Langage C++

Table des matières

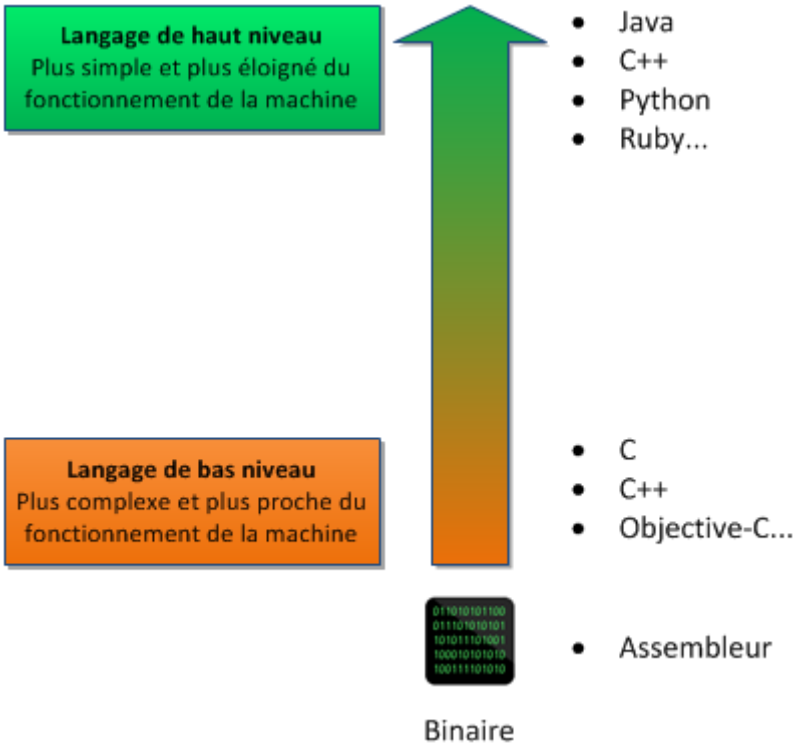
Le C++ est un langage de programmation orienté objet multi paradigmes. Descendant du langage C, il est très apprécié pour ses performances et ses possibilités.

1. Premiers pas

1.1. Introduction

Le C++ est le descendant du langage C. Ces deux langages, bien que semblables au premier abord, sont néanmoins différents. Le C++ propose de nouvelles fonctionnalités, comme la programmation orientée objet (POO). Elles en font un langage très puissant qui permet de programmer avec une approche différente du langage C.

Créé en 1983, le langage C++ continu à être amélioré. Très rapide et portable, il peut être considéré à la fois comme un langage de bas niveau, proche de l'assembleur, et de haut niveau.



1.2. Mon premier programme

Nous allons écrire un programme en mode console qui affichera « Hello world! ».

```
#include <iostream>     // Inclut la bibliothèque iostream (affichage de texte)
using namespace std;    // Indique quel espace de noms on va utiliser

/*
Fonction principale "main"
Tous les programmes commencent par la fonction main
*/
int main()
{
    cout << "Hello world!" << endl; // Affiche un message
    return 0;                       // Termine la fonction main
}
```

La première ligne qui commence par # est une directive de préprocesseur qui est lue par un

programme appelé préprocesseur, un programme qui se lance au début de la compilation.

#include <iostream>
Cette ligne demande d'inclure ce fichier, ce qui nous permet d'utiliser une bibliothèque… d'affichage de messages à l'écran dans une console !

Ce fichier fait partie des fichiers source tout prêts qu'on les appelle bibliothèques qui contiennent la plupart des fonctions de base dont a besoin un programme.

using namespace std;

Si on charge plusieurs bibliothèques, chacune va proposer de nombreuses fonctionnalités. Parfois, certaines fonctionnalités ont le même nom. Pour éviter tout conflit, la ligne using namespace std; indique que l'on utilise l'espace de noms std dans la suite du fichier de code. Cet espace de noms est un des plus connus car il correspond à la bibliothèque standard (std), une bibliothèque livrée par défaut avec le langage C++ et dont iostream fait partie.

int main()

main est la fonction principale du programme, c'est toujours par la fonction main que le programme commence. C'est la seule qui soit obligatoire, aucun programme ne peut être compilé sans elle.

Une fonction a un début et une fin, délimités par des accolades { et } et retourne ou non une valeur. Toute la fonction main se trouve donc entre ces accolades.

Les lignes à l'intérieur d'une fonction s'appellent des instructions.

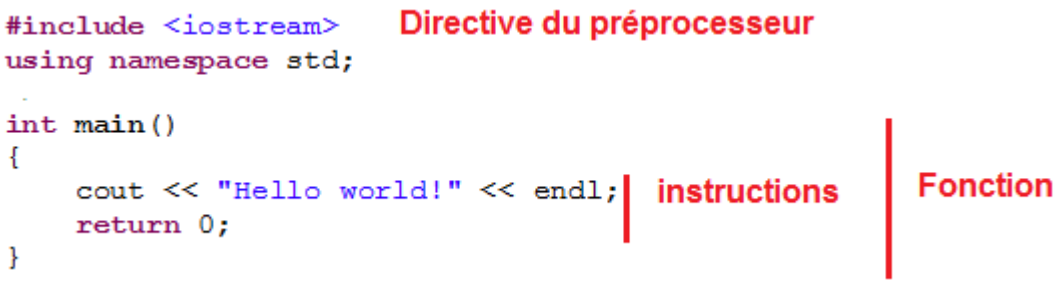
Toute instruction se termine obligatoirement par un point-virgule « ; ».

cout << "Hello world!" << endl;

Le rôle de cout est d'afficher un message à l'écran. Notez que cout est fourni par iostream. L'instruction endl crée un retour à la ligne dans la console.

return 0; la fonction main retourne la valeur 0 de type entière (int).

En pratique, 0 signifie « tout s'est bien passé » et n'importe quelle autre valeur signifie « erreur ».



1.3. Les commentaires

Identique au langage C.

