des classes d'objets : C++, Cobol Objet,Pascal Objet, C#, VB.net, PHP5, etc.
PROGRAMMATION FONCTIONNELLE Des fonctions manipulent des ensembles de données constitués par d'autres fonctions (pas d'effet de bord) Scheme, Lisp, XSLT		
PROGRAMMATION LOGIQUE Un programme est constitué d'une suite de règles et de faits, et d'un "moteur d'inférence" va déduire des conclusions à partir de ces règles appliquées aux faits Prolog		

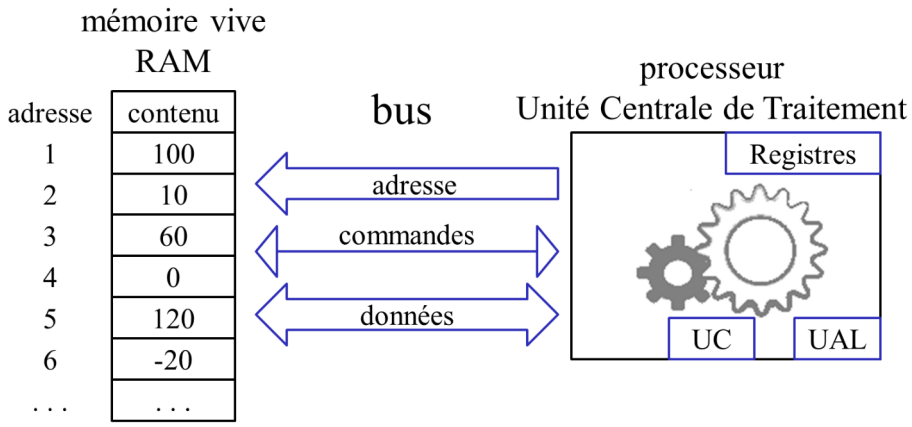
Modes d'exécution des programmes

Mode exéc.	**caractéristique**	**exemples**
interprétés	Les instructions du programme source sont exécutées par un programme interpréteur : celui-ci lit chaque instruction et la traduit en instruction machine (au fur et à mesure) L'exécution est (relativement) lente	Logo PHP Bash Javascript, VBscript
compilés	Le programme source est compilé L'exécution est très rapide	C, C++, Pascal
semi-compilés	Le source est traduit en un code semi-compilé (on parle de bytecode - non directement exécutable par la machine). Un programme particulier, appelé "machine virtuelle" (en anglais : VM, Virtual Machine) se charge de convertir le bytecode en langage machine au moment de l'exécution. Un "ramasse-miettes" (en anglais garbage collector) nettoie la mémoire L'exécution est rapide	Java C#, VB.net, etc.

Architecture des ordinateurs

L'ordinateur est une forme d'automate programmable capable d'exécuter la séquence d'instructions d'un programme. Pour cela interviennent 3 composants principaux :

- la mémoire vive (mémoire centrale, RAM) : c'est la mémoire qui contient les programmes en cours d'exécution : les instructions du programme à exécuter, les données manipulées par ces instructions
- le processeur : coeur de l'ordinateur, il dispose d'un jeu d'instructions de base permettant l'exécution de chaque instruction du programme; une horloge cadence son fonctionnement. Il envoie des commandes au bus pour lire, à partir de la mémoire vive, la prochaine instruction à exécuter et récupérer les données qui y sont associées, et y stocker les résultats de calculs.
- les bus : transportent les informations entre ces 2 composants : commandes issues du processeur (lire une adresse mémoire, écrire une valeur à une adresse mémoire), instructions et données.



D'autres composants interviennent dans l'exécution, parmi lesquels la mémoire de masse matérialisée le plus souvent par un disque dur. Celui-ci stocke de manière permanent programmes installées et données enregistrées sous forme de fichiers.

Les unités de capacité utilisées en informatique sont les suivantes :

Unité de capacité utilisées en informatique

système d'unité classique		**système d'unité informatique (depuis 2008)**	
ko, kilo octet	1000 o, 10^3 octets	kio, kibi octet	1024 o, 2^{10} octets
Mo, Mega octet	1000 000 o, 10^6 octets	Mio, Mebi octet	1 048 576 o, 2^{20} octets
Go, Giga octet	1000 000 000 o, 10^9 octets	Gio, Gibi octet	1 073 741 824 o, 2^{30} octets
To, Tera octet	1000 000 000 000 o, 10^{12} octets	Tio, Tebi octet	1 099 511 627 776 o, 2^{40} octets

Un nouveau système de préfixes a été introduit en 2008 dans le Système International. Le préfixe est appliqué aux octets (aux Bytes en anglais : kB, etc.), mais aussi aux bits (kb, etc.).

La capacité de la mémoire vive est de l'ordre de 4Go, soit environ 4.000.000.000 octets. L'octet est divisé en 8 éléments binaires (BInary digiT, bit). Pour avoir un ordre d'idée :

- un bit permet le stockage d'une seule valeur élémentaire, 0 ou 1.
- un caractère d'un alphabet occupe de 1 à 4 octets (les idéogrammes sont considérés ici)
- un pixel d'une image occupe de 1 à 4 octets (selon la couleur : noir/blanc, 256 couleurs, 65000 couleurs, etc.)

L'espace mémoire vive est réinitialisé à chaque démarrage de l'ordinateur. Il est occupé par de nombreux programmes, parmi lesquels le Système d'Exploitation (Operating System, en anglais - Windows7, Windows8, Linux, Mac OS X,), ensemble de programmes qui assurent le fonctionnement de l'ordinateur et offre des services aux autres logiciels.

Critères qualitatifs des logiciels

Un certain nombre de règles (anglais : rules) d'écriture des programmes vont permettre de construire des applications de qualité. Quelques uns des aspects qualitatifs d'une application :

- La maintenabilité (anglais maintainability) : qualité d'un programme source à être modifié ; qualité du code source à être lisible – lisibilité (anglais : readability) (=utiliser des commentaires, des noms de variables clairs, créer des modules indépendants, etc.)
- La fiabilité (anglais : reliability) : comportement prévisible (= répondre exactement aux besoins de l'utilisateur, prévoir les erreurs et les gérer, etc.)
- La performance (anglais performance) (=optimiser les calculs, etc.)

- La sécurité (anglais : security) : en conflit avec la performance dans la mesure où elle va nécessiter l'ajout de code source supplémentaire, cette qualité est néanmoins fondamentale (=ajouter du code de gestion des erreurs de saisie, etc.)
- La portabilité (anglais : portability) (=écrire sans utiliser des particularités spécifiques à l'ordinateur utilisé, etc.)
- la disponibilité (anglais : availability)

Le programmeur devra avoir en tête ces critères afin de produire des applications de qualité (Cf. CERT C++ Secure Coding Standard , recommandations pour l'écriture de programmes sûrs, issues des CERT (Computer Emergency Response Team) : centres d'alerte et de réaction aux attaques informatiques).

2) Structure générale d'un programme C++ (Index)

Le langage C++

Le langage C a été conçu par deux chercheurs des laboratoires Bell (Brian Kernighan et Dennis Ritchie – K & R) dans les années 1970 (livre de référence "The C programming Language", 1978). Il est normalisé au début des années 1990 : il devient ISO C 90 (appelé aussi ANSI C90), puis des améliorations sont inclues dans la version ISO C 99 (afin d'assurer, autant que possible, la portabilité de C vers C++)

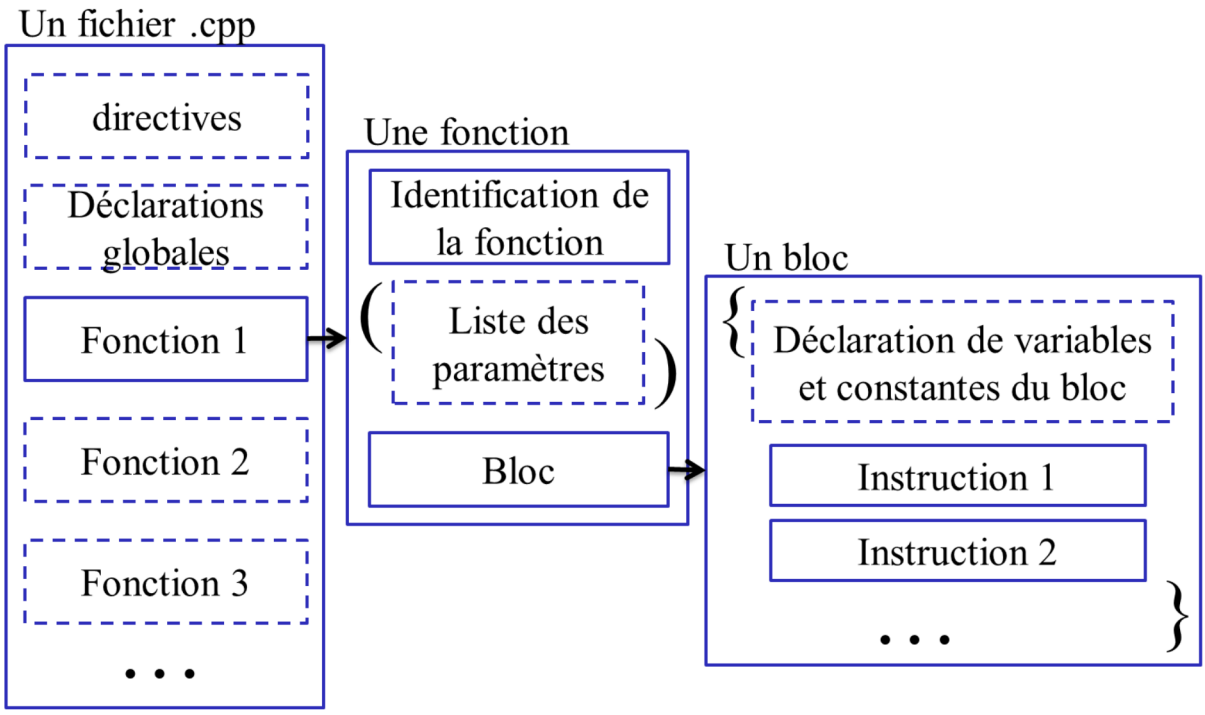
Le C est un langage de haut niveau intégrant des types de données de base, des structures de contrôles et des structures de données. Des bibliothèques donnent accès aux fonctions du système d'exploitation : entrées et sorties (clavier/écran, fichiers, etc.), fonctions mathématiques, etc.

Le langage C++ est une extension du langage C (Bjarne Stroustrup) afin de prendre en compte les évolutions en terme de "paradigmes de programmation" : le paradigme objet est en effet aujourd'hui la référence en conception d'applications. Le langage C++ inclut donc de nouvelles bibliothèques de classes (non compatibles avec le C).

La portabilité du langage C/C++ ("pouvoir compiler un source aussi bien sous Linux que sous Windows", par exemple), est donc toute relative, de nombreux compilateurs incorporant en effet des bibliothèques spécifiques. Il faut donc s'attacher à n'utiliser que les éléments définis dans les standards ou des bibliothèques tierces portables.

C++ a évolué indépendamment de C depuis sa création. Ainsi différentes normes se sont succédées : C++98 (la première) et C++11 (la dernière en date) et la compatibilité ascendante est assurée (un programme C++98 est aussi conforme à la norme C++11, mais pas l'inverse !). De plus, il existe un décalage entre la norme et son implémentation dans les compilateurs : ainsi toutes les améliorations de la version C++11 ne sont pas mises en oeuvre dans tous les compilateurs.

