1. Du C au C++

1.1. Objets et classes

La programmation en C++ diffère de celle en langage C de par son style. En effet, le C++ utilise une collection d'objets comprenant à la fois une structure de données et un comportement, alors que pour le langage C les données et les traitements sont vraiment séparés. L'approche du développement logiciel est fondée sur la modélisation du monde réel.

Les structures de contrôle, boucles, conditions sont identiques de même pour le reste de la syntaxe. En revanche, chaque type devient une classe.

Dissocions maintenant les termes **objet** et **classe** :

Une classe est un type défini par le programmeur ou faisant partie d'une bibliothèque. C'est un ensemble regroupant les mêmes **attributs** ou données membres et disposant des mêmes **méthodes** ou fonctions membres.

Un objet est un élément qui possède sa propre identité, ses propres caractéristiques. Deux objets sont distincts même si tous leurs **attributs** ou **données membres** ont des valeurs identiques. On dit qu'un objet est une **instance** de **sa classe.** C'est l'équivalent d'une variable pour la programmation en langage C.

Attention : une classe ne contient pas d'objets. Une classe aura par défaut, ses données **privées** et ses méthodes **publiques**. C'est le principe de l'**encapsulation**.

La notion de classe sera développée au chapitre 2 : **Définition de classe**

1.2. Classification des objets

Ce paragraphe s'applique tous les objets, car comme cela a été dit précédemment en C++, tous les types sont des classes.

Les objets **automatiques** sont créés par une déclaration soit dans une fonction, soit au sein d'un bloc. Ils sont créés au moment de l'exécution de la déclaration. Ils sont détruits lorsque l'on sort de la fonction ou du bloc. Les objets automatiques sont détruits dans l'ordre inverse de leur création. Leur durée de vie et leur visibilité sont égales à celle du bloc ou de la fonction où ils sont déclarés. On les associe aux variables locales du langage C.

Les objets **statiques** sont créés par une déclaration située en dehors de toute fonction ou par une déclaration précédée du mot clé ***static***, dans une fonction ou dans un bloc. Ils sont créés avant l'entrée dans la fonction ***main*** et détruits après la fin de son exécution. Ces objets sont accessibles par toutes les méthodes constituant le programme et leur durée de vie est égale à celle de l'application. On les associe aux variables globales du langage C.

Les objets **dynamiques** sont créés par l'opérateur ***new*** et doivent être détruits explicitement par l'opérateur ***delete***. La règle de visibilité est égale à celle du bloc ou de la méthode où ils sont déclarés. Leur durée de vie dépend du programmeur, la mémoire est libérée après l'utilisation de l'opérateur ***delete***.

Les objets **temporaires** sont créés lors de l'appel explicite d'une méthode de classe ou d'un constructeur de classe lors d'un passage par valeur ou d'un retour par valeur d'un objet en argument ou encore lors de l'affectation d'un objet existant sur un nouvel objet créé. Ces objets temporaires permettent, tout simplement, la recopie d'un objet.

1.3. Espace de nommage

Les espaces de nommage ou espaces de noms ont été apportés au C++ pour éviter les conflits entre identificateurs globaux qui pourraient apparaître lors de l'utilisation de nombreuses bibliothèques.

La création d'un espace de nom conduit à réaliser une région déclarative de la forme :

Déclaration d'un espace de nom

```
namespace identifiant
{
}
```

La bibliothèque réalisée est placée entre les accolades.

Par exemple, la librairie standard du C++ est placée dans l'espace de nom ***std***. L'accès à chacun de ses membres doit être précédé du nom de l'espace, ici ***std*****,** et de l'opérateur de résolution de portée **::** comme le montre l'exemple suivant qui déclare une chaîne de caractère de type ***string***. Cet objet remplace, en simplifiant, un tableau de caractères et toutes les fonctions de traitement sur les chaînes du langage C. La taille n'a pas besoin d'être définie à l'avance et elle peut évoluer dynamiquement. De nombreuses méthodes permettent d'agir sur la chaîne. L'ensemble de la classe ***string*** est décrite ici : http://www.cplusplus.com/reference/string/string/.

Utilisation d'un élément de l'espace de nom std

```
#include <string>

int main()
{
    std::string uneChaine ;
    //utilisation de la variable uneChaine

    return 0 ;
}
```

Une autre manière de procéder, lorsqu'il n'y a pas de risque de conflits, est d'indiquer l'utilisation d'un espace de nom de manière implicite.

Utilisation d'un élément de l'espace de nom std de manière implicite

```
#include <string>

using namespace std;

int main()
{
    string uneChaine ;
    //utilisation de la variable uneChaine

    return 0 ;
}
```

En utilisant la directive ***using namespace***, il n'est plus nécessaire de faire précéder les éléments du nom de l'espace de nommage, ici ***std***, et de l'opérateur de résolution de portée **::**

1.4. Type booléen

Comme en C, pour une question de rétrocompatibilité, tout ce qui est différent de 0 est VRAI et tout ce qui est égal à 0 est FAUX. Cependant, en C++ le type ***bool*** est une alternative pour les expressions conditionnelles. Il peut prendre deux valeurs ***true*** et ***false***.