2. Les variables

2.1. Déclarer une variable

Il faut indiquer le type de la variable, son nom et sa valeur. Pour ce faire, il y a deux méthode :

- type nom = valeur ; // comme en langage C
- type nom(valeur) ; // propre au langage C++ qui utilise des objets

Ces deux versions sont strictement équivalentes.

Les types de variables sont identiques au langage C à deux exceptions près :

- Le type bool qui ne peut contenir que deux valeurs : true (vrai) ou false (faux), sans guillemets.

Les constantes sont des variables particulières dont la valeur reste constante.

- Le type string qui permet de gérer les chaînes de caractères (string n'est pas un type de variable, mais un objet).

2.2. Les constantes

Pour déclarer une constante, il faut utiliser le mot const juste devant le type de la variable et lui donner obligatoirement une valeur au moment de sa déclaration.

Ex : const float PI(3.14159265359);

Par convention on écrit les noms des constantes entièrement en majuscules afin de distinguer facilement les constantes des variables.

2.3. Afficher le contenu d'une variable

Il faut utiliser cout et les chevrons (<<). À la place du texte à afficher, on met simplement le nom de la variable.

Exemple :

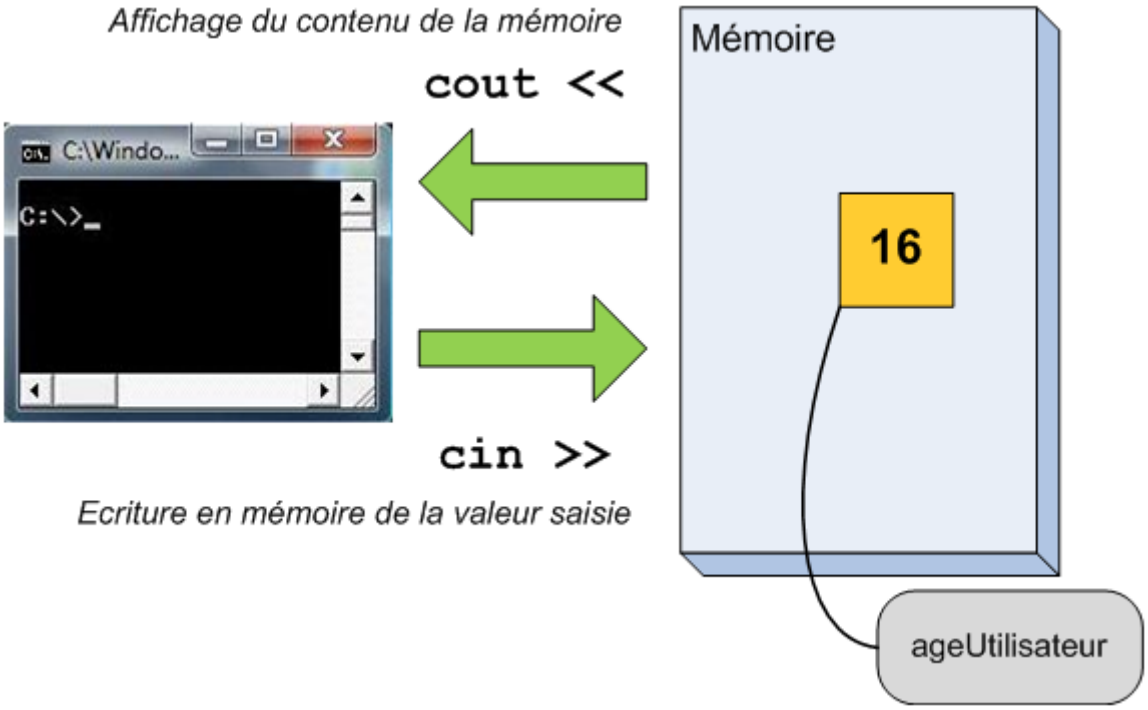
```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    int     qiUtilisateur(150);
    string  nomUtilisateur("Albert Einstein");

    cout << "Vous vous appelez " << nomUtilisateur << " et votre QI vaut " <<
qiUtilisateur << endl;
    return 0;
}
```

2.4. Récupérer une saisie

On utilise cin pour faire entrer des informations dans le programme et les chevrons (>>).



Exemple :

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

int main()
{
    cout << "Quel est votre prenom ?" << endl;
    //On crée une case mémoire pour contenir une chaine de caractères
    string nomUtilisateur("Sans nom");
    //On remplit cette case avec ce qu'écrit l'utilisateur
    cin >> nomUtilisateur;

    cout << "Combien vaut pi ?" << endl;
    //On crée une case mémoire pour stocker un nombre réel
    double piUtilisateur(-1.);
    //Et on remplit cette case avec ce qu'écrit l'utilisateur
    cin >> piUtilisateur;

    cout << "Vous vous appelez " << nomUtilisateur << " et vous pensez que
pivaut " << piUtilisateur << "." << endl;

    return 0;
}
```

Remarque : s'il y a un espace dans la chaîne de caractères à saisir, l'ordinateur va couper après le premier mot. Et la valeur de pi sera fausse !

Il faut récupérer toute la ligne plutôt que seulement le premier mot en utilisant la fonction getline(). Nous verrons plus loin ce que sont exactement les fonctions mais, pour l'instant, voyons comment faire dans ce cas particulier.

Il faut remplacer la ligne cin >> nomUtilisateur; par un getline().

Cependant, si l'on utilise d'abord cin >> puis getline(), pour demander la valeur de pi avant de demander le nom, l'ordinateur ne demande pas son nom à l'utilisateur et affiche n'importe quoi.

Pour pallier ce problème, il faut ajouter la ligne cin.ignore() après l'utilisation des chevrons :

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

int main()
{
    cout << "Combien vaut pi ?" << endl;
    //On crée une case mémoire pour stocker un nombre réel
    double piUtilisateur(-1.);
    //Et on remplit cette case avec ce qu'écrit l'utilisateur
    cin >> piUtilisateur;

    cin.ignore();

    cout << "Quel est votre nom ?" << endl;
    //On crée une case mémoire pour contenir une chaine de caractères
    string nomUtilisateur("Sans nom");
    //On remplit cette case avec toute la ligne que l'utilisateur a écrit
    getline(cin, nomUtilisateur);

    cout << "Vous vous appelez " << nomUtilisateur << " et vous pensez que
pivaut " << piUtilisateur << "." << endl;

    return 0;
}
```

3. Les opérations de base

Identique au langage C.