Structure d'un fichier d'un programme C++ :



Exemple d'un fichier d'un programme C++ :

```cpp
// Fichier modele.cpp
// Auteur : Patrick Dezécache
// Date : 2014/02/01
// Objectif : convertir une température de degrés fahrenheit à degrés celsius

#include <iostream>
using namespace std;

int main()
{
    /* DECLARATIONS */
    float tf; // temperature en degrés fahrenheit, saisi
    float tc; // temperature en degrés celsius, calculé, résultat

    /* INITIALISATIONS */
    cout << "Convertisseur fahrenheit -> Celsius :" << endl;
    cout << "entrer la temperature en degrés fahrenheit :";
    cin >> tf;

    /* TRAITEMENTS */
    tc = (5.0 / 9) * (tf - 32);

    /* RESULTATS */
    cout << endl << tf << " degrés fahrenheit -> " << tc << " degrés Celsius" << endl;

    return 0;
}
```

Explications :

- 1-4 : commentaires ligne
- 6 : directive de compilation (pas du C++ : indique au preprocessor de remplacer cette ligne par le contenu du fichier appelé "iostream" qui contient les déclarations de fonctions utiles pour gérer les entrées-sorties)
- 7 : déclaration d'un espace de nom (sera vu plus tard) (+) Un espace de nom est utilisé pour constituer des groupes de classes ayant la même finalité (objects graphiques, objets de gestion, etc.). L'instruction 'using namespace std;' définit que l'espace de nom par défaut sera 'std' (la classe 'string', chaine de caractères, est définie dans l'espace de nom 'std' de C++; pour y l'utiliser, on doit donc écrire 'std::string', mais comme l'espace de nom par défaut est 'std', on peut écrire plus simplement 'string'. Par exemple, pour des classes personnelles :
  `namespace mesClassesGraphiques {classe Cercle {... définition de ma classe Cercle... } }`
- 9 : début de la définition d'une fonction, ici la fonction **main**. Tout projet C++ doit avoir une fonction `main` dans l'un de ses fichiers. C'est le point d'entrée de l'exécution, la première fonction exécutée lors du lancement du programme
- 10 : début du bloc de définition de la fonction `main`
- 11,15,20 et 23 : commentaires bloc (même si, ici, ils tiennent sur une seule ligne). Ces commentaires distinguent les grandes parties d'une fonction afin qu'elle soit plus lisible.
- 12-13 : déclarations des variables. C++ nécessite la déclaration de variable avant toute utilisation.
- 16-17 : instructions de sortie qui permettent d'initier un dialogue avec l'utilisateur
- 18 : instructions d'entrée d'une donnée qui permettra de récupérer la saisie de l'utilisateur afin d'en stocker la valeur dans une variable
- 21 : expression de calcul d'une valeur et affectation à une variable
- 24 : instruction de sortie qui permet de communiquer le résultat à l'utilisateur
- 26 : instruction de sortie de la fonction `main` avec le renvoie d'une valeur (ici 0)
- 27 : fin du bloc de définition d'une fonction

On devrait toujours trouver, dans le corps d'un bloc, les parties ssuivantes :

- déclaration des constantes et variables représentant les objets manipulés,
- initialisation des variables,
- traitement effectif,
- retour du résultat.

Les commentaires en C++

Les commentaires sont des explications textuelles ajoutées au programme source afin de permettre à celui qui le modifiera par la suite de mieux en comprendre le fonctionnement.

```cpp
/* commentaires bloc,
      sur plusieurs
```

```
lignes */

// commentaire sur une ligne

.... // commentaire de fin de ligne
```

Il est souvent utile d'ajouter des commentaires pour apporter des informations supplémentaires au sujet de données utilisées ou de traitements effectués.
Les commentaires participent à la qualite d'un programme (maintenabilité). Il ne faut cependant pas que, en trop grand nombre, ils masquent les éléments importants du programme.

3) Modéliser les objets d'un problème : variables, constantes, type (Index)

Soit l'exemple suivant :

```
a = b + f(5)
```

Pour que cet exemple puisse avoir un sens, il est indispensable de décrire les objets présents : les identifier et définir leur nature - le type de donnée ou domaine de définition - :

```
float a;
int b = 7;
float f(int);
```

soit :

- a est un nombre réel
- b est un nombre entier
- f est une fonction de type réel qui attend un nombre entier

Quelques repères...

identifiant
: nom unique attribué à un objet

variable
: objet dont le contenu est amené à être modifié au cours de l'éxécution d'un programme

constante
: objet dont le contenu est fixe au cours de l'éxécution d'un programme

type
: le type détermine la nature du contenu d'un objet et les opérations qui lui sont applicables.

Les objets sont des emplacements identifiés par un nom et associés à un espace de la mémoire (à une certaine adresse de la RAM) permettant ainsi le stockage de leur contenu.

En C++, l'identifiant doit commencer par une lettre, il peut contenir des lettres, chiffres et le caractère '_'. Il doit être clair mais concis, ne doit pas faire partie des mots réservés du langage C++. (+)
alignas, alignof, and, and_eq, asm, auto, bitand, bitor, bool, break, case, catch, char, char16_t, char32_t, class, compl, const, constexpr, const_cast, continue, decltype, default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, noexcept, not, not_eq, nullptr, operator, or, or_eq, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_assert, static_cast, struct, switch, template, this, thread_local, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while, xor,

Le langage C++ est **sensible à la casse des caractères** (lettres majuscules et minuscules) : 'carnaval' est différent de 'carnavaL'.

Toute donnée doit être déclarée avant d'être utilisée.

Les types de données élémentaires

Nombres entiers

Le langage C/C++ ne définit pas la capacité exacte des nombres entiers, mais chacun des types entier respectent la règle suivant : 8 bits ≤ char/byte ≤ short int ≤ int ≤ long int ≤ long long int.
On peut considérer les limites suivantes en lien avec un compilateur actuel (MinGW sou sWindows) :

type		espace occupé	plage de valeurs
signed (par défaut)	**char**	1 octet	[-128, +127]
	short int	2 octets (au moins 16 bits)	[-32768,+32767]
	int	4 octets (au moins 16 bits, 32 bits généralement)	[-2.147.483.648, +2.147.483.647]
	long int	4 octets(au moins 32 bits)	Idem.
	long long int	8 octets (au moins 64 bits)	[-9.223.372.036.854.775.808, +9.223.372.036.854.775.807]
unsigned	**char**	1 octet	[0, +255]
	short int	2 octets (au moins 16 bits)	[0,+65535]
	int	4 octets (au moins 16 bits, 32 bits généralement)	[0, 4.294.967.295]
	long int	4 octets (au moins 32 bits)	Idem.
	long long int	8 octets (au moins 64 bits)	[0, +18.446.744.073.709.551.615}

Remarque : Les valeurs signées sont codées selon le complément à 2 (complément à 1, plus 1, ce codage rend les opérations arithmétiques plus simples).

L'inclusion du fichier "climits" (#include <climits>) donne accès à des constantes système fournissant les valeurs limites de chacun des types :

- CHAR_MIN, CHAR_MAX et UCHAR_MAX pour le type char
- SHRT_MIN, SHRT_MAX et USHRT_MAX pour le type short (abbrev. de short int)
- Etc.

Exemple de valeurs littérales entières :

- représentation décimale : -15, 2, 63, 83
- représentation octale : 00, 02, 077, 0123 (prefixé par '0' - zéro -)
- représentation hexadécimale : 0x0, 0x2, 0x3f, 0x53 (préfixé par '0x' - zéro x -)

Exemple de valeurs littérales avec suffixes:

- suffixe **u** ou **U** pour définir un littéral **unsigned int** : 3U
- suffixe **l** ou pour définir un littéral de type **long int** : 3L
- suffixe **ll** ou **LL**pour définir un littéral de type **long long int** : 3LL
- suffixe **ull** ou **ULL** pour définir un littéral de type **unsigned long long int** : 3ULL

Nombres réels à virgule flottante

Les nombres sont dits en virgule flottante (signe, mantisse et exposant) et peuvent stocker de très grandes valeurs, avec une certaine précision .

type	espace occupé	Plage de valeurs permises
float	4 octets	32 bits : 1 bit de signe, 8 bits d'exposant, 23 bits de mantisse [1.175494e-038, 3.402823e+038], Simple précision
double	8 octets	64 bits : 1 bit de signe, 11 bits d'exposant, 52 bits de mantisse [2.225074e-308, 1.797693e+308], Double précision
long double	12/16 octets	80/116 bits : 1 bit de signe, 15 bits d'exposant, 64/112 bits de mantisse, Quadruple précision

Remarque : les implémentations peuvent varier en fonction de l'architecture matérielle (x86-x86-64, Sparc, PowerPC, etc.) et de l'implémentation du compilateur sur cette architecture.

Exemple de valeurs littérales réelles :

- type double par défaut : 1.23, .23, 1., 1.2e-10, -1.8e+15
- suffixe **f** ou **F** pour forcer le type **float** : 1.14159265f
- suffixe **l** ou **L** pour forcer le type **long double** : 1.14159265l

Utilisation de ces préfixes : certains calculs ou fonctions sont susceptibles de recevoir des valeurs d'un certain type de donnée. Les préfixes sont un moyen de le préciser pour des valeurs littérales (l'autre possibilité étant le transtypage, cf. plus bas).

Les nombres codées en virgule flottante permettent la realisation de calculs sur des grands nombres, mais avec une precision limitee. L'espace alloué est en effet limitée, mais la conversion en binaire provoque également, dans certains cas, des pertes de précision.
Attention : les valeurs sont approchées (cf. norme IEEE-754)

Caractères

Ce type de donnée est utilisé pour représenter un caractère du jeu de caractères de la machine, répondant à une certaine codification.

La codification ASCII définit une table de correspondance entre la valeur numérique d'un octet et le caractère qui lui est associé. Par exemple une valeur binaire de '0100 0001' (soit 65 en décimal) correspond à la lettre 'A' :

Codes caractères standard (0 - 127)

-	0	-	1	-	2	-	3	-	4	-	5	-	6	-	7	-
0	000	(nul)	016	(dle)	032	sp	048	0	064	@	080	P	096	`	112	p
1	001	(soh)	017	(dc1)	033	!	049	1	065	A	081	Q	097	a	113	q
2	002	(stx)	018	(dc2)	034	"	050	2	066	B	082	R	098	b	114	r
3	003	(etx)	019	(dc3)	035	#	051	3	067	C	083	S	099	c	115	s
4	004	(eot)	020	(dc4)	036	$	052	4	068	D	084	T	100	d	116	t
5	005	(enq)	021	(nak)	037	%	053	5	069	E	085	U	101	e	117	u
6	006	(ack)	022	(syn)	038	&	054	6	070	F	086	V	102	f	118	v
7	007	(bel)	023	(etb)	039	'	055	7	071	G	087	W	103	g	119	w
8	008	(bs)	024	(can)	040	(	056	8	072	H	088	X	104	h	120	x
9	009	(tab)	025	(em)	041	)	057	9	073	I	089	Y	105	i	121	y
A	010	(lf)	026	(eof)	042	*	058	:	074	J	090	Z	106	j	122	z
B	011	(vt)	027	(esc)	043	+	059	;	075	K	091	[	107	k	123	{
C	012	(ff)	028	(fs)	044	,	060	<	076	L	092	\	108	l	124	|
D	013	(cr)	029	(gs)	045	-	061	=	077	M	093	]	109	m	125	}
E	014	(so)	030	(rs)	046	.	062	>	078	N	094	^	110	n	126	~
F	015	(si)	031	(us)	047	/	063	?	079	O	095	_	111	o	127	□

type	espace occupé	Plage de valeurs permises
signed char	1 octets	1+7 bits : lettres, chiffres, caractères de ponctuation, etc.
unsigned char	1 octets	8 bits : lettres, chiffres, caractères de ponctuation, etc.

Exemple de valeurs littérales caractères:

- valeurs littérales char 'a', 'F', '2' , ';'
- int('a') : fournit la valeur numérique du caractère 'a'

Certains caractères spéciaux (préfixés par \, backslash, antislash) sont utilisés pour définir des valeurs particulières :

- '\t' : correspond à une marque de tabulation
- '\n' : correspond à une marque de retour à la ligne

Booléens

Il est utilisé pour représenter le résultat d'une expression logique, soit vrai soit faux.

type	espace occupé	Plage de valeurs permises
bool	1 octets	true (1), false (0)

Les types numériques entiers peuvent également être utilisés comme des booléens (et vice-versa) pour représenter les valeurs `true` ou `false` :

- `true` : valeur numérique équivalente 1
- `false` : valeur numérique équivalente 0
- toute valeur différente de 0 a `true` pour valeur booléenne
- la valeur 0 a `false` pour valeur booléenne

Pointeur

Un pointeur est un type numérique identifié susceptible de contenir l'adresse d'un autre objet stocké dans la mémoire vive (un pointeur faire référence, pointe, un autre objet et son contenu). (+)
La taille d'un pointeur dépend de l'architecture matérielle, du compilateur et elle détermine sa capacité d'adressage. Les pointeurs ont en général une taille de 32 bits (leur permettant l'adressage de 4.294.967.295 emplacements mémoire, 2^{32}).(cf. chapitre relatif aux pointeurs).