1.5. Surcharge ou polymorphisme de traitement

En C++, des fonctions distinctes peuvent avoir le même nom, si elles possèdent des paramètres relativement différents par leur type ou par leur nombre. Cela permet d'en simplifier l'utilisation, en appelant la même fonction du point de vue de l'utilisateur, on peut lui passer des paramètres différents

Exemple de surcharge

```
int   CalculerPuissance(int, int);
float CalculerPuissance(float, int);
float CalculerPuissance(float, double);
```

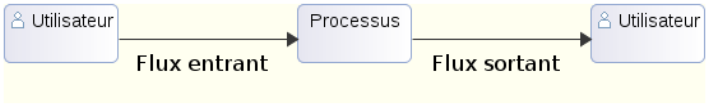
On parle ici de ***polymorphisme de traitement***, c'est le compilateur qui choisit la méthode à utiliser en fonction du type et du nombre de paramètres au moment de l'appel.

Cette surcharge peut également s'appliquer aux opérateurs. Classiquement, un opérateur peut agir sur deux variables de type simple, on pourra surcharger l'opérateur pour qu'il s'applique à de nouveaux types comme les classes pour obtenir un comportement similaire.

1.6. Entrées / Sorties : les flux

1.6.1. Généralités

Tout comme le langage C, le C++ ne dispose pas de mots clés spécifiques pour réaliser les entrées / sorties avec l'utilisateur ou plus généralement avec le système. Il dispose de librairies standards qui permettent de mettre en œuvre cela. En C++, les entrées / sorties sont réalisées à l'aide de mécanismes nommés **flux**. Un flux ou *stream* en anglais est une manière abstraite de représenter un flot de données entre un producteur d'information et un consommateur.



Le flux est toujours vu côté processeur. C'est le processus qui reçoit l'information par le flux entrant et qui l'envoie à travers le flux sortant.

Les entrées et sorties utilisant les flux sont définies par deux classes déclarées dans le fichier d'en-tête ***<iostream>*** :

- ***ostream*** pour *Output stream*, définit le flux sortant. Cette classe surcharge l'opérateur d'insertion **<<**.
- ***istream*** pour *Input stream*, définit le flux entrant. Cette classe surcharge l'opérateur d'extraction **>>**.

1.6.2. Flux standards

Tout comme le langage C utilise l'entrée standard ***stdin*** et la sortie standard ***stdout***, le C++ associe un flux sortant vers cette sortie et un flux entrant depuis cette entrée. On trouve également un flux vers la sortie d'erreur et un dernier vers la sortie technique ou sortie d'information.

- ***cout*** : écrit vers la sortie standard, l'écran,
- ***cin*** : lit à partir de l'entrée standard, le clavier,
- ***cerr*** : écrit sur la sortie d'erreur, cette sortie est non-tamponnée,
- ***clog*** : écrit sur la sortie technique, cette sortie est tamponnée.

Ces quatre instances représentent les flux standards en C++. Elles sont définies dans l'espace de nommage ***std***. Les deux dernières instances sont utilisées pour afficher ou mémoriser les messages d'erreur ou d'information. L'instance ***std::cerr*** ne possède pas de tampon cela implique que le message d'erreur s'affiche directement sans attendre que le tampon de sortie soit plein ou l'utilisation des manipulateurs ***std::flush*** ou ***std::endl***.

Le manipulateur ***std::endl*** écrit une fin de ligne **'\n'** sur le flux sortant puis vide le tampon. Le manipulateur ***std::flush*** vide uniquement le tampon d'écriture, le flux est physiquement envoyé sur l'unité de sortie.

Écriture sur la sortie standard ***bienvenue.cpp***

```
#include <iostream>          // pour cout
using namespace std;  // évite d'écrire std::cout

int main()
{
   cout << "Bienvenue en C++" ; // idem printf("Bienvenue en C++"); du langage C
   return 0;
}
```

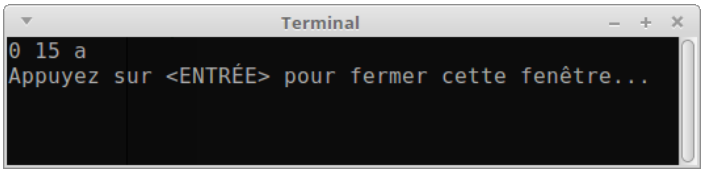
Écriture :

La classe ***ostream*** permet de réaliser un affichage formaté à la manière du ***printf*** de la librairie ***stdio*** en langage C. Ce formatage est beaucoup plus simple puisqu'il suffit d'enchaîner les opérateurs **<<,** comme le montre l'exemple ci-dessous. La surcharge de l'opérateur a été réalisée pour tous les types simples du C++ et les chaînes de caractères.