3.1. Raccourcis d'écriture

Identique au langage C.

3.2. La bibliothèque mathématique

En C++, il existe ce qu'on appelle des bibliothèques « standard », c'est-à-dire des bibliothèques toujours utilisables. Ce sont en quelque sorte des bibliothèques « de base » qu'on utilise très souvent.

La bibliothèque cmath contient de nombreuses fonctions mathématiques toutes prêtes.

Pour pouvoir utiliser les fonctions de la bibliothèque mathématique, il est indispensable de mettre la directive de préprocesseur `#include <cmath>`.

Voici quelques fonctions principales :

fonction	**description**
fabs[1]	Cette fonction retourne la valeur absolue d'un nombre flottant, c'est-à-dire |x|.

1 Équivalent à la fonction abs de la librairie standard stdlib.h (ne fonctionne que sur des entiers)

ceil	Cette fonction renvoie le premier nombre entier après le nombre décimal qu'on lui donne.
floor	C'est l'inverse de la fonction précédente : elle renvoie le nombre directement en dessous.
pow	Cette fonction permet de calculer la puissance d'un nombre.
sqrt	Cette fonction calcule la racine carrée d'un nombre.
sin, cos, tan	Ce sont les trois fonctions sinus, cosinus et tangente utilisées en trigonométrie. Ces fonctions attendent une valeur en radians.
asin, acos, atan	Ce sont les fonctions arc sinus, arc cosinus et arc tangente utilisées en trigonométrie.
exp	Cette fonction calcule l'exponentielle d'un nombre.
log	Cette fonction calcule le logarithme népérien d'un nombre.
log10	Cette fonction calcule le logarithme base 10 d'un nombre.

4. Les conditions

Identique au langage C.

4.1. Conditions imbriquées

Il est évidemment possible d'imbriquées des conditions les unes dans les autres :

```
#include <iostream>

using namespace std;

int main()
{
    string jour("mercredi");

    if ( jour == "lundi" || jour == "mardi" )          // début de la semaine
        cout << "courage !!!";
    else if ( jour == "mercredi" )                     // milieu de la semaine
        cout << "c'est le jour des enfants";
    else if ( jour == "jeudi" || jour == "vendredi" ) // fin de la semaine
        cout << "bientôt le we !";
    else    // il faut traiter les autres jours (cas par défaut)
        cout << "vive le week end !";

    return 0;
}
```

5. Les boucles

Identique au langage C.

6. Les fonctions

Identique au langage C.

6.1. Passage par valeur et par référence

Lorsqu'une variable est passée par valeur à une fonction, celle-ci est copiée dans une nouvelle cellule mémoire. On dit que l'argument a est passé par valeur. Quand la fonction est terminée, la mémoire est libérée et la valeur qui y avait été stockée est perdue : une variable passée en paramètre ne change pas de valeur.

Lorsqu'une variable est passée par référence, celle-ci n'est pas copiée : la fonction utilise directement la variable. Une variable passée en paramètre peut changer de valeur.

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

void echange(double& a, double& b)
{
	double temporaire(a);	//On sauvegarde la valeur de 'a'
	a = b;			//On remplace la valeur de 'a' par celle de 'b'
	b = temporaire;		//Et on utilise la valeur sauvegardée pour mettre
l'ancienne valeur de 'a' dans 'b'
}

int main()
{
	double a(1.2), b(4.5);

	cout << "a vaut " << a << " et b vaut " << b << endl;
	echange(a,b); //On utilise la fonction
	cout << "a vaut " << a << " et b vaut " << b << endl;

	return 0;
}
```

Le passage par référence offre un gros avantage sur le passage par valeur : aucune copie n'est effectuée. Si une fonction reçoit en argument un string contenant un très long texte, alors la copier va prendre du temps et va affecter les performances du programme.

Cependant, pour éviter une modification éventuelle de l'argument on utilise un passage par référence constante :

```
void ma_fonction(string const& texte)
//Pas de copie et pas de modification possible
{
	// instructions
}
```

6.2. Valeurs par défaut

On peut donner des valeurs par défaut à certains paramètres des fonctions pour ne pas être obligé d'indiquer à chaque fois tous les paramètres :

```
#include <iostream>

using namespace std;

// Définition de la fonction
int nombreDeSecondes(int heures = 0, int minutes = 0, int secondes = 0)
{
    int total(0);
```

```
    total  = heures * 60 * 60;
    total += minutes * 60;
    total += secondes;

    return total;
}

// Main
int main()
{
    cout << nombreDeSecondes(1, 10, 25) << endl;
    cout << nombreDeSecondes(1, 10) << endl;
    cout << nombreDeSecondes(1) << endl;
    cout << nombreDeSecondes() << endl;

    return 0;
}
```

7. Les pointeurs

Identique au langage C. On préfèra plutôt parler de référence.

Exemple :

```
#include <iostream>

using namespace std;

int main()
{
    int     ageUtilisateur(18);           //Une variable pour contenir l'âge

    int&    maReference(ageUtilisateur);  //Et une référence sur la variable

    //On peut utiliser à partir d'ici
    //'ageUtilisateur' ou 'maReference' indistinctement
    //Puisque ce sont deux étiquettes de la même case en mémoire

    //On affiche, de la manière habituelle
    cout << "Vous avez " << ageUtilisateur << "  ans. (via variable)" << endl;