Absence de type : void

Le type `void` représente l'absence de type.

Transtypage

Le transtypage correspond à la tentative de changement temporaire du type d'une donnée dans une expression de calcul. Il est réalisé automatiquement lorsqu'il s'agit d'une promotion de type (d'un plus petit vers un plus grand en terme de capacité de stockage), par exemple **short** vers **int** (un **int** peut contenir une valeur de type **short**, mais l'inverse n'est pas vrai).

Il doit être parfois demandé explicitement. Par exemple, la division de 2 entiers donne une valeur entière comme résultat : ici on transtype (cast) dividende, un entier, en **double** afin que le résultat de l'opération conserve la partie décimale :

```
int dividende = 7;
int diviseur  = 3;
double resultat ;
resultat = static cast<double>(dividende) / diviseur ;
```

ou bien à la mode C :

```
int dividende = 7;
int diviseur  = 3;
double resultat ;
resultat = ((double)dividende) / diviseur ;
```

typedef/using : déclaration d'alias de type

L'instruction **typedef** permet de définir un alias sur un type existant. Les déclarations peuvent ensuite déclarer des variables de ce type :

```
typedef type alias;
typedef defType alias;
```

où :

- type : un type de données
- deftype : la définition d'un type de donnée structuré
- alias : l'alias vers ce type de données

Exemple :

```
type unsigned int distance_t;
distance_t d1, d2; // déclaration de 2 variables de ce type
```

(plus)
L'instruction C++ **using** (version C++11) permet, de la même manière, la définition d'un alias sur un type existant.

```
using alias = type;
using alias =  defType;
```

où :

- type : un type de données
- deftype : la définition d'un type de donnée
- alias : l'alias vers ce type de données

Exemple :

```
using distance_t = unsigned int;
distance_t d1, d2; // déclaration de 2 variables de ce type
```

Déclaration des variables

Avant d'utiliser une variable, il est indispensable de la déclarer.

```
type idVar;
type idVar = valInit;
```

où :

- type : un type de données
- idVar : l'identifiant attribué à l'objet (une variable)
- valInit : une valeur initiale pour cet objet

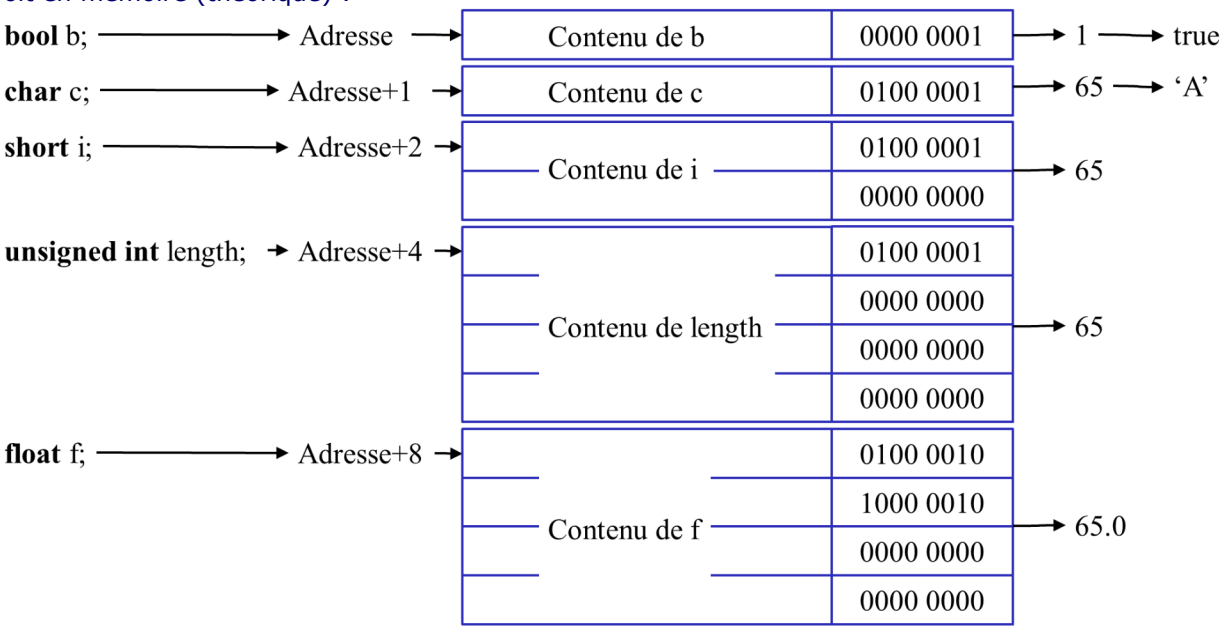
Exemple :

```
bool b;
char c;
short i;
unsigned int length;
float f;
```

Il est possible (et souhaitable) de donner une valeur initiale à toute variable déclarée :

```
bool b = true;
char c = 'a';
short i = 65;
unsigned int length = 65;
float f = 65.0;
```

Soit en mémoire (théorique) :



Un petit programme pour vérifier :

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

int main()
{
    bool b = true;
    char c = 'a';
    short i = 65;
    unsigned int length = 65;
    float f = 65.0;

    cout << "adresse " << "=" << "valeur" << endl;
    cout << &b << "=" << b << endl;
    cout << static_cast<void *>(&c) << "=" << c << endl;
    cout << &i << "=" << i << endl;
    cout << &length << "=" << length << endl;
    cout << &f << "=" << f << endl;

}
```

Soit en mémoire réelle :

```
adresse =valeur
0x28feff=1
0x28fefe=a
0x28fefc=65
0x28fef8=65
0x28fef4=65
```

Il est recommandé de declarer une variable par ligne de code source.

Déclaration des constantes

Les constantes sont des données bien identifiées d'un problème. Ce sont des valeurs littérales qui ont une sens précis dans la résolution du problème, c'est pourquoi il est important de les identifier. Leur valeur restera fixe tout au long de l'exécution des instructions.

```cpp
const type idConst;
const type idConst = init;
```

où :

- const : mot-clef pour 'constante'
- type : un type de données
- idConst : l'identifiant attribué à la constante
- init : valeur fixe affecté à la constante

Par convention, l'identifiant des constantes est en lettres majuscules :

```cpp
const char CONTINUER = 'c';
const unsigned int MAX_LENGTH = +200000;
const double PI = 3.1415927;
```

Il est également possible de définir des constantes 'à la mode C', en utilisant la directive de compilation #define.

```cpp
#define CONTINUER  'c'
#define MAX_LENGTH +200000
#define PI         3.1415927
```

Enumerations

L'énumération associe la définition d'un type de donnée entier à une liste de constantes. Chaque constante prendra une valeur entière à partir de 0.

```cpp
enum idEnum{ NOM1, NOM2, ...};
enum idEnum{ NOM1 = valeur1, NOM2 = valeur2, ...};
```

où :

- idEnum : identifiant de l'énumération
- NOMn : identifiants des constantes constituant l'énumération
- valeurn : valeurs attribuées à ces constantes (par défaut, à partir de 0 et de 1 en 1)

```
enum genre{ MASCULIN = 1, FEMININ = 2};
enum feu{ ROUGE, ORANGE, VERT};
enum booleen{ FAUX = 0, VRAI = 1};
enum carte{ CARREAU = 1, COEUR = 2, TREFLE = 3, PIQUE = 4};
// déclaration d'une variable de ce type
carte uneCarte = COEUR;
feu unFeu = ROUGE;
```

A ces noms de constantes, on affecte un nombre entier qui correspond généralement (sauf indication d'une valeur particulière) à la position de la constante dans la liste : ainsi ROUGE a pour valeur 0, ORANGE a pour valeur 1 et VERT a pour valeur 2.

Type de données chaine de caractères : classe string

C++ définit un type de données étendu (une classe) **string** représentant une classe d'objets chaines de caractères (nécessite l'insertion de l'entête <string>).

```
string ch1; // chaine vide
string ch2("un texte");
string ch3(10,'-'); // contient 10 -
string ch4(ch2,3,5); // contient "texte"
string ch5(ch2,3,5); // contient "texte"
// affectation d'une nouvelle valeur :
ch1 = "ceci est une chaine de caractères";
```

Les chaines de caractères peuvent également être représentées à l'aide de tableaux de caractères (comme en C).

4) Expressions calculées et affectation (Index)

quelques repères...

expression de calcul
: calcul associant des valeurs (littéraux, constantes, variables et autres expression) et des opérateurs. L'évaluation d'une expression fournit un résultat unique d'un type déterminé en fonction des valeurs et des opérateurs utilisés.

Utilisez des parentheses pour definir clairement la priorite des opérations à effectuer dans une expression de calcul.

Les valeurs resultant de l'evaluation d'une expression sont perdues, a moins d'etre conservees : c'est l'objet de l'opération d'affectation.

Affectation

L'affectation est l'opération qui consiste à définir ou à modifier la valeur d'une variable (le contenu stocké à son adresse). La nouvelle valeur doit correspondre au type de données de la variable. L'ancienne valeur est replacée par la nouvelle. L'opérateur d'affectation est '='.

```
lvalue = rvalue;
```

où

- lvalue (left value, partie gauche): l'identifiant d'une variable,
- rvalue (right valeur, partie droite) : l'expression de la nouvelle valeur (littéral, variable, expression, fonction, etc.)

Exemple :

```
int i, j;
i = 0;
i = 1;

i = j = 2;
```

L'opérateur d'affectation renvoie la valeur qui a été affectée, d'où la possibilité d'affectations en cascade.

Expression numériques, calculs

Les expression de calcul numériques fournissent un résultat de type numérique.

Calculs et opérateurs arithmétiques

```
(operande operateur operande)
```

où

- operateur : l"un des opérateurs arithmétiques
- operande : valeur utilisée dans le calcul (littéral, variable, autre expression, fonction, etc.)

Le langage C++ supporte les opérateurs arithmétiques usuels :

- l'addition et la soustraction : +, -
- la multiplication et la division : *, /
- le modulo (reste de la division entière) : %
- l' utilisation des parenthèses pour définir les calculs intermédiaires : ()

opérateur	description	exemple
+	a plus b	(a + b)
-	a moins b	(a - b)
*	a multiplié par b	(a * b)
/	a divisé par b	(a / b)
%	a modulo b	(a % b)

Exemple d'expressions de calcul numérique :

```
(2 + 12);
(i * 2);
(i + 5) * 2);
(i / 3);
```

Ces expressions sont calculées, mais leur résultat est perdu ! Il est souvent nécessaire de conserver ces résultats intermédiaires en utilisant l'opération d'affectation :

```
int i;
i = (2 + 12);
i = (i * 2);
i = (i + 5) * 2;
i = (i / 3);
```

La division de nombres entiers donne comme résultat un quotient entier. Le reste peut être obtenu grâce à l'opérateur modulo : %

```
int i = 2;
int j = 5;
int q = 0;
int r = 0;

q = j / i; // q vaut 2
r = j % i; // r vaut 1
```

Affectation composée

Les opérateurs d'affectation composée combine un opérateur arithmétique et une affectation :

opérateur	exemple	équivalent à
+=	a+=b;	a=a+b;
-=	a-=b;	a=a-b;
*=	a*=b;	a=a*b;
/=	a/=b;	a=a/b;

Incrémentation/Décrémentation

L'incrémentation consiste à ajouter 1 à la valeur d'une variable. La décrémentation consiste à soustraire 1 à la valeur d'une variable. Cette opération peut être réalisée avant utilisation (pré-incrémentation/décrémentation) ou après utilisation (post-incrémentation/décrémentation) de la variable

opérateur	exemple	équivalent à
++	a++	utilise la valeur de a puis l'incrémente
	++a	incrémente a puis utilise sa valeur
--	a--	utilise la valeur de a puis la décrémente
	--a	décrémente a puis utilise sa valeur

Priorité des opérateurs

Certains opérateurs de calculs d'appliquent en priorité par rapport à d'autres. L'usage des parenthèse dans les calculs permet généralement de faire abstraction des priorités. Mais cela n'est pas toujours possible. http://en.cppreference.com/w/cpp/language/operator_precedence

Expressions logiques, tests, conditions

Les expressions de calcul logiques fournissent un résultat de type booléen.

Opérateurs de comparaison (ou opérateurs relationnels)