Écriture formatée ***formatage.cpp***

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

using namespace std;        // évite d'écrire std::cout

int main()
{
      bool sortie = false;
      int  entier = 15;
      char car = 'a';
      cout << sortie << " " << entier << " " << car  << endl;
      return 0;
}
```

Le résultat obtenu est présenté à la figure suivante :



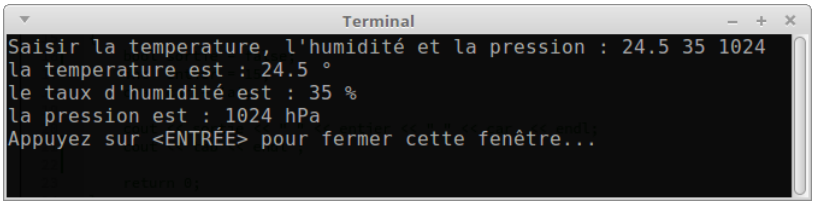
Lecture :

La classe ***istream*** permet la saisie de données à la manière du ***scanf*** de la librairie ***stdio*** du langage C. Pour cette classe, l'opérateur **>>** a été surchargé pour l'ensemble des types simples et les chaînes de caractères.

Lecture ***lecture.cpp***

```
#include <iostream>     // pour cin et cout
using namespace std;

int main()
{
    float temperature;
    float humidite;
    int pression;

    cout << "Saisir la température, l'humidité et la pression : " ;
    cin >> temperature >> humidite >> pression ;

    cout  << "la température est : " << temperature << " °"  << endl;
    cout  << "le taux d'humidité est : " << humidite << " %" << endl;
    cout  << "la pression est : " << pression <<  " hPa" << endl ;

    return 0;
}
```

La saisie des différentes données doit être séparée par un espace, une tabulation, ou un retour chariot. La dernière saisie est prise en compte après un retour chariot [Entrée].

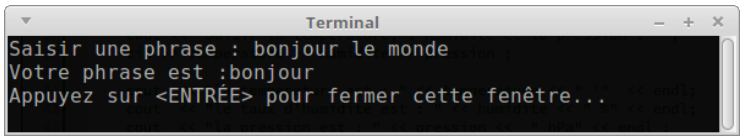


Cas particulier de la lecture des chaînes de caractères :

L'instance ***cin*** permet également la saisie de chaînes de caractères. Comme le montre l'exemple suivant, une difficulté se pose lorsque la chaîne est composée de plusieurs mots.

Lecture chaîne de caractères ***lecturemot.cpp***

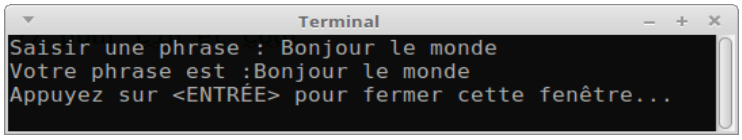
```
#include <iostream>     // pour cin et cout
using namespace std;

int main()
{
    char phrase[80+1];
    cout << "Saisir une phrase : " ;
    cin >> phrase ;
    cout  << "Votre phrase est :" << phrase << endl;

    return 0;
}
```



Seul le premier mot est mémorisé dans la variable. La solution passe par l'appel de la méthode ***getline*** de la classe ***istream***. Le deuxième paramètre tient compte de la longueur maximale de la chaîne pour éviter tout débordement.

Lecture chaîne de caractères avec la prise en compte des espaces ***lecturechaine.cpp***

```
#include <iostream>     // pour cin et cout
using namespace std;

int main()
{
    char phrase[80+1];

    cout << "Saisir une phrase : " ;
    cin.getline(phrase,80) ;

    cout  << "Votre phrase est :" << phrase << endl;

 return 0;
}
```



1.6.3. Les manipulateurs de flux d'entrée-sortie

Ces manipulateurs sont des objets dont l'insertion dans un flux en modifie le fonctionnement. Comme le montre l'exemple suivant, l'affichage de la valeur peut s'effectuer en décimal, ou en hexadécimal.

Exemple de manipulateurs ***manipulateurs.cpp***

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

int main()
{
    int val = 192;

    cout  << "Affichage par défaut     : " << val << endl;
    cout  << "Affichage en hexadécimal : " << hex << val << endl;
    cout  << "Affichage en décimal     : " << dec << val << endl;

    return 0;
}
```

Il existe deux types de manipulateurs, avec et sans argument.

Sans Argument applicable à istream et ostream :

Rôle	Manipulateur	Valeur par défaut
Affichage symbolique des booléens	boolalpha	
	noboolalpha	X
Affichage de la base autre que la base 10	showbase	
	noshowbase	X
Affichage du point décimal	showpoint	
	noshowpoint	X
Affichage du signe + devant les nombres positifs	Showpos	
	noshowpos	X
Affichage des caractères hexadécimaux en majuscule	uppercase	
	nouppercase	X
Affichage des nombres dans une base des bases : décimale, hexadécimale et octale (voir setbase)	dec	X
	hex	
	oct	
Mise en forme de l'affichage : aligné à gauche ou à droite	left	
	right	
Notation des nombres flottants	fixed	
	scientific	

Remarque

Lorsqu'un manipulateur a été envoyé sur le flux de sortie, le fonctionnement du flux reste inchangé jusqu'à la rencontre d'un prochain manipulateur qui contredit le premier. Par exemple, si l'affichage des nombres est demandé en hexadécimal avec le manipulateur hex, tous les nombres s'afficheront par la suite en hexadécimal jusqu'à l'utilisation du manipulateur dec ou oct.