    //Et on affiche en utilisant la référence
    cout << "Vous avez " << maReference << " ans. (via reference)" << endl;

    return 0;
}
```

8. Les tableaux

Identique au langage C.

8.1. Tableau dynamique

Un tableau dynamique est un objet vector. Il faut ajouter la ligne #include <vector> pour utiliser ces tableaux.

On utilise la syntaxe suivante pour le déclarer : vector<type> nom(taille);

Exemple :

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

int main()
{
   int const nombreNotes(5);
   vector<int> tableau(nombreNotes, 3);  //Crée un tableau de 5 entiers valant
tous 3

   for(int i(0); i<nombreNotes; i++)
      cout << tableau[i] << endl;
}
```

Pour changer la taille d'un tableau de façon dynamique, il faut utiliser la fonction push_back() :

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

int main()
{
   vector<double> notes;       //Un tableau vide

   notes.push_back(12.5);      //On ajoute des valeurs
   notes.push_back(19.5);
   notes.push_back(6);

   double moyenne(0);
   for(int i(0); i<notes.size(); ++i)
   //On utilise notes.size() pour la limite de notre boucle
   {
      moyenne += notes[i];   //On additionne toutes les notes
   }

   moyenne /= notes.size();
   //On utilise à nouveau notes.size() pour obtenir le nombre de notes

   cout << "La moyenne est : " << moyenne << endl;

   return 0;
}
```

Pour des raisons d'efficacité, on préférera passer un tableau par référence à une fonction :

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

//Une fonction recevant un tableau d'entiers en argument
void fonction(vector<int>& tab)
{
    //…
}

int main()
```

```
{
	vector<int> tableau(3,2);	//On crée un tableau de 3 entiers valant 2
	fonction(tableau);		//On passe le tableau à la fonction

	return 0;
}
```

Si ce tableau ne doit pas être modifié, on fera un passage par référence constante :

```
void fonction(vector<int> const& tableau)
```

Remarque : si vous utilisez une fonction prototype (définition d'une fonction dans un fichier d'en-tête .h), il faudra spécifier l'espace de nom.

```
void fonction(std::vector<int> const&);
```

Notez qu'il est aussi possible de créer des tableaux multi-dimensionnels de taille variable en utilisant les vector. Pour une grille 2D d'entiers, on devra écrire:

```
vector<vector<int> > grille;
```

Le problème est que ce n'est pas réellement un tableau 2D, mais plutôt un tableau de lignes. Il faudra donc commencer par ajouter des lignes à notre tableau.

```
//On ajoute une ligne de 5 cellules au tableau
grille.push_back(vector<int>(5));
//On ajoute une ligne de 3 cellules contenant chacune le nombre 4 au tableau
grille.push_back(vector<int>(3,4));
```

Chaque ligne peut donc avoir une longueur différente. On peut accéder à une ligne en utilisant les crochets :

```
//Ajoute une case contenant 8 à la première ligne du tableau
grille[0].push_back(8);
```

Finalement, on peut accéder aux valeurs dans les cases de la grille en utilisant deux paires de crochets, comme pour les tableaux statiques. Il faut par contre s'assurer que cette ligne et cette colonne existent réellement :

```
grille[2][3] = 9;	//Change la valeur de la cellule (2,3) de la grille
```

9. Les objets

Les objets sont des variables plus complexes qui peuvent être constitués d'une ou plusieurs autres variables (attributs) et d'une ou de plusieurs fonctions (méthodes). Ainsi, un objet peut lui-même contenir un objet ! Un objet peut représenter absolument ce que l'on veut : une chaise, une voiture, un concept philosophique, une formule mathématique, etc.

Pour cela, il faut déclarer ce qu'on appelle une classe et lui donner un nom.

Exemple :

```
// déclaration de la classe Voiture
class Voiture {
	string	marque;		// marque de la voiture
	int	roue(4);	// le nombre de roues
	int	place;		// le nombre de places passagers
	float	vitesse;	// On ne connaît pas la vitesse
	float	maxi;		// vitesse maxi pour régulateur
	Couleur	carrosserie;	// la couleur est représentée par une classe
```

```
	bool regulateurActive() {
		return maxi == 0;
		}
	void acceler(float consigne) {
		if ( maxi )
			vitesse = ( consigne > maxi ) ? consigne : maxi ;
		else
			vitesse = consigne;
		}
	};	// la déclaration d'une variable se termine par ;
```

Dans le programme principal, on construit une voiture avec cette syntaxe : Voiture v;

Construire un objet s'appelle l'instanciation.

```
int main()
{
	Voiture v;		//Instanciation de l'objet voiture

	v.accelerer(45.0);

	return 0;
}
```

Cependant, la classe qui a été déclarée provoquera une erreur de compilation à cause des droits d'accès sur les attributs et les méthodes de la classe. Pour contrôler les capacités des classes à utiliser les attributs et méthodes les unes des autres, il existe à trois niveaux d'accessibilité :

- public, pour qu'un attribut ou une méthode soit accessible à tous.
- protected, pour que les éléments ne soient accessibles qu'aux classes filles.
- private, pour que les éléments ne soient accessibles à personne si ce n'est la classe elle-même.

Par défaut, tous les éléments d'une classe sont privés. Pour utiliser une méthode, il faut la déclarer publique :

```
// déclaration de la classe Voiture
class Voiture {
	// Tout ce qui suit est privé (inaccessible depuis l'extérieur)
	private:

	string	marque;		// marque de la voiture
	int	roue(4);	// le nombre de roues
	int	place;		// le nombre de places passagers
	float	vitesse;	// On ne connaît pas la vitesse
	float	maxi;		// vitesse maxi pour régulateur
	Couleur	carrosserie;	// la couleur est représentée par une classe

	// Tout ce qui suit est public (accessible depuis l'extérieur)
	public:

	bool regulateurActive() {
		return maxi == 0;
		}
	void acceler(float consigne) {
		if ( maxi )
			vitesse = ( consigne > maxi ) ? consigne : maxi ;
		else
```

```
                vitesse = consigne;
            }
        };
```

Cependant, même s'il est possible de déclarer les attributs comme public, en POO[2] les attributs d'une classe doivent être encapsulés : inaccessible depuis l'extérieur de la classe.