```
(operande operateur operande)
```

où

- operateur : l"un des opérateurs relationnels
- operande : valeur utilisée dans le calcul (littéral, variable, expression, fonction, etc.)

C++ utilise les opérateurs de comparaison (ou opérateurs relationnels (?) les opérateurs relationnels mettent en relation 2 valeurs pour les comparer.) suivants :

opérateur	description	exemple
==	a est égal à b ?	(a == b)
!=	a est différent de b ?	(a != b)
<	a est plus petit que b ?	(a < b)
<=	a est plus petit ou égal à b ?	(a <= b)
>	a est plus grand que b ?	(a > b)
>=	a est plus grand ou égal à b ?	(a >= b)

```
int i = 2;
int j = 5;
bool t = false;

t = (i < j); // t vaut true
```

eviter d'utiliser le caracteres associatif des operateurs de comparaison (une comparaison renvoie en effet true ou false, soit 1 ou 0, ce qui fausserait les autres comparaisons).
Exemple à rejeter :

```
int a = 2;
```

```
int b = 2;
int c = 2;
bool test;
test = ( a < b < c ) ;// test vaut VRAI, FAUX est plutôt attendu
test =  ( a == b == c ) ;// test vaut FAUX, VRAI est attendu
```

Exemple à utiliser :

```
int a = 2;
int b = 2;
int c = 2;
bool test;
test = ( a < b && b < c )  ;  //  FAUX : OK
test = ( a == b && b == c )  ; //  VRAI : OK
```

Opérateurs logiques

C++ utilise les opérateurs logiques suivants :

opérateur	description	exemple
&&	condition1 ET condition2 ?	((a == b) && (b == c))
||	condition1 OU condition2 ?	((a == b) || (b == c))
!	non condition1 ?	!(a < b)

```
int age;
int note;
bool resultat = false;
...
resultat = ((age >= 18) && (note > 10))
   // VRAI si age >= 18 ET note > 10
resultat = ((age >= 18) || (note > 10))
   // VRAI si age >= 18 OU note > 10
```

Précisions : Les conditions sont évaluées de la gauche vers la droite

- dans le cas ET : si la première des conditions n'est pas remplie, les suivantes ne sont pas évaluées
- dans le cas OU : si la première des conditions est vérifiée, les suivantes ne sont pas évaluées

Les conditions ne doivent donc pas contenir d'opérateurs qui pourraient affecter des variables (comme les opérateurs d'incrémentation).

Priorité des opérateurs

Certains opérateurs de calculs d'appliquent en priorité par rapport à d'autres. L'usage des parenthèse dans les calculs permet généralement de faire abstraction des priorités. Mais cela n'est pas toujours possible. http://en.cppreference.com/w/cpp/language/operator_precedence

Opérateur sizeof

L'opérateur sizeof permet de connaître l'espace mémoire occupé par une variable d'un certain type de donnée (en nombre d'octets).

```
sizeof(type);
sizeof(expression);
sizeof idVar;
```

où

- type : type de donnée de base ou type structuré
- expression : une expression
- idVar : une variable d'un certain type

Exemple :

```
char c;
size_t taille;
taille = sizeof c;
taille = sizeof(int);
taille = sizeof(2*4);
```

Le type **size_t** est une redéfinition de **unsigned int**.

Opérateurs de référencement et de déréférencement

L'opérateur de référencement & permet d'obtenir l'adresse mémoire d'un objet (en général d'une variable).

```
&idVar
```

où

- idVar : variable dont on souhaite récupérer l'adresse mémoire

Exemple :

```
int i;
int * ptri = &i;
```

Il est impératif d'initialiser les pointeurs

L'opérateur de déréférentement (ou d'indirection) * donne accès au contenu situé à une adresse mémoire pointée.

```
*idPtr
```

où

- idPtr : identifiant d'une variable pointeur

Exemple :

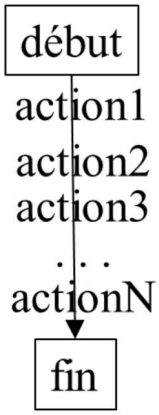
```
*ptri = 2;
*(&i) = 2;
```

soit : le contenu à l'adresse mémoire pointée par ptr1, ou le contenu à l'adresse de i, prend la valeur 2.

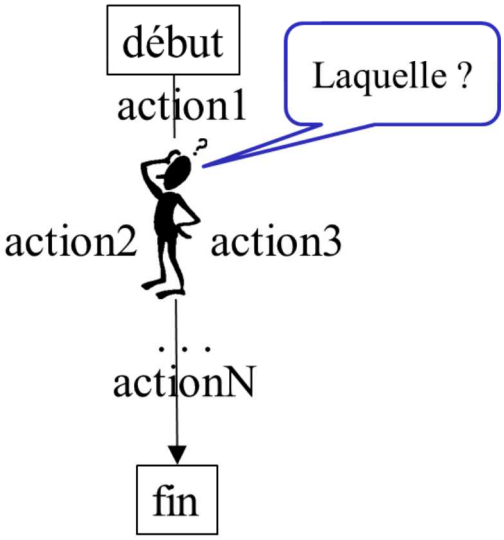
5) Execution conditionnelle (Index)

Une structure conditionnelle offre une rupture dans l'exécution en séquence des instructions d'un programme. Elle permet, si une condition est vérifiée, l'exécution d'un bloc d'instructions; dans le cas contraire, l'exécution d'un bloc alternatif peut être proposé. La condition est exprimée sous forme d'une expression logique.

de la séquence :



au choix :



Exécution conditionnelle

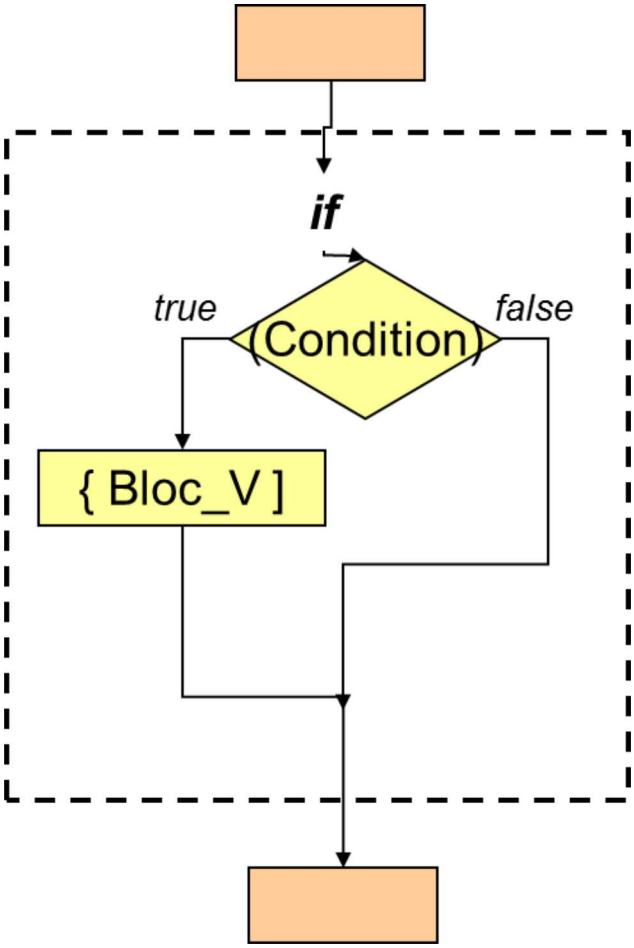
```
if (condition)
{
    // bloc d'instructions exécuté
    // si la condition est vraie
}
```

où :

- condition : une expression logique

Exemple :

```
int i = 1;
int j = 2;
int k;
if (i < k)
  { k = i;}
```



Eviter les operateurs pouvant produire des effets de bord dans les expressions de comparaison apres le premier and ou or (évalué seulement si la première condition est suffisante pour évaluer l'expression)

Exemple à rejeter :

```
int i;
int max;
// ...
if ( (i >= 0 && (++i) <= max) ) {
// ...
}
```

Exemple à utiliser :

```
int i;
int max;
// ...
if ( (i >= 0 && (i + 1) <= max) ) {
  i++;
// ...
}
```

Exécution conditionnelle avec alternative

```
if (condition)
{
    // bloc d'instructions exécuté
    // si la 1ère condition est vraie
}
else if (condition)
```

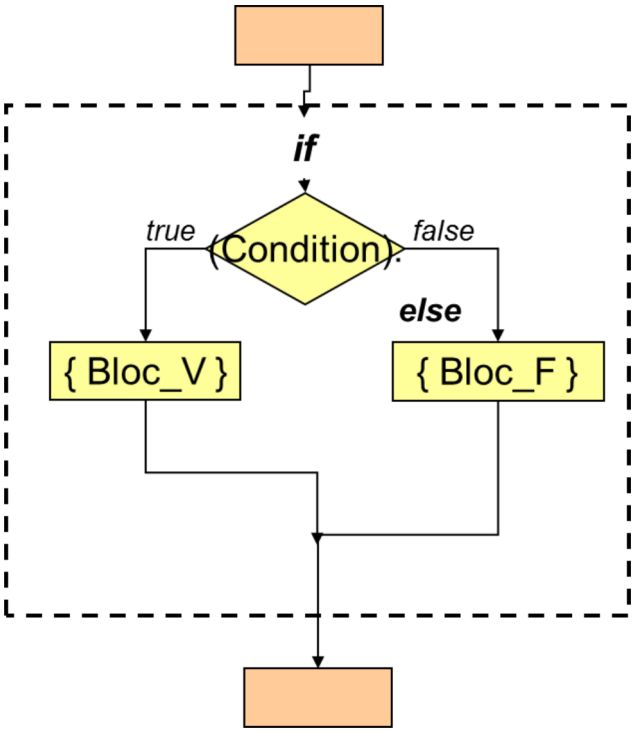
```
{
    // bloc d'instructions exécuté
    // si la 2ème condition est vraie
}
... etc...
else
{
    // bloc d'instructions exécuté
    // si aucune des conditions n'est satisfaite
}
```

où :

- condition : une expression logique

Exemple :

```
int i = 1;
int j = 2;
int k;
if (i < k)
  { k = i;}
  else
  { k = j;}
```



Opérateur conditionnel (ternaire)

Cet opérateur ternaire évalue une expression logique et choisit la valeur retournée

```
condition ? resultat1 : resultat2 ;
```

où

- condition : expression logique évalulée

- resultat1 : valeur de l'expression si la condition est vraie
- resultat2 : valeur de l'expression si la condition est fausse

Exemple :

```
int i,
    j;
...//calculs qui modifient la valeur de i...
...
j = (i < 0) ? (i * -1) : i;
```

cette expression est équivalente à :

```
if (i < 0) {j = (i * -1);} else {j = i;}
```

Choix multiple

Il est parfois utile de comparer une variables avec une liste de valeurs et exécuter une liste d'instructions seon le cas. Une fois le cas traité, l'instruction break permet de sauter à l'instructions qui suit la fin du switch. Si aucune instruction break n'est trouvée, le cas suivant est traité.