Sans Argument applicable à ostream :

Rôle	Manipulateur
Écriture de la fin de ligne '\n' sur le flux de sortie et vidage du tampon	endl
Écriture de la fin de chaîne '\0' sur le flux de sortie	ends
Vidage du tampon de sortie	flush

Sans Argument applicable à istream :

Rôle	Manipulateur
Ignore les espaces précédant la lecture de données	ws
	skipws
Impose de ne pas saisir des espaces devant des données	noskipws

Manipulateur avec paramètres :

Ces manipulateurs demandent l'inclusion du fichier ***<iomanip>***. Ils sont utilisés principalement avec la classe ***ostream***.

Rôle	Manipulateur	Paramètre
Spécifier la largeur d'une zone	setw(int n)	Nombre de caractères
Spécification du caractère de remplissage	setfill(char c)	Caractères de remplissage, par défaut espace
Spécification de la base	setbase(int base)	La base est 8, 10, 16 équivalent à oct, dec, hex.
Spécification du nombre de chiffres significatifs	setprecision(int n)	Nombre de chiffres après la virgule

Exemple

Exemple de manipulateurs ***manipulateurs2.cpp***

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    int val = 192;
    float val2 = 3.141592654;
    bool sortie = true;
    cout  << "Affichage par défaut     : " << val << endl;
    cout  << "Affichage en hexadécimal : " << hex << val << endl;
    cout  << "Affichage en décimal     : " << dec << val << endl;
    cout << hex << val << " " << uppercase << val << " " ;
    cout << showbase  << val << nouppercase << " " << val << endl;
    cout << "+" << setfill('-') << setw(21) << "+" << setfill(' ') << endl;
    cout << "|" << setw(20) << left << "abc" << "|" << endl;
    cout << "|" << setw(20) << right << "abc" << "|" << endl;
    cout << "+" << setfill('-') << setw(21) << "+" << setfill(' ') << endl;
    cout << dec << sortie << " " << boolalpha << sortie << endl;
    cout << val2 << " " << fixed << val2 << " " << scientific << val2 << " ";
    cout << fixed << setprecision(2) << val2 << endl;
    cout << "Entrez un nombre en octal : " ;
    cin >> oct >> val ;
    cout << "vous avez saisie " << dec << val << " en décimal" << endl;
    return 0;
}
```

```
Terminal
Affichage par défaut     : 192
Affichage en hexadécimal : c0
Affichage en décimal     : 192
c0 C0  0XC0 0xc0
+--------------------+
|abc                 |
|                 abc|
+--------------------+
1 true
3.14159 3.141593 3.141593e+00 3.14
Entrez un nombre en octal : 770
vous avez saisie 504 en décimal
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

1.6.4. Les flux basés sur des fichiers

Par analogie, le C++ dispose d'une librairie ***<fstream>*** qui définit deux objets ***ifstream*** et ***ofstream*** permettant de connecter un flux à un fichier texte par défaut. Ce sont des entités du type ***FILE*** rencontrées en langage C avec la bibliothèque ***stdio***. Le premier désigne un flux en provenance d'un fichier, le second désigne un flux vers un fichier. L'objet ***fstream*** est également présent dans cette librairie, il est d'usage général et permet de faire toutes sortes d'entrées, sorties sur un fichier, quel que soit le type de données manipulées. Dans ce cas, il est nécessaire de préciser le mode d'ouverture du fichier ce qui est implicite avec les deux premiers.

Le constructeur de ***ifstream*** et celui de ***ofstream*** reçoivent en paramètre d'entrée le nom d'un fichier texte à ouvrir. Pour le premier objet, l'ouverture se fait en lecture et pour le second en écriture. Comme pour chaque appel système, il est nécessaire de vérifier la bonne ouverture du fichier avant de poursuivre les traitements. Dans le cas d'un fichier en lecture, cela permet de vérifier que le fichier existe bien et que le programme à bien le droit de lire ce fichier. Pour un fichier ouvert en écriture, cela permet de s'assurer que le répertoire où il va être stocké est accessible en écriture ou que le disque n'est pas rempli.

À noter également, lorsqu'un fichier est ouvert en écriture par défaut, si celui-ci existe déjà sur le disque, il est écrasé à moins de le préciser dans un second paramètre.