Pour lire ou initialiser un attribut à l'extérieur de la classe, il faut créer un accesseur : c'est une méthode qui permettra d'accéder à l'attribut privé en dehors de la classe. Par convention, les accesseur qui récupère la valeur d'un attribut commencent par set, ceux qui fixent une valeur par set :

```
float getVitesse() const {
      return vitesse;
      }
void setVitesse(float v) {
      vitesse = v;
      }
```

9.1. Constructeurs et destructeurs

Parmi les différents types de méthode, il existe un type particulier : les constructeurs. Ces constructeurs sont des méthodes qui construisent l'objet désigné par la classe en initialisant les attributs. Un constructeur porte le nom de la classe.

À contrario, un destructeur est une méthode appelée lorsque l'objet est supprimé de la mémoire. Son principal rôle est de désallouer la mémoire (via delete) qui a été allouée dynamiquement. Un destructeur porte le nom de la classe précédé de ~. Un destructeur ne peut prendre aucun paramètre et ne peut pas être surchargé.

Exemple :

```
// déclaration de la classe Voiture
class Voiture {
      // Tout ce qui suit est privé (inaccessible depuis l'extérieur)
      private:

      string      marque;           // marque de la voiture
      int         roue(4);          // le nombre de roues
      int         place;            // le nombre de places passagers
      float       vitesse;          // On ne connaît pas la vitesse
      float       maxi;             // vitesse maxi pour régulateur
      Couleur     carrosserie;      // la couleur est représentée par une classe

      // Tout ce qui suit est public (accessible depuis l'extérieur)
      public:

      // Ce constructeur ne prend pas de paramètre
      Voiture() {
            // Quand on construit une voiture
            vitesse(0);       // elle a une vitesse nulle
            maxi(0);          // et le régulateur n'est pas activé
            }
      // Ce destructeur est à priori inutile dans ce cas
      ~Voiture() {
            // ...
            }
```

2 Programmation Orienté Objet

```
bool regulateurActive() {
      return maxi == 0;
      }
void acceler(float consigne) {
      if ( maxi )
            vitesse = ( consigne > maxi ) ? maxi : consigne ;
      else
            vitesse = consigne;
      }
};
```

9.2. La surcharge de méthode

La surcharge de méthode consiste à garder le nom d'une méthode (donc un type de traitement à faire) et à changer la liste ou le type de ses paramètres.

Comme le constructeur est une méthode, on a le droit de le surcharger lui aussi afin de créer un objet de plusieurs façons différentes :

```
Voiture();  // Constructeur par défaut
Voiture(int place);
Voiture(string marque, int place);
```

Le compilateur crée automatiquement un constructeur par défaut et un "constructeur de copie". C'est une surcharge du constructeur qui initialise l'objet en copiant les valeurs des attributs de l'autre objet.

Pour obtenir une copie conforme de Voiture, il suffit d'écrire :

```
//On crée testarossa en utilisant un constructeur normal
Voiture testarossa("Ferrari", 2);
//On crée spider en copiant tous les attributs de  testarossa
Voiture spider(testarossa);
```

Pour changer le comportement du constructeur de copie, il faut simplement le déclarer dans la classe de la manière suivante :

```
Voiture(Voiture const& autre);
```

9.3. Les méthodes constantes

Les méthodes constantes sont des méthodes de « lecture seule ». Elles possèdent le mot-clé const à la fin de leur prototype et de leur déclaration. Les méthodes constantes ne modifie la valeur d'aucun attributs de la classe.

Pour créer l'objet, il va falloir le créer par le biais d'une allocation dynamique avec new. Sinon, l'objet ne se créera pas tout seul.

```
bool regulateurActive() const {
      return maxi == 0;
      }
```

9.4. Classes et pointeurs

Dans certains cas, il peut être judicieux d'utiliser des pointeurs pour économiser de la mémoire ou optimiser la rapidité d'un programme. Pour utiliser un pointeur sur une classe, il suffit de déclarer la classe en la faisant précéder d'une étoile « * ».

En utilisant un pointeur, l'objet n'est plus contenu dans la classe.

Par sécurité, on initialise le pointeur à 0 dans la partie déclarative des attributs. Le rôle du constructeur étant de faire en sorte que l'objet soit bien construit (donc que tous les pointeurs pointent vers quelque chose), on fera une allocation dynamique avec l'instruction **new** :

```
// déclaration de la classe Voiture
class Voiture {
      // Tout ce qui suit est privé (inaccessible depuis l'extérieur)
      private:

      string      marque;           // marque de la voiture
      int         roue(4);          // le nombre de roues
      int         place;            // le nombre de places passagers
      float       vitesse;          // On ne connaît pas la vitesse
      float       maxi;             // vitesse maxi pour régulateur
      Couleur     *carrosserie(0);  // la couleur est représentée par une classe

      // Tout ce qui suit est public (accessible depuis l'extérieur)
      public:

      // Ce constructeur ne prend pas de paramètre
      Voiture() {
            // Quand on construit une voiture
            vitesse(0);       // elle a une vitesse nulle
            maxi(0);          // et le régulateur n'est pas activé
            carrosserie = new Couleur();  // on initialise le pointeur
            }
      // Ce destructeur est nécessaire dans ce cas
      ~Voiture() {
            // si l'objet Couleur a été créé on le détruit
            if ( carrosserie )
                  delete carrosserie;      // appel au destructeur de Couleur
            }

      };
```

L'objet carrosserie étant un pointeur, lorsque l'objet Voiture est supprimé, l'objet de type Couleur subsiste en mémoire. Un objet Couleur va traîner en mémoire : c'est ce qu'on appelle une fuite de mémoire.

Pour résoudre ce problème, il faut faire un **delete** du pointeur dans le destructeur pour que l'objet Couleur soit supprimé avant l'objet Voiture.