```
switch (intExpr)
{
    case valeur1 :
                // liste d'instructions exécutées
                                    // si intExpr = valeur1
                    break;
    case valeur2 :
    case valeur3 :
                // liste d'instructions exécutées
                // si intExpr = valeur2 ou intExpr = valeur3
                    break;
    default      :
                // liste d'instructions exécutées
                // si aucun des cas n'a été satisfait
                    break;
} // fin du switch
```

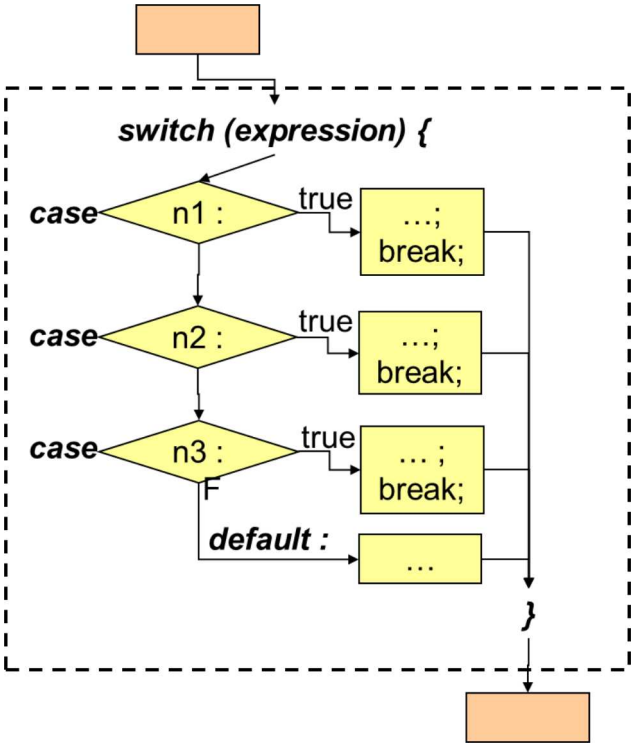
où :

- intExpr : une expression numérique entière
- valeurN : les différents cas de valeurs testées pour cette expression

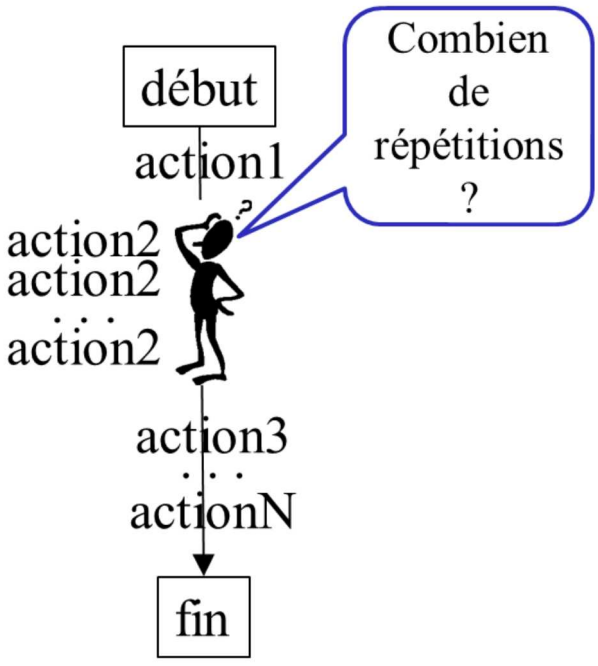
Exemple :

```
int m = 0;
int s = 0;
// traitement qui modifie la valeur de m
...
switch (m)
{
  case 1:
  case 2:
  case 3: s = 12; break;
  case 4:
```

```
  case 5:
  case 6: s = 20; break;
  default: s = 0; break;
}
```



6) Exécution répétée (Index)



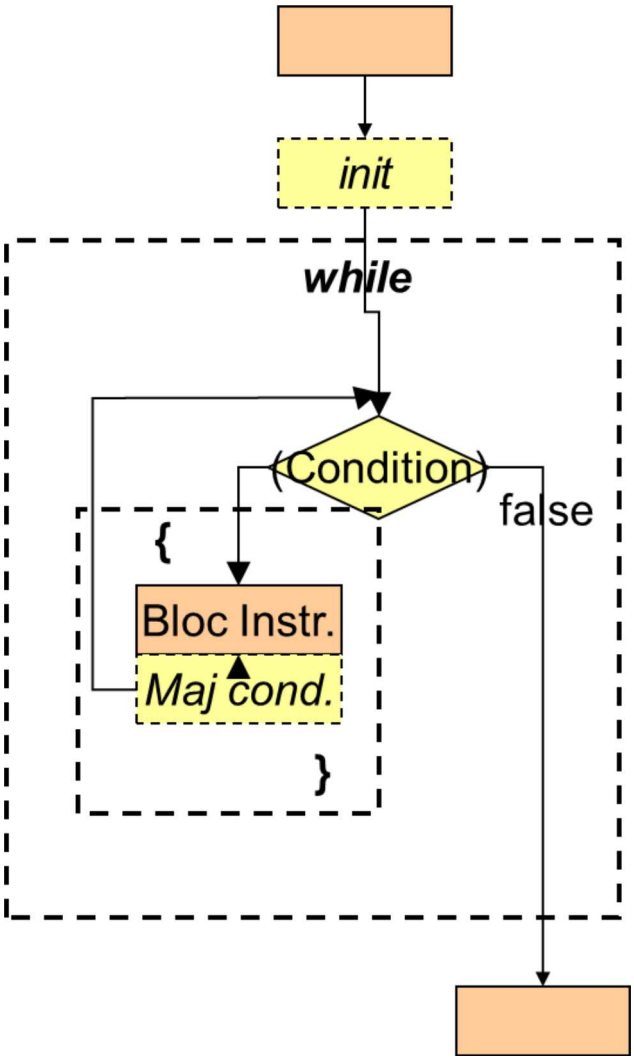
répétition de 0 à n fois

```
while (condition)
{
    //instructions si la condition est vraie et tant qu'elle
```

```
} // fin du while
```

où :

- condition : une expression logique évaluée avant chaque itération, qui si elle est vraie, et tant qu'elle reste vraie, va exécuter les instructions du bloc



La trace d'exécution est la suivante :

1. avoir initialisé les éléments de la condition
2. tester la condition
3. si la condition est vraie, exécuter les instructions sinon sauter à l'instruction qui suit la fin du while
4. retourner en 2 après exécution des instructions

Exemple :

```
int i = 0; // initialiser
while (i < 10) // tester
{
    // instructions exécutée si la condition est vraie
        //   et tant qu'elle reste vraie
        i++;
} // fin du while
```

répétition de 1 à n fois

```
do
{
    // instructions (exécutées au moins une fois)
```

```
} while (condition);
```

où :

- condition : une expression logique évaluée après chaque itération, qui va permette l'exécution des instructions du bloc (exécuté au moins une fois) tant qu'elle reste vraie,

La trace d'exécution est la suivante :

1. exécuter les instructions
2. tester la condition
3. si la condition est vraie, retourner en 1 sinon sauter à l'instruction qui suit la fin du while

Exemple :

```
int i = 0;
do
{
    //instructions si la condition est vraie et tant qu'elle reste vraie
        i++;
} while (i < 10);
```

répétition de 0 à n fois

```
for(initialiser; condition; modifier)
{
    // instructions

} // fin du for
```

où :

- initialiser : une initialisation (et éventuellement déclaration) des variables de contrôle de la boucle
- condition : une expression logique évaluée avant chaque itération, qui, si elle est vraie et tant qu'elle reste vraie, exécute les instructions du bloc
- modifier : une modification des variables de contrôle de la boucle exécutée après chaque itération

La trace d'exécution est la suivante :

1. initialiser une liste de valeurs (généralement une seule valeur, la variable de boucle)
2. tester la condition de poursuite de la boucle
3. si la condition est vraie, exécuter le bloc d'instructions sinon sauter à l'instruction qui suit la fin du for
4. après l'exécution du bloc, modifier les variables utilisée dans le test de la condition, puis retourner en 2

Cette structure de contrôle est équivalente à :

```
initialiser;
while (tester)
{
    // instructions
        modifier;
} // fin du while
```

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

```
    //instructions si la condition est vraie et tant qu'elle reste vraie
};
```

```
int i = 0;
while (i < 10)
{
    //instructions si la condition est vraie et tant qu'elle reste vraie
    i++;
}
```

Sortir d'une boucle : break

L'instruction break permet la sortie d'une boucle prématurément, avant que la condition n'ait été testée. En cas de boucles imbriquées, seule la boucle courante est quittée.

```
break;
```

Exemple :

```
int i = 0;
for (i < 10)
{
    // instructions
        if (i==v) break; // si i est égal à la variable v
        // instructions
        i++;
}
```

Passer à l'itération suivante : continue

L'instruction continue permet le passage à l'itération suivante (retourne au test de la condition ou à la modification dans le cas du for).

```
continue;
```

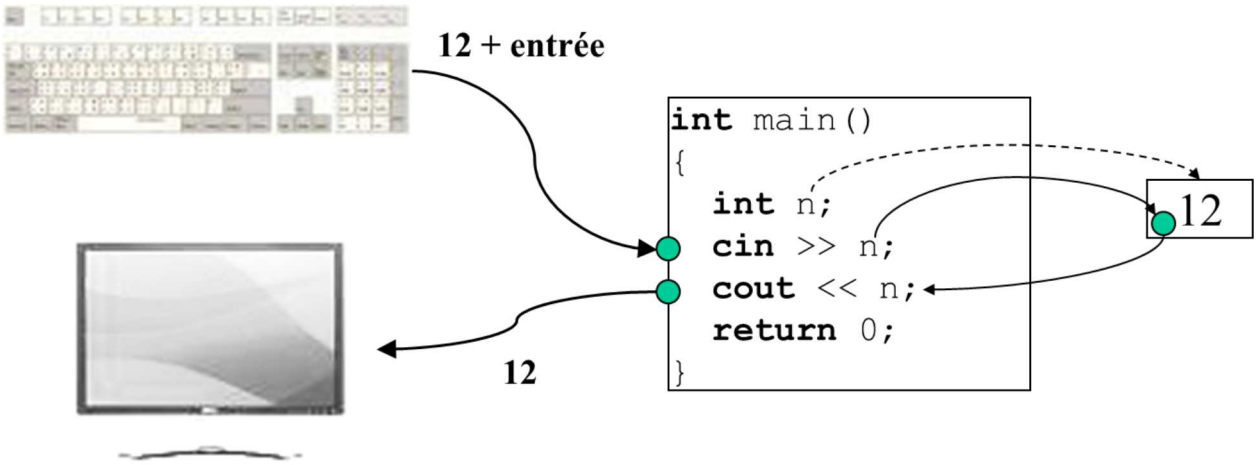
Exemple :

```
int sum = 0;
for (int i=0; i < 100; i++)
{
    if ((i%2)!=0) continue;
        sum+=i;
}
```

7) Entrées/Sorties standard (Index)

La bibliothèque **iostream** (Standard Input / Output Streams Library) définit les fichiers d'entête pour l'accès aux objets standard d'entrée/sortie.

flux	description
cin	flux d'entrée standard : clavier
cout	flux de sortie standard : écran (console)
cerr	flux de sortie des erreurs : écran
clog	flux de sortie des journalisations : écran



Flux de sortie

cout <<

Les flux de sortie formattée est combiné avec l'opérateur d'insertion << L'opérateur d'insertion insère les données qui le suivent dans le flux qui le précéde.

```
cout << donnees;
```

fonctions de formatage

Elles nécessitent l'inclusion de l'entête **iomanip**.

- setw(width) : définir le nombre de caractère de la prochaine insertion
- setfill('c') : définir le caractère de remplissage pour la prochaine insertion
- setprecision(p) : définir la précision des prochaines insertions
- setbase(b) : définir la base (10, 8, 16) des prochaines insertions
- left ou right : définir l'alignement de la prochaine insertion

Exemple (pour les variables suivantes :

```
int n1 = 1, n2 = 2;
double f1 = 3.1415927, f2 = 9.81, f3 = 98.6; // pi m/s2 37deg celsius
char c1 = 'a', c2 = 'z';
string ch1  ="hello", ch2 = "world";
cout << n1 << n2 << f1 << f2 << f3 << c1 << c2 << ch1 << ch2 << endl;
```

résultat :

```
123.141599.8198.6azhelloworld
```

Exemple setw :

```
cout << setfill('.'); // pour montrer les espacements
cout << setw(10) << n1 << setw(10)<< n2 << setw(10)<< f1 << setw(10)<< f2 << setw(10)<< f3
cout << setw(4) << c1 << setw(4)<< c2 << setw(8)<< ch1 << setw(8) <<ch2 << endl;
```

Résultat :

```
.........1.........2...3.14159......9.81......98.6
...a...z...hello...world
```

Exemple setfill :

```
cout << setw(40) << setfill('-') << '-' << endl;
cout << setfill('*');
cout << setw(10) << n1 << setw(10)<< n2 << setw(10)<< f1 << setw(10)<< f2 << endl;
```

```
cout << setfill(' ');
cout << setw(4) << c1 << setw(4)<< c2 << setw(8)<< ch1 << setw(8) <<ch2 << endl;
```

Résultat :

```
--------------------------------------
*********1*********2***3.14159******9.81
   a    z    hello    world
```

Exemple setprecision :

```
cout << setprecision(2);
cout << setw(10)<< f1 << setw(10)<< f2 << setw(10)<< f3 << endl;
```

Résultat :

```
       3.1       9.8        99
a    z    hello    world
```

Exemple left :

```
cout << left;
cout << setw(4) << c1 << setw(4)<< c2 << setw(8)<< ch1 << setw(8) <<ch2 << endl;
```

Résultat :

```
a    z    hello    world
```

Flux d'entrée

cin >>

Les flux d'entrée est combiné avec l'opérateur d'extraction >> L'opérateur d'extraction est suivi de la variable qui va contenir la valeur extraite du flux qui le précéde.