Présentation des différents modes d'ouverture contenus dans la classe ***fstream*** :

Les principaux modes d'ouverture	
app	(**app**end) ouvre le flux en ajout, la prochaine écriture se fera à la fin du fichier. Celui-ci n'est pas détruit.
ate	(**at e**nd) Définit l'indicateur de position de flux à la fin du fichier lors de l'ouverture, utilisé pour la lecture.
binary	Le flux donne accès à un fichier de type binaire plutôt que de type texte.
in	(**in**put) Ouverture du flux en lecture, mode par défaut pour **ifstream**.
out	(**out**put) Ouverture du flux en écriture, mode par défaut pour **ofstream**.

Utilisation de la valeur par défaut, le mode n'est pas précisé. Il est défini par le nom de la classe.

Exemple d'ouverture de fichier texte en lecture ***ouvrefichier.cpp***

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
    ifstream fichier("config.txt");
    if (!fichier.is_open())
        cerr << "Erreur lors de l'ouverture du fichier" << endl;
    else
    {
        // traitement sur le fichier ouvert en écriture
    }
    return 0 ;
}
```

Les drapeaux peuvent être spécifiés et combinés avec l'opérateur OU binaire : |.

Exemple d'ouverture de fichier texte en lecture et écriture ***fichierdrapeauxcombines.cpp***

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
    fstream fichier("exemple.txt", fstream::in | fstream::out);
    if (!fichier.is_open())
        cerr << "Erreur lors de l'ouverture du fichier" << endl;
    else
    {
        // traitement sur le fichier ouvert en lecture et écriture
    }
    return 0 ;
}
```

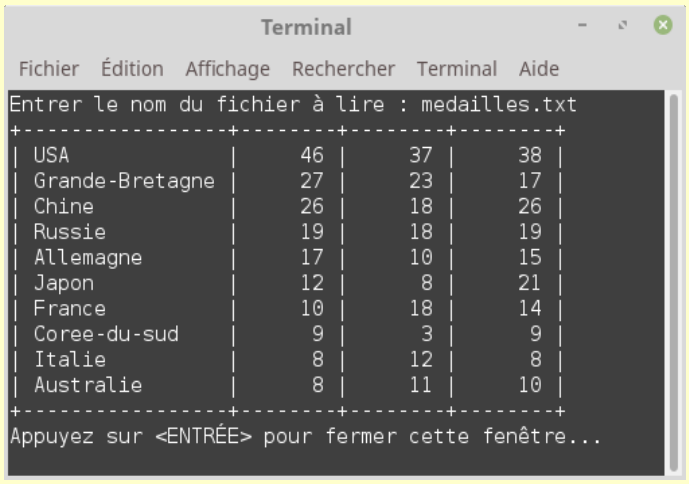
Chaque lecture dans un fichier demande vérification pour être certain que la lecture a bien été effectuée. En effet à la fin du fichier, la ou les variables recevant la donnée ne sont pas forcément affectées. De même pour tester si la fin de fichier est atteinte il est nécessaire de faire au moins une lecture. Typiquement, la lecture d'un fichier s'effectue de la manière suivante :

Exemple de traitement typique de lecture de fichier ***traitementfichier.cpp***

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string motCle ;
    // string remplace un tableau de caractères. la taille de la chaîne est dynamique
    int valeur;
    ifstream fichier("config.txt");
    if (!fichier.is_open())
        cerr << "Erreur lors de l'ouverture du fichier" << endl;
    else
    {
        do
        {
            // le fichier contient sur chaque ligne des couples mot clé + valeur
            fichier >> motCle >> valeur ;
            if (fichier.good())//Si les valeurs ont bien été lues
            {
                // traitement des variables motCle et valeur
            }
        } while(!fichier.eof());
    }
    return 0 ;
}
```

Exercice d'application

a) À partir du fichier texte, « medailles.txt », on souhaite obtenir le résultat :

medailles.txt	Résultat attendu
USA 46 37 38 Grande-Bretagne 27 23 17 Chine 26 18 26 Russie 19 18 19 Allemagne 17 10 15 Japon 12 8 21 France 10 18 14 Coree du Sud 9 3 9 Italie 8 12 8 Australie 8 11 10	

b) On propose ici un extrait de code à compléter.

Exercice d'application ***medailles.cpp***

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <string>

using namespace std;


int main() {
    string nomDuFichier;

    cout << "Entrer le nom du fichier à lire : ";
    cin >> nomDuFichier;

    //Création du flux en lecture sur le fichier
    ifstream leFichier(nomDuFichier.c_str()); // c_str() transforme string en char*

    if (!leFichier .is_open())
        cerr << "Erreur lors de l'ouverture du fichier" << endl;

    else
    {
        string pays;
        int nbOr;
        int nbArgent;
        int nbBronze;

        // A compléter, affichage de la première ligne du tableau
        do
        {
               //récupération des valeurs
            leFichier >> pays >> nbOr >> nbArgent >> nbBronze;

            if (leFichier.good())//Si les valeurs ont bien été lues
            {
                // A compléter, affichage de chaque ligne du tableau
            }
        } while (!leFichier.eof());
        // A compléter, affichage de la dernière ligne du tableau.
    }
    return 0;
}
```

Remarques

Le fichier **medailles.txt** doit se trouver dans le même répertoire que l'exécutable. Pour les autres cas, il est nécessaire de préciser le chemin complet.

c) Vous pouvez ne pas obtenir le bon résultat notamment à partir de l'affichage des informations pour la Corée du Sud, expliquer le problème, et modifier votre fichier texte en conséquence.

d) Écrire un programme qui produit le même résultat, mais dans un fichier texte au lieu de l'écran. Rappel, pour un fichier en écriture il est nécessaire d'inclure la librairie **ofstream**.