NB : l'objet carrosserie étant maintenant un pointeur, il faudra appel aux méthodes de l'objet Couleur non plus avec le point « . » mais avec la flèche « -> ». Exemple :

```
string *getCouleur() const {
      return carrosserie->getCouleur();
      }
```

9.5. L'héritage

Il y a héritage quand on peut dire : « A est un B » :

- une voiture est un véhicule (Voiture hérite de Vehicule)
- une moto est un véhicule (Bus hérite de Vehicule)
- un moineau est un oiseau (Moineau hérite d'Oiseau)
- …

Il existe certains objets dont l'instanciation n'aurait aucun sens. Par exemple, un objet de type Véhicule n'existe pas vraiment dans un jeu de course. En revanche il est possible d'avoir des véhicules de certains types, par exemple des voitures ou des motos. Une moto doit avoir deux roues et une voiture doit en avoir 4 : dans les deux cas elles ont des roues. Dans les cas de ce genre, c'est-à-dire quand plusieurs classes ont des attributs en commun, on fait appel à l'héritage.

Quand une classe A hérite d'une classe B, on dit que la classe A est la « fille » de la classe B et que la classe B est le « parent » (ou la classe « mère ») de la classe A. De cette façon, la classe A contiendra de base tous les attributs et toutes les méthodes de la classe B à l'exception des constructeurs et des destructeurs.

Exemple :

```
// Classe qui ne peut pas être instanciée
class Vehicule {
      protected:  // attributs uniquement accessibles aux classes filles

      int   prix_achat;
      int   nombre_de_roues;
      int   puissance_moteur;

      public:

      Vehicule(int roue) {
            nombre_de_roues = roue;
            }
      virtual ~Vehicule() {
            // même si le destructeur ne fait rien il faut le mettre
            }
      void puissance() const {            // Affiche la puissance
            cout << "La puissance est de " << puissance_moteur << " ch." << endl;
            }
      virtual void motorisation() const { // Affiche le type de motorisation
            cout << "Ce véhicule roule à l'essence." << endl;
            }
      virtual void prix() const = 0;      // Affiche le prix du Vehicule
      };

// Une Voiture est un Vehicule
class Voiture : public Vehicule {   // les : indiquent l'héritage
      bool  est_un_diesel;
      int   nombre_de_portes;

      public:

      // Quand on construit une voiture...
      // il faut appeler le constructeur de la classe mère
      Voiture(bool diesel) : Vehicule(4) {      // elle a 4 roues
            est_un_diesel(diesel);  // on définit sa motorisation
            }
      void motorisation() const {         // Affiche le type de motorisation
            if ( est_un_diesel )
                  cout << "Ce véhicule roule au diesel." << endl;
            else
                  Vehicule::motorisation();     // appel à la classe mère
            }
      void prix() const {                 // Affiche le prix du Vehicule
```

```
            cout << "Cette voiture coûte " << prix_achat << "€." << endl;
            }
      };

// Une Moto est aussi un Vehicule
class Moto : public Vehicule {
      public:

      // Quand on construit une voiture...
      // il faut appeler le constructeur de la classe mère
      Moto() : Vehicule(2) {  // elle a 2 roues
            }
      void prix() const {                 // Affiche le prix du Vehicule
            cout << "Cette moto coûte " << prix_achat << "€" << endl;
            }
      };

int main()
{
    Vehicule *v(0);
    v = new Voiture(true);

    //On crée une Voiture et on met son adresse dans un pointeur de Vehicule
    v->motorisation();  // On affiche "Ce véhicule roule au diesel."

    delete v;           // Et on détruit l'objet voiture

    return 0;
}
```

Dans la méthode motorisation() de la classe Voiture, on a utilisé l'opérateur :: appelé opérateur de résolution de portée qui sert à déterminer quelle fonction (ou variable) utiliser quand il y a ambiguïté ou si il y a plusieurs possibilités.

Dans le corps principal du programme, un pointeur est défini sur l'objet Vehicule. Au moment du delete, il faut appeler le bon destructeur : celui de l'objet Voiture et non celui de l'objet Vehicule. Pour cela, il faut rendre le destructeur virtuel dans la classe Vehicule avec le mot clef virtual devant la déclaration de la méthode.

De même, lors de l'appel à la méthode motorisation(), il faut appeler la méthode de la classe Voiture et non celle de la classe Vehicule. Là aussi, il faut rendre la méthode virtuelle dans la classe Vehicule.

Remarque : la méthode prix() de la classe Vehicule ne possèdent aucun code dans son corps car cela n'a pas de sens dans notre exemple. Une fonction ne possédant qu'une déclaration, sans code implémenté est dite abstraite ou virtuelle pure. En C++, on utilise une syntaxe spéciale [= 0]. Les méthodes abstraites doivent être implémentées dans les classes dérivées. Une classe qui possède au moins une méthode abstraite devient une classe abstraite qui ne peut être instanciée.

Ce problème peut être contourné en déclarant un objet Voiture et non Vehicule :

```
int main()
{
    Voiture *v = new Voiture(true);

    v->motorisation();  // On affiche "Ce véhicule roule au diesel."
```

```
    delete v;           // Et on détruit l'objet voiture

    return 0;
}
```

Cependant, ce code perd la notion de poymorphisme qui permet à un même code d'être utilisé avec différents types pour des implémentations plus abstraites et générales.