```
cin >> variable;
```

L'extraction s'arrête au premier espace rencontré (ou tabulation).

getline

La fonction getline récupère une ligne entière, jusqu'au caractère EOL (end of line):

```
getline(cin,variableChaine);
```

toute donnee provenant d'une source exterieure (saisie, fichier, etc.) doit etre controlee afin de limiter les possibles effets de bord

1. identifier toutes les sources possibles d'acquisition de donnees
2. identifier les points d'acquisition de donnees dans le code source
3. definir les criteres pour les donnees soient valides (nombres, chaines de caracteres, dates, etc.)
4. definir le comportement du programme si une donnee erronee est reçue
5. mettre en œuvre le code source pour controler les donnees

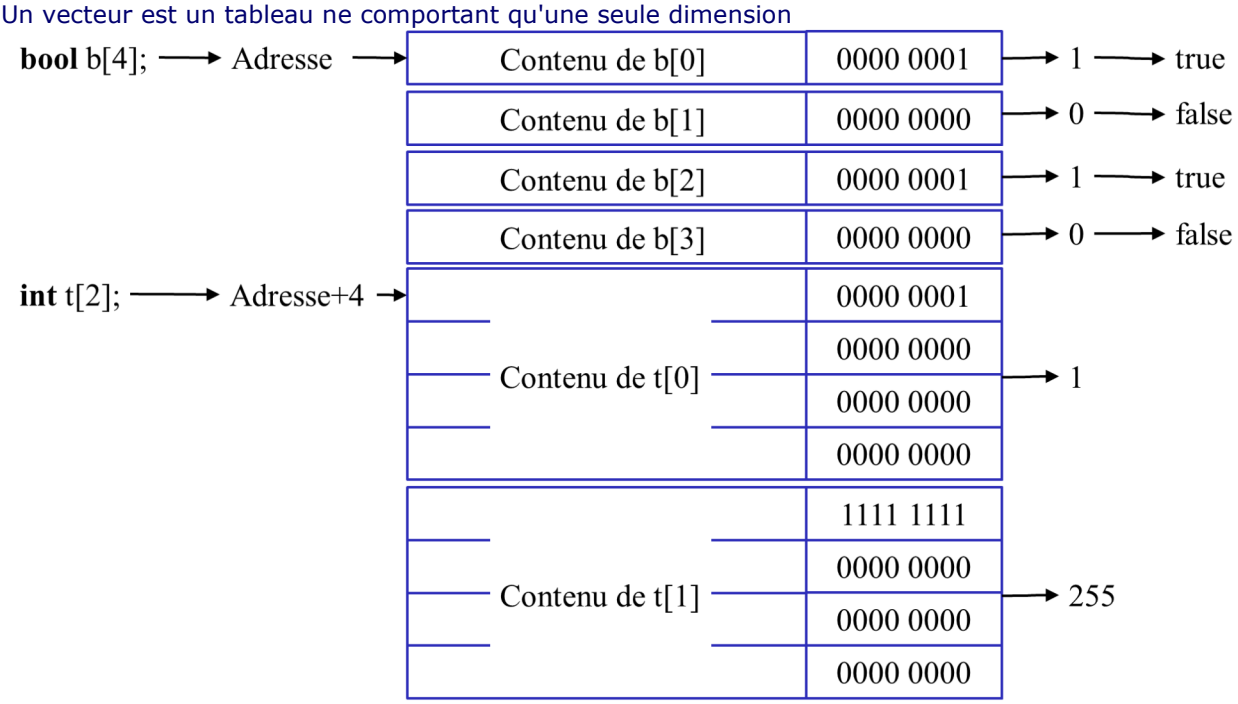
8) Tableaux (Index)

Un tableau est une juxtaposition, sous un nom unique, d'un certain nombre (entier naturel) de variables de même type, auxquelles on accède individuellement grâce à un numéro d'ordre, ou rang, qu'on appelle indice. En C/C++ , les indices sont

numérotés de 0 jusqu'àu nombre d'élements - 1. La taille du tableau est déterminé. On ne pourra la changer au cours de l'exécution. La dimension d'un tableau correspond au niveau d'imbrication de tableaux (cf. Vecteur, Matrice, autre)

Vecteur, tableau à 1 dimension

Un vecteur est un tableau ne comportant qu'une seule dimension



Déclaration

```
type idTab[nb];
type idTab[] = {lv};
```

où :

- type : un type de données
- idTab : l'identifiant attribué au tableau
- nb : nombre d'élements du tableau entre crochets
- lv : liste de valeurs séparées par une virgule permettant de déterminer son nombre d'élements et de les initialiser

Exemple :

```
bool b[10]; // tableau nommé b de 10 booléens
char ch[10]; // tableau nommé ch de 10 caractères
const unsigned int NB = 10; //(constante)
double t[NB]; // tableau nommé t de NB réels (ici 10)
```

Initialisation

Tous les éléments d'un tableau doivent être absolument initialisées avec toute utilisation sinon ils peuvent contenir n'importe quelle valeur : ils occupent en effet un espace mémoire dans lequel un autre programme s'exécutait auparavant.
L'initialisation peut êtte réalisée au moment de la déclaration :

```
type idTab[nb] = {lv};
type idTab[] = {lv};
```

où :

- type : un type de données
- idTab : l'identifiant attribué au tableau
- nb : nombre d'élements du tableau entre crochets

- lv : liste de valeurs séparées par une virgule permettant d'initialiser les élements du tableau : si le nombre de valeurs est inférieur au nombre d'élements, les autres sont initialisés à 0.

Exemple :

```
bool b[10] = {true, false, true, false}; // valeurs 1 0 1 puis des 0
char ch1[10] = {'h','e','l','l','o'}; // valeurs h e l l o puis des caractères nuls
char ch2[10] = "hello"; // valeurs h e l l o puis des caractères nuls
const unsigned int NB = 10; //(constante)
double t[NB] = {0}; // valeur 0 puis des 0
```

ou bien ultérieurement, dans une boucle de traitement :

```
for(int i=0;i<TAILLE;i++)
{
   idTab[i] = vInit;
}
```

où :

- vInit : valeur affectée à l'élement i

exemple :

```
for(int i=0;i<NB;i++)
{
   t[i] = (i + 1);
}
```

Déterminer la taille du tableau : sizeof

Le nombre d'élements d'un tableau peut être retrouvé de manière dynamique : il est égal à l'espace total occupé par le tableau (en nombre d'octets) divisé par l'espace occupé par chacun de ses éléments. L'opérateur **sizeof** peut être utilisé pour déterminer le nombre d'élements d'un tableau dont on connait le type de donnée :

```
int nbElem = ( sizeof idTab / sizeof(type) );
```

Exemple :

```
const unsigned int NB = 10;
int t[NB];
cout << sizeof t << endl; // --> 40
cout << sizeof(int) << endl; // --> 4
cout << ( sizeof t / sizeof(int) ) << endl; // --> 10
```

Ainsi, si on on souhaite être indépendant une valeur de constante de taille de tableau ou si, à l'endroit du programme où on se trouve, on ne connait pas le nombre d'éléments du tableau, on peut le retrouver simplement (à condition de connaitre le type de données du tableau...). Ainsi pour le tableau d'entier nommé tab:

```
for(int i=0;i<(sizeof tab / sizeof(int));i++)
{
   tab[i] = 0;
}
```

Tableaux de caractères : Cas particulier

Les tableaux de caractères sont les seuls accessibles élément par élément mais également de manière globale. On peut ainsi afficher complétement un tableau de caractères : il forme dans ce cas une chaine de caractères. Ainsi, l'exemple ci-dessous affichera "hello hello" :

```
cout << ch1 << " " << ch2;
```

Matrice , tableau à 2 dimensions

Une matrice est un tableau comportant 2 dimensions : chaque élément de sa premère dimension comporte à son tour un tableau d'un certain nombre d'élements (deuxième dimension).
Le nombre d'éléments d'une matrice est égal au nombre de sa première dimension X nombre de sa deuxième dimension.
L'accès à un élement particulier d'un tableau à 2 dimensions nécessite l'utilisation de 2 indices, un pour chacune des dimensions.

Déclaration

Ainsi pour un tableau idTab comportant nb1 tableau de nb2 éléments d'un certain type :

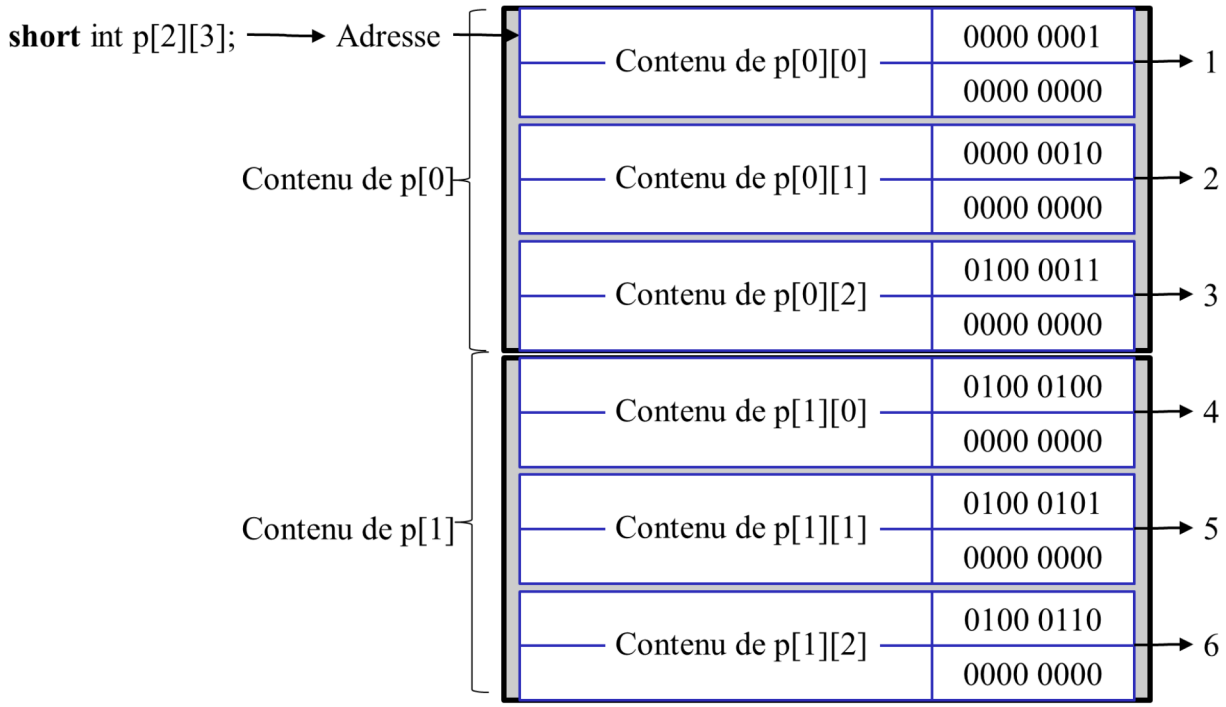
```
type idTab[nb1][nb2];
type idTab[nb1][nb2] = {{lv},{lv}, etc.};
```

où :

- type : un type de données
- idTab : l'identifiant attribué au tableau
- nb1 : nombre d'élements de la dimension 1 du tableau entre crochets
- nb2 : nombre d'élements de la dimension 2 du tableau entre crochets
- lv : liste de valeurs séparées par une virgule permettant de déterminer son nombre d'élements et de les initialiser

Exemple :

```
short int p[2][3]; // tableau de 6 éléments
```



Exemple (ce tableau avait initialisé comme ci-dessous):

```
short int compteur = 1;
for(int i;i<2;i++)
{
  for(int j;j<3;j++)
  {
    p[i][j] = compteur;
    compteur++;
  }
```

```
}
```

Exemple utilisant l'opérateur sizeof:

```
short int compteur = 1;
for(int i;i<(sizeof p / sizeof p[0]);i++)
{
  for(int j;j<(sizeof p[0] / sizeof(short int);j++)
  {
    p[i][j] = compteur;
    compteur++;
  }
}
```

On peut donner un sens à ces dimensions :

- lignes et colonnes d'une matrice
- lignes et colonnes d'un plateau de jeu
- espace à 2 dimensions comportant des valeurs aux intersections des abcisses et ordonnées entières.

au delà , tableau à N dimensions

On peut ainsi définir des tableaux multidimensionnels, mais il faut avoir conscience que l'espace occupé va croitre de manière importante et que cette structure de donnée n'est peut être pas adaptée (si peu d'elements sont utilisés dans chacune ds dimensions).
Ainsi le tableau suivant va avoir 10 X 20 X 30, soit 6000 élements; il nécessitera 6000 opérations pour initialiser chacun de ses éléments et il occupe au total 6000*8 = 48000 octets, 48ko.
Exemple :

```
double p[10][20][20];
for(unsigned int i;i<(sizeof p / sizeof p[0]);i++)
{
  for(unsigned int j;j<(sizeof p[0]/ sizeof p[0][0]);j++)
  {
    for(unsigned int k;k<(sizeof p[0][0] / sizeof(double));k++)
    {
      p[i][j][k] = 0;
    }
  }
}
```

9) Structures (Index)

Un type de données structure (ou enregistrement) permet la définition d'un nouveau type composé de variables de types différents et de représenter ainsi des données complexes.

Définition

```
struct idStr
{
   déclaration1;
   déclaration2;
   ...
   déclarationN;
};
```

où :

- idStr : identifiant attribué à la structure
- déclarationN : la déclaration d'une variable

exemple :

```
struct Point
{
        double x,y,z;
        string n;
};
```

Utilisation

Le type de donnée peut être utilisé comme tout autre type :

```
idStr idVarStr;
```

où :

- idStr : identifiant attribué à la structure
- idVarStr : la déclaration d'une variable du type de donnée idStr

exemple :

```
Point p1, p2;
```

Accès aux membres

L'accès aux variables composant la structure utilise une notation pointée :

```
idVarStr.idVar1 = valeur;
```

où :

- idVarStr : d'une variable du type de donnée idStr
- idVarN : un membre variable de cette structure

exemple :

```
p1.x = 0, p1.y = 0, p1.z = 0;
p1.n = "origine"
```

Imbrication de types structurés

La définition d'un type structure peut comporter d'autres types structurés : exemple :

```
typedef unsigned char byte; // alias sur le type unsigned char
struct Couleur
{
        byte rouge, vert, bleu;
}
struct Point
{
    double x,y,z;
        string n;
        Couleur c;
}
```

Utilisation :

```
Point p1;
```

```
p1.x = 0,p1.y = 0, p1.z = 0;
p1.n = "origine";
p1.c.rouge = 255, p1.c.vert = 255, p1.c.bleu = 255;
```

Tableau de types structurés

Les types structures peuvent être utilisés, comme tout autre type de données, sous forme de tableaux. exemple :

```
const unsigned int NB_POINTS = 20;
Point points[NB_POINTS];
```

Utilisation (tous les points sont noirs):

```
for(int i=0;i<NB_POINTS;i++)
{
    points[i].c.rouge = 0;
    points[i].c.vert = 0;
    points[i].c.bleu = 0;
}
```

Union

le mot clef union permet de redéfinir l'espace mémoire d'un type structure de différentes manières. Dans l'exemple qui suit, le type figure comporte

1. un point de référence (pRef), centre d'un cercle ou d'un polygone régulier
2. puis :
    1. soit une structure pour décrire le cercle, comportant un rayon
    2. soit une structure pour décrire le polygone, comportant un nombre de côtés et une taille de côté

Par exemple :

```
enum figure {CERCLE, POLYREG};
struct Cercle {
  unsigned int r
};
struct Polyreg {
  unsigned int nbc;
  unsigned int c;
};
struct Figure {
        Point pRef;
        figure fig;
        union {
           Cercle unCercle;
           Polyreg unPolyreg;
        };
};
```

Utilisation :

```
Figure f[10];
for(int i=0;i<10;i++)
{
    switch (f[i].fig)
        {
                case CERCLE :
                        cout << "cercle : r=" << f[i].unCercle.r;
                        break;
                case POLYREG :
                        cout << "polygone : " << f[i].unPolyreg.nbc <<"cotes";
                        cout << " c=" << f[i].unPolyreg.c;
                        break;
```

```
    }
}
```

10) Sous-programmes fonctions et procédures (Index)

La notion de sous-programme permet

- d'une part d'attribuer un identifiant à une action complexe et d'y faire appel dans le code source principal,
- et par conséquent, d'alléger de manière significative le code source principal.

De plus, les sous-programmes ainsi définis, peuvent être appelés plusieurs fois, sans devoir être réécrits.

```
type idSsProg (listeParams)
{
  // corps du sous-programme
  // = liste d' instructions à exécuter
}
```

où :

- idSsProg : l'identifiant attribué au sous-programme
- listeParams : paramètres formels (peut être vide), liste de déclarations de variables attendus lors de l'appel du sous-programme

L'appel d'un sous-programme fait référence à son identifiant et à la liste des valeurs effectives passées en argument :

```
idSsProg (listeArgum);
```

où :

- idSsProg : l'identifiant attribué au sous-programme
- listeArgum : augements ou paramètres réels, liste de valeurs effectivement passées lors de l'appel du sous-programme

Procédures

La procédure est un sous-programme de type action, qui effectue un certain traitement sans retourner de valeur de retour. Son type de retour est **void**.

```
void idSsProg (listeParams)
{
  // corps de la procédure
  // = liste d' instructions à exécuter
}
```

Exemple :

```
void afficherNombre (int n)
{
  cout << n;
  return; // optionnel
} // fin
```

L'instruction return est optionnelle; elle peut être placée à n'importe quel endroit et renvoie le contrôle à l'instruction qui suit l'appel. Sans instruction return, le sous-programme s'exécute jusqu'à la fin du code de son corps.

```
int n = 10;
```

```
afficherNombre(n*2);
```

Fonctions

La fonction est un sous-programme de type expression (elle est typée), qui effectue un certain traitement et retourne une valeur unique à la fin de son exécution. Elle peut être placée partout où on attendrait une valeur du type de la fonction.

```
type idSsProg (listeParams)
{
  // corps de la fonction
  // = liste d' instructions à exécuter
  return exprType;
}
```

où :

- type : type de retour de la fonction
- exprType : est une expression calculée, une variable, un littéral, etc. du type de retour de la fonction

Exemple :

```
unsigned int valeurAbsolue (const int n)
{
  unsigned int resultat;

  if (n>=0) {resultat =  n;}
  else {resultat = (n * -1);}

  return resultat;
}
```

ou bien :

```
unsigned int valeurAbsolue (const int n)
{
  if (n>=0) {return n;}
  else {return (n*-1); }
}
```

ou encore (utilisation de l'opérateur conditionnel):

```
unsigned int valeurAbsolue (const int n)
{
  return ((n>=0) ?  n : (n * -1));
}
```

L'instruction return est obligatoire; elle peut être placée à n'importe quel endroit et renvoie une valeur qui pourra être récupérée par l'instruction d'appel.

```
int n = -10;
n = valeurAbsolue(n*2) * n;
```

La fonction 'main'

La fonction 'main' est le point d'entrée de tout programme C++. On devra donc la trouver dans l'un des fichiers qui composent le projet.

Prototype de sous-programme

Le prototype d'un sous-programme comporte uniquement son entête. Il est utile lorsque les définitions des sous-programme se trouvent dans d'autres fichiers sources ou, dans le même code source,après l'appel du sous-programme. Les fichiers source .h comportent généralement des prototypes de sous-programmes.

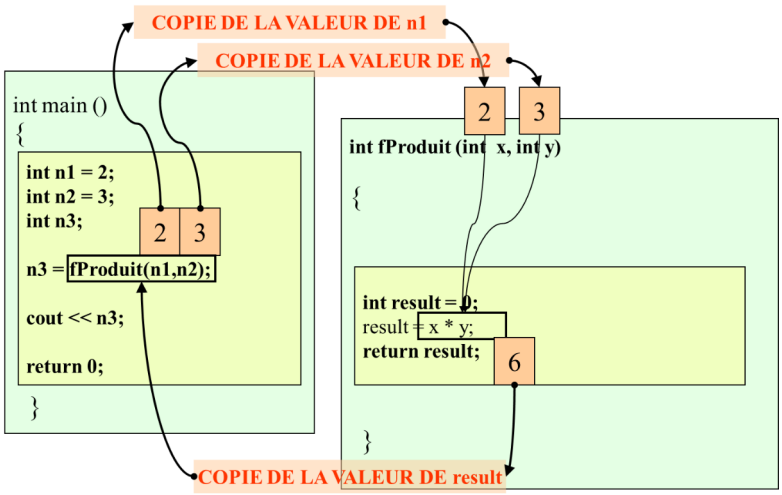
```
void afficherNombre (int);
unsigned int valeurAbsolue (const int);
```

Paramètres

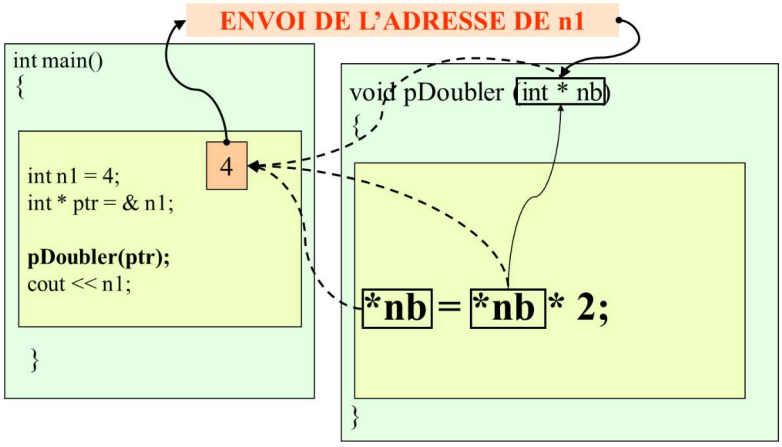
Il existe 2 modes de passage des arguments vers les paramètres définis dans le sous-programme qui vont déterminer la possibilité qu'a un sous-programme de modifier ou non directement la valeur de ces arguments :

- par copie de valeur, par pointeur constant ou par référence constante : le sous-programme a un accès à la valeur sans pouvoir la modifier, soit parce qu'elle a été copiée soit par l'intermédiaire de son adresse ou de sa référence (définie ici comme constante, donc non modifiable)
- par pointeur ou par référence : le sous-programme reçoit l'adresse de la variable argument passée au sous, il peut donc modifier sa valeur

Passage par valeur



Passage par pointeur



Passage par référence

11) Portée et visibilité des déclarations (Index)

La notion de portée (anglais scope) fait référence à la région d'un programme à l'intérieur de laquelle une déclaration (donnée, sous-programme) est connue. La portée démarre à sa déclaration et se prolonge jusqu'à la fin du bloc où elle a été déclarée.

La notion de visibilité fait référence au fait qu'une déclaration effectuée à un niveau inférieur puisse masquer une déclaration effectuée à un niveau supérieur, celle-ci n'étant plus visible jusqu'à la fin de la portée de la déclaration qui la masque (cf. exemple plus bas).

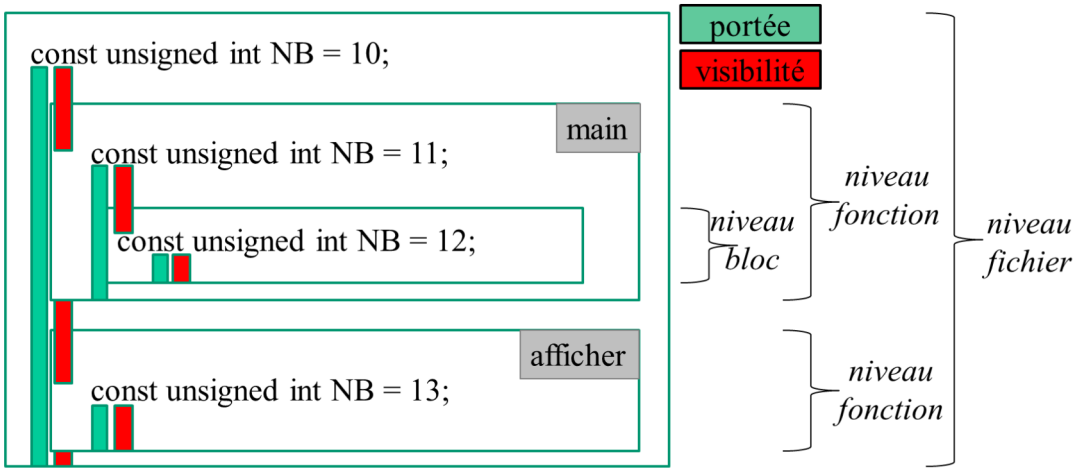
Dans un programme C++, on peut distinguer 3 niveaux de portée (de 1 à 3, 1 étant le plus élevé), une déclaration effectuée au niveau 1 étant accessible des niveaux inférieurs :

1. au niveau du fichier : on y trouve des prototypes de procédures/fonctions, des constantes, des définitions de types structure
2. au niveau d'un sous-programme : on y trouve des déclarations de variables
3. dans un bloc défini par des accolades : déclarations de variables locales au bloc