1.7. Références

Le C++ introduit une nouvelle notion, il est possible d'utiliser la référence d'une instance. Cette référence peut être assimilée à un alias, un autre nom pour désigner la même variable. Une référence ne peut être initialisée qu'une seule fois, lors de sa création. Toute autre modification affecte la variable référencée.

Exemple simple : déclaration, utilisation *reference.cpp*

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

int main()
{
    int val1 = 124;
    int val2 = 0;
    int resultat ;

    int &ref1 = val1; //affectation de la référence ref1 à la variable val1
    resultat = ref1 + 5;  // équivaut à :  resultat = val1 + 5;

    int &ref2 = val2 ; //affectation de la référence ref2 à la variable val2
    ref2 = val1 + 10; // équivaut à :  val2 = val1 + 10;

    cout << "La variable resultat vaut : " << resultat << endl;
    cout << "La variable val2 vaut     : " << val2 << endl;

    return 0;
}
```

```
Terminal
La variable resultat vaut : 129
La variable val2 vaut     : 134
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

La déclaration se fait en faisant précédé le nom par le symbole &. L'utilisation se fait comme une variable simple, il n'y a pas d'étoile comme avec les pointeurs pour désigner le contenu ou plus précisément l'objet pointé.

Une référence peut être considérée comme un pointeur constant, elle ne peut pas être réaffectée ou incrémentée. L'utilisation principale de ces références est le passage de paramètres aux fonctions. Le passage de paramètres par références permet l'échange de données entre fonctions **en entrée** et **en sortie** comme avec les pointeurs tout en gardant la simplicité de l'utilisation de paramètres par valeur.

Exercice d'application

Afin d'étudier le passage de paramètres dans les trois cas de figure, par valeur, par adresse et par référence, le programme suivant est proposé :

Exemple simple : déclaration, utilisation *appelsdefonctions.cpp*

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

int main()
{
    int a = 1;
    int b = -2;
    int c = 0;
    cout << "avant l'appel de Ajouter" << endl;
    cout << "a= " << a << endl;
    cout << "b= " << b << endl;
    cout << "c= " << c << endl;

    // Appel le la fonction Ajouter à compléter dans le tableau suivant le cas.

    cout << "après l'appel de Ajouter" << endl;
    cout << "a= " << a << endl;
    cout << "b= " << b << endl;
    cout << "c= " << c << endl;

    return 0;
}
```

1. À partir du programme précédent, on vous demande d'anticiper le résultat et de compléter l'appel de la fonction dans chacun des cas.

Fonctions	Appel de la fonction dans le programme ci-après	Avant	Après

void Ajouter(int a, int b ,int c)
{
c= a+b ;
}
	Ajouter(a, b, c);	a= 1
		b= -2
		c= 0
			a= 1
			b= -2
			c= 0

void Ajouter(int a, int b ,int *c)
{
*c= a+b ;
}
	Ajouter(a, b, &c);	a= 1
		b= -2
		c= 0
			a= 1
			b= -2
			c= -1

void Ajouter(int a, int b ,int &c)
{
c= a+b ;
}
	Ajouter(a, b, c);	a= 1
		b= -2
		c= 0
			a= 1
			b= -2
			c= -1

2. Comment se nomme chacun de ces trois passages de paramètres pour la variable c ?

De par sa simplicité d'utilisation, le passage de paramètre par référence est à privilégier en C++. Le nombre d'octets transmis lors de l'appel est comparable à celui transmis avec un pointeur, mais l'utilisation s'apparente à une simple variable passée par valeur.

Référence constante

Le compilateur C++ introduit la notion de paramètres constants, ***const***, qui permet d'avoir des paramètres uniquement en entrée. Ainsi, lorsqu'un paramètre transmis par référence est précédé du mot clé **const** à une fonction, celle-ci ne peut pas affecter de valeur à cette variable. Elle doit être utilisée uniquement en lecture, le compilateur se charge de cette vérification.

Exemple de fonction avec un paramètre constant

```
void Afficher(const string &chaine);
```

La fonction Afficher ne peut pas modifier la variable **chaine**, le mot clé **const** indique que le paramètre est uniquement en entrée pour cette fonction, bien qu'il soit passé par référence.

1.8. Constantes en C++

La notion de paramètre constant peut être généralisée à toutes les variables du C++. En effet, le C++ possède de véritables constantes, contrairement au langage C qui utilise le préprocesseur pour les définir.

Exemple de définition de constantes