10. Objet string

10.1. Créer et utiliser des objets string

La création d'un objet ressemble beaucoup à la création d'une variable classique comme int ou double :

```
#include <iostream>
#include <string> // Obligatoire pour pouvoir utiliser les objets string

using namespace std;


int main()
{
    string maChaine; //Création d'un objet 'maChaine' de type string

    return 0;
}
```

Remarque : pour commencer que, pour pouvoir utiliser des objets de type string dans le code, il est nécessaire d'inclure l'en-tête de la bibliothèque string. Initialiser la chaîne lors de la déclaration

Pour initialiser l'objet au moment de la déclaration, il y a plusieurs possibilités. La plus courante consiste à ouvrir des parenthèses :

```
int main()
{
    //Création d'un objet 'maChaine' de type string et initialisation
    string maChaine("Bonjour !");

    return 0;
}
```

On peut l'afficher comme n'importe quelle chaîne de caractères avec cout :

```
int main()
{
    string maChaine = "Bonjour !";

    //Affichage du string comme si c'était une chaîne de caractères
    cout << maChaine << endl;

    return 0;
}
```

Concaténation de chaînes :

```
int main()
```

```
{
    string chaine1("Bonjour !");
    string chaine2("Comment allez-vous ?");

    string chaine3 = chaine1 + " " + chaine2; // Concaténation

    cout << chaine3 << endl;

    return 0;
}
```

Comparaison de chaînes :

```
int main()
{
    string chaine1("texte");
    string chaine2("texte autre");

    if ( chaine1 == chaine2 ) // Faux
        cout << "Les chaines sont identiques." << endl;
    else
        cout << "Les chaines sont differentes." << endl;

    return 0;
}
```

10.2 Opérations sur les string

Le type string propose un nombre important d'autres fonctionnalités dont voici les principales méthodes :

size() permet de connaître la longueur de la chaîne actuellement stockée dans l'objet de type string. Cette méthode ne prend aucun paramètre et renvoie la longueur de la chaîne.

erase() supprime tout le contenu de la chaîne

c_str() renvoie un pointeur vers le tableau de char que contient l'objet de type string.

substr() permet d'extraire une partie de la chaîne stockée dans un string. Elle prend un paramètre obligatoire et un paramètre facultatif :

1. index pour indiquer à partir de quel caractère on doit couper (ce doit être un numéro de caractère).
2. num pour indiquer le nombre de caractères que l'on prend. La valeur rar défaut revient à prendre tous les caractères qui restent.

Exemple :

```
int main()
{
    string chaine("Bonjour !");

    cout << "Longueur de la chaine : " << chaine.size();

    cout << chaine.substr(3) << endl;          // affiche "jour !"
```

```
    return 0;
}
```

Remarque : Pour convertir une string en minuscule ou en majuscule, il faut appliquer la fonction tolower / toupper à chaque caractère, ce que l'on peut réaliser par exemple grâce à transform.

```
#include <cctype>    // pour tolower et toupper
#include <string>    // pour string
#include <iostream>  // pour cout
#include <algorithm> // pour transform

struct my_tolower
{
    char operator()(char c) const
    {
        return tolower(static_cast<unsigned char>(c));
    }
};

struct my_toupper
{
    char operator()(char c) const
    {
        return toupper(static_cast<unsigned char>(c));
    }
};

int main()
{
    string s("ABCDEF");

    transform(s.begin(), s.end(), s.begin(), my_tolower());
    cout << s; // affiche "abcdef"

    transform(s.begin(), s.end(), s.begin(), my_toupper());
    cout << s; // affiche "ABCDEF"

    return 0;
}
```

11. Programmation modulaire

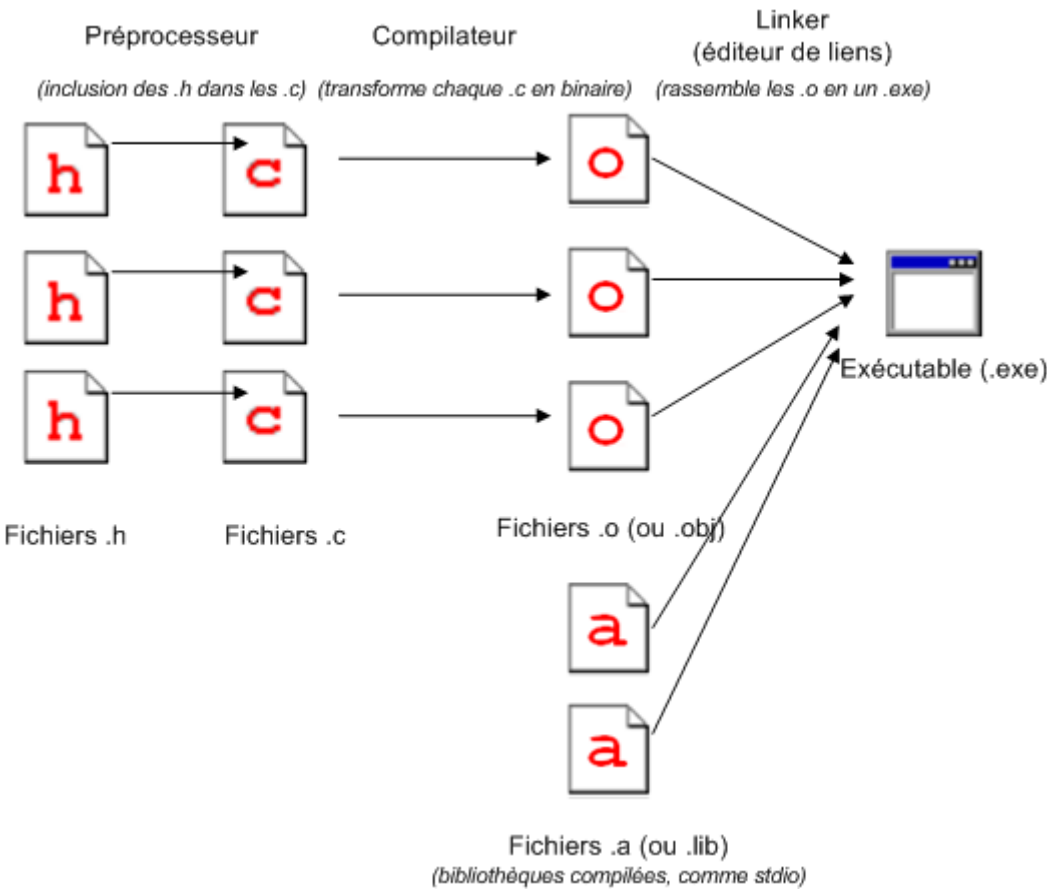
La programmation modulaire consiste à décomposer une grosse application en modules, groupes de fonctions, de méthodes et de traitement, pour pouvoir les développer indépendamment, et les réutiliser dans d'autres applications.