```
#include <iostream>
const unsigned int NB = 10; // niveau fichier
void afficher(unsigned int n);
int main()
{
  cout << "(1)" << NB << endl;
  const unsigned int NB = 11; // niveau fonction
  cout << "(2)" << NB << endl;
  {
    const unsigned int NB = 12; // niveau bloc
        cout << "(3)" << NB << endl;
  }
  cout << "(4)" << NB << endl;
  afficher(NB);
}
void afficher(unsigned int n)
{
  cout << "(5)" << NB << endl;
  const unsigned int NB = 13; // niveau fonction
  cout << "(6)" << NB << endl;
}
```

résultat :

```
(1)10
(2)11
(3)12
(4)11
(5)10
(6)13
```



12) Pointeurs et références (Index)

Pointeurs sur les types de base

A sa déclaration, une variable est associée à un espace mémoire pouvant stocker son contenu. De même, lé définition d'une fonction nécessite un espace mémoire utile au stockage des instructions qui y seront exécutées.

Un pointeur est une variable à part entière (un espace mémoire lui est réservé) qui peut contenir l'adresse mémoire d'un autre

objet : variable, fonction

Les pointeurs permettent d'accéder à n'importe quel emplacement de la mémoire réservée à un programme : ils présentent un risque s'ils sont mal initialisés ou pas correctement utilisés. En effet, c'est au développeur que revient entièrement leur contrôle. Les pointeurs non initialisés (ou mal initialisés) sont la cause d'erreurs fréquentes d'exécution (écran bleu).

Déclaration des pointeurs

Un pointeur est associé au type de données de la variable qu'il va pointer (adresser).

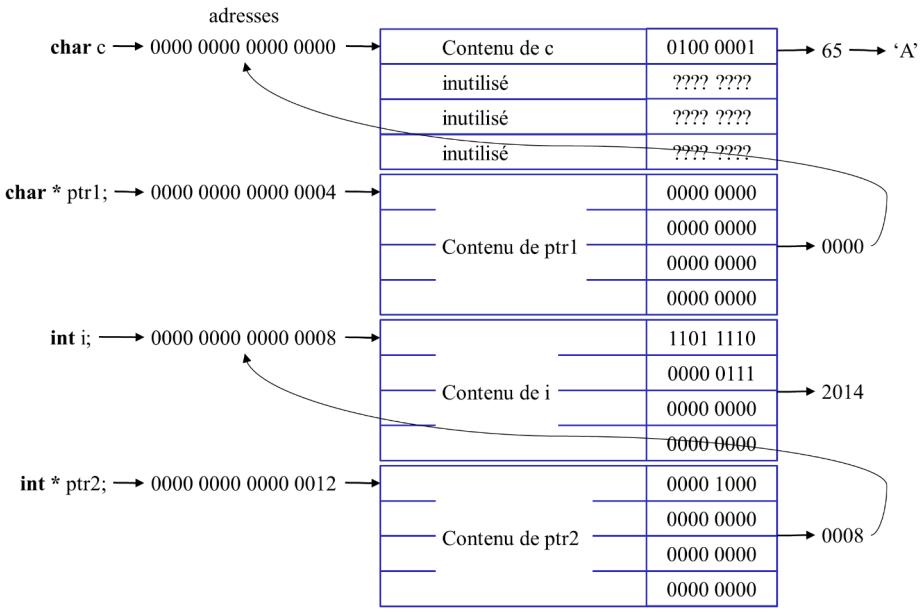
```
type * idPtr1 = NULL;
type * idPtr2 = &idVar;
```

où

- type: type de donnée de la variable pointée
- idPtr1, idPtr2 : identifiants de pointeurs
- NULL : valeur initiale correspondant au 0 binaire
- & : opérateur dit "d'indirection" qui fournit l'adresse mémoire de l'élement qui lui succède
- idVar : la variable pointée

Exemple :

```
char c = 'a';
char * ptrc = &c;
int i = 2014;
char * ptri = &i;
```



Pointeur vers types de base

Déclaration d'un pointeur :

```
type * idPtr;
type * idPtr = NULL;
```

où :

- type : type de donnée de l'objet pointé

- idPtr : identifiant du pointeur

Exemple :

```
int * ptrInt = NULL;
double * ptrDbl = NULL;
char * ptrCh = NULL;
```

Initialisation d'un pointeur :

```
idPtr = & idObj;
```

où :

- idPtr : identifiant du pointeur
- & : opérateur d'indirection, fournit l'adresse de l'objet qui suit
- idObj : identifiant de l'objet pointé

Exemple :

```
int n = 10;
double v = 12.5;
char c = 'a';
ptrInt = & n;
ptrDbl = & v;
ptrCh = & c;
```

Utilisation d'un pointeur :

```
* idPtr
```

où :

- * : opérateur de déréférencement : accès au contenu pointé
- idPtr : identifiant du pointeur

Exemple :

```
(*ptrInt) ++;
(*ptrDbl) +=2;
*ptrCh = *ptrCh + 1;
cout << "n:" << n  << " " << (*ptrInt)<<" " << ptrInt << endl;
cout << "v:" << v  << " " << (*ptrDbl)<<" " << ptrDbl << endl;
cout << "c:" << c  << " " << (*ptrCh)<< " " << ptrCh << endl;
```

Pointeur vers type structure

Déclaration

```
struct point
{
  double x, y;
}
point * ptrPt;
```

Initialisation

```
point p1;
ptrPt = & p1;
```

Utilisation

```
ptrPt->x = 2;
ptrPt->y = 5;
// ou bien :
(*ptrPt).x = 2;
(*ptrPt).y = 5;
```

Pointeur vers les tableaux

Le tableau est un cas particulier de données : lorsqu'on fait référence à un tableau, c'est en fait à l'adresse de son premier élément qu'on fait référence.

Déclaration

```
const unsigned int NB = 10;
int tab[NB];
int * ptrInt;
```

Initialisation

```
ptrInt = tab;
ptrInt = & tab[0];
```

Utilisation

```
ptrInt[4] = 10 ;
*(ptrInt+3) = 10 ;
// est équivalent à :
tab[4] = 10 ;
*(tab+3) = 10 ;
```

Pointeur vers fonctions

Le tableau est un cas particulier de données : lorsqu'on fait référence à un tableau, c'est en fait à l'adresse de son premier élément qu'on fait référence.

Déclaration

```
int fSomme(int n1, int n2)
{
    return (n1 + n2) ;
};
// déclaration d'un pointeur vers fonction à 2 paramètres entiers :
int (*ptrFonc)(int, int);
```

Initialisation

```
ptrFonc = &fSomme;
```

Utilisation

```
int n1 = 10;
int n2 = 20;
int n3 = (*ptrFonc)(n1, n2) ;
```

Exemple complet

```
// définition de fonctions de calculs arithmétiques
int fSomme(int Vnb1, int Vnb2)
  {    return (Vnb1+Vnb2) ; }
int fDiff(int Vnb1, int Vnb2)
  {    return (Vnb1-Vnb2) ;}
int fProduit(int Vnb1, int Vnb2)
  {    return (Vnb1*Vnb2) ; }
```

```
int fQuotient(int Vnb1, int Vnb2)
  {    return (Vnb1/Vnb2) ;}
int fModulo(int Vnb1, int Vnb2)
  {    return (Vnb1%Vnb2) ;}

typedef int (*ptrf)(int, int); // décl. pointeur sur fonction

ptrf Tfonc[5] ; // allocation d'un tableau de pointeurs

int main (void)
{
    int n1, n2, numFonc ;
    Tfonc[0]= &fSomme ;
    Tfonc[1]= &fDiff ;
    Tfonc[2]= &fProduit ;
    Tfonc[3]= &fQuotient ;
    Tfonc[4]= &fModulo ;

    cout << "entrez 2 nombres :" ;
    cin >> n1 >> n2 ;

    cout << " entrez un numéro de fonction (0 à 4) " ;
    cin >> numFonc;

    cout << "  resultat = "  << (*(Tfonc[numFonc]))(n1, n2) );

    return 0 ;
}
```

Pointeurs et allocation dynamique

Références

13) POO et classes (Index)

La Programmation Orientée Objet est un démarche de conception des programmes (un paradigme de programmation) dans lequel interviennent les notions d'objets et de classes d'objets, et d'interactions entre objets.

Dans un programme classique, les "objets" modélisés sont représentés sous forme de listes de variables et les opérations qui les manipulent sous forme de procédures et fonctions.

Déclaration des entêtes des fonctions :(point.h)

```
struct point {
        double x,y;
};
double calcDistance(point p1, point p2);
point rotation(point p1, double a);
void afficher(point p1);
```

Définition des fonctions : (point.cpp)

```
#include
#include
#include "point.h"
using namespace std;

double calcDistance(point p1, point p2)
{
        return sqrt(pow(p2.x - p1.x,2)+(pow(p2.y - p1.y,2)));
```

```
}
point rotation(point p1, double a)
{
        point r;
        r.x = p1.x * cos(a) - p1.y * sin(a);
        r.y = p1.x * sin(a) + p1.y * cos(a);
        return r;
}
void afficher(point p1)
{
        cout << "Point (" << p1.x << "," << p1.y << ")" << endl;
}
```

Fonction principale :(main.cpp)

```
#include
#include "point.h"
using namespace std;
int main()
{
  /* declarations */
  const double PI = 3.1415927;
  point p1, p2;
  double d;
  /* initialisation */
  p1.x = 1, p1.y = 1, p2.x = 2, p2.y = 2;
  /* traitement */
  afficher(p1);
  afficher(p2);
  d = calcDistance(p1,p2);
  cout << "distance=" << d << endl;
  p1 = rotation(p1, PI);
  afficher(p1);
  return 0;
}
```

En POO, les caractérisques qui décrivent un objet et les sous-programmes qui les manipulent sont regroupés dans une superstructure, la classe, qui décrit ainsi un ensemble d'objets de même type. Dans le programme, les "objets" modélisés sont représentés sous une forme plus proche de la réalité.

Déclaration de la classe :

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

class Point {
  private:
    double  x,y;
  public:
    Point (); // point à l'origine
    Point (double , double); // point en x y
    void setXY(double , double); // modifier x y
    double calcDistance(Point p2);
    void rotation(double a);
        string getTexte();
};
```

Définition de la classe :

```
#include
#include "Point.h"
#include
```

```
Point::Point()
{
        x = 0;
        y = 0;
}
Point::Point(double xn, double yn)
{
        x = xn;
        y = yn;
}
void Point::setXY(double xn, double yn)
{
        x = xn;
        y = yn;
}
double Point::calcDistance(Point p2)
{
        return sqrt(pow(p2.x - x,2)+(pow(p2.y - y,2)));
}
void Point::rotation(double a)
{
        Point r(0,0);
        r.x = x * cos(a) - y * sin(a);
        r.y = x * sin(a) + y * cos(a);
        x = r.x;
        y = r.y;
}
string Point::getTexte()
{
     char buffer [50];
     int n, a=5, b=3;
     sprintf (buffer, "Point (%lf, %lf)", x,y);
         return buffer;
}
```

Définition du programme

```
#include
#include "Point.h"
void afficher(Point p)
{
cout << p.getTexte() <<  endl;
}
int main()
{
  /* declarations */
  Point p1, p2;
  const double PI = 3.1415927;
  double d;
  /* initialisation */
  p1.setXY(1,1);
  p2.setXY(2,2);
  /* traitement */
  afficher(p1);
  afficher(p2);
  d = p1.calcDistance(p2);
  cout <<  "distance=" <<  d <<  endl;
  p1.rotation(PI);
  afficher(p1);
  return 0;
}
```