```
#define NB_ELEMENTS 10  // Constante en C Attention il n'y a pas de ;
const int NB_ELEMENTS = 10; // définition d'une constante en C++


int tampon[NB_ELEMENTS];
```

Dans le premier cas, le préprocesseur remplace la chaîne de caractères par la valeur qui suit. Dans le cas du C++, la constante est véritablement typée, c'est bien ici un entier. Cette programmation est plus rigoureuse et, à privilégier en C++.

La réflexion peut être complétée par l'utilisation du mot clé ***const*** avec un pointeur, suivant la déclaration cela ne signifie pas la même chose en fonction de l'objectif voulu. Le souci est de toujours protéger les éléments pour plus de sécurité dans le code.

Déclaration	Signification	(ptr++)	((*ptr)++)
const char *ptr ;	ptr est un pointeur sur un caractère constant.	oui	non
int const *ptr ;	ptr est un pointeur constant sur un caractère.	non	oui
const int * const ptr ;	ptr est pointeur constant sur un caractère constant	non	non

1.9. Allocation et restitution de la mémoire

Le C++ introduit deux nouveaux opérateurs pour la gestion dynamique de la mémoire en remplacement des fonctions ***malloc***, ***calloc***, ***realloc*** et et ***free*** de la librairie standard du langage C. Il s'agit de l'opérateur ***new*** pour allouer de la mémoire et de l'opérateur ***delete*** pour la restituer.

L'opérateur s'utilise de deux manières en fonction de la nature de l'objet pour lequel on souhaite allouer cette mémoire.

- Allocation pour un objet unique : ***new*** *type* ;
- Allocation pour un tableau d'objet : ***new*** *type*[n] ;

Dans les deux cas, l'opérateur ***new*** retourne un pointeur sur un objet *type*. Le C++ offre l'avantage de pouvoir allouer dynamiquement de la mémoire pour un tableau dont la taille n'est pas connue avant la compilation, contrairement au langage C.

Exemples d'allocation

```
int *ptr;       // déclaration du pointeur
ptr = new int ; // allocation mémoire pour un entier

int *tab;
tab = new int [10]; // allocation mémoire pour un tableau de 10 entiers
```

La restitution de la mémoire dynamique est réalisée par l'opérateur ***delete*** dans le premier cas et ***delete [ ]*** dans le second. En effet pour ce dernier cas, l'opérateur ***delete [ ]*** libère chacune des cases du tableau. Il est impératif d'utiliser la forme appropriée pour libérer la mémoire, sous peine de résultats imprévisibles.

Exemples de restitution

```
delete ptr;
delete [] tab; // tab a été alloué avec l'opérateur new int []
```

1.10. Traitement des erreurs : Les exceptions

1.10.1. Introduction

Un programme, même testé et réputé sans erreur, peut être amené à s'arrêter dans des circonstances autres que celles prévues par le programmeur. Il existe différentes sources d'erreurs :

- les erreurs détectées à la compilation, elles doivent être corrigées avant l'exécution
- les erreurs détectées et corrigées à l'exécution,
- les erreurs détectées et non traitées à l'exécution,
- les erreurs non détectées à l'exécution.

L'origine de ces erreurs peut-être multiple :

- les erreurs d'entrée/sortie, l'utilisateur fait une faute dans la saisie d'une donnée, un fichier est corrompu ou inaccessible,
- les erreurs matérielles, l'imprimante est déconnectée, le disque est saturé, la connexion réseau est défaillante,
- les erreurs de programmation, l'indice d'un tableau est invalide, emploie d'une variable non initialisée.

1.10.2. Traitement des erreurs

En présence d'une erreur, un programme se comporte selon trois cas de figure :

- **Idéalement** : il permet de revenir à un état défini et permet de poursuivre ainsi l'exécution.
- **Honnêtement** : il avertit l'utilisateur et permet une sortie correcte après une sauvegarde de l'ensemble du travail en cours.
- **Lamentablement** : il s'interrompt brusquement et affiche au mieux un message d'erreur peu clair.

Une solution consiste à vérifier à l'aide de structure de contrôle tous les cas de dysfonctionnement possibles. Toutefois, cette solution alourdit considérablement le code en cas de contrôle successif. En langage C, le traitement d'erreur respecte certaines conventions de programmations :

- des valeurs de retours spéciales, un programme qui se finit bien retourne 0 ou plutôt **EXIT_SUCCESS** sinon, il retourne **EXIT_FAILURE** deux constantes définies dans **<sdtlib.h>**.
- Le positionnement de drapeaux, par exemple la variable **errno** définie dans **<errno.h>** contient un code d'erreur indiquant la nature de l'erreur.

Exemples de traitement d'erreur en C *erreur.c*

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

int main(void)
{
    int x;
    int y;
    int sortie = EXIT_SUCCESS;

    printf("Entrez deux nombres : ");
    if (scanf("%d %d", &x, &y) != 2)
    {
        printf("Vous deviez saisir deux nombres !\n");
        sortie = EXIT_FAILURE;
    }
    else
    {
        printf("Vous avez saisi : %d et %d\n", x, y);
    }
    return sortie;
}
```