Le développement du code des modules peut être attribué à des (groupes de) personnes différentes, qui effectuent leurs tests unitaires indépendamment.

Cette méthode de regroupement permet de réaliser une encapsulation comparable par certains aspects à celle de la POO, et permet l'organisation du code source en unités de travail logiques. Les modules définissent également des espaces de noms utiles lors de leur utilisation.

Ce style de programmation facilite grandement la ré-utilisabilité et le partage du code, et est particulièrement utile pour la réalisation de bibliothèques.

La figure suivante décrit le processus de la génération d'un programme exécutable :



1. Préprocesseur : le préprocesseur est un programme qui démarre avant la compilation. Son rôle est d'exécuter les instructions spéciales qu'on lui a données dans des directives de préprocesseur, ces fameuses lignes qui commencent par un # comme par exemple la directive #include, qui permet d'inclure un fichier dans un autre. Il met à l'intérieur de chaque fichier .c ou .cpp le contenu des fichiers .h qu'on a demandé d'inclure.À ce moment-là de la compilation, le fichier .c ou .cpp est complet et contient tous les prototypes des fonctions.
2. Compilation : cette étape consiste à transformer les fichiers source en code binaire compréhensible par l'ordinateur. Le compilateur compile tous les fichiers source du projet.

   Le compilateur génère un fichier .o (ou .obj, ça dépend du compilateur) par fichier .c compilé. Ce sont des fichiers binaires temporaires. Si parmi les 10 fichiers .c de votre projet seul l'un d'eux a changé depuis la dernière compilation, le compilateur n'aura qu'à recompiler seulement ce fichier .c.
3. Édition de liens : le linker est un programme dont le rôle est d'assembler les fichiers binaires .o avec les bibliothèques compilées nécessaires (.a ou .lib selon le compilateur). Il les assemble en un seul gros fichier : l'exécutable final ! Cet exécutable a l'extension .exe sous Windows.

Le principe de la programmation modulaire repose sur la décomposition du programme en fichiers :

- .h : en-tête (header) destinés aux déclarations des constantes et des prototypes
- .c (ou .cpp) : fichiers sources des instructions à réaliser

Exemple :

```
// fichier : vehicule.h
#ifndef VEHICULE_H_INCLUDED
#define VEHICULE_H_INCLUDED

class Vehicule {
      protected:  // attributs uniquement accessibles aux classes filles

      int   prix_achat;
      int   nombre_de_roues;
      int   puissance_moteur;

      public:

      Vehicule(int roue) {
            nombre_de_roues = roue;
            }
      virtual ~Vehicule() {
            // même si le destructeur ne fait rien il faut le mettre
            }
      void puissance() const;              // Affiche la puissance
      virtual void motorisation() const;  // Affiche le type de motorisation
      virtual void prix() const = 0;       // Affiche le prix du Vehicule
      };
#endif // VEHICULE_H_INCLUDED

//---- fichier : vehicule.cpp
#include "vehicule.h"   // inclure la définition de la classe

void Vehicule::puissance() const {         // Affiche la puissance
      cout << "La puissance est de " << puissance_moteur << " ch." << endl;
      }
virtual void Vehicule::motorisation() const {   // Affiche la motorisation
      cout << "Ce véhicule roule à l'essence." << endl;
      }

// fichier : voiture.h
#ifndef VOITURE_H_INCLUDED
#define VOITURE_H_INCLUDED

#include "vehicule.h"   // inclure la définition de la classe mère

class Voiture : public Vehicule {   // les : indiquent l'héritage
      bool  est_un_diesel;
      int   nombre_de_portes;

      public:

      // Quand on construit une voiture...
      // il faut appeler le constructeur de la classe mère
      Voiture(bool diesel) : Vehicule(4) {      // elle a 4 roues
            est_un_diesel(diesel);  // on définit sa motorisation
            }
      void motorisation() const;          // Affiche le type de motorisation
      void prix() const;                  // Affiche le prix du Vehicule
      };
#endif // VOITURE_H_INCLUDED
```

```
//---- fichier : voiture.cpp
#include "voiture.h"    // inclure la définition de la classe

void Voiture::motorisation() const {      // Affiche le type de motorisation
      if ( est_un_diesel )
            cout << "Ce véhicule roule au diesel." << endl;
      else
            Vehicule::motorisation();     // appel à la classe mère
      }
void Voiture::prix() const {              // Affiche le prix du Vehicule
      cout << "Cette voiture coûte " << prix_achat << "€." << endl;
      }

// fichier : moto.h
#ifndef MOTO_H_INCLUDED
#define MOTO_H_INCLUDED

#include "vehicule.h"   // inclure la définition de la classe mère

class Moto : public Vehicule {
      public:

      // Quand on construit une voiture...
      // il faut appeler le constructeur de la classe mère
      Moto() : Vehicule(2) {  // elle a 2 roues
            }
      void prix() const;                  // Affiche le prix du Vehicule
      };
#endif // MOTO_H_INCLUDED

//---- fichier : moto.cpp
#include "moto.h" // inclure la définition de la classe

void Moto::prix() const {                 // Affiche le prix du Vehicule
      cout << "Cette moto coûte " << prix_achat << "€" << endl;
      }

/*
 ============================================================================
 Name        : gestion_voiture.cpp
 Author      :
 Version     :
 Description : gestion d'une flotte de voitures
 ============================================================================
 */

#include <iostream>     // Inclut la bibliothèque iostream (affichage de texte)
using namespace std;    // Indique quel espace de noms on va utiliser

#include "voiture.h"    // inclure la définition de la classe à utiliser

/==== fichier principal ====
int main()
{
    Voiture *v = new Voiture(true);
```

```
    v->motorisation();  // On affiche "Ce véhicule roule au diesel."
    delete v;           // Et on détruit l'objet voiture
    return 0;
}
```

Les lignes « `#ifndef …_H_INCLUDED` » sont là pour empêcher le compilateur d'inclure plusieurs fois ce fichier et de provoquer une erreur.