Qui vérifie systématiquement que la saisie s'est bien déroulée ?
Il n'y a aucune garantie que les conventions soient suivies par tout le monde

En C++, un autre mécanisme a été mis en place, la gestion des exceptions. Si un problème survient, une exception est levée. Toute exception peut et doit être attrapée dans le bloc ***try***. Le traitement des exceptions est placé en dehors de la séquence normale, dans le bloc ***catch***. Un premier exemple montre la structure ***try … catch*** que nous détaillerons de manière plus approfondie dans le prochain chapitre sur les classes.

Exemples de traitement avec les exceptions *erreur.cpp*

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

int main()
{
    int nombre1, nombre2 ;
    try
    {
        cout << "Entrez deux nombres entiers : ";
        cin >> nombre1 >> nombre2 ;
        if(nombre2 == 0)
            throw string("Erreur de division par ZÉRO !");
        else
            cout << nombre1 / nombre2 << endl;
    }

    catch(string const& chaine)
    {
        cerr << chaine << endl;
    }
    return 0;
}
```

1.11. Conversion de types

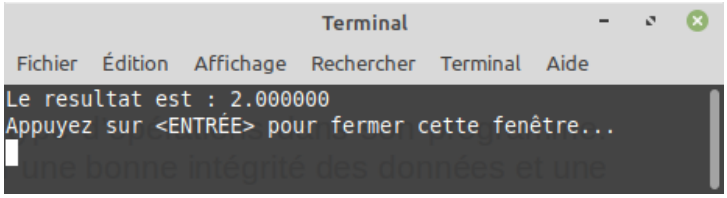
Le langage C++ tout comme le langage C à son époque sont des langages avec un **typage fort**. Il impose des restrictions sur la façon dont les différents types de variables peuvent être convertis les uns aux autres sans aucune instruction de conversion. Le compilateur interdit tout **transtypage**, autre terme utilisé, qui n'a aucun sens et avertit en cas de perte de précision ou risque de mauvaise interprétation de l'information.

Un programmeur peut cependant être amené à utiliser explicitement ce type d'opérations dans son programme. Dans ce cas, il faut le faire à bon escient et ne pas en abuser pour garantir une bonne intégrité des données et une meilleure maintenabilité du code.

Dans l'exemple suivant, on souhaite que le résultat de la division de deux entiers donne un résultat sous la forme d'un réel.

Exemple en langage C

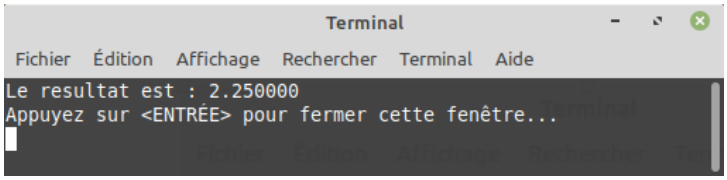
```c
#include <stdio.h>
int main()
{
    int val1 = 9;
    int val2 = 4;
    float resultat;
    resultat = val1/val2;
    printf("Le resultat est : %f\n",resultat);
    return 0;
}
```



Sans opérateur de conversion, le résultat n'est pas celui attendu.
Avec l'opérateur de conversion **(<type>) <expression>** le résultat est tout autre

Exemple en langage C avec l'opérateur de conversion

```c
#include <stdio.h>
int main()
{
    int val1 = 9;
    int val2 = 4;
    float resultat;
    resultat = val1/(float)val2;
    printf("Le resultat est : %f\n",resultat);
    return 0;
}
```

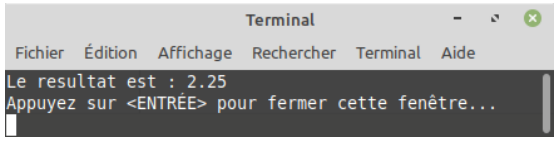


Le langage C++ distingue différentes situations de transtypage et utilise pour cela une syntaxe différente du langage C.

Le cas présent correspond au cas le plus simple, son rôle est d'expliciter les conversions implicites. L'opérateur est dans ce cas **static_cast<type> (expression)**. La conversion de type statique est toujours réalisée sans vérification. L'usage en est donc fait en toute conscience.

Exemple en langage C++ avec l'opérateur de conversion static_cast<>

```c
#include <iostream>
using namespace std;
int main()
{
    int val1 = 9;
    int val2 = 4;
    float resultat;
    resultat = val1/static_cast<float>(val2);
    cout << "Le resultat est : " << resultat << endl;
    return 0;
}
```



D'autres cas de conversion, impliquant une autre syntaxe, seront décrits dans ce document lorsque de nouvelles notions auront été abordées.